Prioritní fronty

- uchovávání množiny S prvků, prvky mají položku key
- (max) nebo (min) - prioritní ~~queue~~ fronta

OPERACE:

INSERT (S, x)

MAXIMUM (S) // returns element with the largest key

EXTRACT-MAX (S) // returns and removes el. s největším klíčem

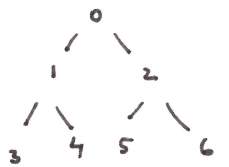
INCREASE-KEY (S, x, k) // zvýší klíč položky x na hodnotu k

Případně další operace: např. union.

Binary heap:

pole A
heap-size // počet položek ve Frontě

Př.



$i \to 2i+1$ left
$\to 2i+2$ right

$j \to \lceil \frac{j}{2} \rceil - 1$ parent

A: [0|1|2|3|4|5|6]

heap-size = 7

Max-heap vlastnost: $A[parent(i)] \geq A[i]$ pro $i > 0$


```
MAX-HEAPIFY (A, i)        // ASSUMPTION (left, right pouze OK)
  l ← left(i)
  r ← right(i)
  if l ≤ heap-size and
     A[l] > A[i]
    largest ← l
  else
    largest ← i
  if r ≤ heap-size and
     A[r] > A[largest]
    largest ← r
  if largest ≠ i
    A[i] ↔ A[largest]
    MAX-HEAPIFY (A, largest)
```

Kde je největší prvek?
Buďto v poloze i nebo některém potomkovi v testech
l < heap-size
r < heap-size

největší do kořene.
rekurzivně opravíme správný podstrom.

podstromu

Složitost: $O(h)$, h je výška uzlu i. Tj. $O(\lg n)$, kde n je počet uzlů v podstromu.

```
HEAP-MAXIMUM (A)
  return A[0]
```

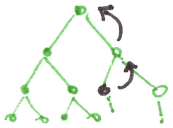
```
HEAP-EXTRACT-MAX (A)
  if heap-size(A) < 1          // prázdná Fronta
    return nil
  max ← A[0]
  A[0] ← A[heap-size(A)-1]     // poslední prvek do kořene
  heap-size(A) ← heap-size(A) - 1
  MAX-HEAPIFY (A, 1)           // probublám jej dolů
  return max
```

Složitost: $O(\lg n)$

Heap-increase-key(A, i, key) // bublání nahoru

```
assert ( key > A[i])
A[i] ← key
while i > 0 and A[parent(i)] < A[i]
    A[i] ⇔ A[parent(i)]
    i ← parent(i)
```

Složitost: $O(\lg n)$



Heap-Insert (A, key)

```
heap_size(A) ← heap-size(A) + 1
A[heap-size(A)-1] ← -∞
Heap-increase-key (A, heap-size(A), key).
```

$O(\lg(n))$

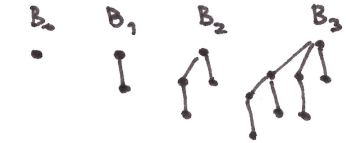
TABULKA SLOŽITOSTI

	Heap	Binomial	Fibonacci
MAKE-~~HEAP~~	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(\lg 1)$	$\Theta(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
~~DELETE~~	~~$\Theta(\lg n)$~~		

Binomial-trees

B_k .... dva B_{k-1} spojeny do jednoho
první je zapojen jako nejlevější potomek druhého

B_0 .... jeden uzel



vlastnosti

B_k

a) obsahuje 2^k uzlů
b) výška je k
c) kořen má stupeň k.

} Indukce: krokem při konstrukci zdvojnásobíme počet uzlů a zvedneme výšku o 1. Počet potomků kořene se zvedne o 1

Počítáno zprava pro potomci kořene vlastně
$B_0, B_1, B_2 \dots B_{k-1}$

Indukce: Tvrzení platí pro B_1 a B_0.

Pokud platí pro B_{k-1}, pak B_{k-1} přidáme nejvýše doleva kopii B_{k-1}, a tedy nanejvýš dostaneme B_k.
Máte Tvrzení tedy platí.

def:

Binomial-heap: H = množina binomial stromů (s klíči v uzlech)

→ každý strom splňuje min-heap property:

$A \leq B, C$

platí v každém uzlu A (kde B,C nevadí)

→ pro $\forall k$ obsahuje max 1 kopii B_k

Vlastnosti:

→ kořeny stromů obsahují z daného stromu min. klíč.

→ Binomial heap s n uzly obsahuje nejvýše $\lfloor \lg n \rfloor + 1$ stromů.

Proč?

$m = \lfloor \lg n \rfloor + 1$

$n = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 + 2^2 + \dots + b_m 2^m$

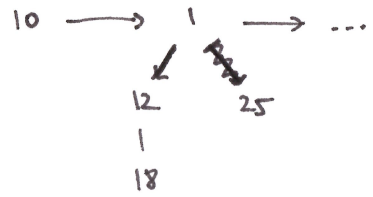
$b_i = 0$ neobsahuje B_i

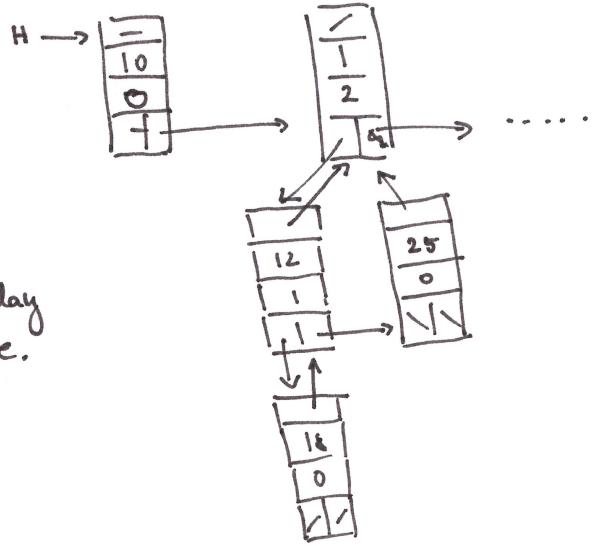
$= 1$ obsahuje B_i

pak součet velikostí stromů sedí na n.

Implementace

```
struct {
    key,
    child,      // nejlevější potomek
    sibling,    // první sourozenec
    degree,     // stupeň
    parent,     // rodič
}
```





Celý binomial heap je seznam kořenů (pomocí sibling).

kořeny jsou uspořádány podle položky degree. (vzestupně)

BINOMIAL-HEAP-MINIMUM (H)

```
y ← NIL
x ← H.
min ← ∞
while x ≠ NIL
    if x.key < min
        min ← x.key
        y ← x
    x ← x.sibling
return y
```

// prvek procházení seznamu a nalezení minimálního klíče

složitost: $\Theta(\lg(n))$

n je počet prvků v heapu.

seznam je $\Theta(\lg(n))$ dlouhý.

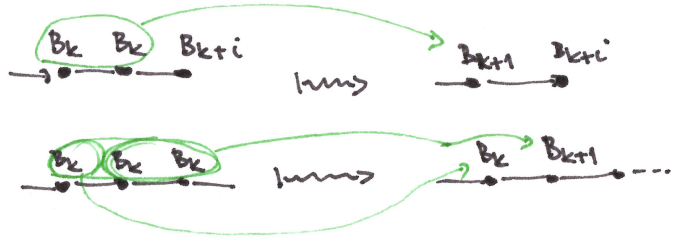
BINOMIAL-HEAP-UNION (H_1, H_2)

Idea:

1) zmergujeme stromy do 1 seznamu se zachováním uspořádání

[může dojít 2 kopie B_k pro nějaké k]

⇒ řeším to tak, že vzniklý seznam projdu zleva a 2 (a 3) následný výskyt B_k řeším spojením dvou kopií B_k do jedné kopie B_{k+1}



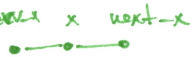
spojení udělám tak, aby byl v kořeni min. klíč.

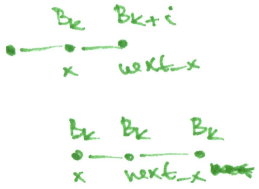
BINOMIAL-LINK (Y, z) // napoj y jako potomka z

```
z.p ← y
y.sibling ← z.child
z.child ← y
z.degree ← z.degree + 1
```

BINOMIAL-HEAP-UNION (H1, H2)

```
H ← BINOMIAL-HEAP-MERGE(H1, H2)
if H == NIL  return NIL
prev_x ← NIL
x ← H
next_x ← x.sibling
while next_x ≠ NIL
    if (x.degree ≠ next_x.degree) or
       (next_x.sibling ≠ NIL AND
        next_x.sibling.degree = x.degree)
        prev_x ← x
        x ← next_x
    else if x.key ≤ next_x.key
        x.sibling ← next_x.sibling
        BINOMIAL-LINK(next_x, x)
    else { if prev_x = NIL
             H ← next_x
           else x_prev.sibling ← next_x
           BINOMIAL-LINK(x, next_x)
           x ← next_x
    next_x ← x.sibling.
return H
```





$\Theta(\lg n)$

$\Theta(\lg n)$

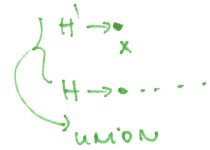
Složitost $O(\lg n_1) + O(\lg n_2)$

n_1 velikost H_1

n_2 velikost H_2

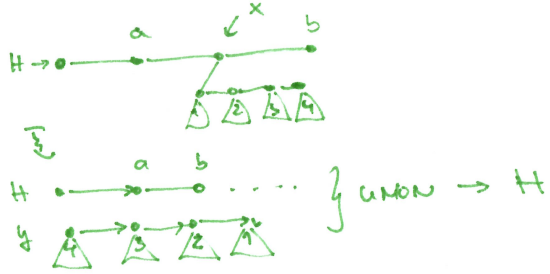
BINOMIAL-HEAP-INSERT (H, x)

```
H' ← NEW_HEAP()
H ← x
x.parent ← NIL
x.child ← NIL
x.sibling ← NIL
x.degree ← 0
H ← BINOMIAL_HEAP-UNION(H, H')
```



BINOMIAL-HEAP-EXTRACT-MIN (H)

```
x ← remove-min-list(H)      // vrátí min kořen ze seznamu kořenů, a vypojí ho
y ← reverse_list(x.child)   // otočím seznam potomků uzlu x a dostanu tak Heap !
H ← BINOMIAL-HEAP-UNION(H, y)
return x
```



BINOMIAL-HEAP-DECREASE-KEY (H, x, k)

```
assert(x.key >= k)
x.key ← k
    y ← x
    z ← y.parent

while z ≠ NIL and y.key < z.key      // „bublám" hodnotu nahoru, dokud nejsem v kořeni nebo není rodič menší než já
        Y.key ↔ z.key
        y ← z
        z ← y.parent
```

Složitost: $\Theta(\lg n)$