

Pokročilé řazení

Karel Richta a kol.

Přednášky byly připraveny s pomocí materiálů, které vyrobili Marko Berezovský a Michal Píše

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

© Karel Richta a kol., 2018

Datové struktury a algoritmy, B6B36DSA
01/2018, Lekce 7

<https://cw.fel.cvut.cz/wiki/courses/b6b36dsa/start>



Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Řazení dělením - diskuse

QUICK-SORT

Diskuse k řazení dělením

QUICK-SORT je typický představitel řešení metodou Rozděl-a-Panuj. Existují varianty dvou typů:

- jednoduché a stabilní, které ale mají větší skryté konstanty a vyžadují $O(n)$ pomocné paměti,
- komplikovanější, nestabilní, ale fakticky rychlejší a nevyžadující extra paměť, tzv. “in place” algoritmy.

Algoritmy prvního typu:

- Vyberou pivota.
- Projdou pole, označí a oindexují čísla menší, rovna a větší než pivot.
- Zahustí čísla menší než pivot doleva, za ně nahustí čísla rovna pivotu a za ně čísla větší než pivot.
- Nad první a třetí částí spustí rekurzivně totéž.

Algoritmus uvedený v předchozí přednášce byl druhého typu. V obou případech záleží hodně na výběru pivota.

Složitost v průměrném případě

- Předpoklad rovnoměrného rozložení: každá z $n!$ možných vstupních posloupností se objevuje s pravděpodobností $1/n!$.
- Je velmi nepravděpodobné, že dělicí poměr na všech úrovních bude stejný. Pravděpodobně některá dělení budou vyvážená a některá nevyvážená.
- Model průměrného případu:
 - např. střídání nejlepšího ($1/1$) a nejhoršího dělicího průměru ($1/n - 1$),
 - vyjde jenom o něco hůře než nejlepší.
- Otočíme pohled: místo modelování a spekulování o dělicích poměrech vnutíme vstupním datům charakter náhodné posloupnosti.
- Typicky: vstupní posloupnost podrobíme randomizaci (rozházení), čímž složitost řazení dělením není závislá na uspořádanosti vstupu.
- Potřebujeme náhodný generátor.

Příklad randomizovaného QUICK-SORT

- Mějme funkci $\text{Random}(a, b)$, která vrátí celé číslo c , $a \leq c \leq b$, s pravděpodobností $1/(b-a+1)$.
- Pak randomizovaný **QUICK-SORT** používá:

```
function SelectRandomPivot(A, low, high) {  
    i ← Random(low, high);  
    A[i] ↔ A[low];  
    return A[low]  
}
```

Diskuse k řazení dělením (pokr.)

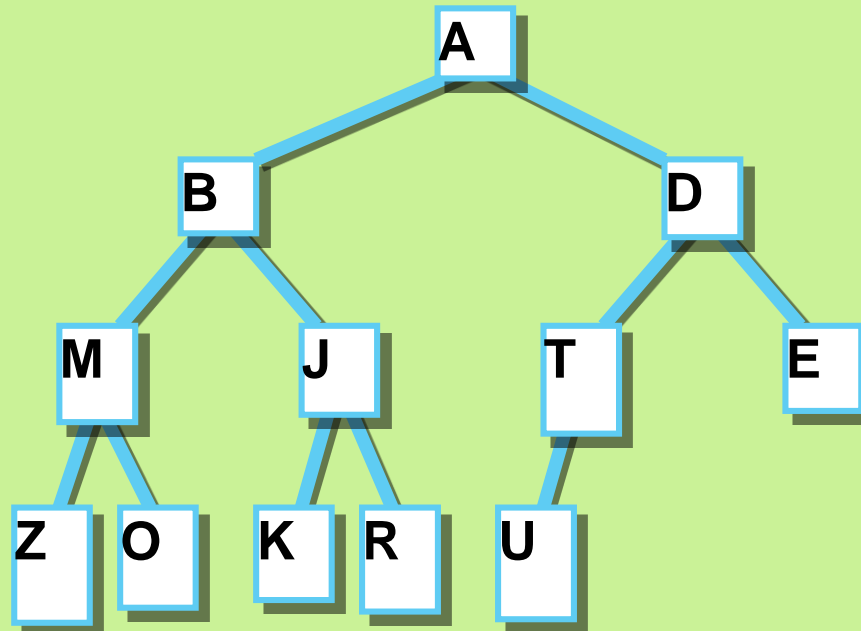
- Časová složitost v nejhorším případě $\Theta(n^2)$.
- Časová složitost v průměrném (a očekávaném) případě $\Theta(n \log n)$.
- Nízká multiplikativní konstanta.
- Potřebuje případně jen konstantní prostor navíc.
- Správnost = částečná správnost + terminace
 - Částečná správnost (invariant)?
 - Terminace?

Řazení pomocí haldy

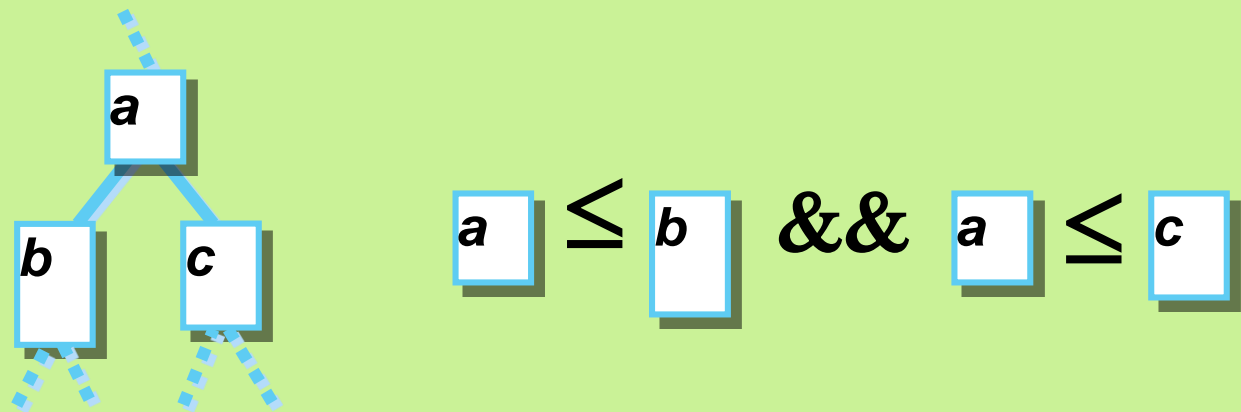
HEAP-SORT

Halda

Halda
(binární)

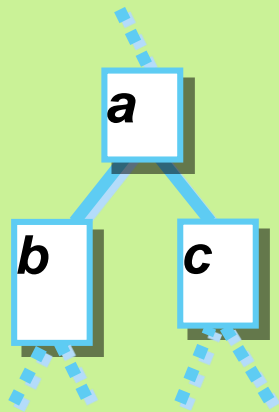


Pravidlo
haldy
(MINHEAP)

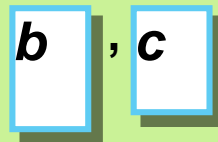
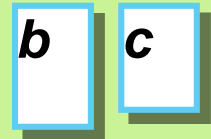


Halda

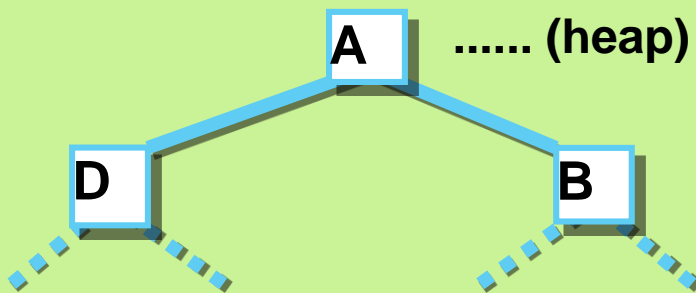
Terminologie



a predecessor, parent of
 předchůdce, rodič



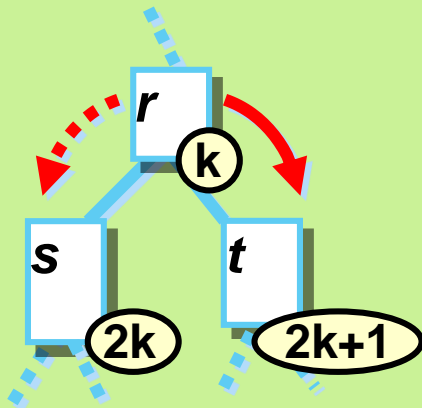
b, **c** successor, child of
 následník, potomek



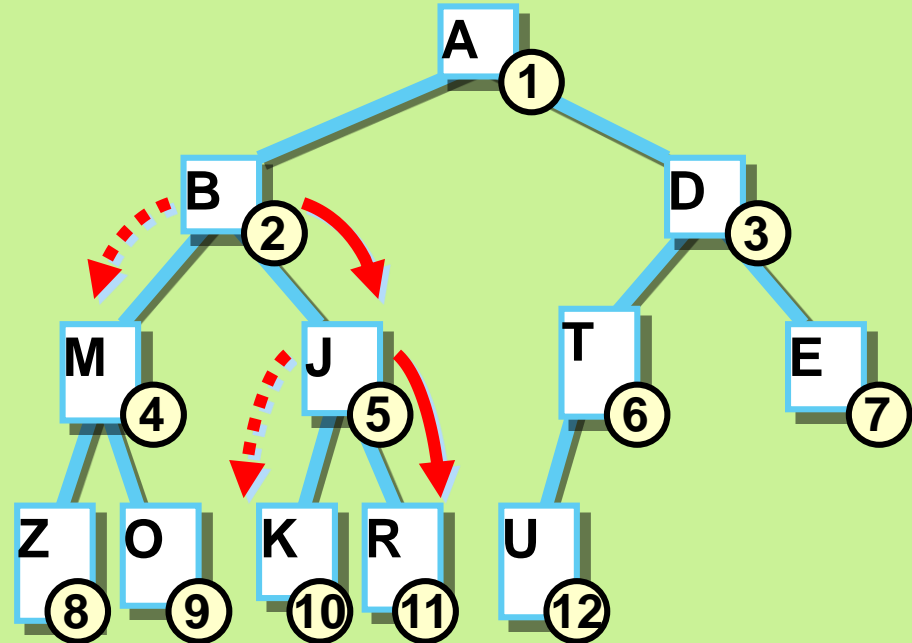
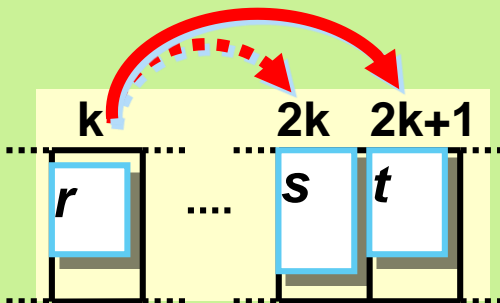
A (heap) top vrchol (haldy)

Halda v poli

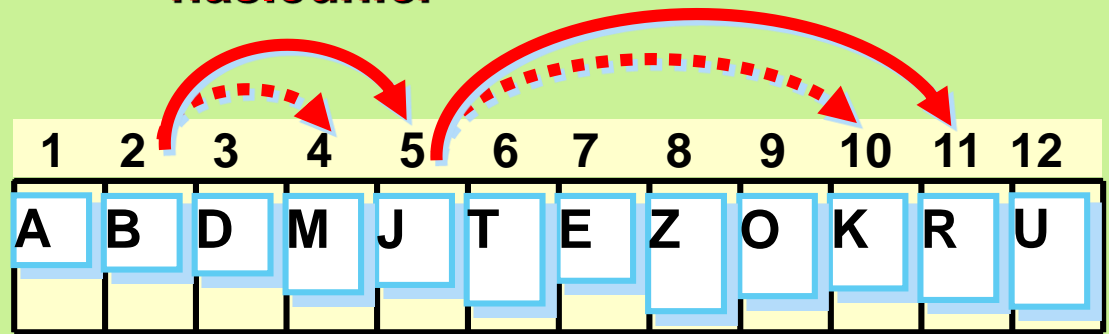
Halda uložená v poli



následníci

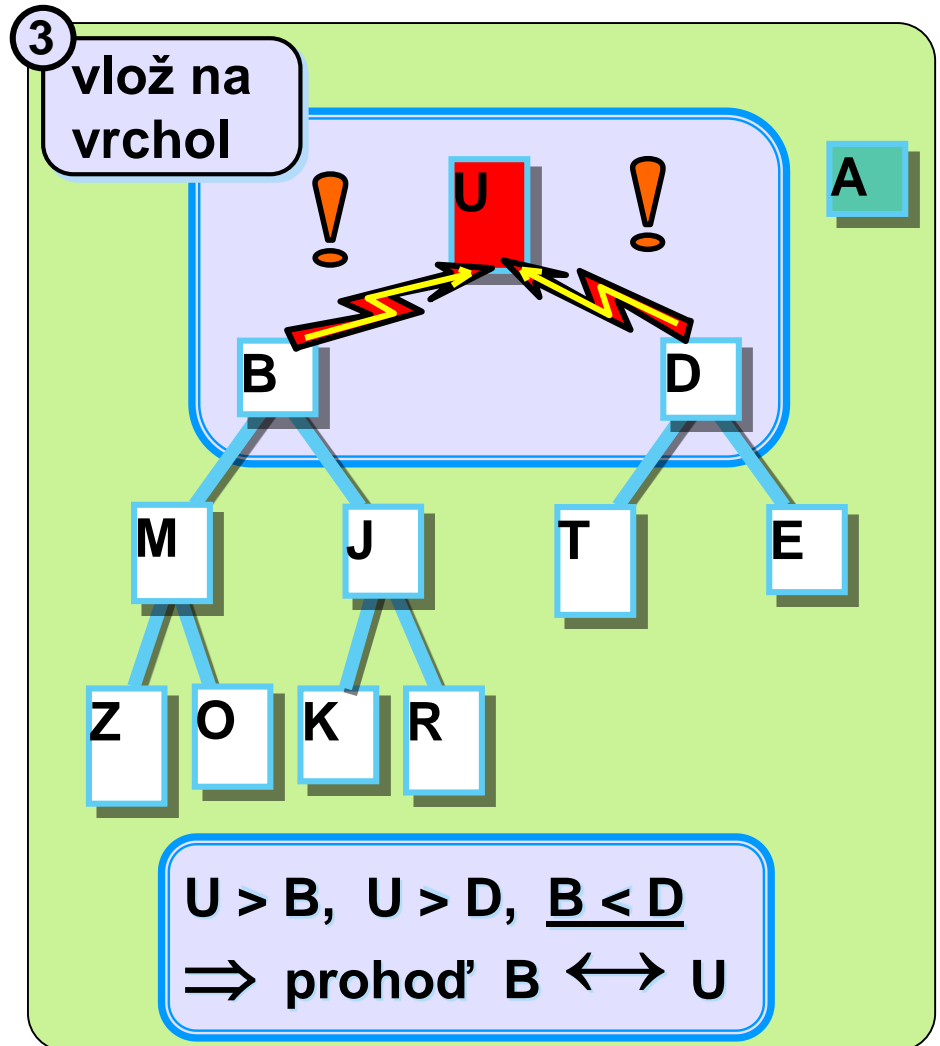
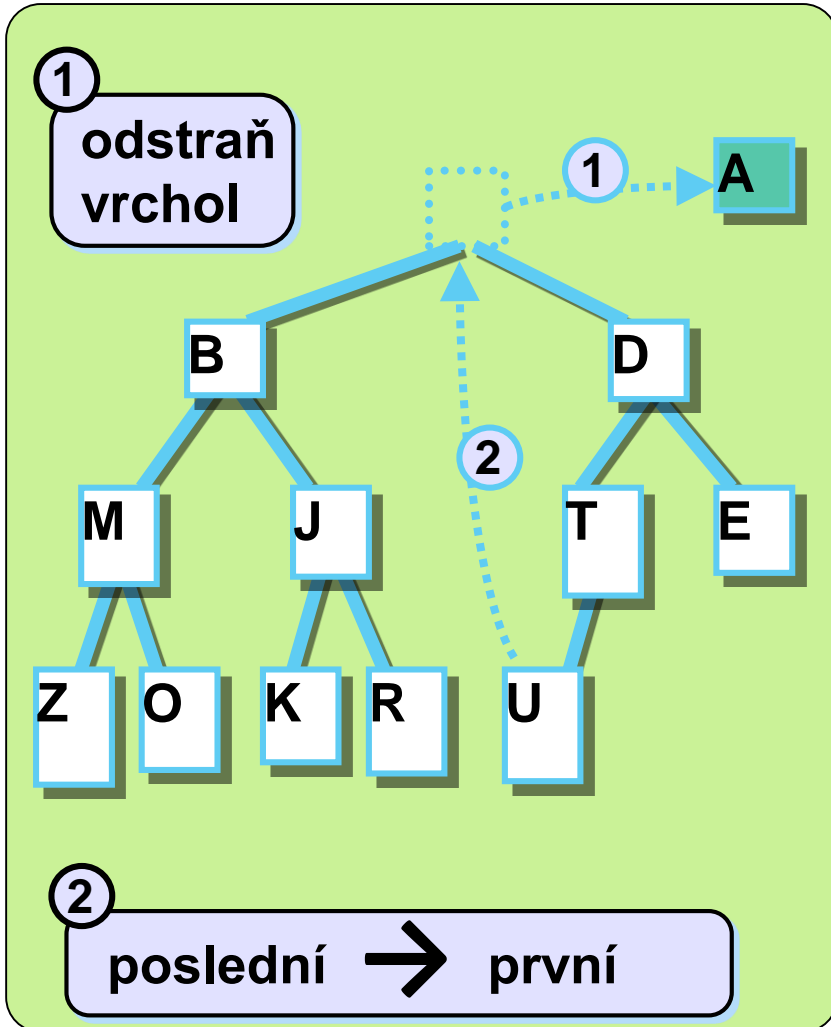


následníci



Oprava haldy

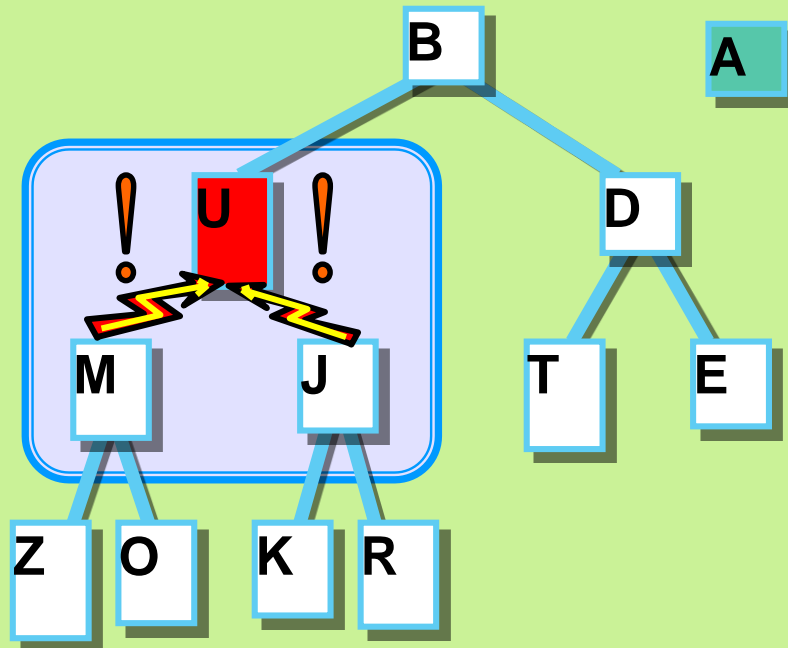
Vrchol odstraněn (1)



Oprava haldy

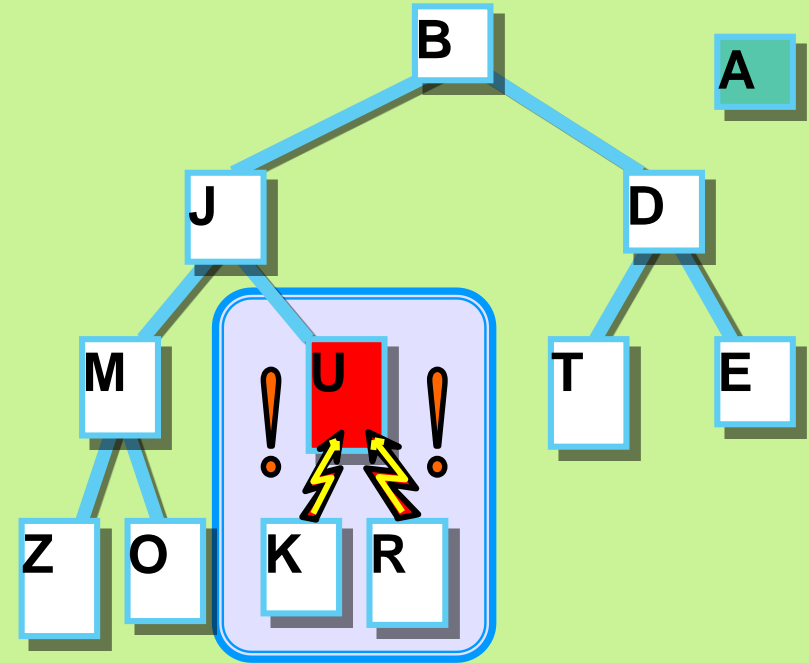
Vrchol odstraněn (2)

3 vlož na vrchol - pokračování



$U > M, U > J, \underline{J < M}$

\Rightarrow prohod' $J \leftrightarrow U$



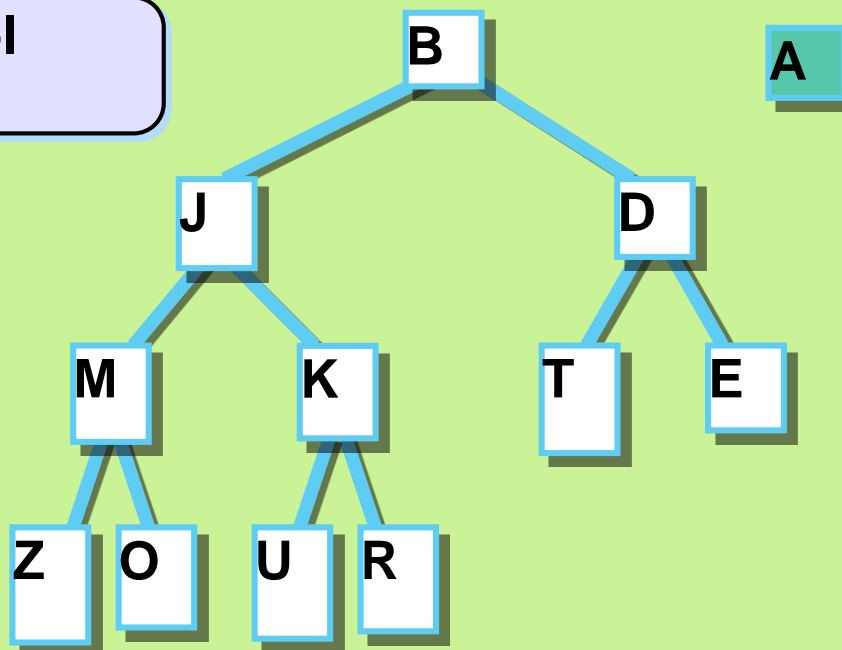
$U > K, U > R, \underline{K < R}$

\Rightarrow prohod' $K \leftrightarrow U$

Oprava haldy

Vrchol odstraněn (3)

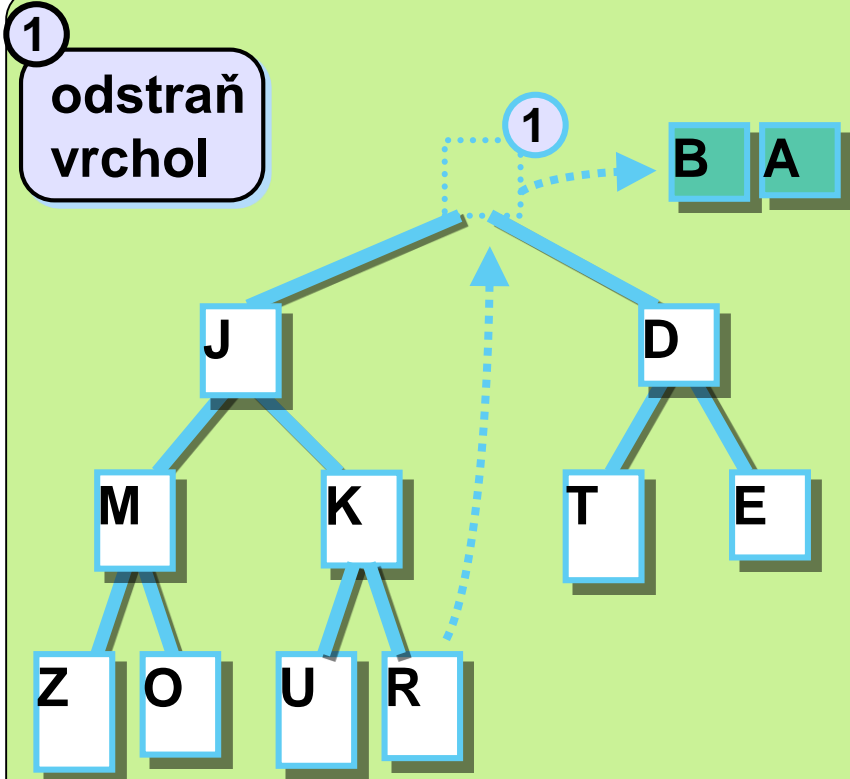
3 vlož na vrchol
- hotovo



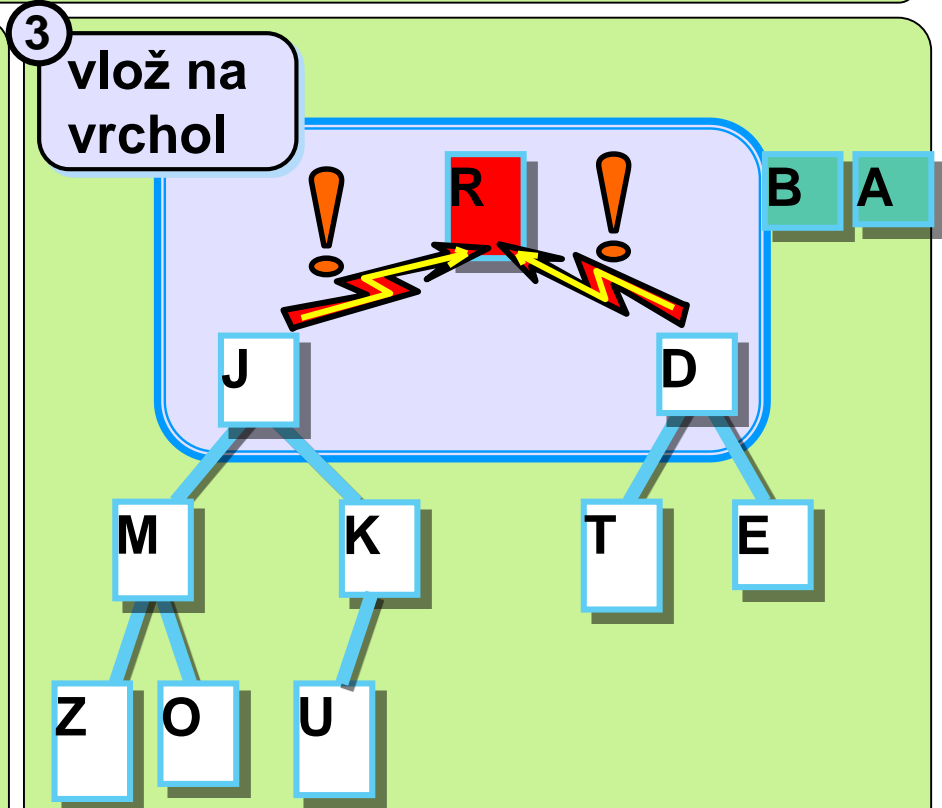
Nová halda

Oprava haldy

Vrchol odstraněn II (1)



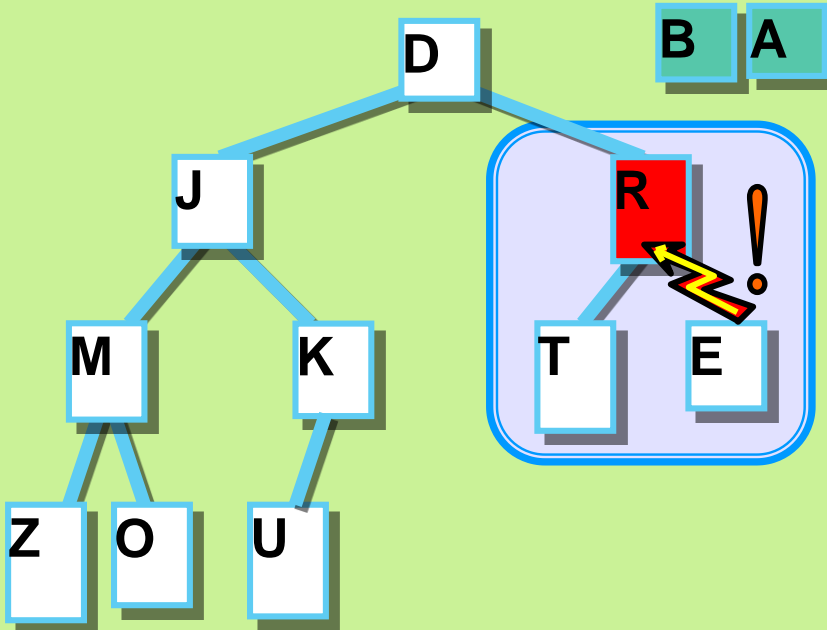
2
poslední → první



Oprava haldy

Vrchol odstraněn II (2)

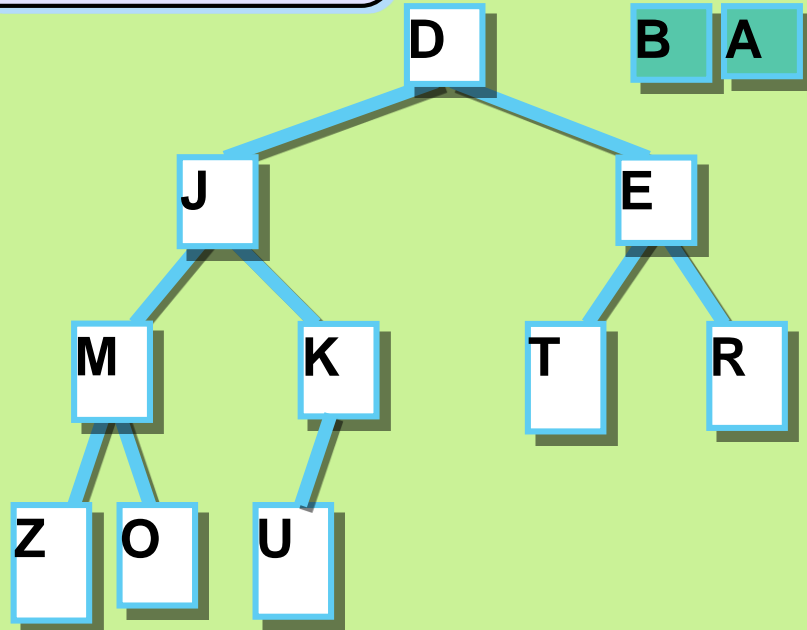
3 vlož na vrchol - pokračování



$R < T, R > E$
 \Rightarrow prohod' $E \leftrightarrow R$

Vrchol odstraněn II (3)

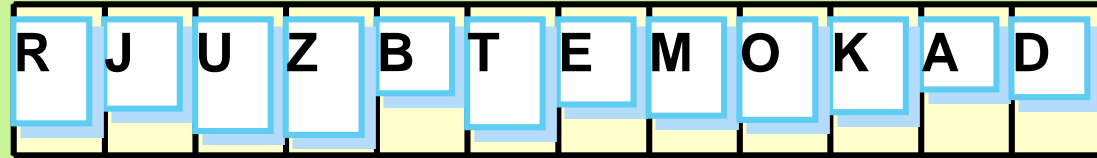
3 vlož na vrchol - hotovo



Nová halda

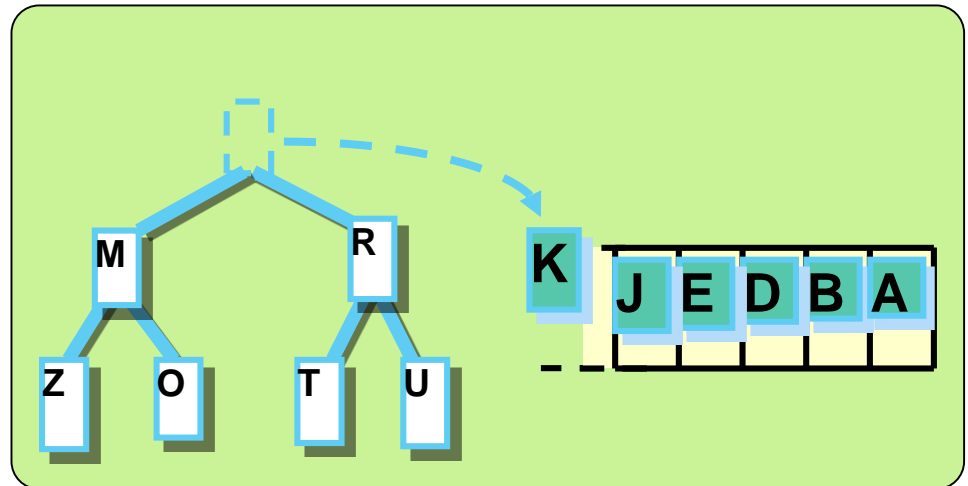
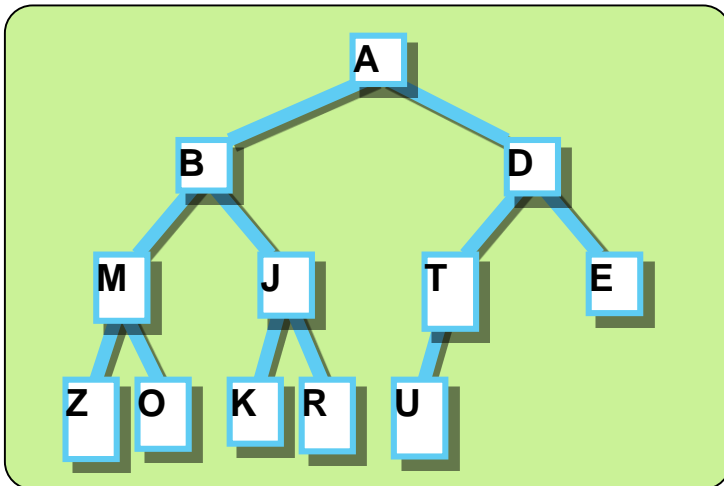
Řazení pomocí haldy - HEAP-SORT

I Neseřazeno

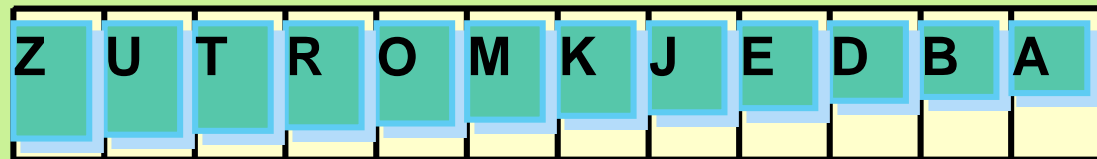


II Vytvoř haldu

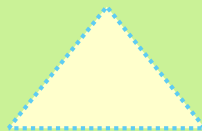
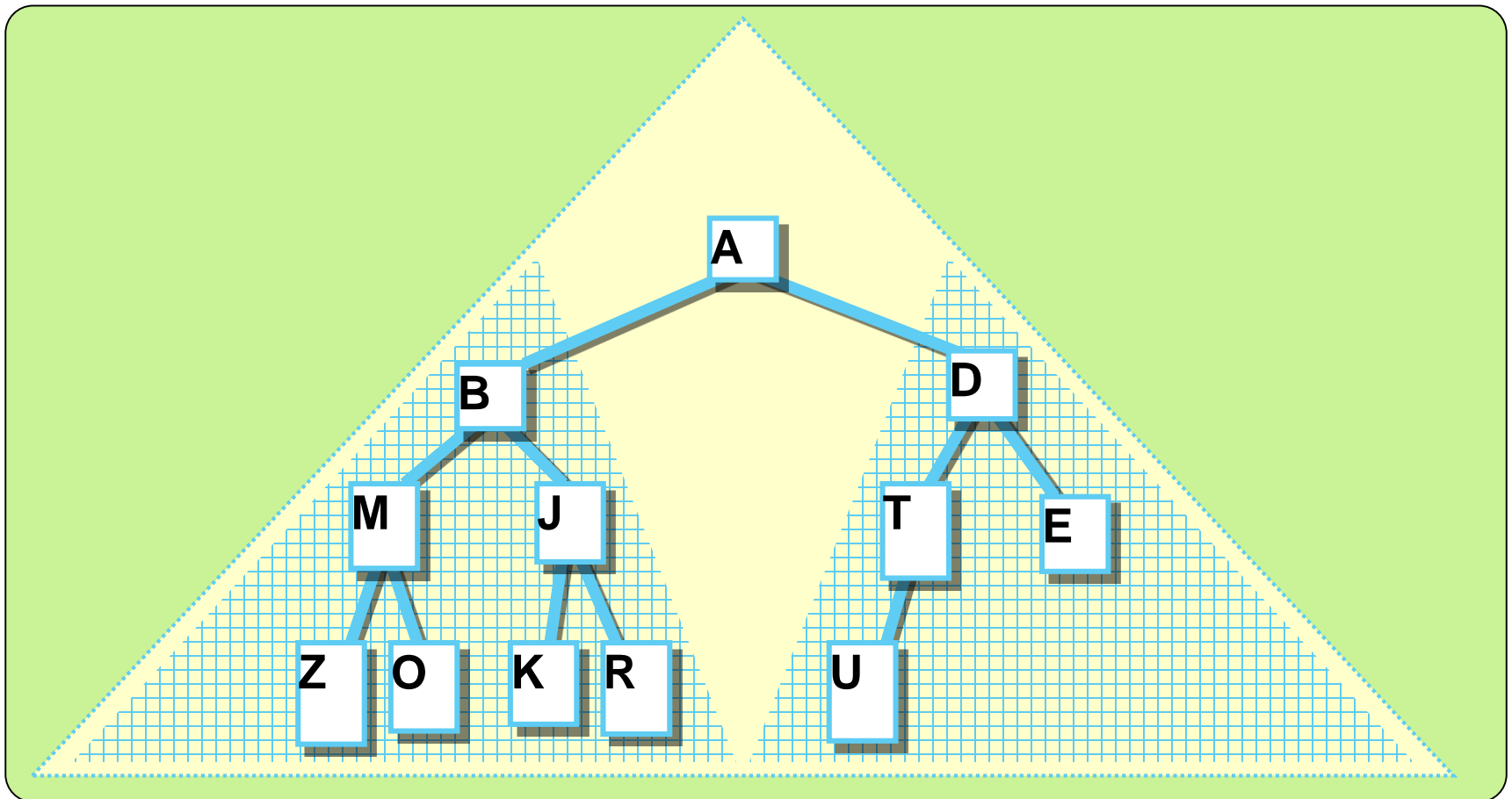
III for ($i = 0$; $i < n$; $i++$)
a[i] = "odstraň vrchol";



IV Seřazeno



Rekurzivní vlastnost „býti haldou“



je halda

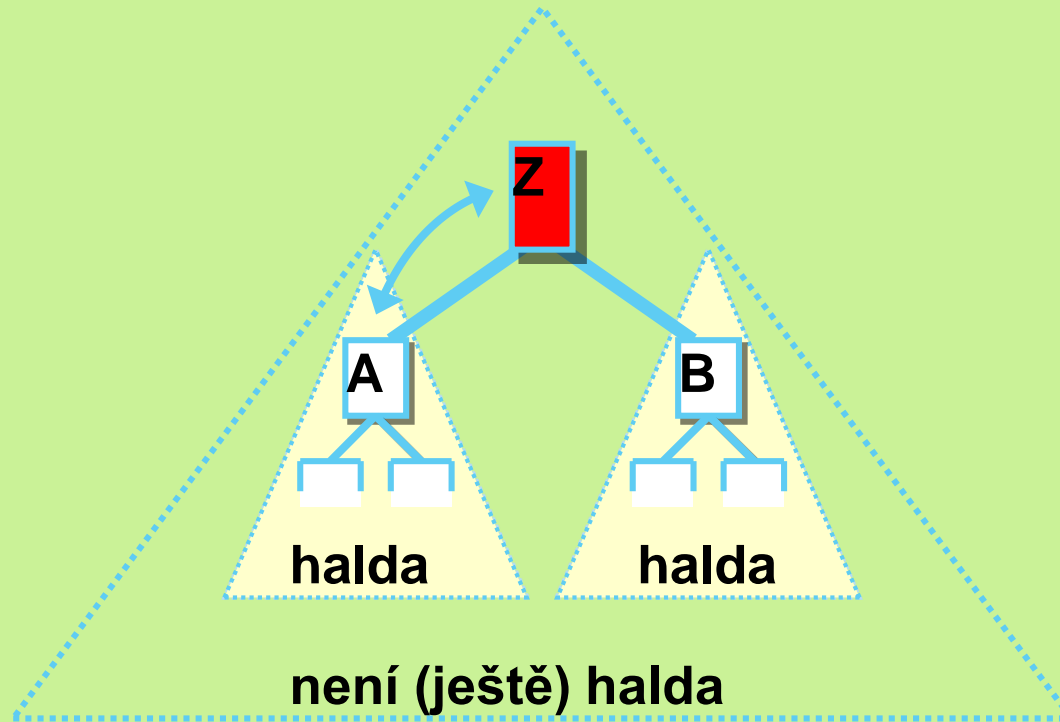


je halda



je halda

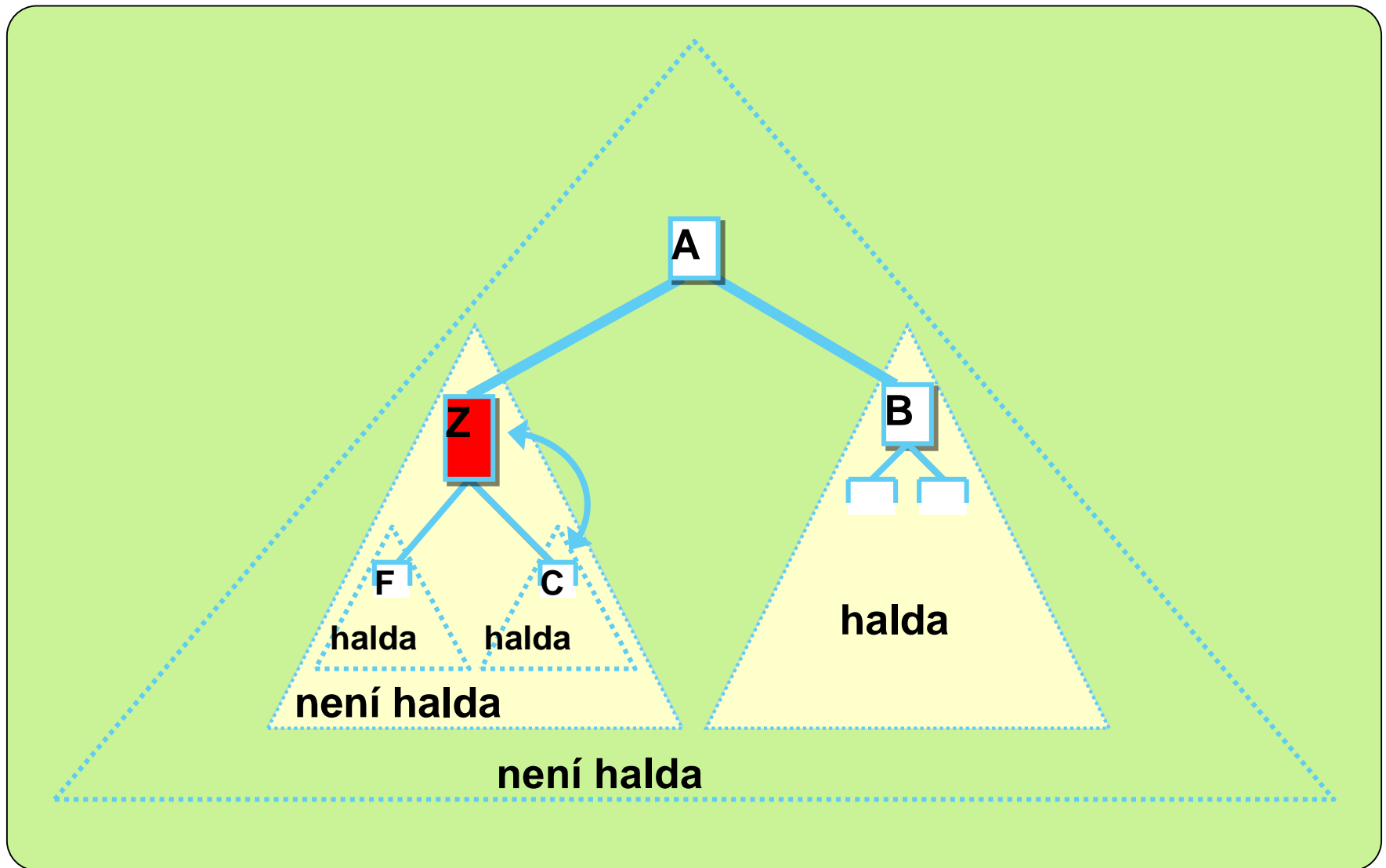
Vytvoř jednu haldu ze dvou menších



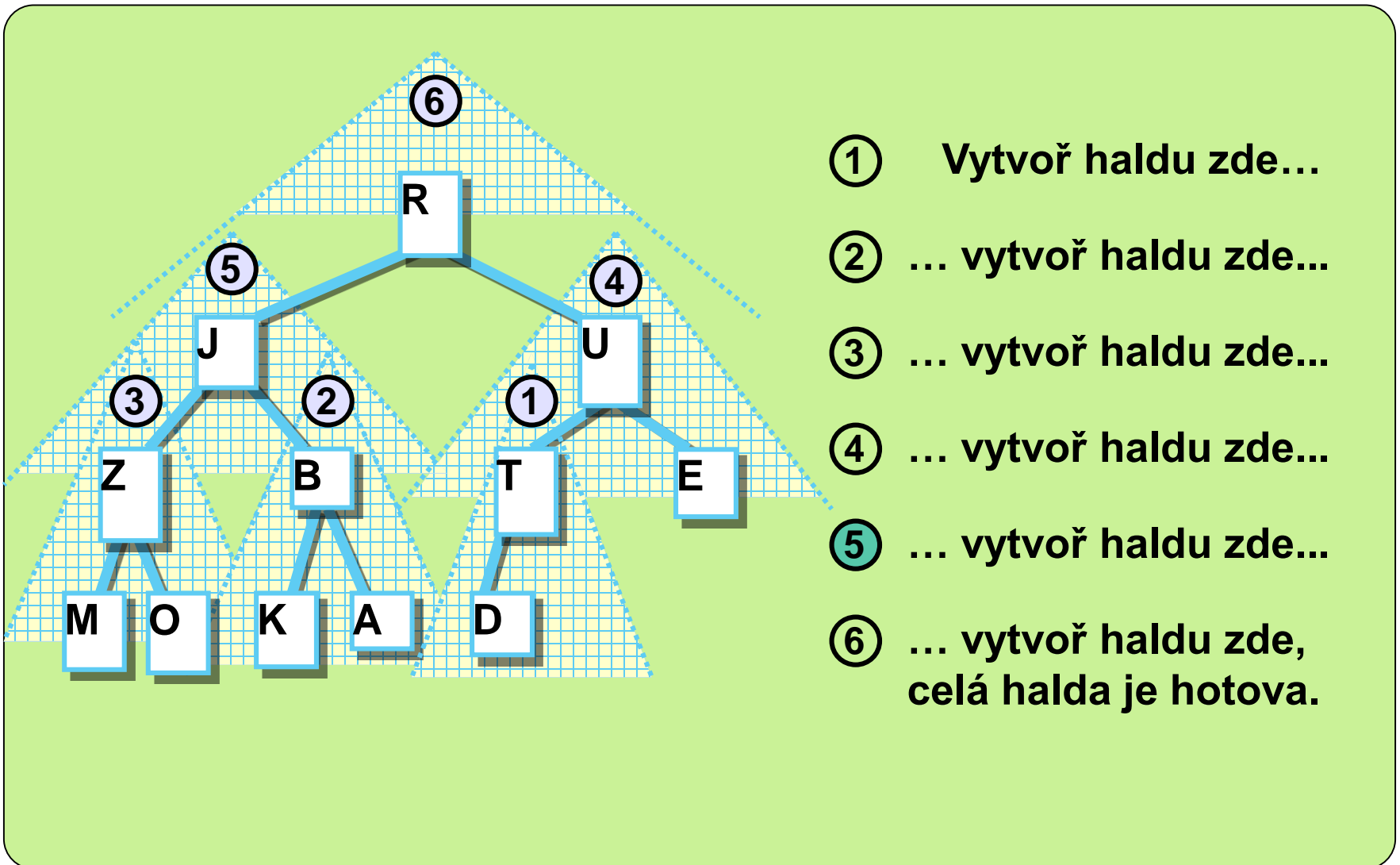
$Z > A$ nebo $Z > B$

\Rightarrow prohod': $Z \leftrightarrow \min(A, B)$

Vytvoř jednu haldu ze dvou menších

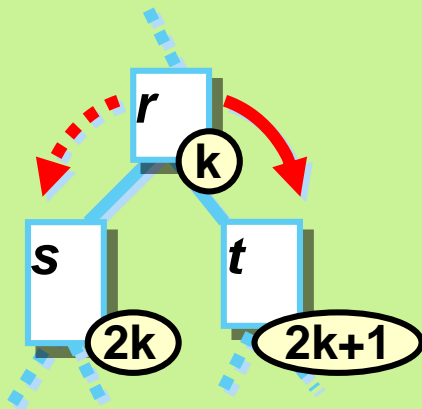


Vytvoř haldu

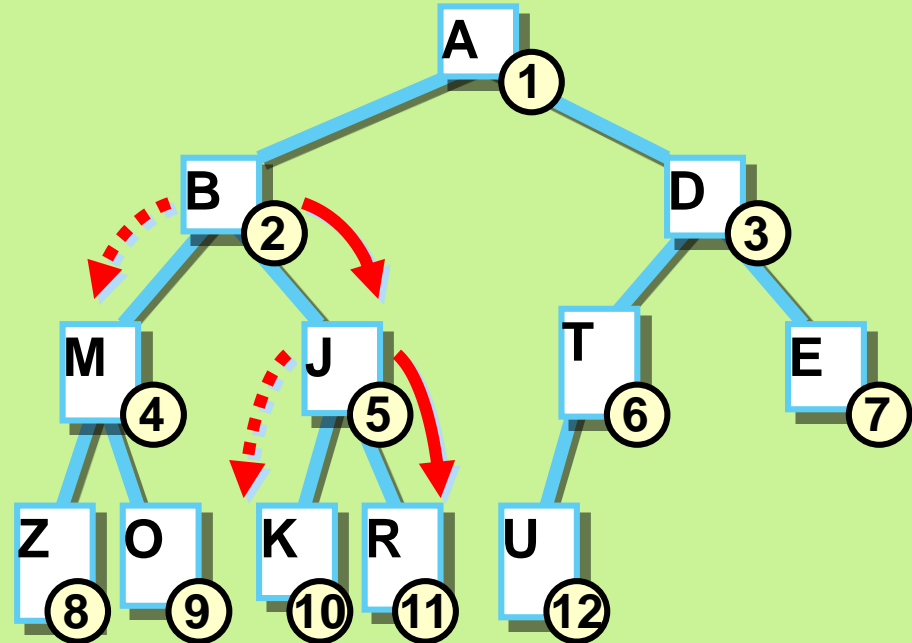
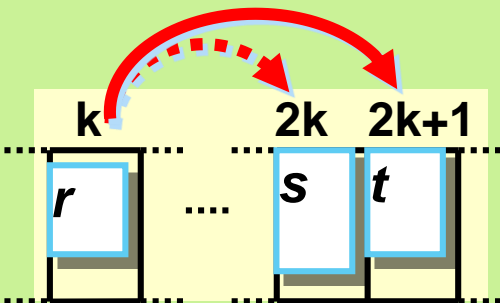


Halda v poli

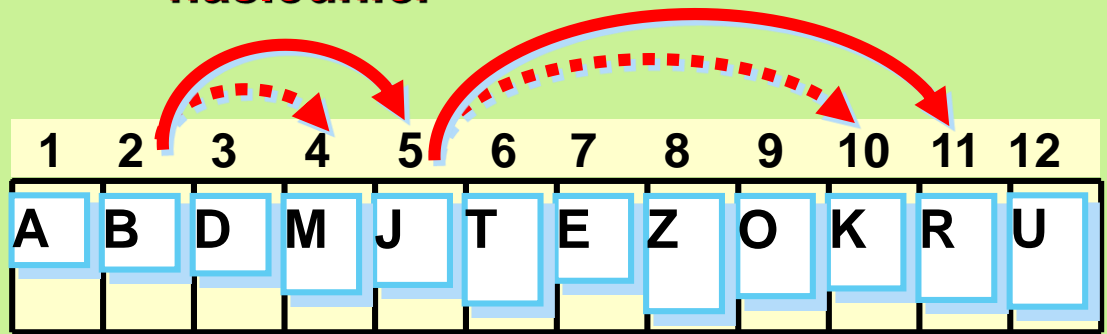
Halda uložená v poli



následníci



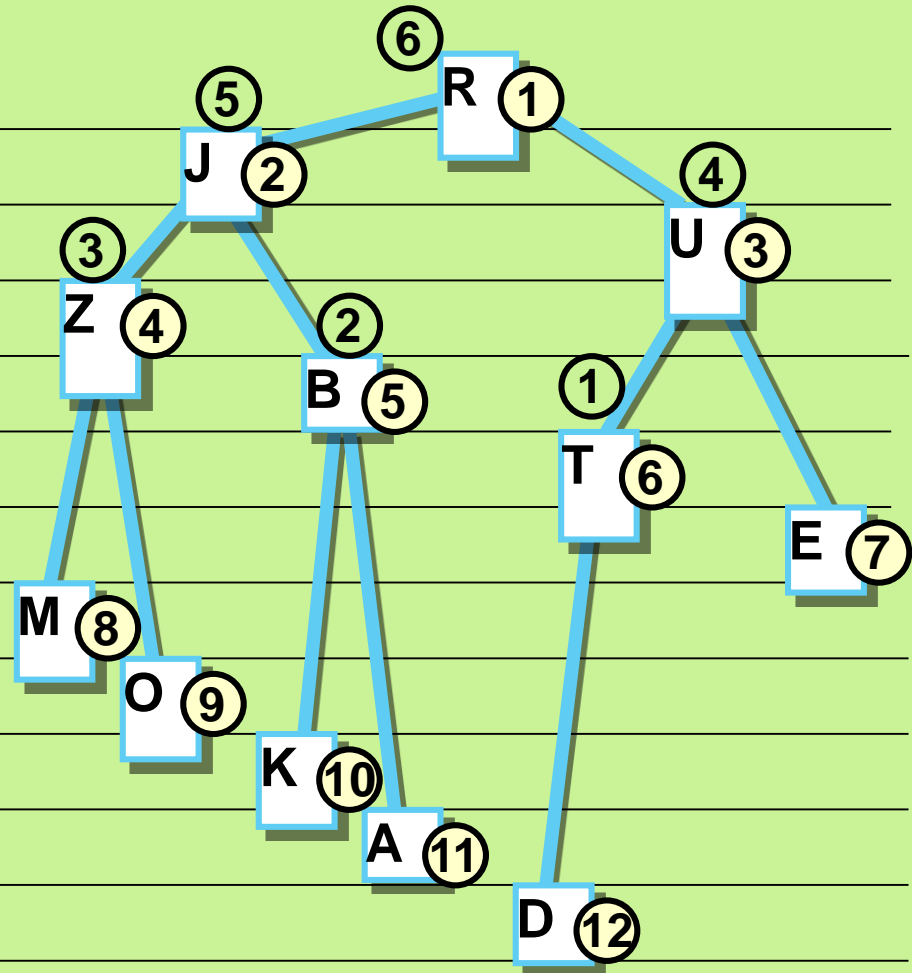
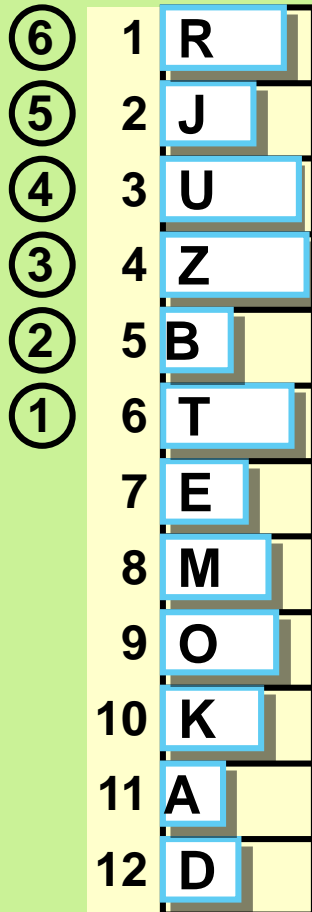
následníci



Tvorba haldy v poli

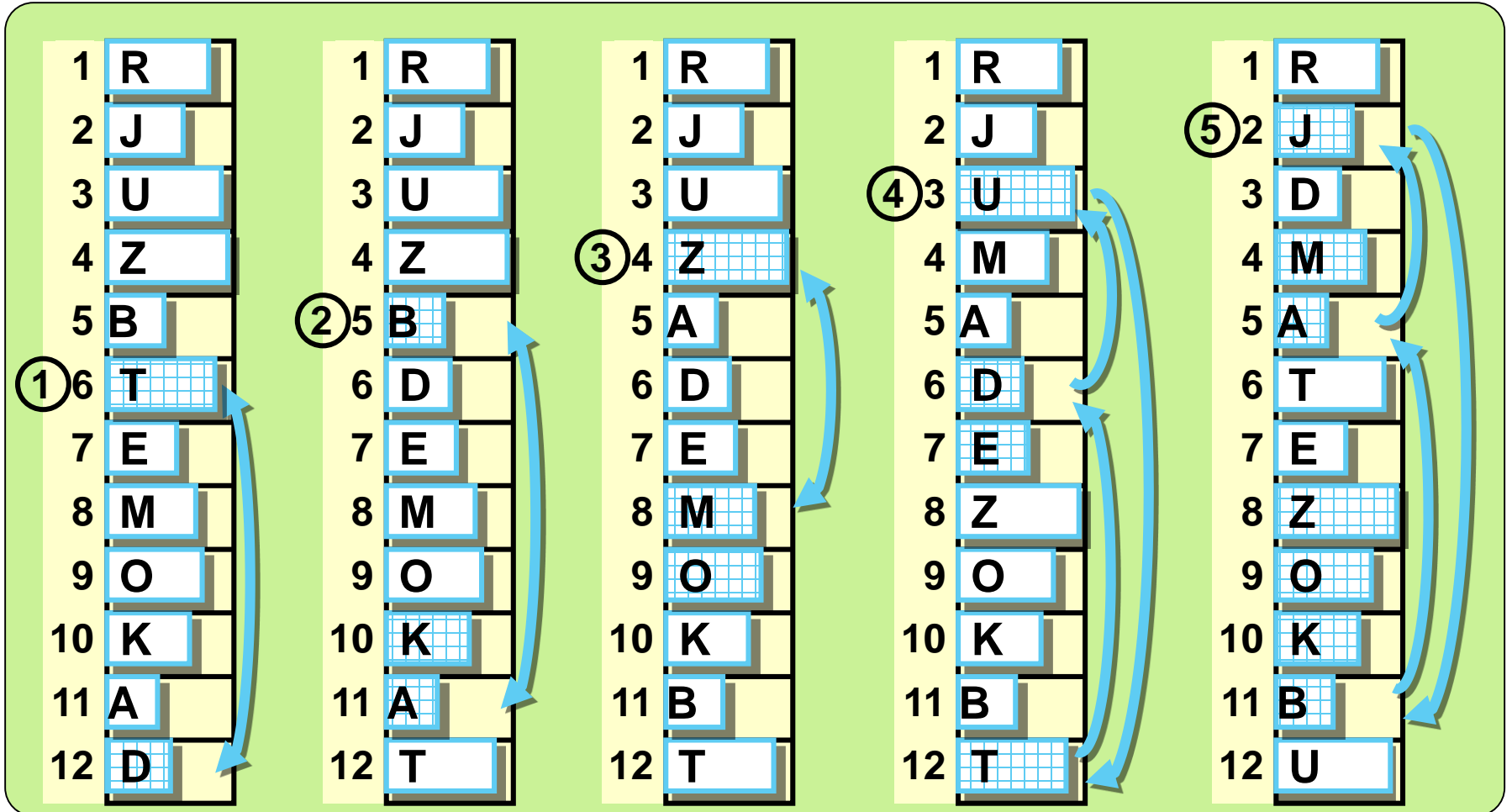
Neseřazeno

Není halda



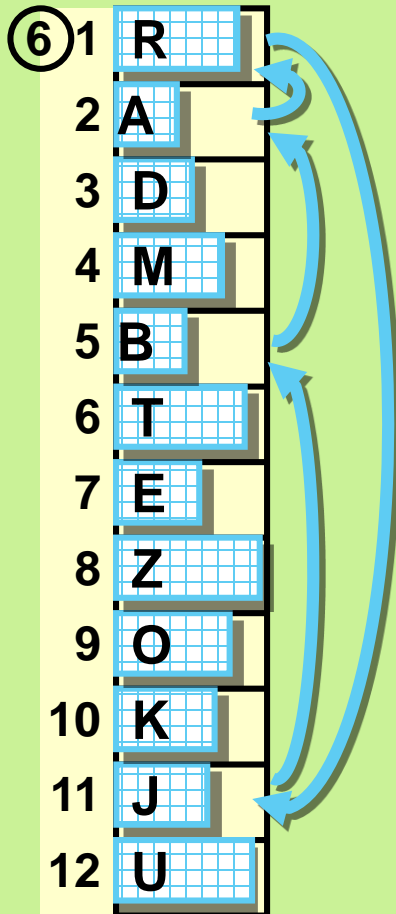
Tvorba haldy v poli

tvorba haldy:

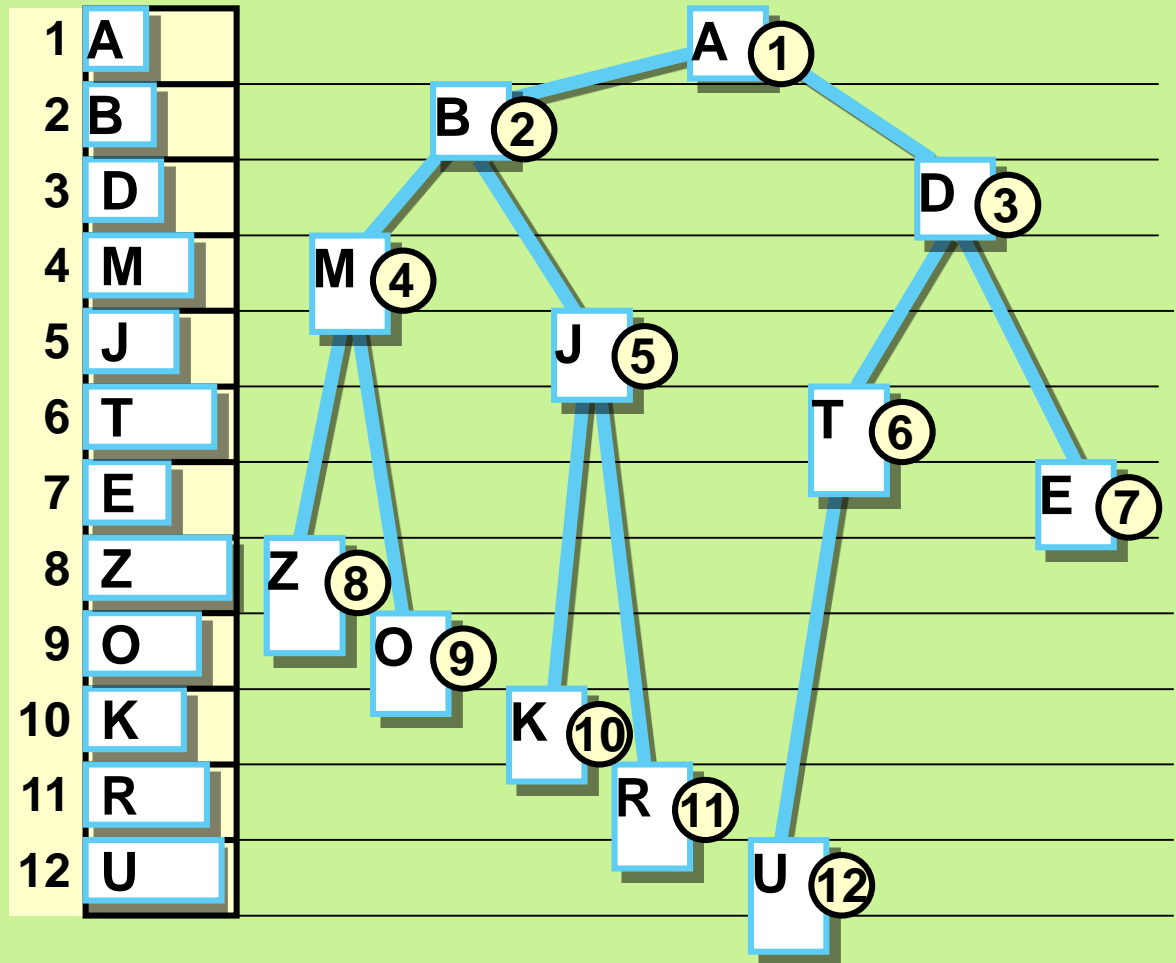


Tvorba haldy v poli

Tvorba haldy

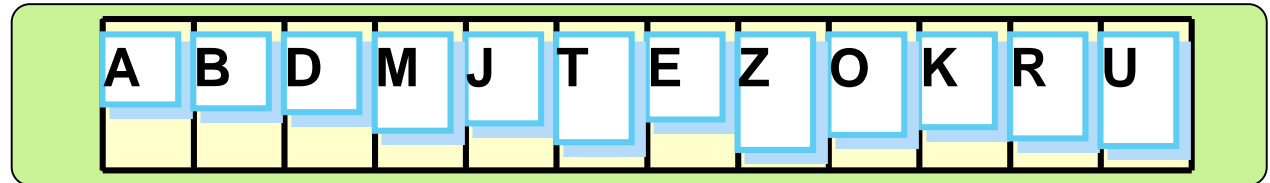


Halda

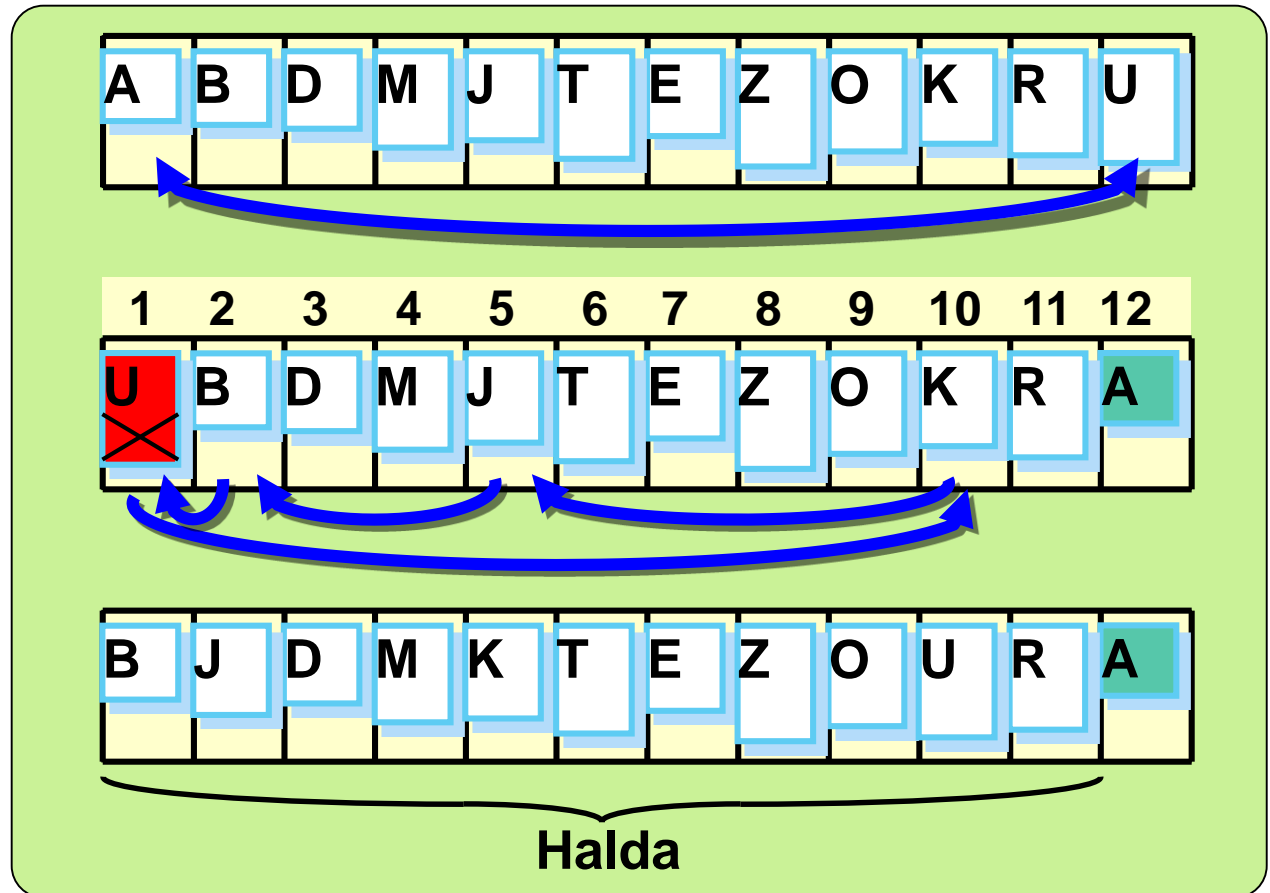


HEAP-SORT

Halda

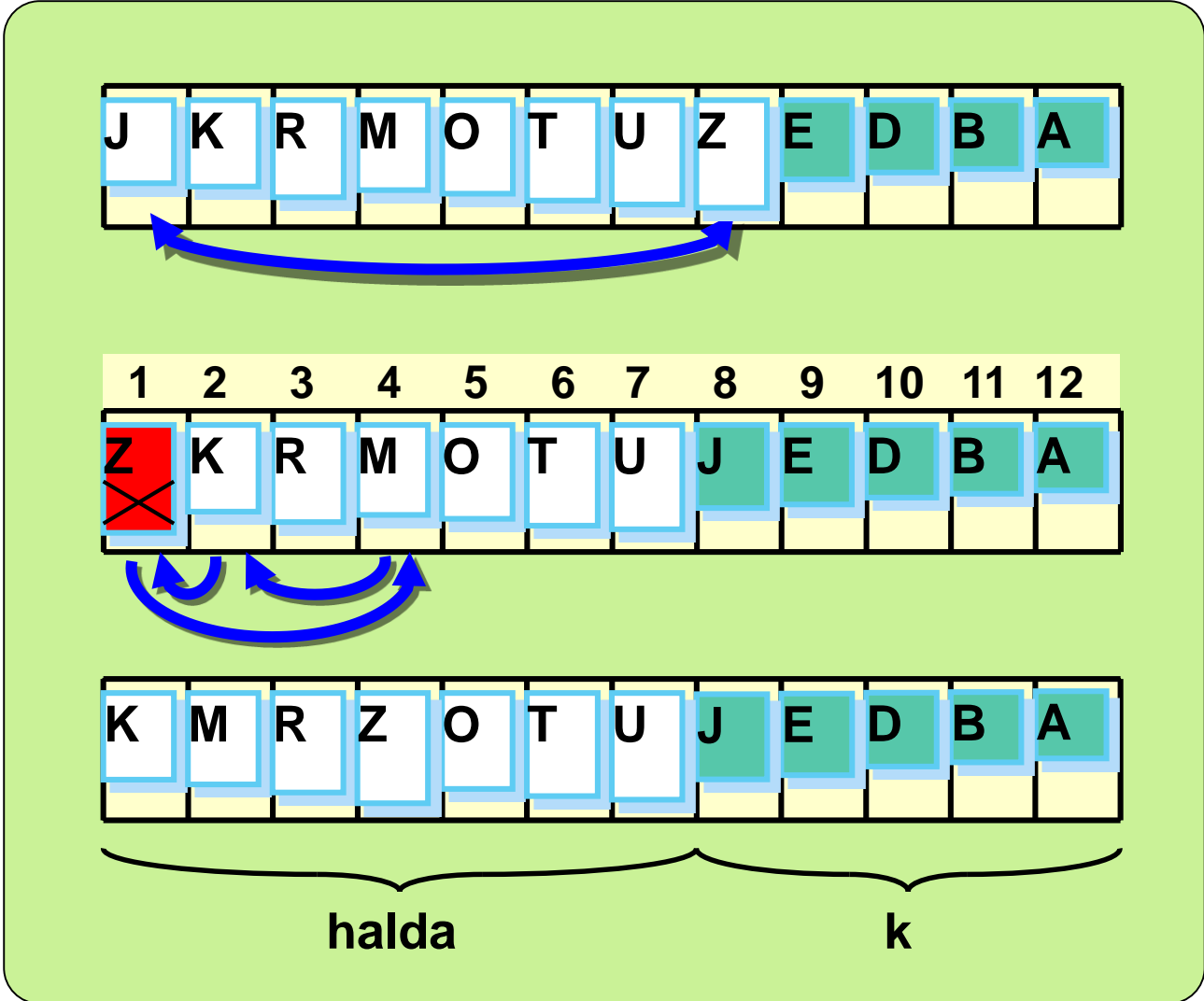


Krok 1



HEAP-SORT

Krok k



HEAP-SORT

```
                                // array: a[1]...a[n]   !!!!  
  
void heapSort(Item a[], int n) {  
    int i, j,  
                                // create a heap  
    for (i = n/2; i > 0; i--)  
        repairTop(array, i, n);  
  
                                // sort  
    for (i = n; i > 1; i--) {  
        swap(a, 1, i);  
        repairTop(array, 1, i-1);  
    }  
}
```

REPAIRTOP pro HEAP-SORT

```

// array:  a[1]...a[n]  !!!!!
void repairTop(Item a[], int top, int bott) {
    int i = top;           // a[2*i] and a[2*i+1]
    int j = i*2;         // are successors of a[i]

    Item topVal = a[top];

                        // try to find a successor < topVal
    if ((j < bott) && (a[j] > a[j+1])) j++;

                        // while (successors < topVal)
                        //           move successors up
    while ((j <= bott) && (topVal > a[j])) {
        a[i] = a[j];
        i = j;  j = j*2; // skip to next successor
        if ((j < bott) && (a[j] > a[j+1])) j++;
    }
    a[i] = topVal;      // put the topVal
}

```

Shrnutí pro HEAP-SORT

repairTop operace nejhorší případ ... $\log_2(n)$ (n =velikost haldy)

vytvoř haldu ... $n/2$ repairTop operací

$$\log_2(n/2) + \log_2(n/2+1) + \dots + \log_2(n) \leq (n/2)(\log_2(n)) = \underline{\underline{O(n \cdot \log_2(n))}}$$

seřad' haldy ... $n-1$ repairTop operací, nejhorší případ:

$$\log_2(n) + \log_2(n-1) + \dots + 1 \leq n \cdot \log_2(n) = O(n \cdot \log_2(n))$$

$$\text{ale i nejlepší případ} = \underline{\underline{\Theta(n \cdot \log_2(n))}}$$

$$\text{celkem ... vytvoř haldu + seřad' haldu} = \underline{\underline{\Theta(n \cdot \log_2(n))}}$$

Asymptotická složitost HEAP-SORT je $\Theta(n \cdot \log_2(n))$

HEAP-SORT není stabilní

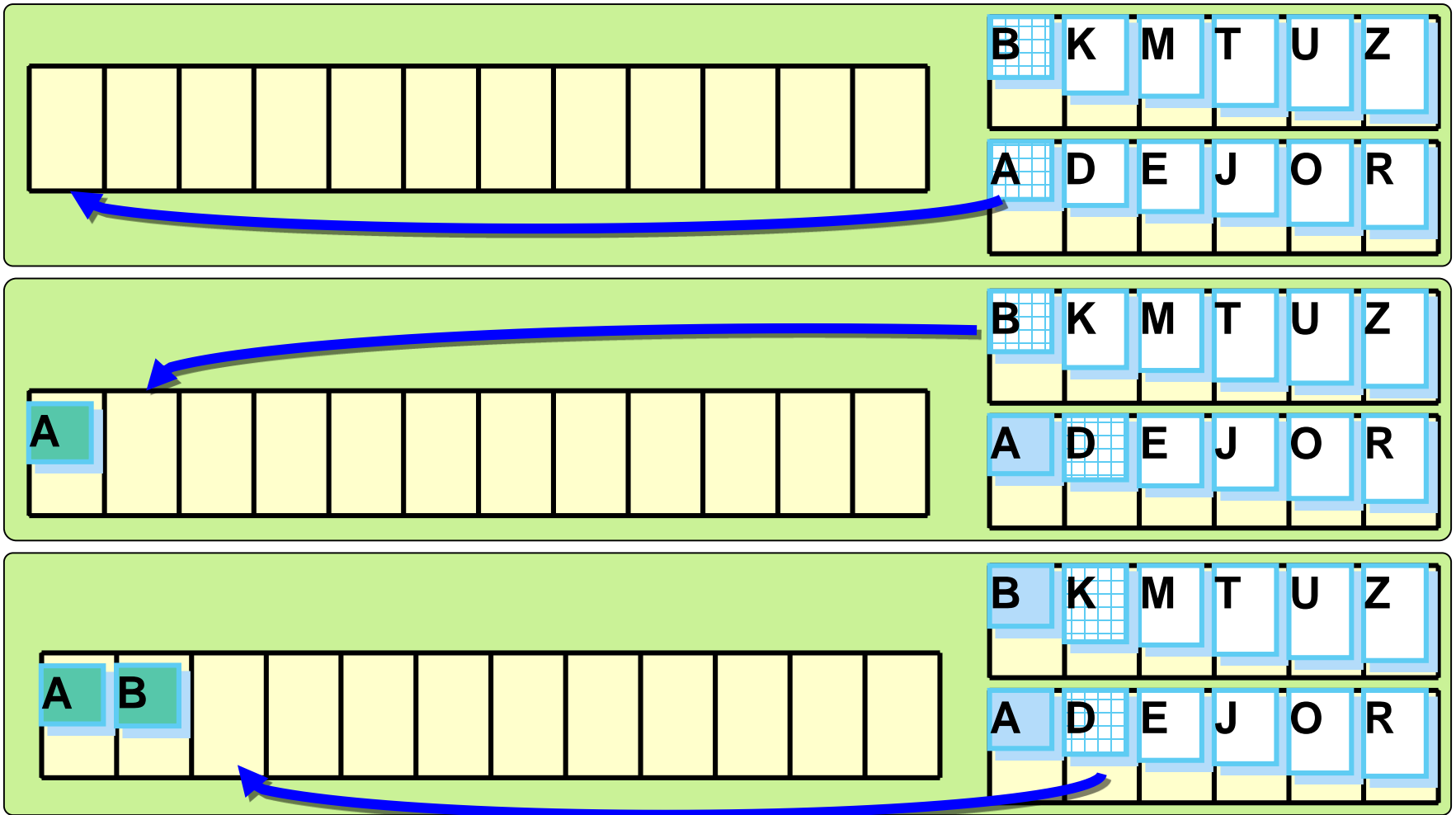
Řazení slučováním

MERGE-SORT

Slučování posloupností

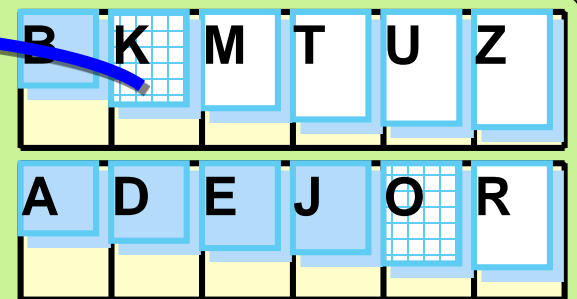
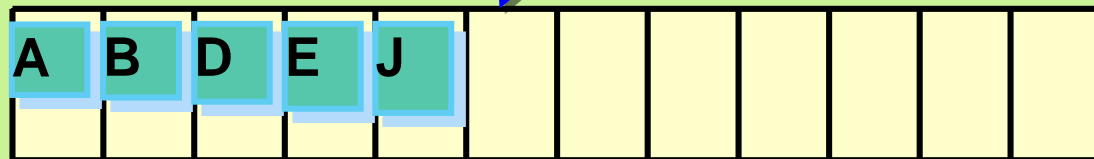
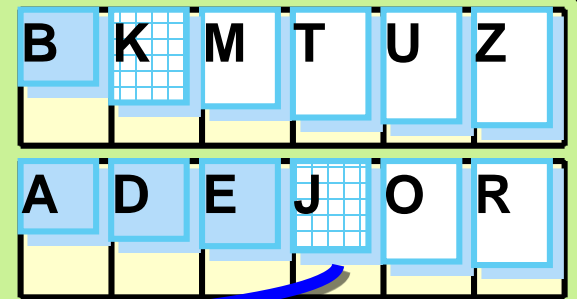
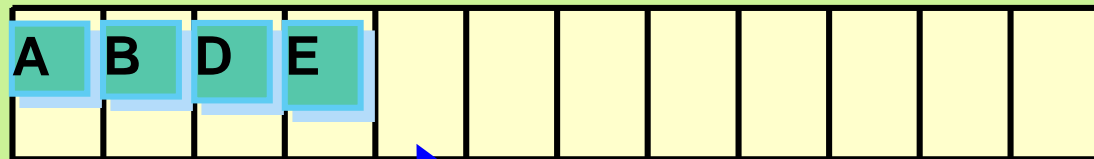
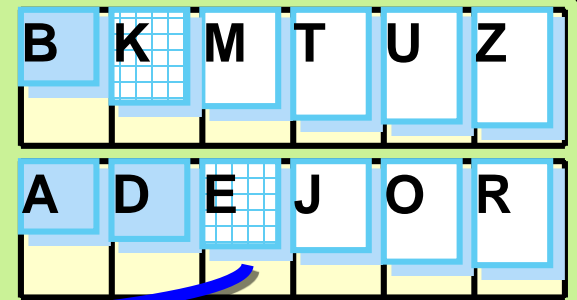
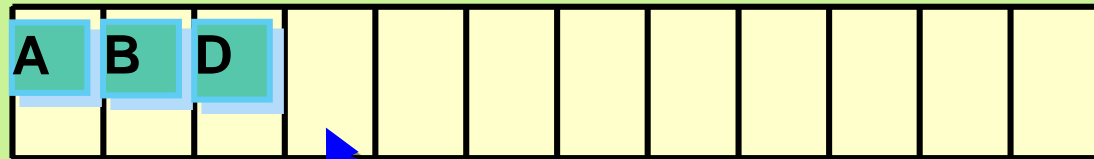
Sluč (slij?) dvě seřazená pole

porovnávané prvky 



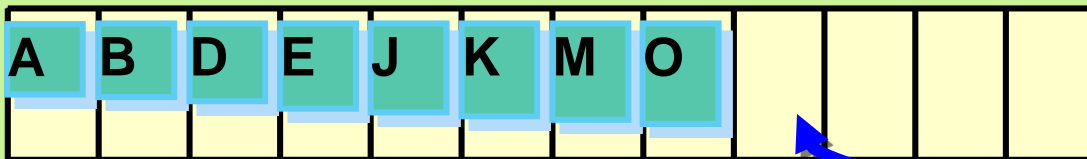
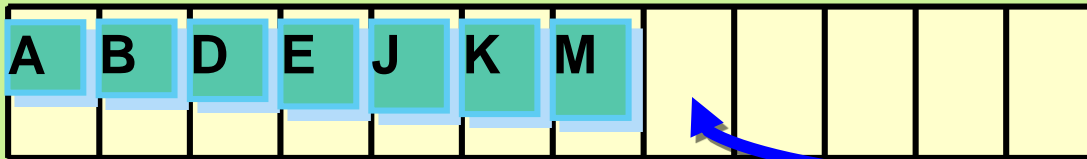
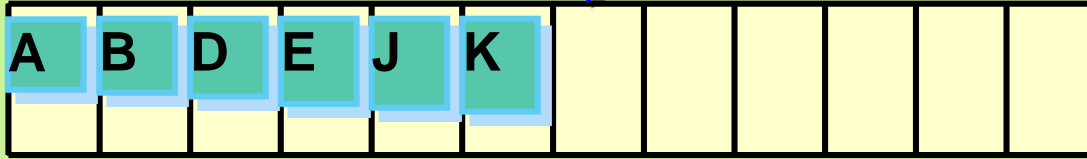
Slučování posloupností

Sluč dvě seřazená pole - pokr.



Slučování posloupností

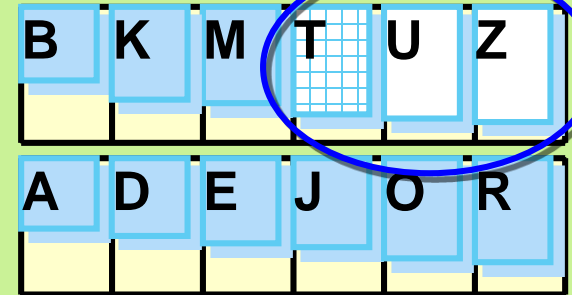
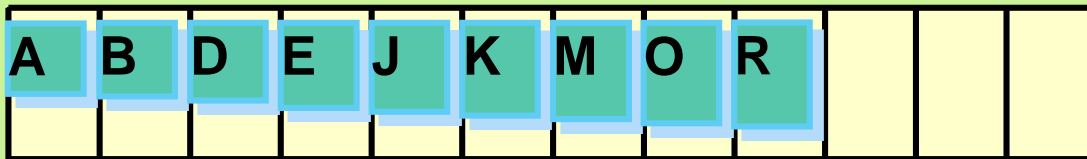
Sluč dvě seřazená pole - pokr.



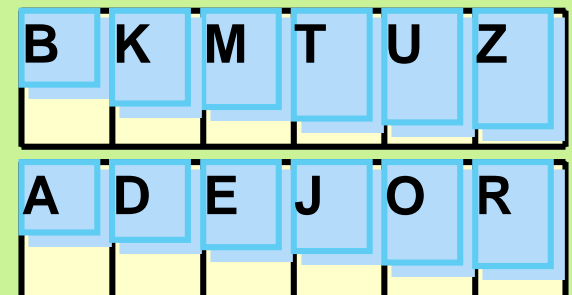
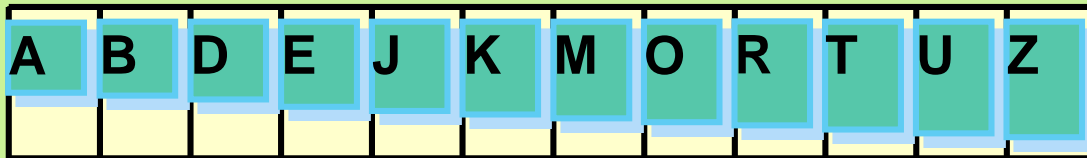
Slučování posloupností

Sluč dvě seřazená pole - pokr.

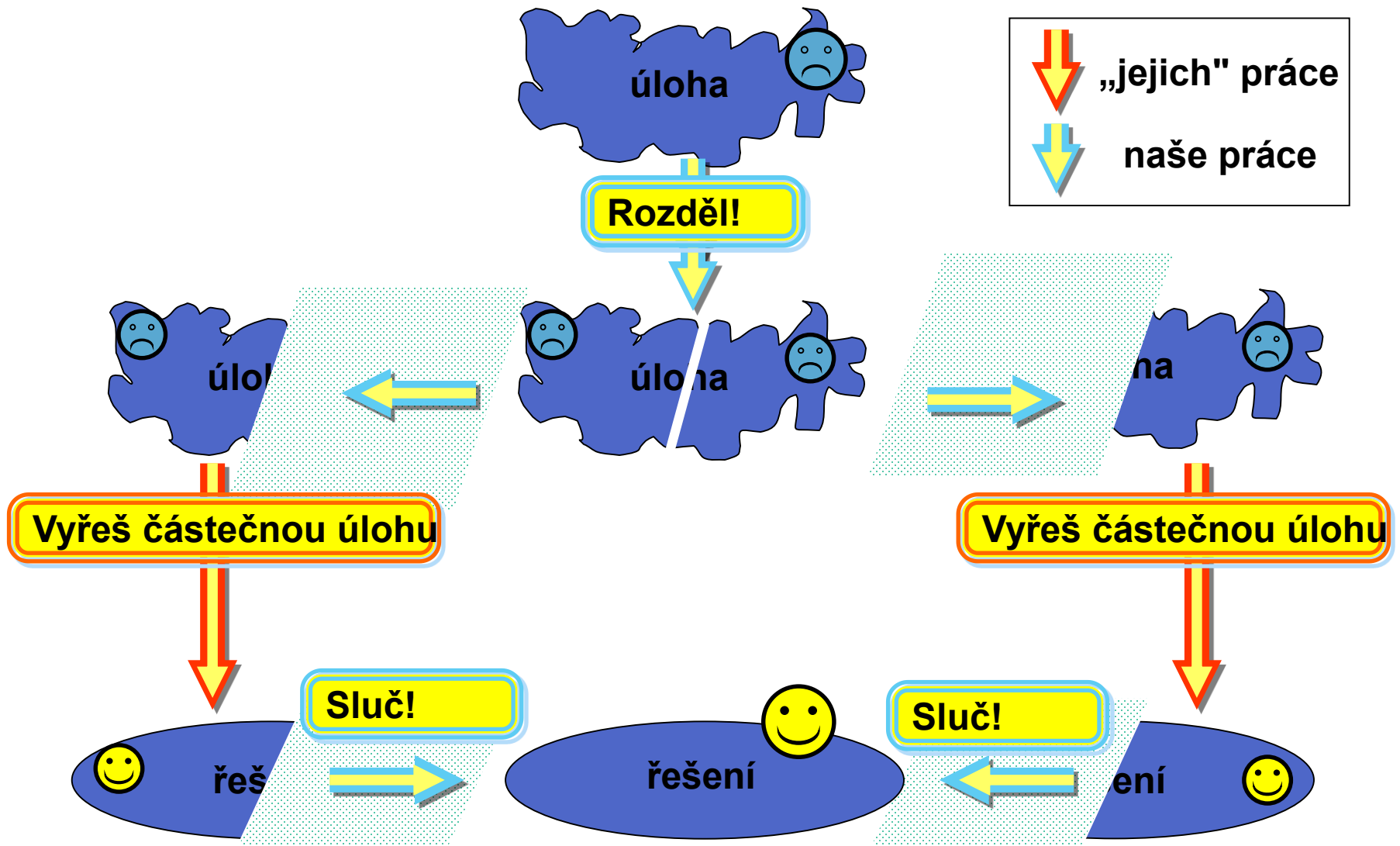
kopíruj zbytek



Seřazeno

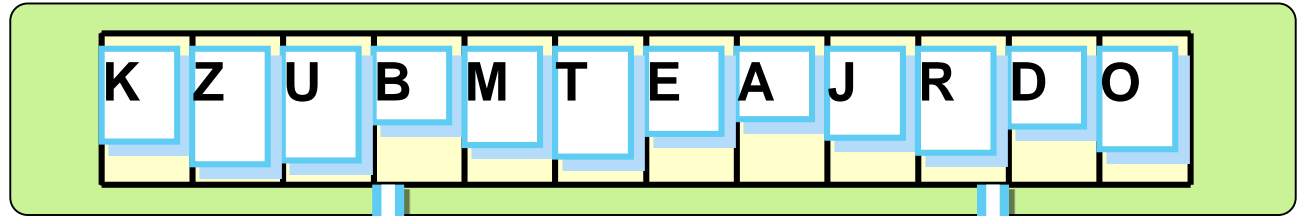


Rozděl a Panuj! Divide & Conquer! Divide et Impera!

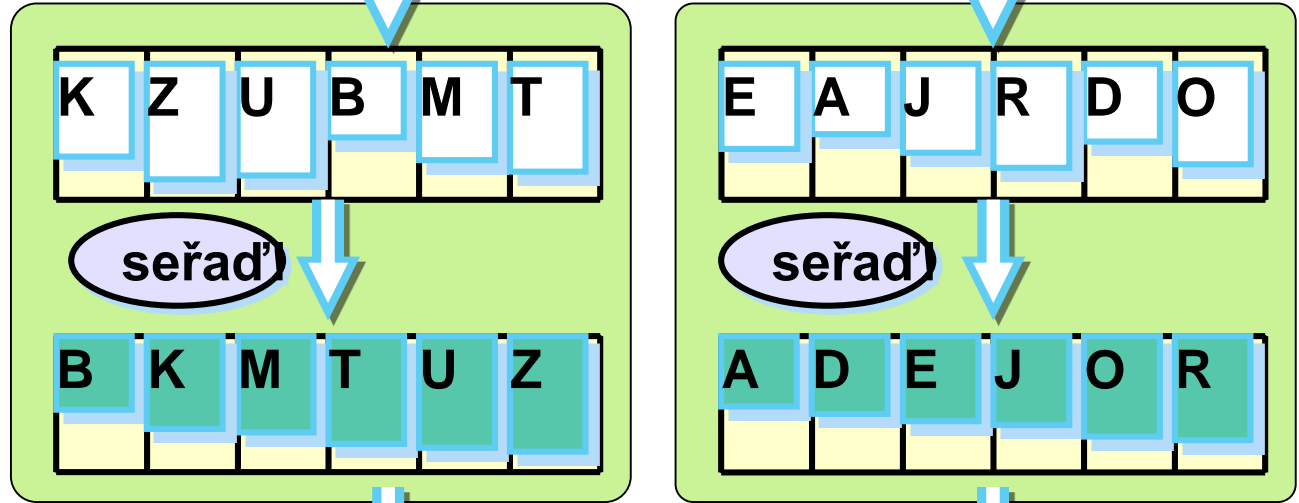


MERGE-SORT

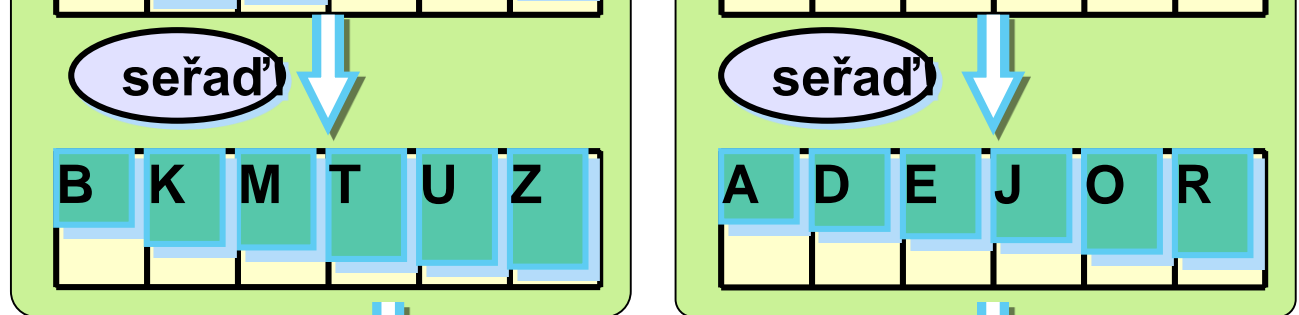
Neseřazeno



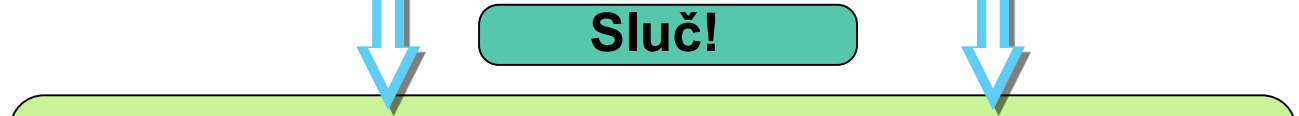
Rozděli!



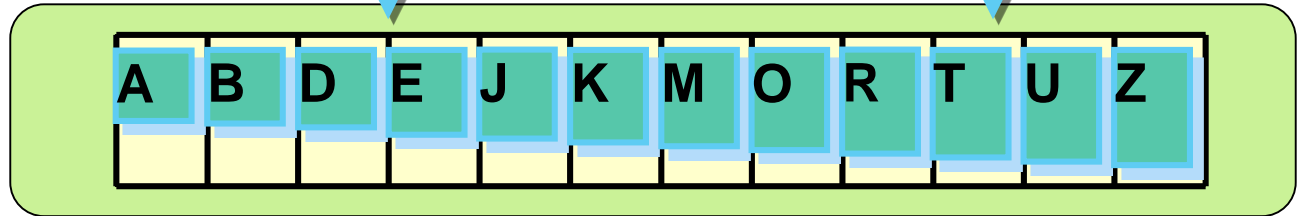
Zpracuj odděleně



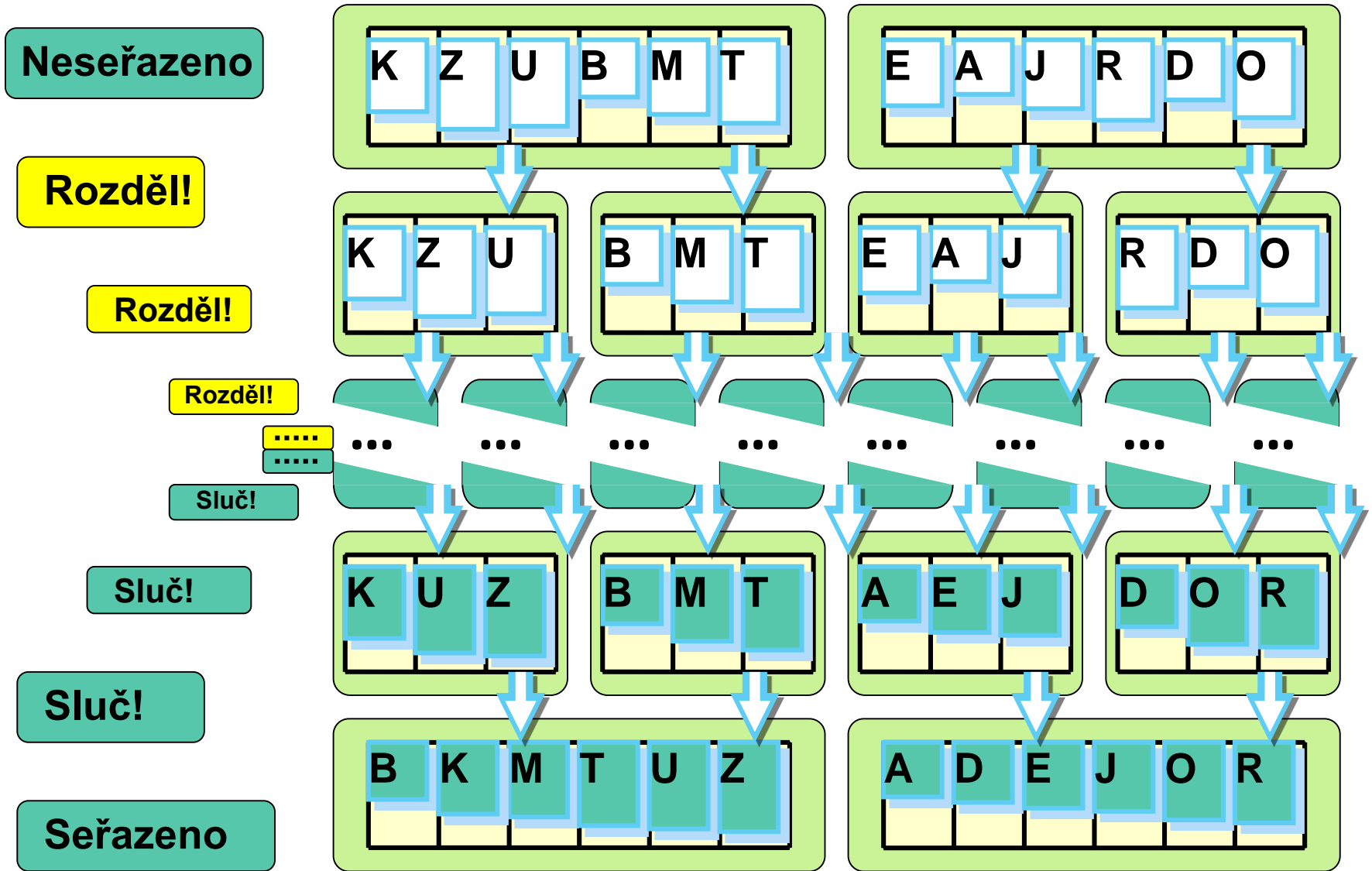
Panuj!



Seřazeno



MERGE-SORT



MERGE-SORT

```

void mergeSort (int a[], int aux[],
                int low, int high) {
    int half = (low+high)/2;
    int i;
    if (low >= high) return;           // too small!
                                        // sort
    mergeSort(a, aux, low, half);       // left half
    mergeSort(a, aux, half+1, high);    // right half
    merge(a, aux, low, high);           // merge halves

                                        // put result back to a
    for (i = low; i <= high; i++) a[i] = aux[i];

    // optimization idea:
    /* swapArray(a, aux) */ // better to swap
                            // references to a & aux!
}

```

MERGE-SORT

```
void merge(int in[], int out[], int low, int high) {
    int half = (low+high)/2;
    int i1 = low;
    int i2 = half+1;
    int j = low;

                                // compare and merge
    while ((i1 <= half) && (i2 <= high)) {
        if (in[i1] <= in[i2]) { out[j] = in[i1]; i1++; }
        else { out[j] = in[i2]; i2++; }
        j++;
    }

                                // copy the rest
    while (i1 <= half) { out[j] = in[i1]; i1++; j++; }
    while (i2 <= high) { out[j] = in[i2]; i2++; j++; }
}
```

Asymptotická složitost MERGE-SORT

Rozdě! $\log_2(n)$ krát \Rightarrow

\Rightarrow Sluč! $\log_2(n)$ krát

Rozdě! $\Theta(1)$ operací

Sluč! $\Theta(n)$ operací

Celkem $\Theta(n) \cdot \Theta(\log_2(n)) = \Theta(n \cdot \log_2(n))$ operací

Asymptotická složitost MERGE-SORT je $\Theta(n \cdot \log_2(n))$

Stabilita MERGE-SORT

Rozděl! **Nepohybuje prvky**

Sluč! **“ if (in[i1] <= in[i2]) { out[j] = in[i1]; ...”**

**Zařad' nejprve levý prvek
když slučuješ stejné hodnoty**

MERGE-SORT je stabilní

Optimalita algoritmů řazení

Definice:

- Algoritmus je **optimální**, pokud má v nejhorším případě složitost rovnou asymptoticky spodní mezi složitosti řešení daného problému.

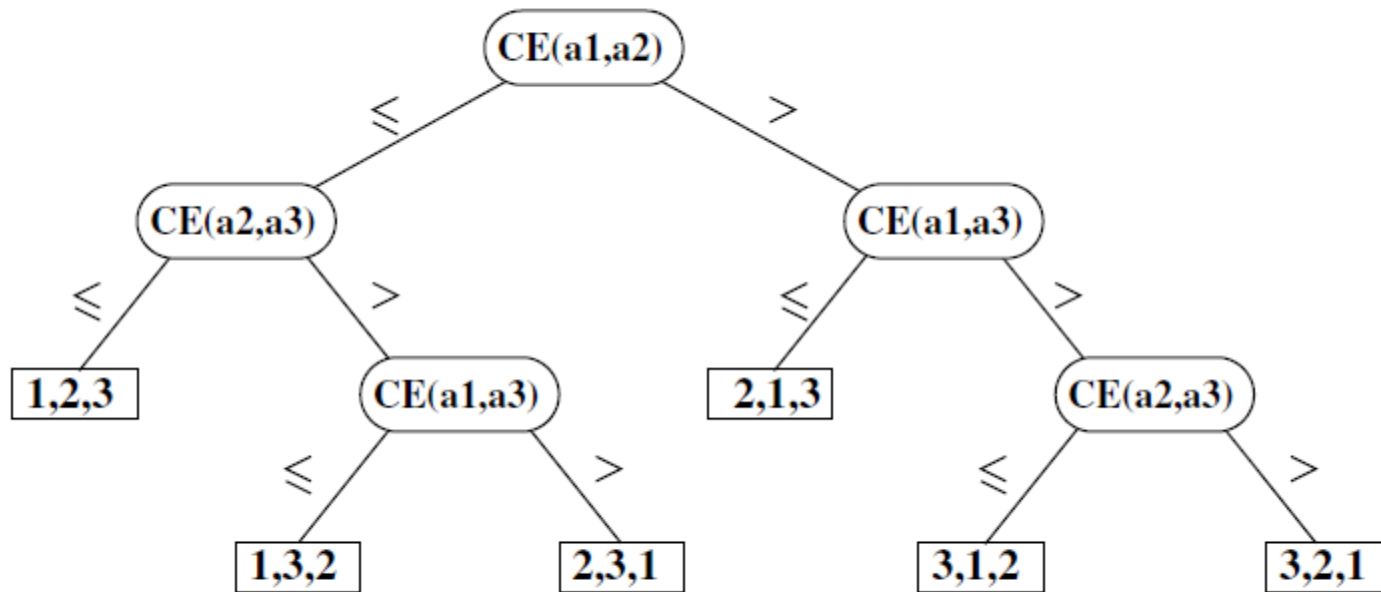
Věta:

- Spodní mez složitosti asociativního řešení problému řazení n čísel je $\Omega(n \log n)$. Asociativní řazení využívá porovnání a případného přesunu (tzv. CE operace – Compare and Exchange).

Důkaz:

- Uvažujme pouze vstupy s n různými čísly $A = \langle a_1, \dots, a_n \rangle$. Libovolný CE algoritmus S lze abstraktně popsat pomocí rozhodovacího stromu $RS(S, n)$:
 - Každý vnitřní uzel stromu odpovídá $CE(a_i, a_j)$ pro nějaká $1 \leq i \neq j \leq n$.
 - Levý podstrom diktuje následné CE operace, pokud $a_i \leq a_j$ nebo je to list stromu.
 - Pravý podstrom diktuje následné CE operace, pokud $a_i > a_j$ nebo to je list stromu.

Příklad rozhodovacího stromu pro $n=3$



Optimalita algoritmů řazení (pokr.)

Důkaz (pokr.):

- Každý list odpovídá právě jedné permutaci $\pi(A)$, kde: $a_{\pi(1)} a_{\pi(2)} \cdots a_{\pi(n)}$.
- Každému provedení algoritmu S odpovídá právě jeden průchod stromem z kořene do nějakého listu.
- Pro libovolný algoritmus S je odpovídající strom plný binární strom s $n!$ listy - jeho hloubka je nejméně $\log(n!) = \Theta(n \log n)$.

Důsledek:

- Algoritmy **Merge-Sort** v každém případě, **Heap-Sort** v nejhorším a **Quick-Sort** v průměrném případě jsou optimální.

Poznámka:

$$\log(n!) = \log(n \cdot (n-1) \cdot \dots \cdot 1) = \log(n) + \log(n-1) + \dots + \log(1) \leq n \log n \Rightarrow \log(n!) = O(n \log n)$$

$$\log(n!) \geq \log(n/2) + \dots + \log(n) = n/2 \cdot \log n/2 \Rightarrow \log(n!) = \Omega(n \log n)$$

$$\log(n!) = O(n \log n) \wedge \log(n!) = \Omega(n \log n) \Rightarrow \log(n!) = \Theta(n \log n)$$

Speciální algoritmy řazení v čase $O(n)$

- Vzhledem k výše popsané nepřekročitelné mezi $\Omega(n \log n)$ lze seřadit n čísel v čase menším, pouze pokud:
 - Je na vstupní hodnoty uvalena dodatečná omezující podmínka a díky tomu můžeme využít nějaké sofistikované vylepšení.
 - Řazení není založeno na porovnání (operaci CE - složitost algoritmu se počítá pomocí základních operací).
- Uvedeme si 3 takové algoritmy:
 - RADIX-SORT
 - COUNTING-SORT
 - BUCKET-SORT

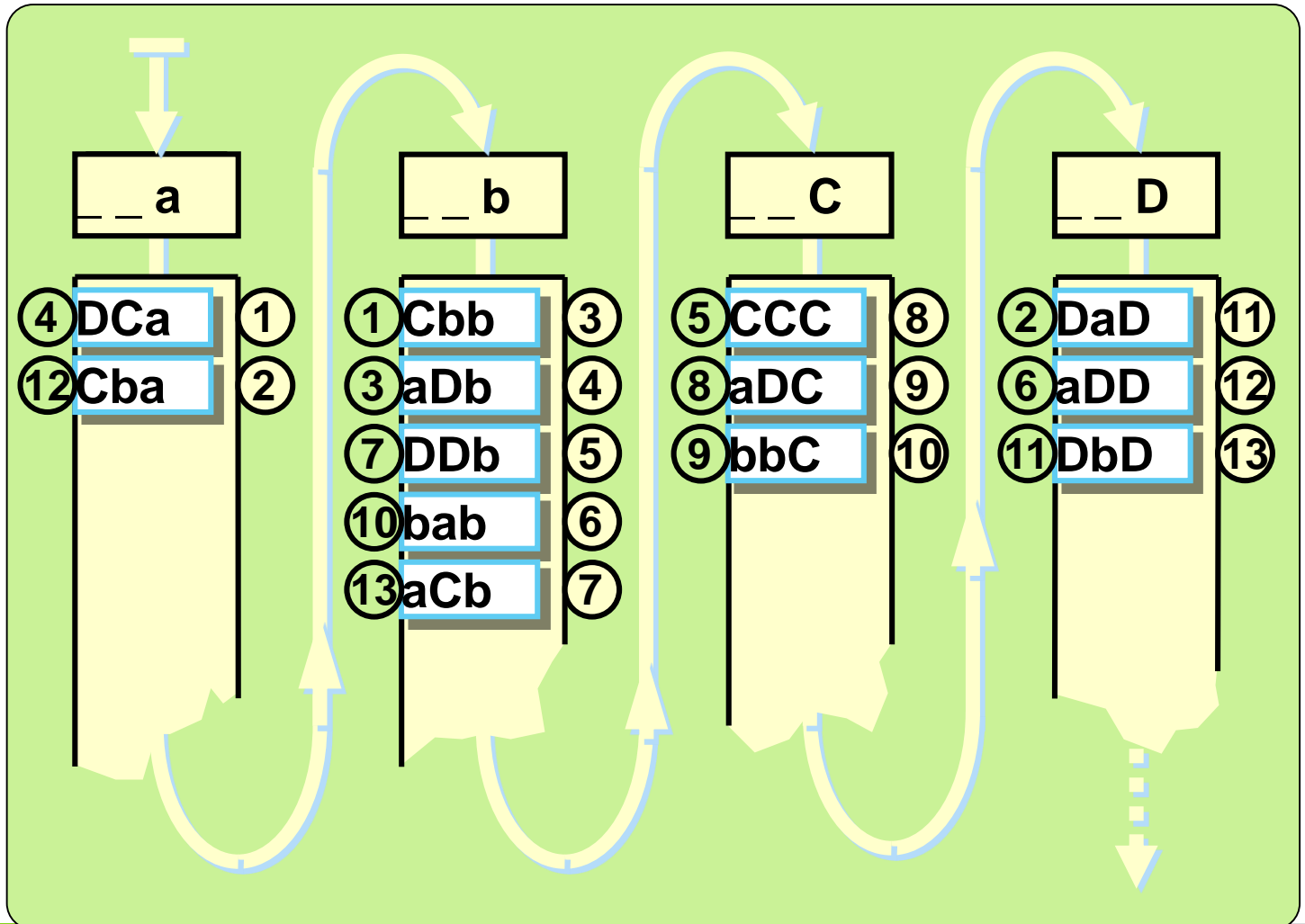
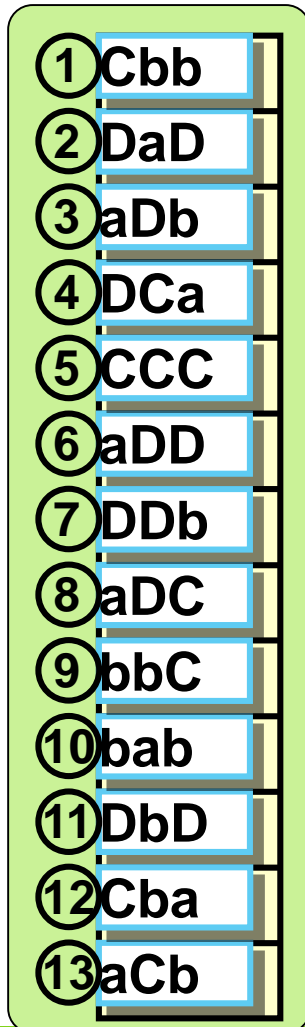
Číslicové řazení

RADIX-SORT

RADIX-SORT

Neseřazeno

řad' podle 3. znaku

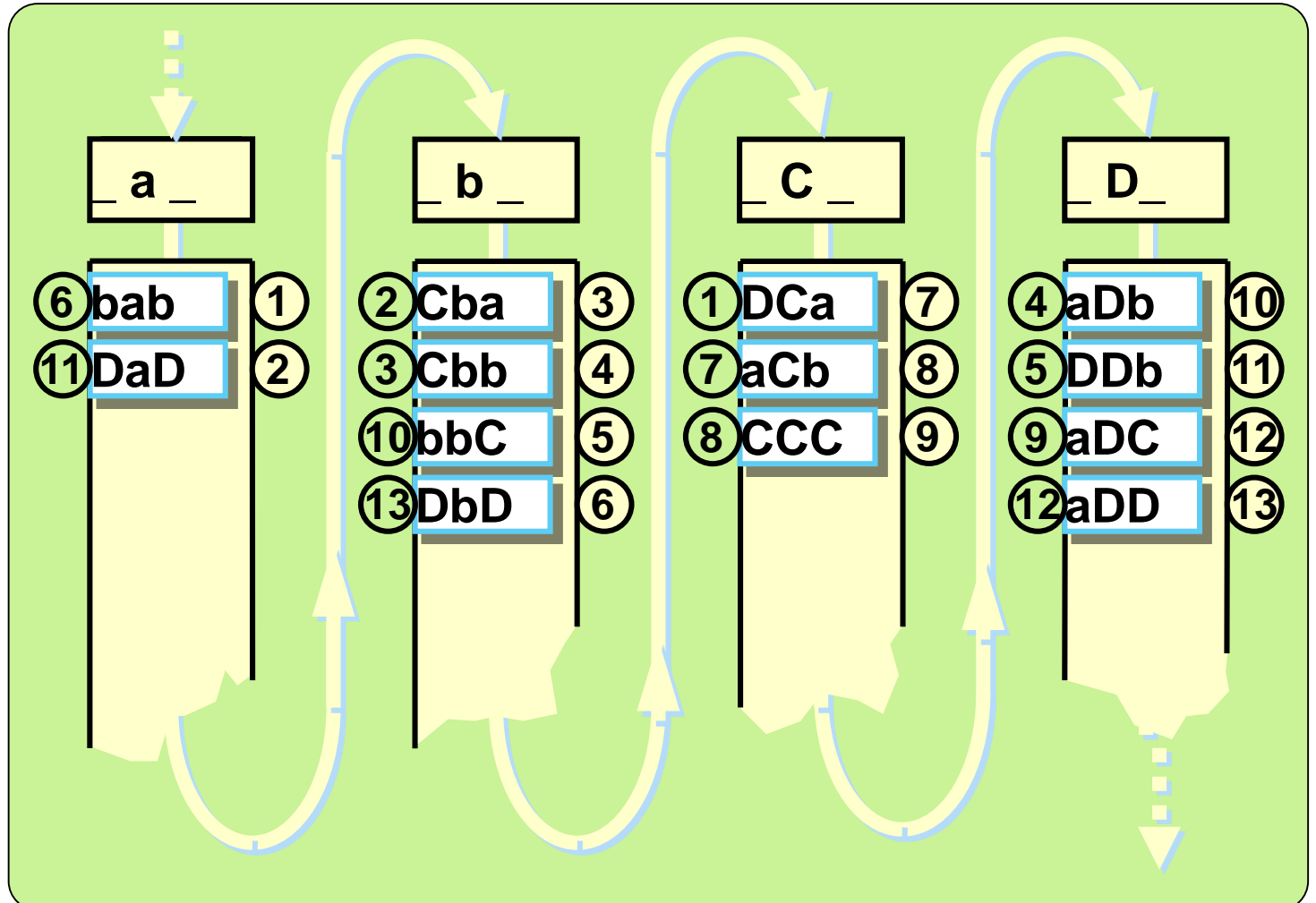


RADIX-SORT

Seřazeno
od 3. znaku

řad' podle 2. znaku

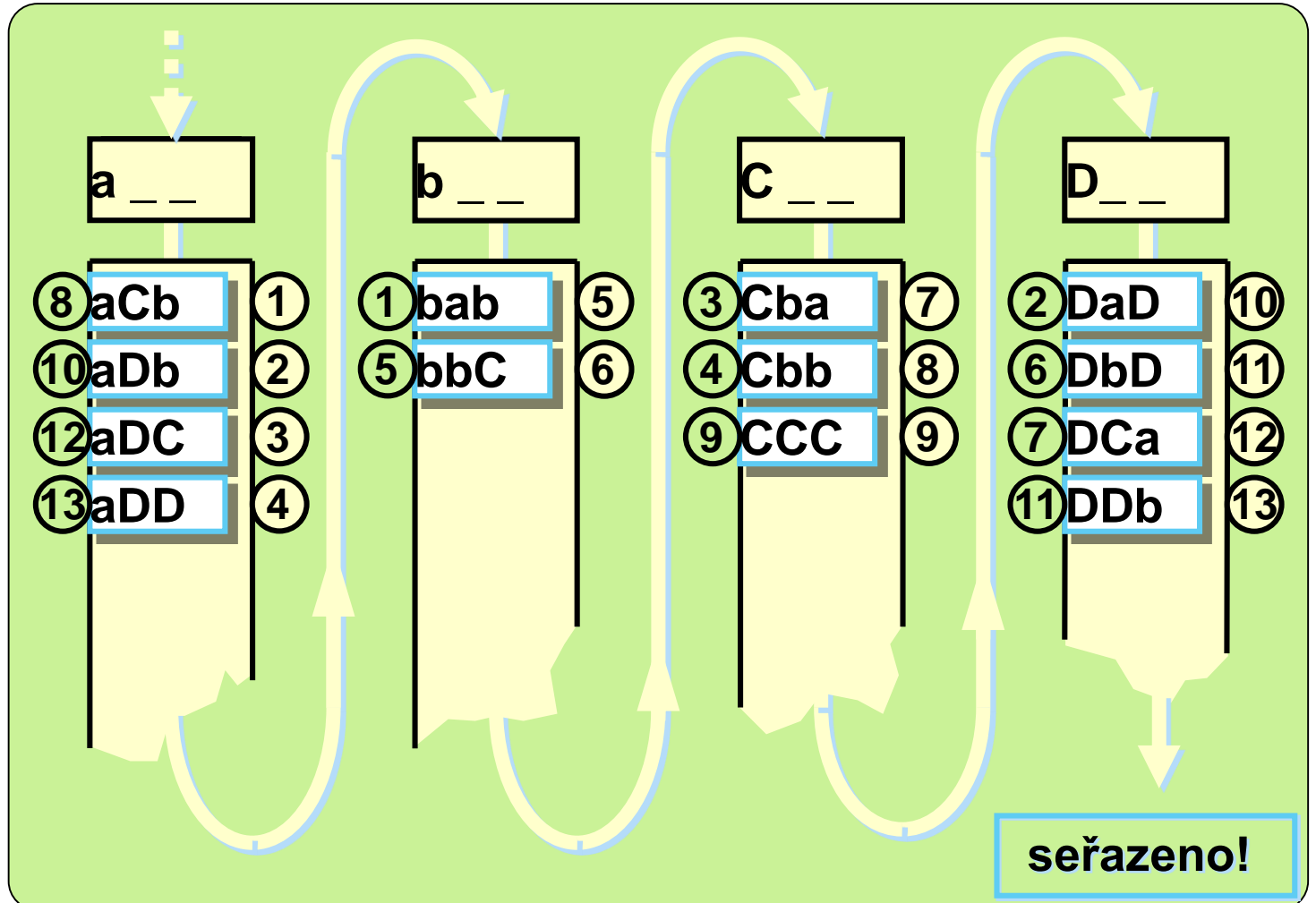
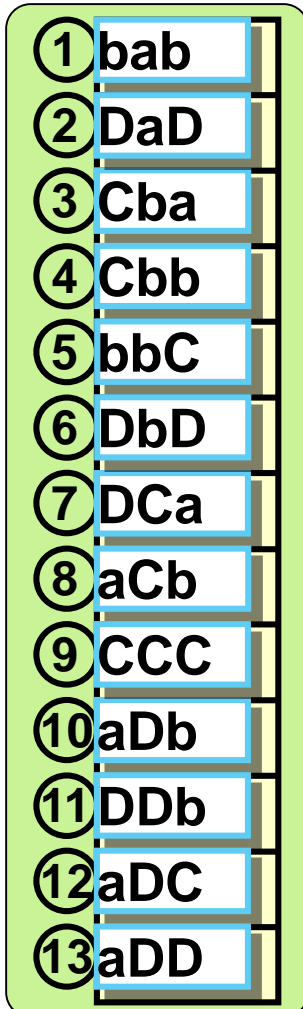
- ① DCa
- ② Cba
- ③ Cbb
- ④ aDb
- ⑤ DDb
- ⑥ bab
- ⑦ aCb
- ⑧ CCC
- ⑨ aDC
- ⑩ bbC
- ⑪ DaD
- ⑫ aDD
- ⑬ DbD



RADIX-SORT

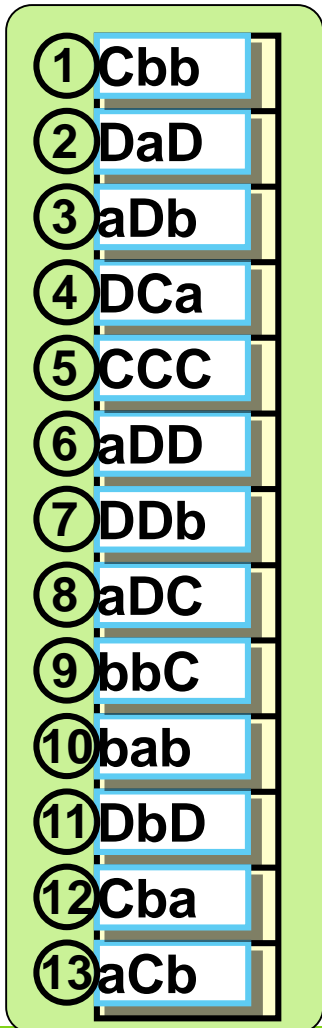
Seřazeno od 2. znaku

řad' podle 1. znaku

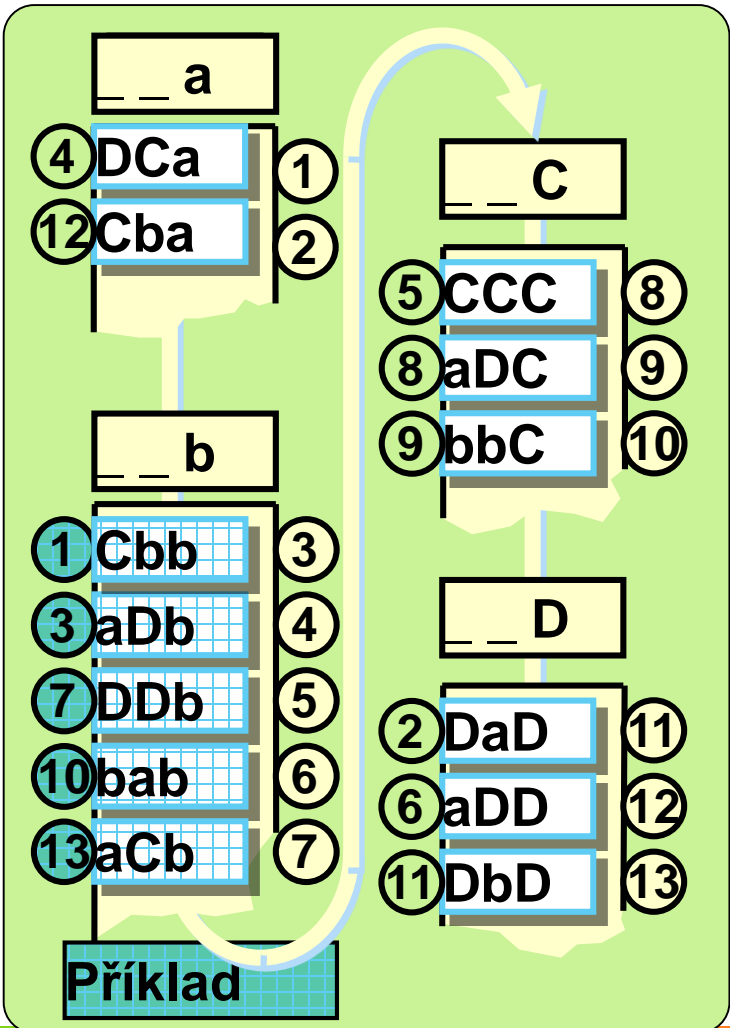


Implementace RADIX-SORT

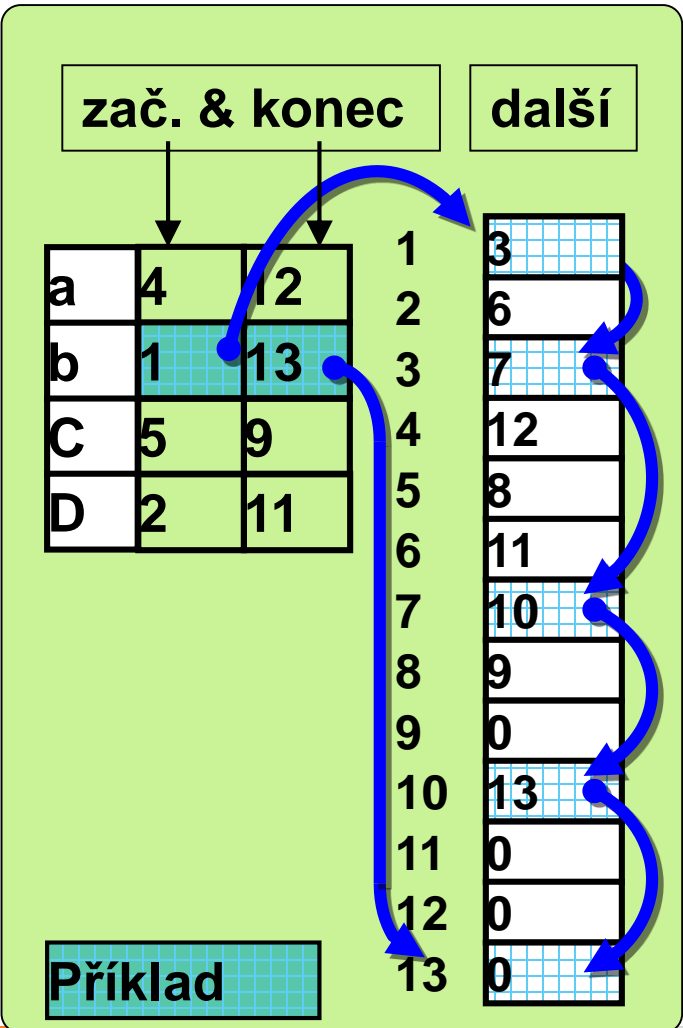
Neseřazeno



seřazeno dle 3. znaku



pomocná pole indexů



Příklad běhu RADIX-SORT

$$n = 9, d = 3, k = 4$$

Vstup	$i = 1$	$i = 2$	$i = 3$	$i = 4$
2101	012 0	21 0 1	1 0 01	0 021
1022	210 1	10 0 1	0 0 21	0 120
2211	22 1 1	12 0 2	1 0 22	1 001
0120	00 2 1	22 1 1	2 1 01	1 022
0021	11 1 1	11 1 1	1 1 11	1 111
1202	21 2 1	01 2 0	0 1 20	1 202
1111	10 0 1	00 2 1	2 1 21	2 101
2121	10 2 2	21 2 1	1 2 02	2 121
1001	12 0 2	10 2 2	2 2 11	2 211

Pseudokód RADIX-SORT

Předpoklady:

- ① Každé vstupní číslo a_i se dá vyjádřit pomocí k číslic v d -ární poziční soustavě.
- ② Pozice se číslují LSD= 1 až MSD= k .

procedure RADIXSORT(A, k)

{// A vstupní pole, případně i výstupní (záleží na implem. (2))

(1) **for** $i \leftarrow 1$ **to** k

(2) **do** seřad' čísla v A podle číslic na pozici i
 pomocí libovolného **stabilního řazení** STABLESORT

}

Možná aplikace: řazení postupně podle několika klíčů, např. den, měsíc, rok.

Korektnost RADIX-SORT

Věta:

- Algoritmus RADIX-SORT řadí vstupní čísla správně a je stabilní.

Důkaz:

- Indukcí podle čísla pozice.
- Dokážeme indukční hypotézu: Po provedení i iterací **for** cyklu platí, že vstupní pole hodnot ořezaných na nižších i -pozicích je správně seřazené.
- Základ indukce: Platí evidentně pro první iteraci, kdy se nahoru přesunou čísla končící číslicí 0, pak následují čísla končící číslicí 1, atd.
- Indukční krok: Předpokládejme, že uvedené tvrzení platí pro j , $1 \leq j < k$.
 - Řazení v $(i = j + 1)$ -té iteraci posouvá směrem nahoru nejprve čísla mající na pozici i číslici 0. Protože tato čísla oříznutá na posledních $i - 1$ pozic byla v předchozích iteracích seřazena správně, bude tato skupina čísel oříznutých na i pozic také seřazena správně.
 - Pak bude podobným způsobem následovat skupina čísel, majících na pozici i číslici 1, po té číslicí 2, atd.
 - Po skončení i -té iterace cyklu budou tedy čísla oříznutá na i pozic seřazena správně.

Shrnutí: RADIX-SORT

d znaků d cyklů

cyklus $\Theta(n)$ operací

celkem $\Theta(d \cdot n)$ operací

$d \ll n \Rightarrow$ $\Theta(n)$ operací

Číslicové řazení nemění pořadí stejných hodnot

Asymptotická složitost Radix sortu je $\Theta(n)$

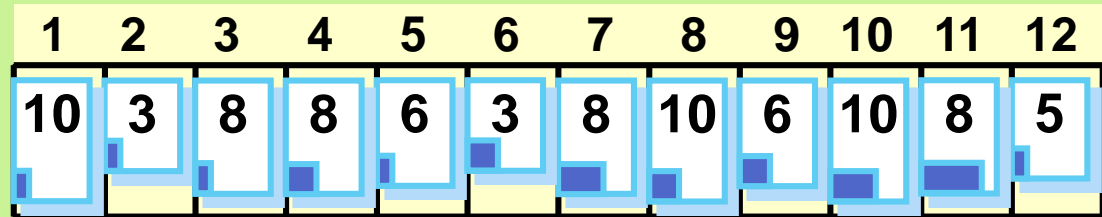
Je to stabilní řazení

Řazení čítáním

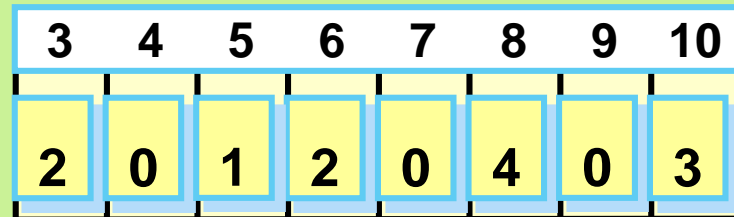
COUNTING-SORT

COUNTING-SORT

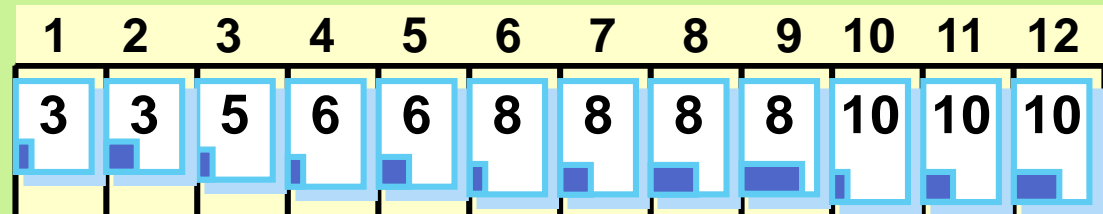
vstup
vstup.length == N



četnost
četnost.length == k
 $k = \max(\text{vstup}) - \min(\text{vstup}) + 1$



výstup
vstup.length == N



COUNTING-SORT

Krok 1

vynulování pole četností

3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0

jeden průchod vstupním polem

10	3	8	8	6	3	8	10	6	10	8	5
----	---	---	---	---	---	---	----	---	----	---	---

naplnění pole četností

3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

COUNTING-SORT

Krok 2

jeden průchod

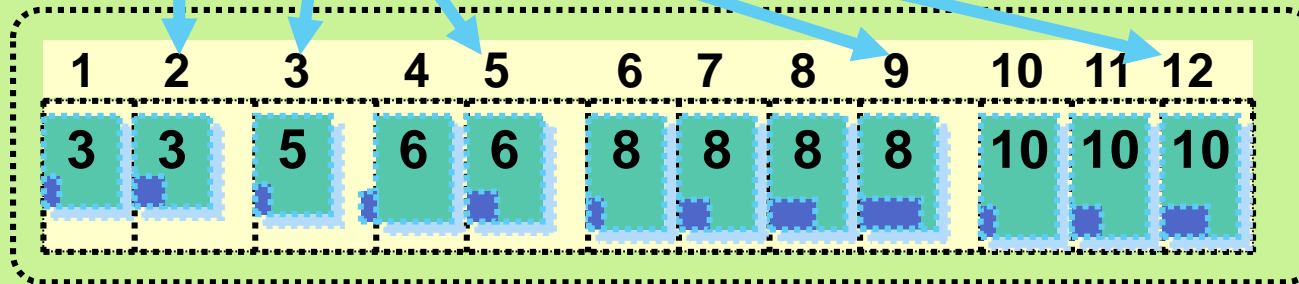
3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

úprava pole četností

```
for (i = dMez+1; i <= hMez; i++)
  cetnost[i] += cetnost[i-1]
```

3	4	5	6	7	8	9	10
2	2	3	5	5	9	9	12

Prvek `cetnost[j]` obsahuje pozici posledního prvku s hodnotou `j` v budoucím výstupním poli.



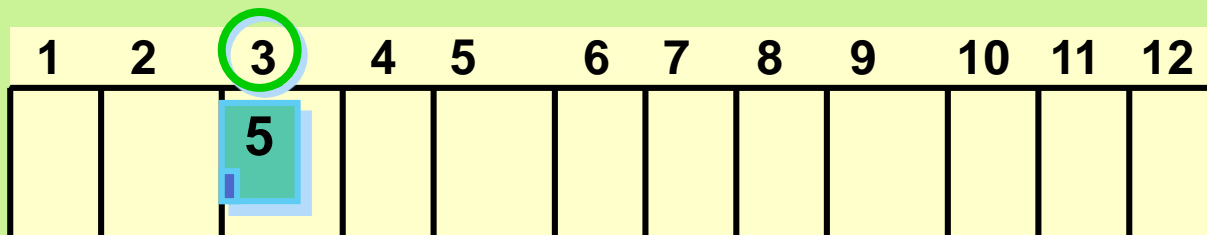
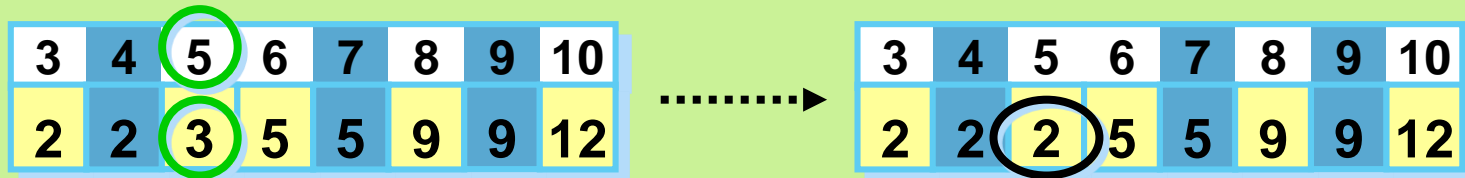
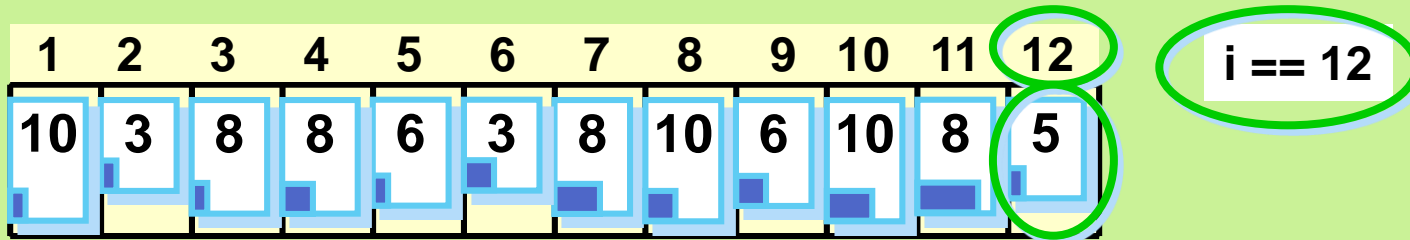
COUNTING-SORT

Krok 3

$i == N$

```

for(i = N; i > 0; i--) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
    
```



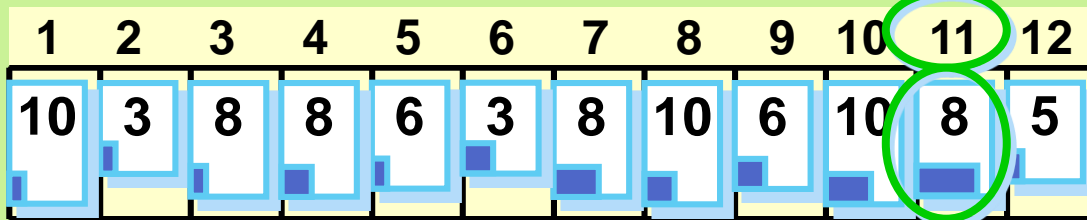
COUNTING-SORT

Krok 3

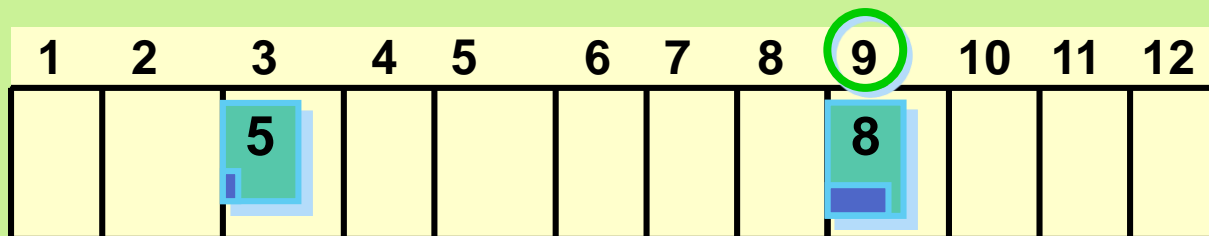
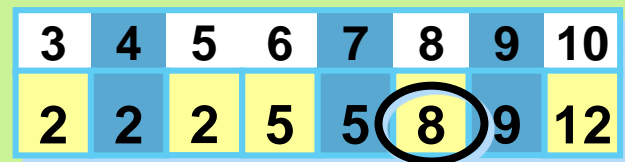
$i == N-1$

```

for(i = N; i > 0; i--) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
    
```



$i == 11$



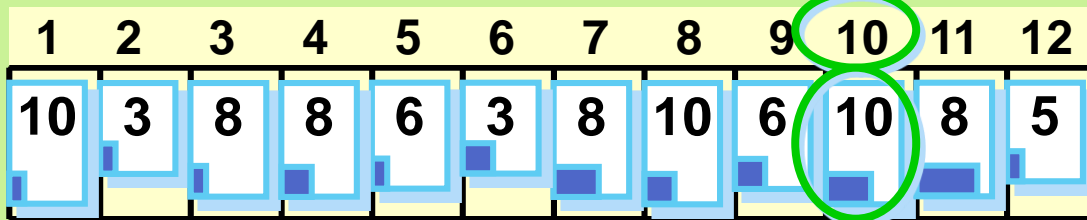
COUNTING-SORT

Krok 3

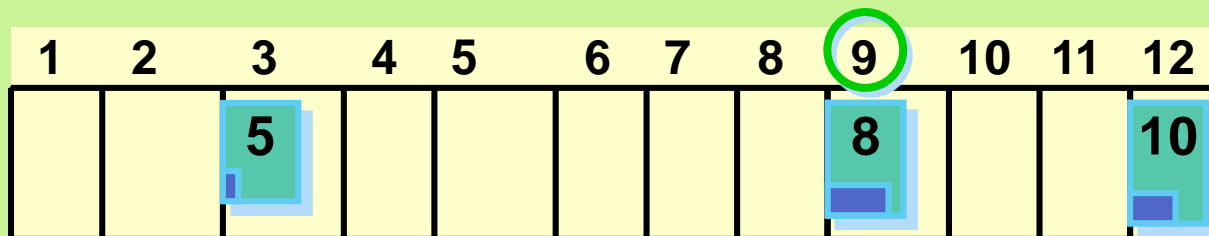
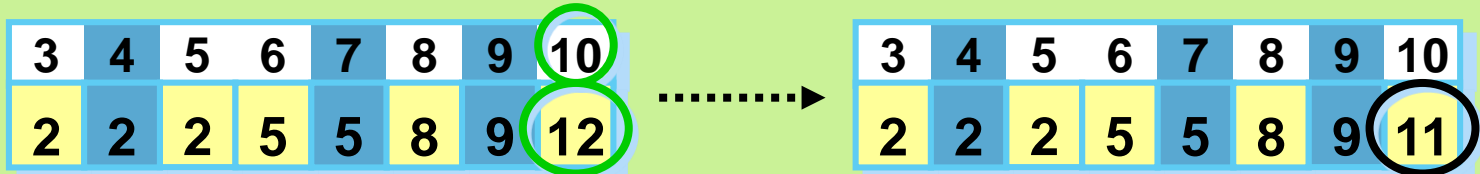
$i == N-2$

```

for(i = N; i > 0; i--) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
    
```



$i == 10$



atd...

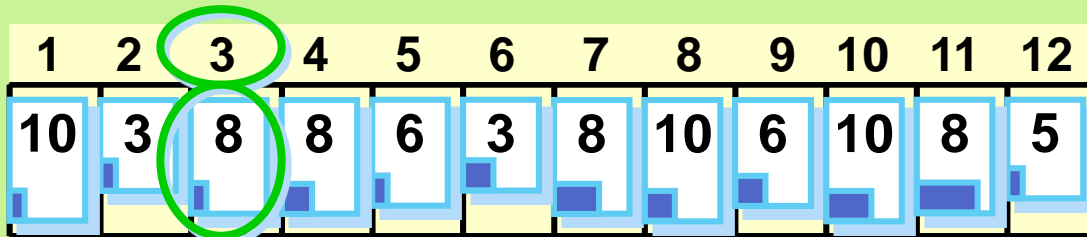
COUNTING-SORT

Krok 3

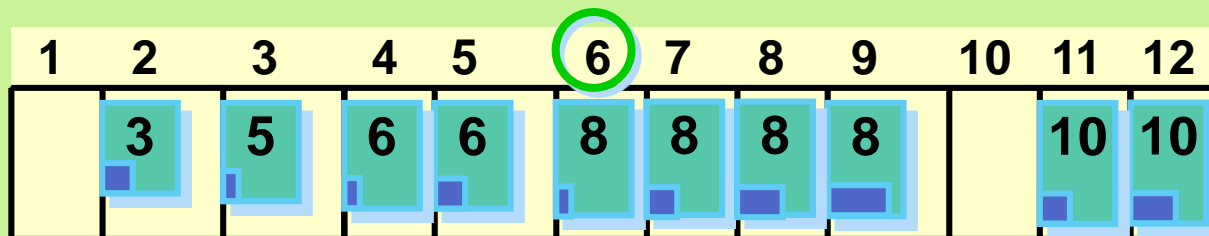
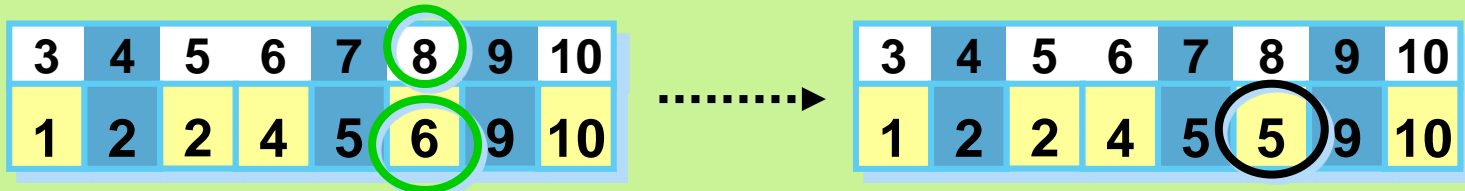
$i == 3$

```

for(i = N; i > 0; i--) {
    vystup[cetnost[vstup[i]]] = vstup[i];
    cetnost[vstup[i]]--;
}
    
```



$i == 3$



atd...

Shrnutí pro COUNTING-SORT

Krok 1 ... $\Theta(N)$ operací

Krok 2 ... $\Theta(k)$ operací

Krok 3 ... $\Theta(N)$ operací

Celkem ... $\Theta(k + N)$ operací

Pokud $k \leq N$,
je asymptotická složitost COUNTING-SORT $\Theta(N)$

COUNTING-SORT je stabilní

N je velikost vstupního pole,
 k je velikost intervalu, v němž všechny vstupní hodnoty leží.

Přihrádkové řazení

BUCKET-SORT

BUCKET-SORT

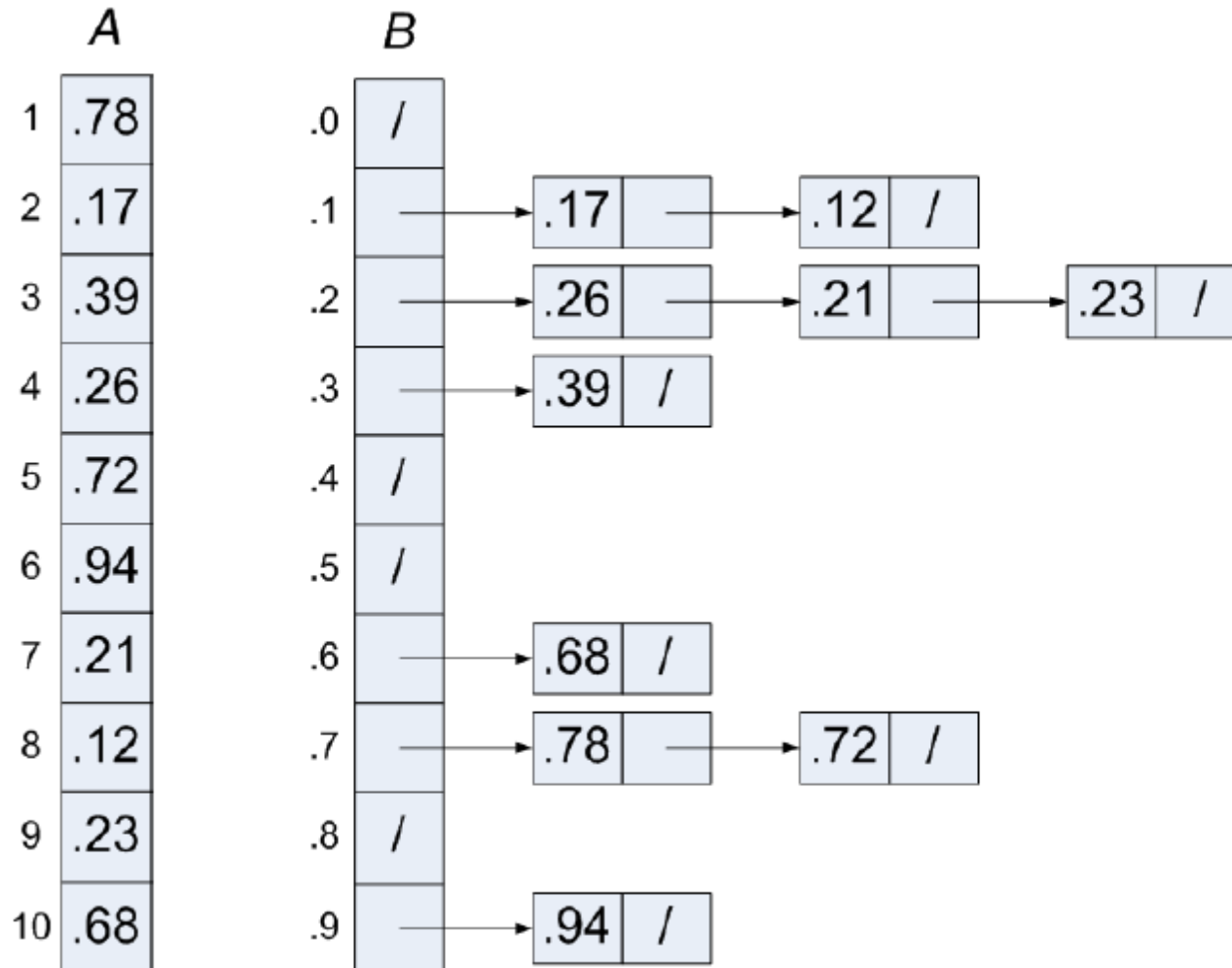
- Vstupní čísla jsou z intervalu $[0, 1)$. Algoritmus je vhodný, pokud jsou vstupní čísla rovnoměrně rozložena v tomto intervalu.
- Hlavní myšlenky:
 - Interval $[0, 1)$ rozdělíme na n přihrádek, do kterých vstupní pole distribuujeme.
 - Díky předpokladu rovnoměrnosti rozdělení předpokládáme, že do jedné přihrádky spadne málo čísel.
 - Nicméně, protože velikost přihrádky nelze dopředu přesně určit, je implementována jako dynamický zřetězený seznam.
 - Každou přihrádku rychle seřadíme a výsledek dostaneme zřetězením seřazených přihrádek.

```

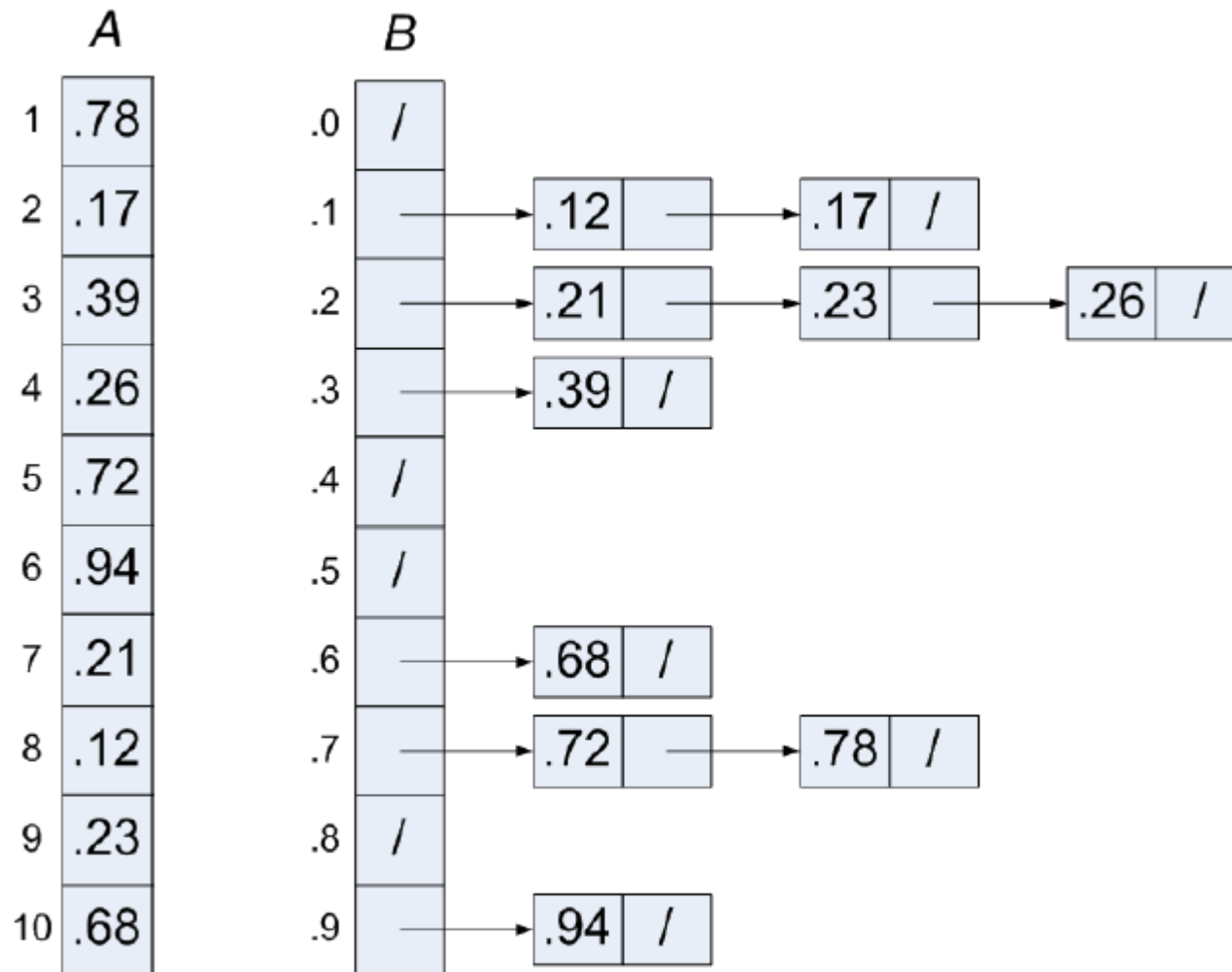
procedure BUCKETSORT( $A, B$ )
  { //  $A$  vstupní pole,  $B$  pole přihrádek
  (1)  $n \leftarrow \text{length}(A)$ ;
  (2) for  $i \leftarrow 1$  to  $n$ 
  (3)   do připoj  $A[i]$  k seznamu v přihrádce  $B[\lfloor n * A[i] \rfloor]$ ;
  (4) for  $i \leftarrow 0$  to  $n - 1$ 
  (5)   do seřaď seznam  $B[i]$  stabilním alg. (např. INSERTIONSORTem)
  (6) zřetěz seznamy  $B[0], B[1], B[2], \dots, B[n - 1]$  do  $A$ .
  }

```

Příklad běhu BUCKET-SORT



Příklad běhu BUCKET-SORT (pokr.)



Korektnost a stabilita BUCKET-SORT

Věta:

- Algoritmus BUCKET-SORT řadí vstupní čísla správně a je stabilní.

Důkaz:

- Uvažujme 2 vstupní hodnoty $A[i]$ a $A[j]$.
- Pokud padnou do stejné přihrádky, pak jsou v ní seřazeny a objeví se na výstupu ve správném pořadí.
- Dvě stejné vstupní hodnoty díky stabilitě řazení v přihrádkách nezmění pořadí.
- Necht' padnou do různých přihrádek $B[i']$ a $B[j']$. Necht' $i' < j'$.
- Funkce $f(x) = \lfloor n * x \rfloor$ je ale neklesající funkce: $f(x) < f(y) \Rightarrow x < y$. Čili:
$$i' = \lfloor n * A[i] \rfloor < j' = \lfloor n * A[j] \rfloor \Rightarrow A[i] < A[j].$$

Shrnutí pro BUCKET-SORT

Asymptotická časová složitost BUCKET-SORT je v nejhorším případě $\Theta(n^2)$, kde n je délka řazené posloupnosti.

Průměrná očekávaná časová složitost je $\Theta(n + k)$, kde k je počet přihrádek.

Pokud $k \leq n$, pak průměrná očekávaná složitost je $\Theta(n)$.

Asymptotická paměťová složitost v nejhorším případě je $\Theta(n \cdot k)$.

BUCKET-SORT je stabilní (pokud je stabilní řazení v přihrádkách)

ŘAZENÍ NA VNĚJŠÍCH MÉDIÍCH

Řazení na vnějších médiích

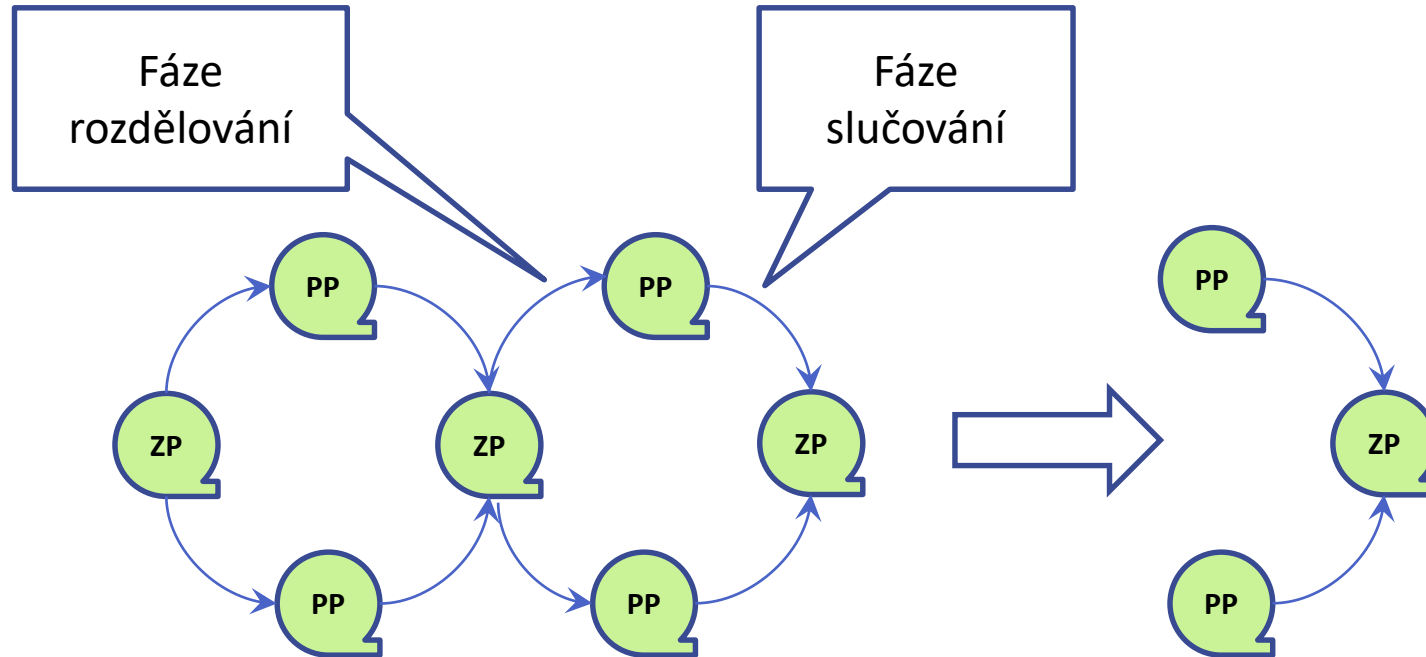
- Metody řazení lze rozlišit na tzv. **vnitřní** řazení (řazení v paměti) a tzv. **vnější** řazení (využívající vnější média).
- **Vnější** řazení lze dále rozlišit na práci s médii se **sekvenčním** přístupem (páska) a **náhodným** přístupem (disk).
- Základním principem řadících metod na vnějších, sekvenčně organizovaných paměťových médiích, je řazení slučováním (merging).
- Pro řazení na vnějších médiích s náhodným přístupem se využívají B-stromy, příp. řazení dělením.

Metoda přímého slučování (STRAIGHT-MERGING)

Tato metoda potřebuje kromě souboru, v němž jsou řazená data, dva další pomocné soubory.

- V prvním kroku se data původního (zdrojového) souboru rozdělí rovnoměrně do dvou pomocných souborů.
- Ve druhém kroku se přečte z každého pomocného souboru jeden prvek, vytvoří se uspořádaná dvojice, která se uloží do zdrojového souboru. Tato akce se opakuje až se ze všech údajů vytvoří soubor uspořádaných dvojic.
- Ve třetím kroku se uspořádané dvojice rozdělí rovnoměrně do dvou pomocných souborů.
- Ve čtvrtém kroku se ze dvou souborů seřazených dvojic vytvoří soubor seřazených čtveřic atd. Tento cyklus se opakuje pro všechny 2^k -tice až po k , pro něž platí, že $2^{k+1} \geq n$. Jako poslední sloučíme poslední 2^k -tici a zbývající seřazenou posloupnost ve druhém souboru.
- Protože typickou implementací vnějšího sekvenčního souboru je magnetická páska, říká se této metodě také přímá metoda tří pásek.

Schéma postupu třípáskových metod



ZP – zdrojová páska, PP – pomocná páska

Algoritmus řazení přímým slučováním

Rozděl prvky souboru f_1 do pomocných souborů f_2 a f_3 ;

$n := 0$;

repeat

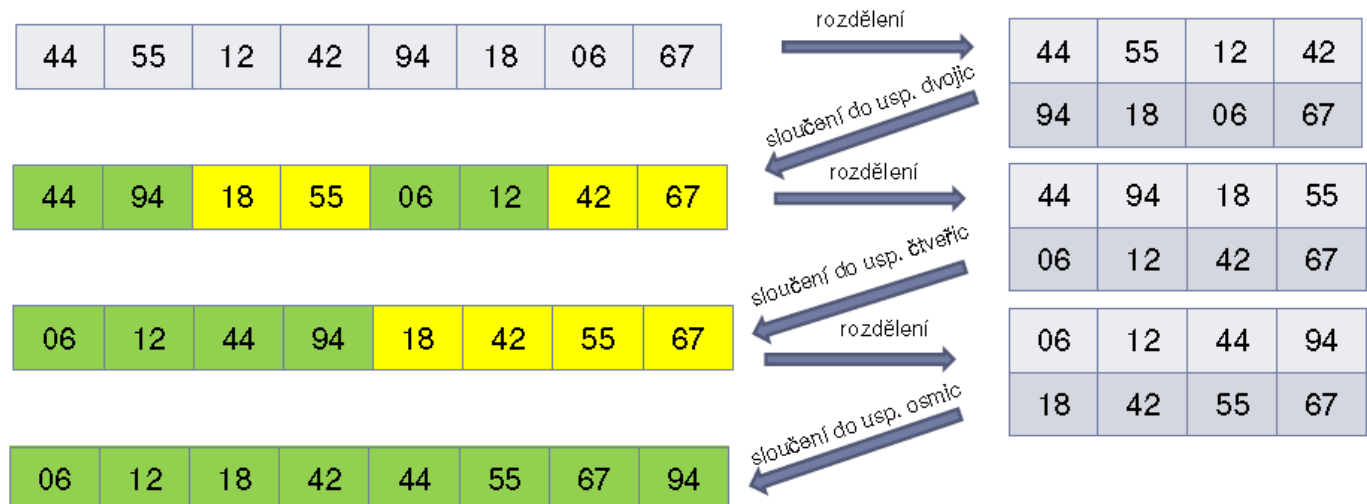
$i := 0$;

Sloučením posloupností o délce 2^n z pásek f_2 a f_3 vytvoř posloupnosti o délce 2^{n+1} a ulož je do souboru f_1 . S každou vytvořenou posloupností zvyš i o 1.

if $i > 1$ then

Rozděl posloupnosti o délce 2^{n+1} ze souboru f_1 do souborů f_2 a f_3

until $i = 1$;



Přirozené slučování (natural merging)

- Na rozdíl od předchozí přímé metody tří pásek, přirozená metoda slučování pomocí tří pásek nevytváří seřazené posloupnosti o délce 2^n při vzrůstajícím n , ale využívá počátečního uspořádání řazeného souboru. V první fázi rozdělování zapisuje střídavě do dvou pomocných souborů celé neklesající posloupnosti.
- Dále se již střídá fáze slučování a rozdělování s tím, že při slučování algoritmus musí rozeznat konec neklesající posloupnosti. Proces končí sloučením posledních dvou posloupností do zdrojového souboru.
- Přirozená metoda čtyř pásek (také přirozené vyvážené slučování, angl. natural balanced merging) podobně jako u přímého slučování, zkracuje proces řazení vyloučením fází rozdělení za cenu použití dalšího pomocného souboru (pásky). V rámci slučovacího procesu se uspořádané n -tice rovnou zapisují do dvou souborů (rovnou se rozdělují).
- Nelze-li u čtyřpáskové metody umístit v závěrečném slučování výsledný seřazený soubor na původní zdrojovou pásku, vloží se do procesu řazení kopírovací cyklus, který umístí seřazený soubor na původní pásku.

Demo pro různé metody řazení

<http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html>

The End