Name: Shuying Tang(st765), Jingchen Li(jl3426) , Yingda Du(yd275)

Code Logic for Search(…)

1. Overall the Search() uses recursion to reach a leaf node and then iterate through the leaf's keys to find the target key.
2. The structure of Search(key) is that it uses an helper function **treeSearch(node, key)**. Inside the treeSearch(), it will recursively call itself but each time, the node goes deeper until reaches leaf node. After reaching the node, it iterates the leaf to find the key and return the value. If nothing find, it returns null.

Code logic for insert(…)

1. Similarly to the case of Search(), an helper function **treeInsert(node, key, entry)** does all the heavy lifting.
2. treeInsert(node, key, entry) will recursive call itself with deeper node and finally insert the key into right leaf. If the leaf is overflow situation, the leaf will split into two leaves. The right leaf is newly created one. Then it returns with entry which consisting of a splitKey and the newly splited leaf to remind the outer treeInsert(…) to change accordingly.
3. When the inner treeInsert(…) returns. The outer treeInsert(…) continues. If the entry returned by inner is null, it return null to the outer outer treeInsert(…) as well. But if the entry returned by inner isn't null, it will use the splitKey in the entry to add the node in the entry at appropriate place. After adding nodes, the outer node checks its own overflow situation. Split itself if overflows and returns with the new entry to the outer outer nodes. Returns with null otherwise.
4. Then the outer outer node will continue with the step 3 until the node itself becomes the root, in this case, overflow won't be a problem and the most outer treeSearch(…) ends.

Code logic for delete(…)

1. Similarly to before, the insert(…) calls an helper function **treeDelete(parent_node, current_node, key, splitKey)** to do the heavy lifting.
2. Similarly to treeInsert(…), treeDelete(parent_node, current_node, key, splitKey) recursively calls itself until go down to leaf and delete accordingly. After deletion, if there's no redistribution or merge to do, or no deletion at all, the most inner treeDelete(…) will return with splitKey set to null. Then the outer treeDelete(…) will just keep return with splitKey set to null until the most outer execution ends.
3. If at the leaf level it needs to redistribute or merge, the most inner function will **findSiblings(node, parent)** to find the target sibling of current node to perform the redistribution or merge at leaf level. Then return with the splitKey that indicates the key in the parent that needs to be changed accordingly. treeDelete(…) at parent node then receives the splitKey and delete the excessive key and children. Then the parent node will check itself to see if need redistribution or merge. If no need, it returns with splitKey to null to allow the whole program to end, if needs, it performs the redistribution or merge at its own level and returns a splitKey to inform its own parent.

4. Logic of step 3 will continues until the most outer function, which is at root level, receives a null for terminating itself or receiving a splitKey to modify itself. Then the root only needs to check itself need to exist, if it's already empty, then set its child as the new root else, since root can be underflow, it just need to return.