Minor 2 Solutions

Q1

Given the Hash function h below, list the table slots touched/probed and show the status of the Hash table after each listed operation (in order from left to right, starting with an empty hash table). The table has 5 slots. Assume open addressing with hi as given: (Deletion is by marking as deleted.)

$$h = h_0 = (\text{ digits }) \ \% \ 5$$
$$h_i = (h_0 + 3{*}i) \ \% \ 5$$

1 (a)

- Insert 873/837

873/837

Slots Probed : 3

- Insert 9734/8843

9734/8843
873/837

Slots Probed : 3, 1

- Insert 280/640

280/640
9734/8843
873/837

Slots Probed : 0

- Delete 9734/8843

280/640
deleted
873/837

Slots Probed : 3, 1

- Search 143

280/640
deleted
873/837

Slots Probed : 3, 1, 4 (Unsuccessful search)

- Insert 14/32

280/640
14/32
873/837

Slots Probed : 0, 3, 1

1(b) Search terminates with failure if an empty slot is probed, or if the probe reaches back to the first probed slot.

Q2

```
if(g == root) {
  root = n;
  n.parent = null;
} else {
  if(g.parent.left == g) g.parent.left = n;
  else g.parent.right = n;
  n.parent = g.parent;
}
p.right = n.left; if(n.left != null) n.left.parent = p;
g.left = n.right; if(n.right != null) n.right.parent = g;
n.left = p; p.parent = n;
n.right = g; g.parent = n;
```

Q3

```
delete(root, key):
    node = search(root, key), get the first node where key is found
    return if not found
    if(node.right == null) {
        if (node == root) root = node.left
        else node's parent.left = node.left // ie, remove node, promoting left child
        node = node.left // The promoted child is gets the node reference
    } else {
         successor = node.right;
         while(successor.left != null) successor = successor.left;
         move successor value to node
         remove successor, promoting any right child
         node = node.left // Start in the left subtree
    }
   while(node != null) {       // Traverse down starting at node
      if(node.key == key) {
         node's parent.left = node.left // remove node promoting the left child
         node = node.left
      } else // node.key must be less then key
         node == node.right
   }


search (node, key):
    if(node.key == key) add key to result
    if(node.key < key) search(node.right, key)
       else search(node.left, key)
}
```

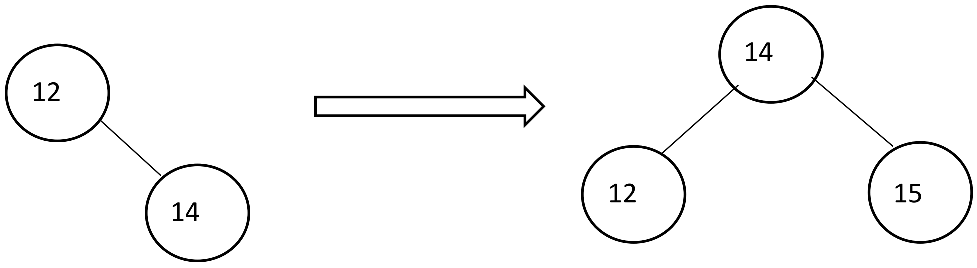
Search always follows one of the child references, making a single pass down the tree. Deletion may make one pass down the right child of the top occurrence of the key and then one pass down its left child. On each of these two passes, it takes follows only one child references, making for at most two passes down the tree. Thus both are O(h).

Q4

We need to lock the root. Locking individual nodes would not suffice for a self-balancing tree such as Red-Black tree. Concurrent insert and delete can end up in deadlock if individual nodes are locked because the tree is traversed top-down as well as bottom-up. Assume one thread t1 is trying to insert and another thread t2 is trying to delete in a common sub-tree. In this case, t1 may lock a node which t2 would need to access and vice-versa, thus leading to a deadlock.

Common incorrect answer

Many of you have mentioned that searching need not lock the root. This is not true in the presence of concurrent insert or delete along with search. Consider the following example, where t1 searches for 14. When t1 is in the node 12, another thread t2 inserts 15 and restructures the tree. Now t1 would not find 14 in the right sub-tree of 12, even though it exists as the parent of 12.



Q5

This question borrows from the concurrent list of Minor 1. Refer to that answer for more explanation. For skiplist, like the regular linked list, nodes between which actions take place at each level must be locked. In other words, insert-after(node) and delete-after(node), both lock node and then node.next. Search may proceed without locking.

Going down, insert retains the locks at each level, because whether insertion happens at a level is determined in the upward pass with coin-tosses. Once the insertion at a level is complete, the locks can be released for that level in the second pass on the way up. Because the lock order is the same left to right and top to down, there would be no deadlocks.

Deletion can unlock similarly, but optimization is easy. With careful implementation, it is possible to unlock the level on the way down before proceeding to the next level, as the key at that level may be deleted immediately. This amounts to the key not being deleted from the skip list yet, it's only removed from its top levels. This is similar to the case that it never got raised up to that level on insertion (based on the coin-toss). Similarly, it is also possible to insert on the way down. An important advantage over balanced trees is that multiple operations can proceed in parallel on skiplists.

Q6

See at the end.

Q7

Both heaps contain their top priority elements at their root nodes, respectively. Also, as stated in the question, both heaps have exactly 2^h nodes and height h.

Merging algorithm:

Let H_1 and H_2 be the two heaps. Create a third heap, H_3 with $2^{(h+1)}$ elements as follows:

1) Make the last node e of H_2 the root of H_3
2) Make H_1 the left subtree of H_3, and H_2 its right subtree
3) Heapify(H_3), by bubbling e down as necessary

The algorithm is correct because both H_1 and H_2 have one element each at the last level. On joining, H_2 loses that extra node, and the last level node of H_1 remains on the extreme left of H_1, leaving the heap structure intact. Heapify fixes the comparison property.

As heaps maintain the last position, step 1 is O(1). Making references to roots of H_1 and H_2 also takes O(1). Heapify takes time proportional to the height of the heap, which is O(height of the heap), and the height is h+1. Note that if heaps are kept in arrays, there may be a copy of H_1 and H_2 required. That would not be O(h).

Q8

Smallest node count for height h: S(h) = S(h-1) + S(h-2) + 1. S(0) = 1, S(1) = 2
This is fibonacci series like: 1, 2, 4, 7 .. (Exact formula given in class, but not expected for the test.)
Largest node count for height h: L(h) = 2L(h-1) + 1
This is 2^h-1: 1, 3, 7, 15 ..
A height h AVL tree with most imbalance and n nodes => n(h) = 1 + S(h-2) + L(h-1) for most imbalance
Now 20 = 1 + S(2) + L(3) = 1 + 4 + 15.

Thus the shorter side may not have fewer than 4 nodes, meaning the left tree has no fewer than 4 nodes and no more than 15. Given that all the left subtree nodes are smaller than the root, the root may never have the numbers 1-4 and 17-20.

Q9

Assume a unique key in each node (as is usual in a balanced tree).We have given only pre-order traversal of red black tree and we need to re-construct the tree (without colors). Pre-order traversal of tree is ROOT-LEFT SUBTREE-RIGHT SUBTREE. Further, since a Red Black trees is a binary search tree, the elements in the left subtree are less than the root and those in the right subtree are greater. Here is a solution.

```
reConstruct(preOrder, start, end)
{
   if start==end then return preOrder[start]
   if start<end then return null

   root=preOrder[start] // Root is listed first in the pre-order traversal of any tree.

   //split holds the first element that is greater than root in preOrder[start+1 to end]
   //The left subtree ends and right subtree begins there by BST property
   //If no element is greater than root key then split=end+1

   split = end+1
   for(i=start+1 to end)
      if(preOrder[i] > root)
         split=i

   root->left = reConstruct(preOrder, start+1, split-1)
   root->right = reConstruct(preOrder, split, end)
}
```

Time complexity - O(n $\log_2 n$). T(n) = T(split) + T(left-part) + T(right-part). Split take O(n) time starting at the left to linearly find the first element greater than the root. Since the tree is known to be R-B tree, the left and the right subtrees are roughly balanced. (You should do a more precise counting.) Hence T(n) ~ O(n) + 2 T(n/2). Thus time is O(n log n).

Other possible solutions that we have allowed are:-

1) Construct the tree from scratch carefully by inserting one node at a time in a BST fashion always starting the scan from the root. This works because the root is listed first and inserted first. Later inserts are in the subtrees. This holds recursively at all levels. This is also an O(n log n) solution because there are n insertions into a tree that never gets taller than O(log n).

2) One nice, O(*n*), solution is to maintain a "range" for each subtree. For example, find the minimum and the maximum keys in the entire input list, Call them min and max, respectively. Root has keys in the range min:max. The left subtree has keys in the range min:root and the right in the range root:max. This process can be recursively followed for each node, progressively restricting the range. The input list is processed in one pass and based on the range of the subtree, the keys are put in the correct tree. Something like this (note that there is a single index advancing continuously, it's not a local variable):

```
construct(min, max):
   key = PO[index];
   if( index >= PO.length  || key < min || key > max )
      return NULL;

   index++;
   subroot = new Node(key);
   subroot->left = construct( min, key);
   subroot->right = construct( key, max);
   return subroot;
```

It takes O(*n*) to find min and max. The rest of it is also O(*n*).

Q10

To find the depth of the tree in a non recursive manner we can do a Breadth First Traversal (also known as Level Order Traversal) of the tree.
Assigning the root node to have a depth of 0. As we go to each subsequent level of the tree,we increment the counter maintaining depth. Finally, the depth of the tree will be equivalent to the last level we encounter.

We push nodes with their depth into a queue. When we pop any element from queue, we insert its children into the queue with increased depth. Eventually the last element we pop will have depth equivalent to the depth of the tree.

Other traversal schemes like DFS are also correct if a suitable global counter for height is maintained correctly and the depth of all leaves is computed correctly. However, such search schemes need to be implemented non-recursively.

```
find_depth(root)
      let Q be queue.       // queue will contain (node, depth) pairs
      Q.enqueue((root,0))
      max_depth = 0

      while (Q is not empty)
           v , curr_depth =  Q.dequeue()
           max_depth = max(max_depth, curr_depth)
```

```
          //processing all the children of v
          for each child w of v in Binary Tree B
                // children of v are one level deeper than it
                Q.enqueue((w, curr_depth + 1))
return max_depth
```

Question 6

Solution

In a 2-4 tree, all leaves have the same height. The height is the maximum when all internal nodes have degree 2 and the minimum when they have degree 4.

a) The shortest tree has 4-nodes at all levels, with 3 keys per node.

$$\sum_{i=0}^{h} 3 * 4^i = n \Rightarrow 3\frac{4^{h+1}-1}{3} = n \Rightarrow h = \log_4(n+1) - 1 \geq \lfloor 0.5 \log n \rfloor, \forall n > 0 \tag{1}$$

One way to see that the last inequality holds is that a 2-4 tree with height h cannot hold more than $4^{h+1} - 1$ keys. $0.5 \log\left(4^{h+1} - 1\right) < h + 1$

b) The tallest 2-4 tree has 2-nodes at all levels, with only one key per node. This means:

$$\sum_{i=0}^{h} 2^i = n \Rightarrow 2^{h+1} - 1 = n \Rightarrow h = \log(n+1) - 1 \leq \log n, \forall n > 0 \tag{2}$$

c) To find a key in a 2-4 tree, the maximum number of nodes to be examined is equal to the height of the tree+1. Thus, from part b we can conclude that the maximum number of nodes to be examined is at most $\log_2(n+1)$.

For each node examined, the maximum number of comparisons happen when the number of keys in a node is the maximum, which is 3 for 4-nodes. A binary search approach for finding which child to take compares the requested key with first the middle key, and then with the left or the right key, resulting in a maximum of 2 comparisons per node.

Therefore, the total number of comparisons is no more than $2\log_2(n+1)$. It is actually $2\log_2(n)$ because if all nodes require two comparisons there are at least log(n) nodes that have more than one key each. See below for a more precise argument.

Extra credit

This extra credit question was not posed correctly. The intent of the question was to find the height of the tree for the worst case nodes, which involves the recognition of the worst nodes. 3-nodes are the worst nodes, because they involve 2 comparisons but eliminate only $\frac{2}{3}^{rd}$ of subtrees. 2-nodes and 4-nodes are more efficient at unproductive path eliminations. 2-nodes eliminate half the paths with 1 comparison, and hence $\frac{3}{4}^{th}$ with 2 comparisons. 4-nodes also eliminate $\frac{3}{4}^{th}$ with two comparisons. (You only need to compare the middle key and then one of the other keys in the worst case.)

On the other hand, the tight bound for general trees can also be computed using the same idea. Such a tree will be populated mostly with the worst case nodes for height – 2-nodes, interspersed with worst-case nodes for elimination – 3-nodes. In particular, there need be only one 3-node per level (to maximize the height) and the worst-case traversal would always find that node to traverse to. This is possible if the single 3-node at each level is the child of a 3-node from the previous level. Now, such a tree has two comparisons at each level and the height is close to $\log n$. Let's see the exact computations:

Let us denote the total number of keys by n and the number of keys in level i by $N(i)$. When all the nodes are 3-nodes, we get:

$$N(i+1) = 3 * N(i)$$

This gives:

$$n = \sum_{i=0}^{h} N(i) = N(0) + N(1) + ... + N(h) = 2 + 2.3 + 2.3^2 + ... + 2.3^h$$

$$\therefore \quad n = 2\frac{3^{h+1}-1}{3-1} \implies h = \log_3(n+1) - 1 = \frac{\log_2(n+1)}{\log_2 3} - 1 \approx 0.63\log_2(n+1) - 1$$

Thus, the total number comparisons in this case are $2 * (h+1)$, i.e. $1.26\log_2(n+1)$ (less than $1.3\log_2(n)$).

Worst-case

The worst case tree is as follows:

Denoting the number of nodes at level $i \in [0, h]$ by $nodes(i)$, we get:

$$nodes(i+1) = 3 + (nodes(i) - 1) * 2$$

i.e. one node at level i is a 3-node (thus has 3 children) and all the other nodes at level i are 2-nodes.

In terms of keys, this becomes (where we represent the number of keys in level i by $N(i)$):

$$N(i+1) = 4 + (N(i) - 2) * 2 = 2 * N(i)$$

For $i = 0$, i.e. the root node, we need it to be a 3-node for maximizing the number of comparisons. That is, $N(0) = 2$. Thus, we get the following expression for the total number of keys:

$$n`1` = \sum_{i=0}^{h} N(i) = N(0) + N(1) + ... + N(h) = 2 + 2^2 + ... + 2^{h+1}$$

$$\therefore \quad n = 2^{h+2} - 2 \implies h = \log_2(n+2) - 2$$

Thus, the total number of comparisons in this worst case are tightly upper bounded by $2*(h+1)$, i.e. $2*\log_2(n+2)-2$.