Třídění I.

Základy algoritmizace – 9. cvičení

Zuzana Petříčková

14. dubna 2020

Třídění

Úloha

- chceme setřídit (tj. uspořádat podle nějakého klíče) data uložená v poli / ve spojovém seznamu / v souboru
  - data chceme podle klíče uspořádat vzestupně (nebo sestupně)

Typy třídění

- **vnitřní** - všechna data jsou uložena ve vnitřní paměti počítače
- **vnější** - data jsou uložena na disku (např. se celá do paměti nevejdou)

My se na tomto cvičení budeme zabývat pouze vnitřním tříděním

- budeme třídit čísla uložená v poli či ve spojovém seznamu

Třídění

Samostatná práce: nastudujte si skripta:

- kapitola 5: Třídění
  - dnešní cvičení se týká hlavně kapitoly 5.1: Vnitřní třídění, 5.3.1 Třídění přímým slučováním, popř. i 5.4 Porovnávací metody třídění

(ke zkoušce doporučuji nastudovat celou kapitolu 5)

Třídění

- existuje celá řada třídících algoritmů, můžeme je rozdělit do tří kategorií:
  - přímé metody
    - krátké a jednoduché algoritmy (obvykle založené na porovnávání)
    - třídí tzv. ,,na místě" (přímo v poli, používají jen konstantně velkou pomocnou paměť)
    - typicky časová složitost $O(n^2)$
    - kdy se hodí? ... když potřebujeme setřídit ,,málo" dat
  - sofistikovanější třídící algoritmy
  - třídící algoritmy speciální, ,,ušité na míru" konkrétním datům

Třídění

- existuje celá řada třídících algoritmů, můžeme je rozdělit do tří kategorií:
  - přímé metody
  - sofistikovanější třídící algoritmy
    - algoritmy náročnější na implementaci
    - typicky časová složitost $O(nlog_2 n)$
    - často využívají pomocnou paměť velikosti $O(n)$
    - kdy se hodí? ... když chceme setřídit ,,hodně" dat
  - třídící algoritmy speciální
    - ,,ušité na míru" konkrétním datům
    - časová složitost může být ještě lepší ... např. $O(n)$ pro bucketSort
    - kdy se hodí? ... když chceme setřídit data s ,,pěknými" nebo jinak specifickými vlastnostmi
    - a další situace (lexikografické třídění, topologické třídění grafu,..)

Přímé metody třídění

- krátké a jednoduché algoritmy (obvykle založené na porovnávání)
- třídí tzv. ,,na místě" (přímo v poli, používají jen konstantně velkou pomocnou paměť)
- typicky časová složitost $O(n^2)$
- kdy se hodí? ... když potřebujeme setřídit ,,málo" dat
- typičtí představitelé:
  - **selection sort** = třídění přímým výběrem minima
  - **insertion sort** = třídění vkládáním
  - **bubble sort** = bublinkové třídění
  - a další (např. shell sort, třídění binárním vkládáním,...)

Selection sort (třídění přímým výběrem minima)

Základní myšlenka (pro pole čísel):
postupně vytváříme setříděné pole zleva doprava:

1. najdeme v poli nejmenší prvek a prohodíme ho s prvkem na indexu 0
2. najdeme ve zbytku pole (od indexu 1) nejmenší prvek a prohodíme ho s prvkem na indexu 1

   ...
3. najdeme ve zbytku pole (od indexu i) nejmenší prvek a prohodíme ho s prvkem na indexu i, $i = 2, \ldots, n - 1$

Varianty:

- **naivně:** prohazuji prvky vždy když narazím na menší
- **lépe:** pamatuji si index minima a prohodím prvky jen jednou

Selection sort (třídění přímým výběrem minima)

Časová složitost:

- podrobná analýza složitosti je ve skriptech, str. 127-128
- stručně:
  - počet porovnání:
    - $C = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ (vždy porovnávám s celým zbytkem pole)
  - počet přiřazení:
    - $M_{min} = 0$ (pro správně setříděné pole)
    - $M_{max} = 3(n-1)$ (pro opačně setříděné pole)
      pro naivní variantu $M_{max} = 3C = 3\frac{n(n-1)}{2}$ (pro opačně setříděné pole)
    - $M_\theta \sim nlog_2 n$ (v průměrném případě, viz skripta)
  - celkem $T(n) = O(n^2)$

Prostorová složitost: $S(n) = O(1)$ (pomocné proměnné)

Insertion sort (třídění vkládáním)

Základní myšlenka (pro pole čísel): postupně vytváříme setříděné pole zleva doprava:

1. pro $i = 1, ..., n - 1$:
   část pole až do indexu $(i - 1)$ je již setříděná. My vezmeme prvek na indexu i a zatřídíme ho doleva na správné místo:
   - **naivně:** prvek, který byl na indexu i, zabubláme na správné místo (postupným prohazováním prvek posouváme o jednu pozici doleva, dokud nedojde na správné místo)
   - **lépe:** prvek na indexu i uložíme do pomocné proměnné (do tzv. akumulátoru) a následně posouváme prvky nalevo od pozice i jeden po druhém o jednu pozici doprava, dokud nenarazíme na prvek menší než prvek v akumulátoru, nakonec vložím prvek z akumulátoru na správné (nyní uvolněné) místo

Insertion sort (třídění vkládáním)

Časová složitost:

- podrobná analýza složitosti je ve skriptech, str. 123-124
- stručně:
  - počet porovnání:
    - $C_{min} = n - 1$ (pro správně setříděné pole)
    - $C_{max} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ (pro opačně setříděné pole)
    - $C_\theta \sim \frac{n^2+n+1}{4}$ (v průměrném případě, viz skripta)
  - počet přiřazení:
    - $M_{min} = 0$ (pro správně setříděné pole)
    - $M_{max} = \sum_{i=1}^{n-1}(C_i + 2) = \sum_{i=1}^{n-1}(i + 2) = \frac{n^2+3n-4}{4}$ (pro opačně setříděné pole)

      pro naivní variantu $M_{max} = 3\sum_{i=1}^{n-1} C_i = 3\frac{n(n-1)}{2}$ (pro opačně setříděné pole)
    - $M_\theta \sim \frac{n^2+9n-10}{4}$ (v průměrném případě, viz skripta)
  - celkem $T(n) = O(n^2)$

Prostorová složitost: $S(n) = O(1)$ (pomocné proměnné)

Insertion sort (třídění vkládáním)

Další vylepšení:

- **shell sort** (Shellovo třídění), viz. skripta, str. 131-133, průměrná časová složitost až $O(n^{1.2})$

Spojový seznam ... selectionSort a insertionSort

```
void Seznam::selectionSort()
{
    Seznam s;
    while (neprazdny())
        s.vlozZacatek(vyjmiMaximum());
    pripoj(s);
}

void Seznam::insertionSort()
{
    Seznam s;
    while (neprazdny())
        s.zatridPrvek(vyjmiZacatek());
    pripoj(s);
}
// pripoj ... presune prvky seznamu s na konec aktualniho seznamu
```

Bubble sort (bublinkové třídění)

Základní naivní varianta (pro pole čísel):

- (n-1)-krát probubláme celé pole zleva doprava:
  - pro $i = 0, ..., n-2$ porovnáme prvky na indexech i a $i+1$, pokud je na indexu $(i+1)$ menší číslo než na indexu i, prvky prohodíme

Vylepšné varianty:

1. probublání neopakujeme $(n-1)$-krát, ale skončíme, pokud se v posledním průchodu žádné prvky neprohodily (pole již je setříděné, nemá smysl pokračovat)
2. neprobubláváme celým polem, ale jen do indexu i (od indexu $(i+1)$ dál již je pole setříděné)
3. neprobubláváme celým polem. V každém průchodu si zapamatujeme nejpravější (poslední) index k prvku, který se prohodil se svým pravým sousedem. V dalším průchodu stačí probublávat jen do indexu $k-1$ (od indexu k dál již je pole setříděné).

Bubble sort (bublinkové třídění)

Další vylepšení: třídění třesením (přetřásáním)

- další vylepšení poslední varianty bublinkového třídění z minulého snímku
- probubláváme polem střídavě zleva a zprava, průběžně aktualizujeme pravou a levou mez, odkud kam bublat
- viz. skripta, str. 130-131

Bubble sort (bublinkové třídění)

Časová složitost:

- podrobná analýza složitosti je ve skriptech, str. 131
- stručně:
  - počet porovnání (bude záviset na variantě algoritmu):
    - pro naivní variantu: $C = (n-1)^2$ (vždy)
    - pro první vylepšení: $C_{min} = n-1$ (pro správně setříděné pole), $C_{max} = (n-1)^2$ (pro opačně setříděné pole)
    - pro druhé vylepšení: $C = \frac{n(n-1)}{2}$ (vždy)
    - pro třetí vylepšení: $C_{min} = n-1$ (pro správně setříděné pole), $C_{max} = \frac{n(n-1)}{2}$ (pro opačně setříděné pole)
    - pro třídění přetřásáním: $C_{min} = n-1$, $C_{max} = \frac{n(n-1)}{2}$, $C_\theta \sim \frac{n^2 - n\log_2 n - kn}{2}$ (v průměrném případě, viz skripta)
  - počet přiřazení:
    - $M_{min} = 0$
    - $M_{max} = 3C_{max}$
  - celkem $T(n) = O(n^2)$

Prostorová složitost: $S(n) = O(1)$ (pomocné proměnné)

Spojový seznam ... bubbleSort

(třetí vylepšená varianta)

```
void Seznam::bubbleSort()
{
    Seznam s;
    Prvek *kam = zarazka;
    while (kam != hlava)
        kam = zabublej(kam);
}

/* zabublej: probubla seznamem az po Prvek kam,
             vrati misto posledni zmeny
*/
```

Sofistikovanější třídící algoritmy

- algoritmy náročnější na implementaci
- časová složitost $O(n log_2 n)$ v průměru nebo vždy
- typickou cenou za nižší časovou složitost je vyšší prostorová složitost, obvykle využívají pomocnou paměť velikosti $O(n)$
- kdy se hodí? ... když chceme setřídit ,,hodně" dat
- typičtí představitelé:
  - **třídění binárním vyhledávacím stromem**
  - **merge sort** = třídění přímým slučováním
  - **třídění haldou**
  - **quick sort** = rychlé třídění

Třídění binárním vyhledávacím stromem

Základní myšlenka (pro pole čísel):

1. z hodnot v poli postavíme binární vyhledávací strom (postupným vkládáním)
2. binární strom projdeme v pořadí INORDER a vložíme prvky zpět do pole

Časová složitost

1. postav strom:
   v nejlepším a průměrném případě: $T(n) = O(nlog_2 n)$,
   v nejhorším případě: $T(n) = O(n^2)$
2. projdi strom: $T(n) = O(n)$ (vždy)
3. celkem:
   - v nejlepším a průměrném případě: $T(n) = O(nlog_2 n)$,
   - v nejhorším případě: $T(n) = O(n^2)$

Prostorová složitost: $S(n) = O(n)$ (bin. vyhl. strom)

Merge sort (třídění přímým slučováním)

Základní myšlenka (pro pole čísel): metoda rozděl a panuj

1. **SPLIT:** rozdělíme pole na dvě poloviny
2. obě poloviny setřídíme (např. rekurzivně, popř. s využitím zásobníku)
3. **MERGE:** setříděná pole slijeme do jednoho
   potřebujeme pomocné pole pro uložení výsledného (slitého) pole

Implementace SPLIT

- máme úsek pole od indexu l do indexu r: $[l, r], l < r$
- rozdělíme ho na dva úseky $[l, \frac{l+r}{2}]$ a $[\frac{l+r}{2} + 1, r]$

Merge sort (třídění přímým slučováním)

Implementace MERGE

- máme dva setříděné úseky pole $[l_1, r_1]$ a $[l_2, r_2]$, kde $l_2 = r_1 + 1$
- nastavíme $i = l_1$, $j = l_2$, $k = 0$
- porovnáme prvky na indexech i a j, menší z obou hodnot zkopírujeme na index k v pomocném poli, zvýšíme o jedničku příslušný z indexů i a j a index k
- pokud dojdeme na konec jednoho z úseků, zbylé hodnoty z druhého úseku překopírujeme do pomocného pole
- překopírujeme hodnoty z pomocného pole zpět do původního pole na indexy $[l_1, ..., r_2]$

Časová složitost:

1. **SPLIT:** $T(n) = 0(1)$
2. **MERGE:** $T(n) = 0(n)$
3. celkem $T(n) = 2T(n/2) + O(n) = O(nlog_2 n)$ (na rozmyšlenou, použijte master theorem = kuchařku)

Prostorová složitost: $S(n) = O(n)$ (pomocné pole)

Merge sort (třídění přímým slučováním)

Implementace pomocí zásobníku

- na zásobník budeme vkládat prvky typu (split, l_1, r_1) a (merge, l_1, r_1, r_2)

```
if (n>0)
    vloz na zasobnik prvek (split,0,n-1)
while(zasobnik neni prazdny) {
    if (na vrchu zasobniku je split){
        vyjmi vrchol zasobniku (split,l,r)
        s = (l+r)/2
        vloz na zasobnik prvek (merge,l,s,r);
        if (l < s)
            vloz na zasobnik prvek (split,l,s)
        if (s+1 < r)
            vloz na zasobnik prvek (split,s+1,r)
    }
    else {
        vyjmi vrchol zasobniku (merge,l,s,r)
        slij prislusne useky pole
    }
}
```

Spojový seznam ... mergeSort (rekurzivně)

```
void Seznam::mergeSort()
{
    if (prvni == zarazka || prvni->dalsi == zarazka)
        return;
    Seznam t, u;
    split(t, u);
    t.mergeSort();
    u.mergeSort();
    merge(t,u);
}
```

Seznam ... mergeSort (zásobník) ... zjednodušeně, pseudokód

```
void Seznam::mergeSort() {
    Seznam *s, *t, *u;
    Zasobnik z; //prvek = ukazatel na seznam nebo trojice ukazatelu
    if (asponDvouprvkovy())
        z.vlozSplit(this);
    while(z.neprazdny()) {
        if (z.naVrchuSplit()){
            s = z.vyjmiSplit();
            s->split(t, u);
            z.vlozMerge(s,t,u);
            if (t.asponDvouprvkovy())
                z.vlozSplit(t);
            if (u.asponDvouprvkovy())
                z.vlozSplit(u);
        }
        else {
            z.vyjmiMerge(s,t,u);
            s->merge(t, u);
            delete t; delete u;
        }
```

Třídění

Další třídící algoritmy

- pokračování příště

Shrnutí: složitost různých třídících algoritmů

	MIN	AVERAGE	MAX	paměť (pro pole)
selectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
bubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
shellSort		$O(n^\alpha)$		$O(1)$
treeSort		$O(nlog_2 n)$	$O(n^2)$	$O(n)$
mergeSort		$O(nlog_2 n)$		$O(n)$
quickSort		$O(nlog_2 n)$	$O(n^2)$	$\Omega(log_2 n)...O(n)$
heapSort		$O(nlog_2 n)$		$O(1)$
bucketSort		$O(n+k)$		$O(n+k)$

- n je délka pole/seznamu
- k je počet přihrádek
- $1 < \alpha < 2$