STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Triedenie – Usporadúvanie 2

28. 4. 2021

letný semester 2020/2021

prednášajúci: Lukáš Kohútka

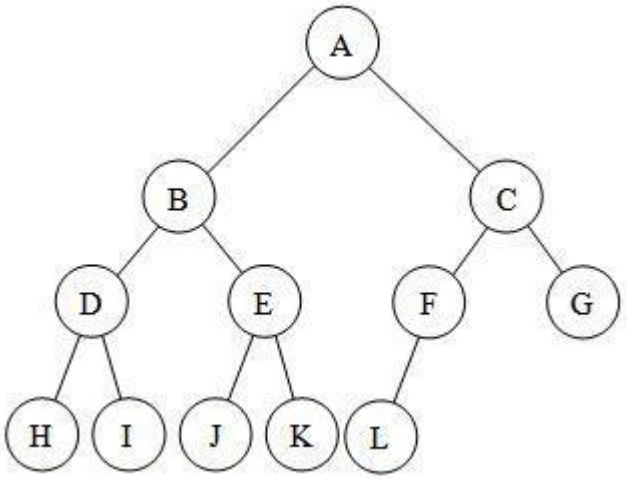
Tree sort a Heap sort

- Heapsort používa binárnu haldu ako dátovú štruktúru
- Naplníme binárnu haldu všetkými údajmi, ktoré chceme zoradiť
- Potom postupne voláme Extract-Min (alebo Max) až kým nevyberieme všetky prvky
- Po každej extrakcii použijeme Select sort

- Tree sort používa binárny strom ako dátovú štruktúru
- Naplníme binárny strom všetkými údajmi, ktoré chceme zoradiť
- Potom použijeme in-order prehľadávanie

Halda (heap)

- Využíva špeciálny typ binárneho stromu
  - Tzv. úplný binárny strom (complete binary tree)
  - Na každej úrovni je úplne naplnený, okrem možno poslednej úrovne (rozdiel oproti plnému binárnemu stromu)
- Halda poskytuje len obmedzené operácie
  - **insert** (pridať prvok), **delete** (odstrániť prvok)
  - **getmax** (vyhľadať najväčší prvok)
- Implementácia pomocou BVS:
  - insert /delete **O(n)**, getmax **O(1)**
- Cieľ je vyváženejšia zložitosť:
  - insert /delete /getmax **O(log n)**



ADT Prioritný rad / front (Priority queue)

- Množina prvkov, ktorým je pridelená priorita (kľúč) – je ich možné podľa priority porovnávať.
- Prvky je možné **vkladať v akomkoľvek poradí** s rôznou prioritou avšak, pri výbere sa **vyberá vždy len prvok s najvyššou prioritou**.

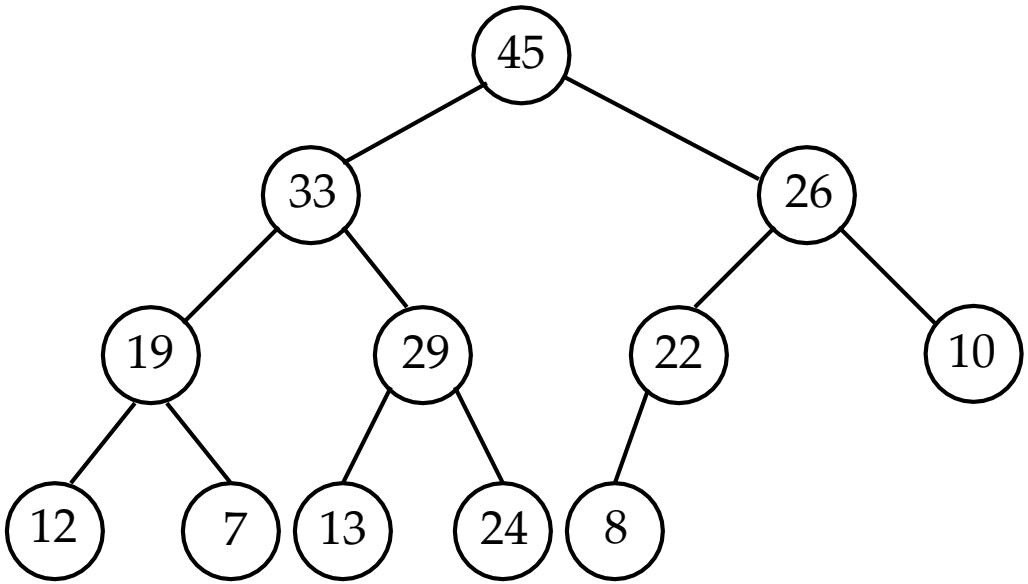
- Operácie:
  - **insert(S,x)** – vloženie prvku x do množiny S
  - **maximum(S)** – vrátenie prvku s najväčším kľúčom
  - **removeMax(S)** – odstránenie prvku s najväčším kľúčom

Implementácia pomocou spájaného zoznamu

- **insert** – pridávanie prvkov na začiatok zoznamu
  - **O(1)**
- **maximum/removeMax** – nájdenie prvku s najväčšou prioritou, ten sa vymaže
  - **O(N)**

Implementácia pomocou binárnej haldy

- Binárna halda je úplný binárny strom, pre ktorý platí, že **hodnota kľúča vo vrchole je väčšia alebo rovná hodnotám kľúčov jeho nasledovníkov**



Vlastnosti binárnej haldy

- Binárna halda má voľnejšie pravidlá usporiadania kľúčov (umiestnenie prvkov) ako binárny vyhľadávací strom
- Nemusí platiť, že ľavý podstrom obsahuje prvky s nižšími hodnotami kľúčov ako pravý podstrom!
- Platí tzv. **haldová vlastnosť**:

kľúč(PARENT(i)) ≥ kľúč(i)
pre všetky vrcholy i okrem koreňa

Dôsledok: koreň stromu (binárnej haldy) má vždy najväčšiu hodnotu kľúča (≥ ako ostatné vrcholy).

Binárna halda - Implementácia vektorom

- Koreň stromu na 1. pozícii **`heap[1]`**
- Nasledovníky vrchola zapísaného na i-tej pozícii vektora sú (ak existujú):
  - **`left(i) = 2*i`**
  - **`right(i) = 2*i + 1`**
  - **`parent(i) = |i/2|`**
- **`heap[i..j]`**, kde `i>=1`, je binárna halda práve vtedy, ak každý prvok nie je menší ako jeho nasledovníky.

- Operácia **`maximum`** – prvok `heap[1]`
  - `O(1)`

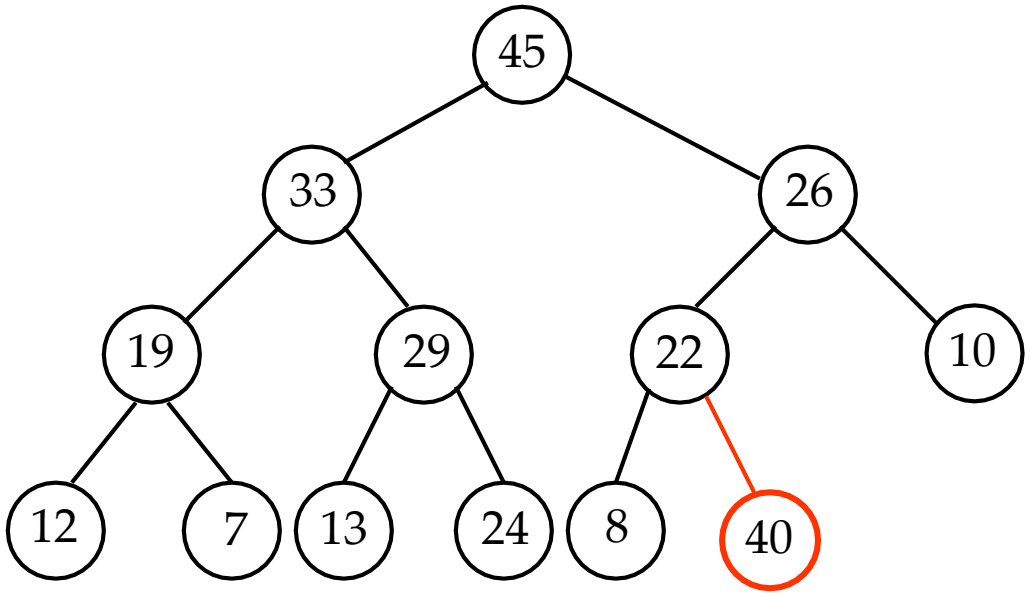
Binárna halda – Operácia insert

1. Vytvorí sa nový vrchol na najnižšej úrovni, označme **v**
2. Ak je **v** koreň stromu (haldy), končíme.
3. Ak kľúč(**v**) ≤ kľúč(parent(**v**)), končíme.
4. Inak (ak kľúč(v) > kľúč(parent(**v**))), vymeníme vrchol **v** so svojím rodičom, a
   pokračujeme na krok 2 pre **v** ← **parent(v)**

   (Ak je kľúč vrchola **v** väčší ako kľúč nového rodiča, vymení sa aj s ním, … opakujeme, kým nie je strom opäť haldou – spĺňa haldovú vlastnosť)

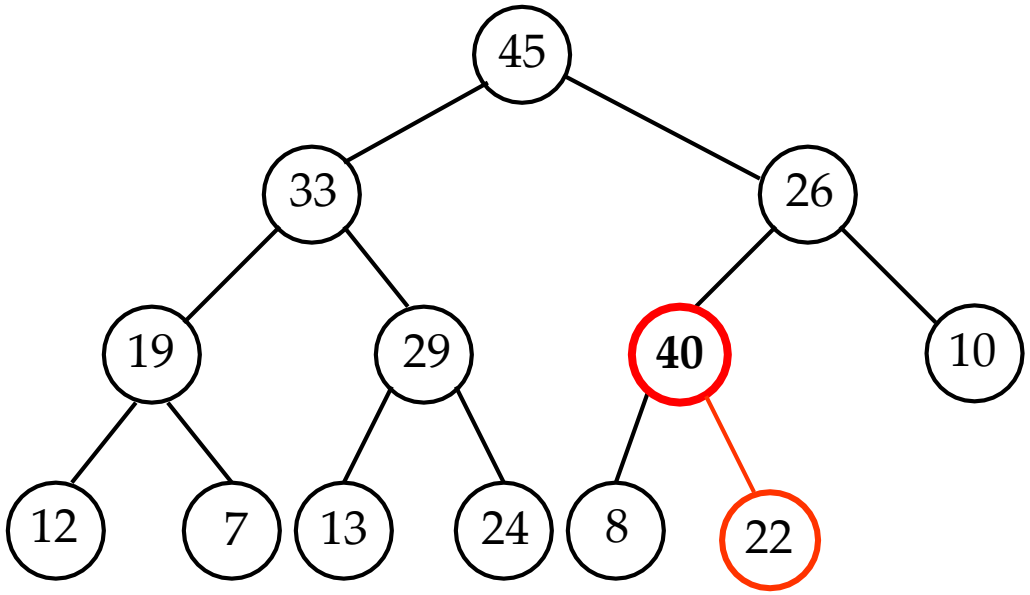
Binárna halda - Vykonanie insert(40)

- Vloženie na najnižšiu úroveň



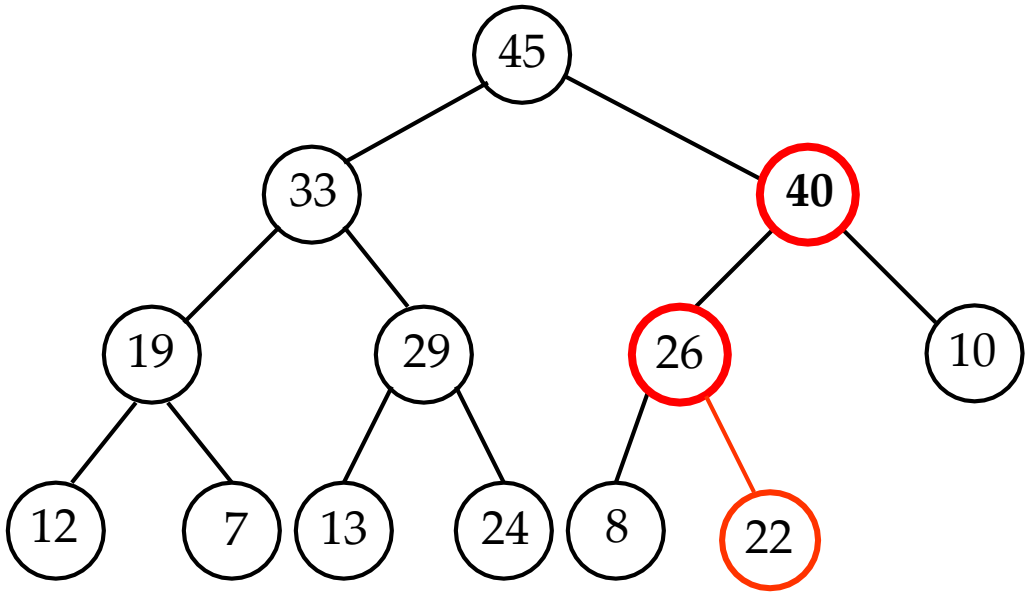
Binárna halda - Vykonanie insert(40)

- Prvá výmena s rodičom (22)



Binárna halda - Vykonanie insert(40)

- Druhá výmena s rodičom (26)



- Hotovo – obnovená haldová vlastnosť
  - v každom vrchole platí **kľúč(PARENT(i)) ≥ kľúč(i)**

Binárna halda - Insert (pseudokód)

```
Heap-INSERT(heap, key):
  heap-size (heap) = heap-size(heap) +
  1   i = heap-size (heap)
  while i > 1 and heap[PARENT(i)] <
      key  do heap[i] = heap[PARENT(i)]
           i = PARENT(i)
  heap[i] = key
```

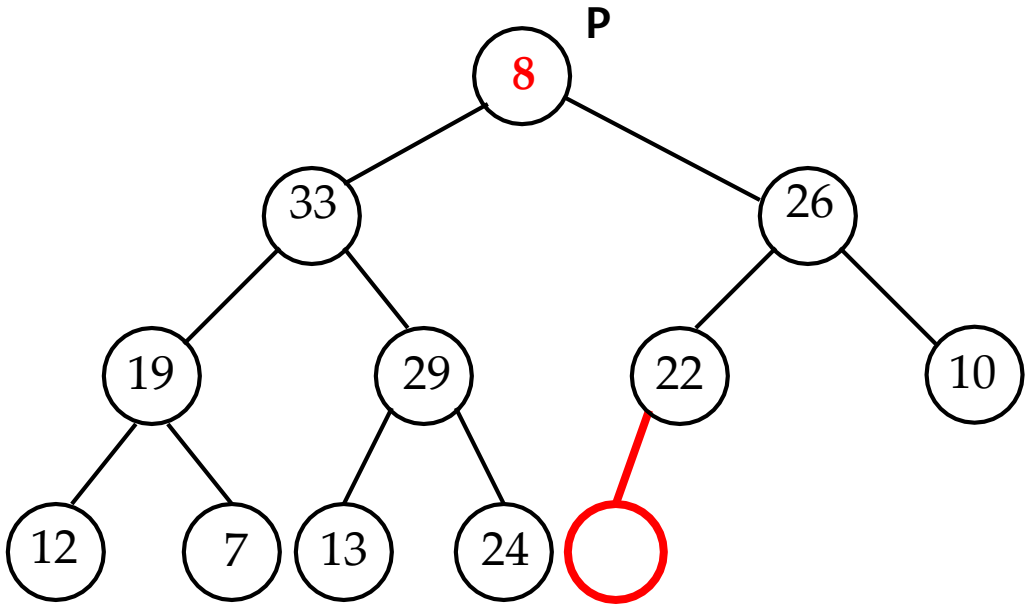
- Zložitosť?
  - O(log n), kde n je počet prvkov v halde
  - Pretože: úplný binárny strom s n prvkami má hĺbku O(log n)

Odstránenie najväčšieho prvku z binárnej haldy

- Odstránime koreň haldy.
- Odstránime najpravejší vrchol na najnižšej úrovni (jeho kľúč označme P) a hodnotu P zapíšeme do koreňa
  - Mohli sme porušiť haldovú vlastnosť!
- Obnovíme haldovú vlastnosť smerom dole
  1. Ak P je list, končíme.
  2. Označme Q najväčšiu z hodnôt priamych potomkov P
     Ak Q ≤ P, teda v potomkoch nie sú väčšie kľúče, končíme.
  3. Inak (ak Q > P) vymeníme vrchol P s vrcholom Q, pokračujeme na krok 1 pre nižšie umiestnený vrchol P.

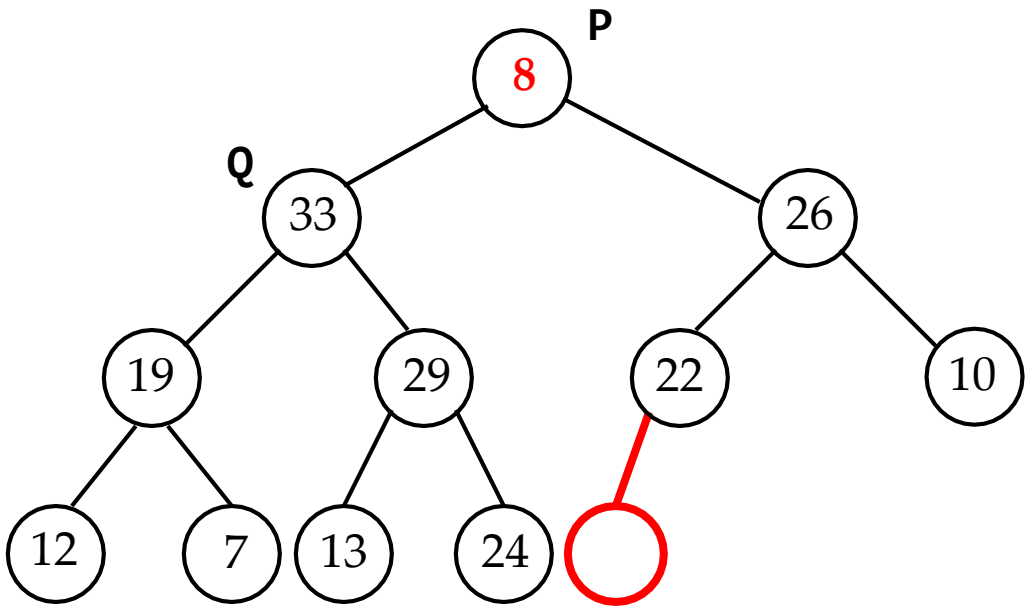
Binárna halda - Vykonanie removeMax

- Odstránime koreň haldy
- Odstránime najpravejší vrchol na najnižšej úrovni (jeho kľúč označme P) a hodnotu P zapíšeme do koreňa



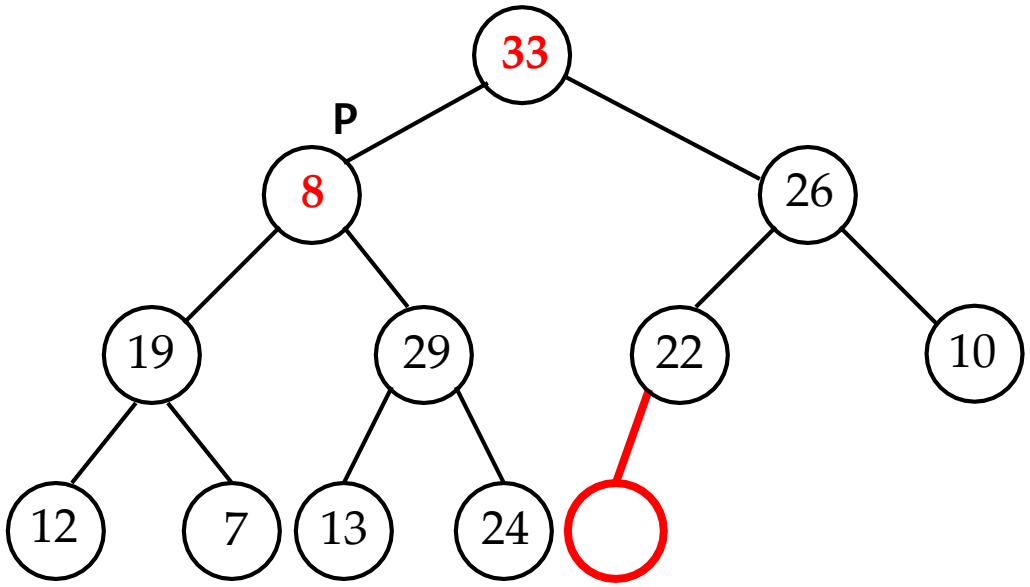
Binárna halda - Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P



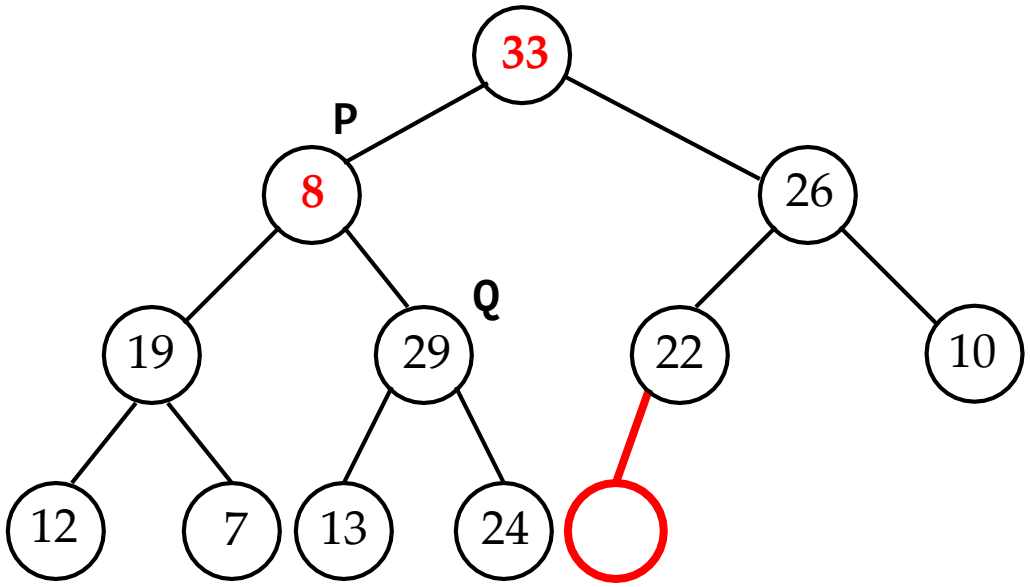
Binárna halda - Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P
- Ak Q > P vymeníme vrchol P s vrcholom Q



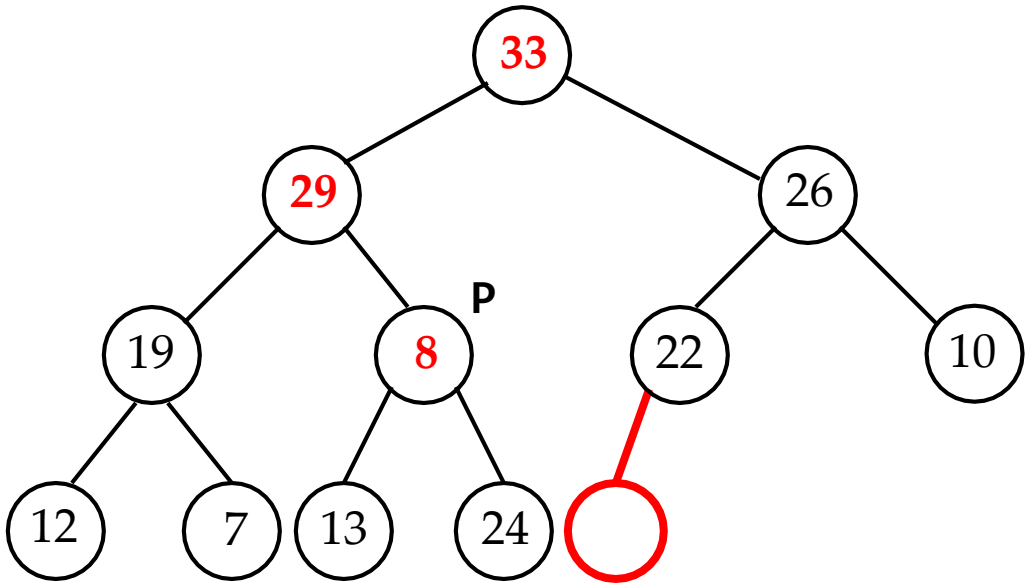
Binárna halda - Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P



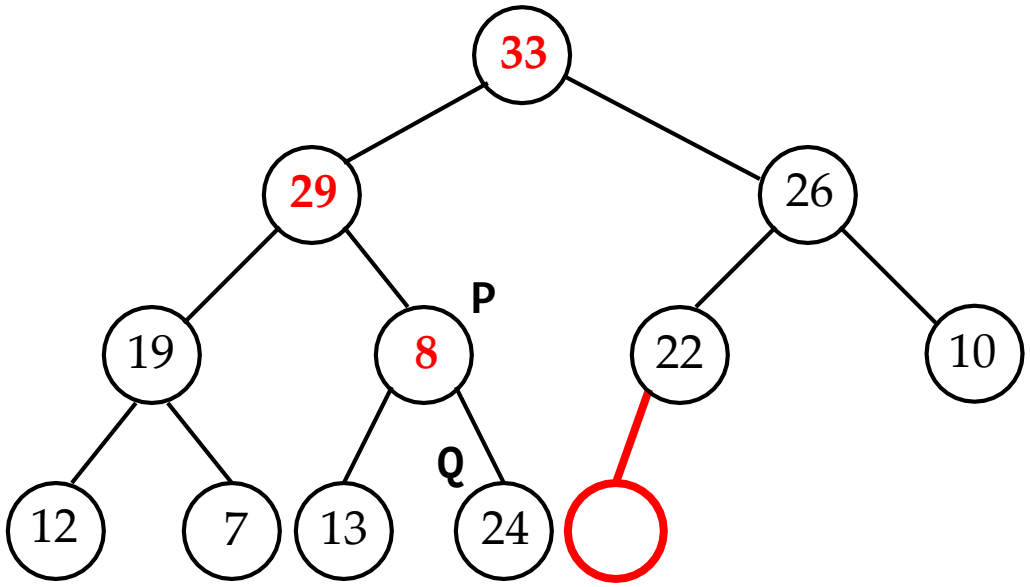
Binárna halda - Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P
- Ak Q > P vymeníme vrchol P s vrcholom Q



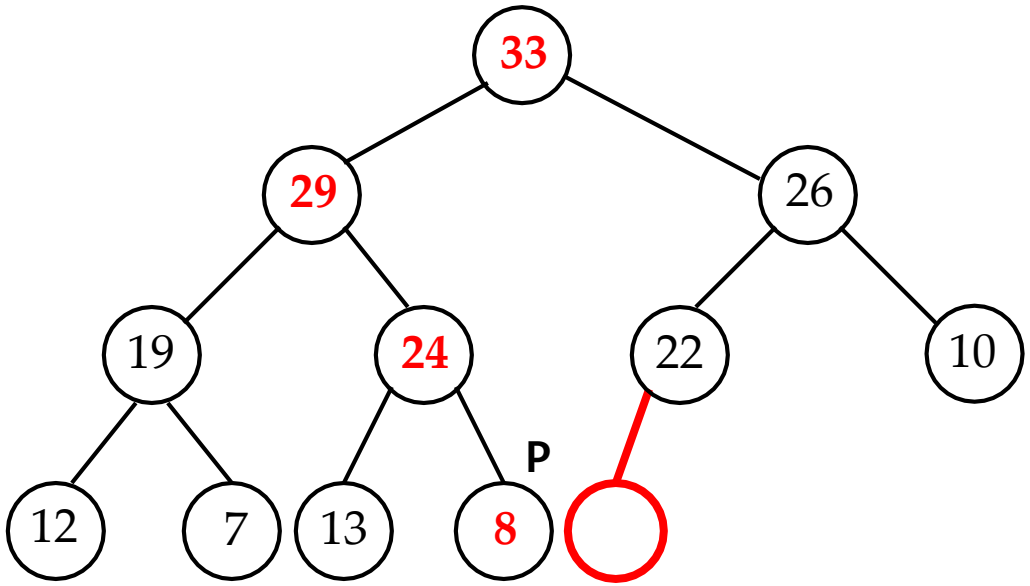
Binárna halda - Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P
-



Binárna halda - Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P
- Ak Q > P vymeníme vrchol P s vrcholom Q
- Ak P je list, končíme.



Binárna halda - extractMax (pseudokód)

```
Heap-EXTRACT-MAX(heap)
  if heap-size(heap) < 1
     then error
  max = heap[1]
  heap[1] = heap[heap-size(heap)]
  heap-size(heap) = heap-size(heap)-1
  HEAPIFY(heap, 1)
  return max
```

Binárna halda - heapify (pseudokód)

```
HEAPIFY(heap, i)
   lavy = left(i)
   pravy = right(i)
   if lavy <= heap-size(heap) and heap[lavy] > heap[i]
       then largest = lavy
       else largest = i
   if pravy <= heap-size(heap) and heap[pravy] > heap[largest]
       then largest = pravy
   if largest <> i
       then exchange (heap[i], heap[largest])
             HEAPIFY(heap, largest)
```

- Zložitosť?
  - O(log n), kde n je počet prvkov v halde

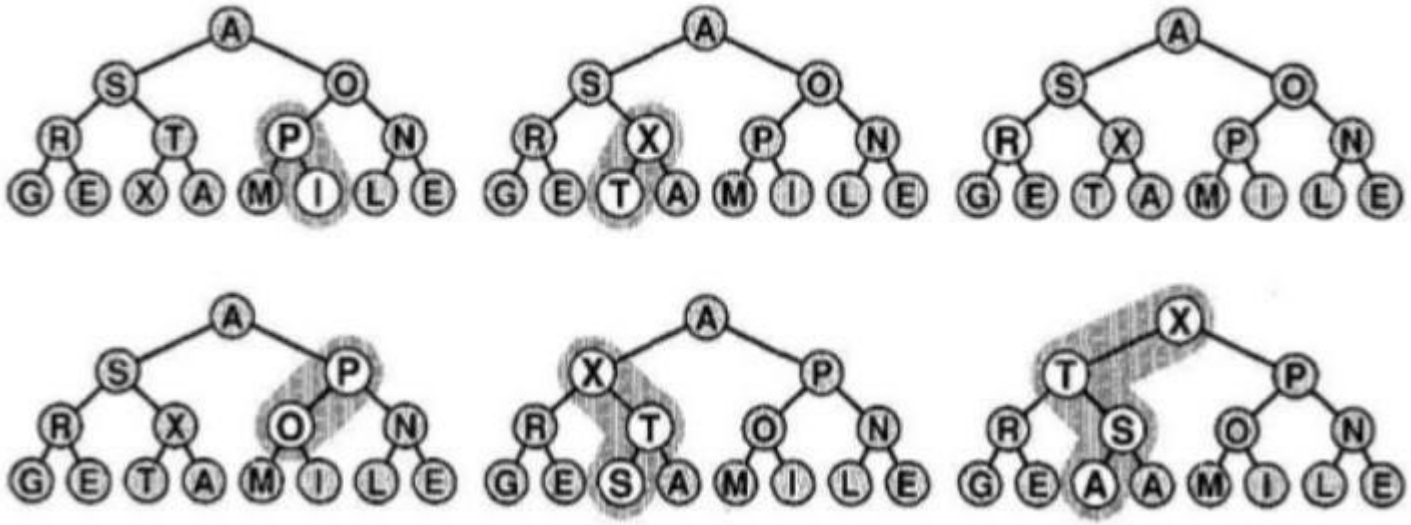
Binárna halda - vytvorenie haldy

- z vektora `heap[1..n]`, kde `n=length(heap)`
- všetky prvky v podvektore `heap[(⌊n/2⌋+1..n]` súlisty a teda aj 1-prvkové haldy
- Pseudokód:

```
BUILD-HEAP(heap):
  heap-size(heap) = length(heap)
  for i = ⌊length[heap] / 2⌋ downto 1
     do HEAPIFY(heap, i)
```

- Zložitosť?
  - Vo výške h je najviac $\frac{n}{2^{h+1}}$ vrcholov, heapify haldy výšky h trvá O(h)
  - $T_{\text{BUILD-HEAP}}(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O(2n) = O(n)$

Binárna halda - vytvorenie haldy (ukážka)



Usporadúvanie haldou (Heapsort)

- Pomocou haldy dokážeme spraviť efektívny triediaci algoritmus, tzv. **Heapsort**
- Postup: **Vytvorím haldu a postupne z nej vyberiem všetky prvky**
- **HEAP-SORT**(A):

```
    BUILD-HEAP(A);
    for i = length(A) downto 2 do
    { A[1] ⇔ A[i];
      heap-size(A) = heap-size(A)–1;
      HEAPIFY(A, 1)
    }
```

- Zložitosť?
  - Vytvorenie (O(n)) a n krát vybratie max (n*O(log n)) = O(n log n)

Porovnávacie algoritmy

- Algoritmus usporadúvania, ktorý prechádza vstupné kľúče a na základe operácie **porovnávania** rozhoduje, ktorý z dvoch prvkov sa má v usporiadanom poli objaviť ako prvý

- Operácia **porovnávania** musí mať tieto vlastnosti:

  Ak $a \leq b$ a $b \leq c$, tak $a \leq c$

  Pre všetky a a b, buď $a \leq b$ alebo $b \leq a$

- Základným limitom je dolné ohraničenie počtu potrebných porovnávaní $\Omega(n \log n)$, ktoré je v najhoršom prípade potrebné na usporiadanie postupnosti

Výhody porovnávacích algoritmov

- Výhody porovnávacách algoritmov
  - Použiteľné as-is pre rôzne dátové typy
    Čísla, reťace, ...
  - Jednoduchá implementácia porovnávania n-tíc v lexikografickom usporiadaní
  - Reverzná funkcia porovnávania = reverzne usporiadaná postupnosť

- Ako prekonať teoretický limit **Ω(n log n)** porovnávacích algoritmov?
  - Zbaviť sa porovnávania prvkov :)
    (budeme vyšetrovať štruktúru hodnôt kľúčov)
  - Obetovať priestorovú zložitosť

Usporadúvanie spočítavaním (Counting sort)

- Usporadúvanie výpočtom poradia
  - Neporovnávame kľúče!
  - Pokúsime sa priamo určiť jeho poradie v postupnosti
- Vstup:
  n čísel v rozsahu 0..k-1

- Určuje počet prvkov menších ako prvok *x*, pomocou čoho zistí správnu pozíciu prvku *x* vo vstupnom poli

Usporadúvanie spočítavaním (Counting sort)

- Určuje počet prvkov menších ako prvok *x*, pomocou čoho zistí správnu pozíciu prvku *x* vo vstupnom poli

- Algoritmus pracuje s tromi poliami:
  - Pole *a[0..n-1]* obsahuje údaje, ktoré sa majú usporiadať
  - Pole *b[0..n-1]* obsahuje konečný usporiadaný zoznam údajov
  - Pole *c[0..k-1]* je použité na počítanie počtu prvkov

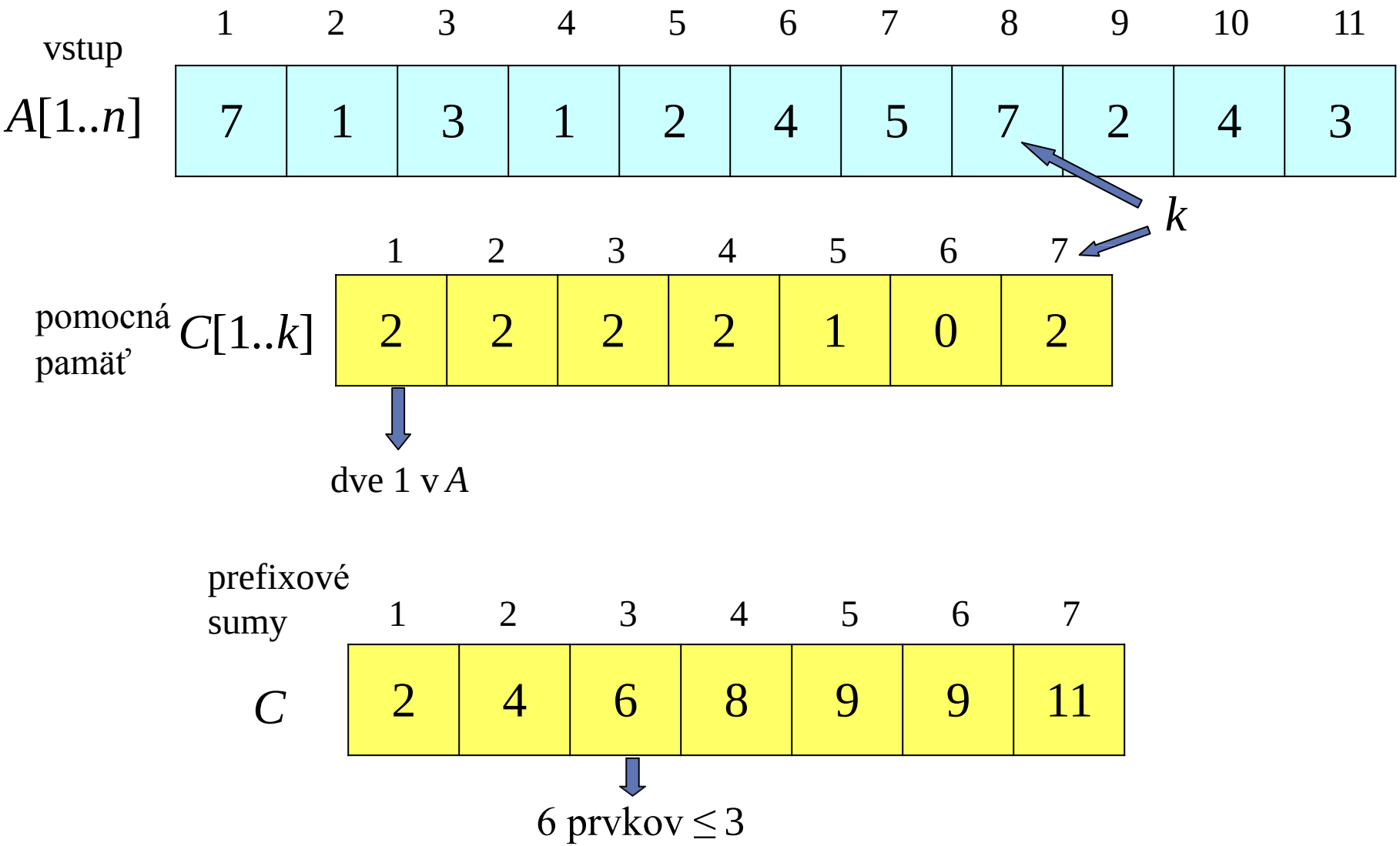
```
// pocet vyskytov konkretnej hodnoty
for(i = 0; i < n; i++)
  c[a[i]]++;

// prefixove sucty: urcime index posledneho prvku s hodnotou j
for(j = 1; j < k; j++)
  c[j] = c[j] + c[j-1];

// prvky z pola a vložíme na prislusny index v poli b
for(i = n-1; i >= 0; i--)
  b[--c[a[i]]] = a[i];
```

Usporadúvanie spočítavaním (Counting sort)

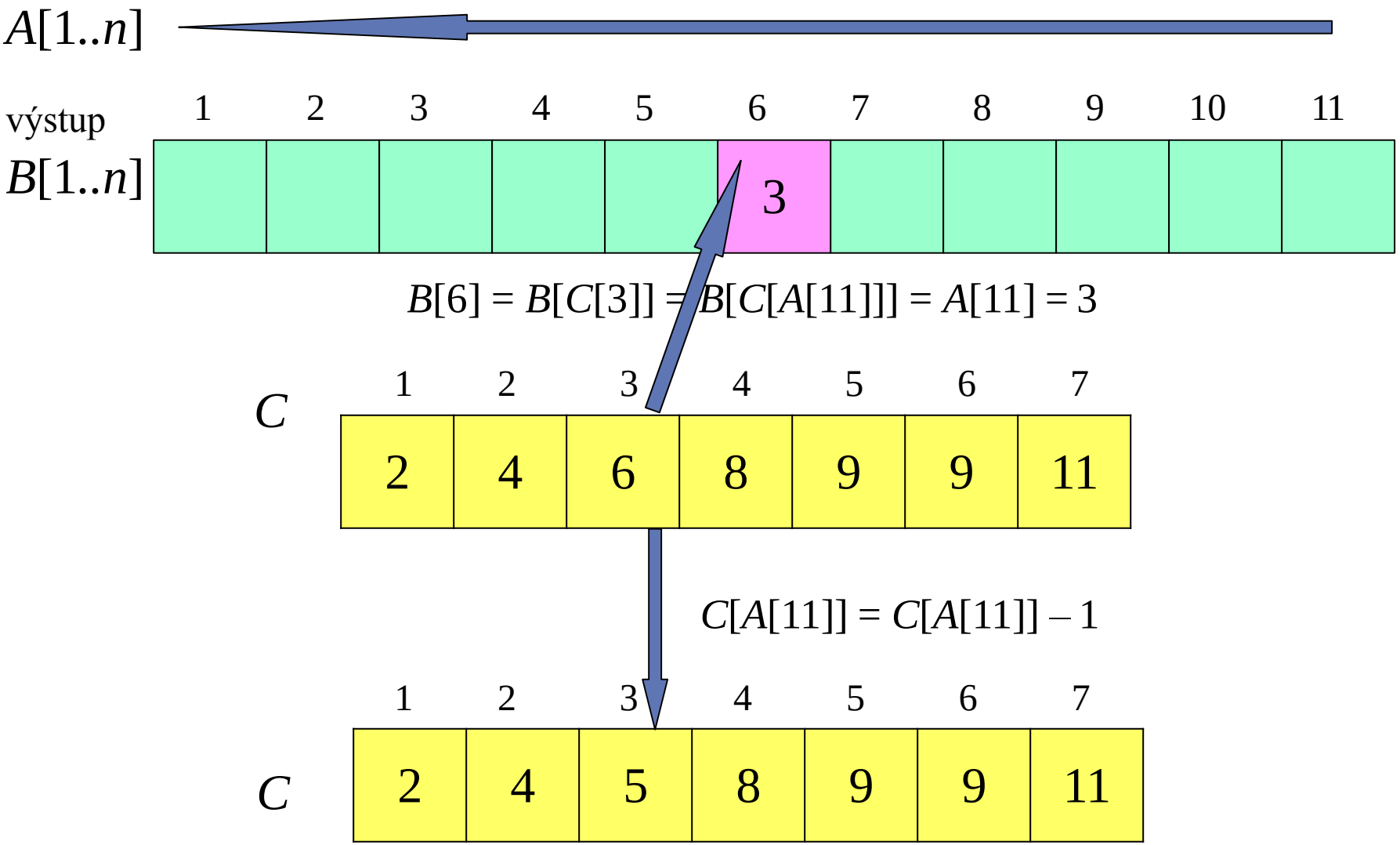
- Koľko operácií algoritmus vykoná?
  - **rádovo n+k+n**
- Koľko pomocnej pamäte potrebuje?
  - **pole veľkosti n a pole veľkosti k**
- Vhodný len pre malé k << n
- Pre veľký rozsah (int) je potrebné veľa pomocnej pamäte

```
// pocet vyskytov konkretnej hodnoty
for(i = 0; i < n; i++)
  c[a[i]]++;

// prefixove sucty: urcime index posledneho prvku s hodnotou j
for(j = 1; j < k; j++)
  c[j] = c[j] + c[j-1];

// prvky z pola a vložíme na prislusny index v poli b
for(i = n-1; i >= 0; i--)
  b[--c[a[i]]] = a[i];
```

Counting sort - príklad



Counting sort - príklad



Stabilný algoritmus

- Algoritmus usporadúvania je stabilný, ak vždy zachová pôvodné poradie prvkov s rovnakými kľúčmi

- Ak prvky s rovnakými kľúčmi sú neodlíšiteľné, tak nie je potrebné sa zaoberať stabilitou algoritmu
  (napr. ak kľúčom je samotný prvok)

- Zachovať pôvodné poradie prvkov je dôležité napr. pri viacnásobnom usporiadaní – najprv podľa priezviska a potom podľa mena.

Stabilný algoritmus (2)

- Každý nestabilný algoritmus sa dá implementovať ako stabilný tým, že sa zapamätá pôvodné poradie prvkov a pri zhodných kľúčoch sa berie do úvahy toto poradie

- Viacnásobné usporiadanie je možné obísť vytvorením jedného kľúča usporiadania, ktorý je zložený z primárneho, sekundárneho, atď.
  - Takéto úpravy nestabilných algoritmov majú negatívny vplyv na výpočtovú zložitosť

Stabilný algoritmus (3)

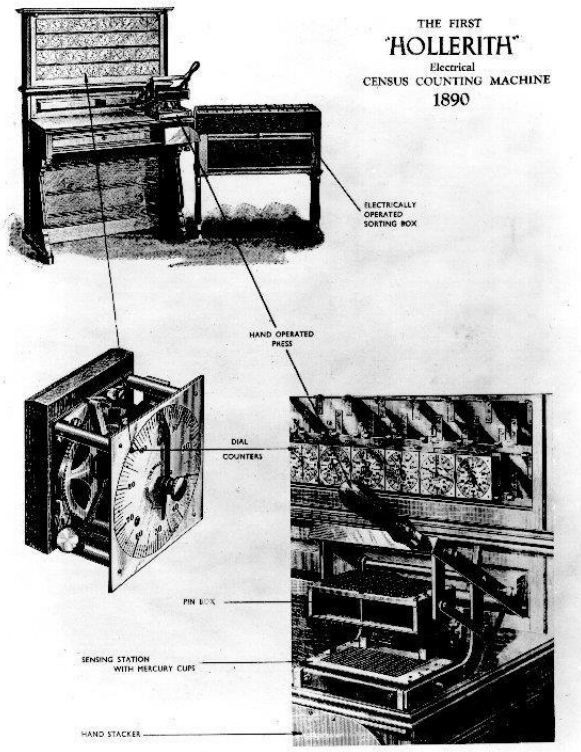
- Príklad – dvojice (kľúč, prvok):
  (4, 5) (2, 7) (2, 3) (5, 6)
- Dve možné usporiadania:
  (2, 7) (2, 3) (4, 5) (5, 6) – zachované poradie prvkov s kľúčmi 2 – stabilné usporiadanie
  (2, 3) (2, 7) (4, 5) (5, 6) – zmenené poradie prvkov s kľúčmi 2 – nestabilné usporiadanie

- Príklad na viacnásobné usporiadanie – dvojice (kľúč 1, kľúč 2):
  (4, 5) (2, 7) (2, 3) (4, 6)
- Usporiadanie najprv podľa kľúča 2, potom podľa kľúča 1:
  (2, **3**) (4, **5**) (4, **6**) (2, **7**) – podľa kľúča 2
  (**2**, 3) (**2**, 7) (**4**, 5) (**4**, 6) – podľa kľúča 1
- Usporiadanie najprv podľa kľúča 1, potom podľa kľúča 2:
  (**2**, 7) (**2**, 3) (**4**, 5) (**4**, 6) – podľa kľúča 1
  (2, **3**) (4, **5**) (4, **6**) (2, **7**) – podľa kľúča 2 – narušené poradie

- Pre zachovanie stability viacnásobného usporadúvania je potrebné usporadúvať postupne podľa kľúčov so zvyšujúcou sa prioritou

Radixové usporadúvanie

- Spracovanie sčítania ľudu USA 1880 trvalo skoro 10 rokov (robí sa každých 10 rokov)
- Herman Hollerith (1860-1929)
- Ako prednášateľ na MIT navrhol prototyp strojov na spracovanie diernych štítkov, doba spracovania ďalšieho sčítania ľudu v 1890 sa tým skrátila na 6 týždňov
- Základná myšlienka:
  začni triediť podľa najnižšieho rádu
- Založil firmu Tabulating Machine Company (1911), ktorá sa spojila s ďalšími firmami v 1924 - vznikla IBM (International Business Machines)



Radixové usporadúvanie – schéma

```
RADIX-SORT(A, d)
    for i ← 1 to d
        do stabilné usporadúvanie(A) podľa i-tej číslice
```

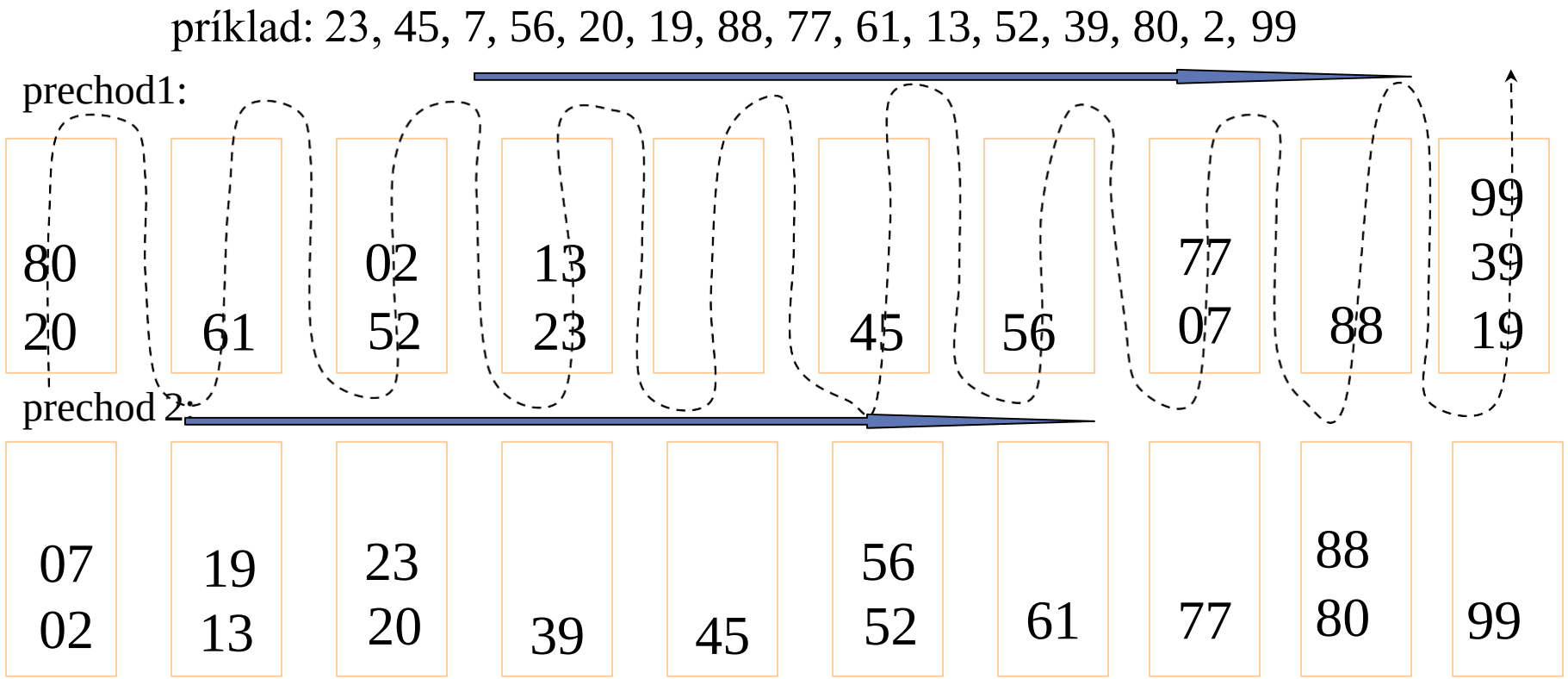
- Radixové usporadúvanie **neporovnáva dva celé kľúče**, ale spracúva a **porovnáva len časti kľúčov**
- Kľúče považuje za čísla zapísané v číselnej sústave so základom *k (radix, koreň)*, pracuje s jednotlivými číslicami:

$$\text{hodnota} = x_{d-1}k^{d-1} + x_{d-2}k^{d-2} + \ldots + x_2k^2 + x_1k^1 + x_0k^0$$

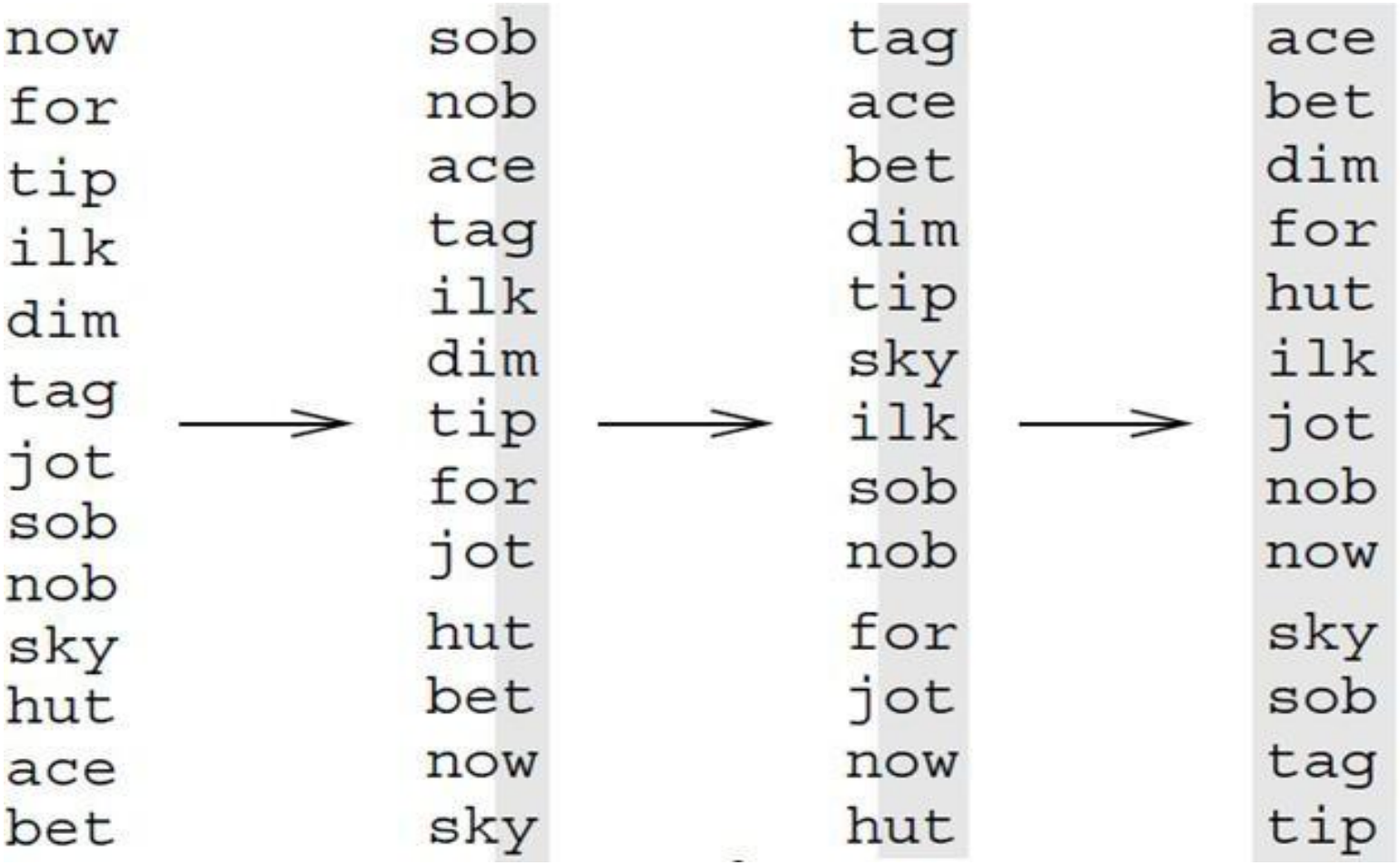
- Dokáže usporadúvať čísla, znakové reťazce, dáta, ...
  (počítače reprezentujú všetky údaje ako postupnosti 1 a 0 – binárna sústava => 2 je základ)
  - Uvažujme problém: usporiadať milión 64-bitových čísiel
  - Prvé riešenie: 64 prechodov cez milión čísiel?
  - Lepšie riešenie: interpretovať ich ako čísla v sústave so základom (radixom) 2^{16}, budú to najviac 4-miestne čísla ... vtedy to algoritmus usporiada len v 4 prechodoch!

Radixové usporadúvanie - príklad

- usporiada množinu čísiel vo viacerých prechodoch, začínajúc od číslic najnižšieho (jednotkového) rádu, potom usporiada podľa číslic najbližšieho vyššieho (desiatkového) rádu atď.



Radixové usporadúvanie – príklad



Radixové usporiadanie

- LSD Radix sort (least significant digit) – usporadúvanie podľa číslic postupuje od poslednej číslice (s najmenšou váhou) k prvej číslici (s najväčšou váhou)– stabilný.
- MSD Radix sort – od prvej číslice k poslednej – lexikografické usporiadanie – nestabilný
- Je dôležité na samotné usporadúvanie podľa jednotlivých číslic použiť nejaký stabilný algoritmus, aby sa nemenilo poradie prvkov s rovnakými číslicami jednej váhy pri usporadúvaní podľa inej váhy.
- Keďže počet možných číslic (ak k=10) je len 10, tak na usporiadanie podľa nich je výhodné použiť usporadúvanie spočítavaním.

Vedierkové usporadúvanie (Bucket sort)

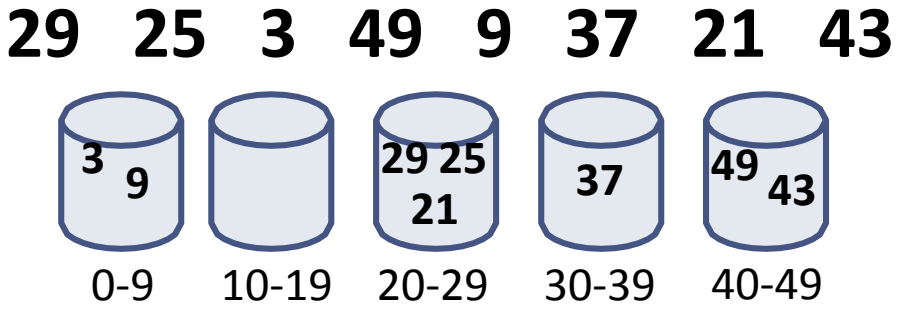
- Predpokladá, že vstup je akoby generovaný náhodným procesom, ktorý prvky distribuuje rovnomerne na celom intervale

- Rozdelí interval na *n* rovnako veľkých disjunktných podintervalov (vedierok – bucketov) a potom do nich rozmiestni vstupné čísla

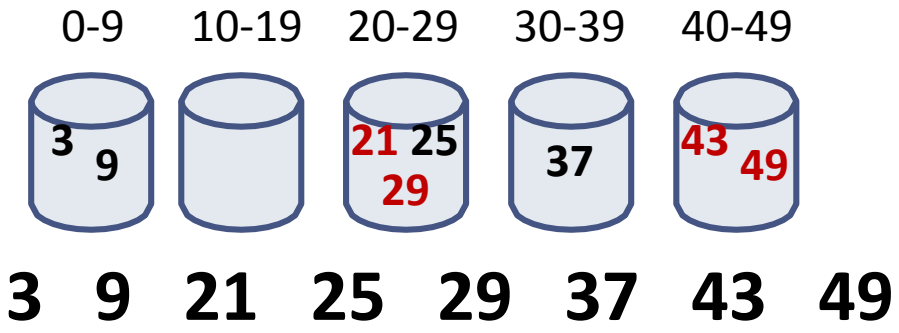
- Osobitne v každom vedierku sa potom tieto čísla usporiadajú

Vedierkové usporadúvanie - príklad

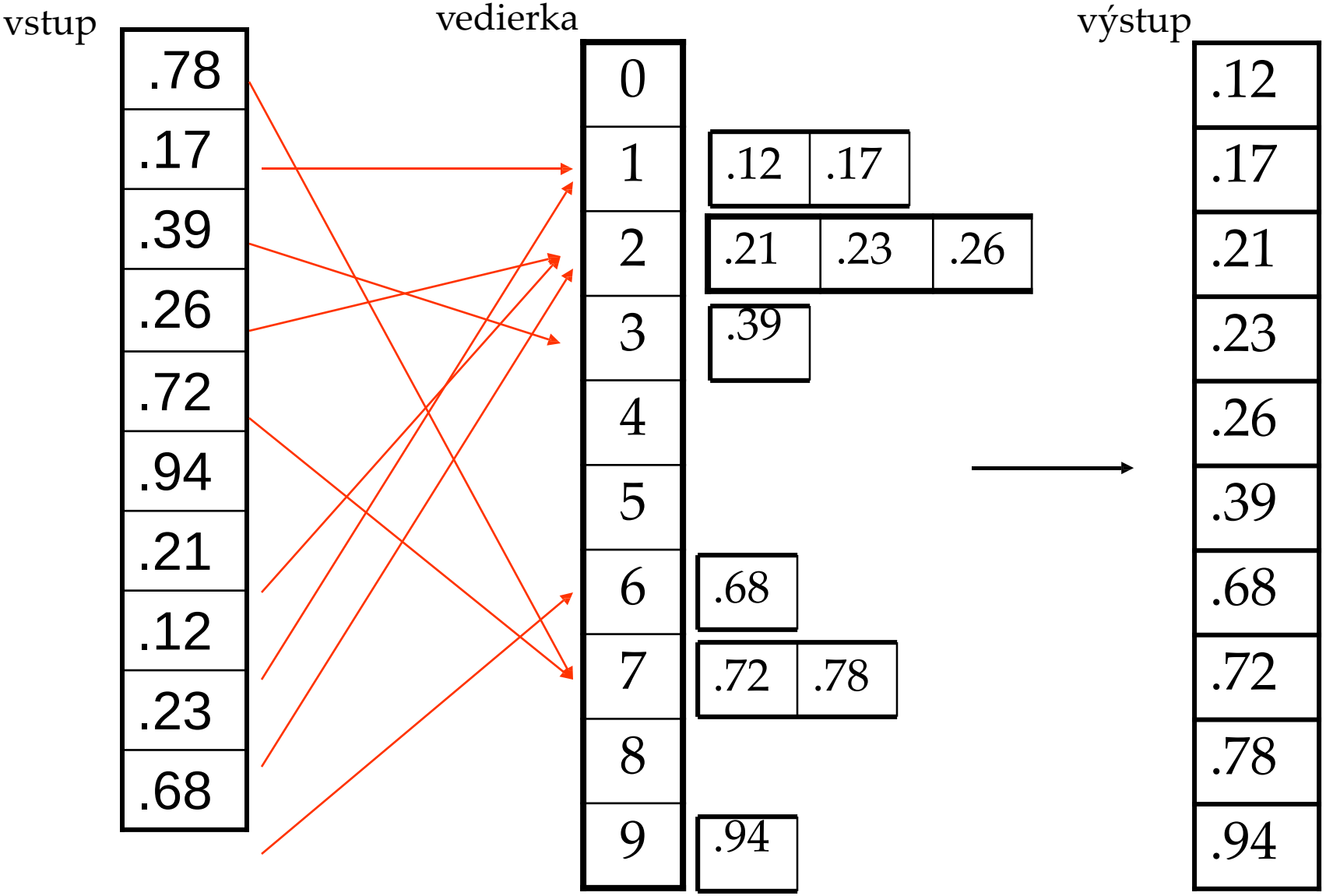
- Vytvoria sa prázdne vedierka veľkosti M/n (M - maximálna hodnota vstupného poľa, n - počet prvkov vstupného poľa)
- Rozptýlenie - prechádzanie vstupným poľom a rozmiestnenie každého prvku do prislúchajúceho vedierka



- Usporiadanie naplnených vedierok
- Zreťazenie vedierok - postupné prechádzanie usporiadaných vedierok a presúvanie prvkov späť do vstupného poľa



vedierko i obsahuje hodnoty z polouzavretého intervalu [i/10, (i + 1)/10).



Analýza zložitosti (Bucket sort)

- Jednotlivé vedierka väčšinou predstavujú spájaný zoznam, do ktorého sa na správne miesto presúvajú prvky zo vstupného poľa (insert sort)

- Činnosti ako vytvorenie vedierok, určenie prislúchajúceho vedierka, presunutie prvku do vedierka a zreťazenie vedierok do výslednej postupnosti trvajú $\mathbf{O(n)}$

- Výpočtová zložitosť usporiadania prvkov vo vedierkach Insert sortom $\mathbf{O(n^2)}$

Analýza zložitosti (Bucket sort)

- Výsledná časová zložitosť závisí od rozloženia prvkov vo vedierkach. Ak sú prvky rozmiestnené nerovnomerne a v niektorých vedierkach ich je veľmi veľa, tak časová zložitosť Insert sortu $O(n^2)$ prevažuje nad lineárnou zložitosťou a predstavuje výslednú zložitosť celého usporadúvania
- Takýto stav sa môže vyskytnúť ak rozsah prvkov m je oveľa väčší ako ich počet
- Preto sa niekedy celková zložitosť značí podobne ako pri Counting sorte $O(n+m)$. Ak $m=O(n)$, tak výsledná časová zložitosť je $O(n)$
- Ak sa počet vedierok rovná počtu vstupných prvkov, tak v priemere to vychádza na jeden prvok v každom vedierku, a preto sa za priemernú zložitosť berie $O(n)$

Ďakujem za pozornosť