## BinaryTree.cpp

```
#include<stdio.h>
#include<stdlib.h>

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define OVERFLOW -1
#define SUCCESS 1
#define UNSUCCESS 0

#define dataNum 5
int i = 0;
int dep = 0;
char data[dataNum] = { 'A', 'B', 'C', 'D', 'E' };

typedef int Status;
typedef char TElemType;

typedef struct BiTNode
{
    TElemType data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;

void InitBiTree(BiTree &T);                              //创建一颗空二叉树
BiTree MakeBiTree(TElemType e, BiTree L, BiTree R);      //创建一颗二叉树T，其中根节点的值为e，L 和R
分别作为左子树和右子树
void DestroyBiTree(BiTree &T);                           //销毁二叉树
Status BiTreeEmpty(BiTree T);          //对二叉树判空。若为空返回TRUE，否则FALSE
Status BreakBiTree(BiTree &T, BiTree &L, BiTree &R);     //将一颗二叉树T 分解成根、左子树、右子树三部分
Status ReplaceLeft(BiTree &T, BiTree &LT);    //替换左子树。若T 非空，则用LT 替换T 的左子树，并用LT 返回T
的原有左子树
Status ReplaceRight(BiTree &T, BiTree &RT); //替换右子树。若T 非空，则用RT 替换T 的右子树，并用RT 返回T 的
原有右子树

int Leaves(BiTree T);
int Depth(BiTree T);

Status visit(TElemType e);
void UnionBiTree(BiTree &Ttemp);


//InitBiTree 空二叉树是只有一个BiTree 指针？还是有一个结点但结点域为空？
void InitBiTree(BiTree &T)
{
    T = NULL;
}

BiTree MakeBiTree(TElemType e, BiTree L, BiTree R)
{
    BiTree t;
    t = (BiTree)malloc(sizeof(BiTNode));
    if (NULL == t) return NULL;
    t->data = e;
    t->lchild = L;
    t->rchild = R;
    return t;
}

Status visit(TElemType e)
{
    printf("%c", e);
```

```
    return OK;
}

int Leaves(BiTree T)   //对二叉树 T 求叶子结点数目
{
    int l = 0, r = 0;

    if (NULL == T) return 0;
    if (NULL == T->lchild && NULL == T->rchild) return 1;
    //问题分解，2 个子问题
    //求左子树叶子数目
    l = Leaves(T->lchild);
    //求右子树叶子数目
    r = Leaves(T->rchild);
    //组合
    return r + l;
}

int depTraverse(BiTree T)     //层次遍历：dep 是个全局变量,高度
{
    if (NULL == T) return ERROR;
    dep = (depTraverse(T->lchild) > depTraverse(T->rchild)) ? depTraverse(T->lchild) : depTraverse(T-
>rchild);
    return dep + 1;
}


void levTraverse(BiTree T, Status(*visit)(TElemType e), int lev)
//高度遍历：lev 是局部变量，层次
{
    if (NULL == T) return;

    visit(T->data);
    printf("的层次是%d\n", lev);

    levTraverse(T->lchild, visit, ++lev);
    levTraverse(T->rchild, visit, lev);

}

void InOrderTraverse(BiTree T, Status(*visit)(TElemType e), int &num)
//num 是个全局变量
{
    if (NULL == T) return;
    visit(T->data);
    if (NULL == T->lchild && NULL == T->rchild) { printf("是叶子结点"); num++; }
    else printf("不是叶子结点");
    printf("\n");
    InOrderTraverse(T->lchild, visit, num);
    InOrderTraverse(T->rchild, visit, num);
}
Status BiTreeEmpty(BiTree T)
{
    if (NULL == T) return TRUE;
    return FALSE;
}
Status BreakBiTree(BiTree &T, BiTree &L, BiTree &R)
{
    if (NULL == T) return ERROR;
    L = T->lchild;
    R = T->rchild;
    T->lchild = NULL;
    T->rchild = NULL;
    return OK;
}
Status ReplaceLeft(BiTree &T, BiTree &LT)
```

```
{
    BiTree temp;
    if (NULL == T) return ERROR;
    temp = T->lchild;
    T->lchild = LT;
    LT = temp;
    return OK;
}
Status ReplaceRight(BiTree &T, BiTree &RT)
{
    BiTree temp;
    if (NULL == T) return ERROR;
    temp = T->rchild;
    T->rchild = RT;
    RT = temp;
    return OK;
}




void UnionBiTree(BiTree &Ttemp)
{
    BiTree L = NULL, R = NULL;
    L = MakeBiTree(data[i++], NULL, NULL);
    R = MakeBiTree(data[i++], NULL, NULL);
    ReplaceLeft(Ttemp, L);
    ReplaceRight(Ttemp, R);
}


int main()
{

    BiTree T = NULL, Ttemp = NULL;

    InitBiTree(T);
    if (TRUE == BiTreeEmpty(T)) printf("初始化 T 为空\n");
    else printf("初始化 T 不为空\n");

    T = MakeBiTree(data[i++], NULL, NULL);

    Ttemp = T;
    UnionBiTree(Ttemp);
    Ttemp = T->lchild;
    UnionBiTree(Ttemp);
    Status(*visit1)(TElemType);
    visit1 = visit;
    int num = 0;
    InOrderTraverse(T, visit1, num);
    printf("叶子结点是 %d\n", num);
    printf("叶子结点是 %d\n", Leaves(T));
    int lev = 1;
    levTraverse(T, visit1, lev);
    printf("高度是 %d\n", depTraverse(T));

    return 0;
}
```

## HashTable.cpp

```
#include<stdio.h>
#include<stdlib.h>
#define SUCCESS 1
#define UNSUCCESS 0
#define OVERFLOW -1
#define OK 1
```

```c
#define ERROR -1
typedef int Status;
typedef int KeyType;

typedef struct{
    KeyType key;
}RcdType;
typedef struct{
    RcdType *rcd;
    int size;
    int count;
    int *tag;
}HashTable;

int hashsize[] = { 11, 31, 61, 127, 251, 503 };
int index = 0;

Status InitHashTable(HashTable &H, int size){
    int i;
    H.rcd = (RcdType *)malloc(sizeof(RcdType)*size);
    H.tag = (int *)malloc(sizeof(int)*size);
    if (NULL == H.rcd || NULL == H.tag) return OVERFLOW;
    for (i = 0; i< size; i++)  H.tag[i] = 0;
    H.size = size;
    H.count = 0;
    return OK;
}

int Hash(KeyType key, int m){
    return (3 * key) % m;
}

void collision(int &p, int m){  //线性探测
    p = (p + 1) % m;
}


Status SearchHash(HashTable H, KeyType key, int &p, int &c) {
    p = Hash(key, H.size);
    int h = p;
    c = 0;
    while ((1 == H.tag[p] && H.rcd[p].key != key) || -1 == H.tag[p]){
        collision(p, H.size);  c++;
    }

    if (1 == H.tag[p] && key == H.rcd[p].key) return SUCCESS;
    else return UNSUCCESS;

}

void printHash(HashTable H) //打印哈希表
{
    int  i;
    printf("key : ");
    for (i = 0; i < H.size; i++)
        printf("%3d ", H.rcd[i].key);
    printf("\n");
    printf("tag : ");
    for (i = 0; i < H.size; i++)
        printf("%3d ", H.tag[i]);
    printf("\n\n");
}
Status InsertHash(HashTable &H, KeyType key);  //对函数的声明
//重构
Status recreateHash(HashTable &H){
    RcdType *orcd;
    int *otag, osize, i;
```

```
    orcd = H.rcd;
    otag = H.tag;
    osize = H.size;

    InitHashTable(H, hashsize[index++]);
    //把所有元素，按照新哈希函数放到新表中
    for (i = 0; i < osize; i++){
        if (1 == otag[i]){
            InsertHash(H, orcd[i].key);
        }
    }
}
Status InsertHash(HashTable &H, KeyType key){
    int p, c;
    if (UNSUCCESS == SearchHash(H, key, p, c)){ //没有相同key
        if (c*1.0 / H.size < 0.5){ //冲突次数未达到上线
            //插入代码
            H.rcd[p].key = key;
            H.tag[p] = 1;
            H.count++;
            return SUCCESS;
         }
         else recreateHash(H); //重构哈希表
    }
    return UNSUCCESS;
}
Status DeleteHash(HashTable &H, KeyType key){
    int p, c;
    if (SUCCESS == SearchHash(H, key, p, c)){
        //删除代码
        H.tag[p] = -1;
        H.count--;
        return SUCCESS;
    }
    else return UNSUCCESS;
}
void main()
{
    printf("-----哈希表-----\n");
    HashTable H;
    int i;
    int size = 11;
    KeyType array[8] = { 22, 41, 53, 46, 30, 13, 12, 67 };
    KeyType key;
    RcdType e;

    //初始化哈希表
    printf("初始化哈希表\n");
    if (SUCCESS == InitHashTable(H, hashsize[index++])) printf("初始化成功\n");

    //插入哈希表
    printf("插入哈希表\n");
    for (i = 0; i <= 7; i++){
        key = array[i];
        InsertHash(H, key);
        printHash(H);
    }

    //删除哈希表
    printf("删除哈希表\n");
    int p, c;
    if (SUCCESS == DeleteHash(H, 12)) {
        printf("删除成功，此时哈希表为：\n");
        printHash(H);
    }
```

```
    //查询哈希表
    printf("查询哈希表\n");
    if (SUCCESS == SearchHash(H, 67, p, c)) printf("查询成功\n");
    //再次插入，测试哈希表的重构
    printf("再次插入，测试哈希表的重构：\n");
    KeyType array1[8] = { 27, 47, 57, 47, 37, 17, 93, 67 };
    for (i = 0; i <= 7; i++){
        key = array1[i];
        InsertHash(H, key);
        printHash(H);
    }
}
```

## LinkList.cpp

```
/**
* @author huihut
* @E-mail:huihut@outlook.com
* @version 创建时间：2016 年 9 月 18 日
* 说明：本程序实现了一个单链表。
*/
#include "stdio.h"
#include "stdlib.h"
#include "malloc.h"
//5 个常量定义
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define OVERFLOW -1
//类型定义
typedef int Status;
typedef int ElemType;
//测试程序长度定义
#define LONGTH 5
//链表的类型
typedef struct LNode {
    ElemType data;
    struct LNode *next;
} LNode, *LinkList;

Status InitList_L(LinkList &L);
Status DestroyList_L(LinkList &L);
Status ClearList_L(LinkList &L);
Status ListEmpty_L(LinkList L);
int ListLength_L(LinkList L);
LNode* Search_L(LinkList L, ElemType e);
LNode* NextElem_L(LNode *p);
Status InsertAfter_L(LNode *p, LNode *q);
Status DeleteAfter_L(LNode *p, ElemType &e);
void ListTraverse_L(LinkList L, Status(*visit)(ElemType e));

//创建包含 n 个元素的链表 L，元素值存储在 data 数组中
Status create(LinkList &L, ElemType *data, int n) {
    LNode *p, *q;
    int i;
    if (n < 0) return ERROR;
    L = NULL;
    p = L;
    for (i = 0; i < n; i++)
    {
        q = (LNode *)malloc(sizeof(LNode));
        if (NULL == q) return OVERFLOW;
        q->data = data[i];
        q->next = NULL;
        if (NULL == p) L = q;
        else p->next = q;
```

```
        p = q;
    }
    return OK;
}

//e 从链表末尾入链表
Status EnQueue_LQ(LinkList &L, ElemType &e) {
    LinkList p, q;
    if (NULL == (q = (LNode *)malloc(sizeof(LNode)))) return OVERFLOW;
    q->data = e;
    q->next = NULL;
    if (NULL == L) L = q;
    else
    {
        p = L;
        while (p->next != NULL)
            p = p->next;
        p->next = q;
    }
    return OK;
}
//从链表头节点出链表到 e
Status DeQueue_LQ(LinkList &L, ElemType &e) {
    if (NULL == L) return ERROR;
    LinkList p;
    p = L;
    e = p->data;
    L = L->next;
    free(p);
    return OK;
}
//遍历调用
Status visit(ElemType e) {
    printf("%d\t", e);
}
//遍历单链表
void ListTraverse_L(LinkList L, Status(*visit)(ElemType e))
{
    if (NULL == L) return;
    for (LinkList p = L; NULL != p; p = p -> next) {
        visit(p -> data);
    }
}
int main() {
    int i;
    ElemType e, data[LONGTH] = { 1, 2, 3, 4, 5 };
    LinkList L;

    //显示测试值
    printf("---【单链表】---\n");
    printf("待测试元素为：\n");
    for (i = 0; i < LONGTH; i++) printf("%d\t", data[i]);
    printf("\n");
    //创建链表 L
    printf("创建链表 L\n");
    if (ERROR == create(L, data, LONGTH))
    {
        printf("创建链表 L 失败\n");
        return -1;
    }
    printf("成功创建包含%d 个元素的链表 L\n 元素值存储在 data 数组中\n", LONGTH);

    //遍历单链表
    printf("此时链表中元素为：\n");
    ListTraverse_L(L, visit);

    //从链表头节点出链表到e
```

```
    printf("\n 出链表到 e\n");
    DeQueue_LQ(L, e);
    printf("出链表的元素为：%d\n", e);
    printf("此时链表中元素为：\n");
    //遍历单链表
    ListTraverse_L(L, visit);

    //e 从链表末尾入链表
    printf("\ne 入链表\n");
    EnQueue_LQ(L, e);
    printf("入链表的元素为：%d\n", e);
    printf("此时链表中元素为：\n");
    //遍历单链表
    ListTraverse_L(L, visit);
    printf("\n");

    return 0;
}
```

### LinkList*with*head.cpp

```
/**
* @author huihut
* @E-mail:huihut@outlook.com
* @version 创建时间：2016 年 9 月 23 日
* 说明：本程序实现了一个具有头结点的单链表。
*/

#include "stdio.h"
#include "stdlib.h"
#include "malloc.h"

//5 个常量定义
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define OVERFLOW -1

//类型定义
typedef int Status;
typedef int ElemType;

//测试程序长度定义
#define LONGTH 5

//链表的类型
typedef struct LNode {
    ElemType data;
    struct LNode *next;
} LNode, *LinkList;

Status InitList_L(LinkList &L);
Status DestroyList_L(LinkList &L);
Status ClearList_L(LinkList &L);
Status ListEmpty_L(LinkList L);
int ListLength_L(LinkList L);
LNode* Search_L(LinkList L, ElemType e);
LNode* NextElem_L(LNode *p);
Status InsertAfter_L(LNode *p, LNode *q);
Status DeleteAfter_L(LNode *p, ElemType &e);
void ListTraverse_L(LinkList L, Status(*visit)(ElemType e));
```

```
//创建包含n 个元素的链表L，元素值存储在data 数组中
Status create(LinkList &L, ElemType *data, int n) {
    LNode *p, *q;
    int i;
    if (n < 0) return ERROR;
    p = L = NULL;

    q = (LNode *)malloc(sizeof(LNode));
    if (NULL == q) return OVERFLOW;
    q->next = NULL;
    p = L = q;

    for (i = 0; i < n; i++)
    {
        q = (LNode *)malloc(sizeof(LNode));
        if (NULL == q) return OVERFLOW;
        q->data = data[i];
        q->next = NULL;
        p->next = q;
        p = q;
    }
    return OK;
}
```

```
//e 从链表末尾入链表
Status EnQueue_LQ(LinkList &L, ElemType &e) {
    LinkList p, q;

    if (NULL == (q = (LNode *)malloc(sizeof(LNode)))) return OVERFLOW;
    q->data = e;
    q->next = NULL;
    if (NULL == L)
    {
        L = (LNode *)malloc(sizeof(LNode));
        if (NULL == L) return OVERFLOW;
        L -> next = q;
    }
    else if (NULL == L->next)
    {
        L -> next = q;
    }
    else
    {
        p = L;
        while (p->next != NULL)
        {
            p = p->next;
        }
        p->next = q;
    }
    return OK;
}

//从链表头节点出链表到e
Status DeQueue_LQ(LinkList &L, ElemType &e) {
    if (NULL == L || NULL == L->next) return ERROR;
    LinkList p;
    p = L->next;
    e = p->data;
    L->next = p->next;
    free(p);
```

```c
    return OK;
}
//遍历调用
Status visit(ElemType e) {
    printf("%d\t", e);
    return OK;
}
//遍历单链表
void ListTraverse_L(LinkList L, Status(*visit)(ElemType e))
{
    if (NULL == L || NULL == L->next) return;
    for (LinkList p = L -> next; NULL != p; p = p -> next) {
        visit(p -> data);
    }
}
int main() {
    int i;
    ElemType e, data[LONGTH] = { 1, 2, 3, 4, 5 };
    LinkList L;

    //显示测试值
    printf("---【有头结点的单链表】---\n");
    printf("待测试元素为：\n");
    for (i = 0; i < LONGTH; i++) printf("%d\t", data[i]);
    printf("\n");
    //创建链表L
    printf("创建链表 L\n");
    if (ERROR == create(L, data, LONGTH))
    {
        printf("创建链表 L 失败\n");
        return -1;
    }
    printf("成功创建包含 1 个头结点、%d 个元素的链表 L\n 元素值存 data 数组中\n", LONGTH);
    //遍历单链表
    printf("此时链表中元素为：\n");
    ListTraverse_L(L, visit);
    //从链表头节点出链表到e
    printf("\n 出链表到 e\n");
    DeQueue_LQ(L, e);
    printf("出链表的元素为：%d\n", e);
    printf("此时链表中元素为：\n");
    //遍历单链表
    ListTraverse_L(L, visit);
    //e 从链表末尾入链表
    printf("\ne 入链表\n");
    EnQueue_LQ(L, e);
    printf("入链表的元素为：%d\n", e);
    printf("此时链表中元素为：\n");
    //遍历单链表
    ListTraverse_L(L, visit);
    printf("\n");
    return 0;
}
```

## RedBlackTree.cpp

```cpp
#define BLACK 1
#define RED 0
#include <iostream>

using namespace std;

class bst {
private:

    struct Node {
        int value;
        bool color;
        Node *leftTree, *rightTree, *parent;

      Node() : value(0), color(RED), leftTree(NULL), rightTree(NULL), parent(NULL){}

        Node* grandparent() {
            if(parent == NULL){
                return NULL;
            }
            return parent->parent;
        }

        Node* uncle() {
            if(grandparent() == NULL) {
                return NULL;
            }
            if(parent == grandparent()->rightTree)
                return grandparent()->leftTree;
            else
                return grandparent()->rightTree;
        }

        Node* sibling() {
            if(parent->leftTree == this)
                return parent->rightTree;
            else
                return parent->leftTree;
        }
    };

    void rotate_right(Node *p){
        Node *gp = p->grandparent();
        Node *fa = p->parent;
        Node *y = p->rightTree;

        fa->leftTree = y;

        if(y != NIL)
            y->parent = fa;
        p->rightTree = fa;
        fa->parent = p;

        if(root == fa)
            root = p;
        p->parent = gp;

        if(gp != NULL){
            if(gp->leftTree == fa)
                gp->leftTree = p;
            else
                gp->rightTree = p;
        }

    }
```

```
void rotate_left(Node *p){
    if(p->parent == NULL){
        root = p;
        return;
    }
    Node *gp = p->grandparent();
    Node *fa = p->parent;
    Node *y = p->leftTree;

    fa->rightTree = y;

    if(y != NIL)
        y->parent = fa;
    p->leftTree = fa;
    fa->parent = p;

    if(root == fa)
        root = p;
    p->parent = gp;

    if(gp != NULL){
        if(gp->leftTree == fa)
            gp->leftTree = p;
        else
            gp->rightTree = p;
    }
}
void inorder(Node *p){
    if(p == NIL)
        return;

    if(p->leftTree)
        inorder(p->leftTree);

    cout << p->value << " ";

    if(p->rightTree)
        inorder(p->rightTree);
}

string outputColor (bool color) {
    return color ? "BLACK" : "RED";
}

Node* getSmallestChild(Node *p){
    if(p->leftTree == NIL)
        return p;
    return getSmallestChild(p->leftTree);
}

bool delete_child(Node *p, int data){
    if(p->value > data){
        if(p->leftTree == NIL){
            return false;
        }
        return delete_child(p->leftTree, data);
    } else if(p->value < data){
        if(p->rightTree == NIL){
            return false;
        }
        return delete_child(p->rightTree, data);
    } else if(p->value == data){
        if(p->rightTree == NIL){
            delete_one_child (p);
            return true;
        }
```

```
            Node *smallest = getSmallestChild(p->rightTree);
            swap(p->value, smallest->value);
            delete_one_child (smallest);

            return true;
        }else{
           return false;
         }
    }
    void delete_one_child(Node *p){
        Node *child = p->leftTree == NIL ? p->rightTree : p->leftTree;
        if(p->parent == NULL && p->leftTree == NIL && p->rightTree == NIL){
            p = NULL;
            root = p;
            return;
        }
        if(p->parent == NULL){
            delete  p;
            child->parent = NULL;
            root = child;
            root->color = BLACK;
            return;
        }

        if(p->parent->leftTree == p){
            p->parent->leftTree = child;
        } else {
            p->parent->rightTree = child;
        }
        child->parent = p->parent;

        if(p->color == BLACK){
            if(child->color == RED){
                child->color = BLACK;
            } else
                delete_case (child);
        }

        delete p;
    }
    void delete_case(Node *p){
        if(p->parent == NULL){
            p->color = BLACK;
            return;
        }
        if(p->sibling()->color == RED) {
            p->parent->color = RED;
            p->sibling()->color = BLACK;
            if(p == p->parent->leftTree)
                rotate_left(p->sibling());
            else
                rotate_right(p->sibling());
        }
        if(p->parent->color == BLACK && p->sibling()->color == BLACK
                && p->sibling()->leftTree->color == BLACK && p->sibling()->rightTree->color == BLACK)
{
            p->sibling()->color = RED;
            delete_case(p->parent);
        } else if(p->parent->color == RED && p->sibling()->color == BLACK
                && p->sibling()->leftTree->color == BLACK && p->sibling()->rightTree->color == BLACK)
{
            p->sibling()->color = RED;
            p->parent->color = BLACK;
        } else {
            if(p->sibling()->color == BLACK) {
                if(p == p->parent->leftTree && p->sibling()->leftTree->color == RED
                        && p->sibling()->rightTree->color == BLACK) {
                    p->sibling()->color = RED;
```

```
                p->sibling()->leftTree->color = BLACK;
                rotate_right(p->sibling()->leftTree);
            } else if(p == p->parent->rightTree && p->sibling()->leftTree->color == BLACK
                    && p->sibling()->rightTree->color == RED) {
                p->sibling()->color = RED;
                p->sibling()->rightTree->color = BLACK;
                rotate_left(p->sibling()->rightTree);
            }
        }
        p->sibling()->color = p->parent->color;
        p->parent->color = BLACK;
        if(p == p->parent->leftTree){
            p->sibling()->rightTree->color = BLACK;
            rotate_left(p->sibling());
        } else {
            p->sibling()->leftTree->color = BLACK;
            rotate_right(p->sibling());
        }
    }
}
void insert(Node *p, int data){
    if(p->value >= data){
        if(p->leftTree != NIL)
            insert(p->leftTree, data);
        else {
            Node *tmp = new Node();
            tmp->value = data;
            tmp->leftTree = tmp->rightTree = NIL;
            tmp->parent = p;
            p->leftTree = tmp;
            insert_case (tmp);
        }
    } else {
        if(p->rightTree != NIL)
            insert(p->rightTree, data);
        else {
            Node *tmp = new Node();
            tmp->value = data;
            tmp->leftTree = tmp->rightTree = NIL;
            tmp->parent = p;
            p->rightTree = tmp;
            insert_case (tmp);
        }
    }
}

void insert_case(Node *p){
    if(p->parent == NULL){
        root = p;
        p->color = BLACK;
        return;
    }
    if(p->parent->color == RED){
        if(p->uncle()->color == RED) {
            p->parent->color = p->uncle()->color = BLACK;
            p->grandparent()->color = RED;
            insert_case(p->grandparent());
        } else {
            if(p->parent->rightTree == p && p->grandparent()->leftTree == p->parent) {
                rotate_left (p);
                rotate_right (p);
                p->color = BLACK;
                p->leftTree->color = p->rightTree->color = RED;
            } else if(p->parent->leftTree == p && p->grandparent()->rightTree == p->parent) {
                rotate_right (p);
                rotate_left (p);
                p->color = BLACK;
                p->leftTree->color = p->rightTree->color = RED;
            } else if(p->parent->leftTree == p && p->grandparent()->leftTree == p->parent) {
```

```
                p->parent->color = BLACK;
                p->grandparent()->color = RED;
                rotate_right(p->parent);
            } else if(p->parent->rightTree == p && p->grandparent()->rightTree == p->parent) {
                p->parent->color = BLACK;
                p->grandparent()->color = RED;
                rotate_left(p->parent);
            }
        }
    }
}

void DeleteTree(Node *p){
    if(!p || p == NIL){
        return;
    }
    DeleteTree(p->leftTree);
    DeleteTree(p->rightTree);
    delete p;
}
public:

    bst() {
        NIL = new Node();
        NIL->color = BLACK;
        root = NULL;
    }

    ~bst() {
        if (root)
            DeleteTree (root);
        delete NIL;
    }

    void inorder() {
        if(root == NULL)
            return;
        inorder (root);
        cout << endl;
    }

    void insert (int x) {
        if(root == NULL){
            root = new Node();
            root->color = BLACK;
            root->leftTree = root->rightTree = NIL;
            root->value = x;
        } else {
            insert(root, x);
        }
    }

    bool delete_value (int data) {
        return delete_child(root, data);
    }
private:
    Node *root, *NIL;
};
```

## SqList.cpp

```
/**
* @author huihut
* @E-mail:huihut@outlook.com
* @version 创建时间：2016年9月9日
* 说明：本程序实现了一个顺序表。
```

```c
*/

#include "stdio.h"
#include "stdlib.h"
#include "malloc.h"

//5 个常量定义
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define OVERFLOW -1

//测试程序长度定义
#define LONGTH 5

//类型定义
typedef int Status;
typedef int ElemType;

//顺序栈的类型
typedef struct {
    ElemType *elem;
    int length;
    int size;
    int increment;
} SqList;

Status InitList_Sq(SqList &L, int size, int inc);    //初始化顺序表 L
Status DestroyList_Sq(SqList &L);                     //销毁顺序表 L
Status ClearList_Sq(SqList &L);                        //将顺序表 L 清空
Status ListEmpty_Sq(SqList L);                        //若顺序表 L 为空表，则返回 TRUE，否则 FALSE
int ListLength_Sq(SqList L);                         //返回顺序表 L 中元素个数
Status GetElem_Sq(SqList L, int i, ElemType &e);     //用 e 返回顺序表 L 中第 i 个元素的值
int Search_Sq(SqList L, ElemType e);                 //在顺序表 L 顺序查找元素 e，成功时返回该元素在表中第一次出现的位置，否则返回-1
Status ListTraverse_Sq(SqList L, Status(*visit)(ElemType e));    //遍历顺序表 L，依次对每个元素调用函数 visit()
Status PutElem_Sq(SqList &L, int i, ElemType e);     //将顺序表 L 中第 i 个元素赋值为 e
Status Append_Sq(SqList &L, ElemType e);             //在顺序表 L 表尾添加元素 e
Status DeleteLast_Sq(SqList &L, ElemType &e);         //删除顺序表 L 的表尾元素，并用参数 e 返回其值

//初始化顺序表 L
Status InitList_Sq(SqList &L, int size, int inc) {
    L.elem = (ElemType *)malloc(size * sizeof(ElemType));
    if (NULL == L.elem) return OVERFLOW;
    L.length = 0;
    L.size = size;
    L.increment = inc;
    return OK;
}

//销毁顺序表 L
Status DestroyList_Sq(SqList &L) {
    free(L.elem);
    L.elem = NULL;
    return OK;
}

//将顺序表 L 清空
Status ClearList_Sq(SqList &L) {
    if (0 != L.length) L.length = 0;
    return OK;
}
```

```c
//若顺序表 L 为空表，则返回 TRUE，否则FALSE
Status ListEmpty_Sq(SqList L) {
    if (0 == L.length) return TRUE;
    return FALSE;
}

//返回顺序表 L 中元素个数
int ListLength_Sq(SqList L) {
    return L.length;
}

// 用 e 返回顺序表 L 中第 i 个元素的值
Status GetElem_Sq(SqList L, int i, ElemType &e) {
    e = L.elem[--i];
    return OK;
}


// 在顺序表 L 顺序查找元素 e，成功时返回该元素在表中第一次出现的位置，否则返回 - 1
int Search_Sq(SqList L, ElemType e) {
    int i = 0;
    while (i < L.length && L.elem[i] != e) i++;
    if (i < L.length) return i;
    else return -1;
}

//遍历调用
Status visit(ElemType e) {
    printf("%d\t",e);
}

//遍历顺序表 L，依次对每个元素调用函数 visit()
Status ListTraverse_Sq(SqList L, Status(*visit)(ElemType e)) {
    if (0 == L.length) return ERROR;
    for (int i = 0; i < L.length; i++) {
        visit(L.elem[i]);
    }
    return OK;
}

//将顺序表 L 中第 i 个元素赋值为 e
Status PutElem_Sq(SqList &L, int i, ElemType e) {
    if (i > L.length) return ERROR;
    e = L.elem[--i];
    return OK;

}

//在顺序表 L 表尾添加元素 e
Status Append_Sq(SqList &L, ElemType e) {
    if (L.length >= L.size) return ERROR;
    L.elem[L.length] = e;
    L.length++;
    return OK;
}

//删除顺序表 L 的表尾元素，并用参数 e 返回其值
Status DeleteLast_Sq(SqList &L, ElemType &e) {
    if (0 == L.length) return ERROR;
    e = L.elem[L.length - 1];
    L.length--;
    return OK;
}

int main() {
    //定义表 L
```

```
    SqList L;

    //定义测量值
    int size, increment, i;

    //初始化测试值
    size = LONGTH;
    increment = LONGTH;
    ElemType e, eArray[LONGTH] = { 1, 2, 3, 4, 5 };

    //显示测试值
    printf("---【顺序栈】---\n");
    printf("表 L 的 size 为：%d\n 表 L 的 increment 为：%d\n", size, increment);
    printf("待测试元素为：\n");
    for (i = 0; i < LONGTH; i++) {
        printf("%d\t", eArray[i]);
    }
    printf("\n");

    //初始化顺序表
    if (!InitList_Sq(L, size, increment)) {
        printf("初始化顺序表失败\n");
        exit(0);
    }
    printf("已初始化顺序表\n");

    //判空
    if(TRUE == ListEmpty_Sq(L)) printf("此表为空表\n");
    else printf("此表不是空表\n");
    //入表
    printf("将待测元素入表：\n");
    for (i = 0; i < LONGTH; i++) {
        if(ERROR == Append_Sq(L, eArray[i])) printf("入表失败\n");;
    }
    printf("入表成功\n");
    //遍历顺序表L
    printf("此时表内元素为：\n");
    ListTraverse_Sq(L, visit);
    //出表
    printf("\n 将表尾元素入表到 e：\n");
    if (ERROR == DeleteLast_Sq(L, e)) printf("出表失败\n");
    printf("出表成功\n 出表元素为%d\n",e);
    //遍历顺序表L
    printf("此时表内元素为：\n");
    ListTraverse_Sq(L, visit);
    //销毁顺序表
    printf("\n 销毁顺序表\n");
    if(OK == DestroyList_Sq(L)) printf("销毁成功\n");
    else printf("销毁失败\n");

    return 0;
}
```

### SqStack.cpp

```
/**
* @author huihut
* @E-mail:huihut@outlook.com
* @version 创建时间：2016 年 9 月 9 日
* 说明：本程序实现了一个顺序栈。
* 功能：有初始化、销毁、判断空、清空、入栈、出栈、取元素的操作。
*/

#include "stdio.h"
```

```
#include "stdlib.h"
#include "malloc.h"

//5 个常量定义
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define OVERFLOW -1
//测试程序长度定义
#define LONGTH 5
//类型定义
typedef int Status;
typedef int ElemType;

//顺序栈的类型
typedef struct {
    ElemType *elem;
    int top;
    int size;
    int increment;
} SqSrack;
//函数声明
Status InitStack_Sq(SqSrack &S, int size, int inc);  //初始化顺序栈
Status DestroyStack_Sq(SqSrack &S);                   //销毁顺序栈
Status StackEmpty_Sq(SqSrack S);                    //判断 S 是否空，若空则返回 TRUE，否则返回 FALSE
void ClearStack_Sq(SqSrack &S);                       //清空栈 S
Status Push_Sq(SqSrack &S, ElemType e);               //元素 e 压入栈 S
Status Pop_Sq(SqSrack &S, ElemType &e);               //栈 S 的栈顶元素出栈，并用 e 返回
Status GetTop_Sq(SqSrack S, ElemType &e);           //取栈 S 的栈顶元素，并用 e 返回

//初始化顺序栈
Status InitStack_Sq(SqSrack &S, int size, int inc) {
    S.elem = (ElemType *)malloc(size * sizeof(ElemType));
    if (NULL == S.elem) return OVERFLOW;
    S.top = 0;
    S.size = size;
    S.increment = inc;
    return OK;
}
//销毁顺序栈
Status DestroyStack_Sq(SqSrack &S) {
    free(S.elem);
    S.elem = NULL;
    return OK;
}
//判断 S 是否空，若空则返回 TRUE，否则返回 FALSE
Status StackEmpty_Sq(SqSrack S) {
    if (0 == S.top) return TRUE;
    return FALSE;
}
//清空栈 S
void ClearStack_Sq(SqSrack &S) {
    if (0 == S.top) return;
    S.size = 0;
    S.top = 0;
}
//元素 e 压入栈 S
Status Push_Sq(SqSrack &S, ElemType e) {
    ElemType *newbase;
    if (S.top >= S.size) {
        newbase = (ElemType *)realloc(S.elem, (S.size + S.increment) * sizeof(ElemType));
        if (NULL == newbase) return OVERFLOW;
        S.elem = newbase;
        S.size += S.increment;
    }
```

```c
    S.elem[S.top++] = e;
    return OK;
}
//取栈 S 的栈顶元素，并用 e 返回
Status GetTop_Sq(SqSrack S, ElemType &e) {
    if (0 == S.top) return ERROR;
    e = S.elem[S.top - 1];
    return e;
}
//栈 S 的栈顶元素出栈，并用 e 返回
Status Pop_Sq(SqSrack &S, ElemType &e) {
    if (0 == S.top) return ERROR;
    e = S.elem[S.top - 1];
    S.top--;
    return e;
}
int main() {
    //定义栈S
    SqSrack S;
    //定义测量值
    int size, increment, i;
    //初始化测试值
    size = LONGTH;
    increment = LONGTH;
    ElemType e, eArray[LONGTH] = { 1, 2, 3, 4, 5 };
    //显示测试值
    printf("---【顺序栈】---\n");
    printf("栈 S 的 size 为：%d\n 栈 S 的 increment 为：%d\n", size, increment);
    printf("待测试元素为：\n");
    for (i = 0; i < LONGTH; i++) {
        printf("%d\t", eArray[i]);
    }
    printf("\n");
    //初始化顺序栈
    if (!InitStack_Sq(S, size, increment)) {
        printf("初始化顺序栈失败\n");
        exit(0);
    }
    printf("已初始化顺序栈\n");
    //入栈
    for (i = 0; i < S.size; i++) {
        if (!Push_Sq(S, eArray[i])) {
            printf("%d 入栈失败\n", eArray[i]);
            exit(0);
        }
    }
    printf("已入栈\n");
    //判断非空
    if(StackEmpty_Sq(S)) printf("S 栈为空\n");
    else printf("S 栈非空\n");
    //取栈 S 的栈顶元素
    printf("栈 S 的栈顶元素为：\n");
    printf("%d\n", GetTop_Sq(S, e));
    //栈 S 元素出栈
    printf("栈 S 元素出栈为：\n");
    for (i = 0, e = 0; i < S.size; i++) {
        printf("%d\t", Pop_Sq(S, e));
    }
    printf("\n");
    //清空栈S
    ClearStack_Sq(S);
    printf("已清空栈 S\n");
    return 0;
}
```