

```
270         childpos = rightpos
271         # Move the smaller child up.
272         heap[pos] = heap[childpos]
273         pos = childpos
274         childpos = 2*pos + 1
275     # The leaf at pos is empty now. Put newitem there, and bubble it up
```

# Algoritmy a programování

## Prioritní fronta, halda

```
283     # Follow the path to the root, moving parents down until finding a place
284     # newitem fits.
285     while pos > startpos: Vojtěch Vonásek
286         parentpos = (pos - 1) >> 1
287         parent = heap[parentpos]
288         if parent < newitem: Department of Cybernetics
289             heap[parentpos] = newitem
290             pos = parentpos
291             continue
292         break
293     heap[pos] = newitem
294
295     def _siftup_max(heap, pos):
296         'Maxheap variant of _siftup'
297         endpos = len(heap)
298         startpos = pos
299         newitem = heap[pos]
300         # Bubble up the larger child until hitting a leaf.
301         childpos = 2*pos + 1 # leftmost child position
302         while childpos < endpos:
```

## Prioritní fronta (Priority Queue)



- Abstraktní datová struktura
- Obsahuje dopředu neznámý počet prvků
- Prvky jsou vnitřně organizovány dle jejich velikosti
- Základní operace
  - přidání prvku (`insert`, `append`, `push`)
  - odebrání nejmenší položky (`pop`, `top`, `getBest`)
- Další operace
  - zjištění počtu prvků (`size`, `isEmpty`)
  - čtení od začátku (bez změny položek)
  - změna prvku

## Příklad

```
1 q = PQ()
2 q.insert(10); q.insert(-1); q.insert(6)
3 print(q.pop(), q.pop(), q.pop())
```

-1 6 10

## Varianty

- Min-fronta — `pop()` vrací nejmenší prvek
- Max-fronta — `pop()` vrací největší prvek
- Prvky obsahují klíč a hodnotu, vnitřně jsou prvky řazeny dle klíče
- Zásobník a fronta jsou speciálním případem prioritní fronty
  - prvky mají prioritu dle pořadí jejich vložení

## Aplikace

- Prioritní fronta je základní ADT pro mnoho algoritmů
  - Hledání  $k$  nejmenších (největších) prvků
  - Prioritní rozvrhování a plánování
  - Hledání cest v grafech (např. Dijkstrův algoritmus)
  - Výpočet kostry grafu (Primův algoritmus)
  - Heapsort
  - Huffmanovo kódování
  - Informované prohledávání stavového prostoru — např. best-first search
- Časová složitost  $\mathcal{O}()$  uvedená u některých algoritmů předpokládá použití prioritní fronty

- Prioritní frontu lze vnitřně implementovat různými způsoby
- Například naivní implementace polem  $x$ :
  - Prvky přidáváme funkcí  $x.append()$
  - $pop()$  nejdříve najde index nejmenšího prvku  $i$  a vrátí  $x.pop(i)$
  - $remove(y)$  najde index  $i$  prvku  $y$ , a smaže ho  $x.pop(i)$

Implementace prioritní fronty	$insert()$ vkládání prvku	$pop()$ nalezení maxima	$remove()$ odebrání prvku
Pole	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Binární strom	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Binární halda	$\mathcal{O}(\log n)^1$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

- Binární halda je velmi efektivní způsob implementace prioritní fronty

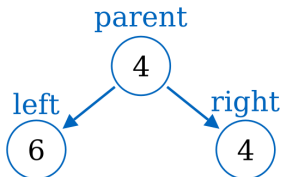
<sup>1</sup>průmerně  $\mathcal{O}(1)$  pro  $n$  prvků, nejhůře  $\mathcal{O}(\log n)$

**Binární halda (binary heap):** binární strom sestavený z prvků

## Min-halda

- Kořen stromu obsahuje nejmenší prvek
- Vlastnost min-haldy haldy: uzel není větší než (oba) jeho potomci

$$x[\text{parent}] \leq \min(x[\text{left}], x[\text{right}])$$



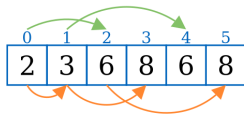
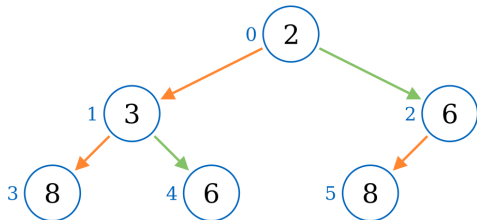
## Max-halda

- Obdobně jako min-heap, ale kořen obsahuje největší prvek a vlastnost haldy je

$$x[\text{parent}] \geq \max(x[\text{left}], x[\text{right}])$$

- Binární haldu lze efektivně implementovat v poli
- Pokud  $P$  je index rodiče, pak index levého ( $L$ ) a pravého ( $R$ ) potomka je:

$$L = 2P + 1 \quad R = 2P + 2$$

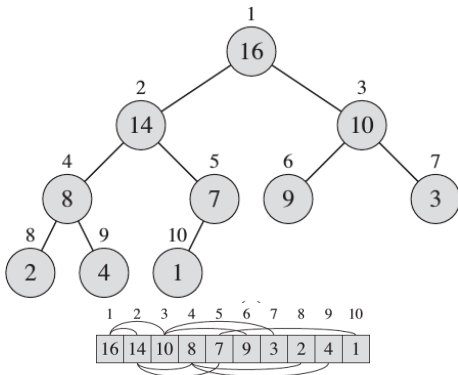


- Výpočet indexu rodiče  $P$  na základě indexů levého a pravého potomka

$$P = (L - 1) // 2 = (R - 1) // 2$$

## Poznámka

- Pozor na indexování
- V Pythonu (C/C++, Java ...) indexujeme pole od **nuly**
- V literatuře se lze setkat s jiným popisem haldy, kde se předpokládá indexování od **jedné**
- V takovém případě je třeba upravit výpočet L,R a P





- Vytvoříme třídu Heap
- Pole pro reprezentaci haldy:  
self.heap
- Hlavní metody:
  - insert, pop
- Pomocné metody:
  - bubbleUp, bubbleDown

```
1 class MinHeap:
2     def __init__(self):
3         self.heap = []
4
5     def insert(self, item):
6         pass #next slides
7
8     def pop(self):
9         pass #next slides
10
11    def bubbleDown(self, idx):
12        pass #next slides
13
14    def bubbleUp(self, idx):
15        pass #next slides
```

## Předpokládané použití

```
1 h = MinHeap()
2 h.insert(1); h.insert(-4); h.insert(10)
3 print(h.top()) #should return -4
4 print(h.top()) #should return 1
5 print(h.top()) #should return 10
```

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5

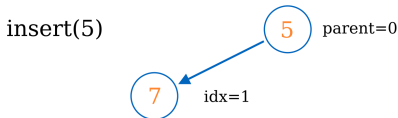
`insert(7)`



- -7: samotný prvek v haldě, vlastost haldy je splněna

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5

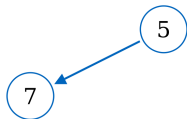


- -2: vlastnost haldy je splněna → není třeba přehazovat prvky

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5

insert(5)

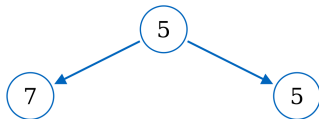


- 1: vlastnost haldy je splněna → není třeba přehazovat prvky

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5

insert(5)

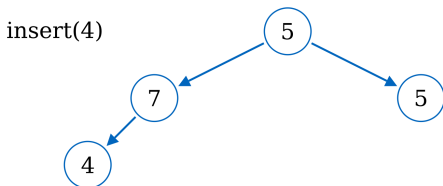


0	1	2
5	7	5

- -5: vlastnost haldy je porušena — rodič je větší než nově vložený prvek

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5

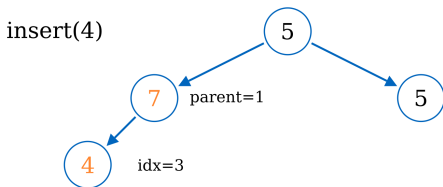


0	1	2	3
5	7	5	4

- -5: vlastnost haldy je porušena — rodič a nově vložený prvek vyměníme

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5

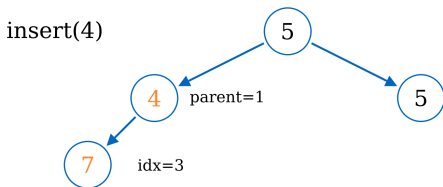


0	1	2	3
5	7	5	4

- -5: vlastnost haldy je porušena — rodič a nově vložený prvek vyměníme

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5



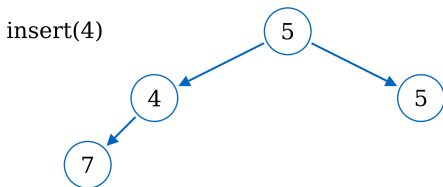
0	1	2	3
5	4	5	7

- -5: po výměně je vlastnost haldy splněna



- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5

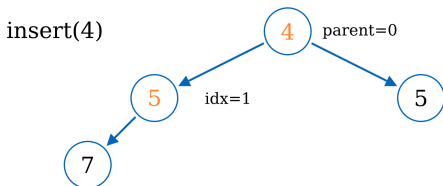


0	1	2	3
5	4	5	7

- -5: po výměně je vlastnost haldy splněna

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

## Příklad vkládání prvků: -7, -2, 1, -5

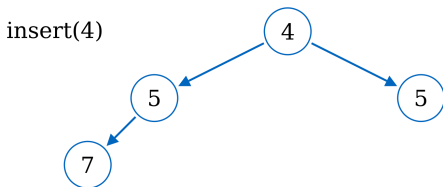


0	1	2	3
4	5	5	7

- -5: po výměně je vlastnost haldy splněna

- Nový prvek vkládáme na konec pole: `self.heap`
- Poté se provede “probublání nahoru”: `self.bubbleUp()`
  - pokud prvek `idx` (a jeho rodič `parent`) porušuje vlastnost haldy, jsou vyměněny
  - probublání pokračuje s rodičem, pak s jeho rodičem ...

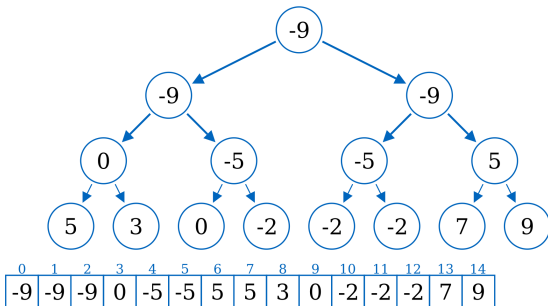
## Příklad vkládání prvků: -7, -2, 1, -5



0	1	2	3
4	5	5	7

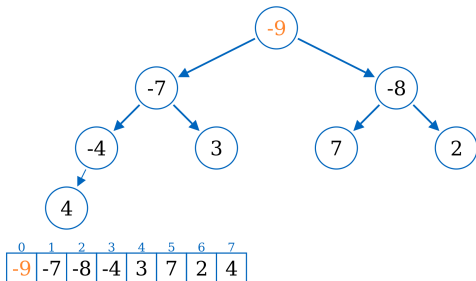
- -5: po výměně je vlastnost haldy splněna

```
1 def add(self, item):
2     self.heap.append(item)
3     self.bubbleUp(len(self.heap)-1)
4
5 def bubbleUp(self, idx):
6     while idx > 0:
7         parent = (idx - 1)//2
8         if self.heap[parent] > self.heap[idx]:
9             self.heap[parent], self.heap[idx] = self.heap[idx],
10            self.heap[parent]
11         idx = parent
```

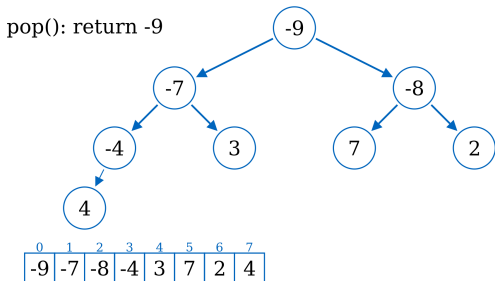


- `pop()`: vrací nejmenší prvek v haldě (min-halda) a **smaže ho**
- Nejmenší prvek na první pozici: `self.heap[0]`
- Poslední prvek se přesune na první prvek a halda se upraví “bubláním dolů” — `bubbleDown()`:
  - pokud je prvek větší než jeden z jeho potomků, tak se s ním vymění
  - výměna probíhá vždy s menším potomkem
  - výměna pokračuje s jeho potomkem atd ...

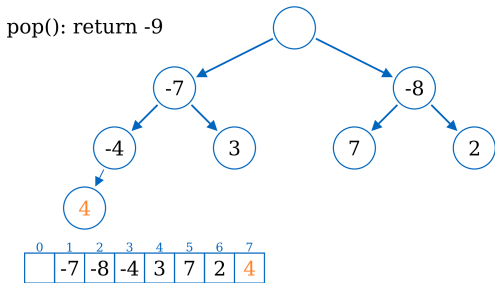
```
1 def pop(self):
2     if len(self.heap) == 0:
3         return None
4     if len(self.heap) == 1:
5         return self.heap.pop()
6
7     v = self.heap[0]
8
9     self.heap[0] = self.heap.pop()
10    self.bubbleDown(0)
11    return v
```



- `pop()` Nejmenší prvek je -9

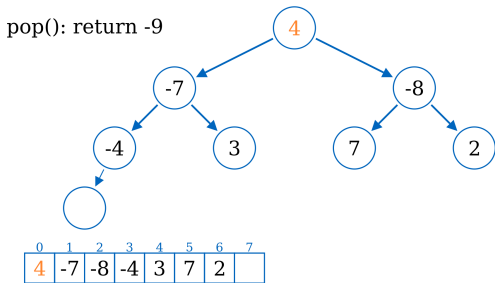


- pop() uložíme si ho pro pozdější použití

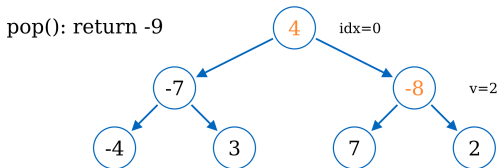


- pop() poslední prvek z pole přesuneme na první prvek



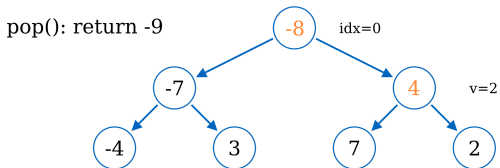


- pop() poslední prvek z pole přesuneme na první prvek



0	1	2	3	4	5	6
4	-7	-8	-4	3	7	2

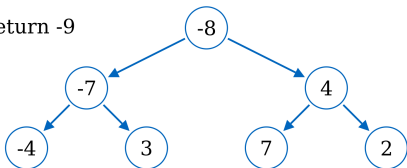
- `pop()` bubbleDown: porovnáme rodiče s nejmenším z potomků



0	1	2	3	4	5	6
-8	-7	4	-4	3	7	2

- `pop()` bubbleDown: pokud je rodič větší, vyměníme

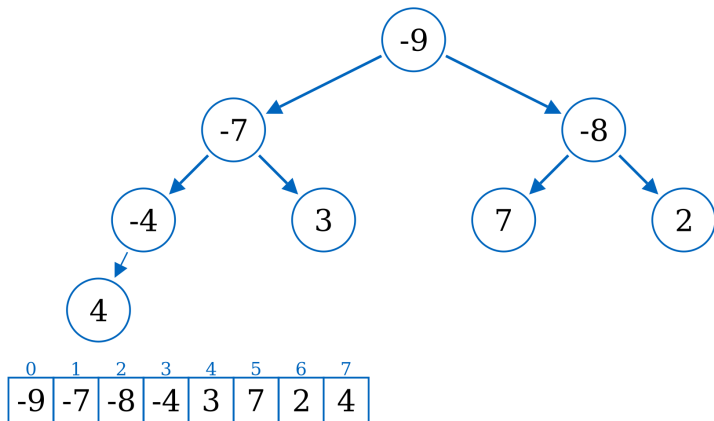
pop(): return -9



0	1	2	3	4	5	6
-8	-7	4	-4	3	7	2

- pop() bubbleDown: nyní je splněna podmínka haldy, bubbleDown končí

```
1     def bubbleDown(self, idx):
2         n = len(self.heap)
3         while idx < n:
4             left = 2*idx+1
5             v = idx
6             if left < n and self.heap[left] < self.heap[idx]:
7                 v = left
8             right = 2*idx + 2
9             if right < n and self.heap[right] < self.heap[v]:
10                v = right
11            if v != idx:
12                self.heap[v], self.heap[idx] = self.heap[idx], self
13                .heap[v]
14                idx = v
15            else:
16                break
```



```
1     def pop(self):
2         if len(self.heap) == 0:
3             return None
4         if len(self.heap) == 1:
5             return self.heap.pop()
6
7         v = self.heap[0]
8
9         self.heap[0] = self.heap.pop()
10        self.bubbleDown(0)
11        return v
```

```
1 from minHeap import MinHeap
2
3 a = [ 10, 1,2, -2, -1, 0, 5, 5 ]
4 print(a)
5 h = MinHeap()
6 for item in a:
7     h.add(item)
8
9 while not h.isEmpty():
10     print(h.pop(), end=" ")
```

```
[10, 1, 2, -2, -1, 0, 5, 5]
-2 -1 0 1 2 5 5 10
```

- Třída `MinHeap` je implementována v souboru `minHeap.py`



- Rozšíření MinHeap na MaxHeap
- Jediná změna je v definici “vlastnosti haldy”, toto se používá v `bubbleUp()` a `bubbleDown()`
- Soubor `maxHeap.py` obsahuje třídu `MaxHeap`

```
1 from maxHeap import MaxHeap
2
3 a = [10, -1, 4, 0, -5, 3, 3]
4 h = MaxHeap()
5 for i in a:
6     h.add(i)
7 print(h.heap)
8 while not h.isEmpty():
9     print(h.pop(), end="␣" )
```

```
[10, 0, 4, -1, -5, 3, 3]
10 4 3 3 0 -1 -5
```

- Vytvoření haldy z pole
- Postupným přidáváním prvků `add()`
  - složitost  $\mathcal{O}(n \log n)$
- “Heapify”
  - pole lze považovat za haldu s tím, že se opakovaně volá `bubbleDown()`
  - složitost  $\mathcal{O}(n)$

- Heapify mění pole tak, aby splnilo vlastnost haldy

```
1 from minHeap import MinHeap
2 from maxHeap import MaxHeap
3
4 def heapify(a, type=MinHeap):
5     """ in-place create of heap from array a """
6     h=type() #either MinHeap of MaxHeap
7     h.heap=a #put all data in a into heap
8     for i in range((len(a)-1)//2, -1, -1):
9         h.bubbleDown(i)
10    return h
```

```
1 from heapify import heapify
2 from minHeap import MinHeap
3
4 a = [10, -1, 2, -4, 5, 6]
5 h = heapify(a, MinHeap)
6 print(h.heap)
7 while not h.isEmpty():
8     print(h.pop(), end = " ")
```

```
[-4, -1, 2, 10, 5, 6]
-4 -1 2 5 6 10
```

- Data jsou vložena do haldy
- Opakovaně odebíráme nejmenší prvek, výsledkem jsou seříděná data
- Složitost  $\mathcal{O}(n \log n)$

```
1 from minHeap import MinHeap
2
3 a = [10, -1, 0, 0, -4, 14, 2]
4
5 h = MinHeap()
6 for i in a:
7     h.add(i)
8
9 sortedA = []
10 while not h.isEmpty():
11     sortedA.append( h.pop() )
12
13 print(a)
14 print(sortedA)
```

```
[10, -1, 0, 0, -4, 14, 2]
[-4, -1, 0, 0, 2, 10, 14]
```

- In-place třídící algoritmus
- Ze vstupního pole vytvoříme MaxHeap
- Největší prvek je na pozici [0], délka pole je  $n$
- Pro všechna  $i = n - 1, n - 2, \dots, 0$ :
  - vyměníme prvek na pozici [0] s prvkem  $i$
  - $n = n - 1$
  - upravíme položky  $0, \dots, n$  tak, aby byla splněna vlastnost haldy, použijeme `bubbleDown()`
- Složitost (nejhorší i průměrná)  $\mathcal{O}(n \log n)$
- Není potřeba pomocná paměť

```
1 # Algoritmus heapsort
2 # Jan Kybic, 2016
3 def bubble_down(a,i,n):
4     while 2*i+1 < n:
5         j=2*i+1
6         if j+1 < n and a[j] < a[j+1]:
7             j+=1
8         if a[i]<a[j]:
9             a[i],a[j]=a[j],a[i]
10            i=j
11
12 def heapSort(a):
13     """ Setrideni pole na miste """
14     n=len(a)
15     for i in range((n-1)//2,-1,-1):
16         bubble_down(a,i,n)
17     for i in range(n-1,0,-1): # od n-1 do 1
18         a[0],a[i]=a[i],a[0]
19         bubble_down(a,0,i)
20     return a
```

```
1 from heapsort import heapSort
2
3 a = [10, -4, 2, 2, -1, 1, -7]
4 print(a)
5 b = heapSort(a)
6 print(b)
```

```
[10, -4, 2, 2, -1, 1, -7]
[-7, -4, -1, 1, 2, 2, 10]
```

- Python (základní knihovna) obsahuje modul `heapq`
- Soubor funkcí pro práci s haldou, která je uložena v poli
- `heappush(h, x)` : přidá prvek `x` do haldy
- `heappop(h)` : odebere prvek z haldy
- `heapify(h)` : vytvoří haldu z pole `h`

```
1 import heapq
2
3 a = [10,4,2,-5,5,11]
4
5 h = [] #our heap
6 for item in a:
7     heapq.heappush(h, item)
8
9 print(h)
10
11 for i in range(len(h)):
12     print(heapq.heappop(h), end=" ")
```

```
[-5, 2, 4, 10, 5, 11]
-5 2 4 5 10 11
```



- Rychlé třídění řetězců nebo celých čísel pevné délky
- Pro každou číslici vytvoříme přihrádku
- Opakujeme od nejméně významného řádu  $i$ 
  - Každý prvek přidáme do přihrádky podle číslice  $i$
  - Obsah přihrádek zřetězíme v pořadí dle hodnoty číslic

```
1 import heapq
2
3 a = [10,4,2,-5,5,11]
4
5 h = [] #our heap
6 for item in a:
7     heapq.heappush(h, item)
8
9 print(h)
10
11 for i in range(len(h)):
12     print(heapq.heappop(h), end="␣")
```

```
[-5, 2, 4, 10, 5, 11]
-5 2 4 5 10 11
```

- Pomocné funkce: určení  $i$ -té číslice, a celkového počtu číslic

```
1 def numDigits(n):  
2     num = 1  
3     while n > 10:  
4         n = n//10  
5         num+=1  
6     return num  
7  
8 def digit(a,n):  
9     return (a//(10**n)) % 10
```

```
[123, 23, 2, 5, 0, 0, 1, 100]  
[0, 0, 1, 2, 5, 23, 100, 123]
```

```
1 def radixSortIntegers(a): #a is array of positive ints
2     maxDigist = numDigits(max(a))
3     p = []
4     for i in range(maxDigist): #sort by i-th digit
5         a = sortByDigit(a, i)
6     return a
7
8 def sortByDigit(a,i):
9     p = [ [] for _ in range(10) ] #buckets for digits 0..9
10    for value in a:
11        c = digit(value,i)
12        p[c].append(value) #put value in c-th bucket
13    result = []
14    for i in range(len(p)):
15        result += p[i] #p[i] is array
16    return result
17
18 a = [123,23,2, 5,0,0,1,100]
19 print(a)
20 b = radixSortIntegers(a)
21 print(b)
```

```
[123, 23, 2, 5, 0, 0, 1, 100]
[0, 0, 1, 2, 5, 23, 100, 123]
```