```c
/* RBTree.c */
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
#include <string.h>
#include "RBTree.h"
#include "LinkStack.h"

/* x表示值，p表示指针 */
#define IsRoot(x)         (!((x).parent))
#define IsLChild(x)       (!IsRoot(x) && (&(x) == (x).parent->lc))
#define IsRChild(x)       (!IsRoot(x) && (&(x) == (x).parent->rc))
#define HasParent(x)      (!IsRoot(x))
#define HasLChild(x)      ((x).lc)
#define HasRChild(x)      ((x).rc)
#define HasChild(x)       (HasLChild(x) || HasRChild(x))
#define HasBothChild(x)   (HasLChild(x) && HasRChild(x))
#define IsLeaf(x)         (!HasChild(x))
//获取x的兄弟节点
#define Sibling(x)        (IsLChild(x) ? (x).parent->rc : (x).parent->lc)
//获取x的叔叔节点
#define Uncle(x)          (IsLChild(*((x).parent)) ? (x).parent->parent->rc : (x).parent->parent->lc)
//节点高度，空树高度为-1
#define stature(p)        ((p) ? (p)->height : -1)
//外部节点也视作黑节点
#define isBlack(p)        (!(p) || (RB_BLACK == (p)->color))
#define isRed(p)          (!isBlack(p))
//红黑树高度更新条件
#define BlackHeightUpdated(x) ( \
    (stature((x).lc) == stature((x).rc)) && \
    ((x).height == (isRed(&x) ? stature((x).lc) : stature((x).lc) + 1)) \
)

static RBTREENODE *nodeNew(int keySize, const void *e)
{
    RBTREENODE *newNode = (RBTREENODE *)malloc(sizeof(RBTREENODE) + keySize);
    if (NULL == newNode)
    {
        return NULL;
    }
    newNode->parent = NULL;
    newNode->lc = NULL;
    newNode->rc = NULL;
    newNode->height = -1;
    newNode->color = RB_RED;
    memcpy(newNode->key, e, keySize);
    return newNode;
}

static void nodeDispose(RBTREENODE *node, RBTreeFree *freeFn)
{
    if (NULL != freeFn)
    {
        freeFn(node->key);
    }
    free(node);
}

//RBTree初始化
void RBTreeNew(RBTREE *rbTree, int keySize, RBTreeCmp *cmpFn, RBTreeFree *freeFn)
{
    assert(keySize > 0);
    assert(NULL != cmpFn);
    rbTree->root = NULL;
    rbTree->hot = NULL;
    rbTree->size = 0;
    rbTree->keySize = keySize;
    rbTree->cmpFn = cmpFn;
    rbTree->freeFn = freeFn;
}

//RBTree判空
```

```c
int RBTreeEmpty(RBTREE *rbTree)
{
    return (rbTree->size == 0);
}

//RBTree规模
int RBTreeSize(RBTREE *rbTree)
{
    return rbTree->size;
}

//RBTree树高度
int RBTreeHeight(RBTREE *rbTree)
{
    return rbTree->root->height;
}

//将当前节点及其左侧分支入栈
static goAlongLeftBranch(RBTREENODE *node, STACK *s)
{
    RBTREENODE *cur = node;
    while (NULL != cur)
    {
        StackPush(s, &cur);
        cur = cur->lc;
    }
}

//RBTree销毁
void RBTreeDispose(RBTREE *rbTree)
{
    if (RBTreeEmpty(rbTree))
    {
        return ;
    }
    STACK brNodeStack;
    //栈中存放avl节点指针
    StackNew(&brNodeStack, sizeof(RBTREENODE *), NULL);
    RBTREENODE *node = rbTree->root, *cur;
    while (1)
    {
        //从当前节点出发，逐批入栈
        goAlongLeftBranch(node, &brNodeStack);
        //所有节点处理完毕
        if (StackEmpty(&brNodeStack))
        {
            break;
        }
        //弹出栈顶节点并访问之
        StackPop(&brNodeStack, &node);
        cur = node;
        node = node->rc; //转向右子树
        nodeDispose(cur, rbTree->freeFn);
    }
    StackDispose(&brNodeStack);
    rbTree->root = NULL;
    rbTree->hot = NULL;
    rbTree->size = 0;
}

//二叉树中序遍历算法（迭代版#1）
static void travIn_V1(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
{
    STACK brNodeStack;
    //栈中存放avl节点指针
    StackNew(&brNodeStack, sizeof(RBTREENODE *), NULL);
    RBTREENODE *node = rbTree->root;
    while (1)
    {
        //从当前节点出发，逐批入栈
        goAlongLeftBranch(node, &brNodeStack);
        //所有节点处理完毕
        if (StackEmpty(&brNodeStack))
```

```
        {
            break;
        }
        //弹出栈顶节点并访问之
        StackPop(&brNodeStack, &node);
        traverseOpFn(node->key);
        node = node->rc; //转向右子树
    }
    StackDispose(&brNodeStack);
}

//二叉树中序遍历算法（迭代版#2，版本#1的等价形式）
static void travIn_V2(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
{
    STACK brNodeStack;
    //栈中存放avl节点指针
    StackNew(&brNodeStack, sizeof(RBTREENODE *), NULL);
    RBTREENODE *node = rbTree->root;
    while (1)
    {
        if (NULL != node)
        {
            StackPush(&brNodeStack, &node);
            node = node->lc;
        }
        else if (!StackEmpty(&brNodeStack))
        {
            StackPop(&brNodeStack, &node);
            traverseOpFn(node->key);
            node = node->rc;
        }
        else
        {
            break;
        }
    }
    StackDispose(&brNodeStack);
}

//定位节点node在中序遍历中的直接后继
static RBTREENODE *succ(RBTREENODE *node)
{
    RBTREENODE *s = node;
    if (NULL != s->rc) //若有右孩子，则直接后继必在右子树中
    {
        s = s->rc;
        while (HasLChild(*s))
        {
            s = s->lc;
        }
    }
    else
    {
        while (IsRChild(*s))
        {
            s = s->parent;
        }
        s = s->parent;
    }
    return s;
}

//二叉树中序遍历算法（迭代版#3）
static void travIn_V3(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
{
    //前一步是否刚从右子树回溯--省去栈，仅O(1)辅助空间
    int backtrack = 0;
    RBTREENODE *node = rbTree->root;
    while (1)
    {
        //若有左子树且不是刚刚回溯，则深入遍历左子树
        if (!backtrack && HasLChild(*node))
        {
```

```c
                node = node->lc;
            } //否则无左子树或刚刚回溯
            else
            {
                traverseOpFn(node->key);
                //右子树非空，深入右子树继续遍历，并关闭回溯标志
                if (HasRChild(*node))
                {
                    node = node->rc;
                    backtrack = 0;
                }
                else //右子树为空则回溯并设置回溯标志
                {
                    node = succ(node);
                    if (NULL == node)
                    {
                        break;
                    }
                    backtrack = 1;
                }
            }
        }
    }

    //RBTree中序遍历（非递归）
    void RBTreeTravIn(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
    {
        if (NULL == traverseOpFn || RBTreeEmpty(rbTree))
        {
            return ;
        }
        travIn_V1(rbTree, traverseOpFn);
    }

    static void travInRecAt(RBTREENODE *node, RBTreeTraverseOp *traverseOpFn)
    {
        if (NULL == node)
        {
            return ;
        }
        travInRecAt(node->lc, traverseOpFn);
        traverseOpFn(node->key);
        travInRecAt(node->rc, traverseOpFn);
    }

    //RBTree中序遍历（递归）
    void RBTreeTravInRec(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
    {
        if (NULL == traverseOpFn || RBTreeEmpty(rbTree))
        {
            return ;
        }
        travInRecAt(rbTree->root, traverseOpFn);
    }

    //RBTree中查找关键码所在节点，hot指向当前节点的父节点
    RBTREENODE *RBTreeSearch(RBTREE *rbTree, const void *e)
    {
        RBTREENODE *node = rbTree->root;
        rbTree->hot = NULL;
        while (NULL != node)
        {
            if (0 == rbTree->cmpFn(e, node->key))
            {
                break ;
            }
            rbTree->hot = node;
            if (0 < rbTree->cmpFn(e, node->key))
            {
                node = node->rc;
            }
            else
            {
```

```
            node = node->lc;
        }
    }
    return node;
}

int max(int a, int b)
{
    return (a > b ? a : b);
}

//更新节点node的黑高度
static int updateHeight(RBTREENODE *node)
{
    //孩子一般黑高度相等，除非出现双黑
    node->height = max(stature(node->lc), stature(node->rc));
    return (isBlack(node) ? (++ node->height) : node->height);
}

//按照"3+4"结构联接3个节点及四颗子树，返回重组后的局部子树根节点的位置（即b）
//子树根节点与上层节点之间的双向联接，均须由上层调用者完成
static RBTREENODE *connect34(RBTREENODE *a, RBTREENODE *b, RBTREENODE *c, RBTREENODE *T0, RBTREENODE *T1, RBTREENODE *T2, RBTREENODE *T3)
{
    a->lc = T0;
    if (T0)
    {
        T0->parent = a;
    }
    a->rc = T1;
    if (T1)
    {
        T1->parent = a;
    }
    updateHeight(a);

    c->lc = T2;
    if (T2)
    {
        T2->parent = c;
    }
    c->rc = T3;
    if (T3)
    {
        T3->parent = c;
    }
    updateHeight(c);

    b->lc = a;
    a->parent = b;
    b->rc = c;
    c->parent = b;
    updateHeight(b);
    return b;
}

//BST节点旋转变换统一算法（3节点+4子树），返回调整后局部子树根节点的位置
//注意：尽管子树根会正确指向上层节点（如果存在），但反向的联接须由上层函数完成
static RBTREENODE *rotateAt(RBTREENODE *grandsonNode)
{
    RBTREENODE *p = grandsonNode->parent;
    RBTREENODE *g = p->parent;
    if (IsLChild(*p)) //zig
    {
        if (IsLChild(*grandsonNode)) //zig-zig
        {
            p->parent = g->parent; //向上联接
            return connect34(grandsonNode, p, g, grandsonNode->lc, grandsonNode->rc, p->rc, g->rc);
        }
        else //zig-zag
        {
            grandsonNode->parent = g->parent; //向上联接
```

```
            return connect34(p, grandsonNode, g, p->lc, grandsonNode->lc,
            grandsonNode->rc, g->rc);
        }
    }
    else //zag
    {
        if (IsLChild(*grandsonNode)) //zag-zig
        {
            grandsonNode->parent = g->parent; //向上联接
            return connect34(g, grandsonNode, p, g->lc, grandsonNode->lc,
            grandsonNode->rc, p->rc);
        }
        else //zag-zag
        {
            p->parent = g->parent; //向上联接
            return connect34(g, p, grandsonNode, g->lc, p->lc, grandsonNode->lc,
            grandsonNode->rc);
        }
    }
}

//红黑树双红调整算法：解决节点node与其父亲均为红色的问题，分为两大类情况：
//RR-1：2次颜色翻转，2次黑高度更新，1~2次旋转，不再递归
//RR-2：3次颜色翻转，3次黑高度更新，0次旋转，需要递归
static void solveDoubleRed(RBTREE *rbTree, RBTREENODE *node) //当前node必为红
{
    //若已（递归）转至树根，则将其转黑，整树黑高度也随之递增
    if (IsRoot(*node))
    {
        rbTree->root->color = RB_BLACK;
        rbTree->root->height ++;
        return ;
    }
    RBTREENODE *p = node->parent;
    if (isBlack(p)) //父节点为黑，则终止调整
    {
        return ;
    }
    //此时p为红，g为黑
    RBTREENODE *g = p->parent;
    RBTREENODE *u = Uncle(*node);
    if (isBlack(u)) //RR-1，叔父节点为黑色
    {
        //node与p同侧（即zig-zig或zag-zag）
        if (IsLChild(*node) == IsLChild(*p))
        {
            p->color = RB_BLACK;
        }
        else //node与p异侧（即zig-zag或zag-zig）
        {
            node->color = RB_BLACK;
        }
        g->color = RB_RED;
        //重新接入原树（向下接入）
        if (IsRoot(*g))
        {
            rbTree->root = rotateAt(node);
        }
        else if (IsLChild(*g))
        {
            RBTREENODE *gg = g->parent;
            gg->lc = rotateAt(node);
        }
        else
        {
            RBTREENODE *gg = g->parent;
            gg->rc = rotateAt(node);
        }
    }
    else //RR-2，叔父节点为红色
    {
        //p由红转黑
        p->color = RB_BLACK;
```

```
        p->height ++;
        //u由红转黑
        u->color = RB_BLACK;
        u->height ++;
        //g由黑转红，高度不变
        g->color = RB_RED;
        solveDoubleRed(rbTree, g);
    }
}

//RBTree中插入关键码
RBTREENODE *RBTreeInsert(RBTREE *rbTree, const void *e)
{
    RBTREENODE *node = RBTreeSearch(rbTree, e);
    if (NULL != node)
    {
        return node;
    }
    //新节点初始化
    node = nodeNew(rbTree->keySize, e);
    if (NULL == node)
    {
        return NULL;
    }
    node->parent = rbTree->hot;
    //空树时指定根
    if (NULL == node->parent)
    {
        rbTree->root = node;
    }
    else
    {
        if (0 < rbTree->cmpFn(e, rbTree->hot->key))
        {
            rbTree->hot->rc = node;
        }
        else
        {
            rbTree->hot->lc = node;
        }
    }
    rbTree->size ++;
    solveDoubleRed(rbTree, node);
    return node;
}

static void swap(void *dataAddr1, void *dataAddr2, int dataSize)
{
    char *tmp = (char *)malloc(dataSize);
    memcpy(tmp, dataAddr1, dataSize);
    memcpy(dataAddr1, dataAddr2, dataSize);
    memcpy(dataAddr2, tmp, dataSize);
    free(tmp);
}

//删除node所指节点
//返回值指向实际被删除节点的接替者，hot指向实际被删除节点的父亲，二者均可能是NULL
static RBTREENODE *removeAt(RBTREENODE **node, RBTREENODE **hot, RBTreeFree *freeFn, int keySize)
{
    RBTREENODE *w = *node; //实际被删除的节点
    RBTREENODE *successor = NULL; //实际被删除节点的接替者

    //若**node的左子树为空，则直接将**node替换为其右子树
    if (!HasLChild(**node))
    {
        *node = (*node)->rc;
        successor = *node;
    }
    else if (!HasRChild(**node)) //右子树为空，对称处理
    {
        *node = (*node)->lc;
        successor = *node;
```

```
    }
    else //左、右子树均存在，则选择**node节点的直接后继作为实际被摘除节点
    {
        w = succ(w); //在右子树中找到直接后继
        swap((*node)->key, w->key, keySize);
        RBTREENODE *u = w->parent;
        successor = w->rc; //此时待删除节点不可能有左孩子
        //隔离被删除节点w
        if (*node == u)
        {
            u->rc = successor;
        }
        else
        {
            u->lc = successor;
        }
    }
    //记录实际被删除节点的父亲
    *hot = w->parent;
    if (successor)
    {
        successor->parent = *hot; //将被删除节点的接替者与hot相联
    }
    nodeDispose(w, freeFn);
    return successor;
}

//红黑树双黑调整算法：解决节点node与被其替代的节点均为黑色的问题
//分为三大类共四种情况
//BB-1 ：2次颜色翻转，2次黑高度更新，1~2次旋转，不再递归
//BB-2R：2次颜色翻转，2次黑高度更新，0次旋转，不再递归
//BB-2B：1次颜色翻转，1次黑高度更新，0次旋转，需要递归
//BB-3 ：2次颜色翻转，2次黑高度更新，1次旋转，转为BB-1或BB-2R
static void solveDoubleBlack(RBTREE *rbTree, RBTREENODE *node)
{
    RBTREENODE *p = node ? node->parent : rbTree->hot;
    if (NULL == p)
    {
        return ;
    }
    //此处node可能为NULL，故不能调用宏Sibling
    RBTREENODE *s = (node == p->lc) ? p->rc : p->lc;
    if (isBlack(s)) //兄弟s为黑
    {
        //s的红孩子（若左、右孩子皆红，左者优先；皆黑时为NULL）
        RBTREENODE *t = NULL;
        if (HasLChild(*s) && isRed(s->lc))
        {
            t = s->lc;
        }
        else if (HasRChild(*s) && isRed(s->rc))
        {
            t = s->rc;
        }
        if (NULL != t) //黑s有红孩子：BB-1
        {
            //备份原子树根p颜色
            RBCOLOR oldColor = p->color;
            RBTREENODE *gg = p->parent;
            RBTREENODE *b;//重平衡后的子树根节点
            if (NULL == gg)
            {
                b = rotateAt(t);
                rbTree->root = b;
            }
            else if (p == gg->lc)
            {
                b = rotateAt(t);
                gg->lc = b;
            }
            else
            {
                b = rotateAt(t);
```

```
                    gg->rc = b;
                }
                if (HasLChild(*b))
                {
                    b->lc->color = RB_BLACK;
                    updateHeight(b->lc);
                }
                if (HasRChild(*b))
                {
                    b->rc->color = RB_BLACK;
                    updateHeight(b->rc);
                }
                b->color = oldColor;
                updateHeight(b);
            }
            else //黑s无红孩子
            {
                s->color = RB_RED;
                s->height --;
                if (isRed(p)) //BB-2R
                {
                    p->color = RB_BLACK; //p转黑，高度不变
                }
                else //BB-2B
                {
                    p->height --; //p保持黑，黑高度下降
                    solveDoubleBlack(rbTree, p);
                }
            }
        }
        else //兄弟s为红：BB-3
        {
            s->color = RB_BLACK;
            p->color = RB_RED;
            RBTREENODE *t = IsLChild(*s) ? s->lc : s->rc; //取t与其父s同侧
            rbTree->hot = p;
            RBTREENODE *gg = p->parent;
            if (NULL == gg)
            {
                rbTree->root = rotateAt(t);
            }
            else if (p == gg->lc)
            {
                gg->lc = rotateAt(t);
            }
            else
            {
                gg->rc = rotateAt(t);
            }
            //继续修正node处双黑--此时的p已转红，故后续只能是BB-1或BB-2R
            solveDoubleBlack(rbTree, node);
        }
    }

    static int rbTreeRemoveBase(RBTREE *rbTree, void *e, RBTreeFree *freeFn)
    {
        RBTREENODE *node = RBTreeSearch(rbTree, e);
        //查不到要删除的节点，删除失败
        if (NULL == node)
        {
            return -1;
        }
        RBTREENODE *hot = NULL; //记录删除节点的父亲
        RBTREENODE *r = removeAt(&node, &hot, freeFn, rbTree->keySize);
        rbTree->hot = hot;
        rbTree->size --;
        if (RBTreeEmpty(rbTree))
        {
            return 0;
        }
        //若被删除的是根节点（树根节点只有单侧孩子），则将当前根节点置黑，并更新其高度
        if (NULL == rbTree->hot)
        {
```

```
            rbTree->root->color = RB_BLACK;
            updateHeight(rbTree->root);
            return 0;
        }
        //assert: 以下，原node（现r）必非根，hot必非空
        //若所有祖先的黑深度依然平衡，则无需调整
        if (BlackHeightUpdated(*(rbTree->hot)))
        {
            return 0;
        }
        if (isRed(r))
        {
            r->color = RB_BLACK;
            r->height ++;
            return 0;
        }
        //assert: 以下，原node（现r）均为黑色
        solveDoubleBlack(rbTree, r);
        return 0;
    }

    //RBTree中删除关键码所在节点，返回值：0成功，!0失败
    int RBTreeRemove(RBTREE *rbTree, void *e)
    {
        return rbTreeRemoveBase(rbTree, e, rbTree->freeFn);
    }

    //RBTree中删除关键码所在节点（无需深度删除关键码），返回值：0成功，!0失败
    int RBTreeRemoveU(RBTREE *rbTree, void *e)
    {
        return rbTreeRemoveBase(rbTree, e, NULL);
    }
```