```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;
    }
    return root;
}

//BTree节点销毁
static void nodeDispose(BTREENODE *node)
{
    VectorDispose(&(node->keyVector));
    VectorDispose(&(node->childVector));
    free(node);
}

//BTree初始化
void BTreeNew(BTREE *bTree, int order, int keySize, BTreeCmp *cmpFn, BTreeFree *freeFn)
{
    assert(order > 2);
    assert(keySize > 0);
    assert(NULL != cmpFn);
    bTree->root = NULL;
    bTree->hot = NULL;
    bTree->size = 0;
    bTree->order = order;
    bTree->keySize = keySize;
    bTree->cmpFn = cmpFn;
    bTree->freeFn = freeFn;
}

//BTree判空
int BTreeEmpty(BTREE *bTree)
{
    return (bTree->size == 0);
}

//BTree规模
int BTreeSize(BTREE *bTree)
{
    return bTree->size;
}

//BTree阶次
int BTreeOrder(BTREE *bTree)
{
    return bTree->order;
}

static void addNode2Queue(void *elemAddr, void *outData)
{
    QUEUE *q = (QUEUE *)outData;
    if (NULL == q)
    {
        return ;
    }
    QueueEn(q, elemAddr);
}

//BTree销毁，按树的层次遍历销毁每个节点
void BTreeDispose(BTREE *bTree)
{
    if (BTreeEmpty(bTree))
    {
        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);
            nodeDispose(node);
        }
    }
    QueueDispose(&nodeQueue);
    bTree->root = NULL;
    bTree->hot = NULL;
    bTree->size = 0;
}

//BTree中查找关键码所在节点，hot指向当前节点的父节点
BTREENODE *BTreeSearch(BTREE *bTree, const void *e)
{
    BTREENODE *node = bTree->root;
    bTree->hot = NULL;
    while (NULL != node) //逐层查找
    {
        //在当前节点中，查找不大于keyAddr的最大关键码
        int rank = VectorSearch(&(node->keyVector), e, 1);
        if ((rank < VectorSize(&(node->keyVector))) &&
        (VectorFind(&(node->keyVector), rank, e)))
        {
            return node; //返回已查找到的节点
        }
        bTree->hot = node;
        //转入对应子树，hot指向其父亲
        node = *(BTREENODE **)VectorGetByPos(&(node->childVector), rank);
    }
    return NULL;
}

//处理上溢分裂
static void solveOverflow(BTREE *bTree, BTREENODE *node)
{
    //递归基，当前节点并未上溢
    if (bTree->order >= VectorSize(&(node->childVector)))
    {
        return ;
    }
    BTREENODE *newNode = nodeNew(bTree);
    assert(NULL != newNode);
    int s = bTree->order >> 1; //轴点
    //节点node右侧bTree->order-s-1个孩子及关键码分裂为右侧节点newNode
    int i = 0;
    for (; i < bTree->order - s - 1; i ++)
    {
        BTREENODE *tmpNode = *(BTREENODE **)VectorGetByPos(&(node->childVector), s +
        1);
        assert(0 == VectorInsertByPos(&(newNode->childVector), (void *)&tmpNode, i));
        assert(0 == VectorRemoveByPosU(&(node->childVector), s + 1));
        void *tmpKey = VectorGetByPos(&(node->keyVector), s + 1);
        assert(0 == VectorInsertByPos(&(newNode->keyVector), tmpKey, i));
        assert(0 == VectorRemoveByPosU(&(node->keyVector), s + 1));
    }
    //移动node最靠右的孩子
    BTREENODE *tmpNode = *(BTREENODE **)VectorGetByPos(&(node->childVector), s + 1);
    assert(0 == VectorInsertByPos(&(newNode->childVector), (void *)&tmpNode,
    bTree->order - s - 1));
    assert(0 == VectorRemoveByPosU(&(node->childVector), s + 1));
    //若newNode的孩子们非空，令他们的父节点统一指向newNode
    if (NULL != *(BTREENODE **)VectorGetByPos(&(newNode->childVector), 0))
    {
        for (i = 0; i < bTree->order - s; i ++)
        {
            BTREENODE *tmpChild = *(BTREENODE
            **)VectorGetByPos(&(newNode->childVector), i);
            tmpChild->parent = newNode;
        }
    }
    BTREENODE *p = node->parent;
    //节点node如果为根节点
    if (NULL == p)
    {
        p = nodeNew(bTree);
        assert(NULL != p);
        assert(0 == VectorInsertByPos(&(p->childVector), &node, 0));
        bTree->root = p;
        node->parent = p;
    }
    void *tmpKey = VectorGetByPos(&(node->keyVector), s);
    //轴点在p中的秩
    int rank = VectorSearch(&(p->keyVector), tmpKey, 1);
    //轴点关键码上升
    assert(0 == VectorInsertByPos(&(p->keyVector), tmpKey, rank));
    assert(0 == VectorRemoveByPosU(&(node->keyVector), s));
    //新节点newNode与父节点p互联
    assert(0 == VectorInsertByPos(&(p->childVector), &newNode, rank + 1));
    newNode->parent = p;
    //上升一层，如有必要则继续分裂----至多递归O(logn)层
    solveOverflow(bTree, p);
}

//BTree中插入关键码
int BTreeInsert(BTREE *bTree, const void *e)
{
    //BTree为空时，新建根节点
    if (NULL == bTree->root)
    {
        BTREENODE *root = rootNew(bTree, e);
        if (NULL == root)
        {
            return -1;
        }
        bTree->root = root;
        bTree->size ++;
        return 0;
    }
    if (NULL != BTreeSearch(bTree, e))
    {
        return -1;
    }
    BTREENODE *tarNode = bTree->hot; //待插入的目标节点
    int rank = VectorSearch(&(tarNode->keyVector), e, 1); //查找合适的插入位置
    if(0 != VectorInsertByPos(&(tarNode->keyVector), e, rank)) //将关键码插入对应位置
    {
        return -1;
    }
    BTREENODE *dataNULL = NULL;
    if (0 != VectorInsertByPos(&(tarNode->childVector), &dataNULL, (rank + 1)))
    //创建一个空子树指针
    {
        return -1;
    }
    bTree->size ++; //更新全树规模
    solveOverflow(bTree, bTree->hot); //如有必要，需做分裂
    return 0;
}

//下溢旋转或合并
static void solveUnderflow(BTREE *bTree, BTREENODE *node)
{
    //分裂下限
    int lowNum = (bTree->order + 1) >> 1;
    //递归基：当前节点并未下溢
    if (lowNum <= VectorSize(&(node->childVector)))
    {
        return ;
    }
    BTREENODE *p = node->parent;
    //递归基：已到根节点，没有孩子的下限
    if (NULL == p)
    {
        //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
        //整树高度降低一层
        if ((0 == VectorSize(&(node->keyVector))) \
            && (NULL != *(BTREENODE **)VectorGetByPos(&(node->childVector), 0)))
        {
            BTREENODE *tmpChild = *(BTREENODE
            **)VectorGetByPos(&(node->childVector), 0);
            bTree->root = tmpChild;
            bTree->root->parent = NULL;
            nodeDispose(node);
        }
        return ;
    }
    int rank = 0;
    //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
    while (node != *(BTREENODE **)VectorGetByPos(&(p->childVector), rank))
    {
        rank ++;
    }
    //case1: node有左兄弟，向左兄弟借关键码
    if (0 < rank)
    {
        BTREENODE *ls = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank - 1);
        //左兄弟足够“胖”，右旋完成当前层及其上所有层的下溢处理
        int lsChildSize = VectorSize(&(ls->childVector));
        if (lowNum < lsChildSize)
        {
            //p借出第rank-1个关键码给node（作为最小关键码）
            assert(0 == VectorInsertByPos(&(node->keyVector),
            VectorGetByPos(&(p->keyVector), rank - 1), 0));
            assert(0 == VectorRemoveByPosU(&(p->keyVector), rank - 1));
            //ls的最大关键码转入p
            int lsKeySize = VectorSize(&(ls->keyVector));
            assert(0 == VectorInsertByPos(&(p->keyVector),
            VectorGetByPos(&(ls->keyVector), lsKeySize - 1), rank - 1));
            assert(0 == VectorRemoveByPosU(&(ls->keyVector), lsKeySize - 1));
            //ls的最右侧孩子过继给node
            assert(0 == VectorInsertByPos(&(node->childVector),
            VectorGetByPos(&(ls->childVector), lsChildSize - 1), 0));
            assert(0 == VectorRemoveByPosU(&(ls->childVector), lsChildSize - 1));
            BTREENODE *tmpChild = *(BTREENODE
            **)VectorGetByPos(&(node->childVector), 0);
            if (NULL != tmpChild)
            {
                tmpChild->parent = node;
            }
            return ;
        }
    } //至此，左兄弟要么为空，要么太“瘦”
    //case2: node有右兄弟，向右兄弟借关键码
    if (VectorSize(&(p->childVector)) - 1 > rank)
    {
        BTREENODE *rs = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank + 1);
        //右兄弟足够“胖”，左旋完成当前层及其上所有层的下溢处理
        int rsChildSize = VectorSize(&(rs->childVector));
        if (lowNum < rsChildSize)
        {
            //p借出第rank个关键码给node（作为最大关键码）
            assert(0 == VectorInsertByPos(&(node->keyVector),
            VectorGetByPos(&(p->keyVector), rank), 0));
            assert(0 == VectorRemoveByPosU(&(p->keyVector), rank));
            //rs的最小关键码转入p
            assert(0 == VectorInsertByPos(&(p->keyVector),
            VectorGetByPos(&(rs->keyVector), 0), rank));
            assert(0 == VectorRemoveByPosU(&(rs->keyVector), 0));
            //rs的最左侧孩子过继给node
            int nodeChildSize = VectorSize(&(node->childVector));
            assert(0 == VectorInsertByPos(&(node->childVector),
            VectorGetByPos(&(rs->childVector), 0), nodeChildSize));
            assert(0 == VectorRemoveByPosU(&(rs->childVector), 0));
            BTREENODE *tmpChild = *(BTREENODE
            **)VectorGetByPos(&(node->childVector), nodeChildSize);
            if (NULL != tmpChild)
            {
                tmpChild->parent = node;
            }
            return ;
        }
    } //至此，右兄弟要么为空，要么太“瘦”
    //case3：左、右兄弟要么为空（但不可能同时），要么都太“瘦”----合并
    if (0 < rank) //与左兄弟合并
    {
        BTREENODE *ls = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank - 1);
        //p的第rank-1个关键码转入ls
        assert(0 == VectorInsertByPos(&(ls->keyVector), VectorGetByPos(&(p->keyVector), rank - 1), VectorSize(&(ls->keyVector))));
        assert(0 == VectorRemoveByPosU(&(p->keyVector), rank - 1));
        //node不再是p的第rank个孩子
        assert(0 == VectorRemoveByPos(&(p->childVector), rank));
        //node的最左侧的孩子过继给ls做最右侧孩子
        assert(0 == VectorInsertByPos(&(ls->childVector), VectorGetByPos(&(node->childVector), 0), VectorSize(&(ls->childVector))));
        assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
        BTREENODE *tmpChild = *(BTREENODE **)VectorGetByPos(&(ls->childVector), VectorSize(&(ls->childVector)) - 1);
        if (NULL != tmpChild)
        {
            tmpChild->parent = ls;
        }
        //node剩余的关键码和孩子，一次转入ls
        while (!VectorEmpty(&(node->keyVector)))
        {
            assert(0 == VectorInsertByPos(&(ls->keyVector), VectorGetByPos(&(node->keyVector), 0), VectorSize(&(ls->keyVector))));
            assert(0 == VectorRemoveByPosU(&(node->keyVector), 0));
            assert(0 == VectorInsertByPos(&(ls->childVector), VectorGetByPos(&(node->childVector), 0), VectorSize(&(ls->childVector))));
            assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
            tmpChild = *(BTREENODE **)VectorGetByPos(&(ls->childVector), VectorSize(&(ls->childVector)) - 1);
            if (NULL != tmpChild)
            {
                tmpChild->parent = ls;
            }
        }
        nodeDispose(node);
    }
    else //与右兄弟合并
    {
        BTREENODE *rs = *(BTREENODE **)VectorGetByPos(&(p->childVector), rank + 1);
        //p的第rank个关键码转入rs
        assert(0 == VectorInsertByPos(&(rs->keyVector), VectorGetByPos(&(p->keyVector), rank), 0));
        assert(0 == VectorRemoveByPosU(&(p->keyVector), rank));
        //node不再是p的第rank个孩子
        assert(0 == VectorRemoveByPos(&(p->childVector), rank));
        //node的最右侧的孩子过继给rs做最左侧孩子
        assert(0 == VectorInsertByPos(&(rs->childVector), VectorGetByPos(&(node->childVector), VectorSize(&(node->childVector)) - 1), 0));
        assert(0 == VectorRemoveByPosU(&(node->childVector), VectorSize(&(node->childVector)) - 1));
        BTREENODE *tmpChild = *(BTREENODE **)VectorGetByPos(&(rs->childVector), 0);
        if (NULL != tmpChild)
        {
            tmpChild->parent = rs;
        }
        //node剩余的关键码和孩子，一次转入rs
        while (!VectorEmpty(&(node->keyVector)))
        {
            assert(0 == VectorInsertByPos(&(rs->keyVector), VectorGetByPos(&(node->keyVector), VectorSize(&(node->keyVector)) - 1), 0));
            assert(0 == VectorRemoveByPosU(&(node->keyVector), 0));
            assert(0 == VectorInsertByPos(&(rs->childVector), VectorGetByPos(&(node->childVector), VectorSize(&(node->childVector)) - 1), 0));
            assert(0 == VectorRemoveByPosU(&(node->childVector), 0));
            tmpChild = *(BTREENODE **)VectorGetByPos(&(rs->childVector), 0);
            if (NULL != tmpChild)
            {
                tmpChild->parent = rs;
```

```c
            }
        }
        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);
}
```