

```

1  /* RBTREE.c */
2  #include <stdlib.h>
3  #include <malloc.h>
4  #include <assert.h>
5  #include <string.h>
6  #include "RBTREE.h"
7  #include "LinkStack.h"
8
9  /* x表示值，p表示指针 */
10 #define IsRoot(x)      (!((x).parent))
11 #define IsLChild(x)    (!IsRoot(x) && (&(x) == (x).parent->lc))
12 #define IsRChild(x)    (!IsRoot(x) && (&(x) == (x).parent->rc))
13 #define HasParent(x)   (!IsRoot(x))
14 #define HasLChild(x)   ((x).lc)
15 #define HasRChild(x)   ((x).rc)
16 #define HasChild(x)    (HasLChild(x) || HasRChild(x))
17 #define HasBothChild(x) (HasLChild(x) && HasRChild(x))
18 #define IsLeaf(x)      (!HasChild(x))
19 //获取x的兄弟节点
20 #define Sibling(x)      (IsLChild(x) ? (x).parent->rc : (x).parent->lc)
21 //获取x的叔叔节点
22 #define Uncle(x)        (IsLChild(*(x).parent)) ? (x).parent->parent->rc :
(x).parent->parent->lc)
23 //节点高度，空树高度为-1
24 #define stature(p)      ((p) ? (p)->height : -1)
25 //外部节点也视作黑节点
26 #define isBlack(p)      (! (p) || (RB_BLACK == (p)->color))
27 #define isRed(p)        (!isBlack(p))
28 //红黑树高度更新条件
29 #define BlackHeightUpdated(x) ( \
30     (stature((x).lc) == stature((x).rc)) && \
31     ((x).height == (isRed(&x) ? stature((x).lc) : stature((x).lc) + 1)) \
32 )
33
34 static RBREENODE *nodeNew(int keySize, const void *e)
35 {
36     RBREENODE *newNode = (RBREENODE *)malloc(sizeof(RBREENODE) + keySize);
37     if (NULL == newNode)
38     {
39         return NULL;
40     }
41     newNode->parent = NULL;
42     newNode->lc = NULL;
43     newNode->rc = NULL;
44     newNode->height = -1;
45     newNode->color = RB_RED;
46     memcpy(newNode->key, e, keySize);
47     return newNode;
48 }
49
50 static void nodeDispose(RBREENODE *node, RBTreeFree *freeFn)
51 {
52     if (NULL != freeFn)
53     {
54         freeFn(node->key);
55     }
56     free(node);
57 }
58
59 //RBTREE初始化
60 void RBTreeNew(RBTREE *rbTree, int keySize, RBTreeCmp *cmpFn, RBTreeFree *freeFn)
61 {
62     assert(keySize > 0);
63     assert(NULL != cmpFn);
64     rbTree->root = NULL;
65     rbTree->hot = NULL;
66     rbTree->size = 0;
67     rbTree->keySize = keySize;
68     rbTree->cmpFn = cmpFn;
69     rbTree->freeFn = freeFn;
70 }
71
72 //RBTREE判空

```

```

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

```

```

146         {
147             break;
148         }
149         //弹出栈顶节点并访问之
150         StackPop(&brNodeStack, &node);
151         traverseOpFn(node->key);
152         node = node->rc; //转向右子树
153     }
154     StackDispose(&brNodeStack);
155 }
156
157 //二叉树中序遍历算法（迭代版#2，版本#1的等价形式）
158 static void travIn_V2(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
159 {
160     STACK brNodeStack;
161     //栈中存放avl节点指针
162     StackNew(&brNodeStack, sizeof(RBTREENODE *), NULL);
163     RBTREENODE *node = rbTree->root;
164     while (1)
165     {
166         if (NULL != node)
167         {
168             StackPush(&brNodeStack, &node);
169             node = node->lc;
170         }
171         else if (!StackEmpty(&brNodeStack))
172         {
173             StackPop(&brNodeStack, &node);
174             traverseOpFn(node->key);
175             node = node->rc;
176         }
177         else
178         {
179             break;
180         }
181     }
182     StackDispose(&brNodeStack);
183 }
184
185 //定位节点node在中序遍历中的直接后继
186 static RBTREENODE *succ(RBTREENODE *node)
187 {
188     RBTREENODE *s = node;
189     if (NULL != s->rc) //若有右孩子，则直接后继必在右子树中
190     {
191         s = s->rc;
192         while (HasLChild(*s))
193         {
194             s = s->lc;
195         }
196     }
197     else
198     {
199         while (IsRChild(*s))
200         {
201             s = s->parent;
202         }
203         s = s->parent;
204     }
205     return s;
206 }
207
208 //二叉树中序遍历算法（迭代版#3）
209 static void travIn_V3(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
210 {
211     //前一步是否刚从右子树回溯--省去栈，仅O(1)辅助空间
212     int backtrack = 0;
213     RBTREENODE *node = rbTree->root;
214     while (1)
215     {
216         //若有左子树且不是刚刚回溯，则深入遍历左子树
217         if (!backtrack && HasLChild(*node))
218         {

```

```

219         node = node->lc;
220     } //否则无左子树或刚刚回溯
221     else
222     {
223         traverseOpFn(node->key);
224         //右子树非空，深入右子树继续遍历，并关闭回溯标志
225         if (HasRChild(*node))
226         {
227             node = node->rc;
228             backtrack = 0;
229         }
230         else //右子树为空则回溯并设置回溯标志
231         {
232             node = succ(node);
233             if (NULL == node)
234             {
235                 break;
236             }
237             backtrack = 1;
238         }
239     }
240 }
241
242
243 //RBTree中序遍历（非递归）
244 void RBTreeTravIn(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
245 {
246     if (NULL == traverseOpFn || RBTreeEmpty(rbTree))
247     {
248         return ;
249     }
250     travIn_Vl(rbTree, traverseOpFn);
251 }
252
253 static void travInRecAt(RBTREENODE *node, RBTreeTraverseOp *traverseOpFn)
254 {
255     if (NULL == node)
256     {
257         return ;
258     }
259     travInRecAt(node->lc, traverseOpFn);
260     traverseOpFn(node->key);
261     travInRecAt(node->rc, traverseOpFn);
262 }
263
264 //RBTree中序遍历（递归）
265 void RBTreeTravInRec(RBTREE *rbTree, RBTreeTraverseOp *traverseOpFn)
266 {
267     if (NULL == traverseOpFn || RBTreeEmpty(rbTree))
268     {
269         return ;
270     }
271     travInRecAt(rbTree->root, traverseOpFn);
272 }
273
274 //RBTree中查找关键码所在节点，hot指向当前节点的父节点
275 RBTREENODE *RBTreeSearch(RBTREE *rbTree, const void *e)
276 {
277     RBTREENODE *node = rbTree->root;
278     rbTree->hot = NULL;
279     while (NULL != node)
280     {
281         if (0 == rbTree->cmpFn(e, node->key))
282         {
283             break ;
284         }
285         rbTree->hot = node;
286         if (0 < rbTree->cmpFn(e, node->key))
287         {
288             node = node->rc;
289         }
290         else
291         {

```

```

292         node = node->lc;
293     }
294 }
295 return node;
296 }
297
298 int max(int a, int b)
299 {
300     return (a > b ? a : b);
301 }
302
303 //更新节点node的黑高度
304 static int updateHeight(RBTREENODE *node)
305 {
306     //孩子一般黑高度相等，除非出现双黑
307     node->height = max(stature(node->lc), stature(node->rc));
308     return (isBlack(node) ? (++ node->height) : node->height);
309 }
310
311 //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）
312 //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
313 static RBTREENODE *connect34(RBTREENODE *a, RBTREENODE *b, RBTREENODE *c, RBTREENODE
314 *T0, RBTREENODE *T1, RBTREENODE *T2, RBTREENODE *T3)
315 {
316     a->lc = T0;
317     if (T0)
318     {
319         T0->parent = a;
320     }
321     a->rc = T1;
322     if (T1)
323     {
324         T1->parent = a;
325     }
326     updateHeight(a);
327
328     c->lc = T2;
329     if (T2)
330     {
331         T2->parent = c;
332     }
333     c->rc = T3;
334     if (T3)
335     {
336         T3->parent = c;
337     }
338     updateHeight(c);
339
340     b->lc = a;
341     a->parent = b;
342     b->rc = c;
343     c->parent = b;
344     updateHeight(b);
345     return b;
346 }
347
348 //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
349 //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
350 static RBTREENODE *rotateAt(RBTREENODE *grandsonNode)
351 {
352     RBTREENODE *p = grandsonNode->parent;
353     RBTREENODE *g = p->parent;
354     if (IsLChild(*p)) //zig
355     {
356         if (IsLChild(*grandsonNode)) //zig-zig
357         {
358             p->parent = g->parent; //向上联接
359             return connect34(grandsonNode, p, g, grandsonNode->lc, grandsonNode->rc,
360                             p->rc, g->rc);
361         }
362         else //zig-zag
363         {
364             grandsonNode->parent = g->parent; //向上联接

```

```

363         return connect34(p, grandsonNode, g, p->lc, grandsonNode->lc,
364                           grandsonNode->rc, g->rc);
365     }
366     else //zag
367     {
368         if (IsLChild(*grandsonNode)) //zag-zig
369         {
370             grandsonNode->parent = g->parent; //向上联接
371             return connect34(g, grandsonNode, p, g->lc, grandsonNode->lc,
372                             grandsonNode->rc, p->rc);
373         }
374         else //zag-zag
375         {
376             p->parent = g->parent; //向上联接
377             return connect34(g, p, grandsonNode, g->lc, p->lc, grandsonNode->lc,
378                             grandsonNode->rc);
379         }
380     }
381 }
382 //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：
383 //RR-1：2次颜色翻转，2次黑高度更新，1~2次旋转，不再递归
384 //RR-2：3次颜色翻转，3次黑高度更新，0次旋转，需要递归
385 static void solveDoubleRed(RBTREE *rbTree, RBTREENODE *node) //当前node必为红
386 {
387     //若已（递归）转至树根，则将其转黑，整树黑高度也随之递增
388     if (IsRoot(*node))
389     {
390         rbTree->root->color = RB_BLACK;
391         rbTree->root->height ++;
392         return ;
393     }
394     RBTREENODE *p = node->parent;
395     if (isBlack(p)) //父节点为黑，则终止调整
396     {
397         return ;
398     }
399     //此时p为红，g为黑
400     RBTREENODE *g = p->parent;
401     RBTREENODE *u = Uncle(*node);
402     if (isBlack(u)) //RR-1，叔父节点为黑色
403     {
404         //node与p同侧（即zig-zig或zag-zag）
405         if (IsLChild(*node) == IsLChild(*p))
406         {
407             p->color = RB_BLACK;
408         }
409         else //node与p异侧（即zig-zag或zag-zig）
410         {
411             node->color = RB_BLACK;
412         }
413         g->color = RB_RED;
414         //重新接入原树（向下接入）
415         if (IsRoot(*g))
416         {
417             rbTree->root = rotateAt(node);
418         }
419         else if (IsLChild(*g))
420         {
421             RBTREENODE *gg = g->parent;
422             gg->lc = rotateAt(node);
423         }
424         else
425         {
426             RBTREENODE *gg = g->parent;
427             gg->rc = rotateAt(node);
428         }
429     }
430     else //RR-2，叔父节点为红色
431     {
432         //p由红转黑
433         p->color = RB_BLACK;

```

```

433     p->height ++;
434     //u由红转黑
435     u->color = RB_BLACK;
436     u->height ++;
437     //g由黑转红，高度不变
438     g->color = RB_RED;
439     solveDoubleRed(rbTree, g);
440 }
441 }
442
443 //RBTree中插入关键码
444 RBREENODE *RBTreeInsert(RBTREE *rbTree, const void *e)
445 {
446     RBREENODE *node = RBTreeSearch(rbTree, e);
447     if (NULL != node)
448     {
449         return node;
450     }
451     //新节点初始化
452     node = nodeNew(rbTree->keySize, e);
453     if (NULL == node)
454     {
455         return NULL;
456     }
457     node->parent = rbTree->hot;
458     //空树时指定根
459     if (NULL == node->parent)
460     {
461         rbTree->root = node;
462     }
463     else
464     {
465         if (0 < rbTree->cmpFn(e, rbTree->hot->key))
466         {
467             rbTree->hot->rc = node;
468         }
469         else
470         {
471             rbTree->hot->lc = node;
472         }
473     }
474     rbTree->size ++;
475     solveDoubleRed(rbTree, node);
476     return node;
477 }
478
479 static void swap(void *dataAddr1, void *dataAddr2, int dataSize)
480 {
481     char *tmp = (char *)malloc(dataSize);
482     memcpy(tmp, dataAddr1, dataSize);
483     memcpy(dataAddr1, dataAddr2, dataSize);
484     memcpy(dataAddr2, tmp, dataSize);
485     free(tmp);
486 }
487
488 //删除node所指节点
489 //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
490 static RBREENODE *removeAt(RBREENODE **node, RBREENODE **hot, RBTreeFree *freeFn,
491 int keySize)
492 {
493     RBREENODE *w = *node; //实际被删除的节点
494     RBREENODE *successor = NULL; //实际被删除节点的接替者
495
496     //若**node的左子树为空，则直接将**node替换为其右子树
497     if (!HasLChild(**node))
498     {
499         *node = (*node)->rc;
500         successor = *node;
501     }
502     else if (!HasRChild(**node)) //右子树为空，对称处理
503     {
504         *node = (*node)->lc;
505         successor = *node;
506     }

```

```

505     }
506     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
507     {
508         w = succ(w); //在右子树中找到直接后继
509         swap((*node)->key, w->key, keySize);
510         RBTreeNode *u = w->parent;
511         successor = w->rc; //此时待删除节点不可能有左孩子
512         //隔离被删除节点w
513         if (*node == u)
514         {
515             u->rc = successor;
516         }
517         else
518         {
519             u->lc = successor;
520         }
521     }
522     //记录实际被删除节点的父亲
523     *hot = w->parent;
524     if (successor)
525     {
526         successor->parent = *hot; //将被删除节点的接替者与hot相联
527     }
528     nodeDispose(w, freeFn);
529     return successor;
530 }
531
532 //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
533 //分为三大类共四种情况
534 //BB-1 : 2次颜色翻转，2次黑高度更新，1~2次旋转，不再递归
535 //BB-2R: 2次颜色翻转，2次黑高度更新，0次旋转，不再递归
536 //BB-2B: 1次颜色翻转，1次黑高度更新，0次旋转，需要递归
537 //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
538 static void solveDoubleBlack(RBTree *rbTree, RBTreeNode *node)
539 {
540     RBTreeNode *p = node ? node->parent : rbTree->hot;
541     if (NULL == p)
542     {
543         return ;
544     }
545     //此处node可能为NULL，故不能调用宏Sibling
546     RBTreeNode *s = (node == p->lc) ? p->rc : p->lc;
547     if (isBlack(s)) //兄弟s为黑
548     {
549         //s的红孩子（若左、右孩子皆红，左者优先；皆黑时为NULL）
550         RBTreeNode *t = NULL;
551         if (HasLChild(*s) && isRed(s->lc))
552         {
553             t = s->lc;
554         }
555         else if (HasRChild(*s) && isRed(s->rc))
556         {
557             t = s->rc;
558         }
559         if (NULL != t) //黑s有红孩子：BB-1
560         {
561             //备份原子树根p颜色
562             RBCOLOR oldColor = p->color;
563             RBTreeNode *gg = p->parent;
564             RBTreeNode *b; //重平衡后的子树根节点
565             if (NULL == gg)
566             {
567                 b = rotateAt(t);
568                 rbTree->root = b;
569             }
570             else if (p == gg->lc)
571             {
572                 b = rotateAt(t);
573                 gg->lc = b;
574             }
575             else
576             {
577                 b = rotateAt(t);

```



```

578         gg->rc = b;
579     }
580     if (HasLChild(*b))
581     {
582         b->lc->color = RB_BLACK;
583         updateHeight(b->lc);
584     }
585     if (HasRChild(*b))
586     {
587         b->rc->color = RB_BLACK;
588         updateHeight(b->rc);
589     }
590     b->color = oldColor;
591     updateHeight(b);
592 }
593 else //黑s无红孩子
594 {
595     s->color = RB_RED;
596     s->height --;
597     if (isRed(p)) //BB-2R
598     {
599         p->color = RB_BLACK; //p转黑，高度不变
600     }
601     else //BB-2B
602     {
603         p->height --; //p保持黑，黑高度下降
604         solveDoubleBlack(rbTree, p);
605     }
606 }
607 }
608 else //兄弟s为红: BB-3
609 {
610     s->color = RB_BLACK;
611     p->color = RB_RED;
612     RBREENODE *t = IsLChild(*s) ? s->lc : s->rc; //取t与其父s同侧
613     rbTree->hot = p;
614     RBREENODE *gg = p->parent;
615     if (NULL == gg)
616     {
617         rbTree->root = rotateAt(t);
618     }
619     else if (p == gg->lc)
620     {
621         gg->lc = rotateAt(t);
622     }
623     else
624     {
625         gg->rc = rotateAt(t);
626     }
627     //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
628     solveDoubleBlack(rbTree, node);
629 }
630 }
631
632 static int rbTreeRemoveBase(RBTREE *rbTree, void *e, RBTreeFree *freeFn)
633 {
634     RBREENODE *node = RBTreeSearch(rbTree, e);
635     //查不到要删除的节点，删除失败
636     if (NULL == node)
637     {
638         return -1;
639     }
640     RBREENODE *hot = NULL; //记录删除节点的父亲
641     RBREENODE *r = removeAt(&node, &hot, freeFn, rbTree->keySize);
642     rbTree->hot = hot;
643     rbTree->size --;
644     if (RBTreeEmpty(rbTree))
645     {
646         return 0;
647     }
648     //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
649     if (NULL == rbTree->hot)
650     {

```

```

651         rbTree->root->color = RB_BLACK;
652         updateHeight(rbTree->root);
653         return 0;
654     }
655     //assert: 以下, 原node (现r) 必非根, hot必非空
656     //若所有祖先的黑深度依然平衡, 则无需调整
657     if (BlackHeightUpdated(*(rbTree->hot)))
658     {
659         return 0;
660     }
661     if (isRed(r))
662     {
663         r->color = RB_BLACK;
664         r->height ++;
665         return 0;
666     }
667     //assert: 以下, 原node (现r) 均为黑色
668     solveDoubleBlack(rbTree, r);
669     return 0;
670 }
671
672 //RBTree中删除关键码所在节点, 返回值: 0成功, !0失败
673 int RBTreeRemove(RBTREE *rbTree, void *e)
674 {
675     return rbTreeRemoveBase(rbTree, e, rbTree->freeFn);
676 }
677
678 //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
679 int RBTreeRemoveU(RBTREE *rbTree, void *e)
680 {
681     return rbTreeRemoveBase(rbTree, e, NULL);
682 }

```