Alexander Shah
Homework 9: Searching Ordered Data and Search Trees
Data Structures

1. a) If no record with the key target is present.

In a sequential search through unordered data, the search must go through all n items in the structure in order to determine that the target is not there as an unordered search must keep iterating until it finds something. In an ordered structure, a sequential search will still iterate through all n items in order to determine the key is not present.

b) If one record with the key target is present and only one is sought.

In an ordered list a sequential search will iterate through the list until the target is found. The search will find the target and return only the one element. In an unordered list the sequential search will still iterate over the n items until it finds the target and returns it. In an unordered search the likelihood of finding the specific target is random as we don't know where in the linear list the target could be, comparatively, the ordered search has an average case of n/2 and the closer the target is to the beginning of the ordered search, the faster it returns the target.

2. a) If more than one record with the key target is present and it is desired to find only the first one.

In both an unordered and ordered list, the sequential search will iterate through the elements until the first instance of the target is found, which is returned. If we don't care about returning more elements, the search is done.

b) If more than one record with the key target is present and it is desired to find them all.

In a typical sequential search, the first instance of the target is returned, if we want to return them all, the search would need to be modified to return a data structure containing the elements or some kind of struct or tuple. Specifically for an unordered search, all elements must be looked at every time the search is run to find all instances, in an ordered search, the elements will be one after another, so a range can be found by searching for a beginning point and ending point which can be returned like pointers or the elements themselves.

3. Write a method delete(key1, key2) to delete all records with keys between key1 and key2 (inclusive) from a binary search tree whose nodes look like this: [Left, keyi, right]

In a BST we can recursively call a delete function with parameters for high and low values of our keys in order to remove elements between them while keeping the tree intact. This can be accomplished by finding the correct nodes in the tree, filling them in with the leftmost nodes (smallest) and once we have removed from all the way right we recurse back up.

```
Method delete(node, key1, key2):

   // key2 should be >= to key1
   if key1 > key2:
```

```
        tmp = key1
        key1 = key2
        key2 = tmp

    // Base case if no nodes left
    if node == null:
        return null
    // recurse left subtree
    if node.key >= key1:
        node.left = delete(node.left, key1, key2)
    // recurse right subtree
    if node.key <= key2:
        node.right = delete(node.right, key1, key2)

    // keys within range
    if node.key >= key1 and node.key <= key2:
        // with one or fewer children, we can snip empty branches
        if node.left == null:
            return node.right
        else if node.right == null:
            return node.left
        else:
            // has left and right child
            // remove what's within range by swapping the smallest node (furthest left) within node's right
            minNode = node.right
            while minNode.left is not null
                minNode = minNode.left
             // replace node
             node.key = minNode.key

            // recursively delete right tree until bst is achieved
            // todo delete(node.right, node.key, key2)
            node.right = delete(node.right, minNode.key, key2)

    return node
```

4. Write a method to delete a record from a B-tree of order n.
[p0, r1, p1, r2, p2, r3, ..., pn-1, rn, pn]

```
method deleteFromBTree(node, key, order)
    if node is null
        return null

    index = getIndex(node, key)
    if index != -1 // key is in node
        if node is leaf
            // remove key from leaf node
```

```
            deleteKeyFromLeaf(node, key)

        // key in internal node, check:
        // - if left child has > min keys, replace with inorder predecessor
        // - if right child has > min keys, replace with inorder successor
        // - if child has min keys, merge left and right children
        else
            if node.children[index].numberOfKeys > order / 2
                predecessor = getPredecessor(node, index)
                node.keys[index] = predecessor
                // recurse
                node.left = deleteFromBTree(node.left, predecessor, order)
            else if node.children[index + 1].numberOfKeys > order / 2
                successor = getSuccessor(node, index)
                node.keys[index] = successor
                // recurse
                node.right = deleteFromBTree(node.right, successor, order)
            else
                // merge children and recurse
                mergeChildren(node, index, order)
                node.child[index] = deleteFromBTree(node.child[index], key, order)

    else
        // key is not in the node
        // keep b-tree properties by borrowing from siblings
        // ensure child has enough keys
        if node.child[index].numberOfKeys == (order - 1) / 2
            if index != 0 and node.child[index - 1].numberOfKeys > (order - 1) / 2
                borrowFromPrev(node, index)
            else if index != node.numberOfKeys and node.child[index + 1].numberOfKeys > (order - 1) / 2
                borrowFromNext(node, index)
            else
                // merge with sibling node
                if index != node.numberOfKeys
                    mergeChildren(node, index)
                else
                    mergeChildren(node, index - 1)
        deleteFromBTree(node.child[index], key, order)

    // handle zero key root
    if node.numberOfKeys == 0
        if node is leaf
            return null
        else
            return node.child[0]

    return node
```