ZALG - 10. cvičení

Shellovo třídění (Shell sort)

- Zakládní myšlenkou je výměna prvků na velké vzdálenosti
- Pole přeuspořádáváme s nějakým krokem. Pokud vezmeme každý h-tý prvek, dostaneme částečně setříděné pole
- Pole setříděné s krokem h představuje h proložených nezávislých polích
- Při Shellově třídění tedy setřídíme pole prvně s krokem h_1, pak s $h_2 < h_1$ až nakonec s krokem 1
- Počet kroků můžeme volit jako $[\log_2 n]-1$ a největší krok jako $2^{\log 2(n)+1}/2$

Shellovo třídění (Shell sort)

Posloupnosti:

Shell's original sequence	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$
Knuth's increments	$\frac{3^k - 1}{2}$	$1, 4, 13, 40, 121, \ldots$
Hibbard's increments	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$
Papernov & Stasevich increments	$2^k + 1$	$1, 3, 5, 9, 17, 33, 65, \ldots$
Sedgwick's icrements	$4^k + 3 \cdot 2^{k-1} + 1$	$1, 8, 23, 77, 281, \ldots$

Sotisfikovanější třídící algoritmy

- Algoritmy náročnější na implementaci
- Typická asymptotická složitost **O(nlog(n))**
- Typická cena za nízkou asymptotickou složitost je vyšší prostorová složitost typicky využívají pomocnou paměť **O(n)**
- Typičtí představitelé:
    - Tree sort
    - Merge sort
    - Heap sort
    - Quick sort

Třídění binárním stromem - Tree sort

1. Z hodnot v posloupnosti sestavíme binární vyhledávací strom
2. Hodnoty ve stromu projdeme INORDER a uložíme do pole (Nebo iterátorem máme-li ho implementovaný)

- Časová složitost:
  - Postav strom - **O(nlogn)** - průměrný případ (nejhorší případ **O(n^2)**)
  - Projdi strom INORDER - **O(n)**
  - Celkem: Průměr - **O(nlogn)**   Nejhorší případ **O(n^2)**
- Prostorová složitost - Binární strom - **S(n)**

Třídění přímým slučováním - Merge sort

- Algoritmus založený na metodě Rozděl a panuj:

1. **ROZDĚL (SPLIT):** Rozdělíme pole na dvě poloviny
2. **VYŘEŠ:** Obě poloviny setřídíme (rozdělujeme do té doby, dokud není pole jednoprvkové) (REKURZE)
3. **SPOJ (MERGE):** Setříděná pole slijeme do jednoho, potřebujeme pomocné pole

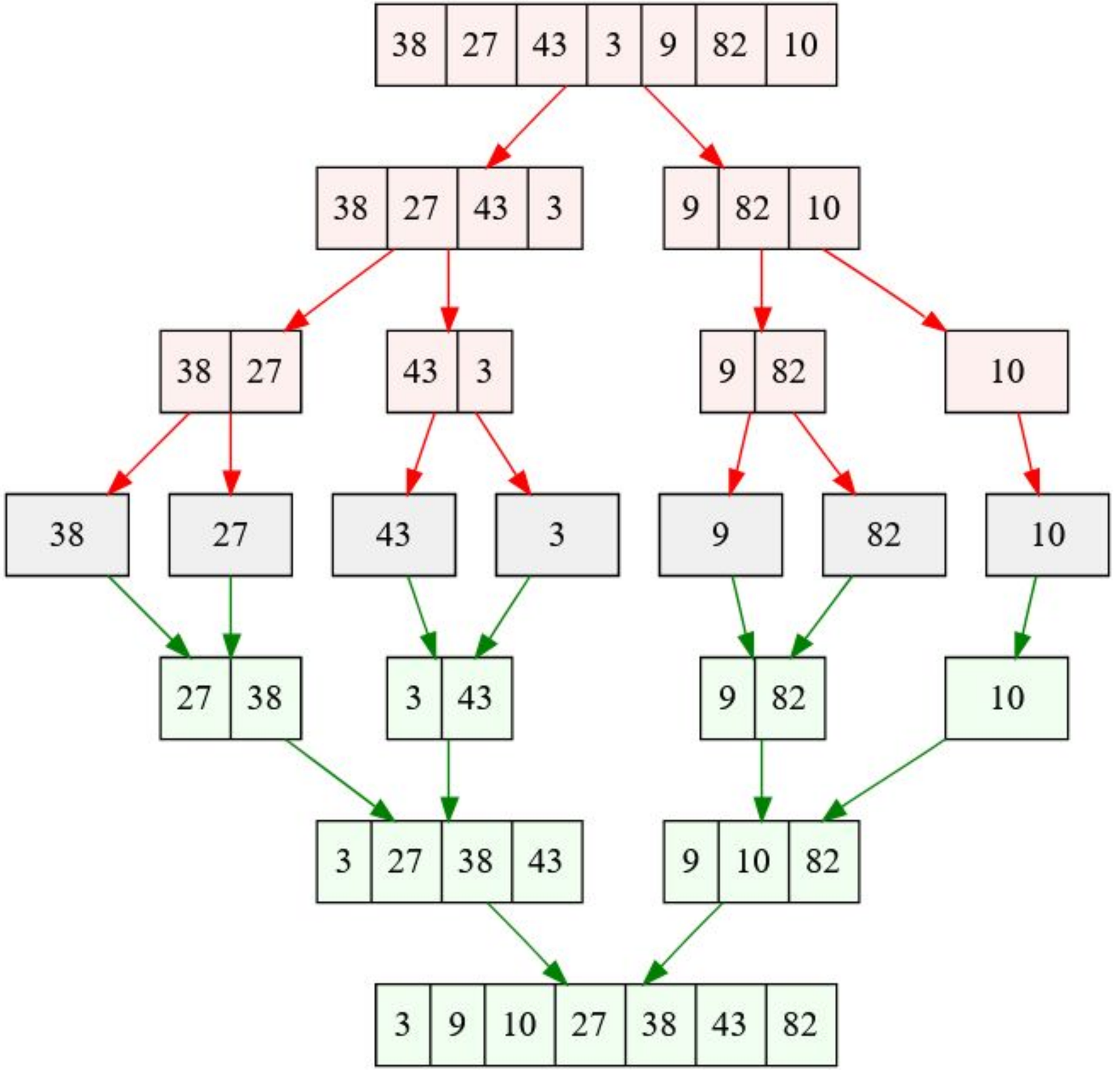
- SPLIT:
    - máme pole na intervalu [l, r] a rozdělíme ho na dva úseky → $[l, \frac{l+r}{2}]$ a $[\frac{l+r}{2} + 1, r]$

Třídění přímým slučováním - Merge sort

MERGE:

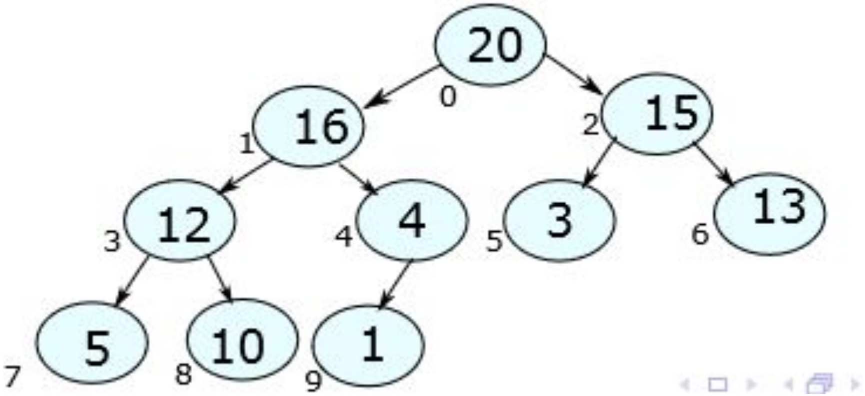
- Máme dvě setříděné posloupnosti na intervalech $[l_1, r_1]$ a $[l_2, r_2]$, kde $(r_1 = l_2 - 1)$
- $i := l_1$, $j := l_2$, $k := 0$
- Porovnáme hodnoty na indexech i a j a menší z nich zkopírujeme do pomocného pole na index k a (index i resp. j zvětším o 1) a opakuj
- Pokud dojdeme na konec nějakého úseku ($i == r_1$ resp. $j == r_2$) zkopírujeme zbytek druhého úseku do pomocného pole
- Pomocné pole překopírujeme zpět do pole na indexy $[l_1, r_2]$

➔ Časová složitost metod:
  - **SPLIT O(1)** **MERGE O(n)**
  - **Celkem: T(n) = 2T(n/2) + O(n)** → KUCHAŘKA (Master Theorem) **O(nlogn)**


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Třídění haldou (Heap sort)

- Založeno na použití binární haldy (binární strom) uložené v poli
- Budeme vycházet z toho, že prvek uložený v kořeni má největší hodnotu (rodič má větší hodnotu než oba jeho potomci) - Max Heap
- V C++ je kořen na indexu 0, a proto vrchol s indexem i má následovníky na indexech 2i+1 a 2i+2



Třídění haldou (Heap sort)

- Základní myšlenka:

1. Z posloupnosti postavíme MAX HEAP
2. Z haldy postupně odebíráme kořen a vkládáme jej na konec pole a prvek z konce pole opět vkládáme do haldy

Třídění haldou (Heap sort)

- Halda je posloupnost prvků h_l , h_{l+1} … h_r, $0 \leq l \leq r$ a pro všechny indexy i = l,...,r/2 platí $h_i \geq h_{2i+1}$ a $h_i \geq h_{2i+2}$
- Pokud pro l ≥ [n/2] vezmeme data A_l, A_{l+1} … A_{n-1} tak už tvoří haldu, neboť pro žádný index i neexistuje prvek na indexu 2i+1. Konkrétně tvoří listové patro binárního stromu
- Pokud k haldě budeme přidávat prvek A_{l-1} , tak tento prvek bude na místě kořene (ne nutně celého stromu, ale nějakého podstromu) a musíme tedy pro něj zkontrolovat podmínky, zda má větší hodnotu než jeho oba potomci. Pokud ne, tak ho prohodíme s větším z jeho potomků. Nově opět zkontrolujeme podmínky, zda má hodnotu větší než jeho následovníci a případně je prohodíme. To budeme opakovat do té doby, než dojdeme do listového patra.

i = 0 → heapify(arr, 6, 0)