PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET METROPOLITAN BEOGRAD

KI202 - JAVA 5: STRUKTURE PODATAKA I ALGORITMI – DEO B

Binarna stabla pretraživanja

Lekcija 02

PRIRUČNIK ZA STUDENTE

KI202 - JAVA 5: STRUKTURE PODATAKA I ALGORITMI – DEO B

Lekcija 02

BINARNA STABLA PRETRAŽIVANJA

- Binarna stabla pretraživanja
- Poglavlje 1: Osnovi o stablima
- Poglavlje 2: Binarna stabla pretraživanja (BST)
- Poglavlje 3: Rekurzija i stabla
- Poglavlje 4: Obilazak binarnog stabla
- Poglavlje 5: Implementacija BST stabla u Javi
- Poglavlje 6: Brisanje čvora iz BST stabla
- Poglavlje 7: Vizuelizacija stabla i MVC
- Poglavlje 8: Iteratori za BST stabla
- Poglavlje 9: Studija slučaja: Kompresija podataka
- Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

Uvod

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

Stablo (tree) obezbeđuje hijerarhijsku organizaciju u kojoj se podaci nalaze u čvorovima. U okviru ove lekcije studenti se upoznaju sa:

- Osnovnom terminologijom stabala kao struktura podataka
- Binarnim stablima i binarnim stablima pretraživanja (BST – binary search tree)
- Implementacijom osnovnih funkcija za rad sa BST stablima (kreiranje stabla i pretraživanje elemenata)
- Upotrebom rekurzije kod implementacije funkcije za rad sa binarnim stablima
- Implementacijom algoritama za obilazak stabla po dubini (DFS) i širini (BFS)
- Postupkom brisanja elemenata iz BST stabla
- Implementacijom klase za BST stablo i klase za iteratore kroz BST stablo u Javi
- Vizuelizacijom binarsnih stabala pretraživanja
- Implementacijom stabla izraza i Hafmanovog stabla kodiranja.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 1

Osnovi o stablima

UVOD U STABLA

Stablo je rekurzivna aciklična struktura podataka koja se sastoji od čvorova koji su povezani granama. Svaki čvor ima najviše jednog roditelja, i nula ili više direktnih potomaka (dece)

Stablo je aciklični povezan graf koji ima sledeće karakteristike:

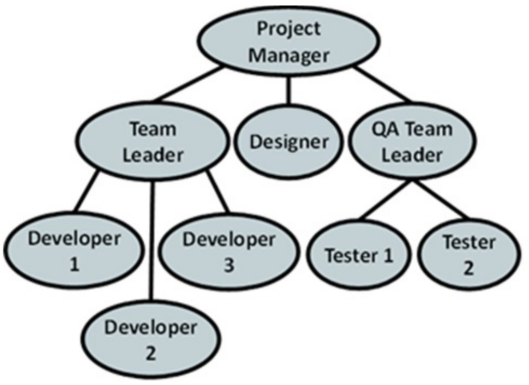
- stablo je rekurzivna struktura podataka koja se sastoji od čvorova koji su povezani granama
- svaki čvor može imati 0 ili više direktnih potomaka (dece)
- svaki čvor ima najviše jednog roditelja
- postoji samo jedan poseban čvor bez roditelja - koren (ako stablo nije prazno)
- svi čvorovi su dostupni iz korena - postoji put od korena do svakog čvora u stablu.

Ako je grana usmerena ka čvoru, onda je to dolazna (engl. indegree) grana. Ako je grana usmerena od čvora, onda je ona odlazna (engl. outdegree) grana.

Ovde outdegree znači broj odlazećih usmerenih ivica od datog čvora u usmerenom dijagramu. Zbir ulaznih i izlaznih grana je jednak stepenu čvora. Ako stablo nije prazno, onda se prvi čvor zove koren. Broj ulaznih grana korena je naravno nula. Svi čvorovi stabla (izuzev korena) moraju imati broj ulaznih grana (indegree) od tačno jedan.

Primer: Hijerahija učesnika na projektu

Dat je tim odgovoran za razvoj nekog softverskog projekta. Između učesnika postoje relacije menadžer-potčinjeni. Tim ima 9 članova, Slika-1.



Slika 1.1 Hijerahija učesnika na projektu

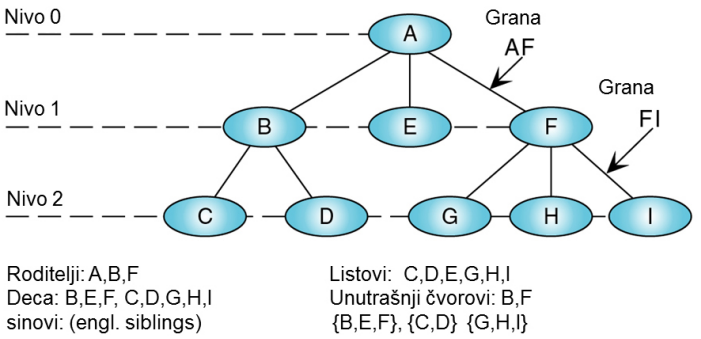
Šta se može dobiti od ove hijerarhije?

Šef developera je "Team Leader", vođa projekta, ali su developeri indirektno podređeni menadžeru projekta ("Project Manager"). Vođa tima je podređen samo menadžeru projekta. S druge strane, "Developer1"nema podređene. Menadžer projekta je najviši u hijerarhiji i nema svog menadžera. Na isti način se mogu opisati i ostali učesnici na projektu.

TERMINOLOGIJA STABLA

Osnovni termini kod stabla su: čvor, grana, roditelj, potomak, faktor grananja, koren, list, unutrašnji čvor, putanja, dužina putanje, dubina čvora i visina stabla

Slika-2 ilustruje osnovne pojmove stabla:



Slika 1.2 Osnovni pojmovi stabla

Svaki krug je čvor a svaka linija je grana (engl. edge). Čvorovi "B", "E", "F" su direktni potomci čvora "A", a čvor "A" njihov roditelj“. Intuitivno možemo reći da je "B" brat (engl. sibling) čvora “E", jer su oba deca čvora "A".

Čvor "A" prethodi čvorovima "C", "D" , "G", "H" i "I" u hijerarhiji, tako da je njihov indirektni roditelj – predak i oni se zovu njegovi potomci (engl. descendants).

Koren je čvor koji nema roditelja. U našem primeru to je čvor "A".

List je čvor koji nema potomaka. U našem primeru listovi su " C ", " D ", " E ", " G ", " H " i " I" .

Unutrašnji čvorovi su čvorovi koji nisu ni listovi ni koren (svi čvorovi, koji imaju roditelja i najmanje jedno dete). Takvi čvorovi su " B " i "F ".

Putanja (engl. path) se zove sekvenca čvorova povezanih ivicama, u kojoj ne postoji ponavljanje čvorova. Primer putanje je sekvenca "C" , "D" , "A " i " E" . Sekvenca " C " , " B " i " G " nije put , jer " B " i " G " nisu direktno povezani.

Dužina putanje je broj grana koje povezuju sekvencu čvorova na putanji. Zapravo jednaka je broju čvorova na putanju minus 1. Dužina u datom primeru za putanju ("C", "B", "A" i "E") je 3.

Dubina (engl. depth) čvora je dužina putanje staze od korena do određenog čvora. U našem primeru "A" kao koren ima dubinu nula, "B" ima dubinu jedan i "C" – dubina dva.

Visina stabla je maksimalna dubina svih njegovih čvorova. U našem primeru visina stabla je 2.

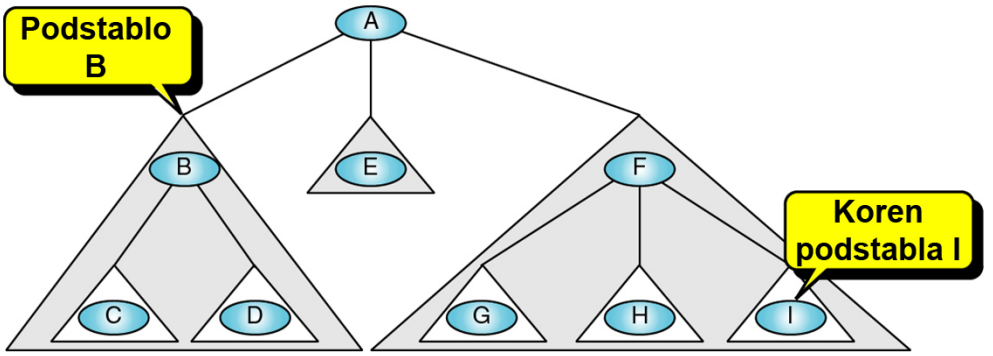
Stepen čvora je broj direktnih potomaka datog čvora. Stepen " A " i " F" je tri , ali stepen "B" je dva. Listovi imaju stepen nula.

Faktor grananja (engl. branching factor) je najveći stepen svih čvorova u stablu. U našem primeru, maksimalni stepen čvorova je 3, tako da je faktor grananja 3.

PODSTABLA

Podstablo je bilo koja povezana struktura ispod korena. Svaki čvor u stablu može da ima 0 ili više svojih podstabala

Podstablo (engl. subtree) je bilo koja povezana struktura ispod korena, Slika-3. Podstablo se takođe može podeliti u nova podstabla.



Slika 1.3 Primer podstabla

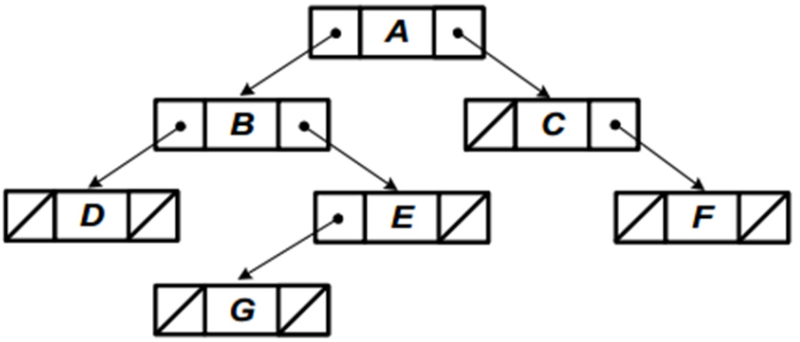
IMPLEMENTACIJA STABLA

Stablo može biti implementirano korišćenjem povezanih struktura podataka (ulančana reprezentacija) i korišćenjem niza. Nizovi su efikasni samo kod potpunih stabala

Stablo može biti implementirano korišćenjem povezanih struktura podataka (ulančana reprezentacija) i korišćenjem niza.

Ulančana reprezentacija stabla

Ova implementacija je fleksibilna za proizvoljne topologije. Najčešće se koristi za implementaciju binarnih stabala. Svaki čvor sadrži informacioni deo i linkove ka svojim potomcima, kao na Slici-4:

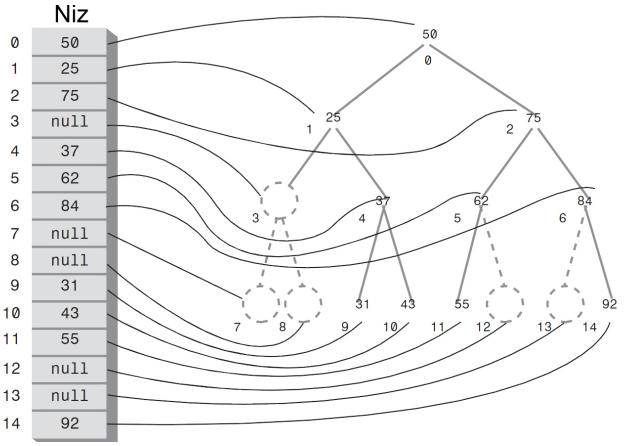


Slika 1.4 Ulančana reprezentacija stabla

Implementacija stabla pomoću niza

Ova metoda implementacija stabla je veoma efikasna za kompletna binarna stabla, Slika-5. O tome smo već govorili kada smo pomenuli *hip stablo* (*heap*) u lekcijama o sortiranju (*heap sort*) i lekciji o implementaciji prioritetnih stabala.

Ova metoda je međutim veoma neefikasna za stabla koja nisu skoro kompletna stabla. Takođe ne predstavlja dinamičnu strukturu podataka.



Slika 1.5 Implemetacija stabla pomoću niza

BINARNA STABLA

Binarno stablo je ono stablo kod koga čvor može imati najviše dva potomka. Ukoliko je implementirano pomoću ulančane strukture, svaki čvor sadrži dva pokazivača ka potomcima

Binarno stablo (engl. binary tree) je skup čvorova koji su povezani na sledeći način:

- svaki čvor može imati levog i desnog potomka (pritom svaki od potomaka može biti i izostavljen. Čak i oba. Dati čvor je roditelj svojih potomaka
- ako je čvor U roditelj čvoru V, tada je U < V .
- čvor koji nema ni levog ni desnog potomka naziva se list (engl. leaf)
- postoji jedan jedinstveni čvor bez roditelja (koren).

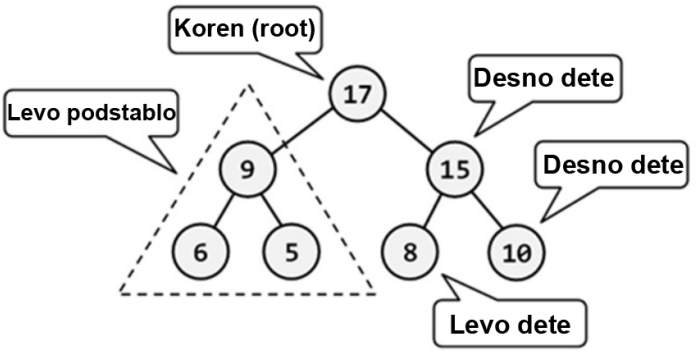
- u stablu nema ciklusa

Struktura binarnog stabla

Ukoliko je stablo implementirano korišćenjem ulančane reprezentacije, svaki čvor u strukturi mora da sadrži podatke koji se čuvaju i dva pokazivača: jedan na levo podstablo i drugi na desno podstablo.

```
Node
 leftSubTree   <pointer to Node>
 data  <dataType>
 rightSubTree <pointer to Node>
End Node
```

Kod binarnog stabla nijedan čvor ne može imati više od dva podstabla, Slika-6.



Slika 1.6 Primer binarnog stabla

Visina, dubina i broj čvorova binarnog stabla

- Maksimalna visina stabla sa N čvorova je data kao $H_{max} = N$.
- Minimalna visina stabla je data kao $H_{min} = [\log_2 N] + 1$
- Ako je poznata visina stabla, tade je $N_{min} = H$ i $N_{max} = 2^H - 1$
- Dubina čvora je broj grana od korena do tog čvora.

Poglavlje 2

Binarna stabla pretraživanja (BST)

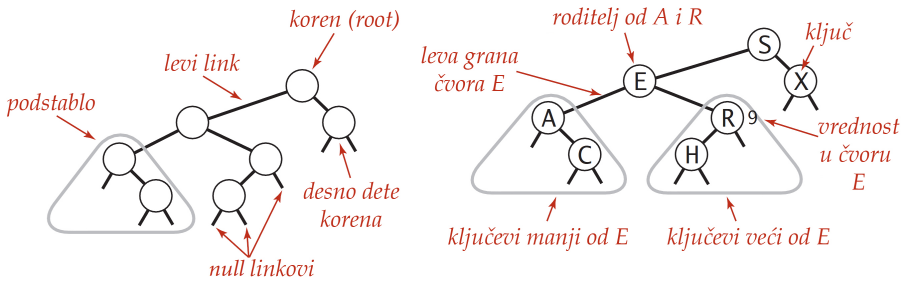
UVODNA RAZMATRANJA

BST zadovoljava ograničenje da je ključ u svakom čvoru veći od ključeva u svim čvorovima u levom podstablu i manji od ključeva u svim čvorovima koju su u desnom podstablu tog čvora

Specijalni tip binarnih stabala se naziva *binarno stablo pretraživanja* (*BST* – binary search tree).

BST zadovoljava ograničenje da je ključ u svakom čvoru veći od ključeva u svim čvorovima u levom podstablu i manji od ključeva u svim čvorovima koju su u desnom podstablu tog čvora.

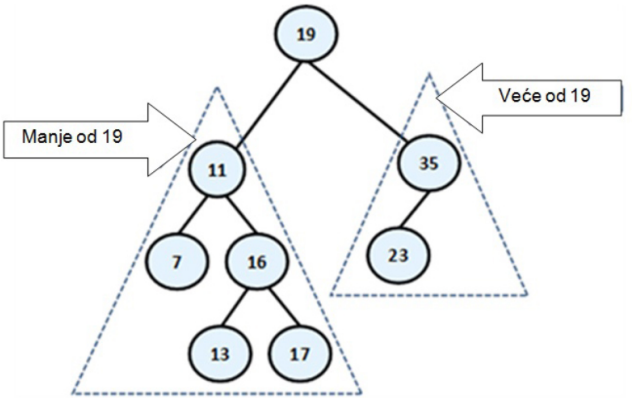
BST omogućava efikasno izvršavanje operacija pretrage, umetanja i uklanjanja elementa. Slika-1 ilustruje razlike između binarnog stabla i binarnog stabla pretraživanja.



Slika 2.1 Razlike između binarnog stabla (BS) i binarnog stabla pretraživanja (BST)

Da bi smo razumeli kako izgleda binarno stablo pretraživanja analiziraćemo primer sa Slike-2.

U korenu stabla je broj 19. U levom podstablu korena se nalaze elementi koji su manji od 19 dok se u njegovom desnom podstablu nalaze elementi koji su veći od 19.

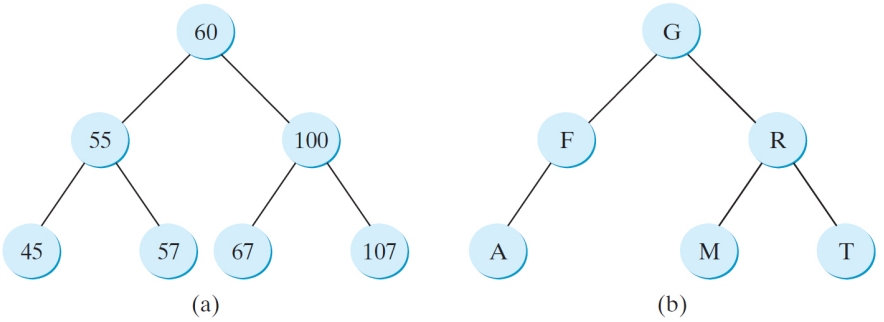


Slika 2.2 Primer binarnog stabla pretraživanja (BST)

PRIMERI BINARNIH STABALA PRETRAŽIVANJA

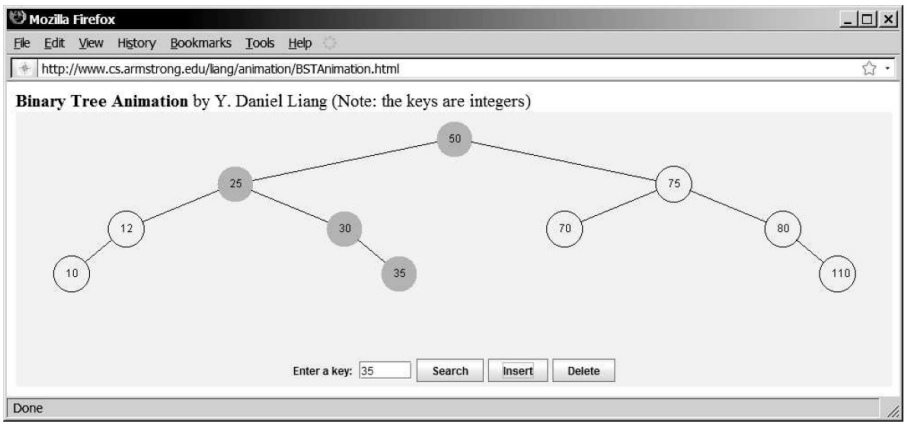
Primeri binarnih stabala pretraživanja su rasprostranjeni svuda oko nas. Najbolji način da se razume kako ono funkcioniše je korišćenje animacije pretrage, dodavanja i brisanja u BST stabl

Binarna stabla prikazana na Slici-3 su oba binarna stabla pretraživanja (BST)



Slika 2.3 Primeri binarnih stabala pretraživanja

Sajt http://www.cs.armstrong.edu/liang/animation/web/BST.html prezentuje animaciju rada binarnog stabla pretraživanja, Slika-4.



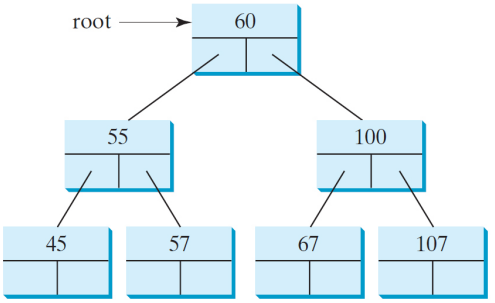
Slika 2.4 Animacija binarnog stabla pretraživanja

PREDSTAVLJANJE BINARNOG STABLA

Svaki čvor binarnog stabla sadrži vrednost (informacioni deo) i dva linka (pokazivača) pod nazivom left (levi) i right (desni) koji referenciraju levog i desnog potomka čvora

Binarno stablo može biti predstavljeno skupom povezanih čvorova. Svaki čvor binarnog stabla sadrži vrednost (informacioni deo) i dva linka (pokazivača) pod nazivom *left* (levi) i *right*

(desni) koji referenciraju levog i desnog potomka čvora, respektivno, kao što je prikazano na Slici-5.



Slika 2.5 Binarno stablo može biti predstavljeno korišćenjem skupa povezanih čvorova.

Čvor binarnog stabla može biti definisan kao klasa, na sledeći način:

```
class TreeNode<E> {
 protected E element;
 protected TreeNode<E> left;
 protected TreeNode<E> right;
 public TreeNode(E e) {
  element = e;
 }
}
```

Promenljiva *root* se odnosi na koren stabla. Ukoliko je stablo prazno, onda je *root* jednako *null*. Naredni deo koda kreira prva tri čvora stabla koje je prikazano na Slici-3.

```
// Create the root node
TreeNode<Integer> root = new TreeNode<Integer>(new Integer(60));

// Create the left child node
root.left = new TreeNode<Integer>(new Integer(55));
// Create the right child node
root.right = new TreeNode<Integer>(new Integer(100));
```

PRETRAŽIVANJE ELEMENTA U STABLU

Neophodno je započeti pretragu od korena, i krenuti naniže ka listovima dok se vrednost nekog čvora ne poklopi sa ključem pretrage ili dok se ne dostigne jedan od listova stabla

Da bi ste pretražili da li se neki element nalazi u binarnom stablu *BST*, neophodno je započeti pretragu od korena, i krenuti naniže ka listovima dok se vrednost nekog čvora ne poklopi sa ključem pretrage ili dok se ne dostigne jedan od listova stabla.

Algoritam pretrage je predstavljen u narednom listingu:

```
BSTpretraga (stablo, ključ)
    IF (prazno_stablo)
        return nije_nadjen;
    IF (ključ == ključ_korena)
        return nadjen;
    IF (ključ > ključ_korena)
        BSPpretraga (levo_podstablo, ključ)
    BSTpretraga (desno_podstablo, ključ)
```

Vremenska složenost prethodnog algoritma je O(h), gde je h visina stabla.

U narednom listingu je dat Java kod koji na sličan način implementira prethodni algoritam.

```
public boolean search(E element) {
 TreeNode<E> current = root; // Start from the root

 while (current != null)
  if (element < current.element) {
   current = current.left; // Go left
  }
  else if (element > current.element) {
   current = current.right; // Go right
  }
  else // Element matches current.element
   return true; // Element is found

 return false; // Element is not in the tree
}
```

Neka u početku *current* pokazuje na koren stabla. Ponoviti sledeće korake sve dok *current* ne postane *null* ili dok se element pretrage ne poklopi sa *current.element*.

- Ako je element manji od *current.element*, nastaviti pretragu u levom podstablu, tj. izjednačiti *current* sa *current.left*.
- Ako je element veći od *current.element*, nastaviti pretragu u desnom podstablu, tj. izjednačiti *current* sa *current.right*.
- Ako je element jednak *current.element*, vratiti vrednost ***true***. Ako je *current* jednako *null*, podstablo je prazno i element se ne nalazi u stablu.

UBACIVANJE ELEMENTA U STABLO

Da bi ste ubacili novi element u binarno stablo neophodno je da locirate poziciju gde ga treba ubaciti. Osnovna ideja je da se locira roditelj novog čvora

Da bi ste ubacili novi element u binarno stablo neophodno je da locirate poziciju gde ga treba ubaciti. Osnovna ideja je da se locira roditelj novog čvora. Algoritam je prikazan u listingu koji sledi:

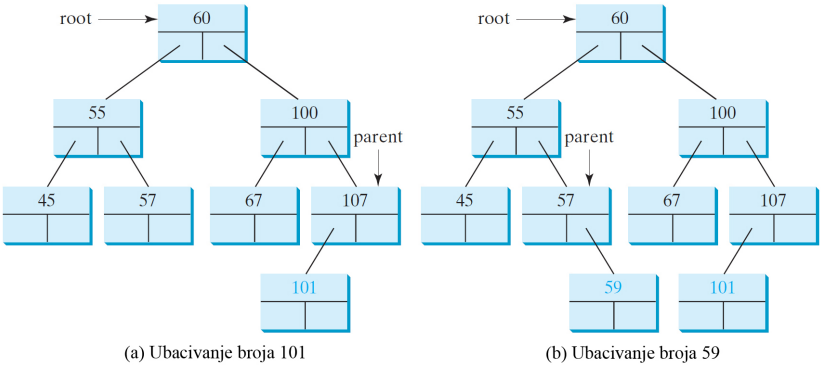
```
boolean insert(E e) {
 if (tree is empty)
  // Create the node for e as the root;
 else {
  // Locate the parent node
  parent = current = root;
  while (current != null)
   if (e < the value in current.element) {
    parent = current; // Keep the parent
    current = current.left; // Go left
   }
   else if (e > the value in current.element) {
    parent = current; // Keep the parent
    current = current.right; // Go right
   }
   else
    return false; // Duplicate node not inserted

  // Create a new node for e and attach it to parent

  return true; // Element inserted
 }
}
```

Ukoliko je stablo prazno kreirati koren koji će primiti vrednost novog elementa. U suprotnom, locirajte roditeljski čvor nakon koga treba ubaciti novi čvor u stablo. Kreirajte novi čvor za element koga ubacujemo, i povežite ga sa roditeljskim čvorom. Ukoliko je novi element manji od roditelja, čvor za novi element se ubacuje u levu granu, odnosno, ako je element veći od roditelja onda se novi čvor ubacuje sa desne strane.

PRIMER UBACIVANJA ELEMENTA U STABLO

Ukoliko je novi element manji od roditelja čvor za novi element se ubacuje u levu granu, odnosno, ako je element veći od roditelja onda se novi čvor ubacuje sa desne strane

Da bi, na primer ubacili broj *101* u stablo sa Slike-5, nakon što se završi ***while*** petlja prethodnog algoritma, pokazivač *parent* pokazuje na čvor *107*, kao što je prikazano na Slici-6a. Novi čvor za *101* postaje levi potomak roditelja. Da bi ste ubacili broj *59* u stablo, nakon što se završi ***while*** petlja algoritma pokazivač *parent* pokazuje na čvor *57*, kao što je prikazano na Slici-6b. Novi čvor za element *59* postaje desni potomak čvora *parent*.



Slika 2.6 Primer ubacivanja dva elementa u binarno stablo pretraživanja a) Ubacivanje broja 101, b) ubacivanje broja 59.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Prikazati rezultat ubacivanja elementa 44 u stablo sa slike **Slike 6b**.

Zadatak 2. Koja je vremenska složenost ubacivanja elementa u **BST**?

Poglavlje 3

Rekurzija i stabla

KREIRANJE DUPLIKATA STABLA

Prvo, kreiramo novi čvor sa istim informacionim delom kao i trenutni čvor. Zatim dodajemo levo, odnosno desno podstablo pozivajući rekurzivno odgovarajući metod za dupliranje podstabla

S obzirom da stabla mogu biti definisana rekurzivno, mnoge funkcije za rad sa stablima se najlakše implementiraju korišćenjem rekurzije. U nastavku će biti prikazane neke od rekurzivnih metoda za klase za čvor stabla i za samo stablo. Koristićemo klasu *BinaryNode* kao čvor stabla:

```
class BinaryNode<AnyType> {
 private AnyType element;
 private BinaryNode<AnyType> left;
 private BinaryNode<AnyType> right;
}
```

Započnimo prvo sa implementacijom metoda *duplicate* klase *BinaryNode* koji kreira duplirano stablo. S obzirom da je ovo metoda čvora *BinaryNode*, bićemo sigurni da poziv ove metode neće rezultirati u praznom stablu. Rekurzivni algoritam je veoma prost: Prvo, kreiramo novi čvor sa istim informacionim delom kao i trenutni čvor. Zatim dodajemo levo podstablo pozivajući metod *duplicate* rekurzivno, odnosno dodajemo i desno podstablo pozivanjem metoda *duplicate* rekurzivno.

U oba slučaja, vršimo rekurzivni poziv nakon provere uslova da podstablo nije prazno, U nastavku je dat kod metode:

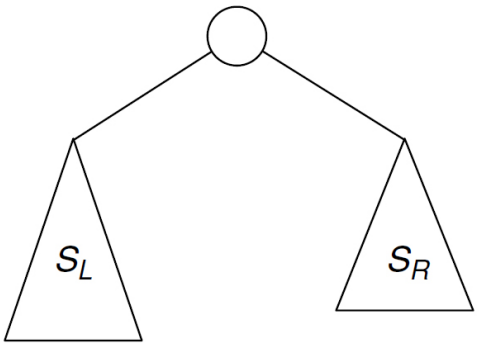
```
/**
 * Return a reference to a node that is the root of a
 * duplicate of the binary tree rooted at the current node.
 */
public BinaryNode<AnyType> duplicate( )
{
 BinaryNode<AnyType> root =
   new BinaryNode<AnyType>( element, null, null );

 if( left != null ) // If there's a left subtree
  root.left = left.duplicate( ); // Duplicate; attach
 if( right != null ) // If there's a right subtree
  root.right = right.duplicate( ); // Duplicate; attach
```

```
 return root; // Return resulting tree
}
```

ODREĐIVANJE BROJA ČVOROVA U STABLU (VELIČINA STABLA)

Veličina stabla se može izračunati kao veličina levog podstabla + veličina desnog podstabla + 1. Najmanje stablo za koje treba odrediti veličinu je prazno stablo (osnovni slučaj rekurzije)

Metod *size* treba da odredi veličinu binarnog stabla koga referencira pokazivač *r*. Ukoliko nacrtamo stablo rekurzivno, kao što je prikazano na Slici-1, možemo da vidimo da se veličina stabla može izračunati kao veličina levog podstabla + veličina desnog podstabla + 1 (jer se koren računa kao čvor), tj. $S_T = S_L + S_R + 1$.



Slika 3.1 Rekurzivni pogled na stablo u cilju određivanja ukupne veličine stabla

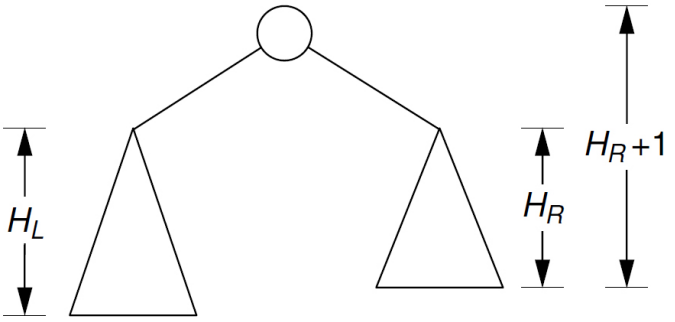
Rekurzivni metod zahteva postojanje osnovnog slučaja koji se rešava bez korišćenja rekurzije. Najmanje stablo za koje treba odrediti veličinu je prazno stablo (ako je *t* jednako *null*), i njegova veličina je očigledno 0. Rekurzivna procedura za određivanje veličine stabla je prikazana u listingu koji sledi:

```
/**
 * Return the size of the binary tree rooted at t.
 */
public static <AnyType> int size( BinaryNode<AnyType> t )
{
 if( t == null )
  return 0;
 else
  return 1 + size( t.left ) + size( t.right );
}
```

ODREĐIVANJE VISINE ČVORA U STABLU

Rekurzivni algoritam se svodi na sledeće: Ako ovu vrednost odredimo za oba podstabla, maksimalna od ove dve vrednosti plus 1 je odgovor koji smo tražili

Konačno, poslednja rekurzivna procedura koju predstavljamo u ovoj sekciji određuje visinu čvora. Implementacija ovog metoda je veoma teška ako ne koristimo rekurziju, ali korišćenjem rekurzije implementacija postaje trivijalna, nakon što se nacrta odgovarajuća skica. Na Slici-2 je prikazano stablo prema rekurentnom principu.



Slika 3.2 Rekurzivni pogled na određivanje visine čvora

Pretpostavimo da levo podstablo ima visinu H_L a desno podstablo visinu H_R. Bilo koji čvor koji se nalazi *d* nivoa dublje u odnosu na koren levog podstabla je *d + 1* nivoa dublji u odnosu na koren celog stabla. Isto važi i za desno podstablo. Stoga, dužina putanje od najdubljeg čvora originalnog stabla je za 1 veća u odnosu na dužinu putanje u odnosu na koren njegovog podstabla.

Ako ovu vrednost odredimo za oba podstabla, maksimalna od ove dve vrednosti plus 1 je odgovor koji smo tražili. Funkcija za određivanje visine stabla je:

$H_T = \text{Max}\ (H_L + 1, H_R + 1)$.

Kod za rekurzivno određivanje visine stabla je prikazan u nastavku.

```
/**
 * Return the height of the binary tree rooted at t.
 */
public static <AnyType> int height( BinaryNode<AnyType> t )
{
 if( t == null )
  return -1;
 else
  return 1 + Math.max( height( t.left ), height( t.right ) );
}
```

Poglavlje 4

Obilazak binarnog stabla

TIPOVI OBILASKA, OBILAZAK PO DUBINI

Postoje dva različita obilaska stabla: Obilazak po dubini (depth-first search - DFS) i Obilazak po širini (breadth-first search - BFS)

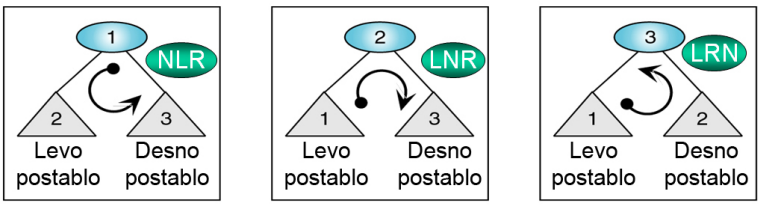
Obilazak binarnog stabla (engl. binary tree traversal) zahteva da se svaki čvor stabla obradi (obiđe) samo jednom. Postoje dva različita obilaska stabla:

- Obilazak po dubini (depth-first search- *DFS*)
- Obilazak po širini (breadth-first search- *BFS*)

Kod obilaska po dubini (depth-first search- DFS), svi potomci deteta se obrađuju pre sledećeg deteta. Kreće se od korena i ispituje se koliko je moguće ići kroz svaku granu pre povratka nazad (engl. backtracking). Postoje tri tipa *DFS* obilaska stabla (Slika-1):

- *Pre-order* (Koren-Levo-Desno) algoritam obilaska prvo obilazi koren, onda levo podstablo i krajnje desno podstablo.
- *In-order* (Levo-Koren-Desno) algoritam obilaska prvo obilazi levo podstablo, zatim koren i onda krajnje levo podstablo.
- *Post-order* (Levo-Desno-Koren) algoritam obilaska prvo obilazi levo postablo, onda desno podstablo i na kraju koren.

Kod obilaska po širini svaki nivo se kompletno procesira pre početka procesiranja sledećeg nivoa.

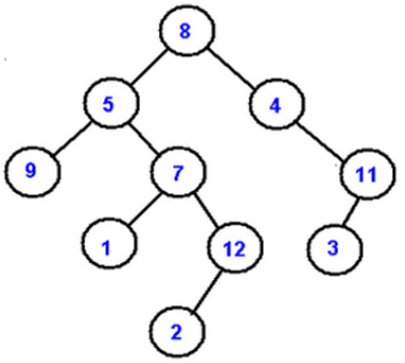


(a) Preorder obilazak **(b) Inorder obilazak** **(c) Postorder obilazak**

Slika 4.1 Tipovi obilaska binarnog stabla po dubini (DFS)

Na Slici-2 je dat primer obilaska binarnog stabla po dubini i širini.

PreOrder: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder: 9, 5, 1, 7, 2, 12, 8, 4, 3. 11
PostOrder: 9,1,2,12,7,5, 3,11,4,8
Po nivou: 8, 5, 4, 9,7,11,1,12, 3,2



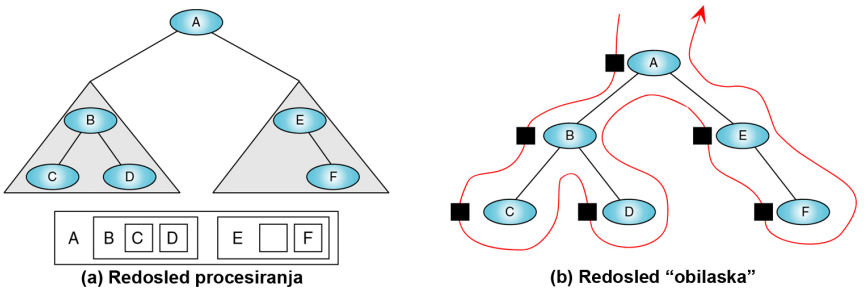
Slika 4.2 Primer obilaska binarnog stabla

Obilazak stabla se može izvršiti i u obrnutom poretku, tj. moguće je obići desno podstablo pre nego se obiđe levo podstablo. Stoga imamo i tri dodatna načina obilaska stabla: obrnuti preorder, inorder i postorder.

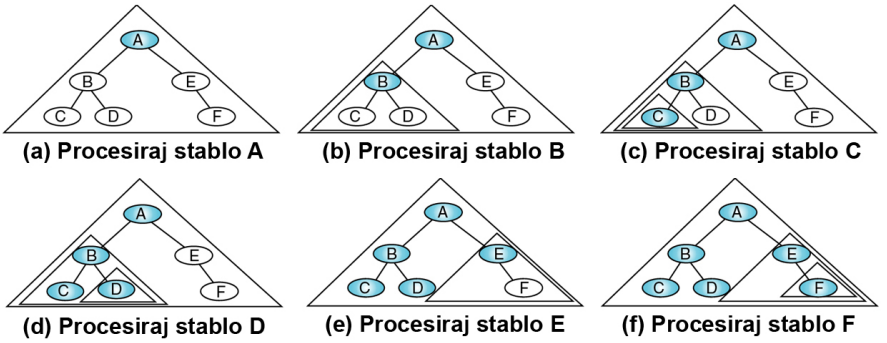
PREORDER OBILAZAK BINARNOG STABLA

Preorder (Koren-Levo-Desno) algoritam obilaska binarnog stabla prvo obilazi koren, zatim levo podstablo i onda krajnje desno podstablo

Preorder (Koren-Levo-Desno) algoritam obilaska binarnog stabla prvo obilazi koren, zatim levo podstablo i onda krajnje desno podstablo. Slika-3 ilustruje *Preorder* obilazak binarnog stabla.



Slika 4.3 Preorder obilazak binarnog stabla



Slika 4.4 Primer Preorder algoritma

Na Slici-4 je opisan postupak *preorder* algoritma korak-po-korak.

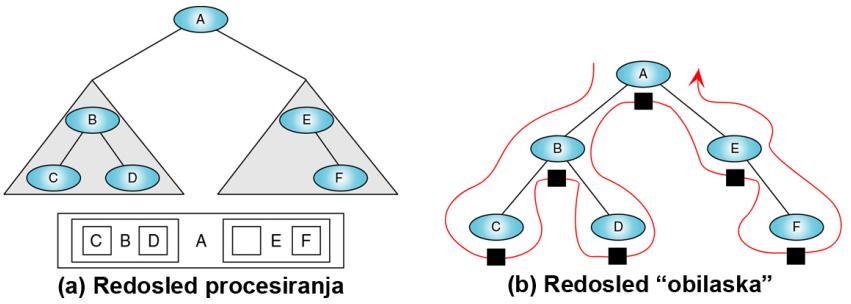
U nastavku je dat kod metode *preorder* koja obilazi i štampa elemente binarnog stabla pretraživanja:

```
public void preorder() {
    preorder(root);
  }

  protected void preorder(TreeNode<E> root) {
    if (root == null) return;
    System.out.print(root.element + " ");
    preorder(root.left);
    preorder(root.right);
  }
```

INORDER OBILAZAK BINARNOG STABLA

In-order (Levo-Koren-Desno) algoritam obilaska prvo obilazi levo podstablo, zatim koren i onda krajnje levo podstablo

In-order (Levo-Koren-Desno) algoritam obilaska prvo obilazi levo podstablo, zatim koren i onda krajnje levo podstablo. Slika-5 ilustruje *Inorder* obilazak binarnog stabla.



Slika 4.5 Inorder obilazak binarnog stabla

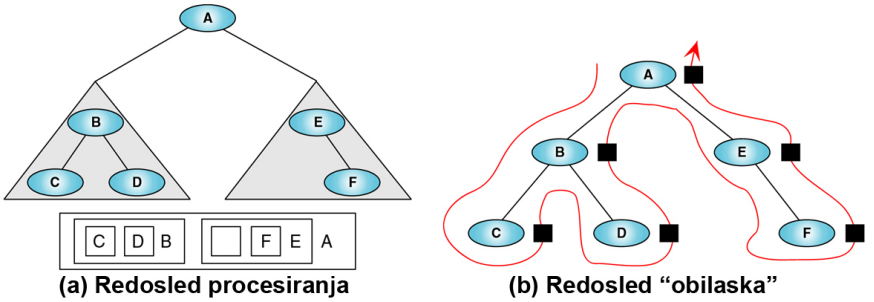
U nastavku je dat kod metode *inorder* koja obilazi i štampa elemente binarnog stabla pretraživanja:

```
public void inorder() {
    inorder(root);
  }

  protected void inorder(TreeNode<E> root) {
    if (root == null) return;
    inorder(root.left);
    System.out.print(root.element + " ");
    inorder(root.right);
  }
```

POSTORDER OBILAZAK BINARNOG STABLA

Post-order (Levo-Desno-Koren) algoritam obilaska prvo obilazi levo postablo, onda desno podstablo i na kraju koren

Post-order (Levo-Desno-Koren) algoritam obilaska prvo obilazi levo postablo, onda desno podstablo i na kraju koren. Slika-6 ilustruje *Postorder* obilazak binarnog stabla.



Slika 4.6 Postorder obilazak binarnog stabla

U nastavku je dat kod metode *postorder* koja obilazi i štampa elemente binarnog stablapretraživanja:

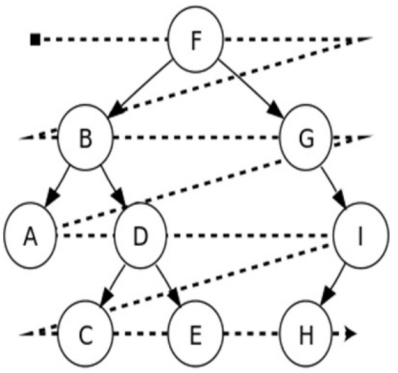
```
public void postorder() {
    postorder(root);
  }

  protected void postorder(TreeNode<E> root) {
    if (root == null) return;
    postorder(root.left);
    postorder(root.right);
    System.out.print(root.element + " ");
  }
```

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

OBILAZAK STABLA PO ŠIRINI (BFS)

Stabla se takođe mogu obilaziti po nivoima, pri čemu se obilazi svaki čvor na nekom nivou pre nego se obilazi čvor na nižem nivou. Ovaj postupak se naziva obilazak stabla po širini (BFS)

Stabla se takođe mogu obilaziti po nivoima, pri čemu se obilazi svaki čvor na nekom nivou pre nego se obilazi čvor na nižem nivou. Ovaj postupak se naziva obilazak stabla po širini. Slika-7 ilustruje koncept obilaska stabla po nivoima – obilazak stabla po širini.



Slika 4.7 Koncept obilaska po nivoima

Opisano pretraživanje po nivoima se naziva Breadth-First Search– *BFS*. Pretraživanje se vrši na svakoj *"dubini"* pre nego se ide na sledeću dubinu. *BFS* počinje u čvoru korena nastavljajući ispitivanje susednih čvorova. Ideja je posetiti čvorove na nivou *"i"* pre posete na nivou *"i+1"*. Tada, za svaki od susednih čvorova, ispituju se njihovi susedni čvorovi koji nisu već posećeni itd.

BFS algoritam može biti napisan na sledeći način:

```
visit(start node)
queue <- start node
WHILE queue is nor empty DO
  x <- queue
  FOR each y such that (x,y) is an edge
                and y has not been visited yet DO
    visit(y)
    queue <- y
  END
END
```

Isečak koda u jeziku Java za BFS može biti napisan kao:

```
public static void levelOrderTraversal(TreeNode startNode) {
 Queue<TreeNode> queue=new LinkedList<TreeNode>();
 queue.add(startNode);
 while(!queue.isEmpty())
 {
  TreeNode tempNode=queue.poll();
  System.out.printf("%d ",tempNode.data);
  if(tempNode.left!=null)
   queue.add(tempNode.left);
  if(tempNode.right!=null)
   queue.add(tempNode.right);
 }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Prikazati inorder, preorder, and postorder obilazak stabli prikazanog na slici **Binarna stabla pretraživanja (BST**) - **Slika 3.**

Zadatak 2. Ako isti skup elemenata ubacimo različitim redosledom u BST stablo, da li će ono izgledati isto? Da li će sva tri tipa obilaska stabla posle biti jednaka?

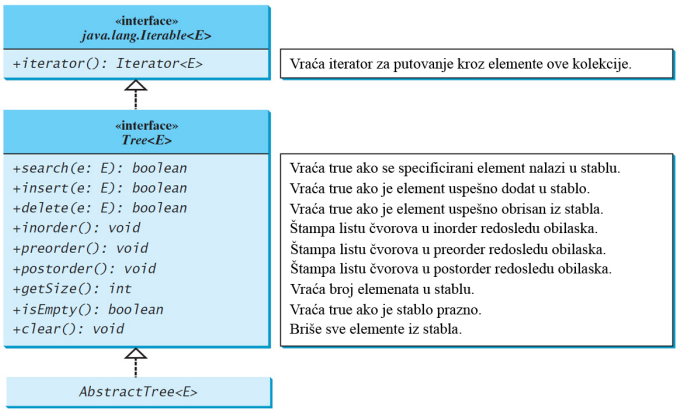
Poglavlje 5

Implementacija BST stabla u Javi

HIJERARHIJA KLASA I INTERFEJSA

Osnovni interfejs koji ćemo definisati za stabla je Tree. Njega će delimično da implementira klasa AbstractTree, dok će klasa BST da implementira sve metode interfejsa Tree

U skladu sa JCF API hijerarhijom, kreiraćemo interfejs *Tree* u kome ćemo definisati skup uobičajenih operacija nad stablima, i zatim ćemo obezbediti i apstraktnu klasu pod nazivom *AbstractTree* koja će delimično da implementira interfejs *Tree*, kao što je prikazano na Slici-1.



Slika 5.1 Interfejs Tree definiše skup osnovnih operacija za rad sa stablima. Apstraktna klasa AbstractTree delimično implementira Tree.

INTERFEJS TREE I KLASA ABSTRACTTREE

U interfejsu Tree ćemo definisati skup uobičajenih operacija nad stablima, obezbedićemo i apstraktnu klasu pod nazivom AbstractTree koja će delimično da implementira interfejs Tree

Interfejs *Tree* može biti implementiran na sledeći način:

```
public interface Tree<E> extends Iterable<E> {
  /** Return true if the element is in the tree */
  public boolean search(E e);

  /** Insert element o into the binary tree
   * Return true if the element is inserted successfully */
```

```
  public boolean insert(E e);

  /** Delete the specified element from the tree
   * Return true if the element is deleted successfully */
  public boolean delete(E e);

  /** Inorder traversal from the root*/
  public void inorder();

  /** Postorder traversal from the root */
  public void postorder();

  /** Preorder traversal from the root */
  public void preorder();

  /** Get the number of nodes in the tree */
  public int getSize();

  /** Return true if the tree is empty */
  public boolean isEmpty();
}
```

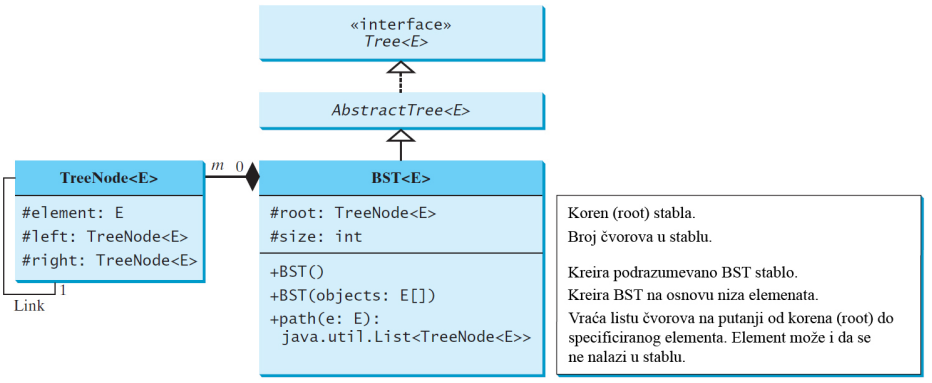
U nastavku je dat primer implementacije apstraktne klase *AbstractTree*:

```
public abstract class AbstractTree<E> implements Tree<E> {
  @Override /** Inorder traversal from the root*/
  public void inorder() {
  }

  @Override /** Postorder traversal from the root */
  public void postorder() {
  }

  @Override /** Preorder traversal from the root */
  public void preorder() {
  }

  @Override /** Return true if the tree is empty */
  public boolean isEmpty() {
    return getSize() == 0;
  }
}
```

KLASA BST

Konkretna klasa BST može biti definisana tako da proširuje klasu AbstractTree, i implementiraće sve ostale metode interfejsa Tree

Konkretna klasa *BST* može biti definisana tako da proširuje klasu *AbstractTree*, kao što je prikazano na Slici-2.



Slika 5.2 Klasa BST definiše konkretno binarno stablo pretrage.

Kompletna implementacija klase za binarno stablo je ostavljena za vežbe. Dovoljno je samo spojiti sve dosad opisane metode za rad sa stablom u jednu jedinstvenu klasu.

IMPLEMENTACIJA KLASE BST

Cilj zadatka je implementacija i upotreba klase za BST stablo. BST klasa treba da sadrži metode za kreiranje, pretraživanje, brisanje i obilašak stabla (DFS, BFS i korišćenjem iteratora)

Implementacija klase za BST u Javi se može videti nastavku sledećem linku:

http://www.cs.armstrong.edu/liang/intro9e/html/BST.html

Na prethodnom linku se nalazi BST stabla i metoda koje su opisane u okviru predavanja: ubacivanje čvora, brisanje čvora, pretraživanje čvora, postorder, preorder i inorder obilaza, kao i implementacija iteratora. Takođe je i implementiran metod koji štampa putanju od korena do određenog čvora E.

Kao zadatak za samostalni rad, studenti mogu dodati u klasu rekurzivne funkcije za određivanje broja elemenata u stablu, visinu čvora i da kreiranju duplikat stabla.

```
public class BST<E extends Comparable<E>>
    extends AbstractTree<E> {
  protected TreeNode<E> root;
  protected int size = 0;

  /** Create a default binary tree */
  public BST() {
  }

  /** Create a binary tree from an array of objects */
  public BST(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      insert(objects[i]);
  }
```

```
@Override /** Returns true if the element is in the tree */
public boolean search(E e) {
  TreeNode<E> current = root; // Start from the root

  while (current != null) {
    if (e.compareTo(current.element) < 0) {
      current = current.left;
    }
    else if (e.compareTo(current.element) > 0) {
      current = current.right;
    }
    else // element matches current.element
      return true; // Element is found
  }

  return false;
}

@Override /** Insert element o into the binary tree
 * Return true if the element is inserted successfully */
public boolean insert(E e) {
  if (root == null)
    root = createNewNode(e); // Create a new root
  else {
    // Locate the parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null)
      if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (e.compareTo(parent.element) < 0)
      parent.left = createNewNode(e);
    else
      parent.right = createNewNode(e);
  }

  size++;
  return true; // Element inserted
}

protected TreeNode<E> createNewNode(E e) {
  return new TreeNode<E>(e);
}
```

```
@Override /** Inorder traversal from the root*/
public void inorder() {
  inorder(root);
}

/** Inorder traversal from a subtree */
protected void inorder(TreeNode<E> root) {
  if (root == null) return;
  inorder(root.left);
  System.out.print(root.element + " ");
  inorder(root.right);
}

@Override /** Postorder traversal from the root */
public void postorder() {
  postorder(root);
}

/** Postorder traversal from a subtree */
protected void postorder(TreeNode<E> root) {
  if (root == null) return;
  postorder(root.left);
  postorder(root.right);
  System.out.print(root.element + " ");
}

@Override /** Preorder traversal from the root */
public void preorder() {
  preorder(root);
}

/** Preorder traversal from a subtree */
protected void preorder(TreeNode<E> root) {
  if (root == null) return;
  System.out.print(root.element + " ");
  preorder(root.left);
  preorder(root.right);
}

/** This inner class is static, because it does not access
    any instance members defined in its outer class */
public static class TreeNode<E extends Comparable<E>> {
  protected E element;
  protected TreeNode<E> left;
  protected TreeNode<E> right;

  public TreeNode(E e) {
    element = e;
  }
}

@Override /** Get the number of nodes in the tree */
```

```
public int getSize() {
  return size;
}

/** Returns the root of the tree */
public TreeNode<E> getRoot() {
  return root;
}

/** Returns a path from the root leading to the specified element */
public java.util.ArrayList<TreeNode<E>> path(E e) {
  java.util.ArrayList<TreeNode<E>> list =
    new java.util.ArrayList<TreeNode<E>>();
  TreeNode<E> current = root; // Start from the root

  while (current != null) {
    list.add(current); // Add the node to the list
    if (e.compareTo(current.element) < 0) {
      current = current.left;
    }
    else if (e.compareTo(current.element) > 0) {
      current = current.right;
    }
    else
      break;
  }

  return list; // Return an array of nodes
}

@Override /** Delete an element from the binary tree.
 * Return true if the element is deleted successfully
 * Return false if the element is not in the tree */
public boolean delete(E e) {
  // Locate the node to be deleted and also locate its parent node
  TreeNode<E> parent = null;
  TreeNode<E> current = root;
  while (current != null) {
    if (e.compareTo(current.element) < 0) {
      parent = current;
      current = current.left;
    }
    else if (e.compareTo(current.element) > 0) {
      parent = current;
      current = current.right;
    }
    else
      break; // Element is in the tree pointed at by current
  }

  if (current == null)
    return false; // Element is not in the tree
```

```
    // Case 1: current has no left children
    if (current.left == null) {
      // Connect the parent with the right child of the current node
      if (parent == null) {
        root = current.right;
      }
      else {
        if (e.compareTo(parent.element) < 0)
          parent.left = current.right;
        else
          parent.right = current.right;
      }
    }
    else {
      // Case 2: The current node has a left child
      // Locate the rightmost node in the left subtree of
      // the current node and also its parent
      TreeNode<E> parentOfRightMost = current;
      TreeNode<E> rightMost = current.left;

      while (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.right; // Keep going to the right
      }

      // Replace the element in current by the element in rightMost
      current.element = rightMost.element;

      // Eliminate rightmost node
      if (parentOfRightMost.right == rightMost)
        parentOfRightMost.right = rightMost.left;
      else
        // Special case: parentOfRightMost == current
        parentOfRightMost.left = rightMost.left;
    }

    size--;
    return true; // Element inserted
  }

  @Override /** Obtain an iterator. Use inorder. */
  public java.util.Iterator<E> iterator() {
    return new InorderIterator();
  }

  // Inner class InorderIterator
  private class InorderIterator implements java.util.Iterator<E> {
    // Store the elements in a list
    private java.util.ArrayList<E> list =
      new java.util.ArrayList<E>();
    private int current = 0; // Point to the current element in list

    public InorderIterator() {
```

```
      inorder(); // Traverse binary tree and store elements in list
    }

    /** Inorder traversal from the root*/
    private void inorder() {
      inorder(root);
    }

    /** Inorder traversal from a subtree */
    private void inorder(TreeNode<E> root) {
      if (root == null)return;
      inorder(root.left);
      list.add(root.element);
      inorder(root.right);
    }

    @Override /** More elements for traversing? */
    public boolean hasNext() {
      if (current < list.size())
        return true;

      return false;
    }

    @Override /** Get the current element and move to the next */
    public E next() {
      return list.get(current++);
    }

    @Override /** Remove the current element */
    public void remove() {
      delete(list.get(current)); // Delete the current element
      list.clear(); // Clear the list
      inorder(); // Rebuild the list
    }
  }

  /** Remove all elements from the tree */
  public void clear() {
    root = null;
    size = 0;
  }
}
```

OPIS METODA KLASE BST

Navodimo opis metoda klase BST

Metod **insert(E e)** klase *BST* (lines 36–64) kreira čvor za element **e** i ubacuje ga u stablo. Ako je stablo prazno, taj čvor postaje koren. Inače, metod nalazi odgovarajućeg roditelja tako da

stablo ostane uređeno. Ako element već postoji u stablu,
metod vraća **false**; inače vraća **true**.

Metod **inorder()** klase BST (lines 71–81) poziva **inorder(root)** da obiđe celo stablo. Metod **inorder(TreeNode root)** obilazi stablo sa zadatim čvorem od kojeg se kreće. Ovaj metod je implemntiran rekurzivno. Rekurzivno obilazi levo podstablo, pa koren, a zatim i desno podstablo. Izlaz iz rekurzije je slučaj kada je stablo prazno.

Metod **postorder()** (lines 84–94) i metod **preorder()** (lines 97–107) su implementirani slično koristeći rekurziju.

Metod **path(E e)** (lines 132–150) vraća putanju sastavljenu od čvorova kao listu. Putanja počinje u korenu i vodi do elementa **e**. Može se desiti da element ne postoji u stablu. Na primer, na slici **Binarna stabla pretraživanja (BST)-Slika-6a**, **path(45)** sadrži čvorove za elemente 60, 55, and 45, a putanja **path(58)** sadrži čvorove za 60, 55, and 57.

Klasa *TestBST* prikazana na sledećem slajdu, kreira binarno stablo pretrage koristeći **BST** strukturu. Program dodaje stringove u stablo (lines 5–11), obilazi stablo u inorder, postorder, and preorder redosledu (lines 14–20), traži element stabla (line 24), i nalazi putanju od čvora koji sadrži element *Peter* do korena (lines 28–31).

TESTIRANJE BST STABLA

Cilj zadatka je testiranje klase za BST

Testiranja kreirane klase za *BST* (TestBST.java):

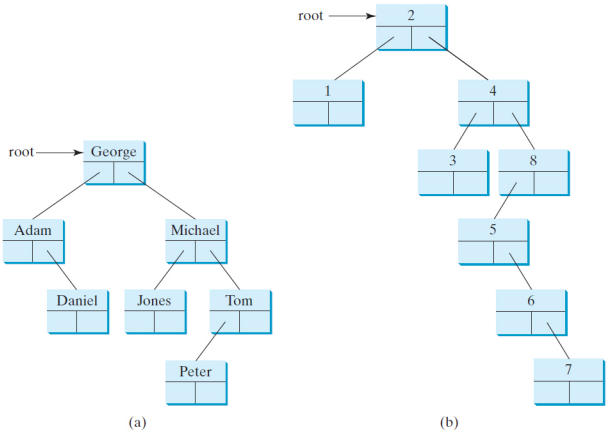
```
public class TestBST {
  public static void main(String[] args) {
    // Create a BST
    BST<String> tree = new BST<String>();
    tree.insert("George");
    tree.insert("Michael");
    tree.insert("Tom");
    tree.insert("Adam");
    tree.insert("Jones");
    tree.insert("Peter");
    tree.insert("Daniel");

    // Traverse tree
    System.out.print("Inorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());

    // Search for an element
    System.out.print("\nIs Peter in the tree? " +
    tree.search("Peter"));
```

```
    // Get a path from the root to Peter
    System.out.print("\nA path from the root to Peter is: ");
    java.util.ArrayList<BST.TreeNode<String>>  path
    = tree.path("Peter");
    for (int i = 0; path != null &amp;amp;&amp;amp; i < path.size(); i++)
    System.out.print(path.get(i).element + " ");

    Integer[] numbers = {2, 4, 3, 1, 8, 5, 6, 7};
    BST<Integer> intTree = new BST<Integer>(numbers);
    System.out.print("\nInorder (sorted): ");
    intTree.inorder();
  }
}
```

Vrši se provera **path != null** na liniji 30 da bismo bili sigurni da je putanja različita od **null** pre pozivanja **path.get(i).** Ovo je jedan primer defanzivnog programiranja kako bi se sprečile greške u budućnosti.

Program kreira još jedno stablo za čuvanje celih brojeva (line 34). Posle ubacivanja svih elemenata stablo treba da izgleda kao na slici **Slika-3.**

Ako se elementi ubacuju u različitom redosledu (na primer, Daniel, Adam, Jones, Peter, Tom, Michael, George), stablo će izgledati drugačije. Međutim, **inorder** obilazak štampa elemente u uvek istom redosledu sve dok je skup elemenata isti. Takav tip obilaska daje sortiranu listu.



Slika 5.3 Stabla kreirana u programu TestBST posle ubacivanja elemenata

ZADACI ZA SAMOSTALNO VEŽBANJE

Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeće zadatke:

Zadatak 1. Dodati sledeće metode u klasu za *BST* koja je opisana na vežbama:

```
/** Displays the nodes in a breadth-first traversal */
public void breadthFirstTraversal()
```

```
/** Returns the height of this binary tree, i.e., the
* number of the nodes in the longest path of the root to a leaf */
public int height()
```

Zadatak 2. Kompletno binarno stablo je ono stablo čiji se svi listovi nalaze na istom nivou. Dodati metod u klasu *BST* koja vraća rezultat ***true*** (tačno) ako je binarno stablo kompletno (Nagoveštaj: Broj čvorova u kompletnom binarnom stablu je $2^{depth-1}$, gde je *depth* dubina stabla).

```
boolean isFullBST()
```

Zadatak 3. Implementirati metode *inorder*, *preorder* i *postorder* u klasi *BST* korišćenjem steka umesto rekurzije. Write a test program that
prompts the user to enter 10 integers, stores them in a BST, and invokes the inorder method to display the elements.

Zadatak 4. Dodati metod u klasu *BST* koji vraća broj listova u stablu na sledeći način:

```
public int getNumberOfLeaves()
```

Zadatak 5. Implementirati **search(element)** metodu pomoću rekurzije.

Zadatak 6. Dodati metod u klasu *BST* koja vraća broj čvorova koji nisu listovi:

```
public int getNumberofNonLeaves()
```

Poglavlje 6

Brisanje čvora iz BST stabla

OSNOVNI SLUČAJEVI BRISANJE ČVORA IZ BST STABLA

Da bi obrisali element iz BST stabla, prvo ga treba locirati u stablu a zatim uzeti u obzir dva slučaja: da li čvor ima ili nema svog levog potomka, pre nego se obriše iz stabla

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Za razliku od ubacivanja elemenata, brisanje elemenata iz stabla je znatno komplikovanije.

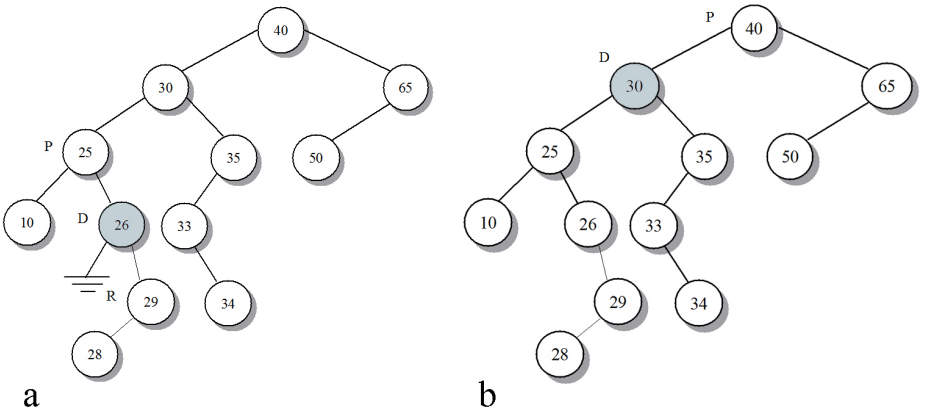
Da bi obrisali element iz binarnog stabla pretrage, neophodno je da se prvo locira čvor koji želimo da obrišemo, a takođe i njegov roditelj.

Neka *current* pokazuje na čvor koji treba biti obrisan iz stabla, i neka *parent* pokazuje na roditelja čvora na koji pokazuje *current* node. Čvor *current* može biti levo ili desno dete čvora *parent*.

Treba biti uzeta u razmatranje dva slučaja:

Slučaj 1: Trenutni čvor *current* nema levog potomka (Slika-1a – čvor 26)

Slučaj 2: Trenutni čvor *current* ima levog potomka (Slika 1b – čvor 30).

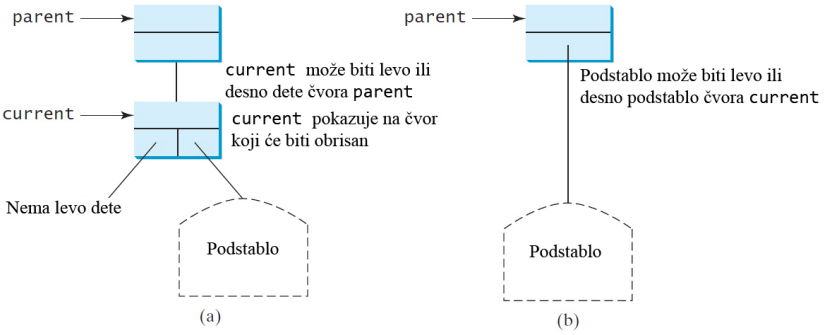


Slika 6.1 Brisanje čvora iz BST stabla a) Slučaj 1: čvor nema levog potomka – čvor 26 b) Slučaj 2: čvor ima čevog potomka – čvor 30

SLUČAJ 1: TRENUTNI ČVOR NEMA LEVOG POTOMKA

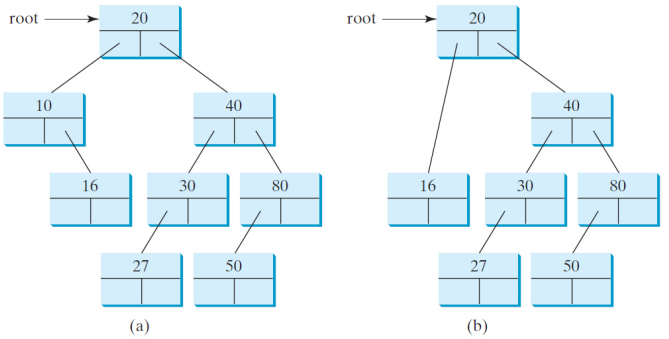
U slučaju da čvor koga brišemo (current) nema levog potomka treba jednostavno povezati njegovog roditelja (parent) sa desnim naslednikom trenutnog (current) čvora

Slučaj 1: Trenutni čvor nema levog potomka, kao što je prikazano na Slici 2a. U ovom slučaju, jednostavno povezati njegovog roditelja (*parent*) sa desnim naslednikom trenutnog čvora (*curent.right*), kao što je prikazano na Slici 2b.



Slika 6.2 Slučaj 1: Trenutni čvor, na koji pokazuje current, nema levog potomka.

Na primer, da bi obrisali čvor 10 iz stabla sa Slike 3a, neophodno je povezati roditelja čvora 10 (*root*, element 20), sa desnim naslednikom čvora 10 (a to je čvor 16), kao što je prikazano na Slici 3b.

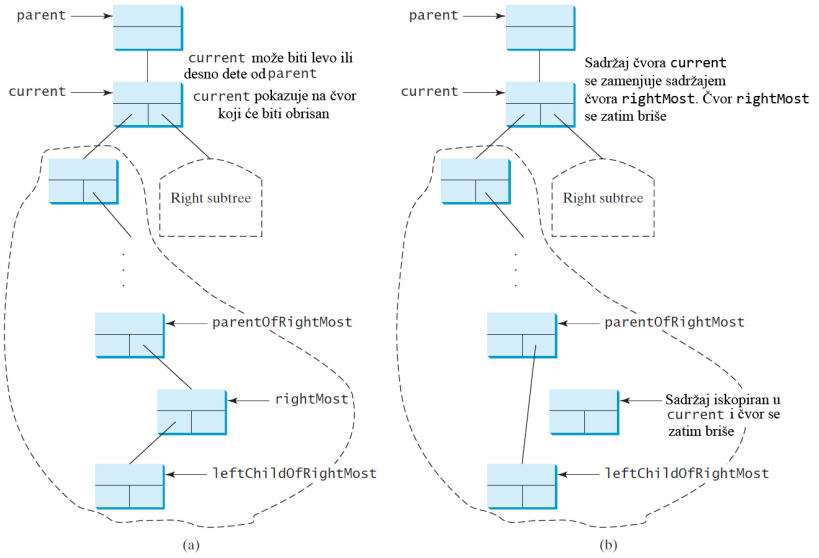


Slika 6.3 Slučaj 1: Brisanje čvora 10 iz stabla sa slike (a) rezultuje u stablu na slici (b).

SLUČAJ 2: TRENUTNI ČVOR IMA LEVOG POTOMKA

U ovom slučaju najveći čvor levog podstabla (rightmost) dolazi na mesto elementa koji se briše. Ako čvor rightmost ima levo podstablo onda se ono prebacuje na roditelja čvora rightmost

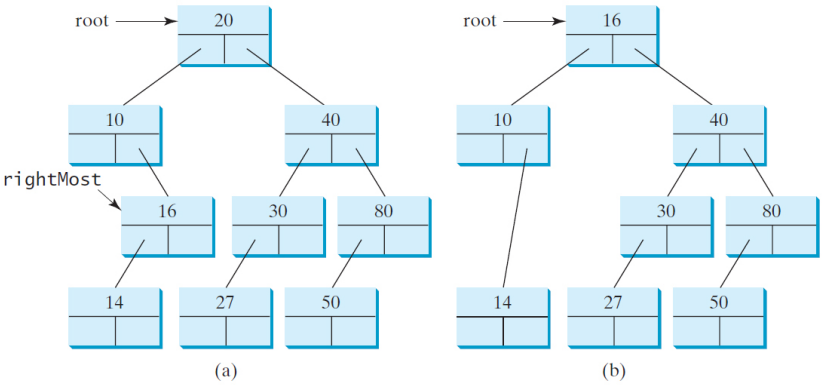
Neka je *rightMost* pokazivač na čvor koji sadrži najveći element u levom podstablu čvora *current* i neka *parentOfRightMost* pokazuje na roditeljski čvor *rightMost* čvora, kao što je prikazano na Slici 4a.



Slika 6.4 Slučaj 2: Postupak brisanja čvora current kada čvor current ima levog potomka

Primetimo da *rightMost* čvor ne može da ima desnog potomka ali može da ima levog potomka.

Zamenimo vrednost u čvoru *current* sa vrednošću koja se nalazi u čvoru *rightMost*, povežimo čvor *parentOfRightMost* sa levim potomkom čvora *rightMost*, a zatim obrišimo čvor *rightMost*, kao što je prikazano na Slici 4b. Na primer, uzmimo slučaj brisanja čvora *20*, iz primera sa Slike 5a.



Slika 6.5 Brisanje čvora 20 iz stabla (a) rezultira u stablu (b).

Čvor *rightMost* ima element sa vrednošću *16*. Umesto vrednosti *20* u čvoru *root* (koji je ustvari *current* za trenutni slučaj) upisujemo vrednost *16*, zatim se briše čvor *rightMost*, dok čvor sa vrednošću *14* postaje desni potomak čvora *10*, kao što je prikazano na Slici 5b.

IMPLEMENTACIJA ALGORITMA ZA BRISANJE ČVORA IZ STABLA

Metod delete locira čvor koji se briše (current) i locira njegovog roditelja. Nakon toga, vrši se brisanje čvora i premošćavanje u zavisnosti od prethodno opisanih slučajeva 1 ili 2

U nastavku je data implementacija metoda *delete* koji briše element iz binarnog stabla. Ukoliko element nije u stablu, metod vraća *false* kao rezultat.

```
public boolean delete(E e) {
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null) {
    if (e.compareTo(current.element) < 0) {
        parent = current;
        current = current.left;
    }
    else if (e.compareTo(current.element) > 0) {
        parent = current;
        current = current.right;
    }
    else
        break; // Element is in the tree pointed at by current
    }

    if (current == null)
    return false; // Element is not in the tree

    // Case 1: current has no left children
    if (current.left == null) {
    if (parent == null) {
        root = current.right;
    }
    else {
        if (e.compareTo(parent.element) < 0)
        parent.left = current.right;
        else
        parent.right = current.right;
    }
    }
    else {
    // Case 2:
    TreeNode<E> parentOfRightMost = current;
    TreeNode<E> rightMost = current.left;

    while (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.right; // Keep going to the right
    }
```

```
    current.element = rightMost.element;

    if (parentOfRightMost.right == rightMost)
        parentOfRightMost.right = rightMost.left;
    else
        // Special case: parentOfRightMost == current
        parentOfRightMost.left = rightMost.left;
    }

    size--;
    return true; // Element inserted
}
```

Metod *delete* locira čvor koji se briše (pod nazivom *current*) i locira njegovog roditelja. Ukoliko je *current* jednako *null*, element se ne nalazi u stablu. U tom slučaju metod *delete* vraća *false* kao rezultat. Može se primetiti da ako je je stablo prazno, onda su oba pokazivača *current* i *parent* jednaki *null*. Takođe se može primetiti da ako je *current* jednak *root*, onda ne postoji roditelj, tj. *parent* je *null*.

U *Slučaju 1*, čvor *current* nema levog naslednika (tj. *current.left* == *null*). Ukolko je *parent* jednak *null*, dodeliti *current.right* korenu root. U suprotnom, dodeliti vrednost *current.right* ili pokazivaču *parent.left* ili *parent.right*, u zavisnosti da li je *current* levi ili desni naslednik čvora *parent*.

U *Slučaju 2*, čvor *current* ima levog naslednika. Algoritam locira najveći (right most) element levog podstabla čvora *current* i takođe njegovog roditelja (pod nazivom *parentOfRightMost*). Vrednost koja se nalazi u čvoru *current* se zamenjuje vrednošću koja je u čvoru *rightMost*; vrednost *rightMost.left* se dodeljuje čvorovima ili *parentOfRightMost.right* ili *parentOfRightMost.left*, u zavisnosti od toga da li je čvor *rightMost* desni ili levi potomak čvora *parentOfRightMost*.

TESTIRANJE ALGORITMA ZA BRISANJE ČVORA

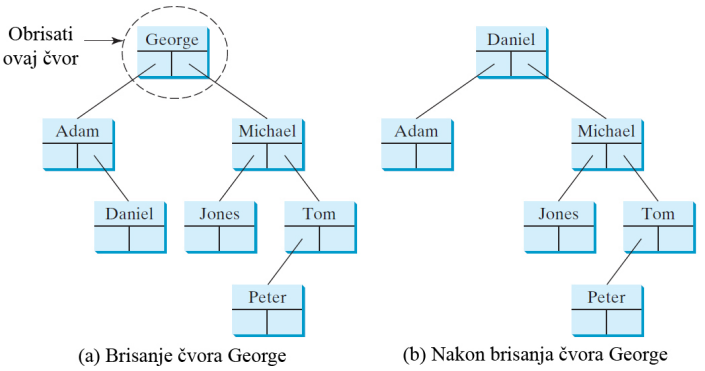
Cilj programa i ilustrativnih primera je detaljan uvid u brisanje čvorova iz BST stabla za različite slučajeve koji mogu da se jave (čvor koji se briše ima/nema levog potomka)

U nastavku je dat listing koji briše elemente iz *BST* stabla.

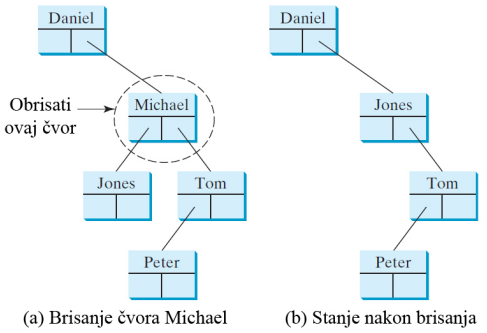
```
public class TestBSTDelete {
  public static void main(String[] args) {
    BST<String> tree = new BST<>();
    tree.insert("George");
    tree.insert("Michael");
    tree.insert("Tom");
    tree.insert("Adam");
    tree.insert("Jones");
    tree.insert("Peter");
```

```
    tree.insert("Daniel");
    printTree(tree);

    System.out.println("\nAfter delete George:");
    tree.delete("George");
    printTree(tree);

    System.out.println("\nAfter delete Adam:");
    tree.delete("Adam");
    printTree(tree);

    System.out.println("\nAfter delete Michael:");
    tree.delete("Michael");
    printTree(tree);
  }

  public static void printTree(BST tree) {
    // Traverse tree
    System.out.print("Inorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

Slike 6 i 7 prikazuju kako se stablo menja nakon što se neki elementi brišu iz njega.



Slika 6.6 Brisanje imena George pripada slučaju 2.



Slika 6.7 Brisanje imena Michael pripada slučaju 2.

ZADACI ZA SAMOSTALNI RAD

Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeće zadatke:

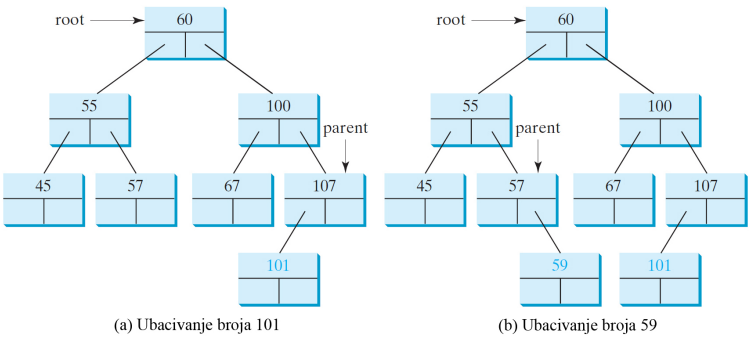
Zadatak 1. Prikazati rezultat brisanja elementa 55 iz stabla prikazanog na slici **Slika-8b.**

Zadatak 2. Prikazati rezultat brisanja elementa 60 iz stabla prikazanog na slici **Slika-8b**.

Zadatak 3. Koja je vremenska složenost brisanja elementa iz **BST**?

Zadatak 4. Da li algoritam ostaje i dalje korektan ako linije 204–208 u implementaciji **BST** metode **delete()** zamene sa sledećim kodom?

```
parentOfRightMost.right = rightMost.left;
```



Slika 6.8 Primer za samostalni rad

Zadatak 5. Napisati program koji iz ulazne datoteke učitava podatke o studentima: Ime, Prezime, Godina upisa, Godina studija, Prosek. Ulazne podatke organizovati u obliku binarnog stabla sortiranog po prezimenu studenta. Za uneto prezime ispisati podatke svih studenata sa tim prezimenom.

Zadatak 6. Napisati program koji pakuje objekte različitih težina u kontejner. Svaki kontejner može da skladišti maksimalno 10 kg robe. Program treba da koristi pohlepni (greedy) algoritam koji neki objekat pakuje u prvi kontejner u koji može da stane. Vaš program treba da zahteva od korisnika da unese ukupan broj objekata i težinu svakog od objekata. Kao rezultat,

program treba da prikaže ukupan broj kontejnera koji je potreban da bi se spakovali objekti, kao i sadržaj u svakom od kontejnera. U nastavku je primer izvršavanja programa:

```
Enter the number of objects: 5
Enter the weights of the objects: 7 5 2 3 5 8
Container 1 contains objects with weight 7 2
Container 2 contains objects with weight 5 3
Container 3 contains objects with weight 5
Container 4 contains objects with weight 8
```

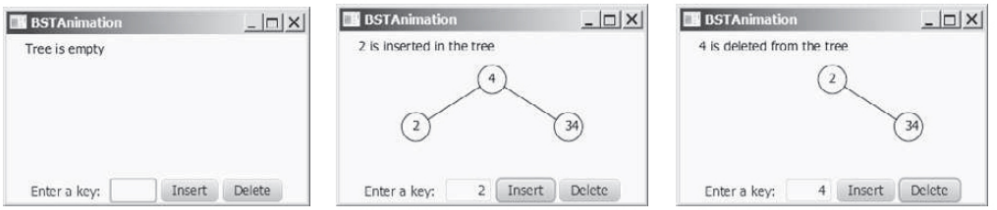
Poglavlje 7

Vizuelizacija stabla i MVC

VIZUELIZACIJA STABLA

Možete da koristite rekurziju da bi prikazali binarno stablo

Kako najlakše da prikažete (nacrtate) binarno stablo? TO je jedna rekurzivna struktura, pa možete da ga prikažete primenom rekurzije. MOžete jednostavno da prikažete koren (root), a zatim da rekurzivno prikažete svako podstablo pojedinačno. Radi uprošćenja, pretpostavićemo da su ključevi prosti celi brojevi manji od 100. Na Slici 1 su prikazani primeri izvršavanja programa. Pa da vidimo kako da to sprovedemo u delo.



Slika 7.1 Predstavljanje binarnog stabla primenom grafičkog programa

KLASE BSTANIMATION I BTVIEW

Cilj zadatka je primena rekurzivnog postupka za vizualizaciju binarnog stabla u Javi, pri čemu se koristi javafx aplikacija i panel u kome se iscrtava stablo

Klasa *BSTAnimation* može biti definisana na sledeći način:

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;

public class BSTAnimation extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
```

```
    BST<Integer> tree = new BST<>(); // Create a tree

    BorderPane pane = new BorderPane();
    BTView view = new BTView(tree); // Create a View
    pane.setCenter(view);

    TextField tfKey = new TextField();
    tfKey.setPrefColumnCount(3);
    tfKey.setAlignment(Pos.BASELINE_RIGHT);
    Button btInsert = new Button("Insert");
    Button btDelete = new Button("Delete");
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(new Label("Enter a key: "),
      tfKey, btInsert, btDelete);
    hBox.setAlignment(Pos.CENTER);
    pane.setBottom(hBox);

    btInsert.setOnAction(e -> {
      int key = Integer.parseInt(tfKey.getText());
      if (tree.search(key)) { // key is in the tree already
        view.displayTree();
        view.setStatus(key + " is already in the tree");
      }
      else {
        tree.insert(key); // Insert a new key
        view.displayTree();
        view.setStatus(key + " is inserted in the tree");
      }
    });

    btDelete.setOnAction(e -> {
      int key = Integer.parseInt(tfKey.getText());
      if (!tree.search(key)) { // key is not in the tree
        view.displayTree();
        view.setStatus(key + " is not in the tree");
      }
      else {
        tree.delete(key); // Delete a key
        view.displayTree();
        view.setStatus(key + " is deleted from the tree");
      }
    });

    // Create a scene and place the pane in the stage
    Scene scene = new Scene(pane, 450, 250);
    primaryStage.setTitle("BSTAnimation"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
```

```
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

Klasa *BTView*:

```
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;

public class BTView extends Pane {
  private BST<Integer> tree = new BST<>();
  private double radius = 15; // Tree node radius
  private double vGap = 50; // Gap between two levels in a tree

  BTView(BST<Integer> tree) {
    this.tree = tree;
    setStatus("Tree is empty");
  }

  public void setStatus(String msg) {
    getChildren().add(new Text(20, 20, msg));
  }

  public void displayTree() {
    this.getChildren().clear(); // Clear the pane
    if (tree.getRoot() != null) {
      // Display tree recursively
      displayTree(tree.getRoot(), getWidth() / 2, vGap,
        getWidth() / 4);
    }
  }

  /** Display a subtree rooted at position (x, y) */
  private void displayTree(BST.TreeNode<Integer> root,
      double x, double y, double hGap) {
    if (root.left != null) {
      // Draw a line to the left node
      getChildren().add(new Line(x - hGap, y + vGap, x, y));
      // Draw the left subtree recursively
      displayTree(root.left, x - hGap, y + vGap, hGap / 2);
    }

    if (root.right != null) {
      // Draw a line to the right node
      getChildren().add(new Line(x + hGap, y + vGap, x, y));
      // Draw the right subtree recursively
      displayTree(root.right, x + hGap, y + vGap, hGap / 2);
    }
```

```
    // Display a node
    Circle circle = new Circle(x, y, radius);
    circle.setFill(Color.WHITE);
    circle.setStroke(Color.BLACK);
    getChildren().addAll(circle,
      new Text(x - 4, y + 4, root.element + ""));
  }
}
```

OPIS METODA

Navodimo opis metoda klase BSTAnimation

U kodu klase *BSTAnimation,* BSTAnimation.java, kreirano je stablo (line 14) i prikaz stabla je postavljen u okviru okna (pane) (line 18). Nakon što se novi ključubaci u stablo (line 37), vrši se njegovo ponovno iscrtavanje da bi se prihvatile poslednje izmene (line 38). Nakon što se ključ obriše iz stabla (line 49), opet se vrši ponovno iscrtavanje stabla da bi se videle izmene nakon brisanja (line 50).

U kodu klase *BTView*, BTView.java, čvor je prikazan kao krug poluprečnika (radius) vrednosti 15 (line 48). Rastojanje izmedju dva susedna nivoa (levels) stabla je definisano u vGap i iznosi 50 (line 25). hGap (line 32) definiše horizontalno rastojanje izmedju dva susedna čvora. Vrednost je umanjena za pola (hGap / 2) u sledećem nivou kada se metod displayTree rekurzivno pozove (lines 44, 51). Primetiti da se vGap ne menja u stablu.

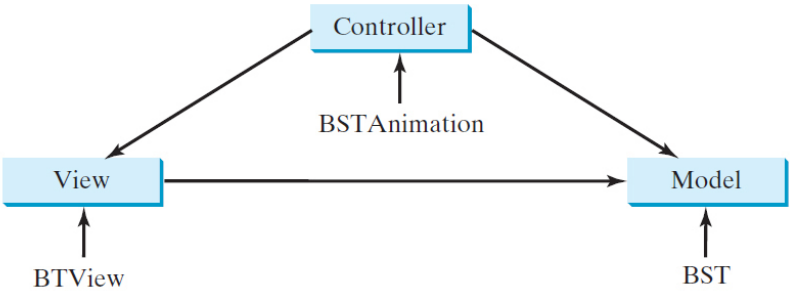
Metod *displayTree* rekurzivno poziva da bi isccrta odgovarajuće levo (lines 33–38) i desno podstablo (lines 40–45) ukoliko stablo nije prazno. U oknu)pane) se dodaje i linija koja povezuje dva čvora (lines 35, 42). Napomenimo da metod prvo dodaje liniju na okno a tek onda dodaje krug (line 52) tako da se krugovi crtaju preko linija da bi se postigli bolji vizuelni efekti .

Program pretpostavlja da su ključevi celi brojevi. Možete lako da modifikujete program korišćenjem generičkih tipova da bi se prikazivali karakteri ili stringovi.

MVC

MVC je model-view-controller (MVC) software architecture

Vizuelizacija stabla je primer model-view-controller (MVC) softverske arhitekture. Ovo je veoma bitna arhitektura za razvoj softvera. Model se koristi za čuvanje i upravljanje podacima. Pogled (view) se koristi za vizuelno predstavljanje podataka. Kontroler upravlja korisničkom interakcijom sa modela i kontroliše pogled (view), kao što je prikazano na Slici-2.



Slika 7.2 Kontroler dobija podatke i skladišti ih u modelu. Pogled prikazuje podatke koji su smešteni u modelu.

MVC arhitektura razdvaja skladešetenje i upravaljane podacima od vizuelnog predstavljanja podataka. Ona ima dve glavne pogodnosti:

■ Omogućava višestruke poglede, tako da podaci mogu biti deljenji preko istog modela. Na primer, uz već postojeći, možete kreirati novi pogled u kome se vizuelizuje stablo sa korenom na levoj strani i stablom koje raste u desno.

■ Ona uprošćava proces pisanja složenih aplikacija i čini komponente skalabilnim i lakšim za održavanja. Izmene izvršene nad pogledom ne utiču na model, i obratno.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

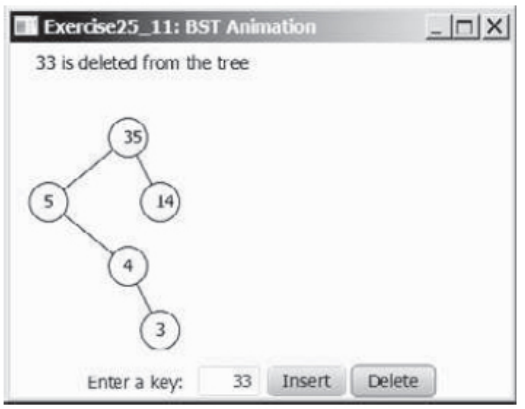
Zadatak 1. Koliko puta se metod **displayTree** poziva ako je stablo prazno? Koliko puta se metod **displayTree** poziva ako stablo ima 100 čvorova?

Zadatak 2. U kom redosledu metod **displayTree** obilazi stablo: inorder, preorder ili postorder?

Zadatak 3. Šta će se desiti ako se linije 47–52 u *BTView* pomere na liniju 33?

Zadatak 4. Napisati novi pogled koji će prikazati stablo horizontalno kao na **Slika-3.**

Zadatak 5. Implementirati **clone** i **equals** metode za *BST* klasu. Dva BST stabla su jednaka ako sadrže iste elemente. Metod kloniranja treba da vrati identičnu kopiju stabla.



Slika 7.3 Binarno stablo prikazano horizontalno

Poglavlje 8

Iteratori za BST stabla

UVODNA RAZMATRANJA

Do sada smo se upoznali sa implementacijom iteratora za sekvencijalne strukture podataka - setovi i liste. Isti princip je moguće koristiti u cilju putovanja kroz elemente binarnog stabla

Metode koje smo implementirali za binarno stablo *inorder()*, *preorder()*, i *postorder()* prikazuju elemente u inorder, preorder i postorder redosledu. Ove metode su limitirane isključivo na prikazivanje elemenata. Ukoliko imamo potrebu da procesiramo elemente stabla a ne da ih prikazujemo, onda ove metode ne mogu biti korišćene. Do sada smo se upoznali sa korišćenjem i implementacijom iteratora za sekvencijalne strukture podataka kao što su setovi i liste. Isti princip je moguće koristiti kod stabla u cilju putovanja kroz elemente binarnog stabla.

Interfejs *java.lang.Iterable*definiše metod *iterator*, koji vraća instancu iteratora *java.util.Iterator*. Ovaj interfejs smo više puta spomenuli i on definiše osnovne metode za putovanje kroz kolekciju kao što su *hasNext()*, *next()* i *remove()*. Interfejs *Tree* proširuje interfejs *java.lang.Iterable*. Pošto je *BST* podklasa od *AbstractTree*, a *AbstractTree* implementira interfejs *Tree*, *BST* je podtip interfejsa *Iterable*.

Moguće je obići binarno stablo u inorder, preorder, ili postorder redosledu. Pošto se obilazak inorder uglavnom koristi, stoga ćemo upravno njega upotrebiti da bi putovali kroz sve elemente u binarnom stablu.

IMPLEMENTACIJA ITERATORA

Moguće je obići binarno stablo u inorder, preorder, ili postorder redosledu. Pošto se obilazak inorder uglavnom koristi, Iterator stabla je implementiran korišćenjem inorder obilaska

Definisaćemo klasu *InorderIterator*unutar klase za *BST* kako bi smo implementirali iterator *java.util.Iterator*.

```
public java.util.Iterator<E> iterator() {
    return new InorderIterator();
  }

  private class InorderIterator implements java.util.Iterator<E> {
```

```
    // Store the elements in a list
    private java.util.ArrayList<E> list =
    new java.util.ArrayList<E>();
    private int current = 0; // Point to the current element in list

    public InorderIterator() {
    inorder(); // Traverse binary tree and store elements in list
    }

    private void inorder() {
    inorder(root);
    }

    private void inorder(TreeNode<E> root) {
    if (root == null)return;
    inorder(root.left);
    list.add(root.element);
    inorder(root.right);
    }

    public boolean hasNext() {
    if (current < list.size())
        return true;

    return false;
    }

    public E next() {
    return list.get(current++);
    }

    public void remove() {
    delete(list.get(current)); // Delete the current element
    list.clear(); // Clear the list
    inorder(); // Rebuild the list
    }
  }
```

Metod *iterator* jednostavno vraća instancu klase *InorderIterator*.

Konstruktor *InorderIterator* poziva *inorder* metod. Metod *inorder(root)* prebacuje sve elemente iz stabla u listu *list*. Elementi stabla se pritom obilaze u inorder poretku.

Jednom kad je *Iterator* objekat kreiran, njegova vrednost *current* se inicijalizuje na 0, što znači da pokazuje na prvi element u listi. Pozivanjem metoda *next()* vraća se trenutni element a *current* se pomera da pokazuje na sledeći element u listi.

Metod *hasNext()* proverava da li je *current* još uvek u opsegu liste *list*. Metod *remove()* briše trenutni element iz stabla. Nakon toga se kreira nova lista. Primetimo da se pri tom vrednost pokazivača *current* nije promenila.

UPOTREBA ITERATORA

Implementacija iteratora nije baš efikasna. Svaki put kada se u toku kretanja iterator obriše neki element, vrši se ponovno kreiranje cele liste. Bolje je korisiti opciju delete klase BST

U narednom listingu koji pakuje niz stringova u binarno stablo pretrage i prikazuje sve stringove velikim slovima.

```
public class TestBSTWithIterator {
  public static void main(String[] args) {
    BST<String> tree = new BST<String>();
    tree.insert("George");
    tree.insert("Michael");
    tree.insert("Tom");
    tree.insert("Adam");
    tree.insert("Jones");
    tree.insert("Peter");
    tree.insert("Daniel");

    for (String s: tree)
    System.out.print(s.toUpperCase() + " ");
  }
}
```

Implementacija iteratora nije baš efikasna. Svaki put kada se u toku kretanja iterator obriše neki element, vrši se ponovno kreiranje cele liste. Stoga je poželjnije koristiti *delete* metod klase *BST* za brisanje elementa. Da bi sprečili da se kod binarnog stabla koristi metod *remove* iteratora, treba jednostavno implementirati *remove* metod na sledeći način:

```
public void remove() {
 throw new UnsupportedOperationException
 ("Removing an element from the iterator is not supported");
}
```

Nakon što smo proglasi metod *remove* za nedozvoljen metod iteratora, možemo da implementiramo iterator mnogo efikasnije bez potrebe da kreiramo listu za elemente stabla.

Moguće je koristiti stek za skladištenje elemenata, pri čemu je element na vrhu steka onaj koji je upravo dobijen metodom *next()*.

Ukoliko je stablo dobro izbalansirano (o čemu će biti više reči u narednoj lekciji, maksimalna veličina steka će biti O(logn).

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Koji metod je definisan u *java.lang.Iterable<E>* interfejsu?

Zadatak 2. Ako se ukloni **extends Iterable<E>** sa linije 1 iz *Tree* klase. Da li će *TestBSTWithIterator* program i dalje moći da se kompajlira bez grešaka?

Zadatak 3. Šta je prednost implementiranja interfejsa *Iterable<E>*?

Zadatak 4.Dodati sledeći metod u **BST** klasu koji vraća iterator za obilaženje stabla u preorder redosledu.

```
/** Returns an iterator for traversing the elements in preorder */
java.util.Iterator<E> preorderIterator()
```

Zadatak 5. Interfejs *java.util.Iterator* definiše **forward** iterator. Postoji i *java.util.ListIterator* interfejs koji definiše iterator koji radi u oba smera (bidirectional). Definisati dvosmerni iterator za BST klasu.

Poglavlje 9

Studija slučaja: Kompresija podataka

UVOD U HAFMANOVO STABLO KODIRANJA

Ideja Hafmanovog kodiranja je da se koriste niži bitovi (bitovi manje veličine) da se šifruju karakteri koji se često javljaju u tekstu, a veći bitovi za karaktere koji se ređe javljaju u tekstu

Kompresija podataka je uobičajen zadatak u informatici. Postoji puno različitih tehnika koje se koriste za kompresiju podataka.

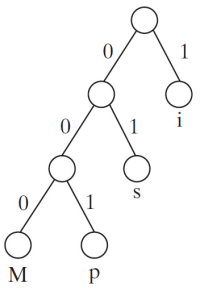
U ASCII kodiranju, svaki karakter se kodira (šifrira) sa 8 bita. Ukoliko se tekst sastoji iz 100 karaktera, neophodno je 800 bita da se predstavi tekst.

Ideja Hafmanovog kodiranja je da se koriste niži bitovi (bitovi manje veličine) da se šifriraju karakteri koji se često javljaju u tekstu, a veći bitovi za karaktere koji se ređe javljaju u tekstu, i da se na taj način smanji ukupna veličina tekstualne datoteke.

Kod Hafmanovog kodiranja, kodovi za karaktere se kostruišu na osnovu učestalosti javljanja karaktera u tekstu, korišćenjem binarnog stabla koje se naziva Hafmanovo stablo kodiranja (Huffman coding tree).

Pretpostavimo da je tekst koji želimo kodirati *Mississippi*. Hafmanovo stablo za ovaj tekst je prikazano na Slici-1a.

Levoj i desnoj grani čvora stabla se dodeljuju vrednosti *0* i *1*, respektivno. Svaki karakter je list stabla. Kod za karaktere se sastoji iz vrednosti grana na putu od korena do odgovarajućeg lista, kao što je prikazano na Slici 1b. Pošto se karakteri *i* i *s* javljaju više nego karakteri *M* i *p* u tekstu, njima su dodeljeni kraći kodovi.



(a) Hafmanovo stablo kodiranja

Znak	Kod	Frekvencija
M	000	1
p	001	2
s	01	4
i	1	4

(b) Tabele kodiranih karaktera

Slika 9.1 Kodovi karaktera se formiraju na osnovu učestalosti pojavljivanja karaktera u tekstu, korišćenjem stabla kodiranja.

Tabela sa vrednostima učestalosti (frekvencije) karaktera u tekstu je takođe prikazana na Slici-1b.

Isto stablo kodiranja se koristi i za dekodiranje sekvence bitova u karaktere. Da bi to uradili, počinje se sa prvim bitom u sekvenci, i prema vrednosti bita, određuje se da li da se ide levom ili desnom granom počevši od korena stabla. Zatim se posmatra sledeći bit i nastavlja da se ide na dole levom ili desnom granom, u zavisnosti od vrednosti bita. Kada se dođe do lista, karakter je pronađen. Sledeći bit u toku (stream) je prvi bit sledećeg karaktera.

Na primer, ulazni tok bitova *011001* je dekodiran kao *sip*, gde *01* predstavlja *s*, *1* predstavlja *i*, a *001* predstavlja *p*.

POSTUPAK GENERISANJA HAFMANOVOG STABLA

Levoj i desnoj grani čvora stabla se dodeljuju vrednosti 0 i 1, respektivno. Kod za karaktere se sastoji iz vrednosti grana na putu od korena do odgovarajućeg lista (gde se nalazi karakter)

Da bi se generisalo Hafmanovo stablo kodiranja (Huffman coding tree), treba koristiti sledeći algoritam:

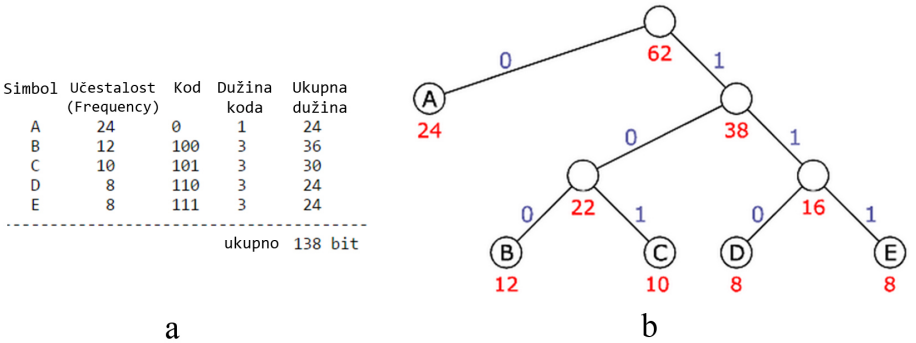
1. Započeti postupak sa šumom stabala. Svako stablo se sastoji iz samo jednog čvora za odgovarajući karakter. Težina svakog čvora je učastalost (frekvencija) pojavljivanja karaktera u tekstu.

2. Ponoviti naredni zahvat u cilju kombinovanja (spajanja) stabala sve dok u šumi ne ostane samo jedno stablo: Izabrati dva stabla sa najmanjim težinama, kreirati novi čvor i proglasiti ga za njihovog roditelja. Težina novog stabla se određuje kao suma težina podstabla.

3. Za svaki unutrašnji čvor stabla, dodeliti njegovoj levoj grani težinu *0*, a desnoj grani težinu *1*. Listovi stabla će predstavljati karaktere iz teksta.

Primer

Dat je dokument sa 5 simbola sa svojim frekvencijama pojavljivanja, Slika- 2a.



Simbol	Učestalost (Frequency)	Kod	Dužina koda	Ukupna dužina
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24
			ukupno	138 bit

Slika 9.2 a)Tipične težine karaktera za uzorak Hafmanovog koda b) Rezultujuće Hafmanovo stablo

Dva najređa simbola *E* i *D* su prvi povezani, zatim *C* i *D*.

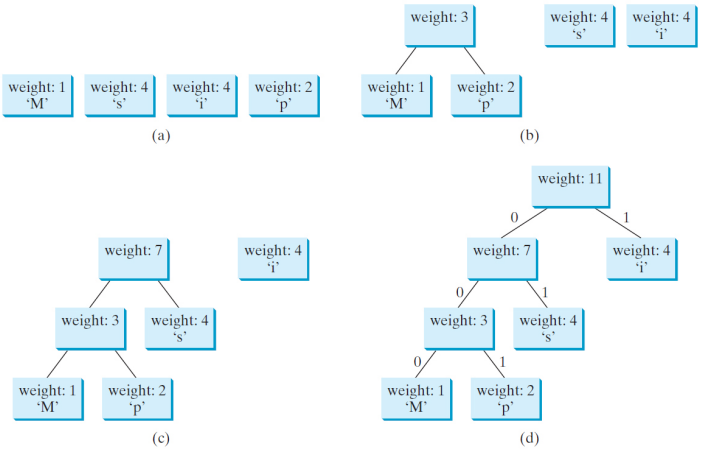
Novi roditelja čvorova imaju sada frekvencije *16* i *22*, respektivno i sažeti su u sledećem koraku.

Dobijeni čvor i preostali simbol *A* su podređeni *root* čvoru koji je stvoren u završnoj fazi. Rezultat je dat na Slici-2b.

PRIMER GENERISANJA HAFMANOVOG STABLA

Inicijalno, šuma se sastoji iz jediničnih stabala. Kombinovanjem stabala kreiraju se nova stabla, iznova i iznova. U jednom trenutku će ostati samo jedno stablo u sistemu – Hafmanovo stablo

U nastavku je primer kreiranja stabla codiranja za tekst *Mississippi*. Inicijalno, šuma se sastoji iz jediničnih stabala (jedan čvor u stablu), kao što je prikazano na Slici 1. Kombinovanjem stabala, kreiraju se nova stabla, iznova i iznova. Pritom se smanje broj stabala, i u jednom trenutku će ostati samo jedno stablo u sistemu. Ceo postupak je prikazan na Slici- 3.



Slika 9.3 Stablo Hafmanovog kodiranja je napravljeno uzastopnim spajanjem dva stabla, iz šume stabala sa najmanjim težinama.

Hafmanov algoritam spada u grupu pohlepnih algoritama (greedy algorithm). Kao što smo ranije spomenuli, pohlepni algortiam se veoma često koriti kod problema optimizacije. Algoritam pravi izbor koji je lokalno optimalan, u nadi da će takvo rešenje dovesti i do globalno-optimalnog rešenja.

U slučaju Hafmanovog stabla, algoritam uvek bira dva stabla sa najmanjim težinama i kreira novi čvor kao njihovog roditelja. Ovo intuitivno lokalno rešenje ustvari vodi i do konačnog optimalnog rešenja za konstruisanje Hafmanovog stabla.

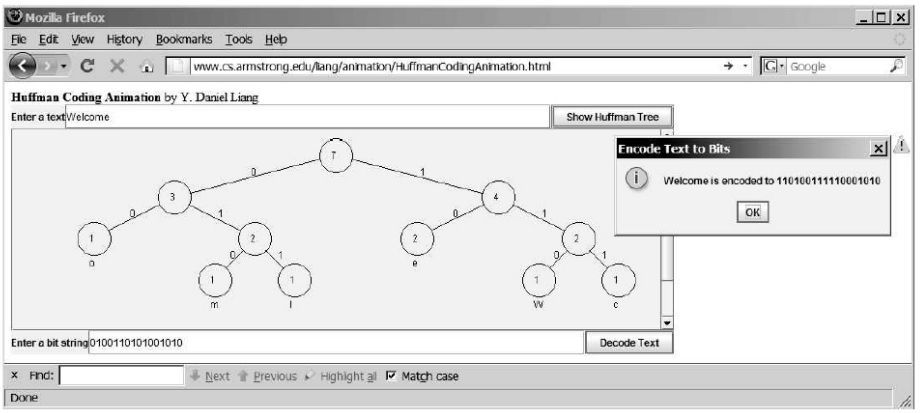
Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ANIMACIJA RADA HAFMANOVOG STABLA

Interaktivna animacija rada Hafmanovog stabla za bolje razumevanje postupka

Interaktivna aplikacija koja prikazuje Hafmanovo kodiranje se nalazi na sledećem linku:

www.cs.armstrong.edu/liang/animation/HuffmanCodingAnimation.html



Slika 9.4 Aplikacija omogućava prikaz Hafmanovog stabla, kao i postupak kodiranja i dekodiranja na osnovu tog stabla

IMPLEMENTACIJA HAFMANOVOG STABLA

Inicijalno se kreiraju jedinična stabla i ubacuju u hip strukturu. U svakoj iteraciji petlje, stabla najmanje težine se izbacuju iz hipa i spajaju u veće stablo, koje se zatim ponovo ubacuje u hip

U narednom listingu je program koji od korisnika zahteva da unese string, zatim na ekranu prikazuje tabelu učestalosti karaktera u stringu, i na kraju prikazuje Hafmanove kodove za svaki pojedinačni karakter.

```
import java.util.Scanner;

public class HuffmanCode {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a text: ");
    String text = input.nextLine();

    int[] counts = getCharacterFrequency(text); // Count frequency

    System.out.printf("%-15s%-15s%-15s%-15s\n",
    "ASCII Code", "Character", "Frequency", "Code");

    Tree tree = getHuffmanTree(counts); // Create a Huffman tree
```

```
  String[] codes = getCode(tree.root); // Get codes

  for (int i = 0; i < codes.length; i++)
  if (counts[i] != 0) // (char)i is not in text if counts[i] is 0
      System.out.printf("%-15d%-15s%-15d%-15s\n",
      i, (char)i + "", counts[i], codes[i]);
}

/** Get Huffman codes for the characters
 * This method is called once after a Huffman tree is built
 */
public static String[] getCode(Tree.Node root) {
  if (root == null) return null;
  String[] codes = new String[2 * 128];
  assignCode(root, codes);
  return codes;
}

/* Recursively get codes to the leaf node */
private static void assignCode(Tree.Node root, String[] codes) {
  if (root.left != null) {
  root.left.code = root.code + "0";
  assignCode(root.left, codes);

  root.right.code = root.code + "1";
  assignCode(root.right, codes);
  }
  else {
  codes[(int)root.element] = root.code;
  }
}

/** Get a Huffman tree from the codes */
public static Tree getHuffmanTree(int[] counts) {
  // Create a heap to hold trees
  Heap<Tree> heap = new Heap<Tree>(); // Defined in Listing 24.10
  for (int i = 0; i < counts.length; i++) {
  if (counts[i] > 0)
      heap.add(new Tree(counts[i], (char)i)); // A leaf node tree
  }

  while (heap.getSize() > 1) {
  Tree t1 = heap.remove(); // Remove the smallest weight tree
  Tree t2 = heap.remove(); // Remove the next smallest weight
  heap.add(new Tree(t1, t2)); // Combine two trees
  }

  return heap.remove(); // The final tree
}

/** Get the frequency of the characters */
public static int[] getCharacterFrequency(String text) {
  int[] counts = new int[256]; // 256 ASCII characters
```

```
    for (int i = 0; i < text.length(); i++)
    counts[(int)text.charAt(i)]++; // Count the character in text

    return counts;
  }

  /** Define a Huffman coding tree */
  public static class Tree implements Comparable<Tree> {
    Node root; // The root of the tree

    /** Create a tree with two subtrees */
    public Tree(Tree t1, Tree t2) {
    root = new Node();
    root.left = t1.root;
    root.right = t2.root;
    root.weight = t1.root.weight + t2.root.weight;
    }

    /** Create a tree containing a leaf node */
    public Tree(int weight, char element) {
    root = new Node(weight, element);
    }

    @Override /** Compare trees based on their weights */
    public int compareTo(Tree t) {
    if (root.weight < t.root.weight) // Purposely reverse the order
        return 1;
    else if (root.weight == t.root.weight)
        return 0;
    else
        return -1;
    }

    public class Node {
    char element; // Stores the character for a leaf node
    int weight; // weight of the subtree rooted at this node
    Node left; // Reference to the left subtree
    Node right; // Reference to the right subtree
    String code = ""; // The code of this node from the root

    /** Create an empty node */
    public Node() {
    }

    public Node(int weight, char element) {
        this.weight = weight;
        this.element = element;
    }
    }
  }
}
```

ANALIZA METODA KLASE HUFFMANCODE

Metod getHuffmanTree vraća kao rezultat Hafmanovo stablo kodiranja. Metod assignCode dodeljuje kodove svakom pojedinačnom karakteru u stablu

Program prvo od korisnika traži da unese tekst, u obliku stringa, a zatim broji učestalost karaktera u tekstu. Metod *getCharacterFrequency* kreira niz *counts* koji pamti pojavljivanje svakog od 256 *ASCII* karaktera u tekstu. Ukoliko se karakter javlja u tekstu, njegov odgovarajući brojčanik (*count*) se uvećava za jedan. Program vrši Hafmanovo kodiranje prema vrednostima u nizu *counts*. Stablo se sastoji iz povezane liste čvorova, pri čemu je za čvor definisana klasa *Node*. Svaki čvor se sastoji iz sledećih svojstava: *element* (čuva karakter), *weight* (čuva težinu podstabla ispod čvora), *left* (veza/link ka njegovom levom podstablu), *right* (veza/link ka njegovom desnom podstablu), i *code* (čuva Hafmanov kod za karakter).

Klasa *Tree* sadrži pokazivač na koren. Od korena je moguće pristupiti bilo kom čvoru u stablu.

Stabla se upoređuju na osnovu njihovih težina (*weights*). Redosled poređenja je namerno preokrenut, kako bi stablo najmanje težine bilo prvo obrisano iz hipa stabala.

Metod *getHuffmanTree* vraća kao rezultat Hafmanovo stablo kodiranja. Inicijalno, kreiraju se jedinična stabla i ubacuju u hip strukturu (*heap*). U svakoj iteraciji ***while*** petlje, stabla najmanje težine se izbacuju iz hipa i spajaju u veće stablo, koje se zatim ponovo ubacuje u hip. Ovaj proces se ponavlja sve dok u hipu (hip u ovom primeru služi da predstavi šumu stabala) ne ostane samo jedno stablo koje je ustvari Hafmanovo stablo kodiranja za uneti tekst.

Metod *assignCode* dodeljuje kodove svakom pojedinačnom karakteru u stablu. Metod *getCode* vraća Hafmanov kod za sve karaktere teksta, koji se nalaze u listovima stabla. Element *codes[i]* sadrži kod za karakter *(char)i*, gde *i*ide od *0*do *255*. Primetimo da je *codes[i]*jednak *null* ako karakter *(char)i* nije deo teksta.

ZADACI ZA SAMOSTALNI RAD

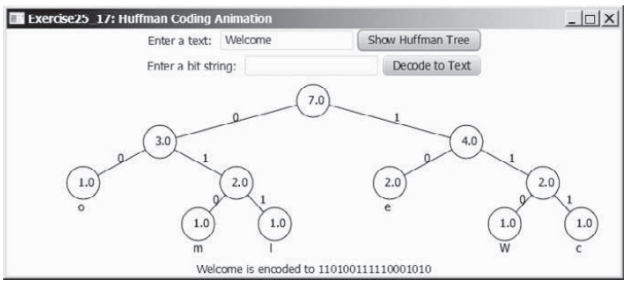
Uraditi samostalno sledeće zadatke

Zadatak 1. Napisati program koji traži od korisnika da unese ime tekstualnog fajla, a zatim prikazuje tabelu frekvencija karaktera iz fajla i Hafmanov kod za svaki karakter.

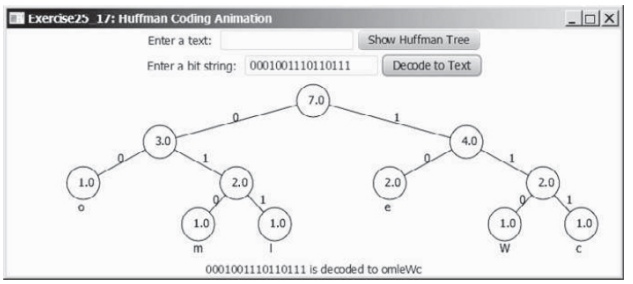
Zadatak 2. Napisati program koji traži unos teksta, a zatim ispisuje Hafmanovo kodiranje unetog teksta, kao što je prikayano na **Slika-5a**. Prikazati težinu podstabla u svakom čvoru. Za svaki list prikazati karakter koji mu odgovara. Prikazati kodirane bitove koji odgovaraju tekstu unetom u label. Kada korisnik pritisne dugme za dekodiranje treba prizati tekst koji je unet, kao što je prikazano na **Slika-5b.**

Zadatak 3. Napisati program koji uzima jedan fajl i vrši kompresiju pomoću Hafmanovog kodiranja i tako dobija drugi fajl. Koristiti **ObjectOutputStream** i **BitOutputStream** za pisanje Hafmanovog koda u fajl.

(a)



(b)



Slika 9.5 (a) Stablo koje odgovara Hafmanovom kodiranju unetog teksta kao i sam kod teksta; (b) Dekodiranje binarnog koda u tekst na osnovu stabla.

Zadatak 4. U prethodnom zadatku smo dobili kompresovan fajl koji sadrži Hafmanov kod. Napisati program koji dekompresuje takav fajl u običan tekstualni.

Zaključak

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

Stablo je rekurzivna aciklična struktura podataka koja se sastoji od čvorova koji su povezani granama. Svaki čvor ima najviše jednog roditelja, i nula ili više direktnih potomaka (dece).

Osnovni termini kod stabla su: čvor, grana, roditelj, potomak, faktor grananja, koren, list, unutrašnji čvor, putanja, dužina putanje, dubina čvora i visina stabla.

Stablo može biti implementirano korišćenjem povezanih struktura podataka (ulančana reprezentacija) i korišćenjem niza. Nizovi su efikasni samo kod potpunih stabala. Binarno stablo je ono stablo kod koga čvor može imati najviše dva potomka.

Binarno stablo pretraživanja (BST) zadovoljava ograničenje da je ključ u svakom čvoru veći od ključeva u svim čvorovima u levom podstablu i manji od ključeva u svim čvorovima koju su u desnom podstablu tog čvora. Svaki čvor binarnog stabla sadrži vrednost (informacioni deo) i dva linka (pokazivača) pod nazivom *left* (levi) i *right* (desni) koji referenciraju levog i desnog potomka čvora.

Kod pretraživanja elemenata u BST stablu, neophodno je započeti pretragu od korena, i krenuti naniže ka listovima dok se vrednost nekog čvora ne poklopi sa ključem pretrage ili dok se ne dostigne jedan od listova stabla. Da bi ste ubacili novi element u binarno stablo neophodno je da locirate poziciju gde ga treba ubaciti. Ukoliko je novi element manji od roditelja čvor za novi element se ubacuje u levu granu, odnosno, ako je element veći od roditelja onda se novi čvor ubacuje sa desne strane. Da bi obrisali element iz BST stabla, prvo ga treba locirati u stablu a zatim uzeti u obzir dva slučaja: da li čvor ima ili nema svog levog potomka, pre nego se obriše iz stabla.

Postoje dva različita obilaska stabla: Obilazak po dubini (depth-first search- DFS) i Obilazak po širini (breadth-first search- BFS). Postoje tri tipa DFS obilaska stabla: preorder, postorder i inorder.

Ideja Hafmanovog kodiranja je da se koriste niži bitovi (bitovi manje veličine) da se šifruju karakteri koji se često javljaju u tekstu, a veći bitovi za karaktere koji se ređe javljaju u tekstu.

REFERENCE

Korišćena literatura

[1] D. Liang, Introduction to Java Programming, Comprehensive Version, deseto izdanje, Prentice Hall, 2014

[2] R. Sedgewick, K.Wayne, Algorithms, 4. Izdanje, Pearson Education, 2011.

[3] M.A. Weiss, Data Structures and Problem Solving Using Java, treće izdanje, Addison Wesley, 2005.

[4] https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

[5] https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm

[6] http://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/