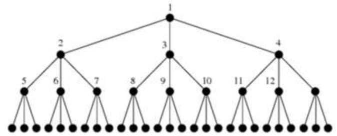
Abstraktni podatkovni tipi in podatkovne strukture

Drevesa: dvojiško, iskalno, uravnoteženo, AVL, večsmerno, k-tiško, B, B+, TTF, rdeče-črno

Tomaž Dobravec, Algoritmi in podatkovne strukture 2

Drevo

Drevo



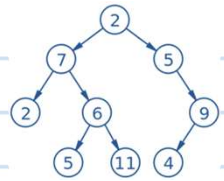
Drevo je abstraktna podatkovna struktura, ki posnema hierarhično drevesno strukturo, v kateri imamo koren (element na najvišjem nivoju) in poddrevesa (ki so hierarhično na nižjem nivoju kot koren). Drevo je predstavljeno kot množica med seboj povezanih elementov.

Uporabljali bomo naslednje izraze:

- ✧ elementom drevesa pravimo __________;
- ✧ __________ drevesa je element, ki je na najvišjem nivoju;
- ✧ korenom poddreves pravimo __________ korena; koren je __________ naslednikom; koren je edino vozlišče brez staršev;
- ✧ nasledniki na istem nivoju (imajo istega starša) so __________;
- ✧ vozlišča, ki nimajo poddreves, imenujemo __________, ostala imenujemo __________;
- ✧ **globina vozlišča** je ____
- ✧ **globina (višina)** je ____
- ✧ **k-tiško drevo** ____
- ✧ **popolno k-tiško** drevo____

- ✧ Kakšna je razlika med drevesom in grafom?

Dvojiško drevo



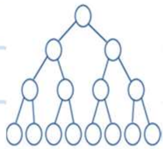
Dvojiško drevo je drevo, v katerem ima vsako vozlišče 0, 1 ali 2 naslednika.

✧ Maksimalna globina dvojiškega drevesa z n vozlišči? ______________________

✧ Minimalna globina dvojiškega drevesa z n vozlišči? ______________________

Popolno dvojiško drevo je dvojiško drevo, v katerem ima vsako vozlišče (razen listov) dva naslednika.

- število notranjih vozlišč: ___________
- število listov: ___________
- skupno število vseh vozlišč: ___________



Implementacija dvojiškega drevesa:

Pregledovanje dvojiškega drevesa

Dvojiška drevesa lahko pregledamo na (vsaj) tri načine:

- ✧ **premi** (preorder) pregled najprej ”obdela” koren, nato levo poddrevo in na koncu desno poddrevo;
- ✧ **vmesni** (inorder) pregled najprej ”obdela” levo poddrevo, nato koren in na koncu desno poddrevo;
- ✧ **obratni** (postorder) najprej obdela levo, nato desno poddrevo in na koncu koren.

Primer implementacije vmesnega pregleda:

```
static void vmesniPregled(Drevo d) {
   if (d != null) {
      vmesniPregled(d.levi);
      System.out.println(d.elt);
      vmesniPregled(d.desni);
   }
}
```

Urejena (iskalna) drevesa

- Če so elementi v drevesu med seboj **primerljivi**, jih lahko shranimo v drevesni strukturi v izbranem vrstnem redu (po nekem pravilu glede na urejenost).

- Elemente tipa `Elt` bomo med seboj primerjali po ključu

- Implementacija v Javi?

Urejeno drevo je drevo, za katero velja:
  - vsi elementi v levem poddrevesu so manjši (<) od korena
  - vsi elementi v desnem poddrevesu so večji (>) od korena

- **Operacija `insert()`**
  - element vedno dodamo v list
  - pri iskanju mesta, upoštevamo pravilo urejenosti: element najprej primerjamo s korenom; če je koren večji od elementa, element (rekurzivno) vstavimo v levo, sicer v desno poddrevo.

Urejena (iskalna) drevesa - `insert()`

- Psevdokoda za operacijo insert()

- Kaj je **najslabši** primer za vstavljanje?

Časovna zahtevnost operacije `insert()`?

- **Opomba**: v urejenem drevesu vmesniPregled() izpiše elemente v urejenem vrstnem redu!

Urejena (iskalna) drevesa - `find()`

- **Operacija find()**
  - sprehodimo se po drevesu, dokler ne najdemo elementa, ki ga iščemo
  - če iščemo v praznem drevesu (drevo == []), potem iskanje ni uspešno, sicer
    - če je element enak korenu -> iskanje je uspešno, končamo
    - sicer: če je element manjši od korena, iskanje (rekurzivno) nadaljujemo v levem, sicer v desnem poddrevesu

- Psevdokoda za operacijo `find()`:

Časovna zahtevnost operacije `find()` ?

Urejena (iskalna) drevesa - `delete()`

- **Operacija delete()**
  - Brisanje elementov v urejenem drevesu je bolj zapleteno, saj moramo ohraniti strukturo urejenosti!
  - Pri brisanju vozlišča `v` imam tri možnosti in sicer: vozlišče `v` a) nima naslednikov, b) ima enega naslednika in c) ima dva naslednika

Časovna zahtevnost operacije delete()?

Urejena (iskalna) drevesa - problem podvojenih vrednosti

- V osnovni definiciji slovarja **ne dovolimo podvajanja** elementov. Kaj pa, če bi v urejenem drevesu kljub vsemu to želeli imeti?

  Primer uporabe: urejeno drevo uporabljamo za shranjevanje števil (ključ je število, vrednost pa je null)

- Če bi dovolili dvojnike, bi bilo treba redefinirati pomen operacij find() in delete(); obe sedaj delata tako, da poiščeta in pobrišeta PRVI element; morda je to OK, morda pa ne! Treba se je dogovoriti!
- Če bi dovolili dvojnike, moramo popraviti definicijo urejenega drevesa, pri tem imamo dve možnosti: a) duplikat pišemo (na primer) vedno v desno vejo ali b) dodatni atribut za štetje

Ka) dovolimo vstavljanje in sicer tako, da v zgornji definiciji namesto > pišemo ≥ (duplikati elementa so v desni veji).

KB) namesto dvojnega vstavljanja samo štejemo število elementov z dvojnim ključem (to pride v poštev, kadar je ključ edina informacija o elementu).

Časovna zahtevnost

	`find()`	`insert()`	`delete()`
dinamična tabela	O(n)	O(n)	O(n)
urejena tabela	O(log n)	O(n)	O(n)
seznam	O(n)	O(1)	O(n)
urejen seznam	O(n)	O(n)	O(n)
preskočni seznam	O(lg(n))	O(lg(n))	O(lg(n))
dvojiško iskalno drevo	O(h)=O(n) $\Omega(\lg n)$*	O(h)=O(n) $\Omega(\lg n)$	O(h)=O(n) $\Omega(\lg n)$

*v optimalnem primeru se da doseči, da je drevo globoko lg(n); v tem primeru so vse tri operacije zelo hitre

Uravnoteženo drevo

Uravnoteženo iskalno drevo

Težava: pri iskalnih drevesih lahko pride do težave, da se velik del elementov skoncentrira v eni od vej; v najslabšem primeru dobimo verigo (vsi elementi v skrajno levi veji ali v skrajno desni veji).

Posledica: vse operacije v iskalnem drevesu so O(n).

Rešitev?

Definicija: Drevo je **uravnoteženo**, če obstaja taka konstanta $m>0$,
da za vsako vozlišče v velja: globina levega in desnega poddrevesa v
se razlikujeta kvečjemu za m.

Definicija: faktor uravnoteženosti (balance factor) v dvojiškem drevesu je razlika v globini levega in desnega poddrevesa: f = levi.visina – desni.visina

AVL drevo

AVL drevo

Definicija: Uravnotežena drevesa pri m=1 se imenujejo **AVL** drevesa.

Lastnosti AVL drevesa:

- ✧ AVL drevo velja za najstarejšo uravnoteženo podatkovno strukturo
- ✧ zaradi uravnoteženosti se AVL drevo nikoli ne izrodi,
- ✧ **višina AVL drevesa** je vedno logaritmična (glede na n), torej

$$h=\Theta(\log n);$$

- ✧ za faktor uravnoteženosti v AVL drevesu zaradi zahteve o uravnoteženosti vedno velja:

Implementacija AVL drevesa?

Kako lahko vzdržujemo uravnoteženost drevesa pri operacijah `insert()` in `delete()`?

Rotacija drevesa

- Rotacija je operacija, ki preoblikuje strukturo drevesa tako, da spremeni nivoje nekaterih vozlišč.
- Rotacija ohrani urejenost drevesa.

- Rotacija je obrnljiva operacija → učinek desne rotacije lahko "izničimo" s primerno levo rotacijo (`leva(desna(T))=T`) in obratno (`desna(leva(T))=T`)

Rotacija drevesa - primer

AVL drevo – operacija `insert()`

Operacija `insert()`: poskrbeti je treba, da bo drevo po vstavljanju ostalo uravnoteženo:

- najprej vstavimo (enako, kot smo vstavili v običajno iskalno drevo),
- nato popravimo uravnoteženost (rotacije).

Psevdokoda za operacijo `insert()`:

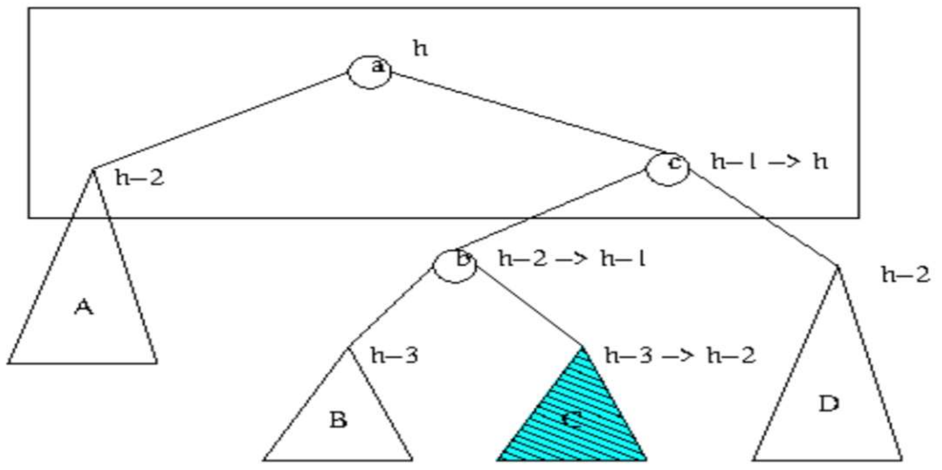
```
insert(w):
    - vstavi w kot v dvojiško iskalno drevo
    - "sprehodi" se od w proti korenu drevesa in poišči prvo neuravnoteženo
      vozlišče; to vozlišče označi z z, njegovega naslednika z y in naslednika
      od y (na poti proti w) z x
    - z-drevo je lahko neuravnoteženo na enega od 4 načinov:

        a) y je levi  naslednik od z, x je levi  naslednik od y (LL)
        b) y je levi  naslednik od z, x je desni naslednik od y (LD)
        c) y je desni naslednik od z, x je desni naslednik od y (DD)
        d) y je desni naslednik od z, x je levi  naslednik od y (DL)
```

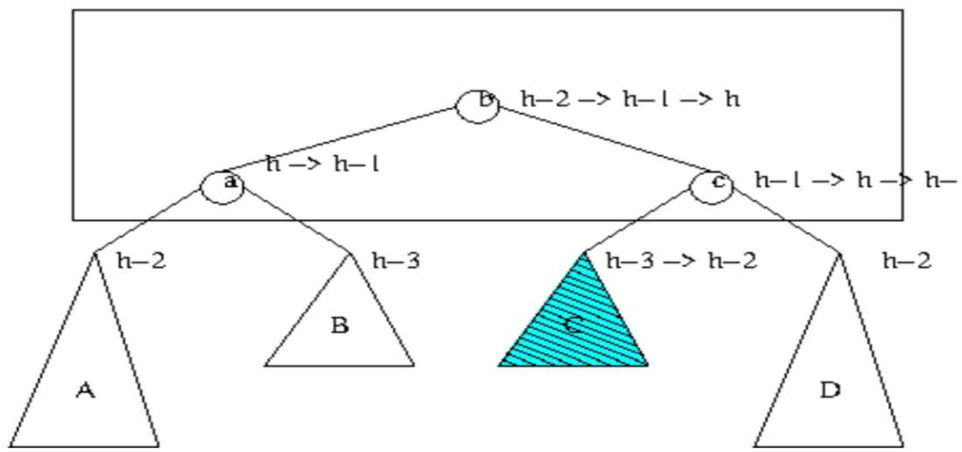
Časovna zahtevnost operacije `insert()` ?

reševanje LL, LD, DD in DL situacije

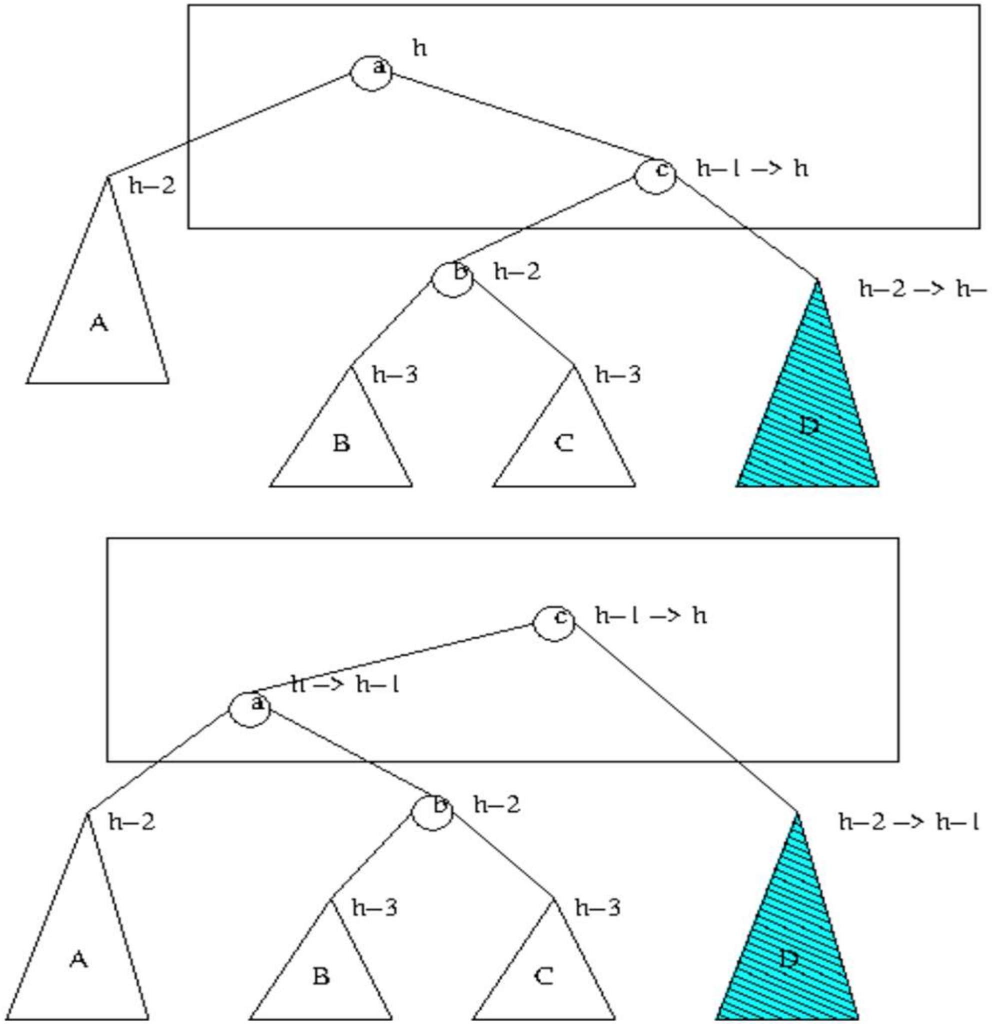
AVL drevo – primer vstavljanja

1/3


a
h
h–2
c
h–1 –> h
b
h–2 –> h–1
h–2
A
h–3
h–3 –> h–2
B
C
D


b
h–2 –> h–1 –> h
a
h –> h–1
c
h–1 –> h –> h–
h–2
h–3
h–3 –> h–2
h–2
B
C
A
D

AVL drevo – primer vstavljanja



AVL drevo – operacija `delete()`

<u>Operacija `delete()`: najprej JE TREBA pobrisati, nato popraviti uravnoteženost.</u>

- Brišemo na enak način kot pri dvojiškem iskalnem drevesu: na mesto izbrisanega elementa vstavimo največji element leve podveje.
- Po brisanju elementa iz podveje se morda poruši uravnoteženost ($|f| = 2$) → opravim rotacije.

<u>Psevdokoda za operacijo `delete()`:</u>

`delete(w):`

- `pobrišem element` **`w`** `(kot v BST), nato pa se od mesta, kjer se je w nahajal, sprehodim proti korenu in iščem prvo vozlišče, v katerem je zaradi brisanja prišlo do neuravnoteženosti; to vozlišče označim z A (A je "problematično" vozlišče)`
- `ločim dva primera: w se je nahajal v (A) desnem oziroma v (B) levem poddrevesu drevesa A`

→

AVL drevo – operacija `delete()`

```
(A) w se nahaja v desnem poddrevesu A
```

```
(B) w se nahaja v levem poddrevesu A
```

Časovna zahtevnost operacije `delete()` ?

Časovna zahtevnost

	find()	insert()	delete()
dinamična tabela	O(n)	O(n)	O(n)
urejena tabela	O(log n)	O(n)	O(n)
seznam	O(n)	O(1)	O(n)
urejen seznam	O(n)	O(n)	O(n)
preskočni seznam	O(lg(n))* *pričakovani čas	O(lg(n))* *pričakovani čas	O(lg(n))* *pričakovani čas
dvojiško iskalno drevo	O(h)=O(n)	O(h)=O(n)	O(h)=O(n)
AVL drevo	O(h)=O(lg n)	O(h)=O(lg n)	O(h)=O(lg n)

AVL drevo je prva od omenjenih struktur, pri kateri vse operacije potekajo (v najslabšem primeru) v logaritmičnem času!

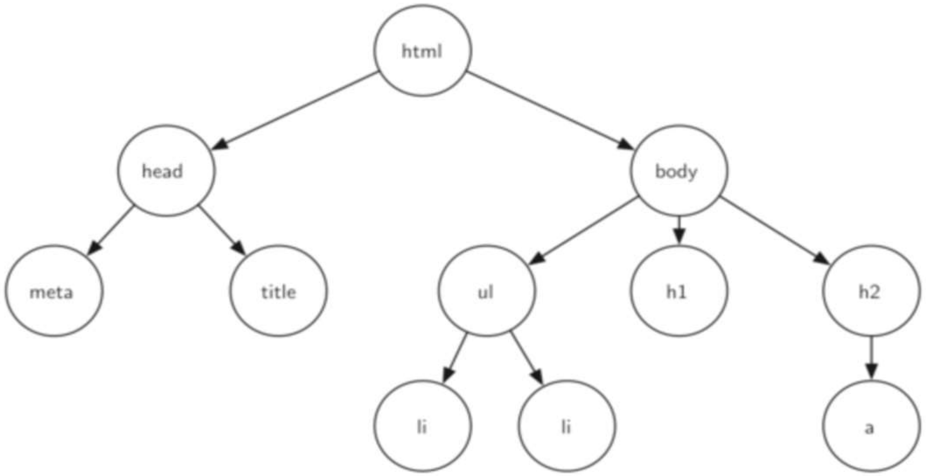
Je AVL drevo edina taka struktura?

Večsmerno drevo

Večsmero drevo

V dvojiškem drevesu imajo vozlišča največ 2 naslednika, v večsmernem drevesu jih imajo lahko več

Primer: predstavitev strukture HTML dokumenta v večsmernem drevesu:



Na področju algoritmike bolj zanimivo drevo, pri katerem pride do ”razvejanost” tudi v vozliščih.
Pri teh drevesih vozlišča ne vsebujejo enega samega podatka ampak jih vsebujejo več.

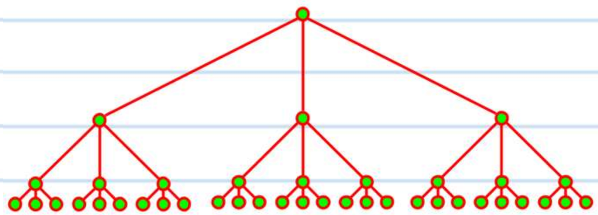
k-tiško drevo

Večsmerno drevo, v katerem vsako vozlišče hrani **do k-1 elementov** (ključ + podatek) in ima **do k naslednikov**, imenujemo **k-tiško drevo.**

Pesvdokoda strukture:

- **Red vozlišča** v je enak številu ključev, ki jih vozlišče v hrani. V k-tiškem drevesu velja: ***1≤red(v)≤k-1***.
- Število poddreves vozlišča v je enako *red(v)+1*

Definicija: Polno k-tiško drevo: vsako vozlišče ima natanko k-1 ključev (red vsakega vozlišča je k-1) in vsako vozlišče, ki ni list, ima natanko k poddreves.



primer: polno 3-tiško drevo

k-tiško drevo

- ✧ **Koliko je elementov v k-tiškem drevesu?**

- ✧ **Globina k-tiškega drevesa?**

Urejeno k-tiško drevo

- ✧ Čemu služijo koreni v k-tiškem drevesu? Elementi lahko nosijo informacije, vendar, je to vse?
- ✧ V kakšnem odnosu so koreni k-tiškega drevesa s poddrevesi?

Podobno, kot smo to naredili pri dvojiških iskalnih drevesih,
lahko tudi pri k-tiških drevesih korene uporabimo kot informacijo,
ki nam pomaga pri sprehodu po drevesu.

Definicija: Urejeno k-tiško drevo je k-tiško drevo, v katerem za vsako vozlišče v in za vsak i, ($0 \leq i < red(v)$), velja:

- vsi elementi v poddrevesu poddrevo[i] so manjši od koren[i],
- vsi elementi v poddrevesu poddrevo[i+1] so večji ali enaki od koren[i].

B-drevo

B-drevo

Definicija po domače: B-drevo reda *b* je urejeno k-tiško drevo, v katerem ima vsako vozlišče najmanj b/2 in največ b poddreves.

Natančneje: **B-drevo reda *b*≥3 je urejeno k-tiško drevo**, v katerem velja:

1. vsako vozlišče, ki ni koren drevesa, ima k_v ključev, kjer je $\left\lceil \frac{b}{2} \right\rceil - 1 \leq k_v \leq b - 1$;
2. koren ima lahko od 1 do b-1 ključev;
3. vsi listi so na isti globini.

Posledice definicije in ostale lastnosti:

✧ vsako vozlišče, ki ni koren in ni list ima od $\left\lceil \frac{b}{2} \right\rceil$ do *b* poddreves;

✧ listi imajo ključe, nimajo pa poddreves (oziroma si ti NULL);

✧ za vsako vozlišče `v`, ki ni list, in za vsak *i*, $0 \leq i < k_v$ velja:

```
v.poddrevo[i] < v.koren[i] < v.poddrevo[i+1]
```

B-drevo - lastnosti

- točka 1 v definiciji nam zagotavlja, da se drevo ne bo izrodilo; že pri najmanjšem b=3 bomo imeli v vsakem vozlišču vsaj 2 poddrevesi, torej bo globina logaritmična!
- ker je B-drevo urejeno, po njem lahko hitro iščemo; ker gremo pri iskanju v globino (in je ta O(logn)), je tudi časovna zahtevnost iskanja O(logn).
- **Ali sta meji za število poddreves (b/2 in b) smiselno določeni?**

B-drevo - lastnosti

- **Največje število elementov v B-drevesu globine h?**

- **Kolikšna je maksimalna globina B-drevesa z n elementi?**

B-drevo - lastnosti

✧ **Kakšna pa je zares časovna zahtevnost iskanja?**

B-drevo – operacija insert()

- ✧ Novi element vedno vstavimo v list (do lista se sprehodimo enako kot pri iskanju)
  - Če je v listu, kamor bomo vstavili, manj kot b-1 elementov → vstavimo in končamo!
  - Če nadaljujemo z vstavljanjem, se bo slej ko prej zgodilo, da bo v listu zmanjkalo prostora → takrat opravimo razcep, **odvečno vozlišče** (rekurzivno) shranimo en nivo više

Katero je “odvečno” vozlišče?

Časovna zahtevnost vstavljanja:

- ✧ sprehod do lista: `O(h)`
- ✧ popravljanj strukture: razcep vozlišča opravimo v konstantnem času, vendar moramo to v najslabšem primeru ponoviti `h`-krat (do korena) → `O(h)`
- ✧ ------------------> skupaj: $T_{insert} = O(h)$

B-drevo – operacija delete()

- ✧ Podobno kot v dvojiškem iskalnem drevesu: izbrisani element nadomestim s skrajno desnim elementom v levem poddrevesu.
- ✧ Ko odstranim element v listu, se lahko zgodi, da bo imel njegov oče premalo otrok → takrat pride do zlivanja - merge (obratna operacija od split).
- ✧ Zlivanje lahko povzroči, da do iste težave (premalo otrok) pride na enem nivoju višje -> zato je treba postopek ponoviti vse do korena (ki pri tem morda izgine!)

Časovna zahtevnost brisanja:

- ✧ sprehod do vozlišča: `O(h)`
- ✧ rekurzivni popravljanje strukture (merge) do korena: `O(h)`
- ✧ ------------------> skupaj: $T_{delete} = O(h)$

Časovna zahtevnost

	`find()`	`insert()`	`delete()`
dinamična tabela	O(n)	O(n)	O(n)
urejena tabela	O(log n)	O(n)	O(n)
seznam	O(n)	O(1)	O(n)
urejen seznam	O(n)	O(n)	O(n)
preskočni seznam	O(lg(n))* *pričakovani čas	O(lg(n))* *pričakovani čas	O(lg(n))* *pričakovani čas
dvojiško iskalno drevo	O(h)=O(n)	O(h)=O(n)	O(h)=O(n)
AVL drevo	O(h)=O(lg n)	O(h)=O(lg n)	O(h)=O(lg n)
B-drevo	$O(\log_b n)$*	$O(\log_b n)$*	$O(\log_b n)$*

* to je res ob predpostavki, da je edina draga operacija branje celega bloka podatkov, iskanje znotraj bloka pa poceni (in ne štejemo); če pa gre pri obeh operacijah za isto časovno zahtevnost, potem moramo namesto $O(\log_b n)$ pisati $O(log\, n)$

B^+-drevo

B+ drevo

- Pri B-drevesih je pomembno, da imamo čim več ključev v enem vozlišču, saj bo zato višina drevesa manjša, posledično bomo opravili manj ”branj”.

- Če je B velikost vozlišča in V velikost enega elementa, potem je $b \doteq B/V$;
  - ker na skupno velikost vozlišča (B) ne moremo vplivati (določena je z arhitekturo), lahko b povečamo le tako, da zmanjšamo V.

- Velikost vozlišča lahko zmanjšamo tako, da v njem hranimo le ključe (informacijo o tem, kako priti do lista), v vse podatke, ki pripadajo ključu, pa hranimo v listih → dobimo B+ drevo

- V B+ drevesu uporabljamo dva tipa vozlišč: notranja (v njih hranimo le ključe) in liste (v njih hranimo ključe in podatke).

Primer: v B+ drevo (b=3) dodaj elemente 8 4 2 1 9 7 5 3

TTF drevo

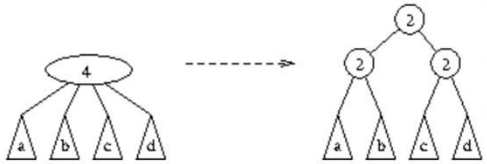
TTF drevo

- ✧ TTF (two-three-four) drevo je B-drevo pri b=4.
- ✧ V TTF drevesih ima vsako vozlišče 1, 2, ali 3 ključe in 2, 3, ali 4 poddrevesa.

- ✧ Primer: v prazno TTF drevo vstavi znake DREVESNICA ABC. Kaj opaziš?

TTF drevo

- Težavi s vstavljanjem v 4-vozlišče bi se lahko izognili tako, da bi na poti od korena do lista (kamor bomo vstavili) vsa 4 vozlišča razbili in sicer takle:



- Ko bi potem v list vstavili nov element, popravljanje ne bi bilo potrebno.
- Toda: s takim razbijanjem se pokvari struktura TTF drevesa.

- Lahko lastnosti, kot jih ima TTF drevo prenesemo na dvojiško drevo (in s tem ohranimo lepe lastnosi iz obeh svetov)?

DA, lahko. Drevo, ki tako nastane, se imenuje rdeče-črno drevo.

Rdeče-črno drevo

Rdeče-črno drevo

- Rdeče-črno (RČ) drevo je dvojiško drevo, v katerem ima vsako vozlišče svojo barvo – rdečo ali črno.

- RČ drevesa lahko vpeljemo na več načinov, mi jih bomo preko TTF dreves: RČ drevo dobimo s preoblikovanjem TTF drevesa tako, da 2-vozlišče pretvorimo v Č , 3-vozlišče v ČR in 4-vozlišče v ČRR vozlišča, takole:

2-vozlišče

3-vozlišče

4-vozlišče

Rdeče-črno drevo

Po pretvorbi TTF drevesa dobimo dvojiško drevo, za katera velja:

- ✧ **1. Vsako vozlišče drevesa je bodisi rdeče bodisi črne barve**
- ✧ **2. Koren drevesa je črn**
- ✧ **3. V drevesu ni dveh sosednjih rdečih vozlišč.**
- ✧ **4. Za vsako vozlišče v velja: vsaka pot od v do lista vsebuje enako število črnih vozlišč.**

Definicija: Dvojiško drevo z lastnostmi 1, 2, 3 in 4 se imenuje rdeče-črno drevo.

Primer pretvorbe TTF drevesa v rdeče-črno drevo

Trditev 1: Obstaja RČ drevo s samimi črnimi vozlišči.

Trditev 2: Globina RČ drevesa je logaritmična, $h = O(\lg n)$.

Vstavljanje v rdeče-črno drevo

Operacija `insert()`:

✧ Vstavljanje v RČ drevo ima kar nekaj posebnih primerov in je precej bolj zapleteno kot vstavljanje v AVL ali B-drevo.

✧ Posebni primeri pri vstavljanju ne prinesejo veliko k časovni zahtevnosti (vedno se zgodi le eden od njih), zato je časovno vstavljanje zelo učinkoviti.

✧ Osnovna ideja vstavljanja:
- novo vozlišče vstavim kot rdeče vozlišče po principu BST,
- če se je s tem porušila struktura RČ drevesa, drevo popravim; pri tem obravnavam dve možnosti:
  - a) stric vstavljenega vozlišča je rdeč in
  - b) stric vstavljenega vozlišča je črn.

Časovna zahtevnost vstavljanja:

✧ BST vstavljanje: `O(h)=O(lg n)`

✧ rekurzivno popravljanje strukture do korena: `h*O(1) = O(h)= O(lg n)`

✧ -----------------> skupaj: $T_{insert} = O(lg\ n)$

Brisanje iz rdeče-črnega drevesa

Operacija `delete()`:

- vozlišče pobrišem kot v klasičnem BST, nato po potrebi opravim prebarvanje + rotacije;
- vstavljanj vs brisanje:
  - pri vstavljanju se je morda porušilo pravilo "ni dveh zaporednih rdečih", pri popravljanju smo gledali barvo **strica**;
  - pri brisanju se morda poruši pravilo "enako število črnih na poti do lista"; problem se torej pojavi le v primeru, da brišemo črno vozlišče; pri popravljanju gledamo barvo **sorojenca** izbrisanega vozlišča.

Časovna zahtevnost brisanja:

- brisanje v BST: `O(h)=O(lg n)`
- rekurzivno prebarvanje od lista do izbrisanjega vozlišča: `O(lg n)`
- -------------------> skupaj: $T_{delete} = O(lg\ n)$

- **Primerjava AVL in RČ dreves**
  - AVL in RČ drevesa so dvojiška iskalna drevesa.
  - AVL drevesa so ”bolj uravnotežena” kot RČ drevesa.
  - AVL drevesa imajo manjšo globino in zato krajši povprečni iskalni čas.
  - Vstavljanje v AVL drevesa je počasnejše kot v RČ drevesih.

    Zato: AVL drevesa uporabljam, kadar imam več ali manj fiksno strukturo (malo dodajanja) in je glavna operacija iskanje, RČ drevesa pa uporabljam, kadar je več dodajanja in manj iskanja.
  - uporaba v praksi:
    - AVL: v podatkovnih bazah
    - RČ: za implementacijo podatkovnih struktur Map, Set, HashMap (java8), v Linux jedru (razporejevalnik), …

Povzetek časovnih zahtevnosti

Časovna zahtevnost

	`find()`	`insert()`	`delete()`
dinamična tabela	O(n)	O(n)	O(n)
urejena tabela	O(log n)	O(n)	O(n)
seznam	O(n)	O(1)	O(n)
urejen seznam	O(n)	O(n)	O(n)
preskočni seznam	O(lg(n))* *pričakovani čas	O(lg(n))* *pričakovani čas	O(lg(n))* *pričakovani čas
dvojiško iskalno drevo	O(h)=O(n)	O(h)=O(n)	O(h)=O(n)
AVL drevo	O(h)=O(lg n)	O(h)=O(lg n)	O(h)=O(lg n)
B-drevo	$O(\log_b n)$	$O(\log_b n)$	$O(\log_b n)$
Rdeče-črno drevo	O(lg n)	O(lg n)	O(lg n)