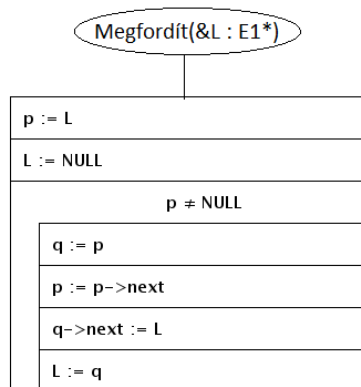


### 3. gyakorlat

Tematika: Feladatok láncolt listákkal. Sor típus láncoltan. C2L listák

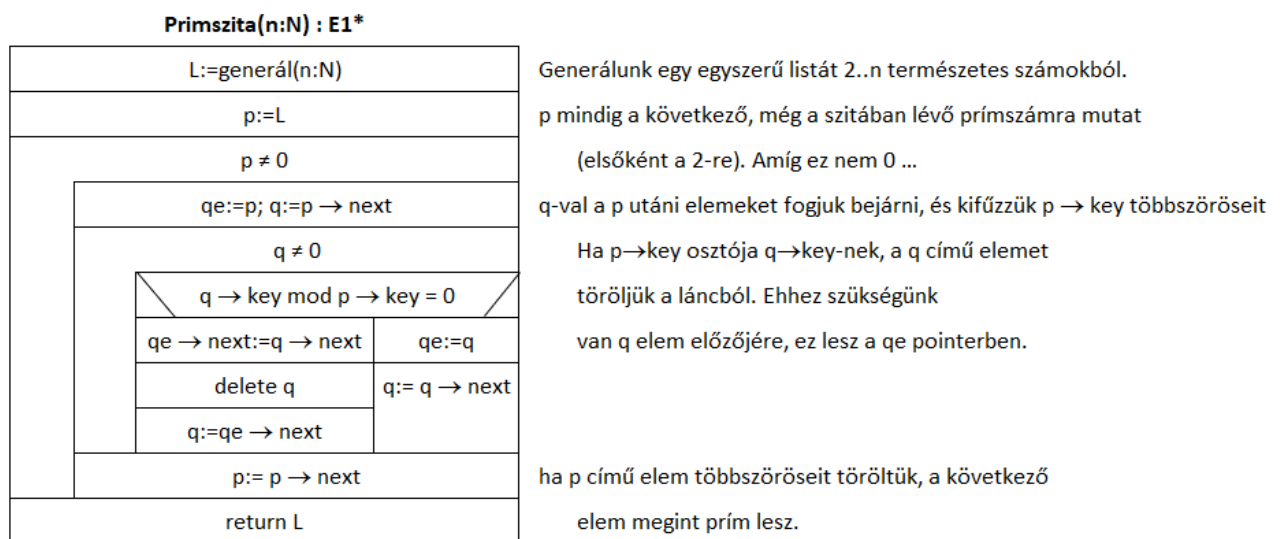
#### 1. feladat: Egyszerű lista megfordítása:

Trükk: Mintha egy verembe raknánk a lista elemeit, az eredmény listának mindig az elejére fűzzük be az eredeti lista aktuális elemét.



#### 2. feladat: Primszita egyszerű listán

Készítsünk egy listát, mely 2-től n-ig tartalmazza a prímeket, az Eratoszthenészi szita algoritmus ötletét felhasználva. Töltsünk fel egy egyszerű listát 2..n természetes számokkal. Az első elem prím, azokat, melyek oszthatók ezzel az elemmel, töröljük a listából. A következő megmaradt szám megint prím. Többszöröseit ismét töröljük. A végén a prímekek maradnak a listánkban.



### 3. gyakorlat

#### generál(n:N) : E1\*

L := 0
i = n downto 2
p := new E1
p → key := i
p → next := L
L := p
return L

Egyszerű lista feltöltése 2..n természetes számokkal.  
Trükk: a ciklust csökkenőleg futtatjuk, így mindig a lista elejére kell befűzni.

### 3. feladat: H1L szétfűzése két listába

**Feladat:** adott egy H1L, egész számokat tartalmaz, a fejelemre L1 mutat. Fűzzük ketté az elemeket: L1-ben maradjanak a páros elemek, egy új L2 H1L listába fűzzük át a páratlan elemeket. Az eredeti sorrendet tartsuk meg, azaz átfűzésnél mindig a lista végére kell majd fűzni. Az L1 listát egyszer lehet bejárni, L2-be a befűzés konstans műveletigényű legyen!

#### Szétfűz(L1:E1\*): E1\*

L2 := new E1    //L2 → next=0
u:=L2
pe:= L1; p:=L1 → next
p ≠ 0
p → key mod 2 = 0
pe:=p      pe → next:= p → next
p:= p → next      u → next:= p
p → next:= 0
u:=p
p:= pe → next
return L2

Első lépésként egy új H1L létrehozását mutatjuk be. Kihasználjuk, hogy E1 konstruktora az elem next pointerét 0-ra állítja!  
u mindig L2 utolsó elemére fog mutatni, ez kezdetben a fejelem.  
pointerek L1 bejárásához

Ha páratlan kulcsú elemmel találkozunk, a lépések:

p kifűzése

p befűzése L2 végére

mivel p az L2 utolsó eleme, 0-ra állítjuk a next pointerét

módosítjuk u-t

p-vel L1 következő elemére állunk

### Házi feladatok listák témakörben:

1. Egy rendezetlen egyirányú fejelemes listában (H1L) keressük meg a 2. legnagyobb elemet, egyszeri bejárással! Feltehető, hogy a listánk legalább két eleme van (a fejelemen kívül) és minden kulcs különböző.
2. Írjuk meg a MaxElem(L:E1\*):E1\* eljárást, ami az L pointerrel azonosított H1L típusú lista (egyik) legnagyobb kulcsú elemét kifűzi a listából, és a kifűzött elem címével tér vissza. Üres lista esetén NIL értékkel térjen vissza.

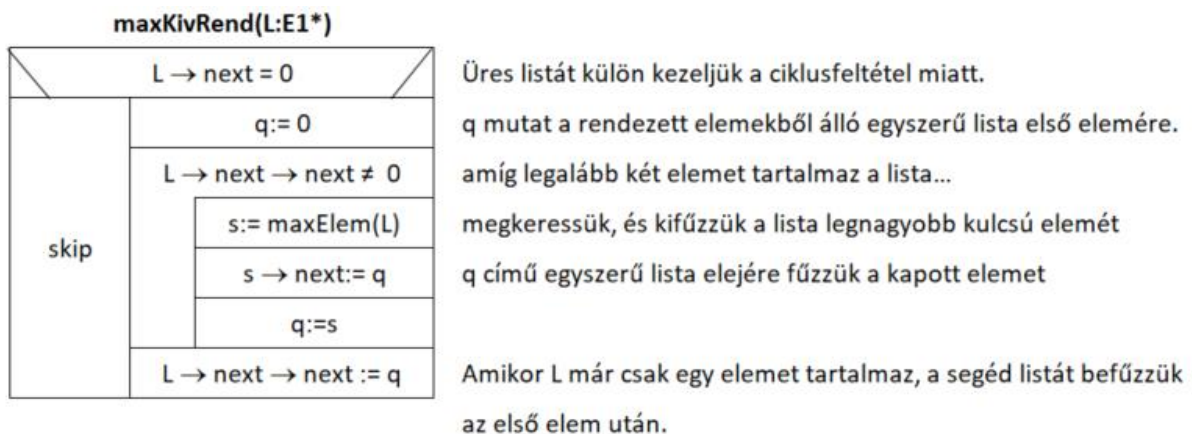
### 3. gyakorlat

#### 4. feladat: MaxKiv rendezés

Rendezzünk egy H1L listát maximum kiválasztó rendezéssel, ami használja a 2. feladat MaxElem eljárását. (Ügyeljünk, hogy a megoldó algoritmus üres listára is működjön!)

Megoldási ötlet: megkeressük és kifűzzük a lista legnagyobb elemét (üres lista esetét külön kell kezelni!). A kifűzött elemekből egy egyszerű listát kezdünk építeni úgy, hogy mindig a lista elejére fűzzük be a rendezendő listából kifűzött elemet, nevezzük az így kapott listát segéd listának. Ez a segédlista nyilvánvalóan növekvően rendezett lesz.

H1L lista rendezése maxKiválaszRendezéssel:



*Észrevehető, hogy egy elemű lista esetén a jobb ág gyorsan és helyesen lefut, így nem indokolt ennek külön esetként történő feldolgozása.*

### 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).

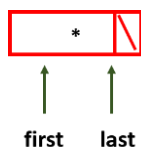
### Sor megvalósítása egyirányú listával

Kétféle módszert nézünk meg (a második kihagyható).

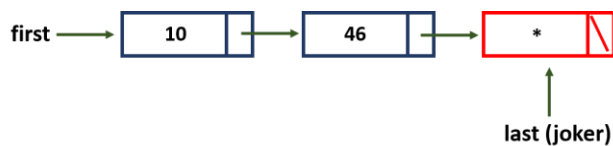
Az első módszernél két pointert alkalmazunk. A lista első eleme a sornak is az első eleme, erre a *first* pointer mutasson. A lista végére (ez a sornak is a vége) mutasson a *last* pointer. Így a műveletek zöme konstans lépésszámú lesz. Kivéve persze a destruktort, és a setEmpty() műveletet, melyeknek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

Ötlet: a lista mindig tartalmaz egy fix (**joker**) lista elemet, ami a sor végén fog elhelyezkedni. Új elem beszúrásakor a beszúrandó kulcsot a joker elembe 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).

### 3. gyakorlat



1. ábra: Üres sor



2. ábra: Sor egyirányú listával ábrázolva

Queue <sup>1</sup>	
-first, last: E1*	//a sor első és utolsó elemére mutató pointererek
-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

<sup>1</sup> Veszprémi Anna és Dr. Ásványi Tibor jegyzete alapján

### 3. gyakorlat

#### 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
-----------------

#### Queue::~~Queue()

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

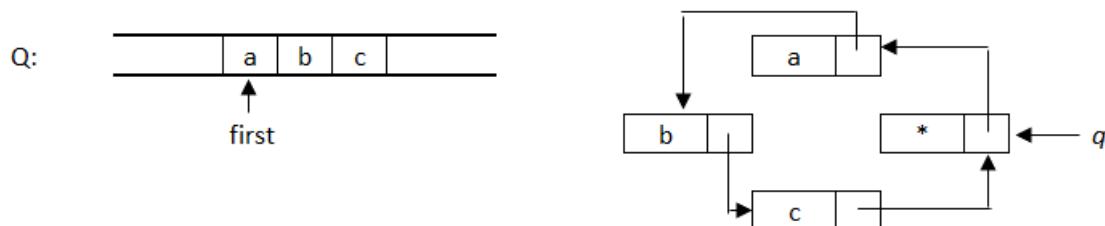
#### Queue::setEmpty()

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

### Sor megvalósítása speciális egyirányú listával (kihagyható)

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 pointerre 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.



### 3. gyakorlat

Készítsük el a sor műveleteit ebben az ábrázolásban!

<b>Queue<sup>2</sup></b>
-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()

#### **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**

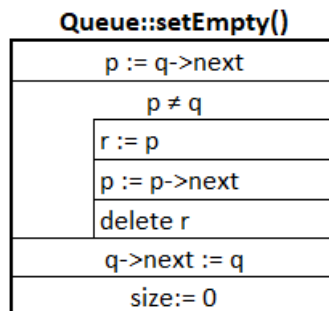
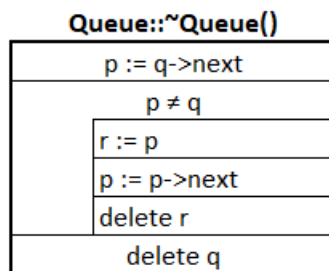
return size
-------------

#### **Queue::isEmpty(): B**

return size = 0
-----------------

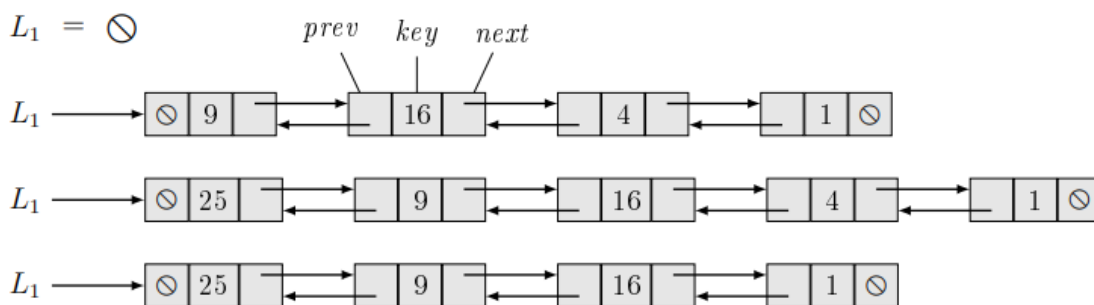
<sup>2</sup> Veszprémi Anna és Dr. Ásványi Tibor jegyzete alapján

### 3. gyakorlat



### Egyszerű kétirányú lista (S2L)

Jegyzet példája:



5. ábra. Az  $L_1$  mutató egyszerű kétirányú listákat (S2L) azonosít egy képzetbeli program futásának különböző szakaszaiban. Az első sorban a listát üresre inicializáló értékadás látható.

Hasonlóan az S1L listához a lista elejének, belsejének és végének kezelése különbözik, például befűzés esetén:

- Egy belső elemnek bal és jobb szomszédja is van, a befűzés négy pointert érint.
- Az első elemnek nincs bal szomszédja, az utolsónak nincs jobb szomszédja.
- Üres listába történő befűzés is külön eset: a befűzött elemnek sem bal, sem jobb szomszédja sincs.

Ezt a lista típust majd a láncolt hasító tábláknál használjuk.

### 3. gyakorlat

## Ciklikus kétirányú listák (C2L)

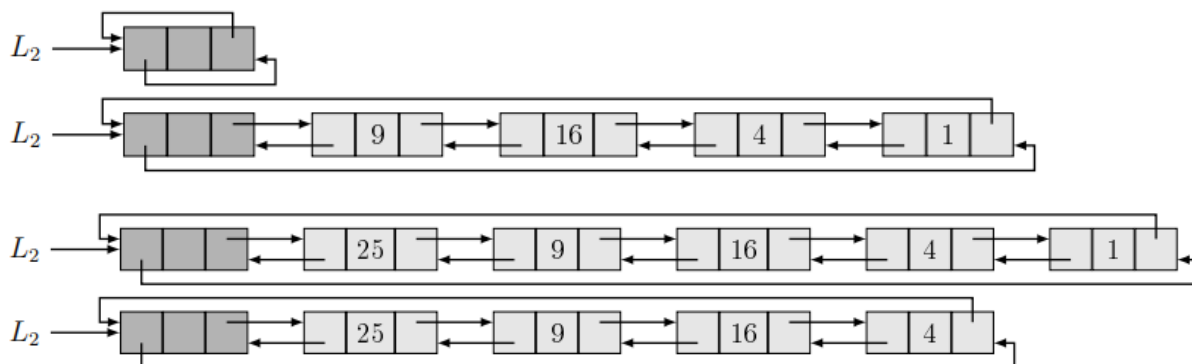
Lehet fejelemes, vagy fejelem nélküli. Legegyszerűbbek a műveletek a fejelemes C2L -ek esetén, így az alábbi feladatokban ezekkel fogunk foglalkozni.

### Elemtípus, alpműveletek

E2
+ <i>prev, next</i> : E2* // refer to the previous and next neighbour or be <b>this</b>
+ <i>key</i> : $\mathcal{T}$
+ E2() { <i>prev</i> := <i>next</i> := <b>this</b> }

Minden adattagja publikus, az üres konstruktor a pointereket „önmagára” állítja!

C2L példák a jegyzetből:



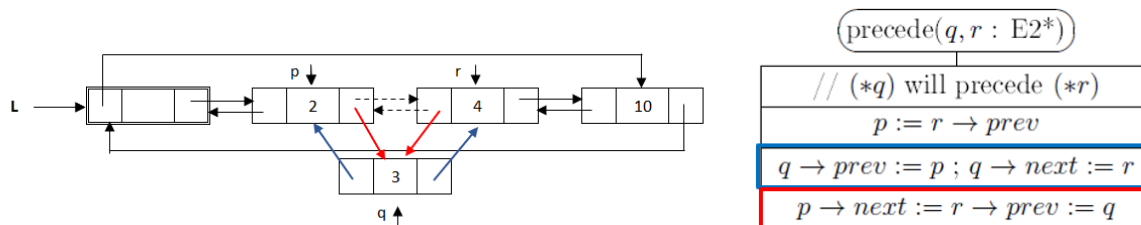
6. ábra. Az  $L_2$  mutató fejelemes kétirányú ciklikus listákat (C2L) azonosít egy képzeletbeli program futásának különböző szakaszaiban. Az első sorban az  $L_2$  lista üres állapotában látható.

Mivel a listaelem konstruktora a pointereket úgy állítja be, hogy azok magára az elemre mutassanak, ezt kihasználva egy új fejelemes C2L lista fejelemének létrehozása:  $L := \text{new E2}$  utasítással történhet!

### Alap műveletek

Beszúrások:

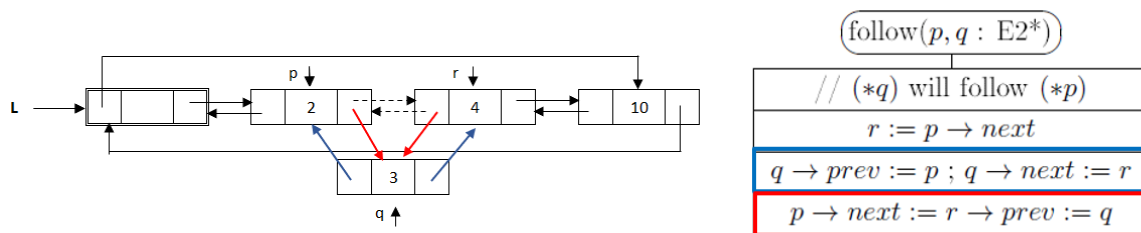
**precede( $q, r: E2^*$ )**





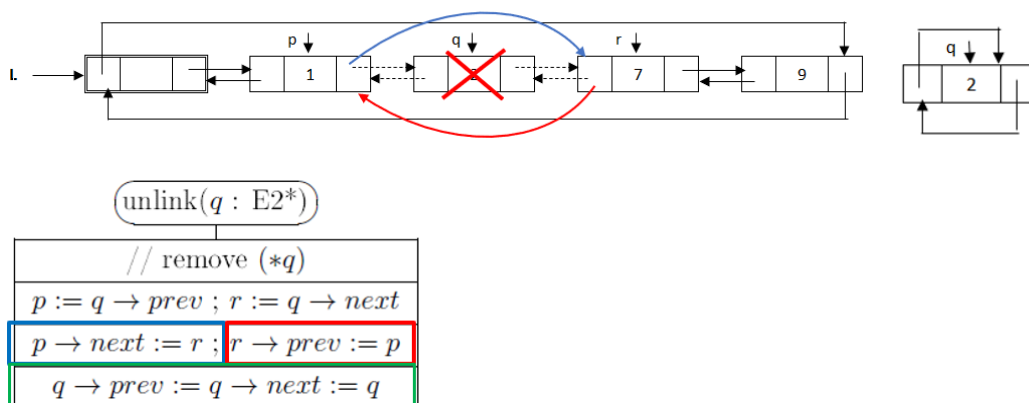
### 3. gyakorlat

**follow(p,q:E2\*)**



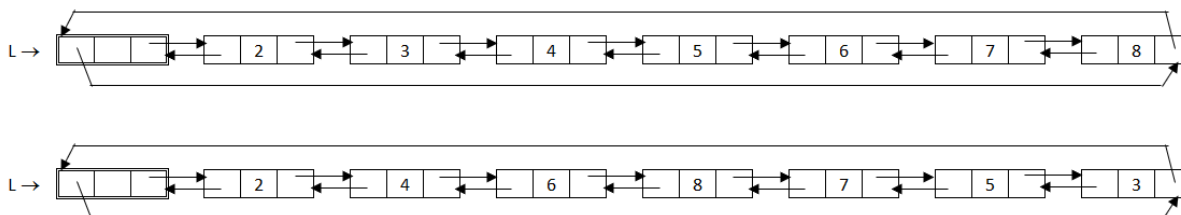
Kifűzés:

**unlink(q:E2\*)**



### Gyakorló feladat 1

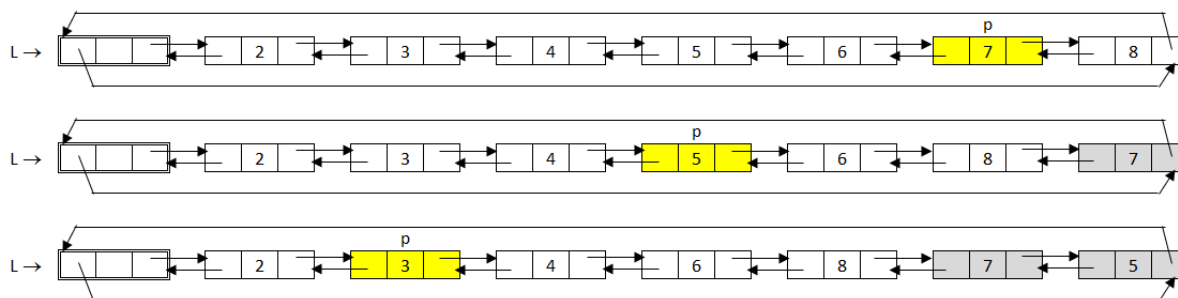
- Adott egy természetes számokat tartalmazó C2L lista. A lista szigorúan monoton növekvően rendezett.
- A lista egyszeri bejárásával rendezzük át az elemeit úgy, hogy a lista elején legyenek a páros számok növekvően, a végén pedig a páratlanok csökkenően.



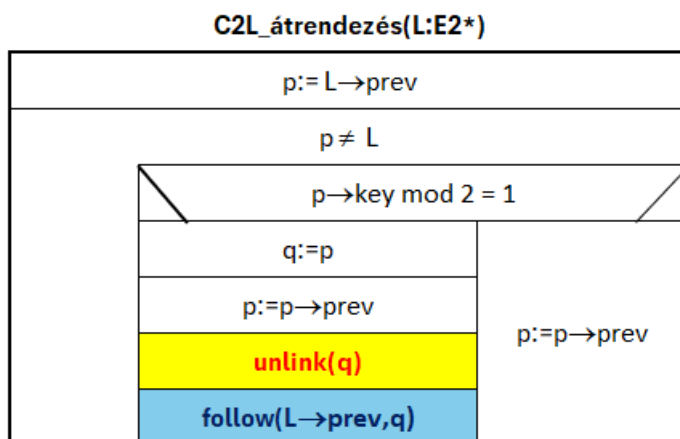
Megoldási terv:

- Fordított irányban járjuk be a listát. A bejáró pointer p lesz.
- Ha a p elem kulcsa páratlan, az elemet kifűzzük, és átláncoljuk a lista végére (a fejelem elé).
- A kiláncoláshoz egy q segéd pointert fogunk használni, p-vel pedig a kiláncolás előtt tovább lépünk.

### 3. gyakorlat



Megoldás:

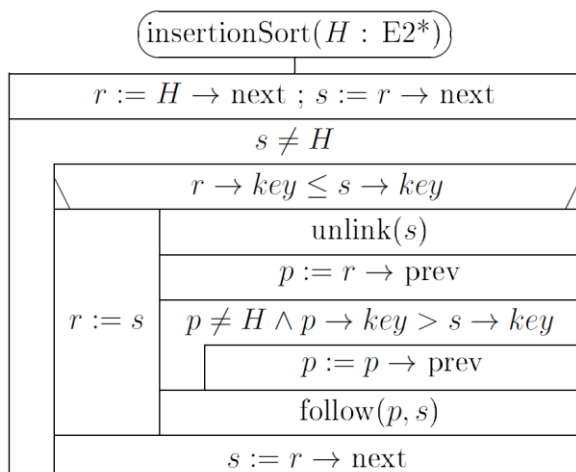


További kérdések:

- unlink és follow nélkül hogyan lehetne megoldani?
- A „follow” helyett hogyan lehetne a „precede”-t használni? precede(q,L)

### Beszűrő rendezés:

Jegyzetben megtalálható a beszűrő rendezés C2L-re (itt is visszafelé halad, a beszűrandó elem helyének megkeresése):



### 3. gyakorlat

#### Rendezett listák összefésülése

Igen gyakoriak azok a feladatok, melyek két rendezett sorozat összefésülésén alapulnak (például egy raktárban tárolt készlet napra-készítése a beszállítás / kiszállítás alapján). Ilyenkor a rendezettség nagyban növeli az algoritmus hatékonyságát. Ha a sorozatok listába vannak fűzve, akkor a pointerek állításával gyorsan és hatékonyan elvégezhető a két lista összefésülése.

Az összefésüléses feladatok általában fejelemes listákkal kapcsolatosak, a lista típusa H1L vagy C2L, de például az előadáson láthattuk a merge sort listás változatát, mely egy S1L listát rendez összefésüléssel. (7.1.6. Az összefésülő rendezés S1L-ekre)

#### Gyakorló feladat 3

L1 és L2 egy-egy szigorúan monoton növekvően rendezett C2L lista fejelemére mutat. Mivel a kulcsok egyediek, a listát halmaznak tekinthetjük. Állítsuk elő L1-ben a két halmaz unióját, úgy, hogy a szükséges elemeket L2-ből átfűzzük L1-be, a többit felszabadítjuk. Így az L2 lista az algoritmus végére kiürül.

##### Megoldási terv:

Egy-egy bejáró pointerrel lépegetünk a listákon. p halad az L1 listában, q az L2 listában.

A pointerek által kijelölt elemek kulcsát hasonlítjuk össze, három eset lehetséges:

- (1)  $p \rightarrow \text{key} < q \rightarrow \text{key}$
- (2)  $p \rightarrow \text{key} = q \rightarrow \text{key}$
- (3)  $p \rightarrow \text{key} > q \rightarrow \text{key}$

A feldolgozó ciklust leállítjuk, ha valamelyik listán körbe érünk, ezért a ciklus feltétele, hogy  $p \neq L1 \wedge q \neq L2$

Ha valamelyik listán körbe érünk:

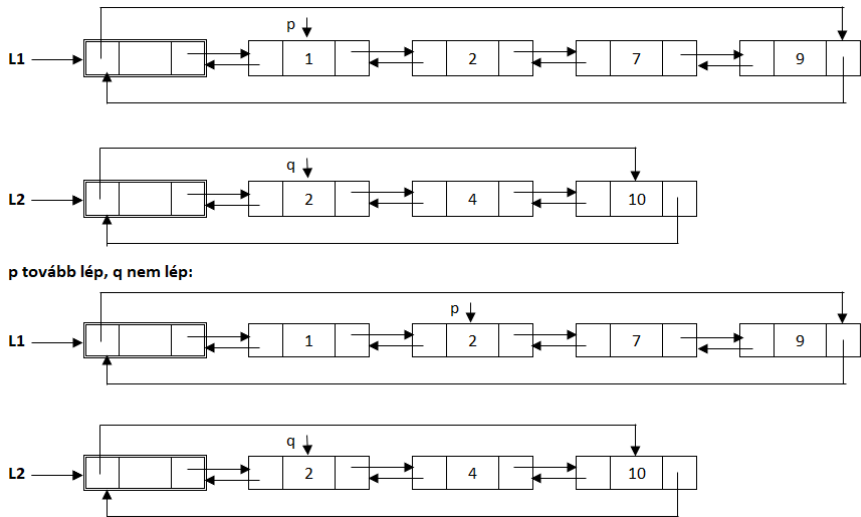
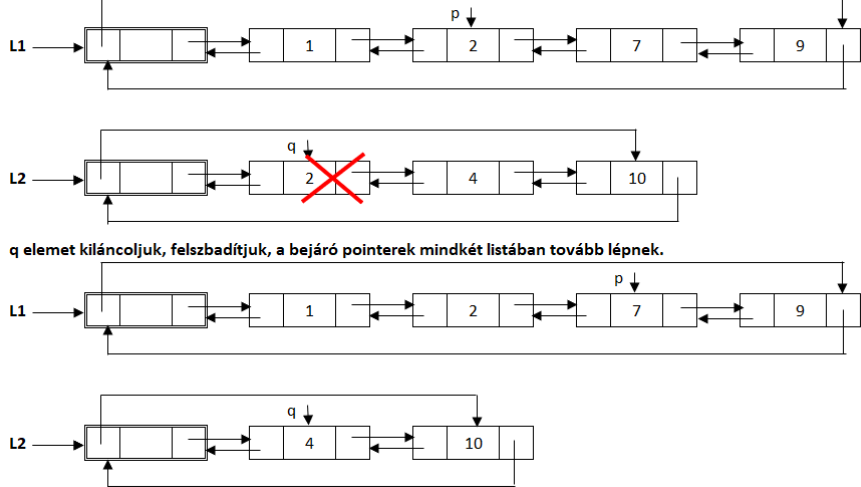
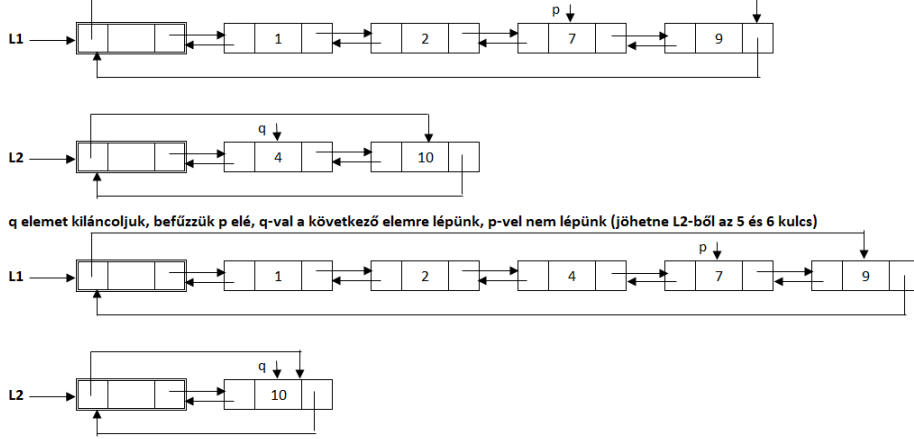
- Ha  $p=L1$  akkor L1, ha  $q=L2$ , akkor L2 listán körbe értünk. (Mivel egyenlő kulcsok esetén mindkét bejáróval lépünk, az is előfordulhat, hogy egyszerre érünk a listák végére.)
- Ha L2 listán értünk körbe, akkor kész vagyunk, ha viszont L2 listában még vannak elemek, azokat be kell fűzni L1 végére. Ezt megtehetjük egyesével, de hatékonyabb, ha a maradék láncrész egyszerre, konstans lépésben fűzzük át.

Megjegyzés:

Észrevehető, hogy valójában q bejáró pointerre nem is lenne szükség, mert az algoritmus során, a ciklus indulásakor  $q=L2 \rightarrow \text{next}$  mindig teljesül. De a rövidebb írásmód, és érthetőbb megfogalmazás miatt q-t használunk L2 aktuális elemének címzéséhez.

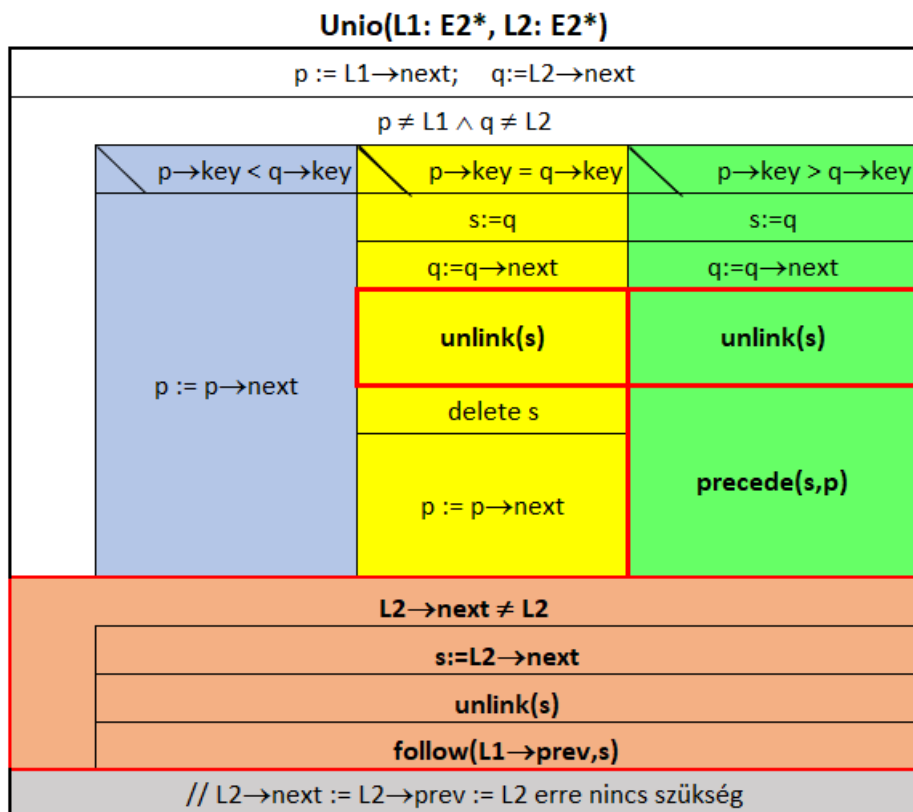
Az esetek szemléltetése:

### 3. gyakorlat

<p>(1) L1 lista aktuális elemében lévő kulcs kisebb, mint az L2 lista aktuális elemének kulcsa, ekkor L1 lista bejáró pointere tovább lép, L2 lista bejárója nem lép.</p>	 <p>p tovább lép, q nem lép:</p>
<p>(2) L1 lista aktuális elemében lévő kulcs egyenlő az L2 lista aktuális elemében lévő kulccsal. Ekkor az L2 listában lévő elemet ki kell láncolni, fel kell szabadítani, a bejáró pointerok mindkét listában tovább lépnek.</p>	 <p>q elemet kiláncoljuk, felszabadítjuk, a bejáró pointerok mindkét listában tovább lépnek.</p>
<p>(3) L1 lista aktuális elemében lévő kulcs nagyobb, mint az L2 lista aktuális elemében lévő kulcs. Ekkor az L2 listában lévő elemet ki kell láncolni, át kell fűzni L1 lista aktuális eleme elé, L2 listában tovább lépünk, L1 listában nem.</p>	 <p>q elemet kiláncoljuk, befűzzük p elé, q-val a következő elemre lépünk, p-vel nem lépünk (jöhetne L2-ből az 5 és 6 kulcs)</p>

### 3. gyakorlat

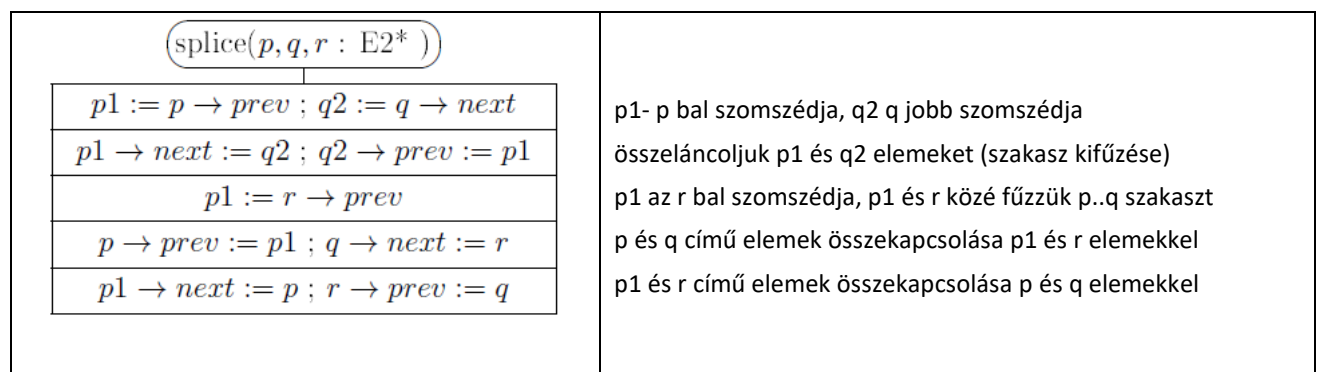
Megoldás:



Hatékonyabbá tehetjük a megoldásunkat, ha amikor főciklusból kilépve, az L2 listában még vannak elemek, azokat nem ciklussal fűzzük át L1 listába, hanem az L2-beli lánc-darabot a két végénél fogva konstans lépésben fűzzük L1 lista végére. Ez már nem oldható meg az alapl műveletekkel, de a jegyzetben definiálva van egy splice nevű elemi algoritmus, mely egy listadarabot tud átfűzni egy adott elem elé.

Megoldás: append – splice algoritmusok a jegyzetből (7.15 példa):

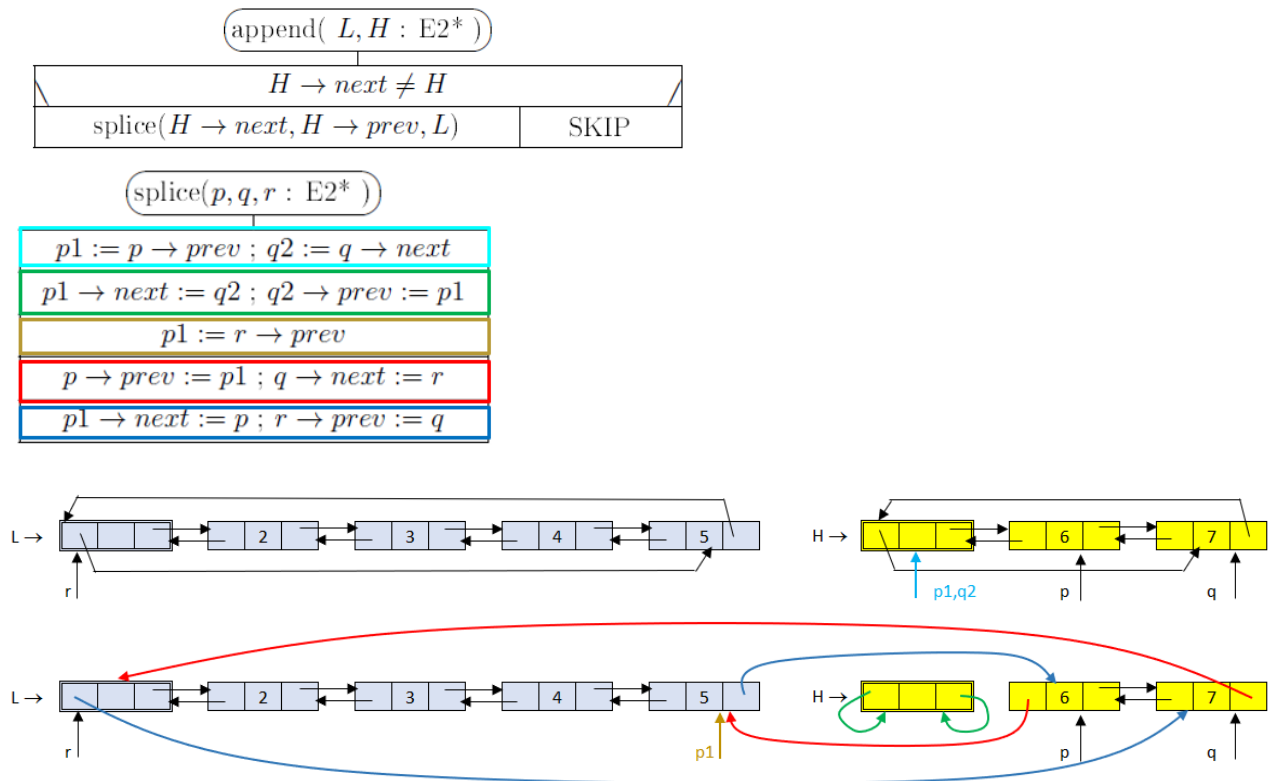
A splice(p, q, r : E2\*) egy olyan elemi listaművelet, amely egy C2L egy adott [p, . . . , q] szakaszát eltávolítja, majd az eltávolított szakaszt egy C2L adott \*r eleme elé befűzi. (Előfeltétel, hogy p..q szakasz ne tartalmazza a fejeleket, p után jöjjön q (p=q lehet), valamint \*r ne legyen a p..q szakasz egy eleme, de r lehet egy másik C2L listának az eleme.)



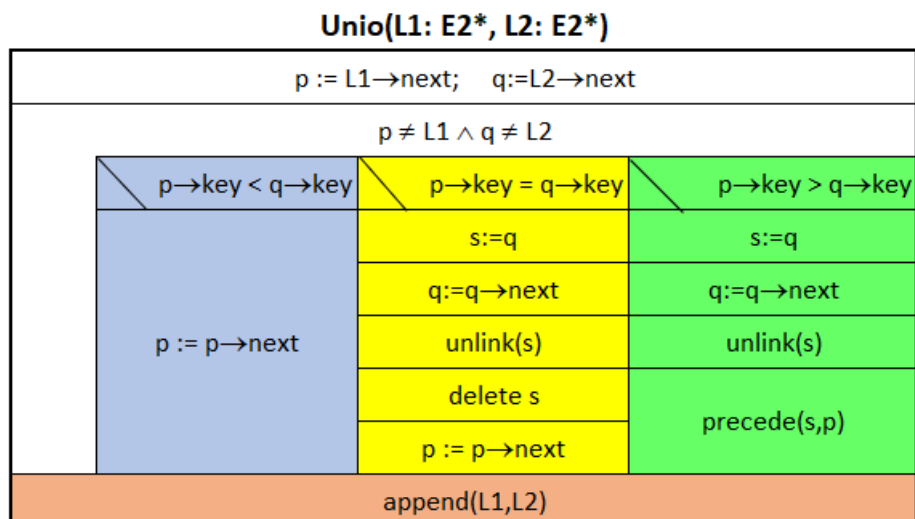
### 3. gyakorlat

Ennek segítségével definálhatjuk az append műveletet, amely egy L C2L lista végére fűzi egy H nem üres C2L lista összes elemét.

Szemléltetés: L végére fűzzük H elemeit:



Az append használatával a megoldás:



Műveletigény

- Vizsgáljuk meg az összefésülő algoritmusunk műveletigényét! Legyen L1 lista hossza: n, L2 lista hossza m.
- Mit mondhatunk  $mT(n, m)$  illetve  $MT(n, m)$  műveletigényről?

### 3. gyakorlat

- Az algoritmus az egyik listán mindenképp végig iterál, tehát:  
 $mT(n,m) = \Theta(n)$  abban az esetben, ha  $L_1$  minden eleme kisebb, mint  $L_2$  első eleme. Miért?  
 Indokoljuk az állítást!  
 $mT(n,m) = \Theta(m)$  abban az esetben, ha  $L_2$  minden eleme kisebb, mint  $L_1$  első eleme. Miért?  
 Indokoljuk az állítást!
- $MT(n,m) = \Theta(n+m)$  Például pontosan  $n+m$  iterációt hajt végre abban az esetben, ha nincsenek azonos elemek, és  $L_1$  utolsó előtti eleme kisebb, a legutolsó pedig nagyobb, mint  $L_2$  legutolsó eleme. Indokoljuk meg, miért?

#### Gyakorló feladat 4

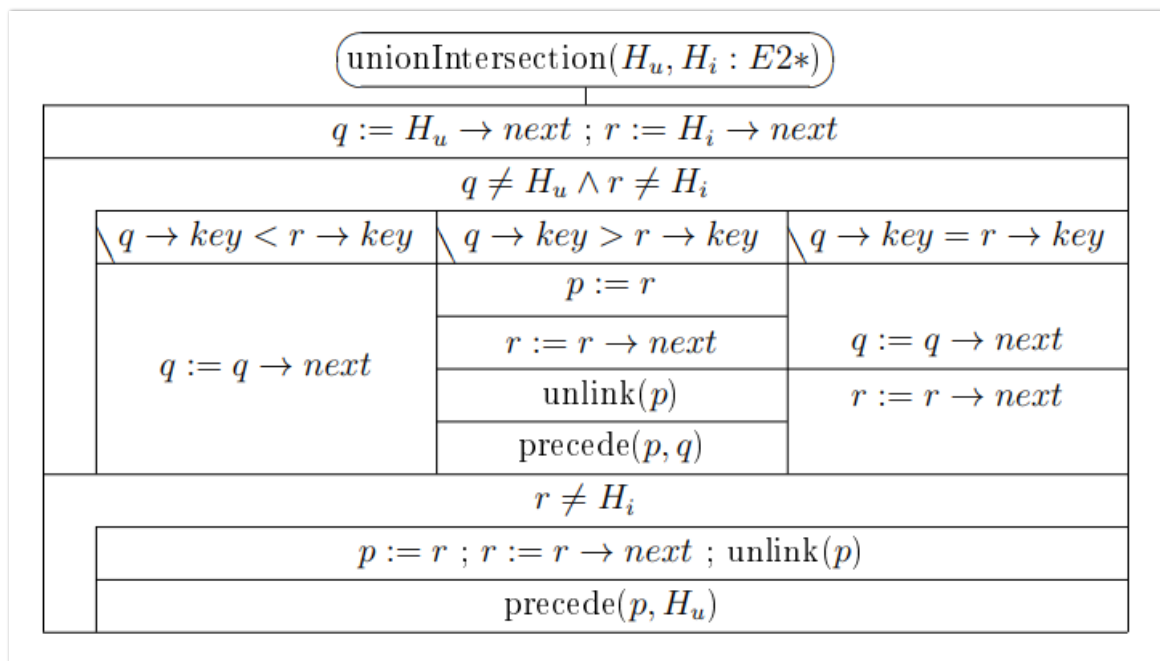
H1L listák összefésülése.

A jegyzetben található a következő algoritmus (7.22 példa):

- Legyenek  $H_u$  és  $H_i$  szigorúan monoton növekvően rendezett C2L-ek!
- Írjuk meg a  $\text{unionIntersection}(H_u, H_i : E2^*)$  eljárást, ami a  $H_u$  listába  $H_i$  megfelelő elemeit átfűzve, a  $H_u$  listában az eredeti listák, mint halmazok unióját állítja elő, míg a  $H_i$  listában a metszetük marad!
- Ne allokáljunk és ne is deallokáljunk listaelemeket, csak az listaelemek átfűzésével oldjuk meg a feladatot!  $MT(n_u, n_i) \in \Theta(n_u + n_i)$ , ahol a  $H_u$  C2L hossza  $n_u$ , a  $H_i$  C2L hossza pedig  $n_i$ . Mindkét lista maradjon szigorúan monoton növekvően rendezett C2L!

Készítsük el az algoritmust ugyanezen feltételekkel H1L listára!

A jegyzet algoritmus:



Ezt írjuk át H1L listákra:

### 3. gyakorlat

unionIntersection( Hu, Hi : E1*)		
qe:=Hu; q:=Hu->next; re:=Hi; r:=Hi->next		
q ≠ 0 ∧ r ≠ 0		
q->key < r-> key	q->key > r-> key	q->key = r-> key
qe:=q q:=q->next	re->next:=r->next	qe:= q q:=q->next  re:=r r:=r->next
	qe->next:=r	
	r->next:=q	
	qe:=r	
	r:=re->next	
r ≠ 0		
qe->next := r		skip
re-> next:= 0		