Stabla

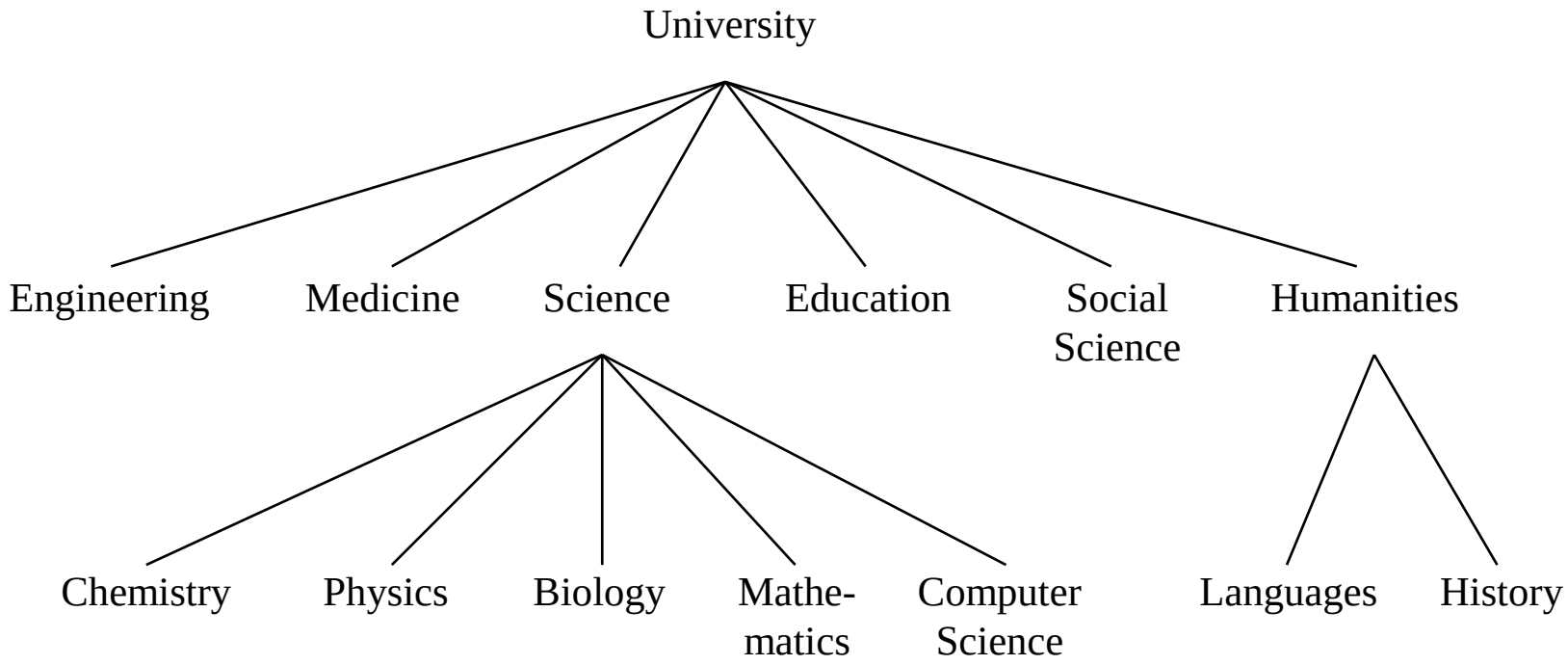
Strukture podataka i algoritmi 2

Stablo

- Stablo je struktura podatka kojom možemo predstaviti hijerarhijsko uređenje elemenata nekog skupa
  - Stablo je hijerarhijski uređena kolekcija elemenata
- Osnova svake hijerarhije je relacija “nadređeni – podređeni” ili “otac – sin” pri čemu podređeni (sin) može imati tačno jednog nadređenog (oca)
- Stablo se sastoji od čvorova i grana koji spajaju te čvorove
  - Dva čvora A i B su povezana granom ukoliko je A otac B
  - Stablo ima jedinstven čvor koji se zove koren (engl. root) – čvor koji nema oca
  - Svi čvorovi osim korena imaju tačno jednog oca
  - Svi čvorovi mogu imati proizvoljan broj sinova

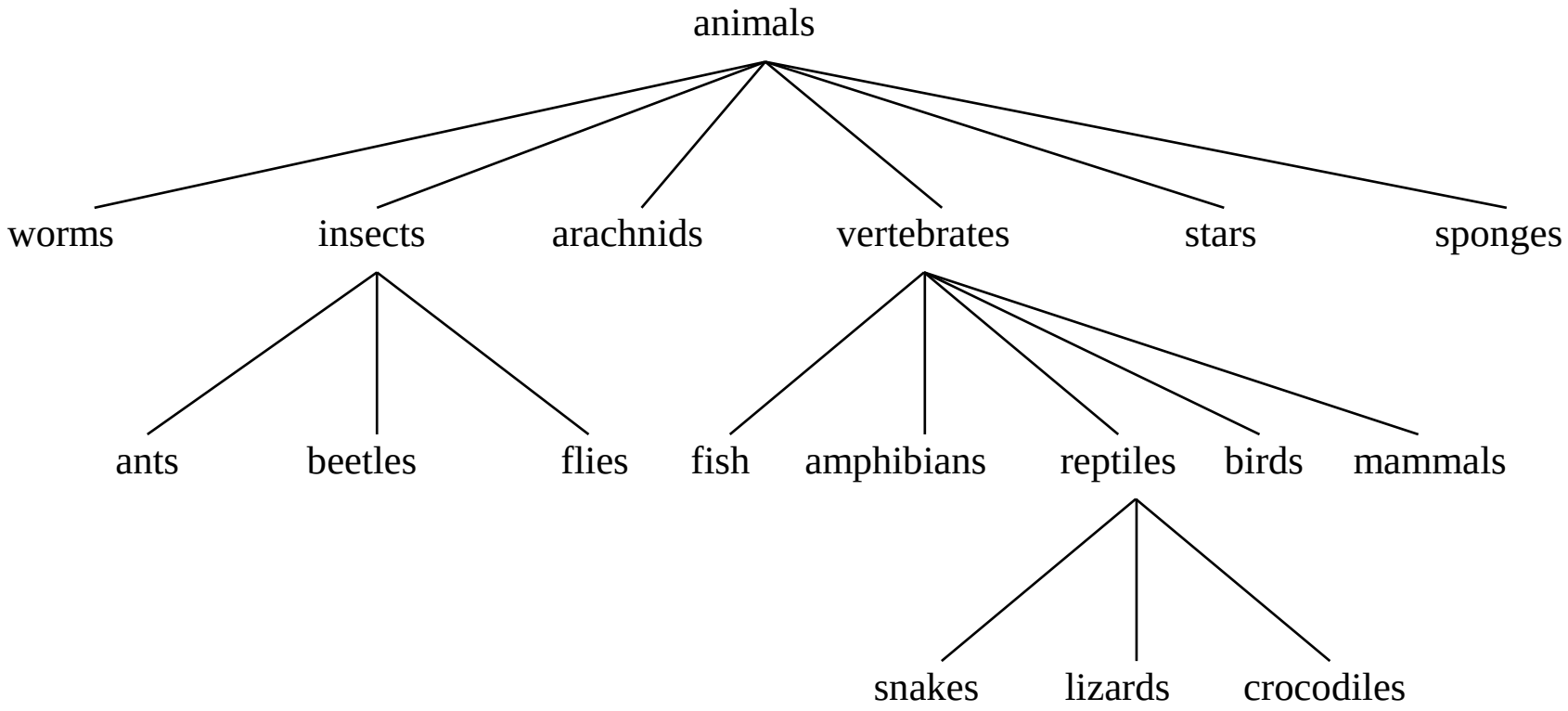
Primeri stabala

- Stablima možemo predstaviti hijerarhijsku organizaciju nekog sistema
  - npr. univerzitet se sastoji od fakulteta, fakultet od departmana, departman od katedri



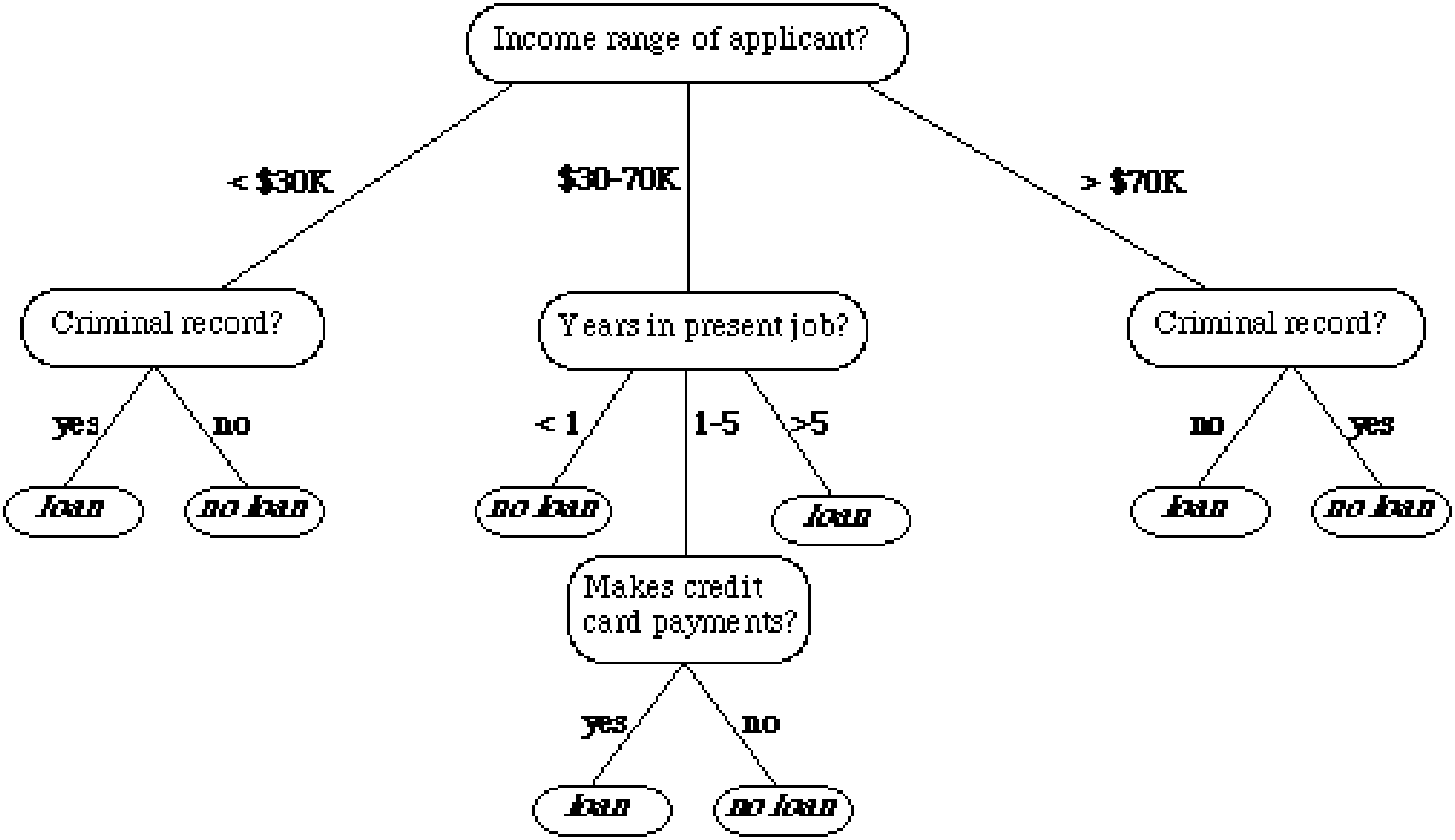
Primeri stabala

- Stablima možemo predstaviti taksonomije
  - Listovi stabla – konkretni objekti
  - Unutrašnji čvorovi stabla – grupe ili klase objekata



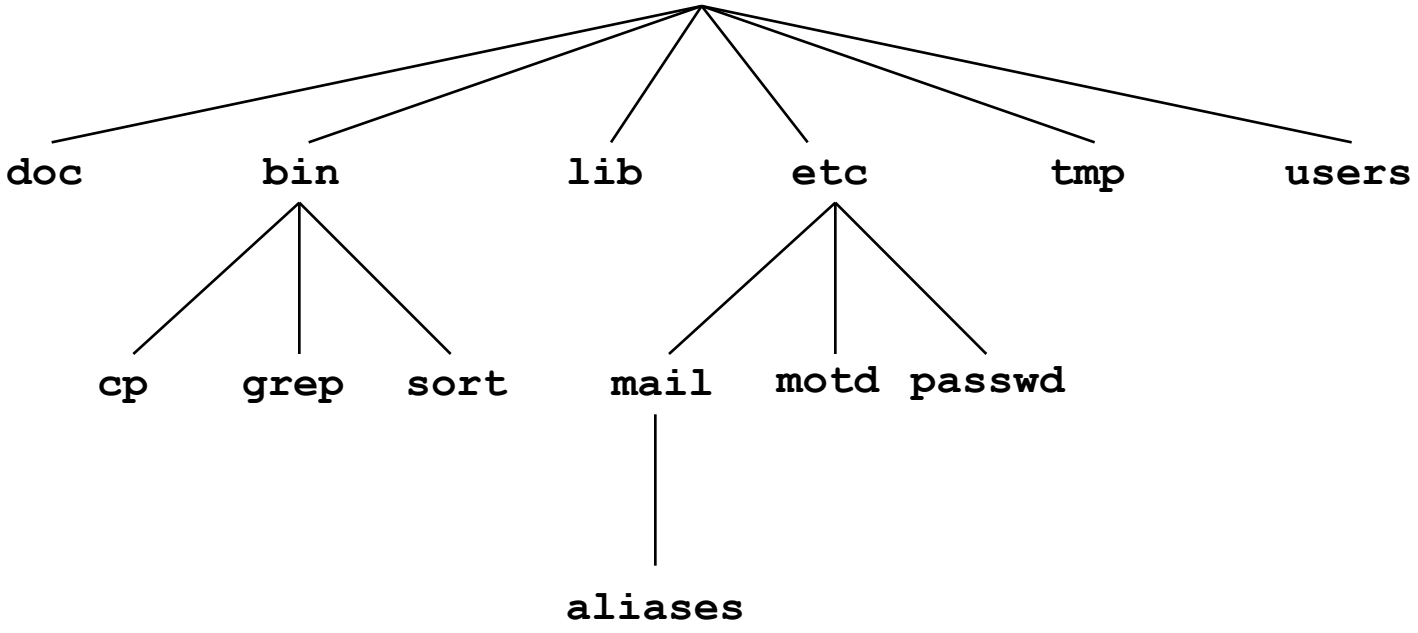
Primeri stabala

- Stabla odlučivanja (engl. *decision trees*) u veštačkoj inteligenciji i mašinskom učenju



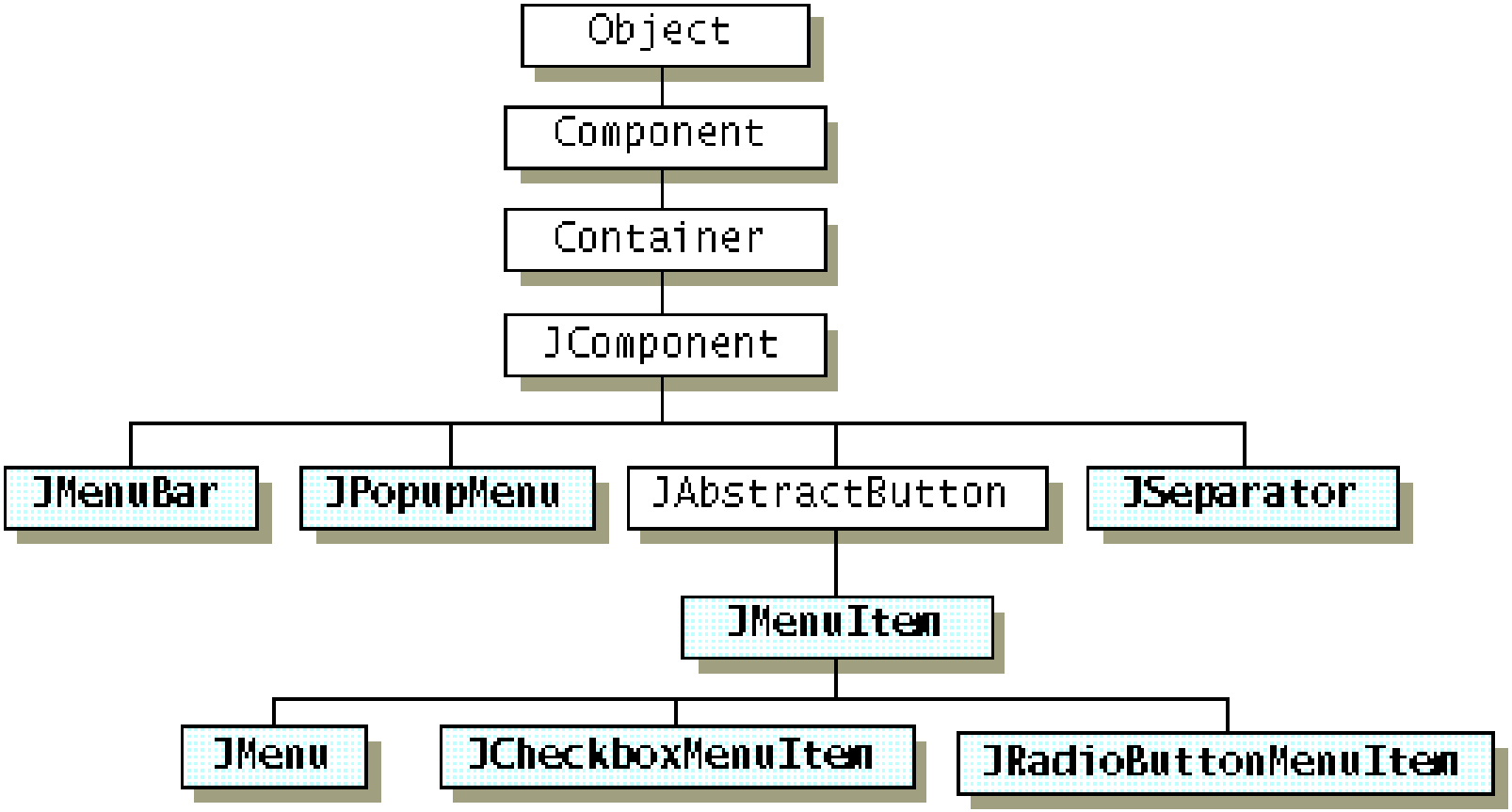
Primeri stabala

- Fajl sistem je stablo
  - Listovi stabla – fajlovi
  - Unutrašnji čvorovi stabla – direktorijumi



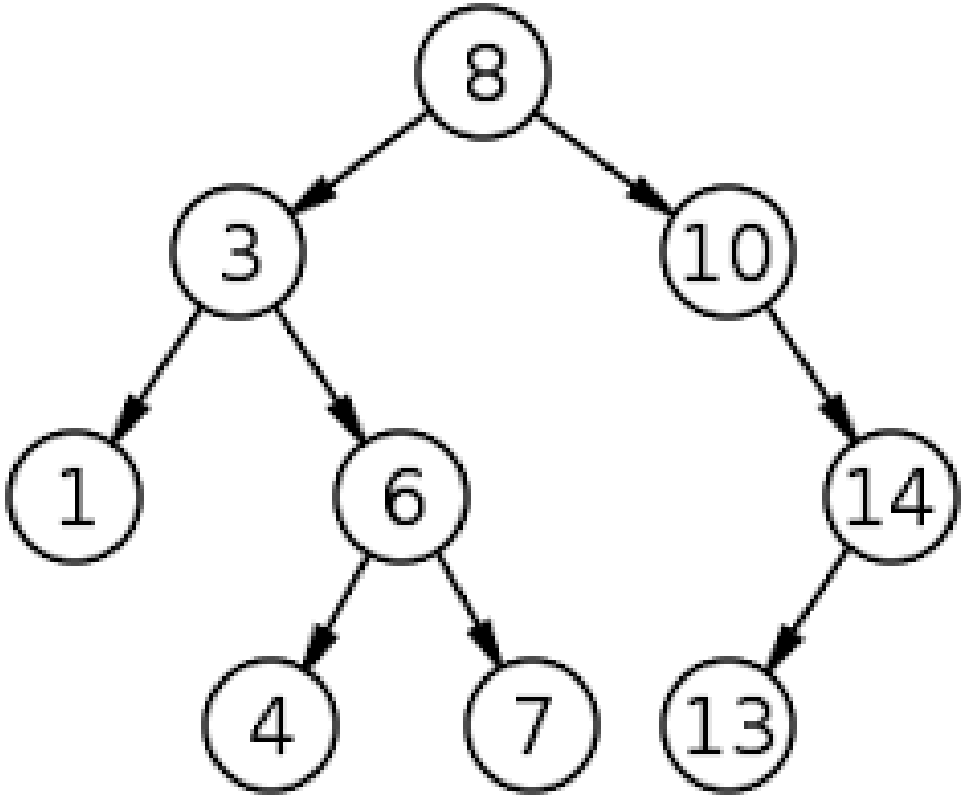
Primeri stabala

- Stabla nasleđivanja (inheritance tree) OO programa
  - Čvorovi stabla su klase
  - Čvor A je roditelj čvora B ukoliko B nasleđuje A



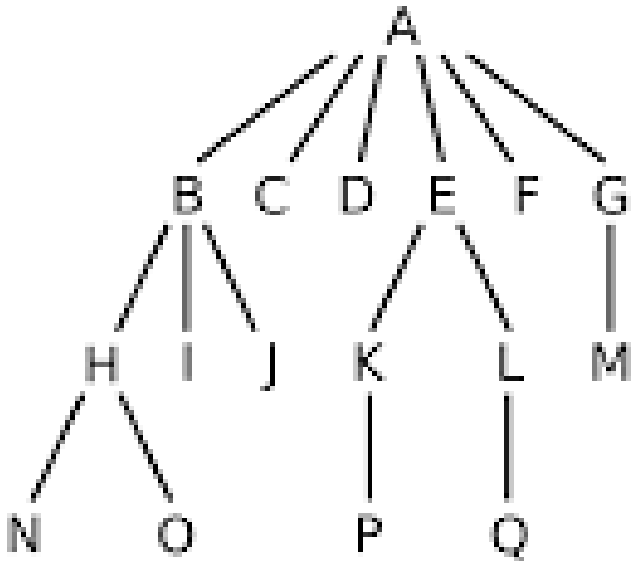
Binarno stablo

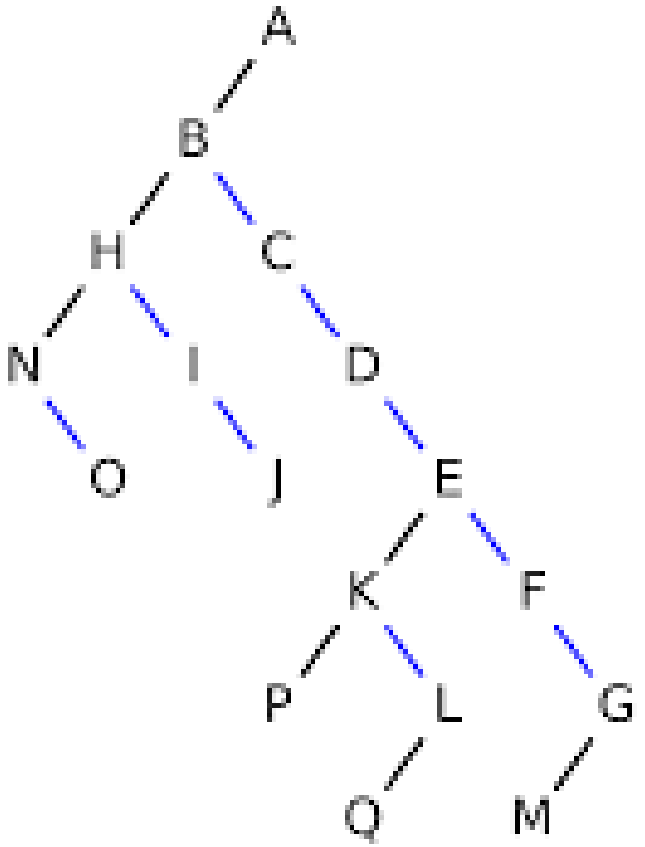
- Binarno stablo je stablo kod koga svaki čvor može imati najviše dvoje dece.



Binarno stablo

- Svako stablo možemo predstaviti binarnim stablom





left pokazivač – pokazivač na prvog sina

right pokazivač – pokazivač na sledeće dete moga oca

Reprezentacija binarnog stabla

- Već smo videli da binarno stablo možemo predstaviti nizom (kod realizacije prioritetne liste)
  - Korenski čvor – prvi element niza T, T[0]
  - Ako je čvor A na poziciji k u nizu T tada su njegovi sinovi na pozicijama 2*k + 1 i 2*k + 2 u nizu T.
  - Ako je čvor A na poziciji k u nizu T tada je njegov otac na poziciji (k – 1)/2
- Nedostatak ovakve reprezentacije je bespotrebno trošenje memorijskog prostora u situacijama kada je stablo **retko**
  - Neka se stablo dubine k sastoji od N čvorova
  - Za stablo dubine k maksimalni broj čvorova je $M = 1 + 2 + 4 + ... + 2^k$
  - $2M = 2 + 4 + 8 + \ldots + 2^k + 2^{k+1}$
  - $M = 2M – M = 2^{k+1} – 1$
  - **Stablo dubine k je retko ako je N << M**

Reprezentacija binarnog stabla

- Dinamička reprezentacija binarnog stabla:
  - Unutar svakog čvora stabla čuvamo pokazivače (reference) na levo i desno dete.
    - Opciono možemo čuvati i treći pokazivač – pokazivač na roditeljski čvor
  - Čuvamo referencu na korenski čvor kako bi svi čvorovi stabla bili dostižni.

```
class BTNode<T> {
    T info;
    BTNode<T> left, right;
    ...
}

class BinaryTree<T> {
    BTNode<T> root;
    ...
}
```

BTNode – klasa koja opisuje jedan čvor binarnog stabla

```
public class BTNode<T extends Comparable<T>> implements Comparable<BTNode<T>>
{
	private T info;
	private BTNode<T> left, right;

	public BTNode(T info) {
		this.info = info;
	}

	public BTNode(T info, BTNode<T> left, BTNode<T> right) {
		this.info = info;
		this.left = left;
		this.right = right;
	}

	public BTNode<T> getLeft()             { return left;             }
	public void setLeft(BTNode<T> left)    { this.left = left;        }
	public BTNode<T> getRight()            { return right;            }
	public void setRight(BTNode<T> right)  { this.right = right;     }
	public T getInfo()                     { return info;             }
	public void setInfo(T info)            { this.info = info;        }
	public String toString()               { return info.toString(); }

	public int compareTo(BTNode<T> otherNode) {
		return info.compareTo(otherNode.info);
	}
}
```

BinaryTree – klasa koja opisuje binarno stablo

```
public class BinaryTree<T extends Comparable<T>> {
	private BTNode<T> root = null;

	public void setRoot(BTNode<T> root) {
		this.root = root;
	}

	public boolean isEmpty() {
		return root == null;
	}

	public BTNode<T> getRoot() {
		return root;
	}

	...
}
```

Konstruisanje i modifikovanje stabla

- Opšte binarno stablo (binarno stablo koje nema neke specijalne osobine) konstruišemo tako što
  - kreiramo čvorove stabla i eksplicitno postavljamo veze između čvorova koristeći konstruktor klase BTNode i/ili setLeft i setRight metode
  - jedan element stabla proglasimo za korenski

- Stablo modifikujemo tako što menjamo vrednosti *left* i *right* referenci čvorova stabla i/ili vrednost reference na korenski element.
  - Metode BTNode.setLeft, BTNode.setRight i BinaryTree.setRoot

Primer konstrukcije binarnog stabla

```
public class BinaryTreeExample {
	public static void main(String[] args) {
		BTNode<String> mika = new BTNode<String>("mika");
		BTNode<String> zika = new BTNode<String>("zika");
		BTNode<String> pera = new BTNode<String>("pera");
		BTNode<String> mara = new BTNode<String>("mara");
		BTNode<String> dara = new BTNode<String>("dara");
		BTNode<String> sara = new BTNode<String>("sara");

		BinaryTree<String> tree = new BinaryTree<String>();
		tree.setRoot(mika);
		mika.setLeft(zika);
		mika.setRight(pera);
		zika.setRight(sara);
		pera.setLeft(mara);
		pera.setRight(dara);
	}
}
```

Primer konstrukcije binarnog stabla

```
BinaryTree<String> tree = new BinaryTree<String>();
tree.setRoot(
        new BTNode<String>("mika",
                new BTNode<String>("zika",
                        null,
                        new BTNode<String>("sara",
                                null,
                                null
                        )
                ),
                new BTNode<String>("pera",
                        new BTNode<String>("mara",
                                null,
                                null),
                        new BTNode<String>("dara",
                                null,
                                null
                        )
                )
        )
);
```

Karakteristike binarnog stabla

- Dve bazične strukturne karakteristike binarnog stabla su njegova
  - veličina – broj čvorova u stablu
  - dubina
- Za svaki čvor binarnog stabla postoji tačno jedan put od čvora ka korenu.
- Dubina binarnog stabla je dužina najvećeg puta od proizvoljnog čvora stabla do korenskog elementa.
  - Ako stablo ima samo jedan element njegova dubina je 0
  - Dubina praznog stabla je jednaka -1.

```
public class BinaryTree<T> {
	private BTNode<T> root = null;
	...

	public int getSize() {
		return root == null ? 0 : getSize(root);
	}

	private int getSize(BTNode<T> current) {
		int leftSize = 0;
		BTNode<T> left = current.getLeft();
		if (left != null)
			leftSize = getSize(left);

		int rightSize = 0;
		BTNode<T> right = current.getRight();
		if (right != null)
			rightSize = getSize(right);

		return 1 + leftSize + rightSize;
	}
}
```

```
public int getDepth() {
     return root == null ? -1 : getDepth(root);
}

private int getDepth(BTNode<T> current) {
     int leftDepth = -1;
     BTNode<T> left = current.getLeft();
     if (left != null)
          leftDepth = getDepth(left);

     int rightDepth = -1;
     BTNode<T> right = current.getRight();
     if (right != null)
          rightDepth = getDepth(right);

     if (leftDepth > rightDepth)
          return 1 + leftDepth;
     else
          return 1 + rightDepth;
}
```

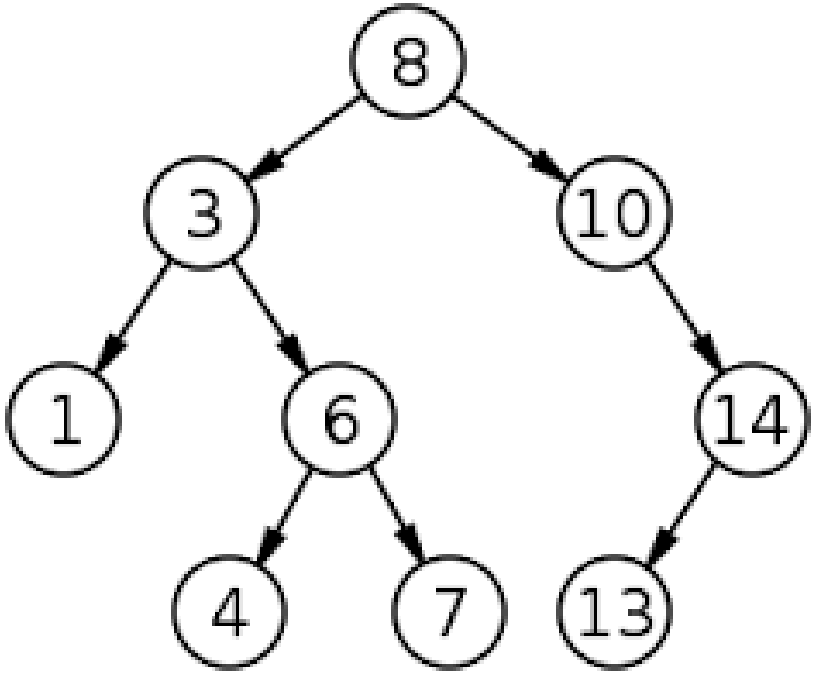
Pretraživanje stabla

- Svaki čvor binarnog stabla sadrži neku informaciju.
- Pretraživanje stabla: da li u binarnom stablu postoji čvor koji sadrži informaciju *x*?
- Opšte binarno stablo možemo pretraživati po dve strategije
  - **Pretraživanje u dubinu (*depth-first search, DFS*)**
    - da li je x sadržan u korenu?
    - ako nije traži x u levom podstablu koristeći isti postupak (rekurzivno)
    - ako x nije nađeno u levom podstablu pretraži desno podstablo koristeći isti postupak (rekurzivno)
  - **Pretraživanje u širinu (*breadth-first search, BFS*)**
    - pretraživanje binarnog stabla po nivoima

DFS

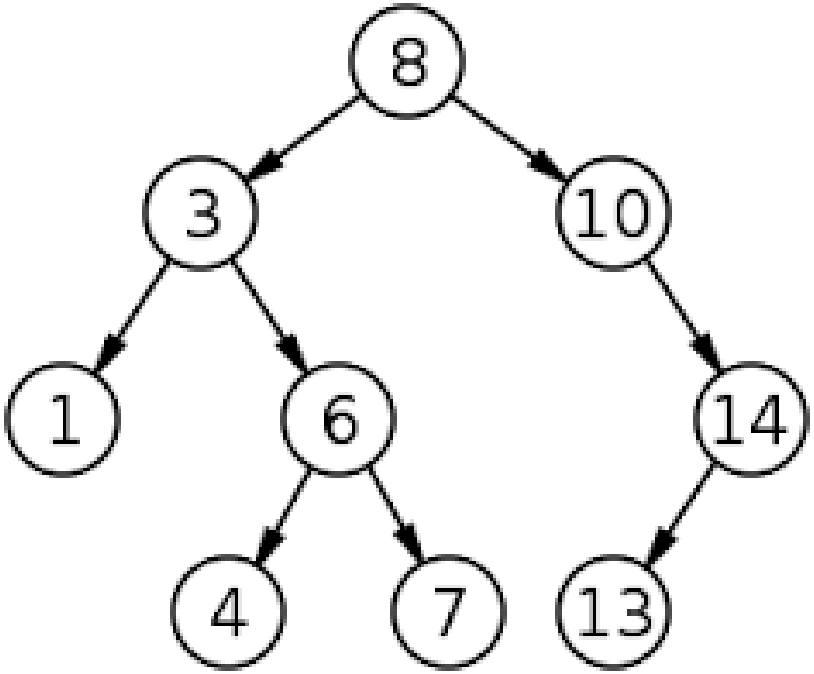
```
public BTNode<T> dfs(T info) {
     return root != null ? dfs(root, info) : null;
}

private BTNode<T> dfs(BTNode<T> current, T info) {
     if (current.getInfo().equals(info)) {
          return current;
     }

     BTNode<T> left = current.getLeft();
     if (left != null) {
          BTNode<T> n = dfs(left, info);
          if (n != null) return n;
     }

     BTNode<T> right = current.getRight();
     if (right != null) {
          BTNode<T> n = dfs(right, info);
          if (n != null) return n;
     }

     return null;
}
```

DFS



- Tražim 14
- DFS čvorove stabla obilazi sledećim redosledom: 8, 3, 1, 6, 4, 7, 10, 14

BFS



- Tražim 14
- BFS čvorove stabla obilazi sledećim redosledom: 8, 3, 10, 1, 6, 14

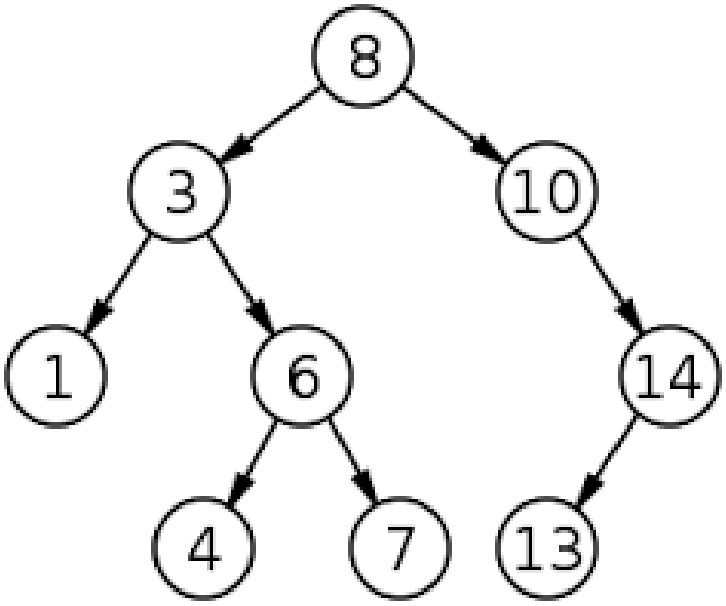
BFS

- Iskoristićemo red opsluživanja kako bi smo realizovali pretraživanje stabla po nivoima.
- Na početku dodamo korenski čvor u red opsluživanja
- Proces pretraživanja
  - f = ukloni prvi element iz reda opsluživanja
  - Da li je informacija koja se traži u f? **Ako jeste kraj**
  - Dodaj f.getLeft() na kraj reda ako f.getLeft() != null
  - Dodaj f.getRight() na kraj reda f.getRight() != null
  - Ponavljaj prethodne korake dok se red opsuživanja ne israzni

BFS

- Tražim 14
- Inicijalizacija Q = [8]

- f = 8, Q: [] → [3, 10]
- f = 3, Q: [10] → [10, 1, 6]
- f = 10, Q: [1, 6] → [1, 6, 14]
- f = 1, Q: [6, 14] → [6, 14]
- f = 6, Q: [14] → [14, 4, 7]
- f = 14 – našao



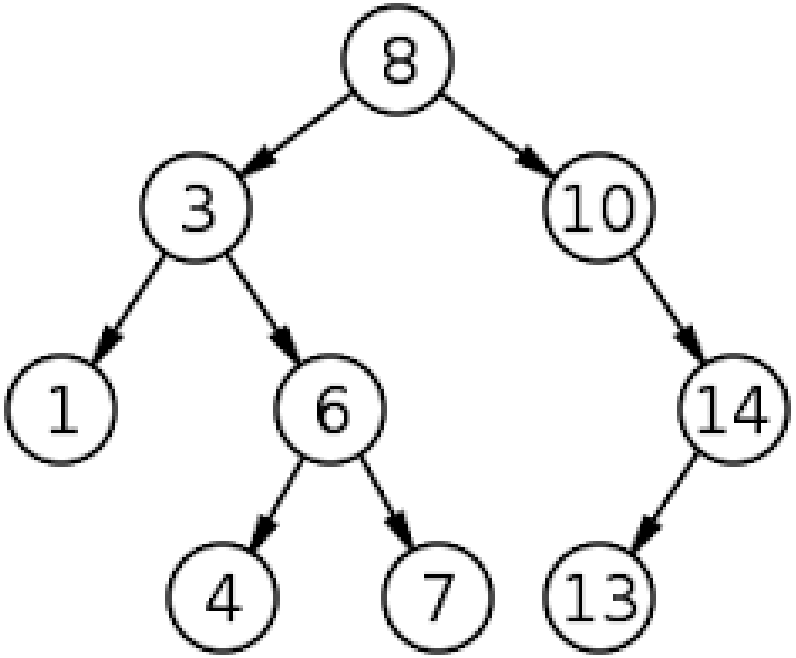
```java
public BTNode<T> bfs(T info) {
	if (root == null) return null;

	LinkedList<BTNode<T>> queue = new LinkedList<BTNode<T>>();
	queue.addLast(root);

	while (!queue.isEmpty()) {
		BTNode<T> f = queue.removeFirst();

		if (f.getInfo().equals(info))
			return f;

		BTNode<T> left = f.getLeft();
		if (left != null) {
			queue.addLast(left);
		}

		BTNode<T> right = f.getRight();
		if (right != null) {
			queue.addLast(right);
		}
	}

	return null;
}
```

DFS ponovo

- **Umesto reda opsluživanja iskoristimo stek**
- **Prvo dodajemo desnog, pa onda levog sina na stek**

  - Tražim 14
  - Inicijalizacija S = [**8**] (vrh steka je boldovan i plav)

  - f = 8, S:[] → [10, **3**]
  - f = 3, S:[10] → [10, 6, **1**]
  - f = 1, S:[10, 6] → [10, **6**]
  - f = 6, S:[10] → [10, 7, **4**]
  - f = 4, S:[10, 7] → [10, **7**]
  - f = 7, S:[10] → [**10**]
  - f = 10, S:[] → [**14**]
  - f = 14 – našao



```
public BTNode<T> dfsIter(T info) {
	if (root == null)
		return null;

	Stack<BTNode<T>> stack = new Stack<BTNode<T>>();
	stack.push(root);
	while (!stack.isEmpty()) {
		BTNode<T> f = stack.pop();
		if (f.getInfo().equals(info))
			return f;

		BTNode<T> right = f.getRight();
		if (right != null) {
			stack.push(right);
		}

		BTNode<T> left = f.getLeft();
		if (left != null) {
			stack.push(left);
		}
	}

	return null;
}
```

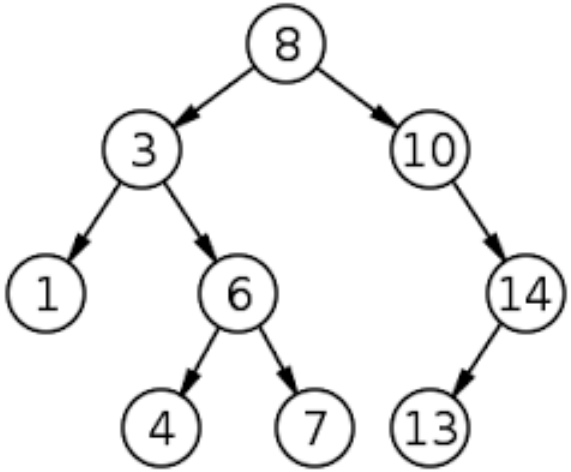
Iteriranje kroz stablo

- Iteriranje – obilazak svih elemenata kolekcije
  - Iteriramo kroz kolekciju ukoliko je potrebno da
    - nad svakim elementom kolekcije uradimo neku operaciju (npr. ažuriranje)
    - da selektujemo elemente kolekcije koji zadovoljavaju neki kriterijum
    - izvedemo neku zbirnu karakteristiku kolekcije (npr. prosek plata radnika)
- Kroz stablo možemo iterirati na više načina
  - pre-order (koren-levo-desno)
  - in-order (levo-koren-desno)
  - post-order (levo-desno-koren)
  - po nivoima

Pre-order

- Pre-order: poseti koren, obiđi levo podstablo, obiđi desno podstablo

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

private void preOrder(BTNode<T> current) {
	if (current != null) {
		System.out.println("Radim nesto nad: " + current);
		preOrder(current.getLeft());
		preOrder(current.getRight());
	}
}
```

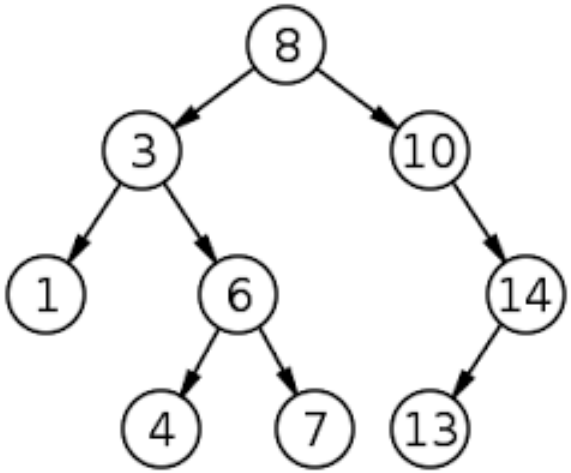


8, 3, 1, 6, 4, 7, 10, 14, 13

In-order

- In-order: obiđi levo podstablo, poseti koren, obiđi desno podstablo

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

private void inOrder(BTNode<T> current) {
	if (current != null) {
		inOrder(current.getLeft());
		System.out.println("Radim nesto nad: " + current);
		inOrder(current.getRight());
	}
}
```

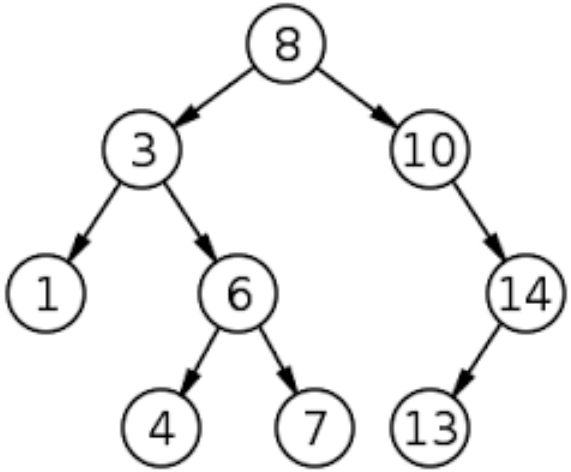


1, 3, 4, 6, 7, 8, 10, 13, 14

Post-order

- Post-order: obiđi levo podstablo, obiđi desno podstablo, poseti koren

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

private void postOrder(BTNode<T> current) {
	if (current != null) {
		postOrder(current.getLeft());
		postOrder(current.getRight());
		System.out.println("Radim nesto nad: " + current);
	}
}
```



1, 4, 7, 6, 3, 13, 14, 10, 8

Hafmanovo stablo kodiranja

- Primenu binarnih stabala (i prioritetne liste) ilustrovaćemo na primeru gramzivog algoritma za kompresiju teksta.
- Kod – bitstring, binarna sekvenca (niz 0 i 1) koja odgovara nekom simbolu (karakteru)
- Kodni sistem: 1-1 preslikavanje skupa simbola u skup kodova
  - Kodiranje: pretvaranje simbola u kod
  - Dekodiranje: pretvaranje koda u simbol
  - Kodovi fiksne dužine – svi simboli su predstavljeni kodovima iste dužine
    - ASCII kodni sistem – svakom karakteru odgovara bitstring dužine 8 → možemo predstaviti maksimalno 256 simbola
    - Prednost: jednostavno dekodiranje

Kompresovanje teksta

- Osnovna ideja kod kompresovanja teksta je:
  - Kreiramo novi kodni sistem uzimajući u obzir samo one karaktere koji se pojavljuju u tekstu
  - Pravimo kodni sistem sa kodovima **promenljive dužine**
  - **Simboli sa većom frekvencijom pojavljivanja imaju kod manje dužine u odnosu na simbole sa manjom frekvencijom**
  - Kodovi moraju biti **prefiksni** da bi dekompresija bila moguća
  - **Prefiksni kodni sistem: ni jedan kod nije prefiks nekog drugog koda**

Primer...

- Kodni sistem

  {a → 0, b → 1, c → 01, d → 11}

  je kodni sistem promenljive dužine, ali nije prefiksni

  - 010111 može značiti “ababbb” ali i “ccbb”, “ccd”, “abcd”, “abada”

- Kodni sistem

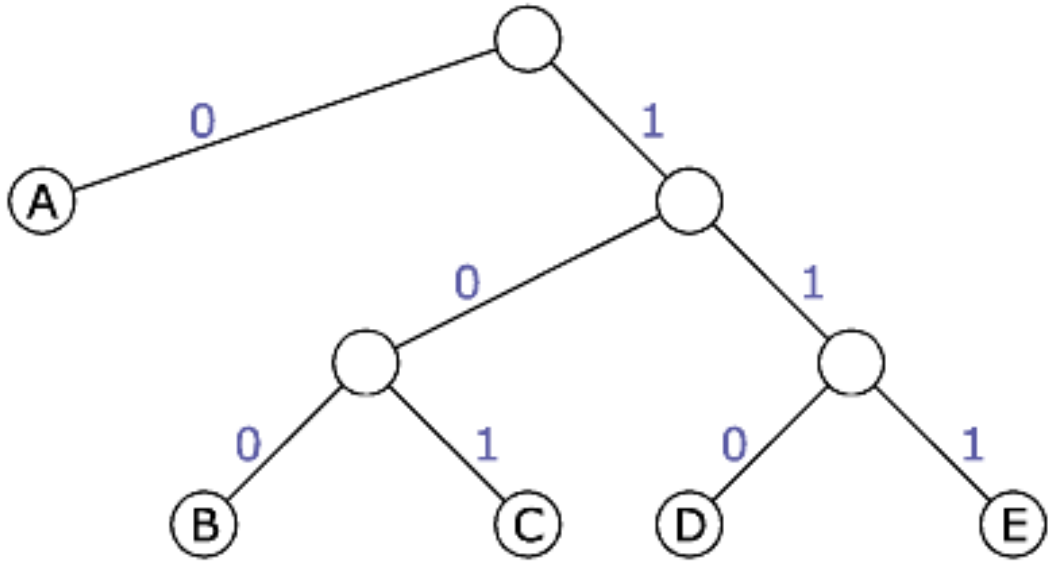
  {a → 0, b → 10, c → 110, d → 111}

  je kodni sistem promenljive dužine koji je prefiksni

  - 010111 se jednoznačno dekodira u “abd”

Prefiksni kodovi i binarna stabla

- Prefiksni kodni sistem se može predstaviti binarnim stablom
  - Listovi – simboli
  - Kod za simbol – put od korena ka simbolu (levo 0, desno 1)
  - {A → 0, B → 100, C → 101, D → 110, E → 111}

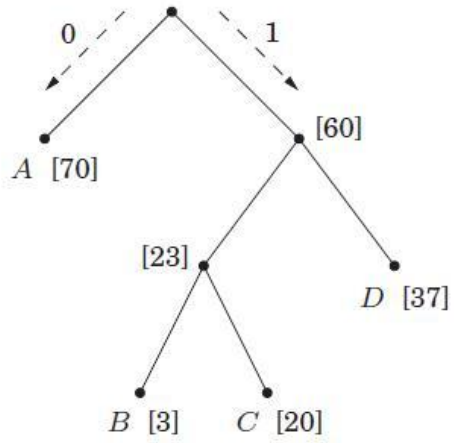


Kompresovanje teksta

- Minimizujemo $\sum_{i=1}^{n} f_i d_i$
  - S – skup karaktera koji se pojavljuju u tekstu
  - n - broj različitih karaktera u tekstu, kardinalnost skupa S
  - f_i – frekvencija i-tog karaktera iz S
  - d_i – dužina koda za i-ti karakter = **dubina odgovarajućeg lista u binarnom stablu kodiranja**
- **Listovi blizu korena treba da imaju veliku frekvenciju, dok listovi daleko od korena treba da imaju malu frekvenciju**

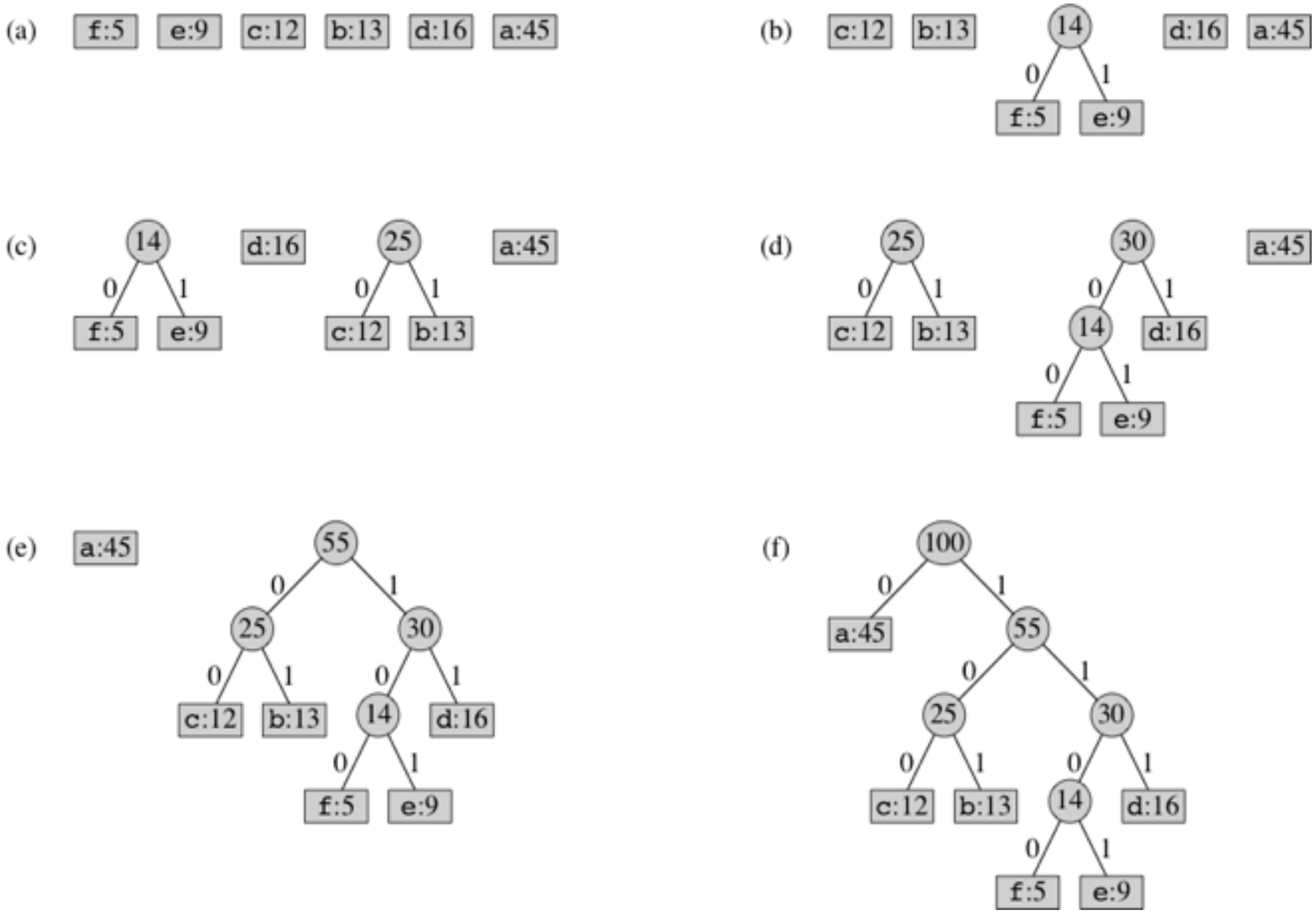
Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.

Symbol	Codeword
A	0
B	100
C	101
D	11



Hafmanovo stablo kodiranja

- D. Huffman je 1952. godine (kao student) pokazao da se **optimalno stablo kodiranja za proizvoljan tekst** može konstruisati **gramzivim algoritmom**.
- Hafmanovo stablo kodiranja
  - Svaki čvor stabla kao informaciju nosi frekvenciju (i karakter za listove)
  - **Frekvencija unutrašnjeg čvora je jednaka zbiru frekvencija njegove dece**
- Stablo konstruišemo od listova ka korenu
  1. Napravimo čvorove koji su listovi stabla i dodamo ih u neki skup S
  2. Neka su A i B dva čvora sa najmanjim frekvencijama u S
  3. Napravimo novi čvor C tako da
     C.right = A, C.left = B, i C.frequency = A.frequency + B.frequency
  4. A i B obrišemo iz S, C dodamo u S
  5. Ponavljamo korake 2, 3, i 4 dok u S ne ostane jedan čvor koji je koren stabla kodiranja


(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45
(e)
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9
(f)
100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

Proces konstrukcije stabla

- Skup S možemo realizovati sortiranom listom, a možemo i prioritetnom listom.

- S – sortirana lista
  1. Listove na početku treba sortirati po frekvencijama – O(n logn)
  2. Selekcija i brisanje dva čvora sa minimalnim frekvencijama – O(1)
  3. Dodavanje novog čvora u S – O(n)
  - **Ukupna složenost konstrukcije stabla - $O(n^2)$**
    - Korake 2 i 3 radimo (n – 1) puta

- S – prioritetna lista
  1. Listove na početku treba dodati u prioritetnu listu – O(n logn)
  2. Selekcija i brisanje dva čvora sa minimalnim frekvencijama – O(log n)
  3. Dodavanje novog čvora u S – O(log n)
  - **Ukupna složenost konstrukcije stabla – O(n logn)**

```
public class HuffmanTree {
	// informacija u cvoru hafmanovog stabla kodiranja
	private class CharFrequency implements Comparable<CharFrequency> {
		char c;    // karakter
		int freq;  // frekvencija

		public CharFrequency(char c, int freq) {
			this.c = c;
			this.freq = freq;
		}

		// koristimo java.util.PriorityQueue koji je MIN-PRIORITY-QUEUE
		// najmanji element ima najveci prioritet
		public int compareTo(CharFrequency other) {
			return freq - other.freq;
		}

		public String toString() {
			return "[" + c + ", f = " + freq + "]";
		}
	}

	// lista listova stabla -- karakteri iz teksta i njihove frekvencije
	private LinkedList<CharFrequency> frequencyList;

	// hafmanovo stablo kodiranja
	private BinaryTree<CharFrequency> hTree;
```

```
public HuffmanTree(String inputText) {
	computeFrequencies(inputText);
	construct();
}

private void computeFrequencies(String inputText) {
	frequencyList = new LinkedList<CharFrequency>();
	HashMap<Character, CharFrequency> frequencyIndex =
		new HashMap<Character, CharFrequency>();

	for (int i = 0; i < inputText.length(); i++) {
		char c = inputText.charAt(i);
		CharFrequency cf = frequencyIndex.get(c);
		if (cf == null) {
			cf = new CharFrequency(c, 1);
			frequencyList.addLast(cf);
			frequencyIndex.put(c, cf);
		} else {
			cf.freq++;
		}
	}
}
```

```
private void construct() {
	int numCharacters = frequencyList.size();
	PriorityQueue<BTNode<CharFrequency>> pq =
		new PriorityQueue<BTNode<CharFrequency>>(numCharacters);

	Iterator<CharFrequency> it = frequencyList.iterator();
	while (it.hasNext()) {
		CharFrequency cf = it.next();
		BTNode<CharFrequency> node = new BTNode<CharFrequency>(cf);
		pq.add(node);
	}

	while (pq.size() >= 2) {
		BTNode<CharFrequency> rightSubtree = pq.poll();
		int rsFreq = rightSubtree.getInfo().freq;
		BTNode<CharFrequency> leftSubtree = pq.poll();
		int lsFreq = leftSubtree.getInfo().freq;
		CharFrequency aggFreq =
			new CharFrequency('#', lsFreq + rsFreq);

		BTNode<CharFrequency> newNode =
			new BTNode<CharFrequency>(
				aggFreq, leftSubtree, rightSubtree);
		pq.add(newNode);
	}

	hTree = new BinaryTree<CharFrequency>();
	hTree.setRoot(pq.poll());
}
```

Formiranje kodova – obilazak stabla

```
public void printCodes() {
	BTNode<CharFrequency> root = hTree.getRoot();
	if (root.getLeft() == null && root.getRight() == null)
		System.out.println(root.getInfo() + " --> 1");
	else
		printCodes(root, "");
}

private void printCodes(BTNode<CharFrequency> current, String code) {
	if (current.getLeft() == null && current.getRight() == null) {
		System.out.println(current.getInfo() + " --> " + code);
	} else {
		BTNode<CharFrequency> left = current.getLeft();
		if (left != null) {
			printCodes(left, code + "0");
		}

		BTNode<CharFrequency> right = current.getRight();
		if (right != null) {
			printCodes(right, code + "1");
		}
	}
}
```

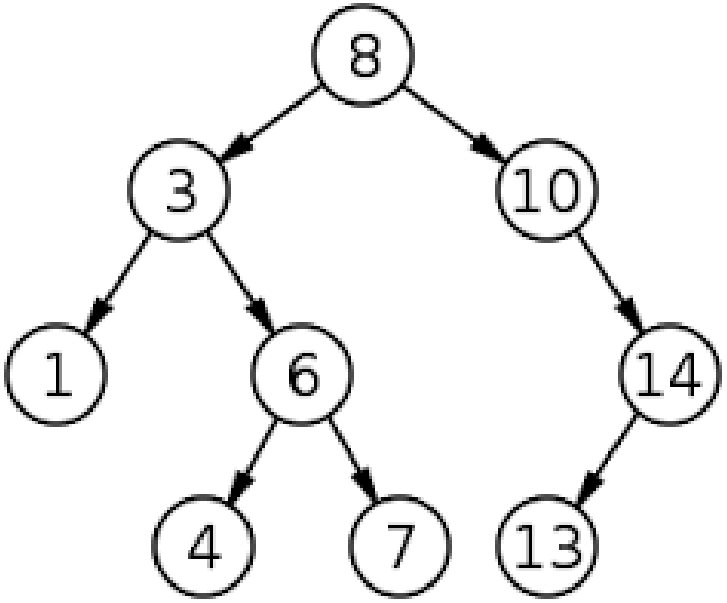
Primer...

```
public static void main(String[] args) {
      HuffmanTree ht = new HuffmanTree("ana voli milovana");
      ht.printCodes();
}
```

```
[n, f = 2] --> 0000
[m, f = 1] --> 0001
[ , f = 2] --> 001
[v, f = 2] --> 010
[i, f = 2] --> 011
[o, f = 2] --> 100
[l, f = 2] --> 101
[a, f = 4] --> 11
```

Binarno stablo pretraživanja

- Binarno stablo pretraživanja (***BST – binary search tree***) je binarno stablo sa sledećim osobinama
  - Čvorovi stabla su jedinstveni (nema duplikata)
  - Za svaki čvor stabla A važi
    - A je veći od svih elemenata u levom podstablu
    - A je manji od svih elemenata u desnom podstablu



Binarno stablo pretraživanja

- Skupove možemo realizovati binarnim stablom pretraživanja
- Tada su elementarne operacije u radu sa skupom (dodavanje, pretraživanje, brisanje) u proseku logaritamske složenosti

```
public interface Set<T> {

        /**
         * Dodaje element u skup.
         * Vraca false ukoliko element vec postoji u skupu
         */
        boolean insert(T element);

        /**
         * Brise element iz skupa.
         * Vraca false ukoliko element ne postoji u skupu.
         */
        boolean remove(T element);

        /**
         * Proverava da li je element u skupu.
         */
        boolean member(T element);
}
```

Pretraživanje BST-a

- **Da li se informacija x nalazi u BST B?**
- Postupak pretraživanja
  - Da li je **x** u korenu stabla B? Ako jeste kraj
  - Ako je **x** manje od **B.info** tada x ne može da bude u desnom podstablu → traži **x** u levom podstablu (rekurzivno na isti način)
    - **Ako levi sin ne postoji → x nije u stablu**
  - Inače, traži x u desnom podstablu (rekurzivno na isti način)
    - **Ako desni sin ne postoji → x nije u stablu**
- Ako je BST balansirano tada u svakom koraku veličinu problema polovimo → O(log n)
- Pretraživanje BST-a je šetnja kroz stablo od korena ka listovima gde iz svakog čvora idemo ili levo ili desno

Dodavanje novog čvora u BST

- Dodavanje novog čvora u BST je uvek kreiranje novog lista BST-a
- Postupak dodavanja informacije **x** u BST B:
  - Ako je B prazno napravi korenski čvor koji sadrži **x**
  - Ako B nije prazno tada se prošetaj kroz stablo od korena ka listovima na sledeći način
    - Ako je **x** manje od informacije u tekućem čvoru idi levo
      - **Ako levi sin ne postoji → novi čvor koji sadrži x postaje levi sin tekućeg čvora**
    - Ako je **x** veće od informacije u tekućem čvoru idi desno
      - **Ako desni sin ne postoji → novi čvor koji sadrži x postaje levi sin tekućeg čvora**
    - Ako je **x** jednako informaciji u tekućem čvoru tada **x** već postoji u stablu → obustavi operaciju dodavanja

```java
public class BST<T extends Comparable<T>> implements Set<T> {
	private BTNode<T> root = null;

	private class SearchResult {
		BTNode<T> node, parent;
		public SearchResult(BTNode<T> node, BTNode<T> parent) {
			this.node = node;
			this.parent = parent;
		}
	}

	private SearchResult search(T info) {
		BTNode<T> current = root, parent = null;
		boolean found = false;

		while (current != null && !found) {
			int cmp = info.compareTo(current.getInfo());
			if (cmp == 0) {
				found = true;
			} else {
				parent = current;
				if (cmp < 0) current = current.getLeft();
				else current = current.getRight();
			}
		}

		return new SearchResult(current, parent);
	}
```

```
public boolean member(T element) {
	SearchResult sr = search(element);
	return sr.node != null;
}

public boolean insert(T element) {
	if (root == null) {
		root = new BTNode<T>(element);
		return true;
	}

	SearchResult sr = search(element);
	if (sr.node != null)
		return false;

	BTNode<T> newNode = new BTNode<T>(element);
	BTNode<T> parent = sr.parent;
	if (element.compareTo(parent.getInfo()) < 0)
		parent.setLeft(newNode);
	else
		parent.setRight(newNode);

	return true;
}
```

Brisanje čvora iz BST

- Obrisati informaciju x iz BST-a B
- Pretražujemo stablo da nađemo čvor koji sadrži x
- Razlikujemo tri slučaja
  - Briše se list – trivijalno
  - Briše se unutrašnji čvor koji ima samo jednog sina – prosto prevezivanje kao kod jednostruko povezanih listi
  - Briše se unutrašnji čvor koji ima oba sina – malo komplikovanije, ali ništa strašno
  - Moramo voditi računa o tome da li brišemo čvor koji je koren stabla

```java
public boolean remove(T element) {
	SearchResult sr = search(element);
	if (sr.node == null)
		return false;

	BTNode<T> toRemove = sr.node;
	BTNode<T> parent = sr.parent;

	// prvi slucaj (cvor je list)
	if (toRemove.getLeft() == null &&
	    toRemove.getRight() == null)
	{
		removeLeaf(toRemove, parent);
	}
	// drugi slucaj (cvor nema jednog sina)
	else
	if (toRemove.getLeft() == null ||
	    toRemove.getRight() == null)
	{
		removeInternalWithOneChild(toRemove, parent);
	} else {
		removeInternal(toRemove, parent);
	}

	return true;
}
```

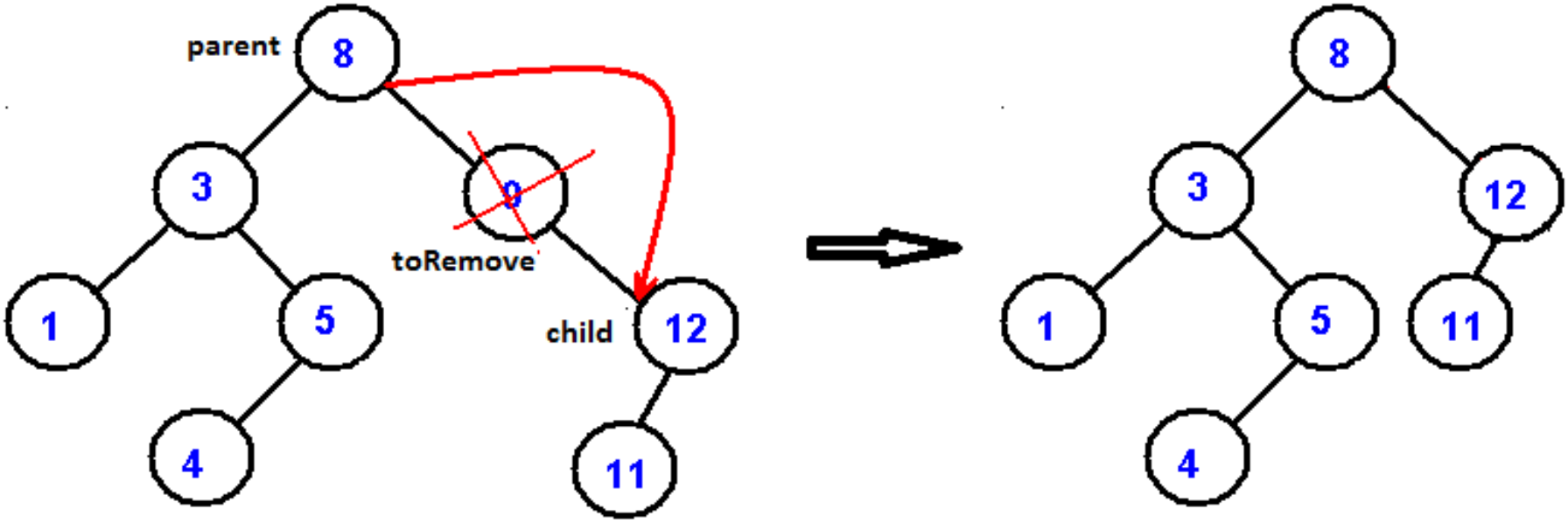
Brisanje čvora koji je list

- Brišemo **toRemove** čiji je otac **parent**
  - parent.setLeft(null) ako je **toRemove** levi sin **parent**
  - parent.setRight(null) ako je toRemove desni sin **parent**
  - Ako je parent == null (ekvivalentno toRemove == root) brišemo koren

```
private void removeLeaf(BTNode<T> toRemove, BTNode<T> parent) {
	if (parent == null) {
		// uklanjamo korenski element
		root = null;
	} else {
		// da li je toRemove sa leve ili desne strane parent
		boolean left = parent.getLeft() == toRemove;
		if (left)
			parent.setLeft(null);
		else
			parent.setRight(null);
	}
}
```

Brisanje čvora koji ima jednog sina

- Brišemo **toRemove** čiji je otac **parent**
- **toRemove** ima jednog sina **child**
- Ako je parent == null tada brišemo koren → root = child
- Inače **child** postaje sin **parent**
  - toRemove sa leve strane parent: parent.setLeft(child)
  - toRemove sa desne strane parent: parent.setRight(child)



Brisanje čvora koji ima jednog sina

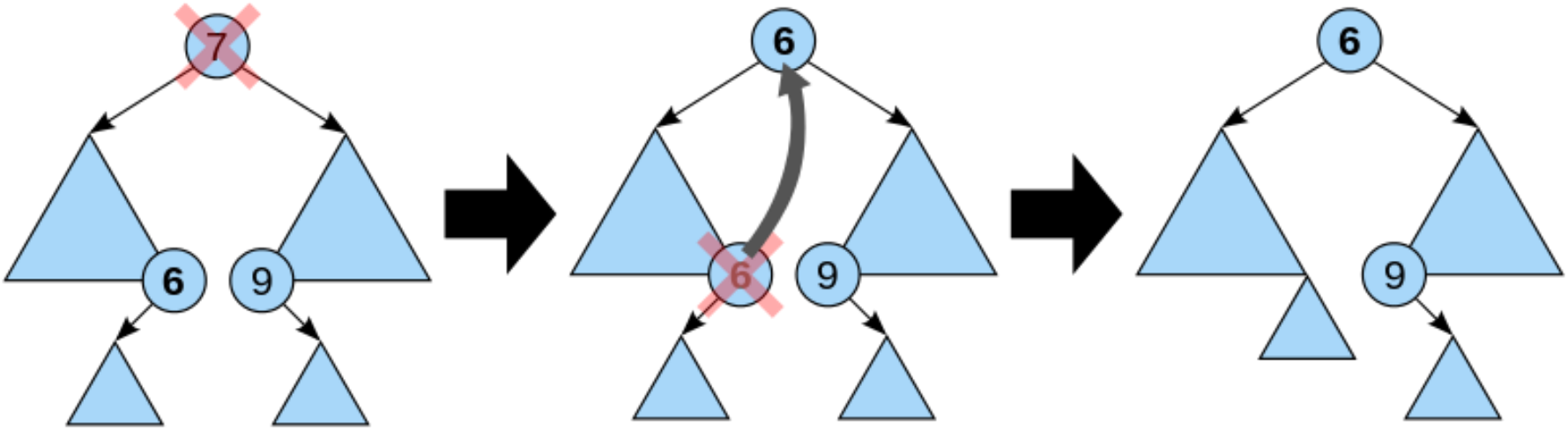
```
private void removeInternalWithOneChild(
        BTNode<T> toRemove, BTNode<T> parent)
{
        // child -- jedino dete toRemove cvora
        BTNode<T> child = toRemove.getLeft();
        if (child == null) {
                child = toRemove.getRight();
        }

        if (parent == null) {
                root = child;
        } else {
                boolean left = parent.getLeft() == toRemove;
                if (left) {
                        parent.setLeft(child);
                } else {
                        parent.setRight(child);
                }
        }
}
```

Brisanje čvora koji ima oba sina

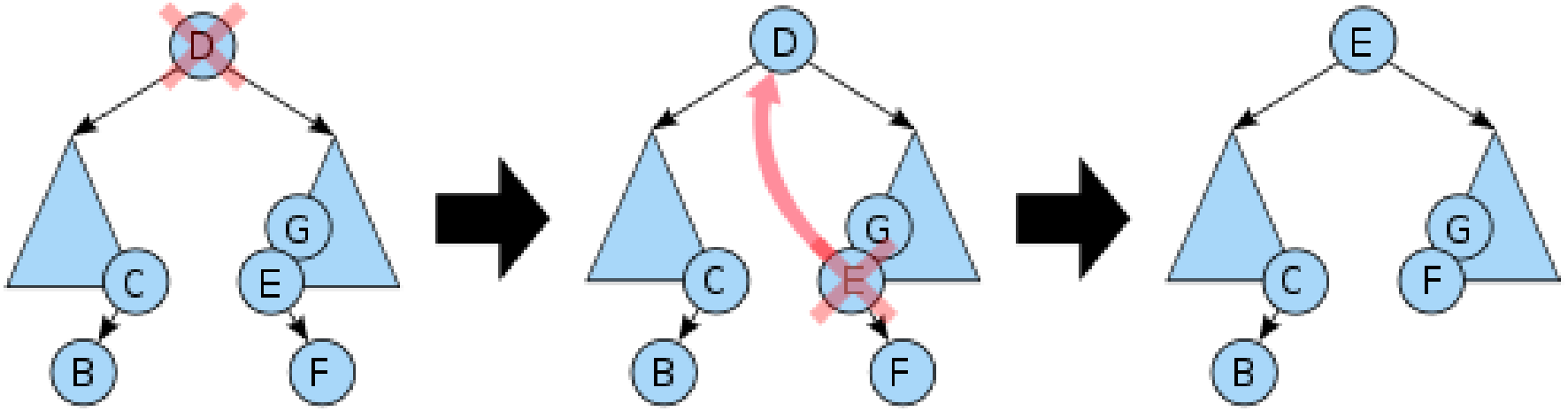
- Ideja: ne brisati čvor C koji ima oba sina, nego
  - obrisati neki drugi čvor koji je ili list ili ima jednog sina
  - Informaciju čuvanu u obrisanom čvoru staviti u C

- Dva načina realizacije trika
  - **Nađemo M - najmanji čvor u desnom podstablu čvora C**
    - M ne može imati levog sina (onda ne bi bio najmanji)
    - Ako zamenimo C sa M očuvava se osobina BST
      - M je veći od svih čvorova u levom podstablu C, jer je M > C, a svi čvorovi u levom podstablu C su manji od C
      - M je najmanji od svih čvorova u desnom podstablu C
  - **Nađemo najveći čvor u levom podstablu čvora C**
  - Potpuno svejedno koji način realizujemo

Maksimalni element u levom podstablu



Minimalni element u desnom podstablu (implementiraćemo ovaj pristup)



Pronalaženje bitnih čvorova

- **Najmanji čvor u desnom podstablu čvora C**
  - Iz C odemo u desno, potom idemo ulevo koliko god možemo

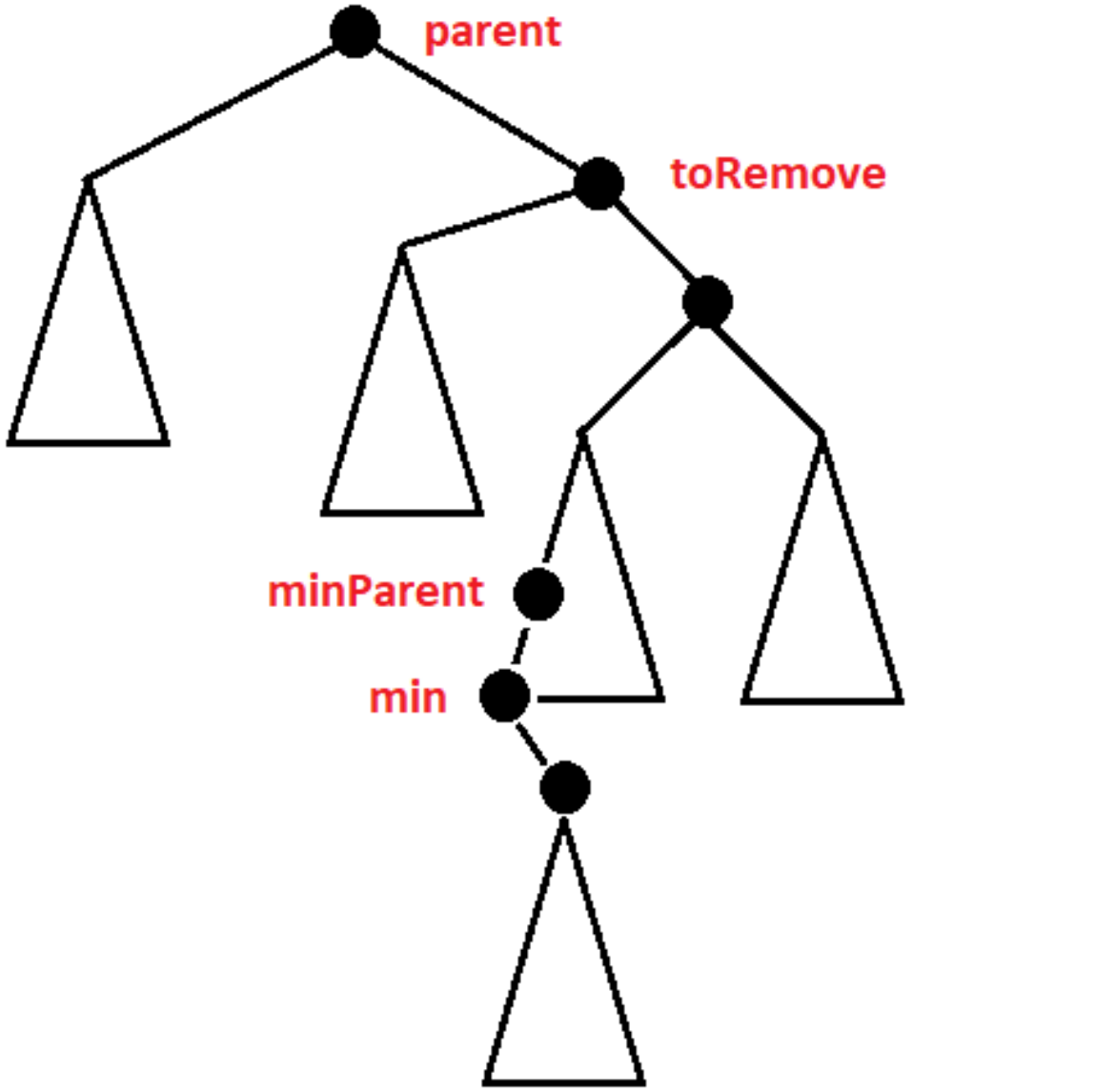
```
min = C.getRight();
while (min.getLeft() != null)
    min = min.getLeft();
```

- **Najveći čvor u levom podstablu C**
  - Iz C idemo u levo, potom idemo udesno koliko god možemo

```
max = C.getLeft();
while (max.getRight() != null)
    max = max.getRight();
```

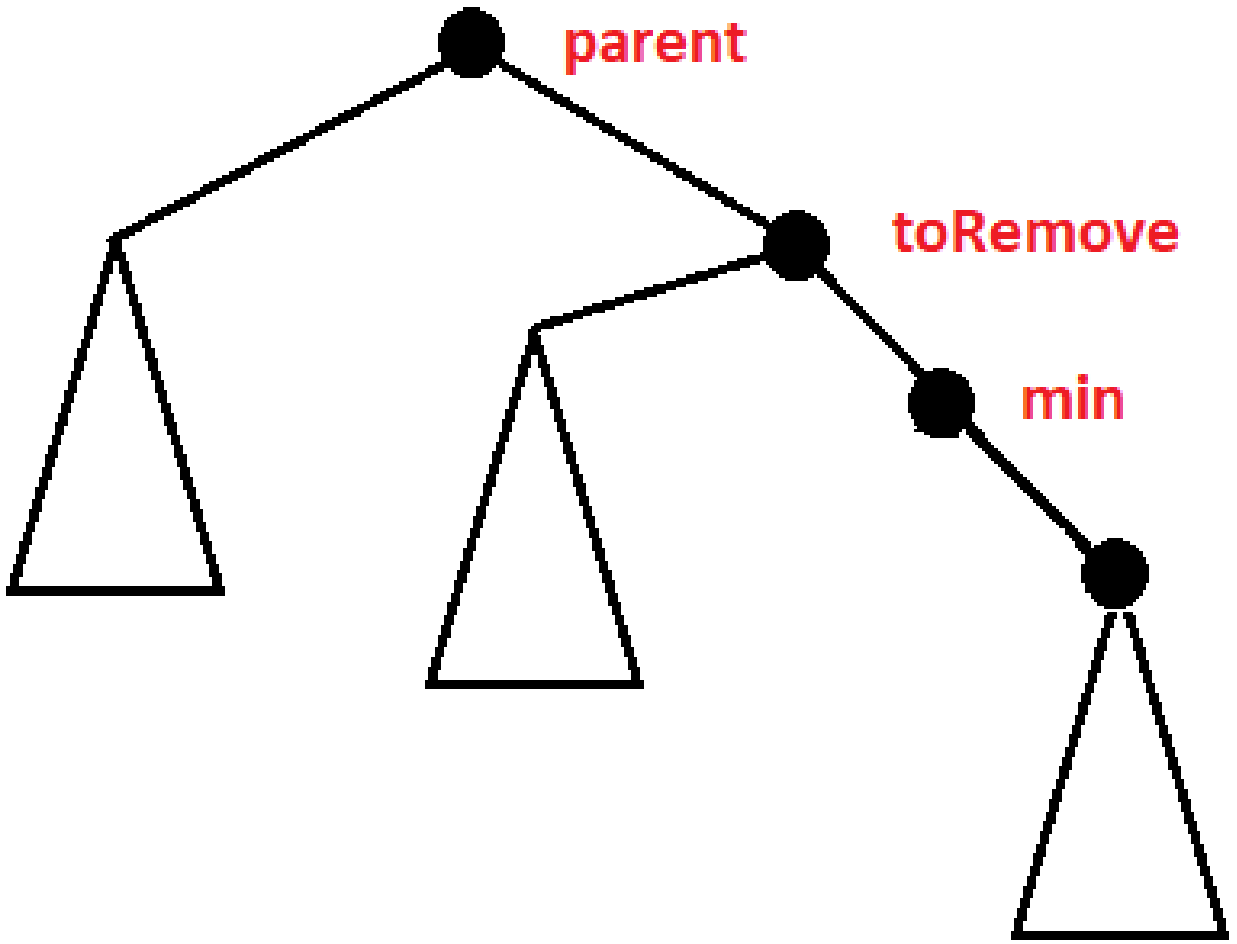
- Treba nam i otac od min (max) zbog prevezivanja ako min (max) nije list

Slučaj 1: bar jedan korak levo



minParent.setLeft(min.getRight())

Slučaj 2: nijedan korak levo




```
if (minParent == toRemove)
    minParent.setRight(min.getRight());
```

```java
private void removeInternal(BTNode<T> toRemove, BTNode<T> parent) {
	// nadji minimum u desnom podstablu i njegovog oca
	BTNode<T> min = toRemove.getRight();
	BTNode<T> minParent = toRemove;
	while (min.getLeft() != null) {
		minParent = min;
		min = min.getLeft();
	}

	// informacija u minimumu
	T minInfo = min.getInfo();

	// izbaci min iz stabla
	if (minParent == toRemove) {
		// nije napravljen nijedan korak u levo
		minParent.setRight(min.getRight());
	} else {
		// napravljen je bar jedan korak u levo
		minParent.setLeft(min.getRight());
	}

	toRemove.setInfo(minInfo);
}
```