Egy adott indexű elem törlésének algoritmusa az elem felszabaduló helyének megfelelően az elem mögött állókat feltömöríti, egy hellyel előre lépteti. Az algoritmusban használni fogjuk a DEC függvényt, melynek hatása az, hogy csökkenti eggyel az argumentuma értékét. Ennek révén a hossz és a vége attributumokat korrigáljuk. Legrosszabb esetben az összes megmaradt elemet mozgatni kell, ezért az időigény $T(n)=\Theta(n)$.

	3.1.3. algoritmus **Törlés tömbből**
	// $T(n)=\Theta(n)$
1	TÖRLÉS_TÖMBBŐL (*A, x, hibajelzés*)
2	// Input paraméter: *A* - a tömb
3	// *x* – a törlendő tömbelem indexe, ha a tömb nem üres és az *x* index létező elemre mutat. A hátrébb álló elemeket egy hellyel előre léptetjük. A tömb megrövidül.
4	// Output paraméter: *hibajelzés* - a beszúrás eredményességét jelzi
5	//
6	**IF** ***hossz****[A]* $\neq 0$
7	**THEN** **IF** ***fej****[A]* $\leq x \leq$ ***vége****[A]*
8	**THEN** **FOR** $i \leftarrow x$ **TO** ***vége****[A]*-1 **DO**
	$A_i \leftarrow A_{i+1}$
	DEC(***hossz****[A]*)
	DEC(***vége****[A]*)
	hibajelzés: ← „sikeres törlés”
	ELSE *hibajelzés*: ← „nem létező elem”
14	**ELSE** *hibajelzés*: ← „üres tömb”
15	**RETURN** (*hibajelzés*)

A lineáris törlési időt konstansra csökkenthetjük, ha a tömbelemek eredeti sorrendjének megörzése nem fontos azáltal, hogy a törlésre ítélt elem helyére a tömb utolsó elemét helyezzük és a tömböt lerövidítjük.

Az eddigiek során nem használtuk ki, hogy a tömbben a kulcsok rendezetten (növekvő sorrend, vagy csökkenő sorrend) követik egymást. Nem is használhattuk ki, hiszen ezt nem tételeztük fel. Ez ismeretlen volt a számunkra. Most azonban élünk azzal a feltételezéssel, hogy a tömbelemek kulcsai növekvő sorrendben vannak. Mit nyerünk ezzel az információval? A műveletek meg kell, hogy őrizzék ezt a rendezettségi állapotot. Vegyük őket sorra!

A keresés időigénye, amely rendezettlen tömbben lineáris volt, feljavítható logaritmikusra az úgynevezett bináris keresés révén. A bináris keresés alapötlete az, hogy a *k* kulcs keresésekor a kulcsösszehasonlítást a tömb középső elemének kulcsával kezdjük. Ha egyezést tapasztalunk, akkor az eljárás véget ér. Ha a *k* kulcs értéke kisebb, mint a megvizsgált elem kulcsa, akkor a tömb középső elemének indexétől kisebb indexű elemek között folytatjuk a keresést. Ha a *k* kulcs értéke nagyobb, mint a megvizsgált elem kulcsa, akkor a tömb középső elemének indexétől nagyobb indexű elemek között folytatjuk a keresést. A méret ezáltal feleződött. A további keresés ugyanilyen elv alapján megy tovább. Minden lépésben vagy megtaláljuk a keresett kulcsú elemet, vagy fele méretű résztömbben folytatjuk a keresést. Ha a résztömb mérete (hossza) zérusra zsugorodik, akkor a keresett kulcs nincs a tömbben.

Megadjuk a rekurzív algoritmust is és az iteratívat is. Mindkettőben a felezést *nyílt index-intervallumra* végezzük, ami azt jelenti, hogy az index-intervallum végek nem tartoznak a keresési index-intervallumhoz. Ez az ötlet jó hatással van az algoritmus szerkezetére. Az iteratív esetben az algoritmus bemenő paraméterei természetes módon a tömb és a keresett kulcs, az algoritmus az egész tömbben keres. A rekurzív algoritmus bemenő paraméterei a rekurzivitás sajátosságai révén szintén természetes módon a tömb, a keresési nyílt index-intervallum két vége valamint a keresett kulcs. Tehát alaphelyzetben ez az algoritmus csak a tömb egy összefüggő részében keres, nem a teljes tömbben. Ha a teljes tömbben akarunk keresni, akkor az algoritmust a

BINÁRIS_KERESÉS_TÖMBBEN (A, ***fej***[*A*] - 1, ***vége***[*A*] + 1, *k*)

sorral kell aktivizálni. Itt feltételeztük, hogy a tömb nem üres.

3.1.4. algoritmus
Bináris keresés tömbben (rekurzív változat)

	// $T(n)=\Theta(\log n)$
1	BINÁRIS_KERESÉS_TÖMBBEN (*A, i, j, k, x*)
2	// Input paraméter: *A* - a tömb
3	// *i* a keresési nyílt intervallum kezdőindexe. A keresésben ez az index még nem vesz részt.
4	// *j* a keresési nyílt intervallum végindexe. A keresésben ez az index már nem vesz részt.
5	// *k* – a keresett kulcs
6	// Output paraméter:: *x* - a k kulcsú elem indexe. **NIL**, ha a kulcs nincs a tömbben.
7	//
8	$x \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
9	**IF** *i* = *x*
10	**THEN** *x* ← **NIL**
11	**ELSE IF** *k* = ***kulcs***[A_x]
12	**THEN RETURN** (*x*)
13	**ELSE IF** *k* > ***kulcs***[A_x]
14	**THEN** BINÁRIS_KERESÉS_TÖMBBEN (*A, x, j. k, z*)
15	**ELSE** BINÁRIS_KERESÉS_TÖMBBEN (*A, i, x, k, z*)
16	*x* ← *z*
17	**RETURN** (*x*)

3.1.5. algoritmus
Bináris keresés tömbben (iteratív változat)

	// $T(n)=\Theta(\log n)$
1	BINÁRIS_KERESÉS_TÖMBBEN (*A, k, x*)
2	// Input paraméter: *A* – a tömb
3	// *k* – a keresett kulcs
4	// Output paraméter: *x* - a *k* kulcsú elem indexe. **NIL**, ha a kulcs nincs a tömbben.

5	//
6	**IF** ***hossz****[A]*=0
7	**THEN** $x \leftarrow$ **NIL**
8	**ELSE** $i \leftarrow$ ***fej****[A]* - 1,
9	$j \leftarrow$ ***vége****[A]* + 1,
10	$x \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
11	**WHILE** $i < x$ **DO**
12	**IF** ***kulcs***$[A_x] = k$
13	**THEN RETURN** (x)
14	**IF** $k >$ ***kulcs***$[A_x]$
15	**THEN** $i \leftarrow x$
16	**ELSE** $j \leftarrow x$
17	$x \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
18	**RETURN** (x)

A másik két művelet hatékonyságára a rendezettség nincs jótékony hatással. A beszúrásnál a helycsinálás továbbra is lineáris ideig fog tartani, és ez lesz a helyzet a törlésnél is a tömörítés idejével.

3.2. A láncolt lista (mutatós és tömbös implementáció)

Egy másik lehetőség a sorozat implementálására a *láncolt lista*.

Definíció: A láncolt lista adatstruktúra
A *láncolt lista* (linked list) olyan dinamikus halmaz, melyben az objektumok, elemek lineáris sorrendben követik egymást. A lista minden eleme mutatót tartalmaz a következő elemre. Műveletei: keresés, beszúrás, törlés.

A láncolt listáról nem feltételezzük, hogy az egymást követő elemei a memóriában valamilyen szabályszerűségnek megfelelően követik egymást. Az elemek lehetnek bárhol, az egyes elemeket mutatón keresztül érhetjük el, nem az index révén.. Ha valamely elemet – például az elsőt - megtaláltuk, akkor a rákövetkezőt is megtaláljuk a benne lévő mutató alapján. Az elem számára helyet foglalni elegendő csak a keletkezése pillanatában. Megszünésekor helyét felszabadíthatjuk. Ennek feltétele, hogy elérhető legyen a dinamikus memória gazdálkodás. A láncolt listák a mutatók és a kulcsok alapján osztályozhatók az alábbi egymást ki nem záró szempontok szerint.

Láncoltság: **Egyszeresen láncolt**, ha csak egy mutató van minden elemben (előre láncolt). A listán az elejétől végig lehet menni a végéig. **Kétszeresen láncolt**, ha van visszafelé mutató mutató is. Ekkor a lista végétől is végig lehet menni a listán az elejéig.

Rendezettség: **Rendezett** a lista, ha a kulcsok valamilyen sorrend szerint (növekvő, csökkenő) követik egymást. **Nem rendezett** a lista, ha a kulcsok sorrendjében nincs rendezettség.

Ciklikusság: **Ciklikus** a lista, ha listavégek elemeinek mutatói nem jelzik a véget, hanem a lista másik végelemére mutatnak. **Nem ciklikus**, ha a listavégi elemek jelzik a lista véget.

Az *L* listának, mint struktúrának az attributuma a ***fej****[L]*, ami egy a lista első elemére mutató mutató. Ha ***fej****[L]* = **NIL**, akkor a lista üres. A listaelemek attributumai az alábbi táblázatban következnek. A tárgyalásban kétszeresen láncolt, nem rendezett, nem ciklikus listáról van szó. A listaelemet az x mutató révén érhetjük el.

Attributum	Leírás
kulcs*[x]*	az elem kulcsa
elő*[x]*	Az *x* mutató által mutatott elemet megelőző elemre mutató mutató. Ha ***elő****[x]* = **NIL**, akkor az *x* mutató által mutatott elem a lista eleje.
köv*[x]*	Az *x* mutató által mutatott elemet követő elemre mutató mutató. Ha ***köv****[x]* = **NIL**, akkor az *x* mutató által mutatott elem a lista vége.

Feladat: Készítsünk lineáris idejű algoritmust a listaelemek sorrendjének megfordítására $\Theta(1)$ mennyiségű további memória felhasználásával. A listán csak egyszer menjünk végig!

Tekintsük át most a műveletek pszeudokódjait. Elsőként a keresés. A keresésnél a listafej információ alapján a lista kezdetét megtaláljuk és a ***köv*** mutatók révén végig tudunk lépkedni a listaelemeken, miközben vizsgáljuk a kulcsok egyezőségét. A legrosszabb eset az, amikor az elem nincs a listában. Ilyenkor minden elem vizsgálata sorrakerül és ez adja a lineáris időt.

	3.2.1. algoritmus **Láncolt listában keresés**
	// $T(n)=\Theta(n)$
1	LISTÁBAN_KERES (*L, k, x*)
2	// Input paraméter: *L* – a lista
3	// *k* – a keresett kulcs
4	// Output paraméter: *x* - a kulcselem pointere. **NIL**, ha a kulcs nincs a listában.
5	//
6	$x \leftarrow$ ***fej****[L]*
7	**WHILE** $x \neq$ **NIL** és ***kulcs****[x]* $\neq k$ **DO**
8	$x \leftarrow$ ***köv****[x]*
9	**RETURN** (*x*)

A beszúrást konstans idő alatt azáltal tudjuk elvégezni, hogy az új elemet mindig a lista legelső eleme elé szúrjuk be. Ekkor mindegy, hogy hány elem van a listában, a beszúrási idő ugyanannyi.

	3.2.2. algoritmus **Láncolt listába beszúrás (kezdőelemként)**
	// $T(n)=\Theta(1)$
1	LISTÁBA_BESZÚR (*L, x*)
2	// Input paraméter: *L* – a lista
	// *x* a beszúrandó elemre mutató mutató

3	// Az új elemet a lista elejére teszi
4	//
5	***köv**[x]* ← ***fej**[L]*
6	***elő**[x]* ← **NIL**
7	**IF** ***fej**[L]* ≠ **NIL**
8	**THEN** ***elő**[**fej**[L]]* ← *x*
9	***fej**[L]* ← *x*
10	**RETURN**

Szintén megoldható konstans idő alatt a beszúrás a lista tetszőleges eleme elé, vagy mögé, amennyiben az een elemre mutató mutató meg van adva. Ha a mutató helyett az elem kulcsa van megadva, akkor lineáris idejű a beszúrás, mivel a beszúrás tényleges elvégzése előtt az elemet meg kell keresni a listában.

Törlésnél az elem nem szűnik meg létezni, csak a listából kiláncolódik, a listán lépkedve többé nem érhető el. Elérhető viszont más úton akkor, ha valahol maradt valamilyen mutató, amely rá mutat. A konstans idő abból adódik, hogy az algoritmus input adataként a törlendő elem mutatóját adjuk meg, így az elemet nem kell keresni. Ha az inputban a törlendő elem kulcsa szerepelne, akkor a kiláncolás előtt előbb a kulcs alapján az elemet meg kellene keresni, ami miatt az idő lineárissá nőne.

3.2.3. algoritmus **Láncolt listából törlés**	
	// $T(n)=\Theta(1)$
1	LISTÁBÓL_TÖRÖL(*L*,*x*)
2	// Input paraméter: *L* – a lista
3	// *x* a törlendő elemre mutató mutató
4	//
5	**IF** ***elő**[x]* ≠ **NIL**
6	**THEN** ***köv**[**elő**[x]]* ← ***köv**[x]*
7	**ELSE** ***fej**[L]* ← ***köv**[x]*
8	**IF** ***köv**[x]* ≠ **NIL**
9	**THEN** ***elő**[**köv**[x]]* ← ***elő**[x]*
10	**ELSE** ***köv**[**elő**[x]]* ← **NIL**
11	**RETURN**

Feladat: Ez az algoritmus nem működik helyesen, ha a lista csak az egyetlen törlendő elemet tartalmazza. Hogyan módosítanánk, hogy akkor is helyesen működjön?

Némi kényelmetlenséget jelent a lista kezelésénél a listavégek állandó vizsgálata, valamint hogy a végeken az algoritmus lépései eltérnek a középen alkalmazott lépésektől. Ennek a feloldása úgynevezett *szentinel* (őrszem, strázsa) alkalmazásával megoldható.

Legyen ***nil**[L]* egy mutató, amely az alábbi szerkezetű elemre mutat:

elő	***kulcs***	***köv***
A lista végére mutat	Speciális, „érvénytelen szerkezetű”	a lista elejére mutat

Ez az elem testesíti meg a nem létező **NIL** elemet. Ezzel az elemmel tulajdonképpen egy ciklikus listát valósítunk meg, melyben egy olyan kulcsú elem van, amelyről azonnal eldönthető, hogy a valódi listához nem tartozhat hozzá. Bármilyen irányban haladunk a listán, mindig jelezni tudjuk a kulcs vizsgálata révén, hogy a lista elejére, vagy a végére értünk. Ennek a listának a sajátossága, hogy ***köv[nil[L]]*** a lista első elemére mutat, ***elő[nil[L]]*** pedig az utolsó elemére. A lista utolsó eleme esetében ***köv***[*x*] = ***nil***[*L*], az első eleme esetében pedg ***elő***[*x*] = ***nil***[*L*]. A ***fej***[*L*] attributumra nincs szükség! Ennek megfelelően az algoritmusaink az alábbi módon változnak, egyszerűsödnek.

3.2.4. algoritmus **Szentineles listában keresés**	
	// $T(n)=\Theta(n)$
1	SZENTINELES_LISTÁBAN_KERES (*L*, *k*, *x*)
2	// Input paraméter: *L* – a lista
3	// *k* – a keresett kulcs
4	// Output paraméter: *x* - a kulcselem pointere. ***nil***[*L*], ha a kulcs nincs a listában.
5	//
6	*x* ←***köv[nil***[*L*]]
7	**WHILE** *x* ≠ ***nil***[*L*] és ***kulcs***[*x*] ≠ *k* **DO**
8	*x* ← ***köv***[*x*]
9	**RETURN** (*x*)

3.2.5. algoritmus **Szentineles listába beszúrás**	
	// $T(n)=\Theta(1)$
1	SZENTINELES_ LISTÁBA_BESZÚR (*L*, *x*)
2	// Input paraméter: *L* – a lista
3	// *x* a beszúrandó elemre mutató mutató
4	// Az új elemet a lista elejére teszi
5	//
6	***köv***[*x*] ← ***köv[nil***[*L*]]
7	***elő[köv[nil***[*L*]]] ← *x*
8	***köv[nil***[*L*]] ← *x*
9	***elő***[*x*] ← ***nil***[*L*]
10	**RETURN**

3.2.6. algoritmus **Szentineles listából törlés**	
	// $T(n)=\Theta(1)$
1	SZENTINELES_LISTÁBÓL_TÖRÖL (*L*, *x*)
2	/// Input paraméter: *L* – a lista
3	// *x* a törlendő elemre mutató mutató
4	//
5	***köv[elő***[*x*]] ← ***köv***[*x*]
6	***elő[köv***[*x*]] ← ***elő***[*x*]
7	**RETURN**

A rendezett lista rendezettségi tulajdonságait sajnos nem tudjuk kihasználni algoritmus gyorsításra. Ha a dinamikus memória gazdálkodás nem elérhető, vagy nem kívánunk vele élni, vagy egyszerűen nem vonzódunk a mutatók használatához (bár ez utóbbi nem úri passzió kérdése egy informatikai rendszer kifejlesztésekor), akkor a láncolt lista adatstruktúra realizálható, implementálható tömbbel is. Ekkor minden tömbelemet kiegészítünk „mutató" mezőkkel, amelyek valójában tömbelem indexeket fognak tárolni. A listafej is egy különálló, egész számot tároló rekesz lesz, amely megmutatja, hogy a tömb melyik eleme számít a lista kezdő elemének. A műveletek realizálásának pszeudokódjában a beszúrásnál most azt is figyelni kell, hogy van-e még fel nem használt rekesz a tömbben, vagyis, hogy a rendelkezésre álló memória korlátos. Természetesen a valódi mutatók használatakor is figyelni kell a memóriát, hiszen minden új elem megjelenése új memóriaigényt támaszt. A tömbös realizációhoz javasolható a következő séma. Legyen a *fej* annak a rekesznek a neve, melyben a lista első elemének a tömbelem indexe található. A **NIL** mutatót szimbolizálja a zérus. A listaelemek tárolására rendelkezésre bocsátott tömb neve legyen *A*, elemeinek indexelése induljon 1-től és tartson *maxn*-ig. Minden tömbelem, mint rekord álljon a kulcsmezőből, az információs mezőkből, valamint a „mutató" mezőkből (előző elemre és a következő elemre mutatók). A zérus realizálja itt is a **NIL** mutatót.

1. Példa: Álljon a listánk a következő kulcsú elemekből: 423, 356, 764, 123, 987, 276, 839. Tömbös realizácóval a következőképpen nézhet ki egy ilyen láncolt lista egy tömbben, amelynek mondjuk 10 eleme van. A rekordokból (sorokból) az információs mezőket kihagyjuk, mert a tárgyalás szempontjából lényegtelenek.

fej 3

index	*A*		
	elő	***kulcs***	***köv***
1	6	764	7
2			
3	0	423	6
4			
5	7	987	9
6	3	356	1
7	1	123	5
8			
9	5	276	10
10	9	839	0

Ha most a fenti listába be kellene szúrni a 247-es kulcsot, akkor ez többféle módon is megtehető. Be lehet szúrni - ha semmilyen megkötés sincs – az új elemet a lista elejére az első elem elé. Másik lehetőség, hogy megmondják, hogy például szúrjuk be a 356-os kulcsú elem mögé. Van ezeken kívül még sok lehetőség. Nézzük az említett két esetet. Az új elemet egyelőre helyezzük el mondjuk a tömb 2-es indexű helyén, ami jelenleg üres és a listához nem tartozik hozzá. Az első esetben mit kell változtatni? A lista első eleme az új elem lesz, tehát az ő ***elő*** mutatója zérus lesz, és a régi első elem ***elő*** mutatója az új elemre fog mutatni, tehát 2-re változik. Meg kell még adni az új első elem ***köv*** mutatóját, amely a régi első elemre mutat, tehát értéke 3 lesz, ami korábban a ***fej*** mutató volt. A ***fej*** mutatónak pedig az új első elemre kell mutatni, tehát értéke 2 lesz. Látható, hogy a művelethez a mutatókat illetően négynek a megváltoztatására volt szükség. Ezek után az új lista tömbös megjelenése alább látható a baloldali táblázatban. A változásokat vastagítva és aláhúzva jelenítettük meg. A másik esetben az új elem ***elő*** mutatója a 356-osra fog mutatni, amely a 6-os helyen van, a ***köv*** mutatója pedig

a 356-os ***köv*** mutatóját kapja, vagyis 1-et. A 356-os ***köv*** mutatója is és a 356 mögött korábban álló elem (a 764-es kulcsú) ***elő*** mutatója is az új elemre mutat, tehát mindkettő értéke 2. Vigyázni kell a mutatók megváltoztatásánál. A 764-es ***elő*** mutatóját hamarabb kell megváltoztatni, mint a 356-os ***köv*** mutatóját, mert ellenkező esetben a 356-os ***köv*** mutatója már nem a 764-esre mutat. Itt is négy mutató változott. Az eredmény az alábbi jobboldali táblázatban látható.

fej **2**	*A*		
index	***elő***	***kulcs***	***köv***
1	6	764	7
2	**0**	**247**	**3**
3	**2**	423	6
4			
5	7	987	9
6	3	356	1
7	1	123	5
8			
9	5	276	10
10	9	839	0

fej 3	*A*		
index	***elő***	***kulcs***	***köv***
1	**2**	764	7
2	**6**	**247**	**1**
3	0	423	6
4			
5	7	987	9
6	3	356	**2**
7	1	123	5
8			
9	5	276	10
10	9	839	0

2. Példa: Az 1. példabeli listából töröljük a 356-os kulcsú elemet. A kulcs alapján először az elemet meg kell keresni, hogy ismerjük a helyét (a tömbelem indexet). A ***fej***-töl elindulva az első elem a 3-as indexű, az ő kulcsa 423, ami nem jó nekünk. A 3-as ***köv*** mutatója 6, és a 6-os indexű kulcsa 356, amit kerestünk. Tehát a listából a 6-os indexűt kell törölni, ami a lista láncából történő kiláncolást jelenti, tehát maga az elem nem semmisül meg. A kiláncoláshoz megnézzük, hogy melyik a megelőző elem. Ez az ***elő*** mutató szerint a 3-as. Ezért a 3-as ***köv*** mutatóját lecseréljük a 6-os ***köv*** mutatójára, azaz 1-re, és a 6-os ***köv*** mutatója szerinti 1-es indexű elem ***elő*** mutatóját lecseréljük a 6-os ***elő*** mutatójára, azaz 3-ra. A törléshez tehát elegendő volt két mutatót megváltoztatni. Az eredmény az alábbi táblázatban látható:

fej 3	*A*		
index	***elő***	***kulcs***	***köv***
1	**3**	764	7
2			
3	0	423	**1**
4			
5	7	987	9
6	3	356	1
7	1	123	5
8			
9	5	276	10
10	9	839	0

3.3. A verem és az objektum lefoglalás/felszabadítás

Definíció: A verem adatstruktúra

A verem (stack) olyan dinamikus halmaz, amelyben előre meghatározott az az elem, melyet a TÖRÖL eljárással eltávolítunk. Ez az elem mindig az időben a

legutoljára a struktúrába elhelyezett elem lesz. Műveletei: beszúrás (push), törlés (pop).

Az ilyen törlési eljárást Utolsóként érkezett – Elsőként távozik (Last In – First Out, LIFO) eljárásnak nevezzük.

A verem jele *S*, attributuma a ***tető[S]***, amely egy mutató, a legutóbb betett (beszúrt) elemre mutat. Itt a tömbös realizációt mutatjuk be. A vermet egy *S* tömb valósítja meg. A kezdő tömbindex az 1-es. Az üres verem esetén a ***tető[S]*** tartalma zérus. Beszúrásnál a ***tető[S]*** tartalma nő eggyel, és az elem a ***tető[S]*** által mutatott indexű rekeszbe kerül. Törlésnél csak a ***tető[S]*** tartalma csökken eggyel elem nem semmisül meg. Figyelnünk kell, hogy lehetetlen esetben ne végezzük el a műveletet. Nincs beszúrás, ha betelt a tömb, nincs törlés, ha üres a verem. Érdemes ezeket a vizsgálatokat külön eljárásként megírni. Alább következnek a megfelelő pszeudokódok.

	3.3.1. algoritmus **Verem megtelt-e**
	// $T(n)=\Theta(1)$
	// tömbös realizáció
1	TELE_VEREM (*S*)
2	// Input paraméter: *S* – a vermet tároló tömb
3	// Az algoritmus IGAZ-at ad vissza, ha a verem megtelt és HAMIS-at, ha nem
4	//
5	**IF** ***tető[S]=tömbméret[S]***
6	**THEN RETURN** (IGAZ)
7	**ELSE RETURN** (HAMIS)

	3.3.2. algoritmus **Verembe beszúrás (~push)**
	// $T(n)=\Theta(1)$
	// Tömbös realizáció
1	VEREMBE (*S, x, hibajelzés*)
2	// Input paraméter: *S* – a vermet tároló tömb
3	// *x* – a beszúrandó elem
4	// Output paraméter: *hibajelzés* - jelzi az eredményességet
5	//
6	**IF** TELE_VEREM (*S*)
7	**THEN** *hibajelzés*←„túlcsordulás”
8	**ELSE** INC(***tető[S]***)
9	$S_{\textbf{\textit{tető}}[S]} \leftarrow x$
10	*hibajelzés*←„rendben”
11	**RETURN** (*hibajelzés*)

	3.3.3. algoritmus **Verem üres-e**
	// $T(n)=\Theta(1)$
	// tömbös realizáció
1	ÜRES_VEREM (*S*)
2	// Input paraméter: *S* – a vermet tároló tömb
3	// Az algoritmus IGAZ-at ad vissza, ha a verem üres és HAMIS-at, ha nem
4	//
5	**IF** ***tető****[S]*=0
6	**THEN RETURN** (IGAZ)
7	**ELSE RETURN** (HAMIS)

	3.3.4. algoritmus **Veremből törlés (~pop)**
	$T(n)=\Theta(1)$
	// Tömbös realizáció
1	VEREMBŐL (*S, x, hibajelzés*)
2	// Input paraméter: *S* – a vermet tároló Tömb
3	// Output paraméter: *x* – a kivett elem
4	// *hibajelzés* - jelzi az eredményességet
5	//
6	**IF** ÜRES_VEREM(*S*)
7	**THEN** *hibajelzés*←„alulcsordulás”
8	**ELSE** $x \leftarrow S_{tető[S]}$
9	**DEC(*tető*[*S*])**
	hibajelzés←„rendben”
10	**RETURN** (*x, hibajelzés*)

1. Példa: Helyezzük el egy kezdetben üres verembe a megadott sorrendben érkező 342, 416, 112 kulcsú elemeket, majd ürítsük ki a veremet.! A verem maximális mérete hat elem.

A verem feltöltése. Beszúrások (push)								A verem kiürítése. Törlések (pop)							
tető	0	***tető***	1	***tető***	2	***tető***	3	***tető***	2	***tető***	1	***tető***	0
1		1	342	1	342	1	342	1	342	1	342	1	342
2		2		2	416	2	416	2	416	2	416	2	416
3		3		3		3	112	3	112	3	112	3	112
4		4		4		4		4		4		4	
5		5		5		5		5		5		5	
6		6		6		6		6		6		6	

A törléseknél láthatóan az elemek nem törlődtek fizikailag, csak már nem elérhetők. Újabb beszúrások esetén természetesen felülíródnak az új elemmel és akkor végképpen elvesznek.

A verem realizálható olyan egyszeresen láncolt, nemrendezett, nemciklikus listával is, ahol a beszúrás mindig a lista első eleme elé történik, és a törlés mindig az első elemet törli.

Egy szép alkalmazása a veremnek és a listának együtt a memóriaterület (objektum) lefoglalása és felszabadítása. Legyen m rekeszünk az adatrekordok tárolására. Legyen $n < m$ rekesz lefoglalva listás szerkezettel. Szabadon áll $m - n$ rekesz további elemek számára. Ezeket a szabad rekeszeket egy egyszeresen láncolt listában tartjuk nyilván. A ***szabad*** *globális mutató* mutat az első szabad helyre. Mindig az első szabad helyet foglaljuk le, vagy a felszabadult hely a szabadok közé az első helyre kerül. Tehát a szabad rekeszek egy vermet alkotnak. A felszabadult rekesz a verembe kerül, rekesz igénylés esetén pedig a veremből elégítjük ki az igényt. Két művelet jelentkezik, az objektum lefoglalása és az objektum felszabadítása. Pszeudokódjaik:

3.3.5. algoritmus **Objektum lefoglalás**	
	// $T(n)=\Theta(1)$
1	OBJEKTUMOT_LEFOGLAL (*x*)
2	// Output paraméter: *x* a lefoglalt hely mutatója
3	//
4	$x \leftarrow$ ***szabad***
5	**IF** ***szabad*** $\neq$ **NIL**
	THEN ***szabad*** $\leftarrow$ ***köv****[x]*
7	**RETURN** (*x*)

3.3.6. algoritmus **Objektum felszabadítás**	
	// $T(n)=\Theta(1)$
1	OBJEKTUMOT_FELSZABADÍT(*x*)
2	// Input paraméter: *x* – mutató, amely a felszabadítandó elemre mutat
3	//
4	***köv****[x]* $\leftarrow$ ***szabad***
5	***szabad*** $\leftarrow x$
6	**RETURN**

2. Példa: Legyen adott 6 rekesz – egy hatelemű tömb - tárolási célra. A tömb kezdetben üres. Véezzük el a megadott sorrendben a felsorolt kulcsokkal a műveleteket. Ha a kulcs előtt egy T betű áll, akkor azt törölni kell, ha nem áll semmi, akkor be kell szúrni. Az elemeket kétszeresen láncolt listában tartjuk nyilván, a beszúrás az egyszerűség kedvéért történjen mindig a lista elejére és az objektum lefoglalás/felszabadítás vermes módszerét alkalmazzuk. A kulcsok felsorolása: 987, 654, 365, 247, T654, 123, T247, 235.

fej	0	***Szabad***	1
	elő	***kulcs***	***köv***
1			2
2			3
3			4
4			5
5			6
6			0

fej	1	***Szabad***	**2**
	elő	***kulcs***	***köv***
1	0	987	0
2			3
3			4
4			5
5			6
6			0

fej	2	***Szabad***	3
	elő	***kulcs***	***köv***
1	2	987	0
2	0	654	1
3			4
4			5
5			6
6			0

fej	3	***Szabad***	4
	elő	***kulcs***	***Köv***
1	2	987	0
2	3	654	1
3	0	365	2
4			5
5			6
6			0

fej	4	***Szabad***	5
	elő	***kulcs***	***köv***
1	2	987	0
2	3	654	1
3	4	365	2
4	0	247	3
5			6
6			0

fej	4	***Szabad***	2
	elő	***kulcs***	***köv***
1	3	987	0
2	3	654	5
3	4	365	1
4	0	247	3
5			6
6			0

fej	2	***Szabad***	5
	elő	***kulcs***	***köv***
1	3	987	0
2	0	123	4
3	4	365	1
4	2	247	3
5			6
6			0

fej	2	***Szabad***	4
	elő	***kulcs***	***köv***
1	3	987	0
2	0	123	3
3	2	365	1
4	2	247	5
5			6
6			0

fej	4	***Szabad***	5
	elő	***kulcs***	***köv***
1	3	987	0
2	4	123	3
3	2	365	1
4	0	235	2
5			6
6			0

3.4. A sor

Definíció: A sor adatstruktúra

A sor (queue) olyan dinamikus halmaz, amelyben előre meghatározott az az elem, melyet a TÖRÖL eljárással eltávolítunk és az az elem is amelyet a BESZÚR eljárással a halmazba beteszünk. Törlésre mindig az elemek közül a legrégebben beszúrt kerül. A beszúrt elem lesz a legfrissebb elem. Műveletek: beszúrás, törlés.

Az ilyen törlést Elsőként érkezik – Elsőként távozik (First In – First Out, FIFO) eljárásnak nevezzük. Ha az elemeket lineárisan egymás mellé felsorakoztatjuk az érkezésük sorrendjében, akkor szemléletesen mondhatjuk, hogy törlésre mindig az első helyen álló elem kerül, a beszúrás pedig az utolsó elemet követő helyre történik. A sornak, mint adatstruktúrának többféle realizációja lehetséges. Itt most a tömbös realizációval foglalkozunk. Legyen a tömb neve Q és legyen a ***tömbméret***$[Q]$ attributum értéke n, azaz a tömb n elemű. Az elemek indexelése legyen 1,2,3,...,n. Tömbös realizáció esetén a tömb a méreténél legalább eggyel kevesebb elemű sort képes csak tárolni. A sor attributumai:

Attributum	Leírás
fej$[Q]$	A sor első elemének a tömbelem indexe.
vége$[Q]$	A sor utolsó elemét követő tömbelem indexe. Az új elem ide érkezik majd, ha még van hely.

A sor elemei a tömbben a ***fej***$[Q]$, ***fej***$[Q]$+1, ***fej***$[Q]$+2,…,***vége***$[Q]$-1 indexű helyeket foglalják el. Ebben az index felsorolásban az indexek ciklikusan követik egymást, azaz az n index után az 1 index következik, ha erre szükség van. A műveletek szempontjából a törlés esetében nyilvánvaló, hogy üres sorból nem lehet elemet törölni. Az üres sor felismerhető abból, hogy ***fej***$[Q]$=***vége***$[Q]$. (Kezdetben ***fej***$[Q]$=***vége***$[Q]$=1.). Ha üres sorból veszünk ki elemet, az hiba (alulcsordulás). A beszúrás esetében nem szabad azt elvégezni, ha a sor már megtelt. Ez a jelenség felismerhető abból, hogy ***fej***$[Q]$=***vége***$[Q]$+1. A beszúrás tele sorba hibát eredményez (túlcsordulás). Nézzük a műveletek pszeudokódjait:

3.4.1 algoritmus
Sorba beszúrás

```
//                                        T(n) = Θ(1)
// Tömbös realizáció
1  SORBA ( Q, x, hibajelzés )
2  //Input paraméter: Q – a tömb
3  //                 x – a beszúrandó elem
4  //Output paraméter: hibajelzés - jelzi az eredményességet
5  //
6  IF   fej[Q]=vége[Q]+1
7       THEN   hibajelzés←„tele sor”
8              RETURN (hibajelzés)
9  Q[vége[Q]] ← x
10 IF   vége[Q] = tömbméret[Q]
```

3.4.2 algoritmus
Sorból eltávolítás

```
//                                        T(n) = Θ(1)
// Tömbös realizáció
1  SORBÓL ( Q, x, hibajelzés)
2  //Input paraméter: Q – a tömb
3  //Output paraméter: x - az eltávolított elem
4  //……:hibajelzés  jelzi az eredményességet
5  //
6  IF   fej[Q]=vége[Q]
7       THEN   hibajelzés← „üres sor”
               RETURN (hibajelzés)
8  x ← Q[fej[Q]]
9  IF   fej[Q] = tömbméret[Q]
```

11	**THEN** ***vége****[Q]* ← 1		8	**THEN** ***fej****[Q]* ← 1
12	**ELSE** INC(***vége****[Q]*)		9	**ELSE** INC(***fej****[Q]*)
13	*hibajelzés*←„sikeres beszúrás”		10	Hibajelzés←„sikeres eltávolítás”
14	**RETURN** (*hibajelzés*)		11	**RETURN** (*x, hibajelzés*)

1. Példa: Végezzük el az alábbi műveleteket egy üres sorból kiindulva. A következő felsorolásban egy kulcsérték annak beszúrását jelenti. A T betű a sorból eltávolítást szimbolizálja. A tömb 5 elemű. A felsorolás: 345, 231, 768, T, 893, T, 259, 478. Az eredmény alább látható. Minden egyes lépésben megmutatjuk a sor állapotát. Vastagítottuk a sorhoz hozzátartozó elemeket.

	kezdet	345	231	768	T	893	T	259	478
fej	1	1	1	1	2	2	3	3	3
vége	1	2	3	4	4	5	5	1	2
1		**345**	**345**	**345**	345	345	345	345	**478**
2			**231**	**231**	**231**	**231**	231	231	231
3				**768**	**768**	**768**	**768**	**768**	**768**
4						**893**	**893**	**893**	**893**
5								**259**	**259**

Sor realizálható láncolt listával is. Törölni mindig a lista első elemét töröljük, beszúrni pedig mindig a lista utolsó eleme után szúrunk be. Elegendő egyszeresen láncolt listával dolgozni, viszont ilyenkor érdemes a lista végének a mutatóját is külön tárolni. Ha kétszeresen láncolt listát használunk, akkor már a ciklikus lista használata a hatékonyabb és ekkor a listavég mutatót nem kell külön tárolni.