Programování II

Martin Pergel, perm@kam.mff.cuni.cz

26. února 2010

Anotace

- Informace
- Třídění
- Dynamické programování

- Přednáška a cvičení jedou dál (jako v zimě),
- hlavní témata:
  - pointery,
  - objekty,
  - algoritmy
- Zkouška:
  - Test na práci s pointery,
  - napsání "programu" [oboje na papír],
  - rozprava (o napsaném),
  - teoretické otázky.

Třídění

- Bylo v zimě, konkrétně jsme si ukazovali algoritmy:
- bubblesort, shakesort, zatřiďování (insert-sort), přímý výběr (select-sort) – důležité je znát algoritmy, není nutné pamatovat si přesné přiřazení algoritmů k názvům. Složitost těchto algoritmů je $\Theta(n^2)$.
- Dále byl Quicksort s analýzou složitosti a vysvětlením různých voleb pivota (a dopady této volby na složitost).
- To ovšem zdaleka není vše, dnes budou:
- Heapsort (alias třídění haldou),
- Mergesort (vulgo třídění sléváním),
- Analýza složitosti problému třídění porovnáním,
- Algoritmy nestojící na porovnání,
- Algoritmy "vnějšího" třídění.

Halda

Co je halda?

Definition

Halda je datová struktura osazená operacemi `postav_haldu` a `extract_min`, kde prvky jsou organizovány do stromu takového, že hodnota obsažená v rodiči je nejvýš taková, jako hodnota v synu.

Pozorování: Minimum je v koření haldy.

Halda pokr.

Většinou pracujeme s binárními haldami a jestě k tomu tzv. *leftest*:

- Binární halda je taková halda, že každý prvek má nejvýše dva potomky, navíc vzdálenosti z kořene do listů se liší (pro jednotlivé listy) nejvýše o jedna.
- V souvislosti s binární haldou mluvíme pro každý prvek o levém resp. pravém synu. Haldu si typicky představujeme nakreslenu s ohledem na tuto levo-pravou "orientaci".
- Binární halda je *leftest*, jestliže existuje nejvýše jeden prvek x, který má právě jednoho syna a to levého. Nalevo od x jsou všechny cesty z kořene do listů o jedna delší než napravo od x.
- Technicky v leftest haldě přidáme prvku x pravého syna s hodnotou ∞.

Halda a její reprezentace

- Binární leftest haldu lze snadno reprezentovat v poli.
- Potomky prvku na pozici i jsou na pozicích $2i$ a $2i + 1$.
- Rodič prvku je na pozici $\lfloor \frac{i}{2} \rfloor$,
- halda je tedy velmi elegantní struktura, kterou se snadno prochází.
- Třídění haldou vypadá tak, že postavíme haldu a n-krát provedeme `extract_min`.
- Zbývá drobnost, impelementovat `postav_haldu` a `extract_min`.

Jak postavit haldu

Pozor, kreativní pseudokód!

```
function zabublej(i:integer);
begin jeste:=true;
      while (2*i<=n) and jeste do {pozor na 2i + 1}
            if (halda[2*i]<halda[i]) or
                  (halda[2*i+1]<hodnota[i]) then
            begin if(halda[2*i+1]<halda[2*i]) then
                        j:=2*i else j:=2*i+1;
                  pom:=halda[i];
                  halda[i]:=halda[j];
                  halda[j]:=pom;{prohod obsahy}
                  i:=j;
            end
            else  jeste:=false;
end;
```

Jak postavit haldu

- Haldu budeme stavět zdola nahoru!
- Do haldy ulož příslušné prvky a udělej:
- ```
  for i:=n div 2 downto 1 do
      zabublej(i);
  ```
- volání `zabublej` rozšíří vlastnost haldy (tedy nerovnosti hodnot) i na prvek i.

Důkaz.

Indukcí: Na začátku platí na dolní úrovni ($n/2$ až n).
Pokud platí od $i+1$ do n, může dojít k chybě jen na pozici i.
Zabublání nahradí současný prvek tím menším ze synů, chyba "propadá" dolů po jedné cestě (tedy je vždy na jednom místě).
Tato chyba buďto zanikne samovolně (test ve funkci `zabublej`), nebo zanikne doputováním do listu. □
  ```

Analýza složitosti

- Voláme funkci `zabublej`, která vždy v nejhorším případě propadne do listu.
- Jaká je vzdálenost do listu?
- Pro první polovinu (procházených) prvků 1, další čtvrtinu 2, pro další osminu 3..., tedy celkem přehazujeme nejvýše:
- $\sum_{i=1}^{\log n} i\frac{n}{2^i} = n\sum_{i=1}^{\log n}\frac{i}{2^i} = n\sum_{i=1}^{\log n}\sum_{j=i}^{\log n}\frac{1}{2^i} = n\sum_{i=1}^{\log n}\frac{1}{2^i}\frac{1-2^{i+1}}{1-\frac{1}{2}} \leq n\sum_{i=1}^{\log n}\frac{1}{2^i}2 = kn$krát.
- Sčítání využívá jen součet geometrické řady a triviální odhady!
- Problém sestavení haldy má tedy složitost $\Theta(n)$ (dolní odhad je triviální, na každý prvek musíme aspoň jednou sáhnout).

Funkce extract_min

```
function extract_min:integer;
begin extract_min:=halda[1];
      halda[1]:=halda[n];
      n:=n-1;
      zabublej(1);
end;
```

- Složitost je $O(\log n)$,
- důkaz korektnosti je jedna iterace indukčního kroku korektnosti stavění haldy, tedy:
- Porucha může nastat jedině v kořeni. Funkce `zabublej` sune poruchu dolů (udržuje ji na jednom místě) a nejpozději v listu porucha zanikne.

Algoritmus heapsort

Analýza složitosti

- Napřed postavíme haldu s lineární složitostí (R. Tarjan),
- pak n-krát odebereme minimum se složitostí $O(\log n)$,
- celkem tedy je složitost heapsortu $O(n \log n)$.
- Výhoda: Heapsort je rychlý, nepoužívá místo navíc, nedělá si zbytečné poznámky (co ještě setřídil a co už ne), vystačí jen s údajem o velikosti haldy.

Mergesort – třídění sléváním

- Máme-li dvě setříděné posloupnosti, k jejich setřídění stačí tyto "slít".
- Posloupnost délky 1 je vždy setříděná.
- Slitím dvou (setříděných) posloupností délky k vznikne setříděná posloupnost délky $2k$.
- Posloupnost délky n je setříděna, pokud koukáme na její setříděný "začátek" délky n.

Mergesort funkce merge

Pozor, kreativní pseudokód!

```
procedure merge(a,b:posl; var kam:posl);
begin i:=1;j:=1;k:=1;
      while(i<length(a)) or (j<length(b)) do
      begin if(i=length(a)) then
            begin kam[k]:=b[j]; inc(k); inc(j);
            end else if(j=length(b))then DTTO...
                  else {oba seznamy neprazdne}
                        if(a[i]<b[j]) then
                        begin c[k]:=a[i];
                              inc(k); inc(i);
                        end else begin
                              c[k]:=b[j]; inc(k,j);
                        end;
end; end;
```

Mergesort – myšlenka implementace

- Třídíme posloupnost c o délce n.
- if $n > 1$ then rozdel c na a a b přibližně stejných délek
- mergesort(a);
- mergesort(b);
- merge(a,b,c);

Mergesort – analýza složitosti

- Předpokládejme, že n je mocnina dvojky (jinak vstup nejvýše zdvojnásobíme falešnými hodnotami).
- Kolikrát budeme dělit vstupní pole [a tedy jak hluboká bude rekurze]?
- Kolikrát musíme n vydělit dvojkou, abychom dostali jedničku? $\log n$-krát.
- Jaká je složitost samotného těla funkce `mergesort`?
- Lineární (v délce vstupu).
- budeme jednotlivé hladiny rekurze analyzovat dohromady (režie každé hladiny rekurze bude lineární).
- Hladin rekurze je $O(\log n)$, tedy celkem je složitost $O(n \log n)$.
- Bylo by též možno použít tzv. Master theorem, tento výpočet je ale totožný s důkazem jedné jeho varianty (té s rovností).

Mergesort – poznámky

- Všimněte si, že Mergesort má pro všechny vstupy stejnou složitost!
- Jde o čistou vydestilovanou rekurzi, kdy žádná funkce vlastně nic nedělá, ale dohromady to funguje!
- Algoritmus poměrně kriticky vyžaduje místo navíc!

Třídění porovnáním

- Třídící algoritmy jsou často založené na porovnání.
- Naše algoritmy (zatím uvedené) jsou široce použitelné, protože nepotřebují o tříděných prvcích vědět víc, než který je větší a který menší.
- Takové nejsou všechny třídící algoritmy!
- Zatím se nám nepodařilo složitostí třídicích algoritmů dostat do $o(n \log n)$. Jsme jen neschopní, nebo je za tím nějaký přírodní zákon?
- Je za tím přírodní zákon.

Algoritmy třídící porovnáním

Definition

Řekneme, že třídicí algoritmus je algoritmem třídění porovnáním, jestliže algoritmus kromě porovnání (dvou čísel) nevyužívá žádné jiné informace o číslech.

- Například algoritmus nesmí spoléhat na to, zda čísla jsou celá, kladná, iracionální...
- Takový algoritmus je vlastně schopen třídit cokoliv, na čem je dobře definováno porovnání na vlastnost *býti menší nebo roven.*

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.