Algoritmi in podatkovne strukture – 2

Slovar

drevesa: osnove, iskalna, uravnotežena

Drevesa

Pri seznamu smo imeli *glavo* in *rep*, sedaj pa imamo *glavo* in več repov – k:

```
public class kDrevo {
  Elt[]    koren;      // teh je k − 1
  kDrevo[] poddrevo;   // teh je k
  ...
}
```

Običajno $k = 2$ in takrat govorimo o dvojiških drevesih. Tedaj imamo samo dve poddrevesi, ki ju imenujemo *levo* in *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

Popolno k-tiško drevo

- (Po)polno k-tiško drevo je drevo, v katerem imajo vsi listi enako globino.
- Število notranjih vozlišč v polnem k-tiškem drevesu globine h je

$$1 + k + k^2 + \cdots + k^{h-1} = \frac{k^h - 1}{k - 1}.$$

- popolna k-tiška drevesa lahko shranimo kot implicitno podatkovno strukturo – kako?

Urejena drevesa

Pri urejenih drevesih velja, da za poljuben $0 \leq i < k - 1$ velja:

- vsi elementi v poddrevesu `d[`i`]` so manjši od `koren[`i`]`
- vsi elementi v poddrevesu `d[`$i + 1$`]` so večji od `koren[`i`]`

Kako je pri dvojiških drevesih?

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

Odslej se bomo ukvarjali samo z dvojiškimi drevesi.

DOMAČA NALOGA: dopolnite naslednje prosojnice tako, da bodo veljale za k-tiška drevesa.

Podatki v drevesu

Podatke v drevesu lahko shranimo na dva različna načina:

- v notranjih in zunanjih vozliščih:

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

- samo v listih:

```
public class Drevo {
  int    koren;
  Object levo, desno;  // bodisi Elt ali Drevo
  ...
}
```

DOMAČA NALOGA: kaj lahko poveste o takšnih drevesih in njihovi popolnosti?

Iskalna drevesa

Pregledi (iskalnega) dvojiškega drevesa:

- 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?)

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?

Iskanje – nerekurzivno

```
public Object Find(int key) {
  Drevo drevo= this;
  while ((drevo != NULL) && (drevo.koren.key != key)) {
    if (key < drevo.koren.key) drevo= levo;
    else                       drevo= desno;
  };
  if (drevo == NULL) return NULL;  // kdaj se to zgodi?
  else return drevo.koren.data;
}
```

Zakaj bi želeli nerekurzivno iskanje?

Se časovna zahtevnost kaj spremeni?

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;
}
```

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

Vstavljanje – tudi nerekurzivno

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

In sedaj, se časovna zahtevnost kaj spremeni?

Brisanje

Kako brišemo:

- če vozlišče nima naslednikov, ni težav – ga izbrišemo,
- če ima vozlišče eno poddrevo, ga izbrišemo in na njegovo mesto postavimo njegovo poddrevo (t.j. njegovega naslednika),
- če ima vozlišče dve poddrevesi, ga nadomestimo z najmanjšim elementom v desnem poddrevesu, ali pa z . . . ???

Brisanje – rezultat

Pri brisanju bomo vrnili dva predmeta `Pair`:

- sam element, ki smo ga izbrisali (koren drevesa, ki smo ga izbrisali) in
- drevo, ki je ostalo, v katerem ni več brisanega elementa

Brisanje – koda

```
public Pair Delete(int key) {
  Pair rezultat;
  if (koren.key < key) {
    if (levo == NULL) return new Pair(NULL, this);
    rezultat= levo.Delete(key);
    levo= rezultat.drevo; rezultat.drevo= this;
    return rezultat;
  };
  else if (koren.key == key) {
    if (levo == NULL)       return new Pair(koren, desno);
    else if (desno == NULL) return new Pair(koren, levo);
    else {
      Elt element= koren;
      rezultat= desno.MinDelete();
      koren= rezultat.elt;   rezultat.elt= element;
      desno= rezultat.drevo; rezultat.drevo= this;
      return rezultat;
    };
  else { ... };
}
```

Brisanje – zahtevnost

In časovna zahtevnost tega postopka? Kaj pa nerekurzivna rešitev?

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 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

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

torej $h = \Theta(\log n)$.

Zgornjo mejo prinese t.i. *Fibonaccijevo drevo*.

AVL drevesa – vzdrževanje uravnoteženosti

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

AVL drevesa – operacije

razred: drevesu (korenu) dodamo informacijo o višini. V resnici bi lahko samo dodali informacijo o razliki višin levega in desnega poddrevesa.

iskanje: nespremenjeno

vstavljanje: v dveh korakih: najprej vstavimo kot prej, potem popravimo uravnoteženost (največ ena)

brisanje: podobno kot prej, le da sedaj pride lahko do neuravnoteženosti že pri brisanju najmanjšega elementa v desnem poddrevesu (lahko vse od lista nazaj do korena – $\Theta(\log n)$).

DOMAČA NALOGA: skodirajte vse metode. Pozor, rezultat tako pri vstavljanju kot brisanju sedaj vsebuje ne splošno drevo, ampak AVL drevo.

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

- Kakšna je velikost?
- Kaj pa najboljši čas?
- In kaj pa povprečen čas?
- So zgornje vrednosti smiselne?
- Recimo, da poznamo vsa poizvedovanja v naprej – optimalni čas.

Primer

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