|  |  |  |  |
| --- | --- | --- | --- |
| **学生学号** | 175064 | **实验课成绩** |  |

*河北工业大学*

**学 生 实 验 报 告 书**

|  |  |
| --- | --- |
| **实验课程名称** | 数据结构 |
| **开 课 学 院** | 国际教育学院 |
| **指导教师姓名** | 宛艳萍 |
| **学 生 姓 名** | 张心宇 |
| **学生专业班级** | 中法计172 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2018 | -- | 2019 | 学年 | 第 | 二 | 学期 |

# 实验内容

【基本要求】

（1） 以链表作为存储结构，实现二叉排序树的建立、查找和删除。

（2） 根据给定的数据建立平衡二叉树。

（3） 比较二叉排序树和平衡二叉树的平均查找长度。

【测试数据】

随机生成。

【实现提示】

（1） 初始，二叉排序树和平衡二叉树都为空树，操作界面给出查找、插入和删除三种操作供选择。每种操作均要提示输入关键字。每次插入或删除一个结点后，应更新平衡二叉树或二叉排序树的显示。

（2） 平衡二叉树或二叉排序树的显示可以采用凹入表形式，也可以采用图形界面画出树形。

# 实验目的

熟练掌握顺序查找、折半查找及二叉排序树、平衡二叉树上的查找、插入和删除的方法，比较它们的平均查找长度。

# 代码清单

## AVLTree.h

// 平衡二叉树  
// Created by gorkr on 19-6-5.  
// 多看书，多思考。  
  
// 体会 height 仅是高度，并不代表层次  
  
#ifndef SORT\_AVLTREE\_H  
#define SORT\_AVLTREE\_H  
  
#endif //SORT\_AVLTREE\_H  
  
#include <iostream>  
#include <stack>  
//#include "BinarySearchTree.h"  
  
  
  
// AVL树节点  
template <typename T>  
class AVLNode : public BinarySearchTreeNode<T>{  
public:  
 int height;  
 AVLNode \* lchild,\* rchild;  
 AVLNode(T data):  
 BinarySearchTreeNode<T>(data),height(0),lchild(nullptr),rchild(nullptr){}  
 AVLNode(T data, AVLNode\* l, AVLNode\* r):  
 BinarySearchTreeNode<T>(data),height(0),lchild(l),rchild(r){}  
 ~AVLNode()= default;  
  
 AVLNode<T>\* leftLeftRotation(AVLNode<T>\* n2);  
 AVLNode<T>\* rightRightRotation(AVLNode<T>\* n1);  
 AVLNode<T>\* leftRightRotation(AVLNode<T>\* n3);  
 AVLNode<T>\* rightLeftRotation(AVLNode<T>\* n1);  
 void dispTree();  
  
 int getHight(AVLNode<T>\* t);  
 int setHight(AVLNode<T>\* &t);  
  
 AVLNode<T>\*& insert(AVLNode<T>\* &t, T key);  
};  
  
template <typename T>  
void AVLNode<T>::dispTree() {  
 std::stack<AVLNode<T>\*> s;  
 std::stack<int> level;  
 std::stack<std::string> type;  
 int width = 4;  
 int above\_width=0;  
 AVLNode<T> \*p;  
 p=this;  
  
 // 先将根入栈  
 s.push(p);  
 level.push(width);  
 type.push("根");  
  
 while(s.size()){  
 p=s.top();s.pop();  
 above\_width = level.top(); level.pop();  
  
 for(int i=1; i<=above\_width; i++){  
 std::cout<<"-";  
 }  
 std::cout<<p->data<<"("<<type.top()<<")"<<std::endl;  
 type.pop();  
  
 if(p->lchild!= nullptr){  
 s.push(p->lchild);  
 level.push(above\_width+width);  
 type.push("左");  
 }  
 if(p->rchild!= nullptr){  
 s.push(p->rchild);  
 level.push(above\_width+width);  
 type.push("右");  
 }  
  
 }  
  
}  
  
  
template <typename T>  
int AVLNode<T>::getHight(AVLNode<T> \*t) {  
 if(t!= nullptr)return t->height;  
 return 0;  
}  
  
template <typename T>  
int AVLNode<T>::setHight(AVLNode<T> \*&t) {  
 return t->height = (getHight(t->lchild)>getHight(t->rchild)?getHight(t->lchild):getHight(t->rchild))+1;  
}  
  
  
/\*\*  
 \*  
 \* @tparam T  
 \* @param n2 失衡节点  
 \* @return 平衡后的根节点  
 \*/  
template <typename T>  
AVLNode<T>\* AVLNode<T>::leftLeftRotation(AVLNode<T> \*n2) {  
 AVLNode<T>\* n1 = n2->lchild;  
 n2->lchild = n1->rchild;  
 n1->rchild = n2;  
  
 // 次序不能乱  
 setHight(n2);  
 setHight(n1);  
 return n1;  
}  
  
  
  
template <typename T>  
AVLNode<T>\* AVLNode<T>::rightRightRotation(AVLNode<T> \*n1) {  
 AVLNode<T>\* n2;  
 n2 = n1->rchild;  
  
 n1->rchild = n2->lchild;  
 n2->lchild = n1;  
  
 // 次序不能乱  
 setHight(n1);  
 setHight(n2);  
 return n2;  
}  
  
  
template <typename T>  
AVLNode<T>\* AVLNode<T>::leftRightRotation(AVLNode<T> \*n3) {  
 n3->lchild = rightRightRotation(n3->lchild);  
 return leftLeftRotation(n3);  
}  
  
template <typename T>  
AVLNode<T>\* AVLNode<T>::rightLeftRotation(AVLNode<T> \*n1) {  
 n1->rchild = leftLeftRotation(n1->rchild);  
 return rightRightRotation(n1);  
}  
  
  
  
/\*\*  
 \*将节点插入t节点以下，并返回根节点  
 \* @tparam T  
 \* @param t  
 \* @param key  
 \* @return 根节点  
 \*/  
template <class T>  
AVLNode<T>\*& AVLNode<T>::insert(AVLNode<T>\* &t, T key)  
{  
 if (t == nullptr) // 这是第一个节点  
 t = new AVLNode<T>(key);  
 else if (key < t->data) // 应该将key插入到"tree的左子树"的情况  
 {  
 t->lchild = insert(t->lchild, key);  
 // 插入节点后，若AVL树失去平衡，则进行相应的调节。  
 if (getHight(t->lchild) - getHight(t->rchild) == 2)  
 {  
 if (key < t->lchild->data)  
 t = leftLeftRotation(t);  
 else  
 t = leftRightRotation(t);  
 }  
 }  
 else if (key > t->data) // 应该将key插入到"tree的右子树"的情况  
 {  
 t->rchild = insert(t->rchild, key);  
 // 插入节点后，若AVL树失去平衡，则进行相应的调节。  
 if (getHight(t->rchild) - getHight(t->lchild) == 2)  
 {  
 if (key > t->rchild->data)  
 t = rightRightRotation(t);  
 else  
 t = rightLeftRotation(t);  
 }  
 }  
  
 setHight(t);  
 return t;  
}

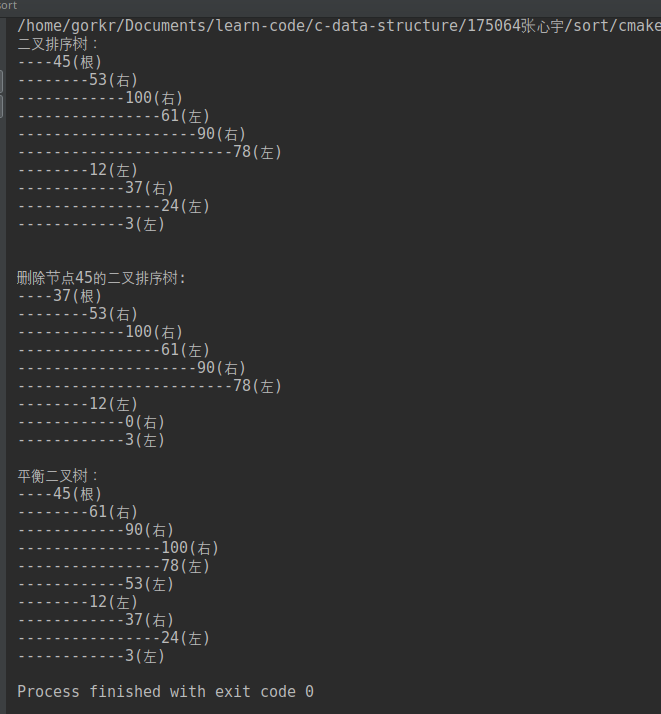
## BinarySearchTree.h

// 二叉搜索树  
// Created by gorkr on 19-6-5.  
//  
  
#ifndef SORT\_BINARYSEARCHTREE\_H  
#define SORT\_BINARYSEARCHTREE\_H  
  
#endif //SORT\_BINARYSEARCHTREE\_H  
  
  
#include <iostream>  
#include <vector>  
#include <stack>  
  
template <typename T>  
class BinarySearchTreeNode{  
 typedef BinarySearchTreeNode Node;  
public:  
 T data;  
 Node \* lchild;  
 Node \* rchild;  
public:  
  
 //friend bool operator!=(const BinarySearchTreeNode<T>\*&, const BinarySearchTreeNode<T>\*&)= default;  
 BinarySearchTreeNode(T data):data(data),lchild(nullptr),rchild(nullptr){};  
 ~BinarySearchTreeNode(){  
 delete(lchild);  
 delete(rchild);  
 }  
 // BinarySearchTreeNode<T> &operator=(const BinarySearchTreeNode<T> &);  
 // 从当前节点向下搜索 \_hot变量记录被搜索到节点的父节点  
 BinarySearchTreeNode<T>\* search(T key,BinarySearchTreeNode<T> \*&\_hot); // 引用作为返回值，可以放在等号左边 // 指针的引用  
 BinarySearchTreeNode<T>\* search(T key);  
 // 从当前节点向下插入；  
 void insert(T key);  
 // 从当前节点向下删除  
 bool del(T key);  
 void traverseInOrder(void (\*visit)(T& e));  
 // 从当前节点向下打印二叉树  
 void dispTree();  
  
};  
  
  
template <typename T>  
class BinarySearchTree{  
private:  
 BinarySearchTreeNode<T>\* root;  
public:  
 BinarySearchTree():root(nullptr){}  
 void insert(T key){  
 if (root) root->insert(key);  
 else  
 root = new BinarySearchTreeNode<T>(key);  
 }  
 bool search(T key){  
 if(root) root->search(key);  
 else return false;  
 }  
 void del(T key){  
 if(root) root->del(key);  
 }  
 void dispTree(){  
 if(root) root->dispTree();  
 }  
 void findIt(T key){  
 auto p = search(key);  
 if(p==nullptr)std::cout<<"没有找到元素"<<key<<std::endl;  
 else std::cout<<"找到元素"<<key<<std::endl;  
 }  
};  
  
  
  
template <typename T>  
void BinarySearchTreeNode<T>::dispTree() {  
 std::stack<BinarySearchTreeNode<T>\*> s;  
 std::stack<int> level;  
 std::stack<std::string> type;  
 int width = 4;  
 int above\_width=0;  
 BinarySearchTreeNode<T> \*p;  
 p=this;  
  
 // 先将根入栈  
 s.push(p);  
 level.push(width);  
 type.push("根");  
  
 while(s.size()){  
 p=s.top();s.pop();  
 above\_width = level.top(); level.pop();  
  
 for(int i=1; i<=above\_width; i++){  
 std::cout<<"-";  
 }  
 std::cout<<p->data<<"("<<type.top()<<")"<<std::endl;  
 type.pop();  
  
 if(p->lchild!= nullptr){  
 s.push(p->lchild);  
 level.push(above\_width+width);  
 type.push("左");  
 }  
 if(p->rchild!= nullptr){  
 s.push(p->rchild);  
 level.push(above\_width+width);  
 type.push("右");  
 }  
  
 }  
  
}  
  
  
template <typename T>  
void BinarySearchTreeNode<T>::traverseInOrder(void (\*visit)(T & e)){  
 if(this->lchild) this->lchild->traverseInOrder(visit);  
 visit(data);  
 if(this->rchild) this->rchild->traverseInOrder(visit);  
}  
  
template <typename T>  
BinarySearchTreeNode<T>\* BinarySearchTreeNode<T>::search(T key,BinarySearchTreeNode<T> \* &\_hot) {  
 if(this->data==key)return(this);  
 else if(key<this->data){  
 if(lchild){  
 \_hot = this;  
 return(this->lchild->search(key, \_hot));  
 }  
 else  
 return nullptr;  
 }  
 else {  
 if(rchild){  
 \_hot = this;  
 return(this->rchild->search(key, \_hot));  
 }  
 else  
 return nullptr;  
 }  
  
}  
  
template <typename T>  
BinarySearchTreeNode<T>\* BinarySearchTreeNode<T>::search(T key){  
 if(this->data==key)return(this);  
 else if(key<this->data){  
 if(lchild)  
 return(this->lchild->search(key));  
 else  
 return nullptr;  
 }  
 else {  
 if(rchild){  
 return(this->rchild->search(key));  
 }  
 else  
 return nullptr;  
 }  
}  
  
  
template <typename T>  
void BinarySearchTreeNode<T>::insert(T key) {  
 if(key<data){  
 if(lchild)  
 lchild->insert(key);  
 else  
 lchild = new BinarySearchTreeNode<T>(key);  
 }else if(key>data){  
 if(rchild)  
 rchild->insert(key);  
 else  
 rchild = new BinarySearchTreeNode<T>(key);  
 }  
}  
  
template <typename T>  
bool BinarySearchTreeNode<T>::del(T key){  
 //BinarySearchTreeNode<T>\* parent = nullptr;  
 BinarySearchTreeNode<T>\* succ = nullptr; // 实际删除节点的阶梯这  
 BinarySearchTreeNode<T>\* d = search(key);  
 if(!d) return false;  
  
 if(d->lchild== nullptr){  
 succ = d->rchild;  
 if(succ){  
 \*d = \*succ;  
 delete(succ);  
 }else{  
 delete(d);  
 }  
  
  
 }  
  
 else if (d->rchild== nullptr){  
 succ = d->lchild;  
 if(succ){  
 \*d = \*succ;  
 delete(succ);  
 }else{  
 delete(d);  
 }  
 }  
 else{  
 BinarySearchTreeNode<T>\* exchnge=d->lchild; // 记录d节点的直接前驱节点， 左转  
 BinarySearchTreeNode<T>\* p = d; // 记录交换节点的父节点  
 while(exchnge->rchild){  
 p = exchnge;  
 exchnge = exchnge->rchild;  
 }  
 d->data = exchnge->data;  
 ((p!=d)?p->rchild:p->lchild) = exchnge->lchild;  
 delete(exchnge);  
 }  
}

## main.cpp

// 引用作为函数参数，直接影响参数变量  
// 形参可以和属性命名相同  
// 模板类外定义函数  
// 必须先判定对象不是null 然后在调用方法 不然就会 段错误  
// 当我们把一个指针做为参数传一个方法时，其实是把指针的复本传递给了方法，也可以说传递指针是指针的值传递。  
  
  
/\*\*  
 \* 二叉查找树给我们带来了很多方便，但是由于其在有序序列插入时就会退化成单链表（时间复杂度退化成 O(n)），AVL-tree就克服了上述困难。  
 \*/  
  
#include <iostream>  
#include "BinarySearchTree.h"  
#include "AVLTree.h"  
  
void testBST(){  
 std::vector<int> raw{45,12,53,3,37,100,24,61,90,78};  
  
 std::cout<<"二叉排序树："<<std::endl;  
 BinarySearchTree<int> tree;  
 for(int i=0; i<10; i++){  
 tree.insert(raw[i]);  
 }  
  
  
 tree.dispTree();  
 //auto p = tree.search(45);  
 std::cout<<bool(tree.search(45))<<std::endl;  
  
  
 std::cout<<"\n\n删除节点45的二叉排序树:"<<std::endl;  
 tree.del(45);  
 tree.dispTree();  
 }  
  
void testAVL(){  
 std::vector<int> raw{45,12,53,3,37,100,24,61,90,78};  
  
 std::cout<<"平衡二叉树："<<std::endl;  
 AVLNode<int>\* tree= nullptr;  
 for(int i=0; i<10; i++){  
 tree->insert(tree, raw[i]);  
 }  
 tree->dispTree();  
}  
  
int main(){  
 testBST();  
 std::cout<<std::endl;  
 testAVL();  
}

# 运行结果



# 分析与思考

1. 理解平衡二叉树中旋转的概念。
2. 测试驱动开发，最起码写一块测试一块。
3. 学习泛型编程。