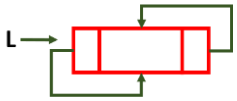
5. gyakorlat

Téma:

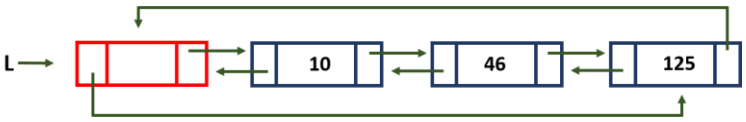
Fejelemes kétirányú ciklikus listák, sor láncolt megvalósítása, sor alkalmazása.

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]

(hangsúlyozzuk, hogy a vizsgán is lesz)

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

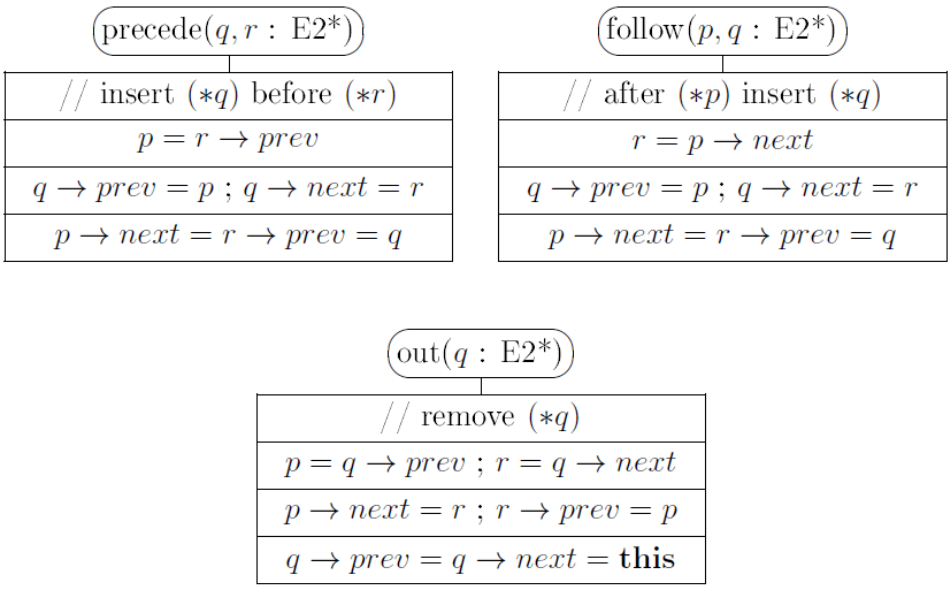
Vegyük észre, hogy az elem konstruktora a pointereket úgy állítja be, hogy azok magára az elemre mutassanak! Ezt kihasználva egy új fejelemes C2L lista fejelemének létrehozása: L:= new E2 utasítással történhet!

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

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



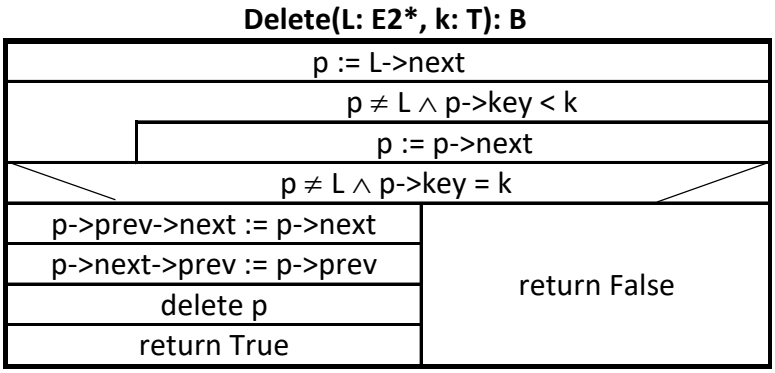
3. ábra: C2L listák alapműveletei

Ezeket a műveleteket a hallgatóknak a vizsgára **tudniuk kell,** így gyakorlaton vegyünk példákat a használatukra. 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, javasoljuk a hallgatóknak, hogy nézzék meg! (Vizsgára tudniuk kell!)

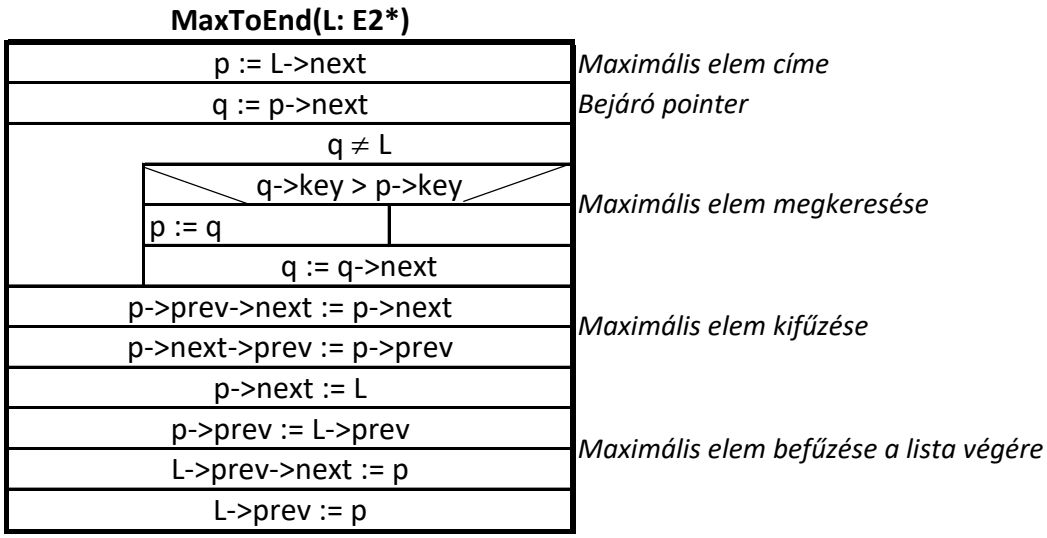
Törlés C2L listából

Készítsünk egy függvényt, ami a paraméterül kapott ***szigorúan monoton növekően 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!



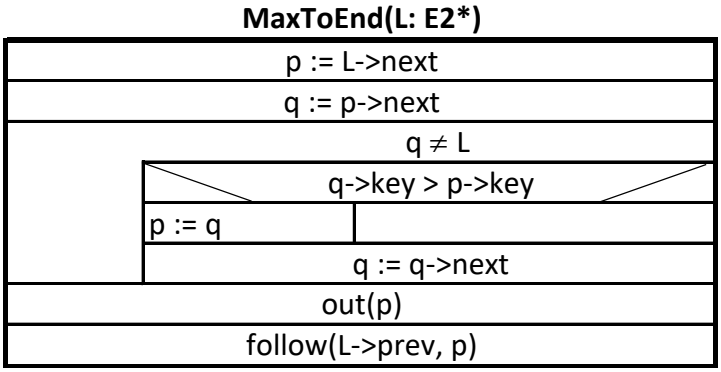
Maximális elem a lista végére

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? Egyik esetet végig követhetjük a csoporttal közösen.

A megoldás során használhatjuk a korábban ismertetett ***out és follow*** műveleteket. Alakítsuk át az algoritmust, hogy ezek meghívásával hajtsa végre a feladatot:



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. Ha végig értünk valamelyik listán, akkor az összefésülő ciklusból kilépve, konstans lépésben fejezzük be az algoritmust!

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

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

Unio(L1: E2*, L2: E2*)

p := L1->next		
q := L2->next		
$p \neq L1 \wedge q \neq L2$		
p->key < q->key	p->key = q->key	p->key > q->key
p := p->next	r := q->next	r := q->next
	q->prev->next := q->next	q->prev->next := q->next
	q->next->prev := q->prev	q->next->prev := q->prev
	delete q	p->prev->next := q
	q := r	q->prev := p->prev
	p := p->next	p->prev := q
		q->next := p
		q := r
$q \neq L2$		
L1->prev->next := L2->next		skip
L2->next->prev := L1->prev		
L2->prev->next := L1		
L1->prev := L2->prev		
L2->next := L2->prev := L2		

Megjegyzések az algoritmushoz:

- ***p*** pointerrel L1, ***q*** pointerrel L2 listán iterálunk,
- ***p->key = q->key*** esetén, q 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 L1 listába ***p*** elem elé,
- Az algoritmus végén L2 listát üresre állítjuk.

Az algoritmuson egyszerűsíthetünk, ha használjuk a C2L listához készült metódusokat:

Unio(L1: E2*, L2: E2*)

p := L1->next		
q := L2->next		
$p \neq L1 \wedge q \neq L2$		
p->key < q->key	p->key = q->key	p->key > q->key
p := p->next	r := q->next	r := q->next
	out(q)	**out(q)**
	delete q	**precede(q, p)**
	q := r	
	p := p->next	q := r
$q \neq L2$		
L1->prev->next := L2->next		skip
L2->next->prev := L1->prev		
L2->prev->next := L1		
L1->prev := L2->prev		
L2->next := L2->prev := L2		

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 átmásolásához szervezhetnénk egy ciklust, mely egyesével, az *out*-tal 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. MT(n,m)=Θ(n+m), ha nincsenek azonos elemek, és L1 utolsó eleme nagyobb L2 valamennyi eleménél.

Sor adattípus

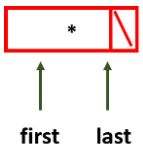
A sor egy ***FIFO*** (***First In First Out***) adatszerkezet, azaz ellentétben a veremmel a legelőször behelyezett elemet vehetjük ki elsőként. Számos implementációja létezik a sornak, pl. statikus vagy dinamikus tömbbel (ld. előadás).

Sor megvalósítása egyirányú listával

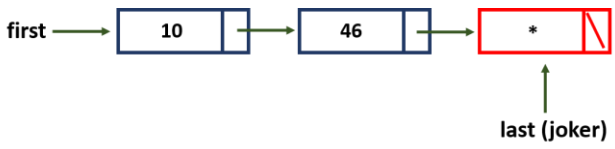
Kétféle módszert is be lehet mutatni a csoportnak. Természetesen elég az egyiket kidolgozni. A műveleteket már tanulták előadáson, és látták a tömbös megvalósítást.

Az első módszernél két pointert alkalmazunk. A lista első eleme a sornak is az első eleme, erre a *first* pointer mutasson. A lista végére (ez a sornak is a vége) mutasson a *last* pointer. Így a műveletek zöme konstans lépésszámú lesz. Kivéve persze a destruktort, és a setEmpty() műveletet, melyeknek le kell bontani a sort ábrázoló listát.

Ötlet: a lista mindig tartalmaz egy fix (***joker***) lista elemet, ami a sor végén fog elhelyezkedni. Új elem beszúrásakor a beszúrandó kulcsot a joker elemben tároljuk el, majd készítünk egy új üres joker elemet, amit a lista végére fűzünk. A joker elem címét tárolja a last pointer. A lista segítségével elméletileg korlátlan hosszúságú sort hozhatunk létre (amíg a new művelet sikeresen le tud futni).



4. ábra: Üres sor



5. ábra: Sor egyirányú listával ábrázolva

Queue[2]
-first, last: E1* //*a sor első és utolsó elemére mutató pointerek* -size: ℕ
+ Queue() + add(x: T) // *új elem hozzáadása a sor végére* + rem(): T // *a sor elején lévő elem eltávolítása* + first() : T // *a sor elején lévő elem lekérdezése* + length(): ℕ + isEmpty(): B + ~Queue() + setEmpty()

Az osztály metódusai

Queue::Queue()

first := last := new E1
first->next = null
size := 0

A konstruktor létrehozza a joker elemet.

Queue::rem(): T

	size = 0
Error	x := first->key
	s := first
	first := first->next
	delete s
	size := size - 1
	return x

Queue::add(x: T)

last->next := new E1
last->key := x
last := last->next
last->next := null
size := size + 1

Queue::first(): T

	size = 0
Error	return first->key

Queue::length(): N

return size

Queue::isEmpty(): B

return size = 0

[2] Veszprémi Anna és Dr. Ásványi Tibor jegyzete alapján

Queue::~Queue()

first ≠ null
p := first
first := first->next
delete p

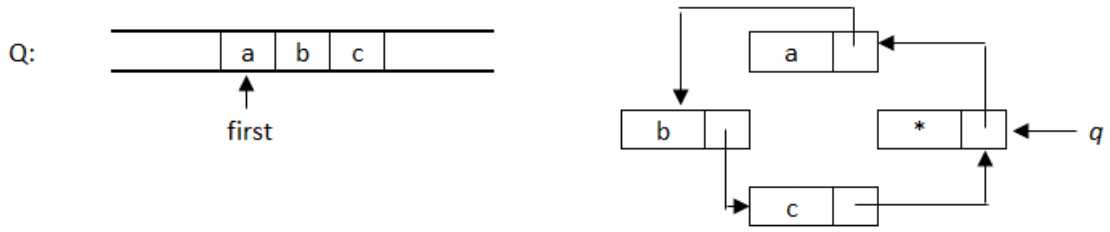
Queue::setEmpty()

first ≠ last
p := first
first := first->next
delete p
size:= 0

Sor megvalósítása speciális egyirányú listával. (haladó csoportoknál)

Ez egy másik, érdekes megvalósítás. Ugyanolyan hatékony, mint az előbbi, de csak egy pointert használ. Érdekes azért is, mert itt bemutatható az egyirányú ciklikus lista egy alkalmazása.

A sort egy egyirányú, ciklikus lista ábrázolja. Az előbb látott trükk itt is alkalmazható, hogy soha ne legyen teljesen üres, a listában mindig lesz egy joker elem, aminek még nincs tartalma. Az ezutáni elem lesz a sor első eleme. A sor műveleteinek megvalósításához elegendő ennek az egy joker elemnek a címét tárolni. A *rem* művelet kiveszi a joker elem utáni elemet a listából, a kivett elem next pointere kerül a joker elem next pointerébe. Az add művelet a joker elembe teszi az új elem adat részét, majd egy új joker elemet szúr be a listába q című elem után, és q pointert arra állítja rá.



Készítsük el a sor műveleteit ebben az ábrázolásban!

Queue[3]
-q: E1* //*a joker elemre mutató pointer* -size: ℕ
+ Queue() + add(x: T) // *új elem hozzáadása a sor végére* + rem(): T // *a sor elején lévő elem eltávolítása* + first() : T // *a sor elején lévő elem lekérdezése* + length(): ℕ + isEmpty(): B + ~Queue() + setEmpty()

[3] Veszprémi Anna és Dr. Ásványi Tibor jegyzete alapján

Queue::Queue()

q := new E1
q->next := q
size := 0

Queue::add(x: T)

r := q->next
q->key := x
q->next := new E1
q := q->next
q->next := r
size:= size+1

Queue::rem(): T

size = 0	
ERROR	r := q->next->next
	x := q->next->key
	delete q->next
	q->next := r
	size:= size-1
	return x

Queue::first(): T

size = 0	
ERROR	return q->next->key

Queue::length(): N

return size

Queue::isEmpty(): B

return size = 0

Queue::~Queue()

p := q->next	
p ≠ q	
	r := p
	p := p->next
	delete r
delete q	

Queue::setEmpty()

p := q->next	
p ≠ q	
	r := p
	p := p->next
	delete r
q->next := q	
size:= 0	

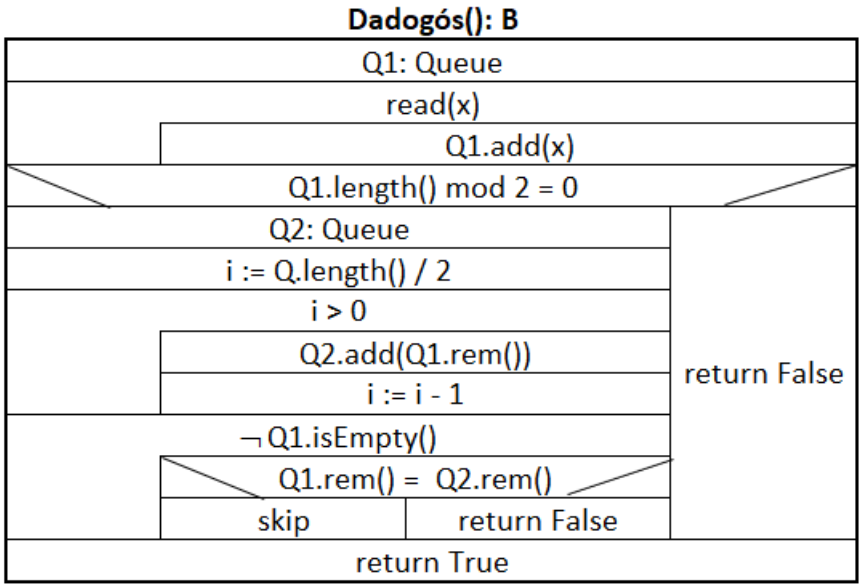
Példa sor alkalmazásra: dadogós szöveg

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”.



Javasolt házi feladatok:

1. Valósítsuk meg a sort a tanult C2L (fejelemes, ciklikus, kétirányú) listával! Alkalmazzuk a tanult listakezelő műveleteket (*out, precede, follow*). Törekedjünk a hatékonyságra!

Ebben az esetben elegendő egy az C2L fejelemére mutató pointert tárolnunk.

Queue
-first: E2* -size: ℕ
+ Queue() + add(x: T) // *új elem hozzáadása a sor végére* + rem(): T // *a sor elején lévő elem eltávolítása* + first() : T // *a sor elején lévő elem lekérdezése* + length(): ℕ + isEmpty(): B + ~Queue() + setEmpty()

Queue::Queue()

first := new E2
size := 0

Queue::add(x: T)

s := new E2
s->key = x
precede(s, first)
size:= size+1

Queue::rem(): T

size = 0	
Error	s:=first->next
	out(s)
	key:=s->key
	delete s
	size:=size-1
	return key

Queue::first(): T

size = 0	
Error	return first->next->key

Queue::length(): N

return size

Queue::isEmpty(): B

return size = 0

Queue::~Queue()

p := first->next	
p ≠ first	
	r := p->next
	out(p)
	delete p
	p := r
delete first	

Queue::setEmpty()

p := first->next	
p ≠ first	
	r := p->next
	out(p)
	delete p
	p := r
size:=0	

2. 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.

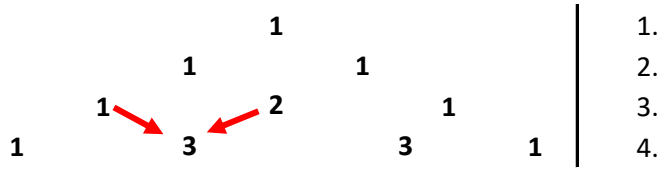
Az algoritmus elágazásaiban kihasználjuk, hogy a ∧ és ∨ operátorok ***lusta kiértékelésűek***!

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

Q3: Queue			
Q2.add(-1)			
¬Q1.isEmpty() ∨ Q2.first() ≠ -1			
	Q2.first() = -1 ∨ (¬Q1.isEmpty() ∧ Q1.first() < Q2.first())	¬Q1.isEmpty() ∧ Q2.first() ≠ -1 ∧ Q1.first() = Q2.first()	Q1.isEmpty() ∨ (Q2.first() ≠ -1 ∧ Q1.first() > Q2.first())
	Q3.add(Q1.rem())	Q3.add(Q1.rem())	Q3.add(Q2.first()
		Q2.add(Q2.rem())	Q2.add(Q2.rem())
Q2.rem() //*-1 végjel eltüntetése*			
return Q3			

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”.

3. 1 db sor és az összeadás művelet segítségével állítsuk elő a Pascal-háromszög k-adik sorát (feltehető, hogy k ≥ 1)!



Megoldás:

Pascal(k: N): Queue

```
Q: Queue
Q.add(1)                    Betesszük az első sorban található 1-est, így a háromszög első sora Q-ban van.
i := 2 to k                 A háromszög i. sora az i-1. sor segítségével számítható ki. Amikor a ciklusba
    e := 0                      belépünk Q-ban az i-1. sor elemei vannak.
    j := 1 to i-1           A Q sorból kiszedegetjük az elemeket, közben már állítjuk elő a háromszög i-dik sorát.
        Q.add(e + Q.first())    Kihasználjuk, hogy az i-1. sor pontosan i-1 elemből áll,
        e := Q.rem()            e-ben mindig az előző menetben kivett elem van, kezdetben pedig nulla.
    Q.add(1)                Még egy 1-est beteszünk Q végére, ezzel a háromszög i. sora Q-ban előállt.
return Q
```

4. L1, L2 C2L listák (halmazok), szigorúan monoton növekő számokat tartalmaznak. Készítsünk egy függvényt, amely visszaadja a két halmaz metszetét egy új listában. Az eredeti listákat ne változtassuk meg! A megoldáshoz használhatók a C2L listákhoz definiált metódusok.

Megoldás: az L1, L2 listák (halmazok) unióját előállító algoritmusból indulhatunk ki.

- ***p->key < q->key*** esetén a listák rendezettségéből tudjuk, hogy ***p->key*** kulcsú elem L2 listában nem szerepelhet, ezért biztosan nem része a metszetnek.
- ***q->key < p->key*** esetén ***q->key*** kulcsú elem nincs L1 listában, ezért q sem része a metszetnek.
- ***p->key = q->key*** esetén megtaláltuk a metszet egyik elemét. Ilyenkor először is egy új üres elemet készítünk, amit az új lista végéhez fűzünk. Azért van szükség új elemre, hogy L1 és L2 listák ne sérüljenek.

Intersect(L1: E2*, L2: E2*): E2*

<table>
<tr><td colspan="4">p := L1->next</td></tr>
<tr><td colspan="4">q := L2->next</td></tr>
<tr><td colspan="4">L3 := new E2</td></tr>
<tr><td colspan="4">$p \neq L1 \wedge q \neq L2$</td></tr>
<tr><td></td><td>p->key < q->key</td><td>p->key = q->key</td><td>p->key > q->key</td></tr>
<tr><td></td><td rowspan="5">p := p->next</td><td>r := new E2</td><td rowspan="5">q := q->next()</td></tr>
<tr><td></td><td>r->key := p->key</td></tr>
<tr><td></td><td>follow(L3->prev, r)</td></tr>
<tr><td></td><td>p := p->next</td></tr>
<tr><td></td><td>q := q->next</td></tr>
<tr><td colspan="4">return L3</td></tr>
</table>

Készült az „Integrált kutatói utánpótlás-képzési program az informatika és számítás-tudomány diszciplináris területein” című EFOP 3.6.3-VEKOP-16-2017-00002 azonosítójú projekt támogatásával.