6. gyakorlat

Téma:

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

Fejelemes kétirányú ciklikus listák (C2L)

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.

union(L1, L2: E2*)

p := L1->next ; q := L2->next			p pointerrel L1, q pointerrel L2 listán "megyünk végig"
p ≠ L1 ∧ q ≠ L2			
p->key < q->key	p->key = q->key	p->key > q->key	p-> key = q->key esetén töröljük az L2 listából a (*q) elemet
p := p->next	unlink(q)	unlink(q)	
	delete(q)	precede(q, p)	p->key > q->key esetén (*q)-t átfűzzük az L1 listába a (*p) elé
	q := L2->next		
	p := p->next	q := L2->next	(q=L2->next ciklusinvariáns)
append(L1, L2) // Ha L2-ben maradtak még elemek, ezeket átfűzzük L1 végére			

append($L, H : \mathrm{E2}^*$)

$H \to next \neq H$	
$p := L \to prev \; ; \; q := H \to next \; ; \; r := H \to prev$	SKIP
$p \to next := q \; ; \; q \to prev := p$	
$r \to next := L \; ; \; L \to prev := r$	
$H \to next := H \; ; \; H \to prev := H$	

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.

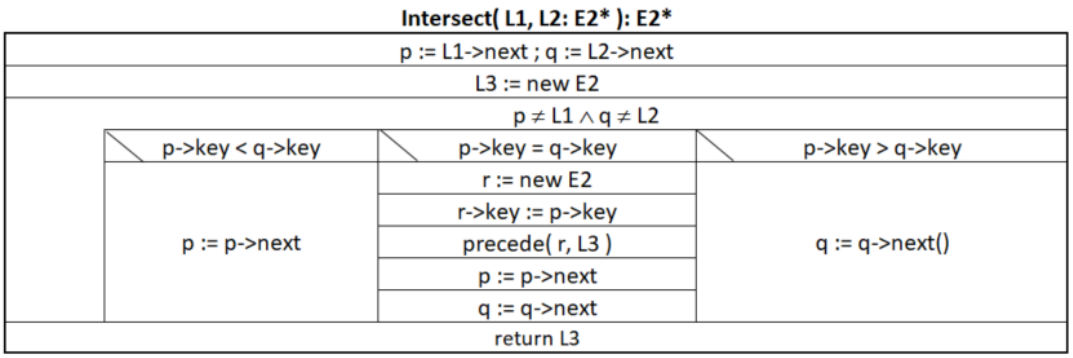
Kérdések, megjegyzések:

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.)

Halmazok metszete

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.



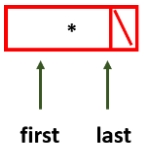
Sor adattípus

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).

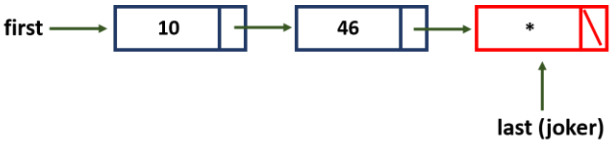
Sor megvalósítása egyirányú listával

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 végelemet (***trailer***, az ábrán „joker"), ami a sor végén fog elhelyezkedni. Új elem beszúrásakor a beszúrandó kulcsot a végelemben tároljuk el, majd készítünk egy új üres végelemet, amit a lista végére fűzünk. A végelem címét tárolja a last pointer. A lista segítségével elméletileg korlátlan hosszúságú sort hozhatunk létre (amíg a **new** művelet sikeresen le tud futni).



1. ábra: Üres sor



2. ábra: Sor egyirányú listával ábrázolva

Queue[1]
-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()

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

Az osztály metódusai

Queue::Queue()

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

Queue::first(): T

	size = 0
Error	return first->key

Queue::length(): N

return size

Queue::isEmpty(): B

return size = 0

Queue::~Queue()

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

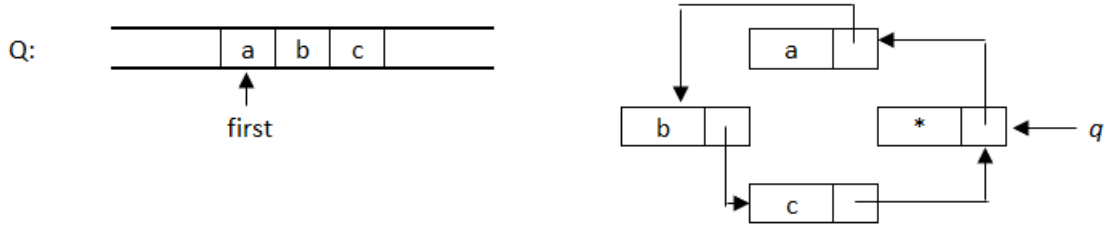
Queue::setEmpty()

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

Sor megvalósítása speciális egyirányú listával.

Ez egy másik, érdekes megvalósítás. Ugyanolyan hatékony, mint az előbbi, de csak egy pointert használ.

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) őrszem elem, ami egyben fej- és végelem, és aminek még nincs tartalma. Az ezutáni elem lesz a sor első eleme. A sor műveleteinek megvalósításához elegendő ennek az őrszem elemnek a címét tárolni. A *rem* művelet kiveszi az őrszem elem utáni elemet a listából, a kivett elem next pointere kerül az őrszem elem next pointerébe. Az add művelet a (joker) őrszem elembe teszi az új elem adat részét, majd egy új (joker) őrszem elemet szúr be a listába q című elem után, és q pointert arra állítja rá.



Készítsük el a sor műveleteit ebben az ábrázolásban!

Queue[2]
-q: E1* //*a joker/őrszem 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()

Queue::Queue()

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

Queue::add(x: T)

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

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

Queue::rem(): T

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

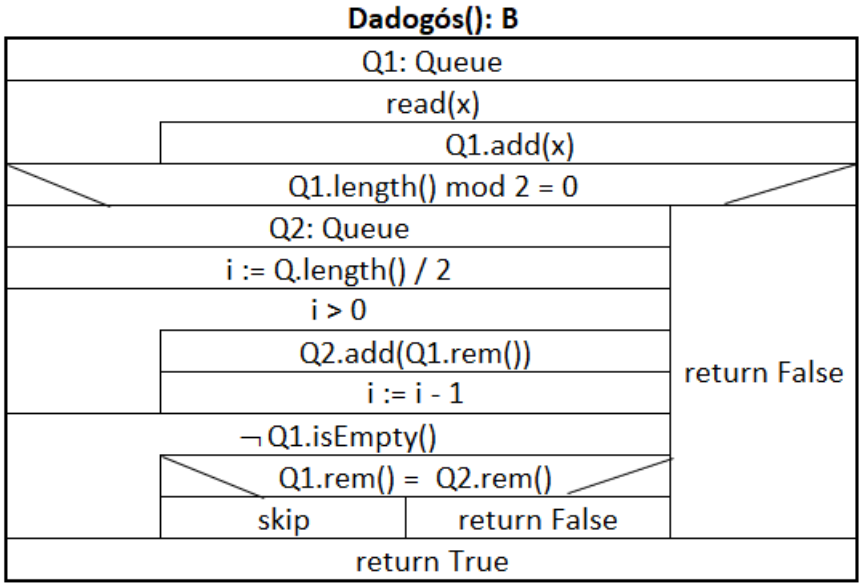
Példa sor alkalmazásra: dadogós szöveg

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”.



Házi feladatok:

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

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

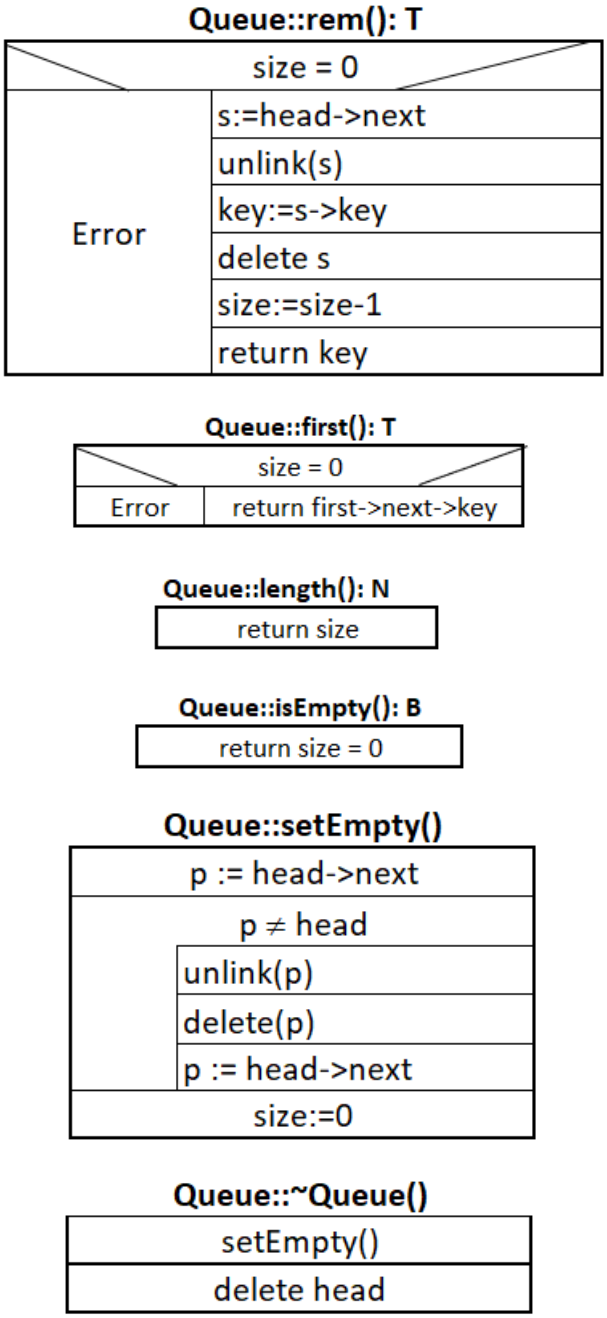
Queue
-head: E2* -size: ℕ
+ Queue() + add(x: T) // *új elem hozzáadása a sor végére* + rem(): T // *a sor elején lévő elem eltávolítása* + first() : T // *a sor elején lévő elem lekérdezése* + length(): ℕ + isEmpty(): B + ~Queue() + setEmpty()

Queue::Queue()

head := new E2
size := 0

Queue::add(x: T)

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



2. Q1, Q2 sorokban egy-egy egynél nagyobb szám prímtényezős felbontása található növekvő sorrendben. Készítsünk egy eljárást, ami a Q3 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.

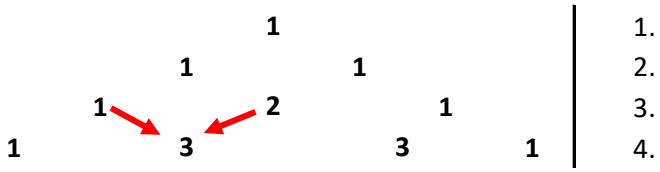
Az algoritmus elágazásaiban kihasználjuk, hogy a $\wedge$ és $\vee$ operátorok ***rövid kiértékelésűek***!

LKKT(Q1, Q2, Q3: Queue)

Q3.setEmpty()		
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*		

Sor átadása paraméterként: hasonlóan a tömbökhöz, a sor átadása értelemszerűen cím szerint történik, ezt nem jelzi külön az & jel! Így viszont a paraméterként kapott Q1 *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 ; Q: Queue)

Q.setEmpty()	
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.*