Dynamické datové struktury II.

Halda. Prioritní fronta. Aplikace.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Halda
2. Operace nad haldou
   - Inicializace
   - Oprava haldy nahoru
   - Oprava haldy dolů
   - Přidání prvku do haldy
   - Nalezení maxima
   - Smazání kořene
   - Třídící algoritmus HeapSort
3. Prioritní fronta

1. Halda

tzv. Heap.

Patří mezi rekurzivní datové struktury.

Reprezentována binárním stromem.

Volnější definice než BST.

Varianty:

- MinHeap: v kořeni minimum, běžná varianta.
- MaxHeap: v kořeni maximum, použití u prioritní fronty.

Vlastnosti min-haldy:

Hodnota v každém uzlu menší než v libovolném z potomků.

V hladinách $1, x-1$ maximální počet prvků n_x

$$n_x = 2^{x-1}.$$

V hladině x prvky řazeny co nejvíce vlevo.

2. Halda, další vlastnosti

Vlastnosti:

- V kořenu minimální prvek, někde v úrovni x maximální prvek.
- Nalezení minima s konstantní složitostí, nalezení maxima se složitostí $O(n)$.
- Při vkládání prvku nemusíme měnit tvar stromu, provádíme pouze vzájemné výměny prvků ve stromu.
- Odpadá nutnost převažovat strom.

Implementace haldy:

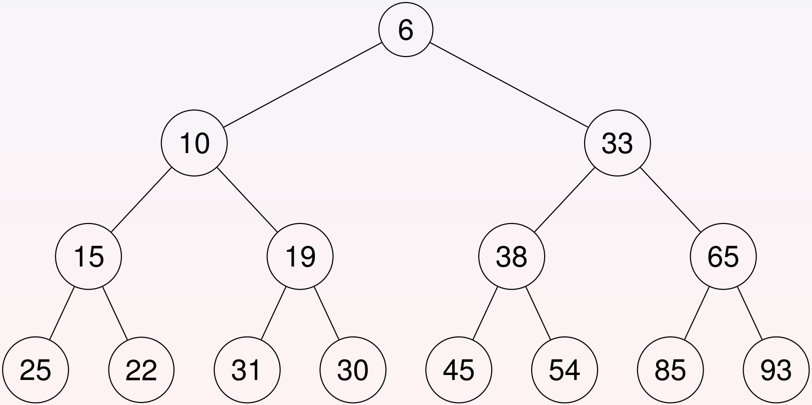
- binárním stromem,
- 1D polem.

Varianta 2:

Častější (nemění se tvar stromu), nutno však znát horní odhad počtu prvků.

Rychlý přístup k prvkům.

3. Ukázka haldy



4. Reprezentace haldy polem

Definice:

Halda představuje takovou posloupnost prvků $h_1, h_2, ..., h_n$, kdy pro každý index $i = l, ..., n/2$ platí

$$h_i \leq h_{2i} \wedge h_i \leq h_{2i+1}.$$

Halda definována posloupností $h_1, h_2, ..., h_n$.

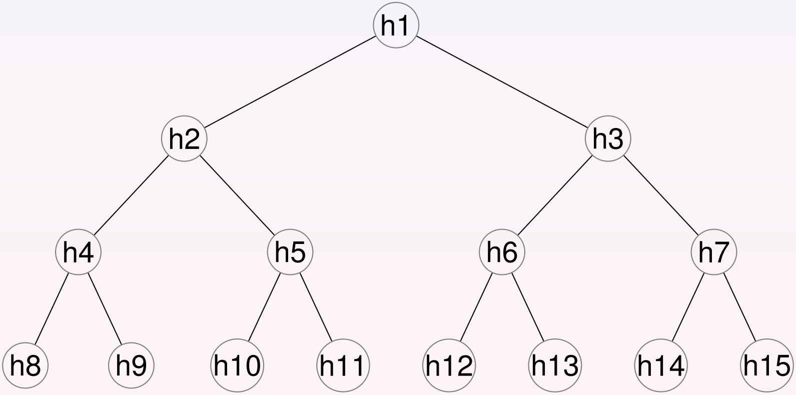
Všichni leví potomci mají sudý index.

Všichni praví potomci mají lichý index.

Př.: Kořen stromu h_1, h_2 levý potomek, h_3 pravý potomek.

Levý potomek uzlu h_2 je h_4, pravý potomek je h_5, atd.

5. Ukázka haldového stromu



h[i]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
h_i	-	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}	h_{11}	h_{12}	h_{13}	h_{14}

6. Reprezentace haldy polem, důsledky

Z důvody neměnnosti tvaru stromu výhodná implementace polem.

Vztah mezi pořadovým číslem uzlu h_i a indexem uzlu $h[i]$ v poli

$$h_i \approx h[i].$$

Vztah mezi prvkem $h[i]$ a rodičem r

$$r = h[i/2].$$

Vztah mezi prvkem $h[i]$ a levým potomkem l

$$l = h[2i].$$

Vztah mezi prvkem $h[i]$ a pravým potomkem p

$$p = h[2i+1].$$

7. Operace nad haldou

Základní operace nad heapem:

- inicializace heapu (Init): $O(1)$,
- přidání prvku do heapu (Insert): $O(\log(n))$,
- oprava heapu (Heapifying): $O(\log(n))$:
  Od listu ke kořeni: fixHeapUp.
  Od kořene k listu: fixHeapDown.
- nalezení minima: $O(1)$, efektivní,
- nalezení maxima: $O(n)$, velmi neefektivní !
- smazání minima: $O(\log(n))$,
- tisk: $O(n)$.

8. Reprezentace haldy a inicializace

Proměnná *n* uchovává aktuální velikost haldy.

Velikost haldy inicializována na hodnotu *maxN*.

Složitost $O(1)$.

```
class Heap:
  def __init__(self, max_n):
      self.n = 0                #Pocet prvku v heapu
      self.h = [0]*(max_n+1)    #Maximalni velikost
```

9. Oprava haldy nahoru

Pohyb haldou od uzlu U směrem ke kořeni haldy, nemá vliv na potomky U.

Stromová reprezentace:

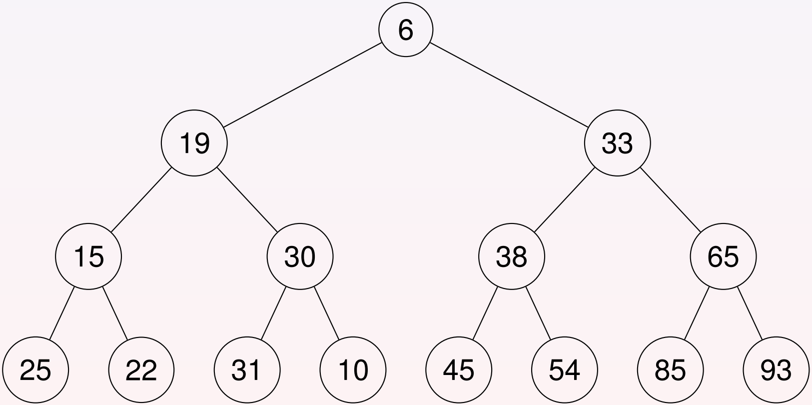
- Pokud předchůdce P uzlu U má vyšší hodnotu, prohodíme $U \Leftrightarrow P$.
- Pokračujeme v předchůdci $U = P$, dokud U není kořen.

Reprezentace polem:

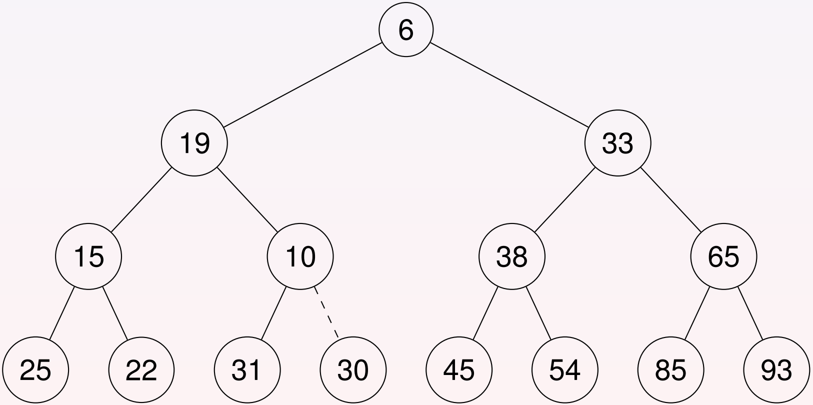
- Pokud prvek $[i/2]$ má menší hodnotu než prvek $[i]$, prohodíme $[i/2] \Leftrightarrow [i]$.
- Pokračujeme v předchůdci $i = i/2$ dokud $i > 1$.

```
def fixHeapUp(i):
    while (i > 1 and (self.h[i//2] > self.h[i])):  #Opakuj pro rodice
        temp = self.h[i//2]       #Prohozeni
        self.h[i//2] = self.h[i]
        self.h[i] = temp
        i = i // 2            #Jdi na rodice
```

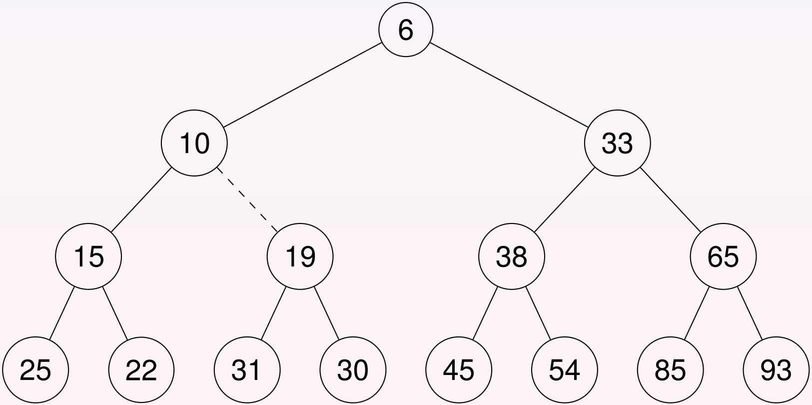
10. Ukázka opravy haldy nahoru



11. Ukázka opravy haldy nahoru (30<->10)



12. Ukázka opravy haldy nahoru (10<->19)



13. Oprava haldy dolů

Pohyb haldou od uzlu U směrem od kořene k listům haldy, nemá vliv na předchůdce U.

Stromová reprezentace:

- Najdi následníky uzlu U: V_L, V_P.
- Pokud $V_P < V_L$, následník $V = V_P$, jinak $V = V_L$ (přidáváme do menšího).
- Pokud $U > V$, prohoď $U \Leftrightarrow V$, jinak ukonči.
- Pokračuj v prohozeném vrcholu, dokud nedosáhneme listu.

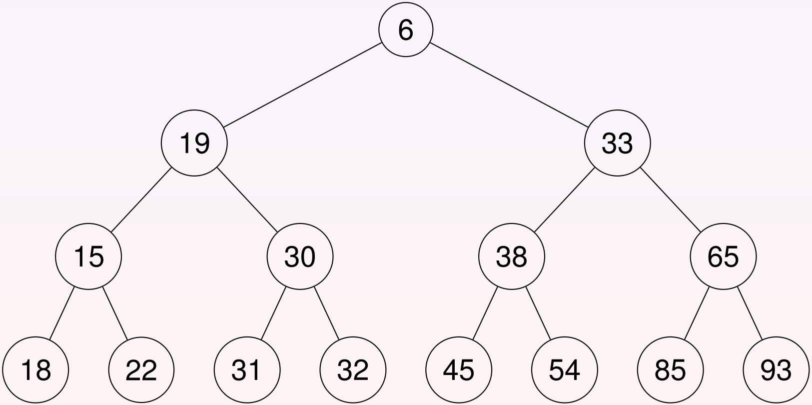
Reprezentace polem:

- Najdi následníky uzlu $[i]$: $[2 * i], [2 * i + 1]$.
- Pokud $[2 * i + 1] < [2 * i]$, následník $[v] = [2 * i + 1]$, jinak $[v] = [2 * i]$ (přidáváme do menšího).
- Pokud $[i] > [v]$, prohoď $[i] \Leftrightarrow [v]$ jinak ukonči.
- Pokračuj v prohozeném vrcholu, dokud nedosáhneme listu

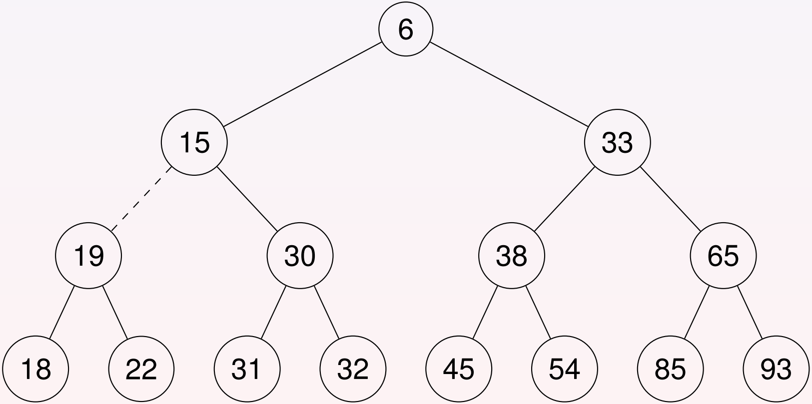
14. Algoritmus pro opravu haldy dolů

```
def fixHeapDown(i):
   while 2 * i <= self.n:    #Existuji oba potomci?
       k = 2 * i             #Levy potomek
       if (k < self.n) and (self.h[k + 1] < self.h[k]): #Porovnani L a
           k = k + 1;    #Index ukazuje na mensiho potomka
       if  self.h[i] >self. h[k]: #Nesplneny podminky, prohodit
           temp = self.h[i]
           self.h[i] = self.h[k]
           self.h[k] = temp
       else:
           break         #Vse v poradku, netreba prohazovat
       i = k             #Pokracuj od prohozeneho potomka
```

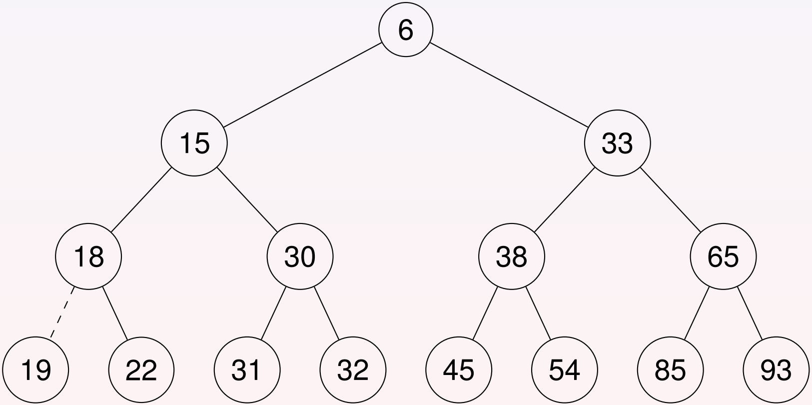
15. Ukázka opravy haldy dolů



16. Ukázka opravy haldy dolů (19<->15)



17. Ukázka opravy haldy dolů (18<->19)



18. Přidání prvku do haldy

Složitost $O(\log(N))$.

Nevhodná konfigurace vstupních dat: v každém kroku přidáváme největší prvek.

V takovém případě musí být opravován strom ke kořeni.

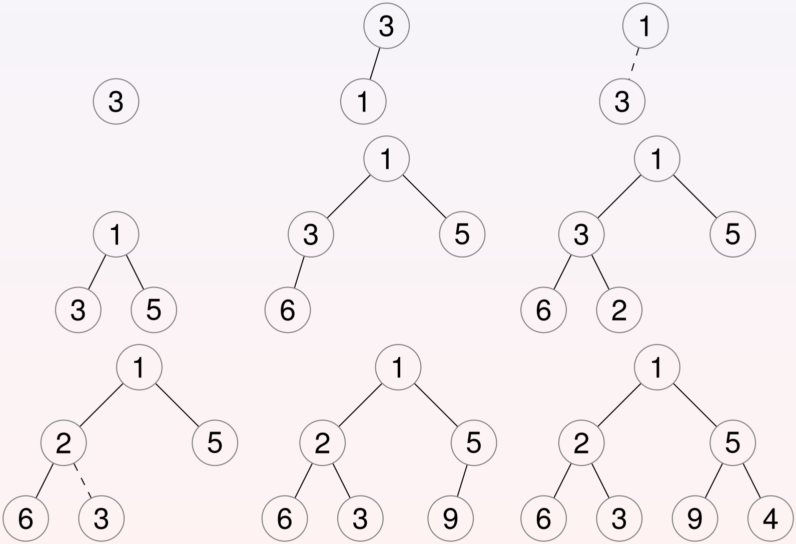
Postup:

Nový uzel s hodnotou x vytvořen na poslední hladině co nejvíce vpravo.

Oprava haldy nahoru U.

```
def add(self, item):
    self.n = self.n + 1         #Prodlouzeni haldy o 1 prvek
    self.h[self.n] = item       #Pridej na konec haldy
    self.fixHeapUp(self.n)      #Oprav strom od tohoto prvku
```

19. Ukázka budování haldy



20. Nalezení maxima v haldě

Maximum se nachází v některém z listu.

Z definice haldy nevyplývá, v kterém listu bude ležet.

Nutno projít všechny listy, tj. posledních $n/2 - 1$ prvků: $h[n/2], ..., h[n-1]$.

Složitost operace $O(n)$.

Překvapivě neefektivní oproti BST.

```
def max(self):
    max_el = self.h[self.n//2]          #Inicializuj maximum
    for i in range(self.n//2+1, n):     #Projdi zbyvajici listy
        if self.h[i] > max_el:          #Aktualizuj maximum
            max_el = self.h[i]

    return max_el
```

21. Smazání kořene

Minimum se nachází v kořeni.

Důsledkem zmenšení haldy o jeden prvek.

Složitost $O(\log(n))$, nutno opravit strom.

Postup:

- Prohození prvků $h[1] \Leftrightarrow h[n]$.
- Zmenšení velikosti o 1 prvek.
- Oprava haldy z vrchu od kořene dolů.

```
def delRoot(self):
    temp = self.h[1]                #Prohozeni h[1] <-> h[n]
    self.h[1] = self.h[self.n]
    self.h[self.n] = temp
    self.n = self.n - 1             #Zmenseni velikosti o 1
    self.fixHeapDown(1)             #Oprava haldy dolu
```

22. Použití haldy

3 základní aplikace:

- *HeapSort*
  Třídící algoritmus, nestabilní.
  Složitost $O(n \log n)$.
- *Prioritní fronta*
  Implementace prioritní fronty.
  Efektivnější než lineární seznam.
  Operace $O(\log n)$.
- *Hledání $k-$tého nejmenšího prvku*
  Opakované odebrání kořene $(k-1)$ x.

23. Heap Sort

Probíhá nad vybudovanou haldou, triviální implementace.

Opakuj, dokud halda neobsahuje pouze kořen:

- Prohození kořene (minima haldy) s prvkem haldy nejnižší úrovně nejvíce vpravo.
- Oprava haldy dolů (v kořeni není minimum).
- Zkrácení haldy o prvek nejnižší úrovně nejvíce vpravo.

Nad polem snadná algoritmizace.

Opakuj, dokud $n > 1$

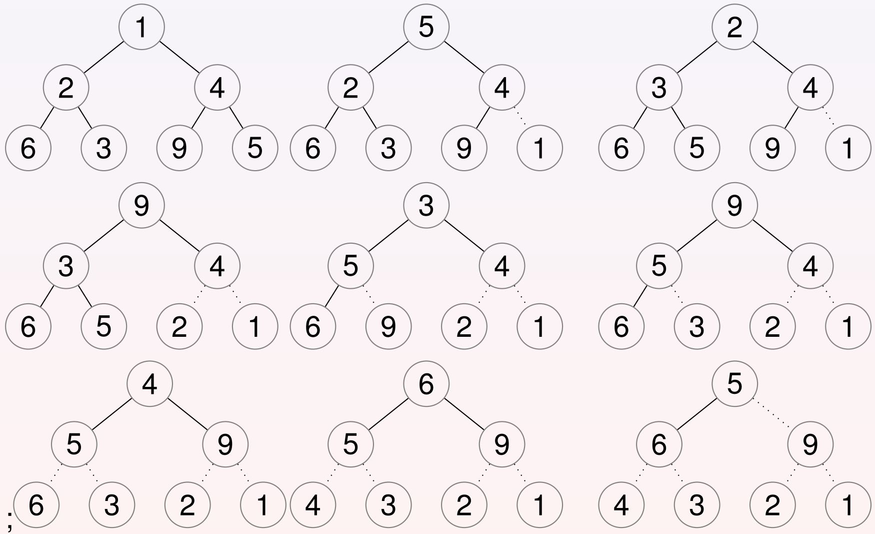
- Prohodíme $h[1]$ a $h[n]$.
- Provedeme opravu haldy dolů z kořene.
- Dekrementujeme n.

První výběr: nejmenší prvek.

Druhý výběr: druhý nejmenší prvek, atd...

Jsou řazeny za koncem zkracované haldy sestupně.

24. Ukázka činnosti algoritmu



25. Algoritmus Heap Sort

Pouze třídící procedura:

```
def heapSort(self, X):
   #Pridej prvek do heapu
   for x in X:
       self.add(x)
   #Opakovane  maz koren
   while (self.n > 1):
      self.delRoot()
      return self.h[1:len(self.h)]
```

Vlastnosti algoritmu:

Algoritmus má složitost $O(n \log n)$.

Asi o 20% pomalejší než MergeSort

Asi o 50% pomalejší než QuickSort.

26. Prioritní fronta

Priority Query.

Někdy nazývána “fronta s předbíháním.

Široce používaná struktura pracující s prvky nestejné váhy.

Prioritní fronta představuje strukturu dvojic

`<weight, value>`

uspořádanou sestupně dle hodnot `weight` (priorita).

Priorita určena ohodnocovací funkcí, udává “význam” prvku.

Prvek s vyšší prioritou může přeběhnout prvek s nižší prioritou.

Princip prioritní fronty:

Prvky přidávány v pořadí na konec fronty.

Odebírány na základě hodnoty klíče.

V každém kroku odebrán prvek s nejvyšší prioritou.

27. Základní operace

Implementace prioritní fronty: lineární seznam, BST, halda (preferována).
Velké rozdíly ve výkonnostních charakteristikách!

Metoda	push	pop	find
Neuspořádaný seznam	$O(1)$	$O(n)$	$O(n)$
Uspořádaný seznam	$O(n)$	$O(1)$	$O(1)$
BST, vyvážený*	$O(h)$	$O(h)$	$O(h)$
Heap	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

*Pro AVL stromy $h = 1.4\log(N)$, pro degenerované případy BST $h = N - 1$, neefektivní.

Operace nad prioritní frontou:

- Vytvoření fronty.
- Přidání položky.
- Smazání položky s největší prioritou
- Změna priority položky.
- Výmaz libovolné položky.
- Smazání fronty.

28. Ukázka prioritní fronty

Weight	Value
204	123
197	239
152	246
142	177
105	432
97	404
92	44
88	164
53	149
45	186
13	98

Weight	Value
204	123
197	239
97	404
152	246
142	177
88	164
13	98
105	432
53	149
92	44
45	186

V praxi nemusí být prvky setříděny.

Na správné pozici pouze první prvek.

29. Implementace PQ pomocí Heapu

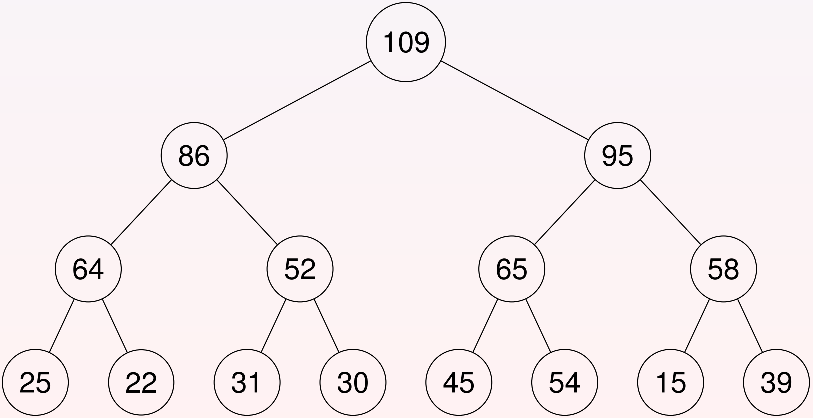
Standardní implementace haldy.

Složitost operací $O(\log(n))$, na rozdíl od BST netřeba převažovat.

Využívána datová struktura **maxHeap:**

V kořeni maximum, nejmenší hodnoty v listech.

U `fixHeapUp()` a `fixHeapDown()` záměna $<$ za $>$.



30. Implementace PQ

Třída PQItem

```
class PQItem:
    def __int__(self, w, value):
        self.w = w;
        self.value = value
```

Třída PQ, princip kompozice:

```
class PQ:
    def __init__(self, max_size):
        self.n = 0
        self.h = [PQItem] * (max_size + 1)
```

Položky haldy objekty typu PQItem.

V praxi nejčastěji implementovány operace push() a pop().

31. Implementace PQ

Třída PQItem:

```
class PQItem:
    def __int__(self, w, value):
        self.w = w;
        self.value = value
```

Třída PQ, princip kompozice:

```
class PQ:
    def __init__(self, max_size):
        self.n = 0
        self.h = [PQItem] * (max_size + 1)
```

PQ implementována jako maxHeap.

Změna relačních operátorů při opravě haldy.

32. Oprava haldy

Oprava haldy nahoru: změna relačního operátoru $< \quad \Rightarrow \quad >$

```
def fhu(self, i):
    #Fix heap up
    while (i > 1) and (self.h[i].w > self.h[i//2].w):
        #Swap h[i] and h[i//2]
        temp =  self.h[i//2]
        self.h[i // 2] =  self.h[i]
        self.h[i] = temp
        #Go to parent
        i = i//2
```

Po opravě stromu maximum v kořeni.

Analogie při opravě haldy dolů: $> \quad \Rightarrow \quad <$.

33. Operace push() a pop()

Operace push(): analogie s add().

Složitost $O(\log(n))$.

```
def push(self, item):
    # Increment n
    self.n = self.n + 1
    # Connect to heap
    self.h[self.n] = item
    # Fix heap up
    self.fhu(self.n)
```

Operace pop(): analogie s delRoot().

Složitost $O(\log(n))$.