

Prioritní fronta, halda

Priority queue, heap

Jan Kybic

<http://cmp.felk.cvut.cz/~kybic>
kybic@fel.cvut.cz

2016–2018



Prioritní fronta

Halda

Heap sort

Prioritní fronta

(priority queue)

Podporuje následující operace pro porovnatelné položky

- ▶ přidání položky (*insert*)
- ▶ odebrání **nejmenší** položky (*pop*)

Může podporovat i další operace

- ▶ nedestruktivní čtení ze začátku (*peek*)
- ▶ zjištění počtu položek ve frontě (*size*)
- ▶ změna položky

Prioritní fronta

(priority queue)

Podporuje následující operace pro porovnatelné položky

- ▶ přidání položky (*insert*)
- ▶ odebrání **nejmenší** položky (*pop*)

Může podporovat i další operace

- ▶ nedestruktivní čtení ze začátku (*peek*)
- ▶ zjištění počtu položek ve frontě (*size*)
- ▶ změna položky

Varianty (ekvivalentní)

- ▶ odebíráme největší místo nejmenší
- ▶ ukládáme *klíč* (prioritu) a *hodnotu*, třídíme podle klíče, vracíme hodnotu

Prioritní fronta

(priority queue)

Podporuje následující operace pro porovnatelné položky

- ▶ přidání položky (*insert*)
- ▶ odebrání **nejmenší** položky (*pop*)

Může podporovat i další operace

- ▶ nedestruktivní čtení ze začátku (*peek*)
- ▶ zjištění počtu položek ve frontě (*size*)
- ▶ změna položky

Varianty (ekvivalentní)

- ▶ odebíráme největší místo nejmenší
- ▶ ukládáme *klíč* (prioritu) a *hodnotu*, třídíme podle klíče, vracíme hodnotu

- ▶ Zásobník i fronta jsou speciálním případem prioritní fronty.

Aplikace prioritní fronty

- ▶ Hledání k nejmenších prvků
- ▶ Prioritní rozvrhování a plánování (fronta s předbíráním)
- ▶ Simulace diskrétních událostí (priorita=čas)
- ▶ Základní prvek mnoha algoritmů
 - ▶ Heapsort (třídící algoritmus)
 - ▶ Huffmanovo kódování
 - ▶ Informované prohledávání grafu
 - ▶ Dijkstrův algoritmus pro hledání nejkratší cesty v grafu
 - ▶ Primův algoritmus pro hledání minimální kostry v grafu.

Implementace prioritní fronty

implementace	vkládání	nalezení minima	odebrání
pole	$O(1)$	$O(n)$	$O(n)$
binární strom	$O(\log n)$	$O(\log n)$	$O(\log n)$
binární halda	$O(1) / O(\log n)^*$	$O(1)$	$O(\log n)$

* $O(1)$ v průměru pro n prvků, nejhorší případ je $O(\log n)$

Prioritní fronta

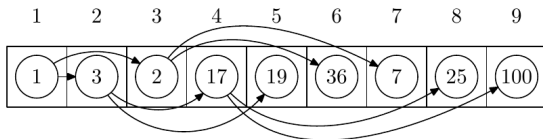
Halda

Heap sort

Binární halda / hromada

(binary heap)

- ▶ Prvky jsou uloženy v poli a_1, a_2, \dots, a_n
- ▶ Potomky a_i jsou a_{2i}, a_{2i+1} (pokud $< n$)
- ▶ **Vlastnost haldy (heap property)**: uzel není větší, než jeho potomci, $a_i \leq a_{2i}, a_i \leq a_{2i+1}$
- ▶ Halda je binární strom, vrstvy až na poslední jsou plně obsazené

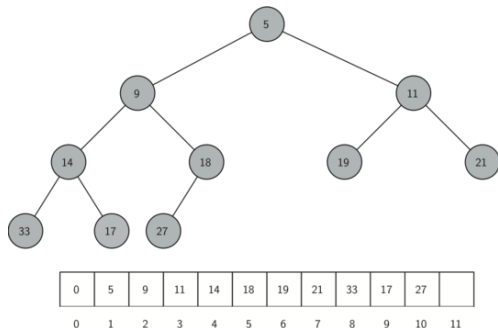


Images courtesy of Jakub Černý, Brad Miller and David Ranum.

Binární halda / hromada

(binary heap)

- ▶ Prvky jsou uloženy v poli a_1, a_2, \dots, a_n
- ▶ Potomky a_i jsou a_{2i}, a_{2i+1} (pokud $< n$)
- ▶ **Vlastnost haldy (heap property):** uzel není větší, než jeho potomci, $a_i \leq a_{2i}, a_i \leq a_{2i+1}$
- ▶ Halda je binární strom, vrstvy až na poslední jsou plně obsazené



Images courtesy of Jakub Černý, Brad Miller and David Ranum.

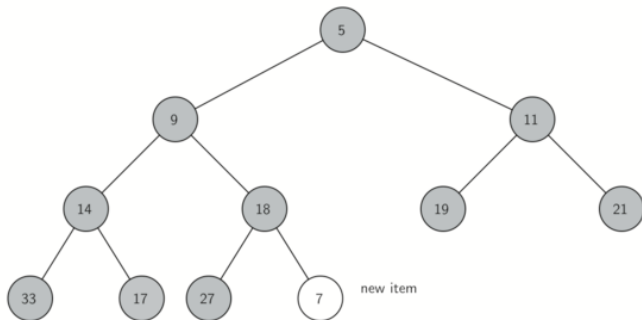
Srovnání haldy a stromu

binární vyhledávací strom (BST)

	BST	halda
uspořádání	úplné	jen minimum
režie	významná	minimální
vyvažování	nutné	automatické

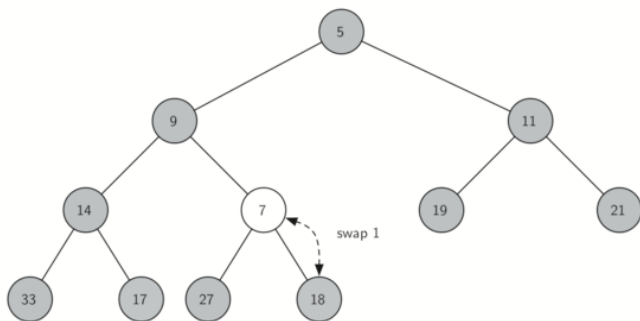
- ▶ halda je významně rychlejší, potřebuje méně paměti
- ▶ strom je univerzálnější, podporuje více operací

Vkládání do haldy



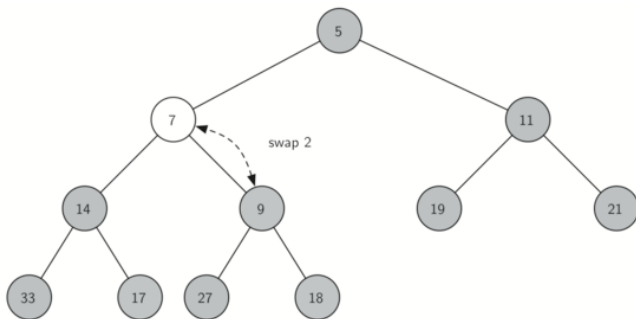
Nový prvek přidáme na konec a probubláme kam patří.

Vkládání do haldy



Nový prvek přidáme na konec a probubláme kam patří.

Vkládání do haldy



Nový prvek přidáme na konec a probubláme kam patří.

Vkládání — implementace

```
class MinHeap:
    def __init__(self):
        self.heap = [] # indexujeme od nuly

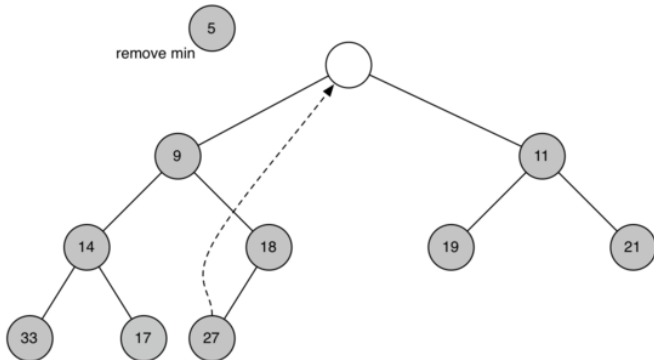
    def bubble_up(self,i): # probublá 'i' ke kořenu
        while i>0:
            j=(i-1)//2 # index rodiče
            if self.heap[i] >= self.heap[j]:
                break
            self.heap[j],self.heap[i]=self.heap[i],self.heap[j]
            i = j

    def insert(self,k):
        self.heap+=[k]
        self.bubble_up(len(self.heap)-1)
```

Indexování od nuly — rodič a_i je $a_{\lfloor(i-1)/2\rfloor}$, potomci jsou a_{2i+1} , a_{2i+2} .

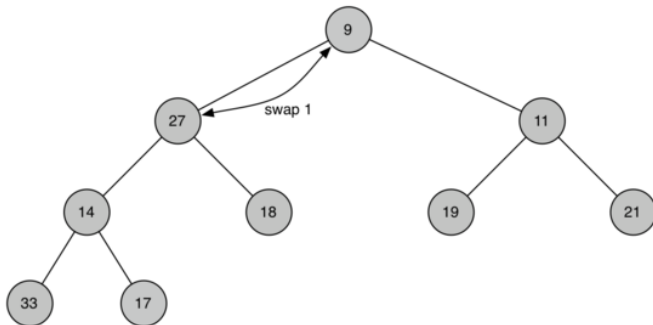
Odebrání nejmenšího prvku

- ▶ Minimum je v kořeni stromu.
- ▶ Halda (pole) se zkrátí o jeden prvek.
- ▶ Tento prvek přesuneme do kořene a 'proubláme' dolů:
 - ▶ Rekurzivně měníme uzel s menším synem, dokud je porušená vlastnost haldy.



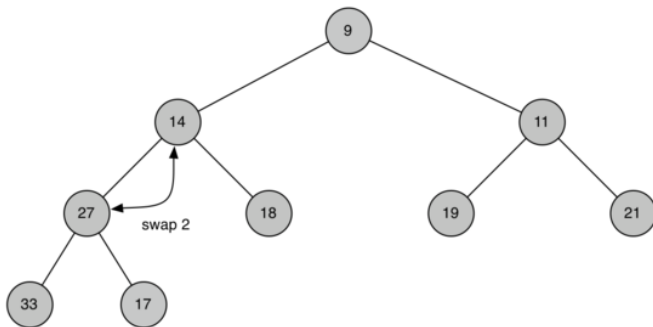
Odebrání nejmenšího prvku

- ▶ Minimum je v kořeni stromu.
- ▶ Halda (pole) se zkrátí o jeden prvek.
- ▶ Tento prvek přesuneme do kořene a 'probubláme' dolů:
 - ▶ Rekurzivně měníme uzel s menším synem, dokud je porušená vlastnost haldy.



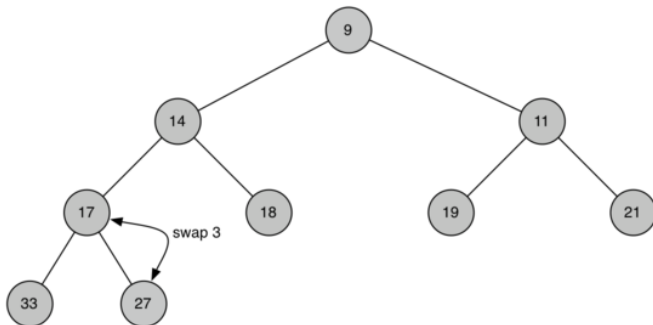
Odebrání nejmenšího prvku

- ▶ Minimum je v kořeni stromu.
- ▶ Halda (pole) se zkrátí o jeden prvek.
- ▶ Tento prvek přesuneme do kořene a 'proubláme' dolů:
 - ▶ Rekurzivně měníme uzel s menším synem, dokud je porušená vlastnost haldy.



Odebrání nejmenšího prvku

- ▶ Minimum je v kořeni stromu.
- ▶ Halda (pole) se zkrátí o jeden prvek.
- ▶ Tento prvek přesuneme do kořene a 'proubláme' dolů:
 - ▶ Rekurzivně měníme uzel s menším synem, dokud je porušená vlastnost haldy.



Odebírání nejmenšího prvku — implementace

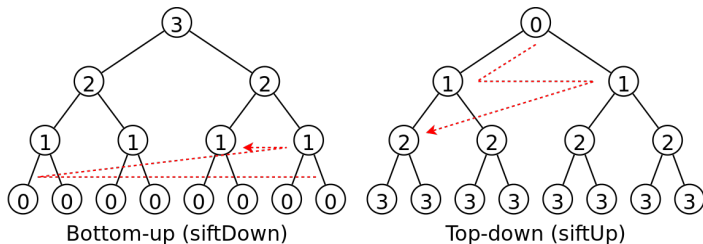
```
def bubble_down(self,i):
    n=self.size()
    while 2*i+1 < n:
        j=2*i+1 # zjistí index menšího syna
        if j+1 < n and self.heap[j] > self.heap[j+1]:
            j+=1
        if self.heap[i]>self.heap[j]:
            self.heap[i],self.heap[j]=self.heap[j],self.heap[i]
        i=j

def pop(self):
    """ odeber a vrať nejmenší prvek """
    element=self.heap[0]
    self.heap[0]=self.heap[-1]
    self.heap.pop() # smaž poslední prvek
    self.bubble_down(0)
    return element
```

Vytvoření haldy

Jak vytvořit haldu z pole

- ▶ Opakované volání insert — složitost $O(n \log n)$
- ▶ Pole prohlásíme za haldu a opakovaně voláme bubble_down — složitost $O(n)$



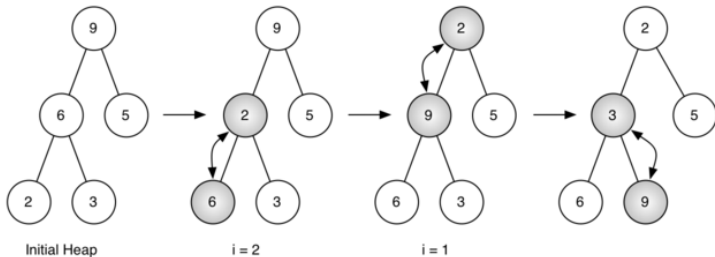
The number in the circle indicates the maximum times of swapping required when adding the node to the heap.

Image courtesy of Swfung8 (Wikipedia).

Vytvoření haldy

Jak vytvořit haldu z pole

- ▶ Opakované volání `insert` — složitost $O(n \log n)$
- ▶ Pole prohlásíme za haldu a opakovaně voláme `bubble_down` — složitost $O(n)$



Vytvoření haldy — implementace

Cyklus přes prvky, které mají potomky, od posledního k prvnímu:

```
def buildHeap(a, type=MinHeap):  
    """ vytvoří haldu z obsahu pole, pole je přepsáno """  
    h=type() # vytvoření MinHeap nebo MaxHeap  
    h.heap=a # kopie  
    for i in range((len(a)-1)//2, -1, -1):  
        h.bubble_down(i)  
    return h
```

heap.py obsahuje MinHeap a MaxHeap

Halda — příklad

```
import heap

h=heap.MinHeap()
for x in [9, 5, 0, 6, 2, 3]:
    h.insert(x) # vkládáme po jednom
print(h.pop()) # nejmenší prvek

0

h.insert(4)
while not h.is_empty():
    print(h.pop(),end=" ", " ")

2, 3, 4, 5, 6, 9,
```


Halda — příklad (2)

```
h=heap.buildHeap([9, 5, 0, 6, 2, 3])  
while not h.is_empty():  
    print(h.pop(),end=" ",)
```

0, 2, 3, 5, 6, 9,

```
h=heap.buildHeap([9, 5, 0, 6, 2, 3],type=heap.MaxHeap)  
while not h.is_empty():  
    print(h.pop(),end=" ",)
```

9, 6, 5, 3, 2, 0,

Prioritní fronta

Halda

Heap sort

Heap sort

třídění haldou

Postup

- ▶ Vytvoříme z pole haldou
- ▶ Opakovaně odebíráme nejmenší prvek → setříděná posloupnost.

Postup

- ▶ Vytvoříme z pole haldu
- ▶ Opakovaně odebíráme nejmenší prvek → setříděná posloupnost.
- ▶ Halda i setříděná posloupnost budou sdílet stejné pole.
- ▶ Použijeme sestupně setříděnou haldu (MaxHeap).

Heap sort

třídění haldou

Postup

- ▶ Vytvoříme z pole haldu
- ▶ Opakovaně odebíráme nejmenší prvek → setříděná posloupnost.
- ▶ Halda i setříděná posloupnost budou sdílet stejné pole.
- ▶ Použijeme sestupně setříděnou haldu (MaxHeap).

Vlastnosti

- ▶ Složitost v nejhorším i průměrném případě $O(n \log n)$.
- ▶ Nepotřebuje pomocnou paměť.

Heap sort — implementace

```
def bubble_down(a,i,n):
    while 2*i+1 < n:
        j=2*i+1 # zjistí index menšího syna
        if j+1 < n and a[j] < a[j+1]:
            j+=1
        if a[i]<a[j]:
            a[i],a[j]=a[j],a[i]
        i=j

def heap_sort(a):
    n=len(a)
    for i in range((n-1)//2,-1,-1):
        bubble_down(a,i,n)
    for i in range(n-1,0,-1): # od n-1 do 1
        a[0],a[i]=a[i],a[0]
        bubble_down(a,0,i)
    return a
```

Heap sort — příklad

```
from heapq import heap_sort
```

```
a=[70, 93, 18, 37, 64, 79, 4, 28, 48, 65]
```

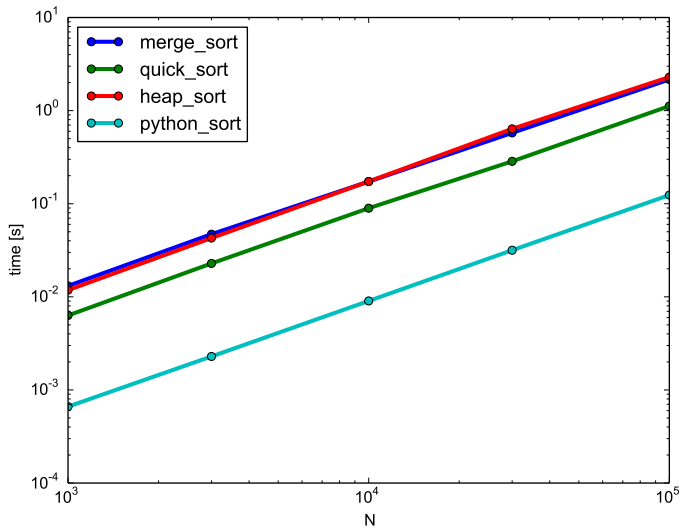
```
heap_sort(a)
```

```
print(a)
```

```
[4, 18, 28, 37, 48, 64, 65, 70, 79, 93]
```

Heap sort — porovnání rychlosti

Empirická složitost



Další haldy

- ▶ *k*-regulární halda — každý uzel *k* potomků.
- ▶ *Fibonacciho halda* (Fredman, Tarjan 1984) — s rostoucí hloubkou klesá maximální počet potomků. Umí vkládat a aktualizovat v čase $O(1)$.
- ▶ *příhrádková halda* (*bucket/radix heap*) — pro malou abecedu o velikosti *k*. Složitost nalezení minima $O(k)$.

Halda v Pythonu

Modul `heapq`. Operace `heapify` (vytvoření), `heappush` (přidání) a `heappop` (odebrání).

```
import heapq
```

```
h=[]
```

```
for x in [9, 5, 0, 6, 2, 3]:
```

```
    heapq.heappush(h,x) # ukládáme po jednom
```

```
print([ heapq.heappop(h) for i in range(len(h))])
```

```
[0, 2, 3, 5, 6, 9]
```

Heapsort a heapq

```
def heap_sort(a):  
    h=a.copy()  
    heapq.heapify(h)  
    for i in range(len(a)):  
        a[i]=heapq.heappop(h)
```

Heapsort a heapq

```
def heap_sort(a):  
    h=a.copy()  
    heapq.heapify(h)  
    for i in range(len(a)):  
        a[i]=heapq.heappop(h)
```

```
x=[75, 61, 5, 66, 35, 55, 20, 48, 41, 85]
```

```
heap_sort(x)
```

```
print(x)
```

```
[5, 20, 35, 41, 48, 55, 61, 66, 75, 85]
```

Heapsort a heapq

```
def heap_sort(a):  
    h=a.copy()  
    heapq.heapify(h)  
    for i in range(len(a)):  
        a[i]=heapq.heappop(h)
```

```
x=[75, 61, 5, 66, 35, 55, 20, 48, 41, 85]  
heap_sort(x)  
print(x)
```

```
[5, 20, 35, 41, 48, 55, 61, 66, 75, 85]
```

- ▶ Potřebuje pomocné pole

Halda — shrnutí

- ▶ Datová struktura pro rychlé hledání minimálního prvku umožňující přidávání.
- ▶ Používáme pro implementace prioritní fronty.
- ▶ Vlastnost haldy — prvky od kořene k listu neklesají / nerostou.
- ▶ Velmi efektivní implementace pomocí pole. Nepotřebuje odkazy.
- ▶ Haldu lze sestavit v čase $O(n)$, odebrání nejmenšího prvku trvá $O(\log n)$.
- ▶ *Heapsort* — zaručená složitost $O(n \log n)$, nepotřebuje pomocnou paměť. Stejně rychlý jako mergesort.

Náměty na domácí práci

- ▶ Implementujte simulátor tiskové fronty pomocí prioritní fronty.
- ▶ Experimentálně vyhodnoťte rychlost haldy oproti vyhledávacímu stromu (pro hledání minima).
- ▶ Zefektivněte 'proublávání', místo výměn použijte přesuny.
- ▶ Napište 'proublávání' rekurzivně.
- ▶ Naprogramujte smazání libovolného prvku z haldy.
- ▶ Implementujte k regulární haldu. Jaká je časová složitost jednotlivých operací?