Třídící algoritmy.

Insertion Sort. Bubble Sort. Select Sort. Shell Sort. Quick Sort. Merge Sort. Heap Sort.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Třídění:

Uspořádávní vzestupné/sestupné prvků posloupnosti X, $X = \{x_1, ..., x_n\}$:

- *Vzestupné setřídění:*
  Pro $\forall x_i \in X \Rightarrow x_i \leq x_{i+1}, i \in \langle 0, n-2 \rangle$.
- *Sestupné setřídění:*
  Pro $\forall x_i \in X \Rightarrow x_i \geq x_{i+1}, i \in \langle 0, n-2 \rangle$.

Nejčastěji posloupnost číselná, prvky stejného datového typu.

Záměna vzestupného třídění za sestupné:

Prohození znaménka $<$ za $>$ v třídícím algoritmu.

Důvod třídění:

Snaha o uspořádání dat a zefektivnění operací s nimi.

Nižší časová/paměťová složitost (např. prohledávání).

Stabilní třídění:

U prvků se stejnou hodnotou zachováno pořadí v setříděné posloupnosti.

2. Dělení třídících algoritmů:

Dvě skupiny třídících algoritmů:

- *Vnitřní třídící algoritmy*
  Prvky tříděné posloupnosti nacházejí uvnitř paměti počítače.
  K prvkům posloupnosti lze přistupovat v libovolném pořadí.
  Počet prvků n předem znám.
- *Vnější třídící algoritmy*
  Data uložena na nějakém periferním zařízení.
  Počet prvků posloupnosti není znám předem.
  K prvkům lze přistupovat pouze sekvenčně.
  Rozsáhlá data, která se nevejdou do paměti počítače.

3. Efektivita třídících algoritmů

Kritéria efektivity třídících algoritmů:

- počet přesunů prvků nutný k setřídění posloupnosti,
- počet porovnání prvků nutný k setřídění posloupnosti.

Náročnost kroků:

Oba kroky mají různou náročnost.

Přesun prvku je výpočetně *náročnější* než jeho porovnání s jiným prvkem

3 operace vs 1 operace.

Minimalizace množství přesunů při třídění.

Efektivnější provádět přesuny prvků na větší vzdálenosti.

Jeden posun o n prvků výhodnější než n posunů o 1 prvek.

Časová složitost třídících algoritmů:

Nejpomalejší algoritmy $O(N^2)$, nejrychlejší $O(N \log N)$.

Nižší složitosti pro *obecná data* při třídění nelze dosáhnout.

Algoritmy neprovádějící porovnávání: Radix Sort, Counting Sort.

Složitost $O(N)$.

Nelze použít pro obecná data (celočíselná).

4. Přehled třídících algoritmů

Existuje velmi mnoho třídících algoritmů.

V praxi se používá pouze malé procento z nich, ostatní představují spíše akademický problém.

Vybrané "elementární" třídící algoritmy (v praxi nepoužívané):

- Třídění přímým vkládáním (Insertion Sort).
- Třídění binárním vkládáním (Insertion Binary Sort).
- Bublinkové třídění (Bubble Sort).
- Třídění přímým výběrem (Select Sort).
- Třídění se zmenšujícím se krokem (Shell Sort).

Vybrané "profesionální" třídící algoritmy:

- Quick Sort.
- Třídění slučováním (Merge Sort).
- Třídění haldou (Heap Sort).

5. Třídění přímým vkládáním:

Insertion Sort.

Vhodný pro částečně setříděné posloupnosti.

Rychlejší než jiné algoritmy s kvadratickou složitostí (Bubble/Select Sort).

Složitost $O(N^2)$, počet přesunů $C_{max} = 0.5(n^2 + n - 2)$, $C_{min} = 0.25(n^2 + 9n - 10)$.

Inspirován chováním hráče karet při jejich třídění.

- Hráč si vezme z balíčku karty a na základě jejich velikosti a barvy je zařazuje mezi ostatní karty.
- Karty má rozděleny na dvě podmnožiny: karty k setřídění a karty již setříděné.
- Z nesetříděné podmnožiny vezme první kartu a vloží ji na správné místo v setříděné podmnožině.
- Postup opakuje, dokud má k dispozici nějakou nesetříděnou kartu.

Základem dalších algoritmů: Insert Binary Sort či Shell Sort.

Součástí Tim Sortu (Python): kombinace Insertion a Merge Sortu.

6. Princip třídění přímým vkládáním:

Dvě podmnožiny: podmnožina setříděných prvků S a podmnožina nesetříděných prvků N.

Start: $\dim(S) = 0$, $\dim(N) = n$.

- Prvek x_0 v nesetříděné podmnožině ponecháme na stejné pozici
- Prvek x_1 porovnáme s prvkem x_0. Je -li $x_1 > x_0$, ponecháme prvky beze změny. V opačném případě zařadíme x_1 na první pozici a odsuneme x_0 o jednu pozici vpravo.
- Prvek x_2 porovnáme s prvky x_0, x_1. Zařadíme prvek na správnou pozici, prvek/prvky větší než x_2 odsuneme směrem doprava.
- Postup opakujeme, dokud nesetříděná část obsahuje alespoň jeden prvek.

End: $\dim(S) = n$, $\dim(N) = 0$.

6. Porovnávání prvků

Již setříděná posloupnost tvořena prvky $\{x_0, ..., x_{i-1}\}$,
Vkládaný prvek označen jako x_i.
Zatím nesetříděná posloupnost tvořena prvky $\{x_i, ..., x_{n-1}\}$.

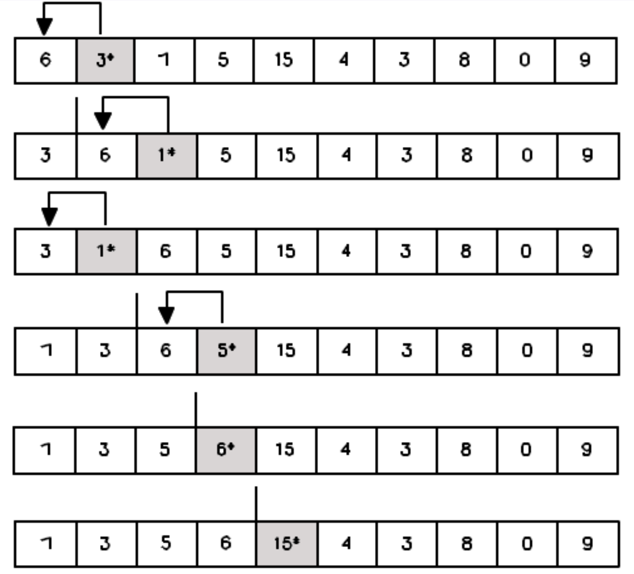
Prvek x_{i-1} představuje *zarážku* dělící posloupnost na dvě podmnožiny.

Princip porovnávání:

Prvek x_i porovnáme s posledním setříděným prvkem x_{i-1}.

- Pokud $x_i \geq x_{i-1}$, prvek x_i je na správné pozici. Inkrementujeme $i = i + 1$, opakujeme postup s následující dvojicí prvků.
- Pokud $x_i < x_{i-1}$, prohodíme prvky x_i a x_{i-1}.
- Prohozený prvek x_{i-1} porovnáme s předchozím prvkem x_{i-2}.
- Pokud není splněna podmínka $x_{i-1} \geq x_{i-2}$, prohodíme prvky x_{i-1} a x_{i-2}
- Pokračujeme analogickým způsobem, dokud vkládaný prvek není na správné pozici $\Rightarrow$ konec.

7. Znázornění algoritmu Insertion Sort



8. Zdrojový kód Insertion Sort:

Realizován dvojicí cyklů.

Vnější probíhá nad všemi prvky.

Vnitřní, dokud není prvek na správném místě.

```
def insertionSort(x):
    for i in range (0, len(x)):
        j = i                               #Zapamatuj index zpracovavane
        p = x[i]
        while (j > 0 and x[j-1] > p):       #Porovnavani se seteridenou c
            temp = x[j]                     #Prohozeni za pouziti
            x[j] = x[j-1]                   #pomocne promenne temp
            x[j-1] = temp
            j = j - 1                       #Dekrementace indexu
.
x = [3, 0, 7, 9]
insertionSort(x)
print(x)
```

9. Modifikace algoritmu přímého vkládání

Nepoužíváme *zarážku*, v každém porovnávání neprohazujeme testovaný prvek s prvkem předchozím.

Prvky pole nacházející se vlevo od testovaného prvku větší než tento prvek posunujeme v každém kroku o 1 pozici vpravo.

Do vzniklé mezery zapíšeme testovaný prvek.

```
def insertionSort(x):
    for i in range (0, len(x)):
        j = i                             #Zapamatuj index zpracovavane
        p = x[i]
        while (j > 0 and x[j-1] > p):     #Porovnavani se seteridenou c
            x[j] = x[j-1]                 #Posun prvku o 1 pozici vprav
            j = j - 1                     #Dekrementace indexu
        x[j] = p;
```

10. Třídění binárním vkládáním

=Insert Binary Sort.

Vylepšené třídění vkládáním, snaha urychlit nalezení místa, kam prvek patří.

Pro urychlení používáme binární vyhledávání.

Výrazné navýšení rychlosti algoritmu: $O(log_2 N) \Rightarrow O(N)$.

Princip algoritmu:

Binární vyhledávání probíhá v setříděné části pole v intervalu $< 0, i - 1 >$.

V indexu l Binary Search bude uložena správná pozice prvku $x[i]$.

Následně odsuneme všechny prvky $< l, i - 1 >$ o jednu pozici vpravo.

Na pozici l přesuneme kopírovaný prvek.

Rekurzivní i nerekurzivní implementace.

11. Zdrojový kód binárního vkládání

```
def insertionBinarySort(x):
   for i in range (0, len(x)):
       j = i
       p = x[i]
       l = 0; r = i - 1            #Levy a pravy index
       while l <= r:               #Binary Search
           k = int((l + r) / 2)    #Index prostredniho prvku
           if (p > x[k]):          #Hledame v leve casti
               l = k + 1
           else:                   #Hledame v prave casti
               r = k - 1
       while j > l:                #Index l = hledana pozice
           x[j] = x[j - 1]         #Posun prvku vpravo od l o 1 pozici
           j = j - 1               #Dekrementace indexu
       x[l] = p                    #Zarazeni prvku na spravne misto
```

12. Bublinkové třídění

Bubble Sort.

Nejpomalejší ze všech uvedených.

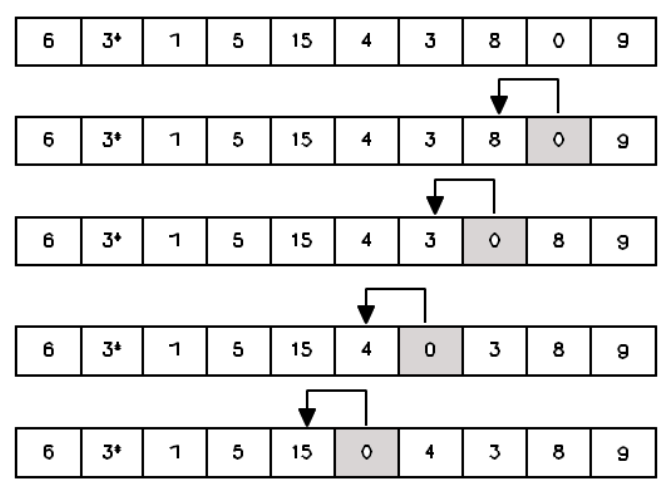
Nevhodné pro částečně setříděné posloupnosti, běží naprázdno (možné odstranit).

Složitost $O(N^2)$, počet přesunů $C_{max} = 0.75(n^2 - n)$.

Princip algoritmu:

- Odvozen od chování vzduchových bublinek limonádě, které postupně stoupají vzhůru k hladině.
- Porovnávány dva sousední prvky x_i a x_{i-1}. Pokud $x_i < x_{i-1}$, prohodíme oba prvky. V opačném případě je ponecháme ve stejném pořadí a porovnáme následující dvojici prvků.
- Porovnávání začíná od konce pole. Prvky s menší hodnotou se postupně přesouvají zprava doleva jako bublinky v limonádě.
- Po prvním průchodu prvek x_0 představuje minimum, v dalším kroku již nemusíme procházet celé pole, ukončíme ho na prvku x_1.

13. Znázornění algoritmu



14. Zdrojový kód

Realizován dvojicí vnořených cyklů.

Vnější probíhá nad všemi prvky, vnitřní o 1 méně.

Varianta s probubláváním nejmenších prvků na počátek.

```
def bubbleSort(x):
    for i in range (0, len(x)):
        for j in range (len(x)-1, i, -1): #Prohledavani od konc
            if x[j] < x[j-1]:             #Probublavani na zaca
                temp = x[j];              #Prohozeni za pouziti
                x[j] = x[j-1];            #pomocne promenne tem
                x[j-1] = temp;
```

15. Zhodnocení algoritmu

Nesymetrická rychlost bublání prvků:

Prvky malých hodnot stoupají rychleji než prvky velkých hodnot klesají.

Po vykonání vnitřního cyklu se minimum nachází na 1. pozici.

Maximum se posune pouze o jednu pozici.

Nevýhoda třídění bubláním:

Značná závislost na vstupních datech, pokud většina prvků setříděna.

Algoritmus běží "naprázdno", neprovádí žádné prohazování.

Proběhnou oba cykly včetně testovací podmínky.

Časová složitost:

Časová složitost třídění bubláním je $O(N^2)$.

Nelze použít pro rozsáhlá data.

16. Modifikace bublinkového třídění

Odstranění prázdných kroků.

Pomocná proměnná typu bool `sort`.

Pokud hodnota True, posloupnost je setříděna.

```
def bubbleSort2(x):
    sort = False
    i = 0
    while not(sort):
        sort = True;                  #Rekneme, ze posloupnost je setridena
        for j in range (len(x)-1, i, -1):
            if x[j] < x[j-1]:         #Probublavani na zacatek
                temp = x[j];          #Prohozeni za pouziti
                x[j] = x[j-1];        #pomocne promenne temp
                x[j-1] = temp;
                sort = False;         #Neni setridena
        i = i + 1
```

17. Třídění přímým výběrem:

Select Sort.

Efektivnější než Insert Sort.

Používá se na dotřídění krátkých posloupností u algoritmu QuickSort.

Složitost $O(N^2)$, počet přesunů $C_{max} = 0.25N^2 + 3(N-1)$.

Popis algoritmu:

Předchozí techniky třídění využívaly porovnání aktuálního prvku x_i s ostatními prvky posloupnosti, na základě výsledku této relace s prvkem x_i prováděly další operace.

Tato metoda třídění nevybírá prvky z nesetříděné části posloupnosti sekvenčně, ale na základě jejich hodnoty.

Prvek je vložen do setříděné části na předem známou pozici, která se již v průběhu třídění nemění, v předchozích algoritmech se jeho hodnota měnila (odsouvání či prohazování).

18. Princip algoritmu

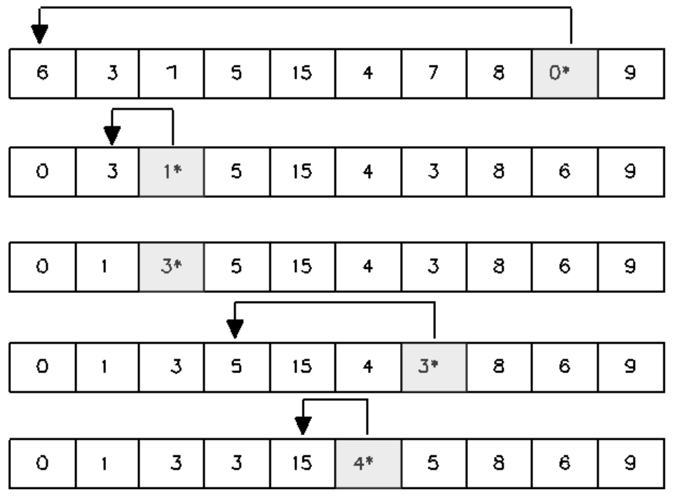
Tříděná posloupnost je rozdělena na dvě části: levá obsahuje již setříděné prvky posloupnosti ve správném pořadí, pravá prvky dosud nesetříděné.

Start: $\dim(S) = 0$, $\dim(N) = n$.

1. V prvním kroku nalezneme hodnotu $x_{min} = \min(x_i), i = 1, .., N$, tj. prohledáváme celou posloupnost prvků.
2. Hodnotu x_{min} prohodíme s hodnotou x_0. Prvek x_0 se nachází na správné pozici, jeho poloha se již v průběhu třídění nebude měnit.
3. V druhém kroku hledáme minimum z nesetříděné části prvků, tj. $x_{min} = \min(x_i), i = 2, .., N$. Tento prvek prohodíme s prvkem x_1. Poloha prvku x_1 se již při dalším třídění nebude měnit.
4. Opakujeme.
5. Poslední nesetříděný prvek x_n bude umístěn na správném místě.

End: $\dim(S) = n$, $\dim(N) = 0$.

19. Ukázka algoritmu



20. Zdrojový kód

Třídění realizováno dvojicí cyklů.

```
def selectSort(x) :
    for i in range (0, len(x) - 1):
        xmin = x[i];                        #Inicializace minima
        imin = i                            #Inicializace jeho indexu
        for j in range (i, len(x)):         #Hledani minima
            if x[j] < xmin:
                xmin = x[j]                 #Minimum
                imin = j                    #Jeho index
        temp = x[i]                         #Prohozeni promennych
        x[i] = xmin;
        x[imin] = temp;
```

Modře znázorněna procedura hledání minima.

21. Modifikace algoritmu

Ve vnitřním cyklu nebudeme hledat minimum, ale pouze index minima.

Ušetření lokální proměnné.

```
def selectSort(x) :
    for i in range (0, len(x) - 1):
        imin = i                        #Inicializace indexu minima
        for j in range (i, len(x)):     #Hledani minima
            if x[j] < x[imin]:
                imin = j                #Jeho index
        temp = x[i]                     #Prohozeni promennych
        x[i] = x[imin]
        x[imin] = temp
```

22. Třídění se zmenšujícím se krokem

=Shell Sort.

Eliminuje neefektivitu spočívající v prohazování na krátké vzdálenosti.

Neprohazuje pouze sousední prvky, provádí prohazování prvků s krokem h.

Takový soubor nazýván $h-$*setříděný*.

Krok h se neustále zmenšuje, pro $h = 1$ je soubor setříděn.

Možnosti volby kroku h:

$$
\begin{aligned}
h_i &= 3h_{i-1} + 1, & h &= \{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, \ldots\},\\
h_i &= 2h_{i-1} + 1, & h &= \{1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \ldots\},\\
h_i &= 1 + 3 \cdot 2^{k-1} + 4^k, & h &= \{1, 8, 23, 77, 281, \ldots\}
\end{aligned}
$$

Pro malá h zbývá málo prohozů, většina vykonána v krocích s větším h.

Snížení složitosti až na $O(N^{3/2})$ (var. 1,2), popř. $O(N^{4/3})$ (var. 3).

Nejpříznivější ze všech elementárních algoritmů.

23. Princip algoritmu

Algoritmus je zobecněním Insert Sortu (nejčastější).

Pro prohazování však existují i jiné metody (např. bublání).

Shell Sort pro $h = 1 \Rightarrow$ Insert Sort nad částečně setříděnou posloupností.

Postup:

- Výpočet iniciační hodnoty kroku $h = 3h + 1$, $h < n$.
- Opakuj pro $h = h/3$
  - Insert Sort s krokem h.
  - Neporovnáváme sousední prvky $x[j]$ a $x[j-1]$, ale $x[j]$ a $x[j-h]$.

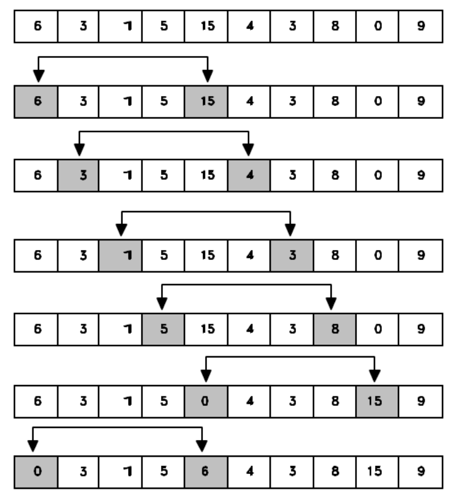
Důsledky:

Ve vnějším cyklu Insert Sort nahradíme inicializaci i=0 za $i = h$

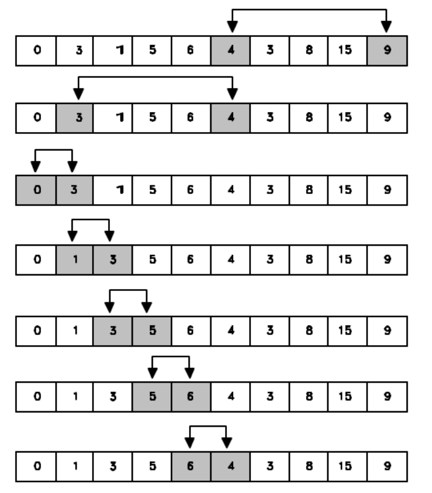
Ve vnitřním cyklu Insert Sortu nahradíme všechny výskyty indexu $j - 1$ za $j - h$.

Jednoduchá implementace, pouze malá změna algoritmu Insert Sort přinese značné zvýšení výkonu.

24. Ukázka algoritmu



25. Ukázka algoritmu



26. Zdrojový kód

```
def shellSort(x):
    h = 1
    while h < len(x)-1:
        h = 3 * h + 1
    h = int(h / 3)1
    h = int(h / 3)
    while(h > 0):
        for i in range (h, len(x)):    #Inkrementace i o h
            j = i
            p = x[i]
            while ((j >= h) and (x[j - h] > p)):
                x[j] = x[j - h];        #Posun prvku o h pozic vpravo
                j = j - h;              #Dekrementace indexu o h
            x[j] = p                    #Zarazeni prvku na spravne mist
        h = int(h/3)                    #Snizeni kroku
```

Modře znázorněn Insertion Sort.

Pozor: netestujeme prvky x[j] a x[j-1] ale x[j] a x[j-h]!

28. QuickSort

Jeden z nejpoužívanějších třídících algoritmů.

Využívá principu rozděl a panuj.

Lze implementovat rekurzivně i nerekurzivně.

Pro většinu vstupních dat nejrychlejší známý algoritmus.

Vysokého výkonu dosáhne prohazování prvků na **velké** vzdálenosti.

Citlivost vůči vstupním datům.

Inspirace:

Posloupnost prvků seřazených sestupně: $\forall i \in \langle 1, n \rangle$ platí $x_i \geq x_{i+1}$.

Rychlé vzestupné setřídění této posloupnosti:

Postupné prohazování

$$x_1 \Leftrightarrow x_n, x_2 \Leftrightarrow x_{n-1}, \dots$$

Výměna prvního prvku s posledním, druhého s předposledním,...

K setřídění nebude potřeba $n - 1$ výměn, ale pouze $n/2$.

29. Princip QuickSortu

- V prvním kroku je zvolena střední hodnota $\overline{x}$, tzv. *pivot*.
- Posloupnost je uspořádána rozdělením na dvě části tak, že levá část obsahuje prvky $x_i < \overline{x}$, pravá část prvky $x_i > \overline{x}$.

$x_i<\overline{x}$	$\overline{x}$	$x_i>\overline{x}$

- Hodnota pivotu $\overline{x}$ může být zvolena náhodně, představovat medián či aritmetický průměr (viz dále).
- Každá z těchto částí je stejným způsobem rekurzivně rozdělena.
- Počty úseků se s každým dělením zdvojnásobují, délky úseků (tj. počty prvků) snižují na polovinu.
- Dělení provádíme tak dlouho, dokud délky úseků nejsou rovny jedné.
- V takovém případě je posloupnost setříděna.

30. Volba pivota

A) první prvek $\overline{x} = x_1$

Pivot je představován prvním prvkem z posloupnosti.

Jedná se o nejméně výhodnou volbu, složitost třídění $O(N^2)$.

Degenerovaný strom, výška $N - 1$.

B) náhodný prvek $\overline{x} = rand(x_i)$

Nejčastěji používaná varianta, složitost algoritmu je $\Theta(N \log N)$, ale $O(N^2)$!

Algoritmus je ve většině případů téměř tak rychlý, jako při použití mediánu.

Pravděpodobnost, že při každém náhodném volání nalezen první prvek, malá.

C) medián $\overline{x} = E(x)$

Nejlepší varianta, strom s výškou $log_2(N)$.

Ale složitost nalezení mediánu je $O(N)$, složitost celkem $O(N^2)$.

Aproximace mediánem ze 3, 5 náhodných hodnot.

Nevýhodou QuickSort-u proměnná časová složitost závisející na pivotovi.

31. QuickSort

Pseudokód algoritmu QuickSort

```
quickSort(x, l, r):
    m = (l + r)//2
    p = x[m]  #Prostredni prvek (odhad medianu)
    /*
    Vlastni trideni
    */

    if condit1:
       quickSort(x, l, m-1)   #Setrideni leve poloviny
    if condit2:
       quickSort(x, m+1, r)   #Setrideni prave poloviny
```

QuickSort lze realizovat s použitím rekurze či převodem na zásobník.

32. Časová složitost QuickSortu: ideální případ

Odhad složitosti:

$$
\begin{aligned}
T(1) &= 1\\
T(n) &= T(n/2) + T(n/2) + n,\\
&= 2T(n/2) + n,\\
&= 2[2T(n/4) + n] + n,\\
&= 4T(n/4) + 2n,\\
&= 8T(n/8) + 3n,\\
&= 2^i T(n/2^i) + in.
\end{aligned}
$$

Substituce $n = 2^i \Rightarrow i = \log_2 n$

$$
\begin{aligned}
T(n) &= nT(1) + n\log_2 n,\\
&= n + n\log_2 n,\\
&\doteq n\log_2 n
\end{aligned}
$$

Doba běhu funkcí velikostí vstupu: $O(n\log_2 n)$.

33. Časová složitost QuickSortu: degenerovaný strom

Odhad složitosti:

$$
\begin{aligned}
T(n) &= n + T(n-1), \\
&= n + n - 1 + T(n-2), \\
&= n + n - 1 + \dots + n - i + T(n-i), \\
&= n + n - 1 + \dots + 1, \\
&= (1+n)\frac{n}{2}, \\
&\doteq n^2.
\end{aligned}
$$

Pivot nejmenší/největší prvek:

Doba běhu kvadratickou funkcí velikostí vstupu: $O(n^2)$.

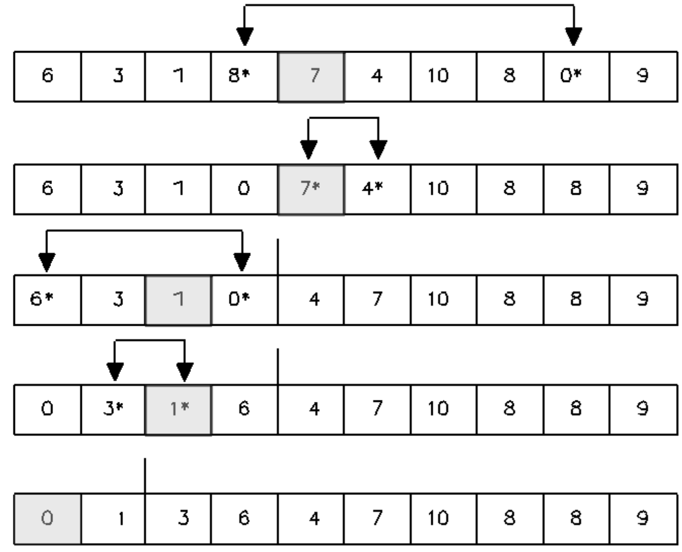
34. Časová složitost QuickSortu: medián exaktně

Odhad složitosti:

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= T(n/2) + T(n/2) + n + n, \\
&= 2T(n/2) + 2n, \\
&= 2(T(n/2) + n), \\
&= 4T(n/4) + 3n, \\
&= 8T(n/8) + 7n, \\
&= iT(n/i) + (i-1)n, \\
&\dots \\
&= nT(1) + (n-1)n, \\
&= n + (n-1)n, \\
&\doteq n^2.
\end{aligned}
$$

Výpočet mediánu v $O(n)$: doba běhu funkcí velikostí vstupu: $O(n^2)$.

35. Znázornění algoritmu QuickSort



36. Zdrojový kód prohazování prvků

Vnější cyklus probíhá, dokud se indexy nezkříží:

```
while i <= j:
        while x[i] < p:
            i = i + 1      #Hledani prvku zleva, ktery se vymeni
        while p < x[j]:
            j = j - 1      #Hledani prvku zprava, ktery se vymeni
        if (i <= j):       #Prohozeni obou prvku
            temp = x[i]
            x[i] = x[j]
            x[j] = temp
            i = i + 1      #Posun leveho indexu vpravo
            j = j - 1      #Posun praveho indexu vlevo
```

Index *i*: posune se na prvek za pivotem.

Index j: posune se na prvek před pivotem.

37. Zdrojový kód QuickSortu

```
def  quickSort(x, l, r):
    i, j = l, r;
    p = x[(l + r) // 2]    #Urceni pivota
    while i <= j:
        while x[i] < p:
            i = i + 1      #Hledani prvku zleva, ktery se vymeni
        while p < x[j]:
            j = j - 1      #Hledani prvku zprava, ktery se vymeni
        if (i <= j):       #Prohozeni obou prvku
            temp = x[i]
            x[i] = x[j]
            x[j] = temp
            i = i + 1      #Posun leveho indexu vpravo
            j = j - 1      #Posun praveho indexu vlevo
    if (l < j):
        quickSort(x,l,j);  #Rekurze: leva cast
    if (r > i):
        quickSort(x,i,r);  #Rekurze: prava cast
```

38. Zhodnocení algoritmu QuickSort

Algoritmus QuickSort je velmi efektivní pro třídění velkých polí.

Nejrychlejší třídící algoritmus.

Efektivita výrazně závislá na volbě pivota.

Optimální, pokud levá a pravá část stejně velké.

Z tohoto důvodu není používán v časově kritických aplikacích.

Pro třídění malých polí ($n < 12$) QS neefektivní (mnoho rekurzivních volání).

Dosahuje horších výsledků než Insert Sort.

V praxi je QuickSort ukončován pro $n < 12$.

Posloupnost následně dotříděna jiným způsobem, např. Select Sort.

Hybridní třídění, přepínání mezi 2 metodami.

Není stabilní.

39. Tříděné sléváním

= Merge Sort.

Založen na principu Rozděl a panuj, rekurzivní algoritmus.

Rozdělení = rekurze (v mediánu), spojení (merge).

Využívá princip **zatřiďování:**

Sloučení 2 menších setříděných posloupností do 1 větší setříděné.

Složitost algoritmu (na rozdíl od QuickSortu) není závislá na pivotovi.

Ve většině případů je MergeSort pomalejší než QuickSort.

Necitlivost vůči vstupním datům, stabilní.

Princip algoritmu:

Rozdělení posloupnosti s n prvky na dvě podposloupnosti s $\frac{n}{2}$ prvky.

Dělení rekurzivně opakováno, dokud podposloupnosti nejsou tvořeny 1 prvkem.

Opakované spojování 2 posloupností v 1, dokud nemá délku n.

Používán jako standardní třídící algoritmus v mnoha progr. jazycích (Java, C++).

Součástí Tim Sortu (Python): hybridní algoritmus (Insertion + Merge).

40. Vlastní algoritmus třídění sléváním

Pole je rozděleno na dvě části.

Na každou část rekurzivně voláno setřídění.

Obě poloviny následně zatříděny (spojeny).

```
def mergeSort(x, l, r):
        mid = (l + r) // 2
        if r <= l:
            return;
        mergeSort(x, l, mid)     #Setrid levou cast
        mergeSort(x, mid + 1, r) #Setrid pravou cast
        merge(x, l, mid, r)      #Spoj obe casti
```

Nevýhodou dodatečná paměť na pole při zatřiďování.

41. Znázornění algoritmu

6	3	1	5	15	4	8	7	9	0
3	**6**	1	5	15	4	8	7	9	0
1	**3**	**6**	5	15	4	8	7	9	0
1	3	6	**5**	**15**	4	8	7	9	0
1	**3**	**5**	**6**	**15**	4	8	7	9	0
1	3	5	6	15	**4**	**8**	7	9	0
1	3	5	6	15	**4**	**7**	**8**	9	0
1	3	5	6	15	4	7	8	**0**	**9**
1	3	5	6	15	**0**	**4**	**7**	**8**	**9**

42. Dvoucestné zatřiďování

Pole a má n prvků, pole b má m prvků, obě setříděná.

Slučované pole c tvořeno $n+m$ prvky

$$a = \{a_i\}_{i=1}^{n}, \quad b = \{b_j\}_{j=1}^{m}, \quad c = \{c_k\}_{k=1}^{m+n}.$$

Platí

$$a_i \leq a_{i+1} \wedge b_j \leq b_{j+1} \Rightarrow c_k \leq c_{k+1}.$$

Při každém průchodu cyklem nalezneme minimální hodnotu z a_i, b_j.

Přidáme ji do slučovaného pole c

$$c_k = \min(a_i, b_j).$$

Nutno vytvořit dvě pomocná pole, dodatečné paměťové nároky.

Dodatečné testování indexu konce polí.

Režie spojená s kopírováním vstupního pole do obou pomocných polí.

V praxi se příliš nepoužívá.

43. Dvoucestné zatřiďování

Zatřídění $a = \{a_i\}_{i=1}^{n}$, $b = \{b_j\}_{j=1}^{m}$ do $c = \{c_k\}_{k=1}^{m+n}$.

```
def merge(c, a, b):
   n = len(a), m = len(b)
   i = 0, j = 0
   for k in range (0, n + m):
      if i == N:                  #Pole a uz nema dalsi prvky
         c[k] = b[j]
         j = j + 1
         continue                 #Skok na dalsi iteraci
      if j == M :                 #Pole b uz nema dalsi prvky
         c[k] = a[i]
         i = i + 1
         continue                 #Skok na dalsi iteraci
     if a[i] < b[j]:
         c[k] =  a[i]             #Pridani mensiho prvku z a
         i = i + 1
     else:
         c[k] = b[j]              #Pridani mensiho prvku z b
         j = j + 1
```

44. Merge Sort: dvoucestné zatřiďování

Zatřiďujeme c pouze v intervalu $< l, r >$: aktuálně zpracovávaný úsek.

```
def merge(c, l, mid, r):
   a = c[l:mid + 1]          #Pomocne pole a
   b = c[mid + 1, r+1]       #Pomocne pole b
   n = len(a), m = len(b)    #Delky obou casti
   i = 0, j = 0
   for k in range (l, r+1):  #Pouze v intervalu <l,r>
      if i == n:             #Pole a uz nema dalsi prvky
         c[k] = b[j]
         j = j + 1
         continue            #Skok na dalsi iteraci
      if j == m :            #Pole b uz nema dalsi prvky
         c[k] = a[i]
         i = i + 1
         continue            #Skok na dalsi iteraci
     if a[i] < b[j]:
         c[k] =  a[i]        #Pridani mensiho prvku z a
         i = i + 1
     else:
         c[k] = b[j]         #Pridani mensiho prvku z b
         j = j + 1
```

45. Zatřiďování na místě

Dvoucestné zatřiďování je neefektivní.

- Nutnost alokace paměti pro pomocná pole.
- Opakované testování dosažení konce pole

Vylepšení:

Pouze jedno pomocné pole, rychlejší.

Levá polovina pomocného pole obsahuje kopii levé poloviny vstupního pole.

Pravá polovina pomocného pole obsahuje kopii pravé poloviny vstupního pole v opačném pořadí.

Index *i* od levého prvku, index *j* od pravého prvku.

Největší prvek přirozenou zarážkou.

46. Zatřiďování na místě

```
def merge (a, l, m, r):
   c = range(n)                 #Pomocne pole
   for i in range (m + 1, l, -1):
      c[i-1] = a[i -1]          #Kopie 1. pole, i -> l
   for j in range (m, p):
      c[p + m - j] = a[j+1]     #Kopie 2. pole opacne, j -> p
   for k in range (l, r+1):
      if (c[i] < c[j]):         #Nalezeni mensiho prvku
         a[k] = c[i]            #Pridani z 1. pole
         i = i + 1
      else:
         a[k] = c[j];           #Pridani z 2. pole
         j = j - 1
```

47. Ukázka zatřiďování na místě

c					a				
$i \Rightarrow$				$\Leftarrow j$	l		*mid*		*p*
					1	3	6	5	15
1	3	6	15	5					
	3	6	15	5	1				
		6	15	5	1	3			
		6	15		1	3	5		
			15		1	3	5	6	
					1	3	5	6	15

48. Halda

Halda (Heap): datová struktura představovaná binárním stromem.

Heap Sort: použita halda s globálním minimem v kořenu.

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 $h_1, h_2, ..., h_n$.

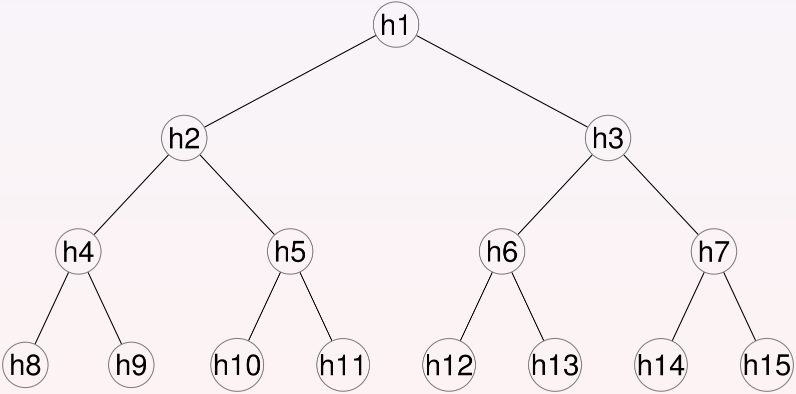
Všichni leví potomci mají sudý index.

Všichni praví potomci mají lichý index.

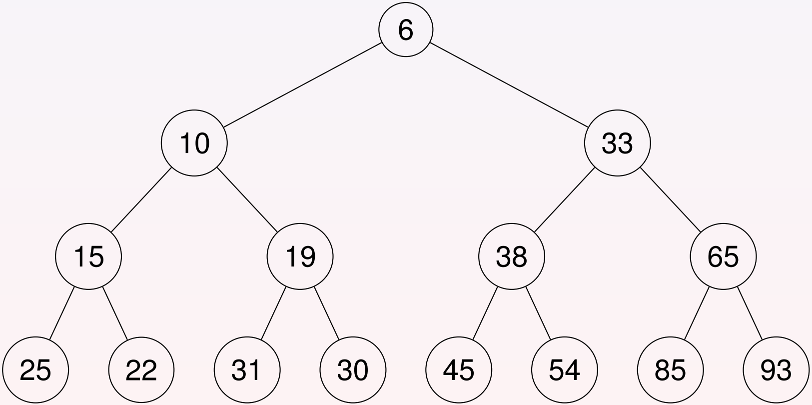
Př.: Kořen stromu h_1, h_2 levý potomek, h_3 pravý potomek.

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

49. Ukázka haldy



50. Ukázka haldy



51. Halda a její reprezentace

Důsledky plynoucí z definice:

- Binární strom je haldou právě když hodnota rodiče je menší nebo rovna hodnotě potomků.
- Žádný uzel nemá menší hodnotu než kořen.

Haldu lze reprezentovat polem, výhodou rychlý přístup.

[]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
h	x	6	10	33	15	19	38	65	25	22	31	30	45	54	85	[illegible]

Vztahy mezi pořadovým číslem uzlu i ve stromu a indexy pole[i]:

Rodič: [$i//2$].

Levý potomek: [$2i$].

Pravý potomek: [$2i+1$].

52. Operace nad heapem

Operace nad heapem nutné k setřídění posloupnosti:

- přidání prvku do heapu,
- oprava heapu nahoru: fixHeapUp,
- oprava heapu dolů: fixHeapDown,
- tisk heapu.

53. Oprava haldy nahoru

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

Stromová reprezentace:

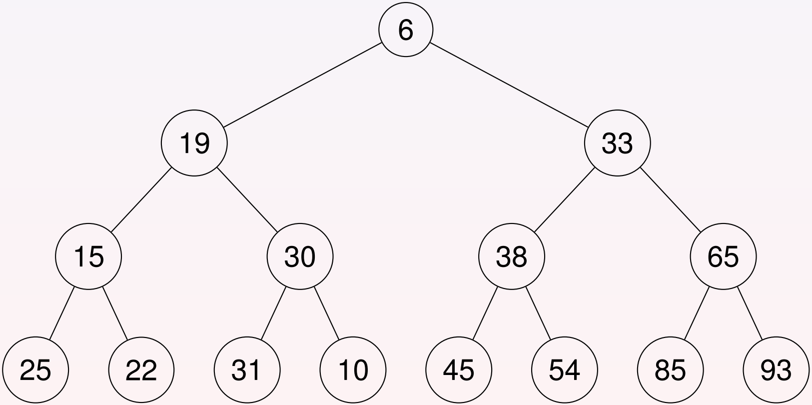
- Pokud předchůdce P uzlu U má vyšší hodnotu, prohodíme $U \Leftrightarrow P$.
- Pokračujeme v předchůdci $U = P$, dokud U není kořen.

Reprezentace polem:

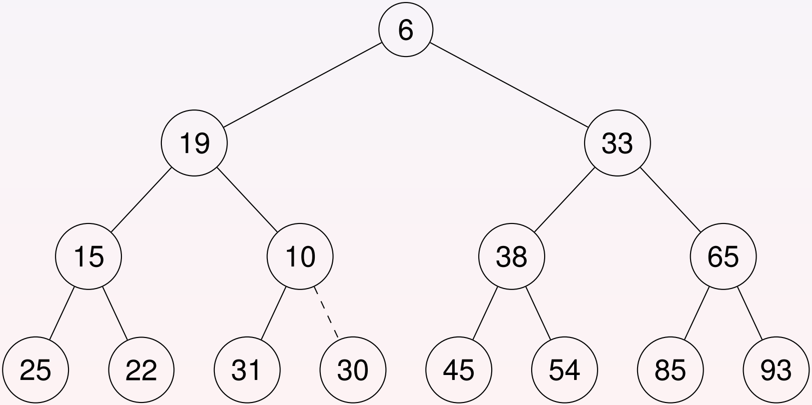
- Pokud prvek $[i/2]$ má menší hodnotu než prvek $[i]$, prohodíme $[i/2] \Leftrightarrow [i]$.
- Pokračujeme v předchůdci $i = i/2$ dokud $i > 1$.

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

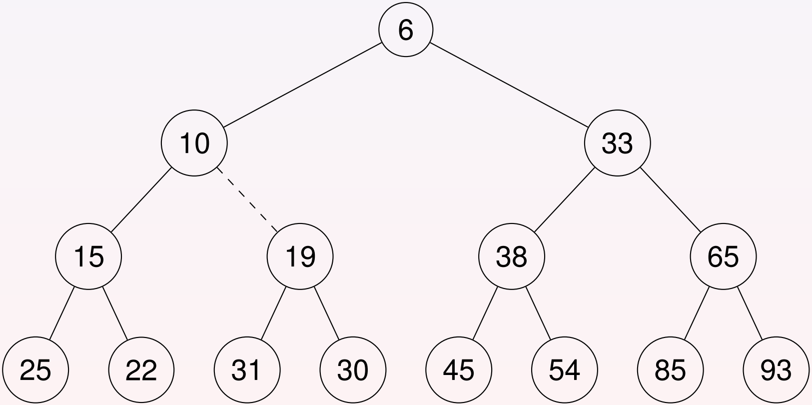
54. Ukázka opravy haldy nahoru



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



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



57. 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.

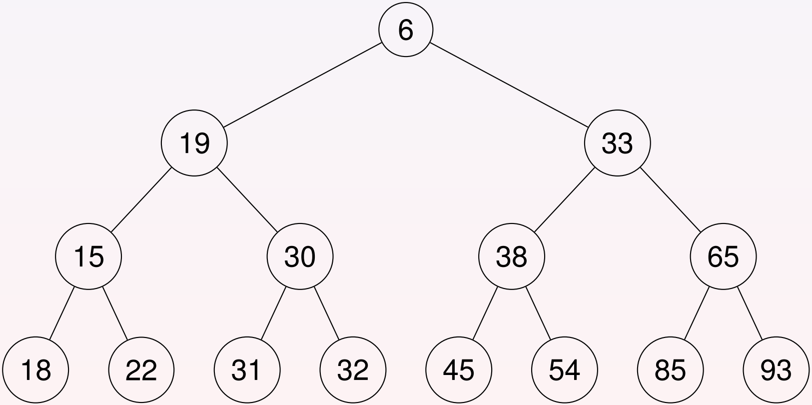
Reprezentace polem:

- 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

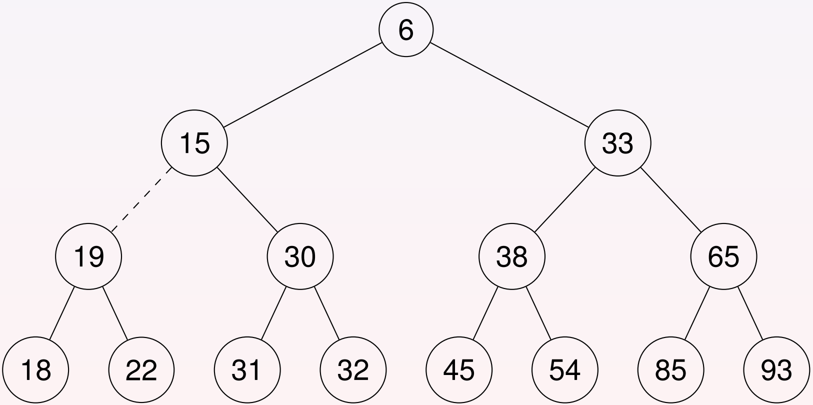
58. Algoritmus pro opravu haldy dolů

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

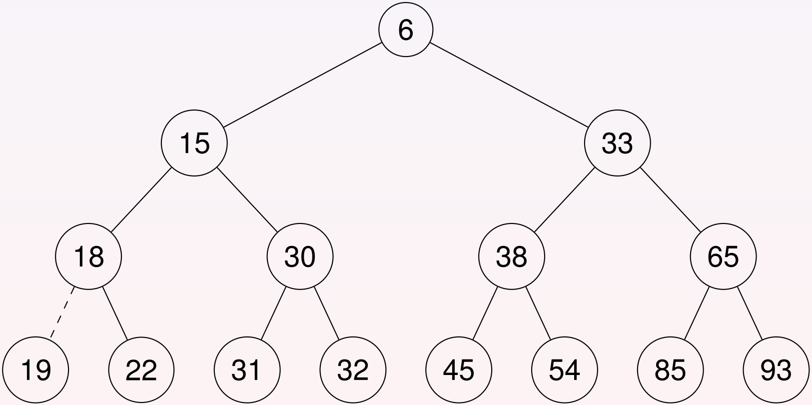
59. Ukázka opravy haldy dolů



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



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



62. Přidání prvku x_i do haldy

Opakuj, pro $\forall x_i \in x$:

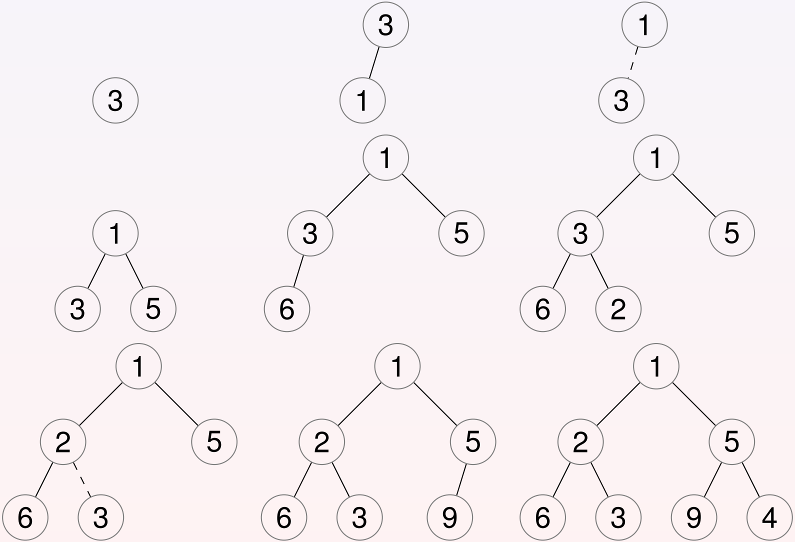
- Nový uzel U s hodnotou x_i vytvořen na poslední hladině.
- Propojení uzlu U s uzlem P ležícím na předposlední hladině co nejvíce vlevo.
- Oprava haldy nahoru U.

Heap představován polem h s délkou $n+1$, tj. o jednu pozici delší než počet tříděných prvků v x (přidáváme od 1).

Heap plněn po řádcích, po každém vložení inkrementován index i o 1.

```
for i in range(0, len(x)):
      h[i+1] = x[i];      #Pridej do heapu
      fixHeapUp(h, i+1); #Oprav heap nahoru do akt. pozice
```

63. Ukázka budování haldy



64. Vlastní třídící algoritmus

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.

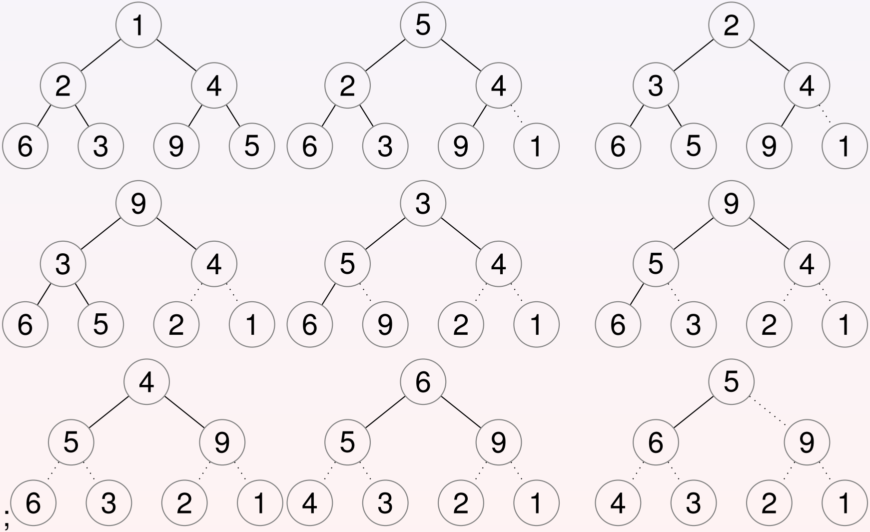
Nad polem snadno algoritmizovatelné.

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.

65. Ukázka činnosti algoritmu



;

66. Algoritmus Heap Sort

Pouze třídící procedura:

```
while (n > 1):              #Dokud haldu netvori jen koren
            temp = h[1]             #Vezmi nejmensi prvek
            h[1] = h[n]             #Prohod s poslednim
            h[n] = temp
            n = n-1                 #Zkrat heap o 1
            fixHeapDown(h, 1, n)  #Oprav heap dolu
```

Vlastnosti algoritmu:

Algoritmus má složitost $O(N \lg_2 N)$.

Asi o 20% pomalejší než Merge Sort a o 50% pomalejší než Quick Sort.

Použití pro hledání $k - t\acute{e}ho$ nejmenšího prvku.

Nevýhodou nestabilita.

67. Zdrojový kód (celek)

Inkrementální konstrukce: přidávání prvků do heapu + oprava nahoru.

Postupné krácení heapu + oprava dolů.

Posloupnost reverzně setříděna.

```
def heapSort(x):
        n = len(x)
        h = range[n + 1]          #Heap o jeden prvek delsi
        for i in range(0; n):     #Pridej vsechny prvky,oprav heap
            h[i+1] = x[i]         #Pridej do heapu
            fixHeapUp(h, i+1)     #Oprav heap shora
        while n > 1:              #Postupne zkracovani heapu
            temp = h[1]           #Vezmi nejmensi prvek
            h[1] = h[n]           #Prohod s poslednim
            h[n] = temp
            n = n - 1             #Zkrat heap
            fixHeapDown(h, 1, n) #Oprav heap dolu
```

68. Přehled třídících algoritmů

Metoda	$O(n)$	$\Theta(n)$	$\Theta(n)\|M$	*Stabilita*
Selection Sort	n^2	n^2	1	A
Bubble Sort	n^2	n^2	1	A
Insert Sort	n^2	n^2	1	A
Shell Sort	n^2	$n \lg n$	1	A
Merge Sort	$n \lg n$	$n \lg n$	n	A
Quick Sort	n^2	$n \lg n$	$\lg n$	N
Heap Sort	$n \lg n$	$n \lg n$	1	N

Další odkazy:

http://en.wikipedia.org/wiki/Sorting_algorithm

http://coderaptors.com/?All_sorting_algorithms

http://www.ida.liu.se/~TDDB28/mtrl/demo/sorting/index.sv.shtml

http://math.hws.edu/TMCM/java/xSortLab/