Babeş-Bolyai Tudományegyetem, Kolozsvár

Informatika záróvizsga tankönyv

2019

Záróvizsga témák

Informatika szak

(a 2019. júliusi vizsgaidőszaktól érvényes)

1. rész: Algoritmusok és programozás (6 téma)

1. Keresés (szekvenciális és bináris), összefésülés, rendezések (kiválasztásos, buborékrendezés, beszúrásos, összefésüléses, 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 az UML-ben. Osztályok közötti kapcsolatok.
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.
    a. Alapfogalmak: változók, vezérlési szerkezetek (if/then/elif/else/fi, for/done, while/do/done, shift, break, continue), előre definiált változók ($0, $1,..., $9, $*, $@, $?), bemenet/kimenet átirányításai (|, >, >>, <, 2>, 2>>, 2>&1, a /dev/null állomány, fordított aposztrófok ``)
    b. Reguláris kifejezések
    c. Alapvető parancsok (működés és a megadott paraméterek hatása): cat, chmod (-R), cp (-r), cut (-d,-f), echo, expr, file, find (-name,-type), grep (-i,-q,-v), head (-n), ls (-l), mkdir (-p), mv, ps (-e,-f), pwd, read (-p), rm (-f,-r), sed (csak a d,s,y parancsok), sleep, sort (-n,-r), tail (-n), test (numerikus operátorok karakterláncokra és állományokra), true, uniq (-c), wc (-c,-l,-w), who

Tartalomjegyzék

1. fejezet Algoritmusok és programozás

1.1. Programozási tételek

A feladatok *feladatosztályok*ba sorolhatók a jellegük szerint. E feladatosztályokhoz készítünk a teljes feladatosztályt megoldó *algoritmusosztály*t, amelyeket *programozási tétel*eknek nevezünk. Bebizonyítható, hogy ezek a megoldások a szóban forgó feladatok *garantáltan helyes és optimális* megoldásai.

A programozási tételek a feladat bemenete és kimenete szerint négy csoportra oszthatók:

- **A.** sorozathoz érték rendelése (1 sorozat – 1 érték)
- **B.** sorozathoz sorozat rendelése (1 sorozat – 1 sorozat)
- **C.** sorozatokhoz sorozat rendelése (több sorozat – 1 sorozat)
- **D.** sorozathoz sorozatok rendelése (1 sorozat – több sorozat)

A. Sorozathoz érték rendelése

1.1.1. Sorozatszámítás

Adott az N elemű X sorozat. A sorozathoz hozzá kell rendelnünk egyetlen S értéket. Ezt az értéket egy, az egész sorozaton értelmezett f függvény (pl. elemek összege, szorzata stb.) adja. Ezt a függvényt felbonthatjuk értékpárokon kiszámított függvények sorozatára, így a megoldás az F_0 semleges elemre, valamint egy kétoperandusú műveletre épül. Az S kezdőértéke a semleges elem. A kétoperandusú műveletet végrehajtjuk minden X_i elemre és az S értékre: $S \leftarrow f(X_i, S)$.

Összeg és szorzat

Egyetlen kimeneti adatot számítunk ki, adott számú bemeneti adat feldolgozásának eredményeként, például a bemeneti adatok összegét, esetleg 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 elem*nek. Feltételezzük, hogy a bemeneti adatok egész számok, amelyeknek a számossága N[1].

```
Algoritmus Összegszámítás(N, X, S):                    { Sajátos eset }
                                  { Bemeneti adatok: az N elemű X sorozat, kimeneti adat: S }
  S ← 0
  Minden i = 1, N végezd el:                  { minden adatot fel kell dolgoznunk }
    S ← S + Xi
  vége(minden)
```

[1] A következőkben az algoritmusok implementálása különböző típusú függvényekként szabadon választható. Ha a függvény egyetlen értéket számít ki, akkor ezt nem kötelező kimeneti paraméterként implementálni, hanem térítheti a függvény.

```
Vége(algoritmus)
```

Az előbbi algoritmus általánosítva:

```
Algoritmus Feldolgoz(N, X, S):
                                  { Bemeneti adatok: az N elemű X sorozat, kimeneti adat: S }
  S ← F₀                        { kezdőérték: az elvégzendő műveletre nézve semleges elem }
  Minden i = 1, N végezd el:                  { minden adatot fel kell dolgoznunk }
    S ← f(S, Xᵢ)                                              { f a művelet (funkció) }
  vége(minden)
Vége(algoritmus)
```

1.1.2. Döntés

Adott az *N* elemű *X* sorozat és az elemein értelmezett *T* tulajdonság. Döntsük el, hogy létezik-e a sorozatban legalább egy *T* tulajdonságú elem!

Elemzés

A sorozat elemei tetszőlegesek, egyetlen jellemzőt kell feltételeznünk róluk: bármely elemről el lehet dönteni, hogy rendelkezik-e az adott tulajdonsággal, vagy nem. A válasz egy üzenet, amelyet az alprogram kimeneti paramétere (logikai változó) értéke alapján ír ki a hívó programegység.

```
Algoritmus Döntés_1(N, X, talált):
                  { Bemeneti adatok: az N elemű X sorozat. Ha az X sorozatban található }
                    { legalább egy T tulajdonságú elem, talált értéke igaz, különben hamis }
  i ← 1                                                    { kezdőérték az indexnek }
  talált ← hamis                                  { kezdőérték a kimeneti adatnak }
  Amíg nem talált és (i ≤ N) végezd el:
    Ha nem T(Xᵢ) akkor                { amíg nem találunk egy Xᵢ-t, amely rendelkezik }
      i ← i + 1                                  { a T tulajdonsággal, haladunk előre }
    különben
      talált ← igaz
    vége(ha)
  vége(amíg)
Vége(algoritmus)
```

A fenti algoritmus megírható tömörebben is:

```
Algoritmus Döntés_2(N, X, talált):
                        { Bemeneti adatok: az N elemű X sorozat. Ha az X sorozatban }
            { található legalább egy T tulajdonságú elem, talált értéke igaz, különben hamis }
  i ← 1
  Amíg (i ≤ N) és nem T(Xᵢ) végezd el:          { amíg nem találunk egy Xᵢ-t, amely }
    i ← i + 1                          { rendelkezik a T tulajdonsággal, haladunk előre }
  vége(amíg)
  talált ← i ≤ N          { kiértékelődik a relációs kifejezés; az érték talált értéke lesz }
Vége(algoritmus)
```

Egy másik megközelítésben el kell döntenünk, hogy az adatok, teljességükben, rendelkeznek-e egy adott tulajdonsággal vagy sem. Más szóval: nem létezik egyetlen adat sem, amely ne lenne *T* tulajdonságú. Ekkor a bemeneti adathalmaz minden elemét meg kell vizsgálnunk. Mivel a döntés jelentése az összes adatra érvényes, a *talált* változót átkereszteljük *mind*-re.

```
Algoritmus Döntés_3(N, X, mind):
                { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: ha az X sorozatban }
                        { minden elem T tulajdonságú, a mind értéke igaz, különben hamis }
  i ← 1
  Amíg (i ≤ N) és T(Xi) végezd el:              { a nem T(Xi) részkifejezés tagadása }
    i ← i + 1
  vége(amíg)
  mind ← i > N                                    { az i ≤ N részkifejezés tagadása }
Vége(algoritmus)
```

1.1.3. Kiválasztás

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

```
Algoritmus Kiválasztás(N, X, hely):
                                        { Bemeneti adatok: az N elemű X sorozat. }
                    { Kimeneti adat: hely, a legkisebb indexű T tulajdonságú elem sorszáma }
  hely ← 1
  Amíg nem T(Xhely) végezd el:     { nem szükséges a hely ≤ N feltétel, mivel a feladat }
    hely ← hely + 1          { garantálja 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 létezik-e *T* tulajdonságú elem a sorozatban! Ha létezik, akkor adjuk meg az első ilyen elem helyét!

```
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 ← 0
  i ← 1
  Amíg (hely = 0) és (i ≤ N) végezd el:
    Ha T(Xi) akkor
      hely ← i
    különben
      i ← 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 ← 1
  Amíg (i ≤ N) és nem T(Xi) végezd el:
    i ← i + 1
  vége(amíg)
  Ha i ≤ N akkor          { ha kiléptünk az Amíg-ból, mielőtt i nagyobbá vált volna N-nél, }
    hely ← i                          { ⇒ találtunk adott tulajdonságú elemet az i. pozición }
  különben
    hely ← 0                                                  { különben nem találtunk }
  vége(ha)
Vége(algoritmus)
```

Ha a feladat azt kéri, hogy keressünk meg minden olyan elemet, 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 másik sorozatban. Ilyenkor *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 tároljuk.

```
Algoritmus Megszámlálás(N, X, db):
                                                { Bemeneti adatok: az N elemű X sorozat }
                                  { Kimeneti adat: db, a T tulajdonságú elemek darabszáma }
  db ← 0
  Minden i = 1, N végezd el:
    Ha T(Xi) akkor
      db ← 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!

Megoldás

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

```
Algoritmus Maximumkiválasztás(N, X, max):
  { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: max, a legnagyobb elem értéke }
  max ← X1
  Minden i = 2, n végezd el:
    Ha max < Xi akkor
      max ← Xi
    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, az algoritmus a következő:

```
Algoritmus MaximumHelye(N, X, hely):
{ Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: hely, a legnagyobb elem pozíciója }
  hely ← 1                                   { hely az első elem pozíciója }
  Minden i = 2, n végezd el:
    Ha Xhely < Xi akkor
      hely ← i          { a maximális elem első előfordulásának 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 algoritmust, amely csak egyszer járja be a sorozatot:

```
Algoritmus MindenMaximumHelye(N, X, db, indexek):
    { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű indexek sorozat }
  max ← X1
  db ← 1
  indexek1 ← 1
  Minden i = 2, n végezd el:
    Ha max < Xi akkor
      max ← Xi
      db ← 1
      indexekdb ← i
    különben
      Ha max = Xi akkor
        db ← db + 1
        indexekdb ← i
      vége(ha)
    vége(ha)
```

```
  vége(minden)
Vége(algoritmus)
```

B. Sorozathoz sorozat rendelése

1.1.7. Másolás

Adott az *N* elemű *X* sorozat és az elemein értelmezett *f* függvény. A bemeneti sorozat minden elemére végrehajtjuk a függvényt, az eredményét pedig a kimeneti 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 ← f(X_i)
  vége(minden)
Vége(algoritmus)
```

1.1.8. Kiválogatás

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

Elemzés

Az elvárások függvényében különböző megközelítések lesznek érvényesek:

- **a.** kiválogatás kigyűjtéssel
- **b.** kiválogatás kiírással
- **c.** kiválogatás helyben (sorrendváltoztatással vagy megőrizve az eredeti sorrendet)
- **d.** kiválogatás kihúzással (segédsorozattal vagy helyben)

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, 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
      pozíciók_db ← i                    { pozíciók_db-ben tároljuk az X_i helyét }
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

b. Kiválogatás kiírással

Ha a feladat „megelégszik” a *T* tulajdonságú elemek kiírásával (nem kéri ezek darabszámát is), az algoritmus a következő:

```
Algoritmus Kiválogatás_b(N, X):
                                   { Bemeneti adatok: az N elemű X sorozat }
  Minden i = 1, N végezd el:
    Ha T(Xi) akkor
      Ki: Xi
    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 sorrendjükben 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 alkalmazunk Minden-t, mivel változik az N!!! }
    Ha nem T(Xi) akkor                  { a T tulajdonságú elemeket tartjuk meg }
      Xi ← XN                           { Xi-t felülírjuk XN-nel }
      N ← N – 1                         { változik a sorozat hossza }
    különben
      i ← i + 1                         { i csak a különben ágon nő }
    vége(ha)
  vége(amíg)
Vége(algoritmus)
```

c2. Ha az eredeti sorozatra nincs többé szükség, de szeretnénk megőrizni az elemek eredeti sorrendjét, akkor a *T* tulajdonságú elemeket felsorakoztatjuk a sorozat elejétől kezdve. 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 a megváltoztatott sorozatnak a számosságát tartja nyilván:

```
Algoritmus Kiválogatás_c2(N, X, db):
          { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: a db elemű X sorozat }
  db ← 0
  Minden i = 1, N végezd el:
    Ha T(Xi) akkor
      db ← db + 1
```

```
        Xdb ← Xi
      vége(ha)
    vége(minden)
Vége(algoritmus)
```

d1. 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_d1(N, X, törölt):
        { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű törölt sorozat }
  Minden i = 1, N végezd el:
    töröltᵢ ← hamis
  vége(minden)
  Minden i = 1, N végezd el:
    Ha nem T(Xᵢ) akkor                    { a T tulajdonságú elemeket tartjuk meg }
      töröltᵢ ← igaz
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

d2. Egy másik megoldás, amely nem hoz létre új helyen, új sorozatot, 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_d2(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(Xᵢ) akkor                    { a T tulajdonságú elemeket tartjuk meg }
      Xᵢ ← speciális érték
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

C. Sorozatokhoz sorozat rendelése

1.1.9. Halmazok

Mielőtt egy *halmazokat* tartalmazó sorozatra vonatkozó műveletet alkalmaznánk, szükséges meggyőződnünk arról, 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?

```
Algoritmus Halmaz_e(N, X, ok):
                    { Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az ok értéke igaz, }
  i ← 1                                                  { ha a sorozat halmaz, különben hamis }
  ok ← igaz
  Amíg ok és (i < N) végezd el:
    j ← i + 1
    Amíg (j ≤ N) és (Xi ≠ Xj) végezd el:
      j ← j + 1
    vége(amíg)
    ok ← j > N                                  { ha véget ért a sorozat, nincs két azonos elem }
    i ← i + 1
  vége(amíg)
Vége(algoritmus)
```

b. Halmazzá alakítás

Alakítsuk halmazzá az *N* elemű *X* sorozatot!

Elemzés

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

```
Algoritmus HalmazzáAlakít(N, X):
                                                    { Bemeneti adatok: az N elemű X sorozat. }
                                                   {Kimeneti adatok: az új N elemű X sorozat (halmaz) }
  i ← 1
  Amíg i < N végezd el:
    j ← i + 1
    Amíg (j ≤ N) és (Xi ≠ Xj) végezd el:
      j ← j + 1
    vége(amíg)
    Ha j ≤ N akkor                                  { találtunk egy Xj = Xi-t }
      Xj ← XN                                  { felülírjuk a sorozat N. elemével }
      N ← N – 1                                        { rövidítjük a sorozatot }
    különben
      i ← i + 1                                                { haladunk tovább }
    vége(ha)
  vége(amíg)
Vége(algoritmus)
```

1.1.10. Keresztmetszet

Hozzuk létre a bemenetként kapott sorozatok keresztmetszetét!

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 rendezettek.

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 ← 0                { Kimeneti adatok: a db elemű Z sorozat, X és Y keresztmetszete }
  Minden i = 1, N végezd el:
    j ← 1
    Amíg (j ≤ M) és (Xi ≠ Yj) végezd el:
      j ← j + 1
    vége(amíg)
    Ha j ≤ M akkor
      db ← db + 1
      Zdb ← Xi
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

1.1.11. Egyesítés (Unió)

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

Elemzés

Az egyesítés algoritmusa hasonló a keresztmetszetéhez. 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(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 egyesítése) }
  Másolás(N, X, Z)              { az X sorozat minden elemét átmásoljuk a Z sorozatba }
  db ← N
  Minden j = 1, M végezd el:
    i ← 1
    Amíg (i ≤ N) és (Xi ≠ Yj) végezd el:
      i ← i + 1
    vége(amíg)
    Ha i > N akkor
      db ← db + 1
```

```
        Z_db ← Y_j
      vége(ha)
    vége(minden)
Vége(algoritmus)
```

1.1.12. Összefésülés

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

Elemzés

Az *Egyesítés(N, X, M, Y, db, Z)* és a *Keresztmetszet(N, X, M, Y, db, Z)* algoritmusok négyzetes bonyolultságúak, mivel a halmazokat implementáló sorozatok nem rendezettek. Ez a két művelet megvalósítható lineáris algoritmussal, ha a sorozatok rendezettek. Így 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 nem érünk. A másik sorozatban megmaradt elemeket átmásoljuk az eredménysorozatba. 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. A sorozatok nem halmazok. }
                                    { Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }
  db ← 0
  i ← 1
  j ← 1
  Amíg (i ≤ N) és (j ≤ M) végezd el:        { amíg sem X-nek, sem Y-nak nincs vége }
    db ← db + 1
    Ha X_i < Y_j akkor
      Z_db ← X_i
      i ← i + 1
    különben
      Z_db ← Y_j
      j ← j + 1
    vége(ha)
  vége(amíg)
  Amíg i ≤ N végezd el:                     { ha maradt még elem X-ben }
    db ← db + 1
    Z_db ← X_i
    i ← i + 1
  vége(amíg)
  Amíg j ≤ M végezd el:                     { ha maradt még elem Y-ban }
    db ← db + 1
    Z_db ← Y_j
```

```
    j ← 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 eredménysorozatba.

```
Algoritmus Összefésülés_2(N, X, M, Y, db, Z):
          { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. A sorozatok halmazok
}
                                  { Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }
  db ← 0
  i ← 1
  j ← 1
  Amíg (i ≤ N) és (j ≤ M) végezd el:
    db ← db + 1
    Ha Xi < Yj akkor
      Zdb ← Xi
      i ← i + 1
    különben
      Ha Xi = Yj akkor
        Zdb ← Xi
        i ← i + 1
        j ← j + 1
      különben
        Zdb ← Yj
        j ← j + 1
      vége(ha)
    vége(ha)
  vége(amíg)
  Amíg i ≤ N végezd el:                    { ha maradt még elem X-ben }
    db ← db + 1
    Zdb ← Xi
    i ← i + 1
  vége(amíg)
  Amíg j ≤ m végezd el:                    { ha maradt még elem Y-ban }
    db ← db + 1
    Zdb ← Yj
    j ← j + 1
  vége(amíg)
Vége(algoritmus)
```

Szerencsés esetben $X_N = Y_M$. Ekkor a két utolsó *Amíg* struktúra nem hajtódott volna végre egyetlen egyszer sem. Kihasználva 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 eredménysorozat hossza pontosan $N + M$. Az algoritmus ismétlőstruktúrája *Minden* típusú lesz.

```
Algoritmus Összefésül_3(N, X, M, Y, db, Z):
      { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. A sorozatok nem halmazok }
                                  { Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }
  i ← 1
  j ← 1
  X[N+1] ← Y[M] + 1
  Y[M+1] ← X[N] + 1
  Minden db = 1, N + M végezd el:
    Ha X[i] < Y[j] akkor
      Z[db] ← X[i]
      i ← i + 1
    különben
      Z[db] ← Y[j]
      j ← j + 1
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

Ha a bemeneti sorozatok halmazokat ábrázolnak és az eredménysorozatnak is halmaznak kell lennie, az algoritmus a következőképpen alakul: a *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 őrszemek révén az *Amíg* struktúrát addig hajtjuk végre, amíg mindkét sorozat végére nem értünk.

```
Algoritmus Összefésül_4(N, X, M, Y, db, Z):
            { Bemeneti adatok: az N elemű X és az M elemű Y sorozat. A sorozatok halmazok }
                                  { Kimeneti adatok: a db elemű Z sorozat (X és Y elemeivel) }
  db ← 0
  i ← 1
  j ← 1
  X[N+1] ← Y[M] + 1
  Y[M+1] ← X[N] + 1
  Amíg (i ≤ N) vagy (j ≤ M) végezd el:
    db ← db + 1
    Ha X[i] < Y[j] akkor
      Z[db] ← X[i]
      i ← i + 1
    különben
      Ha X[i] = Y[j] akkor
        Z[db] ← X[i]
```

```
        i ← i + 1
        j ← j + 1
      különben
        Z_db ← Y_j
        j ← j + 1
      vége(ha)
    vége(ha)
  vége(amíg)
Vége(algoritmus)
```

D. Sorozathoz sorozatok rendelése

1.1.13. Szétválogatás

Válogassuk szét az adott *N* elemű *X* sorozat elemeit 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 új sorozatba

Az adott sorozatból létrehozunk két újat: a *T* 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 ← 0                                  { Bemeneti adatok: az N elemű X sorozat. }
  dbz ← 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 ← dby + 1              { az adott tulajdonságú elemek, az Y sorozatba kerülnek }
      Y_dby ← X_i
    különben
      dbz ← dbz + 1                                { azok, amelyek nem rendelkeznek az }
      Z_dbz ← X_i                        { adott tulajdonsággal, a Z sorozatba kerülnek }
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

b. Szétválogatás egyetlen 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 fogunk *N*

helyre „átrendezni”. A megmaradt elemek az eredeti sorozatban elfoglalt relatív pozícióik fordított sorrendjében kerülnek az új sorozatba.

```
Algoritmus Szétválogatás_2(N, dby, dbz, X, Y):
  dby ← 0{ Bemeneti adatok: az N elemű X sorozat. Kimeneti adat: az N elemű Y sorozat; }
  dbz ← 0          { az első dby elem T tulajdonságú, dbz elem pedig nem T tulajdonságú }
  Minden i = 1, N végezd el:
    Ha T(Xi) akkor                  { a T tulajdonságú elemek az Y sorozatba kerülnek }
      dby ← dby + 1                                    { az első helytől kezdődően }
      Ydby ← Xi

    különben
      dbz ← dbz + 1      { a többi elem szintén Y-ba kerül, az utolsó helytől kezdődően }
      YN-dbz+1 ← Xi
    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 elölről keresünk egy nem *T* tulajdonságú elemet, s ha találunk, azt hátratesszük. 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; }
                    { az első e elem T tulajdonságú, n – e elem pedig nem T tulajdonságú }
  e ← 1                       { balról jobbra haladva az első T tulajdonságú elem indexe }
  u ← N                  { jobbról balra haladva az első nem T tulajdonságú elem indexe }
  segéd ← Xe
  Amíg e < u végezd el:
    Amíg (e < u) és nem T(Xu) végezd el:
      u ← u – 1
    vége(amíg)
    Ha e < u akkor
      Xe ← Xu
      e ← e + 1
      Amíg (e < u) és T(Xe) 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.

1.1.14. 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.

a. Másolással összeépítés

A másolás bármelyik programozási tétellel egybeépíthető. Ilyenkor az X_i bemeneti adatra való hivatkozást $f(x_i)$-re cseréljük.

Példa:

Adjuk meg egy számsorozat elemeinek négyzetgyökeiből álló sorozatot!

Megoldás: másolás + sorozatszámítás

b. Megszámlálással összeépítés

A megszámlálást általában egy döntéssel, kiválasztással vagy kereséssel építhetjük össze.

Példa:

Döntsük el, hogy található-e az N elemű X sorozatban legalább K darab T tulajdonságú elem? Adjuk meg a sorozat K-dik T tulajdonságú elemét!

Megoldás: megszámlálás + döntés + kiválasztás

c. Maximumkiválasztással összeépítés

A maximumkiválasztást összeépíthetjük megszámlálással, kiválogatással.

Példa:

Hány darab maximumértékű elem van az adott sorozatban? Generáljuk ezen elemek indexeinek sorozatát!

Megoldás: Lásd a *MindenMaximumHelye(N, X, db, indexek)* algoritmust.

d. Kiválogatással összeépítés

Olyan feladatoknál, amelyeknek esetében a feldolgozást csak az adott sorozat T tulajdonságú elemeire kell elvégeznünk, alkalmazható a kiválogatással történő összeépítés.

1.2. Lépések finomítása és optimalizálás

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,* vagyis az algoritmus részletes kidolgozása. Következik a helyesség bizonyítása és a bonyolultság kiértékelése. A program megírását (kódolást) a tesztelés követi.

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.

1.2.1. Megoldott feladatok

a. 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* és *b* egészosztási maradékát. 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* értékét megkapjuk, ha *a* és *b* szorzatát elosztjuk az *lnko*-val. Az eredeti két szám értékét az algoritmus „tönkreteszi”, ezért szükséges ezeket elmenteni két segédváltozóba (*x* és *y*).

```
Algoritmus Eukleidész_1(a, b, lnko, lkkt):
                                  { Bemeneti adatok: a, b. Kimeneti adatok: lnko, lkkt }
  x ← a                         { szükségünk lesz a és b értékére az lkkt kiszámításakor }
  y ← b
  r ← a mod 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 ← a mod b                                 { kiszámítjuk az aktuális maradékot }
  vége(amíg)
  lnko ← b                                      { lnko egyenlő az utolsó osztó értékével }
  lkkt ← x*y div lnko                           { felhasználjuk a és b másolatait }
```

```
Vége(algoritmus)
```

Az algoritmust megvalósíthatjuk 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 ← a                          { Bemeneti adatok: a, b. Kimeneti adatok: lnko, lkkt }
  y ← b
  Amíg a ≠ b végezd el:
    Ha a > b akkor
      a ← a - b
    különben
      b ← b - a
    vége(ha)
  vége(amíg)
  lnko ← a
  lkkt ← [x*y/lnko]
Vége(algoritmus)
```

b. Prímszámok

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

```
Algoritmus Prím(n, válasz):
                                 { Bemeneti adat: n. Kimeneti adat: válasz }
  @ Megállapítjuk, hogy n prímszám-e
  Ha n prímszám akkor
    válasz ← igaz
  különben
    válasz ← hamis
  vége(ha)
Vége(algoritmus)
```

Lépések finomítása: Ki kell dolgoznunk azt a módot, ahogyan megállapíthatjuk, 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 *n* 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):
  osztók_száma ← 0               { Bemeneti adat: n. Kimeneti adat: válasz }
  Minden osztó = 1,n végezd el:
    Ha n mod osztó = 0 akkor
      osztók_száma ← osztók_száma + 1
    vége(ha)
  vége(minden)
  válasz ← osztók_száma = 2
```

```
Vége(algoritmus)
```

1.2.2. 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. Azt is tudjuk, hogy az egyetlen páros prímszám a 2. Így elérhetjük, hogy a páros számok lekezelése után csak páratlan számokat vizsgáljunk, és ezeket csak páratlan osztókkal próbáljuk meg elosztani. 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 ← hamis
  különben
    Ha n páros akkor
      prím ← n = 2
    különben
      prím ← igaz
      osztó ← 3
      négyzetgyök ← [√n]                { a négyzetgyök egész része }
      Amíg prím és (osztó ≤ négyzetgyök) végezd el:
        Ha n mod osztó = 0 akkor
          prím ← hamis
        különben
          osztó ← osztó + 2
        vége(ha)
      vége(amíg)
    vége(ha)
  vége(ha)
  válasz ← prím
Vége(algoritmus)
```

Ha ebben az algoritmusban felhasználjuk a matematikából ismert tulajdonságot, éspedig: minden 5-nél nagyobb prímszám $6k \pm 1$ alakú, akkor a vizsgálandó számok száma tovább csökkenthető. Mivel az előbbi állításból következik, hogy prímszámokat keresni csak 6

többszöröseinél 1-gyel kisebb, illetve 1-gyel nagyobb számok között érdemes, a fenti algoritmus a következőképpen változik:

```
Algoritmus Prím(határ):
  Ha n = 1 akkor
    prím ← hamis
  különben
    Ha n páros akkor
      prím ← n = 2
    különben
      Ha n ≤ 5 akkor                                    { n = 3}
        prím ← igaz
      különben
        Ha ((n - 1) mod 6 ≠ 0) és ((n + 1) mod 6 ≠ 0) akkor
          prím ← hamis
        különben
          osztó ← 3
          ...                          { tovább ugyanaz, mint az előző algoritmusban }
```

Továbbá, ismeretes, hogy a négyzetgyököt számoló függvény ismeretlen lépésszámban határozza meg az eredményt, amely valós szám. Ezt elkerülendő, lemondunk a négyzetgyök kiszámításáról és az *Amíg* feltételét a következőképpen írjuk:

```
...
Amíg prím és (osztó * osztó ≤ n) végezd el:
...
```

Így, nem dolgozunk valós számokkal és nem számítjuk ki fölöslegesen a négyzetgyököt.

Ha sok számról kell eldöntenünk, hogy prím-e, érdemes előbb létrehozni *Eratosztenész* szita-módszerével prímszámok sorozatát (megfelelő darabszámmal) és az algoritmusban csak ennek a sorozatnak elemeivel osztani.

```
Algoritmus Prímek(határ, prím):
                                                { határ-nál kisebb számokat vizsgálunk }
                                { a generált prímszámokat a prím logikai tömb alapján lehet értékesíteni }
  Minden i=2,határ végezd el:
    prím_i ← igaz                                  { még nincs kihúzva egy szám sem }
  vége(minden)
  Minden i = 2, határ div 2 végezd el:
    Ha prím_i akkor                                              { ha i még nincs kihúzva }
      k ← 2 * i                                { az első kihúzandó szám (i-nek többszöröse) }
      Amíg k ≤ határ végezd el:
        prím_k ← hamis                                           { kihúzzuk a k számot }
        k ← k + i                                { a következő kihúzandó többszöröse i-nek }
      vége(amíg)
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

1.2.3. 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.

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 a részfeladatokat 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.

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

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!

A modularitás alapelvei

- *A modulokat nyelvi egységek támogassák:* 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 *zárt*nak nevezünk, ha más modulok számára egy jól definiált felületen keresztül elérhető, a többi modul ezt változatlan formában felhasználhatja. Egy modult *nyitott*nak 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.
Egy típus, egy modul: Egy típus műveletei kerüljenek egy modulba.

1.3. Rendező algoritmusok

Összehasonlításon alapuló rendezések

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

1.3.1. Buborékrendezés (*Bubble-sort*)

A rendezés során páronként összehasonlítjuk a számokat 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 BuborékRendezés(n, a):
  k ← n                    { Bemeneti adatok: n, a. Kimeneti adat: a rendezett a sorozat }
  Ismételd
    nn ← k - 1
    rendben ← igaz
    Minden i = 1, nn végezd el:
      Ha a_i > a_i + 1 akkor
        rendben ← hamis
        a_i ↔ a_i + 1
        k ← i                                          { az utolsó csere helye }
      vége(ha)
    vége(minden)
  ameddig rendben
Vége(algoritmus)
```

1.3.2. Egyszerű felcseréléses rendezés

Ez a rendezési módszer hasonlít a buborékrendezéshez, de kötelezően elvégez minden páronkénti összehasonlítást (míg a buborékrendezés bonyolultsága a legjobb esetben $\Omega(n)$, ez az algoritmus mindig $O(n^2)$ bonyolultságú). Ha egy elempár sorrendje nem megfelelő, felcseréli őket.

```
Algoritmus FelcserélésesRendezés(n, a):
                           { Bemeneti adatok: n, a; Kimeneti adat: a rendezett a sorozat }
  Minden i = 1, n - 1 végezd el:
    Minden j = i + 1, n végezd el:
```

```
      Ha a_i > a_j akkor
        a_i ↔ a_j
      vége(ha)
    vége(minden)
  vége(minden)
Vége(algoritmus)
```

1.3.3. 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 MinimumkiválasztásosRendezés(n, a):
                          { Bemeneti adatok: n, a; Kimeneti adat: a rendezett a sorozat }
  Minden i = 1,n-1 végezd el:
    indMin ← i
    Minden j = i+1, n végezd el:
      Ha a_indMin > a_j akkor
        indMin ← j
      vége(ha)
    vége(minden)
    a_i ↔ a_indMin
  vége(minden)
Vége(algoritmus)
```

1.3.4. Beszúró rendezés

A beszúró rendezés hatékony algoritmus kisszámú elem rendezésére. Úgy dolgozik, ahogy bridzsezés közben a kezünkben levő lapokat rendezzük: üres bal kézzel kezdünk, a lapok fejjel lefelé 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 levő 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 BeszúróRendezés(n, a):
                          { Bemeneti adatok: n, a. Kimeneti adat: a rendezett a sorozat }
  Minden j = 2, n végezd el:
    segéd ← a_j                    { beszúrjuk a_j-t az a_1, ..., a_(j–1) rendezett sorozatba }
    i ← j - 1
    Amíg (i > 0) és (a_i > segéd) végezd el:
      a_i+1 ← a_i
      i ← i - 1
    vége(amíg)
    a_i+1 ← segéd
```

```
  vége(minden)
Vége(algoritmus)
```

Lineáris rendezések

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, ezért ezeket az algoritmusokat *összehasonlító rendezés*eknek is nevezzük.

1.3.5. Leszámláló rendezés (ládarendezés, *Binsort*)

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 segédtömb *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 segédtömb elemeinek értékei alapján.

```
Algoritmus LádaRendezés(a, n):
  Minden i = 1, k végezd el:              { Bemeneti adatok: n, a; Kimeneti adat: a }
    segéd_i ← 0
  vége(minden)
  Minden j = 1, n végezd el:
    segéd_{a_j} ← segéd_{a_j + 1}
  vége(minden)
  q ← 0
  Minden i = 1, k végezd el:                          { a segéd tömbnek k eleme van }
    Minden j = 1, segéd_i végezd el:
      q ← q + 1                                    { a segéd_i elemek összege n }
      a_q ← i                                      { tehát a feldolgozások száma n }
    vége(minden)
  vége(minden)
Vége(algoritmus)
```

1.3.6. Számjegyes rendezés (*radixsort*)

Ha egész számokat tároló sorozatot szeretnénk rendezni, elképzelhetjük a számokat egymás alá írva és alkalmazhatjuk a fenti algoritmust rendre, minden számjegy-oszlopra. Ha a legnagyobb szám számjegyeinek darabszáma *d*, a sorozatot *d*-szer vizsgáljuk. A számjegyes rendezés először a legkevésbé fontos számjegy alapján rendez. A számokat az utolsó számjegyük alapján rendezzük oly módon, hogy ha csak ezt a számjegyet tekintjük, növekvő sorrendet lássunk. Ezután a számokat újra rendezzük a második legkevésbé értékes

számjegyük alapján. Ezt mindaddig végezzük, ameddig a számokat mind a *d* számjegy szerint nem rendeztük.

```
Algoritmus SzámjegyesRendezés(a, d):
  Minden i = 1, d végezd el:
    stabil leszámlálással rendezzük az a tömböt az i-edik számjegy szerint
  vége(minden)
Vége(algoritmus)
```

1.4. Rekurzió

1.4.1. Rekurzív alprogramok

Bármely algoritmus megvalósítható *iteratívan és/vagy rekurzívan.* Mindkét technikának a lényege: *bizonyos utasítások ismételt végrehajtása*. Az iteratív algoritmusokban az ismétlést *ciklusokkal* valósítjuk meg. A rekurzív algoritmusokban az ismétlés azáltal valósul meg, hogy az illető alprogram *meghívja önmagát*, amikor még aktív.

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.

Rekurzív algoritmust akkor érdemes tervezni, ha a feladat *eredménye rekurzív szerkezetű*, ha a megoldás legjobb módszere a visszalépéses keresés (*backtracking*) vagy az *oszd meg és uralkodj* módszer (*divide et impera*), illetve ha *a feldolgozandó adatok rekurzívan definiáltak* (pl. bináris fák). Ugyanakkor előfordulhat, hogy túlságosan *igénybe veszi a végrehajtási vermet*, és a *futási ideje nagyobb*, mint az iteratív változatnak

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:

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

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 rekurzívaknak, melyek meghívják önmagukat. Ha ez a hívás az illető alprogram összetett utasításában szerepel, *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.

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 túlcsordulás 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, tehát, 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.

1.4.2. Megoldott feladatok

1. Egy szó betűinek megfordítása

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

Megoldás

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

```
Algoritmus Fordít:    { nincs paraméter, mivel az alprogramban olvasunk be és írunk ki }
  Be: betű
  Ha nem szóköz akkor
    Fordít              { meghívja önmagát, hogy megfordíthassa a fennmaradt részt }
  különben
    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 fenti példában: **Ki:** betű), így természetes módja a feladat megoldásának.

2. Szavak sorrendjének megfordítása

Olvassunk be *n* szót, majd írjuk ki ezeket (tömbhasználat nélkül) a beolvasás fordított sorrendjében!

```
Algoritmus SzavakatFordít_1(n):
  Be: szó                        { az első hívás aktuális paramétere n = szavak száma }
  Ha n > 1 akkor
    SzavakatFordít_1(n-1)
  különben
    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 SzavakatFordít_2(i):
  Be: szó                                { most az első hívás aktuális paramétere 1 }
  Ha i < n akkor
    SzavakatFordít_2(i+1)
  különben
    Ki: 'Fordított sorrendben: '
  vége(ha)
  Ki: szó
Vége(algoritmus)
```

3. Faktoriális

Számítsuk ki az adott *n* szám faktoriálisát!

Megoldás

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

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

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

4. Legnagyobb közös osztó

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

Megoldás

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, m* **mod** *n*)-nel. Tehát fel lehet írni a következő rekurzív definíciót:

$$Lnko(m,n) = \begin{cases} n, & \text{ha } m \bmod n = 0 \\ Lnko(n, m \bmod n), & \text{ha } m \bmod n \neq 0 \end{cases}$$

```
Algoritmus Lnko(m, n):                          { Bemeneti adatok:m, n }
  mar ← m mod n
  Ha mar = 0 akkor
    térítsd n
  különben
    térítsd Lnko(n, mar)
  vége(ha)
Vége(algoritmus)
```

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

5. Descartes-szorzat

Egy rajzon *n* virágot szeretnénk kiszínezni. A festékeket az 1, 2, ..., *m* számokkal kódoljuk. Bármely virág, bármilyen színű lehet, de szeretnénk tudni, hány féle módon lehetne ezeket különböző módon kiszínezni. Tulajdonképpen az M^n *Descartes*-szorzatot kell generálnunk:

```
Algoritmus DescartesSzorzat(i):          { Bemeneti adat: i, az első híváskor = 1 }
  Minden j = 1, m végezd el:
    x_i ← j
    Ha i < n akkor
      DescartesSzorzat(i+1)
    különben
      Kiír
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

6. *k* elemű részhalmazok

Adott két egész szám: n és k ($1 \le k \le n$). Generáljuk rekurzívan az {1, 2, ..., n} halmaz minden k elemet tartalmazó részhalmazát!

Megoldás

Az {1, 2, ..., n} halmaz k elemet tartalmazó részhalmaza egy k elemű tömb segítségével kódolható: x_1, x_2, ..., 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), ugyanakkor ne veszítsünk el egyet sem. Ha az x sorozatba az elemeket szigorúan növekvő sorrendben tesszük ($x_1 < x_2 < ... < x_k$), egy részhalmazt csak egyszer állíthatunk elő. Mivel minden x_i szigorúan nagyobb, mint x_{i-1}, az értékei $x_{i-1} + 1$-től kezdődően $n - (k - i)$-ig nőnek.

```
Algoritmus Részhalmazok(i):              { k és x globális változó, x_i = 0, i = 0, 1, … }
  Minden j = x_{i-1} + 1, n - k + i végezd el:
    x_i ← 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.

7. Fibonacci-sorozat

Generáljuk a Fibonacci-sorozat első n elemét!

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

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
    térítsd 0
  különben
    Ha n = 2 akkor
      térítsd 1
    különben
      térítsd 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, amitől kezdődően újra, meg újra

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 Fib(n):
  Ha n > 2 akkor
    Fib(n-1)
    f_n ← f_n-1 + f_n-2
  különben
    f_1 ← 0
    Ha n = 2 akkor f_2 ← 1 vége(ha)
  vége(ha)
Vége(algoritmus)
```

8. Az {1, 2, ..., *n*} halmaz minden részhalmaza

Generáljuk az {1, 2, ..., *n*} halmaz minden részhalmazát!

Elemzés

A halmazokat az $x_1 < x_2 < ... < x_i$ sorozattal ábrázoljuk, ahol $i = 1, 2, ..., n$. 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 lexikográfikus sorrendben generáljuk. Figyelemre méltó, hogy minden új elem generálása egy új részhalmazhoz vezet.

```
Algoritmus MindenRészhalmaz(i)
  Minden j = x_i-1 + 1, n végezd el:
    x_i ← j
    Kiír(i)
    MindenRészhalmaz(i+1)
  vége(minden)
Vége(algoritmus)
```

A kilépési feltétel lehetne $x_i = n$, de erre nincs szükség, mivel a *Minden* struktúra végső értéke leállítja a végrehajtást: ha $x_i = n$, a ciklusváltozó kezdőértéke $x_i + 1 = n + 1$, tehát nagyobb, mint n (végső érték), így a *Minden* ciklusmagja nem hajtódik végre és a program kilép az aktuális hívásból. Az algoritmust *MindenRészhalmaz*(1) alakban hívjuk meg.

9. Partíciók

Generáljuk az $n \in$ N* szám partícióit!

Megoldás

Partíció alatt azt a felbontást értjük, amelynek során az $n \in$ N* számot pozitív számok összegeként írjuk fel: $n = p_1 + p_2 + ... + p_k$, ahol $p_i \in$ N*, $i = 1, 2, ..., k$, $k = 1, ..., n$. Két partíciót kétféleképpen tekinthetünk különbözőnek: ha vagy az *előforduló értékek* vagy az *előfordulásuk sorrendje* különbözik vagy, ha csak az *előforduló értékek* különböznek.

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:
    pi ← j
    Ha j < n akkor
      Partíció(i+1, n-j)
    különben
      Kiír(i)
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

1.5. 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 feladat minden lehetséges megoldást kér, és csak exponenciális algoritmussal tudjuk megoldani, a *backtracking* (*visszalépéses keresés*) módszert alkalmazzuk, amely exponenciális ugyan, de megpróbálja csökkenteni a generálandó próbálkozások számát.

1.5.1. A visszalépéses keresés általános bemutatása

A visszalépéses keresés azon feladatok megoldásakor alkalmazható, amelyeknek eredményét az $M_1 \times M_2 \times ... \times M_n$ Descartes-szorzatnak azon elemei alkotják, amelyek eleget tesznek bizonyos belső feltételeknek. Az $M_1 \times M_2 \times ... \times M_n$ 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 $x = (x_1, x_2, ..., x_n) \in M_1 \times M_2 \times ... \times M_n$ elemét, hanem csak azokat, amelyeknek 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 $x_1, x_2, ..., x_{i-1}$ már kaptak végleges értéket az aktuálisan generált eredményben. Az x_i-re vonatkozó javaslatot akkor fogadjuk el, amikor $x_1, x_2, ..., x_{i-1}$ 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 amelyeknek esetében a folytatási feltételek nem kérnek további elemeket, *végeredmény*eknek 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: *Megfelel(i)*, ahol *i* az aktuálisan generált elem indexe. Ez az alprogram *igaz* értéket térít vissza, ha az x_i elem az eddig generált x_1, x_2, ..., x_{i-1} elemekkel együtt megfelel a belső feltételeknek, és hamis értéket ellenkező esetben.

```
Algoritmus Megfelel(i):                    { a függvény Megfelel értékét téríti }
  Megfelel ← igaz
  Ha a belső feltételek x1, x2, ..., xi esetében nem teljesülnek akkor
    Megfelel ← hamis
  vége(ha)
Vége(algoritmus)
```

A feladat által kért eredményt a következő algoritmussal generáljuk:

```
Algoritmus RekurzívBacktracking(i):
  Minden mj ∈ Mi értékre végezd el:
    xi ← mj
    Ha Megfelel(i) akkor      { megvalósulnak a belső feltételek x1, x2, ..., xi esetében }
      Ha i < n akkor
        RekurzívBacktracking(i+1)
      különben
        Ki: x1, x2, ..., xn
      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 aktiválódásnál végrehajtja az algoritmus.

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

Összefoglalva, a következő lépéseket kell elvégeznünk:

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

1.5.2. Megoldott feladatok

1. Nyolc 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, ..., 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 $K_i \in \{1, 2, \ldots, 8\}$, $i = 1, \ldots, 8$. Az eddigiekből következik, hogy egy eredmény az $\{1, 2, \ldots, 8\}^8$ Descartes-szorzat eleme. Tehát, ha meg akarjuk oldani a feladatot, tulajdonképpen az $\{1, 2, \ldots, 8\}^8$ Descartes-szorzat egy részhalmazát kell meghatároznunk, azzal a feltétellel, hogy a 8 királynő, amelyek a K_1, K_2, ..., 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 $\{1, 2, ..., 8\}^8$ 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 írnunk. Az elhelyezést a *Királynő*(i) rekurzív alprogram végzi el, a támadási lehetőséget a *NemTámad*(i) logikai függvény ellenőrzi.

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

```
Algoritmus NemTámad(i):
  Jó ← igaz                                   { Jó = lokális változó, K globális }
  j ← 1
  Amíg (j ≤ i-1) és Jó végezd el:
    Ha (Ki = Kj) vagy (i-j = |Ki - Kj| akkor
      Jó ← hamis                              { az i. és j. királynők támadják egymást }
    különben
      j ← j + 1
    vége(ha)
```

```
  vége(amíg)
  térítsd Jó
Vége(algoritmus)

Algoritmus Királynő(i):
  Minden j = 1, 8 végezd el:
    K_i ← j                          { az i-edik királynőt a j-edik oszlopba tesszük }
    Ha NemTámad(i) akkor             { az i-edik királynő nem támadja egyiket sem }
      Ha i < 8 akkor
        Királynő(i+1)
      különben
        Kiír
      vége(ha)
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

Az első hívás alakja: *Királynő*(1).

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 előbbi mintától, 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 $x_i \in \{1, 2, ..., m\}$, ahol m a gyermekek száma, ($i = 1, 2, ..., n$).

Belső feltételek: $x_i \neq x_j$, $i \neq j$, $i, j = 1, 2, ..., n$. A belső feltételek azt fejezik ki, hogy az i. székre csak olyan gyerek ülhet le, aki pillanatnyilag még áll. Az ellenőrzés egyszerűbb lesz, fölhasználunk egy *mégÁll* logikai tömböt, ahol $mégÁll_j$ igaz, ha a j-edik gyerek még nem ült le, és hamis ellenkező esetben. Az $x_i \neq x_j$, $j = 1, 2, ..., i - 1$ feltételek a következőképpen alakulnak át: $mégÁll_{x_i} = igaz$.

Folytatási feltétel: $i < n$ (még van szabad szék)

A *mégÁll* 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 ülhet majd le. Ugyanakkor, a j-edik gyermek egy másik ültetési rendben újból leülhet. A j-edik gyermek felállítása maga után vonja a megfelelő $mégÁll_j$ visszaállítását *igaz*-ra. Ez a megoldás hatékonyabb, mint az, amelyet a mintaalgoritmus alapján készíthetnénk, mivel kevesebb összehasonlítást végez.

```
Algoritmus Variáció(i):
  Minden j = 1, m végezd el:
    Ha mégÁll_j akkor                { a j-edik gyermek még áll }
      x_i ← j
```

```
      mégÁll_j ← hamis                { a j-edik gyermeket leültettük az i-edik székre }
      Ha i < n akkor
        Variáció(i+1)
      különben
        Kiír
      vége(ha)
      mégÁll_j ← igaz   { a j-edik gyermeket felállítjuk, hogy később ülhessen más székre }
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

Az első hívás alakja: *Variáció*(1).

3. Zárójelek

Generáljunk és írjunk ki minden helyesen nyitó és csukó n zárójelet tartalmazó karakterláncot! *Példa:* ha $n = 4$, a helyes karakterláncok:

```
(())
()()
```

Megoldás

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

Belső feltételek: Adott pillanatban ne létezzen több csukó zárójel, mint nyitó, és nyitó nem lehet több mint $n/2$. Mivel a megoldások tere kételemű halmaz, és a két elem esetében a belső feltétel különbözik, lemondunk a *Minden* struktúráról és két *Ha* utasítással ellenőrizzük ezeket.

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

$$\begin{cases} ny < \dfrac{n}{2}, & \text{ha } x_i = '(' \\ z < ny, & \text{ha } x_i = ')' \end{cases} \qquad \forall i = \overline{2,\ n\text{-}1}$$

Folytatási feltételek: Mivel bármely eredményben $x_1 = '('$ és $x_n = ')'$, a hívó programegységben elvégezzük az inicializálásokat: $x_1 \leftarrow '('$ és $x_n \leftarrow ')'$. Tehát, az algoritmus a második helytől kezdődően az $(n - 1)$-dik helyig tesz zárójeleket. Amikor az n-edik karakter következne, leállunk. Az első hívás alakja: *Zárójel*(2, 1, 0).

```
Algoritmus Zárójel(i, ny, cs):
  Ha i = n akkor                                          { kilépési feltétel }
    Ki: x                                                 { x egy karakterlánc }
  különben
    Ha ny < n div 2 akkor
      x_i ← '('
      Zárójel(i+1, ny+1, cs)
    vége(ha)
    Ha cs < ny akkor
      x_i ← ')'
```

```
      Zárójel(i+1, ny, cs+1)
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

4. Labirintus

Egy labirintust egy n soros és m oszlopos L kétdimenziós tömbben tárolunk, amelyben a folyosónak megfelelő elemek értéke 1; ezek az értékek egymás után következnek a labirintust ábrázoló tömbben, egy bizonyos sorban, vagy oszlopban. Egy személyt leengednek ejtőernyővel 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 kér. Legyen egy ilyen út hossza k. Az utat az $x_1, x_2, ..., x_k$ és $y_1, y_2, ..., y_k$ sorozatokkal kódolunk, amelyek azokat a sorokat és oszlopokat tartalmazzák, amelyeknek érintésével kifele haladunk a labirintusból. $x_i \in \{1, 2, ..., n\}$, $y_i \in \{1, 2, ..., m\}$, $i = 1, 2, ..., k$.

Belső feltételek: Az útvonalra a következő belső feltételek érvényesek:

a) Folyosón kell haladnia: $L_{x_i, y_i} = 1$, $i = 1, 2, ..., k$.

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

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: $(x_i, y_i) \neq (x_j, y_j)$, $j = 1,..., i - 1$.

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

$$eredmény_{ij} = \begin{cases} \text{az a lépésszám, amellyel az } (i, j) \text{ helyre léptünk,} & \text{ha lehet lépni az } (i, j) \text{ helyre} \\ 0, & \text{ha nem lehet lépni az } (i, j) \text{ helyre} \end{cases}$$

Egy bizonyos helyről négy irányba léphetünk. Az alábbi kód tartalmaz egy figyelemreméltó egyszerűsítést, ami a folytatási feltételeket illeti. Nem szükséges ellenőriznünk 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 1 lépésszámra hívjuk meg.

```
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 ← lépés                                  { az (i, j) helyre lépünk }
    Ha (i ∈ {1, n}) vagy (j ∈ {1, m}) akkor
      Kiír                          { kijárathoz 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 }
    eredm_ij ← 0    { töröljük az utolsó lépést, hogy egy új útvonalon léphessünk újra ide }
  vége(ha)
```

```
Vége(algoritmus)
```

Második megoldás

```
Algoritmus Út(i, j, lépés):
  Minden irány = 1, 4 végezd el:                    { kiválasztunk egy irányt }
    új_i ← i + x_irány                              { (új_i, új_j) az új koordináták }
    új_j ← j + y_irány
    Ha (új_i ∈ {1, 2, ..., n}) és (új_j ∈ {1, 2, ..., m}) akkor
      Ha (L_új_i,új_j = 1) és (eredm_új_i,új_j = 0) akkor
        eredm_új_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)
        eredm_új_i,új_j ← 0                         { lemondunk az utolsó lépésről }
      vége(ha)
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

Az *Út(i, j, lépés)* 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: $x = (-1, 0, 1, 0)$, $y = (0, 1, 0, -1)$.

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.

Ebben a második megoldásban nem vettük körül a labirintust az első algoritmusban említett kerettel. Ennek következtében szükséges volt ellenőrizni, hogy az új hely, ahova lépni akarunk a labirintuson belül van-e.

Ezt az 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, észrevesszük, hogy mivel az előrehaladás egy kétdimenziós tömbben történik, az alprogram két paramétere (*i*, *j*) annak a helynek a koordinátái, ahova utoljára léptünk.

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

Az *oszd meg és uralkodj* módszer (*divide et impera*) 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 adathalmazra definiáltak. A részfeladatokkal hasonlóan járunk el és a felbontást akkor állítjuk le, amikor a feladat megoldása a lehető legjobban leegyszerűsödött. A maximálisan leegyszerűsített feladatot megoldjuk, majd 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.

1.6.1. Az oszd meg és uralkodj módszer általános bemutatása

A *DivImp*(*bal, jobb, eredm*) algoritmus az a_1, a_2, ..., 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 < ε akkor                { ha a feladat maximálisan leegyszerűsödött }
    Megold(bal, jobb, eredm)     { kiszámítjuk az egyszerű feladat eredm eredményét }
  különben
    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.

1.6.2. Megoldott feladatok

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 részszorzatokra bontjuk: 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, ..., x_n) részfeladat általános alakja: *Szorzat*(x_{bal}, ..., x_{jobb}). Minden részfeladat más-más szorzatot számol ki, tehát a feladatok függetlenek egymástól.

```
Algoritmus Szorzat(bal ,jobb):
  Ha jobb = bal akkor        { Bemeneti adatok: bal, jobb. A függvény a szorzatot téríti }
    térítsd xbal                                          { a részsorozat egy elemből áll }
  különben
    Ha jobb - bal = 1 akkor
      térítsd xbal * xjobb                                { a részsorozat két elemű }
    különben                          { felbontjuk a Szorzat(bal, ..., jobb) feladatot }
      közepe ← (bal+jobb) div 2
      p1 ← Szorzat(bal, közepe)
      p2 ← Szorzat(közepe+1, jobb)
      térítsd p1 * p2                                     { összerakjuk a részeredményeket }
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

2. 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 sorozatban, a sorszámnak megfelelő paraméter értéke legyen 0.

Megoldás

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

1. *keresett* = $x_{közép}$ ⇒ keresett a sorban a *közép* helyen található;
2. *keresett* < $x_{közép}$ ⇒ mivel a sorozat rendezett, a keresett számot a sorozat első (x_1, ..., $x_{közép-1}$) felében keressük tovább;
3. *keresett* > $x_{közép}$ ⇒ a keresett számot a sorozat második ($x_{közép+1}$, ..., 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 BinKeres(x, bal, jobb, keresett, közép):
                          { Bemeneti adatok: x, bal, jobb, keresett. Kimeneti adat: közép }
  Ha bal > jobb akkor
    közép ← 0                                             { keresett nincs a sorozatban }
  különben
    közép ← (bal+jobb) div 2
```

```
    Ha keresett < x_közép akkor
      BinKeres(x, bal, közép-1, keresett, közép)
    különben
      Ha keresett > x_közép akkor
        BinKeres(x, közép+1, jobb, keresett, közép)
      vége(ha)                          { ha keresett = x_közép megvan a pozíció }
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

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

```
Algoritmus BinKeresIteratív(n, x, keresett, közép):
  bal ← 1
  jobb ← n
  megvan ← hamis
  Amíg nem megvan és (bal ≤ jobb) végezd el:
    közép ← (bal+jobb) div 2
    Ha x_közép = keresett akkor
      megvan ← igaz                     { közép tartalmazza a keresett helyét }
    különben
      Ha x_közép > keresett akkor
        jobb ← közép - 1
      különben
        bal ← közép + 1
      vége(ha)
    vége(ha)
  vége(amíg)
  Ha nem megvan akkor
    közép ← 0                           { ha közép értéke 0 ⇒ keresett nem található }
  vége(ha)
Vége(algoritmus)
```

3. Összefésülésen alapuló rendezés (*MergeSort*)

Rendezzünk növekvő sorrendbe egy egész számokból álló sorozatot összefésüléssel!

Megoldás

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. Az adott sorozatot két részre osztjuk, abból a célból, hogy rendezhessük. De ezeket újból felosztjuk, amíg a kapott tömb, amelyet rendeznünk 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:
```

```
    a_i ← x_i
  vége(minden)
  Minden i = közép+1, jobb végezd el:
    b_i ← x_i
  vége(minden)
  a_közép+1 ← végtelen
  b_jobb+1 ← végtelen                                    { strázsák }
  i ← bal
  j ← közép + 1
  Minden k = bal, jobb végezd el:
    Ha a_i < b_j akkor
      x_k ← a_i
      i ← i + 1
    különben
      x_k ← b_j
      j ← j + 1
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

```
Algoritmus Rendez(bal, jobb):
  Ha bal < jobb akkor
    közép ← (bal+jobb) div 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}, …, x_{jobb} rendezett sorozat, amelybe tulajdonképpen ugyanazon sorozat két részsorozatát, az x_{bal}, …, $x_{közép}$ és az $x_{közép+1}$, …, x_{jobb} részsorozatokat fésültük össze. Ezzel magyarázható annak a szükségessége, hogy az összefésülendő sorozatokat átmásoltuk az *a* illetve a *b* sorozatba. A hívó programegységben a *Rendez*(1, *n*) algoritmust hívjuk.

4. Gyorsrendezés (*QuickSort*)

Fölhasználva a *quiksort* algoritmust, rendezzünk növekvő sorrendbe *n* egész számot!

Megoldás

A gyorsrendezés az oszd meg és uralkodj módszeren alapszik, mivel az eredeti sorozatot úgy rendezi, hogy két rendezendő részsorozatra bontja. A részsorozatok rendezése egymástól függetlenül történik. A részeredmények összerakása hiányzik (hasonlóan a bináris kereséshez). Amikor az x_1, …, x_n sorozatot készülünk rendezni, előbb előkészítünk két részsorozatot (x_1, …, x_{m-1} és x_{m+1}, …, x_n) úgy, hogy az x_1, …, x_{m-1} részsorozat elemei kisebbek legyenek, mint az x_{m+1}, …, x_n részsorozat elemei. Közöttük található az x_m, amely nagyobb, mint az x_1, …, x_{m-1} részsorozat bármely eleme, és kisebb, mint az x_{m+1}, …, x_n részsorozat összes eleme.

Azt az elemet, amely meghatározza a helyet, ahol az adott tömb két részre oszlik, *strázsa*nak (*őrszem*) nevezzük. Ennek a helynek a meghatározása kulcskérdés az algoritmus végrehajtása során. A strázsa *m* helyét úgy határozzuk meg, hogy az x_1, …, x_m tömbben legyenek azok az elemek, amelyek kisebbek, mint a strázsa és az x_{m+1}, …, 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 strázsa (és a tömb első részében találhatók) azokkal, amelyek kisebbek, mint a strázsa (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.

```
Algoritmus QuickSort(bal, jobb):
  Ha bal < jobb akkor                { meghatározzuk azt az m helyet, ahol a sorozatot }
                                     { két részre bontjuk, miközben az x_m elem a végleges helyére kerül }
    m ← Strázsa_helye(bal, jobb)
    QuickSort(bal, m)                { hasonlóan járunk el az (x_bal, ..., x_m) részsorozattal }
    QuickSort(m+1, jobb)             { valamint az (x_m+1, ..., x_jobb) részsorozattal }
  vége(ha)
Vége(algoritmus)
```

Látható, hogy a rekurzív hívásoknak megfelelően, az algoritmus meghívja önmagát egy bal meg egy jobb részsorozat rendezése érdekében. Dacára annak, hogy az algoritmus nem tartalmaz összehasonlításokat és felcseréléseket, a sorozat aközben rendeződik, miközben keressük a strázsa *m* helyét:

```
Algoritmus StrázsaHelye(bal, jobb):
  strázsa ← x_bal                  { Bemeneti adatok: bal, jobb. Kimeneti adat: a strázsa helye }
  i ← bal-1
  j ← jobb+1                       { megkeressük azt a j indexet, amelyre bal ≤ j < jobb }
  Ismételd
    Ismételd                       { megkeressük azt a j indexet (jobbról balra), amelyre x_j > strázsa }
      j ← j - 1
    ameddig x_j ≤ strázsa
    Ismételd                       { megkeressük azt az i indexet (balról jobbra), amelyre x_i < strázsa }
      i ← i + 1
    ameddig x_i ≥ strázsa
    Ha i < j akkor
      x_i ↔ x_j                    { felcseréljük ezt a két nem megfelelő tulajdonságú elemet }
    vége(ha)
  ameddig i ≥ j                    { addig folytatjuk a keresést és felcserélést, amíg i kisebb, mint j }
  térítsd j                        { megtaláltuk az új strázsa helyét }
Vége(algoritmus)
```

Megjegyzés

Ez az algoritmus főleg akkor gyors, ha a sorozat nem rendezett, egyébként előfordulhat, hogy az algoritmus négyzetes bonyolultságúvá válik. Ha fennáll ez a veszély, tanácsos a ren-

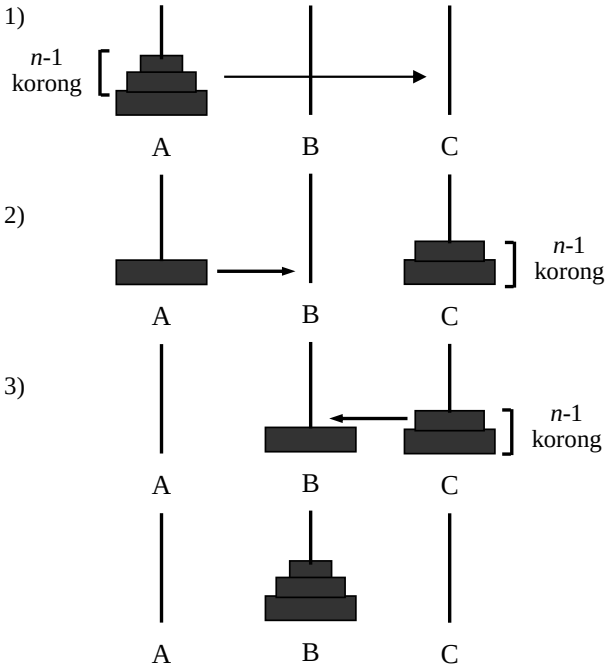
dezés előtt véletlenszerűen átrendezi a bemeneti sorozatot, ezzel biztosítva a különböző permutációk azonos valószínűségét. Ez a módosítás nem javít a legrosszabb futási időn, de biztosítja, hogy a futási idő független lesz a bemeneti elemek sorrendjétől.

5. 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 mozgatás 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:



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 = 1 akkor
    Költöztess egy korongot A-ról B-re
  különben
    Hanoi(n-1, A, C, B)
    Hanoi(1, A, B, C)
    Hanoi(n-1, C, B, A)
  vége(ha)
```

```
Vége(algoritmus)
```

Az algoritmust a *Hanoi(n, A, B, C)* utasítással hívjuk, ahol az aktuális paraméterek értékei `'A'`, `'B'`, `'C'`. A *Költöztess egy korongot A-ról B-re* lehet pl. egy kiírás: `Ki: A, '-', B`.

1.7. Mohó algoritmusok (greedy módszer)

A *greedy módszer*t (*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 annak alapján 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.

Ez az algoritmus a stratégia mohó jellegének következtében kapta 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. Ugyanakkor, a módszert olyankor is alkalmazhatjuk, amikor a feladat pontos megoldását csak exponenciális algoritmussal tudjuk megadni, de ilyenkor számításba vesszük, hogy az eredmény közelítő érték. Ilyenkor *heurisztikus* mohó algoritmusról beszélünk.

Legyen az L halmaz, amelyet az $\{a_1, a_2, ..., a_n\}$ sorozat tartalmaz, és T egy tulajdonság, amelyet az L részhalmazaira definiáltunk: T: $T(L) \rightarrow \{0, 1\}$, ahol $T(\emptyset) = 1$ (igaz, vagyis teljesül T). Ha $T(X) = 1$, akkor $\Rightarrow T(Y) = 1$, 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 adathalmazok 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.

1.7.1. A mohó algoritmus általános bemutatása

A módszer általános alakjának két változata ismeretes. A feladat megoldását az M halmaz tartalmazza, a megoldásokat az L – lehetséges megoldások halmazából – válogatjuk:

```
Algoritmus Greedy_1(L, M):
  M ← ∅
```

```
  Amíg M nem megoldás és L ≠ ∅ végezd el:
    Választ(L, x)                    { kiválasztjuk a legtöbbet ígérő elemet L-ből }
    L ← L\{x}                        { töröljük a legtöbbet ígérő elemet L-ből }
    Ha T(M ∪ {x}) = 1 akkor          { ha lehetséges x-et betenni a megoldásba }
      M ← M ∪ {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 ($\{a_1, a_2, ..., a_n\}$) egymás után vizsgáljuk és a követelményektől függően betesszük az eredménybe vagy sem (nincs szükség ezek törlésére L-ből, mivel egy megvizsgált elemhez nem térhetünk vissza, hiszen a rendezett sorozat elemeit rendre dolgozzuk fel). Az algoritmus ebben a változatban a következő:

```
Algoritmus Greedy_2(n, a, M):
  Feldolgoz(n, a)                    { ez a feldolgozás gyakran rendezés }
  M ← ∅
  i ← 1
  Amíg M nem megoldás és (i ≤ n) végezd el:
    Ha T(M ∪ {a_i}) = 1 akkor                              { ha lehetséges }
      M ← M ∪ {a_i}                                  { a_i-t hozzáadjuk M-hez }
    vége(ha)
    i ← 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.

1.7.2. Megoldott feladatok

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 ← 0                          { Bemeneti adatok: n, a. Kimeneti adatok: k, pozitívak }
  Minden i = 1, n végezd el:
    Ha a_i > 0 akkor
      k ← k + 1
      pozitívak_k ← a_i
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

2. Az átlagos várakozási idő minimalizálása

Egy ügyvédi irodába egyszerre érkezik 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: $M = (k_1, k_2, ..., k_n) \in \{(x_1, x_2, ..., x_n) \mid x_i \in \{1, 2, ..., n\}, x_i \neq x_j \; \forall \; i, j = 1, 2, ..., n, i \neq j\}$.

Az L eredetileg az $\{1, 2, ..., n\}$ 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. Ebben a megközelítésben 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.

Ettől eltérően, a következő algoritmus előbb inicializálja az M halmazt az 1, 2, ..., n sorszámokkal, és növekvő sorrendbe rendezi az idők $t_1, t_2, ..., t_n$ sorozatát, megfelelően módosítva az M halmaz elemeit. A rendezés után: $M = k_1, k_2, ..., k_n$ és $t_1 \leq t_2 \leq ... \leq t_n$. A kiírást az M halmazban található indexpermutáció alapján végezzük.

```
Algoritmus Sorrend(n, t, M, átlag):
                                  { Bemeneti adatok: n, t, M. Kimeneti adatok: átlag, M }
```

```
  Minden i = 1, n végezd el:
    M_i ← i
  vége(minden)
  növekvőSorrendbeRendezés(n, M, t)
                          { növekvően rendezzük a t sorozatot és módosítjuk az M-et is }
  minVárakozásiIdő ← 0
  várakozásiIdő ← 0
  Minden i = 1, n-1 végezd el:
    várakozásiIdő ← várakozásiIdő + t_i      { a t sorozat már növekvően rendezett }
    minVárakozásiIdő ← minVárakozásiIdő + várakozásiIdő
  vége(minden)
  átlag ← minVárakozásiIdő / n
Vége(algoritmus)
```

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 megállókat 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: $L = \{1, 2, \ldots, n\}$. Ismerjük az n megálló közötti $n - 1$ távolságot: $a_1, a_2, \ldots, a_{n-1}$.

Meg kell határoznunk azt a maximális elemszámú $M \subseteq L$ részhalmazt ($M = \{i_1, i_2, \ldots, i_k\}$), amelyben a sorszámok 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 ($a_{j+1} - a_j \geq x$, $j = 1, 2, \ldots, k - 1$).

```
Algoritmus Megállók(n, a, M):
  i ← 1                                  { Bemeneti adatok: n, a. Kimeneti adat: M }
  M_1 ← 1
  távAzUtolsótól ← 0          { az eredménybe betett utolsó megállótól mért távolság }
  Minden j = 2, n végezd el:
    Ha a_j-1 + távAzUtolsótól ≥ x akkor
      i ← i + 1
      M_i ← j
      távAzUtolsótól ← 0
    különben
      távAzUtolsótól ← távAzUtolsótól + 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.

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 sorszámait 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, $i = 1, 2, ..., n$, $a_i < b_i \leq 365$). Határozzuk meg a maximálisan kiszolgálható személyek számát és a bérbeadási időintervallumokat.

Megoldás

A következő algoritmusban $L = \{2, 3, ..., n\}$. 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 AutóKölcsönzés(n, a, b, max, M):
  növekvőSorrendbeRendezés(n, a, b)
  M1 ← 1                          { Bemeneti adatok: n, a, b. Kimeneti adat: max, M }
  max ← 1
  Minden i = 2, n végezd el:
    j ← Mmax
    Ha ai > bj akkor
      max ← max + 1
      Mmax ← i
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

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 $x = (x_1, \ldots, x_n)$ sorozat lesz, ahol $x_i \in [0, 1]$, $i = 1, 2, \ldots, n$ azt fejezi ki, hogy az i-edik árunak mekkora darabját csomagoljuk be. Ezen kívül: $súly_1 \cdot x_1 + súly_2 \cdot x_2 + \ldots + súly_n \cdot x_n \leq S$. Az optimális eredmény az, amely maximalizálja az $f(x) = érték_1 \cdot x_1 + érték_2 \cdot x_2 + \ldots + érték_n \cdot x_n$ függvényt.
Abban a sajátos esetben, amikor minden árut be lehet csomagolni a hátizsákba, $x = (1, 1, \ldots, 1)$. Ezért a továbbiakban feltételezzük, hogy $súly_1 + \ldots + súly_n > S$.

A greedy stratégiának megfelelően, az árukat az *erték*/*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, amely befér.

```
Algoritmus Hátizsák(n, S, súly, érték, sorszám, x):
  csökkenőSorrendbeRendezés(n, súly, érték, sorszám)
  Hely ← S                          { Hely a hátizsákban még szabad helyet jelöli }
  i ← 1
  Amíg (i ≤ n) és (Hely > 0) végezd el:
    Ha súly_i ≤ Hely akkor
      x_i ← 1
      Hely ← Hely - súly_i
    különben
      x_i ← Hely / súly_i
      Hely ← 0
      Minden j = i+1, n végezd el:
        x_j ← 0
      vége(minden)
    vége(ha)
    i ← i + 1
  vége(amíg)
Vége(algoritmus)
```

Az algoritmus végrehajtásának eredménye az x sorozat: $x = (1, \ldots, 1, x_j, 0, \ldots, 0)$ ahol $x_j \in [0, 1)$. Ennek alapján kiírhatjuk a becsomagolt áruk eredeti sorszámait és a hátizsák tartalmának értékét.

De most is, mint minden mohó algoritmus esetében, be kell bizonyítanunk, hogy az algoritmus optimális eredményt határoz meg.

2. fejezet

Objektumorientált programozás

2.1. Objektumorientált fogalmak

2.1.1. Adatvédelem moduláris programozással

Az eljárásközpontú programozás keretében a kódot igyekszünk 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 névtereken 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.

2.1. 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:

2.2. 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* (§2.1.3) 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 megtesszük 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 a 2.1. 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.

2.1.2. 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.

2.3. 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 a 2.3. 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.

A 2.3. 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.

2.1.3. 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 a §2.1.4. 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 függvényt* 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* (védett) 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, védett, illetve privát 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 (§2.2.2) 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 *konstruktornak* 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 a §2.1.5. 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.

2.1.4. 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 (§2.1.1) esetén ismertetett 2.1. kódszöveget módosítjuk úgy, hogy osztályokra vonatkozzon, majd ezt követően vizsgáljuk a tagokra való hivatkozást.

2.4. 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 a 2.4. 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 tagfüggvényeiben a rendszer egyszerűen elvégez egy helyettesítést, azaz minden *tag* helyett *this->tag* lesz. Például a 2.4. 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 megadjuk a fenti esetben a *this* mutatót, ezt automatikusan elhelyezi a rendszer. Mégis, a *this* mutatót explicit módon is használhatjuk, ha erre szükség van.

2.1.5. 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 családnevét és keresztnevét tárolja.

2.5. kódszöveg. A *szemely.h* fejállomány.

```
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 értékeket megadó aktuális paraméterekkel. Pontosabban, ha egy osztálynak van alapértelmezett konstruktora, akkor létrehozható olyan objektum, amely nem tartalmaz inicializáló aktuális 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 a 2.5. kódszöveg tartalmaz egy sajátos tagfüggvényt, a *destruktort*, melyet az objektumok megszűnésekor hív meg a rendszer.

Tekintsük a 2.5. kódszöveget felhasználó alábbi fő függvényt:

2.6. 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 tárban 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 alpértelmezett, 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 keresztnevet 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 megfelelteti 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 a 2.5. és 2.6. 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 szerettünk volna tenni, mivel így, ha az egyik objektum megszűnik, a másiknak is fel lesz szabadítva a memóriaterülete és fordítva. E helyett a másoló konstruktort terheltük túl, amely új memóriaterületet foglal le, és erre másolja a családnevet és keresztnevet.

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 kezdőértékadást ugyanolyan típusú objektummal végezni.

A 2.6. 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 helyezzünk 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 folyatatást. 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 paramétererei 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 a 2.5. 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.

2.7. 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, szemely& afeleseg);
10    hazaspar(char* cs_ferj, char* sz_ferj,
11       char* cs_feleseg, char* sz_feleseg):
12    ferj(cs_ferj, sz_ferj), feleseg(cs_feleseg, sz_feleseg)
13    {
14    }
15    void kiir();
16 };
17 inline hazaspar::hazaspar(szemely& aferj, szemely& afeleseg):
18 ferj(aferj), feleseg(afeleseg)
19 {
20 }
21 void hazaspar::kiir()
22 {
23    cout << "ferj:  ";
24    ferj.kiir();
25    cout << "feleseg:  ";
26    feleseg.kiir();
27 }
28 void main() {
29    szemely Ady("Ady","Endre");
30    szemely 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
feleseg:  Boncza Berta
ferj:  Petofi Sandor
feleseg:  Szendrei Julia
ferj:  Nincs adat
feleseg:  Nincs adat
```

A 2.7. 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 *feleseg* 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.

2.1.6. A destruktor

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

A destruktor neve mindig a $\sim$ karakterrel kezdődik, és ez után az osztály neve következik. A konstruktorhoz hasonlóan a destruktor 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 destruktorok. 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 destruktorban, mivel ez végtelen ciklust eredményezhet.

Egy helyi objektum destruktorát 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 tárban egy objektumot. Ezeket dinamikus módon létrehozott objektumoknak is nevezzük. Ekkor a destruktort 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 destruktor működését, amely minden esetben kiírja, hogy éppen mit végzett, azaz milyen konstruktort vagy destruktort hívott meg. A kiírást most a *printf* függvénnyel végezzük.

2.8. kódszöveg. A destruktor.

```
1  #include <cstdio>
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.

2.2. Az objektumorientált programozási módszer

2.2.1. 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 örökölheti 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.

2.2.2. 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 destruktorok 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.

2.2.3. 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.

2.9. 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?

A 2.9. 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 kötést virtuális tagfüggvények segítségével valósíthatjuk meg. Az $f1$ tagfüggvényt kell virtuálisnak deklarálni. Ezt úgy tehetjük meg, hogy a *virtual* minősítőt használjuk a függvény alaposzálybeli 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 nevezőjé 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.

2.10. 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 nevezo;
7   public:
8       tort(int szamlalo1 = 0, int nevezo1 = 1);
9       /*virtual*/ tort szorzat(tort& r);
10      tort& szoroz(tort& r);
11      void kiir();
12  };
13  tort::tort(int szamlalo1, int nevezo1)
14  {
15      szamlalo = szamlalo1;
16      nevezo = nevezo1;
17  }
18  // két tört szorzatát számolja ki, de nem egyszerüsít
19  tort tort::szorzat(tort& r)
20  {
21      return tort(szamlalo * r.szamlalo, nevezo * r.nevezo);
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 ( nevezo )
32         cout << szamlalo << " / " << nevezo;
33      else
34         cerr << "helytelen tort";
35  }
36  class tort_kiir:  public tort {
37  public:
38      tort_kiir( int szamlalo1 = 0, int nevezo1 = 1 );
39      tort szorzat( tort& r);
```

```
40  };
41  inline tort_kiir::tort_kiir(int szamlalo1, int nevezo1) :
42  tort(szamlalo1, nevezo1)
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 tehetjuk meg, hogy a 2.10. kódszöveg 9. sorából eltávolítjuk a megjegyzés jelét. Ha ezt megtesszük, 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.

2.2.4. 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 tagfügvényeket felül kell írni a származtatott osztályban, ellenkező esetben az illető osztály is absztrakt lesz.

Tekintsük a következő példát

2.11. 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.

2.2.5. 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 létrehozzuk.

2.12. kódszöveg. Absztrakt osztály, amely interfésznek tekinthető.

```
#include <iostream>
using namespace std;
class Jarmu
{
public:
    virtual void Indul() = 0;
    virtual void Megall() = 0;
    virtual void Megy(int km) = 0;
    virtual void All(int perc) = 0;
};
class Bicikli :  public Jarmu
{
public:
    void Indul();
    void Megall();
    void Megy(int km);
    void All(int perc);
};
void Bicikli::Indul() {
    cout << "Indul a bicikli." << endl;
}
void Bicikli::Megall() {
    cout << "Megall a bicikli." << endl;
}
void Bicikli::Megy(int km) {
    cout << "Biciklizik " << km << " kilometert." << endl;
}
void Bicikli::All(int perc) {
    cout << "A bicikli all " << perc << " percet." << endl;
}
class Auto :  public Jarmu
{
public:
    void Indul();
    void Megall();
    void Megy(int km);
    void All(int perc);
};
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.

2.3. Adatszerkezetek a C++-ban

A C++ programozási nyelv szabványos sablon könyvtára különböző beépített adatszerkezeteket biztosít a programozó számára. A szabványos sablon könyvtár *tárolókat* (*container*), *bejárókat* (*iterator*) és algoritmusokat tartalmaz. Az egyes tárolók egy adott adatszerkezetet valósítanak meg és objektumok tárolására alkalmasak. A bejáró általában egy külön osztály segítségével lesz megvalósítva és a tároló elemeit szolgáltatja. Amennyiben a tárolóban elhelyezett objektumok száma futási időben változhat *dinamikus tárolóról* beszélünk, ellenkező esetben pedig *statikus tárolóról*. A szabványos sablon könyvtár tárolói dinamikusak, a klasszikus tömbök viszont statikus tárolóknak tekinthetőek.

A C++ nyelvbeli tárolók közül a *sorozatok* (*sequence container*) és az *asszociatív tárolók* (*associative container*) a legfontosabbak. A szabvány nem írja elő, hogy milyen adatszerkezettel kell megvalósítani az egyes tárolókat, csak ennek interfészét rögzíti, illetve az egyes műveltek bonyolultságára vonatkozó kérdéseket adja meg. Ezekből azonban általában következtetni lehet arra, hogy milyen adatszerkezettel valósították meg az illető tárolót. A leggyakrabban használt sorozatok a *vector*, a *list* és a *deque*. Ezeknek az a jellegzetessége, hogy objektumokat tárolnak, de anélkül, hogy az adatszerkezet valamilyen kulcs szerint rendezve

volna. Az asszociatív tárolók mindig rendezett adatszerkezeteket valósítanak meg. Ezek közül az alábbiakat említjük meg: *set*, *multiset*, *map*, *multimap*.

A vektor (vector) tároló, a hagyományos tömbökhöz hasonlóan, *direkt hozzáférést* (*random access*) biztosít a tárolt objektumokhoz. Ez azt jelenti, hogy egy adott index alapján az egyes elemek gyorsan elérhetőek. Azonban a hozzáadási és törlési műveletek csak a tároló végén végezhetőek el valós időben, a beszúrás egy nagyméretű vektor esetén már lassú lesz. Ezekből arra következtethetünk, hogy az adatszerkezetünk egy hagyományos tömbre alapszik, viszont hangsúlyozzuk, hogy ebben az esetben dinamikus tárolóról van szó.

A lista (list) esetén a hozzáadási és törlési műveletek bárhol hatékonyan elvégezhetőek, viszont nem marad meg a direkt hozzáférést. A tárolót általában egy kétszeresen láncolt listával ábrázoljuk. Amennyiben egyszeresen láncolt listát szeretnénk használni, akkor a *forward_list* sorozattal kell dolgoznunk, melyet a C++ 11-től kezdődően vezettek be. A továbbiakban egy példán keresztül mutatjuk be a list adatszerkezet használatát.

2.13. kódszöveg. Példa a list adatszerkezetre.

```
#include<iostream>
#include<string>
#include<list>
using namespace std;

class Alakzat {
protected:
        string nev;
public:
        Alakzat(string nev) {
                this->nev = nev;
        }
        virtual void kiir() {
                cout << "nev = " << nev << endl;
        }
};

class SzinesAlakzat : public Alakzat {
protected:
        string szin;
public:
        SzinesAlakzat(string nev, string szin) : Alakzat(nev) {
                this->szin = szin;
        }
        void kiir() {
                cout << "nev = " << nev;
                cout << "\tszin = " << szin << endl;
        }
};

class AlakzatLista {
        list<Alakzat*> alakzatok;
public:
        void add(Alakzat* alakzat) {
```

```
35                  alakzatok.push_back(alakzat);
36          }
37          void kiir() {
38                  for (list<Alakzat*>::iterator it = alakzatok.begin();
39                          it != alakzatok.end(); ++it) {
40                                  (*it)->kiir();
41                  }
42          }
43  };
44
45  int main() {
46          AlakzatLista lista;
47          Alakzat *negyzet = new Alakzat("negyzet");
48          lista.add(negyzet);
49          Alakzat *teglalap = new SzinesAlakzat("teglalap", "zold");
50          lista.add(teglalap);
51          lista.kiir();
52          delete negyzet;
53          delete teglalap;
54  }
```

A fenti kód egy *Alakzat* típusú objektumokból álló listát hoz létre, majd kiírja annak tartalmát. Figyeljük meg, hogy az *Alakzat* osztály származtatottjait is tárolhatjuk a listában és ebben az esetben a kiíráskor a leszármazott osztály sajátos tulajdonságai is megjelennek. A *SzinesAlakzat* osztály objektumának kiírásakor a *kiir* függvény virtualitása biztosítja azt, hogy nem csak a név, hanem a szín is ki lesz írva a szabványos kimenetre.

3. fejezet Adatbázisok

3.1. A relációs adatmodell

Az első ABKR-ek hálós vagy hierarchikus adatmodellt használták. 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, K, D_n nem szükségszerűen egymást kizáró halmazok, akkor *R* egy reláció a D_1, D_2, K, D_n halmazokon, ha $R \subseteq D_1 \times D_2 \times \text{K} \times D_n$ (Descartes-féle szorzat).

A relációs adatmodell szempontjából D_i az A_i attribútum értékeinek tartománya (doméniuma). D_i lehet egész számok halmaza, karaktersorok 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}
⋮					
r_i	a_{i1}	...	a_{ij}	...	a_{in}
⋮					
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,..., A_n). A reláció nevét és a reláció attribútumainak a halmazát együtt *relációsémának* nevezzük.

példa: Diákok reláció:

Név	***SzületésiDátum***	***CsopKod***
Nagy Ödön	1975-DEC-13	512
Kiss Csaba	1971-APR-20	541
Papp József	1973-JAN-6	521

példa: Könyvek reláció:

Szerző	***Cím***	***Kiadó***	***KiadÉv***
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 *kulcsjelölt*ről beszélhetünk. Ezek közül egyet *elsődleges kulcs*nak 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.

3.2. Normalizálás

3.2.1. Funkcionális függőségek

Legyen egy reláció
$R(A_1, A_2, ..., A_n)$, ahol A_i attribútumok.
Jelöljük az attribútumok halmazát
$A = \{A_1, A_2, ..., A_n\}$.

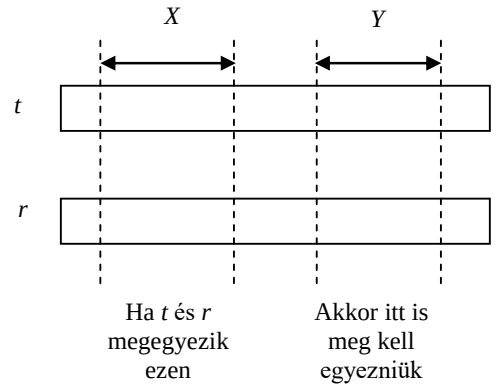
Legyenek *X* és *Y* az *R* reláció attribútumhalmazának részhalmazai, vagyis $X, Y \subset A$. Ezeket a jelöléseket használjuk a továbbiakban, ha esetleg nem ismételjük meg.

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 fomálisan $X \rightarrow Y$-nal jelöljük.

Relációs algebrai műveletek segítségével a következőképpen értelmezhetjük a funkcionális függőséget:

$X \to Y$, 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 $\to$ SzállNév

SzállID $\to$ SzállCím.

Mivel mindkét függőségnek ugyanaz a bal oldala, SzállID, ezért egy sorban összegezhetjük:

SzállID $\to$ {SzállNév, SzállCím}

Szavakban, ha két sorban ugyanaz a SzállID értéke, akkor a SzállNév értéke is ugyanaz kell legyen, illetve a SzállCím értéke is.

Ezenkívül:

ÁruID $\to$ ÁruNév

ÁruID $\to$ MértEgys (azzal a feltevéssel, ha más mértékegységben árulják az árut, más ID-t is kap).

Hasonlóan egy sorban:

ÁruID $\to$ {ÁruNév, MértEgys}

A funkcionális függőséget felhasználva adhatunk még egy értelmezést a reláció kulcsának. Egy vagy több attribútumból álló $\{C_1, C_2, \text{K}, C_k\}$ 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, K, C_k-n megegyezne.
- Nincs olyan valódi részhalmaza $\{C_1, C_2, \text{K}, C_k\}$-nak, amely funkcionálisan meghatározná az *R* összes többi attribútumát, azaz a kulcsnak minimálisnak kell lennie.

példa: a SzállításiInformációk reláció kulcsa a {SzállID, 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. □

*Szuperkulcs*oknak nevezzük azon attribútumhalmazokat, melyek tartalmaznak 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,..., A_n) reláció esetén A_i attribútum *prím*, ha létezik egy C kulcsa az R-nek, úgy hogy $A_i \in C$. Ha egy attribútum nem része egy kulcsnak, akkor *nem prím* attribútumnak nevezzük.

*Triviális funkcionális függőség*ről beszélünk, ha az Y attribútum halmaz részhalmaza az X attribútum halmaznak $(Y \subset X)$, akkor Y attribútum halmaz funkcionálisan függ X attribútum halmaztól ($X \to Y$).

példa: Triviális funkcionális függőség**:** {SzállID, ÁruID} $\to$ SzállID. □

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 $X_1X_2\text{K}\ X_p \to Y_1Y_2\text{K}\ Y_s$ funkcionális függőség, ha az Y-ok közül legalább egy különbözik az X-ektől, vagyis

$$\exists Y_j, j \in [1,s]\ j \in \{1,2,...,s\} \text{ úgy, hogy } Y_j \neq X_k, \forall k \in \{1,2,...,p\}.$$

Teljesen nem triviális egy $X_1X_2\text{K}\ X_p \to Y_1Y_2\text{K}\ Y_s$ funkcionális függőség, ha az Y-ok közül egy sem egyezik meg az X-ek valamelyikével, vagyis

$\forall Y_j, j \in [1,s]\ j \in \{1,2,...,s\}$ -re $Y_j \neq X_k, \forall\ k \in \{1,2,...,p\}$.

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 $Y \to B$-t egy parciális függőség. (B függ a kulcs egy részétől.)

példa: parciális függőségre: SzállID $\to$ SzállNév. □

A SzállításiInformációk relációban {SzállID, ÁruID} a kulcs, tehát

{SzállID, ÁruID} $\to$ 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 $Y \to B$ 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 $C \to Y$ és $Y \to B$, és erre mondhatjuk, hogy B tranzitív függőséggel függ C-től.

példa: Rendelések (<u>RendelésSzám</u>, 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 $\to$ VevőID.

Ezenkívül fennáll a

VevőID $\to$ VevőNév

funkcionális függőség. Tehát a VevőNév tranzitív függőséggel függ a RendelésSzámtól.

Funkcionális függőségek tulajdonságai:

1. Ha C az $R[A_1, A_2,..., A_n]$ reláció egy kulcsa, akkor $C \to \beta, \forall \beta \subset \{A_1, A_2,..., A_n\}$.
2. Ha $\beta \subseteq \alpha$, akkor $\alpha \to \beta$, ez a **triviális** funkcionális függőség vagy reflexivitás.

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\beta \subset \alpha}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \alpha \to \beta$$

3. Ha $\alpha \to \beta$, akkor $\gamma \to \beta, \forall \gamma$ ahol $\alpha \subset \gamma$.

$$\Pi_\gamma(r_1) = \Pi_\gamma(r_2) \underset{\alpha \subset \gamma}{\Rightarrow} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \to \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \gamma \to \beta$$

4. Ha $\alpha \to \beta$ és $\beta \to \gamma$, akkor $\alpha \to \gamma$, ez a funkcionális függőség **tranzitív** tulajdonsága.

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha\to\beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \underset{\beta\to\gamma}{\Rightarrow} \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow \alpha \to \gamma$$

5. Ha $\alpha \to \beta$ és $\gamma \subset A$, akkor $\alpha\gamma \to \beta\gamma$, ahol $\alpha\gamma = \alpha \cup \gamma$.

$$\Pi_{\alpha\gamma}(r_1) = \Pi_{\alpha\gamma}(r_2) \Rightarrow \left| \begin{array}{c} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \end{array} \right. \Rightarrow \Pi_{\beta\gamma}(r_1) = \Pi_{\beta\gamma}(r_2)$$

***Problémák*:**

Azokat a problémákat, amelyek akkor jelennek meg, amikor túl sok információt probálunk egyetlen relációba belegyömöszölni, *anomália*nak 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 töltsük fel a kapott két új reláció sorait az *R* soraiból.

Legyen egy *R* reláció $\{A_1, A_2, \text{K}, A_n\}$ sémával, *R*-et felbonthatjuk *S* és *T* két relációra, amelyeknek sémái $\{B_1, B_2, \text{K}, B_m\}$, illetve $\{C_1, C_2, ..., C_k\}$ úgy, hogy

1. $\{A_1, A_2, \text{K}, A_n\} = \{B_1, B_2, \text{K}, B_m\} \cup \{C_1, C_2, \text{K}, C_k\}$, ahol

$\{B_1, B_2, \text{K}, B_m\} \cap \{C_1, C_2, ..., C_k\} \neq \emptyset$.

2. Az *S* reláció sorai az *R*-ben szereplő összes sornak a $\{B_1, B_2, \text{K}, B_m\}$-re vett vetületei, azaz *R* aktuális előfordulásának minden egyes *t* sorára vesszük a *t* azon komponenseit, amelyek a $\{B_1, B_2, \text{K}, B_m\}$ 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 $\{C_1, C_2, ..., C_k\}$ attribútumok halmazára vett projekciói.

2. $S = \pi_{B_1, B_2, \text{K}, B_m}(R)$; $T = \pi_{C_1, C_2, \text{K}, C_m}(R)$;

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.

3.2.2. 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 skatulyázottak. 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 normálforma (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 tegyen, hogy a létező relációs ABKR-ek kezelni tudják.

Példa: A következő reláció nincs 1NF-ben:

Alkalmazottak:

SzemSzám	Név	Cím			Gyerek1	SzülDát1	…	Gyerek5	SzülDát5
		Helység	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 keresztnevében 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, ezeknél 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 normálformára 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,..., A_n) reláció, mely nincs első normálformában, mivel ismétlődő attribútumokat tartalmaz, felbontással első normálformába hozható. Jelöljük az attribútumok halmazát

$A = \{A_1, A_2, ..., A_n\}$.

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 $A = C \cup I_1 \cup I_2 \cup \ldots \cup I_k \cup J$. A felbontás után kapjuk a következő két relációsémát:

$S(C, I)$ 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 = {SzemSzám}
I = {GyerekNév, SzülDátum}
J = {Név, Helység, Utca, 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 ugyannak 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 normálforma (2NF)

Értelmezés: Egy reláció 2NF formában van, ha első normálformájú (1NF) és nem tartalmaz $Y \to B$ alakú parciális függőséget, ahol B nem prím 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 $\to$ 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 $A = \{A_1, A_2, ..., A_n\}$ és $C \subset A$ egy kulcs. Ha a reláció nincs második normálformában, azt jelenti létezik egy $B \subset A$ nem kulcs $B \cap C = \varnothing$ attribútumhalmaz, mely függ funkcionálisan a kulcs egy részétől, vagyis létezik $D \subset C$, úgy hogy $D \to B$.

Az R relációt felbontjuk két relációra, melyek sémái:

$T(D, B)$ és $S(A - B)$

példa: Amint láttuk a 0. példa SzállításiInformációk relációjában fennállnak a:

SzállID $\to$ {SzállNév, SzállCím}

ÁruID $\to$ {ÁruNév, MértEgys}

funkcionális függőségek, a kulcs pedig a C ={SzállID, ÁruID}.

Első lépésben B = {SzállNév, SzállCím}, D = {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

ÁruID $\to$ {ÁruNév, MértEgys}.

Ebben az esetben B = {ÁruNév, MértEgys}, D = {ÁruID}. Tovább bontjuk a következő két relációra:

```
Áruk (ÁruID, ÁruNév, MértEgys),
Szállít (SzállID, ÁruID, 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:

SzállID	***SzállNév***	***SzállCím***
111	Rolicom	A. Iancu 15
222	Sorex	22 dec. 6

Áruk:

ÁruID	***ÁruNév***	***MértEgys***
45	Milka csoki	tábla

67	Heidi csoki	tábla
56	Milky way	rúd

Szállít:

SzállID	*ÁruID*	*Ár*
111	45	25000
222	45	26500
111	67	17000
111	56	20000
222	67	18000
222	56	22500

Harmadik normálforma (3NF)

Értelmezés: Egy *R* reláció *harmadik normálformában* (3NF) van, ha második normálformában van és nem tartalmaz $Y \to B$ alakú *tranzitív funkcionális* függőséget, ahol *B* nem prím attribútum.
Értelmezés: Egy *R* reláció *harmadik normálformában* (3NF) van, ha létezik az *R*-ben egy $Y \to B$ alakú nem triviális funkcionális függőség, akkor *Y* az *R* reláció szuperkulcsa vagy a *B* prím attribútum (valamelyik kulcsnak része).

A két értelmezés ekvivalens. A második nem kéri a második normálformát, de mivel bármely létező $Y \to B$ funkcionális függőség esetén a bal oldal szuperkulcs, nem lehet annak része. Tehát elég, ha az összes létező funkcionális függőség esetén a bal oldal szuperkulcs, akkor a tranzitív függőség nem létezhet, mert a tranzitív függőség esetén a bal oldal nem kulcs és ez nem megengedett.

példa: A Rendelések reláció nincs 3NF-ben, mivel tartalmaz tranzitív funkcionális függőséget.

RendelésSzám $\to$ VevőID

VevőID $\to$ VevőNév.

Probléma, ha így ábrázoljuk a rendeléseket, hogy ha egy vevő több rendelést is elhelyez, ami lehetséges, akkor a vevő nevét ismételjük. Megoldás: 2 relációra bontjuk a relációt, mely nincs 3NF-ben. □

3NF-re alakítás

Legyen *R* egy reláció, mely 2NF-ben van, viszont nincs 3NF-ben, attribútumainak a halmaza $A = \{A_1, A_2, ..., A_n\}$ és $C \subset A$ elsődleges kulcs. Ha a reláció nincs harmadik normálformában, azt jelenti, hogy létezik egy $B \subset A$ nem kulcs $B \cap C = \varnothing$ attribútumhalmaz, mely tranzitív függőséggel függ a kulcstól, vagyis létezik *D*, úgy hogy $C \to D$ és $D \to B$. 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:

$T(D, B)$ és $S(A - B)$.

példa: A Rendelések reláció esetén: *B* = {VevőNév}, *D* = {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.

3.3. 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:

$$\text{<attribútum\_név>} \left\{ \begin{array}{c} = \\ <> \\ < \\ <= \\ > \\ >= \end{array} \right\} \left\{ \begin{array}{c} \text{<attribútum\_név>} \\ \text{<konstans>} \end{array} \right\}$$

$$\left\{ \begin{array}{c} \text{<attribútum\_név>} \\ \text{<konstans>} \end{array} \right\} \left\{ \begin{array}{c} \text{IS IN} \\ \text{IS NOT IN} \end{array} \right\} \text{<reláció> (melynek egy attribútuma van)}$$

NOT <feltétel>

$$\text{<feltétel>} \left\{ \begin{array}{c} \text{OR} \\ \text{AND} \end{array} \right\} \text{<feltétel>}$$

A továbbiaban 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:

$\sigma_{\text{Fizetés <= 500}}$ (Alkalmazottak)

A lekérdezés eredménye:

SzemSzám	***Név***	***RészlegID***	***Fizetés***
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: $\sigma_{\text{Fizetés > 500 AND RészlegID = 9}}$ (Alkalmazottak)

Az eredmény:

SzemSzám	***Név***	***RészlegID***	***Fizetés***
456777	Szabó János	9	900

2) *Vetítés (Projection)*: Adott *R* egy reláció A_1, A_2,..., A_n attribútumokkal. A vetítés művelet eredményeként olyan relációt kapunk, mely *R*-nek csak bizonyos attribútumait tartalmazza. Ha kiválasztunk *k* attribútumot az *n*-ből: $A_{i_1}, A_{i_2}, \text{K}, A_{i_k}$ -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: $\pi_{A_{i_1}, A_{i_2}, \text{K}, A_{i_k}}(R)$

példa: Ha az Alkalmazottak relációból csak az alkalmazott neve és fizetése érdekel, akkor a következő művelet eredménye a kért reláció:

$\pi_{\text{Név, Fizetés}}$ (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:

A	***B***
12	33
24	46

Legyen R_2 reláció:

B	***C***	***D***
20	55	80
30	67	97
40	75	99

Akkor $R_1 \times R_2$ eredménye:

A	***R_1.B***	***R_2.B***	***C***	***D***
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: $R_1 - R_2$).

példa: Legyen R_1:

SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500

és legyen R_2:

SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
111111	Nagy Éva	2	300
456777	Szabó János	9	900
123444	Vincze Ildikó	1	800

Ekkor R_1 U R_2:

SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
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:

SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
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 kifejezhetőek.

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:

A	*B*	*C*
11	23	32
65	76	82
97	76	82

R_2 reláció:

B	*C*	*D*
23	32	44
23	32	57
76	82	99

$R_1 \bowtie_{A<D} R_2$:

A	$R_1.B$	$R_1.C$	$R_2.B$	$R_2.C$	*D*
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: $B \cap C = \{A_1, A_2, \ldots, A_p\}$. A természetes összekapcsolást a következő képlettel fejezhetjük ki:

$$R_1 \bowtie R_2 = \pi_{B \cup C}(R_1 \bowtie_{(R_1.A_1=R_2.A_1)\wedge(R_1.A_2=R_2.A_2)\wedge K\wedge(R_1.A_p=R_2.A_p)} R_2,$$

ahol $R_i.A_j$ jelöli az A_j attribútumot az R_i relációból, $i \in \{1,2\}$, $j \in \{1,2, \ldots, p\}$.

példa: Legyenek R_1 és R_2 relációk a Théta-összekapcsolás példából, a természetes összekapcsolás eredménye:

$R_1 \bowtie R_2$ eredménye:

A	*B*	*C*	*D*
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 $A_1, A_2, \ldots, A_p$ közös attribútumon. Ha az r_1 és r_2 sorok párosítása sikeres, akkor a párosítás eredményét *összekapcsolt sor*nak 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 nevezzünk

példa: Legyenek a Szállítók, Áruk és Szállít relációk. Ha az összes szállítási információra van szükségünk, akkor kiszámítjuk a Szállít ⋈ Szállítók ⋈ Áruk természetes összekapcsolást, melynek eredménye:

Szállítók:

SzállID	*SzállNév*	*SzállCím*
111	Rolicom	A.Iancu 15
222	Sorex	22 dec. 6

Áruk:

ÁruID	*ÁruNév*	*MértEgys*
45	Milka csoki	tábla
67	Heidi csoki	tábla
56	Milky way	Rúd

Szállít:

SzállID	*ÁruID*	*Ár*
111	45	25000
222	45	26500
111	67	17000
111	56	20000
222	67	18000
222	56	22500

Szállít ⋈ Szállítók ⋈ Áruk eredménye:

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

Relációs algebrai műveletek alkalmazásával újabb relációkat kapunk. Gyakran szükséges egy olyan operátor, amelyik átnevezi a relációkat.

9) *Átnevezés:* Legyen $R(A_1, A_2, \ldots, A_n)$ egy reláció, az átnevezés operátor:

$\rho_{S(B_1,B_2,K,B_n)}(R)$ az R relációt S relációvá nevezi át, az attribútumokat pedig balról jobbra B_1, B_2, …, B_n-né. Ha az attribútum neveket nem akarjuk megváltoztatni, akkor $\rho_S(R)$ operátort használunk.

10) *Hányados (Quotient)*: Legyen R_1 reláció sémája: $\{X_1, X_2,\ldots, X_m, Y_1,Y_2,\ldots,Y_n\}$, R_2 reláció sémája pedig: $\{Y_1, Y_2, \ldots, Y_n\}$, tehát $Y_1, Y_2, \ldots,Y_n$ közös attribútumok ugyanazon értékhalmazzal, és R_1-nek még van pluszba m attribútuma: $X_1, X_2,\ldots, X_m$, R_2-nek pedig a közöseken kívül nincs más attribútuma. R_1 az osztandó, R_2 az osztó. Jelöljük X-szel és Y-nal a következő attribútumhalmazokat: $X = \{X_1, X_2,\ldots, X_m\}$, $Y = \{Y_1,Y_2,\ldots,Y_n\}$. Ebben az esetben jelöljük: R_1 (X, Y), R_2 (Y) a két relációt, melynek hányadosát jelöljük:

R_1 DIVIDE BY R_2 (X)-el

Tehát a hányados reláció sémája $\{X_1, X_2,\ldots, X_m\}$. 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 = $\pi_{\text{SzállID, ÁruID}}(\text{Szállít})$ és a következő sorok az S relációban:

SzállID	***ÁruID***
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:

ÁruID
A1

akkor az S DIVIDE A(SzállID) eredménye:

SzállID
S1
S2

b) esetben A reláció:

ÁruID
A2
A4

akkor S DIVIDE A(SzállID):

SzállID
S1
S4

c) esetben A reláció:

ÁruID
A1
A2
A3
A4
A5
A6

akkor S DIVIDE A(SzállID):

SzállID
S1

3.4. 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.

3.4.1. 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 euró”.

```
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

$$\pi_{A_{i_1},A_{i_2},K,A_{i_k}}(R)$$

a SELECT-SQL parancs segítségével a következőképpen adható meg:

```
SELECT  A_i1 , A_i2 , K , A_ik
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:

$$\pi_{\text{Név, Fizetés}}(\text{Alkalmazottak})$$

SELECT-SQL parancs segítségével a következőképpen írható fel:

```
SELECT Név, Fizetés
FROM Alkalmazottak; □
```

A lekérdezés feldogozása 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 feldogozás 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 arány 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 *egyszerű* vagy *összetett*. Összetett feltétel esetén 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. □

Az SQL rendszerek háromértékű logikát használnak, vagyis egy kifejezés (feltétel) logikai értéke lehet: **igaz (1), hamis (0), ismeretlen (unknown) (0.5)**. Egy kifejezés logikai értéke akkor ismeretlen, ha a kifejezésben szereplő valamelyik operandus értéke NULL. Egy WHERE-beli állítás értékét hamisnak tekintjük akkor is, ha a kifejezés értéke „ismeretlen”. A NOT, AND és OR operátorok igazságértékét a következő táblázat adja meg:

AND	**FALSE**	**NULL**	**TRUE**
FALSE	FALSE	FALSE	FALSE
NULL	FALSE	NULL	TRUE
TRUE	FALSE	NULL	TRUE

OR	**FALSE**	**NULL**	**TRUE**
FALSE	FALSE	NULL	TRUE
NULL	NULL	NULL	TRUE
TRUE	TRUE	TRUE	NULL

NOT	**FALSE**	**NULL**	**TRUE**
	TRUE	NULL	FALSE

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 vezetéknevű alkalmazottat, de a 'Kovácsovics' vezetéknevűt is, ha ilyen létezik az adatbázisban. Ha csak a Kovács vezetéknevűeket 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; □
```

3.4.2. 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 θ;
```

A leggyakrabban használt műveletet, a természetes összekapcsolást

$$R_1 \bowtie R_2 = \pi_{B \cup C}(R_1 \bowtie_{(R_1.A_1=R_2.A_1)\wedge(R_1.A_2=R_2.A_2)\wedge \mathrm{K} \wedge(R_1.A_p=R_2.A_p)} R_2),$$

a következőképpen írhatjuk SQL-ben:

```
SELECT *
FROM R1, R2
WHERE R1.A1 = R2.A1 AND R1.A2 = R2.A2 AND K 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; □
```

Kettőnél több 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 minumum 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, Szállít, Szállítók
WHERE ÁruCsoportok.CsopID = Áruk.CsopID
      AND Áruk.ÁruID = Szállít.ÁruID
      AND Szállít.SzállID = Szállítók.SzállID
      AND ÁruCsoportok.Név = 'édesség'; □
```

Az SQL lehetőséget ad arra, hogy a FROM záradékban szereplő R relációhoz hozzárendeljünk egy másodnevet, melyet *sorváltozó*nak 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 használjuk.

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 <> 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; □
```

Algoritmus egy egyszerű SELECT−SQL lekérdezés kiértékelésére:
Input: R_1, R_2,…, 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, …, 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.SzemSzám = Tanárok.SzemSzám); □
```

3.4.3. 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 CsopKod
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 paranccsal 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 használjuk.

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. □

3.4.4. 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. A `COUNT()` összesítő függvénynek több formája is van:

- `COUNT(*)` - az eredmény-reláció kardinalitását (az összes sor számát) adja vissza
- `COUNT(oszlop_név)`- azon sorok számát adja vissza, ahol oszlop_név értéke NULL-tól különböző érték
- `COUNT(DISTINCT oszlop_név)`- megszámolja, hány különböző értéke van az oszlop_név mezőnek.

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:

SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
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:

RészlegID	*AVG(Fizetés)*	*MIN(Fizetés)*	*MAX(Fizetés)*	*SUM(Fizetés)*
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:

ÁtlagFizetés
800
500
650

példa: Legyen a Szállít (SzállID, ÁruID, Ár) reláció. Egy árut több szállító is ajánlhatja, különböző árban. Sok esetben szükségünk van az átlagárra, amiben ajánlanak egy árut. A következő lekérdezés minden áru esetén meghatározza az átlagárat, 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 feltételét 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. ÁruID = Áruk.ÁruID
WHERE Szállít, Áruk
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:

Tanár Kod	*Név*	*Cím*	*PhD*	*Beosztás Kod*	*Tanszék Kod*	*Fizetés*
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	Váczi u. 52	Y	PRO	ALG	280

a lekérdezés eredménye:

Tanszék Kod	*Beosztás Kod*	*AVG (Fizetés)*
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:

RészlegID	*AVG(Fizetés)*
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 tanársegédeket kivéve a tanárok átlagfizetése nagyobb, mint 240 euró.

```
SELECT TanszékKod, AVG(Fizetés)
FROM Tanárok
WHERE BeosztásKod <> 'TNS'
GROUP BY TanszékKod
HAVING AVG(Fizetés) > 240;
```

3.4.5. Alkérdések

A WHERE záradékban eddig a feltételben skaláris értékeket tudtunk összehasonlítani. Az alkérdések segítségével sorokat vagy relációkat tudunk összehasonlítani. Egy alkérdés egy olyan kifejezés, mely egy relációt eredményez, például egy select-from-where kifejezés.

Alkérdést 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 alkérdést hajtja végre és annak eredményét használja a „fő” lekérdezés, kivéve a korrelált alkérdéseket.

Alkérdéseket 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. *többsoros alkérdés* (multiple-row subquery);
- több oszlopot, több sort, ún. *több oszlopos alkérdés* (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 konstanst egy attribútummal vagy egy másik konstanssal összehasonlíthatjuk. Nagyon fontos, hogy az alkérdés 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 SzemSzám =
4)           (SELECT ManSzemSzám
```

```
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 0. 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 SzemSzám = 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 SzemSzám = 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 bánjunk a csoportosítással.

Példa: Egy *helytelen* SELECT parancs:

```
SELECT SzemSzám, 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 *többsoros alkérdések* esetén a WHERE záradék feltétele olyan *operátorok*at 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. SzemSzám = Managerek.SzemSzám
        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, UtcaSzám
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 > 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 kodja mellett a fizetést is, az értékpárt.

3.4.6. 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 sorváltozóval 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 értéketől. 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 diákot, aki egyedül van a csoportjában 10-es átlaggal. □

3.4.7. 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:

SzemSzám	***Név***	***RészlegID***	***Fizetés***
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

RészlegID	***RNév***	***ManagerSzemSzám***
1	Tervezés	123444
2	Könyvelés	234555
3	Eladás	NULL
9	Beszerzés	456777

Legyen a következő lekérdezés:

```
SELECT * FROM Alkalmazottak
INNER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

Az eredmény:

SzemSzám	*Név*	*RészlegID*	*Fizetés*	*RNév*	*ManagerSzemSzám*
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

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: Tekintsük az alábbi lekérdezést:

```
SELECT * FROM Alkalmazottak
LEFT OUTER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

A lekérdezés eredménye:

SzemSzám	*Név*	*RészlegID*	*Fizetés*	*RNév*	*Manager SzemSzám*
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
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: Tekintsük az alábbi lekérdezést:

```
SELECT * FROM Alkalmazottak
RIGHT OUTER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

A lekérdezés eredménye:

SzemSzám	*Név*	*RészlegID*	*Fizetés*	*RNév*	*Manager SzemSzám*
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: Tekintsük az alábbi lekérdezést:

```
SELECT * FROM Alkalmazottak
FULL OUTER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

A lekérdezés eredménye:

SzemSzám	*Név*	*RészlegID*	*Fizetés*	*RNév*	*ManagerSzemSzám*
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
567765	Katona József	NULL	600	NULL	NULL
556789	Lukács Anna	NULL	700	NULL	NULL
NULL	NULL	3	NULL	Eladás	NULL

Csoportosítás esetén is használhatóak a külső összekapcsolási műveletek. □

példa: „Adjuk meg minden részleg esetén az ott dolgozó alkalmazottak számát! Írassuk ki azon részlegeket is, amelyekhez egyetlen alkalmazott sincs hozzárendelve!”

```
SELECT Részlegek.RészlegID, COUNT(SzemSzám) as AlkalmazottSzám
FROM Részlegek
LEFT OUTER JOIN Alkalmazottak
  ON Alkalmazottak.RészlegID = Részlegek.RészlegID
GROUP BY Részlegek.RészlegID;
```

A lekérdezés eredménye:

RészlegID	*Alkalmazott Szám*
1	1
2	3
3	0
9	2

Példafeladatok

1. i) Tervezzünk **relációs adatbázissémát**, melynek táblái **3NF**-ban vannak és egy software cég következő információit tárolják:

- **tevékenységek**: tevékenység kódja, leírása, tevékenység típusa;
- **alkalmazottak**: alkalmazott kódja, nev, 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! Írjuk fel a funkcionális függőségeket!
ii) Relációs algebrát vagy SELECT-SQL parancsot használva, az **i)** pont adatbázisára vonatkozóan adjuk meg:

a) azokat az alkalmazottakat a nevükkel, akik dolgoznak legalább egy “tervezés” típusú tevékenységnél és **nem** dolgoznak egyetlen “tesztelés” típusú tevékenységnél sem;
b) azokat az alkalmazottakat a nevükkel, akik olyan csoportok vezetői, amelyekhez legalább 10 alkalmazott tartozik!

2. i) Tervezzünk **relációs adatbázissémát**, melynek táblái **3NF**-ban vannak és a következő információikat tárolják:

- **tantárgyak**: tantárgy kódja, megnevezése, kreditek száma;
- **diákok**: diák kódja, neve, születési dátuma, csoportjának kódja, évfolyamra, szakra vonatkozó információk, azon tantárgyak listája, amelyekből vizsgázott (a vizsga dátuma és a jegy is tárolandó)!

Indokoljuk, hogy a táblák **3NF**-ban vannak! Írjuk fel a funkcionális függőségeket!
ii) Relációs algebrát vagy SELECT-SQL parancsot használva, az **i)** pont adatbázisára vonatkozóan adjuk meg:

a) azokat a tantárgyakat a megnevezésükkel, amelyek esetén nincsenek átmenő jegyek (átmenő jegy>=5);
b) 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 adatbázissémát, 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-szükségletek listája (pl.: 1Gb RAM, 512Mb RAM, DVD Reader stb.). Írjuk fel a funkcionális függőségeket, és 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!

4. i) Tervezzünk **relációs adatbázissémát**, melynek táblái **3NF**-ban vannak és filmekről szóló információikat tárolnak:

- **színészek**: színész kódja, neve, neme, weboldala, országa;
- **filmek**: film kódja, címe, megjelenési dátuma, stúdió neve, stúdió weboldala, stúdió országa, rendező neve, rendező weboldala, rendező országa, színészek listája, film típusainak listája!

Írjuk fel a létező funkcionális függőségeket, és 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!

5. Adjunk példát az R(ABCD) reláció olyan soraira, melyekben az ABC→D funkcionális függőség nincs betartva!

6. Legyen R és S két reláció a következő sorokkal:

R:

A	B	C
1	2	2
3	4	5

S:

D	E
1	2
3	6
3	5

Mi lesz a következő lekérdezés eredménye:

$$\rho((D\rightarrow A, E\rightarrow C), S) - \pi_{A;C}(R)?$$

7. Legyen a következő reláció: *Személyek(Kód, Név, SzületésiDátum, Város, Szakma)*, ahol a *Kód* mező a reláció elsődleges kulcsa; és a következő lekérdezés:

$$\pi_{\text{Város}} (\sigma_{\text{Szakma='Programozó'}} (\text{Személyek})).$$

Írjuk le szavakban, mi lesz a lekérdezés eredménye, majd adjuk meg a lekérdezést SQL parancs segítségével!

8. Legyen az *R* reláció szerkezete: *R(a, b)*. A Q_1 és Q_2 lekérdezések eredményei a `SELECT * FROM R` parancs által visszatérített sorok lesznek.

Q_1:
```
UPDATE R SET b = 10 WHERE a = 20;
SELECT * FROM R;
```

Q_2:
```
DELETE FROM R WHERE a = 20;
INSERT INTO R VALUES(20,10);
SELECT * FROM R;
```

Határozzuk meg, hogy a következő kijelentések közül melyek igazak, függetlenül az *R* tábla tartalmától. Magyarázzuk!

a) Q_1 és Q_2 ugyanazt az eredményt adják.
b) Q_1 eredménye mindig részhalmaza (bennfoglaltatik) Q_2 eredményének.
c) Q_2 eredménye mindig részhalmaza (bennfoglaltatik) Q_1 eredményének.
d) Q_1 és Q_2 különböző eredményeket adnak.

9. Az alábbiakban az *S* reláció egy előfordulása látható, a reláció sémája:
S[FK_1, FK_2, A, B, C, D, E], kulcsa: {FK_1, FK_2}.

FK1	FK2	A	B	C	D	E
1	1	a1	b1	c1	7	2
1	2	a_	b3	c1	5	2
1	3	a2	b1	c2	null	2
2	1	a3	b3	c2	null	100
2	2	a3	b3	c3	null	100

Adjunk választ az alábbi kérdésekre:

A. Hány rekordja lesz a lekérdezés eredményének?

```
SELECT *
FROM S
WHERE A LIKE 'a_'
```

a. 5
b. 4
c. 0
d. 1
e. Egyik sem a fentiek közül.

B. Mennyi a különbség a két lekérdezés eredmény-relációinak kardinalitásai között?

```
SELECT FK2, FK1, COUNT(DISTINCT B)
FROM S
GROUP BY FK2, FK1
HAVING FK1 = 1

SELECT FK2, FK1, COUNT(C)
FROM S
GROUP BY FK2, FK1
HAVING FK1 = 2
```

a. 1
b. 0
c. -1
d. -2
e. Egyik sem a fentiek közül.

C. Az alábbi állítások közül melyik helyes?
a. Az alábbi funkcionális függőségek közül legalább egy nincs kielégítve a reláció adatai által: $\{A\} \rightarrow \{B\}$, $\{FK_1, FK_2\} \rightarrow \{A, B\}$, $\{FK1\} \rightarrow \{A\}$.
b. A reláció adatait figyelembe véve, biztosan kijelenthetjük, hogy legalább egy az alábbi funkcionális függőségek közül fennáll az S sémára vonatkozóan: $\{A\} \rightarrow \{B\}$, $\{FK_1\} \rightarrow \{A, B\}$, $\{FK_1\} \rightarrow \{A\}$.
c. Az alábbi funkcionális függőségek közül legalább kettő nincs kielégítve a reláció adatai által: $\{FK_2\} \rightarrow \{A, B\}$, $\{A\} \rightarrow \{E\}$, $\{A, B\} \rightarrow \{E\}$, $\{B\} \rightarrow \{C, E\}$.
d. A reláció adatait figyelembe véve, biztosan kijelenthetjük, hogy legalább kettő az alábbi funkcionális függőségek közül fennáll az S sémára vonatkozóan: $\{FK_2\} \rightarrow \{A, B\}$, $\{A\} \rightarrow \{E\}$, $\{A, B\} \rightarrow \{E\}$, $\{B\} \rightarrow \{C, E\}$.
e. Egyik sem helyes a fentiek közül.

D. Hány rekordja lesz a lekérdezés eredményének?

```
SELECT *
FROM S
WHERE B = 'b1' OR D = 5
```

a. 2
b. 3
c. 1
d. 5
e. Egyik sem a fentiek közül.

4. fejezet Operációs rendszerek

4.1. A Unix állományrendszer

4.1.1. 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 kövegtkező fajta speciális állományokról beszélhetünk: katalógus (directory), eszköz (device), szimbolikus lánc (symbolic link), nevesített FIFO csővezeték (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 csővezeték (pipe), osztott memória szegmensek, üzenetsorok, szemaforok.

Egy *közönséges állomány* oktettjeit feldolgozhatjuk szekvenciálisan, de hozzáférhetünk közvetlenül is egy bizonyos bájthoz, a sorszámának 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 létrehozottt 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 gyökér 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 gyökérhez (/) 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 gyökér katalógustól kezdődően) a `pwd` parancs adja meg.
Létrehozhatunk egy új katalógust az `mkdir` parancs segítségével, egy katalógus törlését pedig a `rmdir` parancs teszi lehetővé.

4.1.2. 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* állományunkat 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=rwxs,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 a 4.1 ábra szemlélteti:

állománynév (tetszőleges hosszúságú)	inode szám

4.1 ábra Egy katalógus bejegyzés szerkezete

Az inode szám kilistázható az ls –i 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, a 4.1.3. 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 helytakarékosság 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 törlése*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:

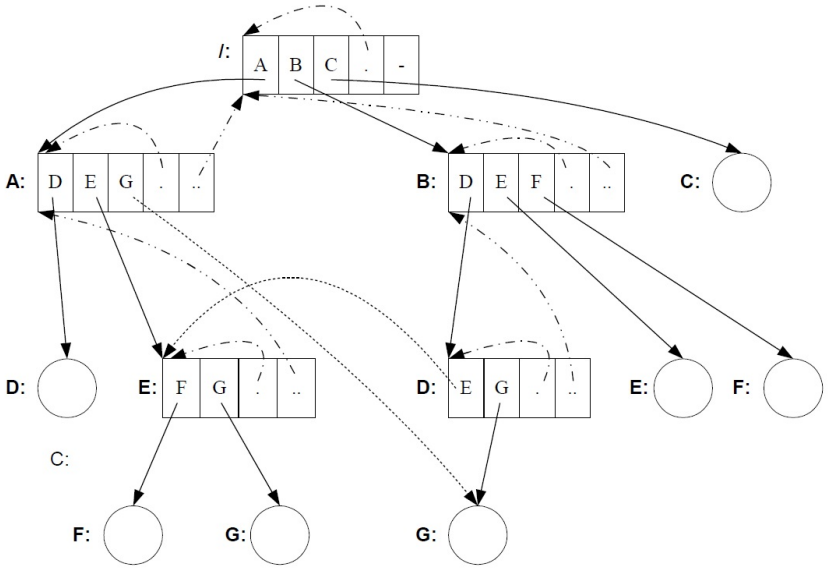
```
$ ln  regi  ujlink
$ ls -li
total 8
2098858 -rw-r--r-- 2 tsim1234 student 19 2013-03-17 19:26 regi
2098858 -rw-r--r-- 2 tsim1234 student 19 2013-03-17 19:26 ujlink
```

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 fájlrendszerben levő fájlok 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.



4.2 ábra Állományrendszer. Egyszerű példa.

A 4.2 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	D/E	./D/E	
/A/E/F	../A/E/F	D/E/F	./D/E/F	
/A/E/G	../A/E/G	D/E/G	./D/E/G	
/B	.			
/B/D	D	./D		
/B/D/G	D/G	./D/G	/A/G	../A/G
/B/E	E	./E		
/B/F	F	./F		
/C	../C			

4.1.3. A UNIX logikai lemez szerkezete

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

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

stb.

Minden egyes Unix alapú fájlrendszernek 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 indirekt címek listájában. Ezen konstansok értékétől függetlenül, egy új állomány bejegyzése, illetve ennek az adataihoz való hozzáférés, hasonló elvek alapján történik.

Mount

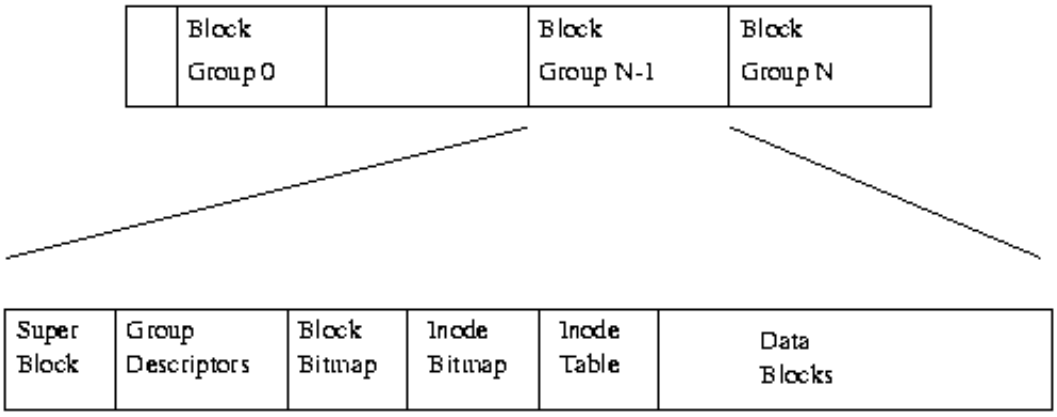
A Unix állományrendszer egységes fájlrendszer, az elérési út nem tartalmaz lemezegység nevet. A különböző logikai vagy fizikai lemezen levő fájlrendszert becsatoljuk (`mount`) a rendszerbe. Egy üres directory-hoz csatlakoztatható az új fájlrendszer, ennek gyökér

katalógusára az eredetileg üres directory nevével hivatkozhatunk. A felhasználó számára észrevétlen, hogy mi melyik fájlrendszerben 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 (`mke2fs`). Az állományok nyilvántartása az inode táblázat segítségével történik. A katalógus a fájlok neve és inode száma között hoz létre kapcsolatot. A directory is egy fájl.

Az ext2 fájlrendszerben a tárolóhely *blokk*okra van felosztva, ezek pedig *blokk csoportok*at alkotnak. A rendszer számára kritikus információk ismétlődnek minden csoportban, amint azt a 4.3 ábra szemlélteti:



4.3 ábra Logikai lemez szerkezete

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

Minden egyes blokk-csoport tartalmazza az ún. *szuperblokk* (super block) másolatát, egy csoport deszkriptort (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 fájlrendszer legelső blokkjában szereplő adatokat használja a rendszer bootoláskor.

A szuperblokk a következő információkat tartalmazza:

- **Magic Number** – *0xEF53* – ext2 esetén.
- **Revision Level** – verzió szám

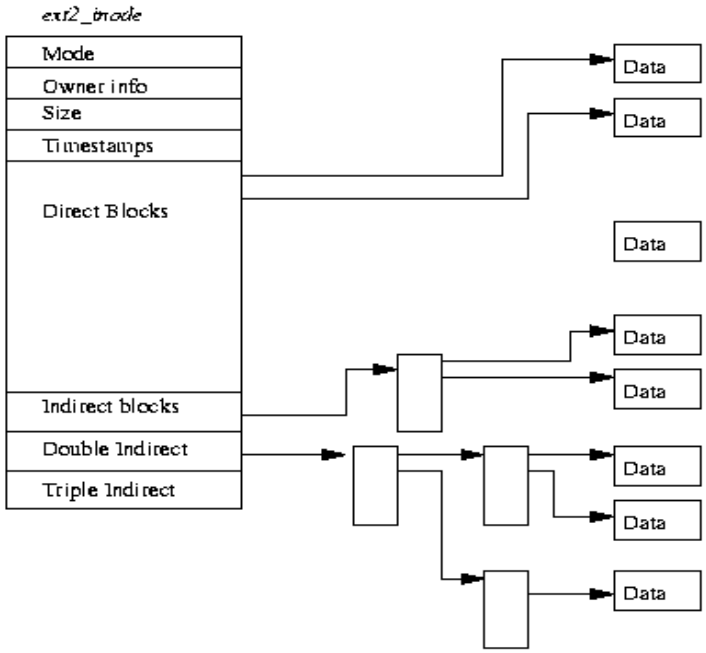
- **Mount Count and Maximum Mount Count** – a fájlrendszer teljes ellenőrzése ajánlott, ha eléri max-ot
- **Block Group Number** – a blokkcsoport száma, amelyikben ez a szuperblokk van,
- **Block Size** – blokk mérete byte-okban
- **Blocks per Group** – blokkok száma egy csoportban
- **Free Blocks** – szabad blokkok a fájlrendszerben
- **Free Inodes** – szabad inode-ok a fájlrendszerben
- **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 direktory-bemenetek 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, a 4.4 ábrán szemléltetett logika szerint:



4.4 ábra Egy állományhoz tartozó adatblokkok nyilvántartása

Az 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.

4.2. Unix folyamatok

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

4.2.1. 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 folyamattábla 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 kódszegmenssel

Az újonnan létrehozott folyamatot *gyerekfolyamat*nak, a `fork()` hívást végrehajtó folyamatot pedig *szülőfolyamat*nak nevezzük. Leszámítva, hogy külön adat-, illetve veremszegmenssel 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:

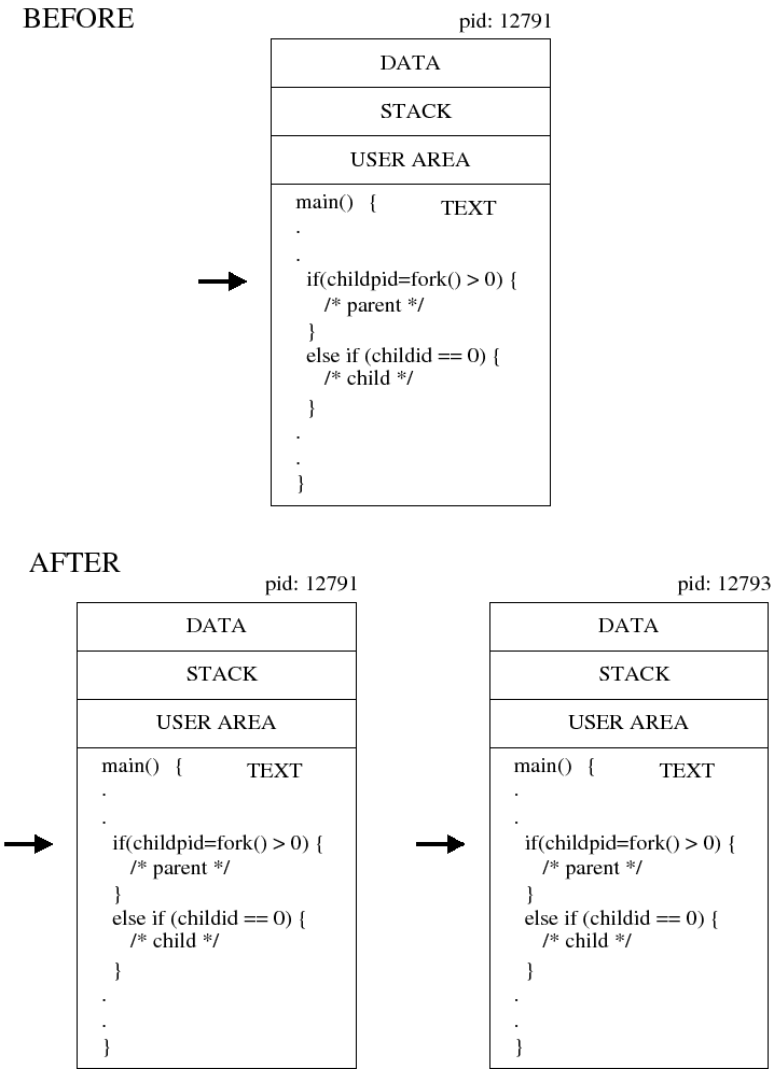
```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid(void); //PPID lekérdezése
pid_t getpid(void);  //PID lekérdezése
```

A 4.5 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 képének 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 */
}
```



4.5 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 másodpercnyi 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(”Letrehoztam a %d GYEREKet\n”,pid);
     for (i=1;i<=10;i++){
       sleep(1);        //1 másorpercnyi 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 GYEREKet
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 folyamattábla 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,
- visszatérithet 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 gyerek 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 másikból. 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 veremszegmens jön létre, a régieket felszabadítja
- a folyamattábla bemenetet örökli az eredeti folyamattól

Az `exec*` utáni utasítás csak hiba esetén hajtódik végre.
A 4.1 táblázat összegzi az `exec*` függvénycsaládba tartozó rendszerhívásokat és ezek jellemzőit (három kritérium szerint hat függvényt kínál fel a rendszer):

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

4.1 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 execle(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).

4.2.2. Folyamatok közti kommunikáció pipe-on keresztül

A pipe mechanizmus

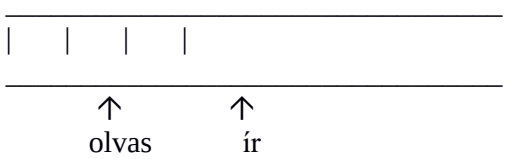
A pipe mechanizmus megjelenését a Unix alapú rendszerekben az indokolta, hogy lehetővé tegye a gyerekfolyamat szülővel való kommunikációját.

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

Pl. `who|sort|less`

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.
A 4.1.3 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 örökli a megnyitott deszkriptorokat. Az 4.6 ábra szemlélteti a pipefile-on keresztül történő kommunikációt.

4.6 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 fájlok esetén használjuk. A pipefileok kezelésére a `read` és `write` függvényeket ajánljuk. Paraméterként meg kell adnunk a pipefile 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 pipefileból próbálunk olvasni, a folyamat blokálódik 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 osszefuzeset valositja meg pipe segitsegevel
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main (){

      int p[2];
      pipe (p);
      if (fork () == 0) {        // elso gyerek
         dup2 (p[1], 1);         // standard kimenet atiranyitasa
         close (p[0]);
         execlp ("who", "who", NULL);
      }
      else if (fork () == 0) {   // masodik gyerek
         dup2 (p[0], 0);         // standard bemenet atiranyitasa
         close (p[1]);
         execlp ("sort", "sort", NULL);// sort vegrehajtasa
      }
      else {                 // szulo
         close (p[0]);
         close (p[1]);
         wait (NULL);
         wait (NULL);
      }
      exit(0);
}
```

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

4.2.3. 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 deszkriptor 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 ősszel 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 álomány 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 a 4.2 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.

A 4.2 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
FIFO megnyitva csak írásra, de olvasó folyamat nincs	várakozik mindaddig, amíg egy másik folyamat meg nem nyitja írásra a *FIFO* állományt	azonnal visszatér, várakozás és hibajelzés nélkül
FIFO megnyitása írásra, de olvasó folyamat nincs	várakozik mindaddig, amíg egy másik folyamat meg nem nyitja olvasásra a *FIFO* állományt	azonnal visszatér, hibajelzéssel: az `errno` értéke `ENXIO`
olvasás *FIFO* vagy pipe fileból, de nincs olvasnivaló adat	várakozik mindaddig, amíg adatok nem kerülnek a *FIFO* állományba, vagy amíg nincs egyetlen olyan folyamat sem, amely írásra nyitotta meg a FIFO 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 *FIFO* vagy pipe fileba, amikor az tele van	várakozik mindaddig, amíg ürül hely a FIFO állományban, majd annyi adatot ír bele, amennyi számára hely van	azonnal visszatér és 0-t térít vissza

4.2 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 folyamatazonosítója 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 befejeződésekor 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 letrehozza a sajat
                                        // fifo-jat */
  fd = open("szerverfifo", O_RDONLY);   /* megnyitja olvasasra */

  do                                    /* amig 0-t nem kuld 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);              /* letrehoz 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 */
  fd1 = open(s, O_RDONLY);
  read(fd1, &t, sizeof(t));             /* a valasz */
  close(fd1);
  unlink(s);                            /* kliensfifo torlese */
  printf("a negyzete: %d\n", t.szam);
  exit(0);
}
```

4.3. Shell programozás és alapvető Unix parancsok

Parancsértelmező (Bourne shell - sh)

4.3.1. 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 lezárnunk 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
```

4.3.2. 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 *string*et (sor elején) tartalmazó sorig
- | – pipeline (csővezeték)
- * – egyezés bármely lánccal (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 írnánk 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 *környezet*ét (*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 metakaraktereinek 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 elveszlődni 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 *karaktersor*. 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:

```
$ valtozonev=karaktersor
```

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

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

```
$ szo1=sivatagban
$ szo2=kutat
$ echo A $szo1 egy csapat $szo2 as
```

Az echo parancs végrehajtásakor, mely egy sor kiírását végzi el, előbb sor kerül a szo1, 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 $szo1 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 $szo1 megkezdodott a ${szo2}as
```

A parancs végrehajtásának eredménye ekkor:

```
A sivatagban megkezdodott a kutatas
```

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 írnunk 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 *igazságértéké*nek 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 végrehajtódik 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, végrehajtódik az `else` utáni parancslista és az `if` végehajtása 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 szóközökkel elválasztott egyszerű szavakat jelöl vagy helyettesítő karaktereket tartalmazó állománynevek szerepelhetnek ott, melyek ki lesznek terjesztve az összes illeszkedő állománynévre, így végül egy állomány-listát 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 generélni 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
Elnezest 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 végrehajtódik 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:

```
while  [  $#  -gt  0 ]
  do
    if  [  -f "$1" ]
      then  sort  $1  |  more
      else  echo  "nincs $1 file"
    fi
    shift
  done
```

```
until  [  $#  -eq  0 ]
  do
    if  [  -f "$1" ]
      then  sort  $1  |  more
      else  echo  "nincs $1 file"
    fi
    shift
  done
```

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.

Reguláris kifejezések

A *reguláris kifejezés* (regular expression) egy egy mintát meghatározó karaktersorozatot jelent, mely akár több konkrét karaktersorra is illeszkedhet. A Unix által rendelkezésre bocsátott eszközök között számos olyan szerepel, mely mintaillesztést használ, ilyen például a **`grep`** vagy **`egrep`** parancs, mely a bemeneti sorok közül kiszűri a megadott mintára illeszkedőket.
A reguláris kifejezésekben szerepelhetnek speciális jelentést hordozó karakterek, ezeket metakaraktereknek hívjuk, hasonlóan a shell-ben a fájlnév behelyettesítéskor használt speciális karakterekhez. Vigyázzunk azonban, mivel két, egymástól különböző, fogalomról van szó, ne keverjük össze a használatukat.
A reguláris kifejezésekben használt metakarakterek a következők:
.[]\^$*
A kiterjesztett reguláris kifejezésekben (extended regular expression, a továbbiakban a ktrk. rövidítést használjuk) az alábbi metakarakterek szerepelhetnek (ezeket pl. az `egrep` mintaillesztő tudja értelmezni):
.[]\^$*+(){}|

Az alábbiakban megadjuk a reguláris kifejezéseket meghatározó szabályokat.

- metakaraktereket nem tartalmazó kifejezés csak sajátmagára illeszkedik (pl. az *abc* reguláris kifejezés kizárólag az *abc* karaktersorra illeszkedik)
- **\c** - a c karakterre illeszkedik (pl. * a *-ra; \\ a \-re)
- **.** (pont) - bármelyik (nem újsor) karakterre illeszkedik (pl. ab. illeszkedik az aba, abb, abc, ... abz, ab0 stb. karaktersorozatokra)
- ha *e* reg. kif., akkor ***e**** az *e* reguláris kifejezés 0 vagy többszöri előfordulására illeszkedik (pl. a* illeszkedik az üres stringre, a, aa, aaa,..-ra
- ***e+*** (ktrk.) – *e* 1 vagy többszöri előfordulására illeszkedik. Helyettesíthető *ee**-al.
- **[...]** – illeszkedik az abban a pozícióban lévő bármely, a zárójelben felsorolt karakterre. (Pl. [aeiou] az angol ábécé bármelyik kisbetűvel írt magánhangzójára illeszkedik)
  - egymás után következő karaktereket rövidíteni lehet Pl. [0-9a-z]
  - a nyitó zárójelet követő ^ a felsorolt karakterek tagadása. (Pl. [^0-9] illeszkedik bármely, nem számjegy karakterre)
  - a - karaktert a \- karakterpáros jelöli.
  - a] zárójel csak a felsorolás első tagja lehet.
- Nevesített karakterosztályok. Ezek konkrét jelentése függhet a nyelvi lokalizációtól. Ahhoz, hogy a hagyományos interpretáció érvényesüljön, fontos, hogy az LC_ALL környezeti változó értéke C-re legyen állítva
  - [:alnum:] – alfanumerikus karakterek bármelyikére illeszkedik (egyenértékű a következő kifejezéssel, az ASCII kódolást tekintve `[0-9A-Za-z]`)
  - [:alpha:] – bármelyik betűre illeszkedik (`[A-Za-z]`)
  - [:cntrl:] – vezérlő karakterek (`[\x00-\x1F\x7F]`)
  - [:digit:] – számjegy
  - [:graph:] – látható karakterek (minden karakter, kivéve a vezérlő karaktereket és szóközöket)
  - [:lower:] – kisbetű (`[a-z]`)
  - [:print:] – látható karakterek és szóközök (minden karakter, kivéve a vezérlő karaktereket)

- [:punct:] – központozásban használt jelek (`[!"\#$%&'()*+,\-./:;<=>?@\[\\\]^_`{|}~]`)
- [:space:] – tetszőleges fehér karakter, az újsort is beleértve (`[ \t\r\n\v\f]`)
- [:upper:] – nagybetű (`[A-Z]`)
- [:xdigit:] – hexa számjegy (`[A-Fa-f0-9]`)
- ***e1* \| *e2*** – **ktrk: e1|e2** – illeszkedik *e1* vagy *e2*-re.
  (Pl [a-z]|\. - az adott pozicióban csak kisbetű vagy pont lehet)
- ^ a sor elejére, **$** a sor végére illeszti a mintát.
  Pl. ^$ vagy ^ *$ - üres sor, ^[^0-9]*$ - számot nem tartalmazó sor
- **\(...\)** – **ktrk: (…)** – illeszkedik a zárójelbe tett kifejezésre, és egyben megjelöli azt (csoportosításra is használt).
- **\n**, ahol n szám - a zárójelezéssel kijelölt mintára hivatkozik, a kijelölés sorrendjében.
  Pl. ^\(.\)\(.\).*\2\1$ – **ktrk:** ^(.)(.).*\2\1$ – olyan sor, ahol a sor első két karaktere tükörszimmetrikus az utolsó két karakterre.
- **c\{m,n\} – ktrk: c{m,n}** –, ahol m és n 256-nál kisebb nemnegativ egész - a minta legalább m-szer, és legfennebb n-szer fordul elő egymás után.
  - csak n - pontosan n előfordulás
  - csak m- legalább m előfordulás

Megoldott példafeladatok

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 lementettel, 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 legtöbb 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 szo1 változóban megjegyezzük egy sor első szavát
   #(szóelválasztónak a szóköz karaktert tekintjük)
   for szo1 in `cat $x | cut -d" " -f1`
    do
    #megvizsgaljuk, hogy a sor nem-e üres, illetve az első szó
    #hosszát
      if [ ! -z "$szo1" ] && [ `expr length $szo1` -ge 5 ]
        then
         echo A $x fileban megtaláltuk $szo1 szót, \
              melynek  hossza `expr length $szo1`
         #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 őt alkotó minden egyes állomány elején egy, az állományt azonosító fejlécet 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 kinyomtassa, a felhasználó használhatja a fentebb említett funkcionalitást 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.

4. példa: bejelentkezett felhasználó folyamatai

Olvassunk be felhasználóneveket a billentyűzetről, üres karaktersor beolvasásáig.

Amennyiben létezik az illető felhasználó, és be van jelentkezve, írjuk ki az általa éppen futtatott folyamatok nevét, és ezek számát (mindeniket csak egyszer vesszük számításba), különben írjunk ki megfelelő hibaüzenetet (nemlétező felhasználó vagy az illető felhasználó nincs bejelentkezve).

Megjegyzések: egy végtelen ciklusban (`:` olyan utasítás, mely mindig 0-t térít vissza) beolvasunk (`read` utasítás) egy felhasználónevet. Ha a beolvasott karaktersor üres, kilépünk a ciklusból (`break`). Megvizsgáljuk, hogy a beolvasott felhasználónév benne van-e az `/etc/passwd` állományban, mely a rendszer felhasználóiról tárol információt (a felhasználónév a sor elején kell szerepeljen, utána pedig egy „:” következik, ez választja ugyanis el az illető felhasználóhoz kapcsolt különböző adatokat egymástól). Ha megtaláltuk a felhasználót, azt is megvizsgáljuk, hogy be van-e jelentkezve (`who` parancs). Ha valamelyik feltétel nem teljesül, kiírjuk a megfelelő hibaüzenetet, különben a `flymtk` változóba mentjük az illető folyamatait (lásd a `ps` parancsot. Ennek „`-u`” opciójával adjuk meg a felhasználót, akinek a folyamatai érdekelnek, illetve az „o” opció segítségével formázzuk a kimenetet. Mivel azt szeretnénk, hogy minden parancs csak egyszer jelenjen meg, ezért rendezzük a kimenet sorait a `sort` parancs „`-u`” opciójával). A `wc` parancs segítségével megszámoljuk a folyamatokat (az elmentett karaktersorban szereplő szavak száma). Fontos, hogy a `ps` parancsot csak egyszer hajtsuk végre, ezért mentettük el a kimenetét egy változóban, hogy ebben számoljuk meg a folyamatok számát, és ne egy újabb `ps` hívás kimenetében, ami megtörténhet, hogy más eredményt adna.

```
#!/bin/sh
while :
do
  echo "Kerek egy felhasznalonevet (ures sor - befejezes):"
  read user
  if [  "$user" = "" ]
    then
      break
    fi

  if grep -q "^$user:" /etc/passwd
    then
       if who|grep -q "^$user "
         then
           #megjegyezzuk egy valtozoban a $user felhasznalo folyamatait
           flymtk=`ps -u $user o comm=|sort -u`
           #kiirjuk a folyamatokat es ezek szamat
           echo "$user felhasznalo folyamatai:"; echo $flymtk
           echo "   `echo  $flymtk|wc -w` folyamatot futtat"
```

```
          else
            echo $user felhasznalo nincs bejelentkezve
        fi
    else
      echo  $user felhasznalo nem letezik a rendszerben
    fi
done
```

4.4. Javasolt feladatok

I.

a. Írjuk le röviden a `fork` rendszerhívás működését, és ennek lehetséges visszatérítési értékeit.

b. Mit ír ki a képernyőre az alábbi programrész, feltételezve, hogy a `fork` rendszerhívás sikeresen hajtódik 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
```

4.5. 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 in C++*. Programare generică si 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., Dragoş 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 şi algoritmi. O perspectivă orientată obiect.*, Ed. Casa Cărţii de Stiinţă, 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 şi Informatică, Centrul de Formare Continuă şi Invăţământ la Distanţă, 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