**CS2040S: Data Structures and Algorithms**

# Discussion Group Problems for Week 6

*For: February 13–February 17*

## 1 Check in and PS4

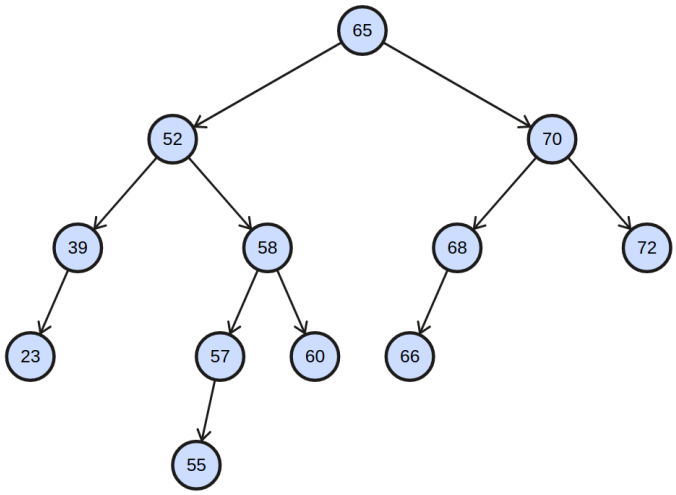
Discuss questions, if you have any, with the tutor and the rest of the class, about the material and content so far.

**Solution:** Your goal here is to find out how the students are doing. You may ask each of them to send you (perhaps anonymously) ONE question before tutorial about something they are confused by. For questions that make sense to answer as a group, ask the students to explain the answers to each other. (For questions that are not suitable for the group, offer to answer them separately.) You can also discuss some of the common mistakes listed in the grading scheme.

## 2 Problems

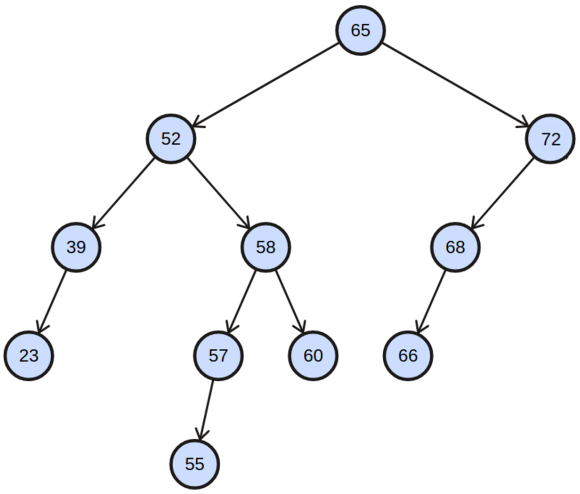
**Problem 1. Trees Review**

The diagram below depicts a BST.

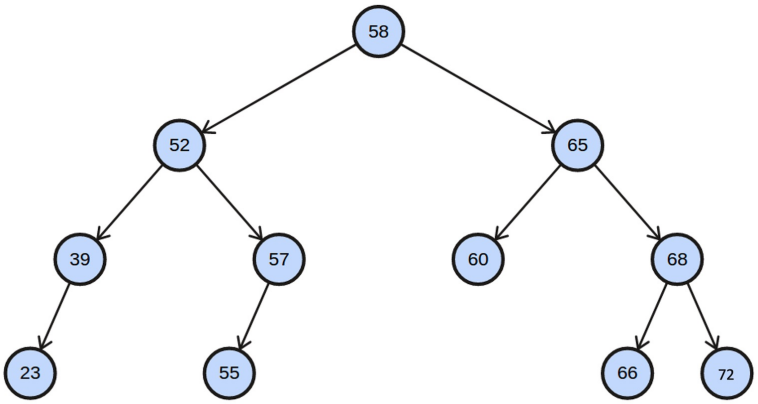



**Problem 1.a.** Trace the deletion of the node with the key 70.

**Solution:** To delete node 70, we first get its successor, which is node 72, copy the value over and delete it. We get the following tree.




Note that after the deletion, the subtree rooted at node 72 is now imbalanced. Suppose this is an AVL tree. A left-left rotation is required for rebalancing. Then, the tree rooted at node 65 becomes imbalanced. A left-right rotation is required. In total, 2 sets of rotations are performed. The following shows the final configuration of the AVL tree.




**Problem 1.b.** Identify the roots of all maximally imbalanced AVL subtrees in the original tree. A maximal imbalanced AVL tree is one with the minimum possible number of nodes given its height $h$.

**Solution:** All nodes are the roots of maximally imbalanced AVL subtrees. In particular, a maximal imbalanced AVL tree has all its subtrees to be maximally imbalanced. This is simply a consequence from the fact that an AVL tree with the minimum possible number of nodes with height $h$ has two subtrees with minimum possible number of nodes with height $h-1$ and $h-2$, namely $S(h) = S(h-1) + S(h-2) + 1$.

**Problem 1.c.** During lectures, we've learnt that we need to store and maintain height information for each AVL tree node to determine if there is a need to rebalance the AVL tree during insertion and deletion. However, if we store height as an `int`, each tree node now requires 32 extra bits. Can you think of a way to reduce the extra space required for each node to 2 bits instead?

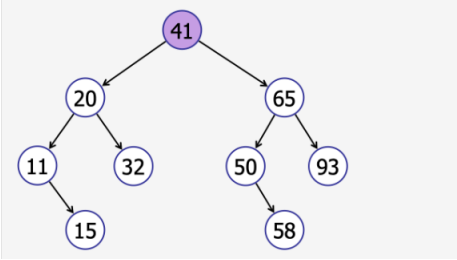
**Solution:** Instead of storing the height, we can store and maintain the balance factor for each node. Balance factor is equal to the difference between the left and right subtrees of a node. For AVL trees, each node can have balance factor of -1, 0 or 1, which only requires 2 bits.

**Problem 1.d.** Given a pre-order traversal result of a binary search tree $T$, suggest an algorithm to reconstruct the original tree $T$.

**Solution:** Given a sequence $A[1..n]$, the algorithm of reconstuction is given as,

1. Set the key of the root node of the tree to be the first element (i.e $A[1]$)
2. Find the position of the first element less than this value(noted as $idx_1$), and the position of the first element larger than this value(noted as $idx_2$)
3. Recurse on both $A[idx_1...(idx_2 - 1)]$ and $A[idx_2...n]$ to have two BST
4. Set the left child of root to be BST returned from first sequence and right child to be BST returned from the second sequence

For illustration, we use the following BST and its pre-order traversal sequence as an example.




We have the sequence: $41, 20, 11, 15, 32, 65, 50, 58, 93$. The first element should be the root of the tree (41). All elements from index $2 - 5$ are nodes rooted at 20 which is the left child of root. All elements from index 7 onwards are nodes rooted at 65 which is the right child of root. So if we recursively construct left tree and right tree we can finally reconstruct the original tree.

**Problem 2. Iterative BFS and DFS**

During lecture, we've learnt how to do tree traversal in various ways. For this question, we'll focus on DFS and BFS. Since you already know how to use DFS and BFS to traverse a tree recursively, can you propose a way using non-recursive DFS and BFS to traverse a tree? Write

your answer in the form of pseudocode.

**Solution:** Recall that Stack is LIFO (Last-In-First-Out) while Queue is FIFO (First-In-First-Out). For DFS, we can actually use stack to do it non-recursively. Specifically, we can first push the root of tree into Stack, perform the following operation repetitively until Stack is empty:

1. pop from Stack and get the popped node and that's the node you're visiting.
2. push all of its children into that Stack

When the Stack is empty, all the nodes have been visited and we have finished the DFS traversal. Similarly, for BFS, all the operations are the same except that now we use Queue instead of Stack to store all the nodes.

**Problem 3. Chicken Rice**

Imagine you are the judge of a chicken rice competition. You have in front of you $n$ plates of chicken rice. Your goal is to identify which plate of chicken rice is best.

**Problem 3.a.** A simple algorithm:

- Put the first plate on your table.
- Go through all the remaining plates. For each plate, taste the chicken rice on the plate, taste the chicken rice on the table, decide which is better. If the new plate is better than the one on your table, replace the plate on your table with the new plate.
- When you are done, the plate on your table is the winner!

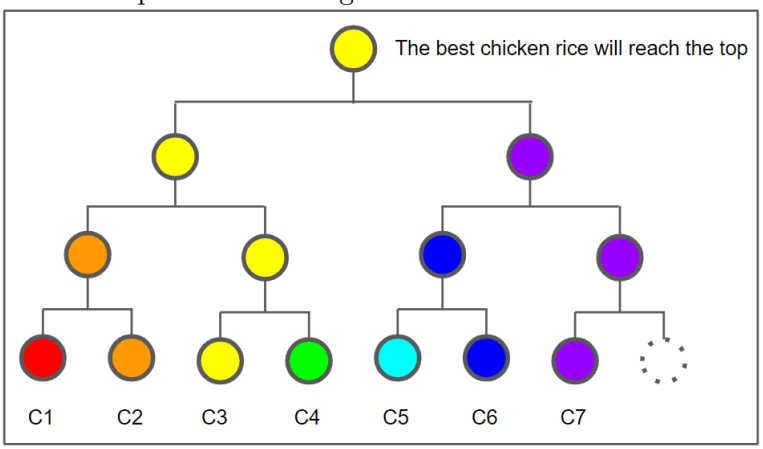
Assume each plate begins containing $n$ bites of chicken rice. When you are done, in the worst-case, how much chicken rice is left on the winning plate?

**Solution:** There would only be one bite! In the worst case, the first plate on the table is the best plate of chicken rice already, and every comparison thereafter, you keep holding onto the same plate of chicken rice, and compare it to the remaining $n-1$ plates, so you end up taking $n-1$ bites out of the same plate.

**Problem 3.b.** Oh no! We want to make sure that there is as much chicken rice left on the winning plate as possible (so you can take it home and give it to all your friends). Design an algorithm to maximize the amount of remaining chicken rice on the winning plate, once you have completed the testing/tasting process. How much chicken rice is left on the winning plate? How much chicken rice have you had to consume in total? (Give a tight asymptotic bound!)

**Solution:** Use a tournament tree! Start with a perfectly balanced binary tree with each plate of chicken rice at a leaf. Work your way up the tree comparing plates, until you get to the root. That's the best chicken rice, and it only took $\log(n)$ bites off that plate. In total for each comparison you will only need to consume 2 bites, and in total you need to make $\leq 2n$ comparisons, so you only need to make at most $O(n)$ bites.

Below is an example with 7 plates of chicken rice. Notice that any plate of chicken rice only needs to be compared with another plate at most $\log n$ times.



**Problem 3.c.** Now I do not want to find the best chicken rice, but (for some perverse reason) I want to find the median chicken rice. Again, design an algorithm to maximize the amount of remaining chicken rice on the median plate, once you have completed the testing/tasting process. How much chicken rice is left on the median plate? How much chicken rice have you had to consume in total? (Give a tight asymptotic bound. If your algorithm is randomized, give your answers in expectation.)

**Solution:** Let's try to run QuickSelect. First, notice that in one level of QuickSelect (when the subarray is size $n$), almost all the plates in the subarray have one bite eaten from them; the unlucky "pivot" plate has $n-1$ bites eaten. If you choose the pivot at random, then the median plate has an expected cost of $(1/n)n + (1-1/n)1 \leq 2$. So at every level of recursion, there are at most two bites consumed from the median plate, in expectation. Since the recursion will terminate in $O(\log(n))$ levels (with high probability and in expectation), we conclude by linearity of expectation that the median plate only has $O(\log(n))$ bites consumed. In total, you have consumed $O(n)$ bites of chicken rice.

Another answer is to use an AVL tree. Inserting each plate into an AVL tree takes $O(\log n)$ comparisons. At that point, you can find the median, e.g., by doing an in-order traversal of the AVL tree. (Or, in class, we will see how to find order statistics in an AVL tree.)

In fact, using a randomized algorithm, this can be solved with only $O(\log\log(n))$ bites to the median plate, but that is much, much more complicated.

**Problem 4. Economic Research**

You are an economist doing a research on the wealth of different generations in Singapore. You have a huge (anonymised) dataset that consists of ages and wealth, for example, it looks something like:

```
1  24  150,000
2  32   42,000
3  18    1,000
4  78  151,000
5  60  109,000
...
```

That is, each row consists of a unique identifier, an age, and a number that represents their amount of wealth.

Your goal is to divide the dataset into "equi-wealth" age ranges. That is, given a parameter $k$, you should produce $k$ different age ranges $A_1, A_2, ..., A_k$ with the following properties:

1. All the ages of people in set $A_j$ should be less than or equal to the ages of people in $A_{j+1}$. That is, each set should be a subset of the original dataset containing a contiguous age range.

2. The sum of wealth in each set should be (roughly) the same (tolerating rounding errors if $k$ does not divide the total wealth, or exact equality is not attainable).

In the example above, if taking the first five rows and $k = 3$, you might output `(3,1),(2,5),(4)`, where the age ranges are $[0, 30), [30, 70), [70, \infty)$ respectively, with the same total wealth of $151,000$.

Notice this means that the age ranges are not (necessarily) of the same size. There are no other restrictions on the output list. You should assume that the given $k$ is relatively small, e.g., 9 or 10, while the dataset is very large, e.g., the population of Singapore. Also note that the dataset is unsorted.

Design the most efficient algorithm you can to solve this problem, and analyse its time complexity.

**Solution:** This is actually just a weight selection problem. We are asking to find the $\frac{1}{k}, \frac{2}{k}, \ldots, \frac{k-1}{k}$ order statistics of the weighted sum.

Let's first talk about how to find a single weight order statistic. In $O(n)$ time, we can compute the total wealth of the population. We can then divide this by $k$ to figure out the "break points" that we are looking for. Thus, we might want to find the smallest ages with total wealth of at most $151,000$.

To do this, we can run a version of QuickSelect. First, use QuickSelect to find the median based on age, and then partition around the median. Next, you can sum the totals on the left and right halves, and decide on which side to recurse on. If your target is on the left, simply recurse on the left. If your target is on the right, then subtract from your target the total wealth of the left half. This partitioning can be done in $O(n)$ time.

When this finishes, you will find the person where the sum of wealth less than that person matches the original target.

You can repeat this for the $k-1$ targets, and then do one final partitioning to build your output lists. In total, this will take $O(nk)$ expected time.

Of course, instead of using QuickSelect to find the median, you can choose a random pivot to partition. And you can also find all $k$ "break points" at once, which is more efficient because you are not wasting a lot of time repeating the work you have already done partitioning the array.

If you use this more efficient version, you should be able to reduce the cost to $O(n \log k)$. Think of it this way: imagine that you first divide the array up into $k$ equal sized parts (not equal wealth, but in terms of the number of elements) by running QuickSort for $\log(k)$ levels of recursion. Each level divides the array in half, so at this point, you have $k$ different equal sized pieces, and you have spent $O(n \log k)$ time doing this.

Now, for each of your $k$ targets, figure out which piece it is in. This might require summing the wealth in each part ($O(n)$ time) and then searching for which of the $k$ pieces it belong in ($O(k \log k)$ time).

Now run the initial algorithm for each target on the correct subarray of size $O(\frac{n}{k})$, which takes $O(\frac{n}{k})$ time each. Since there are $k$ targets, the total time will be $O(n)$.

But in fact, you don't really need to do all this work. If you just run QuickSelect, and recurse on both sides if you have targets on both sides, then you get basically exactly that performance.

The recurrence you get in the latter case is something like:

$$T(n,k) = O(n) + O(k) + T(\frac{n}{2}, k_1) + T(\frac{n}{2}, k_2) \text{ where } k_1 + k_2 = k$$

In this case, the $O(n)$ is to partition, the $O(k)$ is to split the targets between the left and right halves of the partition, and the $T(...)$ parts are for the two recursive calls. That's a little tricky to analyze, but should end up with $O(n \log k)$.

An alternative that might be simpler to analyze is to divide the array in half by sum of wealth rather than by number of elements at each step.

Regardless of the approach, you can get $O(n \log k)$.

A reasonable question to ask is why $O(n \log k)$ seems like a reasonable answer, while $O(n)$ seems unlikely? (Think about what happens as $k$ gets bigger. As $k$ approaches $n$, the algorithm gets closer and closer to QuickSort, and so should converge to $O(n \log n)$.)

**Problem 5. Height of Binary Tree After Subtree Removal Queries**

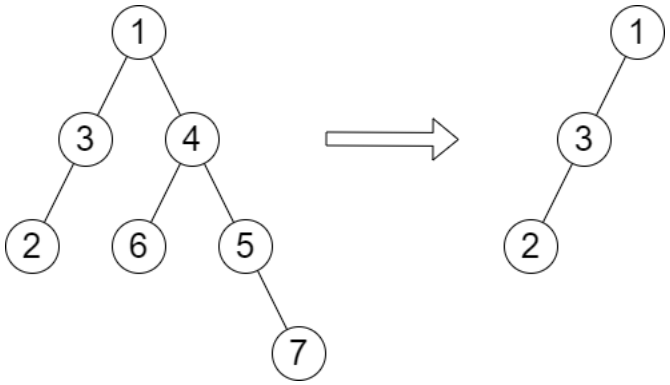
You are given the root of a binary tree with $n$ nodes. Each node is assigned a unique value from 1 to $n$. You are also given an array of queries of size $m$.

You have to perform $m$ **independent** queries on the tree where in the $i$-th query you do the following:

1. Remove the subtree rooted at the node with the value $queries[i]$ from the tree. It is guaranteed that $queries[i]$ will not be equal to the value of the root.
2. Return an array answer of size $m$ where $answer[i]$ is the height of the tree after performing the $i$-th query.

Note that the queries are independent, so the tree returns to its initial state after each query. The height of a tree is the number of edges in the longest simple path from the root to some node in the tree.

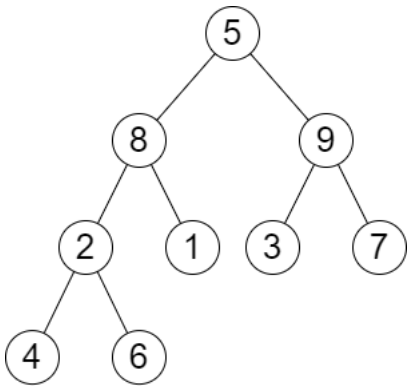
For examples:




Input: Tree data structure as left diagram above, queries = [4]

Output: [2]

Explanation: The diagram above shows the tree after removing the subtree rooted at node with value 4. The height of the tree is 2 (The path $1- > 3- > 2$).




Input: root $= [5, 8, 9, 2, 1, 3, 7, 4, 6]$, queries $= [3, 2, 4, 8]$
Output: $[3, 2, 3, 2]$
Explanation: We have the following queries:

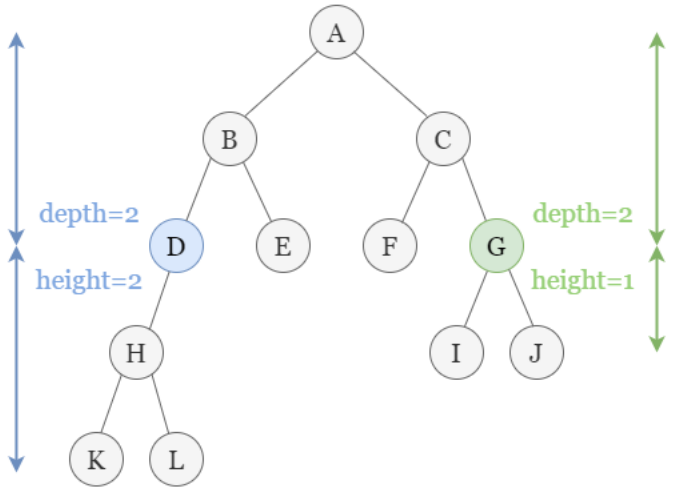
1. Removing the subtree rooted at node with value 3. The height of the tree becomes 3 (The path $5- > 8- > 2- > 4$).
2. Removing the subtree rooted at node with value 2. The height of the tree becomes 2 (The path $5- > 8- > 1$).
3. Removing the subtree rooted at node with value 4. The height of the tree becomes 3 (The path $5- > 8- > 2- > 6$).
4. Removing the subtree rooted at node with value 8. The height of the tree becomes 2 (The path $5- > 9- > 3$).

**Solution:** There are various ways to approach the problem and the naive approach is to compute the height of the tree every time we need to.
Before proceeding to discuss any solution, let's define terms to clarify their meanings:

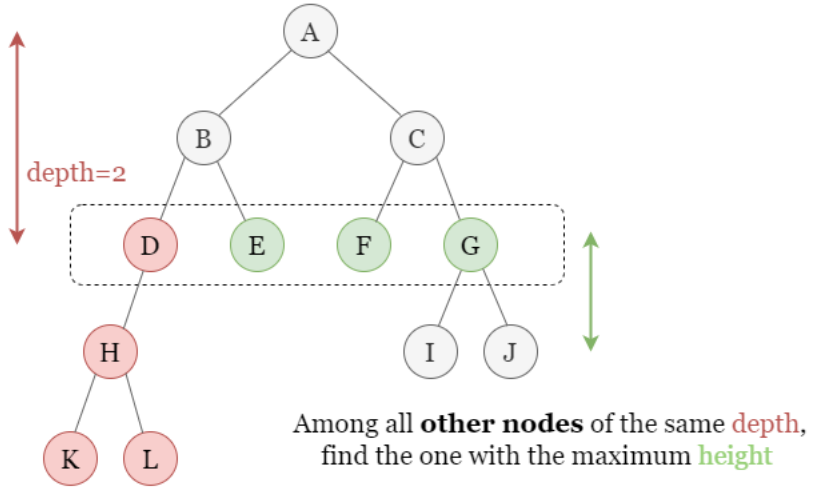
1. The **depth** of a node is the number of edges in the simple path from the root to the given node.

2. The **height** of a node is the number of edges in the longest simple path from the given node to some node in the subtree rooted at the given node.

You can check the diagram below to solidify your understanding.




One solution (and the optimal based on current tool set) is to preprocess the tree to find the remaining height after subtree removal for each node. We can compute the height and depth for each node. The height can be calculated via in-order traversal and depth via DFS. The time complexity for both operations is $O(n)$.



**(Continued)**

When we remove the subtree rooted at node $D$, we will look at all the nodes with same depth $d_0$ as node $D$. We can find the node with maximum height $h_0$ from the remaining nodes and sum it with $d_0$ to get the final result.

As the nodes in the tree are unique values from 1 to $n$, we can store the obtained result in an array with index being the node index and the corresponding value being the pair of height and depth of that particular node.

By iterating the nodes in in-order traversal, we can group all the nodes by their depth (query cost is $O(1)$ for each node) and find the nodes with maximum height among each group. Because we are only interested in the maximum height after subtree removal, it is either the maximum height or the second maximum height and hence we only need to store two instead of all for each depth.

We can hence compute the result for each node when trying to remove that node by the algorithm above. In other words, we now have an array to map the node index to its depth, another array to map the given depth value to the two nodes with maximum height in the given depth. To construct such two arrays requires $O(n)$ and to answer each query of height of subtree removal costs $O(1)$ with the aid of arrays. The overall time complexity will then be $O(n + m)$.

There is another method based on Euler Tour Tree (which is out of syllabus of CS2040S). You may find more details about the data structure on

https://courses.csail.mit.edu/6.851/spring07/scribe/lec05.pdf and

http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/17/Small17.pdf. The data structure provide a neat alternative solution and support for deleting multiple subtrees.