

# Napredne strukture podataka

*(Advanced Data Structures)*

Copyright (C) Nikica Hlupić and Damir Kalpić 2009, All rights reserved

# Preskočne liste (*Skip Lists*)

Pugh, William: "Skip lists: a probabilistic alternative to balanced trees",  
Communications of the ACM 33, June 1990, pp. 668–676

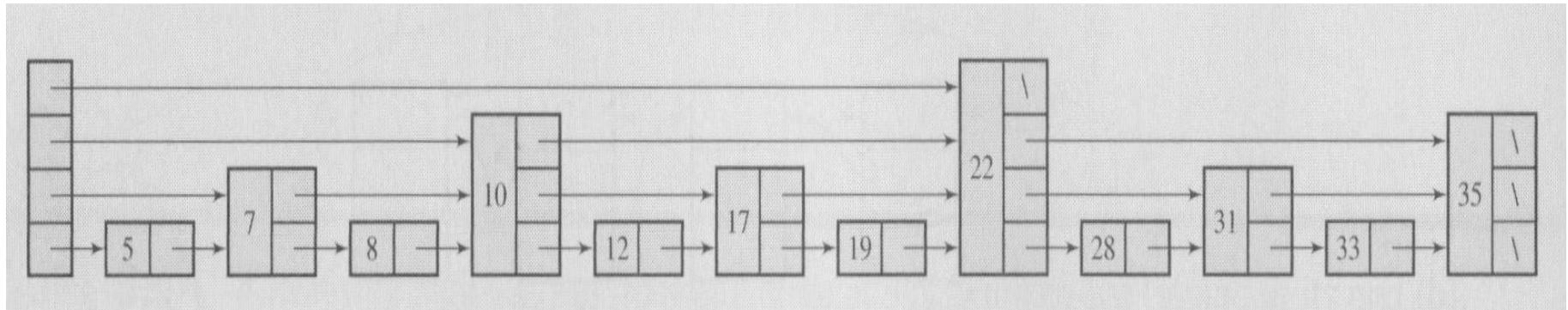
Osnovni nedostatak lista:  $O(n)$  pretraživanje.

Ograničavajuće svojstvo stabala: po prirodi hijerarhijske strukture, logički neprikladne za sve primjene.

Skip liste:

- ključne operacije  $O(\log_2 n)$ ... $O(n)$
- nema hijerarhije
- relativno jednostavno programiranje.

# Savršena (teorijska) struktura skip liste:



Stupanj (*level*) čvora = broj pokazivača.

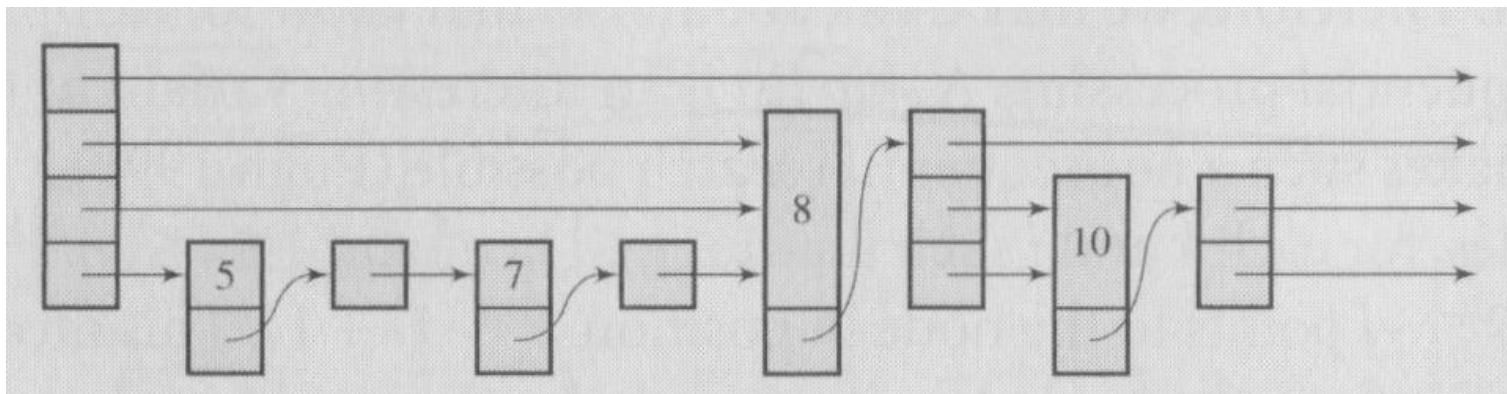
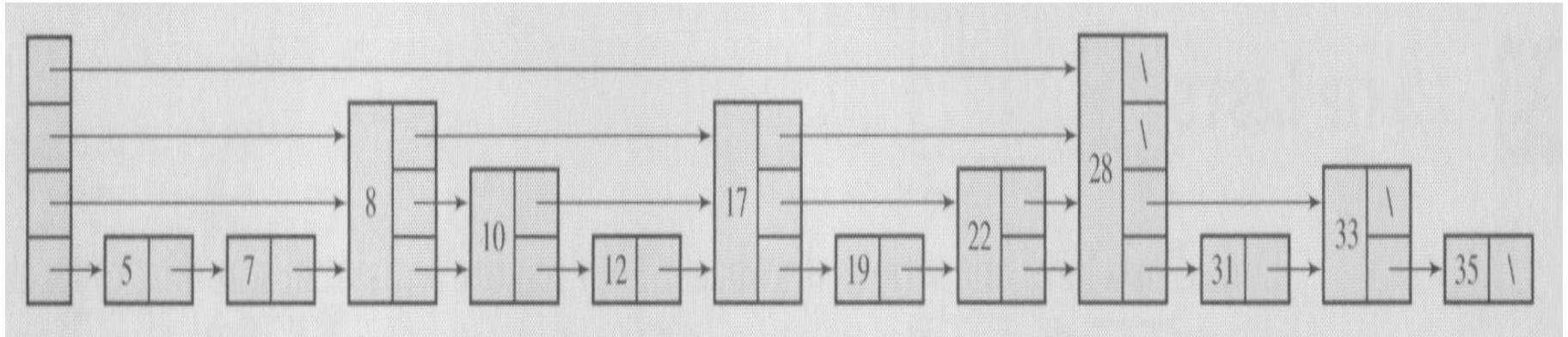
Održavanje savršene strukture je komplikirano i neučinkovito:

- nakon ubacivanja ili brisanja čvora treba restrukturirati sve čvorove iza njega (mijenjaju se stupnjevi čvorova, dakle i broj i odredišta pokazivača!).

U stvarnosti se odustaje od zahtjeva za pravilnim rasporedom čvorova i samo se *nastroji* postići pravilna razdioba njihovih stupnjeva.

“Nastroji” znači da se ni na tome ne inzistira bezuvjetno, nego se osigurava **najveća vjerojatnost pravilne razdiobe stupnjeva čvorova u listi.**

# Uporabna struktura skip liste:



Brzina pristupa podatcima je u prosjeku sumjerljiva brzini u AVL ili RB stablu.

Uporabna struktura skip liste ovisi o dva čimbenika:

1. predviđenom kapacitetu  $n$

- pretpostavljeni najveći broj elemenata u listi

2. vjerojatnosti pojedinih stupnjeva čvora

- najčešće se odabire samo vjerojatnost  $p$  prijelaska čvora u višu razinu. Svaki novi čvor je početno prvog stupnja, a potom mu se stupanj pokuša povećati za jedan, pri čemu je vjerojatnost uspjeha jednaka  $p$ . Prijelasci u višu razinu ponavljaju se sve do prvog neuspjeha, nakon čega čvor ulazi u listu s do tada postignutim stupnjem.

Kapacitet  $n$  i vjerojatnost  $p$  određuju sve teorijske značajke preskočne liste (*skip liste*).

Na primjer, vjerojatnost  $P(k)$  da novi čvor u konačnici postigne  $k$ -ti stupanj (tj. vjerojatnost “dodjeljivanja”  $k$ -tog stupnja novom čvoru) jednaka je vjerojatnosti da  $k-1$  puta uzastopno promijeni razinu i jednom (na kraju) ju ne promijeni:

$$P(k) = [P(\text{prijelaz})]^{k-1} \cdot P(\text{ostanak}) = p^{k-1} \cdot (1 - p) .$$

Očekivani (“srednji”) stupanj čvorova u listi (“srednja visina” liste) je

$$E(k) = \sum_{k=1}^{\infty} k \cdot P(k) = (1-p) \sum_{k=1}^{\infty} k \cdot p^{k-1} .$$

Zbroj koji se pojavio  
je poznat:

$$\sum_{k=1}^{\infty} k \cdot p^{k-1} = \frac{1}{(1-p)^2}$$

pa slijedi

$$E(k) = \sum_{k=1}^{\infty} k \cdot P(k) = \frac{1}{1-p} .$$

Logičan rezultat jer za  $p \rightarrow 0$  (vjerojatnost prijelaska u višu razinu teži ka nuli), srednja visina  $E(k) \rightarrow 1$ .

Za  $p=1/2$ , lista će biti srednje visine =2.

Točan broj  $n_k$  čvorova  $k$ -tog stupnja je nepredvidiv (slučajna varijabla) i stupnjevi teorijski nisu ograničeni, ali može se izračunati najvjerojatniji (očekivani) broj  $E(n_k)$ .

Ako se stupanj čvora jednak točno  $k$  shvati kao uspješan ishod dodjeljivanja stupnja nekom čvoru, onda je u listi od  $n$  čvorova broj  $n_k$  zapravo broj uspješnih ishoda u ukupno  $n$  Bernoullievh pokusa, pri čemu je vjerojatnost uspjeha u jednom pokusu jednaka  $P(k)$ . Varijabla koja nastaje ponavljanjem Bernoullievh pokusa ima binomnu razdiobu, u ovom slučaju

$$n_k \sim b(n_k; n, P(k)).$$

Prema tome, očekivanje  $E(n_k)$  je

$$E(n_k) = n \cdot P(k) = n \cdot p^{k-1} \cdot (1 - p).$$

To su zanimljivi teorijski rezultati, ali kako se zapravo “projektira” (programira) skip-lista?

Stupanj liste  $h$  = stupanj najvišeg čvora.

Prvo treba odrediti potrebni stupanj (visinu) liste za smještaj  $n$  elemenata.

broj čvorova s barem jednim pokazivačem = broj elemenata u listi =  $n$

broj čvorova s barem dva pokazivača =  $n \cdot p$  , ...

broj čvorova s barem  $k$  pokazivača =  $n \cdot p^{k-1}$  ;  $k = 1, 2, \dots, h$

Savršeno strukturirana lista imat će samo jedan čvor najvišeg stupnja. Dakle, stupanj liste mora zadovoljiti nejednakost  $n \cdot p^{h-1} \geq 1$  iz čega slijedi

$$h \leq 1 + \log_p 1/n = 1 + \log_{1/p} n ; p < 1.$$

(uzeti  $\text{floor}(h)$  jer decimalni dio znači samo da za smještaj  $n$  elemenata trebamo još čvorova nižeg stupnja)

Ako se stupnjevi broje od nula,  $h' = h - 1 \Rightarrow h' \leq \log_{1/p} n$  .

**Napomena:** do istog se rezultata dolazi i oslanjajući se na prethodna teorijska razmatranja. Naime, znamo očekivani broj  $n_k$  čvorova  $k$ -tog stupnja. Pravilno strukturirana lista imat će samo (barem) jedan čvor najvišeg stupnja pa će za najvišu razinu  $h$  vrijediti  $E(n_h) \geq 1$ , odnosno

$$n \cdot p^{h-1} \cdot (1-p) \geq 1 .$$

Rješavanjem te nejednadžbe nalazimo

$$h \leq 1 + \log_p \frac{1}{n(1-p)} .$$

Budući da  $p \in [0,1)$ , logaritam postiže maksimum za minimum argumenta, tj. za  $p=0$ , i jasno je da u najgorem slučaju (najviša lista) mora biti

$$h \leq 1 + \log_p \frac{1}{n(1-0)} = 1 + \log_p \frac{1}{n} .$$

▼ Primjer:  $p = \frac{1}{2}$ ,  $n = 12$ , savršena lista

$$h \leq 1 + \log_2 12 = 1 + 3,6 \rightarrow h = 4$$

s barem jednim pokazivačem: svi  $= n$

s barem dva pokazivača:  $\frac{1}{2} \cdot n = 6$

s barem tri pokazivača:  $(\frac{1}{2})^2 \cdot n = 3$

s barem četiri pokazivača:  $(\frac{1}{2})^3 \cdot n = 3/2 \rightarrow 1$

Broj čvorova pojedinog stupnja?

najviših  $= n \cdot p^{h-1} = 12/8 \rightarrow 1$

za jedan nižih  $= n \cdot p^{h-1-1} - \text{broj najviših} = 12/4 - 1 = 2$

za još jedan nižih  $= n \cdot p^1 - \text{broj svih viših} = 12/2 - 3 = 3$

samo jedan pokazivač  $= 12 - (1 + 2 + 3) = 6$

Drugi način: izravno iz formule za  $E(n_k)$

$$E(n_k) = n \cdot P(k) = n \cdot p^{k-1} \cdot (1-p).$$

Očekivani broj čvorova pojedinog stupnja:

$$E(n_1) = 12 \cdot (1/2)^0 \cdot 1/2 = 6$$

$$E(n_2) = 12 \cdot (1/2)^1 \cdot 1/2 = 3$$

$$E(n_3) = 12 \cdot (1/2)^2 \cdot 1/2 = 3/2 \quad \rightarrow \quad 2$$

$$E(n_4) = 12 \cdot (1/2)^3 \cdot 1/2 = 3/4 \quad \rightarrow \quad 1$$

$$E(n_5) = 12 \cdot (1/2)^4 \cdot 1/2 = 12/32 < 1/2 \quad \rightarrow \quad 0$$

Rezultat kojemu je decimalni dio veći ili jednak  $\frac{1}{2}$  treba zaokružiti na prvi veći cijeli broj (npr. funkcijom `ceil`) jer se radi o očekivanju, a ono je izvedeno iz razmatranja savršene liste. Takav rezultat znači da bi savršena lista imala još jedan čvor  $k$ -tog stupnja, ali ne bi bila u cijelosti popunjena.  $\blacktriangle$

## Određivanje stupnja novog čvora - osnovna ideja:

Teorija vjerojatnosti: ako je vjerojatnost uspješnog ishoda nekog pokusa  $p$ , vjerojatnost  $k$  uzastopnih uspješnih ishoda je  $p^k$ .

Ideja: ponavljamo pokus vjerojatnosti  $p$  sve dok završava uspješno, a broj uzastopnih uspjeha imat će razdiobu kakvu trebamo za stupnjeve čvorova.

```
int RandomLevel (p, maxListLevel)
{
    int razina = 1;          //razine se broje od =1
    while ((float) rand() /RAND_MAX < p)
        && (razina < maxListLevel))
        ++razina;
    return razina; }         //P(razina)=p^(razina-1)
```

Nedostatak – višekratno izračunavanje slučajnog broja.  
Može i brže...

## Učinkovitije određivanje stupnja novog čvora:

1. na temelju  $n$  i  $p$  izračunati  $h$
2. odrediti broj  $n_k$  čvorova pojedinog stupnja (mora biti  $\sum n_k = n$ )
3. kapacitet liste  $n$  podijeliti u pretince (blokove) kapaciteta  $n_k$ 
  - redni broj (stupanj) pretinca je jednak stupnju čvorova u njemu, a granice pretinaca određuju se iz  $n_k$   
(npr. prvi pretinac ima granice  $[1, n_1]$ , drugi  $[n_1+1, n_1+n_2]$ , treći  $[n_1+n_2+1, n_1+n_2+n_3]$  itd.)
4. izračunati slučajni broj  $x$  iz intervala  $[1, n]$
5. stupanj čvora odgovara stupnju pretinca kojem pripada (u koji “upada”)  $x$

Intervali se odrede na početku (jednom) pa se određivanje stupnja novog čvora svodi na izračunavanje jednog slučajnog broja i nekoliko usporedbi.

▼ Primjer:  $p = \frac{1}{2}$ ,  $n = 12$

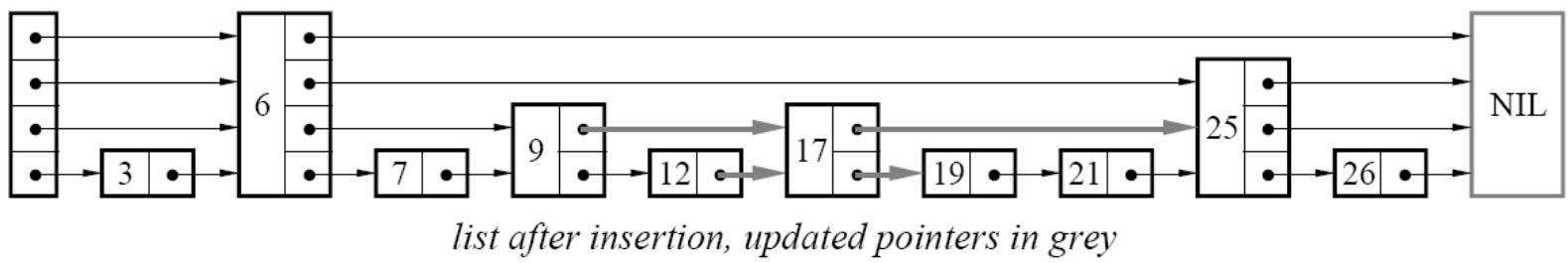
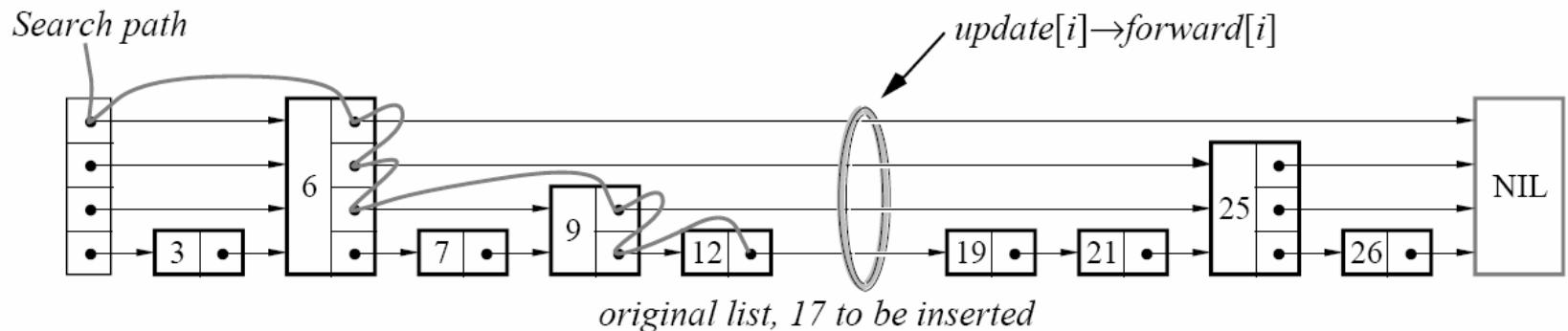
$$h \leq 1 + \log_2 12 = 1 + 3,6 \quad \Rightarrow \quad h = 4$$

stupanj	$n_k$	donja granica	gornja granica
1	6	1	6 ( $= n_1$ )
2	3	7 ( $= n_1+1$ )	9 ( $= n_1+ n_2$ )
3	2	10	11
4	1	12	12

Za niz slučajnih brojeva 5, 3, 11, 7 i 9 dodavali bi se čvorovi redom prvog, prvog, trećeg, drugog i drugog stupnja.



# Pretraživanje i dodavanje čvora u skip-listu:



SkipList, SkipListObj

## Pretraživanje skip-liste:

```
int ListSearch(SkipList* sl, int srkey)
{ //Vraća =1 ako je traženi element 'srkey' u listi 'sl', inače =0.
    int i;
    SkipNode* x = sl->header;
    //'x' pokazuje čvor ispred onog čiji se ključ ispituje.
    for(i = sl->ListLevel; i >= MinLevel; i--)
        //'i' = razina; od najviše prema najnižoj...
    {      while(x->next[i] != NULL
                && x->next[i]->key < srkey)
            x = x->next[i];
        if (x->next[i] != NULL
            && x->next[i]->key == srkey)
            return 1;
    }
return 0;
}
```

Komentar: krenuti od najviše razine i pratiti listu na istoj razini sve dok se ne najde na traženi element, ključ veći od traženog (na toj razini nema traženog) ili NULL pokazivač (kraj liste na toj razini). Ako smo naišli na veći ključ ili kraj liste, spustiti se razinu niže i pregledati nju. Taj postupak ponavljati sve do dna liste.

## Algoritam dodavanja čvora u skip-listu:

Dvije faze: - naći mjesto (bilježiti prethodnike na svim razinama!)  
- dodati čvor

```
AddNode (list, key)
update[MaxLevel] = {NULL}; //prethodnici
x = list->head;
//pretraživanje i bilježenje prethodnika
for (all levels i from ListLevel to MinLevel)
{ while (x->next[i] != NULL
           and x->next[i]->NodeKey < key)
           x = x->next[i];           //kraj while petlje
     update[i] = x;
}
//ako je iza 'x' na najnižoj razini kraj liste ili
//veći ključ, dodati novi čvor
if x->next[MinLevel] == NULL
or x->next[MinLevel]->NodeKey != key
```

## Algoritam dodavanja čvora u skip-listu (nastavak):

```
//dodavanje novog čvora ako je prethodni uvjet
//ispunjeno
level = new node level;           //call RandomLevel()
if level > ListLevel
    for all new levels i above ListLevel
        update[i] = list->head;
    ListLevel = level;
x = MakeNode(key, level);
for all levels i from MinLevel to level
    x->next[i] = update[i]->next[i];
    update[i]->next[i] = x;
```

## Algoritam uklanjanja čvora iz skip-liste:

Dvije faze: - naći čvor (bilježiti prethodnike na svim razinama!)  
- ukloniti i oslobođiti memoriju

```
DelNode (list, key)
update[MaxLevel] = {NULL} ;
x = list->head;
for all levels i from ListLevel to MinLevel
{ while (x->next[i] != NULL and
          x->next[i]->NodeKey < key)
        x = x->next[i];           //Kraj while petlje!
    update[i] = x; }
node = x->next[MinLevel];
if node!=NULL && node->NodeKey == key
//ako je čvor u listi, ukloniti ga
```

## Algoritam uklanjanja čvora iz skip-liste (nastavak):

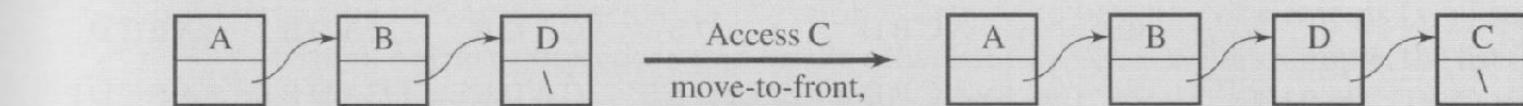
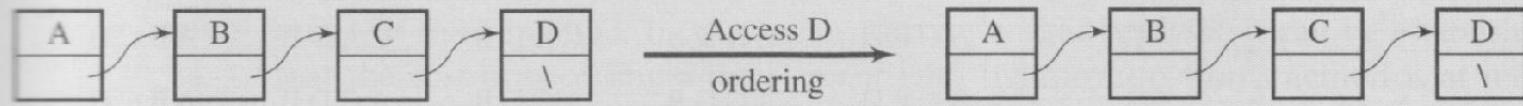
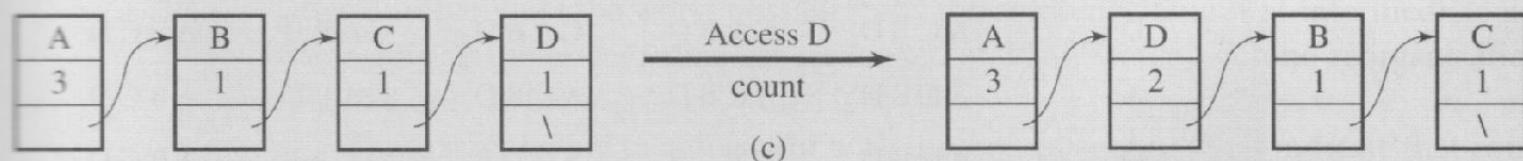
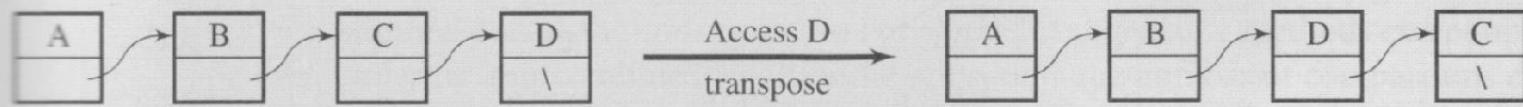
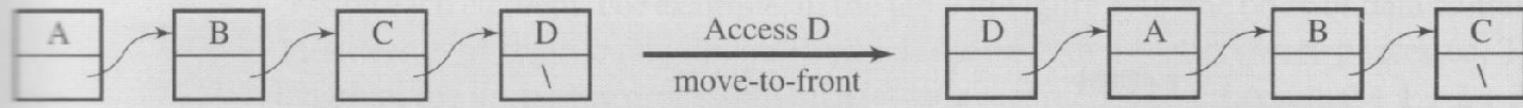
```
//uklanjanje čvora; preusmjeravanje prethodnika
//od 'node' na njegove sljedbenike
{ for all levels i from MinLevel to ListLevel
    if update[i]->next[i] != node
        break;
    update[i]->next[i] = node->next[i] }
release memory occupied by node; //free(node)
//Po potrebi, zabilježiti promjenu visine liste.
while
(list->head->next[list->ListLevel]==NULL
     and list->ListLevel > MinLevel)
list->ListLevel--;
```

# Samoorganizirajuće liste

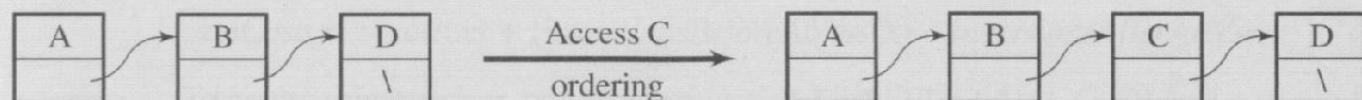
(*Self-Organizing Lists*)

Pretraživanje “običnih” lista može se ubrzati i stalnim mijenjanjem poretku elemenata (*dynamical organizing*) u ovisnosti o raznim kriterijima. Na primjer:

1. *Move-to-front method*: nakon pristupa nekom elementu premjestiti ga na prvo mjesto
2. *Transpose method*: nakon pristupa nekom elementu zamijeniti mu mjesto s prethodnikom
3. *Count method*: elementi u poretku po broju pristupa
4. *Ordering method*: poredak po nekom kriteriju prirodnom za karakter elemenata (npr. po abecedi)



Access C  
move-to-front,  
transpose,  
count



U prve tri metode novi elementi se dodaju na kraj liste (slika *e*), dok se u četvrtoj ubaciju na mjesto određeno kriterijem poretku (slika *f*). Načelno, sve su podjednako brze kad se primjenjuju u odgovarajućim situacijama.

Npr., ordering metoda može ustanoviti da traženog elementa uopće nema u listi i prekinuti pretraživanje, ali dodavanje novih je sporije nego u ostale tri.

Eksperimentalna analiza učinkovitosti tih metoda obično se temelji na odnosu stvarnog i najvećeg mogućeg broja usporedbi. Stvarni broj se dobiva brojanjem usporedbi tijekom testiranja, a najveći mogući zbrajanjem duljina liste prije svake potrage. Time se dobiva prosječni omjer pregledane i ukupne duljine liste tijekom cijelog testiranja.

## Rijetko popunjene tablice (*Sparse Tables*)

Na primjer, kako pohraniti ocjene svih studenata iz svih predmeta na nekom fakultetu (ili cijelom sveučilištu!) u jednom semestru, gdje je ukupno  $P=300$  ponuđenih predmeta i  $S=8000$  studenata?

Prva pomisao je dvodimenzionalna tablica

[predmeti  $\times$  studenti] s ocjenama u poljima.

No, ako svaki student u prosjeku tijekom semestra položi  $PP=6$  predmeta, popunjenost tablice bit će samo  $PP/P = 6/300 \% = 2 \%$ , znači uopće ne koristimo 98 % rezervirane memorije.

Ako su ocjene podatak tipa char (1 byte), tablica zauzima ukupno  $P \cdot S \cdot 1 \text{ byte} = 8000 \cdot 300 \text{ byte} = 2400000 \text{ byte}$   
 $\approx 2,3 \text{ MB}$  memorije.

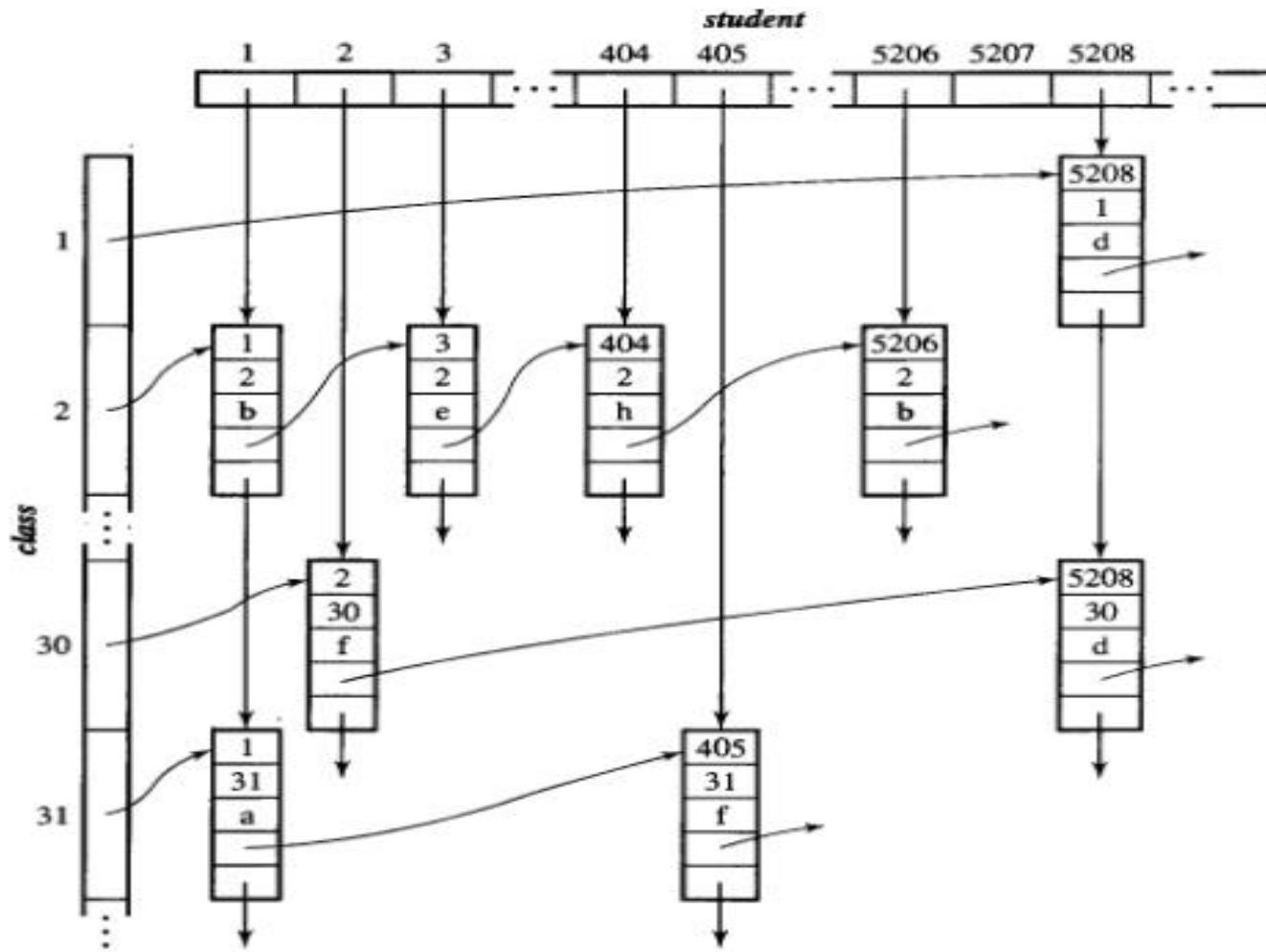
Racionalnije je upotrijebiti dva jednodimenzionalna polja pokazivača Studenti i Predmeti, pri čemu je svaki element tih polja vrh (glava) liste pripadnih podataka.

Tako je svaki element polja Studenti vrh liste predmeta koje je pojedini student položio, dok su elementi polja Predmeti vrhovi lista studenata koji su položili određeni predmet.

Svaki element lista sadrži barem pet podataka:

- oznaku studenta (npr. 2 B)
- oznaku predmeta (npr. 2 B)
- ocjenu (npr. 1 B)
- pokazivač na sljedećeg studenta u listi (npr. 4 B)
- pokazivač na sljedeći predmet u listi (npr. 4 B)

Veličina jednog podatka je 13 B. Popunjenoš te strukture je 100 %, a ukupna potrebna memorija  $8000 \cdot 6 \cdot 13$  B = 624 000 B  $\approx$  0,6 MB memorije.



## Prednosti:

- racionalnije raspolaganje memorijom
- brzo pronalaženje svih podataka jedne skupine koji su u relaciji s jednim podatkom iz druge skupine (npr. svih studenata koji su položili neki predmet ili svih predmeta koje je položio neki student)

## Nedostatci:

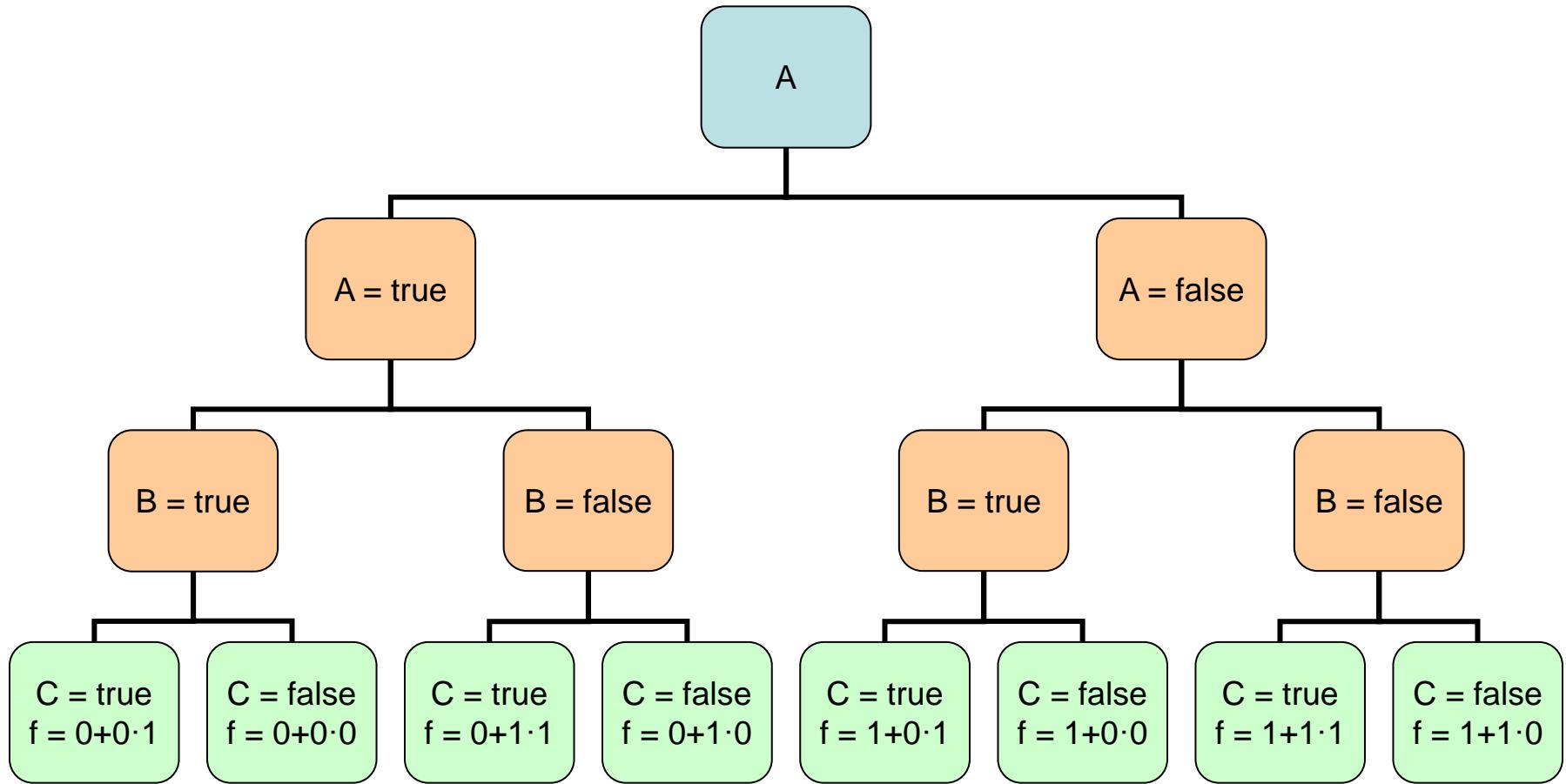
- sporiji pristup pojedinačnim podatcima; umjesto izravnog adresiranja u tablici, treba pretraživati listu
  - čvorovi lista zauzimaju više memorije nego podatak u tablici pa takva struktura brzo prestaje biti svrhovita
- kriterij:  $P \cdot S \cdot (\text{veličina polja u tablici}) > PP \cdot S \cdot (\text{veličina čvora}),$   
dakle  $P \cdot (\text{veličina polja u tablici}) > PP \cdot (\text{veličina čvora})$   
 $\Rightarrow$  u našem primjeru:  $PP < 300 \cdot 1 / 13 \approx 23$

## Stablo odlučivanja (*Decision Tree*)

*Decision Tree* - stablo u kojemu djeca (veze) predstavljaju odluke o uvjetima u nezavršnim (unutarnjim) čvorovima (roditeljima), a listovi su sva moguća rješenja polaznog problema

- olakšavaju rješavanje složenih problema rastavljući komplikirani postupak na niz jednostavnih odluka
- pogodno za razvrstavanje podataka u skupine (klasifikaciju; *classification, clustering*)
- kada za svaki uvjet postoje samo dva moguća odgovora (npr. DA-NE), stablo je binarno i može prikazati (modelirati) rješavanje bilo koje logičke funkcije (variabile mogu biti samo TRUE ili FALSE)

Primjer:  $\overline{A} + \overline{B} \cdot C$



U listovima su sva moguća rješenja zadane funkcije.

# Koja je najveća moguća brzina sortiranja?

Kao teorijsku mjeru brzine uzima se potrebni broj usporedbi pa je pitanje zapravo koliko je najmanje usporedbi potrebno za sortiranje  $n$  elemenata.

Decision Tree sortiranja uspoređivanjem je binarno stablo u kojemu djeca (veze) predstavljaju DA-NE odluke o uvjetima ( $>$  ili  $\leq$ ) u unutarnjim čvorovima, a listovi su svi mogući poretki elemenata.

Poredaka ima  $n!$  pa stablo odlučivanja mora imati najmanje toliko listova (možebitni višak jesu nemogući slučajevi – vidi primjer).

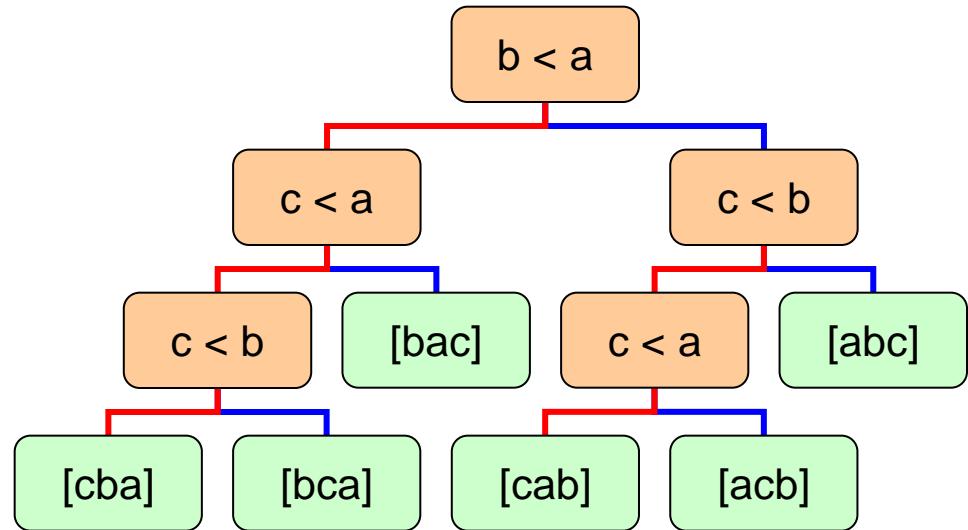
Struktura stabla ovisi o algoritmu i poretku ulaznih podataka.

Primjer:  
sotiranje polja [abc]

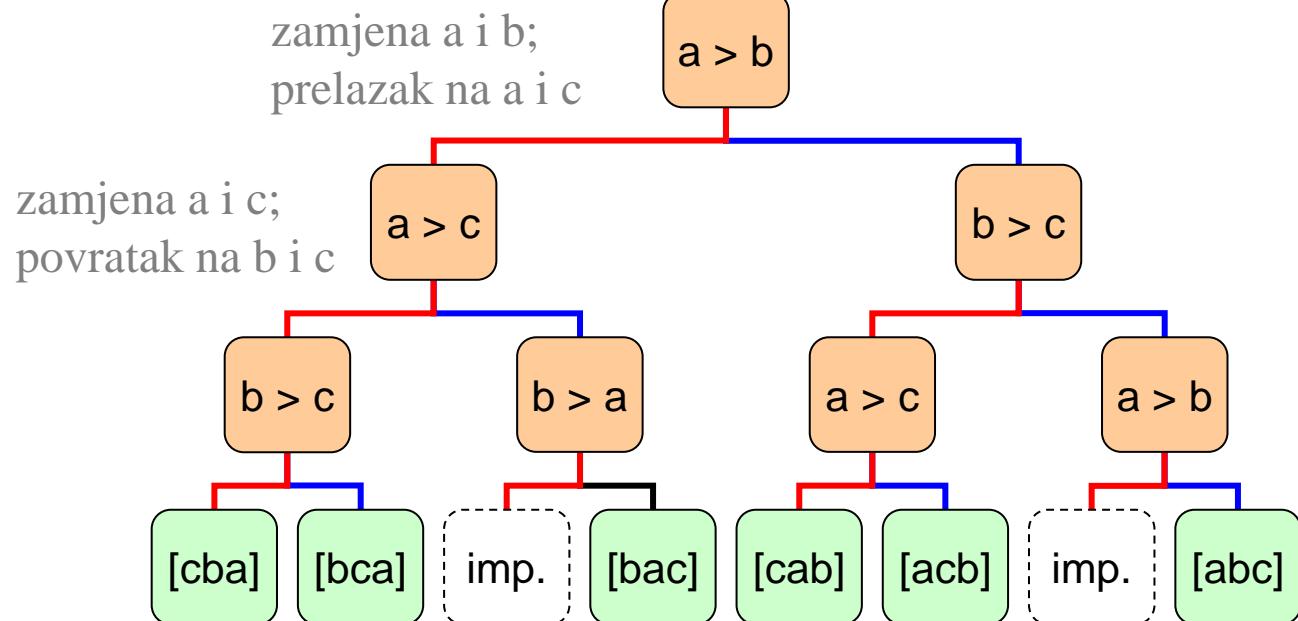
yes, no

## Insertion sort

- broj usporedbi ovisi o poretku ulaznih podataka



Bubble sort  
- broj usporedbi ne  
ovisi o poretku  
ulaznih podataka



Potpuno binarno stablo u  $h$  razina ima  $2^h - 1$  čvorova, od kojih  $2^{h-1}$  listova i  $2^{h-1} - 1$  unutarnjih čvorova.

Označimo:  $m =$  broj listova,

$k =$  broj unutarnjih čvorova.

Bilo kakvo stablo može imati manje ili jednako čvorova kao potpuno stablo pa vrijedi

$$m + k \leq 2^h - 1 \quad ; \quad m \leq 2^{h-1}, k \leq 2^{h-1} - 1.$$

Broj mogućih poredaka prilikom sortiranja  $n$  elemenata je  $n!$  iz čega slijedi da stablo odlučivanja mora imati barem toliko listova. Zaključujemo da mora biti  $n! \leq m \leq 2^{h-1}$ , odnosno

$$2^{h-1} \geq n! \quad \Rightarrow \quad h - 1 \geq \log_2(n!) \quad .$$

U najgorem slučaju moramo doći do najnižeg lista za što nam treba  $h-1$  usporedbi, a gornja relacija kazuje da će to biti njih najviše  $\log_2(n!).$

Dakle, najgori slučaj najbržeg mogućeg sortiranja zahtijeva  $O(\log_2 n!)$  usporedbi.

Sigurna gornja granica te funkcije je  $n \cdot \log_2 n$  jer je  $\log_2 n! \leq \log_2(n \cdot n \cdot \dots \cdot n) = \log_2 n^n = n \cdot \log_2 n$ .

Može se pokazati da je i prosječan broj usporedbi također  $\log_2 n!$  pa je prosječna složenost najbržeg sortiranja opet  $O(n \cdot \log_2 n)$ .

# Brisanje čvorova binarnog stabla za pretraživanje

**Ponoviti gradivo o stablima iz ASP!**

Binarno stablo za pretraživanje (*Binary Search Tree*)

– stablo u kojemu su svi čvorovi lijevog podstabla nekog promatranog čvora manji, a desnog podstabla veći od promatranog čvora

Brisanje čvora mora riješiti tri moguća slučaja:

- 1) čvor je list (nema djece)
- 2) čvor ima samo jedno dijete
- 3) čvor ima dva djeteta

vodeći računa o posebnosti korijena.

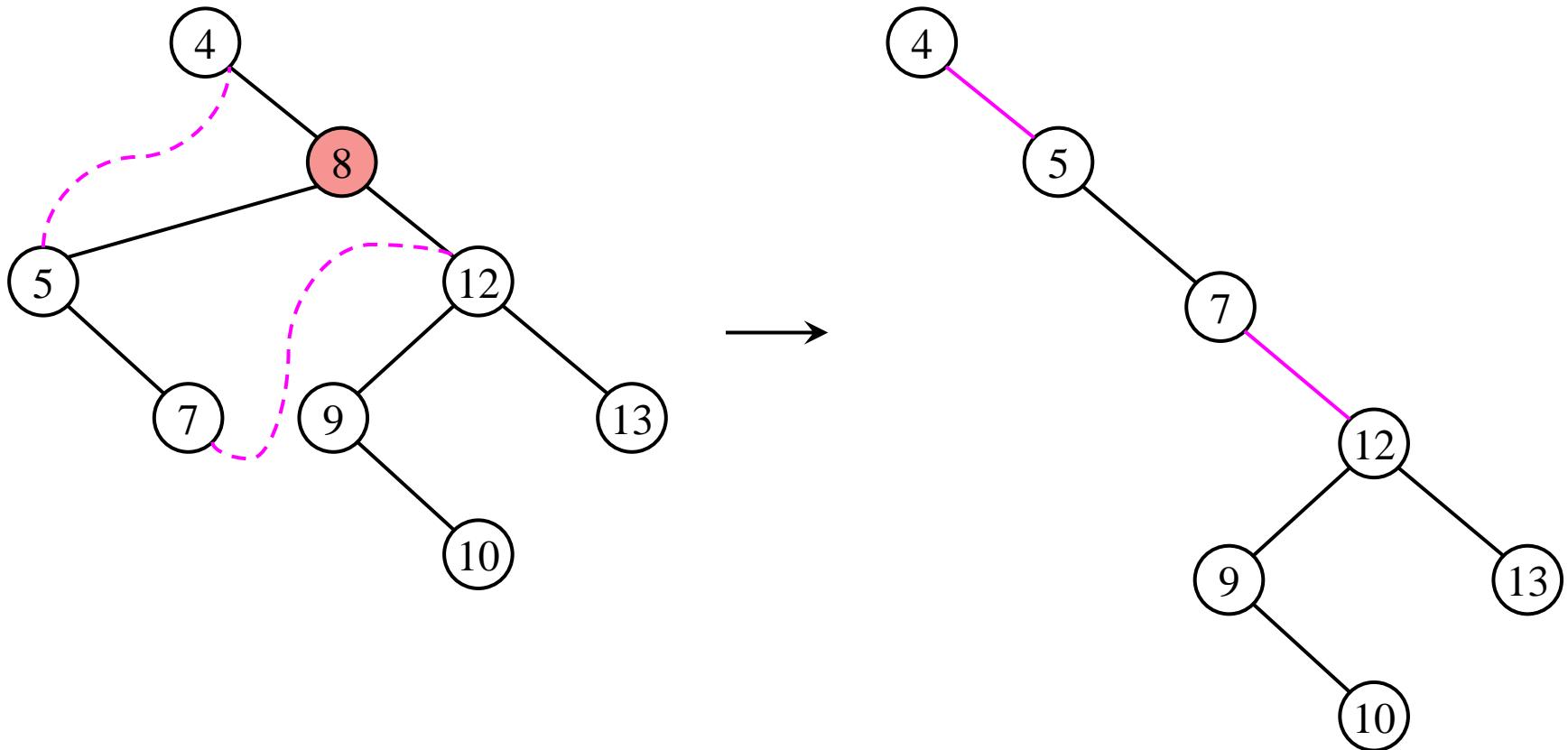
Prva dva slučaja su jednostavna, a treći se najčešće rješava na dva načina:

- a) Brisanje sjedinjenjem (*Deletion by Merging*)
- b) Brisanje kopiranjem (*Deletion by Copying*)

Brisanje sjedinjenjem (*Deletion by Merging*) - osjetno mijenja strukturu stabla (neupotrebljivo za uravnotežena stabla)

1. naći čvor koji se briše i njegovog roditelja
2. naći najveći manji od njega (ili najmanji veći)
  - to je najdesniji u lijevom podstablu (najleviji u desnom) i taj sigurno nema desno (lijevo) dijete
3. desni (lijevi) pokazivač najvećeg manjeg (najmanjeg većeg) usmjeriti na desno (lijevo) dijete čvora koji se briše
4. pokazivač u roditelju usmjeriti na lijevo (desno) dijete čvora koji se briše
5. ukloniti čvor koji se briše (osloboditi memoriju)

# Primjer: brisanje sjedinjenjem



Brisanje Cvora Stabla

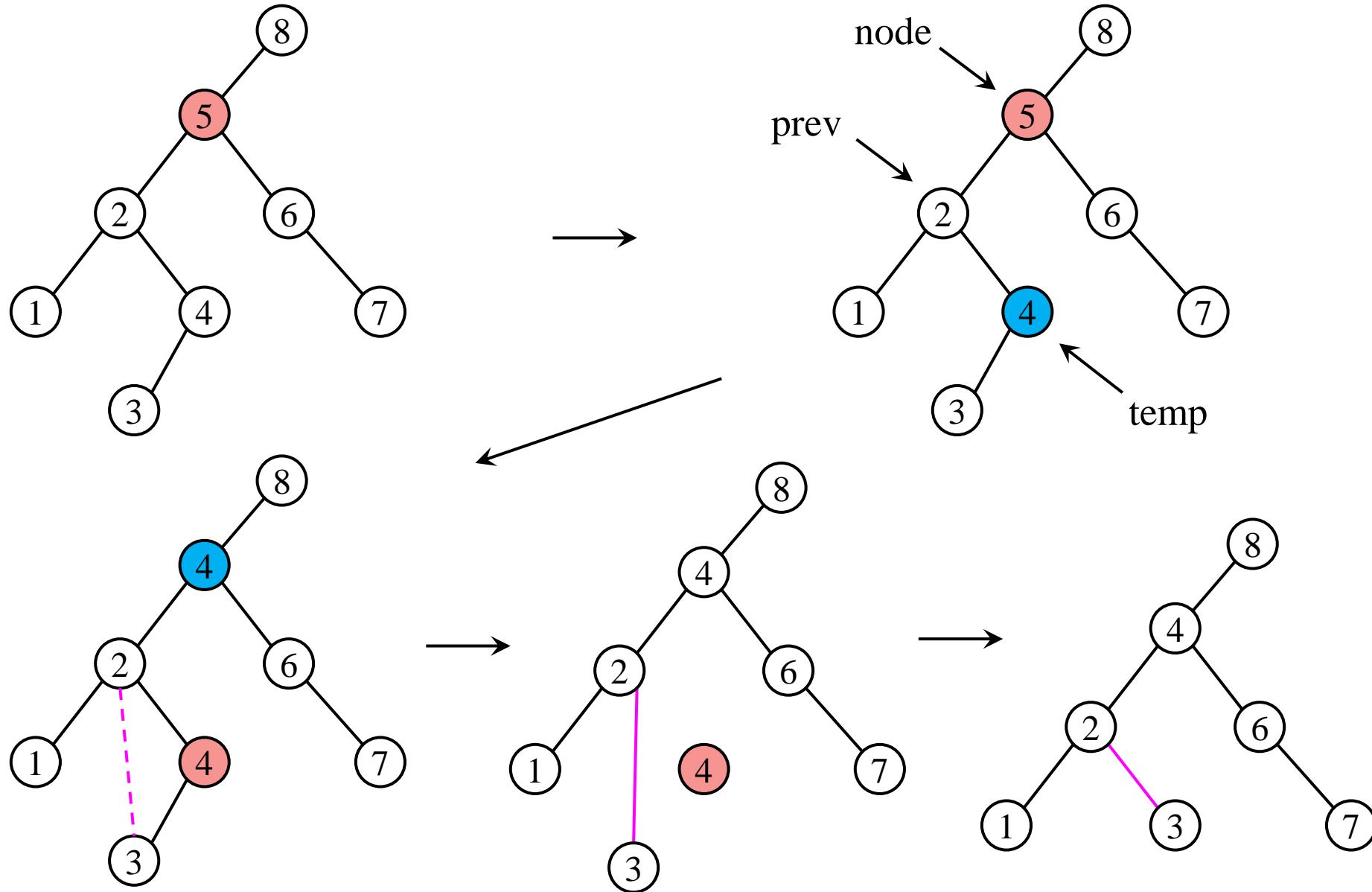
## Brisanje kopiranjem (*Deletion by Copying*)

- umjesto brisanja zadanog čvora, iz stabla se zapravo uklanja zamjenski čvor (“žrtva”)
  - ovim se algoritmom problem brisanja čvora s dva djeteta svodi na brisanje čvora s jednim djetetom ili bez djece (list)
  - minimalno mijenja strukturu stabla
1. naći zamjenski čvor = najbliži prethodnik ili sljedbenik čvora koji se briše (najveći manji ili najmanji veći); zapamtiti njega i njegovog roditelja
    - najbliži je najdesniji u lijevom podstablu (najleviji u desnom) i sigurno nema desno (lijevo) dijete
  2. podatke iz zamjenskog prepisati u čvor koji se briše
  3. desni (lijevi) pokazivač roditelja zamjenskog usmjeriti na dijete zamjenskog (“premostiti” zamjenski)
  4. ukloniti zamjenski (osloboditi memoriju)



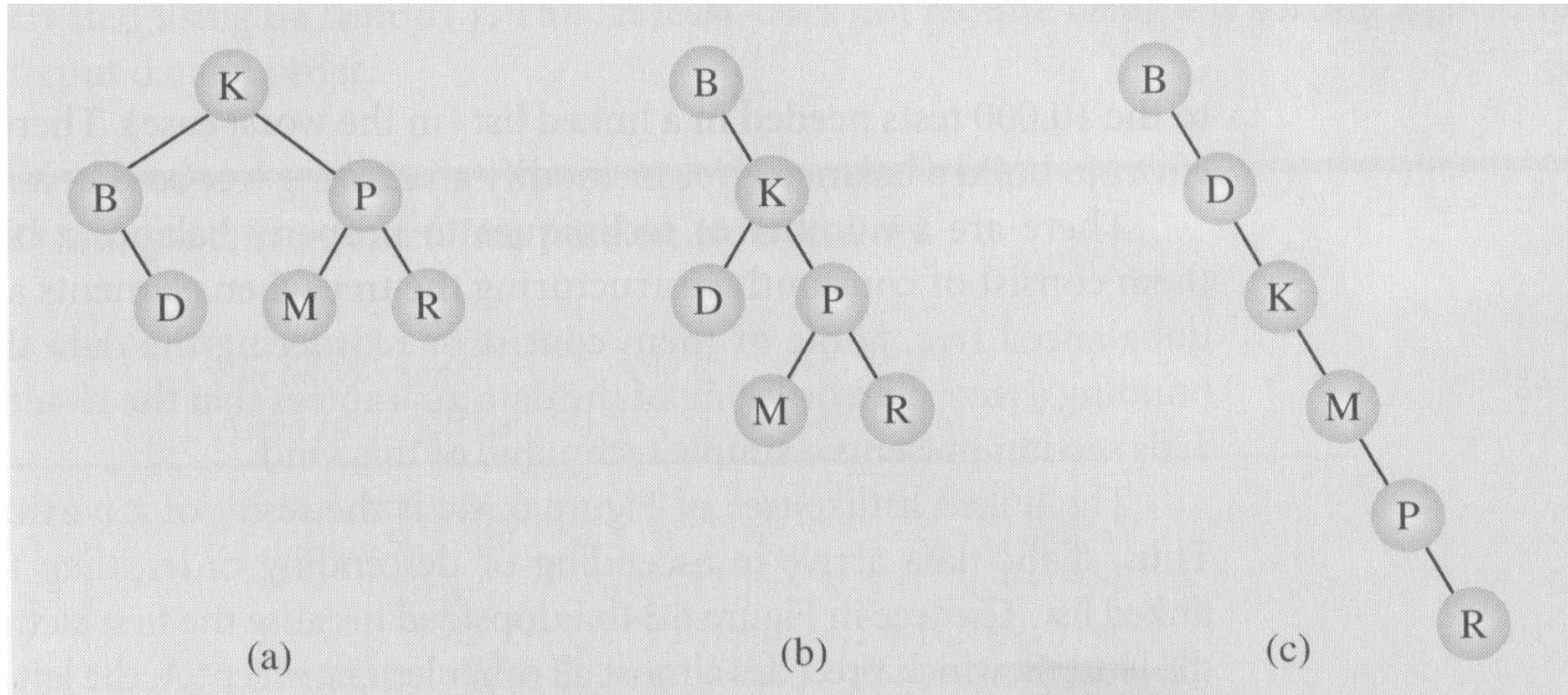
BrisanjeCvoraStabla

# Primjer: brisanje kopiranjem (briše se 5)



## Uravnotežavanje stabla (*Balancing a Tree*)

Osnovna prednost stabla pred listom, brzina pretraživanja, se gubi ako je stablo neprikladne strukture, tj. neuravnoteženo (krajnost: koso stablo = lista; primjer - traženje R).



**Uravnoteženo stablo** (*Balanced Tree*) – stablo u kojemu je razlika visina podstabala svakog čvora najviše jedan.

**Savršeno uravnoteženo stablo** (*Perfectly Balanced Tree*) – uravnoteženo stablo kojemu su svi listovi u najviše dvije razine.

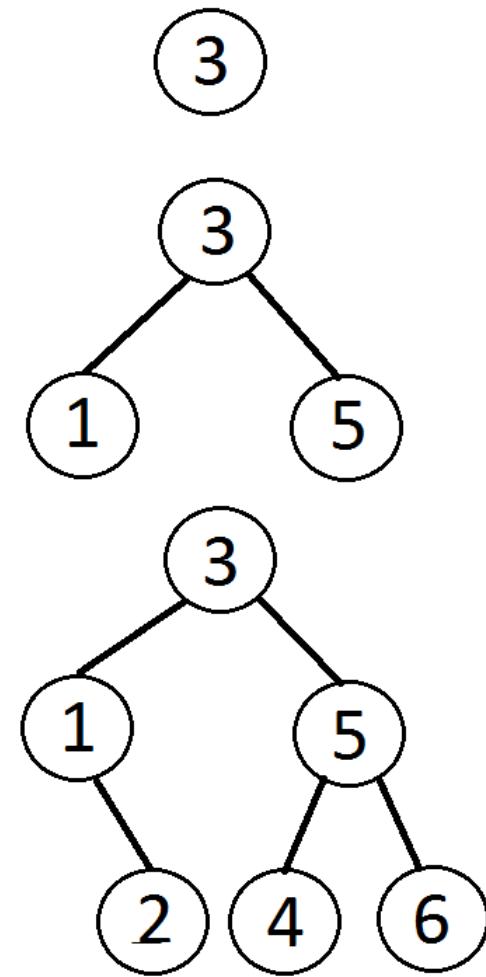
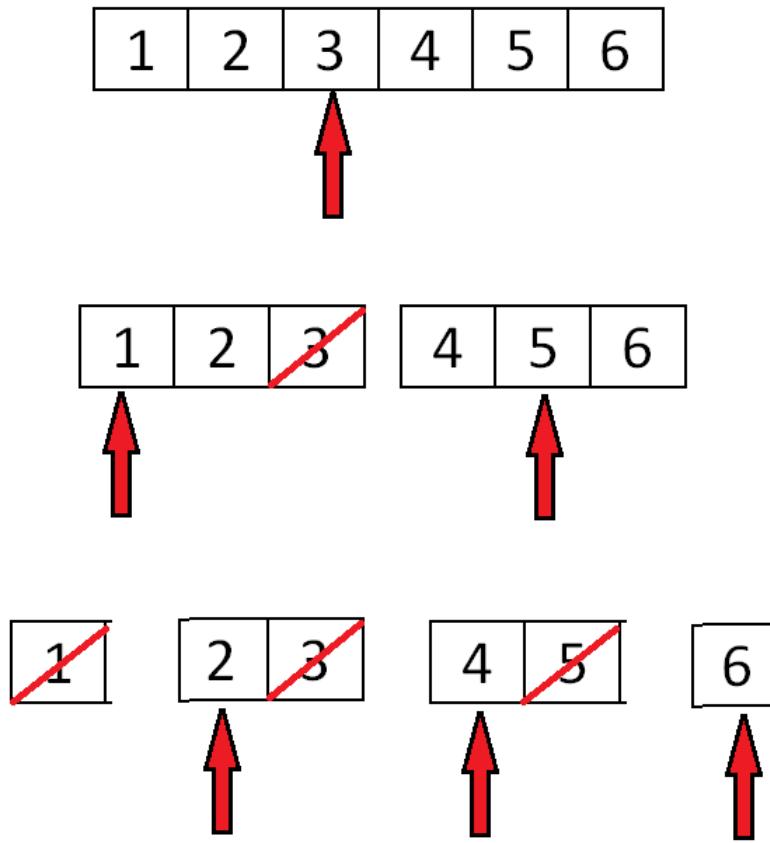
Uravnotežavanje:

- 1) promišljenim redoslijedom upisa podataka
- 2) restrukturiranjem stabla.

## Promišljeni redoslijed upisa podataka

- algoritam sličan binarnom pretraživanju
1. prikupiti i sortirati sve ulazne podatke
  2. korijen stabla = središnji element tog skupa; preostali podatci sada su u dva podskupa
  3. središnji element lijevog podskupa postaje lijevo, a središnji element desnog podskupa postaje desno dijete korijena
  4. s novim podskupima nastalim izdvajanjem središnjih elemenata postupiti na isti način, ponavljajući korake 2 i 3, pri čemu izdvojeni elementi imaju ulogu korijena svojih podstabala

Primjer:



Sortiranje se može izbjegići upisivanjem podataka u neuravnoteženo stablo te potom *inorder* čitanjem i prepisivanjem u neku drugu strukturu, npr. polje.

Opisani algoritam prvo upisuje korijen, zatim njegovo lijevo i desno dijete pa istim redoslijedom djecu djece itd. Za programiranje je osjetno jednostavnije upisati korijen, potom njegovo lijevo dijete pa lijevo dijete lijevog djeteta itd. i tek nakon toga upisivati desnu djecu, od najnižeg čvora prema korijenu.

```
MakeBalTree (data[], left, right)
if left <= right
    middle = (left + right) / 2
    insert data[middle] into the tree
    MakeBalTree (data, left, middle-1)
    MakeBalTree (data, middle+1, right)
```

# Primjer:

Stream of data: 5 1 9 8 7 0 2 3 4 6  
Array of sorted data: 0 1 2 3 4 5 6 7 8 9

---

(a) 0 1 2 3 4 5 6 7 8 9 4

(b) 0 1 2 3 4 5 6 7 8 9 4  
1 → 4

(c) 0 1 2 3 4 5 6 7 8 9 4  
1 → 0 → 2

(d) 0 1 2 3 4 5 6 7 8 9 4  
1 → 0 → 2  
4 → 1 → 0 → 2  
4 → 7 → 5 → 3  
4 → 7 → 5 → 6  
4 → 7 → 5 → 8  
4 → 7 → 5 → 6 → 9

## Nedostatci pripreme ulaznih podataka:

- potreba za dodatnom memorijom
- sortiranje
- rezultat ne mora biti popunjeno stablo

## DSW algoritam

(Colin **D**ay, Quentin F. **S**tout i Bette L. **W**arren)

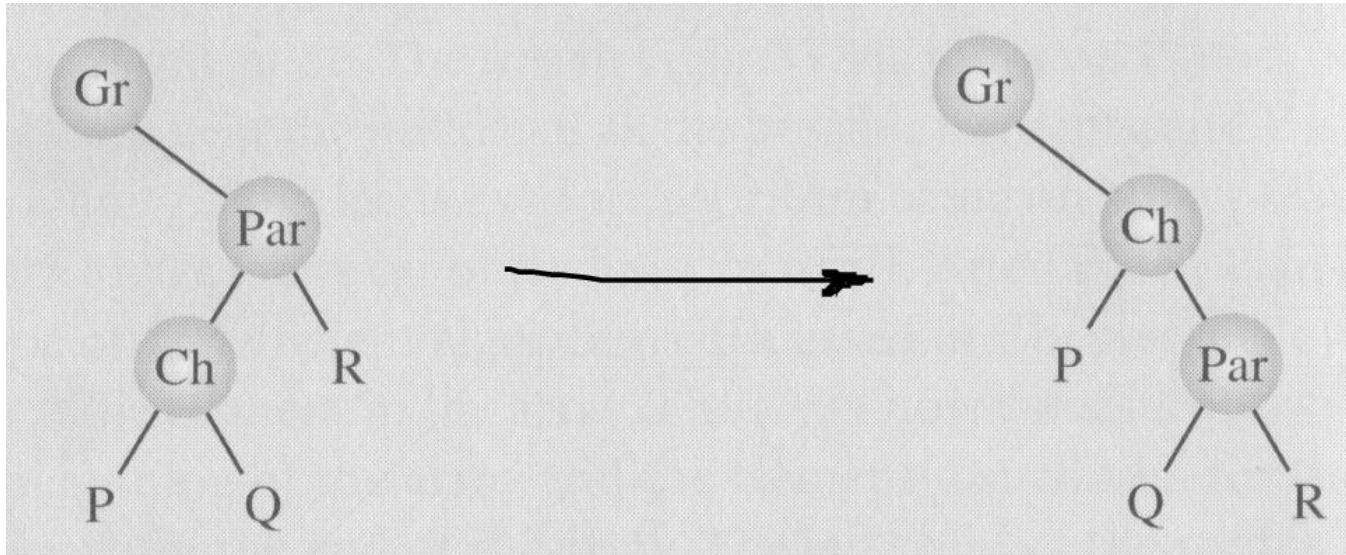
- savršeno uravnotežavanje

Osnova tog algoritma su rotacije u stablu.

Rotacija – postupak kojim dijete postaje roditelj, a roditelj dijete, pri čemu se poštaju definicijska pravila stabla (hijerarhijska struktura stabla s obzirom na zadani kriterij).

Ljeva i desna rotacija su potpuno simetrične  
(isti algoritam sa zamijenjenim značenjem lijevo-desno).

Primjer: desna rotacija Ch oko Par:



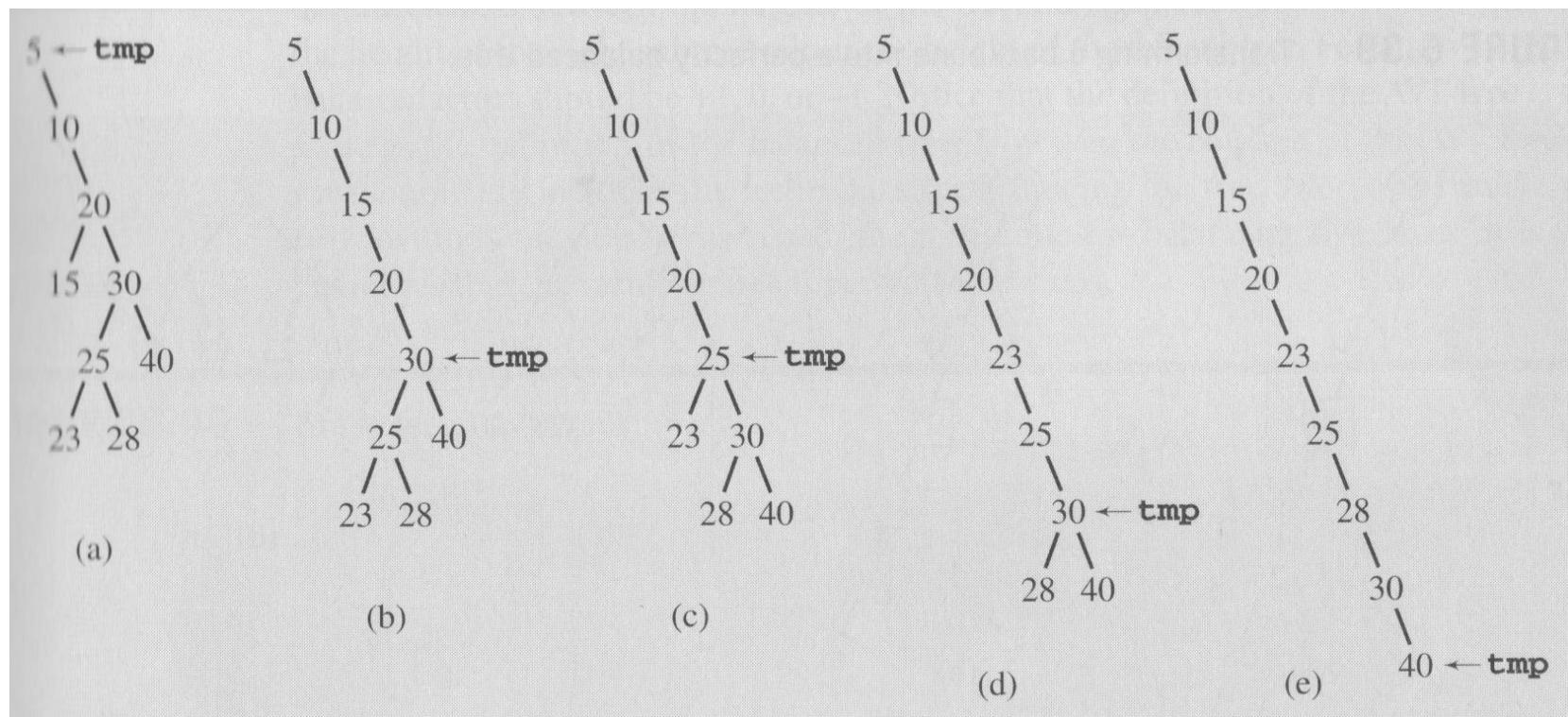
RightRotation (gr, par, ch)

if par *is not the root* //i.e., gr not NULL  
    *redirect pointer in gr to ch;*  
    *redirect left pointer of par to right subtree of ch;*  
    *redirect right pointer of ch to par;*

# DSW algoritam ima dvije faze:

- 1) pretvaranje binarnog stabla u “listoliku” strukturu *kralježnica* (*backbone*; ponekad *vine*)
- 2) pretvaranje *kralježnice* u savršeno uravnoteženo stablo

Prva faza: pretvorba binarnog stabla u kralježnicu



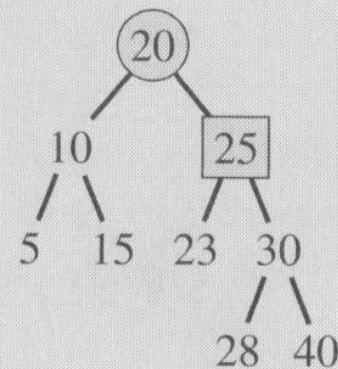
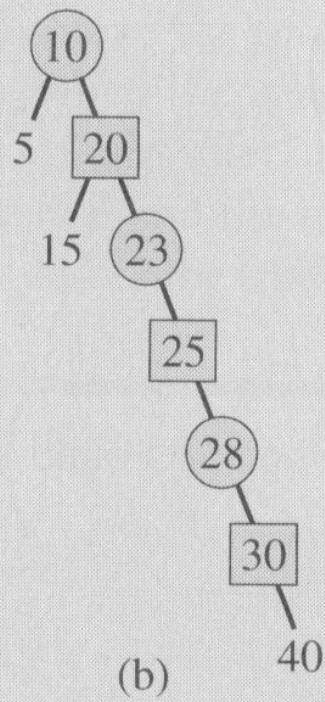
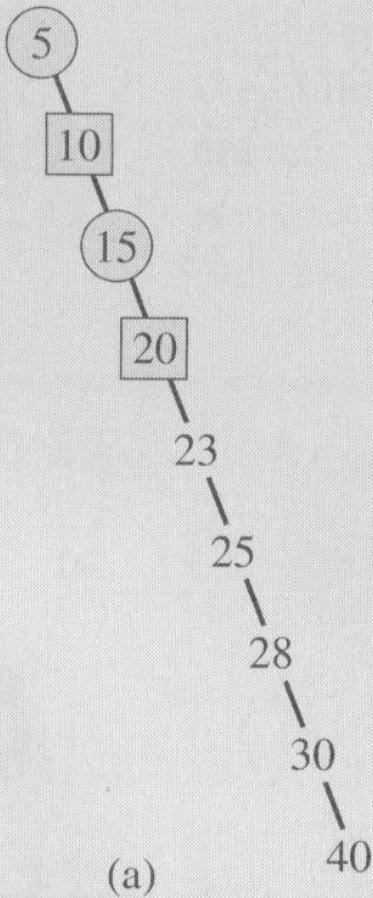
## CreateBackbone (root)

```
tmp = root;  
while (tmp != NULL)  
    if tmp has left child ch  
        right rotate ch about tmp;  
        redirect tmp to the ch;  
    else  
        redirect tmp to its right child;
```

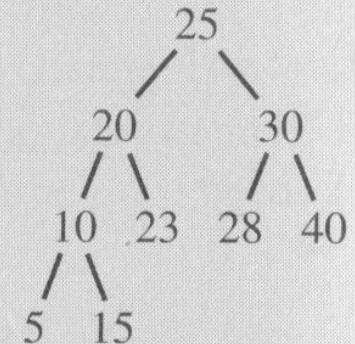
Rotacija iziskuje poznavanje roditelja od tmp pa realizacija zahtijeva još jedan pokazivač.  
Sve se može i simetrično obrnuti, dakle stvarati kralježnicu lijevim rotacijama.

U najgorem slučaju (korijen nema desno podstablo) obavi se najviše  $(n - 1)$  rotacija (složenosti  $O(1)$ ), a do kraja prve faze while uvjet se ispita ukupno  $2(n - 1) + 1 = 2n - 1$  puta. Dakle, složenost je  $O(n)$ .

## Druga faza: pretvorba kralježnice u savršeno stablo



(c)



(d)

Rotacije na slici: okrugli = roditelji,  
kvadratični = djeca koja moraju postati roditelji.

## CreatePerfectTree (n)

$h = \text{largest integer less than or equal to } \log_2(n+1);$

$k = 2^h - 1;$  //broj čvorova u punim razinama budućeg stabla

*make totally  $n-k$  left rotations, taking every second node  
and rotating it about its parent, starting from the top;*

`while (k > 1)`

$k = k/2;$

*make totally  $k$  left rotations, taking every second  
node and rotating it about its parent;*

*(starting from the top)*

Komentar:  $n$  elemenata u potpunom binarnom stablu popunit će  $h$  razina, gdje je  $h$  najveći cijeli broj manji od  $\log_2(n+1)$ , i još će njih  $n - (2^h - 1)$  biti u najnižoj, nepotpunjenoj razini. Taj ostatak se rješava rotacijama prije while petlje, a potpuno popunjene razine rotacijama unutar petlje.

Složenost ove faze također je  $O(n)$  (vidi npr. Drozdek...) pa je ukupna složenost uravnotežavanja stabla DSW algoritmom  **$O(n)$** .

# AVL stabla (izvorno *admissible tree*)

- Georgii Maksimovič Adelson-Velskii, Yevgenii Mikhailovič Landis

Nedostatak DSW algoritma je uravnotežavanje cijelog stabla, iako je ravnoteža najčešće narušena samo lokalno. AVL algoritam uravnotežuje stablo lokalno, ali ne jamči savršenu uravnoteženost cijelog stabla.

AVL stablo je stablo koje zadovoljava AVL definicijsko pravilo.

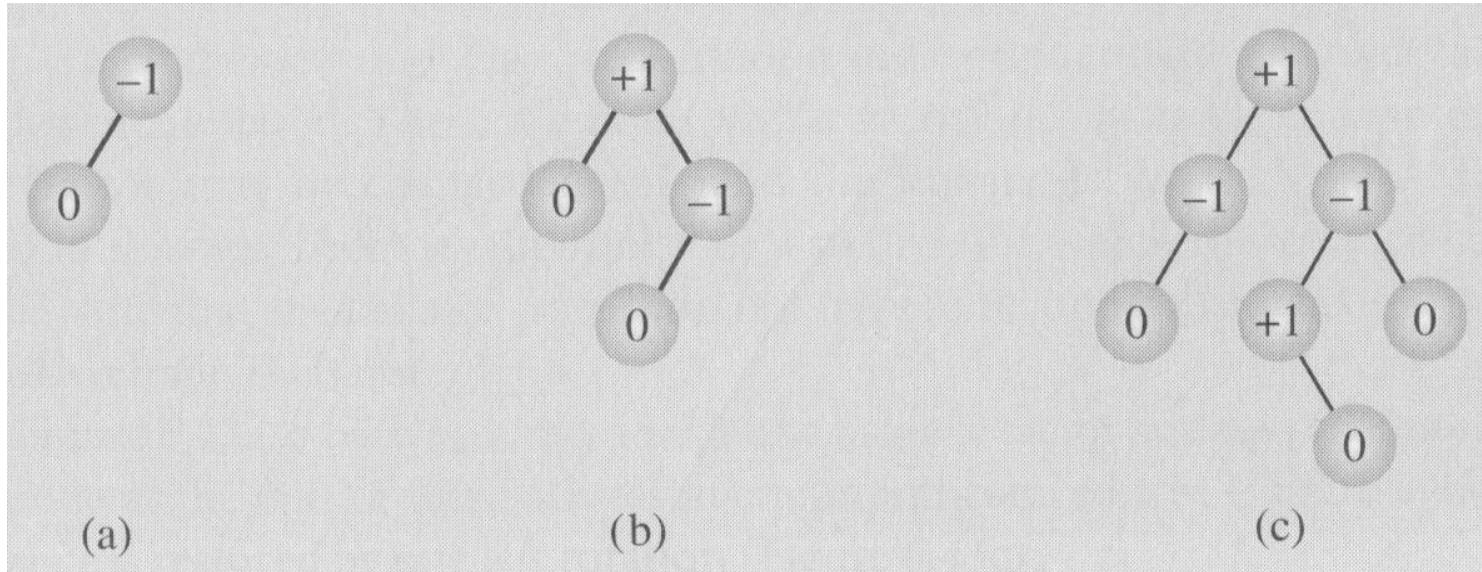
AVL definicijsko pravilo propisuje faktore ravnoteže (*balance factors*) čvorova u stablu.

faktor ravnoteže FR =

(visina desnog podstabla) – (visina lijevog podstabla)

(može i obratno, ali ovo je uobičajeno)

AVL definicijsko pravilo: faktori ravnoteže svih čvorova moraju biti  $-1$ ,  $0$  ili  $1$ .



Broj čvorova u  $h$  razina ne može biti manji od nekog minimuma, a iz definicijskog pravila slijedi rekurzivna relacija kojom se taj minimum može izračunati:

$$\text{AVL}_h = \text{AVL}_{h-1} + \text{AVL}_{h-2} + 1,$$

gdje su  $\text{AVL}_0 = 0$  i  $\text{AVL}_1 = 1$  polazne vrijednosti.

Teorijski se dokazuju sljedeće granice visine AVL stabla u ovisnosti o broju (čvorova)  $n$   
(Drozdek: Appendix A.5):

$$\log_2(n+1) \leq h \leq 1,44 \cdot \log_2(n+2) - 0,328 .$$

Prema tome, složenost pretraživanja AVL stabla je  $\textcolor{red}{O(\log_2 n)}$ .

U savršenom stablu s istim brojem čvorova vrijedi  $h = \log_2(n+1)$ .

$\Rightarrow$  pretraživanje AVL stabla je u najgorem slučaju 44 % sporije od pretraživanja savršenog stabla

# Dodavanje čvora u AVL stablo:

## 1. naći mjesto i ubaciti čvor

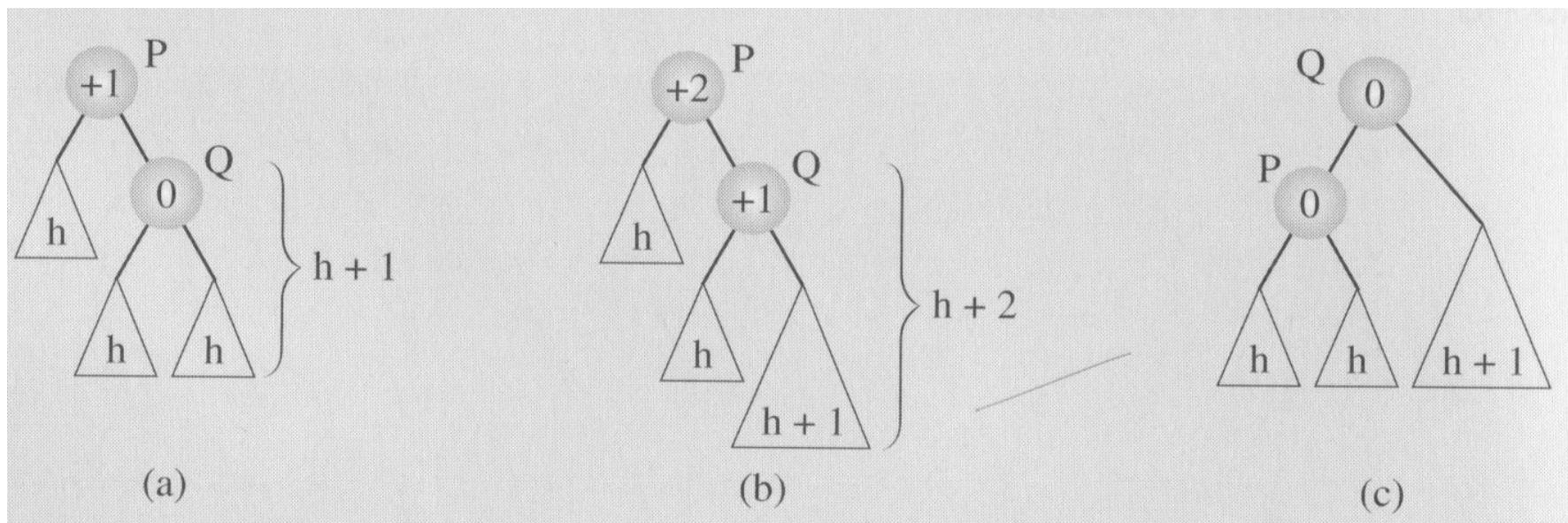
- pretraživanje AVL stabla isto je kao i pretraživanje običnog binarnog *search* stabla

## 2. uravnotežiti stablo

- od roditelja dodanog čvora krenuti prema korijenu i osvježavati faktore ravnoteže (visine podstabala)
- svakom čvoru na tom putu FR se može povećati ili smanjiti za 1 (ili ostati nepromijenjen), ovisno o tome kojem se njegovom podstabalu promijenila visina
- **prvi i jedini** koji će zahtijevati intervenciju bit će onaj kojemu osvježeni FR bude jednak  $-2$  ili  $2$
- moguća su četiri slučaja, po dva simetrična (čvor može ispasti iz ravnoteže samo ako je prethodno imao  $FR = \pm 1$ )

## Situacije 1 i 2 (simetrična)

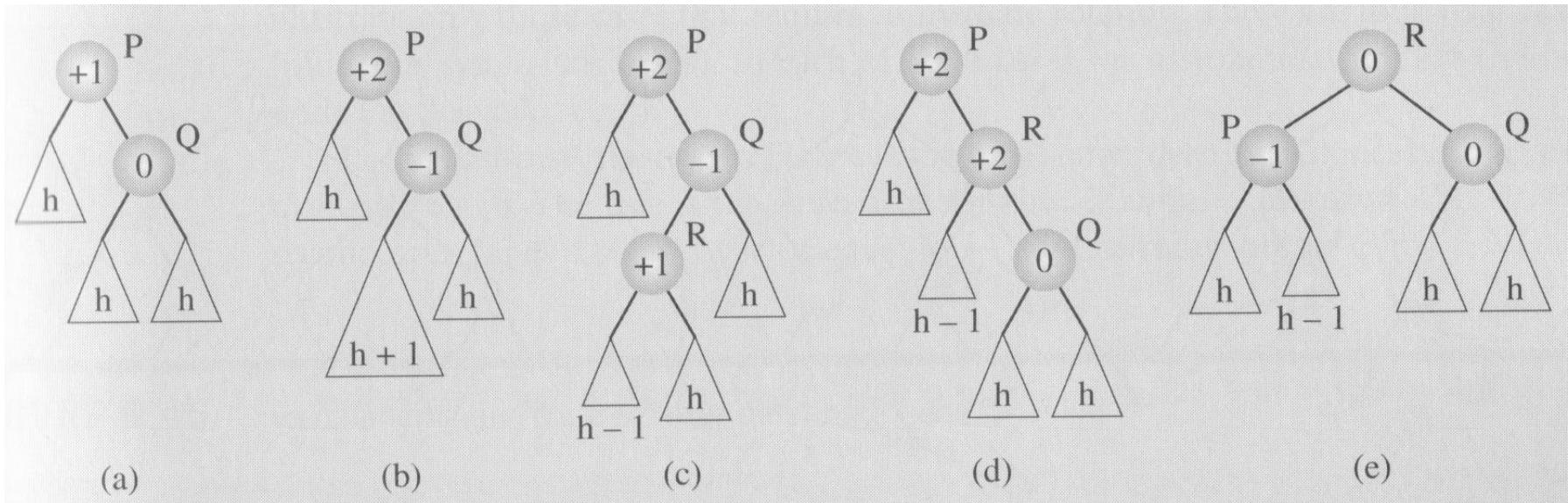
- dodavanje u desno (lijevo) podstablo desnog (lijevog) djeteta ( $P$  = roditelj,  $Q$  = dijete)
- prepoznavanje:  $1 \leftrightarrow FR(P) = 2 \text{ i } FR(Q) = 1$   
 $2 \leftrightarrow FR(P) = -2 \text{ i } FR(Q) = -1$



Rješenje: rotacija djeteta  $Q$  oko roditelja  $P$ .

## Situacije 3 i 4 (simetrična)

- dodavanje u lijevo (desno) podstablo desnog (lijevog) djeteta ( $P$  = roditelj,  $Q$  = dijete)
- prepoznavanje:
  - $3 \leftrightarrow \text{FR}(P) = 2 \text{ i } \text{FR}(Q) = -1$
  - $4 \leftrightarrow \text{FR}(P) = -2 \text{ i } \text{FR}(Q) = 1$



Rješenje:

- 1) rotacija  $R$  oko  $Q$  (slika d)
- 2) rotacija  $R$  oko  $P$  (slika e)

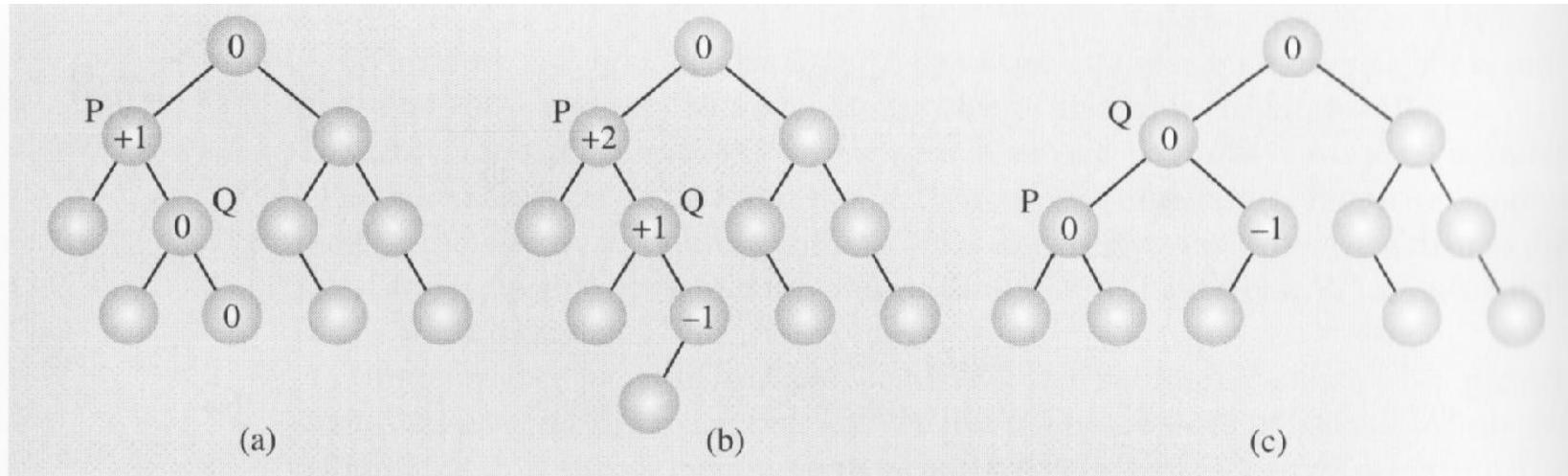
Treba li provjeravati stanje iznad P ili R?

Srećom NE - ukupna visina stabla u sva četiri slučaja ostaje ista ( $=h+2$ )!

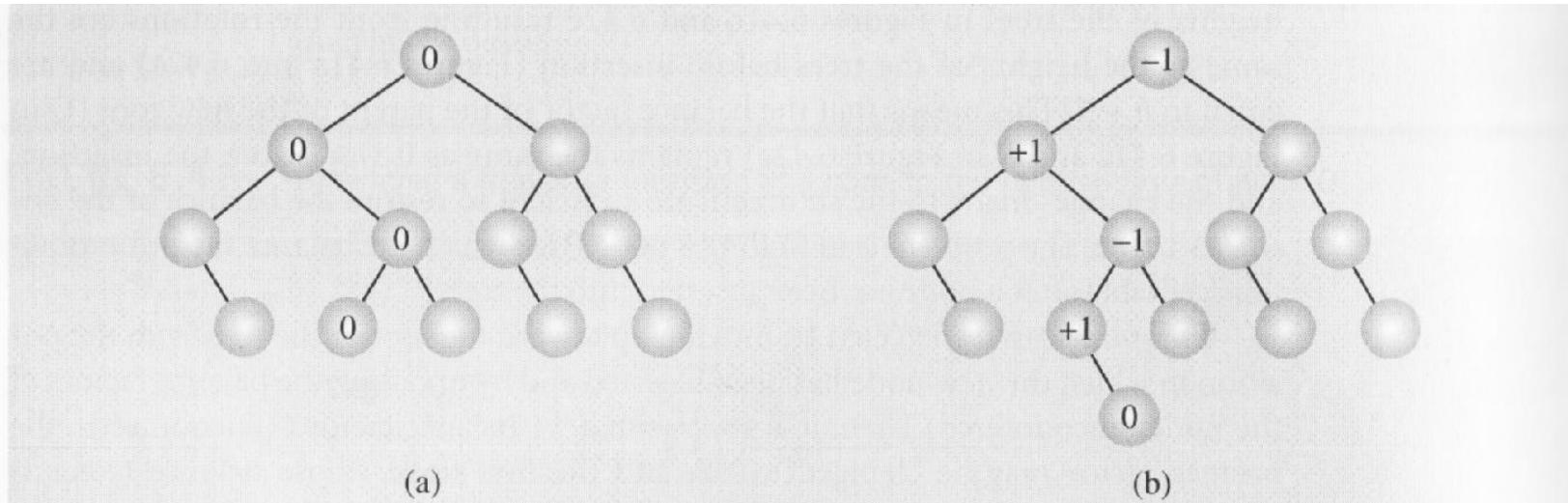
Kako pronaći korijen P podstabla narušene ravnoteže (onaj kojemu je novi FR =  $\pm 2$ )?

- uspinjati se od ubačenog čvora prema korijenu i osvježavati visine podstabala, odnosno faktore ravnoteže, svakog čvora na tom putu
- ako je novi  $FR(x) = 0$ , sve je u redu  $\rightarrow$  gotovo
- ako je novi  $FR(x) = \pm 1$ , nastaviti bez intervencije
- korijen P mogu (ali i ne moraju) postati samo čvorovi koji su prethodno imali faktor ravnoteže  $\pm 1$ , a novi je  $\pm 2$ , pa prvi takav postaje P

# Primjer traženja P:



Ako su svi faktori ravnoteže na putu =0, osvježavanje će biti potrebno sve do korijena, ali bez drugih intervencija.



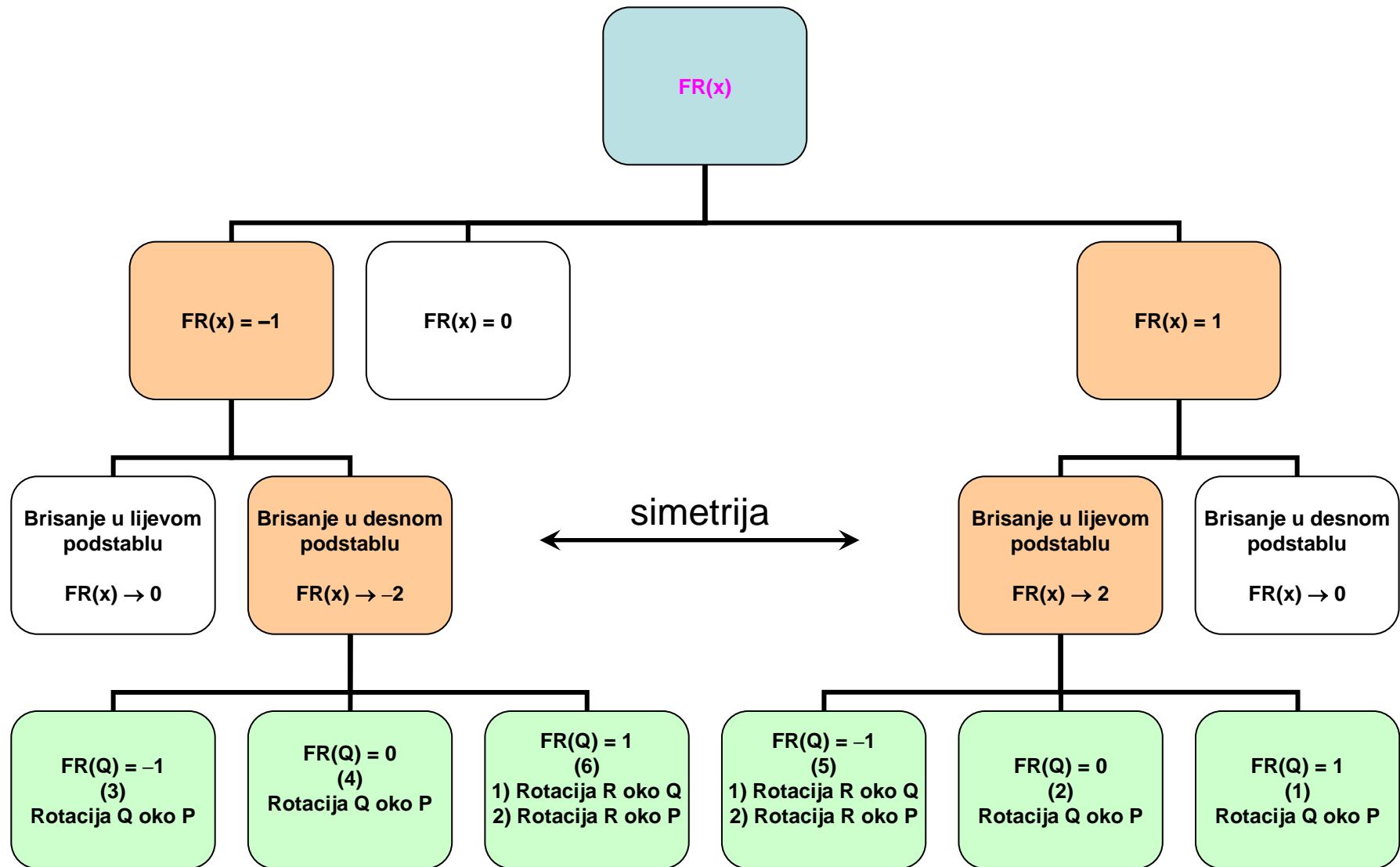
# Brisanje čvora u AVL stablu:

1. ukloniti čvor (*Deletion by Copying!*)
2. uravnotežiti stablo (ne samo do prvog neuravnoteženog!)
  - krenuti od roditelja zamjenskog čvora (čvora iz kojeg su prepisani podatci i koji je stvarno uklonjen iz stabla) prema korijenu i osvježavati faktore ravnoteže FR
  - ako je  $FR(x)$  postao  $\pm 1$ , znači da brisanje nije promijenilo visinu (pod)stabla kojemu je čvor x korijen (podstabla koja su započinjala djecom od x su bila jednaka, a jedno je sada niže)  $\rightarrow$  gotovo
  - ako je  $FR(x)$  postao  $=0$ , brisanje je izazvalo promjenu visine podstabla kojemu je x korijen (podstabla kojima su djeca od x korijeni bila su različite visine, a sada su jednaka) pa osvježavanje treba nastaviti, ali bez ikakve intervencije
  - ako je  $FR(x)$  postao  $\pm 2$ , potrebna je intervencija (tri slučaja + tri simetrična)
  - nakon intervencije, opet treba provjeriti  $FR(x)$  i ako je sada jednak  $\pm 1$ , osvježavanje je završeno;  
za novi  $FR = 0$ , nastaviti s osvježavanjem

## Situacije: promatramo čvor x (osvježen FR)

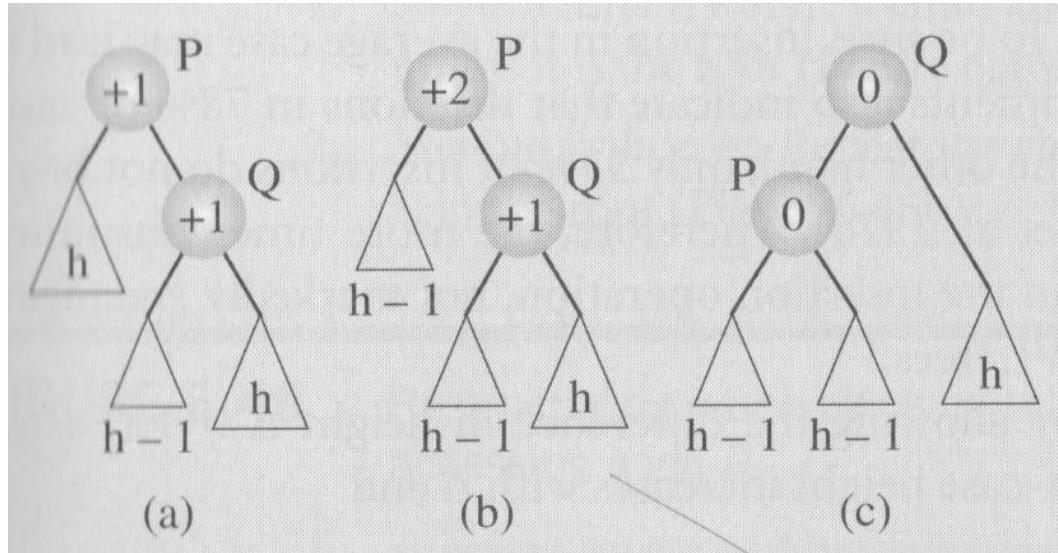
- ako je novi  $FR(x) = \pm 2$ , prije je morao biti  $FR(x) = \pm 1$ ; da je bio  $= 0$ , brisanje ga ne bi izbacilo iz ravnoteže
- brisanje se moglo dogoditi ili u lijevom ili u desnom podstablu od x
- brisanje u lijevom podstablu kada je  $FR(x) = -1$ , kao i brisanje u desnom podstablu kada je  $FR(x) = 1$ , prouzročit će  $FR(x) = 0$  i ta dva slučaja neće zahtijevati intervenciju (ali nastavljamo osvježavanje)
- brisanje u desnom podstablu kada je  $FR(x) = -1$  i brisanje u lijevom podstablu kada je  $FR(x) = 1$  prouzročit će  $FR(x) = \pm 2$  i zahtijevaju intervenciju; u oba slučaja postupak ovisi o djitetu u podstablu koje se nije mijenjalo, a budući da to dijete može imati FR  $-1$ ,  $0$  ili  $1$ , ukupno dobivamo šest različitih situacija

# Stablo odlučivanja za brisanje čvora u AVL stablu



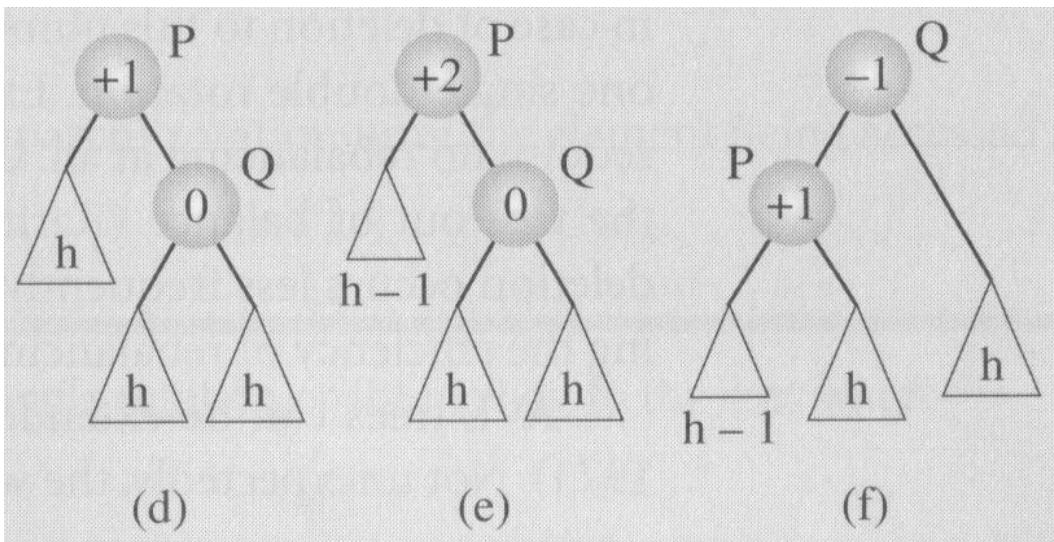
# Slučajevi 1 i 2 (3 i 4 simetrični):

- primjeri za početno stanje  $FR(x=P) = 1$  i  $FR(Q)=1$  (slike a,b,c) te  
 $FR(x=P) = 1$  i  $FR(Q)=0$  (slike d,e,f)



Prepoznavanje:

- 1  $\leftrightarrow FR(P)=2$  i  $FR(Q)=1$
- 2  $\leftrightarrow FR(P)=2$  i  $FR(Q)=0$
- 3  $\leftrightarrow FR(P)=-2$  i  $FR(Q)=-1$
- 4  $\leftrightarrow FR(P)=-2$  i  $FR(Q)=0$



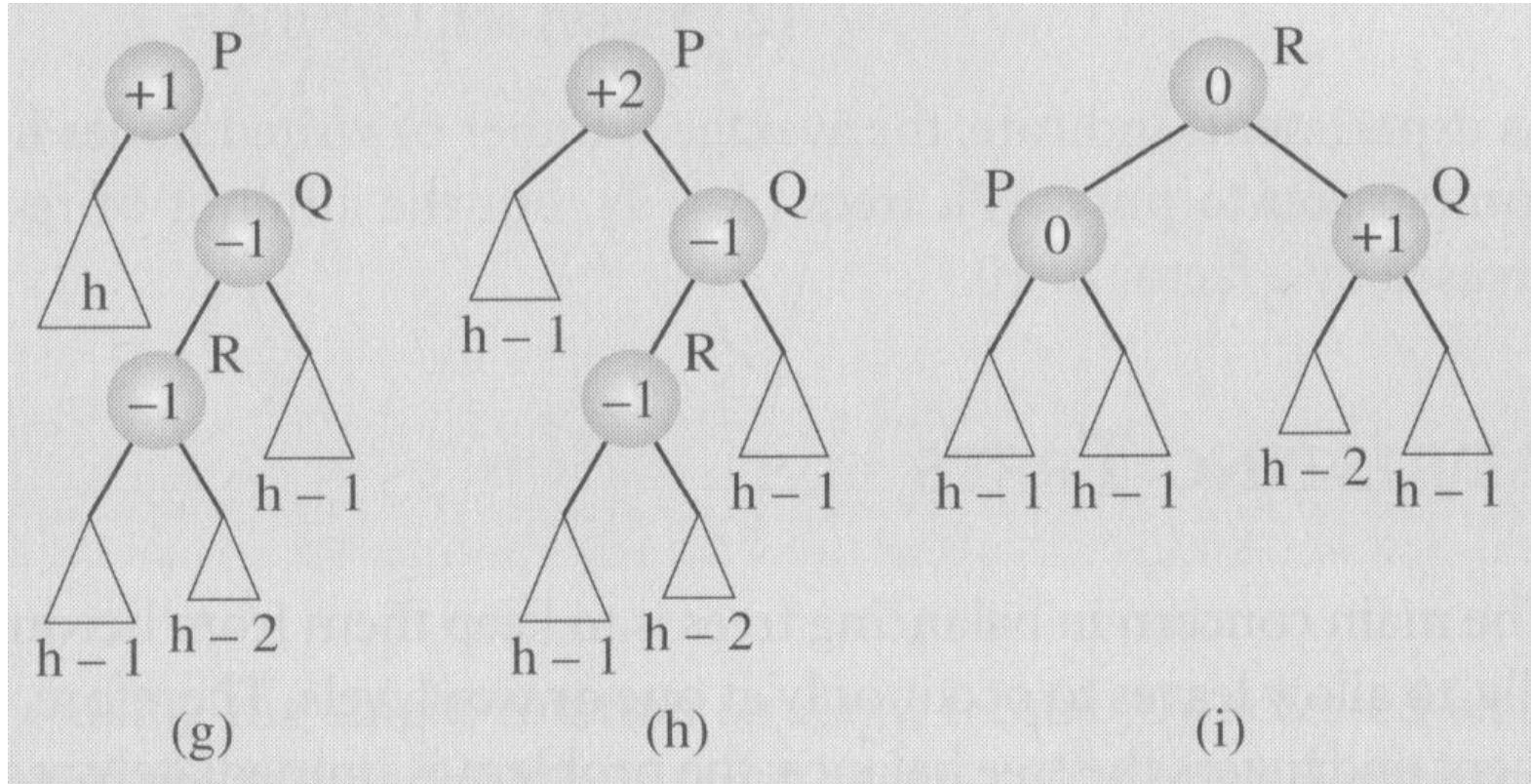
Rješenje:

rotacija Q oko P

- isto kao slučajevi 1 i 2 za dodavanje

## Slučaj 5 (6 simetričan):

- primjer za početno stanje  $FR(x=P) = 1$  i  $FR(Q) = -1$



Prepoznavanje:

$$5 \leftrightarrow FR(P)=2 \text{ i } FR(Q)=-1$$

$$6 \leftrightarrow FR(P)=-2 \text{ i } FR(Q)=1$$

Rješenje je neovisno o  $FR(R)$ :

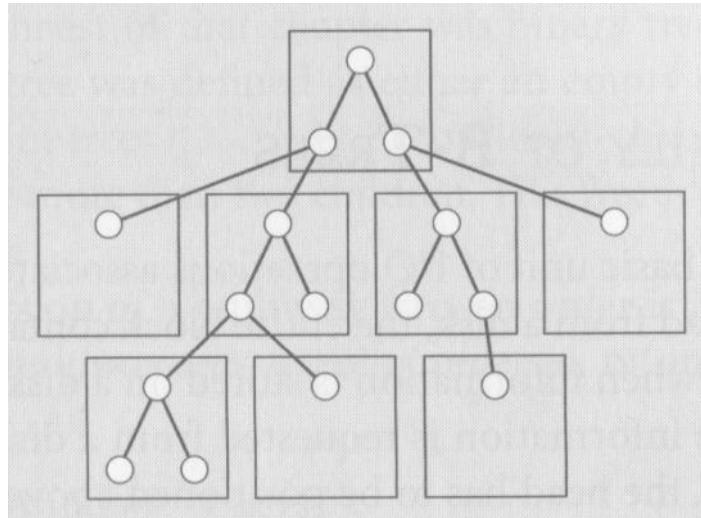
1. rotacija R oko Q
  2. rotacija R oko P
- isto kao slučajevi 3 i 4 za dodavanje

Još jedna vrsta uravnoteženih stabala su crveno-crna (*red-black*) stabla koja idejno proizlaze iz B-stabala 4. reda, ali su posebna vrsta, a ne inačica (varijanta) B-stabala.

# B-stabla

Vanjska memorija (disk) još uvijek ima nedostatke:

- namještanje (pozicioniranje) glave za čitanje/pisanje je relativno sporo (mehanička tromost)
- čitaju se cijeli blokovi podataka pa većinu pročitanih zapravo ne koristimo
- susjedni čvorovi (roditelji i djeca) stabala mogu biti “razasuti” u udaljene blokove pa je prijelaz iz roditelja u dijete spor usprkos njihovoj logičkoj blizini



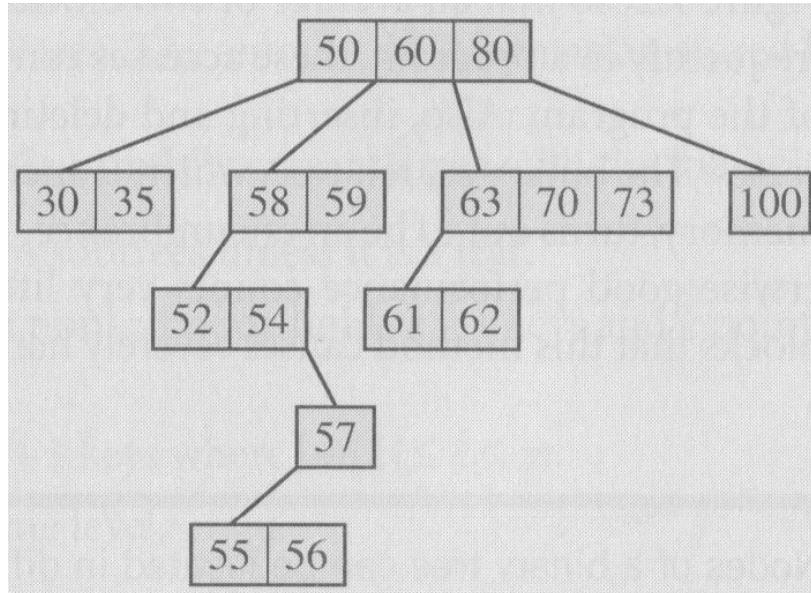
B-stabla ublažavaju posljedice ograničenja vanjskih jedinica povećanjem veličine čvorova u stablu koja se namješta na približno veličinu bloka na disku.  
B-stabla su podvrsta M-stabala.

**M (*multiway*) stabla** - stabla u kojima čvorovi mogu imati proizvoljan broj djece.

**M-stablo  $m$ -tog reda:** M-stablo u kojem čvorovi mogu imati najviše  $m$  djece.

M-search-stablo  $m$ -toga reda je M-stablo sa sljedećim dodatnim svojstvima:

1. svaki čvor ima najviše  $m$  djece i  $m-1$  podataka (ključeva)
2. ključevi u čvorovima su sortirani
3. ključevi u prvih  $i$  djece nekog čvora su manji od  $i$ -toga ključa promatranočvora
4. ključevi u zadnjih  $m-i$  djece nekog čvora su veći od  $i$ -toga ključa promatranočvora



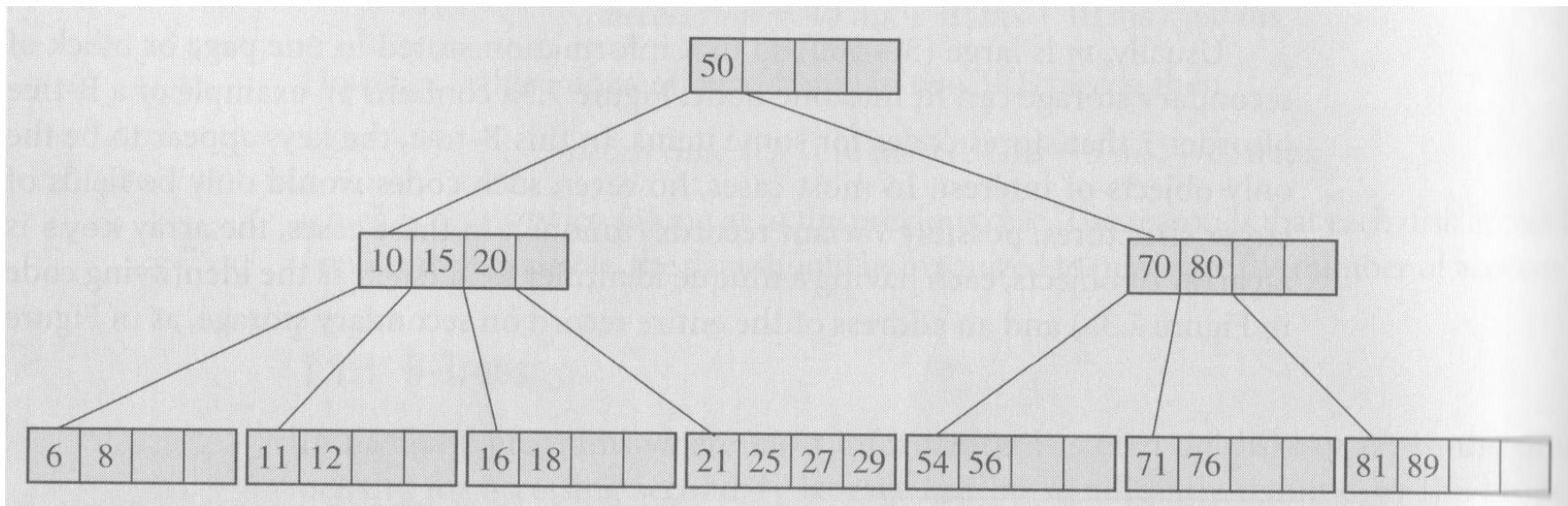
B-stablo  $m$ -tog reda je M-search-stablo sa sljedećim dodatnim svojstvima:

1. korijen ima najmanje dvoje djece, osim ako je ujedno i list (jedini čvor u stablu)
2. svaki čvor, osim korijena i listova, sadrži barem  $k-1$  ključeva i  $k$  pokazivača na podstabla (ima  $k$  djece), pri čemu je  $m/2 \leq k \leq m$  (ako rezultat dijeljenja nije cijeli broj, uzima se najmanji veći cijeli broj)
3. svi listovi sadrže barem  $k-1$  ključeva, pri čemu je  $m/2 \leq k \leq m$  (ako rezultat dijeljenja nije cijeli broj, uzima se najmanji veći cijeli broj)
4. svi listovi su na istoj razini

Sva ta svojstva ostvaruju se posebnim načinom održavanja B-stabla i nisu “prirodna” posljedica logičke apstrakcije.

Čvor B-stabla uobičajeno se realizira kao struktura (klasa) s poljem od  $m-1$  ključeva, poljem od  $m$  pokazivača i još ponekim dodatnim podatkom za olakšavanje održavanja stabla, kao npr. brojem ključeva (upisanih podataka) u čvoru ili oznake list/ne-list itd.

Radi preglednosti, u grafičkim prikazima B-stabala ti dodatni podatci se izostavljaju.



Zbog definicijskih svojstava, B-stabla imaju dvije važne osobitosti:

- popunjenoš im je barem 50 %
- savršeno su uravnotežena (to se postiže posebnim načinom dodavanja novih čvorova)

Algoritam pretraživanja B-stabla:

1. uči u čvor (na početku korijen) i redom pregledavati ključeve sve dok je trenutačni manji od traženog, a još ima neprovjerenih
2. ako je 1. korak završio zbog nailaska na ključ veći od traženog ili zbog dolaska do kraja čvora, spustiti se razinu niže (u odgovarajuće dijete) i nastaviti od koraka 1; ako nema niže razine, nema ni traženog ključa

## Moguće ostvarenje (polja se koriste od indeksa =1, a ne =0):

```
SearchBTree (key, node)
if (node != NULL)
{   for (i=1; i<=node->keyNum    //keyNum je član čvora
        && node->keys[i]<key; ++i);
    if (i>node->keyNum || node->keys[i]>key)
        SearchBTree (key, node->pointers[i]);
    else
        return node; }
else
    no searched key;
```

Napomena: prvi uvjet u prvoj petlji se oslanja na varijablu ‘keyNum’ koja bi trebala biti članska varijabla čvora i sadržavati broj podataka (ključeva) upisanih u čvor (jer struktura čvora predviđa mjesto za  $m-1$  ključeva, a ne moraju sva mesta biti popunjena). Također, drugi uvjet koristi polje ‘keys’ koje bi trebalo biti spremnik (popis) ključeva upisanih u čvor.

## Dodavanje podataka u B-stablo:

- za razliku od *top-down* izgradnje običnih stabala, B-stablo je jednostavnije graditi odozdo prema gore

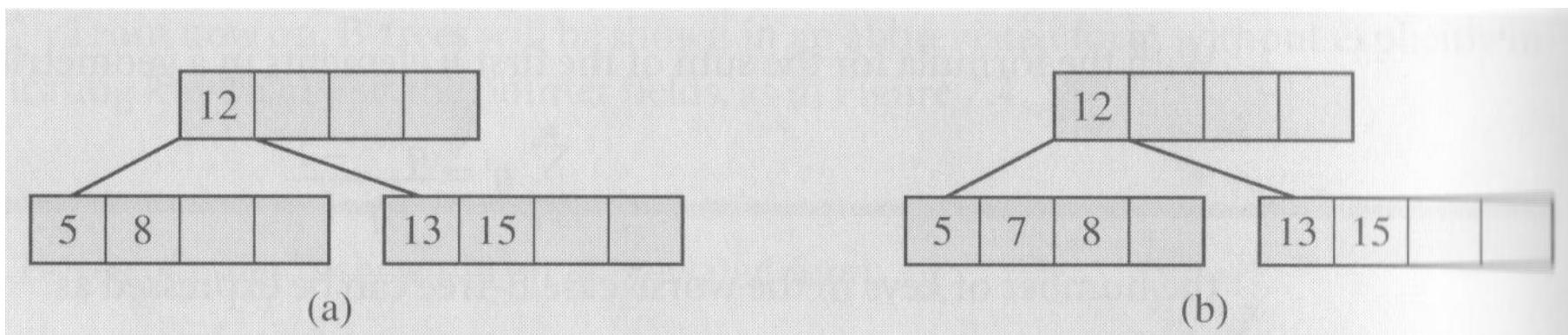
### Algoritam dodavanja podatka u B-stablo:

1. pronaći list u koji bi trebalo smjestiti novi podatak
2. ako ima mesta, upisati novi podatak
3. ako je taj list pun, “rascijepiti” ga
  - napraviti novi list, ravnomjerno razdijeliti elemente (podatke) između ta dva čvora, a središnji upisati u roditelja ako u roditelju ima mesta za njega
4. ako je i roditelj pun, “rascijepiti” i roditelja
  - ponavljati proceduru iz koraka 3 sve dok se ne dođe do čvora u kojem ima mesta ili do korijena
5. ako je i korijen pun, “rascijepiti” ga i napraviti novi korijen

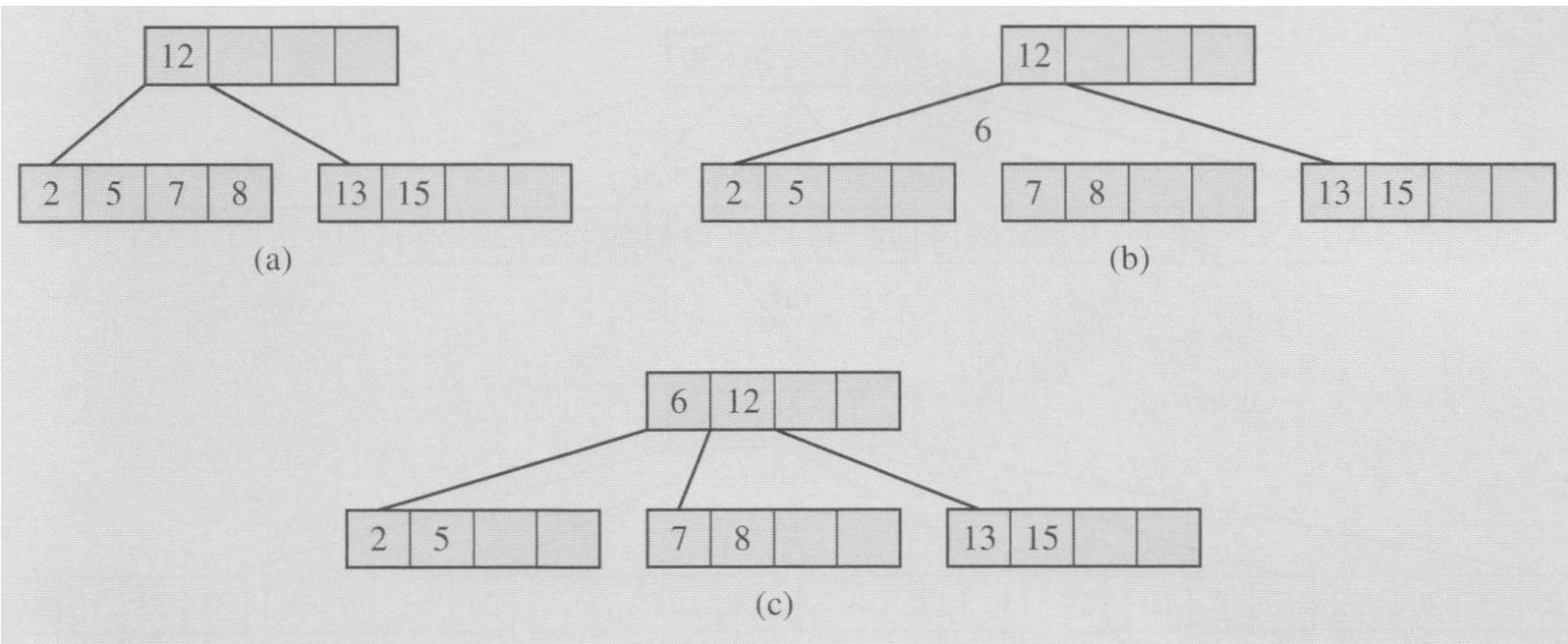
Prilikom ubacivanja novog podatka moguće su tri karakteristične situacije:

1. list u koji treba ići novi element nije pun
  - ubaciti novi element u taj list na odgovarajuće mjesto, pomicući po potrebi prethodni sadržaj

Primjer: dodavanje 7

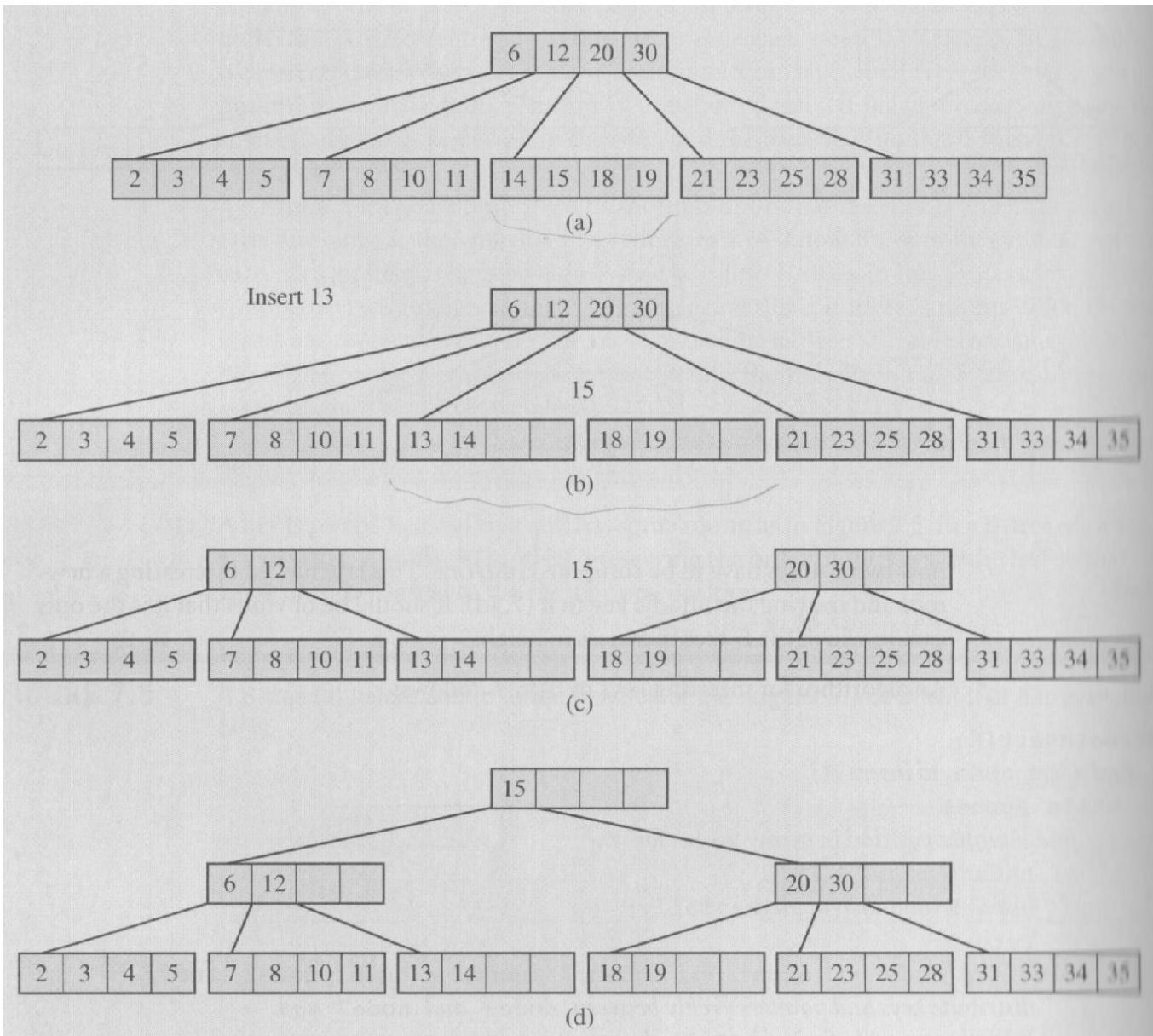


2. list u koji treba ići novi element je pun, ali korijen stabla nije (primjer: dodavanje 6)
- dovoljno je riješiti slučaj kad je list pun, a roditelj nije jer se to samo ponavlja, najviše do korijena
  - list se dijeli, tj. stvara se novi čvor i svi elementi se ravnomjerno raspoređuju, s time da se središnji (diobeni) element upisuje u roditelja

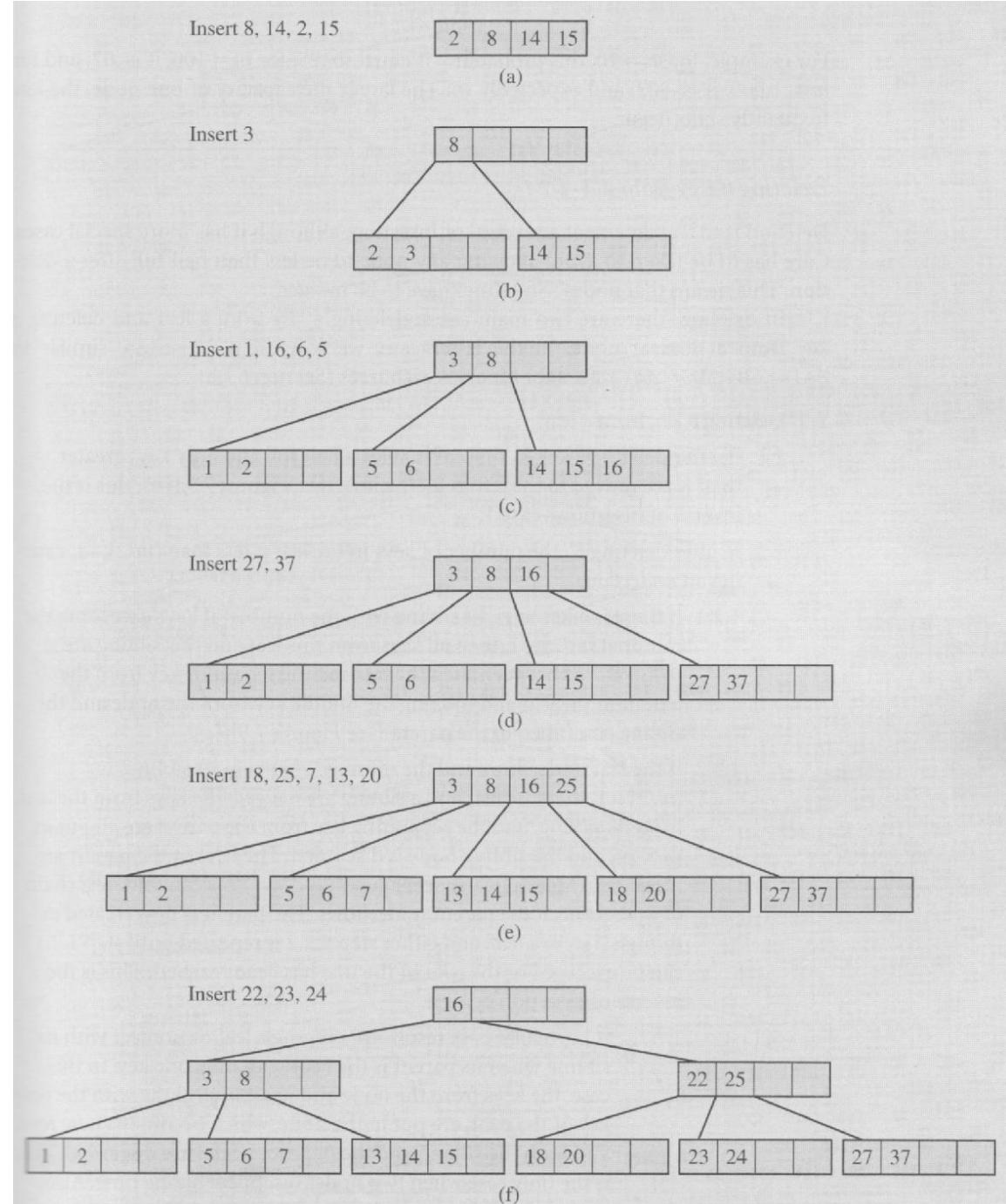


3. list u koji treba ići novi element je pun, a isto tako i korijen stabla
  - složenija inačica 2. slučaja
  - kad se razdijeli korijen, nastaju dva B-stabla koja treba sjediniti
  - sjedinjenje se postiže stvaranjem još jednog čvora koji će biti novi korijen i upisivanjem središnjeg (diobenog) elementa u njega
  - to je jedini slučaj koji završava povisivanjem stabla i zahvaljujući takvom postupku, B-stablo je uvijek savršeno uravnoteženo

# Primjer za 3. slučaj: dodavanje 13



Primjer izgradnje  
B-stabla:  
redom se dodaju 8,  
14, 2, 15, 3, 1, 16,  
6, 5, 27, 37, 18, 25,  
7, 13, 20, 22, 23 i  
24.



BTreeInsert (K)

*find a leaf node to insert K;*

while (true)

*find a proper position in array keys for K;*

if node is not full //case 1

*insert K and increment keyNum;*

return;

else //case 2, case 3

*split node into node1 and node2; // node1 = node, node2 is new*

*distribute keys and pointers evenly between node1 and node2*

*and initialize properly their keyNum's;*

K = middle key; //ideally, K is median

if node was the root //case 3

*create a new root as parent of node1 and node2;*

*put K and pointers to node1 and node2 into the root,*

*and set its keyNum to 1;*

return;

else

node = its parent; //case 2; now process the node's parent

## Brisanje elemenata (podataka) u B-stablu

- dva slučaja

1. brisanje elementa u listu stabla

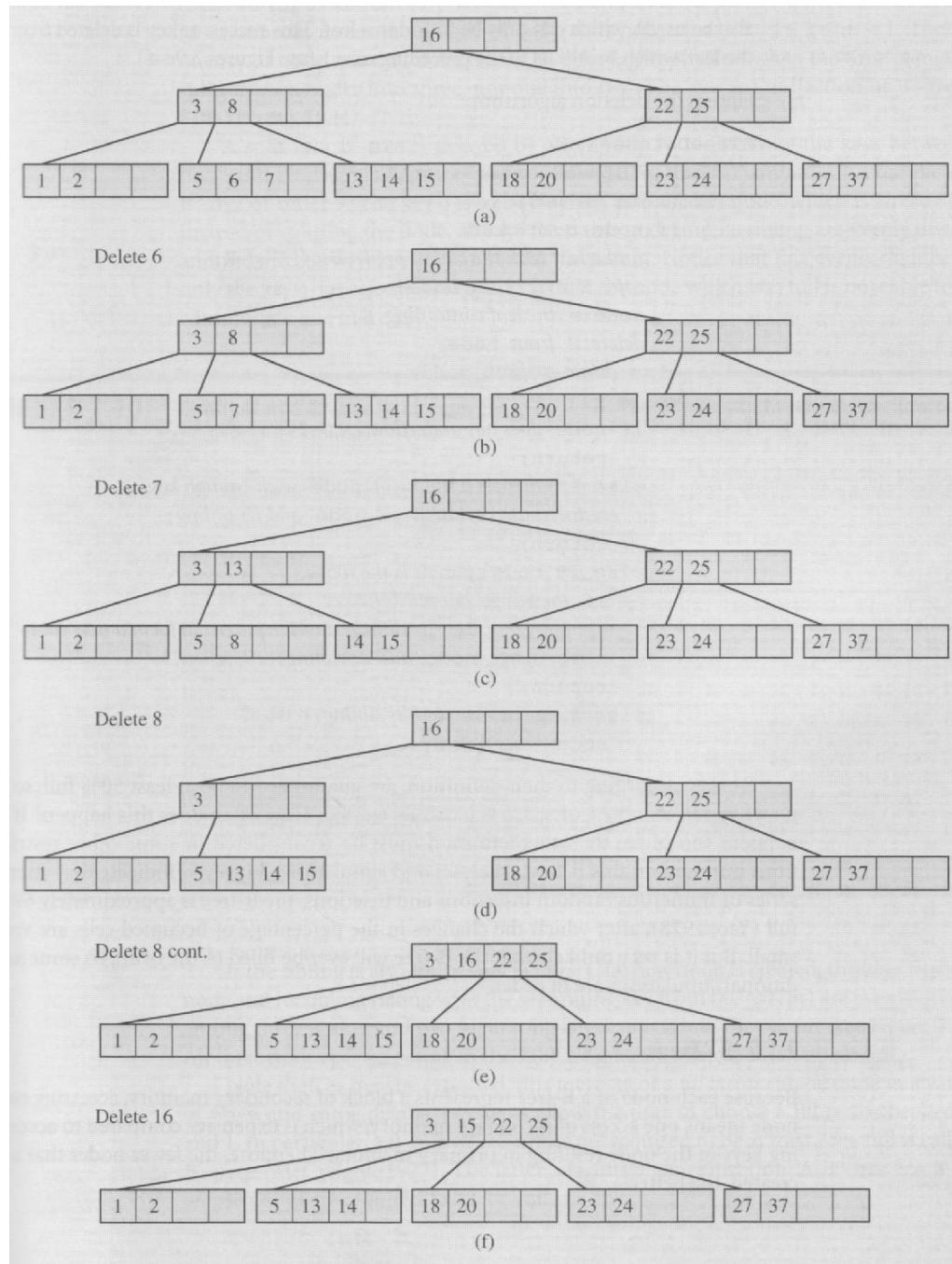
2. brisanje elementa u čvoru koji nije list stabla

- samo po sebi komplikirano zbog restrukturiranja stabla
- svodi se na brisanje elementa iz lista
  - na mjesto elementa koji treba izbrisati upisuje se njegov neposredni prethodnik (ili sljedbenik), a taj može biti samo u listu
  - potom se u listu briše prepisani element standardnim postupkom za brisanje elementa lista  
(primjer: slika e-f u nastavku)

# Brisanje elementa u listu B-stabla

1. list i nakon brisanja elementa ima još barem  $m/2$  ključeva; gotovo
2. broj preostalih elemenata  $<m/2$ 
  - 2.1 ako lijevo ili desno postoji susjed s više od  $m/2$  ključeva
    - elemente lista, tog susjeda i diobeni element iz roditelja ravnomjerno rasporediti u list i susjeda, a kao novi diobeni element u roditelja upisati središnji element ujedinjenog skupa (unije) elemenata (slika b-c); → gotovo
    - to je inverz (obrat) 1. slučaja dodavanja elementa u B-stablo
  - 2.2 sada znamo da susjedi (barem jedan postoji!) imaju točno  $m/2$  ključeva
    - list i susjed se sjedinjuju; svi elementi lista, susjeda i diobeni element iz roditelja upisuju se u list, a susjed se briše iz stabla (slika c-d);
    - to je inverz 2. slučaja dodavanja elementa u B-stablo i može izazvati lančano rasprostiranje te situacije ako sada roditelj ima manje od  $m/2$  elemenata; u tom slučaju se postupak ponavlja postupajući s roditeljem kao s listom sve dok se ne dođe u situaciju 2.1 ili do korijena stabla
- 2.2.1 postupkom 2.2 došli smo do korijena (roditelj je korijen stabla)
  - ako (korijen ima više od jednog elementa)sjediniti trenutačni čvor i susjeda kao u situaciji 2.2; → gotovo
  - inače (dakle korijen ima samo jedan element)sve elemente lista, susjeda i korijena upisati u jedan čvor koji postaje novi korijen, a dva čvora se brišu iz stabla (slika e);  
to je inverz 3. slučaja dodavanja elementa u B-stablo; → gotovo

# Primjer:



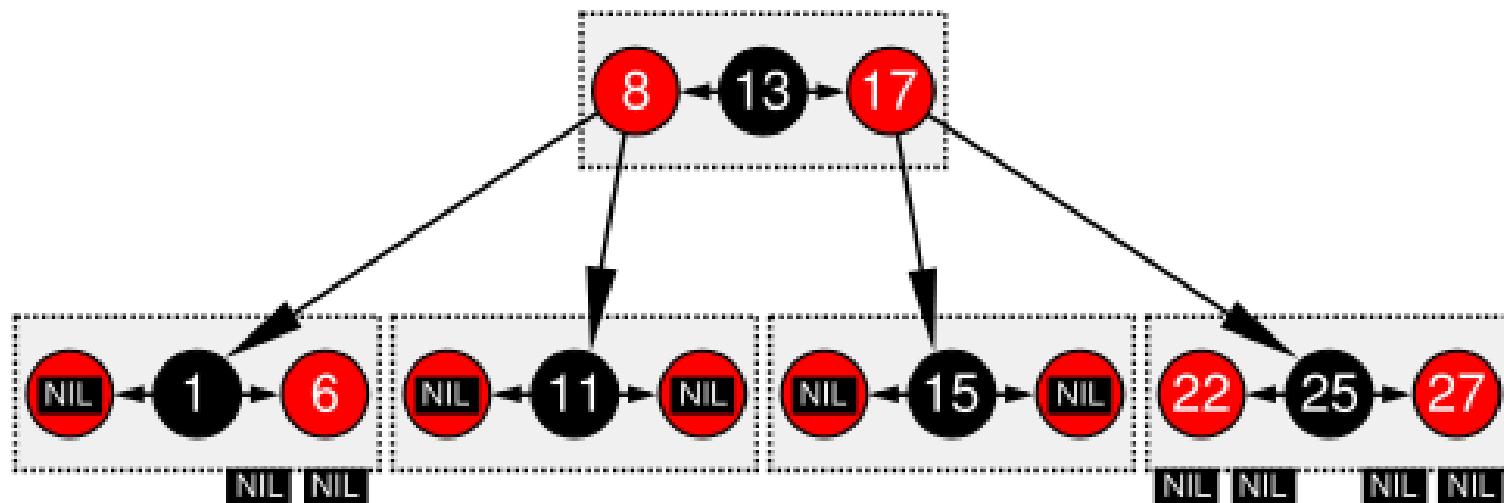
## BTreeDelete (K)

```
node = SearchBTree (K, root); //naći čvor s podatkom za brisanje (K)
if (node != NULL)
    if node is not a leaf
        find a leaf with the closest predecessor S of K;
        copy S over K in node; //ie. replace K with S in node
        node = the leaf containing S; //redirect node
        delete S from node; //delete S in leaf
    else
        delete K from node;
    while (node underflows) //u protivnom imamo slučaj 1
        if the first left or right sibling of node has more than m/2 keys
            redistribute keys between node and its sibling; //slučaj 2.1
            return;
        //u nastavku znamo da susjedi imaju točno m/2 elemenata
    else if node's parent is the root //slučaj 2.2.1
        if the parent has only one key
            merge node, its sibling, and the parent to form a new root;
            return;
        else
            merge node and its sibling; //kao slučaj 2.2,
            return; //samo bez nastavka
    else
        merge node and its sibling; //slučaj 2.2
        node = its parent;
```

# Crveno-crna stabla (Red-Black Trees)

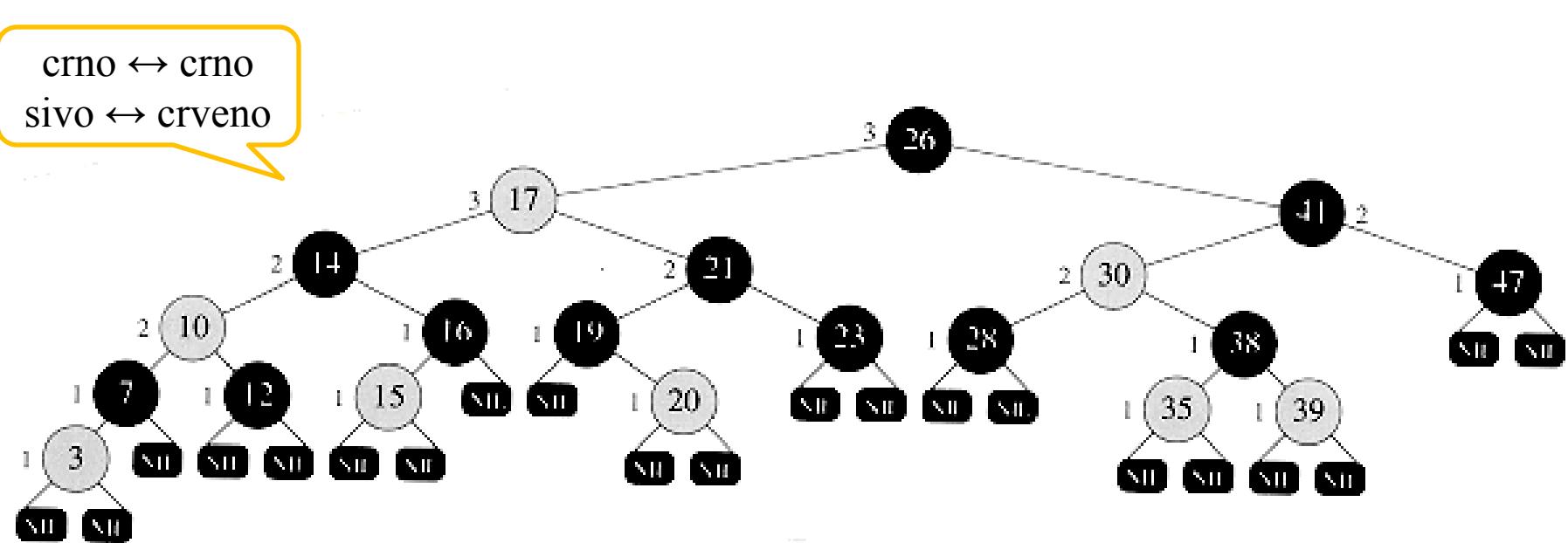
- prilagodba B-stabla smještaju u memoriju računala
  - manji utrošak memorije, a zadržava se uravnoteženost

**Crveno-crno stablo je binarno stablo** koje proizlazi iz B-stabla 4. reda ako mu se elementi čvorova smatraju obojanima prema strogim pravilima.



## Definicijska pravila (prema Cormen, Leiserson, Rivest, Smith):

1. svaki čvor je crven ili crn
2. korijen je crn (neobavezno, ali uobičajeno)
3. svaki list (čvor koji **ne** sadrži informaciju!) je crn
4. oba potomka crvenog čvora su crna
5. svaka staza od nekog čvora do (bilo kojeg) lista koji je njegov potomak prolazi istim brojem crnih čvorova



Listovi u crveno-crnom (RB) stablu ne sadrže informacije pa ne moraju ni postojati, nego roditelji mogu imati NULL pokazivače ili svi pokazivati isti poseban čvor, *sentinel*. Druga varijanta olakšava programiranje, npr. brisanje čvora.

Razlikujemo crvenu i crnu visinu stabla  
(*red and black height*;  $rh(x)$  i  $bh(x)$ ).

$rh(x)$ ,  $bh(x)$  = broj crvenih (crnih) čvorova na putu od čvora x (x se nebroji) do lista koji mu je potomak.

Crveno-crno (RB) stablo približnu uravnoteženost “nasljeđuje” od B-stabla, tj. osiguravaju ju definicijska pravila, a rezultat je brzo pretraživanje.

Učinak definicijskih pravila na uravnoteženost očituje se u sljedećem svojstvu RB stabla:

najduži put od korijena do nekog lista najviše je dvostruko duži od najkraćeg puta od korijena do nekog (drugog) lista, tj. najduži put je najviše dvostruko duži od najkraćeg.  $\square$

*Dokaz:* zbog 4. pravila, nijedan put ne može prolaziti uzastopno dvama crvenim čvorovima. Nadalje, najkraći put imao bi samo crne čvorove, dok bi se u najdužem crveni i crni izmjenjivali u svakom koraku. Budući da, zbog 5. pravila, najduži i najkraći put imaju jednak broj crnih čvorova, a najduži može imati još najviše onoliko crvenih čvorova koliko ima crnih, slijedi da najduži put može biti najviše dvostruko dulji od najkraćeg. ■

Brzinu pretraživanja jamči sljedeći

**Teorem:** Visina RB-stabla s  $n$  unutarnjih čvorova je

$$h \leq 2 \cdot \log_2(n+1). \quad \square$$

*Dokaz:* binarno stablo visine  $h$  ima najviše  $n = 2^h - 1$  čvorova. Zbog 4. pravila, barem polovica visine je crna visina pa je  $hb \geq h/2$ . Budući da je  $n$  veći od ili jednak broju crnih čvorova na putu od korijena do najnižeg lista, slijedi  $n \geq 2^{hb} - 1 \geq 2^{h/2} - 1$ , a iz toga izravno  $h \leq 2 \cdot \log_2(n+1)$ . ■

Pretraživanje binarnog stabla je složenosti  $O(h)$  pa je složenost pretraživanja RB-stabla  **$O(\log_2 n)$** .

- dodavanje i brisanje čvorova su također  $O(\log_2 n)$  operacije
- AVL stabla su strože uravnotežena (niža) pa je pristup podatcima (neznatno) brži, ali im je održavanje (neznatno) komplikiranije (sporije dodavanje i brisanje čvorova)

## Dodavanje čvora u RB-stablo:

- radi lakše analize, uvode se pojmovi čvor-ujak (*uncle*) koji se označava s U, a znači *sibling* roditelja promatranog čvora (roditeljev brat/sestra), i čvor-djed koji se označava s G (*grandfather*), a znači roditelj roditelja
1. ubaciti novi čvor kao u svako drugo binarno *search* stablo i pridijeliti mu **crvenu** boju
  2. restrukturirati stablo (primjenom rotacija i bojanjem čvorova) da bi zadovoljilo definicijska pravila

Koja pravila i kada se mogu narušiti dodavanjem novog čvora i njegovim bojanjem u crveno?

- očito, pravila 1 i 3 će uvijek biti zadovoljena
- pravilo 5 (crne visine jednake) također ostaje zadovoljeno jer novi (crveni) čvor ne mijenja crnu visinu

Ugrožena su jedino pravila 2 (korijen je crn) i 4 (crveni ima crnu djecu) i to:

- a) pravilo 2 ako je novi čvor korijen
- b) pravilo 4 ako je roditelj novog čvora crven.

Važno je uočiti da istodobno može biti prekršeno samo jedno od tih pravila jer:

- ako je prekršeno pravilo 2, novi čvor je (crveni) korijen, a kako su djeca novog automatski crna (sentinel), četvrto pravilo ne može biti prekršeno
- ako je prekršeno pravilo 4, roditelj novog čvora je crven pa sigurno nije korijen i pravilo 2 nije prekršeno.

## Restrukturiranje stabla nakon dodavanja čvora

- novi čvor N, roditelj P, djed G, ujak U

Stablo se restrukturira u jednoj petlji koja redom obavlja navedene provjere i rješava nepravilnosti kako je opisano u nastavku.

### 1. novi čvor je korijen

- samo ga prebojati u crno i gotovo;
  - 5. pravilo ostaje zadovoljeno jer je to dodatni crni čvor u svim putevima u stablu

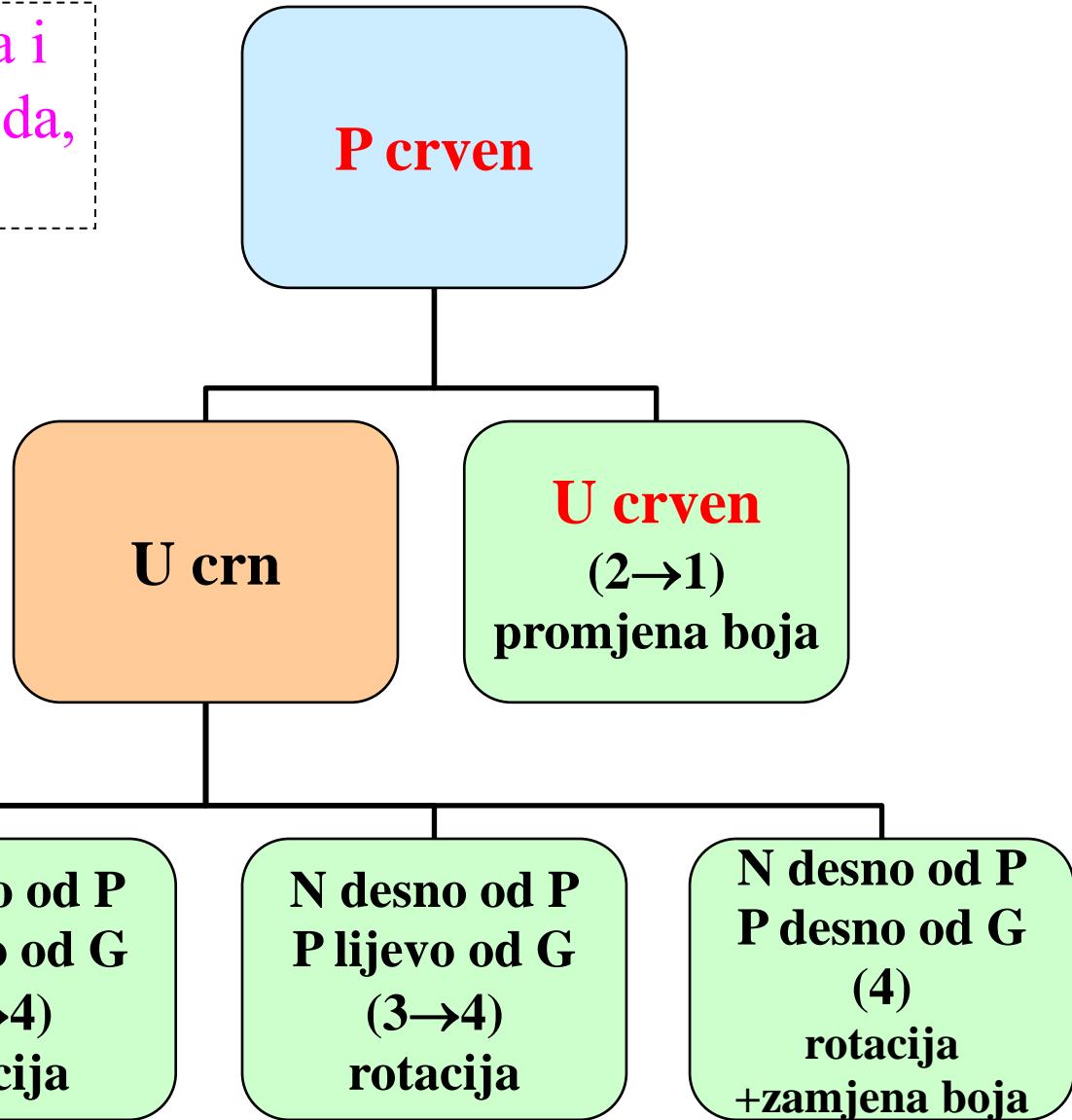
Ako novi čvor nije korijen, onda može biti narušeno samo

4. pravilo, tj. ako RB svojstva stabla uopće jesu narušena, onda je to zato što je roditelj novog čvora također crven.

Kao takav, roditelj ne može biti korijen pa zaključujemo da tada sigurno postoji i djed, a budući da ima crveno dijete, djed je crn.

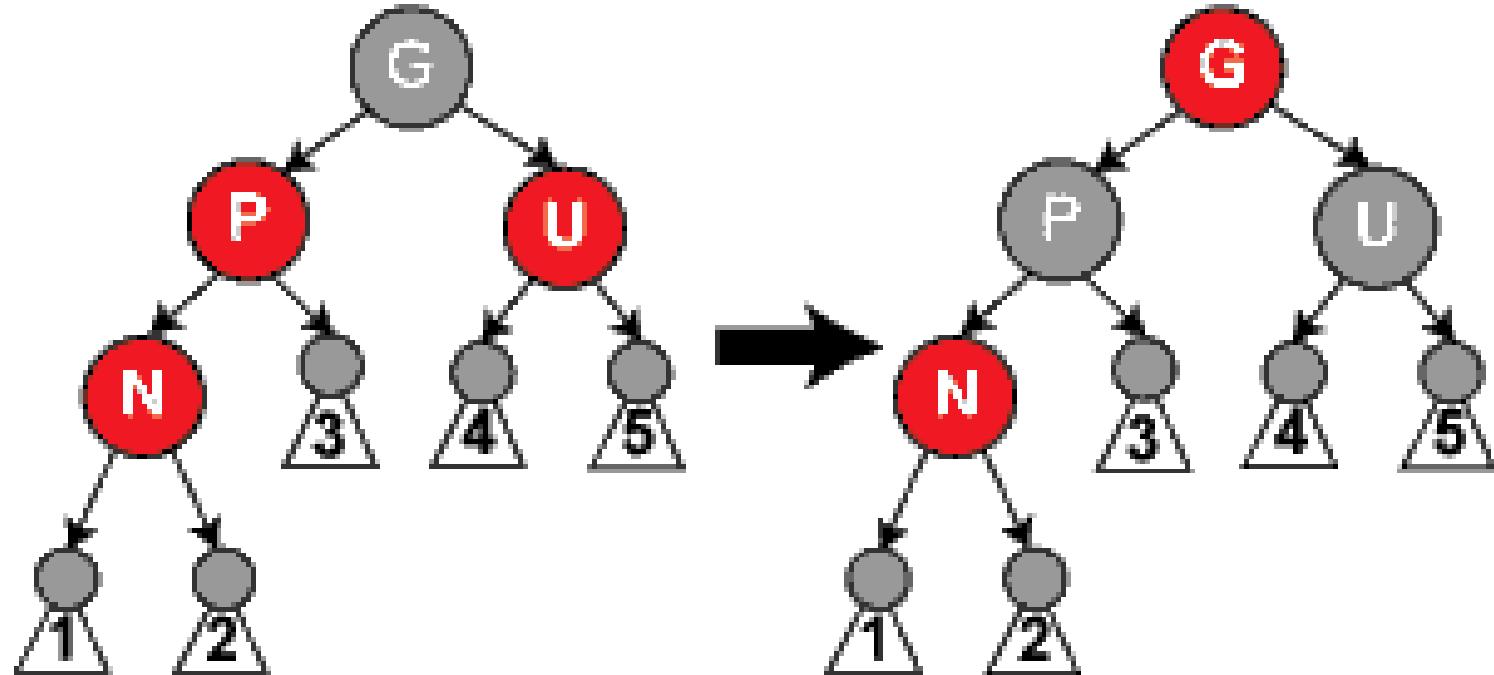
Ostaje razmotriti situacije u kojima je P crveni čvor:

Rješenja ovise o boji ujaka i međusobnom položaju djeda, roditelja i novog čvora.



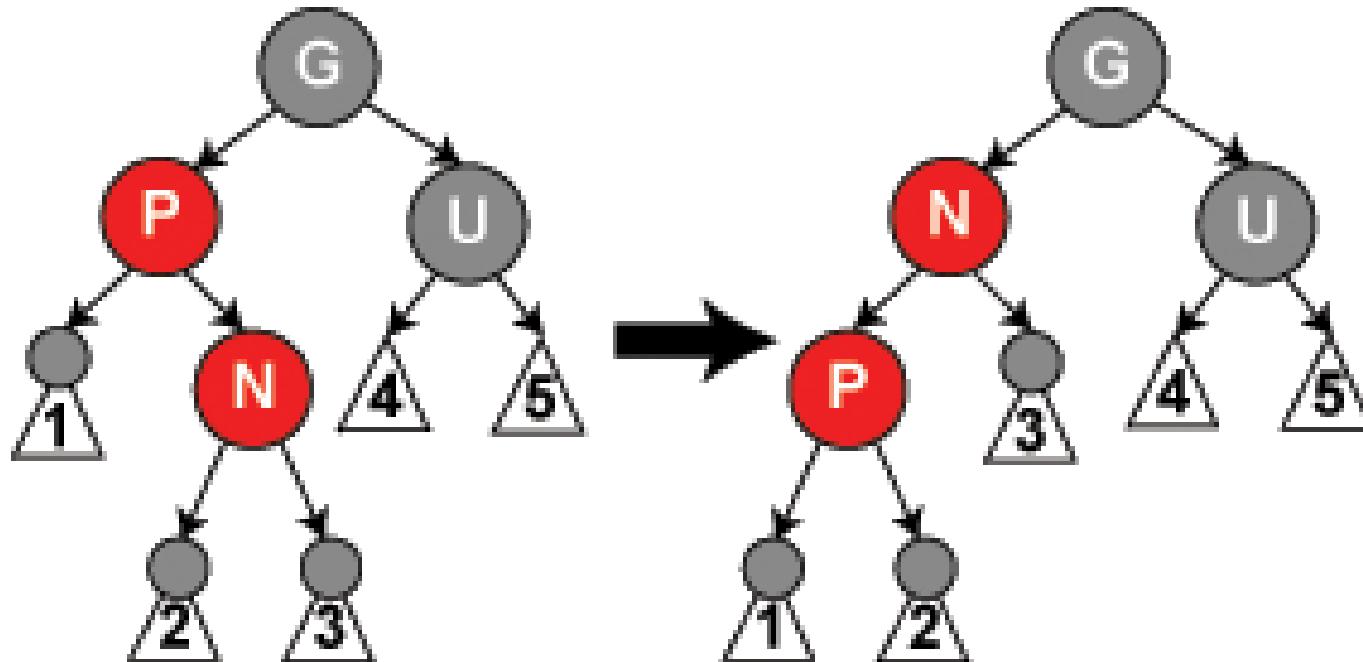
## 2. roditelj (sigurno) i ujak su crveni

- narušeno 4. pravilo (potomak crvenog P je crveni N)
- prebojati P i U u crno (time se rješava problem s 4. pravilom), a G u crveno (radi očuvanja 5. pravila)
  - sada G može narušavati 4. pravilo ako ima crvenog roditelja ili 2. pravilo ako je korijen; viša razina  $\Rightarrow$  konvergencija!
- vratiti se na korak 1 promatrajući G kao novi čvor (N)



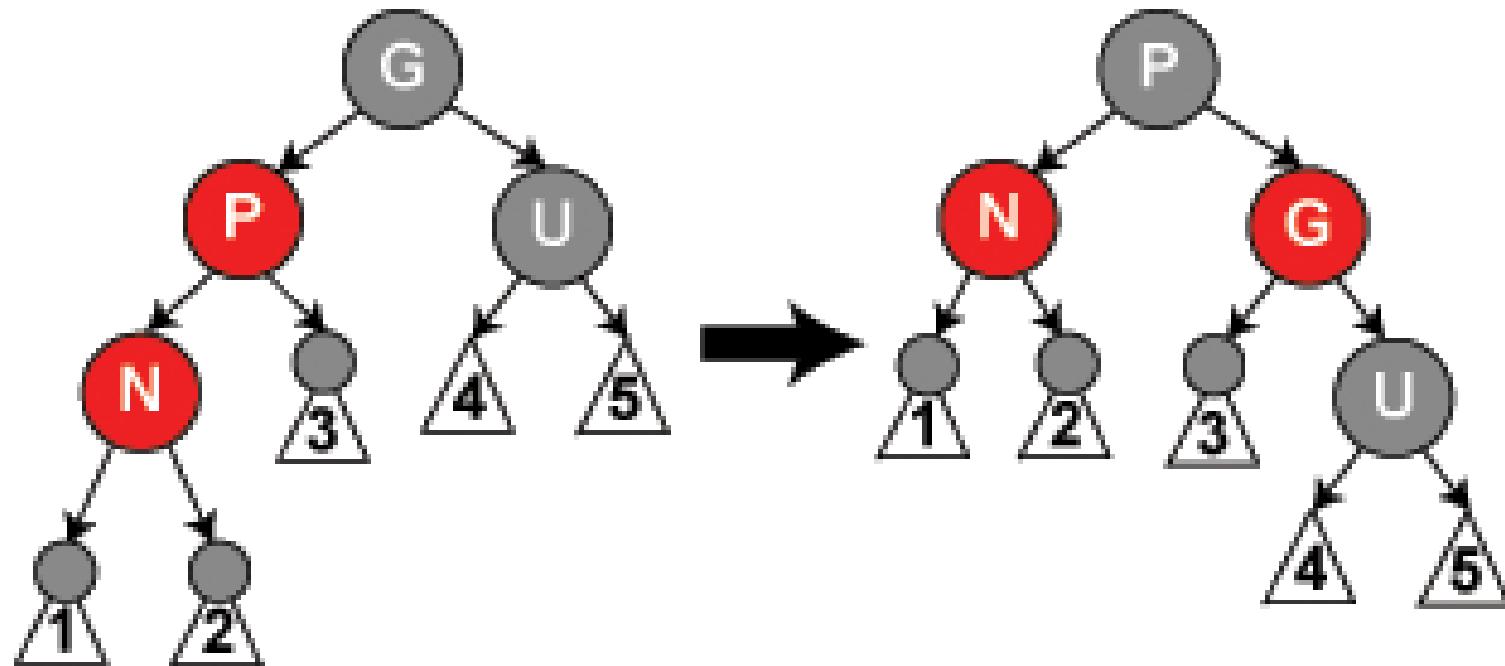
### 3. roditelj crven i ujak crn (“izlomljeni” poredak N, P i G)

- dva simetrična slučaja: N desno dijete od P i P lijevo dijete od G ili N lijevo dijete od P i P desno dijete od G
  - rotacija N oko P, čime se stanje prevodi u “linijski poredak” N, P i G koji se rješava u 4. provjeri
  - nastaviti s 4. provjerom, pridajući P-u ulogu N-a



#### 4. roditelj crven i ujak crn (linijski poredak N, P i G)

- dva simetrična slučaja: N lijevo dijete od P i P lijevo dijete od G ili N desno dijete od P i P desno dijete od G
  - rotacija P oko G
  - zamjena boja P i G (znamo da je G crn jer u protivnom P ne bi mogao biti crven); → gotovo



## RBInsertFixup (N)

//it is assumed that an ordinary inserting routine already inserted node N as a red leaf

while colour (p (N) ) is red //p (N) = parent of N

    if p (N) is left child of g (N)

        U = right child of g (N) ;

        if colour (U) = red

            { colour (p (N) ) = black; //case 1 (2)

                colour (U) = black; //case 1 (2)

                colour (g (N) ) = red; //case 1 (2)

                N = g (N) ;         } //case 1 (2)

    else

        { if N is right child of P //P=p (N) , G=g (N)

            { left-rotate N about P; //case 2 (3)

                N = P;     } //case 2 (3)

            right-rotate p (N) about g (N) ; //case 3 (4)

            colour [ P ] = black; //case 3 (4)

            colour [ G ] = red;     } //case 3 (4)

    else //the same as above, with left and right exchanged

        colour (root) = black; //rješava i novi==korijen

u proceduri

na slajdovima

## Brisanje čvora u RB-stablu:

- jednako kao s B i AVL stablima, brisanje sjedinjenjem ne dolazi u obzir jer bi “razrušilo” cijelo stablo
1. prijepis podataka iz nabližeg prethodnika ili sljedbenika (zamjenski čvor; u nastavku oznaka X)
  2. ukloniti zamjenski čvor; on može imati najviše jedno dijete pa je problem pojednostavljen

Ako je zamjenski čvor crven, više ništa ne treba raditi jer RB svojstva nisu mogla biti narušena. Naime,

- a) crne visine se nisu mogle promijeniti izbacivanjem crvenog čvora
- b) roditelj i dijete zamjenskog čvora mogu biti samo crni (ili sentinel koji je također crn), dakle nismo mogli dovesti dva crvena čvora u odnos roditelj-dijete
- c) zamjenski čvor je crven i kao takav nije mogao biti korijen pa slijedi da je korijen i dalje crn.

Zaključak: složenije situacije mogu nastati samo kada je zamjenski čvor X crne boje.

DeleteRBNODE (node)

*copy content from X (the closest predecessor or successor of node) to node;*

```
child = X's child;      //child must exist, at least as sentinel  
replace X by child;    //incorporate child into the tree at the place  
                        //of X  
if (X->colour == BLACK)  
    RBDeleteFix(child); //RB fixup  
free(X);
```

Dalje na slajdovima vrijede oznake:

X = čvor koji se uklanja (zamjenski čvor), N = dijete od X (već na mjestu X),  
P = roditelj (od X, a potom N), S = sibling od N (drugo dijete roditelja P),  
SL = lijevo dijete siblinga S, SR = desno dijete siblinga S.

Nakon uklanjanja crnog čvora mogu nastati tri problema:

1. ako je uklonjen korijen, mogao je imati samo jedno dijete (N) koje postaje novi korijen, a ono može biti i crveno; povreda 2. pravila (korijen je crn)
2. nakon uklanjanja X, njegovo dijete N i roditelj P su u odnosu dijete-roditelj i ako su oboje crveni, prekršeno je 4. pravilo (djeca crvenog su crna)
3. uklanjanje crnog X znači smanjenje crne visine svih njegovih prethodnika (predaka); povreda 5. pravila

Prvi slučaj je najjednostavniji; dovoljno je prebojati N u crno i sve je riješeno jer se mijenjanjem boje korijena jednako mijenja crna visina svim čvorovima stabla.

Druge dvije povrede RB pravila su međusobno povezane, a rješenje u oba slučaja ovisi o boji čvora N.

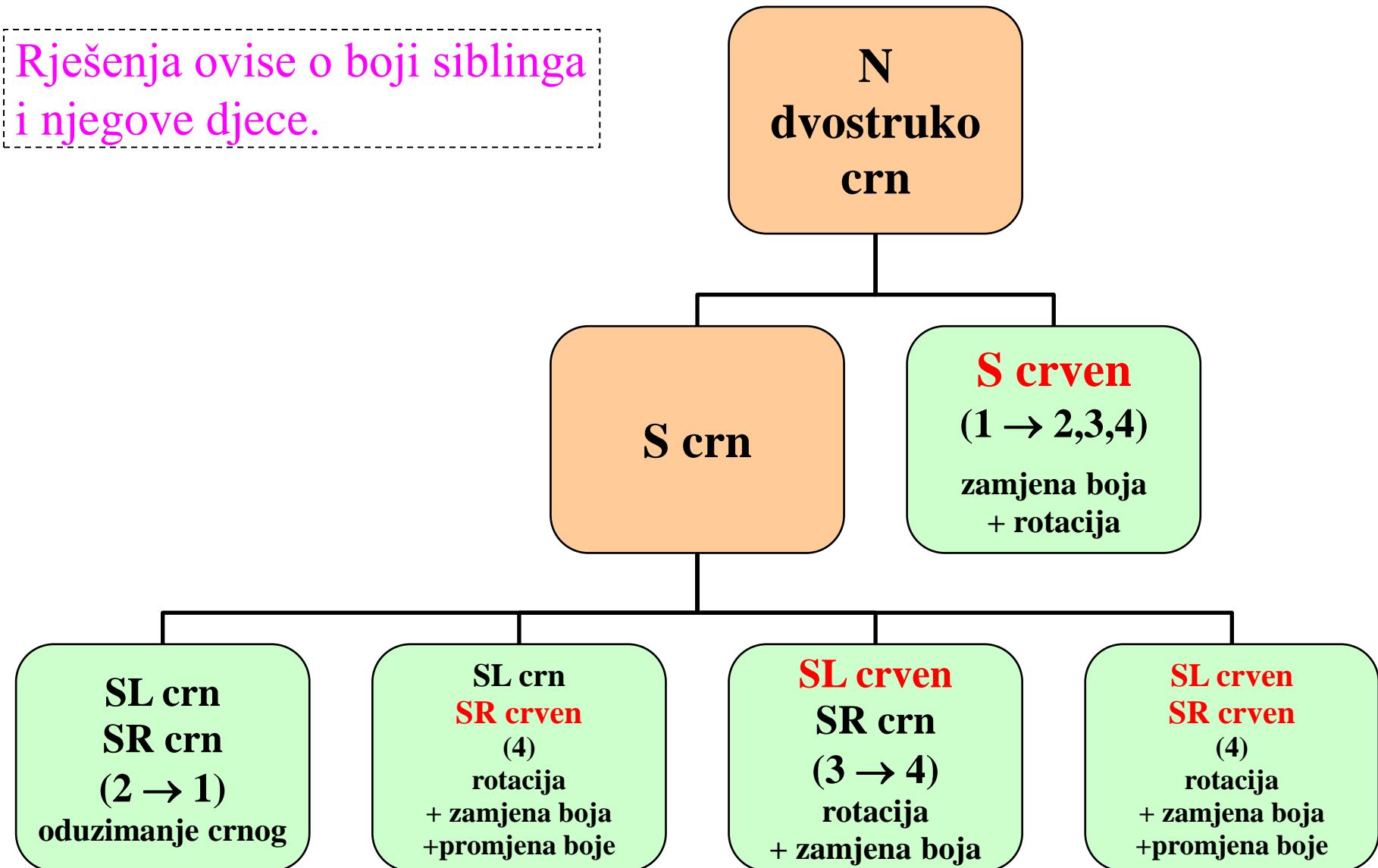
Zamislimo da možemo nekako prenijeti crninu X-a na N. Tada ju uklanjanjem X-a ne bismo izgubili i RB pravila ne bi bila prekršena. Recimo da je to moguće i da tako postupimo. Ako je N prethodno bio crven, postat će crveno-crn i crnoj visini doprinositi =1. Ako je N prethodno bio crn, postat će dvostruko crn i crnoj visini doprinositi =2. Dakle, N može biti ili crveno-crn ili dvostruko crn.

Ako je crveno-crn, dovoljno je prebojati ga u čisto crno i gotovo; stablo će ponovno biti pravilno RB stablo.

Ostaje nam riješiti (ukloniti) dvostruku crninu (“dvostruko crno”). Ideja je prosljediti višak crnog prethodniku i tako taj višak podizati sve dok ne dođe na mjesto gdje ga možemo trajno ugraditi u stablo ili dok ne dođe u korijen, gdje se jednostavno može odbaciti, tj. zanemariti. Trajnu ugradnju u stablo postižemo prikladnim rotacijama i bojanjem čvorova.

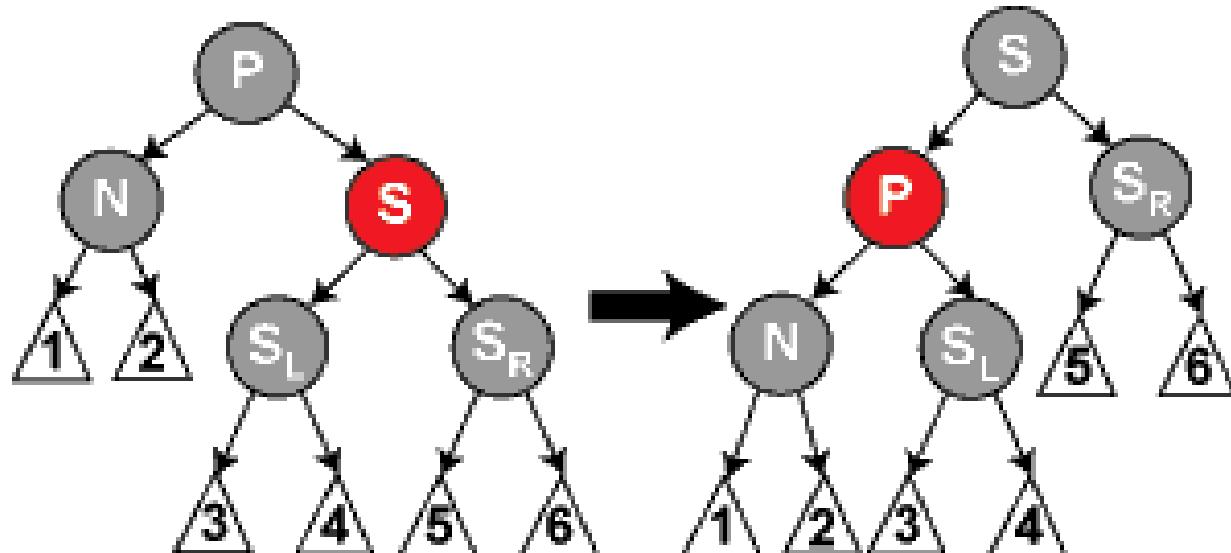
Četiri su (+ četiri simetrična) moguća slučaja, a ovise o boji siblinga S (sigurno postoji!) i njegove djece.

Rješenja ovise o boji siblinga  
i njegove djece.



# 1. sibling S je crven

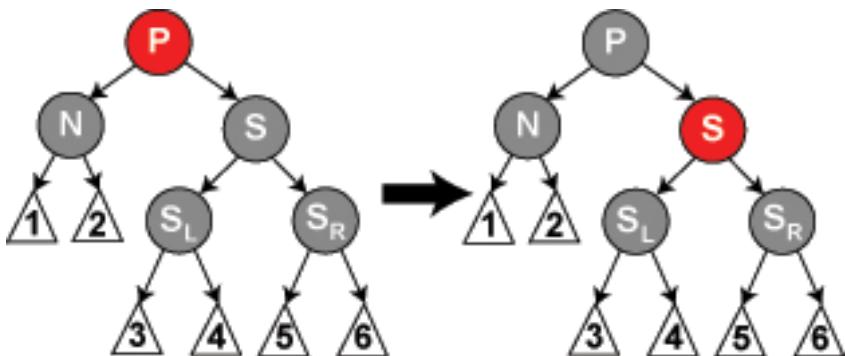
- P je sigurno crn jer ima crveno dijete
- nakon brisanja X crna visina lijevog podstabla od P za jedan je manja od crne visine desnog podstabla (tj. N dvostruko crn)
- zamijeniti boje P i S pa rotirati S oko P (simetrija!)
  - gledano odozgo, iz ostatka stabla, putevi od S na niže (u podstabla 1,2,3,4,5 i 6) imaju imaju isti broj crnih čvorova kao i prije
  - N sada sigurno ima crnog siblinga SL (jer je SL bio dijete crvenog čvora) i crvenog roditelja P → slučajevi 2 ili 3 i 4 (moramo dalje jer crne visine lijevo i desno od P još uvijek nisu iste)



## 2. S crn, djeca od S crna

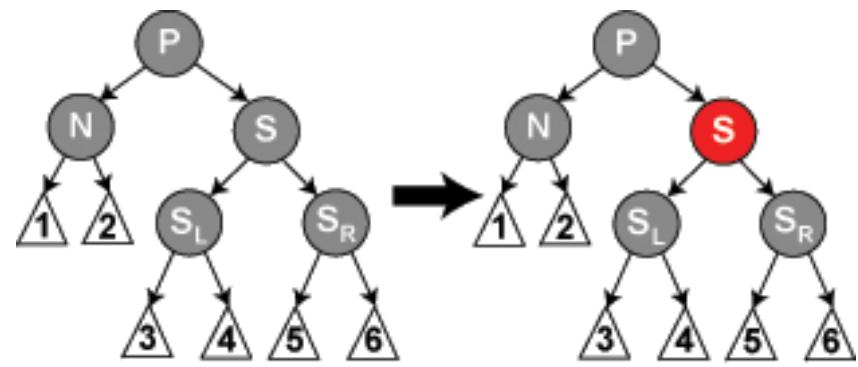
- P nevažan za intervenciju; o njemu ovisi samo daljnje postupanje nakon intervencije → slučajevi 2a i 2b
- oduzeti jedno crno N-u i S-u; N ostaje jednostruko crn, a S postaje crven (drugim riječima, obojati S u crveno)
- taj višak crnoga proslijediti višoj razini, tj. P-u koji time postaje ili crveno-crn ili dvostruko crn
- višak crnog je sada razinu više nego prije (konvergencija!)

P crveno-crn



2a

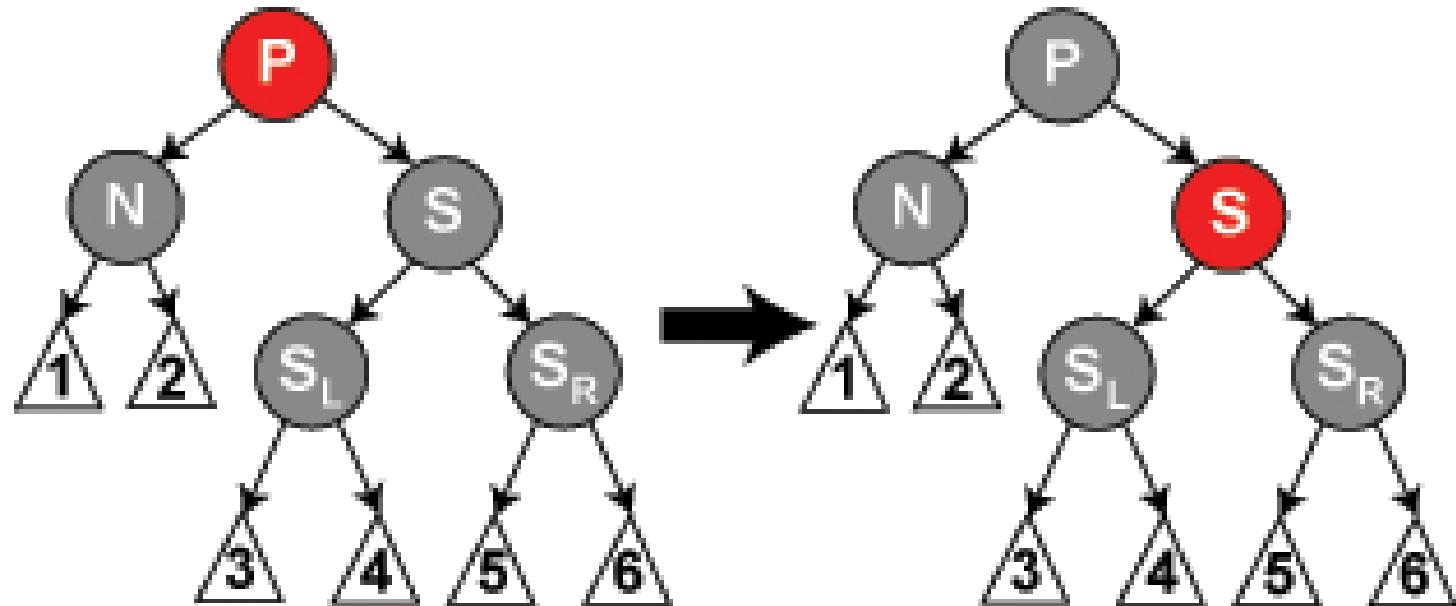
P dvostruko crn



2b

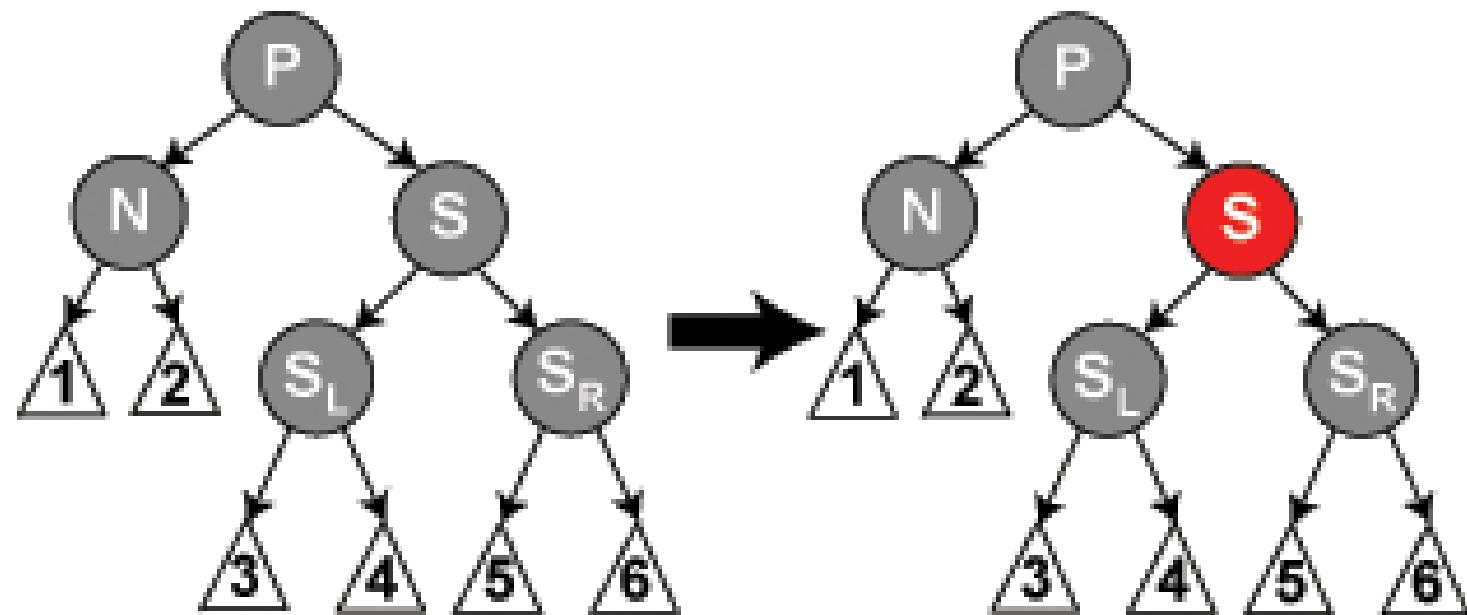
## 2. a) P crven

- oduzeti jedno crno N-u i S-u; N ostaje jednostruko crn, a S postaje crven (drugim riječima, obojati S u crveno)
- taj višak crnog proslijediti P-u koji time postaje crveno-crn
- crno se u P može ugraditi trajno; lijevo podstablo dobiva izgubljeno crno, a desnom se ništa ne mijenja (jer je S crven)
- prebojati P u čisto crno → gotovo



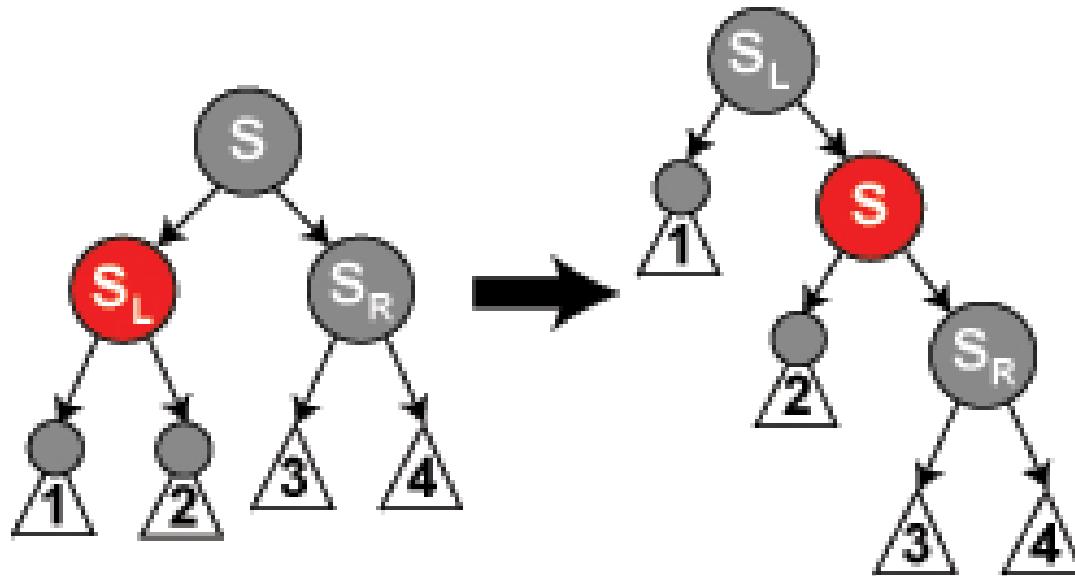
## 2. b) P crn

- oduzeti jedno crno N-u i S-u; N ostaje jednostruko crn, a S postaje crven (drugim riječima, obojati S u crveno)
- taj višak crnog proslijediti P-u koji time postaje dvostruko crn
- ako je P korijen, višak crnog se odbacuje → gotovo;  
u protivnom, nazad na slučaj 1 promatrajući P kao N
- problem je razinu više  $\Rightarrow$  konvergencija!



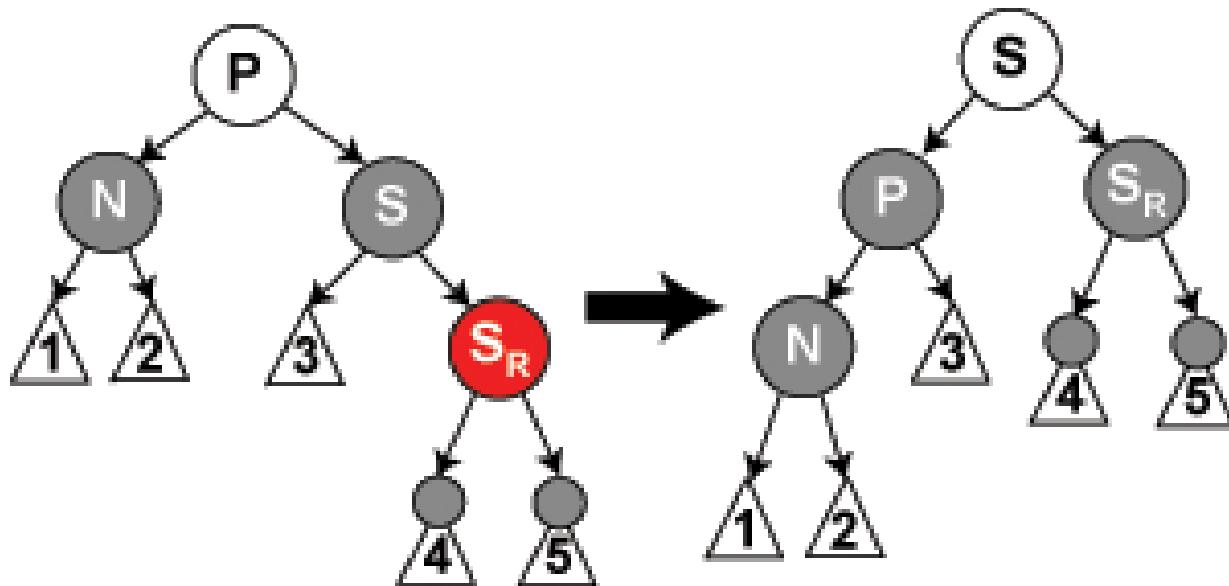
### 3. S crn, SL crven, SR crn, a P nevažan

- S je N-ov sibling, a N je lijevo dijete od P (simetrija!)
- rotirati SL i S, tako da SL postane N-ov sibling, i potom im zamijeniti boje (S-u i SL-u)
- svi putevi i dalje imaju iste crne visine, ali sada N ima crnog siblinga kojemu je desno dijete crveno → slučaj 4
  - u primjeru za slučaj 4, SL je preimenovan u S kao N-ov sibling



#### 4. S crn, SR crven, a P i SL nevažani

- rotirati S oko P (simetrija!)
- potom zamijeniti boje S i P, a SR prebojati u crno → gotovo
  - korijen podstabla ostaje iste boje (dobro za roditelja korijena)
  - gledano od P (kasnije S), putevi u podstabla 3, 4 i 5 zadržavaju isti broj crnih čvorova, a oni preko N dobivaju jedan novi i time nadomještaju raniji gubitak u N (P je ili bio crven pa postao crn zamjenom boje sa S ili je bio crn i prije pa je i S ostao crn i dodao jedno crno u put u podstabla 1 i 2)



## RBDeleteFixup (N)

//it is assumed that an ordinary deleting routine has already replaced black X by N

while N is not root and colour (N) is black

  if N is left child of P

    S = right child of P; // sibling

    if colour (S) = red

      colour (S) = black; // case 1

      colour (p (N)) = red; // case 1

      left-rotate S about P; // case 1

    if colour (SL) = black and colour (SR) = black

      colour (S) = red; // case 2

      N = p (N); // case 2

    else

      if colour (SR) = black

        { colour (SL) = black; // case 3

          colour (S) = red; // case 3

          right-rotate SL about S; } // case 3

          colour (S) = colour (P); // case 4

          colour (P) = black; // case 4

          colour (SR) = black; // case 4

          left-rotate S about P; // case 4

          N = root; // or simply break;

      else // same as if-clause with left and right exchanged

        colour (N) = black;

# Samopodešavajuća stabla (Self-Adjusting Trees)

Primarna namjena binarnih stabala za pretraživanje (*search* stabla) je brz pristup podatcima. DSW algoritam, AVL i crveno-crna stabla to omogućavaju, ali cijena je relativno komplikirano održavanje (sporije dodavanje i brisanje).

Samopodešavajuća stabla su moguća zamjena strogim metodama, a slijede ideju samopodešavajućih lista – podatke kojima se češće pristupa podići na više razine. Zato svaki čvor mora “znati” koliko mu se puta pristupilo.

Dvije osnovne strategije:

1. *Single rotation* – čvor kojem se pristupilo rotirati oko roditelja
2. *Moving to the Root* - čvor kojem se pristupilo postaje korijen stabla.
  - varijanta ove strategije je i tzv. *Splaying*;  
dobra kad se nekim elementima pristupa puno  
češće nego ostalima (vidi Drozdek...)

# Struktura Trie (čita se kao *try*)

Fredkin, Edward (Sept. 1960), "Trie Memory", *Communications of the ACM* 3(9), pp. 490-499

- naziv dolazi od *retrieval* (ponovni dohvati, povrat)
- Trie - stablo za čije se pretraživanje koriste samo dijelovi ključeva pohranjenih podataka
- slična prefiksnom (*prefixed*)  $B^+$  stablu, ali nema komplikacija s određivanjem optimalnih prefiksa
  - ponovimo:  $B^+$  stabla se uvode radi bržeg slijednog pristupa podatcima; unutarnji čvorovi ne sadrže informacije, već ključeve i služe samo za brzo pretraživanje (*index set*); listovi su međusobno povezani u niz (listu; *squence*) i čine *sequence set*; dakle,  $B^+$  stablo je indeks u obliku  $B$  stabla plus lista svih podataka u stablu pa mu odatle i naziv; prefiksno  $B^+$  stablo razvija tu ideju dalje pa indeksi nisu cijeli ključevi, nego najmanji dijelovi ključa (prefiksi) dovoljni za razlikovanje dva susjedna podatka
- dvije vrste čvorova
  - unutarnji = samo kôdovi (svi dijelovi svih ključeva) i pokazivači
  - listovi = podatci

Tipična primjena je npr. *spell checker* koji mora brzo pronaći relativno kratak *string* u skupu od više desetaka tisuća riječi. Trie je bolja od stabla jer jedan čvor ima više od dva podstabla pa je niža i pristup podatcima je brži (visina joj je jednaka najdužoj riječi; ovisi o svojstvima podataka, ali npr. u HR i EN riječi su relativno kratke).

## Prednosti u usporedbi sa stablima

- konačni oblik Trie ne ovisi o redoslijedu upisa podataka
- svaka pojedinačna usporedba je brža jer se ne provjerava cijeli ključ, nego samo jedan njegov element
- budući da su svi podatci u listovima i nisu međusobno povezani, brisanje je vrlo jednostavno

## Nedostatak u usporedbi sa stablima

- zbog dvije vrste čvorova, dodavanje elemenata je komplikiranije nego u stablu (ne jako komplikirano - vidi Drozdek...)

Pretraživanje: iz čvora na  $k$ -toj razini ide se u čvor koji pokazuje pokazivač uz kôd jednak  $k$ -tom dijelu ključa koji se traži.

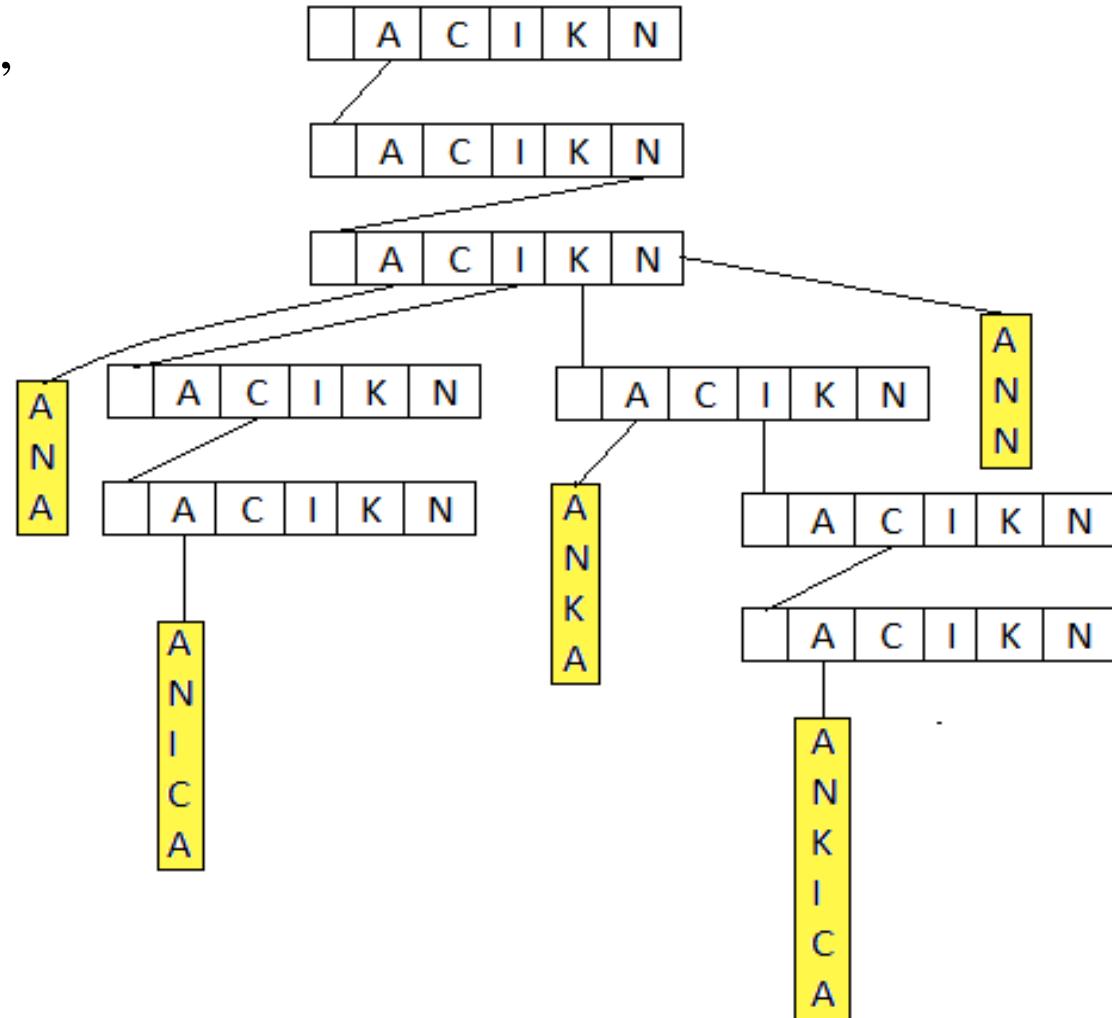
- ako je pokazivač NULL, traženog podatka nema (mjesto za ubacivanje)

Primjer: ANA, ANN, ANICA,  
ANKA, ANKICA

Potrebna slova (kodovi):

A, C, I, K, N

Razine	1	2	3	4	5	6
	A	N	A			
	A	N	N			
	A	N	I	C	A	
	A	N	K	A		
	A	N	K	I	C	A



Što ako želimo  
upisati ANNA?

Rješenje: svakom čvoru dodati poseban kôd (u primjeru #) koji sigurno nije dio nijednog ključa i koji će značiti kraj riječi. Kada se prilikom pretraživanja dođe do posljednjeg elementa ključa (posljednjeg slova), pokazivač nas mora usmjeriti u # dio čvora na nižoj razini, gdje je smješten prefiks, tj. riječ koja završava posljednjim slovom ključa.

Riječi:

NIK

NIKI

NIKA



Da listovi budu listovi, dok nema prefiksa pokazivač mora biti usmjeren izravno u podatak, a ne niži indeks-čvor. Npr., ako se prvo upisuje NIK, a onda NIKI, riječ NIK treba biti normalno smještena, tj. treći pokazivač na trećoj razini treba pokazivati podatak NIK (podatkovna vrsta čvora). Kada dođe NIKI, treći pokazivač na trećoj razini treba preusmjeriti na novi indeks-čvor. U novom čvoru, "I" treba usmjeriti na NIKI, a # na NIK.