

BABEŞ-BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR

Informatika záróvizsga tankönyv



2017

Záróvizsga témák
INFORMATIKA SZAK
(a 2017. júliusi vizsgaidőszaktól érvényes)

1. rész: Algoritmika és programozás (6 téma)

1. Keresés (szekvenciális és bináris), rendezés (kiválasztásos, buborékredezés, quicksort).
A visszalépéses keresés (backtracking).
2. OOP/objektumorientált programozás elemek a Python, C++, Java és C# programozási nyelvekben: osztályok és objektumok; egy osztály tagjai és hozzáférés-módosítók; konstruktorok és destruktorok.
3. Osztályok közötti kapcsolatok: származtatott osztályok és öröklési viszony; metódusok felülírása; polimorfizmus; dinamikus kötés; absztrakt osztályok és interfészek.
4. Osztálydiagramok és objektumok közötti kapcsolatok az UML-ben. Csomagok, osztályok és interfészek. Osztályok és interfészek közötti kapcsolatok. Objektumok. Üzenetek.
5. Lista; asszociatív tömb (map); sajátos műveletek specifikációja (megvalósítás nélkül).
6. Adatszerkezetek és adattípusok azonosítása egy adott feladat megoldása érdekében (az 5. pontban megadott témákra vonatkozóan). Meglévő könyvtárak használata a fenti adatszerkezetek esetén (Python, Java, C++, C#).

2. rész: Adatbázisok (3 téma)

1. Relációs adatbázisok; egy reláció első három normálformája.
2. Adatbázisok lekérdezése a relációs algebra operátoraival.
3. Relációs adatbázisok lekérdezése SQL segítségével (Select).

3. rész: Operációs rendszerek (3 téma)

1. Unix fájlrendszerek szerkezete.
2. Unix folyamatok: létrehozás, fork, exec, exit, wait rendszerhívások; kommunikáció pipe és FIFO állományok segítségével.
3. Unix shell programozás és alapvető Unix parancsok: cat, cp, cut, echo, expr, file, find, grep, less, ls, mkdir, mv, ps, pwd, read, rm, sort, test, wc, who.

I. Algoritmusok és programozás

1. Programozási tételek

A feladatok feladatosztályokba sorolhatók a jellegük szerint. E feladatosztályokhoz készítünk a teljes feladatosztályt megoldó algoritmososztályt. Ezeket az algoritmososztályokat *programozási tételeknek* nevezzük. Bebizonyítható, hogy a megoldások a feladat garantáltan helyes és optimális megoldásai.

A bemenet és a kimenet szerint négy csoportra oszthatók:

1. Sorozathoz érték rendelése (1 sorozat – 1 érték)
2. Sorozathoz sorozat rendelése (1 sorozat – 1 sorozat)
3. Sorozatokhoz sorozat rendelése (több sorozat – 1 sorozat)
4. Sorozatokhoz sorozatok rendelése (1 sorozat – több sorozat)

1.1. Sorozathoz érték rendelése

1.1.1. Sorozatszámítás

Adott az N elemű X sorozat. A sorozathoz hozzá kell rendelnünk egyetlen értéket (S). Ezt az értéket egy, az egész sorozaton értelmezett függvény (f) (pl. elemek összege, szorzata stb.) adja.

Ezt a függvényt felbonthatjuk értékpárokon kiszámított függvények sorozatára, így a megoldás a semleges elemre (F_0), valamint egy kétoperandusú műveletre épül. Az S kezdőértéke a semleges elem. A kétoperandusú műveletet végrehajtjuk minden elemre (X_i) és az értékre (S): $S \leftarrow f(X_i, S)$.

Összeg és szorzat

Egyetlen kimeneti adatot számítunk ki adott számú bemeneti adat feldolgozásának eredményeként. *Példa:* a bemeneti adatok összegét, illetve szorzatát kell kiszámítanunk.

Megoldás

A feladat megoldása előtt szükséges tudni, hogy mely érték felel meg a bemeneti adatok halmazára és az elvégzendő műveletre nézve a semleges elemnek. Feltételezzük, hogy a bemeneti adatok egész számok, amelyeknek a számossága N .

Algoritmus Összegszámítás(N, X, S): { Sajátos eset!!! }
{ Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: S }

$S \leftarrow 0$

Minden $i = 1, N$ **végezd el:**

{ minden adatot fel kell dolgoznunk }

$S \leftarrow S + X_i$

vége(minden)

Vége(algoritmus)

Algoritmus Feldolgoz(N, X, S): { Általános eset!!! }
{ Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: S }

$S \leftarrow F_0$

{ kezdőérték: az elvégzendő műveletre nézve semleges elem }

Minden $i = 1, N$ **végezd el:**

{ minden adatot fel kell dolgoznunk }

$S \leftarrow f(S, X_i)$

{ f a művelet = funkció }

vége(minden)

Vége(algoritmus)

Adott az N elemű X sorozat és az elemein értelmezett T tulajdonság. Adjuk meg a sorozat egy T tulajdonságú elemének sorszámát! (Előfeltétel: már tudjuk, hogy garantáltan létezik ilyen elem.)

Algoritmus Kiválasztás(N , X , hely):

{ Bemeneti adatok: az N elemű X sorozat. }

{ Kimeneti adat: a legkisebb indexű T tulajdonságú elem sorszáma: hely }

hely $\leftarrow 1$

Amíg nem $T(X_{\text{hely}})$ **végezd el:** { nem szükséges a hely $\leq N$ feltétel mivel a feladat }

hely $\leftarrow$ hely + 1 { garانتálja legalább egy T tulajdonságú elem létezését }

vége(amíg)

Vége(algoritmus)

1.1.4. Szekvenciális (lineáris) keresés

Adott az N elemű X sorozat és az elemein értelmezett T tulajdonság. Vizsgáljuk meg, hogy van-e T tulajdonságú elem a sorozatban! Ha van, akkor adjunk meg egyet!

Algoritmus Keres_1(N , X , hely):

{ Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: hely, a legkisebb indexű T }

{ tulajdonságú elem indexe, illetve, sikertelen keresés esetén hely = 0 }

hely $\leftarrow 0$

Amíg (hely = 0) és ($i \leq N$) **végezd el:**

Ha $T(X_i)$ **akkor**

hely $\leftarrow i$

különben

$i \leftarrow i + 1$

vége(ha)

vége(amíg)

Vége(algoritmus)

Az adott elem tulajdonságát az *Amíg* feltételében is ellenőrizhetjük. Más szóval: amíg az aktuális elem tulajdonsága nem megfelelő, haladunk a sorozatban előre.

Algoritmus Keres_2(N , X , hely):

{ Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: hely, a legkisebb indexű T }

{ tulajdonságú elem indexe, illetve, sikertelen keresés esetén hely = 0 }

$i \leftarrow 1$

Amíg ($i \leq N$) és nem $T(X_i)$ **végezd el:**

$i \leftarrow i + 1$

vége(amíg)

Ha $i \leq N$ **akkor** { ha kiléptünk az Amíg-ból, mielőtt i nagyobbá vált volna N -nél, }

hely $\leftarrow i$ { $\Rightarrow$ találtunk adott tulajdonságú elemet, különben nem }

különben

hely $\leftarrow 0$

vége(ha)

Vége(algoritmus)

Ha a követelményben az áll, hogy minden olyan elemet keressünk meg, amely rendelkezik az adott tulajdonsággal: be kell járnunk a teljes adathalmazt, és vagy kiírjuk azonnal a pozíciókat, ahol megfelelő elemet találtunk, vagy megőrizzük ezeket egy sorozatban. Ilyenkor egy *Minden* típusú struktúrát használunk.

1.1.5. Megszámlálás

Adott, N elemű X sorozatban számoljuk meg a T tulajdonságú elemeket!

Elemzés

Nem biztos, hogy létezik legalább egy T tulajdonságú elem, tehát az is lehetséges, hogy az eredmény 0 lesz. Mivel minden elemet meg kell vizsgálnunk (bármely adat rendelkezhet a kért tulajdonsággal), *Minden* típusú struktúrával dolgozunk. A darabszámot a db változóban számoljuk.

Algoritmus Megszámlálás(N, X, db):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: db , a T tulajdonságú elemek száma* }

$db \leftarrow 0$

Minden $i = 1, N$ **végezd el:**

Ha $T(X_i)$ **akkor**

$db \leftarrow db + 1$

vége(ha)

vége(minden)

Vége(algoritmus)

1.1.6. Maximumkiválasztás

Adott az N elemű X sorozat. Határozzuk meg a sorozat legnagyobb (vagy legkisebb) értékét (vagy sorszámát)!

Megoldás

A megoldásban minden adatot meg kell vizsgálnunk, ezért az algoritmus, egy *Minden* típusú struktúrával dolgozik. A max segédváltozó kezdőértéke a sorozat első eleme.

Algoritmus Maximumkiválasztás(N, X, max):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: max , a legnagyobb elem értéke* }

$max \leftarrow X_1$

Minden $i = 2, n$ **végezd el:**

Ha $max < X_i$ **akkor**

$max \leftarrow X_i$

vége(ha)

vége(minden)

Vége(algoritmus)

A maximumot/minimumot tartalmazó segédváltozónak az adatok közül választunk kezdőértéket, mivel így nem áll fenn a veszély, hogy az algoritmus eredménye egy, az adataink között nem létező érték legyen.

Ha a maximum helyét kell megadnunk:

Algoritmus Maximum_helye($N, X, hely$):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: $hely$, a legnagyobb elem pozíciója* }

$hely \leftarrow 1$ { *hely az első elem pozíciója* }

Minden $i = 2, n$ **végezd el:**

Ha $X_{hely} < X_i$ **akkor**

$hely \leftarrow i$

{ *a maximális elem helye (pozíciója)* }

vége(ha)

vége(minden)

Vége(algoritmus)

Ha minden olyan indexet meg kell határoznunk, amely indexű elemek egyenlők a legnagyobb elemmel és nem lehetséges/nem előnyös az adott tömböt kétszer bejárni, mert a maximumhoz tartozó adatok egy másik (esetleg bonyolult) algoritmus végrehajtásának eredményei, írhatunk olyan algoritmust, amely csak egyszer járja be a sorozatot:

Algoritmus Minden_max_2($N, X, db, indexek$):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű indexek sorozat* }

```

max  $\leftarrow X_1$ 
db  $\leftarrow 1$ 
indexek1  $\leftarrow 1$ 
Minden  $i = 2, n$  végezd el:
  Ha max  $< X_i$  akkor
    max  $\leftarrow X_i$ 
    db  $\leftarrow 1$ 
    indexekdb  $\leftarrow i$ 
  különben
    Ha max =  $X_i$  akkor
      db  $\leftarrow db + 1$ 
      indexekdb  $\leftarrow i$ 
  vége(ha)
vége(ha)
vége(minden)
Vége(algoritmus)

```

1.2. Sorozathoz sorozat rendelése

1.2.1. Másolás

Adott az N elemű X sorozat és az elemein értelmezett függvény (f). A bemenő sorozat minden elemére végrehajtjuk a függvényt, az eredményét pedig a kimenő sorozatba másoljuk.

Algoritmus Másolás(N, X, Y):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű Y sorozat* }

```

Minden  $i = 1, N$  végezd el:
   $Y_i \leftarrow f(X_i)$ 
vége(minden)
Vége(algoritmus)

```

1.2.2. Kiválogatás

Adott az N elemű X sorozat és az elemein értelmezett T tulajdonság. Válogassuk ki az összes T tulajdonságú elemet!

Elemzés

Az elvárások függvényében több megközelítés érvényes:

- | | | |
|----|-------------------------|---|
| a) | | k |
| | iválogatás kigyűjtéssel | |
| b) | | k |
| | iválogatás kiírással | |
| c) | | h |
| | elyben | |
| d) | | k |
| | ihúzással | |

a) Kiválogatás kigyűjtéssel

A keresett elemeket (vagy sorszámaikat) kigyűjtjük egy sorozatba. A pozíciók sorozatának (vagy a kigyűjtött elemek sorozatának) hossza legfeljebb az adott sorozatéval lesz megegyező, mivel előfordulhat, hogy a bemeneti sorozat minden eleme adott tulajdonságú. A sorozat számosságát a db változóban tartjuk nyilván.

Algoritmus Kiválogatás_a($N, X, db, \text{pozíciók}$):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű pozíciók sorozat* }

```

db ← 0
Minden i = 1, N végezd el:
  Ha  $T(X_i)$  akkor
    db ← db + 1
    pozíciókdb ← i      { pozíciókdb-ben tároljuk  $X_i$  helyét, ha  $T$  tulajdonsága van }
  vége(ha)
vége(minden)
Vége(algoritmus)

```

b) Kiválogatás kiírással

Ha nincs szükség számolásra, és a feladat „megelégszik” a T tulajdonságú elemek kiírásával:

Algoritmus Kiválogatás_b(N , X , db):

{ *Bemeneti adatok: az N elemű X sorozat* }

```

db ← 0
Minden i = 1, N végezd el:
  Ha  $T(X_i)$  akkor
    Ki:  $X_i$ 
  vége(ha)
vége(minden)
Vége(algoritmus)

```

c) Kiválogatás helyben

Ha a sorozat feldolgozása közben a nem T tulajdonságú elemeket nem óhajtjuk megőrizni, hanem ki szeretnénk zárni ezeket a sorozatból, akkor a feladat specifikációitól függően, a következő lehetőségek közül fogunk választani:

c1. Ha a törlés után nem kötelező, hogy az elemek az eredeti sorrendben maradjanak, akkor a törlendő elemre rámásoljuk a sorozat utolsó elemét és csökkentjük 1-gyel a sorozat méretét:

Algoritmus Kiválogatás_c1(N , X):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a megváltozott elemszámú X sorozat* }

```

i ← 1
Amíg i ≤ N végezd el:      { nem lehet Minden-t alkalmazni, mert változik az  $N$ !!! }
  Ha nem  $T(X_i)$  akkor      { a  $T$  tulajdonságú elemeket tartjuk meg }
     $X_i$  ←  $X_N$               {  $X_i$ -t felülírjuk  $X_N$ -nel }
     $N$  ←  $N - 1$               { rövidül a sorozat }
  különben
    i ← i + 1                { i csak a különben ágon nő }
  vége(ha)
vége(amíg)
Vége(algoritmus)

```

c2. Kiválogatás segédsorozattal

Ha a törlés ideiglenes, akkor a kereséssel párhuzamosan egy logikai tömbben nyilvántartjuk a „törölt” elemeket. A *törölt* tömb elemeinek kezdőértéke hamis lesz, majd a törlendő elemeknek megfelelő sorszámú elemek értéke a *törölt* logikai tömbben igaz lesz:

Algoritmus Kiválogatás_c2(N , X , *törölt*):

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű törölt sorozat* }

```

Minden  $i = 1, N$  végezd el:
    törölt $i$   $\leftarrow$  hamis
vége(minden)
Minden  $i = 1, N$  végezd el:
    Ha nem  $T(X_i)$  akkor                                {  $a$   $T$  tulajdonságú elemeket tartjuk meg }
        törölt $i$   $\leftarrow$  igaz
    vége(ha)
vége(minden)
Vége(algoritmus)

```

c3. Kiválogatás helyben, megőrizve az eredeti sorrendet

Ha az eredeti sorozatra nincs többé szükség, de szeretnénk megőrizni az elemek eredeti sorrendjét, akkor a T tulajdonságú elemeket felsorakoztatjuk a sorozat elejétől elkezdve. Így a kiválogatott elemekkel felülírjuk az eredeti adatokat. Nem használunk egy újabb sorozatot, hanem az adott sorozat számára lefoglalt tárrészt használva helyben végezzük a kiválogatást. A db változó ebben az esetben ennek az „új” sorozatnak a számosságát tartja nyilván:

```

Algoritmus Kiválogatás_c3( $N, X, db$ ):
    { Bemeneti adatok: az  $N$  elemű  $X$  sorozat. Kimeneti adat: a  $db$  elemű  $X$  sorozat }
     $db \leftarrow 0$ 
    Minden  $i = 1, N$  végezd el:
        Ha  $T(X_i)$  akkor
             $db \leftarrow db + 1$ 
             $X_{db} \leftarrow X_i$ 
        vége(ha)
    vége(minden)
Vége(algoritmus)

```

d) Kiválogatás speciális értékkel

Egy másik megoldás, amely nem hoz létre új helyen, új sorozatot, hanem helyben végzi a kiválogatást, anélkül, hogy elmozdítaná eredeti helyükről a T tulajdonságú elemeket, a nem T tulajdonságú elemek helyére pedig egy speciális értéket tesz:

```

Algoritmus Kiválogatás_d( $N, X, törölt$ ):
    { Bemeneti adatok: az  $N$  elemű  $X$  sorozat. Kimeneti adat: az  $N$  elemű  $X$  sorozat }
    Minden  $i = 1, N$  végezd el:
        Ha nem  $T(X_i)$  akkor                                {  $a$   $T$  tulajdonságú elemeket tartjuk meg }
             $X_i \leftarrow$  speciális érték
        vége(ha)
    vége(minden)
Vége(algoritmus)

```

1.3. Sorozatokhoz sorozat rendelése

1.3.1. Halmazok

Mielőtt egy – halmazokat tartalmazó sorozatra vonatkozó műveletet alkalmaznánk, szükséges meggyőződnünk afelől, hogy a sorozat valóban halmaz. Ez azt jelenti, hogy minden érték csak egyszer fordul elő. Ha kiderül, hogy a sorozat nem halmaz, halmazzá kell alakítanunk.

a) Halmaz-e?

Döntsük el, hogy az adott N elemű X sorozat halmaz-e!

Elemzés

Egy halmaz vagy üres, vagy bizonyos számú elemet tartalmaz. Ha egy halmazt sorozattal implementálunk, az elemei különbözők. A következő algoritmussal eldöntjük, hogy a sorozat csak különböző elemeket tartalmaz-e? A döntés eredményét az *ok* kimeneti paraméter tartalmazza.

Algoritmus $\text{Halmaz_e}(N, X, \text{ok})$:

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adatok: az ok értéke igaz, ha a sorozat }
 $i \leftarrow 1$ { *halmaz, különben hamis* }
 $\text{ok} \leftarrow \text{igaz}$
Amíg ok **és** $(i < N)$ **végezd el:**
 $j \leftarrow i + 1$
Amíg $(j \leq N)$ **és** $(X_i \neq X_j)$ **végezd el:**
 $j \leftarrow j + 1$
vége(amíg)
 $\text{ok} \leftarrow j > N$ { *ha véget ért a sorozat, nincs két azonos elem* }
 $i \leftarrow i + 1$
vége(amíg)
Vége(algoritmus)*

b) Halmazzá alakítás

Alakítsuk halmazzá az N elemű X sorozatot!

Elemzés

Ha egy alkalmazásban ki kell zárunk az adott sorozatból a másodszor (harmadszor stb.) megjelenő értékeket, akkor az előbbi algoritmust módosítjuk: amikor egy bizonyos érték megjelenik másodszor, felülírjuk az utolsóval.

Algoritmus $\text{Halmaz_2}(N, X)$:

{ *Bemeneti adatok: az N elemű X sorozat. Kimeneti adatok: az új N elemű X sorozat (halmaz) }
 $i \leftarrow 1$
Amíg $i < N$ **végezd el:**
 $j \leftarrow i + 1$
Amíg $(j \leq N)$ **és** $(X_i \neq X_j)$ **végezd el:**
 $j \leftarrow j + 1$
vége(amíg)
Ha $j \leq N$ **akkor** { *találtunk egy $X_j = X_i$ -t* }
 $X_j \leftarrow X_N$ { *felülírjuk a sorozat N . elemével* }
 $N \leftarrow N - 1$ { *rövidítjük a sorozatot* }
különben
 $i \leftarrow i + 1$ { *haladunk tovább* }
vége(ha)
vége(amíg)
Vége(algoritmus)*

1.3.2. Keresztmetszet

Hozzuk létre azt a sorozatot, amely a bemenetként kapott sorozatok keresztmetszetét tartalmazza.

Elemzés

Keresztmetszet alatt azt a sorozatot értjük, amely az adott sorozatok közös elemeit tartalmazza. Feltételezzük, hogy az adott sorozatok mind különböző elemeket tartalmaznak (halmazok) és nem rendezett sorozatok.

Az N elemű X és az M elemű Y sorozat keresztmetszetét a db elemű Z sorozatban hozzuk létre, tehát Z olyan elemeket tartalmaz az X sorozatból, amelyek megtalálhatók az Y -ban is.

Algoritmus Keresztmetszet(N, X, M, Y, db, Z):

```

    { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. }
    db  $\leftarrow$  0 { Kimeneti adatok: a db elemű Z sorozat, X és Y keresztmetszete }
    Minden  $i = 1, N$  végezd el:
         $j \leftarrow 1$ 
        Amíg ( $j \leq M$ ) és ( $X_i \neq Y_j$ ) végezd el:
             $j \leftarrow j + 1$ 
        vége(amíg)
        Ha  $j \leq M$  akkor
             $db \leftarrow db + 1$ 
             $Z_{db} \leftarrow X_i$ 
        vége(ha)
    vége(minden)
Vége(algoritmus)
```

Áprili.

1.3.3. Egyesítés (Unió)

Hozzuk létre az N elemű X és az M elemű Y sorozatok (halmazok) egyesített halmazát!

Elemzés

Az egyesítés algoritmus a hasonló a keresztmetszethez. Nem alkalmazhatunk összefésülést, mivel a sorozatok nem rendezettek! A különbség abban áll, hogy olyan elemeket helyezünk az eredménybe, amelyek legalább az egyik sorozatban megtalálhatók.

Előbb a Z sorozatba másoljuk az X sorozatot, majd kiválogatjuk Y -ból azokat az elemeket, amelyeket nem találtunk meg X -ben.

Algoritmus Egyesítés(X, Y, Z, M, N, db):

```

     $Z \leftarrow X$  { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. }
    db  $\leftarrow$  N { Kimeneti adatok: a db elemű Z sorozat (X és Y egyesítése) }
    Minden  $j = 1, M$  végezd el:
         $i \leftarrow 1$ 
        Amíg ( $i \leq N$ ) és ( $X_i \neq Y_j$ ) végezd el:
             $i \leftarrow i + 1$ 
        vége(amíg)
        Ha  $i > N$  akkor
             $db \leftarrow db + 1$ 
             $Z_{db} \leftarrow Y_j$ 
        vége(ha)
    vége(minden)
Vége(algoritmus)
```

1.3.4. Összefésülés

Adott két rendezett sorozatból állítsunk elő egy harmadikat, amely legyen szintén rendezett!

Elemzés

Az *Egyesítés* és a *Keresztmetszet* algoritmusok négyzetes bonyolultságúak, mivel a halmazokat implementáló sorozatok nem rendezettek. Ez a két művelet megvalósítható lineáris algoritmussal, ha a sorozatok rendezettek. Természetesen az eredményt is rendezett formában fogjuk generálni.

Ezek a sorozatok nem mindig halmazok, tehát néha előfordulhatnak azonos értékű elemek is. Elindulunk mindkét sorozatban és a soron következő két elem összehasonlítása révén

eldöntjük, melyiket tegyük a harmadikba. Addig végezzük ezeket a műveleteket, amíg valamelyik sorozatnak a végére érünk. A másik sorozatban megmaradt elemeket átmásoljuk az eredményssorozatba. Mivel nem tudhatjuk előre melyik sorozat ért véget, vizsgáljuk mindkét sorozatot.

Algoritmus Összefésülés_1(N, X, M, Y, db, Z):

{ *Bemeneti adatok: az N elemű X és az M elemű Y sorozat.* }

{ *Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel)* }

{ *A sorozatok nem halmazok* }

$db \leftarrow 0$

$i \leftarrow 1$

$j \leftarrow 1$

Amíg ($i \leq N$) **és** ($j \leq M$) **végezd el:**

$db \leftarrow db + 1$

Ha $X_i < Y_j$ **akkor**

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

különben

$Z_{db} \leftarrow Y_j$

$j \leftarrow j + 1$

vége(ha)

vége(amíg)

Amíg $i \leq N$ **végezd el:**

{ *ha maradt még elem X -ben* }

$db \leftarrow db + 1$

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

vége(amíg)

Amíg $j \leq M$ **végezd el:**

{ *ha maradt még elem Y -ban* }

$db \leftarrow db + 1$

$Z_{db} \leftarrow Y_j$

$j \leftarrow j + 1$

vége(amíg)

Vége(algoritmus)

Most feltételezzük, hogy az egyes sorozatokban egy elem csak egyszer fordul elő és azt szeretnénk, hogy az összefésült új sorozatban se legyenek „duplák”. Az előző algoritmust csak annyiban módosítjuk, hogy vizsgáljuk az egyenlőséget is. Ha a két összehasonlított érték egyenlő, mind a két sorozatban továbblépünk és az aktuális értéket csak egyszer írjuk be az eredményssorozatba.

Algoritmus Összefésülés_2(N, X, M, Y, db, Z):

$db \leftarrow 0$

{ *Bemeneti adatok: az N elemű X és az M elemű Y sorozat.* }

$i \leftarrow 1$

{ *Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel)* }

$j \leftarrow 1$

{ *a sorozatok halmazok* }

Amíg ($i \leq N$) **és** ($j \leq M$) **végezd el:**

$db \leftarrow db + 1$

Ha $X_i < Y_j$ **akkor**

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

különben

```

Ha  $X_i = Y_j$  akkor
     $Z_{db} \leftarrow X_i$ 
     $i \leftarrow i + 1$ 
     $j \leftarrow j + 1$ 
különben
     $Z_{db} \leftarrow Y_j$ 
     $j \leftarrow j + 1$ 
vége(ha)
vége(ha)
vége(amíg)
Amíg  $i \leq N$  végezd el: { ha maradt még elem X-ben }
     $db \leftarrow db + 1$ 
     $Z_{db} \leftarrow X_i$ 
     $i \leftarrow i + 1$ 
vége(amíg)
Amíg  $j \leq m$  végezd el: { ha maradt még elem Y-ban }
     $db \leftarrow db + 1$ 
     $Z_{db} \leftarrow Y_j$ 
     $j \leftarrow j + 1$ 
vége(amíg)
Vége(algoritmus)

```

Ha szerencsések lettünk volna $X_N = Y_M$. Ekkor a két utolsó *Amíg* struktúrát nem hajtotta volna végre a program egyetlen egyszer sem. Kihaszználjuk ezt az észrevételt: elhelyezünk mindkét sorozat végére egy fiktív elemet (őrszem). Tehetjük az X sorozat végére az $X_{N+1} = Y_M + 1$ értéket és az Y sorozat végére az $Y_{M+1} = X_N + 1$ értéket. Ha a két egyesítendő sorozat nem halmaz, az eredmény sem lesz halmaz. Észrevesszük, hogy ebben az esetben az eredményssorozat hossza pontosan $N + M$. Az algoritmus ismétlődő struktúrája *Minden* típusú lesz.

Algoritmus *Összefésül_3*(N, X, M, Y, db, Z):

```

 $i \leftarrow 1$  { Bemeneti adatok: az  $N$  elemű  $X$  és az  $M$  elemű  $Y$  sorozat. }
 $j \leftarrow 1$  { Kimeneti adatok: a  $db$  elemű  $Z$  sorozat ( $X$  és  $Y$  elemeivel) }
 $X_{N+1} \leftarrow Y_M + 1$  { A sorozatok nem halmazok }
 $Y_{M+1} \leftarrow X_N + 1$ 

```

```

Minden  $db = 1, N + M$  végezd el:
    Ha  $X_i < Y_j$  akkor
         $Z_{db} \leftarrow X_i$ 
         $i \leftarrow i + 1$ 
    különben
         $Z_{db} \leftarrow Y_j$ 
         $j \leftarrow j + 1$ 
    vége(ha)
vége(minden)
Vége(algoritmus)

```

Ha a bemeneti sorozatok halmazokat ábrázolnak és az eredményssorozatnak is halmaznak kell lennie, az algoritmus a következőképpen alakul: *Minden* struktúra helyett *Amíg*-ot alkalmazunk, hiszen nem tudjuk hány eleme lesz az összefésült sorozatnak (az ismétlődő

értékek közül csak egy kerül be az új sorozatba). Ugyanakkor, az örszemek révén az Amíg struktúrát addig hajtjuk végre, amíg mindkét sorozat végére nem értünk.

Algoritmus Összefésül_4(N, X, M, Y, db, Z):

$db \leftarrow 0$ { *Bemeneti adatok: az N elemű X és az M elemű Y sorozat.* }

$i \leftarrow 1$ { *Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel)* }

$j \leftarrow 1$ { *A sorozatok halmazok* }

$X_{N+1} \leftarrow Y_M + 1$

$Y_{M+1} \leftarrow X_N + 1$

Amíg ($i < N + 1$) **vagy** ($j < M + 1$) **végezd el:** { *figyelem! itt vagy (nem és)* }

$db \leftarrow db + 1$

Ha $X_i < Y_j$ **akkor**

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

különben

Ha $X_i = Y_j$ **akkor**

$Z_{db} \leftarrow X_i$

$i \leftarrow i + 1$

$j \leftarrow j + 1$

különben

$Z_{db} \leftarrow Y_j$

$j \leftarrow j + 1$

vége(ha)

vége(ha)

vége(amíg)

Vége(algoritmus)

1.4. Sorozathoz sorozatok rendelése

1.4.1. Szétválogatás

Adott N elemű X sorozatot válogassuk szét adott T tulajdonság alapján!

Elemzés

A *Kiválogatás*(N, X) algoritmus egy sorozatot dolgoz fel, amelyből kiválogat bizonyos elemeket.

Kérdés: mi történik azokkal az elemekkel, amelyeket nem válogattunk ki? Lesznek feladatok, amelyek azt kérik, hogy két vagy több sorozatba válogassuk szét az adott sorozatot.

a) szétválogatás két sorozatba

Az adott sorozatból létrehozunk két újat: a tulajdonsággal rendelkező adatok sorozatát, és a megmaradtak sorozatát. Mindkét új sorozatot az eredetivel azonos méretűnek deklaráljuk, mivel nem tudhatjuk előre az új sorozatok valós méretét. (Előfordulhat, hogy valamennyi elem átvándorol valamelyik sorozatba, és a másik üres marad.) A dby és dbz a szétválogatás során létrehozott Y és Z sorozatba helyezett elemek számát jelöli.

Algoritmus Szétválogatás_1(N, X, dby, Y, dbz, Z):

$dby \leftarrow 0$ { *Bemeneti adatok: az N elemű X sorozat.* }

$dbz \leftarrow 0$ { *Kimeneti adat: a dby elemű Y és a dbz elemű Z sorozat* }

Minden $i = 1, N$ **végezd el:**

Ha $T(X_i)$ **akkor**

$dby \leftarrow dby + 1$ { *az adott tulajdonságú elemek, az Y sorozatba kerülnek* }

```

 $Y_{dbz} \leftarrow X_i$ 
különb
 $dbz \leftarrow dbz + 1$ 
 $Z_{dbz} \leftarrow X_i$ 
vége(ha)
vége(minden)
Vége(algoritmus)

```

{ azok, amelyek nem rendelkeznek az }
{ adott tulajdonsággal, a Z sorozatba kerülnek }

b) szétválogatás egy új sorozatba

A feladat megoldható egyetlen új sorozattal. A kiválogatott elemeket az új sorozat első részébe helyezzük (az elsőtől haladva a vége felé), a megmaradtakat az új sorozat végére (az utolsótól haladva az első felé). Nem fogunk ütközni, mivel pontosan N elemet kell N helyre „átrendezni”. A megmaradt elemek az eredeti sorozatban elfoglalt relatív pozícióik fordított sorrendjében kerülnek az új sorozatba.

Algoritmus Szétválogatás_2(N , dbz , X , Y):

```

 $dbz \leftarrow 0$  { Bemeneti adatok: az  $N$  elemű  $X$  sorozat. Kimeneti adat: az  $N$  elemű  $Y$  sorozat, }
 $dbz \leftarrow 0$  { ahol az első  $dbz$  elem  $T$  tulajdonságú,  $dbz$  elem pedig nem  $T$  tulajdonságú }
Minden  $i = 1, N$  végezd el:
    Ha  $T(X_i)$  akkor { a  $T$  tulajdonságú elemek az  $Y$  sorozatba kerülnek }
         $dbz \leftarrow dbz + 1$  { az első helytől kezdődően }
         $Y_{dbz} \leftarrow X_i$ 
    különben
         $dbz \leftarrow dbz + 1$  { a többi elem ugyancsak az  $Y$ -ba kerül, az utolsó helytől kezdődően }
         $Y_{N-dbz+1} \leftarrow X_i$ 
    vége(ha)
vége(minden)
Vége(algoritmus)

```

c) Szétválogatás helyben

Ha a szétválogatás után nincs már szükségünk többé az eredeti sorozatra, a szétválogatás elvégezhető helyben. A tömb első elemét kivesszük a helyéről és megőrizzük egy segédváltozóban. Az utolsó elemtől visszafelé megkeressük az első olyat, amely adott tulajdonságú, s ezt előre hozzuk a kivett elem helyére. Ezután a hátul felszabadult helyre előről keresünk egy nem T tulajdonságú elemet, s ha találunk, azt hátrátesszük. Mindezt addig végezzük amíg a tömbben két irányban haladva össze nem találkozunk.

Algoritmus Szétválogatás_3(N , X , db):

```

{ Bemeneti adatok: az  $N$  elemű  $X$  sorozat. Kimeneti adatok: az  $N$  elemű  $X$  sorozat, ahol }
{ az első  $e$  elem  $T$  tulajdonságú,  $n - e$  elem pedig nem  $T$  tulajdonságú }
 $e \leftarrow 1$  { balról jobbra haladva az első  $T$  tulajdonságú elem indexe }
 $u \leftarrow N$  { jobbról balra haladva az első nem  $T$  tulajdonságú elem indexe }
segéd  $\leftarrow X_e$ 
Amíg  $e < u$  végezd el:
    Amíg  $(e < u)$  és nem  $T(X_u)$  végezd el:
         $u \leftarrow u - 1$ 
    vége(amíg)
    Ha  $e < u$  akkor
         $X_e \leftarrow X_u$ 
         $e \leftarrow e + 1$ 
    Amíg  $(e < u)$  és  $T(X_e)$  végezd el:

```

```

    e ← e + 1
vége(amíg)
Ha e < u akkor
    Xu ← Xe
    u ← u - 1
vége(ha)
vége(ha)
vége(amíg)
Xe ← segéd { visszahozzuk a segéd-be tett elemet }
Ha T(Xe) akkor
    db ← e
különben
    db ← e - 1
vége(ha)
Vége(algoritmus)

```

Megjegyzés

Ha egy sorozatot több részsorozatba szükséges szétválogatni több tulajdonság alapján, egymás után több szétválogatást fogunk végezni, mindig a kért tulajdonság alapján. Előbb szétválogatjuk az adott sorozatból az első tulajdonsággal rendelkezőket, majd a félretett adatokból szétválogatjuk a második tulajdonsággal rendelkezőket és így tovább.

Programozási tételek összeépítése

Az egészen egyszerű alapfeladatokat kivéve általában több programozási tételt kell használnunk.

Ilyenkor – ahelyett, hogy simán egymás után alkalmazzuk ezeket, lehetséges egyszerűbb, rövidebb, hatékonyabb, gazdaságosabb algoritmust tervezni, ha összeépítjük őket.

2. Lépések finomítása

Bonyolultabb feladatok esetében a megfelelő algoritmus leírása nem könnyű feladat. Ezért célszerű először a megoldást körvonalazni, és csak azután részletezni. A feladat elemzése során sor kerül a bemeneti és kimeneti adatok megállapítására, a megfelelő adatszerkezetek kiválasztására és megtervezésére, a feladat követelményeinek szétválasztására. Következik a megoldási módszer megállapítása, a megoldás lépéseinek leírása és a *lépések finomítása*, amíg az algoritmus hatékonysága megfelelő lesz. Végül elmaradhatatlanul következnie kell a helyesség ellenőrzésének, és a programkészítés (kódolás) után a tesztelés.

A lépések finomítása az algoritmus *kidolgozását* jelenti, amely a kezdeti vázlattól a végleges, precízen leírt algoritmusig vezet. Kiindulunk a feladat specifikációjából és fentről lefele tartó tervezési módszert alkalmazva újabb meg újabb változatokat dolgozunk ki, amelyek eleinte még tartalmaznak bizonyos, anyanyelven leírt magyarázó sorokat, amelyeket csak később írunk át standard utasításokkal. Így, az algoritmusnak több egymás utáni változata lesz, amelyek egyre bővülnek egyik változattól a másikig.

1. Eukleidész algoritmusa

Határozzuk meg két adott természetes szám legnagyobb közös osztóját (*lnko*) és legkisebb közös többszörösét (*lkkt*) Eukleidész algoritmusával.

Algoritmus Eukleidész_1(*a*, *b*, *lnko*, *lkkt*):

@ kiszámítjuk *a* és *b* *lnko*-ját

{ Bemeneti adatok: *a*, *b*. Kimeneti adatok: *lnko*, *lkkt* }

@ kiszámítjuk *a* és *b* *lkkt*-ét

Vége(algoritmus)

Lépések finomítása: Ki kell dolgoznunk a kiszámítások módját. Ha a két szám egyenlő, akkor *lnko* az *a* szám lesz. Ha *a* kisebb, mint *b*, nincs szükség felcserélésre: az algoritmus elvégzi ezt az első lépésében. Ezután kiszámítjuk *r*-ben *a* egész osztási maradékát *b*-vel. Ha a maradék nem 0, a következő lépésben *a*-t felülírjuk *b*-vel, *b*-t *r*-rel, és újból kiszámítjuk a maradékot. Addig dolgozunk, amíg a maradék 0-vá nem válik. Az utolsó osztó éppen az *lnko* lesz. Az *lkkt*-t úgy kapjuk meg, hogy a két szám szorzatát osztjuk az *lnko*-val. Mivel az eredeti két szám értékét az algoritmus „tönkreteszi”, szükséges elmenteni ezeket két segédváltozóba (*x* és *y*) ahhoz, hogy felhasználhassuk ezeket az *lkkt* kiszámításakor.

Algoritmus Eukleidész_1(*a*, *b*, *lnko*, *lkkt*):

{ Bemeneti adatok: *a*, *b*. Kimeneti adatok: *lnko*, *lkkt* }

x ← *a*

{ szükségünk lesz *a* értékére az *lkkt* kiszámításakor }

y ← *b*

{ szükségünk lesz *b* értékére az *lkkt* kiszámításakor }

r ← maradék[*a*/*b*]

{ kiszámítjuk az első maradékot }

Amíg *r* ≠ 0 **végezd el:**

{ amíg *a* maradék nem 0 }

a ← *b*

{ az osztandót felülírjuk az osztóval }

b ← *r*

{ az osztót felülírjuk a maradékkal }

r ← maradék[*a*/*b*]

{ kiszámítjuk az aktuális maradékot }

vége(amíg)

lnko ← *b*

{ *lnko* egyenlő az utolsó osztó értékével }

vége(ha)

lkkt ← [*x***y*/*lnko*]

{ felhasználjuk *a* és *b* másolatait }

Vége(algoritmus)

Megvalósíthatjuk az algoritmust ismételt kivonásokkal. Amíg a két szám különbözik egymástól, a nagyobbikból kivonjuk a kisebbiket, és megőrizzük a különbséget. Az *lnko* az utolsó különbség lesz. Az *lkkt*-t ugyanúgy számítjuk ki, mint az előző változatban.

Algoritmus Eukleidész_2(*a*, *b*, *lnko*, *lkkt*):

$x \leftarrow a$ { *Bemeneti adatok: a, b. Kimeneti adatok: lnko, lkkt* }

$y \leftarrow b$

Amíg $a \neq b$ **végezd el:**

Ha $a > b$ **akkor**

$a \leftarrow a - b$

különb

$b \leftarrow b - a$

vége(ha)

vége(amíg)

$lnko \leftarrow a$

$lkkt \leftarrow [x*y/lnko]$

Vége(algoritmus)

2. Prímszámok

Adva van egy nullától különböző természetes szám (*n*). Döntsük el, hogy az adott szám prímszám-e vagy sem!

Algoritmus Prím(*n*, *válasz*):

{ *Bemeneti adat: n. Kimeneti adat: válasz* }

@ Megállapítjuk, hogy *n* prím-e

Ha *n* prím **akkor**

válasz $\leftarrow$ igaz

különb

válasz $\leftarrow$ hamis

vége(ha)

Vége(algoritmus)

Lépések finomítása: Ki kell dolgoznunk annak a módját, hogy megállapíthassuk, hogy a szám prím-e. A megoldás első változatában a prímszám definíciójából indulunk ki: egy szám akkor prím, ha pontosan két osztója van: 1 és maga a szám. Első ötletünk tehát az, hogy az algoritmus számolja meg az adott szám osztóit, elosztva ezt sorban minden számmal 1-től *n*-ig. A döntésnek megfelelő üzenetet az osztók száma alapján írjuk ki.

Algoritmus Prím(*n*, *válasz*):

{ *Bemeneti adat: n. Kimeneti adat: válasz* }

osztók_száma $\leftarrow$ 0

Minden *osztó* = 1, *n* **végezd el:**

Ha *maradék*[*n/osztó*] = 0 **akkor**

osztók_száma $\leftarrow$ *osztók_száma* + 1

vége(ha)

vége(minden)

válasz $\leftarrow$ *osztók_száma* = 2}

Vége(algoritmus)

Az algoritmus optimalizálása: A lépésenkénti finomításnak elvben vége van, hiszen van egy helyesen működő algoritmusunk. De, miután teszteljük és figyelmesen elemezzük, rájövünk, hogy az algoritmust lehetséges *optimalizálni*. Észrevesszük, hogy az osztások

száma fölöslegesen nagy. Ezt a számot lehet csökkenteni, mivel ha 2 és $n/2$ között nincs egyetlen osztó sem, akkor biztos, hogy nincs $n/2$ és n között sem, tehát eldönthető, hogy a szám prím, sőt elég a szám négyzetgyökéig keresni a lehetséges osztót, hiszen ahogy az osztó értékei nőnek a négyzetgyökig, az $[n/osztó]$ hányados értékei csökkennek szintén a négyzetgyök értékéig. Ha egy, a négyzetgyöknél nagyobb osztóval elosztjuk az adott számot, hányadosként egy kisebb osztót kapunk, amit megtaláltunk volna előbb, ha létezett volna ilyen.

Továbbá, a ciklus leállítható amint találtunk egy osztót és a *válasz* hamissá vált. A *Minden* típusú ciklust *Amíg* vagy *Ismételd* típusú ciklussal helyettesítjük. Mivel n nem változik a ciklus magjában, a négyzetgyök kiszámíttatását csak egyszer végezzük el. Mivel a páros számok mind oszthatók 2-vel, és a 2 kivételével nem prímelek, „megszabadulunk” a páros számok fölösleges vizsgálatától, és a páratlan számokat csak páratlan osztókkal próbáljuk osztani. Ahhoz, hogy az algoritmusunk tökéletesen működjön akkor is, ha $n = 1$, a következőképpen járunk el:

Algoritmus Prím(n , *válasz*):

{ *Bemeneti adat: n. Kimeneti adat: válasz* }

```

Ha  $n = 1$  akkor
    prím  $\leftarrow$  hamis
különben
    Ha  $n$  páros akkor
        prím  $\leftarrow$   $n = 2$ 
    különben
        prím  $\leftarrow$  igaz
        osztó  $\leftarrow$  3
        négyzetgyök  $\leftarrow$   $\lceil \sqrt{n} \rceil$ 
        Amíg prím és (osztó  $\leq$  négyzetgyök) végezd el:
            Ha maradék[ $n/osztó$ ] = 0 akkor
                prím  $\leftarrow$  hamis
            különben
                osztó  $\leftarrow$  osztó + 2
            vége(ha)
            vége(amíg)
            vége(ha)
            vége(ha)
        válasz  $\leftarrow$  prím
Vége(algoritmus)

```

Ha ebben az algoritmusban felhasználjuk a matematikából ismert tulajdonságot, és pedíg azt, hogy minden 5-nél nagyobb prímszám $6k \pm 1$ alakú, akkor a vizsgálandó számok száma tovább csökkenthető.

2.1. A moduláris programozás alapszabályai

Az eredeti feladatot részfeladatokra bontjuk. Minden rész számára megtervezzük a megoldást jelentő algoritmust. Ezek az algoritmusok legyenek minél függetlenebbek, de álljanak jól definiált kapcsolatban egymással. A részfeladatok megoldásainak összessége tartalmazza a feladat megoldási algoritmusát.

2.1.1. Moduláris dekompozíció

A moduláris dekompozíció a feladat több, egyszerűbb részfeladatra bontását jelenti, amely részfeladatok megoldása már egymástól függetlenül elvégezhető. A módszert általában ismételten alkalmazzuk, azaz az alrendszereket magukat is felbontjuk. Ezzel lehetővé tesszük azt is, hogy a feladat megoldásán egyszerre több személy is dolgozzon. A módszer egy fával ábrázolható, ahol a fa csomópontjai az egyes dekompozíciós lépéseknek felelnek meg.

2.1.2. Moduláris kompozíció

Olyan szoftverelemek létrehozását támogatja, amelyek szabadon kombinálhatók egymással. Algoritmusainkat már meglévő egységekből építjük fel.

2.1.3. Modulok tulajdonságai

Moduláris érthetőség: A modulok önállóan is egy-egy értelmes egységet alkossanak, megértésükhöz minél kevesebb „szomszédos” modulra legyen szükség.

Moduláris folytonosság: A specifikáció „kis” változtatása esetén a programban is csak „kis” változtatásra legyen szükség.

Moduláris védelem: Célunk a program egészének védelme az abnormális helyzetek hatásaitól. Egy hiba hatása egy – esetleg néhány – modulra korlátozódjon!

2.1.4. A modularitás alapelvei

A modulokat nyelvi egységek támogatassák: A modulok illeszkedjenek a használt programozási nyelv szintaktikai egységeihez.

Kevés kapcsolat legyen: Minden modul minél kevesebb másik modullal kommunikáljon!

Gyenge legyen a kapcsolat: A modulok olyan kevés információt cseréljenek, amennyi csak lehetséges!

Explicit interface használata: Ha két modul kommunikál egymással, akkor annak ki kell derülnie legalább az egyikük szövegéből.

Információ elrejtés: Egy modul minden információjának rejtettnek kell lennie, kivéve, amit explicit módon nyilvánosnak deklaráltunk.

Nyitott és zárt modulok: Egy modult zártnak nevezünk, ha más modulok számára egy jól definiált felületen keresztül elérhető, a többi modul ezt változatlan formában felhasználhatja. Egy modult nyitottnak nevezünk, ha még kiterjeszthető, ha az általa nyújtott szolgáltatások bővíthetők vagy, ha hozzávehetünk további mezőket a benne levő adatszerkezetekhez, s ennek megfelelően módosíthatjuk eddigi szolgáltatásait.

Az újrafelhasználhatóság igényei

A típusok változatossága: A moduloknak többféle típusra is működniük kell, azaz a műveleteket több különböző típusra is definiálni kellene.

Adatstruktúrák és algoritmusok változatossága

Egy típus, egy modul: Egy típus műveletei kerüljenek egy modulba.

Reprezentáció-függetlenség

3. Rendezési algoritmusok

Legyen egy n elemű a sorozat. Rendezett sorozatnak nevezzük a bemeneti sorozat olyan permutációját, amelyben $a_1 \leq a_2 \leq \dots \leq a_n$.

3.1. Buborékredezés (Bubble-sort)

A rendezés során páronként összehasonlítjuk a számokat (az elsőt a másodikkal, a másodikat a harmadikkal és így tovább), és ha a sorrend nem megfelelő, akkor az illető két elemet felcseréljük. Ha volt csere, a vizsgálatot újrakezdjük. Az algoritmus akkor ér véget, amikor az elemek páronként a megfelelő sorrendben találhatók, vagyis a sorozat rendezett.

Mivel a sorozat első bejárása után legalább az utolsó elem a helyére kerül, és a ciklusmag minden újabb végrehajtása után, jobbról balra haladva újabb elemek kerülnek a megfelelő helyre, a ciklus lépésszáma csökkenthető. Az is előfordulhat, hogy a sorozat végén levő elemek már a megfelelő sorrendben vannak, és így azokat már nem kell rendeznünk. Tehát, elegendő a sorozatot csak az utolsó csere helyéig vizsgálni.

Algoritmus Buborékredezés(n, a):

$k \leftarrow n$ { Bemeneti adatok: n, a . Kimeneti adat: a rendezett a sorozat }

Ismételd

$nn \leftarrow k - 1$

$rendben \leftarrow igaz$

Minden $i = 1, nn$ **végezd el:**

Ha $a_i > a_{i+1}$ **akkor**

$rendben \leftarrow hamis$

$a_i \leftrightarrow a_{i+1}$

$k \leftarrow i$

vége(ha)

vége(minden)

ameddig $rendben$

Vége(algoritmus)

3.2. Egyszerű felcseréléses rendezés

Ez a rendezési módszer hasonlít a buborékredezéshez, de kötelezően elvégez minden páronkénti összehasonlítást (Ez az algoritmus mindig $O(n^2)$ bonyolultságú). Ha egy elempár sorrendje nem megfelelő, felcseréli őket.

Algoritmus FelcserélésesRendezés(n, a):

Minden $i = 1, n - 1$ **végezd el:** { Bemeneti adatok: n, a ; Kimeneti adat: a rendezett a sorozat }

Minden $j = i + 1, n$ **végezd el:**

Ha $a_i > a_j$ **akkor**

$a_i \leftrightarrow a_j$

vége(ha)

vége(minden)

vége(minden)

Vége(algoritmus)

3.3. Minimum/maximum kiválasztásra épülő rendezés

Növekvő sorrendbe rendezés esetén kiválaszthatjuk a sorozat legkisebb elemét. Ezt az első helyre tesszük úgy, hogy felcseréljük az első helyen található elemmel. A következő lépésben hasonlóan járunk el, de a minimumot a második helytől kezdődően keressük. A továbbiakban ugyanezt tesszük, míg a sorozat végére nem érünk.

Algoritmus Minimumkiválasztás(n, a):

```

Minden  $i = 1, n-1$  végezd el: { Bemeneti adatok:  $n, a$ ; Kimeneti adat: a rendezett a sorozat }
     $ind\_min \leftarrow i$ 
    Minden  $j = i+1, n$  végezd el:
        Ha  $a_{ind\_min} > a_j$  akkor
             $ind\_min \leftarrow j$ 
        vége(ha)
    vége(minden)
     $a_i \leftrightarrow a_{ind\_min}$ 
vége(minden)
Vége(algoritmus)

```

3.4. Beszúró rendezés

A beszúró rendezés hatékony algoritmus kisszámú elem rendezésére. Úgy dolgozik, ahogy bridzsezés közben a kezünkben lévő lapokat rendezzük: üres bal kézzel kezdünk, a lapok fejfelé az asztalon vannak. Felveszünk egy lapot az asztalról, és elhelyezzük a bal kezünkben a megfelelő helyre. Ahhoz, hogy megtaláljuk a megfelelő helyet, a felvett lapot összehasonlítjuk a már kezünkben lévő lapokkal, jobbról balra. A bemeneti elemek helyben rendeződnek: a számokat az algoritmus az adott tömbön belül rakja a helyes sorrendbe, belőlük bármikor legfeljebb csak állandónyi tárolódik a tömbön kívül. Amikor a rendezés befejeződik, az eredeti tömb tartalmazza a rendezett elemeket.

Algoritmus Beszúró_Rendezés(n, a):

```

Minden  $j = 2, n$  végezd el: { Bemeneti adatok:  $n, a$  }
    { Kimeneti adat: a rendezett a sorozat }
    { beszúrjuk  $a_j$ -t az  $a_1, \dots, a_{j-1}$  rendezett sorozatba }
     $segéd \leftarrow a_j$ 
     $i \leftarrow j - 1$ 
    Amíg  $(i > 0)$  és  $(a_i > segéd)$  végezd el:
         $a_{i+1} \leftarrow a_i$ 
         $i \leftarrow i - 1$ 
    vége(amíg)
     $a_{i+1} \leftarrow segéd$ 
vége(minden)
Vége(algoritmus)

```

3.5. Leszámláló rendezés (ládarendezés, binsort)

Az eddigiekben tárgyalt algoritmusok a legrosszabb esetben $O(n^2)$ időben rendeznek n elemet. Ezek az algoritmusok a rendezéshez csak a bemeneti tömb elemein történő összehasonlításokat használják. Éppen ezért, ezeket az algoritmusokat *összehasonlító rendezéseknek* nevezzük.

A most következő rendező algoritmus *lineáris* idejű. Ez az algoritmus nem az összehasonlítást használja a rendezéshez, hanem kihasználja a rendezendő sorozat bizonyos tulajdonságait, éspedig azt, hogy az elemek sorszámozható típusúak, olyan értékekkel, amelyek egy segédtömb indexei lehetnek.

A segéd tömb i -edik elemében azt tartjuk nyilván, hogy hány darab i -vel egyenlő elemet találtunk az eredeti tömbben. A lineáris feldolgozás után felülírjuk az eredeti tömb elemeit a segéd tömb elemeinek értékei alapján.

Algoritmus Ládarendezés(a, n):

Minden $i = 1, k$ **végezd el:**

{ Bemeneti adatok: n, a ; Kimeneti adat: a }

$segéd_i \leftarrow 0$

vége(minden)

Minden $j = 1, n$ **végezd el:**

$segéd_{a_j} \leftarrow segéd_{a_j} + 1$

vége(minden)

$q \leftarrow 0$

Minden $i = 1, k$ **végezd el:**

{ a segéd tömbnek k eleme van }

Minden $j = 1, segéd_i$ **végezd el:**

$q \leftarrow q + 1$

{ a segéd _{i} elemek összege n }

$a_q \leftarrow i$

{ tehát a feldolgozások száma n }

vége(minden)

vége(minden)

Vége(algoritmus)

4. Rekurzió

A rekurzió egy különleges programozási stílus, inkább „technika” mint módszer. A rekurzív programok tömören és világosan kódolják az algoritmusokat, bonyolultságuktól függetlenül. A rekurzív programozás, mint fogalom, a matematikai értelmezéshez közelálló módon került közhasználatba.

Példák

1. A matematikában, egy fogalmat rekurzív módon definiálunk, ha a definíción belül felhasználjuk magát a definiálandó fogalmat. Például, a faktoriális rekurzív definícióját egy adott n szám esetében, a matematikus így fejezi ki:

$$n! = \begin{cases} 1, & \text{ha } n = 0 \\ n \cdot (n-1)!, & \text{ha } n \in \mathbb{N}^* \end{cases}$$

2. A bináris fa *Knuth* által megfogalmazott definíciója már szorosan kapcsolódik az informatikához: Egy bináris fa vagy üres, vagy tartalmaz egy csomópontot, amelynek van egy bal meg egy jobb utóda, amelyek szintén bináris fák.

A programozásban a rekurzió alprogramok formájában jelenik meg, éspedig olyan függvényeket, illetve eljárásokat nevezünk rekurzívoknak, melyek meghívják önmagukat. Ha ez a hívás az illető alprogram összetett utasításában benne foglaltatik, közvetlen (direkt) rekurzióról beszélünk. Ha egy rekurzív alprogramot egy másik alprogram hív meg, amelyet ugyanakkor az illető alprogram hív (közvetve, vagy közvetlenül) akkor közvetett (indirekt) rekurzióról beszélünk. Közvetett rekurzió esetén is arról van szó, hogy egy alprogram meghívja önmagát, hiszen a rekurzív hívás aközben történik, miközben a számítógép azt az összetett utasítást hajtja végre, amely az illető alprogramot alkotja.

Egy alprogram aktív a hívásától kezdődően, addig, amíg a végrehajtás visszatér a hívás helyére. Egy alprogram aktív marad akkor is, ha végrehajtása során más alprogramokat hív meg. Tehát, a rekurzió fogalmát kifejezhetjük úgy is, hogy egy alprogram akkor rekurzív, ha meghívja önmagát, amikor még aktív.

Egy rekurzív alprogram végrehajtása azonos módon történik, mint bármely nem rekurzív alprogramé. A rekurzív eljárások esetében is, hasonlóan a nem rekurzívakhoz, az aktiválás feltételezi a veremhasználatot, ahol a paramétereket, a visszatérés helyének címét, valamint a lokális változókat tárolja (minden aktuális aktiválás idejére) a programozási környezet. Mivel a verem mérete véges, bizonyos számú aktiválás után bekövetkezhet a túlsordulás és a program hibaüzenettel kilép. Mivel ezt a hibát feltétlenül el kell kerülnünk, a rekurzív alprogramot csak egy bizonyos feltétel teljesülésekor hívjuk meg újra. A legutolsó aktiválás alkalmával a feltétel hamis, ennek következtében nem történik újrahívás, hanem a feltétel másik ágának megfelelő utasítás (ennek hiányában, a feltétel utáni utasítás) kerül sorra. Új aktiválás csak az újrahívási feltétel teljesülésekor történik; az újrahívások száma meghatározza a rekurzió mélységét; az előző megjegyzést figyelembe véve, egy rekurzív megoldás csak akkor hatékony, ha ez a mélység nem túl nagy. Ha az újrahívási feltétel egy adott pillanatban nem teljesül, az újraaktiválások sora leáll; ennek következtében a feltétel tagadása a rekurzióból való kilépés feltétele; a feltételnek a rekurzív eljárás paramétereitől kell függnie és/vagy a helyi változóktól; a kilépést a paraméterek és a lokális változók módosulása (egyik hívástól a másikig) biztosítja. Ha ezeket a feltételeket nem tartjuk be, a program hibaüzenettel kilép. Egy újrahívás (közvetlen rekurzió esetén), többször is előfordulhat egy rekurzív eljárásban; ebben az esetben, természetesen, különbözni fognak a visszatérési címek. A rekurzió késlelteti az eljárás azon utasításainak végrehajtását, amelyek a rekurzív hívás utáni részhez tartoznak. Minden eddigi állítás igaz a rekurzív függvények esetében is, csak a hívás módja más; egy rekurzív függvényt egy kifejezésből hívunk meg;

egy rekurzív függvény összetett utasítása, hasonlóan a nem rekurzív függvényekhez, tartalmazni fog egy értékadó utasítást, amely a függvény azonosítójának ad értéket. Ebbe az utasításba kerül, többnyire, az újrahívás.

4.2. Megoldott feladatok

4.2.1. Egy szó betűinek megfordítása

Olvassunk be egymás után több betűt a szóközkarakter megjelenéséig, majd írjuk ki ezeket a betűket fordított sorrendben!

A feladat követelményének megfelelően betűk szintjén fogunk dolgozni. A megfordított kiírás azt jelenti, hogy miután beolvastunk egy betűt, nem írjuk ki, csak azután, hogy megfordítottuk a többi betűt. A fennmaradt rész esetében ugyanígy járunk el; a módszer addig folytatódik, amíg eljutunk az utolsó betűhöz, amikor nincs mit megfordítani. Rekurzív módon ezt a következőképpen lehet leírni:

Algoritmus Fordít:

```

Be: betű { nincs paraméter, mivel az alprogramban olvasunk be és írunk ki }
Ha nem szóköz akkor
    Fordít { meghívja önmagát, hogy megfordíthassa a fennmaradt részt }
különb
    Ki: 'Fordított szó: ' { ez az utasítás egyszer hajtódik végre }
vége(ha)
Ki: betű

```

Vége(algoritmus)

A rekurzió meghatározza az eljárás záró részének az aktiválások fordított sorrendjében való végrehajtását (a mi esetünkben: Ki: betű), így természetes módja a feladat megoldásának.

4.2.2. Szavak sorrendjének megfordítása

Olvassunk be n szót, majd írjuk ki ezeket a beolvasás fordított sorrendjében! Ne használjunk tömböt!

Algoritmus Szavakat_fordít_1(n):

```

Be: szó { az első hívás aktuális paramétere  $n$  = szavak száma }
Ha  $n > 1$  akkor
    Szavakat_fordít_1( $n-1$ )
különb
    Ki: 'Fordított sorrendben: '
vége(ha)
Ki: szó

```

Vége(algoritmus)

Az eredeti feladat n szó megfordítását valósítja meg, a részfeladatok pedig egyre kevesebb szó megfordítását végzik. Ha fordítva indulunk, vagyis „megfordítjuk” egy szónak a sorrendjét, majd a többiét, akkor az algoritmus a következő:

Algoritmus Szavakat_fordít_2(i):

```

Be: szó { most az első hívás aktuális paramétere 1 }
Ha  $i < n$  akkor
    Szavakat_fordít_2( $i+1$ )
különb
    Ki: 'Fordított sorrendben: '
vége(ha)
Ki: szó

```

Vége(algoritmus)

4.2.3. Faktoriális

Írjuk ki az n adott szám faktoriálisát!

Megoldás

Felhasználjuk a faktoriális matematikai definícióját, amit a $Fakt(n)$ alprogramban implementálunk, amely függvény típusú. Az első hívás $Fakt(n)$ -nel történik.

Algoritmus $Fakt(n)$: { Bemeneti adat: n }
Ha $n = 0$ **akkor**
 $Fakt \leftarrow 1$
különben
 $Fakt \leftarrow n * Fakt(n - 1)$
vége(ha)
Vége(algoritmus)

A faktoriális tulajdonképpen kiszámolása akkor történik, amikor kilépünk egy-egy hívásból. Mivel minden egyes alkalommal más-más n paraméterre van szükség, fontos, hogy ezt értékként adjuk át; így kifejezéseket is írhatunk az aktuális paraméter helyére.

A faktoriális nem előnyös rekurzívan számolni, mivel sokkal időigényesebb, mint az iteratív megoldás, hiszen a $Fakt(n)$ függvény $(n+1)$ -szer fog aktiválódni.

4.2.4. Legnagyobb közös osztó

Számítsuk ki két természetes szám $(n, m \in \mathbb{N}^*)$ legnagyobb közös osztóját rekurzívan.

Ha figyelmesen elemezzük Eukleidész algoritmusát, észrevesszük, hogy a legnagyobb közös osztó ($Lnko(m, n)$) egyenlő n -nel (ha n osztója m -nek) különben egyenlő $Lnko(n, maradék[m/n])$ -val. Tehát fel lehet írni a következő rekurzív definíciót:

$$Lnko(m, n) = \begin{cases} n, & \text{ha } maradék[m/n] = 0 \\ Lnko(n, maradék[m/n]), & \text{ha } maradék[m/n] \neq 0 \end{cases}$$

Algoritmus $Lnko(m, n)$: { Bemeneti adatok: m, n }
 $mar \leftarrow maradék[m/n]$
Ha $mar = 0$ **akkor**
 $Lnko \leftarrow n$
különben
 $Lnko \leftarrow Lnko(n, mar)$
vége(ha)
Vége(algoritmus)

Az első hívás történhet például egy kiíró utasításból: $Ki: Lnko(m, n)$.

4.2.5. Descartes-szorzat

Egy rajzon n virágot fogunk kiszínezni. A festékeket az $1, 2, \dots, m$ számokkal kódoljuk. Bármely virág, bármilyen színű lehet, de szeretnénk tudni, hány féle módon lehetne ezeket különböző módon kiszínezni. Tulajdonképpen a következő Descartes-szorzatot kell

$$M^n = \underbrace{M \times M \times \dots \times M}_{n\text{-szer}}, \text{ ha } M = \{1, 2, \dots, m\}.$$

generálnunk:

Algoritmus $Descartes_szorzat(i)$: { Bemeneti adat: i , az első híváskor $= 1$ }
Minden $j=1, m$ **végezd el**:
 $x_i \leftarrow j$

```

Ha  $i < n$  akkor
    Descartes_szorzat_3( $i+1$ )
különben
    Kiír
vége(ha)
vége(minden)
Vége(algoritmus)

```

4.2.6. k elemű részhalmazok

Adva vannak az n és a k ($1 \leq k \leq n$) egész számok. Generáljuk rekurzívan az $\{1, 2, \dots, n\}$ halmaz minden k elemet tartalmazó részhalmazát!

Megoldás

Az $\{1, 2, \dots, n\}$ halmaz k elemet tartalmazó részhalmaza egy k elemű tömb segítségével kódolható: $x_1, x_2, \dots, x_k$. A részhalmaz elemei különbözők és nem számít a sorrendjük. Ezért, a részhalmazok generálása során vigyázunk, hogy az x sorozatba ne generáljuk kétszer vagy többször ugyanazt a részhalmazt (esetleg, más sorrendű elemekkel). Ha az x sorozatba az elemek szigorúan növekvő sorrendben kerülnek ($x_1 < x_2 < \dots < x_k$), egy részhalmazt csak egyszer állíthatunk elő.

Mivel minden x_i szigorúan nagyobb, mint x_{i-1} , az értékei $x_{i-1} + 1$ -től nőnek, $n - (k - i)$ -ig.

Algoritmus Részhalmazok(i) :

```

{  $M = (1, 2, \dots, n)$ ,  $x$  globális  $\Rightarrow x_i = 0$ ,  $i = 0, 20$ ,  $k$  is globális }
Minden  $j = x_{i-1} + 1, n - k + i$  végezd el:
     $x_i \leftarrow j$ 
    Ha  $i < k$  akkor
        Részhalmazok( $i+1$ )
    különben
        Kiír
    vége(ha)
vége(minden)
Vége(algoritmus)

```

A részhalmazokat generáló algoritmust az i paraméter 1 értékére hívjuk meg.

4.2.7. Fibonacci-sorozat

Generáljuk a Fibonacci-sorozat első n elemét!

$$fib(n) = \begin{cases} 0, & \text{ha } n = 1 \\ 1, & \text{ha } n = 2 \\ fib(n-2) + fib(n-1), & \text{ha } n \geq 3 \end{cases}$$

Megoldás

Az n -edik elem kiszámításához szükségünk van az előtte található két elemre. De ezeket szintén az előttük levő elemekből számítjuk ki.

Algoritmus Fibo(n) :

```

Ha  $n = 1$  akkor
    Fibo  $\leftarrow 0$ 
különben
    Ha  $n = 2$  akkor
        Fibo  $\leftarrow 1$ 
    különben
        Fibo  $\leftarrow$  Fibo( $n-2$ ) + Fibo( $n-1$ )

```

vége(ha)
vége(ha)
Vége(algoritmus)

A fenti algoritmus nagyon sokszor hívja meg önmagát ugyanarra az értékre, mivel minden új elem kiszámításakor el kell jutnia a sorozat első eleméhez, amittől kezdődően újból, meg újból generálja ugyanazokat az elemeket.

A hívások számát csökkenthetjük, ha a kiszámolt értékeket megőrizzük egy sorozatban. Legyen ez a sorozat f , amelyet globális változóként kezelünk.

Algoritmus $\text{Fib}(n)$:
Ha $n > 2$ **akkor**
 $\text{Fib}(n-1)$
 $f_n \leftarrow f_{n-1} + f_{n-2}$
különben
 $f_1 \leftarrow 0$
 Ha $n = 1$ **akkor**
 $f_1 \leftarrow 0$
 különben
 $f_2 \leftarrow 1$
 vége(ha)
vége(ha)
Vége(algoritmus)

Megjegyzések:

1. A rekurzió hasznos, ha:
 - a feladat eredménye rekurzív szerkezetű,
 - a feladat szövege rekurzív szerkezetű,
 - a megoldás legjobb módszere a visszalépéses keresés (backtracking) módszer,
 - a megoldás legjobb módszere az oszd meg és uralkodj (divide et impera) módszer,
 - a feldolgozandó adatok rekurzívan definiáltak (pl. bináris fák).
2. A rekurzív programozás legszembeötlőbb előnye az, hogy az algoritmus tömör és világos, természetszerűen tükrözi a folyamatot, amit modellez.
3. Ugyanakkor, nem tanácsos rekurziót alkalmazni, ha a feladatot megoldhatjuk egy egyszerűbb iteratív algoritmussal (mivel így a program gyorsabb lesz). Egy iteratív algoritmusnak az ellenőrzése is könnyebb. Tehát, nem alkalmazunk rekurzív algoritmust például a Fibonacci-sorozat elemeinek generálására, kombinációk generálására, illetve más kombinatorikai feladatok megoldására.

4.2.8. Az $\{1, 2, \dots, n\}$ halmaz minden részhalmaza

Generáljuk az $\{1, 2, \dots, n\}$ halmaz minden részhalmazát!

Elemzés

A halmazokat ebben az esetben is egy $x_1 < x_2 < \dots < x_i$ sorozattal ábrázoljuk, ahol $i = 1, 2, \dots, n$.

Az alábbi algoritmust $i = 1$ -re hívjuk meg. Az x sorozat 0 indexű elemét 0 kezdőértékkel látjuk el. Szükségünk lesz az x_0 elemre is, mivel az algoritmusban a sorozat minden x_i elemét, tehát x_1 -et is az előző elemből számítjuk ki. A j változóban generáljuk azokat az értékeket, amelyeket rendre felvesz az x sorozat aktuális eleme. Ezek a j értékek 1-gyel nagyobbak, mint a részhalmazba utoljára betett elem értéke és legtöbb n -nel egyenlők. Így a

részhalmazokat az úgynevezett lexikográfikus sorrendben generáljuk. Figyelemre méltó, hogy minden új elem generálása egy új részhalmazhoz vezet.

Algoritmus Részhalmaz(i)
Minden $j = x_{i-1} + 1, n$ **végezd el:**
 $x_i \leftarrow j$
 Kiír(i)
 Részhalmaz($i+1$)
vége(minden)
Vége(algoritmus)

A kilépési feltétel ($x_i = n$) el van rejtve a *Minden* típusú struktúrába: ha $x_i = n$, a ciklusváltozó kezdőértéke $x_i + 1$ nagyobb, mint a végső érték, így a *Minden* ciklusmagja nem lesz többet végrehajtva és a program kilép az aktuális hívásból. Az algoritmust *Részhalmazok*(1) alakban hívjuk meg.

4.2.9. Partíciók

Generáljuk az $n \in \mathbb{N}^*$ szám partícióit!

Megoldás

Partíció alatt azt a felbontást értjük, amelynek során az $n \in \mathbb{N}^*$ számot pozitív számok összegeként írjuk fel: $n = p_1 + p_2 + \dots + p_k$, ahol $p_i \in \mathbb{N}^*$, $i = 1, 2, \dots, k$, $k = 1, \dots, n$. Két partíciót kétféleképpen tekinthetünk különbözőnek:

- Két partíció különbözik egymástól, ha vagy az előforduló értékek vagy az előfordulásuk sorrendje különbözik (erre adunk megoldást).
- Két partíció különbözik egymástól, ha az előforduló értékek különböznek (lásd a 8. kitűzött feladatot a fejezet végén).

A generálás során, rendre kiválasztunk egy lehetséges értéket a partíció első p_1 eleme számára és generáljuk a fennmaradt $n - p_1$ szám partícióit. Ez a különbség az n új értéke lesz, amellyel ugyanúgy járunk el. Egy partíciót legeneráltunk, és kiírhatjuk, ha n aktuális értéke 0. Az alábbi algoritmust a *Partíció*(1, n) utasítással hívjuk meg először.

Algoritmus Partíció(i, n):
Minden $j = 1, n$ **végezd el:**
 $p_i \leftarrow j$
Ha $j < n$ **akkor**
 Partíció($i+1, n-j$)
különb
 Kiír(i)
vége(ha)
vége(minden)
Vége(algoritmus)

5. A visszalépéses keresés módszere (backtracking)

Az algoritmusok behatóbb tanulmányozása meggyőzött bennünket, hogy tervezésükkor meg kell vizsgálnunk a végrehajtásukhoz szükséges időt. Ha ez az idő elfogadhatatlanul nagy, más megoldásokat kell keresnünk. Egy algoritmus „elfogadható”, ha végrehajtási ideje *polinomiális*, vagyis n^k -nal arányos (adott k -ra és n bemeneti adatra).

Ha egy feladatot csak exponenciális algoritmussal tudunk megoldani, alkalmazzuk a backtracking (visszalépéses keresés) módszert, amely exponenciális ugyan, de megpróbálja csökkenteni a generálandó próbálkozások számát.

5.1. A visszalépéses keresés általános bemutatása

A visszalépéses keresés azon feladatok megoldásakor alkalmazható, amelyeknek eredményét az $M_1 \times M_2 \times \dots \times M_n$ Descartes-szorzatnak azon elemei alkotják, amelyek eleget tesznek bizonyos belső feltételeknek. Az $M_1 \times M_2 \times \dots \times M_n$ Descartes-szorzat a megoldások tere (az eredmény egy x sorozat, amelynek x_i eleme az M_i halmazból való).

A visszalépéses keresés nem generálja a Descartes-szorzat minden $x = (x_1, x_2, \dots, x_n) \in M_1 \times M_2 \times \dots \times M_n$ elemét, hanem csak azokat, amelyek esetében remélhető, hogy megfelelnek a belső feltételeknek. Így, megpróbálja csökkenteni a próbálkozásokat.

Az algoritmusban az x tömb elemei egymás után, egyenként kapnak értékeket: x_i számára csak akkor „javasolunk értéket”, ha $x_1, x_2, \dots, x_{i-1}$ már kaptak végleges értéket az aktuálisan generált eredményben. Az x_i -re vonatkozó „javaslat”-ot akkor fogadjuk el, amikor $x_1, x_2, \dots, x_{i-1}$ értékei az x_i értékével együtt megvalósítják a *belső feltételeket*. Ha az i -edik lépésben a belső feltételek nem teljesülnek, x_i számára új értéket választunk az M_i halmazból. Ha az M_i halmaz minden elemét kipróbáltuk, visszalépünk az $i-1$ -edik elemhez, amely számára új értéket „javasolunk” az M_{i-1} halmazból.

Ha az i -edik lépésben a belső feltételek teljesülnek, az algoritmus folytatódik. Ha szükséges folytatni, mivel a számukat ismerjük és még nem generáltuk mindegyiket, vagy valamilyen másképp kifejezett tulajdonság alapján eldöntöttük, hogy még nem jutottunk eredményhez, a *folytatási feltételek* alapján folytatjuk az algoritmust.

Azokat a lehetséges eredményeket, amelyek a megoldások teréből vették értékeiket úgy, hogy teljesítik a belső feltételeket, és amelyek esetében a folytatási feltételek nem kérnek további elemeket, végeredményeknek nevezzük.

A belső feltételek és a folytatási feltételek között szoros kapcsolat áll fenn. Ezek kifejezésmódjának szerencsés megválasztása többnyire a számítások csökkentéséhez vezethet.

A belső feltételeket egy külön algoritmusban vizsgáljuk. Legyen ennek a neve *Megfelel*, és paramétere az aktuálisan generált elem i indexe. Ez az alprogram *igaz* értéket térít vissza, ha az x_i elem az eddig generált $x_1, x_2, \dots, x_{i-1}$ elemekkel együtt megfelel a belső feltételeknek, és hamis értéket ellenkező esetben.

Algoritmus *Megfelel*(i):

Megfelel $\leftarrow$ igaz

Ha a belső feltételek $x_1, x_2, \dots, x_i$ esetében nem teljesülnek **akkor**

Megfelel $\leftarrow$ hamis

vége(ha)

Vége(algoritmus)

Az eredményt a következő algoritmussal generáljuk:

Algoritmus $\text{Rekurzív_Backtracking}(i)$:

Minden $m_j \in M_i$ *értékre végezd el*:

$x_i \leftarrow m_j$

Ha $\text{Megfelel}(i)$ **akkor** $\{ \text{megvalósulnak a belső feltételek } x_1, x_2, \dots, x_i \text{ esetében} \}$

Ha $i < n$ **akkor**

$\text{Rekurzív_Backtracking}(i+1)$

különben

Ki: $x_1, x_2, \dots, x_n$

vége(ha)

vége(ha)

vége(minden)

Vége(algoritmus)

Az algoritmust az $i = 1$ értékre hívjuk meg először.

A módszer eredményessége nagymértékben függ a folytatási feltételek szerencsés kiválasztásától. Minél hamarabb állítjuk le egy eredmény generálását, annál kisebb a rekurzió mélysége, de a feltételek nem lehetnek túl bonyolultak, mivel ezeket minden lépésnél végrehajtja az algoritmus.

A módszer azoknak a feladatoknak a megoldásakor alkalmazható, amelyekben a követelményeknek megfelelően minden eredményt meg kell állapítanunk. Ha az $M_1 \times \dots \times M_n$ Descartes-szorzat számossága nem túl nagy, valamint a feltételek biztosítanak egy nem túl mély rekurziót, eredményesen alkalmazható.

Összefoglalva a lényegét, a következő lépéseket kell elvégeznünk:

- az *eredmény kódolása* – meg kell állapítanunk az x_i elemek jelentését az illető feladat esetében, valamint meg kell határoznunk az M_i , $i = 1, 2, \dots, n$ halmazokat.
- a belső, majd a folytatási feltételek megállapítása.
- a $\text{Rekurzív_Backtracking}(i)$ vagy iteratív változatának átírása.

5.2. Megoldott feladatok

5.2.1. 8 királynő a sakktáblán

Írjuk ki az összes lehetséges módját annak, ahogyan 8 királynő elhelyezhető egy sakktáblán úgy, hogy ne támadják egymást. Két királynő támadja egymást, ha ugyanazon a soron, oszlopon, illetve átlón helyezkedik el.

Megoldás

Minden királynőt egymás után elhelyezünk a neki megfelelő sorba az első oszloppal kezdődően, amíg meg nem találjuk azt az oszlopot, amelyben nem támad más, eddig feltett királynőt. Ha egy királynőt nem lehet elhelyezni, visszatérünk az előzőhöz és számára tovább keresünk megfelelő, nagyobb sorszámú oszlopot.

Az eredményt egy egydimenziós tömbbel (K_i , $i = 1, 2, \dots, 8$) kódoljuk. A tömb K_i elemeinek értéke az oszlop sorszáma, ahova az i -edik királynőt tettük (az i -edik sorban).

A sakktáblának 8 oszlopa van, tehát $K_i \in \{1, 2, \dots, 8\}$, $i = 1, \dots, 8$.

Az eddigiekből következik, hogy egy eredmény az $\{1, 2, \dots, 8\}^8$ Descartes-szorzat eleme. Tehát, ha meg akarjuk oldani a feladatot, tulajdonképpen az $\{1, 2, \dots, 8\}^8$ Descartes-szorzat

egy részhalmazát kell meghatároznunk, azzal a feltétellel, hogy a 8 királynő, amelyek a $K_1, K_2, \dots, K_8$ oszlopokban találhatók, ne támadja egymást. A kódolás sajátos módja biztosítja, hogy soronkénti támadási lehetőség nincs, hiszen minden királynő új sorba kerül. De például, ha az első két királynő egymást támadja, nem generálunk fölöslegesen $8^6 = 262144$ elemet a $\{1, 2, \dots, 8\}^8$ Descartes-szorzatból.

A második észrevétel a feladat rekurzív megfogalmazását teszi lehetővé: elhelyezzük az első királynőt, rendre az első sor első, második, ..., 8-dik oszlopába, majd megoldjuk a feladatot a fennmaradt 7 királynő esetében, de úgy, hogy mindig ellenőrizzük, hogy egy új királynő ne támadjon egyet sem a már elhelyezettek közül.

Általánosan megfogalmazva: az i -edik királynő esetében meg kell határoznunk minden helyet, ahova ezt el lehet helyezni az i -edik sorban úgy, hogy ne támadjon egyet sem azok közül, amelyek az első, második, ..., $i-1$ -edik sorban már el vannak helyezve. Tehát elhelyezzük az i -edik királynőt, majd megoldjuk ugyanezt a feladatot az $i+1$ -edik királynő esetében.

Ha minden királynőt elhelyeztük, van egy eredmény, amit ki kell írunk. Az elhelyezést a $Királynő(i)$ rekurzív alprogram végzi el, a támadási lehetőséget a $Nem_támad(i)$ logikai függvény ellenőrzi.

Ahhoz, hogy két királynő ne támadja egymást, a következő relációknak kell teljesülniük: $K_i \neq K_j, i - j \neq |K_i - K_j|, j = 1, 2, \dots, i - 1$.

Algoritmus $Nem_támad(i)$:

$Jó \leftarrow igaz$

{ $Jó =$ lokális változó, K globális }

$j \leftarrow 1$

Amíg $(j \leq i-1)$ **és** $Jó$ **végezd el:**

Ha $(K_i = K_j)$ **vagy** $(i-j = |K_i - K_j|)$ **akkor**

$Jó \leftarrow hamis$

{ az i . és j . királynők támadják egymást }

különb

$j \leftarrow j + 1$

vége(ha)

vége(amíg)

$Nem_támad \leftarrow Jó$

Vége(algoritmus)

Algoritmus $Királynő(i)$:

Minden $j=1, 8$ **végezd el:**

$K_i \leftarrow j$

{ az i -edik királynőt a j -edik oszlopba tesszük }

Ha $Nem_támad(i)$ **akkor**

{ az i -edik királynő nem támadja egyiket sem }

Ha $i < 8$ **akkor**

$Királynő(i+1)$

különb

$Kiír$

vége(ha)

vége(ha)

vége(minden)

Vége(algoritmus)

Az első hívás alakja: $Királynő(1)$.

5.2.2. Variációk

Az óvónéni a karácsonyi ünnepélyre készül. A díszterem színpadán n széket lehet egy sorban elhelyezni, de a csoportban m óvódás van ($n < m$). Írjuk ki minden lehetséges módját annak, ahogy az óvódások leülhetnek az n székre.

Megoldás

Eltérünk az eredeti sablontól, hiszen fölösleges „javasolni”, hogy üljön le egy már leültetett gyermek.

Az eredmény kódolása: Az x_i az i -edik székre ülő gyerek nevének az indexe. Tehát $x_i \in \{1, 2, \dots, m\}$, ahol m a gyermekek száma, ($i = 1, 2, \dots, n$).

Belső feltételek: $x_i \neq x_j, i \neq j, i, j = 1, 2, \dots, n$.

Folytatási feltételek: $x_i \neq x_j, i \neq j, j = 1, 2, \dots, i - 1$.

Mivel x_i értéke egy index, a folytatási feltételek kifejezhetők egyszerűbben egyetlen feltétellel. Azt fogjuk kifejezni, hogy az i -edik székre csak olyan gyerek ülhet le, aki pillanatnyilag még áll. Fölhasználunk egy *még_áll* logikai tömböt, ahol *még_áll_j* igaz, ha a j -edik gyerek még nem ült le, és hamis ellenkező esetben.

Az $x_i \neq x_j, j = 1, 2, \dots, i - 1$ feltételek a következőképpen alakulnak át: *még_áll_{x_i}* = igaz.

A *még_áll* tömb elemeinek kezdőértéke igaz, mivel még senki nem ült le; majd az ültetési folyamat során a megfelelő elemek hamis értéket kapnak. Valahányszor egy ültetési rend megváltozik a j -edik gyermek feláll az i -edik székről és oda más gyermek ül le; a j -edik gyermek felállítása maga után vonja a megfelelő *még_áll_j* visszaállítását igaz-ra.

Ez a megoldás hatékonyabb, mint az, amelyet a sablon alapján készíthetnénk, hiszen kevesebb összehasonlítást fog végezni.

Algoritmus Variáció(i):

Minden $j=1, m$ **végezd el:**

Ha *még_áll_j* **akkor**

{ *a j-edik gyermek még áll* }

$x_i \leftarrow j$

még_áll_j $\leftarrow$ hamis

{ *a j-edik gyermek az i-edik széken ül* }

Ha $i < n$ **akkor**

Variáció($i+1$)

különben

Kiír

vége(ha)

még_áll_j $\leftarrow$ igaz

{ *a j-edik gyermek feláll* }

vége(ha)

vége(minden)

Vége(algoritmus)

5.2.3. Zárójelek

Írjunk ki minden helyesen nyitó és csukó n zárójelet tartalmazó karakterláncot!

Megoldás

Az eredmény kódolása: Ha n páros szám, az eredmények az M^n halmaz elemei, ahol $M = \{ '(', ')' \}$ és $x_i \in M, i = 1, \dots, n$. Ha n páratlan, akkor nincs megoldás.

Belső feltételek: Adott pillanatban ne létezzen több csukó zárójel, mint nyitó, de nyitó sem lehet több, mint $n/2$.

Folytatási feltételek: Amikor elhelyeztük az n -edik karaktert is, az eredmény kiírható.

Jelöljük *ny*-nyel és *cs*-vel a nyitó, illetve a csukó zárójelek számát. A folytatási feltételek különböznek az x_i elemek értékének függvényében:

$$\begin{cases} ny < \frac{n}{2} & , \text{ha } x_i = '(' \\ cs < ny & , \text{ha } x_i = ')' \end{cases} \quad \forall i = 2, 3, \dots, n$$

Mivel bármely eredményben $x_1 = '('$ és $x_n = ')'$, a hívó programegységben elvégezzük az inicializálásokat: $x_1 \leftarrow '('$ és $x_n \leftarrow ')'$. Az első hívás alakja: *Zárójel*(2, 1, 0).

Algoritmus *Zárójel*(i, ny, cs):

Ha i = n **akkor** Ki: x

{ kilépési feltétel }

különben

Ha ny < [n/2] **akkor**

$x_i \leftarrow '('$

Zárójel(i+1, ny+1, cs)

vége(ha)

Ha cs < ny **akkor**

$x_i \leftarrow ')'$

Zárójel(i+1, ny, cs+1)

vége(ha)

vége(ha)

Vége(algoritmus)

5.2.4. Labirintus

Egy labirintust egy $L_{(n \times m)}$ kétdimenziós tömbben tárolunk, amelyben a folyosónak megfelelő elemek értéke 1; ezek az értékek egymás után következnek a labirintusnak megfelelő tömbben, egy bizonyos sorban, vagy oszlopban. Egy személyt ejtőernyővel leengednek a labirintusba az (i, j) helyre. Írjunk ki minden olyan utat, amely kivezet a labirintusból! Egy út nem érintheti kétszer ugyanazt a helyet. A labirintusból a tömb szélén léphetünk ki.

Elemzés

Az eredmény kódolása: A feladat minden kivezető utat kéri. Egy utat az $x_1, x_2, \dots, x_k$ és $y_1, y_2, \dots, y_k$ sorozatokkal kódolunk, amelyek azokat a sorokat és oszlopokat tartalmazzák, amelyeknek érintésével kifele haladunk a labirintusból. $x_i \in \{1, 2, \dots, n\}$, $y_i \in \{1, 2, \dots, m\}$, $i = 1, 2, \dots, k$.

Belső feltételek: Az útvonalra a következő belső feltételek érvényesek:

a) A folyosón kell haladnia: $L(x_i, y_i) = 1$, $i = 1, 2, \dots, k$.

b) Nem léphet kétszer ugyanarra a helyre: $(x_i, y_i) \neq (x_j, y_j)$, $i, j = 1, 2, \dots, k$, $i \neq j$.

c) Biztosítania kell a labirintusból való kijutást: $x_k \in \{1, n\}$ vagy $y_k \in \{1, m\}$

Folytatási feltételek: Tartalmazzák az a) és b) ellenőrzését minden lépésnél. A b) feltétel az i -edik lépésben: $(x_i, y_i) \neq (x_j, y_j)$, $j = 1, \dots, i - 1$.

Az eredményt az $eredm_{ij}$ ($i = 1, 2, \dots, n$, $j = 1, 2, \dots, m$) tömb segítségével tároljuk, amelyben

$$eredm_{ij} = \begin{cases} \text{az a lépésszám mellyel az } (i, j) \text{ helyre léptünk} \\ 0, \text{ ha az útvonal nem halad át az } (i, j) \text{ helyen} \end{cases}$$

Egy bizonyos helyről négy irányba léphetünk.

Algoritmus *Út*(i, j, lépés):

Ha ($L_{ij} = 1$) **és** ($eredm_{ij} = 0$) **akkor**

{ próbálunk az (i, j) helyre lépni; ha (i, j) folyosó és még nem jártunk itt }

$eredm_{ij} \leftarrow \text{lépés}$

{ az (i, j) helyre léptünk }

Ha ($i \in \{1, n\}$) **vagy** ($j \in \{1, m\}$) **akkor**

Kiír

{ kijáráshoz értünk, kiírjuk az eredménytömböt }

vége(ha)

```

Út(i-1, j, lépés+1)           { próbálunk más utat is: felfele lépünk }
Út(i, j+1, lépés+1)           { jobbra lépünk }
Út(i+1, j, lépés+1)           { lefele lépünk }
Út(i, j-1, lépés+1)           { balra lépünk }
eredmij ← 0                    { töröljük az utolsó lépést, hogy új utat választhassunk }
vége(ha)
Vége(algoritmus)

```

Ez a kód tartalmaz egy figyelemreméltó egyszerűsítést, ami a folytatási feltételeket illeti. Nem ellenőriztük azt, hogy kiléptünk-e a labirintusból, mivel a hívás előtt (a labirintus beolvasása után) az L tömböt körülvettük egy 0-ból álló kerettel. Így az algoritmus gyorsabbá válik. Az algoritmust a kiindulási hely koordinátáira (i, j) és 0 lépésszámmra hívjuk meg.

Második megoldás

A következő változatban az $Út$ alprogramban a *lépés* pillanatban megpróbálunk az (i, j) helyről az $(új_i, új_j)$ helyre lépni. Ezeket két konstans tömb (x, y) segítségével állapítjuk meg úgy, hogy ezek a négy szomszédos hely koordinátáit adják meg. Ezeknek a tömböknek az értékei: $x = (-1, 0, 1, 0)$, $y = (0, 1, 0, -1)$.

```

Algoritmus Út(i, j, lépés):
Minden irány=1,4 végezd el:           { kiválasztunk egy irányt }
    új_i ← i + xirány                 { (új_i, új_j) az új koordináták }
    új_j ← j + yirány
    Ha (új_i ∈ {1, 2, ..., n}) és (új_j ∈ {1, 2, ..., m}) akkor
        Ha (Lúj_i, új_j = 1) és (eredmúj_i, új_j = 0) akkor
            eredmúj_i, új_j ← lépés           { az (új_i, új_j) helyre lépünk }
            Ha (új_i ∈ {1, n}) vagy (új_j ∈ {1, m}) akkor
                Kiír                       { kiléptünk a labirintus szélén }
            vége(ha)
            Út(új_i, új_j, lépés+1)
            eredmúj_i, új_j ← 0               { lemondunk az utolsó lépésről }
        vége(ha)
    vége(ha)
vége(minden)
Vége(algoritmus)

```

Azt várnánk, hogy az algoritmus a *Ha* utasítás különben ágán hívja meg önmagát. Ha így járnánk el, elvesztenénk azokat az eredményeket, amelyeknek esetében a labirintus szélén tovább lehet menni, és a kilépés egy másik pontban is lehetséges.

Az algoritmusban nem vettük körül a labirintust az előbbi változat megoldásában említett kerettel. Ennek következtében szükség volt ellenőrizni, hogy az új hely, ahova lépni akarunk a labirintuson belül van-e.

Az előbbi algoritmust az $Út(i, j, 2)$ alakban hívjuk meg, de a hívás előtt $eredm_{ij} \leftarrow 1$, ahol (i, j) a kiindulási hely.

Általánosítva az előbbi feladatban használt rekurzív algoritmust, amely a visszalépéses keresés módosított változata, észrevevesszük, hogy mivel az előrehaladás egy kétdimenziós tömbben történik, az alprogramnak mindig két paramétere lesz (i, j) , amelyek annak a helynek a koordinátái, ahova lépni készülünk. Mivel általában az utat is meghatározzuk, szükséges a *lépés* paraméter is.

6. Az oszd meg és uralkodj módszer (divide et impera)

Az oszd meg és uralkodj (*divide et impera*) módszer alkalmazása akkor ajánlott, amikor a feladatot fel lehet bontani egymástól független részfeladatokra, amelyeket az eredeti feladathoz hasonlóan oldunk meg, de kisebb méretű adathalmaz esetében.

Az eredeti feladatot felbontjuk egymástól független részfeladatokra, amelyek az eredetihez hasonlóak, de kisebb adathalmazzal definiáltak. A részfeladatokkal hasonlóan járunk el és a felbontást akkor állítjuk le, amikor a feladat megoldása a lehető legjobban leegyszerűsödött.

Megoldjuk a maximálisan leegyszerűsített feladatot.

A részfeladatok eredményeiből fokozatosan felépítjük a következő méretű feladat eredményeit, ezek összerakása által. Az utolsó összerakás az eredeti feladat végeredményét adja meg.

Mivel a részfeladatok csak méreteikben különböznek az eredeti feladattól, a divide et impera módszert a legkézenfekvőbben rekurzívan írjuk le. A felbontás megtörténik a rekurzióba való belépéskor, a részeredmények összerakása pedig a kilépéskor.

6.1. Az oszd meg és uralkodj módszer általános bemutatása

A $DivImp(bal, jobb, eredm)$ algoritmus az $a_1, a_2, \dots, a_n$ sorozatot dolgozza fel, tehát $DivImp(1, n, eredm)$ alakban hívjuk meg először. Formális paraméterei a *bal* és a *jobb* (az aktuális részsorozat bal és jobb indexe), valamint *eredm*, amelyben a végeredményt továbbítjuk.

Algoritmus $DivImp(bal, jobb, eredm)$:

Ha $jobb - bal < \varepsilon$ **akkor** { ha a feladat maximálisan egyszerű }
{ kiszámítjuk az egyszerű feladat eredm eredményét }

$Megold(bal, jobb, eredm)$

különb

$Feloszt(bal, jobb, közép)$ { kiszámítjuk a közép indexet, ahol felosztjuk a sorozatot }

$DivImp(bal, közép, eredm1)$ { megoldjuk a feladatot a bal részsorozat esetében }

$DivImp(közép+1, jobb, eredm2)$ { megoldjuk a feladatot a jobb részsorozat esetében }

$Összerak(eredm1, eredm2, eredm)$ { összerakjuk a részeredményeket }

vége(ha)

Vége(algoritmus)

Az oszd meg és uralkodj stratégiát – természetesen – lehet iteratívan is implementálni. Az iteratív algoritmusok mindig gyorsabbak lesznek. A rekurzív változat előnye viszont az átláthatóságában és az egyszerűségében rejlik.

6.2. Megoldott feladatok

6.2.1. Szorzat

Számítsuk ki n valós szám szorzatát oszd meg és uralkodj módszerrel! Egy adott pillanatban csak egy szorzást végezzünk!

Megoldás

Mivel egy adott pillanatban, egy adott művelettel, csak két szám szorzatát tudjuk kiszámítani, a szorzatot részsorzatokra bontjuk. Ezt úgy valósítjuk meg, hogy a szorzótényezőket két csoportra osztjuk, kiszámítjuk egy-egy csoport szorzatát, majd a két csoport kiszámított

szorzatát összeszorozzuk. Ezt a felbontást addig lehet újra, meg újra elvégezni, amíg egy csoport legtöbb két szorzótényezőből nem áll.

A $Szorzat(x_1, \dots, x_n)$ részfeladat általános alakja: $Szorzat(x_{bal}, \dots, x_{jobb})$. Minden részfeladat más-más szorzatot számol ki, tehát a feladatok függetlenek egymástól. Mivel előnyösebb, ha egy szorzatot egy függvénnyel számolunk ki és nem egy eljárással, a $DivImp(bal, jobb, eredm)$ algoritmust a következőképp írjuk át:

```

Algoritmus Szorzat(bal , jobb) :                               { függvény típusú algoritmus }
  Ha jobb = bal akkor                                           { Bemeneti adatok: bal, jobb. Kimeneti adat: Szorzat }
    Szorzat  $\leftarrow$   $x_{bal}$                                        { a részsorozat egy elemből áll }
  különben
    Ha jobb - bal = 1 akkor
      Szorzat  $\leftarrow$   $x_{bal} * x_{jobb}$                              { a részsorozat két elemű }
    különben                                                     { felbontjuk a Szorzat(bal, ..., jobb) feladatot }
      közepe  $\leftarrow$   $[(bal+jobb)/2]$ 
      p1  $\leftarrow$  Szorzat(bal, közepe)
      p2  $\leftarrow$  Szorzat(közepe+1, jobb)
      Szorzat  $\leftarrow$  p1 * p2                                     { összerakjuk a részeredményeket }
    vége(ha)
  vége(ha)
Vége(algoritmus)

```

6.2.2. Minimumszámolás

Állapítsuk meg n egész szám közül a legkisebbet!

```

Algoritmus Minimum(bal, jobb) :                               { függvény típusú algoritmus }
                                           { Bemeneti adatok: bal, jobb. Kimeneti adat: Minimum }
  Ha jobb = bal akkor                                           { a részsorozat egy elemből áll }
    Minimum  $\leftarrow$   $x_{bal}$ 
  különben
    Ha jobb - bal = 1 akkor                                       { a részsorozat két elemből áll }
      Ha  $x_{bal} < x_{jobb}$  akkor
        Minimum  $\leftarrow$   $x_{bal}$ 
      különben
        Minimum  $\leftarrow$   $x_{jobb}$ 
      vége(ha)
    különben                                                     { felbontjuk a Minimum(bal, jobb) feladatot részfeladatokra: }
      közepe  $\leftarrow$   $[(bal+jobb)/2]$ 
      min1  $\leftarrow$  Minimum(bal, közepe)
      min2  $\leftarrow$  Minimum(közepe+1, jobb)
      Ha min1 < min2 akkor
        Minimum  $\leftarrow$  min1                                     { összerakjuk a részeredményeket }
      különben
        Minimum  $\leftarrow$  min2
      vége(ha)
    vége(ha)
  vége(ha)
Vége(algoritmus)

```

6.2.3. Bináris keresés

Adott egy n egész számból álló, növekvően rendezett sorozat. Állapítsuk meg egy adott szám helyét a sorozatban! Ha az illető szám nem található meg, a sorszámnak megfelelő paraméter értéke legyen 0.

Megoldás

Mivel egy bizonyos elemet keresünk, amelynek a helye ismeretlen, az $x_1 < x_2 < \dots < x_n$ sorozat közepén fogjuk először keresni. A következő esetek fordulhatnak elő:

1. $keresett = x_{közép} \Rightarrow$ keresett a sorban a *közép* helyen található;
2. $keresett < x_{közép} \Rightarrow$ mivel a sorozat rendezett, a keresett számot a sorozat első ($x_1, \dots, x_{közép-1}$) felében keressük tovább;
3. $keresett > x_{közép} \Rightarrow$ a keresett számot a sorozat második ($x_{közép+1}, \dots, x_n$) felében keressük tovább.

Következésképpen, ahelyett, hogy a keresett elem megkeresése két részfeladatra bomlana, átalakul egyetlen feladattá: keressük az elemet vagy az x_{bal} , ..., $x_{közép-1}$ sorozatban, vagy az $x_{közép+1}$, ..., x_{jobb} sorozatban. Itt nincs szükség a divide et impera harmadik lépésére (a részeredmények összerakására).

```

Algoritmus Bin_keres( $x$ , bal, jobb, keresett, közép):
    { Bemeneti adatok:  $x$ , bal, jobb, keresett. Kimeneti adat: közép }

    Ha bal > jobb akkor
        közép  $\leftarrow 0$  { keresett nincs a sorozatban }
    különben
        közép  $\leftarrow \lfloor (bal+jobb)/2 \rfloor$ 
        Ha keresett <  $x_{közép}$  akkor
            Bin_keres( $x$ , bal, közép-1, keresett, közép)
        különben
            Ha keresett >  $x_{közép}$  akkor
                Bin_keres( $x$ , közép+1, jobb, keresett, közép)
            vége(ha)
        vége(ha)
    vége(ha)
Vége(algoritmus)
    { ha keresett =  $x_{közép}$  megvan a pozíció }

```

E feladat esetében is létezik egy iteratív megoldás, amely a végrehajtás idejét tekintve hatékonyabb:

```

Algoritmus Bin_Keres_Iteratív(n, x, keresett, közép):
    bal ← 1
    jobb ← n
    megvan ← hamis
    Amíg nem megvan és (bal ≤ jobb) végezd el:
        közép ← [(bal+jobb)/2]
        Ha xközép = keresett akkor
            megvan ← igaz                                { közép tartalmazza a keresett helyét }
        különben
            Ha xközép > keresett akkor
                jobb ← közép - 1
            különben
                bal ← közép + 1
    vége(ha)
    vége(ha)
    vége(amíg)

```

```

Ha nem megvan akkor
    közép ← 0
vége(ha)
Vége(algoritmus)

```

{ ha közép értéke 0 $\Rightarrow$ keresett nem található }

6.2.4. Összefésülésen alapuló rendezés (*MergeSort*)

Rendezzünk növekvő sorrendbe egy egész számokból álló sorozatot összefésüléssel! (Ha két rendezett sorozatból úgy állítunk elő egy harmadikat, hogy ez utóbbi úgyszintén rendezett, összefésülésről beszélünk. De itt nem két rendezett sorozatból kell egy harmadik, ugyancsak rendezettet előállítanunk, hanem egyetlen sorozatot kell rendeznünk.)

Megoldás

Az adott sorozatot két részre osztjuk, abból a célból, hogy rendezhessük. De ezeket újból felosztjuk, amíg a kapott tömb, amelyet rendezni kell, csak egy elemből áll. Az egyelemű tömbök, természetesen rendezettek és megkezdődhet a tulajdonképpeni összefésülés.

Algoritmus Összefésül(bal, közép, jobb):

```

Minden i=bal,közép végezd el:
    ai ← xi
vége(minden)
Minden i=közép+1,jobb végezd el:
    bi ← xi
vége(minden)
aközép+1 ← végtelen
bjobb+1 ← végtelen
i ← bal
j ← közép + 1
Minden k=bal,jobb végezd el:
    Ha ai < bj akkor
        xk ← ai
        i ← i + 1
    különben
        xk ← bj
        j ← j + 1
vége(ha)
vége(minden)
Vége(algoritmus)

```

{ strázsák }

Algoritmus Rendez(bal, jobb):

```

Ha bal < jobb akkor
    közép ← [(bal+jobb)/2]
    Rendez(bal, közép)
    Rendez(közép+1, jobb)
    Összefésül(bal, közép, jobb)
vége(ha)
Vége(algoritmus)

```

Az *Összefésül(bal, közép, jobb)* algoritmus eredménye az $x_{bal}, \dots, x_{jobb}$ rendezett sorozat, amelybe tulajdonképpen ugyanazon sorozat két részsorozatát, az $x_{bal}, \dots, x_{közép}$ és az $x_{közép+1}, \dots, x_{jobb}$ részsorozatokat fésültük össze. Ezzel magyarázható annak a szükségessége, hogy az összefésülendő sorozatokat átmásoltuk az a illetve a b sorozatokba. A hívó programegységben a *Rendez(1, n)* algoritmust hívjuk.

6.2.5. Gyorsrendezés (*QuickSort*)

Fölhasználva a *quicksort* algoritmust, rendezzünk növekvő sorrendbe n egész számot! A gyorsrendezés az oszd meg és uralkodj módszeren alapszik, mivel az eredeti sorozatot úgy rendezi, hogy két rendezendő részsorozatra bontja.

Megoldás

A részsorozatok rendezése egymástól függetlenül történik. A részeredmények összerakása ebből az algoritmusból is hiányzik (mint a bináris keresésből).

Amikor az $x_1, \dots, x_n$ sorozatot készülünk rendezni, előbb előkészítünk két részsorozatot ($x_1, \dots, x_{m-1}$ és $x_{m+1}, \dots, x_n$) úgy, hogy az $x_1, \dots, x_{m-1}$ részsorozat elemei kisebbek legyenek, mint az $x_{m+1}, \dots, x_n$ részsorozat elemei. Köztük található az x_m , amely nagyobb mint az $x_1, \dots, x_{m-1}$ részsorozat bármely eleme, és kisebb mint az $x_{m+1}, \dots, x_n$ részsorozat összes eleme.

Azt az elemet, amely meghatározza a helyet, ahol az adott tömb két részre oszlik, strázsának (őrszem) nevezzük. Ennek a helynek a meghatározása kulcskérdés az algoritmus végrehajtása során. A strázsra m helyét úgy határozzuk meg, hogy az $x_1, \dots, x_m$ tömbben legyenek azok az elemek, amelyek kisebbek, mint a strázsra és az $x_{m+1}, \dots, x_n$ tömbben azok, amelyek nagyobbak annál.

Gyakran választjuk strázsának az x_1 -et. Elindulunk a tömb két szélső elemétől és felcseréljük egymás közt azokat az elemeket, amelyek nagyobbak, mint a strázsza (és a tömb első részében találhatók) azokkal, amelyek kisebbek, mint a strázsza (és a tömb második részében találhatók). Ahol ez a bejárás véget ér, ott fogjuk két részre osztani a tömböt. Egy ilyen feldolgozás során egy elem a végleges helyére kerül.

A részsorozatok rendezése érdekében ezeket hasonló módon bontjuk fel. A felbontás addig folytatódik, amíg a rendezendő részsorozat hossza 1 lesz.

Algorithmus QuickSort (bal, jobb):

Ha $\text{bal} < \text{jobb}$ **akkor** $\{ \text{meghatározzuk azt az } m \text{ helyet, ahol a sorozatot} \}$
 $\{ \text{két részre bontjuk, miközben egy elem } (x_m) \text{ a végleges helyére kerül} \}$

```
m ← Strázsa helye(bal, jobb)
```

```
QuickSort (bal, m)
```

$\{ \text{hasonlóan járunk el az } (x_{bal}, ..., x_m) \text{ részsorozattal} \}$

```
QuickSort(m+1, j+1)
```

$$\{ \text{valamint az } (x_{m+1}, \dots, x_{jobb}) \}$$

vége (ha)

Vége(algoritmus)

Látható, hogy a rekurzív hívásoknak megfelelően, az algoritmus meghívja önmagát egy bal meg egy jobb részsorozat rendezése érdekében. De hol a rendezés, hiszen ez az algoritmus nem tartalmaz összehasonlításokat és felcseréléseket? Ezeket aközben végezzük, miközben keressük a strázsa m helyét:

Algoritmus Strázsa helye(bal, jobb):

strázsa $\leftarrow$ x_{bal} { *Bemeneti adatok: bal, jobb. Kimeneti adat: Strázsa helye* }

$$i \leftarrow \text{bal}-1$$
$$j \leftarrow j_{\text{obb}} + 1$$

$\{ \text{megkeressük azt a } j\text{-t, amelyre } \text{bal} \leq j < \text{jobb} \}$

Ismételd

Ismétekd

$\{ \text{megkeressük azt a } j\text{-t (jobbról balra), amelyre } x_j < \text{strázs} \}$

$$j \leftarrow j - 1$$

ameddig $x_i \leq \text{strázsa}$

Ismétekd

$\{ \text{megkeressük azt az } i\text{-t (balról jobbra), amelyre } x_i > \text{strázs} \}$

$$i \leftarrow i + 1$$

ameddig $x_i \geq \text{strázsa}$

Ha $i < j$ akkor

```

       $x_i \leftrightarrow x_j$                                 { felcseréljük ezt a két nem megfelelő tulajdonságú elemet }
vége(ha)
ameddig  $i \geq j$                                 { addig folytatjuk a keresést és felcserélést, amíg  $i$  kisebb, mint  $j$  }
  Strázsa_helye  $\leftarrow j$                                 { megtaláltuk az új strázsa helyét }
Vége(algoritmus)

```

Megjegyzés

Ez az algoritmus főleg abban az esetben gyors, amikor a tömb elemei nem rendezettek!

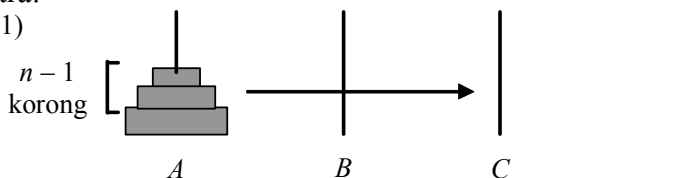
6.2.6. Hanoi tornyok

Adva van három rúd A, B, C; az elsőre fel van fűzve n darab, különböző átmérőjű korong úgy, hogy a korongok az átmérőjük csökkenő sorrendjében helyezkednek el egymás fölött. A másik két rúd üres. Írjuk ki minden lehetséges módját annak, ahogyan a korongokat átköltöztethetjük az A rúdról a B-re, ugyanolyan sorrendben, ahogyan az A-n helyezkedtek el. Közben fel lehet használni, ideiglenesen a C rudat. Egy mozdítás csak egy korongot érinthet, és csak kisebb átmérőjű korongot helyezhetünk egy nagyobb átmérőjű korong fölé.

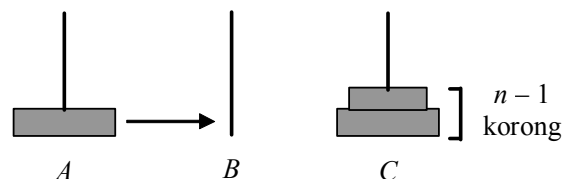
Megoldás

A módszer újból a divide et impera. Az n korong átköltöztetése az A rúdról a B-re felbontható három, ehhez hasonló feladatra:

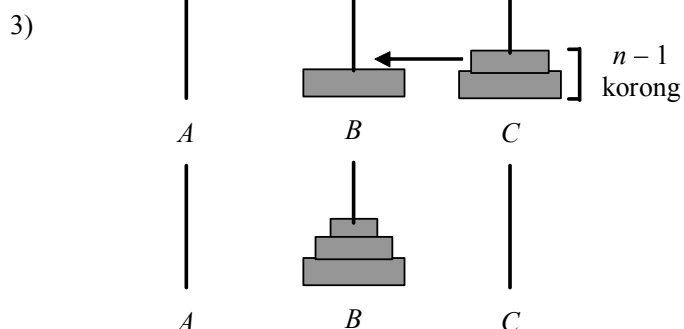
- 1) $n - 1$ korong átköltöztetése az A rúdról a C-re,



- 2) a megmaradt korong áthelyezése B-re,



- 3) $n - 1$ korong áthelyezése C-ről B-re.



A három részfeladat méretét a költöztetendő korongok száma határozza meg: $n - 1$, 1 és $n - 1$. A részfeladatok függetlenek mivel az eredeti rudak konfigurációi, valamint az időközben váltakozva ideiglenesnek használt rudaké különbözők. A feladat felbontása ugyanígy folytatódik, míg olyan részfeladathoz nem érünk, amelynek mérete 1. Ennek megoldása egyetlen korong költöztetését jelenti.

A részeredmények összerakása ebben az esetben is hiányzik.

Algoritmus Hanoi(n, A, B, C) :

Ha $n \geq 1$ **akkor**

Hanoi_1($n-1, A, C, B$)

Tedd a korongot A-ról B-re

```

    Hanoi_1(n-1,C,B,A)
  vége(ha)
Vége(algoritmus)

```

Ennek az algoritmusnak a hívása $\text{Hanoi}(n, A, B, C)$ alakú, ahol A, B, C a három rudat jelképezi, és ha a hívó programegységben az aktuális paraméterek értékei 'A', 'B', 'C', akkor a *Tedd a korongot A-ról B-re* egy egyszerű kiírás: **Ki:** A, '-', B.

7. Mohó algoritmusok (greedy módszer)

A greedy módszert (mohó algoritmusokat) optimum-számításokra használjuk. E feladatok eredményei részhalmazai vagy elemei annak a Descartes-szorzatnak, amelyre a célfüggvény eléri minimumát vagy maximumát.

A mohó algoritmus mindig egyetlen eredményt határoz meg. Ezt az eredményt fokozatosan építjük fel: a feladatokban általában adott egy L halmaz, amelynek meg kell határoznunk egy M részhalmazát, amely megfelel bizonyos követelményeknek (T tulajdonságnak), és amely általában a végeredmény. Az M halmaz eredetileg az üres halmaz. Ehhez, egymás után hozzáadunk L -beli elemeket, amelyeket úgy választunk ki, hogy lokális optimumot biztosítanak. Ezek az elemek azok, amelyek a legtöbbet ígérők az aktuális lépésben, és amelyek megfelelnek a feladatnak az adott pillanatban.

A stratégia mohó jellegének következtében kapta ez az algoritmus a greedy (mohó) elnevezést. Mivel a stratégia egy helyi optimum kiválasztására épül, nem biztosítja a megoldás globális optimalitását, tehát nem mindig határozza meg a legjobb megoldást. Nem lehetünk biztosak a megoldásban, de ha sikerül *bebizonyítani*, hogy az adott feladat esetében a mohó algoritmus optimumot határoz meg, akkor biztonságosan alkalmazható. Ha viszont olyan feladatunk van, amelynek pontos megoldását csak exponenciális algoritmussal tudjuk megadni, sok esetben akkor is alkalmazható, de természetesen számításba vesszük, hogy az eredmény közelítő érték. Ilyenkor *heurisztikus* mohó algoritmusról beszélünk.

Legyen az L halmaz az $\{a_1, a_2, \dots, a_n\}$ sorozat és T egy tulajdonság, amelyet az L részhalmazaira definiáltunk: $T: T(L) \rightarrow \{0, 1\}$, ahol $T(\emptyset) = 1$ (igaz, vagyis teljesül T), ha $T(X)$, akkor $\Rightarrow T(Y)$, bármely $Y \subset X$ részhalmaz esetében. Egy $S \subset L$ részhalmazt eredménynek nevezünk, ha $T(S) = 1$. Minden lehetséges eredményből azt szeretnénk kiválasztani, amely optimalizálja a $T: T(L) \rightarrow R$ adott függvényt. A mohó algoritmus nem generál minden lehetséges részhalmazt (ami exponenciális végrehajtási időhöz vezetne), hanem megpróbál közvetlenül az optimális megoldás felé haladni.

A módszer egyszerű, a programok gyorsak, még nagyméretű adatszerkezetek esetében is. Az egyszerűség abban áll, hogy minden pillanatban, csak az adott kontextusnak megfelelő részfeladatot tekintjük. A módszer különbözik a backtracking (visszalépéses keresés) módszertől mivel, ha egy elemről kiderül, hogy hiába volt sokat ígérő, akkor nem kerül be a megoldásba és soha nem térünk vissza ehhez az elemhez. Fordítva, ha egy elem bekerült egy adott pillanatban egy megoldásba, nem fogjuk kivenni onnan.

7.1. A mohó algoritmus általános bemutatása

A módszer általános alakját két változatban ismertetjük. (A feladat megoldását az M halmaz tartalmazza, a megoldásokat az L – lehetséges megoldások halmazából – válogatjuk):

Algoritmus Greedy_1 (L, M) :

$M \leftarrow \emptyset$

Amíg M nem megoldás és $L \neq \emptyset$ **végezd el:**

Választ (L, x) { kiválasztjuk a legtöbbet ígérő elemet L -ből }

$L \leftarrow L \setminus \{x\}$ { töröljük a legtöbbet ígérő elemet L -ből }

Ha $T(M \cup \{x\}) = 1$ **akkor** { ha lehetséges }

$M \leftarrow M \cup \{x\}$ { ezt hozzáadjuk M -hez }

vége(ha)

vége(amíg)

Vége(algoritmus)

Megjegyzések

1. Ha a kiválasztott elemet töröljük L -ből, akkor biztosítottuk az algoritmus számára, hogy L minden elemét csak egyszer dolgozzuk fel (töröljük, függetlenül attól, hogy betesszük az eredménybe vagy sem).
2. Mivel a bemeneti adatoktól függően nem mindig találunk eredményt, a hívó programegységben meg kell vizsgálnunk, hogy az M halmaz valóban eredmény-e:

Az ilyen típusú feladatok megoldása során gyakran bizonyul előnyösnek, ha a tulajdonképpeni feldolgozás előtt előbb rendezzük a feldolgozandó adatokat (az L halmazt). A rendezett sorozat elemeit ($\{a_1, a_2, \dots, a_n\}$) egymás után vizsgáljuk és a követelményektől függően betesszük az eredménybe vagy sem (nincs szükség ezek törlésére L -ből, mivel egy megvizsgált elemhez nem térünk vissza). Az algoritmus ebben a változatban a következő lesz:

Algoritmus Greedy_2(n, a, M):

Feldolgoz(n, a)

{ ez a feldolgozás gyakran rendezés }

$M \leftarrow \emptyset$

$i \leftarrow 1$

Amíg M nem megoldás és $(i \leq n)$ **végezd el:**

Ha $T(M \cup \{a_i\})$ **akkor**

{ ha lehetséges }

$M \leftarrow M \cup \{a_i\}$

{ a_i -t hozzáadjuk M -hez }

vége(ha)

$i \leftarrow i + 1$

vége(amíg)

Vége(algoritmus)

A fenti algoritmusok lineárisak (eltekintve a $Választ(L, x)$ és a $Feldolgoz(n, a)$ algoritmusok bonyolultságától)! A tulajdonképpeni nehézséget a $Választ(L, x)$, valamint a $Feldolgoz(n, a)$ jelenti, mivel ezekbe „rejtjük” el a célfüggvényt.

7.2. Megoldott feladatok

7.2.1. Összeg

Adott egy n elemű, valós számokból álló sorozat. Határozzuk meg az adott sorozat azon részsorozatát, amelynek összege a lehető legnagyobb.

Megoldás

Alkalmazzuk a $Greedy_1(L, n)$ algoritmust, ahol a $Választ(L, x)$ alprogramnak megfelelően az adott sorozatból kiválasztjuk a szigorúan pozitív elemeket. Ezúttal könnyű belátni, hogy az algoritmus garantáltan maximális összegű részsorozatot határoz meg, hiszen, ha az összeghez hozzáadnánk egy negatív értéket, akkor az kisebbé válna. Ha egy 0 értékű elemet adunk az összeghez, az nem változik. Ebből az észrevételből következik, hogy, ha a sorozat tartalmaz 0 értékeket is, akkor több megoldás is létezik.

Algoritmus Összeg($n, a, k, Pozitívak$):

$k \leftarrow 0$

{ Bemeneti adatok: n, a . Kimeneti adatok: $k, Pozitívak$ }

Minden $i=1, n$ **végezd el:**

Ha $a_i > 0$ **akkor**

$k \leftarrow k + 1$

$Pozitívak_k \leftarrow a_i$

vége(ha)
vége(minden)
Vége(algoritmus)

7.2.2. Az átlagos várakozási idő minimalizálása

Egy ügyvédi irodába egyszerre érkeznek n személy, akiknek az intéznivalóit az ügyvéd ismeri, és így azt is tudja, hogy egy-egy személlyel hány percet fog eltölteni. Állapítsuk meg azt a sorrendet, amelyben fogadnia kellene a személyeket ahhoz, hogy az átlagos várakozási idő minimális legyen.

Megoldás

Az átlagos várakozási idő az n személy várakozási idejének számtani középarányosa, tehát az átlagos várakozási idő csökkentése a várakozási idők összegének csökkentését jelenti.

A minimális várakozási időösszeget a személyekkel való tárgyalási idők növekvő sorrendben való rendezése eredményezi.

Dacára annak, hogy ez természetesnek tűnik, be kell bizonyítanunk, hogy a mohó algoritmus jó megoldási módszer (...).

A mohó algoritmus alkalmazása optimális eredményt biztosít. Ahhoz, hogy minimalizáljuk az átlagos várakozási időt, minimalizálnunk kell a várakozási idők összegét. Egy személy addig várakozik, amíg az összes előtte fogadott személlyel tárgyal az ügyvéd. Ha csak két személy érkezett volna az irodába, akkor az lenne előnyösebb (az átlagos várakozási idő szempontjából), ha előbb a kevesebb időt igénylő személlyel tárgyalna az ügyvéd. Az eredmény tehát a személyek sorszámainak egy olyan permutációja, amelynek megfelelően az ügyvéd minden lépésben a legkevesebb időt igénylő személyt fogadja: $M = (k_1, k_2, \dots, k_n) \in \{(x_1, x_2, \dots, x_n) \mid x_i \in \{1, 2, \dots, n\}, x_i \neq x_j \forall i, j = 1, 2, \dots, n, i \neq j\}$.

Az L eredetileg az $\{1, 2, \dots, n\}$ halmaz. A legtöbbet ígérő x elem az L -ből annak a személynek a sorszáma, akinek a fogadási ideje minimális azok között, akik még az L -hez tartoznak. Ezt hozzáadjuk az M -hez és kizárjuk az L -ből. Az x kizárását az L -ből úgy valósítjuk meg, hogy 0 értéket másolunk rá. Minden lépésnél csak 0-tól különböző értéket választunk az L -ből.

A következő implementáció előbb inicializálja az M halmazt az $1, 2, \dots, n$ értékekkel, és növekvő sorrendbe rendezi az időket, megfelelően módosítva az M halmaz elemeit. A rendezés után: $M = k_1, k_2, \dots, k_n$ és $t_1 \leq t_2 \leq \dots \leq t_n$. A kiírást az M halmazban található indexpermutáció alapján végezzük.

Algoritmus Sorrend($n, t, M, \text{átlag}$):

{ Bemeneti adatok: n, t, M . Kimeneti adatok: átlag, M }

Minden $i=1, n$ **végezd el:**

$M_i \leftarrow i$

vége(minden) { növekvően rendezzük a t sorozatot és módosítjuk az M -et is }

Növekvő_sorrendbe_rendezés(n, M, t)

$vi_min \leftarrow 0$

$vi \leftarrow 0$

Minden $i=1, n-1$ **végezd el:**

$vi \leftarrow vi + t_i$

{ a t sorozat már növekvően rendezett }

$vi_min \leftarrow vi_min + vi$

vége(minden)

átlag $\leftarrow v_{i_min} / n$
Vége(algoritmus)

7.2.3. Buszmegállók

Egy közszállítási vállalat olyan gyorsjáratot szeretne indítani, amely csak a város főutcáján közlekedne, és a már létező n megálló közül használna néhányat. Ezeket a megállókát úgy kell kiválasztanunk, hogy két megálló között a távolság legkevesebb x méter legyen (gyorsjáratról van szó), és a megállók száma legyen a lehető legnagyobb (minél több utas használhassa). Adott a főutcán már meglevő egymás után található megállók közti távolságok sorozata.

Megoldás

Az L halmazt a létező megállók sorszámai alkotják: $L = \{1, 2, \dots, n\}$. Ismerjük az n megálló közötti $n - 1$ távolságot: $a_1, a_2, \dots, a_{n-1}$.

Meg kell határoznunk azt a maximális elemszámú $M \subseteq L$ részhalmazt ($M = \{i_1, i_2, \dots, i_k\}$), amelyben a sorszárok növekvő sorrendben követik egymást (a főutcán található megállóknak egymás utáni sorszámaik vannak), és amelynek megfelelően bármely két kiválasztott megálló között a távolság legkevesebb x méter ($a_{ij+1} - a_{ij} \geq x, j = 1, 2, \dots, k - 1$).

Algoritmus Megállók(n, a, M):

$i \leftarrow 1$ { Bemeneti adatok: n, a . Kimeneti adat: M }
 $M_1 \leftarrow 1$ { az eredménybe betett utolsó megállótól mért távolság }

táv_az_utolsótól $\leftarrow 0$

Minden $j=2, n$ **végezd el:**

Ha $a_{j-1} + \text{táv_az_utolsótól} \geq x$ **akkor**

$i \leftarrow i + 1$

$M_i \leftarrow j$

táv_az_utolsótól $\leftarrow 0$

különben

táv_az_utolsótól $\leftarrow \text{táv_az_utolsótól} + a_{j-1}$

vége(ha)

vége(minden)

Vége(algoritmus)

Látható, hogy az első megállót betettük a megoldásba, majd megkerestük azt a megállót, amelyik megfelelő távol található az elsőtől. Ha találtunk ilyent, betettük a megoldásba. Ezt addig folytattuk, amíg bejártuk az összes, már létező megállót.

7.2.4. Autó bérbeadása

Egy szállítási vállalat autókat kölcsönöz. Egy bizonyos jármű iránt igen nagy az érdeklődés, ezért az igényeket egy évre előre jegyzik. Az igényt két számmal jelöljük, amelyek az év azon napjainak sorszárait jelölik, amellyel kezdődően, illetve végződően igénylik az illető autót.

Állapítsuk meg a bérbeadást úgy, hogy a lehető legtöbb személyt szolgáljuk ki. Adott a személyek száma n , ($n \leq 100$) és az igényelt intervallumok ($a_i, b_i, i = 1, 2, \dots, n, a_i < b_i \leq 365$). Írjuk ki azt a számot, amely a lehetséges legnagyobb az igénylő személyek számából és a bérbeadási időintervallumokat.

A következő algoritmusban az L halmaz: $\{2, 3, \dots, n\}$ mivel M kezdőértéke $\{1\}$ (az első igény – a minimális b_1 – mindig része lesz a megoldásnak, amelyet a greedy stratégia biztosít). Az L halmazt az algoritmus *Minden* típusú struktúrával számítja ki, amelyben sorra veszi a b_i szerint rendezett igényléseket.

Algoritmus Autó_kölcsönzés(n, a, b, max, M):
 Növekvő_sorrendbe_rendezés(n, a, b)
 $M_1 \leftarrow 1$ { *Bemeneti adatok: n, a, b . Kimeneti adat: max, M* }
 $\text{max} \leftarrow 1$
Minden $i=2, n$ **végezd el:**
 $j \leftarrow M_{\text{max}}$
 Ha $a_i > b_j$ **akkor**
 $\text{max} \leftarrow \text{max} + 1$
 $M_{\text{max}} \leftarrow i$
 vége(ha)
vége(minden)
Vége(algoritmus)

7.2.5. Hátizsák

Egy tolvaj betört egy hentesüzletbe, ahol n áru közül válogat. Minden árunak ismeri a súlyát és az értékét. Mivel a hátizsákjába legtöbb S súly fér, szeretne úgy válogatni, hogy a nyeresége maximális legyen. Ha egy áru nem fér be egészében a hátizsákba, a tolvaj levághat belőle egy akkora darabot, amekkora befér a hátizsákba, de ebben az esetben az áru értéke a súlyával arányosan csökken.

Megoldás

A feladat a szakirodalomban „töredékes hátizsák” vagy „folytonos hátizsák” elnevezés alatt ismeretes.

Észrevehető, hogy mivel meg volt engedve, hogy levághatunk az árukból, a hátizsák teljesen megtölthető, és ha minden lépésben azt az árut választjuk, amelynek az *érték/súly* aránya maximális, akkor a hátizsákba csomagolt árumennyiség összértéke is maximális lesz.

Bevezetjük a következő jelöléseket: Az eredmény az $x = (x_1, \dots, x_n)$ sorozat lesz, ahol $x_i \in [0, 1]$, $i = 1, 2, \dots, n$ azt fejezi ki, hogy az i -edik árunak mekkora darabját csomagoljuk be. Ezen kívül: $\text{súly}_1 \cdot x_1 + \text{súly}_2 \cdot x_2 + \dots + \text{súly}_n \cdot x_n \leq S$.

Az optimális eredmény az, amely maximalizálja az $f(x) = \text{érték}_1 \cdot x_1 + \text{érték}_2 \cdot x_2 + \dots + \text{érték}_n \cdot x_n$ függvényt.

Abban a sajátos esetben, amikor minden árut be lehet csomagolni a hátizsákba, $x = (1, 1, \dots, 1)$. Ezért a továbbiakban feltételezzük, hogy $\text{súly}_1 + \dots + \text{súly}_n > S$.

A greedy stratégiának megfelelően, az árukat az *érték/súly* arány szerint csökkenő sorrendbe rendezzük. Az árukat ebben a sorrendben csomagoljuk a hátizsákba, amíg az meg nem telik. Ha egy áru nem fér a hátizsákba, levágunk belőle egy akkora darabot, ami befér.

Algoritmus Hátizsák($n, S, \text{súly}, \text{érték}, \text{sorszám}, x$):
 Csökkenő($n, \text{súly}, \text{érték}, \text{sorszám}$)
 $\text{Hely} \leftarrow S$ { *Hely a hátizsákban még szabad helyet jelöli* }
 $i \leftarrow 1$
Amíg $(i \leq n)$ **és** $(\text{Hely} > 0)$ **végezd el:**
 Ha $\text{súly}_i \leq \text{Hely}$ **akkor**
 $x_i \leftarrow 1$
 $\text{Hely} \leftarrow \text{Hely} - \text{súly}_i$
 különben
 $x_i \leftarrow \text{Hely} / \text{súly}_i$
 $\text{Hely} \leftarrow 0$
 Minden $j=i+1, n$ **végezd el:**

```
       $x_j \leftarrow 0$   
    vége(minden)  
  vége(ha)  
   $i \leftarrow i + 1$   
vége(amíg)  
Vége(algoritmus)
```

Az algoritmus végrehajtásának eredménye az x sorozat: $x = (1, \dots, 1, x_j, 0, \dots, 0)$ ahol $x_j \in [0, 1)$. Ennek alapján kiírhatjuk a becsomagolt áruk sorszámainak (vigyázzunk, hogy az eredeti sorszámozat írjuk ki) és a hátizsák tartalmának értékét.

Be kell bizonyítanunk, hogy az algoritmus optimális eredményt határoz meg.

II. Objektumorientált programozás

1. Objektumorientált fogalmak

1.1. Adatvédelem moduláris programozással

Az eljárásközpontú programozás keretében a kódot igyekezünk eljárásokra és függvényekre bontani. A C és a C++ programozási nyelvekben az eljárásokat és függvényeket egyetlen névvel jellemezzük. Mindkét esetben *függvényekről* beszélünk, de megkülönböztetünk olyan függvényeket, amelyek visszatérítenek egy értéket és olyanokat, amelyek nem. Az eljárásoknak azok a függvények felelnek meg, amelyek nem térítenek vissza semmit. Ebben az esetben a *void* kulcsszóval jelezzük a visszaadandó érték típusának a hiányát.

A nagyobb alkalmazások írásakor felmerül annak a szükségessége, hogy az általunk használt adatok védelmét megvalósítsuk. Ez azt jelentené, hogy csak a függvényeknek egy részével lehessen hozzáférni az adatokhoz. Azért van erre szükség, mert ez által jelentősen csökken a hibalehetőségek száma. Az adatok és a rájuk vonatkozó függvények egyetlen egységet fognak képezni. Így az adatok módosítása csak ezekkel a függvényekkel lesz megvalósítható, másokkal nem.

Az adatok védelmére már a C programozási nyelv is lehetőséget teremtett a *moduláris programozás* által. Ha egy állomány globális hatókörében, tehát a függvényeken, osztályokon és névterekben kívül, egy statikus változót vezetünk be, akkor ezt a változót a deklaráció helyétől az illető állomány (modul) végéig bármely függvényben használhatjuk. Ezzel ellentétben viszont más állományban még akkor sem tudunk hivatkozni az illető változóra, ha abban egy *extern* típusú deklarációt helyezünk el.

A továbbiakban egy olyan példát ismertetünk, amely az adatok védelmét a moduláris programozás segítségével teszi lehetővé. Egy egész elemekből álló vektorokra vonatkozó modult hozunk létre. A vektor elemeit egy *int* típusra hivatkozó mutató segítségével tároljuk. Meg kell adnunk a vektor méretét is, tehát az elemek számát. Ezt a két adatot a függvényeken kívül deklarált statikus változókkal vezetjük be. Az adatok feldolgozását a következő négy függvénnyel végezzük: *epit*, *felszabadit*, *negyzetre* és *kiir*. Az első függvény egy egész elemekből álló tömb és egy egész szám (a méret) segítségével létrehozza a vektort. Ha a vektorra már nincs szükség, a második függvénnyel szabadíthatjuk fel a lefoglalt memóriaterületet. A *negyzetre* függvény a vektor összes elemét négyzetre emeli, és az utolsó függvény kiírja az elemeket. Az alábbi állományban mutatjuk be ennek a modulnak egy lehetséges megvalósítását.

1.1. kódszöveg. A *vektor* modul.

```

1  #include <iostream>
2  using namespace std;
3  static int* elem;
4  static int meret;
5  void epit(int* az_elem, int a_meret)
6  {
7      meret = a_meret;
8      elem = new int[meret];
9      for(int i = 0; i < meret; i++)
10         elem[i] = az_elem[i];
11 }
12 void felszabadit()
13 {
14     delete [] elem;
15 }
16 void negyzetre()
17 {
18     for(int i = 0; i < meret; i++)
19         elem[i] *= elem[i];
20 }
21 void kiir()
22 {
23     for(int i = 0; i < meret; i++)
24         cout << elem[i] << ' ';
25     cout << endl;
26 }
```

Egy külön állományba helyezzük a fő függvényt. Ez a következő lehet:

1.2. kódszöveg. A fő függvényt tartalmazó állomány.

```

1 void epit( int*, int);
2 void felszabadit();
3 void negyzetre();
4 void kiir();
5 //extern int* elem;
6 void main()
7 {
8     int x[] = {1, 2, 3, 4, 5};
9     epit(x, 5);
10    negyzetre();
11    kiir();
12    felszabadit();
13    int y[] = {1, 2, 3, 4, 5, 6};
14    epit(y, 6);
15    //elem[1]=10;
16    negyzetre();
17    kiir();
18    felszabadit();
19 }
```

Végrehajtva a programot az alábbi kimenetet kapjuk:

```

1 4 9 16 25
1 4 9 16 25 36
```

A *vektor* modul függvényeinek meghívása előtt a deklarációkat elhelyeztük a fő függvényt tartalmazó állományban. A *main* függvényben előbb egy öt elemből álló *x* vektorral, majd ezt követően egy hat elemből álló *y* vektorral végeztünk műveleteket.

Hangsúlyozzuk, hogy a *vektor* modul bevezetése nem tette lehetővé azt, hogy egyszerre két vektorral tudjunk dolgozni. Például nem tudunk olyan vektorokra vonatkozó műveletet értelmezni, mint az összeadás, amelyben egyszerre több vektorra volna szükség. Figyeljük meg, hogy az *x* vektor által lefoglalt memóriaterületet fel kellett szabadítani még mielőtt az *y* vektort létrehoztuk volna. Ez egy nagy hátránya ennek a megközelítésnek, éppen ezért a következő pontban azt fogjuk vizsgálni, hogy milyen módon tudunk egy olyan saját adattípust létrehozni, amely megengedi, hogy egyszerre több példánnyal dolgozzunk. Ugyanakkor viszont nem szeretnénk lemondani a védettségről sem, és ez által jutunk el az *osztály* (§1.3) fogalmának a bevezetéséhez.

Vegyük észre ugyanakkor azt is, hogy a *vektor* modul valóban biztosítja az adatok védelmét. Ha a vektort az *elem* mutató segítségével direkt módon próbáljuk módosítani, a 15. sorból eltávolítva a megjegyzés jelét, akkor fordítási hibát kapunk. Ha ugyanezt megteesszük az 5. sorban, ez által elhelyezve egy *extern* típusú deklarációt a kódban, akkor ez az állomány önmagában lefordítható lesz, viszont a *szerkesztéskor* jelez hibát a rendszer. Ahhoz, hogy ez a hiba se jelenjen meg, el kell távolítanunk a *static* kulcsszót az 1.1. kódszöveg 3. sorából. Ekkor már valóban módosítható lesz az illető elem, de ez pontosan azt jelenti, hogy nincs védettség. Futtatáskor a kimenet így módosul:

```

1 4 9 16 25
1 100 9 16 25 36
```

Levonhatjuk tehát a következtetést, hogy a moduláris programozás esetén a védettséget valóban a statikus változók valósítják meg.

A moduláris programozás módszerét az adatok védelmén kívül *adatrejtésre* is használhatjuk. Ennek lényege az, hogy a felhasználó csak azt a felületet kell ismerje, amin keresztül feldolgozhatóak az adatok.

1.2. Absztrakt adattípusok

Az előző pontban egy példát adtunk a védettség megvalósítására moduláris programozással. Megállapítottuk, hogy az adatoknak és függvényeknek ilyen jellegű megadása nem tette lehetővé azt, hogy egyszerre több példánnyal, például két vektorral, dolgozzunk. Ezért szükségszerűen jelenik meg az az igény, hogy az adatokat és függvényeket, egy különálló modulhoz hasonlóan, továbbra is egyetlen egységben tároljuk, de legyen lehetőség arra is, hogy több példányt hozzunk létre.

Természetszerűen merül fel az a lehetőség, hogy a hagyományos struktúra rendeltetésének a kiterjesztése által próbáljuk meg elérni a célunkat. A C++ programozási nyelvben egy struktúrán belül a hagyományos adatokon kívül elhelyezhetünk függvénydeklarációkat, illetve definíciókat is. Ilyen módon egy új típust vezetünk be, amit gyakran *absztrakt adattípusnak* (*elvont adattípusnak*, vagy *felhasználói típusnak*) nevezünk. Tekintsük az alábbi *taxi* elvont adattípusra vonatkozó forráskódot.

1.3. kódszöveg. A *Taxi* felhasználói típus.

```

1  #include <iostream>
2  using namespace std;
3  struct Taxi {
4      int fizetni;
5      int indulas_ar;
6      int menet_ar;
7      int varakozas_ar;
8      bool van_utas;
9      void Kezdes();
10     bool Beul();
11     int Kiszall();
12     void Megy(int km);
13     void All(int perc);
14 };
15 void Taxi::Kezdes()
16 {
17     indulas_ar = 10;
18     menet_ar = 10;
19     varakozas_ar = 3;
20     fizetni = 0;
21     van_utas = false;
22 }
23 bool Taxi::Beul()
24 {
25     if ( van_utas ) return false;
26     van_utas = true;
27     fizetni = indulas_ar;
28     return true;
29 }
30 int Taxi::Kiszall()
31 {
32     if ( !van_utas ) return 0;
33     van_utas = false;
34     return fizetni;
35 }
36 void Taxi::Megy(int km)
37 {
38     if ( van_utas )
39         fizetni += menet_ar * km;
40 }
41 void Taxi::All(int perc)
42 {
43     if ( van_utas )
44         fizetni += varakozas_ar * perc;
45 }
46 void main()
47 {
48     Taxi t1, t2;
49     t1.Kezdes();
50     t2.Kezdes();
51     t1.Beul();
52     t1.Megy(4);

```

```

53     t2.Beul();
54     t1.All(3);
55     t2.Megy(6);
56     t1.Megy(5);
57     cout << "t1-nek fizetni: ";
58     cout << t1.Kiszall() << endl;
59     cout << "t2-nek fizetni: ";
60     // t2.fizetni = 500;
61     cout << t2.Kiszall() << endl;
62 }

```

A program kimenete a következő lesz:

```

t1-nek fizetni: 109
t2-nek fizetni: 70

```

Megjegyezzük, hogy az 1.3. kódszöveg 3-14 soraiban bevezetett struktúra az adatokon kívül függvénydeklarációkat is tartalmaz. Az elvont adattípusokon belül megadott adatokat *adattagoknak*, a függvényeket pedig *tagfüggvényeknek* nevezzük. A tagfüggvényekre az adattagokhoz hasonlóan a tagkiválasztó operátorral (a *pont* operátor), illetve a struktúra-mutató operátorral (a *->* operátor) hivatkozhatunk.

A struktúrán belül elhelyezhetünk függvénydefiníciókat is, de ez általában csak a nagyon egyszerű függvények esetén ajánlott. Ha egy függvény definíciója a struktúrán belül van, akkor *inline függvényként* kezeli a rendszer. Ha csak a függvény deklarációja kerül a struktúra belsejébe, akkor a definíciót, a névterekhez hasonló módon, úgy adjuk meg, hogy a függvény nevét a struktúra neve és a hatókör operátor előzi meg.

Az 1.3. kódszöveg fő függvényéből, illetve a program kimenetéből egyértelműen levonható az a következtetés, hogy a *Taxi* adatszerkezetnek egyszerre több példányával tudunk műveleteket végezni. Az adatok védelme azonban nem valósul meg ebben az esetben. Meggyőződhetünk erről, ha a 60. sorból eltávolítjuk a megjegyzés jelét, és úgy fordítjuk le a kódot. A kimenet a következő lesz:

```

t1-nek fizetni: 109
t2-nek fizetni: 500

```

Tehát a fizetendő összeg módosítható direkt módon, függvénymeghívás nélkül. Ez azt jelenti, hogy nincs biztosítva az adatok védelme. A következő pontban azt vizsgáljuk meg, hogy az absztrakt adattípus fogalma hogyan terjeszthető ki úgy, hogy lehetőséget teremtsen az adatvédelemre.

1.3. Osztálydeklaráció

Az előző pontban megállapítottuk, hogy a felhasználói típus bevezetése lehetővé teszi azt, hogy az adatszerkezetnek egyszerre több példányával tudjunk műveleteket végezni. Ugyanakkor, az adatvédelem nem valósul meg egyszerűen az által, hogy adatokat és függvényeket egyetlen struktúra részeként adunk meg. Annak érdekében, hogy ezt a hiányosságot kiküszöböljék, bevezették az *osztály* fogalmát.

Az *osztály* egy olyan absztrakt adattípus, amely lehetőséget teremt az adattagok és tagfüggvények védelmére. Az *osztálydeklaráció* az előző pontban ismertetett felhasználói típus bevezetéséhez hasonló, azzal a különbséggel, hogy a *struct* kulcsszót a *class* (osztály) fogja helyettesíteni. Az osztály tagjaira való hivatkozás a tagkiválasztó operátorral, illetve a struktúra-mutató operátorral történhet, ugyanúgy mint az egyszerű struktúrák, vagy az előző pontban ismertetett elvont adattípusok esetén. Ezt a kérdést az §1.4. pontban tárgyaljuk részletesebben.

Mivel az osztály egy felhasználói típus, fontos különbséget tennünk maga az osztály, és ennek példányai között. Egy osztály példányait *objektumoknak* nevezzük. Tehát az objektum általában egy változó, amelynek a típusát az osztálya határozza meg.

Azok a függvénydefiníciók, amelyek az osztályon belül vannak *inline függvény* eredményeznek ugyanúgy, mint az előző pontban bevezetett felhasználói típusok esetén. Az osztályon kívül elhelyezett függvénydefiníciók is hasonlóak lesznek, tehát az osztály nevét és a hatókör operátort írjuk a függvénynév elé.

Egy osztályon belül a tagok védelme az *elérhetőség* szabályozása által valósul meg. Az adattagok és tagfüggvények elérhetőségét a *private* (privát), *protected* (védtett) és *public* (nyilvános) kulcsszavakkal szabályozhatjuk. Mivel a tagok elérhetőségét változtathatják meg, *hozzáférés módosítóknak* is nevezzük őket. A hozzáférés módosítókat mint címkéket használjuk, azaz mindig kettőspont követi őket. Az így kapott címkék több részre osztják az osztály törzsét, ez által szabályozva azt, hogy melyek a nyilvános, védtett, illetve pri-

vát tagok. Például a *public* címkét követő összes adattag és tagfüggvény nyilvános lesz, egészen a következő címkéig. Jegyezzük meg azt is, hogy osztályok esetén alapértelmezés szerint a tagok privát elérhetőségűek.

A nyilvános tagok elérhetősége nincs korlátozva. Ezeket tetszőleges függvényben használhatjuk, ahol az illető osztály egy példányával dolgozunk. A privát és védett tagok elérhetősége korlátozott. Egyelőre nem teszünk különbséget köztük, csak később az alosztályok (§2.2) tanulmányozásakor foglalkozunk ezzel a kérdéssel.

Az objektumokra épülő programozás egyik alapelve az, hogy a nem nyilvános tagokat csak az illető osztály tagfüggvényeiben lehet elérni. Ez a szigorú követelmény bizonyos fokig enyhítve van a C++ programozási nyelvben. Enek megfelelően a privát és védett tagok elérhetősége az illető osztály tagfüggvényeire és *barát (friend) függvényeire* korlátozódik. A barát függvény nem tagfüggvénye az illető osztálynak, de ennek ellenére megengedjük, hogy hozzáférjen a privát és védett tagokhoz. Az előbb említett alapelvet figyelembe véve megállapíthatjuk, hogy ajánlott a barát függvények számát a minimálisra csökkenteni.

Az osztályok létrehozásakor mindig egy sajátos tagfüggvényt hív meg a rendszer, amit *konstruktor*nak nevezünk. Általában ezt a függvényt használjuk arra, hogy az adattagokat kezdeti értékkel lássuk el. A C++ nyelvben a konstruktor neve mindig megegyezik az osztály nevével, de a függvénynevek túlterhelése lehetővé teszi, hogy egy osztály több konstruktorral rendelkezzen. A konstruktorokkal az §1.5. pontban foglalkozunk részletesebben.

Az objektum létrehozása a hagyományos változók bevezetéséhez hasonló, tehát előbb az osztály nevét kell megadni, ami egy típusnév, és ezt követően az objektum nevét. Ha egyszerre több objektumot szeretnénk létrehozni, akkor ezeket vesszővel választhatjuk el. Mivel minden egyes új objektum egy konstruktormeghívást is jelent, ezért a deklaráláskor az objektumnév után kerek zárójelben meg kell adni a konstruktor aktuális paramétereit is.

Jegyezzük meg, hogy az előző pontban bevezetett *struct* kulcsszóval jellemzett felhasználói típus is tulajdonképpen egy osztály, tehát használhatók az elérhetőséget szabályozó címkék. A lényeges különbség az, hogy a *struct* kulcsszó esetén a tagok alapértelmezett elérhetősége nyilvános, míg a *class* esetén privát.

1.4. A tagokra való hivatkozás és a *this* mutató

Az előző pontokban láttuk, hogy egy felhasználói típus tagjaira való hivatkozást a tagkiválasztó, illetve a struktúra-mutató operátorral (a *.* és *->* operátorok) végezhetjük. A struktúra-mutató operátort akkor kell használni, ha egy objektumra hivatkozó mutatóval rendelkezünk, ellenkező esetben a tagkiválasztó operátorral dolgozunk.

A továbbiakban moduláris programozás (§1.1) esetén ismertetett 1.1. kódszöveget módosítjuk úgy, hogy osztályokra vonatkozzon, majd ezt követően vizsgáljuk a tagokra való hivatkozást.

1.4. kódszöveg. A *vektor* osztály.

```

1  #include <iostream>
2  using namespace std;
3  class vektor {
4  public:
5      vektor(int* az_elem, int a_meret);
6      ~vektor() { delete [] elem; }
7      void negyzetre();
8      void kiir();
9  private:
10     int* elem;
11     int meret;
12 };
13 vektor::vektor(int* az_elem, int a_meret)
14 {
15     meret = a_meret;
16     elem = new int[meret];
17     for(int i = 0; i < meret; i++)
18         elem[i] = az_elem[i];
19 }
20 void vektor::negyzetre()
21 {
```

```

22     for(int i = 0; i < meret; i++)
23         elem[i] *= elem[i];
24     }
25     void vektor::kiir()
26     {
27         for(int i = 0; i < meret; i++)
28             cout << elem[i] << ' ';
29         cout << endl;
30     }
31     void main()
32     {
33         int x[] = {1, 3, 5, 7, 9};
34         vektor v(x, 5);
35         vektor *p = &v;
36         v.kiir();
37         p->negyzetre();
38         p->kiir();
39         v.kiir();
40     }

```

A fenti kódszöveg fő függvényében előbb a *v* vektort vezettük be, majd a *p* mutatót, amely a *v* vektorra hivatkozik. Ez azt is jelenti, hogy a *p* segítségével előidézett változtatások a *v* vektorban is tükröződnek. Valóban a kimenet a következő lesz:

```

1 3 5 7 9
1 9 25 49 81
1 9 25 49 81

```

Tehát az elemenként négyzetreemelt vektor jelenik meg kétszer a képernyőn. Figyeljük meg, hogy a *v* esetén a tagkiválasztó operátort, a *p* esetén pedig a struktúra-mutató operátort használtuk.

Figyeljük meg, hogy a tagfüggvények belsejében direkt módon hivatkozhatunk az osztály tagjaira, nincs szükség tagkiválasztó, vagy struktúra-mutató operátorra. Mégis, felmerül a kérdés, hogy milyen módon azonosítja a rendszer az illető adattagot, tudva azt, hogy egy osztálynak több objektumát is létrehoztuk. A megoldás a *this* mutató használatában rejlik, mivel a tagfüggvények belsejében a tagokra való hivatkozás ezzel a mutatóval történik.

Pontosabban arról van szó, hogy minden egyes objektumon belül a rendszer létrehozza a *this* mutatót, amely az aktuális objektumra mutat. Például az 1.4. kódszöveg fő függvényében bevezetett *v* objektum esetén a *this* ennek az objektumnak a címe. Ha pedig az ugyanott definiált *p* mutatót tekintjük, akkor a *this* megegyezik *p*-vel.

Ennek alapján már könnyen azonosíthatóak a különböző objektumok tagjai. Az illető osztály tagfüggvényében a rendszer egyszerűen elvégez egy helyettesítést, azaz minden *tag* helyett *this->tag* lesz. Például az 1.4. kódszöveg *negyzetre* tagfüggvénye így alakul:

```

void vektor::negyzetre()
{
    for(int i = 0; i < this->meret; i++)
        this->elem[i] *= this->elem[i];
}

```

Hangsúlyozzuk, hogy nem kell mi megadnunk a fenti esetben a *this* mutatót, ezt automatikusan elhelyezi a rendszer. Mégis, a *this* mutatót explicit módon is használhatjuk, ha erre szükség van.

1.5. A konstruktor

Az előző pontok alapján tudjuk, hogy egy objektum létrehozását a konstruktorral végezzük. Továbbá, a konstruktor neve meg kell egyezzen az osztály nevével. Mégis, mivel a függvények túlterhelhetők, egy osztálynak több konstruktora is lehet, feltéve ha a paraméterlisták különböznek. Fontos, hogy a konstruktor nem térít vissza értéket. A konstruktor deklarációja nem tartalmazhat semmit a visszatérítendő típus helyén, még a *void* kulcsszót sem.

Az alábbi példa több konstruktor együttes használatát szemlélteti. Egy olyan osztályt hozunk létre, amely különböző személyek családnévét és keresztnévét tárolja.

1.5. kódszöveg. A *szemely.h* fejlécállomány.

```

1  #include <iostream>
2  using namespace std;
3  class szemely {
4      char* cs_nev;
5      char* sz_nev;
6  public:
7      szemely();          //alapértelmezett konstruktor
8      szemely(char* cs_n, char* sz_n);
9      szemely(const szemely& sz);    // másoló konstruktor
10     ~szemely();
11     void kiir();
12 };
13 szemely::szemely() {
14     cs_nev = new char[1];
15     *cs_nev = 0;        // 0 és '\0' ugyanaz
16     sz_nev = new char[1];
17     *sz_nev = 0;
18     cout << "Alapertelmezett konstruktor\n";
19 }
20 szemely::szemely(char* cs_n, char* sz_n)
21 {
22     cs_nev = new char[strlen(cs_n)+1];
23     sz_nev = new char[strlen(sz_n)+1];
24     strcpy(cs_nev, cs_n);
25     strcpy(sz_nev, sz_n);
26     cout << "Hagyomanyos konstruktor\n";
27 }
28 szemely::szemely(const szemely& x)
29 {
30     cs_nev = new char[strlen(x.cs_nev)+1];
31     strcpy(cs_nev, x.cs_nev);
32     sz_nev = new char[strlen(x.sz_nev)+1];
33     strcpy(sz_nev, x.sz_nev);
34     cout << "Masolo konstruktor\n";
35 }
36 szemely::~szemely() {
37     cout << "Destruktor\n";
38     delete[] cs_nev;
39     delete[] sz_nev;
40 }
41 void szemely::kiir() {
42     if ( strlen(cs_nev) > 0 )
43         cout << cs_nev << ' ' << sz_nev << endl;
44     else
45         cout << "Nincs adat\n";
46 }

```

Ez a forráskód három konstruktort tartalmaz. Ezek közül a 8. sorbeli konstruktordeklarációt hagyományosnak tekinthetjük abban az értelemben, hogy az adattagok (családnév és keresztnév) kezdeti értékkel való ellátását valósítja meg. Figyeljük meg, hogy két sajátos konstruktor is szerepel a fenti kódban. Az egyik az *alapértelmezett konstruktor*, vagy más néven *alapértelmezés szerinti konstruktor*, a másik a *másoló konstruktor*.

Ha a konstruktor formális paramétereinek listája üres, akkor beszélünk alapértelmezett konstruktorról. Az alapértelmezés szerinti konstruktornak fontos szerepe van azoknak az objektumoknak a létrehozásában, amelyek nem rendelkeznek kezdeti értékek megadó aktuális paraméterekkel. Pontosabban, ha egy osztálynak van alapértelmezett konstruktora, akkor létrehozható olyan objektum, amely nem tartalmaz inicializáló ak-

tuális paraméterekből álló listát. Ez akkor is lehetséges, ha olyan konstruktorunk van, amelynek az összes formális paramétere kezdeti értékkel van ellátva. Tehát az ilyen konstruktort is alapértelmezett konstruktornak nevezhetjük.

A konstruktorokon kívül az 1.5. kódszöveg tartalmaz egy sajátos tagfüggvényt, a *destruktor*, melyet az objektumok megszűnésekor hív meg a rendszer.

Tekintsük az 1.5. kódszöveget felhasználó alábbi fő függvényt:

1.6. kódszöveg. A *szemely* osztály objektumainak létrehozása.

```

1  #include "szemely.h"
2  void main() {
3      szemely BF("Bolyai", "Farkas");
4      BF.kiir();
5      szemely *FGy = new szemely("Farkas","Gyula");
6      FGY->kiir();
7      szemely A; //alapértelmezett konstruktor
8      A.kiir();
9      szemely Gyula(*FGy); // masoló konstruktor
10     Gyula.kiir();
11     delete FGY;
12 }
```

Ennek a kódnak a kimenete a következő lesz:

```

Hagyomanyos konstruktor
Bolyai Farkas
Hagyomanyos konstruktor
Farkas Gyula
Alapertelmezett konstruktor
Nincs adat
Masolo konstruktor
Farkas Gyula
Destruktor
Destruktor
Destruktor
Destruktor
```

Megfigyelhetjük, hogy először a *BF* objektumot hoztuk létre a hagyományos konstruktorral. Ezt követően a szabad tárból jön létre egy objektum, amelyre az *FGy* mutatóval hivatkozhatunk. Itt is a hagyományos konstruktort hívta meg a rendszer, mivel a *new* operátor után az osztály nevet és, kerek zárójelben, az aktuális paraméterek listáját adtuk meg. Az *A* objektumot az alapértelmezett, a *Gyula* objektumot pedig a másoló konstruktorral hoztuk létre.

A alapértelmezett konstruktor mindkét adattagba az üres karakterláncot másolja. Mivel ennek a hossza zéró, a *kiir* tagfüggvény a „*Nincs adat*” üzenetet jeleníti meg. Feltételeztük, hogy ha a családnév üres, akkor a keresztnévet sem adtuk meg.

Egy osztályt úgy is deklarálhatunk, hogy nem adunk meg konstruktort. Jegyezzük meg, hogy ha nincs, a programozó által bevezetett konstruktor, akkor a rendszer létrehoz egy alapértelmezett konstruktort, és ezt hívja meg minden alkalommal, amikor egy új objektum keletkezik. Ez a konstruktor nem ad kezdeti értékeket az adattagoknak.

Ha a programozó létrehozott egy vagy több konstruktort, akkor a rendszer nem generál alapértelmezett konstruktort. Ha ezen konstruktorok közül egyik sem alapértelmezett, és szeretnénk olyan objektumot létrehozni, amely nem tartalmaz aktuális paraméterekből álló listát, akkor kötelesek vagyunk egy alapértelmezett konstruktort definiálni.

A másoló konstruktor célja az, hogy egy objektumot kezdeti értékekkel lásson el egy ugyanolyan típusú objektum segítségével. Általában az

```
osztálynév(const osztálynév & objektum);
```

alakban deklaráljuk, ahol a *const* kulcsszó arra utal, hogy a paraméterként megadott objektum nem változik.

Ha a programozó nem definiál másoló konstruktort, akkor a rendszer létrehoz egy másoló konstruktort, amely az adattagok *bitenkénti másolását* végzi. Ez azt jelenti, hogy megfigyeljük egymásnak a rendszer az

adattagokat, majd a forrás adattag bitjeit rendre átmásolja a cél adattagba. A bitenkénti másolás általában akkor ad helyes eredményt, ha az osztálynak nincsen mutató típusú adattagja.

Például az 1.5. és 1.6. kódszövegek esetén, ha nem definiáltunk volna másoló konstruktort, akkor futási időben hibát észleltünk volna. Pontosabban, kétszer próbálta volna meg felszabadítani ugyanazt a memóriaterületet a rendszer. Ennek a hibának az oka abban rejlik, hogy a *Gyula* objektum létrehozásakor egy bitenkénti másolást végzett a rendszer, tehát a **FGy* objektum *cs_nev* és *sz_nev* adattagjait másolta át. Mivel mindkét adattag értéke egy cím, ezért ezt a címet másoltuk át, tehát a *Gyula* objektum *cs_nev* és *sz_nev* adattagjai ugyanarra a memóriaterületre fognak mutatni, ahova a **FGy* objektum adattagjai. Ez viszont nem az, amit meg szeretnénk volna tenni, mivel így, ha az egyik objektum megszűnik, a másiknak is fel lesz szabadítva a memóriaterülete és fordítva. E helyett a másoló konstruktort terheljük túl, amely új memóriaterületet foglal le, és erre másolja a családnevet és keresztnévet.

Jegyezzük meg, hogy a rendszer akkor hívja meg a másoló konstruktort, ha:

- ugyanolyan típusú objektummal adunk kezdőértéket;
- egy függvénynek a paramétere egy objektum;
- egy függvény objektumot térít vissza.

Ezért, ha van mutató típusú adattag, akkor a másoló konstruktort definiálnunk kell akkor is, ha nincs szándékunkban a kezdőértékkadást ugyanolyan típusú objektummal végezni.

Az 1.6. kódszövegben a *new* operátorral dinamikus módon hoztuk létre az egyik objektumot. A *new* utáni típust követően kerek zárójelt használtunk, és ezen belül adtuk meg a konstruktor aktuális paramétereit.

Lehetőség van arra, hogy egy osztály törzsében osztály típusú tagokat helyezünk el. A következő példa keretében azt vázoljuk fel, hogy ha egy osztályon belül *n* darab különböző osztály típusú tagot helyezünk el, akkor hogyan alakul az illető osztály konstruktora.

```
class oszt {
    oszt_1 ob_1;
    oszt_2 ob_2;
    ...
    oszt_n ob_n;
};
```

Ebben az esetben az *oszt* osztály konstruktorának a fejléce a következőképpen adható meg:

```
oszt(argumentumlista) : objektumlista
```

az *objektumlista* pedig az

```
ob_1(arglista_1), ob_2(arglista_2), ..., ob_n(arglista_n)
```

alakú kell legyen. Természetesen, sem itt, sem az osztálydeklarációban a három pont nem része a szintaxisnak, csak jelzi a folytatást. Az *argumentumlista* az *oszt* osztály konstruktorában a formális paraméterek listája. Továbbá, minden egyes *i* értékre 1-től *n*-ig az *arglista_i* az *ob_i* osztály konstruktorában az aktuális paraméterek listája. Az egyes objektumok aktuális paraméterei az *argumentumlistából* alkotott kifejezések lesznek.

Jegyezzük meg, hogy az objektumlistából hiányoznak azok az objektumok, amelyek nem rendelkeznek a programozó által bevezetett konstruktorral. Ezen kívül hiányozhatnak az objektumlistából azok az objektumok is, amelyekre az alapértelmezett konstruktort szeretnénk meghívni.

Egy másik fontos észrevétel a következő. Ha egy osztálynak egyik adattagja egy objektum, akkor először ennek az objektumnak a konstruktorát hívja meg a rendszer, majd ezt követően lesz végrehajtva az osztály konstruktorának a törzse.

A továbbiakban az 1.5. kódszöveget úgy módosítjuk, hogy eltávolítjuk a konstruktorokból és a destruktorból a kiírásokat, vagyis a 18., 26., 34. és 37. sorokat töröljük. Legyen az így kapott állomány neve *szemely2.h*. Ezt felhasználva a következő példa *házaspárok* adatait tárolja, mégpedig úgy, hogy osztály típusú tagokat használ.

1.7. kódszöveg. Osztály típusú tagok.

```
1  #include "szemely2.h"
2  class hazaspar {
3      szemely ferj;
4      szemely feleseg;
5  public:
```

```

6      hazaspar() // alapértelmezett konstruktor
7      {
8      }
9      hazaspar(szemely& aferj, személy& afelesege);
10     hazaspar(char* cs_ferj, char* sz_ferj,
11             char* cs_felesege, char* sz_felesege):
12     ferj(cs_ferj, sz_ferj), feleseg(cs_felesege, sz_felesege)
13     {
14     }
15     void kiir();
16 };
17 inline hazaspar::hazaspar(szemely& aferj, személy& afelesege):
18 ferj(aferj), feleseg(afelesege)
19 {
20 }
21 void hazaspar::kiir()
22 {
23     cout << "ferj: ";
24     ferj.kiir();
25     cout << "felesege: ";
26     feleseg.kiir();
27 }
28 void main() {
29     személy Ady("Ady", "Endre");
30     személy Csinszka("Boncza", "Berta");
31     hazaspar Hpar(Ady, Csinszka);
32     Hpar.kiir();
33     hazaspar Petofi("Petofi", "Sandor", "Szendrei", "Julia");
34     Petofi.kiir();
35     hazaspar XY;
36     XY.kiir();
37 }

```

A program kimenete a következő lesz:

```

ferj:  Ady Endre
felesege:  Boncza Berta
ferj:  Petofi Sandor
felesege:  Szendrei Julia
ferj:  Nincs adat
felesege:  Nincs adat

```

Az 1.7. kódszöveg három konstruktorral rendelkezik. Az alapértelmezett konstruktor definíciója is az osztályon belülre került, ezért ez helyben kifejtett függvény (*inline* függvény) lesz. Mivel a konstruktor fejlécét úgy adtuk meg, hogy hiányzik a kettőspont, és az azt követő objektumlista, ezért ez a konstruktor az összes osztály típusú tagnak az alapértelmezett konstruktorát hívja meg. Erre utal az is, hogy a fő függvényben az *XY* objektum kiírásakor a „*Nincs adat*” üzenet jelenik meg.

A 9. sorban egy konstruktordeklaráció szerepel, a definíció most az osztályon kívülre került. Mivel azt szeretnénk, hogy ez is helyben kifejtett függvény legyen az *inline* minősítőt használjuk a függvénydefinícióban. Ez a konstruktor a személy osztály másoló konstruktorával hozza létre a *ferj* és *felesege* tagokat.

A harmadik konstruktor a családnevekkel és személynevekkel hozza létre az osztály típusú tagokat. Ezért a *szemely* osztály hagyományos konstruktorát hívja meg a rendszer mindkét adattagra.

1.6. A destruktork

Az eddigi pontok alapján tudjuk, hogy ha egy objektum megszűnik, akkor a rendszer automatikusan végrehajt egy sajátos tagfüggvényt, amit *destruktor*nak nevezünk. A továbbiakban részletesebben vizsgáljuk a destruktort.

A destruktork neve mindig a `~` karakterrel kezdődik, és ez után az osztály neve következik. A konstruktorhoz hasonlóan a destruktork sem térít vissza értéket, és még a `void` típust sem szabad megadni a visszatérítendő érték típusaként.

Felmerül a kérdés, hogy mikor hívódnak meg az egyes destruktorkok. Ez a hatókörtől függ. Egy globális objektum destruktora a `main` függvény végén az `exit` függvény részeként lesz végrehajtva. Ezért nem szabad az `exit` függvényt meghívni a destruktorkban, mivel ez végtelen ciklust eredményezhet.

Egy helyi objektum destruktorkát akkor hívja meg a rendszer, ha annak a blokknak a végére értünk, amelyben be volt vezetve.

Végül tekintsük azt az esetet is, amikor a `new` operátorral hoztunk létre a szabad tárból egy objektumot. Ezeket dinamikus módon létrehozott objektumoknak is nevezzük. Ekkor a destruktork a `delete` operátoron keresztül hívja meg a rendszer. Valóban ekkor lesz felszabadítva a `new` operátor által lefoglalt memóriaterület.

A továbbiakban egy olyan példa keretében szemléltetjük a destruktork működését, amely minden esetben kiírja, hogy éppen mit végzett, azaz milyen konstruktork vagy destruktork hívott meg. A kiírást most a `printf` függvénnyel végezzük.

1.8. kódszöveg. A destruktork.

```

1  #include <stdio>
2  #include <cstring>
3  using namespace std;
4  class kiiras {
5      char* nev;
6  public:
7      kiiras(char* n);
8      ~kiiras();
9  };
10 kiiras::kiiras(char* n)
11 {
12     nev = new char[strlen(n)+1];
13     strcpy(nev, n);
14     printf("Letrehoztam: %s\n", nev);
15 }
16 kiiras::~~kiiras()
17 {
18     printf("Felszabaditottam: %s\n", nev);
19     delete nev;
20 }
21 void fuggv()
22 {
23     printf("Fuggvenymeghivas.\n");
24     kiiras helyi("HELYI");
25 }
26 kiiras globalis("GLOBALIS");
27 void main() {
28     kiiras* dinamikus = new kiiras("DINAMIKUS");
29     fuggv();
30     printf("Folytatodik a fo fuggveny.\n");
31     delete dinamikus;
32 }
```

Végrehajtva a programot, a következő kimenetet kapjuk:

```

Letrehoztam: GLOBALIS
Letrehoztam: DINAMIKUS
Fuggvenymeghivas.
Letrehoztam: HELYI
Felszabaditottam: HELYI
Folytatodik a fo fuggveny.
Felszabaditottam: DINAMIKUS
Felszabaditottam: GLOBALIS
```

A forráskódban egy *kiiras* nevű osztályt vezettünk be, és létrehoztuk ennek három objektumát. Figyeljük meg, hogy a globális objektumot hozta először létre a rendszer, ugyanakkor ennek a destruktora lesz utolsónak végrehajtva. A helyi objektum destruktora a függvényből való kilépéskor, a dinamikus objektumé pedig a *delete* operátor részeként hívódik meg.

2. Az objektumorientált programozási módszer

2.1. Elméleti alapok

Az objektum adattagokat és tagfüggvényeket tartalmaz. Ha nem használunk barát függvényeket a védett tagok csak a tagfüggvényekben érhetők el. Ezt a tulajdonságot *egybezártságnak* (*zártságnak*) nevezzük.

A gyakorlatban viszont nem csak különálló objektumokkal találkozunk. A különböző objektumok közti kapcsolatok is fontosak. Egy osztály öröklöheti egy másik osztály tagjait. Az eredeti osztály neve *alaposztály*, vagy *bázisosztály*. Az örökléssel létrehozott osztályt *származtatott osztálynak* nevezzük. Az adattagok, és a tagfüggvények is öröklődnek. Ha egy osztály több alaposztállyal rendelkezik, akkor *többszörös öröklésről* beszélünk. Az *öröklés* egy másik fontos tulajdonsága az objektumoknak. Az objektumok egy hierarchiát alkothatnak.

Az öröklött tagfüggvények túlterhelhetőek. Nem csak a függvény neve, hanem a paraméterlistája is ugyanaz lehet. Az objektumhierarchia különböző szintjein ugyanannak a műveletnek más és más értelme lehet. Ezt a tulajdonságot *polimorfizmusnak* nevezzük.

2.2. Származtatott osztályok deklarálása

A C++ programozási nyelvben a származtatott osztályokat az alábbi módon adjuk meg:

```
class oszt : alaposztálylista {
    // új adattagok és tagfüggvények
};
```

ahol az alaposztálylista vesszővel elválasztott elemei

```
public alaposztály
protected alaposztály
private alaposztály
```

alakúak kell legyenek. Ha minden egyes esetben a *public* hozzáférésmódosítót használjuk, akkor a

```
class oszt : public oszt_1, ..., public oszt_n {
    // ...
};
```

alakú szerkezetet kapjuk, ahol az *oszt* osztály az *oszt_1*, ..., *oszt_n* osztályok származtatott osztálya. Jegyezzük meg, hogy a konstruktorok és destruktorkok nem öröklődnek. A származtatott osztály konstruktorát az

```
oszt(paraméterlista) :
    oszt_1(lista1), ..., oszt_n(lista_n)
{
    // ...
}
```

módon definiáljuk. A következő pontban olyan példákat adunk származtatott osztályra, amelyek lehetőséget teremtenek a *virtuális tagfüggvények* bevezetésére is.

2.3. Virtuális tagfüggvények

Tekintsük egy olyan példát származtatott osztályra, amelyben az *alap* nevű osztályban két függvényt deklarálunk, és a második meghívja az elsőt. Ugyanakkor a származtatott osztályban csak az elsőt írjuk felül.

1.9. kódszöveg. Virtuális tagfüggvény.

```
1  #include <iostream>
2  using namespace std;
3  class alap {      // az alaposztály
```

```

4     public:
5         void f1();
6         void f2();
7     };
8     class szarm : public alap {
9     public:
10        void f1();
11    };
12    void alap::f1()
13    {
14        cout << "alap:  f1\n";
15    }
16    void alap::f2()
17    {
18        cout << "alap:  f2\n";
19        f1();          // az f2 meghívja az f1-et.
20    }
21    void szarm::f1()
22    {
23        cout << "szarmaztatott:  f1\n";
24    }
25    void main() {
26        szarm s;
27        s.f2();
28    }

```

Figyeljük meg, hogy csak az *f1* tagfüggvényt írtuk felül, az *f2* öröklődik az alaposztálytól. A fő függvényben a származtatott osztálynak hoztuk létre egy objektumát és az erre az *f2* függvényt hívtuk meg. Felmerül a kérdés, hogy ilyen módon melyik *f1* függvény lesz végrehajtva?

Az 1.9. kódszöveg esetén az *f1* függvény kiválasztása fordítási időben történt, ezért az alaposztály *f1* tagfüggvénye lesz végrehajtva. Ezt a tulajdonságot *statikus kötésnek* nevezzük.

Ha a végrehajtandó függvény kiválasztása futási időben történik, akkor *dinamikus kötésről* beszélünk. A dinamikus kötés virtuális tagfüggvények segítségével valósíthatjuk meg. Az *f1* tagfüggvényt kell virtuálisnak deklarálni. Ezt úgy tehetjük meg, hogy a *virtual* minősítőt használjuk a függvény alaposztálybeli deklarációjában. Ebben az esetben az alaposztályt a

```

class alap {
public:
    virtual void f1();
    void f2();
};

```

alakban adjuk meg. Így a származtatott osztálybeli *f1* függvény lesz végrehajtva.

Figyeljük meg, hogy a *virtual* kulcsszót elég egyszer megadni, az alaposztálybeli deklarációban. Ebben az esetben a származtatott osztályban deklarált túlterhelt tagfüggvény is virtuális lesz. Ha egy függvényt virtuálisnak deklaráltunk az alaposztályban, akkor az osztályhierarchia tetszőleges származtatott osztályában virtuális lesz.

A továbbiakban tekintsünk egy másik példát, amelyben felmerül a virtuális tagfüggvények megadásának a szükségszerűsége. Vezessük be a racionális számokra vonatkozó *tort* nevű osztályt, amely két egész típusú adattaggal rendelkezik, melyek a számlálónak és nevezőnek felelnek meg. Az osztály kell rendelkezzen egy olyan konstruktorral, amely a számlálót és a nevezőt kezdeti értékekkel látja el. Alapértelmezetten a számláló értéke legyen 1, a nevezője pedig 0. Továbbá, az osztálynak kell legyen egy *szorzat* és egy *szoroz* nevű tagfüggvénye is. Az első a két tört szorzatát számolja ki, a második pedig az aktuális objektumot módosítja úgy, hogy azt megszorozza a paraméterként megadott objektummal. Ugyanakkor a *tort* osztálynak kell legyen egy olyan tagfüggvénye is, amely az illető racionális számot írja ki.

A fenti osztályt felhasználva egy olyan *tort_kiir* nevű osztályt is létre kell hozni, amely a *szorzat* tagfüggvényt úgy módosítja, hogy a művelet elvégzésén kívül maga a művelet is jelenjen meg a szabványos kimeneten. A *szoroz* tagfüggvényt nem írjuk felül, de a műveletnek ebben az esetben is meg kell jelennie.

1.10. kódszöveg. A szorzat virtuális tagfüggvény bevezetése a racionális számokra vonatkozó osztály esetén.

```

1  #include <iostream>
2  using namespace std;
3  class tort {
4  protected:
5      int szamlalo;
6      int nevezol;
7  public:
8      tort(int szamlalol = 0, int nevezol = 1);
9      /*virtual*/ tort szorzat(tort& r);
10     tort& szoroz(tort& r);
11     void kiir();
12 };
13 tort::tort(int szamlalol, int nevezol)
14 {
15     szamlalo = szamlalol;
16     nevezol = nevezol;
17 }
18 // két tört szorzatát számolja ki, de nem egyszerűsít
19 tort tort::szorzat(tort& r)
20 {
21     return tort(szamlalo * r.szamlalo, nevezol * r.nevezol);
22 }
23 // az aktuális objektumot módosítja
24 tort& tort::szoroz(tort& q)
25 {
26     *this = this->szorzat(q);
27     return *this;
28 }
29 void tort::kiir()
30 {
31     if ( nevezol )
32         cout << szamlalo << " / " << nevezol;
33     else
34         cerr << "helytelen tort";
35 }
36 class tort_kiir: public tort {
37 public:
38     tort_kiir( int szamlalol = 0, int nevezol = 1 );
39     tort szorzat( tort& r);
40 };
41 inline tort_kiir::tort_kiir(int szamlalol, int nevezol) :
42 tort(szamlalol, nevezol)
43 {
44 }
45 tort tort_kiir::szorzat(tort& q)
46 {
47     tort r = tort(*this).szorzat(q);
48     cout << "(";
49     this->kiir();
50     cout << ") * (";
51     q.kiir();
52     cout << ") = ";
53     r.kiir();
54     cout << endl;
55     return r;
56 }
57 int main()

```

```

58     {
59         tort p(3,4), q(5,2), r;
60         r = p.szoroz(q);
61         p.kiir();
62         cout << endl;
63         r.kiir();
64         cout << endl;
65         tort_kiir p1(3,4), q1(5,2);
66         tort r1, r2;
67         r1 = p1.szorzat(q1);
68         r2 = p1.szoroz(q1);
69         p1.kiir();
70         cout << endl;
71         r1.kiir();
72         cout << endl;
73         r2.kiir();
74         cout << endl;
75         return 0;
76     }

```

A programot végrehajtva az alábbi kimenetet kapjuk:

```

15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8

```

Figyeljük meg, hogy a kapott eredmény nem megfelelő, mivel a művelet kiírása csak egy alkalommal jelent meg. Ahhoz, hogy az elvárt eredményt kapjuk, a *szorzat* tagfüggvényt virtuálisnak kell deklarálni, és ezt úgy tehetjük meg, hogy az 1.10. kódszöveg 9. sorából eltávolítjuk a megjegyzés jelét. Ha ezt meg tesszük, akkor a kimenet így módosul:

```

15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8

```

tehát valóban kétszer jelenik meg a műveletre vonatkozó kiírás.

2.4. Absztrakt osztályok

Egy alaposztálynak lehetnek olyan általános tulajdonságai, amelyekről tudunk, de nem tudjuk őket definiálni csak egy származtatott osztályban. Ebben az esetben egy olyan virtuális tagfüggvényt deklarálhatunk, amely nem lesz definiálva az alaposztályban. Azokat a tagfüggvényeket, amelyek deklarálva vannak, de nincsenek definiálva egy adott osztályban, *tiszta virtuális tagfüggvényeknek* nevezzük.

A tiszta virtuális tagfüggvényt a szokásos módon deklaráljuk, de a fejléc után az = 0 karaktereket írjuk. Ez jelzi, hogy a tagfüggvényt nem fogjuk definiálni.

Azokat az osztályokat, amelyek tartalmaznak legalább egy tiszta virtuális tagfüggvényt, *absztrakt osztályoknak* nevezzük. Az absztrakt osztályoknak nem hozhatjuk létre objektumát.

A tiszta virtuális tagfüggvényeket felül kell írni a származtatott osztályban, ellenkező esetben az illető osztály is absztrakt lesz.

Tekintsük a következő példát

1.11. kódszöveg. Absztrakt osztály.

```

1 #include <iostream>

```

```

2   using namespace std;
3   class allat {
4   protected:
5       double suly;      // kg
6       double eletkor;   // ev
7       double sebesseg;  // km / h
8   public:
9       allat( double su, double k, double se);
10      virtual double atlagos_suly() = 0;
11      virtual double atlagos_eletkor() = 0;
12      virtual double atlagos_sebesseg() = 0;
13      int kover() { return suly > atlagos_suly(); }
14      int gyors() { return sebesseg > atlagos_sebesseg(); }
15      int fiatal() { return 2 * eletkor < atlagos_eletkor(); }
16      void kiir();
17  };
18  allat::allat( double su, double k, double se)
19  {
20      suly = su;
21      eletkor = k;
22      sebesseg = se;
23  }
24  void allat::kiir()
25  {
26      cout << ( kover() ? "kover, " : "sovany, " );
27      cout << ( fiatal() ? "fiatal, " : "oreg, " );
28      cout << ( gyors() ? "gyors" : "lassu" ) << endl;
29  }
30  class galamb : public allat {
31  public:
32      galamb( double su, double k, double se):
33          allat(su, k, se) {}
34      double atlagos_suly() { return 0.5; }
35      double atlagos_eletkor() { return 6; }
36      double atlagos_sebesseg() { return 90; }
37  };
38  class medve: public allat {
39  public:
40      medve( double su, double k, double se):
41          allat(su, k, se) {}
42      double atlagos_suly() { return 450; }
43      double atlagos_eletkor() { return 43; }
44      double atlagos_sebesseg() { return 40; }
45  };
46  class lo: public allat {
47  public:
48      lo( double su, double k, double se):
49          allat(su, k, se) {}
50      double atlagos_suly() { return 1000; }
51      double atlagos_eletkor() { return 36; }
52      double atlagos_sebesseg() { return 60; }
53  };
54  void main() {
55      galamb g(0.6, 1, 80);
56      medve m(500, 40, 46);
57      lo l(900, 8, 70);
58      g.kiir();
59      m.kiir();
60      l.kiir();

```

```
61     }
```

A programot futtatva az alábbi kimenetet kapjuk:

```
kover, fiatal, lassu
kover, oreg, gyors
sovany, fiatal, gyors
```

Figyeljük meg, hogy annak ellenére, hogy az *allat* osztályt absztraktnak deklaráltuk, hasznos volt ennek bevezetése, mivel egyes tagfüggvényeket már az alaposztály szintjén definiálni lehetett. Ezek öröklődtek a származtatottakba és így nem kellett őket minden egyes esetben külön-külön megírni.

2.5. Az interfész fogalma

A C++ programozási nyelvben az interfész fogalma nincsen értelmezve abban a formában, ahogyan az létezik a Java és C# programozási nyelvekben. De tetszőleges olyan absztrakt osztályt, amely csak tiszta virtuális függvényeket tartalmaz interfésznek tekinthetünk. Természetesen ebben az esetben nem fogunk deklarálni adattagokat sem az osztályon belül. Az előző pontban bevezetett *allat* nevű osztály adattagokat is és nem virtuális függvényeket is tartalmaz, ezért ez nem tekinthető interfésznek. A továbbiakban egy *Jarmu* nevű absztrakt osztályt adunk meg, amely csak tiszta virtuális tagfüggvényekkel rendelkezik. Ugyanakkor ennek az osztálynak két származtatottját is létrehozunk.

1.12. kódszöveg. Absztrakt osztály, amely interfésznek tekinthető.

```
1  #include <iostream>
2  using namespace std;
3  class Jarmu
4  {
5  public:
6      virtual void Indul() = 0;
7      virtual void Megall() = 0;
8      virtual void Megy(int km) = 0;
9      virtual void All(int perc) = 0;
10 };
11 class Bicikli : public Jarmu
12 {
13 public:
14     void Indul();
15     void Megall();
16     void Megy(int km);
17     void All(int perc);
18 };
19 void Bicikli::Indul() {
20     cout << "Indul a bicikli." << endl;
21 }
22 void Bicikli::Megall() {
23     cout << "Megall a bicikli." << endl;
24 }
25 void Bicikli::Megy(int km) {
26     cout << "Biciklizik " << km << " kilometert." << endl;
27 }
28 void Bicikli::All(int perc) {
29     cout << "A bicikli all " << perc << " percet." << endl;
30 }
31 class Auto : public Jarmu
32 {
33 public:
34     void Indul();
35     void Megall();
36     void Megy(int km);
37     void All(int perc);
```

```

38     };
39     void Auto::Indul() {
40         cout << "Indul az auto." << endl;
41     }
42     void Auto::Megall() {
43         cout << "Megall az auto." << endl;
44     }
45     void Auto::Megy(int km) {
46         cout << "Az auto megy " << km << " kilometert." << endl;
47     }
48     void Auto::All(int perc) {
49         cout << "Az auto all " << perc << " percet." << endl;
50     }
51     void BejarUt(Jarmu *j)
52     {
53         j->Indul();
54         j->Megy(3);
55         j->All(1);
56         j->Megy(2);
57         j->Megall();
58     }
59     int main()
60     {
61         Jarmu *b = new Bicikli;
62         BejarUt(b);
63         Jarmu *a = new Auto;
64         BejarUt(a);
65         delete a;
66         delete b;
67     }

```

A fő függvényben egy *Bicikli* és egy *Auto* típusú dinamikus objektumot deklaráltunk. Ha ezekre az objektumokra a *BejarUt* nevű tagfüggvényt hívjuk meg, különböző eredményt kapunk, annak ellenére, hogy a függvénynek csak egy olyan paramétere van, amely a *Jarmu* absztrakt osztályra hivatkozó mutató.

III. Operációs rendszerek

III.1. A Unix állományrendszer

III.1.1. Állományok típusa

Az operációs rendszerek a különféle, összetartozó adatokat *állományokban* avagy *file-okban* tárolják.

A UNIX megkülönböztet közönséges-, illetve speciális állományokat. A közönséges állomány teljesen strukturálatlan, egyszerűen bájtok sorozata. Egy UNIX file végét nem jelzik speciális karakterek, a filenak akkor van vége, amikor az olvasó rutin hibajelzéssel tér vissza. Standard bemenet esetén a file végét újsorban ^D jelzi.

Egy speciális állomány ezzel szemben meghatározott szerkezetű, különleges célt szolgál. A következő fajta speciális állományokról beszélhetünk: katalógus (directory), eszköz (device), szimbolikus lánc (symbolic link), nevesített FIFO cső (named pipe, FIFO), illetve kommunikációs végpont (socket).

Beszélhetünk továbbá a folyamatok közötti kommunikációt, illetve szinkronizálást szolgáló eszközökről, melyeket a rendszerhívások szintaktikai szempontból szintén állományként látnak. Ezeket az eszközöket a Unix magja kezeli: név nélküli cső (pipe), osztott memória szegmensek, üzenetsorok, szemaforok.

Egy *közönséges állomány* oktettjeit feldolgozhatjuk szekvenciálisan, de hozzáférhetünk közvetlenül is egy bizonyos bájtához, a sorszámanak segítségével.

Egy *katalógusfile* csupán a tartalmát illetően különbözik egy közönséges állománytól. A katalógusban szereplő minden file-hoz (közönséges állomány, alkatalógus, stb.) tartalmaz egy bejegyzést. Minden felhasználó rendelkezik egy úgynevezett *alapkatalógussal* (home directory), mely az általa használt közönséges állományokat, illetve általa létrehozott alkatalógusokat tartalmazza (~ vagy \$HOME).

Minden katalógus két speciális bemenetet tartalmaz:

- " ." (pont) magára a katalógusra mutat;
- " .. " (két egymásutáni pont), a szülőkatalógusra mutat (parent directory).

Minden állományrendszer egyetlen gyöker katalógust (root directory) tartalmaz: /.

A katalógusszerkezetet egy faszerkezet (gráf) határozza meg. Az elérési út megadásánál az elválasztójel a /. Kétféle módon megadott elérési útról beszélhetünk:

- *abszolút elérési út*: a gyökerhez (/) képest megadott hely.
- *relatív elérési út*: az aktuális katalógushoz (.) képest megadott hely (egy elérési út relatív, ha nem a / vagy ~ jelekkel kezdődik).

A katalógus, amelyben a felhasználó éppen dolgozik, az úgynevezett *aktuális katalógus* (current directory). Ennek megváltoztatása a `cd` parancs segítségével lehetséges. Az aktuális katalógus abszolút elérési útját (a gyöker katalógustól kezdődően) a `pwd` parancs adja meg. Létrehozhatunk egy új katalógust az `mkdir` parancs segítségével, egy katalógus törlését pedig a `rmdir` parancs teszi lehetővé.

III.1.2. Állományok jellemzői

Egy állományt az alábbi tulajdonságok jellemeznek:

- név
- inode szám – az állomány tulajdonságait tároló inode tábla megfelelő bemenetének azonosítója
- típus
- méret
- tulajdonos (owner)
- csoport (group)
- hozzáférési jogok
- létrehozás, utolsó hozzáférés ill. utolsó módosítás dátuma és ideje
- láncszám – hány különböző katalógusbemenet hivatkozik ugyanarra az állományra

A következő hozzáférési jogokat különböztetjük meg:

- **olvasási jog** – 4 (read permission): az állomány olvasható, ill. a katalógus tartalma listázható
- **írási jog** – 2 (write permission): az állomány módosítható, ill. a katalógusban állományokat lehet létrehozni és törölni
- **végrehajtási jog** – 1 (execute permission): az állomány programként végrehajtható, ill. a katalógusban levő állományok/ katalógusok hozzáférhetőek, be lehet lépni a katalógusba
- **setuid**: a programfile a file jogaival fut (nem a futtató jogaival!)
- **setgid**: a programfile a file csoportjának jogaival fut
- **sticky**: a katalógusban állományt törölni vagy átnevezni csak a tulajdonos tud

Egy állomány hozzáférési jogai négy csoportba sorolhatóak:

- speciális jogok (setuid – 4, setgid – 2, sticky – 1)
- a file tulajdonosának jogai (owner, owner user)
- a file csoportjának jogai (group)
- mindenki más jogai (other users)

A `chmod` parancs segítségével módosíthatjuk egy állomány hozzáférési jogait. A jogok megadása kétféleképpen történhet: numerikusan vagy szimbolikusan.

A *numerikus* (oktális számokkal történő) megadás esetén a parancs a következőképpen néz ki:

```
chmod [-R] perm-mode file ...
```

ahol *perm-mode* a beállítani kívánt új hozzáférési jogosultság. Több filenevet is meg lehet adni szükség szerint. (A `-R` opcióval rekurzív módon, a megadott katalógus alatti teljes állományrendszeren módosítja a jogosultságokat.) A beállítani kívánt jogokat oktális szám formájában kell megadni, az alábbiak szerint: az olvasás értéke 4, az írásé 2, a végrehajtásé 1, ezeket az értékeket össze kell adni, és így tulajdonosi kategóriánként képződik három oktális számjegy, ezeket kell beírni. Ha például azt akarjuk, hogy a *file1* fileunkat a tulajdonos tudja olvasni, írni, végrehajtani, a csoporttagok végrehajtani és olvasni, a többiek pedig csak olvasni, akkor a jogosultságok kódolása $4+2+1$, $4+1$, 4 , azaz `754` lesz:

```
$ chmod 754 file1
$ ls -l file1
-rwxr-xr-- 1 tsim1234 student 27 2013-03-17 15:56 file1
```

Speciális jogok beállítását is tartalmazó példa:

```
$ chmod 4751 file1
$ ls -l file1
-rwsr-x--x 1 tsim1234 student 27 2013-03-17 15:56 file1
```

A másik megadási mód a *szimbolikus* beállítás, ennek a következő a szintaxisa (a *who*, *op* illetve *perm* között a szóköz csak a láthatóság miatt szerepel):

```
chmod [-R] who op perm file ...
```

ahol *who* a tulajdonosi kategóriát adja meg, lehetséges értékei 'u' (tulajdonos, user), 'g' (csoport, group), 'o' (egyéb, others), illetve 'a' (mindenki, all), ami az előző hármat magában foglaló alapértelmezés.

perm a megfelelő művelet, 'rwxst' lehet a már látott módon.

op értéke += lehet. '+' a megfelelő jog engedélyezését jelenti, '-' a jog letiltását, '=' pedig a jog abszolút értékre állítását. Néhány példa:

```
$ ls -l file1
-rw-rw-rw- 1 tsim1234 student 27 2013-03-17 15:56 file1
$ chmod 754 file1
$ ls -l file1
-rwxr-xr-- 1 tsim1234 student 27 2013-03-17 15:56 file1
$ chmod u-w file1 # tulajdonosnak írásvédett
$ ls -l file1
-r-xr-xr-- 1 tsim1234 student 27 2013-03-17 15:56 file1
$ chmod a+x file1 # mindenkinek végrehajtható
$ ls -l file1
-r-xr-xr-x 1 tsim1234 student 27 2013-03-17 15:56 file1
$ chmod u=rwx,g=rx,o=r file1
$ ls -l file1
-rwsr-xr-- 1 tsim1234 student 27 2013-03-17 15:56 file1
```

Katalógusfile és inode

A fizikai file-ok adatait (a név kivételével) az *inode tábla* tartalmazza (i-bög). Minden fizikai file-nak megfelel egy (és csak egy) inode.

Egy katalógusállomány a katalógusban szereplő minden file-hoz tartalmaz egy bejegyzést. Egy katalógus bejegyzés csak a file nevét és inode számát tartalmazza, amint azt az 1.1 ábra szemlélteti:

állománynév (tetszőleges hosszúságú)	inode szám
--------------------------------------	------------

1.1 ábra Egy katalógus bejegyzés szerkezete

Az inode szám kilistázható az `ls -li` paranccsal. Az inode szám meghatározza az állományt leíró inode-ot.

Egy inode mérete 64 vagy 128 byte (állományrendszerenként különbözik). Egy inode az alábbi információkat tartalmazza az állománnyal kapcsolatban:

- tulajdonosát
- csoportját
- hozzáférési jogait
- hosszát
- létrehozás és utolsó módosítás dátumát
- típusát
- láncszámát – hány különböző katalógusbemenet hivatkozik ugyanarra az állományra

- mutatókat a file által lefoglalt blokkokra (lásd később, az 1.1.3. alfejezetben részletesebben)

Láncolás (link)

Bizonyos esetekben szükség lehet arra, hogy az állományrendszer egy részét több felhasználó megosztva használhassa, például ha egy adatbázishoz többen is szeretnének hozzáférni. A Unix alapú állományrendszerek lehetővé teszik, hogy ugyanazt az állományt több néven is elérhessük. Ezt nevezzük *láncolás*nak. A láncolás kitűnően használható névütközések feloldására, illetve helytekarékosság szempontjából is hasznos lehet.

Kétféle láncolást különböztetünk meg: *merev láncolás* (hard link), illetve *szimbolikus láncolás* (soft link).

*Merev láncolás*kor egy új katalógus bejegyzést hozunk létre, amely az eredeti inode-ra mutat és növeljük az inode-ban a láncszámot. Csak közönséges állományokra alkalmazható. A *láncszám* megadja, hogy hány helyről hivatkozunk ugyanarra a file-ra. Az új file-hivatkozás teljesen egyenértékű az eredetivel (pl. amennyiben módosítjuk az állományt a hard linkkel hivatkozva rá, láthatjuk, hogy az eredeti névvel hivatkozott állomány is módosult).

*File törlések*kor töröljük a directory bemenetet és csökkentjük az inode-ban a láncszámot; ha a láncszám értéke 0 lesz, akkor az inode bejegyzést is töröljük (a file többet nem elérhető).

Pl. Hard link létrehozására:

```
$ ln regi ujlink
$ ls -l
total 8
2098858 -rw-r--r-- 2 tsim1234 student 19 2013-03-17 19:26 regi
2098858 -rw-r--r-- 2 tsim1234 student 19 2013-03-17 19:26 ujlink
```

Láthatjuk, hogy az állományrendszerben két egyenértékű állomány jött létre: a régi neve *regi*, a létrehozott új állományé pedig *ujlink*. Mindkét katalógusbemenet ugyanarra az inode-ra mutat, illetve mindkét állománynál láthatjuk, hogy két helyről történik rá hivatkozás (a láncszám 2).

Hard linket kizárólag ugyanazon az állományrendszeren belül hozhatunk csak létre.

Szimbolikus láncolás (soft link) esetén az új katalógus bejegyzés nem a file inode-jára mutat, hanem egy speciális file-ra, ami tartalmazza a láncolt file nevét. `ln -s` paranccsal hozható létre. A létrehozott file típusa `l` lesz.

```
$ ln -s file1 szimbolikus
$ ls -l
total 8
-rw-r--r-- 1 tsim1234 student 27 2013-03-17 15:56 file1
lrwxrwxrwx 1 tsim1234 student 4 2013-03-17 19:34 szimbolikus -> file1
```

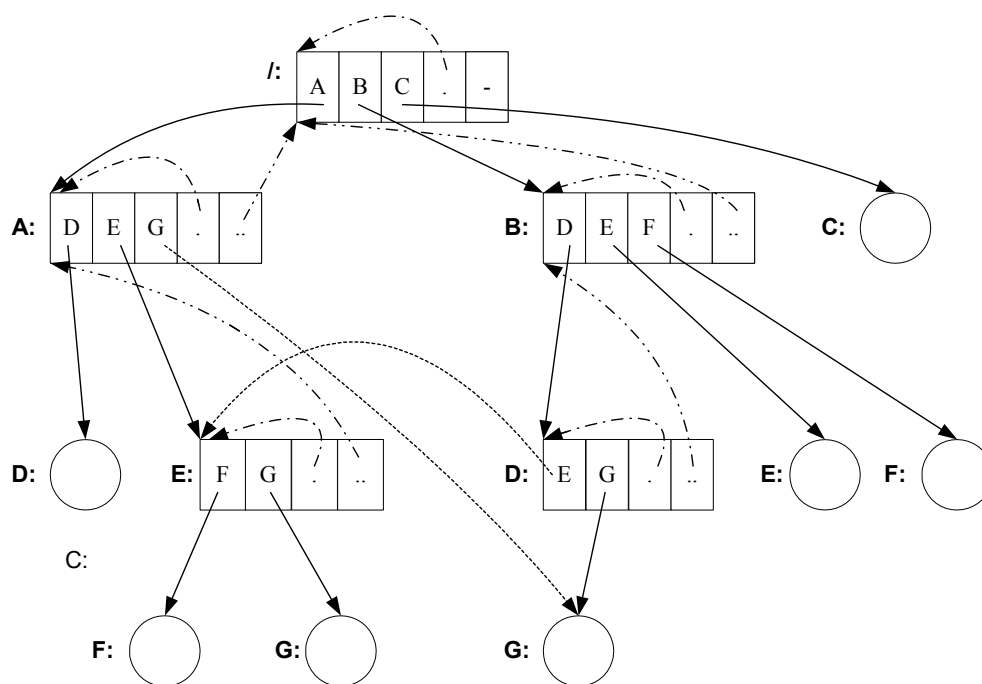
Láthatjuk, hogy a láncszám értéke az eredeti állománynál változatlan. A legtöbb művelet a lánc helyett az eredeti állományon hajtódik végre, kivéve pl. az `mv` és `rm` parancsokat.

A szimbolikus láncnak a hozzáférési jogait nem lehet módosítani, mivel az eredeti állomány jogai számítanak.

Az eredeti állomány törlésekor a lánc megmarad, de érvénytelenné válik.

A szimbolikus láncolás lehetővé teszi katalógus, illetve különböző fílerendszerben levő fileok láncolását is.

Az állományoknak a merev- vagy szimbolikus láncokkal együtt egy *faszerkezet* feleltethető meg. A faszerkezet lényege, hogy bármelyik állomány vagy katalógus egyetlen szülővel rendelkezik. Ebből adódóan bármelyik katalógusról vagy állományról legyen szó, ennek a gyökértől kezdődően egyetlen elérési út (path) felel meg. A katalógus vagy állomány és ennek szülőkatalógusa közötti kapcsolatot *természetes kapcsolat*nak nevezzük. Ez a kapcsolat automatikusan létrejön az alkatalógus vagy állomány létrehozásakor.



1.2 ábra Állományrendszer. Egyszerű példa.

Az 1.2 ábrán egy egyszerű állományrendszerre láthatunk példát. Az ábécé nagy betűivel közönséges állományokat, katalógusokat, illetve láncokat jelöltünk. Természetesen lehetőség van arra, hogy ugyanazt a nevet használjuk az állományrendszer különböző pontjain, hiszen a katalógusszerkezeten belül az elérési úttal együtt egyértelműen meghatározható, hogy melyik állományról van szó.

A közönséges állományokat körökkel jelöltük, a katalógusokat pedig téglalappal.

A kapcsolatokat háromféle nyíl jelöli:

- Folytonos vonal – természetes kapcsolat
- Szaggatott vonal – a saját katalógus, illetve szülőkatalógus esetén
- Pontozott vonal – szimbolikus vagy merev lánc.

A fenti példában 12 csomópontot (közönséges állomány vagy katalógus) különböztetünk meg.

Feltételezzük, hogy a pontozott vonallal jelölt két lánc szimbolikus lánc. A kényelem kedvéért a szimbolikus láncokat az elérési út legvégén szereplő betű alapján neveztük el. A két lánc létrehozása pl. az alábbi parancsok segítségével történhet:

```
cd /A
ln -s /B/D/G G      Az első lánc létrehozása
cd /B/D
```

ln -s /A/E E

A második lánc létrehozása

Feltételezzük, hogy az aktuális katalógus éppen a B. Úgy fogjuk bejárni a fát, hogy előbb a katalógust, majd az alkatalógusait járjuk be balról jobbra. Az alábbi 12 sor mind a 12 csomópontot érinti. Amennyiben többféleképpen is hivatkozhatunk ugyanarra a csomópontra, az egyenértékű hivatkozások ugyanabban a sorban jelennek meg. A szimbolikus linket is használó hivatkozásokat aláhúztuk.

/	..			
/A	../A			
/A/D	../A/D			
/A/E	../A/E	<u>D/E</u>	<u>../D/E</u>	
/A/E/F	../A/E/F	<u>D/E/F</u>	<u>../D/E/F</u>	
/A/E/G	../A/E/G	<u>D/E/G</u>	<u>../D/E/G</u>	
/B	.			
/B/D	D	./D		
/B/D/G	D/G	./D/G	<u>/A/G</u>	<u>../A/G</u>
/B/E	E	./E		
/B/F	F	./F		
/C	../C			

III.1.3. A UNIX logikai lemez szerkezete

A különböző Unix disztribúciók megjelenésével elkerülhetetlenné vált a különböző filerendszerek megjelenése, melyek főképp az egyes disztribúciókra jellemzőek. Például:

- A Solaris az `ufs` állományrendszert használja;
- A Linux előszeretettel használja az `ext2` illetve `ext3` filerendszereket;
- Az IRIX sajátja az `xfs`;

stb.

Minden egyes Unix alapú filerendszernek vannak bizonyos sajátos paraméterei (az illető állományrendszerre jellemző konstans értékek), mint pl.: egy blokk mérete, egy inode mérete, a lemezen tárolt adatokat meghatározó cím hossza, hány direkt címet tartalmaz az inode és hány hivatkozás szerepel a indirect címek listájában. Ezen konstansok értékétől függetlenül, egy új állomány bejegyzése, illetve ennek az adataihoz való hozzáférés, hasonló elvek alapján történik.

Mount

A Unix állományrendszer egységes filerendszer, az elérési út nem tartalmaz lemezegység nevet. A különböző logikai vagy fizikai lemezen levő filerendszert becsatoljuk (`mount`) a rendszerbe. Egy üres directory-hoz csatlakoztatható az új filerendszer, ennek gyökér katalógusára az eredetileg üres directory nevével hivatkozhatunk. A felhasználó számára észrevétlen, hogy mi melyik filerendszerben van.

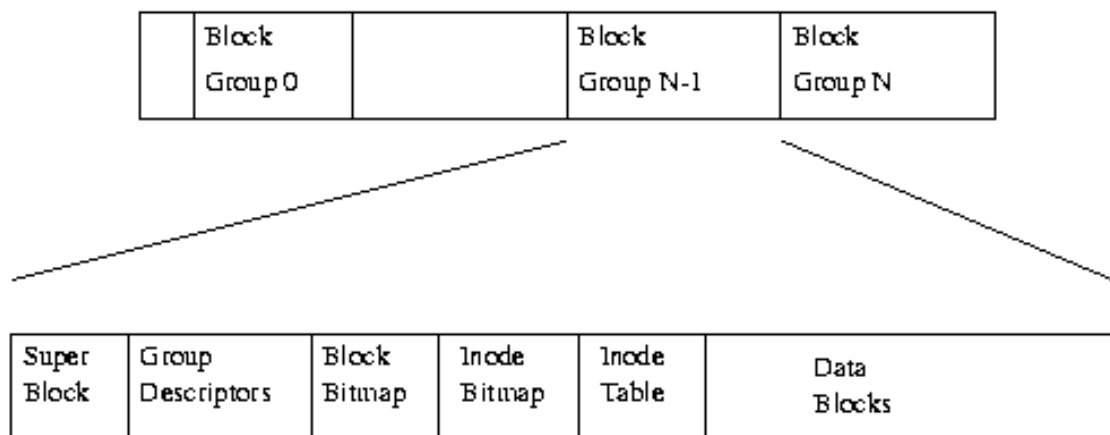
Logikai lemezek és blokkok

Az alábbiakban az `ext2` állományrendszer jellemzőit vesszük alapul.

Íme néhány fontosabb jellemző:

A lemez és memória közötti adatátvitel alapegysége a blokk. Azonos méretű blokkokat használ a rendszer. Egy blokk mérete –ami egyébként változó lehet–, a rendszer generálásakor állítható be (`mkfs`). Az állományok nyilvántartása az inode táblázat segítségével történik. A katalógus a fileok neve és inode száma között hoz létre kapcsolatot. A directory is egy file.

Az ext2 filerendszerben a tárolóhely *blokkokra* van felosztva, ezek pedig *blokk csoportokat* alkotnak. A rendszer számára kritikus információk ismétlődnek minden csoportban, amint azt az 1.3 ábra szemlélteti:



1.3 ábra Logikai lemez szerkezete

Egy bizonyos állomány adatai tipikusan ugyanazon a blokkcsoporton belül foglalnak helyet, amennyiben ez lehetséges. Ez azért jelentős, mivel hosszú, összefüggő adatsorozatot beolvasásakor minimalizálja a lemezhozzáférések számát.

Minden egyes blokk-csoport tartalmazza az ún. *szuperblokk* (super block) másolatát, egy csoport deskriptort (group descriptor), egy blokk bittérképet (block bitmap), egy inode bittérképet (inode bitmap), egy inode táblát (inode table), végül pedig a tulajdonképpeni adatokat tartalmazó blokkokat.

A *szuperblokk* az operációs rendszer bootolásához szükséges fontos információt tartalmaz, emiatt minden blokkcsoport tartalmaz egy biztonsági másolatot róla. Ennek ellenére tipikusan csak a filerendszer legelső blokkjában szereplő adatokat használja a rendszer bootoláskor.

A szuperblokk a következő információkat tartalmazza:

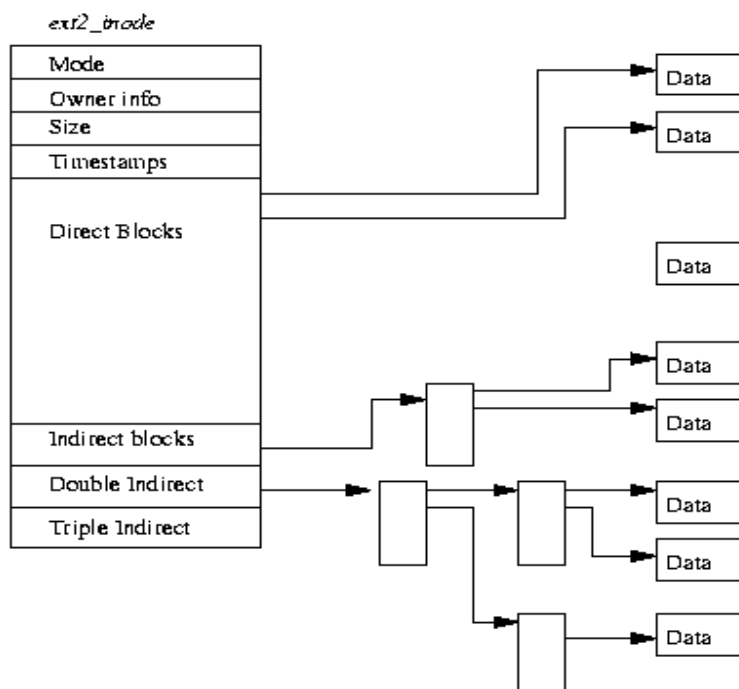
- **Magic Number** – `0xEF53` – ext2 esetén.
- **Revision Level** – verzió szám
- **Mount Count and Maximum Mount Count** – a filerendszer teljes ellenőrzése ajánlott, ha eléri max-ot
- **Block Group Number** – a blokkcsoport száma, amelyikben ez a szuperblokk van,
- **Block Size** – blokk mérete byte-okban
- **Blocks per Group** – blokkok száma egy csoportban
- **Free Blocks** – szabad blokkok a filerendszerben
- **Free Inodes** – szabad inode-ok a filerendszerben
- **First Inode** – első inode

A *csoport deszkriptor* minden egyes blokk csoport esetén az alábbi információt tartalmazza:

- **Blocks Bitmap** – a „block allocation bitmap” blokk száma
- **Inode Bitmap** – az „inode bitmap” blokk száma
- **Inode Table** – az inode tábla kezdő blokkjának a száma
- **Free blocks count, Free Inodes count, Used directory count** – azaz szabad blokkok, szabad inode-ok, illetve használt direktorybemenetek száma

Egy állományhoz tartozó blokkok nyilvántartása

Amint láthattuk, egy állománnyal kapcsolatos információk az illető állományt leíró inode-ban szerepelnek. Az inode az állomány különböző jellemzői mellett az illető állományhoz tartozó adatblokkokat azonosító mutatókat tartalmaz, az 1.4 ábrán szemléltetett logika szerint:



1.4 ábra Egy állományhoz tartozó adatblokkok nyilvántartása

Azt ext2 állományrendszer konkrétan 12 direkt blokkra mutató címet tartalmaz (az állomány első 12 blokkjára tehát közvetlen hivatkozást tartalmaz), Ezt követi egy indirektáló blokkra vonatkozó mutató (mely további közvetlen adatblokkokra vonatkozó mutatókat tartalmaz), majd egy kétszeres indirektáló blokkra, végül pedig egy háromszoros indirektáló blokkra vonatkozó mutató következik.

Egy állomány tetszőleges adatblokkjához való hozzáférés legtöbb 4 lemezhozzáférést igényel. Rövid állományok esetében azonban ennél lényegesen kevesebb hozzáférésre van szükség (hiszen az első 12 blokk adatai közvetlenül elérhetőek). Mindaddig amíg az állomány meg van nyitva, ennek inode-ja be van töltve a belső memóriába.

III.2. Unix folyamatok

Unix folyamatok: létrehozás, fork, exec, exit, wait; kommunikáció pipe illetve FIFO állományon keresztül.

III.2.1. A folyamatkezelést szolgáló fontosabb rendszerhívások

Ebben az alfejezetben a folyamatkezeléshez szükséges legfontosabb rendszerhívások működését mutatjuk be: `fork`, `exit`, `wait` és `exec*`. Kezdjük a folyamat létrehozásáért felelős `fork()` rendszerhívással.

Unix folyamatok létrehozása. A `fork` rendszerhívás.

A Unix operációs rendszerben egy új folyamat létrehozása a `fork()` rendszerhívással történik. Ennek szintaxisa:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Sikeres végrehajtás esetén ennek hatása a következő:

- új folyamatábra bemenet jön létre, melynek tartalma a szülőtől lesz átmásolva
- az adat és veremszegmens duplázva lesz
- mindkét folyamat esetén egy-egy mutató a közös kódszegmensre mutat
- a gyerek örökli a szülőtől a megnyitott állományokat
- a `fork` utáni utasítástól egymástól függetlenül dolgozik a szülő és a gyerek folyamat ugyanazzal a kódszegmennsel

Az újonnan létrehozott folyamatot *gyerekfolyamatnak*, a `fork()` hívást végrehajtó folyamatot pedig *szülőfolyamatnak* nevezzük. Leszámítva, hogy külön adat-, illetve veremszegmessel rendelkeznek, a gyerekfolyamat csupán az alábbiakban különbözik a szülőtől: azonosítója (PID), a szülő azonosítója (PPID), a `fork` hívás visszatérített értéke (sikeres végrehajtás esetén ugyanis a `fork` a rendszerhívást végrehajtó szülőfolyamatban a gyerekfolyamat pid-jét, a gyerekfolyamatban pedig 0-t térít vissza).

A szülőfolyamat azonosítóját, illetve magának a folyamatnak az azonosítóját az alábbi rendszerhívások segítségével kérdezhetjük le:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid(void); //PPID lekérdezése
pid_t getpid(void);  //PID lekérdezése
```

Az 1.5 ábra szemlélteti a `fork` működési mechanizmusát.

Hiba esetén a `fork` -1-et térít vissza, természetesen az `errno` változó megfelelőképpen be lesz állítva, a hiba okát jelezve. Hiba léphet fel a `fork` hívás kapcsán, amennyiben:

- nincs elég szabad memóriaterület, hogy a szülő képeinek másolata létrejöhessen;
- a folyamatok száma meghaladja a megengedett maximális értéket.

A `fork` hívás fentebb leírt viselkedése lehetővé teszi, hogy a szülő, illetve gyerekfolyamat párhuzamos működését a következőképpen adjuk meg:

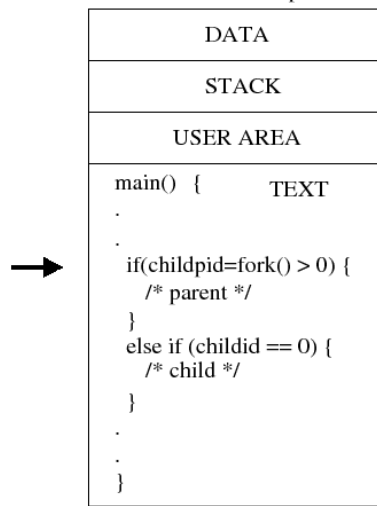
```

pid = fork();
if (pid == 0)
{
    /* gyerek folyamat */
}
else
{
    /* szülő folyamat */
}

```

BEFORE

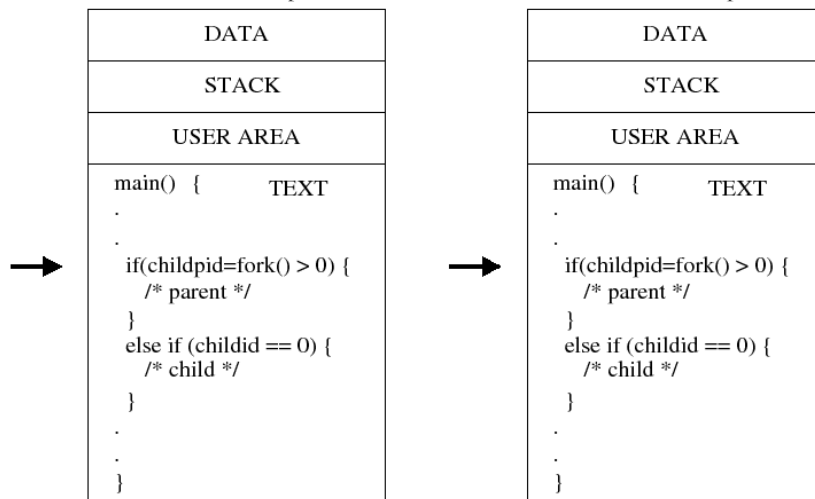
pid: 12791



AFTER

pid: 12791

pid: 12793



1.5 ábra Fork mechanizmus

Ugyanez hibakezeléssel együtt a következőképpen néz ki:

```

switch (fork())

```

```

{
    case -1:
        perror(„fork”);
        exit(1);
    case 0:
        /* gyerek folyamat */
    default:
        /* szülő folyamat */
}

```

A alábbi program a fork használatát példázza:

```

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
int main() {
    int pid,i;
    printf(„\nProgram kezdete:\n”);
    if ((pid=fork())<0){
        perror(„fork() hiba\n”);
        exit(1);
    }
    if (pid==0){//gyerekfolyamat
        for (i=1;i<=10;i++){
            sleep(2);          // 2 másodpercnnyi várakozás
            printf(„\t %d SZULO %d GYEREKE:3*d=%d\n”,getppid(),getpid(),i,3*i);
        }
        printf(„GYEREK vege\n”);
    }
    else{// pid>0 szülőfolyamat
        printf(„Lerehoztam a %d GYEREKket\n”,pid);
        for (i=1;i<=10;i++){
            sleep(1);          //1 másodpercnnyi várakozás
            printf(„%d SZULO: 2*d=%d\n”,getpid(),i,2*i);
        }
        printf(„SZULO vege\n”);
    }
}

```

Szándékosan írtuk úgy a kódot, hogy a gyerekfolyamatnak hosszabb ideig kelljen várakoznia, mint a szülőnek (komplex számítások végzése közepette gyakran megtörténik, hogy az egyik folyamat által végzett műveletek hosszabb időbe telnek, mint a másik folyamat esetében). Ennek következtében a szülő hamarabb befejeződik. A kapott eredmények a következők:

```

Program kezdete:
Lerehoztam a 30584 GYEREKket
30583 SZULO: 2*1=2
      30583 SZULO 30584 GYEREKE:3*1=3
30583 SZULO: 2*2=4
30583 SZULO: 2*3=6
      30583 SZULO 30584 GYEREKE:3*2=6
30583 SZULO: 2*4=8
30583 SZULO: 2*5=10
      30583 SZULO 30584 GYEREKE:3*3=9
30583 SZULO: 2*6=12
30583 SZULO: 2*7=14
      30583 SZULO 30584 GYEREKE:3*4=12

```

```

30583 SZULO: 2*8=16
30583 SZULO: 2*9=18
          30583 SZULO 30584 GYEREKE:3*5=15
30583 SZULO: 2*10=20
SZULO vege
      1 SZULO 30584 GYEREKE:3*6=18
      1 SZULO 30584 GYEREKE:3*7=21
      1 SZULO 30584 GYEREKE:3*8=24
      1 SZULO 30584 GYEREKE:3*9=27
      1 SZULO 30584 GYEREKE:3*10=30
GYEREK vege

```

Az exit és wait hívások

Egy program befejezése az alábbi rendszerhívások segítségével történhet:

- **ANSI C**
`#include <stdlib.h>`
`void exit(int exit_code);`
- **Posix**
`#include <unistd.h>`
`void _exit(int exit);`
- **Rendellenes befejezés**
`#include <stdlib.h>`
`void abort(void);`

Befejezés után a folyamat *zombie* állapotba kerül mindaddig, amíg a szülő egy `wait` függvénnyel le nem kérdezi a befejezési kódot. A zombie állapotban levő folyamat esetében a rendszer minden erőforrást felszabadít, kivéve a folyamatábla bemenetet. Amennyiben a befejezett folyamatot létrehozó szülőfolyamat már korábban véget ért, akkor az illető folyamat szülőfolyamata az 1-es folyamatazonosítójú speciális `init` folyamat lesz.

Az `init` folyamat mindig meghívja a `wait` függvényt.

A folyamat befejeződésekor a rendszer egy `SIGCLD` üzenettel értesíti a szülőfolyamatot.

A szülő bevárhatja valamelyik gyerek befejeződését: `wait`, `waitpid` függvények egyikét használva. Ezek hatására:

- várakozhat (ha minden gyereke fut),
- érzékelheti, hogyha egy gyerek befejeződött,
- visszatéríthet hibát (ha nincs gyereke)

A `wait` illetve `waitpid` hívások szintaxisa a következő:

```

#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int opt);

```

Különbségek a `wait` és a `waitpid` között:

- a `wait` felfüggeszti a hívó folyamatot, amíg a gyerek befejeződik, ezzel szemben a `waitpid` egy külön opciót kínál fel (`opt`), melynek használatával a felfüggesztés elkerülhető,
- a `waitpid` nem mindig az első fiú befejezéséig vár, hanem a `pid` változóban megadott azonosítójú gyerek befejezéséig,
- a `waitpid` az `opt` argumentum segítségével engedélyezi a programok vezérlését.
- A `wait` függvény visszatérési értéke azon gyerekfolyamat azonosítója, amely éppen befejeződött.

- A `waitpid` `-1` értéket térít vissza, ha nem létezik a `pid`-ben megadott azonosítójú folyamat, vagy az nem gyereke a hívó folyamatnak.

A `waitpid` függvényhívásnál megadható `pid` változó lehet:

- `pid = -1` – bármely gyerekre várakozhat; ekvivalens a `wait`-tel,
- `pid > 0` – a `pid` azonosítójú folyamatra várakozik,
- `pid = 0` – bármely olyan folyamatra várakozik, amelynek a csoportazonosítója megegyezik a hívó programéval,
- `pid < -1` – bármely olyan folyamatra várakozik, amelynek a csoportazonosítója megegyezik a megadott érték abszolút értékben.

Külső program végrehajtása; az `exec` függvénycsalád

A legtöbb más operációs rendszerhez hasonlóan a Unix is biztosít lehetőséget arra, hogy elindítsunk egy programot egy másikkól. Ezt a mechanizmust az `exec*` függvénycsalád teszi lehetővé. Amint látni fogjuk, a `fork` illetve `exec*` rendszerhívások kombinálása nagyfokú rugalmasságot biztosít a folyamatkezelést illetően.

Az `exec*` függvénycsalád

- az aktív folyamat kódját egy másikkal helyettesíti (betölt egy új programot)
- új kód, adat és veremsejgmens jön létre, a régieket felszabadítja
- a folyamatábla bemenetet örökli az eredeti folyamattól

Az `exec*` utáni utasítás csak hiba esetén hajtódik végre.

Az 1.1 táblázat összegzi az `exec*` függvénycsaládba tartozó rendszerhívásokat és ezek jellemzőit (három kritérium szerint hat függvényt kínál fel a rendszer):

Függvény	paraméter	keresési út	környezet
<code>execl</code>	lista	<code>./</code>	marad
<code>execv</code>	tömb	<code>./</code>	marad
<code>execlp</code>	lista	<code>PATH</code>	marad
<code>execvp</code>	tömb	<code>PATH</code>	marad
<code>execle</code>	lista	<code>./</code>	változik
<code>execve</code>	tömb	<code>./</code>	változik

1.1 táblázat Az `exec*` függvénycsalád

Az egyes függvények szintaxisa:

```
#include <unistd.h>
int execl(const char *path,
```

```

        /* elérési út */
const char *arg0,
        /* programnév */
const char *arg1,
        /* paraméterek */
...
const char *argn,
NULL);
        /* a paraméterek vége */
int execv(const char *path, char *argv[]);
int execlp(const char *filename,
        /* a futtatható állomány neve */
const char *arg0,
const char *arg1
...
const char *argn,
NULL);
int execvp(const char *filename, char *argv[]);

int execl(const char *path,
const char *arg0,
const char *arg1,
...
const char *argn,
NULL,
char *envp[]);
        /* környezeti változók */
int execve(const char *path, char *argv[], char *envp[]);

```

Az egyes változók jelentése:

- path: mutató egy karaktersorhoz, amely a futtatható állomány keresési útvonalát jelöli,
- filename: mutató a futtatható állomány nevéhez; ha a név nem kezdődik a gyökérrel (és nincs megadva a teljes útvonal), akkor az állományt a PATH változó által definiált katalógusokban keresi a rendszer,
- arg0: mutató a futtatható állomány nevéhez,
- arg1, arg2, ..., argn: mutatók, amelyek a programnak átadott paramétereket jelölik,
- argv: mutató a paramétervektorhoz (a 0-dik paraméter az állomány neve),
- envp: mutató az új környezeti változókhoz, amelyek a vektorban egyenként változó=érték alakban jelennek meg.

Az utolsó paraméter mindig NULL (a paraméterlista végét jelöli).

III.2.2. Folyamatok közti kommunikáció pipe-on keresztül

A pipe mechanizmus

A pipe mechanizmus megjelenését a Unix alapú rendszerekben az indokolta, hogy lehetővé tegye a gyerekfolyamat szülővel való kommunikációját.

Általában a szülő folyamat átirányítja a standard kimenetét (stdout) egy *pipefile*-ba, a gyerekfolyamat pedig a standard bemenetét (stdin) veszi ugyanabból a pipefileből. Az ilyen jellegű kapcsolat jelölésére shell szinten a “|” operátort szokás használni.

Pl. `who | sort | more`

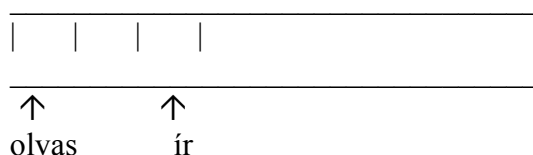
A pipe mechanizmus ugyanakkor C programból is alkalmazható.

A *pipefile* egy speciális név nélküli file (nem tartozik hozzá directory bemenet). Mérete korlátozott, általában 10 (12) blokk.

Az 1.1.3 alfejezetben láthattuk, hogy az inode táblázat egy bemenete 13 (15) címet tartalmaz, amiből 10 (12) direkt cím, majd ezt követi egy egyszeres, egy kétszeres, illetve egy háromszoros indirektáló cím.

Pipefile esetén nincs indirektálás, emiatt az adathozzáférés (egy indirektálást is használó közönséges állományhoz képest) gyors.

A két folyamat (szülő-, illetve gyerekfolyamat) közösen használja a pipefilet: egyik ír, a másik olvas – megnyitáskor két deszkriptort kapunk vissza, egyet írásra, és egyet olvasásra.



Az adatok olvasása/írása a pipefileba úgy történik, mint egy körkörös pufferbe (ha az betelt, kezdődik az elejéről). Az adatok olvasása/írása FIFO elv alapján történik (a legrégebben beírt adat lesz leghamarabb kiolvasva). Egy bizonyos információt csak egyszer lehet kiolvasni. A szinkronizálást a filemutatók közt a rendszer végzi, mégpedig a termelő/fogyasztó elv alapján:

- egy folyamat, amelyik írni akar a pipefile-ba (termelő) csak akkor fog tudni írni (termelni), ha az nem telt meg (amennyiben meg van telve, várakozási állapotba jut, amíg egy másik folyamat ki nem olvas belőle).
- a folyamat, amelyik olvas (fogyasztó) csak akkor olvashat, ha van mit. Különben blokálva lesz (wait állapot), míg egy másik folyamat adatot nem helyez a pipefile-ba.
- a pipefile adataihoz csak szekvenciálisan lehet hozzáférni

Pipe mechanizmus a gyakorlatban

A szülőfolyamat hozza létre a pipefile-t (pipe). Ugyanaz a szülő létrehoz egy vagy több gyerek-folyamatot (`fork` rendszerhívás). Egyes folyamatok írni fognak a pipefile-ba (`write - fd[1]`), mások pedig olvasni (`read - fd[0]`). Elvileg a szülő- és gyerekfolyamat is megkapja az író- és olvasó deszkriptort is, egyetlen pipefile-t mégis csupán egyirányú kommunikációra szokás használni (a nem használt deszkriptorokat zárjuk be!). Fontos, hogy a szülő-gyerek közti kommunikációt szolgáló pipefile-t még a `fork` hívás előtt hozzuk létre, hiszen így a `fork` hívást követően a gyerekfolyamat öröklí a megnyitott deszkriptorokat. Az 1.6 ábra szemlélteti a pipefile-on keresztül történő kommunikációt.



1.6 ábra Kommunikáció szülő és gyerek között pipefile-on keresztül

Kétirányú kommunikáció megvalósításához két pipefile létrehozására van szükségünk.

Pipe létrehozása

A pipefile létrehozása a `pipe` rendszerhívással történik. Ennek szintaxisa:

```
#include <unistd.h>
int pipe(int pfd[2]);
```

A függvény 0-t térít vissza, ha a létrehozás sikerült, és -1-et, ha nem. A `pfd` egy két elemű táblázat, ahol a `pfd[0]`-ból olvasunk, és a `pfd[1]`-be írunk. A `pfd[1]`-be való írás során (write) az adatok a pipe fileba kerülnek, míg a `pfd[0]`-ból olvasva (read) törlődnek onnan. Hiba esetén az `errno` változó a hiba kódját fogja tartalmazni.

Pipe bezárása

A nem használt pipe végeket ajánlatos minél előbb bezárni! Ez a `close` rendszerhívással történik, melynek szintaxisa:

```
#include <unistd.h>
int close(int pfd);
```

A függvény 0-t térít vissza, ha a bezárás sikerült, és -1-et különben. A `pfd` argumentum egy egész szám, tehát csak az állomány egyik végét zárja be.

Pipe írása, olvasása

A pipefile-ba való írás, illetve a beírt adatok kiolvasása az alábbi függvények valamelyikének segítségével történhet:

```
#include <unistd.h>
ssize_t read(int pfd, void *buf, size_t count);
ssize_t write(int pfd, const void *buf, size_t count);
```

vagy

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

A második változatot főként standard fileok esetén használjuk. A pipe fileok kezelésére a `read` és `write` függvényeket ajánljuk. Paraméterként meg kell adnunk a pipe file egyik végének azonosítóját (`pfd`), a `buf` puffer vagy érték, míg a `count` változóba ennek méretét adjuk meg. A függvények visszatérített értéke a pipe-ból sikeresen kiolvasott (beírt) bájtok száma. Korábban említettük, hogy amennyiben üres pipefileből próbálunk olvasni, a folyamat blokáldik a `read` műveleten mindaddig, amíg valaki nem ír a pipe-ba. Fontos azonban megjegyezni, hogy amennyiben a pipefilehoz tartozó összes (!) íródeszkriptort bezártuk, a `read` művelet azonnal visszatér 0 értékkel.

Példa: `who | sort` implementálása pipe illetve `exec*` hívások segítségével

Tekintsük az alábbi összetett shell parancsot:

```
$ who | sort
```

Az alábbi példa a két parancs (`who` és `sort`) pipe-on keresztül történő összefűzését valósítja meg. A szülőfolyamat (mely a shell parancsértelmezőt helyettesíti) két gyerekfolyamatot hoz

létre, ezek pedig megfelelőképpen átirányítják a bemenetüket, illetve kimenetüket. Az első gyerekfolyamat a `who` parancsot hajtja végre, a másik pedig a `sort` parancsot, a szülőfolyamat pedig megvárja a befejeződésüket. A forráskód a következő:

```
//whoSort.c
//a $who|sort shell parancsok összefüzeset valósítja meg pipe segítségével
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main (){

    int p[2];
    pipe (p);
    if (fork () == 0) {          // első gyerek
        dup2 (p[1], 1);          // standard kimenet átirányítása
        close (p[0]);
        execlp ("who", "who", NULL);
    }
    else if (fork () == 0) {     // második gyerek
        dup2 (p[0], 0);          // standard bemenet átirányítása
        close (p[1]);
        execlp ("sort", "sort", NULL); // sort végrehajtása
    }
    else {                       // szülő
        close (p[0]);
        close (p[1]);
        wait (NULL);
        wait (NULL);
    }
    exit(0);
}
```

Megjegyzés: a fenti példa jobb megértéséhez ajánljuk, hogy az olvasó nézzon utána a Unix kézikönyvekben (`man`) a `dup2` rendszerhívás működésének. Esetünkben a `dup2` egyik paramétere egy pipefile deskriptor.

III.2.3. Folyamatok közti kommunikáció FIFO állományon keresztül

A FIFO mechanizmus

A pipe mechanizmus legnagyobb hátránya, hogy csak egymással „rokoni” viszonyban levő folyamatok között használhatjuk: a pipe-on keresztül kommunikáló folyamatok a pipe-ot létrehozó folyamat leszármazottai kell legyenek, hiszen az író-, illetve olvasó deskriptor egyedi, és mindkettő a `fork()` hívás következtében adódik át a gyerekfolyamat(ok)nak.

Az 1985-ös év tájékán jelent meg a *FIFO (névvel ellátott csővezeték vagy pipe)* állomány (Unix System V). A FIFO állomány a közönséges fájl és a pipe kombinációja. A pipe-al szemben a FIFO állománynak van egy szimbolikus neve, és egy katalógus, ahová létrehozzuk, ezt leszámítva, azonban megőrzi a pipe fájlok összes jellemzőit. A FIFO állománynak saját neve van, tehát bármely folyamat meg tudja nyitni, nem csak a közös össel rendelkező folyamatok. Amennyiben az `ls -l` paranccsal kilistázzuk az állományt, a file típusát p-vel (pipe) jelöli a rendszer.

Egy FIFO állomány létrehozása az `mknod` vagy `mkfifo` függvények valamelyikével történik.

Szintaxis:

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod(char *pathname, int mode, 0);
int mkfifo(const char *pathname, mode_t mode);
```

ahol:

- `pathname` – elérési útvonal
- `mode` – típus és hozzáférési jogok (pl. `S_IFIFO|0666`)
- visszatérített érték:
 - 0 sikeres létrehozás esetén
 - -1 hiba esetén

Shell paranccsal is létrehozhatunk FIFO állományt:

```
$ mknod FIFOnev p
```

vagy

```
$ mkfifo FIFOnev
```

FIFO állomány megnyitása

A FIFO állomány megnyitása az `open` rendszerhívással történik. Szintaxisa:

```
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *pathname, int flags);
```

ahol

- `pathname` – elérési útvonal
- `flags` – hozzáférési jogok
 - `O_RDONLY`, csak olvasható,
 - `O_WRONLY`, csak írható,
 - `O_RDWR`, olvasható és írható.
 - `O_NONBLOCK`, `O_NDELAY` – nincs várakozás (lásd az 1.2 táblázatot)
- visszatérített érték:
 - file leíró – sikeres megnyitás esetén
 - -1 – hiba esetén

Az írás, olvasás, bezárás ugyanúgy történik, mint a közönséges állományok esetén (`read`, `write`, illetve `close` függvények). A FIFO állomány törlése pedig az `unlink` hívással történik. Szintaxisa:

```
#include <unistd.h>
int unlink(const char *pathname);
```

A FIFO állomány használata a következő forgatókönyv szerint történik:

Egy folyamat a szimbolikus név alapján létrehozza a FIFO állományt az `mknod` vagy `mkfifo` függvények segítségével. Egy folyamat, amely információt szeretne közölni egy másikkal, megnyitja a FIFO állományt az `open` függvénnyel, és a `write` segítségével beírja az adatokat. Egy másik folyamat, amely az adatokat szeretné kiolvasni, megnyitja a FIFO állományt olvasásra az `open` függvénnyel, majd a `read` segítségével kiolvassa a kívánt információt. Egy folyamat a szimbolikus név alapján törli a FIFO állományt az `unlink` függvénnyel. Az állomány törlése a `rm` shell parancs segítségével is megtehető.

Az 1.2 táblázat összefoglalja, hogy mi történik a FIFO állomány megnyitásakor, valamint írás/olvasáskor, attól függően, hogy az O_NONBLOCK (O_NDELAY) flag be van-e állítva vagy sem.

feltételek	normál	O_NDELAY beállítva
<i>FIFO</i> megnyitva csak írásra, de olvasó folyamat nincs	várakozik mindaddig, amíg egy másik folyamat meg nem nyitja írásra a <i>FIFO</i> állományt	azonnal visszatér, várakozás és hibajelzés nélkül
<i>FIFO</i> megnyitása írásra, de olvasó folyamat nincs	várakozik mindaddig, amíg egy másik folyamat meg nem nyitja olvasásra a <i>FIFO</i> állományt	azonnal visszatér, hibajelzéssel: az <code>errno</code> értéke <code>ENXIO</code>
olvasás <i>FIFO</i> vagy pipe fileből, de nincs olvasnivaló adat	várakozik mindaddig, amíg adatok nem kerülnek a <i>FIFO</i> állományba, vagy amíg nincs egyetlen olyan folyamat sem, amely írásra nyitotta meg a <i>FIFO</i> állományt. A kiolvasott byte-ok számát téríti vissza, ha új adatok jelentek meg vagy 0-t, ha nincs több író folyamat.	azonnal visszatér és 0-t térít vissza
írás <i>FIFO</i> vagy pipe fileba, amikor az tele van	várakozik mindaddig, amíg ürül hely a <i>FIFO</i> állományban, majd annyi adatot ír bele, amennyi számára hely van	azonnal visszatér és 0-t térít vissza

1.2 táblázat Az O_NONBLOCK (O_NDELAY) flag hatása

Példa: kliens/szerver kommunikáció FIFO-n keresztül

A kliens/szerver modell gyakran használt a programozásban. A következőkben a kliens/szerver modellt mutatjuk be, ahol a kommunikáció *FIFO* állományon keresztül történik. A példában a szerver nagyon egyszerű feladatot lát el, hiszen célunk a kommunikáció bemutatása: a kliens küld egy számot a szervernek, mire a szerver válaszként visszaküldi a szám négyzetét.

Megjegyzések:

- a szerver létrehoz egy szerverfifot, amelyre az összes kliens csatlakozni fog,
- minden kliensnek külön *FIFO*-ja van, amelyet a kliens maga hoz létre; ezért amikor a kliens a szervernek elküldi a kérést, valahogyan jeleznie kell, hogy milyen nevű *FIFO*-n keresztül szeretné a választ megkapni; a legegyszerűbb, ha a kliens *FIFO*-jának nevében szerepel a kliens folyamataazonosítója is, így a név egyértelmű,
- a kliens előbb megnyitja a saját *FIFO*-ját olvasásra, s csak azután küldi el az üzenetet a szerver felé,
- a szerver *FIFO*-ja csak a szerver befejeződéskor záródik be,
- a kliens *FIFO*-ját a szerver oldalon a szerver a válaszadás után bezárja; ha újabb kérés érkezik, újból megnyitja,
- ha a kliens befejezte működését be kell zárnia a saját *FIFO*-ját.

Mivel a *FIFO*-n küldött adatok típusa megegyezik a szerverben és a kliensben, a könnyebb kezelhetőség érdekében ajánlatos egy közös adatszerkezetet létrehozni, és ezt egy külön fejlécállományban tárolni. Esetünkben ez a következő lesz:

Közös headerállomány (struktura.h)

```
typedef struct elem
{
    int szam;
    int pid;
} azon;
```

Szerver program (szerver.c):

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include "struktura.h"                /* a fent megadott fejlec */

int main(void)
{
    int fd, fd1;                      /* szerver- es kliensfifo */
    char s[15];                       /* kliensfifo neve; pl. fifo_143 */
    azon t;                           /* kuldeni kivant "csomag" */

    mkfifo("szerverfifo", S_IFIFO|0666); // a szerver létrehozza a saját
                                          // fifo-ját */
    fd = open("szerverfifo", O_RDONLY); /* megnyitja olvasásra */

    do                                /* amig 0-t nem küld egy kliens */
    {
        while(!read(fd, &t, sizeof(t))); /* szam kiolvasasa */
        t.szam = t.szam * t.szam;
        sprintf(s, "fifo_%d", t.pid);    // a pid segitsegevel meghat. a
                                          // kliensfifo nevet
        fd1 = open(s, O_WRONLY);          /* kliensfifo megnyitasa irasra */
        write(fd1, &t, sizeof(t));
        close(fd1);                      /* adatok elkuldve */
    } while (t.szam);
    close(fd);                           /* szerverfifo vege */
    unlink("szerverfifo");               /* torli a szerverfifot */
    exit(0);
}
```

Kliens program (kliens.c):

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include "struktura.h"                /* a fenti fejlecallomany */

int main(int argc, char *argv[])      // a szamot a parancssorban adjuk
{                                     // meg

    int fd, fd1;                      /* kliens- es szerverfifo */
    char s[15];
    azon t;
```

```

if (argc != 2)                                /* nincs megadva argumentum, hiba */
{
    printf("hasznalat: kliens <szam>\n");
    exit(1);
}

sprintf(s, "fifo_%d", getpid());               // meghat. a fifonevet a pid
                                              // segitsegevel
mkfifo(s, S_IFIFO|0666);                      /* létrehoz egy kliensfifot */
fd = open("szerverfifo", O_WRONLY);
t.pid = getpid();                             /* a kuldeni kivant adatok */
t.szam = atoi(argv[1]);                       /* string atalakitasa szamma */
write(fd, &t, sizeof(t));                     /* kuldi a szervernek */
fdl = open(s, O_RDONLY);
read(fdl, &t, sizeof(t));                     /* a valasz */
close(fdl);
unlink(s);                                     /* kliensfifo torlese */
printf("a negyzete: %d\n", t.szam);
exit(0);
}

```

III.3. Shell programozás és alapvető Unix parancsok

Parancsértelmező (Bourne shell - sh)

III.3.1. Egy parancsértelmező – shell – működése

A *parancsértelmező* (*shell* vagy *burok*) egy speciális program, mely egy interfészt biztosít a felhasználó illetve az operációs rendszer magja (az ún. *kernel*) között. Ebből a szemszögből kétféleképpen tekinthetünk a *shell*-re:

1. mint *parancs nyelv*re, mely közvetít a számítógép és a felhasználó között. Amint egy felhasználó bejelentkezik a rendszerbe és/vagy megnyit egy parancsablakot, implicit módon indul a *shell*, mint parancsértelmező. A *shell* egy prompt-ot ír ki a standard kimenetre (ami általában egy terminálhoz van hozzárendelve), arra várva, hogy a felhasználó parancsokat írjon be vagy valamilyen parancsállományt indítson el, esetleg paramétereket is megadva neki.
2. mint *programozási nyelv*re, melynek alapeleme a Unix *parancs* (szemantikailag a programozási nyelvek *hozzárendelés utasításával* tekinthető egyenértékűnek). A klasszikus programozási nyelvekből a *feltétel* igazságértékének megfelelője itt a parancsok sorozatából az utolsónak a *visszatérített értéke*: a 0 érték *igaz*-at (*true*) jelent, ettől különböző érték pedig a *hamis* (*false*) megfelelője. Egy *shell* támogatja a következő fogalmakat: változó, konstans, kifejezés, vezérlő szerkezetek, alprogram. A szintaktikai követelményeket illetően, ezek minimálisra lettek csökkentve: a paramétereket határoló zárójelek elhagyása, változódeklaráció hiánya, stb.

Egy terminálablak megnyitásakor elindított shell aktív marad mindaddig, amíg az illető ablak be nem zárul. A shell gyakorlatilag az alábbi algoritmus szerint működik:

```
Amíg ( be nem zárult a munkafázis )
  Kiírja prompt-ot;
  Olvas a parancssorból;
  Ha ( a sor végén '&' karakter van ) akkor
    Létrehoz egy új folyamatot, mely végrehajtja a beírt parancsot
    Nem vár a végrehajtás befejezésére
  Különben
    Létrehoz egy új folyamatot, mely végrehajtja a beírt parancsot
    Vár a végrehajtás befejezésére
  HaVége
AmígVége
```

Megjegyezzük, hogy amint az a fenti algoritmusból is kiderül, egy parancsot kétféleképpen hajthatunk végre:

- *előtérben (foreground)* – Ebben az esetben a *shell* elindítja a parancs végrehajtását, megvárja ennek befejeződését, majd ezután ismét kiírja a prompt-ot. Újabb parancsot csak ezt követően vihetünk be. Bármely Unix parancs esetén ez az implicit végrehajtásmód.

- háttérben (*background*) – a végrehajtás a háttérben –rejtett módon– zajlik. Ebben az esetben a *shell* elindítja a folyamatot, mely a parancs végrehajtásáért felelős, de nem várja meg ennek befejeződését, hanem azonnal kiírja a prompt-ot, ezzel felkínálva a lehetőséget a felhasználó számára, hogy újabb parancsot indítson. Amennyiben a parancsot a háttérben kívánjuk elindítani, a **'&'** speciális karakterrel kell lezárunk azt.

Egy Unix parancsablakban bármennyi folyamat indítható a háttérben, de csak egyetlen egy előtérben. Példaként tekintsük az alábbi három parancsot, melyből kettőt háttérben indítunk (egy állomány-másolás `-cp-`, és egy fordítás `-gcc-`), illetve egyet előtérben (állomány szerkesztése a `vi` szövegszerkesztővel):

```
$ cp A B &
$ gcc x.c &
$ vi H
```

III.3.2. Shell programozás

A Bourne shell (sh) rövid bemutatása

Az alábbiakban a legegyszerűbb Unix shell, az `sh` használatát mutatjuk be. Kezdjük néhány alapvető szintaktikai konvencióval.

Egy Unix parancs általános alakját a következőképpen adhatjuk meg:

parancsnév [opciók] [kifejezések] [állományok]

- ahol az opció
 - általában 1 betű
 - az opciók csoportja „-“ jellel kezdődik
 - ki-be kapcsolás: -, +

Pl.:

Az aktuális katalógus összes állományának kilistázása (beleértve a rejtett állományokat is), hosszú formátummal:

```
ls -al
```

Az *abc* nevű állomány tulajdonosának végrehajtási jogot adunk az illető állományra vonatkozóan:

```
chmod u+x abc
```

- a kifejezések – a parancs argumentumai
- a mezők között az elválasztó a szóköz
- az állománynevek tekintetében az alábbi konvenciók érvényesek:
 - ennek hossza max. 255 karakterre korlátozott
 - a shell különbséget tesz kis és nagybetű között
 - nincs kiterjesztés
 - néhány speciális karaktert nem ajánlott használni állománynévben: `<>|&[*?-/`
 - akárhány pont „.” szerepelhet az állománynévben, és ezek bárhol megjelenhetnek, esetleg néhány esetben speciális jelentése lehet a pontnak:
 - . a név elején – rejtett állományt jelöl (pl. `.forward`)
 - . az utolsó betű(k) előtt – program forráskódja (pl. `prog.C`, `p.cpp`)
 - amikor állománynevekre hivatkozunk, használhatjuk az alábbi helyettesítő karaktereket:

- ? az állománynévben – egyetlen tetszőleges karaktert helyettesít
 - * az állománynévben – 0 vagy több tetszőleges karaktert jelöl
- Pl.
 a?b lehet aab; a1b; axb; a_b; stb.
 a*b lehet: ab; a1b; aaaaab; a_xxxb; stb.
 a?b*x* lehet: a1bx; a_bcdefx3; de nem lehet abcdx

A shell néhány karakternek vagy karakterkombinációnak speciális jelentést tulajdonít. Ezeket metakaraktereknek nevezzük. Ilyen metakarakterek a következők:

- > – kimenet átirányítása
- >> – kimenet additív átirányítása
- < – bemenet átirányítása
- <<*string* – „here document” – szabványos bemenet következik egészen a *stringet* (sor elején) tartalmazó sorig
- | – pipeline (csővezeték)
- * – egyezés bármely láncsal (üressel is)
- ? – egyezés a filenévben egyetlen karakterrel
- [...] – egyezés a file-névben bármely, a zárójelben levő karakterrel (pl. [abc]; [a-z]; [1-9]; [A-Za-z])
- ; – parancslezáró
- & – parancslezáró háttérfolyamatoknál
- '...' – betű szerint értelmezi a közé írt karaktersort
- "..." – szintén betű szerinti értelmezés, de a shell értelmezi a következő speciális karaktereket: \$, `...`, \
- `...` – a közrezárt parancs helyére (a fordított aposztrófokat is beleértve) a végrehajtás eredménye kerül. Amennyiben például az aktuális katalógus a /home/user1, és a parancssorba az alábbi parancsot írjuk
`$ echo Az aktuális katalógus: `pwd``
 eredményül a következő üzenetet kapjuk a standard kimeneten:
 Az aktuális katalógus: /home/user1
 Amennyiben azonban azt íránk a parancssorba, hogy
`$ echo Az aktuális katalógus: pwd`
 ezt kapnánk:
 Az aktuális katalógus: pwd
- \ – levédi az utána következő karaktert
- # – a sor hátralevő része kommentár
- \$i – \$0,..., \$9 – a shell argumentumai
- \$var – a var változó értéke
- && – p1 && p2 – futtatja a p1 parancsot, ha az sikeres, futtatja p2-t
- || – p1|| p2 – futtatja p1-et, ha az sikertelen volt, futtatja p2-t

A Bourne shell (sh) az alábbi shell változókat kínálja fel:

- \$# – az argumentumok számát adja meg
- \$* – minden argumentum, egyetlen karakterláncként tekintve:
`"$1 $2 . . . $n";`
- @\$ – a parancssor összes argumentuma, stringek sorozataként tekintve:
`"$1" "$2" . . . "$n";`
- \$- – opciók
- \$? – az utoljára végrehajtott parancs visszatérési értéke

- \$\$ – a burok folyamatazonosítója
- \$! – az utolsó háttérben indított folyamat folyamatazonosítója

A shell által létrehozott bármelyik folyamat örököl egy sor standard, meghatározott nevű változót. Ezeknek a változóknak az összessége alkotja az illető folyamat úgynevezett *környezetét* (*environment*). Ezek közül a környezeti változók közül felsorolunk néhányat:

- \$HOME – home directory (vagy alapkatalógus)
- \$IFS – argumentumszavakat elválasztó karakterek (implicit módon a szóköz, <TAB>, illetve újsor karakterek)
- \$MAIL – az elektronikus postát tároló állomány nevét tartalmazza. Amennyiben megváltozik az adott file tartalma, a rendszer üzenetet ír ki. A \$MAILCHECK változó adja meg, hogy milyen időközönként figyelje a rendszer az új levelek érkezését.
- \$PATH – útvonal: a végrehajtható állományok keresési útvonalát adja meg. Amikor beírunk a parancssorba egy shell parancsot, a shell a \$PATH-ban felsorolt, „:”-al elválasztott elérési utakban keres egy megadott nevű végrehajtható állományt. A keresés a \$PATH-ban balról jobbra történik, és amint megvan az első találat, a keresés véget ér. Megjegyezzük, hogy a keresés kizárólag a megadott elérési utakon történik, az aktuális katalógusban csak akkor keres a rendszer, ha ez explicit módon hozzá van adva a PATH változóhoz.

A felhasználó tetszés szerint módosíthatja a PATH értékét. Például ha a meglévő értékhez hozzá szeretnénk adni az aktuális katalógust, az alapkatalógust, és ennek bin nevű alkatalógusát, ezt a következőképpen tehetjük meg:

```
$ PATH=${PATH}:::${HOME}:${HOME}/bin
```

- \$PS1 – prompt karakterlánc, implicit módon \$ közönséges felhasználó esetén (**megj.** a példákban ez a prompt jelenik meg a sor elején), illetve # a root felhasználó esetében
- \$PS2 – parancs folytatásakor használt másodlagos prompt: >
- \$LOGNAME – a bejelentkezett felhasználó azonosítója.
- \$SHELL – a használt parancsértelmezőt adja meg
- \$TERM – a használt terminál típusát adja meg
- \$TZ – a beállított időzónát adja meg

Pozicionális shell változók:

Korábban –a shell metakaraktereknek felsorolásakor– említettük, hogy a \$i (ahol *i* egy számjegy) sajátos jelentéssel bír:

- \$0 – a parancsállomány nevét adja meg
- \$1-\$9 – segítségével hivatkozhatunk a parancssor első 9 argumentumára

Tegyük fel, hogy a parancssorból a következőképpen hívtunk meg egy parancsot:

```
$ parancs arg1 arg2 ... argn
```

Amennyiben a fenti parancs egy parancsállomány (shell script) neve, melyet az alapértelmezett shell fog kiértékelni, akkor a script-en belül az alábbi módon hivatkozhatunk a parancs nevére, illetve az első 9 argumentumra:

\$	parancs	arg1	arg2	...	arg9	arg10	...	argn
	^	^	^	...	^			
				...				
	\$0	\$1	\$2	...	\$9			

Ha több, mint 9 paramétert adtunk meg, nem fog elveszlni egyik sem, azonban egy adott ponton csak az első 9-re hivatkozhatunk a megadott módon.

A burok beépített változóin-, illetve a pozicionális shell változókon kívül a felhasználó definiálhat saját változókat. Egy *var* nevű változó esetében ennek értékére *\$var*-al hivatkozunk. A változók értéke *karaktorsor*. Akkor is, ha egy bizonyos kontextusban egy változót számként interpretálunk, ennek ábrázolása a számjegyeinek megfelelő karakterek ASCII kódjának sorozataként történik.

A változókat nem kell deklarálni, egy változó definiálása gyakorlatilag megegyezik a változónak való első értékadással, és az alábbi módon történik:

```
$ változonev=karaktorsor
```

A kiértékelés során a shell létrehoz egy változót a megadott (*valtozonev*) névvel, melynek értéke a megadott (*karaktorsor*) karaktorsor. Fontos megjegyeznünk, hogy az egyenlőségjel előtt illetve után nincs szóköz! Amennyiben azt szeretnénk, hogy a megadott karaktorsorban egy vagy több szóköz szerepeljen, akkor ezeket le kell védünk.

Egy shell változó *lokális* az őt létrehozó folyamatra nézve. Ezzel együtt van rá lehetőség, hogy a változót *örököljék* az illető folyamat gyerekfolyamatai, amennyiben a változót definiáló folyamatba az alábbi deklarációt írjuk:

```
$ export valtozonev
```

ahol *valtozonev* annak a változónak a neve, amelyet szeretnénk, hogy a gyerekfolyamatok örököljenek.

Egy *változó értékének a behelyettesítése* többféleképpen történhet. Tekintsük azt a két lehetőséget, amelyik a változó értékét adja vissza vagy üres stringet, amennyiben a változó nincs meghatározva:

```
$valtozonev  
${valtozonev}
```

A második formát akkor használjuk, ha az első nem tenné lehetővé, hogy egyértelműen meg lehessen határozni a változó nevét (például amikor az egy karaktorsoron belül található).

Lássunk néhány egyszerű példát. Tegyük fel, hogy a billentyűzetről az alábbi három sort visszük be egymás után:

```
$ szol=sivatagban  
$ szo2=kutat  
$ echo A $szol egy csapat $szo2 as
```

Az *echo* parancs végrehajtásakor, mely egy sor kiírását végzi el, előbb sor kerül a *szol*, illetve *szo2* változók behelyettesítésére a megfelelő értékkel, az eredmény pedig:

```
A sivatagban egy csapat kutat as
```

Ha ezzel szemben az alábbi parancsot írjuk be:

```
$ echo A $szol megkezdodott a $szo2as
```

Az eredmény a következő lesz:

```
A sivatagban megkezdodott a
```

mivel a shell a *\$szo2as* változó értékét próbálja behelyettesíteni, az pedig nincs definiálva, azaz üres string lesz az értéke. Az ehhez hasonló helyzetek elkerülésére használhatjuk a másodikként megadott helyettesítési formát:

```
$ echo A $szol megkezdodott a ${szo2}as
```

A parancs végrehajtásának eredménye ekkor:

A sivatagban megkezdődött a kutatás

Az **sh** 13 kulcsszóval rendelkezik. Ezek az alábbiak:

```
if then else elif fi
case in esac
for while until do done
```

Az sh által használt vezérlő szerkezetek

Az **if** vezérlő szerkezet szintaxisa a következő:

```
if utasítások1
then utasítások2
[elif utasítások3
  then utasítások4
...
elif utasításokn-1
  then utasításokn]
[else utasításokn+1]
fi
```

Egy **if** utasításon belül tehát akárhány **elif ... then** ág szerepelhet, az utasítás végén pedig megjelenhet (de csak egyszer) az **else**

Megjegyezzük, hogy az **if**, **then**, **elif**, **fi** kulcsszavak szintaktikai szempontból úgy viselkednek, mintha külön parancsok lennének, ezért vagy új sorba kell írunk őket, vagy – amennyiben valamelyik nincs külön sorban – a parancsokat egymástól elválasztó „;”-vel kell azt elválasztanunk a sor többi részétől.

A **if**-et vagy **elif**-et követő parancslistának kettős szerepe van: egyrészt a listában levő parancsok végrehajtása, másrészt a végrehajtás *igazságértékének* a meghatározása. Egy parancslista végrehajtásának értéke *true*, amennyiben a listából az utoljára végrehajtott parancs visszatérített értéke *0*. A végrehajtás értéke *false*, ha a visszatérített érték zérótól különböző. A **then** vagy **else** után következő parancslista ennek az igazságértéknek a függvényében hajtódik végre vagy sem.

Az **if** utasítás a következőképpen működik:

- Végre lesz hajtva az **if**-et követő parancslista. Amennyiben a végrehajtott utasítássorozat igazságértéke *true*, akkor a **then** ágon szereplő parancsok sorozata hajtódik végre, és az **if** utasítás végrehajtása befejeződik. Ellenkező esetben (a végrehajtott utasítássorozat igazságértéke *false*) a következő lépés következik:
- amennyiben van egy vagy több **elif** ág, akkor rendre végrehajtottatik az őket követő parancslista, mindaddig, amíg valamelyiknek az igazságértéke igaz (*true*) nem lesz. Ezt követően az utána következő **then** ág parancsai hatódnak végre és az **if** utasítás végrehajtása befejeződik. Ellenkező esetben (vagy egyáltalán nincs **elif** vagy az összes parancslista *false*-ra értékelődik ki), az alábbi lépés következik:
- amennyiben van **else** ág, végrehajtottatik az **else** utáni parancslista és az **if** végrehajtása befejeződik. Ellenkező esetben (nincs **else** ág):
- az **if** végrehajtása befejeződik és az **if**-et követő utasítással folytatódik a végrehajtás.

Az alábbiakban példaként bemutatunk –két változatban– egy parancsállományt, mely egy szöveges állomány sorait ábécésorrendbe rendezve listázza ki. Az állomány nevét a parancssor első paramétereként adjuk meg. Az első változat:

```
if [ $# -eq 0 ]
then echo "Használat: $0 állománynév"
else sort $1 | more
fi
```

A bemutatott változat csupán azt ellenőrzi, hogy megadtunk-e egy paramétert a parancssorban. A következő változat alaposabb ellenőrzést végez (azt is megvizsgáljuk, hogy a paraméterként megadott állomány létezik-e):

```
if [ $# -eq 0 ]
then echo "Használat: $0 állománynév"
elif [ ! -f "$1" ]
then echo "$1 állomány nem létezik"
else sort $1 | more
fi
```

Ismétlő struktúrák

A shell négyféle ismétlő struktúrával rendelkezik: **for** két változatban, **while** és **until**. Ezek szintaxisa:

```
for változónév
do
    utasítások
done
```

```
for változónév in szavak
do
    utasítások
done
```

```
while utasítások1
do
    utasítások2
done
```

```
until utasítások1
do
    utasítások2
done
```

A for ismétlő struktúra

A shell ismétlő struktúrái közül ez a leggyakrabban használt. Két alakja van, mindkettő egy `változónév` nevű kontroll-változót használ (a változó neve természetesen tetszőleges lehet).

Az első formában a `változónév` rendre felveszi a parancssorban megadott összes paraméter értékét: `$1`, `$2`, ..., (tulajdonképpen a `$@` változóból veszi a shell az értékeket). Ezek mindegyikére végrehajtja a ciklus törzsében levő utasításokat.

A második alakban az `in` után következő szavak listája szóközzel elválasztott egyszerű szavakat jelöl vagy helyettesítő karaktereket tartalmazó állománynevek szerepelhetnek ott, melyek ki lesznek terjesztve az összes illeszkedő állománynévre, így végül egy állománylistát kapunk. A `változónév` rendre felveszi a lista elemeinek értékét, és mindegyikre végre lesz hajtva az utasítások sorozata.

Lássunk néhány példát. Az első példa egyenként rendezi és kilistázza a paraméterként megadott állományok tartalmát:

```
for allomany
do
    sort $allomany | more
done
```

Feltételezzük, hogy a parancsállomány neve *rendez*. Ebben az esetben a következő parancs:

```
$ rendez A b C
```

az alábbi parancsokat fogja generálni és végrehajtani:

```
sort A | more
sort b | more
sort C | more
```

Ugyanezt a hatást érjük el, amennyiben az állománynevek a *rendez* parancsállományon belül vannak felsorolva:

```
for allomany in A b C
do
    sort $allomany | more
done
```

a parancsállományt pedig a következőképpen hívjuk meg (ezúttal paraméterek nélkül):

```
$ rendez
```

Végül rendezzük az aktuális katalógus összes olyan állományát, melynek neve „adat”-tal végződik:

```
for allomany in *adat
do
    sort $allomany | more
done
```

Az alábbi példa az összes bejelentkezett felhasználónak küld egy mailt:

```
for x in `who | cut -f1 -d ' '`
do
    mail -s "Udvozlet" ${x}@scs.ubbcluj.ro <<UZENET
    Elnezezt a zavarasert. Ezt az uzenetet csupan a for ciklus
    tesztelese vegett kuldtuk el.
    UZENET
done
```

A **while** és **until** ismétlő struktúrák

A kétféle utasítás hasonlít egymáshoz, amennyiben mindkettő előbb az `utasítások1` utasítássorozatot hajtja végre. A végrehajtott utasítássorozat igazságértékétől (azaz az utolsó parancs visszaadott értékétől) függően végrehajthatódik vagy sem a `do` és `done` közötti `utasítások2` utasítássorozat, majd ismét az `utasítások1` kiértékelésére kerül sor vagy befejeződik a ciklus végrehajtása.

A **while** ciklus végrehajtása akkor fejeződik be, ha az `utasítások1` utasítássorozat utolsó parancsának visszaadott értéke zérótól különböző. Ezzel ellentétben az **until** ciklus akkor fejeződik be, amikor 0-t kapunk vissza.

Az alábbi példában a paraméterként megadott állományok rendezését/kilistázását megvalósító feladatot láthatjuk **while** majd **until** ciklust használva:

<pre>while [\$# -gt 0] do if [-f "\$1"] then sort \$1 more else echo "nincs \$1 file" fi shift done</pre>	<pre>until [\$# -eq 0] do if [-f "\$1"] then sort \$1 more else echo "nincs \$1 file" fi shift done</pre>
---	---

A **true**, **false**, **break**, **continue** utasítások

Egyszerű utasításokról van szó, de végrehajtásuknak kizárólag a ciklikus vezérlő szerkezetek kontextusában van értelme.

A **break** illetve **continue** a **for**, **while** vagy **until** utasítások befejezését illetve a ciklus újraiterálását vonják maguk után. Az említett parancsok a C nyelvből lettek kölcsönözve (ahol kizárólag a legbelső ciklusra vonatkozik a hatásuk), és a shell által kiterjesztve. Szintaxisuk a következő:

```
break [ n ]
continue [ n ]
```

A **break** parancs a ciklus törzsének elhagyását kéri, ezt követően a végrehajtás a ciklus utáni utasítással folytatódik. Amennyiben az `n` paraméter hiányzik, akkor a **break** utasítást tartalmazó legbelső ciklus elhagyására kerül sor. Ha viszont az `n` is jelen van és a **break** legalább `n` egymásba ágyazott ciklus belsejében van, akkor az `n`. ciklust követő utasítással folytatódik a végrehajtás.

A **continue** utasítás a következő iterációval folytatja a ciklus végrehajtását. Az `n` paraméter nélkül a legbelső ciklus lesz újraiterálva, különben az `n`. ciklus, amelybe a **continue** bele van ágyazva.

Az újraiterálás a **for** esetében azt jelenti, hogy a ciklusváltozó a következő értéket kapja meg, **while** és **until** esetében pedig a **while** vagy **until** után következő utasítássorozat lesz ismét végrehajtva.

1. Példa : egy felügyelőprogram

Egy Unix rendszerben a gyakorlatban nemegyszer szükség lehet arra, hogy egy bizonyos katalógus változásait felügyelet alatt tartsuk. Tegyük fel, hogy a felügyelet a következőképpen történik: az első paraméterként (másodpercben) megadott t időközönként a program elvégzi a (második paraméterként megadott) megfigyelt katalógus tartalmának részletes összefoglalását. Amennyiben ez az összefoglalás megegyezik a t másodperccel ezelőtt lementett, a program további t másodpercet vár, majd ismét ellenőrzi a katalógus tartalmát és így tovább. Az első olyan esetben, amikor különbséget talál a program a régi, illetve új tartalom között, kiír egy megfelelő üzenetet, és befejeződik.

A feladatot a `megfigyel` nevű shell script fogja elvégezni, melyet a következőkben mutatunk be.

A programmal kapcsolatos néhány megjegyzés:

- A t illetve `katalogus` változók a két vizsgálat között eltelt időintervallumot valamint a megfigyelt katalógust adják meg. A t változó inicializálása a $\$1$ (első) paraméteren keresztül történik. Amennyiben ez hiányzik, a t változó a 60 implicit értéket kapja. Hasonlóképpen, a `katalogus` változó értékét megadhatjuk a $\$2$ (második) paraméter segítségével, ennek hiányában pedig az alapkatalógus lesz az alapértelmezett érték.
- Az x változó a katalógus tartalmának utolsó előtti összefoglalóját tartalmazza, y pedig a legutolsót jegyzi meg.

```
#!/bin/sh
katalogus=${2-${HOME}}           # $2 hiányában az alapkatalógus
                                   # lesz az alapértelmezett
t=${1-60}                         # $1 hiányában t=60
x=`ls -l $katalogus`             # régi összefoglaló
while true
do
    sleep $t                      # t mp.-t vár
    y=`ls -l $katalogus`         # új összefoglaló
    if [ "$x" != "$y" ]          # megegyeznek?
    then
        echo "A $katalogus katalógus tartalma megváltozott."
        exit 0
    else
        echo "Semmi változás. Várunk újabb $t másodpercet."
    fi
    x=$y                          # megjegyezzük a legutóbbi
                                   # összefoglalót
done
```

Egy ilyen programot különböző helyzetekben használhatunk. Egy lehetséges eset a következő: egy tetszőleges felhasználó két terminálablakot nyitott meg, és az egyikben az alábbi parancsot írja be:

```
$ megfigyel 10
```

Amennyiben a másik terminálablakban módosítjuk a `$HOME` alapkatalógus tartalmát, például létrehozunk egy új állományt a `cat >A` paranccsal, akkor a másik terminálablakban legközb 10 másodpercen belül megjelenik az üzenet, mely a módosulásról értesít.

2. példa: break és continue használata

A `break` és `continue` utasítások használatának példázására tekintsük a következő feladatot: keressünk az aktuális katalógusban egy szöveges állományt, melyben találunk olyan sort, amiben az első szó 5 karakternél hosszabb. A feladatot megoldó program a következő:

```
for x in *
do
  if ! file -b $x | grep -q text
  then
    echo $x nem szöveges állomány. Lássuk a következőt...
    continue
  fi
  #a szol változóban megjegyezzük egy sor első szavát
  #(szóelválasztónak a szóköz karaktert tekintjük)
  for szol in `cat $x | cut -d" " -f1`
  do
    #megvizsgáljuk, hogy a sor nem-e üres, illetve az első szó
    #hosszát
    if [ ! -z "$szol" ] && [ `expr length $szol` -ge 5 ]
    then
      echo A $x fileban megtaláltuk $szol szót, \
        melynek hossza `expr length $szol`
      #kilépünk`
      break 2
    fi
  done
done
```

A szöveges állományok kiválasztását a `file` és `grep` parancsok összekombinálásának segítségével valósítjuk meg. Az első találó szó esetében elhagyjuk a két `for` ciklust a `break` utasítás segítségével. Amennyiben elhagyjuk a `break` paraméterét, ki lesz írva minden állomány első olyan szava, mely megfelel a követelményeknek, ha pedig a `break`-et tartalmazó sort kikommentezzük, az összes találó szót megkapjuk.

3. példa: közönséges állományok összefűzése

Egy olyan **sh** script megírására van szükség, melyet az alábbi módon hívunk meg:

```
$ pall katalogus
```

Ennek hatására pedig a `/tmp` katalógusban hozzon létre egy olyan szöveges állományt, mely magába foglalja a megadott katalógusban vagy ennek alkatalógusaiban található összes kinyomtatható állomány tartalmát. Az eredményként szolgáló szöveges állományban, az öt alkotó minden egyes állomány elején egy, az állományt azonosító fejléccet helyezünk el.

Mikor hasznos egy ilyen alkalmazás? Tegyük fel, hogy egy felhasználónak egy bizonyos katalógusszerkezetben rengeteg szöveges állománya, shell script-je, forráskódja, stb. van. Ahelyett, hogy ezeket külön-külön kellene kinyomtatni, a felhasználó használhatja a fentebb említett funkcionális megvalósító programot.

A `pall` program az `egrep` szűrő segítségével beazonosítja az összes olyan folyamatot, melyek kinyomtathatóak, végül egyesíti ezeket egyetlen nyomtatható állományba. A `pall` program forráskódja a következő:

```
#!/bin/sh
if [ $# -ne 1 ]
then echo "Hasznalat:  $0 katalogus" >&2
    exit 1
fi
if [ ! -d "$1" ]
then echo "$1 nem letezik vagy nem katalogus" >&2
    exit 2
fi
rm /tmp/${LOGNAME}Listazas /tmp/${LOGNAME}Listazni >/dev/null 2>&1

osszSorokSzama=0
find $1 -type f -print | sort | while read file
do
    if file $file | egrep "exec|data|empty|reloc|cannot open" >/dev/null 2>&1
    then
        continue
    else
        sorokSzama=`wc -l <"$file"`
        sor=${osszSorokSzama}" a "`file $file`" allomanyig"
        echo $sor >/dev/tty
        echo $sor >> /tmp/${LOGNAME}Listazni
        echo $sor >> /tmp/${LOGNAME}Listazas
        pr -f $file >> /tmp/${LOGNAME}Listazas
        osszSorokSzama=`expr $osszSorokSzama + $sorokSzama`
    fi
done
echo "Osszesites: $osszSorokSzama sor" >>/tmp/${LOGNAME}Listazni
echo "Osszesites: $osszSorokSzama sor" >>/tmp/${LOGNAME}Listazas
cat /tmp/${LOGNAME}Listazas >>/tmp/${LOGNAME}Listazni
rm /tmp/${LOGNAME}Listazas
```

A program létrehozza a `/tmp/${LOGNAME}Listazas` állományt, melynek elején egy tartalomjegyzék található, ami tartalmazza az állományok nevét, illetve a sorok számában mért hosszát. A `sorokSzama` nevű változó az aktuálisan feldolgozott állomány sorainak számát tartalmazza. Az `osszSorokSzama` változóban összegezzük a sorok számát az állományok összefűzése során.

A kinyomtatható állományokra vonatkozó részletesebb információért ajánljuk az `/usr/share/magic` (Linux alatt), illetve `/etc/magic` (Solaris) állományok tanulmányozását, ugyanis ezeket használja a `file` parancs az állomány típusának meghatározására.

III.4. Javasolt feladatok

I.

- Írjuk le röviden a `fork` rendszerhívás működését, és ennek lehetséges visszatérítési értékeit.
- Mit ír ki a képernyőre az alábbi programrész, feltételezve, hogy a `fork` rendszerhívás sikeresen hajtodik végre? Indokoljuk a választ.

```

int main() {
    int n = 1;
    if(fork() == 0) {
        n = n + 1;
        exit(0);
    }
    n = n + 2;
    printf("%d: %d\n", getpid(), n);
    wait(0);
    return 0;
}

```

c. Mit ír ki a képernyőre az alábbi shell script? Magyarázzuk meg az első három sor működését.

1	for F in *.txt; do
2	K=`grep abc \$F`
3	if ["\$K" != ""]; then
4	echo \$F
5	fi
6	done

II.

a. Adott az alábbi kódrészlet. Adjuk meg azokat a sorokat, amelyek a képernyőn fognak megjelenni, abban a sorrendben, ahogy azok ki lesznek írva, feltételezve, hogy a fork rendszerhívás sikerrel tér vissza. Indokoljuk a választ.

```

int main() {
    int i;
    for(i=0; i<2; i++) {
        printf("%d: %d\n", getpid(), i);
        if(fork() == 0) {
            printf("%d: %d\n", getpid(), i);
            exit(0);
        }
    }
    for(i=0; i<2; i++) {
        wait(0);
    }
    return 0;
}

```

b. Magyarázzuk meg az alábbi shell script működését. Mi történik akkor, ha a *lista.txt* állomány eredetileg hiányzik?

Adjuk hozzá az alábbi kódrészlethez az új *lista.txt* állományt generáló hiányzó sort (a *lista.txt* a megadott kódrészlet által generált változtatásban érintett állományok listáját kell tartalmazza).

```
more lista.txt
```

```
rm lista.txt
for f in *.sh; do
    if [ ! -x $f ]; then
        chmod 700 $f
    fi
done
mail -s "Erintett allomanyok" admin@scs.ubbcluj.ro <lista.txt
```

III.5. Általános könyvészet

1. ***: Linux man magyarul, <http://people.inf.elte.hu/csa/MAN/HTML/index.htm>
2. A.S. Tanenbaum, A.S. Woodhull, *Operációs rendszerek*, 2007, Panem Kiadó.
3. Alexandrescu, *Programarea modernă în C++*. Programare generică și modele de proiectare aplicate, Editura Teora, 2002.
4. Angster Erzsébet: *Objektumorientált tervezés és programozás Java*, 4KÖR Bt, 2003.
5. Bartók Nagy János, Laufer Judit, *UNIX felhasználói ismeretek*, Openinfo
6. Bjarne Stroustrup: *A C++ programozási nyelv*, Kiskapu kiadó, Budapest, 2001.
7. Bjarne Stroustrup: *The C++ Programming Language Special Edition*, AT&T, 2000.
8. Boian F.M. Frentiu M., Lazăr I. Tambulea L. *Informatica de bază*. Presa Universitară Clujeana, Cluj, 2005
9. Boian F.M., Ferdean C.M., Boian R.F., Dragoș R.C., *Programare concurentă pe platforme Unix, Windows, Java*, Ed. Albastră, Cluj-Napoca, 2002
10. Boian F.M., Vancea A., Bufnea D., Boian R.,F., Cobârzan C., Sterca A., Cojocar D., *Sisteme de operare*, RISOPRINT, 2006
11. Bradley L. Jones: *C# mesteri szinten 21 nap alatt*, Kiskapu kiadó, Budapest, 2004.
12. Bradley L. Jones: *SAMS Teach Yourself the C# Language in 21 Days*, Pearson Education, 2004.
13. Cormen, T., Leiserson, C., Rivest, R., *Introducere în algoritmi*, Editura Computer Libris Agora, Cluj, 2000
14. DATE, C.J., *An Introduction to Database Systems (8th Edition)*, Addison-Wesley, 2004.
15. Eckel B., *Thinking in C++*, vol I-II, <http://www.mindview.net>
16. Ellis M.A., Stroustrup B., *The annotated C++ Reference Manual*, Addison-Wesley, 1995
17. Frentiu M., Lazăr I. *Bazele programării*. Partea I-a: Proiectarea algoritmilor
18. Herbert Schildt: *Java. The Complete Reference*, Eighth Edition, McGraw-Hill, 2011.
19. Horowitz, E., *Fundamentals of Data Structures in C++*, Computer Science Press, 1995
20. J. D. Ullman, J. Widom: *Adatbázisrendszerek - Alapvetés*, Panem kiado, 2008.
21. ULLMAN, J., WIDOM, J., *A First Course in Database Systems (3rd Edition)*, Addison-Wesley + Prentice-Hall, 2011.
22. Kiadó Kft, 1998, <http://www.szabilinux.hu/ufi/main.htm>
23. Niculescu, V., Czibula, G., *Structuri fundamentale de date și algoritmi. O perspectivă orientată obiect.*, Ed. Casa Cărții de Știință, Cluj-Napoca, 2011

24. RAMAKRISHNAN, R., Database Management Systems. McGraw-Hill, 2007, <http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
25. Robert Sedgewick: *Algorithms*, Addison-Wesley, 1984
26. Simon Károly: *Kenyerünk Java. A Java programozás alapjai*, Presa Universitară Clujeană, 2010.
27. Tâmbulea L., Baze de date, Facultatea de matematică și Informatică, Centrul de Formare Continuă și Invățământ la Distanță, Cluj-Napoca, 2003
28. V. Varga: *Adatbázisrendszerek (A relációs modelltől az XML adatokig)*, Editura Presa Universitară Clujeană, 2005, p. 260. ISBN 973-610-372-2

IV. Adatbáziskezelés

IV.1. A relációs adatmodell

Az első ABKR-ek hálós vagy hierarchikus adatmodellt használtak. Manapság a relációs adatmodell a legelterjedtebb. A népszerűséget annak köszönheti, hogy nagyon egyszerű deklaratív nyelvvel rendelkezik az adatok kezelésére, illetve lekérdezésére. A relációs adatmodell értékorientált, ez ahhoz vezet, hogy a relációkon értelmezett műveletek eredményei szintén relációk.

Ha adottak a $D_1, D_2, \dots, D_n$ nem szükségszerűen egymást kizáró halmazok, akkor R egy reláció a $D_1, D_2, \dots, D_n$ halmazokon, ha $R \subseteq D_1 \times D_2 \times \dots \times D_n$ (Descartes-féle szorzat).

A relációs adatmodell szempontjából D_i az A_i attribútum értékeinek tartománya (doméniuma). D_i lehet egész számok halmaza, karaktorsorok halmaza, valós számok halmaza stb., n a reláció foka.

Egy ilyen relációt táblázatban ábrázolhatunk:

R	A_1	...	A_j	...	A_n
r_1	a_{11}	...	a_{1j}	...	a_{1n}
$\vdots$					
r_i	a_{i1}	...	a_{ij}	...	a_{in}
$\vdots$					
r_m	a_{m1}	...	a_{mj}	...	a_{mn}

ahol $a_{ij} \in D_j$.

A táblázat sorai a reláció elemei. Nagyon sok esetben a tábla megnevezést használják a reláció helyett. A relációt a következőképpen jelöljük: $R(A_1, A_2, \dots, A_n)$. A reláció nevét és a reláció attribútumainak a halmazát együtt *relációsémának* nevezzük.

példa: Diákok reláció:

<i>Név</i>	<i>SzületésiDátum</i>	<i>CsopKod</i>
Nagy Ödön	1975-DEC-13	512
Kiss Csaba	1971-APR-20	541
Papp József	1973-JAN-6	521

példa: Könyvek reláció:

<i>Szerző</i>	<i>Cím</i>	<i>Kiadó</i>	<i>KiadÉv</i>
C. J. Date	An Introduction to Database Systems	Addison-Wesley	1995
Paul Helman	The Science of Database	IRWIN	1994

A relációs adatmodell tulajdonságai

A relációs adatbázis relációi vagy táblái a következő tulajdonságokkal rendelkeznek:

1. A tábla nem tartalmazhat két teljesen azonos sort, azaz két egyed előfordulás (sor) legalább egy tulajdonság (attribútum) konkrét értékében el kell hogy térjen egymástól.
2. Kulcs értelmezése: egy S attribútumhalmaz az R reláció kulcsa, ha:
 - R relációnak nem lehet két sora, melynek értékei megegyeznek az S halmaz minden attribútumára.
 - S egyetlen valódi részhalmaza sem rendelkezik a) tulajdonsággal.
- Ha a konkrét egyedek több olyan tulajdonsággal is rendelkeznek, amelyek értéke egyedi minden egyes előfordulásra nézve, akkor több *kulcsjelöltről* beszélhetünk. Ezek közül egyet *elsődleges kulcsnak* kell kijelölni. Az is megtörténhet, hogy nincs olyan tulajdonság, amelynek értéke egyedi lenne az egyed-előfordulásokra nézve. Ekkor több tulajdonság értéke együtt fogja jelenteni az elsődleges (összetett) kulcsot. Az 1. tulajdonságból következik, hogy mindig kell legyen elsődleges kulcs, ha más nem, a teljes sor mindig egyedi. Elsődleges kulcs értéke soha nem lehet null vagy üres.
3. A táblázat sorainak vagyis az egyedelőfordulásoknak a sorrendje lényegtelen.
4. A táblázat oszlopaira vagyis a tulajdonságtípusokra, attribútumokra nevükkel hivatkozunk, tehát két attribútumnak nem lehet ugyanaz a neve.
5. A táblázat oszlopainak a sorrendje lényegtelen.

Az adatbázis módosításakor az új információ nagyon sokféleképpen lehet hibás. Ahhoz, hogy az adatbázis adatai helyesek legyenek, különböző feltételeknek kell eleget tenniük.

A *megszorítások* azon követelmények, melyeket az adatbázis adatai ki kell elégítsenek, ahhoz, hogy helyeseknek tekinthessék őket.

Megszorítások osztályozása

1. *Egyedi kulcs feltétel*: egy relációban nem lehet két sor, melyeknek ugyanaz a kulcsértéke, vagyis ha C egy R reláció kulcsa, $\forall t_1, t_2 \in R$ sorok esetén $\pi_C(t_1) \neq \pi_C(t_2)$.
2. *Hivatkozási épség megszorítás*: megkövetelik, hogy egy objektum által hivatkozott érték létezzen az adatbázisban. Ez analóg azzal, hogy a hagyományos programokban tilosak azok a mutatók, amelyek sehova se mutatnak. *Külső kulcs* egy KK attribútum vagy attribútumhalmaz egy R_1 relációból, mely értékeinek halmaza ugyanaz, mint egy R_2 reláció elsődleges kulcsának az értékhalmaza, és az a feladata, hogy az R_1 és R_2 közötti kapcsolatot modellezze. R_1 az a reláció, mely hivatkozik, az R_2 pedig, amelyre hivatkozik. Más megnevezés: az R_2 az apa és az R_1 a fiú (egy sorhoz az R_2 -ből tartozhat több sor az R_1 -ből, az R_2 -ben elsődleges kulcs az attribútum ami a kapcsolatot megteremti. Fordítva nem állhat fenn a kapcsolat, hogy egy sorhoz az R_1 -ből több sor is kapcsolódjon az R_2 -ből). A hivatkozási épség megszorítás a következőket jelenti:
 - az R_2 relációban azt az attribútumot (esetleg attribútumhalmazt), melyre az R_1 hivatkozik elsődleges kulcsnak kell deklarálni,
 - KK minden értéke az R_1 -ből kell létezzen az R_2 relációban, mint elsődleges kulcs értéke.
3. *Értelmezéstartomány-megszorítások*: azt jelentik, hogy egy attribútum az értékeit a megadott értékhalmazból vagy értéktartományból veheti fel.
4. *Általános megszorítások*: tetszőleges követelmények, amelyeket be kell tartani az adatbázisban.

IV.2.Normalizálás

IV.2.1.Funkcionális függőségek

Legyen egy reláció

$R(A_1, A_2, \dots, A_n)$, ahol A_i attribútumok.

Jelöljük az attribútumok halmazát

$A = \{A_1, A_2, \dots, A_n\}$.

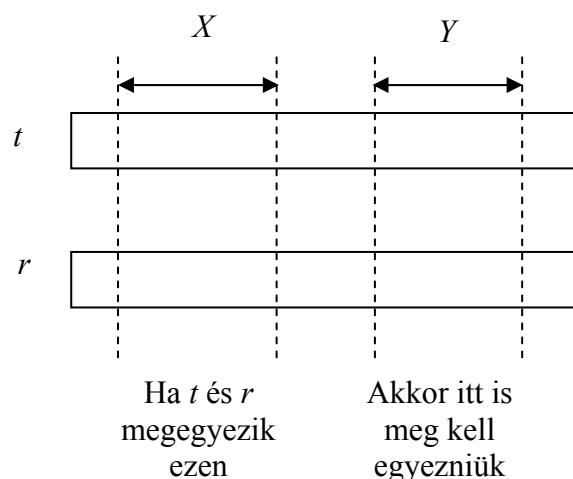
Legyenek X és Y az R reláció attribútumhalmazának részhalmazai, vagyis $X, Y \subset A$. Ezeket a jelöléseket használjuk a továbbiakban, ha esetleg nem ismételjük meg.

X attribútumhalmaz *funkcionálisan meghatározza* Y attribútumhalmazt (vagy Y *funkcionálisan függ* X -től), ha R minden előfordulásában ugyanazt az értéket veszi fel Y , amikor az X értéke ugyanaz.

Másképp: X funkcionálisan meghatározza Y -t, ha R két sora megegyezik az X attribútumain (azaz ezen attribútumok mindegyikéhez megfeleltetett komponensnek ugyanaz az értéke a két sorban), akkor meg kell egyezniük az Y attribútumain is. Ezt a függőséget formálisan $X \rightarrow Y$ -nal jelöljük.

Relációs algebrai műveletek segítségével a következőképpen értelmezhetjük a funkcionális függőséget:

$X \rightarrow Y$, ha $\forall t, r \in R$ sor esetén, melyre $\pi_X(t) = \pi_X(r)$, akkor $\pi_Y(t) = \pi_Y(r)$.



3.1. ábra: A funkcionális függőség két soron vett hatása
példa: SzállításiInformációk reláció:

SzállID	SzállNév	SzállCím	ÁruID	ÁruNév	MértEgys	Ár
111	Rolicom	A. Iancu 15	45	Milka csoki	tábla	25000
222	Sorex	22 dec. 6	45	Milka csoki	tábla	26500
111	Rolicom	A. Iancu 15	67	Heidi csoki	tábla	17000
111	Rolicom	A. Iancu 15	56	Milky way	rúd	20000
222	Sorex	22 dec. 6	67	Heidi csoki	tábla	18000
222	Sorex	22 dec. 6	56	Milky way	rúd	22500

Funkcionális függőségek:

$SzállID \rightarrow SzállNév$

$SzállID \rightarrow SzállCím$.

Mivel mindkét függőségnek ugyanaz a bal oldala, SzállID, ezért egy sorban összegezhetjük:

$SzállID \rightarrow \{SzállNév, SzállCím\}$

Szavakban, ha két sorban ugyanaz a SzállID értéke, akkor a SzállNév értéke is ugyanaz kell legyen, illetve a SzállCím értéke is.

Ezenkívül:

$\text{ÁruID} \rightarrow \text{ÁruNév}$

$\text{ÁruID} \rightarrow \text{MértEgys}$ (azzal a feltevéssel, ha más mértékegységben árulják az árut, más ID-t is kap).

Hasonlóan egy sorban:

$\text{ÁruID} \rightarrow \{\text{ÁruNév}, \text{MértEgys}\}$

A funkcionális függőséget felhasználva adhatunk még egy értelmezést a reláció kulcsának. Egy vagy több attribútumból álló $\{C_1, C_2, \dots, C_k\}$ halmaz a *reláció kulcsa*, ha:

- Ezek az attribútumok funkcionálisan meghatározzák a reláció minden más attribútumát, azaz nincs az R -ben két különböző sor, amely mindegyik $C_1, C_2, \dots, C_k$ -n megegyezne.
- Nincs olyan valódi részhalmaza $\{C_1, C_2, \dots, C_k\}$ -nak, amely funkcionálisan meghatározná az R összes többi attribútumát, azaz a kulcsnak minimálisnak kell lennie.

példa: a SzállításiInformációk reláció kulcsa a $\{\text{SzállID}, \text{ÁruID}\}$, egy szállító egy árut egy árban szállít egy adott pillanatban. Nincs a táblában 2 sor, ahol ugyanaz legyen a SzállID és az ÁruID is. Csak a SzállID nem elég kulcsnak, mert egy szállító több árut is szállíthat, az ÁruID sem elég, mert egy árut több szállító is ajánlhat. $\square$

*Szuperkulcsok*nak nevezzük azon attribútumhalmazokat, melyek tartalmazznak kulcsot. A szuperkulcsok eleget tesznek a kulcs definíció első feltételének, de nem feltétlenül tesznek eleget a minimalitásnak. Tehát minden kulcs szuperkulcs.

Az $R(A_1, A_2, \dots, A_n)$ reláció esetén A_i attribútum *prim*, ha létezik egy C kulcsa az R -nek, úgy hogy $A_i \in C$. Ha egy attribútum nem része egy kulcsnak, akkor *nem prim* attribútumnak nevezzük.

Triviális funkcionális függőségről beszélünk, ha az Y attribútum halmaz részhalmaza az X attribútum halmaznak ($Y \subset X$), akkor Y attribútum halmaz funkcionálisan függ X attribútum halmaztól ($X \rightarrow Y$).

példa: Triviális funkcionális függőség: $\{\text{SzállID}, \text{ÁruID}\} \rightarrow \text{SzállID}$. $\square$

Minden triviális függőség érvényes minden relációban, mivel amikor azt mondjuk, hogy „két sor megegyezik X minden attribútumán, akkor megegyezik ezek bármelyikén is”.

Nem triviális egy $X_1X_2\dots X_p \rightarrow Y_1Y_2\dots Y_s$ funkcionális függőség, ha az Y -ok közül legalább egy különbözik az X -ektől, vagyis

$$\exists Y_j, j \in [1, s] \text{ } j \in \{1, 2, \dots, s\} \text{ úgy, hogy } Y_j \neq X_k, \forall k \in \{1, 2, \dots, p\}.$$

Teljesen nem triviális egy $X_1X_2\dots X_p \rightarrow Y_1Y_2\dots Y_s$ funkcionális függőség, ha az Y -ok közül egy sem egyezik meg az X -ek valamelyikével, vagyis

$$\forall Y_j, j \in [1, s] \text{ } j \in \{1, 2, \dots, s\} \text{ -re } Y_j \neq X_k, \forall k \in \{1, 2, \dots, p\}.$$

Parciális függőség: Ha C egy kulcsa az R relációnak, az Y attribútumhalmaz valódi részhalmaza a C -nek ($Y \subset C$) és B egy attribútum, mely nem része az Y -nak ($B \notin Y$), akkor az $Y \rightarrow B$ -t egy parciális függőség. (B függ a kulcs egy részétől.)

példa: parciális függőségre: $\text{SzállID} \rightarrow \text{SzállNév}$. $\square$

A SzállításiInformációk relációban $\{\text{SzállID}, \text{ÁruID}\}$ a kulcs, tehát

$\{\text{SzállID}, \text{ÁruID}\} \rightarrow \text{SzállNév},$

mivel a kulcs funkcionálisan meghatároz minden más attribútumot, de a SzállNév függ a kulcs egy részétől is.

Tranzitív függőség: Legyen $Y \subset A$ egy attribútumhalmaz és B egy attribútum, mely nem része Y -nak ($B \notin Y$). Egy $Y \rightarrow B$ funkcionális függőség tranzitív függőség, ha Y nem szuperkulcs R relációban és nem is valódi részhalmaza R egy kulcsának.

Honnan a tranzitív elnevezés? Amint látjuk, Y nem kulcs, nem része a kulcsnak, tehát egy nemtriviális funkcionális függőség az, hogy Y funkcionálisan függ az R kulcsától (C -től). Tehát $C \rightarrow Y$ és $Y \rightarrow B$, és erre mondhatjuk, hogy B tranzitív függőséggel függ C -től.

példa: Rendelések (RendelésSzám, Dátum, VevőID, VevőNév, Részletek), egy cég rendeléseit tartalmazó reláció. A különböző vevők rendeléseket helyeznek el a cégnél, a

cég más-más számot ad a különböző rendeléseknek, így a RendelésSzám elsődleges kulcs lesz, tehát kulcs révén funkcionálisan meghatározza az összes többi attribútumot:

RendelésSzám $\rightarrow$ VevőID.

Ezenkívül fennáll a

VevőID $\rightarrow$ VevőNév

funkcionális függőség. Tehát a VevőNév tranzitív függőséggel függ a RendelésSzámtól.

Funkcionális függőségek tulajdonságai:

1. Ha C az $R[A_1, A_2, \dots, A_n]$ reláció egy kulcsa, akkor $C \rightarrow \beta, \forall \beta \in \{A_1, A_2, \dots, A_n\}$.
2. Ha $\beta \subseteq \alpha$, akkor $\alpha \rightarrow \beta$, ez a **triviális** funkcionális függőség vagy reflexivitás.

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow_{\beta \subseteq \alpha} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \alpha \rightarrow \beta$$
3. Ha $\alpha \rightarrow \beta$, akkor $\gamma \rightarrow \beta, \forall \gamma$ ahol $\alpha \subset \gamma$.

$$\Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow_{\alpha \subset \gamma} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow_{\alpha \rightarrow \beta} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \gamma \rightarrow \beta$$
4. Ha $\alpha \rightarrow \beta$ és $\beta \rightarrow \gamma$, akkor $\alpha \rightarrow \gamma$, ez a funkcionális függőség **tranzitív** tulajdonsága.

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow_{\alpha \rightarrow \beta} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow_{\beta \rightarrow \gamma} \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow \alpha \rightarrow \gamma$$
5. Ha $\alpha \rightarrow \beta$ és $\gamma \subset A$, akkor $\alpha\gamma \rightarrow \beta\gamma$, ahol $\alpha\gamma = \alpha \cup \gamma$.

$$\Pi_{\alpha\gamma}(r_1) = \Pi_{\alpha\gamma}(r_2) \Rightarrow \left| \begin{array}{l} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \end{array} \right. \Rightarrow \Pi_{\beta\gamma}(r_1) = \Pi_{\beta\gamma}(r_2)$$

Problémák:

Azokat a problémákat, amelyek akkor jelennek meg, amikor túl sok információt próbálunk egyetlen relációba belegyömöszölni, *anomáliának* nevezzük. Az anomáliáknak alapvető fajtái a következők:

- *Redundancia*: Az információk feleslegesen ismétlődnek több sorban, mint például a SzállításiInformációk reláció esetében a szállító címe ismétlődik.
- *Módosítási problémák*: Megváltoztatjuk az egyik sorban tárolt információt, miközben ugyanaz az információ változatlan marad egy másik sorban. Például, ha a szállító címe változik, de csak egy sorban változtatjuk meg, nem tudjuk, melyik a jó cím. Jó tervezéssel elkerülhetjük azt, hogy ilyen hibák felmerüljenek.
- *Törlési problémák*: Ha az értékek halmaza üres halmazzá válik, akkor ennek mellékhatásaként más információt is elveszthetünk. Ha például töröljük a Rolicom által szállított összes árut, az utolsó sor törlésével elveszítjük a cég címét is.
- *Illesztési problémák*: Ha hozzáilleszteni akarunk egy szállítót, amely nem szállít egy árut sem, a szállító címét kitöltjük úgy, hogy az áruhoz „null” értékeket viszünk be, melyet majd utólag ki kell törölni, ha el nem felejtjük.

Relációk felbontása

Az anomáliák megszüntetésének elfogadott útja a relációk *felbontása* (*dekompozíció*-ja). R felbontása egyrészt azt jelenti, hogy R attribútumait szétosztjuk úgy, hogy ezáltal két új reláció sémáját alakítjuk ki belőlük. A felbontás másrészt azt is jelenti, hogyan töltjük fel a kapott két új reláció sorait az R soraiból.

Legyen egy R reláció $\{A_1, A_2, \dots, A_n\}$ sémával, R -et felbonthatjuk S és T két relációra, amelyeknek sémái $\{B_1, B_2, \dots, B_m\}$, illetve $\{C_1, C_2, \dots, C_k\}$ úgy, hogy

1. $\{A_1, A_2, \dots, A_n\} = \{B_1, B_2, \dots, B_m\} \cup \{C_1, C_2, \dots, C_k\}$, ahol

$$\{B_1, B_2, \dots, B_m\} \cap \{C_1, C_2, \dots, C_k\} \neq \emptyset.$$

2. Az S reláció sorai az R -ben szereplő összes sornak a $\{B_1, B_2, \dots, B_m\}$ -re vett vetületei, azaz R aktuális előfordulásának minden egyes t sorára vesszük a t azon komponenseit, amelyek a

$\{B_1, B_2, \dots, B_m\}$ attribútumokhoz tartoznak. Mivel a relációk halmazok, az R két különböző sorának a projekciója ugyanazt a sort is eredményezheti az S -ben. Ha így lenne, akkor az ilyen sorokból csak egyet kell belevennünk az S aktuális előfordulásába.

3. Hasonlóan, a T reláció sorai az R aktuális előfordulásában szereplő sorok $\{C_1, C_2, \dots, C_k\}$ attribútumok halmazára vett projekciói.

$$2. S = \pi_{B_1, B_2, \dots, B_m}(R); T = \pi_{C_1, C_2, \dots, C_m}(R);$$

Veszteségmentes felbontás

R reláció felbontása S és T relációkra veszteségmentes, ha

$$R = S \bowtie T$$

Fontos, hogy minden felbontás, amit normálformára hozás közben végzünk, veszteségmentes legyen, vagyis ne veszítsünk információt.

IV.2.2. Normálformák

Az adatmodellezés egyik fő célja az optimalizálás, vagyis az adatmodellt alkotó egyedtípusok lehető legjobb szerkezetének a megkeresése. Az optimális adatmodell kialakítására egyéb technikák mellett a normalizálás szolgál. A normalizálás az a folyamat, amellyel kialakítjuk a relációk normálformáját (NF).

A normálformák: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF egymásba skatulyáztak. 2NF matematikailag jobb, mint 1NF, a 4NF jobb, mint a BCNF, az 5NF a legjobb, 3NF alakú reláció szükségszerűen 1NF és 2NF alakú is. Tehát a normálalakok nem függetlenek egymástól, hanem logikusan egymásra épülnek.

Első normál forma (1NF)

Értelmezés: Egy R reláció 1NF –ben van, ha az attribútumoknak csak elemi (nem összetett vagy ismétlődő) értékei vannak. Ez minimális feltétel, melynek egy reláció eleget kell tennie, hogy a létező relációs ABKR-ek kezelni tudják.

Példa: A következő reláció nincs 1NF-ben:

Alkalmazottak:

SzemSzá m	Név	Cím			Gyerek1	SzülDát 1	...	Gyerek5	SzülDát 5
		Helysége	Utca	Szám					

Ahol a Cím összetett attribútum, a Helység, Utca és Szám attribútumokból áll. A Gyerek1, SzülDát1, Gyerek2, SzülDát2, Gyerek3, SzülDát3, Gyerek4, SzülDát4, Gyerek5, SzülDát5 ismétlődő attribútum. Egy személynek több gyereke is lehet, érdekeltek vagyunk a gyerekek keresztnévében és születési dátumukban. Jelenleg 5 gyerekről szóló információt tudunk eltárolni. Problémák az ismétlődő attribútumokkal: van olyan alkalmazott, akinek nincs egy gyereke se, nagyon soknak csak egy gyereke van, ezekenél fölöslegesen foglaljuk a háttértárolót. Jelenleg van a cégnek egy alkalmazottja, akinek 5 gyereke van, de akármikor alkalmaznak még egyet, akinek 6 gyereke van, akkor változtathatjuk a szerkezetet. □

1NF-re alakítás

Ha egy reláció nincs 1NF-ben, mivel tartalmaz összetett attribútumokat, első normál formára hozhatjuk, ha az összetett attribútum helyett beírjuk az azt alkotó elemi attribútumokat. A fenti példa esetén a Cím attribútum nem fog szerepelni a reláció attribútumai között, csak a Helység, Utca és Szám attribútumok.

Ha adott egy $R(A_1, A_2, \dots, A_n)$ reláció, mely nincs első normál formában, mivel ismétlődő attribútumokat tartalmaz, felbontással első normál formába hozható. Jelöljük az attribútumok halmazát

$$A = \{A_1, A_2, \dots, A_n\}.$$

Legyenek C és I az R reláció attribútumhalmazának részhalmazai, vagyis $C, I \subset A$, ahol C kulcs és I ismétlődő attribútumhalmaz, mely tegyük fel, hogy k -szor ismétlődik. Legyen J azon attribútumok halmaza, melyek nem részei a kulcsnak, se nem ismétlődőek, vagyis $J \subset A$, $J \cap C = \emptyset$ és $J \cap I = \emptyset$. Tehát $A = C \cup I_1 \cup I_2 \cup \dots \cup I_k \cup J$. A felbontás után kapjuk a következő két relációsémát:

$$S(C, I) \text{ és } T(C, J).$$

Vagyis az egyik relációban a kulcs attribútum mellett az ismétlődő attribútumok (csak egyszer) fognak szerepelni, a másikban pedig a kulcs mellett azon attribútumok, melyek nem ismétlődőek.

példa: A fenti példa esetén:

$$C = \{\text{SzemSzám}\}$$

$$I = \{\text{GyerekNév}, \text{SzülDátum}\}$$

$$J = \{\text{Név}, \text{Helység}, \text{Utca}, \text{Szám}\}.$$

A két új reláció:

Alkalmazott (SzemSzám, Név, Helység, Utca, Szám)

AlkalmGyerekei (SzemSzám, GyerekNév, SzülDátum)

Ebben az esetben, ha egy alkalmazottnak csak egy gyereke van az AlkalmGyerekei relációban egy sor lesz neki megfelelő, a SzemSzám attribútumnak ugyanazzal az értékével. Ha egy alkalmazottnak 5 gyereke van, 5 sor, ha ugyanakkor az alkalmazottnak még születik egy gyereke, akkor 6 sor tartalmazza az AlkalmGyerekei relációban az illető alkalmazott gyerekeit. Ha egy alkalmazottnak nincs egy gyereke se, az AlkalmGyerekei relációban nem lesz egy sor sem, mely hivatkozna rá a SzemSzám segítségével.

Második normál forma (2NF)

Értelmezés: Egy reláció 2NF formában van, ha első normál formájú (1NF) és nem tartalmaz $Y \rightarrow B$ alakú parciális függőséget, ahol B nem prim attribútum.

Amint látjuk, csak akkor tevődik fel, hogy egy reláció nincs 2NF-ben, ha a kulcs összetett.

példa: A SzállításiInformációk relációja nincs 2NF-ben, mivel a reláció kulcsa a {SzállID, ÁruID} és fennáll a SzállID $\rightarrow$ SzállNév, tehát SzállNév függ a kulcs egy részétől is, tehát létezik parciális függőség.

Megoldás: több relációra kell bontani.

2NF-re alakítás

Legyen R egy reláció, mely attribútumainak a halmaza $A = \{A_1, A_2, \dots, A_n\}$ és $C \subset A$ egy kulcs. Ha a reláció nincs második normál formában, azt jelenti létezik egy $B \subset A$ nem kulcs $B \cap C = \emptyset$ attribútumhalmaz, mely függ funkcionálisan a kulcs egy részétől, vagyis létezik $D \subset C$, úgy hogy $D \rightarrow B$.

Az R relációt felbontjuk két relációra, melyek sémái:

$$T(D, B) \text{ és } S(A - B)$$

példa: Amint láttuk a 3.4. példa SzállításiInformációk relációjában fennállnak a:

$$\text{SzállID} \rightarrow \{\text{SzállNév}, \text{SzállCím}\}$$

$$\text{ÁruID} \rightarrow \{\text{ÁruNév}, \text{MértEgys}\}$$

funkcionális függőségek, a kulcs pedig a $C = \{\text{SzállID}, \text{ÁruID}\}$.

Első lépésben $B = \{\text{SzállNév}, \text{SzállCím}\}$, $D = \{\text{SzállID}\}$. Felbontás után kapjuk a

Szállítók (SzállID, SzállNév, SzállCím) és

SzállInf (SzállID, ÁruID, ÁruNév, MértEgys, Ár)

relációkat.

A Szállítók reláció 2NF-ben van, mivel a kulcs nem összetett, fel sem tevődik, hogy valamely attribútum függjön a kulcs egy részétől.

A SzállInf nincs 2NF-ben, mert fennáll a

$\text{ÁruID} \rightarrow \{\text{ÁruNév}, \text{MértEgys}\}.$

Ebben az esetben $B = \{\text{ÁruNév}, \text{MértEgys}\}$, $D = \{\text{ÁruID}\}$. Tovább bontjuk a következő két relációra:

$\text{Áruk} (\text{ÁruID}, \text{ÁruNév}, \text{MértEgys}),$
 $\text{Szállít} (\text{SzállID}, \text{ÁruID}, \text{Ár}).$

Az Áruk 2NF-ben van, mert a kulcs nem összetett és 1NF-ben van. A Szállít relációban egyetlen nem kulcs attribútum van: az Ár, és az nem függ csak az ÁruID-től, mert különböző szállító különböző árban ajánlhatja ugyanazt az árut, sem a SzállID-től nem függ funkcionálisan, mert egy szállító nem ajánlja ugyanabban az árban az összes árut. A kapott relációk:

Szállítók:

<i>SzállID</i>	<i>SzállNév</i>	<i>SzállCím</i>
111	Rolicom	A. Iancu 15
222	Sorex	22 dec. 6

Áruk:

<i>ÁruID</i>	<i>ÁruNév</i>	<i>MértEgys</i>
45	Milka csoki	tábla
67	Heidi csoki	tábla
56	Milky way	rúd

Szállít:

<i>SzállID</i>	<i>ÁruID</i>	<i>Ár</i>
111	45	25000
222	45	26500
111	67	17000
111	56	20000
222	67	18000
222	56	22500

Harmadik normál forma (3NF)

Értelmezés: Egy R reláció *harmadik normál formában* (3NF) van, ha második normál formában van és nem tartalmaz $Y \rightarrow B$ alakú *transzitiv funkcionális* függőséget, ahol B nem prim attribútum.

Értelmezés: Egy R reláció *harmadik normál formában* (3NF) van, ha létezik az R -ben egy $Y \rightarrow B$ alakú nem triviális funkcionális függőség, akkor Y az R reláció superkulcsa vagy a B prim attribútum (valamelyik kulcsnak része).

A két értelmezés ekvivalens. A második nem kéri a második normál formát, de mivel bármely létező $Y \rightarrow B$ funkcionális függőség esetén a bal oldal superkulcs, nem lehet annak része. Tehát elég, ha az összes létező funkcionális függőség esetén a bal oldal superkulcs, akkor a transzitiv függőség nem létezhet, mert a transzitiv függőség esetén a bal oldal nem kulcs és ez nem megengedett.

példa: A Rendelések reláció nincs 3NF-ben, mivel tartalmaz transzitiv funkcionális függőséget.

$\text{RendelésSzám} \rightarrow \text{VevőID}$

$\text{VevőID} \rightarrow \text{VevőNév}.$

Probléma, ha így ábrázoljuk a rendeléseket, hogy ha egy vevő több rendelést is elhelyez, ami lehetséges, akkor a vevő nevét ismétljük. Megoldás: 2 relációra bontjuk a relációt, mely nincs 3NF-ben. $\square$

3NF-re alakítás

Legyen R egy reláció, mely 2NF-ben van, viszont nincs 3NF-ben, attribútumainak a halmaza $A = \{A_1, A_2, \dots, A_n\}$ és $C \subset A$ elsődleges kulcs. Ha a reláció nincs harmadik normál formában, azt jelenti,

hogy létezik egy $B \subset A$ nem kulcs $B \cap C = \emptyset$ attribútumhalmaz, mely tranzitív függőséggel függ a kulcstól, vagyis létezik D , úgy hogy $C \rightarrow D$ és $D \rightarrow B$. Mivel a reláció 2NF-ben van, B nem függ funkcionálisan C -nek egy részétől, tehát D nem kulcs attribútum.

Az R relációt felbontjuk két relációra, melyek sémái:

$$T(D, B) \text{ és } S(A - B).$$

példa: A Rendelések reláció esetén: $B = \{\text{VevőNév}\}$, $D = \{\text{VevőID}\}$, a felbontás után kapott relációk:

Vevők (VevőID, VevőNév)
RendelésInf (RendelésSzám, Dátum, VevőID)

Egy adatbázis modell kialakítása szempontjából a legkedvezőbb, ha az adatbázist alkotó relációk 3NF -ben vannak.

IV.3.Relációs algebra

A relációs algebrai műveletek operandusai a relációk. A relációt a nevével szokták megadni, például R vagy Alkalmazottak. A műveletek operátorait a következőkben részletezzük. Az operátorokat alkalmazva a relációkra, eredményként szintén relációkat kapunk, ezekre ismét alkalmazhatunk relációs algebrai operátorokat, így egyre bonyolultabb kifejezésekhez jutunk. Egy lekérdezés tulajdonképpen egy relációs algebrai kifejezés. A relációs algebrai műveletek esetén szükségünk lesz feltételekre. A feltételek a következő típusúak lehetnek:

$$\langle \text{attribútum_név} \rangle \left\{ \begin{array}{l} = \\ < \\ < \\ <= \\ > \\ >= \end{array} \right\} \left\{ \begin{array}{l} \langle \text{attribútum_név} \rangle \\ \langle \text{konstans} \rangle \end{array} \right\}$$

$$\left\{ \begin{array}{l} \langle \text{attribútum_név} \rangle \\ \langle \text{konstans} \rangle \end{array} \right\} \left\{ \begin{array}{l} \text{IS IN} \\ \text{IS NOT IN} \end{array} \right\} \langle \text{reláció} \rangle \text{ (melynek egy attribútuma van)}$$

NOT <feltétel>

$$\langle \text{feltétel} \rangle \left\{ \begin{array}{l} \text{OR} \\ \text{AND} \end{array} \right\} \langle \text{feltétel} \rangle$$

A továbbiában lássuk a relációs algebra műveleteit. Az első öt az alapvető művelet, a következőket ki tudjuk fejezni az első öt segítségével.

1) *Kiválasztás (Selection)*: Az R relációra alkalmazott *kiválasztás* operátor f feltétellel olyan új relációt hoz létre, melynek sorai teljesítik az f feltételt. Az eredmény reláció attribútumainak a száma megegyezik az R reláció attribútumainak a számával. Jelölés: $\sigma_f(R)$.

példa: Keressük a kis keresetű alkalmazottakat (akinek kisebb, vagy egyenlő a fizetése 500 euró-val). A lekérdezés a következő:

$$\sigma_{\text{Fizetés} \leq 500} (\text{Alkalmazottak})$$

A lekérdezés eredménye:

<i>SzemSzám</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés</i>
111111	Nagy Éva	2	300
222222	Kiss Csaba	9	400
333333	Kovács István	2	500

példa: Keressük a 9-es részleg nagy fizetésű alkalmazottait (akinek 500 euró-nál nagyobb a fizetése). A lekérdezés: $\sigma_{\text{Fizetés} > 500 \text{ AND RészlegID} = 9}(\text{Alkalmazottak})$

Az eredmény:

<i>SzemSzáma</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés</i>
456777	Szabó János	9	900

2) *Vetítés (Projection)*: Adott R egy reláció $A_1, A_2, \dots, A_n$ attribútumokkal. A vetítés művelet eredményeként olyan relációt kapunk, mely R -nek csak bizonyos attribútumait tartalmazza. Ha kiválasztunk k attribútumot az n -ből: $A_{i_1}, A_{i_2}, \dots, A_{i_k}$ -et, és ha esetleg a sorrendet is megváltoztatjuk, az eredmény reláció a kiválasztott k attribútumhoz tartozó oszlopokat fogja tartalmazni, viszont az összes sorból. Mivel az eredmény is egy reláció, nem lehet két azonos sor a vetítés eredményében, az azonos sorokból csak egy marad az eredmény relációban.

Jelölés: $\pi_{A_{i_1}, A_{i_2}, \dots, A_{i_k}}(R)$

példa: Ha az Alkalmazottak relációból csak az alkalmazott neve és fizetése érdekel, akkor a következő művelet eredménye a kért reláció:

$\pi_{\text{Név, Fizetés}}(\text{Alkalmazottak})$

példa: Legyen ismét a Diákok tábla:

```
CREATE TABLE Diákok (
  BeiktatásiSzám INT PRIMARY KEY,
  Név VARCHAR(50),
  Cím VARCHAR(100),
  SzületésiDatum DATE,
  CsopKod CHAR(3) REFERENCES Csoportok (CsopKod),
  Átlag REAL
);
```

A következő vetítés:

$\pi_{\text{CsopKod}}(\text{Diákok})$

eredménye az összes létező csoportkod a Diákok táblából. Ha egy csoportkod többször is megjelenik a Diákok táblában, a vetítésben csak egyszer fog szerepelni. (Például a Diákok táblában 25 sor esetén a csoportkod '531'-es, a vetítés eredményében csak egyszer fog az '531'-es csoportkod szerepelni.)

3) *Descartes szorzat*. Ha adottak az R_1 és R_2 relációk, a két reláció Descartes szorzata ($R_1 \times R_2$) azon párok halmaza, amelyeknek első eleme az R_1 tetszőleges eleme, a második pedig az R_2 egy eleme. Az eredményreláció sémája az R_1 és R_2 sémájának egyesítése.

Legyen R_1 reláció:

<i>A</i>	<i>B</i>
12	33
24	46

Legyen R_2 reláció:

<i>B</i>	<i>C</i>	<i>D</i>
20	55	80
30	67	97
40	75	99

Akkor $R_1 \times R_2$ eredménye:

<i>A</i>	<i>R₁.B</i>	<i>R₂.B</i>	<i>C</i>	<i>D</i>
12	33	20	55	80
12	33	30	67	97
12	33	40	75	99
24	46	20	55	80
24	46	30	67	97
24	46	40	75	99

4) *Egyesítés*. Ha adottak az R_1 és R_2 relációk, R_1 és R_2 attribútumainak a száma megegyezik, és ugyanabban a pozícióban levő attribútumnak ugyanaz az értékhalmaza, a két reláció egyesítése tartalmazni fogja R_1 és R_2 sorait. Az egyesítésben egy elem csak egyszer szerepel, még akkor is, ha jelen van R_1 - és R_2 -ben is (jelölés: $R_1 \cup R_2$).

5) *Különbség*. Ha adottak az R_1 és R_2 relációk, R_1 és R_2 attribútumainak a száma megegyezik és ugyanabban a pozícióban levő attribútumnak ugyanaz az értékhalmaza, a két reláció különbsége azon sorok halmaza, amelyek R_1 -ben szerepelnek és R_2 -ben nem (jelölés: $R_1 - R_2$). A különbség eredményét grafikusan ábrázolva:

példa: Legyen R_1 :

<i>SzemSzám</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés (euró)</i>
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500

és legyen R_2 :

<i>SzemSzám</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés (euró)</i>
111111	Nagy Éva	2	300
456777	Szabó János	9	900
123444	Vincze Ildikó	1	800

akkor $R_1 \cup R_2$:

<i>SzemSzám</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés (euró)</i>
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500
111111	Nagy Éva	2	300
123444	Vincze Ildikó	1	800

illetve $R_1 - R_2$:

<i>SzemSám</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés (euró)</i>
222222	Kiss Csaba	9	400
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500

Ez az öt az alapvető művelet. Még vannak hasznos műveletek: ezek az öt alapvető művelettel kifejezhetők.

6) *Metszet*: Legyenek az R_1 és R_2 relációk, a két reláció metszete:

$$R_1 \cap R_2 = R_1 - (R_1 - R_2).$$

7) *Théta-összekapcsolás* (θ -Join): Legyenek az R_1 és R_2 relációk. A Théta-összekapcsolás során az R_1 és R_2 relációk Descartes szorzatából kiválasztjuk azon sorokat, melyek eleget tesznek a θ feltételnek, vagyis: $R_1 \bowtie_{\theta} R_2 = \sigma_{\theta}(R_1 \times R_2)$.

példa: Legyenek R_1 és R_2 a következő relációk, számítsuk ki: $R_1 \bowtie_{A < D} R_2$

R_1 reláció:

<i>A</i>	<i>B</i>	<i>C</i>
11	23	32
65	76	82
97	76	82

R_2 reláció:

<i>B</i>	<i>C</i>	<i>D</i>
23	32	44
23	32	57
76	82	99

$R_1 \bowtie_{A < D} R_2$:

<i>A</i>	<i>R₁.B</i>	<i>R₁.C</i>	<i>R₂.B</i>	<i>R₂.C</i>	<i>D</i>
11	23	32	23	32	44
11	23	32	23	32	57
11	23	32	76	82	99
65	76	82	76	82	99
97	76	82	76	82	99

8) *Természetes összekapcsolás* (Natural join): Legyenek az R_1 és R_2 relációk. A természetes összekapcsolás művelete akkor alkalmazható, ha az R_1 és R_2 relációknak egy vagy több közös attribútuma van. Legyen B az R_1 , illetve C az R_2 reláció attribútumainak a halmaza, a közös attribútumok pedig: $B \cap C = \{A_1, A_2, \dots, A_p\}$. A természetes összekapcsolást a következő képlettel fejezhetjük ki:

$$R_1 \bowtie R_2 = \pi_{B \cup C}(R_1 \bowtie_{(R_1.A_1=R_2.A_1) \wedge (R_1.A_2=R_2.A_2) \wedge \dots \wedge (R_1.A_p=R_2.A_p)} R_2),$$

ahol $R_i.A_j$ jelöli az A_j attribútumot az R_i relációból, $i \in \{1, 2\}, j \in \{1, 2, \dots, p\}$.

példa: Legyenek R_1 és R_2 relációk a Théta-összekapcsolás példából, a természetes összekapcsolás eredménye:

$R_1 \bowtie R_2$ eredménye:

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
11	23	32	44
11	23	32	57
65	76	82	99
97	76	82	99

R_1 és R_2 relációk természetes összekapcsolása esetén azokat a sorokat párosítjuk össze, amelyek értékei az R_1 és R_2 sémájának összes közös attribútumán megegyeznek. Legyen r_1 az R_1 egy sora és r_2 az R_2 egy sora, ekkor az r_1 és r_2 párosítása akkor sikeres, ha az r_1 és r_2 megfelelő értékei megegyeznek az összes $A_1, A_2, \dots, A_p$ közös attribútumon. Ha az r_1 és r_2 sorok párosítása sikeres, akkor a párosítás eredményét *összekapcsolt sornak* nevezzük. Az összekapcsolt sor megegyezik az r_1 sorral az R_1 összes attribútumán és r_2 sorral az R_2 összes attribútumán. Az $R_1 \bowtie R_2$ eredményében R_1 és R_2 közös attribútumai csak egyszer szerepelnek.

Egy olyan sort, melyet nem lehet sikeresen párosítani az összekapcsolásban szereplő másik reláció egyetlen sorával sem, *lógó* (dangling) sornak nevezzük

példa: Legyenek a Szállítók, Áruk és Szállít relációk. Ha az összes szállítási információra van szükségünk, akkor kiszámítjuk a Szállít $\bowtie$ Szállítók $\bowtie$ Áruk természetes összekapcsolást, melynek eredménye:

Szállítók:

<i>SzállID</i>	<i>SzállNév</i>	<i>SzállCím</i>
111	Rolicom	A.Iancu 15
222	Sorex	22 dec. 6

Áruk:

<i>ÁruID</i>	<i>ÁruNév</i>	<i>MértEgys</i>
45	Milka csoki	tábla
67	Heidi csoki	tábla
56	Milky way	rúd

Szállít:

<i>SzállID</i>	<i>ÁruID</i>	<i>Ár</i>
111	45	25000
222	45	26500
111	67	17000
111	56	20000
222	67	18000
222	56	22500

Szállít $\bowtie$ Szállítók $\bowtie$ Áruk eredménye:

<i>SzállID</i>	<i>SzállNév</i>	<i>SzállCím</i>	<i>ÁruID</i>	<i>ÁruNév</i>	<i>MértEgys</i>	<i>Ár</i>
111	Rolicom	A.Iancu 15	45	Milka csoki	Tábla	25000
222	Sorex	22 dec. 6	45	Milka csoki	Tábla	26500
111	Rolicom	A.Iancu 15	67	Heidi csoki	Tábla	17000
111	Rolicom	A.Iancu 15	56	Milky way	Rúd	20000
222	Sorex	22 dec. 6	67	Heidi csoki	Tábla	18000
222	Sorex	22 dec. 6	56	Milky way	Rúd	22500

Relációs algebrai műveletek alkalmazásával újabb relációkat kapunk. Gyakran szükséges egy olyan operátor, amelyik átnevezi a relációkat.

9) *Átnevezés*: Legyen $R(A_1, A_2, \dots, A_n)$ egy reláció, az átnevezés operátor:

$\rho_{S(B_1, B_2, \dots, B_n)}(R)$ az R relációt S relációvá nevezi át, az attribútumokat pedig balról jobbra $B_1, B_2, \dots, B_n$ -né. Ha az attribútum neveket nem akarjuk megváltoztatni, akkor $\rho_S(R)$ operátort használunk.

10) *Hányados (Quotient)*: Legyen R_1 reláció sémája: $\{X_1, X_2, \dots, X_m, Y_1, Y_2, \dots, Y_n\}$, R_2 reláció sémája pedig: $\{Y_1, Y_2, \dots, Y_n\}$, tehát $Y_1, Y_2, \dots, Y_n$ közös attribútumok ugyanazon értékhalmazzal, és R_1 -nek még van pluszba m attribútuma: $X_1, X_2, \dots, X_m$, R_2 -nek pedig a közösekén kívül nincs más attribútuma. R_1 az osztandó, R_2 az osztó. Jelöljük X -szel és Y -nal a következő attribútumhalmazokat: $X = \{X_1,$

$X_2, \dots, X_m\}$, $Y = \{Y_1, Y_2, \dots, Y_n\}$. Ebben az esetben jelöljük: $R_1(X, Y)$, $R_2(Y)$ a két relációt, melynek hányadosát jelöljük:

$$R_1 \text{ DIVIDE BY } R_2(X) \text{ -el}$$

Tehát a hányados reláció sémája $\{X_1, X_2, \dots, X_m\}$. A hányados relációban megjelenik egy x sor, ha minden y sorra az R_2 -ből az R_1 -ben megjelenik egy r_1 sor, melyet az x és y sorok összeragasztásából kapunk.

Másként fogalmazva, legyen 2 reláció, egy bináris és egy unáris, az osztás eredménye a bináris reláció azon attribútumait tartalmazza, melyek különböznek az unáris reláció attribútumaitól, és a bináris relációból az attribútumok azon értékeit, melyek megegyeznek az unáris reláció összes attribútum értékével.

példa: Legyen $A = \pi_{\text{ÁruID}}(\text{Áruk})$, $S = \pi_{\text{SzállID}, \text{ÁruID}}(\text{Szállít})$ és a következő sorok az S relációban:

<i>SzállID</i>	<i>ÁruID</i>
S1	A1
S1	A2
S1	A3
S1	A4
S1	A5
S1	A6
S2	A1
S2	A2
S3	A2
S4	A2
S4	A4
S4	A5

a) Legyen A reláció:

<i>ÁruID</i>
A1

akkor az $S \text{ DIVIDE } A(\text{SzállID})$ eredménye:

<i>SzállID</i>
S1
S2

b) esetben A reláció:

<i>ÁruID</i>
A2
A4

akkor $S \text{ DIVIDE } A(\text{SzállID})$:

<i>SzállID</i>
S1
S4

c) esetben A reláció:

<i>ÁruID</i>

A1
A2
A3
A4
A5
A6

akkor $S \text{ DIVIDE } A(\text{SzállID})$:

SzállID
S1

IV.4. Az SQL lekérdezőnyelv

A legtöbb relációs ABKR az adatbázist az SQL-nek (Structured Query Language) nevezett lekérdezőnyelv segítségével kérdezi le és módosítja. Az SQL központi magja ekvivalens a relációs algebrával, de sok kiterjesztést dolgoztak ki hozzá, mint például az összesítések.

Az SQL-nek számos verziója ismeretes, szabványokat is dolgoztak ki, ezek közül a legismertebb az SQL-92 vagy SQL2. A napjainkban használt ABKR-ek lekérdezőnyelvei ezt a szabványt tartják be. Az SQL egy új szabványa az SQL3, mely rekurzióval, objektumokkal, triggerekkel stb. terjeszti ki az SQL2-őt. Számos kereskedelmi ABKR már meg is valósította az SQL3 néhány javaslatát.

IV.4.1. Egyszerű lekérdezések SQL-ben

A relációs algebra vízszintes kiválasztás műveletét:

$$\sigma_f(R)$$

az SQL a SELECT, FROM és WHERE kulcsszavak segítségével valósítja meg a következőképpen:

```
SELECT *
FROM R
WHERE f;
```

példa: Legyen a NagyKer nevű adatbázis a következő relációsémákkal:

```
Részlegek (RészlegID, Név, Helység, ManSzemSzám);
Alkalmazottak (SzemSzám, Név, Fizetés, Cím, RészlegID);
Managerek (SzemSzám);
Árucsoportok (CsopID, Név, RészlegID);
Áruk (ÁruID, Név, MértEgys, MennyRakt, CsopID);
Szállítók (SzállID, Név, Helység, UtcaSzám);
Vevők (VevőID, Név, Helység, UtcaSzám, Mérleg, Hihetőség);
Szállít (SzállID, ÁruID, Ár);
Szerződések (SzerződID, Dátum, Részletek, VevőID);
Tételek (TételID, Dátum, SzerződID);
Szerepel (TételID, ÁruID, RendMenny, SzállMenny).
```

Legyen a következő lekérdezés:

„Keressük azon alkalmazottakat, akik a 9-es részlegnél dolgoznak és a fizetésük nagyobb, mint 500 euro”.

```
SELECT *
FROM Alkalmazottak
WHERE RészlegID = 9 AND Fizetés > 500; □
```

A FROM kulcsszó után adhatjuk meg azokat a relációkat, jelen esetben csak egyet, melyre a lekérdezés vonatkozik, a fenti példa esetén az Alkalmazottak reláció.

A kiválasztás feltételét a WHERE kulcsszó után tudjuk megadni. A példánk esetében azok a sorok fognak a lekérdezés eredményében megjelenni, melyek eleget tesznek a WHERE után megadott feltételnek, vagyis az alkalmazott RészlegID attribútumának az értéke 9 és a Fizetés attribútum értéke nagyobb, mint 500.

A SELECT kulcsszó utáni * azt jelenti, hogy az eredmény reláció fogja tartalmazni a FROM után megadott reláció összes attribútumát.

Az SQL nyelv nem különbözteti meg a kis és nagy betűket. Nem szükséges új sorba írni a FROM és WHERE kulcsszavakat, általában a fenti módon szokták megadni, de lehet egy sorban kis betűkkel is.

```
select * from alkalmazottak where részlegID = 9 and fizetés > 500;
```

A relációs algebra vetítés művelete

$$\pi_{A_{i_1}, A_{i_2}, \dots, A_{i_k}}(R)$$

a SELECT-SQL parancs segítségével a következőképpen adható meg:

```
SELECT  Ai1, Ai2, ..., Aik
FROM R;
```

A SELECT kulcsszó után megadhatjuk az R reláció bármely attribútumát és az eredmény sorok ezen attribútumokat fogják csak tartalmazni, ugyanazzal a névvel, amivel az R relációban szerepelnek.

példa: Legyen a következő relációs algebrai lekérdezés:

$$\pi_{\text{Név}, \text{Fizetés}}(\text{Alkalmazottak})$$

SELECT-SQL parancs segítségével a következőképpen írható fel:

```
SELECT Név, Fizetés
FROM Alkalmazottak; □
```

A lekérdezés feldolgozása során a FROM kulcsszó után megadott relációt a feldolgozó végigjárja, minden sor esetén ellenőrzi a WHERE kulcsszó után megadott feltétel teljesül-e. Azon sorokat, melyek esetén a feltétel teljesül, az eredmény relációba helyezzük. A feldolgozás hatékonyságát növeli, ha a feltételben szereplő attribútumok szerint létezik indexállomány.

A vetítés során kapott eredmény reláció esetén *megváltoztathatjuk az attribútumok neveit* az AS kulcsszó segítségével, ha a FROM után szereplő reláció attribútum nevei nem felelnek meg. Az AS nem kötelező. A SELECT kulcsszó után kifejezést is használhatunk.

példa: Ha például a fizetést nem euró-ban, hanem \$-ban szeretnénk és az euró/dollár arányt mondjuk 1.1, akkor a nagy fizetésű alkalmazottakat a 9-es részlegből a következő paranccsal kapjuk meg:

```
SELECT Név AS Név9, Fizetés * 1.1 AS Fizetes$
FROM Alkalmazottak
WHERE RészlegID = 9 AND Fizetés > 500;
```

Tehát az eredmény reláció két oszlopot fog tartalmazni, melyek nevei: Név9, illetve Fizetés\$. □

A WHERE kulcsszó utáni feltétel lehet *összetett*, használhatjuk az AND, OR és NOT logikai műveleteket. A műveletek sorrendjének a meghatározására használhatunk zárójeleket, ha ezek megelőzési sorrendje nem felel meg. Az SQL nyelvben is, mint a legtöbb programozási nyelvben a NOT megelőzi az AND és OR műveletet, az AND pedig az OR-t.

példa: „Keressük a 3-as és 6-os részleg alkalmazottait akiknek kicsi a fizetése, 200 eurónál kisebb.” A következő paranccsal kapjuk meg:

```
SELECT Név, Fizetés
FROM Alkalmazottak
WHERE (RészlegID = 3 OR RészlegID = 6) AND Fizetés < 200;
```

Ha a zárójelet nem tettük volna ki, akkor csak a 6-os részlegből válogatta volna ki a kis fizetésűeket, és az eredmény relációban a 3-as részlegből az összes alkalmazott szerepelt volna. □

Karakterláncok összehasonlítása esetén használhatjuk a LIKE kulcsszót, hogy a karakterláncokat egy mintával hasonlítsunk össze a következőképpen:

k LIKE m

ahol k egy karakterlánc és m egy minta. A mintában használhatjuk a % és _ karaktereket. A % jelnek a k-ban megfelel bármilyen karakter 0 vagy nagyobb hosszúságú sorozata. Az _ jelnek megfelel egy akármilyen karakter a k-ból. A LIKE kulcsszó segítségével képezett feltétel igaz, ha a k karakterlánc megfelel az m mintának.

példa: SELECT *

```
FROM Alkalmazottak
WHERE Név LIKE 'Kovács%';
```

A lekérdezés eredménye azon alkalmazottakat tartalmazza, kiknek a neve a 'Kovács' karaktersorral kezdődik. Megkapjuk az összes Kovács vezetéknévű alkalmazottat, de a 'Kovácsovics' vezetéknévűt is, ha ilyen létezik az adatbázisban. Ha csak a Kovács vezetéknévűeket akarjuk, akkor a 'Kovács %' mintát használjuk. □

Használhatjuk a

k NOT LIKE m

szűrő feltételt is.

Más szűrőfeltételek a BETWEEN és IN kulcsszóval képezhetők. A BETWEEN kulcsszó segítségével megadunk egy intervallumot, és azt vizsgáljuk, hogy adott oszlop, mely értéke esik a megadott intervallumba. (Az oszlop itt szintén lehet származtatott oszlop, kifejezés.)

WHERE <oszlop> BETWEEN <kifejezés_1> AND <kifejezés_2 >

példa: SELECT Név

```
FROM Alkalmazottak
WHERE Fizetés BETWEEN 300 AND 500;
```

Ugyanazt az eredményt adja, mint a:

```
SELECT Név
FROM Alkalmazottak
WHERE Fizetés >= 300 AND Fizetés <=500; □
```

Az IN operátor után megadunk egy értéklistát, és azt vizsgáljuk, hogy az adott oszlop mely mezőinek értéke egyezik az adott lista valamelyik elemével. (Az oszlop lehet származtatott oszlop, kifejezés is.)

WHERE <oszlop> IN (<kifejezés_1>, <kifejezés_2> [,...])

példa: Legyen az Egyetem nevű adatbázis a következő relációsémákkal:

```
Szakok (SzakKod, SzakNév, Nyelv);
Csoportok (CsopKod, Evfolyam, SzakKod);
Diákok (BeiktatásiSzám, Név, SzemSzám, Cím, SzületésiDatum, CsopKod,
        Átlag);
TanszékCsoportok (TanszékCsopKod, Név);
Tanszékek (TanszékKod, Név, TanszékCsopKod);
Beosztások (BeosztásKod, Név);
Tanárok (TanárKod, Név, SzemSzám, Cím, PhD, TanszékKod, BeosztásKod,
        Fizetés);
Tantárgyak (TantKod, Név);
Tanít (TanárKod, TantKod);
Jegyek (BeiktatásiSzám, TantKod, Datum, Jegy)
```

A diákok összes jegyét eltároljuk a Jegyek relációban, több szemeszterben sok jegye van egy diáknak. A Diákok táblában az utolsó szemeszter vagy utolsó év átlaga szerepel az Átlag oszlopban, ami

alapján eldöntik például, hogy kap-e a diák bentlakást, ösztöndíjat stb.

Keressük az '531'-es, '532'-s és '631'-es csoportok diákjait:

```
SELECT Név
FROM Diákok
WHERE CsopKod IN ('531', '532', '631'); □
```

A SELECT SQL parancs lehetőséget ad az eredmény reláció rendezésére az ORDER BY kulcsszavak segítségével. Alapértelmezés szerint növekvő sorrendben történik a rendezés, de ha csökkenő sorrendet szeretnénk, akkor a DESC kulcsszót használhatjuk.

példa: Ha a fenti lekérdezést kiegészítjük azzal, hogy a diákokat csoporton belül, névsor szerinti sorrendben akarjuk megadni, akkor a SELECT parancsot kiegészítjük az ORDER BY után a megfelelő attribútumokkal a következőképpen:

```
SELECT Név
FROM Diákok
WHERE CsopKod IN ('531', '532', '631')
ORDER BY CsopKod, Név; □
```

példa: A diákokat átlag szerint csökkenő sorrendben adja meg:

```
SELECT Név
FROM Diákok
ORDER BY Átlag DESC; □
```

IV.4.2. Több relációra vonatkozó lekérdezések

A relációs algebra egyik fontos tulajdonsága, hogy a műveletek eredménye szintén reláció, és az eredmény operandus lehet a következő műveletben. Az SELECT-SQL is kihasználja ezt, a relációkat összekapcsolhatjuk, egyesíthetjük, metszetet vagy különbséget is számíthatunk.

A Descartes szorzat

$$R_1 \times R_2$$

műveletét a következő SQL parancs valósítja meg:

```
SELECT *
FROM R1, R2;
```

A Théta-összekapcsolást:

$$R_1 \bowtie_{\theta} R_2$$

a következő paranccsal adhatjuk meg:

```
SELECT *
FROM R1, R2
WHERE  $\theta$ ;
```

A leggyakrabban használt műveletet, a természetes összekapcsolást

$$R_1 \bowtie R_2 = \pi_{B \cup C} (R_1 \bowtie_{(R_1.A_1=R_2.A_1) \wedge (R_1.A_2=R_2.A_2) \wedge \dots \wedge (R_1.A_p=R_2.A_p)} R_2),$$

a következőképpen írhatjuk SQL-ben:

```
SELECT *
FROM R1, R2
WHERE R1.A1 = R2.A1 AND R1.A2 = R2.A2 AND ... AND R1.Ap = R2.Ap ;
```

Ebben az általános esetben a két összekapcsolandó relációnak p darab közös attribútuma van. A gyakorlatban általában a két relációnak egy közös attribútuma van. Amint látjuk, ha több relációban is szerepel ugyanaz az attribútum név, előtagként a reláció nevét használjuk.

példa: Legyenek a következő relációk:

```
Csoportok (CsopKod, Evfolyam, SzakKod);
Diákok (BeiktatásiSzám, Név, Cím, SzületésiDatum, CsopKod, Átlag);
```

Ha a diákok esetén szeretnénk kiírni az évfolyamot és szakkódot is, akkor ezt a következő SQL parancs segítségével érjük el:

```
SELECT Név, CsopKod, Evfolyam, SzakKod
FROM Diákok, Csoportok
WHERE Diákok.CsopKod = Csoportok.CsopKod;
```

Tehát a WHERE kulcsszó után megadjuk a join feltételt. Ha elfelejtjük a join feltételt az eredmény Descartes szorzat lesz, melynek méretei nagyon nagyok lehetnek.

Vannak olyan ABKR-ek, melyek az előbbi feladat megoldására a JOIN kulcsszót is elfogadják (pl. MS SQL Server):

```
SELECT Név, CsopKod, Evfolyam, SzakKod
FROM Diákok INNER JOIN Csoportok
ON Diákok.CsopKod = Csoportok.CsopKod;
```

Később látjuk majd az OUTER JOIN-t is.□

Amint az egyszerű lekérdezéseknél láttuk, a WHERE kulcsszó után a kiválasztás feltételét adtuk meg. Ha több reláció összekapcsolása mellett kiválasztás műveletet is meg akarunk adni, a join feltétel után AND logikai művelettel a kiválasztás feltételét is megadhatjuk.

példa: Az összes harmadéves diák nevét a következő paranccsal is megkaphatjuk:

```
SELECT Név
FROM Diákok, Csoportok
WHERE Diákok.CsopKod = Csoportok.CsopKod
AND Evfolyam = 3; □
```

Több mint két relációt is összekapcsolhatunk természetes összekapcsolással, fontos, hogy az összes join feltételt megadjuk. Ha az összekapcsolandó relációk száma k , és minden két-két relációnak egy-egy közös attribútuma van, akkor a join feltételek száma $k-1$. Ha tehát 4 relációt kapcsolunk össze, a join feltételek száma minimum 3.

példa: A NagyKer nevű adatbázisra vonatkozóan legyen a következő lekérdezés:

„Adjuk meg azon szállítók nevét és címét, kik szállítanak édességet” (ÁruCsoportok.Név = 'édesség')

```
SELECT Szállítók.Név, Szállítók.Helység, Szállítók.UtcaSzám
FROM ÁruCsoportok, Áruk, Szállítók
WHERE ÁruCsoportok.CsopID = Áruk.CsopID
AND Áruk.ÁruID = Szállítók.ÁruID
AND Szállítók.SzállID = Szállítók.SzállID
AND ÁruCsoportok.Név = 'édesség'; □
```

Az SQL lehetőséget ad arra, hogy a FROM záradékban szereplő R relációhoz hozzárendeljünk egy másodnevet, melyet *sorváltozónak* nevezünk. Sorváltozót akkor használunk, ha rövidebb vagy más nevet akarunk adni a relációnak, illetve ha a FROM után kétszer is ugyanaz a reláció szerepel. Ha használtunk másodnevet, akkor az adott lekérdezésben azt kell használnunk.

példa: Keressük azon alkalmazottakat, akik ugyanazon a címen laknak, például férj és feleség, vagy szülő és gyerek.

```
SELECT Alk1.Név AS Név1, Alk2.Név AS Név2
FROM Alkalmazottak AS Alk1, Alkalmazottak AS Alk2
WHERE Alk1.Cím = Alk2.Cím
AND Alk1.Név < Alk2.Név;
```

A lekérdező feldolgozó ugyanazt a relációt kell kétszer bejárja, hogy a kért párokat megtalálja. Ha az $Alk1.Név < Alk2.Név$ feltételt nem tettük volna, akkor minden alkalmazott bekerülne az eredménybe önmagával is párosítva. Ezt esetleg a $\diamond$ feltétellel is megoldhattuk volna, de akkor egy férj–feleség páros kétszer is bekerült volna, csak más sorrendben. Például: ('Kovács István', 'Kovács Sára') és ('Kovács Sára', 'Kovács István') is. Mivel gyereknek lehet ugyanaz a neve, mint a szülőnek, ezért jobb megoldás a: $Alk1.Név < Alk2.Név$ feltételt kicserélni a következő feltétellel:

$Alk1.SzemSzám < Alk2.SzemSzám$; $\square$

Algoritmus egy egyszerű SELECT–SQL lekérdezés kiértékelésére:

Input: $R_1, R_2, \dots, R_n$ relációk a FROM záradék után

Begin

Minden t_1 sorra az R_1 -ből

Minden t_2 sorra az R_2 -ből

...

Minden t_n sorra az R_n -ből

Ha a WHERE záradék igaz a $t_1, t_2, \dots, t_n$ attribútumainak az értékeire

Akkor

A SELECT záradék attribútumainak értékeiből alkotott sort az eredményhez adjuk

End

A relációs algebra *halmazműveleteit* (egyesítés, metszet és különbség) használhatjuk az SQL nyelvben, azzal a feltétellel, hogy az operandus relációknak ugyanaz legyen az attribútumhalmaza. A megfelelő kulcsszavak: UNION az egyesítésnek, INTERSECT a metszetnek és EXCEPT a különbségnek.

példa: Legyenek a Szállítók és Vevők relációk a NagyKer adatbázisból és a következő lekérdezés: „Keressük a kolozsvári cégeket, akikkel kapcsolatban áll a cégünk.” A megoldást a következő lekérdezés adja:

```
(SELECT Név, UtcaSzám
FROM Szállítók
WHERE Helység = 'Kolozsvár')
UNION
(SELECT Név, UtcaSzám
FROM Vevők
WHERE Helység = 'Kolozsvár');
```

példa: Legyenek az Alkalmazottak és Managerek relációk a NagyKer adatbázisból és a „Keressük azon alkalmazottakat, akik nem managerek” lekérdezés:

```
(SELECT SzemSzám, Név FROM Alkalmazottak)
EXCEPT
(SELECT SzemSzám, Név FROM Managerek, Alkalmazottak
WHERE Managerek.SzemSzám = Alkalmazottak.SzemSzám);
```

A fenti parancs esetén a második SELECT parancsban a join műveletre azért volt szükségünk, hogy a managernek keressük meg a nevét is, mert a különbség művelet esetén fontos, hogy az operandus relációknak ugyanaz az attribútumhalmaza legyen.

Ha az alkalmazott névre nem vagyunk kíváncsiak, akkor a következő SQL parancs azon alkalmazottak személyi számát adja meg, akik nem managerek.

```
(SELECT SzemSzám FROM Alkalmazottak)
EXCEPT
(SELECT SzemSzám FROM Managerek);
```

A feladatot oly módon is megoldhatjuk, ha a kereskedelmi rendszer nem támogatja az EXCEPT műveletet, hogy alkalmazzuk a NOT EXISTS vagy NOT IN záradékot.

példa: Legyen az Egyetem adatbázisa, és tegyük fel, hogy van olyan eset, hogy egy fiatal tanársegéd a matematika szakról, tehát elvégezte a matematika szakot, de még diák az informatika szakon. Legyen a következő lekérdezés: „keressük azon tanárokat, akik még diákok”. A megoldás:

```
(SELECT Név FROM Tanárok)
INTERSECT
(SELECT Név FROM Diákok);
```

A feladatot a következőképpen is megoldhatjuk, ha a kereskedelmi rendszer nem támogatja az INTERSECT műveletet:

```
SELECT Név FROM Tanárok
WHERE EXISTS
  (SELECT Név FROM Diákok
   WHERE Diákok.SzemSzáma = Tanárok.SzemSzáma); □
```

IV.4.3. Ismétlődő sorok

Az SQL nyelv relációi az absztrakt módon definiált relációktól abban különböznek, hogy az SQL nem tekinti őket halmaznak, azaz a relációk multihalmazok. A SELECT parancs eredményében szerepelhet két vagy több teljesen azonos sor, viszont van lehetőség ezen ismétlődések megszüntetésére.

A SELECT kulcsszó után a DISTINCT szó segítségével kérhetjük az azonos sorok megszüntetését.

példa: Az Egyetem adatbázisa esetén keressük azon csoportokat, amelyekben vannak olyan diákok, akiknek átlaga kisebb, mint 7.

```
SELECT DISTINCT CsoportKód
FROM Diákok
WHERE Átlag < 7;
```

A parancs a Diákok táblából kiválogatja azokat a sorokat, ahol az átlag kisebb, mint 7, ezen sorok diákokról szóló információkat tartalmaznak, többek között a csoportkódot is. Egy csoportban több diák is lehet, akiknek az átlaga kisebb, mint 7, ezért, ha nem használjuk a DISTINCT kulcsszót, akkor előfordulhat, hogy egy csoportkód többször is szerepel az eredményben. □

A SELECT parancssal ellentétben, a UNION, EXCEPT és INTERSECT halmazelméleti műveletek megszüntetik az ismétlődéseket. Ha nem szeretnénk, hogy az ismétlődő sorok eltűnjenek, a műveletet kifejező kulcsszó után az ALL kulcsszót kell használnunk.

példa: Az Egyetem adatbázisból keressük a személyeket, akik lehetnek tanárok vagy diákok. A következő parancs nem szünteti meg az ismétlődéseket:

```
(SELECT Név FROM Tanárok)
UNION ALL
(SELECT Név FROM Diákok);
```

Tehát, ha van olyan tanár, aki közben diák is, akkor az kétszer fog szerepelni az eredményben. □

IV.4.4. Összesítő függvények és csoportosítás

Az SQL nyelv lehetőséget ad egy oszlopban szereplő értékek összegezésére, vagyis hogy meghatározzuk a legkisebb, legnagyobb vagy átlag értéket egy adott oszlopból. Az *összesítés* művelete egy oszlop értékeiből egy új értéket hoz létre. Ezenkívül a reláció egyes sorait bizonyos feltétel szerint csoportosíthatjuk, például egy oszlop értéke szerint, és a csoporton belül végezhetünk összesítéseket.

Összesítő függvények a következők:

- SUM, megadja az oszlop értékeinek az összegét;
- AVG, megadja az oszlop értékeinek a átlagértékét;
- MIN, megadja az oszlop értékeinek a minimumát;

- MAX, megadja az oszlop értékeinek a maximumát;
- COUNT, megadja az oszlopban szereplő értékek számát, beleértve az ismétlődéseket is, ha azok nincsenek megszüntetve a DISTINCT kulcsszóval.

Ezeket a függvényeket egy skalár értékre alkalmazhatjuk, általában egy SELECT záradékbeli oszlopra.

példa: A következő lekérdezés segítségével megkapjuk az alkalmazottak átlagos fizetését:

```
SELECT AVG(Fizetés)
FROM Alkalmazottak; □
```

példa: Ha az alkalmazottak számára vagyunk kíváncsiak:

```
SELECT COUNT(*)
FROM Alkalmazottak; □
```

Mindkét példa esetén biztosak vagyunk abban, hogy egy alkalmazott csak egyszer szerepel a relációban, mivel a személyi szám elsődleges kulcs.

példa: Az Egyetem adatbázis esetén keressük azon csoportoknak a számát, amelyekben vannak olyan diákok, akik átlaga kisebb, mint 7:

```
SELECT COUNT(DISTINCT CsopKod)
FROM Diákok
WHERE Átlag < 7;□
```

Az eddigi összesítések az egész relációra vonatkoztak. Sok esetben a reláció sorait csoportosítani szeretnénk egy vagy több oszlop értékei szerint. Például az alkalmazottak átlagfizetését minden részlegen belül akarjuk meghatározni. Az Egyetem adatbázisban minden csoport esetén keressük a legnagyobb átlagot, a diákok számát. A csoportosítást a GROUP BY kulcsszó segítségével érjük el. A parancs általános formája:

```
SELECT < csoportosító oszlopok listája >,
       <összesítő-függvény>(<oszlop>)
FROM <reláció>
[WHERE <feltétel>]
[GROUP BY <csoportosító oszlopok listája>]
[HAVING <csoportosítási-feltétel>]
[ORDER BY <oszlop>];
```

A GROUP BY után megadjuk a csoportosító attribútumok (oszlopok) listáját, melyek azonos értéke szerint történik a csoportosítás. Csak ezeket az oszlopokat válogathatjuk ki a SELECT kulcsszó után és azokat, melyekre valamilyen összesítő függvényt alkalmazunk. Azon oszlopoknak, melyekre összesítő függvényt alkalmaztunk, érdemes más nevet adni, hogy könnyebben tudjunk hivatkozni rá.

példa: Legyenek az Alkalmazottak reláció sorai:

<i>SzemSzám</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés (euró)</i>
111111	Nagy Éva	2	300
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
123444	Vincze Ildikó	1	800
333333	Kovács István	2	500

A részlegeken belüli átlagfizetést a következő parancs segítségével kapjuk meg:

```
SELECT RészlegID, AVG(Fizetés), MIN(Fizetés), MAX(Fizetés), SUM(Fizetés)
FROM Alkalmazottak
GROUP BY RészlegID;
```

A kapott eredmény:

<i>RészlegID</i>	<i>AVG(Fizetés)</i>	<i>MIN(Fizetés)</i>	<i>MAX(Fizetés)</i>	<i>SUM(Fizetés)</i>
1	800	800	800	800
2	500	300	700	1500
9	650	400	900	1300

A lekérdezés processzor először rendezi a reláció sorait a csoportosítandó oszlop értékei szerint, utána azokat a sorokat, ahol ezen oszlopoknak ugyanaz az értéke, az eredmény relációban csak egy sor fogja képviselni, ahol megadhatjuk az oszlop értékét, amely a lekérdezett relációban minden sorban ugyanaz. A többi oszlopra csakis összesítéseket végezhetünk. Ha a SELECT kulcsszó után olyan oszlopot választunk ki, melynek értékei különbözőek a lekérdezett relációban, a lekérdezés processzor nem tudja, hogy a különböző értékekből melyiket válassza az eredménybe. Van olyan implementálása a SELECT–SQL parancsnak, mely megengedi, hogy egy olyan oszlopot is kiválasszunk, mely nincs a csoportosító attribútumok között és a processzor vagy az első, vagy az utolsó értéket választja a különböző értékek közül.

A SELECT parancs megengedi viszont, hogy a csoportosító attribútum hiányozzon a vetített attribútumok listájából.

példa: A következő lekérdezés helyes:

```
SELECT AVG(Fizetés) AS ÁtlagFizetés
FROM Alkalmazottak
GROUP BY RészlegID;
```

eredménye pedig:

<i>ÁtlagFizetés</i>
800
500
650

példa: Legyen a Szállít (SzállID, ÁruID, Ár) reláció. Egy árut több szállító is ajánlhatja, különböző árban. Sok esetben szükségünk van az átlagára, amiben ajánlanak egy árut.

A következő lekérdezés minden áru esetén meghatározza az átlagárát, amiben a különböző szállítók ajánlják.

```
SELECT ÁruID, AVG(Ár) AS ÁtlagÁr
FROM Szállít
GROUP BY ÁruID; □
```

A GROUP BY záradékot használhatjuk többrelációs lekérdezésben is. A lekérdezés processzor először az operandus relációkkal a WHERE feltételt figyelembe véve elvégzi a join, esetleg a Descartes szorzat műveletet és ennek az eredmény relációjára alkalmazza a csoportosítást.

példa: Ha a fenti példa esetén kíváncsiak vagyunk az árunak a nevére:

```
SELECT Áruk.Név, AVG(Ár)
FROM Szállít, Áruk
WHERE Szállít.ÁruID = Áruk.ÁruID
GROUP BY Áruk.Név;
```

Remélhetőleg az áru neve is egyedi kulcs, tehát nem fordul elő egy áru név több ÁruID esetén is, mert a fenti példában a Név attribútum szerint csoportosítunk. Ha nem egyedi a név, akkor a fenti lekérdezés az összes azonos nevű árunak az átlagát adja meg, de sok esetben ez megfelel a felhasználónak. Megoldhatjuk úgy is, hogy először ÁruID szerint, majd áru név szerint csoportosítunk, lásd a csoportosítást több oszlopra. □

Amint a SELECT parancsnak az általános formájánál láttuk, lehetséges több csoportosítási attribútum is.

példa: Legyenek a következő relációk az Egyetem adatbázisból:

```
Tanszékek (TanszékKod, Név, TanszékCsopKod);
Beosztások (BeosztásKod, Név);
```

Tanárok (TanárKod, Név, SzemSzám, Cím, PhD, BeosztásKod, TanszékKod, Fizetés);

és a következő lekérdezés: „Számítsuk ki a tanárok átlagfizetését tanszékeken belül, beosztásokra leosztva.”

```
SELECT TanszékKod, BeosztásKod, AVG(Fizetés)
FROM Tanárok
GROUP BY TanszékKod, BeosztásKod
```

Ha a Tanárok tábla tartalma:

<i>Tanár Kod</i>	<i>Név</i>	<i>Cím</i>	<i>PhD</i>	<i>Beosztás Kod</i>	<i>Tanszék Kod</i>	<i>Fizetés</i>
KB12	Kiss Béla	Petőfi u. 12	Y	ADJ	ALG	150
NL03	Nagy László	Kossuth u. 3	Y	ADJ	REN	160
KG05	Kovács Géza	Ady tér 5	N	ADJ	ALG	160
PI14	Péter István	Dóm tér 14	N	TNS	REN	120
NT55	Németh Tamás	Dózsa u. 55	Y	PRO	ALG	300
VS77	Vígh Sándor	Rózsa u. 77	Y	PRO	REN	310
LL63	Lukács Lóránt	Viola u. 63	Y	ADJ	REN	170
LS07	László Samu	Rákóczi u. 7	N	TNS	REN	110
KP52	Kerekes Péter	Váci u. 52	Y	PRO	ALG	280

a lekérdezés eredménye:

<i>Tanszék Kod</i>	<i>Beosztás Kod</i>	<i>AVG (Fizetés)</i>
ALG	ADJ	155
ALG	PRO	290
REN	ADJ	165
REN	PRO	310
REN	TNS	115

példa: Megismételve egy előbbi példát:

```
SELECT Áruk. ÁruID, Áruk.Név, AVG(Ár)
FROM Szállít. ÁruID = Áruk.ÁruID
WHERE Szállít, Áruk
GROUP BY Áruk.ÁruID, Áruk.Név;
```

Az áru név szerinti csoportosítás nem fog újabb csoportokat behozni, de nem válogathatjuk ki a Név oszlopot, ha nem szerepelt a csoportosítási attribútumok között. A vetítés attribútumai között nem kell feltétlenül szerepeljen az ÁruID, de ha egy név többször is előfordul, akkor az eredmény furcsa lesz.

A csoportosítás után kapott eredmény reláció soraira a HAVING kulcsszót használva egy feltételt alkalmazhatunk. Ha csoportosítás előtt szeretnénk kiszűrni sorokat, azokra a WHERE feltételt lehet alkalmazni. A HAVING kulcsszó utáni feltételben azon oszlopok szerepelhetnek, melyekre a SELECT parancsban összesítő függvényt alkalmaztunk.

példa: Keressük azon részlegeket, ahol az alkalmazottak átlagfizetése nagyobb, mint 500 euró, átlagfizetés szerint növekvő sorrendben.

```
SELECT RészlegID, AVG(Fizetés)
FROM Alkalmazottak
GROUP BY RészlegID
HAVING AVG(Fizetés) > 500
ORDER BY AVG(Fizetés);
```

A fenti adatokat figyelembe véve az eredmény reláció a következő lesz:

<i>RészlegID</i>	<i>AVG(Fizetés)</i>
9	650

1	800
---	-----

Ha nem adjuk meg az ORDER BY záradékot, akkor a GROUP BY záradékban megadott oszlopok szerint rendezi az eredményt.

példa: *Helytelen a következő parancs:*

```
SELECT RészlegID, AVG(Fizetés)
FROM Alkalmazottak
WHERE AVG(Fizetés) > 500
GROUP BY RészlegID;
```

példa: Keressük azon tanszékeket, ahol a tanársegédek kivéve a tanárok átlagfizetése nagyobb, mint 240 euró.

```
SELECT TanszékKod, AVG(Fizetés)
FROM Tanárok
WHERE BeosztásKod <> 'TNS'
GROUP BY TanszékKod
HAVING AVG(Fizetés) > 240;
```

Alkalmazhatunk összesítő függvényt a csoportosítás után.

példa: Keressük a tanárok tanszékenkénti átlagfizetéseiből a legnagyobb értéket:

```
SELECT MAX(AVG(Fizetés))
FROM Tanárok
GROUP BY TanszékKod;
```

A lekérdezés processzor először elvégzi a csoportosítást TanszékKod szerint, majd az átlagfizetésekből kiválasztja a legnagyobbat.

IV.4.5. Alkérdeések

A WHERE záradékban eddig a feltételben skaláris értékeket tudtunk összehasonlítani. Az alkérdeések segítségével sorokat vagy relációkat tudunk összehasonlítani. Egy alkérdeés egy olyan kifejezés, mely egy relációt eredményez, például egy select-from-where kifejezés.

Alkérdeést tartalmazó SELECT SQL parancs általános formája a következő:

```
SELECT <attribútum_lista>
FROM <tábla>
WHERE <kifejezés> <operátor>
      (SELECT <attribútum_lista>
       FROM <tábla>);
```

A rendszer először az alkérdeést hajtja végre és annak eredményét használja a „fő” lekérdezés, kivéve a korrelált alkérdeéseket.

Alkérdeéseket annak megfelelően csoportosíthatjuk, hogy az eredménye hány sort és hány oszlopot tartalmaz:

- egy oszlopot, egy sort, vagyis egy *skalár* értéket ad vissza (single-row);
- egy oszlopot, több sort, ún. *több soros alkérdeés* (multiple-row subquery);
- több oszlopot, több sort, ún. *több oszlopos alkérdeés* (multiple-column);

Ha egy attribútum egyetlen értékére van szükségünk, ebben az esetben a select-from-where kifejezés *skalár* értéket ad vissza, mely konstansként használható. A select-from-where kifejezés eredményeként kapott konstans egy attribútummal vagy egy másik konstanssal összehasonlíthatjuk. Nagyon fontos, hogy az alkérdeés select-from-where kifejezése csak egy attribútumnak egyetlen értékét adja eredményül, különben hibajelzést kapunk.

példa: Legyenek a Részlegek és Alkalmazottak relációk a NagyKer adatbázisból, és a következő lekérdezés: „Keressük a 'Tervezés' nevű részleg managerének a nevét.” A megoldás alkérdés segítségével:

```
1) SELECT Név
2) FROM Alkalmazottak
3) WHERE SzemSzáma =
4)     (SELECT ManSzemSzáma
5)     FROM Részlegek
6)     WHERE Név = 'Tervezés');
```

Amint látjuk, az alkérdés (4–6 sorok) csak egy oszlopot választ ki a manager személyi számát, de még abban is biztosak kell legyünk, hogy csak egy 'Tervezés' nevű részleg legyen az adatbázisban. Ezt elérhetjük ha egyedi kulcs megszorítást kérünk a Részlegek relációra a CREATE TABLE parancsban a UNIQUE kulcsszó segítségével. Abban az esetben, ha az alkérdés nulla vagy egynél több sort eredményez, a lekérdezés futás közbeni hibát fog jelezni. Az „Összesítések” alfejezet 6.22. példájának az adatait figyelembe véve az alkérdés eredményül az 123444 személyi számot adja, és a lekérdezés a következőképpen hajtódik végre:

```
SELECT Név
FROM Alkalmazottak
WHERE SzemSzáma = 123444
```

A lekérdezés eredménye: 'Vincze Ildikó' lesz.□

A skalár értéket adó alkérdéssel használható operátorok az: =, <, <=, >, >=, <>.

példa: „Keressük azon alkalmazottakat, kiknek fizetése nagyobb, mint annak az alkalmazottnak, kinek a személyi száma 333333.”

```
SELECT Név
FROM Alkalmazottak
WHERE Fizetés >
      (SELECT Fizetés
       FROM Alkalmazottak
       WHERE SzemSzáma = 333333); □
```

példa: „Keressük azon alkalmazottakat, kiknek a fizetése az összes alkalmazott minimális fizetésével egyenlő.”

```
SELECT Név
FROM Alkalmazottak
WHERE Fizetés =
      (SELECT MIN(Fizetés)
       FROM Alkalmazottak); □
```

példa: „Keressük azon részlegeket és az alkalmazottak minimális fizetését a részlegből, ahol a minimális fizetés nagyobb, mint a minimális fizetés a 2-es ID-jű részlegből.”

```
SELECT RészlegID, MIN(Fizetés)
FROM Alkalmazottak
GROUP BY RészlegID
HAVING MIN(Fizetés) >
      (SELECT MIN(Fizetés)
       FROM Alkalmazottak
       WHERE RészlegID = 2);
```

A lekérdezés processzor először az alkérdést értékeli ki, ennek eredményeként egy skalár értéket (300) kapunk és a fő lekérdezés ezzel a skalár értékkel fog dolgozni. □

Csínján kell bányunk a csoportosítással.

Példa: Egy helytelen SELECT parancs:

```
SELECT SzemSzáma, Név
FROM Alkalmazottak
WHERE Fizetés =
      (SELECT MIN(Fizetés)
       FROM Alkalmazottak
       GROUP BY RészlegID);
```

Az alkérdés több sort is visszaad, pontosan annyit, ahány különböző RészlegID létezik az Alkalmazottak táblában, minden részleg esetén a minimális fizetést adja vissza. Az egyenlőség az alkérdés előtt csak egy skaláris értéket vár. □

A több-soros alkérdések esetén a WHERE záradék feltétele olyan operátorokat tartalmazhat, amelyeket egy R relációra alkalmazhatunk, ebben az esetben az eredmény logikai érték lesz. Bizonyos operátoroknak egy skaláris s értékre is szükségük van. Ilyen operátorok:

- ▶ EXISTS R – feltétel, mely akkor és csak akkor igaz, ha R nem üres.

példa: SELECT Név

```
FROM Alkalmazottak, Managerek
WHERE Alkalmazottak.SzemSzáma = Managerek.SzemSzáma
AND EXISTS
      (SELECT *
       FROM Alkalmazottak
       WHERE Fizetés > 500);
```

A fenti példa csak abban az esetben adja meg a managerek nevét, ha van legalább egy alkalmazott, kinek a fizetése nagyobb, mint 500 euró.

- ▶ s IN R , mely akkor igaz, ha s egyenlő valamelyik R -beli értékkel. Az s NOT IN R akkor igaz, ha s egyetlen R -beli értékkel sem egyenlő.

példa: Legyen a NagyKer adatbázis és a következő lekérdezés: „Adjuk meg azon szállítók nevét és címét, akik valamilyen csokit szállítanak” (Áruk.Név LIKE '%csoki%')

```
1) SELECT Név, Helység, UtcaSzáma
2) FROM Szállítók
3) WHERE SzállID IN
4)      (SELECT SzállID
5)      FROM Szállít
6)      WHERE ÁruID IN
7)      (SELECT ÁruID
8)      FROM Áruk
9)      WHERE Név LIKE '%csoki%')
);
```

A 7–9 sor alkérdése az összes olyan árut választja ki, melynek nevében szerepel a csoki. Legyen a csoki áruk azonosítóinak a halmaza: CsokiID. A 4–6 sor a Szállít táblából azon SzállID-kat választja ki, ahol az ÁruID benne van a CsokiID halmazban. Nevezzük a csokit szállítók azonosítóinak a halmazát CsokiSzállIDk-nak. Az 1–3 sorok segítségével megkaphatjuk a csokit szállítók nevét és címét. □

A kereskedelmi rendszerek különböző mélységig tudják az alkérdéseket kezelni. Van olyan, amelyik csak 1 alkérdést engedélyez.

- ▶ s > ALL R , mely akkor igaz, ha s nagyobb, mint az R reláció minden értéke, ahol az R relációnak csak egy oszlopa van. A > operátor helyett bármelyik összehasonlítási operátort használhatjuk. Az s <> ALL R eredménye ugyanaz, mint az s NOT IN R feltételé.

példa: Legyen a következő lekérdezés:

```
SELECT SzemSzám, Név
FROM Alkalmazottak
WHERE Fizetés > ALL
      (SELECT MIN(Fizetés)
       FROM Alkalmazottak
       GROUP BY RészlegID);
```

Ugyanezt a lekérdezést láttuk egyenlőséggel az alkérdés előtt, helytelen példaként. Mivel az alkérdés több sort is visszaad, a „> ALL” operátort alkalmazva, a Fizetés oszlop értékét összehasonlítja az összes minimális fizetés értékkel az alkérdésből. Tehát a lekérdezés megadja azon alkalmazottakat, kiknek fizetése nagyobb, mint a minimális fizetés minden részlegből. □

- ▶ $s > \text{ANY } R$, mely akkor igaz, ha s nagyobb az R egyoszlopos reláció legalább egy értékénél. A $>$ operátor helyett akármelyik összehasonlítási operátort használhatjuk.

példa: „Keressük azokat a tanárokat, akik beosztása nem professzor, és van olyan professzor, akinek a fizetésénél az illető tanárnak nagyobb a fizetése.”

```
SELECT Név, BeosztásKod, Fizetés
FROM Tanárok
WHERE Fizetés > ANY
      (SELECT Fizetés
       FROM Tanárok
       WHERE BeosztásKod = 'PRO')
AND BeosztásKod <> 'PRO'; □
```

A *több oszlopos alkérdés* esetén, a SELECT kulcsszó után megadhatunk több mint egy oszlopot, és szükségszerűen a fő lekérdezésben is ugyanannyi oszlopot kell megadjunk az összehasonlító operátor bal oldalán is. Az összehasonlítás párokra vonatkozik.

példa: „Keressük azokat a tanárokat, akiknek a fizetése egyenlő az algebra tanszék beosztásnak megfelelő átlag fizetésével.”

```
SELECT Név, BeosztásKod, Fizetés
FROM Tanárok
WHERE BeosztásKod, Fizetés IN
      (SELECT BeosztásKod, AVG(Fizetés)
       FROM Tanárok
       WHERE TanszékKod = 'ALG'
       GROUP BY BeosztásKod); □
```

Az alkérdés meghatározza az algebra tanszéken belül a beosztásoknak megfelelő átlagfizetéseket. A fő lekérdezés akkor fog egy tanárt kiválasztani, ha az alkérdés eredményhalmazában megtalálja a tanár beosztás kódja mellett a fizetést is, az értékpárt.

IV.4.6. Korrelált alkérdések

Az eddig bemutatott alkérdések esetén az alkérdés csak egyszer kerül kiértékelésre és a kapott eredményt a magasabb rendű lekérdezés hasznosítja. A beágyazott alkérdéseket úgy is lehet használni, hogy az alkérdés többször is kiértékelésre kerül. Az alkérdés többszöri kiértékelését egy, az alkérdésen kívüli sorváltóval érjük el. Az ilyen típusú alkérdést *korrelált* alkérdésnek nevezzük.

példa: Az Egyetem adatbázis esetén keressük azon diákokat, akik egyedül vannak a csoportjukban 10-es átlaggal.

```
SELECT Név, CsopKod
FROM Diákok D1
WHERE Átlag = 10 AND NOT EXISTS
    (SELECT D2.BeiktatásiSzám
     FROM Diákok D2
     WHERE D1.CsopKod = D2.CsopKod
      AND D1.BeiktatásiSzám <> D2.BeiktatásiSzám
      AND D2.Átlag = 10);
```

A lekérdezés kiértékelése során a D1 sorváltozó végigjárja a Diákok relációt. Minden sorra a D1-ből a D2 sorváltozó segítségével ismét végigjárjuk a Diákok relációt.

Legyen $d1$ egy sor a Diákok relációból, amelyet a fő lekérdezés az eredménybe helyez, ha megfelel a WHERE utáni feltételnek. Először is a $d1$.Átlag értéke 10 kell legyen és az alkérdés eredménye pedig üres halmaz. Az alkérdés akkor fog sorokat tartalmazni, ha létezik a Diákok relációban egy $d2$ sor, mely esetén ugyanaz a csoport kód, mint a $d1$ sor esetén, az átlag értéke 10 és a beiktatási szám különbözik a $d1$ sor BeiktatásiSzám attribútum értékétől. Ez azt jelenti, hogy az adatbázisban találtunk egy másik diákot, ugyanabból a csoportból, akinek az átlaga 10-es. Mivel az alkérdésben vannak sorok, nem fogja a $d1$ sort kiválasztani. Ha az alkérdés üres halmaz, akkor kiválasztja a $d1$ -et, és ekkor találtunk olyan diákokat, akik egyedül vannak a csoportjában 10-es átlaggal. $\square$

IV.4.7. Más típusú összekapcsolási műveletek

A relációs algebra természetes összekapcsolás műveletét eddig a SELECT parancs segítségével láttuk implementálva. Ha a WHERE záradékban adjuk meg a feltételt, vagy INNER JOIN kulcsszót használunk, csak azok a sorok kerülnek be az eredmény relációba, melyek esetében a közös attribútum ugyanaz az értéke mindkét relációban megtalálható. (A lógó sorok nem kerülnek be az eredménybe.) Bizonyos esetekben szükségünk van a lógó sorokra is.

Az OUTER JOIN kulcsszó segítségével azon sorok is megjelennek az eredményben, melyek értéke a közös attribútumra nem található meg a másik táblában, vagyis a lógó sorok, melyekben a másik tábla attribútumai NULL értékeket kapnak. Tehát a *külső összekapcsolás* (outer join) eredménye tartalmazza a belső összekapcsolás (*inner join*) eredménye mellett a lógó sorokat is. A *külső összekapcsolás* 3-féle lehet:

```
R LEFT OUTER JOIN S ON R.X = S.X
```

eredménye tartalmazza a bal oldali R reláció összes sorát, azokat is, amelyek esetében az X attribútumhalmaz értéke nem létezik az S reláció X értékei között. Ezt a műveletet külső baloldali összekapcsolásnak nevezzük. Az eredmény az S attribútumait is tartalmazza NULL értékekkel.

```
R RIGHT OUTER JOIN S ON R.X = S.X
```

eredménye a jobb oldali S reláció összes sorát tartalmazza, azokat is amelyek esetében az X attribútumhalmaz értéke nem létezik az R reláció X értékei között. Ezt a műveletet külső jobboldali összekapcsolásnak nevezzük. Az eredmény az R attribútumait is tartalmazza NULL értékekkel.

```
R FULL OUTER JOIN S ON R.X = S.X
```

eredménye azon sorokat tartalmazza, melyek esetében a közös attribútum értéke megegyezik mindkét relációban és mind a bal oldali R reláció lógó sorait, mind az S reláció lógó sorait magában foglalja.

példa: Legyenek az Alkalmazottak és Részlegek reláció sorai:

<i>SzemSzá</i> m	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés</i>
111111	Nagy Éva	2	300
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
123444	Vincze Ildikó	1	800
567765	Katona József	NULL	600
556789	Lukács Anna	NULL	700
333333	Kovács István	2	500

<i>RészlegID</i>	<i>RNév</i>	<i>ManagerSze</i> mSzá
1	Tervezés	123444
2	Könyvelés	234555
3	Eladás	NULL
9	Beszerezés	456777

Legyen a következő lekérdezés:

```
SELECT * FROM Alkalmazottak
INNER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

Az eredmény:

<i>SzemSzá</i> m	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés</i>	<i>RNév</i>	<i>ManagerSze</i> mSzá
111111	Nagy Éva	2	300	Könyvelés	234555
222222	Kiss Csaba	9	400	Beszerezés	456777
456777	Szabó János	9	900	Beszerezés	456777
234555	Szilágyi Pál	2	700	Könyvelés	234555
123444	Vincze Ildikó	1	800	Tervezés	123444
333333	Kovács István	2	500	Könyvelés	234555

Tehát azon alkalmazottak esetén, ahol a RészlegID megtalálható a Részlegek táblában megkapjuk a megfelelő részleg nevét és a manager személyi számát. Lógó sorok nem jelennek meg az eredményben. □

példa: A
SELECT * FROM Alkalmazottak

```
LEFT OUTER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

eredménye:

<i>SzemSzá</i> m	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés</i>	<i>RNév</i>	<i>ManagerSze</i> mSzá
111111	Nagy Éva	2	300	Könyvelés	234555
222222	Kiss Csaba	9	400	Beszerezés	456777

				és	
456777	Szabó János	9	900	Beszerzés	456777
234555	Szilágyi Pál	2	700	Könyvelés	234555
123444	Vincze Ildikó	1	800	Tervezés	123444
567765	Katona József	NULL	600	NULL	NULL
556789	Lukács Anna	NULL	700	NULL	NULL
333333	Kovács István	2	500	Könyvelés	234555

Ebben az esetben az Alkalmazottak összes sora, és a lógó sorok is megjelennek az eredményben, a Részlegek attribútumai a lógó sorok esetén NULL értéket kapnak. □

példa: A

SELECT * FROM Alkalmazottak

RIGHT OUTER JOIN Részlegek

ON Alkalmazottak.RészlegID = Részlegek.RészlegID;

eredménye:

<i>SzemSzáma</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés</i>	<i>RNév</i>	<i>ManagerSzemSzáma</i>
111111	Nagy Éva	2	300	Könyvelés	234555
222222	Kiss Csaba	9	400	Beszerzés	456777
456777	Szabó János	9	900	Beszerzés	456777
234555	Szilágyi Pál	2	700	Könyvelés	234555
123444	Vincze Ildikó	1	800	Tervezés	123444
333333	Kovács István	2	500	Könyvelés	234555
NULL	NULL	3	NULL	Eladás	NULL

Ebben az esetben a Részlegek összes sora jelenik meg, mivel ez a jobb oldali reláció. Az Alkalmazottak reláció attribútumai a lógó részleg esetén NULL értékeket kapnak. □

példa: A

SELECT * FROM Alkalmazottak

FULL OUTER JOIN Részlegek

ON Alkalmazottak.RészlegID = Részlegek.RészlegID;

eredménye:

<i>SzemSzáma</i>	<i>Név</i>	<i>RészlegID</i>	<i>Fizetés</i>	<i>RNév</i>	<i>ManagerSzemSzáma</i>
111111	Nagy Éva	2	300	Könyvelés	234555
222222	Kiss Csaba	9	400	Beszerzés	456777
456777	Szabó János	9	900	Beszerzés	456777

234555	Szilágyi Pál	2	700	Könyvel és	234555
123444	Vincze Ildikó	1	800	Tervezés	123444
333333	Kovács István	2	500	Könyvel és	234555
567765	Katona József	NULL	600	NULL	NULL
556789	Lukács Anna	NULL	700	NULL	NULL
NULL	NULL	3	NULL	Eladás	NULL

Példa feladatok

1. i) Tervezzünk **relációs adatbázis sémát**, melynek táblái **3NF**-ban vannak és egy software cég következő információit tárolják:

- **tevékenységek:** tevékenység kód, leírás, tevékenység típus;
- **alkalmazottak:** alkalmazott kód, név, tevékenységek listája, csoport, melynek tagja, csoport vezetője.

Egy tevékenységet a kódja azonosít, egy alkalmazottat szintén. Egy alkalmazott egy csoportnak tagja, egy csoportnak egy vezetője van, aki szintén a cég alkalmazottja. Egy alkalmazott több tevékenységben is részt vehet, illetve egy tevékenységnél több alkalmazott is dolgozhat.

Indokoljuk, hogy a táblák **3NF**-ban vannak. Irjuk fel a funkcionális függőségeket.

ii) Relációs algebrát vagy SELECT-SQL parancsot használva, az **i)** pont adatbázisára vonatkozóan adjuk meg:

- azokat az alkalmazottakat a nevükkel, akik dolgoznak legalább egy “tervezés” típusú tevékenységnél és **nem** dolgoznak egyetlen “tesztelés” típusú tevékenységnél sem.
- azokat az alkalmazottakat a nevükkel, akik olyan csoportok vezetői, amelyekhez legalább 10 alkalmazott tartozik.

2. i) Tervezzünk **relációs adatbázis sémát**, melynek táblái **3NF**-ban vannak és a következő információikat tárolják:

- **tantárgyak:** tantárgykód, megnevezés, kreditek száma;
- **diákok:** diák kód, név, születési dátum, csoportkód, évfolyam, szak, azon tantárgyak listája, amiből vizsgázott (a vizsga dátuma és a jegy is tárolandó).

Indokoljuk, hogy a táblák **3NF**-ban vannak. Irjuk fel a funkcionális függőségeket.

ii) Relációs algebrát vagy SELECT-SQL parancsot használva, az **i)** pont adatbázisára vonatkozóan adjuk meg:

- azokat a tantárgyakat a megnevezésükkel, amelyek esetén nincsenek átmenő jegyek (átmenő jegy ≥ 5)
- azokat a diákokat (név, csoport, sikeres vizsgák száma), akik több, mint 5 vizsgán átmenő jegyet kaptak. Ha egy diáknak több jegye is van egy tárgyból, csak egyszer számoljuk.

3. i) Tervezzünk relációs adatbázis sémát, melynek táblái **3NF**-ban vannak és államvizsgára iratkozott diákokról a következő adatokat tárolják: beiktatási szám, diák neve, elvégzett szak kódja és neve, szakdolgozat címe, irányító tanár kódja és neve, azon intézet kódja és megnevezése, amelyhez az irányító tanár tartozik, a dolgozat megvédéséhez szükséges software-k listája (pl.: VB.Net, MS SQL Server, Oracle, C#, Delphi, C++, IE, stb.), illetve hardver szükséglet listája (pl.: 1Gb RAM, 512Mb RAM, DVD Reader, stb.). Irjuk fel a funkcionális függőségeket, indokoljuk, hogy a táblák **3NF**-ban vannak.

ii) Relációs algebrát vagy SELECT-SQL parancsot használva, legalább egyszer mindegyiket az **i)** pont adatbázisára vonatkozóan adjuk meg:

- a) Azon diákokat (Név, Szakdolgozat címe, Vezető tanár neve), akiknek államvizsga vezető tanára egy adott intézethez tartozik.
- b) Egy adott intézet esetén a diákok számát, akik vezető tanára az adott intézethez tartozik.
- c) Azon tanárokat (név, tanszéke neve), akik nem vezettek államvizsgát.
- d) Azon diákok nevét, akik Oracle-t is és C#-ot is igényeltek.

5. i) Tervezzünk **relációs adatbázis sémát**, melynek táblái **3NF**-ban vannak és filmekről szóló információkat tárolnak:

- **színészek**: színész kód, név, nem, web oldal, ország;
- **filmek**: film kód, cím, megjelenési dátum, stúdió neve, stúdió weboldal, stúdió országa, rendező neve, rendező weboldala, rendező országa, színészek listája, film típus listája.

Irjuk fel a létező funkcionális függőségeket, indokoljuk, hogy a végső táblák 3NF-ban vannak.

ii). Relációs algebrát vagy SELECT-SQL parancsot használva, az i) pont adatbázisára vonatkozóan adjuk meg:

a. azokat az filmeket (cím, megjelenési dátum, stúdió neve), melyekben Julia Roberts és Richard Gere együtt szerepelnek.

b. azokat a színészeket (név, web oldal), akik a legtöbb filmben játszottak.