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：

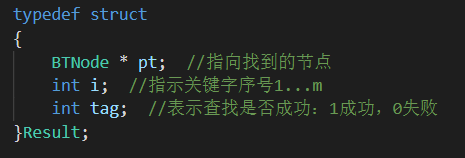
1.查找操作

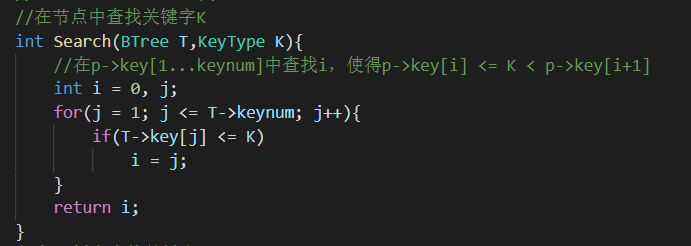
思路：

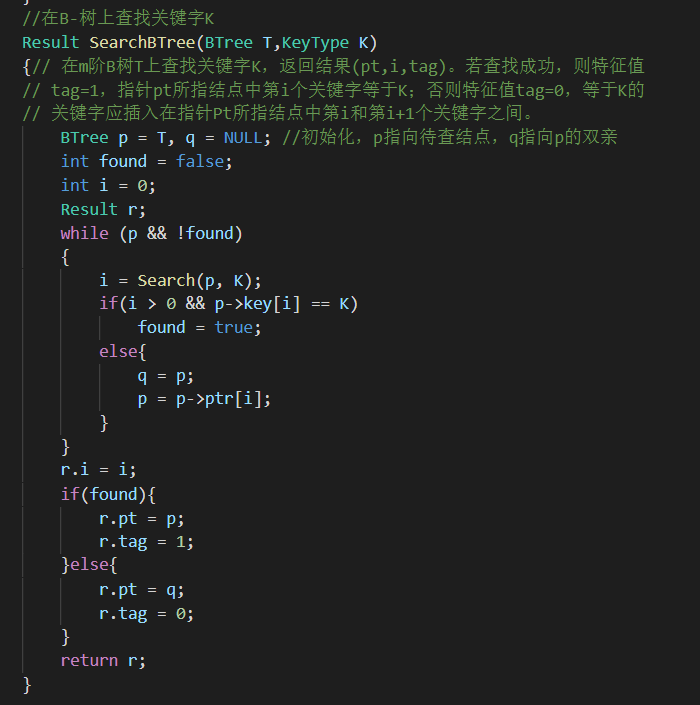
Search函数在一个结点内顺序查找到第一个大于或等于key值的关键字并返回其位置，SearchBTree函数从根节点开始依次向下查找，返回查找结果 (pt, i, tag)。若查找成功，则特征值 tag=1, 指针 pt 所指结点中第 i 个关键字等于 K； 否则特征值 tag=0, 等于K 的关键字应插入在指针 pt 所指结点中第 i 个关键字和第 i+1个关键字之间。

代码：

返回的查找结果记录：







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logmn 。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\*logmn)

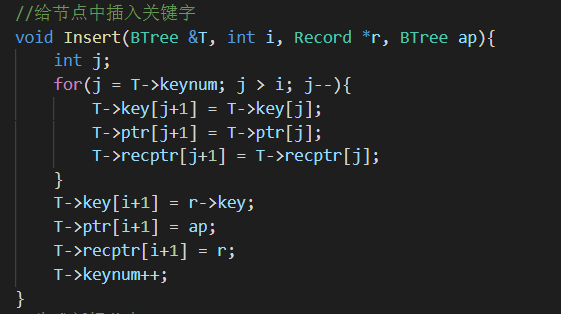
2.插入操作

思路：

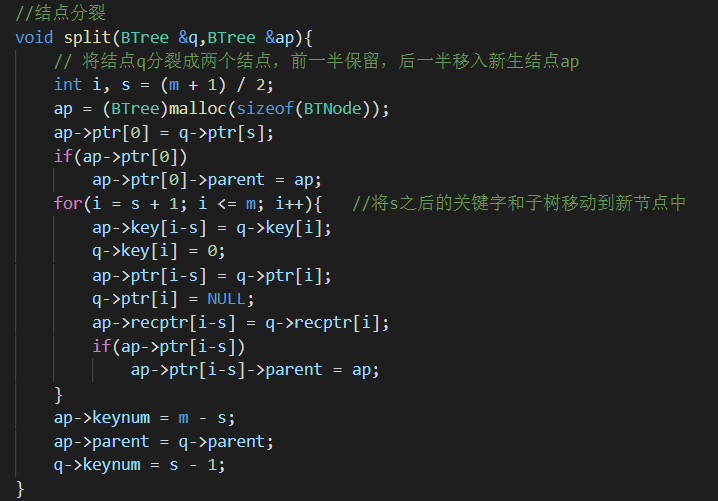
首先，关键字插入的位置必定在最下层的包含信息的结点中。通过查找操作找出应该插入的位置，插入关键字后可分成以下几种情况：1.插入后，该结点的关键字个数n<m，不修改指针；2.插入后，该结点的关键字个数 n=m，则需进行“结点分裂”，令 s = m/2取上整，保留s之前的关键字和指针，建立新结点存储s之后的关键字和指针，并将新结点插入双亲结点；3.若双亲为空，则建新的根结点。

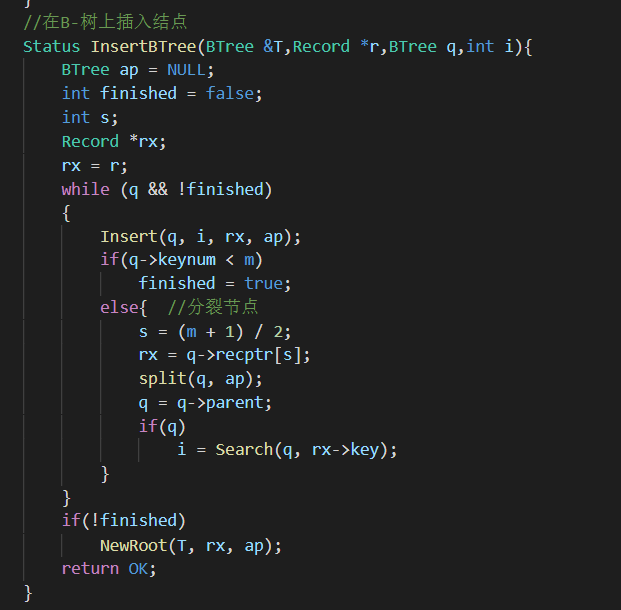
Insert函数在结点中指定位置插入一个关键字；NewRoot函数生成含信息(T,r,ap)的新的根结点\*T，原T和ap为子树指针；split函数将结点q分裂成两个结点，前一半保留，后一半移入新生结点ap；InsertBTree函数先插入节点，再检查是否需要进行分裂或生成新的根结点等，并重复检查产生变化的双亲结点直至所有结点都符合要求；

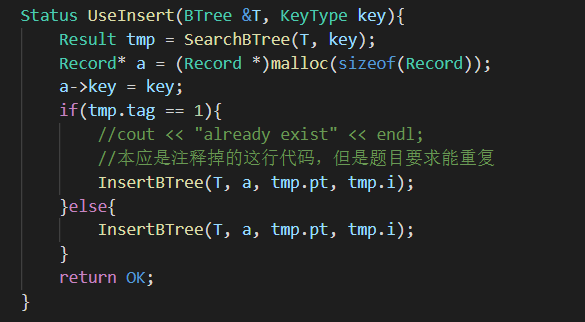
代码：











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\*logmn)。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logmn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\*logmn)。

3.删除操作

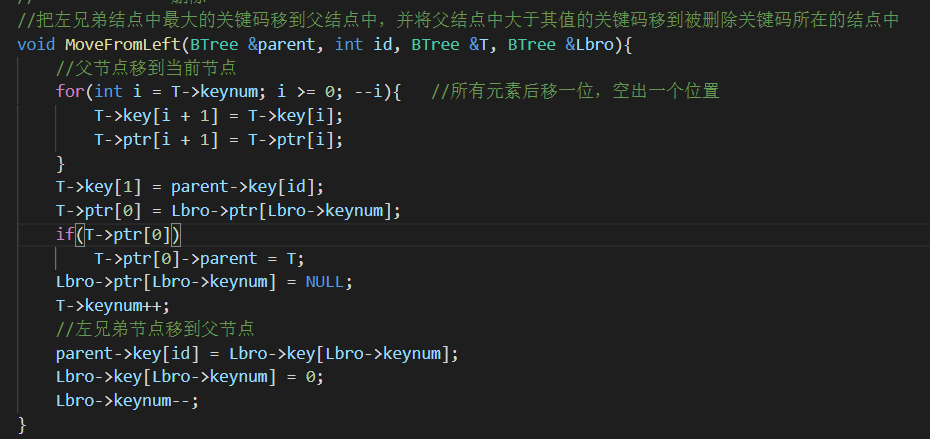
思路：

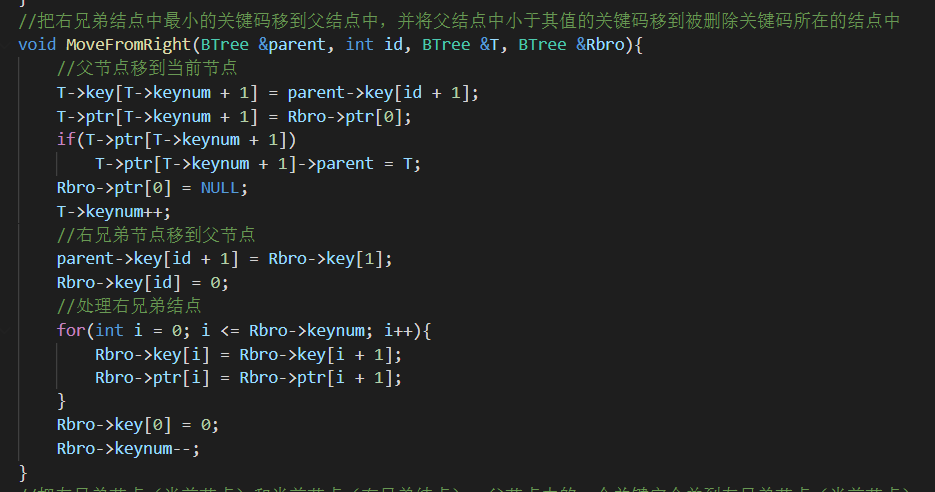
首先要进行查找找到待删除结点，如果是底层结点，就分为三种情况：1.若结点中关键码个数大于[m/2]-1，则直接删除即可；2. 若关键码个数等于[m/2]-1，结点左(或右)兄弟结点的关键码个数大于[m/2]-1，则把左(或右)兄弟结点中最大(或最小)的关键码移到父结点中，并将父结点中大于(或小于)其值的关键码移到被删除关键码所在的结点中；3. 若结点的关键码个数及其左、右兄弟中的关键码个数都等于[m/2]-1，则需要合并该结点、其兄弟结点及父结点中的一个关键码，假设其有右兄弟，且父结点中由指针Ai指向其右兄弟，则将该结点剩下的关键码加上其父结点中Ki一起，合并到Ai所指兄弟结点中，并从父结点中删除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。

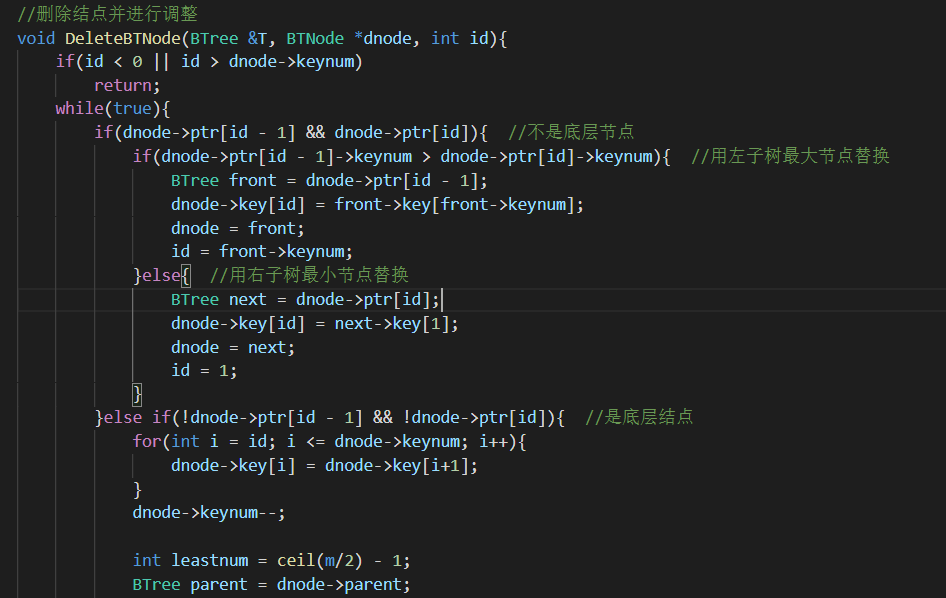
MoveFromLeft和MoveFromRight函数对应情况2，从左（右）兄弟中取关键字。Combine函数对应情况3，将兄弟结点、待删除结点与双亲结点中的一个关键字合并。DeleteBTNode函数则对不同的情况进行区分，决定调用不同的函数、执行不同的操作。

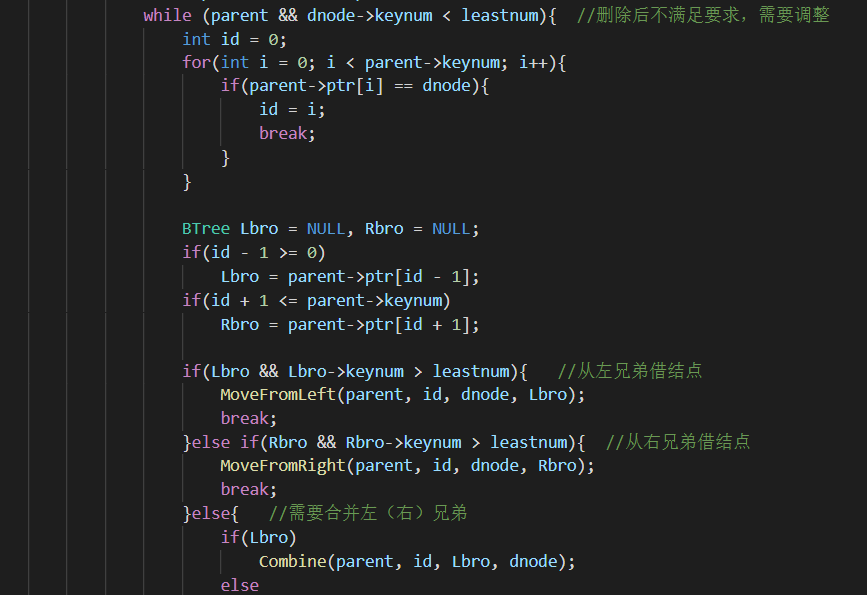
代码：

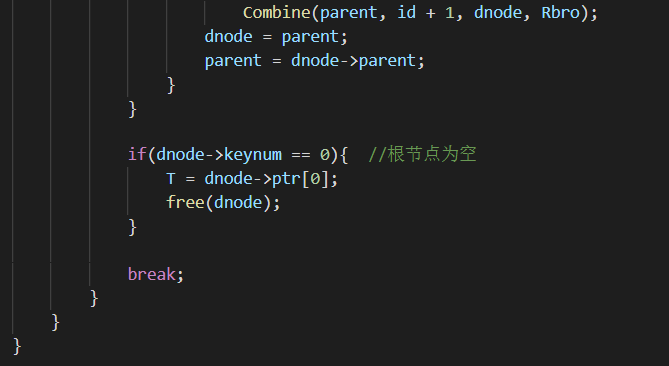












复杂度：

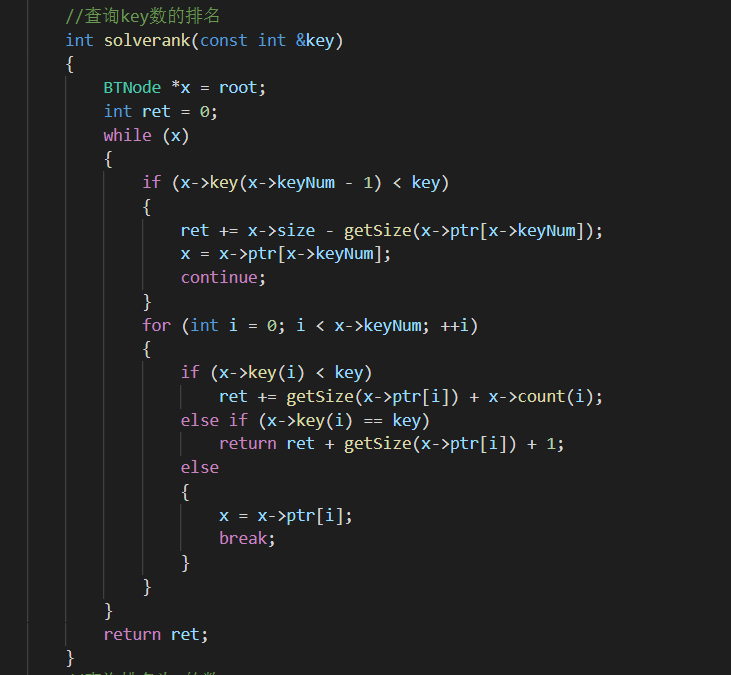
删除操作同样需要先执行查找操作，时间复杂度为O(m\*logmn)。删除数据的过程与插入过程类似，都是最坏情况下要重复大约logmn次。所以删除操作的时间复杂度与插入操作近似相同，也为O(m\*logmn)。

T3369

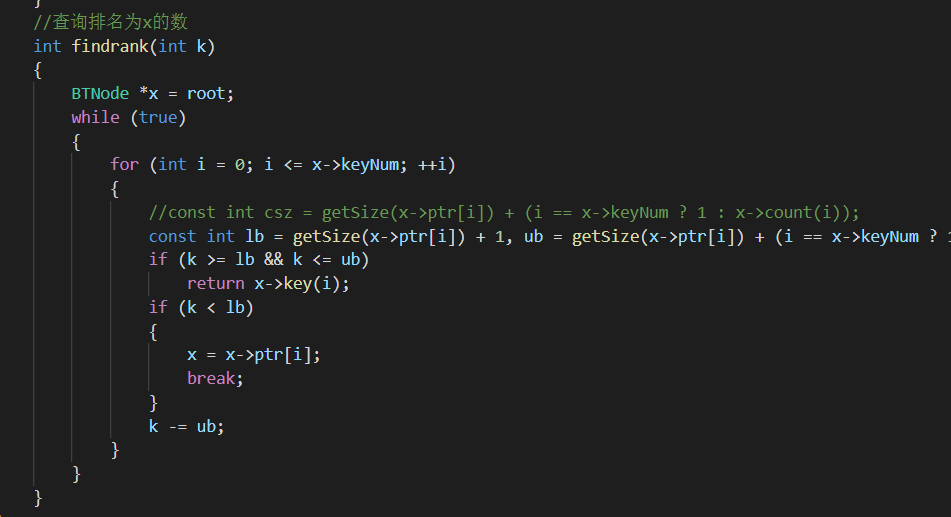
在普通的插入和删除操作中，同时注意对结点的size值进行修改，所以在解决命令3、4、时就可以直接利用size的值来免去顺序查找所有子树的环节。



3.

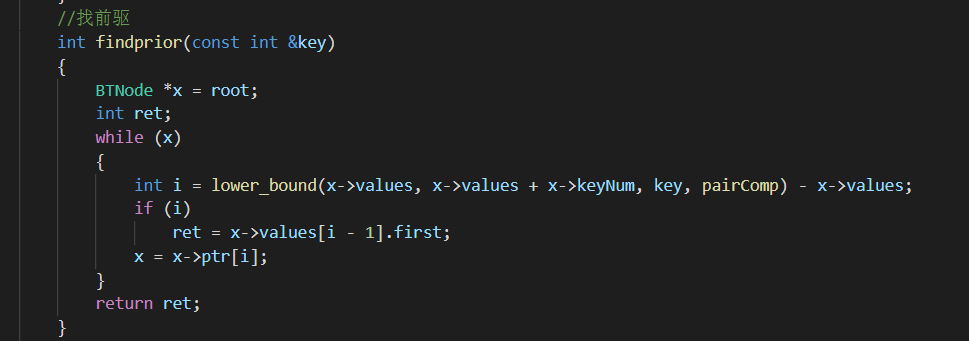


复杂度分析：在求关键字key的排名时，只会进入包含关键字key的子树，其余子树不需要进行遍历，只要得到它们的size即可，即每一层最多只需要遍历一个结点，所以复杂度为O(m\*h)，即O(m\*logmn)

4.

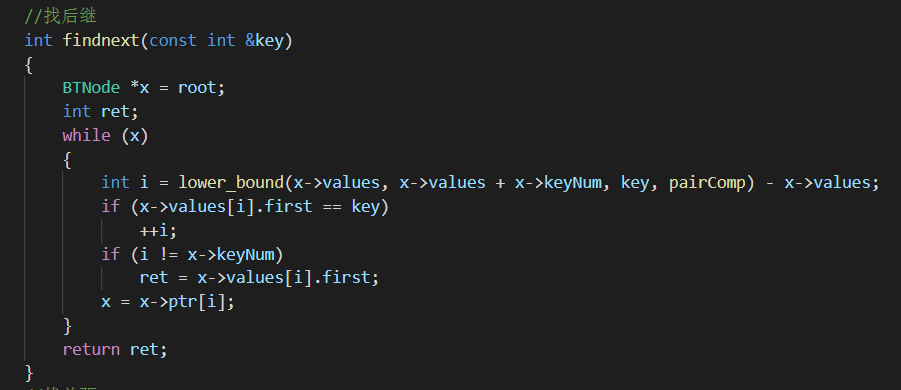
复杂度分析：与3同理，只会进入包含排名为k的关键字的子树，其余子树不需要进行遍历，即每一层最多只需要遍历一个结点，所以复杂度为O(m\*h)，即O(m\*logmn)

5.



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\*logmn)

6.



复杂度分析：与5同理