Stabla

Definicija

Stablo (engl. tree) je konačan neprazan skup čvorova za koga važi:

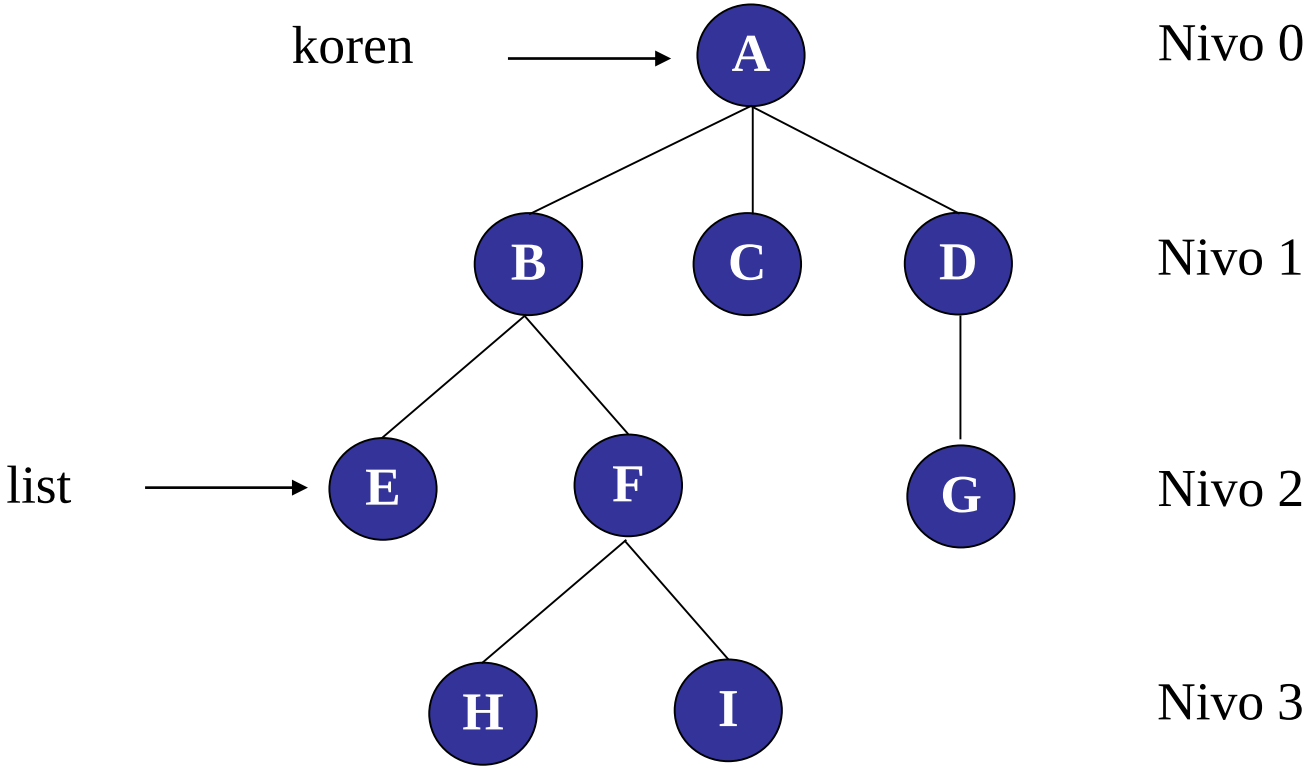
1. Postoji jedan specijalan čvor koji se naziva *koren* (engl. root) stabla.

2. Ostali čvorovi mogu se podeliti u m **disjunktnih podskupova** $(m \geq 0)$ **, od kojih svaki prestavlja stablo.**

Termini

- **stepen čvora** - broj podstabala svakog čvora
- **eksterni čvor**, **terminalni čvor** ili **list** - čvor nultog stepena
- **neterminalni čvor** - čvor stepena ≥ 1
- **nivo** ili **dubina** – nivo u odnosu na koren stabla
- **visina stabla** – max. nivo + 1
- **moment stabla** - broj čvorova u stablu,
- **težina stabla** - broj listova
- **uređeno stablo** – stablo kod koga je bitna relativna uređenost podstabala u svakom čvoru

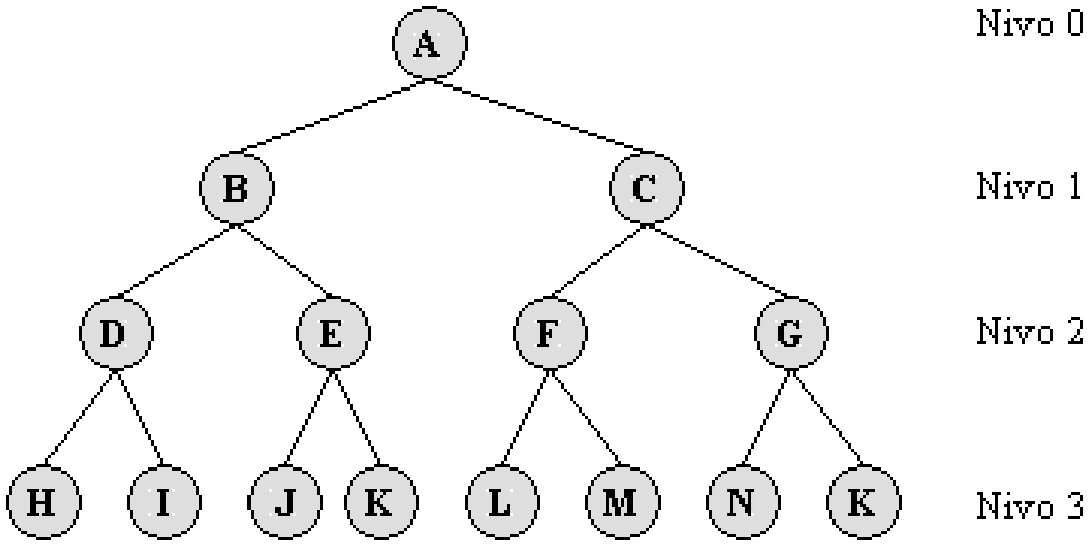
Primer stabla



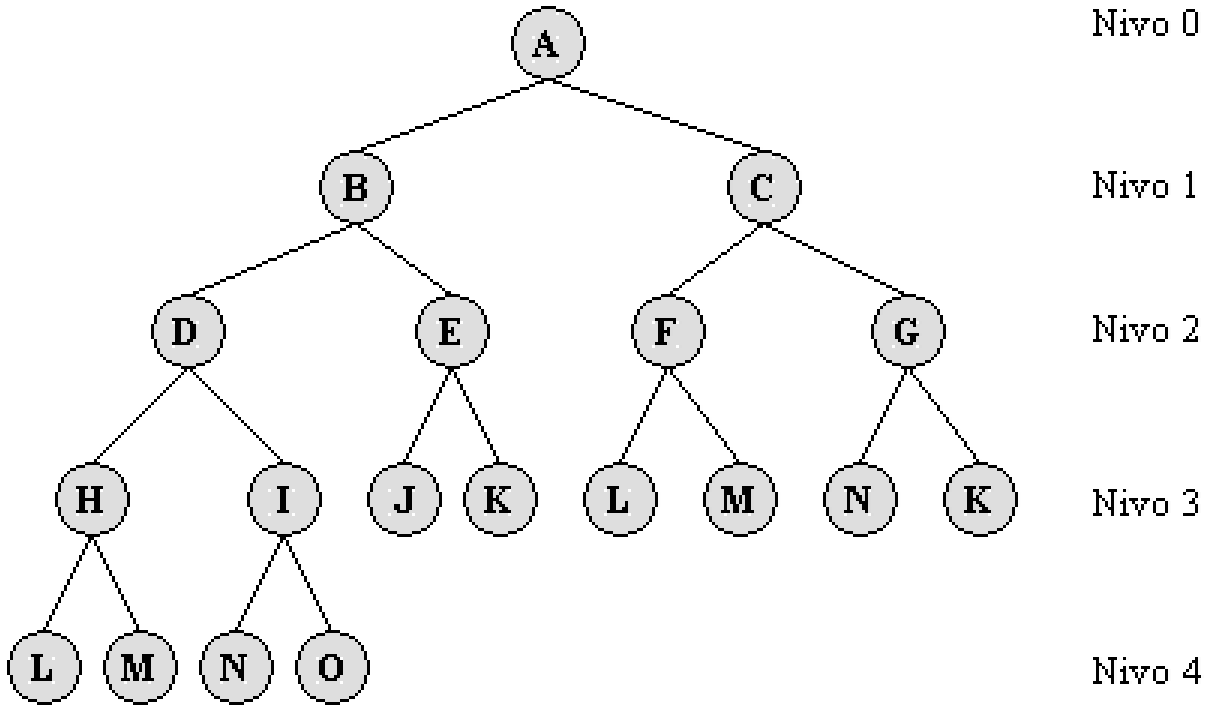
Binarna stabla

- **binarno stablo** - stablo kod koga svaki čvor ima 0, 1 ili 2 direktna sledbenika
- **prošireno** ili **striktno binarno stablo** - binarno stablo koje ima u svakom čvoru 0 ili 2 sledbenika
- **potpuno binarno stablo** - svi nivoi maksimalno popunjeni
- **gotovo potpuno** - binarno stablo kod koga su svi nivoi, osim poslednjeg, potpuno popunjeni, dok je poslednji nivo delimično popunjen sleva udesno
- **uređeno binarno stablo** - binarno stablo kod koga svaki čvor ima svojstvo da je njegova vrednost veća od svih vrednosti u čvorovima levog podstabla, a manja (ili jednaka) od svih vrednosti u čvorovima desnog podstabla
- **uravnoteženo** ili **balansirano binarno stablo** - binarno stablo kod koga se visina levog i desnog podstabla bilo kog čvora razlikuje najviše za jedan
- **perfektno uravnoteženo stablo** - uravnoteženo binarno stablo kod koga su svi listovi na jednom ili dva nivoa.

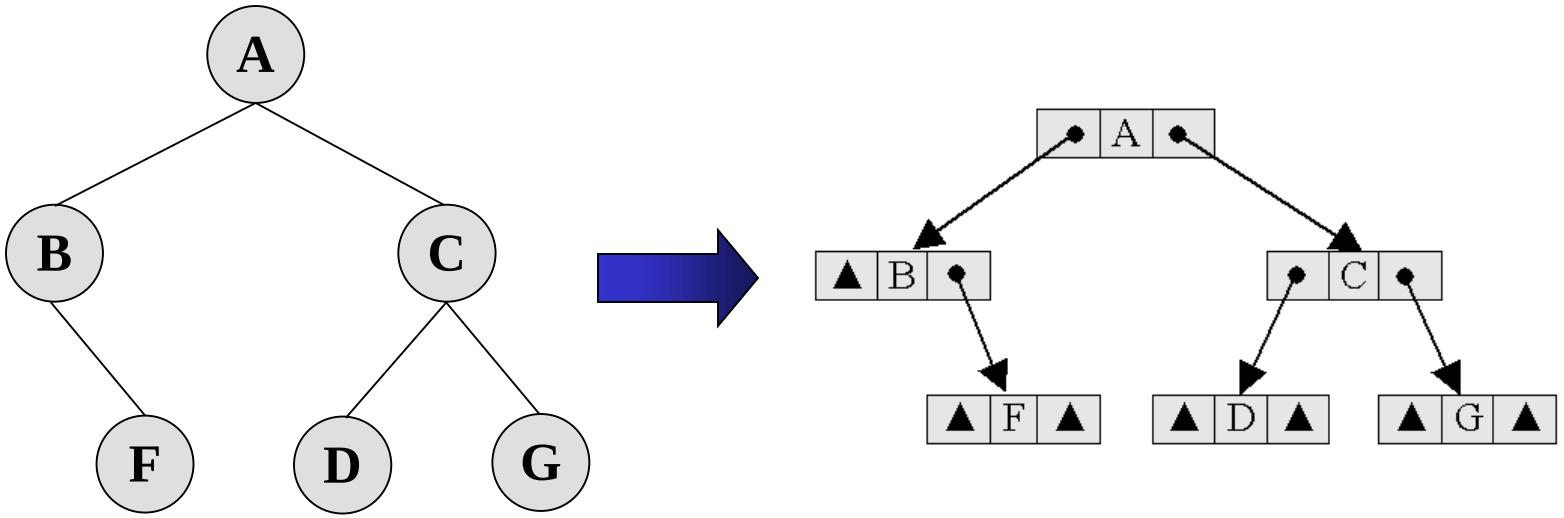
Potpuno binarno stablo sa 4 nivoa (visine 4)



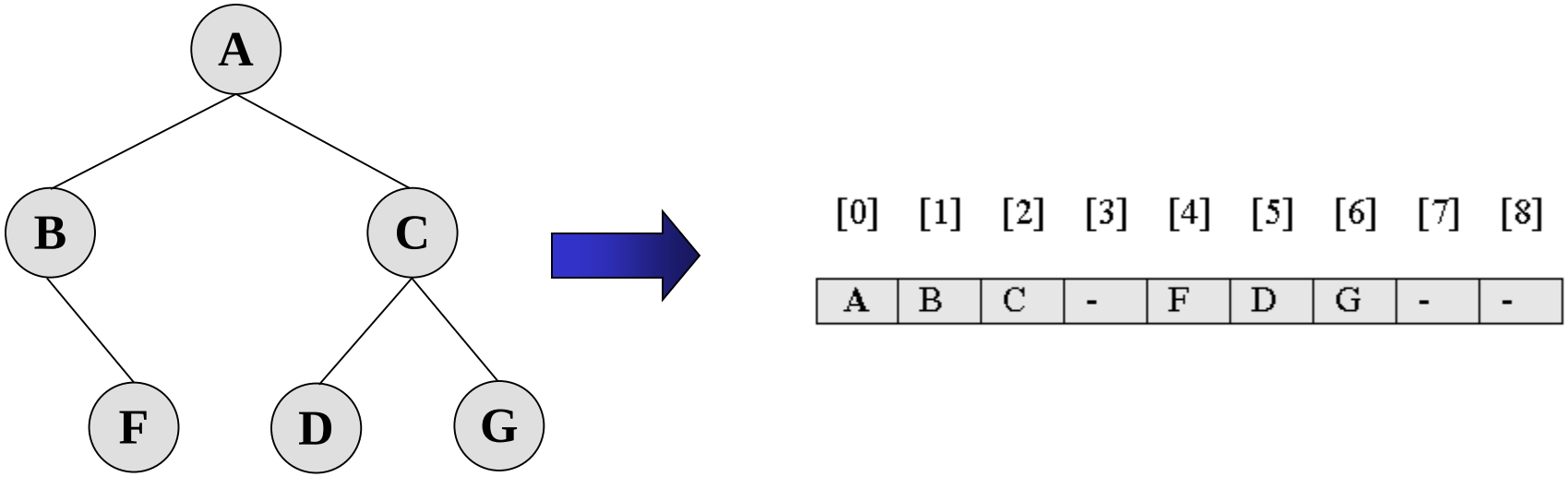
Gotovo potpuno binarno stablo sa 5 nivoa (visine 5)



Memorijska reprezentacija - Dinamička



Memorijska reprezentacija - Statička



U slučaju binarnog stabla ukoliko je roditeljski čvor smešten na poziciji sa indeksom **i**, njegov levi potomak smešten je na poziciji sa indeksom **2i+1**, a desni na poziciji sa indeksom **2i+2**.

Osnovne operacije

- **insert** - dodaje novi čvor u stablo,
- **delete** - briše čvor iz stabla,
- **search** - pronalazi željeni čvor u stablu,
- **traverse** - obilazi sve čvorove stabla,
  - **depthFirst** - obilazak po dubini:
    - **preorder**,
    - **inorder,**
    - **postorder,**
  - **breadthFirst** - obilazak po širini.

Čvor binarnog stabla

```
template <class T>
class BSTNode
{
public:
        T key;
        BSTNode<T> *left, *right;
public:
        BSTNode() { left = right = NULL;};
        BSTNode(int el) { key = el; left = right = NULL;};
        BSTNode(int el, BSTNode<T>* lt, BSTNode<T>* rt)
        {
                key = el; left = lt; right = rt;
        }
        bool isLT(T el){
                if(key < el) return true;
                else return false;
        };
        bool isGT(T el){
                if(key > el) return true;
                else return false;
        };
        bool isEQ(T el){
                if(key == el) return true;
                else return false;
        };
        void setKey(T el){ key = el; };
        T getKey() {return key;};
        void visit() { cout << key << " ";};
};
```

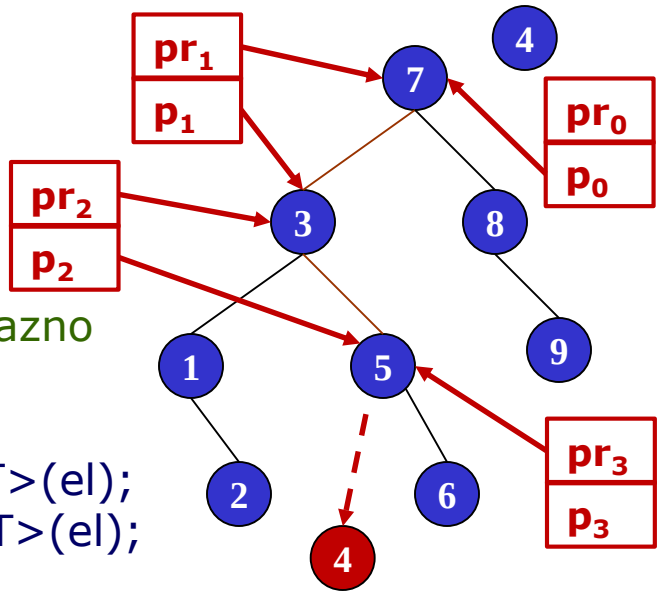
Binarno stablo

```
template <class T>
class BSTree
{
protected:
        BSTNode<T>* root;
        long numOfElements;
public:
        BSTree() { root = NULL; numOfElements=0;};
        ~BSTree() { deleteTree(root); };
        void deleteTree(BSTNode<T>* p);
        bool isEmpty() { return root == NULL;};
        void insert(T el);
        bool isInTree(T el) { return search(el) != NULL; };
        BSTNode<T>* search(T el) {  return search(root,el); };
        BSTNode<T>* search(BSTNode<T>* p, T el);
        void balance(int data[], int first, int last);
        void deleteByCopying(T el);
        void deleteByMerging(T el);
        void preorder() { preorder(root); };
        void inorder() { inorder(root); };
        void postorder() { postorder(root); };
        void inorder(BSTNode<T>* p);
        void preorder(BSTNode<T>* p);
        void postorder(BSTNode<T>* p);
        void breadthFirst();
        void iterativePreorder();
        void iterativeInorder();
        void iterativePostorder();
};
```

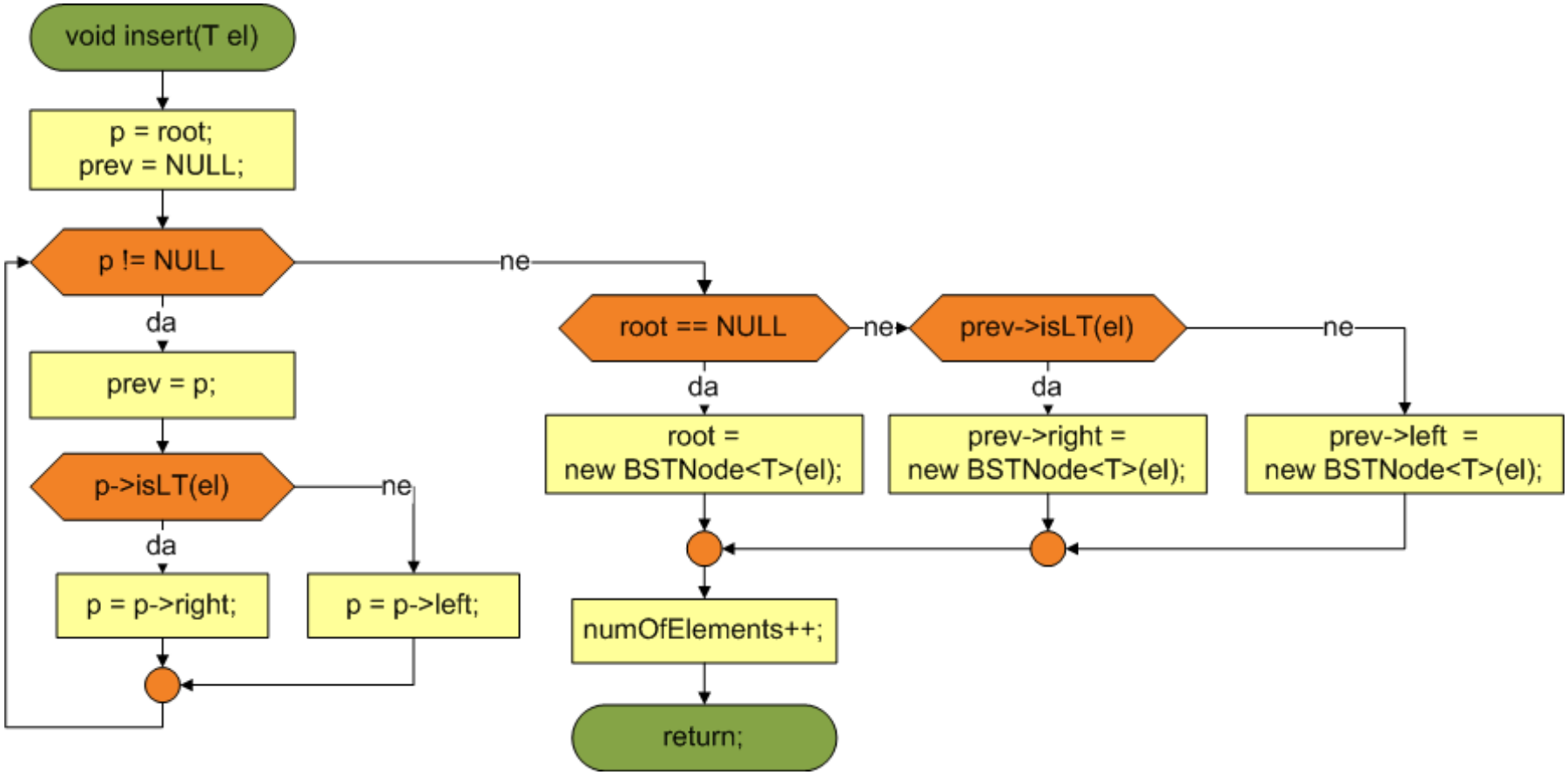
Umetanje u uređeno binarno stablo

```
template <class T>
void BSTree<T>::insert(T el)
{
        BSTNode<T>* p = root, *prev = NULL;
        while (p != NULL) {  // trazenje mesta za umetanje novog cvora
            prev = p;
            if (p->isLT(el))
                p = p->right;
            else
                p = p->left;
        }
        if (root == NULL)    // stablo je prazno
             root = new BSTNode<T>(el);
        else if (prev->isLT(el))
             prev->right = new BSTNode<T>(el);
        else prev->left  = new BSTNode<T>(el);
             numOfElements++;
}
```

insert(4)
it=0: pr->X | p->7 | 7>6
it=1: pr->7 | p->3 | 3<6
it=2: pr->3 | p->5 | 5<6
it=3: pr->5 | p->X



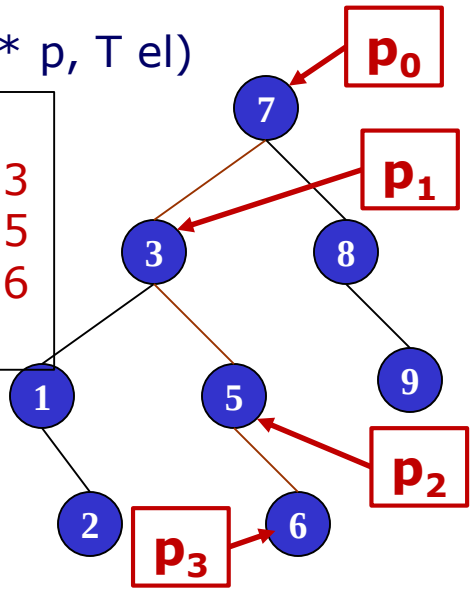
Umetanje u uređeno binarno stablo



Traženje elementa u uređenom binarnom stablu

```
template <class T>
BSTNode<T>* BSTree<T>::search(BSTNode<T>* p, T el)
{
    while (p != NULL)
        if (p->isEQ(el))
            return p;
        else if (p->isGT(el))
            p = p->left;
        else
            p = p->right;
    return NULL;
}
```

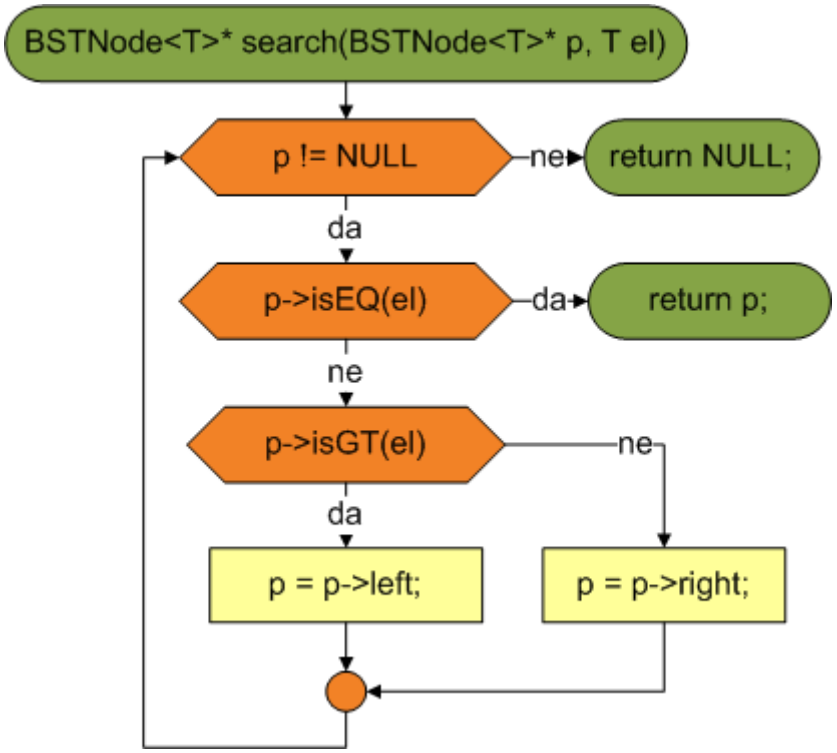
search(6)
it=0: p->7 | 7>6 | p->3
it=1: p->3 | 3<6 | p->5
it=2: p->5 | 5<6 | p->6
it=3: p->6 | 6=6



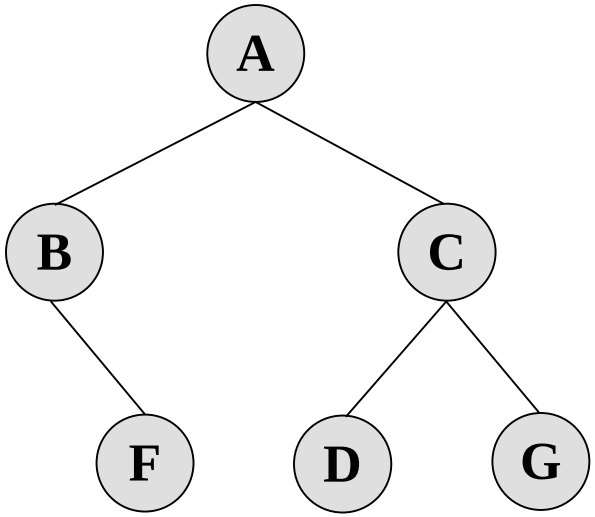
Brisanje stabla

```
template <class T>
void BSTree<T>::deleteTree(BSTNode<T>* p)
{
    if (p != NULL) {
        deleteTree(p->left);
        deleteTree(p->right);
        delete p;
    }
}
```

Traženje elementa u uređenom binarnom stablu



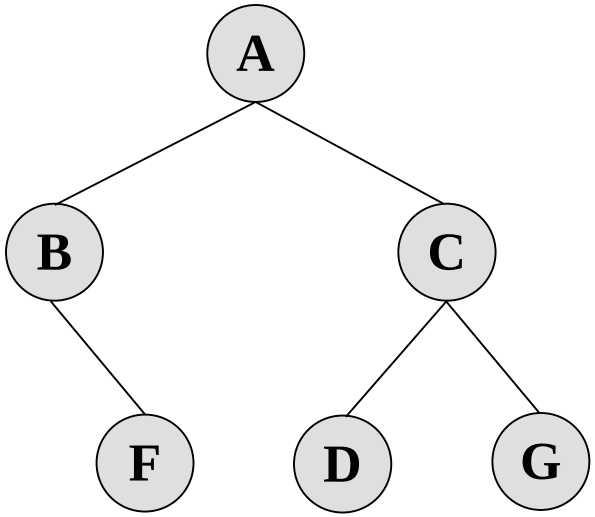
Preorder




```
template <class T>
void BSTree<T>::preorder(BSTNode<T>* p)
{
   if (p != NULL) {
        p->visit();
        preorder(p->left);
        preorder(p->right);
   }
}
```

A B F C D G

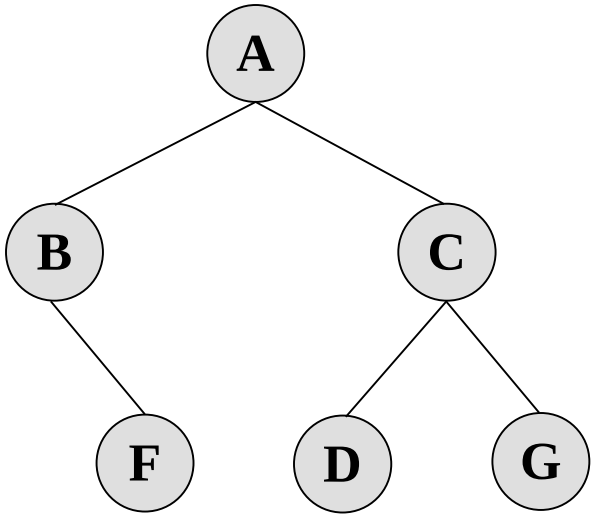
Inorder




```
template <class T>
void BSTree<T>::inorder(BSTNode<T>* p)
{
   if (p != NULL) {
        inorder(p->left);
        p->visit();
        inorder(p->right);
   }
}
```

B F A D C G

Postorder

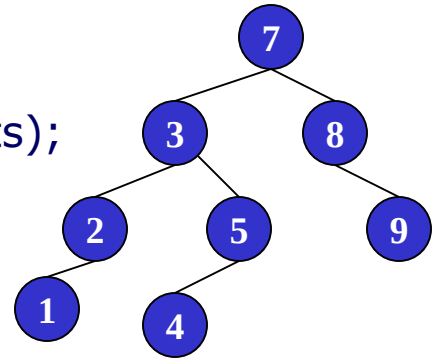



```
template <class T>
void BSTree<T>::postorder(BSTNode<T>* p)
{
   if (p != NULL) {
        postorder(p->left);
        postorder(p->right);
        p->visit();
   }
}
```

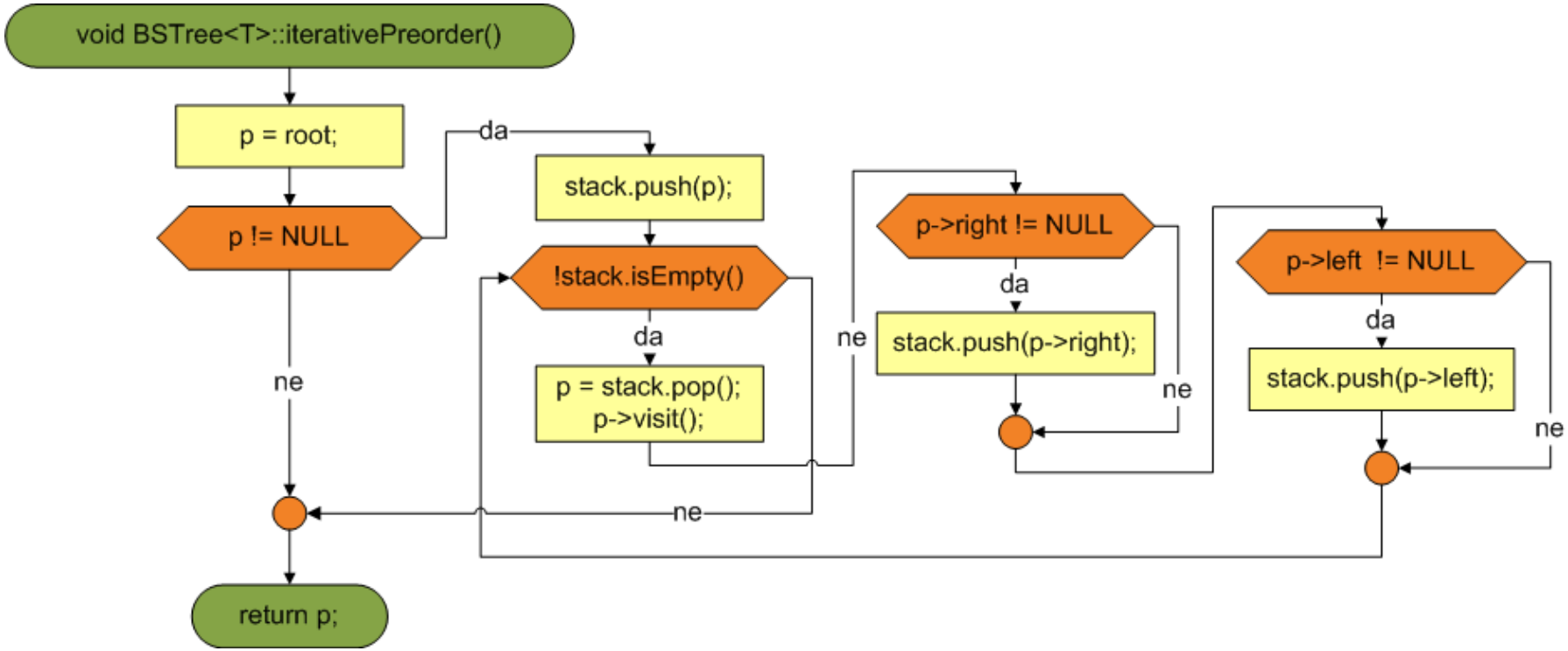
F B D G C A

Iterativni preorder

```
template <class T>
void BSTree<T>::iterativePreorder() {
    BSTNode<T>* p = root;
    StackAsArray<BSTNode<T>*> stack(numOfElements);
    if (p != NULL) {
        stack.push(p);
        while (!stack.isEmpty()) {
            p = stack.pop();
            p->visit();
            if (p->right != NULL)        // levi potomak se stavlja u magacin
                stack.push(p->right);  // posle desnog, da bi se prvi obradio
            if (p->left  != NULL)
                stack.push(p->left);
        }
    }
}
```

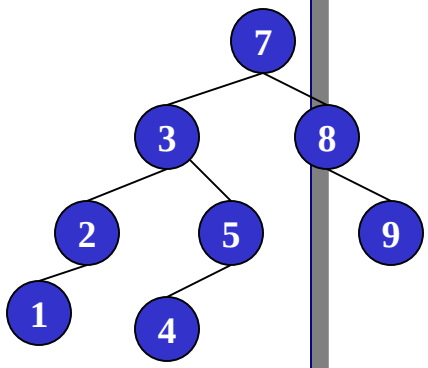


Iterativni preorder

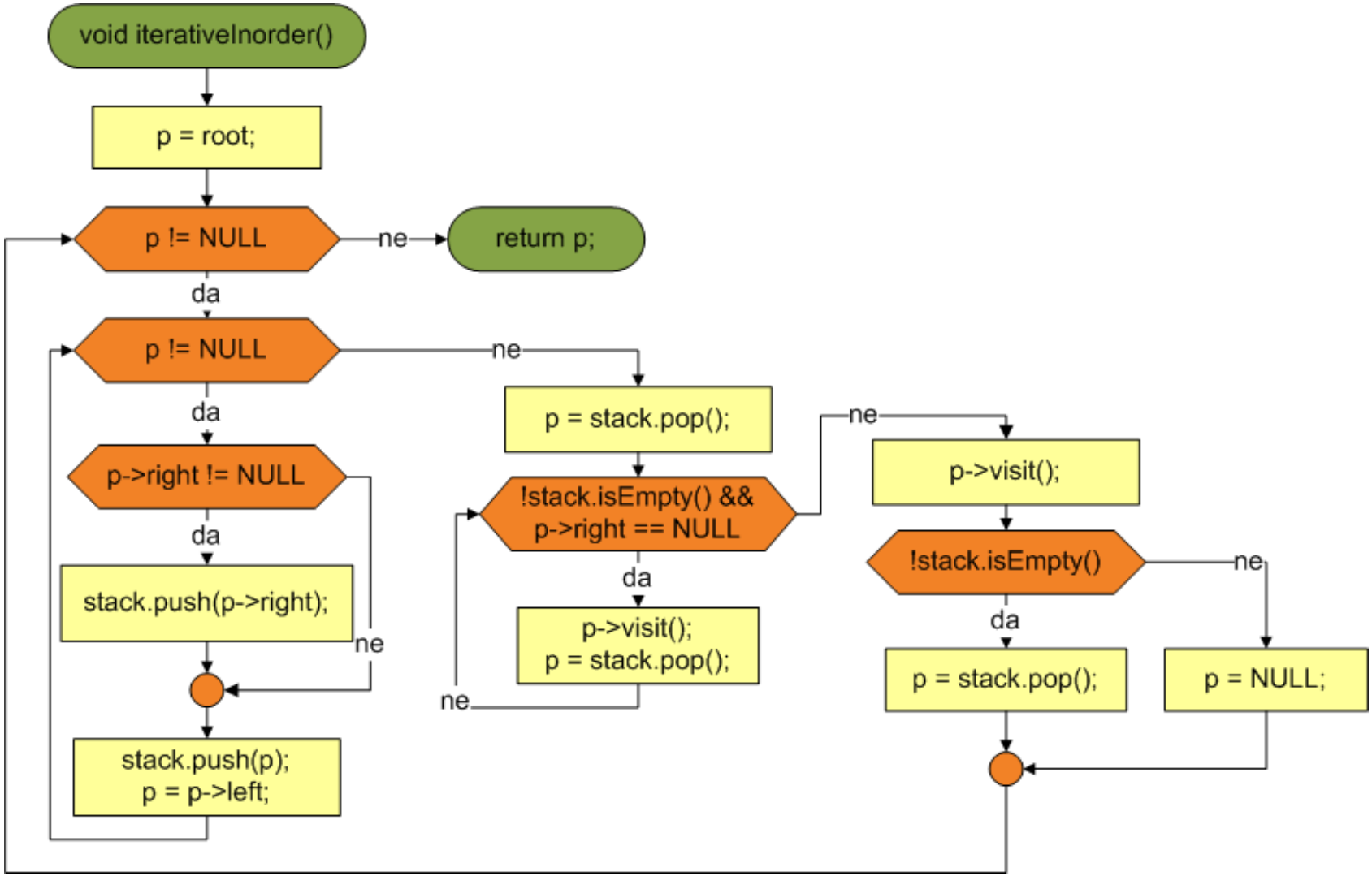


Iterativni inorder

```
template <class T>
void BSTree<T>::iterativeInorder()
{
    BSTNode<T>* p = root;
    StackAsArray<BSTNode<T>*> stack(numOfElements);
    while (p != NULL) {
        while(p != NULL) {
            if (p->right != NULL)
                stack.push(p->right);    // u magacin se stavlja prvo desni
            stack.push(p);               // pa zatim tekuci cvor
            p = p->left;                 // i prelazi na levog potomka
        }
        p = stack.pop();                 // cvor bez levih potomaka
        while (!stack.isEmpty() && p->right == NULL) {
            p->visit();                  // obilazak tekuceg cvora i ostalih
            p = stack.pop();             // bez desnih potomaka
        }
        p->visit();                      // obilazak prvog cvora
        if (!stack.isEmpty())            // i njegovih desnih potomaka
            p = stack.pop();
        else
            p = NULL;
    }
}
```

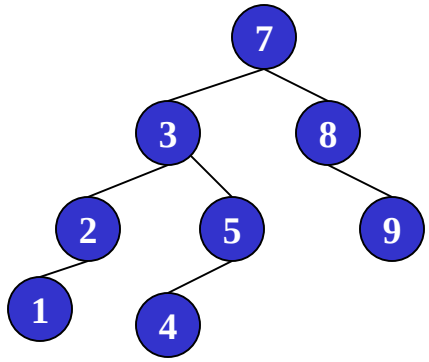


Iterativni inorder

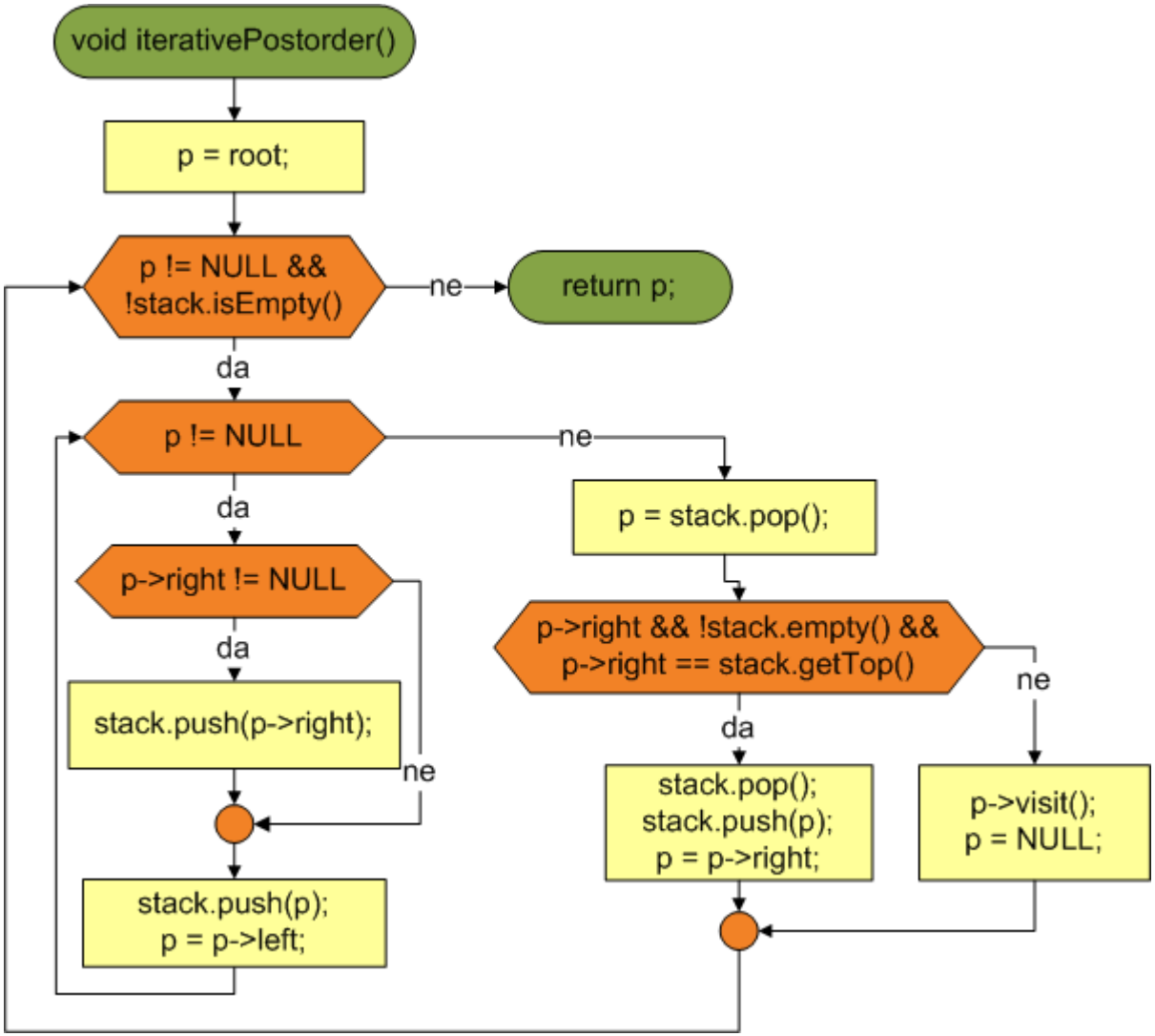


Iterativni postorder

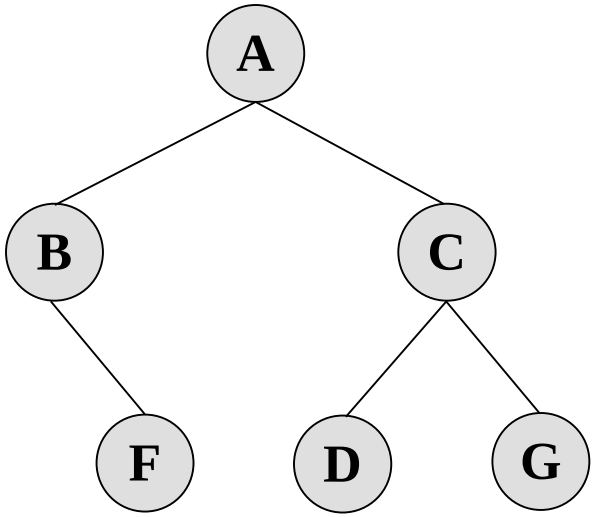
```
template <class T>
void BSTree<T>::iterativePostorder()
{
    BSTNode<T> *p = root, *q = root;
    StackAsArray<BSTNode<T>*> stack(numOfElements);
    while (p != NULL) {
        for ( ; p->left != NULL; p = p->left)
            stack.push(p);
        while (p != NULL && (p->right == NULL || p->right == q)) {
            p->visit(); // nema desnog potomka ili je desni potomak vec obradjen
            q = p;
            if (stack.isEmpty())
                return;
            p = stack.pop();
        }
        stack.push(p);
        p = p->right;
    }
}
```



Iterativni postorder



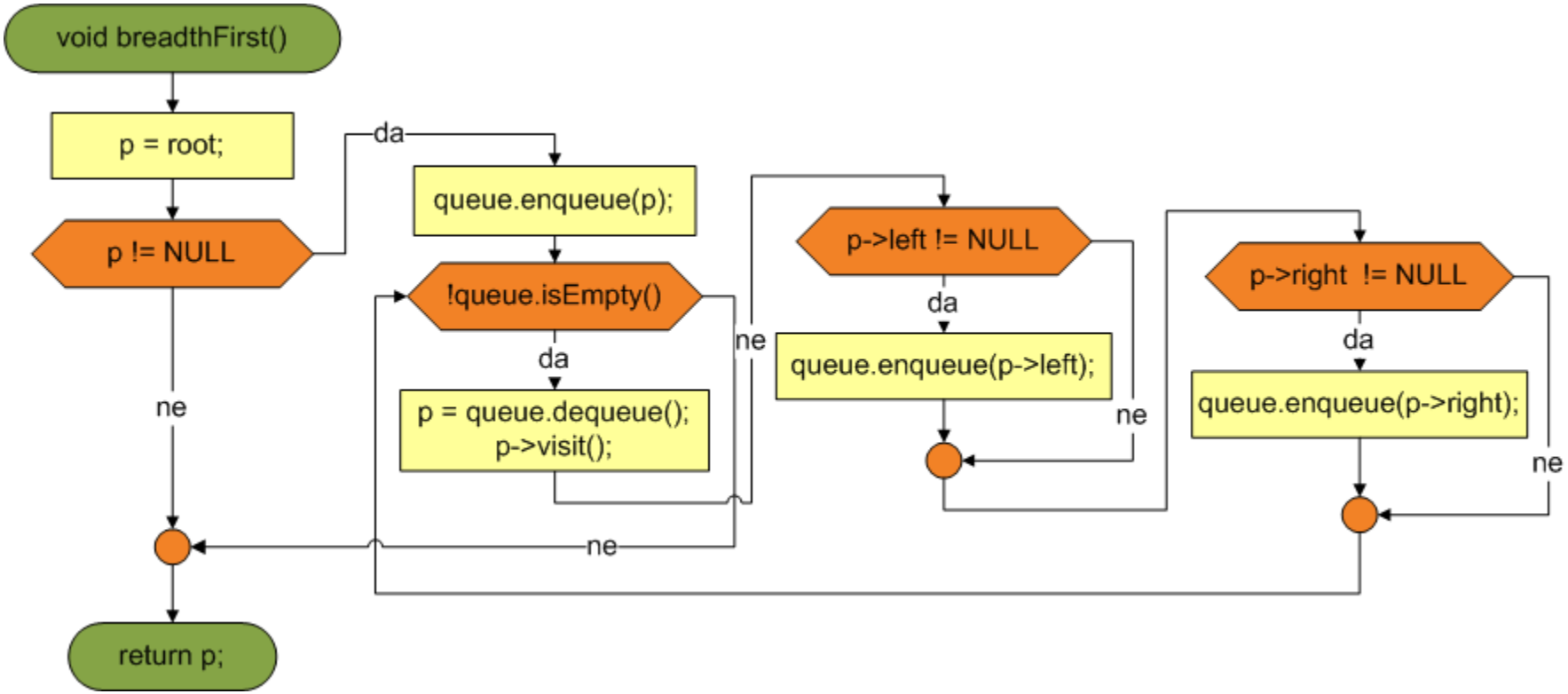
breadthFirst - obilazak po širini



A B C F D G

```
template <class T>
void BSTree<T>::breadthFirst()
{
   BSTNode<T>* p = root;
   QueueAsArray<BSTNode<T>*>
                              queue(numOfElements);
   if (p != NULL) {
          queue.enqueue(p);
          while (!queue.isEmpty()) {
                    p = queue.dequeue();
                    p->visit();
                    if (p->left != NULL)
                              queue.enqueue(p->left);
                    if (p->right != NULL)
                              queue.enqueue(p->right);
          }
   }
}
```

Obilazak stabla po širini



Brisanje čvora iz uređenog binarnog stabla

Postoje tri slučaja brisanja čvora iz uređenog binarnog stabla:

• čvor koji se briše je list - obzirom da nema potomaka jednostavno se ukljanja,

• čvor koji se briše ima jednog potomka (bilo levog ili desnog) - čvor se uklanja, a na njegovo mesto dolazi njegov potomak i

• čvor koji se briše ima oba potomka - najsloženiji slučaj koji se rešava jednim od dva algoritma: **brisanje kopiranjem** (**deleteByCopying**) ili **brisanje mešanjem** (**deleteByMerge**).

Brisanja kopiranjem

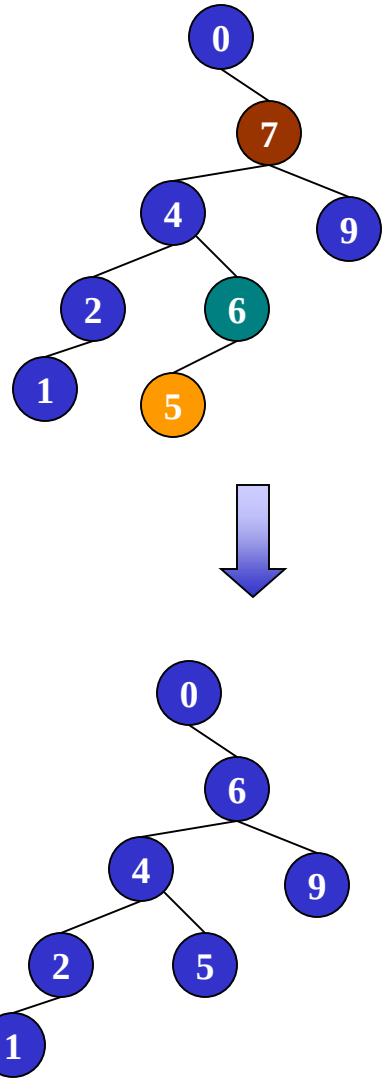
Neophodno je naći element koji će po vrednosti biti veći (ili jednak) od svih elemenata u levom podstablu, a manji od svih elemenata u desnom podstablu, kada bude postavljen na mesto obrisanog elementa. Kandidata za zamenu sa obrisanim elementom su:

- krajnji desni čvor levog podstabla i
- krajnji levi čvor desnog podstabla.

Jedan od ta dva čvora se briše sa svoje pozicije i premešta na poziciju čvora koji se uklanja iz stabla. Ukoliko je premešteni čvor imao jednog potomka, taj potomak prelazi na prethodnu poziciju premeštenog čvora.

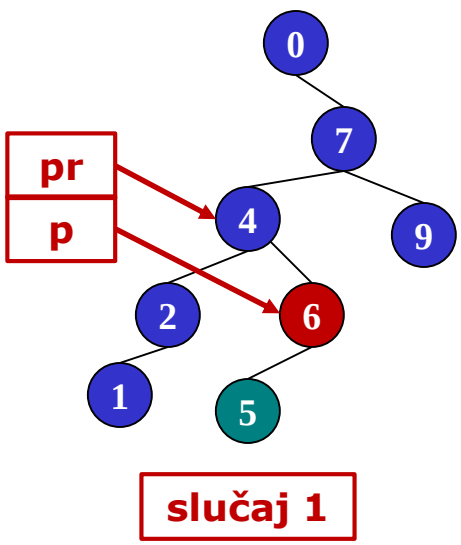
Brisanje kopiranjem

```
template <class T>
void BSTree<T>::deleteByCopying(T el)
{
   BSTNode<T> *node, *p = root, *prev = NULL;
   while (p != NULL && !p->isEQ(el)) {     // nalazenje cvora sa zeljenim el.
       prev = p;
       if (p->isLT(el)) p = p->right;
       else p = p->left;
   }
   node = p;
   if (p != NULL && p->isEQ(el)) {
       if (node->right == NULL)             // cvor nema desnog potomka (1)
           node = node->left;
       else if (node->left == NULL)         // cvor nema levog potomka (2)
           node = node->right;
       else {                                // cvor ima oba potomka (3)
           BSTNode<T>* tmp = node->left;
           BSTNode<T>* previous = node;
           while (tmp->right != NULL) {     // nalazenje krajnjeg desnog cvora
              previous = tmp;               // u levom podstablu
              tmp = tmp->right;
           }
           node->setKey(tmp->getKey());      // prepisivanje reference na kljuc
           if (previous == node)             // tmp je direktni levi potomak node-a
                 previous->left  = tmp->left;  // ostaje isti raspored u levom podstablu
           else previous->right = tmp->left;  // levi potomak tmp-a
           delete tmp;                       // se pomera na mesto tmp-a
           numOfElements--;
           return;
       }
      if (p == root)                         // u slucaju (1) i (2) samo je pomerena
          root = node;                       // referenca node pa je potrebno
      else if (prev->left == p)              // izmeniti i link prethodnog cvora
          prev->left = node;                 // u slucaju (3) ovo nema dejstva
      else prev->right = node;
    delete p;
    numOfElements--;
   }
   else if (root != NULL)     throw new SBPException("Element is not in the tree!");
   else throw new SBPException("The tree is empty!");
}
```

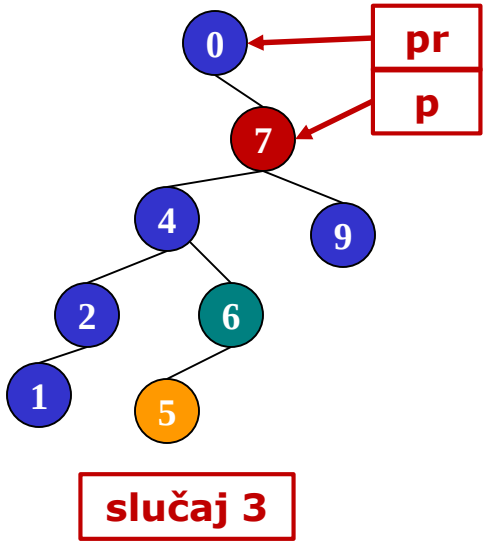


Brisanje kopiranjem – 1.deo

```
template <class T>
void BSTree<T>::deleteByCopying(T el)
{
    BSTNode<T> *node, *p = root, *prev = NULL;
    while (p != NULL && !p->isEQ(el)) { // nalazenje cvora sa zeljenim el.
        prev = p;
        if (p->isLT(el))
            p = p->right;
        else
            p = p->left;
    }
    node = p;
...
```



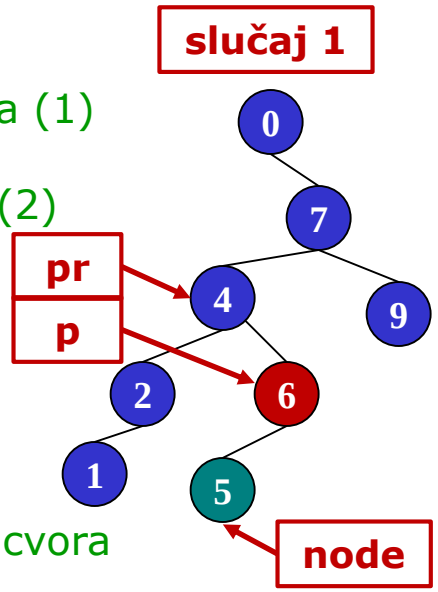
slučaj 1



slučaj 3

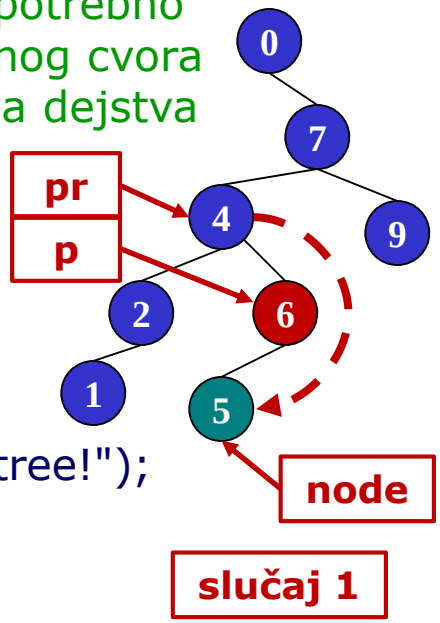
Brisanje kopiranjem – 2.deo

```
...
node = p;
 if (p != NULL && p->isEQ(el)) {
      if (node->right == NULL)          // cvor nema desnog potomka (1)
           node = node->left;
      else if (node->left == NULL)      // cvor nema levog potomka (2)
           node = node->right;
      else                               // cvor ima oba potomka (3)
      {
           BSTNode<T>* tmp = node->left;
           BSTNode<T>* previous = node;
           while (tmp->right != NULL)
           {                                  // nalazenje krajnjeg desnog cvora
              previous = tmp                  // u levom podstablu
              tmp = tmp->right;
           }
           node->setKey(tmp->getKey());       // prepisivanje reference na kljuc
           if (previous == node)              // tmp je direktni levi potomak node-a
                 previous->left  = tmp->left; // ostaje isti raspored u levom podstablu
           else
                 previous->right = tmp->left; // levi potomak tmp-a
           delete tmp;                        // se pomera na mesto tmp-a
           numOfElements--;
           return;
      }
...
```



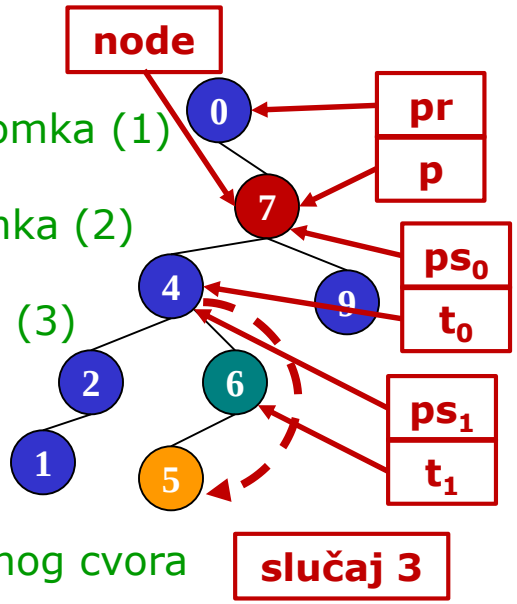
Brisanje kopiranjem – 3.deo

```
...
        if (p == root)                  // u slucaju (1) i (2) samo je pomerena
            root = node;                // referenca node pa je potrebno
        else if (prev->left == p)       // izmeniti i link prethodnog cvora
            prev->left = node;          // u slucaju (3) ovo nema dejstva
        else
            prev->right = node;
        delete p;
        numOfElements--;
    }
    else if (root != NULL)
            throw new SBPException("Element is not in the tree!");
    else
            throw new SBPException("The tree is empty!");
}
```



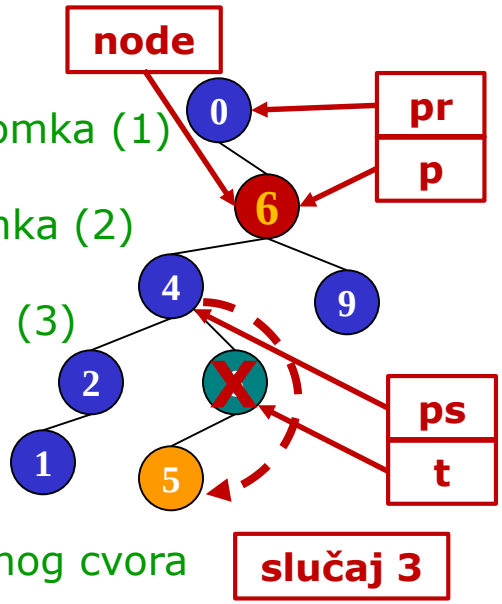
Brisanje kopiranjem – 2.deo

```
...
   node = p;
    if (p != NULL && p->isEQ(el)) {
        if (node->right == NULL)          // cvor nema desnog potomka (1)
            node = node->left;
        else if (node->left == NULL)      // cvor nema levog potomka (2)
            node = node->right;
        else                               // cvor ima oba potomka (3)
        {
            BSTNode<T>* tmp = node->left;
            BSTNode<T>* previous = node;
            while (tmp->right != NULL)
            {                              // nalazenje krajnjeg desnog cvora
                previous = tmp             // u levom podstablu
                tmp = tmp->right;
            }
            node->setKey(tmp->getKey());          // prepisivanje reference na kljuc
            if (previous == node)                 // tmp je direktni levi potomak node-a
                  previous->left  = tmp->left;    // ostaje isti raspored u levom podstablu
            else
                  previous->right = tmp->left;    // levi potomak tmp-a
            delete tmp;                           // se pomera na mesto tmp-a
            numOfElements--;
            return;
        }
...
```



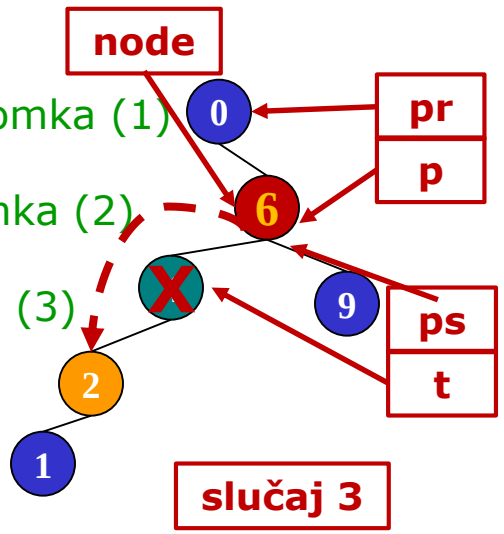
Brisanje kopiranjem – 2.deo

```
...
   node = p;
    if (p != NULL && p->isEQ(el)) {
        if (node->right == NULL)          // cvor nema desnog potomka (1)
            node = node->left;
        else if (node->left == NULL)      // cvor nema levog potomka (2)
            node = node->right;
        else                               // cvor ima oba potomka (3)
        {
            BSTNode<T>* tmp = node->left;
            BSTNode<T>* previous = node;
            while (tmp->right != NULL)
            {                              // nalazenje krajnjeg desnog cvora
                previous = tmp             // u levom podstablu
                tmp = tmp->right;
            }
            node->setKey(tmp->getKey());          // prepisivanje reference na kljuc
            if (previous == node)                 // tmp je direktni levi potomak node-a
                  previous->left  = tmp->left;    // ostaje isti raspored u levom podstablu
            else
                  previous->right = tmp->left;    // levi potomak tmp-a
            delete tmp;                           // se pomera na mesto tmp-a
            numOfElements--;
            return;
        }
...
```



Brisanje kopiranjem – 2.deo

```
...
   node = p;
    if (p != NULL && p->isEQ(el)) {
        if (node->right == NULL)          // cvor nema desnog potomka (1)
            node = node->left;
        else if (node->left == NULL)      // cvor nema levog potomka (2)
            node = node->right;
        else                               // cvor ima oba potomka (3)
        {
            BSTNode<T>* tmp = node->left;
            BSTNode<T>* previous = node;
            while (tmp->right != NULL)
            {                              // nalazenje krajnjeg desnog cvora
                previous = tmp             // u levom podstablu
                tmp = tmp->right;
            }
            node->setKey(tmp->getKey());          // prepisivanje reference na kljuc
            if (previous == node)                 // tmp je direktni levi potomak node-a
                  previous->left  = tmp->left;    // ostaje isti raspored u levom podstablu
            else
                  previous->right = tmp->left;    // levi potomak tmp-a
            delete tmp;                           // se pomera na mesto tmp-a
            numOfElements--;
            return;
        }
...
```



Brisanja mešanjem

Od dva podstabla čvora koji se briše kreira se jedno stablo **mešanjem**, a zatim dodaje kao podstablo roditelju uklonjenog čvora. Mešanje se može ostvariti na jedan od dva načina:

• pronađe se krajnji desni čvor u levom podstablu i njemu se kao desno podstablo dodeli čitavo desno podstablo čvora koji se briše, a na mesto obrisanog čvora dolazi direktni levi potomak obrisanog čvora i

• pronađe se krajnji levi čvor u desnom podstablu i njemu se kao levo podstablo dodeli čitavo levo podstablo čvora koji se briše, a na mesto obrisanog čvora dolazi direktni desni potomak obrisanog čvora.

Za razliku od brisanja kopiranjem, brisanje mešanjem može narušiti balansiranost stabla, obzirom da se menja visina jednog podstabla.

Brisanje mešanjem

```
template <class T>
void BSTree<T>::deleteByMerging(T el)
{
    BSTNode<T> *tmp, *node, *p = root, *prev = NULL;
    while (p != NULL && !p->isEQ(el)) { // nalazenje cvora sa zeljenim el.
        prev = p;
        if (p->isLT(el)) p = p->right;
        else p = p->left;
    }
    node = p;
```

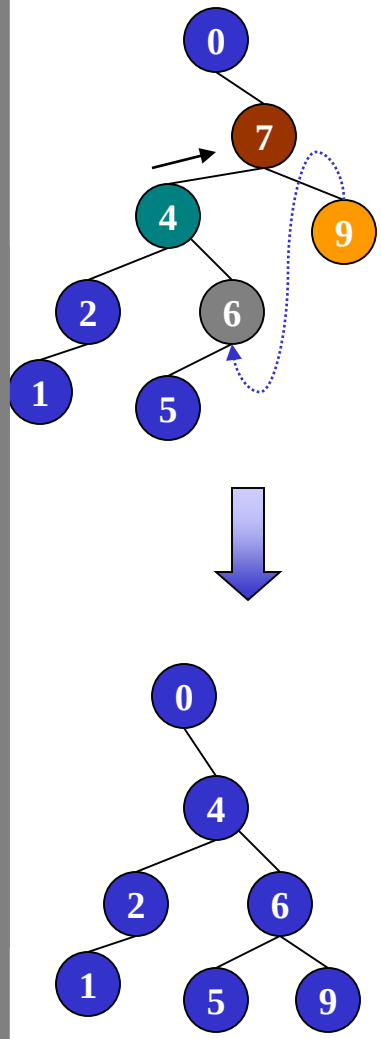
```
tmp = node->left;

// nalazenje krajnjeg desnog cvora u levom podstablu
while (tmp->right != NULL)
    tmp = tmp->right;

// prebacivanje desnog linka node-a u tmp
tmp->right = node->right;

// na tekucem mestu bice prvi levi potomak
node = node->left;
```

```
        numOfElements--;
    }
    else if (root != NULL) throw new SBPException("Element is not in the tree!");
    else throw new SBPException("The tree is empty!");
}
```



Brisanje mešanjem – 1.deo

```
template <class T>
void BSTree<T>::deleteByMerging(T el)
{
   BSTNode<T> *tmp, *node, *p = root, *prev = NULL;
   while (p != NULL && !p->isEQ(el)) { // nalazenje cvora sa zeljenim el.
        prev = p;
        if (p->isLT(el))
            p = p->right;
        else
            p = p->left;
   }
   node = p;
  …
```

Brisanje mešanjem – 2.deo

```
...
if (p != NULL && p->isEQ(el)) {
        if (node->right == NULL)                    // cvor nema desnog potomka (1)
            node = node->left;
        else if (node->left == NULL)                // cvor nema levog potomka (2)
            node = node->right;
        else {                                      // cvor ima oba potomka (3)
            tmp = node->left;
            while (tmp->right != NULL) // nalazenje krajnjeg desnog cvora
                tmp = tmp->right;       //  u levom podstablu
            tmp->right = node->right; // prebacivanje desnog linka node-a u tmp
            node = node->left;          // na tekucem mestu bice prvi levi potomak
        }
  ...
```

Brisanje mešanjem – 3.deo

```
...
        if (p == root)
            root = node;
        else if (prev->left == p)
            prev->left = node;
        else
            prev->right = node;
        delete p;
        numOfElements--;
    }
    else if (root != NULL)
            throw new SBPException("Element is not in the tree!");
    else
            throw new SBPException("The tree is empty!");
}
```

Balansiranje

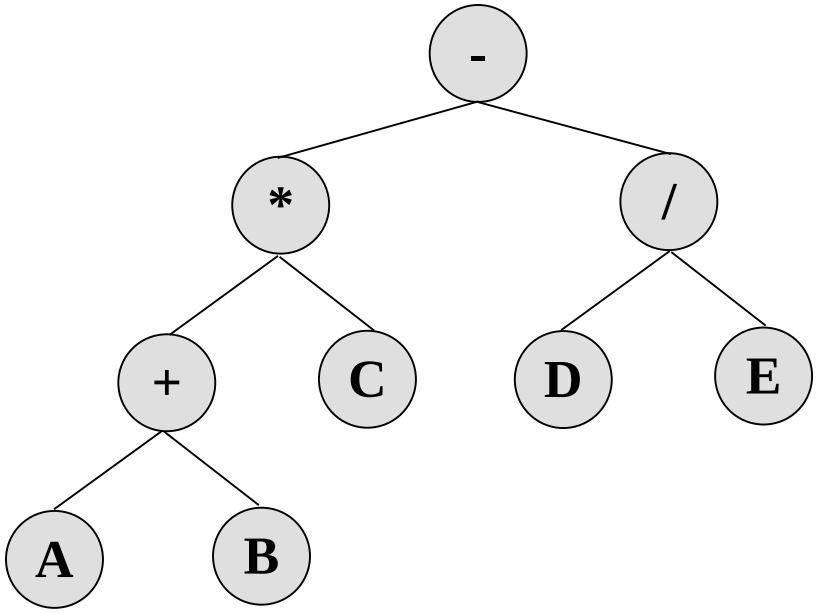
1. sve čvorove stabla iskopirati u polje i obrisati stablo,
2. urediti polje u rastući redosled,
3. formirati novo stablo čiji koren postaje središnji elemenat uređenog polja,
4. levo podstablo kreira se od leve polovine uređenog polja, a
5. desno podstablo kreira se od desne polovine uređenog polja.

Koraci 3, 4 i 5 ponavljaju se rekurzivno sve dok postoji bar jedan elemenat u segmentu uređenog polja na osnovu koga se formira podstablo.

Balansiranje

```
template <class T>
void BSTree<T>::balance(int data[], int first, int last)
{
    if (first <= last) {
        int middle = (first + last)/2;
        insert(data[middle]);
        balance(data,first,middle-1);
        balance(data,middle+1,last);
    }
}
```

Stablo algebarskog izraza

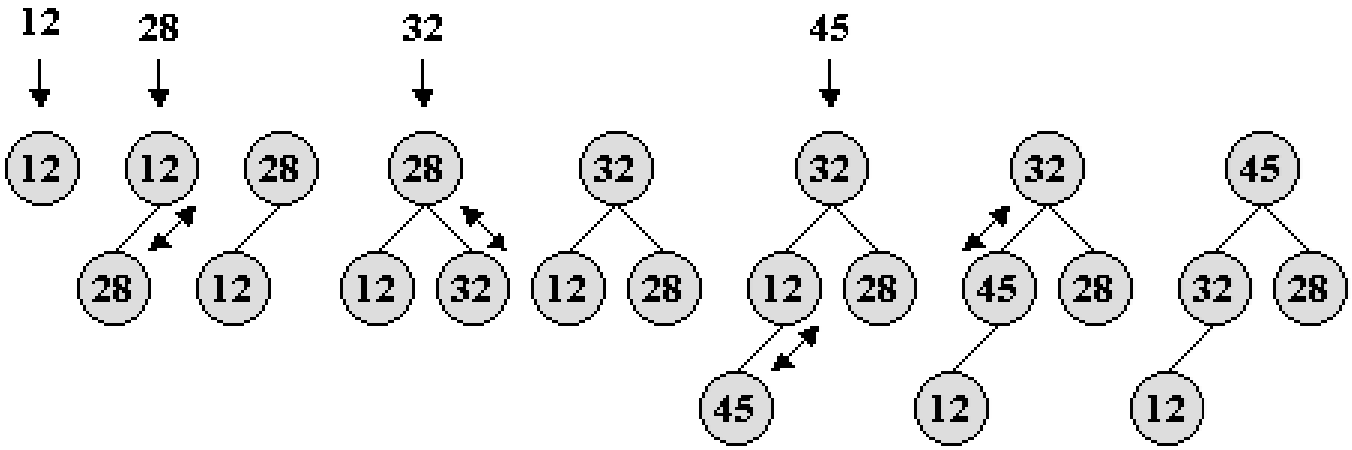


Infiks notacija: (A + B) * C - D / E

Gomila

Gotovo potpuno stablo čiji svaki čvor sadrži ključ sa vrednošću većom/manjom ili jednakom ključu njegovog roditelja, naziva se **gomila** (engl. *heap*). Ukoliko je vrednost roditeljskog ključa veća (ili jednaka) od vrednosti ključeva čvorova njegove dece, radi se o **max gomili** (engl. *maxheap*), a ako je manja ili jednaka, o **min gomili** (engl. *minheap*). Pod pojmom **gomila** najčešće se podrazumeva **binarna gomila**.

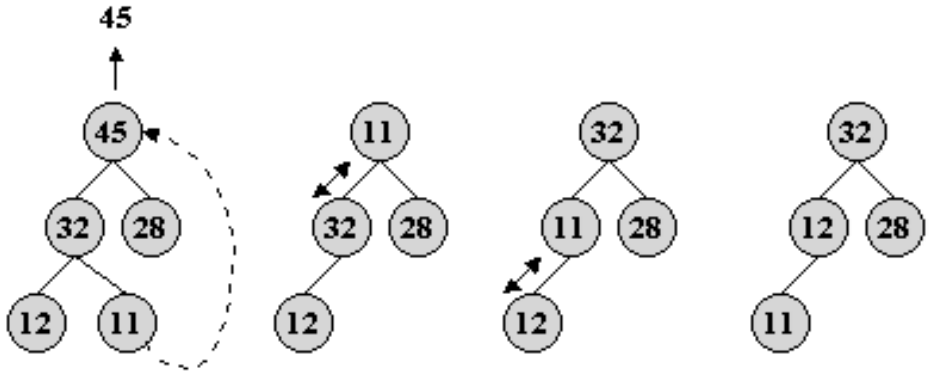
Dodavanje elementa u max gomilu



Novi čvor dodaje se uvek kao list, a zatim se vrednost njegovog ključa upoređuje sa vrednošću ključa roditelja

Brisanje korena gomile

- čita se ključ korena gomile i koren se briše,
- na mesto korena upisuje se poslednji element gomile i
- čvor koji je trenutno na mestu korena pomera se ka listovima sve dok ne zauzme svoje mesto (odnosno, dok stablo ponovo ne postane gomila, obzirom da se kopiranjem poslednjeg elementa na mesto korena narušava struktura gomile).



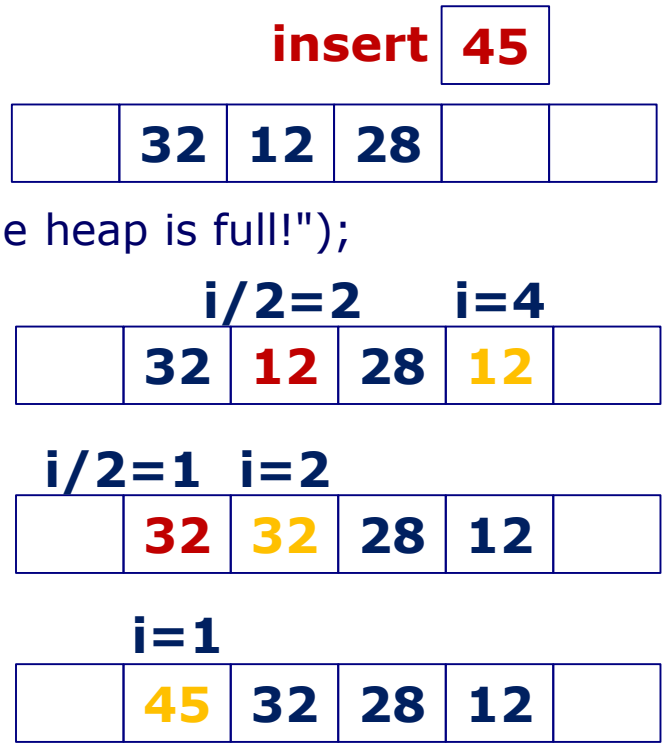
Binarni MaxHeap

```
template <class T>
class BinaryMaxHeap
{
protected:
        T* array;
        long length;
        long numOfElements;

public:
        BinaryMaxHeap (long len){
                length = len;
                numOfElements = 0;
                array = new T[length + 1];
        }
        ~BinaryMaxHeap (){ delete [] array;};
        bool isEmpty(){ return numOfElements == 0; };
        void insert (T el);
        T deleteRoot();
        static void HeapSort(T* a, long n);
};
```

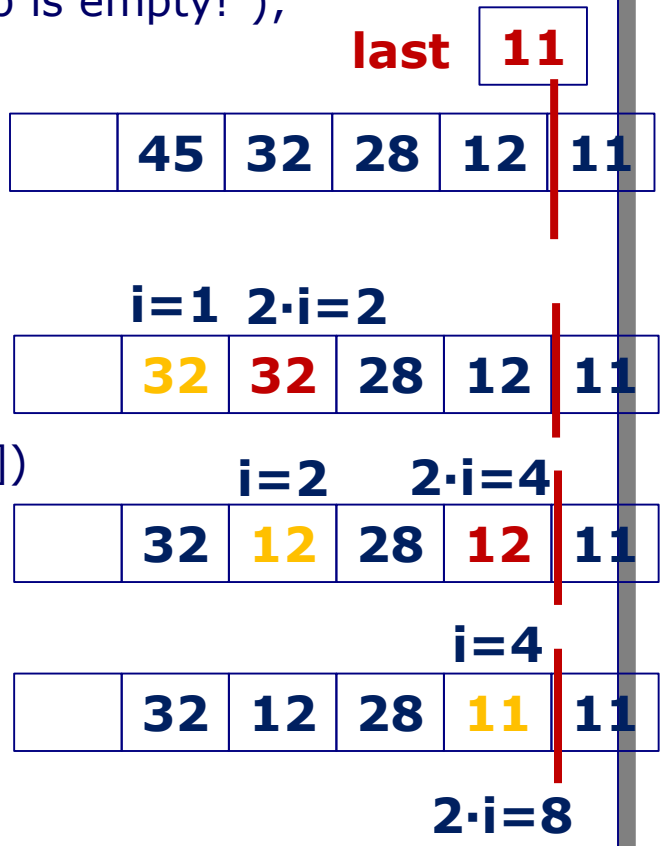
Umetanje (dodavanje) elementa

```
template <class T>
void BinaryMaxHeap<T>::insert (T el)
{
        if (numOfElements == length)
                throw new SBPException("The heap is full!");
        numOfElements++;
        int i = numOfElements;
        while (i > 1 && array [i/2] < el)
        {
                array [i] = array [i / 2];
                i /= 2;
        }
        array [i] = el;
}
```



Brisanje korena

```
template <class T>
T BinaryMaxHeap<T>::deleteRoot ()
{
        if (numOfElements == 0)
                throw new SBPException("The heap is empty!");
        T result = array [1];
        T last = array [numOfElements];
        numOfElements--;
        long i = 1;
        while (2 * i < numOfElements + 1)
        {
                long child = 2 * i;
                if (child + 1 < numOfElements + 1
                && array [child + 1] > array [child])
                        child += 1;
                if (last >= array [child]) break;
                array [i] = array [child];
                i = child;
        }
        array [i] = last;
        return result;
}
```



HeapSort

```
template <class T>
void BinaryMaxHeap<T>::HeapSort(T* a, long n)
{
        BinaryMaxHeap<T> heap(n+1);

        // Umetanje elemenata polja u heap
        for(long i=0; i<n; i++)
            heap.insert(a[i]);

        // Brisanje korena i smestanje u polje
        i = n;
        while(!heap.isEmpty())
            a[--i]=heap.deleteRoot();
}
```