

# Záróvizsga tételsor

## 7. Programozás

Ancsin Ádám

### Programozás

Egyszerű programozási feladat megoldásának lépései (specifikálás, tervezés, megvalósítás, tesztelés). Az adattípus fogalma (típusspecifikáció, műveletek, reprezentáció, invariáns, implementáció). A visszavezetés módszere. A felsoroló típus specifikációja. Felsorolóra megfogalmazott programozási tételek (összegzés, számlálás, maximum kiválasztás, feltételes maximumkeresés, lineáris keresés, kiválasztás). Nevezetes gyűjtemények (intervallum, tömb, sorozat, halmaz, szekvenciális inputfájl) felsorolói.

## 1 Egyszerű programozási feladat megoldásának lépései

### 1.1 Bevezetés

Egy programozási feladat megoldása a kódoláson túl jó néhány tevékenységet tartalmaz. Az első teendő a feladat pontos meghatározása, a specifikáció. Ez a feladat szöveges és formalizált, matematikai leírásán (a specifikáció ún. szűkebb értelmezésén) túl tartalmazza a megoldással szemben támasztott követelményeket, környezeti igényeket is (ami a specifikáció ún. tágabb értelmezése).

A specifikáció alapján meg lehet tervezni a programot, elkészülhet a megoldás algoritmus és az algoritmus által használt adatok leírása. Az algoritmus és az adatszerkezet finomítása egymással párhuzamosan halad, egészen addig a szintig, amelyet a programozó ismeretei alapján már könnyen, hibamentesen képes kódolni. Gyakran előfordul, hogy a tervezés során derül fény a specifikáció hiányosságaira, így itt visszalépésekre számíthatunk.

Az algoritmusírás után következhet a kódolás. Ha a feladat kitűzője nem rögzítette, akkor ez előtt választhatunk a megoldáshoz programozási nyelvet. A kódolás eredménye a programozási nyelven leírt program.

A program első változatban általában sohasem hibátlan, a helyességéről csak akkor beszélhetünk, ha meggyőződünk róla. A helyesség vizsgálatának egyik lehetséges módszere a tesztelés. Ennek során próbaadatokkal próbáljuk ki a programot, s az ezekre adott eredményből következtetünk a helyességre. (Ne legyenek illúzióink afelől, hogy teszteléssel eldönthető egy program helyessége. Hisz hogy valójában helyes-e a program – sajnos – nem következik abból, hogy nem találtunk hibát.)

Ha a tesztelés során hibajelenséggel találkozunk, akkor következhet a hibakeresés, a hibajelenséget okozó utasítás megtalálása, majd pedig a hibajavítás. A hiba kijavítása több fázisba is visszanyúlhat. Elképzelhető, hogy kódolási hibát kell javítanunk, de az is lehet, hogy a hibát már a tervezésnél követtük el. Javítás után újra tesztelni kell, hiszen – legyünk őszinték magunkhoz! – nem kizárt, hogy hibásan javítunk, illetőleg – enyhe optimizmussal állítjuk: – a javítás újabb hibákat fed fel, ...

E folyamat végeredménye a helyes program. Ezzel azonban még korántsem fejeződik be a programkészítés. Most következnek a minőségi követelmények. Egyrészt a hatékonyságot kell vizsgálnunk (végrehajtási idő, helyfoglalás), másrészt a kényelmes használhatóságot. Itt újra visszaléphetünk a kódolási, illetve a tervezési fázisba is. Ezzel elérkeztünk a jó programhoz.

### 1.2 Specifikáció

A programkészítés menetének első lépése a feladat meghatározása, precíz "újrafogalmazása". Milyen is legyen, mit várjunk el tőle? Nézzünk meg néhány – jónak tűnő – követelményt egyelőre címszavakban! (A továbbiakban a specifikáció szűkebb értelmezéséről lesz szó.) A specifikáció legyen:

- helyes, egyértelmű, pontos, teljes
- rövid, tömör, ami legegyszerűbben úgy érhető el, hogy ismert formalizmusokra építjük

- szemléletes, érthető (amit időnként nehezít a formalizáltság)

A specifikáció első közelítésben lehetne a feladatok szövege. Ez azonban több problémát vethet fel:

- mi alapján adjuk meg a megoldást
- mit is kell pontosan megadni?

Például az a feladat, hogy adjuk meg  $N$  ember közül a legmagasabbat. A legmagasabb ember megadása mit jelent? Adjuk meg a sorszámát, vagy a nevét, vagy a személyi számát, vagy a magasságát, esetleg ezek közül mindegyiket? Tanulásgként megállapíthatjuk, hogy a specifikációnak tartalmaznia kell a bemenő és a kimenő adatok leírását.

Bemenet:

$N$  : az emberek száma,  
 $A$  : a magasságukat tartalmazó sorozat.

Kimenet:

$MAX$  : a legmagasabb ember sorszám.

Tudjuk-e, hogy a bemenő, illetve a kimenő változók milyen értéket vehetnek fel? Például az emberek magasságát milyen mértékegységben kell megadni? Az eredményül kapott sorszám milyen érték lehet: 1-től sorszámozunk, vagy 0-tól? Megállapíthatjuk tehát, hogy a specifikációban a bemeneti és a kimeneti változók érték-halmazát is meg kell adnunk.

Bemenet:

$N$  : az emberek száma, természetes szám;  
 $A$  : a magasságukat tartalmazó sorozat, egész számok, amelyek a magasságot centiméterben fejezik ki (a sorozatot 1-től  $N$ -ig indexeljük).

Kimenet:

$MAX$  : a legmagasabb ember sorszám, 1 és  $N$  közötti természetes szám.

Most már a bemenő és a kimenő változók érték-halmazát pontosan meghatároztuk, csupán az a probléma, hogy a feladatban használt fogalmakat és az eredmények kiszámítási szabályát nem definiáltuk. A specifikációnak tehát tartalmaznia kell a feladatban használt fogalmak definícióját, valamint az eredmény kiszámítási szabályát. Itt lehetne megadni a bemenő adatokra vonatkozó összefüggéseket is. A bemenő, illetve a kimenő adatokra kirótt feltételeket nevezzük előfeltételnek, illetve utófeltételnek. Az előfeltétel nagyon sokszor egy azonosan igaz állítás, azaz a bemenő adatok érték-halmazát semmilyen "külön" feltétellel nem szorítjuk meg.

Bemenet:

$N$  : az emberek száma, természetes szám,  
 $A$  : a magasságukat tartalmazó sorozat, egész számok,  
 amelyek a magasságot centiméterben tartalmazzák (a sorozatot 1-től  $N$ -ig indexeljük).

Kimenet:

$MAX$  : a legmagasabb ember sorszám, 1 és  $N$  közötti természetes szám.

Elofeltétel:

$A[i]$ -k pozitívak.

Utófeltétel:

$MAX$  olyan 1 és  $N$  közötti szám, amelyre  $A[MAX]$  nagyobb vagy egyenlo,  
 mint a sorozat bármely eleme (az 1. és az  $N$ . között).

Újabb probléma merülhet fel bármelyik feladattal kapcsolatban: az eddigiek alapján a "várttól" lényegesen különböző – nyugodtan állíthatjuk: "banális" –, az elő- és utófeltételnek megfelelő megoldást is tudunk készíteni.

Itt persze arról a hallgatónak (tehát még meg nem fogalmazott, ki nem mondott) feltételezésről van szó, hogy a bemeneti változók értéke nem változik meg. Ez sajnos nem feltétlenül igaz. A probléma megoldására kétféle utat követhetünk (a későbbiekben mindkettőt alkalmazni fogjuk):

- az utófeltételbe automatikusan beleértjük, hogy "és a bemeneti változók értéke nem változik meg", s külön kiemeljük, ha mégsem így van;
- az elő- és az utófeltételt a program paramétereire fogalmazzuk meg, amelyeket formailag megkülönböztetünk a program változóitól, és emiatt nem a paraméterek fognak változni, hanem a programbeli változók (ebben az esetben természetesen az elő- és az utófeltételben meg kell fogalmazni a paraméterek és a megfelelő programbeli változók értékének azonosságát).

A második megoldásból az következik, hogy meg kell különböztetnünk egymástól a feladat és a program elő-, illetve utófeltételét! Ez hosszadalmasabbá – bár precízebbé – teszi a feladat megfogalmazását, emiatt ritkábban fogjuk alkalmazni.

Előfordulhat, hogy a feladat megfogalmazása alapján nem lehet egyértelműen meghatározni az eredményt, ugyanis az utófeltételnek megfelelő több megoldás is létezik. Ez a jelenség a feladat ún. nemdeterminisztikussága. Ehhez a nemdeterminisztikus feladathoz tehát determinisztikus programot kell írunk, aminek az utófeltétele már nem engedheti meg a nem egyértelműséget, a nemdeterminisztikusságot. E probléma miatt tehát mindenképpen meg kell különböztetnünk egymástól a feladat és a program elő-, illetve utófeltételét!

Bemenet:

N : az emberek száma, természetes szám,  
 A : a magasságukat tartalmazó sorozat, egész számok,  
 amelyek a magasságot centiméterben tartalmazzák (a sorozatot 1-től N-ig indexeljük).

Kimenet:

MAX : a legmagasabb ember sorszáma, 1 és N közötti természetes szám.

Elofeltétel:

A[i]-k pozitívak.

Utófeltétel:

MAX olyan 1 és N közötti szám, amelyre A[MAX] nagyobb vagy egyenlo,  
 mint a sorozat bármely eleme (az 1. és az N. között).

Program utófeltétel:

MAX olyan 1 és N közötti szám, amelyre A[MAX] nagyobb vagy egyenlo,  
 mint a sorozat bármely eleme (az 1. és az N. között) és elotte  
 nincs vele egyenlo.

Megállapíthatjuk ebből, hogy a program utófeltétele lehet szigorúbb, mint a feladaté, emellett az előfeltétele pedig lehet gyengébb.

Visszatekintve a specifikáció eddig "bejárt pályájára" egy szemléletes modellje körvonalazódik a feladatmegoldásnak. Nevezetesen: nyugodtan mondhatjuk azt, hogy a feladatot megoldó program egy olyan automatát határoz meg, amelynek pillanatnyi állapota a feladat paraméterei (a program változói) által "kifeszített" halmaz egy eleme. (E halmaz annyi dimenziós, ahány paraméterváltozója van a programnak; minden dimenzió egyik változó érték-halmaza. Tehát egy konkrét időpillanatban e "gép" állapota: a változóinak abban a pillanatban érvényes értékeinek együttese.) Ezt a halmazt nevezzük a program állapotterének. Amikor megfogalmazzuk az előfeltételt, akkor tulajdonképpen kihasználjuk ebből az állapotteréből azt a részt (azt az altér), amelyből indítva elvárhatjuk az automatánktól (amit a megoldó program vezérel), hogy a helyes eredményt előállítja egy végállapotában. A végállapotot jelöltük ki az utófeltétellel.

Ezt a modellt elfogadva adódik még egy további megoldásra váró kérdés. Akkor ugyanis, amikor a programot írjuk, lépten-nyomon a részeredmények tárolására újabb és újabb változókat vezetünk be. Fölvetődik a kérdés: hogyan egyeztethető össze az imént elképzelt modellel? A válasz egyszerű: minden egyes újabb változó egy újabb dimenziót illeszt az eddig létrejött állapotterhez. Tehát a programozás folyamata – leegyszerűsítve a dolgot – nem áll másból, mint annak pontosításából, hogy hogyan is nézzen ki a megoldó automata állapottere (és persze: hogyan kell az egyik állapotból a másik állapotba jutnia). A feladatban szereplő paraméterek meghatározta "embrionális" állapotteret hívhatjuk paraméterternek, ami csak altere a program valódi állapotterének. Ez is azt sugallja, hogy a feladat előfeltétele gyengébb (azaz az általa kijelölt állapothalmaz) lehet, mint a program előfeltétele.

Foglaljuk most össze, hogy melyek a specifikáció részei! Ezek az eddigiek, valamint a programra vonatkozó további megkötések lesznek.

## 1. A feladat specifikálása

- a feladat szövege,
- a bemenő és a kimenő adatok elnevezése, értékhalmozának leírása,
- a feladat szövegében használt fogalmak definíciói (a fogalmak fölhasználásával),
- a bemenő adatokra felírt előfeltétel (a fogalmak fölhasználásával),
- a kimenő adatokra felírt utófeltétel.

## 2. A program specifikálása

- a bemenő és a kimenő adatok elnevezése, értékhalmozának leírása,
- (a feladat elő-, illetve utófeltételétől esetleg különböző) program elő- és utófeltétel,
- a feladat megfogalmazásában használt fogalmak definíciói.

Ezek az absztrakt specifikáció elemei. Az alábbiak másodlagos, mondhatjuk: technikai specifikáció részei:

- a program környezetének leírása (számítógép, memória- és perifériaigény, programozási nyelv, szükséges fájlok stb.),
- a programmal szembeni egyéb követelmények (minőség, hatékonyság, hordozhatóság stb.).

A technikai specifikáció nélküli leírást a program szűkebb specifikációjának nevezzük.

Progos specifikáció:

$$A = (N : \mathbb{N}, A : \mathbb{N}^{1..N})$$

$$Ef = (\forall i \in 1..N : A_i > 0)$$

$$Uf = (Ef \wedge \forall i \in 1..N : A_{MAX} \geq A_i \wedge \forall j \in 1..MAX - 1 : A_j < A_{MAX})$$

## 1.3 Tervezés

A tervezés során algoritmusleíró eszközöket használunk, amelynek célja a feladatok megoldásának leírása programozási nyelvtől független nyelven. A programozási nyelvek ugyanis szigorú szintaxisúak, a tervezés szempontjából lényegtelen sallangokat tartalmaznak. A programozási nyelven történő tervezés esetén nehezzé válhat a program átírása más nyelvre, más gépre.

Többféle algoritmusleíró eszköz is létezik, mi tanulmányaink során a struktogramot alkalmaztuk.

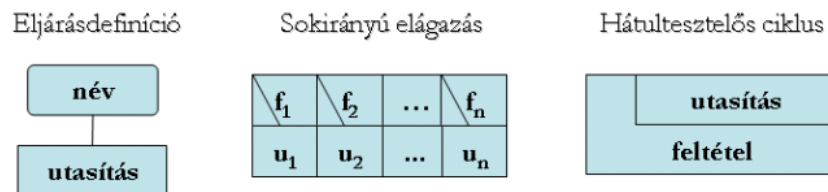
A struktogram a programgráfot élek nélkül ábrázolja. Így egyetlen egy alapelem marad, a téglalap. Ezzel az alapelemmel építhetjük fel a szokásos strukturált alapszerkezeteket (és csak azokat).



ábra 1: A struktogram összetett alapszerkezetei.

Szekvenciánál a téglalapok egymás alatti sorrendje dönti el a végrehajtás sorrendjét. Az elágazásfeltétel igaz értéke esetén az *i* betűvel jelölt bal oldali téglalap utasítását kell végrehajtani, hamis értéke esetén pedig az *n* betűvel jelölt jobb oldali téglalapét. Ha az elágazás valamelyik ága üres, akkor a neki megfelelő téglalap is üres marad. A ciklus előtesztelő, azaz a benne levő utasítást mindaddig végre kell hajtani, amíg a feltétel igaz.

Az utasítások helyén lehet egyetlen elemi utasítás, lehet a három algoritmikus szerkezet valamelyike, és lehet egy eljáráshívás. Ezt a leíróeszközt még többféle elemmel szokták bővíteni: az eljárásdefinícióval, a sokirányú elágazással, illetve a hátultesztelő ciklussal.



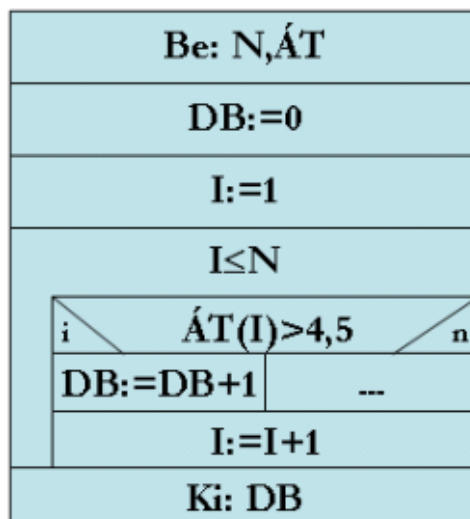
ábra 2: A struktogram további összetett alapszerkezetei.

Sokirányú elágazásnál azt az ágat kell végrehajtani, amelynek igaz értékű a feltétele (közülük minden esetben pontosan egy teljesülhet).

A lokális adatokat az eljárások téglalapjai mellett, az eljárásnév után sorolhatjuk fel.

Nézzük meg ezzel az eszközzel leírva a következő példát!

Feladat: N tanuló év végi átlagának ismeretében adjuk meg a jeles átlagú tanulók számát!



ábra 3: A példafeladat megoldása struktogrammal.

## 1.4 Megvalósítás

A.k.a. kódolás.

## 1.5 Tesztelés

A tesztelés célja, hogy minél több hibát megtaláljunk a programban. Ahhoz, hogy az összes hibát fölfedezzük, kézenfekvőnek tűnik a programot kipróbálni az összes lehetséges bemenő adattal. Ez azonban sajnos nem lehetséges.

Példaként tekintünk a következő - pszeudokóddal megadott - egyszerű programot:

```
Program:
  Változó A,B:Egész
  Be: A,B
  Ki: A/B
Program vége.
```

Mivel  $2^{16}$  különböző értékű egész számot tudunk tárolni, ezért az összes lehetőség  $2^{32}$ , aminek a leírásához már 9 számjegyre van szükség. Ez rengeteg időt venne igénybe, így nem is járható út.

Ha ezt a programot olyan bemenő adatokkal próbáljuk ki, amelyben  $A=0$  vagy  $B=1$ , akkor a program helyesen működik, a hibát nem tudjuk felfedezni. Ezután azt gondolhatnánk, hogy reménytelen

helyzetbe kerültünk: hiszen minden lehetséges adattal nem tudjuk kipróbálni a programot; ha pedig kevesebbel próbáljuk ki, akkor lehet, hogy nem vesszük észre a hibákat. A helyzet azért nem ennyire rossz: célunk csak az lehet, hogy a tesztelést olyan módszerrel hajtsuk végre, amellyel a próbák száma erősen lecsökkenthető.

Tesztesetnek a be- és kimeneti adatok és feltételek együttes megadását nevezzük. Akkor tudunk a tesztelés eredményeiről bármit is mondani, ha van elképzelésünk arról, hogy adott bemenő adatra milyen eredményt várunk.

Fogalmazzuk meg a tesztelés alapelveit:

- A jó teszteset az, ami nagy valószínűséggel egy még felfedetlen hibát mutat ki a programban. Például két szám legnagyobb közös osztóját számoló programot az [5,5] adatpár után a [6,6]-tal teljesen felesleges kipróbálni (ugyanis igencsak rafinált, valószínűtlen elírás esetén viselkedhet a program [6,6]-ra másként, mint [5,5]-re).
- A teszteset nemcsak bemenő adatokból, hanem a hozzájuk tartozó eredményekből is áll. Egyébként nem tudnánk a kapott eredmény helyes vagy hibás voltáról beszélni. A későbbi felhasználás miatt célszerű a teszteseteket is leírni a fejlesztői dokumentációban vagy egy önálló tesztelési jegyzőkönyvben.
- A meg nem ismételhető tesztesetek kerülendők, feleslegesen megnövelik a program-tesztelés költségeit, idejét. Nem is beszélve arról a bosszúságról, amikor a programunk egy hibás futását nem tudjuk megismételni, és így a hiba is felfedetlen marad.
- Teszteseteket mind az érvénytelen, mind az érvényes adatokra kell készíteni.
- Minden tesztesetből a lehető legtöbb információt "ki kell bányászni", azaz minden teszteset eredményét alaposan végig kell vizsgálni. Ezzel jelentősen csökkenthető a szükséges próbák száma.
- Egy próba eredményeinek vizsgálata során egyaránt fontos megállapítani, hogy miért nem valósít meg a program valamilyen funkciót, amit elvárunk tőle, illetve hogy miért végez olyan tevékenységeket is, amelyeket nem feltételeztünk róla.
- A program tesztelését csak a program írójától különböző személy képes hatékonyan elvégezni. Ennek oka, hogy a tesztelés nem "jóindulatú" tevékenység, saját munkájának vizsgálatához mindenki úgy áll hozzá, hogy önkéntelenül jónak feltételezi.

A programtesztelés módszereit két csoportba oszthatjuk aszerint, hogy a tesztelés során végrehajtjuk-e a programot, vagy nem. Ha csak a program kódját vizsgáljuk, akkor statikus (erről nem esik több szó), ha a programot végre is hajtjuk a tesztelés során, akkor dinamikus tesztelésről beszélünk.

### Dinamikus tesztelési módszerek

A dinamikus tesztelési módszerek alapelve az, hogy a programot működés közben vizsgáljuk. Teszteseteket kétféle módon tudunk választani. Egy lehetőség az ún. feketedoboz-módszer, más néven adatvezérelt tesztelés. E módszer alkalmazásakor a tesztelő nem veszi figyelembe a program belső szerkezetét, pontosabban nem azt tekinti elsődleges szempontnak, hanem a teszteseteket a feladat meghatározás alapján választja meg.

A cél természetesen a lehető leghatékonyabb tesztelés elvégzése, azaz az összes hiba megtalálása a programban. Ez ugyan elvileg lehetséges, kimerítő bemenet tesztelést kell végrehajtani, a programot ki kell próbálni az összes lehetséges bemenő adatra. Ezzel a módszerrel azonban, mint korábban láttuk, mennyiségi akadályba ütközhetünk.

Egy másik lehetőség a fehérdoboz-módszer (logika vezérelt tesztelés). Ebben a módszerben a tesztesetek megválasztásánál lehetőség van a program belső szerkezetének figyelembevételére is.

A cél a program minél alaposabb tesztelése, erre jó módszer a kimerítő út tesztelés. Ez azt jelenti, hogy a programban az összes lehetséges utat végigjárjuk, azaz annyi tesztesetet hozunk létre, hogy ezt elérhessük vele. Az a probléma, hogy még viszonylag kis programok esetén is igen nagy lehet a tesztelési utak száma. Gondoljunk a ciklusokra! Sőt ezzel a módszerrel a hiányzó utakat nem lehet felderíteni.

Mivel sem a fehérdoboz-módszerrel, sem a feketedoboz-módszerrel nem lehetséges a kimerítő tesztelés, el kell fogadnunk, hogy nem tudjuk egyetlen program hibamentességét sem szavatolni. A további cél

ezek után az összes lehetséges tesztet halmazából a lehető leghatékonyabb tesztet-csoport kiválasztása lehet.

A tesztelés hatékonyságát kétféle jellemző határozza meg: a tesztelés költsége és a felfedett hibák aránya. A leghatékonyabb tesztet-csoport tehát minimális költséggel maximális számú hibát fed fel.

A feketedoboz- és fehérdoboz-teszteken kívül még érdemes megemlíteni olyan speciális teszteket, amikor nem a helyesség belátása a cél. Ilyen pl. a stresszteszt (nagy adatmennyiséget hogyan bír kezelni a program, jól skálázódik-e) vagy a hatékonysági teszt (végrehajtási idő tesztelése).

## 2 Az adattípus fogalma

### 2.1 Alapfogalmak, jelölések

- $A^*$  az  $A$ -beli véges sorozatok halmazát,  $A^\infty$  az  $A$ -beli végtelen sorozatok halmazát jelöli. A kettő uniója  $A^{**} = A^* \cup A^\infty$  pedig az  $A$ -beli véges vagy végtelen sorozatok halmazát jelenti.
- Legyen  $R \subseteq A \times \mathbb{L}$  egy logikai reláció. Ekkor az  $R$  igazsághalmaza  $\lceil R \rceil ::= R^{-1}(\{igaz\})$
- Legyen  $I$  egy véges halmaz és legyenek  $A_i, i \in I$  tetszőleg véges vagy megszámlálható, nem üres halmazok. Ekkor az  $A = \prod_{i \in I} A_i$  halmazt állapottérnek, az  $A_i$  halmazokat pedig típusérték-halmazoknak nevezzük.
- Feladat: feladatnak nevezünk egy  $F \subseteq A \times A$  relációt.  
A feladat fenti definíciója természetes módon adódik abból, hogy a feladatot egy leképezésnek tekintjük az állapottéren, és az állapottér minden pontjára megmondjuk, hova kell belőle eljutni, ha egyáltalán el kell jutni belőle valahova.
- Program:  
Programnak nevezzük az  $S \subseteq A \times A^{**}$  relációt, ha
  1.  $\mathcal{D}_S = A$  (az állapottér minden pontjához rendel valamit, azaz a program minden pontban csinál valamit)
  2.  $\forall \alpha \in \mathcal{R}_S : \alpha = red(\alpha)$  (az állapot megváltozik, vagy ha mégsem, az az abnormális működés jele)
  3.  $\forall a \in A : \forall \alpha \in S(A) : |\alpha| \neq 0 \text{ és } \alpha_1 = a$

A fenti definícióval a "működés" fogalmát akarjuk absztrakt módon megfogalmazni.

### 2.2 Típus-specifikáció

Először bevezetünk egy olyan fogalmat, amelyet arra használhatunk, hogy pontosan leírjuk a követelményeinket egy típusérték-halmazzal és a rajta végezhető műveletekkel szemben.

A  $\mathcal{T}_S = (H, I_S, \mathbb{F})$  hármast típus-specifikációnak nevezzük, ha teljesülnek rá a következő feltételek:

1.  $H$  az alaphalmaz,
2.  $I_S : H \rightarrow \mathbb{L}$  a specifikációs invariáns,
3.  $T_{\mathcal{T}} = \{(\mathcal{T}, x) | x \in \lceil I_S \rceil\}$  a típusérték-halmaz,
4.  $\mathbb{F} = \{F_1, F_2, \dots, F_n\}$  a típusműveletek specifikációja, ahol  
 $\forall i \in [1..n] : F_i \subseteq A_i \times A_i, A_i = A_{i_1} \times \dots \times A_{i_{n_i}}$  úgy,  
 hogy  $\exists j \in [1..n_i] : A_{i_j} = T_{\mathcal{T}}$

Az alaphalmaz és az invariáns tulajdonság segítségével azt fogalmazzuk meg, hogy mi az a halmaz,  $T_{\mathcal{T}}$ , amelynek elemeivel foglalkozni akarunk, míg a feladatok halmazával azt írjuk le, hogy ezekkel az elemekkel milyen műveletek végezhetők el.

Az állapottér definíciójában szereplő típusérték-halmazok mind ilyen típus-specifikációban vannak definiálva. Az állapottér egy komponensét egy program csak a típusműveleteken keresztül változtathatja meg.

## 2.3 Típus

Vizsgáljuk meg, hogy a típusspecifikációban leírt követelményeket hogyan valósítjuk meg.

A  $\mathcal{T} = (\rho, I, \mathbb{S})$  hármast típusnak nevezzük, ha

1.  $\rho \subseteq E^* \times T$  a reprezentációs függvény (reláció),  
T a típusértékhalmoz,  
E az elemi típusértékhalmoz
2.  $I : E^* \rightarrow \mathbb{L}$  típusinvariáns
3.  $\mathbb{S} = \{S_1, S_2, \dots, S_m\}$ , ahol  
 $\forall i \in [1..m] : S_i \subseteq B_i \times B_i^{**}$  program,  $B_i = B_{i_1} \times \dots \times B_{i_{m_i}}$  úgy,  
hogy  $\exists j \in [1..m_i] : B_{i_j} = E^*$  és  $\nexists j \in [1..m_i] : B_{i_j} = T$

A típus első két komponense az absztrakt típusértékek reprezentációját írja le, míg a programhalmoz a típusműveletek implementációját tartalmazza. Az elemi típusértékhalmoz lehet egy tetszőleges másik típus típusértékhalmoza vagy egy, valamilyen módon definiált legfeljebb megszámlálható halmoz.

## 2.4 Invariáns

Az invariáns lényege, hogy ezt a tulajdonságot soha nem sérthetjük meg. Például halmoz típus esetén nem szabad, hogy megsérüljön az az invariáns tulajdonság, hogy egy halmozban egy elem csak egyszer fordulhat elő.

## 2.5 Reprezentáció

Azt, hogy egy típust milyen típusok segítségével, milyen módszerrel, stb., valósítottunk meg, reprezentációnak nevezzük. Például egy verem típust meg lehet valósítani tömb segítségével, de láncolt listával is.

A reprezentáció a típusspecifikáció típusértékhalmozának leképezése a konkrét típusban, amit a reprezentációs függvény ad meg.

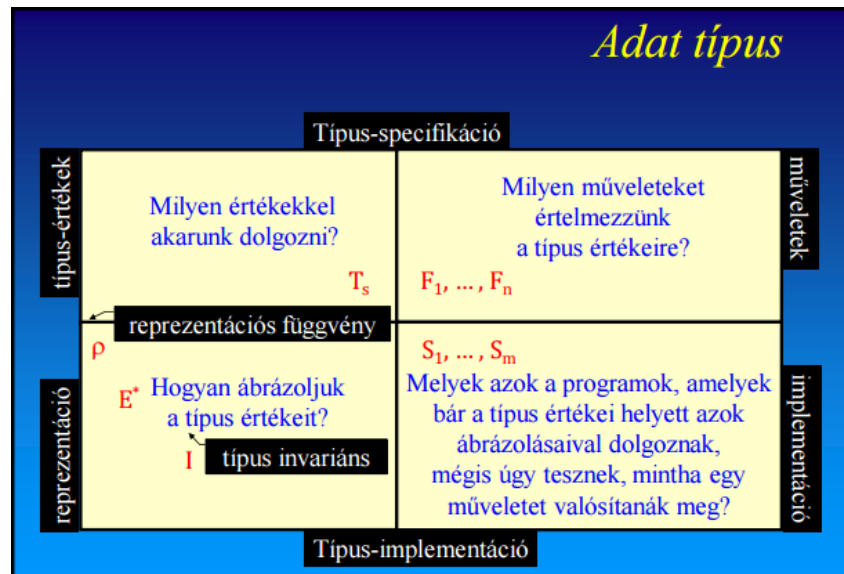
## 2.6 Implementáció

Az implementáció a típusspecifikáció típusműveleteinek megvalósítása a konkrét típus programhalmoza által.

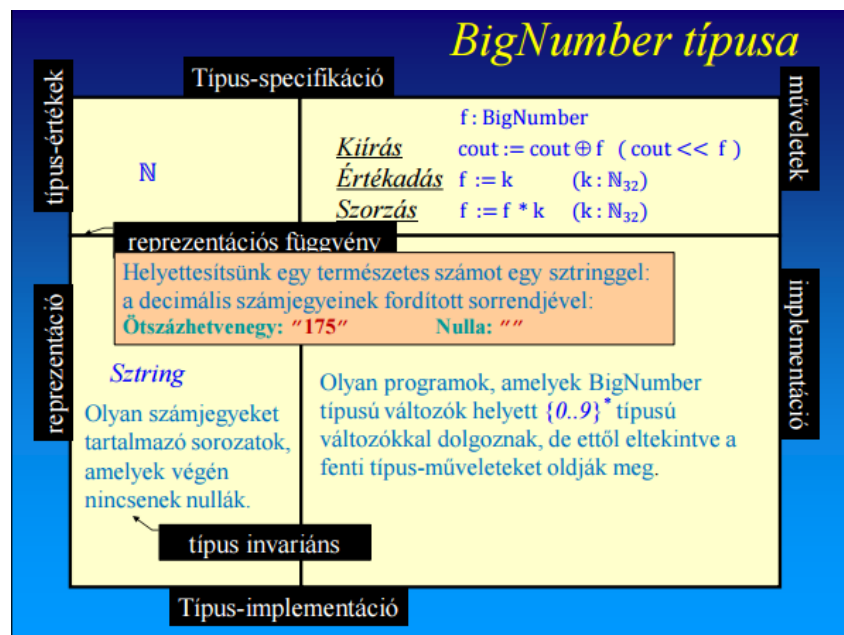
Az implementáció során a típus megvalósításakor a típusértékhalmoz megadását követően definiálni kell a típusműveleteket. Ahogyan a modellben is, a gyakorlatban is az állapottér változásait a program csak a típusműveleteken keresztül végezheti el.



## 2.7 Emészthetőbb módon



ábra 4: Adattípus



ábra 5: BigNumber példa

## 3 A visszavezetés módszere

A programozási feladatok megoldásához különböző programozási mintákat, ún. programozási tételeket használunk fel, ezekre vezetjük vissza a megoldást.

Lépései:

1. Megsejtjük a feladatot megoldó programozási tételt.
2. Specifikáljuk a feladatot a programozási tétel jelöléseivel.
3. Megadjuk a programozási tétel és a feladat közötti eltéréseket:

- intervallum határok: konkrét érték vagy kifejezés (pl.  $[1.. \frac{n}{2}]$ ), a típusuk a  $\mathbb{Z}$  helyett lehet annak valamely része (pl.  $\mathbb{N}$ )
  - $\beta : [m..n] \rightarrow \mathbb{L}$  és/vagy  $f : [m..n] \rightarrow H$  konkrét megfelelői
  - a  $H$  megfelelője a szükséges művelettel
    - $(H, >)$  helyett pl.  $(\mathbb{Z}, >)$  vagy  $(\mathbb{Z}, <)$
    - $(H, +)$  helyett pl.  $(\mathbb{Z}, +)$  vagy  $(\mathbb{R}, *)$
  - a változók átnevezése
4. A különbségek figyelembe vételével a tétel algoritmusából elkészítjük a konkrét feladatot megoldó algoritmust.

## 4 Felsoroló, a felsoroló típus specifikációja

A gyűjtemény (tároló, kollekció, iterált) egy olyan adat (objektum), amely valamilyen elemek tárolására alkalmas.

- Ilyenek az összetett szerkezetű, de különösen az iterált szerkezetű típusok értékei: halmaz, sorozat (verem, sor, fájl), fa, gráf
- De vannak úgynevezett virtuális gyűjtemények is: pl. egész számok egy intervallumának elemei, vagy egy természetes szám prím-osztói

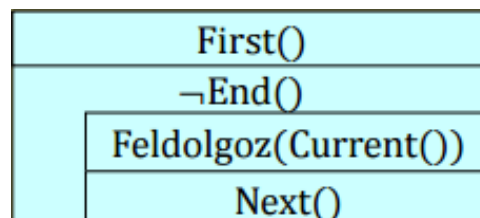
Egy gyűjtemény feldolgozásán a benne levő elemek feldolgozását értjük.

- Keressük a halmaz legnagyobb elemét!
- Hány negatív szám van egy számsorozatban?
- Válogassuk ki egy fa leveleiben elhelyezett értékeket!
- Járjuk be az  $[m .. n]$  intervallum minden második elemét visszafelé!
- Adjuk össze az  $n$  természetes szám prím-osztóit!

A feldolgozni kívánt elemek felsorolását (bejárását) az alábbi műveletekkel szabványosítjuk:

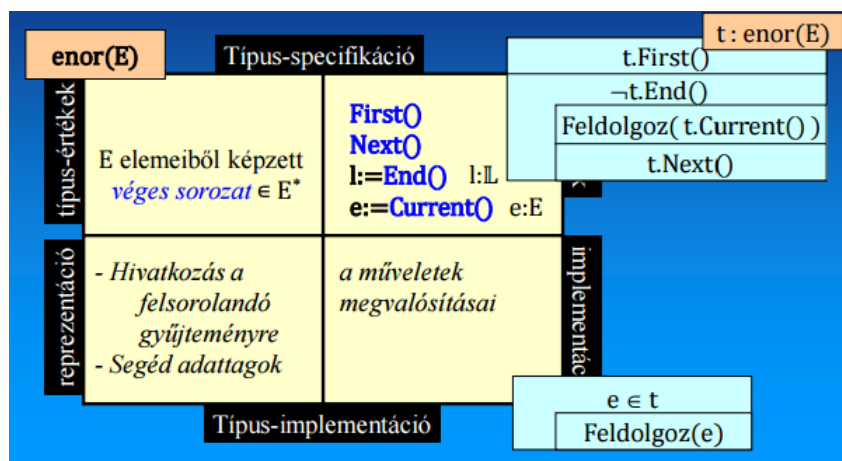
- `First()` : Rááll a felsorolás első elemére, azaz elkezd a felsorolást
- `Next()` : Rááll az elkezdett felsorolás soron következő elemére
- `End()` : Mutatja, ha a felsorolás végére értünk
- `Current()` : Visszaadja a felsorolás aktuális elemét

Egy felsorolásnak különböző állapotai vannak (indulásra kész, folyamatban van, befejeződött), és a műveletek csak bizonyos állapotokban értelmezhetők (máshol a hatásuk nem definiált). A feldolgozó algoritmus garantálja, hogy a felsoroló műveletek mindig megfelelő állapotban kerüljenek végrehajtásra.



ábra 6: A felsorolás algoritmus

A felsorolást sohasem a felsorolni kívánt gyűjtemény, hanem egy külön felsoroló objektum végzi.



ábra 7: A felsoroló objektum és típusa

## 5 Felsorolóra megfogalmazott programozási tételek

## Programozási tételek felsorolókra

### Összegzés

**Feladat:** Adott egy  $E$ -beli elemeket felsoroló  $t$  objektum és egy  $f:E \rightarrow H$  függvény. A  $H$  halmazon értelmezzük az összeadás asszociatív, baloldali nullelemes műveletét. Határozzuk meg a függvénynek a  $t$  elemeihez rendelt értékeinek összegét! (Üres felsorolás esetén az összeg értéke definíció szerint a nullelem:  $0$ ).

**Specifikáció:**

$$\begin{aligned} A &= (t: \text{enor}(E), s: H) \\ Ef &= (t=t') \\ Uf &= (s = \sum_{e \in t'} f(e)) \end{aligned}$$

**Algoritmus:**

$s := 0$ $t.First()$		
$\neg t.End()$		
<table> <tr> <td><math>s := s + f(t.Current())</math></td></tr> <tr> <td><math>t.Next()</math></td></tr> </table>	$s := s + f(t.Current())$	$t.Next()$
$s := s + f(t.Current())$		
$t.Next()$		

### Számlálás

**Feladat:** Adott egy  $E$ -beli elemeket felsoroló  $t$  objektum és egy  $\beta: E \rightarrow \mathbb{L}$  feltétel. A felsoroló objektum hány elemére teljesül a feltétel?

**Specifikáció:**

$$\begin{aligned} A &= (t: \text{enor}(E), c: \mathbb{N}) \\ Ef &= (t=t') \\ Uf &= (c = \sum_{e \in t'} 1) \\ &\quad \beta(e) \end{aligned}$$

**Algoritmus:**

$c := 0$ $t.First()$				
$\neg t.End()$				
<table> <tr> <td><math>\beta(t.Current())</math></td></tr> <tr> <td><math>c := c + 1</math></td></tr> <tr> <td><math>SKIP</math></td></tr> <tr> <td><math>t.Next()</math></td></tr> </table>	$\beta(t.Current())$	$c := c + 1$	$SKIP$	$t.Next()$
$\beta(t.Current())$				
$c := c + 1$				
$SKIP$				
$t.Next()$				

### Maximum kiválasztás

**Feladat:** Adott egy  $E$ -beli elemeket felsoroló  $t$  objektum és egy  $f:E \rightarrow H$  függvény. A  $H$  halmazon definiáltunk egy teljes rendezési relációt. Feltesszük, hogy  $t$  nem üres. Hol veszi fel az  $f$  függvény a  $t$  elemein a maximális értékét?

**Specifikáció:**

$$\begin{aligned} A &= (t: \text{enor}(E), max: H, elem: E) \\ Ef &= (t=t' \wedge |t| > 0) \\ Uf &= ((max, elem) = \max_{e \in t'} f(e)) \end{aligned}$$

**Algoritmus:**

$t.First()$				
$max, elem := f(t.Current()), t.Current()$				
$t.Next()$				
$\neg t.End()$				
<table> <tr> <td><math>f(t.Current()) &gt; max</math></td></tr> <tr> <td><math>max, elem := f(t.Current()), t.Current()</math></td></tr> <tr> <td><math>SKIP</math></td></tr> <tr> <td><math>t.Next()</math></td></tr> </table>	$f(t.Current()) > max$	$max, elem := f(t.Current()), t.Current()$	$SKIP$	$t.Next()$
$f(t.Current()) > max$				
$max, elem := f(t.Current()), t.Current()$				
$SKIP$				
$t.Next()$				

### Kiválasztás

**Feladat:** Adott egy  $E$ -beli elemeket felsoroló  $t$  objektum és egy  $\beta:E \rightarrow \mathbb{L}$  feltétel. Keressük a  $t$  bejárása során az első olyan elemi értéket, amely kielégíti a  $\beta:E \rightarrow \mathbb{L}$  feltételt, ha tudjuk, hogy biztosan van ilyen.

**Specifikáció:**

$$\begin{aligned} A &= (t.enor(E), elem:E) \\ Ef &= (t=t' \wedge \exists i \in [1..|t|]: \beta(t_i)) \\ Uf &= ((elem, t) = \underset{elem \in t'}{\text{select}} \beta(elem)) \end{aligned}$$

**Algoritmus:**

$t.First()$
$\neg \beta(t.Current())$
$t.Next()$
$elem := t.Current()$

### Lineáris keresés

**Feladat:** Adott egy  $E$ -beli elemeket felsoroló  $t$  objektum és egy  $\beta:E \rightarrow \mathbb{L}$  feltétel. Keressük a  $t$  bejárása során az első olyan elemi értéket, amely kielégíti a  $\beta:E \rightarrow \mathbb{L}$  feltételt.

**Specifikáció:**

$$\begin{aligned} A &= (t.enor(E), l:\mathbb{L}, elem:E) \\ Ef &= (t=t') \\ Uf &= ((l, elem, t) = \underset{e \in t'}{\text{search}} \beta(e)) \end{aligned}$$

**Algoritmus:**

$l := \text{hamis}; t.First()$
$\neg l \wedge \neg t.End()$
$elem := t.Current()$
$l := \beta(elem)$
$t.Next()$

### Feltételes maximumkeresés

**Feladat:** Adott egy  $E$ -beli elemeket felsoroló  $t$  objektum, egy  $\beta:E \rightarrow \mathbb{L}$  feltétel és egy  $f:E \rightarrow H$  függvény. A  $H$  halmazon definiáltunk egy teljes rendezési relációt. Határozzuk meg  $t$  azon elemeihez rendelt  $f$  szerinti értékek között a legnagyobbat, amelyek kielégítik a  $\beta$  feltételt.

**Specifikáció:**

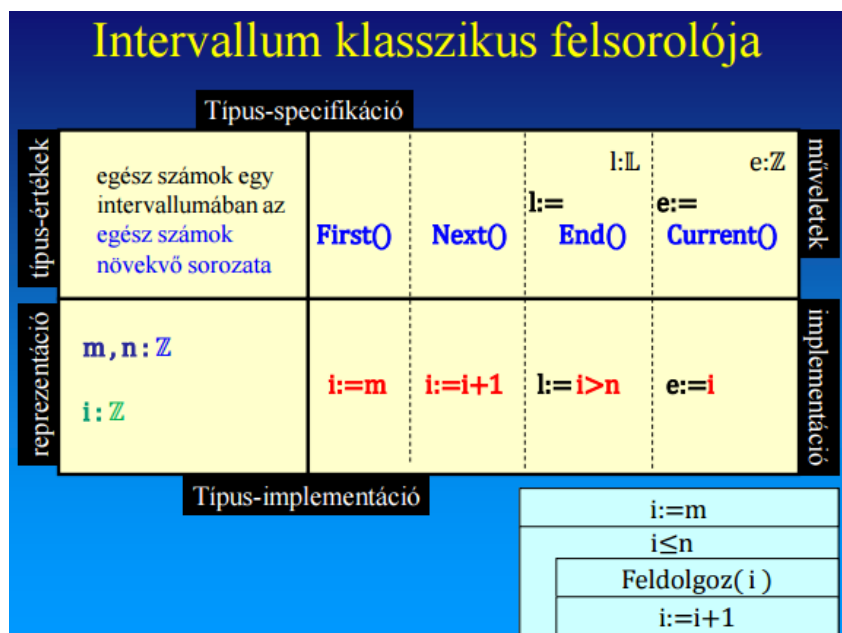
$$\begin{aligned} A &= (t.enor(E), l:\mathbb{L}, max:H, elem:E) \\ Ef &= (t=t') \\ Uf &= ((l, max, elem) = \underset{\substack{e \in t' \\ \beta(e)}}{\text{max}} f(e)) \end{aligned}$$

**Algoritmus:**

$l := hamis; t.First()$			
$\neg t.End()$			
$\neg \beta(t.Current())$	$\beta(t.Current()) \wedge l$		$\beta(t.Current()) \wedge \neg l$
SKIP	$f(t.Current()) > max$		$l, max, elem :=$ $igaz, f(t.Current()), t.Current()$
	$max, elem :=$ $f(t.Current()), t.Current()$	SKIP	
$t.Next()$			

## 6 Nevezetes gyűjtemények felsorolói

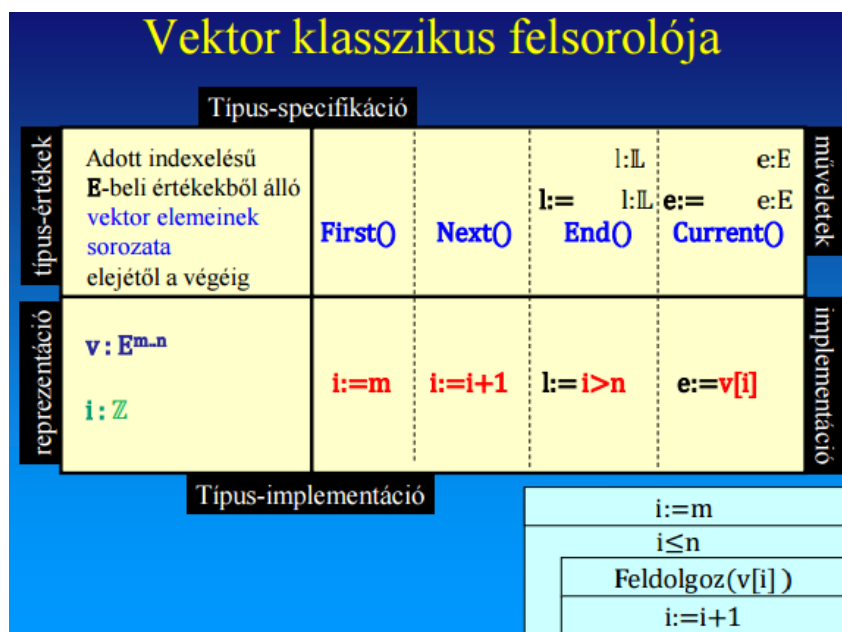
### 6.1 Intervallum



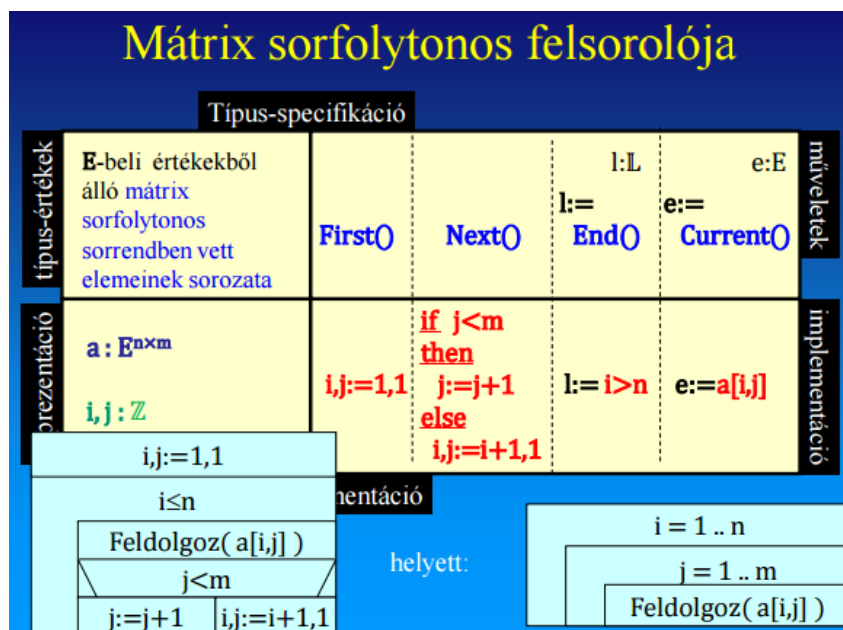
ábra 8: Intervallum felsorolója

### 6.2 Tömb

Itt két különböző tömbtípus felsorolóját mutatjuk be: az egydimenziós (vektor) és a kétdimenziós tömböt (mátrix).



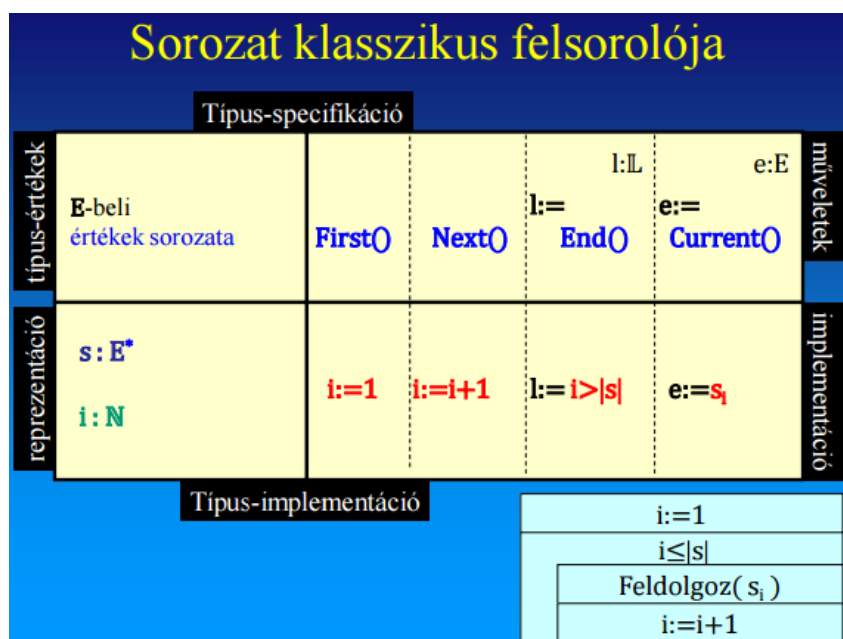
ábra 9: Vektor felsorolója



ábra 10: Mátrix sorfolytonos felsorolója

Megjegyzés: a felsorolás történhet másképpen is, például vektor esetén végezhetjük a felsorolást visszafelé, a tömb végétől kezdve, vagy mátrixnál alkalmazhatunk pl. oszlopfolytonos bejárást.

### 6.3 Sorozat



ábra 11: Sorozat felsorolója

## 6.4 Halmaz

Halmaz felsorolója					
Típus-specifikáció					
tipus-értékek	halmaz $E$ -beli elemeinek sorozata	First()	Next()	$l:L$ $l:=$ End() $e:=$ Current()	műveletek
reprezentáció	$h : \text{set}(E)$	—	$h := h - \{\text{mem}(h)\}$	$l := h = \emptyset$ $e := \text{mem}(h)$	implementáció
Típus-implementáció					
<div>—</div> <div><math>h \neq \emptyset</math></div> <div>Feldolgoz( <math>\text{mem}(h)</math> )</div> <div><math>h := h - \{\text{mem}(h)\}</math></div>					

ábra 12: Halmaz felsorolója

## 6.5 Szekvenciális inputfájl

Szekvenciális inputfájl felsorolója					
Típus-specifikáció					
tipus-értékek	szekvenciális inputfájl E-beli elemeinek sorozata	First()	Next()	$l:\mathbb{L}$ $l:=$ End() $e:=$ Current()	műveletek  <

ábra 13: Szekvenciális inputfájl felsorolója