Řazení

Problém řazení

- je daná množina K nad kterou je definované úplné uspořádání
- vstupem problému řazení je posloupnost $A = (k_1, \ldots, k_n)$ prvků z K
- výstupem je posloupnost $A' = (k'_1, \ldots, k'_n)$, která je takovou permutací posloupnosti A, že $\forall i, j, 1 \leq i < j \leq n$, platí $k'_i \leq k'_j$

Stabilní algoritmy a algoritmy *in situ*

- prvky množiny K mohou být strukturované
- řazení podle **klíče**
- řazení se nazývá **stabilní** právě když zachovává vzájemné pořadí položek se stejným klíčem

- prostorová složitost algoritmů řazení je $\Omega(n)$, protože samotná vstupní posloupnost má délku n
- pro přesnější charakterizaci prostorové složitosti jednotlivých algoritmů uvažujeme tzv. **extrasekvenční prostorovou složitost**, do které nezapočítáváme paměť obsazenou vstupní posloupností
- algoritmy, jejichž extrasekvenční složitost je konstantní, se nazývají **in situ** (*in place*)

Přehled

algoritmus	časová složitost v nejhorším případě	časová složitost v průměrném případě
řazení vkládáním	$\Theta(n^2)$	$\Theta(n^2)$
řazení výběrem	$\Theta(n^2)$	$\Theta(n^2)$
řazení sléváním	$\Theta(n \log n)$	$\Theta(n \log n)$
řazení haldou	$\Theta(n \log n)$	$\Theta(n \log n)$
řazení rozdělováním	$\Theta(n^2)$	$\Theta(n \log n)$
řazení počítáním	$\Theta(k+n)$	$\Theta(k+n)$
číslicové řazení	$\Theta(d(n+k))$	$\Theta(d(n+k))$
přihrádkové řazení	$\Theta(n^2)$	$\Theta(n)$

Přehled

algoritmy založené na porovnávání prvků

vkládáním, Insertion sort in situ, stabilní

výběrem, Selection sort in situ, není stabilní

sléváním, Merge sort asymptoticky časově optimální, není in situ, stabilní

haldou, Heapsort asymptoticky časově optimální, in situ, není stabilní

rozdělováním, Quicksort není časově optimální, extrasekvenční složitost a stabilita závisí od implementace (optimálně in situ, existují stabilní implementace), velmi dobrý v praxi (průměrná složitost je $\Theta(n \log n)$)

algoritmy, které získávají informace jinak než porovnáváním prvků

počítáním, Counting sort vstupní prvky jsou z množiny $\{0, \ldots, k\}$

číslicové řazení, Radix sort zobecnění řazení počítáním

přihrádkové řazení, Bucket sort vyžaduje znalost o pravděpodobnostním rozdělení čísel na vstupu

Řazení

Řazení sléváním (Merge sort)

Rozděl posloupnost na dvě stejně velké podposloupnosti

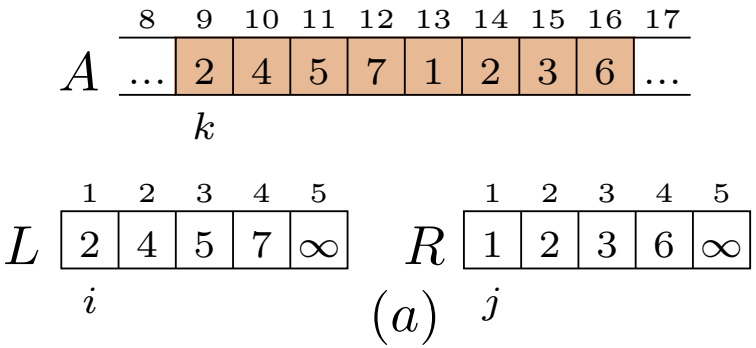
Vyřeš obě podposloupnosti (rekurzivně)

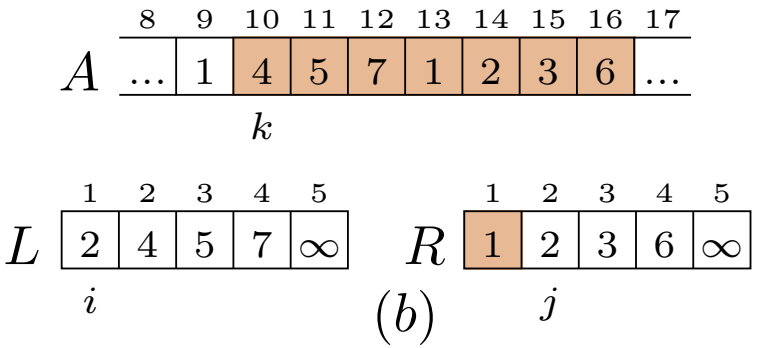
Kombinuj dvě seřazené podposloupnosti do jedné

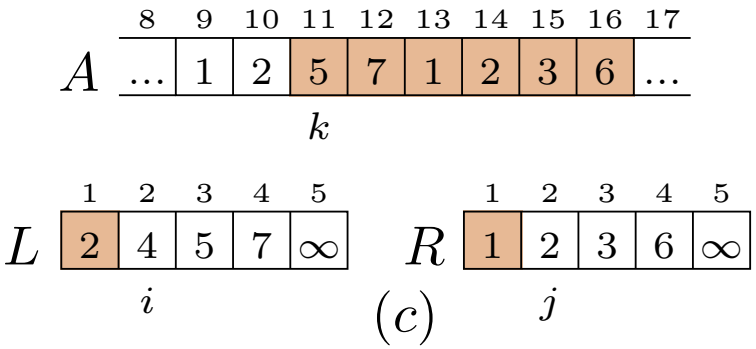
Spojení dvou seřazených posloupností - Merge

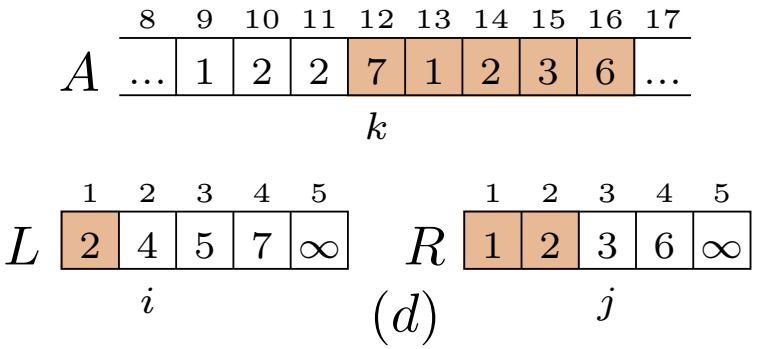
- otázkou je, jak spojit dvě seřazené posloupnosti do jedné, která bude seřazená
- při slévání porovnáváme vedoucí prvky obou posloupností
- menší z porovnávaných prvků přesuneme do výslední posloupnosti

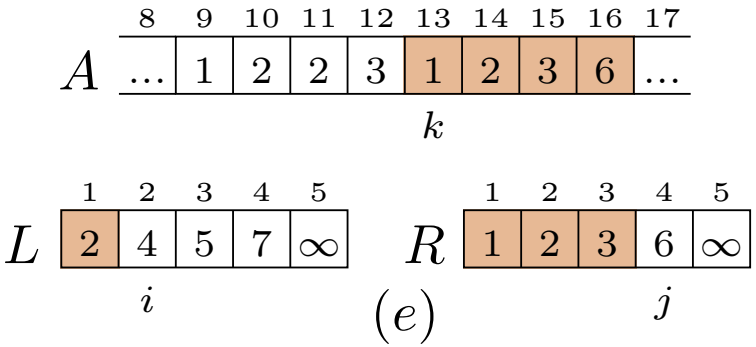
- procedura $\textsc{Merge}$ má 4 parametry
  pole A
  indexy p, q, r takové, že $p \leq q \leq r$
  předpokládáme, že posloupnosti $A[p \ldots q]$ a $A[q+1 \ldots r]$ jsou seřazené
- pro provedení výpočtu je posloupnost $A[p \ldots r]$ seřazená
- pro zjednodušení kódu používáme *sentinel*


A
8 9 10 11 12 13 14 15 16 17
... 2 4 5 7 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j
(a)

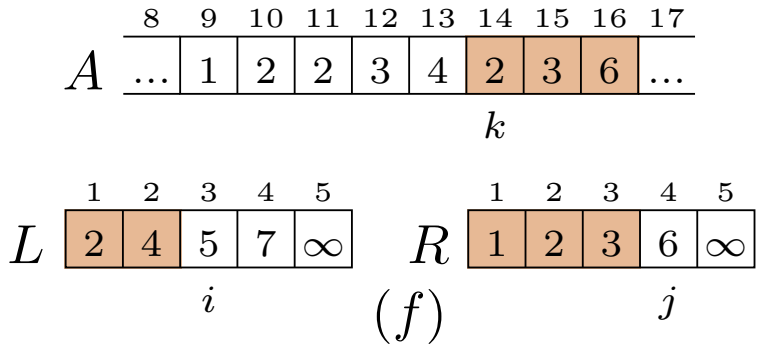

A
8 9 10 11 12 13 14 15 16 17
... 1 4 5 7 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j
(b)


A
8 9 10 11 12 13 14 15 16 17
... 1 2 5 7 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j
(c)

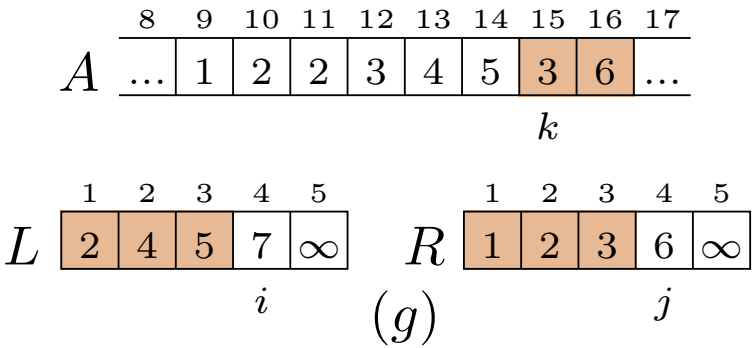

A
8 9 10 11 12 13 14 15 16 17
... 1 2 2 7 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j
(d)


A
8 9 10 11 12 13 14 15 16 17
... 1 2 2 3 1 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

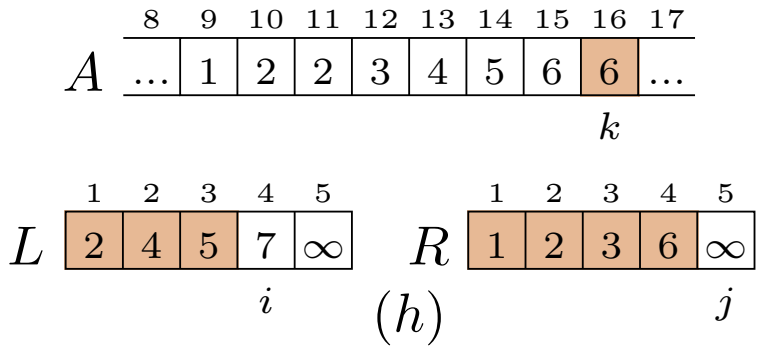
(e)


A
8 9 10 11 12 13 14 15 16 17
... 1 2 2 3 4 2 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

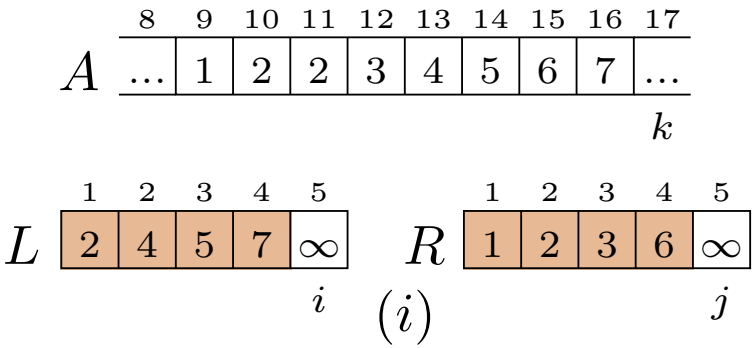
(f)


A
8 9 10 11 12 13 14 15 16 17
... 1 2 2 3 4 5 3 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

(g)


A
8 9 10 11 12 13 14 15 16 17
... 1 2 2 3 4 5 6 6 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

(h)


A
8 9 10 11 12 13 14 15 16 17
... 1 2 2 3 4 5 6 7 ...
k
L
1 2 3 4 5
2 4 5 7 ∞
i
R
1 2 3 4 5
1 2 3 6 ∞
j

(i)

Merge

Procedure Merge(A, p, q, r)

```
1  n_1 ← q − p + 1
2  n_2 ← r − q
3  //nechť L[1 ... n_1 + 1] a R[1 ... n_2 + 1] jsou nové pole
4  for i = 1 to n_1 do L[i] ← A[p + i − 1] od
5  for j = 1 to n_2 do R[j] ← A[q + j] od
6  L[n_1 + 1] ← ∞
7  R[n_2 + 1] ← ∞
8  i ← 1
9  j ← 1
10 for k = p to r do
11     if L[i] ≤ R[j] then A[k] ← L[i]
12                         i ← i + 1
13                    else A[k] ← R[j]
14                         j ← j + 1 fi
15 od
```

Korektnost procedury Merge

Invariant

Na začátku každé iterace cyklu **for** v řádcích 10 - 15 posloupnost $A[p \dots k-1]$ obsahuje $k-p$ nejmenších prvků z $L[1 \dots n_1+1]$ a $R[1 \dots n_2+1]$ a to v pořadí podle velikosti. Navíc, $L[i]$ a $R[j]$ jsou nejmenší prvky mezi těmi prvky ve svých posloupnostech, které ještě nebyly zkopírované do A.

Inicializace Na začátku je $k = p$. Navíc $i = j = 1$ a tedy $L[i]$ a $R[j]$ jsou nejmenší prvky v L a R.

Iterace Předpokládejme, že $L[i] \leq R[j]$. Potom $L[i]$ je nejmenší prvek z těch, které ještě nebyly zkopírované do A. Protože $A[p \dots k-1]$ obsahuje $k-p$ nejmenších prvků, pole $A[p \dots k]$ bude obsahovat $k-p+1$ nejmenších prvků. Zvýšením k a i zaručíme platnost invariantu i po ukončení iterace.

Ukončení Cyklus končí když $k = r+1$. Z platnosti invariantu posloupnost $A[p \dots k-1] = A[p \dots r]$ obsahuje seřazených $k-p = r-p+1$ nejmenších prvků z $L[1 \dots n_1+1]$ a $R[1 \dots n_2+1]$. Pole L a R obsahují v součtu $n_1+n_2+2 = r-p+3$ prvků. Všechny prvky, s výjimkou dvou největších byly, zkopírované do A. Dva největší prvky jsou sentinely.

Složitost procedury Merge

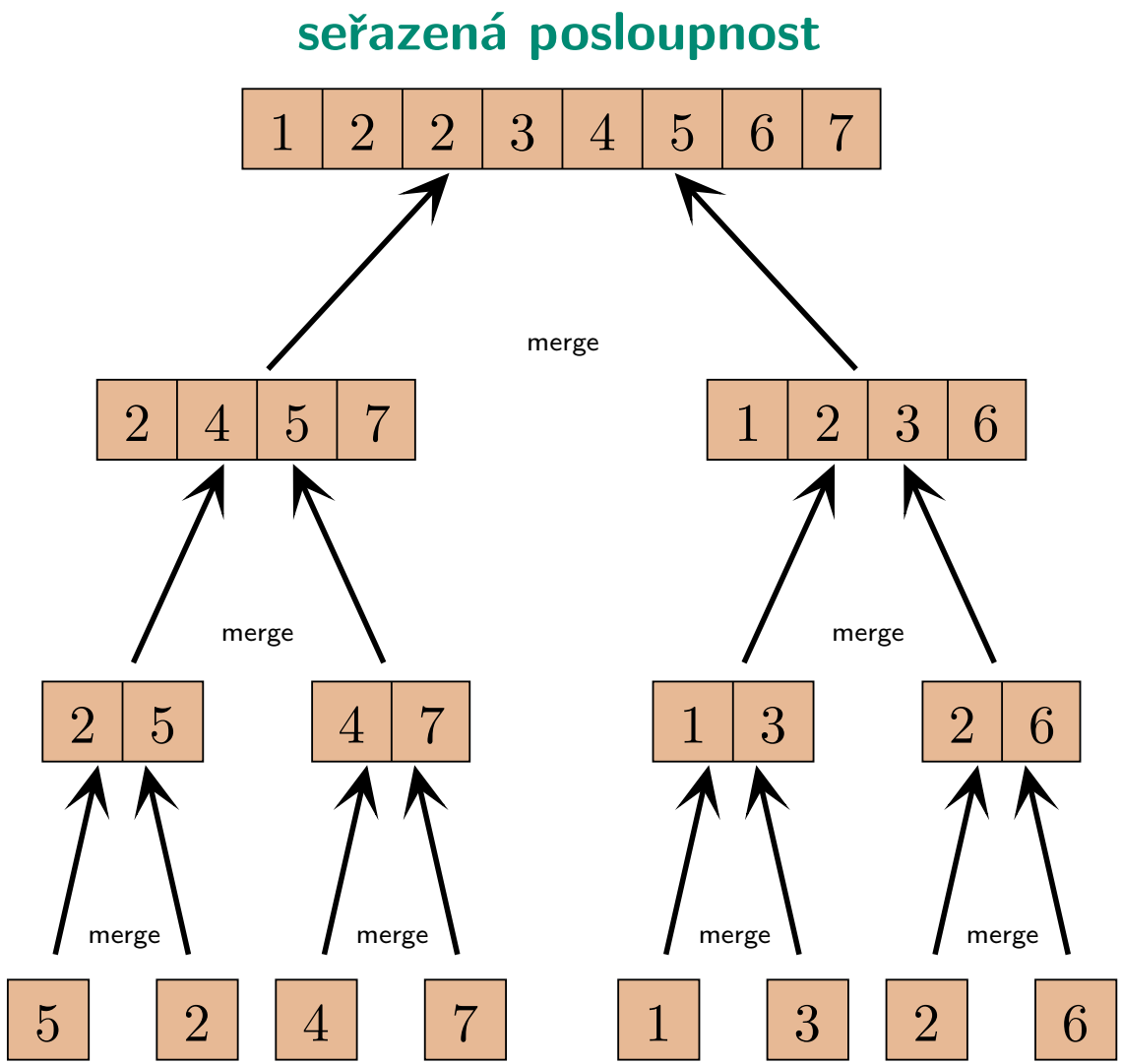
- řádky 1 - 2 a 6 - 9 mají konstantní složitost
- **for** cykly v řádcích 4 a 5 mají v součtu složitost $\Theta(n_1 + n_2) = \Theta(n)$, kde $n = r - p + 1$
- **for** cyklus v řádcích 10 - 15 iteruje n krát, všechny příkazy v řádcích 11 - 14 mají konstantní složitost
- složitost procedury MERGE je $\Theta(n)$

Merge Sort

- využívá proceduru MERGE
- pro seřazení celé posloupnosti voláme MERGE SORT$(A, 1, A.length)$

Procedure Merge Sort(A, p, r)

```
1 if p < r then q ← ⌊(p + r)/2⌋
2              MERGE SORT(A, p, q)
3              MERGE SORT(A, q + 1, r)
4              MERGE(A, p, q, r) fi
```


seřazená posloupnost
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
vstupní posloupnost

Složitost algoritmu Merge Sort

Rozděl rozdělení znamená výpočet indexu, proto má složitost $\Theta(1)$

Vyřeš rekurzívně zpracujeme dvě posloupnosti velikosti $n/2$, časová složitost je $2T(n/2)$

Kombinuj složitost procedury MERGE je $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{ak } n = 1 \\ 2T(n/2) + \Theta(n) & \text{jinak} \end{cases}$$

složitost MERGE SORT je $T(n) = \Theta(n \log n)$

Problém inverzí

motivace

porovnání seznamu preferencí

formulace problému

- je daná posloupnost vzájemné různých čísel $a_1, \ldots, a_n$
- inverzí v posloupnosti je dvojice indexů i, j takových, že $i < j$ a současně $a_i > a_j$
- úkolem je najít všechny inverze v dané posloupnosti čísel

příklad

posloupnost 1, 4, 6, 8, 2, 5 má 5 inverzí

naivní algoritmus

otestuje všechny dvojice indexů, složitost $\mathcal{O}(n^2)$

Problém inverzí - přístup Rozděl a panuj

1. posloupnost rozdělíme na dvě podposloupnosti $a_1, \ldots, a_{\lceil n/2 \rceil}$ a $a_{\lceil n/2 \rceil + 1}, \ldots, a_n$
2. v každé z podposloupností spočítáme inverze
3. spočítáme inverze mezi prvky různých podposloupností

- cílem je navrhnout algoritmus s lepší složitostí než je složitost naivního algoritmu ($\mathcal{O}(n^2)$)
- jestliže chceme, aby časová složitost algoritmu byla $T(n) = \mathcal{O}(n \log n)$, tak musí platit $T(n) \leq 2T(n/2) + \mathcal{O}(n)$, tj. dekompozice a kompozice nesmí překročit lineární složitost
- jak vyřešit kompozici (bod 3) v čase $\mathcal{O}(n)$?

Problém inverzí - kombinuj - pokus 1

otázka jak spočítat inverze (a, b) mezi prvky $a \in A = (a_1, a_2, \ldots, a_k)$ a $b \in B = (b_1, b_2, \ldots, b_l)$?

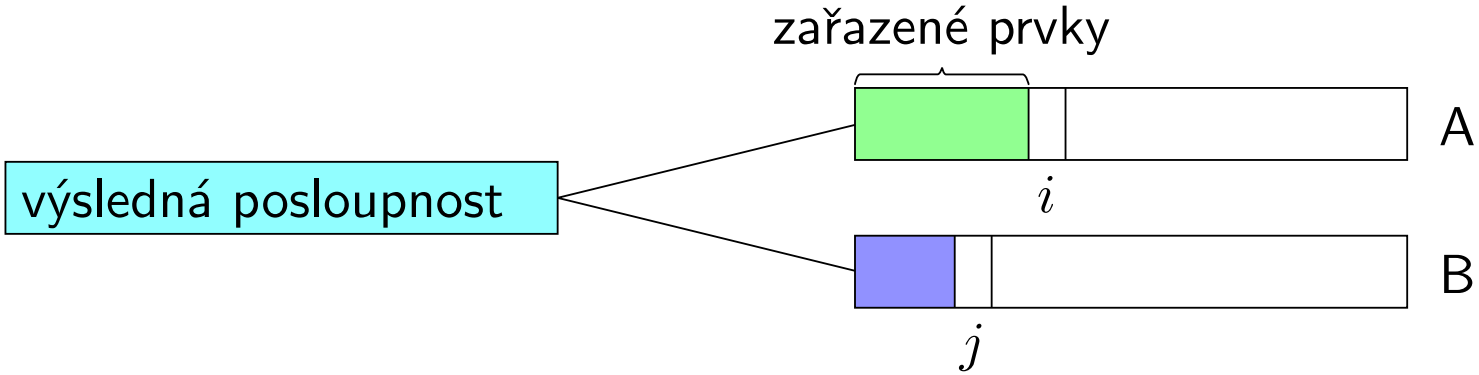
odpověď jednoduché za předpokladu, že A i B jsou seřazené

algoritmus

- seřaď A a B
- pro každý prvek $b \in B$
  binárním vyhledáváním v A urči, kolik prvků v A je větších než b

Problém inverzí - kombinuj

- $A = (a_1, a_2, \ldots, a_k)$, $B = (b_1, b_2, \ldots, b_l)$
- předpokládáme, že
  - prvky v obou posloupnostech jsou seřazeny vzestupně
  - všechny prvky posloupnosti A mají ve vstupné posloupnosti menší index než prvky posloupnosti B
- postupujeme stejně jako v proceduře MERGE
- prvky $a_1, \ldots, a_{i-1}$ a $b_1, \ldots, b_{j-1}$ jsou již zařazené
- porovnáváme prvek a_i s prvkem b_j
  - menší z porovnávaných prvků zařadíme do výstupné posloupnosti
  - jestliže $\mathbf{a_i < b_j}$, tak a_i není v inverzi se žádným z prvků $b_j, b_{j+1}, \ldots, b_l$
  - jestliže $\mathbf{a_i > b_j}$, tak b_j je v inverzi se všemi prvky $a_i, \ldots, a_k$ a proto k počtu inverzí připočteme $k - i + 1$



- inverze mezi prvky zařazenými do výsledné posloupnosti jsou již započítané
- jestliže $\mathbf{a_i < b_j}$, tak do výsledné posloupnosti přesuneme a_i

  $a_i < b_j < b_{j+1} < b_{j+2} < \ldots$
  a_i není v inverzi se žádným z $b_j, b_{j+1}, b_{j+2} \ldots$
- jestliže $\mathbf{a_i > b_j}$, tak do výsledné posloupnosti přesuneme b_j

  $b_j < a_i < a_{i+1} < a_{j+2} < \ldots$
  b_j je v inverzi s každým z $a_i, a_{i+1}, a_{i+2} \ldots$

Algoritmus

Merge_and_Count(A, B)

```
1  i ← 1; j ← 1
2  //i, j jsou indexy prvních nezařazených prvků z A resp. B
3  Count ← 0
4  //Count je počet nalezených inverzí
5  while seznamy A, B jsou neprázdné do
6        porovnej a_i a b_j
7        menší z prvků zařaď do výsledného seznamu
8        if b_j < a_i then zvyš Count o počet nezařazených prvků z A fi
9        zvyš index i resp. j od
10 if jeden seznam je prázdný
11   then zařaď zbývající prvky do výsledného seznamu fi
12 return Count a výsledný seznam
```

Algoritmus

Sort_and_Count(L)

```
1 if length(L) = 1
2   then r ← 0
3    else A ← levá polovina L
4         B ← pravá polovina L
5         (r_A, A) ← Sort_and_Count(A)
6         (r_B, B) ← Sort_and_Count(B)
7         (r, L) ← Merge_and_Count(A, B)
8         r ← r + r_A + r_B fi
9 return (r, L)
```

složitost algoritmu je $T(n) = 2T(n/2) + \Theta(n)$ a proto $T(n) = \mathcal{O}(n \log n)$

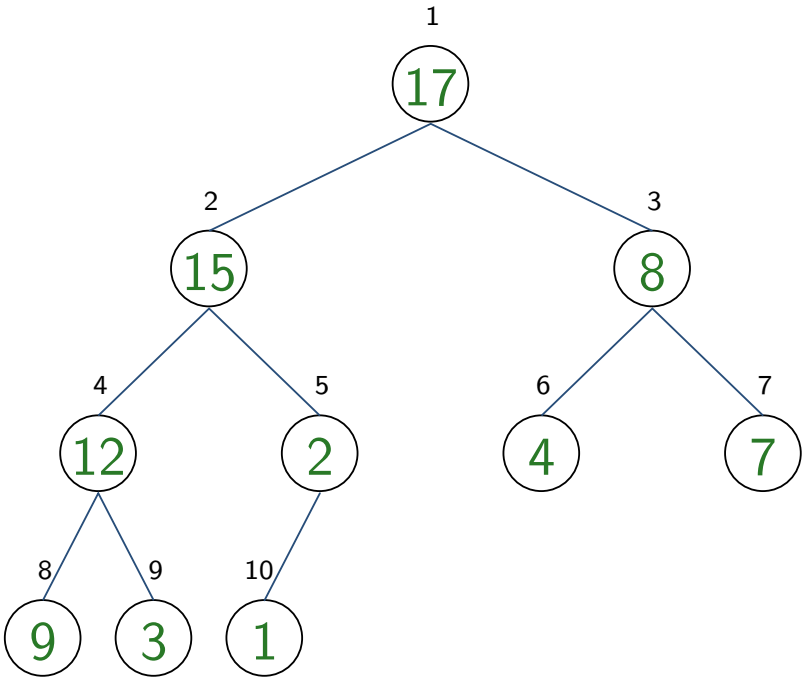
Řazení

Řazení haldou - Heapsort

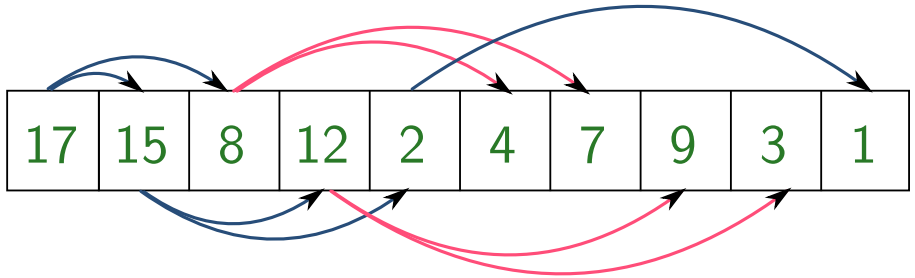
prvky posloupnosti vložíme do (binární) haldy

- halda je datová struktura
- pole
- prvky pole odpovídají vrcholům binárního stromu
- binární strom je **úplný** (vrcholy na všech úrovních s výjimkou předposlední mají právě dva následníky) a **zleva zarovnaný** (nejhlubší úroveň je zaplněná zleva doprava)
- pole reprezentující haldu má dva atributy
  - $\mathbf{A.length}$ počet prvků v poli
  - $\mathbf{A.heap\_size}$ počet prvků haldy uložených v poli
  - $\mathbf{A[1 \ldots A.heap\_size]}$ obsahuje prvky haldy

Halda



(a)



(b)

Halda

- kořen haldy je uložený v $A[1]$
- pro daný index i vypočteme indexy následníků a předchůdce vrcholu $A[i]$ předpisem

$\textsc{Parent}(i)$

  return $\lfloor i/2 \rfloor$

$\textsc{Left}(i)$

  return $2i$

$\textsc{Right}(i)$

  return $2i + 1$

Halda - vlastnost haldy

- rozlišujeme minimovou a maximovou haldu
- v obou případech prvky (čísla) uložené v haldě musí splňovat **vlastnost haldy**

- pro každý vrchol maximové haldy platí, že **hodnota uložená ve vrcholu je větší nebo rovna než hodnoty uložené v jeho následnících**
- největší prvek haldy je uložený v její kořeni

- pro každý vrchol minimové haldy platí symetricky $A[\textsc{Parent}(i)] \leq A[i]$
- výběr mezi minimovou a maximovou haldou závisí od kontextu: řazení od největšího resp. nejmenšího prvku, prioritní fronty, . . .)

Operace nad (maximovou) haldou

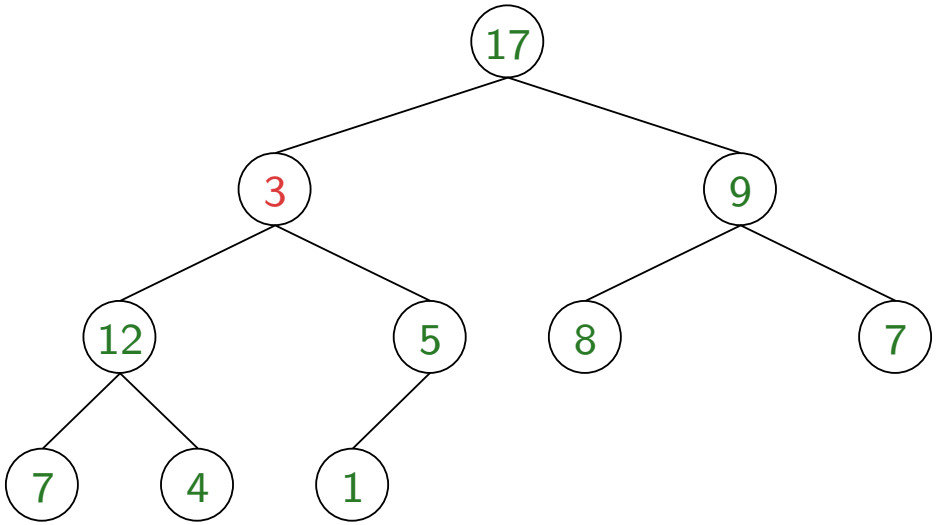
- MAX_HEAPIFY garantuje platnost vlastnosti haldy, složitost $\mathcal{O}(\log n)$
- BUILD_MAX_HEAP vybuduje z pole haldu, složitost $\Theta(n)$
- HEAPSORT seřadí prvky pole, složitost $\mathcal{O}(n \log n)$

- procedury MAX_HEAP_INSERT, HEAP_EXTRACT_MAX , HEAP_INCREASE_KEY a HEAP_MAXIMUM využijeme pro implementaci prioritní fronty

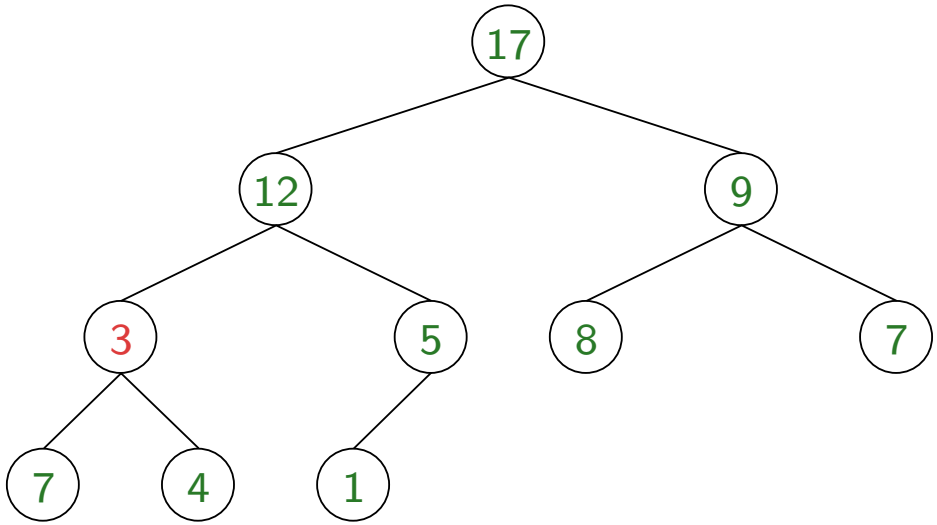
Obnovení vlastnosti haldy

- procedura $\textsc{Max\_Heapify}(A, i)$ předpokládá, že
  - binární stromy s kořeny $\textsc{Left}(i)$ a $\textsc{Right}(i)$ jsou (maximové) haldy a že
  - prvek $A[i]$ může být menší než jeho následníci, tj. nemusí splňovat vlastnost haldy
- procedura modifikuje A tak, že po její provedení strom s kořenem $A[i]$ tvoří haldu
- úprava je založena na přesunu prvku $A[i]$ směrem dolů

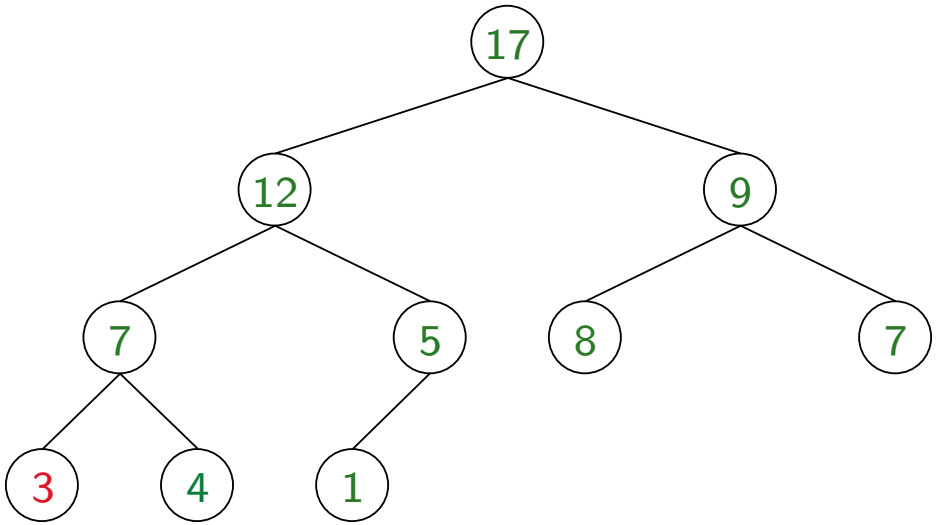
Max_Heapify



(a)



(b)



(c)

Max_Heapify(A, i)

```
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ A.heap_size ∧ A[l] > A[i]
4     then largest ← l
5      else largest ← i fi
6  if r ≤ A.heap_size ∧ A[r] > A[largest]
7     then largest ← r fi
8  if largest ≠ i
9     then swap A[i] a A[largest]
10          MAX_HEAPIFY(A, largest) fi
```

Max_Heapify - složitost

- podstrom s kořenem $A[i]$ má n vrcholů
- procedura je rekurzivní
- složitost dekompozice (tj. výpočet $largest$) a kompozice je konstantní
- rekurzivní volání pro podstrom, který má maximálně $2n/3$ vrcholů
- $T(n) \leq T(2n/3) + \Theta(1)$
- $T(n) = \Theta(\log n)$
- alternativně můžeme složitost operace vyjádřit jako $\mathcal{O}(h)$, kde h je výška stromu (*výška listu je 0*)

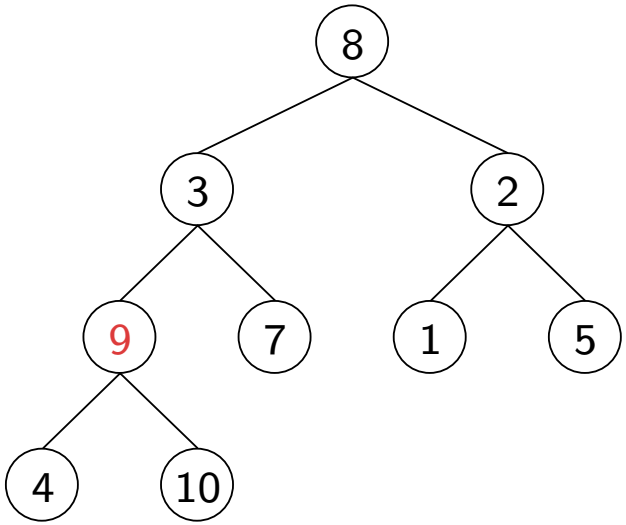
Vytvoření haldy

- využitím procedury MAX_HEAPIFY zkonvertujeme pole $A[1 \dots n]$ na maximovou haldu
- prvky $A[\lfloor n/2 \rfloor + 1], A[\lfloor n/2 \rfloor + 2] \dots A[n]$ jsou listy stromu a proto každý tvoří haldu s 1 vrcholem
- proceduru MAX_HEAPIFY aplikujeme na zbylé prvky pole v pořadí odspodu směrem nahoru a na dané úrovni směrem zprava doleva

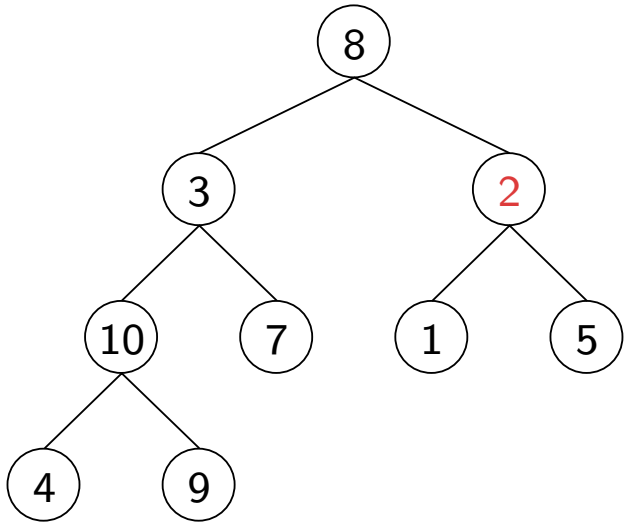
Build_Max_Heap(A)

```
1 A.heap_size ← A.length
2 for i = ⌊A.length/2⌋ downto 1 do
3     MAX_HEAPIFY(A, i) od
```

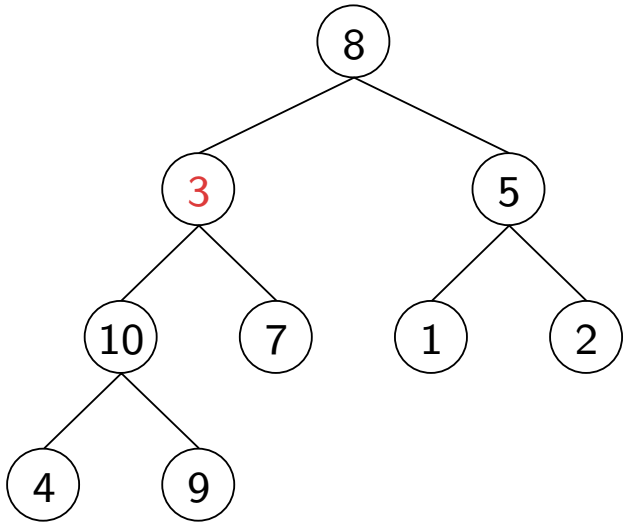
Vytvoření haldy



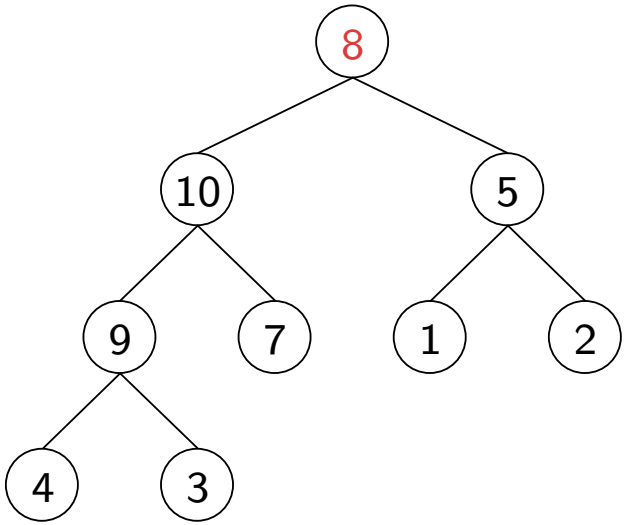
(a)



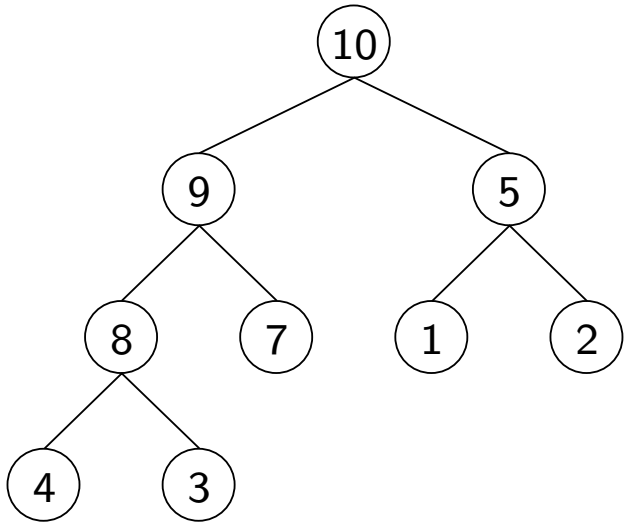
(b)



(c)



(d)



(e)

Build_Max_Heap - korektnost

Invariant cyklu

Na začátku každé iterace **for** cyklu je každý z vrcholů $A[i+1], A[i+2], \ldots, A[n]$ kořenem maximové haldy.

Inicializace na začátku je $i = \lfloor n/2 \rfloor$, vrcholy $A[\lfloor n/2 \rfloor + 1]$, $A[\lfloor n/2 \rfloor + 2]$, $\ldots$,$A[n]$ jsou listy a jsou tedy kořeny (triviální) haldy

Iterace levý a pravý podstrom vrcholu $A[i]$ jsou maximové haldy (platí pro ně invariant), vlastnost haldy může být porušena jedině hodnotou $A[i]$; procedura $\textsc{Max\_Heapify}(A, i)$ vybuduje maximovou haldu s kořenem i

Ukončení cyklus skončí když $i = 0$ a z platnosti invariantu plyne, že A je maximová haldu

Build_Max_Heap - složitost

- složitost procedury MAX_HEAPIFY je $\mathcal{O}(h)$, kde h je výška stromu, na který se procedura aplikuje
- počet podstromů výšky h je nejvýše $\left\lceil \frac{n}{2^{h+1}} \right\rceil$
- kořen má výšku $\lfloor \log n \rfloor$
- celková složitost je proto

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = \mathcal{O}\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = \mathcal{O}(n)$$

při zjednodušování výrazu jsme využili rovnost

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

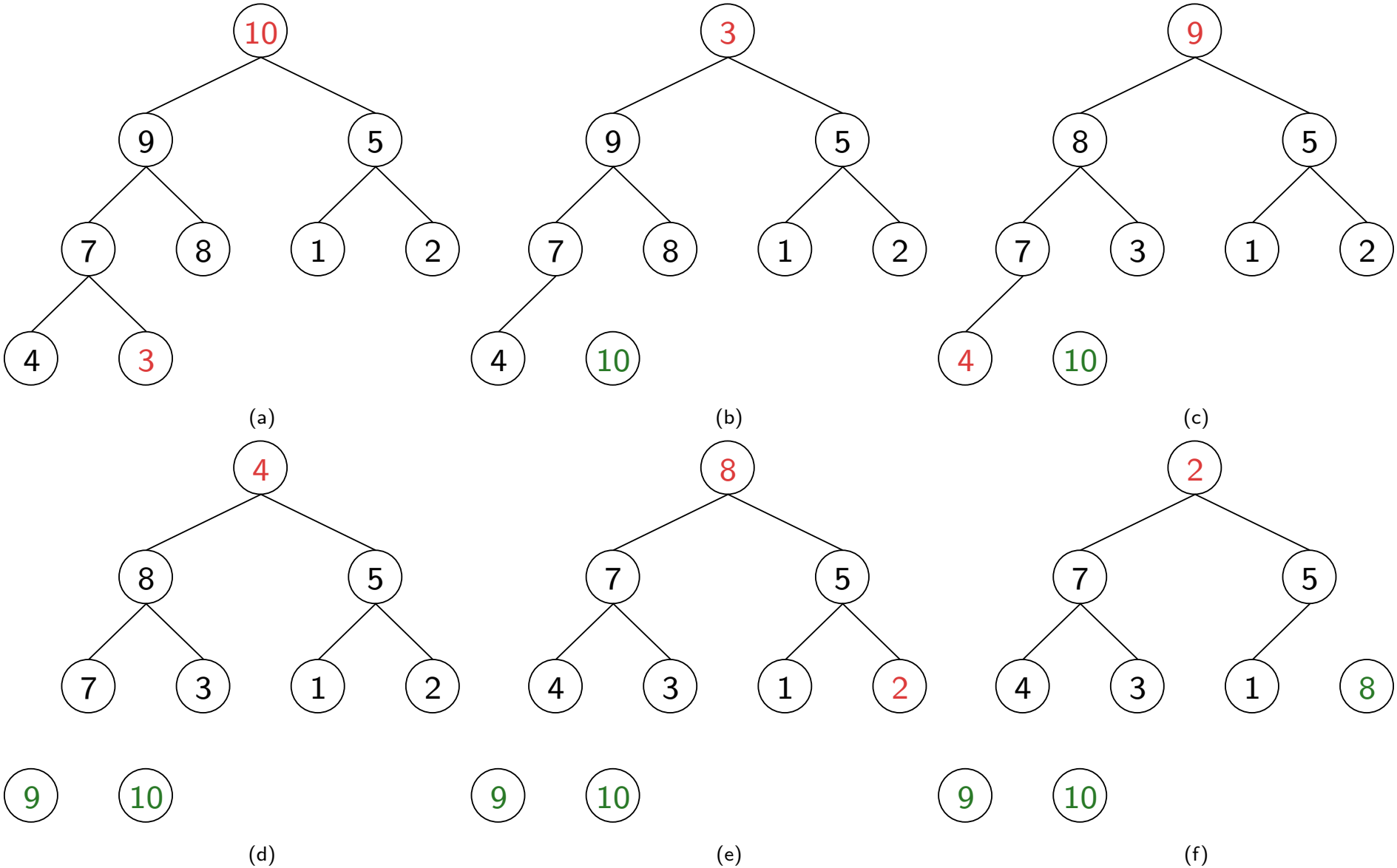
Algoritmus řazení haldou, Heapsort

- použitím procedury $\textsc{Build\_Max\_Heap}$ vybudujeme maximovou haldu nad polem $A[1 \ldots n]$, kde $n = A.length$
- maximální prvek pole A je uložený v kořeni $A[1]$ a proto ho můžeme přesunout na jeho finální pozici $A[n]$ (vyměníme prvky $A[1]$ a $A[n]$)
- prvek, který jsme přesunuli do kořene, může porušit vlastnost haldy a pro obnovení vlastnosti haldy použijeme $\textsc{Max\_Heapify}(A, 1)$
- celý proces opakujeme pro haldu velikosti $n - 1$

Heapsort(A)

```
1 Build_Max_Heap(A)
2 for i = A.length downto 2 do vyměň A[1] a A[i]
3     A.heap_size ← A.heap_size − 1
4     Max_Heapify(A, 1) od
```

Heapsort


10
9
5
7
8
1
2
4
3
(a)
3
9
5
7
8
1
2
4
10
(b)
9
8
5
7
3
1
2
4
10
(c)
4
8
5
7
3
1
2
9
10
(d)
8
7
5
4
3
1
2
9
10
(e)
2
7
5
4
3
1
8
9
10
(f)

Heapsort - složitost

- procedura BUILD_MAX_HEAP má složitost $\mathcal{O}(n)$
- každé z $n-1$ volaní procedury MAX_HEAPIFY má složitost $\mathcal{O}(\log n)$
- algoritmus HEAPSORT má složitost $\mathcal{O}(n \log n)$

Optimalizace a varianty

- strom vyšší arity
- ve vrcholu stromu uložených několik hodnot

- **budování haldy výměnou zdola nahoru** halda je na začátku prázdná a postupně do ní vkládáme prvky vstupní posloupnosti; prvek vložíme na poslední místo (jako list) a v případě porušení vlastnosti haldy ho (rekurzivně) zaměníme s jeho rodičem; časová složitost vybudování haldy je $\Theta(n \log n)$
- **bottom - up heapsort** optimalizuje etapu seřazování prvků; maximální prvek z kořene si vymění místo s posledním prvkem haldy a pro obnovení vlastnosti haldy výměnami se postupuje zdola nahoru
- **Smoothsort** (Edsger Dijkstra) - stejná asymptotická složitost, lepší chování pro vstupní posloupnosti, které jsou téměř uspořádané

Prioritní fronty

- datová struktura pro reprezentaci množiny, nad prvky množiny je definováno uspořádání
- umožňuje efektivní realizaci operací:
  $\textsc{Insert}(S, x)$ vloží prvek x do množiny S
  $\textsc{Maximum}(S)$ vrátí největší prvek množiny S
  $\textsc{Extract\_Max}(S)$ odstraní z množiny S největší prvek
  $\textsc{Increase\_Key}(S, x, k)$ nahradí prvek x prvkem k za předpokladu, že $k \geq x$
- alternativně můžeme definovat prioritní frontu vůči minimálnímu prvku
- **prioritní frontu implementujeme jako maximovou haldu**

Maximum a Extract_Max

- prvky množiny S tvoří haldu A
- maximální prvek haldy je v jejím kořeni; jeho nalezení má konstantní složitost

Heap_Maximum(A)

```
1 return A[1]
```

- odstranění maximálního prvku se implementuje stejně jako v algoritmu řazení
- složitost operace je $\mathcal{O}(\log n)$

Heap_Extract_Max(A)

```
1 if A.heap_size < 1 then return  prázdná fronta fi
2 max ← A[1]
3 A[1] ← A[A.heap_size]
4 A.heap_size ← A.heap_size − 1
5 MAX_HEAPIFY(A, 1)
6 return max
```

Increase_Key

- procedura HEAP_INCREASE_KEY implementuje operaci INCREASE_KEY
- index i identifikuje prvek, který má být operací nahrazen (navýšen)
- nejdříve změníme hodnotu $A[i]$ na novou hodnotu key a potom obnovíme vlastnost haldy

Heap_Increase_Key(A, i, key)

```
1 if key < A[i] then return  nová hodnota je menší než původní fi
2 A[i] ← key
3 while i > 1 ∧ A[PARENT(i)] < A[i] do
4         vyměň A[i] a A[PARENT(i)]
5         i ← PARENT(i) od
```

složitost: $\mathcal{O}(\log n)$

Insert

- procedura MAX_HEAP_INSERT implementuje operaci INSERT
- na konec pole vložíme nový prvek, který je menší než všechny ostatní prvky, symbolicky ho označujeme $-\infty$
- zvýšíme hodnotu vloženého prvku na hodnotu prvku, který chceme vložit do fronty

Max_Heap_Insert(A, key)

1 $A.heap\_size \leftarrow A.heap\_size + 1$
2 $A[A.heap\_size] \leftarrow -\infty$
3 HEAP_INCREASE_KEY$(A, A.heap_size, key)$

složitost: $\mathcal{O}(\log n)$

Řazení

Řazení rozdělováním - Quicksort

Rozděl posloupnost $A[p \dots r]$ na dvě podposloupnosti $A[p \dots q-1]$ a $A[q \dots r]$ tak, aby všechny prvky v $A[p \dots q-1]$ byly menší nejvýše rovné prvkům v $A[q \dots r]$

Vyřeš obě posloupnosti (rekurzivně) seřaď

Kombinuj protože obě podposloupnosti jsou seřazené, není nutný žádný další výpočet

Mergesort

Rozděl posloupnost na dvě posloupnosti poloviční velikosti.

Vyřeš obě podposloupnosti (rekurzivně) seřaď

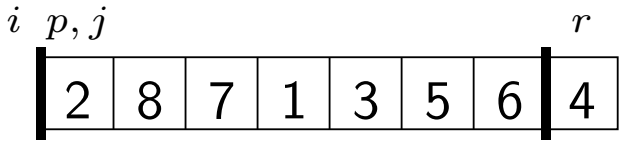
Kombinuj spoj dvě seřazené podposloupnosti do jedné

Quicksort

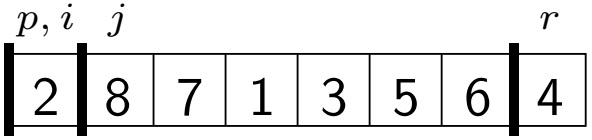
- hlavní částí algoritmu je rozdělování posloupnosti do dvou posloupností požadovaných vlastností
- při rozdělování využíváme **pivota**
- každý prvek posloupnosti porovnáváme s pivotem
- podposloupnosti prvků menších / větších než pivot

Quicksort

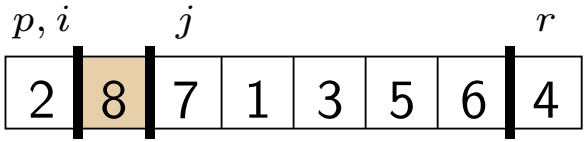
(a)



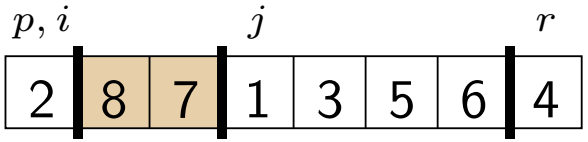
(b)



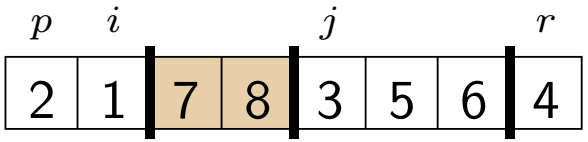
(c)



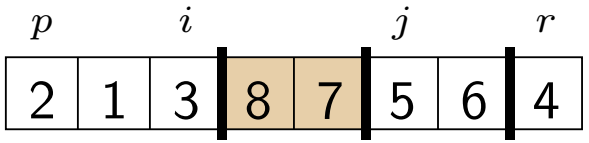
(d)



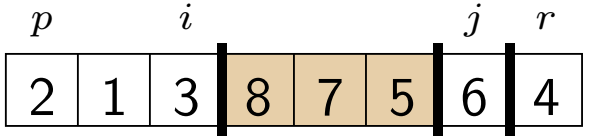
(e)



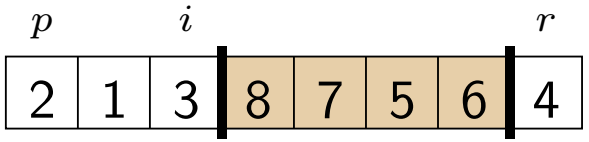
(f)



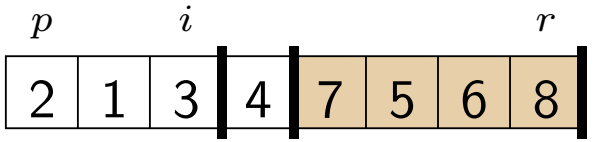
(g)



(h)



(i)



Quicksort(A, p, r)

```
1 if p < r
2   then q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r) fi
```

Partition(A, p, r)

```
1 pivot ← A[r]
2 i ← p − 1
3 for j = p to r do
4     if A[j] ≤ pivot then  i ← i + 1
5                           vyměň A[i] a A[j] fi od
6 return i
```

Korektnost

chceme dokázat, že procedura PARTITION vrátí index i takový, že

- $A[i] = pivot$
- pro $p \leq k \leq i$ platí $A[k] \leq A[i]$
- pro $i < k \leq r$ platí $A[k] > A[i]$

Invariant cyklu

na začátku každé iterace **for** cyklu v řádcích 3 - 6 platí pro každý index k

1. jestliže $p \leq k \leq i$, tak $A[k] \leq pivot$
2. jestliže $i + 1 \leq k \leq j - 1$, tak $A[k] > pivot$

Inicializace

iniciální přiřazení je $pivot \leftarrow A[r]$, $i \leftarrow p - 1$ a $j \leftarrow p$
invariant (triviálně) platí

Korektnost

Invariant cyklu

na začátku každé iterace **for** cyklu v řádcích 3 - 6 platí pro každý index k

1. jestliže $p \leq k \leq i$, tak $A[k] \leq pivot$
2. jestliže $i + 1 \leq k \leq j - 1$, tak $A[k] > pivot$

Iterace

$A[j] > pivot$ - efektem iterace cyklu je zvýšení hodnoty j o 1; invariant platí

$A[j] \leq pivot$ - efektem iterace cyklu je zvýšení hodnoty i a výměna $A[i]$ s $A[j]$, to garantuje zachování platnosti podmínky 1
zachování platnosti podmínky 2 garantuje fakt, že jsme do $A[j-1]$ přesunuli prvek větší než $pivot$

Korektnost

Invariant cyklu

na začátku každé iterace **for** cyklu v řádcích 3 - 6 platí pro každý index k

1. jestliže $p \leq k \leq i$, tak $A[k] \leq pivot$
2. jestliže $i + 1 \leq k \leq j - 1$, tak $A[k] > pivot$

Ukončení

výpočet končí když $j = r + 1$, což spolu s faktem, že po posledním provedení iterace platí invariant, garantuje že pro $p \leq k \leq i$ platí $A[k] \leq A[i]$ a pro $i < k \leq r$ platí $A[k] > A[i]$

v poslední iteraci je $j = p$, $A[j] = pivot \leq pivot$, provede se výměna $A[i]$ s $A[j]$, co garantuje, že po ukončení výpočtu cyklu platí $A[i] = pivot$

Složitost

složitost v nejhorším případě

např. pro vstupní posloupnost, která je již seřazená, nebo která obsahuje stejné prvky

$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$

$T(n) = \Theta(n^2)$

složitost v nejlepším případě

nastává, když při každém rekurzivním volání rozdělí pivot posloupnost na dvě stejně velké podposloupnosti

$T(n) = 2T(n/2) + \Theta(n)$

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

průměrná složitost

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

Alternativní postup rozdělování I

- postupujeme od obou konců posloupnosti až do chvíle, než jsou detekovány dva prvky, které jsou vůči sobě v opačném pořadí; prvky si vymění svou pozici
- při tomto postupu se udělá průměrně 3 krát méně výměn
- algoritmus není stabilní

Hoare Partition(A, p, r)

```
1 x ← A[p]
2 i ← p − 1
3 j ← r + 1
4 while true do
5         repeat j ← j − 1 until A[j] ≤ x od
6         repeat i ← i + 1 until A[i] ≥ x od
7         if i < j then  swapA[i] a A[j] else return j fi od
```

Quicksort(A, p, r)

```
1 if p < r then q ← Hoare Partition(A, p, r)
2                 Quicksort(A, p, q)
3                 Quicksort(A, q + 1, r) fi
```

Alternativní postup rozdělování II

- obě uvedené schémata se chovají špatně v případě, že ve vstupní posloupnosti se prvky opakují

- rozdělovací schéma, které řeší posloupnosti s opakujícími se prvky
- při rozdělování se hledají prvky menší než pivot a větší než pivot
- prvky stejné jako pivot jsou již na své pozici
- prvky menší (větší) než pivot se seřadí rekurzivně

Iterativní verze Quicksortu

algoritmus ve tvaru *tail* rekurze

Tail Recursive Quicksort(A, p, r)

```
1 while p < r do
2         q ← Partition(A, p, r)
3         Tail Recursive Quicksort(A, p, q − 1)
4         p ← q + 1 od
```

Iterative Quicksort(A, p, r)

```
1 stack = [ ]
2 stack.push(p, r)
3 while stack do
4         pos = stack.pop()
5         p, r = pos[1], pos[2]
6         q ← Partition(A, p, r)
7         if q − 1 > p then stack.push((p, q − 1)) fi
8         if q + 1 < r then stack.push((q + 1, r)) fi
9 od
```

Složitost problému řazení

složitost řadících algoritmů založených na vzájemném porovnávání prvků posloupnosti je $\Omega(n \log n)$

1. búno vstupní posloupnost $a_1, \ldots, a_n$ obsahuje vzájemně různé prvky
2. každé porovnání určí větší ze dvou prvků
3. výpočet algoritmu můžeme popsat *rozhodovací stromem*, jehož vnitřní vrcholy jsou označeny y porovnávaných prvků a mají dva syny odpovídající vztahu $<$ a $>$
4. výpočet na konkrétním vstupu představuje cestu v rozhodovacím stromě z kořene do listu; jeho složitost je úměrná délce cesty
5. každý list jednoznačně určuje seřazení $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ vstupních prvků
6. algoritmus musí mít možnost vypočítat každou možnou permutaci vstupních prvků
7. počet různých permutací je $n!$
8. strom musí mít alespoň $n!$ listů $\Rightarrow$ má hloubku alespoň $\log(n!) = \Omega(n \log n)$

Řazení

Řazení počítáním - Counting Sort

předpokládá, že vstupní posloupnost obsahuje celá čísla z intervalu $0 \ldots k$, kde k je nějaké pevně dané přirozené číslo

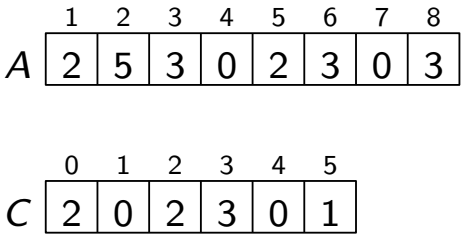
jestliže $k = \mathcal{O}(n)$, tak složitost řazení počítáním je $\Theta(n)$

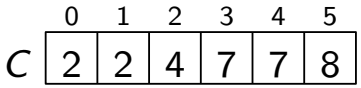
Counting sort

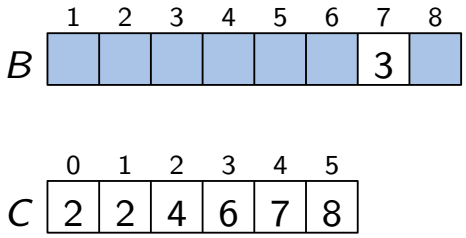
- vstupní posloupnost $A[1 \ldots n]$
- pole $B[1 \ldots n]$ obsahuje seřazenou posloupnost
- pole $C[0 \ldots k]$ se využívá v průběhu výpočtu

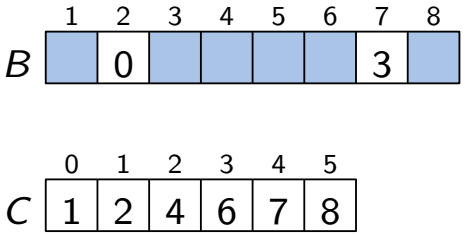
- pro každou hodnotu $i = 0, 1, \ldots, k$ spočítáme, kolik je ve vstupní posloupnosti čísel i, výsledný počet uložíme do $C[i]$
- pro každou hodnotu $i = 0, 1, \ldots, k$ spočítáme, kolik je ve vstupní posloupnosti čísel **menších nebo rovných** i, využijeme k tomu hodnoty napočítané v předcházejícím kroku a výsledný počet uložíme opět do $C[i]$
- procházíme vstupní posloupnost od konce a každé číslo uložíme do B přímo na jeho pozici, která je určená počtem menších nebo rovných čísel; hodnoty v C průběžně aktualizujeme

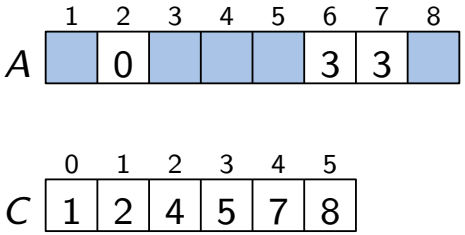
Counting sort

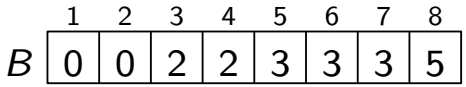












Counting_Sort(A, B, k)

```
1  //inicializace C[0...k]
2  for i = 0 to k do
3      C[i] ← 0 od
4  for j = 1 to A.length do
5      C[A[j]] ← C[A[j]] + 1 od
6  //C[i] obsahuje počet čísel rovných i
7  for i = 1 to k do
8      C[i] ← C[i] + C[i − 1] od
9  //C[i] obsahuje počet čísel menších nebo rovných i
10 for j = A.length downto 1 do
11     B[C[A[j]]] ← A[j]
12     C[A[j]] ← C[A[j]] − 1 od
```

Časová složitost

- cyklus na řádcích 2 - 3 (inicializace C) – složitost $\Theta(k)$
- cyklus na řádcích 4 - 5 (počet čísel $= i$) – složitost $\Theta(n)$
- cyklus na řádcích 7 - 8 (počet čísel $\leq i$) – složitost $\Theta(k)$
- cyklus na řádcích 10 - 12 (přesun z A do B) – složitost $\Theta(n)$
- celková složitost $\Theta(k + n)$

- v praxi používaný pro $k = \mathcal{O}(n)$
- stabilní algoritmus: prvky se stejnou hodnotou se ve výstupní posloupnosti vyskytují ve stejném pořadí jako ve vstupní posloupnosti *(důležité např. pro radix sort)*

Varianty a optimalizace

- v případě, že vstupní posloupnost obsahuje pouze čísla (a ne složitější datové objekty s klíčem), tak druhý cyklus algoritmu je možné vynechat a zapisovat do pole B přímo čísla
- algoritmus se dá využít k odstraňování duplicitních klíčů (pole C nahradíme bitovým polem)
- umožňuje efektivní paralelizaci (vstupní posloupnost rozdělíme na stejně velké podposloupnosti a pro každou z nich počítáme frekvence výskytu paralelně)
- extrasekvenční složitost algoritmu je $\mathcal{O}(n + k)$

Číslicové řazení - Radix Sort

- řazení čísel podle číslic na jednotlivých bitech
- postup zleva doprava (most significant digit, MSD) - používá se např. pro lexikografické uspořádání
- postup zprava doleva (least significant digit, LSD), stabilní řazení
- dá se použít i pro řazení položek, které nemají číselný charakter
- používá se např. když potřebujeme seřadit položky vzhledem k různým klíčům

Radix_Sort(A, d)

```
1 for i = 1 to d do
2     použij stabilní řazení a seřaď položky podle ite číslice
3 od
```

Radix Sort

Lema 1

Danou posloupnost n čísel s d číslicemi, přičemž číslice můžou nabývat k různých hodnot, seřadí RADIX_SORT *korektně v čase $\Theta(d(n+k))$ za předpokladu, že stabilní řazení, které využívá, má složitost $\Theta(n+k)$.*

- složitost je garantovaná např. při použití algoritmu Counting sort
- jestliže d je konstanta a $k = \mathcal{O}(n)$, pak časová složitost číslicového řazení je lineární

Varianty

- řazení binárních čísel (*lze zobecnit pro libovolnou číselnou soustavu*)
- nechť každé číslo má b bitů, zvolíme $r \leq b$
- číslo rozdělíme na $\lceil b/r \rceil$ skupin po r bitech
- každou skupinu chápeme jako číslo z intervalu 0 až $2^r - 1$
- při řazení postupujeme po skupinách, použijeme Counting sort pro $k = 2^r - 1$

Lema 2

Danou posloupnost n binárních b bitových čísel RADIX_SORT *korektně seřadí v čase $\Theta((b/r)(n + 2^r))$ za předpokladu, že stabilní řazení, které využívá, má složitost $\Theta(n + k)$ pro čísla z intervalu 0 až k.*

otázka vhodné volby parametru r pro dané n a b závisí od poměru veličin n a b

$\lfloor b < \log n \rfloor$ pro $r \leq b$ platí $(n + 2^r) = \Theta(n)$, optimální je proto volba $r = b$ pro kterou je celková složitost číslicového řazení $(b/b)(n + 2^b) = \Theta(n)$

$\lfloor b \geq \log n \rfloor$
- pro $r = \lfloor \log n \rfloor$ je složitost je $\Theta(bn/\log n)$
- pro $r > \lfloor \log n \rfloor$ je složitost je $\Omega(bn/\log n)$
- pro $r < \lfloor \log n \rfloor$ hodnota výrazu (b/r) klesá a hodnota výrazu $n + 2^r$ zůstává $\Theta(n)$

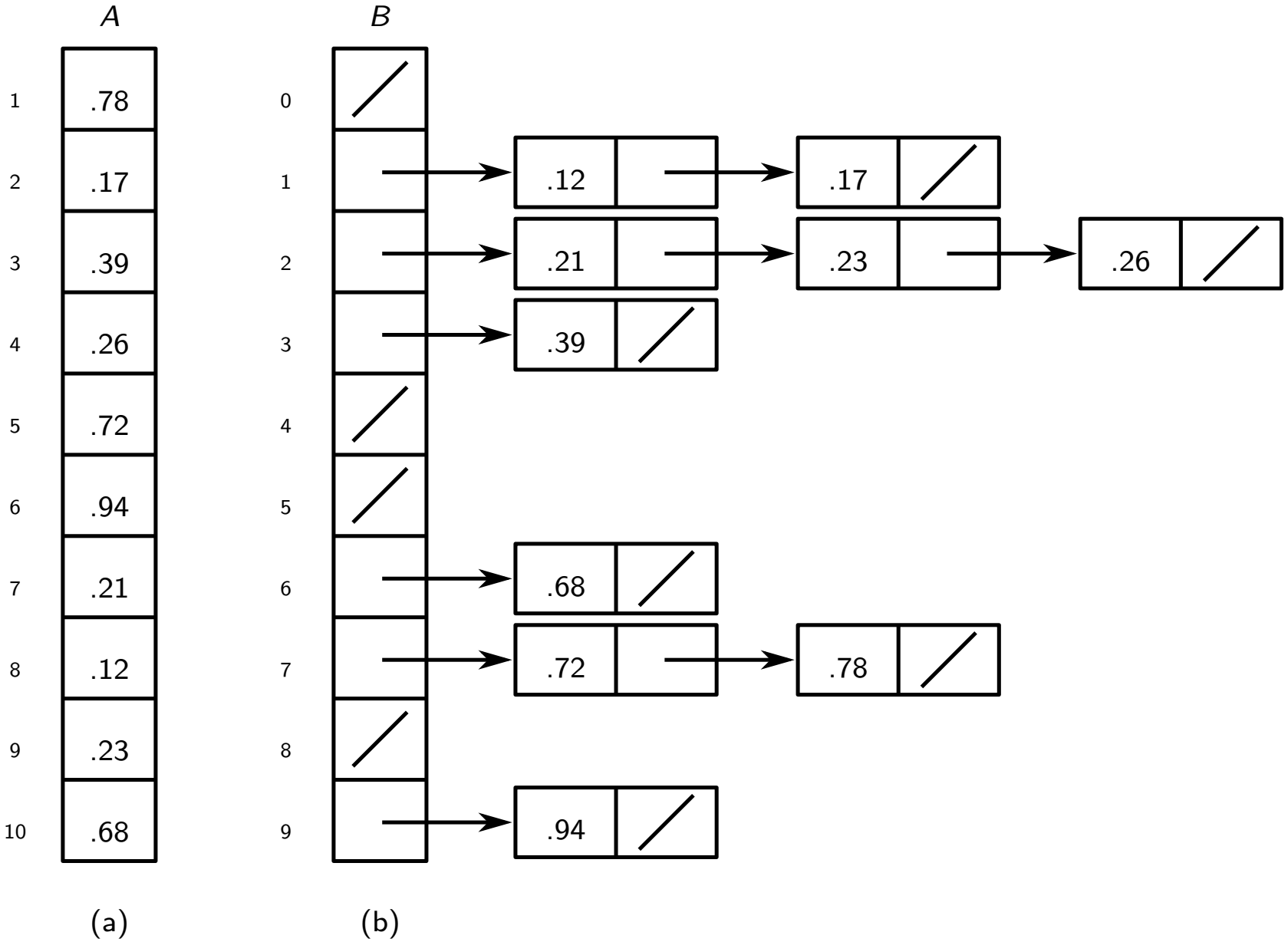
Přihrádkové řazení - Bucket Sort

předpokládá, že

- vstupní posloupnost obsahuje čísla z intervalu $[0 \ldots 1)$
- čísla rovnoměrně pokrývají celý interval

- interval $[0 \ldots 1)$ rozdělíme na stejně velké podintervaly - **koše**
- vstupní čísla rozdělíme dle jejich hodnoty do košů
- seřadíme prvky v každém koši

Bucket sort


A
B
1
.78
2
.17
3
.39
4
.26
5
.72
6
.94
7
.21
8
.12
9
.23
10
.68
0
1
2
3
4
5
6
7
8
9
.12
.17
.21
.23
.26
.39
.68
.72
.78
.94
(a)
(b)

Bucket sort

Bucket_Sort(A)

```
1 //B[0...n-1] je nové pole
2 n ← A.length
3 for i = 0 to n - 1 do
4     B[i] ← prázdny seznam od
5 for i = 1 to n do
6     přidej A[i] do seznamu B[⌊n · A[i]⌋] od
7 for i = 0 to n - 1 do
8     seřaď prvky seznamu B[i] použitím řazení vkládáním od
9 spoj seznamy B[0], B[1], ..., B[n - 1] do jednoho seznamu
```

- nechť n_i označuje počet prvků v koši $B[i]$
- složitost je $T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)$
- očekávaná složitost je pro vstup s uniformně rozdělenými čísly $\Theta(n)$