```c
/* Splay.c */
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
#include <string.h>
#include "Splay.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)

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

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

//Splay初始化
void SplayNew(SPLAYTREE *splay, int keySize, SplayCmp *cmpFn, SplayFree *freeFn)
{
    assert(keySize > 0);
    assert(NULL != cmpFn);
    splay->root = NULL;
    splay->hot = NULL;
    splay->size = 0;
    splay->keySize = keySize;
    splay->cmpFn = cmpFn;
    splay->freeFn = freeFn;
}

//Splay判空
int SplayEmpty(SPLAYTREE *splay)
{
    return (0 == splay->size);
}

//Splay规模
int SplaySize(SPLAYTREE *splay)
{
    return splay->size;
}

//将当前节点及其左侧分支入栈
```

```
static goAlongLeftBranch(SPLAYNODE *node, STACK *s)
{
    SPLAYNODE *cur = node;
    while (NULL != cur)
    {
        StackPush(s, &cur);
        cur = cur->lc;
    }
}

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

//二叉树中序遍历算法（迭代版#1）
static void travIn_V1(SPLAYTREE *splay, SplayTraverseOp *traverseOpFn)
{
    STACK splayNodeStack;
    //栈中存放avl节点指针
    StackNew(&splayNodeStack, sizeof(SPLAYNODE *), NULL);
    SPLAYNODE *node = splay->root;
    while (1)
    {
        //从当前节点出发，逐批入栈
        goAlongLeftBranch(node, &splayNodeStack);
        //所有节点处理完毕
        if (StackEmpty(&splayNodeStack))
        {
            break;
        }
        //弹出栈顶节点并访问之
        StackPop(&splayNodeStack, &node);
        traverseOpFn(node->key);
        node = node->rc; //转向右子树
    }
    StackDispose(&splayNodeStack);
}

//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(SPLAYTREE *splay, SplayTraverseOp *traverseOpFn)
{
    STACK splayNodeStack;
    //栈中存放avl节点指针
    StackNew(&splayNodeStack, sizeof(SPLAYNODE *), NULL);
    SPLAYNODE *node = splay->root;
    while (1)
    {
        if (NULL != node)
        {
            StackPush(&splayNodeStack, &node);
            node = node->lc;
        }
        else if (!StackEmpty(&splayNodeStack))
        {
            StackPop(&splayNodeStack, &node);
            traverseOpFn(node->key);
            node = node->rc; //转向右子树
        }
        else
        {
            break;
        }
    }
    StackDispose(&splayNodeStack);
}

//定位节点node在中序遍历中的直接后继
static SPLAYNODE *succ(SPLAYNODE *node)
{
    SPLAYNODE *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(SPLAYTREE *splay, SplayTraverseOp *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;
    SPLAYNODE *node = splay->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))
        {
            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;
            }
        }
    }
}

//Splay中序遍历（非递归）
void SplayTravIn(SPLAYTREE *splay, SplayTraverseOp *traverseOpFn)
{
    if (NULL == traverseOpFn || SplayEmpty(splay))
    {
        return ;
    }
    travIn_V1(splay, traverseOpFn);
}

static void travInRecAt(SPLAYNODE *node, SplayTraverseOp *traverseOpFn)
{
    if (NULL == node) //递归基
    {
        return ;
    }
    travInRecAt(node->lc, traverseOpFn);
    traverseOpFn(node->key);
    travInRecAt(node->rc, traverseOpFn);
}

//Splay中序遍历（递归）
void SplayTravInRec(SPLAYTREE *splay, SplayTraverseOp *traverseOpFn)
{
    if (NULL == traverseOpFn || SplayEmpty(splay))
    {
        return ;
    }
    travInRecAt(splay->root, traverseOpFn);
}

//在节点p与lc（可能为空）之间建立父（左）子关系
static void attachAsLChild(SPLAYNODE *p, SPLAYNODE *lc)
{
    p->lc = lc;
    if (lc)
    {
        lc->parent = p;
    }
}

//在节点p与rc（可能为空）之间建立父（右）子关系
static void attachAsRChild(SPLAYNODE *p, SPLAYNODE *rc)
{
    p->rc = rc;
    if (rc)
    {
        rc->parent = p;
    }
}

//Splay树伸展算法，从节点node出发，自下而上做伸展
//调整之后新树根应为被伸展的节点，故返回该节点的位置以便上层函数更新使用
static SPLAYNODE *sSplay(SPLAYNODE *node)
{
    if (NULL == node)
    {
        return NULL;
    }
    SPLAYNODE *p, *g;
    //自下而上，反复对node做双层伸展
    while ((p = node->parent) && (g = p ->parent))
    {
        //每轮之后，node都以原曾祖父为父
        SPLAYNODE *gg = g->parent;
        if (IsLChild(*node)) //zig
        {
```

```c
            if (IsLChild(*p)) //zig-zig
            {
                attachAsLChild(g, p->rc);
                attachAsLChild(p, node->rc);
                attachAsRChild(p, g);
                attachAsRChild(node, p);
            }
            else //zig-zag
            {
                attachAsRChild(g, node->lc);
                attachAsLChild(p, node->rc);
                attachAsLChild(node, g);
                attachAsRChild(node, p);
            }
        }
        else //zag
        {
            if (IsLChild(*p)) //zag-zig
            {
                attachAsRChild(p, node->lc);
                attachAsLChild(g, node->rc);
                attachAsLChild(node, p);
                attachAsRChild(node, g);
            }
            else //zag-zag
            {
                attachAsRChild(g, p->lc);
                attachAsLChild(p, g);
                attachAsRChild(p, node->lc);
                attachAsLChild(node, p);
            }
        }
        if (NULL == gg)
        {
            node->parent = NULL;
        }
        else
        {
            (g == gg->lc) ? attachAsLChild(gg, node) : attachAsRChild(gg, node);
        }
    } //双层伸展结束时，必有g == NULL，但p可能非空
    p = node->parent;
    if (NULL != p)
    {
        if (IsLChild(*node)) //zig
        {
            attachAsLChild(p, node->rc);
            attachAsRChild(node, p);
        }
        else //zag
        {
            attachAsRChild(p, node->lc);
            attachAsLChild(node, p);
        }
    }
    node->parent = NULL;
    return node;
}

//Splay中查找关键码所在节点，无论查找成功与否，伸展树的根都指向最后被访问的节点
SPLAYNODE *SplaySearch(SPLAYTREE *splay, const void *e)
{
    if (NULL == splay->root)
    {
        return NULL;
    }
    SPLAYNODE *node = splay->root;
    //hot指向当前节点的父节点
    splay->hot = NULL;
    while (NULL != node)
    {
        if (0 == splay->cmpFn(e, node->key))
        {
            break ;
        }
        else if (0 < splay->cmpFn(e, node->key))
        {
            splay->hot = node;
            node = node->rc;
        }
        else
        {
            splay->hot = node;
            node = node->lc;
        }
    }
    node = node ? node : splay->hot;
    //将最后一个被访问的节点伸展至根
    splay->root = sSplay(node);
    return splay->root;
}

//Splay判断某关键码是否在节点中
int SplayFind(SPLAYTREE *splay, SPLAYNODE *node, const void *e)
{
    if (NULL == node)
    {
        return 0;
    }
    return (0 == splay->cmpFn(node->key, e));
}

//Splay中插入关键码
SPLAYNODE *SplayInsert(SPLAYTREE *splay, const void *e)
{
    //原树为空
    if (SplayEmpty(splay))
    {
        SPLAYNODE *newNode = nodeNew(splay->keySize, e);
        if (NULL == newNode)
        {
            return NULL;
        }
        splay->root = newNode;
        splay->size ++;
        return splay->root;
    }
    //此时node指向根节点且非空
    SPLAYNODE *node = SplaySearch(splay, e);
    //目标节点已存在
    if (SplayFind(splay, node, e))
    {
        return node;
    }
    SPLAYNODE *newNode;
    if (0 < splay->cmpFn(e, node->key)) //待插入的节点大于根节点
    {
        newNode = nodeNew(splay->keySize, e);
        if (NULL == newNode)
        {
            return NULL;
        }
        newNode->lc = node;
        newNode->rc = node->rc;
        node->parent = newNode;
        if (HasRChild(*node))
        {
            node->rc->parent = newNode;
            node->rc = NULL;
        }
    }
    else //待插入节点小于根节点
    {
        newNode = nodeNew(splay->keySize, e);
        if (NULL == newNode)
        {
            return NULL;
        }
        newNode->lc = node->lc;
        newNode->rc = node;
        node->parent = newNode;
        if (HasLChild(*node))
        {
            node->lc->parent = newNode;
            node->lc = NULL;
        }
    }
    splay->root = newNode;
    splay->size ++;
    return splay->root;
}

//Splay中删除关键码所在节点，返回值：0--成功，!0--失败
static int splayRemoveAt(SPLAYTREE *splay, void *e, SplayFree *freeFn)
{
    SPLAYNODE *node = SplaySearch(splay, e);
    //查不到要删除的节点，删除失败
    if ((NULL == node) || (!SplayFind(splay, node, e)))
    {
        return -1;
    }
    //经过SplaySearch后，待删除的节点已被伸展至树根
    SPLAYNODE *w = splay->root;
    if (!HasLChild(*(splay->root))) //若无左子树则直接删除
    {
        splay->root = splay->root->rc;
        if (NULL != splay->root)
        {
            splay->root->parent = NULL;
        }
    }
    else if (!HasRChild(*(splay->root))) //若无右子树则直接删除
    {
        splay->root = splay->root->lc;
        if (NULL != splay->root)
        {
            splay->root->parent = NULL;
        }
    }
    else //左、右子树都存在
    {
        SPLAYNODE *lTree = splay->root->lc;
        lTree->parent = NULL;
        splay->root->lc = NULL; //暂时将左子树切除
        splay->root = splay->root->rc; //只保留右子树
        splay->root->parent = NULL;
        //以原树根为目标，做一次失败查找，右子树最小节点必伸展至根，且其左子树必为空
        SplaySearch(splay, e);
        splay->root->lc = lTree;
        lTree->parent = splay->root;
    }
    nodeDispose(w, freeFn);
    splay->size --;
    return 0;
}

//Splay中删除关键码，返回值：0--成功，!0--失败
int SplayRemove(SPLAYTREE *splay, void *e)
{
    return splayRemoveAt(splay, e, splay->freeFn);
}

//Splay中删除关键码（关键码非深度删除），返回值：0--成功，!0--失败
int SplayRemoveU(SPLAYTREE *splay, void *e)
{
    return splayRemoveAt(splay, e, NULL);
}
```