4. VEREM

A mindennapokban is találkozunk *verem* alapú tároló struktúrákkal. Legismertebb példa a névadó, a mezőgazdaságban használt verem. Az informatikában legismertebb verem-alkalmazások az eljáráshívások végrehajtása során a vezérlésátadások kezelése, valamint a kifejezések lengyelformára alakítása, illetve ennek alapján a kifejezések kiértékelése.

4.1. A verem absztrakt adattípus

Az E alaptípus feletti $V = V(E)$ halmazban mindazon *vermek* megtalálható, amelyek véges sok E-beli elemből épülnek fel. Ide értjük az *üres vermet* is, amely nem tartalmaz elemet; ezt ennek ellenére, mint $V(E)$-beli adattípust, típusosnak tekintjük. A verem *műveletei* közé soroljuk az üres verem létrehozását (*Üres*), a verem üres állapotának a lekérdezését (*Üres-e*), adat betételét (*Verembe*), adat kivételét (*Veremből*) és annak a verembeli elemnek a lekérdezését (*Felső*), amely kivételre következik.

Az utolsó művelet neve (*Felső*) utal arra, amit intuitív módon tudunk a veremről: az utoljára betett elemet lehet kivenni, amely függőleges elrendezésű verem esetén felül helyezkedik el. A veremből való kivétel és a felső elem lekérdezésének műveletét ahhoz az *előfeltételhez* kötjük, hogy a verem nem lehet üres. Absztrakt szinten úgy tekintünk a veremre, mint amelynek a befogadóképessége nincs korlátozva, vagyis a *Tele-e* lekérdezést itt nem vezetjük be.

A verem adattípus absztrakt leírása során *nem* támaszkodhatunk szerkezeti összefüggésekre, azok nélkül kell specifikálnunk ezt az adattípust. Ennek kétféle módját ismertetjük.

4.1.1. Algebrai specifikáció

Először megadjuk a verem műveleteit, mint leképezéseket. Ezek közül talán csak az *Üres* művelet értelmezése lehet szokatlan: egyrészt *létrehoz* egy vermet, amely nem tartalmaz elemeket (lásd: deklaráció a programnyelvekben), másrészt az *üres verem konstans* neve is. Az *Üres* tehát egy konstans, ezért, mint leképezés nulla argumentumú. Az alábbi műveleteket vezetjük be.

$Üres: \to V$	Üres verem konstans; az üres verem létrehozása
$Üres\text{-}e: V \to L$	A verem üres voltának lekérdezése
$Verembe: V \times E \to V$	Elem betétele a verembe
$Veremből: V \to V \times E$	Elem kivétele a veremből
$Felső: V \to E$	A felső elem lekérdezése

Megjegyezzük, hogy a Veremből műveletet úgy is lehetne definiálni, hogy a kivett elemet nem adja vissza, hanem „eldobja". Az a művelet, amelyet így vezetnénk be, egy törlő utasítás lenne; ennek eredménye nem egy (*verem, elem*) rendezett pár, hanem csak az új verem lenne.

Megadjuk a leképezések *megszorításait*. A *Veremből* és a *Felső* műveletek értelmezési tartományából ki kell vennünk az üres vermet, arra ugyanis ez a két művelet nem értelmezhető.

$$D_{Veremből} = D_{Felső} = V \setminus \{Üres\}$$

Az *algebrai specifikáció* logikai axiómák megadásával valósul meg. Sorra vesszük a lehetséges művelet-párokat és mindkét sorrendjükről megnézzük, hogy értelmes állításhoz jutunk-e. Az alábbi *axiómákat* írjuk fel; magyarázatukat alább adjuk meg.

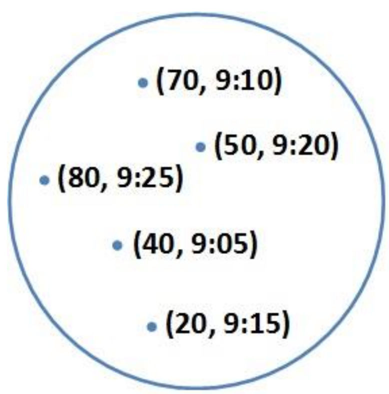
1. *Üres-e* (*Üres*) vagy *v* = *Üres* → *Üres-e* (*v*)
2. *Üres-e* (*v*) → *v* = *Üres*
3. ¬*Üres-e* (*Verembe* (*v*, *e*))
4. *Veremből* (*Verembe* (*v*, *e*)) = (*v*, *e*)
5. *Verembe* (*Veremből* (*v*)) = *v*
6. *Felső* (*Verembe* (*v*, *e*)) = *e*
7. *Felső* (*v*) = *Veremből* (*v*).2

Az 1. axióma azt fejezi ki, hogy az üres verem konstansra teljesül az ürességet állító predikátum. Ezt változó használatával egyenlőségjelesen is megfogalmaztuk. A 2. állítás az üres verem egyértelműségéről szól. Ezt követi annak formális megfogalmazása, hogy ha a verembe beteszünk egy elemet, akkor az már nem üres. A 4-5. axiómapár a verembe történő elhelyezés és az elem kivétel egymásutánját írja le, mindkét irányból. Mindkét esetbe a kiinduló helyzetet kapjuk vissza. (Megjegyzendő, hogy a két axióma közül a másodikban a *Verembe* művelet argumentum-száma helyes, ugyanis a belső *Veremből* művelet eredménye egy (*verem*, *elem*) pár.) Végül, az utolsó két állítás a felső elem és a két vermet módosító művelet kapcsolatát adja meg.

Egy ilyen axiómarendszertől először is azt várjuk, hogy helyes állításokat tartalmazzon. Természetes igény a teljesség is. Ez azt jelenti, hogy ne hiányozzon az állítások közül olyan, amely nélkül a verem meghatározása nem lenne teljes. Végül, a redundancia kérdése is felvethető: van-e olyan állítás a specifikációban, amely a többiből levezethető?

4.1.2. Funkcionális specifikáció

A *funkcionális specifikáció* módszerében pusztán *matematikai eszközök* használatával olyan verem-fogalmat vezetünk be, amely talán közelebb áll a szemléletünkhöz, mint az axiomatikus leírás eredménye. Absztrakt szinten úgy tekintjük a vermet, mint (*elem*, *idő*) rendezett párok halmazát. Az időpontok azt jelzik, hogy az egyes elemek mikor kerültek a verembe. Kikötjük, hogy az időpontok mind *különbözők*. Ezek után tudunk a legutoljára bekerült elemre hivatkozni. A 4.1. ábrán szereplő absztrakt veremnek öt eleme van és utoljára a 80-as került a verembe.



4.1. ábra. A verem, mint érték-időpont párok halmaza (ADT)

Formálisan ez például a következő módon írható le:

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

Ha a verem *műveleteit* szeretnék specifikálni, akkor azt most már külön-külön egyesével is megtehetjük, nem kell az egymásra való hatásuk axiómáit meggondolni. Definiáljuk például a *Veremből* műveletet. Az ismert *elő-, utófeltételes* specifikációs módszerrel azt írjuk le formálisan, hogy ez a művelet a veremből az utoljára betett elemet veszi ki, vagyis azt, amelyikhez a legnagyobb időérték tartozik. (Ha az olvasónak még sem lenne ismerős az alábbi jelölésrendszer, akkor elég, ha a módszer lényegét informális módon érti és jegyzi meg.)

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

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

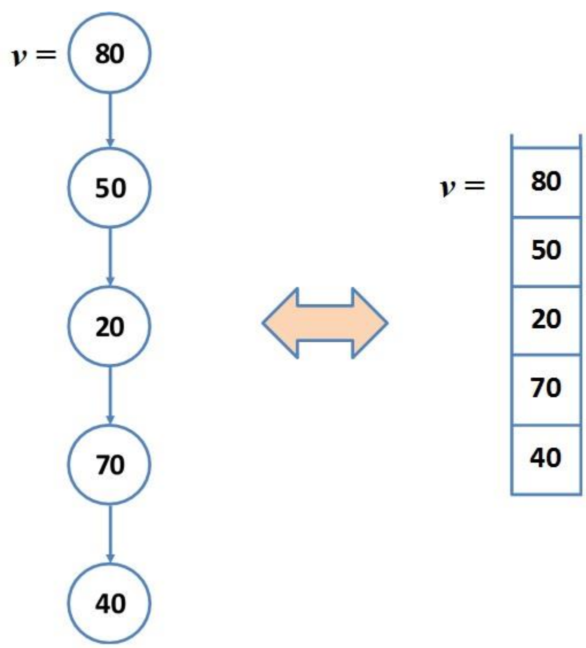
$$Q = (v = v' \wedge v' \neq \emptyset)$$

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

Hangsúlyozzuk, hogy a fenti absztrakt reprezentáció csupán matematikai és nem tartalmaz semmiféle utalást a verem adattípus implementálásának a módjára.

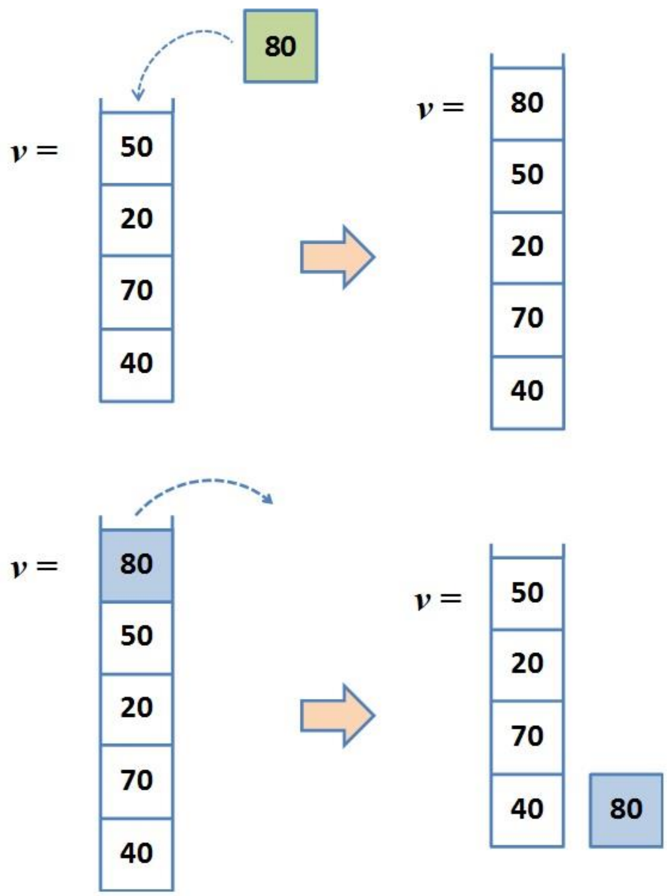
4.2. A verem absztrakt adatszerkezet

A verem, ha az *absztrakt szerkezetét* nézzük, elemeinek *lineáris* struktúrájaként mutatkozik. A 4.2. ábra szemlélteti a verem ADS-et, mint egy lineáris gráfot, valamint azt a megjelenési formát, ahogyan a veremre gondolunk, illetve, ahogyan a szakmai kommunikációban hivatkozunk rá. A két szerkezet lényegében nem különbözik egymástól; szemmel láthatóan megfeleltethetők egymásnak.



4.2. ábra. A verem, mint rákövetkező elemek (speciális lineáris gráf, ADS)

Az ADS szinten természetesen a műveletek is változatlanul jelen vannak. Ennek a szintnek a lényeget kifejező ábrázolási módja felhasználható arra, hogy a műveletek hatását szemléletesen bemutassuk. A 4.3. ábra a *Verembe* és a *Veremből* műveletek hatását illusztrálja.



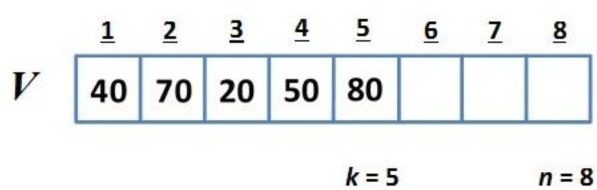
4.3. ábra. Verem-műveletek szemléltetése: *Verembe* és *Veremből* (ADS)

4.3. A verem reprezentációja

A verem adattípust egyaránt lehet tömbösen és láncoltan ábrázolni. Sorra vesszük ezt a két reprezentálási módot.

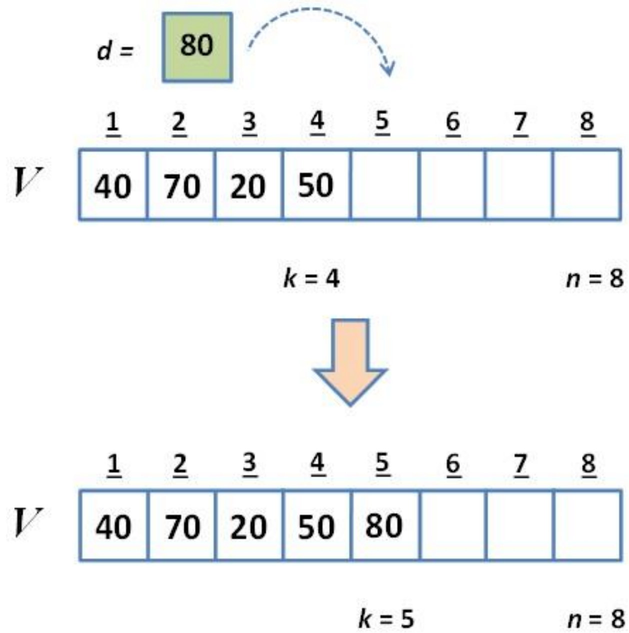
4.3.1. Tömbös ábrázolás

A verem *tömbös ábrázolásában* a $v[1..n]$ tömb mellett a felső elem k indexét, valamint a *hiba* logikai változót használjuk. A verem v jelölése ezeket a komponenseket együttesen jelenti. A 4.4. ábrán látható veremben tömbös ábrázolásba ugyanaz, mint amelyet a 4.1. ábra jelenített meg.

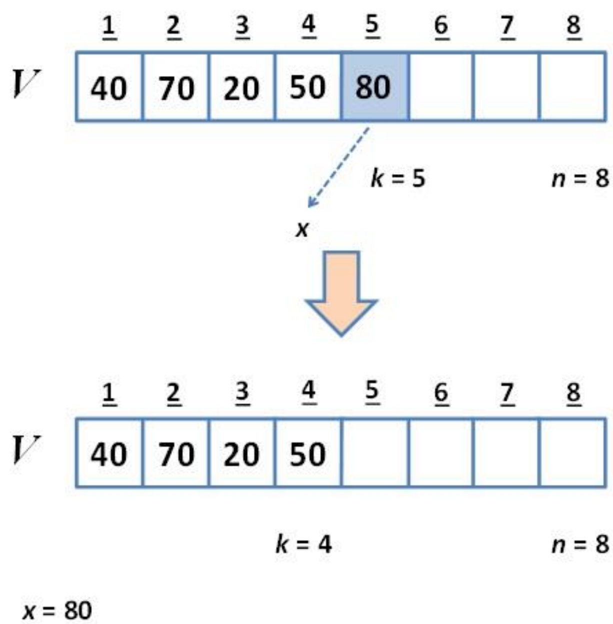


4.4. ábra. Verem tömbös reprezentálása

A tömbös reprezentáció is alkalmas a műveletek hatásának bemutatására. A 4.5.a és a 4.5.b ábra a *Verembe* és a *Veremből* műveletek eredményét mutatja be.

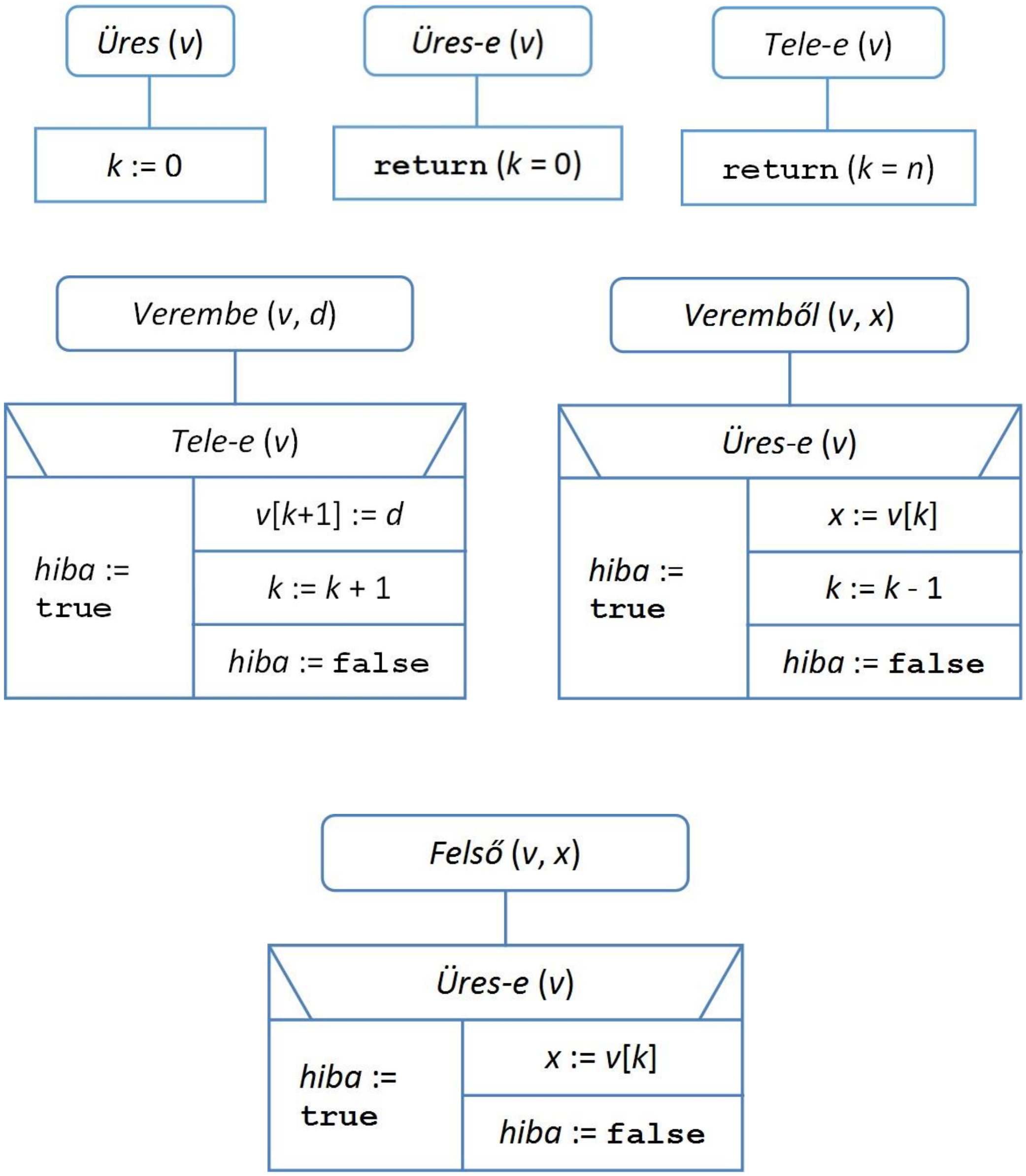


4.5.a. ábra. Verem-műveletek szemléltetése: *Verembe* (tömbös reprezentáció)



4.5.b. ábra. Verem-műveletek szemléltetése: *Veremből* (tömbös reprezentáció)

Megadjuk a verem műveleteinek algoritmusait a tömbös reprezentációra. Mivel az elemeket tároló tömb betelhet, ezért bevezetjük a *Tele-e* műveletet is, amely még ADT és ADS szinten nem szerepelt. Az üres vermet $k = 0$ jelenti, míg $k = n$ utal a tele veremre. A műveletek elvégzése után a *hiba* változó mindig értéket kap, sikeres művelet esetén *hamisat*, ellenkező esetben pedig a hibára utaló *igaz* értéket. A műveletek algoritmusai a 4.6. ábrán láthatók.



4.6 ábra. Verem műveletei tömbös reprezentáció esetén

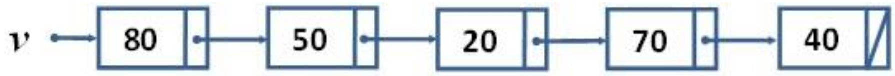
Könnyen látható, hogy a tömbös reprezentációban a verem minden művelete – függetlenül a verem méretétől – konstans időben végrehajtható:

$$T_{op}(n) = \Theta(1)$$

ahol *op* a fenti verem-műveletek bármelyikét jelentheti.

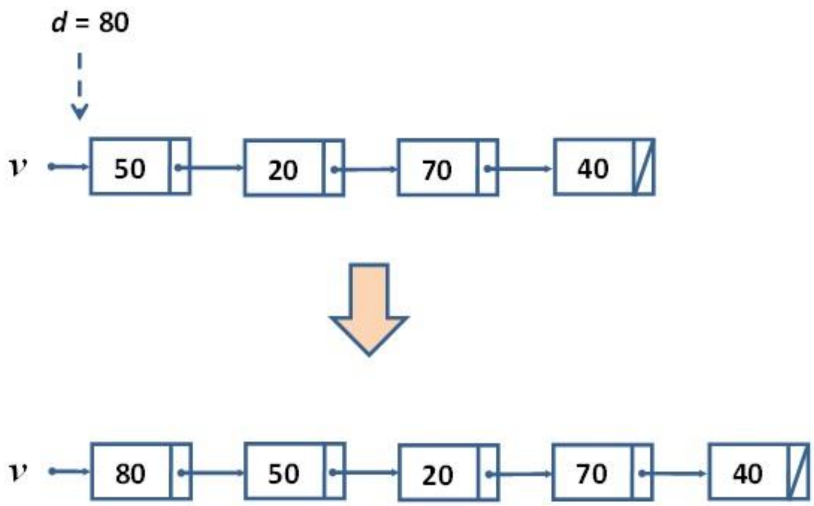
4.3.2. Láncolt ábrázolás

A verem láncolt reprezentációjában a *v* pointer típusú változó nem csak az adatstruktúrához biztosít hozzáférést, hanem egyben a verem *felső elemére* mutat. Ezért nem kell külön bevezetni egy felső elem mutatót. Üres verem esetén *v* = NIL.

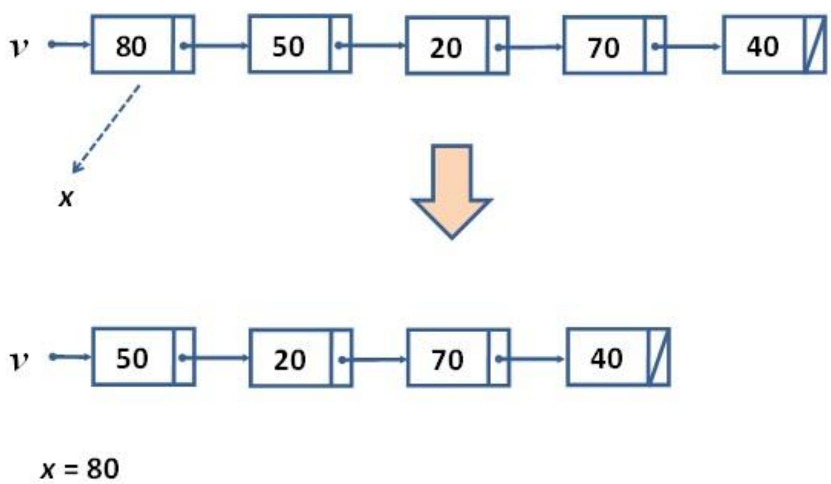


4.7. ábra. Verem láncolt ábrázolása

A 4.7. ábrán látható verem megegyezik azzal, mint amelyet absztrakt szinten, illetve tömbösen vezettünk be. Az elemek sorrendje értelemszerűen olyan, hogy a verem utoljára betett felső eleme a lánc elején található. A beszúrás és kivétel ilyen módon mindig a lista első elemére vonatkozik. Ennek a két műveletnek a hatását mutatja be a 4.8.a és a 4.8.b ábra.



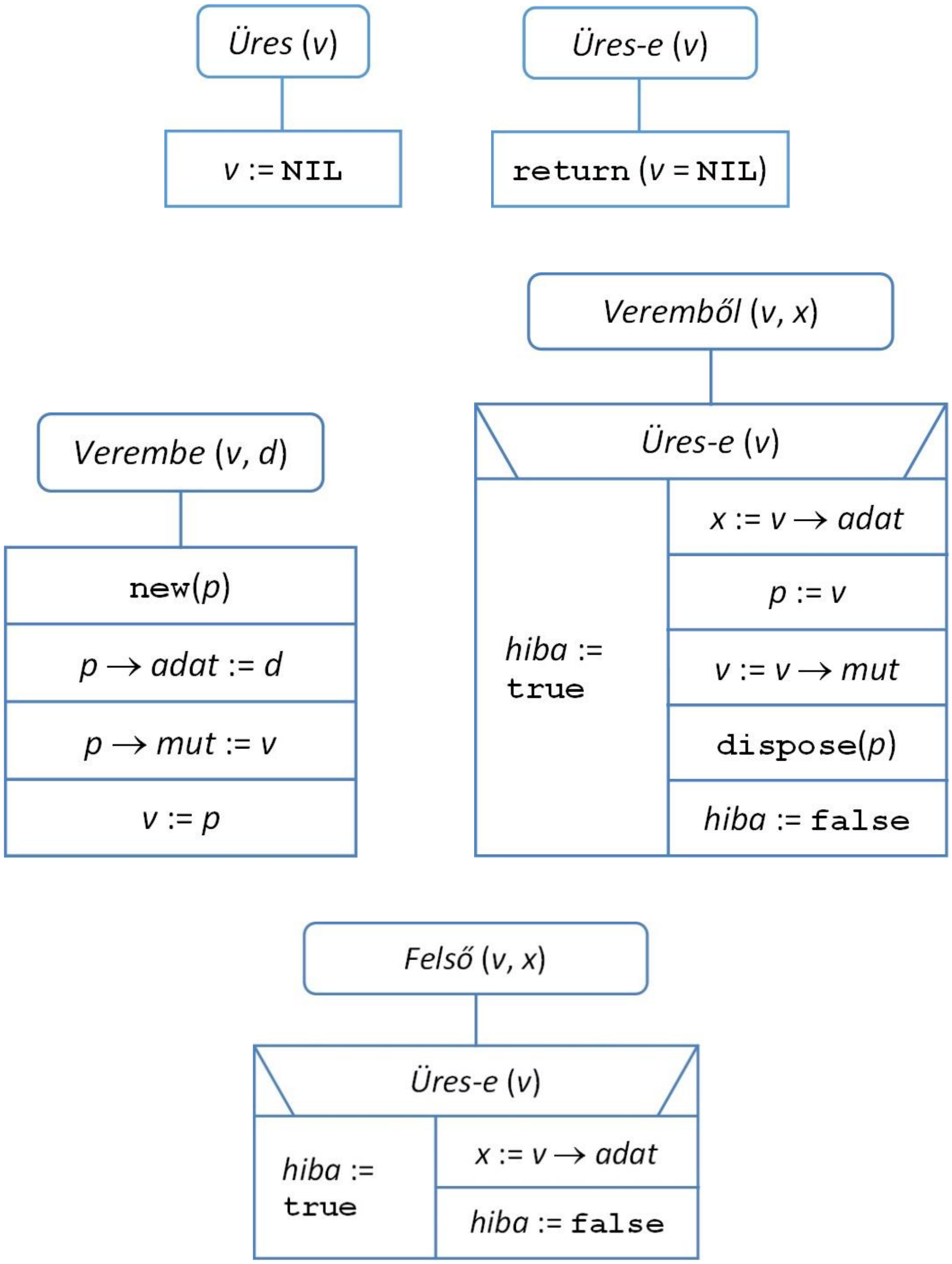
4.8.a. ábra. Verem-műveletek szemléltetése: *Verembe* (láncolt ábrázolás)



4.8.b. ábra. Verem-műveletek szemléltetése: *Veremből* (láncolt ábrázolás)

A láncolt ábrázolású verem műveletei között ismét *nem* szerepel a *betelt* állapot lekérdezése, mivel ebben a reprezentációban nem számolunk a tárolókapacitás gyakorlati felső korlátjával.

A 4.9. ábrán látható algoritmusokban a verembe beszúrandó értéket adjuk meg a *Verembe* utasítás paramétereként, illetve a kiláncolt felső elem értékét kapjuk meg a *Veremből* utasítás paraméterében. Ennek megfelelően a művelete részeként a beszúrandó listaelemet létre kell hozni (*new*), illetve a kiláncolt listaelemet fel kell szabadítani (*dispose*).



4.9. ábra. A verem műveletei láncolt ábrázolás esetén

Úgy is meg lehet írni az utóbbi két műveletet, hogy a *Verembe* egy *kész listaelem* pointerét kapja meg, míg a *Veremből* a *kiláncolt* listaelem mutatóját adja vissza. Ezzel a paraméter átadás-átvételi móddal majd a bináris keresőfa műveleteinél találkozunk.

A láncolt ábrázolás esetén is fennáll az, hogy a verem minden művelete – függetlenül a verem méretétől – konstans időben végrehajtható:

$$T_{op}(n) = \Theta(1)$$

ahol *op* a verem-műveletek bármelyikét jelentheti.

4.4. A 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 két blokkban, hanem váltakozva alkalmazzuk, akkor egy sorozatnak nem csak a megfordítottját tudjuk képezni, hanem számos átrendezését meg tudjuk valósítani. Itt most két jellegzetes alkalmazást mutatunk be.

4.4.1. Kifejezések lengyelformája

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. (Valójában fordított lengyelformáról kellene beszélnünk, de ez a jelző gyakran elmarad a napi szóhasználatban.)

Mind a *lengyelformára hozás*, mind pedig annak *kiértékelése* egy *vermes* algoritmus. Nézzük először a lengyelformára hozást. Egy aritmetikai kifejezés lengyelformájára a következők jellemzők:

- nincs benne zárójel,
- az operandusok sorrendje egymáshoz képest változatlan,
- minden műveleti jel közvetlenül az operandusai után áll.

Az utóbbi egyszerű állításban az, hogy egy műveleti jelről meg tudjuk állapítani, hogy a kifejezés mely egységei az operandusai, feltételezi az aritmetikai kifejezések felépítésének és értelmezésének ismeretét. Ezt előtanulmányaink sok éve alatt megbízhatóan elsajátítottuk. Néhány egyszerű, ám jellegzetes példán mutatjuk be a lengyelformát:

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

A példák alapján olyan *vermes algoritmust* tudunk létrehozni, amely megfelelően jár el az operandusokkal, a műveleti jelekkel, figyelembe véve precedenciájukat, illetve a zárójeleket is jól kezeli. Az algoritmust nem írjuk fel a szokásos formában, hanem csak a főbb pontjait fogalmazzuk meg az olyan kifejezésekre, mint amelyet a 4.10. ábrán láthatunk. Ezekben szerepelhet az *értékadás*, a *négy alapművelet* mellett a *hatványozás* jele is, valamint érvényesülhet a *zárójelek* módosító hatása.

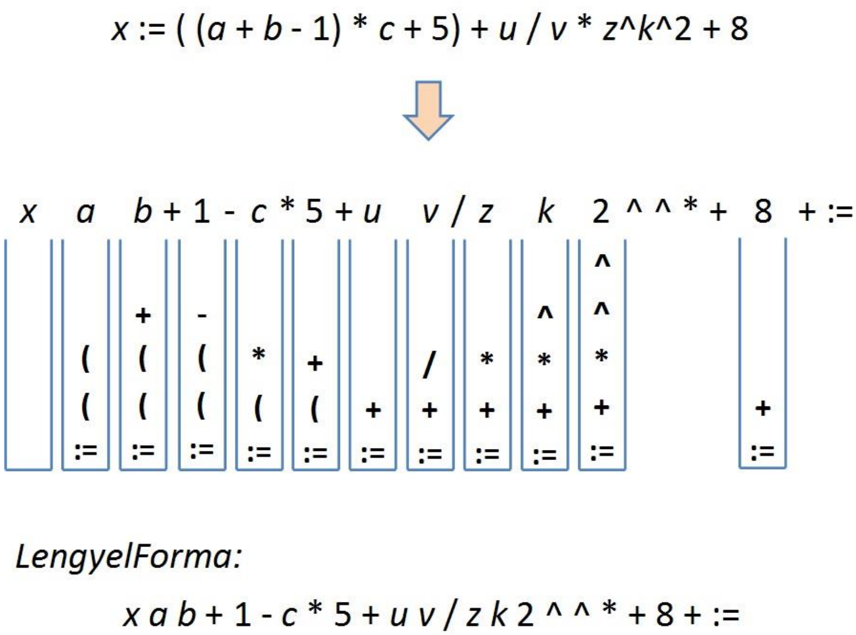
A műveleti jelek *precedenciája* a következő:

:=, ^, {*, /}, {+, -},

ahol a * és a /, valamint a + és a – műveleti jelek precedenciára rendre azonos. Ha két *azonos precedenciájú* művelet kerül egymás után, akkor általában *balról-jobbra* kiértékelési szabály érvényes, kivéve, ha *két hatványozás* követi egymást: ott *jobbról-balra* kell a hatvány értékét kiszámítani.

A vermes algoritmus inputja egy aritmetikai kifejezés, amelyet egységenként olvas. Outputja a kifejezés lengyelformája, amelyben az eredetitől eltérő sorrendben jelennek meg a műveleti jelek és zárójeleket nem tartalmaz. Az eljárás ezt egy *verem* használatával éri el. A vermes algoritmus alapvető jellemzői a következők:

- az *operandusok* (változók, számok) az inputról *közvetlenül* az outputra kerülnek, vagyis nem kerülnek be a verembe;
- a *nyitó zárójel* kötelezően beíródik a verembe, mintegy új vermet nyit az eredetin belül egy részkifejezés lengyelformájának létrehozására;
- a *csukó zárójel* a verem tartalmát az első nyitó zárójelig kiüríti, és a lengyelformába írja, majd a veremben lévő nyitó zárójelet kiveszi a veremből és „eldobja”;
- a *műveleti jelek* a precedencia-szabály figyelembe vételével a verembe íródnak: egy műveleti jel a nála kisebb precedenciájú műveleti jel fölé tehető a veremben, a nagyobb precedenciájút azonban ki kell előbb venni a veremből és a lengyelformába ki kell írni;
- *azonos precedenciák* találkozása esetén – a hatványjel kivételével – a bejövő műveleti jel csak úgy helyezhető a verembe, ha előtte a felső elemet kivesszük onnan; két hatványjel esetében fordítva járunk el: a bejövő a másik fölé bekerül a verembe;
- a *kifejezés vége* kiüríti a vermet és az elemeket a lengyelformába írja ki.



4.10. ábra. Kifejezés lengyelformára hozása

A lengyelformára hozás algoritmusának illusztrációja látható a 4.10. ábrán. A verem-tartalmakat mindig abban a pillanatban tünteti fel az ábra, amikor az operandusok kiírásra kerültek a lengyelformába.

4.4.2. Helyes zárójelezés feldolgozása (egymásba ágyazott folyamatok kezelése)

Az *egymásba ágyazott folyamatok* kezelésre példát nyújt az eljáráshívások láncolata a programokban. Ennek legegyszerűbb modellje a *helyes zárójelezés* feldolgozása. A nyitó zárójel egy folyamat kezdetét, míg a csukó zárójel a folyamat befejezését jelenti. Egy beágyazott folyamat elkezdése kor a befoglaló folyamat megáll, és csak a hívott folyamat befejeződése után folytatódik.

Definiáljuk először a *helyes zárójelezés* H nyelvét. Kétféle meghatározást is adunk.

1. Definíció: A helyes zárójelezések $H \subset \{ (,) \}^*$ nyelvére teljesülnek az alábbiak:

(1) $\varepsilon \in H$
(2) Ha $h \in H$, akkor $(h) \in H$
(3) Ha $h_1, h_2 \in H$, akkor $h_1h_2 \in H$

Hozzá szokták tenni, hogy csak az (1), (2) és (3) pontok alkalmazásával nyert sorozatok a helyes zárójelezések, más nem az.

2. Definíció: A helyes zárójelezések H nyelvét pontosan azok a $h \in \{ (,) \}^*$ sorozatok alkotják, amelyekre a következő két feltétel teljesül:

(1) a nyitó és a csukó zárójelek száma a teljes sorozatban megegyezik;
(2) a sorozat bármely kezdőszeletében legalább annyi nyitó zárójel található, mint amennyi csukó zárójel fordul elő.

A definíció formálisan is felírható:

$$h \in H \Leftrightarrow l_((h) = l_) (h) \wedge \forall u \in Pre\,(h)\text{: } l_((u) \geq l_) (u)$$

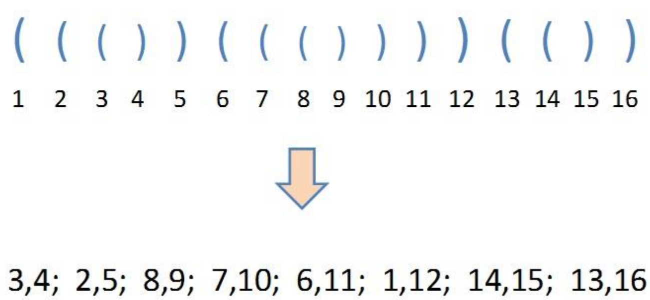
Az általánosan elterjedt $l\,(s)$ jelölés az s sorozat hosszát (a benne lévő karakterek számát) jelöli, ennek általánosításaként bevezetjük a $l_x(s)$ jelölést:

$l_x(s)$: 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).

Megfogalmazunk egy egyszerű feladatot, amely erősen egyszerűsített formában az egymásba ágyazott folyamatok kezelésének a lényegét tartalmazza. Ez nem más, mint egy folyamat kezdetének és a befejezésének az összepárosítása. Egy folyamat befejezése esetén ugyanis meg kell keresnünk a folyamat elkezdésére utaló bejegyzést és azt törölni kell. A folyamatok kezdetét és befejezését absztrakt formában egy nyitó és csukó zárójelpár azonosítja.

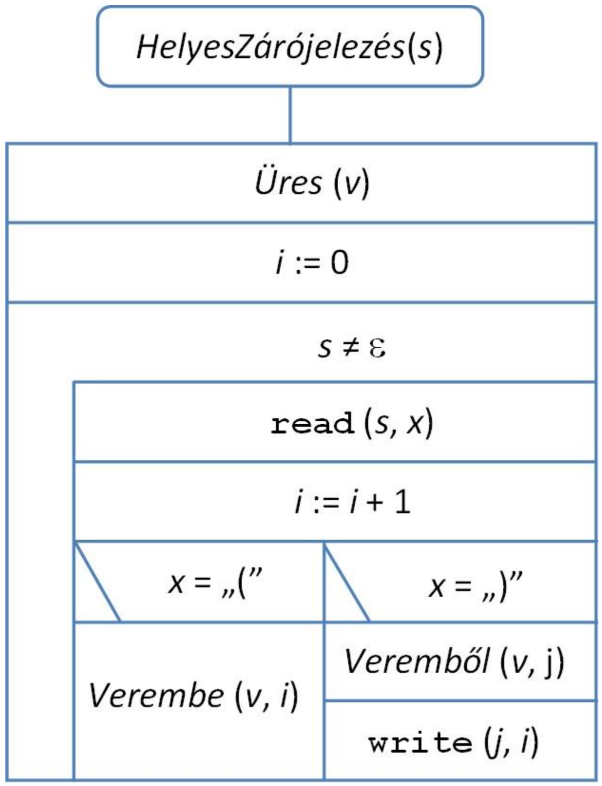


4.11. ábra. Helyes zárójelezés: összetartozó nyitó és csukó zárójelpárok

Feladat: Adott egy olyan input karaktersorozat, amely garantáltan helyes zárójelezést tartalmaz. Azonosítsuk az összetartozó nyitó és csukó zárójeleket olyan módon, hogy egymás után, párban írjuk ki az összetartozó zárójelpárok sorszámait.

A 4.11. ábrán egy példát láthatunk az összetartozó nyitó és csukó zárójel-párok azonosítására.

Ezek után megfogalmazzuk a helyes zárójelezés feldolgozásának algoritmusát, amely a 4.12. ábrán látható.



4.12. ábra. Helyes zárójelezés: összetartozó zárójelpárok meghatározása

A nyitó zárójeleket – pontosabban annak sorszámát – mindig beírjuk a verembe. Ha csukó zárójelet olvasunk, akkor a verem tetején találjuk a hozzá tartozó nyitó zárójel sorszámát, amelyet kiveszünk a veremből és a csukó zárójel sorszámával együtt kiírjuk az outputra.