```c
/* Avl.c */
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
#include <string.h>
#include "Avl.h"
#include "LinkStack.h"
#include "LinkQueue.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)
//理想平衡条件
#define Balanced(x)      (stature((x).lc) == stature((x).rc))
//平衡因子
#define BalFac(x)        (stature((x).lc) - stature((x).rc))
//AVL平衡条件
#define AvlBalanced(x)   ((-2 < BalFac(x)) && (BalFac(x) < 2))
//在左、右孩子中取更高者，在AVL平衡调整前，借此确定重构方案
#define tallerChild(p)   ( \
    stature((p)->lc) > stature((p)->rc) ? (p)->lc : ( /*左高*/ \
    stature((p)->lc) < stature((p)->rc) ? (p)->rc : ( /*右高*/ \
    IsLChild(*(p)) ? (p)->lc : (p)->rc /*等高：与父亲p同侧者(zig-zig或zag-zag)优先*/ \
    ) \
    ) \
)

static AVLNODE *nodeNew(int keySize, const void *e)
{
    AVLNODE *newNode = (AVLNODE *)malloc(sizeof(AVLNODE) + keySize);
    if (NULL == newNode)
    {
        return NULL;
    }
    newNode->parent = NULL;
    newNode->lc = NULL;
    newNode->rc = NULL;
    newNode->height = 0;
    memcpy(newNode->key, e, keySize);
    return newNode;
}

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

//Avl初始化
void AvlNew(AVLTREE *avlTree, int keySize, AvlCmp *cmpFn, AvlFree *freeFn)
{
    assert(keySize > 0);
    assert(NULL != cmpFn);
    avlTree->root = NULL;
    avlTree->hot = NULL;
    avlTree->size = 0;
```

```
    avlTree->keySize = keySize;
    avlTree->cmpFn = cmpFn;
    avlTree->freeFn = freeFn;
}

//Avl判空
int AvlEmpty(AVLTREE *avlTree)
{
    return (avlTree->size == 0);
}

//Avl规模
int AvlSize(AVLTREE *avlTree)
{
    return avlTree->size;
}

//Avl树高度
int AvlHeight(AVLTREE *avlTree)
{
    return avlTree->root->height;
}

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

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

static void visitAlongLeftBranch(AVLNODE *node, STACK *s, AvlTraverseOp *traverseOpFn)
{
    AVLNODE *x = node;
    while (NULL != x)
    {
        traverseOpFn(x->key);
        StackPush(s, &(x->rc));
```

```
146             x = x->lc;
147         }
148     }
149
150     //Avl先序遍历（非递归）
151     void AvlTravPre(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
152     {
153         if (NULL == traverseOpFn || AvlEmpty(avlTree))
154         {
155             return ;
156         }
157         STACK avlNodeStack;
158         StackNew(&avlNodeStack, sizeof(AVLNODE *), NULL);
159         AVLNODE *node = avlTree->root;
160         while (1)
161         {
162             visitAlongLeftBranch(node, &avlNodeStack, traverseOpFn);
163             if (StackEmpty(&avlNodeStack))
164             {
165                 break;
166             }
167             StackPop(&avlNodeStack, &node);
168         }
169         StackDispose(&avlNodeStack);
170     }
171
172     static void travPreRecAt(AVLNODE *node, AvlTraverseOp *traverseOpFn)
173     {
174         if (NULL == node)
175         {
176             return ;
177         }
178         traverseOpFn(node->key);
179         travPreRecAt(node->lc, traverseOpFn);
180         travPreRecAt(node->rc, traverseOpFn);
181     }
182
183     //Avl先序遍历（递归）
184     void AvlTravPreRec(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
185     {
186         if (NULL == traverseOpFn || AvlEmpty(avlTree))
187         {
188             return ;
189         }
190         travPreRecAt(avlTree->root, traverseOpFn);
191     }
192
193     //二叉树中序遍历算法（迭代版#1）
194     static void travIn_V1(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
195     {
196         STACK avlNodeStack;
197         //栈中存放avl节点指针
198         StackNew(&avlNodeStack, sizeof(AVLNODE *), NULL);
199         AVLNODE *node = avlTree->root;
200         while (1)
201         {
202             //从当前节点出发，逐批入栈
203             goAlongLeftBranch(node, &avlNodeStack);
204             //所有节点处理完毕
205             if (StackEmpty(&avlNodeStack))
206             {
207                 break;
208             }
209             //弹出栈顶节点并访问之
210             StackPop(&avlNodeStack, &node);
211             traverseOpFn(node->key);
212             node = node->rc; //转向右子树
213         }
214         StackDispose(&avlNodeStack);
215     }
216
217     //二叉树中序遍历算法（迭代版#2，版本#1的等价形式）
218     static void travIn_V2(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
```

```
{
    STACK avlNodeStack;
    //栈中存放avl节点指针
    StackNew(&avlNodeStack, sizeof(AVLNODE *), NULL);
    AVLNODE *node = avlTree->root;
    while (1)
    {
        if (NULL != node)
        {
            StackPush(&avlNodeStack, &node);
            node = node->lc;
        }
        else if (!StackEmpty(&avlNodeStack))
        {
            StackPop(&avlNodeStack, &node);
            traverseOpFn(node->key);
            node = node->rc;
        }
        else
        {
            break;
        }
    }
    StackDispose(&avlNodeStack);
}

//定位节点node在中序遍历中的直接后继
static AVLNODE *succ(AVLNODE *node)
{
    AVLNODE *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(AVLTREE *avlTree, AvlTraverseOp *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;
    AVLNODE *node = avlTree->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;
            }
        }
    }
}

//Avl中序遍历（非递归）
void AvlTravIn(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
{
    if (NULL == traverseOpFn || AvlEmpty(avlTree))
    {
        return ;
    }
    travIn_V1(avlTree, traverseOpFn);
}

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

//Avl中序遍历（递归）
void AvlTravInRec(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
{
    if (NULL == traverseOpFn || AvlEmpty(avlTree))
    {
        return ;
    }
    travInRecAt(avlTree->root, traverseOpFn);
}

//在以s栈顶节点为根的子树中，
//找到最高左侧可见叶节点（highest leaf visible from left）
static void gotoHLVFL(STACK *s)
{
    AVLNODE *x;
    StackTop(s, &x);
    while (NULL !=x)
    {
        if (HasLChild(*x)) //尽可能向左
        {
            if (HasRChild(*x)) //若有右孩子，优先入栈
            {
                StackPush(s, &(x->rc));
            }
            StackPush(s, &(x->lc));
        }
        else
        {
            //此处x->rc可能为空，
            //左、右孩子都不存在时，将NULL入栈，便于while退出
            StackPush(s, &(x->rc));
        }
        StackTop(s, &x);
    }
    StackPop(s, NULL); //返回之前，弹出栈顶空节点
}

//Avl后序遍历（非递归）
void AvlTravPost(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
{
    if (NULL == traverseOpFn || AvlEmpty(avlTree))
```

```
365         {
366             return ;
367         }
368         STACK avlNodeStack;
369         StackNew(&avlNodeStack, sizeof(AVLNODE *), NULL);
370         AVLNODE *x = avlTree->root;
371         StackPush(&avlNodeStack, &x);
372         while (!StackEmpty(&avlNodeStack))
373         {
374             AVLNODE *nodeTop;
375             StackTop(&avlNodeStack, &nodeTop);
376             //若栈顶非当前节点之父，则必为其右兄
377             if (nodeTop != x->parent)
378             {
379                 gotoHLVFL(&avlNodeStack);
380             }
381             StackPop(&avlNodeStack, &x);
382             traverseOpFn(x->key);
383         }
384         StackDispose(&avlNodeStack);
385     }
386
387     static void travPostRecAt(AVLNODE *node, AvlTraverseOp *traverseOpFn)
388     {
389         if (NULL == node)
390         {
391             return ;
392         }
393         travInRecAt(node->lc, traverseOpFn);
394         travInRecAt(node->rc, traverseOpFn);
395         traverseOpFn(node->key);
396     }
397
398     //Avl后序遍历（递归）
399     void AvlTravPostRec(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
400     {
401         if (NULL == traverseOpFn || AvlEmpty(avlTree))
402         {
403             return ;
404         }
405         travPostRecAt(avlTree->root, traverseOpFn);
406     }
407
408     //Avl层序遍历
409     void AvlTravLevel(AVLTREE *avlTree, AvlTraverseOp *traverseOpFn)
410     {
411         if (NULL == traverseOpFn || AvlEmpty(avlTree))
412         {
413             return ;
414         }
415         QUEUE avlNodeQueue;
416         QueueNew(&avlNodeQueue, sizeof(AVLNODE *), NULL);
417         AVLNODE *node = avlTree->root;
418         QueueEn(&avlNodeQueue, &node);
419         while (!QueueEmpty(&avlNodeQueue))
420         {
421             QueueDe(&avlNodeQueue, &node);
422             traverseOpFn(node->key);
423             if (HasLChild(*node))
424             {
425                 QueueEn(&avlNodeQueue, &(node->lc));
426             }
427             if (HasRChild(*node))
428             {
429                 QueueEn(&avlNodeQueue, &(node->rc));
430             }
431         }
432         QueueDispose(&avlNodeQueue);
433     }
434
435     //Avl中查找关键码所在节点，hot指向当前节点的父节点
436     AVLNODE *AvlSearch(AVLTREE *avlTree, const void *e)
437     {
```

```
438         AVLNODE *node = avlTree->root;
439         avlTree->hot = NULL;
440         while (NULL != node)
441         {
442             if (0 == avlTree->cmpFn(e, node->key))
443             {
444                 break ;
445             }
446             avlTree->hot = node;
447             if (0 < avlTree->cmpFn(e, node->key))
448             {
449                 node = node->rc;
450             }
451             else
452             {
453                 node = node->lc;
454             }
455         }
456         return node;
457     }
458
459     int max(int a, int b)
460     {
461         return (a > b ? a : b);
462     }
463
464     //更新节点node的高度
465     static int updateHeight(AVLNODE *node)
466     {
467         return node->height = 1 + max(stature(node->lc), stature(node->rc));
468     }
469
470     //更新节点node及其祖先的高度
471     static void updateHeightAbove(AVLNODE *node)
472     {
473         AVLNODE *cur = node;
474         while (NULL != cur)
475         {
476             updateHeight(cur);
477             cur = cur->parent;
478         }
479     }
480
481     //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）
482     //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
483     static AVLNODE *connect34(AVLNODE *a, AVLNODE *b, AVLNODE *c, AVLNODE *T0, AVLNODE
        *T1, AVLNODE *T2, AVLNODE *T3)
484     {
485         a->lc = T0;
486         if (T0)
487         {
488             T0->parent = a;
489         }
490         a->rc = T1;
491         if (T1)
492         {
493             T1->parent = a;
494         }
495         updateHeight(a);
496
497         c->lc = T2;
498         if (T2)
499         {
500             T2->parent = c;
501         }
502         c->rc = T3;
503         if (T3)
504         {
505             T3->parent = c;
506         }
507         updateHeight(c);
508
509         b->lc = a;
```

```
510         a->parent = b;
511         b->rc = c;
512         c->parent = b;
513         updateHeight(b);
514         return b;
515     }
516
517     //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
518     //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
519     static AVLNODE *rotateAt(AVLNODE *grandsonNode)
520     {
521         AVLNODE *p = grandsonNode->parent;
522         AVLNODE *g = p->parent;
523         if (IsLChild(*p)) //zig
524         {
525             if (IsLChild(*grandsonNode)) //zig-zig
526             {
527                 p->parent = g->parent; //向上联接
528                 return connect34(grandsonNode, p, g, grandsonNode->lc, grandsonNode->rc,
                    p->rc, g->rc);
529             }
530             else //zig-zag
531             {
532                 grandsonNode->parent = g->parent; //向上联接
533                 return connect34(p, grandsonNode, g, p->lc, grandsonNode->lc,
                    grandsonNode->rc, g->rc);
534             }
535         }
536         else //zag
537         {
538             if (IsLChild(*grandsonNode)) //zag-zig
539             {
540                 grandsonNode->parent = g->parent; //向上联接
541                 return connect34(g, grandsonNode, p, g->lc, grandsonNode->lc,
                    grandsonNode->rc, p->rc);
542             }
543             else //zag-zag
544             {
545                 p->parent = g->parent; //向上联接
546                 return connect34(g, p, grandsonNode, g->lc, p->lc, grandsonNode->lc,
                    grandsonNode->rc);
547             }
548         }
549     }
550
551     //Avl中插入关键码
552     AVLNODE *AvlInsert(AVLTREE *avlTree, const void *e)
553     {
554         AVLNODE *node = AvlSearch(avlTree, e);
555         if (NULL != node)
556         {
557             return node;
558         }
559         //新节点初始化
560         node = nodeNew(avlTree->keySize, e);
561         if (NULL == node)
562         {
563             return NULL;
564         }
565         node->parent = avlTree->hot;
566         //空树时指定根
567         if (NULL == node->parent)
568         {
569             avlTree->root = node;
570         }
571         else
572         {
573             if (0 < avlTree->cmpFn(e, avlTree->hot->key))
574             {
575                 avlTree->hot->rc = node;
576             }
577             else
578             {
```

```
579                 avlTree->hot->lc = node;
580             }
581         }
582         avlTree->size ++;
583 
584         AVLNODE *g = avlTree->hot;
585         for (; g; g = g->parent)
586         {
587             if (!AvlBalanced(*g))
588             {
589                 //重新接入原树
590                 if (IsRoot(*g))
591                 {
592                     avlTree->root = rotateAt(tallerChild(tallerChild(g)));
593                 }
594                 else if (IsLChild(*g))
595                 {
596                     AVLNODE *u = g->parent;
597                     u->lc = rotateAt(tallerChild(tallerChild(g)));
598                 }
599                 else
600                 {
601                     AVLNODE *u = g->parent;
602                     u->rc = rotateAt(tallerChild(tallerChild(g)));
603                 }
604                 break;
605             }
606             else
607             {
608                 updateHeight(g);
609             }
610         }
611         return node;
612     }
613 
614     static void swap(void *dataAddr1, void *dataAddr2, int dataSize)
615     {
616         char *tmp = (char *)malloc(dataSize);
617         memcpy(tmp, dataAddr1, dataSize);
618         memcpy(dataAddr1, dataAddr2, dataSize);
619         memcpy(dataAddr2, tmp, dataSize);
620         free(tmp);
621     }
622 
623     //删除node所指节点
624     //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
625     static AVLNODE *removeAt(AVLNODE **node, AVLNODE **hot, AvlFree *freeFn, int keySize)
626     {
627         AVLNODE *w = *node; //实际被删除的节点
628         AVLNODE *successor = NULL; //实际被删除节点的接替者
629 
630         //若**node的左子树为空，则直接将**node替换为其右子树
631         if (!HasLChild(**node))
632         {
633             *node = (*node)->rc;
634             successor = *node;
635         }
636         else if (!HasRChild(**node)) //右子树为空，对称处理
637         {
638             *node = (*node)->lc;
639             successor = *node;
640         }
641         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
642         {
643             w = succ(w); //在右子树中找到直接后继
644             swap((*node)->key, w->key, keySize);
645             AVLNODE *u = w->parent;
646             successor = w->rc; //此时待删除节点不可能有左孩子
647             //隔离被删除节点w
648             if (*node == u)
649             {
650                 u->rc = successor;
651             }
```

```
652             else
653             {
654                 u->lc = successor;
655             }
656         }
657         //记录实际被删除节点的父亲
658         *hot = w->parent;
659         if (successor)
660         {
661             successor->parent = *hot; //将被删除节点的接替者与hot相联
662         }
663         nodeDispose(w, freeFn);
664         return successor;
665     }
666
667     static int avlRemoveBase(AVLTREE *avlTree, void *e, AvlFree *freeFn)
668     {
669         AVLNODE *node = AvlSearch(avlTree, e);
670         //查不到要删除的节点，删除失败
671         if (NULL == node)
672         {
673             return -1;
674         }
675         AVLNODE *hot = NULL; //记录删除节点的父亲
676         removeAt(&node, &hot, freeFn, avlTree->keySize);
677         avlTree->size --;
678
679         AVLNODE *g = hot;
680         for (; g; g = g->parent)
681         {
682             if (!AvlBalanced(*g))
683             {
684                 AVLNODE *localRoot;
685                 if (IsRoot(*g))
686                 {
687                     localRoot = rotateAt(tallerChild(tallerChild(g)));
688                     avlTree->root = localRoot;
689                 }
690                 else if (IsLChild(*g))
691                 {
692                     AVLNODE *u = g->parent;
693                     localRoot = rotateAt(tallerChild(tallerChild(g)));
694                     u->lc = localRoot;
695                 }
696                 else
697                 {
698                     AVLNODE *u = g->parent;
699                     localRoot = rotateAt(tallerChild(tallerChild(g)));
700                     u->rc = localRoot;
701                 }
702                 g = localRoot;
703             }
704             updateHeight(g); //即便g未失衡，其高度也可能降低
705         }
706         return 0;
707     }
708
709     //Avl中删除关键码所在节点，返回值：0成功，!0失败
710     int AvlRemove(AVLTREE *avlTree, void *e)
711     {
712         return avlRemoveBase(avlTree, e, avlTree->freeFn);
713     }
714
715     //Avl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
716     int AvlRemoveU(AVLTREE *avlTree, void *e)
717     {
718         return avlRemoveBase(avlTree, e, NULL);
719     }
```