5. Třídění s pomocí datových struktur

Algoritmizace

Jiří Balun

Obsah

Rekapitulace a motivace

Na minulé přednášce

- datová struktura: strom
- Merge Sort – třídění slučováním
- dolní odhad složitosti algoritmů třídění porovnáním
- Radix Sort – číslicové třídění

Na této přednášce

- datové stuktury: halda a seznam
- Bucket Sort – přihrádkové třídění
- Heap Sort – třídění haldou

Datová struktura seznam

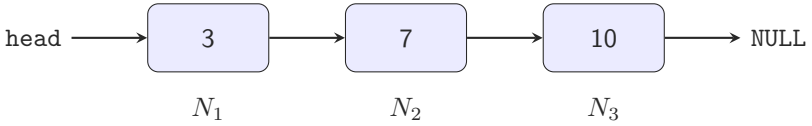
Motivace: proč potřebujeme *seznam*

Problém: Pole má pevnou velikost a prvky jsou v paměti uloženy za sebou. Co když chceme často vkládat nové prvky nebo neznáme předem počet ukládaných prvků?

Nevýhoda pole:

- vložení nebo odstranění prvku uprostřed pole vyžaduje posun zbylých prvků
- změna velikosti pole je obtížná nebo nemožná bez nové alokace

Řešení: Použijeme **seznam** – dynamickou strukturu, kde jsou prvky spojeny odkazy.



Obrázek: seznam se třemi uzly (N_1, N_2 a N_3) obsahující hodnoty 3, 7 a 10.

Datová struktura seznam

Definice:

- seznam tvoří **uzly (nodes)**, které obsahují:
  - `data` – uložená hodnota
  - `next` – ukazatel na následující uzel
- poslední uzel ukazuje na `null` (konec seznamu)
- první uzel nazýváme **hlava** (`head`)

Výhody:

- snadné vložení a odebrání prvku (stačí upravit ukazatele)
- délku seznamu (tj. počet prvků) lze libovolně měnit
- paměť se alokuje dynamicky
- vložení nového prvku na začátek je v čase $\Theta(1)$

Příklad: vkládání do seznamu

Příklad

Cíl: vložit nový uzel se zadanou hodnotou na začátek seznamu L.

Před vložením:

$$\texttt{head} \to [3] \to [7] \to [10] \to \texttt{null}$$

Po vložení uzlu s hodnotou 5:

$$\texttt{head} \to [5] \to [3] \to [7] \to [10] \to \texttt{null}$$

Princip:

- vytvoříme nový uzel N,
- nastavíme $N.\texttt{data} \leftarrow 5$ (zápis $N.\texttt{data}$ značí přístup ke složce `data` v uzlu N),
- nastavíme $N.\texttt{next} \leftarrow \texttt{head}$, což je uzel $[5]$, ze kterého je dosažitelný i zbytek seznamu,
- poté změníme `head` na N, tj. N je nyní první uzel seznamu L.

Bucket Sort

Motivace: proč Bucket Sort

Cíl: chceme setřídit čísla, která jsou rovnoměrně rozložená v intervalu $[0, 1)$.

Pozorování:

- pro některé typy dat je zbytečné provádět všechna porovnání (jako např. v Merge Sortu)
- hodnoty můžeme před samotným tříděním rozdělit do „přihrádek“ (angl. *buckets*) podle jejich velikosti, například na malé, střední,. . . , velké atd.
- každá přihrádka pak obsahuje jen malý počet prvků – ty lze snadno a rychle setřídit

Realizace:

- dopředu nevíme kolik prvků bude v jaké přihrádce, proto pro jejich implementaci použijeme **datovou strukturu seznamu**
- Bucket Sort tedy netřídí na místě (potřebuje další $\mathcal{O}(n)$ paměti pro pole a seznamy)
- netřídí porovnáním – pro rozdělení do přihrádek využíváme navíc informaci o tom, do jaké příhrádky, tj. podintervalu $[0, 1)$, daný prvek patří
- stabilita Bucket Sortu je závislá na zvoleném způsobu třídění jednotlivých přihrádek

Bucket Sort

Algorithm 1: bucket-sort

Input: array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ such that all values $a_i \in [0, 1)$
Output: sorted array A

1 initialize new array $B[0\,..\,n-1]$
2 **for** $i \leftarrow 0$ **to** $n-1$ **do**
3 $\quad B[i] \leftarrow$ `head` of new empty list
4 **for** $i \leftarrow 0$ **to** $n-1$ **do**
5 $\quad$ insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
6 **for** $i \leftarrow 0$ **to** $n-1$ **do**
7 $\quad$ sort$(B[i])$
8 write content of lists $B_0, \ldots, B_{n-1}$ (in this order) into A

- přihrádka $B[i]$ je seznam pro hodnoty z intervalu $[i/n, (i+1)/n)$
- pro setřídění $B[i]$ na řádku 7 lze použít např. Insertion Sort, který je efektivní pro málé množství prvků (a lze jej snadno implementovat pro třídění seznamů)

Příklad: Bucket Sort

Příklad (část 1/2)

Vstupní pole:

$$\langle 0.72, 0.17, 0.39, 0.21, 0.78, 0.94, 0.26, 0.12, 0.23, 0.68 \rangle$$

Krok 1: rozdělení do přihrádek (10 přihrádek pro intervaly $[0.0, 0.1), [0.1, 0.2), \ldots$)

B_0 : NULL
B_1 : $\boxed{0.12} \rightarrow \boxed{0.17} \rightarrow$ NULL
B_2 : $\boxed{0.23} \rightarrow \boxed{0.26} \rightarrow \boxed{0.21} \rightarrow$ NULL
B_3 : $\boxed{0.39} \rightarrow$ NULL
B_4 : NULL
B_5 : NULL
B_6 : $\boxed{0.68} \rightarrow$ NULL
B_7 : $\boxed{0.78} \rightarrow \boxed{0.72} \rightarrow$ NULL
B_8 : NULL
B_9 : $\boxed{0.94} \rightarrow$ NULL

Příklad (část 2/2)

Krok 2: setřídění jednotlivých přihrádek (obvykle Insertion Sortem)

$$
\begin{aligned}
B_0 &: \ \texttt{NULL} \\
B_1 &: \ \boxed{0.12} \rightarrow \boxed{0.17} \rightarrow \texttt{NULL} \\
B_2 &: \ \boxed{0.21} \rightarrow \boxed{0.23} \rightarrow \boxed{0.26} \rightarrow \texttt{NULL} \\
B_3 &: \ \boxed{0.39} \rightarrow \texttt{NULL} \\
B_4 &: \ \texttt{NULL} \\
B_5 &: \ \texttt{NULL} \\
B_6 &: \ \boxed{0.68} \rightarrow \texttt{NULL} \\
B_7 &: \ \boxed{0.72} \rightarrow \boxed{0.78} \rightarrow \texttt{NULL} \\
B_8 &: \ \texttt{NULL} \\
B_9 &: \ \boxed{0.94} \rightarrow \texttt{NULL}
\end{aligned}
$$

Krok 3: spojení přihrádek

$$
\langle \overbrace{0.12, 0.17}^{B_1}, \overbrace{0.21, 0.23, 0.26}^{B_2}, \overbrace{0.39}^{B_3}, \overbrace{0.68}^{B_6}, \overbrace{0.72, 0.78}^{B_7}, \overbrace{0.94}^{B_9} \rangle
$$

Časová složitost Bucket Sortu

Předpoklady analýzy:

- vstupní hodnoty jsou **rovnoměrně rozloženy** v intervalu $[0, 1)$
- máme n prvků a použijeme n přihrádek
- v průměru připadne do jedné přihrádky **jen několik málo prvků**

Časová náročnost jednotlivých kroků:

- inicializace a rozdělení prvků do přihrádek: $\Theta(n)$
- setřídění jednotlivých přihrádek (např. Insertion Sortem):
  - každá přihrádka obsahuje v průměru konstantně mnoho prvků, tj. $\Theta(1)$
  - třídíme n přihrádek v čase $\Theta(1)$, dohromady: $\Theta(n)$
- přepsání přihrádek zpět do původního pole: $\Theta(n)$
- **celková průměrná časová složitost:** $\Theta(n)$

Složitost v nejhorším případě:

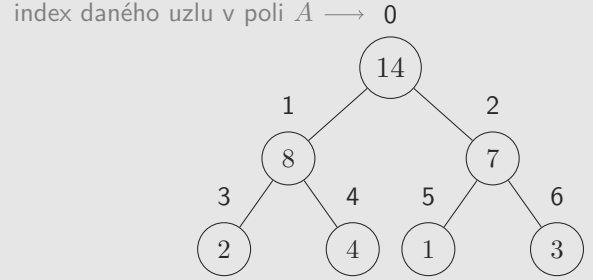
- pokud všechny prvky spadnou do jedné přihrádky a třídíme je Insertion Sortem: $\Theta(n^2)$
- v praxi se to stává jen tehdy, když vstup **není rovnoměrně rozložen**

Datová struktura halda

Příklad: max-halda (max-heap)

Příklad

Max-halda $A = \langle 14, 8, 7, 2, 4, 1, 3 \rangle$ a odpovídající stromová reprezentace:



- kořen obsahuje největší prvek (14),
- každý rodič má hodnotu větší nebo rovnou hodnotám svých potomků,
- strom je úplný – úrovně jsou zaplňovány zleva doprava,
- například $\text{left}(2) = 5$ a $\text{right}(2) = 6$, tj. $A[\text{left}(2)] = 1$ a $A[\text{right}(2)] = 3$.

Procedury pro přístup k uzlům

Algorithm 2: parent

Input: index i
Output: index of parent of node i

1 **return** $\lfloor (i-1)/2 \rfloor$

Algorithm 3: left

Input: index i
Output: index of left child of node i

1 **return** $2i+1$

Algorithm 4: right

Input: index i
Output: index of right child of node i

1 **return** $2i+2$

Halda (heap)

Intuice: halda je stromová datová struktura (reprezentovaná v poli), která je vhodná pro rychlé získání největšího (max-heap) nebo nejmenšího prvku (min-heap).

Definice

Pole $A[0\,..\,n-1]$ se nazývá **max-halda**, pokud pro každý index $i = 1, \ldots, n-1$ platí, že $A[i] \leq A[\text{parent}(i)]$ (této nerovnosti říkáme vlastnost max-haldy).

- max-heap: každý uzel je větší než jeho potomci a kořen obsahuje největší prvek
- analogicky můžeme definovat min-haldu
- reprezentuje se jako **úplný binární strom**
  - všechny úrovně haldy jsou zcela zaplněny, kromě poslední (ta nemusí)
  - poslední úroveň je zaplňována zleva doprava
  - výška haldy je $\Theta(\log n)$ vzhledem k počtu uložených prvků
- heap-size$(A) = m$ určuje velikost haldy v A, tj. podpole $A[0\,..\,m]$

Heap Sort

Motivace: proč potřebujeme Heap Sort

Cíl: najít třídicí algoritmus s optimální časovou složitostí, který nepotřebuje další paměť.

Pozorování:

- Merge Sort třídí v čase $\Theta(n \log n)$, ale potřebuje $\Theta(n)$ paměti navíc
- Insertion Sort a Quick Sort třídí na místě, ale v nejhorším případě trvají $\Theta(n^2)$
- halda nám umožňuje rychle najít a odstranit největší prvek

Myšlenka Heap Sortu:

- v nesetříděné části pole udržujeme **max-haldu** (třídíme na místě)
- opakovaně „vytahujeme“ největší prvek z kořene haldy a ukládáme jej na konec pole

Heap Sort se skládá ze 3 procedur:

1. max-heapify(A, i) – opraví vlastnost max-haldy pro uzel i
2. build-max-heap(A) – sestaví v poli A max-haldu
3. heap-sort(A) – samotné třídění využívající haldy a předchozích dvou procedur

Procedura max-heapify

Algorithm 5: max-heapify

Input: array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ and index i
Output: max-heap with root $A[i]$

```
1  l ← left(i)
2  r ← right(i)
3  if l ≤ heap-size(A) and A[l] > A[i] then
4      largest ← l
5  else
6      largest ← i
7  if r ≤ heap-size(A) and A[r] > A[i] then
8      largest ← r
9  if largest ≠ i then
10     swap(A[i], A[largest])
11     max-heapify(A, largest)
```

Intuice procedury *Max-Heapify*

Cíl procedury: obnovit vlastnost **max-haldy** v uzlu i, pokud může nastat, že nějaký jeho přímý **potomek je větší** než hodnota uzlu i.

Předpoklad:

- levý i pravý podstrom uzlu i již jsou **max-haldy** (pro jiné vstupy procedura nefunguje)
- jediná možná chyba je v kořeni těchto dvou hald, tedy na pozici i

Hlavní myšlenka:

- max-heapify „opravuje“ haldu shora dolů a zajišťuje, že ze dvou max-hald vznikne opět jedna max-halda
- porovnáme hodnotu v uzlu i s hodnotami obou jeho potomků
- největší z těchto tří uzlů prohodíme do kořene
- tím vzniknou **dvě max-haldy**, ale u uzlu potomka mohla vzniknout nová chyba
- proto voláme max-heapify rekurzivně tam, kde se prvek posunul

Příklad: max-heapify

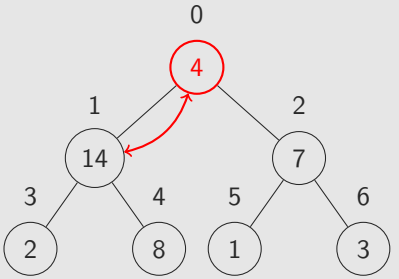
Příklad (část 1/3)

Vstupní halda A, kde heap-size$(A) = 6$**:**

$$A = \langle 4, 14, 7, 2, 8, 1, 3 \rangle$$

Voláme max-heapify$(A, 0)$**:**

- porovnáme $A[0] = 4$ s jeho potomky,
- levý potomek: left$(0) = 1$ a $A[1] = 14$,
- pravý potomek: right$(0) = 2$ a $A[2] = 7$,
- největší z nich je 14, proto $largest \leftarrow 1$,
- vyměníme $A[0]$ a $A[largest]$, tj. prvky 4 a 14,
- po výměně musíme pokračovat rekurzivním voláním max-heapify$(A, 1)$.



Poznámka: všimněte si, že levý i pravý podstrom uzlu 4 již tvoří max-haldy.

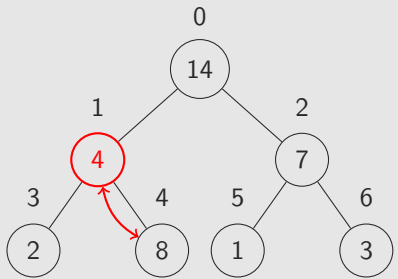
Příklad (část (2/3)

Po první výměně máme:

$$A = \langle 14,\ 4,\ 7,\ 2,\ 8,\ 1,\ 3 \rangle$$

Voláme max-heapify$(A, 1)$**:**

- porovnáme $A[1] = 4$ s jeho potomky
- levý potomek: left$(1) = 3$ a $A[3] = 2$
- pravý potomek: right$(1) = 4$ a $A[4] = 8$
- největší z nich je 8, proto $largest \leftarrow 4$
- vyměníme $A[1]$ a $A[largest]$, tj. prvky 4 a 8
- po výměně musíme pokračovat rekurzivním voláním max-heapify$(A, 4)$



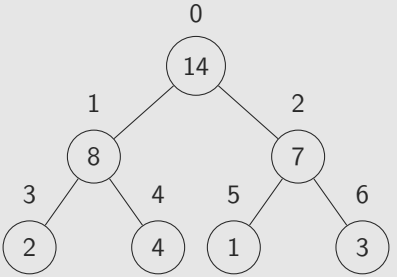
Příklad (3/3)

Po druhé výměně:

$$A = \langle 14, 8, 7, 2, 4, 1, 3 \rangle$$

Voláme max-heapify$(A, 4)$**:**

- platí, že left$(4) = 9 >$ heap-size(A) a zároveň right$(4) = 10 >$ heap-size(A),
- uzel $A[4] = 4$ tedy nemá žádné potomky a rekurze končí.



Výsledek: vlastnost max-heapu byla obnovena.

Časová složitost procedury *Max-Heapify*

Co procedura dělá:

- porovná uzel i s nejvýše dvěma jeho potomky – to je **konstantní čas** $\Theta(1)$
- pokud je některý potomek uzlu i větší, prohodí tyto prvky a rekurzivně se zavolá na potomka, ve kterém došlo k prohození

Jak hluboko může rekurze pokračovat?

- v nejhorším případě se prvek „propadne“ z kořene až na list
- halda je **úplný binární strom**, proto má výšku

$$h = \Theta(\log n)$$

- na každé úrovni se provádí jen konstantní množství práce (porovnání a prohození)
- max-heapify trvá jen čas úměrný výšce haldy, tedy:

$$T(n) = \Theta(\log n)$$

Sestavení haldy

Intuice: max-haldu stavíme z neuspořádaného pole tak, že zavoláme max-heapify na všech nelistových uzlech (začneme od posledního nelistového uzlu až po kořen).

Algorithm 6: build-max-heap

Input: array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$
Output: max-heap of array A

```
1 heap-size(A) ← n − 1
2 for i ← ⌊n/2⌋ − 1 downto 0 do
3 |   max-heapify(A, i)
```

- na řádku 1 nastavujeme heap-size$(A) \leftarrow n - 1$, tj. chceme sestavit haldu z celého A
- uzly v druhé polovině A jsou listové, a proto tvoří triviální jednoprvkové max-haldy
- je snadné vidět, že složitost je nejhůře $\mathcal{O}(n \log n)$, protože $(\lfloor n/2 \rfloor - 1)$-krát voláme max-heapify se složitostí $\mathcal{O}(\log n)$ (podrobnější analýzou lze ukázat i lepší odhad $\mathcal{O}(n)$)

Příklad: build-max-heap

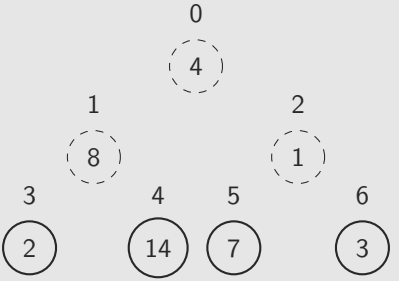
Příklad (1/3)

Vstupní pole A obsahující 7 prvků, tj. $n = 7$:

$$A = \langle 4,\ 8,\ 1,\ 2,\ 14,\ 7,\ 3 \rangle$$

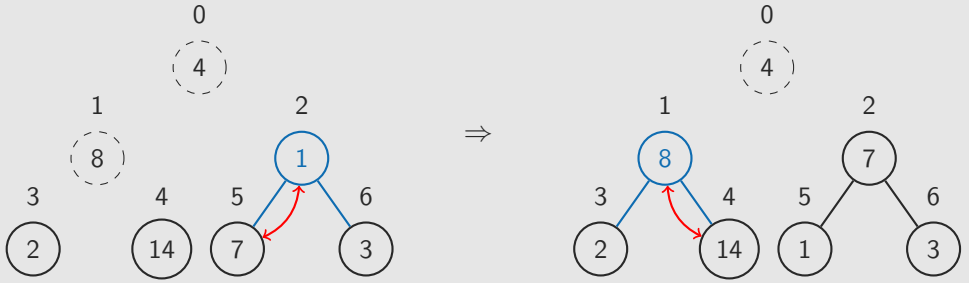
Voláme build-max-heap(A)**:**

1. nastavíme heap-size$(A) = n - 1 = 6$ (tedy 6 je největší index pole A),
2. na začátku každý listový uzel tvoří triviální jednoprvkovou max-haldu (uzly v $A[3\,..\,6]$).



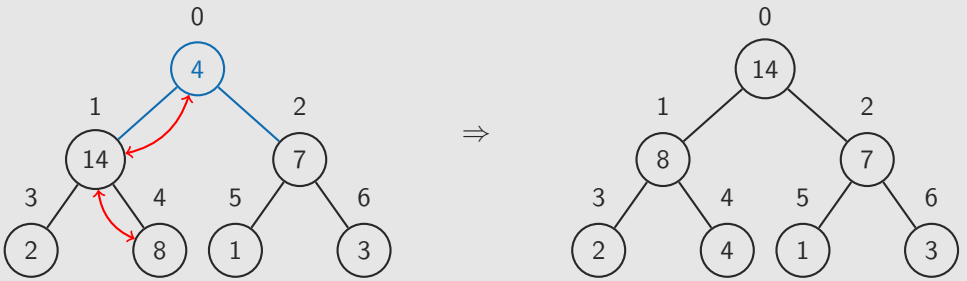
Příklad (2/3)

3. voláme max-heapify$(A, 2)$, který vytvoří haldu ze třech uzlů $A[2]$, $A[5]$ a $A[6]$:
   - levý podstrom (uzel $A[5]$) i pravý podstrom (uzel $A[6]$) jsou max-haldy,
   - můžeme z nich vytvořit novou haldu s kořenem $A[2]$, přičemž prohodíme prvky 1 a 7,
4. a poté voláme max-heapify$(A, 1)$, který vytvoří haldu ze třech uzlů $A[1]$, $A[3]$ a $A[4]$:
   - levý podstrom (uzel $A[3]$) i pravý podstrom (uzel $A[4]$) jsou max-haldy,
   - můžeme z nich vytvořit novou haldu s kořenem $A[1]$, přičemž prohodíme prvky 8 a 14.



Příklad (3/3)

5. nakonec voláme max-heapify$(A, 0)$, který vytvoří haldu z celého pole A:
   - levý podstrom (daný uzlem $A[1]$) i pravý podstrom (daný uzlem $A[2]$) jsou max-haldy,
   - můžeme z nich vytvořit novou haldu s kořenem $A[0]$,
   - přídáním dojde nejdříve k prohození prvků 4 a 14, a poté 4 a 8.



Heap Sort

Intuice: v každé iteraci vezmeme z haldy největší prvek a vyměníme jej s prvkem na konci pole – tím porušíme vlastnost max-haldy, ale zase ji opravíme pomocí max-heapify.

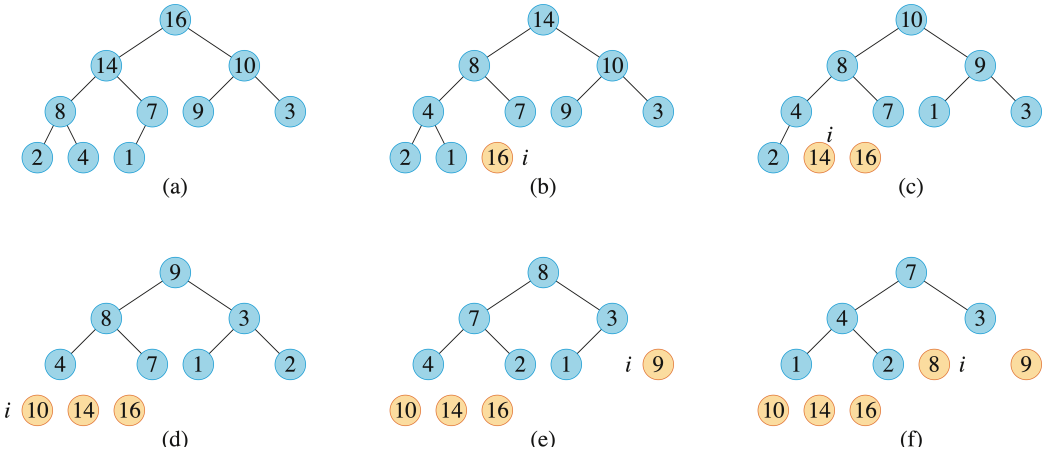
Algorithm 7: heap-sort

Input: array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$
Output: sorted array A

```
1 build-max-heap(A)
2 for i ← n − 1 downto 1 do
3 |   swap(A[0], A[i])
4 |   heap-size(A) ← heap-size(A) − 1
5 |   max-heapify(A, 0)
```

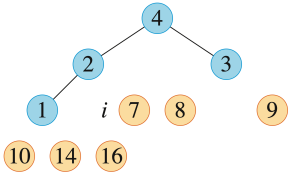
- pomocí zmenšování heap-size(A) dělíme pole na nesetříděnou (tj. $A[0\,..\,\text{heap-size}(A)]$ obsahuje haldu) a setříděnou část ($A[\text{heap-size}(A) + 1\,..\,n - 1]$)
- Heap Sort třídí na místě pomocí porovnání, ale není stabilní

Příklad – Heap Sort (část 1/2)

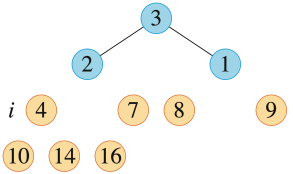


Obrázek: v části a) začínáme s max-haldou.
Zdroj: Introduction to Algorithms (Cormen et al.)

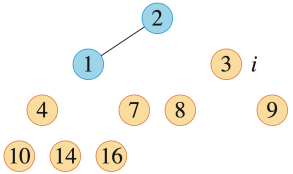
Příklad – Heap Sort (část 2/2)



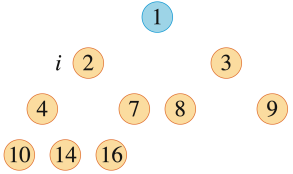
(g)



(h)



(i)



(j)

(k)

Zdroj obrázku: Introduction to Algorithms (Cormen et al.)

Korektnost algoritmu *Heap Sort*

Invariant 1 – před každou iterací cyklu: $A[0\,..\,\text{heap-size} - 1]$ **je max-halda**

- `build-max-heap` vytvoří počáteční max-haldu
- každé volání `max-heapify` po výměně kořene haldu znovu opraví

Invariant 2 – po každé iteraci cyklu: prvek na pozici i je i-tý největší prvek

- největší prvek haldy je vždy v kořeni
- výměna $A[0]$ a $A[i]$ umístí největší aktuální prvek na konec
- tento prvek se již dále nemění (je zatříděn na správném místě)

Invariant 3 – zmenšováním `heap-size` třídíme pole odzadu

- s každým krokem se hledá maximum v menší a menší haldě
- po $n - 1$ iteracích je celé pole setříděno vzestupně

Závěr: Heap Sort vždy postupně umisťuje největší prvky na jejich konečné pozice. Proto po dokončení cyklu platí, že A je **vzestupně setříděné pole**.

Časová složitost *Heap Sortu*

Pozorování:

- Heap Sort musí vždy sestavit haldu bez ohledu na vstupní pole (např. i pro setříděné)
- halda je sama o sobě nesetříděná a musíme ji setřídit (vždy stejným procesem)
- proto se Heap Sort **chová stejně v nejlepším, průměrném i nejhorším případě**

Složky celkové práce:

1. procedura build-max-heap(A) vytváří max-haldu v lineárním čase $\Theta(n)$
2. hlavní cyklus (od $i = n - 1$ zmenšujeme k 1)
   - výměna prvků: $\Theta(1)$
   - zmenšení heap-size(A): $\Theta(1)$
   - volání max-heapify$(A, 0)$: $\Theta(\log n)$

   Celkem se max-heapify volá $(n - 1)$-krát:

$$(n - 1) \cdot \Theta(\log n) = \Theta(n \log n)$$

Celková časová složitost:

$$T(n) = \Theta(n) + \Theta(n \log n) = \Theta(n \log n)$$

Závěrem

Problém třídění:

- Bucket Sort – přihrádkové třídění
  - časová složitost v průměrném případě je $\Theta(n)$
  - časová složitost v nejhorším případě je $\Theta(n^2)$
  - paměťová složitost je $\Theta(n)$
- Heap Sort – třídění haldou
  - časová složitost je ve všech případech $\Theta(n \log n)$
  - v nejhorším a průměrném případě je tedy je optimální (stejně jako Merge Sort)

Samostudium

- pořádkové statistiky – na 2. slidech od prof. Bělohlávka (131–138)