Podatkovne strukture in algoritmi

Andrej Brodnik
UP FAMNIT

Slovar

posplošena drevesa, B drevesa,
2-3 drevesa, R-Č drevesa

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.

Drevesa

Pri dvojiških drevesih smo imeli *koren* in dve *poddrevesi*:

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

Pri urejenih/iskalnih drevesih dodatno velja:

$$\texttt{levo} < \texttt{koren} < \texttt{desno} \ .$$

Posplošimo definicijo na več poddreves.

Večsmerno drevo

Naj bo k neka *konstanta*, potem je k-tiško drevo definirino kot:

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

Velikost

V k-tiškem drevesu višine h je največ

$$(k-1) + (k-1)k + \cdots + (k-1)k^{h-1} = \frac{(k-1)k^h - 1}{k-1} = k^h$$

elementov.
Drugače povedano, k-tiško drevo z n elementi je visoko vsaj

$$h \geq \log_k n = \frac{\lg n}{\lg k} \quad .$$

Kako smo prišli do zadnjega ulomka?

Urejeno k-tiško drevo

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

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

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

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/dostopi (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:

$$\texttt{Podatek data}[0..B-1]$$

Vozlišče k-tiškega drevesa je veliko največ B.

B-drevo

Za *B-drevo reda* $b \geq 2$ velja:

vozlišče - ključi vozlišče v ima k_v ključev `v.key[i]` $(0 \leq i < k_v)$, kjer $\lceil b/2 \rceil - 1 < k_v \leq b$, razen korena, ki ima lahko tudi samo en ključ;

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 za vozlišče v velja `v.sub[i]` $<$ `v.key[i]` $<$ `v.sub[i + 1]` za $0 \leq i < k_v$;

listi vsi listi so na isti globini h.

- Prvi dve vrstici zagotavljata *eksponentno povečevanje* elementov na posameznem nivoju in posledično *logaritemsko globino*;
- tretja vrstica omogoča iskanje po principu *deli in vladaj*; in
- zadnja vrsta zagotavlja *uravnoteženost*, ki zagotavlja enak čas pri operacijah nad katerimkoli podatkom v drevesu.

Vozlišče 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
  ...
}
```

Lastnosti

- Najmanjši b, za katerega je definicija smiselna, je $b = 4$. Tedaj ima vozlišče v

$$\lceil b/2 \rceil - 1 = 1 \leq k_v < b = 4$$

  ključev, oziroma najmanj dve in največ štiri 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 h največ

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

Primer

Primer — naj bo:

- $B = 4\text{kB}$ – običajna velikost prebranega bloka z diska;
- velikost elementa 32B, od česar naj bo 6B ključ (dovolj za EMŠO) in 26B podatkov;
- referenca na poddrevo naj bo 8B (64 bitov);
- za zapis k potrebujemo 2B.

V velikost B spravimo vozlišče B-drevesa, če $b = 102$.

- Za $n = 2.066.880$ (število prebivalcev Slovenije na 1.1.2018) je $h \leq 5$. To pomeni, da bomo pri iskanju ključa pregledali kvečjemu pet vozlišč – drugače, *dostopili* bomo samo do petih vozlišč!
- Za $n = 7.661.384.500$ (približno število prebivalcev na svetu) je $h \leq 7$. To pomeni, da bomo pri iskanju ključa pregledali kvečjemu sedem vozlišč – drugače, *dostopili* bomo samo do sedmih vozlišč!

B+ drevesa

- Na B ne moremo vplivati: definira računalniška arhitektura, oziroma pomnilniška hierarhija – npr. dolžina predpomnilniške vrstice, velikost sektorja na disku, velikost ethernet paketa ipd..
- b lahko povečamo, če v vozlišča ne dajemo celotnih elementov, ampak samo njihove ključe, medtem ko podatke hranimo v listih – govorimo o $B+$ drevesih:

```
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?)
  ...
}
```

Primer – nadaljevanje

- $B = 4$kB – običajna velikost prebranega bloka z diska;
- velikost ključa 6B (dovolj za EMŠO);
- referenca na poddrevo naj bo 8B (64 bitov);
- za zapis k potrebujemo 2B.

V velikost B spravimo vozlišče B-drevesa, če $b = 291$.

- Za $n = 2.066.880$ (število prebivalcev Slovenije na 1.1.2018) je $h \leq 4$ (bilo 5).
- Za $n = 7.661.384.500$ (približno število prebivalcev na svetu) je $h \leq 6$ (bilo 7).

opomba: Podatki so relativno majhni, samo 26B v primerjavi s 6B ključa.

Iskanje

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

```
data 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:

dostopov: $h = O(\log_b n)$;

primerjav: $bh = b \log_b n = \frac{b}{\lg b} \lg n$ (v vozlišču lahko samo $b/2$ ključev in v korenu samo 2).

Iskanje – analiza

Časovna zahtevnost je:

dostopov: $h = O(\log_b n)$;

primerjav: $bh = b \log_b n = \frac{b}{\lg b} \lg n$ (v vozlišču lahko samo $b/2$ ključev in v korenu samo 2).

Če bi namesto linearnega iskanja po vozlišču uporabili razpolavljanje, bi se število primerjav zmanjšalo na

$$(\lg b)h = \lg b \log_b n = \frac{\lg b}{\lg b} \lg n = \lg n$$

primerjav, kar je primerljivo z dvojiškimi drevesi.

Vstavljanje

- Novi element `Elt` vedno vstavimo v list – do tam se sprehodimo na enak način kot pri iskanju.

Vstavljanje

- Novi element `Elt` vedno vstavimo v list – do tam se sprehodimo na enak način kot pri iskanju.
- Če je v vozlišču v manj kot b elementov, element `Elt` vstavimo tako, da se ohrani naraščajoče zaporedje ključev. Zaključimo preprosto vstavljanje.

Vstavljanje

- Novi element `Elt` vedno vstavimo v list – do tam se sprehodimo na enak način kot pri iskanju.
- Če je v vozlišču v manj kot b elementov, element `Elt` vstavimo tako, da se ohrani naraščajoče zaporedje ključev. Zaključimo preprosto vstavljanje.
- Če je vozlišču v z vstavljanim elementom $b + 1$ element, jih uredimo in razdelimo na tri dele:
  - prvih $\lceil b/2 \rceil - 1$ elementov $\Rightarrow$ vozlišče v,

Vstavljanje

- Novi element `Elt` vedno vstavimo v list – do tam se sprehodimo na enak način kot pri iskanju.
- Če je v vozlišču v manj kot b elementov, element `Elt` vstavimo tako, da se ohrani naraščajoče zaporedje ključev. Zaključimo preprosto vstavljanje.
- Če je vozlišču v z vstavljanim elementom $b + 1$ element, jih uredimo in razdelimo na tri dele:
  - prvih $\lceil b/2 \rceil - 1$ elementov $\Rightarrow$ vozlišče v,
  - element r in

Vstavljanje

- Novi element `Elt` vedno vstavimo v list – do tam se sprehodimo na enak način kot pri iskanju.
- Če je v vozlišču v manj kot b elementov, element `Elt` vstavimo tako, da se ohrani naraščajoče zaporedje ključev. Zaključimo preprosto vstavljanje.
- Če je vozlišču v z vstavljanim elementom $b + 1$ element, jih uredimo in razdelimo na tri dele:
  - prvih $\lceil b/2 \rceil - 1$ elementov $\Rightarrow$ vozlišče v,
  - element r in
  - preostali elementi $\Rightarrow$ vozlišče v_n

Vstavljanje

- Novi element `Elt` vedno vstavimo v list – do tam se sprehodimo na enak način kot pri iskanju.
- Če je v vozlišču v manj kot b elementov, element `Elt` vstavimo tako, da se ohrani naraščajoče zaporedje ključev. Zaključimo preprosto vstavljanje.
- Če je vozlišču v z vstavljanim elementom $b+1$ element, jih uredimo in razdelimo na tri dele:
  - prvih $\lceil b/2 \rceil - 1$ elementov $\Rightarrow$ vozlišče v,
  - element r in
  - preostali elementi $\Rightarrow$ vozlišče v_n

  in vrnemo staršu (r, v_n).

Vstavljanje – popravljanje

- Če starš obstaja, vstavi r in v_n tako, da se ohranja urejenost. Če je v staršu še bil prostor zaključimo, sicer ponovno razdelimo elemente in nadaljujemo pri staršu.

Vstavljanje – popravljanje

- Če starš obstaja, vstavi r in v_n tako, da se ohranja urejenost. Če je v staršu še bil prostor zaključimo, sicer ponovno razdelimo elemente in nadaljujemo pri staršu.
- Če pa starša ni, naredimo nov koren v, r, v_n.

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

Brisanje

- 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?
- karkoli že naredimo, ponavljamo rekurzivno do korena, ki lahko, izgine

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	$O(\log n)$	$O(\log n)$	$O(\log n)$
B drevo	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$

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	$O(\log n)$	$O(\log n)$	$O(\log n)$
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 ≫robnem stanju≪.

Dvojiška (AVL) in B-drevesa

- V obeh primerih mora struktura imeti:
  1. *urejenost*, kar omogoča iskanje (in ostale operacije) po načelu ≫deli in vladaj≪ (lokalnost) in s tem pri iskanju ne zahteva iskanja po celi strukturi; in
  2. *uravnoteženost*, kar omejuje najslabši čas operacij
- pri 1 podobno obe vrsti dreves,
- pri 2 sta različni rešitvi:
  - pri dvojiških drevesih dodajamo vedno nova vozlišča pri listih ter nato uporabimo vrtenja, da drevo uravnotežimo;
  - pri B-drevesih pa dodajamo nova vozlišča pri korenu, kar ohranja liste na isti globini in drevo uravnoteženo.
- prva rešitev (programsko) izgleda precej bolj zapletena kot druga
- ali lahko drugo rešitev uporabimo pri dvojiških drevesih?

2-3-4 (TTF) drevesa

Vzemimo B-drevesa z redom $b = 3$:

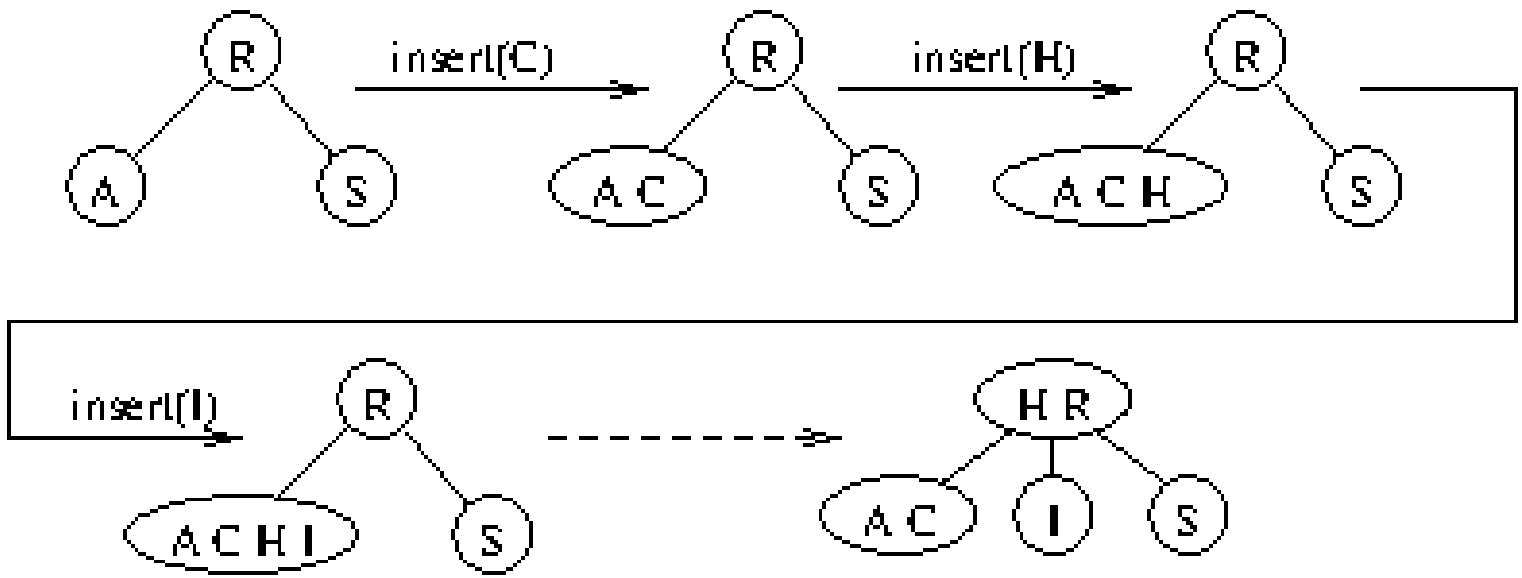
- imajo v vozlišču do 3 elemente (tudi enega);
- 2, 3 ali 4 naslednike (*2-3-4 drevesa* TTF drevesa) in
- v drevesu višine h je *najmanj* 2^h elementov in če obrnemo, je drevo, ki ima n elementov visoko največ $\lg n$.

Operacije nad TTF drevesi

iskanje enako kot pri B-drevesih, ker so drevesa uravnožena in urejena – časovna zahtevnost $O(\log n)$ *primerjav*

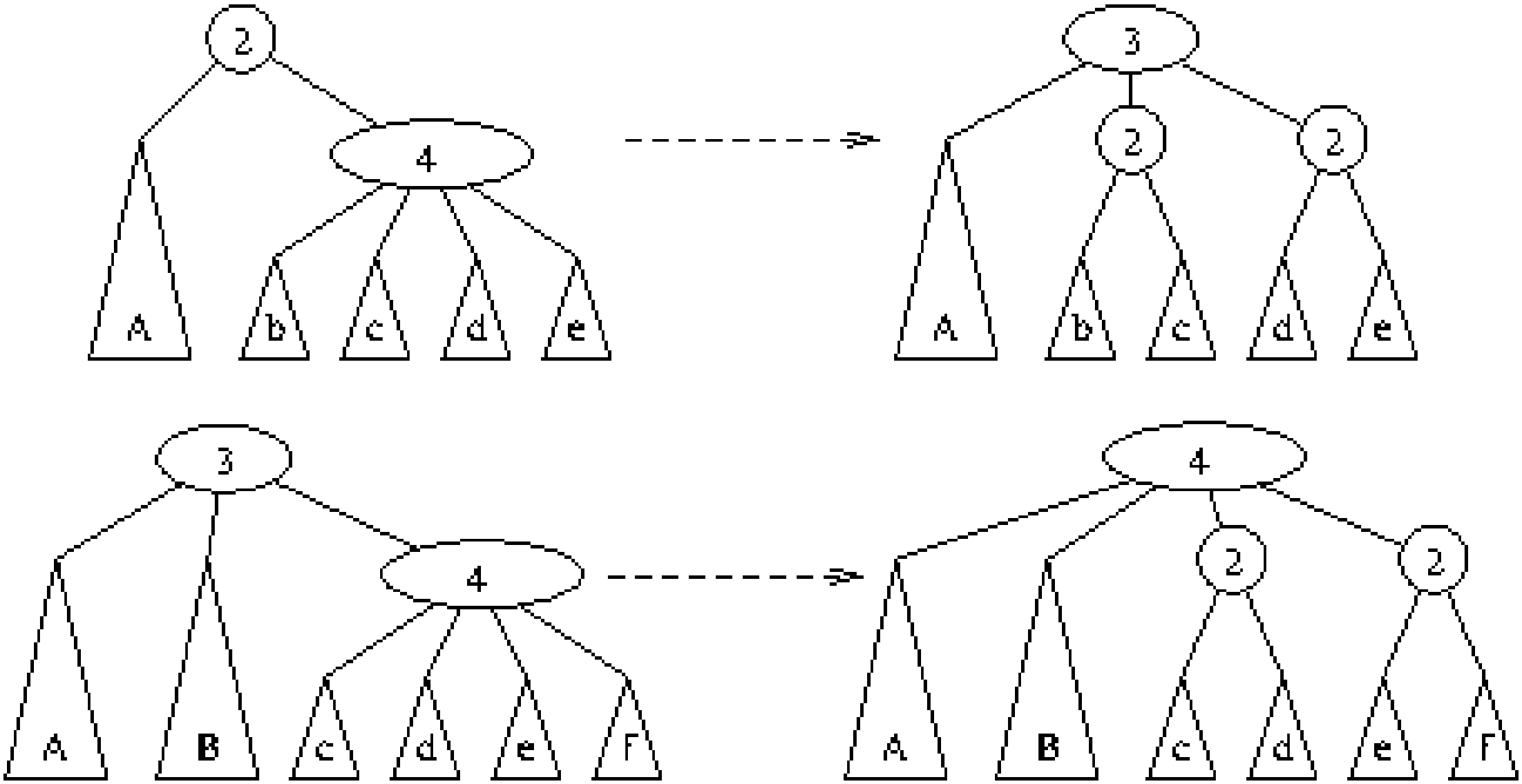
vstavljanje pričnemo z (neuspešnim) iskanjem in pridemo do lista nato imamo dve možnosti:

- prišli smo do 2 ali 3 lista: preprosto vstavimo novi element
- prišli smo do 4 lista, ki ga razcepimo in popravljamo vozlišča do korena.

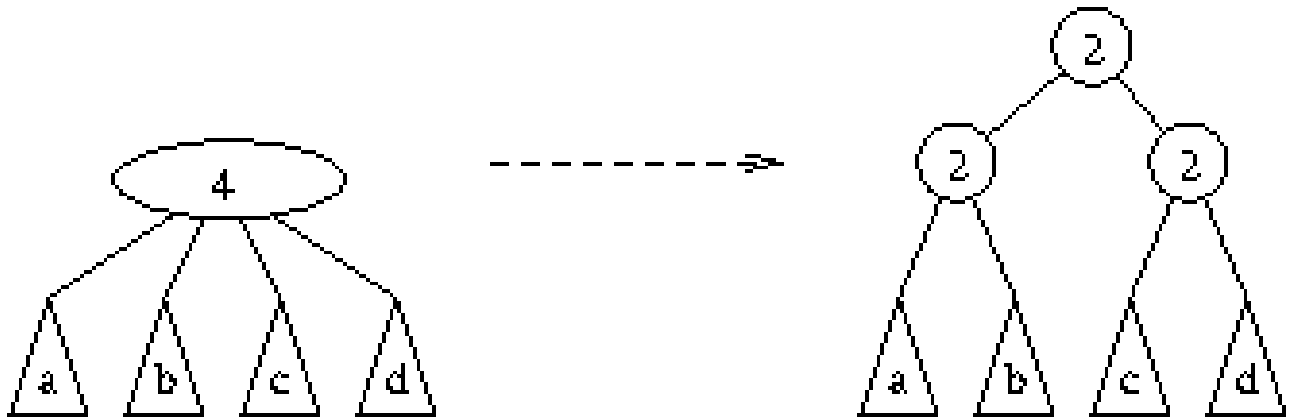


Operacije nad TTF drevesi – nadalj.

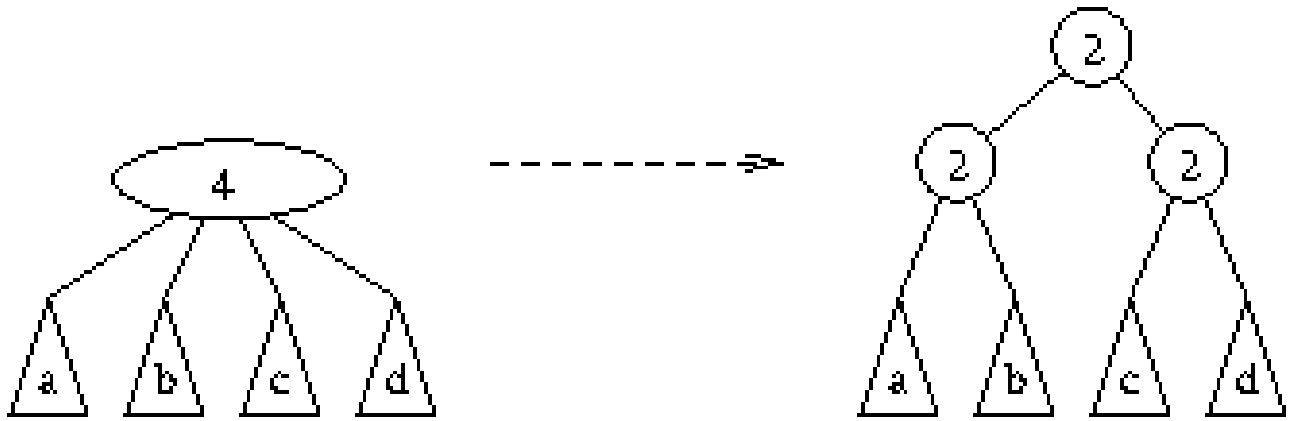
Drugi možnosti se izognemo tako, da zagotovimo, da list nikoli ne bo 4-vozlišče in na ta način nikoli ne dobimo na dnu 4-vozlišča ter nam ni potrebno cepiti staršev.



Operacije nad TTF drevesi – nadalj.

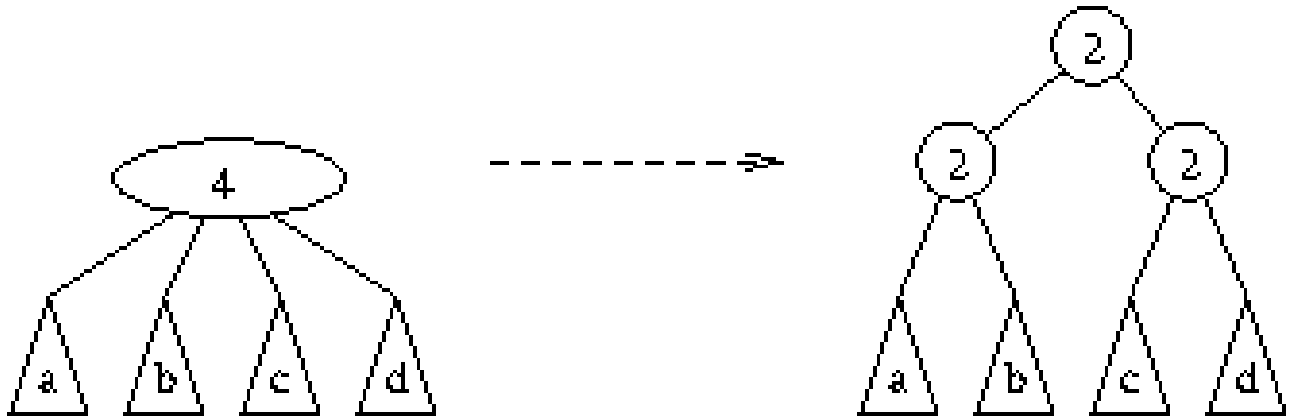


Operacije nad TTF drevesi – nadalj.



izločanje tudi enako kot pri B-drevesih – časovna zahtevnost $O(\log n)$ *primerjav*

Operacije nad TTF drevesi – nadalj.



izločanje tudi enako kot pri B-drevesih – časovna zahtevnost $O(\log n)$ *primerjav*

Tvorimo drevo:

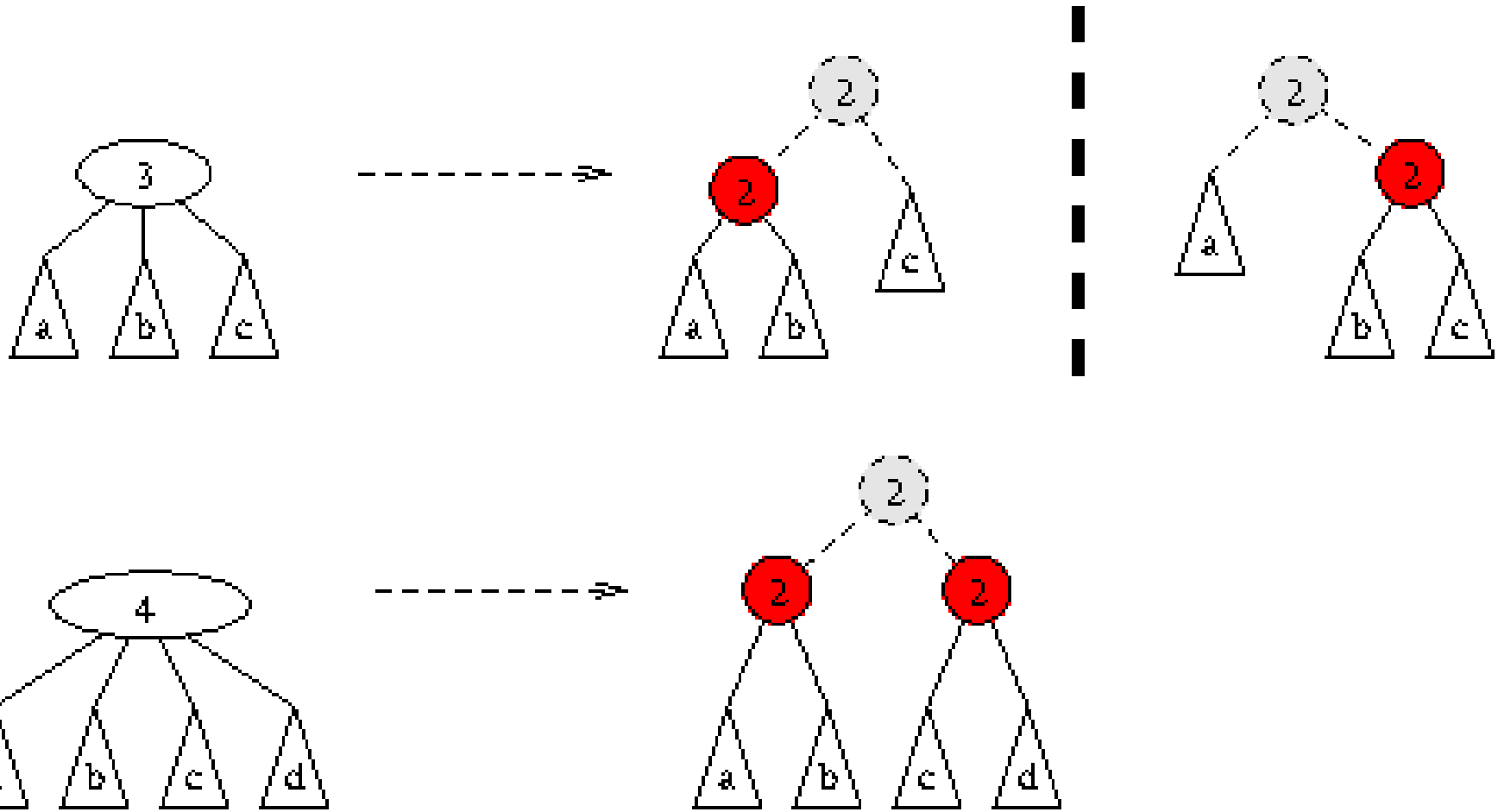
A S E A R C H I N G

Rdeče črna drevesa – *red-black trees*

- B-drevesa so zelo koristna, če štejemo število dostopov, če pa štejemo število primerjav, se ne obnašajo nič bolje od dvojiških dreves
- zato preoblikujmo naša TTF drevesa v običajna dvojiška drevesa. Posamezna vozlišča preoblikujemo (preslikamo) v dvojiška vozlišča.

Rdeče črna drevesa – nadalj.

- 2-vozlišča se preprosto preslikajo v dvojiška vozlišča
- 3-vozlišča se preslikajo v eno od obeh možnosti (a konsistentno uporabljajmo eno, vseeno katero)
- 4-vozlišča se preslikajo v tri 2-vozlišča



Lastnosti

Kako dokazati naslednje trditve:

- vsi listi (neobstoječa vozlišča – `null`) so črni;

Lastnosti

Kako dokazati naslednje trditve:

- vsi listi (neobstoječa vozlišca – `null`) so črni;
- če je vozlišče rdeče, potem sta oba naslednika črna;

Lastnosti

Kako dokazati naslednje trditve:

- vsi listi (neobstoječa vozlišca – `null`) so črni;
- če je vozlišče rdeče, potem sta oba naslednika črna;
- vzemimo poljubno (notranje) vozlišče v in vse liste l_i v poddrevesu, kateremu je v koren; potem je število črnih vozlišč na poti od v do lista l_i enako za vse liste poddrevesa;

Lastnosti

Kako dokazati naslednje trditve:

- vsi listi (neobstoječa vozlišca – `null`) so črni;
- če je vozlišče rdeče, potem sta oba naslednika črna;
- vzemimo poljubno (notranje) vozlišče v in vse liste l_i v poddrevesu, kateremu je v koren; potem je število črnih vozlišč na poti od v do lista l_i enako za vse liste poddrevesa;
- višina drevesa je največ 2-kratna višina FFT drevesa in zato je še vedno $O(\log n)$;

Lastnosti

Kako dokazati naslednje trditve:

- vsi listi (neobstoječa vozlišca – `null`) so črni;
- če je vozlišče rdeče, potem sta oba naslednika črna;
- vzemimo poljubno (notranje) vozlišče v in vse liste l_i v poddrevesu, kateremu je v koren; potem je število črnih vozlišč na poti od v do lista l_i enako za vse liste poddrevesa;
- višina drevesa je največ 2-kratna višina FFT drevesa in zato je še vedno $O(\log n)$;
- posledično vse operacije imajo zahtevnost $O(\log n)$ *primerjav* (in *dostopov*).

Lastnosti

Kako dokazati naslednje trditve:

- vsi listi (neobstoječa vozlišca – `null`) so črni;
- če je vozlišče rdeče, potem sta oba naslednika črna;
- vzemimo poljubno (notranje) vozlišče v in vse liste l_i v poddrevesu, kateremu je v koren; potem je število črnih vozlišč na poti od v do lista l_i enako za vse liste poddrevesa;
- višina drevesa je največ 2-kratna višina FFT drevesa in zato je še vedno $O(\log n)$;
- posledično vse operacije imajo zahtevnost $O(\log n)$ *primerjav* (in *dostopov*).

Preglejte podrobnosti operacij, ki slone samo na zgornjih lastnostih in ne na TTF drevesih.

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	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
B drevo	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$
RB-drevo	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

- Kakšna je prostorska zahtevnost?
- In koliko je primerjav? Kako velika je konstanta skrita v $O()$?