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

Datové struktury a algoritmy, B6B36DSA
04/2017, Lekce 7

https://moodle.fel.cvut.cz/course/view.php?id=1238

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 dělící poměr na všech úrovních bude stejný. Pravděpodobně některá dělení budou vyvážená a některá nevyvážená.
- Model průměrného případu:
  - např. střídání nejlepšího (1/1) a nejhoršího dělícího průměru (1/n − 1),
  - vyjde jenom o něco hůře než nejlepší.
- Otočíme pohled: místo modelování a spekulování o dělících poměrech vnutíme vstupním datum 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 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.)

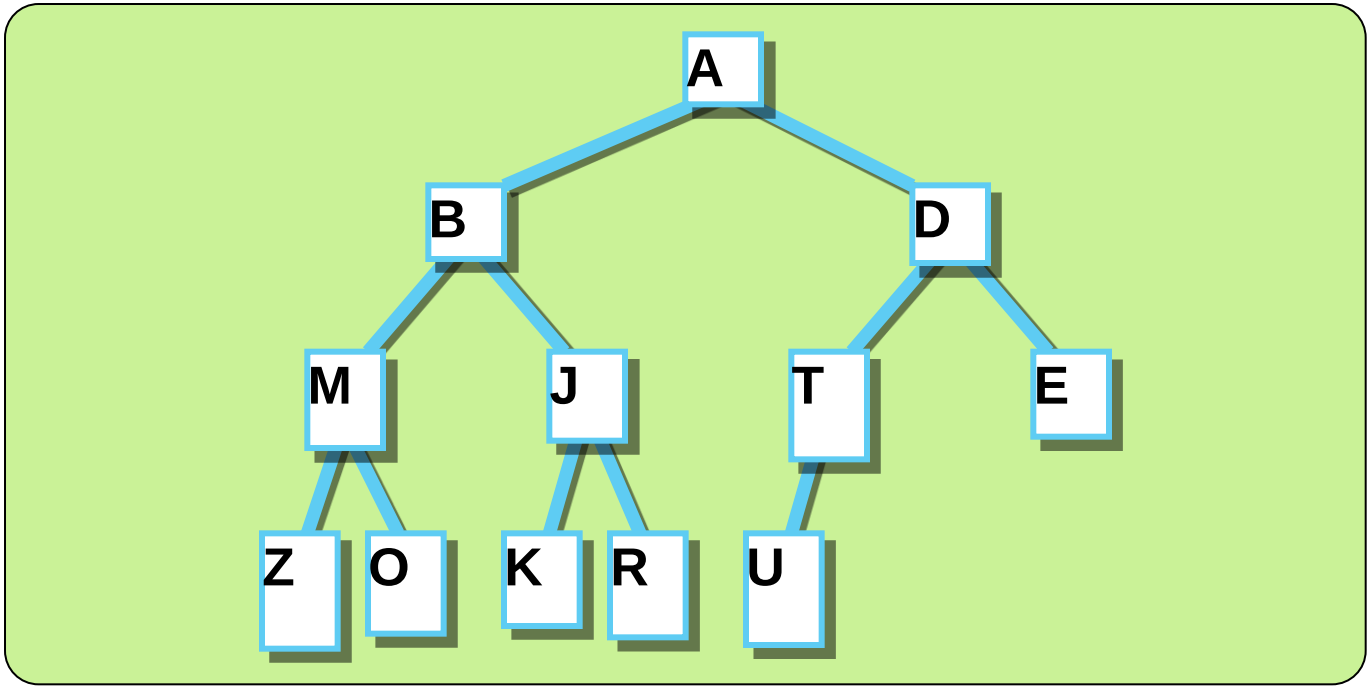
- Časová složitost v nejhorším případě $\Theta(n^2)$.
- Časová složitost v průměrném (a očekávaném) případě $\Theta(n \log n)$.
- Nízká multiplikativní konstanta.
- Potřebuje případně jen konstantní prostor navíc.
- Správnost = částečná správnost + terminace
  - Částečná správnost (invariant)?
  - Terminace?

Řazení pomocí haldy

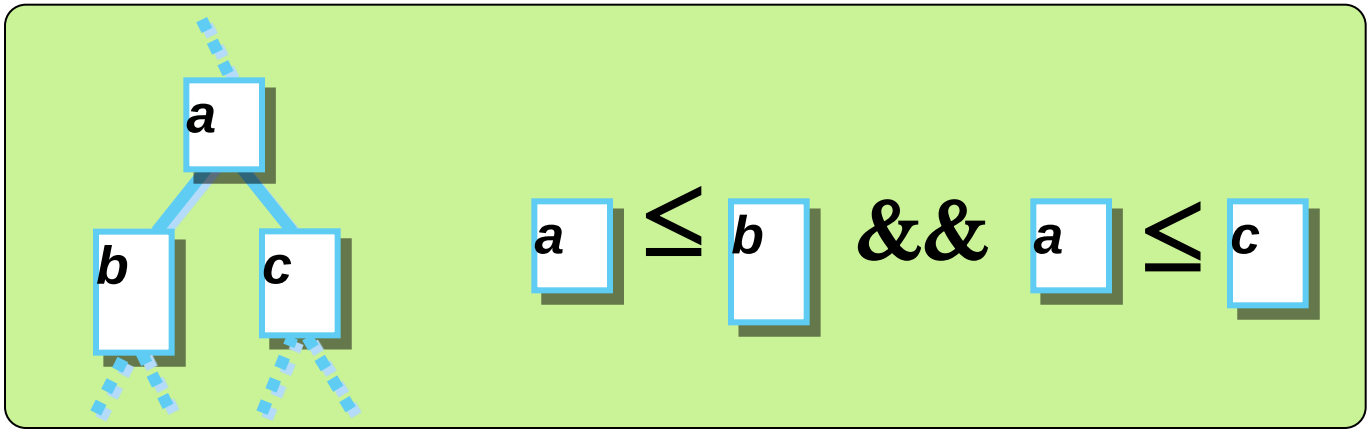
HEAP-SORT

Halda

Halda (binární)

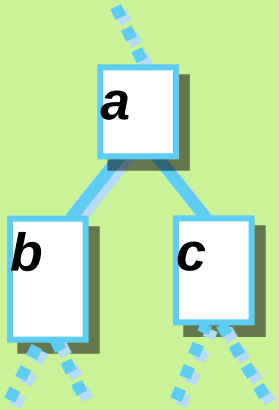


Pravidlo haldy (MINHEAP)



Halda

Terminologie



a ..... predecessor, parent of

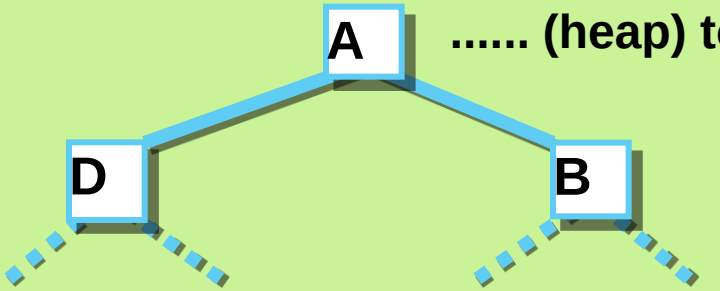
..... předchůdce, rodič

b *c*

b , *c* ..... successor, child of

..... následník, potomek

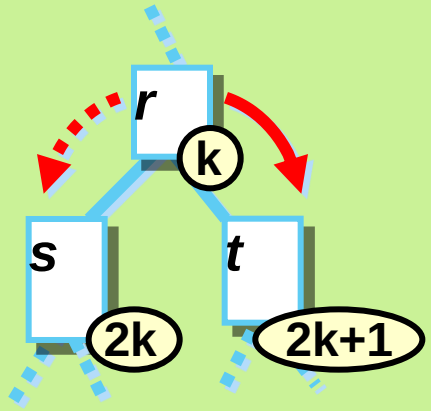
a

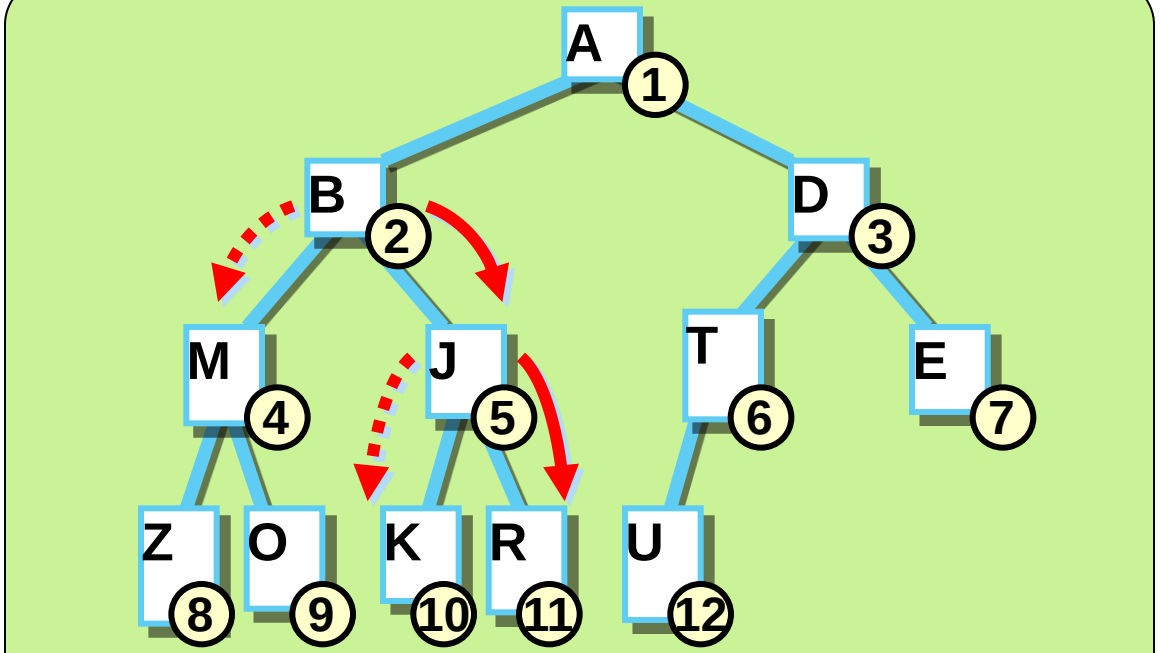


..... (heap) top ..... vrchol (haldy)

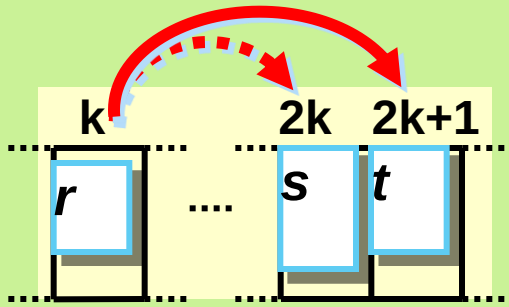
Halda v poli

Halda uložená v poli

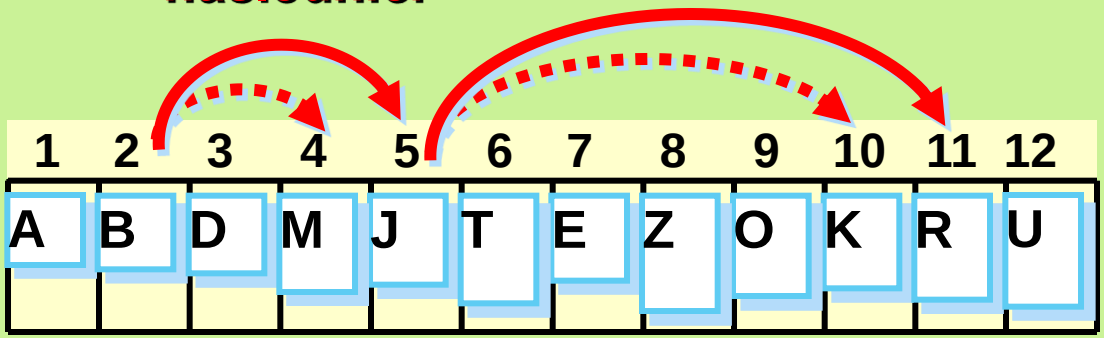




následníci

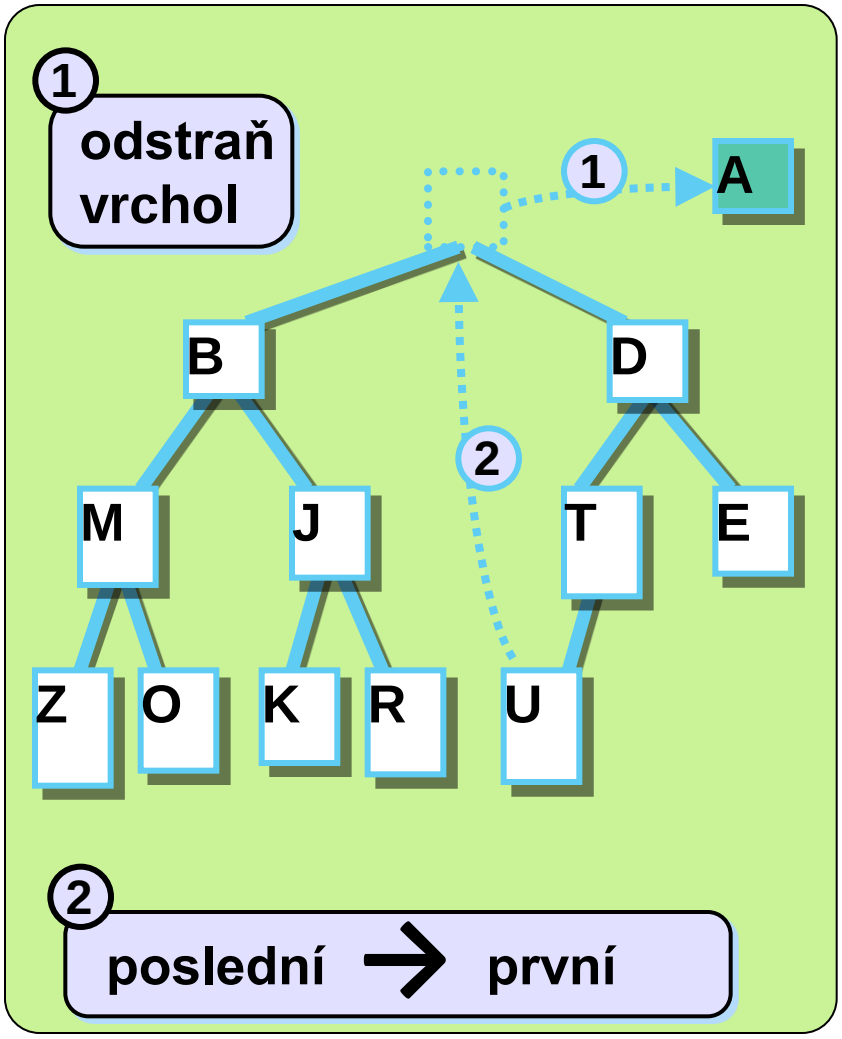


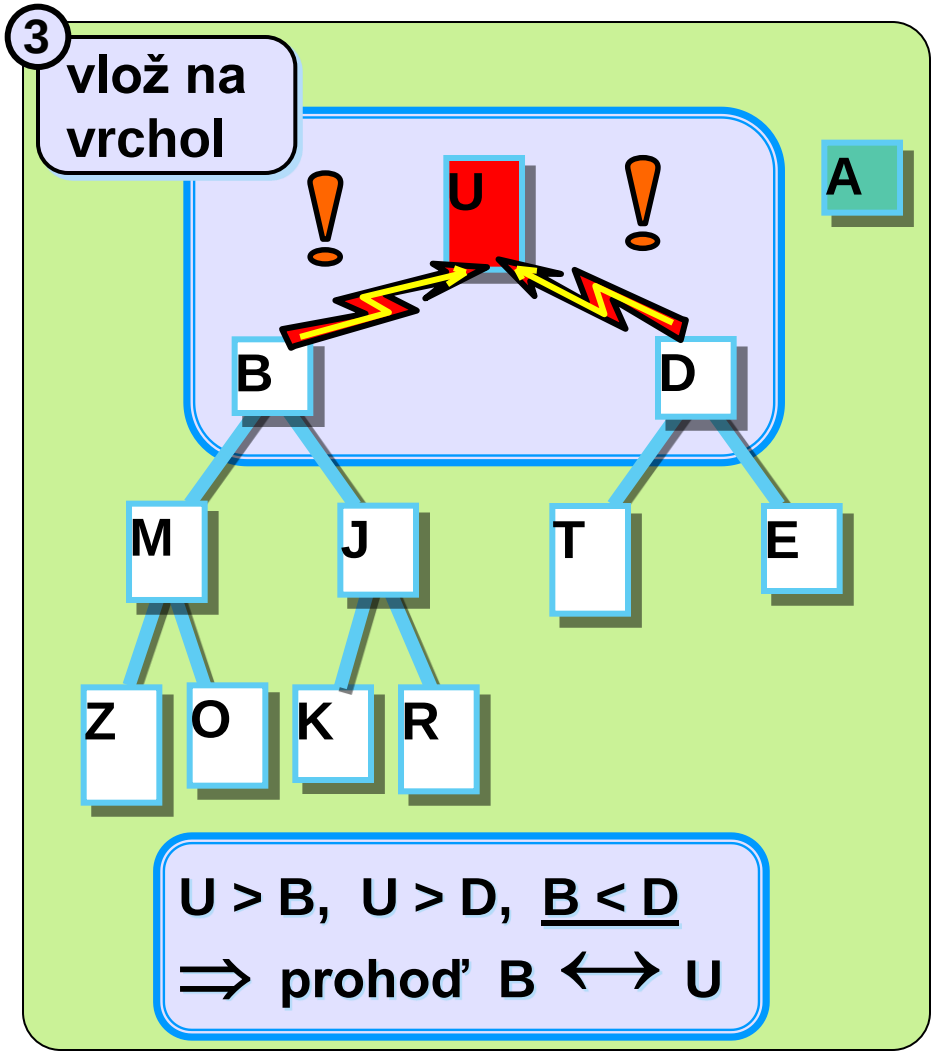
následníci



Oprava haldy

Vrchol odstraněn (1)

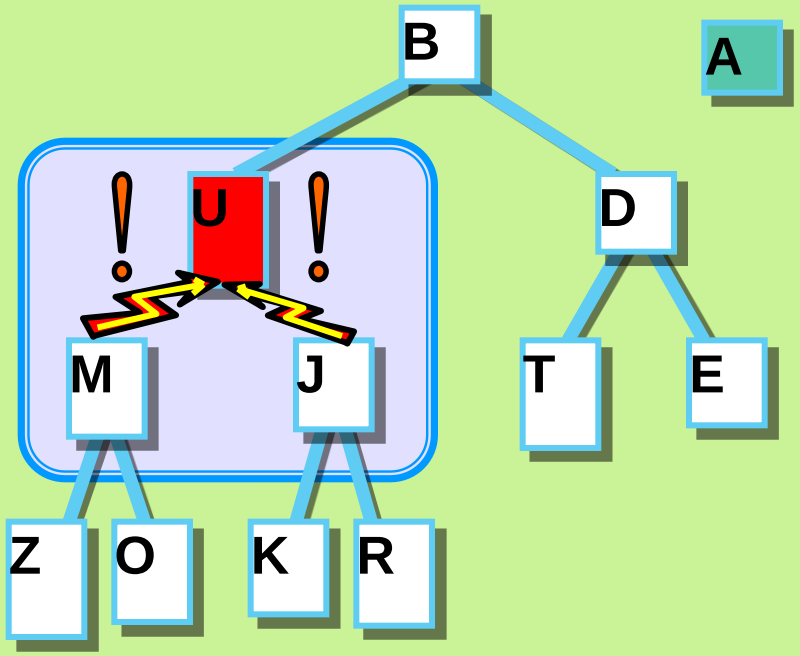




Oprava haldy

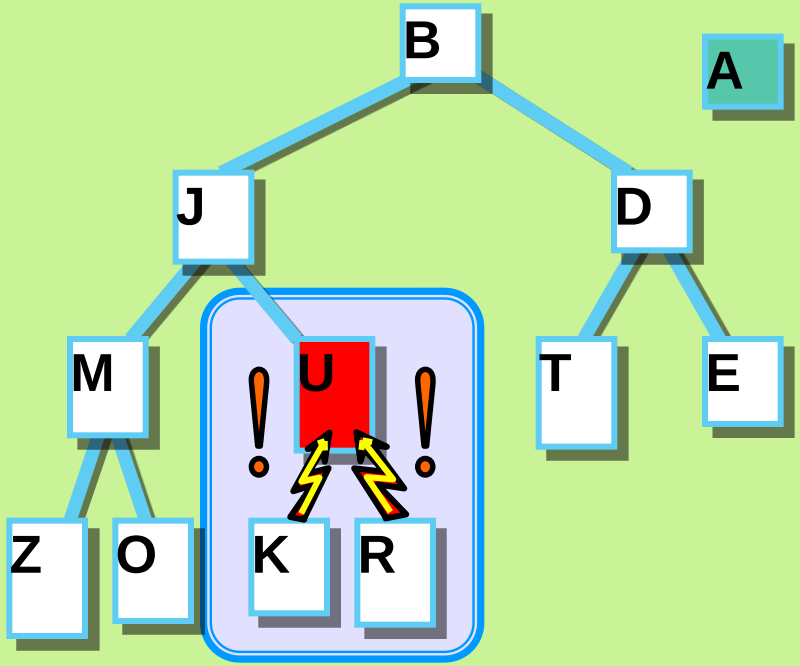
Vrchol odstraněn (2)

3 vlož na vrchol - pokračování



U > M, U > J, J < M

⇒ prohoď J ↔ U



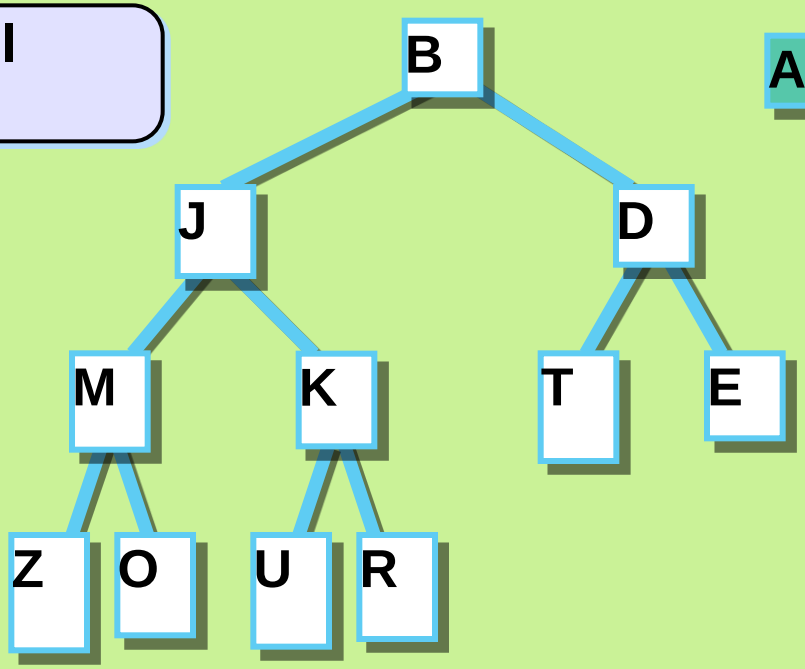
U > K, U > R, K < R

⇒ prohoď K ↔ U

Oprava haldy

Vrchol odstraněn (3)

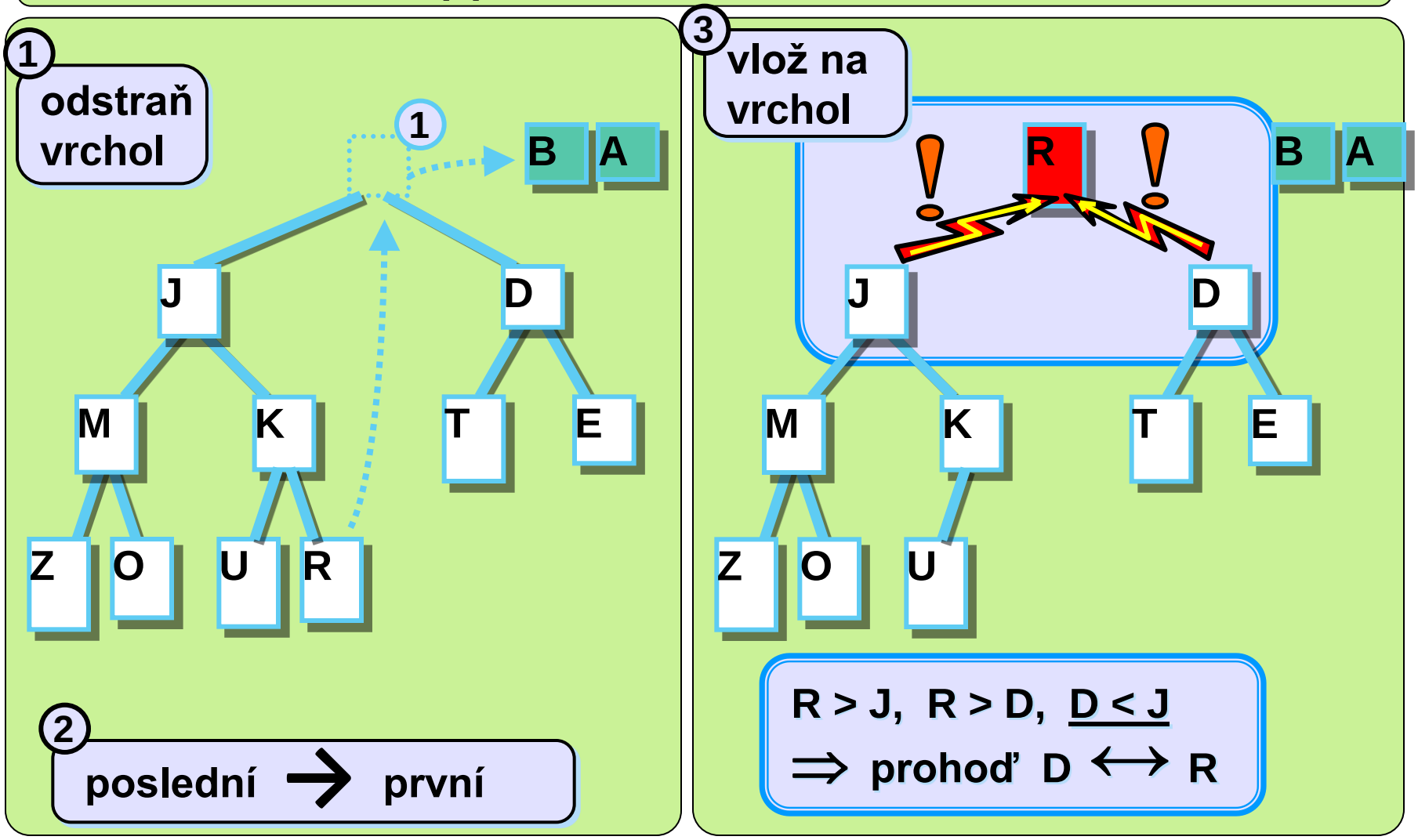
3 vlož na vrchol
- hotovo



Nová halda

Oprava haldy

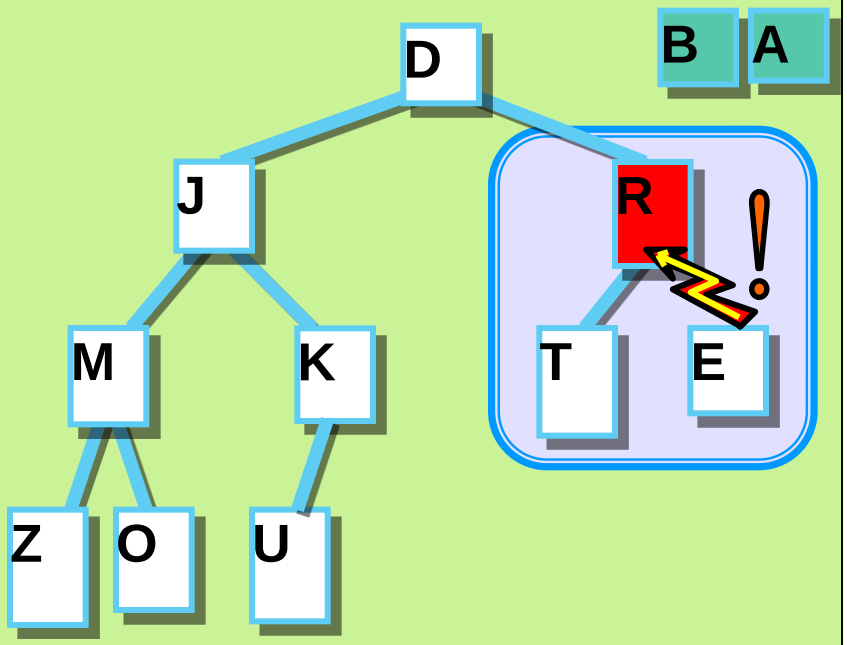
Vrchol odstraněn II (1)



Oprava haldy

Vrchol odstraněn II (2)

3 vlož na vrchol - pokračování

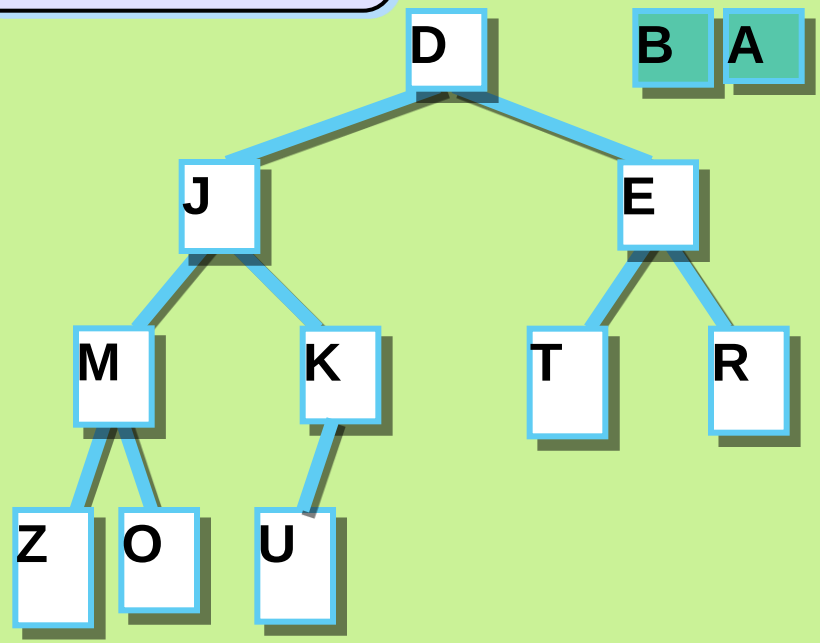


$R < T,\ R > E$

$\Rightarrow$ prohoď $E \leftrightarrow R$

Vrchol odstraněn II (3)

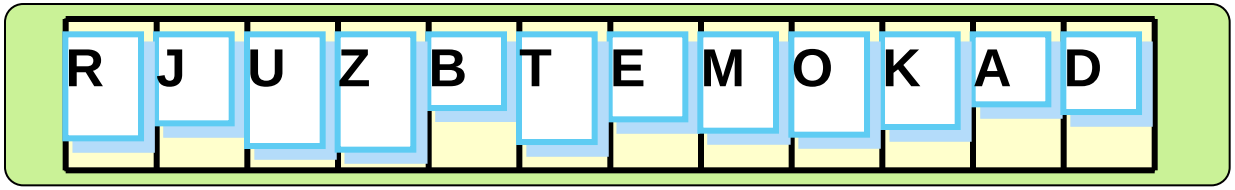
3 vlož na vrchol - hotovo



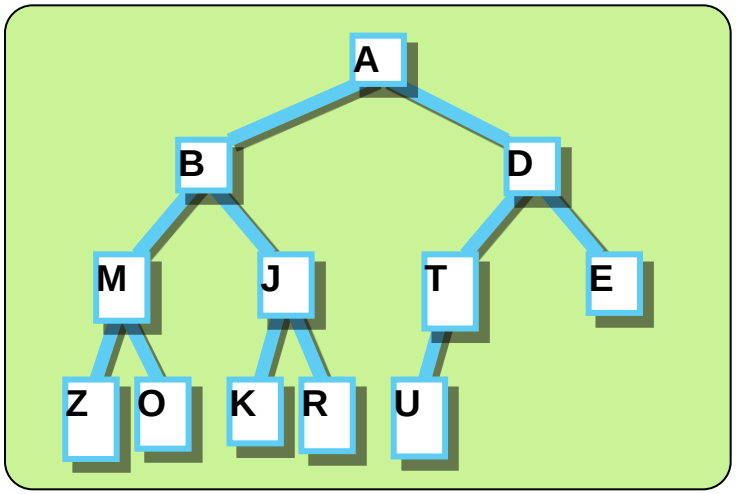
Nová halda

Řazení pomocí haldy - HEAP-SORT

I Neseřazeno

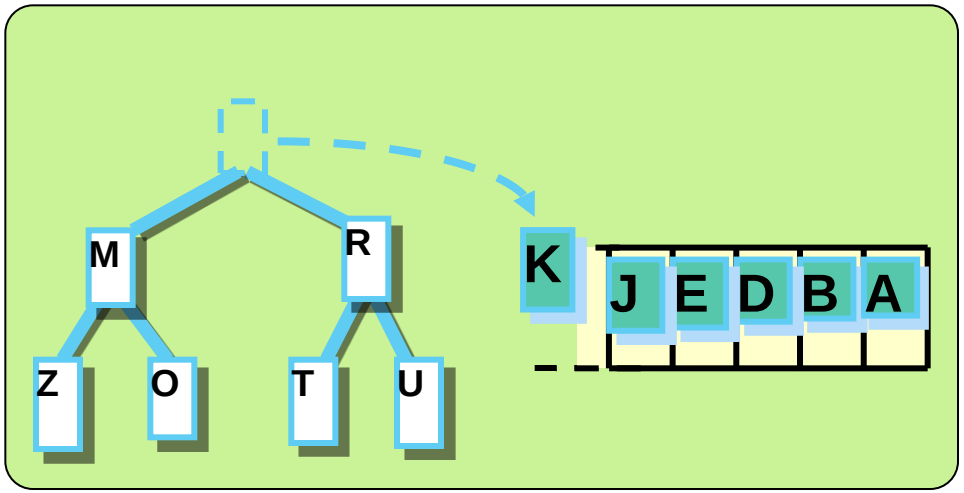


II Vytvoř haldu

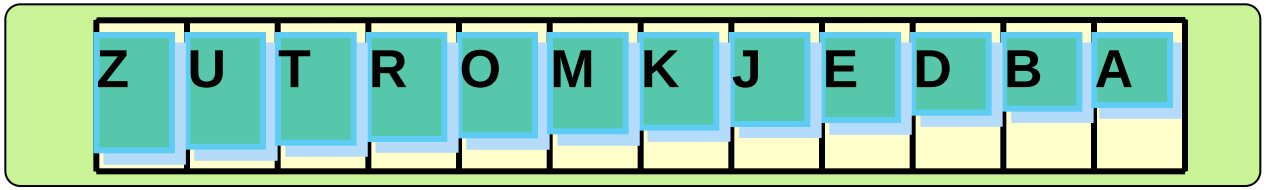


III

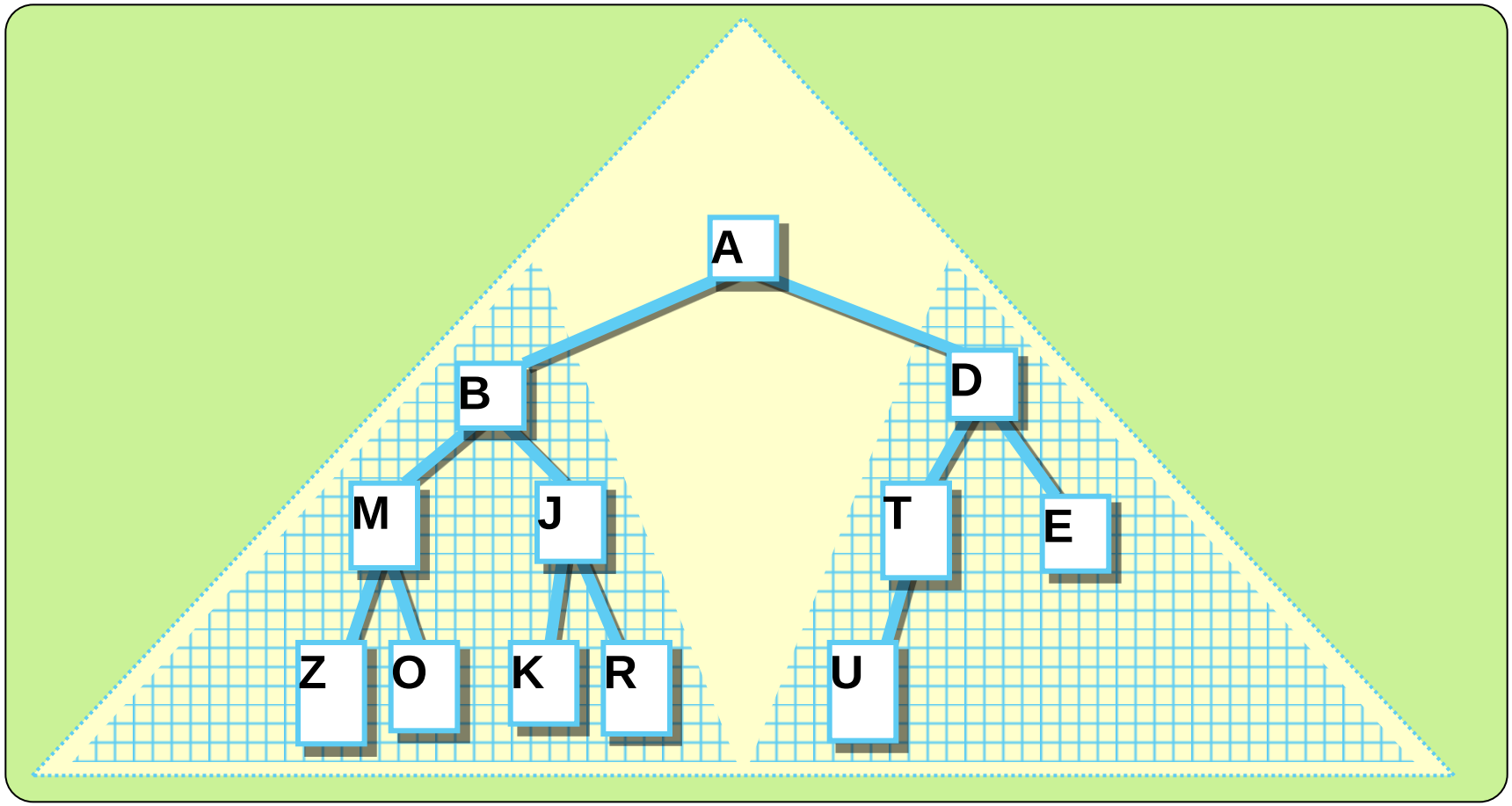
```
for (i = 0; i < n; i++)
    a[i] = “odstraň vrchol”;
```



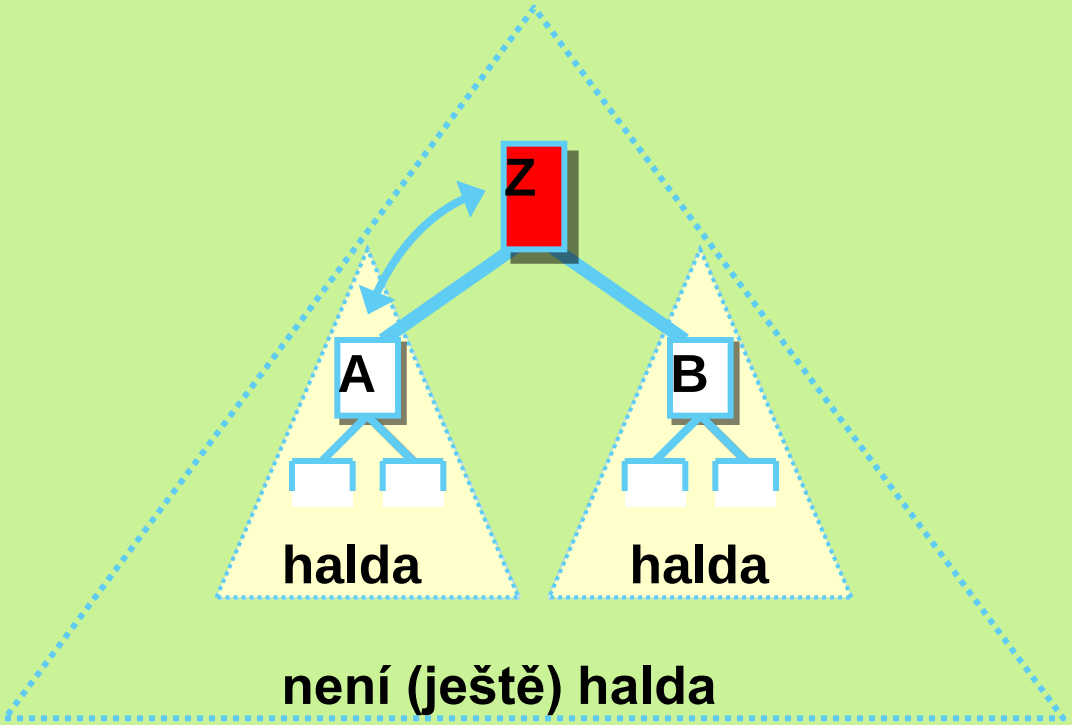
IV Seřazeno



Rekurzivní vlastnost „býti haldou”



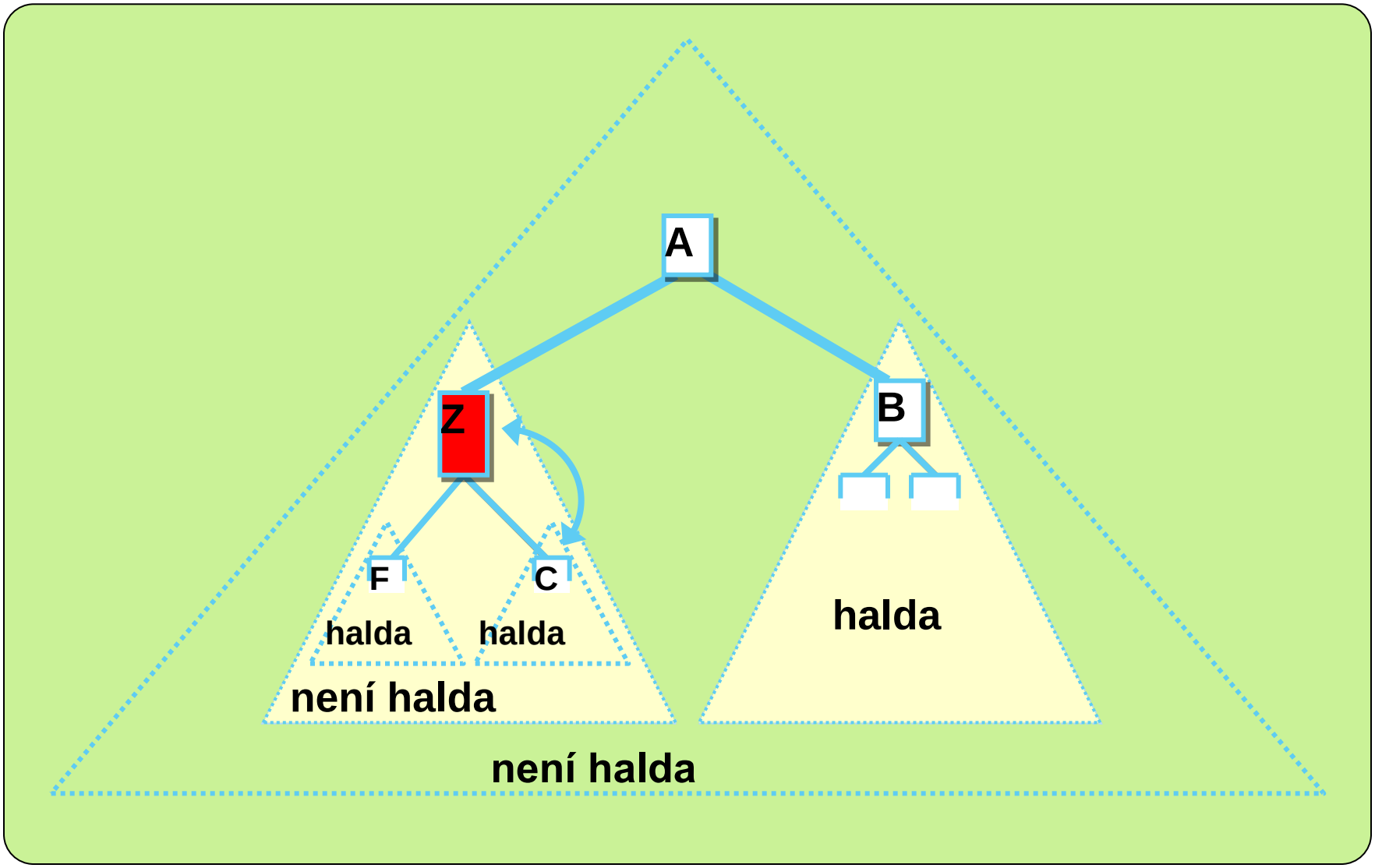
Vytvoř jednu haldu ze dvou menších



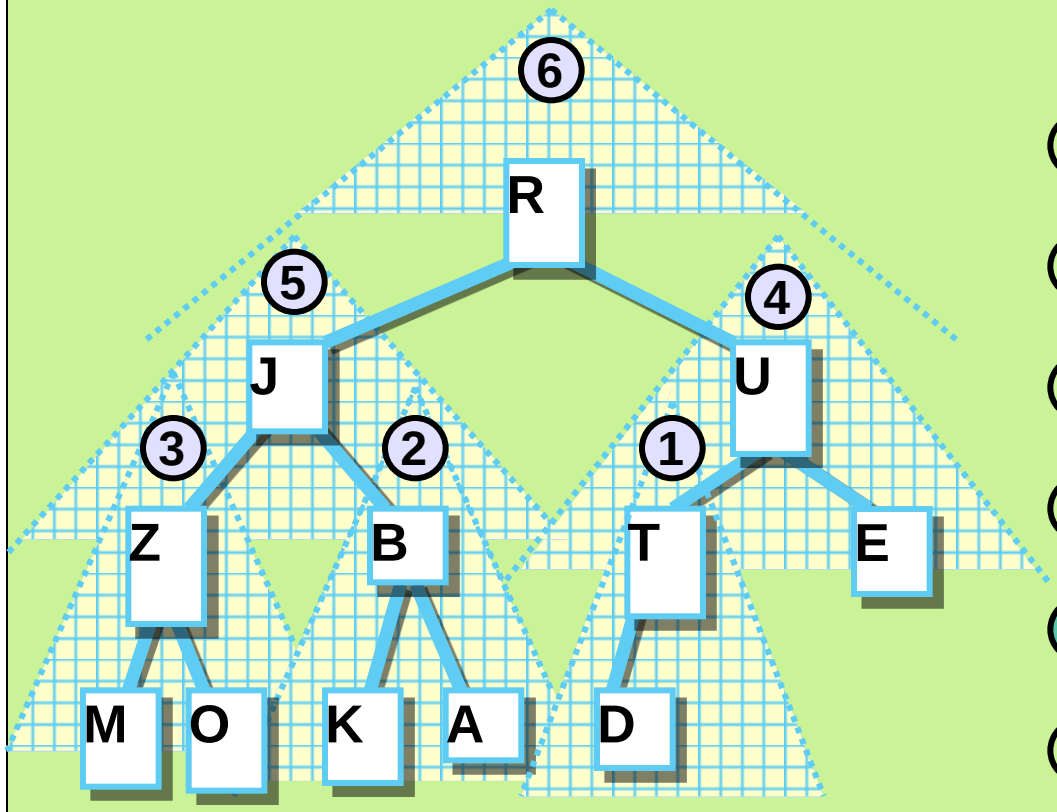
$Z > A$ nebo $Z > B$

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

Vytvoř jednu haldu ze dvou menších



Vytvoř haldu



① Vytvoř haldu zde…

② … vytvoř haldu zde…

③ … vytvoř haldu zde…

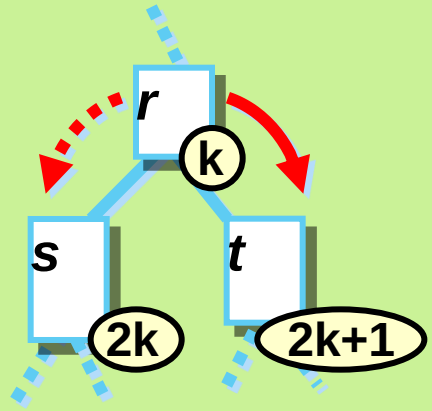
④ … vytvoř haldu zde…

⑤ … vytvoř haldu zde…

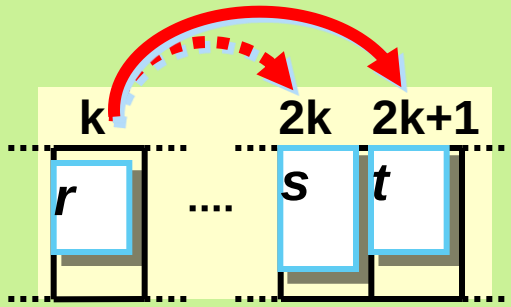
⑥ … vytvoř haldu zde, celá halda je hotova.

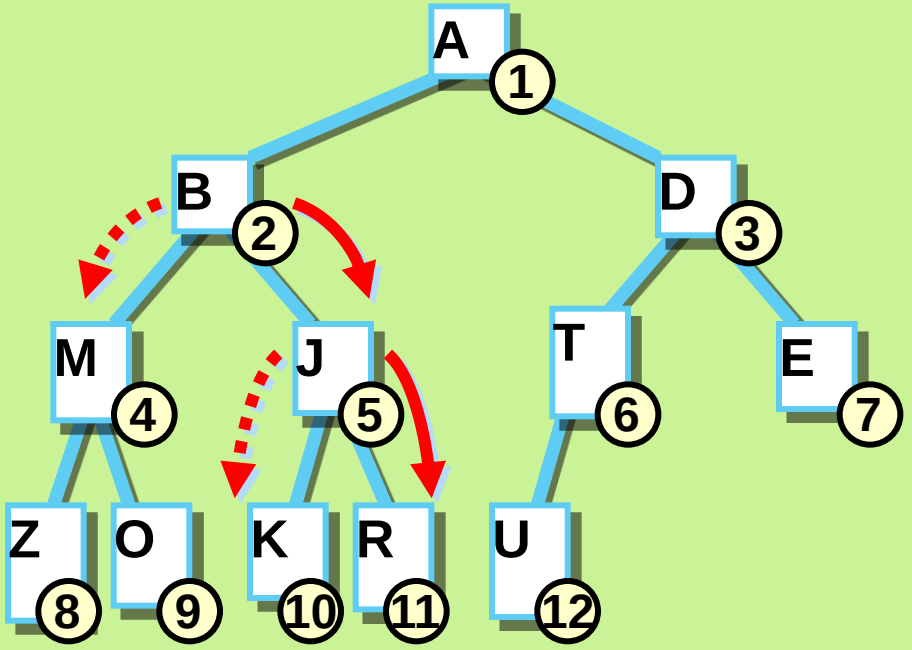
Halda v poli

Halda uložená v poli

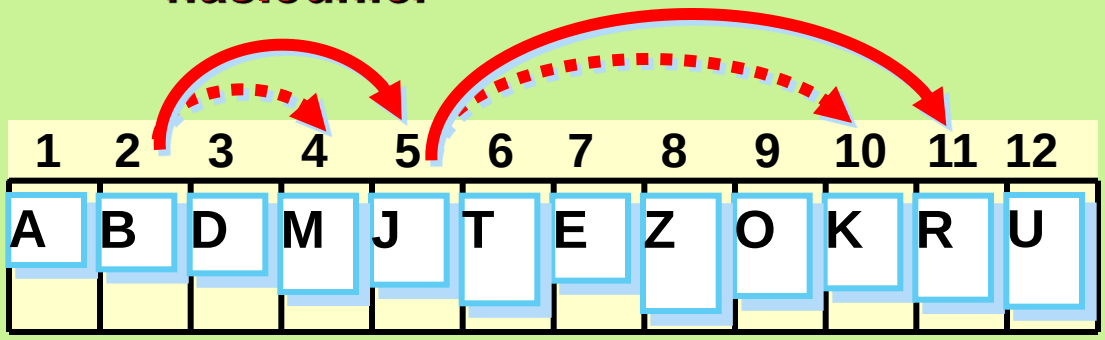


následníci





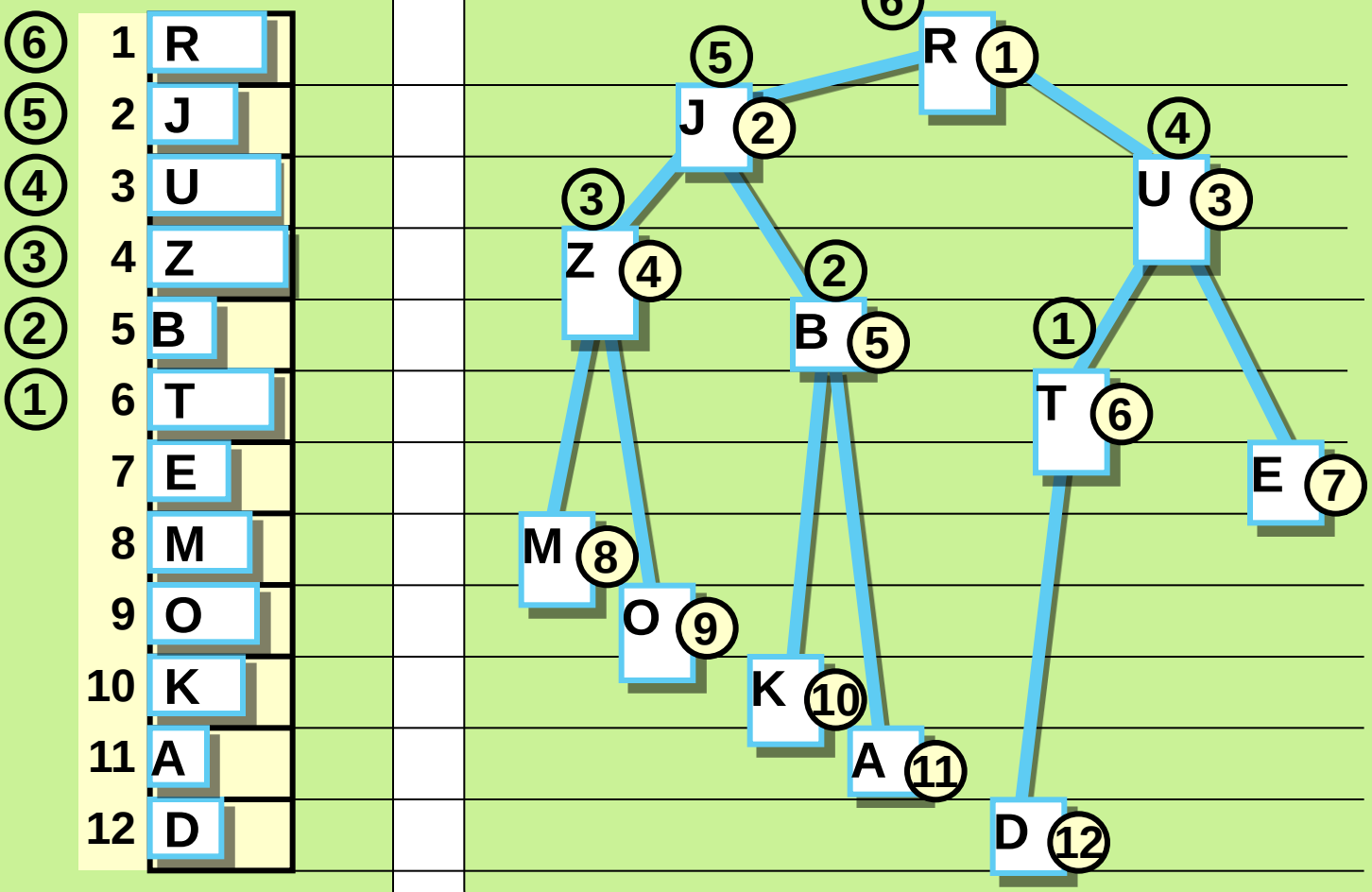
následníci



Tvorba haldy v poli

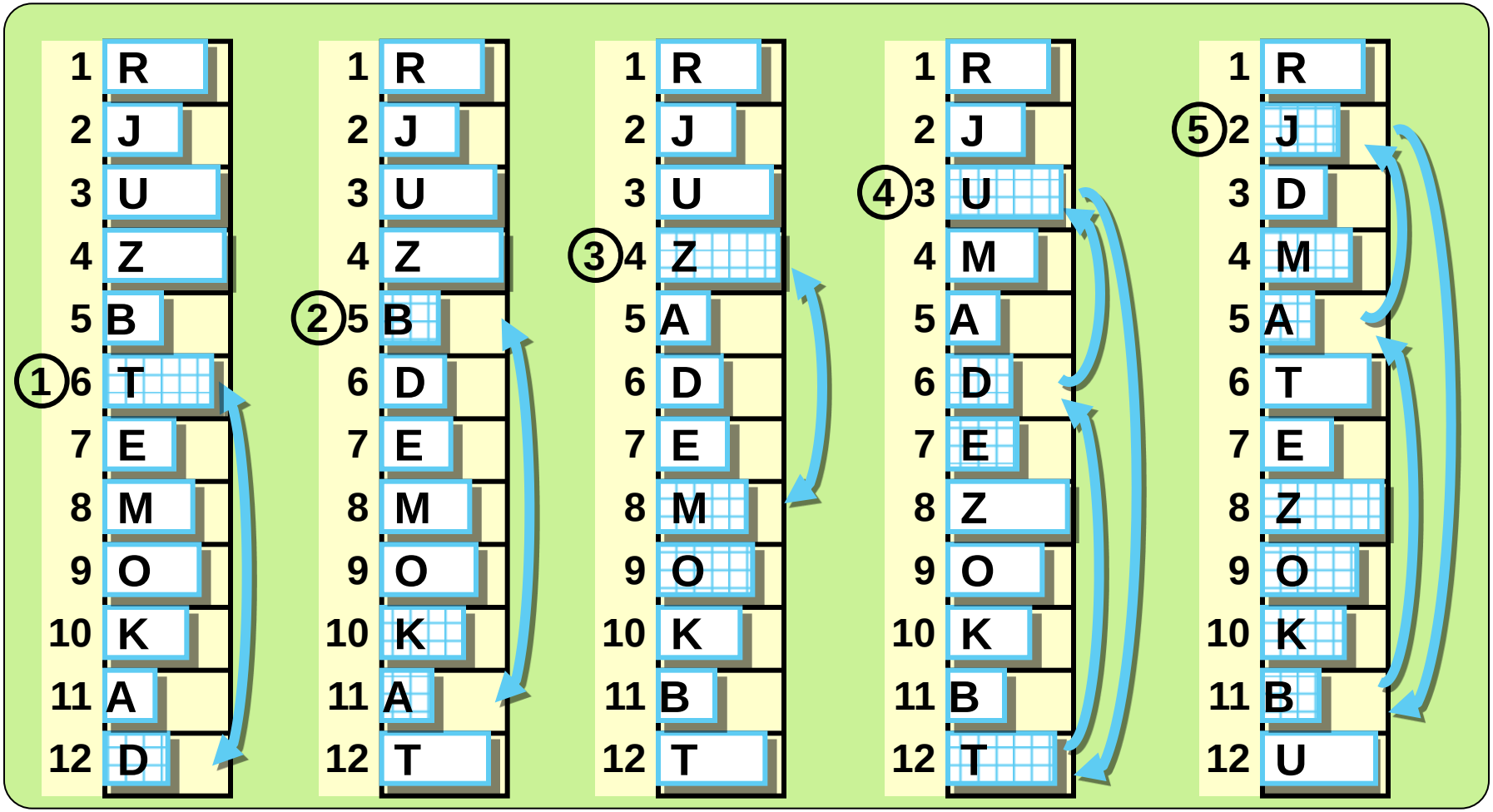
Neseřazeno

Není halda



Tvorba haldy v poli

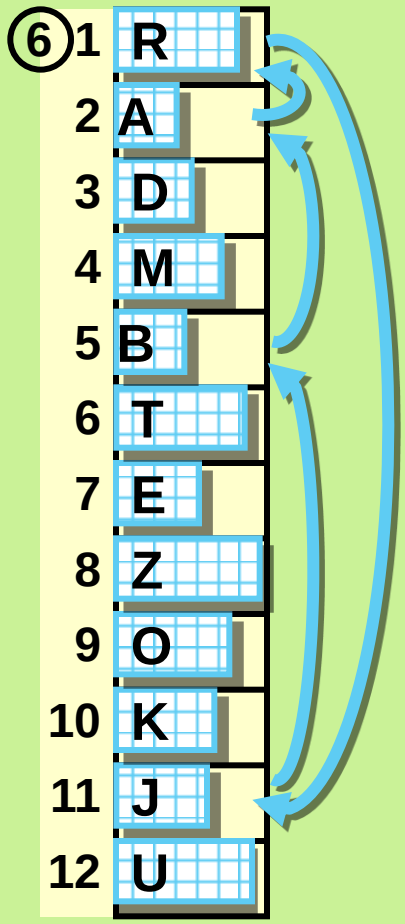
tvorba haldy:

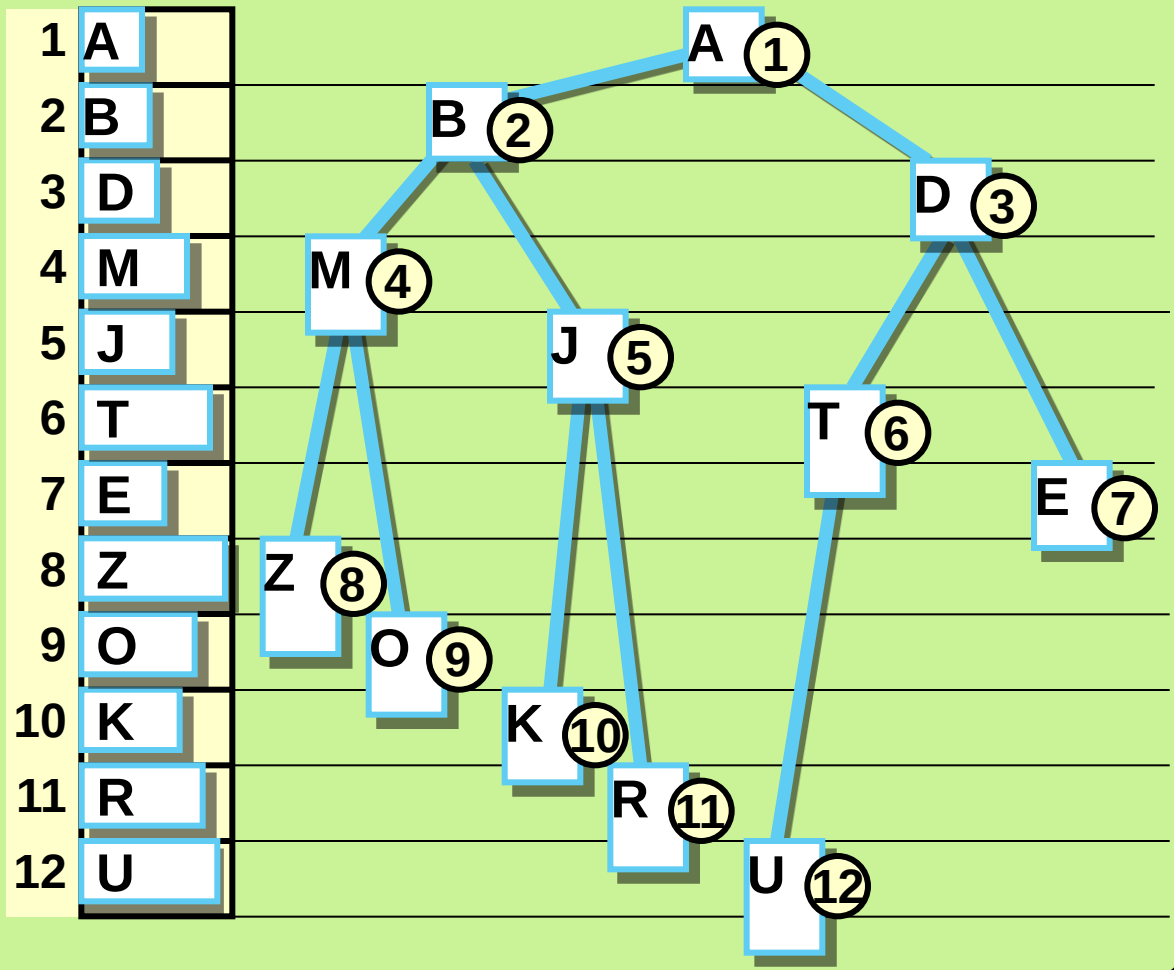


Tvorba haldy v poli

Tvorba haldy

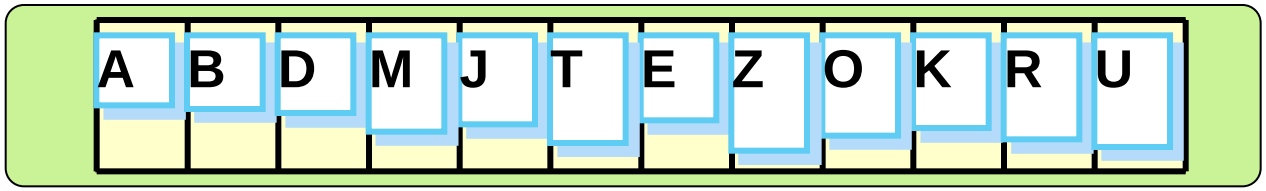
Halda



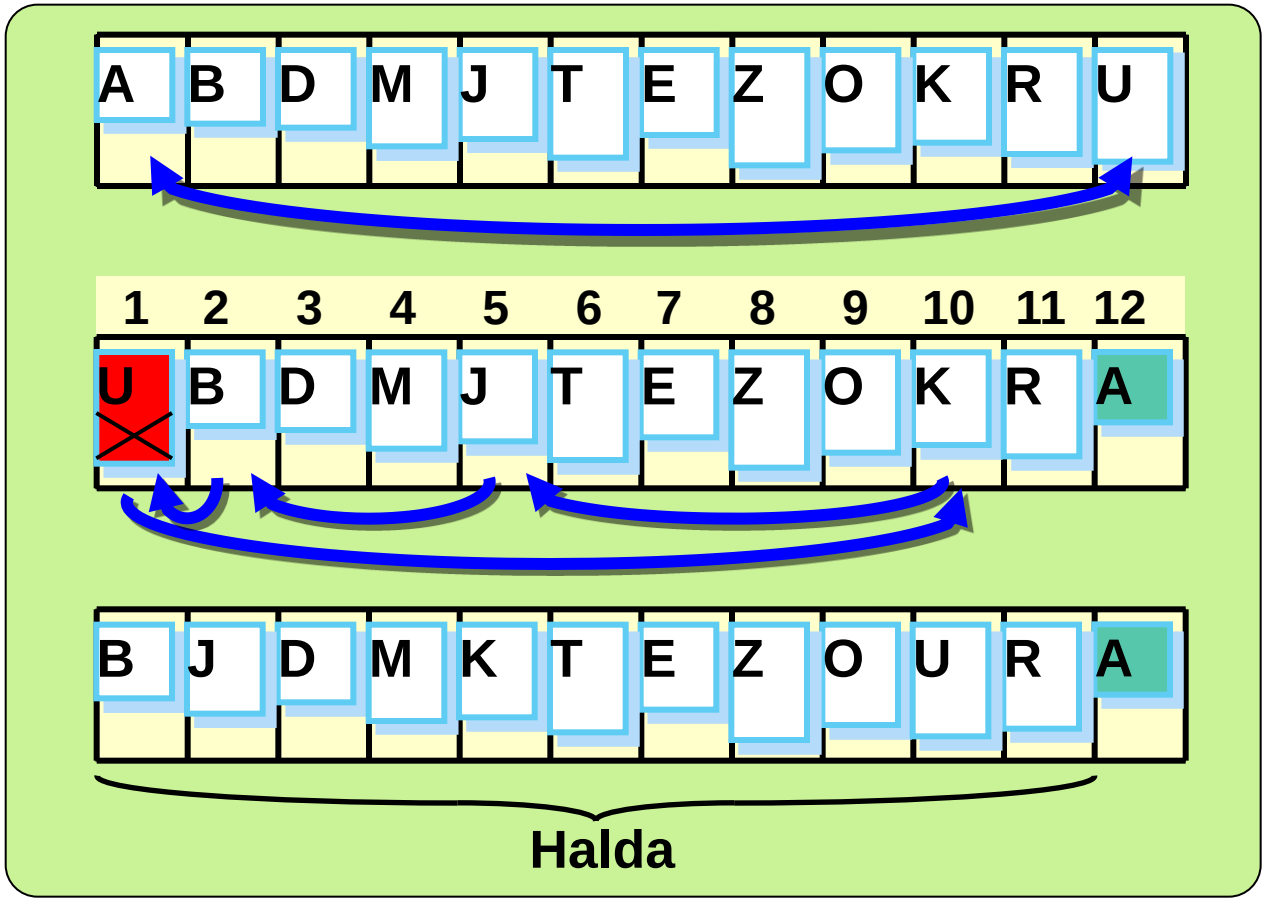


HEAP-SORT

Halda

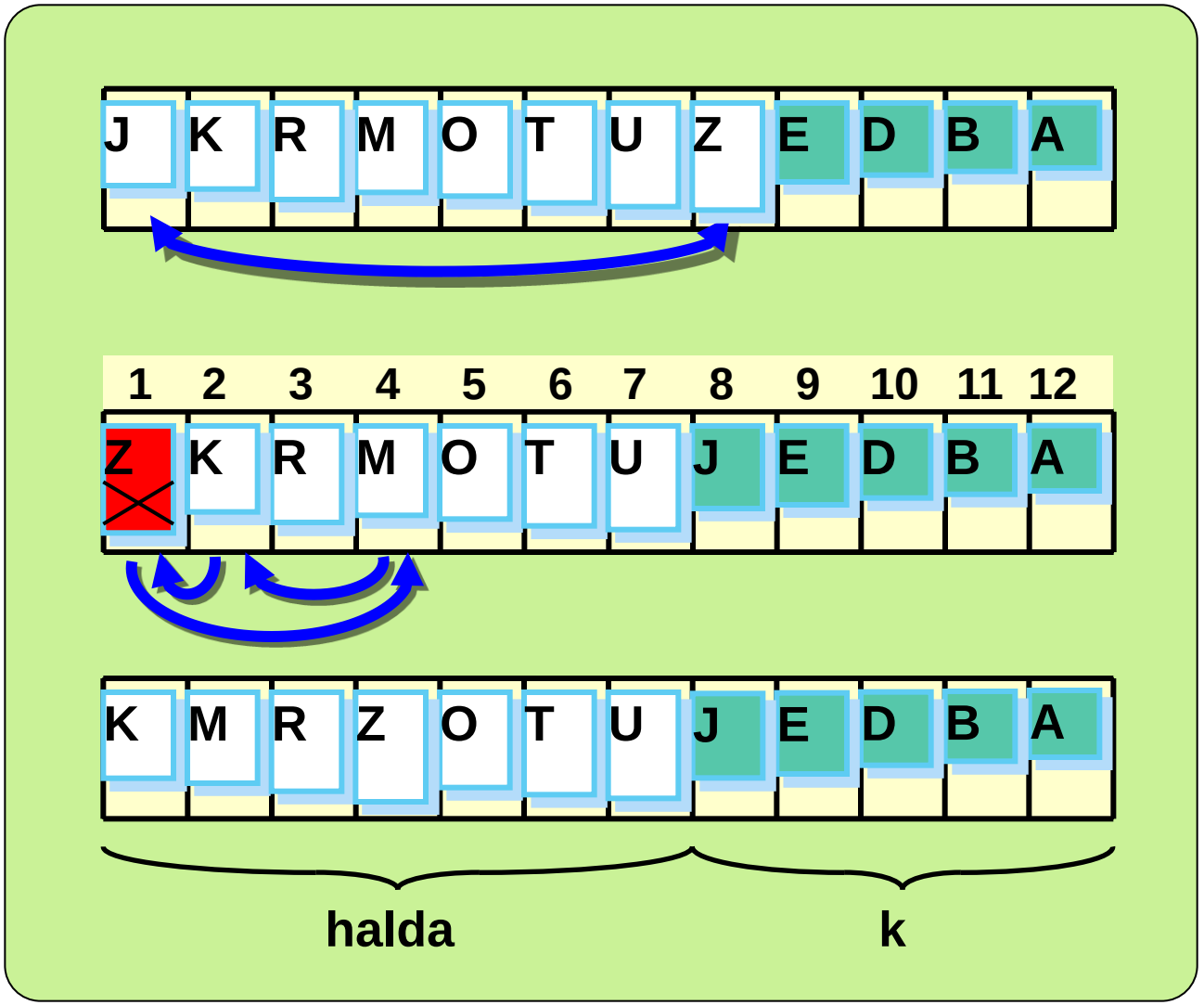


Krok 1



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

$\log_2(n/2) + \log_2(n/2+1) + \ldots + \log_2(n) \leq (n/2)(\log_2(n)) = O(n \cdot \log_2(n))$

seřaď haldy ... n-1 repairTop operací, nejhorší případ:

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

ale i nejlepší případ = $\Theta(n \cdot \log_2(n))$

celkem ... vytvoř haldu + seřaď haldu = $\Theta(n \cdot \log_2(n))$

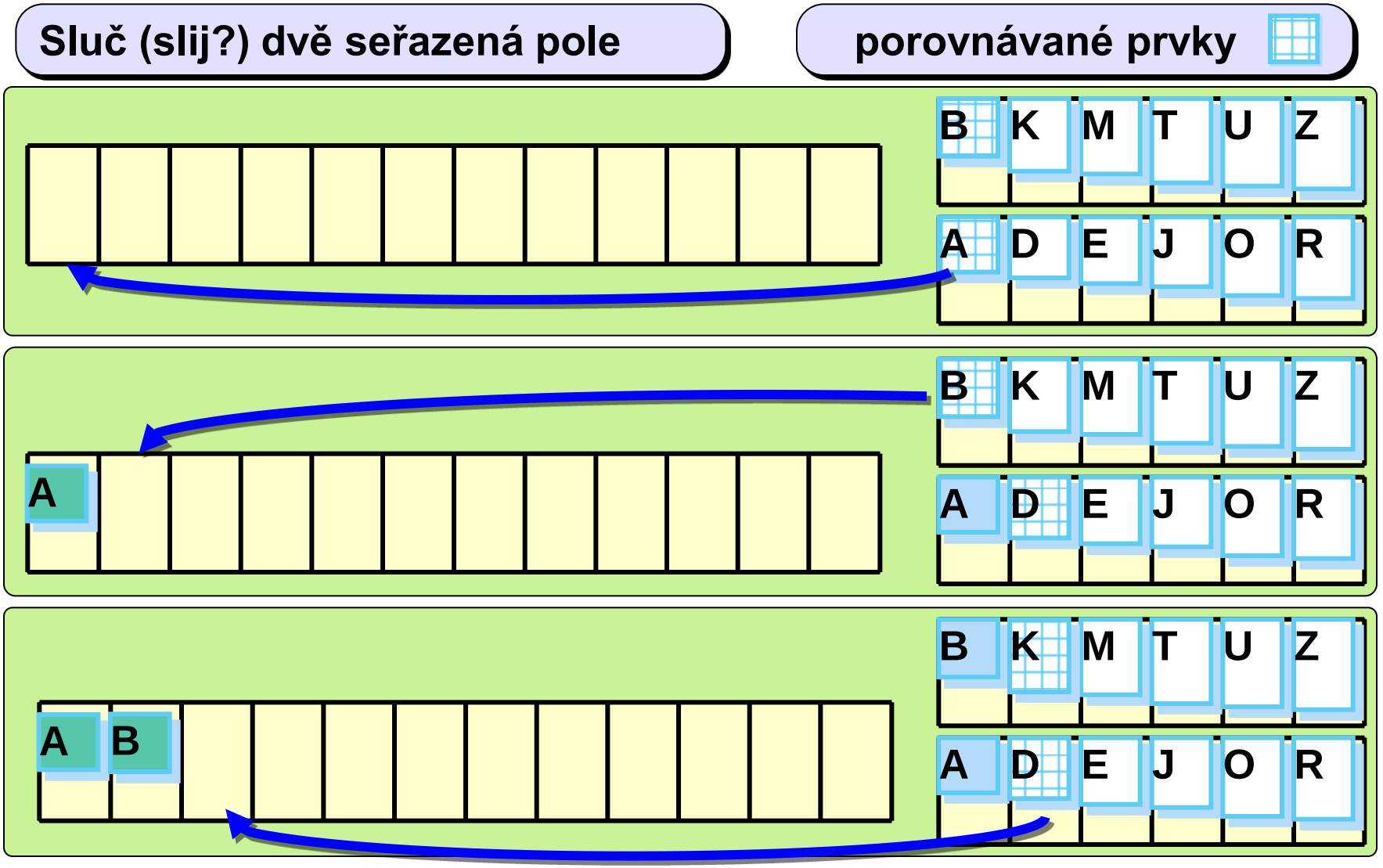
Asymptotická složitost HEAP-SORT je $\Theta(n \cdot \log_2(n))$

HEAP-SORT není stabilní

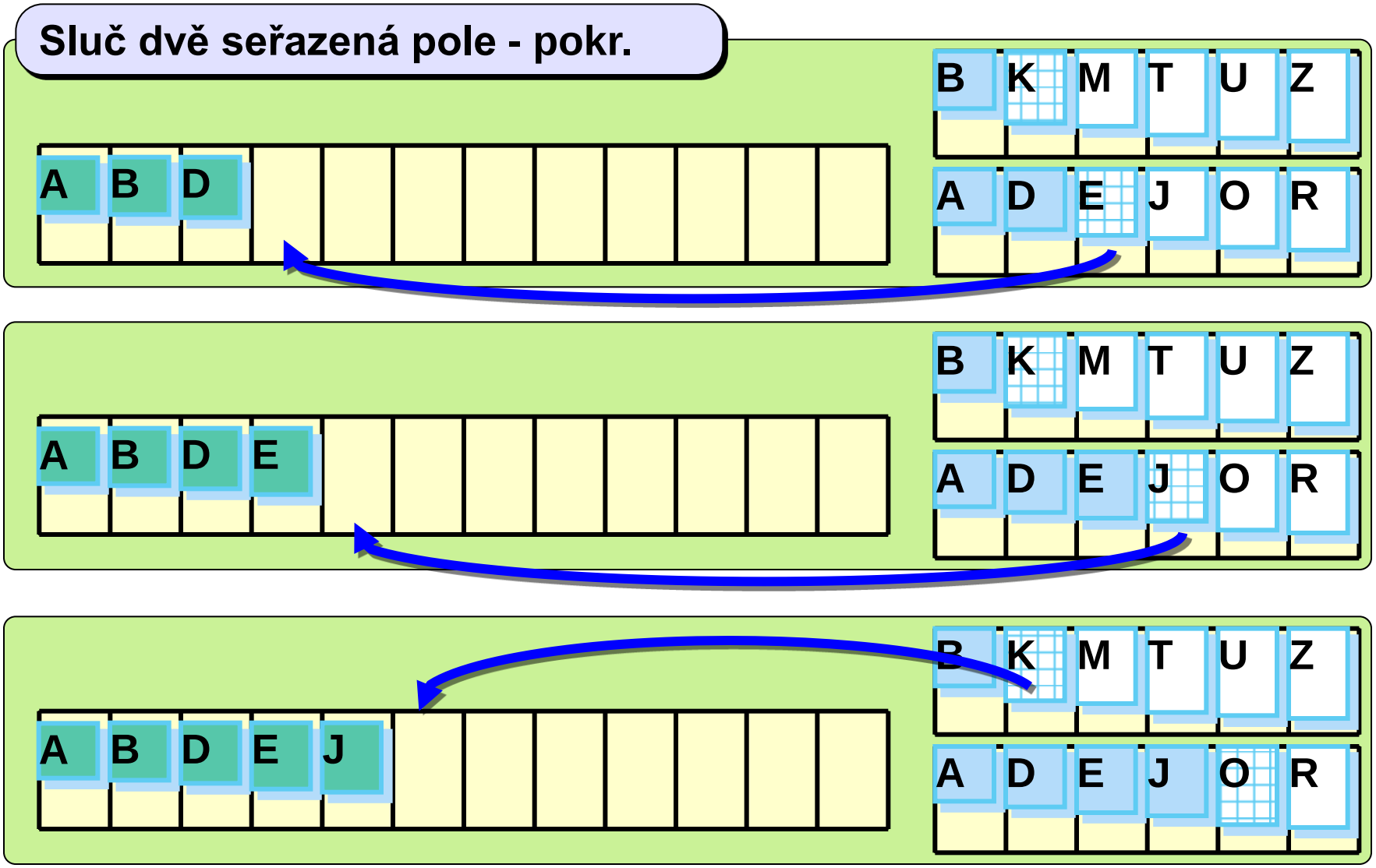
Řazení slučováním

MERGE-SORT

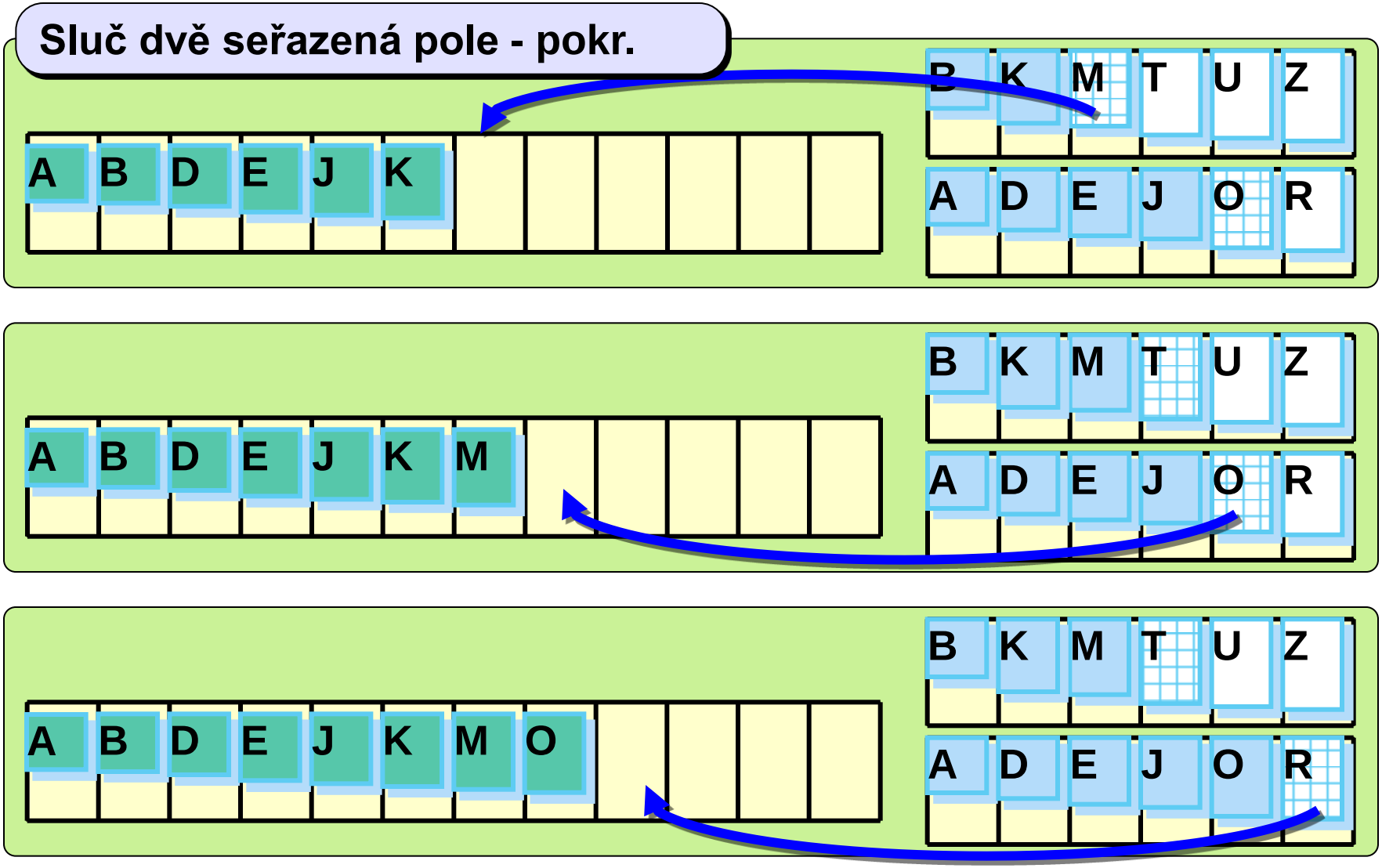
Slučování posloupností



Slučování posloupností



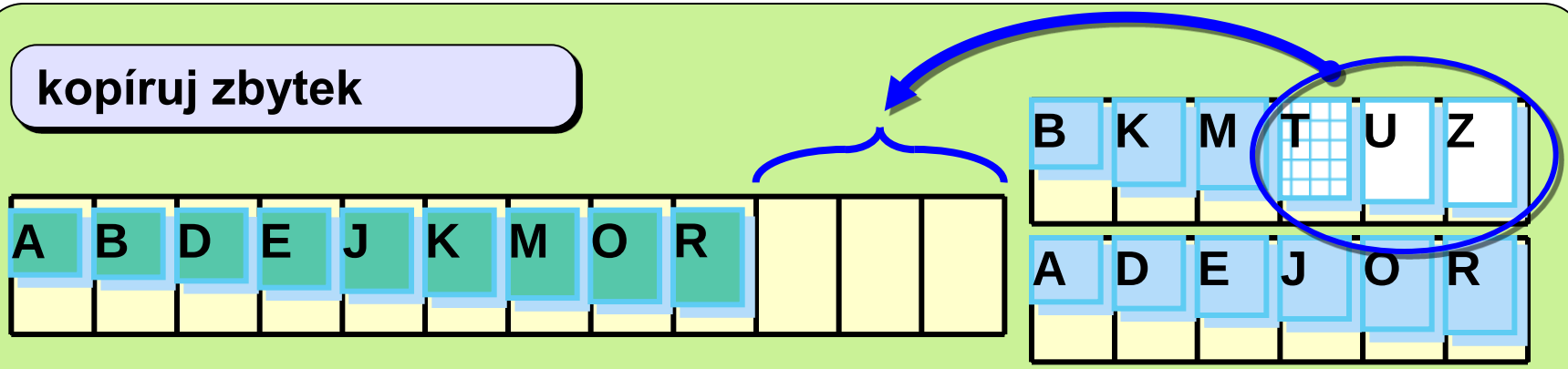
Slučování posloupností



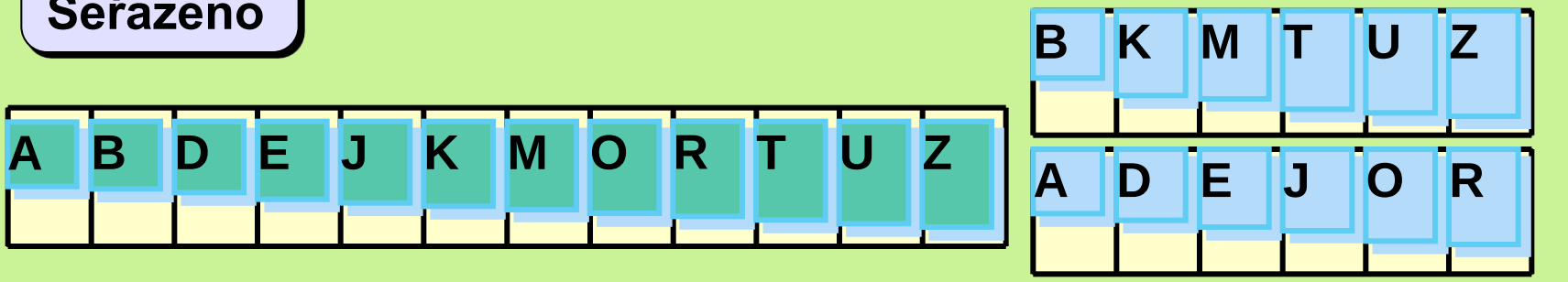
Slučování posloupností

Sluč dvě seřazená pole - pokr.

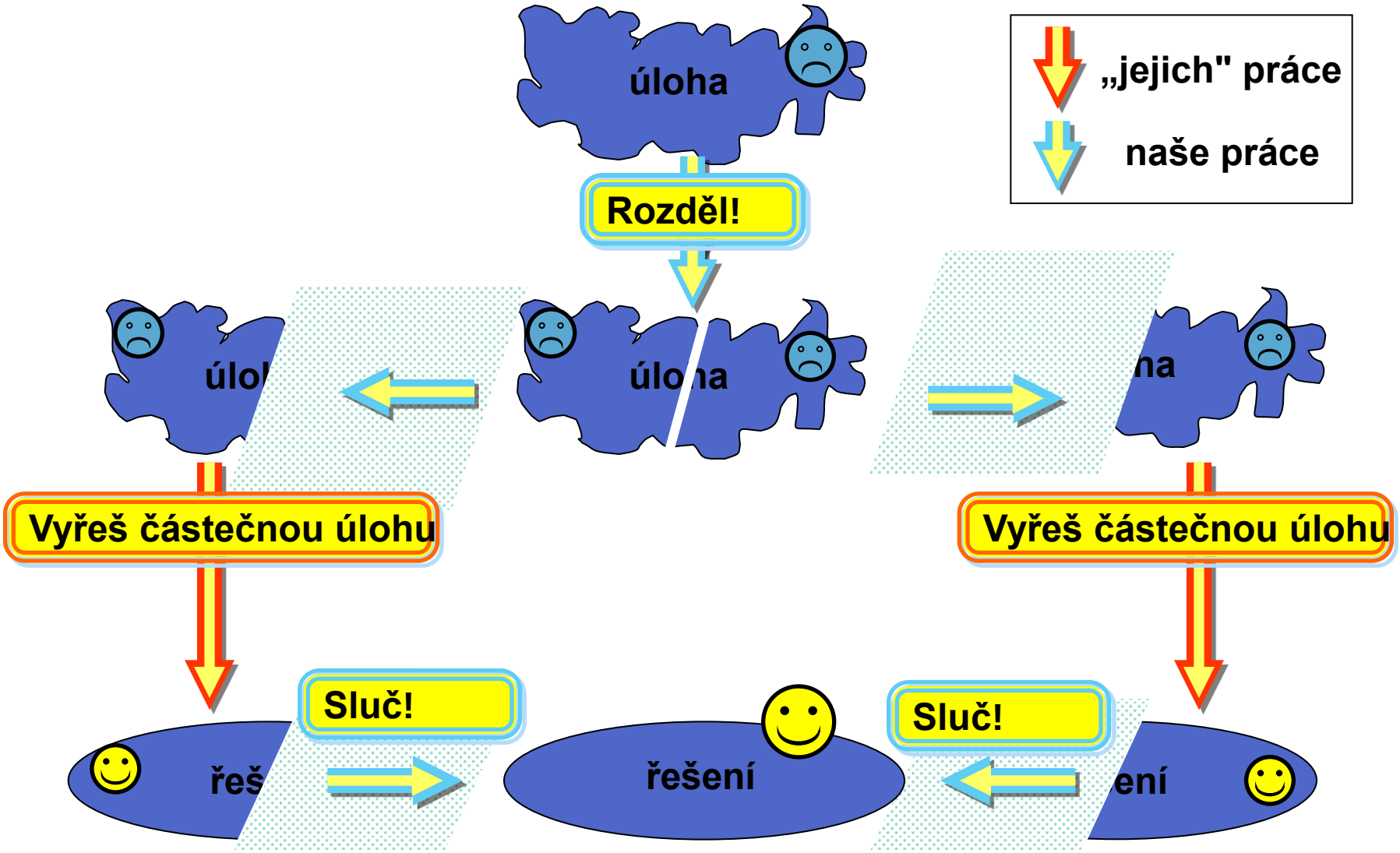
kopíruj zbytek



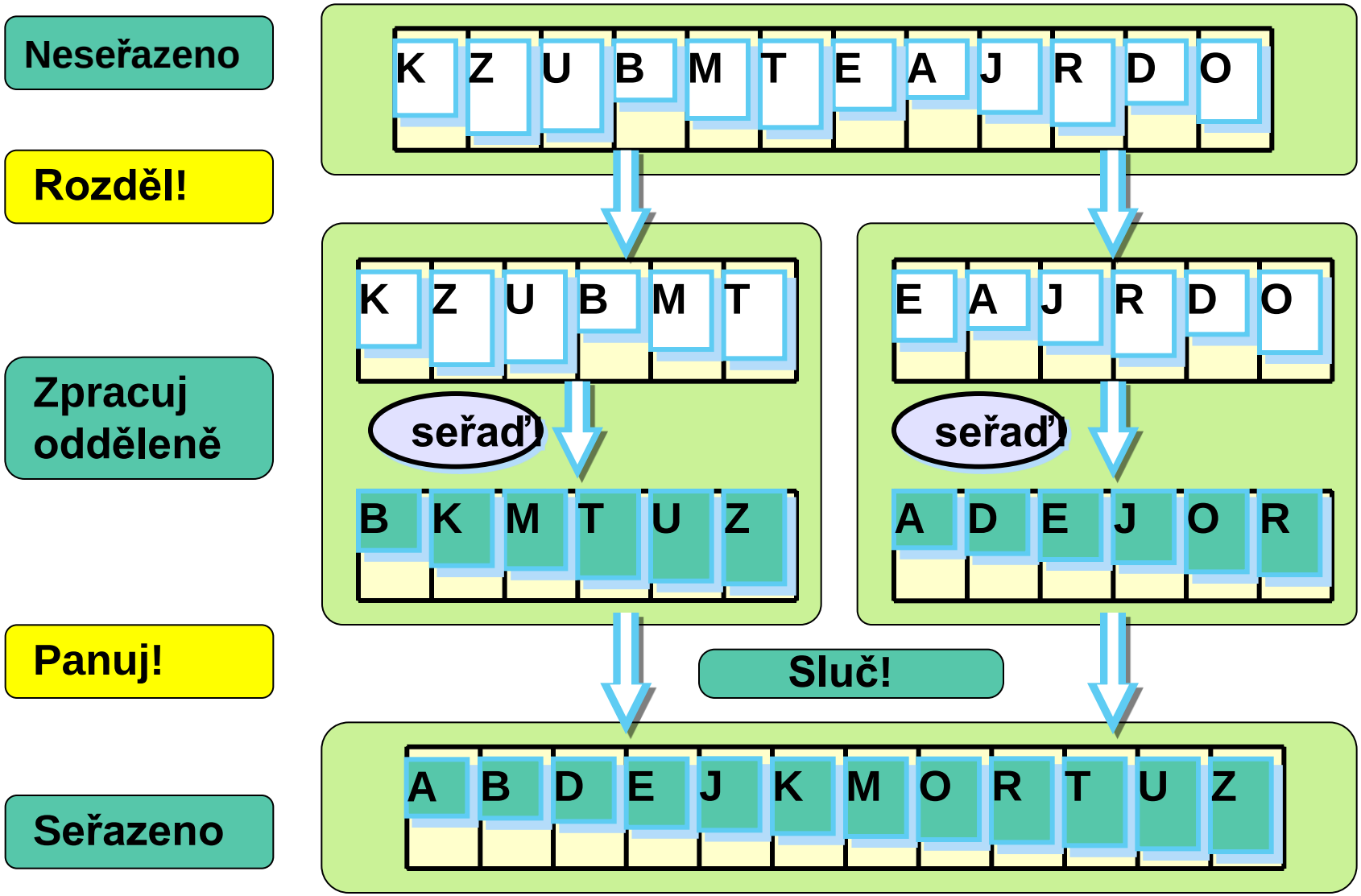
Seřazeno



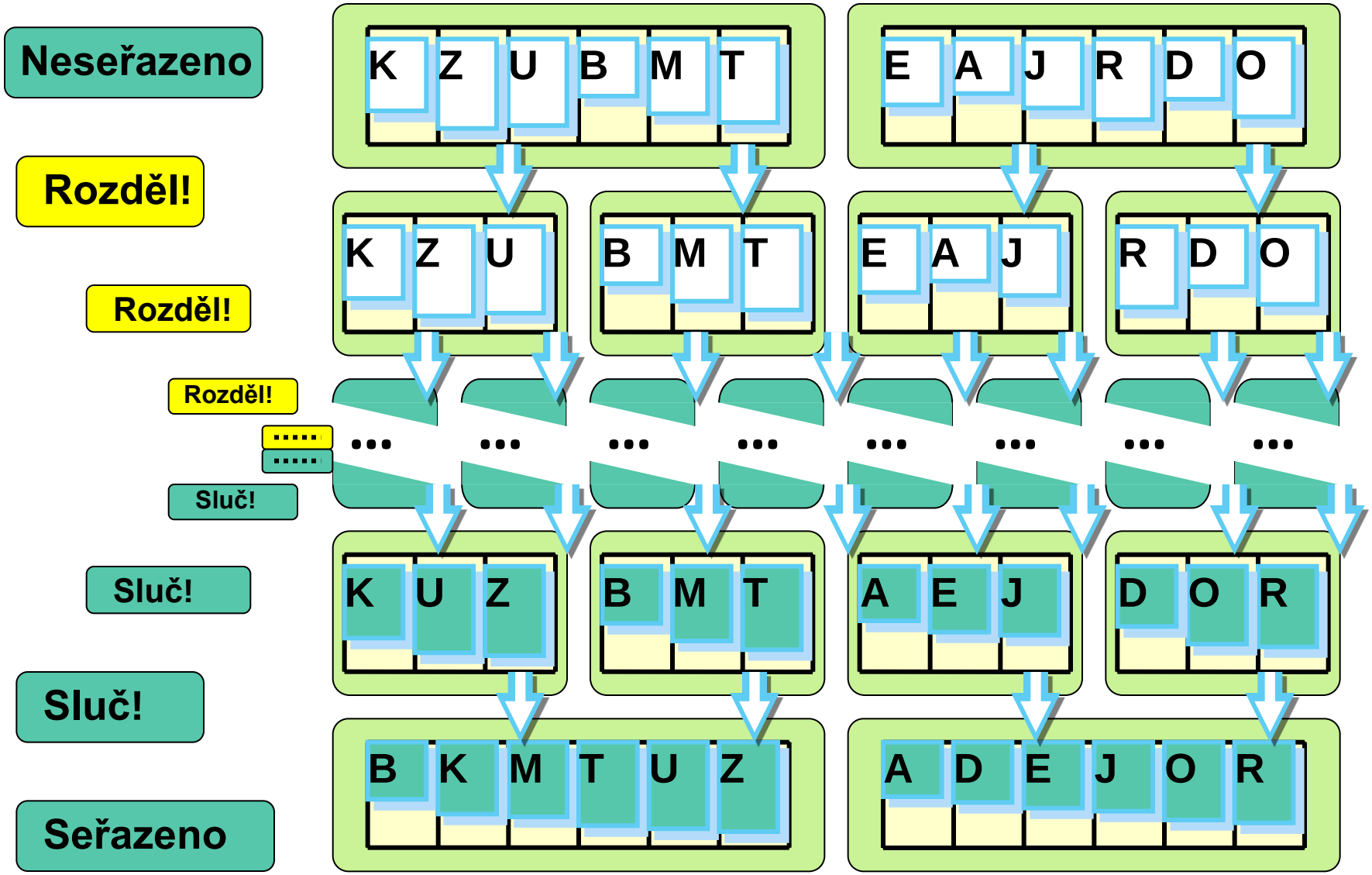
Rozděl a Panuj! Divide & Conquer! Divide et Impera!



MERGE-SORT


Neseřazeno
Rozděl!
Zpracuj odděleně
Panuj!
Seřazeno
K Z U B M T E A J R D O
K Z U B M T
E A J R D O
seřaď
seřaď
B K M T U Z
A D E J O R
Sluč!
A B D E J K M O R T U Z

MERGE-SORT


Neseřazeno
Rozděl!
Rozděl!
Rozděl!
Sluč!
Sluč!
Sluč!
Seřazeno
K Z U B M T
E A J R D O
K Z U
B M T
E A J
R D O
K U Z
B M T
A E J
D O R
B K M T U Z
A D E J O R

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

Rozděl! ..... $\log_2(n)$ krát $\Rightarrow$

$\Rightarrow$ Sluč! ..... $\log_2(n)$ krát

Rozděl! ..... $\Theta(1)$ 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]; …”

Zařaď 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.

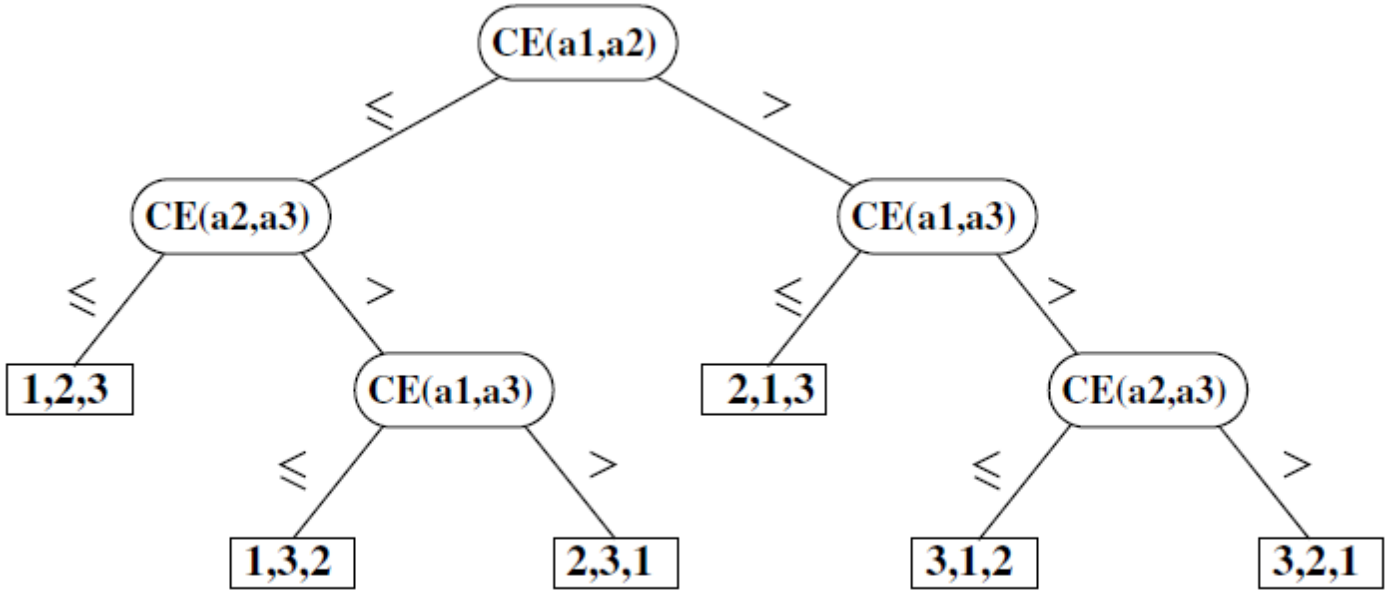
Věta:

- 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 = <a1, . . , an>. 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:

$\log(n!) = \log(n*(n-1)*\ldots*1) = \log(n)+\log(n-1)+\ldots+\log(1) \leq n \log n \Rightarrow \log(n!) = \mathrm{O}(n \log n)$

$\log(n!) \geq \log(n/2)+\ldots+\log(n) = n/2 * \log n/2 \Rightarrow \log(n!) = \Omega(n \log n)$

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

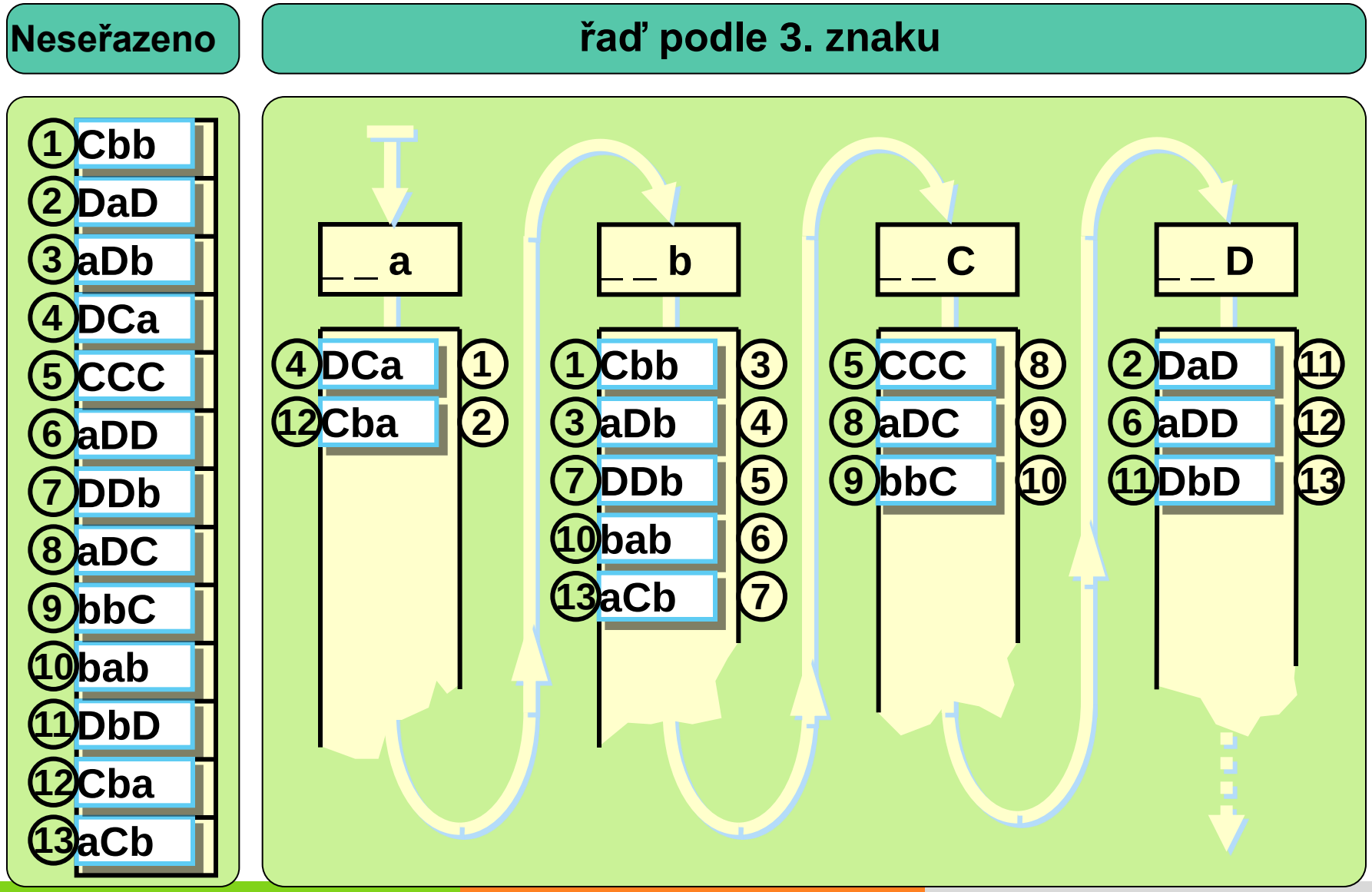
Speciální algoritmy řazení v čase O(n)

- 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

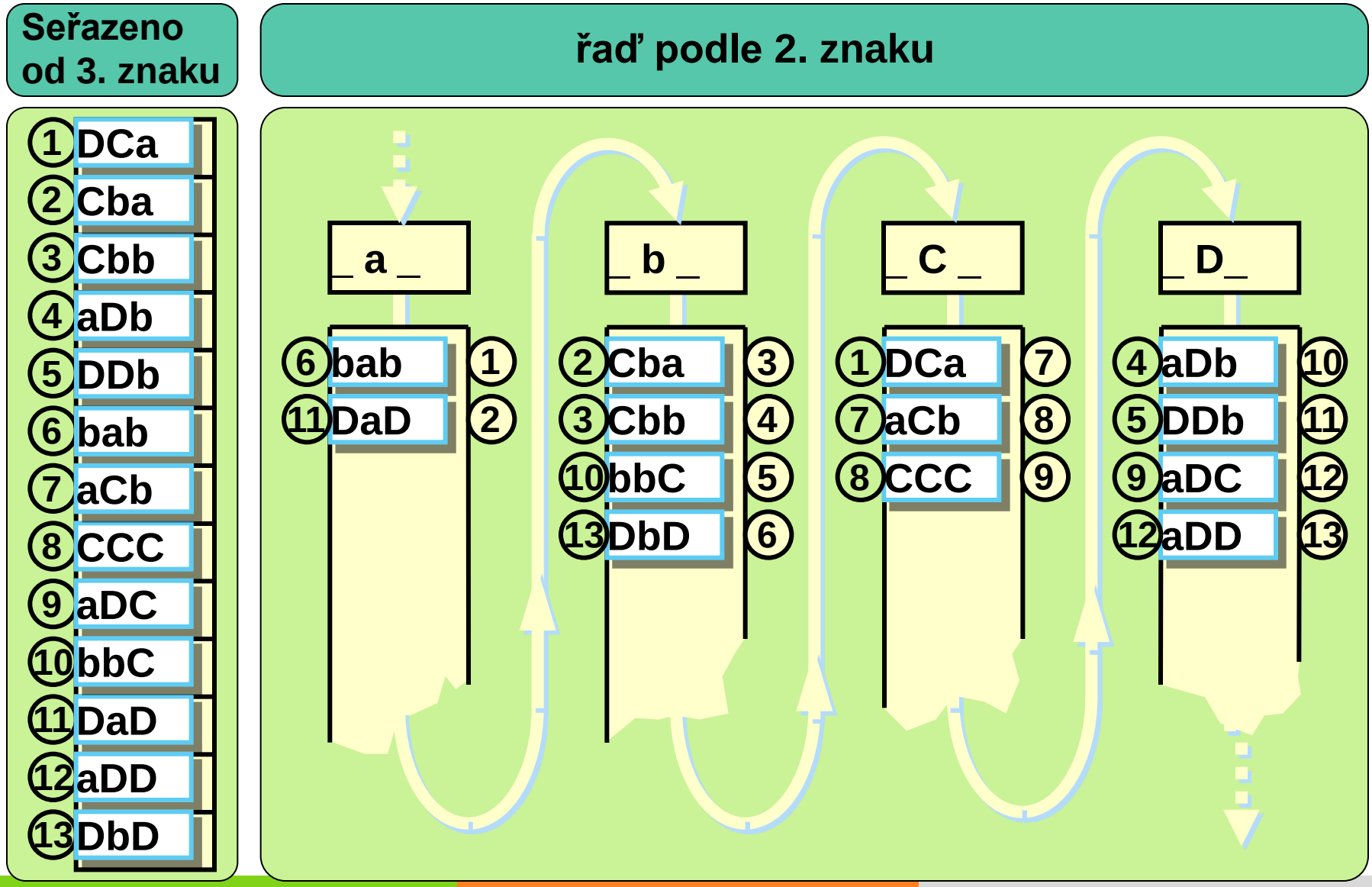
Číslicové řazení

RADIX-SORT

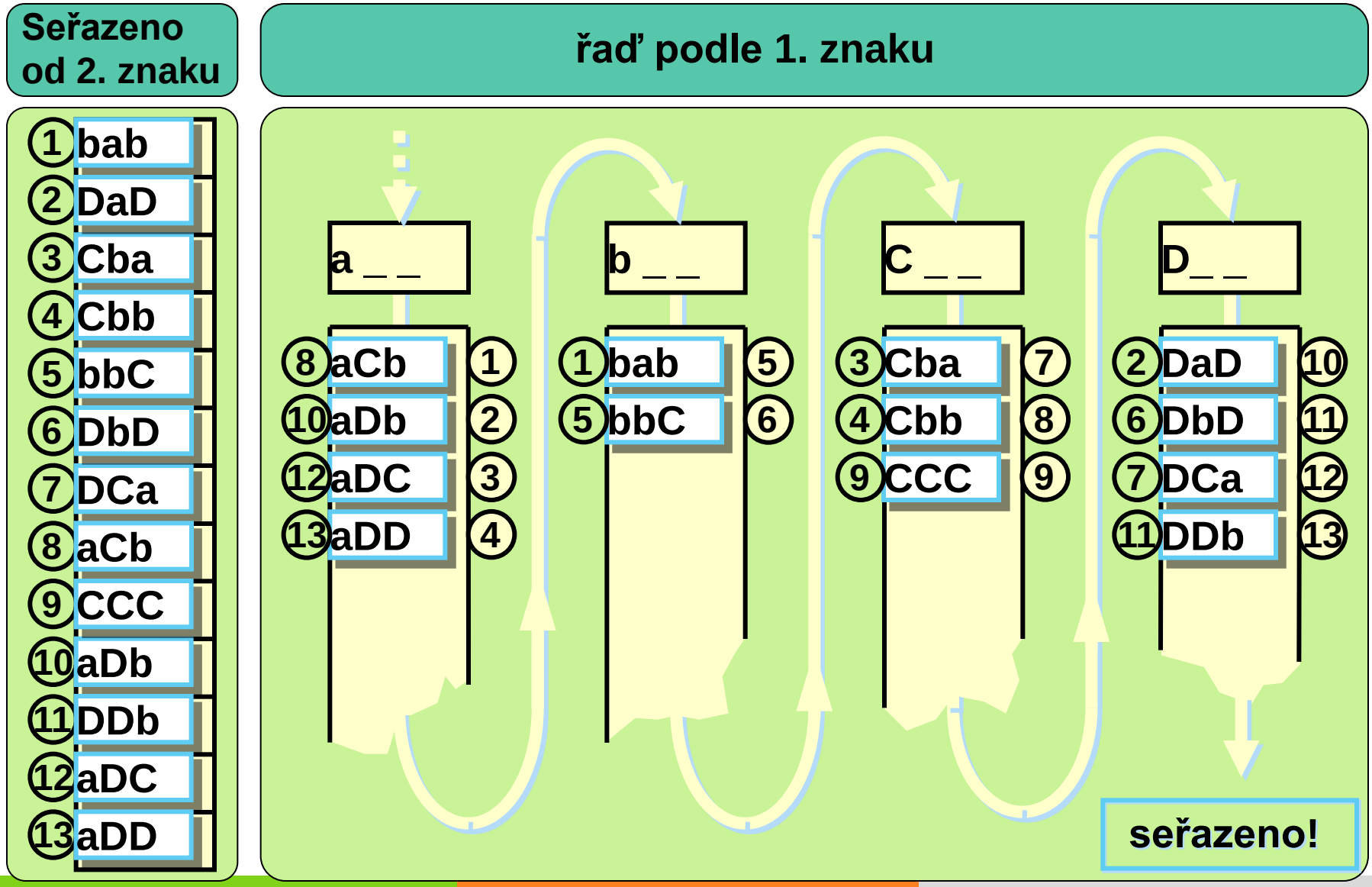
RADIX-SORT


Neseřazeno
řaď podle 3. znaku
1 Cbb
2 DaD
3 aDb
4 DCa
5 CCC
6 aDD
7 DDb
8 aDC
9 bbC
10 bab
11 DbD
12 Cba
13 aCb
_ _ a
4 DCa 1
12 Cba 2
_ _ b
1 Cbb 3
3 aDb 4
7 DDb 5
10 bab 6
13 aCb 7
_ _ C
5 CCC 8
8 aDC 9
9 bbC 10
_ _ D
2 DaD 11
6 aDD 12
11 DbD 13

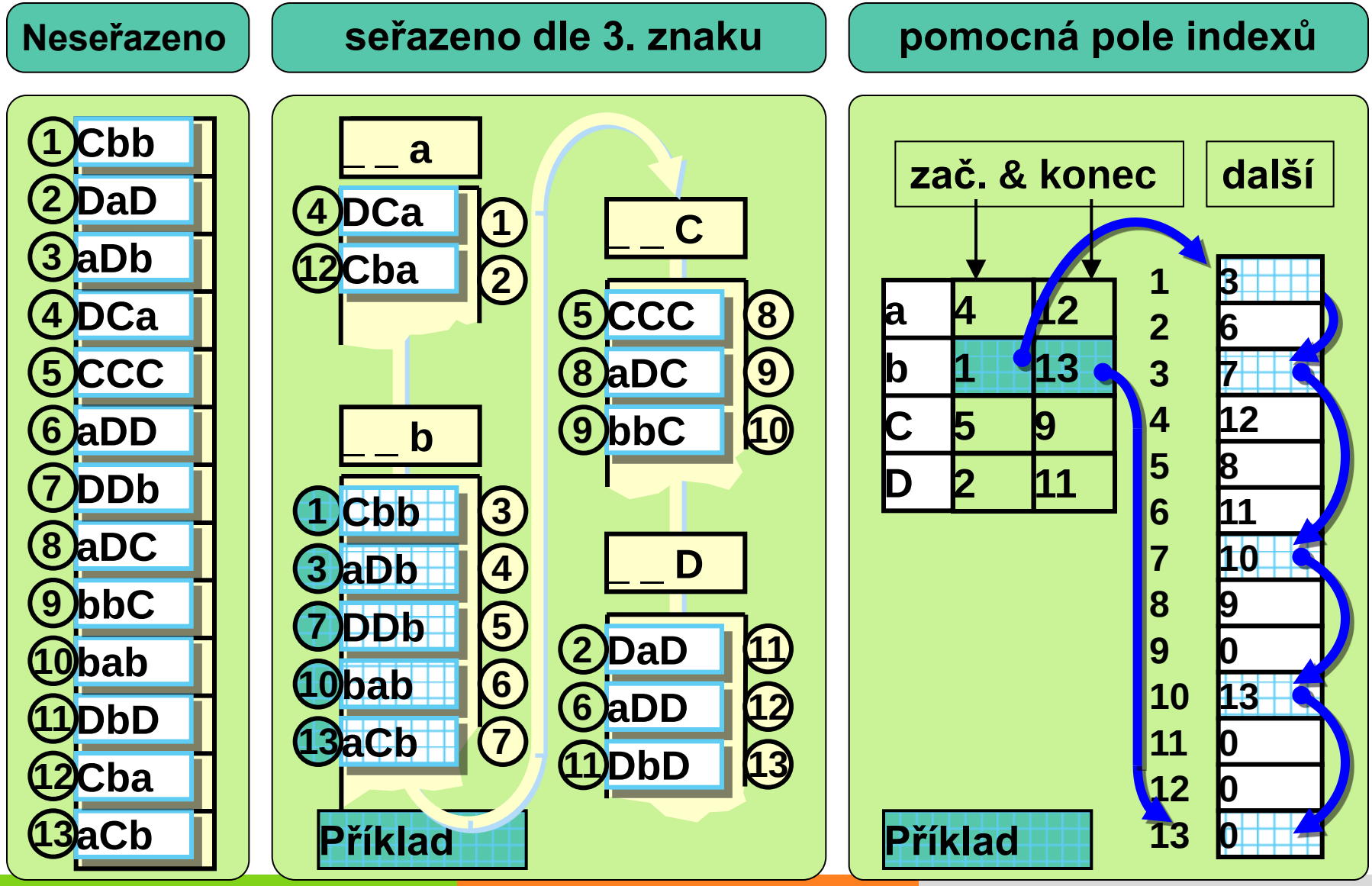
RADIX-SORT


Seřazeno od 3. 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
řaď podle 2. znaku
_ a _
6 bab 1
11 DaD 2
_ b _
2 Cba 3
3 Cbb 4
10 bbC 5
13 DbD 6
_ C _
1 DCa 7
7 aCb 8
8 CCC 9
_ D _
4 aDb 10
5 DDb 11
9 aDC 12
12 aDD 13

RADIX-SORT


Seřazeno od 2. znaku
řaď podle 1. znaku
1 bab
2 DaD
3 Cba
4 Cbb
5 bbC
6 DbD
7 DCa
8 aCb
9 CCC
10 aDb
11 DDb
12 aDC
13 aDD
a _ _
b _ _
C _ _
D_ _
8 aCb 1
10 aDb 2
12 aDC 3
13 aDD 4
1 bab 5
5 bbC 6
3 Cba 7
4 Cbb 8
9 CCC 9
2 DaD 10
6 DbD 11
7 DCa 12
11 DDb 13
seřazeno!

Implementace RADIX-SORT



Příklad běhu RADIX-SORT

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

Vstup	$i = 1$	$i = 2$	$i = 3$	$i = 4$
2101	012**0**	2**1**01	**1**001	**0**021
1022	210**1**	1**0**01	**0**021	**0**120
2211	221**1**	1**2**02	**1**022	**1**001
0120	002**1**	2**2**11	**2**101	**1**022
0021	111**1**	1**1**11	**1**111	**1**111
1202	212**1**	0**1**20	**0**120	**1**202
1111	100**1**	0**0**21	**2**121	**2**101
2121	102**2**	2**1**21	**1**202	**2**121
1001	120**2**	1**0**22	**2**211	**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** seřaď 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

$d << n \Rightarrow$ ..... $\Theta(n)$ 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

1	2	3	4	5	6	7	8	9	10	11	12
10	3	8	8	6	3	8	10	6	10	8	5

četnost
cetnost.length == k
k = max(vstup) - min(vstup) + 1

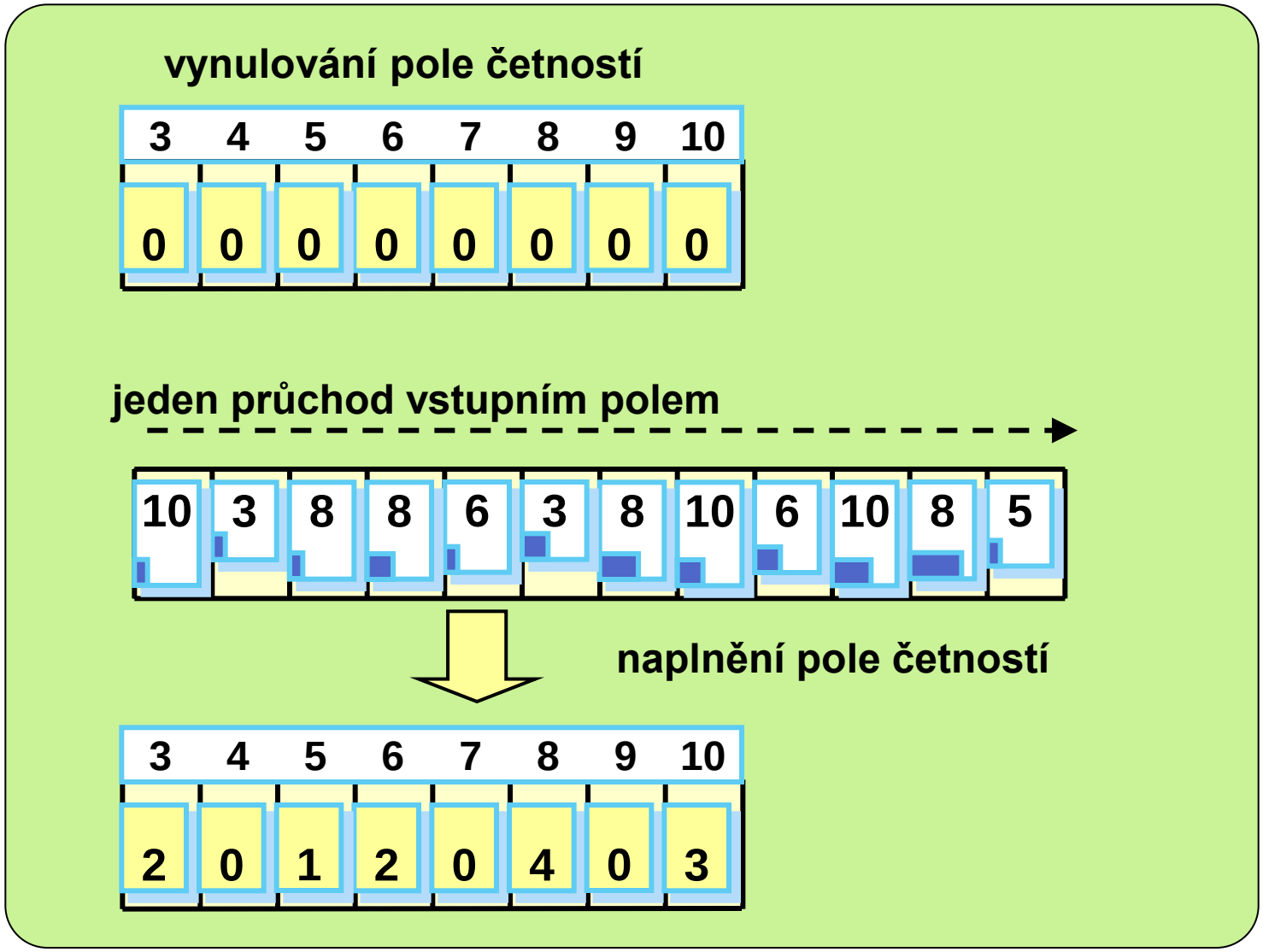
3	4	5	6	7	8	9	10
2	0	1	2	0	4	0	3

výstup
vstup.length == N

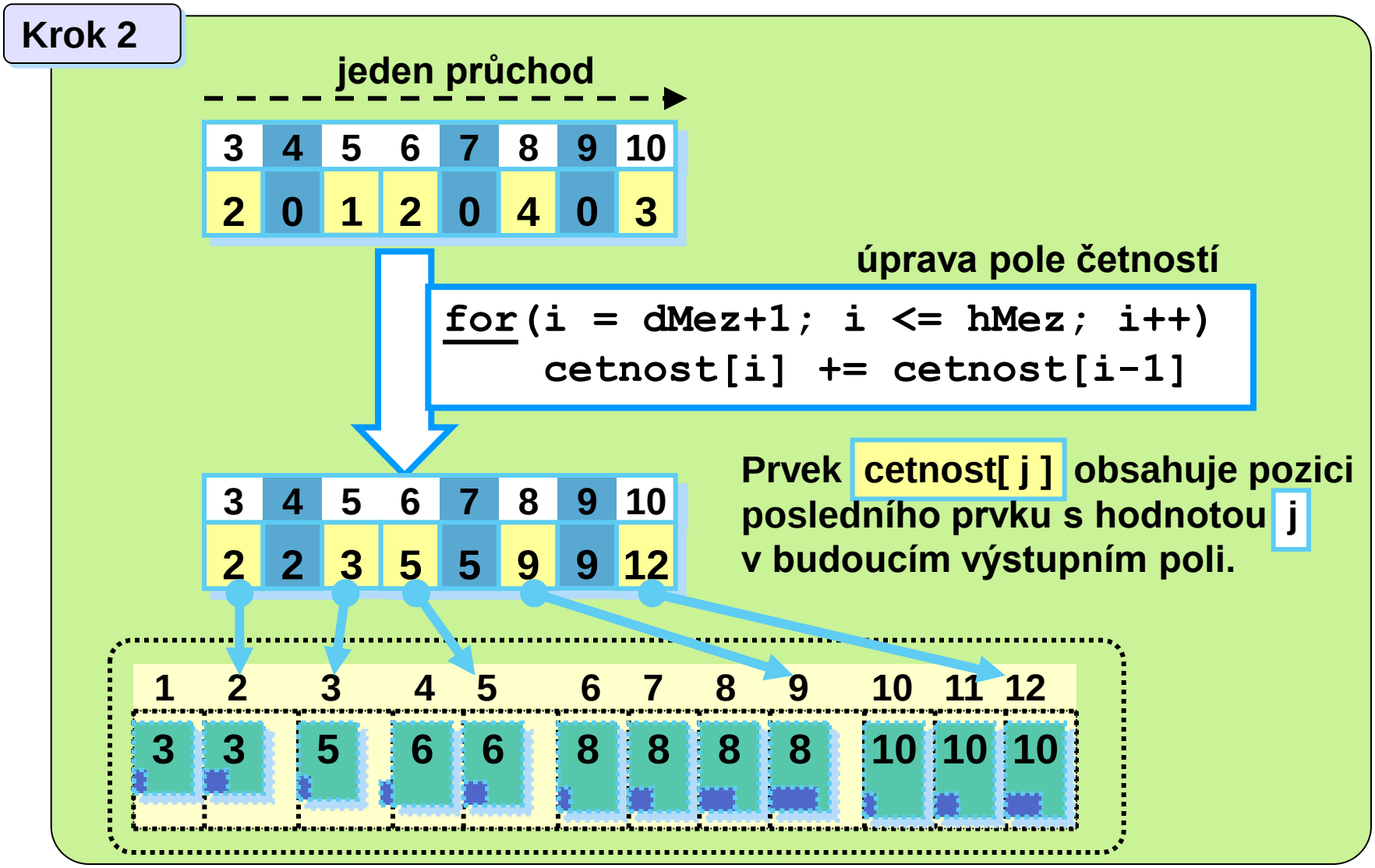
1	2	3	4	5	6	7	8	9	10	11	12
3	3	5	6	6	8	8	8	8	10	10	10

COUNTING-SORT

Krok 1



COUNTING-SORT

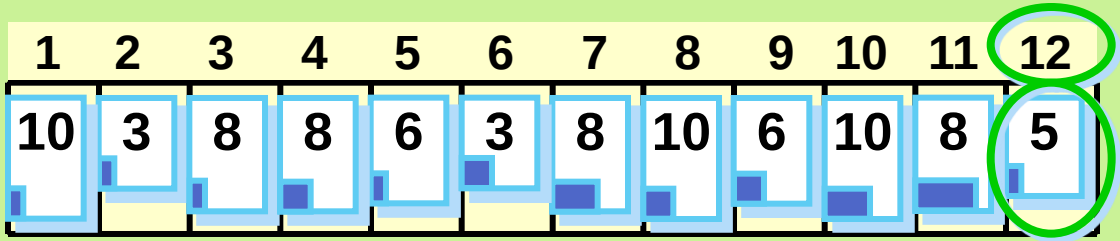


COUNTING-SORT

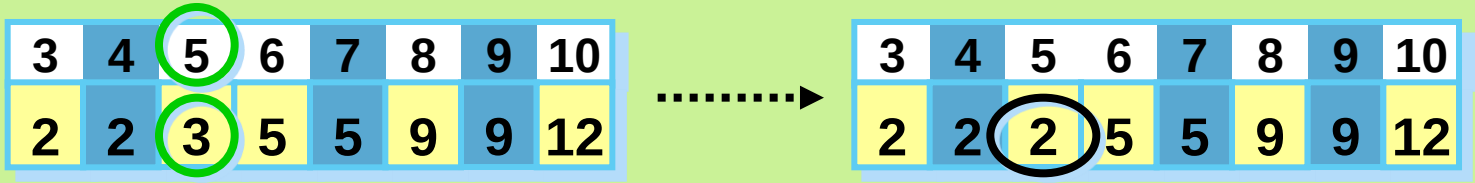
Krok 3

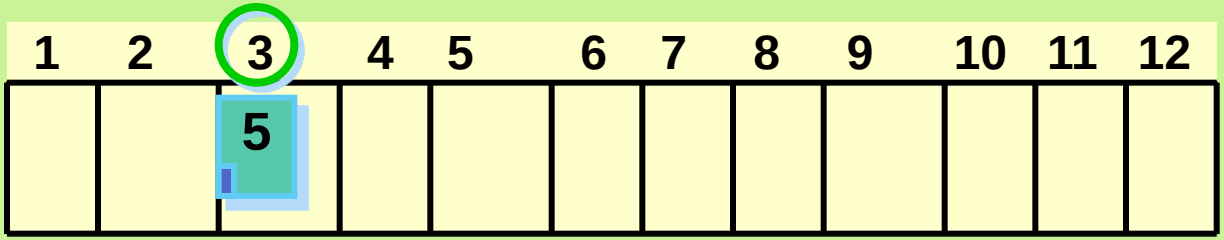
`i == N`

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



`i == 12`



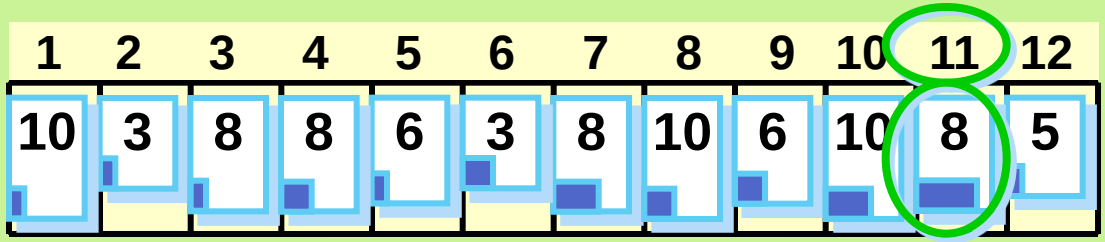


COUNTING-SORT

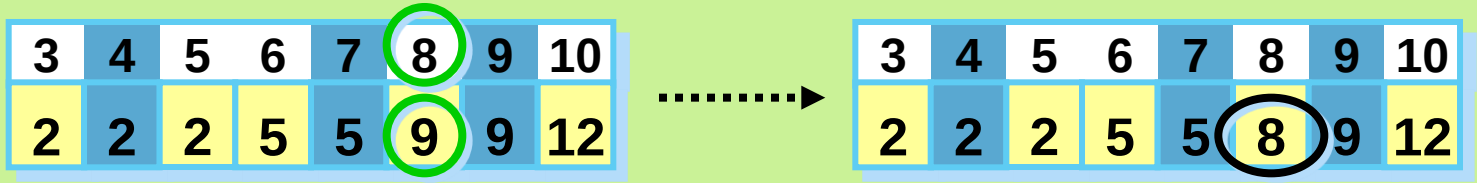
Krok 3

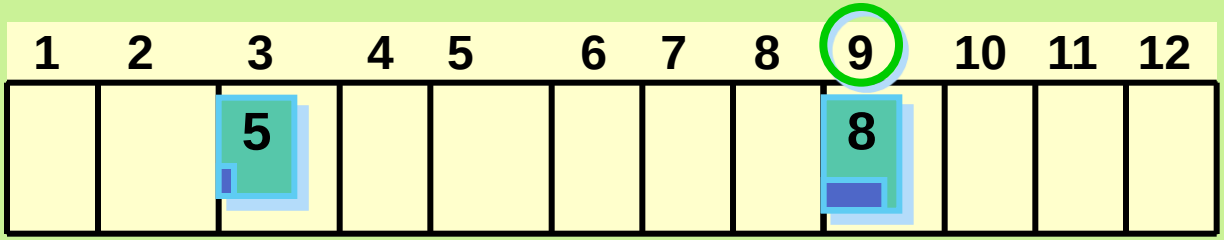
`i == N-1`

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



i == 11



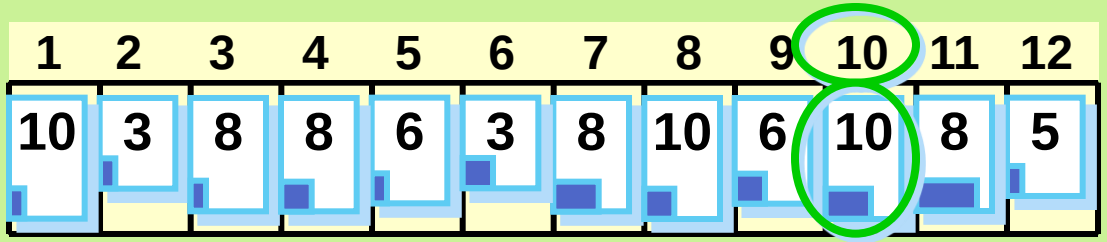


COUNTING-SORT

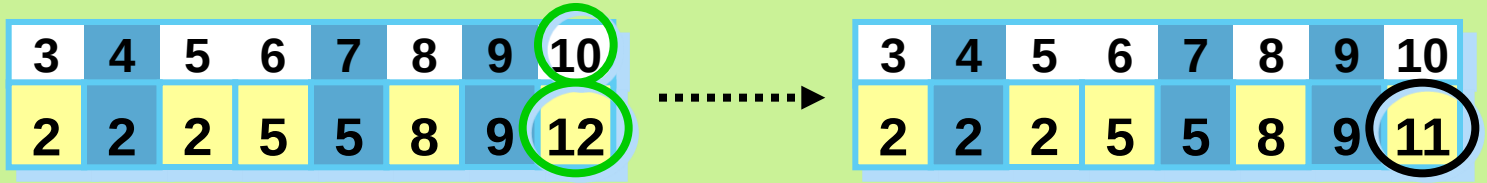
Krok 3

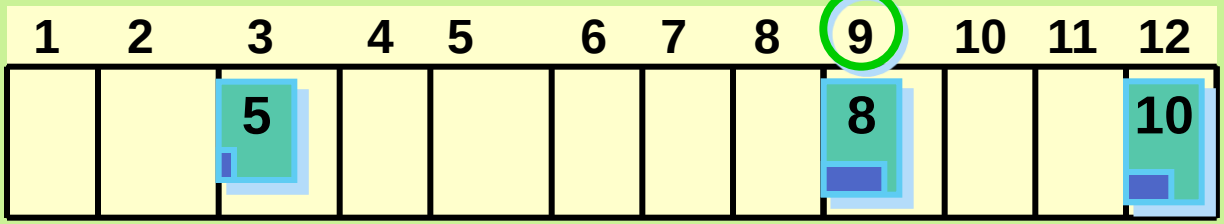
`i == N-2`

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



`i == 10`





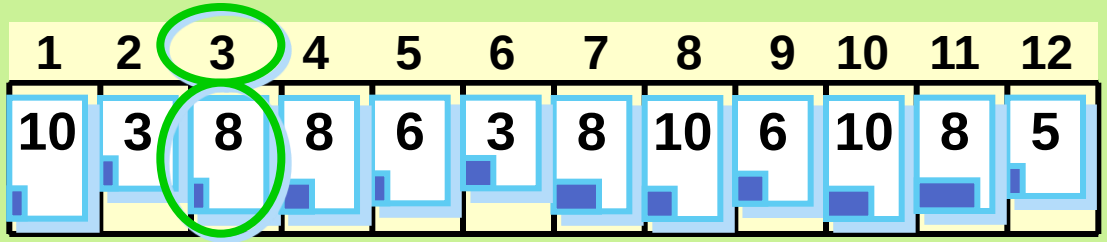
atd...

COUNTING-SORT

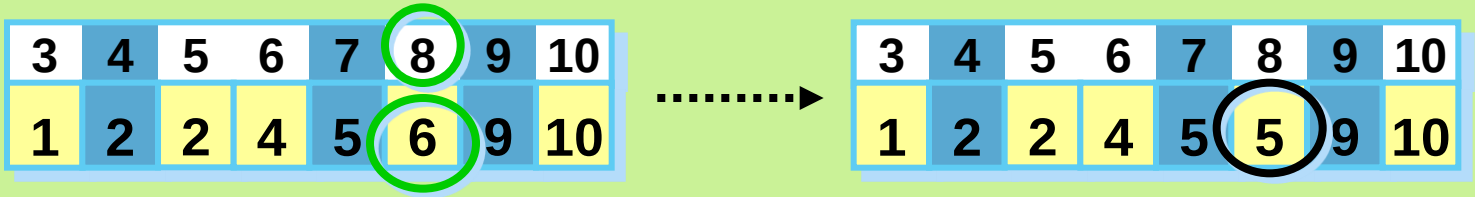
Krok 3

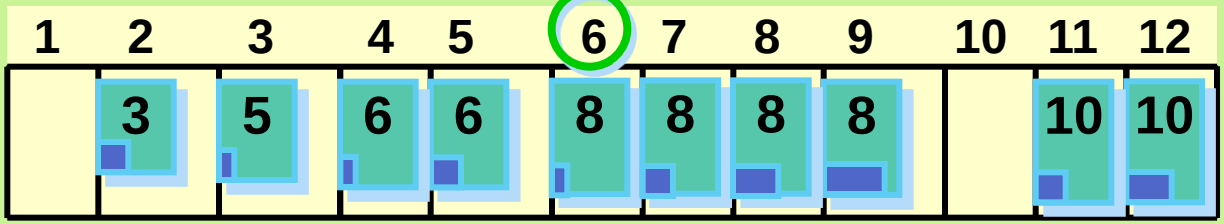
`i == 3`

```
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

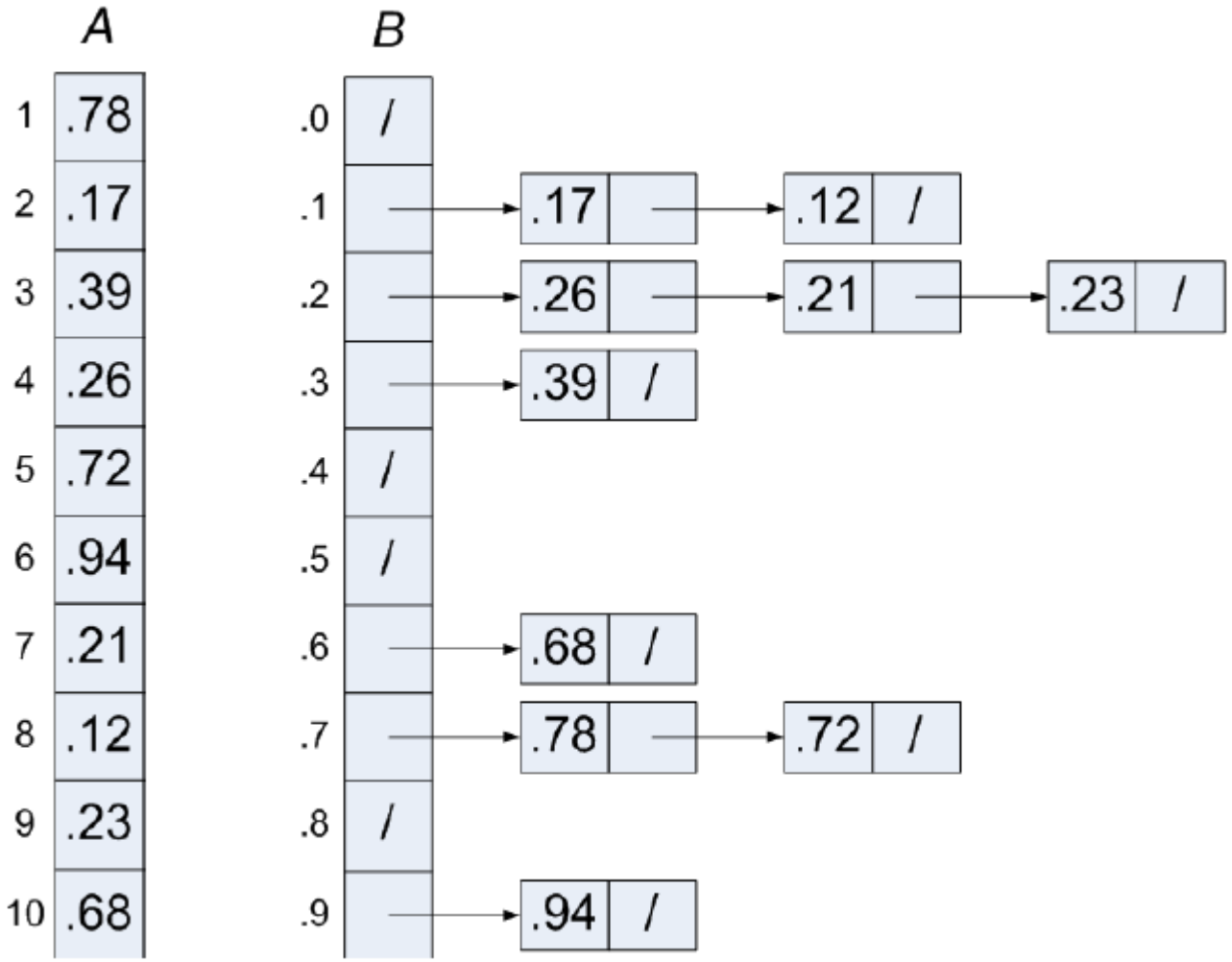
BUCKET-SORT

BUCKET-SORT

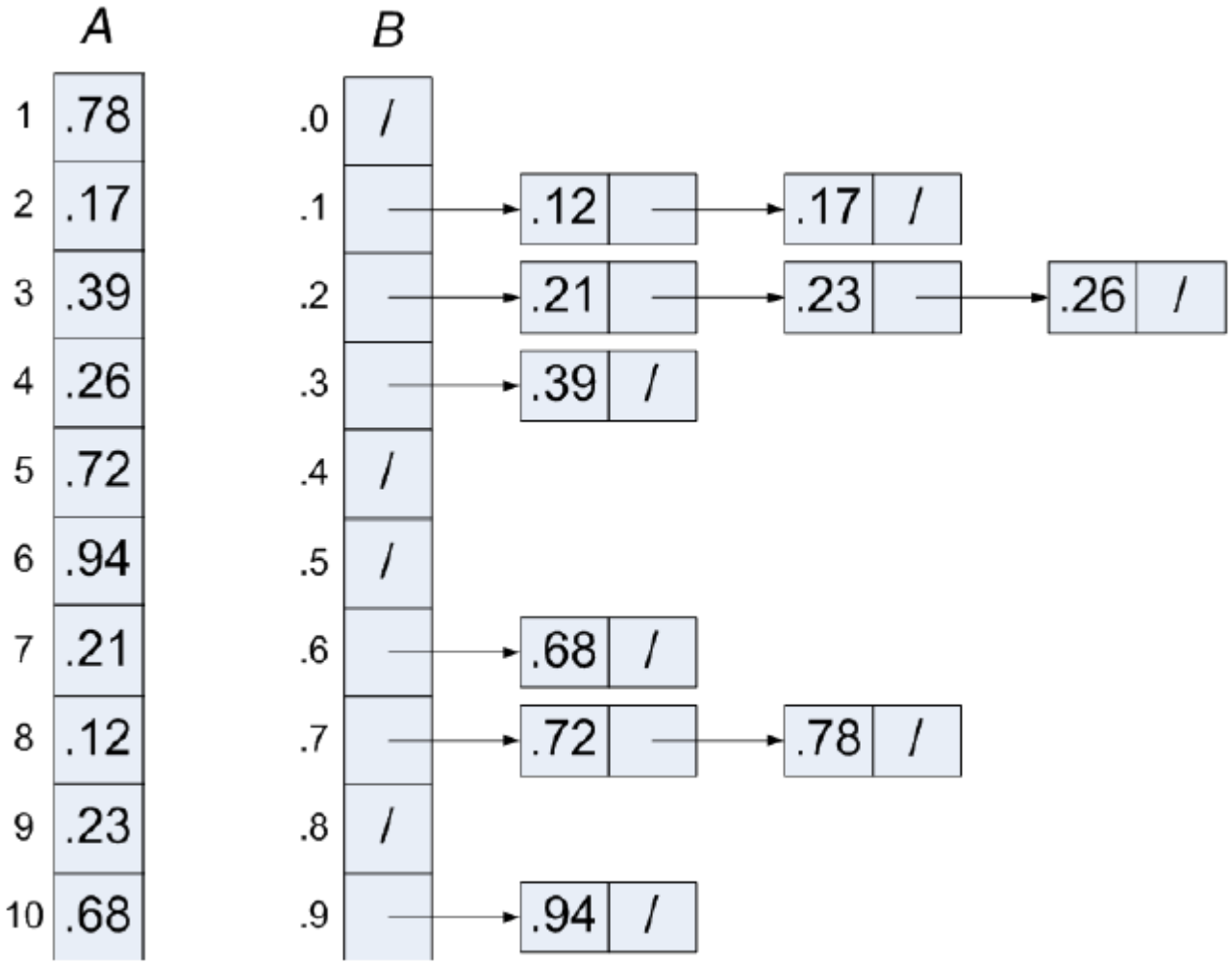
- Vstupní čísla jsou z intervalu [0, 1). Algoritmus je vhodný, pokud jsou vstupní čísla rovnoměrně rozložena v tomto intervalu.
- Hlavní myšlenky:
  - Interval [0, 1) rozdělíme na n přihrádek, do kterých vstupní pole distribuujeme.
  - Díky předpokladu rovnoměrnosti rozdělení předpokládáme, že do jedné 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 ← length(A);
(2)   for i ← 1 to n
(3)     do připoj A[i] k seznamu v přihrádce B[⌊n * A[i]⌋];
(4)   for i ← 0 to n − 1
(5)     do seřaď seznam B[i] stabilním alg. (např. InsertionSortem)
(6)   zřetěz seznamy B[0], B[1], B[2], ..., 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 stabilnosti řazení v přihrádkách nezmění pořadí.
- Nechť padnou do různých přihrádek B[i'] a B[j']. Nechť 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 $\Theta(n*k)$.

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

Řazení na vnějších médiích

Řazení na vnějších médiích

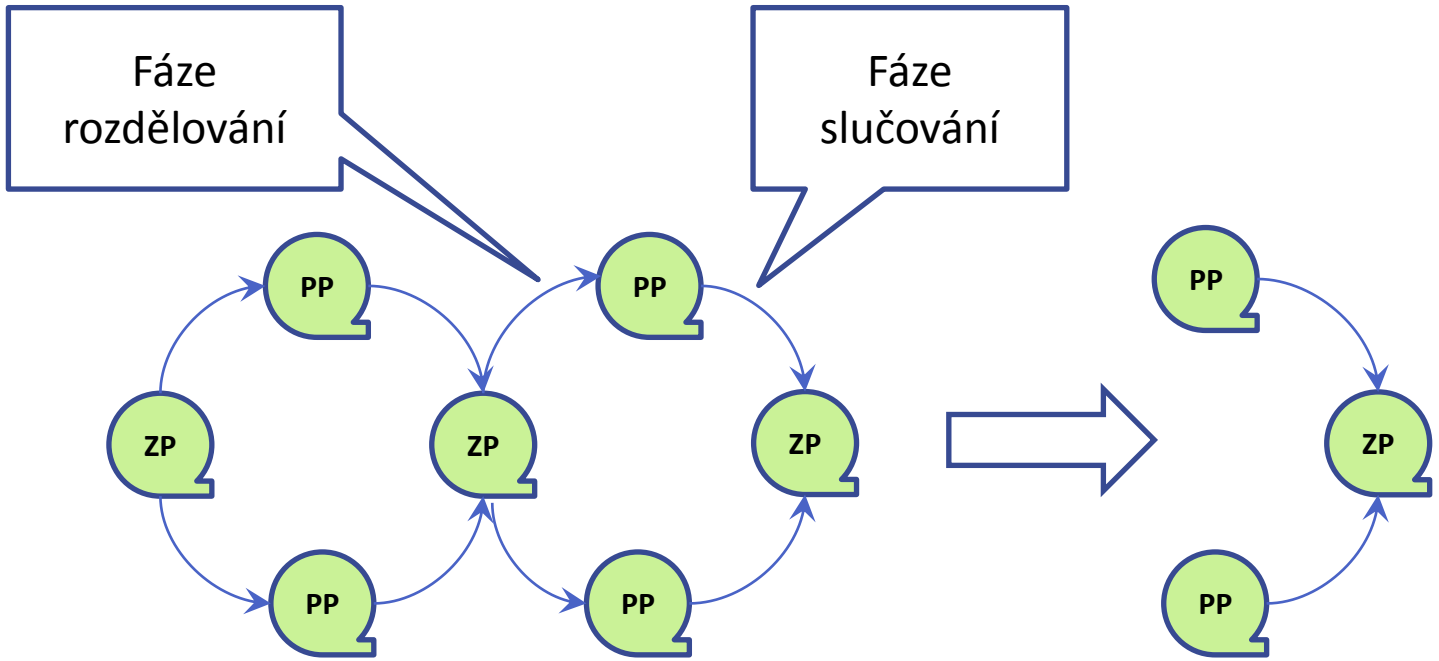
- Metody řazení lze rozlišit na tzv. **vnitřní** řazení (řazení v paměti) a tzv. **vnější** řazení (využívající vnější média).
- **Vnější** řazení lze dále rozlišit na práci s médii se **sekvenčním** přístupem (páska) a **náhodným** přístupem (disk).
- Základním principem řadících metod na vnějších, 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 $2^{k+1}>=n$. 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