Podatkovne strukture in algoritmi

Andrej Brodnik
UP FAMNIT

Vrste s prednostjo

binomska kopica in Fibonaccijeva kopica

Vrsta s prednostjo

Osnovne operacije nad vrsto s prednostjo:

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.

Vrsta s prednostjo

Osnovne operacije nad vrsto s prednostjo:

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.

Operacije nad posplošeno vrsto s prednostjo:

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.

Vrsta s prednostjo

Osnovne operacije nad vrsto s prednostjo:

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.

Operacije nad posplošeno vrsto s prednostjo:

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`.

Dosedanje izvedbe

operacija	**seznam**	**drevo**	**kopica**
dodajanje	$O(n)$	$O(\log n)$	$O(\log n)$
najmanjši	$O(1)$	$O(\log n)$	$O(1)$
odreži	$O(1)$	$O(\log n)$	$O(\log n)$
izločanje	$O(n)$	$O(\log n)$	$O(\log n)$
spreminjanje	$O(n)$	$O(\log n)$	$O(\log n)$
zlij	$O(\mu)$	$O(\mu \log n)$	$O(n)$
levi sosed	$O(n)$	$O(\log n)$	$O(n)$
desni sosed	$O(n)$	$O(\log n)$	$O(n)$

Komentarji:

izločanje: poznamo referenco elementa

spreminjanje: čas brisanja + čas vstavljanja

zlij: $\mu = \min(|S_1|, |S_2|)$

Dvojiška kopica

- *invarianca 1*: vsi elementi v L in D so večji ali enaki k
  - zagotavlja urejenost podatkovne strukture in
  - posledično hitro iskanje (najmanjšega elementa) – $O(1)$
- *invarianca 2*: podkopici L in D sta približno enako veliki
  - zagotavlja uravnoteženost strukture in zato učinkovita popravljanja – $O(\log n)$
- Kaj pa zlivanje?

Zlivanje kopic

- imamo kopici `A` in `B` ter tvorimo kopico `C`
- koren kopice `C` naj bo `min(A.koren, B.koren)` – recimo, da je to (WLOG) `A.koren`; in
- koren druge kopice postane naslednik novega korena – koren kopice `A` dobi *še enega* naslednika

Zlivanje kopic

- imamo kopici `A` in `B` ter tvorimo kopico `C`
- koren kopice `C` naj bo `min(A.koren, B.koren)` – recimo, da je to (WLOG) `A.koren`; in
- koren druge kopice postane naslednik novega korena – koren kopice `A` dobi *še enega* naslednika
- nova kopica *ni* več dvojiška
- še vedno omogoča hitro iskanje (invarianca 1)
- ni pa nujno (Murphy), da je uravnotežena (invarianca 2), kar pomeni, da popravljanja niso več učinkovita

Kaj smo dobili?

Kaj smo dobili?

- iskanje: $O(1)$ – **super**

Kaj smo dobili?

- iskanje: $O(1)$ – **super**
- zlivanje: $O(1)$ (zato tudi vstavljanje) – **super**

Kaj smo dobili?

- iskanje: $O(1)$ – **super**
- zlivanje: $O(1)$ (zato tudi vstavljanje) – **super**
- popravljanje: $O(\text{višina kopice})$ – **katastrofa**

Kaj smo dobili?

- iskanje: $O(1)$ – **super**
- zlivanje: $O(1)$ (zato tudi vstavljanje) – **super**
- popravljanje: $O(\text{višina kopice})$ – **katastrofa**
- Kaj sedaj?

Rešitev

TEHNIKA: Če nekje malce popustimo, bomo morda drugje lahko veliko pridobili – *princip ravnoteženja*.

Rešitev

TEHNIKA: Če nekje malce popustimo, bomo morda drugje lahko veliko pridobili – *princip ravnoteženja*.

Kaj nam pa omejuje mejo, do katere lahko popustimo? Koliko lahko v tem primeru popustimo?

Rešitev

TEHNIKA: Če nekje malce popustimo, bomo morda drugje lahko veliko pridobili – *princip ravnoteženja*.

Kaj nam pa omejuje mejo, do katere lahko popustimo? Koliko lahko v tem primeru popustimo?

- naj bo zlivanje (in iskanje) malce počasnejše – $O(\text{višina kopice})$
- bo pa zato struktura bolj uravnotežena, kar bo razorožilo Murphyja (nasprotnika, *adversary*) in bo zato tudi višina kopice manjša (upamo, da $O(\log n)$)

Rešitev

TEHNIKA: Če nekje malce popustimo, bomo morda drugje lahko veliko pridobili – *princip ravnoteženja*.
Kaj nam pa omejuje mejo, do katere lahko popustimo? Koliko lahko v tem primeru popustimo?

- naj bo zlivanje (in iskanje) malce počasnejše – $O(\text{višina kopice})$
- bo pa zato struktura bolj uravnotežena, kar bo razoroţilo Murphyja (nasprotnika, *adversary*) in bo zato tudi višina kopice manjša (upamo, da $O(\log n)$)
- *ideja*:
  - vrsta s prednostjo naj sestoji iz $O(\log n)$ podstruktur (kopic/dreves);
  - drevesa naj bodo različno velika;
  - a največja naj ne bo višje od $O(\log \frac{n}{2}) \Rightarrow$ nekako je v njem $\frac{n}{2}$ elementov;
  - vračamo se k eksplicitnim podatkovnim strukturam (reference)

Bitni register

Dvojiški zapis poljubnega števila: $1010110_2 = 86$.

```
public int Inc(BitRegister x) {
  i=0; c= 1;
  while (c > 0) {c= 0; if (x[i] == 1) c= 1;
    x[i]= 1 - x[i];
  }
}
```

Časovna zahtevnost: $O(\lg n)$ – v najslabšem primeru.

Bitni register

Dvojiški zapis poljubnega števila: $1010110_2 = 86$.

```
public int Inc(BitRegister x) {
  i=0; c= 1;
  while (c > 0) {c= 0; if (x[i] == 1) c= 1;
    x[i]= 1 - x[i];
  }
}
```

Časovna zahtevnost: $O(\lg n)$ – v najslabšem primeru.

```
x= 0;
for (i= 1, n) {
  Inc(x)
```

Časovna zahtevnost `Inc()`:

- v najslabšem primeru $O(\lg n)$
- amortizirano z uporabo *seštevalne metode* (*aggregate*): $O(1)$

Ideja zlivanja

- novo števko dodamo na levo, ko *velikost števila preraste* največje število, ki ga lahko predstavimo s prejšnjim številom števk; in
- dodana števka *omogoča zapis še enkrat več vrednosti*;
- število števk je $\lceil \lg n \rceil$

Ideja zlivanja – nadalj.

- seštevanje (zlivanje) dveh števil – vedno zlivamo istoležeči števki, ki pa predstavljata tudi enako veliki vrednosti:

$$\begin{array}{rr} & 1010110_2 \\ + & 110010_2 \\ = & 10001000_2 \end{array}$$

- čas potreben, da zlijemo dve števili, je sorazmeren:

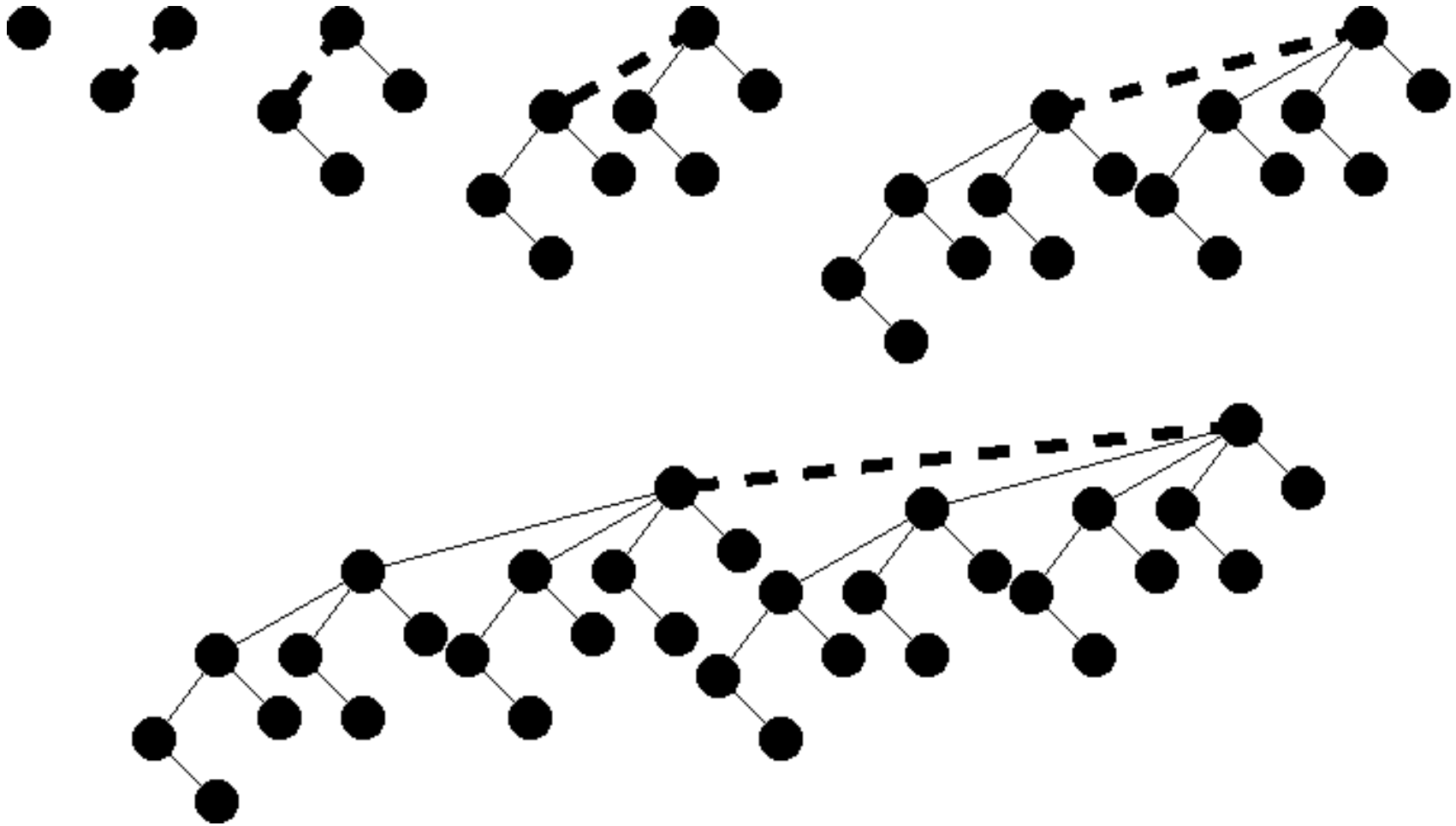
$$\begin{array}{rl} & (\text{število števk večjega števil}) \times (\text{čas seštevanja dveh števk}) \\ = & \lceil \lg n \rceil \times O(1) \\ = & O(\log n) \end{array}$$

Binomska drevesa

DEFINICIJA: binomsko drevo je rekurzivna podatkovna struktura, kjer velja:

- koren je najmanjši element v drevesu (urejenost);
- posamezen element je binomsko drevo B_0
- binomsko drevo B_k sestoji iz dveh poddreves B_{k-1}, pri čemer je:
  - manjši od korenov poddreves tudi koren celotnega drevesa in
  - poddrevo z večjim korenom prvi naslednik celotnega drevesa B_k

Primeri binomskih dreves



Lastnosti binomskega drevesa B_k

1. koren je najmanši element v drevesu [iz definicije]
2. koren ima k naslednikov, pri čemer je prvo poddrevo B_{k-1}, drugo B_{k-2} in tako naprej do k-tega, ki je B_0 [z indukcijo po k]
3. v drevesu je $n = 2^k$ elementov [z indukcijo po k]
4. višina drevesa je $k = \lg n$ [z indukcijo po k]
5. število vozlišč na globini i je $\binom{k}{i}$ [z indukcijo po k] – zato so to *binomska drevesa*
6. ker ≫zlivamo≪ samo enako velika drevesa, je struktura uravnotežena
7. zlivanje dveh dreves lahko naredimo v času $O(1)$

Dokažite zgornje lastnosti.

Binomska vrsta (s prednostjo)

Definicija:

- Binomska vrsta (kopica) sestoji iz *seznama* binomskih dreves B_0, B_1, ..., $B_{\lg(n/2)}$
- velikost zasedenega prostora je očitno $O(n)$

- POZOR: odnosi med koreni kosameznih binomskih dreves v kopici niso določeni!

Iskanje najmanjšega elementa

Ker gre za seznam, je sorazmerno dolžini seznama, ki pa je $O(\log n)$.

```
public int min(int x) {
  int min;
  binTree head;
  head= binHeap.head();
  tail= binHeap.tail();
  min= head.min();
  for (; tail != nil; head= tail.head(), tail= tail.tail())
    min= Min(min, head.min());
  return min;
}
```

Kako smo že to izboljševali?

Zlivanje

- zlijemo enako veliki drevesi
- to počnemo na podoben način kot smo pri polnem seštevalniku (*full adder*)
  - ena števka iz prvega števila, ena iz drugega ter prenos
  - rezultat je nova števka ter prenos
  - rezultat ima enako število števk kot večje od števil ali eno več
- Čas zlivanja je sorazmeren dolžini daljšega sezname – $O(\log n)$

Zapišite še kodo za metodo `merge` ter ostale operacije.

Osnovne operacije

dodajanje: isto kot zlivanje, le da zlivamo celo kopico z B_0

najmanjši: glej zgoraj

odreži: ko odrežemo koren binomskega drevesa, nasledniki tvorijo v resnici seznam binomskih dreves, kar je ponovno kopica. Zato lahko naredimo naslednje:

1. iz izvorne kopice H izločimo drevo, z najmanjšim korenom B_k in dobimo H' – $0(1)$
2. drevesu B_k odrežemo koren in iz naslednikov naredimo kopico H_k – $O(1)$
3. kopici H' in H_k zlijemo v rezultat – $O(\log n)$

Posplošena vrsta s prednostjo

spreminjanje: za zmanjševanje podobno kot pri dvojiški kopici element splava navzgor dokler je potrebno – $O(\log n)$. Za zvečevanje razmislite za domačo nalogo!

izločanje: naredimo v dveh korakih:

1. najprej zmanjšamo prednost na $-\infty$ – $O(\log n)$
2. nato odrežemo najmanjši element – $O(\log n)$

zlij: glej zgoraj

Posplošena vrsta s prednostjo

spreminjanje: za zmanjševanje podobno kot pri dvojiški kopici element splava navzgor dokler je potrebno – $O(\log n)$. Za zvečevanje razmislite za domačo nalogo!

izločanje: naredimo v dveh korakih:

1. najprej zmanjšamo prednost na $-\infty$ – $O(\log n)$
2. nato odrežemo najmanjši element – $O(\log n)$

zlij: glej zgoraj

sosed: (levi in desni) – zahteva iskanje po celi strukturi – $O(n)$

Zapišite še kodo za ostale operacije.

Zahtevnost

operacija	**seznam**	**drevo**	**kopica**	**binomska**
dodajanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
najmanjši	$O(1)$	$O(1)$	$O(1)$	$O(1)$
odreži	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Zahtevnost

operacija	**seznam**	**drevo**	**kopica**	**binomska**
dodajanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
najmanjši	$O(1)$	$O(1)$	$O(1)$	$O(1)$
odreži	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
izločanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
spreminjanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
zlij	$O(\mu)$	$O(\mu \log n)$	$O(n)$	$O(\log n)$

Zahtevnost

operacija	**seznam**	**drevo**	**kopica**	**binomska**
dodajanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
najmanjši	$O(1)$	$O(1)$	$O(1)$	$O(1)$
odreži	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
izločanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
spreminjanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
zlij	$O(\mu)$	$O(\mu \log n)$	$O(n)$	$O(\log n)$
levi sosed	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$
desni sosed	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$

Komentarji:

izločanje: poznamo referenco elementa

spreminjanje: čas brisanja + čas vstavljanja

zlij: $\mu = \min(|S_1|, |S_2|)$

V razpredelnici so asimptotične vrednosti, kaj pa v praksi? In velikost struktur?

Odvečno delo

Kaj se zgodi, če samo velikokrat:

- dodajamo element,
- izločamo element,
- zlivamo kopici

Pri tem opravljamo nepotrebno delo, ki skrbi za strukturo kopice.

Lena binomska kopica

- pustimo strukturo kopice vnemar, ampak samo ohranjanjamo (dvojno povezan) seznam (obroč / krog) dreves; in
- šele, ko je potrebno, popravimo strukturo (operacije nad najmanjšim elementom).

Vse operacije iz prve skupine sedaj opravimo v konstantnem času, medtem ko so druge operacije veliko dražje.

A izkaže se, da je cena drugih operacij sorazmerna (odvisna) od števila opravljenih prvih operacij v nizu $\rightarrow$ če pogledamo celoten niz operacij nismo nič izgubili.

To je dobro znana tehnika amortizacije.

Kaj pridobimo?

Amortizacija

Nekatere operacije so zelo hitre, druge pa zahtevajo več časa. Koliko časa pa zahtevajo vse operacije skupaj in koliko v povprečju posamezna operacija(v najslabšem primeru)?
Tehnike za amortizacijsko analizo:

- seštevalna metoda (*aggregate method*)
- računovodska metoda (*accounting method*)
- metoda potenciala (*potential method*)

Mi bomo za analizo uporabili metodo potenciala.

Amortizacijska analiza

Operacije pretvorijo H v H':

- Potencial podatkovne strukture $\Phi(H)$ je nenegativna vrednost.
- Amortizirana cena operacije $\hat{c}$ je definirana kot

$$\hat{c} = c + (\Phi(H') - \Phi(H)) \ ,$$

  kjer je c dejanska cena ter sta $\Phi(H)$ in $\Phi(H')$ potenciala pred in po operaciji.
- Vse opearacije morajo ohranjati potencial nenegativen, ali poenostavljeno $\Phi(H') - \Phi(H)$ ne sme biti negativna.

Amortizacijska analiza binomske vrste

Definirajmo kot potencial podatkovne strukture število binomskih dreves $\Phi(H) = t(H)$

Amortizacijska analiza binomske vrste

Definirajmo kot potencial podatkovne strukture število binomskih dreves $\Phi(H) = t(H)$

Operacije, ki nas zanimajo:

`Min` : $O(1)$, saj samo vrnemo staro vrednost, medtem ko $\Phi(H') - \Phi(H) = 0$

Amortizacijska analiza binomske vrste

Definirajmo kot potencial podatkovne strukture število binomskih dreves $\Phi(H) = t(H)$

Operacije, ki nas zanimajo:

`Min` : $O(1)$, saj samo vrnemo staro vrednost, medtem ko $\Phi(H') - \Phi(H) = 0$

`Insert` : $O(1)$, ker samo dodamo v povezan seznam in $\Phi(H') - \Phi(H) = t(H') - t(H) = (t(H) + 1) - t(H) = 1$

Amortizacijska analiza binomske vrste

Definirajmo kot potencial podatkovne strukture število binomskih dreves $\Phi(H) = t(H)$

Operacije, ki nas zanimajo:

`Min` : $O(1)$, saj samo vrnemo staro vrednost, medtem ko $\Phi(H') - \Phi(H) = 0$

`Insert` : $O(1)$, ker samo dodamo v povezan seznam in $\Phi(H') - \Phi(H) = t(H') - t(H) = (t(H) + 1) - t(H) = 1$

`Merge` : $O(1)$, ker povežemo dva povezana seznama, postavimo minimum obeh kopic in $\Phi(H') - (\Phi(H_1) + \Phi(H_2)) = t(H') - (t(H_1) + t(H_2)) = 0$, saj je število dreves enako vsoti številu dreves v zlitih kopicah.

`DelMin` : $O(\log n)$, ker

$$\Phi(H') - \Phi(H) = (t(H) - 1 + D(H)) - (H) = D(H) - 1 \ ,$$

kjer je $D(H)$ največje število naslednikov korena in je $\lg n$.

Fibonaccijeva kopica

Zelo preprosto povedano je to binomska kopica z amortiziranimi operacijami ter rahlo spremenjenima funkcijama `Delete` in `DecreaseKey`, kar zahteva spremenjeno definicijo potenciala.

Pri analizi nastopajo Fibonaccijeva stevila in zato Fibonaccijeva kopica.

Doslej smo se pri analizi časovne zahtevnosti vedno spraševali, kakšen je čas *ene operacije v najslabšem primeru*. Sedaj se sprašujemo, kako slabo je lahko izvajanje *niza operacij v najslabšem primeru*.

To, da druga vrednost ni slabša kot vsota prvih vrednosti je očitno. Pa je lahko boljša?

Razmislite!

V knjigi preberite poglavje o amortizirani analizi Fibonaccijevih kopic.

Zahtevnost

operacija	**seznam**	**drevo**	**kopica**	**binomska**	**Fibonaccijeva**
dodajanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
najmanjši	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
odreži	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
izločanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^{\dagger}$
spreminjanje	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^{\dagger}$
zlij	$O(\mu)$	$O(\mu \log n)$	$O(n)$	$O(\log n)$	$O(1)$
levi sosed	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
desni sosed	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$

Komentarji:

izločanje: poznamo referenco elementa

spreminjanje: čas brisanja + čas vstavljanja

zlij: $\mu = \min(|S_1|, |S_2|)$

†: če ne popravljamo najmanjšega elementa je $O(1)$