Programozás

2. bővített kiadás
írta Fábián Zoltán
Budapest, 2007 április

1 Bevezetés

Amikor elkezdtem programozással foglalkozni, még a Commodore 64-es gépek voltak divatban, és az amatőrök kedvenc programozási nyelve a BASIC volt. Hamarosan a nyelv minden csínjával-bínjával tisztában voltam, majd rávettettem magam a gép gépi kódú programozására is. Sokat programoztam. Mégis nagy gondom volt, hogy körülbelül 1000 sor hosszú programok írásakor a programjaim már nem sikerültek igazán, tele voltak hibával, megmagyarázhatatlanul viselkedtek, áttekinthetetlenekké váltak. Ekkor kezdtem el az egyetemet, az informatika szakot. Az első évek egyikében volt egy „Módszeres programozás" című tárgyunk. A kurzus végére rájöttem, hogy korábbi programjaim nem is működhettek hibátlanul, mert a kezemben nem volt semmi módszer, amitől hibátlanok lehettek volna.

Sokféle rendszer használata során rájöttem, hogy a programozás egyfajta gondolkodási forma, ugyanis a problémák rendszeresen különböző környezetekben ugyanazok, csak a megjelenési formájuk különbözik. Ez azt jelenti, hogy a gondolkozási formát kell alapvetően elsajátítani annak, aki meg akar tanulni programozni, utána már az egyes programozási nyelvek specialitásai könnyen mennek.

A jegyzet nem foglalkozik az egyes programozási nyelvek speciális kérdéseivel, hanem a programozást, a problémák megközelítésének módszerét tárgyalja. A programozás tanulásának sorrendjében következnek az egyes fejezetek.

Akik az Informatikai alapismeretek tantárgy középszintű vagy emelt szintű érettségijére készülnek, vagy az Informatika tantárgy emelt szintű érettségijére készülnek, azoknak javaslom az alábbi fejezetek áttanulmányozását:

- Alapfogalmak
- Algoritmusok
- Adatszerkezetek, adattípusok, adattárolás
- Egyszerű adatszerkezetek
- Összetett adatszerkezetek
- Elemi algoritmusok, programozási tételek

Akik a Számítástechnikai Programozó, Műszaki informatikai mérnökasszisztens, Rendszerinformatikus és hasonló OKJ-s képzés vizsgájára készülnek a fentieken kívül a többi fejezetet is sikeresen forgathatják.

Fábián Zoltán fz@szily.hu

Budapest, 2007. május 20.

2 Alapfogalmak

A programozás valós problémák számítógépes megoldása. Milyen tulajdonságai fontosak a számítógépnek? A számítógép

- általános célú (univerzális)
- automatikus vezérlésű
- elektronikus
- digitális.

Mi a különbség az ember és a számítógép között? Hogyan számol az ember?

- Olvassa az utasításokat
- Értelmezi
- Végrehajtja
- Rátér a következő utasításra, vagy arra, amit előírnak

Milyen utasítások vannak?

- Bemeneti/kimeneti (input/output)
- Adatkezelési
- Aritmetikai
- Döntési

A számolási tevékenységet végző ember számára magyar nyelven lehet olyan utasításokat adni, amit képes végrehajtani. A magyar nyelv kellően összetett ahhoz, hogy a bonyolultabb munkafolyamatokat is megértse az ember. A számítógép számára a vezérlőegység által értelmezhető és végrehajtható utasítások (parancsok) adhatók. Kezdetben az ember megtanulta ezeket az utasításokat (a számítógép nyelvét). A kommunikáció ezen a nyelven lassú, nehézkes, sok hibával járt. Feltesszük a kérdést, hogy miért nem tanul meg inkább a számítógép magyarul? A probléma összetett.

Valakinek meg kellene tanítania rá.

- A nyelv kétértelmű, a számítógép félreértheti
- Az élő nyelvben sok minden függ a szövegkörnyezettől
- Szemantika probléma (el kellene magyarázni a szavak jelentését)

Megoldási lehetőség, ha az ember is és a számítógép is megtanul egy közös nyelvet! Ez a nyelv kifejezi a szándékainkat, azaz mit szeretnénk kezdeni a számítógéppel. Mit is?

Szeretnénk megmondani neki, hogy milyen tevékenységet hajtson végre. A számítógép azt tudja megcsinálni, amit megmondunk neki.

Meg szeretnénk mondani, hogy hogyan hajtson végre dolgokat. Ha megmondjuk a hogyant, akkor a számítógép képes elvégezni a feladatot.

3 Algoritmusok

Az ember a fejében megbúvó gondolatokat tevékenységek sorozatára fordítja le. Ez cselekvéssorozat az **algoritmus**.

Definíció

*Az **algoritmus** egy cselekvési sorozat formális leírása.*

Az algoritmus elsősorban az embernek szóló formális leírás. Az algoritmusok leírásának jól definiált módszerei vannak.

Az algoritmusok fontos tulajdonságai:

- Végesség – véges idő alatt be kell fejeződnie
- Meghatározott – Minden egyes lépésének pontosnak kell lennie. Az élő nyelv nem feltétlenül alkalmas erre, ezért az algoritmus egy formális nyelv
- Bemenet/Input – Meg kell adni a bemeneti paramétereit, vagy ha ezek nincsenek, akkor ezt is jelölni kell
- Kimenet/Output – Meg kell adnia a kimeneti adatokat
- Elvégezhető – Olyannak kell lennie, amelyet egy ember is el tud végezni megfelelő sok papír és ceruza felhasználásával.
- Univerzális – Kellően általános ahhoz, hogy megfelelő peremfeltételek teljesülése esetén a feladatát elvégzi

Definíció

*Az algoritmus **vezérlési szerkezete** meghatározza, hogy a programban (algoritmusban) leírt utasításokat a program milyen sorrendben hajtsa végre.*

Definíció

*A **szekvencia** olyan program vezérlési szerkezet, amelyben az utasítások leírásának sorrendje meghatározza a végrehajtásuk sorrendjét is. Tipikusan az egymás alá írt utasítások leírási sorrendje a végrehajtási sorrend. Az egy utasításból álló program speciális szekvenciának fogható fel.*

Definíció

*A **szelekció (más néven elágazás)** olyan vezérlési szerkezet, amely egy logikai feltétel hatására a program végrehajtása két vagy több irányra szétbomlik és a feltétel eredményétől függően különböző szekvenciát hajt végre a program.*

Ha a feltétel eredménye igaz vagy hamis, akkor egy vagy két különböző szekvencia végrehajtása között dönt a program. Ilyenkor **kétirányú elágazásról** beszélünk.

Ha a feltétel eredményeinek összessége (eredményhalmaza) egymástól jól megkülönböztethető értékeket jelent, akkor **több irányú elágazásról** beszélünk. Ekkor a lehetséges eredményekhez hozzárendelünk szekvenciákat, amelyeket a program végrehajt, és ha a megjelölt eredmények egyike sem jön létre, akkor az alapértelmezett szekvenciát hajtjuk végre.

Definíció

*A **ciklus** olyan programszerkezet, amely egy szekvenciát ismétel.*

A ciklusnak a lényege az, hogy bizonyos utasítások sorozatát ismétli. A ciklus az alábbi részekből áll:

Ciklus fej – A ciklusfejben határozzuk meg azokat a feltételeket, amelyek teljesülése esetén a program végrehajtja a ciklus magot.

Ciklus mag – Az a szekvencia, amelyet a program ismételten végrehajt.

Ciklusok fajtái:

Elöltesztelő ciklus – a ciklusmag végrehajtását meghatározó feltétel a ciklusfejben van. A feltétel vizsgálata egyszer mindenképpen lezajlik. A ciklusfejbe a ciklus végrehajtásának a feltételét írjuk (a bennmaradás feltétele), vagyis ha a feltétel éppen igaz, akkor végrehajtjuk a ciklusmagot. Lehetséges olyan helyzet, hogy a program végrehajtása során a ciklus feltétele eleve hamis, ekkor a ciklus mag sohasem hajtódik végre.

Hátultesztelő ciklus – a ciklusfej nem tartalmaz feltételt, a ciklusmag mindenképpen lefut egyszer, és a ciklusmag lefutása után vizsgáljuk meg a feltétel teljesülését. Itt általában szintén a bennmaradás feltételét írjuk le.

Megszámlálásos ciklus – A ciklus fejben a ciklusmag ismétlésének számát határozzuk meg. Ez is elöltesztelő ciklus, tehát ha az ismétlések száma 0 lenne, akkor a ciklusmag nem fut le egyszer sem.

Definíció

*A **változó** a programban megadott névvel megkülönböztetett memóriaterület. Itt tárolhatunk adatokat a programban.*

A változók tulajdonságai:

a változó neve – erre speciális szabályok vannak (Általában alfanumerikus karakterek lehetnek plusz aláhúzás jel)

a változó típusa – azt jelenti, hogy milyen fajta adatot tárolhat a változó (szöveg, szám, dátum, stb...). Ha egy programozási nyelv típusos, akkor a változó deklarálásakor (létrehozásakor) megadott típusú értéket lehet csak tárolni a változóban (pl C, Pascal, Java, Basic). Ha a programozási nyelv enyhén típusos, akkor a változó deklarációja nem jelenti egyúttal a típusának is a meghatározását, sőt esetenként a típusa változhat is futás közben.

A változó értéktartománya – a változó által felvehető lehetséges értékek halmaza.

a változók értéke – a program futása során értéket adhatunk a változónak. A változókat a program futása elején inicializálni kell, azaz kezdőértéket kell nekik adni. A programozási nyelvek általában automatikusan az inicializálást elvégzik.

A változó címe – a memóriának az a helye, ahol futás közben az adott változó elhelyezkedik. A változó címének kezelését a programozó szinte sohasem maga végzi, hanem egy szimbolikus névvel, a változónévvel hivatkozik az adott helyre. A konkrét memóriacím vagy a program fordításakor, vagy futás közben dől el.

A változók olyan szimbólumok, amelyek használatával utasítjuk a programot, hogy futáskor biztosítsa az változó típusának megfelelő méretű helyet a memóriában. A program fordításakor az általunk használt szimbólumok lefordulnak olyan memóriacímekké, ahol a megfelelő típusú adatokat lehet tárolni. A program a memóriába való betöltődéskor ezeket a címeket kiegészít az operációs rendszer egy pillanatnyi alapmemóriacímmel és így tudja kezelni végül is a programokat.

Definíció

*A **konstans** a programban megadott névvel megkülönböztetett érték. A program futása során a konstansnak csak egyszer lehet értéket adni. Általában olyan adatokat tárolunk benne, amelyek a program teljes területén állandóak, és elérhetőnek kell lenniük.*

A program fordításakor a konstans helyére az adat bináris megfelelője kerül, vagyis az értéket „bedrótozzuk” a programba.

Definíció

***Az** eljárás **a program önálló, zárt egysége, amelyek a főprogramtól függetlenül definiált. Az eljárásokat egyedi névvel azonosítjuk a programon belül**.*

Az eljárás leírását nevezzük az eljárás definiálásának. Ha egy programban a definiált eljárást le akarjuk futtatni, akkor „meghívjuk" az eljárást, azaz nevével hivatkozunk rá.

A program végrehajtása ekkor átkerül az eljárás definiálásának helyére, majd amikor befejezte az eljárást, akkor a program végrehajtása visszakerül a hívási hely utáni utasításra.

Az eljárások a hívási helytől paramétereken keresztülkaphatnak értékeket. A paraméterek olyan változók vagy konstansok, amelyek a hívás helyén léteznek. Az eljárás belsejében ezekre az átadott értékekre hivatkozni lehet, azok értékeit lehet módosítani. A paraméterátadás módjától függően a módosított paraméterek az eljárás befejezése után érvényre jutnak a hívó programrészben vagy nem. A paraméterátadás módjáról később részletesen szólunk.

Definíció

*A **függvények** olyan programozási zárt egységek, amelyek minden tulajdonságukban megegyeznek az eljárásokkal, de a függvények rendelkeznek egy visszatérési értékkel, amelynek következtében a függvények kifejezésekben is részt vehetnek, ők megadott értékű és típusú értéket képviselnek.*

Definíció

***Paraméterátadásnak** hívjuk azt, amikor a hívó programrész értékeket ad át a meghívott eljárásnak vagy függvénynek. Ezt az értéket a hívott eljárás vagy függvény fel tudja használni, módosíthatja, stb.*

A paraméterátadásnak az alábbi formái léteznek:

Érték szerinti paraméterátadás: Ekkor a híváskor a hívási helyen lévő értékekről másolat készül, és ezzel a másolattal dolgozik a meghívott eljárás. Az adat átvételekor ugyanolyan típusú paramétert kell jelölnünk, mint az átadáskor, de az átadott és az átvett paraméterek neveinek nem kell megegyeznie. Az eljáráson belül bármilyen módosítás az eljárás befejezése után nem jut érvényre. Paraméterként megadott típusú változó, konstans szerepelhet.

Cím szerinti paraméterátadás: Ekkor a hívás helyén szereplő paramétereknek a memórián belüli helyét (címét) adja át a program az eljárásnak. Az adat átvételekor ugyanolyan típusú paramétert kell jelölnünk, mint az átadáskor, de az átadott és az átvett paraméterek neveinek nem kell megegyeznie. Ha az eljárás módosítja a paraméter értékét, akkor természetesen ez a módosítás az eljárás befejezése után is hatályban marad. Konstansokat cím szerint nem lehet átadni eljárásnak

Megjegyzés

> *A Pascal és a C más és más módon kezeli a paraméterátadást. A lényeg az, hogy a Pascalban a hívó és a hívott eljárásban a változók, adatok típusának és számának meg kell egyeznie, míg a C nyelvben a típusnak sem kell mindig megegyezni, és lehetőség van arra, hogy kevesebb adatot vegyünk át, mint amennyit átadna a hívó eljárás. Bár a rendszer bizonyos esetekben figyelmeztet a turpisságra, de a program lefordul és fut is.*

Egy futtatható EXE, COM vagy BAT fájl is tud átvenni az operációs rendszertől adatokat a PC-ken. Ekkor mindig érték szerinti a paraméterátadás.

3.1 Algoritmuselíró módszerek, nyelvek

Háromféle algoritmusleíró módszer terjedt el Magyarországon. Az első leírási módszert még a második generációs gépek idején találták ki, ma is sokan használják. **Ez a folyamatábra**. A folyamatábrát elsősorban az egyszerűbb rendszerek futtatási folyamatainak ábrázolására találták ki. Előnye az, hogy vizuálisan megmutatja a programok futásának menetét. Az oktatásban, az egyszerűbb folyamatok ábrázolásában nagy szerepe van.

Ahogy a programozás egyre inkább tudománnyá vált a folyamatábra már nem felelt meg az egzakt és absztrakt ábrázolásnak, a programozók kitalálták a **struktogram**-ot. A struktogram precízen ábrázolja a folyamatokat, ugyanakkor olvasása esetenként nehézkes, a valódi programozástól kicsit elrugaszkodott. Formalizmusa precíz, de nehezen értékelhető vizuális ábrákat eredményez.

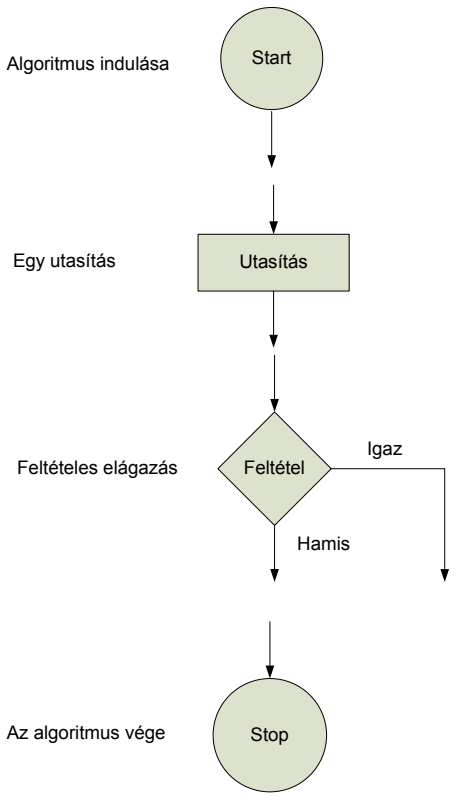
A programozás oktatásában vezették be a **mondatszerű leírást** vagy **pszeudo-kódot**. A mondatszerű leírás nagy tért hódított, mivel nagyon jól illeszkedik a Pascal programozási nyelv szintaktikájához, kellően szabadon definiálhatók az egyes programozási elemek, ugyanakkor nem vizuális. A mondatszerű leírás használata esetén a kódolás esetenként a magyar nyelvről az angol nyelvre való fordításra egyszerűsödik.

A felsorolt három algoritmus-leíró módszer egyenrangú, mind a három alkalmas arra, hogy kisebb-nagyobb rendszerek algoritmusait megfogalmazzuk segítségükkel.

Itt nem tárgyaljuk, de a bonyolult rendszerek leírásához a fenti módszerek egyike sem elegendő ma már, ezért az objektum-orientált programok leírásához az UML (=Universal Modelling Language) az egyik legígéretesebb leírónyelv, amit a jegyzet későbbi részében részletesen tárgyalunk

3.1.1 Folyamatábra

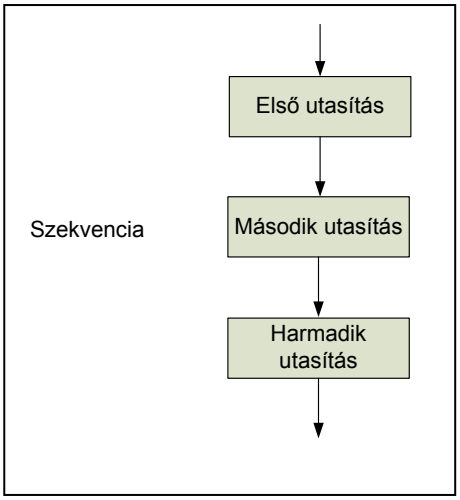
A folyamatábrák használatakor grafikai jelekkel jelöljük az egyes programozási egységeket. A grafikai jeleket nyilakkal kötjük össze, a program futásának megfelelő módon. Általában egy program vagy egy eljárás fentről lefelé vagy balról jobbra hajtódik végre, legalábbis a folyamatábra rajzolásánál erre kell törekedni. A szokásos feldolgozási lépéseket az alábbi módon szimbólumokkal jelöljük:



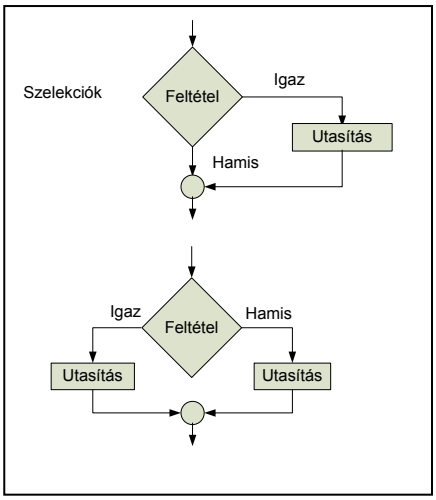
A folyamatábrában egy eljárás ábrázolása úgy jelenik meg, hogy a téglalapba beírjuk az eljárás nevét, majd egy külön ábrában lerajzoljuk az eljárás folyamatábráját.

Gondot jelent egy függvény ábrázolása, ugyanis a folyamatábra esetén nincsen megoldás a függvény visszatérési értékének jelzésére. A három alapvető programvezérlési szerkezet jelzése itt található meg.

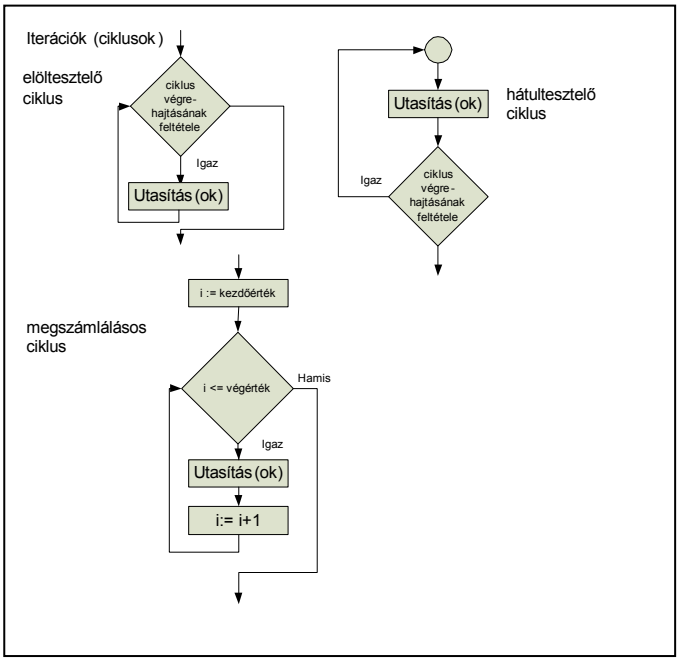
Szekvencia



Szelekció (feltételes elágazás)

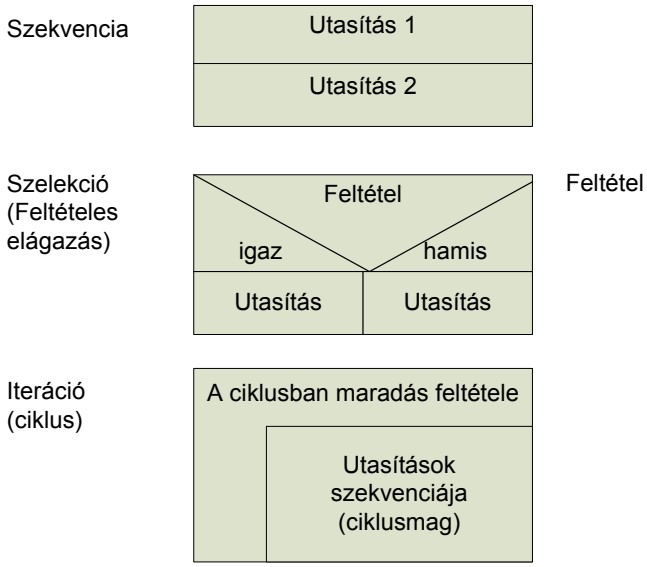


Iteráció (Ciklus)

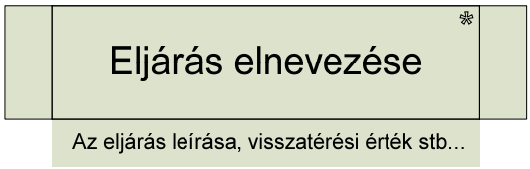


3.1.2 Struktogram

A struktogram algoritmus leíró módszer elsősorban a professzionális programozás oktatása során használandó. Ennek a leíró módszernek az előnye rendkívül egzakt módjában és tömörségében van. Talán kezdők számára nem annyira áttekinthető, mint a folyamatábra, de annak hibáit kiküszöböli. A lényeg az, hogy egy programot egymásba ágyazott dobozokkal szemléltetek és minden doboz egyúttal további programszerkezetek tárolója is.



Itt is problematikus az eljárások és függvények kibontása. Erre lehet az alábbi jelölést használni.



Ezután a megfelelő névvel alkotunk egy újabb struktogramot, és ott bontjuk ki az eljárást.

3.1.3 Mondatszerű leírás

A mondatszerű leírásnak akkor van értelme, ha a programozó valamennyire tisztában van egy számítógépes program működésével, érti az utasítások egymásutániságát, az értékadás, az elágazások, a ciklusok fogalmát. Érti azt, hogy mit jelent egy eljárás hívása és mit jelent egy függvény hívása. Mint látjuk majd, az algoritmus-leíró nyelv egyfajta magyar nyelvű programozási nyelv.

A magyarországi programozás-oktatásban középszinten és egyetemi szinten is több területeken preferálják ezt a módszert, ráadásul az érettségi és az OKJ-s vizsga is tartalmazhat ilyen módon megfogalmazott feladatokat, ezért ezt egy kicsit részletesebben tárgyaljuk.

A mondatszerű leírás a következő szintaktikai (=helyesírási) szabályokat tartalmazza:

1. Egy sorba egy utasítást írunk

2. Változó nevének, típusának és értelmezési tartományának megadása

```
Byte i Egész szám
Tomb A[N] szövegekből álló tömb
```

Egy változó típusát nem kell mindig külön deklarálnunk, ha az értékadáskor kiderül a típus.

```
j := 23
```

3. Beviteli és kiviteli utasítások

```
Ki:    Kiíró utasítás
Be:    Adatbeviteli utasítás
```

4. Program struktúrák

Program jelzése

```
Program Név
  ……
Program vége
```

Eljárás jelzése

```
Eljárás Név(paraméterlista)
  ……
Eljárás vége
```

Függvény jelzése

```
Függvény Név(paraméterlista)
  ……
  Nev := visszatérési érték
Függvény vége
```

Az eljárásokat és függvényeket az alábbi módon hívhatjuk meg a programból:

```
Név( paraméterlista )
```

vagy

```
Érték := Név( paraméterlista )
```

5. Egy változó értékadása

```
B := érték
```

Pl.

```
B := 65
```

A két oldalon azonos típusú értékek vannak

6. Vezérlési szerkezetek

Feltételes elágazás

```
Ha feltétel igaz akkor Utasítás
```

vagy

```
Ha feltétel igaz akkor
  Utasítások …
Elágazás vége
```

vagy

```
Ha feltétel igaz akkor
  Utasítások …
Különben
  Utasítások …
Elágazás vége
```

Ciklusok

Definíció: Megszámlálásos ciklus
Olyan ciklus

```
Ciklus cv :=kezdőértéktől végértékig lépésközzel
  ……
Ciklus vége
```

vagy

```
Ciklus amíg feltétel igaz
  ……
Ciklus vége
```

vagy

```
Ciklus
  ……
amíg feltétel
Ciklus vége
```

3.2 Program vagy algoritmus specifikáció

Amikor egy programot vagy algoritmus készítünk meg kell határozni, hogy milyen típusú és értéktartományú adatokat kap a program és milyen kimeneti adatokat várunk a programtól.

Definíció: Programspecifikáció (algoritmus specifikáció)
A program bemeneti és kimeneti adatainak a meghatározása. Meghatározzuk a program bemenő és kimenő adatainak a

- Nevét,
- Értelmét
- Típusát,
- Értéktartományát

Meg kell határoznunk, hogy a bemenő adatok ismeretében milyen kimenő adatokat tekintünk helyesnek és melyeket helytelennek, azaz meg kell határoznunk az összefüggést a bemenő és kimenő adatok között.

3.3 Algoritmusok dokumentálása

Bármelyik algoritmusleíró módszert is használjuk, fontos a program részletes dokumentálása. Az absztrakt módszerek használata esetén (folyamatábra, struktogram) ez fokozottabban érvényes.

Az eljáráshívások helyén le kell írni a nem magától értetődő eljárások szerepét, az átadott paramétereket és a visszakapott értékeket.

Az eljárás kifejtésekor le kell leírni az **eljárás fejlécénél**, az átvett értékek és visszaadott értékeket, név és típus szerint.

Röviden le kell írni a függvény vagy eljárás szerepét.

A programhoz vagy algoritmushoz akkor kell megjegyzéseket fűzni, ha a kód nem teljesen triviális az átlagos fejlesztő számára. Figyelembe kell venni, hogy a programozók egy-két hónap után elfelejtik a saját kódjukat is, tehát inkább több megjegyzést, mint kevesebbet célszerű használni.

Az algoritmusok leírásánál mindig törekedjünk a világos áttekinthető leírásra, inkább több helyet hagyjunk egy-egy programrésznek, mint kevesebbet.

Használjuk a bekezdéses írásmódot. Ez azt jelenti, hogy az elágazások és ciklusok belseje kezdődjön a médián (papíron, a szövegszerkesztőben) beljebb, mint az elágazás vagy ciklus fejléce.

Ugyanígy az eljárások eljárás magja is kezdődjön beljebb az eljárás fejlécénél.

4 Adatszerkezetek, adattípusok, adattárolás

Az informatikában az adatokat nem fizikai megjelenési formájukban tároljuk, hanem kódolva. **Kódoláson** azt értjük, amikor egy adatsorozatot, jelek sorozatával helyettesítjük, és a jelsorozatot tároljuk, továbbítjuk. A kódolás célja sokféle lehet. Az adatok érthetőbb vagy egyértelmű megjelenítése, tömörebb tárolása, gyorsabb átvitele, stb. A kódolandó adat félesége meghatározza azt, hogy hányféle jellel lehet kódolni az adatot. Gyakori eset, hogy egy már kódolt jelsorozatot továbbkódolunk, és így többszörös kódolás jön létre.

Ha az eredeti adatokat vissza akarjuk nyerni, akkor a kódolás fordítottjának, a **dekódolásnak** kell lezajlania. Ha egy adatsort többszörösen kódolunk, akkor a dekódolások sorozata a kódolás sorozatával fordított sorrendben zajlik le.

A programozási nyelvek az adatokat bizonyos keretek között tárolják.

A számítógépek fizikai felépítéséből következően az adatokat bitek sorozataként tárolják a memóriában. Tekintettel arra, hogy egy bit csak kétféle adatot tartalmazhat, ezért a biteket nagyobb egységekbe szervezik és a processzorok kényelmes módot adnak a nagyobb egységekben történő adatkezeléshez.

Bár az adatok tárolásának módja programnyelvenként, sőt a programnyelv implementációiként is változhat, ugyanakkor vannak olyan alapvető adatszerkezetek, amelyek a gépek tulajdonságai miatt azonosak.

A számítógépek az adatokat **byte**, **word** (2 byte = szó) vagy **duplaszó** (4 byte) hosszú egységekben kezelik. A processzorok típusától és a memóriák szervezésétől függően a háromféle adatkezelés sebessége más és más. Erre példa, hogy a nyolc bites processzorok a byte hosszú adatokat kezelték a leggyorsabban, a 80286-os processzorok a szó hosszúságú adatokat, míg a 386-os és annál fejlettebb Intel processzorok a duplaszó hosszú adatokat kezelik a leggyorsabban. Ha optimális sebességű programot akarunk írni, ekkor az adattípusok meghatározásánál erre is figyelni kell.

Amikor egy programozási nyelv adatainak tárolását vizsgáljuk, akkor feltételezzük, hogy van egy kellően nagy memóriánk, amely byte nagyságú egységekből áll. A byte-ok sorfolytonosan helyezkednek el a memóriában.

Az adatok tárolása során a számítógépet nem érdekli egy adat értelme, jelentése. A programozónak kell módot találnia arra, hogy az adatot jelentésének megfelelően kezelje. A számítógépes programok azonban nyilvántarthatják bizonyos formai módszerekkel az adatok egyéb paramétereit.

Az adatok jellemzői:

Minden adat rendelkezik **típussal**

Az adattípus meghatározza az adat lehetséges **értékét**. Az adattípus lehetséges értékeinek halmaza az **értéktartomány**.

Az adat által a háttértáron vagy a memóriában elfoglalt hely nagysága az adat **mérete**. Egyes adattípusok mérete eleve adott, más adattípusok mérete a definíció során alakul ki a rész adattípusok méreteiből következően, míg vannak olyan adattípusok, amelyeknek mérete a program futása közben dől el

Fontos az adat elhelyezkedése a memóriában. Ez az adat **címe**. A programok az adat címének ismeretében képesek hozzáférni az adat értékéhez, azt a megadott helyről kiolvassák és szükség esetén módosítják, illetve visszaírják a háttértárakra.

Az adat típusa egyértelműen meghatározza az adattal végezhető műveleteket (pl. numerikus adatokkal matematikai műveleteket lehet végezni, de szöveggel nem ...).

Egyes adattípusok mérete eleve adott, más adattípusok mérete a definíció során alakul ki a rész adattípusok méreteiből következően, míg vannak olyan adattípusok, amelyeknek mérete a program futása közben dől el.

Alap adattípusok – a rendszerekben előre definiált típusok. A rendszerekben általában van lehetőség az alap adattípusok és a korábban definiált típusok felhasználásával új típusok definiálására is.

Bővített adattípusok – A rendszerekben meglévő alap adattípusok felhasználásával létrejövő új típusok, amelyek az eredeti alaptípus valamilyen tulajdonságának a módosításával jönnek létre. Ilyen módosítás lehet pl. az értéktartomány szűkítése, bővítése.

Korábban már definiáltuk a **konstans** és a **változó** fogalmát. Amikor deklarálunk egy változót, vagy egy konstanst létrehozunk az ún. erősen típusos nyelvek esetén – ilyen a C, C++, Pascal, Visual Basic – a változónak egyúttal megadjuk a típusát is. Más nyelvek esetén a változó típusa futás közben, az első értékadáskor dől el. Vannak olyan nyelvek, ahol a változó az értéktől függetlenül változtathatja adattípusát – pl. PHP.

4.1 Az adattípusok osztályozása megszámlálhatóság szerint

A halmazelméletből ismert fogalom a megszámlálhatóság. Amikor egy halmaz elemeit hozzá tudjuk rendelni a pozitív egész számok halmazához, akkor mondjuk, hogy a halmaz **megszámlálható számosságú**. Ez a gyakorlatban azt jelenti, hogy egyesével végig tudok valamilyen módon lépkedni a halmaz elemein. Ilyen halmaz például az egész számok halmaza, a betűk halmaza, stb. Az ilyen adattípusokat szokás **felsorolási** adattípusoknak is hívni.

A valós számok halmaza azonban nem megszámlálható, mivel mindig tudok két olyan számot mondani, amelyek között végtelen számú valós szám található, és amelyek nem bejárhatók. Ennek a **halmaznak a számossága nem megszámlálható**. Ilyen adattípus a valós számok vagy lebegőpontos számok halmaza. Ezen a ponton meg kell állni, ugyanis beleütközünk az abba a ténybe, hogy a valós számítógépek fizikailag korlátosak minden szempontból, azaz végtelen számú értéket véges helyen nem tudunk tárolni! Kompromisszumot kell kötnünk, és azt mondjuk, hogy azok az adattípusok nem megszámlálható számosságúak, amelyek részére nem áll rendelkezésre megfelelő kapacitás. Persze ez egy kicsit sántít, de ez van.

4.2 Az adattípusok osztályozása bonyolultság szerint

Definíció

*Az **egyszerű adattípusokat** a rendszer egységként kezeli és nem bontható szét összetevőkre.*

Definíció

***Összetett adattípusok** más adattípusok valamiféle kombinációjaként jönnek létre. A rendszer nem feltétlenül tudja egyben kezelni az összetett adattípust, ebben az esetben a programozónak kell gondoskodni a feldolgozásról. Az összetett adattípusnak vannak önállóan is feldolgozható rész adatai.*

Az összetett adattípusokból is készíthetünk további összetételeket, így meglehetősen bonyolult adatstruktúrákat lehet kialakítani.

Az adattípusok helyett szokás az **„adatszerkezet”** kifejezést is használni.

4.3 Egyszerű adattípusok

Az egyszerű adatszerkezeteket minden számítógépes rendszer ismeri. Ilyenek a numerikus, karakter, logikai típusok. Az egyszerű adattípusokat a gépek processzor szinten, azaz gépi kódú utasításokkal tudják kezelni, ezért használatuk a lehető leggyorsabb programokat eredményezi.

4.3.1 Numerikus típusok

Tetszőleges alapú számok ábrázolása

A számokkal való műveletek során hozzászoktunk a 10-es alapú számrendszerhez, de a számítógépeknek mindegy, hogy mi a használt számrendszer alapja (elvileg). Technikailag célszerű olyat választani, amely illeszkedik a számítógépek belső lelkivilágához, ezért a legjobban elterjedt rendszerek a kettes (bináris), 16-os (hexadecimális) és a 8-as (oktális) számrendszer. A **számrendszer alapszáma** egyúttal **megadja a** szám leírásánál **használható jelek számát** is.

4.3.1.1 Kettes számrendszerbeli számok ábrázolása

A kettes számrendszer esetén a számokat 0 és 1 jelek sorozatával jelöljük. Annyi jelet írunk le, hogy a jelek száma kettő hatványaival legyen egyenlő (1, 2, 3, 8, 16, 32, 63 jel), Egy jelet egy bit-nek hívunk (**Bi**nary Digi**t**). Ennek alapján beszélhetünk 4, 8, 16 stb... bites számokról.

Egy szám átalakítása 10-esből kettesbe.
Elosztjuk a 10-es számrendszerbeli számot 2-vel, a **maradék** lesz a jobbról számított legutolsó bit bit, majd a hányadost osztjuk 2-vel és a legszélső lesz az utolsó előtti bit, stb...)

Az így kapott bitsorozatot kiegészítjük annyi nullával, hogy megfeleljen a kívánt tárolási méretnek. (8 bit, 16 bit, 32 bit, stb...)

Egy szám átalakítása kettes számrendszerből 10-esbe
A jobbról számított első bit jelenti a kettő 0. hatványát, a következő kettő 1. első hatványát, stb... Ha 1 van a megfelelő helyiértéken, akkor az eredményhez hozzáadom a megfelelő 2 hatványt.

00010101 = 0*128+0*64+0*32+1*16+0*8+1*4+0*2+1*1 =23

0 és 1 közötti 10-es számrendszerbeli számok bináris ábrázolása.
A számot elosztom 2-vel. A hányados lesz a kettedes pont utáni első jegy. A maradékot elosztom 2 következő hatványával, és így folytatom, amíg a megfelelő tárolási méretet el nem érem.

Osztandó /osztó	Hányados	Maradék
0,7 / 0,5	=> 1	0,2
0,2 / 0.25	=> 0	0,2
0,2 / 0,125	=> 1	0,075
0,075 / 0,0625	=> 1	0,0125
0,0125 / 0,03125	=> 0	0,0125
0,0125 / 0,015625	=> 0	0,0125
0,0125 / 0,0078125	=> 1	0,0046875
...		

Az így kapott szám: 0,10110001

Látható, hogy az eredmény nem pontos, mivel az utolsó osztás után is van maradék! Az utolsó bit értékét kerekíteni kell. Általában igaz, hogy az egyik számrendszerben megadott véges szám nem feltétlenül lesz végesek más számrendszerben.

4.3.1.2 Kettes komplemens számábrázolás

Az tetszőleges méretű előjeles egész számok ábrázolásának elve. A számot 2^n db bittel írjuk le, ahol n=8, 16, 32, stb. A legmagasabb helyiértékű bit az úgynevezett előjelbit. Az előjelbit 0 értéke jelenti azt, hogy a szám pozitív vagy nulla, az előjelbit 1-es értéke jelenti azt, hogy a szám negatív.

Hogyan állapíthatjuk meg egy 10-es számrendszerbeli számból a kettes komplemens alakját?

Vesszük a szám abszolút értékét. Ha a szám pozitív, akkor ez nem módosít semmit, ha negatív, akkor elhagyjuk az előjelet.

Ennek a számnak kiszámítjuk a bináris értékét. (korábban láttuk)

Az így kapott bitsorozatot kiegészítjük annyi nullával, hogy megfeleljen a kívánt tárolási méretnek. (8 bit, 16 bit, 32 bit, stb...)

Ha az eredeti szám pozitív volt, akkor végeztünk.

Ha az eredeti szám negatív volt, akkor a kapott bináris számban minden bit értékét felcserélem 0-ról 1-re és 1-ről 0-ra. Ekkor kapom az **egyes komplemens értéket.**

A végén hozzáadunk 1-et.

Pl.

Eredeti szám	22	=>	00010100
Egyes komplemens:		=>	11101011 + 1
Kettes komplemens:	-22	=>	11101100

4.3.1.3 Oktális számok ábrázolása (8-as számrendszer)

Az oktális számoknál a használható jelek 0,1,2,3,4,5,6,7 Az oktális számokat mindig hármasával csoportosítva írjuk le. A 10-es számrendszerből az alábbiak szerint számoljuk át a számot 8-asba. Az eredeti számot elosztjuk 8-cal, és a maradék lesz a legkisebb helyiértékű szám. A hányadost osztjuk megint 8-cal és a maradék lesz a következő jegy, stb...

196 / 8 => 24, maradék 4
24 / 8 => 3, maradék 0
3/ 8 => 0, maradék 3

Az eredmény: 304

A szokásos jelölése az oktális számoknak az, hogy balról kiegészítjük egy 0-val a számot, azaz 0304.

Visszafelé hasonlóképpen járhatunk el, mint a bináris ábrázolásnál.

Bináris számokat átalakítani oktálissá egyszerű, mivel 3 bit alkot egy oktális számjegyet, azaz:

0304 => 000 011 000 100, ami persze csak 8 bites lesz, azaz 11000100.

4.3.1.4 Hexadecimális számok ábrázolása (16-os számrendszer)

A hexadecimális számoknál a használható jelek 0,1,2,3,4,5,6,7 ,8,9,A,B,C,D,E,F. A hexadecimális számokat mindig négyesével csoportosítva írjuk le. A 10-es számrendszerből az alábbiak szerint számoljuk át a számot 16-osba. Az eredeti számot elosztjuk 16-tal, és a maradék lesz a legkisebb helyiértékű szám. A hányadost osztjuk megint 16-tal és a maradék lesz a következő jegy, stb.

196 / 16 => 12, maradék 4 => 4
12 / 16 => 0, maradék 12 => C
Az eredmény: C4

A szokásos jelölései a hexadecimális számoknak: 0xC4, #C4, $C4

Visszafelé hasonlóan járunk el, mint a bináris számoknál.

Bináris számokat hexadecimálissá és vissza konvertálni egyszerű, mivel 4 bit alkot egy hexadecimális számjegyet:

192 => 0xC4 => 0110 100

4.3.1.5 Tetszőleges alapú számrendszerek (nem törzsanyag érettségire)

Az x alapú számrendszer esetén a használt jelek 0-tól (x-1) –ig terjednek.

Ha a számrendszer alapja x, akkor 10-esből erre az alapra úgy konvertálunk, hogy elosztjuk az eredeti számértéket az alappal és a maradék lesz a számrendszerbeli szám utolsó jegye. A hányadost osztjuk megint és így tovább.
Visszakonvertálni úgy kell:

Ha van egy x alapú számom: $a_1,a_2...a_n$, jelsorozat az alábbi polinom értékét jelöli:

$a_1 *X^{n-1} + a_2 *X^{n-2}... + a_n *X^0$

4.3.1.6 Byte

A mérete 1 byte, értéke 0..255 közötti egész szám lehet (2^8 érték)

Ennek egy bővített adattípusa a rövid egész vagy shortint, vagy short, amelynek az értékei

–128..,0, ..+127 közötti egész értékek lehetnek.

A szám előjelét a legmagasabb helyiértékű bit előjele jelöli. Ha a bit értéke egy, akkor negatív számról vagy a nulláról van szó. 8 bites rendszerekben a leggyorsabb adatkezelést biztosítják

4.3.1.7 Word - Szó

Mérete 2 byte, értéke 0..65535 közötti egész szám lehet. (2^{16} db érték)

4.3.1.8 Integer - Egész

Mérete két byte, értéke –32768..,0,..+32767 közötti egész értéket veheti fel. (2^{16} érték)

A szám előjelét a legmagasabb helyiértékű bit előjele jelöli. Ha a bit értéke egy, akkor negatív számról vagy nulláról van szó. 16 bites rendszerekben a leggyorsabb adatkezelést biztosítják az Integer és Word típusok.

Lehetséges műveletek:

Összeadás	+	A + B
Kivonás	-	A – B
Egész típusú osztás	Div vagy /	A div B vagy A / B
Egész típusok osztásakor maradéka	Mod vagy %	A Mod B vagy A % B
Eggyel való növelés, csökkentés	++, --	A++, B--

Összehasonlításokban

< kisebb, mint, > nagyobb mint, <= kisebb egyenlő mint, >= nagyobb egyenlő mint,

== egyenlő, <> vagy # nem egyenlő relációk lehetnek.

A fenti típusok bővítései a longint (Pascal), long int, unsigned long. Ezek 4 byte hosszú, adatok, ugyanazokkal a feltételekkel, amelyek a fentiekben tárgyaltunk. (2^{32} érték)

4.3.1.9 Lebegőpontos számábrázolás

A 10-es számrendszerben definiált számok ábrázolása összetett feladat, ráadásul rendszerenként kissé eltérő lehet. Külön problémát jelent a 0 ábrázolása is és a negatív számok ábrázolása. Elöljáróban elmondom, hogy a lebegőpontos számokat gyakran 4 bájton, azaz 32 biten ábrázolják, amelyből 1 bit az előjelbit, 23 a mantissza és 8 bit a karakterisztika (kitevő). Az előjelbit értéke 1-ha a szám negatív, 0 ha a szám pozitív.

Egy tízes számrendszerbeli lebegőpontos szám ábrázolásához az alábbi folyamaton kell végigmenni:

átalakítjuk a számot kettes számrendszerbelivé, külön az egészrészt és külön a tört részt: pl. -486,17 => 111100110.001010111000010

Normalizáljuk a számot (A normalizálásnál addig toljuk el a bináris pontot, amíg a bináris ponttól balra nem kerül az első 1-es bit): 1.11100110001010111000010 x 2^8

Innen külön kezeljük a mantisszát és a karakterisztikát.

A mantissza legfelső bitje mindig egyes, hiszen a normalizálásnál így toltuk el a bináris pontot. Ezt a bitet elhagyjuk (!) és a mantissza elé írjuk az előjelbitet, ami a példánkban 1-es. 11110011 00010101 11000010 lesz a mantissza

A karakterisztikánál trükközünk egyet, a kitevő értékéhez hozzáadunk 128-at. Ennek az indoka az, hogy ezáltal – kitevő is pozitív számmá változna, tehát a kitevő 8 => 8+127 =135 => 10000111

A karakterisztikát az Intel processzoros gépeken mindig a mantissza előtt tárolják a memóriában, tehát a teljes szám így fog kinézni: 10000111 **1**1110011 00010101 11000010

Lehetséges műveletek

Összeadás	+	A + B
Kivonás	-	A – B
Osztás	/	A / B
Matematikai függvények.	sin, cos, tg, atg exp, log, …	
Kerekítés, törtrészképzés	Round, trunc	

Összehasonlításokban

< kisebb, mint, > nagyobb mint, <= kisebb egyenlő mint, >= nagyobb egyenlő mint,

== egyenlő, <> vagy # nem egyenlő relációk lehetnek.

A real bővítésének felelnek meg a double 8 byte, extended, long double 10 bytet ípusok.

Általában az összeadás, kivonás, szorzás, osztás esetén az értékek lehetnek kevert típusúak is, azaz egész típus és lebegőpontos is, de az eredmény mindig olyan típusú lesz, mint a legnagyobb helyet elfoglaló érték vagy lebegőpontos. Az olyan függvények, amelyek lebegőpontos értéket várnak, nem elégednek meg egész típusú paraméterekkel és fordítva.

4.3.2 Karakter típus, kódlapok, karakterkódolás

Definíció

*A **karakter** egy írásjelet tároló adattípus.*

A karakteres adattípus egy karaktere a 8 bites karakterkezelésnél 1 byte helyet foglal, azaz 256 különböző értéket vehet fel. Valójában a karakterek tárolásakor a számítógép egy byte információt tárol, de hogy az milyen karakternek felel meg – milyen betűnek – az a kódolási rendszerektől függ.

A PC-k világában az **ASCII** (American Code for Information Interchange) karakterkódolási rendszer terjedt el. Ez a kódrendszer 7 bites, azaz a kódrendszernek csak az első 127 karaktere definiált.

A 0 – 31 kódú karakterek nem látható karakterek. A képernyőn mozgó kurzort – alfanumerikus képernyőn – a nyomtató fejét és egyéb olyan jeleket definiáltak, amelyeknek nincsen látható formájuk, de azért fontosak. Néhány példát csak: 0 – nem látható, 1- ☺, 2 - ☻, 10 – soremelés, 13 – kocsi vissza (Enter), 27 – Esc, …

A 32 – Szóköz. 33-47-ig különböző írásjelek vannak, 48 –tól 58-ig a 0,1,2,…9 számjegyek, 58-tól 63-ig írásjelek, 64 - @ jel (kukac).

65-tól 90-ig az angol ABC nagybetűi, A, B, C, D,….Z-ig. 91-től 96-ig írásjelek, majd 97-től –122-ig angol ABC kisbetűi, a, b, c, d,….z-ig. 123-tól 127-ig megint írásjelek.

Sok olyan nyelv van, amely ezen kívül még más írásjeleket is használ. Ezeket eredetileg a felső 128 jel közé tették volna be, de ez nem volt elegendő. Ennek a problémának a megoldására találták ki, az un. kódlapokat.

A **kódlapok** az ASCII karakterkészlet egyfajta kiterjesztései. A különböző kódlapok esetén az alsó 128 karakter megegyezik, de a felső 128 karakter az illető kódlapban érvényes jeleket jelent. A korrekt megjelenítés érdekében a futtató hardvernek, az operációs rendszernek, a nyomtatónak és minden egyéb eszköznek képesnek kell a különböző kódlapok használatára, csak úgy lehet korrekt megjelenítést elérni.

A probléma véglegesnek tűnő megoldása a 16 bites **Unicode** kódkészlet használata. Ekkor egy karakter két byte helyet foglal el a memóriában és ennek megfelelően 65535 féle jelet lehet megjeleníteni. Ez már elegendő a kínai és a japán valamint az egyéb nem európai nyelvek jeleinek a megjelenítésére is. Hátránya, hogy a karakterek több helyet foglalnak el. A mai operációs rendszerek alapvetően alkalmasak ennek a kódrendszernek a használatára.

4.3.2.1 Szövegek tárolása – sztring adattípus

Ha a számítógépek memóriájában a karaktereket egymás után tároljuk le, és valamilyen módon megadjuk, hogy hol van a karakterlánc vége, akkor egy új adattípust, a **karakterláncot** vagy közkeletű nevén **sztringet (string)** kapunk. Kétféle karakterlánc kezelési módszer terjedt el.

Az adott hosszúságú sztringek kezelése – A Pascalban a karakterek maximálisan 254 byte hosszúak lehetnek, mivel a 0. Helyen álló karakter kódja adja meg a sztring hosszát. Ez a karakter egyébként sohasem jelenik meg a képernyőn, de a program ennek alapján tudja megállapítani a sztring végét. Ennek az a hátránya, hogy nem lehet tetszőlegesen hosszú a karakter. A Pascalban előre meg kell mondani, hogy a karakter milyen hosszú lesz.

0 végű sztringek kezelése – A C nyelvben és annak minden folyományában a 0 végű karakterkezelés terjedt el, ami azt jelenti, hogy a sztring addig tart, amíg nem nulla karakterek jönnek egymás után. Ennek előnye, hogy elvileg tetszőleges hosszúságú, a gyakorlatban maximum 65535 karakter hosszú stringeket lehet kezelni. A hátránya, hogy azokban az esetekben, amikor szükséges tudni a sztring hosszát egy rutinnak végig kell szaladni a sztring elejétől a 0 kódú karakterig és meg kell számolnia a karaktereket – ez kis lassulást okoz a

sztring műveleteknél. Megjegyzendő, hogy a Borland Pascal 7-es verziójától létezik a **Pstring** nevű adattípus, amely 0 végű karakterkezelést tesz lehetővé a Borland Pascal nyelvekben.

A karakterekkel nem lehet semmiféle matematikai műveletet végezni. Vannak azonban alapvető műveletek, amelyeket minden programozási nyelvben el lehet végezni. A műveletek általában függvény alakban léteznek és a különböző nyelveken más és más a megvalósításuk is. Azt is el kell mondani, hogy általában azok a műveletek, amelyek karakterre alkalmazhatók, alkalmazhatók sztringekre is.

A fenti adattípusokkal lehetséges műveletek:

Összefűzés	+	'A' + 'B' => 'AB'
Karakter kódja	ASCII(k), ORD(k)	ASC('A') => 65
Kód alapján katakter	CHR(szám)	CHR(66) => 'B'
Sztring bal oldala	Left(sztring, szám)	LEFT('ABC',1) => 'A'
Sztring jobb oldala	Right(sztring, szám)	Right('ABCD',2) => 'CD'
Sztring középső karakterei	Mid(sztring, szám, dbszám)	MID('ABCDE',2,3) => 'BCD'

Összehasonlításokban

A karaktereket ugyanazokkal a jelekkel hasonlítjuk össze, mint a karakterláncokat,

< kisebb, mint, > nagyobb mint, <= kisebb egyenlő mint, >= nagyobb egyenlő mint,

== egyenlő, <> vagy # nem egyenlő relációk lehetnek.

Mindig balról jobbra kezdi a rendszer karakterenként az összehasonlítást és az első különbségnél már el is dönti az eredményt. Ha két karakterlánc ugyanazt tartalmazza végig, de az egyik rövidebb, mint a másik, akkor az a „kisebb".

Olyan adatok, esetén, amikor az adatokat több különböző típusú operációs rendszerből kezelik célszerű az adatokat karakterekként tárolni és lehetőleg az ASCII kódrendszert felhasználni rá.

Megjegyzés:

A Pascal és a C nyelv sztringkezelése között van különbség. A Pascalban a programozási nyelv része a sztrinkezelő függvények és eljárások sorozata, a C-ben viszont külön könyvtári függvények vannak, amelyeket a programhoz kell szerkeszteni.

4.3.3 Logikai típus

A logikai adattípus két értéket vehet fel, Igaz, vagy Hamis értékeket. Ez a két érték különböző programnyelveken lehet a True-False, Yes-No, Igen-Nem, Nem nulla – Nulla értékpárosok valamelyike.

Bár a logikai adatokat elvileg egy bit is egyértelműen jelzi, de a gépek bináris felépítése és a gépi kódú utasításkészletek mindig egy byte méretű adatként kezelik.

A logikai adatokra lehet alkalmazni a következő műveleteket:

<, >, <=, >=, <> Logikai És, Logikai Vagy, Logikai Kizáró Vagy, Tagadás.

A szokásos jelölések – gyakorlatilag a legtöbb programozási környezetben:

<, >, <=, >=,<>, #, AND, OR, XOR, NOT.

4.3.4 Mutatók, pointerek

Definíció

A mutató olyan változó, amely a memóriában lévő adat memóriacímét tartalmazza.

A memóriában tárolt adatoknak mindig van címük. Azokat a változókat, amelyek más adatok memóriacímét tartalmazzák, mutatóknak vagy pointereknek nevezzük. A pointerek az egyes programozási nyelvekben és operációs rendszerekben nem kezelhetők egységesen, nem egyforma a méretük sem, de egy adott operációs rendszer, memóriamodell és fejlesztőeszköz esetén pontosan meghatározhatók a pointerek méretei.

Ha az adatok egy 64Kb-os memóriaterületen elférnek, akkor elegendő a pointernek csak a terület elejéhez viszonyított eltolást (offszet) tárolni. Ebben az esetben a pointer méretére elegendő csak 2 byte. Ha nagyobb memóriaterületet akarunk megcímezni, annak megfelelően kell a pointerek méretét növelni, általában 4 byte vagy 8 byte lesz a pointer mérete.

Definíció

A mutató (pointer) típusa annak az adatnak vagy változónak a típusából ered, amilyen adatra vagy változóra a pointer mutat.

Ilyenkor a pointernek más típusú adatra vagy változóra nem mutathat.

A Pascal és a C, mint két alapvető programozási nyelv kissé más szintaktikával jelöli a mutatókat és a végezhető műveletek köre is más és más.

A PC-k esetén a pointerek általában segmens:offset, módon, 4 bájton tárolódnak. Az Intel processzorok tulajdonságai miatt a tárolás alsó két bájtja az offszete, a felső két bájt a szegmens tartalmazza.

Pascal nyelv	
v változó címének letárolása p pointerbe	P:=@v vagy p:=Addr(v)
Hivatkozás a mutatott p változóra	p^
Értékadás **v** változónak pointer segítségével	p^ := 133
A „sehová sem mutató” pointer konstans	Nil vagy Ptr(0,0)
Összehasonlítások	<>,=
p pointer offszetje	Ofs(p)
p pointer szegmense	Seg(p)

A C nyelv lehetőségei a pointerekkel való műveleteket nagyon elősegítik, de a pointerek használata a programokat kissé áttekinthetetlenné teheti. Ha az ilyen programot jól írjuk meg, akkor a pointerek használata hatékonyságjavulást okoz.

C nyelv	
v változó címének letárolása p pointerbe	p = &v
Hivatkozás a mutatott **v** változóra	*p
Értékadás **v** változónak pointer segítségével	*p = 133
A „sehová sem mutató” pointer konstans	Null
Összehasonlítások	==, !=
pointer által mutatott változó módosítása	*p = *p + 33

4.3.5 Megszámlálható és nem megszámlálható adattípusok

Az eddig tárgyalt összes egész típusú adat, a karakter típus és a logikai típus egyben megszámlálható típus is. A megszámlálható típusokat lehet létrehozni úgy is, hogy felsoroljuk a típust alkotó lehetséges értékeket. A megszámlálható adattípusok egyes elemei bitmintaként tárolhatók, ezért általában a megszámlálható adattípusok tárolási egységei a byte 2 hatványai.

A fenti adattípusokkal lehetséges műveletek:

Első adat	Típus(0)	0.
Adattípus következő adata	Succ(adat)	Succ(’A’) => ’B’
Adattípus előző adata	Pred(adat)	Predd(199) => 198

4.3.6 Konverziók

A különböző nyelveken az egyes adattípusokat más és más függvényekkel lehet átkonvertálni. Erre azért van szükség, mert az egyes programozási nyelvek a különböző típusokat másképpen kezelik. A konverzió alól a C nyelv bizonyos mértékig mentesíthető, mivel a C-ben a rövid egész típust lehet karakternek is tekinteni, továbbá sztringet lehet rövid egészek tömbjének tekinteni stb… Az adatkonverziók leggyakoribb eljárásai:

Karakter kódja	ASCII(k), ORD(k)	ASC(’A’) => 65, ORD(’B’) => 66
Kód alapján katakter	CHR(szám)	CHR(66) => ’B’

Realból integer	Round(Real)	Round(1.321) => 1
Integerből Real	Int(egész típusú szám)	Int(344) => 344.00
Sztringből szám	VAL(sztring, szám, dbszám)	VAL('123.34', változó, hibakód) 'A Pascalban ez úgy működik, hogy a hibakód megmutatja, hogy a konverzió hibátlan volt-e'
Számból Sztring	STR(szám)	STR(1876.01) => '1876.00'

4.3.7 Globális- és lokális adatok kezelése, az adatok láthatósága

Amikor egy programban adatokkal dolgozom általában nincsen szükségem a programban felhasznált bármelyik adat elérésére. A jól strukturált programokban egy eljárás vagy függvény jól specifikálhatóan csak bizonyos adatokkal dolgozik. A feldolgozandó adatokat vagy az őt hívó eljárástól kapja meg, vagy a függvényben – eljárásban keletkezik és esetleg az őt hívó eljárásnak adja vissza, vagy amikor kilép a programrészletből, akkor elenyészik, mert nincsen rá szükség.

A BASIC programozási nyelv eredetileg globális adatokkal dolgozott csak, azaz minden adatot mindig el lehetett érni a program minden pontjáról. Bonyolultabb programok esetén az adatok következetes kezelése nehézkes, sőt csaknem megoldhatatlan.

A probléma megoldására vezették be a **globális** és a **lokális adat/változók** fogalmát.

Egy **adat lokális** egy programegységre nézve, ha abban a programegységben az adat elérhető, esetleg módosítható, míg a programegységet hívó programegységekből az adat nem látszik. Általában az ilyen adat akkor jön létre, amikor elkezdi végrehajtani a programegységet a program és akkor szűnik meg, ha kilép belőle.

Egy **adat globális** egy programegységre nézve, ha az adat már létezik akkor, amikor elkezdődik az egység végrehajtása. Nyilván ha egy adat globális egy programegységre nézve, akkor az egységből meghívott programrészekre is globális az adat.

Az **adatok láthatóságának** kérdése összefügg az adatok globalitásával is.

Általában ha egy **adat globális** egy programegységre nézve, akkor látható is, de ez nem minden programozási nyelven igaz.

A Pascalban csak azok a globális adatok láthatók, amelyek a programszövegben előrébb vannak deklarálva, vagy speciális kulcsszóval utasítjuk a fordítót, hogy tegye láthatóvá máshonnan is. A C nyelven a header fájlokban kell megadni azoknak a globális változóknak a definícióját, amelyeket más eljárásokban látni szeretnénk.

Ha egy eljárásban ugyanolyan nevű változót definiálunk, mint amilyen egy globális változó neve, akkor a **lokálisan definiált** változó válik láthatóvá, eltakarja a globális változót. Ez még akkor így van, ha a két változó típusa nem egyezik meg. Bizonyos nyelvek ebben az esetben fordításkor hibaüzenetet adnak, de ezzel most nem foglalkozunk. Ha kilépünk a kérdéses eljárásból, akkor természetesen megint az eredeti globális változó válik Ha a lokális adat létrejöttekor meghívunk egy eljárást és egyébként a nyelv szintaktikája engedi, akkor a meghívott eljárásban is látható az adat, hiszen arra az eljárásra nézve a kérdéses adat globális.

Vannak olyan speciális nyelvek, amelyek képesek létrehozni egy beágyazott – azaz valahonnan meghívott - eljárásban is a teljes programra nézve globális adatokat. Ilyen a Clipper programozási nyelv.

4.3.7.1 Numerikus feladatok

4.3.7.2 Állapítsuk meg egy számról, hogy prímszám-e? (prímteszt)

Specifikáció

Adott egy szám, állapítsuk meg a számról, hogy prím-e (Prímszám az a pozitív egész szám, amely csak 1-gyel és önmagával osztható!)

```
program primvizsgalat;
  szam,i int;
  prim   boolean;
  jel    char;

  //adat beolvasása

  Ki: ”Kérem a vizsgálandó számot”
  Be: szam
   //tegyük fel, hogy prím

  prim:=igaz;
  szam:=Abs(szam);
  //Ha páros, akkor nem prím
  Ha szam mod 2 = 0 akkor
     Ki: ”2, nem prím”
     prim:=FALSE; end;
   elágazás vége

  //főciklus
  i:=1;
  Ciklus amíg ((i*i<=szam) és prim )
     //      A páratlan számokat próbáljuk osztani
     Ha szam mod (i*2+1) = 0 akkor
        Ki: ”  ”,i*2+1
        prim:=hamis;
     elágazás vége
    i :=i+1
   ciklus vége

  //ha 2,3,5 volt, akkor prím
  Ha (szam=2) vagy (szam=3) vagy (szam=5) akkor prim:=igaz;

   Ha szam=1 akkor       // az 1 nem prím
     Ki: Nem prím
     prim:=FALSE
   Elágazás vége

   Ha prim akkor
     Ki: 'A szám prim'
   Különben
     Ki:'A számm nem prím'
   Elágazás vége
Program vége
```

4.3.7.3 Erasztothenészi szita

Specifikáció

Adjuk meg a prímszámokat N-ig.

Megoldás elve

Felsoroljuk az egész számokat 1-től N-ig éés sorban vizsgáljuk őket. Ha találunk egy prímszámot, akkor annak a többszöröseit bejelöljük, hogy nem prímek és a megvizsgálandó számok közül sokat kizártunk.

```
program eratosztenesz;
   Szita[N], logikai értékekkel teli

   Ciklus i:=1-től N-ig
     Szita[i] := igaz
   Ciklus vége,   for i:=1 to N do szita[i]:=TRUE;

   Ki: 2  //mert a 2 prím, a szamítás a 3-mal indul

  i:=3;
  { ciklus, amíg i nem lesz nagyobb a határ négyzetgyökénél }
  Ciklus amig i<= Négyzetgyök(N)
     Ha szita[(i-1) div 2] akkor
        Ki: i
        //A páratlan számszoros elemeket kiejtjük
        koz:=2*i;
        // az elsô a 3-szorosa a talált prímnek }
        j:=3*i;
        Ciklus amig j<=2*N+1          //{ a többszörös törlése a prímek közül }
           szita[(j-1) div 2]:=hamis;
           j:=j+koz;
        Ciklus vége
     Elágazás vége
     i:=i+2;
   Ciklus vége

   Ki: A talált prímek: 2,
   j:=1;
   Ciklus i:=1-től N-ig
     Ha szita[i] akkor
        J:=J+1
        Ki: 2*i+1
     Elágazás vége
   Ciklus vége
Eljárás vége
```

Feladatok

Írj programot, amely bekér két egész számot és meghatározza a két szám legnagyobb közös osztóját.

Írj programot, amely bekér két egész számot és meghatározza a két szám legkisebb közös többszörösét!

Írj programot, amely meghatározza egy szám prímtényezős felbontását!

Írj programot, amely bekér egy numerikus adatot a billentyűzetről, és addig nem fogadja el a választ, a bevitt adat hibátlan nem lesz! A hiba létéről értesítse az adatbevitelt végzőt!

Írj programot, amely megvalósít egy egyszerű 4 alapműveletes számológépet az egész számok körében! A program figyelmeztessen arra, hogyha egy művelet eredménye túlcsordul vagy alulcsordul és ne hajtsa azt végre! Az osztás eredményét, amennyiben nem egész írja hányados és maradék alakban!

Írj programot, amely megvalósít egy egyszerű számológépet, amely képes a 4 alapművelet, a négyzetre emelés, gyökvonás és néhány egyéb matematikai függvény elvégzésére! Ha nem megfelelő paramétereket adunk a matematikai függvénynek, akkor figyelmeztessen, de a program álljon le futási hibával! A program figyelmeztessen arra, ha egy művelet eredménye túlcsordul vagy alulcsordul és ne hajtsa azt végre!

Írj egyszerű titkosító programot! A program a bevitt vagy parancssorból paraméterként megadott stringet titkosítsa úgy hogy a karakterek ASCII kódját manipulálja. A legegyszerűbb módszer, az ASCII kód eltolása, bonyolultabb, ha az ASCII kód módosításához valamilyen helytől függő függvényt is használunk, de a legjobb az un. nyílt kódú titkosítás.

Írj programot, amely bevitt vagy parancssorból paraméterként megadott stringben lévő ékezetes karaktereket a táviratokban szereplő megoldással helyettesíti, azaz é = ee, Ő=OEE, ö=oe, stb…

Ír programot, amely bevitt vagy parancssorból paraméterként megadott stringben lévő egymás utáni több szóközt egy szóközzel, a tabulátor jelet 8 szóközzel helyettesíti!

A feladathoz fel lehet használni a stringek tömb természetét, az STR(), a LEN() függvényt. A STR() függvény ebben az esetben nem formázza a szöveget, hanem tizedestört alakban írja ki adja meg az eredményt!

4.4 Összetett adatszerkezetek

Az eddigiek során csak egyszerű adatszerkezetekről volt szó, de ezek általában nem elegendőek a programok által feldolgozandó adatok tárolására. A továbbiakban olyan adatszerkezeteket tárgyalunk, amelyek egyszerűbb adatszerkezetek összeépítésével jönnek létre.

A létrejött adatszerkezetek helyet foglalnak a memóriában.

Az adatszerkezetek tárolásához szükséges hely nagysága az adatszerkezetet alkotó egyszerűbb adatok számára szükséges helyek nagyságából tevődik össze. Memóriaszervezési és adminisztratív okokból alkalmanként az összetett adatszerkezet mérete nagyobb, mint az alkotó adatok méreteinek összege, azonban a méretnövekedés általában nem túl nagy az adatszerkezet teljes méretéhez képest, csak néhány bájtot jelent.

Feladatok

Írj programot, amely egy text fájlból beolvas egy szöveget és szavanként új sorba írja ki a képernyőre (a szó olyan karaktersorozat, amelyet alfanumerikus karakterket tartalmaz csak)!

4.4.1 Halmaz típus

Nem minden programozási nyelvben van meg ez a típus. A halmaz elemeit általában fel kell sorolni. A halmaz elemei ugyanolyan típusúak. Az elemeknek sorszáma van, általában 0..255 közötti sorszámuk lehet.

Művelet	Jel	Példa
Létrehozás	Set Of alaptípus	Var B : SET of ’A’..’Z’
Értékadás	:=	A := [’A’..’Z’]
Metszet	*	A:=A*B
Egyesítés	+	A:= A + B
Különbség	-	A := A-B
Egyenlőség	=	A = B
Tartalmazás	<=, >=	A <= B
Elem vizsgálata	IN	c IN [’y’,’Y’,’n’,’N’]

4.4.2 Tömbök, sorozatok

Definíció

*Amikor azonos típusú adatokat tárolunk a memóriában egymás után, akkor beszélhetünk tömbről vagy sorozatról. A tömböt alkotó adatokat a tömb **elemeinek** hívjuk. Az egyes elemekre a tömb **indexével** hivatkozunk. Az index általában egy sorszám. Egy tömb deklarációjakor meg kell adnunk **a tömb elemeinek típusát** és a **tömb méretét**.*

Ha a tömbnek csak egy indexe van, akkor egy dimenziós tömbről beszélünk. Az egy dimenziós tömböket a számegyenes egész helyein lévő adatokkal szemléltethetjük.

Ha egy tömbnek két indexe van, akkor két-dimenziós tömbről beszélünk, és így tovább… A két-dimenziós tömböket egy sík egész koordinátájú pontjaiban elhelyezkedő adatokkal szemléltethetjük.

A tömböket a memóriában a programok **sorfolytonosan tárolják**, ami két dolgot jelent. Egyrészt jelenti, hogy a tömb elemei egymás után helyezkednek el a memóriában, másrészt többdimenziós tömb esetén a mindig az utolsó index növekedik.

Például egy T[10][3][2] deklarációjú tömb elhelyezkedése a memóriában, ahol az indexek 0..9,0..2,0..1 ig terjednek:

T[0][0][0], T[0][0][1], T[0][1][0], T[0][1][1], T[0][2][0], T[0][2][1], T[1][0][0], T[1][0][1], T[1][1][0], T[1][1][1], …

Valójában a többdimenziós tömböt leképezzük egy-dimenziós tömbbe. A leképezés során a több dimenziós indexekhez hozzárendelünk egy egy-dimenziós indexet. Nézzük a fenti deklarációt! Ha a T[i][j][k] elem elhelyezkedését vizsgáljuk, akkor az egy dimenziós indexet az alábbi képlet adja meg:

```
Index := 3*2*i + 2*j +k
```

Általában, ha a tömb dimenziói $n_1,n_2,n_3,n_4,\ldots n_m$, nagyságúak és az indexek $i_1,i_2,i_3,\ldots i_m$, akkor a tetszőleges $T[i_1][i_2][i_3]\ldots[i_m]$ elem egy dimenziós indexe:

$index := i_1*n_2 + i_2*n_3 + i_3*n_4 +\ldots..+ i_{m-1}*n_m + i_m$

Ennek segítségével a tömb első elemének memóriacíme ismeretében, valamint a tömbelem méretéből ki lehet számolni tetszőleges elem memóriacímét.

Ha a fenti deklarációban a első elem címe **M**, akkor az i,j,k indexű tömbelem memóriacíme

$Cim_{i,j,k} := index * elemméret + M,$

azaz

$Cím_{i,j,k} := (3*2*i + 2*j +k) * elemméret + M$

Általános esetben,

$Cím_{i1,i2,\ldots,in} := (i_1*n_2*n_3*\ldots*n_m + i_2*n_3*\ldots*n_m +\ldots+ i_{m-2}*n_m * n_{m-1} + i_{m-1}*n_m + i_m) * elemméret + M$

A fenti képleteket **címfüggvénynek** hívják.

Felvetődik a kérdés, hogy az egy dimenziós elhelyezkedés ismerete eseté a több dimenziós elhelyezkedés kiszámolható-e egyértelműen.

A fenti példa alapján az Index ismeretében az i,j,k-t a következő algoritmussal lehet kiszámolni:

```
I := index div (3*2),
Ma:= index mod (3*2)
J := ( Ma ) div 2
K := Ma mod 2
```

Általános esetben a következő algoritmust kell használni:

$i_1 := Cím \text{ div } (n_2*n_3*\ldots*n_m)$
$Ma := Cím \text{ mod } (n_2*n_3*\ldots*n_m)$
$I_2 := Ma \text{ div } (n_3*\ldots*n_m)$
$Ma := Ma \text{ mod } (n_3*\ldots*n_m)$
…
$I_m := Ma \text{ mod } n_m$

Az egyes programozási nyelvek más és más szintaktikát használnak, de alapvetően ugyanazt érhetjük el velük.

A Pascalban a tömb kezdő és utolsó indexét kell megadnunk, innen tudjuk az tömb méretét, és meg kell adni az elemek típusát.

```
T: Array [első..utolsó] of Típus
```

Hivatkozás egy elemre: `T[5]`

Több dimenziós tömböt az alábbi módon lehet deklarálni:

```
T1: Array [első..utolsó, másik_első..másik_utolsó] of Típus
```

Hivatkozás egy elemre: `T1[5,6]`

A C-ben a tömb elemeinek darabszámát kell megadnunk és a tömb indexe 0-tól az elemszám-1 –ig tart.

```
Típus T[elemszám]
```

Hivatkozás egy elemre: `T[5]`

Több dimenziós tömböt az alábbi módon lehet deklarálni:

```
Típus T1[elemszám, elemszám1]
```

Hivatkozás egy elemre: `T1[5,6]`

A fentiek alapján könnyen kiszámolható egy elemnek a memóriában elfoglalt helye és a tömbnek szükséges memória mérete is. A tömbök elemeit a különböző nyelvek általában sorfolytonosan tárolják a memóriában, azaz több dimenziós tömb esetén először az első sor elemeit, majd a második sor elemeit és így tovább.

A tömb adatok feldolgozása szorosan összefügg a ciklusokkal. Ha olyan feladatot kell megoldani, amely a tömb minden egyes elemének feldolgozását jelenti, akkor megszámlálásos ciklusokat kell alkalmazni, ha pedig a tömb elemein valamilyen tulajdonság meglétét vizsgáljuk, akkor `ciklus amíg a feltétel nem igaz` jellegű ciklust kell használnunk.

Több dimenziós tömbök feldolgozásakor annyi egymásba ágyazott ciklust kell használnunk, ahány dimenziója van a tömbnek.

Definíció

Asszociatív tömb*nek olyan tömböt nevezünk, amelyben az index tetszőleges karaktersorozat lehet. Ilyen megoldást lehet látni PHP-ban, és más, egyéb, alapvetően szövegfeldolgozásra kifejlesztett scriptnyelvekben.*

Gyakorló feladatok tömbökre:

Egy 25x80-as mátrixba írjunk véletlenszerűen látható karaktereket. Írjunk programot, amely minden oszlopot eggyel balra léptet, az első oszlopot pedig az utolsó oszlop helyére teszi. (Esetleg képernyőn!)

Töltsük fel egy stringekből álló tömböt egy text file sorainak szavaival (szó, amit nem alfanumerikus karakter határol, azaz nem 0..9, és a..z, A,..,Z!

Tükrözzük egy adatokkal feltöltött két-dimenziós mátrixot a bal-felső jobb-alsó átlójára (főátló)!

Egy NxN-es mátrix adatait forgassuk el 90 fokkal! (pl. A[1,2] = A[2,N), stb.)

Töltsünk fel két NxM-es mátrixot adatokkal. Írassuk ki az összeg mátrix és a különbség mátrixot! (C[i,j] = A[i,j]-B[i,j])

Egy tömbben növekvő számokat tárolunk úgy, hogy a tömb középső elemébe írjuk a legkisebbet, a bal oldalára a következőt, a jobb oldalára a következőt, jobb oldalára a következőt és így folytatjuk, amíg be nem telik a tömb. Írjunk programot, amely meghatározza két tetszőleges sorszámú elem különbségét!

A föld felszínének magasságadatait két-dimenziós tömbben tároljuk. A felszín feletti magasságokat színekkel jelöljük (kék, tengerszint, zöld 200m alatt, barna 1000m alatt, sötétbarna 1000-2000mközött stb.) Írjunk programot, amelyik kiírja a képernyő megfelelő helyére a domborzat magasságának megfelelő színkódot.

Tároljunk egy dimenziós tömbben két-dimenziós mátrix bal alsó háromszögének adatait. Adjuk meg a címfüggvényt! Az egy dimenziós tömb adatait írjuk ki háromszög alakban!

Írjunk programot, amely kiszámolja a Pascal háromszög adatait elhelyezi az adatokat egy két dimenziós mátrix bal felső sarkában!

Egy sakktáblán véletlenszerűen ugrál egy ló sakkfigura. Írjunk programot, amely szimulálja az ugrálást. Álljon meg az ugrálás, amikor ugyanarra a mezőre ugrik még egyszer!

4.4.2.1 Gauss elimináció

Specifikáció

Készítsünk programot, amely meghatározza egy N ismeretlenes egyenletrendszer megoldásait. A program kérje be az együtthatókat, majd írja ki az eredményt.

A megoldás feltétele

Matematikából bizonyítható, hogy megoldás akkor van

ha **n** ismeretlen esetén legalább **n db** egyenlet van.

ha nincsenek egymásnak ellentmondó egyenletek,

ha nincsenek egymásból származtatható egyenletek.

A megoldás módszere

Tudjuk azt, hogy az egyenletrendszer egyenleteit konstanssal beszorozhatjuk, az egyenleteket összeadhatjuk és kivonhatjuk egymásból, az egyenletrendszer megoldása nem változik.

Találjunk ki módszert, amivel a fenti műveletekkel az eredményhez eljutunk.

Nézzük az alábbi példát:

```
I.     5x + 3y +4z = 8
II.   7x + 9y +2z = 2
III. - x -  y - z = 3
```

1. lépés - Az I. egyenletet 7/5 szörösét vonjuk ki a II. egyenletből

```
I.     5x +   3y +   4z =  8
II.     0 + 4,8y - 3,6z = -9,2
III. - x -    y -    z =  3
```

2. lépés - Az I. egyenlet -1/5-szeresét vonjuk ki a III. egyenletből

```
I.     5x +   3y +   4z = 8
II.     0 + 4,8y - 3,6z = -9,2
III.    0 - 0,4y - 0,2z =  4,6
```

Vegyük észre, hogy a II. és III. egyenletben az x együtthatója 0, tehát a teljes tag értéke 0!

3. lépés - Az I. egyenletet beszorozzuk 1/5-tel.

```
I.   x  +0,6 y + 0,8z =  1,6
II.  0  + 4,8y - 3,6z = -9,2
III. 0  - 0,4y - 0,2z =  4,6
```

A fenti lépések eredményeként az első sort 1 oszlopában az ismeretlen együtthatója 1, az első oszlop többi együtthatója pedig 0.

4. lépés - A II. egyenlet, -0,4/4,8 szorosát kivonjuk a III.-ból.

```
I.   x + 0,6y + 0,8z =  1,6
II.  0 + 4,8y - 3,6z = -9,2
III. 0 -  0   - 0,5z =  3,83
```

5. lépés - A II. egyenletet osztjuk 4,8-cal!

```
I.   x + 0,6y + 0,8 z = 1,6
II.  0 +    y - 0,75z =-1,92
III. 0 -    0 - 0,5 z = 3,83
```

A II. egyenlet 2. ismeretlenének az együtthatója 1, az alatta lévőé 0.

6. lépés - A III. egyenletet szorzom 1/-0,5-tel

```
I.   x +0,6y + 0,8 z =  1,6
II.  0 +   y - 0,75z = -1,92
III. 0 -   0 +     z = -7,67
```

A **z** értéke megvan.

7. lépés - Helyettesítsük be a **z** értékét a II. egyenletbe:

```
I.    x  +0,6 y +0,8 z  =   1,6
II.  0  +     y + 5,7525  =  -1,92
```

Vagyis y = -7,67

8. lépés - Helyettesítsük be z és y értékét az I. egyenletbe:

Megkapjuk x értékét

```
I. x- 4,602 - 6,136 = 1,6
```

Vagyis x = 12,338

A fenti gondolatmenet alapján az algoritmus szavakban:

Az egyen ismeretlenjeinek együtthatóit beviszem egy **n** sorból és **n+1** oszlopból álló kétdimenziós tömbbe. Ennek a tömbnek beszélhetünk a **főátlójáról**. Ezek az A[i,i] elemek.

Az egyenletek összeadása, a tömb megfelelő sorban lévő elemeinek összeadását, egy egyenlet szorzása a megfelelő sor elemeinek szorzását jelentik.

Ciklikusan végigmegyek a sorokon i=1-től n-ig.

Az i. sor esetén a főátló együtthatóját felhasználva (A[i,i]) a következő sorokból (j. sor) kivonom az i. sor valahányszorosát és ez így eliminálom a j. sorban az i. főátló alatti elemet:

A[j,i] = A[j,i] – A[i,i] * A[j,i]*1/A[i,i] =>0

A többi elemet pedig ugyanezzel az együtthatóval kiszámolom az i. sor megfelelő elemei alapján:

A[j,k] = A[j,k] – A[i,k] * A[j,i]*1/A[i,i]

Lesz egy belső ciklusom j= i+1-től n-ig

Ezen belül egy harmadik ciklusom, amely k=j-től megy n+1-i

```
Eljárás Gauss elimináció;
   Tömb: A[N,N+1]
   Ciklus i:= 1-től N-ig       //N db transzformáció lesz

      S := A[i,i]              //A főátló eleme

      Ciklus j:= i+1-től N+1-ig
         Ciklus k:= 1-től N-ig
            A[j,k] = A[j,k] – A[i,k]*A[j,k]/s
         Ciklus vége
      Ciklus vége

      // végigosztom az i. sort a főátlóban lévő elemmel
      Ciklus k := i-től N+1-ig
         A[i,k] := A[i,k] / S
      Ciklus vége
   Ciklus vége

   //Ez itt a Gauss elimináció vége
   //Ezen a ponton a főátló alatt csupa 0 együttható található, a főátlóban
   //csupa 1.

   //Az ismeretlenek értékének kiszámítása következik. Az ismeretlen értéke

   //Az A[i,N+1] elemekbe kerül
   Ciklus i:=N-től 1-ig -1-esével
      S := A[i,N+1]
      Ciklus k:=i+1-től N-ig
         S:= S- A[i,k]*A[k,N+1]
      Ciklus vége
   Ciklus vége

Eljárás vége
```

A fenti algoritmust természetesen kissé másképpen is meg lehet valósítani.

4.4.2.2 Gauss-Jordan elimináció

Gauss-Jordan eliminációnak hívjuk, amikor a főátló felett is kinullázzuk a tömböt. Ekkor a Gauss eliminációval már a főátló alatt kinulláztuk a tömböt. Az N. sor, N. oszlopának értékét felhasználva kinullázzuk az N. oszlop elemeit. Mivel az utolsó sorban csak az N. és az N+1-ik oszlopban van érték, ezért a feljebb lévő sorokban csak ezek az értékek változnak.

Utána az N-1-ik sorral és az N-1-ik oszloppal nullázzuk az átló feletti elemeket.

A végeredmény az lesz, hogy a főátlóban csupa 1 áll és az N+1-ik oszlop tartalmazza az ismeretlenek értékét.

4.4.3 Rekord típus

A rekord adattípus elemei különböző típusúak lehetnek. Egy rekord alkotóit mezőknek hívjuk. A rekordok tárolása a memóriában a definiciók sorrendjében történik.

A rekordok mérete általában nem korlátos, bár a Pascalnál természetesen megvan a 64 kb-os korlát. A szokásos jelölések a következők.

	Pascal	C

Definició	Type nev =
	Record
	1.mezőnév:típus
	2.Mezőnév:típus
	……
	end;
		Typedef struct nev {
		Típus 1.mezőnév;
		Típus 2.Mezőnév;
		……;
		};

	Type tanulo = Record
	Nev : string[20];
	Neme: string[4];
	……;
	end;
	Var Gyerek: Tanulo;
		Typedef struct tanulo {
		char nev[26];
		char Neme[4];
		……;
		};
		Tanulo Gyerek

Hivatkozás egy rekord mezőire	Tanulo.nev	Tanulo.nev

A Pascalban és a C-ben is léteznek változó rekordszerkezetű adattípusok. A C-ben ezeket unionoknak hívják. Ekkor egy mező értékétől függően a rekord többi mezőjében tárolt adatok típusa, mérete is változhat. Az union esetén a lefoglalt méret a legnagyobb összetevő mérete lesz. Amikor az union mezőire hivatkozunk, akkor a lefoglalt területből csak annyit használ a rendszer, amibe az éppen tárolt adattípus elfér, de a többi memóriát nem szabadítja fel, hanem továbbra is lefoglalva marad.

Természetesen rekordokból is lehet definiálni tömböket. Ilyet tipikusan akkor használhatunk fel, ha egy adatfájl ugyanolyan típusú rekordokból épül fel, mint amiből a tömböt építjük fel. A fájl adatait beolvassuk a tömbbe, majd ott feldolgozzuk.

4.5 Felhasználók által definiálható típusok

4.5.1 Sor adattípus, FIFO

Az eddigi adatszerkezetek majdnem minden programozási nyelvben megtalálhatók a nyelv részeként. A továbbiakban olyan adatszerkezeteket tekintünk át, amelyek általában a nyelvekben nincsenek explicit módon utasításszinten megvalósítva, a programozónak saját magának kell létrehoznia azokat.

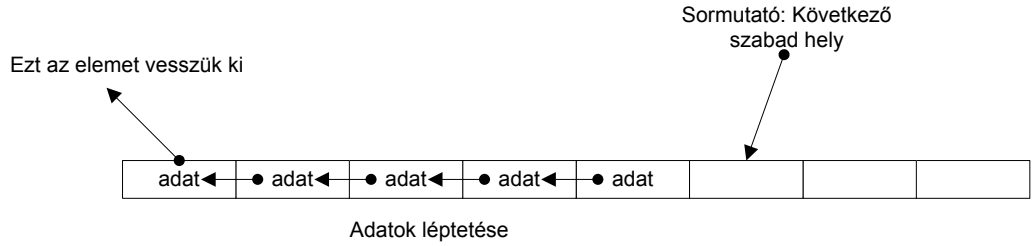
A következő adattípusok alapja egy legalább egy dimenziós tömb. Ennek az egy dimenziós tömbnek a szerkezetére építjük rá a következőkben az adatszerkezeteket. Természetesen, ha a programozó dinamikus memóriakezeléssel tömbök nélkül is meg tudja oldani a feladatot, akkor talán gyorsabb feldolgozó programokat kap, de tömbök felhasználásával egyszerűbben megvalósíthatók az adatszerkezetek.

A sor egy olyan adatszerkezet, amelybe az egymás után beírt adatokat a beírás sorrendjéből vehetjük ki (FIFO – First In First Out). Az adattípusnak akkor van létjogosultsága, amikor két nem egyforma sebességgel működő rendszer között adatátvitelt akarunk megvalósítani. Akkor is hasznos két rendszer közé egy soros adatszerkezetet ékelni, ha az adatokat átadása vagy az adatok átvétele nem folyamatos, hanem lökésszerű. Például ilyen lehet a számítógép telefonos kapcsolata vagy egy számítógép és a hálózat kapcsolata.

A sor adatszerkezetnek két utasítása van. Az IN utasítás egy elemet betesz a sorba, az OUT utasítás kiveszi a sor következő elemét. A betételnél arra kell vigyázni, hogy ha megtelt a sor részére fenntartott memóriaterület, akkor nincsen hely a beteendő adat részére – hibaüzenetet kell adni -, ha üres a sor, akkor pedig a

kivételnél kell visszaadni hibaüzenetet. Az is megoldás, hogy üres sor esetén olyan adatot adunk vissza, amely nem fordulhat elő érvényesen az adatok között. A sort kétféleképpen is meg lehet megvalósítani.

Egyszerű megvalósítás



Az első megvalósítás egyszerű és lassú. A sor adatszerkezetnek van egy sor mutatója. Ez a mutató mutatja meg, hogy hol van a következő szabad elem a tömbben. Ha betettünk egy elemet, akkor a szabad hely mutatója növekedik eggyel. Ha kiveszünk egy elemet, akkor az első elemet vesszük ki a tömbből, és minden elemet eggyel kisebb indexű helyre másolunk.

```
T[N] elemű tömb jelenti a sort. A tömb indexe 1-től indul. Sormutató jelenti az
első üres helyre mutató változót. Kezdetben a sormutató értéke = 1.

Függvény IN(adat) : Logikai              (Ha Hamis, akkor tele van a sor,
  Ha Sormutato = N+1 akkor           ha igaz, akkor akkor nincsen tele)
    In := Hamis
  Különben
    T[sormutato] := adat
    sormutato := sormutato + 1
    In := Igaz
  Elágazás vége
Eljárás vége


Függvény OUT
  Ha sormutato = 1 akkor
    Ki:"Üres a sor"
    OUT := NIL
  Különben
    OUT := T[1]
    Ciklus i:=2-től sormutato-ig
      T[i-1]:=t[i]
    Ciklus vege
    sormutato := sormutato - 1
  Elágazás vége
Eljárás vége
```

A megvalósítás hátránya a kivételnél a ciklikus másolás. Ez lassítja a kivételt.

Második megvalósítás

A második megvalósításnál is egy N elemű, T tömb jelenti a sort, ahol az N elemű tömböt 0-tól N-1-ig indexeljük. Itt két mutatót használunk. A `sorba` jelenti azt a helyet, ahová a következő elemet beteheti az IN utasítás. A `sorbol` jelenti azt a helyet, ahonnan ki lehet venni a következő elemet. Az IN utasítás beteszi a `sorba` által mutatott helyre az új adatot, majd növeli a `sorba` mutatót, míg az OUT utasítás a `sorbol` ál-

tal mutatott helyről veszi ki az adatot és növeli a `sorbol` mutatót. Ha bármelyik mutató túlmutat a tömb utolsó elemén, akkor a mutatót a tömb első elemére irányítjuk.

Itt nagyon fontos annak a megállapítása, hogy mit jelent az, hogy a sor üres és mit jelent az, hogy a sor tele van. Normális esetben a két mutató nem mutathat ugyanarra a helyre. Ekkor tudunk elemet betenni a sorba, és tudunk elemet is kivenni a sorból. Mikor üres a sor? Ha a psorbol ugyanoda mutat, mint a psorba, ekkor ugyanis a kiveendő adat megegyezik a következő beteendő adat helyével. Mit jelent az, hogy tele van a sor? A `sorba` mutató utoléri a `sorbol` mutatót, azaz ugyanaz az értékük??? Ez így nem lehetséges, hiszen ugyanaz jelenti mind a két eseményt.

A megoldás egyszerű és majdnem ugyanez! Legyen tele a sor tele, akkor ha a `sorba` eggyel kisebb, mint a `sorbol` mutató. Legyen üres a sor, ha a `sorbol` mutató eggyel kisebb, mint a `sorba` mutató.

A fenti okoskodás eredményeképpen a sor mérete eggyel kisebb lesz, mint szeretnénk, hiszen egy elemet soha nem tudunk kihasználni. Sebaj, növeljük meg a tömb méretét N+1-re, azaz a tömb 0-tól N-ig indexelt, így újra N db elemet tudunk tárolni maximálisan a sorban. Természetesen inicializálni kell a sort, azaz üres sorral kezdünk a program elején.

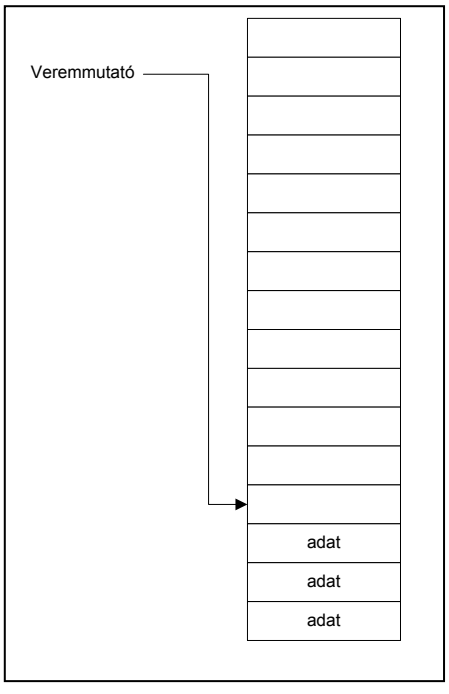
Még egy problémára kell választ találni. Mi van, ha akármelyik mutató elérte az utolsó helyet és növelnem kell tovább? Ebben az esetben a mutató N-nel való osztásának maradékát kell összehasonlítani a másik mutatóval. A két kód a következőképpen néz ki.

Inicializálás:

```
sorba  := 2
sorbol := 1
……
Eljárás IN(adat)
   Ha ((sorba+1) mod N) = sorbol akkor
     Ki:” Tele van a sor, adatvesztés”
   Különben
     T[sorba] := adat
     sorba := (sorba + 1) mod N
   Elágazás vége
Eljárás vége

Függvény OUT
   Ha ((sorbol+1) mod N ) = sorba akkor
     Ki:”Üres a sor”
     OUT := NIL
   Különben
     OUT := T[sorbol]
     sorbol := (sorbol + 1) mod N
   Elágazás vége
Eljárás vége
```

A második eljárás gondolatmenete bonyolultabb, de láthatólag gyorsabb kódot eredményez.



4.5.2 Verem adattípus

A verem olyan adatszerkezet, amelybe az adatokat betehetjük, illetve kivehetjük. A veremből mindig az utoljára bevitt adatot vehetjük ki először (LIFO — Last In First Out, vagy vagy stack-nek is hívják)

Rengeteg helyen használjuk a verem adatszerkezetet. Minden program, amely az operációs rendszerben fut, sőt maga az operációs rendszer is rendelkezik veremmel. Amikor egy program meghív egy eljárást, a program futásának pillanatnyi címét a verembe teszi el a gép, majd az eljárásból visszatérve a veremből kapja vissza a futás megszakadásának helyét és így tudja ugyanonnan folytatni a gép a program végrehajtását.

Ha általában a veremből kivett adatok mennyisége vagy sorrendje nem egyezik meg a betett adatok mennyiségével és sorrendjével, akkor egy program könnyen olyan helyzetbe kerülhet,

hogy a program folytatásának címe nem megfelelő. Ennek következménye általában a program lefagyása.

A különböző programozási nyelvek eljárásai, függvényei szintén vermek felhasználásával adják át egymásnak a paramétereket.

A verem adatszerkezetnek van egy veremmutatója (stack pointer). A verem mérete korlátos. A veremmutató azt mutatja, hogy hol van az első szabad hely a veremben. Mint említettük két művelet van a veremmel kapcsolatban.

Az adat betétele a verembe – ezt PUSH utasításnak szokás hívni. Egy adatot akkor lehet betenni, ha a verem még nem telt meg. Ha tele van a verem és még egy elemet be akarok tenni a verembe, akkor az utasítás hibaüzenettel kell, hogy visszatérjen.

Ha a verem nincsen tele, akkor a veremmutató által mutatott helyre beteszem az adatot, majd a veremmutatót növelem eggyel.

A korábbiaknak megfelelően egy T nevű, N elemű tömb lesz a verem. A tömb indexe 1-től indul és N-ig tart. T globális változó hogy az eljárást a program szükséges helyéről el lehessen érni.

```
Eljárás PUSH(adat)
   Ha Veremmutato = N+1 akkor
     Ki:” Tele van a verem”
     Adat := lehetelen adat
   Különben
     T[veremmutato] := adat
     Veremmutato := veremmutato + 1
   Elágazás vége
Eljárás vége
```

A PUSH műveletben az adat paramétert célszerű cím szerinti paraméterátadással használni.

Az adat kivétele a veremből szokásosan a POP nevet viseli. Itt is felmerülhet egy hiba, nevezetesen, hogy van-e adat a veremben. Üres a verem, ha a veremmutató a verem első helyére mutat. A POP utasítás ilyenkor hibaüzenettel tér vissza. Ha nem üres a verem, akkor kivesszük az adatot, majd csökkentjük a veremmutatót eggyel.

```
Függvény POP
   Ha Veremmutato = 1 akkor
     Ki:”Üres a verem”
     POP := NIL
   Különben
     POP := T[veremmutato]
     Veremmutato := veremmutato - 1
   Elágazás vége
Eljárás vége
```

Megjegyzés:

A POP műveletet célszerű függvényként definiálni. Mind a két műveletnél hiba esetén is szükséges visszatérő adatot generálni, célszerűen az adat szabályos körülmények között nem előforduló adat legyen.

A fenti algoritmus megvalósíthatjuk Objektum orientált módon is, ekkor a két definiált eljárás és függvény lesz az objektum két metódusa.

A verem felhasználására egy jellegzetes alkalmazást mutatunk be.

4.4.1. Lengyel forma

Lukasewich lengyel matematikus az 50-es években a matematikai formulák olyanfajta átalakítását dolgozta ki, amelynek segítségével a fordítóprogram könnyen ki tudja számítani a kifejezés értékét. (Pontosabban: olyan kódot generál, amely – végrehajtva – kiszámítja a kifejezés értékét.) Erre azért volt szükség, mert az ember által megszokott „infix” és zárójeles írásmód nem látszott alkalmas struktúrának a kiértékelés céljára. (Infix jelölés amikor X+Y, a két operandus közé tesszük a műveleti jelet). A bevezetett új ábrázolási formát a szerző tiszteletére *lengyelformának* is nevezik. Másik elnevezés a **postfix forma.**

Mind a lengyelformára hozás, mind pedig annak kiértékelése vermet használó algoritmus, és jobban alkalmazható kifejezések kiértékelésére, mint az infix forma.

A forma lényege, hogy minden műveleti jel közvetlenül az operandusai mögött áll.

A lengyelformára hozott kifejezés előnyei:

nincs benne zárójel, azaz a műveletek precedenciáját nem kell megváltoztatni

az operandusok sorrendje egymáshoz képest változatlan,

Egyszerű példák:

a + *b*	›	*a* b +	
a * *b* + *c*	›	*a b* * *c* +	(eltérő precedenciák)
a * (*b* + *c*)	›	*a b c* + *	(zárójel hatása)
a + *b* – *c*	›	*a b* + *c* -	(azonos precedenciák)
a ^ 2 ^ 3	›	*a* 2 3 ^ ^	(hatványozás esetén fordítva: *a* ^ 2 ^ 3 = a ^ 8)

4.5.3 Lista adattípus

Definíció

*A **lista** olyan adattípus, lista elemekből áll. A lista egy eleme tartalmazza a tárolt adatot, és tartalmazza a következő listaelem helyét a memóriában. Ilyenkor **egy irányú listáról** beszélünk, mivel az adatok elérése csak a listaelemek sorrendjében történhet. A lista első elemét egy speciális adat, a **listafej** adja meg.*

Definíció

***Kétirányú listáról** beszélünk, ha a lista eleme nem csak a következő, hanem az előző adat helyét is tartalmazza.*

Definíció

Körkörös listáról beszélünk, ha az utolsó elem mutatója az első elem mutatójára mutat vissza.

A listákat legegyszerűbben egy korábban már tárgyalt adattípusra, a tömbre építhetjük rá. Ebben az esetben a listaelem rekordokból álló tömb. A tömb egy eleme a lista egy eleme. A tömbelem egy rekordjában az egyik összetevő tartalmazza a tárolandó adatot, a másik a következő elem indexét tartalmazza. A listákat olyan helyen célszerű használni, ahol az adatok elérése nem sorfolytonos, az adatok közé veszünk fel új elemeket és törlünk is ki.

A gyakorlatban a lemezek FAT tábláját, a programok memóriakezelését és sok egyéb területet listakezeléssel valósítanak meg.

Minden lista rendelkezik **listafejjel**. A listafej megmutatja, hogy hol található a lista első eleme, ugyanis nem biztos, hogy a lista első eleme egyúttal a fizikailag is a listát megvalósító alaprendszer első eleme. A listafej egy változó. Ha a listafej lehetetlen, vagy un. NULL értéket tartalmaz, akkor a listában nem tárolunk értékes adatot. Ha a listafej érvényes helyre mutat, akkor ott a lista egy eleme található. A listában szereplő elemek láncot alkotnak. A lista utolsó elemét úgy jelöljük, hogy a mutató lehetetlen címet, NULL elemet tartalmaz (pl. negatív számot).

Ha a listába beszúrunk egy elemet, honnan tudjuk, hogy a memóriába hova tudjuk elhelyezni azt? Másik probléma, hogyha a listából kitörlünk egy elemet, akkor a felszabaduló memóriaterületet hogyan tudjuk később ismét hasznosítani?

A megoldás a Szabadlista használata.

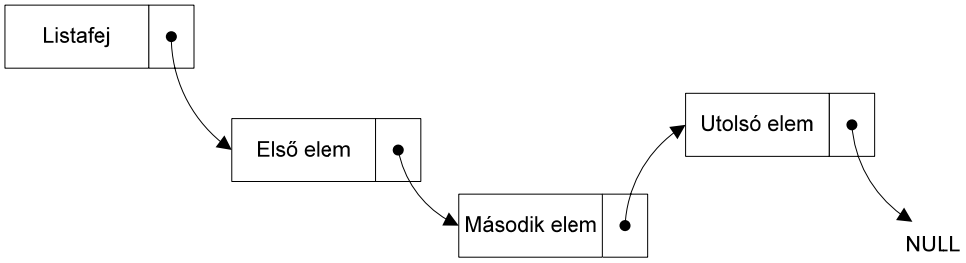
Definíció

A szabadlista olyan lista, amely nem tartalmaz adatokat, de az üres területeket listába fűzve tárolja.

Ha az adatokat tartalmazó listából kitörlünk egy adatot, akkor azt a szabadlista végére fűzzük, ha egy új elemet akarunk a listánkba betenni, akkor a szükséges helyet a szabadlistából vesszük el. Kezdetben, amikor még üres a lista, minden elem a szabadlistába van felfűzve. A memória akkor telik meg, ha a szabadlistában már nincsen elem.

Lista bejárása

A lista adatait úgy lehet elérni, hogy a lista elejétől a végéig végigjárjuk az elemeket, és minden elem feldolgozása után a következő elemre ugrunk, amíg el nem érjük az utolsó elemet.



A lista egyfajta megvalósítása a következő lehet. Egy N elemű, T tömb tartalmazza a listát. A tömb elemeit alkotó rekordok az `adat` és a `kovetkezo` nevű mezőkből áll, ahol az `adat` nevű mező tetszőleges adatot tartalmazhat, míg a `kovetkezo` egy egész típusú, megfelelő méretű mező.

```
Típusdefiníció:
L = rekord
     Adat : tetszőleges adattípus
     Kovetkezo: egész típus
   Rekord vége

T[N], L típusú elemekből álló tömb.
```

A fent definiált adatszerkezetben a lista elejéről el lehet jutni a lista végéig úgy, hogy egyesével végigmegyünk a lista elemein. Ezt a lista bejárásának hívják.

Lista bejárása:

```
Eljárás Bejaras
   i := Listafej
   Ciklus amíg i <> NULL
      Ki: T[i].adat                              //***
      i := T[i].kovetkezo
   Ciklus vege
Eljárás vege
```

Hogyan tudjuk a listában szereplő adatok összegét vagy a listában lévő elemek számát megadni? A programozási tételek megfelelő részeit áttanulmányozva ezekre a kérdésekre választ kaphatunk. A listában való mozgás megfelel a tömbökben való lépésnek!

Az egyirányú listákat csak az elejéről kezdve lehet bejárni!

Sok kérdés közül egy, az utolsó elem megkeresése a listában

```
Függvény Vegkeres(elso_elem)
   i := elso_elem
   Ciklus amig T[i].kovetkezo <> NULL
     i := T[i].kovetkezo
   Ciklus vege
   Vegkeres := i
Fuggvény vége
```

Hogyan kereshetünk meg egy adott elemet a listában? Az alábbi példában függvényként adjuk meg az algoritmust, és paraméterként adjuk át a keresett adatot.

Egy elem keresése

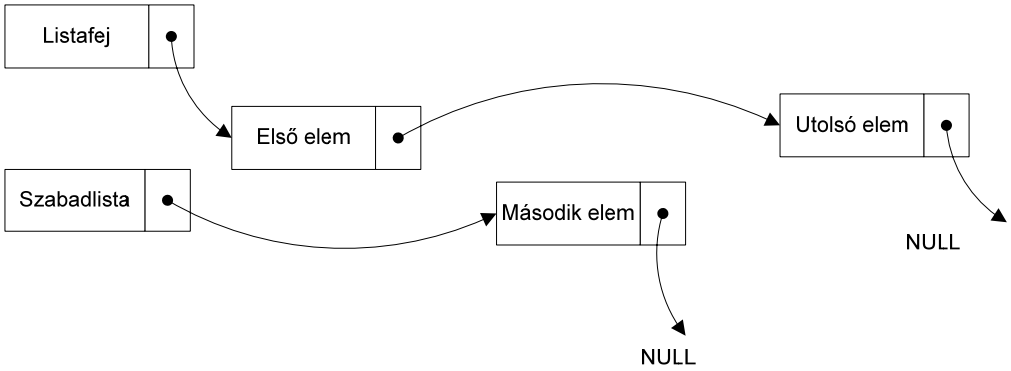
```
Függvény Kereses(Keresett_adat)
   i := Listafej
   Ciklus amíg (i<>NULL) és (T[i].adat <> keresett_adat)
     i := T[i].kovetkezo
   Ciklus vege
   Ha i <> NULL akkor
     Ki: ”A keresett adat sorszama:”, i
     Ki: ” A keresett adat:”, T[i].adat
     Kereses := i
   Kulonben
     Ki: “Nem létezik a keresett adat”
     Kereses := NULL
   Elágazás vége
Függvény vege
```

Hogyan tudunk egy új elemet betenni a listába, és egy elemet kitörölni a listából? Először egy keresett elemet töröljünk ki a listából. A törléshez felhasználjuk az előbb definiált keresés algoritmust egy kicsit módosítva. A Kereses algoritmus `elozo` paraméterét cím szerint adjuk át!

```
Eljárás Torlés
   Be: torlendo_adat
   elozo  := NULL
   elso_elem := Listafej
   i := Keresés(torlendo_adat, elozo, elso_elem)
   Ha i <> NULL akkor
     T[elozo].kovetkezo := T[i].kovetkezo
   Elágazás vége
Eljárás vége

Függvény Kereses(keresett_adat, elozo, elso_elem)
   i := elso_elem
   Ciklus amíg (i<>NULL) és (T[i].adat <> keresett_adat)
     elozo := i
     i := T[i].kovetkezo
   Ciklus vege
   Ha i <> NULL akkor
     Kereses := i
   Kulonben
     Kereses := NULL
   Elágazás vége
Függvény vége
```

Ha a 2. elem a kitörlendő adat, akkor az 1. elem következő mutatóját kell átállítani úgy, hogy a 3. elemre mutasson.



Mi történik ilyenkor a felszabadult hellyel? Két lehetőség van.

Az első esetben a felszabadult hely mutatóját NULL-re állítjuk, az adatmezővel nem törődünk, hiszen ettől kezdve nem használjuk az ott tárolt adatokat. A lista inicializálásakor a linkelési adatokkal fel kell tölteni a tömböt és később is következetesen tartani magunkat az elhatározáshoz. Ebben az esetben a törlő eljárás így változik:

```
Eljárás Torlés1
  Be: torlendo_adat
  elozo  := NIL
  elso_elem := Listafej
  i := Keresés(torlendo_adat, elozo, elso_elem)
  Ha mutato <> NIL akkor
    T[elozo].kovetkezo := T[i].kovetkezo
    T[i].kovetkezo := NIL
  Elágazás vége
Eljárás vége
```

A másik lehetőség bonyolultabb, de a valósághoz jobban közelít. Minden listakezelő rendszerben általában két listát tartanak nyilván. Az egyik lista a foglalt elemeket tartalmazza, míg a másik lista a szabad elemeket tartja nyilván. A foglalt elemek listájából való törlés azt jelenti, hogy az elemet áttesszük a szabad listába. A szabad és a foglalt lista elemei összességében kiadják a listát tartalmazó rendszer összes elemét.

Legyen a szabadlista feje a SzListafej, a szabadlista első elemére mutató elem az SzListafej.mutato. Ekkor úgy módosul a program, hogy miután megtaláltuk a törlendő elemet, az elemet betesszük a szabadlista elejére.

```
Eljárás Torlés1
  Be: torlendo_adat
  elozo  := NIL
  elso_elem := Listafej
  i := Keresés(torlendo_adat, elozo, elso_elem)
  Ha i <> NIL akkor
    T[elozo].kovetkezo := T[i].kovetkezo
    Sz_Elso_elem := Sz_Listafej
    T[i].adat := NIL
    T[i].kovetkezo := Sz_Elso_elem
  Sz_Listafej := i
  Elágazás vége
Eljárás vége
```

Új adat a lista végére

Hogyan lehet új adatot betenni a lista végére? Végig kell menni a lista elemein, majd az utolsó elemet megtalálva a szabad lista első elemébe kell betenni az adatot, az utolsó elem mutatóját ráirányítani az új elemre és a szabadlista fej mutatóját ráirányítani a szabadlista következő elemére. A beszúrt elem mutatóját NIL-re kell állítani, valahogy így:

```
Eljárás Ujadat_a_végére
   Be:  Adat
   vegso     := Vegkeres(Listafej)
   Regi_Sz_elso    := Sz_Listafej
   Sz_elso         := T[Regi_Sz_elso].kovetkezo
   T[vegso].kovetkezo          :=Regi_Sz_elso
   T[Regi_Sz_elso].adat        := Adat
   T[Regi_Sz_elso].kovetkezo   := NIL
   Sz_Listafej                 := Sz_elso
Eljárás vége
```

A fentiek alapján megoldható az adat beszúrása tetszőleges helyre is.

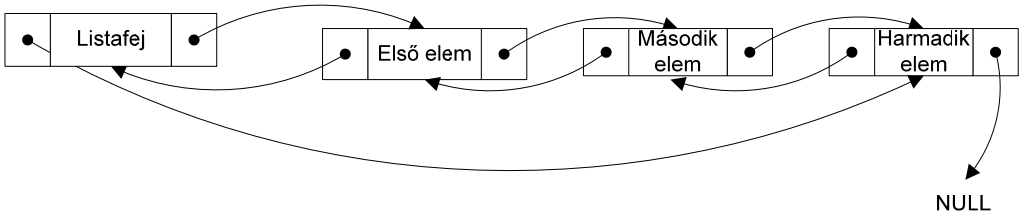
Kétirányú listák.

Gyakran előfordul, hogy a listákban nem csak az elejétől a vége felé kell mozogni, hanem visszafelé is szükséges. Erre a bonyolultabb megoldás az, mindig jegyezzük azt, hogy hányat mozdultunk a listában. A visszafelé mozgást is elölről kezdjük el, számoljuk, hogy hányat léptünk és eggyel előbb megállunk, mint korábban. Érezhetően bonyolult. Egyszerűbb megoldás, ha a lista rekordszerkezetébe felveszünk még egy mezőt. Ez a mező az előző elemre mutat majd. A lista definíciója ekkor így alakul:

Típusdefiníció:

```
L1 = rekord
   Adat      : tetszőleges adattípus
   Kovetkezo : egész típus
   Elozo     : egész típus
Rekord vége
```

T[N], L1 típusú elemekből álló tömb.



Nyilvánvalóan ekkor a Listafej tartalmaz még egy adatot, mégpedig az utolsó elem indexét, hiszen csak így lehet visszafelé bejárni a végéről a listát.

Körkörös lista

A kétirányú listának az egyik speciális esete a körkörös lista, amikor a lista utolsó elemének mutatója az első elemre mutat. A listafej is egy a lista elemei közül, de a lista létrejöttekor automatikusan generálja a rendszer. Ilyen adattípussal lehet megvalósítani pl. egy kooperatív multitaszkos (Windows 3.1 multitaszk rendszere) rendszert, amikor a rendszerben futó programok egymásnak adják át a vezérlést a nekik kiszabott idő lejártakor.

Feladatok:

Határozzuk meg egy listában tárolt számértékek összegét!

Válasszuk ki a lista legnagyobb elemét!

Egy növekvően rendezett listába szúrjunk be egy elemet a nagyságának megfelelő helyre, vagyis a bejárás, növekvő nagyságú elemeket írjon ki!

Fordítsuk meg egy egy-irányú listában az elemek sorrendjét!

Adott két rendezett lista, fűzzük össze a két lista elemeit úgy, hogy a továbbiakban is rendezett lista legyen!

Alakítsunk ki egy listát egy tömbből. A listát töltsük fel növekvő adatokkal, majd véletlenszerűen tegyünk át elemeket a szabadlistába. A billentyűzetről bevitt adatokat szúrjuk be nagyság szerinti helyükre. Minden beszúrt adat után írassuk ki a tömb állapotát, a beírt adatot és a listában elfoglalt helyét.

4.5.4 A gráf, mint matematikai fogalom

A matematika egyik gyakran használt fogalomrendszere a gráfelmélet. Miért is jó ez nekünk? Nagyon sok helyen az informatikában a gráfelmélet eredményeit használjuk, hogy bizonyos programozási feladatot megoldjunk. Ilyen például a hálózati forgalomirányítás, egy tudásadatbázis, mint például a Windows operációs rendszerek regisztrációs adatbázisa.

Most néhány szemléletes fogalmat vezetünk be ebből a körből.

Definíciók:

Gráfnak *nevezzük a csúcspontok és a csúcspontokat összekötő élek halmazát.*

Útvonal: Két kiválasztott pont közötti élek sorozata, amelyeken az egyik pontból eljutunk a másikba.

Irányított gráf: Ha az éleken csak egyik irányba lehet haladni.

Összefüggő egy gráf, ha minden két pontja között létezik összeköttetés. Nem összefüggő egy gráf, ha van olyan két pontja, ami között nem létezik összeköttetés.

Kör van a gráfban, ha van olyan két pontja, amelyet legalább két különböző útvonal köt össze.

Definíció:

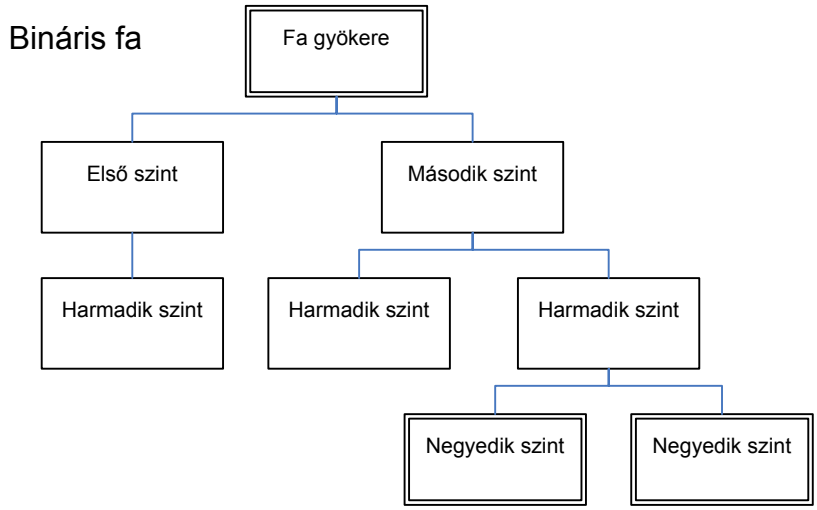
A fa olyan gráf, amelyikben nincsen kör.

A továbbiakban ezt a kérdéskört tekintjük át egy kicsit.

Definíció:
Bináris *fa az olyan fa, amelyiknek csúcspontjaiból maximum két útvonal indul el, azaz 0, 1 vagy 2***.**

4.5.5 Fa adattípus

A fákat olyan esetekben használjuk, ha az adatok között valamilyen alá és fölérendeltségi viszony jelenik meg. A fák legismertebb felhasználási területe a lemezek könyvtárszerkezete.



A fákban nincsenek hurkok. Két csúcspont között a fát csak egyértelműen lehet bejárni. A fák elemei az ábrán a dobozokban vannak. A fa gyökere egyúttal a hierarchiában legmagasabb helyen álló elem.

Bináris fák

A bináris fákat alapvetően rendezési feladatok megoldására lehet felhasználni.

Minimális magasságú fa

Olyan fa, amelyben az egyes ágak hosszai a lehető legrövidebbek. Az elvi határ N elem esetén $\log_2 N$ az ág hossza

Kiegyensúlyozott fa

Ha bármely elem esetén a bal és a jobb oldali részfák magassága legfeljebb egy elemmel tér el.

Tökéletesen kiegyensúlyozott fa

Ha minden részfájában elhelyezett elemek száma maximum eggyel tér el.

Fás rendezés, keresőfa, rendezőfa

Sorban vigyünk be adatokat egy bináris fába. Az első elemet helyezzük el. Ha a következő bevitt elem kisebb, mint az előző, akkor a gyökérelemtől balra helyezzük el, ha nagyobb vagy egyenlő, akkor jobbra. Az egymás után bevitt elemekkel járjuk be a fa ágait, és ha egy csomóponttal összehasonlítva az éppen bevitt elem kisebb, akkor tőle balra menjünk, ha nagyobb vagy egyenlő, akkor tőle jobbra menjünk.

Az így kialakult fában minden elemre igaz lesz, hogy a tőle balra elhelyezkedő elemek nála kisebbek lesznek, a jobbra elhelyezkedő elemek pedig nagyobb vagy egyenlők lesznek vele. Ennek megfelelően, ha balról jobbra bejárjuk a fát úgy, hogy mindig balra tartva lemegyünk a legmélyebb szintre, majd ha már nem tudunk tovább menni, kiírjuk az ott talált elemet. Ez lesz a legkisebb elem.

Egy szinttel visszalépve kiírjuk a következő elemet, ez lesz a nagyságrendi sorrendben a második, majd ha van jobb oldali ág, akkor oda lemegyünk. Ha a jobb oldali ágat is bejártuk, akkor egy szinttel visszalépve kiírjuk a fenti elemet, majd annak a jobb oldalát járjuk be. Nagyság szerint rendezett bejárást kapunk. Egy bináris fa egyfajta megvalósítása a következő lehet:

Típusdefiníció:

```
Fa =   Rekord
     Bal_mutato : egész típus
     Adat :valamilyen adattípus
     Jobb_mutato : egész típus
   Rekord vége
Legyen T[N], Fa típusú, N elemű tömb.
```

A `T[i].adat` tartalmazza az adatot, `T[i].Bal_mutato` jelenti a fában tőle balra elhelyezkedő elem, míg a `T[i].Jobb_mutato` jelenti a fában tőle balra elhelyezkedő elem pozícióját.

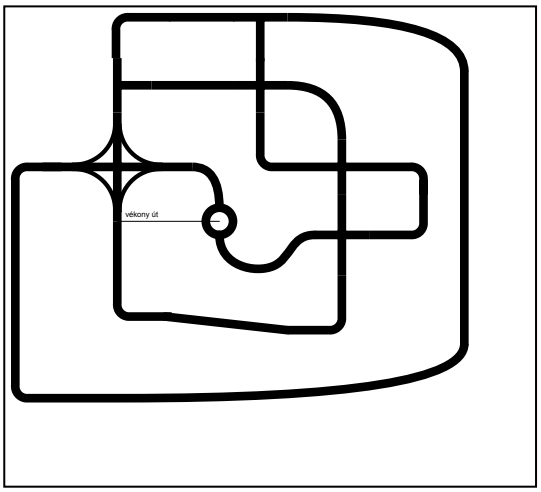
A fa feltöltése adatokkal.

```
Eljárás Feltoltes
   Be:T[1].adat
   T[1].Bal_mutato  := NIL
   T[1].Jobb_mutato := NIL
   Be:Uj_adat
   i:=1
   Ciklus amíg Uj_adat <> NIL és (I <= N )
      T[i].adat:=Uj_adat
      Uj_elem_Beszúrása(1,i)
      i := i + 1
      Be:Uj_adat
   Ciklus vége
Eljárás vége

Eljárás Új_elem_Beszúrása(p,i)
   Ha T[i].adat < T[p].adat akkor
      Ha T[p].Bal_mutato = NIL akkor
         T[i].Bal_mutato :=NIL
         T[i].Jobb_mutato :=NIL
         T[p].Bal_mutato := i
      Kulonben
         Új_Elem_Beszúrása(T[p].Bal_mutató,i)
      Elágazás vége
   Különben
      Ha T[p].Jobb_mutato = NIL akkor
         T[i].Bal_mutato :=NIL
         T[i].Jobb_mutato :=NIL
         T[p].Jobb_mutato := i
      Kulonben
         Új_Elem_Beszúrása(T[p].Jobb_mutató,i)
      Elágazás vége
   Elágazás vége
Eljárás vége
```

Természetesen az `Új_Elem_Beszúrása` függvény rekurzív, önmagát meghívó módon írható csak meg. Megjegyzendő, hogy N elem esetén a rekurzió mélysége jósolhatóan csak Log_2N lesz, amely 1024 elem esetén csak 10.

4.5.6 Gráf adattípus



A gráf adattípus az egyik legelvontabb, legnehezebben kezelhető, de mégis sok esetben szükséges adattípus. A gráfok szemléltetéséhez képzeljünk el egy országot, az országban lévő településeket, és a településeket összekötő utakat. Ezek így együtt gráfot alkotnak.

A települések az adott rendszerben a gráf csomópontjai, a településeket összekötő utak a gráf élei és az egyes utak hossza a gráf élének súlyozása.

Több kérdést lehet feltenni, amelyekre számítógépes programmal keressük a választ:

El lehet-e jutni az egyik városból a másikba?

Hány km a távolsága két tetszőlegesen kiválasztott városnak?

Ha a teherautónak X km-re elegendő üzemanyaga van, merre kell mennie, hogy egy adott városból eljusson egy másikba. Melyik a legrövidebb út két város között? Adott egy több várost érintő körút. Melyik az optimális útvonal…

Ezen kívül rengeteg hasonló kérdést lehet feltenni, de a megválaszolásuk túlmegy a jegyzet keretein, ezért csak néhány támpontot adunk itt a továbbiakhoz.

A gráfokat gyakran ábrázoljuk egy két-dimenziós tömbben **(csúcsmátrix g[n,m])** A tömbnek annyi oszlopa és sora van, ahány város van. Ha két csúcspont között van útvonal, akkor a megfelelő sor és oszlop által meghatározott helyen a távolságtól függő pozitív számot írunk, amely csúcspontok között nincsen út, ott nullát írunk. Természetesen a csúcspontoknak saját magukhoz nem vezet út, ezért az átlóban csupa nulla szerepel.

Tehát g[i,j] igaz, ha i-ből j-be vezet él, vagy g[i,j]=az i-ből j-be vezető él hossza (súlya), vagy ha nem vezet él, akkor g[i,j]=? vagy 0, ami épp praktikusabb.

Ha a gráf irányítatlan, g[i,j]=g[j,i]. g[i,i] értéke szintén változó lehet, az aktuális helyzettől függ.

	1. város	2. város	3. város	4. város	5. város	6. város	7. város
1. város	0	1	1	0	1	1	0
2. város	1	0	1	0	0	0	1
3. város	1	1	0	1	1	0	0
4. város	0	0	1	0	1	1	1
5. város	1	0	1	1	0	0	0
6. város	1	0	0	1	0	0	0
7. város	0	1	0	1	0	0	0

Ritka gráfokat (kevés benne az él) szomszédsági listával szoktak ábrázolni, ha pl. kevés a hely a csúcsmátrixra. Ilyenkor g[i] annak a (dinamikus) listának az első tagjára mutat, melynek elemei azoknak a pontoknak a sorszámai, melyekbe i-ből vezet él.

Léteznek **multigráfok**, melyekben két pont között több él is vezethet, ill. lehetnek olyan élek, melyek egy pontból ugyanabba a pontba mutatnak. Lehet, hogy g[i,j] értékét csak függvénnyel kaphatjuk meg, mindig menet közben számoljuk ki. Páros gráfokban nincsenek **izolált pontok** (melyekhez nem vezet él) és a pontokat két csoportra lehet osztani úgy, hogy egy csoporton belül bármely két pont között nem vezet él.

4.5.7 Gráfbejárás:

Egy kezdőpontból elindulunk, és bizonyos koncepció alapján elmegyünk ebből mindenféle pontokba. Színezzük be a gráf pontjait:

fehér az legyen, amelyikben még nem jártunk

fekete az, ahol már jártunk és az összes kivezető élt megvizsgáltuk

szürkék pedig azok a pontok, melyekbe már eljutottunk valamelyik feketéből, de még nem vizsgáltuk meg az összes kivezető élt.

Kezdetben mindegyik pont fehér, csak a kezdőpont szürke, a végén mindegyik (kezdőpontból elérhető pont) fekete lesz. Fekete pontból csak fekete pontba vagy szürkébe vezethet él. Ha egy szürke pontból megvizsgáltuk az összes kivezető élt (ha voltak fehérbe vezető élek, azokat a pontokat beszürkítettük), a szürke pontok közül kell választanunk egyet, amelyiket legközelebb vizsgálunk. Ha azt választjuk, amelyik legrégebben lett szürke, szélességi keresést hajtunk végre (a pontokat egy sorban (FIFO) tároljuk). Ha azt, amelyik legutóbb lett szürke, akkor ez a mélységi keresés (a használt adatszerkezet ilyenkor verem, LIFO).

4.5.8 Szélességi keresés

n pontból álló súlyozatlan gráfra működik. Ha tav[i]=?, akkor az i. pont színe fehér; ha tav[i]<? és az i. pont bent van a sorban, akkor a pont szürke; egyébként fekete. sorbol(i) kiveszi a sorból a legelső pontot és a sorszámát berakja i-be; a sorba(i) berakja a sor végére az i. pontot. A sor legegyszerűbben dinamikus listával valósítható meg. tav[i] az i. pont távolsága a kezdőponttól úgy, hogy az i. pontba szulo[i]-ből megyünk. inf jelenti ?-t.

```
Eljárás szélességi keresés(pont):
   uressor;
   tav[1..n]:=inf;
   szulo[pont]:=0;
   tav[pont]:=0;
   sorba(pont);
   ciklus amig nem_ures_a_sor)
     sorbol(pont);
     Ciklus i:=1 to n
     Ha (g[pont,i]) és (tav[i]=inf) akkor
        sorba(i);
        szulo[i]:=pont;
        tav[i]:=tav[pont]+1;
     elágazás vége
   Ciklus vége
Eljárás vége
```

A szélességi keresés megadja súlyozatlan gráfban a legrövidebb utat a kezdőpont és az abból elérhető összes pont között. Írd meg az utat_nyomtat(i) eljárást, ami kiírja az utat a kezdőpontból az i. pontba a tav és a szulo tömbök alapján!

4.5.8.1 Példák:

Szélességi keresés segítségével sok kérdésre választ lehet még adni:

legrövidebb út a-ból b-be

a-ba érkező összes legrövidebb út (a gráf transzponáltjának felhasználásával: minden élt megfordítunk, magyarul g[i,j]-ből g[j,i] lesz)

összefüggő -e a gráf?

páros -e a gráf?

gráf komponenseinek meghatározása

4.5.9 Mélységi keresés

n pontból álló súlyozatlan gráfra működik. Itt vermet használunk, az ebben lévő pontok a szürkék. Ha szulo[i]=? és volt[i]=0, akkor az i. pont fehér, nincs bent a veremben. Ha szulo[i]<? és volt[i]=0, akkor az i. pont szürke, bent van a veremben, de még nem vizsgáltuk meg a belőle kivezető összes élt. Ha szulo[i]<? és volt[i]=1 lesz, akkor az i. pont fekete, minden kivezető éle szürke pontba vezet, de csak akkor vesszük ki a veremből, ha az összes olyan pontot, amibe ezen a ponton keresztül jutottunk el, már kivettük a veremből.

A főprogram megvizsgálja az összes pontot, így biztos, hogy a végén minden pont fekete lesz. A verem_teto visszaadja a verem tetején lévő elem értékét, a verembol eljárás pedig kiveszi azt a veremből. Használunk

még egy másik vermet is, amibe az elhagyás sorrendjében kerülnek be a fekete pontok. Vagyis ha egy pontból megvizsgáltuk az összes kivezető élt, és az abból elérhető összes pontra is megtettük ugyanezt, akkor rakjuk be a pontot a másik verembe. Magyarul a veremben az i pont alatt lesz az összes olyan pont, amit később értünk el, mint i-t. Itt egy rekurzív algoritmust adunk meg.

```
Eljárás mk-bejar(pont):
  Ciklus i:=1 től n
    Ha (g[pont,i]) és (szulo[i]=inf) akkor
      szulo[i]:=pont;
      mk-bejar(i);
    elágazás vége
  Ciklus vége
  masik_verembe(pont);
Eljárás vége
```

4.5.10 További algoritmusok

4.5.11 Topologikus rendezés

Mélységi kereséssel tudunk egy irányított, körmentes gráfot topologikusan rendezni: ez a pontok egy olyan felsorolását jelenti, melyben minden i,j-re igaz, hogy a felsorolásban i megelőzi j-t, ha i-ből j-be vezet él. A másik verem használatával a veremben föntről lefele ilyen sorrendben lesznek bent a pontok.

Példa

Készítsünk gráfot, melynek pontjai a ruhadarabok, és ha egyikből másikba vezet él, az azt jelenti, hogy egyiket előbb kell fölvenni, mint másikat (mondjuk zoknit a cipő előtt)!

Ha erre végrehajtunk topologikus rendezést (vagyis mélységi keresést, és a pontokat fölírjuk fordított elhagyási sorrendben), akkor megkapunk egy lehetséges öltözködési sorrendet.

4.5.12 Egy pontból kiinduló leghosszabb utak

Irányított, körmentes (súlyozott vagy súlyozatlan) gráfban az egy pontból kiinduló leghosszabb utakat is a topologikus rendezés segítségével határozhatjuk meg legkönnyebben. A kiinduló pontra hívjuk meg az mk-bejar() függvényt, majd a másik verem felhasználásával megkeressük a leghosszabb utakat: az lhtav vektorban lesznek a legnagyobb távolságok a kezdőponttól, lhszulo[i] pedig az a pont lesz, melyből a leghosszabb út során i-be megyünk.

```
Eljatras lhu:
    lhtav[1..n]:=0;
    masik_verembol(pont); //ez lesz a kezdőpont, mert mk-bejar()-T ezzel hívtuk
                          // meg
    lhszulo[pont]:=0;
    Ciklus amig(nem_ures_a_masik_verem)
        Ciklus i:=1 től n
            Ha g[pont,i]<inf akkor
                Ha  lhtav[i]<lhtav[pont]+g[pont,i] akkor
                    lhtav[i]:=lhtav[pont]+g[pont,i];
                    lhszulo[i]:=pont;
                 Elágazás vége
             Elágazás vége
        Ciklus vége
        masik_verembol(pont);
     Ciklus vége
Eljárás vég
```

4.5.13 Legrövidebb utak súlyozott gráfban egy kezdőpontból

A legrövidebb utak súlyozott gráfban egy kezdőpontból probléma a szélességi kereséshez hasonló algoritmussal oldható meg. A szürke pontok közül ilyenkor azt kell választani, amelyiknek az eddig számított, kezdőponttól való távolsága a legkisebb. A leggyorsabb megoldás az, ha prioritásos sorban tároljuk a szürke pontokat, a tav tömb megfelelő értékei alapján rendezve, mindig a legkisebb ilyen értékűt tudjuk kiszedni a sorból. A prioritásos sor itt dinamikus listával van megvalósítva, de lehet máshogy is, a hivatalos megoldást,

az Algoritmusokban lehet olvasni. A prisorba(i) berakja a sorba az i. elemet, a tav[i]-nek megfelelő helyre. prisorrendez(i) az i. elemet a tav[i]-nek megfelelő helyre rakja át a prioritásos soron belül (olyankor használjuk, ha tav[i]-nak megváltozott az értéke).

```
Eljárás lrus(pont):
     tav[1..n]:=inf;
     szulo[1..n]:=0;
     tav[pont]:=0;
     prisorba(pont);
     Ciklus (nem_ures_prisor)
            prisorbol(pont);
            Ciklus i:=1 től n-ig
                Ha g[pont,i]<inf akkor
                     Ha tav[i]>tav[pont]+g[pont,i] akkor
                          Ha tav[i]=inf akkor
                                tav[i]:=tav[pont]+g[pont,i];
                                prisorba(i);
                          különben
                                tav[i]:=tav[pont]+g[pont,i];
                                prisorrendez(i);
                          Elágazás vége
                          szulo[i]:=pont;
                      Elágazás vége
                  Elágazás vége
            Ciklus vége
     Ciklus vége
Eljárás vége
```

4.5.14 Legrövidebb utak minden csúcspárra

Az eddigi gráfalgoritmusok mind mohó stratégia alapján működtek, ez végre egy dinamikus programozással készült megoldás. Súlyozatlan és súlyozott gráfra is működik. Ilyen feladat pl. gyár (Válogatóverseny 2002/7)

Ha g[i,j] azt jelenti, hogy van -e él i-ből j-be, ill. az él hossza i-ből j-be, akkor legyen o[i,j] az, hogy van-e út i-ből j-be, ill. a legrövidebb út hossza i-ből j-be. Ezt az eredmény tömböt dinamikus programozással kapjuk meg: v[i,j,k] legyen az, hogy van -e út i-ből j-be úgy, hogy a közbülsők közül mindegyik pont sorszáma kisebb vagy egyenlő, mint k (ill. a legrövidebb út hossza ugyanígy). Legyen még v[i,j,0]:=g[i,j], ez lesz a rekurzió kilépési feltétele. Látjuk? Három-változós dinamikus programozás! A függvény definíciója (ha k>0): v[i,j,k]=v[i,j,k-1] vagy (v[i,k,k-1] és v[k,j,k-1]) (logikai vagy, és) ill. v[i,j,k]=min(v[i,j,k-1],v[i,k,k-1]+v[k,j,k-1]).

Tehát vagy nincs bent az útban a k. pont (v[i,j,k-1]) vagy benne van, és ezen keresztül megy az út (v[i,k,k-1] és v[k,j,k-1] ill. v[i,k,k-1]+v[k,j,k-1]), ezek közül vesszük az egyiket ill. ezek közül választjuk a kisebbet.

A következő algoritmus g-ből kiszámítja o-t, de csak két tömböt használ, g lesz v[i,j,k-1], o pedig v[i,j,k] (k=1,2,3, ..., n).

```
Eljárás lrum:
  Ciklus k:=1 től n-ig
     Ciklus i:=1 től n-ig
        Ciklus j:=1 től n-ig
          o[i,j]:=min(g[i,j],g[i,k]+g[k,j]); //v agy o[i,j]:=g[i,j] or (g[i,k]
                                             // and g[k,j])}
           g:=o;
        Ciklus vége
     Ciklus vége
  Ciklus vége
Eljárás vége
```

Több algoritmus alapja a szélességi keresés. Itt egy sor segítségével fedezi fel a program a gráfot, és épít fel ez alapján egy fát. Kezdetben a kezdőpontot és-t szürkére színezi, majd a szürke csúcsok mindegyikének megkeresi a még fehér szomszédjait. Ezeket szürkére színezi, s felveszi egy sorba. Miután a szürke csúcs minden szomszédját meghatároztuk, a csúcsot feketére festjük. A kezdőcsúcs távolsága 0, minden más csúcs

távolsága végtelen. Mivel a fehér oldalra fehér betűket nem érdemes írni, a következő színeket alkalmazzuk az soron következő ábrákon:

4.5.15 Fájl adattípus

Definíció

***Fájloknak** hívjuk azokat az adattárolási egységeket, amelyekben a számítógép programok tárolják az általuk feldolgozandó adatokat. Fizikailag a háttértáron elhelyezkedő összefüggő adathalmazt tekinthetjük fájloknak.*

A lemezeken elhelyezkedő fájloknak van nevük, fizikai elhelyezkedésük a lemezen, méretük, attribútumaik és más és más rendszereken még sok egyéb jellemzőik is. Ha az operációs rendszerben futó program fel akarja dolgozni egy lemezen elhelyezkedő fájlban lévő adatokat, akkor azt először tudatnia kell az operációs rendszerrel egyértelműen, hogy melyik a kérdéses fájl, aminek hatására az operációs rendszer fizikailag megkeresi a fájlt, és az adatait hozzáférhetővé teszi a program számára.

Fájl megnyitása

A fájl megnyitása az a folyamat, amikor az operációs rendszer a felhasználói programok részére írhatóvá és/vagy olvashatóvá teszi a fájlokat.

A már megnyitott fájl esetén a feldolgozó programnak nincsen szüksége a fájl nevére, hiszen az operációs rendszer dolga a név (és esetleg az elérési útvonal) alapján a fájllal kapcsolatos fizikai teendők intézése, éppen ezért a megnyitáskor az operációs rendszer egy sorszámot ad a megnyitott fájlnak. Ezt a sorszámot a fájl **handlerének** (filepointer, filemutató stb...) szokás nevezni.

A továbbiakban a kezelő program minden fájllal kapcsolatos művelet esetén a handlerre hivatkozik. Ha egy fájlt megnyitottunk, akkor a feldolgozó programnak már mindegy, hogy a fájl fizikailag hol vagy hogyan helyezkedik el.

Logikailag a fájlokat úgy tekintjük, hogy a feldolgozó program parancsára az operációs rendszer megnyit egy adatátviteli csatornát a program és a külvilág között, majd ezen a csatornán keresztül közlekednek az adatok. A program számára a fájl egy megnyitott adatátviteli csatorna.

A fájl megnyitása egyenlő az adatátviteli csatorna megnyitásával. A fájlokat megnyithatjuk **csak olvasásra**. Ekkor az adatok áramlása csak a háttértárról a program felé lehetséges. A fájlokat megnyithatjuk **csak írásra** is. Ebben az esetben az adatok a programból a háttértár felé közlekednek. Vannak esetek, amikor **írásra és olvasásra** nyitjuk meg a fájlt, ekkor kétirányú adatátvitel történik.

Ha a fájlt írásra és olvasásra nyitjuk meg, akkor ismerni kell valamilyen módon azt a helyet, ahová éppen írhatunk, vagy ahonnan olvashatunk a fájlból. Ezt a helyet a fájl pointere mutatja meg. A fájl megnyitásakor a pointer a fájl első elemére mutat és minden olvasási és írási művelet után egy egységnyit mozdul előre automatikusan. A pointer értékét ki lehet olvasni, és át lehet állítani.

A megnyitott fájlok kezelésekor figyelnünk kell arra, hogy a fájlnak van-e valamilyen felismerhető belső szerkezete. Alapvetően háromféle fájltípust különböztethetünk meg,

Szekvenciális fájlok. A fájl adatai csak sorban, egymás után dolgozható fel. Ilyennek például a billentyűzetről bevitt karakterek sorozata is, vagy egy nyomtatóra elküldött adatok sorozata. Pascal nyelven ezt **Text** típusnak hívjuk.

Beszélhetünk **relatív elérésű** (**vagy direkt elérésű**) **fájl**okról is. Ilyenkor ismerjük a fájl adattárolási egységének méretét, és ennek a méretnek az egész számú többszöröseivel tudunk előre és hátra mozogni a fájlban, azaz a fájl bármelyik részét fel tudjuk dolgozni, bővíthetjük. Pascal nyelven ezt a fajta fájlt típusos fájlnak hívjuk és **File of típus** módon definiáljuk. (Adatbázisok esetén használhatók az ilyen fájlok)

A harmadik a **típusnélküli fájl**. Ekkor az adatok direkt elérésűek, de a fájlnak nincsen ismétlődő belső szerkezete. Ekkor az adattárolási egység általában byte. Ha a fájlnak mégis van valamilyen rejtett belső struktúrája, akkor a programozónak kell megírnia azt a kódot, amely feldolgozza fájlban található adatokat. (Például a BMP fájlok belső szerkezete is ilyen típus nélküli). Pascalban ezt a fájltípus egyszerűen **File**-ként definiáljuk.

A továbbiakban összefoglaljuk, hogy milyen műveletek lehetnek fájlokon.

	Pascal	C, C++
Minden fájltípus esetén a fájl megnyitása.	Reset	fopen
Fájl megnyitása írásra	Rewrite	fopen()
A file bezárása	Close()	close()
Olvasás a fájlból	Read(f,…), Readln(f,….)	getc(), fgets(), fread(), sscanf()
Írás fájlba	Write(f,…), Writeln(f,…)	fputc(), printf(), fputs(), fprintf(), fwrite()
Nagyobb adatblokk beolvasása fájlból	BlockRead()	fread()
Nagyobb adatblokk kiírása fájlba	BlockWrite()	fwrite()
Fájl pointer mozgatása	Seek()	fseek()

Természetesen nem adtunk kimerítő leírást minden műveletre.

Feladatok

Szekvenciális fájlok

Készítsük el bemeneti szövegfájl másolatát, amelyben minden szót csak pontosan egy szóköz választ el.

Egy szövegállományban bizonyos részeket % jelek közé tettünk. Készítsünk két kimeneti szövegfájlt. Az egyikben a szöveggel megjelölt részek legyenek, a másikban a jelöletlenek.

Készítsünk programot, amely egy bemeneti szövegfájlban kicseréli egy adott kifejezés minden előfordulását egy másikra, és a módosított fájlt írja ki a kimenetre.

Feladatok

Relatív elérésű fájlok

Készítsük el egy adatfájlt, amelyben gépkocsik adatait tároljuk. A további feladatok erre a fájlra vonatkoznak. Az alábbi feladatokat külön-külön programban is meg lehet valósítani, de célszerű egy menürendszerrel vezérelt program részévé tenni.

Készíts rutint, amely az adatfájlt feltölti adatokkal, módosíthatja a rekordokat, továbbá törli a rekordokat.

Készíts rutint, amely megkeresi egy adott mező, adott értékkel rendelkező elemét.

Készíts rutint, amely kiírja az adatfájl adatait táblázatos formában a képernyőre, és összegzi az ár jellegű mezők tartalmát.

Készíts rutint, amely az adattáblán böngészést engedélyez, a kurzormozgató billentyűk segítségével.

Készíts rutint, amely egy megadott tetszőleges szempont szerint fizikailag rendezi az adatokat. (nehéz!)

Készíts rutint, amely egy indexfájlban tárolja egy tetszőleges rendezési szempont szerint az adatok logikai sorrendjét

Típus nélküli fájlok

Írjunk programot, amely megjeleníti egy BMP fájl összes lényeges paraméterét a képernyőn! A file-ok szerkezetének leírását kérd el tanárodtól, vagy keresd meg az Interneten!

A wav fájlok szerkezete is megkereshető. Írj programot, amely a legfontosabb adataikat kiírja a képernyőre! A fájl szerkezetének megállapításához használd fel az iskola Internet kapcsolatát!

4.5.16 Objektum adattípus, osztályok

A programozás módszereinek fejlődési iránya olyan, hogy a programozást minden módon igyekszik a hétköznapi gondolkodáshoz közelíteni. Ennek a fejlődésnek az eredménye az objektumok megjelenése bizonyos programozási nyelvekben. Az objektumokat először a Smalltalk programozási nyelvben, majd a Turbo Pascalban, illetve később a C nyelv bővítéseként megvalósult C++ nyelvben vezették be. Az objektumok megjelenése hatott a programozás módszereire is. Manapság már majdnem minden modern programozási nyelv tartalmazza az objektum orientált programozás (= Object Oriented Programming => OOP) sok elemét. Vannak olyan nyelvek, ahol teljeskörűen, míg másokban csak részlegesen valósították meg az OOP elemeit. Objektum Orientált elemeket tartalmaz a fent felsorolt nyelveken kívül a Delphi, Visual Basic, C#, C++, PHP, Java, Javascript, hogy csak a néhány leggyakrabban használt rendszert említsem.

Objektumok a világban

A valós világban létező dolgok tulajdonságokkal jellemezhetők. A valós világ dolgait általánosságban az angol nyelv object-nek, azaz objektumoknak hívja. A programozásban a valós világ egyes dolgainak tehát objektumok felelnek meg. A programozásban az objektumokat tulajdonságaikkal (tulajdonság = property) jellemezzük. Ha egy objektum megváltozik, az azt jelenti, hogy egy vagy több tulajdonsága megváltozik.

Üzenetek az objektumok között

Mitől változik meg egy objektum állapota. A külvilágból ingerek érik – ezeket általánosságban üzeneteknek (üzenet= message) hívjuk – és az objektum az ingerekre reagál, kijelzi, törli, módosítja valamelyik tulajdonságát, állapotát, és üzenetet küldhet más objektumnak. Az objektumokban metódusoknak hívjuk az üzenetet küldő, fogadó és azokra reagáló eljárásokat.

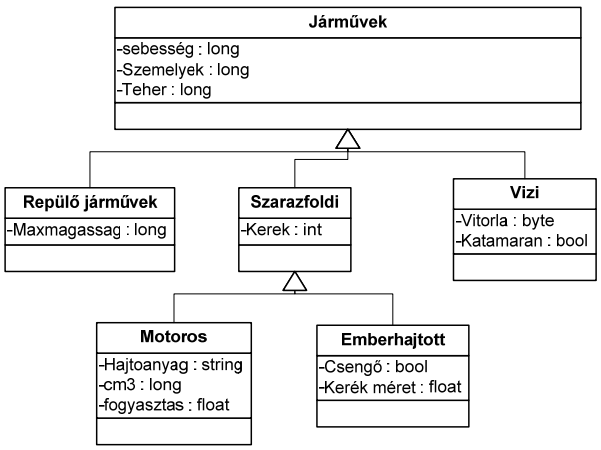
Osztályok az objektumok csoportjai

Az objektumokat tulajdonságaik alapján csoportokba, osztályokba sorolhatjuk. Például a járműveknek vannak olyan tulajdonságaik, amelyek alapján megalkothatjuk a járművek osztályát. Járműnek nevezzük azt, ami közlekedik, szállít, stb.

A járműveknek vannak olyan tulajdonságai, amelyek minden járműre igazak, például a szállítható személyek száma, a szállítható teher mennyisége, a maximális sebessége, stb.,

Ugyanakkor megalkothatjuk az osztályok specializált eseteit is, mint például a földön közlekedő járművek, a repülő járművek vagy a vízen közlekedő járművek.. Egyes fajtáknak ugyanakkor vannak speciális tulajdonságai, mint például a repülő járműveknek a maximális emelkedési magasság, a földön közlekedő járművek motoros, vagy nem motoros, stb.

Amikor egy osztályba tartozó objektumok egy csoportja egy vagy több plusz tulajdonság alapján megkülönböztethető az osztály többi tagjától, akkor e tagokat **részosztályba** sorolhatjuk, amelynek a tagjai minden tulajdonsággal rendelkeznek, amivel a fő osztály tagjai rendelkeznek, de további közös speciális tulajdonságaik is vannak. A részosztály esetenként tovább bontható részekre. A részekrebontást **specializációnak** hívjuk. Az alábbi ábra a járművek osztály specializációjának vázlatát mutatja:



A fő osztály a *Járművek* nevű osztály. A *Repülő járművek*, a S*zárazföldi* és a *Vízi* járművek a Járművek osztály leszármazottjai. A leszármazott osztályok minden tulajdonsággal rendelkeznek, amivel az ősük, de további tulajdonságok is érvényesek rájuk.

A valós világ modellezése

A programok a valós világot modellezik. Amikor a valós világot modellezni akarjuk, akkor érdemes a világ objektumaiból csoportokat alkotni közös jellemzőik alapján. Ha egy csoport egyes tagjainak vannak speciális tulajdonságai, akkor azokból érdemes egy leszármazott csoportot alkotni.

Definíció

*A programokban a valós világ objektumainak csoportjait **osztályok**nak (=class) nevezzük. Az osztály absztrakt fogalom. Olyan összetett adattípus, amelyet a programozó meglévő elemi vagy más összetett adattípusokból épít fel.*

*Az osztályban lévő összetevő adattípusokat az osztály **tulajdonságainak** hívjuk.*

*Az osztályok tartalmazzák azt a programkódot is, amelyek segítségével az osztály tagjai tudnak reagálni a külső környezetből érkező üzenetekre. Ez a programkód formailag függvényekből és eljárásokból áll. Az így definiált eljárásokat és függvényeket **metódusoknak** hívjuk.*

Röviden tehát az osztály mindig egy adattípust jelöl, a valóság objektumainak egy csoportját. Ha ennek az osztálynak egy konkrét objektumát akarjuk kezelni, akkor létre kell hozni az osztály egy példányát.

Definíció

***Példányosítás**nak nevezzük, amikor egy osztály konkrét előfordulását létrehozzuk. Az így létrejött programegységet **objektumnak** (=Object) hívjuk. A programokban ez egy változó, aminek típusa az osztály, amiből példányosítottuk. Az **objektumok** olyan zárt programozási egységek, amelyek az kezelni kívánt adatokon kívül tartalmazzák azokat az eljárásokat és függvényeket is, amelyek az objektumok megfelelő kezelésére képesek.*

Hogy egy programozási egységet objektumoknak tekintsünk az alábbiakban tárgyalt három tulajdonság megléte szükséges.

Egységbe zárás

Az objektumoknak azt a tulajdonságát, hogy az adatmezőkön kívül az őt kezelő eljárások vagy függvények is az adattípus részét képezik, **egységbezárásnak** (**encapsulation**) hívják. Az objektum adatait manipuláló függvényeket, eljárásokat **metódusoknak** hívjuk.

Speciális metódusok

Az objektumok speciális metódusai.

A **konstruktor** metódus akkor fut le, amikor egy objektumot, futás közben létrehozunk. A konstruktor biztosítja a helyet a memóriában az adatok és az egyéb kódok részére.

A **destruktor** metódus akkor fut le, amikor az objektum megszűnik. Annyi a feladata, hogy a megfelelő memóriaterületeket felszabadítja, a megnyitott fájlokat lezárja.

A fejlesztőrendszerek automatikusan létrehozzák az egyszerűbb objektumok konstruktorjának és destruktorjának kódját, a programozónak esetleg hivatkozni sem kell rájuk.

Öröklődés

Az objektumok általában tartalmazhatnak más objektumokat is. A tartalmazott objektumokat **ős objektumnak** szokás hívni, míg a tartalmazó objektumot **leszármazott objektumnak,** vagy gyereknek hívjuk. A leszármazott objektumok örökölik őseik tulajdonságait, azaz minden eljárást, metódust és adatmezőt, amivel azok rendelkeznek. Ennek megfelelően a leszármazott objektumban is használhatjuk az ős objektum metódusait.

Többrétűség (Polimorfizmus)

Ha egy leszármazott objektum metódusát ugyanolyan néven definiáljuk, mint egy ősének metódusát, akkor a program futási idejében fizikailag más eljárást kell használnia a rendszernek – az éppen használt objektumtól függően. Például a **rajzol()** nevű metódus nem lehet ugyanaz pont, vonal, kör és sokszög esetén. A polimorfizmus azt eredményezi, hogy a program a megfelelő metóduspéldányt választja ki futás közben a rendelkezésre álló ugyanolyan nevű szimbólumok közül.

Az objektum orientált programozás – az objektumok használata kissé más gondolkozást kíván meg a programozóktól, mint a hagyományos procedurális programozás. A program tervezése áttolódik az objektumok hierarchiájának átgondolt megtervezésére, továbbá az objektumok részeinek megfelelő kód létrehozására. A legtöbb fejlesztőrendszerben már létezik az objektumoknak egy olyan hierarchiája, amely a képernyőkezelésnél, a felhasználói felület kialakításánál elengedhetetlen.

Megjegyzés

> *Az objektum orientált programozás a Windows 3.1 elterjedésével vált népszerűvé, mivel ekkor nagymértékben eltolódott hangsúly a programok fejlesztése során a felhasználói felület felé, azt pedig egyszerű módon csak objektum orientált programozással lehetett fejleszteni. Általában a grafikus felületű programok eseményvezéreltek, ami szintén az OOP irányába viszi el a fejlesztést.*

Egy osztályt használni algoritmusleíró nyelven így lehet:

```
Vizi = Osztály kiterjesztve Jarmu
       Vitorla: Byte
       Katamaran: Bool
     Metodusok
       Indul (parameterlista)
       Megall(paramléterlista)
       Init() konstruktor
       Close() destruktor
  Osztaly vége

Változo : Vizi
```

Pascalban

```
Vizi = Object(TObject);
  Vitorla : byte;
  Katamaran: Boolean;
  Procedure indul(...)
  Function  Megall() : boolean;
  Constructor Init();
  Destructor close();
End;
```

PHP-ben

```
Class Vizi {
  Var Vitorla : byte;
  Var Katamaran: Boolean;
  Function indul(){...}
  Function  Megall(){...}
  Vizi();         //Ez itt a konstruktor, a neve megegyezik az osztály nevével
}                 //PHP-ban nincsen destruktor. Amikor a változó megszűnik,
                  //akkor automatikusan felszabadul minden erőforrás.
```

JAVA-ban

```
class Vizi {
  Var Vitorla : byte;
  Var Katamaran: Boolean;
  Public void indul(...)
  Public int megall(){...}
  public Vizi(){...}
}
```

Értéket adni egy objektum egy változójának így lehet:

```
Valtozo->mezo1 = Adat
```

vagy így

```
Valtozo.mezo1 = Adat
```

Egy objektum egy metódusát így hívhatjuk meg: Egy metódusra így lehet:

```
Valtozo->indul(paraméterek…);
```

Vagy így

```
Valtozo.megall();
```

A fenti példákból látszik, hogy a különböző nyelvi megvalósítások meglehetősen hasonlítanak egymáshoz. Ebből adódóan a szintaktika és annak értelme is hasonló. A különbségeket sajnos az adott programozási nyelv részletes tanulmányozásakor kell elsajátítani.

Formailag hasonlít a leírás a rekordok használatához, de itt a változóra való hivatkozás biztosítja, hogy már fordítási időben csak azokat az eljárásokat – metódusokat – hajtassuk végre a változón, amivel azt a változót lehet kezelni.

A jegyzet végén kitérünk az OOP-ra, részletesen megnézzük azokat a módszereket, amelyek segítségével hatékonyan lehet nagy méretű programokat készíteni az OOP felhasználásával.

Feladatok:

Építsük fel az alábbi objektumstruktúrát: kétdimenziós pont, egyenes szakasz, törtvonal, körív, kör! Az objektumokba vegyük bele a pontok, illetve egyéb alkotóelemek színét is!

5 Elemi algoritmusok, programozási tételek

A továbbiakban olyan algoritmusokat tárgyalunk meg, amelyek a programozás során rendszeresen előforduló feladatok megoldására kész választ adnak. Ezeket az algoritmusokat programozási tételeknek hívják. A programozási tételek azok az építőkövek, amelyek egy program létrehozása során minduntalan előfordulnak. A programozási tételek bizonyítható módon elvégzik a feladataikat, tehát ha a programozási tételeket helyesen használja a programozó, akkor helyesen működő programot tud írni.

Az alábbi linken megtalálható több programozási tétel és egyéb algoritmus demonstrációs programja, igaz többnyire angol nyelven. http://www.cs.bme.hu/~kiskat/sza/anim.html

Az elemi algoritmusok során használt közös jelölések

Minden programban egy vagy több A[N] jelű, N elemű vagy B[M] jelű, M elemű tömb tartalmazza a kiinduló tetszőleges típusú adatokat. Esetenként a C[K] jelű tömb K elemű tartalmazza az eredmény értékeket. T az elemeknek egy tulajdonsága, amit A[N].T-vel jelölünk. (Ezt a jelölést a rekordokból álló tömböktől vettük át. Egyszerűbb esetekben a T tulajdonság maga az elem értéke.)

A tömböket csak a tárgyalás egyszerűsége miatt alkalmaztuk, azok lehetnek azonos típusú rekordokból álló fájlok vagy listák is. Ennek megfelelően a programozási tételek megfogalmazhatók azonos hosszúságú rekordokból álló fájlokra és listákra is. Ilyen esetekben az alábbi műveleteket kell kicserélni az algoritmusokban:

Tömb	File	Lista
Egy tömbelem vizsgálata	Egy rekord beolvasása egy változóba, majd a megfelelő mező vizsgálata	A listaelem beolvasása és a megfelelő mező vizsgálata
Iteráció egy ciklusban	Lépés a file következő rekordjára	Lépés a következő listaelemre
A tömb végének vizsgálata	End Of File vizsgálata	A listamutató 0-e vizsgálat
Egy tömb elem értékének az átírása	Ugrás az adott sorszámú rekordra, majd írás a rekordba, írás, után a rekordmutatót eggyel visszaállítjuk	A Listaelem kiírása
tömbindex	rekordsorszám	Egy számláló típusú érték, a listafej esetén 0 és minden iteráció esetén növekedik eggyel.

5.1 Bonyolultság

Az algoritmus bonyolultságát nem a konkrétan végrehajtandó műveletek számával szokás jelölni, hanem N elem esetén az N-től való függéssel.

- O(1) : konstans futási idejű algoritmus jelent, azaz nem függ az elemszámtól az algoritmus
- O(N) : azt jelenti, hogy az elemszámmal lineárisan változik a futási idő
- $O(N^2)$: az elemszámmal négyzetesen változik a futási idő
- O(log N) : az elemszámmal logaritmikusan változik a futási idő
- O (N log N) : az elem számmal arányosan és annak logaritmusával változik a szükséges futási idő
- O(N!) : az elemszám faktoriálisával változik a futási idő

Elég nagy N esetén:

- $O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2) < O(N!)$

5.2 Sor, érték tételek

Olyan tételekről van szó, amelynél egy sorhoz egy értéket rendelünk hozzá. Azaz a kiinduló adatok A[N], N elemű tömbben vannak.

5.2.1 Összegzés tétel

Specifikáció

Adott egy A nevű N elemű tömb. A tömb elemei fel vannak töltve numerikus adatokkal. Írassuk ki az A[N] tömb elemeinek összegét.

```
Osszeg := 0
Ciklus i:=1-től N-ig
  Osszeg := Osszeg+A[i]
Ciklus vége
Ki: Osszeg
```

5.2.2 Átlagszámítás

Specifikáció
*Adott egy A nevű N elemű tömb. A tömb elemei fel vannak töltve numerikus adatokkal. Írassuk ki az A[N] tömb elemeinek átlagát***.**

Az átlagszámítás visszavezethető az összegzési tételre. Vigyázni kell arra, hogy míg a tömb elemei lehetnek egész típusúak, az átlag kiszámításánál vagy valós típust kell alkalmaznunk, vagy egész számok osztását kell használnunk.

```
Osszeg := 0
Atlag  := 0
Ciklus i:=1-től N-ig
  Osszeg := Osszeg + A[i]
Ciklus vége
Atlag : = Osszeg / N
Ki: Atlag
```

5.2.3 Eldöntés

Specifikáció

Adott egy A nevű N elemű tömb. A tömb elemei fel vannak töltve tetszőleges, de azonos típusú adatokkal. Döntsük el, hogy a tömbben létezik-e T tulajdonságú elem! A választ irassuk ki

A keresett elemet a K nevű változó tartalmazza majd, amit bekér a program. A keresett tulajdonság tehát: K A választ a `Letezik` nevű logikai típusú változó fogja tartalmazni.

```
Be: K
i:=1
Ciklus amíg NEM ( (i <= N) és ( A[i].T == K ) )
  i := i + 1
Ciklus vége
Letezik := (i <= N)
```

Itt két fontos elemet meg kell magyaráznunk. A keresési ciklus fejlécében két feltétel szerepel. Az első feltétel biztosítja, hogy a tömb vége után már ne keressünk, míg a második feltétel azt adja meg, hogy a keresett tulajdonság az aktuális elemre igaz-e.

Az utolsó sorral kapcsolatban azt jegyzem meg, hogy a legtöbb programozási nyelvben az értékadás jobb oldalán álló kifejezést értékeli ki először a program, és annak igazságértékét (igaz vagy hamis) adja értékül a bal oldalon álló változónak. Ha egy nyelv nem támogatja a fenti kifejezést, akkor a következő sorokkal lehet helyettesíteni az utolsó sort.

```
Ha i <= N akkor    Letezik := Igaz
          különben Létezik := Hamis
```

5.2.4 Keresések

5.2.4.1 Lineáris keresés

Specifikáció

Adott egy A nevű N elemű tömb. A tömb elemei fel vannak töltve tetszőleges, de azonos típusú adatokkal. Döntsük el, hogy a tömbben létezik-e T tulajdonságú elem! Ha létezik, akkor adja meg az elem sorszámát, különben adjon vissza egy lehetetlen sorszámot.

Ha létezik a keresett elem, akkor a `Sorszám` nevű változó a keresett elem indexét fogja tartalmazni, ha nincs adott tulajdonságú elem, akkor lehetetlen értéket fog a változó tartalmazni. A `Sorszám` változó kezdőértékét úgy kell megválasztani, hogy a tömb legkisebb indexű eleménél is kisebb legyen.

```
Be: K
Sorszam := -1
i:=1
Ciklus amíg NEM ((i <= N) és ( A[i].T == K ) )
   i := i + 1
Ciklus vége

Letezik := (i <= N)
Ha (i <= N ) akkor    Sorszam := i
             különben sorszam := Lehetetlen érték
```

Ha a fenti eljárást függvényként használjuk, akkor a függvény értékének a lekérdezésénél azt kell vizsgálni, hogy a Sorszámként visszaadott érték benne van-e a tömb értelmezési tartományában.

Hatékonyságvizsgálat

A lineáris keresés átlagosan N/2-ször fut le, mivel véletlenszerű elemek esetén és nagy számú keresést feltételezve átlagosan az N/2-ik lesz a keresett elem, a bonyolultság, tehát O(N). Ennél sokkal jobb eredményt ad a bináris keresés.

5.2.4.2 Bináris keresés

Specifikáció

N elemű tömb azonos típusú elemekkel van feltöltve. A tömb elemei T tulajdonság szerint növekvően rendezettek. Döntsük el, hogy a tömbben létezik-e T tulajdonságú elem! Ha létezik, akkor adja meg az elem sorszámát, különben adjon vissza egy lehetetlen sorszámot.

A keresési eljárás elve az, hogy megnézzük a tömb középső elemét. Ha megtaláltuk a keresett elemet, akkor befejeztük a keresést. Ha a középső elem T tulajdonsága kisebb, mint a keresett elem, akkor nyilván a felső fél tartományban kell keresni a keresett elemet, ha pedig kisebb, akkor az alsó fél tartományban kell keresni. Így minden egyes összehasonlítás során kizárjuk a maradék elemek felét.

```
Be: K
Also  := 1
Felso := N
Kozepso := ( Also + felso ) / 2

Ciklus amíg ( Also <= Felso ) és (A[Kozepso].T <> K )
Ha A[Kozepso].T > K Akkor
     Felso := Kozepso - 1
Különben
     Also  := Kozepso + 1
Elágazás vége
Kozepso := int(( Also + Felso ) / 2)
  Elágazás vége
Ciklus vége

Ha (A[Kozepso].T = K)   Akkor       Sorszam := Kozepso
            Különben    Sorszam :=  -1
```

A rutinban szereplő osztások természetesen az adott programozási nyelv egész típusú osztásának felelnek meg, és egész típusú eredményt kell visszaadniuk.

Hatékonyságvizsgálat:
$\log_2 N$ keresés alatt megtaláljuk az eredményt, illetve eldönthetjük, hogy az elem benne van-e a keresett elemek között. Például N =1024 esetén 10 összehasonlítás elegendő, mivel $\log_2 1024 = 10$. Ennek a keresésnek a hatékonysága sokkal jobb, mint a lineáris keresésé. Miért nem ezt a módszert használjuk mindig? Nem minden esetben biztosítható automatikusan, hogy a keresésre használt halmaz rendezett legyen, ha pedig nem rendezett, a rendezés időbe telik. A bonyolultság tehát $O(\log_2 N)$

5.2.5 Megszámlálás

Specifikáció:

Egy N elemű tömb A nevű tömb azonos típusú elemekkel van feltöltve. Határozzuk meg, hogy a tömbben hány db. T tulajdonságú elem van.

Itt az a kérdés, hogy az adott tulajdonság hányszor fordul elő, tehát definiálunk egy `Szamlalo` nevű egész értékeket felvevő változót, amelynek a kezdő értéke 0. Az fogja tartalmazni a kérdéses elemszámot. A korábbi tételekkel ellentétben itt nyilvánvalóan végig kell nézni a teljes tömböt, ezért nem `ciklus amíg`, hanem `ciklus …-tól …ig` típusú ciklust kell használni.

```
Be: K
Szamlalo := 0
Ciklus (i = 1-től N-ig)
  Ha A[i].T == K akkor Szamlalo := Szamlalo + 1
Ciklus vége
Ha (i <= N ) akkor Sorszam := i
```

5.2.6 Maximum kiválasztás (minimum kiválasztás) tétele

Specifikáció

Egy N elemű tömb A nevű tömb azonos típusú elemekkel van feltöltve. Keressük meg az A[N] tömb elemei közül a T tulajdonság szerinti legnagyobb elemet és a sorszámát valamint az értéket magát adjuk meg eredményül.

Az `Ertek` nevű változó tartalmazza majd a legnagyobb elem értékét, és a `Hely` mutatja meg a legnagyobb érték helyét.

```
Ertek := A[1]
Hely  := -1

Ciklus i = 1-től N-ig
Ha A[i].T > Ertek.T akkor
Ertek := A[i]
Hely  := i
  Elágazás vége
Ciklus vége

Ki: Ertek, Hely
```

Nyilván a legkisebb elemet (minimumkiválasztás) úgy tudjuk kiválasztani, hogy a relációs jelet megfordítjuk. Ha a tömb nullával kezdődik, akkor a ciklusunk is 0-val kezdődik, N-1-ig tart és a feltételezett legnagyobb elem is a 0. lesz. A fenti két algoritmus bonyolultsága O(N)

5.3 Sor, több érték

Az eddigi programozási tételek egy értéket szolgáltattak. A továbbiakban olyan tételek nézünk, amelyek több értéket adnak vissza. Ennek megfelelően az eredményeket is egy tömbben kapjuk vissza. A tömb néha ugyanaz, mint a bemeneti értékeket tartalmazó tömb, de néha másik tömb vagy tömbök is lehetnek.

5.3.1 Kiválogatás tétele

Specifikáció

Adott N elemű A nevű tömb. Az A tömb bemeneti értékei közül írassuk ki a K tulajdonsággal rendelkezőket.

```
Be: K
Ciklus (i = 1 -től N-ig)
  Ha A[i].T == K akkor Ki: A[i]
Ciklus vége
```

Nyilván a fenti algoritmus nem tudja visszaadni az összes értéket az őt meghívó eljárásnak, ezért ennek egy javítását fogjuk megnézni a következőkben.

5.3.2 Kiválogatás tétele módosítása

Specifikáció:

Adott N elemű A nevű tömb. Az A tömb bemeneti értékei közül másoljuk át a K tulajdonsággal rendelkező elemeket egy C[K], K elemű eredménytömbbe. Feltétel, hogy C[K] tömb elemszáma nagyobb vagy egyenlő legyen A[N] elemszámával, azaz K>=N. Az algoritmus végén C[j] az eredmény tömb utolsó értékes eleme.

```
Be: Elem
j := 0
Ciklus (i = 1 -től N-ig)
  Ha A[i].T == Elem.T akkor
    j := j+1
    C[j] := A[i]
  Elágazás vége
Ciklus vége
```

A fenti algoritmus magától értetődő.

5.3.3 Összefuttatás tétele

Specifikáció:

Adottak A[N] és B[M] tetszőleges adatokkal feltöltött, és T tulajdonság szerint rendezett tömbök.

Állítsuk elő azt a T tulajdonság szerint rendezett C[K], K elemű eredménytömböt, amelyben A és B elemei közül mindegyik annyiszor szerepel, ahányszor azok a forrástömbökben benne vannak.

Nyilván az eredménytömbnek K := M + N eleműnek kell lennie.

```
i := 1
j := 1
l := 0
Vege := Hamis
Ciklus amíg Nem Vege
   l := l + 1
   Ha A[i].T < B[j].T akkor
     C[l] := A[i]
     i := i + 1
     Ha i > N Akkor Vege := Igaz
   különben
     C[l] := A[j]
     j := j + 1
     Ha j > M Akkor Vege := Igaz
   Elágazás vége
Ciklus vége

Ha i > N Akkor
   Ciklus amíg j<= M
     C[l] := B[j]
     l := l + 1
     j := j + 1
   ciklus vege
különben
   Ciklus amíg i<= N
     C[l] := B[i]
     l := l + 1
     i := i + 1
   ciklus vege
Elágazás vége
```

Az algoritmus első része akkor ér véget, amikor az egyik tömb elemeiből kifogytunk, utána a másik tömb elemeit kell átmásolni az eredménytömbbe. Csúnya megoldás, hogy ott van a másoló ciklus. Ezt úgy küszöbölhetjük ki, hogy az A és B tömb utolsó eleme után beteszünk a két tömbbe, a legnagyobb elemnél nagyobb elemet.

A[N+1] elemű és B[M+1] elemű, ahol A[N+1] := ∞ és B[M+1] = ∞;

```
Eljárás Összefuttatás2()
  k:=0
  i:=1; j:=1
  Ciklus amíg i<N vagy j<M
    k:= k+1
    Elágazás
       A[i] < B[j] esetén  C[k]:= A[i]; i:=i+1
       A[i] = B[j] esetén C[k]:= A[i]; i:=i+1; j:=j+1
       A[i] > B[j] esetén C[k]:= B[j]; j:=j+1
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

5.3.4 Unió képzése

Adottak A[N] és B[M], N és M elemű tetszőleges adatokkal feltöltött tömbök

```
Eljárás Egyesítés():
  Z:=X; k:=N
  Ciklus j=1-től M-ig
    i:=1
    Ciklus amíg i≤N és B[j]≠A[i]
      i:=i+1
    Ciklus vége
    Ha i>N akkor k:=k+1; C[k]:=B[j]
  Ciklus vége
Eljárás vége.
```

5.3.5 Metszet képzése

Specifikáció:

Adottak A[N] és B[M], N és M elemű tetszőleges adatokkal feltöltött tömbök.

Első lehetőség: Állítsuk elő azt a T tulajdonság szerint rendezett C[K], K elemű eredménytömböt, amelyben azok az elemek szerepelnek, amelyek A-ban és B-ben is benne vannak.

Az eredménytömbnek K := Min(N,M) eleműnek kell lennie.

```
Konstans MaxN:Egész(???)
Típus THk=Tömb(1..MaxN:TH)
Eljárás Metszet():
  Db:=0
  Ciklus i=1-től N-ig
    j:=1
    Ciklus amíg j≤M és X(i)≠Y(j)
      j:=j+1
    Ciklus vége
    Ha j≤M akkor Db:=Db+1; Z(Db):=X(i)
  Ciklus vége
Eljárás vége.
```

5.3.6 Különbség képzése

Specifikáció:

Adottak X[N] és Y[M], N és M elemű tetszőleges adatokkal feltöltött tömbök.

Állítsuk elő azt a Z[] eredménytömböt, amelyben azok az elemek szerepelnek, amelyek A-ban benn vannak és B-ben nincsenek benne. Z elemszáma Max(N,M)lesz.

```
Eljárás Metszet():
  Db:=0
  Ciklus i=1-től N-ig
    j:=1
    Ciklus amíg j≤M és X(i)≠Y(j)
      j:+1
    Ciklus vége
    Ha j>M akkor Db:+1; Z(Db):=X(i)
  Ciklus vége
Eljárás vége.
```

5.4 Rendezések

A rendezési algoritmusok a programozás leggyakrabban használt eljárásai. A rendezések legfőbb szempontja, hogy a rendezés helyben történjen, azaz az eredménytömb megegyezik a bemenő adatok tömbjével. Ebből az okból kifolyóan bár az eredmény tömb, de mégsem sorolhatjuk be a Tömb -> Tömb típusú algoritmusok közé.

A rendezések hatékonyságának legfontosabb szempontja az, hogy hány összehasonlítás és hány csere történik a rendezés során. Bár a modern számítógépek néhány ezer érték esetén gyorsan el tudják végezni a

rendezéseket, azonban nem ritka a több százezer vagy több millió bemenő adat. Az ilyen adatmennyiségeknél nyilván a leggyorsabb számítógép is sok időt tölt el a rendezés során. Mindig a megfelelő rendezési eljárást kell alkalmazni. A rendezések során gyakran alkalmazzuk a Csere eljárást, amely kicseréli a két bemenő paramétert egymással.

A rendezési algoritmusok hatékonyságát meghatározza az, hogy hány összehasonlítást végez, és hány elem cseréjét végzi az algoritmus. A hatékonyság vizsgálata esetén a legjobb eset, amikor minden elem a helyén van és a legrosszabb eset, amikor pont ellentétes sorrendben vannak az elemek. A tipikus esetben véletlenszerű sorrendben vannak az elemek.

Az alábbi linken megtalálható egy sor rendezési algoritmus megvalósítása és animációja. A demonstrációkhoz általában JAVA VM-et kell telepíteni a számítógépre.

http://www.cs.bme.hu/~kiskat/sza/anim.html

5.4.1 Egyszerű csere

Az egyszerű csere nem a rendezésekhez tartozik, mégis nagyon fontos eljárás. Legyen `a` és `b` cím szerint átadott két paramétere a Csere nevű eljárásnak.

```
Eljárás Csere( &a, &b )
  c := a
  a := b
  b := c
Eljárás vége
```

5.4.2 Ciklikus permutáció

Ez az eljárás az egyszerű csere továbbvitele. Ez az eljárás nem szükséges a rendezésekhez, de érdekesen használható más esetekben. Egy A[N], N elemű tömb elemeit egy hellyel léptetni kell úgy, hogy az első elem az utolsó helyre kerül és a többi elem egy indexxel kisebb helyre kerül.

```
Eljárás Permutáció( A[N] )
  C := A[1]
  Ciklus i:= 1-től N-1 -ig
    A[i] := A[i+1]
  Ciklus vége
  A[N] := C
Eljárás vége
```

A fenti eljárásban érdemes megfigyelni, hogy a ciklus N-1 –ig megy, mivel ha N-ig menne, akkor az értékadásban az i = N esetben az i+1 túlmutatna a tömb határán.

Hasonló feladat, amikor egy tömb elemeit eggyel nagyobb indexű helyre kell léptetni. Ekkor az algoritmus így néz ki:

```
Eljárás Léptetés( A[N] )
  C := A[1]
  Ciklus i:= N-1-től 1 -ig
    A[i+1] := A[i]
  Ciklus vége
Eljárás vége
```

Az első helyen szereplő elemet menteni kell esetleges későbbi felhasználásra számítva.

5.4.3 Buborék rendezés

A buborékalgoritmus elve az, hogy az egymás utáni elemeket összehasonlítom. Ha a nagyobb értékű elem alacsonyabb indexű helyen van, mint a kisebb értékű, akkor kicserélem őket. Majd tovább folytatom az összehasonlítást, amíg a tömb végére nem érek. Ekkor a legnagyobb elem a legnagyobb sorszámú helyen lesz. A második menetben csak N-1 –ig megyek, stb.

```
Ciklus i := N –től 1 lefelé –1 -esével
  Ciklus j := 1-től i-1 -ig
    Ha A[j].T > A[j+1].T akkor Csere( A[j], A[j+1] )
  Cilus vége
Ciklus vége
```

Az eljárás hatékonysága: Legrosszabb esetben N*N/2 csere, legjobb esetben 0 csere kell, tehát átlagosan N*N/4 csere szükséges átlagosan. A probléma az, hogy túlságosan sokszor cseréljük az elemeket és túl sok összehasonlítást végzünk!

5.4.4 Minimum kiválasztásos (maximum kiválasztás) rendezés

A maximum kiválasztás elve az, hogy ha egy tömb 1..N eleme közül kiválasztjuk a legkisebbet, majd azt a legelső elem helyére tesszük, akkor a 2..N elem nála már csak nagyobb lehet. Ekkor a 2..N elemből is kiválasztjuk a legkisebbet, majd a 2. Helyre másoljuk. A végén az elemek növekvő sorrendben lesznek.

```
Ciklus i := 1 –től N-1 –ig
  Ciklus j := i+1-től N-ig
    Ha A[j].T < A[i].T akkor Csere( A[j], A[i] )
  Cilus vége
Ciklus vége
```

A rendezés hatékonysága hasonlít az buborékos rendezésre, mivel itt is sok összehasonlítás és sok csere van, N*(N-1)/2 csere szükséges átlagosan. Ha a kérdéses elemek összetettebb elemek, akkor a rengeteg csere nagyobb adatmozgatást eredményezhet. Ennek az adatmozgásnak a lecsökkentésére javítjuk az algoritmust.

```
Ciklus i := 1 –től N-1 –ig
  k := i
  Ciklus j := i+1-től N-ig
    Ha A[j].T < A[k].T akkor k:= j
  Cilus vége
  Csere( A[k], A[i] )
Ciklus vége
```

A javítás azon alapul, hogy a belső ciklusban nem cserélek, csak értékadást végzek. Mivel egy csere egy szubrutinhívásból és három értékadásból áll, ezért bonyolultabb adatszerkezet esetén ez az eljárás gyorsabb.

5.4.5 Beszúrásos rendezés

A módszer azon alapul, hogyha feltesszük, hogy egy tömb első i-1 eleme rendezett, akkor az i-ik elemet kiveszem a helyéről, majd megkeresem a helyét és beszúrom a megfelelő pontra. Ehhez természetesen feljebb kell tolni a nálánál nagyobb elemeket egy hellyel. Itt fontos, hogy az A tömbnek legyen 0. eleme is, amely nem tartalmaz értékes elemet, hanem biztosan minden elemnél kisebb értéket kell tartalmaznia.

```
Ciklus i := 2 –től N –ig
  Ha A[i].T < A[i-1].T akkor
    Ment :=A[i]
    j := i-1
    Ciklus amíg A[j].T => Ment.T
      A[j+1] := A[j]
      j := j-1
    Ciklus vége
    A[j] := Ment
  Elagazas vege
Ciklus vége
```

A fenti algoritmus gyorsabb az előző két rendezési módszernél, mert a belső ciklus nem fut tovább, ha megtalálta a beillesztendő elem helyét. Ezért az úgynevezett majdnem rendezett esetekben (amikor csak kevés elem van rossz helyen) lényegében O(N) a hatékonysága.

5.4.6 Shell rendezés

A shell rendezés elve az, hogy a rendezendő adatokat részlistákra bontjuk, amelyeket külön – külön rendezetté tesszük a beszúrásos rendezés segítségével, a részben rendezett elemekből nagyobb részlistákat alkotunk, amit szintén rendezetté teszünk. Legvégül a teljes adatsor lesz a lista, vagyis végső soron minden listát beszúrásos rendezéssel rendezünk.

Vegyünk egy N elemű tömböt és vegyük minden H-ik elemét. Ekkor lesz H db listánk, amelyben az elemek száma int(N/H). Mekkora legyen a H kezdő értéke és a többi értéke?

A rendezés akkor lesz gyors, ha minden rendezési menet nagy távolságokra mozgatja az adatokat, ezért célszerűen megfelelően nagy H értékkel kell kezdeni a rendezést, majd a H értékét folyamatosan csökkentve végül a H értéke 1 lesz. Az eredeti javaslat szerint H értékét 2 hatványaival határozták meg, vagyis mondjuk 10000 elem esetén H kezdő értéke 8192, majd 4096, 2048, ...32, 16, 8, 4, 2, 1.

A kutatások azt mutatták ki, hogy a kezdő elemszámtól függően más és más a legcélszerűbb H számsorozat. Egyes kutatók azt állítják, hogy ha a H értékeket az alábbi módon számoljuk ki

H = 3*H +1, ez megfelelően optimális lesz. Ennek alapján egy lehetséges H számsorozat az alábbi:

1, 4, 13, 40, 121, 364, 1093, stb...

Vagyis kiválasztjuk ennek a számsorozatnak a megfelelően nagy elemét, amellyel kezdjük a rendezést, majd ezt csökkentve hajtjuk végre a rendezést.

Legyen A[N] a rendezendő adat, és legyen az cols[16] a fenti számsorozatot csökkenő sorrendben tartalmazó 16 elemű (megfelelő méretű) tömb. Célszerűen a tömböt 0-val kezdődően indexeljük!

```
Eljárás shell ()
    cols = {1391376, 463792, 198768, 86961, 33936, 13776, 4592,
                    1968, 861, 1093, 364, 121, 40, 13, 4, 1}
    ciklus k=0-tól k=15-ig
        h= cols[k];
        ciklus i=h-tól n-1-ig
            v=a[i];
            j=i;
            ciklus amíg (j>=h && a[j-h]>v)
                a[j]=a[j-h];
                j=j-h;
            Ciklus vége
            a[j]=v;
        Ciklus vége
    Ciklus vége
Eljárás vége
```

5.4.7 Gyorsrendezés (quicksort)

A máig is leggyorsabb rendezési eljárást Hoare dolgozta ki 1962-ben. A keresztségben a Quicksort vagy Gyorsrendezés nevet kapta. Használatához rekurzív programozási nyelv szükséges, de szerencsére a legtöbb nyelv rendelkezik ezzel a képességgel. A felsorolt algoritmusok közül messze a leggyorsabb, de mivel mindennek van ára, nagy tömegű adatok esetén a program vermének nagynak kell lennie.

Az eljárás elve a következő. Vegyük a tömb számtanilag középső elemét. Balról kezdjük el megkeresni az első olyan elemet, amely nagyobb vagy egyenlő a középső elemmel, majd jobb oldalról kezdve keressük meg az első olyan elemet, amely kisebb vagy egyenlő a középső elemmel. Ha találtunk két ilyen elemet, akkor cseréljük ki őket.

A következőkben folytassuk a fenti tulajdonságú elemek keresését és cseréjét, amíg a bal és a jobb oldali elemek találatai össze nem érnek. Ha összeértek, az azt jelenti, hogy a középső elemtől balra csupa kisebb jobbra pedig csupa nagyobb elem áll.

Rekurzív hívással futtassuk le az algoritmust a bal oldali, majd a jobb oldali tartományra is. Amikor az eljárásból visszajön a program, akkor a megfelelő tartomány már rendezett.

Az eljárásban az A[N] tömböt adottnak tesszük fel, a Quicksort eljárás két index paramétert kap, ami az eljárásra nézve lokális változókat jelentettek.

```
Eljaras Quicksort(Bal, Jobb)
   I := Bal
   J := Jobb
   Kozepsokulcs := A[(i+j)/2].T

   Ciklus amíg i <= j
     Ciklus amíg A[i].K< Kozepsokulcs
       i := i + 1
     Ciklus vége
     Ciklus amíg A[j].K > Kozepsokulcs
       j := j - 1
     Ciklus vége
     Ha i<j akkor
       Csere( A[i] , A[j] )
       i:=i+1
       j:=j-1
     Elágazás vége
   Ciklus vége

   Ha Bal < j    akkor Quicksort(Bal, j-1)
   Ha i   < Jobb akkor Quicksort(i+1, Jobb)
Eljárás vége
```

5.4.8 Nem kötelező rendezési algoritmusok

5.4.8.1 Összefésüléses rendezés(Merge sort)

Ez a rendezési algoritmus rekurzión alapul. A lényege az, hogy rendezzük a tömb első felét, majd a második felét és végül a két részt összefésüljük. A rendezés maga rekurzív, tehát az első fél rendezése rekurzív módon zajlik le. A rekurzió véget ér, ha a rendezendő rész már csak egy elemből áll. Rosszabb a hatékonysága, mint a gyorsrendezésé!

5.4.8.2 Láda Rendezés (Bin sort)

Ez a rendezés csak meglehetősen speciális esetben használható. Adott A[N] tömb és M pozitív egész szám, amelyekre igaz, hogy minden i-re $0 \leq A[i] < M$ és A[i] pozitív egész szám! (látható, hogy meglehetősen szigorú feltételek vannak!) Hozzuk létre a kezdetben -1-gyel feltöltött Lada[M] tömböt.

A rendezés elve a következő. Végigmegyek A tömb elemein, és a Lada[A[i]] = A[i]-vel, majd a végén végigmenve a lada tömbön visszakapjuk az elemeket sorrendben. Ha egy elemből több is van, akkor az adott lada elembe egy

```
Ciklus i=0-tól M-1-ig
    Ciklus i=0 -tól N-ig
       Lada[A[i]] =A[i]
    Ciklus vége
Ciklus vége
Ciklus i=0-tól M-ig
  Ha M[i]>0 akkor Ki: M[i]
Ciklus vége
```

Az algoritmus javítása, ha egy elemből több is lehet, akkor minden láda legyen egy lista. És a ládákat tároljuk közös memóriaterületen!

```
Ciklus i=0-tól M-1-ig
    Ciklus i=0 –tól N-ig
       Listaba(A[i])
    Ciklus vége
Ciklus vége

Ciklus i=0-tól M-ig
  Ha Nem Ures(Lista[i]) akkor
     Ki: Listakiiratása(Lista[i])
   Elágazás vége
Ciklus vége
```

Ez a rendezés kissé nagyvonalúan gazdálkodik a memóriával. A listák kezelését korábban láttuk. Az algoritmus hatékonysága O(N+M). Kevesebb memóriával az alábbiakat lehet tenni:

```
Ciklus i=0-tól N-1-ig
    Ha A[i] <> i akkor csere(a[i],a[a[i])
Ciklus vége
```

Ebben az esetben nincsen nagy memóriafelhasználás, de a csere miatt 3* lassabb lesz az eljárás.

5.4.8.3 Radix rendezés

A Radix rendezés a Láda rendezés általánosítása. Legyen n egészünk, 0 és N^2 között, és ezeket kell rendeznünk.) A ládarendezéshez képest, ekkor $M = N^2$ és ekkor $O(M+N) = O(N^2)$. Két lépésben rendezünk.

1. lépésben N ládát használunk és az A[i] elemet az A[i mod N] –ik ládába tesszük.

2. lépésben végigmegyünk a ládákon és a ládákban lévő elemeket (amelyek listát alkothatnak) betesszük ugyanezen ládákba az alábbi módon: A[i] elemet az integer(A[i] /N)-ik ládába tesszük.

```
Ciklus i = 0 –tól N-1-ig
    Listába(lada[ A[i ] mod N], A[i] )
Ciklus vége

Ciklus i = 0-tól N-1-ig
    K = Listából(Lada[i] )
   Listába(Lada[ int(A[i] / N) ], A[i])
Ciklus vége
```

A Radix rendezés hatékonysága O(N log N). A hatékonyság levezetését hagyjuk az egyetemre ☺.

5.4.8.4 Láncrendezés

A feladat egy láncolt lista rendezése úgy, hogy csak a pointereket változtathatjuk meg, az adatmezőket nem.

5.4.8.5 Kupacrendezés (=Halom, heap sort)

Bináris fa – Olyan fa, amelyben minden pontnak van egy szülője (kivéve a gyökeret) és legfeljebb két fia.

Pont magassága – A leghosszabb, levélig vezető úton levő élek száma.

Fa magassága – A gyökér magassága.

Kupac – Egy olyan majdnem teljes bináris fa, amelyben minden elemre teljesül a következő: A[SZÜLŐ[i]] >= A[i].

Egy tömb és egy kupac közötti kapcsolat:
i - tömbindex
SZÜLŐ(i) = alsó egész rész(i/2)
BAL(i) = 2*i
JOBB(i) = 2*i+1

Példa a kupacra: 16,14,10,8,7,9,3,2,4,1

Bináris faként történő ábrázolása:

16

14 10

8 7 9 3

2 4 2

Ennek a fának a magassága 3. Minden (n hosszú, A) tömbre teljesül az, hogy A[n div 2 + 1, ..., n] kupac.

```
Eljárás Kupacrendezés (A)
    Kupacot_épít(A)
    ciklus i = hossz[A]-tól 2-ig visszafelé -1-vel do
        Csere(A[1],A[i])
        kupacméret[A]--
        Süllyeszt (A,1)
     Ciklus vége
Eljárás vége

Eljárás Kupacot_épít(A)
     kupacméret[A] = hossz[A]
     ciklus i = hossz[A]/2 –től 1-ig visszafelé -1-vel
        Süllyeszt(A,i)
     Ciklus vége
Eljárás vége

Eljárás Süllyeszt (A,i)
     l = BAL(i)
     r = JOBB(i)
     Ha l<=kupacméret[A] és A[l]>A[i] akkor
        legnagyobb=l
     különben
        legnagyobb=i
     Elágazás vége

     Ha r<=kupacméret[A] és A[r]>legnagyobb akkor
        legnagyobb = r
     Ekágazás vége

     Ha legnagyobb <>i akkor
        Csere(A[i],A[legnagyobb])
        Süllyeszt(A,legnagyobb)
     Elágazás vége
Eljárás vége
```

5.5 Programozási tételek alkalmazása

Az alábbiakban olyan feladatokat jelölünk meg, amelyben a programozási tételeket lehet alkalmazni.

Típus feladatok:

Egy repülő indul az egyik kontinensről a másikra és repülés közben rendszeresen méri az alatta lévő felszín tengerszint feletti magasságát. A mért érték nulla – ekkor tenger felett repül – vagy pozitív – ekkor szárazföld felett repül. Készítsünk olyan programot, a Top - Down módszer felhasználásával, amelyik a következőkre képes:

Szimulálja a méréseket véletlenszerűen, figyelve arra, hogy az első és az utolsó mérés szárazföld felett történt. Az eredményeket fájlba menti.

Grafikusan kirajzolja a felszínt, és elrepít felette egy kis repülőt (mérés közben vagy a mérések lezajlása után)

Kiírja a képernyőre és a fájlba is:

- Milyen távol van egymástól a két kontinens?
- Hol vannak a szigetek partjai (előtte tenger, utána szárazföld vagy fordítva)?
- Hány sziget van a két kontinens között?

- Hány hegycsúcsot talált (A hegycsúcs az a hely, ami előtt és mögött kisebb a tengerszint feletti magasság)?
- Át tud-e menni a két kontinens között egy kajakos, ha egyszerre csak egy adott távolságot tud evezni, mert ha többet evez, akkor elpusztul?
- Mekkora a szigetek átlagos távolsága?
- Van-e leszállópálya valamelyik szigeten (olyan rész, amely vízszintes legalább két mérés távolságig)
- Hány darab apró sziget van (maximum 3 méréshosszúságú)?
- Szeretünk alföldön élni. Van-e olyan rész, amely sík vidék, elég nagy és alföld? Keressük meg ezt a helyet!
- Adjuk meg a leghosszabb sziget kezdőpontját!

A fenti kérdésekre választ ad úgyis, hogy véletlen-szél gátolja, vagy segíti a repülőgép útját

> *Töltsünk fel adatokkal egy két-dimenziós tömböt! Írjunk programot, amely kiírja a legnagyobb elemet tartalmazó sor számát!*

Írjunk programot, amely névsorba rendezi egy osztály véletlenül beírt neveit!

Írjuk programot, amely a képernyő legfelső sorában lévő kiírás karaktereit ciklikusan permutálja. A kiírandó szöveg legyen hosszabb a képernyő szélességénél!

Hőmérsékletet mérünk a hét minden napján reggel, délben és este. Készítsünk programot, amely kiírja a reggeli, déli és az esti hőmérsékletek átlagát, kiírja a hét átlagosan leghidegebb napját, és meghatározza, hogy melyik napon a legmagasabb a hőmérséklet és mikor.

5.5.1 Numerikus algoritmusok

Állapítsuk meg két bekért szám legnagyobb közös osztóját

Állapítsuk meg két bekért szám legkisebb közös többszörösét

5.6 Szövegfile-ok kezelése

A szöveges fájlok kezelése a programozás területének egyik tipikus feladata. Miért is az? A szöveges fájlokon csakis sorrendben lehet végighaladni, tehát visszatérni nem lehet korábbi állapotokba. További problémát jelent az, hogy a szöveges fájl sorainak hossza tetszőleges, és a sor végét egy un. sorvége jel (CR/LF) zárja. A szöveges fájl hossza sem ismert általában, azt a fájlvége (EOF) jelöli.

Milyen általános megfontolásokat lehet ebben a tárgyban elmondani?

A szöveges fájlok típusfeladataiban általában meg kell nyitni egy vagy több szöveges fájlt, majd az eredményt ki kell írni másik fájlba, és/vagy a képernyőre is.

A szokásosan használt programozási nyelveken van egy fájlmegnyitás parancs. A fájl a háttértárak egyikén található fizikai fájl, a programunk azonban annak egy logikai megfelelőjét kezeli a memóriában. Ez azt jelenti, hogy egy speciális file típusú változót kell létrehoznunk a memóriában, és a fizikai fájl paramétereit hozzá kell rendelnünk ehhez a fájl változóhoz, majd a fájlt meg kell nyitni. Írásra, olvasásra, hozzáírásra tudunk megnyitni egy szöveges fájlt. Ha tudnánk a fájl egy-egy sorának hosszát, akkor képesek lennénk a fájlban előre és hátra lépegetni, de a szöveges fájlokban erre nincsen lehetőség a korábbiak miatt.

6 Rekurzió

Rekurziónak hívjuk azt a módszert, amikor egy értéket vagy egy állapotot úgy definiálunk, hogy definiáljuk a kezdőállapotát, majd általában egy állapotát az előző véges számú állapot segítségével határozzuk meg. Ez a fajta meghatározás gyakran rövidebb és jobban használható, mintha valamilyen zárt alakot használunk. A rekurzív programozásnál a programok **önmagukat hívják** meg és az aktuális állapotuk elmentésére **vermet** (stack) használnak. A rekurzív programok a feladat megoldását visszavezetik addig, amíg a megoldás triviális (kezdőérték), majd ebből állítják elő az általános értéket. Mivel a verem véges, ezért mindig biztosítani kell egy **végfeltételt**, amely biztosítja azt, hogy a rekurzió véget ér. Ha ez nem történik meg, akkor a rekurzív program a végtelenségig hívná magát, azonban a verem gyorsan megtelik, és hibával leáll a program.

6.1 A rekurzív eljárások, függvények

A programozási nyelvek általában biztosítják a programozók számára azt, hogy egyes eljárások önmagukat hívhassák meg. Az alábbi általános leírásban egy függvény kapcsán mutatjuk be a rekurziót.

```
Függvény Rekurziv(Bemenő paraméter)

Bemenő paraméter módosítása

Ha Feltetel(Bemeno parameterre) = igaz akkor
     Eredmény := Kezdőérték
Különben
   Eredmény := Rekurziv(Bemeno parameter)
Elágazás vége
Rekurziv := eredmény
Eljárás vége
```

A fenti eljárásban a rekurzív hívás az eljárásban végzett műveletek után helyezkedik el. Az ilyen rekurziót **jobb rekurziónak** hívjuk. Ha a rekurzív hívás először jön létre, majd később következnek a módosító műveletek, **bal rekurzióról** beszélünk.

A számítógépek eljáráshívási mechanizmusa úgy működik általában, hogy az eljárás meghívásakor a program a pillanatnyi futási címet verembe menti, illetve a vermen keresztül átadja a paramétereket is. A meghívott eljárás a veremből kiveszi a paramétereket és felhasználja, és az elmentett utasításcímet otthagyja.

Amikor vége szakad egy eljárásnak, akkor a visszatérési utasítás hatására kiveszi a veremből az előzőleg elmentett futási címet, majd ez alapján, a címen mutatott utasítás utáni utasításon folytatja a program végrehajtását.

A fenti eljárásban definiált minden változó lokális, ezért amikor az eljárás meghívja önmagát, ai változónak egy új „példánya” jön létre, függetlenül a többitől.

A feltételvizsgálat biztosítja, hogy egy bizonyos feltétel megléte esetén a rekurzió véget érjen. Ha rossz feltételt állítunk a rekurzióban, akkor előfordulhat, hogy végtelenül sokszor önmagát hívja meg a rekurzió, és a verem betelik. A program hibaüzenettel leáll. Nézzünk néhány egyszerűbb rekurzióval megoldható feladatot:

6.1.1 Fibonacci számok:

A Fibonacci számok sorozata olyan számsorozat, amelyben az i-edik elem az i-1 és az i-2-ik elem összegéből jön ki. Az F(0) := 1 és az F(1) :=1. Matematikailag: F(i) := F(i-1) + F(i-2) Készítsünk olyan rekurziót tartalmazó programot, amely megadja az F(N)-t.

```
Függvény Fibonacci(N)
   Ha N=0 vagy N=1 akkor
      Fibonacci := 1
   Különben
      Fibonacci := Fibonacci( i-1) + Fibonacci (i-2)
   Elágazás vége
Függvény vége
```

6.1.2 N alatt a K kiszámolása

A feladat egy matematikai definíciónak megfelelő érték kiszámolása. Ha van N db elemünk, és ki akarunk venni közüle K darabot, akkor N alatt a K féleképpen tudjuk kivenni, például a 90 db lottószámból hány féle módon tudunk 5 db-ot kiválasztani.

```
Függvény N_alatt_a_K(n, k)
  Ha k = 0 vagy k = n akkor
    N_alatt_a_K := 1
  Különben
    N_alatt_a_K := N_alatt_a_K(n-1,k-1)+ N_alatt_a_K(n-1,k)
  Elágazás vége
Függvény vége.
```

Ennek a feladatnak a megértéséhez ismerni kell a matematikában a Pascal háromszögnek nevezett fogalmat. Itt magyarázatot nem adunk a fogalomra, matematika tantárgyban a kombinatorika részen lehet ennek az elméleti alapjaival megismerkedni.

6.1.3 Hanoi torony

Van három pálcikánk, A,B,C jelű. Az A jelű pálcikán nagyság szerint csökkenő módon N darab korong van. Milyen sorrendben tudjuk átvinni a C jelű pálcikára a korongokat, úgy hogy szintén nagyság szerint csökkenő módon legyenek, ha csak egyesével mozgathatjuk őket, mindig egyik pálcikáról a másikra téve.

```
Eljárás Hanoi(N, A, C, B)
  Ha N>0 akkor
    Hanoi (N-1, A, B, C)
    Ki: N, mozgatás A-ról C-re
    Hanoi( N-1, B, C, A)
  Elágazás vége
Eljárás vége
```

A fenti eljárást úgy lehet ellenőrizni, ha először N=1 –re majd, N=2-re ellenőrizzük, azaz végigjátsszuk.

A fenti rekurzív algoritmusok mindegyike a matematikai gondolkodás alapján jól érthető, azonban a rekurzióknak alapvető hibája, hogy futás közben viszonylag sok helyre van szükség a veremben, továbbá a sok verem művelet miatt viszonylag lassúak az algoritmusok. A veremkezelő műveletek gépi szinten a leglassabb műveletek közé tartoznak.

6.1.4 Binomiális együttható előállítása

1. Rekurzív specifikáció és algoritmus:

$$Bin(N,K)=\begin{cases}1, & \text{ha } N=1\\ 1, & \text{ha } K=1\\ Bin(N,K-1)\cdot\dfrac{N-K+1}{K} & \text{egyébként}\end{cases}$$

```
Függvény Bin(N, K) : egész
  Ha (N = 1) vagy (K = 1)
  akkor
    Bin:=1
  különben
     Bin := Bin(N,K-1)· (N-K+1)/K
  Elágazás vége
Függvény vége
```

2. Rekurzív specifikáció és algoritmus a pascal háromszög elve alapján:

$$Bin(N,K)=\begin{cases}1, & \text{ha } N=1\\ 1, & \text{ha } K=1\\ Bin(N-1,K-1)+Bin(N-1,K) & \text{egyébként}\end{cases}$$

```
Függvény Bin(N, K) : egész
  Ha (N = 1) vagy (K = 1)
  akkor
    Bin:=1
  különben
    Bin := Bin(N-1, K-1) +
    Bin(N-1, K)
  Elágazás vége
Függvény vége
```

6.1.5 Backtrack algoritmus - Visszalépéses keresés

A megoldástér szisztematikus bejárása. Elindulunk egy irányba, feltételezve, hogy az a jó irány. Ha "zsákutcába" jutottunk, akkor visszalépegetünk addig a legközelebbi pontig, ahol tudunk más utat is választani. A megvalósítás technikája általában rekurzió, viszont ez a probléma minden esetben implementálható rekurzió nélkül is (ELSŐFIÚ, TESTVÉR).

Olyan feladat megoldására alkalmas, ahol **eredményül egy sorozatot kell kapni**. E sorozat minden egyes tagját egy-egy sorozatból kell kikeresni, de az egyes keresések összefüggnek egymással. Minden egyes új választás a korábbitól függhet (f_k függvény), a későbbiektől azonban nem. Egyes esetekben nem csak a korábbiaktól, hanem saját jellemzőjétől is függhet a választás (f_t függvény).

A megoldás legfelső szintjén keresünk az i. sorozatból megfelelő elemet. Ha ez sikerül, akkor lépünk tovább az i+1. sorozatra, ha pedig nem sikerült, akkor lépünk vissza az i-1. sorozatra, s abban keressük az újabb lehetséges elemet.

A keresés befejeződik, ha mindegyik sorozatból sikerült elemet választanunk (i>N) vagy pedig a visszalépések miatt már az elsőből sem tudunk újabbat választani (i<1).

Bemenet: $N \in \mathbf{N};\ M \in \mathbf{N}^N;\ \forall i \in [1..N]: S_i \in H^{M_i}$

$f_t : \mathbf{N} \times \mathbf{H} \to \mathbf{L}$ {Az adott sorozat indexelt eleme választható-e?}

$f_k : \mathbf{N} \times \mathbf{H} \times \mathbf{N} \times \mathbf{H} \to \mathbf{L}$ {Az egyik sorozat egy adott eleme összeegyeztethető-e a másik sorozat egy adott elemével?}

M(i) {Az i. sorozat elemszáma (minden egyes részsorozat lehet különböző hosszúságú)}

Kimenet: $Van \in \mathbf{L};\ X \in \mathbf{N}^N$

X(i) Az i. sorozatból az X(i). elemet választottuk a megoldásba

Előfeltétel: –

Utófeltétel: $Van \equiv \left(\exists X \in \mathbf{N}^N : \forall i \in [1..N]: \left(X(i) \in [1..M(i)]\right) \text{ és } \left(f_t(i,\ S_i(X_i))\right) \text{ és } \left(\forall j < i : f_k(i,\ S_i(X_i),\ j,\ S_j(X_j))\right) \right)$

Algoritmus:

```
Visszalépé ses keresés (N, M, X, Van)
  X( ) := 0
  i := 1
  Ciklus amíg (i > 0) és (i ≤ N)
     Jóesetvála sztás(i, Van, Melyik)
     Ha Van akkor
        X(i) := Melyik
        i := i + 1
     különben
        X(i) := 0
        i := i – 1
     Elágazás vége
  ciklus vége
  Van := (i ≥ N)
  Ha Van akkor Ki : X( )
Visszalépé ses keresés vége
```

ekkor van megoldás

Ha ez nem teljesül, akkor nincsen megoldás. Tehát, ha mindent kipróbálva visszaléptünk az első előtti

Innen kezdi legközelebb a keresést.

A Jóesetkeresés lényegében az S(i)-ben egy adott tulajdonságú elem lineáris keresése.

```
Eljárás Jóesetválasztás(i, Van, Melyik)
  j := X(i) + 1
  Ciklus amíg (j ≤ M(i)) és ((Rosszválasztás(i, j)) vagy (nem f_t(i, j)))
     j := j + 1
  ciklus vége
  Van := (j ≤ M(i))
  Ha Van akkor Melyik := j
Eljárás vége
```

Megmondja, hogy az i. sorozat j. eleme f_t tulajdonságú-e

Ha f_t nem teljesül, akkor már mindegy, hogy f_k teljesül-e, vagy nem.

A Rosszválasztás függvény általános esetben a feladattól függő f_k függvény kiszámolását jelenti.

```
Függvény Rosszválas ztás (i, j): logikai
     l := 1
   ciklus amíg (l < i) és (f_k(i, j, l))
      l := l + 1
   ciklus vége
   Rosszválas ztás := (l < i)
Rosszválas ztás vége
```

Megmondja, hogy az i. sorozat j. eleme összeegyeztethető-e az őt megelőző *l*. elemmel.

Kiválogatás visszalépéses kereséssel:

```
Eljárás(N, M, Db, Y)
    X( ) := 0
    i := 1
    Db := 0
    Ciklus amíg (i > 0)
        Ciklus amíg (i > 0) és (i ≤ N)
            Jóesetválasztás(i, Van, Melyik)
            Ha Van akkor
                X(i) := Melyik
                i := i + 1
            különben
                X(i) := 0
                i := i – 1
            Elágazás vége
        ciklus vége
        Ha i > N akkor
            Db := Db + 1
            Y(Db) := X( )
            i := i – 1
        Elágazás vége
    ciklus vége
Eljárás vége
```

A kiválogatás előfeltétele az, hogy létezik olyan sorozat, amely minden eleme megfelel az f_t és f_k függvényeknek.

Ha egy megoldást találunk, azt felírjuk Y-ba, és visszalépünk, mintha ez nem is lenne megoldás.

Maximumkiválasztás visszalépéses kereséssel:

```
Eljárás(N, M, Y)
    X( ) := 0
    Y( ) := −∞
    i := 1
    Ciklus amíg (i > 0)
        Ciklus amíg (i > 0) és (i ≤ N)
            Jóesetválasztás(i, Van, Melyik)
            Ha Van akkor
                X(i) := Melyik
                i := i + 1
            különben
                X(i) := 0
                i := i − 1
            Elágazás vége
        ciklus vége
        Ha i > N akkor
            i := i − 1
            Ha X > Y akkor
                Y := X
            Elágazás vége
        Elágazás vége
    ciklus vége
Eljárás vége
```

A kiválogatás tételének speciális esete, amikor valamilyen szempontból a legjobb megoldást kell megadnunk.

Ha olyan megoldást találunk, amely jobb az eddigi legjobbnál (ami az Y), akkor azt felírjuk Y-ba, és visszalépünk, hátha találunk jobbat.

Feladatok

8 vezér probléma - Hányféleképpen lehet egy 8x8-as sakktáblán 8 vezért elhelyezni úgy, hogy azok ne üssék egymást? (Általánosabban megfogalmazva: NxN-es sakktábla, N darab vezér.)

Tic-Tac-Toe - A feladat az, hogy generáljuk le az összes lehetséges Tic-Tac-Toe játszmát. Készítsünk statisztikát arról, hogy hány végződik a kezdő győzelmével stb.

Sakk-huszár - Keressünk egy olyan huszárugrás-sorozatot, amely minden egyes pontot érint a sakktáblán pontosan egyszer, és ugyan oda tér vissza, ahonnan indult.

Labirintus - Adva van egy M*N-es labirintus, (1,1) a bejárat, (M,N) a kijárat. Keressünk egy utat a bejárattól a kijáratig.

Aknakereső - Adva van egy NxM-es tábla (1<=N,M<=15). A tábla minden egyes pontján adva van az, hogy körülötte hány akna található. A feladat egy lehetséges aknatérkép meghatározása.

```
Pl.
4 6
232321

2 4 3 2 4 3
3 3 3 3 5 2
1 1 2 1 3 2

Megoldása:

1 1 1 0 1 1
1 0 0 1 0 0
0 0 0 0 0 1
1 1 0 1 1 1
```

(Sorba bejárjuk a pontokat. Először azt feltételezzük, hogy ott akna van. Amint ellentmondásra jutunk, visszalépünk, és a megfelelő helyen átváltunk "nincs akná"-ra.)

Mágikus lámpa - Adott MxN darab, mátrixszerűen elhelyezett lámpa. Alapból mindegyik lámpa ki van kapcsolva. A feladat az, hogy midegyik lámpát bekapcsoljuk. Ha az (i,j) lámpának a kapcsolóját átkapcsoljuk, akkor az (i-1,j), az (i,j), az (i,j-1), az (i,j+1) és az (i+1,j) lámpa is invertálódik.

6.2 Rekurzió és a ciklusok

Kérdés az, hogy van-e olyan eset, amikor érdemes rekurziót használni, ciklusok helyett, illetve lehet-e rekurzív algoritmusokat nem rekurzívvá átalakítani. Először megmutatjuk, hogyan kell átírni ciklusokat tartalmazó eljárást rekurzívvá:

Elöltesztelő ciklus esetén

```
Eljárás Cikl(x)
  Ciklus amíg Felt(x) = igaz

    Vegreh(x)
  Ciklus vége
Eljárás vége
```

```
Eljárás Rek(x)

      Ha Felt(x) akkor
            Vegreh(x)
            Rek(x)
      Elágazás vége
Eljárás vége
```

Hátultesztelő ciklus esetén

```
Eljárás Cikl(x)
  Ciklus
    Vegreh(x)
    amíg Felt(x) = igaz
  Ciklus vége
Eljárás vége
```

```
Eljárás Rek(x)
      Vegreh(x)
      Ha Felt(x)= igaz akkor
            Rek(x)
      Elágazás vége
Eljárás vége
```

Az utolsó esetben egy megszámlálásos ciklust írunk át rekurzívvá. A ciklus előtt adunk kezdőértéket egy változónak és a ciklusban is módosítjuk a változó tartalmát a korábbi értéke és egy tömb aktuális értéke alapján. (Például bármilyen összegzés elvégezhető így.)

```
Függvény R(bemenő paraméterek)
  S:=0
  Ciklus i:=1- től N-ig
    S:= fn(S, A[i])
  Ciklus vége
  R := S
Függvény vége
```

A fenti függvényben az Fn() tetszőleges összegző típusú függvényt jelent. A rekurzív változat így néz ki:

```
Függvény R ( bemenő paraméterek, i)
  Ha i>N akkor
R:= kezdőérték
  Különben
    R := Fn(A[i], R( paraméterek, i+1) )
  Elágazás vége
Függvény vége
```

6.3 Rekurzív adatszerkezetek

Itt visszatérünk az adatszerkezetek témára. A korábbi tanulmányaink alapján megismert adatszerkezetek közül a lista, a bináris fa, adatszerkezetek rekurzióval is definiálhatók.

A lista definíciója:

I:=0 esetén a lista üres,

I>0 esetén a lista az új elem + az addigi listából jön létre.

A bináris fa definíciója:

I:=0 esetén a bináris fa üres,

I>0 esetén a bináris fa := Bal oldali bináris fa + új elem + Jobb oldali bináris fa.

Ha mutató típusú változóval oldjuk meg a következő elem elérését, illetve a listaelem helyét a memóriában is mutató adja meg, akkor a Lista^.érték jelenti a listaelem értékét, illetve a LISTA^.mutató a következő listaelemre mutat. Ebben az esetben a lista bejárása a következő rekurzióval oldható meg:

```
Eljárás Bejárás (Lista)
  Ha Lista<>NIL akkor
    Ki: Lista^.érték
    Bejárás(Lista^.mutató)
  Elágazás vége
Eljárás vége
```

Ha a bináris fát a listával analóg módon a következőképpen jelöljük: Fa^.érték jelenti az aktuális elem értékére mutató pointer, Fa^.Bal és a Fa^.Jobb pedig a bal részfa illetve a jobb részfára mutató pointer, akkor a fa bejárás a következő:

```
Eljárás Balkozepjobb (Fa)
  Ha Fa<>NIL akkor
    Balkozepjobb(Fa^.Bal)
Ki: Fa^.érték
    Balkozepjobb(Fa^.Jobb)
  Elágazás vége
Eljárás vége
```

Feladatok

Valósítsuk meg a LOGO-ból ismert geometriai, ismétlődő mintákat az PASCAL, C vagy algoritmusleíró nyelven!

6.4 Mikor használjunk, és mikor ne használjunk rekurzív algoritmusokat?

A rekurzió mellett szólnak azok az esetek, amikor a feladat megfogalmazása rekurzív.

Korábban láttuk a Fibonacci számokat. Matematikai elvek, számsorozatok, sorozatok definiálása gyakran rekurzív, ekkor célszerű használni rekurziót.

Ha egy feladatban felismerjük, hogy egy rész megoldása lényegében ugyanaz, mint az egész megoldása (Quicksort algoritmus)

Mikor ne használjunk rekurziót? A rekurzióval szembeni szokásos kifogások:

A rekurzió a ciklusnál bonyolultabb programszerkezet, s mindketten ugyanarra a célra szolgálnak: valamilyen tevékenységek ismételt végrehajtására

Mivel a számítógépek nem tartalmaznak rekurzív lehetőségeket, ezért amit számítógéppel meg lehet oldani, azt rekurzió nélkül is meg lehet oldani

- Nagy központi tárigénnyel járhat
- Nagymértékű futási időnövekedést okoz

7 Bevezető, avagy mért kell módszeresen programozni?

Hogyan kell nagyobb lélegzetű programozási feladatokat megoldani?

A programozás folyamata nem nyelv-specifikus, mindig ugyanazokon az elveken alapul. A programozás alapvetően egyfajta gondolkodási, algoritmizálási, feladat-megoldási eljárás. A programozás alkotó munka.

A programozás módszerének története annak a története, hogy az egyszerű pár soros alkalmazások írásától hogyan jutott el a programozók társadalma a manapság már létező összetett rendszerekig és hogyan lép tovább talán a közeljövőben akár a mesterséges intelligenciáig.

7.1 A monolitikus programozás

A programozás tárgyalásánál nem kezdhetünk úgy egy könyvet, „hogy már az ókori görögök is ....", mivel az ókori görögöknek nem volt még számítógépük, de igaz ma már történelem, de az 1950-es évektől kezdve azért voltak már ilyen gépek, voltak programozók és írtak programokat is.

Az időszak mai szemmel egyszerű programjai más technikával készültek, mint a maiak, sokkal kevésbé voltak összetettek, mint a maiak. Ezek a programok általában egy-egy programozó kezétől születtek. Azt lehetett mondani, hogy

egy program – egy programozó.

Az ellátandó feladatok nem voltak túl bonyolultak, az egy programozó a megoldások algoritmusait átlátta, a programokat lekódolta, a programok általában lineáris felépítésűek voltak, egy-egy elágazással és egy-egy ciklussal. A programokat viszonylag könnyen és gyorsan meg lehetett írni. A programoknak nem volt belső struktúrájuk. Ezek voltak a monolitikus programok és a programozási módszert **monolitikus programozásnak** hívjuk.

7.2 A kezdő programozó - frontális támadás módszere

A kezdő programozó miután megírt több apró – egyenként néhány tucat soros - programot, gyakran azt hiszi, hogy akkor néhány ezer soros programok a fejlesztése is ugyanaz lesz, mint a rövidebbeké, csak tovább tart. Ő a monolitikus programozás időszakánál tart.

A kezdő programozó a programozás tanulása során felcsipegeti a tudásmorzsákat, a trükköket, a megszakítási címeket és a spéci megoldásokat. Esetleg optimalizálási megoldásokról is hall, sőt dokumentálásról is olvasott.

Egy új, esetleg összetettebb feladat megoldásához is úgy viszonyul, mint a korábban megírt rövidebb programokhoz.

Nagy vonalakban átgondolja az adatszerkezetet, a megoldási módszert, amit a feladat elején még nem lát. Arra majd időben kitalál valamilyen megoldást – gondolja magában. Rövid tervezgetés után, előveszi a szeretett és jól megtanult programozási nyelvet (4GL vagy egyéb fejlesztő eszközt) és elkezdi írni a programot, az első szótól folyamatosan haladva az utolsó felé. Természetesen, mivel a program hosszabb, mint szokott, ezért gyorsan újabb és újabb eljárásokat talál ki a feladatok megoldására, újabbnál újabb alacsonyszintű rendszerkezelő programrészletet dolgoz ki. Lehetőleg a program minden részével foglalkozik – frontális támadás módszere -, mivel csupa globális változót használ, és az esetlegesen meglévő programmodulok ezeken a változókon keresztül kommunikálnak. A program írása az utolsó sor utáni END.(csukó kapcsos zárójel vagy RETURN, stb.…) szavakkal végződik.

Mivel az így megírt program általában nem működik, a fejlesztő mágus hamarosan megunja az állandó javításokat. A programot javítani senki sem tudja, még ő sem. A javítások újabb problémákat vetnek fel.

Viszonylag gyorsan lehet kis programokat fejleszteni, de a programot már az első pillanattól a fejében kell tartania fejlesztőnek.

7.3 A moduláris programozás

Köztudott, hogy egy ember egy bizonyos bonyolultság után a teendőket nem tudja átlátni, csak ösztöneire, megérzéseire hagyatkozhat. A programokat ösztönből azonban nem lehet megírni.

A moduláris programozás azt jelenti, hogy a problémát olyan részfeladatokra bontjuk, amelyeknek a bonyolultsága már nem okoz gondot, amit már egy modulban – monolitikusan meg tud írni egy programozó, azaz csökkentjük a probléma bonyolultságát.

Ha több ember együtt dolgozik egy munkán, akkor az elvégzendő feladatot szintén részekre kell bontani, és a részeknek az összekapcsolását, összekapcsolódását meg kell tervezni. Itt is a megfelelő megoldás az, hogy a programokat bontsuk modulokra. A részek közötti együttműködési felületet interface-nek hívjuk, a programozási módszert **moduláris programozásnak**.

A továbbiakban felmerül a kérdés, hogy milyen elvek alapján, hogyan bontsuk részekre a programot?

7.4 Top - Down dekompozíciós módszer

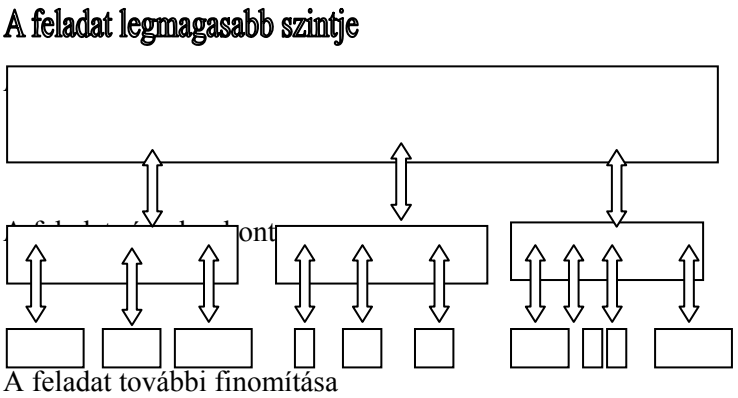
A Top – Down módszer abból indul ki, hogy a feladatot nagyobb viszonylag egymástól független egységekre bontja. Ezek az egységek csak **jól definiált interface**-eken kommunikálnak egymással, más kapcsolat az egyes modulok között nem lehet.

Az egyes részek fejlesztői kívülről csak dokumentált definíciókat használhatnak fel.

Ha egy részleg a fejlesztés során rájön, hogy a mások által is használt külső paraméterek nem illeszkednek megfelelően az általa megoldandó feladathoz, akkor két dolgot tehet:

Saját hatáskörében kidolgozza a külső kapcsolatok olyan belső értelmezését, amely már a céljainak megfelel, de kifelé csak a szabványos felületen érintkezik a többi modullal.

A többi modul fejlesztőjével értekezve javasol egy olyan megoldást a közösen használt külső paraméterekre, amelyek mindenki számára megfelelőek.



A feladat további finomítása

Mind a két módszernek vannak előnyei és hátrányai. Az első esetben a kérdéses modul megvalósítása az optimálisnál bonyolultabb lehet, következésképpen lassabb, esetleg a szükségesnél nagyobb erőforrást lefoglaló. Ebben az esetben a belső modul és a külső között lennie kell egy konvertáló modulnak is, amely, amely a belső, megváltoztatott szabványok által adott eredményeket átfordítja a külső kapcsolatok által megkívánt formára. További problémák forrása lehet, ha a belső módosítás esetleg csak részben teljesíti a többi modul által megkívánt feltételeket. Ha a többi modul kíván változtatásokat a közös paramétereken, akkor a belső nem egyeztetett megvalósítás akadálya lehet a közös, módosított paraméter rendszernek.

Előnyére válik a megoldásnak, hogy nem kíván egyeztetést a többi modul megvalósítójával, belső ügy maradhat, a módosítás a többieket nem feltétlenül érinti.

A második módszer mindenképpen szimpatikusabb, ugyanis a módosítások közkinccsé válása során az esetleges más által kezdeményezett módosítások is átgondoltabbá válnak. Az egyeztetés néha persze lehetetlen és mindenesetre több átgondolást kíván, de akkor később kisebb a feltételütközések lehetősége, az egyes modulok kapcsolódási pontjai továbbra is szabványosak maradhatnak, a teljesítmény az elvárható optimális közelében mozog.

A Top-Down módszer lényege, hogy a megfogalmazott **részfeladatokat további részfeladatokra osztjuk**, amelyeket további részfeladatokra és így tovább. A megoldandó problémát ésszerű határig bontjuk egyre alacsonyabb szintű.

Meddig folytassuk a részekre bontást?

Általános recept nincsen, addig, amíg a modult már programozói szinten átlátjuk. Előfordulhat, hogy egyes modulokat lebontunk, míg más modulokat készen kapunk kidolgozva. A részekre bontás folyamatában figyelni kell az egyensúlyra. Törekedni kell arra, hogy a megoldandó feladat minden részén egyenletesen haladjunk előre a részekre bontás folyamatában. Ezt az **„párhuzamos finomítás" elvének** hívjuk. Az elv alkalmazásánál a finomítással együtt finomodik az adatstruktúra is. Tehát az adatmodell és az eljárásmodell együtt közeledik a végső cél felé.

Hogyan lehet kódolásnál a módszert alkalmazni?

Az elkészülő program vázát építjük fel, a bejelentkező menüt írjuk meg, a fő funkciókat megjelenítjük a képernyőn. Ha például a program elején megjelenik egy menü, akkor megírjuk azokat a modulokat, amelyek a legfontosabbak, pl. fájl megnyitás, keresés, bezárás, a többi modult beírjuk a programba, de csak „üres" eljárásként, azaz a nevét adjuk meg és az eljárás átveszi az esetleges paramétereket. Ha szükséges, akkor előre megadott teszteredményeket ad vissza.

A kódolásnál is figyelni kell arra, hogy a modulok megírását is lehetőleg egyenletesen végezzük. Ne fordulhasson olyan eset elő, hogy a program egyes részei kiválóan működnek, míg más részek esetleg alig vannak megírva.

Megjegyzés:

A Top- Down módszer rövid kis programoknál nem igazán hatékony, mivel a végeredmény csak viszonylag sok fejlesztői befektetés után jelenik meg. Nagy projektek megalkotásánál viszont a kezdeti hosszabb előkészítő munka meghozza a gyümölcsét. Kevesebb lesz a logikai, adatszerkezeti hiba, a nyelvi függés és az improvizatív megoldás a programban. Áttekinthetőbb lesz a program kód is. Könnyebb kialakítani a következetes névmegadási konvenciókat is.

A legtöbb programozási nyelven meg lehet valósítani az ilyen fejlesztést, sőt a 4GL nyelvek egyenesen támogatják az ilyen fejlesztést azáltal, hogy a megfelelő objektumok megalkotásánál felkínálnak előre megírt programmodul vázakat, amelyeket később csak ki kell tölteni a megfelelő tartalommal.

7.5 Down-Up kompozíciós módszer

Ez a programfejlesztésnek egy másik sokat alkalmazott módszere. Elsősorban rövid, speciális programoknál célszerű használni. Lényege, hogy alulról felfelé építjük fel a programokat. Megírjuk az elemi építőköveket, algoritmusokat, majd azokból építjük össze a magasabb szintű struktúrákat, majd a programot összeállítjuk.

Ennél a módszernél a programozónak teljes rálátással kell rendelkeznie a megoldandó problémára. A részletek megoldásánál ügyelnie kell az esetleges kapcsolatokra is.

Elsősorban hardverközeli programozásnál lehet használni a módszert, ahol is a legfontosabb a hardverhez illeszkedő programrészek megírása, és ezek a programrészek esetleg alapvetően befolyásolják a program többi részének működését is. A hardverrel kapcsolatos fejlesztéseknél az is fontos, hogy a programozó viszonylag gyorsan meggyőződjön elgondolásai helyességéről, ezért olyankor nem is szokott, nem is akar komplett felhasználói felületet adni a programjának, hiszen a egyszerűbb paraméterezés a programok funkcióinak ellenőrzésére éppen elegendő. A hardverek programozásánál eleve léteznek olyan módszerek, amelyek a strukturált programozás szabályait felrúgják, ún. trükköket alkalmaznak.

Megjegyzés:

Illusztrációként álljon itt egy példa: Assembly nyelven írt programok esetén tipikus, hogy a rövidebb kód érdekében egy RETURN parancsot lehagynak. Az alábbi példában egy meghívott rutin meghív egy másik eljárást:

„Szabályos" példa	*„Trükkös" példa*
;belépési pont	*;belépési pont*
.....	*…….*
.....	*……*
JSR Másik_Eljaras_cím	*JMP Masik_Eljaras_cim*
RETURN	

A fenti példában a RETURN utasítást megspórolták, mivel a „Masik_Eljaras_Cim"-en kezdődő rutin végén feltehetőleg van egy RETURN, ami majd visszaviszi a programunkat az eredeti hívási helyre. Ez a trükk 1 byte megtakarítást és ami még fontosabb egy RETURN végrehajtásának megtakarítását eredményezte. A RETURN parancs során több belső művelet zajlik le, ami a program sebessége szempontjából nem lényegtelen, hogy hányszor zajlik le.

A Down - Top módszer igazából nem alkalmas nagyobb rendszerek tervezésére, mivel a fejlesztő először a részletekkel törődik, majd abból építene egységes egészet, ami általában nehezen sikerül neki.

7.6 Vegyes módszer

A fent említett két módszer tisztán a gyakorlatban csak nagyon ritkán jelenik meg. A leggyakoribb eset az, hogy a program fejlesztése során a program vázát a Top Down módszerrel tervezik meg, és finomítják ameddig a hardver-közeli, rendszer-közeli részekhez nem érnek, ugyanakkor bizonyos sebesség- vagy memóriakritikus részeket a Down - Top módszerrel oldják meg. Az így kialakult részeket, úgy illesztik össze, hogy mind a Top – Down módszer során megalkotott egységes felületekbe beilleszkedjenek a Down Top módszer konkrét megoldásai.

7.7 További programozási elvek

A továbbiakban olyan programozási elveket fogalmazunk meg, amelyek akár a program algoritmusának elkészítése, akár a kódolás során jól használható elvek.

7.7.1 Taktikai elvek

Niklaus Wirth egy cikkében az alábbi elveket fogalmazta meg a programok modulokra való bontásáról:

A párhuzamos finomítás elvét.

Korábban már említettük

Visszatérés az ősökhöz

A programok fejlesztésének bármely szintjén világossá válhat, hogy az általunk választott megoldás nem vezet célhoz. Ekkor olyan pontra kell visszatérni a programozás bármelyik szakaszában, ahonnan lényeges változtatást tudunk végrehajtani az eredeti megoldáshoz képest.

A döntések elhalasztásának elve

A programozási feladatok gyakran annyira összetettek, hogy a programozó nem láthatja át a megoldás minden aspektusát. Ekkor kell felhasználni az elvet. Mindig úgy programozzunk, hogy egy időben csak egy problémát oldjunk meg.

Döntések nyilvántartásának elve

Ha a program fejlesztése közben egy ponton szűkítjük a továbbiakban egy adat értelmezési tartományát, vagy a program többi részére is kihatással levő döntést hozunk, akkor azt a döntés pillanatában dokumentálni kell és a továbbiakban az illető adatokra való minden hivatkozás során figyelembe kell venni az értelmezési tartomány szűkítését. Például, ha egy adatbevitel során csak a pozitív egész számok jöhetnek szóba, akkor az algoritmusban és a kódolásnál is a bevitel helyén kell a bevihető adatok körét szűkíteni.

Az adatok elszigetelésének elve

Egy program fejlesztése során a programot egymással kapcsolatot tartó struktúrákkal valósítjuk meg. A program futása közben felhasznált adatok között vannak olyanok, amelyek a teljes programra tartoznak, és vannak olyanok, amelyek csak egy részfeladat megoldása közben szükségesek. Azokat az adatokat, amelyeket a program bármely részében fel akarunk használni globális adatoknak kell definiálni és azokat, amelyeket csak egy programmodul belsejében használunk, lokálisaknak kell definiálni. A ciklusokban, megszámlálásokban és egyéb hasonló helyeken igénybe vett változókat munkaváltozóknak hívjuk és a programozás minden szintjén ragaszkodni kell az azonos elnevezésekhez és természetesen mindig lokális változóknak kell őket definiálni.

Nyílt architektúra elve

Egy programozási feladatot igyekezni kell mindig a lehető legáltalánosabban megfogalmazni. Ez a későbbiekben a program módosításának, karbantartásának könnyítését eredményezheti.

A döntés elrejtésének elve

Mindig csak olyan finomítást hozzunk, amelynek a kihatása minél kisebb területre korlátozódik, lokális. Ha változtatnom kell valamit a programomban, akkor annak minél kisebb kihatása legyen.

7.7.2 Taktikai elvek

A technológiai elveket az algoritmusok írásánál és a kódolásnál is jól használhatjuk.

Bekezdéses struktúrák használata

Akár algoritmust írunk, akár kódolunk az elkészülő programszöveg olvashatósága elsőrendű feltétel. A ciklusok, elágazások, eljárások magját mindig célszerű egy tabulátorral beljebb írni és a struktúra azonos szintjén levőket azonos oszlopban elkezdeni.

Barátságos programok írása

A programoknak a felhasználó felé olyan képet kell mutatniuk, amely a felhasználó kegyeit keresi. A programnak magáról mindig el kell mondania azt a szükséges információt, ami elegendő a kezdő felhasználónak is a program használatához.

A képernyőn keresztüli adatbevitelnél megfelelő tájékoztató szövegnek kell megjelennie, továbbá a képernyőre kiírt adatoknak is a megfelelő kontextusban kell megjelennie.

Megjegyzések használata

A programszövegek – algoritmusok – írása közben szükséges a szövegbe olyan részel beírása, amely a nehezebben követhető programrészek működését szavakkal is megmagyarázza. Ez elengedhetetlen, mivel általában a programok fejlesztői sem emlékeznek a program minden részére megfelelően pár hónappal a fejlesztés után. A komment számukra is megkönnyíti a javításokat, illetve azok számára is, akik korábban nem foglalkoztak az adott feladattal.

Menütechnika használata

Azt hiszem ma már minden kisebb feladatot végrehajtó programnak is lehet menüje vagy menürendszere. A felhasználó dolgát megkönnyítjük vele.

Bolondbiztosság

Minden jól megírt program az adatok bevitelekor leellenőrzi, hogy a bevitt adat megfelel-e a programban előírt értéktartományoknak, és ha nem, akkor megfelelő üzenet után újra bekéri az adatokat. A jó program oly módon kezeli a programokat, hogy hibás adatbevitel esetén se száll el. Azt szokták mondani, hogy ha egy 6-8 éves gyerek egy programot nem tud hibás leállásra bírni, akkor tekinthető a program bolond-biztosnak.

A moduláris programozás felhasználásával a programozó jól működő programokat írhat – de ezt semmi sem garantálja.

8 Strukturált programozás módszere

8.1 A moduláris programozás előnyei

A moduláris programozással írt programok előnyei nyilvánvalóak

- Részprogramok könnyen áttekinthetők
- Könnyebben megírható
- Könnyebben tesztelhető
- Több modul írható egy időben (párhuzamos problémamegoldás)
- Könnyebben javítható
- A modulok szabványosíthatók
- Modulkönyvtárakban tárolhatók
- Újrafelhasználhatók

A fenti elvek alkalmazásával már használható programokat lehetett írni, de nem lehetett bizonyíthatóan helyes programokhoz jutni. A 70-es évek elején párhuzamosan, de egymástól függetlenül több programozási irányzat alakult ki, amelyek a strukturált programozás kialakulásához vezettek

8.2 Dijkstra: Hierarchikus programozás

A strukturált módszertan legabsztraktabb változatát **Dijkstráék** dolgozták ki. Hierarchikus programozásnak hívták ezt a módszert.

A strukturált programozás átveszi a moduláris programozás top-down módszerét. A megoldandó feladathoz a program egy absztrakt programsorozat határértékeként alakul ki. Ebben a sorozatban egy későbbi absztrakt program egy őt megelőző program egy változtatásával áll elő úgy, hogy tekintjük valamely tevékenységét, és fokozatosan finomítjuk. A megelőző program valamely tevékenységét kifejtjük (részfeladat). A sorozat minden absztrakt programja mellett ott van egy absztrakt számítógép, amelynek utasításkészlete megegyezik a programban használt utasításokkal. Végül egy konkrét gép konkrét utasításkészletére készül el a program.

8.3 Mills: Funkcionális programozás

Ugyanaz az elve, mint a Dijkstra féle programozásnak, csak itt az eljárásmodell az elsődleges, az adatmodell másodlagos. Tevékenység mellett osztja szét a problémát részproblémákra. A feladat határozza meg a program szerkezetét.

8.4 Wirth: A Programok részekre való bontásának elvei

Niklaus Wirth a programozás modulokra való bontásánál használt elveket fogalmazott meg

8.5 Jackson és Warnier: Adatorientált programozási módszertan

A top-down módszert vallja, de az adatmodell az elsődleges. Ha van egy probléma, fel kell derítenem az általa érintett adatok szerkezetét. A program szerkezetét az adatok szerkezete határozza meg.

Hátránya: csak adatfeldolgozási területen alkalmazható.

8.6 Boehm és Jacopini

’66-ban publikálnak egy cikket, amelyben leírják, hogy minden algoritmus felépíthető a következő három vezérlési szerkezet segítségével:

- Szekvencia (Soros program)
- Szelekció (Elágazást tartalmazó program)
- Iteráció (ciklust tartalmazó program)

1968-ban Mills ’68-ban bebizonyította a fenti állítást és az alábbiakkal egészítette ki:

Egy program akkor jó, ha a szerkezete leírható egy szekvenciaként, amely szekvencián belül a fenti három vezérlési szerkezet megengedett.

Egy szekvenciaelembe a külvilágból egy ponton lehet belépni és egy ponton kilépni.

Ha egy program ilyen, akkor az strukturált. Kellemesen dokumentálható, módosítható.

A Jackson féle elv azon alapul, hogy az adatszerkezetek szintén leírhatók e három vezérlési szerkezet segítségével, ugyanis egy állomány nem más, mint rekordok iterációja, egy fix rekord pedig mezők szekvenciája. Az adatszerkezetek meghatározzák a program szerkezetét.

A fenti elvek nem engedik meg a goto utasítás használatát, ugyanis a 2. pontot sértik. Ennek ellenére az utasítás a legtöbb programozási nyelvben megengedett maradt, bár használatát csak speciális esetben engedélyezik és nem ajánlják.

A **strukturált programozás** módszertana a ’70-es évek elején analízis módszertanná, rendszerfejlesztési, tervezési technológiává válik. A 1970 - 1980-as évek uralkodó módszertana. A strukturált programozás a modulok és az adatstruktúrák összefüggését tartja fontosnak a program tervezése során.

9 Objektum-orientált programozás – OOP

A felhasználói programok a 70-es években egyre bonyolultabbak lettek, és a programfejlesztés önálló iparággá vált. A fejlesztők azt vették észre, hogy a programfejlesztési projectek átlagosan 70%-kal tovább tartottak, mint a tervezett és átlagosan csak 30%-uk vált sikeressé. Ez oda vezetett, hogy az abban az időben már meglévő objektum — orientált programozási lehetőségeket kezdték kutatni.

A jegyzetben korábban már megismerkedtünk az osztály fogalmával, mint adatszerkezettel és annak példányosított formájával az objektummal.

Az alábbi rövid felsorolással az OOP elterjedésének állomásait szeretnénk felvázolni:

1969 Alen Key diplomamunkája során egeret használt, ikonokat, ablakokat és menürendszert

1972 konkrét elképzelések OOP és SMALTALK programozási nyelvről.(Dahl –Nygaard)

1981 Eiffel + eljárásorientált nyelvek egy részében OOP eszközrendszert vezettek be

1980-as évek végén a C programozási nyelv bővítéseként létrejött a C++, majd a Pascalt is kibővítették OOP elemekkel.

Az 1990-es évek közepén megszületett a Java, teljesen OOP nyelv.

1990-es évek vége. Majdnem minden modern programozási nyelvnek megszületik az Objektum orientált kiterjesztése.

Az OOP nyelvek nem matematikai elveken, inkább a bonyolultság növekedésének gyakorlati kezelése céljából jöttek létre.

9.1 Az OOP alapja

Az adat és a funkcionális modell nem elválasztható. Az adatok és a rajtuk végrehajtható műveletek elválaszthatatlanok. Ennek a következménye az **egységbezárás**.

9.1.1 Az Objektum

Az **Objektum** a változó általánosítása. Az objektumnak van

állapota (attribútum), az állapotot tetszőlegesen komplex adatok segítségével írjuk le (adatelemek + szerkezet)

viselkedése: ezt **módszereknek** (metódus) hívjuk (függvények és eljárások írják le.) A metódusok az alábbi kategóriákba soroklhatók

az objektumok állapotának lekérdezésére való metódus (az érték lekérdezése)

egyik állapotból másik állapotba vivő metódus (értékadások)

Egy objektum csak önmagával azonos és minden mástól különböző. Felmerül a kérdés, hogy ha két objektum azonos állapotban van, milyen a viszonyuk egymással:

Azonos állapotú két objektum-e

Ugyanarról az objektumról van-e szó

9.1.2 Az osztály

Az **Osztály** az adatttípus fogalmának általánosítása. Az azonos attribútumú és metódusú objektumok együttese.

Az objektumok az osztály példányai. Amikor egy osztály egy konkrét példányát létrehozzuk a programban, ezt hívják **példányosításnak**.

9.1.3 Egységbezárás

Egységbezárás - A hatáskör fogalmának általánosítása. A struktúrált programozásnál a lokális- globális változók kérdése volt.

Olyan eszközrendszer, mely segítségével megmondhatom, hogy az osztály attribútumaiból és metódusaiból kívülről mi látszik. (Igazi OOP-nél attribútumok általában nem látszanak, a metódusok közül csak a publikusak, és csak a specifikációs részük látszik)

Absztrakt osztályról beszélünk, amikor a specifikációt és az implementációt szétválasztjuk.

9.1.4 Öröklés

Az **Öröklés** a korábbi programozási módszertanokban nem létezett. Az öröklés az újrafelhasználhatóság kritériumával jelent meg.

szuper osztály elsődleges a kapcsolatban, alosztály csak hozzá kapcsolódhat.

Alosztály egy szuperosztály leszármazottja örökli a szuperosztály attribútumait és metódusait

- Az alosztálynak új attribútumokat és metódusokat definiálhatunk
- Elhagyhatunk, átnevezhetünk attribútumokat és metódusokat
- Megváltoztathatjuk a láthatósági viszonyokat
- Felülbírálhatja a metódusok implementációit.

Az öröklődés lehetőségei:

- Egy alosztálynak egy szuperosztálya lehet (1-szeres öröklődést támogató nyelvek) A hierarchia egy Fával írható le.
- Egy alosztálynak több szuperosztálya lehet (többszörös öröklődést támogató nyelvek) A hierarchia gráffal írható le.

9.1.5 Polimorfizmus

Polimorfizmus (többalakúság)

Példány polimorfizmus (egy konkrét háromszög példánya a háromszög osztálynak is és a poligon osztálynak is.)

Metódus polimorfizmus: Az alosztály újraimplementálhat egy metódust kérdés melyik kód fut le. Ezt a kötés mondja meg.

9.1.6 Kötés

Korai kötés: fordításkor eldől, hogy a meghívott módszerhez melyik kód tartozik.

Késői kötés: futás közben dől el, hogy a metódusspecifikációhoz melyik kód tartozik.

Minden példány tudja melyik osztály közvetlen példánya (aktuális példány)

Minden példány egy jól definiált példány aktuális példánya

(előny, pl. Ha egy módszer (A) hív egy másikat (B) a másikat újraimplementálom és meghívom (A)-t. ez esetben az újraimplementált fut le az első esetbe az eredeti)

9.1.7 Üzenet

Az **Üzenet** az objektumok közötti kapcsolat általánosítása és megoldása.

Egy program fejlesztésekor osztályokat definiálunk, létrehozzuk az öröklődési hierarchiákat. majd az osztályokból létrehozom az objektumokat.

A program futása közben az objektumok működnek, hatnak egymásra, üzeneteket küldenek egymásnak, válaszolnak az üzenetekre. Az üzenet valójában egy metódus meghívása (eljárás vagy függvényhívás)

9.1.8 Az Objektum orientált nyelvek fajtái

Tiszta OO nyelvek

Csak az objektumorientált eszközrendszer van

Van standard osztályhiearchia. A programozó mindig ezt a hiearchiát bővíti, ezekhez fűz objektumokat...

Minden eszköz objektum (az osztály, a módszer és az attributum is)

Ilyen programozási nyelvek Smalltalk, Eiffel, Java

Hibrid nyelvek

Eljárásorientált nyelvek objektumorientált eszközrendszerrel kiegészítve.

Részben vagy teljesen megvalósítják az OOP lehetőségeit

C++, PHP, Visual Basic, Borland Pascal, Delphi

Objektumon alapuló nyelvek

10 Nagyobb rendszerek fejlesztésének lépései

10.1 A rendszer tervezése

10.1.1 Egy nagyobb rendszer fejlesztésének megkezdése, előkészítése

Nagyobb, több hónapos vagy éves munkát is igénylő rendszerek tervezésénél figyelni kell sok olyan szempontra is, amely nem merül fel a rövidebb programok fejlesztésekor. Először is a „nagy rendszer” fejlesztője leggyakrabban nem saját szórakozásából áll neki a rendszer kifejlesztésének.

A rendszer fejlesztéséhez kell egy Megrendelő és a Fejlesztő. A Megrendelő egy bizonyos célt el szeretne érni a fejlesztendő szoftverrel, általában a munkáját szeretné könnyebbé tenni, amire esetleg még pénzt is hajlandó áldozni. Itt van az első buktató.

Sajnos a szoftverek felhasználói általában hozzá vannak szokva, hogy az általuk használt programok szinte ingyen állnak rendelkezésükre, hiszen szerte a világon az illegális szoftverhasználat elterjedt jelenség. Az illegális szoftver ingyen van. Azt is tudják, hogy egy nagyobb általános célú programcsomag, pl. egy Office bizonyos esetekben a CD áránál nem kerül sokkal többe, így azt hiszik, hogy egy számukra egyszerűnek tűnő programrendszer kifejlesztése is kb. ugyanaz a nagyságrend. Nem akarunk most itt gazdasági számításokba bocsátkozni, de egy multinacionális szoftverfejlesztő cég azért adja olyan olcsón a termékét, mivel több millió példányt ad el. Attól annak a fejlesztése több százezer mérnökórába került!

A Fejlesztőtől az előzetes puhatolózások során árat kérnek. A Fejlesztő ekkor még nem tudja megbecsülni a munka mennyiségét, ezért érdemben nem is tud nyilatkozni, nem is szabad nyilatkoznia. A munkájának értéke a tervezési folyamat során válik mind a két fél számára világossá. A fejlesztési folyamat értékét több féle módon lehet megközelíteni.

A fejlesztéshez szükséges idő alapján. Ekkor a fejlesztésre fordított munkaidőt beszorozzuk egy egységnyi óradíjjal és így kijön a fejlesztés értéke. Ez a díj a Megrendelőknek általában sok, így nem jöhet létre a kapcsolat. A megrendelés értékének felső becslésére megfelel.

A fejlesztés során létrejövő megtakarítás alapján. Ez konkrét megrendelés és konkrét fejlesztés esetén a megrendelés előtt nem határozható meg egzaktul. A becslés során bármelyik fél rosszul járhat. Nem célszerű használni ezt a módszert, csak akkor, ha konkrét számokkal kimutatható az eredmény.

Megéri-e a fejlesztőnek. A fejlesztő megbecsüli a fejlesztési erőforrásokat és megvizsgálja, hogy mennyiért éri meg neki a fejlesztés. Alapul azt veheti, hogy ha nem fejlesztene, akkor az alatt az idő alatt mekkora értéket tudna termelni. Ez a legbizonytalanabb módszer és gyakorlatilag alku kérdése ezen az alapon értéket mondani.

Talán a legjobb módszer, hogy a fejlesztés teljes volumenére a fejlesztés elején nem mondunk semmilyen végleges árat, hanem a fejlesztés adott lépéseihez kötünk értéket, így a megoldás első teljes, dokumentált terve egy sarokpont. Annak az elfogadására adunk árat. Ekkor már amúgy is eléggé körvonalazódnak a feladatok. Az első tesztváltozat átadására, a javított változat átadására majd a végleges változat átadására ütemezzük a többi értékelést. Minden egyes nagy lépést dokumentálni kell.

Szakértő

A fejlesztés során a Megrendelő részéről is sok munkát kell befektetni a kívánt cél eléréséhez. Az első kívánalom a Megrendelővel szemben, hogy jelöljön ki egy személyt (vagy többet), akik a fejlesztés megvalósítása során a Megrendelő részéről Szakértőként működhetnek, azaz rálátásuk van a megoldandó feladatra, a Fejlesztő feladatra vonatkozó szakmai kérdéseire válaszolni tudnak, és a fejlesztés során elkészülő részeredményeket tesztelni is tudják (képesek rá). Elvárható a Szakértőtől, hogy számítástechnikai kérdésekben a megfelelő szinten otthon legyen, hiszen sokszor olyan technikai jellegű kérdéseket is el kell tudnia bírálni, amelyeknek a konkrét feladathoz nem vagy csak áttételesen van köze.

Alapkövetelmény, hogy a Fejlesztő technikai kérdésekben csak a Szakértővel konzultál. A megrendelő oldalán több személynek is lehetnek ötletei. Ezek az ötletek a tervezés és a project előrehaladása során sokszor már megvalósult részekkel kerülnek ellentétbe. A Fejlesztő ezekre az ötletekre csak akkor reagálhat, ha a Megrendelő részéről a Szakértő már eldöntötte, hogy az ötlet beleillik-e az addig megvalósult elképzelésekbe vagy akadályozza a megvalósulást. A Szakértő a szűrő, ami számára néha meglehetősen kellemetlen helyzetet teremthet.

A Fejlesztő is a Szakértőnek adja át az elkészült részeket és a Szakértő dolga, hogy az elkészült részeket átvegye és tesztelje. A folyamat végén a szakértő lesz az a személy (lesznek azok a személyek), aki a rendszert érti és bizonyos hibák esetén javítani is tud.

Bizalmas adatkezelés

Az előkészítő megbeszélések alatt és később is a Fejlesztő bizalmas adatok, tesztadatok birtokába juthat. Azokat teljesen megbízhatóan, bizalmasan kell kezelnie.

Az első megbeszélések

A Fejlesztőnek a munka elején több alapos megbeszélést is kell folytatnia a Megrendelővel, annak tisztázás végett, hogy mi is a pontos feladat. Az első megbeszélések általában csak tapogatózó jellegűek, amikor is a felek tisztázzák a Megrendelés feltételeit, és körvonalazzák a feladatot.

Gyakori eset, hogy a Megrendelő és a Fejlesztő nem érti egymást, mivel az egyik félnek nincs meg az a fajta tudása, ami másiknak. Az első megbeszélések idején le kell tisztázni azokat a fogalmakat, amikről a fejlesztés során szó lesz, mindenkinek le kell írni minden olyan információt, ami a közös munkát segíti. Gyakran a feladat véglegesen csak a második, vagy még későbbi konzultációkon tisztázódik.

Ügyvitelszervezés

Egy ügyviteli probléma számítógépes megvalósítása gyakran a Megrendelő ügyviteli rendszerének az újraszervezését is maga után vonja. Ha ennek a szükségessége felmerül, akkor a fejlesztőnek ragaszkodnia is kell az átszervezéshez.

Az első terv

Ekkor születik meg az első terv a feladat megoldására. Ez a terv a Megrendelő számára minél részletesebb, elsősorban vizuális információkat tartalmazó terv. Abból kell kiindulni, hogy a Megrendelő alapvetően a saját problémáit látja, a saját elképzeléseit érti. Az első terveknek olyanoknak kell lenniük, amit a megrendelő megért, azaz azt a nyelvet kell használni, amit ő is ért. Természetesen a fejlesztőnek a saját nyelvezetét és fogalomrendszerét is a terv mögé kell tennie, sőt azt használnia is kell.

10.1.2 A rendszer tervezése

10.1.2.1 A programspecifikáció

Bemenő adatok – input

A tervezés során meg kell határozni a rendszer bemenő adatait. Meg kell állapítani, hogy milyen bizonylatok, milyen adatai szolgáltatnak adatokat a fejlesztendő rendszer számára.

Meg kell állapítani, hogy ezek az adatok nem tartalmaznak-e ellentmondást, arra figyelmeztetni kell a megrendelőt.

Meg kell állapítani azt is, hogy nincsen – e többszörös adatbevitel a rendszerbe. Ezt, ha csak lehet, el kell kerülni.

Meg kell állapítani, hogy a bevitt adatok mindig egzaktak maradnak-e. Gyakori, hogy az adatokat a kézi nyilvántartásokban szépítik vagy becslik. Az ilyen adatközlés nem eredményezhet jó végeredményeket sem. Az adatok jósága érdekében célszerű szem előtt tartani a következőket:

Célszerű az adatokat bevitelkor valamilyen egységes formátumra konvertálni, különös tekintettel a keresésekben, szűrésekben és indexekben szereplő adatokra. Az ilyen adatokat célszerű Nagybetűs formában tárolni és a bevitelnél eleve így konvertálni.

Hacsak lehetséges a program az adatokat előre megadott listákból várja.

Célszerű meghatározni a kötelezően kitöltendő adatok körét és a programozás során a megfelelő kóddal kell biztosítani a mindenkori kötelező bevitelt.

Törzsadatok

A törzsadatok azok az adatok, amelyek a rendszer használata során nem, vagy alig változnak. Ennek megfelelően a törzsadatok bevitele a többi adatok bevitelétől eltérő módú lehet. Adott esetben célszerű meglévő adatállományokat felhasználni erre a célra, vagy külön segédprogramokat szerkeszteni erre a célra. A törzsadatok bevitele is a munka értékének egyik paramétere. A törzsadatokat nem célszerű fixen beépíteni a végleges rendszerbe, mert azok az idők folyamán változhatnak.

Kimenő adatok – Output

A kimenő adatok általában képernyők, képernyős- és nyomtatott listák formájában jelennek meg. A két féle listának lehetőleg azonos, vagy hasonló formában kell megjelennie. A listák formájukban áttekinthetőknek és logikusaknak kell lenni.

10.1.2.2 Képernyőtervek

Nem hangsúlyozható eléggé, hogy a majdani felhasználó mennyire másképpen gondolkodik, mint a fejlesztő. Ennek megfelelően a legegyszerűbb formában a felhasználó a képernyőterveket érti meg. Számára a program tervezete elsősorban a képernyőterveket jelenti. Ha azon a feliratok megfelelően vannak elhelyezve, az adatok pedig a feliratoknak megfelelően jelennek meg, az a felhasználó számára maga a tökély. Éppen ezért a listákon nem szabad technikai jellegű feliratoknak, utalásoknak megjelenni, hanem a felhasználó fogalmait kell használni.

Célszerűen a menürendszereket is úgy kell megszerkeszteni, hogy azok a felhasználó fogalmainak megfeleljenek. Nem szabad olyan fogalmakat erőltetni, amelyek a felhasználók számára zavart okozhatnak, de a felhasználót nem szabad tudatlannak tekinteni. Azokat a fogalmakat, amelyeket a mindennapi életben használ, ha azok esetleg számítógépes fogalmak is, tudottnak lehet tekinteni.

10.1.2.3 Adatszerkezetek tervezése

A fentiek alapján már elkészíthető a rendszer adatszerkezete. Az adatszerkezetekre általános szabály nincsen. Az adatszerkezet az adatok tárolására szolgáló keret. Az adatszerkezet tervezésekor ajánlatos a következő elvekre figyelni.

Egyszeres adatbevitel, ne kelljen bevinni ugyanazt az adatot több helyen vagy több féle módon. **Egyszeres adattárolás**, ne legyen egy adat több helyen tárolva, és

A törzsadatokat úgy kell tervezni, hogy lekérdezésekben, listákban jól megjeleníthetőek legyenek.

Ha csak lehet, célszerű **kódokat használni szövegek** helyett. A kódok szerkezetét rögzíteni kell, esetleg formai előírásokat is be kell vezetni, hogy a kódok mindig megfelelő formátumúak legyenek. Inkább szöveg és szám keverékét használjuk, mint csak számot.

Egy adott mező méretének meghatározásakor arra kell figyelni, hogy szövegek esetén a mező mérete elegendő legyen a tipikus adatok tárolására – a kivételesen extra méretű adatokat lehet rövidíteni. Figyelni kell arra is, hogy bizonyos utasítások, adatbázis-kezelő parancsok a teljes adattartalmat átolvassák, hogy a megfelelő eredményt szolgáltassák. A **túl nagy mezők** az adatelérést lassítják esetenként jelentősen.

A **numerikus mezők** mérete az előrelátható legnagyobb értéket tudja fogadni, ezen belül numerikus eredménymezők esetén figyelni kell a nagyságrendek ugrására, továbbá a megfelelő mennyiségű tizedes helyre.

Megjegyzés típusú mezőket ne használjunk, ha nem muszáj. A megjegyzés majdnem minden nyelven feleslegesen sok helyet foglal el, sőt sokszor a megjegyzés adatbázis csak nő. Megjegyzésben keresni, indexelni, szűrni körülményes.

Logikai mezőket lehet használni, de figyelni kell arra, hogy mi is lesz a mező értelme.

10.1.2.4 Összefüggések az adatok között

Gyakori, hogy a különböző adatok összevetéséből számított új adatok jelennek meg a programban. Külön kérdés, hogy az így létrejövő új adatokkal mi legyen, töröljük őket, vagy a programra bízzuk az esetleges újraszámolást.

Olyan számított adatok esetén, ahol az eredményt időrendben egy adott állapotban rögzíteni kell, mint például importáru esetén az árfolyamot, a számított adatokat rögzíteni kell, és naplózni kell azokat a

körülményeket – itt például a valutaárfolyamot – ahogyan a számított adat létrejött. Ha a kiindulási adatok és a kiszámítás módja nem változik, akkor a számított érték mindig újra előállítható, így az értéket nem kell rögzíteni, elegendő csak a kiindulási értékeket tárolni.

10.1.2.5 Felhasználói felület – user interface

Mivel ma már a programokat széles körben használják, sokszor nem hozzáértő felhasználók is, ezért rendkívül fontos a programok felhasználói felülete. Ez indokolja azt, hogy a program tervezésekor különös figyelmet fordítsunk a felhasználói felületre.

Általában egy program indítására a program egy adott viselkedéssel válaszol. Ez a program alapbeállítása vagy idegen szóval **default** beállítása. A fejlesztőnek meg kell határozni a default tulajdonságokat, amelyeket a felhasználó módosíthat.

10.1.2.5.1 Paraméteres indítás

Ekkor a program a parancssorban megadott paraméterek hatására a default viselkedéshez képest más tulajdonságokkal kezd működni. A paraméterek parancssori használata a nagygépes operációs rendszerektől ered. A PC-s világban a UNIX operációs rendszeren át a DOS-os programok is átvették a paraméterek használatát, majd a Linux rendszerek alapvetően megmaradtak ilyeneknek. Az ilyen programok használatának megtanulása felesleges terhet ró a felhasználóra, abszolút barátságtalan a kezelőfelület.

A hozzáértő felhasználó viszont így tudja a programok **legmegfelelőbb beállításait** használni és így tudja más programok által meghívva a kívánt viselkedésre bírni a legegyszerűbben. Ez a fajta felhasználói felület még a grafikus rendszerek megjelenésével sem ment ki a divatból. A paraméterek egyike a /?, ?, vagy a /h. Ezek a paraméterek szokás szerint a programok használatáról nyújtanak segítséget.

A paraméteres indítás esetén **hátrány** lehet a nem megfelelő paraméterek feldolgozása. Mi történjen nem megfelelő paraméterek használata esetén. Általános szabály nem létezik. Olyan választ kell adni, amely szöveges formában tájékoztatja a felhasználót a hibáról és annak megoldási módjáról. Az ilyen hibaüzenet nem jó: „Runtime Error, errorcode 1201 in line 1324”. Ha a programozó ezt a módszert használja, akkor vigyáznia kell, hogy:

Csak azokat a paramétereket fogadja el a program, amelyek a program viselkedésén változtatnak;

A paraméterek írásmódja tetszőleges legyen, azaz kisbetű – nagybetű nem számíthat;

A DOS-ban a paraméterek előtt szokásosan a „-„ vagy a „/” jel áll, lehetőség szerint tartani kell ezt a konvenciót;

A DOS-ban bizonyos jelek nem megengedettek a paraméterekben – ezeket nem szabad használni;

A paraméterként adott karakterek, szavaknak utalnia kell a megadott funkcióra, anyanyelven vagy angolul, ráadásul a nyelv kérdésében következetesen az anyanyelvet vagy az angolt kell választani, keverni nem lehet őket.

Célszerű fenntartani egy paramétert egy rövidebb vagy hosszabb HELP kiíratására. A HELP-nek akkor is célszerű megjelennie, ha hibás paraméterezést adtunk a programnak;

Ha megoldható, akkor célszerű a paramétereket tetszőleges sorrendben bekérni és célszerű a paraméterek hiányában valamilyen default viselkedést előírni. Ez a akár a párbeszédes adatbekérés is lehet.

A paraméteres felület előny, ha az adott operációs rendszerben automatizálni akarunk folyamatokat. Ily módon a windows rendszerek is meglehetősen összetett Batch programozási nyelvvel rendelkeznek.

10.1.2.5.2 Párbeszédes felület

A program az indítás után megkérdezi a beviendő paramétereket, majd az eredményt kiírja képernyőre, általában a bevitt adatok alá.

Ez a fajta párbeszédes, üzengetős interface a nagygépes világból átöröklött, ma is meglévő kezelői mód. Ezek a programok egyáltalán nem tekinthetők „barátságosnak”, a felhasználónak könnyű rossz válaszokat adni a feltett kérdésre. Csak nehezen találhatók meg a programban adott lehetőségek. Ha adatbevitelkor hibás típusú adatot viszünk be, előfordulhat, hogy a program futás közbeni hibával elszáll és a felhasználó csak találgathat, hogy mi volt a hiba oka.

Kezdő programozók által gyakran használt módszer. Tanuláskor a legegyszerűbben így lehet adatbevitelt létrehozni a program részére. Ha a programozó mégis ezt a felületet választja, akkor vigyáznia kell:

A adatbevitel során a bevitt adatok típusát ellenőrizze a program és a bevitelkor figyelmeztessen a típus-, vagy nagyságrendi hibákra

Hibás adatbevitel esetén kínálja fel az újrapróbálkozás vagy a kilépés lehetőségét

A beviendő adatok előtt mindig írja ki pontosan, hogy mit vár a felhasználótól, és egyértelműen jelölje meg az adatbevitel helyét is a kurzorral

Ha az adatbevitel során eldöntendő kérdést vagy kérdéseket teszünk fel, akkor mindig ugyanazokat a karaktereket válasszuk a válaszadásra – lehetőleg a válasz Igen/Nem anyanyelvi esetben I vagy N, angol Yes/No esetben pedig Y/N legyen – kisbetű, nagybetű ne számítson

Az eldöntendő kérdés mindig pontosan legyen megfogalmazva és legyen kiírva a lehetséges válasz is;

- Ne fogadjon el más karaktereket az eldöntésnél
- A bevitt adatok után a program valamilyen választ adhat, ez jól elkülönülve jelenjen meg a képernyőn, lehetőleg a következő sorok egyikében, a sor elején kezdve
- Nem célszerű színeket használni, csak ha valamiért ki kell emelni egy sort. A színek a monokróm képernyőn esetleg nem látszanak elég jól.

A fenti két módszert nagyon gyakran együtt alkalmazzák DOS-os, Linuxos, Netware környezetben. Ha a paraméterek nincsenek megadva a parancssorban, akkor kérdéseket tesznek fel és így biztosítják a hiányzó adatokat.

10.1.2.5.3 Menüs Interface

Mind alfanumerikus, mind grafikus környezetben a mai programok többsége elképzelhetetlen menük nélkül, ennek ellenére vizsgáljuk meg, hogy mikor nem szükséges menüket használni.

Alapvetően a menü választást jelent több lehetőség közül. Ha nincsen választási kényszer, akkor menüre sincsen szükség.

Azokban a programokban, amelyeket egy adott célra fejlesztettek ki, felesleges menüket alkalmazni, hiszen a program elindítása eleve a kért funkciót is elindítja.

Ha egy programot egy másik programból indítunk el – pl. keretprogramból egy tömörítőt-, akkor a keretprogram hordozza a választásokhoz szükséges információt, tehát ekkor sem kell menü.

Minden egyéb esetben van a menüknek létjogosultsága.

Hogyan jelenjen meg egy menü?

A kezdő programozók legegyszerűbb menüje a következő:

Egymás alá felsoroljuk a menüpontokat, mindegyik elé egy sorszámot írunk, majd feltesszük a kérdést, hogy melyiket választja. A választástól függően egy CASE szerkezettel kiválasztjuk a megfelelő menüpontot, majd a megfelelő eljárást meghívjuk. Az eljárásból visszatérve a menüt újra kirajzoljuk és kezdődik minden elölről.

Ennek egy kicsit módosított változata, ha nem sorszámot adunk a menüpontnak, hanem karakter, illetve ha a menüpont egyik betűje kiemelt, akkor azt a billentyűt megnyomva indul a kérdéses funkció.

Ennél egy kicsit bonyolultabb megoldás, amikor a menü nem csak egy egyszerű felsorolás, hanem a menüpontok között a megfelelő irányú kurzormozgató billentyűkkel is lehet mozogni. Ekkor gondoskodni kell arról, hogy az aktuálisan kijelölt menüpontnak más legyen a színe, mint a többi menüpontnak, illetve monokróm monitoron legyen intenzívebb a megjelenése. Az ilyen menükben a **nyilakon** kívül célszerűen a **Home** és az **End** a menü első illetve utolsó pontjára ugrik, az **ESC** jelentése kilépés a menüből, az **Enter** elindítja az aktuális menüpontot.

Popup menü

Amikor egy menü megjelenik a képernyő közepén esetleg letakarva a mögötte lévő képernyőterületet, akkor azt **előugró** vagy angolul **POPUP menünek** hívjuk. Ha egy menüpont kiválasztásával egy másik menü nyílik meg, akkor azt **almenünek** hívják. Az így létrejött menüből és almenükből álló rendszert **menürendszernek hívjuk** és ennek a menürendszernek a kezdete a főmenü. Egy menürendszer mindig ábrázolható egy fa struktúrával is, ezért leírásokban a menürendszer egy almenüpontját egyértelműen meg lehet

adni, ha a főmenütől kezdve felsorolom azokat az almenüpontokat, amelyek elvisznek a megfelelő funkcióhoz.

A bonyolultabb menürendszerű programokban a felhasználó elveszhet a sok menü között, ezért felmerült az igény, hogy a menüpontokhoz magyarázatot is kell fűzni. DOS-os környezetben az aktuálisan kiválasztott menüponthoz tartozó magyarázat általában a képernyő alsó sorában jelenik meg, míg Windowsos környezetben manapság alkalmaznak kis keretben megjelenő leírást, amikor az egérrel megállunk egy menüponton. Ezt angolul **tooltip-nek** hívják.

Külön feladatot jelent menükezelésnél az egér használata. Az egér pozícióját windowsos parancssorban nem feltétlenül lehet feldolgozni, azt a saját programmal külön le kell programozni.

A menürendszerben mindig van egy várakozó rész, amely a billentyűlenyomásra vár. Ezt a grafikus felületek automatikusan biztosítják. Itt lehet beiktatni az egér események figyelését is, és ha az egér valamelyik billentyűjét lenyomjuk, akkor utána le kell kérni a pozícióját, meg kell vizsgálni, hogy menüponton áll-e és ha igen, akkor meg kell határozni a menüpontot. Innen több választás lehetséges. Az egyszerűbb esetben a program betölti a billentyűzet pufferbe a megfelelő billentyűt és hagyja a várakozó ciklust tovább haladni. A következő körben a ciklus észleli a karakter megérkezését és feldolgozza azt. Bonyolultabb esetben az egérkoordináták alapján a menüponthoz tartozó eljárást rögtön elindítja, azaz a billentyűzet és az egér menükezelése párhuzamos lesz.

10.1.2.5.4 Gyorsító billentyűk

A régebbi DOS-os programoknál a menürendszer utolsó almenüjén keresztül lehetett egy-egy beállítási funkcióhoz eljutni. Gyakran az almenü szomszédos menüpontjának eléréséhez pedig elölről kellett kezdeni a bóklászást a menürendszerben. Ez megnehezítette a kezelést, ezért a menüket ellátták gyorsbillentyűkkel, amit angolul SHORT-CUT-nak hívnak. Ezek segítségével a menürendszer nélkül lehetett elindítani funkciókat. A gyorsító billentyűk használata csak sok gyakorlás után válik természetessé, hiszen ezeket az ALT-CTRL-SHIFT-Billentyű kombinációkat nehéz megjegyezni. Illetve minél bonyolultabbak a használt programok, annál jobban kell vigyázni, hogy a short-cut-ok mindig következetesen ugyanazt jelentsék. A programozásuk is nehezebb, hiszen a menürendszerben biztosítani kell azt, hogy bármikor a megfelelő short-cut hatására a megfelelő eljárás induljon el. Nem könnyű feladat. A gyorsítóbillentyűk használata azonban megkönnyítheti és meggyorsíthatja a programok használatát.

10.1.2.5.5 Párbeszédablakok

A menürendszerek használatának könnyítésére a párbeszédablakokat is használunk a felhasználói interface szokásos részeként. A párbeszéd ablakok (angolul DIALOG) olyan a képernyőn keretben megjelenő objektumok, amelyek több kezelőfelületet, gombot, beviteli helyet, kapcsoló, választási lehetőséget tartalmaznak. Az egyes kezelőszervek beállításai csak akkor válnak véglegessé, ha a párbeszédablakon megjelenő elfogadó nyomógombot – Ok, Rendben vagy valami hasonló feliratú – megnyomjuk. Ha a szintén általános Mégsem, Mégse vagy Cancel feliratú gombot aktiváljuk, akkor a beállítások nem véglegesítődnek. A párbeszédablak megjelenő kezelőszervei között a TAB és a SHIFT+TAB billentyűkkel lehet általában mozogni. DOS-os környezetben eléggé körülményes megvalósítani a párbeszédablakokat, de nem lehetetlen. Objektum orientált módszerrel programozva léteznek ilyen könyvtárak a Turbo PASCAL és a Turbo C nyelven is. Minden windowsos programozási nyelv természetesen rendelkezik ilyen lehetőséggel.

10.1.2.6 Segítségadás a programokban

A felhasználói interface-hez szorosan kapcsolódó, ámde mégis különálló téma a help – segítség kérdése. Kezdetben a segítség a felhasználói kézikönyvet, később az egyszerűbb programok használatakor a segítség a /? vagy hasonló paraméterrel történő indításkor egy vagy néhány oldalnyi legördülő szöveg megjelenését jelentette.

Ma már a segítség több mindent takar és több szintje is lehet. A legegyszerűbb valóban a megjelenő egy oldalnyi információ. Bonyolultabb programok, menürendszerek esetén ez már nem elég. Ilyenkor jelennek meg a helyzethez alkalmazkodó segítő rendszerek. A korábban már említett menüponthoz kapcsolódó egysoros információ lehet a segítő rendszer második lépcsőfoka, de ekkor még mindig fennáll a tudatlan felhasználó alulinformáltságának veszélye. A harmadik lehetőség, amikor a kedves felhasználó megnyomja az F1 billentyűt és egy ablakban megjelenik egy részletes szöveges, képes leírás, amelyben mindenféle, az adott menüponttal kapcsolatos tudnivalót leír a fejlesztő. A DOS-os Norton Guide vagy a Windows help rendszere ennél is tovább megy. Nem elég, hogy megjelenik a szöveg, de a szövegben utalások vannak más

szövegrészekre is, amelyekre egyszerű kattintással lehet átlépni. Az ilyen HELP rendszert HYPERTEXT rendszernek hívják. Az ilyen rendszerek komoly programozást igényelhetnek.

10.1.2.7 A tervezés lépései.

Egy számítógépes rendszer megtervezése általában nem egyszerű feladat. A megvalósítandó rendszer rengeteg összetevőből állhat. Nem biztos, hogy csak egy egyszerű program kifejlesztéséről van szó, hanem gyakran hardverből, megfelelő operációs rendszerből, kifejlesztett szoftverből álló összetett rendszert kell megtervezni és megvalósítani, a kívánt feladatok elvégzésére. Az egyes komponenseket – a hardvert, az operációs rendszert és a szoftvert - sokszor nem is lehet elválasztani egymástól. A fejlesztés tervezése során nagyon fontos a célszoftver kifejlesztéséhez felhasználandó eszköz gondos kiválasztása. A következőkben az egyes komponenseknél figyelembe veendő szempontokat fogjuk megtárgyalni.

10.1.2.8 A megfelelő hardverháttér megállapítása.

A szükséges hardver függ a futtató operációs rendszertől, a használt fejlesztőeszköztől is.

A multitaszkos operációs rendszerek világában azt lehet mondani, hogyha egy számítógép egy adott operációs rendszert lassan futtat, akkor a felhasználói programokat is lassan futtatja majd.

Gyakori eset az, hogy a megrendelő a meglévő hardverparkot alkalmasnak tartja a feladat elvégzésére, ugyanakkor a fejlesztő a saját rendszerét használja tesztelésre és a fejlesztőnek kompromisszumokat kell kötni.. Ha a fejlesztő kompromisszumokat köt, nem veszi figyelembe az alkalmazandó operációs rendszer és fejlesztőeszköz kívánalmait, akkor saját magának nehezíti meg a helyzetét. Az elkészült rendszer mélyen a kívánt teljesítmény alatt fog teljesíteni.

Általában figyelembe kell venni a hardver tervezésénél a gép processzorát, memóriaméretet, a szükséges háttértár méreteket. Ezen kívül a hálózat meglétét vagy szükségességét, modem és egyéb szükséges eszközöket is figyelembe kell venni. Szükség esetén elő lehet írni megfelelő grafikus felbontást is. Akövetkezőkben az elterjedt rendszerek alapvető hardverkövetelményeit soroljuk fel (2007-et írunk)

Ma már gyakorlatilag DOS-os programokat nem fejlesztenek,.

A Windows 95, Windows98, Millenium Edition, Windows NT 4, Intel PII-es (Celeron) processzoron 32 MB RAM-mal elfogadható teljesítményt nyújtanak.

A Windows 2000 kissé elavult már, de sokfelé élő rendszerek futnak ilyen operációs rendszeren 64 MB minimum, 256 MB Ram már elegendő, 500 MHZ-es P-II/Celeron.

A Windows XP minimum 128 MB RAM-ot használ és 256 MB elég jó neki. 1-2 GHz-es Celeron/P-III/P4 processzor célszerű alá.

A Linux operációs rendszer, ugyanolyan hardveren jobban teljesít mint egy XP, ha nem használkunk grafikus felületet.

10.1.2.9 A megfelelő operációs rendszer

Az előző fejezetben felsorolt operációs rendszerek azok, amelyek a leggyakrabban szóba jöhetnek. Most azt vizsgáljuk meg, hogy melyik rendszer milyen alkalmazások esetén a legalkalmasabb a fejlesztés és a futtatás szempontjait figyelembe véve.

Meglepő, de nagyon sok esetben a DOS alkalmas futtató rendszer! Az okok meglehetősen sokrétűek.

A DOS 25 éve jelent meg! A BIOS és a DOS hívásai abszolút publikáltak, a fejlesztő rendszerek sokadik generációi készítenek DOS-os programokat. Az x86-os processzorok real módját már minden fejlesztő rendszer készítő oda-vissza kiismerte.

A DOS-os programok általában stabilak. A számítógépek gyorsan bebootolnak DOS esetén. A memóriakezelés gyors, mert kevés információt kell változtatni. A DOS-os driverek kis méretüknek köszönhetően általában assemblyben készültek, de a szabványok már olyan jól ismertek, hogy akik ilyen programokat készítenek volt idejük jó drivereket írni. A hálózatkezelése tökéletesen megoldott. Gyakorlatilag minden PC-s környezetben működő hálózatnak van kliensprogramja hozzájuk. Az MS-DOS 5.0 nagy áttörés volt stabilitás szempontjából is. Ettől a verziótól számítva a memóriakezelése is jónak mondható. A DOS 640 KB memóriájából 620-630 is szabaddá tehető egyes esetekben. A memóriaméretnek megfelelő programok gyorsan betöltődnek, gyorsan harcra készen állnak. Ha egy program mégis lefagy, akkor legfeljebb egy-két lezáratlan állomány marad a lemezen. Akár egy floppyról is lehet futtatni egy DOS-os rendszert, még akkor is ha az

alkalmazás esetleg overlay technikát használ – RAMDISK használatával. Hálózati környezetben akár a szerverről is futhatnak az alkalmazások.

Fejlesztői szempontból célszerű DOS-os alkalmazást készíteni, mivel nincs multitaszkolás, nincsenek az alkalmazás alól más alkalmazások által kihúzott programok. Az operációs rendszer erőforrásait könnyű elérni. Az operációs rendszer szolgáltatásait megszakításokkal, a környezeti változók alkalmazásával, a keresési útvonalak használatával könnyen felhasználhatjuk.

Hátrányok. A grafikus gyakorlatilag nincsen. Bizonyos fejlesztőrendszerek olyan programokat fordítanak, amelyek memóriaigénye közelít a 640 KB-hoz, azaz krónikus memóriahiányban kezdenek szenvedni az alkalmazások. Ezt SWAP és overlay technikával küszöbölik ki, illetve egyes linker programok alkalmassá teszik az általunk írt kódot a DOS ún. protected üzemmódjában való alkalmazásra. Ekkor a program részére nem csak a 640 KB memória áll rendelkezésre, hanem a gépben lévő összes XMS memóriát fel tudja használni a program.

Generálisan nem megoldott probléma a magyar nyelvű ékezetes üzenetek megfelelő kiírása a képernyőre és a nyomtatóra.

A Windows 3.1/3.11-es shell instabil a Windows kooperatív multitask koncepciója miatt. Ha a rendszerben lévő egyik program kisajátítja magának a processzort, azaz nem adja át a vezérlést a többi programnak, akkor a rendszer menthetetlenül összeomlik. Kiment a divatból.

A Windows használata közben sok olyan dolog felmerül, ami a DOS-os programok használóit nem érinti. A multitask miatt a rendszerben elindított programok kárt okozhatnak más futtatott alkalmazás fájljaiban futás közben, esetleg megmagyarázhatatlan hibákat előidézve. A kezelés során az egér használata bizonyos esetekben komoly problémák forrása lehet. A fejlesztők is néha akarva-akaratlanul az egér használatát írják elő még akkor is, ha nem a legcélszerűbb. Mindazonáltal el kell mondani, hogy olyan munkahelyen, ahol a gépet office-szerű programok futtatásra is használják, ott a Windows kezelése nem jelent általában problémát.

A grafikus programok nagyobbak, mint a karakteres felületű programok, ezáltal lassabbak. Ezt tudomásul kell venni.

A nyomtatás és a képernyőképek a Windows-ban egyszerűen tervezhetők. A grafikus programok felülete megfelel programokkal gyorsan tervezhetők.

A nyomtatás tekintetében is egyszerű a helyzet. A nyomtatók rendszerszinten definiáltak, azaz a windowsos programok tervezőinek nyomtatáskor a nyomtató fajtájtól függetlenül, a Windows szolgáltatásait felhasználva nyomtathatnak. A modern 4GL rendszerek eleve képesek a kódot a képernyőn megjelenő objektumokhoz igazítani, így forrásszinten áttekinthetőbb programok készíthetők.

A Windowsos programok futtatása esetén felmerül egy probléma. A különböző Windows rendszerek nem feltétlenül kompatibilisek egymással. Bizonyos állományaik nyelvi jellemzőkkel rendelkeznek, amelyeket nem mindig lehet figyelmen kívül hagyni. Azonos nevű könyvtári fájlokat tartalmazhat a Windows és a fejlesztői program is. Az egyes könyvtári fájlok viszont a fejlesztés különböző stádiumaiban vannak, tehát nem ugyanarra képesek. A Windows egy fájl betöltésekor megnézi, hogy a memóriában létezik-e és ha igen, akkor még egyszer nem tölti be. Ez futás közben potenciális és nehezen felderíthető hibaforrás lehet. Célszerű stratégia mindig az azonos nevű és azonos funkciójú fájlok közül a legújabb dátumú vagy legújabb verziószámú használata. Ha a rendszerben eredetileg lévőnél régebbit helyezünk a rendszerbe, akkor az esetlegesen kijavított hibákat visszacsempésszük a rendszerbe és előre nem látható hibák jelentkezhetnek. Ezt a problémát hívják „DLL Hell” -nek (= dll pokol).

Egy programfejlesztő barátom egyszer azt mondta, hogy azért nem szeret Windowsban programozni, mert akkor nem tudja, hogy a háttérben mi történik. Erre én azt feleltem, hogy ha a program jól működik, akkor nem is kell. A probléma csak az, hogy a programok nem mindig a szabvány szerint működnek.

A Linux rendszereken is felmerül a parancssori (=konzol) illetve a grafikus felületű program kérdése. Mindig amire szükség van.

10.1.2.10 A felhasználható fejlesztőeszközök kiválasztási szempontjai.

Nyilvánvalóan az előzőekkel szorosan összefügg a fejlesztésre használható rendszer, vagy rendszerek kiválasztása. Hogy melyik rendszert válasszuk ki, arra nézve a következő elveket érdemes figyelembe venni.

Nézzük meg, hogy mennyire alacsony szintű – rendszer közeli – a feladat!

Rendszer közeli alkalmazások fejlesztéséhez Assembly, C, C++, esetleg a Pascal nyelv ajánlható. A létrejövő program a fenti sorrendnek megfelelő méretű lesz, a futási sebessége fordított lesz, a fejlesztéshez szükséges idő – egyenlő tudásszinteket feltételezve – Pascal esetén a leggyorsabb, és Assemblyben a leglassabb.

Kisebb segédprogramok fejlesztésekor az Assembly-t érdemes kihagyni, de a C és a Pascal továbbra is jó választás. Ezek általános célú nyelvek, minden elvégezhető velük, amire általában egy programban szükséges lehet. Célszerűen script nyelvet is lehet használni!

Összetettebb alkalmazások fejlesztésére célszerű olyan fejlesztőrendszert keresni, amely az adott területre kihegyezett. Ekkor a harmadik generációs programozási nyelvek – C, Pascal – már nem elég hatékonyak.

Vizuális, grafikus programok esetén használhatjuk a Microsoft Visual Stúdió (Basic, C, C#) a Sun féle Java, Borland C-Builder, Delphi, és még sok fejlesztő rendszer programjait.

Nézzük meg a feladat bonyolultságát!

Gyakran előfordul, hogy egyszerűbb feladat megoldására már létezik az adott szoftver. A kisebb adatbázis-kezelő célrendszereknek, és utility programoknak se szeri, se száma a világ szoftvertermésében. Célszerű először tájékozódni, hogy létezik-e az adott feladatra kész program, majd ha nem találunk, akkor álljunk neki a fejlesztésnek.

Egyszerűbb adatbázis-kezelési feladatok megoldására olyan fejlesztőrendszert célszerű választani, amelyben egyszerű sablonok adatainak kitöltésével lehet a megfelelő célszoftvert összeállítani. Ilyenek pl. a Microsoft Access, dBase, a Database Manager. Egyszerűbb alkalmazások összeállítása 2-3 órai munkával már megoldhatók.

Több adattáblát tartalmazó programok fejlesztésekor már több variáció is szóba jöhet, hogy csak a legismertebbeket említsük, Microsoft Access, CA-Visual Object, Delphi, Microsoft Foxpro/Visual Foxpro, Visual Dbase, stb… A felsorolt rendszerek mindegyike 4GL rendszerű.

Nézzük meg a futtatási platformot!

DOS-os alkalmazást Windows alól futtatni nem ajánlatos.

A DOS-os programok súlyos korlátja a DOS memóriakezelése. A nagyobb DOS programok 500-600 kb-ot is lefoglalnának maguknak, ezért esetleg bizonyos memóriakezelési kompromisszumokat is szükséges kötni.

A felhasználói felület a grafikus rendszereken sokkal barátságosabb lesz, gyorsabban ki is fejleszthető. A megfelelő hardvert, azonban biztosítani kell, azaz processzorral, memóriával és szabad winchester hellyel nem szabad takarékoskodni.

Célszerű, ha lehetséges kliens-szerver alkalmazásokat írni, mivel ilyenkor a számítási feladatok egy részét a szerver átvállalja a munkaállomásoktól.

10.1.2.11 Fizetős, Shareware, Freeware programok. OEM, Multi licensz.

A szerzői jog úgy rendelkezik, hogy a szoftverek jogvédettek, azaz a fejlesztők szellemi termékei. A vásárló a program megvásárlásakor csak a program felhasználási jogát vásárolja meg, nem kerül tulajdonába maga a szoftver. Ennek megfelelően nem rendelkezhet korlátlanul a szoftvere felett.

Általános szabály, hogy a fizetős alkalmazások megvásárlásakor egy telepítőkészlet, továbbá valamiféle sorozatszám, vagy egyéb azonosító kerül a felhasználó birtokába. Gyakori megoldás, hogy a telepítőkészlet az internetről letölthető és csak a sorozatszám érkezik a vásárlás kapcsán.

Mikor jogtiszta és mikor illegális a szoftver felhasználása?

Ha a felhasználó rendelkezik a használatot lehetővé tevő sorozatszámmal, azonosítókkal, továbbá bizonyítani tudja, hogy a szoftvert vásárolta vagy bármiféle hivatalos úton jutott hozzá. Ilyen hivatalos út lehet ajándékozás is például. A szoftver tulajdonosa gyakran előírja a megvásárolt szoftver elidegenítésének lehetőségeit

is. Általában igaz, hogy egy megvásárolt szoftvert csak akkor lehet továbbadni, ha a szoftvert minden tartozékával együtt – telepítő készlettel, felhasználói kézikönyvekkel, sorozatszámokkal, igazoló dokumentumokkal stb... együtt adják át a vásárlónak.

OEM szoftverek (=Original Equipment Manufacturer).

Ez olyan konstrukció, amikor a szoftvert egy összeszerelt számítógéppel vagy valamilyen meghatározott alkatrésszel együtt kell eladni. Ez elsősorban az operációs rendszerek némelyikénél szokás.

Mennyiségi licensz

Gyakori, hogy egy cég egy bizonyos szoftverből többet vásárol. Ekkor mennyiségi kedvezmény alapján a cég alacsonyabb áron juthat hozzá a szoftver használatához. Gyakori ilyenkor, hogy az eladó csak egy adathordozót biztosít a felhasználónak, vagy csak az internetről való letöltés lehetőségét biztosítja.

A megvásárolt szoftverek licenszelését gyakran kötik felhasználóhoz és/vagy számítógéphez. Általában az igaz, hogy a megvásárolt szoftvert egy időben csak egy gépre szabad feltelepíteni.

Shareware programok

Olyan szoftverek, amelyek bizonyos, a szerző által meghatározott körülmények között teljes körűen, vagy bizonyos meghatározott megszorításokkal használhatók. A megszorítások sokrétűek lehetnek. Lehetséges, hogy a program egyfolytában csak egy megadott ideig működik, vagy az indulásakor valamilyen figyelmeztetés jelenik meg, de egyébként akármeddig, teljes körűen használható. A lehetőségek száma nagy.

A jogtiszta használat akkor valósul meg, ha a licenszben leírt feltételekkel használja a programot és amint megszűnt a licensz által biztosított feltétel, akkor letörli. Általában a shareware programok szerzői előírják, hogy a megadott idő után köteles megvásárolni a terméket a felhasználó.

Próbaváltozat

Gyakori, hogy az interneten elérhetők egyes szoftverek próbaváltozatai, amelyek meghatározott ideig és meghatározott feltételek mellett működnek, majd a feltételek lejárta után automatikusan megszűnnek működni és kikényszerítik a megvásárlást vagy törlést. Ezek egyfajta shareware programok

Freeware programok

Ez a programkategória azt jelenti, hogy a programot szabadon, megkötöttségek nélkül lehet használni. Ez nem jelenti azt, hogy a programhoz ingyen lehet hozzájutni, lehetséges, hogy pénzért kell megvásárolnia. Ha a felhasználó a programhoz hozzájutott, akkor azt szabadon terjesztheti, felhasználhatja tetszőleges számú gépen és formában, akár más alkalmazások részeként is.

Nyílt forráskód

Ha a program fejlesztője a programok forráskódját is közzéteszi, akkor nyílt forráskódú programról beszélünk. A nyílt forráskód azt jelenti, hogy bárki a forráskódot módosíthatja szabadon. Általában a fejlesztő elvárja, ha valaki módosítja a forráskódot, akkor arról értesítse őt.

Azt gondolná a naiv ember, hogy a nyílt forráskóddal a fejlesztők lemondanak anyagi előnyökről és így a munkájuk értékét veszti. Gyakorlatban a nyílt forráskódú rendszerek komoly alternatívát jelentenek az üzleti rendszereknek, mivel a nyíltság miatt gyakran a világ összes részéből dolgoznak emberek a kódon, ezáltal nagyon stabil kiérlelt kódok jönnek létre. A XX. Sz végén a Linux operációs rendszer példája mutatja, hogy igenis lehet akár üzleti alapú működés is a nyílt forráskódú szoftverekkel.

10.1.2.12 A terv dokumentálása

A rendszerfejlesztés megkezdése előtt nagyon fontos feladat az elkészítendő rendszer paramétereinek rögzítése, azaz a terv dokumentálása. Ebben a dokumentumban minden olyan alapvető paramétert rögzíteni kell, amelyek a későbbiekben meghatározzák az elkészítendő rendszert. A terv dokumentációja védi a megrendelőt és a fejlesztőt is. A megrendelő ebben a dokumentumban tudhatja meg véglegesen, hogy mit is fog kapni a fejlesztési folyamat végén, míg a fejlesztő ebben a dokumentumban rögzíti le, hogy mit is kell neki nyújtania a fejlesztés során. A dokumentációnak tartalmaznia kell a következőket:

A megkívánt rendszer bemenő és kimenő paraméterei.

Milyen forrásadatok alapján, milyen listák, nyomtatási képek jelennek meg.

A rendszer összes funkciójának leírását

A fejlesztés során előre el kell tervezni, hogy milyen funkciókra akarjuk alkalmazni a rendszert. A fejlesztés folyamata közben az utólagos igények kivédésének ez a legmegfelelőbb helye.

Az adatszerkezeteket.

A felhasznált adatszerkezetek eleve meghatározzák, hogy milyen adatok nyerhetők ki egyszerűen a rendszerből. Az adatszerkezetek megtervezése a rendszer későbbi használhatóságának alappillére. Ha nem kellően átgondolt a rendszer, vagy kevés adat nyerhető ki belőle vagy túlságosan bonyolult vagy nem kellően rugalmas. A rendszerhez és a vele támasztott követelményekhez illeszteni kell az adatszerkezeteket. Arra is gondolni kell, hogy ha a rendszernek további fejlesztési irányai is lehetnek, akkor azokra a fejlesztésekre is gondolni kell, azaz biztosítani kell az adatszerkezetek olyan kiterjesztését, amelyeket a jelenleg kifejlesztett rendszerek továbbra is használhatnak, de a később kifejlesztett rendszer is felhasználhatja a jelenlegi adatszerkezetek adatait.

Az esetlegesen előrelátható további fejlesztések valamiféle kereteit.

A tesztadatok és az éles adatok felvitelének módját.

A tesztelés módját, a helyesség megállapításainak kritériumait.

A felhasznált fejlesztőeszközt, és a fejlesztőeszköz hardverét

A futtatórendszer hardver és szoftverkritériumait.

A szükséges hardverkonfigurációt, ami a processzor típusa, memória, szükséges terület a HDD-n, kell-e floppy vagy nem, és ha igen, akkor milyen, a megjelenítő minimális felbontását, egyéb kiegészítő eszközöket. A szoftvernél a szükséges operációs rendszert, és minden egyéb kiegészítő szoftvert, ami a futtatáshoz szükséges. Célszerű megjelölni a minimális és az optimális konfigurációt is.

Hálózati alkalmazásoknál, a szükséges hálózati feltételeket, (Hálózati operációs rendszer típusa, kapcsolódás típusa, stb…)

A tervben indokolni kell, a fenti szoftver-, és hardverválasztásokat.

A tervnek tartalmaznia kell a megvalósítás ütemezését, szakaszokra bontva, határidőkkel, felelősökkel. Mindenképpen kell tartalmaznia a kapcsolattartó nevét.

A tervben le kell fektetni, hogy a fejlesztés során előre nem látható módosításokat a felek hogyan kezelik, azaz ki a felelős, ki dönti el, hogy az esetleges módosítás mennyiben elfogadható és hogy annak a költségeit hogyan viselik. Azt is célszerű leírni, hogy nem teljesen a leírások szerinti működés vagy a fejlesztés csúszása hogyan befolyásolja az egész tervezetet.

10.1.3 Megvalósítás

Egy program fejlesztése programozói munka. A programozónak a fejlesztés megkezdésekor szüksége van minden keretfeltételre, amely a munka végzését segíti. A tervezés során már mindent tud, ami a rendszer bemeneteit, kimeneteit illeti. A tervben megadott eszközökkel, minél gyorsabban, minél hatékonyabban el kell készítenie a kívánt rendszert. Egy üzleti céllal kifejlesztett programban nagyon fontos, hogy a program szerkezete jól strukturált legyen, azaz célszerű a rendszert a tervezetben meghatározott módon, logikus sorrendben fejleszteni. A fejlesztés sorrendiségét több dolog meghatározhatja, ízlés kérdése, hogy ki milyen módszert követ.

A harmadik generációs rendszerekben a fejlesztés központi része egy jó editor, debugger és profiler rendszer. Ezeket a modulokat az integrált IDE rendszerek tartalmazzák. Az ilyen rendszerek nem támogatják közvetlenül a strukturált programtervezést, bár esetleg maga a nyelv kínálja a struktúrákat. Az ilyen fejlesztőeszközök esetén a programozónak magának kell gondoskodnia a fejlesztés során a hatékonyságról, azaz az adatszerkezetek logikus létrehozásáról, a változónevek következetes és áttekinthető használatáról, az eljárások és függvények következetes használatáról. Célszerű az ilyen fejlesztőeszközök használatakor gondoskodni a különböző programrészek külön modulokba szervezéséről, az egyes modulok közötti interface-ek szabványos együttműködéséről.

A fejlesztést például célszerű úgy szervezni, hogy:

Az adatszerkezetek létrehozása, az adatokat közvetlenül kezelő programrészek létrehozásához kapcsolódjon.

Újrafelhasználható kódot írjunk. Ez nem egyszerű, meglehetősen sok általánosítást tartalmazhat, ami plusz munkát jelent, de az elején befektetett munka később sokszorosan megtérül. Ha léteznek már előre elkészített standardizált modulok, akkor azokat megfelelően össze kell gyűjteni, a konkrét feladat szerint esetleg módosítani. Ha az ilyen rendszereket módosítjuk, akkor figyelni kell az esetleges más programokban lévő kapcsolatokra is. A visszafelé érvényes kompatibilitás nagyon fontos szempont.

Célszerű már az elején kidolgozni azokat az eljárásokat, amelyek az adatok tesztelésével kapcsolatosak.

A következő lépésben például el lehet készíteni a program fő menüpontjait

A modern 4GL fejlesztőeszközök kínálják a Top-Down módszer alkalmazását, gyakorlatilag másképpen nem is lehet őket hatékonyan használni. Ezekben az esetekben a fejlesztés sorrendjének célszerűen a képernyő, a listaformátumok, a menük megtervezésével kell kezdeni, majd a módszert követve kialakítani az egyre alacsonyabb szintű programrészeket, az egyes képernyőobjektumok tulajdonságainak meghatározásával (prototípus készítése).

Az egyre jobban kidolgozott részletek természetesen állandó kontrollt igényelnek. Minden egyes elkészült programmodult a lehetőségekhez képest azonnal ki kell próbálni, tesztelni kell. A felmerülő hibákat ki kell javítani. Ebben a szakaszban nagy hasznát veszi a fejlesztő, ha a tervezés során és a fő modulokban megfelelően strukturálta a programját, azaz csak a megfelelő interface-eken keresztül kommunikálnak az egyes programrészek. A hibakeresés is könnyebb így.

A munka során célszerű a legkritikusabb részeket megírni először. Azokat lehet ezeknek tekinteni, amelyek az adatok bevitelét, az adatok kezelését, az adatok helyességét biztosítják.

A későbbiekben az adatok megjelenítéséért felelős részek kerülhetnek sorra. Ezek a részek általában nem hatnak vissza az adatokra, azokat nem módosíthatják, tehát elrontani sem tudják.

A fejlesztés során gyakran jó szolgálatot tesznek az üres eljárások, procedúrák. Ezek olyan meghívható eljárások, amelyek a program későbbi szerkezetében benne lesznek, funkciójuk lesz, de a fejlesztésnek a korai szakaszaiban még nem kell őket kidolgozni. Általában egy megjegyzést meg lehet jeleníteni bennük, esetleg a később feldolgozandó paramétereket is át lehet adni nekik, de a kidolgozásukra majd csak később kerül sor.

A felhasználót folyamatosan tájékoztatni kell a fejlesztés eredményeiről, akár úgy is, hogy tesztváltozatokat hagyunk nála.

A legvégén a kényelemmel kapcsolatos még ki nem elégített igényeket célszerű sorra venni. Itt gondolunk elsősorban a Help, a gyorshelp, a gombok megfelelő feliratairа. Meg kell jegyezni, hogy a 4GL nyelvek ezekben is igen nagy szolgálatot tesznek, mivel az adatok tervezése során meg kell adni az adat típusán, méretén kívül elnevezést, magyarázatot stb…

Amikor úgy gondoljuk, hogy észen vagyunk a programunkkal, akkor egy módszeres teszten kell átesnie a programnak, amely minden funkcióját módszeresen végigteszteli. Ebben a szakaszban általában fény derül korábban észre nem vett hibákra is. Vannak hibakeresési, tesztelési módszerek, amelyeket a későbbiekben fogunk tárgyalni. Segítségükkel meg lehet találni a hibák nagy részét. A megtalált hibákat ki kell javítani, majd a javítás után újra tesztelni a megfelelő részeket.

A helyesen megtervezett programok egyes moduljai függetlenek, így az egyik részben keletkezett hibák javítása nem hat a más részekre. A nem eléggé meggondoltan tervezett programokban előfordul az, ahogy az egyik rész hibájának kijavítása más részekben okoz hibát. Az ilyen programok hibamentesek soha nem is lehetnek. Minél később derülnek ki az ilyen függőségek, annál nehezebb a programot normális állapotba hozni.

Ha nem találtunk hibát, akkor sor kerülhet egy elsődleges hatékonyságvizsgálatra. Itt vizsgálni kell, hogy az éles adatok mennyiségéhez mérhető adatok esetén a futtató gépeken milyen teljesítményt produkál a rendszerünk. A hatékonysági problémák oka akár az algoritmusok, akár az adatszerkezetek szintjén is kereshetők. Meg kell keresni a nem megfelelő hatékonyság okát, és a megfelelő beavatkozással meg kell megszüntetni. Természetesen utána újabb tesztperiódus következik.

Előbb – utóbb a fejlesztő nem talál több hibát. Ez nem azt jelenti, hogy nincs is. Ennek pszichológiai okai vannak. Hiába törekszik valaki módszeresen dolgozni, de a fejlesztés során a figyelmét elkerülő részletek általában később is elkerülik a figyelmét. Ilyenkor a külső tesztelők találhatják meg a hibát. A felhasználó megkapja a program tesztváltozatát.

A szakzsargonban **Alfa változatnak** hívják a még alig tesztelt változatokat, illetve azokat, amelyeket csak maga a fejlesztő tesztelt és még fejleszteni akar rajta. **Béta változatnak** nevezik azt a programot, amelyben a fejlesztő már nem talál hibát és kiadja külsősöknek tesztelésre. Ezekben a programokban a fő funkciók már eléggé megbízhatóan működnek, de korántsem tekinthető befejezettnek a program semmilyen szempontból.

(Állítólag a COREL cég vezette be a nemzetközi szoftveres köztudatba a Bétatesztelés fogalmát, mert volt idő, amikor a szoftverei az eladás után fél évvel kezdtek csak használhatókká válni, sok javítókészlet, és patch – olyan kisebb terjedelmű javítás, amely egy-két modult lecserél, esetleg a program bináris formájában kicserél néhány gépi kódú utasítást - kiadása után.)

10.1.4 Javított változat

A felhasználó tesztelése közben elkészítjük a felhasználói dokumentációkat, majd a hibalista kézhezvétele után megkezdjük a hibajavítást, majd újabb hibakeresési, tesztelési, esetleg hatékonyság-vizsgálati eljárás következik. Az így tesztelt programot újra átadjuk a Felhasználónak tesztelésre.

A fenti ciklusok addig folytatódnak, amíg a felhasználó ki nem jelenti, hogy a program a tesztadatokkal jól fut. Ekkor kell kipróbálni az éles adatokkal, majd ha azokkal is hibátlanul fut, akkor a programot átadhatónak lehet tekinteni, és a Fejlesztő átadja, a Felhasználó pedig átveszi.

10.1.5 Végleges változat, és továbbfejlesztés

A végleges változatot a Felhasználó üzemszerűen használja, majd egy idő múlva csak azt használja. A program használata közben persze kiderülhetnek újabb hibák is, amelyek súlyuktól függően hibás működést is eredményezhetnek vagy csak szépséghibának tekinthetők. Általában fél év üzemszerű használat kell ahhoz, hogy egy komolyabb fejlesztésről kiderüljön, hogy mindenben megfelel és hibátlan.

Gyakran fél-egy év használat során derülnek ki azok a hiányosságok, amelyek a programok továbbfejlesztését indokolttá teszik. A továbbfejlesztés során biztosítani kell az addigi verzió üzemszerű működését, illetve egy korábbi stabil változathoz való visszatérés lehetőségét. Ennek megfelelően az adatszerkezeteket módosítani csak nagyon indokolt esetekben szabad. A továbbfejlesztést azonban úgy kell tekinteni, mint egy új projectet, igaz a feladat általában nem olyan mértékű, mint korábban, és a körülmények is tisztábbak.

10.1.6 Verziókezelés

A verziók elnevezése során nincsen egységesen elfogadott gyakorlat, de az alábbiak jól használhatók. A programnak van fő verziószáma, és al-verziószámai.

A **fő verziószám** akkor változik, hogyha a program egészére kiható lényeges változást építettek be a programba. Ilyenkor gyakran a program külső kinézete is változik.

Az **al-verziószámot** akkor változtatják, ha kisebb módosításokat hajtottak végre a programban, amely egyes tulajdonságokat módosított.

Ha csak hibajavításokat tartalmaz az új verzió, akkor gyakran az al-verziószám további része változik vagy egy fordítási sorszám változik, vagy egy dátum változik, amely a verziószám részét képezi. Ezt általában **patch**-nek (=folt) hívják.

Ha egy szoftverben az alapvető tulajdonságokat generálisan módosítják, akkor az gyakran nem hibajavításnak, hanem úgynevezett **Service Pack**-nak hívják.

Néhány példa

Manapság a nagyobb szoftvergyártók egész egyszerűen egy évszámmal jelölik szoftvereiket (Microsoft Office 2007), de gyakran még az előző évben jeleni meg a következő évi azonosítóval ellátott szoftver ☹.

A Linux rendszereken a kernel verziószámában a páratlan verzió mindig kísérleti állapotot jelent, míg a páros al-verzió a stabil kernelt jelenti.

2.3a – Jelenti, hogy egy szoftver 2 verziójának 3 változata, amiben valami kis javítást még végeztek

Gyakran mellékelnek a szoftverekhez mellékelnek egy history jellegű leírást (különösen a netről letölthetők esetén), amely leírja, hogy az egyes verziókban mit javítottak, módosítottak. Ekkor a felhasználó eldöntheti, hogy a módosított verziót érdemes-e, kell-e neki letölteni és feltelepíteni.

10.2 A megvalósítás gyakorlati eszközei

Az algoritmusok általában még nem eléggé egzaktak, hogy valamilyen automatizmus feldolgozza őket, ezért azokat át kell ültetni egy programozási nyelv kereteibe.

A **programozási nyelveknek** előre lefoglalt szavai vannak. A lefoglalt szavak listáját **utasításkészletnek** hívjuk. Az utasításokat csak bizonyos formai szabályok betartásával használhatjuk, ezeknek a formai szabályoknak az összességét hívjuk a nyelv **szintaktikájának** neveznek. A szövegeket ezek alapján értelmezi és fordítja le a fordítóprogram, úgynevezett **gépi kódra**. A gépi kód az a számsorozat, amelyet a számítógép processzora értelmezni tud és végre tud hajtani.

Például:	*MOV AX,134*	*vagy*	*Gotoxy(11,13);*
	ADD AX,75		Writeln('Üdvözlöm Önöket!');

A bal oldali oszlop két sora a számítógép memóriájában 8 byte-nyi helyet foglal el, összead két egész számot, és egy memóriahelyre leteszi őket. Az egyszerűbb felhasználói programok is minimum néhány ezer byte hosszúak, azaz ilyen utasításokból jó néhány száz kell. Ezt hívják assembly programozásnak. Lassú és nehézkes, de a gép erőforrásait így lehet a legjobban kihasználni.

A jobb oldali oszlop két utasítása kiírja a képernyő 13. sorának 11. oszlopától kezdődően, hogy Üdvözlöm Önt. Ez is két soros program, de valami értelmeset is végrehajt.

10.2.1 Compiler, Interpreter, P-kód, Virtual Machine

Egy tetszőleges nyelven megírt program lehetséges, hogy szintaktikailag helyes utasításokat tartalmaz, de nem biztos, hogy a program értelmes dolgokat művel. A program gondolatmenetének hibátlanságát úgynevezett **szemantikai szabályokkal** ellenőrzik. Általában a rendszerek nem nagyon figyelnek oda a szemantikai szabályokra, de vannak olyan rendszerek, amelyek felismerik, ha a fejlesztő szintaktikailag helyes, de szemantikailag értelmetlen utasításokat ír le.

10.2.1.1 Compilerek

A **compilerek** a futás előtt ellenőrzik a programkódot szintaktika és szemantika szempontjából, majd gépi kódra fordítják a programot. Ha hibát észlel kijelzi a fejlesztőnek, megjelölve a megfelelő helyet a programszövegben. A szemantikai szabályoknak való megfeleltetést általában nem jelzi kritikus hibáknak a fordító (Warning), azaz a program ezektől még lefordítható. Gyakran előre kifejlesztett módszerek segítségével a program képes a végső futtatható változatot optimalizálni is.

A forrásszöveget a compiler úgynevezett object formátumba fordítja le, vagy más néven **object kód**ba. Az object kód az Intel által szabványosított köztes állapot a forrásszöveg és a gépi kód között. Az így keletkezett fájl a DOS - Windows rendszerekben a .OBJ kiterjesztést kapják. Ebben az állapotban az úgynevezett linker – szerkesztő – programok veszik át a további irányítást és az egyes programrészek .OBJ fájljaiból, továbbá az előre elkészített LIB fájlokból összeszerkeszti a végleges futtatható programot. A LIB fájlok szabványosan tartalmaznak több OBJ fájlt egységbe csomagolva. A LIB fájlok szerkesztését a Microsoft rendszerek esetén a LIB.EXE a Borland eszközöknél a TLIB.EXE végzi.

A grafikus alkalmazások esetén még egy lépés zajlik le, a programokhoz hozzá kell szerkeszteni a képernyőn megjelenő objektumok forrását is. Ezt a Resource Compiler programok végzik el.

A compileres rendszerek tagadhatatlan előnyei a következők:

Mint látható, a compileres rendszerek a fejlesztés során fordítják le és tesztelik a forrásszöveget. Mivel ezek a rendszerek a fordítás során áttekintik az egész rendszert is, ezért a szintaktikai, szemantikai hibák többségét képesek kiszűrni az ilyen rendszerek.

A keletkezett gépi kód a processzor típusára optimalizált lehet.

A fordítónak meg lehet adni, hogy az elkészült gépi kód milyen ellenőrzéseket végezzen a futás közben. Memória verem ellenőrzés, memóriatúlcsordulás ellenőrzése, tömbhatárok ellenőrzése, adattípus megfeleltetés, stb – ilyen típusú ellenőrzések jöhetnek szóba, ezeket a fejlesztő kívánsága szerint befordítja a compiler a programba. Amikor a program már végső tesztelt, hibátlan állapotba kerül, akkor a fordítási opciókat átállítva a program a lehető legkisebb és leggyorsabb lehet.

Hátrányok is vannak persze:

A compileres rendszerekben a szerkesztés, fordítás, tesztelés fejlesztési ciklus akár nagyon hosszú is lehet, hiszen a fordításkor a rendszer áttekinti a teljes fordítani valót. Persze léteznek olyan eljárások, amelyek az egyes modulok lefordításának, a forrásszövegek időpontjának elemzéséből meg tudják határozni, hogy mit kell újrafordítani és mit nem. Az újraszerkesztést, azonban általában nem lehet megspórolni. Lassabb gépeken egy teljes fejlesztési ciklus sok időt vehet igénybe.

Nem lehet párbeszédes üzemben működtetni ezeket a rendszereket. A program futását megszakítva általában nem lehet folytatni a programot ugyanattól a ponttól, ha közben megváltoztattuk a program állapotát.

Bár vannak hibakereső programok, de nem feltétlenül használhatók minden körülmények között.

A Microsoft Visual programok, C#, Borland Pascal, Borland C++, Delphi compileres rendszerek.

10.2.1.2 Interpreteres rendszerek:

Klasszikus ilyen rendszer a BASIC programozási környezet és nyelv. A rendszer lényege, hogy a program bevitelekor a rendszer egy szintaktikai ellenőrzést végez a bevitt utasításon. A program végrehajtása során a rendszer utasításonként elvégzi a következőket:

Beolvassa a következő utasítást, és eldönti, hogy az utasításkészlet melyik utasítása.

Ellenőrzi, hogy az adott utasítás szintaktikailag helyes-e, az átadott paraméterek típusa, esetleg mérete megfelel-e az utasítás kívánalmainak

Ha hibátlan, akkor meghívja az utasításhoz tartozó előre elkészített kódrészletet.

Figyeli, hogy a végrehajtás során nem jön-e létre valamilyen hiba. Ha hibára fut a rendszer, akkor hibakódot kell generálnia. Ha hibátlan a végrehajtás, akkor veszi a következő utasítást.

Hátrányok:

A fejlesztő dolga annak megállapítása, hogy szemantikailag helyes-e a program.

A program optimalizálása automatikusan semmilyen formában nem jön létre.

Az így futtatott program sokkal lassabb, mint a megfelelő compilerrel lefordított gépi kódú program.

A program futtatásához speciális futtatókörnyezet szükséges, amelynek tartalmaznia kell minden olyan segédállományt, amely az esetleg előforduló összes utasítás megfelelő gépi kódját tartalmazza. Ez a program méretét feleslegesen megnöveli, mivel olyan modulok is bent vannak a memóriában, amelyekhez tartozó utasítást nem is hajt végre a program. A programok mérete nagyobb, mint a megfelelő compilerrel fordított programoké.

Az ilyen programok memóriaszervezése bonyolultabb, mint a compileres társaiké.

Előnyök:

Az interpreteres rendszerekben a fejlesztési ciklus lerövidül, gyorsan lehet dolgozni.

Az elkészült programrészleteket gyorsan ki lehet próbálni, tesztelni.

Kisebb teljesítményű gépeken is ugyanolyan gyors a program, mint a nagyobbakon – (azaz ugyanolyan lassú)

Basic, Javascript, PHP, Minden script alapú nyelv...

10.2.1.3 P-kódú és Virtual Machine alapú nyelvek

Ez egy öszvér megoldás. A fejlesztő rendszer compilere és linkere futtatható programot fordít, de a program bizonyos részei un. P-kóddá fodítódnak, ami átmenet a gépi kód és a programszöveg között. Ezt a kódot futás közben értékeli ki a rendszer. Önállóan futtatható EXE fájl az eredmény, de nem optimális a program sebesség és méret szempontjából. Az ilyen rendszerek igyekeznek a korábbi két rendszer előnyeit ötvözni. Ilyen a **Clipper**.

A multiplatformos programozás előtérbe kerülése miatt alakultak ki a Virtual Machine alapú nyelvek. (Java)

Ebben az esetben a fordítóprogram olyan kódot generál, amelyet minden platformon a megvalósított megfelelő futtatókörnyezet dolgoz fel és fordít le az adott platform által használt gépi kódra. Compileres megoldásról van szó. A kulcsmomentum a hordozhatóság!

10.2.1.4 Mikor melyiket

A gyakorlatban a gépek teljesítményének növekedésével a compileres rendszerek előtérbe kerültek. Vannak olyan feladatok, amelyeket nem lehet interpreterrel megoldani, míg más feladatok megoldása csakis azzal képzelhető el.

A Pascal, a C, C++ nyelvek és más harmadik generációs nyelvek compileres rendszerekként valósulnak meg, míg a BASIC, Visual BASIC régebbi változatai, illetve a Microsoft Office termékcsaládjának mindegyik darabja interpreteres rendszer. A Word az Excel vagy az Access programozása interpreter közreműködésével jön létre.

A Clipper P-kódot hoz létre. A rendszer ezt a köztes kódot nem ellenőrzi futás közben szintaktikai és szemantikai szempontból. Csak a memóriakezelés, a változók értékhatárait ellenőrzi. Az így keletkezett program végrehajtási sebessége valahol a compiler és interpreter nyelvek között van, mérete is, mivel a p-kódra fordításkor csak azok a modulok kerülnek be a futtatható file-ba, amelyek benne vannak a fordított rendszerben.

A Java pedig a hordozhatóság miatt VM-et használ.

A script nyelvek – Action Script, Javascript, PHP – mind egy futtatókörnyezetet igényelnek, amelyek vagy az operációs rendszerre épülnek, vagy esetleg egy böngészőhöz kapcsolódnak.

10.2.2 A programozási nyelvek szintjei

Alacsonyszintű nyelvek - assembly

Ebben a nyelvben egy gépi kódú utasítás egy assembly utasításnak felel meg. A programok fejlesztése nehézkes, viszont a gép erőforrásait maximálisan ki lehet használni, a program a lehető leggyorsabb és a legkisebb helyet foglalja el. A mikroprocesszor típusától függően más gépfajtán más a nyelv is!

Középszintű nyelvek C, C++

Egy utasításhoz néha egy gépi utasítás, néha több tartozik. Viszonylag jól lehet kihasználni a gép adta lehetőségeket, de furcsa módon a C programok általában gépfüggetlenek. Van olyan program, amit a különböző gépfajtákra elkészítettek, mégis az eredeti programszövegük 70-80 %-ban azonos, mert C-ben írták őket. Ez assembly nyelven nem lehetne. Gyors a fejlesztés, a programok rövidek, gyorsak.

Magas szintű nyelvek - Pascal, dBase, Clipper, Fortran, BASIC

Ezeknek a nyelveknek az utasításai egészen összetett tevékenységeket hajtanak végre. A programok fejlesztése gyors, a gép tulajdonságait is ki lehet használni, mivel a nyelvekhez előre megírt programkönyvtárak állnak rendelkezésre. A programok mérete nagy, legalább néhányszor tíz kilobyte.

Nagyon magas szintű fejlesztőeszközök

Az utóbbi években elterjedtek olyan fejlesztőeszközök, amelyek segítségével gyorsan összetett alkalmazásokat lehet létrehozni, akár programozói tudás nélkül is. Ezeket 4GL (4. Generation Language - Negyedik generációs) nyelveknek hívják. Visual Basic, Clarion, ReMind, Magic, Borland Delphi, Borland C Builder, Visual FoxPro, CA Visual Object stb.

Az elkészült alkalmazás természetesen nem a leggyorsabb és legkisebb lesz, amit csak ki lehet találni, de gyorsan elkészül és könnyen, egységesen felhasználható, mivel nem a programozó tudásán múlik ez a tulajdonsága.

Megjegyzések:

A robot vezérlése alkalmas arra, hogy segítségével elsajátítsa valaki az algoritmikus gondolkodás elemeit. Az algoritmusok minden eleme felfedezhető a robot tevékenységében.

Amikor a robot egymás utáni lépéseket tesz meg a vezérlő programban utasítások sorozatát kell kiadni.

Amikor figyelni kell egy tárgyat vagy képet és össze kell hasonlítani dolgokkal, akkor az elágazás vezérlési szerkezetet használjuk.

Amikor oda-vissza járkál, akkor ciklus vezérlési szerkezetet kell használni.

Az egyre bonyolultabb tevékenységeket részegységekre bontva eljárásokat írunk. A látás, információ bevitelével egyenlő, a beszéd pedig információ kivitel, azaz input-output műveleteket is használunk.

A C nyelvet B. Kernighan tervezte, mikor egy operációs rendszer kifejlesztésére kapott megbízást. Az assembly helyett írt hozzá egy fordítót, majd segítségével megírta a UNIX operációs rendszer egy változatát. A nyelv a professzionális fejlesztők között népszerű lett. Ennek továbbfejlesztése az objektum orientált nyelv a C++. Vegyes nyelvű programozás esetén az assembly-n kívül C-t szoktak használni. Általában más nyelvek elfogadják társnak, amikor vegyes nyelven írnak egy programot.

A Pascal nyelvet eredetileg N. Wirth a Zürich Műszaki Egyetem tanára oktatási célra tervezte. A Borland cég Turbo Pascal-jával lett világhírű a nyelv, ami a szabvány Pascal-nak a kiterjesztése. Mára általános célú nyelvként tetszőleges programozási feladatra használják.

A BASIC nyelv a hetvenes években lett népszerű a játékgépek miatt. Könnyű vele működő programot írni, ezért a kezdők előszeretettel használják. Soha nem szabványosították. A Microsoft által továbbfejlesztett Visual Basic az egyik leggyakrabban használt windowsos fejlesztőeszköz. A DOS is tartalmaz egy QBASIC nevű változatot.

A DBASE programozási nyelve kifejezetten adatbázis-kezelési feladatok megoldására való és a DBASE adatbázis-kezelő rendszerben lehet vele programokat írni és a rendszer segítségével futtathatók. Eredetileg ennek a fordítóprogramjaként született meg a CLIPPER programozási nyelv, ami mára önálló életet él és hatékonyabb mint az eredeti. Segítségével az alkalmazások önálló .EXE fájlként futtathatók. Professzionális nyelv.

10.2.3 A programozási nyelvek másik féle osztályozása

10.2.3.1 Procedurális nyelvek

A programozási nyelveket szokás más módon is osztályozni. A legelterjedtebb és leggyakrabban használatos nyelvekben – Pascal, C, C++, Basic, stb. – van egy közös momentum. Ezek úgynevezett **procedurális-,** vagy más néven **Neumann elvű programozási nyelvek.** Közös bennük, hogy a programot egyfajta tevékenységek sorozataként képzelik el, ami mellesleg a leggyakrabban valóban a leghasznosabb is. Tulajdonságaik:

Vannak változóik, vannak adatszerkezeteik.

A program és a memória fizikailag közös memóriában helyezkedik el, azaz a program állapotát a memória állapota egyértelműen meghatározza. (Ha kimentjük a memóriatartalmat, majd kikapcsoljuk a gépet és bekapcsolás után visszatöltjük a memóriatartalmat, akkor a program ugyanonnan fut tovább)

Az adatok beolvasása és kiírása a memóriában történő adatmásolás eredménye.

Vannak vezérlési szerkezeteik.

A programnak mindig van eleje, a programozó által megszabottan a program minden pillanatban ismert helyen fut, és a megadott eljárások lefutása után a program véget ér.

Sajnos ezek a nyelvek bizonyos feladattípusok megoldására nem mindig célravezetők.

10.2.3.2 Automata elvű programozási nyelvek

Képzeljünk el egy olyan esetet, amikor egy ipari robotot kell vezérelni egy számítógéppel. A robot állapota, helye, sebessége és még néhány paramétere a fontos. Az ilyen robotokat vezérlő számítógépek általában nem nagy teljesítményű PC-k, hanem a gép képességeihez tervezett fedélzeti számítógépek, a megfelelő célorien-

tált nyelvvel. Az ilyen programozási nyelveket **automata elvű programozási nyelveknek** hívják. Tulajdonságaik:

A program és a vezérléshez szükséges adatok elkülönülten vannak a számítógép különböző célú memóriaterületein.

- Az ilyen nyelveken nincsen változó, nincsen értékadás.
- Az ipari berendezés állapota egyúttal a program állapotát is jellemzi. A program változtatja a robot állapotát.
- A külvilág felől a program paramétereken keresztül kapja meg az indításhoz szükséges adatokat.

Mivel csak bizonyos célok végrehajtására hozzák létre a robotot, általában az ilyen rendszer csak adott típusú adatokkal tud dolgozni. Ennek a programozási nyelvnek a tipikus példája a LOGO programozási nyelv.

10.2.3.3 Függvényelvű programozási nyelvek

A harmadik nagy nyelvcsaládot a **függvényelvű programozási nyelvek** alkotják. Az ilyen nyelveken a programot függvényként képzelik el. A program futtatása során a programot alkotó függvényt értékeljük ki. Ezeknek a nyelveknek a tulajdonságai:

Nincsenek változók, de a függvényeknek vannak paramétereik és van visszatérési értékük.

Van feltételes elágazás, hiszen egy függvényt lehet úgy definiálni, hogy egy feltétel esetén egyfajta definíciója legyen, míg ha a feltétel nem igaz, akkor más definíció legyen értelmes.

Van rekurzió, de nincsen ciklus, hiszen mint majd később látjuk minden ciklikus eljárást át lehet fogalmazni rekurzív eljárássá.

Az adatbevitel egy függvény paraméterátadásával valósul meg, az adatok kiírása a függvények eredménye.

10.2.3.4 Logikai nyelvek

A programozási nyelvek egy speciális fajtáját jelentik a **logikai nyelvek**. A logikai nyelvek szerint a program egy kiértékelendő logikai formula. A program végrehajtása a formula kiértékelését jelenti. Tulajdonságai hasonlóak a függvényelvű programozási nyelvekhez, csak egy lényeges különbség van. Létezik bennük a visszalépéses keresés, a **backtrack** eljáráson alapuló kiértékelési rendszer.

Ennek hatására az ilyen nyelveken olyan kérdéseket lehet feltenni, hogy bizonyos alapadat halmazra feltesszünk kérdéseket, és a program megadja a választ, hogy a kérdésre adott válasz igaz vagy hamis. Ezt oly módon teszi meg, hogy végigpróbálja az adathalmaz összes lehetséges állapotát, megvizsgálja, hogy a kérdés milyen választ ad, és ha igaz a válasz, akkor az adathalmaz pillanatnyi állapotát adja vissza eredménynek.

A logikai nyelvek családjának legelterjedtebb tagja a **Prolog** programozási nyelv.

Ha egy logikai nyelven meg lehet oldani, hogy a program a feltett kérdésekre adott válaszokat megjegyezze, akkor a program új adatok birtokába jut, az bővül azoknak az adatoknak a köre, amelyekre új kérdéseket lehet feltenni. Az ilyen nyelven megvalósított programok tehát egyre bővülő tudásbázisú programok — tanuló programok. Valójában ezek a programok a feltett kérdéseket és a rájuk adott választ programkód formájában elmentik, és az újabb futtatáskor az új kérdésre adott válasz már a rendszer adatai között van.

A **LISP** programozási nyelv alkalmas az ilyen feladatok megoldására. Egyetlen hibája, hogy nem compileres, hanem interpreteres nyelv, ennek megfelelően igazán nagy méretű programok nem futnak rajtuk elfogadható sebességgel. A Magyar Tudományos Akadémia mesterséges intelligencia kutatásával foglalkozó kutatói használják a fent felsorolt logikai típusú programozási nyelveket.

10.3 Programkódolás

A programozási feladatok megoldása során eddig nem vettük figyelembe az egyes programozási nyelvek speciális tulajdonságait. Sajnos az algoritmus-leíró nyelveken elkészített programokat még a kiválasztott programozási nyelveken kódolni kell. A kódolás során szembetalálkozunk az egyes programozási nyelvek eltérő tulajdonságaival, nem csak a szintaktikus szabályok, hanem a szemantikai szabályok terén is.

Eleve az algoritmus-leíró nyelvek igazából procedurális nyelveknek tekinthetők, azaz az algoritmus kódolása egy valódi programozási nyelvre legkönnyebben ilyen típusú nyelvekre megy.

10.3.1 Programozási tételek használata

A programozási tételek a felhasznált nyelvtől függően más és más alakot ölthetnek, amikor egy konkrét nyelven megvalósítjuk őket. Ennek megfelelően a programozási nyelvek leírásánál kerültük a nyelvfüggő részek használatát. A programozási tételek ismerete elősegíti, hogy bizonyos típusfeladatok megoldásánál felismerve a megfelelő programozási tételt, a programozó sokkal gyorsabban kódolja le az algoritmust.

Szűkebb memória vagy háttértár esetén a programozási tételeket nem lehet teljes mértékben használni. Ebben az esetben ragaszkodva a tételek lelkéhez, kerülő utakat kell keresni. Például, ha túl nagy annak a tömbnek a mérete, amin munkát akarunk végezni, akkor a tömböt helyettesíthetjük egy véletlen elérésű fájllal is, ahol a fájl rekordjai helyettesítik a tömb elemeit. Természetesen a program végrehajtási sebessége már jóval lassabb lesz, mintha az adat csak a memóriában foglalna helyet, és a kód is bonyolultabb lesz, de megfelelően hajtja végig a nagyobb adathalmazra is a kellő műveleteket.

Ha egy programozási nyelv nem támogatja a rekurziót, akkor a tanult módon a rekurzív algoritmusokat át kell és lehet írni ciklusokat tartalmazó formába.

10.3.2 Egyes programozási nyelvek eltérő kódolási lehetőségei, módszerei

A különböző programozási nyelvek rendkívül sokféle lehetőséget biztosítanak arra, hogy ugyanazt az algoritmust hány féleképpen lehessen kódolni. A továbbiakban a programozási nyelvek olyan eltérő tulajdonságaira szeretnénk felhívni a figyelmet, amelyek az algoritmus-leíró nyelven megírt programok kódolásánál problémákat okozhatnak.

Névadási szabályok

Az algoritmusok írásakor egy alapvető szabály, hogy beszédes azonosítókat használjunk. Az eltérő nyelvek az azonosítók számára eltérő előírásokat adhatnak.

Gyakori a név hosszának maximalizálása. A BASIC programozási nyelv régebbi kiadásai, de a Standard Pascal, Assembly és a C régebbi implementációi maximalizálták a név hosszát. A BASIC esetén ez két karakter volt, a Pascal és a C esetén nyolc karakter. A Clipper programozási nyelv tíz karakterben maximalizálja az azonosítók hosszát. Manapság a Visual Basic, a Borland Pascal, és a C is legalább 20 karaktert engedélyez az azonosítók használatára.

Kisbetű – nagybetű. A Pascal, a BASIC, a Clipper közömbös a betű írásának módjára, nagybetűre konvertálja belsőleg az összes azonosítót, míg a C/C++, Assembly, Java, Javascript, PHP és Visual Basic következetesen elkülöníti a kisbetűs és a nagybetűvel írott azonosítókat. Ez sokszor igen nehezen megtalálható hibaforrás, ha a programozó sokat használta a Pascal nyelvet.

A BASIC nyelv korábban a változók definiálásakor a változó nevével megadta annak típusát is. Nem volt arra lehetőség, hogy tetszőleges nevet adhassunk a változónak. Kódoláskor sok hibát okozhatott.

Beszédes azonosítókat kell használni. Ha a nyelv engedélyezi, akkor az algoritmusban használt hosszú neveket célszerű használni kivéve, ha interpreteres nyelvről van szó. Ez esetben valamilyen ésszerű kompromisszumot kell kötni a név érthetősége és végrehajtási idejének optimalizálása között. Célszerű betartani néhány jól bevált szokást:

Néhány jól bevélt szokás a változónevek névadásaira

- az abc első betűit paramétereknek használjuk,
- az i,j,k,l… betűket index, illetve segédváltozóknak használjuk,
- az x,y,z.. koordináta kijelölésére alkalmas vagy segédváltozóknak.
- a min, max azonosítókat, illetve ezeknek az összetételeit mindig valamilyen maximális vagy minimális érték kijelölésére célszerű használni.
- a nagybetűs azonosítókat konstansokra használjuk.

Kezdőérték adásának szabálya

Az egyes programozási nyelvek új változók létrejöttekor nem egyforma módon kezelik a változók kezdőértékeit. Különösen igaz ez az alacsony szintű nyelveken. Ha egy tömb kezdőértékeit kiolvassuk C nyelven, valószínűleg nagy meglepetésben lesz részünk. A tömb memóriaszemetet fog tartalmazni. A Pascal, a Basic, a Clipper és még sok más nyelv a létrejött változó kezdőértékének az aktuális típusnak megfelelő 0-át, üres sztringet vagy hasonlót ad, de ebben nem lehetünk 100%-ig biztosak. Ennek megfelelően mindig kell kezdőértéket adni egy újonnan létrejött változónak, akár lokális, akár globális változóról van szó.

Nem minden programozási nyelv támogat minden típusú adatot

A különböző nyelvek más és más mértékben támogatják az eltérő adattípusokat. A kódolásnál ezt figyelembe kell venni.

A lista adatszerkezetet közvetlenül nagyon kevés nyelv támogatja.

Az Assembly nyelvnek természetesen csak a byte illetve az integer, illetve a mutató adattípus létezik. Semmilyen bonyolultabb adatszerkezet definiálására nincsen a nyelvben lehetőség.

A BASIC például a rekordokat, a halmazokat illetve egyéb összetettebb adattípusokat nem támogat,, míg a Pascal, C igen.

A mutatókból felépített adatszerkezetek olyan szerkezetek, amelyek nem minden nyelvben támogatottak (C-ben).

A numerikus típusokra a Pascalnak, a C-nek több, de a Basic-nek csak egy lebegőpontos típusa van.

Az egész numerikus típusok esetén a legtöbb nyelv ad egy célszerűen használható alapértelmezést, a kétbájtos, integer típust. Ha az értelmezési tartományba nem fér be az érték, akkor az adott nyelv lehetőségei között keresni kell egy ilyen típust. A C és a Pascal kínál hosszabb egész típusokat, de BASIC nem.

A pointereket a Clipper, a BASIC, illetve a Visual Basic nem támogatja. Ennek megfelelően dinamikus adatfoglalás sem lehetséges a Visual Basic-ben.

A Pointerek használhatók és használandók a C, C++, Pascal nyelvekben. Nincsenek pointerek a Java, és a Basicben.

Új adattípusok létrehozására nincsen lehetőség több nyelvben, mint például a BASIC, Javascript, PHP vagy a Clipper nyelveken, más esetekben azonban igen.

Csak az objektum-orientált nyelvek támogatják az objektumok használatát. Ennek megfelelően a Borland Pascal, Borland C++, a Microsoft C++, Visual Basic, Java, Javascript nyelvek objektum-orientáltak. A Clippernek van objektum-orientált kiterjesztése (Fivewin), azaz definiálhatók és felhasználható benne objektumok.

A tömbök kezelése

A tömbök kezelése sem minden nyelvben egyforma. A Basic, Pascal, C, C++ nyelvekben előre kell definiálni egy tömb méretét, míg a Visual Basic, Clipper nyelvekben lehetőség van a futás közbeni tömbdefiniálásra, sőt a tömb méretének változtatására is. A Java, Javascript, PHP és Clipperben például új adatot lehet a tömb részévé tenni, régi adatot ki lehet venni, eközben a tömb többi adata megmarad, és a tömb mérete megfelelően változik. A Clipperben és PHP-ban a tömbben nem csak azonos típusú adatok lehetnek. Menet közben is tudunk egy tömb részévé tenni más tömböket. Assemblyben nincsen tömb.

Sztringek mérete

A legtöbb nyelven a stringeknek a mérete korlátos, de nem egyformán. A BASIC, Visual Basic és a Pascal nyelvek csak a maximálisan 255 karakter hosszú stringeket támogatják. A C és a Clipper, PHP stringjeinek mérete korlátlan.

Eltérő láthatósági szabályok

A változók láthatósága nagyon fontos kérdés. A Pascal, a C, a Clipper és a Visual Basic, Javascript, Java, PHP újabb verziói nagyon kifinomult láthatósági rendszerrel rendelkeznek, azonban ezek sem teljesen egyformák. Lényegében azonban az algoritmus-leíró nyelv szabályaihoz nagyon közel állnak

Eltérő feltételvizsgálat.

Az egyes programozási nyelvek a kifejezések kiértékelésekor több féle elvet követnek. Egyes nyelvek a kifejezésben szereplő minden műveletet, függvényt kiértékelnek, míg mások egy kifejezés kiértékelésekor csak addig vizsgálják a kifejezést, amíg az eredmény nem válik egyértelművé.

Az aritmetikai kifejezéseket általában mindig kiértékelik a nyelvek. Sajnos ilyenkor jönnek elő a nullával való osztás, illetve a lehetetlen függvényargumentumok esetei is. Ezekben az esetekben gyakran a programozónak kell olyan kódot írnia, amely kizárja az ilyen eseteket.

A logikai kifejezések kiértékelésénél a nyelvek általában beérik azzal, hogy mihelyt biztos végeredmény, a kifejezés többi elemét már nem vizsgálják.

A kifejezések **precedenciája** (erősorrend – melyik lesz az először kiértékelt rész-kifejezés egy összetett kifejezésben) is hasonló. Ha egyenlő egy kifejezésben több rész precedenciája, akkor célszerűen zárójelezéssel lehet megváltoztatni a kiértékelési sorrendet. Ha nem vagyunk biztosak egy összetett kifejezés kiértékelési sorrendjével kapcsolatban, akkor célszerű használni a zárójeleket, inkább többet, mint kevesebbet.

A kifejezéseket általában a rendszerek balról jobbra értékelik ki. A fordításkor meg lehet adni olyan opciót, hogyha egy logikai kifejezés értéke eldől a teljes kiértékelés vége előtt, akkor ne folytassa a kiértékelést.

Assemblyben az összetett kifejezések kiértékelésére meg kell írni az arra alkalmas kiértékelő kódot.

Tömbhatárok vizsgálata

A programozási nyelvek másként viselkednek, ha az indexváltozók tömbhatáron túlra mutatnak. A legtöbb nyelv ilyenkor futás közbeni hibát generál és a program leáll, vagy hibakezelő ágra fut. A leállás természetesen hiba, amit korrigálni kell. Van olyan eset, hogy az index túlfut a határon, de mégsem okoz a programban valódi hibát, mert a túlmutatás során kapott adat soha sem lesz kiértékelve, és azt az adatot sohasem módosítják. Ebben az esetben a hibaüzenetnek nem célszerű megjelennie.

A Pascal és a C nyelvben van olyan lehetőség, hogy a rendszer olyankódot generáljon, amely ellenőrzi a határokat, de ezt az ellenőrzést ki is lehet kapcsolni. A legtöbb egyéb nyelv szigorúan őrzi a tömbök határait.

Ha bonyolult a kód és nem tudjuk pontosan megállapítani, hogy milyen körülmények között lépi túl a megadott területet a program, akkor a legegyszerűbb megoldás, hogy megnöveljük a tömb méretét annyival, amennyi a túllépés.

Ciklus típusok

A különböző nyelvek nem használják ugyanazokat a ciklus fajtákat. A kódolásnál erre figyelni kell. Az elöltesztelő ciklusokat a (`Ciklus amíg`) a legtöbb nyelv így ismeri, és ugyanúgy használja. A hátulesztelős ciklusoknál a Pascalnak a feltétel kiértékelése pont az ellentéte, mint a többi nyelvnek,

```
Ciklus
   ……
Amíg feltétel igaz
ciklus vége

repeat
   ……
Until nem feltétel
```

A megszámlálásos ciklusnál a legtöbb nyelvben megvan a lépésköz változtatásának a lehetősége, kivéve a Pascalt. Itt a `for` ciklus lépésköze csak +1 vagy –1 lehet.

Rekurzió léte

Nem minden programozási nyelv szereti használni a rekurziót, de a legtöbb nyelven azért meg lehet oldani. A Pascal, a C, a Clipper vagy a Visual Basic, PHP, Javascript, Java közvetlen rekurzióval is elboldogul, de az egyes nyelvek kényesek a rekurzió mélységére. A C, a Pascal esetén lehet állítani a verem méretét, így a rekurzió mélységét megbecsülve könnyen tudjuk alkalmazni azt. A BASIC nyelv nem tartalmaz rekurziót, illetve a változók elhelyezésére nem ad semmiféle módszert.

Az algoritmusok és a programozási nyelvek kapcsolata

Az eddigiek alapján nyilvánvaló, hogy az algoritmusok bár nyelvfüggetlenek, a programkódolásnál nem mindig vihetők át egy az egyben az adott programozási nyelvre, sőt esetenként az adott nyelven előnyösebb más, az eredeti algoritmussal egyenértékű kódot készíteni. Természetesen ez nem mindig szerencsés, figyelembe véve a későbbi javítások, hibakeresések tényét, de a programok hatékonyságának javításánál nagymértékben segíthet, ha a kódolásnál kihasználjuk a nyelv sajátosságait. Ha ilyen eltéréseket viszünk be a kódba, azt megfelelően dokumentálni kell.

A programszöveg strukturáltsága

A legtöbb esetben közömbös, hogy a program szövege egy szöveges fájl vagy több, de ha több fájl, akkor azok gyakran külön fordítási egységet is alkothatnak.

A Pascal, a C, PHP, Java, Javascript és Clipper esetén alkalmazhatók az úgynevezett include fájlok, amelyek olyan kódot tartalmaznak, amelyet fordítás közben a fordító beemel a szövegbe és együtt fordítja le őket. Más esetekben a fordító több különálló fordítási egységet hoz létre, amely a program fordításának különböző fázisában, de a szerkesztés előtti fázisban szerkesztődik össze.

A C, és a Clipper nyelvek igénylik úgynevezett header fájlok használatát, amelyben a rendszer előre lefordított programjai találhatók.

A Windows alatti programozás nagy előnye, hogy a Windows egységes belső programfelépítésének köszönhetően lehet olyan programot alkotni, amelynek bizonyos részei más és más programozási nyelven készülnek, és a kapcsolat közöttük csak a futtatáskor jön létre. Ez az újrafelhasználható programkomponensek használatát nagyban elősegíti.

Memóriaméret

Az egyes nyelvek a fejlődés során különböző stratégiákat alkalmaztak a memóriakezelésre és ezzel kapcsolatban az egyes program- és adatmodulok méretére vonatkozóan. Ezek a stratégiák mindig az optimális sebességű kód megalkotását célozták. Tudvalevő, hogy ha egy programkód mérete nagyobb, mint 64 kb, akkor a memóriacímzésre nem elég két bájtot használni. Ebben az esetben a kód lassabb lesz.

A Pascal, a BASIC és a Visual Basic azt választotta, hogy az egy függvénybe, eljárásba tartozó programkód mérete nem lépheti túl a 64 kilobájtot. Régebben a Pascal kód nem lehetett nagyobb ennél a méretnél.

Az adatszegmensnél is volt hasonló korlát. Az egy struktúrába tartozó adatok mérete nem lehet nagyobb 64 kilobájtnál.

A C nyelv memóriamodelleket állított fel. A különböző memóriamodellek a kódra és az adatok méretére adnak felső korlátokat egymástól függetlenül, illetve adnak korlát nélküli memóriakezelést.

Egy táblázatot adunk meg:

	Adatok mérete < 64 k	Adatok mérete > 64 k
Kódméret < 64 k	Small modell	Compact modell
Kódméret > 64 k	Medium modell	Large modell

Tiny esetén az adat + kód mérete < 64 K

Huge esetén minden modul statikus adatai külön 64 k-ban tárolódik, míg a **Large** modell esetén az összes statikus adat 64 K lehet csak

A Clipper a C nyelv Large modelljét örökölte (C-ben fejlesztették). A Clipper használja az ún. overlay technikát, amikor a memória egy részét a winchesteren tárolva mindig a megfelelő kód és adatrészletet tölti be a memóriába, ezzel DOS alatt nagyobb programokat tud futtatni, mint a rendelkezésre álló DOS memória.

Alprogramok hívása

Az egyes programozási nyelvek más módot használnak a programok eljárásainak meghívására.

Ez főleg a paraméterátadást tekintve különbözik. Ez alapvetően a Pascal és a C nyelv paraméterátadására vezethető vissza. A Pascal esetén mindig ugyanannyi és ugyanolyan típusú változónak kell lennie a hívás helyén és a hívott eljárás fejlécében, míg a C esetén nem baj, ha nem egyeznek meg a számok, a megfelelő változó értéke vagy elveszik, vagy ha nem kap értéket, akkor véletlenszerű értékkel töltődik fel. Ha egy átvett paraméter nem kap értéket, akkor csak arra kell vigyázni, hogy a nem meglévő értékét ne használjuk fel.

A fentiekkel szemben a Pascal paraméterátadása gyorsabb, ezért a Windows alapú programozási nyelvekben különösen, ha más nyelvű fejlesztéseket is akarunk használni, akkor külön deklarálni kell minden nyelven a Pascal rendszerű paraméterátadást. Mivel a C nyelv ANSI féle szabványosítása nem a Pascal paraméterátadási konvenciót használja, ezért a Windows alá írt C nyelvű programok többek között ezért sem vihetők át egyszerűen más rendszerek alá

10.4 A programok tesztelése, hibakeresés

A programok tesztelésének célja, hogy a program vajon a bemenetekre a specifikáció alapján megfelelő kimenetet szolgáltatja-e. A specifikációnak megfelelő programot **helyes programnak** hívják. A programok tesztelése és a hibakeresés során arra törekszünk, hogy az eredeti specifikációnak minél jobban megfelelő, illetve megfelelő programot állítsunk elő. Olyan tesztmódszereket kell használni és olyan hibakereső eszközöket, amelyek a hibák nagy részét kiszűrik. Néhány tapasztalati ténybe, azonban bele kell nyugodni:

- A program hibáinak száma és súlyossága exponenciálisan nő a mérettel
- A hibajavítás után az összes tesztelést célszerű lefolytatni
- A hibát megszüntető okokat kell megtalálni és kijavítani
- Gyakran egy hiba megszüntetése több másik hiba megjelenését vonja maga után
- A program készítője a legrosszabb tesztelő. A fejlesztőn kívül mással is teszteltetni kell a programot.

A fenti tényeken kívül még egy továbbiról is kell szót ejteni. Nagyobb méretű programok esetén 100%-osan hibátlan programról nem lehet beszélni. A világon a legszéleskörűbb tesztelésnek a Windows95-öt vetették alá. A program kiadása többek között ezért csúszott majdnem egy évet. Mégsem lett hiba nélküli. A tapasztalat azt mutatja, hogy ha a kód 95 – 99%-a hibátlannak bizonyul, akkor a programot késznek kell nyilvánítani. A maradék hibák kijavításának költsége ugyanis minden tekintetben olyan nagy, hogy nem éri meg az összes hibát kijavítani.

A fentiek alapján a tesztelés kritériumai

- A jó tesztelés nagy valószínűséggel felfedi a hibákat
- A jó tesztelési eljárásoknak megismételhetőknek kell lenni
- Érvényes és érvénytelen adatokra is kell tesztelni
- Minden tesztesetet maximálisan ki kell használni, azaz a legtöbb hibát fel kell deríteni

Fel kell tenni a kérdést, hogy **miért nem azt teszi** a program, amit kellene volna és **miért azt teszi**, amit nem kellett volna.

10.4.1 Statikus tesztelési módszerek

A statikus tesztelési módszerek a programkód vizsgálatán alapulnak. Ekkor nem futtatjuk a programot. Bár sokan azt gondolják, hogy a ami hardvereknél nincs nagy jelentősége ezeknek a módszereknek, hiszen a fordítók és interpreterek olyan jók, hogy mindenre választ adnak, minden hibát megtalálnak. Sajnos gyakori, hogy a fordítók nem találnak meg minden hibát, amint az alábbiakban látszik is. További probléma, hogy egyes rendszerek és fejlesztések esetén a fordítás folyamata olyan hosszú, hogy mindenképpen már a kódot is tesztelni kell. (Gyakori, hogy a fejlesztők olyan gépen dolgoznak – pénzhiány miatt, amelyek az adott fejlesztő rendszer követelményeit éppen csak kielégítik. Ilyenkor egy fordítás 5 –15 perc, de akár több óra hosszú is lehet.)

Kódellenőrzés

A legegyszerűbb lehetőség. Kinyomtatjuk, vagy a képernyőn átnézzük a kódot, miután begépeltük. Célszerű olyan editort használni, amely kiemeli az adott nyelv kulcsszavait, esetleg színnel vagy más módon elkülöníti az adatokat, az értékadó utasításokat. Ha lehet az program bevitelekor használni kell a strukturált írásmódot, ha akkor nem tettük meg, akkor utólag javítani kell a kódon. Figyelni kell, hogy az egyes programozási egységek kezdet és vége (begin … End, {…}, Procedure …. Return, stb…) megvan-e?

Szintaktikai ellenőrzés

A legtöbb fejlesztő eszköz ma már szintaktikailag ellenőrzi a program kódját és a megfelelő sorban ki is írja a hibaüzeneteket. Az interpreteres nyelvek gyakran már a programsor bevitelekor elvégzik az ellenőrzést, míg a compileres nyelvek csak a fordítás során.

Szemantikai ellenőrzés

Lehetséges, hogy egy szintaktikailag helyes program valójában nem azt teszi, ami a dolga lenne. Az interpreteres rendszerek szemantikailag nem ellenőrizhetik a bevitt kódot, hiszen általában nincsen a teljes rendszerre rálátásuk. Ekkor csak a programozó tudja végiggondolni, hogy programja valóban logikailag megfelelő, az alkalmazott algoritmusok valóban a kellő végeredményt adják, és a kódolás megfelel az algoritmusnak.

A compileres rendszerek esetén előfordul, hogy a fordító figyelmeztet bizonyos utasítások szemantikai problémáira. Gyakran találnak az ilyen rendszerek felesleges változókat, olyan kódrészleteket, amelyek sohasem futnak le, mindig biztosan azonos értéket felvevő változókat, stb. Az ilyen rendszerek használata sem teszi nélkülözhetővé a kézzel történő kódellenőrzést. Azok a compileres rendszerek, amelyek kódoptimalizálást végeznek, gyakran olyan assembly kódot hoznak létre az optimalizálás során, amely logikailag nem felelnek meg az algoritmusnak. Ekkor ki kell kapcsolni az optimalizálást.

Inicializálatlan változók

Meg kell keresni a kódban az inicializálatlan változókat, és kezdőértéket kell nekik adni. Sok váratlan hiba ilyen okokra vezethető vissza. Egyes rendszerek csak lefoglalják a változók számára a memóriaterületet, de nem adnak automatikusan nekik értéket. Ilyenkor az adott területen lévő véletlenszerű memóriaszemét lesz a kezdőérték.

Felesleges utasítások kiszűrése

Gyakran kódoláskor az algoritmusnak megfelelő kódot írunk, holott az adott nyelv ugyanazt a funkciót esetleg gyorsabban is meg tudja oldani. Az is előfordul, hogy például egy for ciklus ciklusváltozójának egy eljárásba való belépéskor külön értéket adunk. Ezekben az esetekben felesleges utasításokat helyezünk el, amelyek biztosan lassítják a programunkat.

Keresztreferencia táblázat

Ha a programunkban lévő változók értékeinek változását nem tudjuk követni, akkor célszerű keresztreferencia táblázatot készíteni. Erre a legtöbb fordító képes. Ez egy olyan táblázat, amely felsorolja, hogy az adott változó hol kap értéket, illetve hol történik hivatkozás rá a program során. Ennek alapján megállapíthatjuk, hogy mely változókat használjuk a leggyakrabban. Az interpreteres nyelvek egy részénél – BASIC – nem mindegy, hogy milyen sorrendben definiáljuk a változókat, a korábban definiált változókat a rendszer gyorsabban éri el.

Típuskeveredés

Egyes interpreteres nyelvek a bevitelkor nem ellenőrzik, hogy az értékadó utasítások két oldalán ugyanolyan típusú értékek szerepelnek-e. Más rendszerek, mint például a Clipper fordítási időben nem is tudja elvileg sem meghatározni egy változó típusát. Ezeknek a rendszereknek általában ez erősségük, hiszen futáskor dőlhet el, hogy az adott változó milyen értéket kap, „szabadabb” kódot lehet létrehozni, de egyúttal a tesztelésnél és hibakeresésnél hátrányuk, hiszen csak futás közben derülhetnek ki az esetleges problémák.

10.4.2 Dinamikus tesztelési módszerek

A programok hibáinak egy részét a statikus tesztelési módszerekkel ki lehet szűrni, de vannak olyan helyzetek, hogy csak a futás közbeni ellenőrzés segít. Hogy egy-egy teszt minél több tulajdonságot áruljon el a programról az alábbi módszereket lehet alkalmazni:

10.4.2.1 Fehér doboz módszerek

Az utasítások lefedésének elve

A program minden utasítását legalább egyszer végre kell hajtani. (Sajnos tapasztaltam már olyan esetet, amikor egy fejlesztő eszköz nem volt hibamentes és a leírások szerint jó program nem azt hajtotta végre, amit kellett.)

Döntések lefedésének elve

A programban lévő döntések minden következményét végig kell próbálni. A döntéseket igaz és hamis esetben is végig kell próbálni.

A feltételek lefedésének elve

A programban lévő feltételes elágazásokat minden feltételre ki kell próbálni, illetve az logikai összekötő műveleteket minden lehetséges helyzetre ki kell próbálni.

10.4.2.2 Fekete doboz módszerek

Ekvivalencia osztályok készítése

A lehetséges bemenő adatokat oly módon kell csoportosítani, hogy milyen kimenő adatot várunk tőlük. Ezeket ekvivalencia osztályoknak hívjuk. Nyilván ha egy ekvivalencia osztály egy elemére a helyes kimenetet kapjuk, akkor az osztály többi elemére is helyes eredményt kell kapnunk. Minden ekvivalencia osztályra tesztelni kell a programot.

Határeset analízis

Ha a lehetséges bemenő adatok ekvivalencia osztályait helyesen is állapítottuk meg, és úgy találjuk, hogy az osztályokra megfelelő választ ad a program, még mindig meg kell vizsgálni, hogy az ekvivalencia osztályok határeseteit hogyan kezeli le a program. Gyakran az ilyen helyzetben adott hibás eredmény helytelen algoritmusra, gondolatmenetre vagy túlzott egyszerűsítésre vezethető vissza

Stressz teszt

A programokat biztosan rossz bemenő adatokkal is tesztelni kell. A programok fejlesztése során a fejlesztő általában feltételezi, hogy a felhasználó csak helyes dolgokat művel, pedig ez nem így van. A felhasználó sokkal gyakrabban téved, hibázik, mint azt a legtöbb fejlesztő képzelné.

10.4.2.3 Speciális tesztek

Hatékonysági tesztek

A programok tesztelésének utolsó fázisa, annak megállapítása, hogy milyen hatékony a program, illetve mennyire jól felhasználható. A tesztelések során meg kell állapítani, hogy a program a meghatározott hardveren egyáltalán fut-e, megfelelő sebességgel fut-e. Ha nem fut, vagy a sebessége nem megfelelő, akkor meg kell keresni azokat az okokat, amelyek a megfelelő futást megakadályozzák, és annak megfelelően kell módosítani a programot, akár az algoritmusok szintjére is visszamenve.

Biztonsági teszt

A programoknak stabilaknak kellene lenniük, nem szabadna előre látható okok miatt lefagyniuk. Ezekre valók a biztonsági tesztek. A programot nagyon kevés és túl sok adattal is tesztelni kell, hogy nincsenek-e memóriakezelési problémái, szándékosan a legelfogadhatatlanabb adatokat kell bevinni, szabálytalanul leállítani, hogy kiderüljön, vajon a program képes-e helyreállítani az ilyen esetekben is a működését.

Például adatbázis-kezelő rendszerek esetén fontos, hogy szabálytalan leállás után az adatok ne sérüljenek meg, és az újraindítás után folytatható legyen a munka)

10.4.3 Hibakeresési módszerek

A következőkben néhány általános jellegű módszert adunk a hibák megkeresésére.

Indukciós módszer

Az indukciós módszer lényege az, hogy keresünk egy olyan bemeneti adathalmazt, amire a program jól működik, majd ennek az adathalmaznak megkeressük azt a határát, amelyre szerintünk szintén működik a program. A kiterjesztésen belül még próbálkozunk. Ha mindent rendben találunk, akkor tovább tágítjuk a bemenő adatok körét és továbbra is vizsgáljuk, hogy az adatokra a program helyesen válaszol-e.

Ha bármilyen ponton úgy találjuk, hogy a program a bemeneti adatokra hibás választ ad, akkor megpróbáljuk leszűkíteni a bemeneti adatoknak azt a halmazát, amelyre hibás választ ad a program.

Egy példán keresztül szeretném illusztrálni az indukciós módszert. Ha egy program bizonyos bemeneti paraméterek hatására jól működik, pl 100, 101, 102 adatokra jól működik, akkor feltételezzük, hogy legalább ezerig jól fog működni. Megpróbáljuk ebben a nagyságrendben. Tegyük fel, hogy nincs probléma.

Ekkor feltesszük, hogy a program az 1000-es nagyságrendbeli adatokkal is jól működik. Ha például 1000, 1001, 1002-re nem jól működik a program, akkor keresünk még ugyanabban a nagyságrendben, új adatokat, például 1099-et, amelyre feltevésünk szerint jól kellene működnie. Ha arra az adatra jól működik, akkor a hiba az ezer környéki tesztadatokra jellemző, ha hibázik, akkor nyilvánvalóan a nagyobb adatokra is hibásan reagál.

Dedukciós módszer

Ha a programunk bizonyos bemeneti adatokra hibásan, más adatokra jól reagál, akkor megvizsgáljuk, hogy a bemeneti adatokban mi az a közös tulajdonság, ami szerint a végeredmény osztályozódik. Ha találtunk ilyen tulajdonságot, akkor jósolunk, és a jóslat alapján új tesztadatokat állítunk elő, majd teszteljük a programot. Ha a tesztadatokra a várt eredmények születtek, akkor feltételezésünk jó volt, a hiba valóban a bemenő adatok közös tulajdonságai alapján kereshető meg. Ha nem a feltételezéseknek megfelelő eredmények születnek, akkor nem sikerült megtalálni az eredeti bemeneti adatokban a csoportképző faktort, tehát új tulajdonságot kell keresni rajtuk. Például:

Egy program a 3, 5, 7, 11 adatokra jó, a 10, 12 adatokra hibás eredményt ad. Feltételezzük, hogy a jó bemeneti adatokban a közös az, hogy a számok prímszámok vagy páratlan számok. Kipróbálunk olyan értékeket, amelyek prímek, például a 17 és kipróbáljuk a 2-t is. Az egyik feltételezésünket ki fogja ejteni a két próba.

Visszalépéses módszer

Ennek a módszernek az a lényege, hogy a program végeredményét vizsgálom, hasonlítom össze a specifikációban megjelöltekkel és a programban visszafelé lépkedve keresem meg a hibás lépést.

Tesztesetek felhasználása

A program specifikációja alapján teljes körű, körültekintően összeállított teszthalmazt készítünk és a tesztesetek felhasználásával, valamint a kapott végeredmény felhasználásával következtetünk a hiba okára.

10.4.4 Hibakeresési eszközök

A modern fejlesztő rendszerek majd mindegyike rendelkezik már valamilyen hibakereső szolgáltatással. A régebbi rendszerek a program futása közben beálló hibára gyakran egy memóriacím, esetleg a stack, kiírásával reagáltak és általában az operációs rendszer nem túl széleskörű szolgáltatásait használták erre a célra. Később megjelentek az olyan fejlesztő rendszerek, amelyek a hiba megjelenésekor kiírták, hogy a forrásszöveg melyik sorában, milyen jellegű hiba történt, akár egy hibakóddal, akár szövegesen is.

A mai integrált fejlesztőeszközök némelyike képes arra, hogy a hibánál kijelezze a hibás utasítást, adjon tippet a hiba okára, írja ki a verem állapotát, a változók pillanatnyi értékét. A továbbiakban átnézzük, hogy melyek azok az eszközök, amelyeket a fejlesztő felhasználhat hibakeresésre.

Nyomkövetés

A program végrehajtása során a nyomkövető eszköz kiírja, hogy melyik utasításnál, melyik sorban fut éppen a program. A Borland Pascal vagy Borland C/C++ esetén a program a képernyő egyik ablakában megjeleníti a program kimenetét, a másik ablakában pedig megjeleníti a forrásszöveget és jelzi, hogy éppen hol tart a program. A legtöbb komolyabb hibakereső rendszer képes arra, hogy az így futó programot bármelyik pillanatban leállítsa.

Debugging

A **debugger** eredetileg a futtatható programok gépi kódú visszafejtésére alkalmas eszközt és a **debugging** magát a visszafejtés folyamatát jelentette. Manapság már nem mindig az assembly nyelvű kód vizsgálata a célszerű. Vannak olyan eszközök, amelyek alkalmasak arra, hogy ha egy program fejlesztési nyelvét ismerjük, akkor képes legyen visszaállítani az eredeti forráskódot több-kevesebb sikerrel. Ilyen eszköz létezik assembly, Pascal, C, Clipper nyelvekre – tudomásom szerint.

Töréspontok elhelyezése

Ha egy nagyobb program működését vizsgáljuk, a nyomkövetés alkalmazása esetleg annyira lelassíthatja a program működését, hogy elviselhető időn belül nem jutunk el a vizsgálat alá vont helyre. Ilyenkor segít az, hogy töréspontokat helyezünk el a programban. Ezeken a pontokon a program megáll és megvizsgálható az állapota. A hibakereső rendszer tulajdonságaitól függően azután a töréspontoktól kezdve a program folytatható vagy félbeszakad. A nagyobb tudású rendszerek a képernyőtartalmat elmentik a töréspont előtt, majd visszaállítják a töréspont után.

Részeredmények, állapot kiíratása

Gyakori hibakeresési lehetőség. A program bizonyos vizsgált részeire kiíró utasításokat teszünk, amelyek tájékoztatnak a program, vagy csak bizonyos változók pillanatnyi állapotáról. Az állapot kiíratása során célszerű megvizsgálni az adott programozási egységben szereplő változók értékét, az adott eljárásnak átadott paraméterek és a visszaadott paraméterek értékét. A kiíratás összekapcsolható töréspontok elhelyezésével is.

Lépésenkénti végrehajtás

A töréspontok után a programot egy ideig lépésenként is végrehajtathatjuk. Ilyenkor a programkódban lépünk egyet majd figyeljük a program új állapotát. Figyelünk a képernyőtartalomra, a változók és paraméterek értékére, a fájlok állapotára. A lépésenkénti végrehajtás során gyakran eljárást vagy függvényt hívunk meg a forráskódban. Ilyenkor általában két választásunk van. Vagy a meghívott eljárást, függvényt is lépésenként hajtatjuk végre, vagy a kérdéses részt a program teljes sebességgel végzi el. Nyilván, ha biztosak vagyunk az adott függvény, vagy eljárás hibátlanságában, akkor az utóbbit választjuk. A lépésenkénti végrehajtást a program teljes sebességű további futtatásával is lehet általában kombinálni.

Tervezés fázisába való visszalépés

Ha végképpen nem találjuk meg a hibákat az eddigi „nagyágyúkkal”, akkor bizony vissza kell lépni a tervezés fázisába és vagy algoritmus szinten, vagy kódolási szinten újra kell gondolni a hibás részt, hátha valami elemi szemantikai hiba okozza a hibás működést.

Ciklusok befejeződésének vizsgálata

Sok hiba felderítésének módja, a ciklusok befejeződésének vizsgálata. A programkódok legalább 30% ciklusokból áll, és a ciklusokba helytelen kilépési feltételei, vagy a megszámlálásos ciklusoknál a ciklusváltozó kezdő és végértékének helytelen meghatározása az oka az olyan fajta hibáknak, amelyek nem minden esetben fordulnak elő. Ezt a vizsgálatot a határeset analízissel is célszerű összekötni.

Változók értékének menet közbeni kiíratása

Állapotdefiníció

Egy gyakran használható módszer. A programozó a programkód vagy algoritmus alapján meghatározza, hogy bizonyos kezdőértékek alapján a programnak egy adott helyen milyen állapotba kell kerülnie. Ha eltérést tapasztal a megkívánt és a tapasztalt állapot között, akkor a hiba okát nyilván a kettő között kell keresni és nyilván a különbséget kell tüzetesen megvizsgálni.

10.4.5 A tesztelők személye, szervezett tesztek

Már korábban is említettük, de most részletesebben szólunk a tesztelők személyéről. Köztudott, hogy egy program fejlesztője egy idő múlva a triviális hibákat sem veszi észre. Ennek könnyen belátható okai vannak. Egy probléma megoldásának során a fejlesztő gyakran elgondol bizonyos algoritmusokat, amelyeknek nem minden részletét tisztázza le, majd ha a megvalósítás során az algoritmus hibái nem lesznek nyilvánvalóak, akkor gondolatban az algoritmus kérdéses részét jónak fogadja el. Megkönnyebbül attól a gondolattól, hogy az algoritmusnak az a része jó. Ettől az állapottól csak keservesen tud megszabadulni és csak úgy, ha újra elemzi, és újra megalkotja a kérdéses részt. Sajnos az embernek a hiba felismeréséig, megtalálásáig nehéz eljutnia.

A programozás során szinte elkerülhetetlen, hogy a programozó kisebb – nagyobb mértékben ne térjen el az eredeti elgondolásoktól. Ekkor az algoritmus és a kód már nem fog teljesen megfelelni egymásnak. Ez is okozhat olyan hibákat, amelyeket később a fejlesztő nem vesz észre.

A fejlesztés során gyakori, hogy a részletkérdésekbe annyira beleássa magát a fejlesztő, hogy nagyobb összefüggéseket nem fedez fel.

Milyen módon lehet kikerülni ezeket a csapdákat, hogyan lehet a tesztelést, a hibakeresést hatékonyabbá tenni.

A tesztek és a hibakeresés során nem elég a képernyőn vizsgálni a forrásszöveget, ki is kell nyomtatni. A hagyományos papír alapú vizsgálódás sok olyan összefüggést feltár, ami a képernyőn nem válik nyilvánvalóvá.

Mindig kell olyan tesztelő személyeket találni, akik az adott fejlesztési fázisban nem vettek részt, az adott kódrészletet nem ismerik. A fejlesztőnek hagyni kell a másik személyt, hogy amennyire lehet, egyedül értse meg az adott részletet, és egyedül keresse meg a kérdéses hibát.

A tesztelő személyek között kell lennie olyannak, aki számítástechnikai, fejlesztői szempontból vizsgálja a programot és kell lennie olyannak is, aki a felhasználó szakmai szempontjából nézi a programot. A fejlesztő és a majdani felhasználó nem egyforma módon közelíti meg az adott problémát, ennek megfelelően nem ugyanazok a dolgok fontosak egyik számára, mint a másik számára.

A legjobb tesztelés az éles adatokon, éles helyzetben történő tesztelés. Egy program elkészülte után az igazi felhasználók visszajelzései lesznek igazán a felhasználói teszt szempontjából lényegesek.

10.5 Hatékonyságvizsgálat, optimalizálás

A programok hatékonyságát több mérőszám tudja csak leírni. A program hatékonyságának legfontosabb kritériumai,

- A program futásának sebessége,
- A program mérete
- A program bonyolultsága.

Manapság a szoftverek széleskörű elterjedtségének köszönhetően a programok hatékonyságához még hozzájön a programok felhasználhatóságának, barátságosságának szempontja is.

E fenti kritériumok közül sajnos az alábbi összefüggések állnak fent:

- A programok futásának sebessége és a program bonyolultsága általában egymásnak ellentmondó fogalmak.
- A program mérete arányos a program bonyolultságával.
- A program mérete fordítottan arányos a program futásának sebességével.

10.5.1 Rendszerek hatékonyságának megállapítása

A programok hatékonyságának megállapításánál a legjobban mérhető fogalom a futási idő megállapítása. A futási idő megállapításánál mindig több mérést kell végezni, és figyelni kell az egyes futási időket. Csak így lehet megbízhatóan kiszűrni a számítógépen futó alkalmazások, az operációs rendszer és a hardver egyéb paraméterének változásait. Azt is célszerű megállapítani, hogy különböző konfigurációkon hogyan működik a program, melyek azok a paraméterek, amelyek a program maximális hatékonyságát, sebességét biztosítják.

A mérések során fel kell jegyezni az **átlagos**, a **maximális** és a **minimális** futási időt. Természetesen az átlagos idő lesz a legjellemzőbb a program sebességére. Korábban jeleztük az algoritmusok hatékonyságának jellemzőit (O(1), O(N), stb...) Általában a szoftverek hatékonysága is hasonlóképpen jellemezhető.

10.5.1.1 Egzakt módszerek

A program sebességét úgy tudjuk megállapítani, hogy a mérendő rész elején és végén megmérjük a gép belső órájának időpontját, majd a két időt kivonjuk egymásból.

Vannak olyan szoftverek, amelyek segítségével a futási időket egzakt módon mérni lehet. Ezek a **profiler**nek nevezett programok. Segítségükkel a program optimális futása szempontjából sok különböző mérést lehet elvégezni. A Borland Pascal része egy ilyen profiler program.

Speciális operációs rendszer szolgáltatások hívásából mégis megállapítható a program futás közbeni mérete.

Egy program barátságossága nem mérhető egzakt módon.

10.5.1.2 Kézi módszerek

A programok sebességét sokszor a forrásszöveg hiányában nem tudjuk mérni egzaktul. Ekkor stopperrel egy batch állományból indítva lehet lefuttatni a programot és mérni a sebességet. Ha a program futása túl gyors, akkor a batch fájlba egy olyan ciklust kell beiktatni, amely kellően sokszor fut le ahhoz, hogy a program sebessége mérhető legyen. Ha megmértük a program sebességét, akkor meg kell mérni a batch file futásának sebességét is, így kaphatjuk meg a program futásának sebességét.

A program memóriában elfoglalt méretét gyakran nem tudjuk megállapítani, de a háttértáron elfoglalt méretből, az adatállományok méretéből, esetleg következtethetünk rá.

A program bonyolultsága nem egzakt fogalom, ezért mérése nem is lehet egzakt. Viszonyítani kell a feldolgozandó adatok bonyolultságát a programkód összetettségéhez. Ha a kettő között kiugró aránytalanságot találunk, akkor a kód túl bonyolult.

Egy program barátságosságának mérése csak több, különböző képzettségű tesztelő tesztelése után állapítható meg. Célszerű a programot egymástól független személyekkel teszteltetni, és előre leírni azokat a szempontokat, amelyeket pontozniuk kell egy előre megadott skála szerint. Előre el kell döntenünk, hogy az egyes kategóriákat a végső értékelésnél milyen súllyal vesszük figyelembe. A több személy az egyéni ízlésbeli különbségeket, míg a különböző képzettség a képzettségből fakadó hibákon való átsiklást küszöböli ki. Ha van hasonló feladatokat ellátó másik program, akkor célszerű a két program összehasonlító tesztelése is.

10.5.2 Globális optimalizálás

A tesztelések során kiderülnek a program gyenge pontjai, elsősorban a méret és a sebességbeli hiányosságok. A program sebességére és méretére több módszer lehetséges.

A programban legfontosabb, hogy a program egésze működjön optimálisan, ezért elsősorban globálisan kell megtalálni azokat a tényezőket, amikkel javítani lehet a program sebességét.

Meg kell érteni a program során alkalmazott algoritmus és annak megfelelően, esetleg gyorsabbra cserélni azt.

Csökkentsük a ciklusok végrehajtási számát (Például, ha N egész szám közül keressük a prímeket, akkor N/2 helyett elegendő csak négyzetgyök N-ig keresni a prímeket)

A ciklusok végrehajtási idejét csökkentsük

Fel kell használni a feladat speciális tulajdonságait, (pl. ha a keresés rendezett halmazon történik, akkor a gyorskeresés $\log_2 N$ lépés alatt zajlik le átlagosan, míg lineáris keresésnél csak N/2 lépés alatt.

Matematikai elvek, ismeretek, definíciók használata.

Kisebb méretű, gyorsabban feldolgozható adattípusok használata. Ne használjunk különböző típusokat egy kifejezésben

Függvények ismételt kiértékelése helyett – ha nem változik a függvény értéke menet közben – temporary változóban tároljuk a függvény értékét.

A feltételeinket egyszerűsítsük

Az adattípusokat gondosan válogassuk meg, a helykihasználás és a feldolgozás sebességének figyelembevételével.

Bekapcsoljuk a fordítókba beépített kódoptimalizáló opciókat. Ennek az a veszélye, hogy bizonyos esetekben a rendszer nem azt a gépi kódú programot hozza létre, amit mi elterveztünk.

10.5.3 Lokális optimalizálás

Ha már kifogytunk a globális optimalizálás ötleteiből, akkor jó szolgálatot tehet, ha megvizsgáljuk, melyek azok a programok, amelyekben a legtöbbet tartózkodik a program. Ezeket az eljárásokat kell gyorsabbá tenni, ekkor az egész program futása felgyorsul. Milyen módon?

Átvizsgáljuk az algoritmust és megnézzük, hogy nincsen-e gyorsabb, helyettesítő algoritmus.

A speciális esetek kiszűrése. Az eljárásban konkrét értékadással vagy egyéb módon kizárjuk a speciális eseteket, a szélső értékeket külön kezeljük le, nem próbálunk általános megoldást adni ezekre az esetekre.

Hatékonyabb programkódolási technikát alkalmazunk. A tapasztalat azt mutatja, hogy a hatékonyabb programkódok egyúttal nehezebben követhetők is. Elsősorban C nyelven van annak jelentősége, hogy milyen kódot ír le a programozó, mivel a leírt kódtól nagymértékben függ a gépi kódú végeredmény.

Egyes fordítók a fordítás során keletkezett kódot optimalizálják, a programozó által leírt kódhoz képest gyorsabb assembly kódot hoznak létre. Sajnos egyes esetekben a kódoptimalizálás váratlan eredményeket is hozhat, ezért kellő körültekintéssel kell alkalmazni.

Rendszerközeli betéteket alkalmazunk (Assembly – csínján kell bánni vele!)

Trükköket alkalmazunk, amely kihasználja a processzor, az operációs rendszer vagy a nyelv egyes speciális tulajdonságait. (Vigyázni kell vele, könnyen érthetetlenné és visszafejthetetlenné tehetjük a programunkat)

Kisebb méretű, gyorsabban feldolgozható adattípusok használata. Megjegyzendő, hogy a PC-ken az Integer adatokat dolgozza fel leggyorsabban az operációs rendszer.

Ha már leteszteltük a programot, akkor kikapcsoljuk a fordítóprogram hibaellenőrző kapcsolóját, ami gyorsabb és kisebb programkódot hoz létre.

10.5.4 Hatékonyság transzformációk

Legyenek az alábbi definiciók:

F – Feltétel

U1, U2, U3, ... utasítások

Ha F akkor U1;U2 különben U1;U3	U1 Ha F akkor U2 különben U3 Ha az F-et nem módosítja az U1. Ha F akkor U2 különben U3 U1 Ha az utasítások sorrendje nem számít.
Ha F akkor U1;U3 különben U2;U3	Ha F akkor U1 különben U2 U3 Mindig megcsinálhatjuk.
Ha F1 és F2 akkor U	Ha F1 akkor Ha F2 akkor U Akkor, ha F1 rövid, F2 bonyolult. (A Pascalban beállítható, hogy ne értékelje ki F2-t, ha F1 hamis.)
Ha F1 akkor U1 különben Ha F2 akkor U2	Ha F1 akkor U1 különben U2 Ha F1= NOT F2.
Ha F akkor U1 Ha F akkor U2	Ha F akkor U1;U2 Ha U1-nek nincs hatása F-re.
Ciklus amíg CF U1;U3 Cvége Ciklus amíg CF U2;U3 Cvége	Ciklus amíg CF U1;U2;U3 Cvége Ha U1 és U2 függetlenek egymástól, és nincsenek hatással CF-re, legfeljebb U3-ra.

U Ciklus amíg Cf U Cvége	Ciklus U amíg CF Mindig megcsinálhatjuk, de van, hogy nem célszerű.
Ciklus amíg CF Ha F akkor U Cvége	Ha F akkor Ciklus amíg CF U Cvége Ha F nem változik a ciklus futása alatt.
Ciklus amíg CF U1;U2 Cvége	U1 Ciklus amíg CF U2 Cvége Ha az U1 végrehajtása független a ciklustól és saját maga korábbi végrehajtásától.

10.6 Dokumentáció

10.6.1 A dokumentáció formája

Fel lehet tenni a kérdést, hogy a dokumentáció milyen formában jelenjen meg, papíron vagy elektronikus úton. Mind a két megoldásnak vannak előnyei és hátrányai.

A papíron lévő dokumentáció áttekinthetőbb, jobban lehet benne böngészni, ismerősebb a használata egy kezdő felhasználónak vagy a rendszerrel most ismerkedőnek, de bizonyos speciális információkat nehezebb megtalálni benne, még akkor is, ha tartalomjegyzék, és tárgymutató van benne. Nem utolsósorban egy több száz oldalas könyv papír- és nyomtatási költsége tetemes lehet.

Az elektronikus dokumentációban könnyebb szavakra, kifejezésekre keresni, az elfoglalt tárterület minimális, tartalomjegyzék tárgymutató alapján lehet keresni benne, és papírköltség sincsen, ugyanakkor a problémakörrel frissen ismerkedők nehezebben tudják megtenni az első lépéseket, mivel nincs olyan áttekintésük a dolgokról.

A célszerűség azt diktálja, hogy általában legyen elektronikus és papír alapú dokumentáció is egy rendszerhez, figyelve arra, hogy a rendszer felhasználásának különböző fázisaiban más és más a célszerű formátum. Ha a dokumentáció terjedelme nem túl nagy, akkor mindenképpen mind a két formában célszerű közreadni, ha a méret indokolja, akkor pedig meg kell teremteni annak a lehetőségét, hogy az elektronikus dokumentációt megfelelő formában ki lehessen nyomtatni.

A „nagy” szoftveres cégek, mint a Microsoft, Novell, Lotus stb… a felhasználói és a fejlesztői dokumentációkat manapság csak elektronikus úton mellékelik a szoftverez, de olyan formában, hogy azt bárki kinyomtathatja. Ezekez néha külön is meg lehet vásárolni, súlyos tízezrekért.

10.6.1.1 Az elektronikus dokumentáció szokásos eszközei

Szövegszerkesztő

Bármilyen dokumentáció is készül, egy szövegszerkesztő alapvetőt alapvetően használni kell. Néha célszerű egyszerű ASCII editort használni, de figyelni kell a kódlapokra. DOS-os 437-es kódlapot használó editor a Windowsban nehezen olvasható fájlokat eredményez, míg 852-es kódlapot használva a 437-es kódlapban

nem lehet olvasni az állományokat és ekkor még nyomtatásról nem is beszéltünk. Célszerűen használható windows platformon egy Microsoft Office vagy egy Open Office is.

Szoftverek formális leírására használható a Microsoft Visio, amely UML eszközökkel, és egyéb alkalmazásleíró vizuális eszközökkel rendelkezik.

Adobe Acrobat formátum

Az Adobe cég régóta piacon van a PDF (=Portable Document Format) formátummal. A fájlok szerkesztéséhez az Acrobat Composer-t vagy más PDF nyomtatót (Primo PDF printer driver) lehet használni, vagy más eszközöket. A PDF állományok előállításának elterjedt módszere, hogy a rendszerbe egy PDF nyomtatót telepítünk, ami a kimenetet PDF állományként generálja. Az Acrobat Reader (ACROREAD.EXE) program olvassa, jeleníti meg az ilyen fájlokat. A fájlokat kinyomtatva, a papíron ugyanúgy jelenik meg a tartalom, mint a képernyőn. Az Acrobat Reader freeware-ként is hozzáférhető. Az olvasáshoz lehet használni a kevesebbet tudó és free Foxit Reader-t is.

HTML formátum

Az Internet elterjedésével egyre gyakrabban adják a programok dokumentációjának egy részét HTML formátumban. Az ilyen fájlok szerkesztéséhez végső soron egy tetszőleges editor, no meg egy jó HTML kézikönyv is elég. Ha komolyabb oldalakat akarunk tervezni, akkor célszerű lehet valamilyen erre a témára kihegyezett programot használni, például a Netscape Composer, Microsoft Frontpage, Adobe PageMill, Macromedia Dreamweaver, stb. programokat vagy más az internetről letölthető free programokat is. A HTML oldalak megjelenítéséhez csak egy böngésző program kell, ami létezik akár DOS-os környezetben is.

10.6.1.2 A papír alapú dokumentáció

Nyilván a papír alapú dokumentáció elkészítése előtt létrejön egy elektronikus formátum is. Ehhez mindenképpen a korábbiakban felvázolt eszközök kellenek. A kinyomtatás során azonban vigyázni kell, hogy a nyomtatók alapértelmezésének beállított A/4-es (amerikai nyomtatóknál a Letter) papírmérettel kinyomtatott dokumentáció nem használható jól, túlságosan nagy. Elterjedt formátum a B/5-ös, amely az A/5-nél és az A/4 között van. Nem tartja meg az aranymetszés szabályait, annál kicsit szélesebb.

A dokumentációban lévő ábrákat, fényképeket megfelelő minőségben csakis lézernyomtatók képesek kinyomtatni, de csak fekete-fehérben. A színes festékszórós elven működő nyomtatók csak speciális papír felhasználásával produkálnak megfelelő minőséget.

10.6.2 Felhasználói dokumentáció

A felhasználó a program használata során több különböző fázison megy keresztül. A program telepítése után csak ismerkedik a számára kialakított rendszerrel, ekkor sok alapvető ismeretre van szüksége. A használat során egyre több aspektusát ismeri a rendszernek. Ekkor már inkább rövid, emlékeztető segítségre van csak szüksége.

A felhasználói dokumentáció készítésének célja, hogy a programrendszer felhasználója el tudjon igazodni a telepítés, a használatbavétel, az üzemszerű használat és az esetlegesen felmerülő hibák során. Ennek megfelelően a felhasználói dokumentációnak is több részből kell állnia. Az egyes részeknek elektronikus vagy papír alapon célszerű létezni, esetleg mind a két formában.

A felhasználói dokumentációnak a következő részeket kell tartalmaznia:

- Általános leírás a rendszerről, amiben a rendszer célja, a képességei le vannak írva.
- A rendszer hardverfeltételei: (minimális, ajánlott) processzor, memória, megjelenítő fajtája, szükséges hely a háttértáron, nyomtató kell-e, egér kell-e, egyéb speciális hardver kell-e.
- A rendszer szoftverfeltételei: operációs rendszer fajtája, verziószáma, esetlegesen szükséges kiegészítő, együttműködő programok, mint pl. megjelenítők, szövegszerkesztők, stb…
- Hálózati alkalmazás esetén, a hálózati operációs rendszer fajtáját, egyéb ismérveit.
- A rendszer telepítésének módja, lehetőleg lépésről-lépésre leírva.
- A rendszer indítása
- A felhasználói Interface általános leírása (menürendszerének, párbeszédablakok)
- Az üzemszerű működéshez szükséges részek leírása – pontonként.
- A karbantartási feladatok elvégzéséhez szükséges részek leírása – pontonként.

- A képernyőn megjelenő listák, beviteli helyek, nyomtatási listák leírása.
- Előforduló hibaüzenetek magyarázata, és azok javításának módja.
- GYFK – Gyakran Feltett Kérdések. A programok működése során a felhasználók általában ugyanazokba a problémákba botlanak bele, és ugyanazokat a kérdéseket teszik fel. A kérdéseket és a rájuk adott válaszokat is célszerű befoglalni a dokumentációba
- A felhasználói segítségkérés és a válasz módja.
- További fejlesztési tervek, irányok.

10.6.3 Fejlesztői dokumentáció

A programok készítése során a fejlesztő előbb-utóbb szembetalálja magát azzal a helyzettel, hogy a régebben írt programok működésére, programrészletekre már nem emlékszik tisztán. Gyakori az is, különösen hosszabb fejlesztés alatt, hogy a tervezés során már jól megtervezett részleteken nem igazodik ki. A későbbi továbbfejlesztéseket, hibajavításokat sem feltétlenül ugyanazok a személyek végzik, akik annak idején a rendszert fejlesztették. Mindezek az okok igazolják a program készítése során a fejlesztői dokumentáció létrehozását.

A fejlesztői dokumentáció célja, hogy a rendszer fejlesztése, a későbbi hibakeresés, illetve a továbbfejlesztések során a rendszerről részletes, bárki hozzáértő által felhasználható dokumentáció legyen. A dokumentációnak minden olyan szükséges információt tartalmaznia kell, ami alapján egy teljesen idegen fejlesztő is sikeresen elvégezhesse a szükséges beavatkozásokat.

A fejlesztői dokumentációt nem szokás átadni a megrendelőnek. Ennek két oka van. Általában a megrendelő nem is tudja használni a fejlesztői dokumentációt. A másik ok pedig az, hogy ha átadjuk a fejlesztői dokumentációt, akkor evvel szabad utat adunk másoknak is a rendszer továbbfejlesztésére.

A fejlesztői dokumentációnál nem elsőrendű fontosságú, hogy papíron is meglegyen a dokumentáció, de mindenképpen olyan formában legyen, hogy évekkel később is felhasználható legyen.

- A fejlesztői dokumentációnak tartalmaznia kell a következőket:
- A rendszer tervezése során létrejött minden dokumentációt, azaz
- A rendszer célját, (specifikáció) a tervezésének alapelveit,
- Képernyőterveket,
- Nyomtatási listákat,
- Az adatszerkezeteket,
- Algoritmusokat (Az algoritmusokat magyarázatokkal kell ellátni)
- A fejlesztés hardverfeltételeit: (minimális, ajánlott) processzor, memória, megjelenítő fajtája, szükséges hely a háttértáron, nyomtató kell-e, egér kell-e, egyéb speciális hardver kell-e.
- A fejlesztés szoftverfeltételeit: operációs rendszer fajtája, verziószáma, esetlegesen szükséges kiegészítő, együttműködő programok, mint pl. képszerkesztők, megjelenítők, szövegszerkesztők, egyéb editorok stb…
- Hálózati alkalmazás esetén, a hálózati operációs rendszer fajtáját, egyéb ismérveit.
- A felhasznált fejlesztő eszközök leírását:
- A fejlesztő eszköz elnevezése, verziószáma,
- Esetleges kiegészítő library-k neve, verziószáma,
- Esetleg maga a kiegészítő rendszer elektronikusan.
- Az elkészült szoftver teljes forrásszövegét, megfelelő megjegyzésekkel ellátva
- Az esetleges javítások, fejlesztések verzióit
- A javítások és fejlesztések követésének állomásait, mit javítottak, hogyan javították, mit fejlesztettek
- A továbbfejlesztéssel kapcsolatos terveket, lehetőségeket.

Mivel egy nagyobb rendszer esetén a fejlesztések során együtt változik a felhasználói dokumentáció is a rendszerrel, ezért a felhasználói dokumentációt is tárolni kell a fejlesztői dokumentáció mellett és adott esetben visszamenőleg meg kell őrizni azt, több verzióban is.

10.7 Betanítás, oktatás

Egy nagyobb rendszer bevezetése esetén a felhasználóknak gyakran gondot okoz az áttérés a korábban használt rendszerekről az újonnan kifejlesztett rendszerre. A felhasználók - teljesen jogosan – a munkájukat akarják csak végezni. Őket alapvetően nem érdekli, hogy milyen módon végzik a munkájukat, csak hatéko-

nyan végezhessék. Egy átállás a hatékony munkavégzésben mindenképpen gátolja őket, ugyanis az átállás során egy ideig együtt kell végezniük a munkát a régi módon és az újonnan bevezetett rendszerben is, hiszen nem biztosítja semmi azt, hogy az új rendszer hibamentes és tökéletes.

Gyakran a munka számítógépesítését csak a főnök akarja, mivel így nagyobb intenzitású munkára tudja az alkalmazottakat rávenni. Gyakran az alkalmazottak nem is értik, hogy mire jó nekik a számítógépes rendszer.

Még manapság is benne van a munkát végzőkben a félelem a számítógépes rendszerektől, hiszen a mai 40-es korosztály még nem használt fiatalabb korában gépet. Bár a helyzet változik, mivel a mai fiatalok az oktatásban kapcsolatba kerülnek gépekkel, de azt is figyelembe kell venni, hogy az emberek képességei nem egyformák, és adott esetben olyannak kell használni egy szoftvert, aki esetleg a betűvetéssel is hadilábon áll.

A felhasználókban sok előítélet él a számítógépes rendszerekkel kapcsolatban. Jobban megjegyzik azokat az eseteket, amikor egy számítógépes rendszer nem a megfelelő módon működött és bosszúságot okozott nekik. (Mellesleg sokszor a „számítógéppel segített ügyintézés" tovább tart, mint a kézzel végzett)

A fenti problémák megoldását a megfelelő teljesítményű hardver és a megfelelően elkészített szoftveren kívül csakis a betanítás és oktatás jelentheti. A korábbiakban beszéltünk arról, hogyan lehet megfelelő szoftvert készíteni, a hardver pedig általában pénzkérdés

Az oktatás során két féle oktatást lehet elképzelni. Általános jellegűt, ami során a gép kezelését sajátítják el a dolgozók, illetve a konkrét rendszerhez tartozó oktatást. A dolgozóknak rendelkezniük kell általános felhasználói ismereteikkel is egy szoftverrendszer használatakor, hiszen vannak olyan helyzetek, amikor nem csak az adott szoftverhez tartozó problémákat kell megoldani. Gyakran egy alkalmazott számítógépén több különböző fejlesztésből származó, különböző célú programok foglalnak helyet, az alkalmazott pedig felváltva vagy akár párhuzamosan is használja őket.

10.7.1 Általános informatikai jellegű oktatás

Az általános jellegű informatikai oktatást a cégek nem szeretik, hiszen álláspontjuk szerint ne ők fizessék meg az alkalmazottak oktatását, hanem azok már okosan jöjjenek hozzájuk dolgozni. Sajnos egy munkahelyen az alkalmazottak tudásszintje korántsem egységes a számítógépek használata terén. Azok az alkalmazottak, akik valamilyen képzésben korábban részt vettek, gyakran nem használván a tanultakat, el is felejtik azokat.

Egy programrendszer készítőjének meg kell becsülnie a majdani felhasználók oktatásának optimális mértékét. Egy ilyen oktatásnak az a célja, hogy a fejlesztett rendszer használói a jövőben nagyjából egy kötelezően minimális szinten álljanak. Ennek a szintnek elegendőnek kell lennie az általános számítógép-kezelési eljárások elvégzéséhez, az operációs rendszer működtetéséhez, fájlkezelési műveletek végzéséhez, jogosultságok, belépések és kilépések, nyomtatási feladatok elvégzéséhez. Ha multitaszk operációs rendszert használnak, akkor az ebből adódó sajátosságokat is ismerni kell a felhasználóknak. Célszerű oktatni őket a dokumentumok és szoftverek elválasztásának szükségességére. Célszerű olyan gyakorlati tudnivalókat átadni, amelyeket később mankóknak használhatnak. Természetesen nem cél, hogy professzionális felhasználók legyenek az általunk betanítottak.

Tudatosítani kell a majdani diákokban, hogy ha a tanfolyam után hamarosan nem hasznosítják a friss ismereteiket, akkor azok holt ismeretek lesznek, majd hamarosan el is felejtődnek.

Néhány szempont az oktatáshoz:

A felhasználók többnyire munka mellett vagy munka után részesülnek az oktatásban, ennek megfelelően nem mindig állnak szellemi képességeik legmagasabb fokán.

A tanítandó anyagot úgy kell megtervezni, hogy minden szükséges gyakorlati tudnivaló benne legyen, amit a felhasználónak a későbbiekben használnia kell. Inkább tervezzünk kicsivel több anyagot a felhasználóknak, mint a feltétlenül szükséges. A túlzásoktól tartózkodjunk.

Ha nem kell, ne akarjunk elméleti ismereteket átadni, hiszen általában ilyenkor bonyolult fogalmakat, elveket kell magyaráznunk, amiket az átlagos felhasználó elsőre nem is ért meg.

A tananyag tervezésénél kövessünk egyfajta gondolatmenetet. Arra fűzzük fel mondanivalónk lényegét. Ez lehet a gép használatbavételének sorrendje (bekapcsolás, az elinduló operációs rendszer kezelőfelülete, segédprogramok, elindítása stb…), vagy lehet feladatközpontú is (Hogyan lehet egy feladatot elvégeznem?). Célszerűen lehet keverni a két módszert is.

Az anyagban kell kellő időt tartalékolni a gyakorlásra. Legalább annyit kell gyakorlással tölteni, mint amennyit az új ismeret átadására fordítottunk.

Nem szabad az anyagnak nagyon „sűrűnek” lenni. Ha oktatás közben észrevesszük, hogy a kedves felhasználó nem ért egy kukkot sem, akkor meg kell állni, és újra el kell magyarázni a meg nem értett részeket. Ennek megfelelően esetleg az időből kifutunk, de az nem olyan nagy baj.

Az időbeosztást gondosan meg kell tervezni.

- Nem célszerű egy hét alatt lezavarni napi négy órában egy tanfolyamot, mert a résztvevőkre zúduló hatalmas információmennyiséget nem tudják feldolgozni.
- A heti egy alkalom azt eredményezi, hogy az eltelt idő miatt a résztvevők többsége nem emlékszik a korábban elhangzottakra. Célszerű legalább heti két alkalmat adni az oktatásra, lehetőleg nem egymás utáni napokon.
- A napi óraszám a 2-4 tanítási órában jelölhető meg, két tanítási órát egyben tartva. 1,5 óra után mindenképpen szünetet kell tartani.

Hagyni kell a tervezés során egy vagy több alkalmat a felmerülő kérdések megválaszolására.

Célszerű valamilyen vizsgát szervezni az utolsó alkalomra. Mivel a kezdeti tudás sok féle lehet, ezért a tanfolyam hatékonyságát úgy lehet megmérni, hogy ugyanazokból a kérdésekből egy felmérést végzünk a tanfolyam elején és a végén rendezett felmérés eredményét összevetjük az elején elvégzett teszttel.

Egy számítógépes tanfolyam célszerűen úgy működik, hogy minden hallgató egyedül ül egy megfelelő gép előtt – ez alapfeltétel. A tanfolyam anyagának rövid tömör jegyzetéről is célszerű gondoskodni, akár témavázlat formájában is.

10.7.2 Rendszer betanításához szükséges oktatás

Az általunk fejlesztett rendszer betanítása csak az után képzelhető el, hogy meggyőződtünk, hogy általában a jövendő felhasználók a megfelelő minimális szinten képesek kezelni a számítógépet. Az oktatás során azt a sorrendet célszerű követni, ahogy a dolgozók a rendszert használni fogják. Az anyag nagyjából a következő legyen.

- A rendszer célja, tudása. A rendszer képességei és a jelenlegi valóság kapcsolata.
- A rendszer telepítése (azoknak, akiket ez érint)
- A rendszer indítása, hardver és szoftverfeltételei
- A program menüpontjainak, áttekintése először, általános tudnivalók a program működéséről, kapcsolatok más programokkal.
- A program áttekintése a munkafolyamatok sorrendjében. A képernyők magyarázata, listák megtekintése, nyomtatási képek megtekintése.
- Egyes feladatok begyakoroltatása.
- Mire kell vigyázni
- Mit kell tenni, ha hiba van.

Az időbeosztásra, a keretfeltételekre (gépek száma stb…) vonatkozó korábban elmondottak most is érvényesek. Nyilván lesz a felhasználók között, akik a fejlesztés során együttműködtek a fejlesztővel, azaz többet tudnak a rendszerről. Hagyni kell őket az oktatás során érvényesülni, hogy ezáltal is elmélyítsék tudásukat.

Meg kell kívánni azt, hogy az első tanítási szakasz után a felhasználóknak legyen lehetőségük a tanultak begyakorlására, azaz legyen rá idő és hely.

10.8 Garancia, az elkészült rendszerek további gondozása

A jelenlegi magyarországi törvények szerint minden eladott új termékre kötelező egy évig jótállást adni az eladónak. A szoftverek terén a jótállás kissé összetett probléma. Ha valaki bemegy a boltba és megvesz egy dobozos terméket, akkor tulajdonképpen nem a szoftvert veszi meg, hanem a szoftver felhasználási jogát! A szoftver tulajdonosa továbbra is az eredeti fejlesztő. **Ebből következően nem köteles semmiféle garanciát adni az eladónak a szoftver működőképességére vonatkozóan!**

Az úgynevezett „dobozos termékeken” vagy a belsejükben általában található egy licensz szerződésnek nevezett papír, amely azt mondja ki többek között, hogy a program használati jogát abban az állapotban vásárolták meg, ahogyan azt a dobozba csomagolták. Ha a program nem működik az elvárásoknak megfelelően, akkor sincsen semmiféle jogi következménye az eladóra nézve. Az eladó nem vállal garanciát a

program működéséért. Ugyanígy a fejlesztő se vállal garanciát. Ha a szoftver kárt okoz, akkor egyes szerződések a dobozos termék áráig vállalnak garanciát, de ez alapvetően nem jellemző.

Más a helyzet akkor, amikor egy konkrét megrendelésre kell konkrét programot fejleszteni. Ekkor elvárható, hogy a program ne működjön hibásan. Azért nem hibátlant írok, mert a korábban megtárgyaltuk, hogy hibátlan program nem létezik.

Általában a Megrendelő és a Fejlesztő közötti alku tárgya, hogy a program működésére vonatkozó garancia milyen és hogyan lehet érvényesíteni, illetve mire terjed ki.

A fejlesztőnek kötelezően elő kell írni olyan procedúrákat, amelyek megvédik a program által használt adatokat a megsemmisüléstől. Ezek lehetnek a programba beépített **mentési**, illetve **backup eljárások**, de lehetnek a rendszergazda számára előírt napi, heti, havi mentések. Ha ezeket nem futtatják rendszeresen, akkor természetesen nem lehetnek biztosak abban, hogy egy áramszünet vagy egy hardver meghibásodása nem teszi –e tönkre hosszú idő munkáját.

A program működésének helyességének a tesztelési időszak alatt kell kiderülnie. A tesztelést pedig a majdani felhasználónak is el kell végeznie.

Ezek ellenére a józan Fejlesztői döntés az, hogy a végleges változat átadása után még legalább egy évig minden javítást, a szoftver tudását nem jelentősen növelő módosítást ingyen és bérmentve vagy valamilyen átalánydíjat felszámítva kell elvégezni. El kell dönteni minden hibajavításnál, hogy kinek a hibájából fordult elő, mi a hiba oka. Ha a fejlesztett szoftver a ludas, akkor a javításért nem szabad kérni semmit. Ha a futtató hardver vagy operációs rendszer hibás, akkor vis maior esete van, bölcs döntés, ha a hibát kijavítja a fejlesztő, és semmiféle ellenszolgáltatást nem kér, ha azonban a felhasználó hibája vagy vírusfertőzés okozta a program hibás működését, akkor megfontolás tárgya egyfajta kiszállási díj kérése.

Az egy éven túli gondozás, illetve a továbbfejlesztés lehetősége külön megállapodás tárgya a megrendelő és a fejlesztő között.

Más a helyzet akkor, ha egy szoftvert több tucat megrendelőnek is értékesítettünk. Feltehetően ez a szoftver kellően stabil ahhoz, hogy ő maga programhibákat ne okozzon. Ilyen helyzetben a fenti garancia – kiszállunk és megjavítjuk – nem működik megfelelően. Ekkor meg kell szervezni egy helpdesk lehetőséget, egy telefonszámot, levél vagy E-mail címet, ahol feltehetik a hibával kapcsolatos kérdéseiket a felhasználók és arra rövid időn belül választ is kapnak. Ha nem voltunk kellően körültekintőek és a programunk hibáktól hemzseg, akkor kötelességünk a hibák kijavítása után mindenkinek elküldeni a javított programváltozatot vagy értesíteni őket arról, hogy hogyan kaphatják meg a javítást. Ez felvet még egy kérdést. Mi van azokkal, akik a programot jogtalanul használják. Ők is kérhetek javítást, szupportot? Természetesen nekik nem jár, de a jogosságot nekünk egy konkrét hívás esetén le kell ellenőriznünk. Ha kis eladott darabszámról van szó, akkor könnyű ezt ellenőrizni, de dobozos termékek esetén a visszaküldött regisztrációs kártya a jogosság elismerésének az alapja. A nagy fejlesztő cégek általában átadják a szupportot a kereskedőknek, mondván ők vannak közelebb a vásárlóhoz, legyen ez az ő költségük.

Azt szokták mondani, hogy egy program életciklusa általában nem hosszabb 3-5 évnél. Ennek megfelelően nekünk szoftverfejlesztőknek csak maximum 5 évre kell munkát szereznünk, mert legkésőbb 5 év múlva úgyis megkeresnek a korábbi megrendelők és kérik a továbbfejlesztést.

11 Modern alkalmazásfejlesztési módszerek

11.1 A probléma

A programozási nyelvek fejlődésével a megoldandó feladatok is egyre bonyolultabbakká váltak, így hiába lettek egyre hatékonyabb eszközök a programozók kezében a megoldandó feladatok az eszközöket elégtelenné tették.

Megfigyelhető a programozás-módszertanok fejlődésében az alábbi irány:

- Egyszerű feladatok – strukturálatlan programok – gépközeli programozási nyelv
- A feladatok bonyolódnak – moduláris programok – középszintű programozási nyelv
- A feladatok többrétűek, a felhasználói felület fontossá válik – strukturált programozási módszer – magas szintű nyelvek.
- Grafikus rendszerek, elsődleges a felhasználói felület – Módszer? - Objektum orientált programozási nyelvek.

A fenti lista alapján világossá vált, hogy új módszerre is szükség van.

Az OOP programok megjelenésekor nem volt módszer, amivel igazából kihasználhatták az OOP előnyeit. Jellemzővé vált, hogy a programok mérete bonyolultsága és ezzel párhuzamosan az együttdolgozó fejlesztők száma is nőtt. Az OOP-hoz tartozó módszerek a 90-es években fejlődtek ki és a 90-es évek végére váltak szabványosakká és a 2000-ik év után terjedtek el széles körben.

A programozás mindig is a valóság egy szeletének modellezését tűzte ki céljául. A programnyelvek és módszerek a valóság kiszemelt részét egyre inkább lemodellezik.

A modellezés során azonban mindig lezajlik a valóság egyszerűsítése – ezt hívják **absztrakciónak**, – és az általános esetek megkeresése – ezt hívják **általánosításnak**.

Minél összetettebb egy feladat, a megoldásához annál bonyolultabb modellt kell alkotni, azonban a bonyolultság átcsap az áttekinthetetlenségbe, ha nem vigyázunk. Az OOP megjelenésével olyan programtervezési módszereket kellett találni, amelyek illettek a bonyolultabb feladatok megoldásához, de kellően egyszerűen és áttekinthetően és megfelelően pontos leírást adtak a valóságra.

11.1.1 A modellalkotás hármas szintje

Az OOP megjelenésével megjelent a szoftverek tervezésének, azaz a modellalkotás hármas szintjéről:

Koncepcionális szint – A valóság elemzése és fogalmak kiválasztása, amelyek leírják a valóságos folyamatok számunkra szükséges részét. Ezen a szinten a valóság objektumainak és a közöttük lévő kapcsolatoknak a szintjén alkothatunk fogalmakat és absztrakt modelleket.

Technikai szint – Ezen a szinten megalkotjuk azokat a technológiai elveket, amelyekkel átültetjük a modellünket a konkrét számítógépes megvalósításba.

Implementációs szint – Megalkotjuk a számítógépes programot.

A fenti szintek esetén mindig beszélhetünk vázlatosabb/elvi és konkrét/technikai nézetről. Az elvi nézet a feladatot mindig általánosan fogalmazza meg, a konkrét megfogalmazás figyelembe veszi a sajátosságokat is.

Amikor egy magasabb szinten megfogalmazunk egy elvi elemet, az egyúttal meghatároz alacsonyabb szinten lévő lehetséges megoldási módokat, azaz **absztrakt megoldási** módot jelöl ki.

Amikor a modellt elkészítjük, tehát megírjuk a konkrét programot, akkor az absztrakt megoldásokat konkretizáljuk, azaz a **modellt leképezzük**.

A modellek készítésekor létezik egy absztrakciós szint, ami a jelölésrendszerhez kapcsolható. AZ OOP-nak szüksége van új jelölésrendszerre, hiszen az algoritmusleíró nyelv, a folyamatábra és a struktogram nem elegendő.

11.2 Fejlesztési filozófiák

Az alábbi fejlesztési elvek jellemzően a struktúrált módszertanokra jellemzők:

11.2.1 Folyamatorientált módszer

11.2.2 Adatfolyam orientált módszer

11.2.3 Strukturált módszertan

11.2.4 Objektum-orientált módszertan

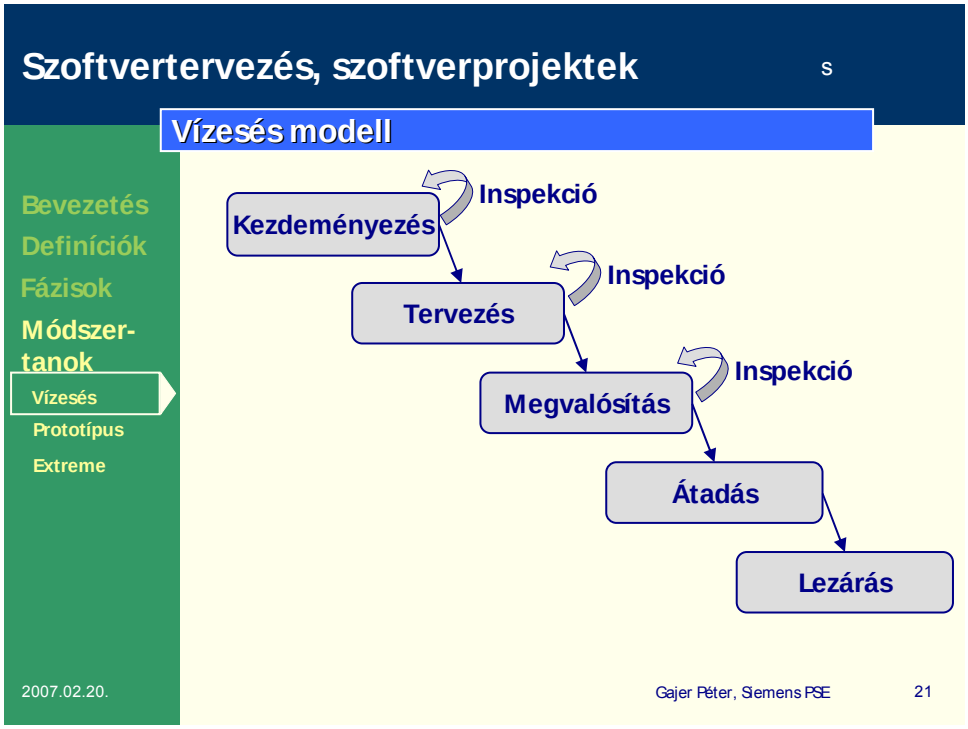
11.3 Modellek

11.3.1 Vízesés modell

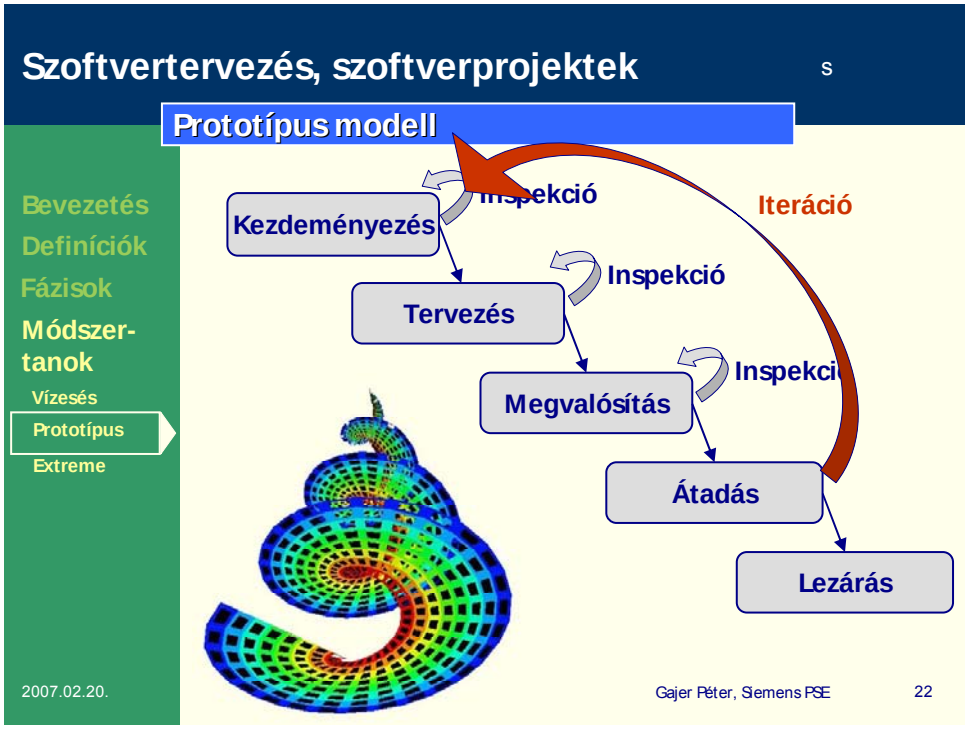
A fázisok szigorúan egymást követik, különállóak. Egy tevékenység akkor kezdődik, ha az előző végetért.

Javított változat:

- Az egyes lépések után validálás és visszalépési lehetőség,
- A fázisok után mérföldkövek (milestones), és előírásos dokumentáció
- Kiterjesztés a rendszerfelügyeletre
- Logikai és fizikai tervezés szétválasztása



Prototípus modell



Spirálmodell

11.4 Módszertanok

A programfejlesztés bizonyos mértékig zsákutcába került a 70-es évektől kezdődően. A szoftverek komplexitásának növekedésével és az objektumorientált nyelvek megjelenésével, terjedésével egyre sürgetőbb lett új, mindenki által használható tervezési, elemzési módszerek és egységes nyelvek, jelölésrendszerek kidolgozása. A ’90-es évek elejéig több javaslat is született, ezek többsége azonban nem felelt meg az előbb felsorolt igényeknek (legalábbis nem mindegyiknek), illetve a fejlesztendő rendszer bonyolultságának, az egyre növekvő problématér nagyságának kezelése is kívánalmakat hagyott maga után.

Ekkor azonban jött a *Nagy Áttörés*: olyan új módszertanok jelentek meg, amelyek sikeresen megfeleltek a kihívásnak, és kiállták a fejlesztők próbáját. Ilyen eszközök voltak például: Coad-Yourdon, Fusion, Martin-Odell, OMT, OOSE, Shlaer-Mellor, stb. [1]

Először is gondoljuk végig, hogy egy átlagos, hétköznapi fejlesztőnek (persze ha létezik ilyen…) milyen igényei lehetnek egy tervezési módszerrel kapcsolatban:

- jól elkülöníthető, személyekre és csoportokra egyértelműen felbontható, pontosan definiált feladatokat határozzon meg, elősegítve a hatékony csapatmunkát;
- pontosan definiálja a fejlesztendő terméket;
- a termék minőségének méréséhez (és eléréséhez) egyértelmű mérési szempontokat állapítson meg;
- a felhasználói igényeket teljes mértékben vegye figyelembe, a fejlesztők és a megrendelők között magasfokú együttműködést tegyen lehetővé;
- az egyéni és csoportos feladatok egységes kezelését is tegye lehetővé, ugyanakkor ne jelentsen túl szigorú megkötéseket sem, ezáltal biztosítson teret az egyéni fejlesztői kreativitásnak.

Természetes igény, hogy mindezen követelményeknek egy szemléletes, sokatmondó ábrázolásmódot feleltessünk meg, amely érthető mindenki számára. Ezáltal a fejlesztők közötti belső, és a kereskedők, megrendelők, felhasználók felé irányuló külső kommunikáció is egyszerűbbé, egyértelműbbé válhat.

A szoftverek komplexitásának növekedésével és az objektumorientált nyelvek megjelenésével, terjedésével egyre sürgetőbb lett új, mindenki által használható tervezési, elemzési módszerek és egységes nyelvek, jelölésrendszerek kidolgozása. A ’90-es évek elejéig több javaslat is született, ezek többsége azonban nem felelt

Három élvonalbeli fejlesztő (Grady Booch, Ivar Jacobson és James Rumabugh) felismerte az igényt egy egységes módszertan kidolgozására, így összefogtak, hogy az addig megszületett metodikák előnyös tulajdonságait kiemeljék, és azok alapján *egy* közös koncepciót dolgozzanak ki. Az ő munkásságuknak köszönhetően, több más szerző módszertanának és tapasztalatainak felhasználásával született meg 1997-ben az

UML (Unified Modeling Language) *modellezőnyelv*, illetve 1998-ban a RUP (Rational Unified Process) *módszertan* (később több, UML-re építő módszertant is kidolgoztak a szoftvercégek).

11.4.1 Inkrementális fejlesztési módszertan

Mindeközben alakult ki az a fejlesztési metodika is, amelyet *inkrementális fejlesztésnek* nevezünk. Ezt az a felismerés hívta életre, hogy a gyakorlatban a legtöbb szoftver életének nagyobbik részét teszi ki (és a költségek nagyobb részét is igényli) az újabb és újabb verziók előállítása, a különböző módosítások végrehajtása, korábbi hibák kijavítása, hiányosságok pótlása. Ebben a folyamatban az új szoftver mindössze az első változat előállítását képviseli. A „semmiből" létrehozott verzió után az újabbakat mindig a meglévő változatok módosításával, kiegészítésével hozzuk létre.

Egy-egy verzió megvalósítása azonban hosszú ideig (akár évekig is) elhúzódhat, közben a fejlesztők egyre több dolgok megtanulnak, tapasztalnak, így az új ismeretek fényében szükség lehet a követelmények módosítására (ugye ismerős a helyzet?). Ezért célszerű lehet egy verzió készítését több kisebb lépésben, alváltozatok egymásutánjaként előállítani. Ezt a folyamatot nevezzük a fejlesztés inkrementális modelljének. Ennek során lehetőség van arra, hogy a rendszer problematikus részeit fejlesszük ki először, majd ehhez hozzáfejlesztjük az újabb és újabb részleteket. (Először többnyire a rendszer viselkedését és kezelési stílusát érzékeltetjük, vagyis a felhasználói felületeket készítjük el.)

A rendszer tehát nem egyenletesen fejlődik, hanem bizonyos részeivel egészen a megvalósításig előreszaladunk, tapasztalatokat gyűjtünk, és az elkészült részekhez rakjuk hozzá a még hiányzókat.

Architektúra szemléletű fejlesztés

A másik, meghatározó paradigma az *architektúraszemléletű fejlesztés*. Ez azt jelenti, hogy a rendszer architektúráját különböző nézetekben képzeljük el, így a modellt sokkal jobban le lehet írni, mintha egyetlen diagramba szeretnénk mindent besűríteni.

11.5 Eszközök UML - Unified Modelling Language

Ez a jelölésrendszer az OOP elterjedése kapcsán alakult ki, de Ő maga nem módszer. A rendszer neve UML Unified Modelling Language – Egységesített Modellező Nyelv.

Az UML egy szabványos, egységesített *modellezőnyelv*, amelynek segítségével a fenti leírások, fejlesztési modellek rendkívül jól szemléltethetőek; a tervezés, a specifikáció, a dokumentálás mind grafikus formában, beszédes ábrák, diagramok, táblázatok segítségével végezhető. A legtöbb vezető szoftvervállalat felismerte már az UML-ben rejtőző lehetőségeket, foglalkozik a szabvány továbbfejlesztésével, így ma már az UML-eszközök kínálata szerfelett széles, mindenki megtalálhatja a számára legmegfelelőbbet.

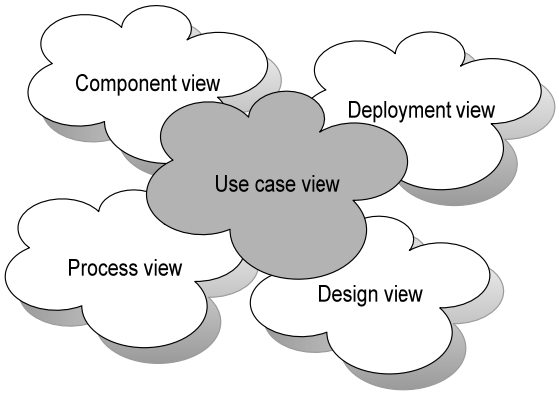
Az UML használatára elterjedt eszközök: Microsoft Visio, ArgoUML – ingyenes, Umbrello – UNIX alapú, ingyenes, Poseidon for UML, Dia - ingyenes

A szoftverfejlesztők egymás közötti, és a felhasználók felé irányuló kommunikációt csak egy közösen elfogadott, mindenki által ismert modellezőnyelv teszi lehetővé. A legtöbb fejlesztési módszertan ajánl valamiféle jelölésrendszert, viszont a nyelv és a tervezési módszer élesen elkülöníthető (ha kész a terv, és azt valamilyen módon ismertetni tudjuk a többiekkel, akkor már mindegy, hogy hogyan, milyen módon jutottunk el a kész tervig). Ezt ismerték fel az UML készítői, és fejlesztették ki a jelölésrendszert.

A grafikus szemléltetés rendkívül hatékony módszer, azonban szem előtt kell tartanunk, hogy a fejlesztésben részt vevő különböző szakemberek mind különböző szemszögből szemlélik a rendszert, illetve ugyanaz a szakember is láthatja másképp ugyanazt a rendszert a fejlesztés más és más szakaszaiban. Ezért olyan módszert kell alkalmazni, amely képes kezelni ezt a sokrétűséget, viszont kellően egyszerű ahhoz, hogy széles körben elterjedjen.

11.6 Az UML nézetei, diagramjai

Az UML-ben mindez úgy valósul meg, hogy egy rendszerhez több különböző nézetet rendelünk, melyek kiegészítik, és bizonyos értelemben át is fedik egymást (lásd az ábrát). A rendszerről oly módon kapunk teljes képet, ha ezekre a nézetekre *együtt*, egy egészként tekintünk.



Az UML nézetei

A használati eset nézet (use case view)

A rendszer viselkedését, funkcionalitását írja le a szereplők és a feladatok megjelölésével, a felhasználó szemszögéből nézve. (A *szereplő* (actor) olyan személy vagy elem, amely kapcsolatban áll a rendszerrel, és aktívan kommunikál azzal, funkciókat indít el, vagy hajt végre.) A használati esetek jól meghatározott funkciók, amelyek végrehajtása üzenetváltást kíván. Meghatározó szerepet játszanak a fejlesztési folyamatban, hiszen a működés leírása a többi nézetet is jelentősen befolyásolja.

A komponens/implementációs nézet (component view)

A rendszer struktúráját, a programkomponensek, állományok kapcsolatát írja le. Elsősorban a programfejlesztők használják, hiszen az elemek, kódkomponensek egyetlen működőképes rendszerré integrálását valósítja meg.

A folyamatnézet (process view)

A rendszert folyamataira, végrehajtható egységeire bontva ábrázolja. Célja a párhuzamosítható műveletek felismerése, az aszinkron események megfelelő kezelése, ezáltal hatékony erőforrás-gazdálkodás elérése.

A telepítési/működési nézet (deployment view)

A rendszer fizikai felépítését rögzíti, a hardvertopológiát, az adott szoftverkomponensek által igényelt erőforrásokat írja le.

A logikai/tervezési nézet (design view)

Azokat az elemeket, feltételeket határozza meg, amelyek a megfelelő működéshez kellenek. Elsősorban a tervezők és fejlesztők számára fontos, hiszen a rendszer statikus struktúráját, az együttműködést, az objektumok közötti kommunikációt írja le. Itt kell pontosan meghatározni a belső struktúrát és interfészeket is.

11.7 Elemek és relációk

A rendszer egyes nézeteinek statikus és dinamikus sajátosságait különböző diagramokkal fejezhetjük ki, amelyek a rendszer elemei közötti *relációkat* írják le, több különböző szemszögből nézve.

Az UML négy relációtípust különböztet meg:

függőség (dependency)

Két elem között akkor áll fenn, ha az egyik (a független) elem változása hatással van a másik (a függő) elemre. Kölcsönös a függőség akkor, ha mindegyik elem hatással van a másikra.

Grafikus ábrázolásában a szaggatott nyíl a független elem felé mutat.

asszociáció (association)

Az objektumok kapcsolatát, ezek struktúráját határozza meg. Speciális esete a rész-egész viszony, amely kétféle lehet: aggregáció vagy kompozíció. Aggregáció esetén a rész az egészhez tartozik, de önmagában is létező entitás, míg kompozíció esetén a rész önmagában nem létezhet, csak az egész elemeként.

A szemléltető nyílon jelöljük az asszociáció irányát, multiplicitását. Rész-egész viszony esetén az egésznél lévő vonalvég egy csúcsára állított, aggregációnál „lyukas”, kompozíciónál tömött rombusz.

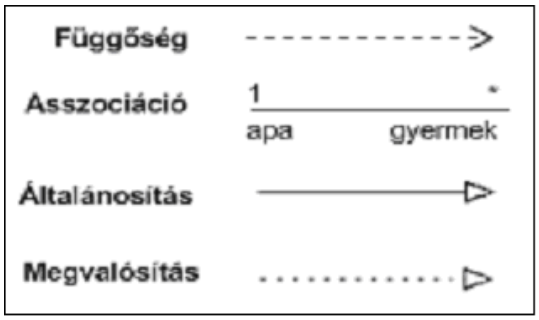
általánosítás és specializáció (generalization/ specification)

Az objektumok speciális viszonya, gyermek-szülő kapcsolat, amelyben a fölérendelt elem az általános, az alárendelt a specializált. Ábrázolása egy „lyukas” nyíl, amely a szülő felé mutat.

megvalósítás (realization)

Annak kifejezése, hogy egy osztály biztosít egy másikat arról, hogy elvégez számára egy bizonyos feladatot.

Grafikus szimbóluma egy szaggatott, „lyukas” fejű nyíl.



A relációk UML szimbólumai

11.8 Diagramok

A *diagramok* olyan gráfok, amelyek csomópontjai elemeket, élei az elemek közötti kapcsolatokat képviselik. A különböző diagramok közös elemeket is tartalmazhatnak, hiszen ugyanazt a rendszert ábrázoljuk többféle megközelítésben. Az UML-ben két nagy csoportról, statikus és dinamikus diagramokról beszélünk.

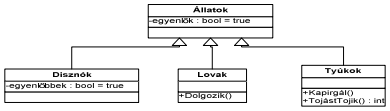
A statikus diagramoknak öt fajtáját különböztetjük meg:

Osztály-diagram (class diagram)

Az osztályok, interfészek, ezek együttműködésének és kapcsolataiknak ábrázolására szolgál.

Egy tipikus osztálydiagramot mutat a következő ábra, melyet Orwell: Állatfarmja alapján rajzolhatunk fel. (Bizonyára mindenki ismeri a nagysikerű regényt.)

Az alábbi ábráról jól látható, hogy az osztályokat olyan téglalapokkal jelöljük, amelyek három részből állnak: felső harmadában az osztály neve, középen az osztály attribútumai, alul pedig az osztályhoz tartozó metódusok szerepelnek.



Osztálydiagram („Minden állat egyenlő, de egyes állatok egyenlőbbek a többinél”)

A nyilak mentén érvényesek az öröklési szabályok, azaz minden gyermek-osztály rendelkezik egy `bool` típusú `egyenlők` attribútummal, míg a Disznók osztálynak még egy `egyenlőbbek` nevű attribútuma is van. A Lovak dolgoznak, a Tyúkok kapirgálnak és bizonyos számú tojást tojnak (például éves szinten).

Az attribútumok előtt szereplő apró jelek a következőket jelenthetik:

+ : public

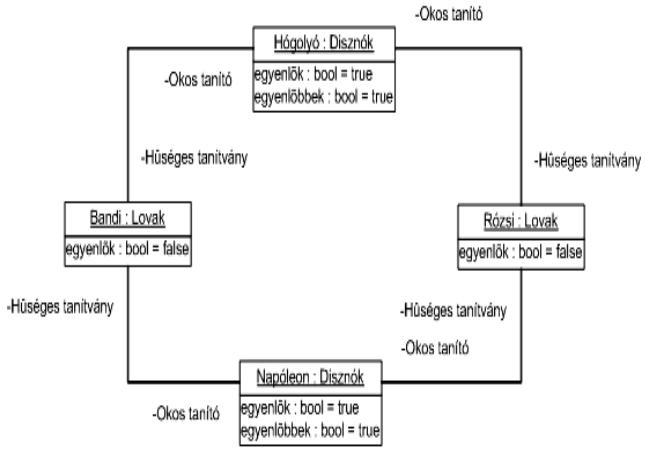
- : private

: protected

attribútumról van szó.

Objektum-diagram (object diagram)

Az osztály-diagram elemeinek pillanatnyilag létező példányait, azok kapcsolatait szemléleti.

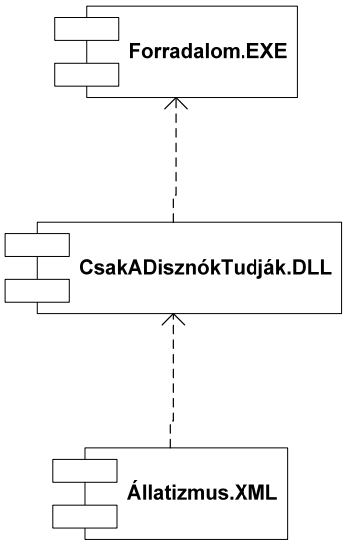


Objektumdiagram („A tanítás és a szervezés munkája természetszerűen a disznókra hárult, mert általánosan elismerték, hogy ők a legokosabbak az állatok között… Leghűségesebb tanítványuk a két igásló lett, Bandi és Rózsi…")

Az objektumokat két részből álló téglalapok reprezentálják, amelyek felső felében az objektum nevét és osztályát, alsó felében attribútumait tároljuk (aktuális értéküket is feltüntetve).

Komponens-diagram (component diagram)

A komponensek egymáshoz való viszonyát fejezi ki. Ha egy komponens osztályok, interfészek és köztük lévő kapcsolatok együttese, ez az ábrázolásmód szoros kapcsolatban áll az osztály-diagrammal.

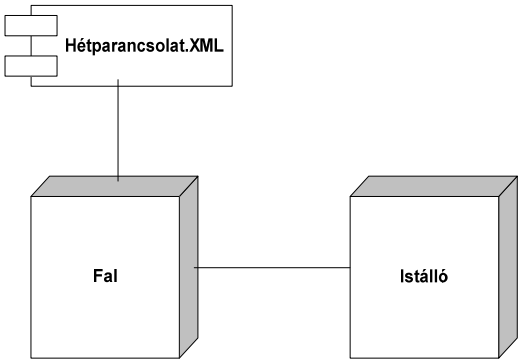


Komponens-diagram („Nem tudták, hogy az Őrnagy megjósolta Forradalomra mikor kerül sor, és jó okkal nem is gondolhattak arra, hogy az ő életükben bekövetkezik, de világosan látták, hogy kötelességük felkészülni rá. …a rendszernek az Állatizmus nevet adták.”)

A komponenseket a fent látható módon téglalapokkal jelöljük, bal oldalukon két kis „fogacskával”. A szaggatott nyilak az mutatják, hogy mely komponens melyiknek szolgáltat valamilyen információt, eljárást, stb.

Telepítési/működési diagram (deployment diagram)

A futás közben igényelt erőforrásigényt, és a csomópontokon működő komponenseket ábrázolja.

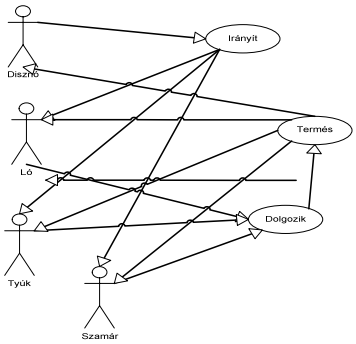


Telepítési diagram („Ezt a Hétparancsolatot most fel fogják írni a falra, ezek változtathatatlan törvények lesznek, és az Állatfarm valamennyi állatának mindörökké ezek szerint kell élnie.”)

Az erőforrásokat téglatestek ábrázolják, ezek közötti kapcsolatot, illetve a szoftverkomponensek „hovatartozását” szemlélteti a fenti ábra.

Használati eset (use case) diagram

A valós rendszer *szereplőit*, ezek kapcsolatát és tevékenységeit mutatja be. A rendszer szervezése, viselkedésének leírása és ellenőrzése szempontjából létfontosságú!

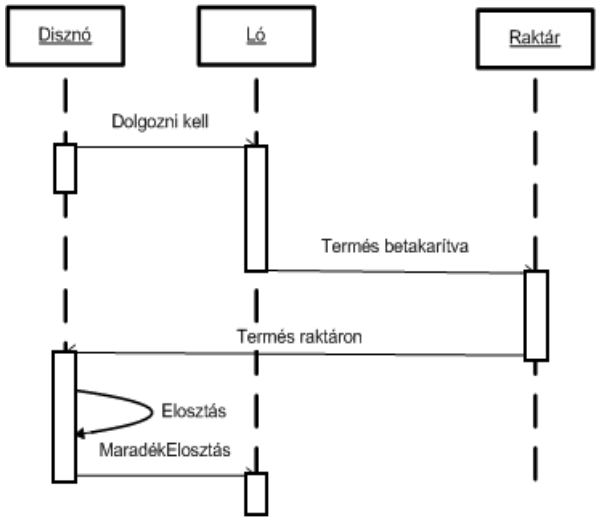


Használati eset diagram („Mennyit gürcöltek, izzadtak, hogy begyűjtsék a szénát!... De a disznók olyan okosak voltak, hogy minden nehézséget le tudtak győzni.”)

Az UML diagramok másik csoportja, a dinamikus (más néven viselkedés-) diagramok az objektumok egymásra hatását, kommunikációját, üzenetváltásait mutatják be. Négy típus sorolható ide:

Szekvenciadiagram (sequence diagram)

Az üzenetek küldésének és fogadásának időrendi sorrendjét határozza meg, a használati esetekből kiindulva.

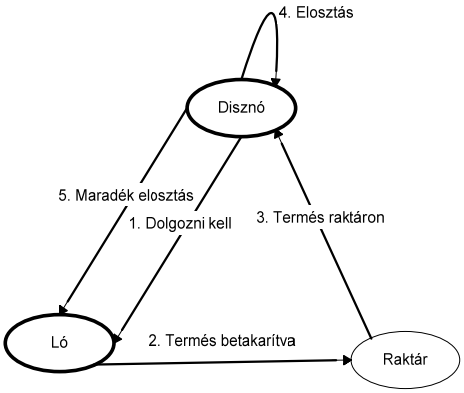


Szekvenciadiagram („Az állatok ebben az évben úgy dolgoztak, mint a rabszolgák.”)

Itt az üzenetváltás szereplőit nagy téglalapokkal jelöljük az ábra tetején. Az idő múlását a függőleges tengely szemlélteti, a cselekvéseket a szaggatott vonalú tengelyeken lévő hosszú téglalapok (ezek hossza a cselekvés időtartamával arányos). Az üzenetek a küldőtől a fogadóig húzott nyíllal szerepelnek az ábrán.

Együttműködési diagram (collaboration diagram)

Az üzeneteket váltó objektumok kapcsolatát, és az üzenetváltás struktúráját ábrázolja. A szekvenciadiagramból egyszerű algoritmus alapján megkapható.

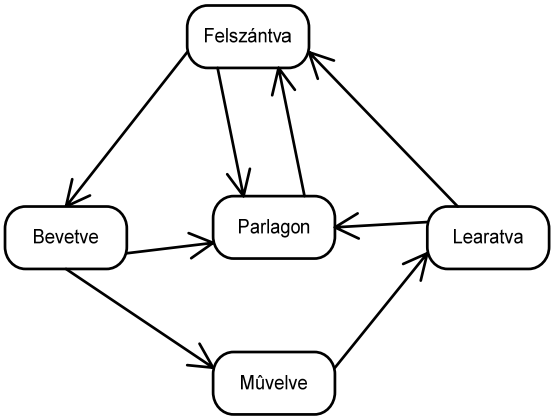


Együttműködési diagram a szekvenciadiagram alapján

Az üzenetküldőket és –fogadókat itt egyszerű ellipszisek jelképezik, az üzenetek itt is a fentihez hasonló nyilak, rajtuk az üzenet küldésének relatív időpontja (az üzenet sorszáma) és az üzenet neve.

Állapot- vagy állapotátmeneti diagram (state-chart)

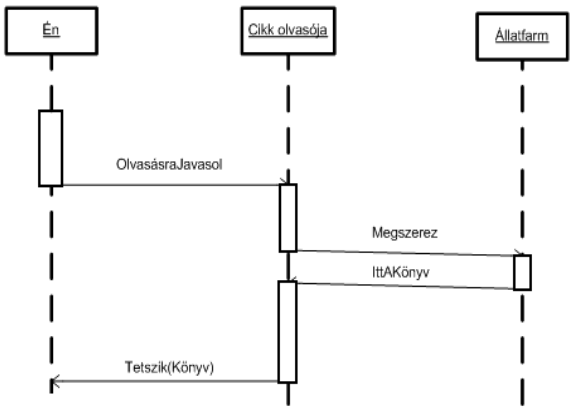
A diagram csomópontjai állapotok, az irányított élek az állapotok közötti átmeneteket reprezentálják. Rendkívül fontos az eseményorientált viselkedés vizsgálatánál.



Állapotátmenet-diagram („Ez a munka szigorúan önkéntes volt, de aki kihúzta magát belőle, annak felére csökkentették a fejadagját.”)

Aktivitás- vagy tevékenység-diagram (activity diagram)

Speciális állapotdiagram, amely a végrehajtandó tevékenységek folyamatát mutatja. Jelentősége az objektumok vezérlési folyamatainak tervezésénél a legnagyobb.



Aktivitás-diagram

Amint ezekből az egyszerű példákból is látszik, az UML rendkívül sokoldalú modellezőnyelv. Remélem, sikerült mindenkit meggyőznöm arról, hogy a valós (vagy valósnak vélt) folyamatok több oldalról, több szempontból történő vizsgálata mennyire fontos.

Az UML alkalmas arra is, hogy mindennapi munkánkban alkalmazzuk. Az UML előnye, hogy a jelölésre koncentrál, nem a konkrét tervezési módszerekre vagy megvalósításokra, így bármit le lehet írni vele. Egy UML-lel megtervezett rendszer pedig megvalósítható C++-ban, Javaban, C#-ban, VB.NET-ben, PHP-ban ...

11.8.1 Prototípus (prototype)

11.8.2 Az eXtrém Programozás- egy új programfejlesztési paradigma (módszer)

Az eXtrém Programozás az alábbiképpen foglalható össze:

- Az emberségesség és a hatékonyság összeegyeztetésére tett kísérlet
- Társadalmi jellegű változásokra irányuló mechanizmus
- A fejlődés egyik útja
- Fejlesztési stílus
- Szoftverfejlesztési diszciplína

Az XP fő célja, hogy csökkentse a változások költségvonzatát. A hagyományos rendszerfejlesztési módszertanokban (pl. SSADM), a rendszerrel szemben támasztott követelmények adottak a projekt elején, és gyakran nem is változnak meg. Ez azzal jár, hogy minél később kell változtatni a követelményeken, ami pedig szoftverfejlesztési projektekben a legvégén sem szokatlan, annál magasabbak lesznek a költségek.

Az XP arra törekszik, hogy ezeket a költségeket csökkentse azáltal, hogy más alapvető értékeket, elveket, és gyakorlatot vezet be. Egy XP-t használó rendszerfejlesztési projekt sokkal rugalmasabb lesz a röptében bekövetkező változásokkal szemben.

Az XP értékei

Nem sokkal ezelőttig csak 4 érték volt fontos az XP-ben, de a második kiadásban bevezettek egy ötödiket is. Ez az öt érték:

- Kommunikáció
- Egyszerűség
- Visszajelzés
- Bátorság
- Tisztelet

Kommunikáció

Az egyik alapvető fontosságú feladat a szoftverrendszer-készítés során, hogy valaki megmondja a fejlesztőknek, hogy mi a feladat. A korábbi módszertanok során ez főként dokumentumok gyártásával történt meg. Az XP technikákat tekinthetjük gyors információrendszerező és -terjesztő technikáknak is, amelyeknek célja, hogy a fejlesztőcsapat minél gyorsabban szerezze meg a szükséges tudást. A cél az, hogy minden fejlesztő ugyanúgy lássa a rendszert, ahogy a majdani felhasználók is látni fogják. Ezért az XP szereti az egyszerű terveket, metaforákat, a majdani felhasználók és a mostani fejlesztők együttműködését, a gyakori szóbeli kommunikációt, és a visszajelzéseket.

Egyszerűség

Az XP azt a megközelítést támogatja, hogy kezdjük el a lehető legegyszerűbben, és folyamatosan dolgozzuk át a programot, hogy egyre jobb és jobb legyen. A különbség eközött a megközelítés között, és a hagyományos megközelítések között, hogy a mai igényeknek megfelelő programot tervezünk és írunk, nem pedig a holnapi, a jövő heti, vagy a jövő hónapi igényeknek megfelelőt. Az XP ellenzői ezt hátrányként fogják fel, mondván, hogy így megeshet, hogy a következő hónapban több munkába fog kerülni átdolgozni a rendszert, ha nem készültünk fel előre új követelményekre, és azt állítják, hogy ez az időveszteség kitesz annyit, mint a megvalósított, de később szükségtelennek bizonyuló feature-ökre fordított idő. Az egyszerűség elve viszont pontosan azt mondja, hogy a program túlbonyolódik, ha tele lesz mindenféle feature-rel, amiről már rég min-

denki elfelejtette, hogy mire való, és a bonyolultság okozta többletmunka viszont sokszorosan meghaladja a folyamatos átdolgozás által generált munkát.

Az előző értékhez, a kommunikációhoz kapcsolódóan, az egyszerűség megkönnyíti és elősegíti a kommunikációt, mert egy egyszerű tervet és a hozzátartozó egyszerű kódot a csapat minden programozója könnyen megért.

Visszajelzés

Az XP-ben a visszajelzés a fejlesztés több területére is vonatkozik:

Visszajelzés a rendszertől: a részegység-tesztek készítésével a programozók közvetlen visszajelzést kapnak arról, hogy milyen állapotban van a rendszer egy-egy módosítás után.

Visszajelzés az ügyféltől: a funkcionális teszteket a programozók együtt készítik az ügyféllel, így mindketten konkrét visszajelzést kapnak arról, hogy milyen állapotban van a rendszer funkcionalitása. Ilyenfajta közös tesztelést 2-3 hetente célszerű végezni, így az ügyfélnek megvan a lehetősége a fejlesztés irányítására.

Visszajelzés a csapattól: amikor az ügyfél új igényekkel áll elő, a csapat rögtön tud rá reagálni, és visszajelzést tud adni, hogy mennyi ideig fog tartani a dolog, és mennyibe fog kerülni.

A visszajelzés szorosan összefügg az egyszerűséggel és a kommunikációval. A rendszerhibákat könnyű jelenteni, hiszen egy részegység-teszt megírása simán megmutatja, hogy mi nem jó a rendszerben, így a rendszer maga mutatja meg a javításra szoruló részeket. Az ügyfél is rendszeresen tudja tesztelni a rendszert, a saját igényeinek megfelelően, amiket az XP-ben 'felhasználói történet'-nek hívnak. Hogy Kent Becket idézzük, 'Az optimizmus szakmai ártalom a programozóknál, és a visszajelzés rá a gyógyír.'

Bátorság

Az XP bátorságra buzdító doktrínáját a legjobban gyakorlati példákkal lehet megvilágítani. Az egyik, hogy mindig a mai igények kielégítésére kell a programot tervezni és megírni. Ez az erőfeszítés azért szükséges, hogy ne gabalyodjunk bele a tervezésbe, és nehezítsük meg saját magunknak a hosszútávú munkát. (Értelemszerűen a 'mai igények' az összes olyan igényt jelentik, amiket ma ismerünk.) Bátorság kell ahhoz is, hogy ne ijedjünk meg attól, hogy a kódot folyton át kell dolgozni. Az átdolgozás azt jelenti, hogy átnézzük a kódot, és olyan változtatásokat eszközölünk rajta, amik egyszerűbbé és átláthatóbbá teszik. Bátorság kell ahhoz is, hogy eldobjunk már megírt kódot. Ahhoz is kell bátorság, hogy felismerjük, hogy nincs tovább értelme körbe-körbe járni egy adott problémán, mert valószínűleg ha másnap visszajövünk és újrakezdjük, egy perc alatt meglesz a megoldás, csak ki kell zökkennie az agynak a rossz kerékvágásból.

Tisztelet

Az XP-ben a tiszteletnek is többféle aspektusa van. Tiszteljük a többi csapattagot, mert dacára a folyamatos változásoknak és integrációnak, sosem csekkelünk be olyan kódot, ami nem fordul le, vagy hibás abban az értelemben, hogy a részegység-teszt elhasal rajta. Úgy általában semmit nem teszünk, ami a többiek munkáját hátráltatja. Magunkat is tiszteljük annyira, hogy csak jó minőségű munkát adunk ki a kezünk közül, és mindig a legjobb tervet igyekszünk kitalálni, és az átdolgozásokat is lehető legjobban megtervezni.

Elvek

Az elvek, amelyek az XP alapját képezik, következnek a fent leírt értékekből, és arra valók, hogy segítsenek döntéseket meghozni. Az elvek konkrétabbak, mint az értékek, és jobban használhatók iránytűként konkrét helyzetekben.

Gyors visszajelzés

A visszajelzés akkor a leghasznosabb, ha hamar megérkezik. A konkrét esemény és a róla érkező visszajelzés között eltelt idő kritikus fontosságú a tanulság levonásának és az esetleges változtatások tekintetében. Az XP-ben, a hagyományos módszertanokkal ellentétben, az ügyféllel való kapcsolattartás sok pici esetben fordul elő, hogy az ügyfélnek tiszta képe legyen arról, hogy mi történik a fejlesztésben. Így a visszajelzései alapján lehet a projektet irányítgatni.

A részegység-tesztek is fontosak a gyors visszajelzés érdekében. Amikor az ember a kódot írja, a részegység-teszt a leggyorsabb közvetlen visszajelzés arra, hogy milyen hatása lett az új kódnak. Továbbá, ha a változás olyan funkcionalitást érint, ami nincs a programozó látóterében, és álmában sem gondolja, hogy arra hatása

lehet az ő kódjának, a részegység-tesztek ezeket is észre fogja venni, és a bug nem akkor derül ki, amikor a rendszer már élesben üzemel.

Inkrementális változtatások

Az XP fáklyavivői azt mondják, hogy Rómát sem egy nap alatt építették. Nem lehet egyszerre nagy változtatásokat készíteni egy rendszeren. Az XP fokozatos változtatásokat javasol. Megeshet, hogy ettől egy rendszernek három hetente lesz új kiadása, mindig csak apró változásokkal. A sok kis lépéssel az ügyfél is jobban látja, hogy merre halad a projekt.

Örömmel fogadott változások

Tuti, hogy semmi sem tuti. Ez az elv azt mondja ki, hogy nem elég, hogy ne tegyünk semmit a változások ellen, pont ellenkezőleg, örüljünk nekik. Ha az egyik szokásos napi találkozó során kiderül, hogy az ügyfél igényei drámaian megváltoztak, akkor a programozók örüljenek neki, és kezdjék el tervezgetni az új iterációhoz szükséges terveket.

Tevékenységek

Az XP-ben alapvetően négyféle tevékenység van.

Kódolás

Az XP már emlegetett fáklyavivői szerint a rendszerfejlesztés egyetlen igazán hasznos végterméke a kód, bár a 'kód' kifejezést szélesebb értelemben használják, mint a hagyományos szemlélet hívei. Kódolás nélkül nincs semmi.

A kódolás jelentheti diagramok megrajzolását is, amikből aztán program lesz, vagy scriptek írását egy webalapú rendszerhez, vagy egy C#-ban készülő objektum megírását, amit majd aztán le kell fordítani.

A kódolás néha ahhoz is kell, hogy a legjobb megoldást megtaláljuk. Az XP szerint előfordulhat az, hogy ha egy problémának több, látszólag ugyanolyan jó megoldása van, akkor mindet meg kell írni, és automatizált tesztekkel kell eldönteni, melyik a legjobb.

A kódolás az egyik eszköz a gondolatok kifejezésére, konkrétan a programozási problémákról keletkező gondolatok kifejezésére. Egy programozó, aki egy bonyolult programozási problémával küszködik, lehet, hogy nem tudja elmagyarázni rendesen a megoldás lényegét a kollégáinak, de meg tudja írni, és meg tudja nekik mutatni a kész kódot. (akár pszeudokód formában is.) A kód, mondják eme álláspont hívei, mindig tiszta és egyértelmű, és nem lehet többféleképpen értelmezni, csak úgy, ahogy a számítógép. A többi programozó pedig úgy fejezheti ki a véleményét a témával kapcsolatban, hogy beleírnak a kódba, vagy hozzátesznek.

Tesztelés

Semmiben sem lehetsz biztos, amíg nem próbáltad ki. A tesztelés általában nem az ügyfél kérése, sőt nem is jól felfogott érdeke. Rengeteg szoftvert adnak ki rendes tesztelés nélkül, ami működik, többé-kevésbé. Az XP azt mondja, hogy nem lehetsz biztos a kódod működőképességében, amíg alaposan ki nem próbáltad. Ez felveti azt a kérdést, hogy pontosan mi is az, amiben nem lehetsz biztos.

Nem biztos, hogy azt kódoltad le, amire gondoltál. Ennek a bizonytalanságnak az eloszlatására vannak a részegység-tesztek. Ezek automatizált tesztek, amik a kódot tesztelik. A programozónak annyi tesztet kell írnia, amennyit csak tud, hogy lehetőleg minden ágát letesztelje a kódnak. Ha minden teszt sikeresen lefut, akkor a kódolás készen van.

Nem biztos, hogy amire gondoltál, az tényleg az, amit az ügyfél akar. Ezért kellenek az elfogadási tesztek, amiket az ügyféllel kell elvégezni, a release-tervezés felfedező fázisában.

Meghallgatás

A programozók nem feltétlenül tudnak az üzleti oldaláról annak a projektnek, amin dolgoznak, noha a rendszerrel támasztott követelményeknek az üzleti oldalról kell érkezniük. Annak érdekében, hogy a programozók megértsék, hogy mire akarják a programjukat használni, meg kell hallgatniuk az üzleti oldal mondanivalóját is. Azt kell belőle meghallaniuk, hogy mire van szüksége az ügyfélnek. Ezenfelül jó, ha azt is megértik, hogy miért van rá szüksége az ügyfélnek, hogy visszajelzést tudjanak adni az ügyfélnek a saját problémájáról, és ezáltal ő maga is jobban értse, hogy mi kell neki.

Az ügyfél és a programozók közötti kommunikáció a Tervezési Játékban van kifejtve, amit az XP leendő és újdonsült hívei a vonatkozó szakirodalomban találnak kifejtve.

Tervezés

Csupán az egyszerűséget véve alapul, mondhatnánk, hogy a rendszert nem is kell tervezni, elég, ha kódolunk, tesztelünk, és odafigyelünk az ügyfélre. Ha ezeket jól csináljuk, a végeredmény egy tutira működő rendszer lesz. A gyakorlatban ez nagyon nincs így. Messzire el lehet jutni tervezés nélkül, de a végén biztos, hogy zsákutcába fogsz kerülni. A rendszer túl bonyolulttá válik, és a belső függőségek átláthatatlanná válnak.

Ezt úgy lehet elkerülni, hogy logikus részekre kell a rendszert bontani. Ha logikus, és jól elkülönülő részekre sikerül a rendszert szétszedni, akkor a függőségek nem fognak gondot okozni, vagyis a rendszer egy részének megváltoztatása nem teszi tönkre a rendszert, és az egyes részegységek bonyolultsága sosem fogja túllépni a kritikus küszöböt.

Gyakorlat

Ezt részletesebben nem fejtem ki, mert sok dokumentáció található az Interneten erről. Alapvetően 12 gyakorlati módszer van, 4 csoportra osztva:

Visszajelzés részletezve:

- Párban programozás
- Tervezési Játék
- Tesztelésen Alapuló Fejlesztés
- Egész Csapat
- Folyamatos feldolgozás
- Folyamatos integráció
- Tervek fejlesztése
- Kicsi release-ek
- Közös nézőpont
- Kódolási Szabvány
- Közös Tulajdonú Kód
- Egyszerű terv
- Rendszer-metafora
- Programozói jólét
- Fenntartható tempó

Az XP ellentmondásos részei

A legellentmondásosabb, legproblémásabb része a dolognak a változás-menedzsment. Mivel az ügyfél közvetlenül kommunikál a programozókkal, leginkább szóban, így két rossz dolog történhet. Az egyik, hogy az összevissza csapongó változásai költséges átdolgozásokhoz vezetnek, a másik az ún. 'becsúszó featuritis', vagyis hogy az ügyfél szép lassan egyre többet és többet követel. Az XP azért nem szereti lepapírozni az ügyfél kéréseit, mert azt mondják, hogy ez a rugalmasság kárára megy, és az esetek túlnyomó részében tovább tart a papírozás, mint maga a munka. (Célszerű az ügyféllel lepapíroztatni a kérdést!)

Nincsenek követelmény-listák, és specifikációk, tehát nehéz számonkérni bármit is. Értelemszerűen ennek következményeként olyankor érdemes XP-t használni, amikor vagy eleve szóba sem kerül a számonkérés, mert pl. belső az ügyfél, vagy olyankor, amikor az elszámolásnak kizárólag az eltöltött idő az alapja, és lehet hülye a t. ügyfél, csak fizesse ki.

Nincs Központi Nagy Terv, ergo megeshet, hogy egy húzósabb változás teljes újratervezéssel jár. Ebből következik az, hogy csak rutinos programozóknak való, mert minél rutinosabb egy programozó, annál nagyobb változás fog kelleni ahhoz, hogy tényleg totál újra kelljen tervezni mindent. Ha meg kiderül, hogy fogpiszkáló, hanem űrrakéta, akkor újra kell mindent tervezni.

Az ügyfél képviselője szerves része a projektnek. Ezt sokaknak stresszt okoz, hiszen minden hiba és tévedés azonnal nyilvánvaló az ügyfél számára is. Ugyanakkor ha az ügyfél képviselője rosszul végzi a munkáját, akkor azon az egész projekt megbukhat. Ha az ügyfél nincs jelen, általában mégis mindenki azt kívánja, hogy bárcsak jelen lenne...

2003-ban megjelent egy könyv, amelyik mindenféle javításokat javasolt az XP-be. A vita arról, hogy mi jó ebből, és mi nem, a mai napig folyik az interneten. A könyv központi gondolata az, hogy az XP elemei függenek egymástól, de igen kevés olyan projekt van, ami egyszerre az összes elemet át tudná venni, viszont ha nem veszi át valaki az összes elemet, akkor nem ér semmit az egész. Egy másik gondolat a könyvből, hogy a 'kollektív tulajdon' fogalma már a szocializmusban is ismert volt, és inkább az lett belőle, hogy ami mindenkié, az tutira nem az enyém, tehát hagyjanak vele békén. A vitából valami olyasmi szűrődik ki, hogy az XP elemei között valóban van egy függőségi fa, és valóban vannak kulcsfontosságú elemek, amelyek hiánya összeomláshoz vezet, de ez minden módszertanra igaz, és ahol valamelyik ilyen kulcsfontosságú elem hiányzik, ott általában már az elején látszik, hogy nem jó ötlet XP-vel próbálkozni. Az XP nem csodaszer a társadalom minden problémájára. Az XP ezenfelül viszonylag kis létszámú fejlesztőcsoportokra lett kitalálva, és a közös tulajdon gondolata ilyen léptékben még működhet, ha a motiváció megvan hozzá. Az nem az XP feladata, hogy motiváljon.

12 Szervezési ismeretek

12.1 Rendszerelméleti alapok

Rendszer

A rendszer egy több alkotóelemből álló, egymással kapcsolatban álló és egymással együttműködő elemek halmaza.

Részrendszer

Egy rendszer jól elkülöníthető része, amely a többi részrendszerrel jól definiálható kapcsolatokon keresztül kommunikál.

Alrendszer

Egy rendszer jól definiálható feladatokat végző része. Jól definiálható felületeken kapcsolódik a többi részrendszerrel.

Elem

Egy rendszer alkotórésze

Környezet

Egy rendszerrel kapcsolatot tartó viszonyok összessége. A környezet határozza meg a rendszer bemenő paramétereit, az kapja vissza a kimenő eredményeket, az határozza meg a működési körülményeket.

Input

A rendszerbe bemenő adatok halmaza

Output a rendszer által visszaadott eredmények halmaza. Egy rendszer outputja lehet más rendszer inputja is.

12.1.1 Elemzés

A

12.1.2 Modellezés

12.1.3 Szervezet elemzés

Cél – Folyamat

Minden szervezet valamilyen céllal jön létre és a működésének menete a követendő cél elérését szolgálják. Céljának elérését egymással kapcsolatban álló, egymással párhuzamosan működő vagy esetleg egymással konkuráló folyamatok alkotják. A folyamatok lehetnek egymás mellérendelt vagy alárendelt viszonyban egymással.

Szervezet kapcsolati rendszere

Semmiféle szervezet nem működhet önmagában. Mindegyik szervezetnek vannak külső kapcsolatai, amelyek szintén összefügghetnek egymással, illetve lehetnek függetlenek is egymástól. A kapcsolatok lehetnek alárendelt, mellérendelt kapcsolatok a szervezettel.

Feladatkör

A szervezetnek minden esetben a céljaiból levezethető feladati vannak. Ezeknek a feladatoknak a halmaza a feladatkör. A feladatok elvégzésével a szervezet céljai is megvalósulhatnak.

Hatáskör

Azon dolgok halmaza, amelyeke a szervezet működése során befolyását tudja érvényesíteni.

Felelősségi kör

Egy szervezet működése során tevékenysége során hat a környezetére és ezeknek a hatásoknak következményei lehetnek. Azok a jelenségek, amelyek kizárólag a szervezet működésének eredményei, a szervezet helyes, célnak megfelelő működése során előre tervezetten keletkezhetnek. Ebben az esetben a szervezet felelősséget vállalhat a tevékenységéért, amelyben azt vállalja, hogy a működése során csak bizoynos események az előre megadott módon következnek be.

12.1.4 Szervezet - szervezeti felépítés

szervezeti felépítési formák és működésük főbb jellemzői

lineáris, funkcionális mátrix szervezeti formák

12.1.5 Gazdasági rendszerszervezés

- szervezés fogalma, fajtái, szakterületei (cél, irányultság)
- alap vagy fejlesztő
- folyamat, szervezet, munka információ

12.1.6 Ismeretelméleti alapfogalmak

Adat, információ, hír

Ezekről a témákról volt szó az adatbázis-kezelés című jegyzetben, továbbá az Informatika kezdőknek jegyzetben.

információ mértéke és hasznossága

A Shannon féle elméletben az információ mértéke a bit, byte, kbyte, stb...

Hasznosságát nem tudjuk mérni, azaz szemantikai szempontból nem mérhető az információ egzakt módon.

hír

Olyan adat, amely információ és amely valódi döntéseket indukálhat.

hírforrás

A hírforrás az adatszolgáltató, de nem az adat fizikai előállításával foglalkozik, hanem az információvá alakításval

Adó

Az amely az adatot fizikai valójában szolgáltatja

Kódoló

Az adatokat egyik megjelenési formájából átalakítja más formába.

Csatorna

Olyan adatátviteli szabványok és eljárások sorozata, amely az adó és a vevő között felépülve alkalmas az adatok folyamatos, de legalábbis hosszú távú átvitelére. Az **adatátviteli csatorna működése** során zajok lehetnek, amelyek az átvitt adatok deformálódását, hibáját okozhatják. Az adatátviteli csatornák egyik értékmérője a sebességük, míg másik értékmérője a hibátlanságuk, zajtalanságuk. Megjegyzendő, hogy a zaj nem minden esetben káros az átvitt adatokra. Főleg abban az esetben nem, ha kellő redundanciával rendelkezik az átviendő adat.

A vevőhöz megérkezve az adatot visszaalakítjuk eredeti formájára, akkor **dekódoló egységről** beszélünk.

Az informatika fogalma

Az informatika az ismeretek megismerésének, azok célszerű elrendezésének és kezelésének tudománya. Az informatikus az a szakember, aki ebben a tudományban jártas.

A informatika tárgyköre

Az informatika a valósággal, a róla alkotott képpel, a rendelkezésre álló technikai erőforrásokkal foglalkozik.

Az informatika kapcsolata a szervezéssel

Az informatika a megoldandó feladatok és a rendelkezésre álló erőforrások megszervezését végzi, azaz a szervezés az informatikai folyamatok egy része.

Irányítás

A folyamatok irányítása egy visszacsatolásos folyamat. A folyamat működését irányítja, megfelelő pontokon visszajelzéseket kap az irányító folyamat, és a visszajelzések alapján az irányító eszköz (irányító folyamat, team stb..) módosítja a folyamat paramétereit.

12.1.7 Rendszerfejlesztési projekt

A projekt fogalma

A projekt hosszabb időn keresztül zajló, több ember összehangolt munkáját igénylő fejlesztési feladat! (Nem programozási, vagy szervezési és programozási)

Az információrendszer (IR) fejlesztési projekt feladata

Az ilyen projektek azt a célt szolgálják, hogy egy információs rendszert alkosson a csapat, amelyben a világ valamelyik részének megkönnyítik az életét.

Egy cégen belül egyszerre több párhuzamosan futó és esetleg egymásnak nem teljesen megfelelő projekt is haladhat, ami kellemetlen ellentmondásokhoz, erőforrás pazarláshoz és egyéb gondokhoz vezethet, ezért ilyen esetben meg kell alkotni a projektek koordináló vagy kormányzó bizottságát. Ennek a bizottságnak a feladat, hogy a felmerülő ellentmondásokat elsimítsa. E bizottság tagjainak nem adminisztratív vagy felső vezetőknek kell lenniük, hanem esetleg a projekt teamek vezetőinek, hiszen az erőforrások elosztása és az ellentétek elsimítása, az együttműködés megszervezésének feladata amúgy is rájuk hárul..

Ezzel el is mondtuk, hogy minden **projktnek szükséges egy vezetőjének** lennie. Ez az ember az, aki összefogja a csapatot és amennyiben bármiféle külső kapcsolatot is kell találni, a projektvezető az, aki a külső kapcsolatokat megszervezi, stb...

A projektek általában nem lerögzített és bebetonozott fejlesztői csapatok, hanem kétféle módon is változik a csapat összetétele:

az egyes projektek között van átjárás és egyik projektben egyik ember programozó, míg a másik projektben vezető lehet és fordítva.

A projektben résztvevők a munka egyes fázisaiban többen vagy kevesebben vannak, attól függően, hogy mennyi és milyen fajta munkára van szükségre a projektnek.

Ezt a munkamegosztás nevezzük **projekt szervezésnek.**

A helyes megközelítésben egy fejlesztő cégnél például az emberek bér, munkaügyi stb.. szempontból tartozhatnak valahová, de a projektekkel kapcsolatban mindig máshol találhatók meg. Azt a folyamatot, amikor megtervezzük, hogy a dolgozók milyen esetekben hol és mit dolgozzanak **mátrix szervezésnek** hívják.

A projektvezetés feltételei

A projetek megfelelő szintű vezetéséhez szükséges, a megfelelő szervezeti keretek biztosítása, a humán erőforrások biztosítása és a megfelelő adminisztrációs erőforrás biztosítása.

A projekt vezetője átlátja a folyamat minden részletét és ő vezeti le a projektet a kialakulástól a végső átadásig. A projektvezetés folyamata során a vezetőség feladata

a tervezés (durva becslés, projekt szintű szabályok) kialakítása,

A projekt ütemezése, azaz az egyes lépcsőfokok elérésének az ellenőrzése és teszteltetése!

A projekt ütemezésének mindenkori korrigálása,

A projektben résztvevő személyekre kiosztandó feladat kiosztása

A projekt végső fázisában az előre megállapított pontokon és módszerekkel a teljesítmények figyelése, értékelése

12.2 Az információrendszer fejlesztés életciklusa

12.2.1 Rendszerelemzés

Előzetes helyzetfelmérés

Az információs rendszer életének első fázisa a rendszerelemzés. Meg kell vizsgálni, hogy a jelenlegi rendszernek melyek a korlátai, milyen egységekből (egyed, egyedtípusból épülnek fel) és miért szükséges a változtatás. A rendszer felmérése során nem törekedünk az azonnali, átfogó rendszerelemzésre, hanem inkább iteratív módon fokozatosan közelítjük meg a megoldandó feladatot.

Rendszertanulmány készítése

Az elemzések eredménye egy rendszertanulmány, aminek segítségével hozzá lehet fogni a konkrét feladatok megoldásához. Ez a tanulmány kiindulópontja a későbbi munkának. A rendszerterv elsősorban a laikusok, illetve a rendszer jelenlegi üzemeltetői számára készített tanulmány.

12.2.2 Rendszertervezés

Átfogó helyzetfelmérés

A rendszertanulmány alapján, az informatikus elvégez egy átfogó felmérést, amelyben a folymatoka minden aspektusból megvizsgál és fokozatosan felderíti a követelményeket, a részleteket, stb.. Ehhez interjúkon vesz részt, amelyekben a megrendelő oldaláról szakértők vesznek részt, míg a rendszer tervezője a szakértők segítségével pontosítja a feladatokat. A helyzetfelmérés alapján a szükségeshez egyre jobban közelítő rendszertervet készít az informatikus.

A **rendszerterv** tartalmazza a szükséges adatszerkezeteket, a használandó eszközöket, a szükséges erőforrásokat, a kritikus algoritmusokat és általában szakmai szempontból előkészíti a rendszerfejlesztés további lépéseit,

12.2.3 Kivitelezés

Programtervezés

A következő lépcső a program megtervezése. A programtervezés során már a figyelembe vett nyelvi elemekkel együtt a rendszertervben lévő adatszerkezeteket és eljárási szabályokat figyelembe véve tervezzük meg a szoftvert. Ez az a szint, ahol még az alap algoritmusok segítségével elindulhat a Down-Top, vagy a Top Down módszer segítségével a rendszer részletezése.

Programozás

A következő lépés a a programozás maga. A programtervezés során részletekre bontott programot a programozók modulonként leprogramozzák, majd az elkészült modulokat összeépítik.

12.2.4 Tesztelés, a rendszer bevezetése

A tesztelés folyamatát tervezni kell. A tesztelésnek az alábbi elvek szerint és módszerek szerint kell lezajlania. (A „Módszeres programozás” c. jegyzet megfelelő részei elolvasandók)

12.2.4.1 A programok tesztelésének célja

A programok tesztelésének célja, hogy a program vajon a bemenetekre a specifikáció alapján megfelelő kimenetet szolgáltatja-e.

A specifikációnak megfelelő programot **helyes programnak** hívják. A programok tesztelése és a hibakeresés során arra törekszünk, hogy az eredeti specifikációnak minél jobban megfelelő, illetve megfelelő programot állítsunk elő. Olyan tesztmódszereket kell használni és olyan hibakereső eszközöket, amelyek a hibák nagy részét kiszűrik. Néhány tapasztalati ténybe, azonban bele kell nyugodni:

A program hibáinak száma és súlyossága exponenciálisan nő a mérettel

A hibajavítás után az összes tesztelést célszerű lefolytatni

A hibát megszüntető okokat kell megtalálni és kijavítani

Gyakran egy hiba megszüntetése több másik hiba megjelenését vonja maga után

A program készítője a legrosszabb tesztelő. A fejlesztőn kívül mással is teszteltetni kell a programot.

A fenti tényeken kívül még egy továbbiról is kell szót ejteni. Nagyobb méretű programok esetén 100%-osan hibátlan programról nem lehet beszélni.

12.2.4.2 A tesztelés kritériumai

A jó tesztelés nagy valószínűséggel felfedi a hibákat

A jó tesztelési eljárásoknak megismételhetőknek kell lenni

Érvényes és érvénytelen adatokra is kell tesztelni

Minden tesztesetet maximálisan ki kell használni, azaz a legtöbb hibát fel kell deríteni

Fel kell tenni a kérdést, hogy **miért nem azt teszi** a program, amit kellene volna és **miért azt teszi**, amit nem kellett volna.

12.2.4.3 Statikus tesztelési módszerek

A statikus tesztelési módszerek a programkód vizsgálatán alapulnak. Ekkor nem futtatjuk a programot.

Kódellenőrzés

A legegyszerűbb lehetőség. Kinyomtatjuk, vagy a képernyőn átnézzük a kódot, miután begépeltük. Célszerű olyan editort használni, amely kiemeli az adott nyelv kulcsszavait, esetleg színnel vagy más módon elkülöníti az adatokat, az értékadó utasításokat. Ha lehet az program bevitelekor használni kell a strukturált írásmódot, ha akkor nem tettük meg, akkor utólag javítani kell a kódon.

Szintaktikai ellenőrzés

A legtöbb fejlesztő eszköz ma már szintaktikailag ellenőrzi a program kódját és a megfelelő sorban ki is írja a hibaüzeneteket. Az interpreteres nyelvek gyakran már a programsor bevitelekor elvégzik az ellenőrzést, míg a compileres nyelvek csak a fordítás során.

Szemantikai ellenőrzés

Az interpreteres nyelvek esetén csak a programozó tudja végiggondolni, hogy programja valóban logikailag megfelelő, az alkalmazott algoritmusok valóban a kellő végeredményt adják, és a kódolás megfelel az algoritmusnak.

A compileres rendszerek esetén előfordul, hogy a fordító figyelmeztet bizonyos utasítások szemantikai problémáira. Gyakran találnak az ilyen rendszerek felesleges változókat, olyan kódrészleteket, amelyek sohasem futnak le, mindig biztosan azonos értéket felvevő változókat, stb. Azok a compileres rendszerek, amelyek kódoptimalizálást végeznek, gyakran olyan kódot hoznak létre az optimalizálás során, amely logikailag nem felelnek meg az algoritmusnak. Ekkor ki kell kapcsolni az optimalizálást.

Inicializálatlan változók

Meg kell keresni a kódban az inicializálatlan változókat, és kezdőértéket kell nekik adni.

Felesleges utasítások kiszűrése

Gyakran kódoláskor az algoritmusnak megfelelő kódot írunk, holott az adott nyelv ugyanazt a funkciót esetleg gyorsabban is meg tudja oldani.

Keresztreferencia táblázat

Ha a programunkban lévő változók értékeinek változását nem tudjuk követni, akkor célszerű keresztreferencia táblázatot készíteni. Erre a legtöbb fordító képes. Ez egy olyan táblázat, amely felsorolja, hogy az adott változó hol kap értéket, illetve hol történik hivatkozás rá a program során. Ennek alapján megállapíthatjuk, hogy mely változókat használjuk a leggyakrabban.

Típuskeveredés

Egyes interpreteres nyelvek a bevitelkor nem ellenőrzik, hogy az értékadó utasítások két oldalán ugyanolyan típusú értékek szerepelnek-e.

12.2.4.4 Dinamikus tesztelési módszerek

A programok hibáinak egy részét a statikus tesztelési módszerekkel ki lehet szűrni, de vannak olyan helyzetek, hogy csak a futás közbeni ellenőrzés segít. Hogy egy-egy teszt minél több tulajdonságot áruljon el a programról az alábbi módszereket lehet alkalmazni:

12.2.4.5 Fehér doboz módszerek

Az utasítások lefedésének elve

A program minden utasítását legalább egyszer végre kell hajtani.

Döntések lefedésének elve

A programban lévő döntések minden következményét végig kell próbálni. A döntéseket igaz és hamis esetben is végig kell próbálni.

A feltételek lefedésének elve

A programban lévő feltételes elágazásokat minden feltételre ki kell próbálni, illetve az logikai összekötő műveleteket minden lehetséges helyzetre ki kell próbálni.

12.2.5 Fekete doboz módszerek

Ekvivalencia osztályok készítése

A lehetséges bemenő adatokat oly módon kell csoportosítani, hogy milyen kimenő adatot várunk tőlük. Ezeket ekvivalencia osztályoknak hívjuk. Minden ekvivalencia osztályra tesztelni kell a programot.

Határeset analízis

Ha a lehetséges bemenő adatok ekvivalencia osztályait helyesen is állapítottuk meg, és úgy találjuk, hogy az osztályokra megfelelő választ ad a program, még mindig meg kell vizsgálni, hogy az ekvivalencia osztályok határeseteit hogyan kezeli le a program. Gyakran az ilyen helyzetben adott hibás eredmény helytelen algoritmusra, gondolatmenetre vagy túlzott egyszerűsítésre vezethető vissza

Stressz teszt

A programokat biztosan rossz bemenő adatokkal is tesztelni kell. A programok fejlesztése során a fejlesztő általában feltételezi, hogy a felhasználó csak helyes dolgokat művel, pedig ez nem így van. A felhasználó sokkal gyakrabban téved, hibázik, mint azt a legtöbb fejlesztő képzelné.

12.2.6 Speciális tesztek

Hatékonysági tesztek

A programok tesztelésének utolsó fázisa, annak megállapítása, hogy milyen hatékony a program, illetve mennyire jól felhasználható. Ha nem fut, vagy a sebessége nem megfelelő, akkor meg kell keresni azokat az okokat, amelyek a megfelelő futást megakadályozzák, és annak megfelelően kell módosítani a programot, akár az algoritmusok szintjére is visszamenve.

Biztonsági teszt

A programoknak stabilaknak kellene lenniük, nem szabadna előre látható okok miatt lefagyniuk. Ezekre valók a biztonsági tesztek.

12.2.6.1 Tesztállapotok

Az elkészült program a tesztelés fázisain végigmenve különböző állapotokba kerül.

A fejlesztők belső tesztelését **alfa tesztnek hívjuk**. Az ilyen állapotban lévő programra azt mondjuk, hogy α teszt változat.

Ha a programot már a jövendő felhasználók egy kisebb csoportja tesztelheti, akkor ezt az állapotot β állapotnak hívjuk, a tesztelőket béta tesztelőknek.

Gyakori, hogy egy program elkészültének fokát a következő vagy ehhez hasonló módon jelezzük: 0.01, 0. 11, stb...

Ekkor az elkészült, letesztelt program verziószámának az 1.0-át szokás írni.

A tesztelés folyamatát a fenti módszerek figyelembevételével meg kell tervezni és a tesztek tapasztalatait, a bemenő adatokat és az eredményeket jegyzőkönyvben rögzíteni kell. Ezt hívják tesztelési dokumentációnak, amely a fejlesztői dokumentáció része.

12.2.7 Program dokumentálása

12.2.8 Rendszer felhasználói kézikönyve

Egy rendszer felhasználói kézikönyve (dokumentációja) az alábbiakat kell hogy tartalmazza:

A felhasználói dokumentációnak a következő részeket kell tartalmaznia:

Általános leírás a rendszerről, amiben a rendszer célja, a képességei le vannak írva.

A rendszer hardverfeltételei: (minimális, ajánlott) processzor, memória, megjelenítő fajtája, szükséges hely a háttértáron, nyomtató kell-e, egér kell-e, egyéb speciális hardver kell-e.

A rendszer szoftverfeltételei: operációs rendszer fajtája, verziószáma, esetlegesen szükséges kiegészítő, együttműködő programok, mint pl. megjelenítők, szövegszerkesztők, stb…

Hálózati alkalmazás esetén, a hálózati operációs rendszer fajtáját, egyéb ismérveit.

A rendszer telepítésének módja, lehetőleg lépésről-lépésre leírva.

A rendszer indítása

A felhasználói interface általános leírása (menürendszerének, párbeszédablakok)

Az üzemszerű működéshez szükséges részek leírása – pontonként.

A karbantartási feladatok elvégzéséhez szükséges részek leírása – pontonként.

A képernyőn megjelenő listák, beviteli helyek, nyomtatási listák leírása.

Előforduló hibaüzenetek magyarázata, és azok javításának módja.

GYFK – Gyakran Feltett Kérdések. A programok működése során a felhasználók általában ugyanazokba a problémákba botlanak bele, és ugyanazokat a kérdéseket teszik fel. A kérdéseket és a rájuk adott válaszokat is célszerű befoglalni a dokumentációba

A felhasználói segítségkérés és a válasz módja.

További fejlesztési tervek, irányok.

12.3 A fejlesztést segítő egyéb rendszerek

12.3.1 Verziókezelő szoftverek

A programfejlesztésben gyakran dolgoznak együtt többen egy projekten. Fontos szempont, hogy az egyik fejlesztő által végzett munkát a másik fejlesztő ne írhassa felül, ezért megjelentek az úgynevezett verziókövető rendszerek. Ezeknek a rendszereknek a használata azon alapul, hogy a fejlesztők egy központi repository-nak nevezett adatbázisból kérik ki az éppen módosítani vagy fejleszteni kívánt forráskódot. Ekkor a verziókövető szoftver lezárja mások elől a fájlhoz való hozzáférést, azaz más nem tudja módosítani a fájlt, csak akkor, ha a fejlesztő visszatölti a repository-ba a módosított forrásfájlt. HA e közben ás is lekérte a kérdéses fájlét és a módosítást feltölti a szerverre, akkor az ne m fogja felülírni a korábbi verziót, hanem megjelenik egy verzióütközés. Ilyenkor a munkacsoport megfelelő jogosultsággal ellátott személye lekéri a fájl különböző változatait és vizuálisan is megtekintve összefésülheti a két változtatást, egy verziót létrehozva így.

A program fejlesztése során meg lehet határozni ágakat (Branch), amelyek egymástól elválhatnak. Erre akkor van szükség, amikor egy szoftver különböző verziói párhuzamosan élnek egymás mellett. Például egy kifejlesztett 1.x-es verziót elkezdik továbbfejleszteni 2.0-ra, de a munka során kiderülnek olyan kódrészletek, amelyek alapján még az érvényes 1.0 verziót is frissíteni akarják.

A fejlesztők munkájuk során lekérhetik bármelyik ág fájljait és felváltva dolgozhatnak rajta.

A verziókövető szoftverek általában szerver-kliens felépítésűek. A szerver kezeli a a reporitory-t. A kliensek változatos protokollokon keresztül (FTP://, WebDAV://, http://, https:// svn://, svn+ssl, lokális hálózat, lokális fájlrendszer, stb...) érik el és kezelis azt. A fejlesztő számítógépén pedig van egy megfelelő kliens program.

Ilyen szoftverek:

CVS – ingyenes Linux-on terjedt el, létezik Windows alatti portolása is. TortoiseCVS az egyik legjobb windows alatti kliens. (http://cvs.sourceforge.net)

Subversion (SVN) – A CVS nehézkes használatát kiküszöbölő változat. Létezik Linux és Windowsos szerver is. TortoiseSVN az egyik legjobb Windows alatti kliens. (http://tortoisesvn.sourceforge.net)

SourceSafe – A Microsoft megoldása a problémára. Nem ingyenes.

12.4 Forráskód generáló szoftverek

Ezek olyan szoftverek, amelyek képesek egy grafikus felület segítségével forráskódot generálni. Általában a felhasználói interfész alapján. A programok a felületet sablonok alapján állítják össze. A sablonok milyenségétől függ a forráskód minősége. Az így elkészített felhasználói felületek gyakran lefordíthatók, futtathatók. Az így elkészült alkalmazást **prototípusnak** hívjuk. A prototípus tehát üzleti logika és adatbázis háttér nélküli futtatható felhasználói felület.

A forráskód generáló szoftverek gyakran Objektum-orientált technológiával állítják elő a kódot.

Ilyen szoftverek:

Delphi – Pascalra alapuló Objektum orientált rendszer. (Borland cég terméke)

C-Builder (Borland Cég terméke)

Visual Basic, Visual C++, Visual C#

Eclipse egyes pluginokkal kiegészítve (Java)

NetBeans (JAVA)

CodeCharge (PHP)

És még sokan mások...

12.5 CASE eszközök szerepe a programozásban

CASE eszköz definiálása

Computer Adided system Engineering – Számítógéppel segített rendszer tervezés

Olyan eszközök, amelyek segítségével informatikai rendszerek egyes tervezési és megvalósítási lépéseit számítógéppel végezhetünk el.

CASE szoftverek csoportosítása

A CASE eszközök nem helyettesítik a megfelelő tervezést, hanem csak segítik azt. Milyen módon segítenek a CASE eszközök és milyen CASE eszközük léteznek?

Adatszótárak definiálásának lehetősége

Űrlapok definiálása és azok alapján az adatstruktúrák meghatározásának eszközei

A logikai tervezés eszközei (adatszerkezetek, logikai kapcsolatok, stbb...)

A dokumentálás eszközei (a modellekből automatikusan állít elő megfelelő módon dokumentációt)

Alkalmazási vázlat, amellyel gyorsítani lehet a fejlesztési folymatot.

Adatbeviteli és kimeneti formátumok gyors tervezése és megvalósítása (Riport writer, dialogus ablak készítő)

Tesztelési lehetőségek beépítése

Hibakereső rendszer beintegrálva

A CASE eszközök az információfejlesztési folymat majdnem minden lépésében szerepelhetnek, de nem tipikusan nem szerepelhetnek a rendszer elemző fázisokban. Azt semmiféle CASE eszköz nem tudja helyettesíteni.

A **CASE környezet jellemzői** közé tartozik a grafikus felület, látványos, célszerűen kialakított menürendszerek, ablakozó felületek, diagrammok.

CASE eszközök és 4GL nyelvek kapcsolata

A 4GL rendszerek alapvetően szorosan összefüggnek a CASE eszközökkel. A 4GL rendszerek az alapjai sokszor egy CASE eszköznek, illetve egy CASE eszköz alkalmas valamilyen 4GL rendszer nyelvi elemeit generálni.

Pl. C-Buildr, Delphi, Visual Basic, Access, Visual DBASE, CA-Visual Object, stb...

12.6 A programozó, szervező és felhasználó informális kapcsolata

Az Informatikai rendszer fejlesztése során az együttműködő partnerek szerei az alábbiak.

A **rendszerszervező**, vagy más néven szervező az, aki a rendszert mintegy felülről tudja szemlélni, annak látja összes fontos tulajdonságát, szakmai és informatikai szempontból is. Ismeri az alkalmazott módszereket, a rendszer alkotóelemeit, de nem feladata a program legelemibb részeinek ismerete, a hibalehetőségek minden részének ismerete.

A **programozó** az a személy, aki a rendszerterv utasításai alapján elkészíti az egyes programmodulokat, majd azt összeépíti magasabb szintű modulokká. Ha a rendszer fejlesztésén többen is dolgoznak programozóként, akkor kell közöttük lenni egy „vezetőprogramozónak”, akinek szerepe az, hogy vitás helyzetekben eldöntse, hogy az adott program megfelel-e a rendszertervnek, tanáccsal lássa el a modulok programozóit, segítsen az egyes modulokon dolgozó programozók munkájának összehangolásában, továbbá ellenőrizze azt, hogy az egyes modulok valóban megfelelnek-e minden szempontból a rendszerterv előírásainak, beleértve a tesztelést és a teszteredmények dokumentálását is.

A **felhasználó** az a személy, aki végső soron használja majd az információs rendszert. Ezek közül a fejlesztés során célszerű kinevezni egy un. Szakértőt, aki a megrendelő, azaz a majdani felhasználók oldaláról közreműködik a rendszer fejlesztésében. A vitás kérdések során ennek a személynek a dolga, hogy különösen a rendszerelőkészítés, majd a beüzemelés, tesztelési fázisban eldöntse, hogy egy adott megoldás vajon megfelel-e, mit kell módosítani az informatikai rendszeren és mi felel meg.

Érdekes e hármas kapcsolata. Ugyanis a hivatalos – megrendelő szállító – kapcsolaton kívül e személyek gyakran a közös munka miatt olyan viszonyba kerülnek, ami során több információ áramlik a fejlesztőhöz, mint amennyi minimálisan szükséges lenne. Ez nem baj, sőt jó dolog, ugyanis ennek nyomán használhatóbb rendszertervek és használhatóbb rendszerek készülhetnek el, mint ha csak formális kapcsolat lenne a személyek között. (Gyakran egy rendszer minőségét nem az általánosított szabályok határozzák meg, hanem a kivételkezelések egyszerű és a szokásokhoz igazodó volta)

13 Zárószó

A fenti jegyzet természetesen nem lehet teljes, hiszen az informatika és azon belül a programozás annyira szakosodott, hogy a teljes palettát egy könyvbe nem is lehetne összefoglalni. Azok számára, akik további elméleti jellegű tanulmányokat szeretnének önállóan elvégezni vagy a jegyzetben lévő egyes részek iránt érdeklődnek, az alábbi irodalmakat ajánlom. A könyvek egy részét meg lehet vásárolni a nagyobb könyvesboltokban, esetleg antikváriumokban.

Benkő	Programozzunk C nyelven	ComputerBooks, 1997
Benkő	Programozzunk Pascal nyelven	ComputerBooks, 1997
Kernighan - Richie	A C programozási nyelv	Műszaki Könyvkiadó, 1985
	Bevezetés a PHP5 programozásába (UML, OOP, PHP)	Panem Kiadó
	JAVA 2 Útikalauz programozóknak 1.3 I-II-III (JAVA, OOP,	ELTE TTK Hallgatói alapítvány
Vég Csaba	Alkalmazásfejlesztés a Unified Modelling Language jelölésrendszerével	Logos 2000 Bt. 1999
Simon Harris, James Ross	Kezdőkönyv az algoritmusokról	SZAK kiadó
Erich Gamma, Richard HHelm, Ralph Johnson, John Vlissides	Programtervezési Minták	Kiskapu - Addison Wesley

Titkosítási algoritmusok
Programozási paradigma

Absztrakt osztály
Tervezési minta (design pattern)
Prototípus
Verifikációs teszt, komponens teszt
UML (aktivitás, szekvencia, állapot, osztály diagramm, használati eset)
Szimmetrikus titkosítás, nyilvános kódú titkosítás, hash kód képzése
WSDL, SOAP. UDDI, Service broker, service provider
OOp, eseményorientált nyelv
Procedurális programozás, OOP, generikus programozás
Logikai programozás
Klasszikus (vízesés) modell
Prototípus modell
Inkrementális modell
Spirál modell

Algoritmusok, algoritmus leírási módszerek
OOP