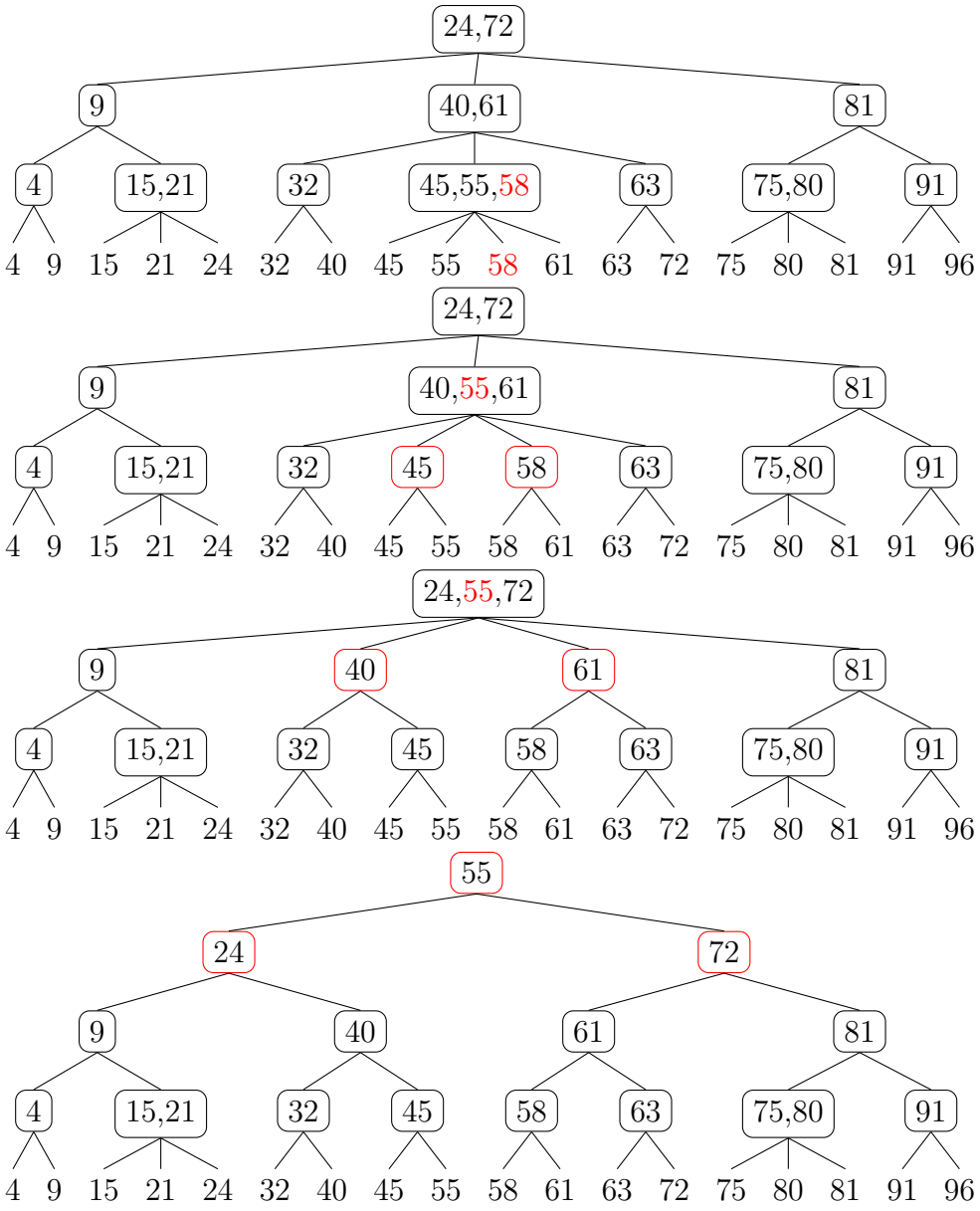
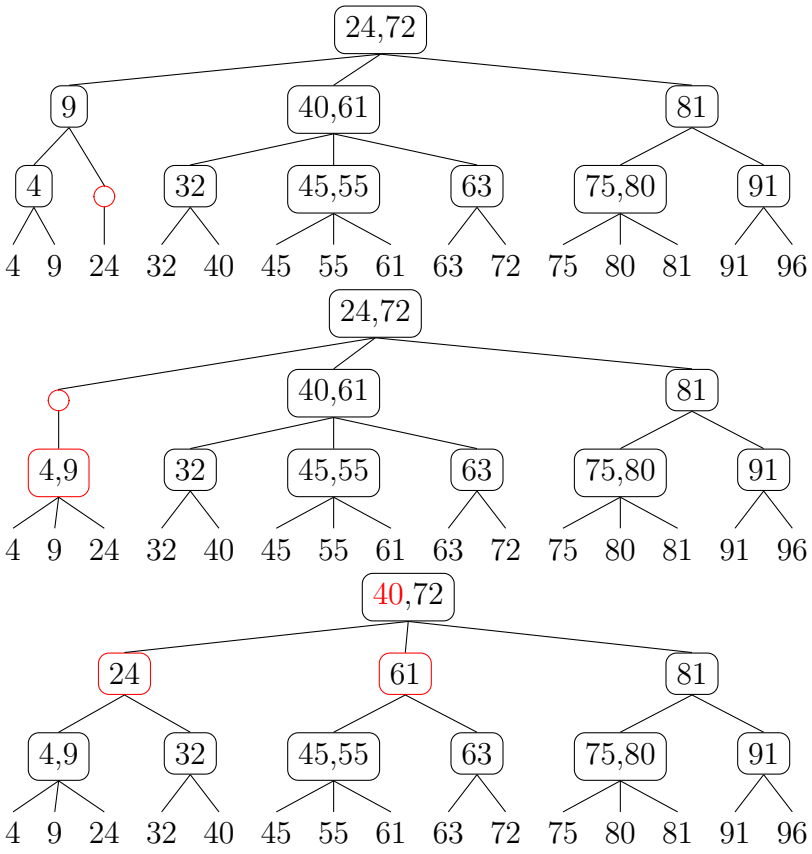
Homework 5, Solution set

1.a.



b.



2. We use the following recursive procedure: It the node is a leaf, add the item stored at the leaf to the output. Otherwise, recursively process the node's subtrees in left to right order.

Sort23Tree(v)
if v is a leaf **then** add the item at v to the end of the output
else
  for (each child w of v in left to right order) **do** Sort23Tree(w)
  end for
end if

Clearly, this takes time $O(\text{number of internal nodes} + n)$. It remains to argue that there are only $O(n)$ internal nodes.

To see this bound, we argue as follows. Let the depth of the 2–3 tree be h. There are at most $n/2$ internal nodes at depth $h-1$, $n/4 = n/2^2$ at depth $h-2$, ..., at most $n/2^i$ at depth $h-i$, ..., and 1 at depth 0. This is a total of at most $n/2+n/4+\ldots+n/2^i+\ldots+1 = n-1$ internal nodes.

3. a. The depth will increase exactly if every node on the path from the root to the parent of the new item has three subtrees. For the initial insertion causes the parent of the inserted node to have four children. Now, each resulting split causes the parent of the split node to gain a fourth child, and it will split in turn. This process propagates to the root, and the split of the root increases the depth of the 2-3 tree by one.

However, if there is a node with just two children on this path, then when and if the

splitting process backs up to this node, the node will acquire a third child, but the splitting will not proceed further.

b. The depth will decrease exactly if for every node on the path from the root to the parent of the new the sibling of each node on this path has two children. For the initial deletion causes the parent of the deleted node to have one child. As its sibling has just two children, the only option is to join these nodes, causing their parent to now have just one child. This process propagates to the root: Each join causes the parent of the joined node to lose a child, reducing it to having one child. As its sibling has just two children, the only option is to join the parent and its sibling. The final action is to remove the root as it has just one child, thereby reducing the height of the 2–3 tree by one.

However, if there is a node with three children on this path, then if and when the joining process backs up to this node, even if the node loses a child, it will still have two children so the joining will not proceed further. Likewise, if there is a node on the path with an adjacent sibling with three children, then if and when the joining process backs up to this node, it will lose one child but will then take a child from this adjacent sibling, leaving both it and its sibling with two children. Again, the joining process will not proceed further.

4. If the delete queries have the form Delete(ht), we use a 2–3 tree indexed by height to store the experiments. To support the count queries, at each node v, we will keep an additional field, $v.num$, which will hold the total number of leaves in the subtree rooted at v. To obtain the total number of experiments in the height range $[h_1, h_2]$, we perform searches for h_1 and h_2 in the h-tree, and sum up the numbers for all subtrees between these two paths, plus 1, including the righmost leaf only if it stores an experiment of height h_2 (the other possibility is that it stores an experiment of greater height). The running time of this query is $O(\log n)$, for it traverses two path in the h-tree.

Aside from maintaining the num fields, insertions and deletions will cost $O(\log n)$ time. For these queries, we update the $v.num$ fields on the traversed path as follows.

1. If the only change to the structure of the tree is to add (or delete) a leaf ℓ, for every ancestor v of ℓ, we add (or subtract) 1 from $v.num$.

2. If we join two subtrees rooted at v_1 and v_2, for the new node v, $v.num$ is given by $v.num = v_1.num + v_2.num$.

3. If we split a node v, with children u_1, u_2, u_3, u_4, into v_1 and v_2, where v_1 receives u_1 and u_2 as its children, and v_2 receives u_3 and u_4, then $v_1.num = u_1.num + u_2.num$, and $v_2.num = u_3.num + u_4.num$.

4. If we transfer a subtree rooted at u from a subtree rooted at v_1 to a subtree rooted at v_2, then $v_1.num \leftarrow v_1.num - u.num$ and $v_2.num \leftarrow v_2.num + u.num$.

As this adds an $O(1)$ cost to the processing of each node on the traversed paths, the overall cost of each query is still $O(\log n)$.

To be able to perform Delete(id) queries, we will use two 2–3 trees to store the experiments. One 2–3 tree, called the id-tree, is indexed by id, the other, called the h-tree, is indexed by height. The leaves hold the experiments, which are cross-linked, so we can go

between the leaf nodes in the two 2–3 trees storing the same experiment in $O(1)$ time. Alternatively, we could keep pointers at the leaves to a common record, where the record has pointers back to the two leaves.

To insert a new experiment, we insert it in both 2–3 trees, cross-linking the two leaves when they are inserted. This takes $O(\log n)$ time per tree, and therefore $O(\log n)$ time in total.

To delete an experiment, which will be identified by id, we delete it in the id-tree, and when the leaf is found, we access the corresponding leaf in the h-tree, allowing us to also perform the deletion in the h-tree, starting from the leaf node to be deleted (i.e. there is no need for a search in the h-tree).

The count queries are performed as before.

Thus the cost per query is still $O(\log n)$.