《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 计算机类 | 姓名 |  | 学号 |  |
| 实验布置日期 | | 2019.11.14 | | 提交  日期 | 2019.12.27 | | 成绩 |  |

课程实践实验7：二叉树的实现

## 问题描述及要求

产生一个菜单驱动的演示程序，用以说明二叉树的使用。元素由单个键组成，键为单个字符。用户能演示的二叉树基本操作至少包括：构造二叉树，按先序、中序、后序、层序遍历这棵二叉树，求二叉树的深度、宽度，统计度为0，1，2的结点数等。

二叉树采用链式存储结构。

对二叉查找树做上述工作，且增加以下操作：插入、删除给定键的元素、查找目标键。

## 二、概要设计

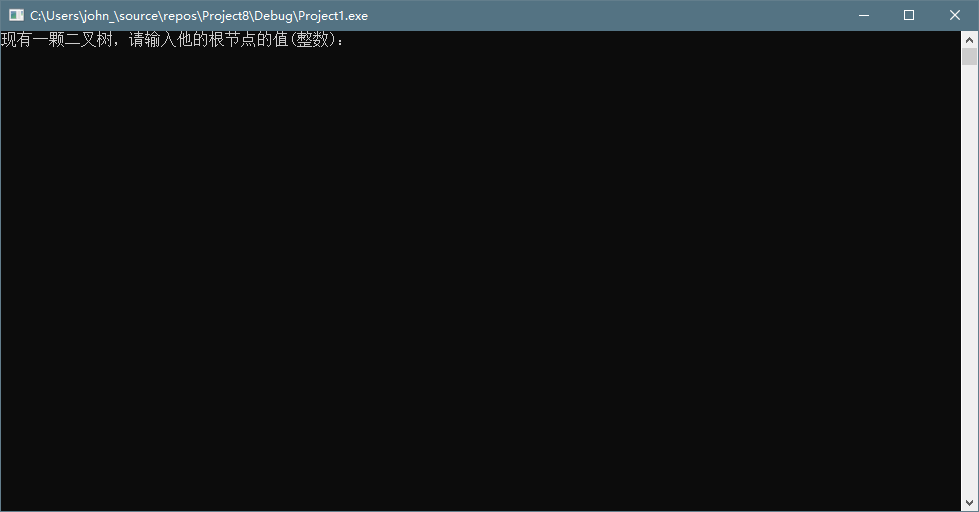
**（1）对实验内容的理解和二次概括。**

**该实验需要使用模板类，设计一个二叉树的类，并且实现在图中添加节点，遍历以及计算高度，删除等算法的功能。**

**（2）给出系统的功能列表**

* **添加节点构建二叉树**
* **前序中序后序遍历**
* **输出高度，尺寸**
* **删除节点**

**（3）程序运行的界面设计**



**（4）程序结构设计，包括：对已有程序的使用，自己将设计哪些程序文件，各部分关系描述。**

将设计一个二叉树类，利用链表和的结构存放节点和节点的左子树，右子树。利用递归算法简便写出遍历和添加删除结点的算法

## 三、详细设计

开始

按序生成二叉树

进行前序中序遍历

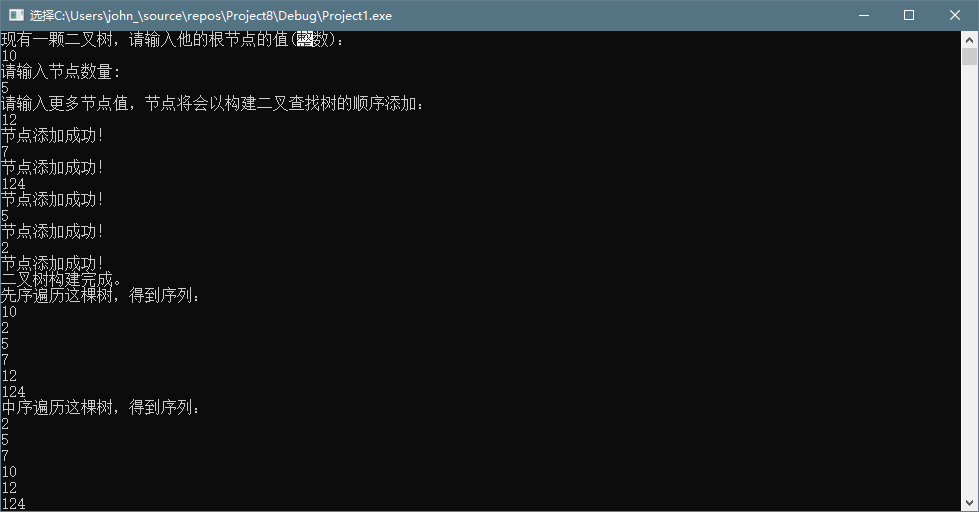
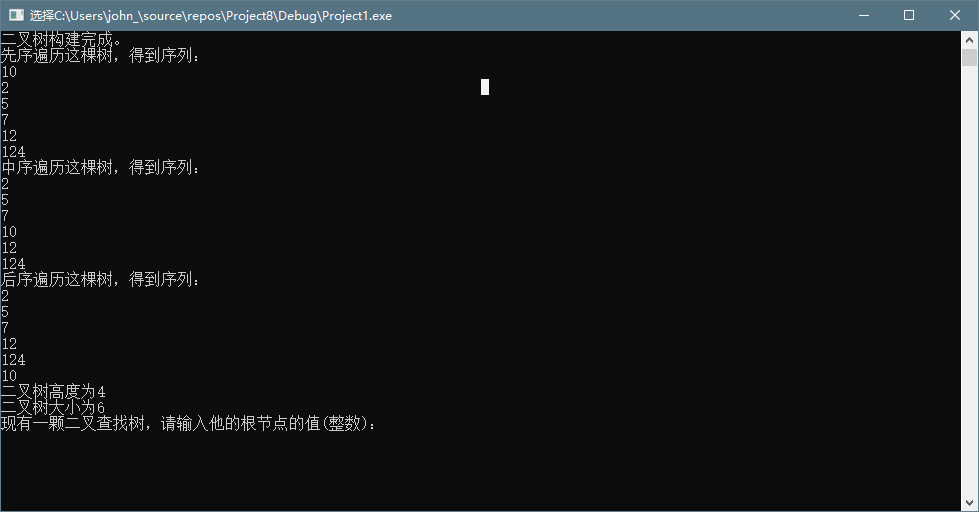
输入需要添加的节点

根据要求删除节点

终止

## 四、实验结果

紧扣题目要求设计提供相应的测试方法和结果。可以给出具体的测试用例，每个测试用例一般可列出：

* 测试输入：
* 
* 测试目的：检查程序是否正常运作
* 正确输出：
* 实际输出：
* 
* 测试结论：通过

## 五、实验分析与探讨

测试结果的分析：结果较为正常。

**实验设计、实现过程中遇到的问题：当新建指针没有指定其left和right指针所指的是NULL时，系统有可能会认为其指针指向0xCDCDCDCD，从而引发错误。**

## 六、小结

本次实验完成了二叉树的构建，以及遍历，添加，删除的基本功能。主要掌握了二叉树结构的定义，和基本功能的递归算法实现。

## 附录：源代码

#include <iostream>

using namespace std;

template <class Entry>

struct Binary\_node

{

public:

// data members:

Entry data;

Binary\_node<Entry> \*left;

Binary\_node<Entry> \*right;

// constructors:

Binary\_node();

Binary\_node(const Entry &x)

{

data = x;

left = NULL;

right = NULL;

}

};

enum Error\_code

{

success, fail, not\_present, duplicate\_error

};

template<class Entry>

void print(Entry data)

{

cout << data << endl;

}

template <class Entry>

class Binary\_tree

{

protected:

// Add auxiliary function prototypes here.

Binary\_node<Entry> \*root;

public:

Binary\_tree();

Error\_code insert(const Entry &new\_data);

Error\_code search\_and\_insert(Binary\_node<Entry> \* &sub\_root, const Entry &new\_data);

bool empty() const;

void preorder(void(\*visit)(Entry &));

void recursive\_preorder(Binary\_node<Entry> \*sub\_root, void(\*visit)(Entry &));

void inorder(void(\*visit)(Entry &));

void recursive\_inorder(Binary\_node<Entry> \*sub\_root, void(\*visit)(Entry &));

void postorder(void(\*visit)(Entry &));

void recursive\_postorder(Binary\_node<Entry> \*sub\_root, void(\*visit)(Entry &));

int size() const;

int recursive\_size(Binary\_node<Entry> \*sub\_root) const;

void clear();

void recursive\_clear(Binary\_node<Entry> \* &sub\_root);

void deleteleaf();

void recursive\_deleteleaf(Binary\_node<Entry>\* &sub\_root);

int height() const;

int recursive\_height(Binary\_node<Entry> \*sub\_root) const;

Binary\_node<Entry> \*recursive\_copy(Binary\_node<Entry> \*sub\_root);

Binary\_tree(const Binary\_tree<Entry> &original);

Binary\_tree & operator = (const Binary\_tree<Entry> &original);

void set\_root(Entry x)

{

root = new Binary\_node<Entry>(x);

}

};

template <class Entry>

Error\_code Binary\_tree<Entry> ::insert(const Entry &new\_data)

{

return search\_and\_insert(root, new\_data);

}

template <class Entry>

Error\_code Binary\_tree<Entry> ::search\_and\_insert(Binary\_node<Entry> \* &sub\_root, const Entry &new\_data)

{

if (sub\_root == NULL)

{

sub\_root = new Binary\_node<Entry>(new\_data);

return success;

}

else if (new\_data < sub\_root->data)

return search\_and\_insert(sub\_root->left, new\_data);

else if (new\_data > sub\_root->data)

return search\_and\_insert(sub\_root->right, new\_data);

else return duplicate\_error;

}

template <class Entry>

int Binary\_tree<Entry>::size() const

{

return recursive\_size(root);

}

template <class Entry>

int Binary\_tree<Entry> ::recursive\_size(Binary\_node<Entry> \*sub\_root) const

/\* Post: The number of entries in the subtree rooted at sub\_root is returned. \*/

{

if (sub\_root == NULL) return 0;

return 1 + recursive\_size(sub\_root->left) + recursive\_size(sub\_root->right);

}

template <class Entry>

int Binary\_tree<Entry>::height() const

{

return recursive\_height(root);

}

template <class Entry>

int Binary\_tree<Entry> ::recursive\_height(Binary\_node<Entry> \*sub\_root) const

/\* Post: The height of the subtree rooted at sub\_root is returned. \*/

{

if (sub\_root == NULL) return 0;

int l = recursive\_height(sub\_root->left);

int r = recursive\_height(sub\_root->right);

if (l > r) return 1 + l;

else return 1 + r;

}

template <class Entry>

Binary\_tree<Entry> ::Binary\_tree()

/\* Post: An empty binary tree has been created. \*/

{

root = nullptr;

}

template <class Entry>

bool Binary\_tree<Entry> ::empty() const

/\* Post: A result of true is returned if the binary tree is empty. Otherwise, false is returned. \*/

{

return root == NULL;

}

template <class Entry>

void Binary\_tree<Entry> ::clear()

{

recursive\_clear(root);

}

template <class Entry>

void Binary\_tree<Entry>::recursive\_clear(Binary\_node<Entry> \* &sub\_root)

{

if (sub\_root != NULL)

{

recursive\_clear(sub\_root->left);

recursive\_clear(sub\_root->right);

delete sub\_root;

sub\_root = NULL;

}

}

template <class Entry>

void Binary\_tree<Entry> ::preorder(void(\*visit)(Entry &))

/\* Post: The tree has been been traversed in inorder sequence.

Uses: The function recursive\_inorder \*/

{

recursive\_preorder(root, visit);

}

template <class Entry>

void Binary\_tree<Entry> ::recursive\_preorder(Binary\_node<Entry> \*sub\_root, void(\*visit)(Entry &))

{

if (sub\_root != NULL)

{

(\*visit)(sub\_root->data);

recursive\_inorder(sub\_root->left, visit);

recursive\_inorder(sub\_root->right, visit);

}

}

template <class Entry>

void Binary\_tree<Entry> ::inorder(void(\*visit)(Entry &))

/\* Post: The tree has been been traversed in inorder sequence.

Uses: The function recursive\_inorder \*/

{

recursive\_inorder(root, visit);

}

template <class Entry>

void Binary\_tree<Entry> ::recursive\_inorder(Binary\_node<Entry> \*sub\_root, void(\*visit)(Entry &))

{

if (sub\_root != NULL)

{

recursive\_inorder(sub\_root->left, visit);

(\*visit)(sub\_root->data);

recursive\_inorder(sub\_root->right, visit);

}

}

template <class Entry>

void Binary\_tree<Entry> ::postorder(void(\*visit)(Entry &))

/\* Post: The tree has been been traversed in inorder sequence.

Uses: The function recursive\_inorder \*/

{

recursive\_postorder(root, visit);

}

template <class Entry>

void Binary\_tree<Entry> ::recursive\_postorder(Binary\_node<Entry> \*sub\_root, void(\*visit)(Entry &))

{

if (sub\_root != NULL)

{

recursive\_inorder(sub\_root->left, visit);

recursive\_inorder(sub\_root->right, visit);

(\*visit)(sub\_root->data);

}

}

template <class Entry>

Binary\_tree<Entry>::Binary\_tree(const Binary\_tree<Entry> &original)

{

root = recursive\_copy(original.root);

}

template <class Entry>

Binary\_node<Entry> \*Binary\_tree<Entry> ::recursive\_copy(Binary\_node<Entry> \*sub\_root)

/\* Post: The subtree rooted at sub\_root is copied, and a pointer to the root of the new copy is returned. \*/

{

if (sub\_root == NULL) return NULL;

Binary\_node<Entry> \*temp = new Binary\_node<Entry>(sub\_root->data);

temp->left = recursive\_copy(sub\_root->left);

temp->right = recursive\_copy(sub\_root->right);

return temp;

}

template <class Entry>

void Binary\_tree<Entry>::recursive\_deleteleaf(Binary\_node<Entry>\* &sub\_root)

{

if (sub\_root == NULL)

return;

if (sub\_root->left == NULL && sub\_root->right == NULL)

{

delete sub\_root; sub\_root = NULL; return;

}

recursive\_deleteleaf(sub\_root->left);

recursive\_deleteleaf(sub\_root->right);

}

template <class Entry>

void Binary\_tree<Entry>::deleteleaf()

{

recursive\_deleteleaf(root);

}

template <class Record>

class Search\_tree : public Binary\_tree<Record> {

public:

int size() const;

int recursive\_size(Binary\_node<Record> \*sub\_root) const;

Error\_code insert(const Record &new\_data);

Error\_code remove(const Record &old\_data);

Error\_code tree\_search(Record &target) const;

Binary\_node<Record>\* search\_for\_node(Binary\_node<Record>\* sub\_root, const Record &target) const;

Error\_code search\_and\_insert(Binary\_node<Record> \* &sub\_root, const Record &new\_data);

Error\_code search\_and\_destroy(Binary\_node<Record>\* &sub\_root, const Record &target);

Error\_code remove\_root(Binary\_node<Record> \* &sub\_root);

private:

// Add auxiliary function prototypes here.

Binary\_node<Record> \*root;

};

template <class Record>

int Search\_tree<Record>::size() const

{

return recursive\_size(root);

}

template <class Record>

int Search\_tree<Record> ::recursive\_size(Binary\_node<Record> \*sub\_root) const

/\* Post: The number of entries in the subtree rooted at sub\_root is returned. \*/

{

if (sub\_root == NULL) return 0;

return 1 + recursive\_size(sub\_root->left) + recursive\_size(sub\_root->right);

}

template <class Record>

Binary\_node<Record> \*Search\_tree<Record> ::search\_for\_node(Binary\_node<Record>\* sub\_root, const Record &target) const

{

if (sub\_root == NULL || sub\_root->data == target)

return sub\_root;

else if (sub\_root->data < target)

return search\_for\_node(sub\_root->right, target);

else return search\_for\_node(sub\_root->left, target);

}

template <class Record>

Error\_code Search\_tree<Record> ::tree\_search(Record &target) const

{

Error\_code result = success;

Binary\_node<Record> \*found = search\_for\_node(root, target);

if (found == NULL)

result = not\_present;

else

target = found->data;

return result;

}

template <class Record>

Error\_code Search\_tree<Record> ::insert(const Record &new\_data)

{

return search\_and\_insert(root, new\_data);

}

template <class Record>

Error\_code Search\_tree<Record> ::search\_and\_insert(Binary\_node<Record> \* &sub\_root, const Record &new\_data)

{

if (sub\_root == NULL)

{

sub\_root = new Binary\_node<Record>(new\_data);

return success;

}

else if (new\_data < sub\_root->data)

return search\_and\_insert(sub\_root->left, new\_data);

else if (new\_data > sub\_root->data)

return search\_and\_insert(sub\_root->right, new\_data);

else return duplicate\_error;

}

template <class Record>

Error\_code Search\_tree<Record> ::remove(const Record &target)

/\* Post: If a Record with a key matching(符合） that of target belongs to the Search\_tree a code of success is returned and the co. rresponding node is removed from the tree. Otherwise, a code of not\_present is returned.

Uses: Function search\_and\_destroy \*/

{

return search\_and\_destroy(root, target);

}

template <class Record>

Error\_code Search\_tree<Record> ::search\_and\_destroy(Binary\_node<Record>\* &sub\_root, const Record &target)

{

if (sub\_root == NULL || sub\_root->data == target)

return remove\_root(sub\_root);

else if (target < sub\_root->data)

return search\_and\_destroy(sub\_root->left, target);

else

return search\_and\_destroy(sub\_root->right, target);

}

template <class Record>

Error\_code Search\_tree<Record> ::remove\_root(Binary\_node<Record> \* &sub\_root)

{

if (sub\_root == NULL) return not\_present;

Binary\_node<Record> \*to\_delete = sub\_root;

// Remember node to delete at end.

if (sub\_root->right == NULL) sub\_root = sub\_root->left;

else if (sub\_root->left == NULL) sub\_root = sub\_root->right;

else { // Neither subtree is empty.

to\_delete = sub\_root->left;

// Move left to find predecessor(前驱）.

Binary\_node<Record> \*parent = sub\_root; // parent of to\_delete

while (to\_delete->right != NULL)

{ // to delete is not the predecessor.

parent = to\_delete;

to\_delete = to\_delete->right;

}

sub\_root->data = to\_delete->data; // Move from to\_delete to root.

if (parent == sub\_root) sub\_root->left = to\_delete->left;

else parent->right = to\_delete->left;

}

delete to\_delete; //Remove to\_delete from tree.

return success;

}

int main()

{

Binary\_tree<int> bt;

int root\_value,n,node\_value;

cout << "现有一颗二叉树，请输入他的根节点的值(整数)："<<endl;

cin >> root\_value;

bt.set\_root(root\_value);

cout << "请输入节点数量:"<<endl;

cin >> n;

cout << "请输入更多节点值，节点将会以构建二叉查找树的顺序添加："<<endl;

for (int i = 0; i < n; i++)

{

cin >> node\_value;

int result;

result=bt.insert(node\_value);

if (result == 0) cout << "节点添加成功！" << endl;

}

cout << "二叉树构建完成。" << endl;

cout << "先序遍历这棵树，得到序列：" << endl;

bt.preorder(print);

cout << "中序遍历这棵树，得到序列：" << endl;

bt.inorder(print);

cout << "后序遍历这棵树，得到序列：" << endl;

bt.postorder(print);

cout << "二叉树高度为" << bt.height() << endl;

cout << "二叉树大小为" << bt.size() << endl;

Search\_tree<int> st;

int root\_value2, n2, node\_value2;

cout << "现有一颗二叉查找树，请输入他的根节点的值(整数)：" << endl;

cin >> root\_value2;

st.set\_root(root\_value2);

cout << "请输入需要添加的节点数量:" << endl;

cin >> n;

cout << "请输入更多节点值，构建二叉查找树：" << endl;

for (int i = 0; i < n; i++)

{

cin >> node\_value2;

int result;

result = st.insert(node\_value2);

if (result == 0) cout << "节点添加成功！" << endl;

else cout << "节点删除失败！";

}

cout << "请输入需要删除的节点数量:" << endl;

cin >> n;

cout << "请输入需要删除的节点值：" << endl;

for (int i = 0; i < n; i++)

{

cin >> node\_value2;

int result;

result = st.remove(node\_value2);

if (result == 0) cout << "节点删除成功！" << endl;

else cout << "节点删除失败！" << endl;

}

system("pause");

}