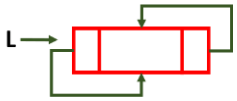
5. gyakorlat

Téma:

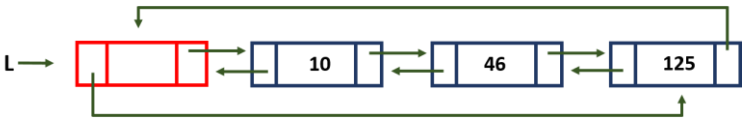
Fejelemes kétirányú ciklikus listák, sor láncolt megvalósítása, sor alkalmazása.

Fejelemes kétirányú ciklikus listák (C2L)

Alapvetően **C2L** alatt a fejelemes kétirányú ciklikus listát értjük. A C2L elemei két pointert tartalmaznak, az egyik (***prev***) az aktuális elemet megelőző, a másik (***next***) a következő elemre mutat. Használjuk a fejelemet is, a műveletek átláthatóbb és egyszerűbb megvalósítása miatt. A C2L ciklikus, azaz az utolsó elemének next pointere a fejelemre, a fejelem prev pointere pedig az utolsó elemre mutat. Az üres C2L lista egy fejelemből áll, melynek mindkét pointere magára a fejelemre mutat. Az ilyen listák elemeinek ábrázolásához az E2 nevű osztályt használjuk.



1. ábra: Üres C2L lista



2. ábra: C2L lista

Az E2 osztály[1]

(A vizsgán is lesz!)

E2
+ prev, next: E2* + key: T
+ E2() {prev = next = this} //*konstruktor*

Vegyük észre, hogy az elem konstruktora a pointereket úgy állítja be, hogy azok magára az elemre mutassanak! Ezt kihasználva egy új fejelemes C2L lista fejelemének létrehozása: L:= new E2 utasítással történhet!

A C2L listákhoz definiált műveletek[1]

- ***precede metódus***: listaelem beszúrása egy másik lista elem elé. A metódus első paramétere a beszúrandó listaelemre mutató pointer, a második paramétere arra a listaelemre mutató pointer, ami elé az első paramétert akarjuk beszúrni.
- ***follow metódus***: a precede metódushoz hasonló, itt most az első paraméterben lévő listaelem után szúrjuk be a második paraméterben lévő listaelemet.

[1] Dr. Ásványi Tibor jegyzete alapján

- ***unlink metódus***: a megadott listaelem kifűzése a listából. A metódus a kifűzött listaelem pointereit önmagára állítja.

precede(q, r : E2*)
// $(*q)$ will precede $(*r)$
$p := r \rightarrow prev$
$q \rightarrow prev := p$; $q \rightarrow next := r$
$p \rightarrow next := r \rightarrow prev := q$

follow(p, q : E2*)
// $(*q)$ will follow $(*p)$
$r := p \rightarrow next$
$q \rightarrow prev := p$; $q \rightarrow next := r$
$p \rightarrow next := r \rightarrow prev := q$

unlink(q : E2*)
// remove $(*q)$
$p := q \rightarrow prev$; $r := q \rightarrow next$
$p \rightarrow next := r$; $r \rightarrow prev := p$
$q \rightarrow prev := q \rightarrow next := q$

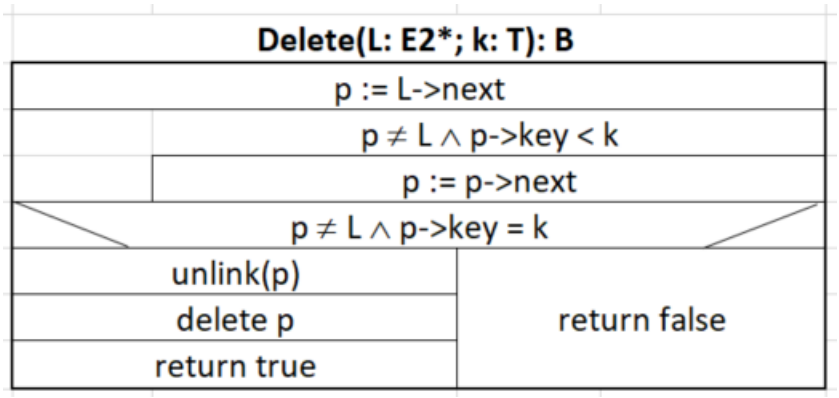
3. ábra: C2L listák alapműveletei

Ezeket a műveleteket a vizsgára tudni kell. Az algoritmusokban használt pointerek nevének logikája: ábécé sorrendben a három használt pointer: *p q r* ez mindig három egymás utáni listaelemet jelöl a listából, ebben a sorrendben. A két befűző műveletnél *q*-t fűzzük *p* és *r* közé, ehhez vagy *p*-t, vagy *r*-et adjuk meg paraméterként. Törlésnél elég *q*-t megadni, de itt is *p*-t és *r*-et használ a két szomszéd címének tárolásához!

Az ***előadás jegyzetben megtalálható a beszúró rendezés algoritmusa*** C2L listára, mely bemutatja ezen műveletek használatát, ezt otthon dolgozza fel mindenki! (Vizsgára tudniuk kell!)

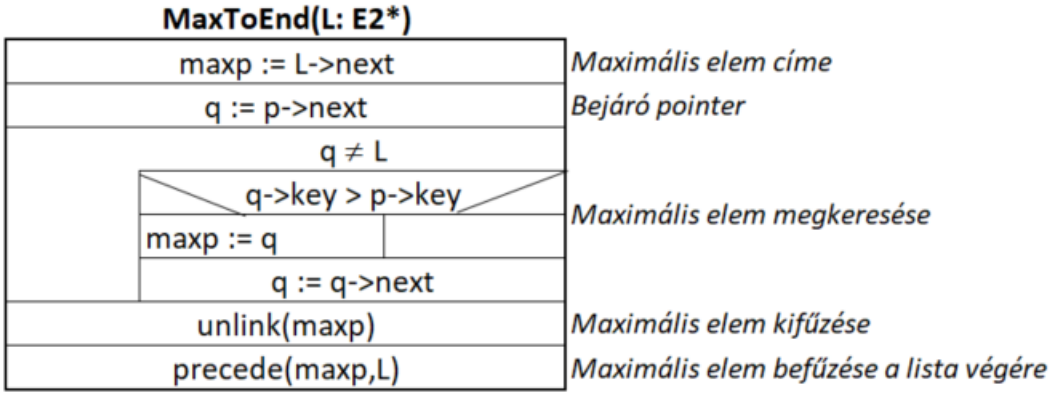
Törlés C2L listából

Készítsünk egy függvényt, ami a paraméterül kapott ***szigorúan monoton növekvően rendezett*** C2L listából törli a paraméterül kapott kulcsú elemet, amennyiben ilyen van. A függvény logikai értékkel tér vissza, ami a törlés sikerességét jelzi. Használjuk ki a lista rendezettségét!



Maximális elem a lista végére

Készítsünk egy eljárást, ami egy C2L lista maximális kulcsú elemét a lista végére fűzi. Tegyük fel, hogy a listában minden kulcs csak egyszer szerepel.



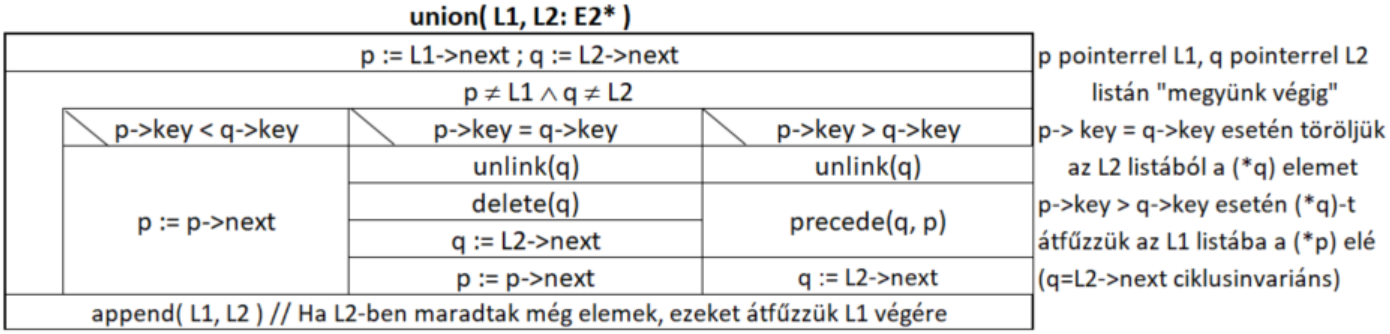
Érdekes meggondolni, hogy a fenti algoritmus helyesen működik-e üres lista, illetve egy elemű lista esetén? Mit fog csinálni?

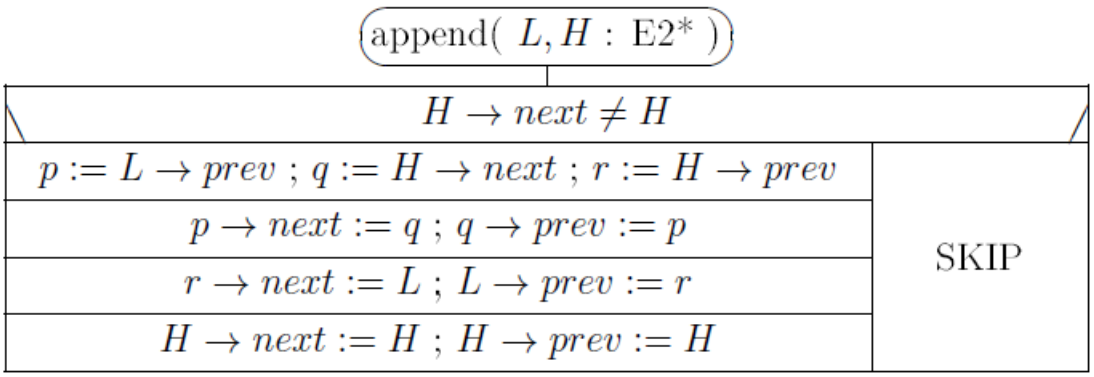
Halmazok uniója

Adott két szigorúan monoton növekően rendezett C2L lista (halmazt ábrázolnak): ***L1, L2***. L1-ben állítsuk elő a két halmaz unióját. L2 elemeit vagy átfűzzük, vagy felszabadítjuk. Mikor valamelyik lista végére érünk, akkor az összefésülő ciklusból kilépve, konstans számú lépésben fejezzük be az algoritmust!

A megoldás során ***összefuttatjuk*** a két listát, kihasználva azok rendezettségét. Az azonos kulcsú elemeket töröljük L2 listából. Ha az L1 lista végére érünk, de L2 listában még maradnak elemek, akkor az L2 elemeit (a fejelem kivételével) az L1 végére fűzzük, konstans számú lépésben.

Mivel L2 valamennyi elemét kiszedtük a listából, végül L2 fejelemének pointereit az üres listának megfelelően önmagára állítjuk.





Megjegyzések az algoritmushoz:

- ***p*** pointerrel L1, ***q*** pointerrel L2 listán iterálunk,
- ***p->key* = *q->key*** esetén a ***q*** című elemet kifűzzük L2 listából, majd felszabadítjuk, hiszen az unióban csak egyszer szerepelhet egy adott kulcsú elem,
- ***p->key* > *q->key*** esetén a ***q*** című elemet átfűzzük az L1 listába a ***p*** című elem elé.
- Az algoritmus végén az L2 lista maradék elemeit (ha vannak) az L1 lista végére fűzzük, és az L2 listát üresre állítjuk.

Kérdések, megjegyzések:

1. A fenti algoritmusban a *precede* helyett használhatjuk-e a *follow*-t, ha igen hogyan, ha nem miért nem? (Igen, p→prev és q paraméterekkel)
2. L2 lista megmaradt elemeinek L1 végére fűzéséhez miért nem használhatjuk a follow metódust? (Mert elvesztenénk L2 elemeit csak L2 első elemét fűzné L1 végére).
3. L2 lista végének átfűzéséhez szervezhetnénk egy ciklust, mely egyesével, az *unlink*-kel kifűzi az elemeket L2-ből, a *precede* segítségével pedig befűzi L1 végére. Viszont ekkor nem lenne konstans ennek a lépésnek műveletigénye. Mit mondhatunk a befejező lépés műveletigényről ilyenkor? (O(m))
4. Ha L1 lista hossza *n* és L2 lista hossza *m*, mit mondhatunk, mennyi lenne mT(*n*,*m*) és MT(*n*,*m*)? (mT(n,m) – egyik listán mindenképp végig iterálunk, így Θ(n), ha L2 minden eleme nagyobb L1 legnagyobb eleménél, vagy Θ(m), ha L2 minden eleme kisebb L1 legkisebb eleménél.
   Innét mT(n,m)∈Θ(min(n,m)).
   MT(n,m)∈Θ(n+m). (A legrosszabb esetben nincsenek azonos elemek, és L1 utolsó eleme nagyobb L2 valamennyi eleménél.)