

```

1  /* BTree.c */
2  #include <stdlib.h>
3  #include <malloc.h>
4  #include <assert.h>
5  #include <string.h>
6  #include "BTree.h"
7  #include "LinkQueue.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
24 //初始化非根节点
25 static BTREENODE *nodeNew(BTREE *bTree)
26 {
27     BTREENODE *newNode = (BTREENODE *)malloc(sizeof(BTREENODE));
28     if (NULL == newNode)
29     {
30         return NULL;
31     }
32     newNode->parent = NULL;
33     //给关键词向量多分配一个节点空间, 便于以后上溢分裂
34     VectorNew(&(newNode->keyVector), bTree->keySize, bTree->order, 1, bTree->cmpFn,
bTree->freeFn);
35     //给孩子向量多分配一个节点空间, 便于以后上溢分裂
36     VectorNew(&(newNode->childVector), sizeof(BTREENODE *), (bTree->order + 1), 1,
NULL, NULL);
37     return newNode;
38 }
39
40 //初始化根节点
41 static BTREENODE *rootNew(BTREE *bTree, const void *e)
42 {
43     BTREENODE *root = (BTREENODE *)malloc(sizeof(BTREENODE));
44     if (NULL == root)
45     {
46         return NULL;
47     }
48     root->parent = NULL;
49     //给关键词向量多分配一个节点空间, 便于以后上溢分裂
50     VectorNew(&(root->keyVector), bTree->keySize, bTree->order, 1, bTree->cmpFn,
bTree->freeFn);
51     if (0 != VectorInsertByPos(&(root->keyVector), e, 0))
52     {
53         free(root);
54         return NULL;
55     }
56     //给孩子向量多分配一个节点空间, 便于以后上溢分裂
57     VectorNew(&(root->childVector), sizeof(BTREENODE *), (bTree->order + 1), 1,
NULL, NULL);
58     BTREENODE *dataNULL = NULL;
59     if (0 != VectorInsertByPos(&(root->childVector), &dataNULL, 0))
60     {
61         free(root);
62         return NULL;
63     }
64     if (0 != VectorInsertByPos(&(root->childVector), &dataNULL, 1))
65     {
66         free(root);
67         return NULL;
68     }

```

```

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 +
185             1);
186         assert(0 == VectorInsertByPos(&(newNode->childVector), (void *)&tmpNode, i));
187         assert(0 == VectorRemoveByPosU(&(node->childVector), s + 1));
188         void *tmpKey = VectorGetByPos(&(node->keyVector), s + 1);
189         assert(0 == VectorInsertByPos(&(newNode->keyVector), tmpKey, i));
190         assert(0 == VectorRemoveByPosU(&(node->keyVector), s + 1));
191     }
192     //移动node最靠右的孩子
193     BTREENODE *tmpNode = *(BTREENODE **)VectorGetByPos(&(node->childVector), s + 1);
194     assert(0 == VectorInsertByPos(&(newNode->childVector), (void *)&tmpNode,
195         bTree->order - s - 1));
196     assert(0 == VectorRemoveByPosU(&(node->childVector), s + 1));
197     //若newNode的孩子非空，令他们的父节点统一指向newNode
198     if (NULL != *(BTREENODE **)VectorGetByPos(&(newNode->childVector), 0))
199     {
200         for (i = 0; i < bTree->order - s; i++)
201         {
202             BTREENODE *tmpChild = *(BTREENODE
203                 **)VectorGetByPos(&(newNode->childVector), i);
204             tmpChild->parent = newNode;
205         }
206     }
207     BTREENODE *p = node->parent;
208     //节点node如果为根节点
209     if (NULL == p)
210     {
211         p = nodeNew(bTree);
212         assert(NULL != p);
213     }
214     p->childVector[s] = newNode;
215     p->keyVector[s] = node->keyVector[bTree->order - s - 1];
216     p->parent = NULL;
217     node->parent = p;
218     node->childVector[bTree->order - s - 1] = NULL;
219     node->keyVector[bTree->order - s - 1] = NULL;
220     node->parent = p;
221 }
222
223 //BTree插入
224 void BTreeInsert(BTREE *bTree, const void *e)
225 {
226     BTREENODE *node = BTreeSearch(bTree, e);
227     if (NULL == node)
228     {
229         node = nodeNew(bTree);
230         node->parent = bTree->root;
231         node->keyVector[bTree->order - 1] = e;
232         bTree->root->childVector[bTree->order - 1] = node;
233         bTree->size++;
234     }
235     else
236     {
237         //插入失败
238         return ;
239     }
240 }
241
242 //BTree删除
243 void BTreeDelete(BTREE *bTree, const void *e)
244 {
245     BTREENODE *node = BTreeSearch(bTree, e);
246     if (NULL == node)
247     {
248         return ;
249     }
250     else
251     {
252         //删除成功
253         //删除节点node
254         BTREENODE *parent = node->parent;
255         int i = VectorFind(&(parent->keyVector), node->keyVector[0], 0);
256         parent->childVector[i] = node->childVector[0];
257         parent->keyVector[i] = node->keyVector[0];
258         node->parent = NULL;
259         node->keyVector[0] = NULL;
260         node->childVector[0] = NULL;
261         bTree->size--;
262     }
263 }
264
265 //BTree遍历
266 void BTreeTraverse(BTREE *bTree)
267 {
268     BTREENODE *node = bTree->root;
269     while (node)
270     {
271         BTreeTraverse(node->childVector[0]);
272         printf("%d ", node->keyVector[0]);
273         node = node->parent;
274     }
275 }
276
277 //BTree初始化
278 void BTreeInit(BTREE *bTree)
279 {
280     bTree->root = nodeNew(bTree);
281     bTree->size = 0;
282     bTree->order = 4;
283     bTree->keyVector = VectorNew(bTree->order);
284     bTree->childVector = VectorNew(bTree->order);
285     bTree->parentVector = VectorNew(bTree->order);
286     bTree->parentVector[0] = NULL;
287     bTree->root->parent = NULL;
288     bTree->root->keyVector[0] = NULL;
289     bTree->root->childVector[0] = NULL;
290 }
291
292 //BTree销毁
293 void BTreeDestroy(BTREE *bTree)
294 {
295     QueueDispose(&bTree->parentQueue);
296     QueueDispose(&bTree->childQueue);
297     QueueDispose(&bTree->keyQueue);
298     QueueDispose(&bTree->rootQueue);
299     QueueDispose(&bTree->nodeQueue);
300     QueueDispose(&bTree->hotQueue);
301     QueueDispose(&bTree->tmpQueue);
302     QueueDispose(&bTree->tmpKeyQueue);
303     QueueDispose(&bTree->tmpChildQueue);
304     QueueDispose(&bTree->tmpParentQueue);
305     QueueDispose(&bTree->tmpKeyChildQueue);
306     QueueDispose(&bTree->tmpKeyParentQueue);
307     QueueDispose(&bTree->tmpChildParentQueue);
308     QueueDispose(&bTree->tmpKeyChildParentQueue);
309     QueueDispose(&bTree->tmpKeyChildParentChildQueue);
310     QueueDispose(&bTree->tmpKeyChildParentParentQueue);
311     QueueDispose(&bTree->tmpKeyChildParentChildParentQueue);
312     QueueDispose(&bTree->tmpKeyChildParentParentChildQueue);
313     QueueDispose(&bTree->tmpKeyChildParentParentChildParentQueue);
314     QueueDispose(&bTree->tmpKeyChildParentParentChildParentChildQueue);
315     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentQueue);
316     QueueDispose(&bTree->tmpKeyChildParentParentChildParentChildParentQueue);
317     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildQueue);
318     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentQueue);
319     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildQueue);
320     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentParentQueue);
321     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentQueue);
322     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentChildQueue);
323     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentQueue);
324     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentChildParentQueue);
325     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentQueue);
326     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentChildQueue);
327     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentQueue);
328     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentChildParentQueue);
329     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildQueue);
330     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentParentQueue);
331     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentQueue);
332     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildQueue);
333     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentParentQueue);
334     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentQueue);
335     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildQueue);
336     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentParentQueue);
337     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentQueue);
338     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildQueue);
339     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentParentQueue);
340     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentQueue);
341     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildQueue);
342     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentParentQueue);
343     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentQueue);
344     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildQueue);
345     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentParentQueue);
346     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentQueue);
347     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentChildQueue);
348     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentParentQueue);
349     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentQueue);
350     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentChildQueue);
351     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentParentQueue);
352     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentQueue);
353     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentChildQueue);
354     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentParentQueue);
355     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentChildParentQueue);
356     QueueDispose(&bTree->tmpKeyChildParentParentChildParentParentChildParentChildParentParentChildParentParentChildParentChildParentChildParentChildParentChild
```

```

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)))
254     //创建一个空子树指针
255     {
256         return -1;
257     }
258     bTree->size ++; //更新全树规模
259     solveOverflow(bTree, bTree->hot); //如有必要，需做分裂
260     return 0;
261 }
262
263 //下溢旋转或合并
264 static void solveUnderflow(BTREE *bTree, BTREENODE *node)
265 {
266     //分裂下限
267     int lowNum = (bTree->order + 1) >> 1;
268     //递归基：当前节点并未下溢
269     if (lowNum <= VectorSize(&(node->childVector)))
270     {
271         return ;
272     }
273     BTREENODE *p = node->parent;
274     //递归基：已到根节点，没有孩子的下限
275     if (NULL == p)
276     {
277         //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
278         //整树高度降低一层
279         if ((0 == VectorSize(&(node->keyVector))) \
280             && (NULL != *(BTREENODE **)VectorGetByPos(&(node->childVector), 0)))
281         {
282             BTREENODE *tmpChild = *(BTREENODE

```

```

282         **)VectorGetByPos (&(node->childVector), 0);
283         bTree->root = tmpChild;
284         bTree->root->parent = NULL;
285         nodeDispose(node);
286     }
287     return ;
288 }
289 int rank = 0;
290 //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
291 while (node != *(BTREENODE **)VectorGetByPos (&(p->childVector), rank))
292 {
293     rank ++;
294 }
295 //case1: node有左兄弟，向左兄弟借关键码
296 if (0 < rank)
297 {
298     BTREENODE *ls = *(BTREENODE **)VectorGetByPos (&(p->childVector), rank - 1);
299     //左兄弟足够“胖”，右旋完成当前层及其上所有层的下溢处理
300     int lsChildSize = VectorSize (&(ls->childVector));
301     if (lowNum < lsChildSize)
302     {
303         //p借出第rank-1个关键码给node（作为最小关键码）
304         assert(0 == VectorInsertByPos (&(node->keyVector),
305             VectorGetByPos (&(p->keyVector), rank - 1), 0));
306         assert(0 == VectorRemoveByPosU (&(p->keyVector), rank - 1));
307         //ls的最大关键码转入p
308         int lsKeySize = VectorSize (&(ls->keyVector));
309         assert(0 == VectorInsertByPos (&(p->keyVector),
310             VectorGetByPos (&(ls->keyVector), lsKeySize - 1), rank - 1));
311         assert(0 == VectorRemoveByPosU (&(ls->keyVector), lsKeySize - 1));
312         //ls的最右侧孩子过继给node
313         assert(0 == VectorInsertByPos (&(node->childVector),
314             VectorGetByPos (&(ls->childVector), lsChildSize - 1), 0));
315         assert(0 == VectorRemoveByPosU (&(ls->childVector), lsChildSize - 1));
316         BTREENODE *tmpChild = *(BTREENODE **)
317             VectorGetByPos (&(node->childVector), 0);
318         if (NULL != tmpChild)
319         {
320             tmpChild->parent = node;
321         }
322         return ;
323     }
324 } //至此，左兄弟要么为空，要么太“瘦”
325 //case2: node有右兄弟，向右兄弟借关键码
326 if (VectorSize (&(p->childVector)) - 1 > rank)
327 {
328     BTREENODE *rs = *(BTREENODE **)VectorGetByPos (&(p->childVector), rank + 1);
329     //右兄弟足够“胖”，左旋完成当前层及其上所有层的下溢处理
330     int rsChildSize = VectorSize (&(rs->childVector));
331     if (lowNum < rsChildSize)
332     {
333         //p借出第rank个关键码给node（作为最大关键码）
334         assert(0 == VectorInsertByPos (&(node->keyVector),
335             VectorGetByPos (&(p->keyVector), rank), 0));
336         assert(0 == VectorRemoveByPosU (&(p->keyVector), rank));
337         //rs的最小关键码转入p
338         assert(0 == VectorInsertByPos (&(p->keyVector),
339             VectorGetByPos (&(rs->keyVector), 0), rank));
340         assert(0 == VectorRemoveByPosU (&(rs->keyVector), 0));
341         //rs的最左侧孩子过继给node
342         int nodeChildSize = VectorSize (&(node->childVector));
343         assert(0 == VectorInsertByPos (&(node->childVector),
344             VectorGetByPos (&(rs->childVector), 0), nodeChildSize));
345         assert(0 == VectorRemoveByPosU (&(rs->childVector), 0));
346         BTREENODE *tmpChild = *(BTREENODE **)
347             VectorGetByPos (&(node->childVector), nodeChildSize);
348         if (NULL != tmpChild)
349         {
350             tmpChild->parent = node;
351         }
352         return ;
353     }
354 } //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),
352     VectorGetByPos(&(p->keyVector), rank - 1), VectorSize(&(ls->keyVector))));
353     assert(0 == VectorRemoveByPosU(&(p->keyVector), rank - 1));
354     //node不再是p的第rank个孩子
355     assert(0 == VectorRemoveByPos(&(p->childVector), rank));
356     //node的最左侧的孩子过继给ls做最右侧孩子
357     assert(0 == VectorInsertByPos(&(ls->childVector),
358     VectorGetByPos(&(node->childVector), 0), VectorSize(&(ls->childVector))));
359     assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
360     BTREENODE *tmpChild = *(BTREENODE **)VectorGetByPos(&(ls->childVector),
361     VectorSize(&(ls->childVector)) - 1);
362     if (NULL != tmpChild)
363     {
364         tmpChild->parent = ls;
365     }
366     //node剩余的关键码和孩子，一次转入ls
367     while (!VectorEmpty(&(node->keyVector)))
368     {
369         assert(0 == VectorInsertByPos(&(ls->keyVector),
370         VectorGetByPos(&(node->keyVector), 0), VectorSize(&(ls->keyVector))));
371         assert(0 == VectorRemoveByPosU(&(node->keyVector), 0));
372         assert(0 == VectorInsertByPos(&(ls->childVector),
373         VectorGetByPos(&(node->childVector), 0), VectorSize(&(ls->childVector))));
374         assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
375         tmpChild = *(BTREENODE **)VectorGetByPos(&(ls->childVector),
376         VectorSize(&(ls->childVector)) - 1);
377         if (NULL != tmpChild)
378         {
379             tmpChild->parent = ls;
380         }
381     }
382     nodeDispose(node);
383 }
384 else //与右兄弟合并
385 {
386     BTREENODE *rs = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank + 1);
387     //p的第rank个关键码转入rs
388     assert(0 == VectorInsertByPos(&(rs->keyVector),
389     VectorGetByPos(&(p->keyVector), rank), 0));
390     assert(0 == VectorRemoveByPosU(&(p->keyVector), rank));
391     //node不再是p的第rank个孩子
392     assert(0 == VectorRemoveByPos(&(p->childVector), rank));
393     //node的最右侧的孩子过继给rs做最左侧孩子
394     assert(0 == VectorInsertByPos(&(rs->childVector),
395     VectorGetByPos(&(node->childVector), VectorSize(&(node->childVector)) - 1),
396     0));
397     assert(0 == VectorRemoveByPosU(&(node->childVector),
398     VectorSize(&(node->childVector)) - 1));
399     BTREENODE *tmpChild = *(BTREENODE **)VectorGetByPos(&(rs->childVector), 0);
400     if (NULL != tmpChild)
401     {
402         tmpChild->parent = rs;
403     }
404     //node剩余的关键码和孩子，一次转入rs
405     while (!VectorEmpty(&(node->keyVector)))
406     {
407         assert(0 == VectorInsertByPos(&(rs->keyVector),
408         VectorGetByPos(&(node->keyVector), VectorSize(&(node->keyVector)) - 1),
409         0));
410         assert(0 == VectorRemoveByPosU(&(node->keyVector), 0));
411         assert(0 == VectorInsertByPos(&(rs->childVector),
412         VectorGetByPos(&(node->childVector), VectorSize(&(node->childVector)) -
413         1), 0));
414         assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
415         tmpChild = *(BTREENODE **)VectorGetByPos(&(rs->childVector), 0);
416         if (NULL != tmpChild)
417         {
418             tmpChild->parent = rs;
419         }
420     }
421     nodeDispose(node);
422 }

```

```

405     }
406 }
407 nodeDispose(node);
408 }
409 //上升一层，如有必要则继续合并----至多递归O(logn)层
410 solveUnderflow(bTree, p);
411 }
412
413 //BTree中删除关键码
414 int BTreeRemove(BTREE *bTree, void *e)
415 {
416     BTREENODE *tarNode = BTreeSearch(bTree, e);
417     //目标关键码不存在，删除失败
418     if (NULL == tarNode)
419     {
420         return -1;
421     }
422     //确定目标关键码在节点tarNode上的秩，肯定合法
423     int rank = VectorSearch(&(tarNode->keyVector), e, 1);
424     //若tarNode非最底层节点，则查找其后继
425     if (NULL != *(BTREENODE **)VectorGetByPos(&(tarNode->childVector), 0))
426     {
427         //在右子树中一直向左就可找出keyAddr的后继
428         BTREENODE *u = *(BTREENODE **)VectorGetByPos(&(tarNode->childVector), rank +
429             1);
430         while (NULL != *(BTREENODE **)VectorGetByPos(&(u->childVector), 0))
431         {
432             u = *(BTREENODE **)VectorGetByPos(&(u->childVector), 0);
433         }
434         //将keyAddr的后继者与之交换
435         if (0 != VectorSwap(&(tarNode->keyVector), &(u->keyVector), rank, 0))
436         {
437             return -1;
438         }
439         tarNode = u;
440         rank = 0;
441     }
442     if (0 != VectorRemoveByPos(&(tarNode->keyVector), rank))
443     {
444         return -1;
445     }
446     if (0 != VectorRemoveByPos(&(tarNode->childVector), rank + 1))
447     {
448         return -1;
449     }
450     bTree->size--;
451     solveUnderflow(bTree, tarNode); //如有必要，需做旋转或合并
452     return 0;
453 }
454
455 //BTree层序遍历
456 void BTreeTravLevel(BTREE *bTree, BTreeTraverseOp *traverseOpFn, void *outData)
457 {
458     if ((NULL == traverseOpFn) || (bTree->size <= 0))
459     {
460         return ;
461     }
462     QUEUE nodeQueue;
463     QueueNew(&nodeQueue, sizeof(BTREENODE *), NULL);
464     BTREENODE *node = bTree->root;
465     QueueEn(&nodeQueue, &node);
466     while (!QueueEmpty(&nodeQueue))
467     {
468         QueueDe(&nodeQueue, &node);
469         if (NULL != node)
470         {
471             VectorTraverse(&(node->childVector), addNode2Queue, &nodeQueue);
472             VectorTraverse(&(node->keyVector), traverseOpFn, outData);
473         }
474     }
475     QueueDispose(&nodeQueue);
476 }

```