```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
```

```
73  static goAlongLeftBranch(SPLAYNODE *node, STACK *s)
74  {
75      SPLAYNODE *cur = node;
76      while (NULL != cur)
77      {
78          StackPush(s, &cur);
79          cur = cur->lc;
80      }
81  }
82
83  //Splay销毁
84  void SplayDispose(SPLAYTREE *splay)
85  {
86      if (SplayEmpty(splay))
87      {
88          return ;
89      }
90      STACK splayNodeStack;
91      //栈中存放splay节点指针
92      StackNew(&splayNodeStack, sizeof(SPLAYNODE *), NULL);
93      SPLAYNODE *node = splay->root, *cur;
94      while (1)
95      {
96          //从当前节点出发，逐批入栈
97          goAlongLeftBranch(node, &splayNodeStack);
98          //所有节点处理完毕
99          if (StackEmpty(&splayNodeStack))
100         {
101             break;
102         }
103         //弹出栈顶节点并访问之
104         StackPop(&splayNodeStack, &node);
105         cur = node;
106         node = node->rc; //转向右子树
107         nodeDispose(cur, splay->freeFn);
108     }
109     StackDispose(&splayNodeStack);
110     splay->root = NULL;
111     splay->hot = NULL;
112     splay->size = 0;
113 }
114
115 //二叉树中序遍历算法（迭代版#1）
116 static void travIn_V1(SPLAYTREE *splay, SplayTraverseOp *traverseOpFn)
117 {
118     STACK splayNodeStack;
119     //栈中存放avl节点指针
120     StackNew(&splayNodeStack, sizeof(SPLAYNODE *), NULL);
121     SPLAYNODE *node = splay->root;
122     while (1)
123     {
124         //从当前节点出发，逐批入栈
125         goAlongLeftBranch(node, &splayNodeStack);
126         //所有节点处理完毕
127         if (StackEmpty(&splayNodeStack))
128         {
129             break;
130         }
131         //弹出栈顶节点并访问之
132         StackPop(&splayNodeStack, &node);
133         traverseOpFn(node->key);
134         node = node->rc; //转向右子树
135     }
136     StackDispose(&splayNodeStack);
137 }
138
139 //二叉树中序遍历算法（迭代版#2，版本#1的等价形式）
140 static void travIn_V2(SPLAYTREE *splay, SplayTraverseOp *traverseOpFn)
141 {
142     STACK splayNodeStack;
143     //栈中存放avl节点指针
144     StackNew(&splayNodeStack, sizeof(SPLAYNODE *), NULL);
145     SPLAYNODE *node = splay->root;
```

```
146         while (1)
147         {
148             if (NULL != node)
149             {
150                 StackPush(&splayNodeStack, &node);
151                 node = node->lc;
152             }
153             else if (!StackEmpty(&splayNodeStack))
154             {
155                 StackPop(&splayNodeStack, &node);
156                 traverseOpFn(node->key);
157                 node = node->rc; //转向右子树
158             }
159             else
160             {
161                 break;
162             }
163         }
164         StackDispose(&splayNodeStack);
165     }
166
167     //定位节点node在中序遍历中的直接后继
168     static SPLAYNODE *succ(SPLAYNODE *node)
169     {
170         SPLAYNODE *s = node;
171         if (NULL != s->rc) //若有右孩子，则直接后继必在右子树中
172         {
173             s = s->rc;
174             while (HasLChild(*s))
175             {
176                 s = s->lc;
177             }
178         }
179         else
180         {
181             while (IsRChild(*s))
182             {
183                 s = s->parent;
184             }
185             s = s->parent;
186         }
187         return s;
188     }
189
190     //二叉树中序遍历算法（迭代版#3）
191     static void travIn_V3(SPLAYTREE *splay, SplayTraverseOp *traverseOpFn)
192     {
193         //前一步是否刚从右子树回溯--省去栈，仅O(1)辅助空间
194         int backtrack = 0;
195         SPLAYNODE *node = splay->root;
196         while (1)
197         {
198             //若有左子树且不是刚刚回溯，则深入遍历左子树
199             if (!backtrack && HasLChild(*node))
200             {
201                 node = node->lc;
202             } //否则无左子树或刚刚回溯
203             else
204             {
205                 traverseOpFn(node->key);
206                 //右子树非空，深入右子树继续遍历，并关闭回溯标志
207                 if (HasRChild(*node))
208                 {
209                     node = node->rc;
210                     backtrack = 0;
211                 }
212                 else //右子树为空则回溯并设置回溯标志
213                 {
214                     node = succ(node);
215                     if (NULL == node)
216                     {
217                         break;
218                     }
```

```
                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
        {
```

```
292                 if (IsLChild(*p)) //zig-zig
293                 {
294                     attachAsLChild(g, p->rc);
295                     attachAsLChild(p, node->rc);
296                     attachAsRChild(p, g);
297                     attachAsRChild(node, p);
298                 }
299                 else //zig-zag
300                 {
301                     attachAsRChild(g, node->lc);
302                     attachAsLChild(p, node->rc);
303                     attachAsLChild(node, g);
304                     attachAsRChild(node, p);
305                 }
306             }
307             else //zag
308             {
309                 if (IsLChild(*p)) //zag-zig
310                 {
311                     attachAsRChild(p, node->lc);
312                     attachAsLChild(g, node->rc);
313                     attachAsLChild(node, p);
314                     attachAsRChild(node, g);
315                 }
316                 else //zag-zag
317                 {
318                     attachAsRChild(g, p->lc);
319                     attachAsLChild(p, g);
320                     attachAsRChild(p, node->lc);
321                     attachAsLChild(node, p);
322                 }
323             }
324             if (NULL == gg)
325             {
326                 node->parent = NULL;
327             }
328             else
329             {
330                 (g == gg->lc) ? attachAsLChild(gg, node) : attachAsRChild(gg, node);
331             }
332         } //双层伸展结束时，必有g == NULL，但p可能非空
333         p = node->parent;
334         if (NULL != p)
335         {
336             if (IsLChild(*node)) //zig
337             {
338                 attachAsLChild(p, node->rc);
339                 attachAsRChild(node, p);
340             }
341             else //zag
342             {
343                 attachAsRChild(p, node->lc);
344                 attachAsLChild(node, p);
345             }
346         }
347         node->parent = NULL;
348         return node;
349     }
350
351     //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
352     SPLAYNODE *SplaySearch(SPLAYTREE *splay, const void *e)
353     {
354         if (NULL == splay->root)
355         {
356             return NULL;
357         }
358         SPLAYNODE *node = splay->root;
359         //hot指向当前节点的父节点
360         splay->hot = NULL;
361         while (NULL != node)
362         {
363             if (0 == splay->cmpFn(e, node->key))
364             {
```

```
365                 break ;
366             }
367             else if (0 < splay->cmpFn(e, node->key))
368             {
369                 splay->hot = node;
370                 node = node->rc;
371             }
372             else
373             {
374                 splay->hot = node;
375                 node = node->lc;
376             }
377         }
378         node = node ? node : splay->hot;
379         //将最后一个被访问的节点伸展至根
380         splay->root = sSplay(node);
381         return splay->root;
382     }
383
384     //Splay判断某关键码是否在节点中
385     int SplayFind(SPLAYTREE *splay, SPLAYNODE *node, const void *e)
386     {
387         if (NULL == node)
388         {
389             return 0;
390         }
391         return (0 == splay->cmpFn(node->key, e));
392     }
393
394     //Splay中插入关键码
395     SPLAYNODE *SplayInsert(SPLAYTREE *splay, const void *e)
396     {
397         //原树为空
398         if (SplayEmpty(splay))
399         {
400             SPLAYNODE *newNode = nodeNew(splay->keySize, e);
401             if (NULL == newNode)
402             {
403                 return NULL;
404             }
405             splay->root = newNode;
406             splay->size ++;
407             return splay->root;
408         }
409         //此时node指向根节点且非空
410         SPLAYNODE *node = SplaySearch(splay, e);
411         //目标节点已存在
412         if (SplayFind(splay, node, e))
413         {
414             return node;
415         }
416         SPLAYNODE *newNode;
417         if (0 < splay->cmpFn(e, node->key)) //待插入的节点大于根节点
418         {
419             newNode = nodeNew(splay->keySize, e);
420             if (NULL == newNode)
421             {
422                 return NULL;
423             }
424             newNode->lc = node;
425             newNode->rc = node->rc;
426             node->parent = newNode;
427             if (HasRChild(*node))
428             {
429                 node->rc->parent = newNode;
430                 node->rc = NULL;
431             }
432         }
433         else //待插入节点小于根节点
434         {
435             newNode = nodeNew(splay->keySize, e);
436             if (NULL == newNode)
437             {
```

```
438                 return NULL;
439             }
440             newNode->lc = node->lc;
441             newNode->rc = node;
442             node->parent = newNode;
443             if (HasLChild(*node))
444             {
445                 node->lc->parent = newNode;
446                 node->lc = NULL;
447             }
448         }
449         splay->root = newNode;
450         splay->size ++;
451         return splay->root;
452     }
453
454     //Splay中删除关键码所在节点，返回值：0--成功，!0--失败
455     static int splayRemoveAt(SPLAYTREE *splay, void *e, SplayFree *freeFn)
456     {
457         SPLAYNODE *node = SplaySearch(splay, e);
458         //查不到要删除的节点，删除失败
459         if ((NULL == node) || (!SplayFind(splay, node, e)))
460         {
461             return -1;
462         }
463         //经过SplaySearch后，待删除的节点已被伸展至树根
464         SPLAYNODE *w = splay->root;
465         if (!HasLChild(*(splay->root))) //若无左子树则直接删除
466         {
467             splay->root = splay->root->rc;
468             if (NULL != splay->root)
469             {
470                 splay->root->parent = NULL;
471             }
472         }
473         else if (!HasRChild(*(splay->root))) //若无右子树则直接删除
474         {
475             splay->root = splay->root->lc;
476             if (NULL != splay->root)
477             {
478                 splay->root->parent = NULL;
479             }
480         }
481         else //左、右子树都存在
482         {
483             SPLAYNODE *lTree = splay->root->lc;
484             lTree->parent = NULL;
485             splay->root->lc = NULL; //暂时将左子树切除
486             splay->root = splay->root->rc; //只保留右子树
487             splay->root->parent = NULL;
488             //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
489             SplaySearch(splay, e);
490             splay->root->lc = lTree;
491             lTree->parent = splay->root;
492         }
493         nodeDispose(w, freeFn);
494         splay->size --;
495         return 0;
496     }
497
498     //Splay中删除关键码，返回值：0--成功，!0--失败
499     int SplayRemove(SPLAYTREE *splay, void *e)
500     {
501         return splayRemoveAt(splay, e, splay->freeFn);
502     }
503
504     //Splay中删除关键码（关键码非深度删除），返回值：0--成功，!0--失败
505     int SplayRemoveU(SPLAYTREE *splay, void *e)
506     {
507         return splayRemoveAt(splay, e, NULL);
508     }
```