

ABSTRAKTNI PODATKOVNI TIPI

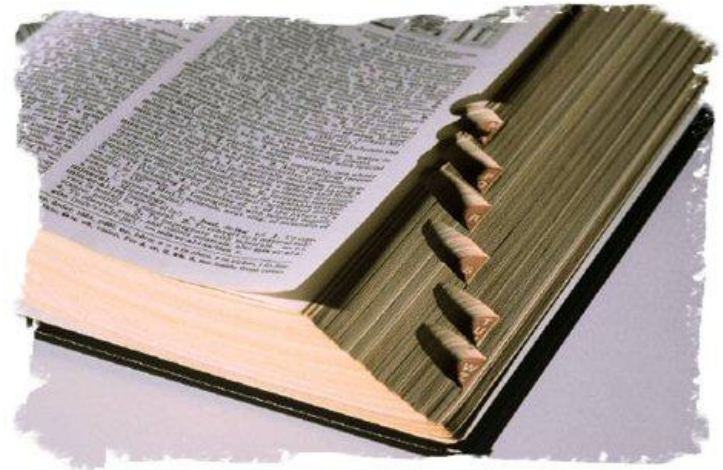
Osnovni abstraktni podatkovni tipi, ki jih potrebujemo za razvoj algoritmov so:

- seznam (list)
- vrsta (queue)
- sklad (stack)
- preslikava (map)
- množica (set)
- drevo (tree)



- SLOVAR (angl. *dictionary*) – poseben primer *ADT množica*, ki omogoča samo vstavljanje, brisanje in iskanje elementa





ADT SLOVAR

(angl. *dictionary*)



ADT SLOVAR

Slovar je poseben primer podatkovnega tipa množice, ki omogoča samo tri osnovne operacije:

- VSTAVLJANJE elementa
- BRISANJE elementa
- ISKANJE elementa

Za učinkovito iskanje elementov (ki se zahteva pri *ADT slovar* za razliko od *ADT seznam*) potrebujemo relacijo **urejenosti** med elementi.

Urejenost je definirana bodisi na elementih samih bodisi na delih elementov – ključih (keys)

ADT SLOVAR

Osnovne operacije definirane za ADT DICTIONARY:

- `MAKENULL(D)` - naredi prazen slovar D
- `MEMBER(x, D)` - preveri, če je element x v slovarju D
- `INSERT(x, D)` - vstavi element x v slovar D
- `DELETE(x, D)` - zbriše element x iz slovarja D

Zaradi urejenosti elementov v slovarju je možno učinkovito implementirati tudi operacije:

- iskanje minimalnega elementa
- iskanje maksimalnega elementa
- iskanje predhodnika (predecessor)
- iskanje naslednika (successor)
- izpis elementov na danem intervalu

PRIMERI UPORABE SLOVARJA



- slovar črkovalnika: pri črkovanju je treba hitro preveriti, če je dana beseda v slovarju; uporabnik lahko dodaja nove besede v slovar
- kompresija besedila zahteva izračun frekvence vsake besede: slovar kreiramo tako, da besede dodajamo v slovar:
 če besede še ni v slovarju,
 jo dodamo in postavimo števec besede na 1
 sicer števec besede povečamo za 1
- vsaka podatkovna baza je pravzaprav slovar:
 - elementi so urejeni po ključih, zaradi hitrega iskanja
 - ker so baze shranjene na (relativno) počasnem disku, je treba izbrati podatkovno strukturo, ki minimizira število dostopov do diska



IMPLEMENTACIJE SLOVARJA

Zgoščena tabela

Prednost: časovna kompleksnost vseh operacij je pod določenimi pogoji reda $O(1)$

Slabosti:

- fiksna podatkovna struktura,
- fiksna zgoščevalna funkcija (zaradi sovpadanja se lahko izrodi),
- neučinkovite operacije, ki temeljijo na urejenosti elementov po ključih

Drevesne strukture

Lastnosti:

- časovna kompleksnost osnovnih operacij je reda $O(\log n)$
- časovna kompleksnost operacij na podlagi urejenosti elementov slovarja je reda $O(\log n)$

pri čemer je n število elementov slovarja.

ISKALNA DREVEŠA

- binarna iskalna drevesa (binary search trees);
- lomljena drevesa (splay trees), ki sta jih uvedla Daniel Sleator in Robert Tarjan leta 1983;
- rdeče-črna drevesa (red-black trees), ki jih je definirali R. Bayer leta 1972, ime “rdeče-črna drevesa” pa sta uvedla Leo Guibas in Robert Sedgwick leta 1978;
- AVL-drevesa, katerih ime je sestavljeno iz prvih črk dveh avtorjev (Adel’son-Vel’skii in Landis), nastala pa so leta 1962;
- 2-3 drevesa je definirali J.E. Hopcroft leta 1970;
- B-drevesa so posplošitev 2-3 dreves, definirala pa sta jih Bayer in McCreight leta 1972;
- k-d drevesa, ki jih je opisal Bentley leta 1975.

BINARNO ISKALNO DREVO

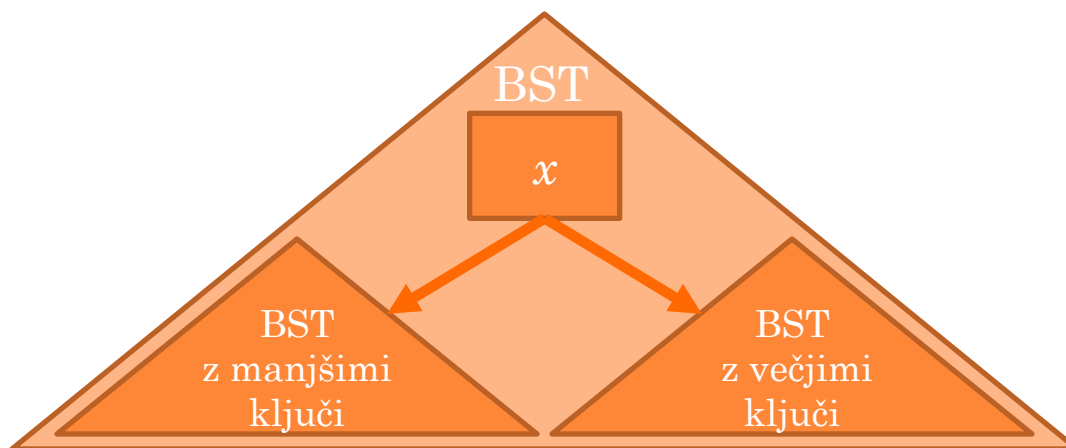
Binarno iskalno drevo (binary search tree, BST) je najbolj preprosta drevesna implementacija slovarja.

Rekurzivna definicija:

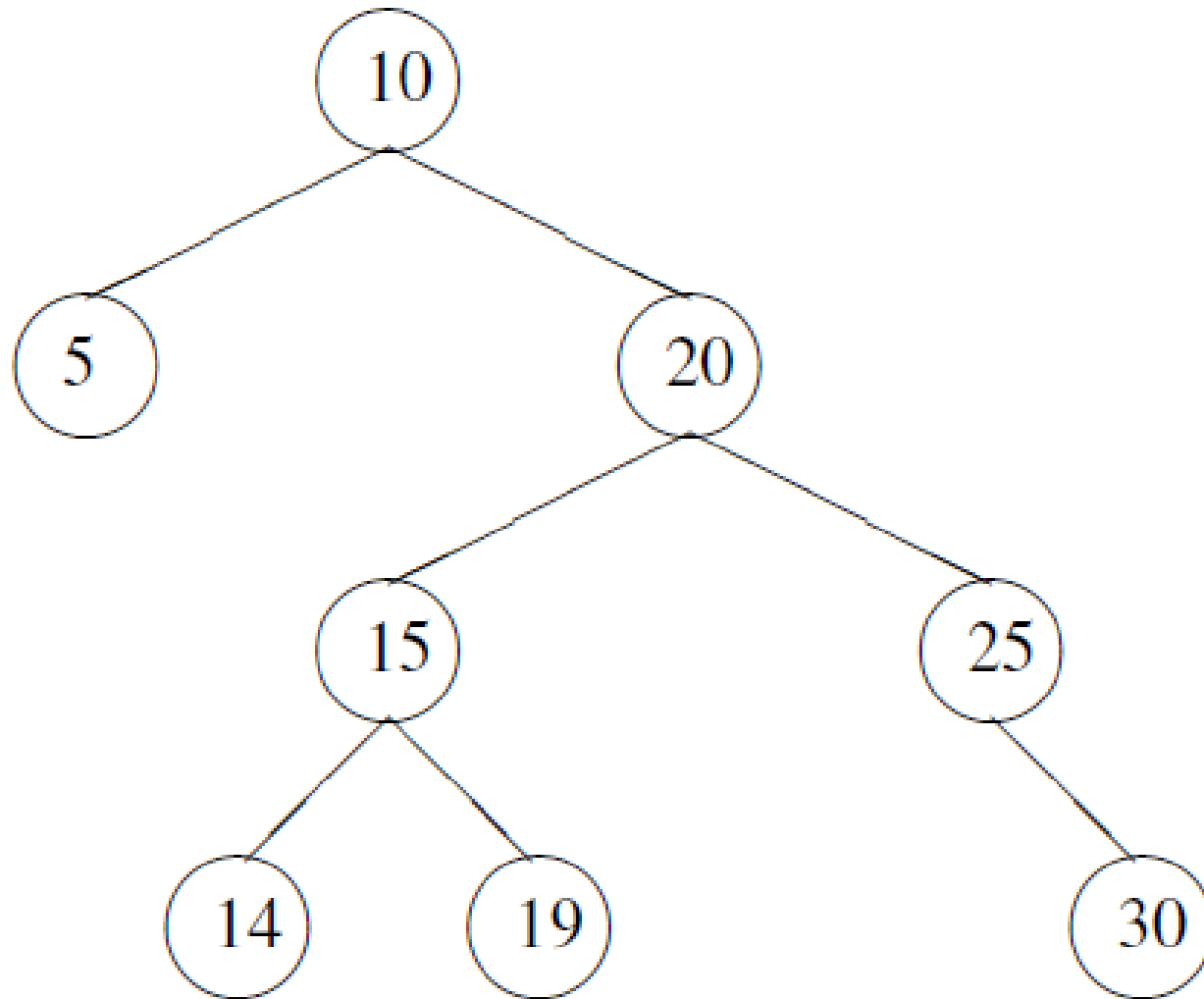
- prazno drevo je BST
- drevo z oznako (ključem) korena x z levim in desnim poddrevesom L in R je BST, če velja:

$$\forall y \in L: y < x \wedge \forall y \in R: y > x$$

in sta tudi L in R BST



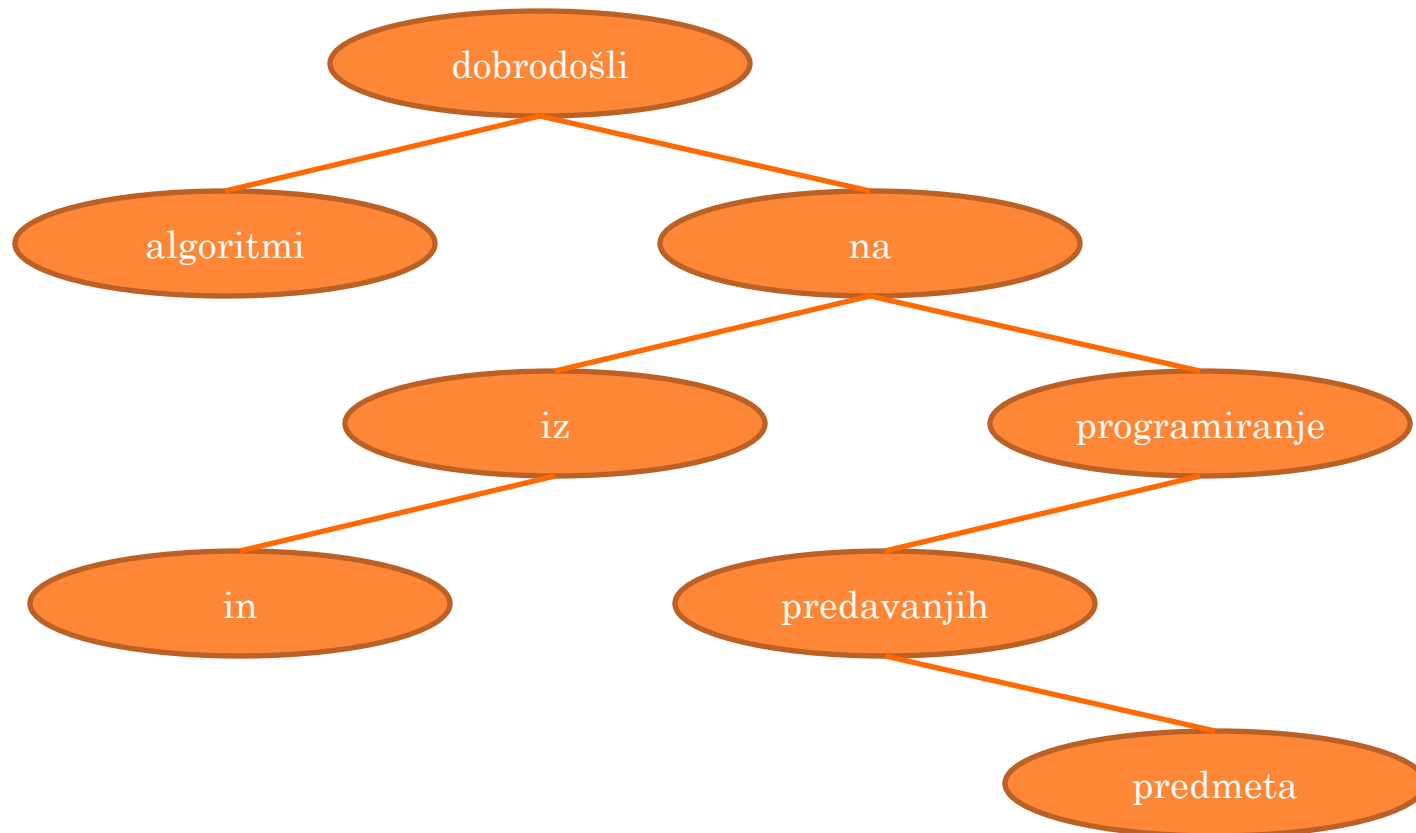
PRIMER BST



PRIMER BST

BST vsebuje besede iz stavka

“dobrodošli na predavanjih iz predmeta programiranje in algoritmi”



IMPLEMENTACIJA BST

Drevo je podano kot referenca na vozlišče v korenu:

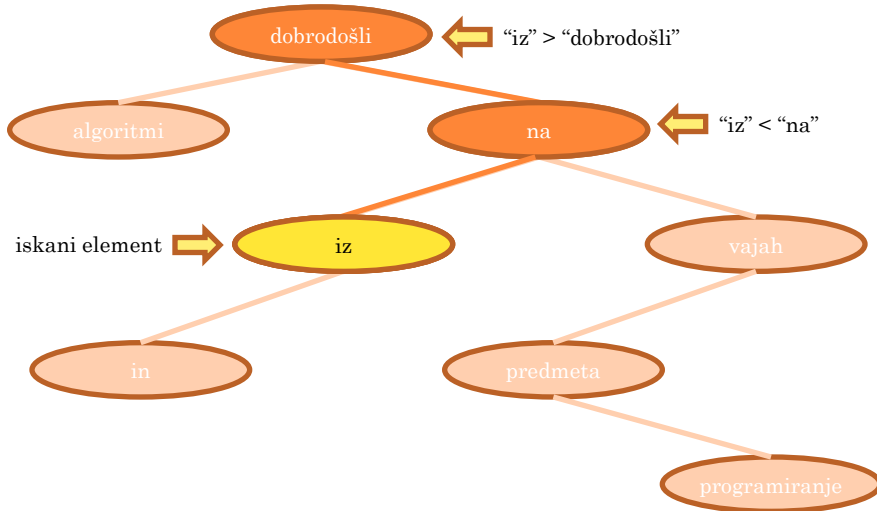
```
public class BSTree implements Dictionary {  
    BSTreeNode rootNode; // referenca na koren  
    ...  
}
```

Vozlišče je definirano kot:

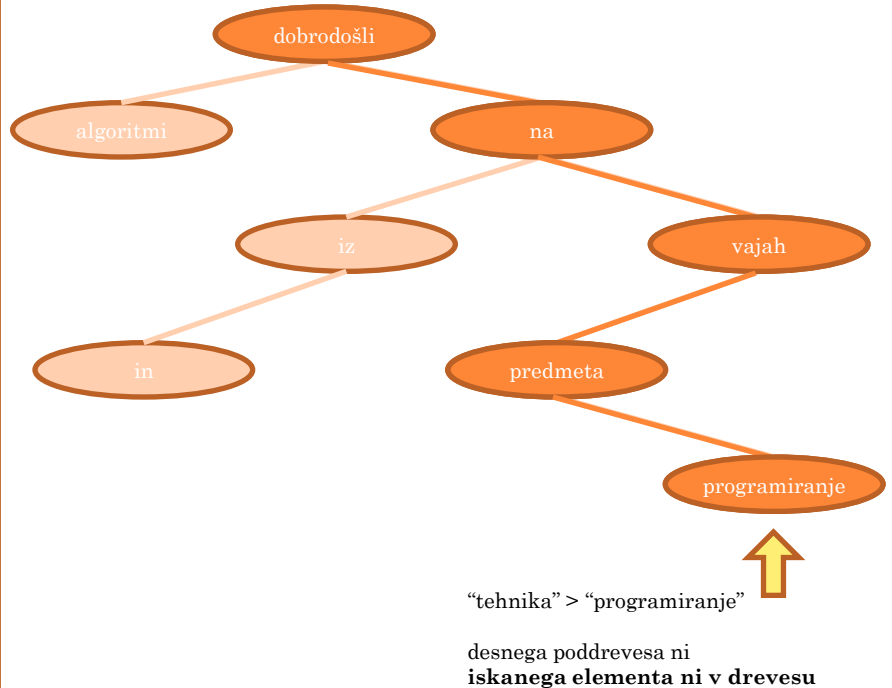
```
public class BSTreeNode {  
    Comparable key; // ključ  
    BSTreeNode left; // referenca na levo poddrevo  
    BSTreeNode right; // referenca na desno poddrevo  
    ...  
}
```


ISKANJE ELEMENTA V BST

Iskanje ključa "iz"



Iskanje ključa "tehnika"



Maksimalno število korakov pri iskanju elementa je enako višini drevesa.

Časovna kompleksnost pri poravnem drevesu je reda $O(\log n)$ pri izrojenem drevesu pa $O(n)$.

ISKANJE ELEMENTA V BST

Rekurzivno iščemo v enem od poddreves.

Dva robna pogoja: 1) če elementa ni v drevesu: pridemo v prazno poddrevo.

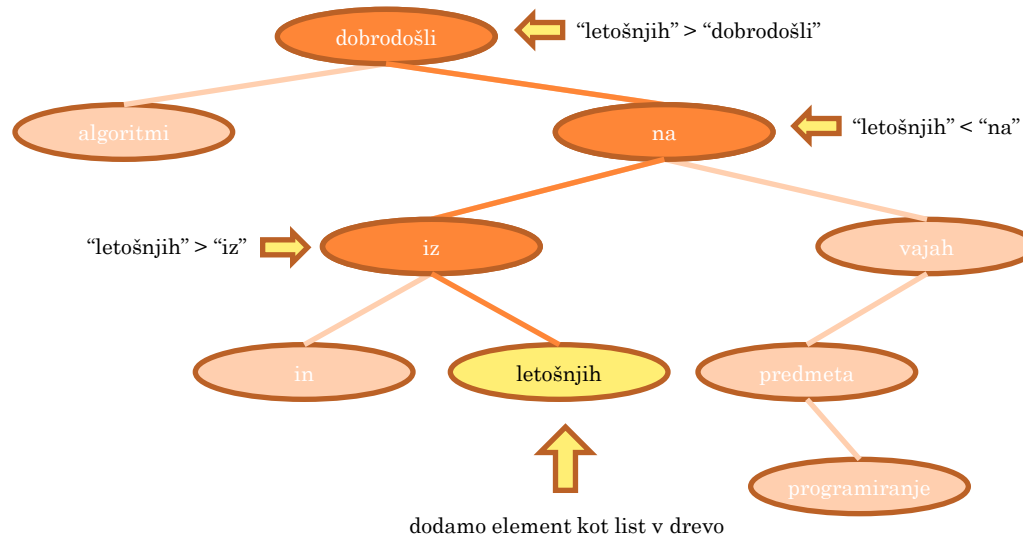
2) če element je v drevesu: ga najdemo v korenu poddrevesa.

```
private boolean member(Comparable x, BSTreeNode node) {  
    if (node == null)  
        return false;  
    else if (x.compareTo(node.key) == 0)  
        return true;  
    else if (x.compareTo(node.key) < 0)  
        return member(x, node.left);  
    else  
        return member(x, node.right);  
} // member
```


DODAJANJE ELEMENTA V BST

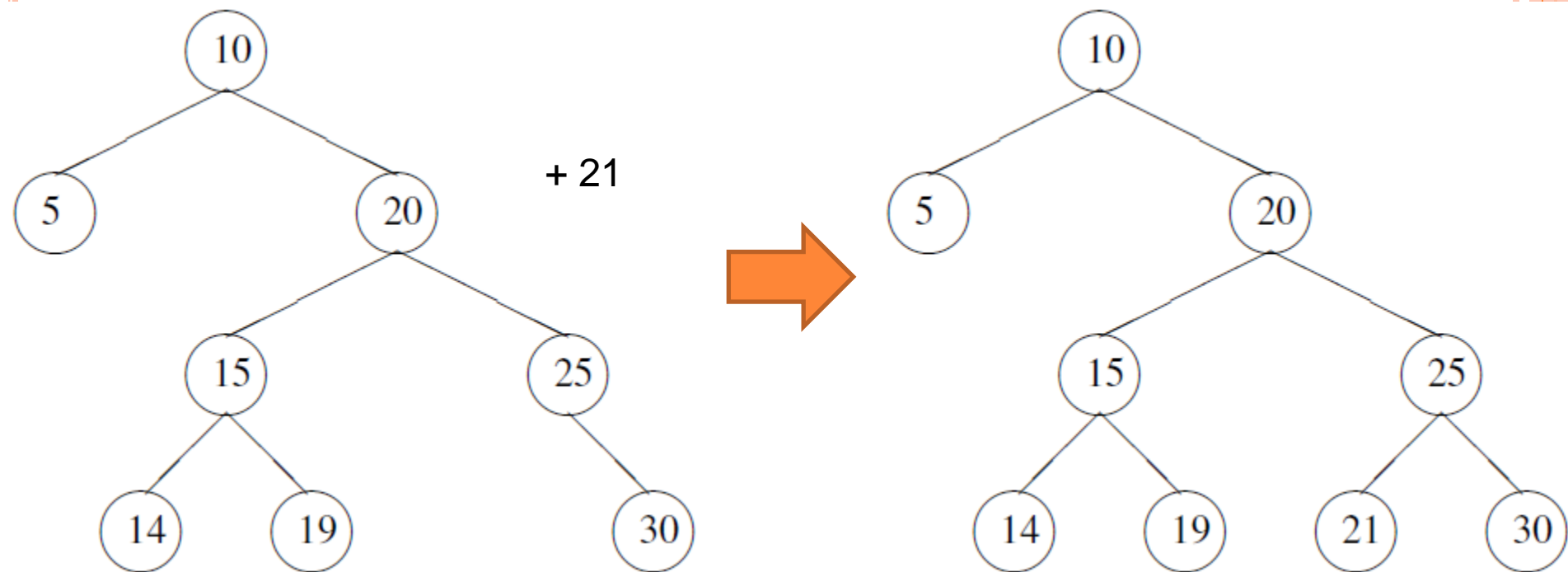
Dodajanje elementa kot **LIST** v drevo

Dodajanje ključa “letošnjih”



Maksimalno število korakov pri dodajanju elementa kot list je enako višini drevesa

DODAJANJE ELEMENTA V BST



DODAJANJE ELEMENTA V BST

Rekurzivno dodamo v enega od poddreves.

(Normalni) robni pogoj: prazno poddrevo zamenjamo z listom.

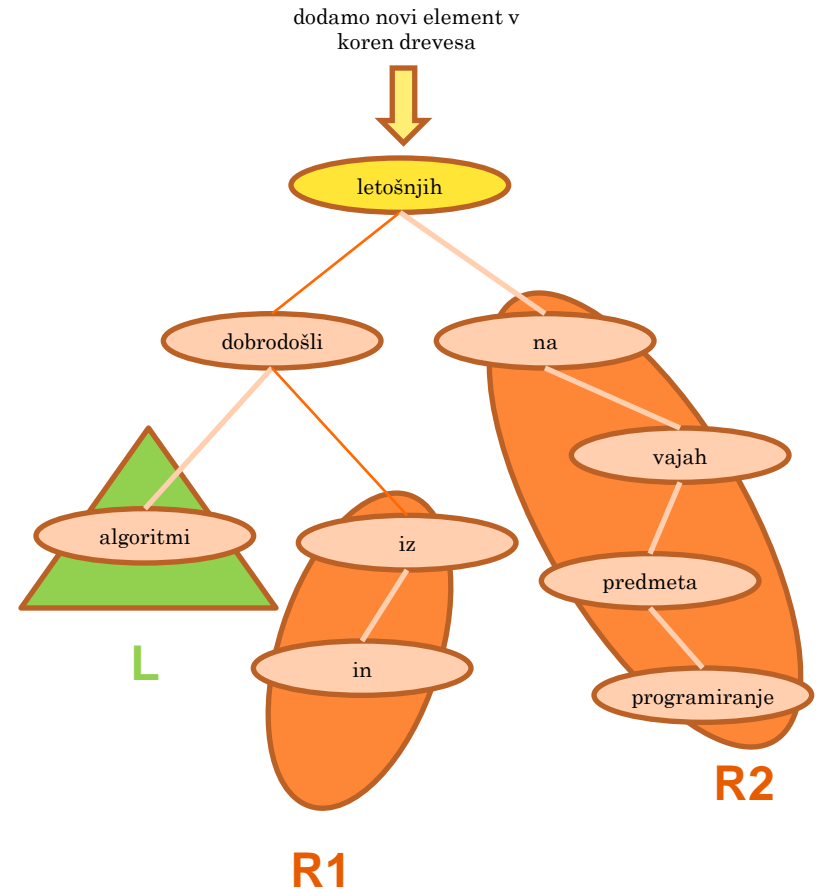
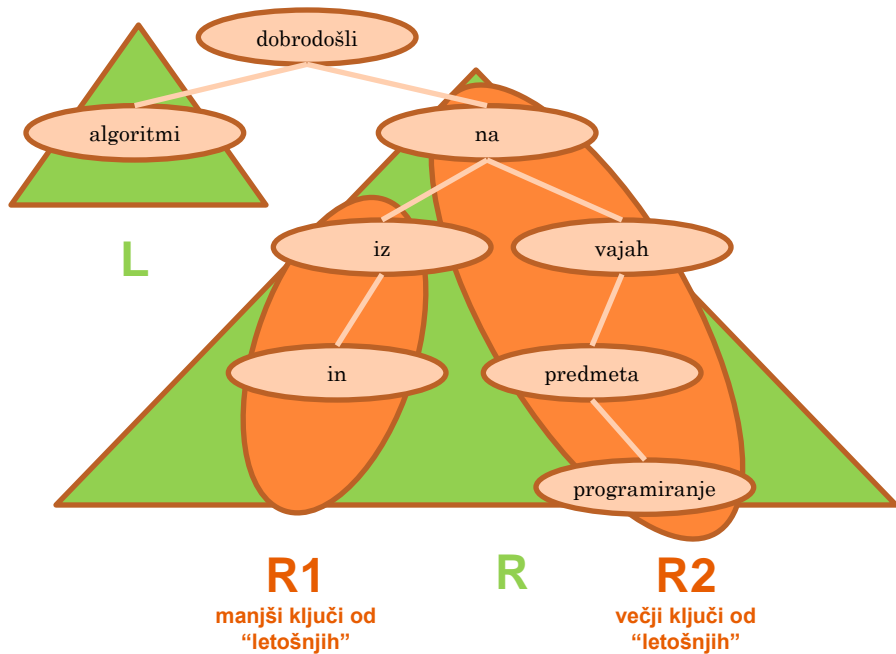
Izredni robni pogoj: element je že v drevesu...

```
protected BSTreeNode insertLeaf(Comparable x, BSTreeNode node) {  
    if (node == null)  
        node = new BSTreeNode(x);  
    else if (x.compareTo(node.key) < 0)  
        node.left = insertLeaf(x, node.left);  
    else if (x.compareTo(node.key) > 0)  
        node.right = insertLeaf(x, node.right);  
    else  
        ; // duplicate; do nothing or throw exception  
    return node;  
} // insertLeaf
```


DODAJANJE ELEMENTA V BST

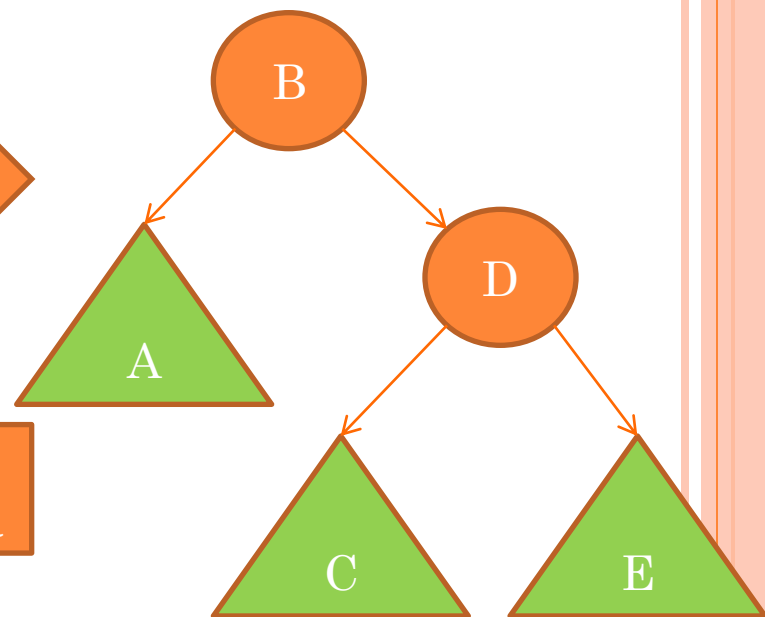
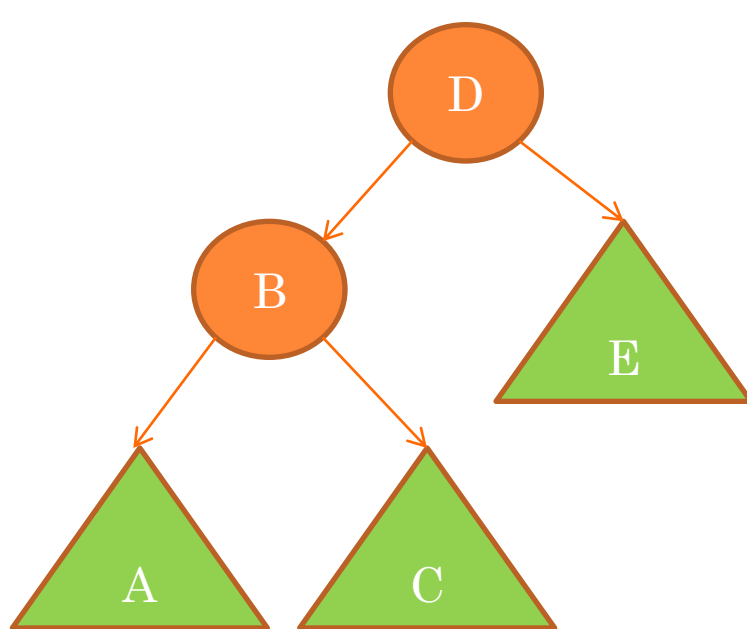
Dodajanje elementa v **KOREN** drevesa

Dodajanje ključa “letošnjih”

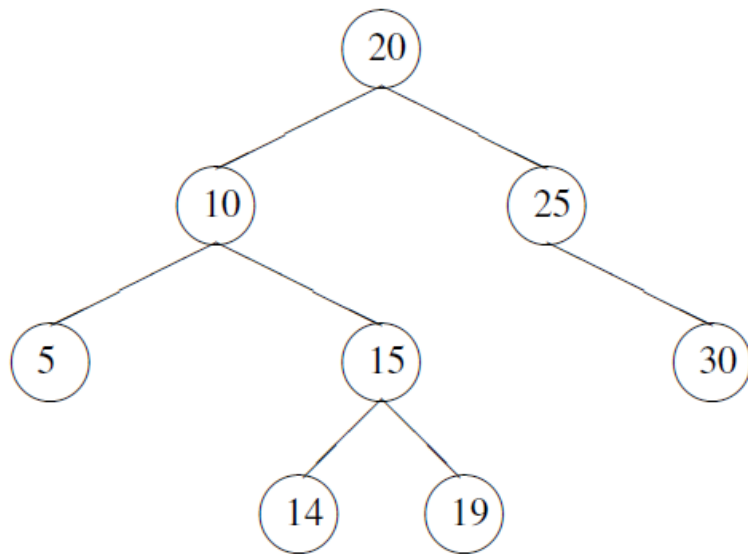


Maksimalno število korakov je sorazmerno višini drevesa

ROTACIJA

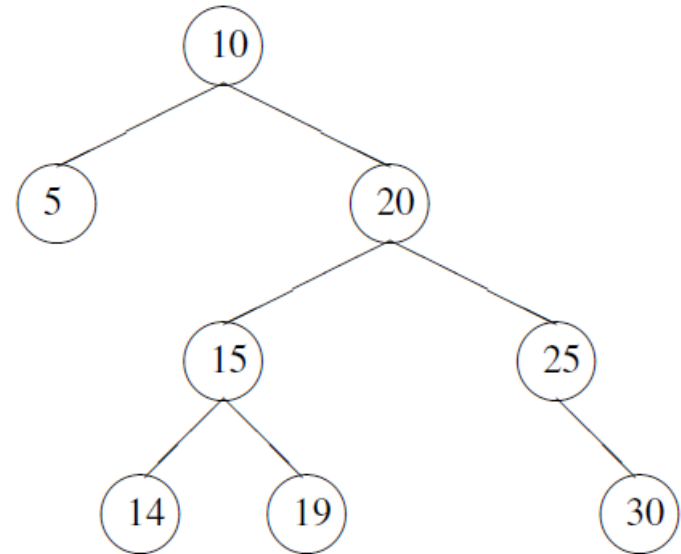


ROTACIJA




desna
rotacija

leva
rotacija



ROTACIJA

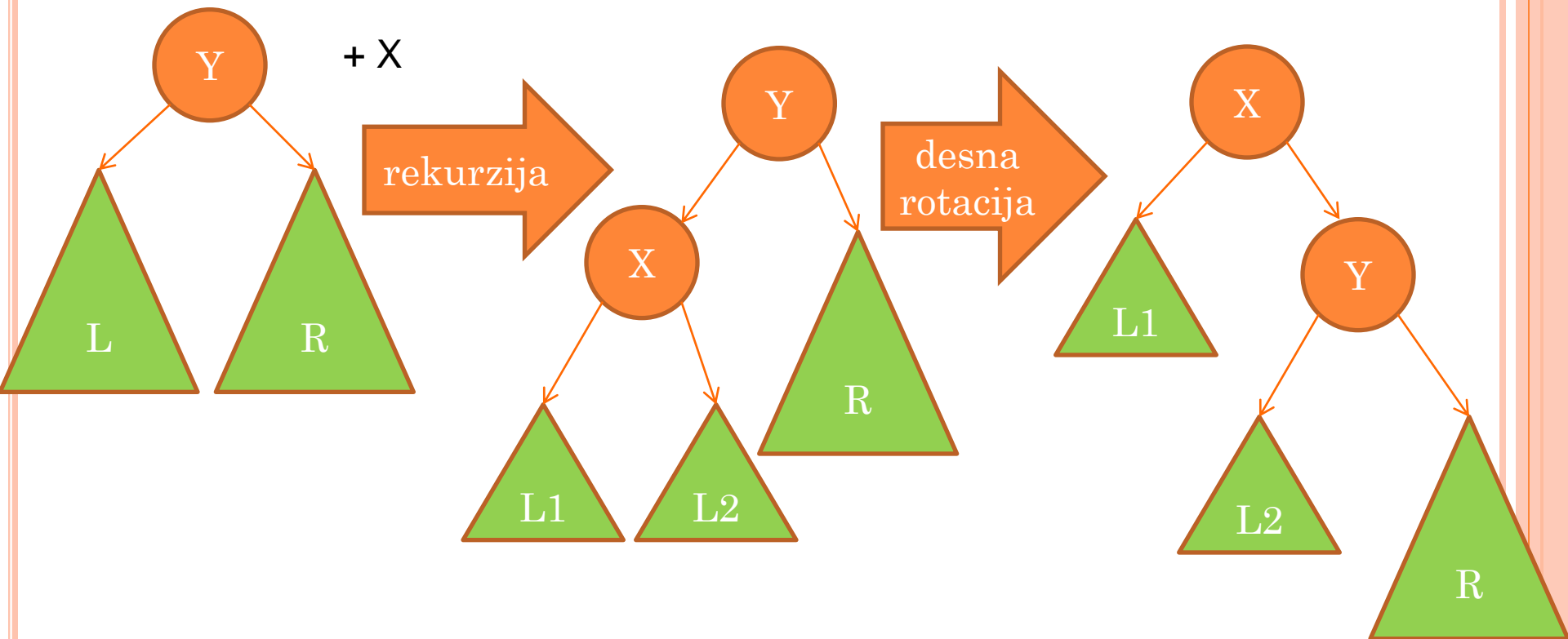


```
protected BSTreeNode rightRotation(BSTreeNode node) {  
    BSTreeNode temp = node;  
    node = node.left;  
    temp.left = node.right;  
    node.right = temp;  
    return node;  
}
```

```
protected BSTreeNode leftRotation(BSTreeNode node) {  
    BSTreeNode temp = node;  
    node = node.right;  
    temp.right = node.left;  
    node.left = temp;  
    return node;  
}
```

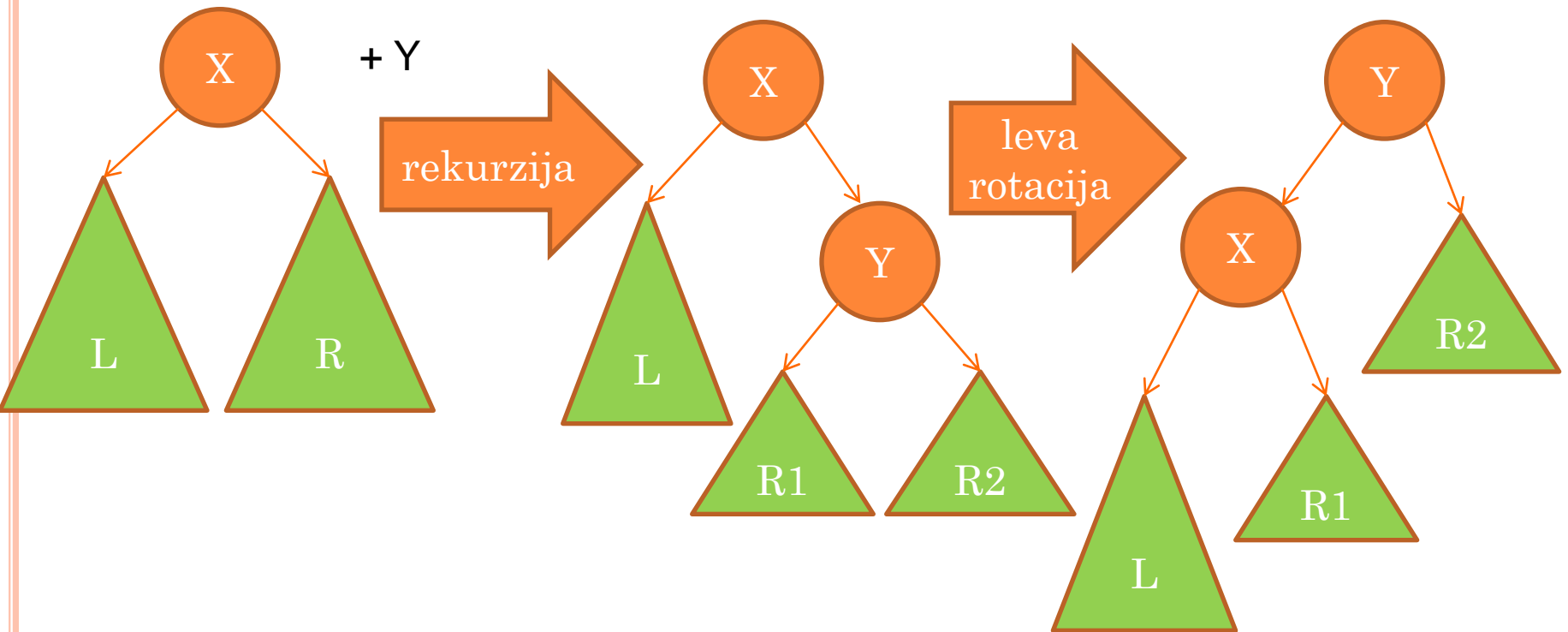

DODAJANJE ELEMENTA V KOREN

1. Novi element X manjši od korena: $X < Y$
rekurzija doda X v koren levega poddrevesa



DODAJANJE ELEMENTA V KOREN

2. Novi element Y večji od korena: $Y > X$
rekurzija doda Y v koren desnega poddrevesa



DODAJANJE ELEMENTA V KOREN

```
private BSTreeNode insertRoot(Comparable x, BSTreeNode node) {  
    if (node == null)  
        node = new BSTreeNode(x);  
    else if (x.compareTo(node.key) < 0) {  
        node.left = insertRoot(x, node.left);  
        node = rightRotation(node);  
    }  
    else if (x.compareTo(node.key) > 0) {  
        node.right = insertRoot(x, node.right);  
        node = leftRotation(node);  
    }  
    else  
        ; // duplicate; do nothing or throw exception  
    return node;  
} // insertRoot
```

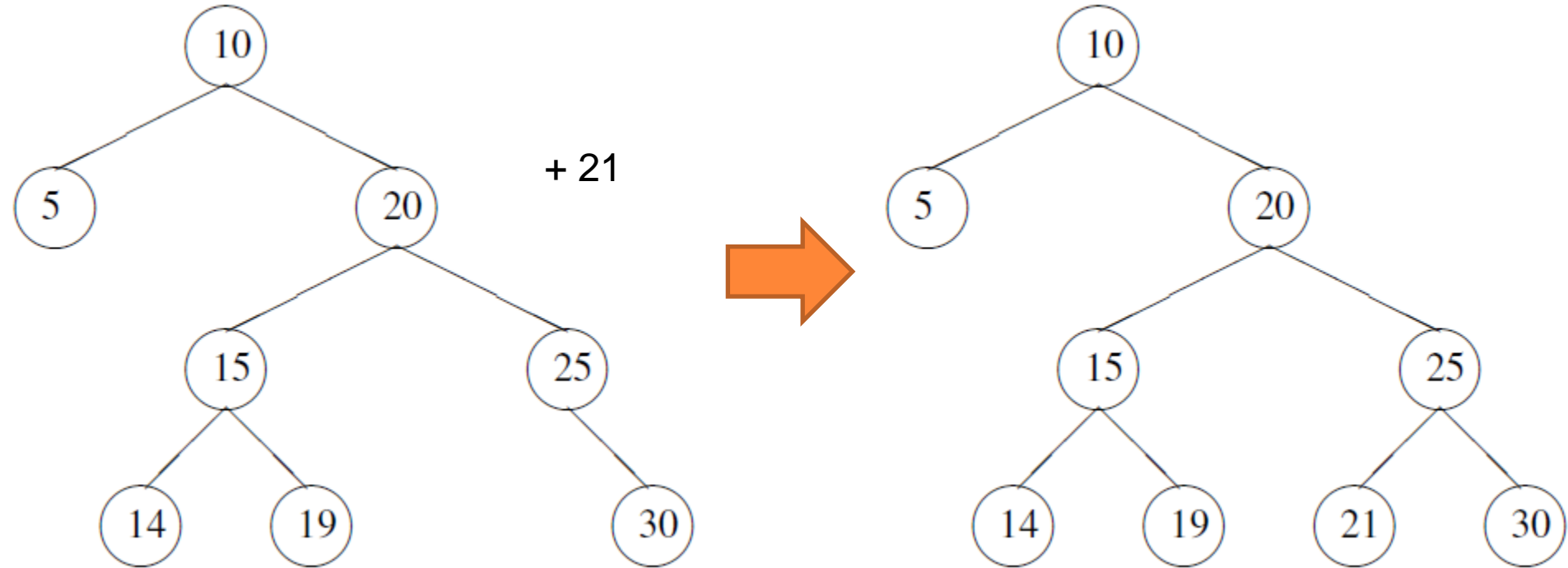
Rekurzija gre najprej v globino, in element se doda kot list.

Pri vračanju i rekurzije se izvaja zaporedje rotacij, ki dvigne element v koren.

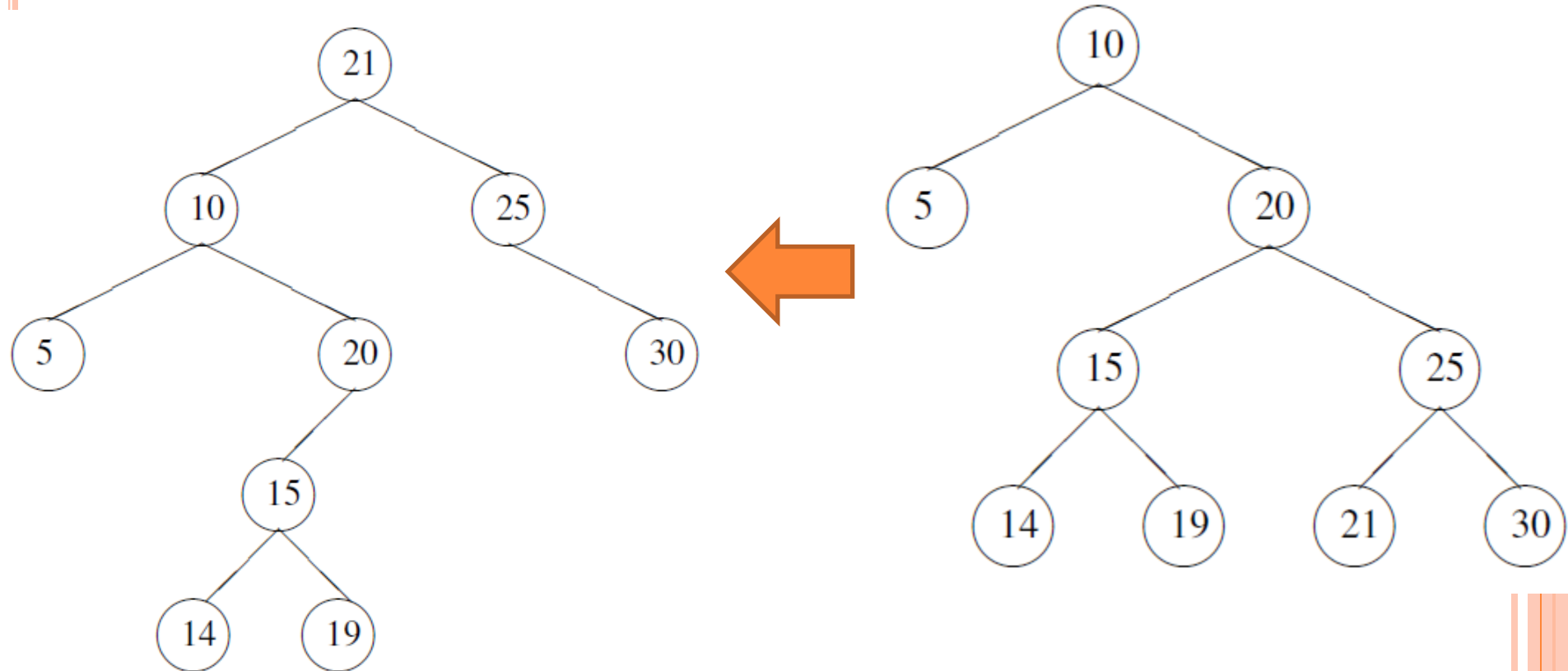
Časovna zahtevnost je sorazmerna višini drevesa.



PRIMER DODAJANJA V KOREN



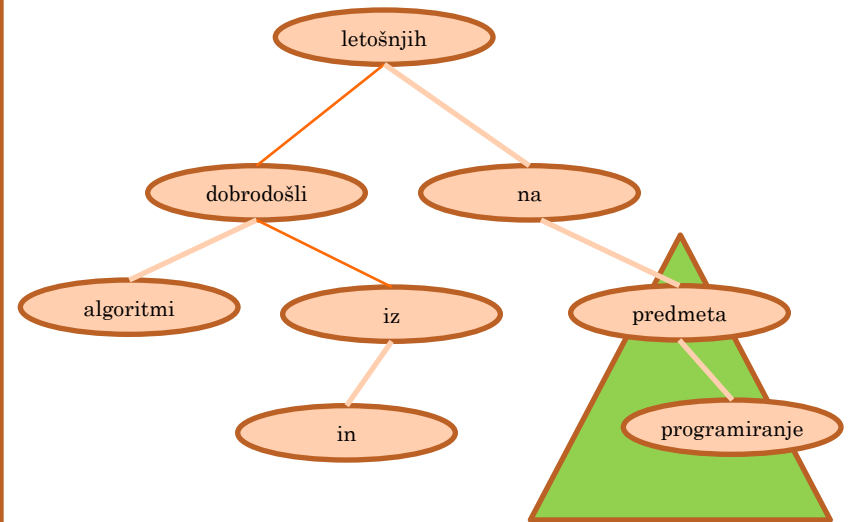
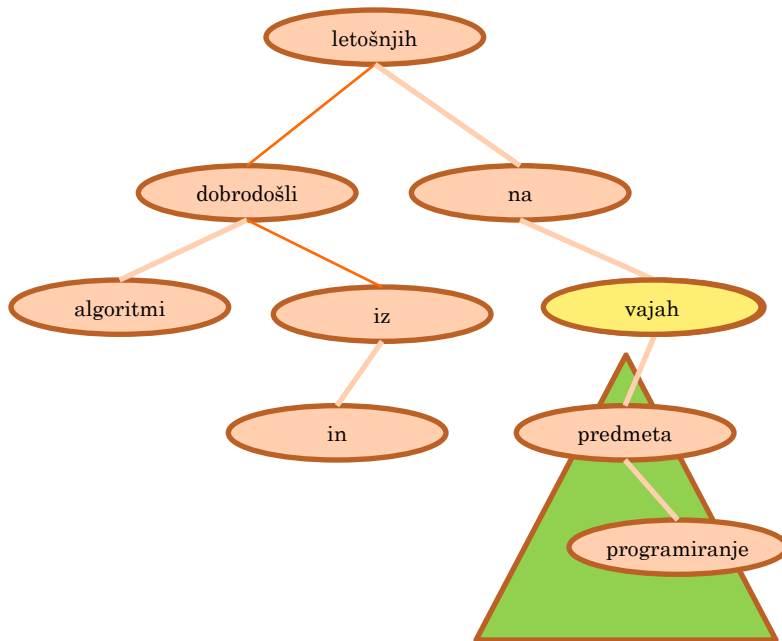
PRIMER DODAJANJA V KOREN



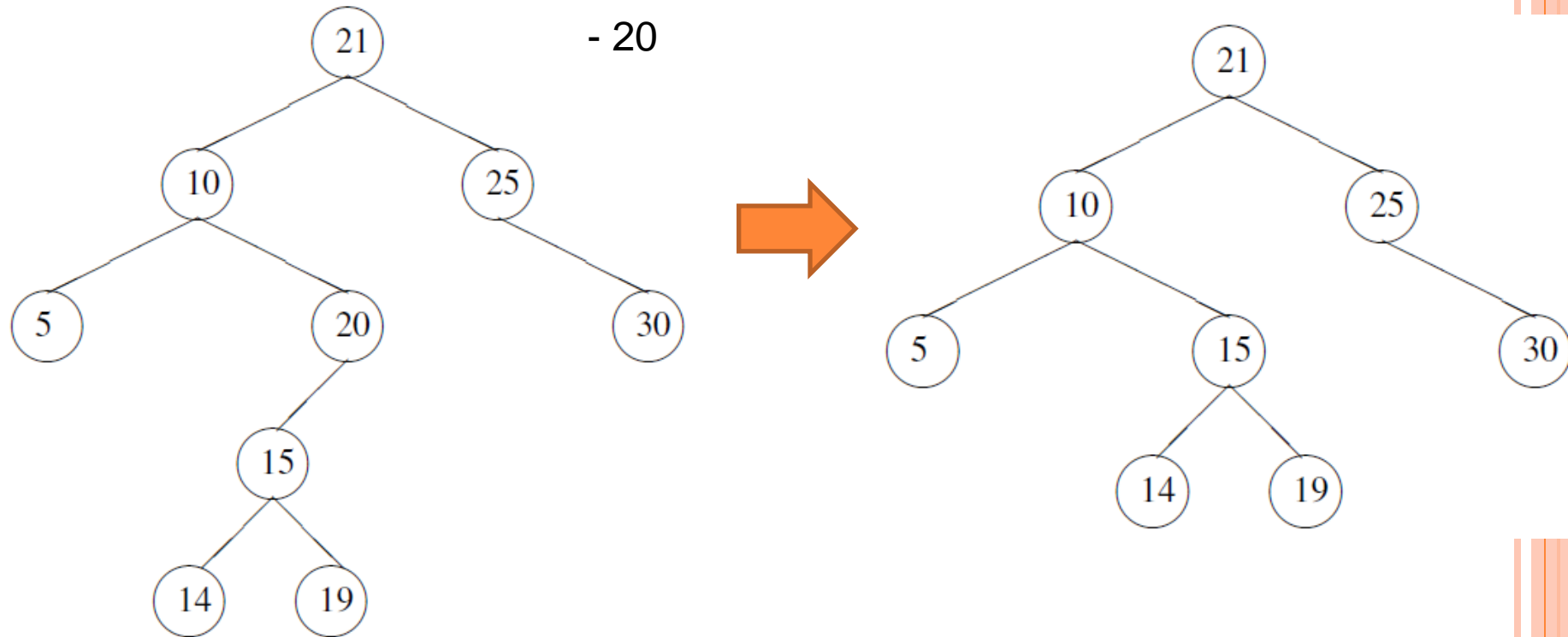
BRISANJE ELEMENTA IZ BST

Brisanje elementa, ki ima samo enega sina

Brisanje ključa “vajah”



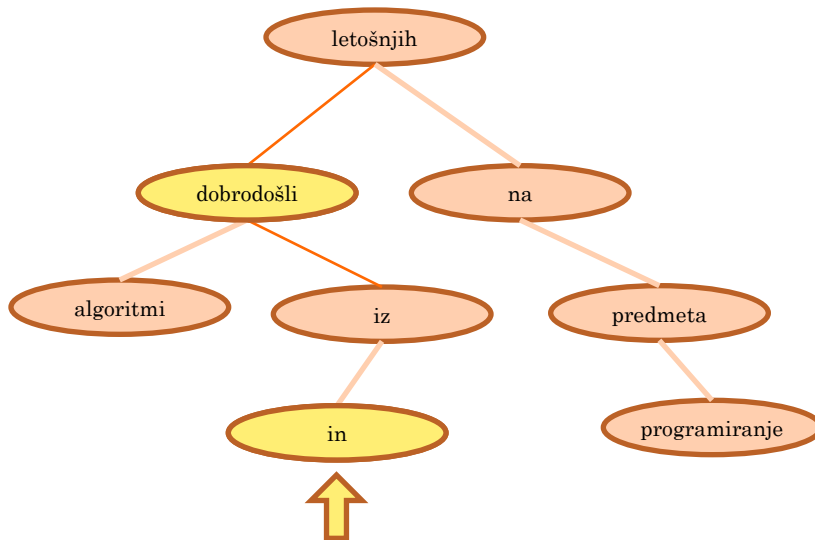
BRISANJE ELEMENTA IZ BST



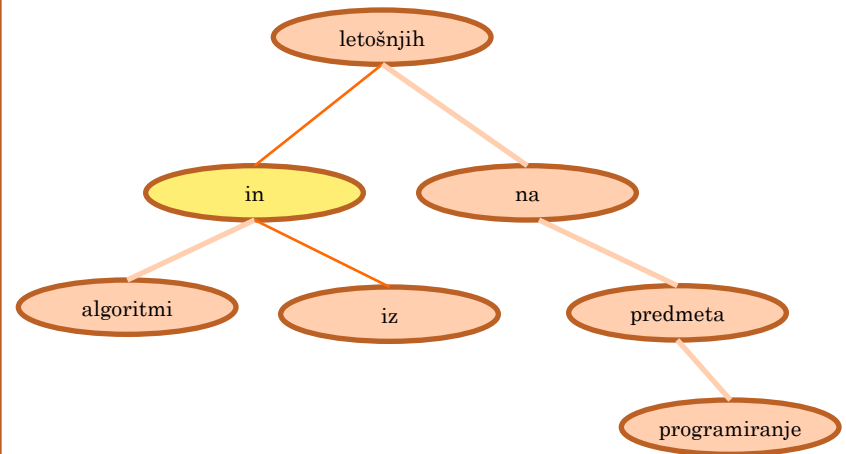
BRISANJE ELEMENTA IZ BST

Brisanje elementa, ki ima oba sina

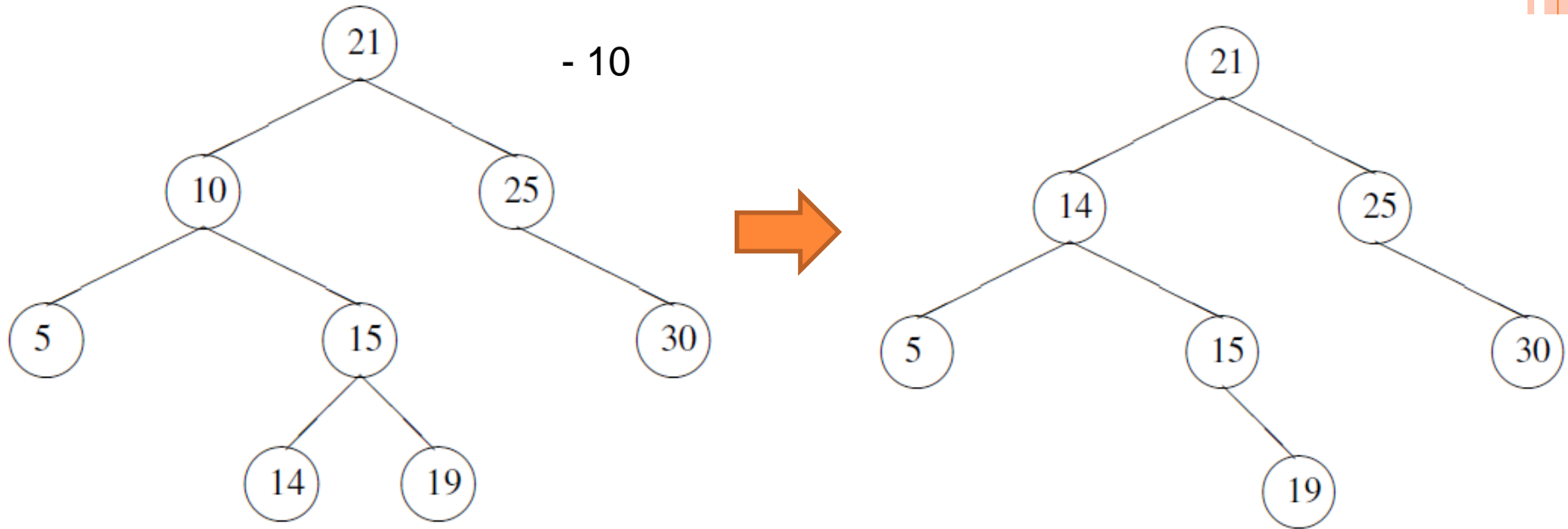
Brisanje ključa “dobrodošli”



najmanjši element desnega poddrevesa



BRISANJE ELEMENTA IZ BST



// za prenos minimalnega kljuca iz desnega poddrevesa

private Comparable minNodeKey ;

```
public BSTreeNode delete(Comparable x, BSTreeNode node) {  
    if (node != null) {  
        if (x.compareTo(node.key) == 0) { // delete node  
            if (node.left == null) // empty left  
                node = node.right;  
            else if (node.right == null) // empty right  
                node = node.left;  
            else {  
                node.right = deleteMin(node.right); // delete min from right  
                node.key = minNodeKey; // put it into the node  
            }  
        }  
        else if (x.compareTo(node.key) < 0)  
            node.left = delete(x, node.left);  
        else  
            node.right = delete(x, node.right);  
    } // if (node != null)  
    return node;  
} // delete
```



BRISANJE MINIMALNEGA

```
private BSTreeNode deleteMin(BSTreeNode node) {  
    if (node.left != null) {  
        node.left = deleteMin(node.left); // ohranjamo strukturo  
        return node;  
    }  
    else {  
        minNodeKey = node.key; // prenos kljuca  
        return node.right; // ohrani desno poddrevo  
    }  
} // deleteMin
```



BRISANJE ELEMENTA IZ BST

Element najprej poiščemo.

Zatem ga nadomestimo z minimalnim elementom v desnem poddrevesu (lahko tudi z maksimalnim v levem poddrevesu).



Časovna zahtevnost je sorazmerna višini drevesa.

Višina drevesa:

Poravnano: $O(\log n)$

Izrojeno: $O(n)$



ZAKLJUČEK

V praksi je časovna zahtevnost operacij na BST v povprečju sprejemljiva - $O(\log n)$.

V najslabšem primeru se BST lahko izrodi v seznam, kar pomeni časovno zahtevnost operacij reda $O(n)$.

V bolj zahtevnih aplikacijah je potrebno uporabljati (približno) poravnana drevesa.

IZZIV: Kako izrodimo BST?

Na koliko načinov ga lahko izrodimo?

Sestavite splošni algoritem, ki zgradi (naključno izbrano) izrojeno BST.