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

//初始化非根节点
static BTREENODE *nodeNew(BTREE *bTree)
{
    BTREENODE *newNode = (BTREENODE *)malloc(sizeof(BTREENODE));
    if (NULL == newNode)
    {
        return NULL;
    }
    newNode->parent = NULL;
    //给关键码向量多分配一个节点空间，便于以后上溢分裂
    VectorNew(&(newNode->keyVector), bTree->keySize, bTree->order, 1, bTree->cmpFn, bTree->freeFn);
    //给孩子向量多分配一个节点空间，便于以后上溢分裂
    VectorNew(&(newNode->childVector), sizeof(BTREENODE *), (bTree->order + 1), 1, NULL, NULL);
    return newNode;
}

//初始化根节点
static BTREENODE *rootNew(BTREE *bTree, const void *e)
{
    BTREENODE *root = (BTREENODE *)malloc(sizeof(BTREENODE));
    if (NULL == root)
    {
        return NULL;
    }
    root->parent = NULL;
    //给关键码向量多分配一个节点空间，便于以后上溢分裂
    VectorNew(&(root->keyVector), bTree->keySize, bTree->order, 1, bTree->cmpFn, bTree->freeFn);
    if (0 != VectorInsertByPos(&(root->keyVector), e, 0))
    {
        free(root);
        return NULL;
    }
    //给孩子向量多分配一个节点空间，便于以后上溢分裂
    VectorNew(&(root->childVector), sizeof(BTREENODE *), (bTree->order + 1), 1, NULL, NULL);
    BTREENODE *dataNULL = NULL;
    if (0 != VectorInsertByPos(&(root->childVector), &dataNULL, 0))
    {
        free(root);
        return NULL;
    }
    if (0 != VectorInsertByPos(&(root->childVector), &dataNULL, 1))
    {
        free(root);
        return NULL;
    }
```

```
69      return root;
70  }
71
72  //BTree节点销毁
73  static void nodeDispose(BTREENODE *node)
74  {
75      VectorDispose(&(node->keyVector));
76      VectorDispose(&(node->childVector));
77      free(node);
78  }
79
80  //BTree初始化
81  void BTreeNew(BTREE *bTree, int order, int keySize, BTreeCmp *cmpFn, BTreeFree
    *freeFn)
82  {
83      assert(order > 2);
84      assert(keySize > 0);
85      assert(NULL != cmpFn);
86      bTree->root = NULL;
87      bTree->hot = NULL;
88      bTree->size = 0;
89      bTree->order = order;
90      bTree->keySize = keySize;
91      bTree->cmpFn = cmpFn;
92      bTree->freeFn = freeFn;
93  }
94
95  //BTree判空
96  int BTreeEmpty(BTREE *bTree)
97  {
98      return (bTree->size == 0);
99  }
100
101 //BTree规模
102 int BTreeSize(BTREE *bTree)
103 {
104     return bTree->size;
105 }
106
107 //BTree阶次
108 int BTreeOrder(BTREE *bTree)
109 {
110     return bTree->order;
111 }
112
113 static void addNode2Queue(void *elemAddr, void *outData)
114 {
115     QUEUE *q = (QUEUE *)outData;
116     if (NULL == q)
117     {
118         return ;
119     }
120     QueueEn(q, elemAddr);
121 }
122
123 //BTree销毁，按树的层次遍历销毁每个节点
124 void BTreeDispose(BTREE *bTree)
125 {
126     if (BTreeEmpty(bTree))
127     {
128         return ;
129     }
130     QUEUE nodeQueue;
131     QueueNew(&nodeQueue, sizeof(BTREENODE *), NULL);
132     BTREENODE *node = bTree->root;
133     QueueEn(&nodeQueue, &node);
134     while (!QueueEmpty(&nodeQueue))
135     {
136         QueueDe(&nodeQueue, &node);
137         if (NULL != node)
138         {
139             VectorTraverse(&(node->childVector), addNode2Queue, &nodeQueue);
140             nodeDispose(node);
```

```
141             }
142         }
143         QueueDispose(&nodeQueue);
144         bTree->root = NULL;
145         bTree->hot = NULL;
146         bTree->size = 0;
147     }
148
149     //BTree中查找关键码所在节点，hot指向当前节点的父节点
150     BTREENODE *BTreeSearch(BTREE *bTree, const void *e)
151     {
152         BTREENODE *node = bTree->root;
153         bTree->hot = NULL;
154         while (NULL != node) //逐层查找
155         {
156             //在当前节点中，查找不大于keyAddr的最大关键码
157             int rank = VectorSearch(&(node->keyVector), e, 1);
158             if ((rank < VectorSize(&(node->keyVector))) &&
                (VectorFind(&(node->keyVector), rank, e)))
159             {
160                 return node; //返回已查找到的节点
161             }
162             bTree->hot = node;
163             //转入对应子树，hot指向其父亲
164             node = *(BTREENODE **)VectorGetByPos(&(node->childVector), rank);
165         }
166         return NULL;
167     }
168
169     //处理上溢分裂
170     static void solveOverflow(BTREE *bTree, BTREENODE *node)
171     {
172         //递归基，当前节点并未上溢
173         if (bTree->order >= VectorSize(&(node->childVector)))
174         {
175             return ;
176         }
177         BTREENODE *newNode = nodeNew(bTree);
178         assert(NULL != newNode);
179         int s = bTree->order >> 1; //轴点
180         //节点node右侧bTree->order-s-1个孩子及关键码分裂为右侧节点newNode
181         int i = 0;
182         for (; i < bTree->order - s - 1; i ++)
183         {
184             BTREENODE *tmpNode = *(BTREENODE **)VectorGetByPos(&(node->childVector), s +
                1);
185             assert(0 == VectorInsertByPos(&(newNode->childVector), (void *)&tmpNode, i));
186             assert(0 == VectorRemoveByPosU(&(node->childVector), s + 1));
187             void *tmpKey = VectorGetByPos(&(node->keyVector), s + 1);
188             assert(0 == VectorInsertByPos(&(newNode->keyVector), tmpKey, i));
189             assert(0 == VectorRemoveByPosU(&(node->keyVector), s + 1));
190         }
191         //移动node最靠右的孩子
192         BTREENODE *tmpNode = *(BTREENODE **)VectorGetByPos(&(node->childVector), s + 1);
193         assert(0 == VectorInsertByPos(&(newNode->childVector), (void *)&tmpNode,
            bTree->order - s - 1));
194         assert(0 == VectorRemoveByPosU(&(node->childVector), s + 1));
195         //若newNode的孩子们非空，令他们的父节点统一指向newNode
196         if (NULL != *(BTREENODE **)VectorGetByPos(&(newNode->childVector), 0))
197         {
198             for (i = 0; i < bTree->order - s; i ++)
199             {
200                 BTREENODE *tmpChild = *(BTREENODE
                    **)VectorGetByPos(&(newNode->childVector), i);
201                 tmpChild->parent = newNode;
202             }
203         }
204         BTREENODE *p = node->parent;
205         //节点node如果为根节点
206         if (NULL == p)
207         {
208             p = nodeNew(bTree);
209             assert(NULL != p);
```

```
210             assert(0 == VectorInsertByPos(&(p->childVector), &node, 0));
211             bTree->root = p;
212             node->parent = p;
213         }
214         void *tmpKey = VectorGetByPos(&(node->keyVector), s);
215         //轴点在p中的秩
216         int rank = VectorSearch(&(p->keyVector), tmpKey, 1);
217         //轴点关键码上升
218         assert(0 == VectorInsertByPos(&(p->keyVector), tmpKey, rank));
219         assert(0 == VectorRemoveByPosU(&(node->keyVector), s));
220         //新节点newNode与父节点p互联
221         assert(0 == VectorInsertByPos(&(p->childVector), &newNode, rank + 1));
222         newNode->parent = p;
223         //上升一层，如有必要则继续分裂----至多递归O(logn)层
224         solveOverflow(bTree, p);
225     }
226
227     //BTree中插入关键码
228     int BTreeInsert(BTREE *bTree, const void *e)
229     {
230         //BTree为空时，新建根节点
231         if (NULL == bTree->root)
232         {
233             BTREENODE *root = rootNew(bTree, e);
234             if (NULL == root)
235             {
236                 return -1;
237             }
238             bTree->root = root;
239             bTree->size ++;
240             return 0;
241         }
242         if (NULL != BTreeSearch(bTree, e))
243         {
244             return -1;
245         }
246         BTREENODE *tarNode = bTree->hot; //待插入的目标节点
247         int rank = VectorSearch(&(tarNode->keyVector), e, 1); //查找合适的插入位置
248         if(0 != VectorInsertByPos(&(tarNode->keyVector), e, rank)) //将关键码插入对应位置
249         {
250             return -1;
251         }
252         BTREENODE *dataNULL = NULL;
253         if (0 != VectorInsertByPos(&(tarNode->childVector), &dataNULL, (rank + 1)))
            //创建一个空子树指针
254         {
255             return -1;
256         }
257         bTree->size ++; //更新全树规模
258         solveOverflow(bTree, bTree->hot); //如有必要，需做分裂
259         return 0;
260     }
261
262     //下溢旋转或合并
263     static void solveUnderflow(BTREE *bTree, BTREENODE *node)
264     {
265         //分裂下限
266         int lowNum = (bTree->order + 1) >> 1;
267         //递归基：当前节点并未下溢
268         if (lowNum <= VectorSize(&(node->childVector)))
269         {
270             return ;
271         }
272         BTREENODE *p = node->parent;
273         //递归基：已到根节点，没有孩子的下限
274         if (NULL == p)
275         {
276             //若作为根节点的node已不含关键码，却有唯一的非空左孩子，则其左孩子变为树根
277             //整树高度降低一层
278             if ((0 == VectorSize(&(node->keyVector))) \
279                 && (NULL != *(BTREENODE **)VectorGetByPos(&(node->childVector), 0)))
280             {
281                 BTREENODE *tmpChild = *(BTREENODE
```

```
                **)VectorGetByPos(&(node->childVector), 0);
282             bTree->root = tmpChild;
283             bTree->root->parent = NULL;
284             nodeDispose(node);
285         }
286         return ;
287     }
288     int rank = 0;
289     //确定node是p的第rank个孩子，此时node可能不含关键码，故不能通过关键码查找
290     while (node != *(BTREENODE **)VectorGetByPos(&(p->childVector), rank))
291     {
292         rank ++;
293     }
294     //case1: node有左兄弟，向左兄弟借关键码
295     if (0 < rank)
296     {
297         BTREENODE *ls = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank - 1);
298         //左兄弟足够"胖"，右旋完成当前层及其上所有层的下溢处理
299         int lsChildSize = VectorSize(&(ls->childVector));
300         if (lowNum < lsChildSize)
301         {
302             //p借出第rank-1个关键码给node（作为最小关键码）
303             assert(0 == VectorInsertByPos(&(node->keyVector),
                VectorGetByPos(&(p->keyVector), rank - 1), 0));
304             assert(0 == VectorRemoveByPosU(&(p->keyVector), rank - 1));
305             //ls的最大关键码转入p
306             int lsKeySize = VectorSize(&(ls->keyVector));
307             assert(0 == VectorInsertByPos(&(p->keyVector),
                VectorGetByPos(&(ls->keyVector), lsKeySize - 1), rank - 1));
308             assert(0 == VectorRemoveByPosU(&(ls->keyVector), lsKeySize - 1));
309             //ls的最右侧孩子过继给node
310             assert(0 == VectorInsertByPos(&(node->childVector),
                VectorGetByPos(&(ls->childVector), lsChildSize - 1), 0));
311             assert(0 == VectorRemoveByPosU(&(ls->childVector), lsChildSize - 1));
312             BTREENODE *tmpChild = *(BTREENODE
                **)VectorGetByPos(&(node->childVector), 0);
313             if (NULL != tmpChild)
314             {
315                 tmpChild->parent = node;
316             }
317             return ;
318         }
319     } //至此，左兄弟要么为空，要么太"瘦"
320     //case2: node有右兄弟，向右兄弟借关键码
321     if (VectorSize(&(p->childVector)) - 1 > rank)
322     {
323         BTREENODE *rs = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank + 1);
324         //右兄弟足够"胖"，左旋完成当前层及其上所有层的下溢处理
325         int rsChildSize = VectorSize(&(rs->childVector));
326         if (lowNum < rsChildSize)
327         {
328             //p借出第rank个关键码给node（作为最大关键码）
329             assert(0 == VectorInsertByPos(&(node->keyVector),
                VectorGetByPos(&(p->keyVector), rank), 0));
330             assert(0 == VectorRemoveByPosU(&(p->keyVector), rank));
331             //rs的最小关键码转入p
332             assert(0 == VectorInsertByPos(&(p->keyVector),
                VectorGetByPos(&(rs->keyVector), 0), rank));
333             assert(0 == VectorRemoveByPosU(&(rs->keyVector), 0));
334             //rs的最左侧孩子过继给node
335             int nodeChildSize = VectorSize(&(node->childVector));
336             assert(0 == VectorInsertByPos(&(node->childVector),
                VectorGetByPos(&(rs->childVector), 0), nodeChildSize));
337             assert(0 == VectorRemoveByPosU(&(rs->childVector), 0));
338             BTREENODE *tmpChild = *(BTREENODE
                **)VectorGetByPos(&(node->childVector), nodeChildSize);
339             if (NULL != tmpChild)
340             {
341                 tmpChild->parent = node;
342             }
343             return ;
344         }
345     } //至此，右兄弟要么为空，要么太"瘦"
```

```
346         //case3: 左、右兄弟要么为空（但不可能同时），要么都太"瘦"----合并
347         if (0 < rank) //与左兄弟合并
348         {
349             BTREENODE *ls = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank - 1);
350             //p的第rank-1个关键码转入ls
351             assert(0 == VectorInsertByPos(&(ls->keyVector),
                VectorGetByPos(&(p->keyVector), rank - 1), VectorSize(&(ls->keyVector))));
352             assert(0 == VectorRemoveByPosU(&(p->keyVector), rank - 1));
353             //node不再是p的第rank个孩子
354             assert(0 == VectorRemoveByPos(&(p->childVector), rank));
355             //node的最左侧的孩子过继给ls做最右侧孩子
356             assert(0 == VectorInsertByPos(&(ls->childVector),
                VectorGetByPos(&(node->childVector), 0), VectorSize(&(ls->childVector))));
357             assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
358             BTREENODE *tmpChild = *(BTREENODE **)VectorGetByPos(&(ls->childVector),
                VectorSize(&(ls->childVector)) - 1);
359             if (NULL != tmpChild)
360             {
361                 tmpChild->parent = ls;
362             }
363             //node剩余的关键码和孩子，一次转入ls
364             while (!VectorEmpty(&(node->keyVector)))
365             {
366                 assert(0 == VectorInsertByPos(&(ls->keyVector),
                    VectorGetByPos(&(node->keyVector), 0), VectorSize(&(ls->keyVector))));
367                 assert(0 == VectorRemoveByPosU(&(node->keyVector), 0));
368                 assert(0 == VectorInsertByPos(&(ls->childVector),
                    VectorGetByPos(&(node->childVector), 0), VectorSize(&(ls->childVector))));
369                 assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
370                 tmpChild = *(BTREENODE **)VectorGetByPos(&(ls->childVector),
                    VectorSize(&(ls->childVector)) - 1);
371                 if (NULL != tmpChild)
372                 {
373                     tmpChild->parent = ls;
374                 }
375             }
376             nodeDispose(node);
377         }
378         else //与右兄弟合并
379         {
380             BTREENODE *rs = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank + 1);
381             //p的第rank个关键码转入rs
382             assert(0 == VectorInsertByPos(&(rs->keyVector),
                VectorGetByPos(&(p->keyVector), rank), 0));
383             assert(0 == VectorRemoveByPosU(&(p->keyVector), rank));
384             //node不再是p的第rank个孩子
385             assert(0 == VectorRemoveByPos(&(p->childVector), rank));
386             //node的最右侧的孩子过继给rs做最左侧孩子
387             assert(0 == VectorInsertByPos(&(rs->childVector),
                VectorGetByPos(&(node->childVector), VectorSize(&(node->childVector)) - 1),
                0));
388             assert(0 == VectorRemoveByPosU(&(node->childVector),
                VectorSize(&(node->childVector)) - 1));
389             BTREENODE *tmpChild = *(BTREENODE **)VectorGetByPos(&(rs->childVector), 0);
390             if (NULL != tmpChild)
391             {
392                 tmpChild->parent = rs;
393             }
394             //node剩余的关键码和孩子，一次转入rs
395             while (!VectorEmpty(&(node->keyVector)))
396             {
397                 assert(0 == VectorInsertByPos(&(rs->keyVector),
                    VectorGetByPos(&(node->keyVector), VectorSize(&(node->keyVector)) - 1),
                    0));
398                 assert(0 == VectorRemoveByPosU(&(node->keyVector), 0));
399                 assert(0 == VectorInsertByPos(&(rs->childVector),
                    VectorGetByPos(&(node->childVector), VectorSize(&(node->childVector)) -
                    1), 0));
400                 assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
401                 tmpChild = *(BTREENODE **)VectorGetByPos(&(rs->childVector), 0);
402                 if (NULL != tmpChild)
403                 {
404                     tmpChild->parent = rs;
```

```
            }
        }
        nodeDispose(node);
    }
    //上升一层，如有必要则继续合并----至多递归O(logn)层
    solveUnderflow(bTree, p);
}

//BTree中删除关键码
int BTreeRemove(BTREE *bTree, void *e)
{
    BTREENODE *tarNode = BTreeSearch(bTree, e);
    //目标关键码不存在，删除失败
    if (NULL == tarNode)
    {
        return -1;
    }
    //确定目标关键码在节点tarNode上的秩，肯定合法
    int rank = VectorSearch(&(tarNode->keyVector), e, 1);
    //若tarNode非最底层节点，则查找其后继
    if (NULL != *(BTREENODE **)VectorGetByPos(&(tarNode->childVector), 0))
    {
        //在右子树中一直向左就可找出keyAddr的后继
        BTREENODE *u = *(BTREENODE **)VectorGetByPos(&(tarNode->childVector), rank + 1);
        while (NULL != *(BTREENODE **)VectorGetByPos(&(u->childVector), 0))
        {
            u = *(BTREENODE **)VectorGetByPos(&(u->childVector), 0);
        }
        //将keyAddr的后继者与之交换
        if (0 != VectorSwap(&(tarNode->keyVector), &(u->keyVector), rank, 0))
        {
            return -1;
        }
        tarNode = u;
        rank = 0;
    }
    if (0 != VectorRemoveByPos(&(tarNode->keyVector), rank))
    {
        return -1;
    }
    if (0 != VectorRemoveByPos(&(tarNode->childVector), rank + 1))
    {
        return -1;
    }
    bTree->size --;
    solveUnderflow(bTree, tarNode); //如有必要，需做旋转或合并
    return 0;
}

//BTree层序遍历
void BTreeTravLevel(BTREE *bTree, BTreeTraverseOp *traverseOpFn, void *outData)
{
    if ((NULL == traverseOpFn) || (bTree->size <= 0))
    {
        return ;
    }
    QUEUE nodeQueue;
    QueueNew(&nodeQueue, sizeof(BTREENODE *), NULL);
    BTREENODE *node = bTree->root;
    QueueEn(&nodeQueue, &node);
    while (!QueueEmpty(&nodeQueue))
    {
        QueueDe(&nodeQueue, &node);
        if (NULL != node)
        {
            VectorTraverse(&(node->childVector), addNode2Queue, &nodeQueue);
            VectorTraverse(&(node->keyVector), traverseOpFn, outData);
        }
    }
    QueueDispose(&nodeQueue);
}
```