Anotace

- Třídění (dokončení),
- Pointery, dynamické proměnné

Algoritmy třídění porovnáním – pokr.

- Pro účely dolního odhadu složitosti třídění porovnáním budeme počítat za "krok" algoritmu jen a jen porovnání dvou čísel.
- Ukážeme tedy, že jakýkoliv korektní třídicí algoritmus musí pro vstup délky n provést $\Omega(n \log n)$ porovnání.
- Pozorování: Algoritmus třídění porovnáním běží pevně (přesně) definovaným způsobem, do chvíle, kdy porovná dvě čísla (tam se může rozvětvit podle výsledku porovnání).

Rozhodovací strom

Definition

Rozhodovací strom algoritmu $\mathcal{A}$ je strom, jehož listy odpovídají jednotlivým možným výsledkům algoritmu $\mathcal{A}$. Strom je organizován tak, že výpočet algoritmu $\mathcal{A}$ začíná v kořeni a každé větvení směrem k listu odpovídá nějakému rozhodnutí programu.

- Algoritmy třídění porovnáním se smějí rozhodovat jen na základě porovnání.
- Větvení v rozhodovacím stromě tudíž budou odpovídat jednotlivým porovnáním.
- Uvážíme takový algoritmus, který místo setřídění oznámí, jak zapermutovat vstup, abychom získali setříděnou posloupnost (jde evidentně o ekvivalentní problémy).

Použití rozhodovacích stromů

- Složitost algoritmu v nejhorším případě odpovídá maximální hloubce listu v příslušném rozhodovacím stromě.
- Chceme tedy ukázat, že v rozhodovacím stromě jakéhokoliv třídícího algoritmu existuje list v hloubce $\Theta(n \log n)$.
- Použijeme standardní počítací argument, tedy ukážeme, že má-li mít strom dostatečný počet listů (na setřídění každé posloupnosti), musí mít hloubku alespoň $\frac{n}{2} \log n$.
- Listů je třeba aspoň $n!$, jinak najdeme dvě různé permutace (čísel 1...n), které algoritmus nechá zapermutovat stejně (a tedy aspoň jednu permutaci nesetřídí).
- Strom je binární (každý vrchol má nejvýš dva syny), tudíž hloubka stromu musí být aspoň $\log_2 n! \geq \log_2 n^{\frac{n}{2}} = \frac{n}{2} \log_2 n$.
- Odhad faktoriálu v Kapitolách z diskrétní matematiky.

Třídění porovnáním – poznámky

- Dolní odhad třídění porovnáním ukazuje složitost $\Theta(n \log n)$ pro algoritmy Quicksort a Heapsort (pro Mergesort také, ale tam jsme ji nahlédli snáze.
- Co když porovnání nevrací dvě hodnoty ale více?
- Větvíme vždy na konečně možností, neuděláme tedy rozhodovací strom binární, ale k-ární a logaritmus jen změní základ, tedy se změní multiplikativní konstanta.

Radix- alias bucketsort

- Třídíme-li celá (přirozená) čísla, jejichž velikost je omezena, můžeme využít toho, že v desítkovém zápisu je na každé pozici jen jedna číslice z deseti.
- Můžeme tedy čísla rozhodit na hromádky podle těchto číslic.
- Naivní algoritmus by zřejmě začal od začátku, vytřídil "nejmenší" a jel dále.
- To by sice fungovalo, ale bylo by to ošklivé.
- My začneme od poslední číslice.

Bucketsort

Pozor, kreativní pseudokód!

```
procedure bucketsort(pole);
begin
      for i:=0 to delka do
      begin
            rozhod cisla z pole do pole0 az pole9;
                  podle cislice na i-tem miste;
            pole:=pole0+pole1+pole2+...+pole9;
      end;
end;
```

Bucketsort – analýza

- Složitost: `delka` je konstanta, tedy konstantně-krát prolezeme pole délky n, tudíž složitost je $\Theta(n)$ (dolní odhad opět triviálně n, na každý prvek musíme "kouknout").
- Korektnost:
  - Pokud se dvě čísla liší, liší se na nějaké pozici.
  - Uvažme nejvyšší řád, na kterém se liší. Na tomto řádu má menší číslo menší číslici a tudíž:
  - při rozhazování podle tohoto řádu se menší číslo dostane před větší.
  - Dále jdou čísla do stejné přihrádky a tudíž se jejich pořadí nemění.
- Co je s naším dolním odhadem $\Omega(n \log n)$?
- Nemá splněné předpoklady, bucketsort čísla vůbec neporovnává.

Paměti

- Počítače mají několik typů pamětí:
- Operační (zpravidla RAM),
- persistentní (disky, diskety, magnetofonové pásky, děrné štítky).
- Prvním občas říkáme vnitřní, druhým vnější.
- Vnitřní paměť je rychlá, kdežto vnější paměť je velká.
- Ve vnitřní paměti můžeme "dovádět", ve vnější paměti hledání trvá déle (je vhodné načíst údaje za sebou, ne hledat po celé paměti).
- Naše třídicí algoritmy jsou určitě vhodné pro vnitřní paměť (obzvlášť heapsort, který nepotřebuje paměť navíc, ale s haldou nevybíravě otřásá).
- Pro vnější paměti je vhodný mergesort.

Vnější třídění

- Načti co největší blok do operační paměti, sestav haldu a při extract-min místo zakořenění posledního zkus načíst další prvek.
- Je-li tento prvek aspoň tak velký jako poslední prvek, co jsme poslali na výstup, zabublej ho.
- Je-li tento prvek menší než poslední prvek, co jsme poslali na výstup, nepřidávej ho do haldy, dokonči nad haldou heapsort a založ novou haldu.
- Takto získáme v mezích možností dlouhé setříděné bloky, na které můžeme pustit `mergesort`.

Vnější třídění s více páskami

- Dnes pro praktické použití už asi passé, ale:
- Stává se, že máme k dispozici pásek více (dnes třeba disků).
- Pak je výhodou mergovat ze všech ostatních na jednu.
- Na každé pásce ovšem máme několik bloků.
- Dojde-li obsah jedné pásky, doběhneme současné mergované "bloky" a začneme mergovat na uvolněnou pásku.
- Jak zorganizovat počty bloků na jednotlivých páskách?
- Zobecněnými Fibonacciho čísly.

Organizace počtu bloků pro mergesort

- Předpokládejme, že máme tři pásky.
- Chceme, aby slévání skončilo jedním dlouhým blokem na jedné pásce":
  (0,0,1)
- Tudíž předtím na ostatních dvou páskách muselo být po jednom bloku:
  (1,1,0)
- Bezprostředně předtím jsme mergovali ze třetí pásky (protože se vyprázdnila) a museli jsme mergovat někam (BÚNO na druhou). Tedy předtím vypadalo rozmístění bloků takto:
  (2,0,1)

- Předtím tudíž výpočet vypadal takto:
  (0,2,3)
- Předtím zase takto:
  (3,5,0)...
- Tedy vždy jde o dvě po sobě jdoucí Fibonacciho čísla (a třetí je nula).
- Vychází totiž rekurence $a_n = a_{n-1} + a_{n-2}$, obecně $a_n = a_{n-1} + ... + a_{n-k}$.

K čemu jsou dynamické proměnné?

- K mnoha algoritmům bychom potřebovali pole proměnlivé délky
- nebo aspoň jinou datovou strukturu proměnlivé délky.
- Jak implementovat frontu a zásobník?
- Použijeme pointery.

Pointery alias ukazatele

- Paměť je organizována lineárně v podobě jednotlivých adres.
- Na těchto adresách jsou ukládány hodnoty (kód, data).
- Paměť zpravidla adresujeme přirozenými čísly, která zapisujeme v šestnáctkové soustavě.
- Na data v paměti si tedy můžeme ukazovat.
- Pod těmito ukazateli můžeme mít údaje libovolných typů, z čehož vyplývá jisté nebezpečí.
- Z toho vzplývá jisté nebezpečí a nutnost být obezřetný.

Pointery – syntax a sémantika

- S pointery pracujeme zpravidla tak, že definujeme datový typ *ukazatel na něco.*
- *Ukazatel na* řekneme pomocí operátoru stříšky:
- `type pint=^integer; {ukazatel na integer}`
- Následně definujeme proměnnou příslušného typu:
  `var ukaz:pint;`
- Pod pointer "koukneme"opět pomocí operátoru stříšky:
  `writeln(ukaz^);`
- Jenže ono to zdaleka není tak snadné!

Organizace paměti s ohledem na program

- Program sestává z kódu, tzv. statických dat, prostoru zásobníku a oblasti haldy.
- Kam ukazuje pointer, který si lehkomyslně vyrobíme?
- Pokud s ním budeme zacházet rozumně, bude ukazovat někam do oblasti haldy, k tomu ale máme ještě daleko.
- Kam tedy pointer ukazuje?
- V lepším případě na adresu 0, v horším na náhodně vybraný kus paměti.
- Pořádek v paměti hlídá allokátor, který ví, které části paměti jsou použité a které ne a může nám přidělit prostor.

Allokátor a jeho použití

- Abychom mohli pod pointer kouknout, musíme ho někam nasměrovat. A to buďto na už existující pointer (`var a,b:pint;... naalokuj b;... a:=b;`),
- anebo "urafnutím" nové paměti: `new(a);`
- Funkci `new` předáváme jako parametr proměnnou typu ukazatel.
- Funkce `new` najde v paměti vhodné místo a nasměruje na něj příslušný pointer.
- Cvičení zimního učiva: Bere `new` parametr hodnotou nebo referencí?
- Od chvíle, kdy máme naalokováno, můžeme pod pointer koukat i zapisovat:
- `new(a); a^:=5; writeln(a^);` Ale pozor na přesměrování ukazatele!

Pointery a práce s nimi

- `var a,b:pint;`
- `new(a);` – naallokuje místo pro proměnnou typu integer
- `a^:=5;` – zapíše pod pointer hodnotu 5.
- `b^:=a^;` – okopíruje pod pointer `b` hodnotu, na kterou ukazuje pointer `a`.
- `b:=a;` – okopíruje pointer (tedy `a` a `b` ukazují na stejné místo).
- Co v kontextu uvedeného kódu udělá `b^:=10; writeln(a^);`?
- Pozor, více pointery si můžeme ukazovat na to samé místo!