由于最开始的要求只有查找插入和删除，所以是按照书上标准的方式写的，更加详细，但是因为 B-树不能有效的进行顺序查找，不太适合通过该题，所以“B-Tree.cpp”仅用于查看查找插入删除操作。
“newBTree.cpp”则是主要为了通过该题所写，所以查找插入删除操作不太详细，结点信息在普通的 B-树之上增加了表示关键字出现次数、结点及其所有子孙的关键字总数的变量，以方便进行顺序查找。
（所以如果按照基本的要求应该只需要看“B-Tree.cpp”）

B-树查找、插入、删除操作:

1.查找操作

思路:

Search 函数在一个结点内顺序查找到第一个大于或等于 key 值的关键字并返回其位置，SearchBTree 函数从根节点开始依次向下查找，返回查找结果 (pt, i, tag)。若查找成功，则特征值 tag=1，指针 pt 所指结点中第 i 个关键字等于 K； 否则特征值 tag=0，等于 K 的关键字应插入在指针 pt 所指结点中第 i 个关键字和第 i+1 个关键字之间。

代码:

返回的查找结果记录:

```
typedef struct
{
    BTNode * pt;  //指向找到的节点
    int i;  //指示关键字序号1...m
    int tag;  //表示查找是否成功：1成功，0失败
}Result;
```

```
//在节点中查找关键字K
int Search(BTree T,KeyType K){
    //在p->key[1...keynum]中查找i，使得p->key[i] <= K < p->key[i+1]
    int i = 0, j;
    for(j = 1; j <= T->keynum; j++){
        if(T->key[j] <= K)
            i = j;
    }
    return i;
}
```

```
//在B-树上查找关键字K
Result SearchBTree(BTree T,KeyType K)
{// 在m阶B树T上查找关键字K，返回结果(pt,i,tag)。若查找成功，则特征值
// tag=1，指针pt所指结点中第i个关键字等于K；否则特征值tag=0，等于K的
// 关键字应插入在指针Pt所指结点中第i和第i+1个关键字之间。
    BTree p = T, q = NULL; //初始化，p指向待查结点，q指向p的双亲
    int found = false;
    int i = 0;
    Result r;
    while (p && !found)
    {
        i = Search(p, K);
        if(i > 0 && p->key[i] == K)
            found = true;
        else{
            q = p;
            p = p->ptr[i];
        }
    }
    r.i = i;
    if(found){
        r.pt = p;
        r.tag = 1;
    }else{
        r.pt = q;
        r.tag = 0;
    }
    return r;
}
```

复杂度:

假设 m 阶 B-树有 n 个关键字，则 Search 函数顺序查找的时间复杂度为 O(m)，假设树的高度为 h，约为 $\log_m n$ 。SearchBTree 函数在查找到 key 之前，每层都需要调用一次 Search 函数，所以总的时间复杂度为 O(m*h)，即 $O(m*\log_m n)$

2.插入操作

思路:

首先，关键字插入的位置必定在最下层的包含信息的结点中。通过查找操作找出应该插入的位置，插入关键字后可分成以下几种情况：1.插入后，该结点的关键字个数 n<m，不修改指针；2.插入后，该结点的关键字个数 n=m，则需进行“结点分裂”，令 s = m/2 取上整，保留 s 之前的关键字和指针，建立新结点存储 s 之后的关键字和指针，并将新结点插入双亲结点；3.若双亲为空，则建新的根结点。

Insert 函数在结点中指定位置插入一个关键字；NewRoot 函数生成含信息(T,r,ap)的新的根结点*T，原 T 和 ap 为子树指针；split 函数将结点 q 分裂成两个结点，前一半保留，后一半移入新生结点 ap；InsertBTree 函数先插入节点，再检查是否需要进行分裂或生成新的根结点等，并重复检查产生变化的双亲结点直至所有结点都符合要求；

代码:

```
//给节点中插入关键字
void Insert(BTree &T, int i, Record *r, BTree ap){
    int j;
    for(j = T->keynum; j > i; j--){
        T->key[j+1] = T->key[j];
        T->ptr[j+1] = T->ptr[j];
        T->recptr[j+1] = T->recptr[j];
    }
    T->key[i+1] = r->key;
    T->ptr[i+1] = ap;
    T->recptr[i+1] = r;
    T->keynum++;
}
```

```
//生成新根节点
void NewRoot(BTree &T,Record *r,BTree ap){
    // 生成含信息(T,r,ap)的新的根结点*T，原T和ap为子树指针
    BTree p;
    p = (BTree)malloc(sizeof(BTNode));
    p->ptr[0] = T;
    T = p;
    if(T->ptr[0])
        T->ptr[0]->parent = T;
    T->parent = NULL;
    T->keynum = 1;
    T->key[1] = r->key;
    T->recptr[1] = r;
    T->ptr[1] = ap;
    if(T->ptr[1])
        T->ptr[1]->parent = T;
}
```

```
//结点分裂
void split(BTree &q,BTree &ap){
    // 将结点q分裂成两个结点，前一半保留，后一半移入新生结点ap
    int i, s = (m + 1) / 2;
    ap = (BTree)malloc(sizeof(BTNode));
    ap->ptr[0] = q->ptr[s];
    if(ap->ptr[0])
        ap->ptr[0]->parent = ap;
    for(i = s + 1; i <= m; i++){   //将s之后的关键字和子树移动到新节点中
        ap->key[i-s] = q->key[i];
        q->key[i] = 0;
        ap->ptr[i-s] = q->ptr[i];
        q->ptr[i] = NULL;
        ap->recptr[i-s] = q->recptr[i];
        if(ap->ptr[i-s])
            ap->ptr[i-s]->parent = ap;
    }
    ap->keynum = m - s;
    ap->parent = q->parent;
    q->keynum = s - 1;
}
```

```
//在B-树上插入结点
Status InsertBTree(BTree &T,Record *r,BTree q,int i){
    BTree ap = NULL;
    int finished = false;
    int s;
    Record *rx;
    rx = r;
    while (q && !finished)
    {
        Insert(q, i, rx, ap);
        if(q->keynum < m)
            finished = true;
        else{  //分裂节点
            s = (m + 1) / 2;
            rx = q->recptr[s];
            split(q, ap);
            q = q->parent;
            if(q)
                i = Search(q, rx->key);
        }
    }
    if(!finished)
        NewRoot(T, rx, ap);
    return OK;
}
```

```
Status UseInsert(BTree &T, KeyType key){
    Result tmp = SearchBTree(T, key);
    Record* a = (Record *)malloc(sizeof(Record));
    a->key = key;
    if(tmp.tag == 1){
        //cout << "already exist" << endl;
        //本应是注释掉的这行代码，但是题目要求能重复
        InsertBTree(T, a, tmp.pt, tmp.i);
    }else{
        InsertBTree(T, a, tmp.pt, tmp.i);
    }
    return OK;
}
```

复杂度:

每个插入操作都要先执行查找操作，时间复杂度为 $O(m*\log_m n)$。插入操作最耗时的情况是插入后分裂操作不断向上重复进行，要重复 $\log_m n$ 次（层数），而分裂操作的时间复杂度约为 O(m/2)，所以最终插入操作的时间复杂度近似于查找操作，为 $O(m*\log_m n)$。

3.删除操作

思路:

首先要进行查找找到待删除结点，如果是底层结点，就分为三种情况：1.若结点中关键码个数大于[m/2]-1，则直接删除即可；2. 若关键码个数等于[m/2]-1，结点左(或右)兄弟结

点的关键码个数大于[m/2]-1, 则把左(或右)兄弟结点中最大(或最小)的关键码移到父结点中，并将父结点中大于(或小于)其值的关键码移到被删除关键码所在的结点中；3. 若结点的关键码个数及其左、右兄弟中的关键码个数都等于[m/2]-1，则需要合并该结点、其兄弟结点及父结点中的一个关键码，假设其有右兄弟，且父结点中由指针 Ai 指向其右兄弟，则将该结点剩下的关键码加上其父结点中 Ki 一起, 合并到 Ai 所指兄弟结点中, 并从父结点中删除 Ki。如果不是底层结点，则查找对应的的子树中的最小关键码将其替换，然后删除替换后的底层结点即可。

MoveFromLeft 和 MoveFromRight 函数对应情况 2，从左（右）兄弟中取关键字。Combine 函数对应情况 3，将兄弟结点、待删除结点与双亲结点中的一个关键字合并。DeleteBTNode 函数则对不同的情况进行区分，决定调用不同的函数、执行不同的操作。

代码：

```
//把左兄弟结点中最大的关键码移到父结点中，并将父结点中大于其值的关键码移到被删除关键码所在的结点中
void MoveFromLeft(BTree &parent, int id, BTree &T, BTree &Lbro){
    //父节点移到当前节点
    for(int i = T->keynum; i >= 0; --i){   //所有元素后移一位，空出一个位置
        T->key[i + 1] = T->key[i];
        T->ptr[i + 1] = T->ptr[i];
    }
    T->key[1] = parent->key[id];
    T->ptr[0] = Lbro->ptr[Lbro->keynum];
    if(T->ptr[0])
        T->ptr[0]->parent = T;
    Lbro->ptr[Lbro->keynum] = NULL;
    T->keynum++;
    //左兄弟节点移到父节点
    parent->key[id] = Lbro->key[Lbro->keynum];
    Lbro->key[Lbro->keynum] = 0;
    Lbro->keynum--;
}
```

```
//把右兄弟结点中最小的关键码移到父结点中，并将父结点中小于其值的关键码移到被删除关键码所在的结点中
void MoveFromRight(BTree &parent, int id, BTree &T, BTree &Rbro){
    //父节点移到当前节点
    T->key[T->keynum + 1] = parent->key[id + 1];
    T->ptr[T->keynum + 1] = Rbro->ptr[0];
    if(T->ptr[T->keynum + 1])
        T->ptr[T->keynum + 1]->parent = T;
    Rbro->ptr[0] = NULL;
    T->keynum++;
    //右兄弟节点移到父节点
    parent->key[id + 1] = Rbro->key[1];
    Rbro->key[id] = 0;
    //处理右兄弟结点
    for(int i = 0; i <= Rbro->keynum; i++){
        Rbro->key[i] = Rbro->key[i + 1];
        Rbro->ptr[i] = Rbro->ptr[i + 1];
    }
    Rbro->key[0] = 0;
    Rbro->keynum--;
}
```

```
//把左兄弟节点（当前节点）和当前节点（右兄弟结点）、父节点中的一个关键字合并到左兄弟节点（当前节点）
void Combine(BTree &parent, int id, BTree &Lbro, BTree &T){
    //当前结点（右兄弟结点）移到左兄弟结点（当前结点）
    for(int i = 0; i <= T->keynum; i++){
        Lbro->key[Lbro->keynum + 1 + i] = T->key[i];
        Lbro->ptr[Lbro->keynum + 1 + i] = T->ptr[i];
        if(Lbro->ptr[Lbro->keynum + 1 + i]){
            Lbro->ptr[Lbro->keynum + 1 + i]->parent = Lbro;
        }
    }
    Lbro->key[Lbro->keynum + 1] = parent->key[id];
    Lbro->keynum += T->keynum + 1;

    free(T);
    for(int i = id; i <= parent->keynum; i++){
        parent->key[i] = parent->key[i + 1];
        parent->ptr[i] = parent->ptr[i + 1];
    }
    parent->keynum--;
}
```

```
//删除结点并进行调整
void DeleteBTNode(BTree &T, BTNode *dnode, int id){
    if(id < 0 || id > dnode->keynum)
        return;
    while(true){
        if(dnode->ptr[id - 1] && dnode->ptr[id]){  //不是底层节点
            if(dnode->ptr[id - 1]->keynum > dnode->ptr[id]->keynum){  //用左子树最大节点替换
                BTree front = dnode->ptr[id - 1];
                dnode->key[id] = front->key[front->keynum];
                dnode = front;
                id = front->keynum;
            }else{  //用右子树最小节点替换
                BTree next = dnode->ptr[id];
                dnode->key[id] = next->key[1];
                dnode = next;
                id = 1;
            }
        }else if(!dnode->ptr[id - 1] && !dnode->ptr[id]){  //是底层结点
            for(int i = id; i <= dnode->keynum; i++){
                dnode->key[i] = dnode->key[i+1];
            }
            dnode->keynum--;

            int leastnum = ceil(m/2) - 1;
            BTree parent = dnode->parent;
```

```
            while (parent && dnode->keynum < leastnum){  //删除后不满足要求，需要调整
                int id = 0;
                for(int i = 0; i < parent->keynum; i++){
                    if(parent->ptr[i] == dnode){
                        id = i;
                        break;
                    }
                }

                BTree Lbro = NULL, Rbro = NULL;
                if(id - 1 >= 0)
                    Lbro = parent->ptr[id - 1];
                if(id + 1 <= parent->keynum)
                    Rbro = parent->ptr[id + 1];

                if(Lbro && Lbro->keynum > leastnum){   //从左兄弟借结点
                    MoveFromLeft(parent, id, dnode, Lbro);
                    break;
                }else if(Rbro && Rbro->keynum > leastnum){  //从右兄弟借结点
                    MoveFromRight(parent, id, dnode, Rbro);
                    break;
                }else{   //需要合并左（右）兄弟
                    if(Lbro)
                        Combine(parent, id, Lbro, dnode);
                    else
                        Combine(parent, id + 1, dnode, Rbro);
                    dnode = parent;
                    parent = dnode->parent;
                }
            }

            if(dnode->keynum == 0){  //根节点为空
                T = dnode->ptr[0];
                free(dnode);
            }

            break;
        }
    }
}
```

复杂度:

删除操作同样需要先执行查找操作，时间复杂度为 $O(m*\log_m n)$。删除数据的过程与插入过程类似，都是最坏情况下要重复大约 $\log_m n$ 次。所以删除操作的时间复杂度与插入操作近似相同，也为 $O(m*\log_m n)$。

T3369

在普通的插入和删除操作中，同时注意对结点的 size 值进行修改，所以在解决命令 3、4、时就可以直接利用 size 的值来免去顺序查找所有子树的环节。

所属题目	P3369 【模板】普通平衡树
评测状态	Accepted
评测分数	100
提交时间	2021-12-25 19:06:28

3.

```
//查询key数的排名
int solverank(const int &key)
{
    BTNode *x = root;
    int ret = 0;
    while (x)
    {
        if (x->key(x->keyNum - 1) < key)
        {
            ret += x->size - getSize(x->ptr[x->keyNum]);
            x = x->ptr[x->keyNum];
            continue;
        }
        for (int i = 0; i < x->keyNum; ++i)
        {
            if (x->key(i) < key)
                ret += getSize(x->ptr[i]) + x->count(i);
            else if (x->key(i) == key)
                return ret + getSize(x->ptr[i]) + 1;
            else
            {
                x = x->ptr[i];
                break;
            }
        }
    }
    return ret;
}
```

复杂度分析：在求关键字 key 的排名时，只会进入包含关键字 key 的子树，其余子树不需要进行遍历，只要得到它们的 size 即可，即每一层最多只需要遍历一个结点，所以复杂度为 O(m*h)，即 $O(m*\log_m n)$

4.
```
//查询排名为x的数
int findrank(int k)
{
    BTNode *x = root;
    while (true)
    {
        for (int i = 0; i <= x->keyNum; ++i)
        {
            //const int csz = getSize(x->ptr[i]) + (i == x->keyNum ? 1 : x->count(i));
            const int lb = getSize(x->ptr[i]) + 1, ub = getSize(x->ptr[i]) + (i == x->keyNum ? 1
            if (k >= lb && k <= ub)
                return x->key(i);
            if (k < lb)
            {
                x = x->ptr[i];
                break;
            }
            k -= ub;
        }
    }
}
```

复杂度分析：与 3 同理，只会进入包含排名为 k 的关键字的子树，其余子树不需要进行遍历，即每一层最多只需要遍历一个结点，所以复杂度为 O(m*h)，即 $O(m*\log_m n)$

5.

```
//找前驱
int findprior(const int &key)
{
    BTNode *x = root;
    int ret;
    while (x)
    {
        int i = lower_bound(x->values, x->values + x->keyNum, key, pairComp) - x->values;
        if (i)
            ret = x->values[i - 1].first;
        x = x->ptr[i];
    }
    return ret;
}
```

复杂度分析：每一层只进入一个节点，这个节点中要进行查找， 所以复杂度为 O(m*h)，即 $O(m*\log_m n)$

6.

```
//找后继
int findnext(const int &key)
{
    BTNode *x = root;
    int ret;
    while (x)
    {
        int i = lower_bound(x->values, x->values + x->keyNum, key, pairComp) - x->values;
        if (x->values[i].first == key)
            ++i;
        if (i != x->keyNum)
            ret = x->values[i].first;
        x = x->ptr[i];
    }
    return ret;
}
```

复杂度分析：与 5 同理