Adatszerkezetek és algoritmusok

Felsőfokú szakirányú továbbképzés

Egyetemi jegyzet

2010

Tartalomjegyzék

Tárgy célja

Ebben a jegyzetben a Pázmány Péter Katolikus Egyetem - Információs Technológiai Karán esti képzés keretein belül oktatott adatszerkezetek és algoritmusok tantárgy előadásain és gyakorlatain elhangzott anyagokat igyekeztem összegyűjteni, rendszerezni. Mivel a tárgy viszonylag rövid idő alatt több alapvető programozáselméleti, algoritmikus és adatstruktúrákkal kapcsolatos tudást nyújt, ezért több esetben a törzsanyaghoz további magyarázatok tartoznak, a teljesség igénye nélkül. A tárgyat olyanok hallgatják, akik valamilyen mérnöki (villamosmérnök, informatikai mérnök/szakember) vagy a szakiránynak megfelelő (matematikai, gazdasági) tudással rendelkeznek, ezért néhány esetben igyekeztem olyan példákat mutatni, amelyek ezekhez a korábbi tanulmányokhoz köthetők.

Az elméleti tudás mellé, párhuzamosan, a JAVA nyelv is ismertetésre kerül alapszinten. A cél, az hogy miniprogramok írására, algoritmikus megtervezésére bárki képes legyen a tárgy elvégzése után. (Természetesen akadnak olyanok is akik korábban már tanultak programozni, más nyelven. A jegyzet elkészítése során igyekeztem nekik is „kedvükben járni” egy-egy különbség, csemege említésével.)

Röviden felsorolva a tárgy alapvető célkitűzéseit

- Algoritmikus szemlélet kialakítása, néhány példán keresztül bemutatva a programozás mibenlétét.
- Programozási alapstruktúrák megismerése, nyelvfüggetlenül, hogy későbbiekben más szintaktikára is átültethető legyen a tudás. (Természetesen példákhoz a JAVA nyelvet használjuk a jegyzetben.)
- Java programozási nyelv alapismeretek
- Alapvető adatszerkezetek megismerése valamint implementációja.
- Rendező algoritmusok analitikus vizsgálata, implementációja.

1. fejezet

Bevezetés a programozásba

1.1. Algoritmusok

1.1.1. Mit jelent programozni?

Legelsőként azt a kérdést járjuk körül, hogy milyen részfeladatokat rejt magában a programozás, mint tevékenység. A programozás feladata nemcsak az utasítások kódolását foglalja magában, hanem annál több, a feladat illetve rendszer megtervezésére vonatkozó problémák megoldását is jelenti.

- **Részletes megértés**: A feladatot, amit meg kell oldani csak úgy tudjuk a számítógép számára érthető formában, programkódként elkészíteni, ha mi saját magunk is tisztában vagyunk a követelményekkel, a probléma részleteivel. Ez egy sokrétű és több módszer ismeretét és alkalmazását igénylő feladat, hiszen a problémát gyakran nem mi magunk találjuk ki, hanem a szoftver „megrendelésre" kell, hogy elkészüljön, valaki más elképzelései alapján. (Ezzel részletesebben a szoftvertervezés során foglalkozunk.)
- **Rendszer tervezése**: Bonyolultabb esetekben a programunk nem „egy szuszra" fog elkészülni, hanem több részfeladatból, részprogramból áll össze, amelyeket szintén meg kell terveznünk. A különböző rendszeregységek és feladataik szintén tervezés igényelnek. (Itt nemcsak a részek, mint önálló egységek, hanem azok összekapcsolódása, illetve azok elkészülési sorrendje is fontos.)
- **Receptkészítés**: Az egyes egységek, illetve a program egészére el kell végeznünk az algoritmusok, módszerek gép számára érthető módon történő leírását. Ez azt jelenti, hogy az egyes problémák megoldására el kell készítenünk a legapróbb lepésekből álló sorozatot, amely lépéssorozatot a gép is megért és végrehajt. (Jellemzően ezen egyszerű utasítások, vagy olyan már előre megírt funkciók, amelyeket szintén elemi lépésekből állnak össze.)
- **Módszer keresése**: A probléma megoldásához szükségünk van a megoldás elkészítéséhez, kiszámításához módszerekre. Például egy szám négyzetgyökét különböző közelítő numerikus számításokkal lehet megoldani. Egy másik módszer lehet az, hogy egy táblázatban eltároljuk az megoldásokat és egy keresési problémává alakítjuk át a négyzetgyök-számítás feladatát. (Természetesen itt megkötésekkel kell élnünk, mivel az összes lehetséges valós szám és négyzetgyökének párosaiból álló táblázat végtelen mennyiségű tárhelyet igényelne.)
- **Matematika**: A számítógép – természetéből fakadóan – többnyire matematikai műveleteket ismer, abból is az alapvetőeket. Az előző példára visszatérve, a négyzetgyök

kiszámítására egy lehetséges közelítő módszer a Newton-Rhapson-féle iterációs módszer. Ebben az esetben ismételt műveletekkel egyre pontosabb eredményt kapunk a számítás során, ami elemi matematikai lépésekből áll. Más feladatok esetén másféle matematikai modellre, módszerre van szükségünk.

Egy hétköznapi példát vegyünk, amely egy robotot arra utasít, hogy az ebéd elkészítéséhez szerezze be az egyik összetevőt, a burgonyát. Természetesen a feladatot egyetlen utasításban is meg lehet fogalmazni, de ahhoz hogy ezt végrehajtani képes legyen finomítani kell, további lépésekre kell bontani.

- Hozz krumplit!
- Hozz egy kilogramm krumplit!
- Hozz egy kilogramm krumplit most!
- Hozz egy kilogramm krumplit a sarki közértből most!
- Menj el a sarki közértbe, végy egy kosarat, tegyél bele egy kilogramm krumplit, adj annyi pénzt a pénztárosnak, amennyibe a krumpli kerül, tedd le a kosarat, gyere ki a közértből, s hozd haza a krumplit.

Nyilván ez sem lesz elég, hiszen a haladással, a fizetéssel és további műveletekkel kapcsolatosan még finomabb lépésekre kell bontani. (Jelenleg egyetlen robot sem rendelkezik olyan mesterséges intelligenciával, hogy ezt a problémát önállóan megoldja.)

Amikor egy feladatot megoldunk, egy algoritmust elkészítünk, a megoldás kulcsa a lépések meghatározásában rejlik. Hasonlóan a bevásárlás példájából az algoritmusunkat egyre finomabb lépésekre tudjuk felbontani.

1.1.2. Mi az algoritmus?

Az algoritmus olyan lépések sorozata, amely megold egy jól definiált problémát. (Itt érdemes azt megjegyezni, hogy a probléma jól definiáltsága olyan kritérium, amelyet nem minden esetben sikerül teljesíteni. Előfordul, hogy a nem ismerjük jól a problémát, vagy csak az algoritmus kidolgozása során veszünk észre olyan helyzeteket, amelyeket a feladat kitűzésénél nem vettek figyelembe.) A következő néhány pontban az algoritmus számítógépes vonzatait és tulajdonságait vizsgáljuk.

- A számítógépes algoritmus (elemi) utasítások sorozata a probléma megoldására. Itt már további lépésekre utasításokra bontjuk a megoldásunkat, a korábbiakban írtakkal összhangban.
- Jó algoritmus kritériumai
  - Helyesség – vizsgáljuk, hogy a megalkotott algoritmus, a feladatkiírás feltételei mellett, minden esetre jó megoldást ad-e.
  - Hatékonyság – vizsgáljuk, hogy az algoritmus a rendelkezésre álló különböző erőforrásokkal mennyire bánik gazdaságosan.
- Algoritmus és program kapcsolata, algoritmusok lehetséges leírása
  - Pseudo kód – olyan kód, amit az emberek értenek meg, de már a számítógép számára is megfelelően elemi utasításokat tartalmaz.
  - Programnyelvi kód – amelyet begépelve szintaktikailag érvények kódot kapunk, *fordíthatjuk*[1] és futtathatjuk.

[1] Az a művelet, amikor a programnyelvi kódot a programozást támogató környezetek, programok a számítógép számára közvetlenül értelmezhető utasításokká, programmá alakítjuk.

Algoritmusok helyessége

Egy algoritmus helyes, ha a kitűzött feladatot korrekt módon oldja meg. Azaz a feladatspecifikációban meghatározott megszorításokat figyelembe véve, minden lehetséges esetre, bemenetre (inputra) megfelelő eredményt, kimenetet (output) szolgáltat. A helyességet különböző technikákkal lehet bizonyítani, ezek többsége azonban nagyon költséges módszer.

- Ellenpéldával bizonyítás – cáfolat, ez a legegyszerűbb, azonban ellenpélda találása nem mindig könnyű. Természetesen, amennyiben nem találunk ellenpéldát, az nem jelenti azt, hogy a kód bizonyosan helyes.
- Indirekt bizonyítás – ellentmondással bizonyítás, ahogyan azt a matematikában megszoktuk. Ellentmondást keresünk az állítások, eredmények között, amihez abból a feltételezésből indulunk ki, hogy a megoldásunk helytelen.
- Indukciós bizonyítás – ha n-re helyes megoldást ad és ebből következik az, hogy $n+1$-re is helyes megoldást fog adni illetve igaz továbbá, hogy $n = 1$ helyes a megoldást, akkor bizonyítottuk, hogy minden lehetséges inputra jól fog működni az algoritmusunk. (Például a faktoriális számítás esetén gondolkodhatunk így.)
- Bizonyított elemek használata – levezetés, olyan részprogramokat használunk, amelyek bizonyítottan jól működnek. A további elemek, részprogramok helyes működésének *formális* bizonyítása a mi feladatunk.

Algoritmusok hatékonysága

Egy algoritmus hatékony, ha nem használ több erőforrást, mint amennyi föltétlen szükséges a feladat megoldásához. Ezek az erőforrások legtöbbször a használt memória mennyisége, valamint az idő (processzor használati ideje). Legtöbbször igaz, hogy a futási idő javításához több memóriára van szükség, valamint kevesebb memóriahasználatot több lépéssel tudunk csak elérni. Az alábbi felsorolással összefoglaljuk, hogy milyen módszerekkel lehetséges a hatékonyságot vizsgálni

- Benchmark – futtatás és időmérés. A módszer csak a megvalósított algoritmusok esetén használható, tehát előzetes becslést a hatékonyságra nem lehet végezni vele.
- Elemzés, ami az alábbiakat foglalja magában
  - Műveletek megszámolása, egy meghatározott (legjobb/legrosszabb/átlagos) esetben mennyi művelet végrehajtására van szükség az eredmény megadásához. A műveletek számát, az megadott input méretéhez viszonyítva nagyságrendileg szokás megadni. (Ezt a későbbiekben fogjuk használni.)
  - Komplexitás elemzés – az algoritmus bonyolultságának vizsgálata.

1.1.3. Feladat specifikálása

A feladatspecifikáció az alábbi hármasból áll

- Leírom, hogy milyen adat áll rendelkezésre a feladat megoldásához.
- Leírom, hogy milyen adatokat/eredményeket kell kapnom a feladat megoldása végén.
- Leírom, hogy mit kell végeznie a programnak, azaz mi a feladat (Emberi nyelven).

Ezek mindegyike egy-egy feltétel. Például kiköti, hogy a rendelkezésre álló adatok milyen típusúak, milyen értéktartományból kerülhetnek ki. A megkötések például a helyességvizsgálatkor is fontosak.

Példa a négyzetgyök számításra

Bemenet (input):	a szám, aminek a négyzetgyökét keressük: x
Kimenet (output):	egy olyan szám y, amire igaz, hogy $y^2 = x$ illetve $\sqrt{x} = y$
Feladat (program):	a program számolja ki a bemenet négyzetgyökét

Miután specifikáltuk a feladatot lehet részletesebben vizsgálni a problémát. Az előző példánkat folytatva:

- Mit csinálok, ha az x negatív ...
- Mit csinálok, ha az x pozitív, vagy nulla ...

Ezekre a kérdésekre, valamint azt, hogy hogyan lehet algoritmikusan megfogalmazni és leírni keressük továbbiakban a választ.

1.2. Alapfogalmak

Mindenekelőtt néhány alapfogalmat fogunk definiálni.

1.2.1. Típus, kifejezés, változó

Típusnak nevezzük egy megadott értékhalmazt és az azokon értelmezett műveletek összességét.

Egy programozási nyelvben, algoritmusban minden egyes értéknek van aktuális típusa. Ez alapján dönthető el, hogy mit jelent az az érték, milyen műveletek érvényesek, és hogy azokat a műveleteket hogyan kell elvégezni.

Egyszerű esetben, ha a számokra gondolunk ezek a kérdések trivialitások. Bonyolultabb összetett típusok esetén azonban feltétlen vizsgálni kell.

Példák

Egészek:	Értékek: $0 \dots 1000$	Műveletek: $+$ és $-$
Logikai:	Értékek: `igaz`, `hamis`	Műveletek: $\wedge$, $\vee$ és $\neg$
Karakter:	Értékek: `a` ... `z`	Műveletek: $<$ (reláció a betűrend szerint)

A **kifejezés** ...

- olyan műveleteket és értékeket tartalmaz, amelyek együttesen értelmezhetőek, van jelentésük.
- esetén is beszélünk típusról, például az $1 + 1$ egy szám típusú kifejezés.
- lehet összetett kifejezés, ahol a részkifejezések is érvényes kifejezések. Például $5 + 6 * (7 + 8)$.

A kifejezéseket szét lehet bontani a kiértékelés szerint. A kiértékeléshez a műveletek precedenciáját (sorrendjét) kell figyelembe venni, illetve a zárójelezést.

Például

$5 + 6 * (7 + 8)$ | $(5 + 6) * (7 + 8)$

$5 + 6 * \underbrace{(7 + 8)}$ | $\underbrace{(5 + 6)} * \underbrace{(7 + 8)}$

Látható, hogy a zárójelezés hatására megváltozik a részkifejezésekre bontás, illetve a kiértékelés is.

A **változó** egy névvel (*címkével*) jelölt, adott típushoz tartozó elem. A memóriában egy számára fenntartott területre kerül a változó aktuális értéke. Egy változónak mindig van aktuális értéke, így nem teljesen ugyanaz, amit a matematikai alapfogalmaink során ismeretlenként használunk. (Ez fontos, mivel olyankor is van valami értéke, amikor még nem adtunk neki. Ez az érték azonban előre ismeretlen, futásonként más és más, a memóriacella külső hatások miatti, vagy egy előző program után ottmaradt értékét jelenti.)

- A változónak van neve és típusa
- Kifejezésben is szerepelhet: $1+x$. Ez csak akkor érvényes, ha létezik olyan $+$ művelet az x változóhoz, hogy számmal összeadás (Praktikusan x például szám)

Mielőtt használatba veszünk egy változót, jeleznünk kell a változó bevezetését. Ezt hívjuk *deklarációnak*. A deklarálás után a programnyelvet értelmező rendszer ismeri a változót – *addig nem*. Minden típusú változónak van egy alapművelete, az értékadás, tehát értéket bármilyen változónak adhatunk. Azonban ügyelni kell arra, hogy az értékadás, mint kifejezés helyes legyen. Az értékadás során határozzuk meg, hogy mit tartalmazzon a hozzárendelt memóriacella. Az alábbiakban néhány példát láthatunk Java nyelven a változók használatára: (Java-ban a programkód folyamának gyakorlatilag tetszőleges pontján bevezethetünk változókat, nincs szükség azokat egy programblokk elejére tenni. Mindig a típus neve szerepel először, majd a használni kívánt változók nevei.)

Deklaráció példa – Java nyelven

```
int x;
int y;
double pi;
```

Létrehozunk három változót a memóriában, amelyek kezdő értéke ismeretlen, mivel nem adtuk meg. A változók `int` illetve `double` típusúak, amelyek egész valamint racionális számok. A változók nevei rendre `x`, `y`, `pi`.

Értékadás példa – Java nyelven

```
x = 10;
y = 10 * x;
pi = 3.14;
```

Az előzőekben deklarált változók értéket kapnak. Az `x` értéke egy előre rögzített szám. Az `y` értéke azonban már számított érték egy másik változótól függ. Természetesen a program futása során minden egyes pillanatban ez az érték pontosan meghatározható.

Vegyes típusú kifejezésről akkor beszélünk, ha az egyes részkifejezések típusa különböző.

Példa a vegyes típusú kifejezésre

$x < 5$

Az $<$ operátor (művelet) olyan, hogy (itt) két számot fogad, ám logikai típusú eredményt ad vissza. (A logikai típus kétféle értéket vehet fel: `igaz` vagy `hamis`.) Amit matematikailag megszokhattunk érvényes kifejezésnek, az több esetben nem helyes a programozás szempontjából. Vegyük az alábbi hibás példát:

Hibás példa

$2 < x < 5$

Ha felbontjuk, akkor a $2 < x$ eredménye logikai, azonban egy logikai és egy szám között viszont nem értelmezhető a $<$ operátor. Javítani úgy lehet, hogy két logikai kifejezést kötünk össze egy újabb operátorral.

Hibás példa javítása
$(2 < x)\&\&(x < 5)$

Ahol az $\&\&$ operátor a *logikai és* operátor ($\wedge$). Az és akkor és csak akkor igaz, ha mindkét oldalán levő érték `igaz`. Az előző példát megvizsgálva csak akkor lesz igaz a teljes kifejezés értéke, ha az `x` egyidejűleg kisebb, mint öt és nagyobb, mint kettő, ami megfelel az eredeti matematikailag leírt kifejezésnek.

1.3. Vezérlési szerkezetek

A jegyzet ezen részében a programozás során használt elemi szerkezetekkel ismerkedünk meg. Továbbá a Java nyelven történő használatukkal.

1.3.1. Szekvencia

A szekvencia a legegyszerűbb programkonstrukció. Lényege, hogy a program adott részében egymás után végrehajtandó utasítások kerülnek felsorolásra, amely a felsorolás rendjében kerül majd futtatásra. A legegyszerűbb algoritmusok is tartalmaznak legalább egy szekvenciát, ami akár egyetlen utasításból is állhat. A szekvenciákban lehetőségünk van a legtöbb programozási nyelv esetén további szekvenciákat leírni. A belső szekvenciák a külső számára egyetlen egységként értelmezendőek, tehát ahova lehet programutasítást vagy lépés írni, ugyanoda szekvencia is elhelyezhető. Vegyük az alábbi példát: (Java-ban a szekvenciát, hasonlóan a c++-hoz kapcsos zárójelek közé tesszük. Ez hasonlít például a **`begin-end`** párosra.)

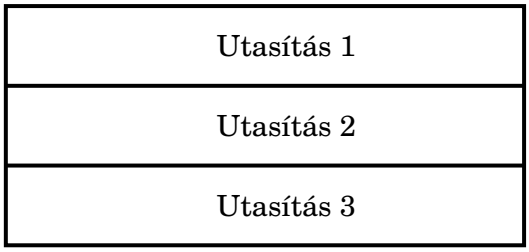
Szekvencia példa

`{`	Szekvencia eleje
`int x;`	Létrejön x
`int y;`	Létrejön y
`x = 10;`	x-be 10 került
`y = 10 * x;`	y-ba $x * 10 = 100$ került
`x = 20;`	x-be 20 került
`}`	Szekvencia vége

A szekvenciát rajzosan az alábbi módon lehet ábrázolni, struktogrammal:

1.3.2. Egyszeres elágazás

Algoritmusok során előfordul, hogy egy változó értékétől, vagy a műveletek elvégzése során az aktuális állapottól függően adódnak teendők. Például egy abszolútértéket számító algoritmus attól függően, hogy a bemeneti érték negatív vagy sem más-más utasítást hajt végre. (Ha a bemenet nagyobb vagy egyenlő nullával, akkor a kimenet a bemenettel lesz egyenlő. Ellenkező esetben meg kell szorozni -1-el.



1.1. ábra. Szekvencia struktogram

Abszolútérték számítás – elágazás példa
$y = |x|$, ha $x \geq 0$ akkor $y = x$, különben $y = -1 * x$

Az egyszeres elágazás szintaktikája Java nyelven az alábbi (ahol az **`else`** előtti utasítás végén, kivéve, ha szekvencia, kötelező a `;`)

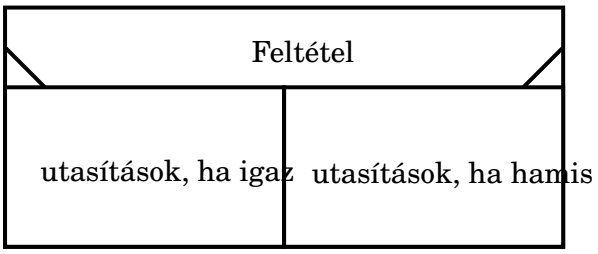
Egyszeres elágazás – Java
`if` (*`logikai feltétel`*) *`utasítás`* **`else`** *`más utasítás`*

A feltétel logikai típus, bármi lehet. Az **`if`** rész után következik egy utasítás, vagy utasításszekvencia, ami akkor fut le, ha a feltétel igaz volt. Az **`else`** rész után következik egy utasítás, vagy utasítás-szekvencia, ami akkor fut le, ha a feltétel hamis volt. Az **`else`** elhagyható!

Egyszeres elágazás – Java – abszolútérték

```
if (x < 0)
y = -1 * x;
else
y = x;
```

Az elágazást rajzosan az alábbi módon lehet ábrázolni, struktogrammal illetve folyamatábrával.

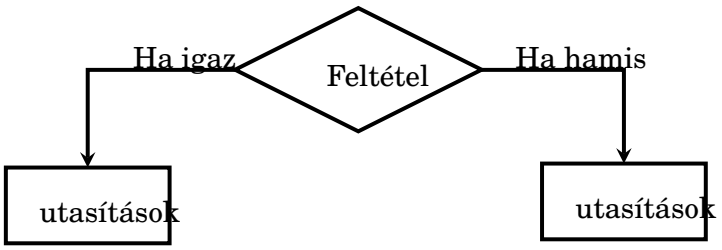


1.2. ábra. Egyszeres elágazás struktogram

1.3.3. Többszörös elágazás

Hasonlóan az egyszeres elágazáshoz itt is a feltételnek megfelelően kell cselekedni. Itt több lehetséges esetet is fel lehet sorolni:

Többszörös elágazás – Java



1.3. ábra. Egyszeres elágazás struktogram

```
switch (kifejezés)
{
case 1. eset: utasítás break;
case 2. eset: utasítás break;
default: utasítások break;
}
```

A **case**-ek száma tetszőleges. A **break** elhagyható, ám akkor a következő eset(ek)hez tartozó utasítások is lefutnak („átcsorog"). A **default** az alapértelmezett eset, elhagyható. Tekintsük az alább elágazási szerkezetet:

Többszörös elágazás problémája

```
if (x == 1)
nap = "Hétfő";
else if (x == 2)
nap = "Kedd";
else if (x == 3)
nap = "Szerda";
```

Ugyanez többszörös elágazással:

Többszörös elágazás problémája

```
switch (x)
{
case 1: nap = "Hétfő"; break;
case 2: nap = "Kedd"; break;
case 3: nap = "Szerda"; break;
case 4: nap = "Csütörtök"; break;
case 5: nap = "Péntek"; break;
case 6:
case 7: nap = "Hétvége"; break;
default: nap = "Nincs ilyen nap!"; break;
}
```

1.3.4. Ciklus(ok)

Sokszor a cél elérése érdekében ugyanazt a műveletsort kell többször egymás után elismételni, például a szorzás, faktoriális számítása, szöveg sorainak feldolgozása (hány betű

van egy sorban), . . . A ciklusokkal lehetőség van arra, hogy bizonyos szekvenciákat, utasításokat ismételtessünk.

Elöltesztelő ciklus. Amíg a logikai feltétel igaz, a *ciklusmag*ban található utasítást, vagy utasítás-szekvenciát ismétli:

Elöltesztelő WHILE ciklus

```
while (logikai feltétel)
ciklusmag
```

Elöltesztelő WHILE ciklus – példa

```
int x = 0;
while (x < 10)
x = x + 1;
```

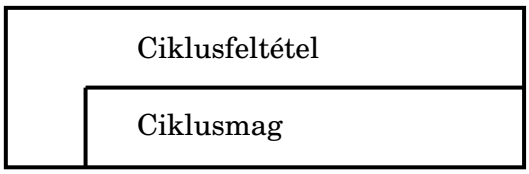
(x a ciklusváltozó ebben az esetben, valamint a ciklusmag az `x = x + 1`). A példakód az x értékét növeli egyesével kilencig.

Hátultesztelő ciklus. Amíg a logikai feltétel igaz, a *ciklusmag*ban található utasítást, vagy utasítás-szekvenciát ismétli, de legalább egyszer lefut a ciklusmag. (Fontos, hogy amíg a feltétel `igaz`, ellentétben más programozási nyelveken megszokottal.)

Elöltesztelő WHILE ciklus

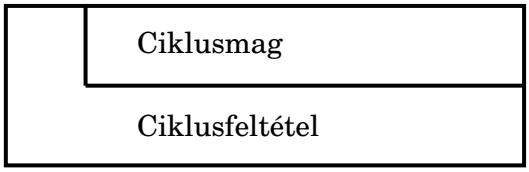
```
do
ciklusmag
while (logikai feltétel)
```

Elöltesztelő esetben, ha a feltétel hamis, egyszer sem fut le. (Először gondolkodunk, aztán cselekszünk, majd újra gondolkodunk . . .)



1.4. ábra. Elöltesztelő ciklus struktogram

Hátultesztelő esetben, ha a feltétel hamis, egyszer akkor is lefut. (Először cselekszünk, aztán gondolkodunk, hogy kell-e újra . . .)



1.5. ábra. Hátultesztelő ciklus struktogram

FOR ciklus. Vegyük a következő WHILE ciklust, ami kiszámolja 2^{10}-t.

Elöltesztelő WHILE ciklus

```
int x = 0;
int y = 1;
while (x < 11)
{
x = x + 1;
y = y * 2
}
```

Ez a következő módon rövidíthető le:

FOR ciklus

```
int y = 1;
for (int x = 0; x < 11; x = x + 1)
y = y * 2;
```

Formálisan:

FOR ciklus

```
for (inicializálás; logikai feltétel; léptetés)
ciklusmag
```

Az inicializálás a ciklusba való belépéskor fut le. A logikai feltétel vizsgálata minden egyes ciklusmag–futtatás előtt megtörténik. A léptetés pedig a ciklusmag lefutása után következik. A léptetés után újra kiértékeli a feltételt, és ennek megfelelően lép be a ciklusba.

Pár szó a ciklusról. A ciklusnak mindig van egy kezdeti állapota, egy feltétele és egy ciklusmagja. Az állapottól függ, hogy egyáltalán belépünk-e a ciklusba. Célszerűen a ciklusmag változtatja az állapotot, vagyis befolyásolja az ismétlődés feltételéül szolgáló állapotot. Könnyen lehetséges egy olyan ciklus írása, amely sosem áll le:

Végtelen ciklus

```
int x = 0;
int y = 0;
while (x < 10)
{ y = x + 1; }
```

Mivel az x értéke sosem változik, mindig kisebb lesz, mint 10, azaz örökké ismétlődik a ciklusmag. A végtelen ciklusok gyakran hibák, vagy hibás módon (például egy olyan esetben következik be, amire az algoritmus tervezése során nem számítottunk) működő algoritmusok eredményei. (Persze vannak kivételek, van, hogy a végtelen ciklus hasznos, például a felhasználó beavatkozására tétlenül vár a program.)

1.4. Programozási tételek

Ismerjük az alapvető programszerkezeteket, a szekvenciát, a ciklust és az elágazást. Ezek és tetszőleges kombinációjuk segítségével bármilyen program megírására képesek va-

gyunk, azaz ezt a sort követő része a jegyzetnek teljesen felesleges is lehetne.

A továbbiakban hasznos konstrukciókkal, bizonyítottan helyesn működő algoritmusokkal, adatszerkezetekkel fogunk foglalkozni. Először is a legalapvetőbb „tételnek” nevezett algoritmusokkal ismerkedünk meg.

1.4.1. Összegzés tétele

Bizonyos, sorban érkező értékek összegzésére szolgál. Például kapunk egy számsorozatot, aminek az matematikai összegére vagyunk kíváncsiak. A tételben szereplő részek közül az alábbiak tetszőlegesen cserélhetők más műveletre:

- Kezdőérték – milyen számról kezdjük az összegzést
- Összegzőfüggvény / – operátor (Lehetséges, hogy az elemeket nem összeadni, hanem összeszorozni akarjuk, vagy az összegzés előtt még valamilyen leképezést (függvényt) szeretnénk meghívni.

Az összegzés tétele használható

- Átlag
- Összeg ($\sum$)
- Szorzat ($\prod$)
- Szorzatösszeg
- Vektorszorzat
- Faktoriális
- ...

számítására.

Összegző – FOR ciklussal

```
int osszeg = 0;
for (int i = 0; i < értékek száma; i++)
osszeg = osszeg + következő érték;
```

Összegző – WHILE ciklussal

```
int osszeg = 0;
int i = 0;
while (i < értékek száma)
{
osszeg = osszeg + függvény(következő érték)
i += 1;
}
```

1.4.2. Számlálás tétele

Bizonyos, sorban érkező értékek leszámlálására szolgál, valamint a „mennyi?”, „hány?” kérdések megválaszolása. A tételben szereplő részek közül az alábbiak tetszőlegesen cserélhetők más műveletre:

- Feltétel, azaz hogy mit akarunk összeszámolni

- Növelő függvény – mi mennyit ér a számolásban

A számlálás tétele használható

- Osztók számának megállítására
- Szavak leszámlálására egy szövegben
- ...

Számláló – FOR ciklussal

```
int szamlalo = 0;
for (int i = 0; i < értékek száma; i++)
if (feltétel(következő érték))
szamlalo++;
```

Számláló – WHILE ciklussal

```
int szamlalo = 0;
int i = 0;
while (i < értékek száma
{
if (feltétel(következő érték))
szamlalo++;
i += 1;
}
```

1.4.3. Lineáris keresés tétele

Bizonyos, sorban érkező értékek között egy bizonyos megkeresésére; a „van-e?”, „hányadik?” kérdések megválaszolása használható. A tételben szereplő részek közül az alábbiak tetszőlegesen cserélhetők más műveletre:

- Feltétel – amit keresünk

A lineáris keresés tétele tétele használható

- Annak eldöntése, hogy szerepel-e egy érték a sorozatban
- Prímszám (Osztó) keresésére
- ...

Lineáris keresés

```
boolean van = false;
int hol = 0;
int i = 0;
while ((i < értékek száma) && !van)
{
if (feltétel(következő érték))
{
hol = i;
van = true;
```

```
}
i += 1;
}
```

1.4.4. Maximumkeresés tétele

A „Mennyi a leg…?” kérdés megválaszolásásra használható. (Szélsőértékkeresésre.) Cserélhető a tételben a

- Feltétel, hogy figyelembe vesszük-e az aktuálusan vizsgált értéket
- Reláció – min/max
- Függvény, amivel transzformációt lehet végrehajtani az aktuális értéken.

A maximumkeresés tétele tétele használható

- Minimum-, maximumkeresés
- Zárójelek mélységének számolására
- …

Maximumkeresés

```
int max = függvény(első elem);
int hol = 0;
int i = 0;
while (i < értékek száma)
{
if (max < függvény(következő elem)
{
hol = i;
max = függvény(következő elem);
}
i += 1;
}
```

A tételekben előfordult új jelölések.

- `i++` ⇔ `i = i + 1`
- `i+=10` ⇔ `i = i + 10`
- `!`*`logikai változó`* ⇔ Logikai tagadás. (`!igaz = hamis`);
- **`boolean`** ⇐ Logikai típus
- **`true`** ⇐ logikai igaz
- **`false`** ⇐ logikai hamis

1.4.5. Az elemenkénti feldolgozásról

Az előzőekben ismertetett tételek mindegyike olyan, hogy az adathalmaz, ami rendelkezésre áll, egyszeri végigolvasásával eldönthető a feladatban megfogalmazott kérdésre a válasz. Idetartozik további pár algoritmus például az összefésülés, válogatás, stb.

Egy adathalmaz elemenként feldolgozható, ha egyszerre csak pár elemmel dolgozunk és ha elég egyszer végignézni mindegyiket. (Ezek gyakorlatban a bemenettől lineárisan időben függő problémákat illetve algoritmusokat jelenti.)

Ugyanakkor vannak olyan problémák, amik így nem megoldhatóak, például a sorozat növekvő/csökkenő sorrendbe rendezése, egy adathalmaz mediánjának számítása, vagy annak eldöntése, hogy van-e két egyforma elem az adathalmazban. (De az eldönthető, hogy van-e még egy olyan, mint az első.)

1.5. (El)Gondolkodtató feladat

A feladat egy olyan algoritmus megvalósítása, amely képes megmondani, hogy egy adott pénzmennyiség milyen címletű és hány érmére váltható fel leggazdaságosabban. Efféle algoritmusokat használnak a postán ahhoz, hogy a pénzes postás a lehető legkevesebb, ugyanakkor elégséges mennyiségű címletekkel induljon el reggel.

Gondolkodtató feladat

Bemenet: a felváltandó összeg x

Kimenet: **y** vektor, ami tartalmazza az egyes érmék darabszámát, illetve n az összes darabszámot

Feladat: a program számolja ki, hogy az x hogyan váltható fel a legkevesebb érmére

---

Lehetséges megoldás. Egyszerűsítés kedvéért vegyük forintban. Először vizsgáljuk meg, hogy hány darab 200-as kell, azután a százasoknak megfelelő értéket vonjuk ki a felváltandó összegből és nézzük meg 50-esekre, és így folytassuk tovább ...

Mohó algoritmus

```
int x = 295;
int [] y = {0, 0, 0, 0, 0, 0}
int n = 0;
y[0] = x / 200;
x = x - (y[0] * 200);
y[1] = x / 100;
x = x - (y[0] * 100);
y[2] = x / 50;
x = x - (y[1] * 50);
y[3] = x / 20;
x = x - (y[2] * 20);
y[4] = x / 10;
x = x - (y[3] * 10);
y[5] = x / 5;
```

Ugyanez ciklussal:

Mohó algoritmus – ciklussal

```
int x = 295;
int [] y = {0, 0, 0, 0, 0, 0}
int [] c = {200, 100, 50, 20, 10, 5};
int n = 0;
```

```
for (int i = 0; x < c.length; i = i + 1)
{
y[i] = x / c[i];
x = x - (y[i] * c[i]);
n = n + y[i]
}
```

Ez az algoritmus hatékonynak tűnik, nagyon keveset lehet már javítani rajta. Azonban helyes-e?

Forintban nyilvánvalóan, ennek hamar utána tudunk járni, de mi történik, ha az alábbi érméink vannak: $\mathbf{c} = (25, 20, 5, 1)$? Nézzük meg, mi történik, ha az $x = 42$. (Ehhez nem is kell csodaországba mennünk, hiszen van a földön olyan ország, ahol hasonló érmék vannak.)

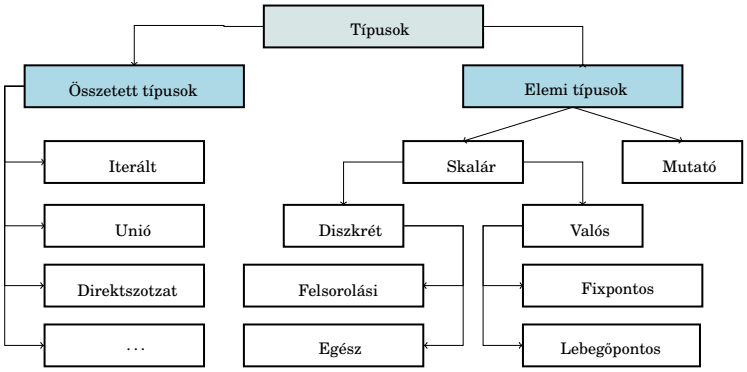
Ránézésre látható, hogy két húszas és két egyes érme a helyes megoldás, de vajon az algoritmusunk is ekként számol? Mohó módon, kezdjük a 25-tel. Kell belőle egy, marad 17. Ezt pedig az érméket felhasználva összesen 5 érmét kell elhasználnunk, tehát a felváltást 6 érmével oldjuk meg. Ez több, mint amit ránézésre megállapítottunk, tehát az algoritmusunk ebben az esetben aligha nevezhető helyesnek.

Kérdés, hogy akkor mi a megfelelő algoritmus erre a problémára. Lehetséges egy olyan megoldás, hogy kipróbáljuk az összes érvényes felváltást, megnézve, hogy melyik a legjobb. Vajon ez a módszer hatékony?

1.6. Típusok

Az előző részben szerepelt egy definíció, miszerint **típus**nak nevezzük egy megadott értékhalmazt és az azokon értelmezett műveletek összességét.

A típusokat az alábbiak szerint lehet osztályozni



1.6. ábra. Típusok osztályozása

A típusok elemi vagy összetett típusok. Összetett esetben meglévő elemi típusokból kiinduló valamilyen konstrukcióról van szó. Például Descartes szorzat esetén két típus lehetséges elemeinek minden lehetséges párbeli kombinációi lesznek az új típus értékei. Összetett típusok továbbá olyan konstrukciók mint a tömb vagy vektor, amely (azonos) elemi típusok sorozatából álló értékek tárolására használhatók.

Elemi típusok esetén megkülönböztetünk:

- Skalárt, amelyek egyetlen érték tárolására képesek. Egy skalár lehet

  - Diszkrét típus, amely valamilyen felsorolható értékeket tárol. Az egész számok egy meghatározott intervallumon szintén felsorolhatóak, így ők is a diszkrét típus csoportjába tartoznak.
  - Valós, valamilyen valós (racionális) szám/érték tárolására alkalmas típusok. Lehetnek fixpontosak, vagy lebegőpontosak, ami a tizedespont rögzítettségére utal. (Lebegőpontos esetben nincs megkötve, hogy a tizedespont előtti jegyek például az egész részt a tizedespont utáni rész pedig a törtrészt jelenti. Lebegőpontos számok a $x * 10^y$ alakúak, ahol az x és y is tetszőleges szám lehet. Természetesen a tetszőlegességnek az ábrázolási pontosság korlátot szab.)
- Mutatót, aminek az értékei a memória lehetséges címei. Mutató esetén az érték által meghatározott memóriaterületen található a számunkra hasznos információ. (Ehhez a memóriát mint rekeszek sorozatát kell elgondolni, ahol minden rekeszbe kerülhet érték és az egyes rekeszeknek egyedi sorszámuk van.)

1.6.1. Adatok ábrázolása – fizikai szint

Fizikai szintet a memória két állapot megjegyzésére képest. Ez merevlemez esetén mágnesezett/nem mágnesezett vagy memória esetén feszültség alatt levő/feszültségmentes állapot. Matematikai értelemben a két állapotnak a 1 és 0 értéket feleltetjük meg. Egy pozícióban egyetlenegy érték tárolható ez lesz a 0 vagy 1. Ezt nevezzük bitnek. 8 biten lehet ábrázolni egy *bájt*ot, kettes számrendszerből tízesre átírva ez 0 és 255 közötti számokat tesz lehetővé, tehát összesen 256 különböző érték. Ezt fogjuk bájnak nevezni. A bájtokat kettő hatványai szerint szokás további egységekbe foglalni, szélesítendő az ábrázolható értékek halmazát.

- Két bájt (16 bit): 0 és 65535 között
- Négy bájt (32 bit): 0 és 4294967295 között, (32-bites rendszerekben ezt *szó*nak (word) is hívják)

A tárolni kívánt érték típusától függ az értékek jelentése, amiket a könnyebbség kedvéért számokként fogunk fel. Például 16 biten (2 bájton) tárolni lehet:

- Előjel nélküli egész számokat ($0 \ldots 65535$)
- Előjeles egész számokat ($-32768 \ldots 0 \ldots 32767$). (Első bit előjelbit)
- Karaktereket (úgynevezett Unicode táblázat alapján). Minden egyes értékhez egy karaktergrafika rendelhető, amely ebben az esetben a világ összes írásformájára elegendő számú helyet biztosít. (Korábban illetve a kompatibilitás megőrzése érdekében 8 bájton tároltak egy-egy karaktert, ami legföljebb 256 különböző karakternek volt elég. Könnyen utánajárhatunk annak, hogy ez a teljes latin abc és a hozzá kapcsolódó számoknak és írásjeleknek sem elég, ha a lehetséges nemzeti ékezetes karaktereket is szeretnénk tárolni.)
- ...

Milyen adatokat lehet tárolni a memóriában:

- Logikai értékeket – ahol a logikai változó értéke igaz vagy hamis lehet.
- Racionális számokat, meghatározott tizedes pontossággal
- Karaktersorozatokat (szövegeket)
- Memóriarekeszek címét
- Programutasításokat (az egyes bájtoknak utasítások felelnek meg)
- Dátumot, időt (például eltelt másodpercekben mérve)

- Az előzőek sorozatát vektorok formájában
- ...

A fentiekből látható, hogy a 64-es értéknek megfelelő bináris szám 01000000 sorozat függően attól, hogy miként értelmezzük több mindent jelenthet. Lehet a „@” karakter. Lehet a 64, mint szám. Lehet hogy eltelt időt jelent másodpercben.

1.6.2. Memória szerveződése

Pár mondat erejéig a memória logikai szerveződéséről lesz szó. A memória rekeszekre van osztva, ahol a rekeszek hossza rendszerenként más, manapság például 32/64 bit szokásosan egy asztali PC-ben. Egyetlen rekeszben mindig legfeljebb egy érték van, akkor is, ha a tárolandó érték kevesebb helyen is elférne. (Tehát a logikai változó tárolása is ugyanúgy egyetlen teljes rekeszt elfoglal.) A programozó (így a gépi utasításokra fordított program) tudja, hogy melyik rekeszben milyen típusú érték van, hogy kell értelmezni, beleértve a programkódra vonatkozó információkat. (A rekeszeknek tudunk nevet adni, tulajdonképpen ezek a változók. Ez közvetve feloldódik a rekeszek címére, a program futó kódjában ténylegesen a cím kerül felhasználásra.) Változókon keresztül a rekeszeket lehet együttesen is kezelni (összefogni). Például tömbök, karaktersorozatok ... (Amikor egy rekeszbe egy másik rekesz címét tesszük és a hivatkozottat elérjük azt a referencia feloldásának nevezzük.)

1.6.3. A valós számok a memóriában

Egyrészt fontos megjegyezni, hogy a valós számok halmazából, csakis a racionális számokat tudjuk adott esetben pontosan tárolni. (Természetesen a tárolás pontosság itt is korlát.) Az irracionális számok (végtelen tizedestört alakúak) pontos tárolására nincs mód. A racionális számok az $X * 2^Y$ alakban ábrázoljuk. Ahol az X és Y értéke kerül tényleges tárolásra. A hosszak például egy 32 bites tárolás esetén 23 bit az X és 8 bit az Y, illetve még egy bit az előjelbit. Egy IEEE szabvány esetén a lebegőpontos számábrázolás az alábbi alakot ölti:

- Az X minden esetben $1.\overbrace{xxxxxxxxxxxxxxxxxxxxxxx}^{23db}$ alakú, ahol az x egy bináris érték
- Az exponens (Y), amely a maradék biteken kerül kódolásra adja meg a kettedes pont helyét. (A kettedes pont a tizedes pont mintájára képzelendő el, nem feledjük el ugyanis, hogy itt nem tízes, hanem kettes számrendszerben kell gondolkodni.)
- A vezető egyest valójában nem tároljuk, mivel mindig egy, az előző felírásban a kettedes pont előtti számjegy.

A tárolás következményei

Az előbb leírt tárolási módszerekből könnyen láthatjuk az alábbi gondolatok fontosságát és érvényességét:

- Nagyon fontos tudni az értékek típusát, mert legbelül a fizikai szinten minden egyforma.
- Nem végtelen a precizitás számok esetén, tehát matematikai problémáknál ezt föltétlen figyelembe kell venni.

- Nem végtelen az ábrázolható számok intervalluma, azaz ha egy bájton tárolunk és vesszük azt a kifejezést, hogy $255+1$, akkor is kapunk eredményt, mégpedig azt hogy $255 + 1 = 0$. Ezt jelenséget túlcsordulásnak hívjuk, van egy párja is alulcsordulás néven, amihez a $0 - 1 = 255$ példa tartozik.
- Racionális számoknál ha két eltérő nagyságrendű számot adunk össze, például 23 kettedes jegynél nagyobb a nagyságrendbeli különbség, akkor $A \neq 0$: $A + B = B$ előfordulhat, mivel az összeadás után a hasznos jegyekből az A-hoz tartozó értékek eltárolása lehetetlen adott pontosság mellett. (Ilyen esetben előfordulhat, hogy a precizitás növelése megoldja a problémát. (Például 32 bites lebegőpontos helyett 64 bites lebegőpontos számok használata.)
- Nem mindig igaz, pontosságvesztés miatt, hogy $(x/y) * y = x$, tehát valós számoknál **ne** használjunk egyenlőségvizsgálatot.

1.7. Objektumorientált programozás

A típusok megismeréséhez Java nyelven szükséges pár fogalom az objektumorientált programozási szemléletből, mivel a Java nyelv az objektumorientált szemléletet követi.

1.7.1. Modellezés

A programozáshoz a valós világot modellezzük, amely során olyan absztrakt fogalmakat vezetünk be a programozás során amelyek megfeleltethetők a világban található tárgyaknak, entitásoknak, fogalmaknak. A modellezéshez az alábbi alapelveket használjuk fel

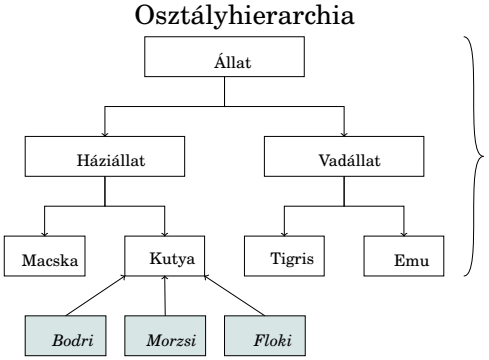
- Absztrakció – az a szemléletmód, amelynek segítségével a valós világot leegyszerűsítjük, úgy, hogy csak a lényegre, a cél elérése érdekében feltétlenül szükséges részekre összpontosítunk. Elvonatkoztatunk a számunkra pillanatnyilag nem fontos, közömbös információktól és kiemeljük az elengedhetetlen fontosságú részleteket.
- Megkülönböztetés – az objektumok a modellezendő valós világ egy-egy önálló egységét jelölik, a számunkra lényeges tulajdonságaik, viselkedési módjuk alapján megkülönböztetjük.
- Osztályozás – Az objektumokat kategóriákba, osztályokba soroljuk, oly módon, hogy a hasonló tulajdonságokkal rendelkező objektumok egy osztályba, a különböző vagy eltérő tulajdonságokkal rendelkező objektumok pedig külön osztályokba kerülnek. Az objektum-osztályok hordozzák a hozzájuk tartozó objektumok jellemzőit, objektumok mintáinak tekinthetők.
- Általánosítás, specializálás – Az objektumok között állandóan hasonlóságokat vagy különbségeket keresünk, hogy ezáltal bővebb vagy szűkebb kategóriákba, osztályokba soroljuk őket.

Objektum példa
Sok embernek van kutyája különböző névvel és jellemző tulajdonságokkal – objektumok (példányok)
A kutyák, mint egy állatfaj egyedei sok mindenben hasonlítanak is – például mindegyik tud ugatni

---

1.7.2. Osztályhierarchia

Természetesen, ha szükség van további állatok reprezentálására a program során, akkor további osztályokat, objektummintákat vezethetünk be. Legyenek ezek például a macskák a tigrisek és az emuk. A macskákban és a kutyákban lehetnek olyan közös tulajdonságok, amelyeket egy felsőbb absztrakciós szinten leírhatunk, a háziállatok szintjén. Hasonló gondolat mentén tudjuk kialakítani az alábbi struktúrát, amelyben a kapcsos zárójellel jelölt rész az osztályhierarchia, illetve néhány Kutya objektumpéldán is jelölésre került.



1.7. ábra. Osztályhierarchia példa

Az objektumorientált nyelvek eszközeivel a fenti ábra teljes egészében megvalósítható, olyan módon, hogy a Macska osztály minden egyes tulajdonságát örökölni képes a Háziállat osztálynak illetve felfelé a hierarchiában.

1.7.3. Objektumok és állapotaik

Az objektumorientált program egymással kommunikáló objektumok összessége, ahol minden objektumnak megvan a feladata. **Az objektum** információt tárol, kérésre feladatokat hajt végre; belső állapota van, üzeneten keresztül lehet megszólítani és megváltoztatni; valamint felelős; feladatainak korrekt elvégzéséért.

Objektum = adattagok + műveletek (függvények)

Az objektumoknak Mindig van egy állapota, amit a mezők (objektumhoz tartozó változók) pillanatnyi értékei írnak le. Az objektum műveleteket hajt végre, melyek hatására állapota megváltozhat. Két objektumnak akkor lesz ugyanaz az állapota, ha az adattagok értékei megegyeznek.

1.7.4. Osztály és példány

Az **osztály** (class) olyan *objektumminta* vagy **típus**, mely alapján példányokat (objektumokat) hozhatunk létre. A **példány** (instance) egy osztályminta alapján létrehozott objektum. Minden objektum születésétől kezdve egy osztályhoz tartozik, életciklusa van megszületik, él, meghal. Létrehozásához inicializálni kell – speciális művelet, függvény, a neve **konstruktor**, ami a változóknak kezdőértéket ad, valamint az objektum működéséhez szükséges tevékenységek végrehajtja.

Példányváltozó: objektumpéldányonként helyet foglaló változó – minden példánynak van egy saját.

Osztályváltozó: osztályonként helyet foglaló változó – az osztályhoz tartozó változó.

Példányfüggvény (-metódus): objektumpéldányokon dolgozó metódus, művelet, amely a meghatározott példány változóit éri el, illetve azokon végezhet műveleteket.

Osztályfüggvény (-metódus): osztályokon dolgozó metódus, művelet, amely az osztályváltozókat éri el.
Láthatóság: Lehetőség van arra, hogy bizonyos függvényeket (műveleteket), változókat az osztályhasználó számára láthatatlanná tegyünk. Ez az *(információ elrejtésének alapelvéhez)* tartozik, ahol arról van szó, hogy az objektumot használó számára csak a számára szükséges elemei legyenek elérhetőek az objektumból. Azaz ne tudja úgy megváltoztatni az állapotát, hogy azt ne műveleten keresztül tegye, vagy ne tudjon előállítani nem konzisztens (érvényes) állapotot az objektumban.

1.7.5. Öröklődés

Az állat osztálynak vannak bizonyos tulajdonságai (mezői) és függvényei. Amennyiben elkészítjük a *háziállat* osztályt, nyilvánvaló, hogy sok olyan tulajdonsága lesz, mint ami az *állat*nak. Kézenfekvő az ötlet, hogy ezt a programozás során a OOP-t támogató nyelv is kezelje. Erre lehetőség az OOP nyelvekben, hogy a *háziállat* osztályt az *állat* osztály **leszármazottja**ként létrehozni, ekkor az összes mezőt és függvényt **örökli** a háziállat, ami az állatban megvolt. Természetesen további függvényeket és mezőket vehetünk a *háziállat*ba.

(Az öröklődés, dinamikus kötés és polimorfizmus (statikus és dinamikus típus) nagyon messzire elvinne minket, így elméletben többről nem esik szó. Fontos megjegyezni azonban, hogy a fentebbiek alapelvek, ennél sokkal színesebb paletta áll rendelkezésre)

2. fejezet

Java és Eclipse

2.1. Java típusok

A Java egy objektumorientált nyelv, aminek az a következménye, hogy minden beépített típus a primitíveket kivéve objektum.

2.1.1. Java primitívek

A primitív típusok eddigi fogalmainkkal jól leírhatóak, minden egyest tárol primitív típusú értékhez egy dedikált rész tartozik a memóriában. Java nyelven az alábbi primitív típusok érhetőek el:

- **`boolean`**: Logikai típus, lehetséges értéke **`true`** – igaz, vagy **`false`** – hamis.
- **`byte`**: 8-bites előjeles egész.
- **`short`**: 16-bites előjeles egész.
- **`int`**: 32-bites előjeles egész.
- **`long`**: 64-bites előjeles egész.
- **`float`**: 32-bites lebegőpontos racionális szám.
- **`double`**: 64-bites lebegőpontos racionális szám.
- **`char`**: 16-bites Unicode karakter.

2.1.2. Objektum típusok Javaban

Java esetén az objektum típusú változóknak az értéke egy referencia, amely az objektumpéldány helyét (címét) mondja meg a memóriában. Amikor a változót használjuk automatikusan megtörténik a példányra való referálás, tehát a címet közvetlenül nem tudjuk elérni. Amikor egy értéket adunk egy objektumpéldányt referáló változónak, akkor vagy egy új példány létrehozásával vagy egy meglévő példányra referáló változóval tudjuk megtenni. (Ez utóbbi esetben ugyanarra a példányra fog két változó hivatkozni.) (Primitív típusok esetén a változó értéke a ténylegesen tárolt érték.)

2.1.3. Csomagoló osztályok

Az objektumelvű szemlélet miatt a primitív típusoknak Javaban vannak objektum párjaik, amelyek a következők:

- `Boolean`: **boolean**
- `Byte`: **byte**

- `Short`: **short**
- `Integer`: **int**
- `Long`: **long**
- `Float`: **float**
- `Double`: **double**
- `Character`: **char**

Ezeket csomagoló osztályoknak hívjuk, gyakorlatilag a primitív társukkal felcserélhetőek a legtöbb esetben. Fontos, hogy nem megváltoztatható az értékük, ami azt jelenti, hogy az objektumpéldány állapota a létrehozás után állandó. Amennyiben egy csomagoló osztály példányának értékét megváltoztatjuk egy kifejezésben, akkor új példány jön létre a memóriában és a változó az új példányra fog hivatkozni.

2.1.4. Karakterláncok

Hasznos típus a karakterlánc – String (objektum), amelybe szövegeket lehet eltárolni. Hasonlóan a csomagoló típusokhoz a karakterláncok sem változtathatóak meg. Amikor új értéket adunk, akkor egy új példány jön létre, amelyre a korábbi változó fog hivatkozni.

Deklaráció

```
String s;
```

Értéket adni idézőjelek között megadott szöveggel lehet:

Értékadás

```
String s = "Szervusz világ";
```

Ha idézőjelet szeretnénk belevinni a szövegbe akkor:

Idézőjelek egy karakterláncban

```
String s = "Szervusz \"szép\" világ";
```

Karakterek számának lekérdezése egy `String` esetén

Karakterek száma

```
String s = "Szervusz világ";
int meret = s.length();
```

2.1.5. Tömbök

A tömb ahhoz hasonlít, amit matematikában vektornak hívunk. A memóriában folytonosan több ugyanolyan típusú terület foglalódik le deklarációkor, amelyet indexelten lehet elérni. [1]

Java nyelven egy egészekből álló tömb deklarációja a következőképpen történik:

Deklaráció

[1] A folytonos memóriaterületen való elhelyezkedés fontos, ugyanis hiába van sok szabad memória, azonban, ha az nem folytonos nem tudunk maximális méretű tömböt lefoglalni. Megjegyezendő, hogy további korlátok is vannak a dinamikusan kezelhető memória nagyságára vonatkozóan.

```
tombtipusa [] tombneve;
```

Egy létrehozott tömb hossza nem változtatható meg a későbbiekben, viszont lehetőség van újabb, nagyobb deklarációjára. Egy tömbnek értéket adni többféleképpen lehet:

Értékadás – Felsorolással

```
int [] tombneve;
tombneve = {1,2,3,4,5};
```

Ugyanakkor létrehozható egy tömb kezdőértékek nélkül is, csak a méret megadásával:

Értékadás – Üres létrehozása

```
int [] tombneve;
tombneve = new int[10];
```

Ebben az esetben egy új objektumpéldány kerül létrehozásra, amelynek típusa egy egészekből álló tömb típus.

Illetve a tömb értékadásánál lehetőség van egy másik tömbbel egyenlővé tenni

Értékadás – Másik tömbbel

```
int [] masiktomb = {0, 1};
int [] egyiktomb = masiktomb;
```

Fontos, hogy ekkor a memóriában egyetlen tömb lesz csak, ugyanakkor kétféle változónévvel lehet elérni, két változó referál rá.

A tömbök tartalmát indexeléssel érjük el, a számozás 0-val kezdődik.

Például

```
int [] egyiktomb = new int[10];
```

Az előző tömb esetén $0 \ldots 9$ indexek érvényesek, a többi kiindexel a tömbből.

Egy tömb méretének megismerését a következő példa mutatja be:

Tömbméret

```
int tombmerete = egyiktomb.length;
```

A egy tömb deklarációja során implicit módot egy új típust hozunk létre. Ezzel a típuskonstrukcióval lehetőség van egy további tömb típusát meghatározni, amelyet a következő példa mutat be:

Deklaráció

```
tipus [] [] matrix;
```

Ebben a példában ezáltal egy kétdimenziós tömböt hoztunk létre. Tovább folytatva többdimenziós tömböket is létre lehet hozni, a korlát a memória mérete. (Kétdimenziós vektorok a mátrixok).

2.1.6. Műveletek

A következő műveletek értelmezettek egész típusokon, ahol a sorrend a precedencia sorrendjében került leírásra:

- **Növelés, csökkentés**: ++, −−
- **Multiplikatív**: ⋆, /, % (Szorzás, Maradékos osztás, és maradékos osztás maradéka)
- **Additív**: +, −
- **Bitenkénti eltolás**: <<, >> (Balra, jobbra) A bitenkénti eltolás esetén gyakorlatilag kettővel való szorzásnak (balra) illetve kettővel való osztásnak felel meg (jobbra).
- **Bitenkénti műveletek**: ∼, &, |, ^ (Negálás, és, vagy, kizáró vagy)
- **Relációs**: ==, !=, <, <=, >, >=
- **Unáris**: +, − (előjelek)
- **Értékadás**: A változónak új értéket ad = (Kombinálható más művelettel: +=)

Racionális típusok esetén a műveletek:

- **Növelés, csökkentés**: ++, −−
- **Multiplikatív**: ⋆, /, % (Szorzás, Osztás, és maradékos osztás maradéka. Figyelem itt az osztás nem maradékos.)
- **Additív**: +, −
- **Relációs**: ==, !=, <, <=, >, >=
- **Unáris**: +, − (előjelek)
- **Értékadás**: A változónak új értéket ad. =

A következő művelek értelmezettek logikai típusokon:

- **Tagadás**: !
- **Relációs**: ==, !=
- **Logikai és, vagy**: &&, ||
- **Értékadás**: A változónak új értéket ad. = (Az érték `true` vagy `false`)

Karakterláncok esetén pedig az alábbi érvényes műveleteink léteznek.

- **Értékadás**: A változónak új értéket ad. =
- **Összefűzés**: + Több különböző karakterláncot fűz össze

A műveletek esetén felmerülő lehetséges problémák a típusra vezethetők vissza bizonyos esetekben. Mivel a műveletek mindig tartoznak egy típushoz is, ezért a típus dönti el egy kifejezésben, hogy milyen operátor kerül alkalmazásra. Például, ha a $10/3$ osztást szeretnénk elvégezni, akkor azt várjuk el, hogy az eredmény $3.333\ldots$ legyen. Ezzel szemben Javaban a `10 / 3 = 3`, mivel a 10 egész szám, ezért az egészhez tartozó / operátort veszi, ami a maradékos osztás. Azonban `10D / 3 = 3.3333 ...`, ahol a `D` jelöli, hogy ez itt egy `double` típusú, tehát nem egész. (Helyette `10.0`-t is írhatunk.)

2.2. Java osztályok

Javaban osztály létrehozására az alábbi szintaxis szerint lehet létrehozni. Ez meghatározza az osztálynak a lehetséges változóit és műveleteit.

Osztály létrehozása

```
public class osztálynév extends szülő [és még más]
{
public int mezonev;
private String mezonev;
...
public osztályneve (paraméterek)
{ // Konstruktor }

public int fuggvenyneve (paraméterek)
{ ...}
...
}
```

A mezők és és függvények előtti lehetséges kulcsszavakból néhány:

- **public**: mindenki számára elérhető a program egészében.
- *nincs kulcsszó*: adott osztályban, leszármazottjaiban (öröklődés) és a csomagban érhető el.
- **protected**: csak az adott osztályban és leszármazottjaiban, a csomag többi osztályában már nem érhető el.
- **private**: csak az adott osztályban érhető el, a leszármazottakban már nem.
- **static**-kal jelöljük az osztálymezőt illetve függvényt.
- Ha egy mező `final`, akkor nem módosítható.
- Ha egy osztály **final**, akkor nem örökölhető belőle tovább.

(További kulcsszavak az **abstract**, **synhronized**, **volatile**, **native** . . . , amelyek számunkra most nem relevánsak.)

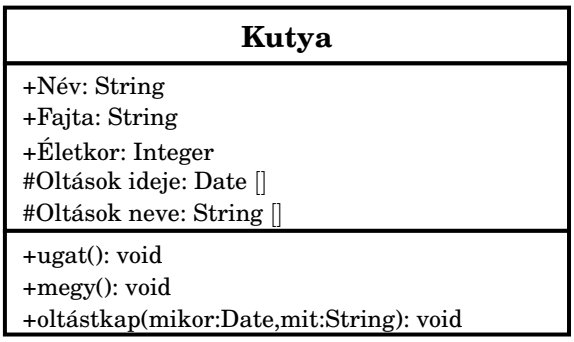
Az osztályok elemeinek láthatósági szabályozása az eszköz, amivel a külvilág számára el tudjuk rejteni egy objektum belső szerkezetét, állapotát. Az eddigi példánk folytatása, a Kutya osztály Java nyelven:

Kutya osztály

```
public class Kutya extends Allat
{
public String nev;
public String fajta;
public Integer eletkor;
private Date [] oltasok_ideje;
private String [] oltasok_neve;
public Kutya ()
{
oltasok_ideje = new Date[10];
oltasok_neve = new String[10];
}
public void ugat()
{
}
public void megy()
{
}
```

```
public void oltastkap(String s, Date mikor)
{
}
}
```

Látható, hogy az oltásokkal kapcsolatos információkat nem lehet közvetlenül módosítani, csakis egy függvényen keresztül. Ez azért jó, mert így nem lehet olyan állapotot előidézni, ami szerint a Kutya osztály egy példányának az oltások ideje és az oltások megnevezése tömb eltérő elemszámú (ami nyilvánvalóan nem érvényes állapot).



2.1. ábra. A Kutya osztály UML diagramja

Az előző *kutya* osztály, mint típus az alábbiak szerint deklarálható

Deklaráció

```
Kutya bodri;
```

Ez még csak deklaráció, a tényleges példány létrehozása a `new` kulcsszóval történik.

Példányosítás

```
bodri = new Kutya();
```

A `new` kulcsszó hatására a memóriában létrejön egy új `Kutya` objektum, valamint lefut annak a konstruktora. A korábban említettek szerint, amikor az `Object` osztály bármely leszármazottját (legyen az tömb, `String`, `Double`, `Kutya` ...) deklaráljuk, akkor a változó, ami lefoglalásra kerül a memóriában egy **referenciát** (memóriacímet) tartalmaz értékként, nem az objektumot magát. Ez referencia alapértelmezésben `null`, azaz nincs hozzátartozó objektumpéldány. (Tehát a változó képes egy adott típusú objektumra hivatkozni, de éppen nem hivatkozik egyre sem.) Ahhoz hogy hivatkozzon, létre kell hozni egy új példányt, vagy egy meglévő hivatkozást kell átadni értékként (egy meglévő példányt):

Példányosítás

```
Kutya morzsi = new Kutya();
Kutya rex = morzsi;
```

A második miatt egyetlen `Kutya` példány van a memóriában, csak két névvel is hivatkozhatunk rá: `morzsi` és `rex`. **Egy objektumváltozó értéke Java-ban egy referencia a memóriában!** Az olyan objektumpéldányokra amelyekre nincs olyan változó ami a referenciát tartalmazza úgy kell tekintenünk, mint ami nincs is. (Ez Java esetén au-

tomatikusan felszabadításra kerülő memóriaterületet jelent, más nyelveken ez elveszett memóriaterület.)

Egy objektumpéldány mezőit és tagfüggvényeit a példányon keresztül lehet meghívni. (Természetesen ez csak a látható elemekre vonatkozik, ahol a láthatóságot a fentebb leírt kulcsszavak határozzák meg.)

Tagfüggvények, mezők

```
bodri.ugat();
String kutyaneve = bodri.nev;
```

Egy osztály osztálymezőit és függvényeit az osztályon keresztül javasolt elérni. (Lehetséges egy példányon keresztül is, de az ellentmond a példányfüggetlenségnek.)

Osztálymező

```
Kutya.ALAPÉRTELMEZETTUGATÁSIHANGERŐ
```

2.3. Függvények és metódusok

Korábban már több helyen érintőlegesen szó volt róla, ezért most vegyük részleteiben a függvény fogalmát.

A függvények olyan részprogramok, műveletek, amelyeknek a programokhoz hasonlóan vannak bemenő paramétereik, valamilyen műveletet végeznek el és egy eredménnyel térnek vissza. (Ez nagyon hasonlít a matematikai értelemben vett függvény fogalomhoz, ahol is a bemenetei paraméterekhez egy eredményt rendelünk. Ugyanakkor bizonyos tekintetben nagyon különbözik attól, például egy objektumfüggvény az objektum állapotának megváltoztatására is alkalmas.)

Minden Java programban van egy függvény, a `main` függvény, ami a program elindulásakor kezd futni. Ha a `main` függvény futása befejeződik, akkor a program is befejeződik. A `main` további függvényeket hív(hat) meg, illetve a függvények is továbbiakat hívhatnak meg.

A következő main függvény egy i szám négyzetét számolja ki, amely függvényben a *KIIR* egy absztrakt parancs, a képernyőre való kiírást jelenti.

Példa `main` függvényre

```
public static void main(String [] arguments)
{
int i = 10;
int negyzet = i*i;
KIIR(negyzet);
}
```

Vegyük az alábbi példaprogramot, amely egy függvény egy tetszőleges háromszög magasságát számolja ki.

Területszámítás

```
public double terulet(double oldal, double magassag)
{
return (oldal * magassag) / 2;
```

```
}

public static void main(String [] arguments)
{
double side = 10;
double height = 8;
double eredmeny = terulet(side, height);
}
```

Ebben a kódrészletben található egy területszámító függvény, aminek a neve `terulet`. A függvénydeklarációnak az alábbi szerkezete van:

Függvényszignatúra

visszatérésitipus `fuggvenynev (`*parameterdeklarációk*`)`

A példa esetén a függvény deklarált paraméterei a **double** `oldal`, **double** `magassag`. Ezeket hívjuk *formális paraméterek*nek, Attól formális, hogy a függvényen belül bármilyen hívás esetén ezekkel a változókkal (változókban) érjük el a függvényhívás *aktuális paramétere*einek értékét. Az aktuális paraméterek ebben a példában a `side`, `height`, amiknek az értékei rendre 10 és 8. A függvényhívás a **double** `eredmeny = terulet(side, height);` sorban történik, ahol is a paraméterek helyén lévő kifejezések kiértékelődnek. Ezután átkerül a futtatás a függvényhez, amely végrehajtja az utasításokat és a **return** kulcsszó mögötti értéket visszaadja eredményként a hívónak, aminek hatására ebben az esetben a **double** `eredmeny` változó értéke a kiszámított terület lesz.

A függvény visszatérés típusát hívják a függvény típusának is. A függvény neve lehet bármi, kivéve a nyelv fenntartott szavait. A szignatúra alapján használjuk a függvény, amiben a paraméterek vesszővel elválasztott változódeklarációk. A szignatúrát (lenyomatot) követi a függvény törzse, ami

- használhatja a bemenő paramétereket, új változókat.
- tartalmaz (legalább) egy **return**, amit a visszatérési típusnak megfelelő kifejezés követ – ez lesz a függvény visszatérési értéke.
- egy **return** utasítással befejeződik, még akkor is, ha van mögötte további utasítás.
- (Annak a függvénynek, aminek nincs visszatérési típusa, **void** a típusa. A **void** a „semmilyen típus”, nem hozható létre belőle példány.)

2.3.1. Paraméterek

A *formális paraméterek* a függvényszignatúrában vannak deklarálva, új változók! Ezek a függvény hívásakor felveszik az *aktuális paraméterek* értékét, Java nyelv esetén érték szerint. Tehát a formális és aktuális paraméter más-más terület a memóriában. A változónév azonban lehet ugyanaz! A függvények aktuális paraméterei lehetnek további függvényhívások!

A visszatérési érték, a függvény befejeztekor, annak az értékadó utasításnak a jobb oldala lesz, amiben a függvény neve szerepelt

2.3.2. Az érték szerinti paraméter átadás és következményei

Java nyelven, amikor függvényt hívunk, az aktuális paraméter értéke átmásolódik a formális paramétert jelentő változóba. (Létrejön(nek) a függvényszignatúra szerinti új vál-

tozó(k), és az értékük az lesz, ami a függvényhíváskor az aktuális paraméter volt.) Tehát egy külön változó, aminek pont ugyanaz az értéke.

Az objektumoknak azonban az értéke a referencia, tehát a referencia másolódik át, így az eredeti objektum egy példányban marad meg, csak kétféle névvel lehet hivatkozni rá. (Aminek az a következménye, hogy ha a függvény megváltoztatja az objektum állapotát, akkor az „eredeti" objektumot változtatja meg.

A csomagoló típusok ugyan objektumok, tehát egyetlen példány létezik, mivel változtatáskor új jön létre, a tényleges hatás ugyanaz, mint a primitívek esetén. (Azaz, ha a függvény megváltoztatja az értékét, akkor az mégsem lesz hatással az eredeti objektumra nézve.)

Paraméterek példa

```
public void fuggveny(Double d, int i, Kutya k)
{
d = 5.0;
i = 10;
k.oltastkap("Veszettség", new Date())
}
public static void main()
{
Double dd = 2.0; // Double d = new Double(2.0)
int ii = 10;
Kutya kk = new Kutya();
fuggveny(dd, ii, kk);
}
```

A `main` függvényben létrejön egy `Double` referencia ami egy példányra mutat, aminek `2.0` az értéke. Létrejön egy **`int`**, aminek `10` az értéke, valamint egy új `Kutya`. Ezek mindegyike átadódik paraméterként a függvénybe, a következő módon. A `Double` referencia átmásolódik, a példány nem, egyetlen van belőle továbbra is. Az `ii` értéke egy új `int`-be másolódik át. A `kk` referencia is átmásolódik. Megváltoztatjuk a `d` értékét, ami azt jelenti, hogy létrejön egy új `Double` példány és a `d` erre fog referálni. (Ez a megváltoztathatatlanság miatt van. A `dd` továbbra is `2.0`.) Az `i` eleve egy másik változó (példány) a memóriában, annak az értéke megváltozik, semmi hatása sincs az `ii`-re. A `k` objektum állapotát egy függvényén keresztül változtatjuk, ez pedig az egyetlen létező példányt módosítja, így a `kk`-t is.

2.4. Változók láthatósága

Egy változó láthatósági körének nevezzük azt a részt a programban, ahol az adott változó és általa képviselt memóriaterület elérhető, módosítható. Egy változó hatáskörének nevezzük azt a részt programban, ahol a változó deklarációja érvényben van. Egy függvényen belül deklarált változó csak a függvényen belül látható és arra terjed ki a hatásköre is. Egy utasításblokkon belül deklarált változó hasonlóan viselkedik a függvény esetében leírtakhoz. Az blokkon, függvényen belüli változókat *lokális változónak* hívjuk.

(Léteznek globális változók is, amelyeket több függvényből el lehet érni, használatuk azonban nem javasolt. (Itt nem az osztályokról van szó, ez programnyelvtől független fogalom.)

Ha egy blokkon belül további blokkok vannak, akkor ott is deklarálhatunk új változókat *azonos* névvel is (a külső blokkokban azonos nevű változók szerepelnek). Ekkor a belső blokkban található deklaráció elfedi a külső változót, tehát az nem látható. Ugyanakkor mindkettő érvényes deklarációja van, hatáskörrel rendelkezik.

Példa a hatáskörre

```
int i = 10;
int j = 100;
{
int i = 50;
}
i++;
j++;
```

Az első `i`-nek a teljes példára kiterjed a határköre, azonban a belső részben nem látható. A második `i` csak a belső részben látható és a hatásköre is ugyanaz. Nem engedi látni a külső.

2.5. További eszközök

2.5.1. Foreach ciklus

Az alábbi **for** ciklus egy tömb elemein lépked végig, a tömb elemeinek összegét kiszámolandó (összegzés tétele szerint):

For ciklus – összegzés példa

```
int [] tomb = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int osszeg = 0;
for (int i = 0; i<tomb.length; i++)
osszeg += tomb[i];
```

Több programnyelv esetén lehetőség van arra, hogy ezt tömörebben le lehessen írni, az úgy nevezett foreach ciklusok segítségével.

Foreach ciklus – összegzés példa

```
int [] tomb = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int osszeg = 0;
for (int aktualis : tomb)
osszeg += aktualis;
```

Foreach ciklus

```
for (elemek típusa változó : aminvégigkelllépni)
ciklusmag;
```

A ciklus elején deklarált változó az iterációk során felveszi a bejárni kívánt objektum összes értékét. Nem használható azonban olyan esetekben, ahol meg kell változtatni az egyes értékeket. (Például megszorozni a tömb egyes elemeit.)

2.5.2. Típuskonverzió

A különböző típusok között lehetséges konvertálni. A konvertálás történhet implicit, vagy explicit módon.

Implicit típuskonverzió. Amikor egy operátor kiértékelésénél különböző típusok vannak, ugyanakkor egyik szűkítése a másiknak, a nyelv automatikus bővítő konverziót hajt végre. (Tehát int típusból automatikusan **long**-ra konvertál, amennyiben erre szükség van.) Ugyanígy a csomagoló és primitív párok között is automatikus konverzió történik.

```
int i = 10;
long l = 100;
double d = 200;
dd = 6.6;
(i < l)
(d > l)
(dd = d)
(d = i)
```

Az összehasonlításhoz szükséges, hogy az `i` és `l` változók azonos típusúak legyenek. Hasonlóan igaz ez a többi példára is. Mindig a bővebb típus felé történik a konverzió, tehát a kisebb bitszámon tároltból nagyobb bitszámú, az egészből lebegőpontos lesz.

Explicit típuskonverzió. Lehetőségünk van „kézzel” kényszeríteni a típuskonverziót – ezt nevezzük explicit konverziónak.

Explicit típuskonverzió

```
(újtípus) kifejezés;
```

Az új típus nevét kell zárójelek között megadni az átkonvertálandó kifejezés elé. Fontos, hogy a zárójelezések segítségével meg lehet változtatni a konvertálandó részét a kifejezésnek. Gyakran használatos eszköz az osztás pontosságának beállítására:

Osztás példa

```
int i = 10;
double d1 = i / 3; // = 3.0;
double d2 = (double)i / 3; // = 3.33333;
```

További, függvényekkel támogatott konverzió Java nyelven. A beépített csomagoló típusok segítségével lehetőség van karakterlánc és számok közötti konverzióra is. Ezek azonban már függvényhívások Javaban.

Egész karakterláncban

```
String s = "20";
int i = Integer.parseInt(s);
int i = Integer.parseInt(s, 16); // Hexadecimális
```

A `patseInt` második paramétere egy szám amivel megadhatjuk hogy a felismerni kívánt karaktersorozatot milyen számrendszer szerint értelmezze a függvény. Ha nem adjuk meg azt a paramétert, akkor automatikusan dönti el az következőket figyelembe véve. Egy hexadecimális szám a „`0x`” karakterekkel kezdődik. Ez nyolcas számrendszerbeli szám esetén mindig van egy nulla a szám elején. (Vezető nulla.)

String-é alakítani is lehet.

Szám karakterlánccá

```
String s1 = Double.toString(100);
String s2 = Integer.toHexString(10);
```

2.5.3. Felsorolási típus

Olyan típus definiálható, ahol a lehetséges értékeket saját magunk adjuk meg.

Enum

```
enum Nap { HÉTFŐ, KEDD, SZERDA, CSÜTÖRTÖK, PÉNTEK, SZOMBAT, VASÁRNAP
}
Nap n = Nap.SZERDA;
```

Ekkor létezni fog egy Nap nevű típus, ahol pontosan meg van határozva az, hogy milyen értékeket vehet fel, sőt még egy explicit sorrendiség is létezik az egyes értékek között. A felsorolási típus használható a többszörös elágazásban, mint eset, valamint a foreach ciklusokban is a bejárandó kifejezés helyén.

További információ: http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html

2.5.4. IO műveletek Javaban

A beolvasásra egy egyszerűsített módszer kerül ismertetésre. A Java úgynevezett folyamokat (Stream) kezel a be- és kimeneti műveletekből. Ezen eszközök rendkívül széles skálája áll rendelkezésre, ami segítségével a tényleges IO eszköztől függetlenül lehet kezelni a beolvasást és kiírást. (Például egy hálózati kapcsolat hasonlóan olvasható és írható, mint egy USB port, vagy egy fájl.)

2.5.5. Beolvasás

A `java.util.Scanner` osztállyal lehetséges a konzolról, valamint fájlokból sorokat, illetve meghatározott típusú elemeket beolvasni. A `Scanner` osztály példányosítása:

Scanner

```
Scanner sc = new Scanner(bemenet)
```

A bemenet lehet:

- Fájl (`File`).
- IO Csatorna (Konzol, `InputStream`, `FileReader`).
- `String`.

Néhány példa:

Konzolról beolvasás

```
Scanner sc = new Scanner(System.in);
```

Fájlból beolvasás

```
Scanner sc = new Scanner(new File(fájlneve));
```

A `Scanner` alkalmas speciális elválasztójelekkel írt fájlok olvasására, reguláris kifejezések értelmezésére is.

Következő sor olvasása

```
String s = sc.nextLine();
```

Következő `int` olvasása

```
int i = sc.nextInt();
```

Van-e következő sor

```
boolean b = sc.hasNextLine();
```

Ha nem olyan típus következik a beolvasandó csatornán, amit kérünk, akkor hiba keletkezik.

2.5.6. Kiírás

Míg a `System.in` a konzolos bemenetet (billentyűzetet) egyedül képviseli, addig kimenetből hagyományosan kétféle áll rendelkezésre: `System.out` és `System.err`. Az első a szabványos kimenet, a második a szabványos hibakimenet. Mindkettő a képernyőre ír ki, a látványos különbség, hogy a hibakimenet pirossal jelenik meg. [2]

Használatuk:

Szövegkiírás újsor jellel a végén

```
System.out.println("Szervusz világ");
```

Szövegkiírás újsor jel nélkül

```
System.out.print("Szervusz világ");
```

[2] Részben történelmi részben praktikussági okai vannak ennek. A Java nyelv és környezet alapvetően konzolos, tehát karakteres bemenettel és kimenettel rendelkezik, amelytől a modern PC-s operációs rendszerek esetén elszokhattunk. Természetesen ez nem jelenti azt, hogy a Java nyelv esetén nem lehet kényelmes grafikus felhasználói interfészt készíteni.

A `print()` paramétere bármi lehet, a primitíveket a csomagoló osztályon keresztül, a többi osztályt a `toString()` tagfüggvény segítségével alakítja át a Java karakterlánccá. Saját osztály esetében célszerű írni egy `toString()` függvényt.

`toString()` példa

```
public String toString();
{
return "Kutya, név: " + nev + " faj: " + fajta;
}
```

Ha nincs `toString()` függvénye egy osztálynak, akkor a szülőosztály (végsősoron az `Object`) megfelelő függvényét használja, de az legtöbbször nem informatív, mivel az objektumreferencia értékét írja ki.

A `System.out`-hoz hasonló fájl-kimenetet létrehozni a következőképpen lehet:

Fájlba írás példa

```
PrintStream ps = new PrintStream(
new FileOutputStream(filenév));
```

Más módszerek is léteznek a fájlok használatára a kimenet / bemenet iránytól, valamint az írni kívánt egységtől, illetve puffer használatától függően.

Itt röviden megfigyelhető egy példa a `Stream`-ek használatára.

2.5.7. Megjegyzések, dokumentációk a kódban

Lehetőség van a kódban megjegyzések, kommentek elhelyezésére a következő módon

---

Egysoros megjegyzés

```
int i = 0; // számláló
```

Többsoros megjegyzés

```
/* Ez itt egy többsoros megjegyzés eleje
közepe
és vége
*/
```

Ha ezekkel a karakterekkel találkozik a programot értelmező fordító, akkor figyelmen kívül hagyja az adott szakaszt és nem próbálja meg utasításként értelmezni. Arra jók a megjegyzések, hogy az emberi olvasó számára nyújtsanak támpontot a programkód funkciójával, az algoritmus működésével kapcsolatban.

Lehetőség van a kódban függvények és osztályok előtt dokumentációs megjegyzések elhelyezésére. Ezeket a megjegyzéseket JavaDoc kommentnek hívjuk, segítségükkel az elkészített programról kóddokumentáció hozható létre pár kattintással. A JavaDoc-ban elhelyezhetőek hivatkozások, html kódok is.

JavaDoc példa

```
/** Ez itt egy JavaDoc
függvény leírása
```

```
@author SzerzőNeve
@param egy paraméter leírása
@return visszatérési érték leírása
*/
```

2.5.8. Csomagok

A különböző osztályokat tartalmazó forrásfájlok, JAVA kódok úgynevezett csomagokba rendezhetők. Azt, hogy melyik csomagba tartozik egy Java osztály, a fájl elején egy **`package`** kulcsszó mögé írt névvel kell jelezni. (Amikor létrehozzuk Eclipse-ben az osztályt, megkérdezi, hogy milyen csomagba kerüljön.) Az osztályokat mindig a csomag nevén keresztül lehet elérni:

- `programcsomag.Osztály a = new programcsomag.Osztály()` – példa a létrehozásra
- `java.util.Math.sqrt()` – osztályfüggvény hívása esetén

Egyazon csomagban levő osztályok látják egymást, nem kell kiírni a csomagnevet!

Amennyiben szeretnénk elérni, hogy ne kelljen más csomagbéli osztályok neve elé kiírni minduntalan a csomag nevét, lehetséges importálni (láthatóvá tenni) egy teljes csomagot, vagy osztályt egy csomagból. A kulcsszó az `import`.

- **`import`** `java.util.Math` – csak a `Math` osztály importálása
- **`import`** `java.util.*` – minden osztály importálása a `java.util` csomagból

Ezután a `Math.sqrt()` „közvetlenül” használható. A csomagon belül alcsomagok is létrehozhatóak, a *-os import az alcsomagokra nem fog vonatkozni.

2.6. Java

2.6.1. Hogyan működik?

A Java program a számítógépen egy Java Virtuális Gépen fut (JVM), ami lehetővé teszi, hogy ugyanaz a Java program tetszőleges operációs rendszeren (amire a JVM elkészült, például Windows, Linux, MacOS) és géparchitektúrán fusson. (Például mobiltelefonon is.)

Az általunk elkészített *forráskód*ból a fejlesztőkörnyezet egy Java *bájtkód*ot készít, amit a JVM értelmez, és az adott rendszernek megfelelő *gépi kódú* utasítás-sorozattá alakít át.

Ennek megfelelően a Java környezet eleve két részből áll:

- JVM – virtuális gép, a JRE része (Java Runtime Environment)
- JDK – Java Development Kit (a fejlesztői Java, tartalmaz egy JRE-t is)

Jelenleg a 6-os verzió Update 16 a legfrissebb. JRE-t letölteni a http://www.java.com/ oldalról lehet, de ez csak a programok futtatására elég! (A mobilok esetén ez a virtuális gép beépített.)

A fejlesztésre alkalmas Java, ami tartalmaz példákat, forráskódokat, teljes JavaDoc-ot, az a JDK. Többféle verzió áll rendelkezésre:

- Java SE – Standard Edition (Alap verzió, nekünk bőven elég)
- Java EE – Vállalati verzió (Alap + további szolgáltatások)

- Java ME – Mobil verzió (Például telefonokra)

A Java SE-t a http://java.sun.com/javase/downloads/?intcmp=1281 oldalról lehet letölteni.

Ezek az eszközök, programok pusztán a megírt forráskód fordítására, futtatására, (stb.) alkalmasak. Ez természetesen elég lehet, hiszen a kódot egy tetszőleges szövegszerkesztővel is meg lehet írni (Jegyzettömb), de sokkal kényelmesebb környezetek is rendelkezésre állnak.

A félévben az Eclipse IDE (Integrated Development Environment) programot használjuk, ami Java nyelven íródott. (Az egyszerű szövegszerkesztésnél sokkal bonyolultabb feladatok gyors elvégzésére is alkalmas.) A http://www.eclipse.org/downloads/ oldalról az Eclipse IDE for Java Developers választandó.

2.7. Eclipse használata röviden

Az Eclipse programot először elindítva egy Welcome képernyőd minket, amit bezárni a hozzátartozó fülecskén lehet.

Az Eclipse minden egyes indításkor (kivéve, ha másként rendelkezünk a beállításokban) megkérdezi a használni kívánt munkaterület (workspace) nevét. A workspace-ek egy több projektet összefoglaló, állapotukra emlékező környezetek, több lehet belőlük, azonban mindegyik egy saját könyvtárban. Hagyjuk, hogy az alapértelmezett könyvtárban lévőt használja az Eclipse.

Amennyiben sikeresen elindult a program és az üdvözlőképernyőt is becsuktuk az alábbi ábrához hasonló kép tárul elénk. Az ábra azt mutatja, hogy hogyan kell a menürendszer használatával egy új Java projektet létrehozni.

Új projekt létrehozása során meg kell adnunk a projekt nevét, valamint további paramétereket, mint például, hogy melyik JRE-t használja a futtatáshoz. A projekt neve tartalmazhat ékezetes magyar karaktereket, szóközt, de javasolt inkább a szabványos latin abc karaktereit használni.

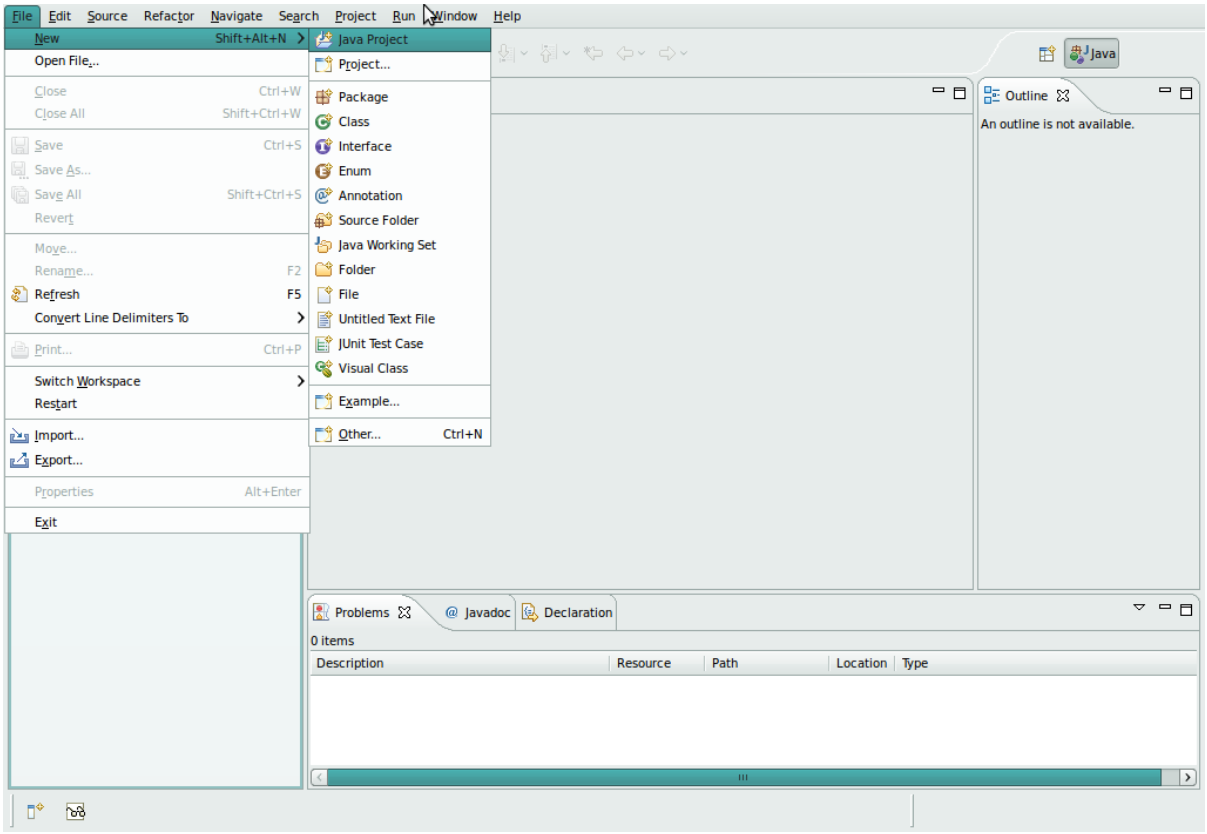
Ha kész a projekt osztályokat hozhatunk létre a varázsló segítségével. Első és legfontosabb osztályunk a `main` függvény tartalmazó osztály lesz, amire a program futtatásához feltétlen szükség van. Kattintani a File menüpont New és azon belül New class pontjára kell.

Megadható létrehozandó osztály neve, módosító kulcsszavak az elérhetőséggel kapcsolatban. Továbbá, hogy mely csomagba tartozzon. (A csomagokról lesz szó később.) Ide beírható nem létező csomag neve is, az Eclipse automatikusan létre fogja hozni a megfelelő nevű csomagokat. Megadható a szülőosztály neve is, amelyet egy listából is kikereshetünk. (Superclass) Valamint az interfészek. Az ezt követő opciók sorrendben:

- Létrehozza a main függvényt ebben az osztályban. A létrejött függvény teljesen üres.
- A szülőosztály konstruktorainak megfelelő konstruktorokat létrehoz ebben az osztályban is.
- Az absztrakt, meg nem valósított metódusokat megvalósítja. (Üres függvényként.) Az öröklődésnél van szerepe.

A létrejött osztályban a `main` függvényt megszerkesztve a programot elindítani az eszköztár megfelelő gombjával lehet, vagy a Run menüpont Run utasításával. (A gomb egy zöld körben levő háromszög.) Bizonyos esetekben az Eclipse rákérdez, hogy milyen alkalmazást akarunk futtatni.

A program futása során létrejövő kimeneteit alapértelmezésben alul lehet olvasni a Console feliratú fülnél. (Ezek a fülek tetszőlegesen áthelyezhetők, bezárhatók.



2.2. ábra. Új Java projekt létrehozása

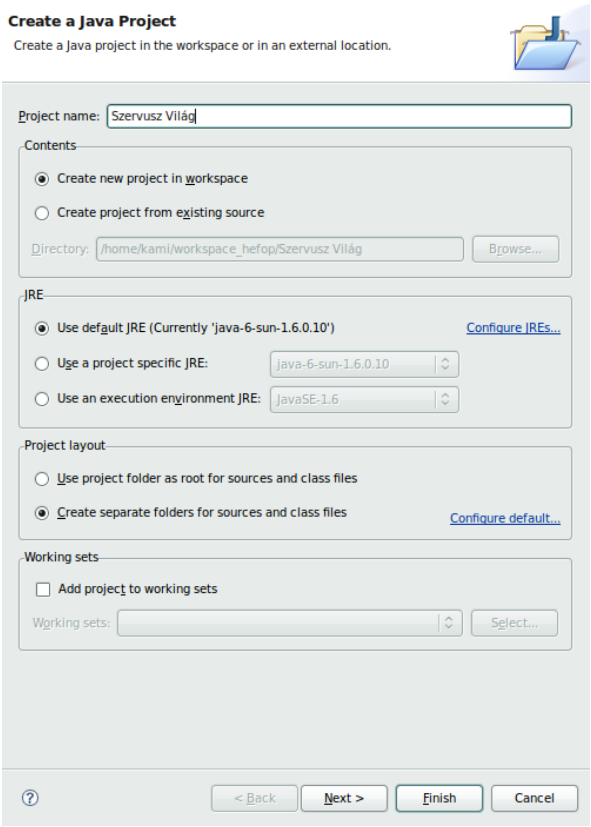
2.8. Rekurzió

Rekurzív egy függvény, ha a számítás során rész-számításokhoz önmagát hívja meg. Általában egy egyszerűbb eset visszavezetésére használjuk, vagy a rész-esetek további kibontására. Ennek megfelelően a függvényhívások száma jócskán növekedik egy bonyolultabb számítás elvégzéséhez.

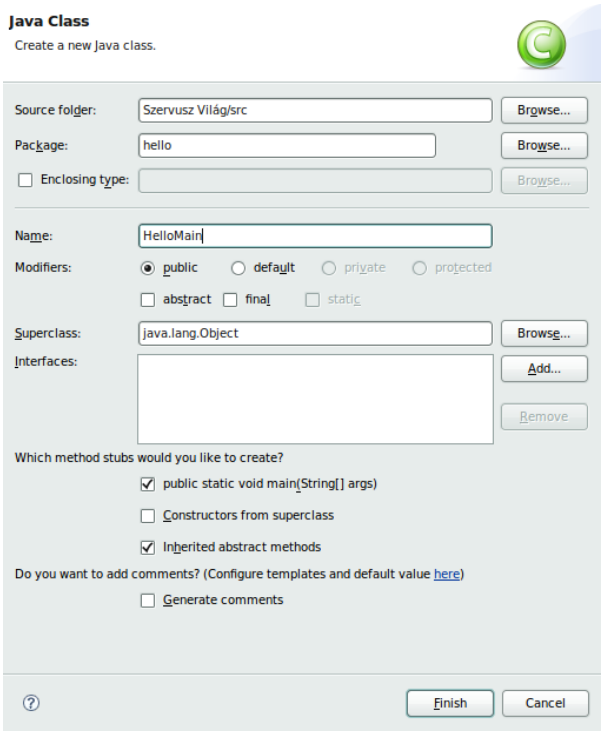
Például, ha egy faktoriálist szeretnénk kiszámolni, akkor rekurzív függvényhívással is meg lehet oldani a problémát a következőképpen: Adott n, az ehhez tartozó $n!$ az nem más, mint $n * ((n-1)!)$. Azaz a problémát egy szorzássá alakítottuk át, most már csak az eggyel kisebb szám faktoriálisát kell kiszámolni. Ezt tudjuk folytatni tovább, egészen addig, amíg $n = 0$, mivel annak ismert a faktoriálisa.

Rekurziós módszerek esetén, mivel a függvény saját magát hívogatja, nehéz követni, hogy hol tart. A faktoriális számítása esetén ez nem gond, mivel nincs benne elágazás. Azonban a legtöbb problémánál elágazások vannak a függvényben.

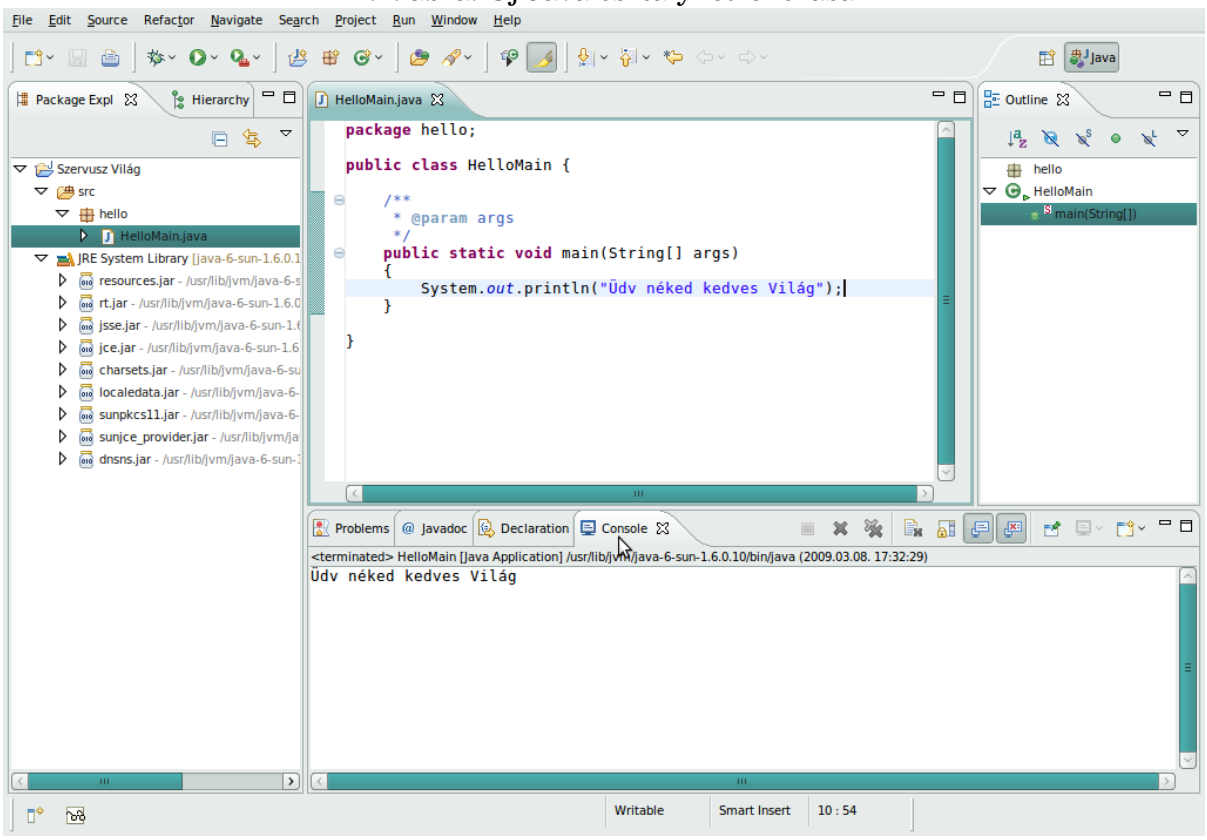
Azt fontos megjegyezni, hogy egy függvény futása addig nem fejeződik be, amíg nem futtat egy **`return`** utasítást. Addig a belső változói deklaráltak, és van értékük. Amikor egy függvényt hív, akkor annak a függvénynek is lesz egy saját területe, és így tovább. Tehát, ha végiggondoljuk, hogy $n = 60$ esetre összesen hatvan hívás történik. Minden esetben lefoglalásra kerül a függvényhez tartozó összes változó és paraméter, valamint némi memória szükséges a függvényállapot tárolására, amelyből a hívás történt. Ez jelentékeny memóriahasználat a probléma egyszerűségéhez viszonyítva. Faktoriális számítása esetén lineárisan növekszik a tárigény, azonban vannak olyan problémát, ahol sokkal gyorsabban fogy el a memória a rekurziós számításnál. (Szemben egy nem-rekurzív megoldással.)



2.3. ábra. Új Java projekt létrehozása – Paraméterek



2.4. ábra. Új Java osztály létrehozása



2.5. ábra. A main függvény és az eredmény

3. fejezet

Absztrakt adatszerkezet

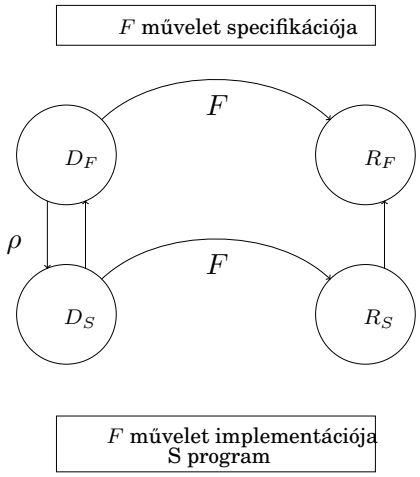
Emlékezzünk a típus definíciójára, miszerint a típus a felvehető értékek halmazát és azokon végezhető műveletek összességét jelenti.

Ebben a fejezetben a típus specifikálással és absztrakcióval foglalkozunk, legelőször áttekintve a típus absztrakció szintjeit.

A **reprezentációs szint**en a típusértékek ábrázolásáról beszélünk. A legabsztraktabb szint, amelyen meghatározzuk a típusértékekhez tartozó ábrázoló elemek halmazát, valamint a műveletek kétirányú reprezentációs leképezését. Tehát a típusértékek lehetnek például a napok: HÉTFŐ, KEDD, SZERDA, CSÜTÖRTÖK, PÉNTEK, SZOMBAT, VASÁRNAP. Az ehhez tartozó ábrázoló elemek egy lehetséges esetben a természetes számok 1 és 7 között értelemszerű megfeleltetéssel. A *nap* típus esetén egy művelet lehetségesen a rákövetkező nap megadása. Ennek a típusértékek halmazán van egy jól definiálható működése, hogy milyen érték esetén mi lesz a művelet eredménye. Ezt a műveletet kell átültetni a természetes számok megszorított halmazára. A reprezentációs szinthez tartozó fogalmak:

- ábrázoló *elemek* H halmaza (típus-szerkezet), a példában ezek a számok.
- az ábrázoló elemek és a típusértékek *kapcsolatát* leíró leképezés: $\rho : H \to T, \rho \subseteq H \times T$
- a *típus-invariáns* kiválasztja a hasznos ábrázoló elemeket: $\mathbf{I} : H \to \mathbf{L}, [\mathbf{I}]$ Ez a leképezés választja ki a számok közül a használandó $1..7$ intervallumot.

A leképzést az alábbi ábra szemlélteti: F az eredeti művelet, a felső szint a a műveletet



3.1. ábra. F művelet leképezése S programra.

mint függvényt ábrázolja, ahol az értelmezési tartomány D_F és az értékkészlet meghatározott R_F. Ezzel szemben az alsó szinten leképezés utáni ρ ábrázoló elemek halmazán működő S program megfelelő halmazait ábrázoljuk.

3.1. ADT

Absztrakt adattípus (abstract data type) a típus-specifikáció (közvetett) megadására szolgál. Nem szükséges, hogy egy konkrét programozási környezetben ábrázoljuk a típusértékeket, alapvetően matematikai eszközökkel is megadható. ADT leírás esetén elég a műveletek programjainak csak a hatását ismerni.

Absztrakt a programozási környezet számára és a megoldandó feladat számára, amely adattípust a későbbiekben egy kiváltó (konkrét) típussal helyettesítünk.

Az ADT megközelítés a típus szemléletének ez a legmagasabb szintje semmilyen feltételezéssel nem élünk a típus szerkezetéről, megvalósításáról. Ahogyan az előzőekben említve van, a specifikációban csak tisztán matematikai fogalmakat használhatunk. (Ez a szint nem a formalizálás mértékétől absztrakt, lehet informálisan is gondolkodni, beszélni ADT szinten!)

Az ADT leírás részei:

- típusérték-halmaz (milyen lehetséges értékek vannak)
- műveletek (mint leképezések, ahogyan azt az előző ábrán láthattuk)
- megszorítások (értelmezési tartományok, nem minden művelet értelmezhető az összes lehetséges értéken. Gondoljunk a nullával való osztásra.)
- axiómák, amelyek biztosítják a típus és műveleteinek helyes felépítését és működését. (Az axiómák alapigazságok, amiknek mindig teljesülnie kell.)

Kérdések, amelyeket a specifikáció során vizsgálni kell, vagy vizsgálni lehet:

- helyesség (ellentmondás-mentesség, az egyes axiómák közötti ellentmondásokat meg kell szüntetni)
- teljesség a leírt axiómák és specifikációk hiánytalanul leírják a típus és műveleteinek működését. *nehéz bizonyítani*)
- redundancia (ugyanazt a tulajdonságot, axiómát nem fogalmaztuk-e meg többféleképpen, többször. *vizsgálata a működés szempontjából nem fontos*)

Például egy ADT funkcionális specifikációja az alábbiakból áll:

- típusérték-halmaz
- műveletek
- állapottér
- paramétertér
- előfeltétel
- utófeltétel

A funkcionális specifikációhoz a típus matematikai reprezentációját használjuk. Ez semmilyen módon nem kell, hogy utaljon a típus ábrázolási módjának megválasztására a megvalósítás során, teljesen más is lehet, pusztán matematikai eszközöket használunk. (Fontos, hogy leggyakrabban nem így fogjuk ténylegesen megvalósítani, implementálni.)

3.2. ADS

Az absztrakt adatszerkezet (abstract data structure) során a típus alapvető – absztrakt – szerkezetét egy irányított gráffal ábrázoljuk. A gráfban a csúcsok az adatelemek az élek a rákövetkezési relációk. Tehát adatelemek valamilyen struktúra illetve reláció szerinti reprezentációját adja meg egy gráf.[1]

ADS szinten is lehet ábrázolni a műveleteket, mégpedig a műveletek hatása szemléltethető az ADS-gráf változásaival. Egy új adatelem elhelyezése a gráfban növeli a csomópontok számát, valamint új éleket adunk hozzá.

3.3. Adatszerkezetek osztályozása

Az adatszerkezetek osztályozásához legelőbb definiáljuk az adatszerkezet fogalmát a következőképpen: Az adatszerkezet egy $\langle A, R\rangle$ rendezett pár, ahol az A az adatelemek véges halmaza, valamint az R az A halmazon értelmezett valamilyen reláció ($A \times A$). Itt a reláció absztrakt fogalomként értendő mindösszesen két adatelem valamely kapcsolatát jelenti. (Tehát nem a hagyományos értelemben megszokott kisebb $<$ vagy nagyobb $>$ összefüggésről van szó, hiszen általános esetben ezt nem is feltétlen tudjuk definiálni két adatelem között. Logikai kapcsolatot jelent két adatelem között. Ezt a logikai kapcsolatot jelöljük az ADS gráf éleivel.)

Az osztályozás többféleképpen lehetséges az adatszerkezetekre nézve:

Az adatelemek típusa szerint.

- **Homogén**: Az adatszerkezet valamennyi eleme azonos típusú. Például mindegyik szám.
- **Heterogén**: Az adatszerkezet elemei különböző típusúak lehetnek. Vegyesen szerepelnek különböző típusok, például számok és karakterek.

Az elemek közti R reláció szerint.

- **Struktúra nélküli**. Az egyes adatelemek között nincs kapcsolat. Nem beszélhetünk az elemek sorrendjéről (pl. halmaz). (Gyakorlatilag a reláció nem határoz meg viszonyt az elemek között, minden egyes elem egyenrangú, nincs elé-/alárendeltség.)
- **Asszociatív címzésű**. Az adatelemek között lényegi kapcsolat nincs, ugyanakkor az adatszerkezet elemei tartalmuk alapján címezhetőek, azaz egy adatelem megtalálásához a tartalmával képzett címzéssel megtalálható a szerkezetben.
- **Szekvenciális**. A szekvenciális adatszerkezet olyan $\langle A, R\rangle$ rendezett pár, amelynél az R reláció tranzitív lezártja teljes rendezési reláció. Minden egyes adatelempárról megmondható hogy egymáshoz képest milyen viszonyban állnak. Ez vagy közvetlenül történik, mert az R értelmezett a kiválasztott páron, vagy pedig tranzitíven[2] A szekvenciális adatszerkezetben az egyes adatelemek egymás után helyezkednek el logikai módon, vagyis ez a fizikai reprezentációt nem befolyásolja. Az adatok között egy-egy jellegű a kapcsolat: minden adatelem csak egy helyről érhető el, és az adott elemről csak egy másik látható. Az egyes adatelemekről a szomszédjaik megállapíthatóak. Két kitüntetett elemről beszélhetünk, ami az első és az utolsó.

[1] A gráf egy olyan konstrukció, amelyet csomópontok és az azokat összekötő vonalak alkotják. Ez utóbbiakat hívjuk éleknek. Egy gráf irányított, ha az éleknek van irány, tehát az összekötő vonalak nyilak. Irányított gráfban az irányítás rákövetkezőségeket definiál.

[2] Tranzitivitás: Ha a relációban áll b-vel és b relációban áll c-vel, akkor a is relációban áll c-vel.

- **Hierarchikus**. A hierarchikus adatszerkezet olyan $\langle A, R \rangle$ rendezett pár, amelynél van egy kitüntetett r elem, ez a gyökérelem úgy, hogy r nem lehet végpont [3]. $\forall a \in A \setminus \{r\}$ elem egyszer és csak egyszer végpont, vagyis minden r-en kívüli elem a relációban egyszer lehet végpont. $\forall a \in A \setminus \{r\}$ elem r-ből elérhető, azaz minden elem elérhető az r-ből a relációk követésével. Az adatelemek között egy-sok jellegű kapcsolat áll fenn. Minden adatelem csak egy helyről érhető el (egyetlen megelőzője van), de egy adott elemből akárhány adatelem látható (akárhány rákövetkezője lehet. Ilyen például a fa, összetett lista, B-fa.).
- **Hálós**. A hálós adatszerkezet olyan $\langle A, R \rangle$ rendezett pár, amelynél az R relációra semmilyen kikötés nincs. Az adatelemek között a kapcsolat sok-sok jellegű: bármelyik adatelemhez több helyről is eljuthatunk, és bármelyik adatelemtől elvileg több irányban is mehetünk tovább (Például: általános gráf, irányított gráf).

Az adatelemek száma szerint.

- **Statikus**. Egy statikus adatszerkezetet rögzített számú adatelem alkot. A feldolgozás folyamán az adatelemek csak értéküket változtathatják, de maga a szerkezet, az abban szereplő elemek száma változatlan. Következésképpen az adatszerkezetnek a memóriában elfoglalt helye változatlan a feldolgozás során.
- **Dinamikus**. Egy dinamikus adatszerkezetben az adatelemek száma egy adott pillanatban véges ugyan, de a feldolgozás során tetszőlegesen változhat. Dinamikus adatszerkezetek lehetnek rekurzív vagy nem-rekurzív, lineáris vagy nem-lineáris struktúrák. Egy adatszerkezet rekurzív, ha definíciója saját magára való hivatkozást tartalmaz. Ha egyetlen ilyen hivatkozás van, akkor lineáris a struktúra, ha több, akkor nem-lineáris. Mivel a dinamikus adatszerkezetek feldolgozása során az adatelemek száma változik, egy-egy elemnek területet kell lefoglalnunk, illetve a lefoglalt területeket fel kell szabadítanunk, így felvetődik a tárolóhely újrahasznosításának problémája.

[3] A végpont gyakorlatilag itt a reláció jobboldalát jelenti, azaz ha végpont akkor a relációban szereplő két elem közül a rákövetkezőről van szó. Például az $a \rightarrow b$ esetén a b a végpont és így a rákövetkező elem. Értelemszerűen az a a megelőző elem. Az a-ból elérhető a b.

4. fejezet

Adatszerkezetek

4.1. Verem

A Verem adatszerkezet olyan, mint egy szűk verem. Belekerülnek sorban az elemek. Az első legalulra esik, a második rá, és így tovább. „Kimászni” mindig a legfelső tud, és ha ránézünk felülről, akkor mindig a legfelsőt látjuk csak.

4.1. ábra. Verem példa

Az előző példában a legelsőként a Kutya kerül ki a veremből, amiután a Farkas fog látszani a tetején. Legelsőként a Sün került bele és a végén jön ki. A Verem (Stack) egy LIFO adatstruktúra. (Last In, First Out)

4.1.1. ADT leírás

Az alábbi módon lehet definiálni: A V verem E alaptípus felett jön létre. Műveletei:

- empty: $\longrightarrow V$ (Üres verem létrehozása.)
- isEmpty: $V \rightarrow L$ (Állapot lekérdezése: üres-e.)
- push: $V \times E \rightarrow V$ (Új elem beszúrása.)
- pop: $V \rightarrow V \times E$ (Legfelső elem kivétele a veremből.)
- top: $V \rightarrow E$ (Legfelső elem lekérdezése.)

Az egyes műveleteknél szerepel, hogy milyen bemenettel rendelkeznek és milyen típusú kimenetet állítanak elő. Megszorítás, hogy a pop és top műveletek értelmezési tartománya: $V \setminus \{empty\}$, azaz üres veremből nem lehet kivenni és nem lehet megnézni, hogy mi van a tetején.

A V jelenti a vermet mint típust. E típusú elemek kerülhetnek a verembe és a fentiekben használt L jelenti a logikai típust.

A verem műveletekhez tartozó axiómák, amelyeknek logikai kifejezésként minden esetben igaznak kell lenniük:

- isEmpty(empty) – Egy üresnek létrehozott verem legyen üres.
- isEmpty(v) $\rightarrow$ v = empty – Ha egy v veremre a lekérdezés, hogy üres-e igaz, abból következik, hogy a v az „üresverem”.
- $\neg$ isempty(push(v,e)) – Ha beteszünk egy elemet egy v verembe, az nem lehet üres ezután.
- pop(push(v,e)) = (v,e) – Ha beteszünk egy elemet egy v verembe, majd kivesszük akkor az eredeti vermet és elemet kell kapnunk.
- push(pop(v)) = v – Ha kiveszünk egy elemet a veremből, majd visszatesszük az eredeti v vermet kell kapnunk.
- top(push(v,e)) = e – Ha beteszünk egy elemet a verembe, majd megnézzük a verem tetjét a betett elemet kell látnunk.

4.1.2. ADT funkcionális leírás

Matematikai reprezentáció, miszerint a verem rendezett párok halmaza, ahol az első komponens a veremben elhelyezett (push) érték, a második komponens a verembe helyezés (push) időpontja. Megszorítás (invariáns): az időértékek különbözőek. Ez egy jól kezelhető matematikai modell, azonban nem így implementáljuk, hiszen aligha találni bonyolultabb implementációs módszert.

A modellhez tartozó függvényekre el kell készíteni a specifikációt, ami a pop esetén az alábbi:

- $A = V \times E$ – állapottér (v és e lesz egy-egy aktuális érték)
- $B = V$ – paramétertér (v')
- $Q = (v = v' \wedge v' \neq 0)$ – előfeltétel, miszerint a bemeneti verem az nem üres.
- $R = ((v = v' \setminus \{(e_j, t_j)\}) \wedge (e = e_j) \wedge ((e_j, t_j) \in v') \wedge (\forall i((e_i, t_i) \in v' \wedge i \neq j) : t_j > t_i))$ – utófeltétel, a kifejezés szakaszai az alábbiak: A kimenet v és e, ahol a v verem az eredeti verem, kivéve az e_j, t_j párt; és az e az e_j. Teljesülni kell annak, hogy az e_j és t_j pár eredetileg benne volt a veremben, valamint minden más e_i és t_i párra igaz, hogy az időindex az kisebb mint t_j.

4.1.3. Statikus verem reprezentáció

A statikus reprezentáció esetén a veremben tárolható elemek maximális száma rögzített, így például használhatunk egy fix méretű tömböt, ami tömbbe fogjuk a verembe betett elemeket elhelyezni. (Természetesen folyamatosan ügyelni kell arra, hogy a behelyezés és kivétel a verem szabályoknak megfelelő legyen.) A következő egységei lesznek a Verem típusnak a reprezentáción belül:

- Max mező, a verem maximális méretét határozza meg, emiatt lesz egy újabb függvény, ami lekérdezi, hogy tele van-e a verem.
- Max méretű tömb, amiben a verembe kerülő elemeket tároljuk.
- $Head$ változó, ami azt mutatja, hogy hol a verem teteje.

A következő szakaszban egy Java nyelven megvalósított statikus verem forráskódját nézzük meg. Vegyük észre, hogy a Verem osztály megvalósításánál a reprezentációhoz tartozó speciális mezők nem láthatóak az osztályon kívül senki számára. Korábban volt szó

arról, hogy egy osztály felelős a saját állapotáért és annak változásért. Egy verem osztálynak a verem axiómák által támasztott követelményeknek kell megfelelniük, amit érvényes állapotokat jelentenek. A megvalósított műveletek ezeket az érvényes állapotokat fenntartják. Azonban ha kívülről beavatkoznánk és megváltoztatnánk például az index értékét, akkor helytelen állapotba kerülne az objektum, a verem. Ezen hibák kiküszöbölését segítik a változók láthatóságának helyes beállításai.

Implementáció statikusan

Az első kódszakaszban elsőként a Verem osztály mezői kerülnek deklarálásra, az előzőeknek megfelelően egy `int` típusú `head` nevű változó az indexelésre, valamint egy `elemek` nevű változó, ami a verembe tehető `int` értékeket tartalmazó tömb lesz. Az indexelő változó a verem reprezentációjára használt tömb azon indexét tartalmazza, ahova következő betétel során elem kerülhet. (Tehát a legelső szabad pozíciót. A megvalósításnál természetesen lehetséges, hogy a legfelső elfoglalt elemet indexeljük, csak a megfelelő függvényekben erre figyelni kell.)

A meződeklarációk után a konstruktor található, ami az `empty()` üres verem létrehozására, illetve a verem kiürítésére használatos függvény. Amikor üres a verem a `head` index nulla.

```
package verem;

public class Verem_statikus
{
    // Következő szabad pozíciót mutatja
    private int head;
    // Tömb amiben az elemek lesznek
    private int[] elemek;

    /**
     * Konstruktor, ami létrehoz egy üres vermet
     */
    public Verem_statikus()
    {
        empty();
    }

    /**
     * Egy maximum tíz elemet tároló verem létrehozása üresen
     */
    public void empty()
    {
        head = 0;
        elemek = new int[10];
    }
```

4.2. ábra. Statikus verem kód 1. rész

A `push()` függvény egy elemet betesz a verembe. A betétel csak akkor történhet meg ha nincs még tele a verem. Erre van a feltétel vizsgálat, ami hiba esetén a konzolra kiírja a probléma forrását. Ha azonban nincs tele a verem, akkor a tömbbe beírhatjuk a betenni kívánt elem értékét, mégpedig a `head`-edik pozícióba, mivel az jelenti a következő szabad helyet a tömbben. A `head` index növelése után készen is van a betétel.

A `top()` függvény a legfelső elem értékével tér vissza. A legfelső tényleges elem a `head-1`-edik pozícióban van, mivel a `head` a legelső szabad pozíciót tárolja. Amikor üres

veremből kérdezzük le a legfelső elemez, akkor is ad vissza a függvény egy értéket, ami a -1. Az érték-visszaadásra kényszerből kerül sor, az efféle hibakezelés rossz megoldás, mivel nem tudunk különbséget tenni a hiba és a -1 értéket legfelső elemként tartalmazó verem teteje között. (Egyelőre nincsen jobb eszköz az ismereteink között, amivel lehetne orvosolni ezt a problémát.)

```
/**
 * Egyetlen elem betétele a verembe
 * @param elem A beteendő elem
 */
public void push(int elem)
{
    if (isFull())
    {
        System.out.println("Tele van");
    }
    else
    {
        elemek[head] = elem;
        head++;
    }
}

/**
 * Legfelső elem megtekintése
 * @return mi van legfelül
 */
public int top()
{
    if (isEmpty())
    {
        System.out.println("Üres");
        // Ez itt csúnya nagyon, de most nincs más eszköz
        return -1;
    }
    else
    {
        return elemek[head-1];
    }
}
```

4.3. ábra. Statikus verem kód 2. rész

A `pop()` függvény hasonló a `top()`-hoz. A különbség csupán annyi, hogy a `head` index csökkentésére is szükség van, hiszen kivesszük a veremből az értéket. Megfigyelhető, hogy az érték tényleges eltüntetésére nem kerül sor. Erre azonban nincsen szükség, hiszen a tömbben lehet bármi, minket csak az általunk nyilvántartott és betett értékek érdekelnek. Amennyiben kiveszünk a veremből, akkor az a pozíció a szabad pozíciók közé fog tartozni a veremben használt tömbben. Ugyanis ha ezen után beteszünk valamit a verembe, az pont a kivett érték helyére kerül. (A létrehozott változókban is van érték, a memóriacella aktuális értéke, ami számunkra nem hordoz hasznos információt, gyakorlatilag szemét. Amikor létrejön a tömb az egyes pozíciókban már eleve lesznek értékek, amikkel ugyanúgy nem törődünk mint a kivettel.)

Az utolsó szakaszban az üresség és magteltség vizsgálatára kerül sor. Üres a verem, ha a legelső szabad pozíciót a tömb legelső indexe. Tele a verem, ha a legelső szabad pozíció kiindexel a veremből. (Megjegyzés: egy n hosszú verem 0 és $n-1$ között indexelhető.)

```java
/**
 * Legfelső elem kivétele és visszaadása
 * @return legfelső elem
 */
public int pop()
{
    if (isEmpty())
    {
        System.out.println("Üres");
        // Ez itt csúnya nagyon, de most nincs más eszköz
        return -1;
    }
    else
    {
        // Még azt mondja a head, hogy mi a következő szabad pozíció
        head--;
        // Most a head az utolsó foglaltra mutat
        return elemek[head];
        // Visszatért, és az is igaz, hogy az új következő szabadon van,
        // mivel "kivettünk" egy elemet
    }
}
```

4.4. ábra. Statikus verem kód 3. rész

```java
    /**
     * Verem ürességének vizsgálata
     * @return üres-e
     */
    public boolean isEmpty()
    {
        return (head == 0);
    }

    /**
     * Megvizsgálja, hogy tele-e a verem
     * @return tele-e
     */
    public boolean isFull()
    {
        return (head == elemek.length);
    }
}
```

4.5. ábra. Statikus verem kód 4. rész

4.1.4. Dinamikus verem reprezentáció

Dinamikus reprezentáció esetén nem tömbben tároljuk az értékeket, hanem erre a célra speciálisan létrehozott Elem osztályokban, amik egy láncolatnak lesznek a csomópontjai. Az Elem osztály tartalmaz egy értéket, valamint egy referenciát egy következő elemre, így lehet majd tárolni, hogy „mi van a veremben egy érték alatt”. A *Head* referencia tárolja a verem elejét, ami ha nem mutat érvényes elemre, akkor üres a verem. A dinamikus reprezentáció esetén tetszőleges számú elem elfér a veremben. (Természetesen a memóriakorlát továbbra is él.)

Implementáció dinamikusan

Az első szakaszban a dinamikus verem belső osztálya a `Node` található meg legelsőként. A `Node` tartalmazza egy aktuális csomópont értékét, valamint egy referenciát a következő `Node`-ra. A gyors létrehozás érdekében egy két-paraméteres konstruktor a mezőknek értékeket ad. A dinamikus veremben egy `head` nevű mező fogja a verem tetejét (azon végét, ahonnan kivenni és ahova betenni lehet). A `head` egy `Node` típusú referencia, ami a verem létrehozáskor **`null`**, azaz nem referál sehova sem. Ez egyben az üresség feltétele is. Akkor és csak akkor üres a verem ha `head` **`null`**.

```
package verem;

public class VeremDinamikus
{
    class Node
    {
        public int ertek;
        public Node kovetkezo;

        public Node(int ertek, Node kovetkezo)
        {
            this.ertek = ertek;
            this.kovetkezo = kovetkezo;
        }
    }

    private Node head;

    public VeremDinamikus()
    {
        head = null;
    }

    public boolean isEmpty()
    {
        return (head == null);
    }
```

4.6. ábra. Dinamikus verem kód 1. rész

Elem betételekor egy új Node létrehozására van szükség. Az új `Node` rákövetkezője az aktuális head, így történik a lánc fűzése, valamint az értéke a betenni kívánt érték. Természetesen az újonnan létrehozott `Node` lesz a verem új teteje, ezt az értékadással valósítjuk meg. A `pop()` függvény, hasonlóan a statikus esethez egy vizsgálattal kezdődik. Ha nem üres a verem, akkor eltároljuk a legfelső `Node`-ban található értéket egy ideiglenes változóba, majd lefűzzük a legfelső csomópontot a veremről, azaz a verem új teteje a első csomópontot követő `Node` lesz. Végül visszatér az eltárolt értékkel. Az eltárolásra azért van szüksége, mert a fűzésnél elveszítjük az egyetlen referenciát a legfelső csomópontra, amiben a hasznos érték található. (Viszont a referenciák elvesztésére is szükség van, mert ebből tudja a Java környezet, hogy már nem használjuk azt a csomópontot, így felszabadíthatja a hozzátartozó memóriaterületet.

A `top()` hasonló a `pop()`-hoz, éppen csak nem történik meg a lefűzés.

```java
public void push(int elem)
{
    // A head referencia értéke egy olyan új csomópont lesz,
    // amiben a beszúrandó érték szerepel,
    // valamint a rákövetkezője az eddigi head
    head = new Node(elem, head);
}

public int pop()
{
    if (isEmpty())
    {
        System.err.println("Üres a verem - hiba");
        // Még mindig csúnya
        return -1;
    }
    else
    {
        // Megjegyezzük a visszatérési értéket
        int visszaterni = head.ertek;
        // Megváltoztatjuk a head értékét a következőre -> kivesszük
        head = head.kovetkezo;
        return visszaterni;
    }
}
```

4.7. ábra. Dinamikus verem kód 2. rész

```java
    public int top()
    {
        if (isEmpty())
        {
            System.err.println("Üres a verem - hiba");
            // Még mindig csúnya
            return -1;
        }
        else
        {
            return head.ertek;
        }
    }

}
```

4.8. ábra. Dinamikus verem kód 3. rész

4.2. Sor

A Sor adatszerkezet olyan, mint egy várakozás sor, például a postán. Belekerülnek sorban az elemek. Az első legelőre, a második mögé az elsőnek, és így tovább. (Az alábbi jelölésben a sor első elemét *dőlt*, az sorban következő szabad helyet pedig aláhúzott kiemeléssel van feltüntetve.)

20	30	2	1	0	3	_		

Mindig a legelső tud kijönni, tehát a legrégebben bekerül elem. Ez az előző példában a 20. Ránézvén a soron következő elemet láthatjuk mint első elem. A következő bekerülő elem a 3 mögé fog kerülni. A Sor (Queue) egy FIFO adatstruktúra. (First In, First Out)

4.2.1. ADT Axiomatikus leírás

A Sor ADT axiomatikus leírása: Az alábbi módon lehet definiálni: A S sor E alaptípus felett jön létre. Műveletei:

- Empty: $\longrightarrow S$ (az üres sor konstruktor – létrehozás)
- IsEmpty: $S \rightarrow L$ (üres a sor?)
- In: $S \times E \rightarrow S$ (elem betétele a sorba)
- Out: $S \rightarrow S \times E$ (elem kivétele a sorból)
- First: $S \rightarrow E$ (első elem lekérdezése)

Az egyes műveleteknél szerepel, hogy milyen bemenettel rendelkeznek és milyen típusú kimenetet állítanak elő. Megszorítás, hogy az Out és First értelmezési tartománya: $S \setminus \{Empty\}$ azaz üres sorból nem lehet kivenni és nem lehet megnézni, hogy mi van az elején.

4.2.2. Statikus sor reprezentáció

A statikus reprezentáció esetén a sorban tárolható elemek maximális száma rögzített, így például a veremhez hasonlóan használhatunk egy fix méretű tömböt. A következő egységei lesznek a Sor típusnak a reprezentáción belül:

- Max mező, a sor maximális méretét határozza meg, emiatt lesz egy újabb függvény, ami lekérdezi, hogy tele van-e a verem.
- Max méretű tömb, amiben a sorba kerülő elemeket tároljuk.
- $Head$ változó, ami azt mutatja, hogy hol a sor eleje. `head ∈ [1, max]`
- $Tail$ változó, ami azt mutatja, hogy hol a sor vége, vagyis az első üres helyet `tail ∈ [1, max]`

Nyilvánvalóan akkor van ténylegesen tele a a sor, ha a statikus reprezentációban használt tömb esetén nincs már szabad pozíció a tömbben. Tegyük fel az előzőekben leírt példát kiindulásnak. Ha kivesszük az első két elemet és továbbiakat teszünk be a végén, az alábbiakat kapjuk.

_		*2*	1	0	3	9	10	5

Ekkor láthatóan nincs tele teljesen a tömb, ugyanakkor a betételnél kifutunk a tömbből. Ezt úgy tudjuk orvosolni, ha a tömb lehető legjobb kihasználtsága érdekében körkörösen fogjuk használni, azaz ha a végén kifut egy index, azt beléptetjük az elején. Tehát egy újabb elem beszúrása az első tömbpozícióba fog történni az alábbi módon:

123	_	*2*	1	0	3	9	10	5

Ezek után pusztán egyetlen problémát kell megoldani. Ugyanis ha beszúrunk még egy elemet, akkor:

123	321	*2*	1	0	3	9	10	5

Itt a Head és a Tail indexek pontosan egyeznek, tehát azt mondhatjuk, hogy a tömb tele van, ha a Head és a Tail ugyanazt az értéket mutatja. (Emlékeztetőül: a Head a soronkövetkező elem indexe, a Tail pedig az első szabad pozíció.) Ugyanakkor, ha mindent kiveszünk a sorból:

		_						

A sor üres az első szabad pozíció a példát folytatva a harmadik, ugyanakkor a következő kivehető elem is a harmadik az indexek szerint. Ebből azt a következtetés vonjuk le, hogy a sor üres, ha a Head és a Tail ugyanazt az értéket tartalmazza.

Így a Head és Tail változó egyenlősége esetén kétféle esemény fordulhat elő, amelyet nem tudunk megkülönböztetni. A probléma megoldására több lehetőség áll rendelkezésre:

- Vezessünk be még egy jelzőt a reprezentációba, ami mutatja, hogy a sor üres-e, a neve legyen empt. Kezdetben biztosan igaz, később vizsgáljuk, és megfelelően állítjuk. (Ha kiveszünk akkor kiürülhet a sor, egyéb esetben nem.)
- Vezessünk be még egy attribútumot a reprezentációba, ami mutatja, hogy hány elem van a sorban. Ha a számláló az a maximális betehető értékek számával egyenlő, akkor tele van a sor, különben biztosa nem.

Implementáció statikusan

A kód első részében az előzőeknek megfelelően a változók deklarációja történik, valamint az `empty()` konstruktor elkészítése. Üres sor létrejöttekor a `head` és `tail` változók az `elemek` tömb első pozíciójára vagyis a nulla indexre mutassanak, valamint kezdetben a sor üres. Az ürességet és teliséget lekérdező függvény pedig figyelembe veszi az ürességet jelző logikai változót.

```
package sor;

public class SorStatikus
{
    private int [] elemek = new int[10];
    private int head;
    private int tail;
    private boolean ures;

    public void empty()
    {
        ures = true;
        head = 0;
        tail = 0;
    }

    public boolean isEmpty()
    {
        return ures;
    }

    public boolean isFull()
    {
        return ((head == tail) && (!ures));
    }
```

4.9. ábra. Statikus sor kód 1. rész

A következő függvény a sorban elemet elhelyező `In()` függvény. Az elhelyezéskor az eddigieknek megfelelően ellenőrizzük, hogy a sor nincs-e tele. Az elem behelyezésekor biztosan nemüres sort fogunk kapni. Az elem elhelyezése a `tail`-edik indexen történik, mivel az jelenti a következő szabad pozíciót. Ezek után történik a `tail` index növelés, valamint, ha a növelést követően kiindexel a tömbből akkor a körkörösség értelmében a nulladik pozícióra fog mutatni.

A következő függvény a sorból az első elemet kivevő `Out()` függvény. A visszaadandó értéket egy ideiglenes változóba kimásoljuk, mivel a **`return`** utasítás előtt kell minden

```
public void In(int amit)
{
    if (isFull())
    {
        System.err.println("Tele van - nem lehet még pakolni bele");
    }
    else
    {
        ures = false;
        elemek[tail] = amit;
        tail++;
        if (tail == elemek.length)
        {
            tail = 0;
        }
    }
}
```

4.10. ábra. Statikus sor kód 2. rész

akciót végrehajtani. (Az index megváltoztatása előtt, könnyebb az aktuális visszaadandó értéket megtalálni.) Az érték megjegyzését követően hasonlóan az `In()` függvényhez az index körkörös léptetése történik meg. Majd annak a vizsgálata következik, hogy a kivétel után üressé vált-e a sor.

```
public int Out()
{
    if (isEmpty())
    {
        System.err.println("Üres - nem lehet kivenni belőle");
        return -1;
    }
    else
    {
        int ideiglenestarolo = elemek[head];
        head++;
        if (head == elemek.length)
        {
            head = 0;
        }
        ures = (head == tail);
        return ideiglenestarolo;
    }
}
```

4.11. ábra. Statikus sor kód 3. rész

A legutolsó függvény a `First`, amely a `head`-edik pozícióban található értékkel tér vissza.

4.2.3. Dinamikus sor reprezentáció

Dinamikus reprezentáció esetén a veremhez hasonlóan nem tömbben tároljuk az értékeket, hanem erre a célra speciálisan létrehozott Elem osztályokban, amik egy láncolatnak lesznek a csomópontjai. Az Elem osztály tartalmaz egy értéket, valamint egy referenciát egy következő elemre, így lehet majd tárolni, hogy „mi van a sorban egy érték után, ki következik”. A *head* referencia tárolja a sor elejét, ami ha nem mutat érvényes elemre, akkor

```
public int First()
{
    if (isEmpty())
    {
        System.err.println("Üres - nem lehet megnézni az elsőt");
        return -1;
    }
    else
    {
        return elemek[head];
    }
}
```

4.12. ábra. Statikus sor kód 4. rész

üres a sor. A `tail` referencia mutatja a sor végét, ahova az új elemek fognak kerülni.

Implementáció dinamikusan

A kód eleje a dinamikus veremhez képest mindösszesen a `tail` mező deklarációjával egészült ki. A dinamikus sor belső osztálya a `Node`. Tartalmazza egy aktuális csomópont értékét, valamint egy referenciát a következő `Node`-ra. A dinamikus sorban egy `head` nevű mező jelzi a sor elejét. A `head` egy `Node` típusú referencia, ami a verem létrehozáskor **`null`**, azaz nem referál sehova sem. Ez lesz egyben az ürességség feltétele is. Akkor és csak akkor üres a verem ha `head` egy **`null`** referencia. Szintén referencia a `tail`, ami a sor másik végét jelenti, ahova az új elemek érkezéskor bekerülnek. (Természetesen üres sor esetén a `tail` is **`null`**.

A sorban új elem betételekor az első feladat az új csomópont példányosítása. Függetlenül a sor korábbi állapotától egy új csomópontot senki sem fogja követni és az értéke a beteendő érték. Amennyiben a sor üres volt a betevés előtt a `head` referencia is az újonnan betett most egyetlen csomópontra kell, hogy mutasson. Ellenkező esetben ezt nem szabad megtenni, azonban helyette fűzni kell a meglévő láncot. Azaz a `tail` által mutatott csomópont következőjeként kell megtenni az új elemet. Utolsó lépésként a beszúrt elemet a `tail`-be tesszük, mivel az a vége a sornak.

Az első elem kivétele esetén megjegyezzük a visszatérési értéket, majd a `head`-et léptetjük, aminek az lesz a következménye, hogy az eddigi `head` csomópontra vonatkozó referenciánkat elveszítjük és a következő elem lesz a sor eleje. A sor kiürülése esetén a `tail`-t is **`null`**ra kell állítani, mivel az a specifikációnkban szerepelt. (A `head` automatikusn **`null`** lesz, mivel az utolsó csomópont rákövetkezője a csomópont példányosítása során **`null`** automatikusan, valamint az utolsó csomópontot biztosan nem követi semmi sem, tehát a következő csomópontot jelző mezőt nem állítottuk el.) Legvégül visszatér a függvény az elmentett értékkel.

A `First` függvény a `head` referencia által mutatott csomópontban tárolt értékkel tér vissza.

4.3. Lista, Láncolt Lista

4.3.1. Szekvenciális adatszerkezet

A szekvenciális adatszerkezet olyan $\langle A, R \rangle$ rendezett pár, amelynél az R reláció tranzitív lezártja teljes rendezési reláció. A szekvenciális adatszerkezetben az egyes adatelemek

```java
package sor;

public class SorDinamikus
{
    class Node
    {
        public int ertek;
        public Node kovetkezo;

        public Node(int ertek, Node kovetkezo)
        {
            this.ertek = ertek;
            this.kovetkezo = kovetkezo;
        }
    }

    private Node head;
    private Node tail;

    public SorDinamikus()
    {
        head = null;
        tail = null;
    }

    public boolean isEmpty()
    {
        return (head == null);
    }
```

4.13. ábra. Dinamikus sor kód 1. rész

```java
public void In(int elem)
{
    Node ujNode = new Node(elem, null);
    // Eredetileg üres a sor
    if (tail == null)
    {
        // A fej is az új elemre fog mutatni
        head = ujNode;
    }
    else
    {
        // Lánc fűzése
        tail.kovetkezo = ujNode;
    }
    tail = ujNode;
}
```

4.14. ábra. Dinamikus sor kód 2. rész

egymás után helyezkednek el. Az adatok között egy-egy jellegű a kapcsolat: minden adatelem csak egy helyről érhető el, és az adott elemről csak egy másik látható. Két kitüntetett elem az első és az utolsó.

Tipikus és legegyszerűbb példa a lista, ahol gondolhatunk egy tennivaló listára, amelynek tételei vannak, felvehetünk és törölhetünk tetszőlegeset közülük.

```
public int Out()
{
    if (isEmpty())
    {
        System.err.println("Üres a sor - hiba");
        return -1;
    }
    else
    {
        // Megjegyezzük a visszatérési értéket
        int visszaterni = head.ertek;
        // Megváltoztatjuk a head értékét a következőre -> kivesszük
        head = head.kovetkezo;
        // Ha kiürül
        if (head == null)
        {
            tail = head;
        }
        return visszaterni;
    }
}
```

4.15. ábra. Dinamikus sor kód 3. rész

```
public int First()
{
    if (isEmpty())
    {
        System.err.println("Üres a sor - hiba");
        return -1;
    }
    else
    {
        return head.ertek;
    }
}
```

4.16. ábra. Dinamikus sor kód 4. rész

4.3.2. Lista adatszerkezet

Homogén adatszerkezet, azaz azonos típusú véges adatelemek sorozata. Lehetséges jelölése a $L = (a_1, a_2, \ldots a_n)$, amennyiben üres listáról beszélünk, úgy az elemszám nulla, $n = 0$, vagyis $L = ()$.

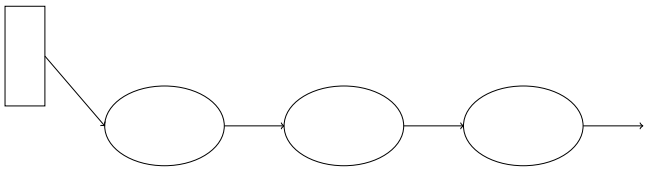
A láncolt lista egy olyan adatszerkezet, amelynek minden eleme tartalmaz egy (vagy több) mutatót (hivatkozást) egy másik, ugyanolyan típusú adatelemre, ami rákövetkezőséget jelenti a lista esetén. A lánc első elemének a címét a lista feje tartalmazza. A listafej nem tartalmaz információs részt, azaz tényleges listabeli adatot. A lánc végét az jelzi, hogy az utolsó elemben a rákövetkező elem mutatója üres.

Kétszeresen láncolt esetben vissza irányban is vannak hivatkozások, tehát a lista egy eleme mindkét szomszédjára vonatkozóan tartalmaz referenciát, továbbá nemcsak a fejelem, hanem a végelem is külön hivatkozással kerül eltárolásra.

4.3.3. Absztrakciós szint

Végiglépkedhetünk a lista elemein, beszúrhatunk és törölhetünk tetszés szerint. Az ábrán egy egyirányú láncolású lista található, ahol a téglalalp a listafej, az ellipszisek az értékes

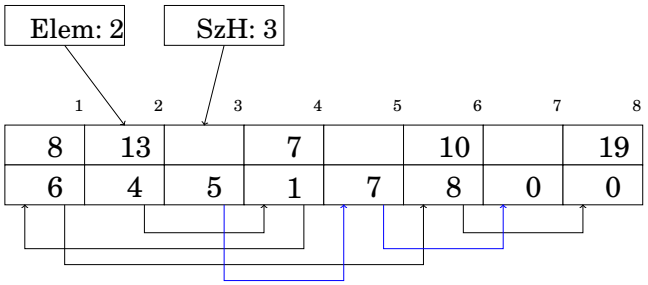
adatot is tartalmazó láncszemek.



4.17. ábra. Lista intuitív ADS/ADT

4.3.4. Statikus reprezentáció

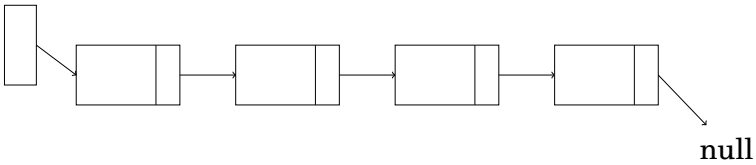
Statikus reprezentáció esetén egy táblázatot használunk, amiben érték, index párokat helyezünk el. Az indexek jelentik a rákövetkezőségeket, tehát ez fogja a lista elemei közötti ligokai sorrendet kialakítani. (A táblázatban elfogalalt pozíció és rákövetkezőség nem azonos a listában elfoglalt pozícióval és rákövetkezőséggel.) Tudjuk, hogy melyik az első értéket tartalmazó pozíció, valamint az első szabad helyet tartalmazó pozíció. A szabad helyekből is listát képezünk. Anna kaz elemnek amelyiknek nincs rákövetkezője, az a lista vége, illetve a szabad helyek listájának vége. Ennek a megoldásnak az az előnye, hogy a beszúrások és törlések esetén nem kell ügyelünk a lista táblázatbeli folytonosságára, így hatékonyabb (gyorsabb) és rendelkezésre álló memóriát maradéktalanul kihasználó megoldást kapunk.



4.18. ábra. Lista statikus reprezentáció

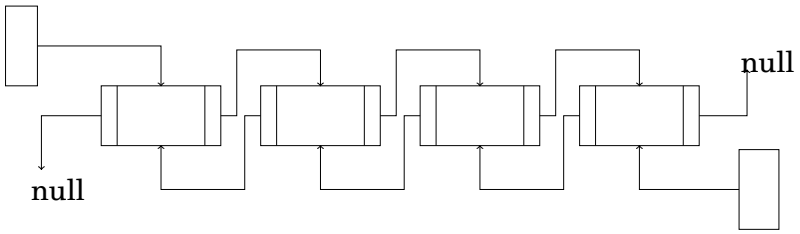
4.3.5. Dinamikus reprezentáció

Az elemek láncolását használjuk az első dinamikus reprezentációban, ami az egirányú láncolás esetén a sor esetén megismert módszerrel gyakorlatilag azonos. Minden elem referenciát tartalmaz a rákövetkezőjére. A kétirányú láncolás az alábbi ábra mutatja be.



4.19. ábra. Lista dinamikus reprezentáció – egyirányú láncolás

Az egirányú láncoláshoz képest a különbség az, hogy mindkét szomszédjára tartalmaz referenciát egy csomópont.



4.20. ábra. Lista dinamikus reprezentáció – kétirányú láncolás

4.3.6. Kétirányú láncolt lista megvalósítása

A lista állapotváltozói:

- `Head`: referencia az első elemre. Ami **`null`**, ha üres a lista.
- `Tail`: referencia az utolsó elemre. Ami **`null`**, ha üres a lista.
- `Akt`: egy kiválasztott elemre mutat, lehet léptetni előre és hátra. Amikor üres a lista akkor **`null`** az értéke.

Az `Akt` segítségével tudjuk a listában tárolt elemeket elérni, lekérdezni, megváltoztatni. Ezt a referenciát a megvalósított műveleteken keresztül fogjuk befolyásolni, a lista aktuálisan vizsgált elemét fogja jelenteni.

Műveletek

- `insertFirst(E)`: az E elemet beszúrja a lista elejére.
- `insertLast(E)`: az E elemet beszúrja a lista végére.
- `removeFirst()`: az első elemet törli a listából.
- `removeLast()`: az utolsó elemet törli a listából.
- `getAktValue()`: az aktuális elem lekérdezése.
- `setAktValue(E)`: az aktuális elem értékének megváltoztatása.
- `stepFirst()`: az aktuális elemet az elsőre lépteti.
- `stepLast()`: az aktuális elemet az utolsóra lépteti.
- `stepForward()`: az aktuális elemet Tail felé lépteti eggyel.
- `stepBackward()`: az aktuális elemet Head felé lépteti eggyel.
- `insertBefore(E)`: az E elemet beszúrja az aktuális elé.
- `insertAfter(E)`: az E elemet beszúrja az aktuális mögé.
- `removeAkt()`: az első elemet törli a listából.
- `isLast()`: lekérdezi, hogy az aktuális a lista végén van-e.
- `isFirst()`: lekérdezi, hogy az aktuális a lista elején van-e.
- `isEmpty`: lekérdezi, hogy üres-e a lista.

A műveletek leírása pseudókódban

A konstruktor, ami létrehoz egy üres listát, az össze referencia értékét átállítja **`null`**-ra, ami üres listát fog eredményezni.

Konstruktor
`Head`←**`null`**
`Tail`←**`null`**

```
Akt←null
```

Akkor üres a lista, ha fej és vég referenciák **null** értékűek.

```
isEmpty()
return Head==Tail==null
```

Az aktuálist jelentő refrencia értékére vonatkozó lekérdezések.

```
isLast()
return Akt==Tail
```

```
isFirst()
return Akt==Head
```

Az aktuális referencia által mutatott listaelem értékének lekérdezése és beállítása.

```
getAkt()
HA Akt≠null AKKOR return Akt.Ertek
```

```
setAkt(ujertek)
HA Akt≠null AKKOR Akt.Ertek←ujertek
```

Az aktuális referencia léptetésének műveletei:

```
stepForward()
HA Akt≠null ÉS ¬isLast() AKKOR Akt←Akt.Kovetkezo
```

```
stepBackward()
HA Akt≠null ÉS ¬isFirst() AKKOR Akt←Akt.Elozo
```

```
stepLast()
Akt←Tail
```

```
stepFirst()
Akt←Head
```

A beszúrási műveleteket fokozatosan építjük fel, az egyes eseteket visszavezetve korábbi esetekre. Kezdjük azzal a függvénnyel, ami a lista elejére szúr be egy új adatot. A beszúrás esetén mindenképpen létre kell hozni egy új csomópontot, aminek az értékét be kell állítani. Mivel első elemként kerül beszúrásra ezért a megelőzője biztosan a **null**, a rékövetkezője pedig az addigi `Head`. Azonban ha eredetileg üres volt a lista akkor a `Head` és a `Tail` értékét kell az új csomópontra állítani, míg ha már tartalmazott már (legalább) egy elemet, akkor a korábban első elemként tárol csomópont megelőzőjeként kell beállítani

az aktuálisan beszúrtat, továbbá az újonnan beszúrt lesz a `Head`.

---

insertFirst(ertek)

```
Akt←ujCsomopont←ÚJ Node
ujCsomopont.Ertek←ertek
ujCsomopont.Elozo←null
ujCsomopont.Kovetkezo←Head
HA isEmpty()
AKKOR
Head←Tail←ujCsomopont
KÜLÖNBEN
Head.Elozo←ujCsomopont
Head←ujCsomopont
```

Az utolsóként való beszúrást teljesen hasonlóan lehet megvalósítani, mint az elsőként való beszúrást, azonban az üres listába való beszúrást az `insertFirst()` függvénnyel oldjuk meg. (Itt a következőség és a megelőzőség felcserélődik az előző függvényhez képest.)

---

insertLast(ertek)

```
HA isEmpty() AKKOR insertFirst(ertek)
KÜLÖNBEN
Akt←ujCsomopont←ÚJ Node
ujCsomopont.Ertek←ertek
ujCsomopont.Elozo←Tail
ujCsomopont.Kovetkezo←null
Tail.Kovetkezo←ujCsomopont
Tail←ujCsomopont
```

Aktuális elem elé való beszúrás esetén, amennyiben az aktuális az első, vagy üres a lista visszavezetjük az első beszúró függvényünkre. Az új csomópont létrehozása során be kell állítanunk az adatot, az új csomópont megelőző és következő csomópontját. A következője az aktuális maga, hiszen az aktuális elé szúrunk be. A megelőző az aktuális megelőzője. Az aktuálist megelőző csomópont rákövetkezője lesz az újonnan létrehozott, valamint az aktuálist megelőzőnek is be kell állítani az új csomópontot. Végül az új csomópontot tesszük meg aktuálisnak.

---

insertBefore(ertek)

```
HA isEmpty() VAGY isFirst() AKKOR insertFirst(ertek)
KÜLÖNBEN
ujCsomopont←ÚJ Node
ujCsomopont.Ertek←ertek
ujCsomopont.Elozo←Akt.Elozo
ujCsomopont.Kovetkezo←Akt
Akt.Elozo.Kovetkezo←ujCsomopont
Akt.Elozo←ujCsomopont
Akt←ujCsomopont
```

A rákövetkezőként való beszúrást megpróbáljuk visszavezetni vagy üres listában való

beszúrásra, vagy utolsónak való beszúrásra. Ha egyik sem sikerül, akkor viszont biztosan tartalmaz annyi elemet a lista hogy meg tudjuk tenni azt, hogy léptetünk előre és megelőzőként szúrjuk be így ekvivalens megoldást kapva. (Ez természetesen nem a leghatékonyabb, azonban ez a legegyszerűbb.)

---

insertAfter(ertek)
HA isEmpty() VAGY isLast() AKKOR insertLast(ertek)
KÜLÖNBEN
stepForward()
insertBefore(ertek)

A törlések esetén hasonlóan eseteket vizsgálunk. Elsőként a megfelelő beszúrás párjaként az első elem kitörlését vizsgáljuk. Ilyenkor első feladat az aktuális elemre mutató referencia léptetése. (Gondoljuk meg, hogy ez minden esetben működik-e. Mi történik, ha az utolsó elemet töröljük a listából?) A `Head` referencia léptetése után, ha azt tapasztaljuk, hogy a `Head` referencia értéke **`null`**, akkor a `Tail`t is **`null`**ra állítjuk, hiszen kiürül a lista. Ellenkező esetben a `Head` által mutatott csomópont megelőzőjét állítjuk **`null`**ra, hiszen annak már nincs tényleges megelőzője.

---

removeFirst()
HA ¬isEmpty() AKKOR
HA isFirst() AKKOR Akt←Head.Kovetkezo
Head←Head.Kovetkezo
HA Head≠**null** AKKOR
Head.Elozo←**null**
KÜLÖNBEN
Tail←**null**

Az utolsó elem törlése. Amennyiben ez az egyetlen elem a listában visszavezetjük az előző esetre. Ellenkező esetben simán elvégezzük a törlést, nem kell törődnünk azzal, hogy kiürül a lista, tehát csak az utolsót megelőző elem rákövetkezőkét állítjuk **`null`**ra, illetve a `Tail` léptetjük eggyel visszafelé.

---

removeLast()
HA ¬isEmpty() AKKOR
HA Tail==Head AKKOR removeFirst(); VÉGE
HA isLast() AKKOR Akt←Tail.Elozo
Tail←Tail.Elozo
Tail.Kovetkezo←**null**

Aktuális elem törlése, amennyiben kiürülne a lista, vagy visszavezethető a korábbi függvényekre meghívjuk azokat a törlő függvényeket. Ha ez nem lehetséges akkor a törlendő elem biztosan közbülső elem, csakis a megelőző csomópont rákövetkezőjét és a rákövetkező csomópont megelőzőjét kell rákövetkezőre és a megelőzőre állítani. (Azaz a megelőző rákövetkezője az aktuális rákövetkezője, illetve a rákövetkező megelőzője az aktuális megelőzője kell, hogy legyen, ahhoz hogy az aktuális elemet kiszedjük a listából.) Továbbá az aktuális elemet kell egy listabeli elemre állítani.

```
removeAkt()
HA ¬isEmpty() AKKOR
HA isFirst() AKKOR removeFirst(); VÉGE
HA isLast() AKKOR removeLast(); VÉGE
Akt.Elozo.Kovektkezo←Akt.Kovektezo
Akt.Kovetkezo.Elozo←Akt.Elozo
Akt←Akt.Kovetkezo
```

JAVA kódok

Az alábbiakban a pseudó kódnak megfelelő Java kódok kerülnek ismertetésre.

```
package lista;

public class LancoltLista
{

    private class Csomopont
    {
        int Ertek;

        Csomopont Kovetkezo;
        Csomopont Elozo;

        public Csomopont(int value, Csomopont Next, Csomopont Prev)
        {
            Ertek = value;
            Kovetkezo = Next;
            Elozo = Prev;
        }
    }

    private Csomopont Head;
    private Csomopont Tail;
    private Csomopont Akt;

    public LancoltLista()
    {
        Head = null;
        Tail = null;
        Akt = null;
    }
```

4.21. ábra. Láncolt lista kód 1. rész

4.4. Fa

Ebben a részfejezetben a hierarchikus adatszerkezetek egyikével a fával, azon belül is a bináris keresési fával, majd a kupac adatszerkezettel ismerkedünk meg.

4.4.1. Hierarchikus adatszerkezetek

Definíció szerint a hierarchikus adatszerkezetek olyan $\langle A, R \rangle$ rendezett pár, amelynél van egy kitüntetett r elem, ez a gyökérelem, úgy, hogy:

```
public boolean isEmpty()
{
    return (Head == null);
}

public boolean isFirst()
{
    return (Akt == Head);
}

public boolean isLast()
{
    return (Akt == Tail);
}
```

```
public int getAkt()
{
    if (!isEmpty())
    {
        return Akt.Ertek;
    }
    else
    {
        return -1;
    }
}

public void setAkt(int ujertek)
{
    if (!isEmpty())
    {
        Akt.Ertek = ujertek;
    }
}
```

4.22. ábra. Láncolt lista kód 2. rész

```
public void stepBackward()
{
    if (!isEmpty() && !isFirst())
    {
        Akt = Akt.Elozo;
    }
}

public void stepForward()
{
    if (!isEmpty() && !isLast())
    {
        Akt = Akt.Kovetkezo;
    }
}
```

```
public void stepFirst()
{
    Akt = Head;
}

public void stepLast()
{
    Akt = Tail;
}
```

4.23. ábra. Láncolt lista kód 3. rész

- r nem lehet végpont
- $\forall a \in A \setminus \{r\}$ elem egyszer és csak egyszer végpont
- $\forall a \in A \setminus \{r\}$ elem r-ből elérhető

Az adatelemek között egy-sok jellegű kapcsolat áll fenn. Minden adatelem csak egy helyről érhető el, de egy adott elemből akárhány adatelem látható. A hierarchikus adatszerkezetek valamilyen értelemben a szekvenciális adatszerkezetek általánosításai. (Az elérhetőség ebben az értelemben rákövetkezőségek sorozatát jelenti, valamint a végpont egy $a \rightarrow b$ jellegű kapcsolat esetén a b értéket jelöli, a kezdőpont pedig az a értéket.)

4.4.2. Fa adatszerkezet

A fa egy hierarchikus adatszerkezet, mely véges számú csomópontból áll, és két csomópont között a kapcsolat egyirányú, az egyik a kezdőpont, a másik a végpont, valamint van

```
public void insertFirst(int ertek)
{
    Csomopont ujCsomopont = new Csomopont(ertek, Head, null);
    Akt = ujCsomopont;
    if (isEmpty())
    {
        Head = ujCsomopont;
        Tail = ujCsomopont;
    }
    else
    {
        Head.Elozo = ujCsomopont;
        Head = ujCsomopont;
    }
}

public void insertLast(int ertek)
{
    if (isEmpty())
    {
        insertFirst(ertek);
    }
    else
    {
        Csomopont ujCsomopont = new Csomopont(ertek, null, Tail);
        Akt = ujCsomopont;
        Tail.Kovetkezo = ujCsomopont;
        Tail = ujCsomopont;
    }
}
```

4.24. ábra. Láncolt lista kód 4. rész

a fának egy kitüntetett csomópontja, ami nem lehet végpont, ami a fa gyökere. Ezen kívül az összes többi csomópont pontosan egyszer végpont. (Végpont és kezdőpont itt a rákövetkezőségi kapcsolatnál a rákövetkezőséget jelölő nyílra vonatkozik. Eszerint csak a gyökér nem rákövetkezője semminek sem.)

Az előző definíció leírható egy rekurzióval is, azaz a fa definiálása során felhasználjuk a fa definícióját.

- A fa vagy üres, vagy
- Van egy kitüntetett csomópontja, ez a gyökér.
- A gyökérhez 0 vagy több diszjunkt fa kapcsolódik. Ezek a gyökérhez tartozó részfák.

A fákkal kapcsolatos algoritmusok többsége rekurzív.

Elnevezések és további definíciók

- A fa *csúcsai* az adatelemeknek felelnek meg.
- Az *élek* az adatelemek egymás utáni sorrendjét határozzák meg – egy csomópontból az azt követőbe húzott vonal egy él.
- A *gyökérelem* a fa első eleme, amelynek nincs megelőzője, az egyetlen csomópont amibe nincs befutó él.
- *Levélelem* a fa azon eleme, amelynek nincs rákövetkezője, belőle nem fut ki él.
- *Közbenső elem* az összes többi adatelem, ami nem gyökér és nem levél.

```
public void insertBefore(int ertek)
{
    if (isEmpty() || isFirst())
    {
        insertFirst(ertek);
    }
    else
    {
        Csomopont ujCsomopont = new Csomopont(ertek, Akt, Akt.Elozo);
        Akt.Elozo.Kovetkezo = ujCsomopont;
        Akt.Elozo = ujCsomopont;
        Akt = ujCsomopont;
    }
}

public void insertAfter(int ertek)
{
    if (isEmpty() || isLast())
    {
        insertLast(ertek);
    }
    else
    {
        stepForward();
        insertBefore(ertek);
    }
}
```

4.25. ábra. Láncolt lista kód 5. rész

```
public void removeFirst()
{
    if (!isEmpty())
    {
        if (isFirst())
            Akt = Head.Kovetkezo;
        Head = Head.Kovetkezo;
        if (Head!=null)
        {
            Head.Elozo = null;
        }
        else
        {
            Tail = null;
        }
    }
}
```

```
public void removeLast()
{
    if (!isEmpty())
    {
        if (Tail == Head)
        {
            removeFirst();
            return;
        }
        if (isLast())
            Akt = Tail.Elozo;

        Tail = Tail.Elozo;
        Tail.Kovetkezo = null;
    }
}
```

4.26. ábra. Láncolt lista kód 6. rész

- Minden közbenső elem egy *részfa* gyökereként tekinthető, így a fa részfákra bontható: *részfa*: „t” részfája „a”-nak, ha az „a” gyökere, azaz közvetlen megelőző eleme „t”-nek, vagy „t” részfája „a” valamely részfájának.
- *Elágazásszám*: közvetlen részfák száma, azt mondja meg, hogy egy adott csomópontból hány él indul ki.
- A fa *szintje* a gyökértől való távolságot mutatja.

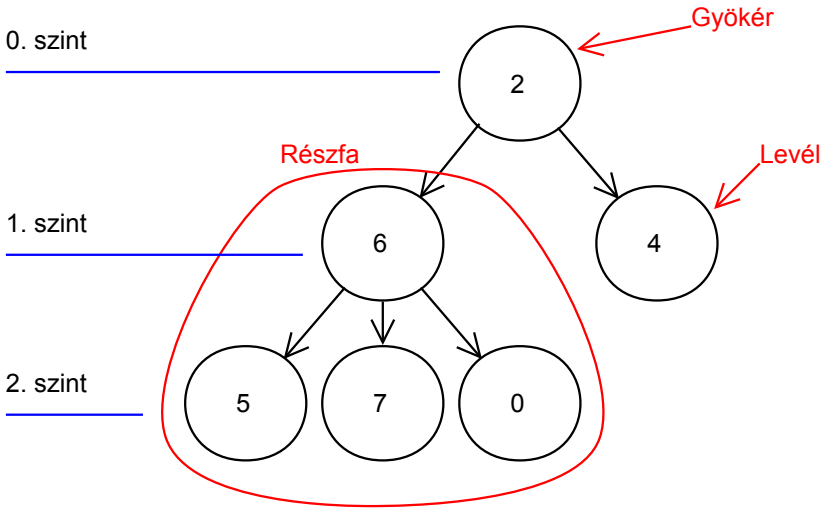
```
public void removeAkt()
{
    if (!isEmpty())
    {
        if (isFirst())
        {
            removeFirst();
            return;
        }
        if (isLast())
        {
            removeLast();
            return;
        }
        Akt.Elozo.Kovetkezo = Akt.Kovetkezo;
        Akt.Kovetkezo.Elozo = Akt.Elozo;
        Akt = Akt.Kovetkezo;
    }
}
```

4.27. ábra. Láncolt lista kód 7. rész

  - A gyökérelem a 0. szinten van.
  - A gyökérelem rákövetkezői az 1. szinten, a rákövetkezők pedig a 2. szinten. ...

- A fa szintjeinek száma a fa *magassága*, azaz a legnagyobb számú szint ha 5, akkor a fa magassága 6.
- *Csomópont foka*: a csomóponthoz kapcsolt részfák száma, azt mutatja meg ez a szám, hogy hány él indul ki az adott csomópontból.
- *Fa foka*: a fában található legnagyobb fokszám.
- *Levél*: 0 fokú csomópont, nincs belőle kimenő él.
- *Elágazás* (közbenső vagy átmenő csomópont): $\neq 0$ fokú csomópont.
- *Szülő (ős)*: kapcsolat (él) kezdőpontja (csak a levelek nem szülők).
- *Gyerek (leszármazott)*: kapcsolat (él) végpontja (csak a gyökér nem gyerek) Ugyanazon csomópont leszármazottai egymásnak testvérei. (Hasonlatosan a családfában megszokott módon.)
- *Szintszám*: gyökértől mért távolság. A gyökér szintszáma 0. Ha egy csomópont szintszáma n, akkor a hozzá kapcsolódó csomópontok szintszáma $n + 1$.
- *Útvonal*: az egymást követő élek sorozata. Minden levélelem a gyökértől pontosan egy úton érhető el.
- *Ág*: az az útvonal, amely levélben végződik.
- *Üresfa* az a fa, amelyiknek egyetlen eleme sincs.
- *Fa magassága*: a levelekhez vezető utak közül a leghosszabb. Mindig eggyel nagyobb, mint a legnagyobb szintszám.
- *Minimális magasság*ú az a fa, amelynek a magassága az adott elemszám és fafokszám esetén a lehető legkisebb. (Valójában ilyenkor minden szintre a maximális elemszámú elemet építjük be.)
- Egy fát *kiegyensúlyozottnak* nevezünk, ha csomópontjai azonos fokúak, és minden szintjén az egyes részfák magassága nem ingadozik többet egy szintnél. Például egy kétfokú fa esetén a bal részfa és a jonn részfa magassága legfeljebb egyel tér el egymástól tetszóleges csomópont esetén.
- *Rendezett fa*: ha az egy szülőhöz tartozó részfák sorrendje lényeges, azok rendezet-

tek. Ilyenkor valamilyen szabály határozza meg azt, hogy melyik részfában milyen elemek helyezkedhetnek el.



4.28. ábra. Fa elnevezések

4.4.3. Bináris fa

A bináris fa olyan fa, amelynek minde csúcspontjából maximum 2 részfa nyílik (azaz a fa fokszáma 2). Ebből kifolyólag egy szülő mindig a gyerekek között (és fölött) helyezkedik el. (Ennek a bejárásoknál lesz szerepe.) Egy bináris fa akkor *tökéletesen kiegyensúlyozott*, ha minden elem bal-, illetve jobboldali részfájában az elemek száma legfeljebb eggyel tér el.

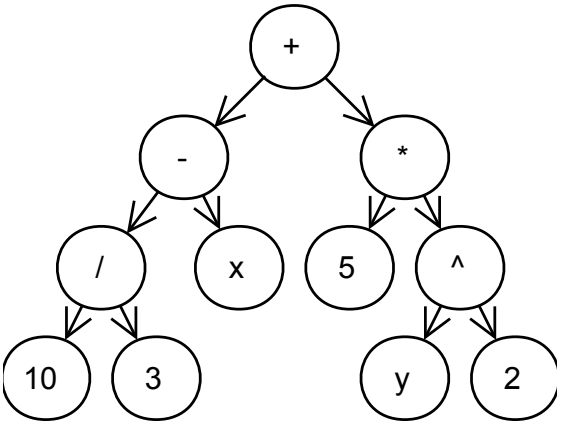
Teljesnek nevezünk egy bináris fát, ha minden közbenső elemének pontosan két leágazása van és *majdnem teljes*: ha csak a levelek szintjén van esetleg hiány. (Tehát ha lerajzoljuk, akkor a jobb szélén a legutolsó szinten hiányzik néhány levél.

Speciális bináris fák

Kiszámítási- vagy kifejezésfa. Korábban foglalkoztunk kifejezésekkel. Minden kifejezés a kiértékeléshez szétbontható részkifejezésekre, és annak megfelelően összetevőkre. (Operátorok és operandusok.) Ezt egy fában is ábrázolni lehet, ahol

- Az a struktúra, amely egy nyelv szimbólumai és különböző műveletei közötti precedenciát jeleníti meg.
- Aritmetikai kifejezések ábrázolására használják.
- Minden elágazási pont valamilyen operátort,
- A levélelemek operandusokat tartalmaznak.
- A részfák közötti hierarchia fejezi ki az operátorok precedenciáját, illetve a zárójelezést.

A $((10/3) - x) + (5 * y^2)$ kifejezés fája:



4.29. ábra. Kifejezésfa példa

Fa műveletek

Lekérdező műveletek:

- Üres-e a fa struktúra.
- Gyökérelem értékének lekérdezése.
- Meghatározott elem megkeresése, az arra vonatkozó referencia visszaadása.
- A megtalált tetszőleges elem értékének lekérdezése.

Módosító műveletek:

- Üres fa létrehozása – konstruktor.
- Új elem beszúrása.
- Meghatározott elem kitörlése.
- Összes elem törlése.
- Egy részfa törlése.
- Részfák kicserélése egymással.
- Gyökér megváltoztatása.
- Egy meghatározott elem értékének megváltoztatása.

Fa bejárások

A bejárási algoritmusok egy tetszőleges fa összes csomópontján végiglépkednek egy meghatározott módszer szerint. Rekurziós módszerek tetszőleges fa esetén:

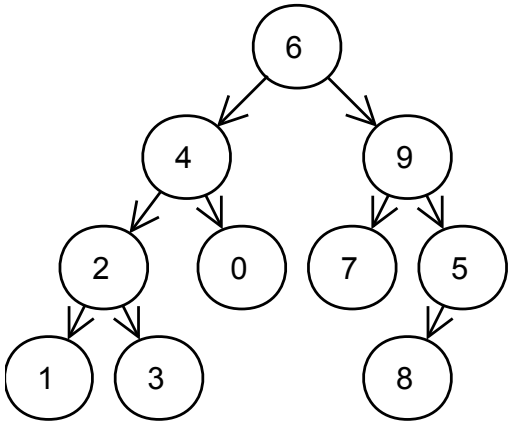
- Pre-order bejárás. (Először a gyökér kiírása (érintése) majd a részfák ugyanilyen módszerű bejárása)
- Post-oreder bejárás. (Először a részfák bejárása – bal, majd jobb –, majd legvégül a gyökér érintése)

Bináris fák esetén még egy bejárási módszer:

- In-order bejárás (Először a balgyerek bejárása, majd a gyökér érintése, azután a jobbgyerek bejárása)

A bejárások esetén az előző algoritmusok „receptek”. Egy aktuális csomópontban a recept meghatározza, hogy mi történik. Például inorder esetben a teljes bal részfára alkalmazzuk először a receptet, majd kiírjuk az aktuális csomópont értékét, majd folytatjuk a jobb részfával.

Például tekintsük az alábbi részfát:



4.30. ábra. Példa fa

Preorder
6, 4, 2, 1, 3, 0, 9, 7, 5, 8

Postorder
1, 3, 2, 0, 4, 7, 8, 5, 9, 6

Mivel bináris fa volt a példa ezért lehetséges az inorder bejárás is:

Inorder
1, 2, 3, 4, 0, 6, 7, 9, 8, 5

4.4.4. Fa reprezentációs módszerek

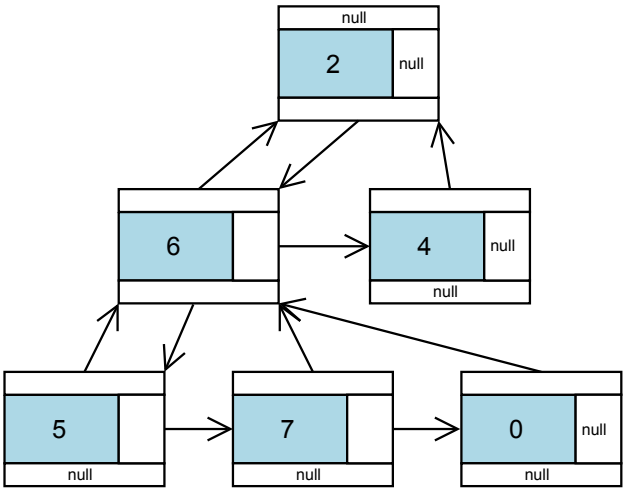
Röviden áttekintünk néhány a fa reprezentálása alkalmas módszert.

Balgyerek-jobbtestvér

Minden csomópont ismeri a szülőjét, egyetlen (legbaloldalibb) gyermekét és a közvetlen jobbtestvért. Ezzel lehetséges, hogy bármely csomópontnak tetszőleges számú gyereke legyen, amik gyakorlatilag egy láncolt listát alkotnak. Ezeket referenciák segítségével írhatjuk le.

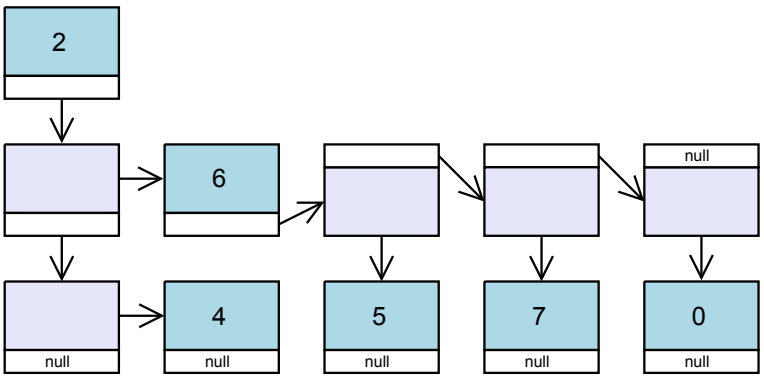
Multilistás ábrázolás

Minden csomópont egy láncolt lista. A lista első eleme tartalmazza az adatot, a többi csomópont már csak hivatkozásokat a leszármazottakra (gyermekcsomópontokra). Ennek



4.31. ábra. Balgyerek-jobbtestvér reprezentáció

megfelelően kétféle csomópont található, a listák fajtája szerint.



4.32. ábra. Multilistás ábrázolás reprezentáció

Aritmetikai reprezentáció

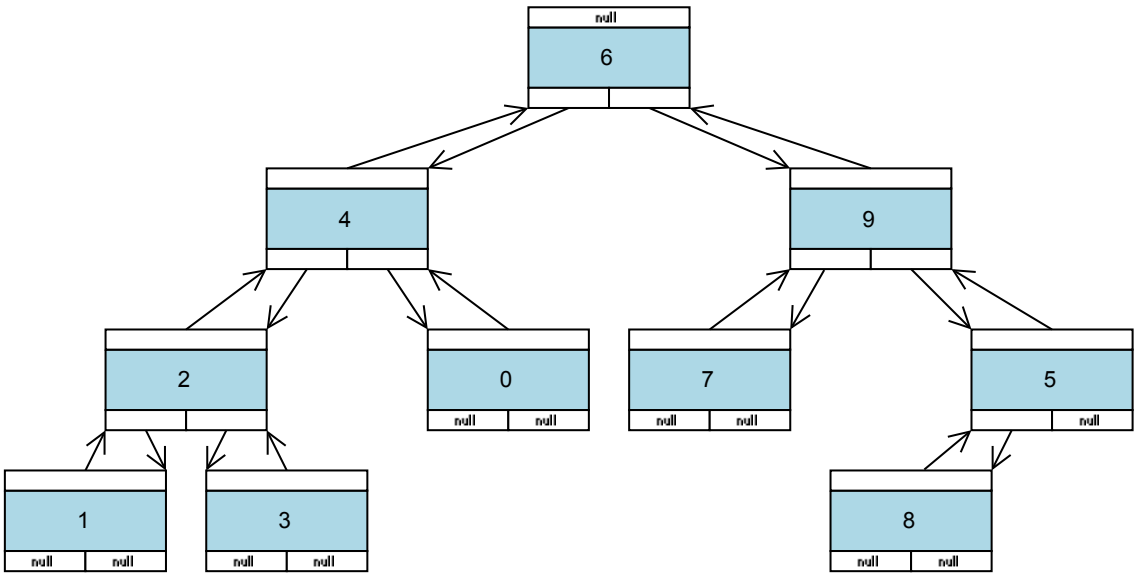
Amennyiben meghatározunk egy felső korlátot a csúcsok fokszámára, úgy lehetséges az alábbi reprezentációval a fát tárolnunk. Veszünk egy tömböt, amibe sorfolytonosan és így szintfolytonosan beleírjuk az egyes szinteken található értékeket. A korábbiakban bemutatott bináris fa (ami esetén a fokszám korlát kettő) aritmetikai ábrázolásban: (A teljes fához képest hiányzó értékek helyét kihagyjuk, hogy a későbbi beszúrás esetén rendelkezésre álljon a beszúrandó elemnek a hely.)

6	4	9	2	0	7	5	1	3	-	-	-	-	8	-

4.33. ábra. Aritmetikai ábrázolás

Láncolt ábrázolás

Szintén korlátos fokszám esetén használható, a továbbiakban ezt fogjuk a programok során alkalmazni. Minden csomópont ismeri a szülőjét, valamint a jobb és bal gyerekét, egy-egy referenciával hivatkozik a megfelelő csomópontokra. További referencia mutat a gyökérre. (Ez általánosítása a kétirányú láncolt listának, ahol a rákövetkező elem a két gyerek, a megelőző elem pedig a szülő.)



4.34. ábra. Láncolt ábrázolás

4.5. Bináris keresési fák

Az előző szakaszban a fák fogalmával ismerkedtünk meg, valamint a tudjuk, hogy egy bináris fa azt jelenti, hogy egy csomópontnak legfeljebb két gyereke lehet. Ezekhez tulajdonságokhoz hozzáadva a keresőfa tulajdonságot egy nagyon hasznos konstrukciót kapunk.

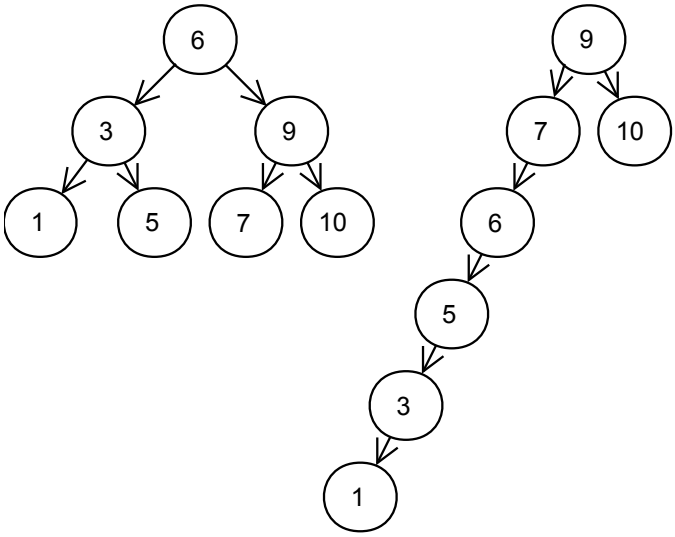
Mit is jelent a keresőfa:

- A rendezési fa (vagy keresőfa) olyan fa adatszerkezet, amelynek kialakítása a különböző adatelemek között meglévő rendezési relációt követi.
- A fa felépítése olyan, hogy minden csúcsra igaz az, (bináris esetben) hogy a csúcs értéke nagyobb, mint tetszőleges csúcsé a tőle balra lévő leszálló ágon és a csúcs értéke kisebb minden, a tőle jobbra lévő leszálló ágon található csúcs értékénél. (Részfa csúcsainál.)
- A T fa bármely x csúcsára és $bal(x)$ bármely y csúcsára és $jobb(x)$ bármely z csúcsára $y < x < z$

A rendezési fa az őt tartalmazó elemek beviteli sorrendjét is visszatükrözi. Ugyanazokból az elemekből különböző rendezési fák építhetők fel. Figyeljük meg a példákat: (A bezúrást úgy végezzük el, hogy mindig elindulunk a gyökérből és aszerint haladunk a jobbra, vagy balra, hogy a beszúrandó elem kisebb-e vagy nagyobb az aktuálisan vizsgáltnál. Amennyiben egy olyan helyre jutunk, ahol nincs részfa, akkor a beszúrandó elemet betesszük oda. Ellenkező esetben haladunk felefé tovább.)

Első sorrend
6,3,1,9,7,5,10

Második sorrend
9,7,6,5,10,3,1



4.35. ábra. Bináris keresőfa felépítése

4.5.1. Tulajdonságok

Inorder bejárással a kulcsok rendezett sorozatát kapjuk. Az algoritmus helyessége a bináris-kereső-fa tulajdonságból indukcióval adódik.
Egy n csúcsú bináris kereső fa bejárása $\mathcal{O}(n)$[1] ideig tart, mivel a kezdőhívás után a fa minden csúcspontja esetében pontosan kétszer (rekurzívan) meghívja önmagát, egyszer a baloldali részfára, egyszer a jobboldali részfára. (A rekurziós algoritmus átírható ciklusra is.)

4.5.2. Műveletek

Keresés. A T bináris keresési fában keressük a k kulcsú elemet (csúcsot). A keresés, ha létezik a keresett csúcs, akkor visszaadja az elem címét, egyébként `null`-t. Ennek az algoritmusnak z algoritmust megadjuk rekurzív és iteratív megoldásban is.

A keresés alapötlete, hogy elindulunk a egy csomópontból (gyökér) megvizsgáljuk, hogy megtaláltuk-e az keresett értéket, vagy kimentünk-e a fából. (Levél gyereke mindig `null`.) Ha egyik sem, akkor eldöntjük a kulcs alapján, hogy a fában merre tovább. Ha a keresett érték kisebb, mint az aktuális csomópont, akkor balra, különben jobbra haladunk tovább.

`Fában-keres(x, k)` – rekurzív

[1] A pontos definícióját lásd a következő fejezetben.

```
HA x = null VAGY k = kulcs[x]
akkor return x
HA k < kulcs[x]
AKKOR RETURN Fában-keres(bal[x], k)
KÜLÖNBEN RETURN Fában-keres(jobb[x], k)
```

Fában-keres(x, k) – iteratív

```
CIKLUS AMÍG x ≠ NULL ÉS k ≠ kulcs[x]
HA k < kulcs[x]
AKKOR x ← bal[x]
KÜLÖNBEN x ← jobb[x]
return x
```

Minimum keresés. Tegyük fel, hogy T ≠`null`. Addig követjük a baloldali mutatókat, amíg NULL referenciát nem találunk. Ez gyakorlatban a legbaloldalibb elemet jelenti a fában, ami szükségszerűen a legkisebb is.

Fában-minimum (T) – iteratív

```
x ← gyökér[T]
CIKLUS AMÍG bal[x] ≠ null
x ← bal[x]
return x
```

Lefut $\mathcal{O}(h)$ idő alatt, ahol h a fa magassága. Hasonlóan megkereshető a maximum érték is, ami a legjobboldalibb elem.

4.6. Kupac (Heap)

A kupac adatszerkezet bevezetéséhez néhány fogalomra van szükség.

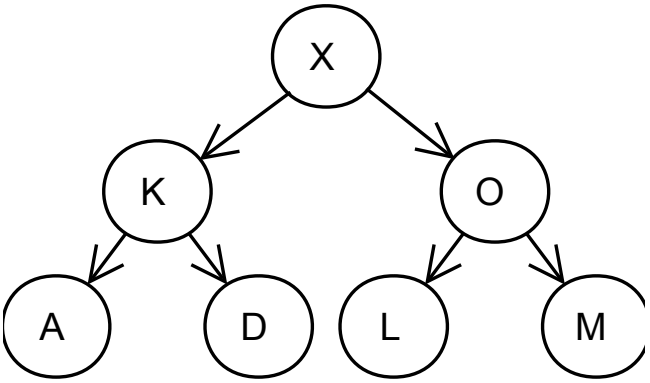
Egy bináris fa *teljes*, ha a magassága h, és $2^h - 1$ csomópontja van. Egy h magasságú bináris fa *majdnem teljes*, ha üres; vagy a magassága h, és a bal részfája $h - 1$ magas és *majdnem teljes* és jobb részfája $h - 2$ magas és *teljes*; vagy a magassága h, és a bal részfája $h - 1$ magas és *teljes* és jobb részfája $h - 1$ magas és *majdnem teljes*.

A gyakorlatban, amikor a fát felrajzoljuk a majdnem teljesség az jelenti, hogy a legutolsó szinten jobbról visszafelé hiányozhatnak értékek pont úgy, hogy ha elegendő érték lenne, az utolsó sor jobb szélén, akkor teljes lenne a fa. A majdnem teljes fákat balról „töltjük fel”. (Avagy szintenként haladunk a feltöltéssel és balról jobbra . . .)

4.6.1. Kupac tulajdonság

Egy majdnem teljes bináris fa heap (kupac) tulajdonságú, ha üres, vagy a gyökérben lévő kulcs nagyobb, mint mindkét gyerekében, és mindkét részfája is heap tulajdonságú. Nagyon fontos, hogy egy ez másik definíció a bal/jobb gyerek értékére vonatkozóan, a bináris keresési fához képest!

Reprezentálásuknál kihasználjuk a tömörítettséget és majdnem teljességet így aritmetikai reprezentációval tömbben tároljuk az értékeket és az egy indexfüggvény számítja a szülőt és a gyerekeket.

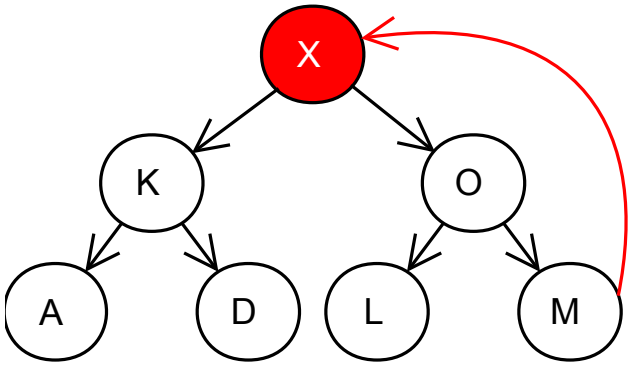


4.36. ábra. Kupac példa

4.6.2. Műveletek

Gyökér törlése

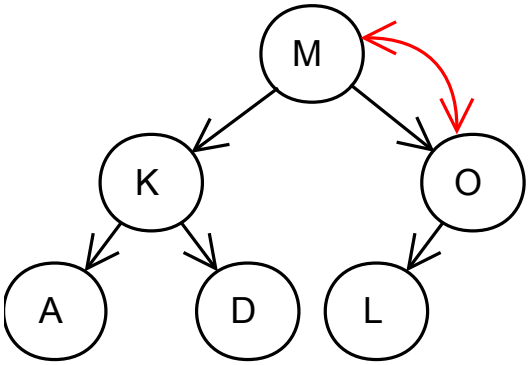
A gyökér a legnagyobb elem, ami a kupac tulajdonság betartásából következik. Eltávolítása után a legalsó szint legjobboldalibb elemét tesszük fel a helyére, hogy a majdnem teljes tulajdonság megmaradjon. Ezzel azonban elrontjuk kupac tulajdonságot, amit helyre kell állítani. A helyreállításhoz cseréljük fel a gyökeret a nagyobb gyerekével. Ezzel a lépéssel a nagyobb gyereket tartalmazó részfában rontottunk el a kupac tulajdonságot. (Nem feltétlenül romlott el.) Így ismételjük a kupac tulajdonság helyreállítását, amíg szükséges.



4.37. ábra. Gyökér törlése

Beszúrás

Amikor beszúrunk, tegyük a következő szabad pozícióra, a legalsó szint legjobboldalibb elemének tesszük fel a helyére, hogy a majdnem teljesség megmaradjon. Ezzel valószínűleg elrontjuk kupac tulajdonságot, amit helyre kell állítani, a törléshez hasonlóan. Cseréljük fel az újonnan beszúrtat a szülőjével. Ezt ismételjük egészen a fa tetejéig, vagy amíg szükséges.



4.38. ábra. Gyökér törlése

4.6.3. Indexfüggvények

Indexfüggvények aritmetikai reprezentáció esetén. A tömbben az indexfüggvények segítségével tudjuk megállapítani egy csomópont gyerekének indexét, illetve szülőjének indexét.

Balgyerek(k)
`RETURN 2k`

Jobbgyerek(k)
`RETURN 2k+1`

Szülő(k)
`RETURN ⌊k/2⌋`

Az indexek helyességének végiggondolását az olvasóra bízom. (Amennyiben lerajzoljuk a reprezentációt és a szerepeket, könnyen megoldásra jutunk.)

4.7. Használat

A kupacot meg lehet konstruálni fejjel lefelé is, amikor is a legkisebb elem van a kupac tetején.
A kupac például használható elemek rendezéséhez, elsőbbségi sor megvalósításához. Az elsőbbségi sor egy olyan sor, amikor nemcsak az utolsó pozícióba lehet bekerülni a sorban, hanem fontossági alapon előbbre is. (A fontosság az, ami alapján a kupacban meghatározzuk a pozíciót.) A kupac tömbös reprezentációját lineárisan ki lehet olvasni, ami megfeleltethető egy sornak.

5. fejezet

Algoritmusok

5.1. Algoritmusok műveletigénye

Korábban esett szó az algoritmusok hatékonyságáról, ebben a fejezetben három olyan definíciót vezetünk be, amivel összehasonlíthatóvá válnak az algoritmusok műveletigényei, hatékonysága. Két szempontot lehet figyelembe venni az egyik a lépésszám, vagyis az program által megkívánt lépések mennyisége, ami közvetlenül a futási időre van hatással. A másik az algoritmus által igényelt memória mérete. Mindkettőt a bemenő adatok méretével arányosan lehet vizsgálni, innentől fogva jelentse n a bemenet (input) méretét. A lépésszám ennek valamilyen függvény lesz $f(n)$.

A hatékonyság vizsgálatánál az $f(n)$-et vizsgáljuk. Azonban az összehasonlításnál az alábbi példákat vegyük figyelembe:

- $100n$ vagy $101n$, általában mindegy
- n^2 vagy n^3 már sokszor nagy különbség, de néha mindegy
- n^2 vagy 2^n már mindig nagy különbség

Ahhoz, hogy ezt matematikailag is kezelni tudjuk bevezetünk három fogalmat.

5.1.1. Függvények rendje

Ordó

Definíció – Ordó
Ha $f(x)$ és $g(x)$ az $\mathbf{R}^+$ egy részhalmazán értelmezett valós értékeket felvevő függvények, akkor $f = \mathcal{O}(g)$ jelöli azt a tényt, hogy vannak olyan $c, k > 0$ állandók, hogy $|f(x)| \leq c*|g(x)|$ teljesül, ha $x \geq k$.

Ekkor a g aszimptotikus felső korlátja f-nek; „f nagy ordó g”.[1]

Például
$100n + 300 = \mathcal{O}(n)$, hiszen $k = 300$; $c = 101$-re teljesülnek a feltételek.
$100n + 300 \leq 101n$, ha $n \geq 300$

Azt jelenti, hogy az f függvény egy meghatározott „idő” után alatta van biztosan a g függvény konstans-szorosának.

[1] Az ordo latin szó, jelentése rend.

Omega

Definíció – Omega
Ha $f(x)$ és $g(x)$ az $\mathbf{R}^+$ egy részhalmazán értelmezett valós értékeket felvevő függvények, akkor $f = \Omega(g)$ jelöli azt a tényt, hogy vannak olyan $c, k > 0$ állandók, hogy $|f(x)| \geq c*|g(x)|$ teljesül, ha $x \geq k$.

Ekkor a g aszimptotikus alsó korlátja f-nek.

Például
$100n - 300 = \Omega(n)$, hiszen $n \geq 300$; $c = 99$-re teljesülnek a feltételek.

Ez gyakorlatilag megfordítja az előző definícióban meghatározott szerepeket.

Theta

Definíció – Theta
Ha $f = \mathcal{O}(g)$ és $f = \Omega(g)$ is teljesül, akkor $f = \Theta(g)$.

Ekkor a g aszimptotikus éles korlátja f-nek.
Például
$100n - 300 = \Theta(n)$, az eddigiek alapján.

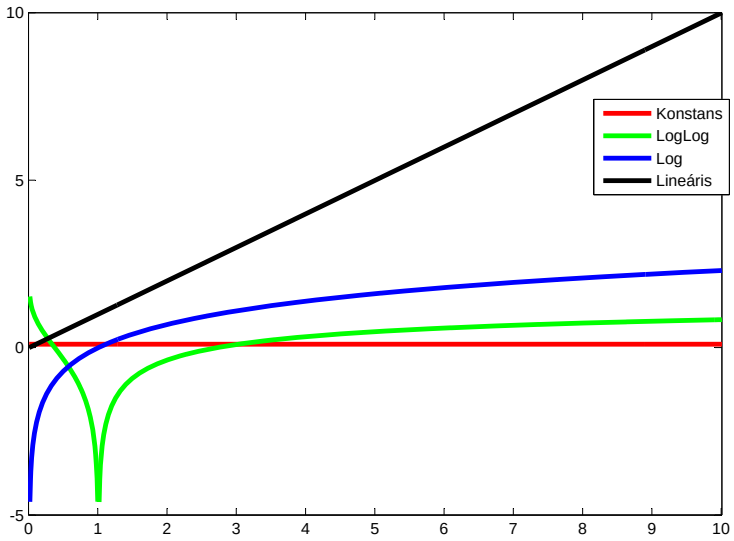
5.1.2. Sorrend

Az alább sorrend írható fel a rendek között, ahol is növekvő komplexitással kerültek sorban a függvények.

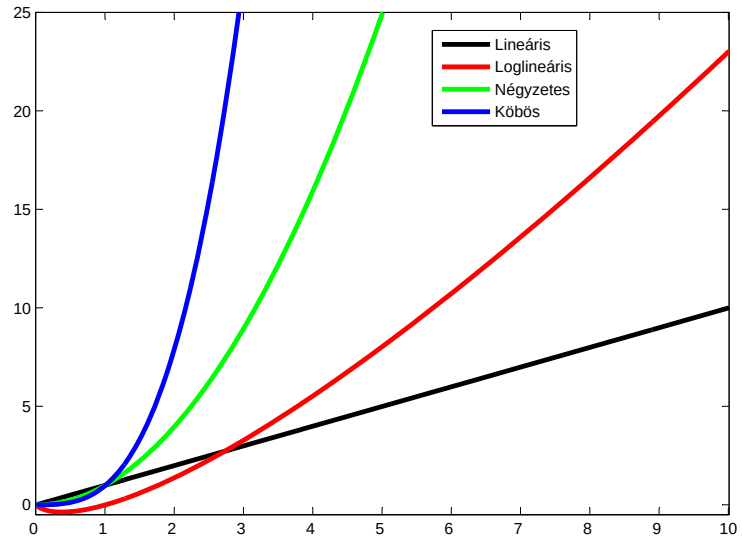
- Konstans – $\mathcal{O}(1)$
- Loglogaritmukus – $\mathcal{O}(\log \log n)$
- Logaritmikus – $\mathcal{O}(\log n)$
- Lineáris – $\mathcal{O}(n)$
- Linearitmikus (Loglineáris) – $\mathcal{O}(n \log n) = \mathcal{O}(\log n!)$
- Négyzetes – $\mathcal{O}(n^2)$
- Köbös – $\mathcal{O}(n^3)$
- Polinomiális (Algebrai) – $\mathcal{O}(n^c)$, ha $c > 1$
- Exponenciális (Geometriai) – $\mathcal{O}(c^n)$
- Faktoriális (Kombinatoriális) – $\mathcal{O}(n!)$

Mindez ábrázolva:

Időben nyilvánvalóan akkor lesz hatékony egy algoritmus, ha a sorrenben minél kisebb függvény rendjében függ a bemenet méretétől a feldolgozás ideje, vagyi a lépések száma. Sajnos azonban vannak olyan problémák, amelyeket nem tudunk hatékonyan megoldani, például lineáris vagy polinomiális időben. (Például létezik a problémák egy olyan osztálya amelyek „nehéz” feladatoknak számítanak és polinom időben egy megoldásjelölt helyessége dönthető el csupán. Ilyen egy szám prím felbontása is, amikor egy tetszőleges számot felírunk prímszámok szorzataként.)



5.1. ábra. Függvények rendje



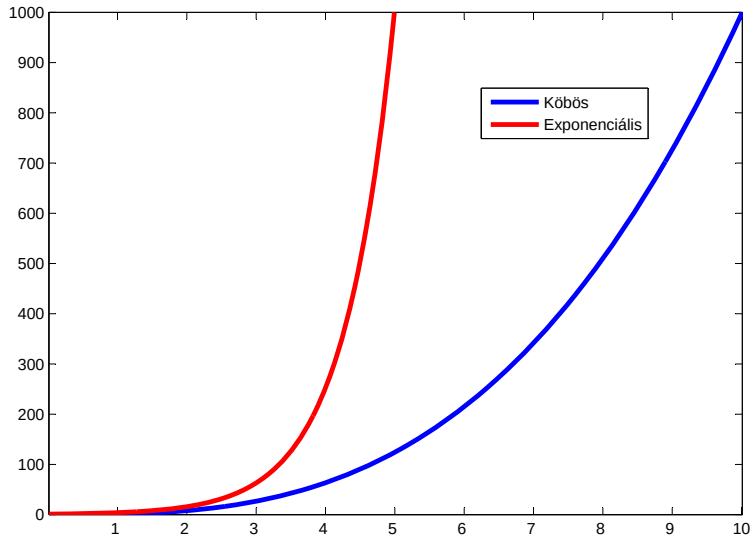
5.2. ábra. Függvények rendje

5.2. Lengyelforma

A lengyelforma egy speciális formája a kifejezések felírásának. Az eddigi megszokott formában az úgynevezett *infix* módosn írtuk fel a kifejezést. Az infix forma esetén a műveleti jel (operátor) a műveletben szereplő értékek (operandusok) között szerepel. A kifejezéseket a műveleti jel elhelyezésétől függően lehet még *postfix*, vagy *prefix* módon leírni. Prefix abban az esetben, ha a az operátor az operandusok előtt van, illetve postfix, amennyiben az operandusok mögött helyezkedik el az operátor.

Példa *infix* kifejezésre

$a * b + c$



5.3. ábra. Függvények rendje

Példa *prefix* kifejezésre

$ab * c+$

Példa *infix* kifejezésre

$+ * abc$

Hagyományos módon a matematikában az infix kifejezéseket használjuk. J. Lukasewitz lengyel matematikus használta először a postfix és prefix jelölés, ezért hívják lengyelformának. Helyes lengyelformát a számítógép sokkal könnyebben értékel ki, és egyszerűbb algoritmust lehet írni rá.

Első példa lengyelformára

$(a + b) * (c + d) \quad \Rightarrow \quad ab + cd + *$

Második példa lengyelformára

$(a + b * c) * (d * 3 - 4) \quad \Rightarrow \quad abc * +d3 * 4 - *$

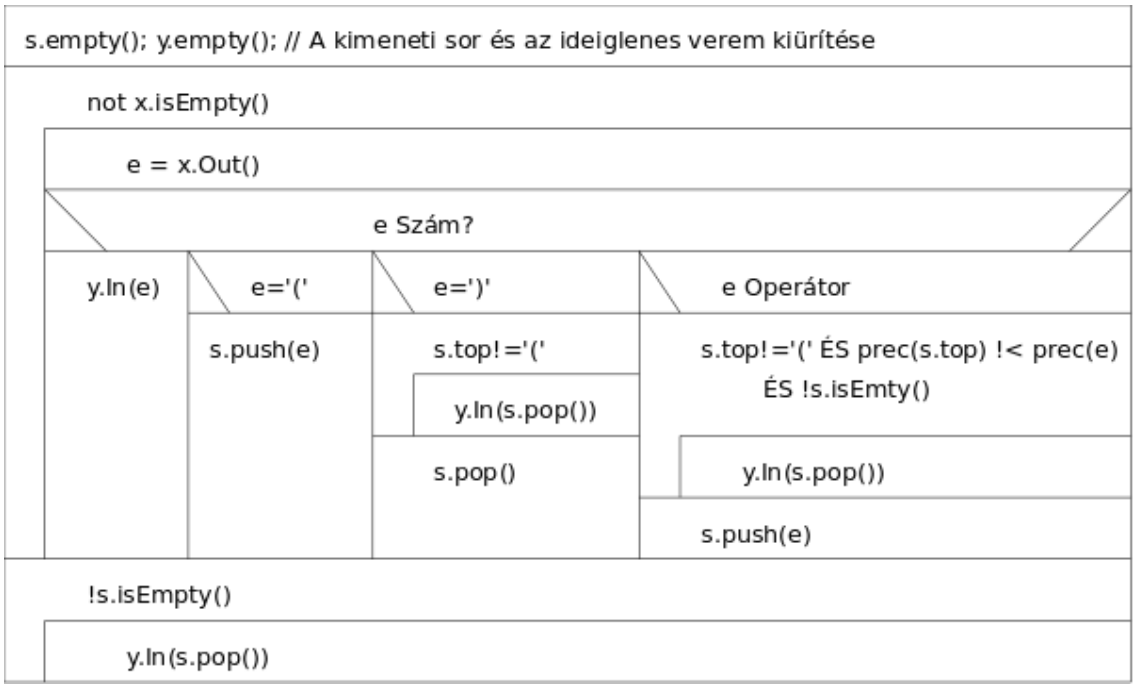
A lengyelformának a következő előnyei vannak a feldolgozás során

- A műveletek olyan sorrendben érkeznek, ahogy ki kell értékelni őket, vagyis a számítások sorrendjében
- A műveletek mindig a operandusok után állnak (postfix), két operandus beolvasása után rögvest végrehajtható a művelet (és eltárolható az eredmény újabb operandus gyanánt).
- Vermekkel lengyelformára lehet alakítani és az átalakított kifejezés kiértékelhető.
- Nem tartalmaz zárójeleket, a precedencia a formába beépítetten megtalálható.

5.2.1. Lengyelformára alakítás

A lengyelformára alakításnak több, egyszerű szabálya van. A feldolgozása alogritmusa használ egy `x` sort, ami a bemenő jeleket tartalmazza. Továbbá egy `y` sort, amibe az eredmény kerül, továbbá egy `s` segédvermet az átalakításhoz. Attól függően, hogy milyen karakter érkezik kell az alábbi szabályok közül egyet alkalmazni:

- Nyitózárójel esetén tegyük át a zárójelet az `s` verembe, az `x` sorból!
- Operandust írjuk ki a kimeneti `y` sorba.
- Operátor esetén: legfeljebb egy nyitózárójelig vegyük ki az `s` veremből a nagyobb prioritású operátorokat és írjuk ki az `y` sorba, majd ezt az operátort tegyük be az `s` verembe!
- Csukózárójel: a nyitózárójelig levő elemeket egyesével vegyük ki az `s` veremből és írjuk ki az `y` sorba, valamint vegyük ki a nyitózárójelet a veremből!
- Kifejezés végét elérve írjuk ki az `s` verem tartalmát az `y` sorba.



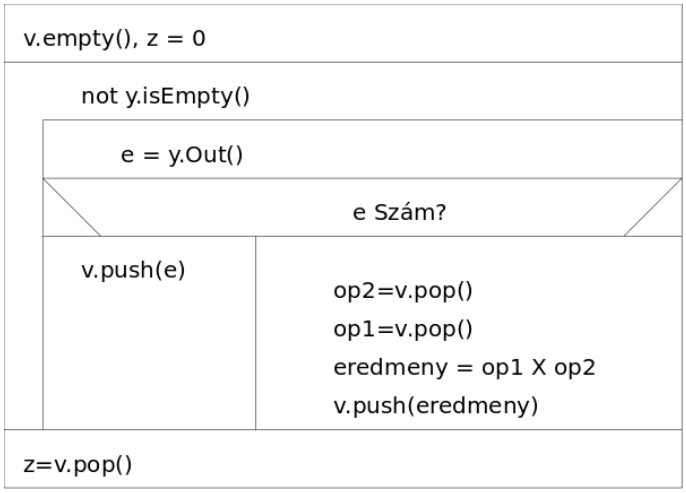
5.4. ábra. A lengyelformára alakítás stuktogrammja

5.2.2. Lengyelforma kiértékelése

A lengyelformára hozott kifejezés kiértékeléséhez egy `v` vermet használunk. Az y sorból egyesével vesszük az elemeket és az alábbi szabályok szerint járunk el:

- Ha operandus, akkor tegyük át a `v` verembe.
- Ha operátor, akkor vegyük ki a második operandust, majd az első operandust a `v` veremből. Végezzük el a műveletet és tegyük az eredményt a `v` verem tetejére.

Az algoritmus befejeztével a `v` veremben egyetlen érték van (ha mindent jól csináltunk) és az az érték a kifejezés értéke.



5.5. ábra. A lengyelforma kiértékelésének stuktogrammja

5.2.3. Lengyelforma példa

A bemutatott példát egy papíron érdemes követni, lépésről-lépésre felírva a kimenet és a verem állapotát. Vegyük az alábbi kifejezést: $(1+2)*(3+4)$. Amennyiben a szabályok szerint haladunk, legelőször egy nyitózárójellel találkozunk. Ez átkerül a verembe. A számot kiírjuk a kimenetre, majd a $+$ operátor következik. A veremben a nyitózárójel van tehát nem veszünk semmit sem, hanem a betesszük a $+$ jelet is verembe. Következik egy csukózárójel, tehát mindent kiveszünk a veremből nyitózárójelig. (Ekkor a kimeneti sorban az áll, hogy $1\,2+$.) A szorzás jele bekerül a verem, majd a kifejezés második felével hasonlóan bánunk el mint az első felével. Azaz a nyitózárójel a $*$ felé kerül a veremben, kiírjuk a 3-at, majd a $-$ is bekerül a verembe, a sorra pedig semmi, hiszen nyitózárójelig nincsen fontosabb művelet a veremben. A csukózárójel hatására kikerül a $-$ jel. (Ekkor a kimeneten az alábbi található: $1\,2+3\,4-$.) Mivel a bemeneti kifejezés végére értünk a maradék szimbólumokat is kiírjuk a veremből, aminek eredménye: $1\,2+3\,4-*$. Ez az átalakított kifejezés.

Ezt követően a kiértékelés menete az alábbiak szerint történik. Két szám érkezik egymást követően, bekerülnek a verembe, jön egy operátor melynek értelmében összeget számolunk és az eredményt tesszük a verembe. (3) Azután szintén jön két szám, így a veremben már három elem lesz: $3\,3\,4$. Ezek után a legfelső kettőm végrehajtjuk a $-$ műveletet. $(3\ -1)$ Majd a legvégén a $*$ műveletet. A veremben egyetlenegy szám lesz, ami a végeredmény is egyben: -3.

5.3. Rendezések

Ebben a szakaszban a rendezési problémával ismerkedünk meg, majd néhány jól használható rendezőalgoritmussal.

5.3.1. Rendezési probléma

A rendezési probléma formálisan a alábbi módon definiálható. Adott a bemenet és a kimenet:

Bemenet

n számot tartalmazó $(a_1, a_2, \ldots, a_n)$ sorozat

Kimenet
A bemenő sorozat olyan $(a'_1, a'_2, \ldots, a'_n)$ permutációja, hogy $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Ahol a kimenetben tehát a megadott sorozat elemei szerepelnek valamilyen más sorrendben, ami sorrendre igaz, hogy rendezve van. (Itt a $\leq$ jel egy absztrakt műveletet jelöl, ami a rendezés alapjául szolgál, az összehasonlításhoz szükséges.)

A probléma általánosítása, amikor a sorozat, amit rendezni szeretnénk, elemei nem számok, hanem összetett adatszerkezetek, például osztályok. Minden egyes elem tartalmaz egy **kulcsot**, amely kulcs lesz a rendezés alapjául szolgáló adatelem, tehát a $\leq$ absztrakt műveletet terjesztjük ki tetszőleges típusú (a_n) sorozatokra.

Rendezési reláció

A rendezési reláció definíciója: Legyen U egy halmaz, és $<$ egy kétváltozós reláció U-n. Ha $a, b \in U$ és $a < b$, akkor azt mondjuk, hogy „a kisebb, mint b”. A $<$ reláció egy rendezés, ha teljesülnek a következők:

- $a \not< a \; \forall a \in U$ elemre ($<$ irreflexív) Egy elem önmagánál nem kisebb.
- Ha $a, b, c \in U$, $a < b$, és $b < c$, akkor $a < c$ ($<$ tranzitív).
- Tetszőleges $a \neq b \in U$ elemekre vagy $a < b$, vagy $b < a$ fennáll ($<$ teljes).

Ha $<$ egy rendezés U-n, akkor az $(U; <)$ párt rendezett halmaznak nevezzük.

Példa
$\mathbb{Z}$ az egész számok halmaza. A szokásos $<$ rendezés a nagyság szerinti rendezés.

Itt viszont már a szokásos műveletet jelenti a $<$.

A következőkben néhány következik a rendezésre. Az eredeti sorozat az alábbi elemeket tartalmazza, ahol egy egyes elemek összetett típusok.

Személy = Név $\times$ Magasság $\times$ Születés

Abigél	Janka	Zsuzsi	Dávid	Dorka
132	128	92	104	70
1996	1998	2001	2000	2002

A rendezés eredménye, amikor a név a kulcs a rendezéshez:

Név szerint

Abigél	Dávid	Dorka	Janka	Zsuzsi
132	104	70	128	92
1996	2000	2002	1998	2001

A rendezés eredménye, amikor a születési év a kulcs a rendezéshez:

Születési év szerint

Abigél	Janka	Dávid	Zsuzsi	Dorka
132	128	104	92	70
1996	1998	2000	2001	2002

A következőekben három négyzetes (lassú) majd két hatékonyabb rendező algoritmusa kerül ismertetésre.

5.3.2. Buborék rendezés

Egyszerűsítésként rendezzük az $A[1 \dots n]$ tömböt! A tömb elemtípusa tetszőleges T típus, amire egy teljes rendezés értelmezhető.

Buborék rendezés alapötlete: a tömb elejétől kezdve „felbuborékoltatjuk" a legnagyobb elemet. Utána ugyanezt tesszük az eggyel rövidebb tömbre, stb. Végül, utoljára még az első két elemre is végrehajtjuk a „buborékoltatást". A buborékoltatást során mindig két elemet vizsgálunk csak és ha rossz sorrendben vannak a tömbben (inverzióban állnak) akkor felcseréljük.

A sorozat rendezett akkor, ha nincs az elemek között inverzió. Ez a rendezés az inverziók folyamatos csökkentésével rendez.

Buborék rendezés példa – Első futam
Az összehasonlított elemeket ki vannak emelve:

12	*5*	6	2	10	11	1
5	*12*	*6*	2	10	11	1
5	6	*12*	*2*	10	11	1
5	6	2	*12*	*10*	11	1
5	6	2	10	*12*	*11*	1
5	6	2	10	11	*12*	*1*
5	6	2	10	11	1	**12**

Egyetlen menet után a legnagyobb elem felkúszott a tömb végére. A következő lépésben eggyel rövidebb tömbön végezzük el ugyanezt. A következőképpen történik

---

Buborék rendezés példa – Második futam

5	*6*	2	10	11	1	**12**
5	*6*	*2*	10	11	1	**12**
5	2	*6*	*10*	11	1	**12**
5	2	6	*10*	*11*	1	**12**
5	2	6	10	*11*	*1*	**12**
5	2	6	10	1	**11**	**12**

A módszert folytatva rendezett tömböt kapunk.

Műveletigény

A műveletigény kiszámításához az alábbi gondolatmenetet követjük:

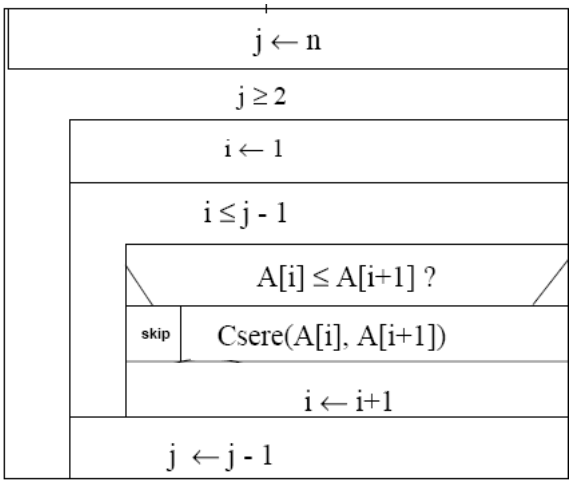
- Első menetben a tömb hosszának megfelelő összehasonlítás: n
- Legrosszabb esetben ugyanennyi csere, legjobb esetben nincsen csere.
- Az összehasonlítások száma állandó, a legrosszabb esetbeli cserék számával azonos.

- Ezt ismételjük eggyel rövidebb tömbre és így tovább:

$$n + (n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n} i = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$

A bemenet számának négyzetes függvénye az algoritmus lépésszáma, ezáltal lefutási ideje. Ez kellően nagy input esetén nagyon lassú futást eredményez, gondoljunk egy 100000 méretű tömbre például.

Algoritmus



5.6. ábra. Buborékrendezés struktogramja

Csere függvény

```
public void csere(int[] tomb, int i, int j)
{
int ideiglenes = tomb[i];
tomb[i] = tomb[j];
tomb[j] = ideiglenes;
}
```

Buborékrendezés

```
public void buborek(int[] tomb)
{
for (int j = tomb.length-1; j>0; j-)
for (int i = 0; i<j; i++)
if (tomb[i] > tomb[i+1])
csere(tomb, i, i+1)
}
```

Ez kifejezetten egy olyan algoritmus ami ugyan jó eredményt ad a rendezésre, de ennél lassabban csak úgy tudánk megoldani a problémát, ha direkt belekevernénk rendezés közben.

5.3.3. Maximum kiválasztásos rendezés

Ezzel az algoritmussal a buborék rendezéshez képest kevesebb lépéssel hajtjuk végre a rendezést. A buborék rendezésnél mindig a legnagyobb elemet tesszük fel a tömb végére, sok csere során. A maximum kiválasztásos rendezés kevesebb cserével, minden egyes futamban összesen egy cserével oldja meg a feladatot.

Keressük meg a tömbben a legnagyobb elemet és cseréljük fel a tömbben legutolsó elemmel. Ezután eggyel rövidebb résztömbre ismételjük az eljárást, addig, amíg az 1 hosszú tömböt kell rendeznünk, ami önmagában rendezett.

A maximális elemet lineáris kereséssel találhatjuk meg a tömbben.

Maximum kiválasztásos rendezés példa
A lineáris keresés lépéseit kihagytuk az alábbi példában.

12	5	6	2	10	11	1
1	5	6	2	10	11	**12**
1	5	6	2	10	*11*	**12**
1	5	6	2	10	**11**	**12**
1	5	6	2	*10*	**11**	**12**
1	5	6	2	**10**	**11**	**12**
1	5	*6*	2	**10**	**11**	**12**
1	5	2	**6**	**10**	**11**	**12**
1	*5*	2	**6**	**10**	**11**	**12**
1	2	**5**	**6**	**10**	**11**	**12**
1	*2*	**5**	**6**	**10**	**11**	**12**
1	**2**	**5**	**6**	**10**	**11**	**12**
1	**2**	**5**	**6**	**10**	**11**	**12**

Műveletigény

Az összehasonlítások száma a keresésekben, minden egyes lépésben a (rész)tömb hossza, tehát az előzőekben megmutatott:

$$\mathcal{O}(n^2)$$

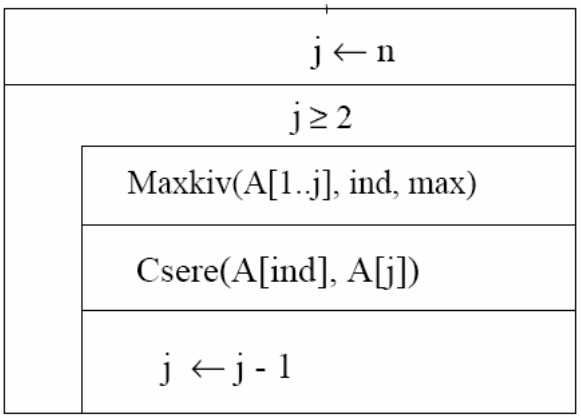
A cserék száma, a maximum kiválasztásnak köszönhetően, legfeljebb n tehát

$$\mathcal{O}(n)$$

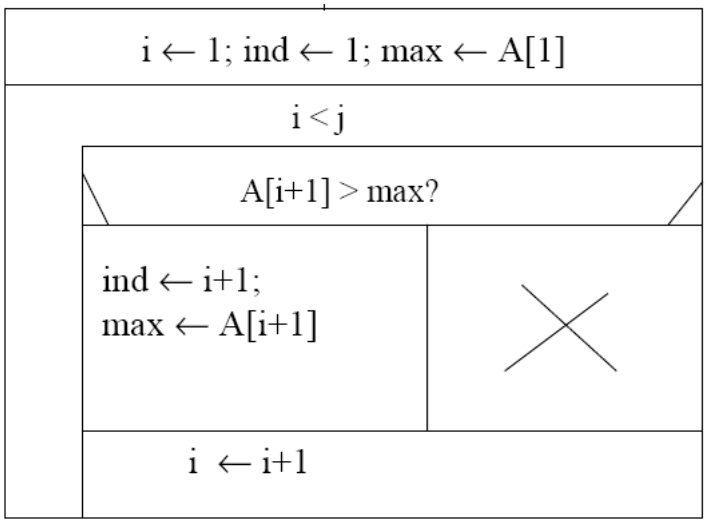
Így az algoritmus lépésszáma:

$$\mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$$

Látható, hogy ez is egy négyzetes, azaz lassú rendezési algoritmus. Azonban fontos megjegyezni, hogy a cserék száma kevesebb, ezáltal futási időben jobb, mint az előző, buborék rendezés.



5.7. ábra. Maximum kiválasztásos rendezés struktogrammja



5.8. ábra. Maximum kiválasztásos rendezés struktogrammja

Algoritmusa

Maximum kiválasztásos rendezés

```
public void maxkiv(int[] tomb)
{
for (int j = tomb.length-1; j>0; j-)
{
int index = 0;
for (int i =1; i<=j; i++)
if (tomb[index] < tomb[i])
index = i;
csere(tomb, index, i)
}
}
```

Gyakorta alkalmazzuk ezt az algoritmus napi életünkben is, például amikor a kár-

tyákat a kezünkben elrendezzük. (Illetve annak kicsit továbbfejlesztett és a következő algoritmussal kevert változatát.)

5.3.4. Beszúró rendezés

A beszúró rendezés az alábbi ötleten alapul: Tekintsük az tömböt rendezettnek. Egy új elem beszúrása történjen a megfelelő helyre, így a tömböt rendezettnek tartjuk meg.
Az alapgondolaton túl, tudjuk még, hogy egyetlen elem mindig rendezett. Az elején vesszük az egész tömb egy részét, a bal oldalról számított 1 hosszú résztömböt. Ebbe a résztömbbe szúrjuk be a megfelelő helyre a következő elemet, amit a tömb második eleme. A beszúrás után a rendezett résztömbünk már 2 hosszú. Ebbe is beszúrjuk a következő elemet és így tovább.

Nézzük meg a következő példát!

Beszúró rendezés példa
Az első elem önmagában rendezett. Ehhez szúrunk be egy másodikat, majd harmadikat, ...

12	5	6	2	10	11	1
5	**12**	6	2	10	11	1
5	**6**	**12**	2	10	11	1
2	**5**	**6**	**12**	10	11	1
2	**5**	**6**	**10**	**12**	11	1
2	**5**	**6**	**10**	**11**	**12**	1
1	**2**	**5**	**6**	**10**	**11**	**12**

A beszúráshoz a beszúrandó elem helyét lineáris kereséssel határozzuk meg a tömb elején kezdve. Amikor megleltük a pozíciót, akkor a maradék elemeket egyesével felfelé másoljuk, majd beszúrjuk a beszúrandót. (Ehhez a beszúrandót külön eltároljuk.)

Műveletigény

Az összehasonlítások száma a legrosszabb esetet tekintve szintén nem változik, ám az összehasonlítás szempontjából legrosszabb eset a cserék szempontjából a legjobb eset.

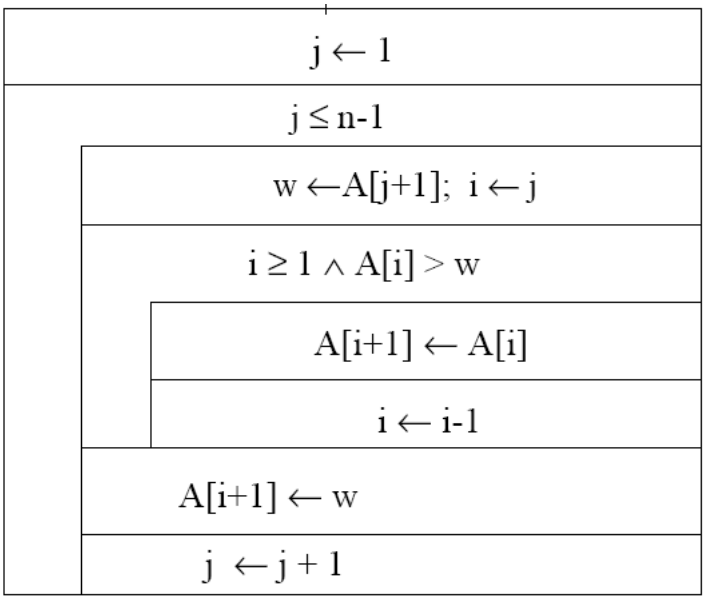
Ennek belátásához vegyünk egy eleve rendezett tömböt. A beszúrás mindig az utolsó pozícióba fog történni, hiszen sorban vannak, emiatt a cserék száma minimális. Azonban a lineáris keresés a tömb elejétől végéig összehasonlítja a beszúrandót a tömb elemeivel. Ez pedig a maximális érték.

A cserék szempontjából legrosszabb esetben a cserék száma szintén $\mathcal{O}(n^2)$–el becsülhető, amikor tömbben rendezünk. (Láncolt lista esetén például hatékonyabb.)

Algoritmus

Beszúró rendezés

```
public void maxkiv(int[] tomb)
{
for (int j = 0; j<tomb.lenght-1; j++)
{
```



5.9. ábra. Beszúró rendezés struktogrammja

```
int elmentve = tomb[j+1];
for (int i=j; (i>=0)&&(tomb[i]>elmentve); i-)
tomb[i+1] = tomb[i];
tomb[i+1] = elmentve;
}
}
```

Szintén ehhez hasonlót használunk a valós életben is, legtöbbször amikor dolgozatokat rendezünk pontszám szerint, csak egyidejűleg több elemet szúrunk be általában.

5.3.5. Gyorsrendezés – Quicksort

Az eddigieknél egy lényegesen hatékonyabb, a lépésszámot tekintve nem négyzetes nagyságrendű algoritmussal ismerkedünk meg.

Hatékony rendezési algoritmus – C.A.R. Hoare készítette, 1960-ban. Típusát tekintve az „Oszd meg és Uralkodj” elvet követi, amelyet a következőképpen kell érteni: Két fázisra osztható az algoritmus, rekurzívan hívja meg magát a részfeladatokra. (Természetesen a rekurziót a hatékonyság érdekében ki lehet váltani ciklussal is.) A fázisok

- Partíciós fázis – Oszd a munkát két részre!
- Rendezési fázis – Uralkodj a részeken!

Megosztási fázis. Válassz egy „strázsát” (pivot), egy tetszőleges elemet, majd válasszunk ennek egy pozíciót úgy, hogy minden elem tőle jobbra nagyobb legyen, és minden elem tőle balra kisebb legyen!

Uralkodási fázis. Alkalmazd ugyanezt az algoritmust mindkét félre, mint részproblémára.

A megosztást ábrázolva:

Kisebb elemek	Strázsa	Nagyobb elemek

A megosztási fázisban természetesen nem tudunk találni mindig egy olyan elemet, ami teljesíti a feltételeket. A kiválasztott strázsához képest vizsgáljuk a résztömb többi elemét és úgy cserélünk fel nehány elemet, hogy a feltétel igazzá váljon.

- Egyszerű pivot választás esetén legyen (rész)tömb balszélső eleme a pivot!
- A résztömb alső és felső felétől induljunk el egy indexszel a tömb közepe felé.
- A bal indexszel lépegetve felfelé megkeressük az első elemet, ami nagyobb mint a strázsa, tehát rossz helyen áll. Ugyanígy lefelé lépegetve megkeressük az első elemet ami kisebb mint a strázsa, tehát a jobb oldalon állva rossz helyen áll. A két „rossz" elemet felcseréljük.
- Addig folytatjuk az előző cserélgetést, amíg a két index össze nem találkozik.
- Ha megvan a bal-indexnél lévő elemet a pivottal felcseréljük.
- Ezek után az egész algoritmust alkalmazzuk a bal résztömbre és a jobb résztömbre.

Egyetlen megosztás során, garantáljuk azt, hogy a pivot elem a rendezés végeredménye szerinti jó helyre kerül, valamint szétosztjuk az elemeket kétfelé.

Példa

Gyorsrendezés példa
Balszélső a strázsa.

9	5	6	2	10	11	1

Vegyük a két indexet

9	5	6	2	10	11	1

A jobb oldalsó rossz, a bal jó. Ezért a jobb-index változatlan, míg a bal lép felfelé.

9	5	6	2	10	11	1
9	5	6	2	10	11	1
9	5	6	2	10	11	1

Most jön a csere.

9	5	6	2	1	11	10
9	5	6	2	1	11	10

Végül a pivot elemet betesszük a bal helyére.

1	5	6	2	**9**	11	10

Ezzel a kilenc a helyére került és a tőle balra, majd tőle jobbra levő résztömbökre hajtjuk végre ugyanezt az algoritmust.

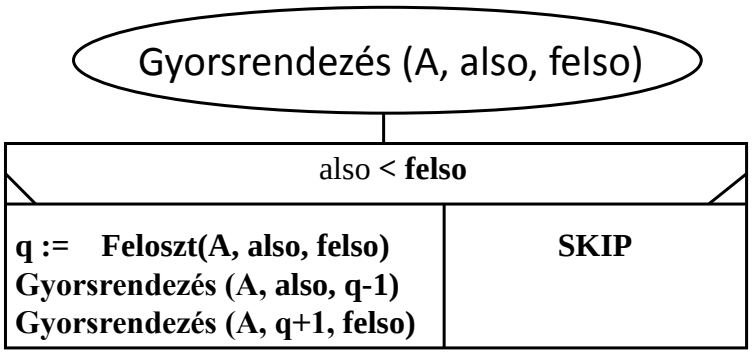
Műveletigény

Felosztás: vizsgálj meg minden elemet egyszer $\mathcal{O}(n)$
Uralkodás: az adatok kétfelé osztása $\mathcal{O}(\log_2(n))$ – Ez a szétbontások optimális mennyisége. Összesen a kettő szorzata, ami $\mathcal{O}(n \log_2(n))$, ez pedig jobb, mint az eddig megismert rendezéseink, de van egy apró gond ugyanis, ha eredetileg egy rendezett sorozatot adunk bemenetnek, akkor az algoritmusunk minden egyes lépésben felosztja a rendezendő tömböt egy 0 és egy $n-1$ hosszú tömbre, majd azon folytatja a rendezést. (A bal a pivot és minden más nagyobb tőle.) Ennek a műveletigénye pedig $n\mathcal{O}(n)$, ami egyenlő $\mathcal{O}(n^2)$-el.

Tehát azt kaptuk, hogy rossz esetben a gyorsrendezés olyan lassú, mint a buborék rendezés. Lehet ezen segíteni különböző strázsa választási stratégiával. Minden a pivot választáson múlik, ugyanis ha tudunk jól úgy pivotot választani, hogy a partíciók mérete közel azonos legyen, akkor hatékonyan működik az algoritmus. Lehet több strázsát választani, vagy pedig véletlenszerűen választani. (Ami a valószínűségek természetéből adódóan átlagosan jó eredményt szolgáltat.)

Általánosságban elmondható, hogy a gyorsrendezés kevés művelettel gyorsan rendez, azonban nem stabil az ideje, tehát viszonylag nagy határok között ingadozik.

Algoritmus



5.10. ábra. Gyorsrendezés struktogrammja

„Ilyet ember kézzel nem csinál . . . ”

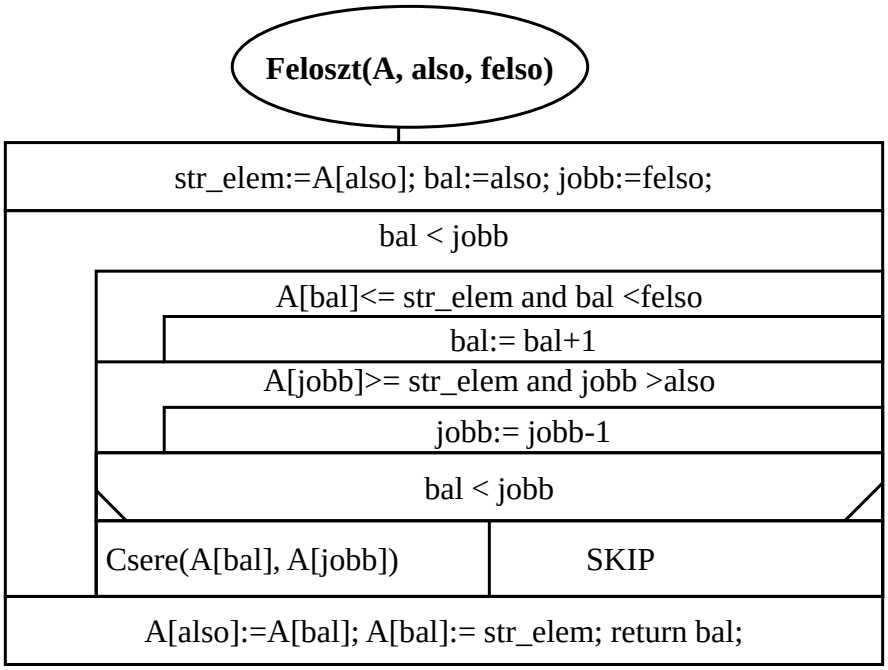
5.3.6. Edényrendezés

Végül egy szintén gyors rendezővel ismerkedünk meg.

Tegyük fel, hogy tudjuk, hogy a bemenő elemek ($A[1 \dots n]$ elemei) egy m elemű U halmazból kerülnek ki. Például $\forall i$-re igaz, hogy $i \in [1 \dots m]$. Lefoglalunk egy U elemeivel indexelt B tömböt (m db ládát), először mind üres. A B segédtömb elemei lehetnek bármi, például láncolt lista.

Az edényrendezés két fázisból fog állni, először a ládák szerint (azaz, hogy milyen érték tartozik ahhoz a ládához) kigyűjtjük az elemek, majd sorban visszahelyezzük az eredeti tömbbe.

Kigyűjtés. Először meghatározzuk rendezendő tömb legkisebb és legnagyobb elemét. Ezek után lefoglalunk egy megfelelő méretű segédtömböt, amibe az elemeket fogjuk gyűjteni. (Ez a tömb a legnagyobb és a legkisebb elem közötti különbség plusz egy.) A se-



5.11. ábra. Gyorsrendezés struktogrammja

gédtömbbe, fogjuk gyűjteni a rendezendő tömb elemeit, aszerint, hogy melyik rekeszbe tartoznak.

Összefűzés. A segédtömbbe kigyűjtött elemeket azután sorban visszafűzzük az eredeti tömbbe, és ezzel kész a rendezés.

Edényrendező – példa

2	2	1	1	5	3	2	5	4

Ebben az esetben látható, hogy a lehetséges értékek 1 és 5 között vannak, ezért egy $5 - 1 + 1 = 5$ hosszú segédtömböt kell lefoglalni.

2	2	1	1	5	3	2	5	4	Segédtömb:	1	2	3	4	5

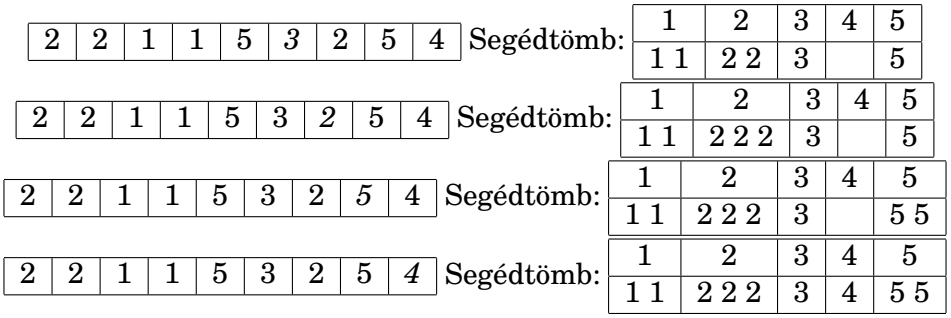
2	2	1	1	5	3	2	5	4	Segédtömb:	1	2	3	4	5
											2			

2	*2*	1	1	5	3	2	5	4	Segédtömb:	1	2	3	4	5
											2 2			

2	2	*1*	1	5	3	2	5	4	Segédtömb:	1	2	3	4	5
										1	2 2			

2	2	1	*1*	5	3	2	5	4	Segédtömb:	1	2	3	4	5
										1 1	2 2			

2	2	1	1	*5*	3	2	5	4	Segédtömb:	1	2	3	4	5
										1 1	2 2			5



Ezután az edények tartalmát egyszerű lineáris kiolvasással az eredeti tömbbe helyezzük.

Második fázis

1	1	2	2	2	3	4	5	5

Amivel a rendezés be is fejeződött.

Műveletigény

Lépésszám.

- Segédtömb létrehozása: $\mathcal{O}(m)$
- Kigyűjtő fázis $\mathcal{O}(n)$
- Visszarakó fázis $\mathcal{O}(n+m)$
- Összesen $\mathcal{O}(n+m)$.

Ez jobb, mint az eddigi rendezőink!, hiszen egy lineáris idejű rendezőt kapunk. Azonban ennek súlyos ára van! Az edényrendező felhasznál egy segédtömböt, ami bizonyos esetekben akkora, mint az eredeti tömb (esetleg nagyobb is). Tehát a térbeli (memória) komplexitása eddigi rendezőinkhez képest nagyobb. Edényrendező akkor éri meg, ha a rendezendő értékek értékkészletének halmaza kicsi. Nyilvánvaló, hogy vannak olyan bemenetek, amelyek kétszer már nem férnek el a memóriában.

Edényrendezőt köznapi életben akkor használunk, amikor a pakli kártyát a sorba rendezéshez először színek szerint szétdobáljuk.

5.3.7. Kupacrendezés

A kupac adatszerkezetet rendezőként is lehet alkalmazni. Gondoljuk el, hogy a rendezendő tömb értékeit egyszerűen beszúrjuk egy kupacba majd kiolvassuk onnan, úgy hogy mindig eltávolítjuk a gyökeret, mint legkisebb elemet.

Műveletigény

A kupacnál a beszúrás és maximum törlés műveletigénye, legfeljebb a fa magasságával arányos ($\mathcal{O}(h)$), ami az $\mathcal{O}(\log n)$). Ezt megszorozzuk az összes beszúrandó elem számával ami n beszúrás esetén $\mathcal{O}(n \log n)$ eredményt ad. Figyelembe véve, hogy egy egyaránt felső korlátja a beszúrásnak és a kitörlésnek, az kapjuk, hogy rendezés teljes műveletigénye $\mathcal{O}(2 * n \log n)$, ami pedig szintén $\mathcal{O}(n \log n)$.

Ez a műveletigény a gyorsrendezővel esik egy rendben. A kupacrendezés azonban stabil lepésszámú rendező, nem függ a rendezés semmitől, úgy mint a gyorsrendezés esetén a pivot megválasztásától. Általában a gyorsrendezés kevesebb lépéssel dolgozik, azaz valójában gyorsabb, mint a kupacrendezés, azonban a gyorsrendezés képes „elromlani” nem megfelelő strázsa esetén.