Assignment 12 – Search Trees and Hashing

Write pseudo-code not Java for problems requiring code. You are responsible for the appropriate level of detail.

1. 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	key$_i$	right

Some pointers:
- if it’s a leaf, just delete it and no problem or rebalance is needed.
- if one child, promote one child to root, take link coming and and point it to child.
- if two children, take maximum child in the left subtree, or minimum node in right subtree. and promote taht to be new root, that will keep the binary search tree valid. You can’t just redirect link coming into the parent node with 2 children because you have to pick a child. This two child case will be recursive.

psuedocode:

```
class Node
    int key
    Node left
    Node right
    Node(int data)
            this.data = data
            this.left = null
            this.right = null

// used to find in order successor in the right subtree of root
def findLeftMost(root)
        if (root == null)
            return null;
        while (root.left != null)
            root = root.left;
        return root;


def deleteNodeSingle(root)
            // nodes with only one child or no
            if (root.left == null) {
                        child = root.right;
                        root = null;
                        return child;
            }
            else if (root.right == null) {
                        child = root.left;
                        root = null;
                        return child;
            }

            // node with two children: get the in-order successor in the right subtree
            next = leftMost(root.right);

            // copy the inorder successor's content to this root node
            root.data = next.data;

            // delete the inorder successor
            root.right = deleteNodeSingle(root.right);

            return root


// recursive function to find node in given range and delete it in preorder manner
def removeRange( node, int low, int high)
        if (node == null)
            return null

        // First fix the left and right subtrees of node
        node.left = removeRange(node.left, low, high);
        node.right = removeRange(node.right, low, high);

        // if given node is within the range, delete it
        if (node.data >= low && node.data <= high)
            return deleteNodeSingle(node);

        // Root is out of range, then just return it
        return node;
```

2. Write a method to delete a record from a B-tree of order n.

p_0	r_1	p_1	r_2	p_2	r_3	…….	p_{n-1}	r_{n-1}	p_n

Some pointers:

- Since you have a lower-bound on the number of elements in a node, if removing your elements violates this invariant, then you need to restore it, which generally involves merging with a neighbour (or stealing some of its elements).
- If you merge with a neighbor, then you need to remove an element in the parent node, which triggers the same algorithm. And you apply recursively till you get to the top.

Algo:
1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node, delete the desired key from the node and you're done.
3. If the leaf node doesn't contain at least m/2 keys, then complete the keys by taking the element from the right or left sibling.

    3a. If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the appropriate (the parent element in between the two children) element down to the node where the key is deleted.

    3b. If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move the appropriate element down to the node where the key is deleted.

4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the appropriate element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too. So recursively do this again on the parent.

we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.
pseudo code:

```
def deleteKeyFromNodeInBTree(parentNode, currentNode, keyToDelete)
   If (isLeaf(currentNode)] == TRUE)
      Search for keyToDelete in current-node

      If (keyToDelete not found)
         return false // no key found

      If (total number of keys in currentNode > LOWERBOUNDOFBTREE)
         foundKey = Remove the key in currentNode
         Return foundKey

         // these are more complex cases
      leftSibling, rightSibling = Get left-sibling-node and right-sibling-node of current-node

      // attempt to borrow from left Sibling
      If (LeftSibling is found AND total number of keys in leftSibling > LOWERBOUNDOFBTREE)
         foundKey = Remove keyToDelete from currentNode
```

```
        Perform right rotation function on left sibling
        Return foundKey

        // attempt to borrow from right sibling
      If (rightSibling is found AND total number of keys in rightSibling > LOWERBOUNDOFBTREE)
        foundKey = Remove deletedKey from currentNode
        Perform left rotation
        Return foundKey

      // if the siblings have too few nodes, gotta merge
      If (leftSibling is not NULL)
        Merge current-node with left-sibling-node
      Else
        Merge current-node with right-sibling-node
      End If

        // parent now might be in violation so we need to rebalance upward.
      Return Rebalance-BTree-Upward(current-node)
    End If
```

//The preemptive merge removal scheme involves increasing the number of keys in all single-key, non-root nodes encountered during traversal. The merging always happens before any key removal is attempted. Preemptive merging ensures that any leaf node encountered during removal will have 2 or more keys, allowing a key to be removed from the leaf node without violating the 2-3-4 tree rules.

```
    Find predecessor-node of current-node
    Swap the right-most key of predecessor-node and deleted-key of current-node
    Delete-Key-From-Node(predecessor-parent-node, predecessor-node, deleted-key)
```

3. If a hash table contains *tablesize* positions and *n* records currently occupy the table, the load factor *LF* is defined as *n/tablesize.* Suppose a hash function uniformly distributes *n* keys over the *tablesize* positions of the table and the table has load factor *LF*. Show that of new keys inserted into the table, (n-1) * (*LF*/2) of them will collide with a previously entered key. Think about the accumulated collisions over a series of collisions.

The load factor is computed as: $\alpha = N/M$ where M = size of the table N = number of keys that have been inserted in the table.

If the hash function makes the keys uniformly distributed, each key has the same probability of being hashed into any table slot, and that probability is 1/T.

key: 0, probability of collision: 0
key: 1, probability of collision: 1/T
key: 2, probability of collision: 2/T (it could either collide with 0 or 1)
key: 3, probability of collision: 3/T

key: n-1, probability of collision: (n-1)/T

number of collisions = sum(i=0 to n-1) * i/T = (1/T)sum(i=0 to n-1) * i = (1/T)*[n(n-1)/2] = (n/T)*(n-1)/2 = **(n-1) * lf/2.**

Intuitive proof using examples:
Does this make sense when I plug in numbers? Say tablesize = 2, n = 2 so plugging into the formula = 1 * 1/2 = 1/2. So, 1/2 of them collided with a previously entered key which makes sense since the last key inserted will have a % of 50% to collide.

tablesize = 3, n = 3, plugging into the formula, I get 2 * 1/2 = 1 which does not make sense. It shouldn't ever be 100%. It eventually gets very close to 100% but not 100% right? What am I doing wrong?

To grader: I am lost. What is going on? Why are my intuitive examples not making sense?

****EDIT****
Maybe the intuitive approach is more like this:
Starting from an empty table of say tablesize = 10, so 10 open spots:
if we insert 1 record, (n-1) * (lf/2) = 0 * 1/10 * 1/2 = 0
if we insert 2 record, (n-1) * (lf/2) = 1 * 2/10 * 1/2 = 1/10
if we insert 3 record, (n-1) * (lf/2) = 2 * 3/10 * 1/2 = 3/10 nodes will collide
…
if we insert 10 record, (n-1)* (lf/2) = 9 * 10/10 * 1/2 = 4.5 inserts will collide

so this intuitively makes sense… as the number of inserts grows, the number of collisions grows pretty quickly as n gets closer to table size.

4. Assume that *n* random positions of a *tablesize*-element hash table are occupied, using hash and rehash functions that are equally likely to produce any index in the table. The hash and rehash functions themselves are not important. The only thing that is important is they will produce any index in the table with equal probability. Start by counting the number of insertions for each item as you go along. Use that to show that the average number of comparisons needed to insert a new element is *(tablesize* + 1*)/(tablesize*-n+1). Explain why linear probing does not satisfy this condition.

table size is number of buckets. explain why the formula works. as it grows it will do this. not an actual proof. check table size =1 and tablesize = 0

when tablesize = 10, and n = 0, function (tablesize + 1)/(tablesize-n+1) = 1. So 1 comparison
when tablesize = 10, and n = 1, function (tablesize + 1)/(tablesize-n+1) = 11/10. So 1.x comparison.
when tablesize = 10, and n = 2, function (tablesize + 1)/(tablesize-n+1) = 11/(9). So some number slightly greater than 1 again and bigger than n =1.
…
when tablesize = 10, and n = 10, function (tablesize + 1)/(tablesize-n+1) = 11/1. So 11 comparisons. This makes sense since the table is now full. This won't even be inserted since there isn't any space.

So as N grows larger, the number of comparisons gets bigger since the probability of collision goes up.

Linear probing does not satisfy this because it depends on how the inserts are done. In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also. The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element. So if the inserts are clustered… it will take a lot more time than if the inserts were not clustered.