Blok 03
Struktura podataka i algoritmi

prof. Dr Vidan Marković

Univerzitet Singidunum
2023/2024

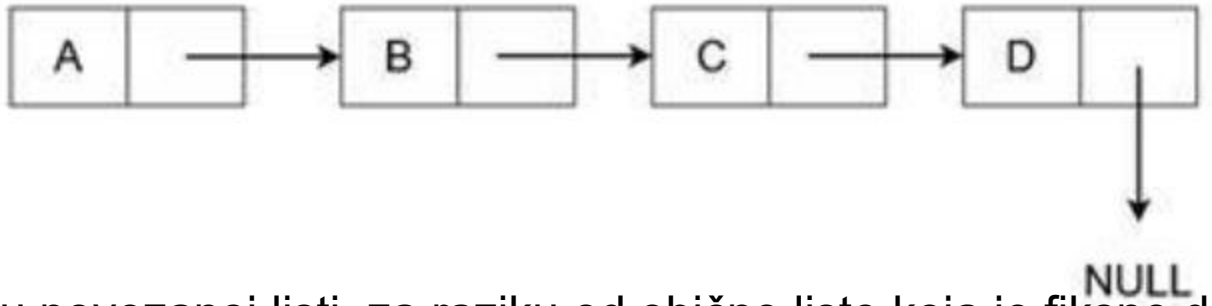
Sadržaj

1. Uvod u strukture podataka

2. Pregled struktura podataka

3. Diskusija

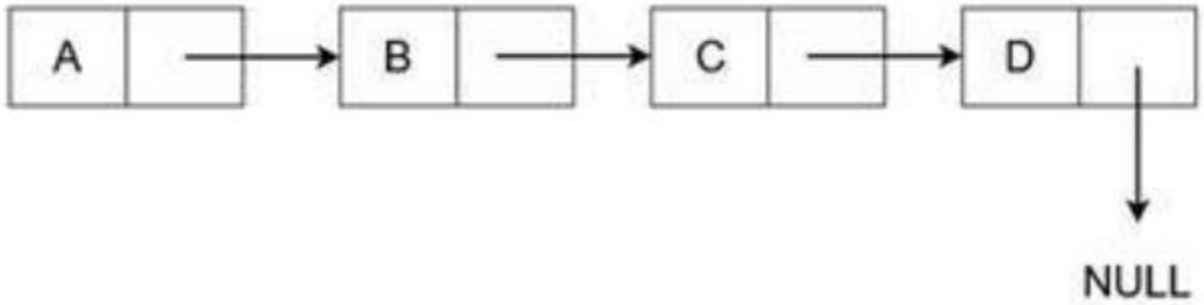
Data strukture pregled: povezane liste



- Pristup elementima u povezanoj listi, za raziku od obične liste koja je fiksne dužine i gde se elementima može pristupati u konstantnom vremenu (poznate memorijske lokacije koje se mogu izračunati), ovde moramo ići u linearnom vremenu jer moramo proći kroz sve čvorove da bi pronašli poziciju traženog elementa.

- Dakle imamo linernu kompleksnost izvršavanja: **O(n).**

Data strukture pregled: povezane liste



- Pristup elementima u povezanoj listi, za raziku od obične liste koja je fiksne dužine i gde se elementima može pristupati u konstantnom vremenu (poznate adrese memorijskih lokacija koje se mogu izračunati), ovde moramo ići u linearnom vremenu jer moramo proći kroz sve čvorove da bi pronašli poziciju traženog elementa.

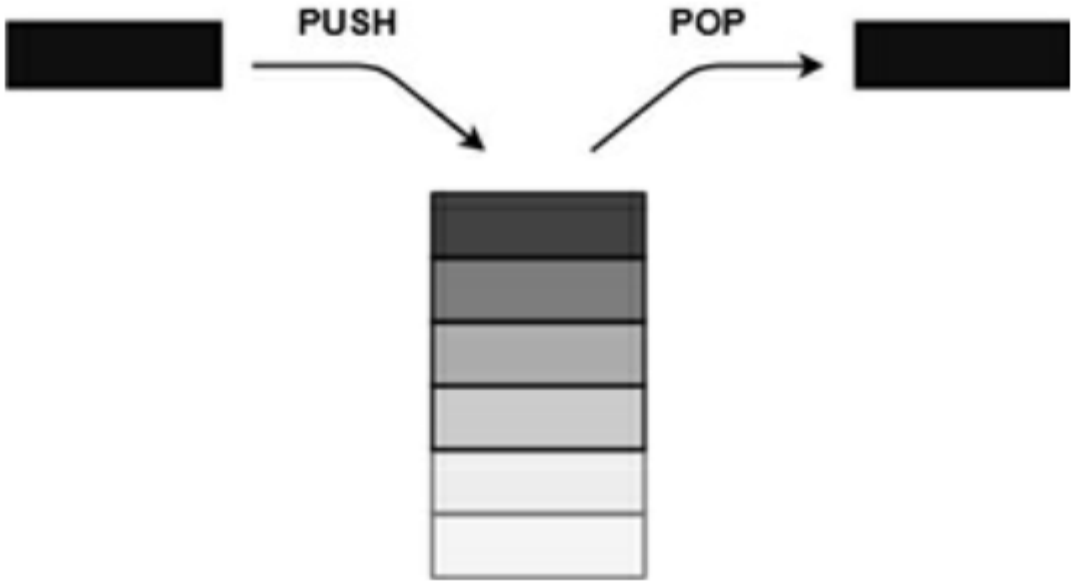
- Dakle za pronalaženje adrese elementa imamo linernu kompleksnost izvršavanja: **O(n).**

- Isti slučaj je i za pronalaženje vrednosti na traženoj adresi.

- Ako želimo da ubacimo novi element na početak liste onda konstantno vreme O(1), a kad idemo sa iteratorom do mesta na listi gde treba da ubacimo novi element onda je ubacivane u lineranom O(n).

- Sa brisanjem je ista logika za kompleksnost izvršavanja kao i za ubacivanje elemenata.

Data strukture pregled: Stack

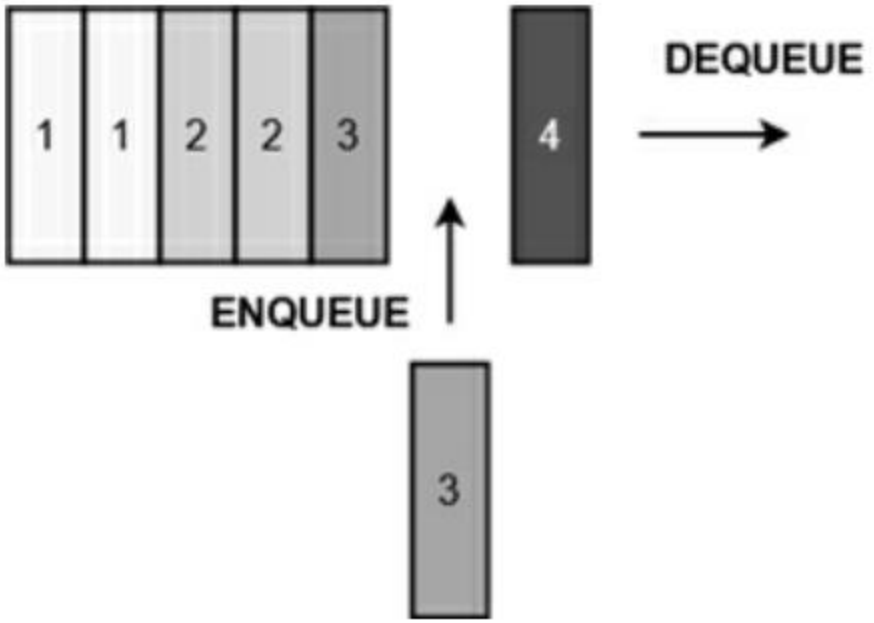


- LIFO princip rada.
- Imamo i *peek* instrukciju koja omogućava da se pristup elementu na vrhu bez njegovog povlačenja sa stacka.
- Kompleksnost izvršavanja ubacivanja i brisanja je u konst. vremenu **O(1).** Traženje po stacku se obično ne radi jer je on prevashodno namenjen za rekurzije i pozive funkcija (odatle i čuveni StackOverflow exception), ako baš treba da se pretražuje onda je isti slučaj kao sa povezanim listama i radi se u lineranom vremenu **O(n)**.

Data strukture pregled: Queue (red)

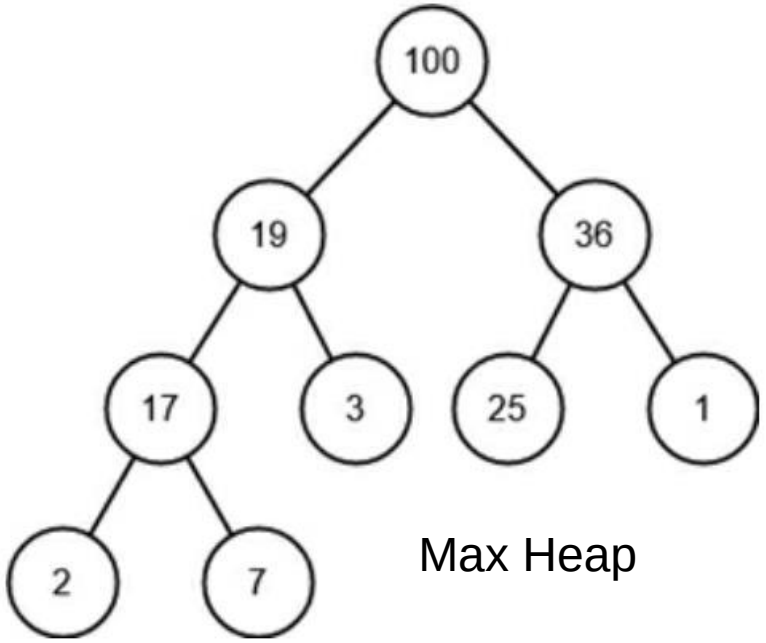
- FIFO princip rada.
- Kompleksnost izvršavanja ubacivanja i brisanja je u konst. vremenu **O(1).** Traženje po Redu se može završiti u lineranom vremenu **O(n)**. Slično kao za Stack.

Data strukture pregled: Priority Queue (red)



- Ni LIFO ni FIFO princip rada.
- Običan Red u kome svaki element ima svoj prioritet (pomenuto u Dijkstra algoritmima) tako da se ubacivanje i brisanje vrši po tim principima.
- Kompleksnost izvršavanja ubacivanja i brisanja je u lineranom vremenu **O(n)**. Međutim, postoji opcija da se engueue i dequeue uradi u logaritamskom vremenu na bazi korišćenja **Heap-a**.

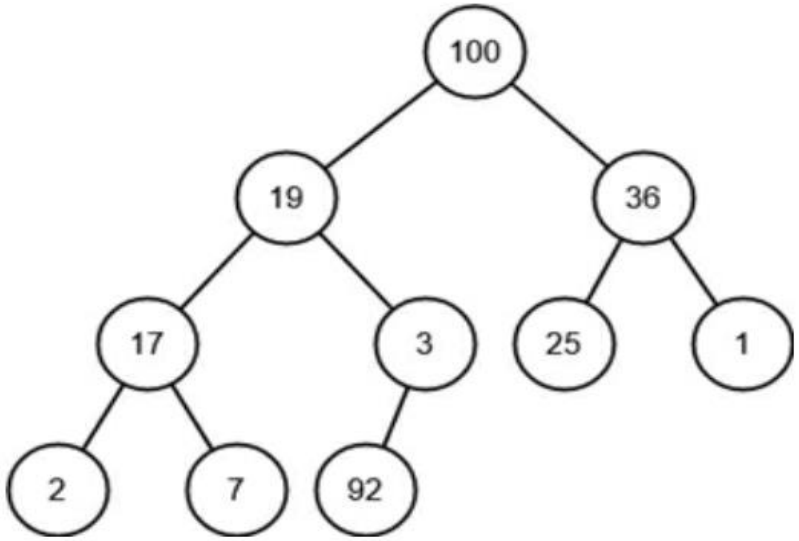
Data strukture pregled: Heap



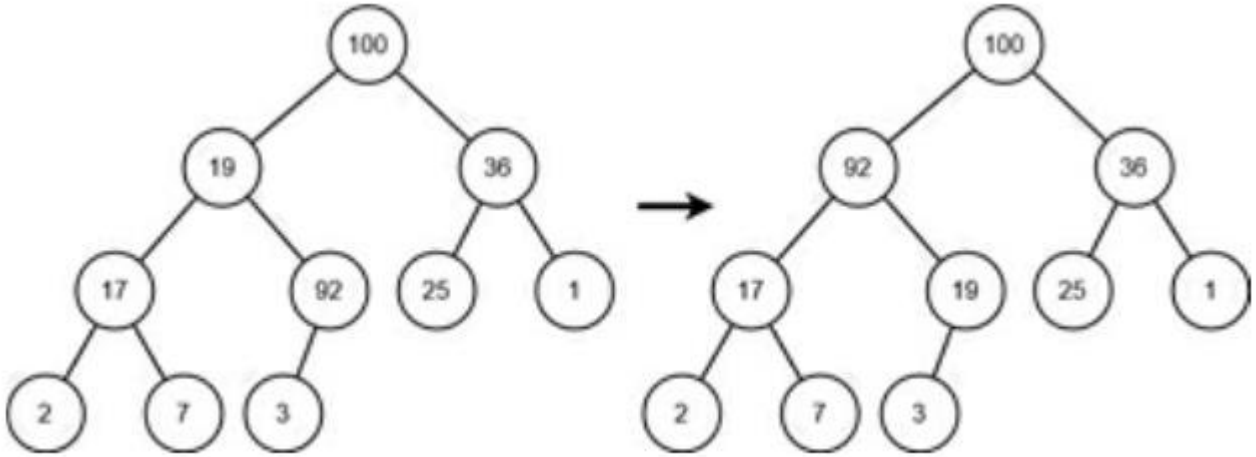
Max Heap

- Heap je u suštini samo drvo, dakle graf bez krugova i petlji. Samo što ovo drvo ima Heap osobinu.
- Svaki čvor u (binarnom) heapu ima levo i desno dete (koji mogu biti Null što čvor pretvara u list).
- Heap osobina zahteva da je svako dete ima manju vrednost od roditelja – **Max Heap**, ili ima veću vrednost od roditelja **Min Heap**.

Data strukture pregled: Heap



1. Dodajemo novi elemenat



2. Heapify-up

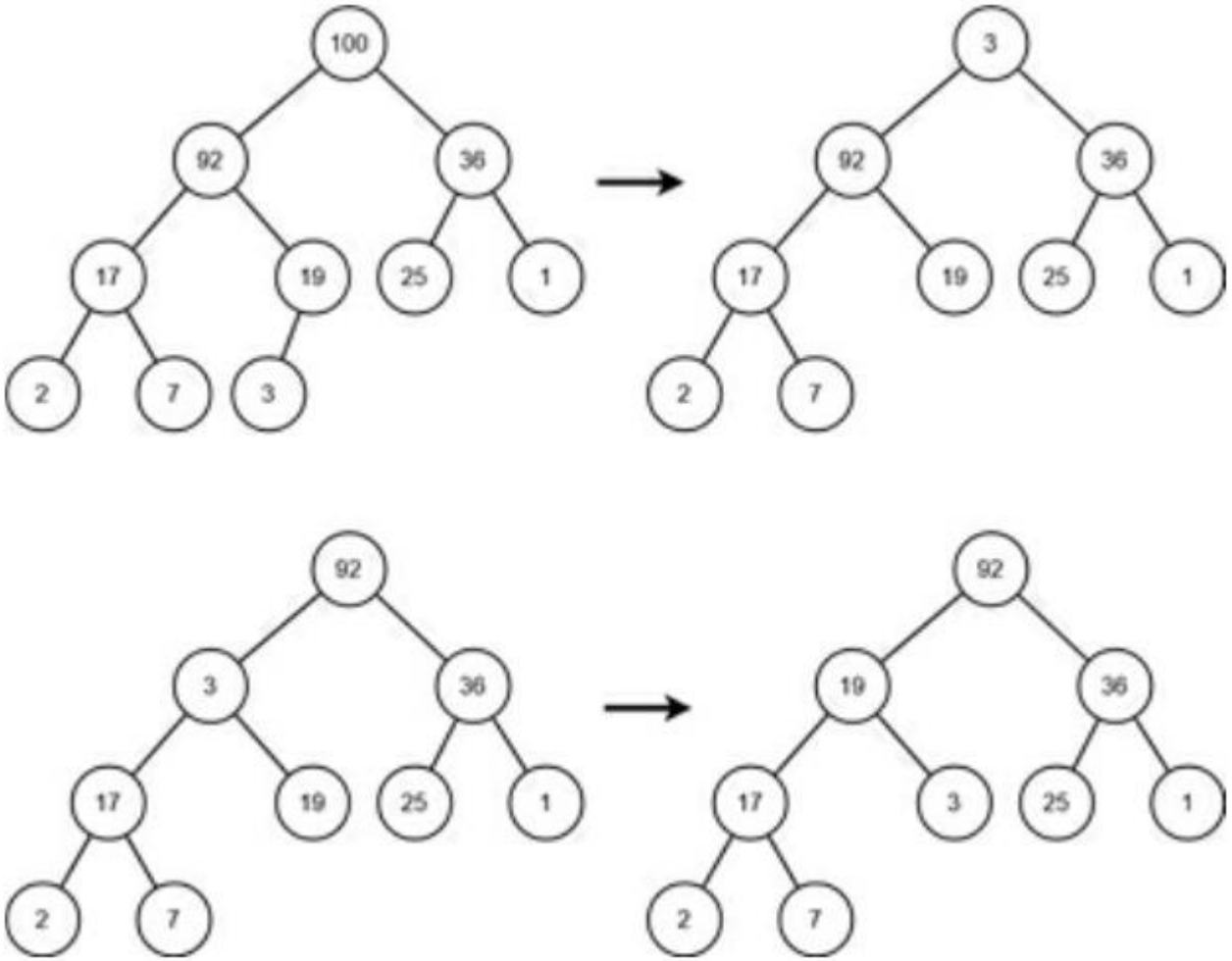
- **Ubacivanje** novog elementa u Heap zahteva tzv **Heapify-up** proceduru. Uradi se samo append ali to očigledno narušava Heap osobinu!
- Poredimo vrednost novog elementa (92) sa roditeljem i ako je veći uradimo swap pozicija.
- Sve ovo radimo dok ne dođemo do roditelja sa većom vrednošću.
- Kompleksnost izvršavanja je u logaritamskom vremenu **O(log n). Isto je i za Min Heap samo obrnuto.**

Data strukture pregled: Heap

- **Brisanje (dequeue)** elementa zahteva **Heapify-down** proceduru.

- Mi bi smo prvo obrisali root čvor, zato što ima najveći priority, pa onda uradili h-d proceduru. Nakon brisanja roota, zamenili bi ga sa poslednjim čvorom Heapa. Onda bi pogledali njegovu decu i uporedili sa najvećim od njih (ili najmanjim kod Min Heapa). Ako je naš element manji od najvećeg deteta onda uradimo swap.Ovo ponavljamo sve dok heap osobina nije zadovoljena.

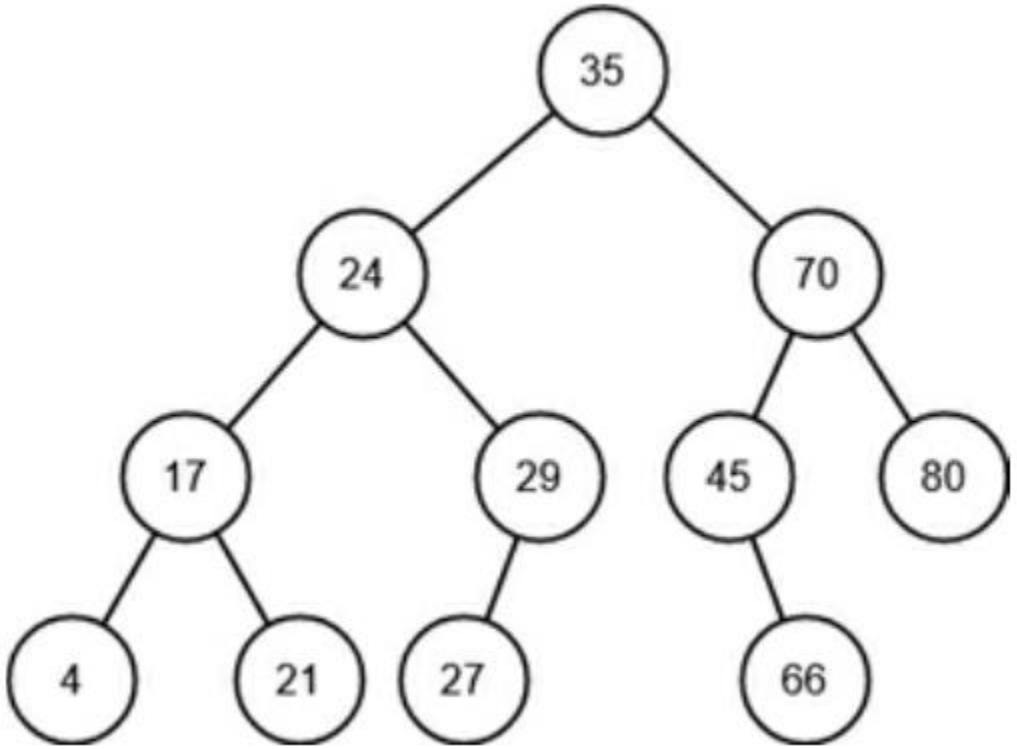
- Kompleksnost izvršavanja je u log vremenu **O(log n).**



Heapify-down

Data strukture pregled: Binarno stablo za pretraživanje

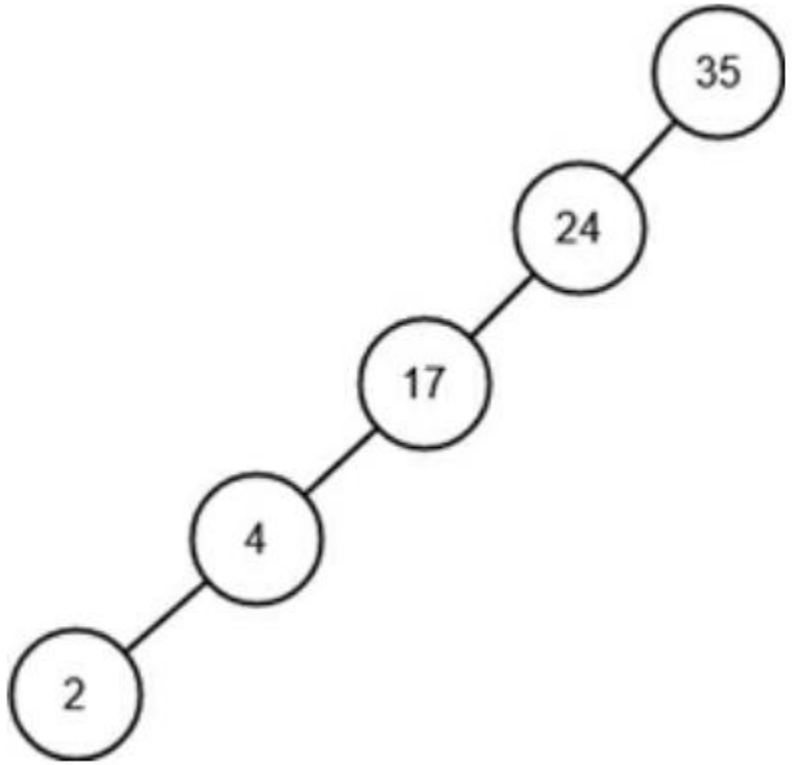
- Samo ime govori da je tipa stabla (drveta).
- Slično Heapu, ali podrazumeva da su deca u levom podstablu manja od roditelja, a u desnom podstablu veća od roditelja.
- Kompleksnost izvršavanja je u log vremenu **O(log n) u slučaju kada nam je BSC balansirano!**



BSC

Data strukture pregled: Binarno stablo za pretraživanje

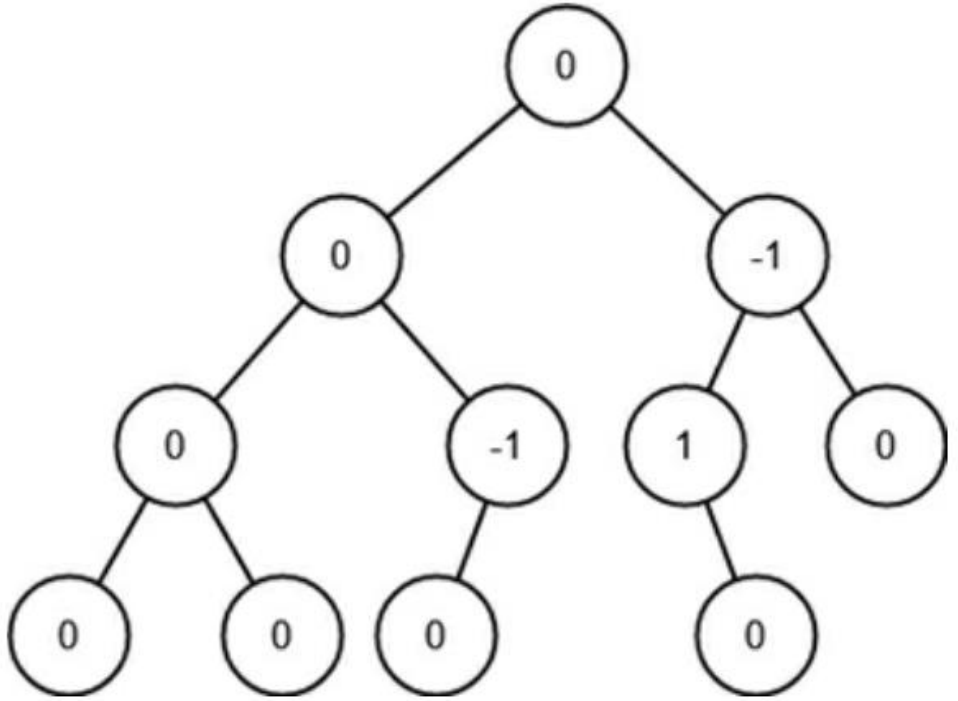
- Na slici je najgori slučaj nebalansiranog BSC koje je i dalje regularno.
- Dobro je što je verovatnoća pojave ovakve situacije mala.
- Kompleksnost izvršavanja pretaživanja je u lin vremenu **O(n)**
- Takodje za najgori scenariju ubacivanje i brisanje je **O(n).**
- **Zato se radi balansiranje BST-a, tj. rešenje su tzv. samo-balansirajuća stabla.**



Potpuno nebalansirano BSC

Data strukture pregled: Samobalansirajuća stabla - AVL

- Adelson-Veski i Landis stablo se bazira na definisanju granica razlike u visini stabla izmedju podstabla svakog noda.
- Kada primetimo da postoji razlika veća od maks dozvoljene onda odmah rebalansiramo stablo.
- Da bi se ovo omogućilo definiše se tzv. **balans faktor** za svaki čvor.
- On se izračunava tako što se od visine desnog podstabla oduzme visina levog podstabla. Ako su oba iste visine onda se dobija 0. Ako je desno veće od levog dobija se pozitivna vrednost i obrnuto.
- Napomena: ne mešati BF sa vrednostima nodova!

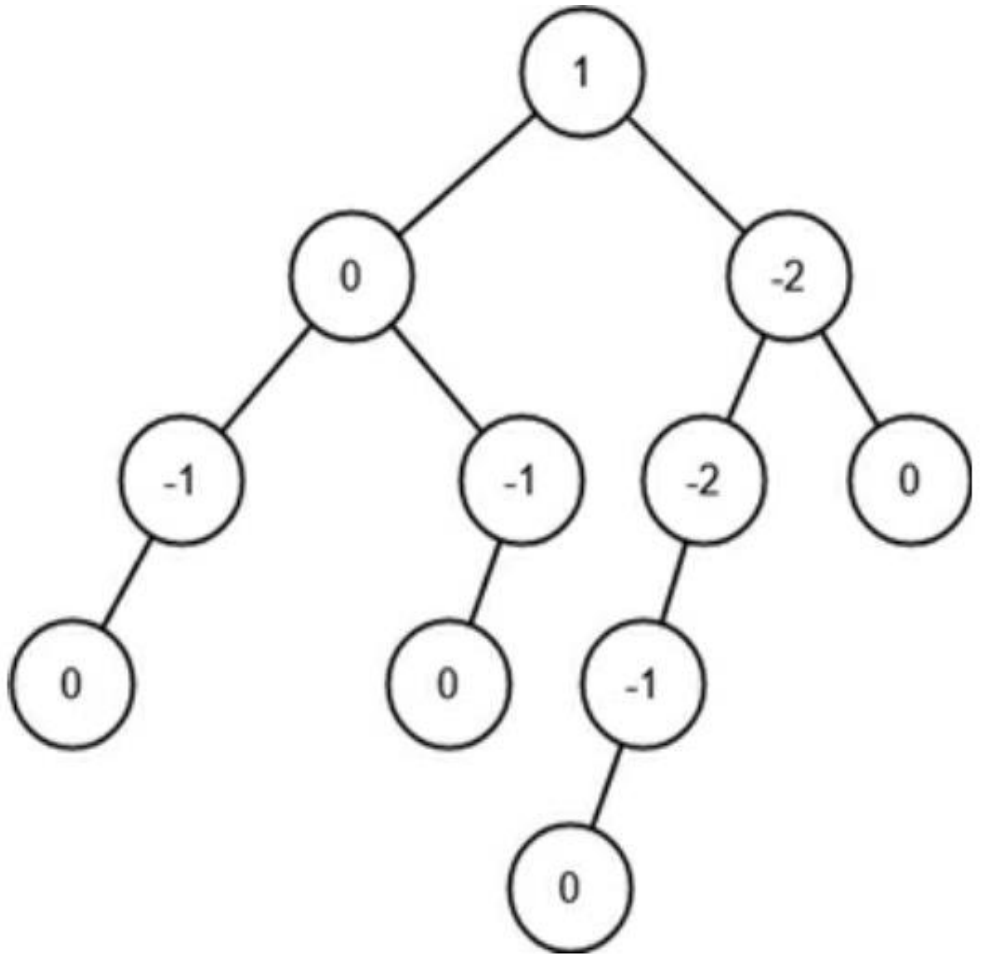


$$balance\ factor = height(right) - height(left)$$

$$balance\ factor \in \{-1,0,1\} \rightarrow balanced$$

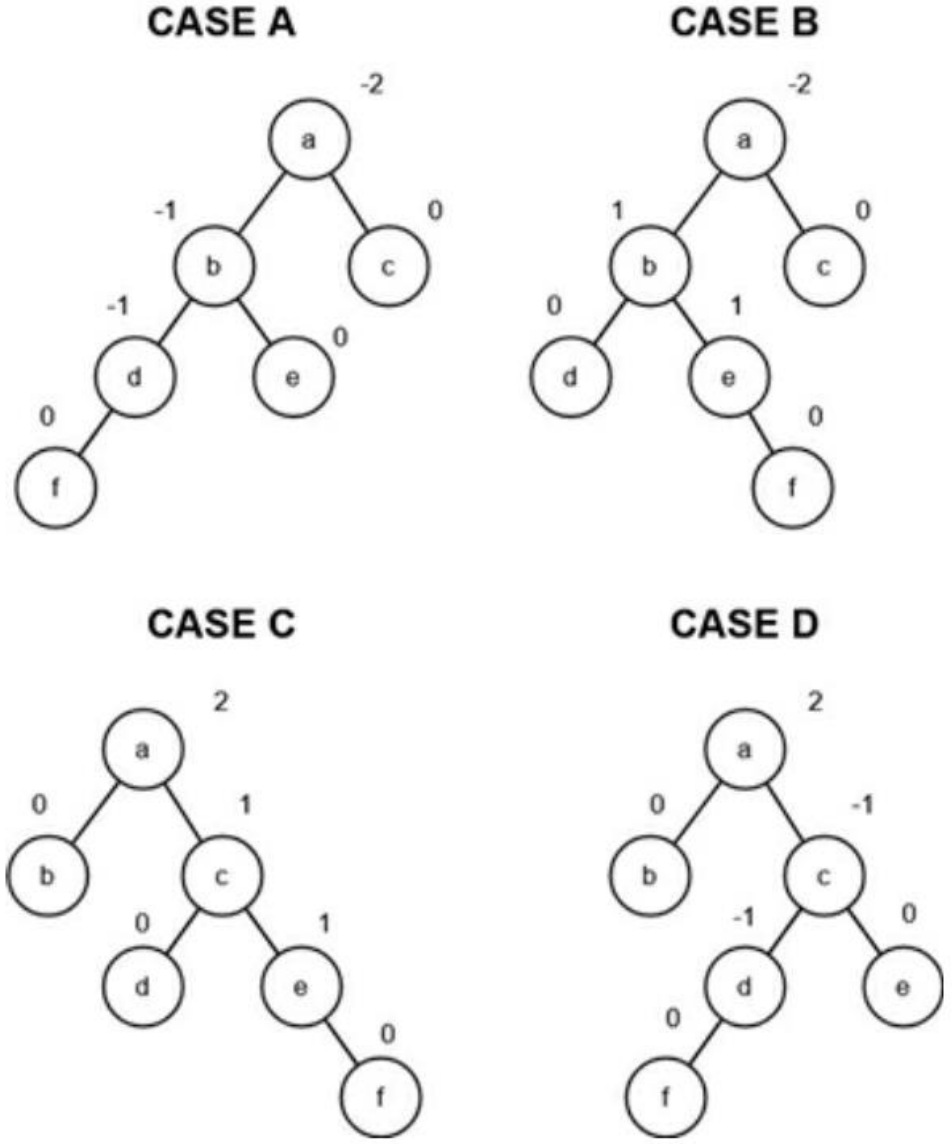
Data strukture pregled: Samobalansirajuća stabla - AVL

- Primer: ne balansirano drvo
- Zašto je nebalansirano? Objasniti.
- Kada balansirano drvo pravimo ne balansiranim?



Data strukture pregled: Samobalansirajuća stabla - AVL

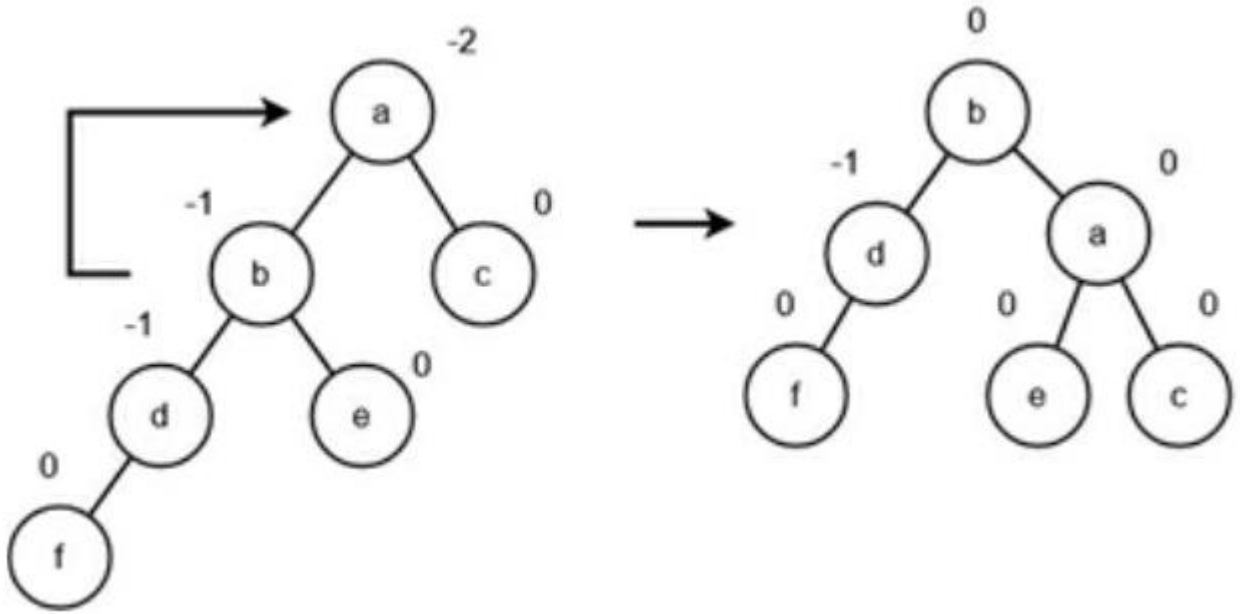
- Četiri slučaja nebalansiranosti.
- Hajde da ih analiziramo!



Data strukture pregled: Samobalansirajuća stabla - AVL

Rešenje za slučaj A:

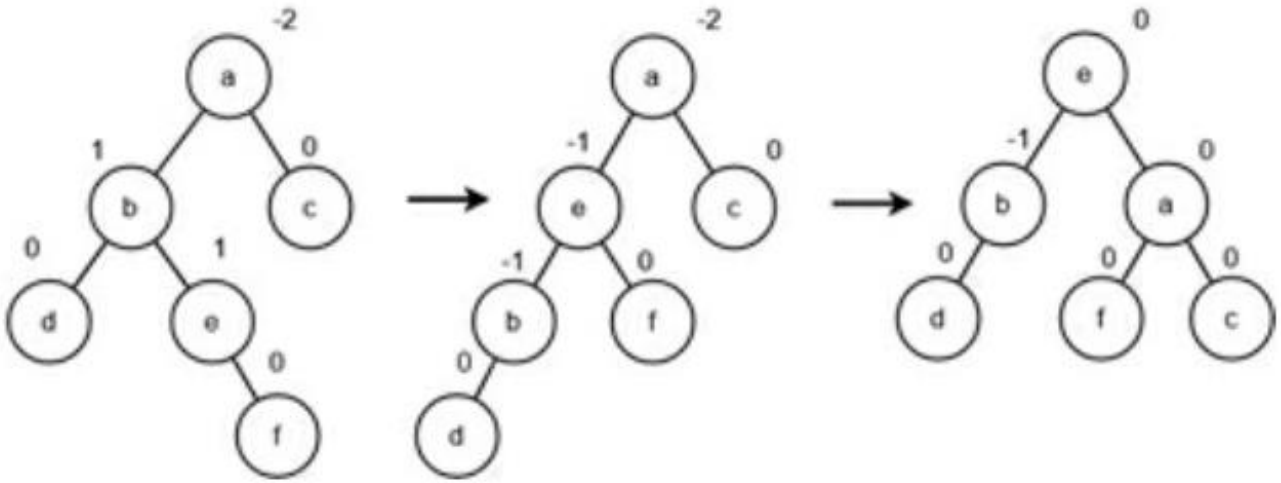
- Rotacija b i a pozicija.
- Desno dete b postaje levo dete a.



Data strukture pregled: Samobalansirajuća stabla - AVL

Rešenje za slučaj B:

- Dupla leva-desna Rotacija e i b pa e i a pozicija.



Rešenja za slučajeve C i D su ista kao i za A i B samo obrnuta rotacija.

Kompleksnost izvršavanja je log vremenu **O(log n).**

Samobalansirajuća stabla – B stablo

- Sva stabla koja smo posmatrali su poprilično dobra kada su podaci u RAMu.
- Međutim, kada govorimo o velikoj količini podataka, onda je očekivano da se nalaze i na hard diskovima i na eksternim medijima.
- Kada nam trebaju podaci sa diska onda tražimo preko određene adrese podake sa diska (znatno sporiji od RAMa) i prebacujemo deo podataka u RAM.
- Spoljni mediji za skladištenje obično imaju **blok-orijentisanu strukturu** i svaki put kada pristupimo nekim podacima **pristupamo punom blokčejnu**.
- Stoga ima smisla kombinovati više čvorova kako bismo *spljoštili* naše drvo i **smanjili** količinu pristupa.

Samobalansirajuća stabla – B stablo

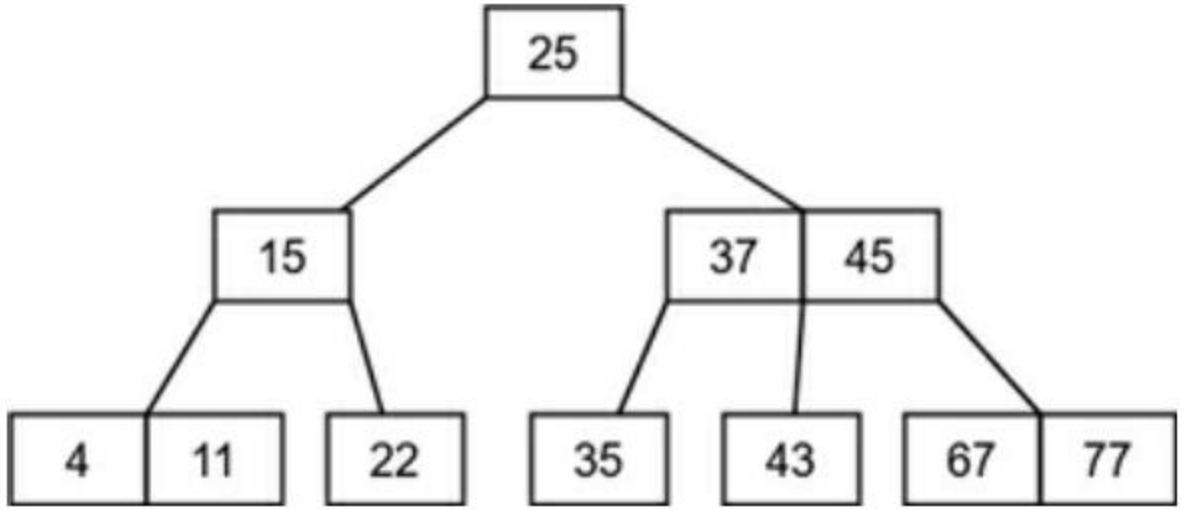
- Već smo videli da složenost umetanja, brisanja i pronalaženja vremena zavisi od visine stabla.
- Dakle, ako nađemo način da smanjimo visinu, takođe možemo da smanjimo količinu pristupa memoriji.
- Sve to može da se uradi pomoću takozvanog B-Drveta, a to je stablo za samo-balansiranje, u kojem se više čvorova kombinuje u jedan blok.
- Ideja ovde je da pustimo drvo da raste više u smislu **širine** nego u smislu visine.

Definicija B-Drveta reda m:

1. Svi listovi-čvorovi su na istom nivou
2. Svaki čvor može imati maksimalno m dečjih-čvorova
3. Svaki unutrašnji čvor (znači ne čvor sa listom) ima minimum [m/2] dečijih-čvorova
4. Osnovni čvor ima minimum dva dečja-čvora.
5. Svaki čvor ima jedan dečiji-čvor više nego što ima ključeva

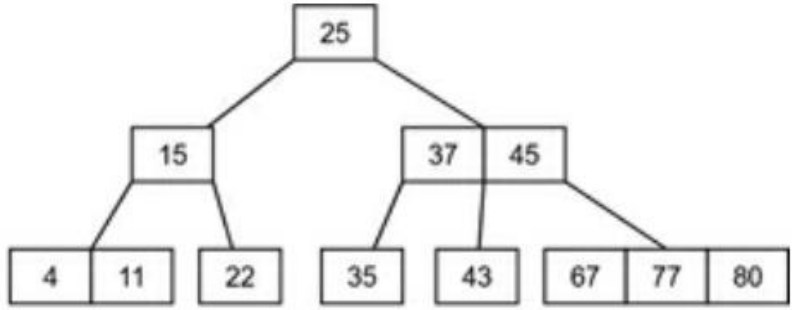
Samobalansirajuća stabla – B stablo

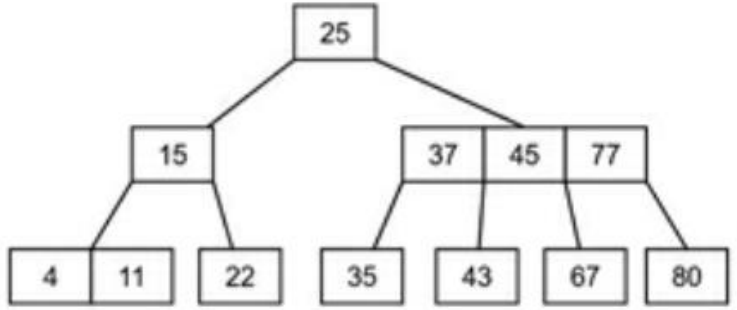
- Primer B-stabla trećeg reda. Možete videti da jedan čvor može da ima najviše tri dečja čvora i da svaki čvor ima jedno dete više nego što ima ključeva.
- Kada želimo da nađemo vrednost, tačno znamo gde da se krećemo.

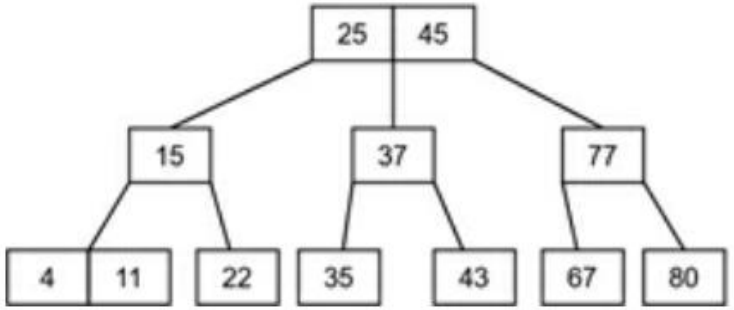


Samobalansirajuća stabla – B stablo

- Primer B-stabla trećeg reda i operacije **ubacivanja** vrednost 80 u dato B stablo**.**
- Ubacujemo 80 na kraj ali dobijamo 3 elementa u bloku što nije dozvoljeno.
- Zato ga guramo gore...ista logika za ostale operacije.
- Kompleksnost izvršavanja je u log vremenu **O(log n).**

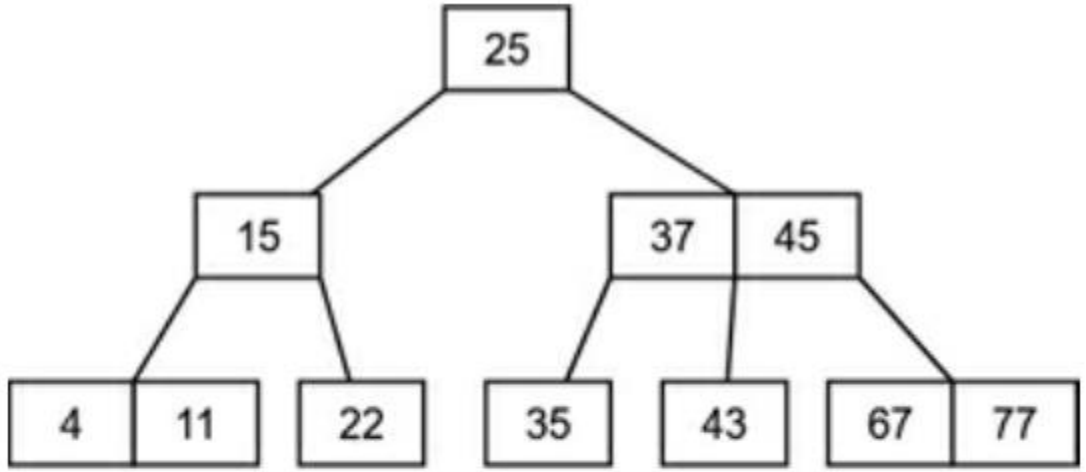


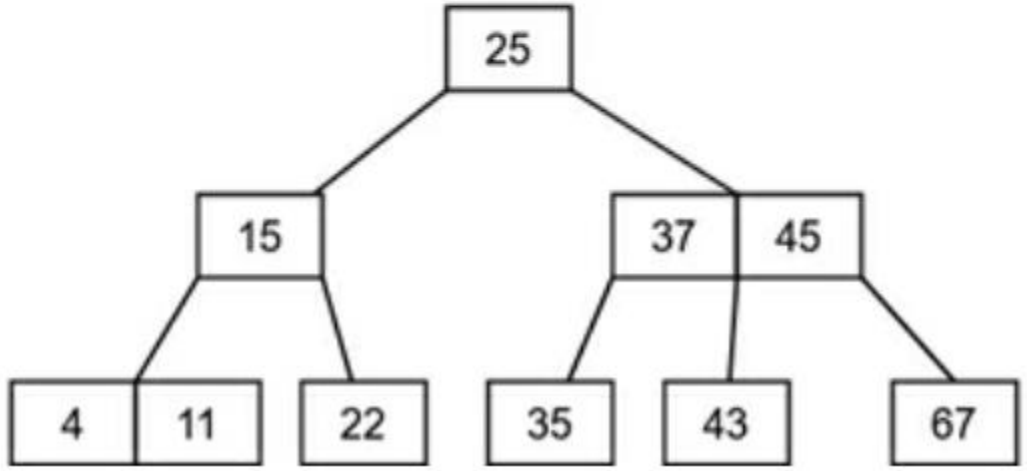




Samobalansirajuća stabla – B stablo

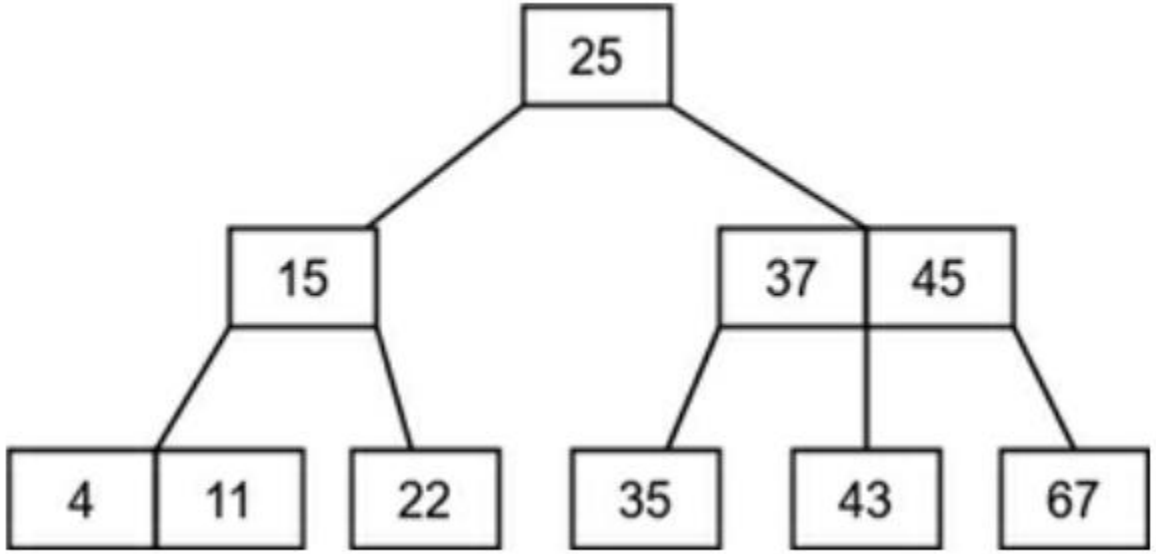
- Primer B-stabla trećeg reda i operacije **brisanja** vrednost 77**.**
- Trivijalno ne treba balansirati sva pravila i dalje važe.





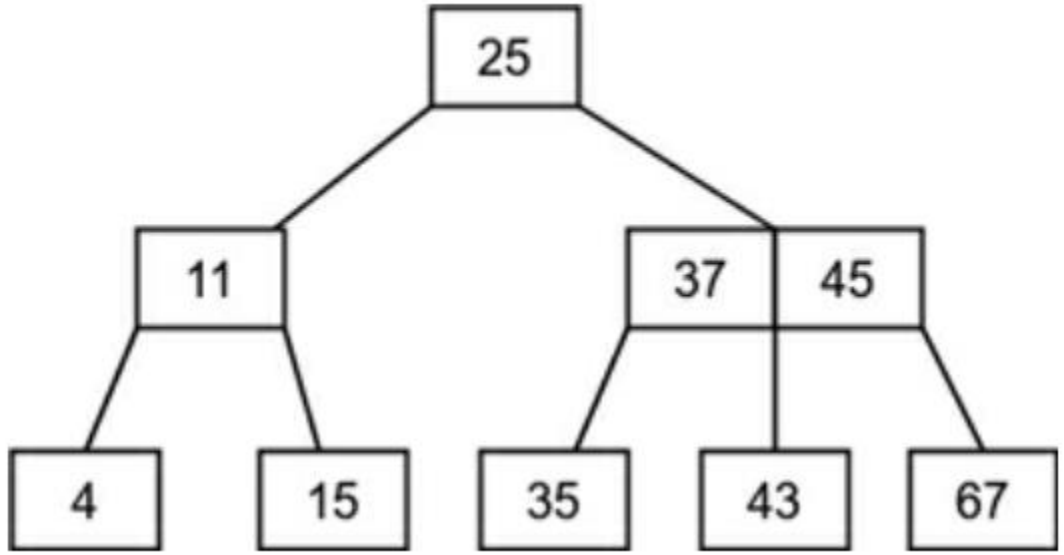
Samobalansirajuća stabla – B stablo

- Primer B-stabla trećeg reda i operacije **brisanja** vrednost 22**.**
- Problem jer se ugrožava pravilo za roditelja da ima barem 2 dete-čvora!



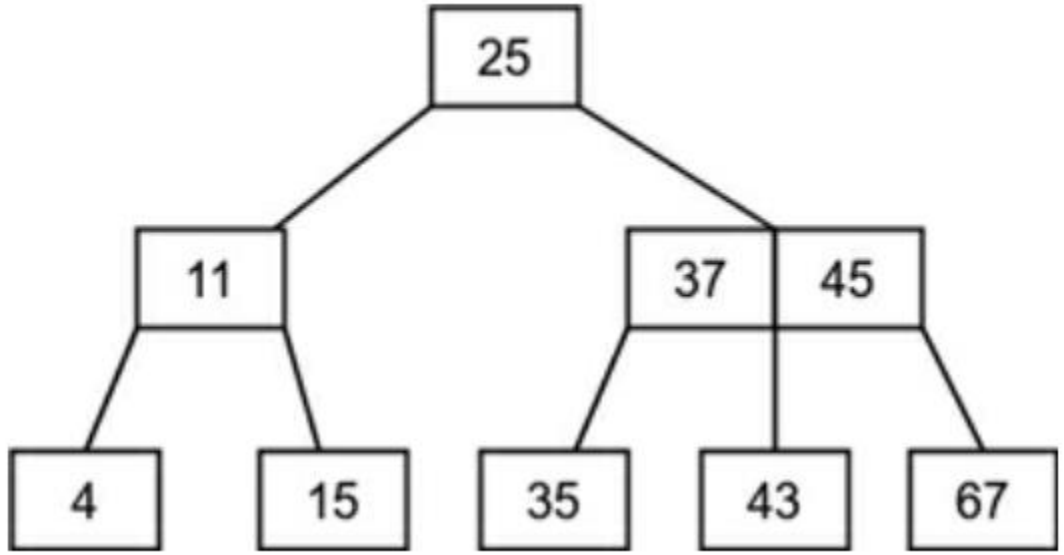
Samobalansirajuća stabla – B stablo

- Rešenje: rotacija
- Vrednost 15 zamenjuje vrednost 22, a vrednost 11 preuzima nadređeni i zamenjuje 15.



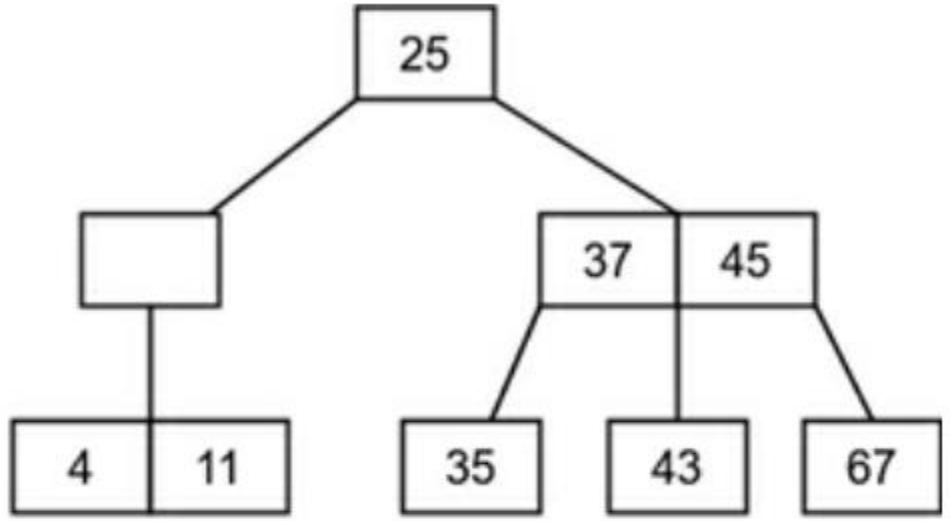
Samobalansirajuća stabla – B stablo

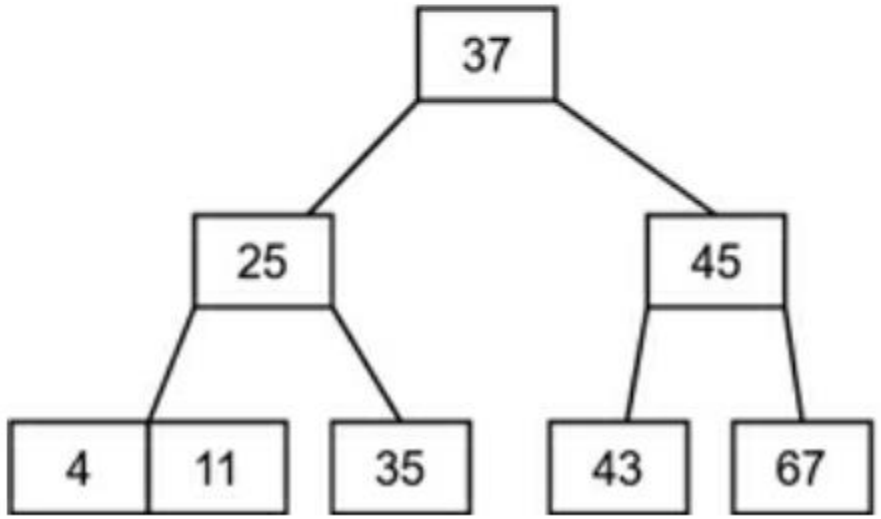
- Šta ako brišemo sada 15?
- Opet ulazimo u problem.
- Nemamo dovoljno ključeva za to. Zbog toga se okrećemo na drugačiji način i kombinujemo 4 i 11 u jedan nod.



Samobalansirajuća stabla – B stablo

- Okrećemo na drugačiji način i kombinujemo 4 i 11 u jedan nod.
- Roditelj ovog noda je prazan.
- Zato moramo da pozajmimo ključ od njegovog brata i zamenimo koren sa njim.
- Obratite pažnju na to da levo dete desnog pod-drveta tada postaje desno dete levog pod-drveta.

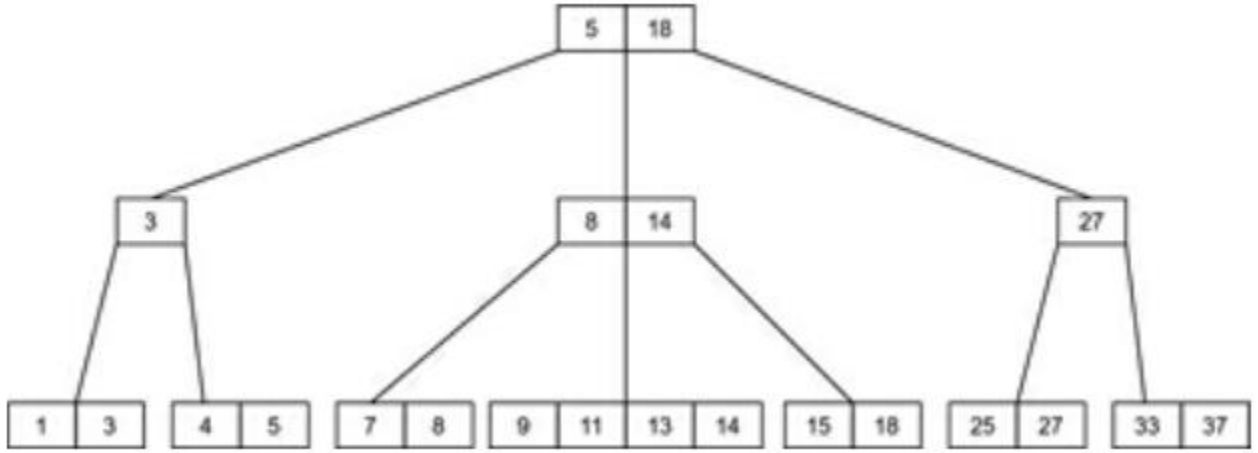




Samobalansirajuća stabla – B stablo

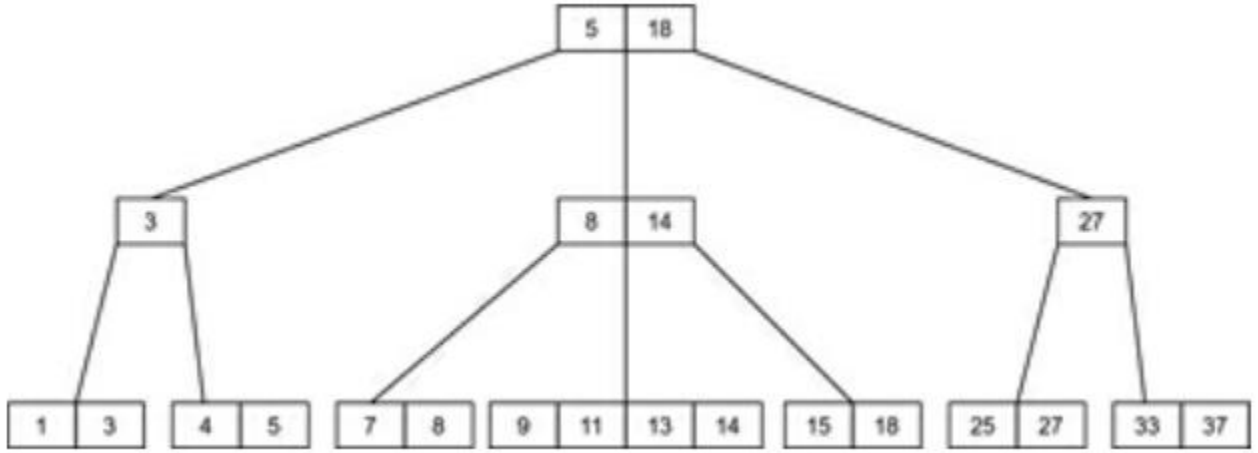
- B-Drveće je veoma efikasno i kao u AVL drveću, sve velike operacije imaju logaritamski najgoru složenost izvršenja.
- Povrh ovih, njihova blokčein struktura je super korisna kada radimo sa sporim spoljnim skladišnim medijima, pošto nam omogućava da umanjimo količinu pristupa.

Samobalansirajuća stabla – B * drvo



- Slično B stablu.
- Najveća razlika je u tome što su svi **podaci sadržani u čvorovima lista.**
- Sve vrednosti koje dolaze pre sloja lista se koriste samo za **navigaciju**.

Samobalansirajuća stabla – B * drvo



- Vrednost ključa za navigaciju je uvek najveći ključ levog podređenog čvora.
- Količina ključeva u listu je određena parametrom koji se zove k*.
- U našem lisnatom čvoru možemo imati k* do 2k* elemenata.
- Gore navedena stabla imaju parametre m = 3 i k* = 2 i zato čvorovi lista mogu imati dva, tri ili četiri ključa.

Samobalansirajuća stabla – B * drvo

- Kompleksnosti izvršavanja za pretraživanje, umetanje i brisanje su iste kao i za normalno B-Drveće.
- Međutim, u B*-Drveću uvek moramo da idemo na čvorove lista da bismo pristupili podacima.
- To ipak ne menja prosečan broj potrebnih koraka, s obzirom da B*-Drveće obično ima **manju visinu** od običnog B-Drveća.
- Generalno gledano, B-Drveće i B*-Drveće se često koriste u sistemima baza podataka i sistemima datoteka. Njihova logaritamska kompleksnost i struktura blokova su savršeni za ova polja.

Hashing-heširanje

- Obično se hashing čuje u domenu passworda i Cyber security-ja.
- Kakve veze ima sa nama u okviru data struktura i algoritama?
- Problem sa velikim setovima podataka gde je poprilično neefikasno tražiti vrednost poređenjem vrednosti ključeva, hashing nam može pomoći da izračunamo njihovu poziciju –> da ne lutamo „na slepo“
- Dakle definišemo hash funkciju kojom izračunavamo poziciju tražene vrednosti.

Hashing-heširanje: primer

$$h(s) = ord(s[0])$$

- U ovom slučaju hash funkcija samo daje num vrednost za svaki unutrašnji znak, gde je parametar s ime. Ovde ćemo pretpostaviti da je „A“ 0, „B“ 1 itd.

0 (“A”)	Anna 432, Angela 787
1 (“B”)	Bob 892
2 (“C”)	
3 (“D”)	Dora 990, Daniel 211
4 (“E”)	
5 (“F”)	
6 (“G”)	
7 (“H”)	Henry 232
…	…
25 (“Z”)	

Hashing-heširanje: primer

$$h(s) = ord(s[0])$$

- Na ovaj način bi mogli kreirati hash tabelu, gde bi za Geralda npr izračunali da treba da se pronađe na 6tom mestu.
- Problem: Šta ćemo sa Gailom?

0 (“A”)	Anna 432, Angela 787
1 (“B”)	Bob 892
2 (“C”)	
3 (“D”)	Dora 990, Daniel 211
4 (“E”)	
5 (“F”)	
6 (“G”)	
7 (“H”)	Henry 232
…	…
25 (“Z”)	

Hashing-heširanje: primer

$$h(s) = ord(s[0])$$

- Ne možemo sačuvati dve različite vrednosti pod istom adresom.
- Došlo je do tzv **kolizije**, da nje nema kompleksnost izvršenja bi bila u konst. vremenu, a sa njom idemo u lin vreme!

0 (“A”)	Anna 432, Angela 787
1 (“B”)	Bob 892
2 (“C”)	
3 (“D”)	Dora 990, Daniel 211
4 (“E”)	
5 (“F”)	
6 (“G”)	
7 (“H”)	Henry 232
…	…
25 (“Z”)	

Hashing-heširanje

- Moramo odgovoriti zato na dva ključna pitanja:
- Koja je DOBRA hash funkcija?
- Kako efikasno rešiti koliziju?

Hashing-heširanje: hash funkcija

Metoda **deljenja**: $h(k) = k \bmod m$

- k je prirodan broj. Ova metoda obezbeđuje izuzetno brzo heširanje, ali treba inteligentno izabrati m. Dobra praksa su prime brojevi. Loša praksa su stepeni od 2 ili 10.

Metoda **množenja**: $h(k) = \lfloor m(k * A - \lfloor k * A \rfloor) \rfloor$

- A je iracionalan broj. Njega množimo sa našim ključem.Nakon toga oduzimamo intedžer deo rezultata, čime dobijamo rezultat između 1 i 0. Nakon toga množimo rezultat sa veličinom tabele m i zaokružujemo na dole.
- Perfektna vrednost za A je tzv **zlatni racio** 0.6180339887 $A = \phi^{-1} = \frac{\sqrt{5}-1}{2}$
- Formula za taj racio je data.

Hashing-heširanje: rešenje kolizije

Rešenje **kolizije**:

- Koju god hash funkciju da uzmemo uvek postoji mogućnost za dolazak do kolizije.
- S obzirom da naši algoritmi nisu komplikovani kao SHA-256 na primer to se može često dogoditi.

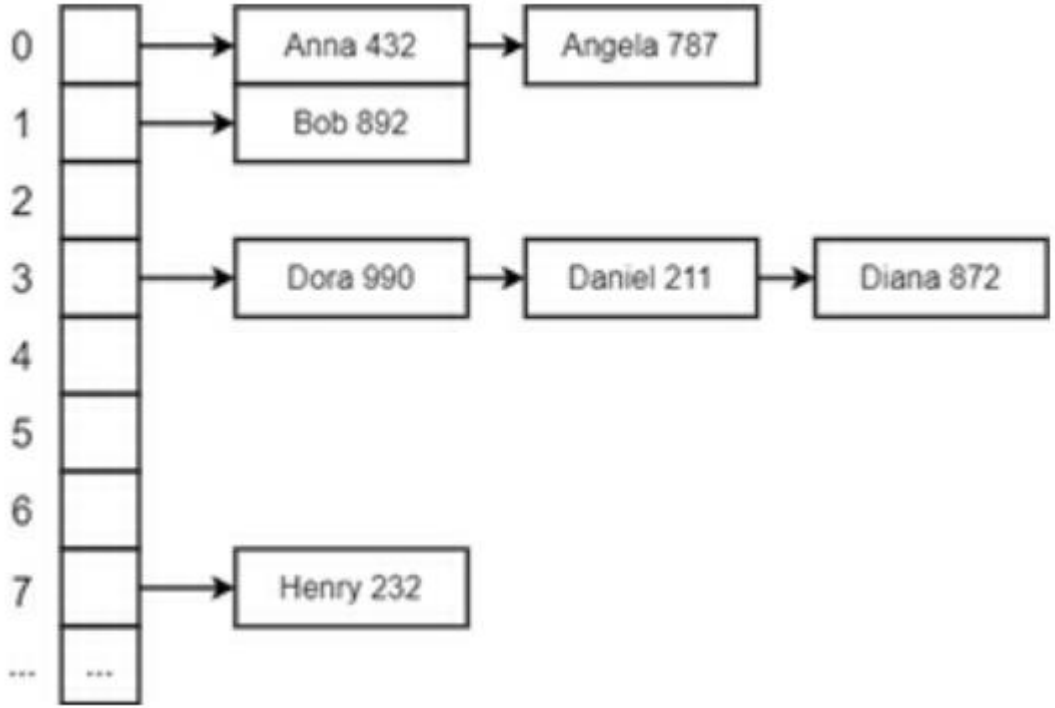
Hashing-heširanje: rešenje kolizije

Posebno listanje:

- Imamo dva scenarija. Prvi kada prvi put element dolazi u našu hash tabelu, kreiramo novu listu i on postaje Head liste, uradi se append i doda se na listu

.

- Da bi došli do njega moramo se pozicionirati na odgovarajuću listu pa po čvorovima liste ići dok ne dođemo do traženog elementa



Hashing-heširanje: rešenje kolizije

Otvoreno adresiranje:

- Ovde smeštamo podatke u običan niz.
- Takođe dodajemo flag sa tri stanja: slobodno, zauzeto i opet slobodno.
- Primenjujemo **probing** mehanizam za koliziju. Ako dodje do nje idemo na prvo naredno slobodno mesto.
- Opet slobodno nam obezbeđuje info da je element sa tog mesta obrisan, što nam je važna inoformacija kod pretraživanja.

0 (“A”)	Anna 432	Occupied
1 (“B”)	Angela 787	Occupied
2 (“C”)		Free
3 (“D”)	Dora 990	Occupied
4 (“E”)		Free
5 (“F”)		Free
6 (“G”)		Free
7 (“H”)	Henry 232	Occupied
…	…	…
25 (“Z”)		Free

Hashing-heširanje: rešenje kolizije

Linerni probing:

- Dodavanje parametra i kod naše hash funkcije.
- Ovaj parametra ukazuje na to koliko je već bilo kolizija.

$$h(k,i) = (h'(k) + i) \ mod \ m$$

$$m = 8, \ \ h'(k) = k \ mod \ m$$

- Primer: kolizija nastaje na k 22 i 14

Inserted values for k	10	19	31	22	14	16
Resulting hashes of h'(k)	2	3	7	6	6	0

Hashing-heširanje: rešenje kolizije

$$m = 8,\ \ h'(k) = k\ mod\ m$$

Rešenje problema probingom:

- Probamo sa i = 1, pa ako ne ide nastavljamo dalje dok ne pronadjemo slot koji nije pre njega korišćen.

Inserted values for k	10	19	31	22	14	16
Resulting hashes of h'(k)	2	3	7	6	6	0

$$h(14,1) = (h'(14) + 1)\ mod\ 8 = 7\ mod\ 8 = 7$$

$$h(14,2) = (h'(14) + 2)\ mod\ 8 = 8\ mod\ 8 = 0$$

S obzirom da je 16 sa indeksom 0 isti proces ponovimo za 16 i dođemo do 1

0	1	2	3	4	5	6	7
14	16	10	19			22	31

Hashing-heširanje: rešenje kolizije

Kvadratni probing:

- Da bi rešili probem lineranog probinga sa izborom sledeće pozicije, kvadratni probing posmatra sa kvadratnim uvećavanjem razmaka.
- Time ne gledamo samo na poziciju do elementa kada dođe do kolizije nego na udaljene pozicije takođe.

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Hashing-heširanje: rešenje kolizije

Dupli hešing:

- Ovo je optimalno rešenje kolizije.

- Ima skoro istu efikasnost kao unformno hashiranje

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$