**2022212153 陈祥烨 第十章作业**

**第一题**

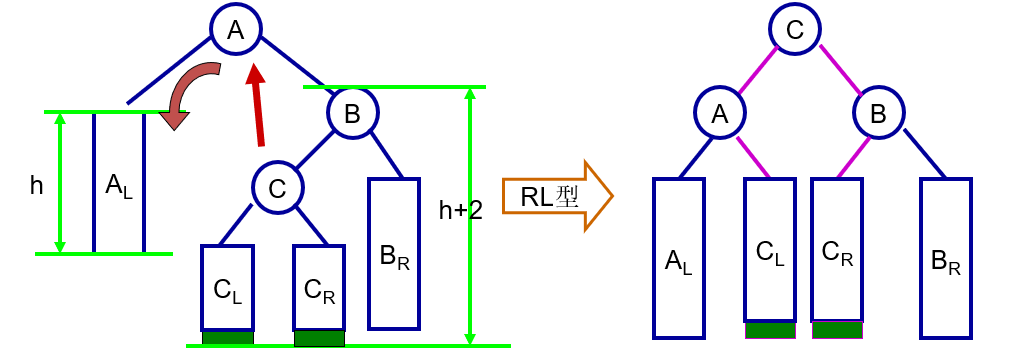
**题目：**

**② 在构造平衡二叉树的过程中，若A的平衡因子变为-2，A的左孩子的平衡因子为0，右孩子的平衡因子为1，则进行何种调整？**

**答：**

由题目可知，不平衡的类型为 RL 型，即 A 右孩子的左子树高了。

解决方案为 A 换为 A 的右孩子的左子树的根 C ， A 成为 C 的左子树的根，A 原先的右子树根 B 作为 C 的右子树的根，现在还需分配 C 的左子树和右子树， C 的左子树变为 A 的右子树， C 的右子树变为 B 的右子树。

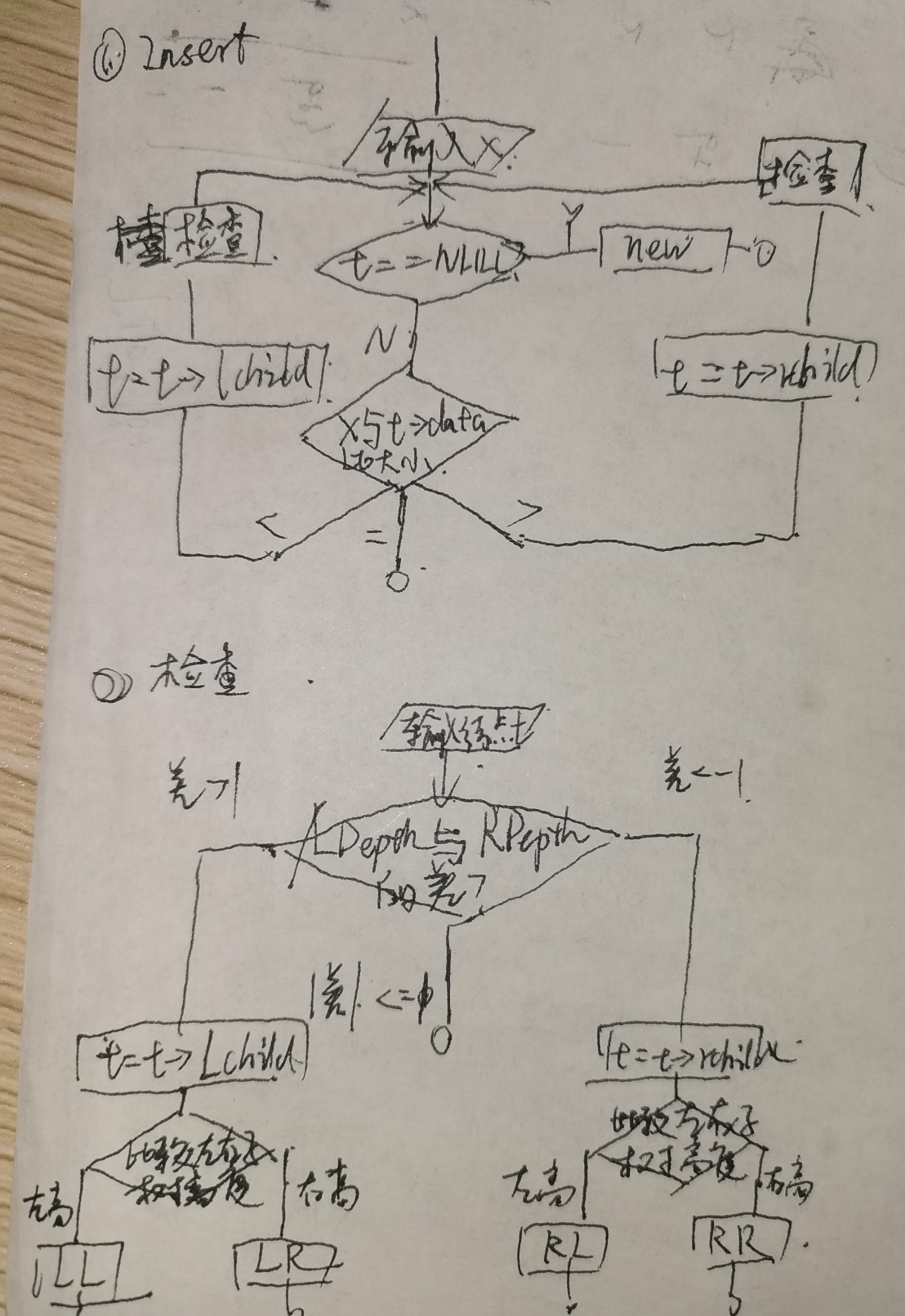
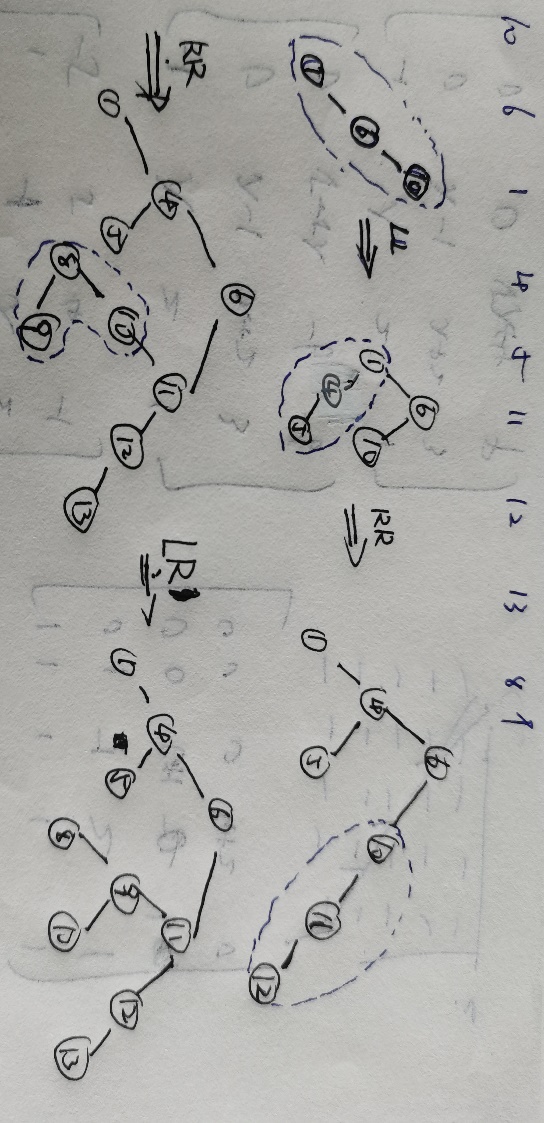
****

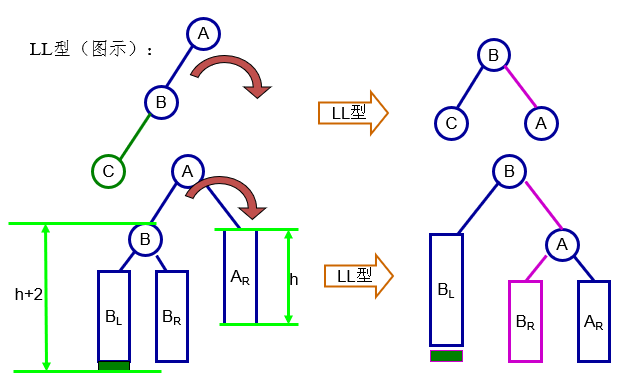
**第二题**

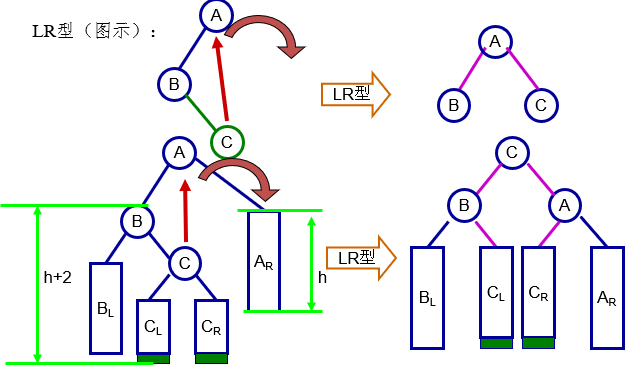
**题目：**

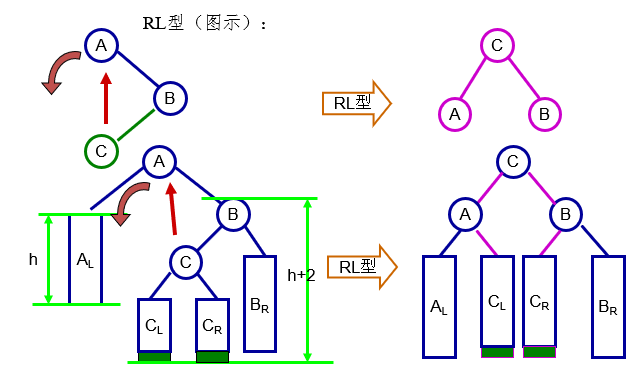
**根据输入序列10，6，1，4，5，11，12，13， 8， 9，构造一棵平衡二叉树，要求写出构建过程。**

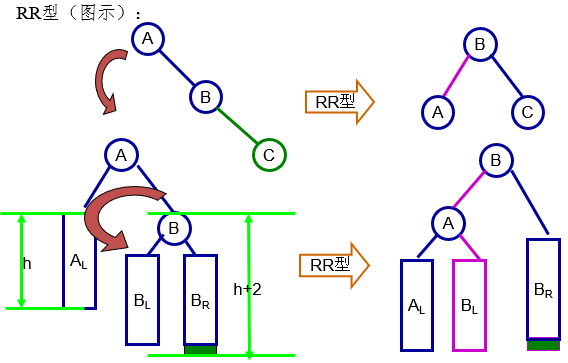
**思路:**

 ****

****

****

****

****

**代码：**

**AvlTree.h**

#pragma once

#include<iostream>

using namespace std;

struct AvlNode

{

int data; //值域

AvlNode\* rchild, \* lchild;

AvlNode(int d)

{

data = d;

rchild = NULL;

lchild = NULL;

}

};

class AvlTree

{

public:

//构造函数

AvlTree();

~AvlTree() {};

void Insert(int x); //插入元素对外接口

void PreorderTraversal(); //先序遍历对外接口

void InorderTraversal(); //中序遍历对外接口

private:

//插入元素内部实现

void Insert(AvlNode\* &t, int x);

//四种类型转化

AvlNode\* LLadjust(AvlNode\* a);

AvlNode\* LRadjust(AvlNode\* a);

AvlNode\* RLadjust(AvlNode\* a);

AvlNode\* RRadjust(AvlNode\* a);

//遍历

void PreorderTraversal(AvlNode\* t); //前序遍历

void InorderTraversal(AvlNode\* t); //中序遍历

//相关功能

int GetHeight(AvlNode\* t); //得到高度

void Delete(AvlNode\* t); //删除以此结点为根的树

private:

AvlNode\* root;

};

**AvlTree.cpp**

#include"AvlTree.h"

//----------------------------函数的构造和析构---------------------------------------

AvlTree::AvlTree()

{

root = NULL;

}

AvlTree::~AvlTree()

{

Delete(root);

}

//----------------------------遍历---------------------------------------

void AvlTree::PreorderTraversal() //先序遍历对外接口

{

PreorderTraversal(root);

}

void AvlTree::InorderTraversal() //中序遍历对外接口

{

InorderTraversal(root);

}

void AvlTree::PreorderTraversal(AvlNode\* t) //先序遍历

{

if (t == NULL) { return; }

cout << t->data << " ";

PreorderTraversal(t->lchild);

PreorderTraversal(t->rchild);

}

void AvlTree::InorderTraversal(AvlNode\* t) //中序遍历

{

if (t == NULL) { return; }

InorderTraversal(t->lchild);

cout << t->data << " ";

InorderTraversal(t->rchild);

}

//----------------------------插入---------------------------------------

void AvlTree::Insert(int x)

{

Insert(root, x);

}

void AvlTree::Insert(AvlNode\* &t, int x)

{

if (t == NULL)

{

t= new AvlNode(x);

return;

}

else if (x < t->data)

{

Insert(t->lchild, x);

if (GetHeight(t->lchild) - GetHeight(t->rchild) > 1) //左子树高度 比 右子树高度 大2

{

AvlNode\* left = t->lchild;

if (GetHeight(left->lchild) > GetHeight(left->rchild)) //LL型

{

t = LLadjust(t);

}

else //LR型

{

t = LRadjust(t);

}

}

}

else if (x > t->data)

{

Insert(t->rchild, x);

if (GetHeight(t->rchild) - GetHeight(t->lchild) > 1) //右子树高度 比 左子树高度 大2

{

AvlNode\* right = t->rchild;

if (GetHeight(right->lchild) > GetHeight(right->rchild)) //RL型

{

t = RLadjust(t);

}

else //RR型

{

t = RRadjust(t);

}

}

}

}

//----------------------------位置调整---------------------------------------

AvlNode\* AvlTree::LLadjust(AvlNode\* a)

{

AvlNode\* b = a->lchild;

a->lchild = b->rchild;

b->rchild = a;

return b;

}

AvlNode\* AvlTree::LRadjust(AvlNode\* a)

{

AvlNode\* c = a->lchild->rchild;

AvlNode\* b = a->lchild;

b->rchild = c->lchild;

a->lchild = c->rchild;

c->lchild = b;

c->rchild = a;

return c;

}

AvlNode\* AvlTree::RLadjust(AvlNode\* a)

{

AvlNode\* c = a->rchild->lchild;

AvlNode\* b = a->rchild;

b->lchild = c->rchild;

a->rchild = c->lchild;

c->lchild = a;

c->rchild = b;

return c;

}

AvlNode\* AvlTree::RRadjust(AvlNode\* a)

{

AvlNode\* b = a->rchild;

a->rchild = b->lchild;

b->lchild = a;

return b;

}

//----------------------------工具包---------------------------------------

int max(const int n, const int m)

{

return (m > n) ? m : n;

}

//----------------------------相关功能---------------------------------------

int AvlTree::GetHeight(AvlNode\* t)

{

if (t == NULL)return 0;

return max(GetHeight(t->lchild), GetHeight(t->rchild))+1;

}

void AvlTree::Delete(AvlNode\* t)

{

if (t == NULL)return;

Delete(t->lchild);

Delete(t->rchild);

delete t;

}

**test.cpp**

#include"AvlTree.h"

void test()

{

AvlTree tree;

int value;

while(cin>>value)

{

if (value == -1)break;

tree.Insert(value);

tree.PreorderTraversal(); cout << endl;

tree.InorderTraversal(); cout << endl;

}

cout << "先序遍历:" << endl;

tree.PreorderTraversal();cout << endl;

cout << "中序遍历:" << endl;

tree.InorderTraversal();cout << endl;

}

int main()

{

test();

return 0;

}

**测试：**

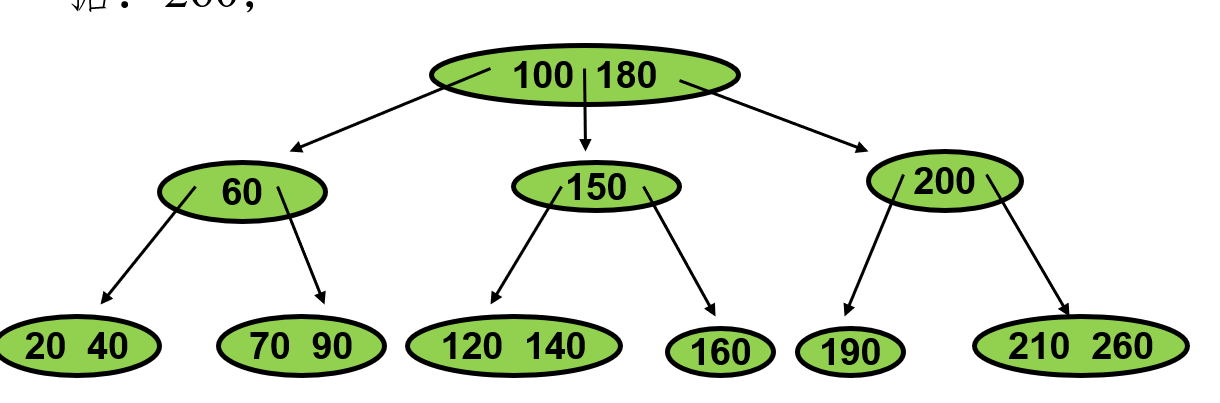
**10 6 1 4 5 11 12 13 8 9 -1**

****

**第三题**

**题目：**

**3阶B-树的插入与删除操作，插入数据：80，195，删除数据：260；**

****

**思路：**

插入方法：

首先，插入到叶子结点中，通过递归完成插入操作。完成插入后，每次对遍历过得结点进行检查：若不溢出，如果为根结点，则将data[0]和data[2]分裂出来做子结点存储到chilld[0],child[1]，data[1]作头结点，如果不为根结点，data[1]插入到父结点中，将列出来的按照顺序插入到父结点的孩子数组child中。

删除关键字

若删除的关键字不在叶子结点中，用其前驱或后继来替换

删除叶子结点中的关键字

删除关键字后依然能满足B-树的条件，则结束

否则，若左右兄弟中有多余的关键字，则调整一下，否则合三为一。

**代码：**

**Test.cpp**

#include"BTree.h"

void test()

{

BTree tree;

int value;

while (cin >> value)

{

if (value == -1)break;

tree.Insert(value);

}

}

int main()

{

test();

return 0;

}

**BTree.h**

#pragma once

#ifndef BTREE\_H

#define BTREE\_H

#include<iostream>

using namespace std;

struct BTNode //三阶B-树结点

{

bool isleaf; //是否为叶子结点 True 为是，FALSE为否

int count; //当前元素数量

int data[3]; //元素

BTNode\* parent; //父节点

BTNode\* child[4]; //孩子结点数组

BTNode();

BTNode(int x);

void Insert(int x); //叶子结点元素插入

};

class BTree

{

private:

BTNode\* root;

public:

BTree();

~BTree();

void Insert(int x); //插入元素对外接口

private:

void Insert(BTNode\* t, int x); //插入元素内部实现

void Destroy(BTNode\* t);

};

#endif // !BTREE\_H

**BTree.cpp**

#include"BTree.h"

//----------------------- BTNode 结构体的构造与函数实现 --------------------------

BTNode::BTNode()

{

count = 0;

isleaf = true;

parent = NULL;

for (int i = 0; i < 3; i++)

{

child[i] = NULL;

}

}

BTNode::BTNode(int x)

{

count = 1;

isleaf = true;

parent = NULL;

data[0] = x;

for (int i = 0; i < 3; i++)

{

child[i] = NULL;

}

}

void BTNode::Insert(int x)

{

if (count == 0)data[0] = x;

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

{

if (x < data[i])

{

for (int j = count; i < j; j--)

{

data[j] = data[j - 1];

}

data[i] = x;

}

else if (data[i] == x)break;

else if (i + 1 == count)

{

data[count] = x;

}

}

count++;

}

//----------------------- BTree 类的构造与函数实现 --------------------------

BTree::BTree()

{

root = new BTNode;

}

BTree::~BTree()

{

Destroy(root);

}

void BTree::Insert(int x)

{

Insert(root, x);

}

void BTree::Insert(BTNode\* t, int x)

{

//插入

if (t->isleaf) //叶子结点

{

t->Insert(x);

}

else //非叶子结点

{

if (t->count == 1) //元素个数为 1

{

if (x > t->data[0])

Insert(t->child[0], x);

else

Insert(t->child[1], x);

}

else if (t->count == 2) //元素个数为 2

{

if (x < t->data[0])

Insert(t->child[0], x);

else if (x > t->data[1])

Insert(t->child[2], x);

else

Insert(t->child[1], x);

}

}

//检查

if (t->count == 3)

{

if (t->parent == NULL) //如果为根结点

{

t->child[0] = new BTNode(t->data[0]);

t->child[0]->parent = t;

t->child[1] = new BTNode(t->data[2]);

t->child[1]->parent = t;

t->count = 1;

t->data[0] = t->data[1];

t->isleaf = false;

}

else //如果不为根结点

{

//分裂

BTNode\* node1 = new BTNode(t->data[0]);

node1->parent = t->parent;

node1->child[0] = t->child[0];

node1->child[1] = t->child[1];

for (int i = 0; i < 4; i++)

{

if (node1->child[i] != NULL)node1->isleaf = false;

}

BTNode\* node2 = new BTNode(t->data[2]);

node2->parent = t->parent;

node2->child[0] = t->child[2];

node2->child[1] = t->child[3];

for (int i = 0; i < 4; i++)

{

if (node1->child[i] != NULL)node1->isleaf = false;

}

//连接

if (t->parent->count == 1) //如果父结点的元素数量为 1

{

if (t->data[1] < t->parent->data[0])

{

t->parent->data[1] = t->parent->data[0];

t->parent->data[0] = t->data[1];

t->parent->count++;

t->parent->child[2] = t->parent->child[1];

t->parent->child[0] = node1;

t->parent->child[1] = node2;

}

else

{

t->parent->data[1] = t->data[1];

t->parent->count++;

t->parent->child[1] = node1;

t->parent->child[2] = node2;

}

}

else //如果父结点的元素数量为 2

{

if (t->data[1] < t->parent->data[0])

{

t->parent->data[2] = t->parent->data[1];

t->parent->data[1] = t->parent->data[0];

t->parent->data[0] = t->data[1];

t->parent->count++;

t->parent->child[3] = t->parent->child[2];

t->parent->child[2] = t->parent->child[1];

t->parent->child[0] = node1;

t->parent->child[1] = node2;

}

else if (t->data[1] < t->parent->data[2])

{

t->parent->data[2] = t->data[1];

t->parent->count++;

t->parent->child[2] = node1;

t->parent->child[3] = node2;

}

else

{

t->parent->data[2] = t->parent->data[1];

t->parent->data[1] = t->data[1];

t->parent->count++;

t->parent->child[3] = t->parent->child[2];

t->parent->child[1] = node1;

t->parent->child[2] = node2;

}

}

}

}

}

void BTree::Destroy(BTNode\* t)

{

if (t == NULL)

{

return;

}

for (int i = 0; t->child[i] != NULL || i< t->count; i++)

{

Destroy(t->child[i]);

}

delete t;

t = NULL;

}

**第四题**

**题目：**

**基于三种数据结构的查找方法的优缺点？**

顺序表的优点：

无须增加额外的存储空间表示结点间的逻辑关系。

可以方便地随机存取表中任一结点。

顺序表的缺点：

插入和删除运算不方便，通常须移动大量结点，效率较低。

难以进行连续的存储空间的预分配，尤其是当表变化较大时。

二叉查找树的优点:

二叉排序树是一种比较有用的折中方案.

数组的搜索比较方便,可以直接用下标,但删除或者插入某些元素就比较麻烦.链表与之相反,删除和插入元素很快,但查找很慢。

二叉排序树就既有链表的好处,也有数组的好处。在处理大批量的动态的数据是比较有用.时间性能远优于顺序表

二叉查找树的缺点:

顺序存储可能会浪费空间(在非完全二叉树的时候)，但是读取某个指定的节点的时候效率比较高O(0)

链式存储相对二叉树比较大的时候浪费空间较少，但是读取某个指定节点的时候效率偏低O(nlogn)

在空间不受限制时，且不需要高频率的排序操作时，二叉查找树不如散列表。反之二叉查找树优于散列表。