

Раздел «Алгоритмы» . BinarySearchTreeCPP :

## Реализация двоичного дерева поиска на C++

- Двоичное дерево поиска — описание, теория и задачи

Здесь приведены два класса, реализующие двоичное дерево поиска: `TreeNode` и `Tree`. Первый класс содержит описание узлов дерева, а второй реализует основные операции с ними.

Класс `TreeNode`:

```
#ifndef _NODE_H
#define _NODE_H

template <class NODETYPE> class TreeNode
{
    friend class Tree<NODETYPE>;
public:
    TreeNode(const NODETYPE &);
    NODETYPE get_data();
protected:
    TreeNode* left;          /* указатель на левого ребенка */
    TreeNode* right;         /* указатель на правого ребенка */
    TreeNode* parent;        /* указатель на родителя */
    NODETYPE data;           /* ключ */
};

template<class NODETYPE>
TreeNode <NODETYPE>::TreeNode(const NODETYPE &a)
{
    data=a;
    left=right=0;
}

template <class NODETYPE>
NODETYPE TreeNode <NODETYPE>::get_data()
{
    return data;
}
#endif
```

Класс `Tree`:

```
#ifndef _TREE_H
#define _TREE_H

template <class NODETYPE> class Tree;
#include "node.h"

template <class NODETYPE>
class Tree
{
public:
    Tree(); /* конструктор */
    int insert_node(const NODETYPE &); /* вставляет узел */
    void delete_node(TreeNode<NODETYPE> *); /* удаляет узел */
    void inorder_walk(TreeNode<NODETYPE> *); /* печатает все ключи в неубывающем порядке */
    TreeNode<NODETYPE> * find_max(TreeNode<NODETYPE> *); /* находит узел с минимальным значением ключа и возвращает указатель */
    TreeNode<NODETYPE> * find_min(TreeNode<NODETYPE> *);
    TreeNode<NODETYPE> * find_node(TreeNode<NODETYPE> *, const NODETYPE &);
    TreeNode<NODETYPE> * find_successor(const NODETYPE &); /* находит элемент с ключом, следующим за данным числом */
    TreeNode<NODETYPE> * get_root(); /* возвращает указатель на корень дерева */
private:
    TreeNode<NODETYPE> *root; /* собственно, сам корень */
};

template<class NODETYPE>
Tree<NODETYPE>::Tree()
{
    root=0; /* в начале дерево пусто */
}

template<class NODETYPE>
int Tree<NODETYPE>::insert_node(const NODETYPE &x)
{
    TreeNode<NODETYPE> * n=new TreeNode<NODETYPE>(x); /* создаем новый узел, его мы будем вставлять */
    TreeNode<NODETYPE> * ptr;
    TreeNode<NODETYPE> * ptr1;

    n->parent=n->left=n->right=0; /* он - лист */
    ptr=root;
    while(ptr!=0) /* идем от корня и ищем подходящее место для нашего нового элемента, оно должно быть слева */
    {
        ptr1=ptr; /* будущий родитель нового узла */
        if(x < ptr->get_data() ) /* по определению нашего дерева - слева значение ключа меньше родителя, */
            ptr=ptr->left;
        else
            ptr=ptr->right; /* справа - больше */
    }
    n->parent=ptr1;
    if(ptr1==0) /* дерево было пусто? */
        root=n;
    else
    {
        if(x < ptr1->get_data() )
            ptr1->left=n;
        else
            ptr1->right=n;
    }
}
```

Поиск

Поиск

Раздел «Алгоритмы»

[Главная](#)[Форум](#)[Ссылки](#)[EI Judge](#)

Инструменты:

[Поиск](#)[Изменения](#)[Index](#)[Статистика](#)

Разделы

[Информация](#)[Алгоритмы](#)[Язык Си](#)[Язык Ruby](#)[Язык Ассемблера](#)[EI Judge](#)[Парадигмы](#)[Образование](#)[Сети](#)[Objective C](#)

Logon&gt;&gt;

```

        ptr1->right=n;
    }
    return 0;
}

/* возможны три случая - если у z нет детей, то помещаем 0 в соответствующее поле
 * родителя z, если у z есть один ребенок, то можно вырезать z, соединив его родителя напрямую с
 * его ребенком. Если же детей двое, то требуются некоторые приготовления: мы находим следующий
 * (в смысле порядка на ключах) за z элемент y; у него нет левого ребенка (всегда). Теперь можно
 * скопировать ключ и дополнительные данные из вершины y в вершину z, а саму вершину y удалить
 * описанным выше способом */
template<class NODETYPE>
TreeNode<NODETYPE>* Tree<NODETYPE>::delete_node(TreeNode<NODETYPE>* z)
{
    TreeNode<NODETYPE>* y;
    TreeNode<NODETYPE>* x;
    if(z->left == 0 || z->right == 0) /* в этой и следующих двух строках ищем вершину y, которую мы потом вырежем */
        y=z;
    else
        y=find_successor(z->get_data());
    if(y->left!=0) /* x - указатель на существующего ребенка y или 0 если таковых нет */
        x=y->left;
    else
        x=y->right;
    if(x!=0) /* эта и следующие 9 строк - вырезание y */
        x->parent=y->parent;
    if(y->parent == 0)
        root=x;
    else
    {
        if (y== (y->parent)->left)
            (y->parent)->left=x;
        else
            (y->parent)->right=x;
    }
    if(y!=z) /* если мы вырезали вершин, отличную от z, то ее данные перемещаем в z */
        z->data=y->get_data();
    return y;
}

template<class NODETYPE>
TreeNode<NODETYPE>* Tree<NODETYPE>::find_max(TreeNode<NODETYPE>* x)
{
    while(x->right!=0) /* здесь все очевидно - самое максимальное значение у самого правого */
        x=x->right;
    return x;
}

template<class NODETYPE>
TreeNode<NODETYPE>* Tree<NODETYPE>::find_min(TreeNode<NODETYPE>* x)
{
    while(x->left!=0)
        x=x->left;
    return x;
}

template<class NODETYPE>
TreeNode<NODETYPE>* Tree<NODETYPE>::find_successor(const NODETYPE & val)
{
    TreeNode<NODETYPE>* x=find_node(root,val); /* получим указатель на ноду с ключом val */
    TreeNode<NODETYPE>* y;
    if(x == 0)
        return 0;
    if(x->right!=0) /* если у нее есть правые дети, то следующий элемент - миним */
        return find_min(x->right);
    y=x->parent;
    while(y!=0 && x == y->right) /* иначе - идем вверх и ищем первый элемент, являющийся левым
 * потомком своего родителя */
    {
        x=y;
        y=y->parent;
    }
    return y;
}

template<class NODETYPE>
TreeNode<NODETYPE>* Tree<NODETYPE>::find_node(TreeNode<NODETYPE>* n,
    const NODETYPE & val)
{
    if(n==0 || val==n->get_data())
        return n;
    if(val > n->get_data() )
        return find_node(n->right,val);
    else
        return find_node(n->left,val);
}

template<class NODETYPE>
void Tree<NODETYPE>::inorder_walk(TreeNode<NODETYPE>* n)
{
    if(n!=0)
    {
        inorder_walk(n->left);
        cout<<n->get_data()<<endl;
        inorder_walk(n->right);
    }
}

template<class NODETYPE>
TreeNode<NODETYPE>* Tree<NODETYPE>::get_root()
{
    return root;
}
#endif





```

А теперь небольшой примерчик того, как это все работает:

```
#include <iostream>
#include <iomanip>
#include "tree.h"
using namespace std;
int main()
{
    Tree<int> intTree;          /* создаем новой бинарное дерево с ключем типа int */
    int a;
    cout<<"10 numbers:"<<endl; /* заполняем его */
    for(int i=0;i<10;i++)
    {
        cin>>a;
        intTree.insert_node(a);
    }
    cout<<endl<<"inorder walk:"<<endl; /* обходим */
    intTree.inorder_walk(intTree.get_root()); /* вот для этого понадобился метод get_root() ;-) */
    cout<<endl<<"Minimum is: "<<(intTree.find_min(intTree.get_root())->get_data())<<endl;
    cout<<endl<<"Maximum is: "<<(intTree.find_max(intTree.get_root())->get_data())<<endl;
    cout<<"Enter node value U want to delete:"; /* попробуем удалить узел с ключем а */
    cin>>a;
    intTree.delete_node(intTree.find_node(intTree.get_root(),a)); /* если их несколько, то удалится первый найденный */
    cout<<endl<<"Now inorder walk:"<<endl;
    intTree.inorder_walk(intTree.get_root()); /* посмотрим на результат */
}
```

-- AntonPolyakov - 06 Apr 2004

- tree.h: класс Tree
- node.h: класс TreeNode

Attachment	Action	Size	Date	Who	Comment
 tree.h	manage	2.7 K	06 Apr 2004 - 20:48	AntonPolyakov	класс Tree
 node.h	manage	0.4 K	06 Apr 2004 - 20:48	AntonPolyakov	класс TreeNode
 main.cpp	manage	0.7 K	06 Apr 2004 - 20:49	AntonPolyakov	тест для этих классов, очень примитивный
 Makefile	manage	0.2 K	06 Apr 2004 - 20:51	AntonPolyakov	для GNU/Linux, Unix

Copyright © 2003-2022 by the contributing authors.