FAKULTET ORGANIZACIONIH NAUKA
UNIVERZITET U BEOGRADU

Siniša Nešković

STRUKTURE PODATAKA I ALGORITMI

- skripta -

Beograd, maj 2006.

Sadržaj:

PRETRAŽIVANJE

U ovom delu ćemo razmotriti problem pronalaženja određene informacije u velikom skupu podataka. Kao što ćemo videti, izvesne metode organizacije podataka (t.j. strukture podataka) čine proces pronalaženja efikasnijim. S obzirom da je proces pretraživanja vrlo čest u obradi podataka, poznavanje metoda i tehnika organizacije podataka pretraživanja je vrio važno.

Pre nego što predemo na konkretne metode uvedimo neke osnovne termine. **Tabela** ili **datoteka** je grupa elemenata od kojih se svaki naziva **zapis** ili **čvor**. Ove termine upotrebljavaćemo u najopštijem smislu i ne bi trebalo da se pomešaju sa sličnim terminima u PASCAL-u ili COBOL-u.

Svakom zapisu je pridružen **ključ** kojim se on može razlikovati od ostalih zapisa.Odnos između ključa i zapisa može biti različit. U najprostijem slučaju ključ je unutar zapisa kao jedan njegov deo (polje). Takvi ključevi se nazivaju **internim.** U ostalim slučajevima, može postojati posebna tabela ključeva sa pokazivačima (logičkim ili fizičkim) na zapise. Takvi ključevi se nazivaju **eksternim**. Za svaku datoteku postoji najmanje jedan skup ključeva (moguće je više) koji su jedinstveni, tj. ne postoje dva zapisa sa istim ključem. Takvi ključevi se nazivaju **primanim.** Na primer, ako je datoteka realizovana kao niz, indeks nekog elementa u nizu je ključ tog elementa. Pošto bilo koje polje ili kombinacija polja nekog zapisa može biti ključ u nekoj određenoj primeni, ključevi ne moraju uvek biti jedinstveni. Na primer, ako se u datoteci studenata ime studenta uzme kao ključ, takav ključ verovatno neće biti jedinstven. Ovakvi ključevi se nazivaju **sekundarni** ključevi.

Proces pretraživanja je algoritam koji prihvata argument *a* i pronalazi jedan ili više zapisa u datoteci ili tabeli čija vrednost ključa je *a*. Može se desiti da proces pretraživanja bude neuspešan, t.j. da ne postoji zapis sa takvim ključem. U tom slučaju, često je poželjno u datoteku ubaciti zapis sa takvim ključem. Takvo pretraživanje se naziva pretraživanje sa ubacivanjem.

Primetite da do sada nismo ništa rekli o strukturi podataka u kojoj se datoteka implementira. To može biti niz, spregnuta lista, stablo ili čak mreža (graf). Pošto su različite metode pretraživanja pogodnije za neke strukture podataka, izbor implementacije datoteke (t.j. izbor strukture podataka) je često određen metodom pretraživanja koja se ima na umu.

Sekvencijalno pretraživanje

Najprostiji algoritam pretraživanja je sekvencijalno pretraživanje koji se primenjuje na nizove i spregnute liste. On se sastoji u tome da se ključ svakog zapisa ispituje u redosledu kako su zapisi smešteni sve dok se ne nađe traženi zapis.
Neka se u nizu čuvaju zapisi deklarisani kao

```
class Element
{
      public int Key;
      public object Value;
}
```

i neka promenljiva ***Poz*** sadrži indeks zapisa koji tražimo ako on postoji ili -1 ako on ne postoji. Algoritam sekvencijalnog pretraživanja izgleda:

```
public static int SequentialSearch(Element[] aArray, int aTargetKey)
{
	int Poz = -1;
	for (int i = 0; i < aArray.length; i++)
	{
		if ((aArray[i].Key == aTargetKey)
		{
			Poz = i;
			break;
		}
	}
	return Poz;
}
```

Realizacija datoteke kao spregnute liste ima prednost u tome što se veličina memorijskog prostora potrebnog za smeštanje datoteke može menjati prema potrebi. Definišimo prvo čvor liste na sledeći način:

```
	class CvorListe
	{
		int Kljuc;
		CvorListe Sledeci;
		public CvorListe(int aKljuc, CvorListe aSled)
		{
			Kljuc = aKljuc;
			Sledeci = aSled;
		}
	}
```

Pretpostavimo da promenljiva Glava pokazuje na prvi zapis datoteke, a polje Kljuc sadrži vrednost ključa zapisa koji tražimo. Algoritam koji sekvencijalno pretražuje datoteku i ako je pretraživanje neuspešno vrši ubacivanje, izgleda ovako:

```
		CvorListe Prethodni = null;
		CvorListe Tekuci = Glava;
		while (Tekuci != null)
		{
			if (Tekuci.Kljuc == TargetKey)
				return tekuci;
			Prethodni = Tekuci;
			Tekuci = Tekuci.Sledeci;
		}
		// nije nađen
		CvorListe novi = new CvorListe(kljuc, null);
		if (Prethodni == null) // lista je bila prazna
			Glava = novi;
		else
			Prethodni.Sledeci = novi;
		return novi;
```

Efikasnost sekvencijalnog pretraživanja

Ako pretpostavimo da nema ubacivanja i izbacivanja i da vršimo sekvencijalno pretraživanje datoteke veličine n zapisa, tada broj poređenja koje moramo da obavimo prilikom pretraživanja zavisi od toga gde se nalazi zapis sa ključem jednakim argumentu. Ako je zapis na početku, samo jedno poređenje je potrebno; ako je zapis poslednji, tada je potreban n poređenja. U proseku, za uspešno pretraživanje je potrebno (n+1)/2 poređenja, a za neuspešno n poređenja (u oba slučaja O(n)).

Međutim, čest slučaj je da se nekim zapisima češće pristupa nego nekim drugim. Ako se takvi zapisi stave na početak datoteke, tada prosečan broj poređenja može značajno smanjiti. Pretpostavimo da *P(i)* označava verovatnoću da se zapis na *i* -toj poziciji traži i da važi:

$$P(1) * P(2) * P(3) +, .., + P(n) = 1.$$

Tada je prosečan broj poređenja jednak :

$$P(1) + 2P(2) * 3P(3) +... + nP(n).$$ (Zašto ?)

Znači, za datu veliku datoteku, sortiranjem zapisa datoteke u opadajućem redosledu verovatnoća traženja postiže se efikasnije pretraživanje.

Na žalost, verovatnoće *P(i)* su vrlo retko poznate unapred. a često se ta verovatnoća menja u vremenu. Zato bi bilo poželjno imati algoritam koji će stalno da vrši preuređivanje datoteke tako da zapisi kojima se češće pristupa budu bliži početku. Postoji više metoda kako se ovo može postići. Jedna od njih, poznata kao **prebaci na početak**, efikasna samo za spregnute liste, uvek kada je pretraživanje uspešno nađeni zapis pomeri sa njegove pozicije na početak liste. Drugi metod **je metod transpozicije** u kome se pretraženi zapis zameni sa svojim prethodnikom.Tako će zapisi kojima se *češće* pristupa postepeno doći na početak liste.

Pretraživanje sortirane datoteke

Ako je datoteka smeštena u rastućem ili opadajućem redosledu vrednosti ključeva, jedna očigledna prednost u odnosu na nesortiranu datoteku kod sekvencijalnog pretraživanja je da se može otkriti da nema zapisa sa traženim ključem pre nego što ispitamo sve zapise. Naime, čim prilikom pretraživanja naiđemo na zapis čiji ključ je veći (u slučaju da su zapis sortirani u rastućem redosledu) od traženog tada možemo zaključiti da ne postoji zapis sa traženim ključem.

Međutim, postoje metode pretraživanja koje takođe pretražuje sortiranu datoteku, ali su mnogo efikasnije od sekvencijalnog pretraživanja.

Binarno pretraživanje

Najefikasniji metod za pretraživanje sortirane tabele bez upotrebe dodatnog prostora je binarno pretraživanje. Argument traženja se poredi ključem zapisa koji se nalazi u sredini tabele. Ako su jednaki onda se pretraživanje uspešno završava. Inače, bilo jedna ili druga polovina tabele se na isti način pretražuje, a koja, zavisi od rezultata poređenja i redosleda sortiranja tabele. Ako je argument manji od ključa zapisa na sredini tabele i ako je tabela sortirana u rastućem redosledu tada se pretražuje prva polovina tabele; inače druga. Dakle, algoritam je rekurzivan:

```
public static int BinRek(Element[] aArray,int aTarget,int aLeft, int aRight)
{
      if (aLeft <= aRight)
      {
            int middle = (aLeft + aRight) / 2;
            if (aArray[middle].Key == aTarget)
                  return middle;

            if (aArray[middle].Key <= aTarget)
                  return BinRek(aArray, aTarget, middle, aRight);
            else
                  return BinRek(aArray, aTarget, aLeft, middle);
      }
      return -1;
}
```

Međutim, sporost izvršavanja rekurzivne procedure čini je nepogodnom za primene gde je efikasnost primarna. U takvim slučajevima se koristi nerekurzivna procedura:

```
            public static int BinarySearch(Element[] aArray, int aTarget)
            {
                  int Result = -1;
                  int left = 0;
                  int right = aArray.length - 1;
                  while (right >= left)
                  {
                        int middle = (left + right) / 2;
                        if (aArray[middle].Key > aTarget)
                              right = middle - 1;
                        else if (aArray[middle].Key < aTarget)
                              left = middle + 1;
                        else
                        {
                              Result = middle;
                              break;
                        }
                  }

                  return Result;
            }
```

Svaki korak u binarnom pretraživanju smanjuje broj kandidata na pola. Znači, maksimalni broj poređenja ključeva je približno $\log_2 N$ gde je N broj zapisa. Za malo N performanse algoritma nisu mnogo bolje od sekvencijalnog pretraživanja, ali za veliko N binarno pretraživanje je superiorno. Za $N = 10^6$ broj poređenja je približno 20 u odnosu na 500.000 kod sekvencijalnog pretraživanja.

Na žalost, binarno pretraživanje se može primenjivati samo na sortirane nizove (tj. kada su zapisi fizički susedni). To ima za posledicu da je ubacivanje novih zapisa otežano, jer se mora vršiti pomeranje zapisa. Sve to ograničava praktičnu primenu binarnog pretraživanja.

Interpolaciono pretraživanje

Ukoliko su vrednosti ključeva uniformno raspoređene između minimalne vrednosti ključa **Min** i maksimalne vrednosti ključa **Max** i te vrednosti su unapred poznate, tada se može postići pretraživanje sortiranog niza koje je brže od binarnog. Naime, pod napred navedenim uslovima, možemo pretpostaviti da postoji veza između pozicije vrednosti ključa u skupu vrednosti ključeva i pozicije u nizu zapisa sa tim ključem.

Na primer, kada pretražujemo abecedno sortiran telefonski imenik, a tražimo nekoga ko se preziva na B tada imenik nećemo otvoriti na sredini (kao kod binarnog pretraživanja) nego bliže početku imenika, jer pozicija slova B je bliža početku abecede, pa pretpostavljamo da će se i osoba sa prezimenom na B nalaziti bliže početku imenika.

Dakle, ako sa P_{min} i P_{max} označimo pozicije zapisa sa minimalnom odnosno maksimalnom vrednošću ključa, a sa MAX i MIN maksimalnu i minimalnu vrednost, tada poziciju P_k zapisa sa ključem K možemo odrediti iz proporcije:

$$\frac{P_k - P_{\min}}{P_{\max} - P_{\min}} = \frac{K - MIN}{MAX - MIN}$$

odnosno:

$$P_k = P_{\min} + \frac{K - MIN}{MAX - MIN} * (P_{\max} - P_{\min})$$

Postupak pretraživanja je sledeći. Na početku, P_{min} postavimo na 1 a P_{max} na N. Pozicija P_k zapisa sa traženim ključem se približno odredi (interpolira; otuda naziv algoritma) po prethodnoj formuli. Ako na toj poziciji nije zapis sa ključem K tada se postupa slično kao kod binarnog pretraživanja. Ako je K veće od ključa zapisa na poziciji P_k, onda P_{min} postaje $P_k + 1$ (tj. levi kraj niza). Ako je K manje od P_k, tada P_{max} postaje $P_k - 1$ (tj. desni kraj niza). Pretraživanje se rekurzivno nastavlja nad nizom sa promenjenim granicama sve dok se ne nađe zapis ili otkrije da ga nema.

Pod pretpostavkom da su ključevi uniformno raspoređeni, interpolaciono pretraživanje zahteva prosečno $\mathbf{\log_2(\log_2 N)}$ poređenja, što je bolje od binarnog pretraživanja.

Međutim ako ključevi nisu uniformno raspoređeni tad performanse algoritma mogu biti znatno pogoršane. U najgorem slučaju, vrednost P_k može biti $P_{min}+ 1$ ili $P_{max} - 1$ odnosno interpolaciono pretraživanje se degeneriše u sekvencijalno (dok binarno i u najgorem slučaju ostaje $\log_2 N$).

U praksi, česta je situacija da se ključevi grupišu oko neke vrednosti tj. da nisu uniformno raspoređeni. Na primer, u telefonskom imeniku je mnogo više Petrović-a nego Željić-a. Varijanta ovog algoritma, zvana **robusno interpolaciono pretraživanje** ili **brzo (fast search**), pokušava da popravi performanse i u slučaju neuniformne raspodele ključeva. To se postiže uvođenjem promenljive **Razmak (R)** tako da je uvek $\mathbf{P_k - P_{min}}$ i $\mathbf{P_{max} - P_k}$ veće od **R.** U početku, R je postavljen na vrednost:

$$R = \sqrt{P_{\max} - P_{\min} + 1}\,,$$

a **Pom** (pomoćna promenljiva) na:

$$Pom = P_{\min} + \frac{K - MIN}{MAX - MIN} * (P_{\max} - P_{\min})$$

P_k je sada jednako:

$$P_k = Minimum(P_{\max} - R, Maximum(Pom, P_{\min} + R)$$

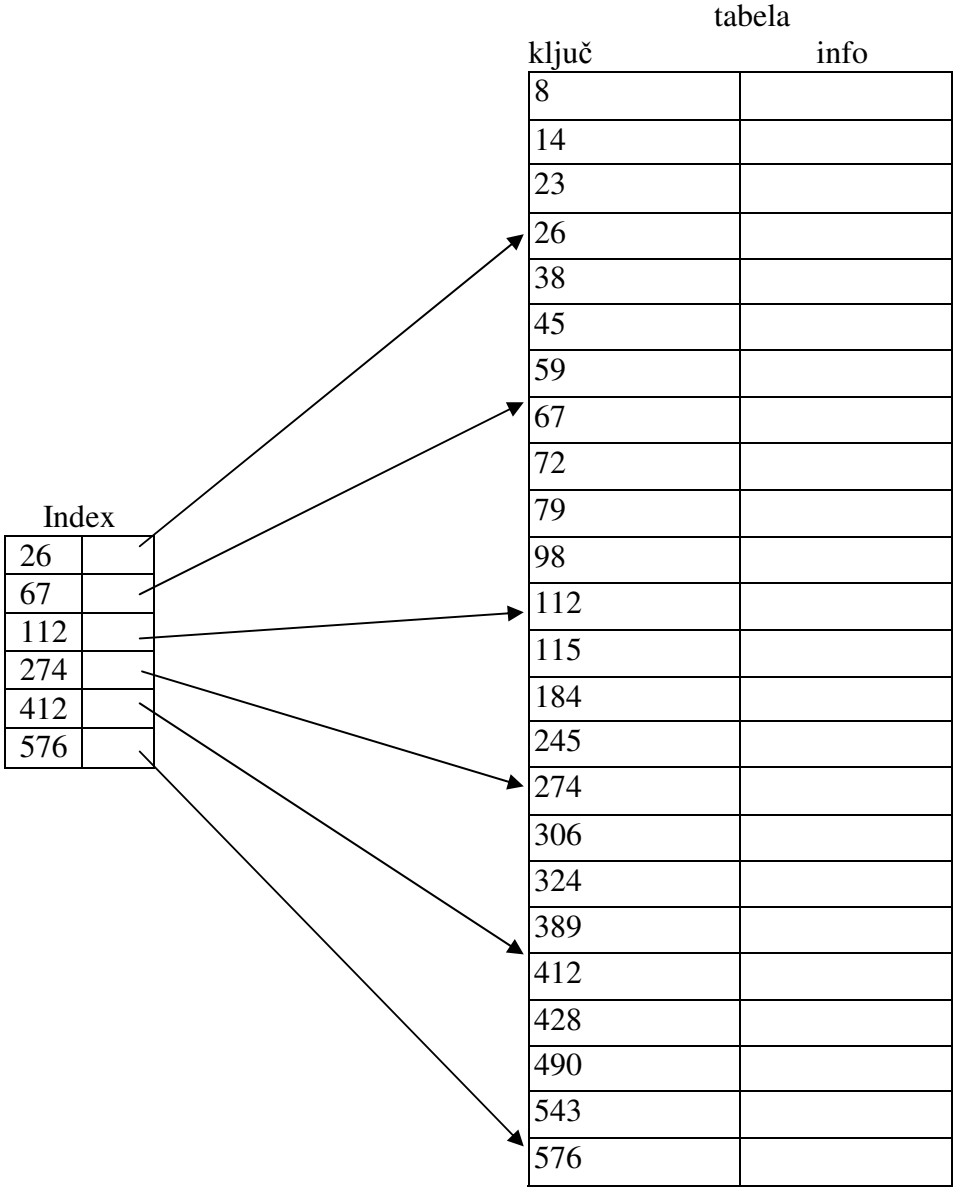
Zapravo, ovim se garantuje da je P_k, sledeća pozicija koja će se ispitati, najmanje **R** pozicija udaljena od krajeva intervala $\mathbf{P_{min}}$ i $\mathbf{P_{max}}$. Kada otkrijemo da se K (argument traženja) nalazi u većem delu intervala vrednost **Razmak** se duplira. Na taj način se izbegavaju grupisanja ključeva sa bliskom vrednošću. Kada je K u manjem delu intervala, tada se **Razmak** vraća na kvadratni koren novog intervala.

Robusno interpolaciono pretraživanje u proseku zahteva $\mathbf{\log_2(\log_2 N)}$, a u najgorem slučaju $\mathbf{(\log_2 N)^2}$ poređenja.

Ipak, treba reći da i obično i robusno interpolaciono pretraživanje zahteva veliki broj izračunavanja što može imati značajne efekte na performanse algoritma.

Indeks-sekvencijalno pretraživanje

Ova metoda pored sortirane datoteke koristi i dodatnu pomoćnu tabelu zvanu **indeks**. Sortirana datoteka veličine n je podeljena u k blokova sa m zapisa. Najveći ključ iz svakog bloka je smešten u indeks zajedno sa pokazivačem na blok iz koga je uzet. Sledeća slika ilustruje jedan primer:



Indeks je takođe sortiran. Pretraživanje počinje od indeksa traženjem prvog ključa koji je veći ili jednak traženom. Kada se naiđe na takav ključ onda se preko pokazivača pristupa bloku u kome se zapis pronalazi posle najviše *m* poređenja.

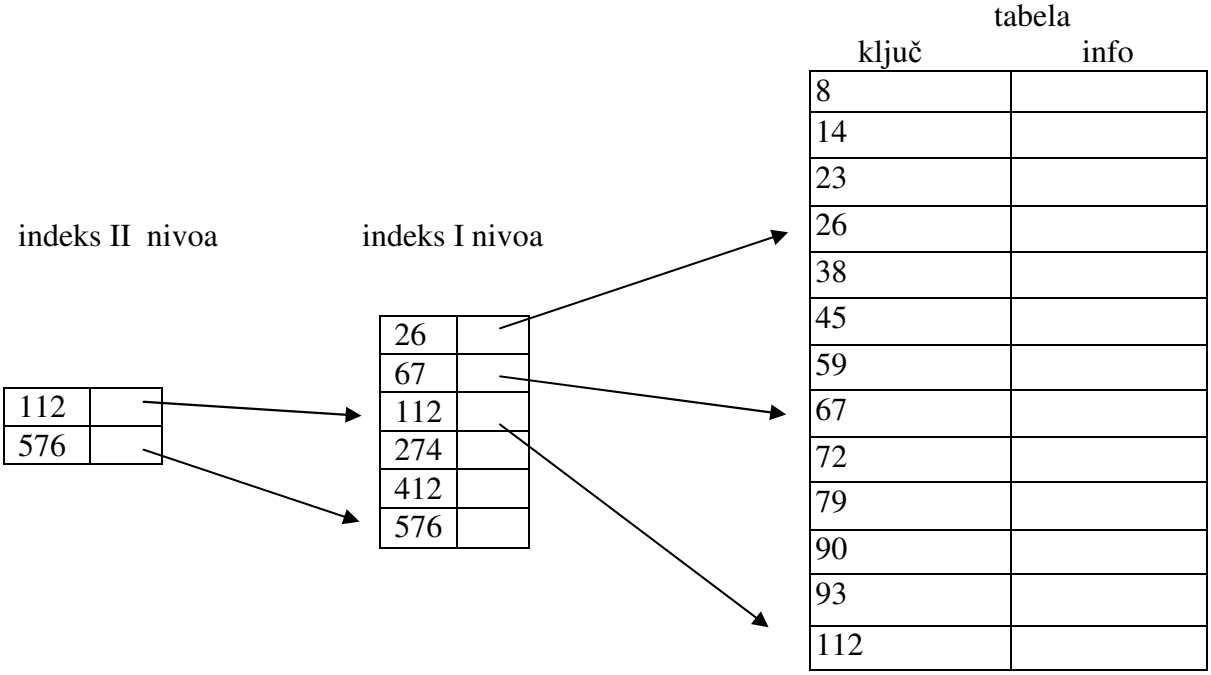
Prosečan broj poređenja prilikom pretraživanja datoteke je jednak prosečnom broju poređenja kod pretraživanja indeksa plus prosečnom broju poređenja kod pretraživanja bloka:

$$P = \frac{k}{2} + \frac{m}{2} = \frac{k + \frac{n}{k}}{2}, \text{ jer je } m = \frac{n}{k}$$

Optimalna vrednost za k se dobija iz uslova:

$$\frac{\partial P}{\partial k} = 0 \Rightarrow \frac{1}{2} + \frac{n}{2k^2} = 0 \Rightarrow k = \sqrt{n}$$

U slučaju da je k veliki broj, odnosno pretraživanje indeksa neefikasno, moguće je formirati novi indeks na drugom nivou kojim se indeksira indeks na prvom nivou:



Indeks-sekvencijalno pretraživanje se može primeniti kada su indeks i datoteka realizovani kao spregnuta lista ili niz. U prvom slučaju se povećava potreban prostor (zbog pokazivača), ali je ubacivanje i izbacivanje olakšano.

Međutim kako je indeks-sekvencijalna organizacija obično realizovana na diskovima, upotreba pokazivača značajno povećava broj pristupa disku, pa se mnogo češće koristi niz za implementaciju indeksa i datoteke (tj. zapisi se fizički smeštaju sekvencijalno što omogućava da se jednom pristupu disku učita ceo blok zapisa). Ali, tada je osnovni problem ubacivanje novih zapisa i ključeva, jer se može desiti da u bloku nema mesta. Tada se mora takav zapis smestiti u posebno izdvojen prostor (što pogoršava performanse pretraživanja) ili se vrši pomeranje zapisa da bi se napravilo mesta za nov zapis što prouzrokuje i promenu indeksa (skupo i nepraktično).

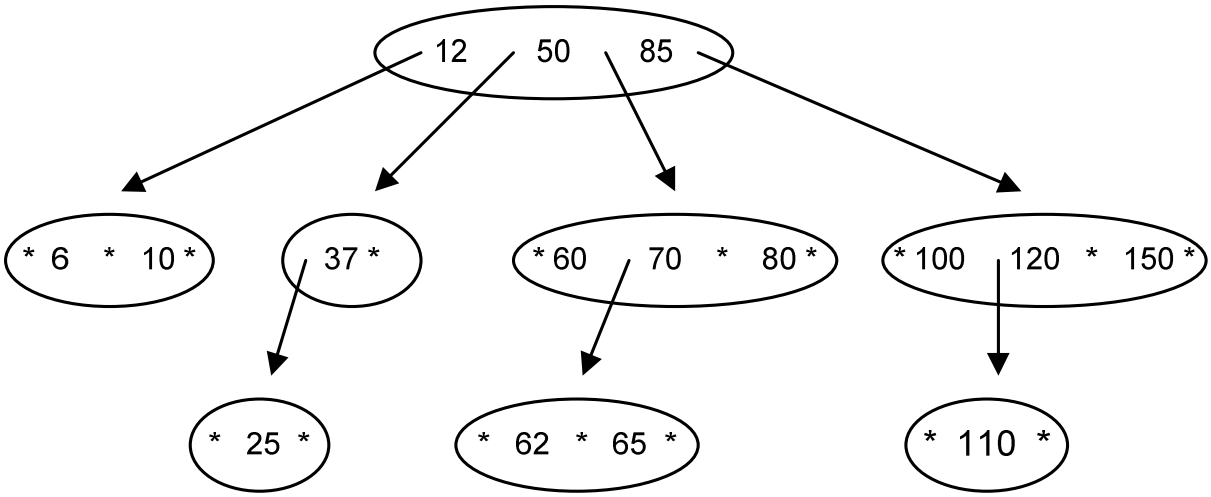
Pretraživanje stabala

U prethodnom delu smo videli nekoliko metoda pretraživanja kada je datoteka organizovana kao niz ili lista. Sada ćemo videti neke od metoda organizovanja datoteke kao stabla.

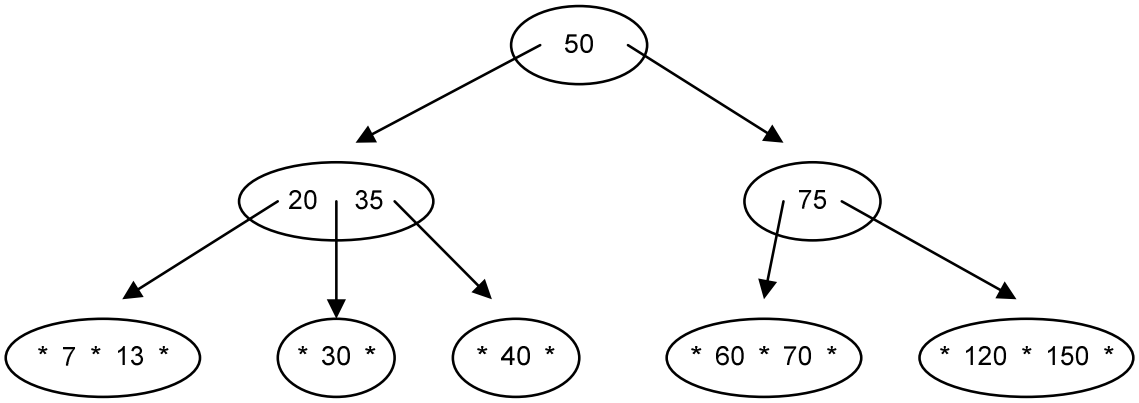
Binarno stablo traženja (BST) o kome je govoreno u poglavlju o stablima se može iskoristiti za implementaciju datoteke sa vrlo efikasnim pretraživanjem. Kao što smo videli, najefikasnije pretraživanje je onda kada je BST balansirano (svi listovi na istoj visini) koje u tom slučaju iznosi $\log_2 N$ tj. jednako je visini stabla. Međutim problem nastaje zbog ubacivanja novih čvorova jer mogu pokvariti balansiranost stabla što opet pogoršava efikasnost pretraživanja. Problem je rešen sa AVL stablima koja odgovarajućim rotacijama balansiraju stablo.

Višegransko (n-arno) stablo traženja

U slučaju BST, svaki čvor sadrži jedan ključ i pokazivače na 2 podstabla. Levo podstablo sadrži ključeve koji su manji od ključa korena a desno podstablo one ključeve koji su veći. Ovaj koncept možemo uopštiti. Opšte višegransko stablo traženja (VST) reda n je opšte stablo kod koga svaki čvor ima n ili manje podstabala, a sadrži ključeva za jedan manje od broja stabala. Znači, ako ima čvor ima 4 podstabla, onda ima 3 ključa. Pored toga, ako su s_1, s_2, ..., s_m m podstabla nekog čvora koji sadrži ključeve k_{1j}, k_2, ..., k_{m-1} u sortiranom rastućem redosledu tada svi ključevi u podstablu S_1 su manji ili jednaki od ključa k_1; svi ključevi u podstablu s_j, (2<= j <= m-1) su veći od k_j-1 a manji od k+1 i svi ključevi u podstablu S_m su veći od K_{m-1}.Podstablo S, se naziva levo podstablo ključa k_j, a njegov koren levi sin ključa k_j. Slično, S_j se naziva desno podstablo za ključ k_{j-1} a njegov koren desni sin za k_{j-1}. Sledi ilustracija nekoliko višegranskih stabala.



a) VST reda 4



b) balansirano VST reda 3

Čvor koji sadrži maksimalni broj ključeva i podstabla se naziva **pun** čvor. **Polulist** se naziva onaj čvor koji ima bar jedno prazno podstablo. Ako su svi polulistovi na istoj visini za VST se kaže da je balansirano.

Pretraživanje VST

Pretpostavimo da svaki čvor **p** ima: polje **brojač$_p$** čija vrednost je broj podstabala koje ima čvor p; zatim polja **s(p, l)** do **s(p, r)** (r = brojač$_p$) koji su pokazivači na podstabla čvora p; ključeve **k(p, l)** do **k(p, q)** (q = brojač$_p$-1). Podstablo na koje s(p, i) pokazuje sadrži sve ključeve u stablu između **k(p, i-1)** do **k(p, i).**

Pretpostavimo da imamo funkciju PretCvor(p, ključ) koja vraća najmanji ceo broj ***j*** takav da ključ <= k(p,j) ili brojač ako je ključ veći od svih ključeva čvora p. Rekurzivan algoritam pretraživanja PretVST(vst, ključ) koji vraća pokazivač na čvor koji sadrži ključ (ili *null* ako ga nema) i postavlja globalnu promenljivu Poz na poziciju ključ-a u čvoru izgleda ovako:

```
TreeNode PretVST(TreeNode RootVST, int kljuc)
{
      TreeNode node = RootVST;
      if (node == null)
      {
            Poz = 0;
            return null;
      }
      int i = PretCvor(node, kljuc);
      if (kljuc == k(node, i))
      {
            Poz = i;
            return node;
      }
      return PretVST(s(node, i), kljuc);
}
```

Funkcija PretCvor pronalazi poziciju najmanjeg ključa u čvoru koji je veći ili jednak od argumenta pretraživanja. Ovo se može postići bilo sekvencijalnim bilo binarnim pretraživanjem. Izbor zavisi od broja ključeva u čvoru tj. od reda VST. Čak je moguće ključeve unutar čvora organizovati kao binarno stablo.

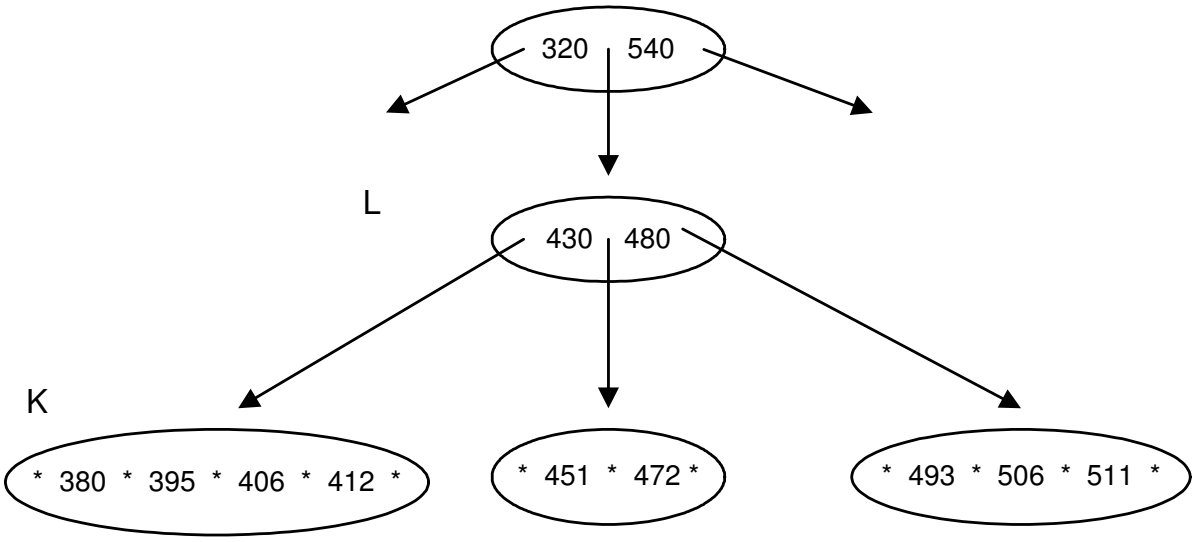
Očigledno je da je u slučaju balansiranog VST pretraživanje najefikasnije. Međutim, problem je održati balansiranost kod ubacivanja ili izbacivanja čvorova iz stabla (a ove operacije su česte u praksi). Ubacivanje u VST se može analogno rešiti kao kod BST. Međutim, to može dovesti do nebalnsiranosti VST. Zato se koristi jedna druga metoda koja se koristi za specijalan slučaj VST tzv. B-stabla, a koja dovodi i do efikasnijeg korišćenja prostora u čvorovima.

B-stabla

Balansirano višegransko stablo traženja reda **n+1** u kome svaki čvor različit od korena sadrži najmanje **n div 2** ključeva se naziva **B-stablo** reda n, Razmotrimo sada algoritme ubacivanja i izbacivanja iz B-stabla,

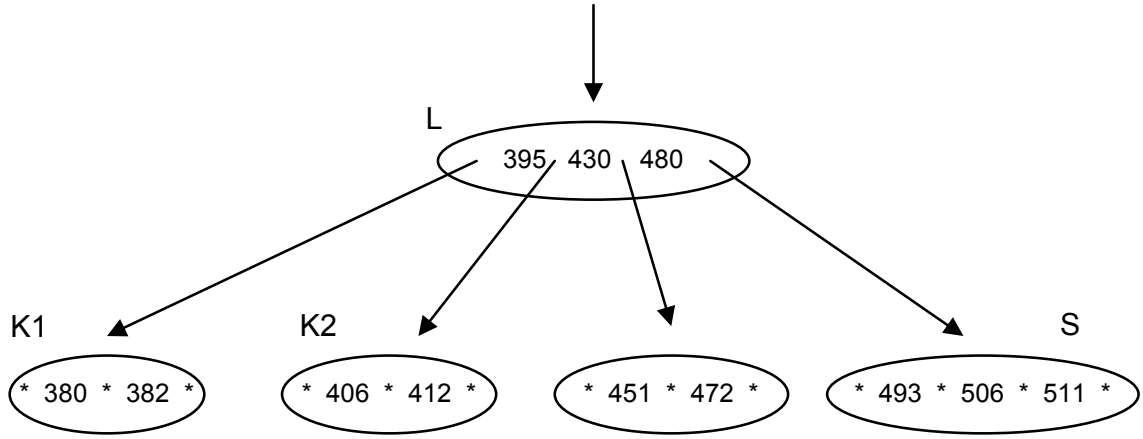
Ubacivanje u B-stablo

Prvo treba da pronađemo list u koji treba ubaciti novi ključ. Pronalaženje lista je kao pretraživanje ključa kada bi se on nalazio u stablu i ono se vrši na isti način kao i kod VST. Zatim, ako u listu ima mesta, ključ se ubaci, Međutim ako nema mesta tad se list pocepa u dva Lista: levi i desni. N ključeva (kojih čine N-1 ključeva iz lista i ključ koji se ubacuje) sortiranih u rastućem redosledu se podele u dve grupe. Prva n div 2 polovina se smešta u levi novokreirani list, a druga n div 2 polovina u desni list. Srednji ključ (pretpostavimo da je N neparno) iz pomenutog sortiranog niza ključeva se diže na viši nivo i smešta u naredni čvor-otac (ako ima mesta). Njegov levi i desni pokazivač pokazuju na novokreirani levi i desni list, respektivno. Pogledajmo to na jednom primeru. Posmatrajmo B-stablo:



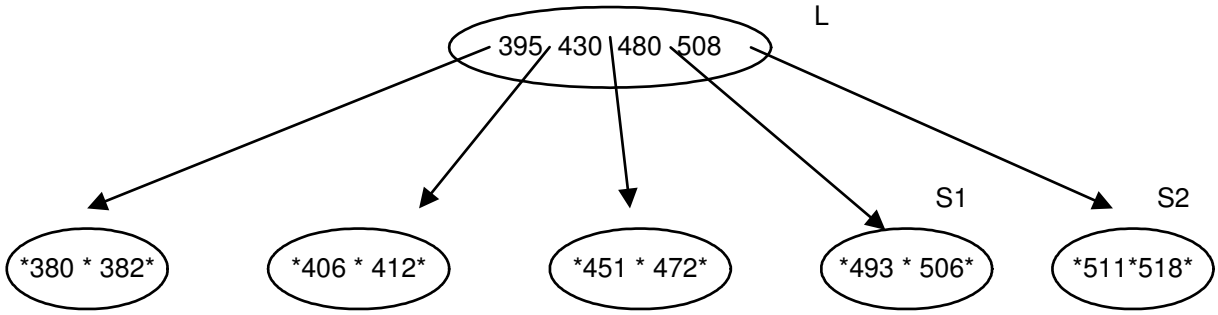
a)Inicijalno B-stablo

Pokušajmo da ubacimo ključ 382. On. treba da se ubaci u list K. Međutim, čvor K je pun. Zato se on cepa u dva čvora K1 i K2. Niz ključeva 380, 382, 395, 406, 412 se podeli na dva dela tako da 380. 382 ide u K1, 406, 412 ide u K2, a srednji ključ 395 se penje na viši nivo u čvor L .



b) Posle ubacivanja 382

Ubacimo ključ 518 u prethodno stablo. On se ubacuje u čvor S. Zatim. ubacimo 508. On takođe treba da se ubaci u čvor S, ali sada nema mesta (jer je S pun) tako da se čvor S cepa u čvorove Sl i S2 i dobijamo sledeće B-stablo:

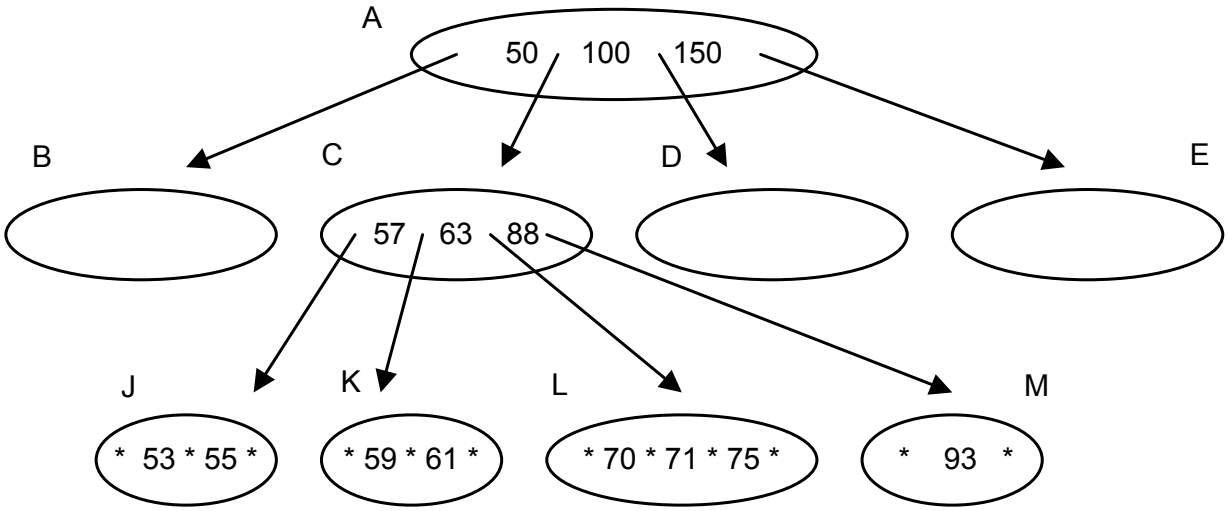


c) Posle ubacivanja 518 i 508;

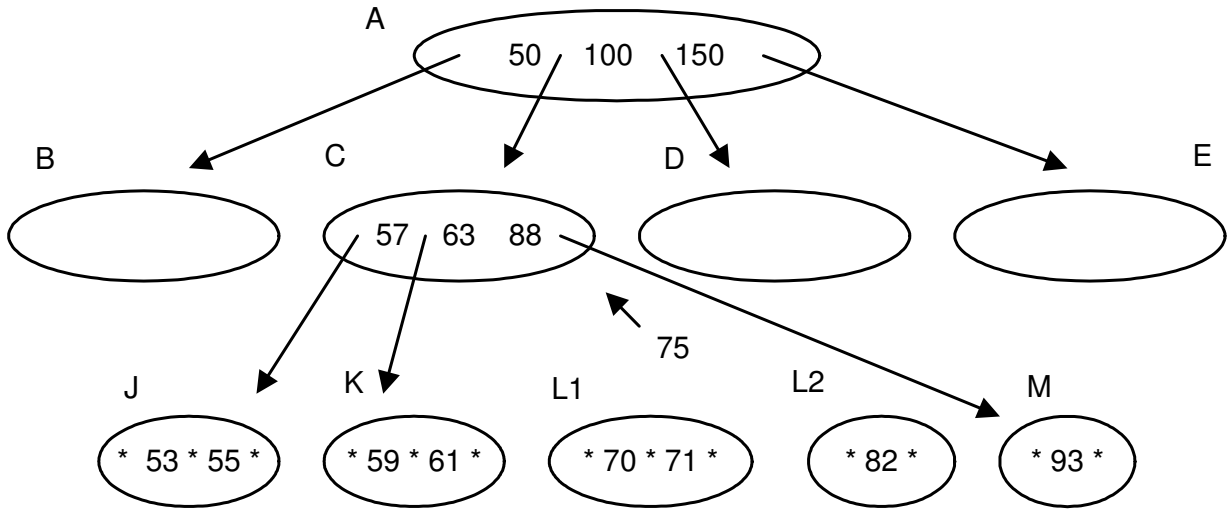
Kada je red B-stabla n parno, tada se niz ključeva iz čvora koji se cepa ne može podeliti u dva jednaka dela. U tom slučaju se u jedan novokreirani čvor (bilo levi, bilo desni) stavi veći deo niza ključeva.

Kao što smo videli iz prethodnog primera, balansiranost B-stabla se održava i posle ubacivanja. Postavlja se pitanje šta ako ključ koji u slučaju cepanja lista se diže na viši nivo treba ubaciti u čvor-otac koji je pun. Pa jednostavno, sada se čvor-otac cepa u dva čvora, a srednji ključ iz niza ključeva (koji se formira kada se na N ključeva oca doda ključ sa nižeg nivoa) se diže na viši nivo i ubacuje u nadređeni čvor. Ako na višem nivou nema mesta, postupak se (rekurzivno) ponavlja. Tako se može doći do korena B-stabla. U tom slučaju koren se cepa u dva čvora *i* pošto nema višeg nivoa formira se novi čvor koji postaje koren B-stabla i u njega se ubacuje srednji ključ sa prethodnog nivoa (zajedno sa pokazivačima na svoja dva podstabla). Tada će se visina B-stabla povećati za jedan.

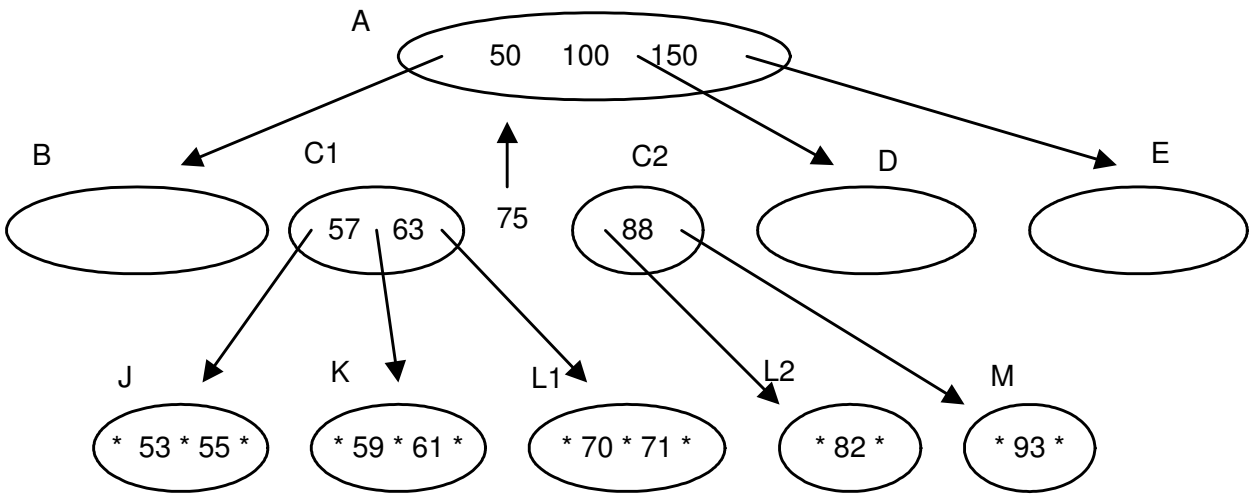
Pretpostavimo da imamo sledeće B-stablo:



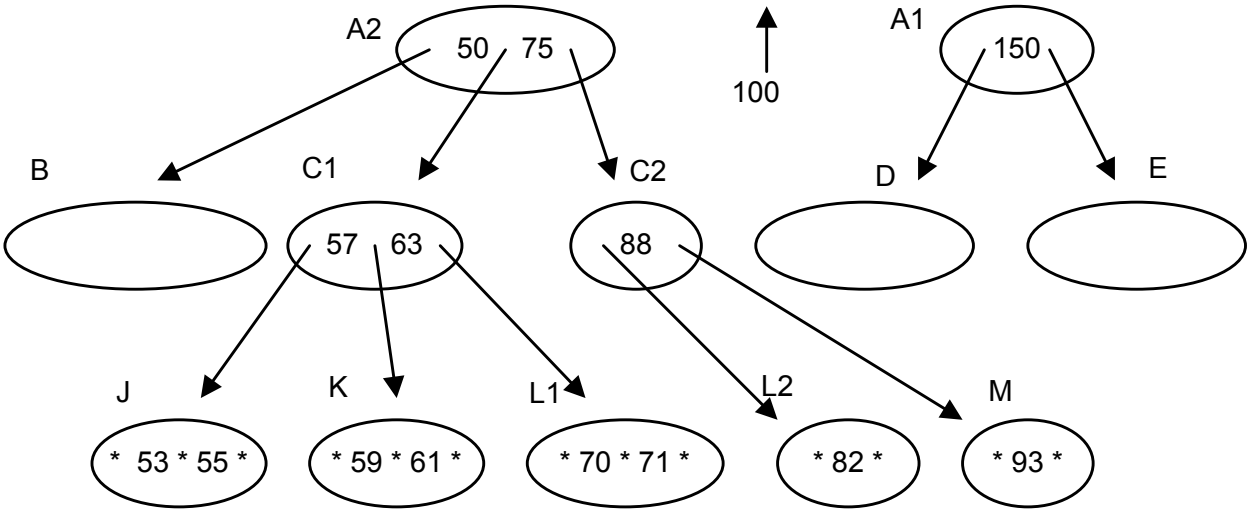
Pretpostavimo da želimo ubacivanje ključa *82*. Njega treba ubaciti u čvor L, ali nema mesta. Prema tome, čvor L se cepa u L1 i L2, a srednji ključ 75 treba da se ubaci u čvor C .



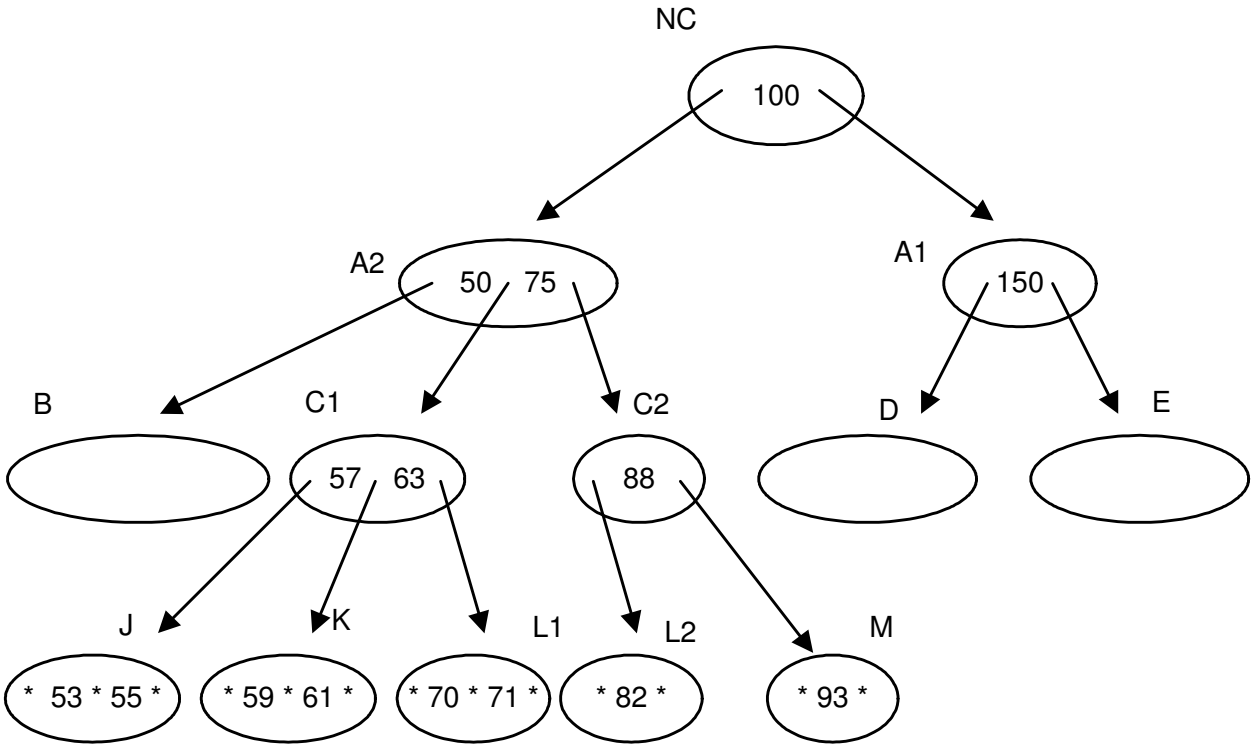
Međutim. ni u čvoru C nema mesta. pa se i on cepa u čvorove C1 i C2 a srednji ključ 75 se diže na visi nivo:



Pošto u čvoru A nema mesta, algoritam se ponavlja i izvrši se njegovo cepanje u čvorove A1 i A2, a srednji ključ 100 treba da se digne na viši nivo :



Kako je čvor a bio koren tj. nema višeg nivoa, kreira se novi čvor NČ koji postaje koren stabla i u koga se ubacuje ključ 100 sa nižeg nivoa.



Primetite da B-stablo raste u visinu kada se cepa koren, a u širinu kada se cepaju ostali čvorovi.

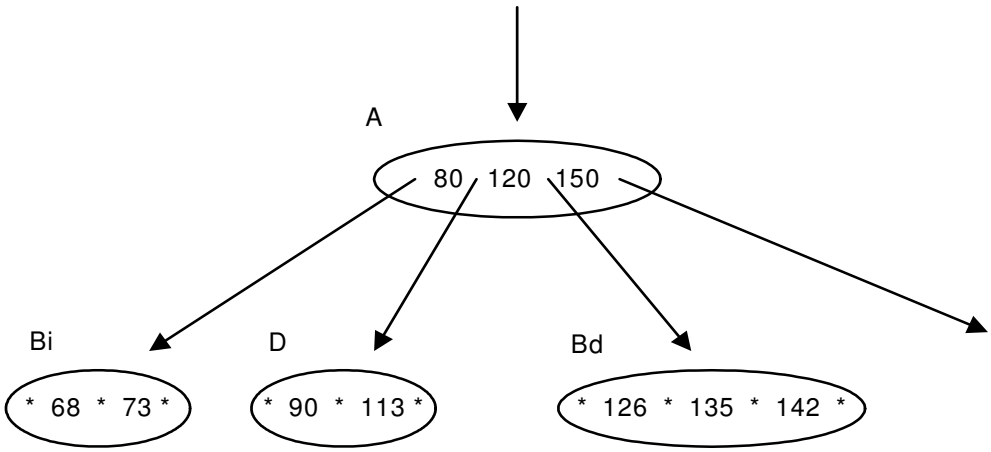
Izbacivanje iz B-stabla

Prilikom izbacivanja ključa iz B-stabla treba očuvati uslove integriteta:

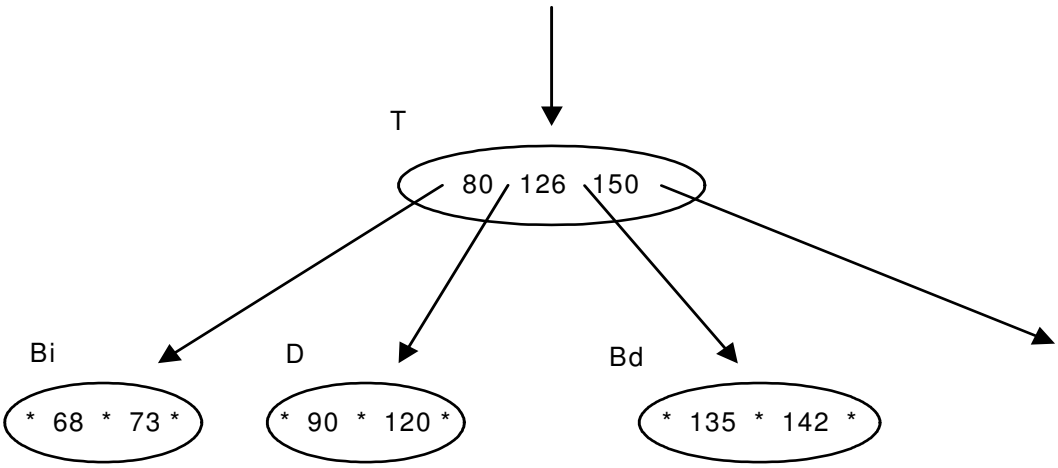
- da stablo bude **balansirano**
- i da u svakom čvoru bude **najmanje n div 2** ključeva

Razmotrimo dva slučaja izbacivanja: kada se ključ koji se izbacuje nalazi u čvoru koji nije list i kada je taj čvor list. Kada se izbacuje ključ iz čvora koji nije list, njegov sledbenik koji mora biti u listu (najveći levi ili najmanji desni) se prebacuje na njegovo mesto, pa se problem svodi na izbacivanje ključa iz lista. Stoga, dalje ćemo razmatrati samo izbacivanje iz lista.

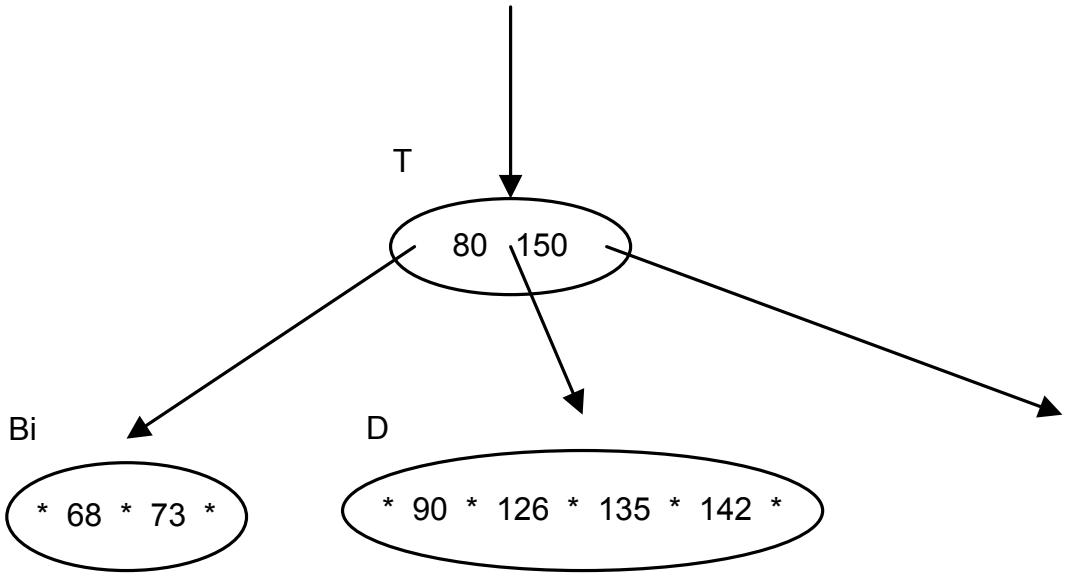
Ukoliko u listu ima više od **n div** 2 ključeva, onda se ključ jednostavno izbaci. Međutim, može se desiti da ima tačno **n div** 2 ključeva, pa bi se izbacivanjem ključa narušio prethodni, drugi, uslov integriteta B-stabla. Takva situacija se naziva podkoračenje U toj situaciji najjednostavnije rešenje je ispitati levog ili desnog brata. Ako brat sadrži više od **n div** 2 ključeva tada se ključ KS iz čvora oca koji razdvaja dva lista brata spusti u čvor iz koga se prethodno izbaci željeni ključ, a ključ brata (prvi ili poslednji što zavisi da li je desni ili levi brat, respektivno) se prebacuje na mesto ključa KS, Na primer, neka iz B-stabla reda 4 prikazan na slici :



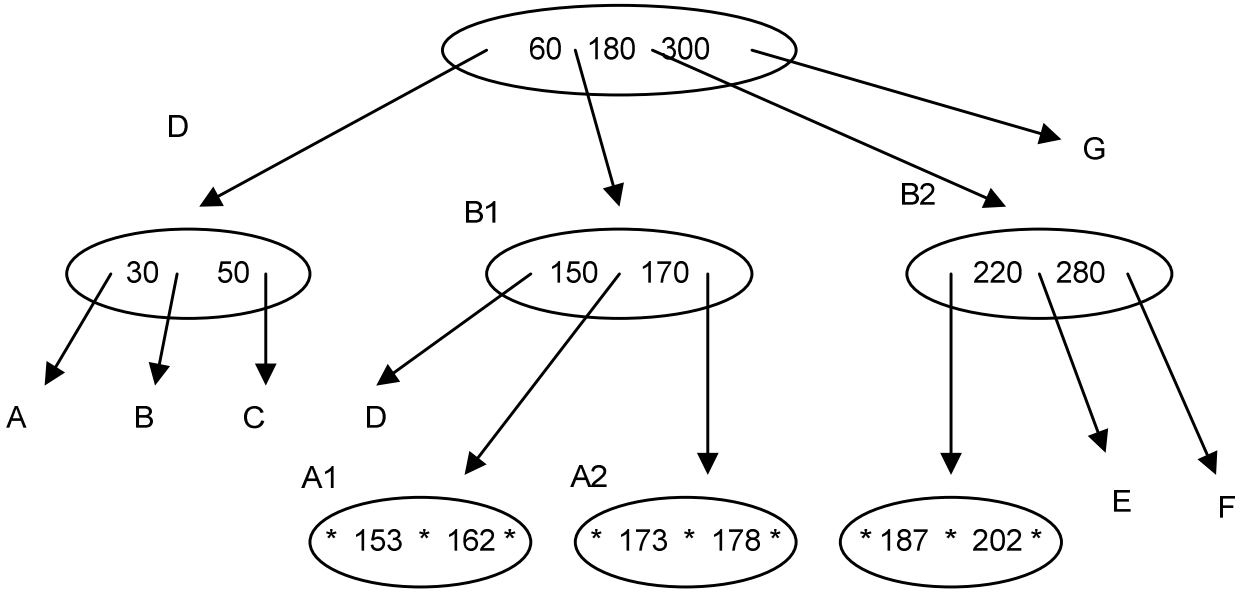
želimo da izbacimo Ključ 113. On treba da se izbaci iz ključa D. Međutim tada bi nastupilo potkoračenje i zato, pošto desni brat Bd ima vise od 2 ključa u čvor D se spušta ključ 120 iz A, a na njegovo mesto dolazi 126 iz Bd.



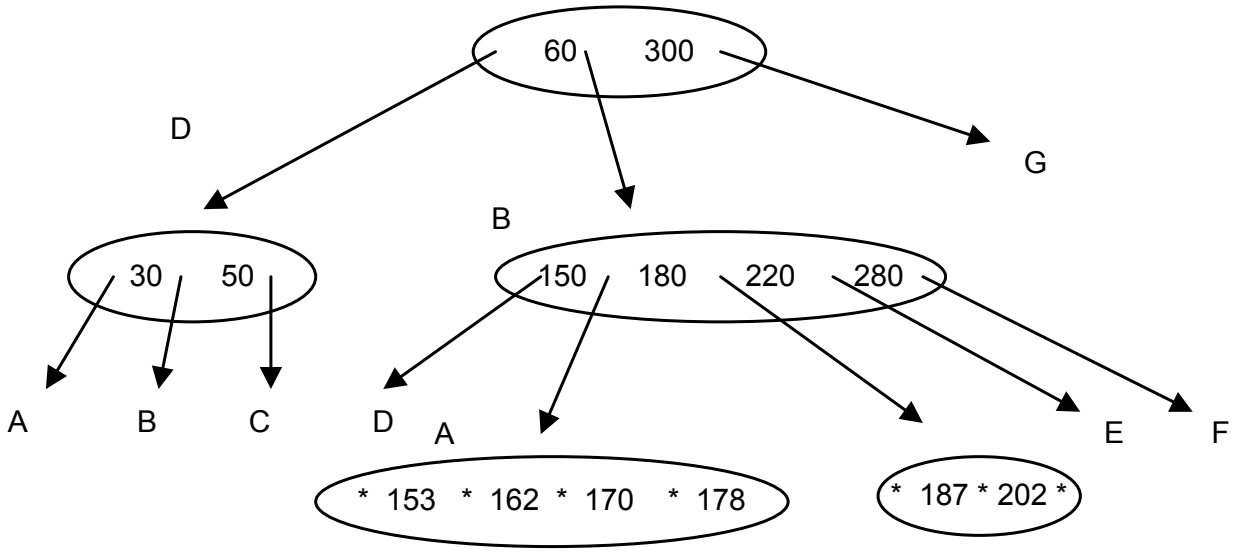
Međutim, može se desiti da i u levom i u desnom bratu ima po tačno n **div** 2 ključeva, pa se ne može uzeti ključ iz brata. U tom slučaju, čvor iz koga se izbacuje i jedan od njegove braće se spajaju u jedan čvor koji sadrži pored njihovih ključeva i ključ spušten sa višeg nivoa, od oca. Na primer da iz prethodnog stabla izbacimo 120. To zahteva spajanje čvorova D i Bd kao i spuštanje ključa 126 u novi čvor nastao spajanjem što se vidi na sledećoj slici:



Ali. prilikom spuštanja ključa sa višeg nivoa, može se desiti podkoračenje u čvoru iz koga se ključ uzima. U tom slučaju, postupak se ponavlja. Taj čvor ce od svog brata pozajmiti ključ, ako on ima "slobodnih" ključeva. Ako nema, onda se on spaja sa svojim bratom a uzima se ključ od njihovog oca. Ako ni otac nema "viška" ključeva onda se on spaja sa svojim bratom, ako može i ceo postupak se ponavlja rekurzivno. Na primer, izbacivanjem ključa 173 iz stabla :



i posle spajanja čvorova A1 i A2 u A i B1 i B2 u B dobija se stablo:



U najgorem slučaju, primenom ovog postupka izbacivanja će se doći do korena stabla iz koga treba spustiti ključ na niži nivo a koren ima samo jedan ključ. Tada, čvor na nižem nivou, nastao spajanjem, postaje novi koren stabla (stari koren se uništava). Na taj način, B-stablo će smanjiti visinu za jedan.

Znači, B-stablo je jedna dinamička struktura podataka u kojoj visina varira. Ubacivanjem se visina može povećati a izbacivanjem smanjiti.

Implementacija i efikasnost B-stabala

B-stabla se najčešće koriste za smeštanje podataka na spoljne memorije sa direktnim pristupom kao što je disk. Čvor B-stabla se obično implementira kao stranica (blok) diska. Na taj način. jednim pristupom disku se čita čvor B-stabla. Kako je vreme pozicioniranja glave diska na određenu stranicu (zapravo stazu u kojoj je stranica) mnogo veće od vremena čitanja stranice, veličina stranice ne igra bitnu ulogu u vremenu pristupa. Zato se veličina stranice. tj. red B-stabla maksimizira, jer je cilj jednim pristupom učitati što više zapisa. Naravno, veličina stranice je limitirana veličinom bafera, tj. veličinom unutrašnje memorije. U komercijalnim implementacijama nije ništa neobično da red B-stabla bude nekoliko stotina.

Drugi faktor koji treba uzeti u obzir je zauzeće prostora. Kako svaki čvor mora imati najmanje **n div 2** ključeva, najmanja popunjenost stranice iznosi 50%. U praksi, prosečno iskorišćenje prostora iznosi približno 69%.

U prethodnim primerima smo prikazivali samo ključeve u čvorovima stabla. Međutim. uz ključeve idu i podaci. ostali deo zapisa. Oni mogu biti pridruženi ključu u čvoru ili ključu može biti produžen pokazivač gde se oni nalaze, U ovom drugom slučaju je potreban još jedan dodatni pristup ali u stranicu može da stane više ključeva, tj *veći* je red stabla. Kao što ćemo u sledećem delu videti, što je red stabla *veći* to je efikasnost veća. tj. potreban je manji broj pristupa. U prvom slučaju. pošto podaci obično zauzimaju više prostora od pokazivača. stane manje ključeva pa je red stabla manji ali nije potreban dodatni pristup.

Razmotrimo sada efikasnost B-stabla. Neka je red stabla **m-1** a **n** ukupan broj ključeva u stablu. Svaki čvor sadrži najmanje **q = (m-1) div 2** ključeva. Sledeća tabela ilustruje minimalni i maksimalni broj čvorova *i* ključeva u stablu na nivou 1,2,3 i proizvoljnom nivoi i kao i u celom stablu :

Visina B-stabla	Minimum		Maksimum	
	čvorova	ključeva	čvorova	ključeva
1	1	1	1	m-1
2	2	2q	m	(m -1)m
3	2(q+1)	2q(q+l)	m^2	(m -l)m^2
• • •	• • •	• • •	• • •	*. • •
i	$2(q+1)^{i-2}$	$2q(q+1)^{i-2}$	m^{i-1}	(m -i)m^{i-1}
• • •	• • •	• • •	• • •	• • •
ukupno za visinu d	$\frac{2(q+1)^{d-1}-1}{q}+1$	$2(q+1)^{d-1}$	$\frac{m^d-1}{m-1}$	m^d-1

Iz minimalnog broja ključeva pronalazimo da je maksimalna visina (d) B-stabla:

$n = 2(q+1)^{d-1}$.

Logaritmujući levu i desnu stranu dobijamo:

$\log_{q+1} n = \log_{q+1} 2(q+1)^{d-1}$.

Sređivanjem dobijamo:

$\log_{q+1} n - \log_{q+1} 2 = (d-1) \log_{q+1}(q+1)$,

a iz toga sledi

$d = \log_{q+1}(n/2) + 1$

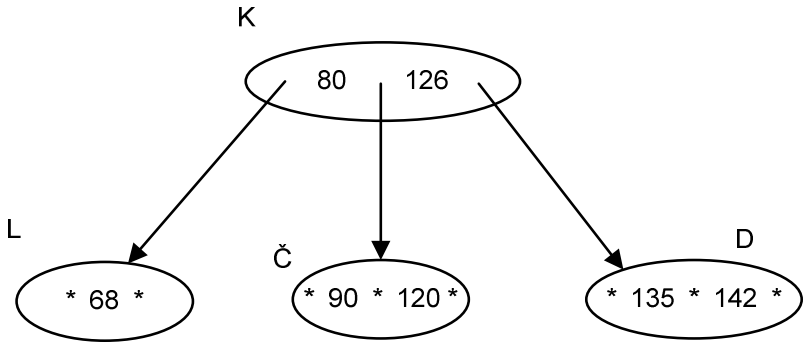
Znači, maksimalni broj pristupa čvorovima B-stabla prilikom pretraživanja (jednak visini stabla) iznosi $\log_{q+1}(n/2) + 1$ gde je n broj ključeva, a q minimalni broj ključeva u čvoru stabla. Tako na primer, za $\mathbf{n = 10^6}$ ključeva i **q = 100**, maksimalni broj pristupa iznosi 4 (?!!!).

Prilikom ubacivanja ili izbacivanja broj pristupa se povećava u slučaju cepanja (odnosno spajanja) čvorova zbog pristupa braći ili precima što u slučaju implementacije na disku može biti značajan gubitak vremena. Broj pristupa se može smanjiti ako se u unutrašnjoj memoriji zapamte stranice kojima se pristupa njih nema mnogo) tako da se za ponovno obraćanje njima ne mora pristupati disku.

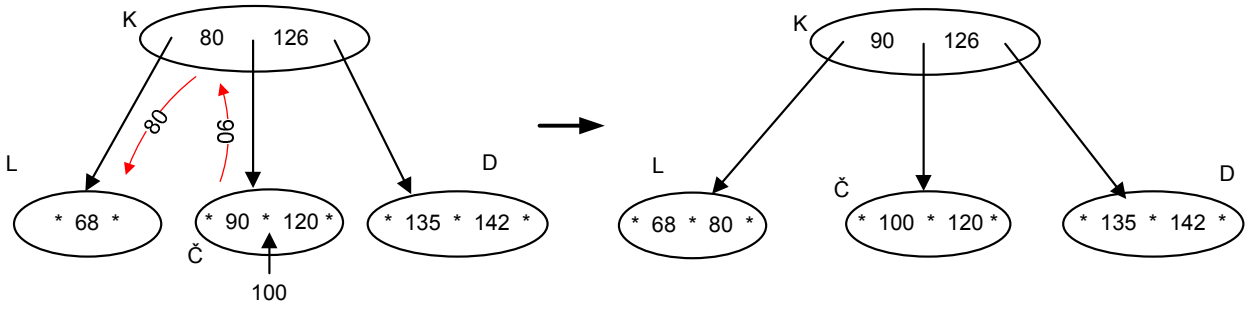
B* stabla

Jedan od načina za poboljšavanje efikasnosti B-stabla je u poboljšanju algoritma ubacivanja. Kada je potrebno ubaciti ključ u čvor koji je pun, umesto da se čvor pocepa kako je to bilo po prethodnom algoritmu ubacivanja, prvo se pogleda da li kod levog ili desnog brata ima mesta za ubacivanje ključeva. Ako ima, onda se ključevi raspodele između čvora i njegovog brata uključujući i odgovarajući ključ iz oca. Tako se cepanje odlaže sve dok se ne desi da niko od braće nema mesta. U tom slučaju, cepanje čvora je neminovno. Na taj način, algoritam se malo komplikuje, ali se postiže efikasniji pristup, a i prostor se još efikasnije koristi. Stablo sa ovakvim algoritmom ubacivanja se naziva B*-stablo (be zvezda stablo).

Neka je dato sledeće B* stablo:



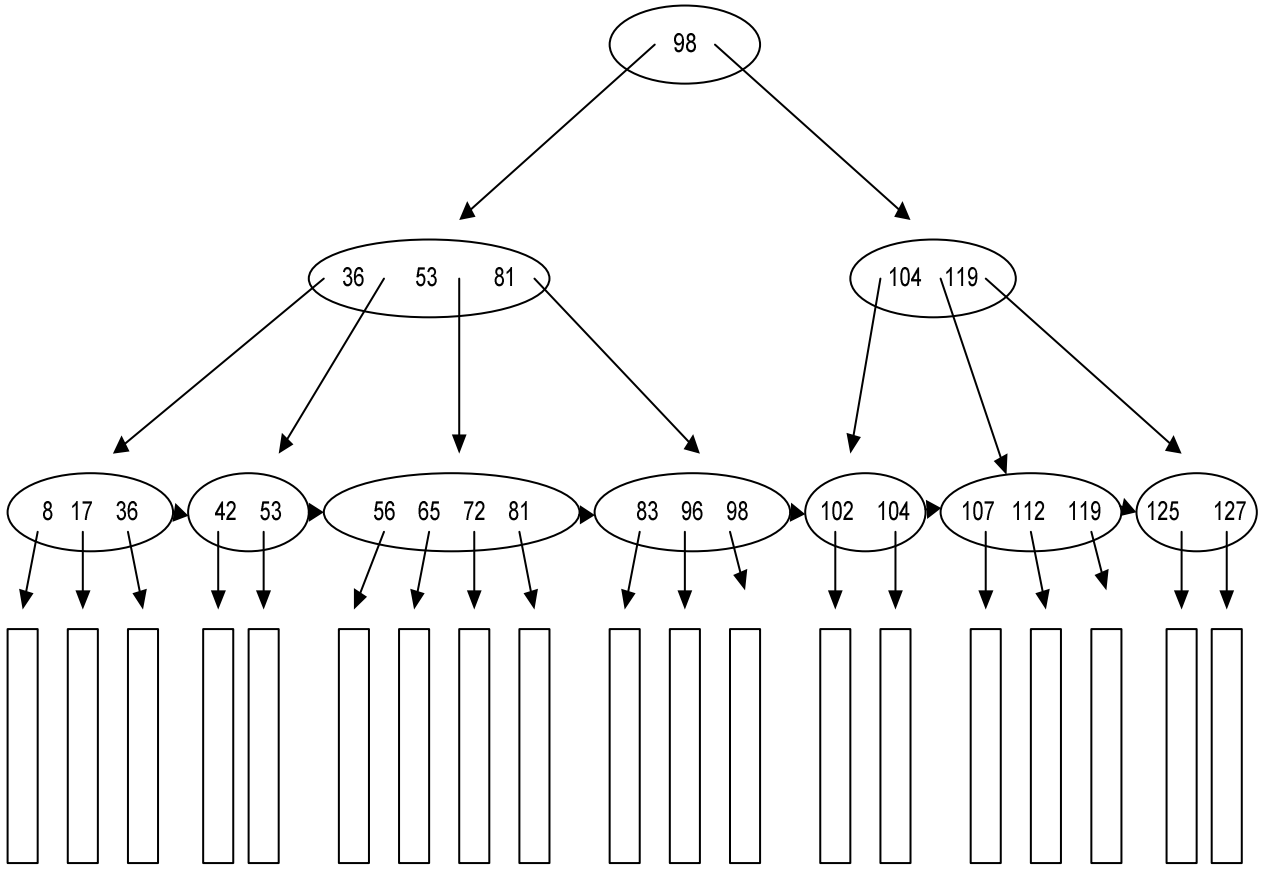
Recimo da se u dato stablo koje je reda 2 (maksimalni broj elemenata/ključeva u čvoru je 2) ubacuje ključ **100**. Prema pravilima ubacivanja on bi trebalo da se ubaci u čvor ***Č***. Međutim, pošto u tom čvoru nema mesta (ima maksimalan broj ključeva), onda se proverava da li su levi ili desni brat popunjeni, odnosno, da li neki od njih ima mesta da primi jedan element. Pošto levi brat (čvor ***L***) ima mesta, onda se vrši pozajmljivanje na sledeći način: zajednički roditeljski element za dva čvora kod kojih se vrši pozajmljivanje (element **80** u čvoru ***K***) se spušta u brata koji ima dovoljno mesta (čvor ***L***), a na njegovo mesto dolazi najmanji (ukoliko se pozajmljuje levom bratu) ili najveći element (ako se pozajmljuje desnom bratu) iz čvora u koji se vrši inicijalno ubacivanje. U našem slučaju element **80** se iz čvora ***K*** spušta u čvor ***L***, a na njegovo mesto dolazi element **90** iz čvora ***Č***. Ovo je prikazano na sledećoj slici.



Ako su i levi i desni brat popunjeni vrši se cepanje na način kao kod običnog B-stabla.
Izbacivanje se takođe vrši na isti način kao i kod B-stabla.

B+ stabla

Jedan od glavnih nedostataka B-stabla je nepodesnost strukture za sekvencijalni pristup ključevima što je u nekim situacijama potrebno. B^+-stablo omogućava sekvencijalni pristup ključevima, ali zadržava i mogućnost direktnog pristupa. U B^+-stablu svi ključevi se nalaze u listovima stabla, a neki se nalaze i u ostalim čvorovima tako da omogućavaju pronalaženje pozicije zapisa kod direktnog pristupa. Svi listovi stabla su označeni što omogućuje sekvencijalni pristup zapisima. Ključevima u listu su pridruženi pokazivači na zapise sa podacima. Sledeća slika ilustruje jedno **B^+-**stablo:



+
B-stablo

Pretraživanje je slično kao kod B-stabla samo što kad se pronađe ključ se pridruži do lista stabla gde se nalazi isti ključ, ali sa **pokazivačem** na zapis.

B^+-stablo se može smatrati proširenje indeks-sekvencijalne organizacije u kome su indeksi dinamički tipa B-stablo. Svaki nivo u stablu je zapravo indeks sledećeg nivoa.

B^+-stablo zadržava istu efikasnost traženja i ubacivanja B-stabla, ali povećava efikasnost kod pronalaženja sledećeg, susednog ključa u odnosu na nađeni (GET NEXT operacija). Potreban je, u najgorem slučaju, jedan pristup desnom bratu-listu. Kod B-stabla prethodna operacija je zahtevala penjanje i spuštanje kroz stablo što je neefikasno.

Zbog svih ovih, napred navedenih osobina, B* i B^+ stabla su vrlo popularna za implementaciju baza podataka u savremenim sistemima za upravljanje bazom podataka (naročito relacionog tipa).

Pretraživanje transformacijom ključa u adresu - Hashing

Ovom metodom se uspostavlja direktna veza između vrednosti ključa i adrese gde se taj ključ (tj. zapis) nalazi.

Najjednostavniji način je da vrednost ključa bude adresa zapisa. Međutim, to nije praktično rešenje (zašto?). Zato se vrši transformacija nad vrednošću ključa da se dobije adresa zapisa. Ovu transformaciju možemo tretirati i kao funkciju čiji argument je vrednost ključa k, a vrednost A=h(k). Ovakva funkcija se naziva hash funkcija pa otuda i naziv za transformaciju hashing.

Od funkcije transformacije se zahteva da sve ključeve preslikava na adresni prostor predviđen za smeštanje zapisa tj. na ograničen opseg celobrojnih vrednosti. Takođe, funkcija transformacije ne bi trebala nikada da dva različita ključa preslika na istu adresu. Slučaj kada dva različita ključa dobiju istu adresu se naziva **kolizija** ili grupisanje ključeva.

Dakle, da bismo primenili ovu metodu pretraživanja, potrebno je rešiti sledeća dva problema:

1. Pronaći funkciju transformacije h(k) koja ima vrednosti uniformno raspodeljene na intervalu 0 do M-l tj. da ne dovodi do kolizije ključeva:
2. Naći metod rešavanja kolizije ključeva.

U praksi se pokazalo da je gotovo nemoguće naći funkciju transformacije koja neće dovoditi do kolizije. Najčešće korišćena funkcija transformacije je **metod ostataka od deljenja** kod koje se vrednost ključa celobrojno deli sa veličinom M i ostatak od deljenja se koristi kao adresa tj. **h(k) = k mod M**. Vrednosti se nalaze u intervalu od 0 do M-l. Najbolji rezultati, tj. najmanji broj kolizija, se dobija kada je M prost broj nešto veći od raspoloživog adresnog prostora. Eksperimentima se pokazalo da tada ova funkcija ima najbolje ponašanje u odnosu na ostale funkcije transformacije.

Za funkciju transformacije se još koristi i metoda kvadriranja ključa i uzimanja nekoliko srednjih cifara za vrednost funkcije. Takođe se koristi i metoda deljenja cifara ključa u segmente i izvođenjem aritmetičkih i logičkih operacija nad njima kako bi se dobila vrednost funkcije tj. **adresa.**

U slučaju kolizije ključeva, primenjuju se generalno **dve tehnike**.

Prva tehnika se naziva otvoreno adresiranje. Kada se primenom funkcije h(k) dobije adresa koja je već zauzeta, tada se na dobijenu vrednost primenjuje nova funkcija r(k). Ona se primenjuje sve dok se dobije slobodna adresa. Najjednostavnija funkcija r(k) je $r_{i+1}(k)=r_i(k)+1$, $r_1(k)=h(k)+l$, tj. funkcija čija vrednost je sledeća adresa u odnosu na prethodno izračunatu adresu. Ovom metodom se zapravo sekvencijalno, linearno ispituju adrese sve dok se ne nađe slobodna adresa. Zato se ova metoda naziva **linearno probanje.** Mogući su i drugi oblici funkcije r(k).

Druga tehnika problem kolizije rešava olančavanjem zapisa čiji ključevi imaju istu vrednost za h(k). Za smeštanje olančanih zapisa se može koristiti isti memorijski prostor kao i za ostale zapise bez kolizije tzv. primarni prostor a može se i rezervisati poseban prostor za njih. U prvom slučaju se zauzima manje prostora, tj. efikasnije se koristi prostor ali može dovesti do povećanog broja kolizija. Naime, smeštanjem zapisa čije adrese su zauzete u primarni prostor se zapravo zauzimaju adrese za neke zapise koje bi oni dobili sa funkcijom h(k). Slučaj da jedan zapis primenom funkcije h(k) dobije adresu koja je zauzeta od strane zapisa čija vrednost h(k) nije ta adresa se naziva **sekundarno** grupisanje.

Primena metode transformacije ključa u adresu je sledeća. Prvo se svi zapisi smeste

korišćenjem funkcije h(k). Slučajevi kolizije se rešavaju primenom neke od pomenutih tehnika. Pretraživanje se vrši na sledeći način. Primenom funkcije h(k) na argument traženja se dobija adresa za koju se proveri da li se na njoj nalazi traženi zapis. Ako se ne nalazi, tada se izabrana tehnika rešavanja kolizije primenjuje na prethodno dobijenu adresu kao i kod smeštanja. Ona se primenjuje sve dok se ne nađe traženi zapis ili se ustanovi da ga nema. Znači, u slučaju otvorenog adresiranja, funkcija r(k) se primenjuje sve dok se nađe zapis, dok u slučaju olančavanja se pretraži lista zapisa koji imaju istu vrednost za h(k).