Podatkovne strukture in algoritmi

Andrej Brodnik
UP FAMNIT

Slovar

osnove, povezan seznam,
dvojiška, iskalna in uravnotežena drevesa

Elementi v slovarju

Elementi, ki so shranjeni v slovarju imajo lahko naslednje lastnosti:

- lahko so samostojni
- lahko sestoje iz ključa in podatkov:

```
public class Elt {
  public Object key;
  public Object data;
}
```

Matematično gledano je `Elt` podoben **preslikavi (map)**, saj slika `key` v `data`. Zato se v knjižnicah slovar pojavlja pod imenom `map`.

Pri tem predmetu bomo predpostavili:

- da so elementi slovarja vedno pari in
- da so ključi vedno cela števila, da poenostavimo razlago.

Slovar

Imamo slovar S nekakšnih elementov. S tem slovarjem želimo početi vsaj naslednje operacije:

- dodajanje: `Insert(`S`, x)` – v slovar S dodamo nov element `x`.
- iskanje: `Find(`S`, x) --> y` – v slovarju S poiščemo element `x`. Rezultat `y` je lahko Boolova vrednost `true` ali `false`, ali pa neki podatki povezani z elementom `x`.
- izločanje: `Delete(`S`, x) -->` S – iz slovarja S izločimo element `x`. Rezultat `y` je lahko Boolova vrednost `true` ali `false`, ki sporoči ali je bil element uspešno izločen ali ne, ali pa operacija ničesar ne vrne.

Slovar je v resnici podoben matematični strukturi množica, le da pri slovarju elementi lahko sestoje še iz podatkov (glej `y` pri iskanju).

Primer

Imejmo slovar, v katerem so elementi: $S = \{45, 2, 17, 46\}$. Potem lahko sledimo operacijam nad njim.

Pozor: Ničesar nismo rekli o tem, kako so ORGANIZIRANI *(shranjeni, strukturirani) elementi v slovarju S!*

Izvedba slovarja

Ogledali si bomo več izvedb slovarja:

- izvedba s seznamom: neurejen, urejen
- izvedba z dvojiškimi drevesi: iskalna, uravnotežena, AVL, rdeče-črna drevesa
- izvedba z večsmernimi drevesi: B-drevesa, 2-3 drevesa
- izvedba s preskočni seznami
- izvedba z razpršenimi tabelami: odprto naslavljanje in veriženje
- kot Bloomov filter

Katerokoli izvedbo lahko razširimo z dodatnimi podatki, ki nam omogočajo dodatne operacije nad elementi: *rang* in *izbira*.

Seznami

Recimo, da imamo seznam $[45, 2, 17, 46]$, potem velja:

- da je 45 glava seznama in $[2, 17, 46]$ rep;
- da je 2 glava seznama in $[17, 46]$ rep;
- da je 17 glava seznama in $[46]$ rep;
- da je 46 glava seznama in $[]$ rep;

Seznam sestoji iz:

- glave, ki bo v našem primeru iz razreda `Elt` in
- repa, ki je ponovno seznam.

Poseben seznam je prazen seznam (`null`).

Naš seznam je sedaj naslednji:

$$\underline{[45\underline{[2\underline{[17\underline{[46\underline{[\texttt{null}]}]}]}]}]}$$

Iskanje

```
public Object Find(int key) {
  if this == NULL return NULL
  else if (glava.key == key) return glava.data;
  else return rep.Find(key);
}
```

Vstavljanje

```
public Seznam Insert(Elt element) {
  return new Seznam(element, this);
}
```

Brisanje

```
public Object Delete(int key) {
  if (this == NULL) return NULL
  else if (glava.key == key) return rep;
  else return new Seznam(this.glava, rep.Delete(key));
}
```

Urejeni seznam

Intuicija:

- drago se je sprehoditi čez cel seznam in še posebej tedaj, ko elementa ni v seznamu;
- če bi bili elementi urejeni po velikosti, bi hitreje videli, da elementa ni v seznamu.

Namesto navadnega seznama lahko uporabimo urejen seznam, v katerem so elementi urejeni po velikosti.

Iskanje

```
public Object Find(int key) {
  if this == NULL return NULL
  else if (glava.key == key) return glava.data;
  else if (glava.key > key) return NULL;
  else return rep.Find(key);
}
```

Vstavljanje

```
public Seznam Insert(Elt element) {
  poišči prvi večji element
  vstavi pred njega element
}
```

Brisanje

```
public Object Delete(int key) {
  if (this == NULL) return NULL
  else if (glava.key == key) return rep;
  else if (glava.key > key) return this;
  else return new Seznam(this.glava, rep.Delete(key));
}
```

Analiza

	`Find`	`Insert`	`Delete`
seznam	$O(n)$	$O(1)$	$O(n)$
urejen seznam	$O(n)$	$O(n)$	$O(n)$

- Kakšna je velikost?
- Kaj pa najboljši čas?
- In kaj povprečen čas?
- So zgornje vrednosti smiselne?
- Recimo, da poznamo vsa poizvedovanja vnaprej – optimalni čas.
- KISS – *Keep it simple, stupid.*

Drevesa

Pri seznamu smo imeli *glavo* in *rep*, sedaj pa imamo *glavo* in dva repa:

```
public class Drevo {
  Elt koren;
  Drevo levo, desno;    ...
}
```

Nekaj definicij

- vozlišča, ki so koreni poddreves, pravimo, da so *nasledniki* (*children*) korena
- obratno, koren je naslednikom *starš* (*parent*)
- nasledniki so si med seboj *sorodniki* (*siblings*)
- vozlišča drevesa, ki nimajo poddreves se imenujejo *listi* (*leaves*) ali *zunanja vozlišča* (*external nodes*), ostala pa se imenujejo *notranja vozlišča* (*internal nodes*)
- posebno vozlišče je *koren* (*root*) – to je vozlišče, ki nima staršev
- globina vozlišča v je razdalja od v do korena drevesa – vključno s korenom in v
- globina ali višina drevesa je število vozlišč na najdaljši poti od korena do nekega lista – h
- popolno k-tiško drevo je drevo, kjer ima vsako notranje vozlišče bodisi k naslednikov ali nobenega

Pregledovanje dvojiških dreves

- premi (*preorder*) najprej »obdela« koren, nato levo poddrevo in na koncu desno poddrevo
- vmesni (*inorder*) najprej »obdela« levo poddrevo, nato koren in na koncu desno poddrevo
- obratni (*postorder*) najprej »obdela« levo poddrevo, nato desno poddrevo in na koncu koren

Vmesni pregled

```
public void VmesniPregled() {
  if (levo != NULL) levo.VmesniPregled();
  System.out.println(koren.key);
  if (desno != NULL) desno.VmesniPregled();
}
```

- Kako izpiše `VmesniPregled`, če imamo opravka z urejenim drevesom?
- Spremenite zgornjo metodo, da boste obiskali drevo po premi in po obratni poti. Kako se sedaj izpišejo elementi urejenega drevesa?
- Kako popraviti definicijo zgornje metode, da boste lahko namesto izpisa (`System.out.println`) opravili poljubno operacijo nad korenom.
  (NAMIG: bistvo rešitve je v uporabi rokovalnika – kako?)

Urejena (iskalna) drevesa

Pri urejenih drevesih velja:

- vsi elementi v levem poddrevesu so manjši od `koren` in
- vsi elementi v desnem poddrevesu so večji od `koren`.

Ali je zgornja definicija zadovoljiva, če imamo v drevesu več enakih elementov?

Primer

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

Iskanje

```
public Object Find(int key) {
  if (koren.key == key) return koren.data;
  else if (key < koren.key)
    if (levo == NULL) return NULL;
    else return levo.Find(key);
  else
    if (desno == NULL) return NULL;
    else return desno.Find(key);
}
```

Časovna zahtevnost je $\Theta(h)$. Kako velik je lahko h? Kako majhen je lahko h? Torej?

Vstavljanje

```
public Drevo Insert(Elt element) {
  if (element.key < koren.key)
    if (levo == NULL) levo= new Drevo(element, NULL, NULL);
    else levo= levo.Insert(element);
  else
    if (desno == NULL) desno= new Drevo(element, NULL, NULL);
    else desno= desno.Insert(element);
  return this;
}
```

- Vedno dodajamo v list.
- Časovna zahtevnost je $\Theta(h)$.
- Kako velik je lahko h? Kako majhen je lahko h? Torej?

Brisanje

Kako brišemo:

- če vozlišče nima naslednikov, ni težav – ga izbrišemo,
- sicer v desnem (levem) poddrevesu poiščemo največji (najmanjši) element in ga vstavimo na mesto izbrisanega drevesa;
- po potrebi na tako prenešenem (izbrisanem) elementu izvedemo rekurzivni popravek.

Časovna zahtevnost je $\Theta(h)$.

Napišite programsko kodo.

Uravnotežena drevesa

Težava: pri »običajnih« iskalnih drevesih: globina drevesa je lahko v najslabšem primeru n.

Rešitev: »uravnotežena drevesa«, za katera velja $h = \Theta(\log n)$.

Definicija. Drevo je *uravnoteženo*, če je razlika v številu elementov v levem in desnem poddrevesu največ 1.

Težko zagotoviti, saj vstavljanje lahko traja $O(n)$ – zato definicijo uravnoteženosti omilimo.

Drevesa AVL

Definicija. Drevo je *uravnoteženo*, če za vsako vozlišče velja: globini levega in desnega poddrevesa se razlikujeta kvečjemu za $O(1)$.

Pri AVL (Adel'son-Velskiĭ in Landis) uravnoteženega drevesih je globina levega poddrevesa največ za ena različna od globine desnega poddrevesa.

Za AVL drevesa velja, da je globina drevesa h z n notranjimi vozlišči vedno

$$\log(n+1) \leq h \leq 1.4404 \log(n+2) - 0.328 \quad ,$$

torej $h = \Theta(\log n)$.
Zgornjo mejo prinese t.i. *Fibonaccijevo drevo*.

Drevesa AVL – nadalj.

Razširimo definicijo vozlišča:

- doslej – levo in desno poddrevo in
- ključ; dodamo
- višina (razlika višin)

Operacije:

iskanje: nespremenjeno;

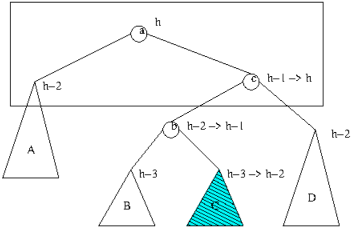
vstavljanje: v dveh korakih: najprej vstavimo kot prej, potem popravimo uravnoteženost;

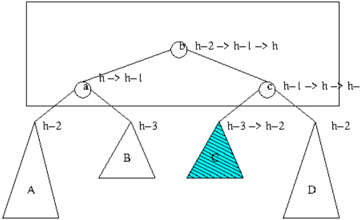
brisanje: podobno kot prej, le da sedaj pride lahko do neuravnoteženosti že pri brisanju najmanjšega elementa v desnem poddrevesu.

- Neuravnoteženo drevo moramo nekako popraviti. Imamo več možnih napak, ki jih odpravimo z *vrtenji* (*rotation*).

Skodirajte vse metode. Pozor, rezultat tako pri vstavljanju kot brisanju sedaj vsebuje ne splošno drevo, ampak AVL drevo.

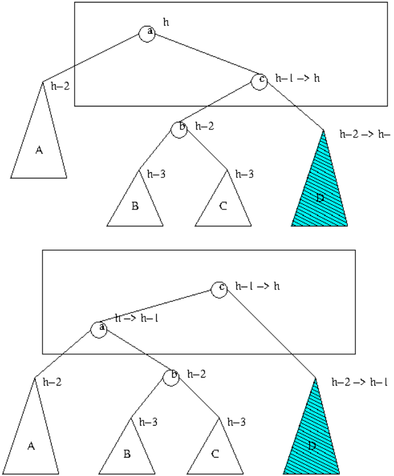
Vstavljanje – primer 1





- Če je popravek potreben, je to *samo eden*!
- Časovna zahtevnost: $\Theta(\log n)$).

Vstavljanje – primer 2



- Sedaj pride lahko do neuravnoteženosti že pri brisanju najmanjšega (največjega) elementa v desnem (levem) poddrevesu (lahko vse od lista nazaj do korena
- Časovna zahtevnost: $\Theta(\log n)$).

Analiza

Časovna zahtevnost:

	`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	$O(\log n)$	$O(\log n)$	$O(\log n)$

- Kakšna je velikost? Je razlika med seznamom in drevesom?
- Kaj pa najboljši čas?
- In povprečni čas?
- So zgornje vrednosti smiselne?
- Recimo, da poznamo vsa poizvedovanja v naprej – optimalni čas.