Podatkovne strukture in algoritmi

Andrej Brodnik
UP FAMNIT

Vrste s prednostjo

osnove, izvedba s kopico

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) --> y` – 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.

Posplošeni slovar

Poleg omenjenih operacij imamo še opearcije:

- rang: `Rank(`S`, x) --> i` – `x` je `i`-ti element po velikosti v S.
- izbira: `Select(`S`, i) --> x` – `x` je `i`-ti element po velikosti v S. Če takšnega elementa ni, vrne `null`.
- levi sosed: `Left(`S`, x) --> y` – `y` je največji element v S, ki je manjši od `x`. Če takšnega elementa ni, vrne `null`.
- desni sosed: `Right(`S`, x) --> y` – `y` je najmanjši element v S, ki je večji od `x`. Če takšnega elementa ni, vrne `null`.
- sosed: `Neighbour(`S`, x) --> y` – v slovarju S poiščemo element `y`, ki je najbližji element `x`. Če takšnega elementa ni, vrne `null`.

Pri posplošenem slovarju imamo opravka z urejeno množico elementov.

Posplošeni slovar

Poleg omenjenih operacij imamo še opearcije:

- **rang:** `Rank(`S`, x) --> i` – `x` je `i`-ti element po velikosti v S.
- **izbira:** `Select(`S`, i) --> x` – `x` je `i`-ti element po velikosti v S. Če takšnega elementa ni, vrne `null`.
- **levi sosed:** `Left(`S`, x) --> y` – `y` je največji element v S, ki je manjši od `x`. Če takšnega elementa ni, vrne `null`.
- **desni sosed:** `Right(`S`, x) --> y` – `y` je najmanjši element v S, ki je večji od `x`. Če takšnega elementa ni, vrne `null`.
- **sosed:** `Neighbour(`S`, x) --> y` – v slovarju S poiščemo element `y`, ki je najbližji element `x`. Če takšnega elementa ni, vrne `null`.

Pri posplošenem slovarju imamo opravka z urejeno množico elementov. Posebej zanimivi sta operaciji:

- **najmanjši:** `Min(`S`)` $\equiv$ `Right(`S`,` $-\infty$`) --> y`
- **največji:** `Max(`S`)` $\equiv$ `Left(`S`,` $+\infty$`) --> y`

Vrsta s prednostjo

Imamo urejeno množico elementov S.

```
public class OrdElt extends Elt {
  public boolean Bigger(OrdElt other) { ... }
}
```

Mi se bomo omejili na ključe iz množice celih števil.

Vrsta s prednostjo

Imamo urejeno množico elementov S.

```
public class OrdElt extends Elt {
  public boolean Bigger(OrdElt other) { ...  }
}
```

Mi se bomo omejili na ključe iz množice celih števil.

Nad njimi želimo početi naslednje operacije:

dodajanje: `Insert(`S`, x)` – v S dodamo nov element `x`.

najmanjši: `Min(`S`) --> y` – v S poiščemo najmanjši element `y`.

odreži: `DelMin(`S`)` – iz S izločimo najmanjši element.

Posplošena vrsta s prednostjo

Poleg omenjenih, so možne še operacije:

izločanje: `Delete(`S`, x) --> y` – iz 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.

spreminjanje: `Decrease(`S`, x, d)` – v S elementu `x` zmanjšamo (povečamo) vrednost za `d`.

zlij: `Merge(`S_1`, `S_2`) -->` S – zlije vrsti s prednostjo v novo vrsto s prednostjo.

levi sosed: `Left(`S`, x) --> y` – v S poiščemo element `y`, ki je največji element, kateri je še manjši od `x`. Če takšnega elementa ni, vrne `null`.

desni sosed: `Right(`S`, x) --> y` – v S poiščemo element `y`, ki je najmanjši element, kateri je še večji od `x`. Če takšnega elementa ni, vrne `null`.

Danes se bomo ukvarjali samo z osnovno obliko vrst s prednostjo.

Izvedba s seznamom

Najpreprostejša oblika izvedbe vrste s prednostjo je *urejen* seznam:

$$(2, 8, 10, 11, 13, 19, 20, 22, 23, 29)$$

Izvedba s seznamom

Najpreprostejša oblika izvedbe vrste s prednostjo je *urejen* seznam:

$$(2, 8, 10, 11, 13, 19, 20, 22, 23, 29)$$

Najmanjši element najdemo v $O(1)$ času in prav tako ga odrežemo v $O(1)$ času.

Izvedba s seznamom

Najpreprostejša oblika izvedbe vrste s prednostjo je *urejen* seznam:

$$(2, 8, 10, 11, 13, 19, 20, 22, 23, 29)$$

Najmanjši element najdemo v $O(1)$ času in prav tako ga odrežemo v $O(1)$ času.
Čas dodajanja je sorazmeren dolžini seznama, oziroma, v najslabšem primeru moramo narediti n primerjav.

Izvedba s seznamom

Najpreprostejša oblika izvedbe vrste s prednostjo je *urejen* seznam:

$$(2, 8, 10, 11, 13, 19, 20, 22, 23, 29)$$

Najmanjši element najdemo v $O(1)$ času in prav tako ga odrežemo v $O(1)$ času.
Čas dodajanja je sorazmeren dolžini seznama, oziroma, v najslabšem primeru moramo narediti n primerjav.

Kaj je dobrega v tej izvedbi?
Kaj je slabega?

Katere lastnosti opazujemo?

Izvedba z drevesom

- čas iskanja najmanjšega elementa: $O(\log n)$
- čas izločanja najmanjšega elementa: $O(\log n)$
- čas dodajanja elementa: $O(\log n)$

Zakaj vedno v najslabšem primeru logaritemski čas?

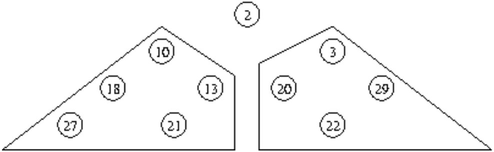
Opažanja

- v iskalnem drevesunajmanjši element je skoraj vedno v listu in kar dve operaciji imata opravka z njim – $O(\lg n)$
- elementi v strukturi so urejeno (vmesni obhod tvori urejen seznam elementov)

Glede prve lastnosti bi bilo pa dobro, če bi bil najmanjši element v korenu. Druga lastnost pa zahteva preveč dela za upravljanje strukture.

Kopica

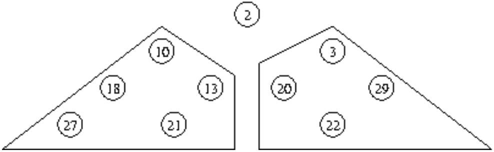
Kopico (rekurzivno) definiramo z naslednjimi lastnostimi:



- kopica sestoji iz korena in dveh podkopic, ki pa sta lahko prazni;

Kopica

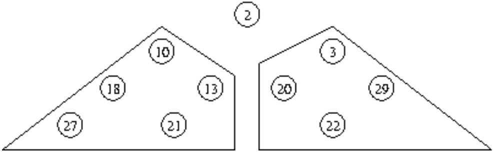
Kopico (rekurzivno) definiramo z naslednjimi lastnostimi:



- kopica sestoji iz korena in dveh podkopic, ki pa sta lahko prazni;
- najmanjši element je v korenu;

Kopica

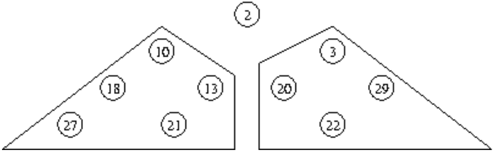
Kopico (rekurzivno) definiramo z naslednjimi lastnostimi:



- kopica sestoji iz korena in dveh podkopic, ki pa sta lahko prazni;
- najmanjši element je v korenu;
- v vsaki od podkopic je (približno) enako število elementov.

Kopica

Kopico (rekurzivno) definiramo z naslednjimi lastnostimi:

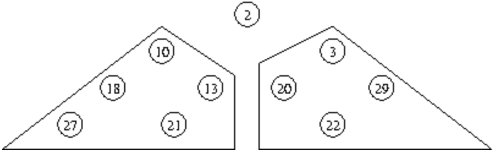


- kopica sestoji iz korena in dveh podkopic, ki pa sta lahko prazni;
- najmanjši element je v korenu;
- v vsaki od podkopic je (približno) enako število elementov.

Ali je pomemembno kateri elementi so v kateri od podkopic?

Kopica

Kopico (rekurzivno) definiramo z naslednjimi lastnostimi:



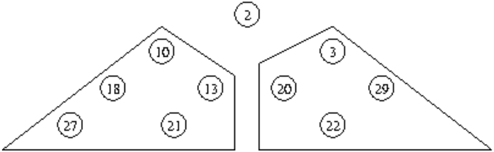
- kopica sestoji iz korena in dveh podkopic, ki pa sta lahko prazni;
- najmanjši element je v korenu;
- v vsaki od podkopic je (približno) enako število elementov.

Ali je pomemembno kateri elementi so v kateri od podkopic?

In operacije sedaj? Predvsem vstavljanje in rezanje najmanjšega elementa.

Kopica

Kopico (rekurzivno) definiramo z naslednjimi lastnostimi:



- kopica sestoji iz korena in dveh podkopic, ki pa sta lahko prazni;
- najmanjši element je v korenu;
- v vsaki od podkopic je (približno) enako število elementov.

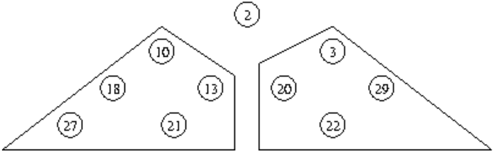
Ali je pomemembno kateri elementi so v kateri od podkopic?

In operacije sedaj? Predvsem vstavljanje in rezanje najmanjšega elementa.

Kaj pa zahtevnosti?

Kopica

Kopico (rekurzivno) definiramo z naslednjimi lastnostimi:



- kopica sestoji iz korena in dveh podkopic, ki pa sta lahko prazni;
- najmanjši element je v korenu;
- v vsaki od podkopic je (približno) enako število elementov.

Ali je pomemembno kateri elementi so v kateri od podkopic?

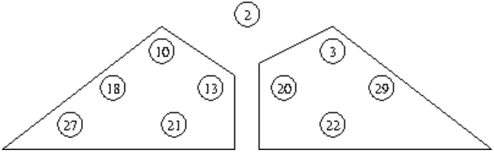
In operacije sedaj? Predvsem vstavljanje in rezanje najmanjšega elementa.

Kaj pa zahtevnosti?

Kaj lahko posplošimo?

Kopica

Kopico (rekurzivno) definiramo z naslednjimi lastnostimi:



- kopica sestoji iz korena in dveh podkopic, ki pa sta lahko prazni;
- najmanjši element je v korenu;
- v vsaki od podkopic je (približno) enako število elementov.

Ali je pomemembno kateri elementi so v kateri od podkopic?

In operacije sedaj? Predvsem vstavljanje in rezanje najmanjšega elementa.

Kaj pa zahtevnosti?

Kaj lahko posplošimo?

Kako izgleda vozlišče v pomnilniku?

Implicitne in eksplicitne podatkovne strukture

Obe vrsti struktur se shranjujeta v pomnilniku, le da pri *eksplicitnih podatkovnih strukturah* uporabljamo za sprehajanje po strukturi reference, ki se tudi *hranijo* v strukturi (pomnilniku).

Reference tudi zasedajo prostor v pomnilniku.

Ali lahko naredimo polje kot eksplicitno podatkovno strukturo? Kako? Kako je sploh definirano polje? Zakaj bi jo želeli narediti kot eksplicitno podatkovno strukturo?

Implicitna dvojiška drevesa

- če je v drevesu en element – preprosto ga shranimo;
- če je v drevesu n elementov, od katerih je n_l v levem in n_r v desnem, potem to naredimo tako, da (rekurzivno) pripravimo polje velikosti n:
  - damo koren na indeks 0 in z njim shranimo vrednost n_l;
  - levo poddrevo na indekse $1 ... n_l - 1$;
  - desno poddrevo na indekse $n_l ... n - 1$.

Opisan postopek velja za *kakršnokoli* dvojiško drevo.

Implicitna dvojiška drevesa

- če je v drevesu en element – preprosto ga shranimo;
- če je v drevesu n elementov, od katerih je n_l v levem in n_r v desnem, potem to naredimo tako, da (rekurzivno) pripravimo polje velikosti n:
  - damo koren na indeks 0 in z njim shranimo vrednost n_l;
  - levo poddrevo na indekse $1 \dots n_l - 1$;
  - desno poddrevo na indekse $n_l \dots n - 1$.

Opisan postopek velja za *kakršnokoli* dvojiško drevo.

Posplošitve? Kaj pa prostor? Je res to povsem implicitna podatkovna struktura?

Kopica

Kopica ni poljubno dvojiško drevo. Poglejmo jo malce drugače – od spodaj navzgor. Opazimo:

- spodnji nivo je (lahko) levo poravnan
- vsi drugi nivoji so (lahko) polni

Se kaj spremeni delovanje kopice, če se držimo te ureditve?

Implicitna kopica

- imejmo polje velikosti n, ki ga indeksiramo od 1 (če so indeksi od 0, malce »telovadbe«);
- koren shranimo na indeks 1;
- naslednjo plast na indekse 2 ter 3 in tako naprej

Implicitna kopica

- imejmo polje velikosti n, ki ga indeksiramo od 1 (če so indeksi od 0, malce »telovadbe«);
- koren shranimo na indeks 1;
- naslednjo plast na indekse 2 ter 3 in tako naprej

Pomembno je samo, ali se preprosto sprehajamo po kopici navzgor in navzdol:

- koren leve podkopice elementa i je na $2i$ (zakaj?);
- koren desne podkopice elementa i je na $2i + 1$ (zakaj?);
- starš elementa i je na $\lfloor i/2 \rfloor$.

Vstavljanje

- novi element damo na dno kopice – zadnji (ali naslednji) element polje je možno mesto;
- primerjamo ga z njegovim staršem in ju zamenjamo, če je potrebno; postopek ponavljamo *dokler je potrebno oziroma do korena*

Koliko primerjav je potrebnih?

Izločanje najmanjšega elementa – metoda 1

1. izločimo koren in na njegovo mesto postavimo zadnji element kopice;
2. POTAPLJANJE: primerjamo (novi) koren ter ga zamenjamo z manjšim od korenov podkopic (zakaj?) in rekurzivno nadaljujemo *dokler je potrebno ali do lista*

Se struktura ohranja? Zakaj?

Izločanje najmanjšega elementa – metoda 1

1. izločimo koren in na njegovo mesto postavimo zadnji element kopice;
2. POTAPLJANJE: primerjamo (novi) koren ter ga zamenjamo z manjšim od korenov podkopic (zakaj?) in rekurzivno nadaljujemo *dokler je potrebno ali do lista*

Se struktura ohranja? Zakaj?

Koliko primerjav?

Izločanje najmanjšega elementa – metoda 2

1. izločimo koren
2. POTAPLJANJE: manjšega od korenov podkopic postavimo v koren (zakaj?) in rekurzivno nadaljujemo *do lista*
3. ko smo prišli do dna, na mesto, kjer smo izločili zadnji element, prestavimo zadnji element kopice (zakaj?)
4. DVIGOVANJE: prestavljeni element primerjamo s staršem in, če je manjši, ju zamenjamo; rekurzivno ponavljamo *dokler je potrebno ali do korena*

Se struktura ohranja? Zakaj?

Izločanje najmanjšega elementa – metoda 2

1. izločimo koren
2. POTAPLJANJE: manjšega od korenov podkopic postavimo v koren (zakaj?) in rekurzivno nadaljujemo *do lista*
3. ko smo prišli do dna, na mesto, kjer smo izločili zadnji element, prestavimo zadnji element kopice (zakaj?)
4. DVIGOVANJE: prestavljeni element primerjamo s staršem in, če je manjši, ju zamenjamo; rekurzivno ponavljamo *dokler je potrebno ali do korena*

Se struktura ohranja? Zakaj?

Koliko primerjav tokrat?

Primerjava metod

Višina kopice je $\lg n$ in recimo, da nadomestni element konča svojo pot k nivojev nad listi. Potem imamo:

- pri metodi 1: $2(\lg n - k) = 2\lg n - 2k$ primerjav in
- pri metodi 2: $\lg n + k$ primerjav.

Če velja:

$$k = \frac{\lg n}{3}$$

je vseeno, katero metodo uporabimo.

Koliko elementov je v plasti i (0 so listi)? Ali lahko iz tega sklepamo kaj na verjetnost k?

Lahko kaj povemo o času vstavljanja?

Se dá metodo 2 izboljšati?

Zahtevnost

	`Min`	`DelMin`	`Insert`
urejen seznam	$O(1)$	$O(1)$	$O(n)$
uravnoteženo drevo	$O(\log n)$	$O(\log n)$	$O(\log n)$
dvojiška kopica	$O(1)$	$O(\lg n)$	$O(\lg n)$

- Pri kopicah (Floyd 64, Williams 64) je različen vodilni koeficient in drugi člen.

Zahtevnost

	`Min`	`DelMin`	`Insert`
urejen seznam	$O(1)$	$O(1)$	$O(n)$
uravnoteženo drevo	$O(\log n)$	$O(\log n)$	$O(\log n)$
dvojiška kopica	$O(1)$	$O(\lg n)$	$O(\lg n)$

- Pri kopicah (Floyd 64, Williams 64) je različen vodilni koeficient in drugi člen.
- Kaj pa `Delete(`S`, x)`, `Decrease(`S`, x, d)`?

Zahtevnost

	`Min`	`DelMin`	`Insert`
urejen seznam	$O(1)$	$O(1)$	$O(n)$
uravnoteženo drevo	$O(\log n)$	$O(\log n)$	$O(\log n)$
dvojiška kopica	$O(1)$	$O(\lg n)$	$O(\lg n)$

- Pri kopicah (Floyd 64, Williams 64) je različen vodilni koeficient in drugi člen.
- Kaj pa `Delete(`S`, x)`, `Decrease(`S`, x, d)`?
- In kaj `Merge(`S_1`, `S_2`)`?