Двоично дърво

Изготвил:
гл.ас. д-р Нора Ангелова

Двоично дърво

Двоично дърво
(свързано представяне)

```
template <class T>
struct node
{
    T inf;
    node<T> *left, *right;
};
```

Двоично дърво
(свързано представяне)

```
template <class T>
class BinTree
{
private:
  node<T> *root;
  void DeleteBinTree(node<T>* &) const;
  void Copy(node<T> * &, node<T>* const&) const;
  void CopyBinTree(BinTree<T> const&);
  void pr(const node<T> *) const;
  void CreateBinTree(node<T> * &) const;
```

Двоично дърво
(свързано представяне)

```
public:
  BinTree();
  ~BinTree();
  BinTree(BinTree<T> const&);
  BinTree& operator=(BinTree<T> const&);
  T RootBinTree() const;
  BinTree<T> LeftBinTree() const;
  BinTree<T> RightBinTree() const;
  node<T>* GetRoot();
  bool empty() const;
  void print() const;
  void Create();
  void Create3(T, BinTree<T>, BinTree<T>);
};
```

Двоично дърво
(свързано представяне)

```
template <class T>
BinTree<T>::BinTree()
{
  root = NULL;
}

template <class T>
BinTree<T>::~BinTree()
{
  DeleteBinTree(root);
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
BinTree<T>::BinTree(BinTree<T> const& r)
{
  CopyBinTree(r);
}

template <class T>
BinTree<T>& BinTree<T>::operator=(BinTree<T> const& r)
{
  if (this != &r)
  {
    DeleteBinTree(root);
    CopyBinTree(r);
  }
  return *this;
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
void BinTree<T>::DeleteBinTree(node<T>* &p) const
{
  if (p)
  {
     DeleteBinTree(p->left);
     DeleteBinTree(p->right);
     delete p;
     p = NULL;
  }
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
void BinTree<T>::CopyBinTree(BinTree<T> const& r)
{
  Copy(root, r.root);
}
template <class T>
void BinTree<T>::Copy(node<T> * & pos, node<T>* const & r) const
{
  pos = NULL;
  if (r)
  {
    pos = new node<T>;
    pos->inf = r->inf;
    Copy(pos->left, r->left);
    Copy(pos->right, r->right);
  }
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
bool BinTree<T>::empty() const
{
  return root == NULL;
}

template <class T>
T BinTree<T>::RootBinTree() const
{
  return root->inf;
}

template <class T>
node<T>* BinTree<T>::GetRoot()
{
  return root;
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
BinTree<T>& BinTree<T>::LeftBinTree() const
{
  BinTree<T> t;
  Copy(t.root, root->left);
  return t;
}

template <class T>
BinTree<T>& BinTree<T>::RightBinTree() const
{
  BinTree<T> t;
  Copy(t.root, root->right);
  return t;
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
void BinTree<T>::pr(const node<T>*p) const
{
  if (p)
  {
    pr(p->left);
    cout << p->inf << " ";
    pr(p->right);
  }
}

template <class T>
void BinTree<T>::print() const
{
  pr(root);
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
void BinTree<T>::Create3(T x, BinTree<T> l, BinTree<T> r)
{
    root = new node<T>;
    root->inf = x;
    Copy(root->left, l.root);
    Copy(root->right, r.root);
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
void BinTree<T>::CreateBinTree(node<T>* & pos) const
{
   T x; char c;
   cout << "root: ";
   cin >> x;

   pos = new node<T>;
   pos->inf = x;
   pos->left = NULL;
   pos->right = NULL;

   cout << "left BinTree of: " << x << " y/n? ";
   cin >> c;
   if (c == 'y') CreateBinTree(pos->left);

   cout << "right BinTree of: " << x << " y/n? ";
   cin >> c;
   if (c == 'y') CreateBinTree(pos->right);
}
```

Двоично дърво
(свързано представяне)

```
template <class T>
void BinTree<T>::Create()
{
  CreateBinTree(root);
}

int main()
{
  BinTree<int> binTree;
  binTree.Create();
  binTree.print();

  system("pause");
  return 0;
}
```

Двоично дърво
(свързано представяне)

Да се напише функция, която създава ново дърво като увеличава всеки от върховете на двоично дърво от цели числа с дадено цяло число.

```
typedef BinTree<int> IntBinTree;
IntBinTree addTreeElem(int a, IntBinTree const& t)
{
  IntBinTree nt;

  if (!t.empty())
    nt.Create3(
      t.RootBinTree() + a,
      addTreeElem(a, t.LeftBinTree()),
      addTreeElem(a, t.RightBinTree())
    );

  return nt;
}
```

Двоично дърво
(свързано представяне)

Да се напише функция, която създава ново дърво като увеличава всеки от върховете на двоично дърво от цели числа с дадено цяло число.

```
typedef BinTree<int> IntBinTree;
template <class T>
IntBinTree addTreeElem(int a, node<T>* root)
{
  IntBinTree t;

  if (root)
    t.Create3(
      root->inf + a,
      addTreeElem(a, root->left),
      addTreeElem(a, root->right)
    );

  return t;
}
```

Двоично наредено дърво

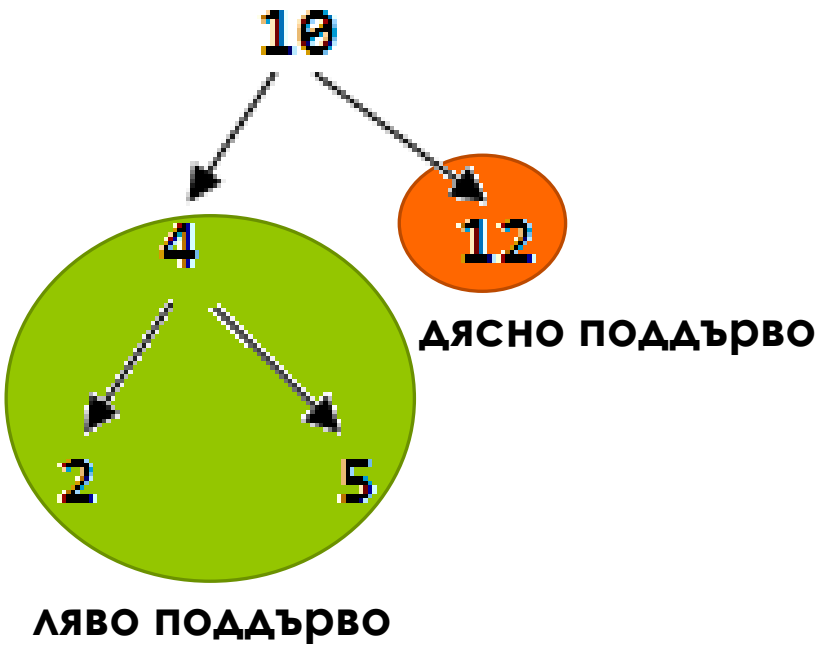
Двоично наредено дърво

Двоично наредено дърво от тип T е рекурсивна структура от данни и се дефинира по следния начин:

- Празното двоично дърво е двоично наредено дърво;
- Непразно двоично дърво, върховете на лявото поддърво на което са по-малки от корена, върховете на дясното поддърво са по-големи от корена и лявото, и дясното поддърво са двоично наредени дървета от тип T.

Двоично наредено дърво

Примери:



Двоично наредено дърво

Включване на елемент

Нека `tree` е двоично наредено дърво от тип T. Включването на елемента `a` от тип T в `tree` се осъществява по следния начин:

- Ако `tree` е празното двоично дърво, новото двоично наредено дърво е с корен елемента `a` и празни ляво и дясно поддървета.
- Ако `tree` не е празно и `a` е по-малко от корена му, елементът `a` се включва в лявото поддърво на `tree`.
- Ако `tree` не е празно и `a` е не по-малко от корена му, елементът `a` се включва в дясното поддърво на `tree`.

Двоично наредено дърво

Свойства

- Смесеното обхождане (ЛКД) сортира върховете във възходящ ред.

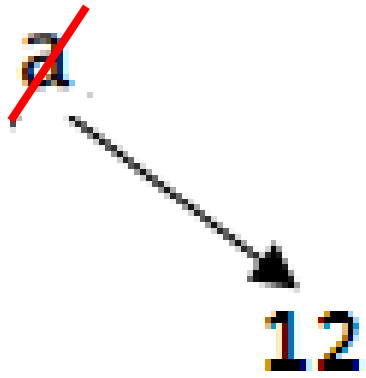
- Обхождането по метода (ДКЛ) сортира върховете в низходящ ред.

Двоично наредено дърво

Изтриване на елемент

Нека `tree` е двоично наредено дърво от тип T. Изтриване на елемента `a` от `tree` се осъществява по следния начин:

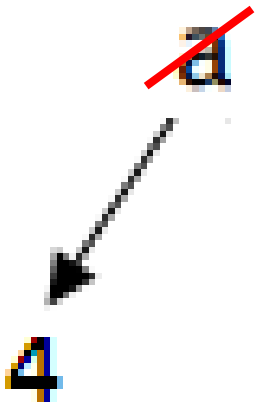
- Ако `a` е корен на `tree` с празно ЛПД, то новото двоично наредено дърво е ДПД на `tree`.



Двоично наредено дърво

Изтриване на елемент

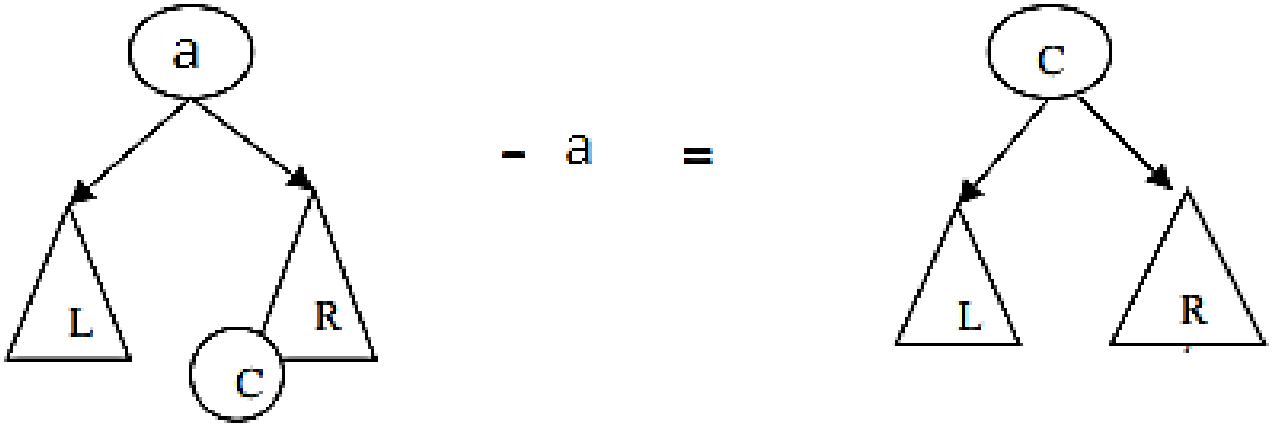
- Ако a е корен на `tree` с празно ДПД, то новото двоично наредено дърво е ЛПД на `tree`.



Двоично наредено дърво

Изтриване на елемент

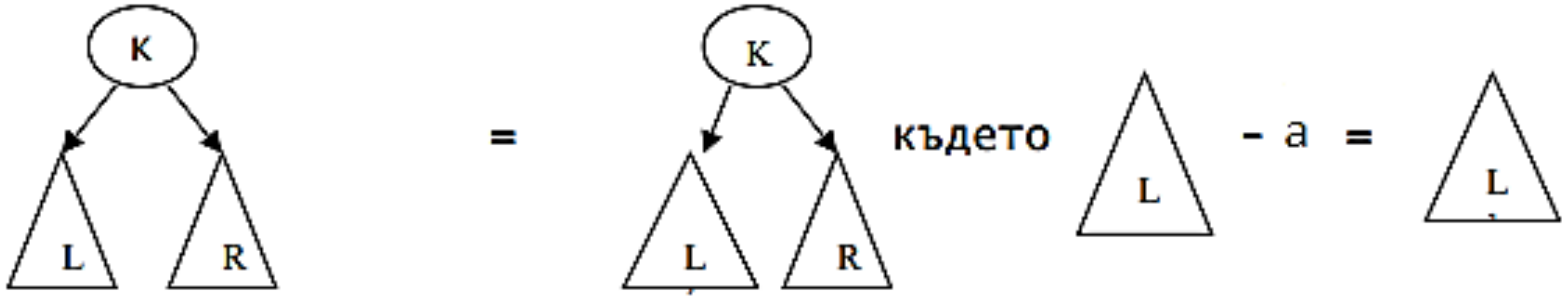
- Нека a е корен на `tree` с непразни ляво и дясно поддървета и c е най-лявото листо от ДПД на `tree`. Новото двоично наредено дърво има корен елемента c, ЛПД е ЛПД на `tree` и ДПД е двоичното наредено дърво, получено от ДПД на `tree` след изключване на елемента c.



Двоично наредено дърво

Изтриване на елемент

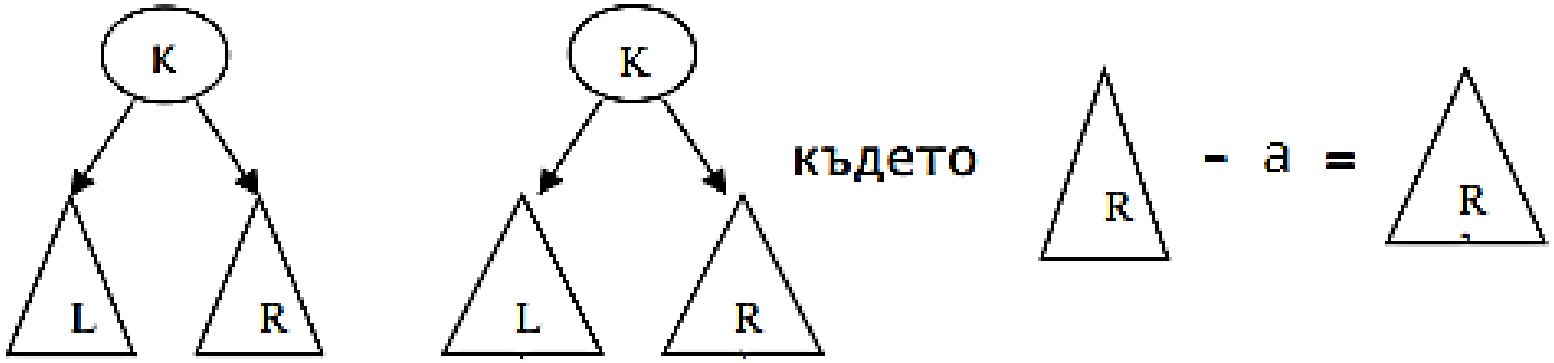
- Ако ***k*** е корен на `tree` с непразни ляво и дясно поддървета и стойността на `a` е по-малка от стойността на ***k***, то новото двоично наредено дърво има корен ***k***, ЛПД е ЛПД на `tree`, от което е изключен елемента `a`, и ДПД е ДПД на `tree`;



Двоично наредено дърво

Изтриване на елемент

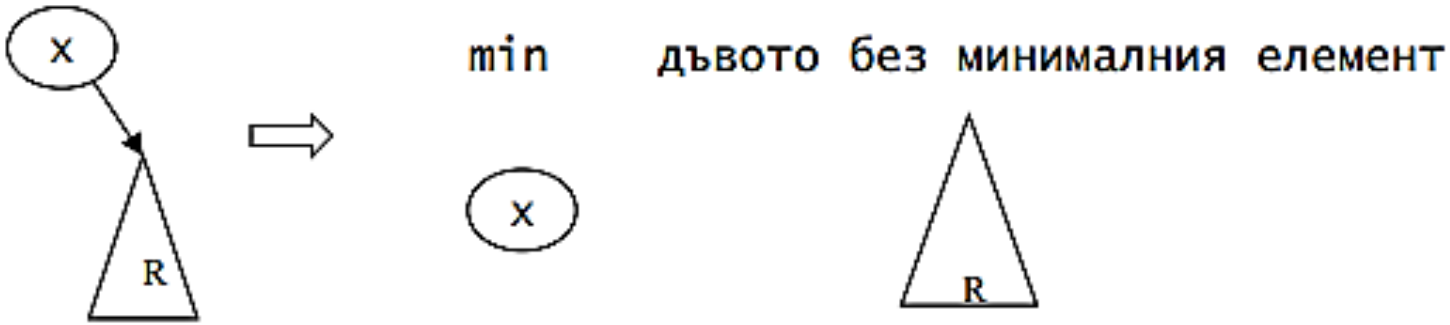
- Ако ***k*** е корен на `tree` с непразни ляво и дясно поддървета и стойността на `a` е по-голяма от стойността на ***k***, то новото двоично наредено дърво има корен ***k***, ЛПД е ЛПД на `tree` и ДПД е ДПД на `tree`, от което е изключен елемента `a`.



Двоично наредено дърво

Намиране на минимален елемент и дървото без миналния му елемент

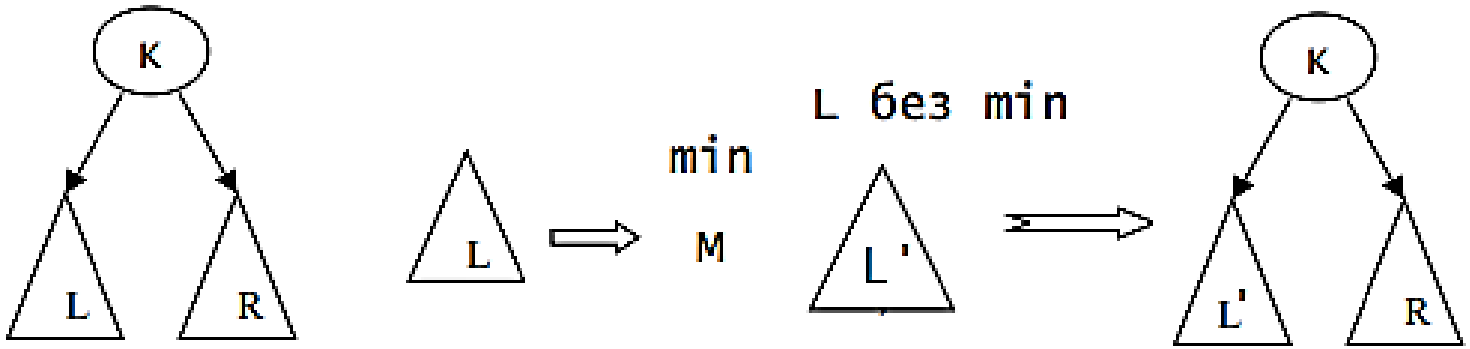
- Ако лявото поддърво на подразбиращото се двоично наредено дърво е празно, минималният му елемент е корена, а дървото без минималния елемент е дясното му поддърво



Двоично наредено дърво

Намиране на минимален елемент и дървото без миналния му елемент

- Ако лявото му поддърво е непразно, на това двоично наредено дърво се намира минималния елемент и дървото без минималния елемент.
  След това се конструира двоично наредено дърво от корена, лявото поддърво без минималния му елемент и дясното поддърво на подразбиращото се дърво.



Двоично наредено дърво
(свързано представяне)

```
template <class T>
struct node
{
  T inf;
  node<T> *left;
  node<T> *right;
};
```

Двоично наредено дърво

(свързано представяне)

```
template <class T>
class BinOrdTree
{
private:
  node<T> *root;
  void DeleteTree(node<T>* &) const;
  void Copy(node<T>* &, node<T>* const&) const;
  void CopyTree(BinOrdTree<T> const&);
  void pr(const node<T> *) const;
  void Add(node<T> *&, T const &) const;
```

Двоично наредено дърво

(свързано представяне)

```
public:
  BinOrdTree();
  ~BinOrdTree();
  BinOrdTree(BinOrdTree<T> const&);
  BinOrdTree& operator=(BinOrdTree<T> const&);
  T RootTree() const;
  node<T>* GetRoot() const;
  BinOrdTree<T> LeftTree() const;
  BinOrdTree<T> RightTree() const;

  bool empty()const;
  void print() const;
  void AddNode(T const & x);
  void DeleteNode(T const&);
  void Create3(T, BinOrdTree<T>, BinOrdTree<T>);
  void Create();
  void MinTree(T &, BinOrdTree<T> &) const;
};
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
BinOrdTree<T>::BinOrdTree()
{
  root = NULL;
}

template <class T>
BinOrdTree<T>::~BinOrdTree()
{
  DeleteTree(root);
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
BinOrdTree<T>::BinOrdTree(BinOrdTree<T> const& r)
{
  CopyTree(r);
}

template <class T>
BinOrdTree<T>& BinOrdTree<T>::operator=(BinOrdTree<T> const& r)
{
  if (this != &r)
  {
    DeleteTree(root);
    CopyTree(r);
  }

  return *this;
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::DeleteTree(node<T>* &p) const
{
  if (p)
  {
    DeleteTree(p->left);
    DeleteTree(p->right);
    delete p;
    p = NULL;
  }
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::CopyTree(BinOrdTree<T> const& r)
{
  Copy(root, r.root);
}
template <class T>
void BinOrdTree<T>::Copy(node<T>* &pos, node<T>* const &r) const
{
  pos = NULL;
  if (r)
  {
    pos = new node<T>;
    pos->inf = r->inf;
    Copy(pos->left, r->left);
    Copy(pos->right, r->right);
  }
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
bool BinOrdTree<T>::empty()const
{
  return root == NULL;
}

template <class T>
T BinOrdTree<T>::RootTree()const
{
  return root->inf;
}

template <class T>
node<T>* BinOrdTree<T>::GetRoot() const
{
  return root;
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
BinOrdTree<T> BinOrdTree<T>::LeftTree() const
{
  BinOrdTree<T> t;
  Copy(t.root, root->left);
  return t;
}

template <class T>
BinOrdTree<T> BinOrdTree<T>::RightTree() const
{
  BinOrdTree<T> t;
  Copy(t.root, root->right);
  return t;
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::pr(const node<T>*p) const
{
  if (p)
  {
    pr(p->left);
    cout << p->inf << " ";
    pr(p->right);
  }
}

template <class T>
void BinOrdTree<T>::print() const
{
  pr(root);
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::Add(node<T>* &p, T const & x)const
{
  if (!p)
  {
    p = new node<T>;
    p->inf = x;
    p->left = NULL;
    p->right = NULL;
  }
  else
  if (x < p->inf)
    Add(p->left, x);
  else
    Add(p->right, x);
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::AddNode(T const & x)
{
  Add(root, x);
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::Create()
{
  root = NULL;
  T x; char c;
  do
  {
    cout << "> ";
    cin >> x;
    AddNode(x);
    cout << "next elem y/n? "; cin >> c;
  } while (c == 'y');
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::Create3(T x, BinOrdTree<T> l, BinOrdTree<T> r)
{
  root = new node<T>;
  root->inf = x;
  Copy(root->left, l.root);
  Copy(root->right, r.root);
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::MinTree(T &x, BinOrdTree<T> &mint) const
{
  T a = RootTree();
  if (!root->left)
  {
    x = a;
    mint = RightTree();
  }
  else
  {
    BinOrdTree<T> t1;
    LeftTree().MinTree(x, t1);
    mint.Create3(a, t1, RightTree());
  }
}
```

Двоично наредено дърво
(свързано представяне)

```
template <class T>
void BinOrdTree<T>::DeleteNode(T const& x)
{
  if (root)
  {
    T a = RootTree();
    BinOrdTree<T> t;
    if (a == x && !root->left) *this = RightTree();
    else
    if (a == x && !root->right) *this = LeftTree();
    else
    if (a == x)
    {
      T c;
      RightTree().MinTree(c, t);
      Create3(c, LeftTree(), t);
    }
```

Двоично наредено дърво
(свързано представяне)

```
else
    if (x < a)
    {
      t = *this;
      *this = LeftTree();
      DeleteNode(x);
      Create3(a, *this, t.RightTree());
    }
    else
    if (x > a)
    {
      t = *this;
      *this = RightTree();
      DeleteNode(x);
      Create3(a, t.LeftTree(), *this);
    }
  }
}
```

Двоично дърво
(свързано представяне)

Да се напише шаблон на булева функция, чрез който се проверява дали елемент се съдържа в двоично наредено дърво.

```
template <class T>
bool member(T a, BinOrdTree<T> const& t)
{
  if (t.empty()) return false;
  if (a == t.RootTree()) return true;
  if (a < t.RootTree()) return member(a, t.LeftTree());
  else return member(a, t.RightTree());
}
```

Двоично дърво
(свързано представяне)

Да се напише шаблон на булева функция, чрез който се проверява дали елемент се съдържа в двоично наредено дърво.

```
template <class T>
bool member(T a, node<T>* root)
{
  if (!root) return false;
  if (root->inf == a) return true;
  if (a < root->inf) return member(a, root->left);
  else return member(a, root->right);
}

template <class T>
bool member(T a, BinOrdTree<T> const& t)
{
  return member(a, t.GetRoot());
}
```

Двоично дърво
(свързано представяне)

Да се напише шаблон на функция Del, чрез който се изтрива елемент от двоично наредено дърво. Функцията да не използва други методи на класа.

```
template <class T>
void BinOrdTree<T>::Del(node<T>* &t, const T& x)
{
  if (!t) return;
  if (x < t -> inf) Del(t -> left, x);
  else if (x > t -> inf) Del(t -> right, x);
  else
  {
    node<T> *p;
    if (!(t -> left)) { p = t; t = t -> right; delete p; }
    else if (!(t -> right)) { p = t; t = t -> left; delete p; }
    else
    {
      p = t -> right;
      while (p -> left) p = p -> left;
      t -> inf = p -> inf;
      Del(t -> right, p -> inf);
    }
  }
}
```

```
cout << “КРАЙ”;
```