Algoritmi in podatkovne strukture – 2

Slovar

B-drevesa in 2-3 drevesa

Podatki in računalniška arhitektura

- Arhitektura računalnika definira cene posameznih operacij.
- Ozko grlo predstavlja prenos podatkov med posameznimi sklopi (pomnilniška hierarhija): procesorjevi registri, predpomnilnik, pomnilnik, disk, omrežje itd.
- Ko so podatki že v registrih, je cena operacije odvisna od cene posameznih procesorjevih operacij in običajno štejemo *primerjave*.
- V ostalih primerih štejejo predvsem prenosi podatkov med podsklopi – štejo pogledi/dospoti (ang. *probes*). Ti lahko trajajo tudi več desetkrat toliko kot ena primerjava.
- V enem dostopu pa ne prestavimo samo enega podatka med podsklopoma, ampak jih prestavimo več – pač glede na na velikost prestavljenega bloka in glede na velikost podatka.
- Recimo, da jih prestavimo b – v resnici prestavimo polje velikosti b podatkov:

$$\texttt{Podatek data[}b\texttt{]}$$

B-drevesa in dvojiška drevesa

Osnovne razlike:

- Vozlišča imajo lahko tudi več kot dva naslednika (odvisno od stopnje vozlišča).
- V vozliščih hranimo več ključev (odvisno od stopnje vozlišča).
- Vsi listi so na isti globini h – *uravnoteženost*

Definicija

B-drevo reda b ($b \geq 2$) je drevo, ki zadošča naslednjim lastnostim:

vozlišče - ključi vsako vozlišče v ima k_v ključev, kjer $\lceil b/2 \rceil - 1 \geq k_v < b$, razen korena, ki ima lahko tudi samo en ključ. Ključe označimo z `v.key[i]`, kjer $0 \leq i < k_v$ in velja `v.key[i]` $<$ `v.key[i+1]` – urejenost ključev.

vozlišče - poddrevesa vozlišče v, ki ima k_v ključev, ima $k_v + 1$ poddreves `v.sub[i]` ($0 \leq i \leq k_v$) razen listov, ki nimajo poddreves.

vozlišča - urejenost velja, da so vsi elementi ≫levo≪ od ključa `v.key[i]` (v poddrevesu `v.sub[i]`) manjši od `v.key[i]` in elementi ≫desno≪ od ključa `v.key[i]` (v poddrevesu `v.sub[i+1]`) večji od (ali enaki) `v.key[i]`. To velja za vse $0 \leq i < k_v$.

listi vsi listi so na isti globini h.

Prvi dve vrstici zagotavljata eksponentno povečevanje elementov na posamezni ravni, tretja omogoča iskanje po principu deli in vladaj in zadnja zagotavlja uravnoteženost, ki zagotavlja enak čas pri operacijah nad katerimkoli podatkom v drevesu.

Lastnosti

- Najmanjši b, za katerega je definicija smiselna, je $b = 3$. Tedaj ima vozlišče v

$$\lceil b/2 \rceil - 1 = 1 \geq k_v < b = 2$$

ključev, oziroma dve ali tri poddrevesa. Zato takšnim drevesom rečemo tudi *2-3 drevesa*.

- Imejmo B-drevo reda b in višine h. V takšnem drevesu je lahko največ b^h in najmanj $(b/2)^{h-1} \cdot 2$ elementov.
- Če obrnemo in imamo B-drevo reda b z n elementi, potem je njegova višina največ

$$1 + \log_{\lceil b/2 \rceil} \frac{n+1}{2} \ .$$

- Recimo, da je $b = 199$ in imamo v drevesu $1.999.998$ ključev, potem je $h \leq 3$. To pomeni, da bomo pri iskanju ključa pregledali kvečjemu tri vozlišča – drugače, *dostopili* bomo samo do treh vozlišč!

B drevesa

- V drevesu hranimo elemente – `Elt` (ki pa sestoje iz ključa in podatka).
- Višina drevesa je odvisna od števila ključev v vozlišču (reda) in *večji je red*, *nižje je drevo* ter *hitrejše so operacije*.

```
public class bTree {
  Elt   elt[b];      // podatki
  bTree sub[b+1];    // poddrevesa
  int   k;           // število elementov v vozlišču
  ...
}
```

B+ drevesa

- Fizična velikost (število zlogov) posameznega vozlišča definira računalniška arhitektura, oziroma pomnilniška hierarhija ter je ne moremo spreminjati – npr. dolžina predpomnilniške vrstice, velikost sektorja na disku, velikost ethernet paketa ipd..
- Torej je najbolje, če v vozlišča ne dajemo celotnih elementov, ampak samo njihove ključe, medtem ko podatke hranimo ločeno – govorimo o *B*+ drevesih.
- Na teh predavanjih bomo imeli opravka z B+ drevesi.

```
public class bPlusTree extends bPlusNode {
  Key       key[b];     // ključi
  bPlusNode sub[b+1];   // poddrevesa ali podatki
  int       k;          // število elementov v vozlišču
  bool      leaf;       // notranje vozlišče ali list (potrebno?)
  ...
}
```

- Pri listu imamo v `bPlusData` podatke

```
public class bPlusData extends bPlusNode {
  ...      data;         // podatki
  ...
}
```

Iskanje

Iskanje ključa `key` v B-drevesu s korenom `bTree` opravi naslednji preprosti algoritem:

```
bPlusData Search(Key key) {
  int i= 0;
  while ( (i < k) && (key[i] > key) ) i++;
  if (! leaf)          return sub[i].Search(key);
  else {
    if (key[i] == key) return sub[i];
    else               return DataInvalid;
  }
}
```

Iskanje – analiza

Časovna zahtevnost je $h = O(\log_b n)$ dostopov (pri tem ne štejemo še dodatnega dostopa do podatka), oziroma

$$b \log_b n = \frac{b}{\lg b} \lg n$$

primerjav ključev (ni povsem točno, ker je v vozlišču lahko samo $b/2$ ključev in v korenu samo 2).

Če bi namesto linearnega iskanja po vozlišču uporabili bisekcijo, bi se število primerjav spremenilo na

$$\lg b \log_b n = \frac{\lg b}{\lg b} \lg n = \lg n$$

primerjav, kar je primerljivo z dvojiškimi drevesi.

Vstavljanje

Vstavljamo element `Elt` s ključem `Elt.key` in recimo, da še ne obstaja takšen ključ v drevesu.

1. Novi element vedno vstavimo v list – do tam se sprehodimo na enak način kot pri iskanju. Pri tem opazimo, da imamo pri iskanju vedno opravka s ključem `v.key[i]`, ki je večji od vstavljanega ključa in poddrevesom `v.sub[i]`, v katerega vstavljamo (pozor, obstaja poseben primer, ko vstavljamo v zadnje poddrevo).
2. Če je v listu (vozlišču) še prostor (manj kot $b-1$ element), element vstavimo med elementa tako, da se ohrani naraščajoče zaporedje ključev.
3. Če v vozlišču ni prostora, pomeni, da imamo $b+1$ element, ki so urejeni po velikosti glede na ključe. Razdelimo jih na tri dele:
   - prvih $\lceil b/2 \rceil - 1$ elementov,
   - en element in
   - preostali elementi.

   Iz prvega in tretjega dela naredimo dve novi vozlišči b_1 in b_2 (za eno lahko sicer uporabimo kar staro vozlišče), ki sta v resnici B-drevesi.

Na koncu vrnemo staršu obe poddrevesi in srednji element: b_1, x, b_2 in starš mora sedaj:

- nadomestiti `v.sub[i]` z b_2 in
- predenj in pred `v.key[i]` vstaviti b_1 odnosno x.

4. Pri staršu ponovimo bodisi korak 2 bodisi 3. Seveda v drugem primeru ponavljamo korak naprej proti korenu.
5. Če pa moramo korak 3 opraviti pri korenu (v bistvu razpolovimo koren drevesa), dobi celo drevo nov koren, ki bo imel samo en ključ.

V najslabšem primeru razpolovimo h vozlišč in zatorej potrebujemo največ nekako $2h + 1$ dostopov, kar je $2\log_b n + 1$.

B-drevesa – brisanje

Postopek in njegove lastnosti

- Podobno kot pri binarnih drevesih – izbrisani ključ nadomestimo s skrajnim levim (desnim) ključem v desnem (levem) poddrevesu
- pri tem se lahko zmanjša število elementov v listu pod mejo $\frac{b}{2}$ – kaj sedaj?
- karkli že naredimo, ponavljamo rekurzivno do korena, ki lahko, izgine

Zapletenost

	`Find`	`Insert`	`Delete`
seznam	$O(n)$	$O(1)$	$O(n)$
urejen seznam	$O(n)$	$O(n)$	$O(n)$
binarno drevo	$O(n)$	$O(n)$	$O(n)$
AVL drevo	$O(\log n)$	$O(\log n)$	$O(\log n)$
B drevo	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$

- In koliko je primerjav?
- OPAŽANJE: popravek je potreben pri staršu, če je tudi starš v $\gg$robnem stanju$\ll$.

Primer

- vstavimo: 20, 11, 3, 1, 30, 15, 13, 12, 47, 17, 100, 110.
- izločimo: 1, 3, 11, 12, 20.