3. gyakorlat

Sor adattípus

A sor egy ***FIFO*** (***First In First Out***) adatszerkezet, azaz ellentétben a veremmel a legelőször behelyezett elemet vehetjük ki elsőként. Számos implementációja létezik a sornak, pl. statikus vagy dinamikus tömbbel (ld. előadáson, vagy a jegyzetben a 45-47. old.).

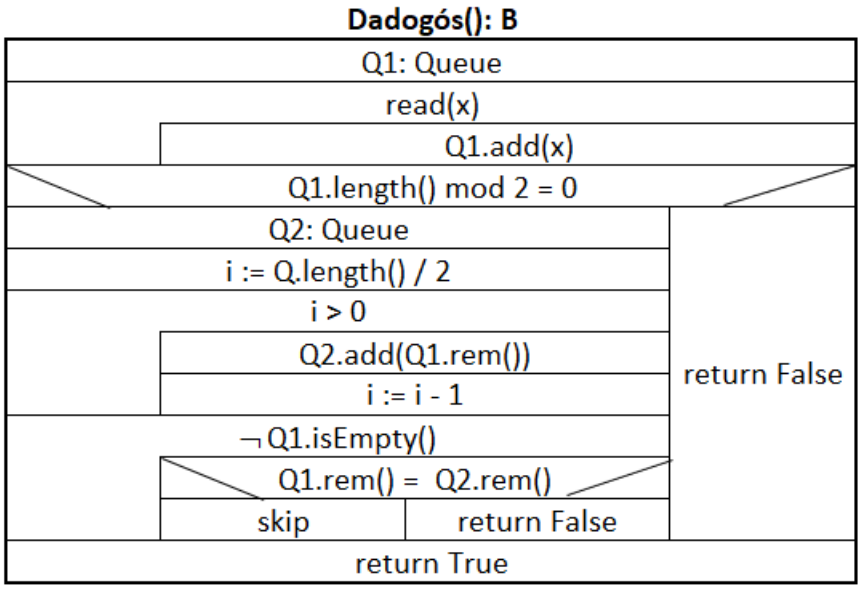
Példa sor alkalmazásra: 1. dadogós szöveg

Oldjuk meg *két sor* segítségével a következő feladatot:

Olvassunk be karakterenként egy szöveget (hossza nem ismert), és döntsük el, hogy „dadogós” –e.

Pl.: *abcabc* dadogós, *abccbb* nem dadogós

Az első *sorba* beolvassuk a teljes szöveget karakterenként. Ha a *sor* mérete páratlan, akkor a beolvasott szöveg biztosan nem „dadogós” ezért nem folytatjuk tovább a vizsgálatot. Ha az első *sor* mérete páros, akkor az első *sor* első felét átmozgatjuk a második *sorba*. Ezt követően egy közös ciklussal végig megyünk mindkét *soron* a ciklus minden lépésében kivesszük mindkét sorból az első elemet és összehasonlítjuk őket, ha nem egyeznek meg, akkor a szöveg nem „dadogós”.



Példa sor alkalmazására: 2. sorok összefésülése

Q1, Q2 sorokban egy-egy egynél nagyobb szám prímtényezős felbontása található növekvő sorrendben. Készítsünk egy függvényt, ami egy új sorba előállítja a legkisebb közös többszörös prímtényezős felbontását. Q1 sor lebontható, Q2 maradjon meg!

Trükk: Q2 végére szúrjunk egy ideiglenes „végjelet”, pl. -1-et, ezzel tudjuk vizsgálni, hogy hol van a sor vége.

Megoldás: Q2 sor végére teszünk egy -1 végjelet. A Q2 sor feldolgozása során az elejéről kiolvasott elemeket rendre a sor végére fűzzük, így a -1 után az algoritmus végére ismét helyes sorrendben meglesznek a Q2 eredeti elemei.

3. gyakorlat

Az algoritmus elágazásaiban kihasználjuk, hogy a ∧ és ∨ operátorok ***lusta kiértékelésűek***!

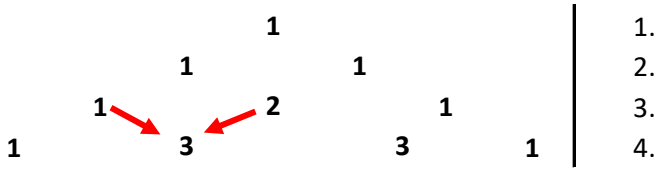
LKKT(Q1: Queue, Q2: Queue): Queue

<table>
<tr><td colspan="4">Q3: Queue</td></tr>
<tr><td colspan="4">Q2.add(-1)</td></tr>
<tr><td colspan="4">¬Q1.isEmpty() ∨ Q2.first() ≠ -1</td></tr>
<tr><td></td><td>Q2.first() = -1 ∨ (¬Q1.isEmpty() ∧ Q1.first() < Q2.first())</td><td>¬Q1.isEmpty() ∧ Q2.first() ≠ -1 ∧ Q1.first() = Q2.first()</td><td>Q1.isEmpty() ∨ (Q2.first() ≠ -1 ∧ Q1.first() > Q2.first())</td></tr>
<tr><td></td><td>Q3.add(Q1.rem())</td><td>Q3.add(Q1.rem())
Q2.add(Q2.rem())</td><td>Q3.add(Q2.first()
Q2.add(Q2.rem())</td></tr>
<tr><td colspan="4">Q2.rem() //-1 végjel eltüntetése</td></tr>
<tr><td colspan="4">return Q3</td></tr>
</table>

Sor átadása paraméterként: hasonlóan a tömbökhöz, a sor átadása értelemszerűen cím szerint történik, ezt nem jelzi külön az & jel! Így viszont a paraméterként kapott sor feldolgozás közben „kiürül”.

Szorgalmi házi feladat:

1 db sor és az összeadás művelet segítségével állítsuk elő a Pascal-háromszög k-adik sorát (feltehető, hogy k ≥ 1)!



Láncolt listák

Egy vagy két irányúak lehetnek.
Összehasonlítás a tömbbel:

- Előny: a rendezett beszúrás/törlés nem igényel elemmozgatást. Persze a beszúrás/törlés helyének megtalálása rendezett esetben O(n).
- Hátrány: nem indexelhető konstans műveletigénnyel, csak O(n)-nel!

Egyirányú lista

Listaelem típusa (jegyzetből):

E1
$+key : \mathcal{T}$ ... // satellite data may come here $+next$: **E1***
$+\mathbf{E1}()\ \{\ next := \oslash\ \}$

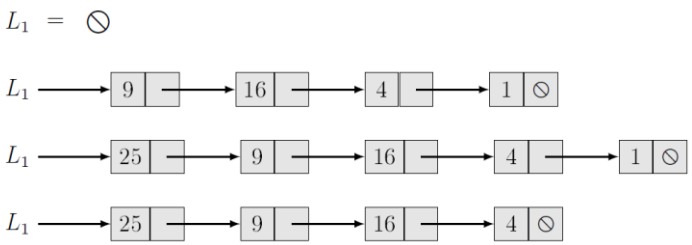
Bejárásához pointereket használunk: p,q: E1*

Elem adattagjainak elérése: p→key, p→next (Helyes még (*p).key, (*p).next)

Ha új listaelemet szeretnénk létrehozni: p = new E1,
feleslegessé vált listaelem felszabadítása: delete p (utána már nem hivatkozhatunk rá!)

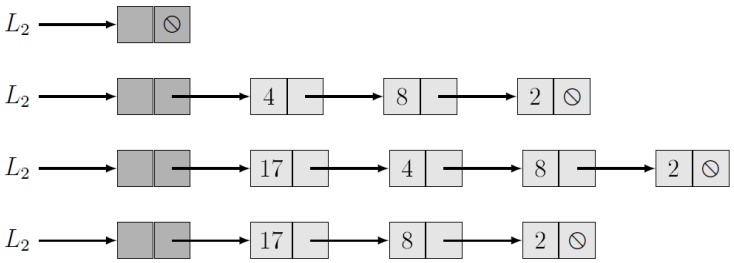
Egyirányú listák fajtái (jegyzetből)

1. **Egyszerű, egyirányú láncolt lista (S1L):** a legegyszerűbb forma, az első elemre egy pointer mutat. Ha még nincs eleme a listának, ez a pointer 0 (Null, Nil) értékű.



3. ábra. Az L_1 mutató egyszerű egyirányú listákat azonosít egy képzeletbeli program futásának különböző szakaszaiban. Az első sorban a lista, üres lista állapotában látható.

2. **Fejelemes egyirányú láncolt lista (H1L):** Gyakori trükk, hogy egy valódi adatot nem tároló elemet helyezünk el a lista elejére. Célja: a lista elején (vagy az üres listával) végzett műveletek megkönnyítése, mert így a lista elejére mutató pointerünk soha nem 0 értékű, továbbá az első elem előtt is van egy listaelem. (Léteznek végelemes listák is.)



4. ábra. Az L_2 fejelemes listákat azonosít egy képzeletbeli program futásának különböző szakaszaiban. Az első sorban a lista, üres lista állapotában látható.

Megemlíthető a ciklikus egyirányú lista is, de a feladatokban nem fogjuk használni.

Kétirányú listákkal majd a következő gyakorlaton foglalkozunk.

Feladatok egyirányú listák témakörben

1. feladat: S1L kezelésének bemutatása az alábbi feladaton keresztül:

Egy halmazt ábrázolunk egyszerű listával (egy adott kulcs csak egyszer fordulhat elő). Típus műveletek:

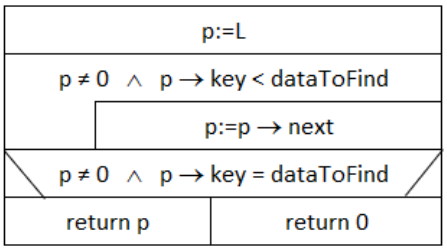
- egy elem benne van-e a halmazban: **keresés** kulcs alapján,
- egy elem hozzávétele a halmazhoz (ha még nincs benne): **beszúrás,**
- egy elem eltávolítása a halmazból (ha benne van): **törlés.**

3. gyakorlat

Gondoljuk meg a **rendezetlen** és **rendezett** ábrázolás közötti különbséget! Legyen most a listánk **növekedően rendezett**! Készítsük el a keresés, beszúrás, törlés algoritmusát:

Keresés:

findInS1L(L:E1*, dataToFind :T) : E1*



Sikertelen keresés eredménye: 0 pointert adunk vissza, így nem kell a logikai változó, amit a lineáris keresés tételénél tanultak.

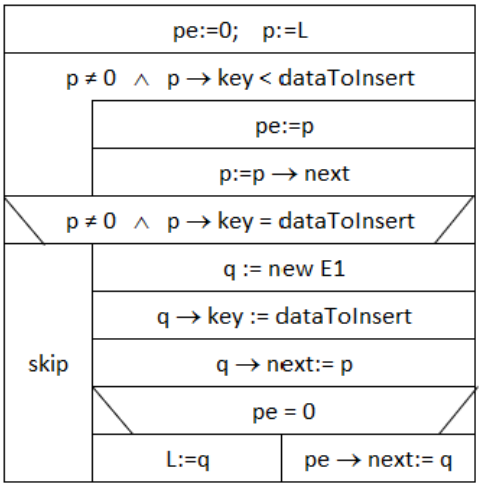
Beszúrás:

A kereséshez hasonló ciklussal indul, meg kell keresni a kulcs helyét a rendezett sorozatban. Rajzoljuk le az eseteket!

- Leggyakoribb, hogy a lista belsejébe szúrunk be. Ilyenkor mely elemek címe kell a művelethez?
- Változik-e valami, ha a lista utolsó eleme után fűzzük be az új elemet?
- Mennyiben más a lista elejére történő befűzés?
- Hogyan kell üres listába befűzni?

A műveletekhez világos, hogy két elem címe kell: az az elem, ami elé fogunk befűzni (az első olyan, melynek kulcsa nagyobb a beszúrandó kulcsnál), valamint az előtte lévő elem címe. Ehhez két bejáró pointert használunk. Megoldható persze egy bejáró pointerrel is. Egyirányú listák esetén, ha a lista felépítését megváltoztatja az algoritmus, kevesebb hibával jár, ha mindig két bejáró pointert használunk. Próbáljuk a fenti négy eset közös elemeit megtalálni! Kiderül, hogy csak az a különbség, hogy az az elem, amelyik az aktuális elem előtt van, az nem mindig létezik, így egyedül ezt kell majd egy elágazással kezelni.

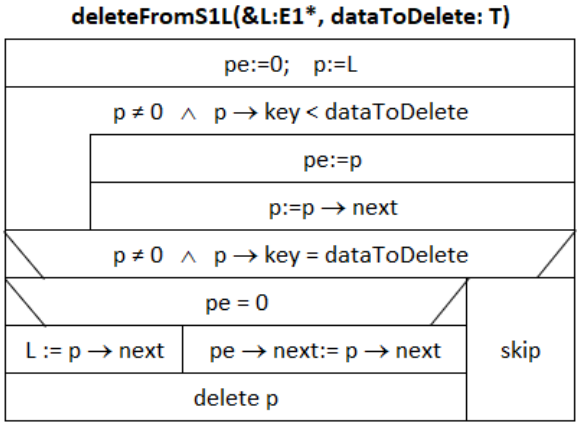
insertIntoS1L(&L: E1*, dataToInsert: T)



A két bejáró pointer: pe, és p.
A pe pointert 0-ról indítjuk; ez jelzi majd, hogy nincs még „előző” elem!

Ha már van k kulcsú elemünk, nem történik semmi.
A beszúrásnál csak az a lépés kerül elágazásba, amikor beszúrt elem előtti elemnek a next pointerét módosítjuk, ugyanis, ha nincs előző elem, akkor L módosul!
Mivel a műveletnél L pointer módosulhat, így fontos, hogy az cím szerint átvett paraméter legyen!

Törlés:

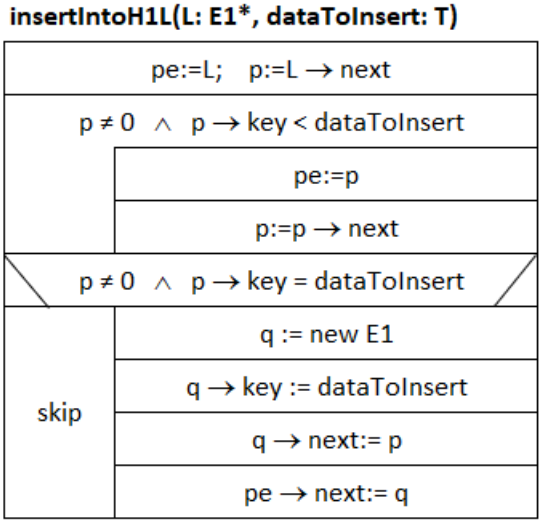


Hasonlóan a beszúráshoz, itt is pe és p a két bejáró pointer.
Ha az adott kulcsú elem nem található meg, akkor nem történik semmi.
Itt is fontos, hogy L cím szerinti paraméter.
A delete művelet a memóriaszivárgás elkerülése miatt fontos!

2. Nézzük meg fejelemes listára is (H1L) ugyanezeket a műveleteket!

Keresés: csak az indulás különbözik, a fejelemben nincs kulcs, így p:= L→next az induló lépés.

Beszúrás: indulásnál pe a fejelemre mutathat, hiszen az első elem előtt ott van mindig a fejelem, p pedig hasonlóan az előbbi algoritmushoz a lista első elemére mutat. Mivel pe nem lehet 0, nincs szükség az elágazásra a befűzésnél.



Fejelemes lista esetén a fejelem címét megadó paraméter nem cím szerint adódik át, hiszen a fejelem nem változtatja meg a címét, így az arra mutató pointer biztosan nem fog változni.

Törlés: indulás hasonlóan, itt sem kell elágazás a kifűzésnél, mert pe nem lehet 0 értékű.

Szorgalmi házi feladat

Egy rendezetlen egyirányú fejelemes listában (H1L) keressük meg az (egyik) legnagyobb kulcsú elemet, egyszeri bejárással! Üres lista esetén a NIL pointert adja vissza az algoritmus, nem üres lista esetén az elemet fűzze ki a listából, és címét adja vissza az algoritmus.