V1.3

Algoritmusok és Adatstruktúrák

II. félév

Hernyák Zoltán

E másolat nem használható fel szabadon, a készülő jegyzet egy munkapéldánya.

A teljes jegyzetről, vagy annak bármely részéről bármely másolat készítéséhez a szerző előzetes írásbeli hozzájárulására van szükség. A másolatnak tartalmaznia kell a sokszorosításra vonatkozó korlátozó kitételt is. A jegyzet kizárólag főiskolai oktatási vagy tanulmányi célra használható!

A szerző hozzájárulását adja ahhoz, hogy az EKF számítástechnika tanári, és programozó matematikus szakján, a 2001 tanévtől a tárgyat az EKF TO által elfogadott módon felvett hallgatók bármelyike, kizárólag saját maga részére, tanulmányaihoz egyetlen egy példány másolatot készítsen a jegyzetből.

A jegyzet e változata még tartalmazhat mind gépelési, mind helyességi hibákat. Az állítások nem mindegyike lett tesztelve teljes körűen. Minden észrevételt, amely valamilyen hibára vonatkozik, örömmel fogadok.

Eger, csütörtök, 2003. szeptember 25. 23:10
Hernyák Zoltán

Algoritmusok és adatstruktúrák II. félév

Hernyák Zoltán

Eljárások készítése - bevezetés

Az összetett utasítások kifejtésének egyik módja **eljárások** készítése. Egy eljárásnak...

- ... van **neve**, azonosítója. Az eljárás nevének képzésére ugyanazon szabályok érvényesek, mint a változók neveire. Az eljárás neve a programon belül **teljesen egyedi**, ugyanilyen nevű változó sem lehet!
- ... lehetnek **paraméterei**. A paraméterek változókban átadott értékek, amelyek az eljárás működését befolyásolják. A paraméterek számát, és típusát az eljárás fejrészében, a neve mögött kell megadni, deklarálni.
- ... lehetnek **lokális változói**. A lokális változók az eljárás fejrészének megadása után, az eljárás törzsének definiálása előtt kell hogy szerepeljen.
- … van **törzse**, amely az eljárást – mint összetett utasítást – alkotó utasítások sorozata.

Példa az eljárás megadására:

```
ELJÁRÁS TombBekeres
  I : egész
EKEZD
  Ciklus I:=1-től N-ig
     Be: A[I]
  Cvége
```

Mint láthatjuk, az eljárást az `ELJÁRÁS` kulcsszóval vezetjük be, melyet az eljárás neve követ, jelen esetben az `TombBekeres`. Ezen egyszerű példában nincsenek az eljárásnak paraméterei. Az eljárásnak egyetlen lokális változója van, az `I`. Az eljárás törzsét az `EKEZD` kulcsszóval vezetjük be, majd következnek az utasítások. Az eljárást az `EVÉGE` kulcsszó zárja.

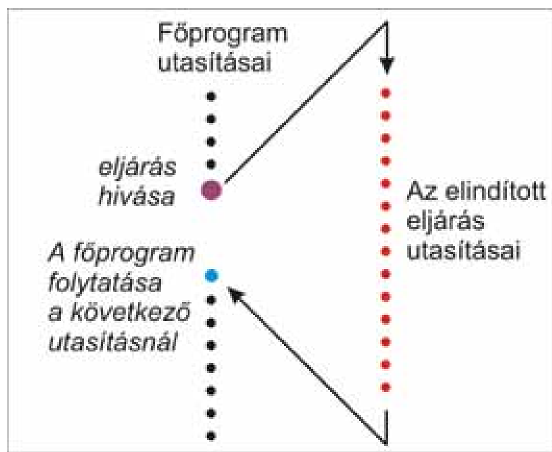
Az eljárást az alábbi módon lehet aktivizálni:

```
PROGRAM Proba

DEKLARÁCIÓ
 A : tömb [1..10] egész-ből

 TombBekeres
 ...
AVÉGE
```

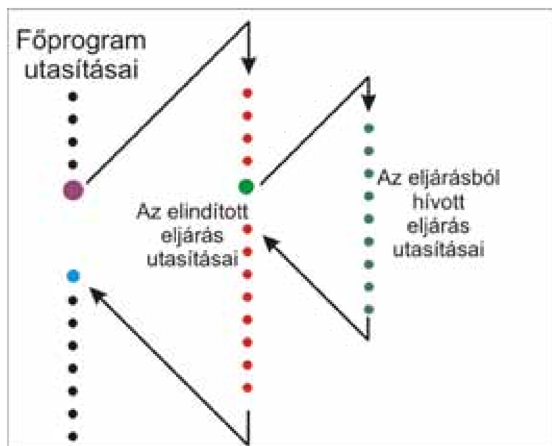
A fenti példában a `TombBekeres` összetett utasításnak minősül, s alapesetben a számítógép nem ismer fel, nem ért meg. De a fenti eljárás leírása után már tudja, hogy valójában mi az a tevékenység, melyet végre kell hajtania.

A program végrehajtása mindig az `AKEZD` kulcsszónál kezdődik. Ez a fő algoritmus a program **főprogramja**. A főprogramban állhatnak egyszerű utasítások, és összetett utasítások is. Ez utóbbiaknál természetesen le kell írni, mit is értünk alattuk, vagyis meg kell az eljárások törzsét. Ezt az eljárások ***kifejtésének*** nevezzük.

Egy ilyen algoritmus végrehatása az alábbi módon értelmezhető:



Egy eljárásban lehet összetett utasításokat is használni, vagyis egy eljárásból lehet további eljárásokat hívni. Ezeknek végrehajtása az előzőekhez analóg módon zajlik.



Másik példa:

```
PROGRAM Proba

DEKLARÁCIÓ
  A : tömb [1..10] egész-ből
  I : egész

ELJARAS TombElemKiir(K:egész)
EKEZD
  Ki: "A(",K,")=",A(K)
EVÉGE

AKEZD
  Ciklus I:=1-től 10-ig
     TombElemKiir( I );
  Cvége
AVÉGE
```

A fenti példában az eljárásnak van paramétere, `K`. A `K` változó csak az eljárásban van értelmezve, a `K` deklarációja ezen eljárásra *lokális*. A program egyéb területein a `K` nem használható!

Az eljárásban a `K` értéke megegyezik az `I` értékével, ezen értéket a `K` az eljárás hívásakor automatikusan felveszi.

Az eljárás fejrészében feltüntetett paraméterlistát ***formális paraméterlistának*** nevezzük. A formális paraméterlistában a paraméterek - mint változók - neveit, és típusát kell megadni felsorolás jellegűen.

Az eljárást ***aktivizálni***, meghívni, elindítani a nevével lehet. A híváskori paraméterlistát ***aktuális paraméterlistának*** nevezzük.

```
PROGRAM Proba

DEKLARÁCIÓ
  A : tömb [1..10} egész-ből
  I : egész

ELJÁRÁS KiirParosSzam(X:egész)
EKEZD
  HA  X mod 2=0   AKKOR
     Ki: "A szám páros"
  KÜLÖNBEN
     Ki: "A szám páratlan"
  HVÉGE
EVÉGE

AKEZD
  ...
  Ciklus I:=1-től 10-ig
    KiirParosSzam( A[I] )
  CVége
  ...
```

A fenti példában az `X` sorra felveszi az `A` tömb minden egyes elemének értékét, és mindegyik esetben kiírja, hogy az adott tömbelem páros-e, vagy sem.

Az eljárásokban a *program*-ban deklarált változókat is el lehet érni, azok a program egészére nézve érvényesek. Így ezen változókat ***globális*** változóknak nevezzük. Ezek alapján a fenti példát az alábbi módon is lehetett volna írni:

```
PROGRAM Proba

DEKLARÁCIÓ
  A   : tömb [1..10} egész-ből
  I   : egész

ELJÁRÁS KiirParosSzam
EKEZD
  HA  A[I] mod 2=0   AKKOR
     Ki: "A szám páros"
  KÜLÖNBEN
     Ki: "A szám páratlan"
  HVÉGE


AKEZD
  ...
  Ciklus I:=1-től 10-ig
    KiirParosSzam
  CVége
  ...
AVÉGE
```

Az összetett utasítások mások fajtája a függvény. A függvény annyiban különbözik az eljárástól, hogy van visszatérési típusa, és értéke. Vagyis nemcsak végrehajtódik - mint egy összetett utasítás - hanem vissza is ad egy értéket:

```
DEKLARÁCIÓ
  A : egész

FÜGGVÉNY Paros_E(X:egész):logika
FKEZD
  HA  X mod 2=0 AKKOR
      Paros_e := IGAZ
  KÜLÖNBEN
      Paros_e := HAMIS
  HVÉGE


AKEZD
  Ciklus I:=1-től 10-ig
    Be: A
    HA Paros_e(A) AKKOR
       Ki: "Ez egy páros szám"
    KÜLÖNBEN
       Ki: "Ez egy páratlan szám"
    HVÉGE
  CVége
AVÉGE
```

A függvényeknél a fejrész mögött fel kell tüntetni, hogy milyen típusú értéket adnak vissza - jelen esetben ez egy logikai típusú függvény lesz.

A visszatérési értéket úgy lehet megadni, hogy a függvény nevét, mint adott típusú változót fogjuk fel, és értéket helyezünk el benne. Ezen értékkel tér vissza a függvény.

Az eljárásokat és függvényeket közös néven ***alprogramoknak*** nevezzük.

Lokális és globális változók

A változók jellemzőit két csoportra oszthatjuk:
- **hatókör** : a program szövegezésén belül hol érhető el az adott változó ?
- **élettartam** : az adott változó mikor keletkezik, és szűnik meg ?

A ***globális változók*** az alábbi jellemzőkkel rendelkeznek:
- maga a program deklarációs részen kerülnek deklarálásra
- a program egész területén elérhetőek, és érvényben vannak: az eljárásokban és a függvényekben is, valamint a főprogramban.
- ezen változók a program indulásának pillanatában keletkeznek, és a program teljes futásának időtartama alatt megmaradnak, csupán a program befejezésekor szűnnek meg.

A ***lokális változók*** ezzel szemben az alábbi jellemzőkkel bírnak:
- *alprogramok* belsejében kerülnek deklarálásra
- csakis az őt deklaráló alprogram törzsében érhetőek el
- az alprogram aktivizálásakor jönnek létre, és amint ezen alprogram befejezi a futását, a változó megszűnik.

Az alprogramok *paraméter-változói* lokális változóknak tekinthetőek, egyetlen tulajdonságban térnek el tőlük: egy hagyományos értelemben vett lokális változó kezdőértéke nem definiált, míg a paraméter-változó kezdőértéke az alprogram aktivizálásakor automatikusan beállítódik - az aktuális paraméterlistának megfelelően.

Amennyiben ugyanolyan nevű lokális változót deklarálunk, amilyen nevű globális változó is létezik, úgy az alprogramon belül a lokális változó ,,látszik" csak, elfedve a globális párját, míg az alprogram törzsén kívül ezen változónévvel a globális változót lehet elérni:

```
PROGRAM Akarmi

DEKLARÁCIÓ
  A : Egész

ELJÁRÁS Kiir( A:egész )
EKEZD
 Ki: A
EVÉGE

AKEZD
  A := 10
  Kiir(20)
  Ki: A
AVÉGE
```

A Kiir eljárásban az A változó lokális, egy egészen más változót jelöl, mint a globális A. Ez kiderül onnan, hogy ha lefuttatjuk a fenti programot, akkor a „20”, majd „10” jelenik meg a képernyőn. Az eljárás indulásakor ugyanis a lokális A felveszi az aktuális paraméterlistában szereplő 20 értéket, melyet az eljárás kiír. Eközben a globális A változóban továbbra is a 10 érték marad, mely a főprogramba visszatérve meg is jelenik a következő kiíró utasítás hatására.

A lokális változó lehet más típusú is, mint a globális párja:

```
PROGRAM Akarmi

DEKLARÁCIÓ
  A : szöveg

ELJÁRÁS Kiir( A:egész )
EKEZD
 Ki: A*2
EVÉGE

AKEZD
  A := "Hello"
  Kiir(20)
  Ki: A
AVÉGE
```

Ezen a példán a lokális változó egész típusú, ezért számolható ki a A*2 kifejezés, melynek eredménye jelen példában 40 lesz, hiszen az A az eljárás hívásának pillanatában felveszi a 20 kezdőértéket. Ugyanakkor a főprogrambeli „Ki: A” utasítás a szöveg típusú A változót jeleníti meg a képernyőn, melynek értéke "Hello".

Nézzük az alábbi példát:

```
PROGRAM Akarmi

DEKLARÁCIÓ
  A : szöveg

ELJÁRÁS Kiir2
EKEZD
 Ki: A
EVÉGE

ELJÁRÁS Kiir1( A:egész )
EKEZD
 Ki: A*2
 Kiir2


AKEZD
  A := "Hello"
  Kiir(20)
```

A fenti program szintén a `40 Hello`-t jeleníti meg a képernyőn, ugyanis a `Kiir1`-n belül az `A` változó lokális, de a `Kiir2`-ben az `A` a globális változót jelenti, hiszen a `Kiir2` eljárásnak egyáltalán nincs lokális változója, pláne nincs ugyanilyen nevű lokális változója, amely elfedné a globális változót.

Leszögezhetjük, hogy a lokális változók ***szigorúan*** az őt deklaráló alprogramra nézve lokális, az alprogram törzsében, kódrészében szerepelhet el, és fedheti el a globális párját. Amennyiben az alprogramból további alprogramot hívunk, az nem veszi át, nem örökli automatikusan az őt aktivizáló alprogram lokális változóit. Minden alprogram a többitől **független** lokális változó-csoporttal dolgozik!

Cím szerinti, és érték szerinti paraméterátadás

A paramétereket az alprogramok két módon vehetik át. Az eddigi példákban mindig az ,,egyszerűbb" módot használtuk, az érték szerinti paraméterátadás-átvételt.

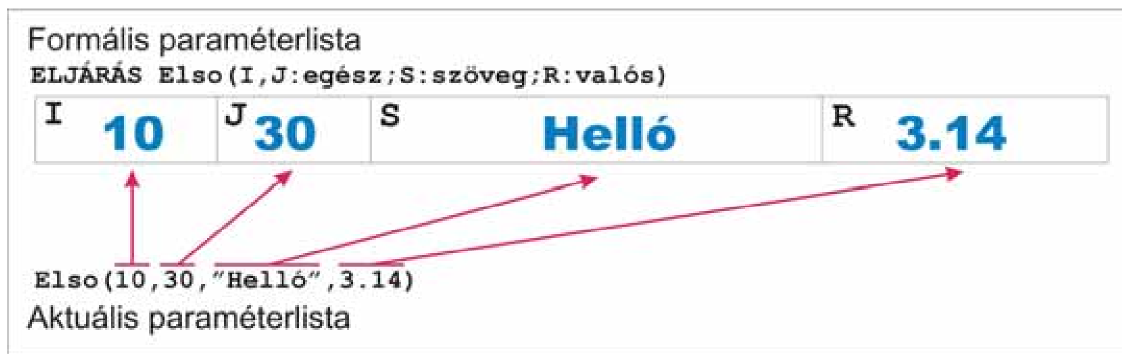
Az érték szerinti paraméterátadás esetén a formális paraméterlistában felsorolt változók egyszerűen átveszik az aktuális paraméterlistában felsorolt értékeket. A háttérben egyszerű értékadás zajlik le:

```
ELJÁRÁS Elso(I,J:egész;S:szöveg;R:valós)
EKEZD
  KI: I+J, S, R/2


AKEZD
 ...
 Elso(10,30,"Hello",3.14)
 ...
AVÉGE
```

A fenti példában az `Elso` eljárás indulása pillanatában automatikusan, láthatatlanul az alábbi értékadások hajtódnak végre:

```
  I:=10
  J:=30
  S:="Hello"
  R:=3.14
  ... eljárás elindul ...
```

A háttérben:



Természetes, hogy az aktuális paraméterlistában megadott értékeknek rendre megfelelő típusúaknak kell lennie. Az alábbi példa helytelen:

```
AKEZD

 ...
 Elso(10,3.14,"Hello",30)
 ...
AVÉGE
```

A második érték, a `3.14` a formális paraméterlistában soron következő `J`-nek adódna át, de a `J` egy egész típusú változó, mely nem veheti fel a fenti értéket. A számítógép ***nem fogja kitalálni***, melyik értéket melyik paraméterváltozónak akarjuk átadni, erre ***figyeljünk mi oda***!

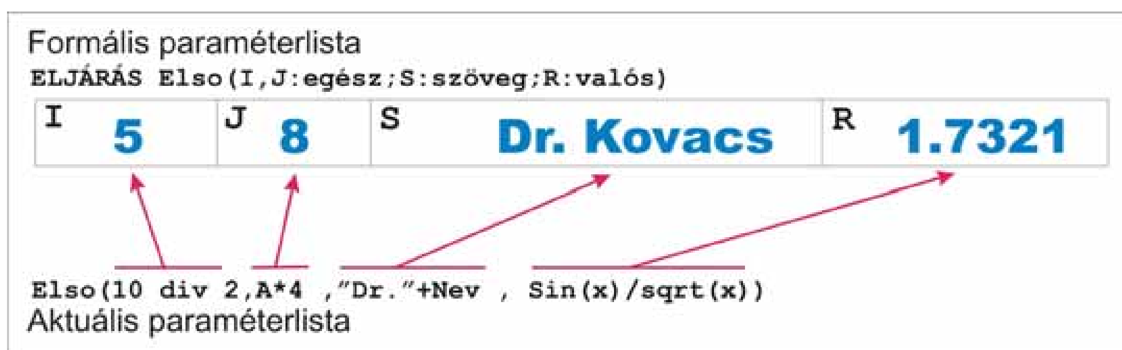
Ezen paraméterátadási technikát ***érték szerinti*** paraméterátadásnak nevezzük. Jellemzője, hogy az aktuális paraméterlistában csak arra kell ügyelni, hogy az aktuális érték az őt átvevő formális paraméter típusának megfeleljen, vagy azzal kompatibilis legyen. Az aktuális paraméter helyén állhat:

- konstans
- változó
- kifejezés

Valójában a hívás helyén mindig kifejezés áll, hiszen a konstans is egy kifejezés – amely egy tagú, típusa megegyezik a konstans típusával. Hasonlóan a változó is egytagú kifejezésnek fogható fel.

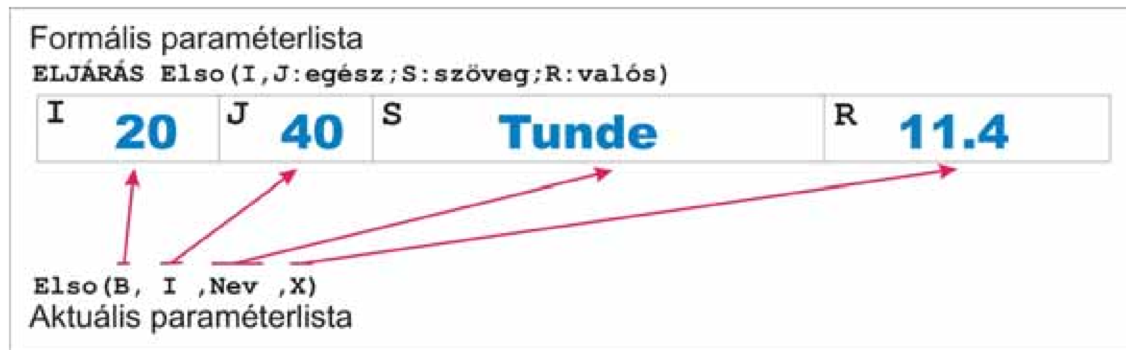
A fenti példában a hívás helyein rendre konstansok álltak. Nézzünk másmilyen példákat:

```
AKEZD
 A   := 2;
 X   := 60 fok;
 Nev := "Kovacs"
 Elso(10 div 2,A*4 ,"Dr."+Nev , Sin(x)/sqrt(x))
AVÉGE
```



És amennyiben a hívás helyén változó neve áll:

```
AKEZD
 B   := 20;
 Nev:= ”Tunde”
 I   := 40;
 X   := Max/Db;
 Elso(B, I ,Nev ,X)
 ...
AVÉGE
```



Figyeljük meg, hogy az `I` változó a főprogramban is deklarálva van, és értéke 40, de a formális paraméterlistában is szerepel ilyen nevű változó. Ez utóbbi lokális az eljárásra nézve. Vagyis az eljáráson belül az `I` ez utóbbit fogja jelenteni, az eljáráson belül az `I` értéke 20-ról indul, meg is változtatható, a főprogrambeli `I` e közben tartja az ettől független 40 értékét.

Ez utóbbira még egy példa:

```
PROGRAM Akarmi
DEKLARÁCIÓ
  X:egész

ELJÁRÁS Elso(A:egész)
EKEZD
 HA A>0 AKKOR
    A := A/2
 HVÉGE
 Ki: A
EVÉGE

AKEZD
  X:=10
  Elso( X )
  Ki: X
AVÉGE
```

A fenti példában az eljárás hívásakor átadódik az `A` változó számára az `X` aktuális értéke, a `10` (`A:=X`). E pillanattól kezdve az `X` és az `A` változó függetlenül létezik egymástól. Az `A` értéke jelen esetben az eljárás futása közben megváltozik, ezért az eljárásban az `5` érték íródik ki, de eközben az `X` változó tartja az értékét, a 10-t.

Az, hogy az aktuális paraméterlistában megfelelő típusú kifejezést kell írni az jelenti, hogy az aktuális paraméterlistában szereplő kifejezések típusa konvertálható típusú kell legyen. A típus konvertálható, ha eleve megegyezik a fogadó változó típusával (☺), vagy abba természetes módon átalakítható információvesztés nélkül. Az algoritmus keretein belül tárgyalt típusok közül ez lényegében csak az egész -> valós között áll fenn. Gyakorlati programozásban, egy adott programozási nyelvben azonban

több fajta egész típus is létezhet (byte, shortint, integer, longint, word, …), vagy több fajta valós. Néhány programozási nyelven megengedett az egész <-> karakter, egész <-> logikai típus konverziók is. Erről az adott programozási nyelv dokumentációja ad bővebb felvilágosítást.

A másik paraméterátadási technika a cím szerinti paraméterátadás.

Ennek során a hívás helyén az előzőektől eltérően csakis változó neve szerepelhet. Ennek oka, hogy ez a paraméterátadás az előző technikától eltérően **kétirányú**!

```
PROGRAM Akarmi
DEKLARÁCIÓ
  A:tömb [1..10] valós-ból
  AMin,AMax : valós


ELJÁRÁS Szelsoertekek(CÍM Min,Max:valós)
 I:egész
EKEZD
  Min := A[1]
  Max := A[1]
  Ciklus I:=2 to 10
   HA A[I]<Min AKKOR
      Min := A[I]
   HVÉGE
   HA A[I]>Max AKKOR
      Max := A[I]
   HVÉGE


AKEZD
  ...
  SzelsoErtekek(AMin,AMax)
AVÉGE
```

Ennek során az eljárás indulásakor a MIN és MAX változó valójában ugyanazon a memóriaterületen fog elhelyezkedni, mint az aktuális paraméterlistában szereplő AMin, AMax változók. Az eljárás futása közben a Min és Max változóba kerülnek a tömb szélső értékei. Ha érték szerinti lenne a paraméter-átvétel, akkor ezen eljárásnak (az időhúzáson kívül) semmi haszna sem lenne. Ugyanis ez esetben a Min és Max változók sima lokális változók lennének, az eljárásból kilépéskor megszűnnének létezni – s a bennük tárolt értékek is eltűnnek (elenyészik az elektronikus világegyetemben).

De mivel cím szerinti paraméterátadás-átvétel történt, az eljárás futásának idejére a Min és az AMin változók ugyanazon a memóriaterületen foglaltak helyet, így amikor a Min változóba értéket tettünk (értékadás révén), ugyanakkor az AMin is felvette ugyanezen értéket. Így bár az eljárás befejezte a futását (és a Min elenyészett), az AMin-ben lévő érték megmarad.

A cím szerinti paraméterátadás felfogható változó átnevezésnek is: az eljárás futása során az „AMin” változóra „Min” néven hivatkozunk.

A cím szerinti paraméterátadás révén lehet megoldani azt a problémát, hogy egy alprogram egynél több értéket adjon vissza. Ha az alprogram csak végrehajt bizonyos tevékenységsorozatot (pl. egy tömb kiírása a képernyőre), akkor eljárást írunk. Ha egyetlen értéket kell visszaadni (pl. eldönteni egy egész számról, hogy prím-e), akkor függvényt írunk. Ha több értéket is vissza kell adnia, akkor megoldható, hogy függvényt írunk (mely visszaadja az egyik értéket, a többit cím szerinti paraméterátadások révén kapjuk vissza), vagy eljárást írunk (mely az összes értéket cím szerinti értékadás révén adja vissza).

A gyakrabban használt paraméterátadási technika az érték szerinti.

Előnyei:
- A hívás helyére kifejezést is írhatunk
- Amennyiben változót is írunk a hívás helyére, akkor is biztosak lehetünk benne, hogy az eljárás nem változtatja meg annak értékét (csak egyirányú kapcsolat)

Ez utóbbi a programbeli hibák elkerülésében fontos – amikor nem értjük hogy egy érték hogy került egy változóba, mindig gyanakodhatunk egy elindított eljárásra.

Hátrányai:
- A lokális változók általában egy speciális memóriaterületen (munkaterület) foglalnak helyet, melynek mérete korlátos. Túl sok lokális változó használata mellet ezen terület elfogyhat.
- Futásidőben lassú.

Ez utóbbinak oka az, hogy ekkor a változók duplán foglalnak helyet a memóriában – az aktuális paraméterlistában szereplő változó és a formális listában szereplő lokális változó egy másik területen. Az értékek átmásolása az eljárás hívásakor byte-ok kiolvasását jelenti az egyik helyről (forrás), és átmásolása a fogadó helyre (cél). Ez nagyobb memóriaméretű változók esetében (pl. rekordok, tömbök) jelentős időbe is kerülhet, főleg ha az eljárást a program futása során sokszor hívjuk.

Cím szerinti paraméterátadás esetén az előnyök hátrányokká, a hátrányok előnyökké válnak.

Előnyei:
- A lokális változók nem kerülnek plusz memóriába
- A paraméterátadás-átvétel gyors (mindig 4 byte másolása a memóriában)

Hátrányai:
- A hívás helyére csakis ugyanazon típusú változó kerülhet
- Az eljárás lefutása mellékhatásokkal is járhat: a cím szerint átadott változók értéke megváltozhat az eljárás futása alatt

Vektorok tárolása

A vektorok homogén adatszerkezetek, melyek maximális mérete általában a deklarálásukkor eldől. Amennyiben a vektor egy elemének eltárolásához C byte-ra van szükség[1], úgy egy N elemű vektor tárigénye N*C byte. Ezen byte-ok a memóriában egy összefüggő (folytonos) területen helyezkednek el. Ezen területnek van egy kezdőcíme (ahol a vektor első eleme helyezkedik el). Ezen kezdőcímtől kezdve a vektor elemei indexük szerinti növekvő sorrendben tárolódnak el.

A vektorokra jellemző a *véletlen* elérés, melynek lényege, hogy bármely vektorelem elérése ugyanannyi időbe kerül. Ezt szokták *direkt* elérésnek is hívni.

Amennyiben egy V vektor I elemét szeretnénk lekérdezni, úgy a számítógép egy képlettel meghatározza ezen elem tényeleges memóriacímét, majd ezen címen található adatokat átteszi a RAM memóriából az operatív területre[2].

- Ezen képlet kiszámítása minden elem esetén ugyanannyi időt vesz igénybe
- A RAM bármely byte-jának elérése ugyanannyi időt vesz igénybe

A fenti két ok miatt lehetséges a vektorok *véletlen* sebességű elérése.

[1] Hogy melyik adattípus mennyi memóriát köt le, az adott programozási nyelv dokumentációjában található, egyébként a „*bevezetés az informatikába*” tantárgyban tanultak adhatnak támpontot.
[2] Ez lényegében a processzor valamely belső regisztere.

A fenti képletet címfüggvénynek nevezzük.

A vektorok esetén (amennyiben a címfüggvény az alábbi:

V: tömb [1..N] _______-ből
Címfüggvény(V, I) = (V vektor első elemének memóriacíme)
+ (I-„első elem sorszáma”) * (egy elem tárigénye)

Mint látjuk, a címfüggvényben sehol nem szerepel az, hogy a vektor N elemet tárol. Ezért fordulhat elő az (bizonyos programnyelveknél), hogy ha túllépjük a vektor indexhatárait (akár negatív, akár pozitív irányban), akkor is „van” ott elem, kapunk valamely értéket. Ezen érték azonban már nem a vektorhoz tartozik, pusztán a címfüggvény által meghatározott memóriaterületen ezen érték található. *Ezen érték „lekérdezése” sosem okoz hibát, megváltoztatása annál inkább!*

A képletben szereplő „első elem sorszáma” a fenti példában „1” érték (a vektor `1`..N indexhatárokkal készült. Amennyiben ettől eltérünk (pl: vektor [-10..+10] … alakú a deklarálás), akkor értelemszerűen a legkisebb index (újabb példában ez -10).

A képlet első fele megadja, hogy a vektor tárolása melyik memóriacímen kezdődik. Ugyanis az „I” elem ettől relatív „arrébb” van. A második fele adja meg, hogy mennyivel „arréb”. Az „I” elem előtt **I**-1 elem van eltárolva, mindegyik C byte-ba kerül, tehát (**I**-1)*C byte. A képletben azonban nem **I**-1 van írva, hanem **I**-„első elem sorszáma”, mert ha a vektor [-10..+10] indexelésű, akkor az **I**=4 elem előtt nem 3 elem van eltárolva, hanem 4-(-10) = 14 elem (-10,-9,-8,...,0,1,2,3 – tessék összeszámolni)!

A mátrixok címfüggvénye

Az N*M mátrix memóriaigénye N*M*C (ahol C az egy mátrix-elem memóriaigénye). A mátrixok tárolása sorfolytonosan történik, vagyis egy N sorból és M oszlopból álló mátrix felfogható egy N db M hosszú vektornak is. Vagyis először eltárolásra kerül az első sor M eleme, majd a második sor M eleme, stb. A teljes mátrix egy összefüggő tárterületet foglal el a memóriában. A mátrix elemeinek elérése is *véletlen* idejű – minden elemet ugyanannyi időbe kerül elérni.

A mátrix címfüggvénye:
V: tömb [A..N,B..M] ________-ből
Címfüggvény(V, I, J) = (V mátrix első elemének memóriacíme)
+ (I-B)* M* (egy elem tárigénye)
+ (J-A)* (egy elem tárigénye)

A képlet első fele a mátrix tárolásának kezdetét adja meg.
A második sor szerint a **I**,J elem előtt **I**-1 sor egészben van eltárolva, ez soronként M*C byte-ba kerül.
A harmadik sorban az **I**. sor J. eleme előtt J-1 elem van eltárolva, ez (J-1)*C byte tárolása.

Természetesen megint nem „-1”-ket írtunk, hanem „- első elem sorszáma” formában.

Hézagosan kitöltött mátrixok

A hézagosan kitöltött mátrixok lényege, hogy nagyon kevés (aránytalanul kevés, gyakran kevesebb, mint 10%) eleme van kitöltve a mátrixnak.

Egy NxM mátrix memóriaigénye, melynek minden egyes eleme külön-külön C byte memóriát igényel: N*M*C. Ha ténylegesen csak K elem hordoz értékes információt, akkor ténylegesen csak K*C az információt hordozó memóriamennyiség. Amennyiben K*C <<[3] N*M*C, és N*M*C számottevően nagy tárkapacitást köt le, úgy érdemes elgondolkodni más tárolási elven:

A szabálytalan rendszerességgel kitöltött mátrixok esetén érdemes a mátrixot egy rekord szerkezetű vektorban tárolni, pl. az alábbi módon:

Tegyük fel, hogy a mátrix egyébként valós adattípusú elemeket tárolt volna:

[3] << : jóval kevesebb, számottevően kevesebb

```
MATRIX_REKORD
    X: egész
    Y: egész
    E: valós
REKORD_DEFINICIO_VEGE
T:vektor [1..K] MATRIX_REKORD-ból
```

A T vektor elemei egy-egy rekord, melyek X,Y mező-párja tárolja azt az információt, hogy az „E"-ben tárolt érték a mátrix melyik pozícióján lett volna eltárolva.
Ezen adatszerkezet használatához alprogramokat kell írni, melyeken keresztül megoldható a „mátrix" kezelése, melyek segítségével lekérdezhető egy mátrix-beli elem, és eltárolható a „mátrix"-ban egy új elem.

A módszer hátrányai:

- A mátrix valamely X,Y elemének elérése e pillanattól kezdve egy algoritmuson keresztül történik, amely bináris vagy lineáris keresés segítségével „megnézi" a tárolt vektorban, hogy a keresett elem éppen megvan-e, s ha igen, mennyi az értéke. Mindkét keresésre jellemző, hogy nem minden elem esetén egyforma a keresés ideje, vagyis bizonyos elemeket *hamarabb*, másokat *később* ér el a mátrixon belül. Tehát ezen adatszerkezet elérése már nem *véletlen idejű*.

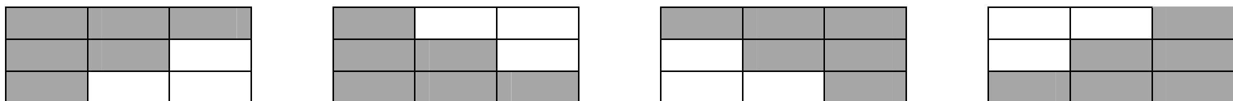
- A másik hátránya, hogy ezen módszerrel maximum K mátrix-beli elem tárolása lehetséges. Amennyiben a mátrix kitöltése dinamikus (a program futása közben keletkeznek a mátrix elemei), úgy elképzelhető, hogy a tárolóhely betelik. *Ennek feloldása a láncolt listák segítségével történhet, melyek mérete a számítógép operatív memóriájának mérete is lehet, de ne felejtsük el, hogy a láncolt listák esetén még a „következő elem" típusú mutatók tárigénye is megnöveli az elemek tárigényét (a pointerek tárigénye általában 4 byte!).*

- A harmadik hátrány, hogy egy mátrixbeli elem tárolása már nem C byte-ba kerül, hanem C', ahol C'=C+2*(egész típus tárigénye). Tehát a N*M*C helyett K*C'. Ezen módszer tehát akkor éri meg, ha a K*C' << N*M*C. Ez gyakran azon múlik, hogy a C' ne legyen lényegesen nagyobb a C-nél. Vagyis ha a mátrix pl. logikai elemeket tárol (melynek tárigénye legtöbb programnyelven 1 byte), s egy egész típus tárigénye 2 byte, ezen módszerrel minden egyes logikai érték eltárolása 5 byte-ba kerül. Könnyen kiszámítható, hogy ezen példa esetén ezzel a módszerrel csak akkor érdemes tárolni a mátrixot, ha K<<N*M/5.

Háromszögmátrixok

A háromszög mátrixok speciális formájú hézagosan kitöltött mátrixok. Specialitásuk abban rejlik, hogy a mátrixnak csak az egyik fele van elemekkel feltöltve. Ez általában több elemet jelent, hogy a a hézagosan kitöltött mátrixok tárolási technikája hatékonyan csökkentse a memóriaigényt. Ezért a programozók lustaságból teljes mátrixot használnak. Ugyanakkor a háromszögmátrixok tárolása megoldható kis memóriafelhasználással, és az elérés sebessége is *véletlen* marad.

A háromszögmátrixok leggyakrabban négyzetes mátrixok.
A háromszögmátrixokból ez alapján négy fajta létezhet:



A háromszög mátrixok optimális kezelése érdekében egy akkora vektort kell deklarálni, amely mérete lehetővé teszi a mátrix értékes elemeinek eltárolását. Nagy könnyebbség, hogy ez háromszögmátrixok estén előre meghatározható: 1..N a számok összege (első sorban 1 elem, második sorban 2 elem, …, N. sorban N elem). Ez pedig N*(N+1)/2.

Tehát az „A: tömb [1..N,1..N] ____-ből" tárolása egy „V:tömb [1..N*(N+1)/2] ____-ből" vektorral oldahtó meg. A vektor elemei folytonosan tárolódnak el a memóriában. Ezzel szemben szükségünk lesz egy címfüggvényre, amely az eredeti A mátrixbeli I,J elemhez hozzárendeli a V valamely elemét.

A címfüggvény annyiban rendhagyó, hogy igazából azt a képletet kell megtalálni, hogy az „A" mátrix I,J eleme a „V" vektor melyik elemére van leképezve. Innentől kezdve a vektor címfüggvénye segítségével megtalálható a vektoron belül ezen elem.

Mind a négy formára található címfüggvény (mindegyikre más), az első típusra pl. az alábbi:

- az egyszerűség kedvéért feltesszük, hogy az mátrix sor és oszlopindexe is 1-től indul
- feltételezzük, hogy az I,J indexpár a felső háromszögmátrix szabályai szerinti megfelelő elemre hivatkozik (J<=N-I+1)

```
VektorIndex( I, J ) =    ( I-1 )*N
                         - ( ( ( ( I-1 )*i) /2)+I-1)
                         + J-1
                         + 1;
```

Az első sor szerint a I. sor előtt ennyi elem lenne eltárolva, ha minden sor teljes sor lenne.
A második sor szerint az I. sor előtt ennyi elem hiányzik a hézagos kitöltés miatt.
A harmadik sor szerint az I. soron belül a J. elem előtt ennyi elem van eltárolva.
A negyedik sor azért ad 1-et az egészhez, mert a vektor első elemének sorszáma nem 0, hanem 1.

A legérdekesebb a második sor. Gondoljuk meg, az első sorból 0 elem, a második sorból 1, a harmadik sorból 2, …, az I. sorból I-1 elem hiányzik. Vagyis ha arra vagyunk kiváncsiak, hogy az első I-1 sorból hány elem hiányzik, akkor először össze kell adni a számokat *1..I-1*-ig, majd mivel soronként egyel kevesebb elem hiányzik, mint a sor indexe, ezért soronként csaltunk, így ezt utólag korrigálni kell. Ha valaki utánaszámol, megkapja a fenti képletet.

A fenti módszerrel, és némi gondolkodással megkapható a maradék három mátrixra is a vektorindex leképező függvény.

Gráfok tárolása

Egy N csomópontból álló gráf tárolása egy NxN mátrix segítségével lehetséges, ahol a sorok és oszlopok találkozásánál van a két szóban forgó csúcs közötti élhez tartozó információ.

- Amennyiben az élek tényleges adatokat hordoznak (pl. a csúcspontok közötti távolságok (pl. városok)), akkor a találkozási ponton a mátrixban ezen adat van eltárolva.

- Amennyiben csak a *„van él közöttük"/"nincs él közöttük"* típusú információt kell eltárolni, úgy a mátrix lehet logikai típusú.

- Amennyiben a gráf nem irányított, úgy a mátrix lehet háromszögmátrix (helytakarékos), hiszen úgyis szimmetrikus a főátlójára nézve.

- Amennyiben a gráf irányított, úgy külön tárolandó az A->B és B->A élek adatai, ezért a mátrix nem (biztos, hogy) szimmetrikus a főátlójára nézve.

- Amennyiben az élhez több információ is tartozik, úgy a mátrix elemei rekord-ok, melyek mezői tárolják az összes szükséges információt.

Mátrixműveletek- Feltöltése elemekkel

Feladat: a mátrix elemeinek bekérése billentyűzetről

```
DEKLARÁCIÓ
        A: tömb [1..N,1..M] egész-ből
AKEZD
        Ciklus I := 1-től N-ig
                Ciklus J:=1-től M-ig
                        A(I,J) elem feltöltése
                CVÉGE
        CVÉGE
AVÉGE
```

Mátrixműveletek- két mátrix összeadása C[ ] := A[ ] + B[ ]

Feladat: előállítani az A+B mátrixot. Ezen mátrixot úgy kapjuk, hogy a megfelelő indexű elemeket összeadjuk, ez képezi az új mátrix elemeit.

```
DEKLARÁCIÓ
        A,B,C: tömb [1..N,1..M] egész-ből
AKEZD
        Ciklus I := 1-től N-ig
                Ciklus J:=1-től M-ig
                        C(I,J) := A(I,J)+B(I,J)
                CVÉGE
        CVÉGE
AVÉGE
```

Mátrixműveletek- két mátrix szorzása C[ ] := A[ ] * B[ ]

Feladat: az A*B mátrix előállítása. A C mátrix I,J elemét úgy kapjuk, hogy az A matrix I sorának minden egyes elemét rendre megszorozzuk a B mátrix J oszlopában levő elemekkel, és képezzük ezen szorzat-elemek összegét.

```
DEKLARÁCIÓ
        A: tömb [1..M,1..N] egész-ből
        B: tömb [1..N,1..K] egész-ből
        C: tömb [1..M,1..K] egész-ből
AKEZD
        Ciklus I := 1-től M-ig
                Ciklus J:=1-től K-ig
                        Szorzat := 0
                        Ciklus L := 1-től N-ig
                                Szorzat := Szorzat+A(I,L)*B(L,J)
                        CVÉGE
                        C(I,J) := Szorzat
                CVÉGE
        CVÉGE
AVÉGE
```

Mátrixműveletek- mátrix transzponálása – tükrözés a főátlóra

Feladat: a négyzetes mátrix elemeinek tükrőzése a főátlóra

```
DEKLARÁCIÓ
        A: tömb [1..N,1..N]
AKEZD
        Ciklus I := 1-től N-ig
                Ciklus J:=1-től I-1-ig
                        C := A(I,J)
                        A(I,J) := A(J,I)
                        A(J,I) := C
                CVÉGE
        CVÉGE
AVÉGE
```

Rekurzió

Rekurzióról akkor beszélünk, ha egy eljárás **saját magát hívja** újra.

- Amennyiben az eljárás törzsében tényleges szerepel saját magának ismételt indítása, akkor **közvetlen rekurzióról** beszélünk.
- Amennyiben egy „A" eljárás egy „B" eljárást hív, amely meghívja az „A" eljárást, akkor **közvetett rekurzióról** beszélünk.

Sokkal gyakoribb a közvetlen rekurzió, ráadásul az itt ismertetett elvek igazak a közvetett rekurzióra is, ezért most lényegében csak a közvetlen rekurzióról fogunk beszélni.

Ha egy eljárás elindul, akkor mindig valamennyi memória lekötésével jár. A számítógépnek el kell tárolni, hogy melyik eljárásból, azon belül melyik pontjáról hívódott meg az adott eljárás – hogy annak befejeztével a hívó pont utáni utasítás végrehajtásával tudja folytatni a program végrehajtását. A memóriakapacitás mindig véges, ezért a rekurzió csak bizonyos mélységig folytatható, ezek után a program futási hibával (memória elfogyott) le fog állni. Másrészről végtelen mélységig semmiképpen sem folytatható a rekurzív hívások ismétlése, hiszen az algoritmus alaptulajdonsága a végesség – vagyis véges lépésben be kell fejeződjön a futása.

A rekurzív algoritmusok mindig tartalmaznak egy feltételt (**bázisfeltétel**), mely ha teljesül, akkor a rekurzió befejeződik, az eljárás ekkor már nem hívja meg saját magát, hanem visszatér, s e miatt már minden szint elkezd visszatérni, s előbb-utóbb visszatér az eredeti kiinduló pontra (pl. a főprogramba).

A rekurzió során ezen bázisfeltételt közelíteni kell. Amennyiben a program nem közeledik ezen feltétel teljesüléséhez, akkor nem lenne semmi, amely megállítja a rekurzió ismétlését, s a program az előbb ismertetett probléma miatt futási hibával leállna.

A rekurzió ezen viselkedési formája miatt sokban emlékeztet a ciklusokra, ahol a bázisfeltétel analóg a ciklus vezérlő feltételével, s a ciklus magban is lennie kell olyan résznek, amely kihatással van a ciklus vezérlő feltételére – hogy az előbb-utóbb logikai előjelet váltva befejezze a ciklusmag ismételt végrehajtását.

Valójában a fenti analógia kihasználása mellett a ciklusok mindig átírhatóak rekurzióra, s viszont!

Nézzünk példákat:

Rekurzió - hatványozás

Pl. hatványozás: egy „A" szám „n" hatványa az alábbi forma szerint határozható meg:

$A^0=1$

$A^n=A^{n-1}*A$, ha n>0

Ez a matematikai stílusú definíció is rekurzív definíció, hiszen A^n meghatározása során kihasználjuk is a hatványozást használjuk fel.

A fenti definíció értelmében A=5, n=3 esetén az alábbi módon kaphatjuk meg az 5^3 értékét:
Mivel 3>0, ezért a definíció második formáját használjuk: $5^3=5^{3-1}*5=5^2*5$
Még nem tudjuk, mennyi 5^2, de a definíció szerint $5^2=5^{2-1}*5=5^1*5$
Még nem tudjuk, mennyi 5^1, de a definíció szerint $5^1=5^{1-1}*5=5^0*5$
A definíció szerint bármely A szám 0. hatványa 1, tehát tudjuk, hogy $5^0=1$.
Ennek segítségével visszafele helyettesítve megkaphatjuk az eredeti kérdés, 5^3 értékét is.

Látszik a rekurzió? Mindig az A^n definícióját használtuk fel egy A^n érték kiszámításához. Ugyanakkor folyamatosan közeledtünk a bázisfeltételhez, hogy n=0 esetén az eredmény fixen 1.

Ha valaki azt a feladatot kapná, hogy írjon algoritmust, amely tetszőleges „A" és „n" esetén meghatározza A^n értékét, igazából nem a hatványozás fenti definícióját használná fel, hanem az „*empirikus*" ismeretet, mely szerint A^n nem más, mint A*A*A*…*A, ahol a szorzatban az „A" n-szer szerepel.

```
Eredmeny := 1
Ciklus I:=1-től n-ig
  Eredmeny := Eredmeny * A
CVÉGE
Ki: Eredmeny
```

Amennyiben az eredeti, matematikai definíciót használjuk fel, az alábbi függvény megírására kerül sor:

```
1. Függvény Hatvany(A:valós;N:egész):valós
2. FKEZD
3.   HA N=0  AKKOR
4.      Hatvany := 1
5.   KÜLÖNBEN
6.      Hatvany := Hatvany(A,N-1) * A
7.   HVÉGE

9.
10.      AKEZD
11.         Ki:  Hatvany(5,3)
12.      AVÉGE
```

Mint látjuk, a függvény törzse lényegében megfelel a matematikai definíciónak. Ugyanakkor próbáljuk meg értelmezni a működését:

A program végrehajtása a 11 sorban kezdődik, elindítja a hatvány c. függvényt, átadva neki érték szerint az A-ba az 5, N-be a 3 értéket. Ekkor a memóriában létrejön két változó, A és N névvel, és rendre 5 és 3 kezdőértékkel. Ekkor indul el „először" a `Hatvany` nevű függvény:

1. indítás (11.sorból)	A=5	N=3

A1: A függvény indulásakor létrejön az A és N változók, rendre 5 és 3 kezdőértékkel. A függvény 3. sorában a feltétel nem teljesül (hiszen N=3 <> 0), ezért a KÜLÖNBEN ágra kerül a végrehajtás. A 6. sor szerint a visszatérési értéket úgy kell kiszámolni, hogy *valamit* meg kell szorozni A-val. Az, hogy mennyi az A, azt a gép *tudja*, A=5. A *valami* azonban *Hatvany(A,N-1)*. Ezt a gép nem tudja mennyi, de tudja róla, hogy ez egy függvényhívás (hiszen van ilyen nevű függvény, a paraméterezése stimmel) tehát elindítja ezen függvényt, hogy be tudja fejezni a kifejezés kiértékelését. *Az, hogy e közben is a Hatvany nevű függvényen belül van, a számítógépet cseppet sem zavarja ☺.* Ekkor indul el `Hatvany` másodszor, átadva neki az aktuális paraméterlista szerint a 5 és 2 értékeket.

1. indítás (11.sorból)	A=5	N=3
2. indítás (6.sorból)	A=5	N=2

A2: A függvény indulásakor újra létrejön az A és N változók, rendre 5 és 2 kezdőértékkel. A függvény ezen elindított „példánya” mit sem tud az előző állapotról, az ős számára az A és N változók az ő számára létrehozott új változók lesznek, ezért ő számára az A=5, N=2! Ezért a 3. sorban a feltétel megint nem teljesül, N=2 <> 0, ezért újra a KÜLÖNBEN ágra ugrik, ahol is megpróbálja kiszámolni a visszatérési értékhez a kifejezést. Azonban a HATVANY(A,N-1) számára csak egy függvényhívás, ezért újra elindítja a Hatvany nevű függvényt az új aktuális paraméterlista szerint 5 és 1 értékekkel.

1. indítás (11.sorból) A=5 N=3
2. indítás (6.sorból) A=5 N=2
3. indítás (6.sorból) A=5 N=1

A3: A függvény indulásakor újra létrejön az A és N változók, rendre 5 és 1 kezdőértékkel. A függvény ezen elindított „példánya” mit sem tud az előző állapotról, az ős számára az A és N változók az ő számára létrehozott új változók lesznek, ezért ő számára az A=5, N=1! Ezért a 3. sorban a feltétel megint nem teljesül, N=1 <> 0, ezért újra a KÜLÖNBEN ágra ugrik, ahol is megpróbálja kiszámolni a visszatérési értékhez a kifejezést. Azonban a HATVANY(A,N-1) számára csak egy függvényhívás, ezért újra elindítja a Hatvany nevű függvényt az új aktuális paraméterlista szerint 5 és 0 értékekkel.

1. indítás (11.sorból) A=5 N=3
2. indítás (6.sorból) A=5 N=2
3. indítás (6.sorból) A=5 N=1
4. indítás (6.sorból) A=5 N=0

A4: A függvény indulásakor újra létrejön az A és N változók, rendre 5 és 0 kezdőértékkel. A függvény ezen elindított „példánya” mit sem tud az előző állapotról, az ős számára az A és N változók az ő számára létrehozott új változók lesznek, ezért ő számára az A=5, N=0! A 3. sorban a feltétel ezért most teljesül, N=0 = 0, ezért az AKKOR ágra lép, ahol meghatározza a visszatérési értéket – ez pedig 1. Ezek után a HVÉGE utáni utasításra lép, amely a függvény végét jelzi. Ekkor a memóriából eltűnik ezen függvény-példány számára létrehozott lokális változók: A és N. Majd a program visszatér oda, ahonnan hívták: az előző példány 6. sorára, és folytatja azon példány futását… (*Fv visszatérési érték 1*).

1. indítás (11.sorból) A=5 N=3
2. indítás (6.sorból) A=5 N=2
3. indítás (6.sorból) A=5 N=1

A3: A számítógép végre meghatározta, hogy mennyi is Hatvany(A,N-1), ezért végre be tudja fejezni a Hatvany(A,N-1)*A kifejezés kiértékelését, ez 5. Ezek után (mivel a KÜLÖNBEN ágon már nincs más utasítás), a HVÉGE utáni sorra ugrik. Ezen sor a függvény végét jelzi. Ekkor a memóriából eltűnik ezen függvény-példány számára létrehozott lokális változók: A és N. Majd a program visszatér oda, ahonnan hívták: az előző példány 6. sorára, és folytatja azon példány futását… (*Fv visszatérési érték 5*).

1. indítás (11.sorból) A=5 N=3
2. indítás (6.sorból) A=5 N=2

A2: A számítógép végre meghatározta, hogy mennyi is Hatvany(A,N-1), ezért végre be tudja fejezni a Hatvany(A,N-1)*A kifejezés kiértékelését, ez 25. Ezek után (mivel a KÜLÖNBEN ágon már nincs más utasítás), a HVÉGE utáni sorra ugrik. Ezen sor a függvény végét jelzi. Ekkor a memóriából eltűnik ezen függvény-példány számára létrehozott lokális változók: A és N. Majd a program visszatér oda, ahonnan hívták: az előző példány 6. sorára, és folytatja azon példány futását… (*Fv visszatérési érték 25*).

1. indítás (11.sorból) A=5 N=3

A1: A számítógép végre meghatározta, hogy mennyi is Hatvany(A,N-1), ezért végre be tudja fejezni a Hatvany(A,N-1)*A kifejezés kiértékelését, ez 125. Ezek után (mivel a KÜLÖNBEN ágon már nincs más utasítás), a HVÉGE utáni sorra ugrik. Ezen sor a függvény végét jelzi. Ekkor a memóriából

eltűnik ezen függvény-példány számára létrehozott lokális változók: A és N. Majd a program visszatér oda, ahonnan hívták: a főprogram 11. sorára, és folytatja a főprogram futását… (*Fv visszatérési érték 125*).

A0: A főprogram végre megtudta, mennyi Hatvany(5,3), ez 125, ezt az értéket kiírja a képernyőre.

Rekurzió - faktoriális

Nézzünk másik példát: faktoriális kiszámítása.
A faktorizálás jele a felkiáltójel (!). Az empirikus definíció szerint a faktorizálás azt jelenti, hogy össze kell szorozni a számokat 1..N-ig. Pl. 4!=1*2*3*4

A matematikai definíció szerint:

0! = 1
N! = (N-1)!*N, ha N>0

A definíció értelmezését, és használatával a faktor kiszámítását hasonlóan kell végezni, mint a hatványozás esetén.

Hogyan írunk erre iterációs algoritmust?

```
Eredmeny := 1
Ciklus I:=1-től n-ig
  Eredmeny := Eredmeny * I
CVÉGE
Ki: Eredmeny
```

Hogyan írunk erre rekurzív programot?

```
1. Függvény Faktor(N:egész):valós
2. FKEZD
3.   HA N=0  AKKOR
4.      Faktor  := 1
5.   KÜLÖNBEN
6.      Faktor:= Faktor(N-1) * N
7.   HVÉGE
8. FVÉGE
9.
10.      AKEZD
11.         Ki:  Faktor(3)
12.      AVÉGE
```

Elemezzük ki a futást (kissé gyorsított módon):

A0: A program a 11. soron kezdődik, elindítva a Faktor nevű fv-t a 3 aktuális értékkel.

1. indítás (11.sorból)	N=3

A1: A függvény indulásakor létrejön az N változó, 3 kezdőértékkel. A függvény 3. sorában a feltétel nem teljesül (hiszen N=3 <> 0), ezért a KÜLÖNBEN ágra kerül a végrehajtás. A 6. sor szerint a visszatérési értéket úgy kell kiszámolni, hogy *valamit* meg kell szorozni N-el. Elindítja a Faktor c. függvényt N-1, azaz 2 aktuális értékkel.

1. indítás (11.sorból)	N=3
2. indítás (6.sorból)	N=2

A2: A függvény indulásakor (újra) létrejön az N változó, 2 kezdőértékkel. A függvény 3. sorában a feltétel nem teljesül (hiszen N=2 <> 0), ezért a KÜLÖNBEN ágra kerül a végrehajtás. A 6. sor

szerint a visszatérési értéket úgy kell kiszámolni, hogy *valamit* meg kell szorozni N-el. Elindítja a Faktor c. függvényt N-1, azaz 1 aktuális értékkel.

1. indítás (11.sorból) N=3
2. indítás (6.sorból) N=2
3. indítás (6.sorból) N=1

A3: A függvény indulásakor (újra) létrejön az N változó, 1 kezdőértékkel. A függvény 3. sorában a feltétel nem teljesül (hiszen N=1 <> 0), ezért a KÜLÖNBEN ágra kerül a végrehajtás. A 6. sor szerint a visszatérési értéket úgy kell kiszámolni, hogy Faktor(N-1)-t meg kell szorozni N-el. Elindítja a Faktor c. függvényt N-1, azaz 0 aktuális értékkel.

1. indítás (11.sorból) N=3
2. indítás (6.sorból) N=2
3. indítás (6.sorból) N=1
4. indítás (6.sorból) N=0

A4: A függvény indulásakor (újra) létrejön az N változó, 0 kezdőértékkel. A függvény 3. sorában a feltétel teljesül (hiszen N=0 = 0), ezért az AKKOR ágra kerül a végrehajtás. A 4. sor szerint a visszatérési érték 1. Ezek után a HVÉGE, majd az EVÉGE. A fv ezen példánya befejezi a futását, a számára készült lokális változók megszűnnek, és visszatér a hívó helyre. (*visszatérési érték 1).*

1. indítás (11.sorból) N=3
2. indítás (6.sorból) N=2
3. indítás (6.sorból) N=1

A3: A 6. sorban szereplő kifejezés kiértékelése tovább folyik. Az első fele ezek szerint 1, a második fele jelenleg 1 (ezen fv példány számára N=1 még mindig), ezért a kifejezés kiszámítva 1. Ezek után a HVÉGE, majd az EVÉGE. A fv ezen példánya befejezi a futását, a számára készült lokális változók megszűnnek, és visszatér a hívó helyre. (*visszatérési érték 1).*

1. indítás (11.sorból) N=3
2. indítás (6.sorból) N=2

A2: A 6. sorban szereplő kifejezés kiértékelése tovább folyik. Az első fele ezek szerint 1, a második fele jelenleg 2 (ezen fv példány számára N=2 még mindig), ezért a kifejezés kiszámítva 2. Ezek után a HVÉGE, majd az EVÉGE. A fv ezen példánya befejezi a futását, a számára készült lokális változók megszűnnek, és visszatér a hívó helyre. (*visszatérési érték 2).*

1. indítás (11.sorból) N=3

A1: A 6. sorban szereplő kifejezés kiértékelése tovább folyik. Az első fele ezek szerint 2, a második fele jelenleg 3 (ezen fv példány számára N=3 még mindig), ezért a kifejezés kiszámítva 6. Ezek után a HVÉGE, majd az EVÉGE. A fv ezen példánya befejezi a futását, a számára készült lokális változók megszűnnek, és visszatér a hívó helyre. (*visszatérési érték 6).*

A0: A 11. sorban szereplő kifejezés értéke már ismert, a képernyőre kiíródik a 6.

Ezen példában ami érdekes, hogy a kifejezés második fele N, ami minden példányban más és más értéket képvisel, hiszen minden példány számára külön N lokális változó áll rendelkezésre.

Ha jól megfigyeltük, a rendszer viselkedése sokban hasonlít a verem használatára. Amikor újabb függvény hívódik meg, akkor a verem tetejére kerül egy újabb „egységcsomag", mely tartalmazza, hogy melyik sorra kell visszatérni a kilépés után, és melyik lokális változónak mennyi az aktuális értéke. Amikor az függvény befejezi a futását, eltávolítja a verem tetején lévő elemet (ekkor szűnnek meg a lokális változók), és visszatér oda, ahol a verem tetején lévő „egységcsomag" adja. Valójában ez több mint hasonlatosság – a számítógép ilyenkor pontosan vermet használ ezen működés megvalósítására. Márpedig a verem véges kapacitású. De hát fel is hívtuk a figyelmet, hogy a rekurziót nem lehet, csak adott sokszor ismételve végrehajtani!

Rekurzió – Fibonacci számok

A harmadik iskolapéldája a rekurziónak a Fibonacci számsorozat. Ezen számsorozat első két eleme 1, a harmadik elem az 1. és 2. elem összege (tehát 2), a negyedik elem a 2. és 3. összege (tehát 3). Általában az elemeket az előző két elem összegeként kell kiszámítani.

Fibonacci számsorozat első elemei: 1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,…
Mint látjuk, a sorozat elemeinek értéke rohamosan nő.

Feladat: számítsuk ki a Fibonacci számsorozat n. elemét (n>0).
Hogyan lehet ezt iterációval kiszámítani?

```
DEKLARÁCIÓ
   fib1,fib2,x,I : egész

  fib1 :=0;
  fib2 :=1;
  Ciklus I :=1-től N-ig
      x:=fib1+fib2;
      fib1:=fib2;
      fib2:=x;
  CVÉGE
  Ki: fib1
```

Ez csak egy megoldás a sok közül, lehet máshogy is kiszámítani.

Nézzük, hogyan néz ki a Fibonacci sorozat matematikai definíciója:

1. Fibonacci szám = 1
2. Fibonacci szám = 1

n. Fibonacci szám = (n-1). Fibonacci szám + (n-2). Fibonacci szám, ha n>2

Ennek alapján a rekurzív fv:

```
1.  Függvény Fibonacci(N:egész):egész
2.  FKEZD
3.    HA N<3 AKKOR
4.          Fibonacci:=1
5.    KÜLÖNBEN
6.          Fibonacci := Fibonacci(N-1) + Fibonacc(N-2)
7.    HVÉGE

9.
10.       AKEZD
11.          Ki:  Fibonacci(4)
12.       AVÉGE
```

A 10. sorban szereplő kifejezés kiszámítása most már csak egyetlen apró meglepetést tartogat a számunkra: kövessük végig!

A0: A program a 11. soron kezdődik, elindítva a Fibonacci nevű fv-t a 4 aktuális értékkel.

A0 - 1. indítás (11.sorból) N=4

A1: A függvény indulásakor létrejön az N változó, 4 kezdőértékkel. A függvény 3. sorában a feltétel nem teljesül (hiszen N=4 < 3 hamis), ezért a KÜLÖNBEN ágra kerül a végrehajtás. A 6. sor szerint a visszatérési értéket úgy kell kiszámolni, hogy *valamit* meg kell szorozni *valamivel*-el. A kifejezés első felének kiértékelése végett elindítja a Fibonacci nevű fv-t N-1, azaz 3 aktuális értékkel.

A0 - 1. indítás (11.sorból)   N=4
A1 - 2. indítás (6.sor első feléből)   N=3

A2: A függvény indulásakor létrejön az N változó, 3 kezdőértékkel. A függvény 3. sorában a feltétel nem teljesül (hiszen N=3 < 3 hamis), ezért a KÜLÖNBEN ágra kerül a végrehajtás. A 4. sor szerint a visszatérési értéket úgy kell kiszámolni, hogy *valamit* meg kell szorozni *valamivel*-el. A kifejezés első felének kiértékelése végett elindítja a Fibonacci nevű fv-t N-1, azaz 2 aktuális értékkel.

A0 - 1. indítás (11.sorból)   N=4
A1 - 2. indítás (6.sor első feléből)   N=3
A2 - 3. indítás (6.sor első feléből)   N=2

A3: A függvény indulásakor létrejön az N változó, 2 kezdőértékkel. A függvény 3. sorában a feltétel teljesül (hiszen N=2 < 3 igaz), ezért az AKKOR ágra kerül a végrehajtás. A 4. sor szerint a visszatérési érték 1. HVÉGE. FVÉGE. A fv ezen példánya befejezi a futását, a számára készült lokális változók megszűnnek, és visszatér a hívó helyre. (*visszatérési érték 1).*

A0 - 1. indítás (11.sorból)   N=4
A1 - 2. indítás (6.sor első feléből)   N=3

A2: A számítógép végre tudja, hogy a 6. sorban feltüntetett kifejezés első fele 1, de második felét még nem tudja mennyi. Ezért a második felének kiszámítása végett újra elindítja a Fibonacci nevű függvényt N-2, azaz 1 aktuális értékkel.

A0 - 1. indítás (11.sorból)   N=4
A1 - 2. indítás (6.sor első feléből)   N=3
A2 - 3. indítás (6.sor második feléből)   N=1

A3: A függvény indulásakor létrejön az N változó, 1 kezdőértékkel. A függvény 3. sorában a feltétel teljesül (hiszen N=1 < 3 igaz), ezért az AKKOR ágra kerül a végrehajtás. A 4. sor szerint a visszatérési érték 1. HVÉGE. FVÉGE. A fv ezen példánya befejezi a futását, a számára készült lokális változók megszűnnek, és visszatér a hívó helyre. (*visszatérési érték 1).*

A0 - 1. indítás (11.sorból)   N=4
A1 - 2. indítás (6.sor első feléből)   N=3

A1: A számítógép végre tudja, hogy a 6. sorban feltüntetett kifejezés első fele 2. A második felének kiszámítása végett újra elindítja a Fibonacci c. függvényt N-2, azaz 2 értékkel.

A0 - 1. indítás (11.sorból)   N=4
A1 - 2. indítás (6.sor második feléből)   N=2

A2: A függvény indulásakor létrejön az N változó, 2 kezdőértékkel. A függvény 3. sorában a feltétel teljesül (hiszen N=2 < 3 igaz), ezért az AKKOR ágra kerül a végrehajtás. A 4. sor szerint a visszatérési érték 1. HVÉGE. FVÉGE. A fv ezen példánya befejezi a futását, a számára készült lokális változók megszűnnek, és visszatér a hívó helyre. (*visszatérési érték 1).*

A0 - 1. indítás (11.sorból)   N=4

A1: A számítógép végre tudja, hogy a 6. sorban feltüntetett kifejezés első fele 2, a második fele pedig 1. A kifejezés értéke tehát 3. HVÉGE. FVÉGE. A fv ezen példánya befejezi a futását, a számára készült lokális változók megszűnnek, és visszatér a hívó helyre. (*visszatérési érték 3).*

A0: A számítógép kiírja, hogy a 4. Fibonacci szám értéke 3.

A végrehajtás érdekessége, hogy N>2 esetén a kifejezés mindkét tagjára elindul rekurzívan a kiszámítás.

Rekurzió – “Festő algoritmus”

Gyakori probléma, hogy egy mátrix valamely pontjából elindulva be kell járni a környező elemeket (majd azok környező elemeit) mindaddig, amíg környező elemek megfelelnek valamely T tulajdonságnak. A tipikus példa, ahonnan az algoritmus neve is ered – a grafikus képernyőn valamely színű területet át kell színezni – elindulván a terület valamely belső pontjából.

Az algoritmus rekurzív megoldása egyszerűsége és szépsége miatt érdemel említést, de fontos tudni, hogy ezen megoldás a *befestés* során nagyon sok memóriát köt le, ezért nagy terület átszínezése esetén bizony kifogyhatunk a memóriából (és *stack overflow* hibaüzenettel a program leáll).

```
1.   T:tömb [1..N,1..M] egész-ből
2.
3.   ELJÁRÁS Befest(X,Y:egész)
4.   EKEZD
5.      T[X,Y] := ÚjSzín
6.      HA X>1 és T[X-1,Y]=RÉGISZIN AKKOR Befest(X-1,Y)
7.      HA X<N és T[X+1,Y]=RÉGISZIN AKKOR Befest(X+1,Y)
8.      HA Y>1 és T[X,Y-1]=RÉGISZIN AKKOR Befest(X,Y-1)
9.      HA Y<M és T[X,Y+1]=RÉGISZIN AKKOR Befest(X,Y+1)
10.  EVÉGE
```

Az egyszerűség kedvéért, mivel az AKKOR ágon mindig csak egy utasítás (a rekurzív hívás) szerepel, ezért eltekintettünk a feltételek teljes kiírásának formájától. A

```
HA X>1 és T[X-1,Y]=RÉGISZIN AKKOR Befest(X-1,Y)
```

helyes írása módja:

```
HA X>1 és T[X-1,Y]=RÉGISZIN AKKOR
      Befest(X-1,Y)
HVÉGE
```

A fenti algoritmus az átvett X,Y pont színét az „Új Szín”-re állítja, majd ezen pont 4 irányban elhelyezkedő szomszédjait ellenőrzi, és amerre még a „Régi Szín” van, azon koordinátákra meghívja saját magát.

Ezen algoritmus segítségével sok olyan probléma is megoldható, amelyről első pillantásban ez nem látszik egyértelműen:

- A Windows-ból jól ismert „Aknakereső” program lelke ez az algoritmus.
- Tegyük fel, hogy egy NxM-es logikai típusú mátrix elemei közül néhány „igaz”, mások „hamis”. Az „igaz” a padlót jelenti, a „hamis” egy szakadékot. Kérdés, hogy a mátrix bal széléről át lehet-e jutni a jobb szélére úgy, hogy csak „igaz” elemeken *lépkedünk*. Egy mezőről csak a 4 szomszédos elem valamelyikére léphetünk át.
- Egy fenti típusú mátrixban az „igaz” elem jelzi a padlót, a „hamis” a falat. Az így adott *labirntus* egyik pontjából el lehet-e jutni egy másik pontba (pl. a kijárathoz). *Labirintus bejárási problémakör!*
- Egy hegység fölé egy 1m x 1m négyzethálót borítunk, és egy NxM mátrixban eltároljuk a négyzeteken belüli – tengerszint fölötti mért magasságokat. Kérdés, hogy a hegység egy adott X,Y csúcsát körbe lehet-e járni folyamatosan (pl.) 500-800 méter magasságon belül maradva?

Quick-sort rendezés:

```
ELJÁRÁS GyorsRendezes(CÍM A:tömb [1..N] valós-ból, Also,Felso: egész)
   I,J : egész
   X   : valós
EKEZD
   I  := Also
   J := Felso
   CIKLUS
        X := A[ (Also+Felso)/2 ]
        CIKLUS AMIG A[ I ]  < X
            I := I + 1
        CVÉGE
        CIKLUS AMIG A[ J ]  > X
            J := J - 1
        CVÉGE
        HA I<J AKKOR
            CSERE: A[I], A[J]
        HVÉGE
        HA I<=J AKKOR
            I := I+1
            J := J-1
        HVÉGE
   CVÉGE_HA I>J
   HA Also < J AKKOR
        GyorsRendezes(A,Also,J)
   HVÉGE
   HA I < Felso AKKOR
        GyorsRendezes(A,I,Felso)
   HVÉGE
EVÉGE
```

Unióképzés tétele

```
AKEZD
  Ciklus I:=1-től N-ig
     Z[I]:=X[I]
  Ciklus vége
  Db:=N
  Ciklus J:=1-től M-ig
     I:=1
     Ciklus amíg I<=N és X[I]<>Y[J]
        I:=I+1
     Ciklus vége
     Ha I>N akkor
        Db:=Db+1
        Z[Db]:=Y[J]
     Ha vége
  Cvége
AVÉGE
```

Metszetképzés tétele

```
AKEZD
   Db:=0
   Ciklus J:=1-től M-ig
       I:=1
       Ciklus amíg I<=N És X[I]<>Y[J]
             I:=I+1
       Ciklus vége
       Ha I<=N akkor
           Db:=Db+1
           Z[Db]:=Y[J]
       Ha vége
   Cvége
AVÉGE
```

Visszalépéses keresés általános algoritmusa

Adva van N sorozat, melyek akár eltérő hosszúak is lehetnek. Az M[1..N] vektor tárolja, hogy melyik sorozat hány elemből áll. Feladat: mindegyik sorozatból kiválasztani 1 elemet úgy, hogy a kiválasztott elemek ne kerüljenek ellentmondásba egymással.

Megoldás: az X[1..N] vektorba kerülnek a kiválasztott elemek sorszámai.

Inicializázálás:

```
        X [ 1 ] := 0
```

Vezérlő ciklus:

```
        I := 1
        Ciklus amíg 1<=I és I<=N
                HA Van_Jó_Eset(I)  AKKOR
                        I:=I+1
                        X[ I ]:=0
                KÜLÖNBEN
                        I:=I-1
                HVÉGE
        CVÉGE
        HA I>N AKKOR
                VAN_MEGOLDÁS
        KÜLÖNBEN
                NINCS_MEGOLDÁS
        HVÉGE
```

FÜGGVÉNY Van_Jó_Eset(I): logikai

```
FKEZD
        ISMÉTELD
                X [ I ] := X [ I ] +1
        AMÍG X [ I ] <= M [ I ]  és Rossz_Eset( I, X [ I ] )
        Van_Jó_Eset :=  ( X [ I ] <= M [ I ] )
FVÉGE
```

FÜGGVÉNY Rossz_Eset (I, X[I]) : logikai

```
  J : egész
FKEZD
        J := 1
        Ciklus amíg J<I    és   J,X(J) nem zárja ki I,X(I)-t
                J := J + 1
        CVÉGE
        Rossz_Eset := J < I
FVÉGE
```

8 vezérre specializálva a feladatot:
Adva van egy sakktábla, melyre 8 királynőt kell felrakni úgy, hogy a királynők ne üssék egymást. *E miatt minden sorba csak egy királynő tehető fel. A királynők akkor ütik egymást, ha egy oszlopba kerülnek, vagy egy másik királynő mezőjéről kiinduló átlón helyezkednek el.*

8 db sorozatunk van (N=8) (*egy sakktáblán 8 sor van)*, mindegyik sorozat 1..8 tartalmaz számokat (a sakktáblán 8 oszlop van, és a sorozatok lényegében az oszlopok sorszámait tartalmazzák). Mivel minden sorozat egyenlő hosszú, ezért M[I]=8 minden elemre.. Minden sorozatból egyetlen számot kell kiválasztani úgy, hogy ha az **I**. sorozatból az X(I) értéket választjuk ki, az azt jelenti, hogy az **I**. sorban az X(I). oszlopra teszünk egy királynőt.

Az általános algoritmusban a „*J<I és J,X(J) nem zárja ki I,X(I)*”-t az alábbi logikai kifejezésre kell cserélni: **J<I és X(I)<>X(J) és |X(I)-X(J)|<>I-J**

Egy megoldás pl.:1,7,5,8,2,4,6,3

Lengyel formula – lengyel formára alakítás algoritmusa.

Az algoritmusok írása során sokszor fordul elő, hogy kifejezéseket írunk, pl. numerikus kifejezéseket. Ezen kifejezések kiértékelése igen bonyolult feladat – tartalmaz különböző prioritású műveleti jeleket, opernadusokat, és a prioritásokat felülbíráló zárójeleket. A műveleti jelek az ún. infix jelölés szabályai szerint az operandusok *között* foglalnak helyet, a kiszámítás során egy zárójelen belül sem a balról jobbra szabály a mérvadó, hanem figyelni kell a prioritásokat.

E formájában a kifejezés kiértékelése igen bonyolult feladat. A megoldás a kifejezés egy másik formára hozása segíti elő. Ezen forma a postfix forma – más néven fordított lengyel forma.

A postfix jelölés szabályai szerint a műveleti jeleket az operandusaik *után* kell írni, és ezen alakban <u>mindig a balról-jobbra</u> kiértékelési sorrendet kell használni. Nincs sem zárójel, sem prioritás.

Pl:

Infix: `((A-B*2)*(C+D))/(X+Y)`
Postfix: `A B 2 * - C D + * X Y + /`

Az infix szerint először az B*2 –t kell kiszámítani, majd az A-részeredmény, majd a C+D –t, a két részeredményt összeszorozni, kiszámítani az X+Y –t, és a két utolsó részeredményt elosztani egymással. E közben persze folyamatosan figyelni a zárójelekre (jelen

1. Postfix: `A B 2 * - C D + * X Y + /`
   A postfix alak szerint balról az első műveleti jel a *. Ez egy kétoperandusú műveleti jel – tehát két operandus kell hozzá. Ezen két operandus a tőle balra levő két operandus lesz, így a B és a 2. Számoljuk ki ezen kifejezés értékét, és a részeredményt tegyük be a felhasznált „jelek” helyére:
2. Postfix: `A „B*2” - C D + * X Y + /`
   A forma szerint az első műveleti jel a -. Ez egy kétoperandusú művelet, vegyük ki a tőle balra eső két operandust, az A-t és a *B*2* részeredményt. Végezzük el közöttük a műveletet, és a felhasznált jelek helyére tegyük vissza.
3. Postfix: `„B*2-A” C D + * X Y + /`
   A kifejezés jelen formája szerint a balról első műveleti jel a +. Vegyük ki a tőle balra levő két operandust (C és D). végezzük el a műveletet, és tegyük vissza a részeredményt.
4. Postfix: `„B*2-A” „C+D” * X Y + /`
   A kifejezés jelen formája szerint a balról első műveleti jel a *. Vegyük ki a tőle balra levő két operandust (C+D és B*2-A), végezzük el a műveletet, és tegyük vissza a részeredményt.
5. Postfix: `„B*2-A * C+D” X Y + /`
   A kifejezés jelen formája szerint a balról első műveleti jel a +. Vegyük ki a tőle balra levő két operandust (X és Y), végezzük el a műveletet, és tegyük vissza a részeredményt.

6. `Postfix:` *„B*2-A * C+D” „X+Y”* /

   A kifejezés jelen formája szerint a balról első műveleti jel a /. Vegyük ki a tőle balra levő két operandust(a két maradék részeredményt), végezzük el a műveletet, és tegyük vissza a részeredményt.

7. `Postfix:` *„B*2-A * C+D* / X+Y”

   A kifejezés jelen formája szerint nincs benne műveleti jel. A kifejezés jelen állapota a végeredmény.

Amennyiben A=4, B=1, C=6, D=10, X=3, Y=5, úgy a kiértékelés közben a postfix forma a következőképen alakul:

Postfix:

```
A B 2 * - C D  + * X Y +  /
4 1 2 * - 6 10 + * 3 5 +  /
4   2   - 6 10 + * 3 5 +  /
    2     6 10 + * 3 5 +  /
    2       16   * 3 5 +  /
         32        3 5 +  /
         32          8    /
                 4
```

Vagyis a kifejezés végeredménye **4**. Amennyiben az infix formát számoljuk ki a szokásos módon, ugyanezen eredményt kapjuk.
A probléma tehát kettős: ismert egy infix formájú kifejezés (forrás), mely tartalmaz(hat) zárójelezést, és különböző prioritású operátorokat.

- ezen kifejezést postfix alakra kell hozni
- a postfix alakot ki kell értékelni

Bár a probléma kettős, mégis ezen két lépést leprogramozni mégis egyszerűbb, mint közvetlenül az infix alakot kiértékelni.

A probléma jelentősége adott – minden programozási nyelv fordítóprogramja tartalmazza ezen algoritmuspárt, hiszen a forráskódban a programozók infix alakban írják a kifejezést – a program futása során persze ezen kifejezéseket ki kell értékelni.

A fordított Lengyel formára hozás algoritmusa az alábbi:

(Prioritások – pl.)

Nyitó zárójel	0
Üres verem	0
+ -	1
* /	2
Hatványozás	3

```
CIKLUS AMÍG van elem a forrásban
    Következő elem beolvasása a forrásból X-be
    ELÁGAZÁS
      HA X egy operandus
            EREDMENY := EREDMENY + X
      HA X egy nyitó zárójel
            VEREMBE ( X )
      HA X egy záró zárójel
            CIKLUS AMÍG a verem teteje nem nyitó zárójel
                    KIVESZ VEREMBŐL Y-ba
                    EREDMENY := EREDMENY + Y
            CVEGE
            KIVESZ VEREMBŐL Y-ba (nyitó zárójel)
      HA X egy műveleti jel
        CIKLUS AMÍG a verem tetején levő elem prioritása >= X prioritása
                    KIVESZ VEREMBŐL Y-ba
                    EREDMENY := EREDMENY + Y
        CVÉGE
        VEREMBE X
    EVÉGE
CVÉGE
CIKLUS AMÍG verem nem üres
        KIVESZ VEREMBŐL Y-ba
        EREDMENY := EREDMENY + Y
CVÉGE
```

Példák:

```
((A-B)*(C+D))/(X+Y)
(A-B) *(C+D)/(X+Y)
 A*B/C^(X+Y)*D
 A*B/C^(X+Y)+D

 A B + C - 5 / *
 A B C * + E / F -
 A B + X Y + *
 A B X Y + ^ C * +
```

A lengyel-formára hozott kifejezések kiértékelése

```
VEREM ÜRESRE
CIKLUS AMÍG NEM ÜRES POSTFIX_SOR
    Elem := POSTFIX_SOR.ELSŐ_ELEME
    ELÁGAZÁS Elem típus szerint
            HA Elem=ADAT AKKOR VEREMBE(Elem)
            HA Elem=MŰVELET AKKOR
                    Elem1 := VEREM TETEJE
                    Elem2 := VEREM TETEJE
                    Eredmeny := Kiszamol(Elem1,Elem2,Művelet)
                    VEREMBE( Eredmeny )
    EVÉGE
CVÉGE
VÉGEREDMÉNY := VEREM TETEJE
```

Pointerek

Az eddig ismert változók az alábbi jellemzőkkel bírnak:

- van nevük
- van típusuk
- van aktuális értékük

A változó aktuális értéke a memória egy meghatározott területén helyezkedik el. A változó neve igazából e memóriaterület kezdőcímét határozza meg, a típusa pedig abban segít, hogy az ezen a területen lévő byte-sorozatot hogyan kell értelmezni, valamint azt is megadja, hogy ezen kezdőcímtől kezdve hány byte tartozik az adott változóhoz.

Más szempont szerint a változóknak van:

- hatókörük
- élettartam

A hatókör lehet teljes programra kiterjedt (globális), vagy a program szövegének csak bizonyos részén használható (lokális). Az eddigi ismeretek szerint a program fő deklarációs részén megadott változók globális hatókörrel rendelkeznek, az alprogramok formális paraméterlistájában megadott változók, ill. az alprogramon belül deklarált változók lokális hatókörrel rendelkeznek.

Az élettartam azt adja meg, hogy a változó a program futásának időtartama alatt mikor keletkezik, és mikor szűnik meg. Amennyiben a program indulásakor keletkezik (kap memóriacímet és területet), és a program teljes futási időtartama alatt végig ugyanezen memóriaterület felett rendelkezik, és csak a program befejeztével szabadul fel ezen terület – akkor a változó élettartama **statikus**. Ha a program futása alatt a változó többször is *létrejön*, és *megszűnik*, vagyis többször kap memóriaterületet (gyakran nem mindig ugyanazt a területet), és ezen terület aztán fel is szabadul, akkor a változó élettartama **dinamikus**. Az eddigi ismeretek szerint a program fő deklarációs részén megadott változók *statikus* élettartammal rendelkeznek, az alprogramok formális paraméterlistájában megadott változók, ill. az alprogramon belül deklarált változók *dinamikus* élettartammal rendelkeznek.

Ez utóbbiról tudni kell, hogy a fent említett esetekben a dinamikus változók létrehozása és megszüntetése automatikus folyamat, szabályozni nem lehet!

A most ismertetésre kerülő típus szinte minden szempontból kívül áll ezen az osztályozási rendszeren, ez a ***mutató*** típus (magyarul pointer ☺).

A mutató típusú változó egy dupla változónak fogható fel. Szinte minden jellemzőjéből kettő van, kivéve a nevét. Neve csak egy van.

```
DEKLARÁCIÓ
    S: szöveg
    P: mutató egy szöveg-re
    …
```

Az „S" nevű változó egy hagyományos változó. Tartozik hozzá egy memóriaterület, ahol az értéke (pl. „Helló") eltárolásra kerül.

A „P" nevű változó elsődleges típusa a *mutató* típus. Ennek megfelelően tartozik hozzá egy memóriaterület, ahol az aktuális értéke eltárolásra kerül.

Ezen a területen (mely legtöbb esetben négy byte) tárolt érték azonban egy újabb memóriaterület kezdőcíme[4]. Ezen másodlagos memóriaterületen tárolódik a P változó *másodlagos* típusához illő

[4] Vagy egy speciális *üres* érték. Erről később.

érték. Jelen példában a P másodlagos típusa a „szöveg”, hiszen a deklaráció szerint „P mutató egy szöveg-re”.

Miért jó ez nekünk? Tudni kell, hogy ezen másodlagos adatterület nem biztos, hogy a P teljes élettartama alatt életben van. Ez azt jelenti, hogy az elsődleges adatterület a P élettartama alatt él, és foglalja a maga kis területét a memóriából, a másodlagosról azonban a <u>*programozó*</u> dönthet.

Egy ilyen mutató típusú változóhoz kezdetben nem tartozik másodlagos memóriaterület. Ekkor azt mondjuk, hogy a „P” *nem mutat sehova*. Ekkor a P értéke (figyelem, elsődleges típusának elsődleges adatterületének értelmében) egy speciális érték, amelyet NIL[5]-nek nevezünk. Amikor a P értéke NIL, ez azt jelenti, hogy nincs mögötte másodlagos terület. *Ezt a „HA P=NIL AKKOR …” és a „HA P<>NIL AKKOR…” formában tudjuk ellenőrízni.*

Amikor a programozó úgy dönt, hogy most kell a másodlagos terület is, akkor kér egy megfelelő, éppen szabad memóriaterületet a P számára. Ezt a *Foglal(P)* utasítás segítségével lehet elérni[6]. Ekkor a P elsődleges területére egy érvényes memóriacím kerül, mely a másodlagos adatterületre mutat. Ezen adatterület ekkor még természetesen inicializálatlan!

Amikor a P-hez tartozik másodlagos adatterület, akkor arra a P^ (olvasd: *P mutat,* vagy *P kalap)* módon tudunk hivatkozni.

Amikor a P-hez tartozó másodlagos adatterületre már nincs szükség, akkor azt a *Felszabadit(P)* utasítás segítségével tudjuk felszabadítani. Ekkor e terület újra felhasználhatóvá válik más, mutató típusú változók *Foglal* kérésének kielégítésére. Másik hatása, hogy a P értéke újra NIL lesz.

Súlyos hibának számít, és a program leállásához vezet[7]*, ha a P^ segítségével a P másodlagos adatterületére hivatkozunk akkor, amikor nincs is neki!*

A pointer típusú változók tehát elsődleges adatterületükre vonatkozóan lehetnek statikusak (ha a P a program fő deklarációs részén volt feltüntetve), és lehet dinamikus is (ha a P egy paraméterváltozó, vagy lokális változó). A P másodlagos adatterülete azonban mindenképpen dinamikus, méghozzá a programozó által vezérelt módon dinamikus. Ugyanis a program szövegébe beszőtt *Foglal* és *Felszabadít* utasítások szabályozzák ezen terület létét – és nem létét.

Az „adott típusra mutató” típusú változók mindenütt használhatók, ahol az adott típusú változó használható. Pl:

```
S := „Hello”          <->    P^:= „Hello”
Ki: S                 <->    Ki: P^
          P^:=S, stb.
```

Miért jó ez? Mert egy pointer aránylag kis tárigényű (4 byte), míg a mögöttes tárolt érték nagy is lehet (pl egy nagy méretű vektor vagy mátrix). Elképzelhető, hogy ezen másodlagos terület nem kell a program futásának időtartama alatt, de az statikus élettartam túl sok, a dinamikus élettartam meg lokális hatókörhöz tartozik, és e miatt nem megfelelő. Ekkor használunk pointer típust, amely lehet globális hatókörű, de az *élettartama* lehet szűkebb, mint a teljes program időtartama. Pointerek segítségével megoldható, hogy egyszerre mindig annyi memóriát kössünk le, amennyi a program adott fázisában szükséges – így a különböző részeredmények (melyek más-más időben állnak rendelkezésre) a memória ugyanazon területein helyezkedjenek el.

---

[5] A latin NIHIL azt jelenti: „semmi”.
[6] Turbo Pascal-ban `New(P)` az utasítás formája.
[7] A gyakorlatban inkább a teljes gép lefagyásához, vagy újraindulásához ☺

Fák, bináris fák, bejárási stratégiák

A **Fa** egy összetett adatszerkezet, amely tárolási kapacitása elméletben végtelen, gyakorlatban vagy a memóriakapacitás szab véget, vagy a Fa megvalósításának technikája (pl. vektoros szimuláció esetén a vektor mérete).

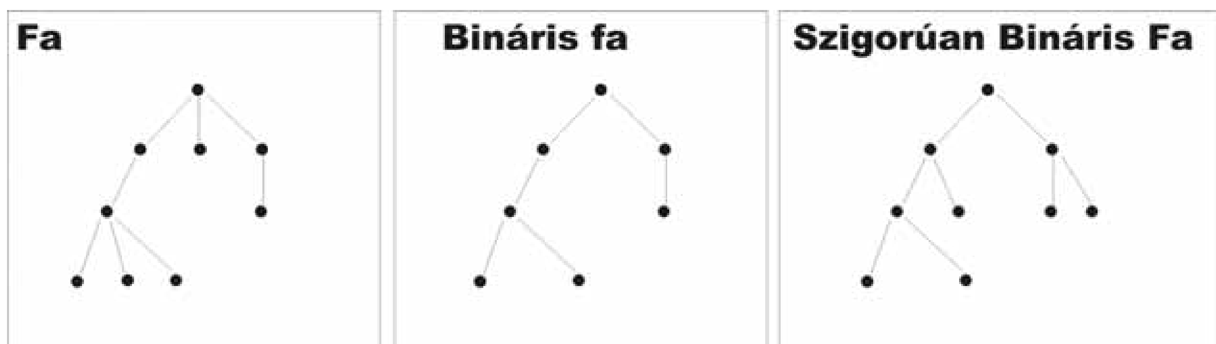
A Fa adatszerkezet esetén nem egyértelmű a homogenitás – magát a Fá-t alkotó elemekből is három fajta van: gyökér, csomópont, levélelem. Ugyanakkor ezen elnevezések nem utalnak arra, hogy maga a Fa homogén-e, vagy sem. A fenti elnevezések inkább arra utalnak, hogy az adott elem a Fán belül hol helyezkedik el – milyen a viszonya a többi Fa elemmel. A homogenitást az adja, hogy a Fa tényleges információhordozó része – az adat – egyforma típusokból áll-e. Nos, ez egyszerű esetben elmondható, vagyis a Fa homogén. Ugyanakkor a gyakorlati alkalmazás során a csomópontok általában más jellegű, más típusú információt hordoznak, mint a levélelemek. Ezt természetesen az adott programozási nyelv által megengedett „trükkök” segítségével lehet elérni. Tehát a gyakorlati életben a Fa inkább inhomogén. Ugyanakkor látni fogjuk, hogy a Fá-val kapcsolatos algoritmusok magja megírható általánosan.

A Fa valójában a gráfok speciális esete – olyan gráf, amelynek van egy kitüntetett pontja (kiinduló pont – Fa esetén ez a gyökérelem), ezen pontból kiindul *néhány* él, amelynek a végén csomópontok találhatóak, amelyekből *kiindulhatnak* újabb *élek (0,1,2,3,… akárhány él)*. A Fa attól speciális, hogy a gyökérelemet leszámítva minden elemnek (csomópontnak) csak egy *szülő* eleme lehet, a pontba csak egy él vezethet *be*, de akárhány él vezethet ki.

A Fa mint adatszerkezettel kapcsolatos fogalmak:

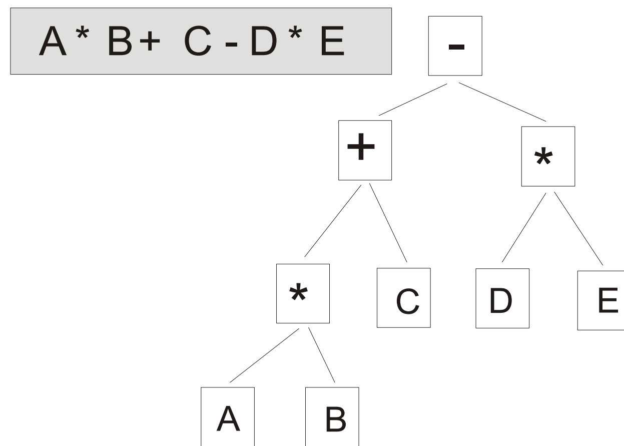
- van kezdőpont	az új elem csak egy elemhez kapcsolódik
- gyökérelem, root	kezdőelem
- csomópont	van leágazása
- levélelem	nincs belőle leágazó él
- élek, irányultság	csomópontból ->
- Út, út hossza	egy elem megkeresésének útja
- szülő, gyerek	egy szülőtől közvetlenül leszármazott elemek
- testvérek	egy csomópontból ágaznak le

- minden elem pontosan egy úton érhető el
- bináris fa: maximum két elem minden csomóponton
- szigorúan bináris fa: pontosan két elem minden csomóponton



Bejárási stratégiák:

KBJ preorder:	root	bal	jobb
BKJ inorder	bal	root	jobb
BJK postorder	bal	jobb	root




```
Preorder:              - + * A B C * D E
Inorder:               A * B + C – D * E
Postorder              A B * C + D E * -
```

TÍPUS Fa_Elem = rekord
Bal : mutató egy Fa_Elem-re
Adat : <tetszőleges adattípus>
Jobb: mutató egy Fa_Elem-re
TVÉGE

Eljárás Preorder (P:mutató egy Fa_Elemre)
HA P <> NIL
P^.ADAT feldolgozása
Preorder(P^.Bal)
Preorder(P^.Jobb)
HVÉGE

Eljárás Inorder (P:mutató egy Fa_Elemre)
HA P <> NIL
Inorder(P^.Bal)
P^.ADAT feldolgozása
Inorder(P^.Jobb)
HVÉGE

Eljárás Postorder (P:mutató egy Fa_Elemre)
HA P <> NIL
Postorder(P^.Bal)
Postorder(P^.Jobb)
P^.ADAT feldolgozása
HVÉGE

Tömbös szimuláció:

A tömbös szimuláció során egy N elemű vektorban eltároljuk a csomópontok által hordozott értéket, plusz leírjuk, hogy a bal oldali leágazás és a jobb oldali leágazás hányadik vektorelemnél van. Amennyiben nincs leágazás, azt a (0) értékkel jelöljük. A vektor 0. eleme speciális tulajdonságú – mindkét leágazása (-1)-el van jelölve. Ezen elem hasonló funkciót tölt be, mint a strázsaelem –az algoritmus felírását könnyíti meg. A vektor (1) sorszámú eleme kell legyen a gyökérelem. E pozíciótól

kezdve a fa csomópontjai tetszőleges sorrendben tárolhatók el a vektorban – csak a leágazások vektorpozíciói legyenek helyesen kitöltve!

Sorszám	Elem értéke	Bal	Jobb
0		-1	-1
1	-	8	7
2	A	0	0
3	B	0	0
4	C	0	0
5	D	0	0
6	E	0	0
7	*	5	6
8	+	9	4
9	*	2	3

Ilyen jellegű szimuláció esetén is elvégezhető a fa bejárása. A hozzájuk tartozó algoritmusok a jegyzet végén található mintaprogramokból kikereshető.

Állományok

Az állományok olyan adatszerkezetek, melyek tárolása az eddigiektől eltérően nem a számítógép operatív tárában, hanem valamely háttértárolón (leggyakrabban valamely megfelelő kapacitású winchesteren) történik meg. Ezt a tárolást jellemzi az, hogy ezen adatok mennyisége igen nagy is lehet (a maximális méretet az operációs rendszer képességei, ill. a háttértár kapacitása jelenti). Másik fontos jellemzője, hogy az adatok elérése nagyságrendekkel lassúbb, mint memória esetén. Amennyiben a file valamely hálózati kiszolgálón található – ezen sebesség újabb nagyságrendekkel csökken.

A megfelelő hatékonyságú algoritmus írásához (a megfelelő futási sebesség eléréshez) alaposan ismerni kell az adott háttértár adatelérési, adattárolási módszereit (sávok, szektorok, pufferek, stb), és az adott file-típus lehetőségeit.

A file-ok szerkezetüket tekintve az alábbi csoportokba sorolhatóak:

- Fix szerkezetű
- Nem fix szerkezetű
  - Bináris
  - Text

A fix szerkezetű file-ok leginkább a vektorokra hasonlítanak. A fix szerkezet jelen esetben a homogén felépítő típust jelenti. A file minden egyes eleme, komponense ugyanazon típusú. Ez a típus lehet egyszerű (egész, valós, stb…), vagy összetett (leggyakrabban rekord). A lényeg az, hogy minden elem ilyen típusú legyen. Ez az ok biztosítja a direkt elérést.

A nem fix szerkezetű file-ok ennek ellenkezőjét jelentik: a file-ban tárolt információk nem egy homogén adattípushoz tartoznak, lényegében felváltva hol ilyen-hol olyan típusú adatok vannak benne. Ez a bináris, nem fix szerkezetű file-ok jellemzői.

A Text file-okat is ebbe a csoportba soroljuk, bár ez egy érdekes kérdés. Ugyanis a text fileok alapvetően homogén adattípusból épül fel: sorokból. Azonban ezek a sorok *változó hosszúak*. Másrészről a sorokat további részekre lehet bontani, amennyiben a feldolgozás ezt megköveteli: szavakra, akár karakterekre is. E miatt a homogenitás valójában nem egyértelműen igaz. Másrészről a változó hosszú sorok miatt nincs lehetőség a direkt elérést, ezen indokok miatt a text fileokat a nem fix szerkezetű csoportba tesszük.

A fileokon belüli adatok elérése három csoportba sorolható:

Szekvenciális elérés: ez azt jelenti, hogy a adatok elérése csak valamilyen sorrendben történhet meg. Elindulván egy adott pontból (ez leggyakrabban a file eleje), végig kell olvasni a kívánt adat előtti minden adatot, hogy elérhessünk a file arrébb lévő kívánt helyre. Ezen hely pontos pozíciója nem határozható meg egy képlet segítségével, ezen pozíció az előtte tárolt adatok milyenségétől függ.

Pl. text fileok esetén ez nyilvánvaló: ha a 10. sort szeretném beolvasni, annak file-on belüli helyét az előtte lévő (1..9) sorok hossza határozza meg.

Ezen elérés jellemzi többek között a láncolt listákat is: az n. adat eléréséhez a lista fejétől elindulva tudok csak eljutni, átlépkedve az n. előtti adatokon.

Direkt elérés: ezen elérés során, amennyiben tudom, hogy a kívánt adat hányadik a file-n belül – egy képlettel kiszámítható a pontos pozíció, és közvetlenül oda tudok állni a file-mutatóval, majd az előtte lévő adatok átlépésével közvetlenül elérhetem, módosíthatom az adott komponenst.

Ilyen jellemzővel bír a fix szerkezetű file-k. Hiszen vektoroknál tanult címfüggvényhez hasonlóan - amennyiben ismerjük az egy komponens tárolásához szükséges byte-ok számát -, kiszámítható, hogy a kívánt n. adat a file hanyadik byte-ján kezdődik.

Ezen elérés a vektorok eléréséhez hasonlítható. A vektor bármelyik elemét közvetlenül lekérdezhetjük az előtte lévő elemek elérése nélkül is.

Minden direkt eléréssel is rendelkező file-t lehet szekvenciális módban is kezelni! Ez az állítás nyílvánvalóan igaz, hiszen a direkt elérés azt jelenti, hogy a file-ban *tetszőleges* pozícióra közvetlenül rá tudok *ugrani*, pozícionálni, vagyis a pozíció szerint növekvő sorrendben is be tudom járni a file-t.

Indexelt elérés: Ezen elérés a direkt elérésű fileok esetén kínál plusz sebességnövekedést. A lényege, hogy a fileban tárolt komponensek (rekordok) valamely részei (mezői) szerint egy külön kigyüjtés készül, melyben szerepel a komponens adott része, és a komponens file-n belüli pozíciója. Ezen *kigyűjtés*-t rendezzük a rész szerint növekvő sorrendben. Amennyiben valamely komponenst keressük a file-ban, akkor ezen kigyűjtésben a bináris (logaritmikus) keresés segítségével nagy sebességgel megtalálhatjuk a kívánt rekordot, majd annak pozíciójának ismeretében, a direkt elérés segítségével közvetlenül ráálván a keresett rekordra, kiolvashatjuk azt.

Tipikus felhasználási példa az adatbázis-kezelő rendszerek. Tegyük fel, hogy van egy fix szerkezetű file, amelyben a hallgatók személyi adatait tároljuk (név, lakcím, születési dátum, stb…). Amennyiben a feladat az, hogy keressük ki a „Kovács Géza” nevű ember lakcímét, akkor igazából nem segít a direkt elérés, hiszen nem tudjuk, hogy ezen névhez tartozó rekord hányadik! A megoldás az lehet, hogy elindulunk a file elejéről, minden rekordot megvizsgálunk, hogy ő-e a keresett nevű ember (szekvenciális keresés). Ez lassú.
A rekord egy része, a nevek szerint készítsünk egy segédtáblát, melyben benne vannak a nevek, és hogy melyik névhez melyik pozíciójú rekord tartozik. Majd ezen segédtáblát rendezzük a rész, a név szerint. Ekkor ezen segédtáblában a bináris keresés segítségével gyorsan meg lehet találni a „Kovács Géza”-t, és kiolvasva a rekord pozícióját, a direkt elérés segítségével rálépve az eredeti file-ban ezen rekordra, gyorsan megtalálható a valójában keresett lakcím.

Az indexek készítéséről, használatáról még lesz szó a programozási nyelvek, és az adatbázis-kezelés c. tantárgyakban.

Láncolt listák

```
program LancoltLista; {Rendezetlen es Rendezett Lancolt Lista kezelese}
uses crt;
type PRekord = ^TRekord;  { mutató egy TRekord-ra }
     TRekord = record
        Nev  : string[30];
        Kov  : PRekord;
     end;

var Start : PRekord;
```

```
{.........................................................................}
function  UjElem(Nev:string):PRekord;
var P:PRekord;
begin
  Getmem(P,Sizeof(TRekord));
  P^.Nev:=Nev;
  P^.Kov:=NIL;
  UjElem:=P;
end;
{.........................................................................}
procedure ListaInit;
begin
  Start:=UjElem('');
end;
{.........................................................................}
function Kovetkezo(P:PRekord):PRekord;
begin
  if P=NIL then Kovetkezo:=NIL
           else Kovetkezo:=P^.Kov;
end;
```

```
{.............................................................}
function Elozo(P:PRekord):PRekord;
begin
  if P=NIL then Elozo:=NIL
           else Elozo:=P^.Elo;
end;
{............................................................}
procedure ListaKiir; {a teljes lista feldolgozasa}
var P:PRekord;
begin
   P:=Start;
   while Kovetkezo(P)<>NIL do begin
       P:=Kovetkezo(P);
       writeln(P^.Nev);
   end;
end;
{.............................................................}
procedure Vegehez_Ad(UjNev:string);
var P:PRekord;
begin
    P:=Start;
    while Kovetkezo(P)<>NIL do P:=Kovetkezo(P);
    P^.Kov  := UjElem(UjNev);
end;
{.............................................................}
procedure AdottHelyreBeszur(UjNev:string;I:integer); {I. elem moge}
var P,PUj:PRekord;
begin
    P:=Start;
    while (Kovetkezo(P)<>NIL) and (I>0) do begin
          P:=Kovetkezo(P);
          I:=I-1;
    end;
    PUj      := UjElem(UjNev);
    PUj^.Kov := P^.Kov;
    P^.Kov   := PUj;
end;
{.............................................................}
procedure Torol(I:integer);
var P:PRekord;
begin
    P:=Start;
    I:=I-1;
    while (Kovetkezo(P)<>NIL) and (I>0) do begin
          P:=Kovetkezo(P);
          I:=I-1;
    end;
    P^.Kov:=Kovetkezo(Kovetkezo(P));
end;
{.............................................................}
function  HelyetKeresi(Nev:string):integer;
var P:PRekord;I:integer;
begin
    P:=Start;I:=0;
    while (Kovetkezo(P)<>NIL) and (Kovetkezo(P)^.Nev<Nev) do begin
          P:=Kovetkezo(P);
          I:=I+1;
    end;
    HelyetKeresi:=I;
end;
```

```
{...............................................................}
procedure HelyereBeszur(UjNev:string);
var I:integer;
begin
    I:=HelyetKeresi(UjNev);
    AdottHelyreBeszur(UjNev,I);
end;
{..............................................................}
procedure HelyerebeszurGyors(UjNev:string);
var P,PUj:PRekord;
begin
    P:=Start;
    while (Kovetkezo(P)<>NIL) and (Kovetkezo(P)^.Nev<UjNev) do begin
          P:=Kovetkezo(P);
    end;
    PUj       := UjElem(UjNev);
    PUj^.Kov := P^.Kov;
    P^.Kov    := PUj;
end;
{...............................................................}
procedure TeljesListaTorol;
var P,PKov:PRekord;
begin
  P:=Start;
  while Kovetkezo(P)<>NIL do begin
        PKov   :=Kovetkezo(P);
        FreeMem(P^.Kov,Sizeof(TRekord));
        P^.Kov :=NIL;
        P      :=PKov;
  end;
end;
{...............................................................}
```

BEGIN

```
 ListaInit;
 ClrScr;
 Vegehez_Ad('Adam');
 Vegehez_Ad('Zsolt');
 AdottHelyreBeszur('Balazs',1);   {0:elso elem lesz,1:masodik elem lesz, 2:harmadik}
 HelyereBeszur('Csongor');
 HelyereBeszurGyors('Tibor');
 Torol(2);                  {0,1:1.elem  2:2. 3:3. 4..:semmit}
 ListaKiir;
 TeljesListaTorol;
END.
```

Hatékonyság-vizsgálat.

Algoritmus hatékonysága
Programkód hatékonysága

Végrehajtási idő
Helyfoglalás
Bonyolultság

Fontos jellemzők:

Minimális/maximális/várható (átlagos) végrehajtási idő
Maximális, minimális, átlagos tárkapacitás
Programkód hossza

Algoritmus hatékonysága:

1. válasszunk jó algoritmust
2. csökkentsük a ciklusok végrehatásának számát
   ciklusok egyszeri végrehajtási idejét
3. feltételvizsgálatok számát csökkentsük
4. kivételes eseteket küszöböljük ki

Programkód hatékonyság

1. használjunk gyorsabb számolást és kisebb helyfoglalást lehetővé tevő adattípusokat
2. hozzuk a feltételeket egyszerűbb alakra
3. kerüljük a különböző adattípusokkal való műveleteket
4. fv-ek kiszámításának csökkentése

Programtranszformációk:

1. Ha U1 nem változtatja meg F-et

```
HA F    AKKOR      U1:U2
        KÜLÖNBEN   U1:U3
```

```
U1
HA F    AKKOR U2
        KÜLÖNBEN U3
```

2. Nincs megkötés

```
HA F    AKKOR      U1:U3
        KÜLÖNBEN   U2:U3
```

```
HA F    AKKOR      U1
        KÜLÖNBEN   U2
U3
```

3. Ha a nyelv nem optimalizál

```
HA F1 ÉS F2 AKKOR U
```

```
HA F1 AKKOR HA F2 AKKOR U
```

4. Ha F1 és F2 közül az egyik biztosan teljesül

```
HA F1   AKKOR U1
        KÜLÖNBEN HA F2 AKKOR U2
```

```
HA F1   AKKOR U1
        KÜLÖNBEN U2
```

5. Ha U1 nem befolyásolja F-et

```
HA F AKKOR U1
HA F AKKOR U2
```

```
HA F AKKOR U1:U2
```

6. Ha U1 és U2 független egymástól és U3-tól és CF-re csak U3-nak van hatása

```
CIKLUS AMIG CF
        U1:U3
CVEGE
CIKLUS AMIG CF
        U2:U3
CVEGE
```

```
CIKLUS AMIG CF
        U1:U2
        U3
CVEGE
```

7. Nincs megkötés

```
U
CIKLUS AMIG CF
        U
CVEGE
```

```
CIKLUS
        U
AMIG CF
```

8. Ha F értéke nem változik a ciklus futása alatt

```
CIKLUS AMIG CF
        HA F AKKOR U
CVEGE
```

```
HA F AKKOR
        CIKLUS AMIG CF
                U
        CVEGE
HVEGE
```

9. Ha U1 végrehajtása független a ciklustól és saját maga korábbi végrehajtásától

```
        CIKLUS AMIG CF
                U1
                U2
        CVEGE
```

```
U1
CIKLUS AMIG CF
        U2
CVEGE
```

Program-helyesség bizonyítása.

- Matematikai módszer: minden programsorra elő- és utófeltételt kell szabni, és be kell bizonyítani, hogy az előfeltétel teljesülése esetén, valamint a programsor működését ismervén az utófeltétel teljesül. A következő programsorának előfeltétele természetesen az előző sor utófeltételének figyelembevételével kell megkonstruálni. Ha következetesen végigvisszük, akkor a program bemenő feltételeit ismervén megkapjuk a program utófeltételét. (problémás a ciklusok és a feltételes elágazások elő- és utófeltételét kidolgozni, mivel ezek nem önálló programsorok, hanem programblokkok).
- Empirikus módszer: a programot kipróbáljuk különböző bemenő értékekkel. Ha a program valóban azt csinálja, akkor valószínűsíthető, hogy a program jó.
    - Tesztelés: a program azt teszi-e, amit várunk tőle
    - Hibakeresés (ha a tesztelés során probléma merült fel)
    - Hibajavítás
    - Újra tesztelés (ellenőrizni, hogy nem keletkezett-e újabb hiba)

Statikus tesztelés

A program forráskódjának „olvasgatása”.

- kódellenőrzés: a program szövegének összevetése az algoritmussal
- formai ellenőrzés: ez szintaktikai ellenőrzés, általában a fordítóprogram végzi.
- tartalmi ellenőrzés: belső ellentmondások keresése
    - a változó még nem kapott kezdőértéket, de mi már használjuk
    - vezérlési problémák
        - végtelen ciklus lehetősége
        - végtelen rekurzió
    - elágazási ág sohasem hajtódik végre
    - stb.

Dinamikus tesztelés – fekete doboz módszer

Nem ismerjük a program kódját, de ismerjük a feladatát. Többszöri kipróbálás révén próbálunk következtetni a működési zavarokra. *Csak nagyon tapasztalt szakember tudja hatékonyan végezni.* Ekvivalencia osztályokat képezünk a bemenő adatokból, amelyekből ha mintát veszünk, akkor mindig jól/rosszul kell működnie a programnak.

Pl: a program feladata: maximum 40 karakteres sztringben számolja meg, hány magánhangzó fordul elő. Az ekvivalencia-osztályok:

- 0 hosszú szöveg
- több mint 40 karakter hosszú szöveg

- nincs benne magánhangzó
- van benne magánhangzó

Határeset-elemzés: az ekvivalencia osztályok határfelületét külön tesztelni
- kipróbálni 0 és 1 hosszú szavakra
- kipróbálni 39,40,41 hosszú szavakra
- van benne magánhangzó esetet megbontani
    - csak egyfajta magánhangzó, összesen 1 db
    - csak egyfajta magánhangzó, többször is előfordul
    - többfajta magánhangzó, de mindegyik csak egyszer fordul elő
    - többfajta magánhangzó, vegyes előfordulási számmal

Dinamikus tesztelés – fehér doboz módszer

Ismerjük a programkódot, és annak ismeretében tervezzük meg a tesztelést.
- utasítások egyszeri lefedése: olyan bemenő adatokat adjunk meg, amellyel elérjük, hogy minden utasítás (ciklusok, feltételek mélyén levők) legalább egyszer végrehajtódjon
- döntéslefedés: minden HA minden ága (AKKOR, KÜLÖNBEN) legalább egyszer végrehajtásra kerüljön (többszörös elágazások minden ága is)
- részletlefedés: összetett HA (és-el vagy-al összekapcsolt logikai kifejezések) minden részfeltétele kiértékelődjön

Hibakeresés

Feladata: meghatározni a program azon sorát, sorait, amely miatt a program nem megfelelően működik.

Hibakeresés - Indukciós módszer:

A tesztadatok ismeretében próbálunk rájönni a hiba okára. Ha van valami elképzelésünk, direkte ennek ellenőrzésére megfelelő tesztadatokkal újra próbáljuk a programot. Ha a gyanú nem igazolódott be, újabb összefüggéseket keresünk azon tesztadatok között, amelyekre hibásan működik a program. Bonyolult a helyzet, ha több teszteredményre is hibás a működés, de más-más okból. Ezen módszernél nincs szükség a programkód ismeretére.

Hibakeresés - Dedukciós módszer:

A tesztadatokból először egy listát készítünk, melyben minden lehetséges okot feltételezünk, majd ezeket egyesével elkezdjük kizárni újabb tesztelés révén.

Hibakeresés – Visszalépéses módszer:

A program részeredményeit figyelve próbáljuk meghatározni a folyamat azon részét, ahol már hibás a működés. E ponttól visszafele haladva keressük a hibát okozó programrészt.

Hibakeresés módszerei

- kiírás: a változók értékeit időnként kiírjuk a képernyőre, file-ba, vagy másik monitorra. Ha a változó értékeit nem tudjuk kiírni, akkor kis üzeneteket íratunk ki, mely jelzi számunkra futás közben, hogy melyik programrészen áll a program (feltételek belsejébe, ciklusok magjába).
- állapotellenőrzés: a program kódba plusz feltétel-ellenőrzéseket teszünk be, melyek belsejében valamilyen kiírás történik. Ennek során tudjuk ellenőrizni, hogy bizonyos összefüggések előfordultak-e.
- töréspontok: a program kódjában a futtató rendszer számára értelmezhető módon töréspontot helyezünk el. Erre a pontra érve a futtató rendszer megállítja a program futását, és a programozó vagy tesztelő veszi át az irányítást. Ezen a ponton a tesztelő ellenőrizheti a változók értékét, esetleg meg is változtathatja azokat, majd folytathatja a program futtatását. A töréspontokhoz

időnként feltételeket is meg lehet adni pluszban, s a futás megállása csak akkor következik be, ha a program épp az adott programsoron halad át, és a plusz feltétel is bekövetkezik.

- nyomkövetés: a futás leállása után a program sorokat egyesével, lépésenként lehet végrehajtani, minden lépés után ellenőrizni lehet a változók értékeit. Általában a nyomkövető rendszerek támogatják a ciklusok egy lépésben történő végrehajtatását, az eljárások és függvények egy lépésben történő végrehajtatását.
- nyomkövetés hibától visszafelé: a futtató rendszer feljegyzi a különböző programsorokhoz tartozó programstátuszt (változók akkori értékét), és a törésponton megállva nem csak előrefele haladva lehet a program sorait lépésenként végrehajtatni, hanem visszafelé is. Ez a módszer rendkívül erőforrás-igényes (sok memória, sok lemezterület). E miatt kevés fejlesztő eszköz támogatja. Egyszerűbb változata annyit közöl, hogy milyen sorok végrehajtása történt meg eddig a pontig (napló-file).

```
program KBJ_Bejaras; { Preorder }
uses crt;
type TRekord = record
      Elem : char;
      Bal,Jobb:integer;
end;
const a:array [0..9] of TRekord =
  (
 {0}  (Elem:' ';Bal:-1;Jobb:-1),
 {1}  (Elem:'-';Bal:8;Jobb:7),
 {2}  (Elem:'A';Bal:0;Jobb:0),
 {3}  (Elem:'B';Bal:0;Jobb:0),
 {4}  (Elem:'C';Bal:0;Jobb:0),
 {5}  (Elem:'D';Bal:0;Jobb:0),
 {6}  (Elem:'E';Bal:0;Jobb:0),
 {7}  (Elem:'*';Bal:5;Jobb:6),
 {8}  (Elem:'+';Bal:9;Jobb:4),
 {9}  (Elem:'*';Bal:2;Jobb:3) );

var Verem : array [1..100] of integer;
    VM    : integer;
    Mut   : integer;
    Irany : integer;

procedure Push(X:integer);
begin
 inc(vm);
 Verem[vm]:=x;
end;

function Pop:integer;
begin
  Pop:=Verem[vm];
  dec(vm);
end;

BEGIN
  vm:=0; clrscr;
  push(-1);
  mut:=1;
  while mut>=0 do begin
        while mut>0 do begin
              write(a[mut].elem);
              if a[mut].jobb<>0 then begin
                 push(a[mut].jobb);
              end;
              mut:=a[mut].bal;
        end;
        mut:=pop;
  end;
end.
```

```
program BKJ_Bejaras; { Inorder }
uses crt;
type TRekord = record
      Elem : char;
      Bal,Jobb:integer;
end;
const a:array [0..9] of TRekord =
(
 {0}  (Elem:' ';Bal:-1;Jobb:-1),
 {1}  (Elem:'-';Bal:8;Jobb:7),
 {2}  (Elem:'A';Bal:0;Jobb:0),
 {3}  (Elem:'B';Bal:0;Jobb:0),
 {4}  (Elem:'C';Bal:0;Jobb:0),
 {5}  (Elem:'D';Bal:0;Jobb:0),
 {6}  (Elem:'E';Bal:0;Jobb:0),
 {7}  (Elem:'*';Bal:5;Jobb:6),
 {8}  (Elem:'+';Bal:9;Jobb:4),
 {9}  (Elem:'*';Bal:2;Jobb:3) );

var Verem : array [1..100] of integer;
    VM    : integer;
    Mut   : integer;
    Irany : integer;

procedure Push(X:integer);
begin
 inc(vm);
 Verem[vm]:=x;
end;

function Pop:integer;
begin
  Pop:=Verem[vm];
  dec(vm);
end;

BEGIN
  vm:=0; clrscr;
  push(-1);
  mut:=1;
  while mut>=0 do begin
        while mut>0 do begin
              push(mut);
              mut:=a[mut].bal;
        end;
        mut:=pop;
        if mut>0 then begin
           write(a[mut].elem);
           mut:=a[mut].jobb;
        end;
  end;
end.
```

```
program BJK_Bejaras; { Postorder }
uses crt;
type TRekord = record
      Elem : char;
      Bal,Jobb:integer;
end;
const a:array [0..9] of TRekord =
  (
 {0}  (Elem:' ';Bal:-1;Jobb:-1),
 {1}  (Elem:'-';Bal:8;Jobb:7),
 {2}  (Elem:'A';Bal:0;Jobb:0),
 {3}  (Elem:'B';Bal:0;Jobb:0),
 {4}  (Elem:'C';Bal:0;Jobb:0),
 {5}  (Elem:'D';Bal:0;Jobb:0),
 {6}  (Elem:'E';Bal:0;Jobb:0),
 {7}  (Elem:'*';Bal:5;Jobb:6),
 {8}  (Elem:'+';Bal:9;Jobb:4),
 {9}  (Elem:'*';Bal:2;Jobb:3) );
var Verem : array [1..100] of integer;
    VM    : integer;
    Mut   : integer;
    Irany : integer;

procedure Push(X:integer);
begin
 inc(vm);
 Verem[vm]:=x;
end;

function Pop:integer;
begin
  Pop:=Verem[vm];
  dec(vm);
end;

BEGIN
  vm:=0; clrscr;
  Push(0);Push(0);
  Mut:=1;
  while Mut>0 do begin
        while mut>0 do begin
              push(mut);
              push(a[mut].Bal);
              mut:=a[mut].bal;
        end;
        repeat
              irany:=pop;
              mut:=pop;
              if irany=a[mut].jobb then begin
                 write(a[mut].elem,' ');
              end;
        until (irany<>a[mut].jobb);
        if irany=a[mut].bal then begin
           push(mut);
           push(a[mut].jobb);
           mut:=a[mut].jobb;
        end;
  end;
end.
```

```
Program Sakktabla_Bejarasa_Lolepesben;
uses crt;
const lepesek:array [1..8,1..2] of integer =
  ( (1,2),(2,1),(2,-1),(-1,2),(-2,1),(1,-2),(-1,-2),(-2,-1));
var lovak:array [1..64] of integer;
    scr: array [1..8,1..8] of integer;
    i,m,xx,x,yy,y:integer;

procedure Kepernyo;
begin
 writeln('-¦¦T¦¦T¦¦T¦¦T¦¦T¦¦T¦¦T¦¦¬');
 for i:=1 to 7 do begin
   writeln('-  -  -  -  -  -  -  -  -');
   writeln('+¦¦+¦¦+¦¦+¦¦+¦¦+¦¦+¦¦+¦¦+');
 end;
 writeln('-  -  -  -  -  -  -  -  -');
 writeln('L¦¦+¦¦+¦¦+¦¦+¦¦+¦¦+¦¦+¦¦-');
end;

procedure Kirak(x,y,m:integer; ok:boolean);
begin
 gotoxy((x-1)*3+2,(y-1)*2+2);
 if ok then begin write(m:2); scr[x,y]:=1;end
       else begin write('  '); scr[x,y]:=0;end;
 gotoxy(1,20);
 for i:=1 to m do write('-');clreol;
 gotoxy(60,20);write('-');
end;

BEGIN
 clrscr; Kepernyo;
 for i:=1 to 64 do lovak[i]:=1;
 for x:=1 to 8 do for y:=1 to 8 do scr[x,y]:=0;
 m:=1; { 1..64  hanyadik lepesnel tart }
 x:=1;y:=1; Kirak(x,y,m,true);

 repeat
   if lovak[m]>8 then begin { rossz eset}
      Kirak(x,y,m,false);
      lovak[m]:=1;
      dec(m);
      if m=0 then begin writeln('nincs megoldas !');halt;end;
      x:=x-lepesek[ lovak[m],1 ];
      y:=y-lepesek[ lovak[m],2 ];
      inc(lovak[m]);
   end
   else begin   { jo eset }
        xx:=x+lepesek[  lovak[m],1 ];
        yy:=y+lepesek[  lovak[m],2 ];
        if (xx<1) or (xx>8) or (yy<1) or (yy>8) or (scr[xx,yy]<>0)
               then inc(lovak[m])
               else begin
                x:=xx;
                y:=yy;
                inc(m);
                Kirak(x,y,m,true);
        end;
   end;
 until m>64;
END.
```

Egy megoldás:

1	14	19	8	25
6	9	2	13	18
15	20	7	24	3
10	5	22	17	12
21	16	11	4	23