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:


```
public class BinaryTree {
        public Node root;

        public class Node {
            public Node right;
            public Node left;
            public int key;
        }

        public delete(key1, key2) {
            for node in nodes{ // iterate over each node
                if(node.key > key1 && node.key < key2) {
                    BSTRemove(this.BinaryTree, node.key); // extend code from zybook
                }
            }
        }

        BSTRemove(tree, key) { // This is from the Zybook assignment, no need to duplicate this code…
          par = null
          cur = tree->root
          while (cur is not null) // Search for node
            if (cur->key == key) // Node found
              if (!cur->left && !cur->right)        // Remove leaf
                if (!par) // Node is root
                  tree->root = null;
                else if (par->left == cur)
                  par->left = null
                else
                  par->right = null
              else if (cur->left && !cur->right)    // Remove node with only left child
                if (!par) // Node is root
                  tree->root = cur->left
                else if (par->left == cur)
                  par->left = cur->left
                else
                  par->right = cur->left
              else if (!cur->left && cur->right)    // Remove node with only right child
                if (!par) // Node is root
                  tree->root = cur->right
```

```
            else if (par->left == cur)
              par->left = cur->right
            else
              par->right = cur->right
          else                          // Remove node with two children
            // Find successor (leftmost child of right subtree)
            suc = cur->right
            while (suc->left is not null)
              suc = suc->left
            cur = Copy suc                  // Copy successor to current node
            BSTRemove(cur->right, suc->key)     // Remove successor from right subtree
          return // Node found and removed
        else if (cur->key < key) // Search right
          par = cur
          cur = cur->right
        else                    // Search left
          par = cur
          cur = cur->left
      return // Node not found
    }
}
```

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

```
public class BTree() {
   public order n;
   public Node head;

   public class Node() {
      public Node parent;
      public Node[] children;
      public int[] vals;
   }

   public void delete(int record) {
      Node recordNode = findRecord(record);
      if(!recordNode) {
         return false; // delete nothing if the node does not exist
      }
      recordNode.vals.remove(record);
      if(!recordNode.children) { if this is a leaf node, you are done
         return;
      }
      int nodeIndex = 0;
      while(vals[nodeIndex] < record) { // find the index of the record
         nodeIndex++;
```

```
    }
    leftNode = recordNode.children[nodeIndex];
    rightNode = recordNode.children[nodeIndex+1];
    if(leftNode.children.count() + rightNode.children.count() < this.order) { // if we can merge children
        merge(recordNode, record); // if we can merge children, merge children
    else {
        bringChildValUp(recordNode, record); // fill parent node and  restructure children
    }
return;
}



public Node findRecord(int record) {
    Node node = this.head;
    for(int n = 0; n < this.order; n++) {
        if(node.children.contains(record) {
            return node;
        }
    }
    return null; // return null if the node does not exist
}

public void merge(Node node, int val) {
    int mergeBelowIndex = 0;
    while(node.vals[mergeBelowIndex] < val) { // find which two nodes to merge
        mergeBelowIndex++;
    }
    Node leftNode = node.children[mergeBelowIndex];
    Node rightNode = node.children[mergeBelowIndex+1];
    leftNode.vals = leftNode.vals.append(rightNode.vals); append right vals to left node
    node.children.remove(mergeBelowIndex+1) // remove by index the right node (already copied)
}

public void bringChildValUp(Node node, int val) {
    int bringUpFromIndex = 0;
    while(node.vals[bringUpFromIndex] < val) { // find which two nodes can be brought up
        bringUpFromIndex ++;
    }
    Node leftNode = node.children[bringUpFromIndex];
    Node rightNode = node.children[bringUpFromIndex +1];
    if(leftNode.vals.count > rightNode.vals.count) {
        node.vals[bringUpFromIndex] = leftNode.vals[-1]; //bring up last val from leftNode
        leftNode.vals.remove(leftNode.vals[-1]);
    else {
        .vals[bringUpFromIndex] = rightNode.vals[1]; //bring up first val from rightNode
         rightNode.vals.remove(rightNode.vals[0]);
    }
}
```

}

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.**

lf is the load factor of the table. So for the first key, there will be a lf of 0, and for the last key there will be a lf of lf, and there will be a linear connection between these two points as the table fills up. So to get the number of keys that collide, we can take the average of the max and min loadfactor, then multiply by the number of insertions that can collide. The average load factor of the table is lf/2. Then the number of collisions that may have a collision are n-1 (since the first entry cannot collide). Therefore, the number of keys that will collide is (n-1)*lf/2.

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.

The average number of insertions into an empty table is 1. If you have a hash table with size tablesize, and only one entry, there is a (tablesize-1)/tablesize chance that you only have to do one insertion. Then there is a 1/tablesize chance that you have to do 2 insertions. So the average number of insertions is 2/tablesize + tablesize-1/tablesize. Then if you have 2 insertions, there is a (tablesize-2)/tablesize chance that you only have to do one insertion, and a 2/tablesize chance that you have to do 2 or more insertions. The chance that you have to do 3 insertions is 1/tablesize * 2/tablesize, and the chance that you have to do 2 insertions is (1-tablesize)/tablesize. So the average number of insertions for a table of size tablesize and 2 entries is (tablesize-2)/tablesize + 2 * (1-tablesize)/tablesize + 3 * 1/tablesize * 2/tablesize. This simplifies to (tablesize + 1) / (tablesize – 1). Continuing this process for any n, we get the equation (tablesize + 1) / (tablesize + 1 – n).

However this assumes that the next choice in the hashing function will be random. This is not the case for a linear probe. In a linear probe it is possible that as you increase the load factor, that long strings of full slots start to appear. As the strings get longer, the chances that you land on the string increase, and thus increases the length of the string in an endless cycle. Landing on the string means that the chance the next insert is empty is exponentially less, so you will not get as favorable number of inserts as you would in a random hash, especially if the table is closer to being full.