



# 《数据结构》抽象数据结构

题 目 二叉排序树

学 院 计算机学院

专 业 软件工程

班 别 15级3班

学 号 3115005311

姓 名 邓智成

2016年12月10日

**1. 题目**

采用链表存储结构，实现抽象数据类型BinarySortTree。

ADT BinarySortTree{

**数据对象D：**D是具有相同特性的数据元素的集合。各个数据元素均含有类型相同，

可唯一标识数据元素的关键字。

**数据关系R**：数据元素同属一个集合

**基本操作P:**

InitBST(&BST);

操作结果：构造一个空二叉排序树。

DestroyBST(&BST);

初始条件：二叉排序树BST存在。

操作结果：销毁二叉排序树BST。

SearchBST(BST, key);

初始条件：二叉排序树BST存在,key为何关键字类型相同的给定值。

操作结果：若BST中存在其关键字等于key的数据元素，则函数值为元素的值或在表中的位置。否则为”空”。

InsertBST(&BST, e);

初始条件：二叉排序树BST存在。e为待插入的数据元素/

操作结果：若BST中不存在其关键字等于e.key的数据元素，则插入e到BST。

DeleteBST(&BST, key);

初始条件：二叉排序树BST存在,key为何关键字类型相同的给定值。

操作结果：若BST中存在其关键字等于key的数据元素，删除之。

PreOrderTraverseBST(BST, Visit());

初始条件：二叉排序树BST存在,Visti是对结点操作的应用函数。

操作结果：先序对每个结点调用函数Vist()一次且至多一次，一旦Visit()失败，则操作失败

} ADT BinarySortTree

**2．存储结构定义**

公用头文件BST.h:

#include <conio.h>

#include <stdio.h>

#include <stdlib.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define IBFEASIBLE -1

#define OVERFLOW -2

typedef int Status;

typedef struct node {

int key;

struct node \*lChild, \*rChild;

}Node, \*BST;

Status InitBST(BST &t);

Status DestroyBST(BST &t);

Node\* SearchBST(BST t, int key);

Status InsertBST(BST &t, Node\* newNode);

Status DeleteBST(BST t, int key);

void PreOrderTraverseBST(BST t, void(\*Visit) (Node\* node));

void InOrderTraverseBST(BST t, void(\*Visit) (Node\* node));

void PostOrderTraverseBST(BST t, void(\*Visit) (Node\* node));

**3. 算法设计**

//初始化节点

Status InitBST(BST &t) {

if (NULL == t) {

t = (Node\*)malloc(sizeof(Node));

if (NULL == t) {

return OVERFLOW; //内存不足

}

t->lChild = t->rChild = NULL;

}

else

return ERROR; //该节点已经存在

return OK;

}

//销毁二叉排序树

Status DestroyBST(BST &t) {

if (NULL == t) {

return OK;

}

else {

DestroyBST(t->lChild);

DestroyBST(t->rChild);

free(t);

t = NULL;

}

return OK;

}

//搜索key值的结点

Node\* SearchBST(BST t, int key) {

Node \*left;

Node \*right;

if (t != NULL && t->key == key) {

return t;

}

else if (t != NULL) {

left = SearchBST(t->lChild, key); //搜索左子树

if (left == NULL) {

right = SearchBST(t->rChild, key); //搜索右子树

if (right != NULL)

return right;

else

return NULL;

} else {

return left;

}

}

else {

return NULL;

}

}

//插入新结点

Status InsertBST(Node\* &node, Node\* newNode) {

if (newNode->key == NULL) {

return ERROR;

} //newNode不能为空

if (NULL == node) {

node = newNode;

return TRUE;

} //在新的节点插入目标节点

if (newNode->key == node->key) {

return FALSE;

} //如果节点key值在二叉树存在，则报错

if (newNode->key < node->key) {

return InsertBST(node->lChild, newNode);

} //插入到左子树

return InsertBST(node->rChild, newNode);

//插入到右子树

}

//执行删除节点

bool DeleteNode(Node\* &node, Node\* &father) {

//如果被删除节点没有左子树和右子树，直接删除

if (!node->lChild && !node->rChild) {

if (node == father->lChild) {

father->lChild = NULL;

}

else {

father->rChild = NULL;

}

free(node);

}

//如果有左子树，无右子树，把左子树接到父节点上

else if (node->lChild && !node->rChild) {

if (node == father->lChild) {

father->lChild = node->lChild;

}

else {

father->rChild = node->lChild;

}

free(node);

}

//如果有右子树，无左子树，把右子树接到父节点上

else if (node->rChild && !node->lChild) {

if (node == father->lChild) {

father->lChild = node->rChild;

}

else {

father->rChild = node->rChild;

}

free(node);

}

//如果有左子树又有右子树

else {

//找出小于被删除结点中最大的节点

Node\* maxNodeFather = node;

Node\* maxNode = node->lChild;

while (maxNode->rChild) {

maxNodeFather = maxNode;

maxNode = maxNode->rChild;

}

//将被删除节点的key值换成上述得出的结点

node->key = maxNode->key;

//将上述节点的左子树接到它的父节点上

if (maxNodeFather != node) {

maxNodeFather->rChild = maxNode->lChild;

}

else {

maxNodeFather->lChild = maxNode->lChild;

}

//释放空间

free(maxNode);

}

return true;

}

//删除key值得节点

Status DeleteBST(Node\* node, int key) {

Node\* father;

//找到key值对应的节点和父节点

while (node) {

if (node->key == key) {

break;

}

father = node;

node = (key < node->key)?node->lChild : node->rChild;

}

if (!node) {

return ERROR; //找不到目标key值得节点，返回ERROR

}

DeleteNode(node, father);

return OK;

}

//浏览结点的函数

void Visit(Node\* node) {

printf("%d ", node->key);

}

//先序遍历

void PreOrderTraverseBST(BST T, void(\*Visit) (Node\* node)) {

if (T != NULL) {

Visit(T);

PreOrderTraverseBST(T->lChild, Visit);

PreOrderTraverseBST(T->rChild, Visit);

}

}

//中序遍历

void InOrderTraverseBST(BST T, void(\*Visit) (Node\* node)) {

if (T != NULL) {

InOrderTraverseBST(T->lChild, Visit);

Visit(T);

InOrderTraverseBST(T->rChild, Visit);

}

}

//后序遍历

void PostOrderTraverseBST(BST T, void(\*Visit) (Node\* node)) {

if (T != NULL) {

PostOrderTraverseBST(T->lChild, Visit);

PostOrderTraverseBST(T->rChild, Visit);

Visit(T);

}

}

**4．测试**

//打印数组中的数字

void PrintNum(int\* num, int n) {

int i = 0;

for (; i < n; i++) {

printf("%d ", num[i]);

}

printf("\n");

}

int main() {

int num[1000] = {15, 55, 6, 7, 25, 2, 1, 66, 94, 64};

int n = 10;

BST T = NULL;

InitBST(T); //初始化根节点

T->key = num[0];

int k = 1;

for (; k < n; k++) {

Node\* node = NULL;

InitBST(node);

node->key = num[k];

InsertBST(T, node);

} //初始化二叉排序树

while (1) {

printf(“请添加key");

PrintNum(num, n);

printf("请选择\n");

printf("1.添加节点并重新构造二叉树\n2.删除节点\n3.先序遍历\n4.中序遍历\n5.后序遍历\n6.搜索节点\n");

int select;

scanf("%d", &select);

switch (select) {

case 1: {

printf("请添加key:");

int addNum;

scanf("%d", &addNum);

int i;

int flag = 0;

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

if (num[i] == addNum) {

printf("在数组中提存在该数字\n");

flag = -1;

break;

}

}

if (flag == -1) {

continue;

}

num[n] = addNum;

n++;

DestroyBST(T); //销毁原二叉排序树

InitBST(T); //初始化根节点

T->key = num[0];

i = 1;

for (; i < n; i++) { //初始化二叉排序树

Node\* node = NULL;

InitBST(node);

node->key = num[i];

InsertBST(T, node);

}

break;

}

case 2: {

printf("请输入key:");

int delNum;

scanf("%d", &delNum);

int i;

int index;

int flag = 0;

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

if (num[i] == delNum) {

flag = 1;

index = i;

break;

}

}

if (flag == 0) {

printf("数组中没有这个key\n");

continue;

}

DeleteBST(T, delNum); //删除节点

for (i = index; i < n - 1; i++) {

num[i] = num[i + 1];

}

n--;

break;

}

case 3:

PreOrderTraverseBST(T, Visit);

printf("\n");

break;

case 4:

InOrderTraverseBST(T, Visit);

printf("\n");

break;

case 5:

PostOrderTraverseBST(T, Visit);

printf("\n");

break;

case 6:

int searchKey;

printf("请输入要搜索节点的key:");

scanf("%d", &searchKey);

Node\* node = SearchBST(T, searchKey); //搜索节点

if (node != NULL) {

printf("搜索成功，节点key值 %d\n", node->key);

}

default:

break;

fflush(stdin);

}

}

**5．时间复杂度**

Status InitBST(BST &t)——*O*(1)

Status DestroyBST(BST &t) ——*O*(n)

Node\* SearchBST(BST t, int key) ——*O*(n)

Status InsertBST(BST &t, Node\* newNode) ——*O*(n)

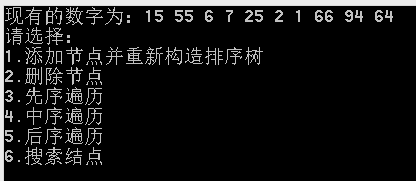
Status DeleteBST(BST t, int key) ——*O*(n)

void PreOrderTraverseBST(BST t, void(\*Visit) (Node\* node)) ——*O*(n)

void InOrderTraverseBST(BST t, void(\*Visit) (Node\* node)) ——*O*(n)

void PostOrderTraverseBST(BST t, void(\*Visit) (Node\* node)) ——*O*(n)

**6. 测试情况**



图示为用10个数字初始化二叉排序树

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\BNJ~4U`OEHJ2TA[{LVFVK3A.png

先序遍历

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\B~CC5NLJ5}5(YBAC8F@B~EB.png

中序遍历

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\4D20($~_)`WQER%@12K%QUK.png

后序遍历

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\Y]4I}J9_5SM7R6D2RT9FL~4.png

图示为往数组里添加数字49，并销毁、再次初始化二叉排序树

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\`E@Y4)QZZ4FL~]AWAT1AF_8.png

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\SSECEA{~R5F@K`HQ578R7AD.png

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\BE[{CVQ~U$68R([YR33F9`B.png

图示分别为添加了49后的先序、中序、后序遍历

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\Z`QB3BHJ)`ZY(YPVS8S~2)B.png

图示为删除数字15后的数组情况

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\LLX)5O%%L1YR%CZS9{252E9.png

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\%))67U{S`@4ANXQ1G7F6RHX.png

C:\Users\bcdzc\AppData\Roaming\Tencent\Users\394894672\QQ\WinTemp\RichOle\2ZU}$SJZ3QK)[{$CTT[V{VJ.png

图为删除15后的先序、中序、后序遍历

**7．思考与小结**

上述二叉排序树是采用链式存储结构来实现，二叉排序树的构造过程实质上就是排序的过程，它是二叉树作媒介，将一个任意的数据序列变成一个有序序列。二叉排序树的构造一般是采用陆续插入结点的办法逐步构成的。具体构造的思路是：  
①以待排序的数据的第一个数据构成根结点；  
②对以后的各个数据，逐个插入结点，而且规定：在插入过程的每一步，原有树结点位置不再变动，只是将新数据的结点作为一个叶子结点插入到合适的位置，使树中任何结点的数据与其左、右子树结点数据之间的关系仍然符合对二叉排序树的要求。

二叉排序树也不难实现，而且在需要对新添加元素进行排序的场合，二叉排序树还是有很大的适用性的。