ALGORITMY A DATOVÉ STRUKTURY

5. Sorting – řazení - třídění

Ing. Igor Kopetschke – TUL, NTI

http://www.nti.tul.cz

TECHNICKÁ UNIVERZITA V LIBERCI

Řadíme nebo třídíme?

- Slovo „sort“ lze přeložit oběma způsoby
- V české literatuře se více používá termín třídění
- Věcně správnější je patrně řazení – chceme data seřadit, ne rozdělit je do tříd.
- V algoritmech využíváme relaci uspořádání – data chceme uspořádat…
- Řešení otázky ponechme ústavu pro jazyk český
- Důležité je vědět, že oba termíny obvykle označují tutéž skupinu algoritmů.

Tři „typy“ algoritmů

- **Školní** – jednoduché algoritmy, obvykle kvadratická složitost. Vhodné pro malé množiny dat a pro pochopení principu řazení dat.
- **Praktické** – algoritmy používané v praxi, složitost lepší než kvadratická, často podpora ve standardní knihovně jazyka.
- **Teoretické** – vědecké práce, které (zatím) do praxe nepronikly.

Typ (velikost) dat

- Podle typu (velikosti) řazených dat dělíme algoritmy na **vnější** a **vnitřní**.
- **Vnitřní** používáme pro data, která lze najednou uchovat v operační paměti.
- **Vnější** v případě rozsáhlých dat načítaných průběžně z disku.

Strukturovaná data

- V řadě případů netřídíme jednoduché datové typy jako int či char, ale strukturované záznamy.
- **Stabilní** algoritmy – vzájemné pořadí údajů se stejným klíčem zůstane zachováno.
- **Nestabilní** alg. – toto pořadí nelze zaručit.
- Z většiny nestabilních algoritmů lze využitím pomocné datové struktury učinit stabilní.

Částečně seřazená data

- V řadě případů jsou data částečně uspořádaná předem.
- **Přirozený** algoritmus – je na takových datech rychlejší.
- **Nepřirozený** algoritmus – je na nich stejně rychlý jako na náhodných datech.

Co budeme řadit?

- Máte 6 nízkých kanastových karet – od 2 po 9.
- Karty leží na stole = **vstup**
- Cílem je seřadit tyto karty podle velikosti.
- Pro držení karet máme k dispozici jednu ruku = **paměť**
- Druhá ruka slouží k manipulaci s kartami = **I/O** kanál

SelectSort

- Velmi jednoduchý algoritmus
- Snadná implementace
- Složitost algoritmu je vždy kvadratická $O(N^2)$
- Vnitřní, nestabilní, nepřirozená

SelectSort - princip

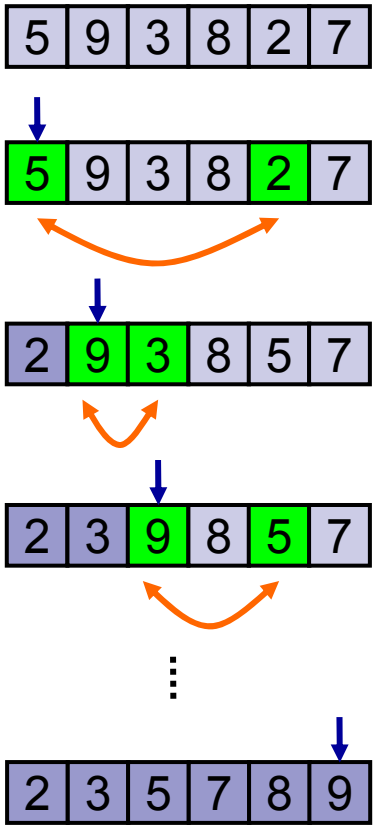
- pracuje na principu nalezení **minimálního** prvku v **nesetříděné** části posloupnosti a jeho zařazení na **konec** již **setříděné** posloupnosti.

1) V posloupnosti najdeme nejmenší prvek a vyměníme ho s prvkem na první pozici
   a. Rozdělení posloupnosti na dvě části.
   b. Setříděná část obsahuje pouze jeden prvek, nesetříděná n-1.

2) V nesetříděné části najdeme nejmenší prvek
   a. Vyměníme ho s prvním prvkem v nesetříděné části
   b. Dojde k zařazení tohoto prvku do setříděné části.

3) Obsahuje-li nesetříděná část více než jeden prvek, pokračujeme bodem 2, jinak je třídění ukončeno.

SelectSort – řazení výběrem



původní neuspořádaná množina

krok 1.

krok 2.

krok 3.

krok 6. výsledek = uspořádaná množina

SelectSort – Python

```
def select_sort (pole):
  for i in range (0,len(pole)):
    min = i
    for j in range (i+1,len(pole)):
      if pole[j] < pole[min]:
        min = j
    pole[i], pole[min] = pole[min], pole[i]
```

InsertSort

- přirozený algoritmus třídění karet na ruce
- jednoduchá implementace
- efektivní na malých množinách
- Složitost – nejhorší $O(N^2)$, nejlepší $O(N)$
- dokáže řadit data tak, jak přicházejí na vstupu – online algoritmus
- Vnitřní i vnější, přirozený, stabilní
- efektivní způsob testování, zda jsou data uspořádaná nebo ne

InsertSort - princip

- Pracuje na principu **vkládání** prvku na jeho **správné** místo v posloupnosti.
- K tomu využívá **pomocný** prvek, zpravidla **nultý** prvek posloupnosti.

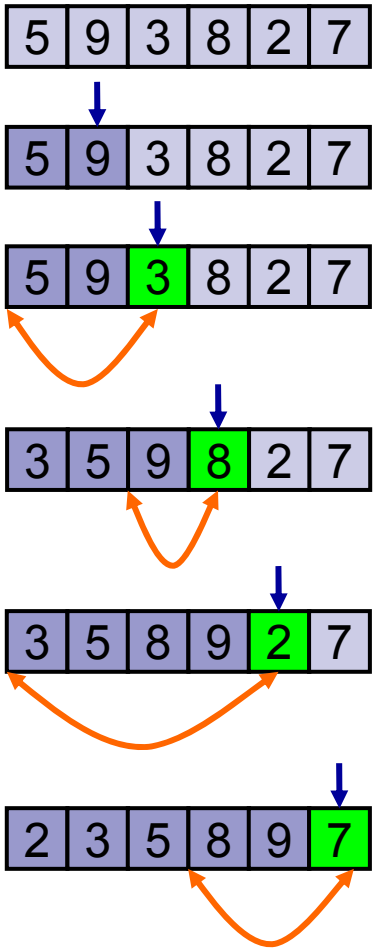
1) První prvek pole ponecháme na svém místě.

2) Vezmeme druhý prvek a porovnáme jej s prvním.
   a. Je-li menší, zařadíme ho na první místo a první prvek posuneme
   b. V opačném případě ponecháme na místě

3) Vezmeme třetí prvek a porovnáme jej s prvními dvěma prvky
   a. Je-li menší než některý z nich, zařadíme jej na odpovídající pozici a následující prvky podle potřeby posuneme
   b. Jinak je ponecháme na původních místech.

4) Obdobně postupujeme i s ostatními prvky v poli.

InsertSort



původní neuspořádaná množina

krok 1.

krok 2.

krok 3.

krok 4.

krok 5.

InsertSort - Python

```
def insert_sort (pole):
  for i in range (1,len(pole)):
    save = pole[i]
    j = i
    while (j>0 and pole[j-1] > save):
      pole[j] = pole[j-1];
      j -= 1
    pole[j] = save
```

BubbleSort – řazení záměnou

- Název od „probublávání“ větších prvků na konec (začátek) tříděné množiny.
- Složitost
  - nejhorší $O(N^2)$ – v krajním případě $(N^2+N)/2$ kroků
  - nejlepší $O(N)$
- Přirozená, stabilní, vnitřní
- I když má stejnou složitost jako InsertSort je pomalejší – má více elementárních operací v datech
- Varianta kdy probubláváme v obou směrech se nazývá Shaker či CoctailSort – složitost opět $O(N^2)$

BubbleSort - princip

- Pracuje na principu systematického porovnávání dvojice sousedních čísel
- Pokud menší číslo následuje po větším - výměna
- Maximální prvek "probublá" na konec.

1) Posloupnost rozdělíme na dvě části, setříděnou a nesetříděnou. Setříděná část je prázdná.

2) Postupně porovnáme všechny sousední prvky v nesetříděné části a pokud nejsou v požadovaném pořadí, prohodíme je.

3) Krok 2 opakujeme tak dlouho, dokud nesetříděná část obsahuje více než jeden prvek. Jinak algoritmus končí.

BubbleSort – řazení záměnou

původní neuspořádaná množina

5	9	3	8	2	7

průběh prvního kroku

5	9	3	8	2	7
5	3	9	8	2	7
5	3	8	9	2	7
5	3	8	2	9	7
5	3	8	2	7	9

průběh druhého kroku

3	5	8	2	7	9
3	5	8	2	7	9
3	5	2	8	7	9
3	5	2	7	8	9

průběh třetího kroku

3	5	2	7	8	9
3	2	5	7	8	9
3	2	5	7	8	9

průběh čtvrtého kroku

2	3	5	7	8	9
2	3	5	7	8	9

pátý a poslední krok

2	3	5	7	8	9

BubbleSort – Python

```
def bubble_sort(pole):
  swap_test = False
  for i in range (0,len(pole)-1):
    for j in range (0,len(pole)-i-1):
      if pole[j] > pole[j+1] :
        pole[j], pole[j+1] = pole[j+1], pole[j]
        swap_test = True
    if swap_test == False:
      break
```

QuickSort

- Autorem je **C. A. R. Hoare** – 1962 (26)
- Algoritmus typu Rozděl a Panuj (Divide and Conquer)
- Klíčovým problémem je volba pivota
  - první (poslední) prvek
  - náhodný prvek
  - medián pole, medián 3-5 prvků
- Výběr pivota určuje složitost algoritmu

QuickSort

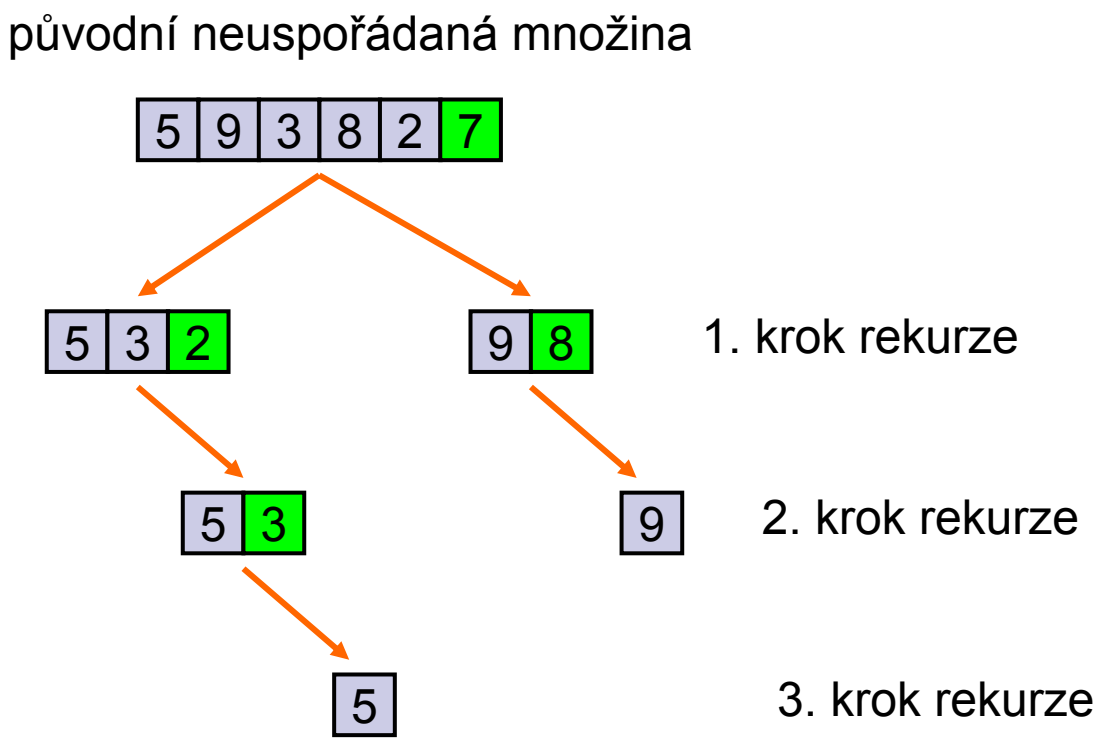
- Nejhorší případ $O(N^2)$, nejlepší a průměrný $O(N \log N)$.
- Logaritmickou složitost nelze zaručit, ale reálné aplikace a testy ukazují, že na (pseudo)náhodných datech je vůbec nejrychlejší ze všech obecných řadicích algoritmů.
- Díky D&C je dobře paralelizovatelný.
- Vnitřní, nestabilní, nepřirozený

QuickSort - princip

- Pracuje na principu rozdělení pole řazených prvků na dvě části a tyto potom seřadit.
- Využívá rekurzi

1) Zvolit dělící prvek – pivota. Tento je umístěn na konečné pozici
2) Projdeme pole zleva dokud nenalezneme větší prvek než dělící prvek
3) Dále ho projdeme zprava, dokud nenalezneme menší prvek než dělící prvek
4) Tyto prvky pak vyměníme.
5) Kroky 2-4 opakujeme až do kompletního setřídění.

QuickSort – rychlé řazení



QuickSort - Python

```
def QuickSort(l):
   if l == []:
       return [] #ukončení rekurze
   left = []
   right = []
   pivot = l.pop()
   middle = [pivot]
   for item in l:
       if item < pivot:
              left.append(item)
       else:
              right.append(item)
   return QuickSort(left) + middle + QuickSort(right)
```

MergeSort

- Autor – John von Neuman (1945)
- D&C algoritmus
- Logaritmická složitost O(N log N) - vždy
- Větší paměťové nároky – obvykle potřebuje odkládací ADT o velikosti N.
- Výhoda – stabilní, paralelizovatelný, vyšší výkon na sekvenčních médiích
- Implicitní řazení v řadě jazyků – např. GNU C a Java.

MergeSort - princip

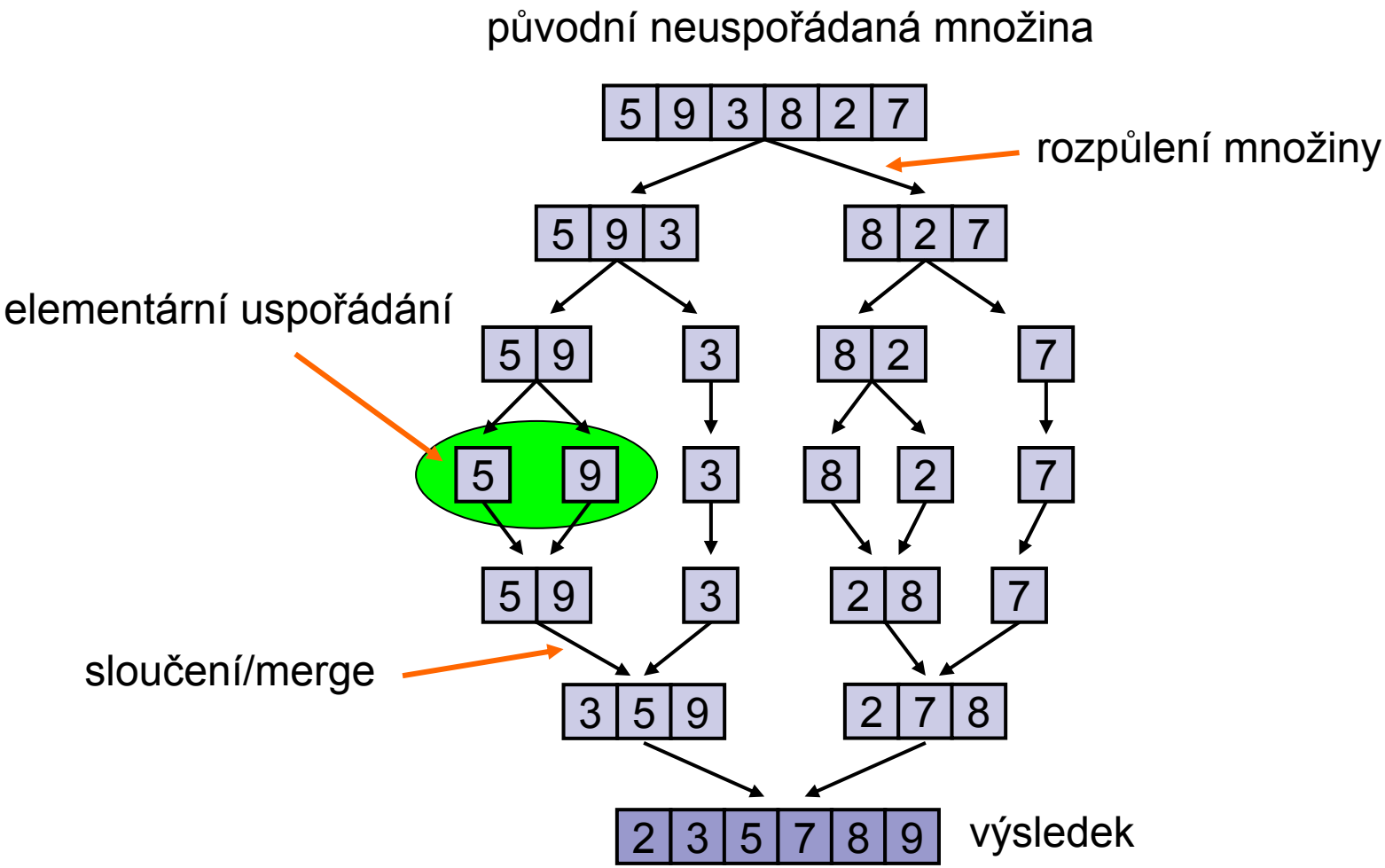
- Je opakem QuickSortu
- Místo dělení posloupnosti slučování podposloupnosti
- Základní myšlenka celého algoritmu
  - Setřídit kratší posloupnost zabere méně kroků.
  - Spojit dohromady dvě setříděné posloupnosti tak, aby výsledek byl setříděný, je snadnější než když jsou posloupnosti nesetříděné.

1) Rozdělí neseřazenou množinu dat na dvě podmnožiny o přibližně stejné velikosti

2) Rekurzivně (MergeSortem) setřídíme každou vzniklou podmnožinu

3) Sloučíme obě podmnožiny

MergeSort – řazení slučováním



MergeSort – Python

```
def mergesort(list):
    if len(list) < 2:
        return list
    else:
        middle = len(list) / 2
        left = mergesort(list[:middle])
        right = mergesort(list[middle:])
        return merge(left, right)
```

list[:middle] – slice index, vybere z listu prvky s indexem 0 až middle, včetně.

MergeSort – Python, fce merge

```
def merge(left, right):
    result = []
    i ,j = 0, 0
    while(i < len(left) and j < len(right)):
        if (left[i] <= right[j]):
            result.append(left[i])
            i = i + 1
        else:
            result.append(right[j])
            j = j + 1
    result += left[i:]
    result += right[j:]
    return result
```

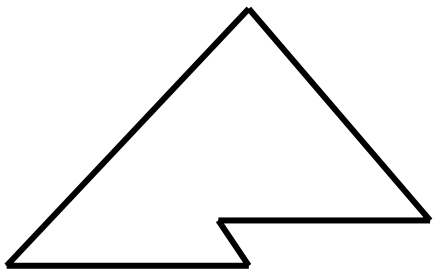
HeapSort – řazení haldou

- Co je to halda?
- Strom (ADT), který splňuje vlastnost haldy.
- Vlastnost haldy: pokud B je potomek A, pak platí že h(A) >= h(B). Funkce h(X) udává hodnotu klíče uzlu X.
- Typ stromu určuje zároveň typ haldy.
- Pro potřeby řazení využijeme **binární** haldu.

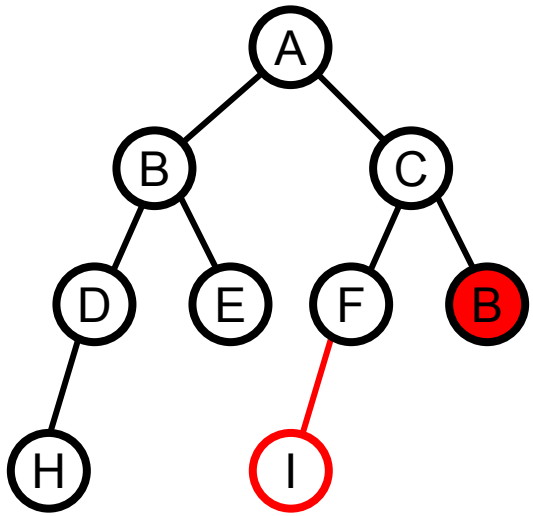
HeapSort – binární halda

- Binární halda je binární strom, pro který platí:
  - Vlastnost tvaru: strom je buď vyvážený, nebo se poslední úroveň stromu zaplňuje zleva doprava.
  - Vlastnost haldy

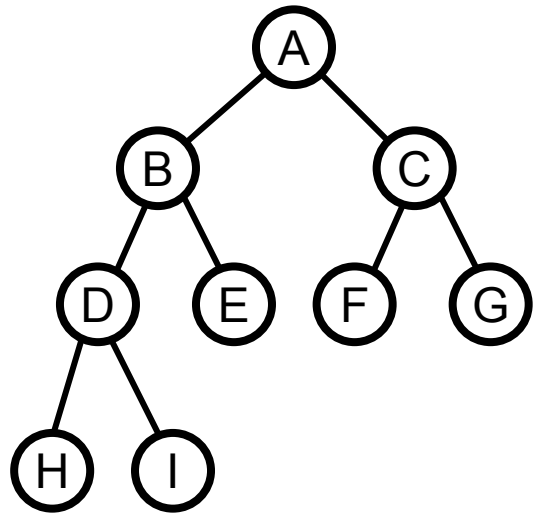
schema binární haldy



není binární halda



minimální binární halda



HeapSort – řazení haldou

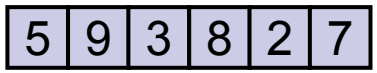
- Logaritmická složitost O(N log N) – vždy
- Pokud pro haldu využijeme vstupní pole, nemá HS žádné paměťové nároky navíc.
- Horší možnosti paralelizace
- V průměru pomalejší než QuickSort – ale vhodnější pro rozsáhlé kolekce neznámých dat.

HeapSort - princip

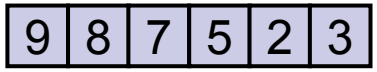
- Haldy se pak využívá k tomu, aby se do ní uspořádaly prvky vstupní posloupnosti.
- Poté se z haldy vybírají prvky do výstupní posloupností, počínaje minimálním (tedy nejvyšším) prvkem.

1) Po načtení procházet polem od jeho začátku metodou nahoru()
2) Vlevo od okamžité pozice budeme vytvářet haldu (fáze vytvoření haldy). Prvky řazeného pole v uvedené implementaci jsou uloženy v **pole[1]** až **pole[pocet]**.
3) Následuje fáze řazení, kdy vyměníme kořen haldy (první prvek pole s posledním prvkem haldy a v poli obnovíme metodou dolu() haldu zmenšenou o jeden prvek.
4) Krok 2 opakujeme tak dlouho, dokud není pole seřazeno.

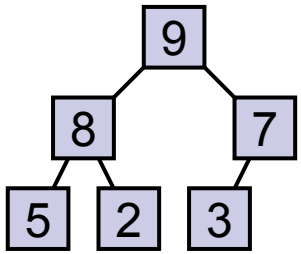
HeapSort



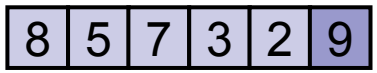
původní množina



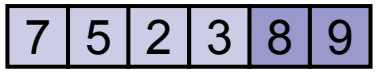
první halda, uložená do pole

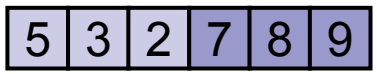


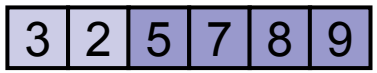
první (max)halda zobrazená jako strom

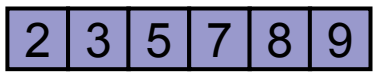


kořen haldy na konec pole
obnovit haldu v n-1 prvcích









výsledek

HeapSort - Python

```
def heapsort(pole):
    first = 0;
    last = len(pole)-1;
    create_heap(pole,first,last)
    for i in range(last,first,-1):
        print pole
        pole[i], pole[first] = pole[first], pole[i]
        establish_heap_property(pole,first,i-1)

def create_heap(pole,first,last):
    i = last / 2;
    while(i >= first):
        establish_heap_property(pole,i,last)
        i -= 1
```

HeapSort - Python

```
def establish_heap_property(pole,first,last):
    while 2 * first + 1 <= last:
        k = 2 * first + 1;
        if k < last and pole[k] < pole[k+1] :
            k += 1
        if pole[first] >= pole[k]:
            break
        pole[first], pole[k] = pole[k], pole[first]
        first = k
```

Princip fce je podobný jako BubbleSort

- maximální prvky přesouváme na začátek pole
- neprocházíme již setříděné větve stromu

Obecné vlastnosti řadících alg.

- Představili jsme si šest řadících algoritmů.
- Krom složitosti a paměťové náročnosti, jsou důležité i další vlastnosti, již výše uvedené
- Tyto vlastnosti – pro zopakování:
  - typu řazených dat
  - chování na strukturovaných datech
  - chování na částečně seřazených datech

Jaký algoritmus je tedy nejlepší?

- Neexistuje nejlepší univerzální řešení
- Vhodný algoritmus musíme vždy vybrat s využitím:
  - toho co víme o řazených datech
  - toho co víme o řadících algoritmech.

Závěrečné porovnání

Název		Časová složitost					
Anglicky	**Česky**	**Nejlepší**	**Průměr**	**Nejhorší**	**Dodatečná paměť**	**Stabilní**	**Přirozený**
SelectSort	Řazení výběrem	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	zprav. ne	ne
InsertSort	Řazení vkládáním	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano
BubbleSort	Bublinkové řazení	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano
QuickSort	Rychlé řazení	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	ne	ne
MergeSort	Řazení slučováním	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	ano	ano
HeapSort	Řazení haldou	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	ne	ne

.. A to je pro dnešek vše

DĚKUJI ZA POZORNOST