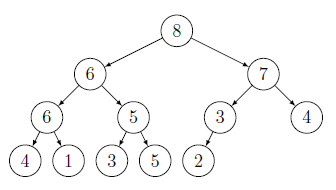
Algo1 2021-es ZH2-höz készült segédlet. Simán lehetnek benne hibák, felelősséget nem vállalok ezekre. Nyílt címzéses dolgok már csak bele vannak hányva, de azért remélem tud segíteni valamennyit.

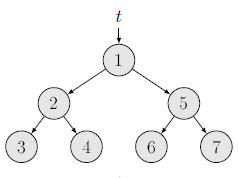
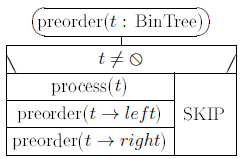
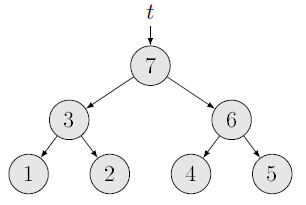
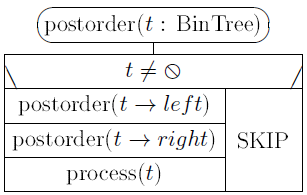
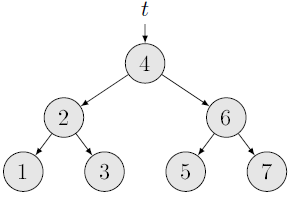
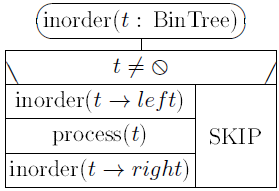
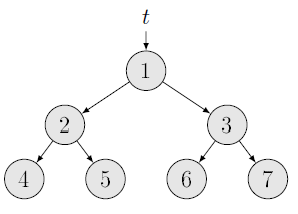
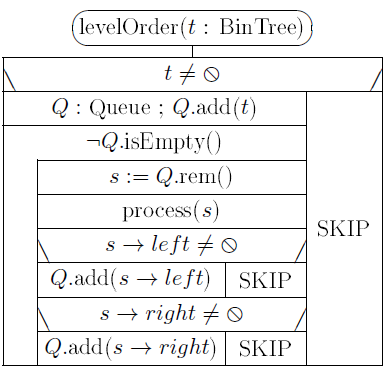
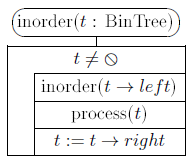
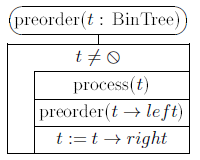
Készítette: Tasi Zoltán

# Bináris fák

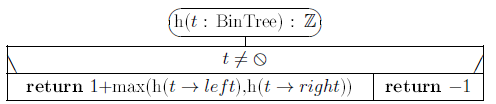
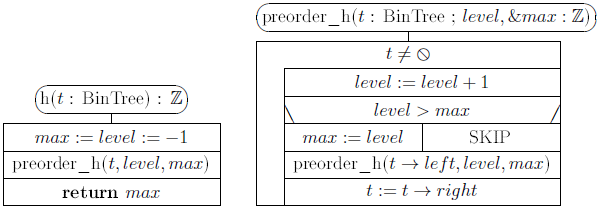
## Alapfogalmak

* **Csúcs (node**): a fa egy eleme
  + **Levél**: csúcs aminek nincs gyereke
  + **Belső csúcs** (internal node): aminek van gyereke (tehát nem levél)
  + **Gyökér** (root): aminek nincs szülője
  + Testvérek: egy szülő (közvetlen) gyerekei
  + Leszármazott: gyerekek és azok gyerekei
  + Ősök: szülő és annak ősei
* **Csúcs részfája**: egy csúcs jobb vagy bal gyereke általi fa, vagy akár önmaga
  + Valódi részfa: Részfa, ami nem üres és nem önmaga
* **Egy fa szintjei**: Az i-edik szintű csúcsok összessége
  + Gyökér van a nulladik szinten
* **Fa magassága (h)** = legmélyebben fekvő levelek szintszáma
  + Üres fa magassága: -1
* **Fa mérete (n)** = fa csúcsainak száma
* Üres fa: még gyökércsúcs sincsen
* **Listává torzult fa**: Ahol minden csúcsnak egy gyereke van
* **Szigorúan bináris fa**: ahol minden belső csúcsnak két gyereke van
  + **Teljes bináris fa**: ahol ráadásul minden levél azonos szinten van
    - Teljes, h magasságú bin. fáknak csúcsainak (nem leveleinek) száma):
      * 1+2+4+…+2h = 2h-1
  + **Majdnem teljes bináris fa**: Ha egy teljes fa levélszintjéről nulla, egy vagy több levelet elveszünk (de nem az összeset)
    - Majdnem teljes fa ≤ Teljes fa
    - Üres fa is majdnem teljes fa
    - Csúcsainak száma: h = AlsóRész[ log(n) ]
    - **Balra tömörített bináris fa**
      * Más néven **szintfolytonos bináris fa**
      * Alsó szintjén egyetlen levéltől balra sem lehet új levelet beszúrni
      * Tehát csak az alsó szint jobb széléről hiányozhatnak csúcsok
      * 

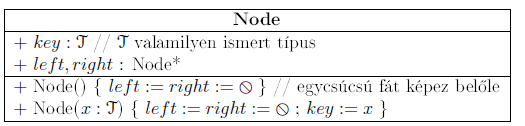
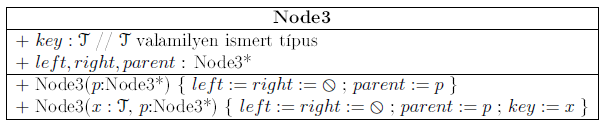
## Bin fa bejárások

* Mindegyik bejárás műveletigénye: T ∈ θ (n), ahol n a fa mérete
* Üres fára: üres program
* Preorder: Root → Left → Right
  + 
* Postorder: Left → Right →Root
  + 
* Inorder: Left → Root → Right
  + Általában legtöbb információt tartalmazza
  + 
* Szintenkénti:
  + Más néven: Szintfolytonos, Breadth First, Level Order
  + Gyökértől kezdve szintenként, balról jobbra
  + 
  + A sorban hivatkozásokat tárolunk: következő szinthez tartozó csúcsokat (gyerekeket)
* Végrekurziók ciklussá alakítása: konstans szorzóval gyorsíthat
  + 

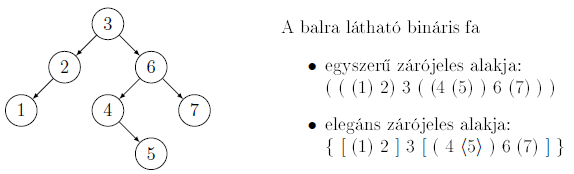
## Fa magassága

* Egyszerűen: (post order gyakorlatilag)
  + 
* Preorder bejárással
  + 

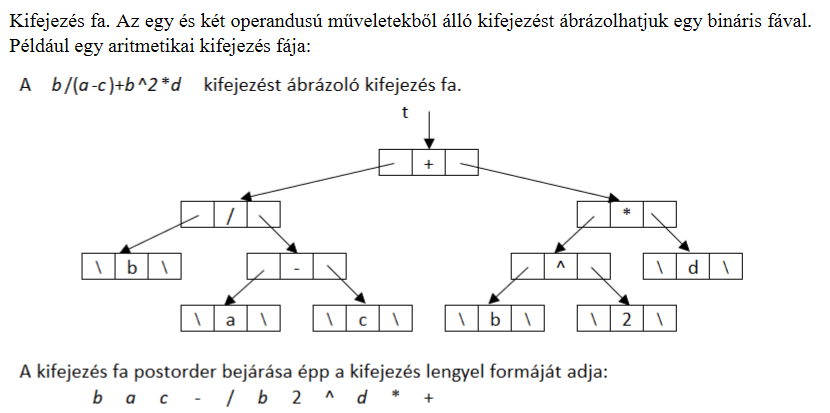
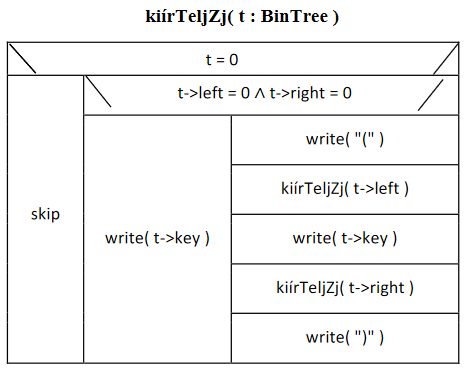
## Bináris fa láncolt ábrázolása

* Üres fa: null pointer, ∅
* Eddigi „BinTree” absztrakt osztály megvalósítása: Node\*
  + Node (egyszerű):
    - 
  + Node3 (szülő elemre pointerrel):
    - 
    - Itt ha: parent == null => ez a gyökércsúcs

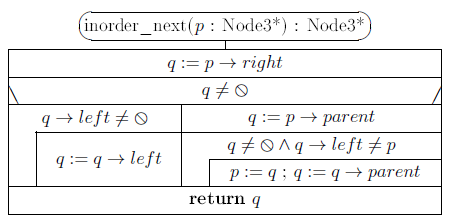
## Bináris fa zárójelezett, szöveges formája:

* ( balRészFa Gyökér jobbRészFa )
* Üres fa: „”
* 

## Kifejezés fa

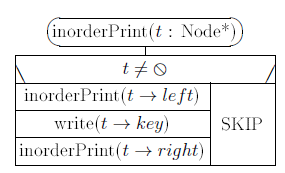
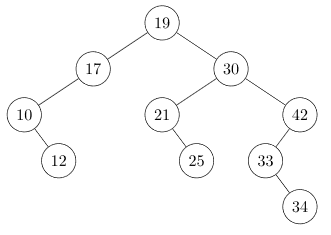
* (7. gyak vége)
* 
  + (itt a \ jel = ∅)
* Rekurzív algoritmus, ami kiírja egy adott kifejezés fa teljesen zárójelezett alakját:
  + 
  + Kimenet a fenti fa esetén: ( ( b / (a-c) + ( (b^2) \* d ) )

Mellékes: Node3\*-val inorder bejárás szerinti következő csúcs megadása (ea 9.3):

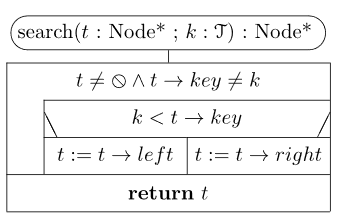
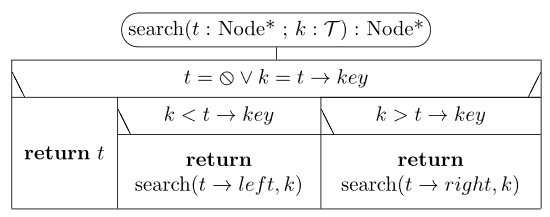
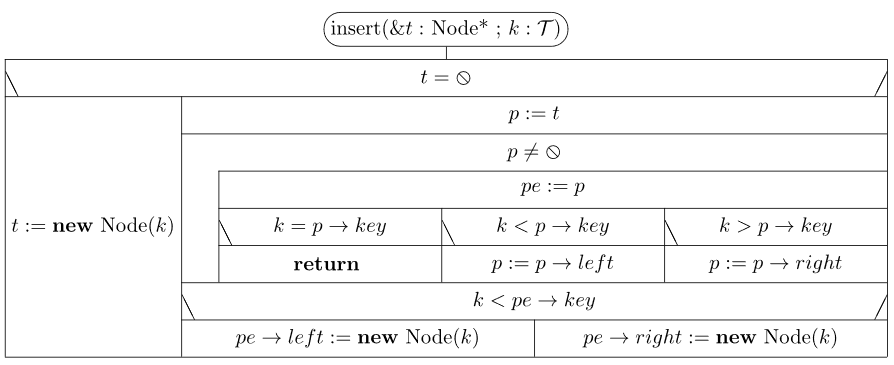
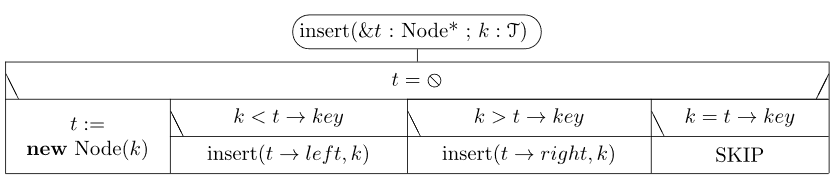
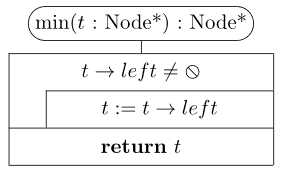
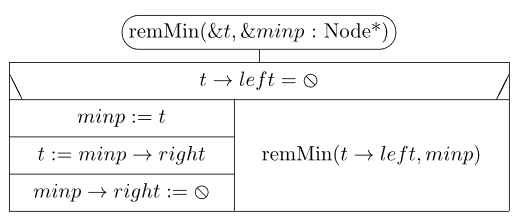
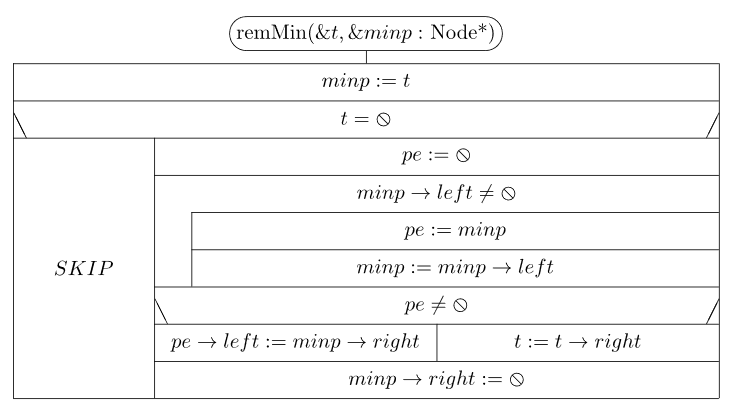
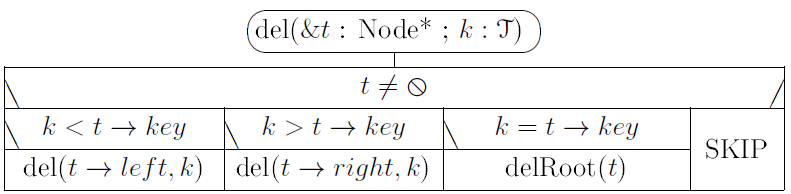
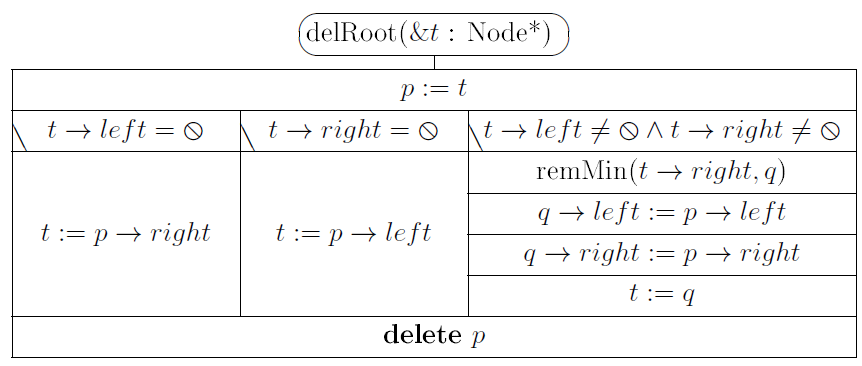
* 

# Bináris keresőfák:

## Alap fogalmak

* Keresőfa:
  + Tetszőleges csúcs kulcsánál a bal részfájában minden csúcs kulcsa kisebb, a jobb részfájában minden csúcs kulcsa nagyobb.
  + Ez nem engedi meg az egyenlőséget, tehát csak különböző elemek lehetnek!
    - (minden kulcs egyedi)
    - Bal részfa bármelyik eleme < gyökér < jobb részfa bármelyik eleme
* Rendezőfa:
  + Mint a keresőfa, de itt az egyenlőség is megengedett
    - (lehetnek benne duplikált elemek)
    - Bal részfa bármelyik eleme ≤ gyökér ≤ jobb részfa bármelyik eleme
* Bináris keresőfa inorder bejárása
  + Gyakorlatilag a fa kulcsainak mon. növ. sorrendben kiírása
  + 
* Sorozatból bináris keresőfa készítése:
  + 9,17,30,10,21,12,42,25,33,34
  + 
* Keresőfa átlagos magassága: O (lg n)

## Bináris keresőfa alapműveletei:

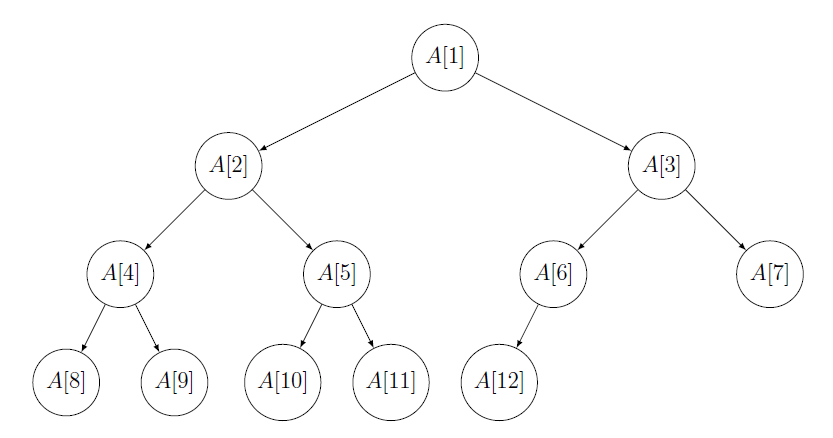
* Mindegyik műveletigénye: MT(h) ∈ Θ (h), ahol h = fa magassága
  + Keresőfa átlagos magassága: O (lg n)
  + Ezért beszúrás/törlés sokkal hatékonyabb, mint tömbökkel/listákkal
  + De ha listává torzul a keresőfa => hatékonyág láncolt listához hasonlít
* **t** = fa, **k** = csúcs amivel dolgozunk
* **search(t, k)**: A t fában megkeresi a k kulcsú elemet, NULL-t ad vissza, ha nincs benne.
  + Iteratívan vs rekurzív verzió
  +  
* **insert(t, k)**: Beszúrja a k kulcsú elemet, ha még nincs a fában.
  + Ez is először gyakorlatilag megkeresi a beszúrandó elemet
  + Iteratívan:
  + 
  + Rekurzívan:
  + 
* **min(t)**: A minimális kulcsú csúcs címét adja vissza.
  + Pontosabban az inorder bejárás szerinti első elemet (a bal alsó csúcs)
  + Iteratívan:
    - 
* **remMin(t, minp)**: A fából kifűzi a minimális kulcsú (bal alsó) csúcsot, és a címét a minp pointerben adja vissza.
  + Kifűzés = min csúcs helyére teszi a jobb oldali részfáját
  + Rekurzív verzió:
  + 
  + Iteratív verzió:
  + 
* **del(t, k)**: Törli a k csúcs elemet, amennyiben az megtalálható a fában.
  + Először megkeresi a k kulcs hellyét (ha nincs akkor kilép) (del)
  + Utána két eset (delRoot):
    - Ha a csúcs egyik részfája üres → törlendő csúcs helyére teszi a másik részfát
    - Ha a csúcsnak két gyereke van → remMin(k, minp)-el kiveszi a jobb részfából a minimális elemet, és a törlendő csúcs (k) helyére teszi
  + 
  + 

# Kupacok

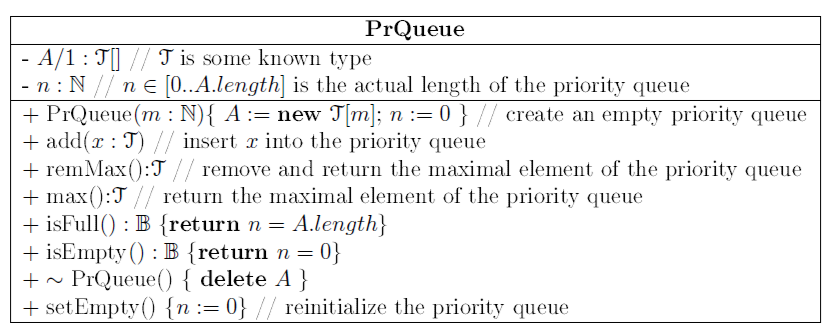
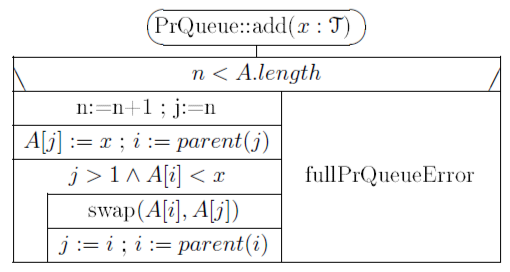
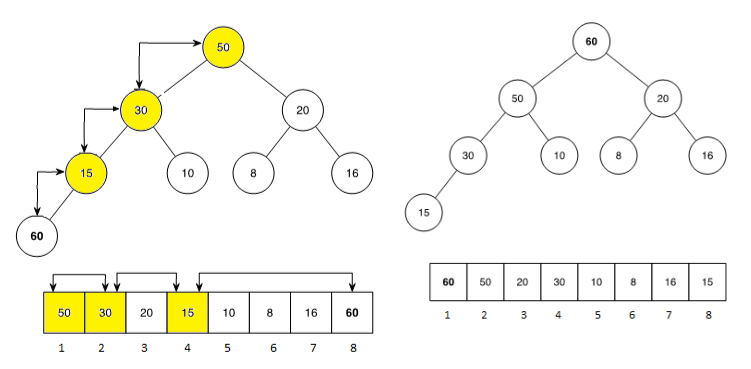
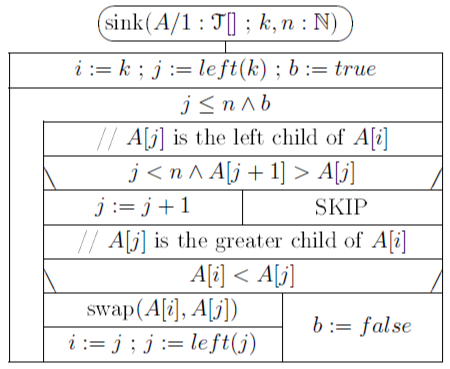
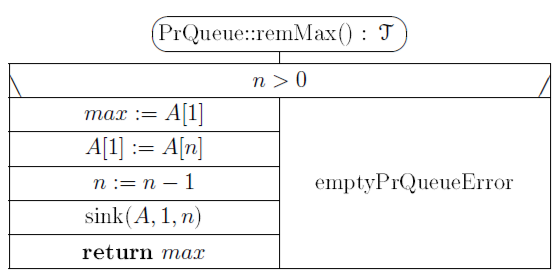
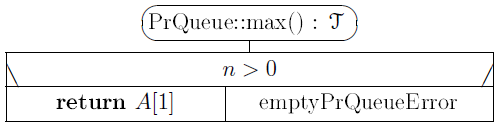
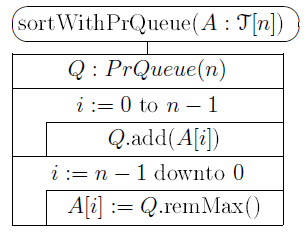
## Fogalmak

* **Maximum-kupac (heap)**: szintfolytonos bináris fa
  + Minden belső csúcs kulcsa nagyobb-egyenlő, mint a gyerekeié
    - De kupac bal- és jobboldali részfájai között nincs kapcsolat
  + Ezt fogjuk érteni az egyszerű *kupac* alatt
  + Bármely nemüres kupac maximum eleme a gyökércsúcs
    - Minimum a levelek között
  + Kupac részfái is kupacok
* **Minimum-kupac**:
  + Aminek minden belső csúcs kulcsa kisebb-egyenlő, mint a gyerekeié
* **Csonka kupac:**
  + Minden szülő-gyerek párosban a szülő kulcsa nagyobb-egyenlő mint a gyerek
  + Kivéve! ha a szülő a gyökércsúcs
  + (Mintha a gyökér meg nem lenne a helyén, de minden más igen – átmeneti adatszerkezet)

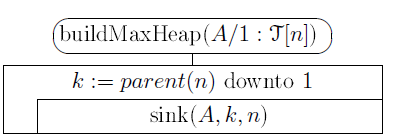
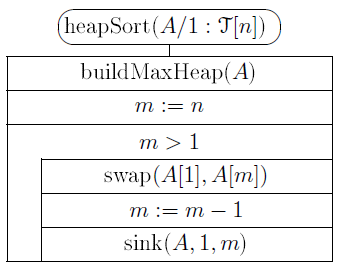
## Szintfolytonos bináris fák aritmetikai ábrázolása

* Tömbben ábrázolás: A : T[m]
  + m = tömb max mérete
  + n = fa csúcsainak száma
    - Ha n == 0 → üres fa
  + 1-től indexelt tömb
  + 
* Fa gyökércsúcsa: A[1]
* A[i] gyerekei: A[left(i)] és A[right(i)]
  + Ha left(i) < n → A[i] csúcsnak két gyereke van
  + Ha left(i) = n → A[i] csúcsnak egy (bal oldali) gyereke van
  + Ha left(i) > n → A[i] csúcs egy levélcsúcs
  + right(i) = left(i) + 1
  + left(i) = 2i
  + Összesítve:
    - **A[i] bal gyereke = A[2i]**
    - **A[i] jobb gyereke = A[2i+1]**
    - **A[i] szülője = A[1/2]**
      * Alsó kerekítés
      * Csak akkor, ha i > 1!

## Elsőbbségi sor kupaccal

* Elsőbbségi sor: Az elsőbbségi sor egy zsák (multihalmaz), amelybe be tudunk tenni újabb elemeket, és ki tudjuk választani, illetve kivenni az *egyik maximális elemét.*
* Az elsőbbségi sor aktuális elemeit az A[1..n] résztömb tartalmazza, ami egy kupac.
  + ennek max mérete: m
* 
* Műveletigények:
  + MTadd(n), MTremMax (n) ∈ Θ (lg n)
    - mivel az *add* és *sink* fő ciklusa max *fa magassága*szor fut le
  + mTadd(n), mTremMax(n), Tmax(n) ∈ Θ (1)
* **add(x)**
  + Ha van üres hely, beszúrom a legvégére (utolsó levél után), majd emelést hajtunk végre a kupacban
    - Emelés: addig cserélem a szülőivel, amíg nem lesz kisebb/egyenlő vele
  + 
  + 60-as elem beszúrása:
  + 
* **sink(A, k, n)**
  + k: süllyesztendő elem indexe, n: csúcsok száma
  + i: szülő indexe, j: (nagyobb) gyerek indexe
  + Ez állít helyre egy csonka kupacot
  + Süllyesztendő gyereket mindig az aktuálisan nagyobb gyerekével cseréli meg
    - Amíg levélszint fölött van (j < n)
    - Amíg a nagyobbik gyereke nagyobb nála (b)
  + 
* **remmax()**
  + visszatér a max elemmel, amit töröl, majd beteszi helyére au utolsó levelet, és addig süllyeszti amíg helyre nem áll a kupac megint
  + 
* **max()**
  + 
* **Tömb rendezése elsőbbségi sorral**
  + 
  + Felépít egy elsőbb. sort a tömbből
    - Majd a rendezendő tömböt újra feltöltöm a sor elemeivel, végétől kezdve mindig belerakom a max elemet
  + Műveletigénye: O (n \* log n)
    - De memóriaigény: rendezendő tömbbel azonos méretű segédtömb

## Kupacrendezés (heap sort)

* Gyakorlatilag sortWithPrQueue optimalizálása: helyben rendezővé
  + Θ (1) memóriát használ
  + mT ∈ Θ (n)
  + MT ∈ Θ (n)
* Nem stabil
* **buildMaxHeap()**
  + Tömb kupaccá alakítása, lineáris műveletigénnyel
  + A lényege:
    - Utolsó levél szülőjétől kezdve visszafelé haladunk (balra szintfolytonosan)
    - Ezeket a belső csúcsokat csonka kupacoknak tekintjük, először csak egy magasságúak
      * Mivel csak a gyökerük van rossz helyen, azt süllyesztjük
    - Végül az egész fa gyökércsúcsát süllyesztjük, és így helyben kupaccá alakítottuk a tömbünket
  + 
* **heapSort()**
  + 
    - Első lépés: tömb kupaccá alakítása
    - Utána ciklikusan:
      * Eddigi maximum a tömb legvégére kerül
        + Gyakorlatilag kivesszük a kupacból, helyére az utolsó elem
      * Kialakult csonka kupac helyreállítása süllyesztéssel

# Radix rendezés: lineáris időben

## Összehasolnító rendezések tételek

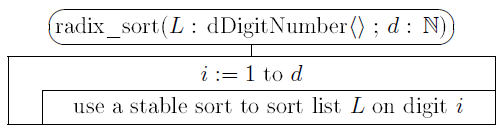
**Összehasonlító rendezés**: ahol az elemek rendezéséhez csak az elemek összehasonlításával nyerünk információt

Tétel: nem lehet lineáris időnél gyorsabban rendezni (∀ rendezésre: mT(n) ∈ Ω (n)

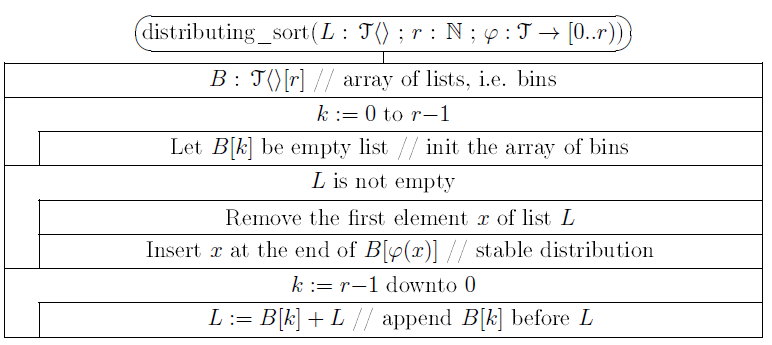
Tétel: Tetszőleges összehasonlító rendezésre MT(n) ∈ Ω (n \* lg n).

Tétel: Bármely összehasonító rendezés végrehajtásához a legrosszabb esetben MC(n) 2 (n lg n) kulcsösszehasonlítás szükséges.

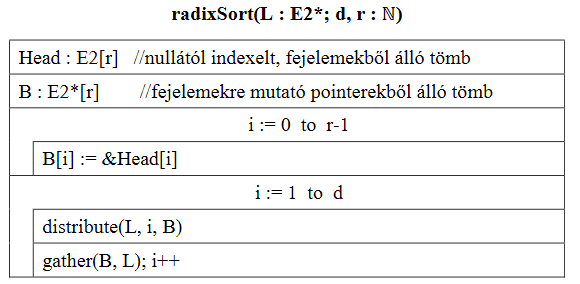
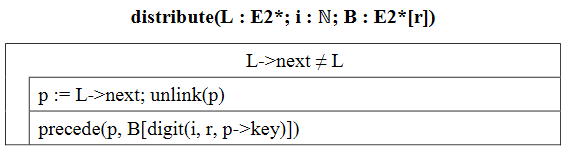
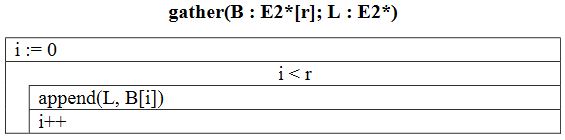
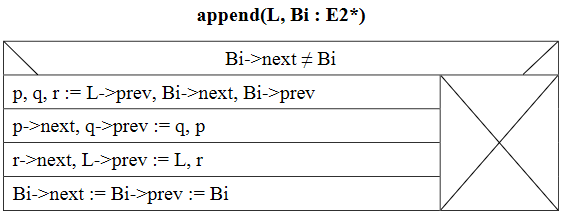
## Radix rendezés

* Kulcsokra kikötések:
  + **r** alapú számrendszerben
  + **d** számjegyű
  + Előjel nélküli, egész számok
  + dDigitNumber-el jelöljük a típust
  + Számok számjegyeit jobbról balra sorszámozzuk
* Stabil rendezés
* Nem összehasonlító rendezés, kulcsok számjegyeit nézi csak!
* Műveletigényük: T ∈ Θ (n) (tehát lineáris)
* radix\_sort()
  + L: dDigitNumber típusú számok sorozata
  + 
    - Első menet után: legkevésbé szignifikáns számjegyek alapján rendezett
    - Második menet után: jobbról második számjegy alapján rendezett
  + A belső segédfüggvény több fajta is lehet, de stabilnak kell lennie

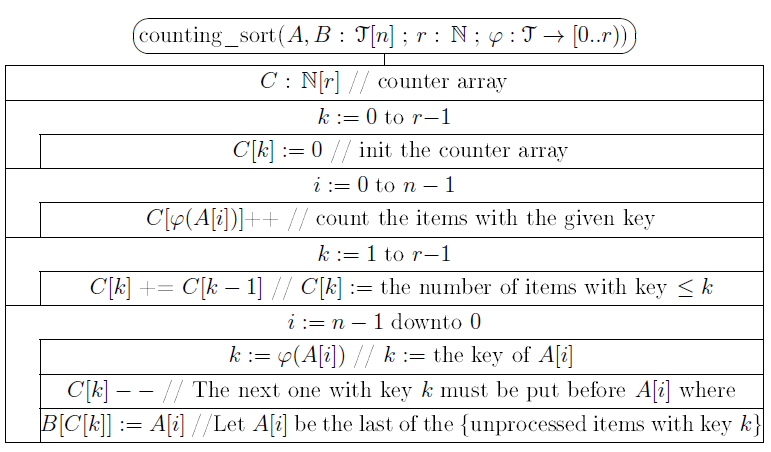
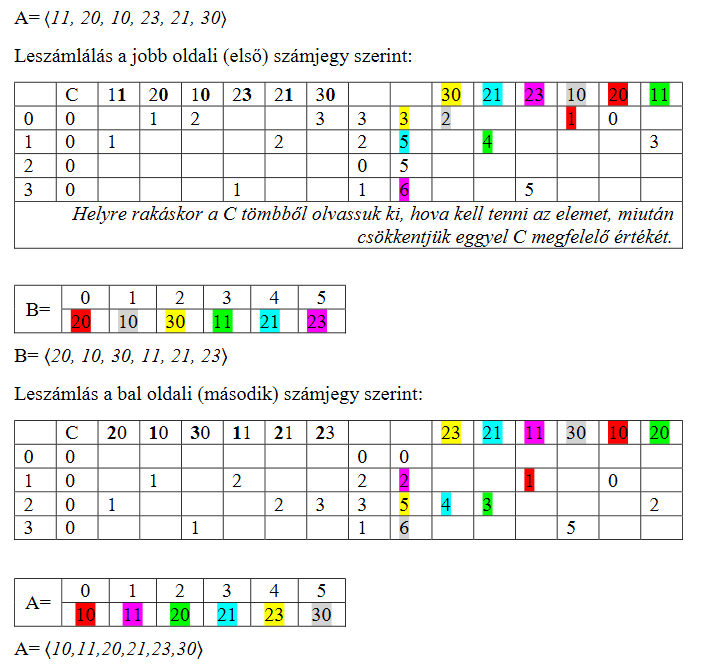
## Szétválogató rendezés (distributing sort)

* Láncolt listákra optimális
* Kulcskiválasztó függvény: ϕ (x)
  + Kinyeri a megfelelő számjegyet az adott számból
* 
* Ezt magában nem használjuk, csak a radix rendezés segédfüggénye (ciklusmag)
  + Inkább egy általános megfogalmazás, mindjárt megvalósítjuk radix-ban

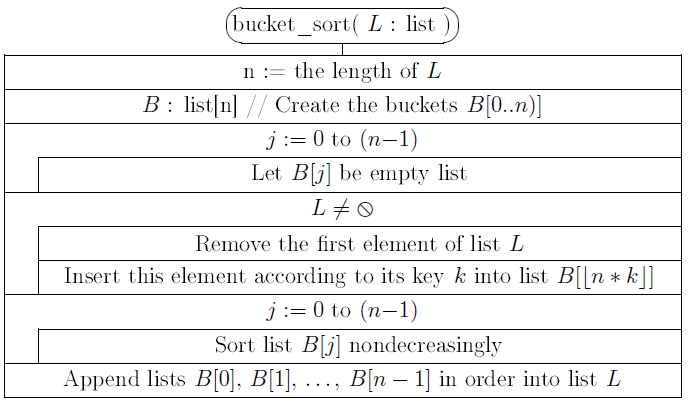
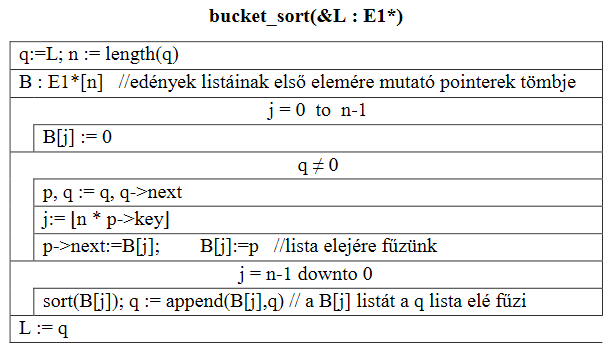
## Radix rendezés implementációja C2L láncolt listákra

* Műveletigény: T(n) ∈ Θ (n)
* Distributing sort-ot használja fel
* (itt az polc szót egy számjegyhez tartozó listaként használom)
* radixSort()
  + 
  + Első ciklus: B inicializálása gyakorlatilag
  + Második ciklus:
    - jobbról balra szétosztjuk az összes számjegyet
    - majd visszafűzzük őket
  + Tradix\_sort(n, d, r) ∈ Θ ( n + d\*(n + r) )
* distribute()
  + 
  + Szétosztjuk az L elemeit a B-ben lévő polcokra az *i* helyiérték alapján
    - Tehát a megfelelő számjegy-lista végére fűzzük az elemet
  + Tdistribute (n) ∈ Θ (n) (ahol n = L elemszáma)
* gather()
  + 
  + Tgather (r) ∈ Θ (r)
* append()
  + 
  + p: eredeti lista vége
  + q: polc lista eleje
  + r: polc lista vége
  + Utolsó sor: polc lista üresre állítása
  + Tappend ∈ Θ (n)

## Leszámláló rendezés (counting sort)

* Radix sort ideális segédrendezése, ha az eredeti sorozat egy tömb
* Stabil
* Műveletigénye: T(n) ∈ Θ (n)
* ϕ: keresett számjegyet adja vissza, r: számrendszer alapja
* 
  + Utolsó előtti ciklus: (k = 1 to r-1)
    - Minta táblázat első színezett oszlopa
    - Eredmény: C[k] azt mutatja meg, hogy hány k-nál kisebb-egyenlő értékű elem van
  + Utolsó ciklus: (i = n-1 downto 0)
    - Lényeg: visszafelé elkezdem olvasni az eredeti tömböket, és a counter tömb alapján megnézem hova kéne mennie, illetve update-elem a countert
    - Tehát megkeresem counterben mennyi az értéke ennek a számjegynek, azt eggyel csökkentem (≤ k miatt), majd a csökentett érték lesz az indexe az eredmény tömbben
* 

## Edényrendezés (bucket sort)

* Feltesszük, hogy a rendezendő elemek kulcsai a [0; 1) valós intervallum elemei.
  + Ha nem, de tudunk alsó-felső korlátokat mondani, könnyen átalakíthatjuk [0; 1)-ekké
* Csak akkor hatékony, ha az input kulcsai a [0; 1) valós intervallumon egyenletesen oszlanak el.
  + Így egy bucketben lévő számok várhatóan kevesen lesznek
* Ha az eredeti elemek kulcsai nem [0, 1) között vannak, akkor is lehet normálni őket oda
  + ha minden kulcsra: kulcs ∈ [min, max) → (kulcs – min) / (max – min) ∈ [0, 1)
* Alapból nem garantáltan stabil! (belső segédrendezésen múlik)
* Műveletigény:
  + mT (n) ∈ Θ (n)
  + AT (n) ∈ θ (n) (ha egyenletesen oszlanak el)
  + MT (n) az edényeken belüli rendezéstől függ (ha mindegyik egy edénybe kerül)
* bucket\_sort()
  + 
* listával bucket sort()
  + 

# Hasító táblák

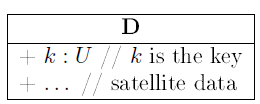
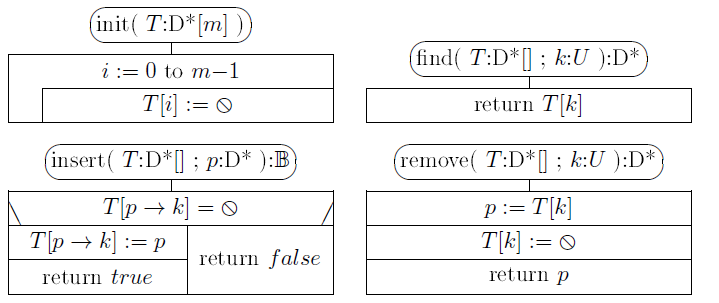
Hasító tábláknál műveletigények általában:

* Inicializálásra: T ∈ Θ (n)
* Keresés, törlés, beszúrásra:
  + AT ∈ Θ (1)
  + MT ∈ Θ (n)

## Jelölések

* m : a hasító tábla mérete
* T[0..m) : a hasító tábla
* T[0]; T[1]; …. ; T[m - 1] : a hasító tábla rései (slot-jai)
* ∅ : üres rés a hasító táblában (direkt címzésnél és a kulcsütközések láncolással való feloldása esetén)
* E : üres rés kulcsa a hasító táblában (nyílt címzésnél)
* D : törölt rés kulcsa a hasító táblában (nyílt címzésnél)
* n : a hasító táblában tárolt adatok száma
* α = n = m : a hasító tábla kitöltöttségi aránya (load factor)
* U : a kulcsok univerzuma; k; k’; ki ∈ U
* h : U → 0..(m - 1) : hasító függvény

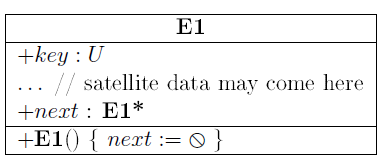
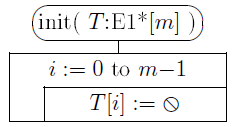
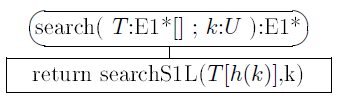
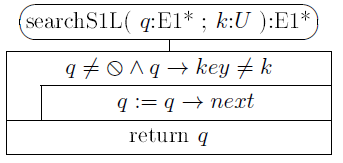
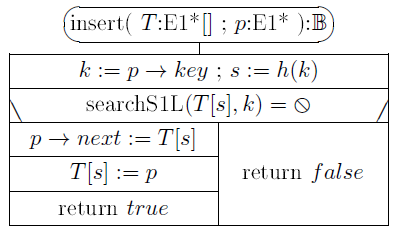
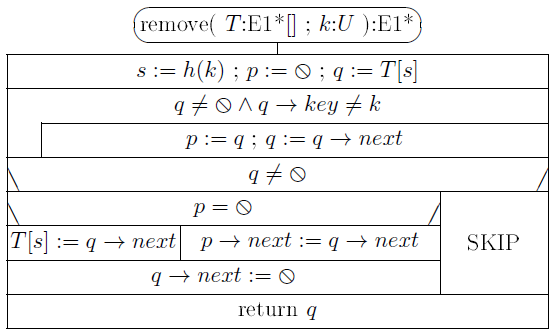
## Direkt címkézéses tábla

* Feltételek
  + U = [0..m)
  + m ≥ n, de m nem túl nagy (nagyjából max kétszer akkora)
* Lényeg: kulcs direkt megegyezik a táblában a slot-okkal
* D: adatrekord típusa
  + 
* műveletek:
  + 

## Hasító függvényes táblák

* Ha direkt címkézés nem alkalmazható/gazdaságos (pl mert m túl nagy n-hez képest)
* Hasító függvényeket alkalmazunk, ez adja meg hogy melyik „résbe” kerüljenek az adatok
  + Így: a k kulcsú adatot a táblának a T[ h(k) ] résben tároljuk
* Egyszerű egyenletes hasítás: ha a kulcsokat a rések között egyenletesen szórja szét
* Kulcsütközés: ha két különböző adatra ugyan azt az rést adja vissza a hasító függvény

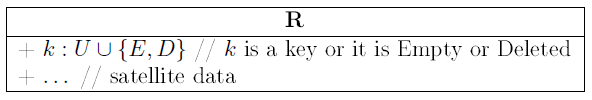
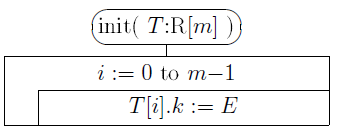
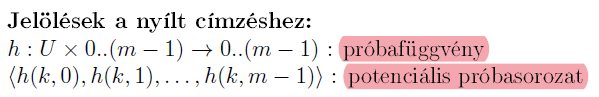
## Kulcsütközések feloldása láncolással

* Hasítótábla rései egyszerű láncolt listák
* Művelet igény:
  + Tinit(m) ∈ Θ(m).
  + többi művelet:
    - mT ∈ Θ(1), MT(n) ∈ Θ(n)
    - AT(n, m) ∈ Θ (1 + n/m)
* 
* init()
  + 
* Keresés
  + 
  + 
* insert()
  + 
* remove()
  + 

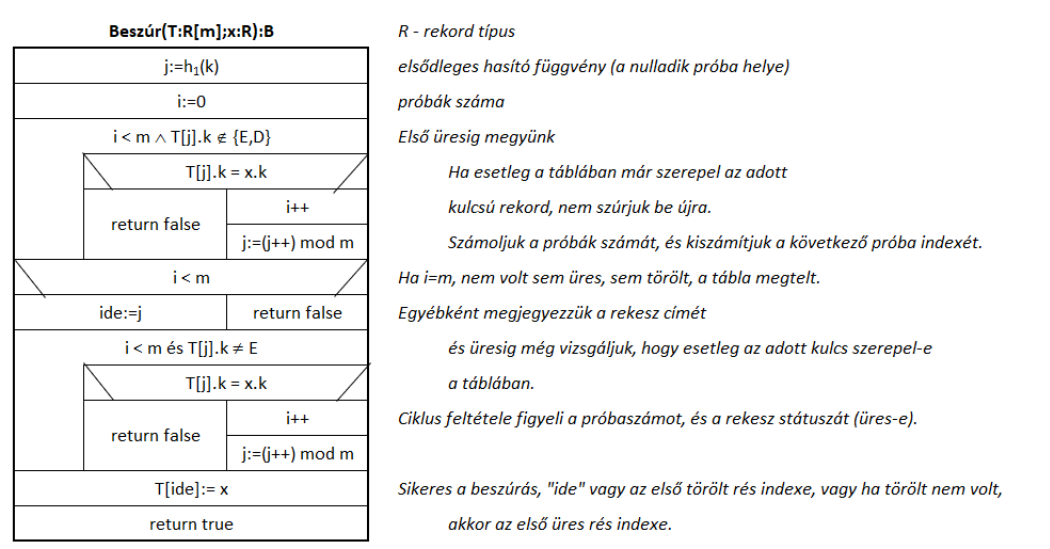
## Jó hasító függvények

* Osztó módszer: h(k) = k mod m
  + ha a kulcsok egész számok
  + akkor jó, ha az m olyan prím, ami nincs közel kettő hatványhoz
* Kulcsok [0 ; 1) intervallumon: h(k) = Alsórész(k\*m)
  + bucket sort alapja gyakorlatilag
* Szorzó módszer: h(k) = Alsórész( {k\*A} \* m )
  + A: konstans, 0 ≤ A < 1
    - okosok szerint jó: A = (sqrt(5-1))/2 = 0,618
  + {x} = x törtrésze
    - tehát: 0 ≤ {k\*A} < 1
  + akkor jó, ha a kulcsok valós számok (gyakorlatilag bucket sort általánosítása)
    - „nem érzékeny a hasító tábla méretére” ??

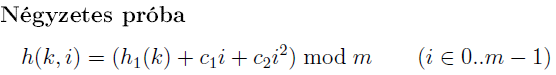
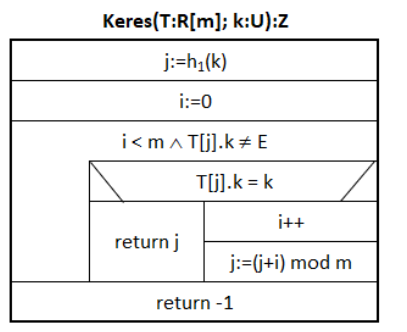
## Nyílt címzés

* (ezt a részt már nem volt időm rendesen kidolgozni, de a fontos stukik benne vannak)
* Feltétel: Nincsenek duplikált kulcsok
* R: Adatrekord, közvetlen a tábla réseiben vannak
  + 
* 
* 

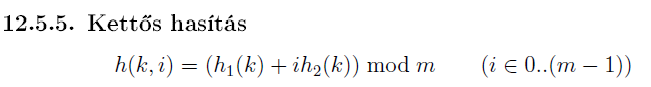
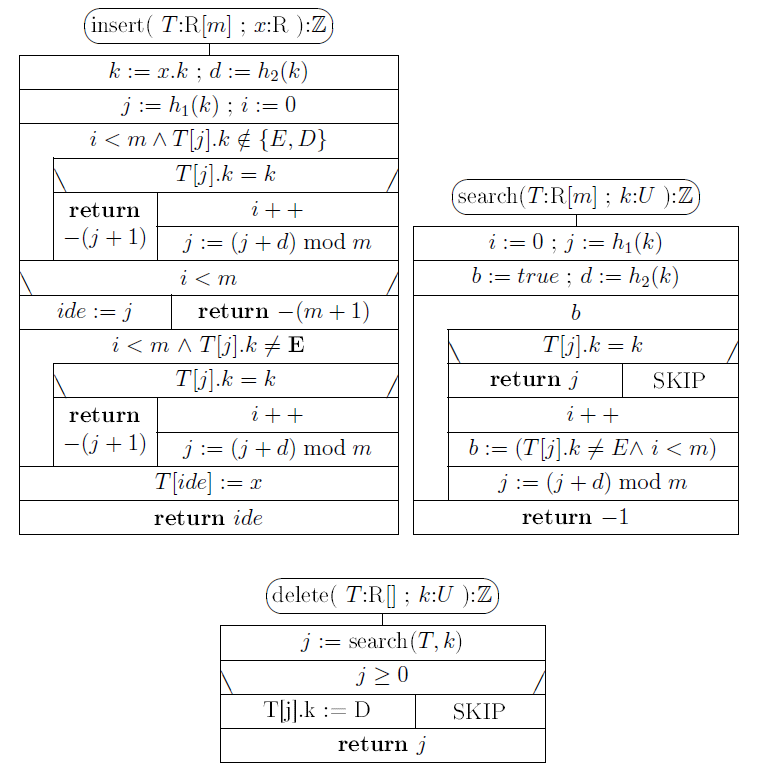
### Lineáris próba:

* 
  + 

### Négyzetes próba

* 
  + c = 0.5 jó választás
  + 

### Kettős hasítás

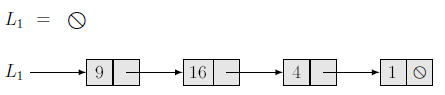
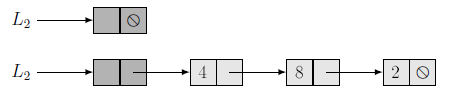
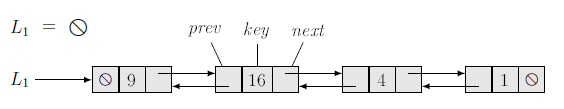
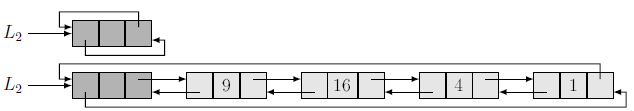
* 
  + 

# Függelék

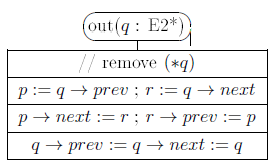
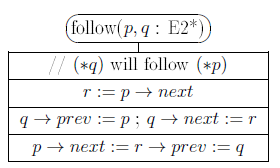
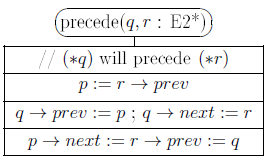
Gyakorlatilag a ZH1 anyagából ami még kellhet ZH2-höz

## Listák:

### Jelölések:

* S1L: Simple 1-way List
  + 
* H1L: Header node (fejelem) + 1-way List)
  + 
* S2L: Simple 2-way List
  + 
* C2L: Cyclic 2-way List (fejelemmel)
  + 

### C2L alap műveletei:

* jelölésel:
  + p: előző
  + q: beszúrandó/keresendő/stb elem
  + r: következő elem
* Ha az L lista végére akarok beszúrni q-t:
  + follow( L->prev, q )
  + precede( q, L )
* 

## Műveletigény jelölések

* T(n): műveletigény
* Függvények jellemzése:
  + Theta: T (n) ∈ Θ(n): aluról és felülről is az n pozitív együtthatós lineáris függvényeivel becsülhetőek (**Általános becslés, „kb ennyi idő alatt”**)
  + Omega: Ω (g) olyan fgv-ek, amelyek aszimptotikusan kisebbek a g-nél (elhanyagolhatók g-hez képest) (**alsó becslés**)
  + Nagy ordó: O(g): az maximum g-vel arányos (**felső becslés**)
* Programok műveletigényének jellemzése:
  + Minimális műveletigény: mT(n)
  + Maximális műveletigény: MT(n)
  + Átlagos műveletigény: AT(n)
    - (részletes számítás nem kell, általában fele/kicsit nagyobb a max-nak)