II. ALAPVETŐ ADATTÍPUSOK

II. ALAPVETŐ ADATTÍPUSOK

Ebben a részben az alapvető adattípusokat (adatszerkezeteket) tekintjük át. Abban a kérdésben, hogy melyek ezek, a szakmai közfelfogás lényegében megegyezik; a szakkönyvek is nagyjából ugyanazokat ismertetik.

Az elemi adattípusokat, azok műveleteit, az egyes programnyelvekben meglévő lehetőségeket ismertnek vesszük és itt nem tárgyaljuk. Ilyenek például a különböző egészek, valós számok, karakterek, pointer típus.

A típuskonstrukciókat is ismertnek vesszük, ezek ismertetése a „Bevezetés a programozásba" c. tárgy keretein belül megtörtént. Ezek a rekord (szekvencia), alternatív adatszerkezet (unió) és az iterált, ill. ezek speciális változatai. Megjegyezzük, hogy a három alapvető adatszerkezet elemeit rendre a három alapvető programszerkezettel lehet elérni és feldolgozni: rekord részeit (mezőit) szekvenciával, az unió alkotóit elágazással és az iterált szerkezet elemeit ciklussal.

3. Tömbök

3.1. ADT

Legyen T egy E alaptípus feletti $k\,(\geq 1)$ dimenziós tömb típus. Vezessük be az $I = I_1 \times ... \times I_k$ indexhalmazt, ahol $\forall j \in [1\,..\,k]: I_j = [1\,..\,n_j]$ (1 helyett kezdődhetne az indexelés m_j-vel is, de az egyszerűség kedvéért 1-et fogunk használni).

Az $A \in T$ tömbnek így $N = n_1 \cdot n_2 \cdot ... \cdot n_k$ eleme van, melyek halmazát jelölje $\{a_1,...,a_N\}$. Ekkor mindig van egy $f : I$ a $\{a_1,...,a_N\}$ kölcsönösen egyértelmű leképzés (ami egyfajta indexfüggvény, de nem az 1. fejezetben bevezetett indexfüggvény).

Jelölés: $A[i_1,i_2,...,i_k]$ a tömbnek az a_j elemét jelöli, ha az $i_1,i_2,...,i_k$ indexekből alkotott rendezett *k*-as *f* szerinti képe a *j* érték: $f(i_1,i_2,...,i_k) = j$.

Megjegyzés: A tömbökkel kapcsolatban olykor megadásra kerül még egy R invariáns is, pl. szimmetrikus vagy hézagosan kitöltött mátrix esetén ennek a fennálló tulajdonságnak a kifejezésére.

Műveletek (melyek bármely *k* esetén szerepelnek): indexelés (=tömbelem kiválasztása) a már bevezetett $A[i_1,i_2,\ldots,i_k]$ kifejezéssel, ami szerepelhet értékadó utasítás bal oldalán, és jobb oldalán is. Így lehetséges:

1. tömbelem lekérdezése,
2. tömbelem módosítása.

A tömb „merev” adatszerkezet, a mérete nem változik, mert nem lehet a tömbbe egy elemet beszúrni, és nem lehet a tömbből egy elemet kitörölni.

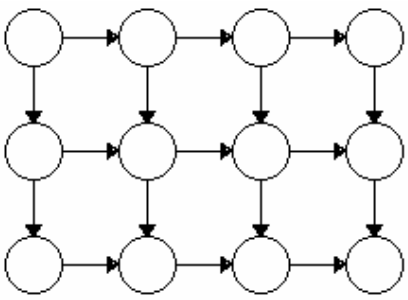
Elnevezés: $k = 1$ esetén a *vektor*, $k = 2$ esetén a *mátrix* elnevezés használatos, míg az általános esetben *k-dimenziós tömbnek* nevezzük az adattípust.

3.2. ADS

Az a megegyezés született, hogy *j* szerinti rákövetkezést is definiálnak: $köv_j\, A[i_1,...,i_j,...,i_k] = A[i_1,...,i_{j+1},...,i_k]$ (ahol $i_j < n_j$).

Minden belső elemnek *k* db rákövetkezője van (azért csak a belsőknek, mert definíció szerint a dimenzió határánál nincs rákövetkező).

A tömb $k \geq 2$ esetén ortogonális adatszerkezet (lásd: 2.2. alfejezet)



Valójában a tömbről nem ilyen képet őrzünk fejünkben, ahogyan a verem sem egy lineáris gráf formájában rögzül a memóriánkban. A vektor egy beosztásokkal ellátott szalag, a

2-dimenziós tömb egy mátrix, a 3-dimenziós tömb egy cellákra osztott téglatest alakját ölti gondolatainkban.

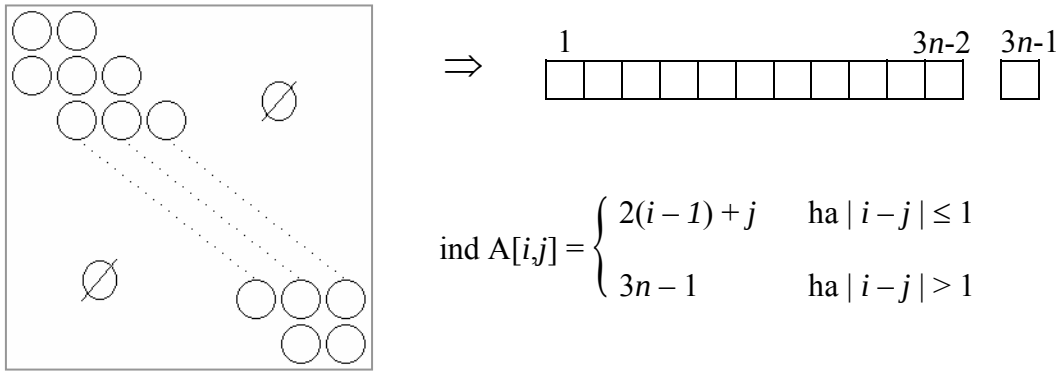
3.3. Reprezentációs szint

3.3.1. Aritmetikai ábrázolás

Csak akkor használjuk ezt a módot, ha a hatékony ábrázolás úgy kívánja. Az elemeket általában sorfolytonosan, ill. oszlopfolytonosan helyezzük egy vektorba.

(Gyakorlaton több ilyen példát is részletesen kidolgozunk.)

Példa: Tridiagonális mátrix sorfolytonos elhelyezése vektorban ($n \times n$-es mátrix esetén):



$$\text{ind } A[i,j] = \begin{cases} 2(i-1)+j & \text{ha } |i-j| \le 1 \\ 3n-1 & \text{ha } |i-j| > 1 \end{cases}$$

A tridiagonális mátrixban csak a főátlóban és a közvetlenül „hozzá tapadó” két átlószerű alakzatban találhatók értékes elemek, a többi elem értéktelen számunkra (pl. nulla). Az elemeket sorfolytonosan elhelyezzük egy megfelelő méretű vektorban. Szokás szerint az értékes elemek után még egy nullát is leteszünk, hogy a nem értékes tartománybeli címek is mutathassanak valahová a vektorban.

3.3.2. Láncolt ábrázolás

Tipikusan a hézagosan kitöltött mátrixot (más néven: ritka mátrixot) ábrázolják így.

ÁBRA!!!

4. Verem

A mindennapok során gyakran találkozunk a verem tároló struktúrával. Legismertebb példa a névadó, a mezőgazdaságban használt verem. Több távolsági buszon is a vezető a különböző pénzérméket egy-egy kis verembe tudja betenni, ill. azokból veszi elő. A kávéházakban az alátét tányérokat gyakran egy veremként működő tároló mélyedésben tartják, stb.

4.1. ADT

Műveletek:

Empty:	$\rightarrow V$	Üres verem konstans; az üres verem „létrehozása”
IsEmpty:	$V \rightarrow L$	A verem üres voltának lekérdezése
Push:	$V \times E \rightarrow V$	Elem betétele a verembe
Pop:	$V \rightarrow V \times E$	Elem kivétele a veremből
Top:	$V \rightarrow E$	A felső elem lekérdezése

Megszorítások:

$D_{Pop} = D_{Top} = V \setminus \{Empty\}$

a) Algebrai specifikáció: ahogyan az 1.fejezet 2.1.1.-es pontjában

1. *IsEmpty* (*Empty*) vagy $v = Empty \rightarrow IsEmpty(v)$
2. *IsEmpty* (v) → $v = Empty$
3. ¬*IsEmpty* (*Push*(v,e))
4. *Pop* (*Push*(v,e)) = (v,e)
5. *Push* (*Pop*(v)) = v
6. *Top* (*Push*(v,e)) = e

b) Funkcionális specifikáció:

Egy $v \in V$ verem absztrakt szinten elempárok halmazának tekinthető, ahol az elempár első komponense a verembe betett E-beli érték, a második komponens pedig a behelyezés időpontja:

$$v = \left\{ (e_i, t_i) \,\middle|\, i \in \{1,\ldots,n\} \wedge n \geq 0 \wedge \forall i, j \in \{1,\ldots,n\} : i \neq j \rightarrow t_i \neq t_j \right\}.$$

Empty:

A = V_v

B = $V_{v'}$

Q = ($v = v$’)

R = ($v = \varnothing$)

Isempty:

$A = \underset{v}{V} \times \underset{l}{L}$

$B = \underset{v'}{V}$

$Q = (v = v')$

$R = (Q \wedge l = (v' = \varnothing))$

Push:

$A = \underset{v}{V} \times \underset{e}{E}$

$B = \underset{v'}{V} \times \underset{e'}{E}$

$Q = (v = v' \wedge e = e')$

$R = (e = e' \wedge v = v' \cup \{(e', t)\} \wedge \forall i\ ((e_i, t_i) \in v') : t_i < t)$

Pop:

$A = \underset{v}{V} \times \underset{e}{E}$

$B = \underset{v'}{V}$

$Q = (v = v' \wedge v' \neq \varnothing)$

$R = (v = v' \setminus \{(e_j, t_j)\} \wedge e = e_j \wedge (e_j, t_j) \in v' \wedge \forall i\ ((e_i, t_i) \in v' \wedge i \neq j) : t_j > t_i)$

Top:

$A = \underset{v}{V} \times \underset{e}{E}$

$B = \underset{v'}{V}$

$Q = (v = v' \wedge v' \neq \varnothing)$

$R = (Q \wedge e = e_j \wedge (e_j, t_j) \in v' \wedge \forall i\ ((e_i, t_i) \in v' \wedge i \neq j) : t_j > t_i)$

Még egyszer hangsúlyozzuk, hogy a fenti absztrakt reprezentáció csupán matematikai, nem így implementáljuk a verem adattípust!!!

4.2. ADS

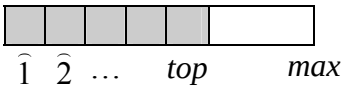
A verem lineáris adatszerkezet; az alapvető szerkezetet egyirányú gráf ábrázolja.

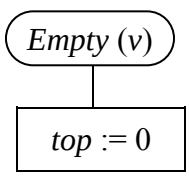
A veremre úgy gondolunk, mint ahogyan az a bal oldali ábrán látható, nem pedig lineáris gráfként. Az ADS szinten természetesen a műveletek is változatlanul jelen vannak.

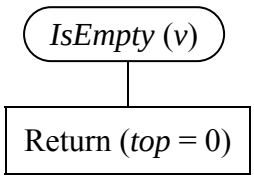
4.3. Reprezentációs szint

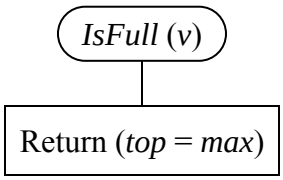
4.3.1. Aritmetikai ábrázolás

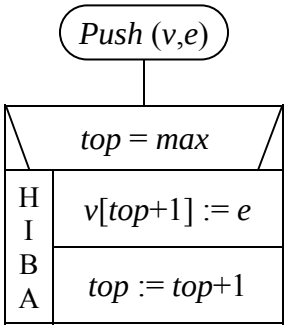
Bevezetjük a *top* változót, mely mindig a legfelső elem indexe (*top* ∈ [0..*max*]). A *v* vermet ábrázoló tömb: *v*[1..*max*]. A szokásos műveletek mellett használni kell egy új, *IsFull* műveletet is, mivel a tömb betelhet, és ezt figyelni kell.

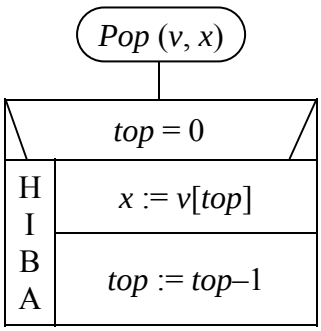


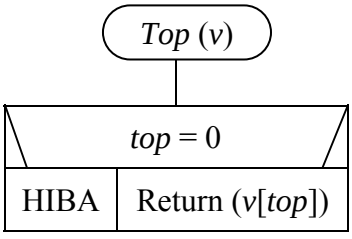






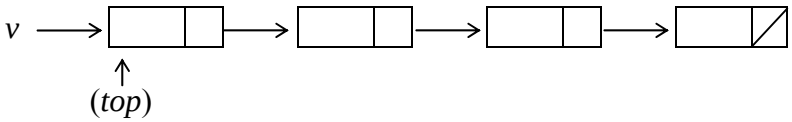




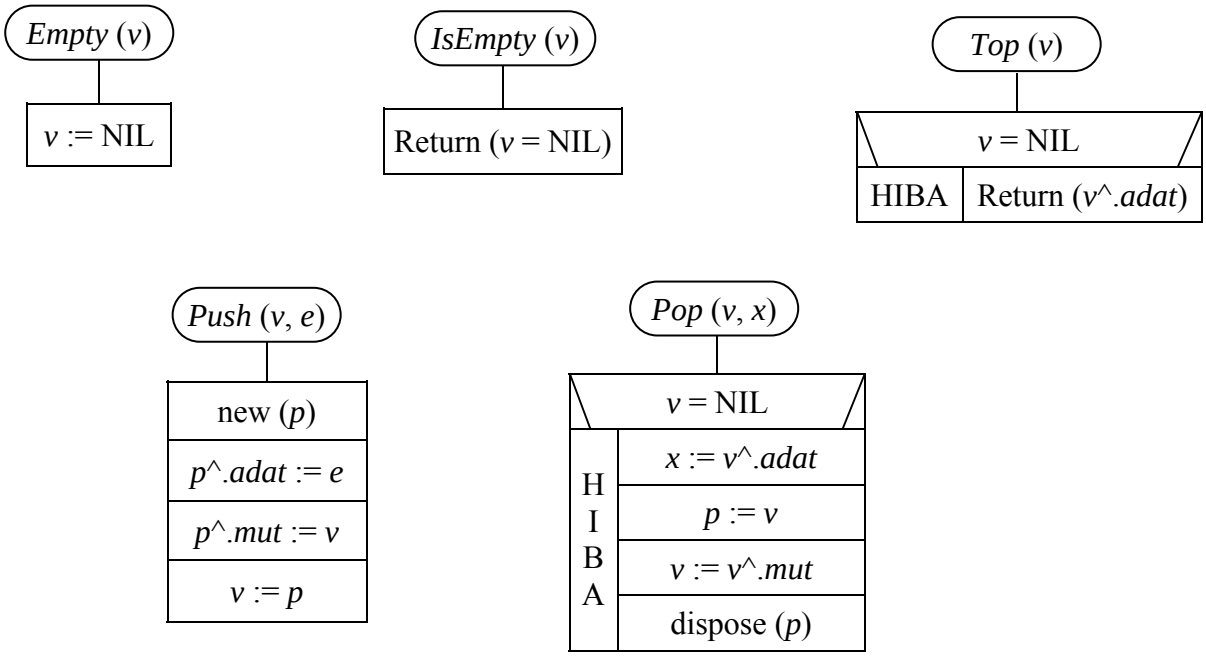


4.3.2. Láncolt ábrázolás

Itt a *v* változó egy pointert jelöl.



Ebben az ábrázolásban *v* = *top*, tehát utóbbi felesleges, ezért nem is használjuk.



Ebben a felfogásban a verembe beszúrandó értéket adjuk meg a *Push* utasítás paramétereként, illetve az értéket kapjuk meg a *Pop* utasítás paraméterében. Ennek megfelelően a beszúrandó listaelemet létre kell hozni (*new*), illetve a kiláncolt listaelemet fel kell szabadítani (*dispose*).

Úgy is meg lehet írni az utóbbi két műveletet, hogy a *Push* egy kész listaelem pointerét kapja meg, a *Pop* pedig a kiláncolt listaelem mutatóját adja vissza.

4.4. Verem alkalmazásai

A verem adatszerkezetnek számos alkalmazásával találkozhatunk az algoritmusok és programok világában. Alapvetően egy sorozat megfordítására alkalmas: ABCD → DCBA. Ha azonban a verembe írást és a kivételt nem elkülönítve, egymás után, hanem „összekeverve" alkalmazzuk, akkor a sorozat számos átrendezését meg tudjuk valósítani. Itt most két jellegzetes alkalmazást mutatunk be.

4.4.1. Lengyelforma

Lukasewich lengyel matematikus az 50-es években a matematikai formulák olyanfajta átalakítását dolgozta ki, amelynek segítségével a fordítóprogram könnyen ki tudja számítani a kifejezés értékét. (Pontosabban: olyan kódot generál, amely – végrehajtva – kiszámítja a kifejezés értékét.) Erre azért volt szükség, mert az ember által megszokott „infix" és zárójeles írásmód nem látszott alkalmas struktúrának a kiértékelés céljára. A bevezetett új ábrázolási formát a szerző tiszteletére *lengyelformának* is nevezik. Másik elnevezés a posztfix forma.

Mind a lengyelformára hozás, mind pedig annak kiértékelése vermes algoritmus.

A lengyelformára hozott kifejezés jellemzői:

a) nincs benne zárójel,

b) az operandusok sorrendje egymáshoz képest változatlan,

c) minden műveleti jel közvetlenül az operandusai mögött áll.

Egyszerű példák:

$a + b$	→	a b +	
$a * b + c$	→	$a\ b * c +$	(eltérő precedenciák)
$a * (b + c)$	→	$a\ b\ c + *$	(zárójel hatása)
$a + b - c$	→	$a\ b + c -$	(azonos precedenciák)
a ^ 2 ^ 3	→	a 2 3 ^ ^	(hatványozás esetén fordítva: a ^ 2 ^ 3 = a ^ 8)

VIZSGÁRA: számot kell adni a lengyelformára hozás algoritmusáról úgy, hogy megadjuk a verem tartalmát az eljárás minden lépésében (struktogram nem kell).

4.4.2. A helyes zárójelezés feldolgozása (egymásba ágyazott folyamatok kezelése)

Egymásba ágyazott folyamatok kezelése: például egymásba ágyazott eljáráshívások a programokban. Ennek legegyszerűbb modellje a helyes zárójelezés feldolgozása.

Feladat: Helyes zárójelezésben az összetartozó zárójelpárok megkeresése és a párok sorszámainak kiíratása.

Pl.: (( )) ((( ))) ( ) ⇒ (2, 3), (1, 4), (7, 8), (6, 9), (5, 10), (11, 12).
1 2 3 4 5 6 7 8 9 10 11 12

Definiáljuk a helyes zárójelezés nyelvét (HZ)! Két fajta definíciót is adunk.

1. Definíció: HZ ⊂ { (,) }* nyelvre teljesülnek az alábbiak:

a) $\varepsilon \in$ HZ

b) Ha $h \in$ HZ, akkor $(h) \in$ HZ

c) Ha $h_1, h_2 \in$ HZ, akkor $h_1 h_2 \in$ HZ

d) Csak az a, b és c pontok alkalmazásával nyert sorozatok a helyes zárójelezések, más nem.

2. Definíció: $h \in HZ \Leftrightarrow l_{(}(h) = l_{)}(h) \wedge \forall u \in Pre(h): l_{(}(u) \geq l_{)}(u)$

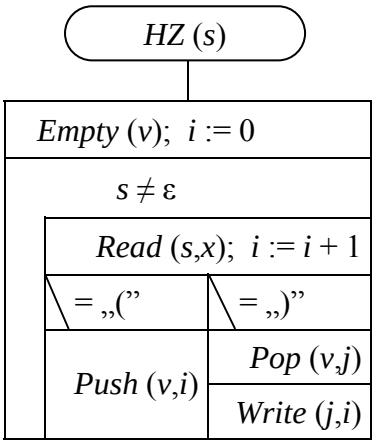
Az általánosan elterjedt l(*s*) jelölés az *s* sorozat hosszát (a benne lévő karakterek számát) jelöli, ennek általánosításaként bevezetjük a $l_x(s)$ jelölést:

$l_x(s)$: *s* szövegben előforduló *x* karakterek száma.

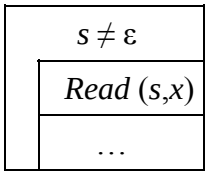
A másik jelölést az s sorozat kezdőszeleteinek halmazára vezetjük be:

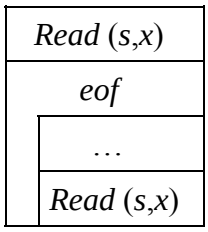
Pre(*s*): *s* karaktersorozat összes prefixuma (az üres karaktertől a teljes *s*-ig). -

Ezek után a helyes zárójelezés algoritmusa a következő:



A „Bevezetés a programozáshoz” c. tárgyban a szekvenciális input file-oknál használt *sx,dx,x:read* helyett *Read*(*s*,*x*) absztrakt olvasó utasítást használjuk, ahol *s* a szekvenciális input file, és *x* az a változó, melybe a következő elemet olvassuk. Ha nem szükséges az előreolvasásos technika alkalmazása a feladat megoldásához, akkor általában az $s \neq \varepsilon$ feltétel teljesüléséig olvasunk, vagyis addig, amíg ki nem ürül a fájl. Ha előreolvasásra lenne szükség, akkor bevezetjük az *eof*(*s*), vagy ha nem okoz félreértést, akkor csak az *eof* logikai változót, amely akkor áll igazra, ha az üres fájlból próbálunk olvasni.





A második ábrán tehát az *eof* használata: *eof* = *igaz* $\Leftrightarrow$ $s = \varepsilon$ -ból történt az olvasás. Valójában tehát *sx,dx,x:read* tartalmilag ekvivalens *Read*(*s*,*x*) + *eof* használatával.

5. Sor

A sor is ismert tárolási struktúra a köznapi éltből.

5.1. ADT

Műveletek:

Empty:	$\rightarrow S$	Üres sor konstans; az üres sor „létrehozása”
IsEmpty:	$S \rightarrow L$	A sor üres voltának lekérdezése
In:	$S \times E \rightarrow S$	Elem betétele a sorba
Out:	$S \rightarrow S \times E$	Elem kivétele a sorból
First:	$S \rightarrow E$	Az első elem lekérdezése

Megszorítások:

$D_{Out} = D_{First} = S \setminus \{Empty\}$

a) Algebrai specifikáció:

1. $IsEmpty(Empty)$ vagy $s = Empty \rightarrow IsEmpty(s)$
2. $IsEmpty(s) \rightarrow s = Empty$
3. $\neg IsEmpty(In(s,e))$
4. $Out(In(Empty,e)) = (Empty,e)$
5. $\neg IsEmpty(s) \rightarrow Out(In(s,e))_2 = Out(s)_2$
6. $\neg IsEmpty(s) \rightarrow In(Out(s)_1,e) = Out(In(s,e))_1$
7. $First(s) = Out(s)_1$

Itt az 1-es, illetve a 2-es index a pár első, illetve második komponensét jelöli.

b) Funkcionális specifikáció:

Egy $s \in S$ sor absztrakt szinten elempárok halmazának tekinthető, ahol az elempár első komponense a sorba betett E-beli érték, a második komponens pedig a behelyezés időpontja:

$$s = \left\{ (e_i, t_i) \,\middle|\, i \in \{1,\ldots,n\} \wedge n \geq 0 \wedge \forall i, j \in \{1,\ldots,n\} : i \neq j \rightarrow t_i \neq t_j \right\}.$$

Empty:

A = $\underset{s}{S}$

B = $\underset{s'}{S}$

Q = ($s = s'$)

R = ($s = \varnothing$)

Isempty:

$A = \underset{s}{S} \times \underset{l}{L}$

$B = \underset{s'}{S}$

$Q = (s = s')$

$R = (Q \wedge l = (s' = \varnothing))$

In:

$A = \underset{s}{S} \times \underset{e}{E}$

$B = \underset{s'}{S} \times \underset{e'}{E}$

$Q = (s = s' \wedge e = e')$

$R = (e = e' \wedge s = s' \cup \{(e', t)\} \wedge \forall i\ ((e_i, t_i) \in s') : t_i < t)$

Out:

$A = \underset{s}{S} \times \underset{e}{E}$

$B = \underset{s'}{S}$

$Q = (s = s' \wedge s' \neq \varnothing)$

$R = (s = s' \setminus \{(e_j, t_j)\} \wedge e = e_j \wedge (e_j, t_j) \in s' \wedge \forall i\ ((e_i, t_i) \in s' \wedge i \neq j) : t_j < t_i)$

Top:

$A = \underset{s}{S} \times \underset{e}{E}$

$B = \underset{s'}{S}$

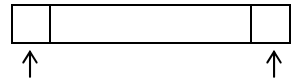
$Q = (s = s' \wedge s' \neq \varnothing)$

$R = (Q \wedge e = e_j \wedge (e_j, t_j) \in s' \wedge \forall i\ ((e_i, t_i) \in s' \wedge i \neq j) : t_j < t_i)$

Még egyszer hangsúlyozzuk, hogy a fenti absztrakt reprezentáció csupán matematikai, nem így implementáljuk a sor adattípust!!!

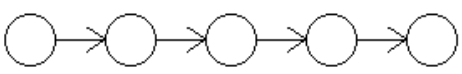
5.2. ADS

A sor a veremhez hasonlóan lineáris adatszerkezet.



eleje *vége*

Ez a szerkezet izomorf a következő irányított gráffal:



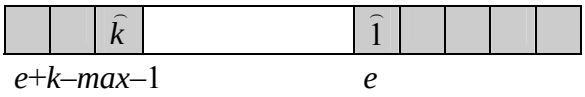
A gráf fordítva is irányítható, ADS szinten nem dönthető el, hogy a nyilak hátra felé irányulnak a sorban, vagy hátulról előre mutatnak. A veremre úgy gondolunk, mint ahogyan az a bal oldali ábrán látható, nem pedig lineáris gráfként. Az ADS szinten természetesen a műveletek is változatlanul jelen vannak.

5.3. Reprezentációs szint

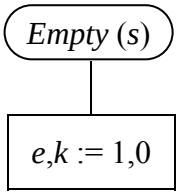
5.3.1. Aritmetikai ábrázolás:

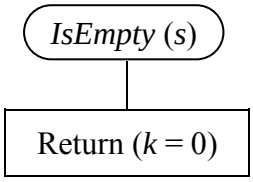
Az *s* sor ebben az ábrázolásban egy rekordszerkezet, melynek része egy *s*[1..*max*] tömb, amely a sor elemeit tartalmazza, egy $e \in \{1, \ldots, max\}$ index, amely mindenkor az első elemre mutat, valamint egy $k \in \{0, \ldots, max\}$ változó, amely a sor elemszámát jelzi.

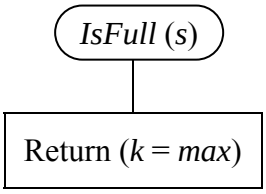
A sor úgy működik, hogy a sor végére rakjuk, ill. az elejéről veszünk ki az elemeket, ezért célszerű a tömb elejére „beengedni” a tömb végén „kilógó” elemeket, különben betelne úgy a tömb, hogy egyes indexei felhasználatlanok. Egy tipikus sor lassan „körbemegy” a tömbön.

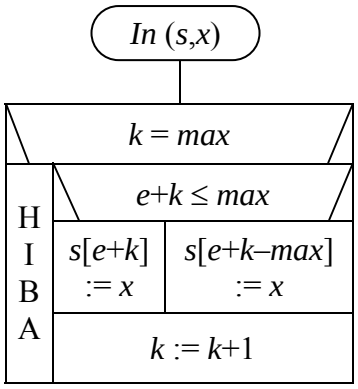


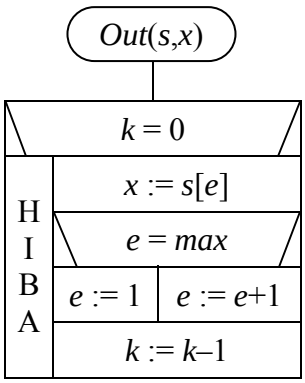
A műveletek között szerepel az *IsFull* is.

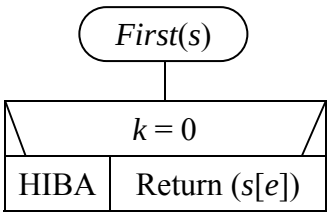






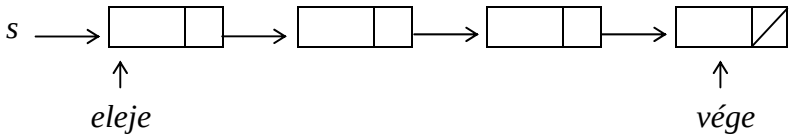




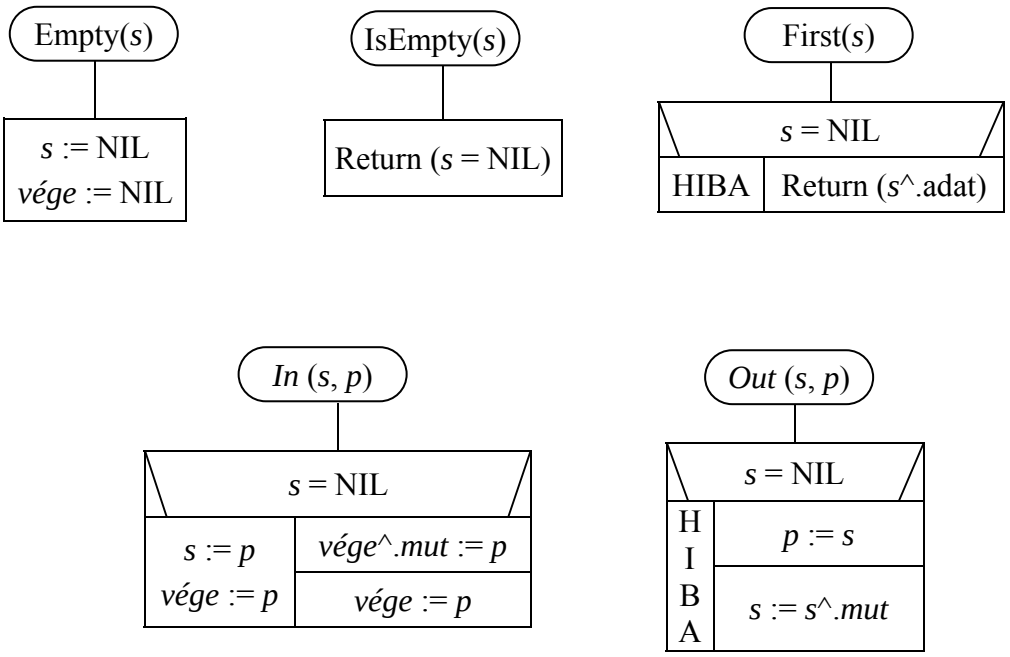


5.3.2. Láncolt ábrázolás

Ekkor az *s* változó egy pointert jelöl.



Az *eleje* pointert nem használjuk, mivel megegyezik *s*-sel. A műveletek:



Most máshogyan írtuk meg a beszúró és törlő műveletet, mint a veremnél. A beszúró egy elkészített listaelem mutatóját kapja meg és az elemet csak be kell fűznie a lista elejére. A törlő pedig a kiláncolt első elem mutatóját adja vissza.

5.4. A sor alkalmazásai

1. Egyes pufferelési eljárásokban (pl. klaviatúránál).
2. Gráfok, ill. fák szélességi bejárásánál (pl. bináris fák szintfolytonos bejárása, lásd ott!).
3. Elméleti érdekesség: a sor „körbetekerésével” szimulálható a verem adatszerkezet, de a sor csak két veremmel valósítható meg.

Megjegyzés: Említenünk kell még egy lehetséges adatszerkezetet, a kétvégű sort.