**CS2040S: Data Structures and Algorithms**

# Discussion Group Problems for Week 6

*For: February 17–February 21*

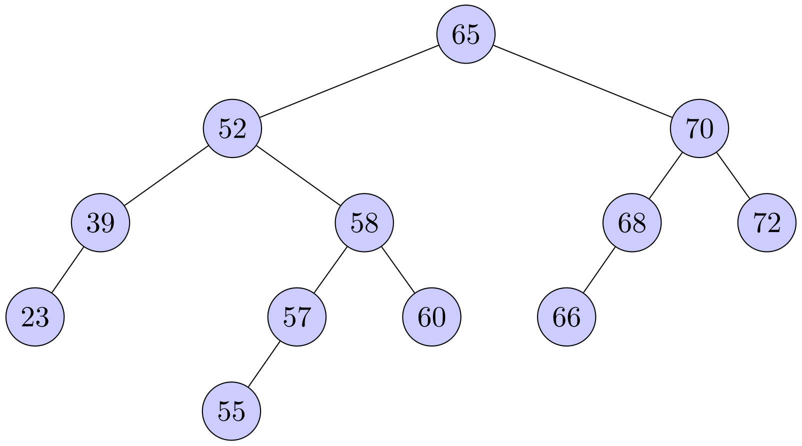
## 1 Check in and PS4

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

## 2 Problems

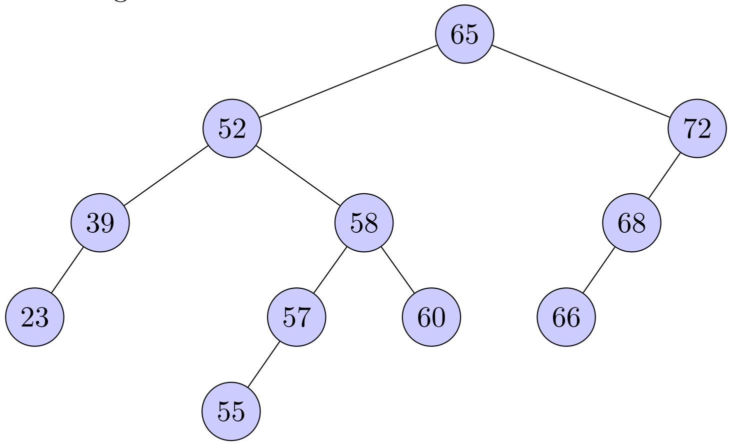
**Problem 1. Trees Review**

The diagram below depicts a BST.



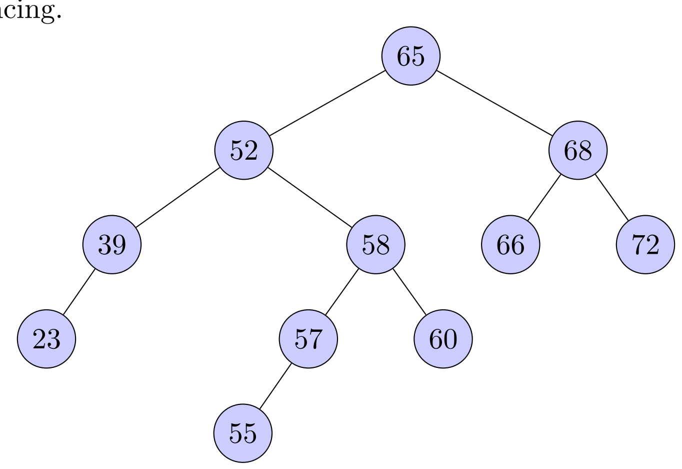
**Problem 1.a.** Trace the deletion of the node with the key 70.

**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.

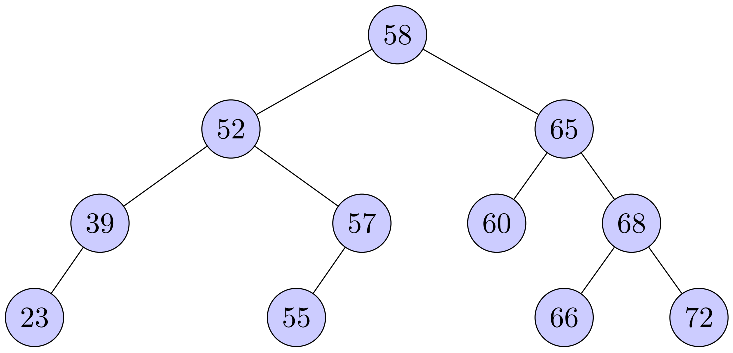


**Problem 1.b.** Suppose now that the BST is an AVL Tree. Is the resulting tree balanced? If not, balance the tree.

**Solution:** After the deletion, the subtree rooted at node 72 is now imbalanced. A left-left rotation is required for rebalancing.



**Solution: (Continued)** 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.c.** A maximal imbalanced AVL tree is an AVL tree with the minimum possible number of nodes given its height $h$. Identify the roots of all the subtrees in the original tree that are maximally imbalanced.

**Solution:** All nodes in the original tree are the roots of maximal imbalanced AVL trees. In particular, all subtrees of a maximal imbalanced AVL tree are maximally imbalanced as well. 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.d.** 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?

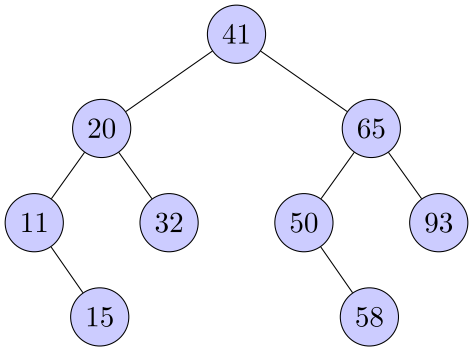
**Solution:** Instead of storing the height, we can store and maintain the balance factor for each node. Balance factor is equal to the difference between the left and right subtrees of a node. For AVL trees, each node can have balance factor of -1, 0 or 1, which only requires 2 bits.

**Problem 1.e.** 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 In-Order Tree Traversal**

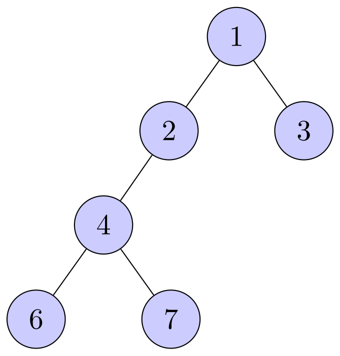
During lecture, we've learnt how to do tree traversal in various ways. For this question, we'll focus on In-Order Traversal. Since you already know how to use In-Order Traversal to traverse a tree recursively, can you propose a way using non-recursive In-Order Traversal to traverse a tree? Write your 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 In-Order Traversal, we can actually use a stack to do it non-recursively.
Specifically, starting from the root node of the tree, perform the following operations repeatedly:

1. Push the current node into the tree and visit its left child until the current node has no left child (i.e., the current node is `NULL`)

2. Then, pop a node from the stack and print its value before moving to the right child.

Repeat until the stack is empty **and** the current node is `NULL` (i.e., we have no more nodes to explore)
Example:



The steps for the above tree is as follows:

1. Starting from 1, visit 2, 4 and 6 and push all of them onto the stack. (Current Stack: `[6 4 2 1]`)

2. Then, we try to visit the left child of 6, but since 6 has no left child, the node has value `NULL`. Hence, we pop 6 from the stack and print its value. (Current Stack: `[4 2 1]`)

3. We try to visit the right child of 6, which results in `NULL` again, and thus pop 4 from the Stack and print its value. (Current Stack: `[2 1]`)

4. We visit the right child of 4, which is 7 and push that onto the Stack. (Current Stack: `[7 2 1]`)

5. Then, there's no left child so we pop 7 and print. (Current Stack: `[2 1]`) It has no right child as well, so we pop 2 and print. (Current Stack: `[1]`)

6. Pop 1 and print since 2 has no right child, and visit its right child, 3. Push 3, then since it has no left child, immediately pop it. (The stack is now empty, and the right child is `NULL` so we exit)

**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. However, you have a very poor taste memory! You can only compare the plates of chicken rice pairwise and you forget the taste of both plates immediately after comparing them.

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

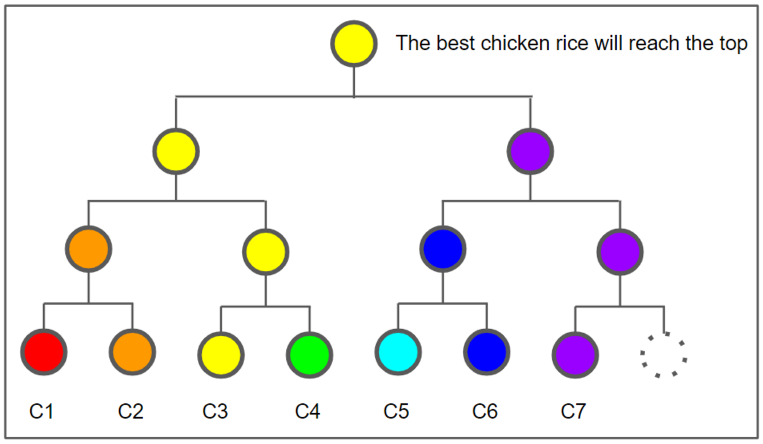
- Put the first plate on your table.
- Go through the remaining plates. Take a bite out of the chicken rice on the table and the chicken rice on the new plate to compare them. If the new plate is better than the one on the table, replace the plate on your table with the new plate.
- When you are done, the plate on your table is the winner!

Assume each plate contains $n$ bites of chicken rice in the beginning. 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.  Unification of Valeria**

The kingdom of Valeria is divided into numerous factions, each controlling a land ID that determine their territory. However, after years of conflict and political turmoil, the High Council has decided that the factions must be merged into K dominions to restore stability.

You are a member of the High Council and have been tasked with merging the factions. You have been given the following dataset:

```
1  3  150,000
2  4  42,000
3  1  1,000
4  8  151,000
5  7  109,000
...
```

Each row of the dataset consists of a faction's unique identifier, a unique land ID and a number that represents their power level.

Your goal is to divide these factions into $k$ dominions such that each dominion roughly has the same total power level to make sure no dominion can dominate. Furthermore, we want the factions within a dominion to fall under one contiguous range of land IDs so they are neighbouring. This range cannot overlap with another group's range.

That is given a parameter $k$, you need to output a list of $k$ sets of unique identifiers $(A_1, A_2, \ldots, A_k)$, such that each set has the following properties:

1. All the land IDs in set $A_j$ should be less than or equal to the land ID of faction in $A_{j+1}$.
2. The sum of power levels in each set should be (roughly) the same (tolerating rounding errors if $k$ does not divide the total power level, 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 land ID ranges are $[0, 4), [4, 8), [8, \infty)$ respectively, with the same total power level of $151,000$.

Notice this means that the land ID ranges are not (necessarily) of the same size. Some factions have merged with others factions during previous conflicts due to which some land IDs could be missing. These IDs don't have to be accounted for in the range. There are no other restrictions on the output list. You should assume that the given $k$ will be relatively small, e.g., 9 or 10, but the dataset can still be very large, e.g., every small faction in Valeria. Also note that the dataset is unsorted.

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

**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.
We first begin by finding the total power level of all faction in $O(n)$ time. We then divide this by $k$ to get the target sum for each dominion.
The algorithm is as follows:

1. Start by picking a pivot based on land ID (either randomly or using QuickSelect to get the median of land ID in $O(n)$ time).

2. Partition the array based on the pivot, such that all factions with land ID $\leq$ pivot are on the left, and all factions with land ID $>$ pivot are on the right. This takes $O(n)$ time.

3. Next, you can compute the sum the power levels on the left part and right part (We include the pivot in the left partition). This takes $O(n)$ time.

4. If your target is lesser than the left sum, simply recurse on the left part. If your target is greater than the left sum, then subtract from your target the total power of the left half and recurse on the right part.

5. We finish when we find the faction where the sum of power level of factions with lesser than or equal to land ID of that faction matches the target value. This land ID is one of our breakpoints. The members of the first dominion are the factions with land ID less than or equal to the land ID of the faction you found. This breakpoint is found in $O(n)$ time by our QuickSelect process. It is also good to note that this is $O(n)$ **expected** time complexity if pivot is picked randomly.

6. Repeat this process for a target value of $2 * target$ . The members of the second dominion will be the factions with land IDs between previous breakpoint and this newly computed breakpoint.

7. Now, you can repeat this process for a target value of $3 * target, \ldots, (k-1) * target$ for the computing the remaining breakpoints.

8. In total, you are performing k-1 iterations of QuickSelect. This has the time complexity of $O(kn)$.

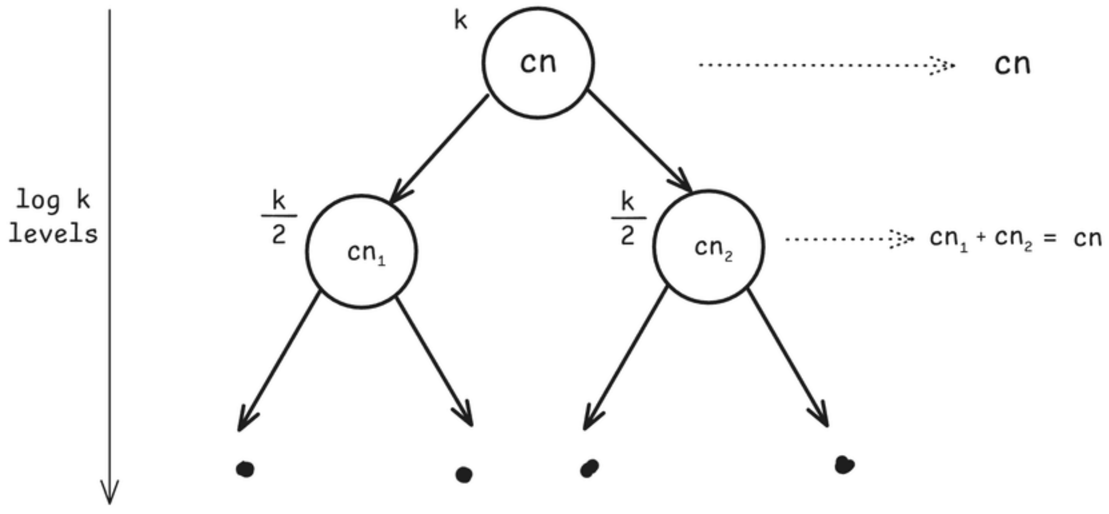
9. In the end, we can perform an algorithm similar to (k-1) pivot partition using all the breakpoints we found as pivots to find which set each faction belongs to. This takes $O(nlogk)$ time.

The overall time complexity is $O(kn)$ as the dominant term is $O(kn)$.

**Solution:** **(Continued)** Is this the most efficient algorithm? Is there a better algorithm that can solve it in one go?
Instead of finding the breakpoints one by one sequentially, we can find all the breakpoints through divide and conquer.
The idea is to find the $\lfloor \frac{k}{2} \rfloor$ breakpoint. Now, the breakpoints 1 through k/2 lie on the left part and k/2 to k-1 lie on the right part (as a consequence of the fact that land IDs on the left part will be smaller than pivot and land IDs on the right will be larger). Basically, we divided it into two more subproblems.
Remember that the ith breakpoint is the land ID of the faction where the sum of power levels of factions with land ID less than or equal to the land ID of that faction is equal to $i * target$.
To find this $\lfloor \frac{k}{2} \rfloor$ breakpoint, we can use the QuickSelect algorithm mentioned previously that computes the sum of each part to decide on which part to recurse on. This takes $O(n)$ time.
The recurrence relation looks something like this:

$$T(n,k) = T(n_1, \frac{k}{2}) + T(n_2, \frac{k}{2}) + O(n) \text{ where } n_1 + n_2 = n$$

Now, drawing the recursion tree for this recurrence relation:



The cost at each level is $O(n)$ and the height of the recursion tree is $\log k$. Why not $\log n$ ? Will we ever reach base case due to n before reaching base case due to k ? If it is possible, then it would imply we have a case such that $n = 0$ and $k > 0$. This would mean that we can create a dominion with 0 factions which is not allowed.
The total cost for finding the breakpoints using divide and conquer will be cost per level * height of tree = $O(n \log k)$.
Similar to before, we build the final sets using a (k-1) pivot partition using all the breakpoints we found as pivots. This takes $O(n \log k)$ time.
The overall time complexity is $O(n \log k)$.

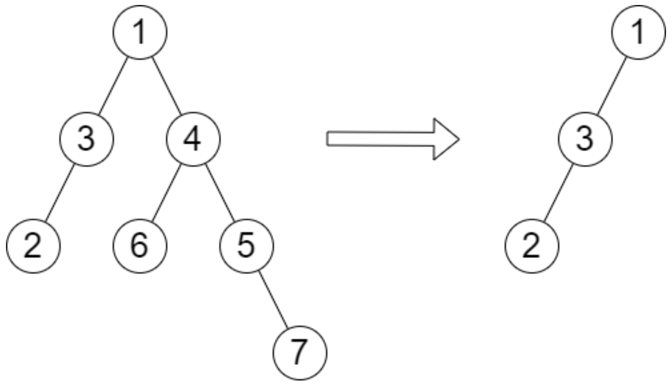
**Problem 5. (Optional) Height of Binary Tree After Subtree Removal Queries**

You are given the root of a binary tree with $n$ nodes where each node is assigned a unique value from 1 to $n$. You are also given an array of queries of size $m$, $queries$, where in the $i$-th query you do the following: 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.

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 a tree with $n$ nodes, design the most efficient algorithm you can to run all $m$ queries and output an array of size $m$, $answers$, where $answers[i]$ is the height of the tree after performing the $i$-th query.

Example 1:

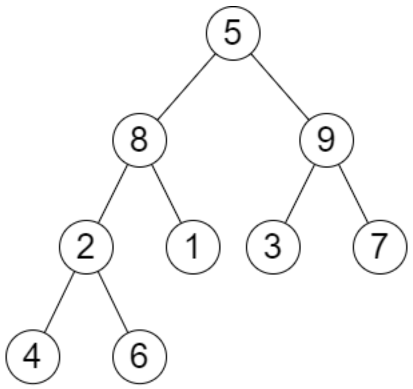


Input: Tree data structure as left diagram above, queries = [4]

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 $1->3->2$).

Example 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:

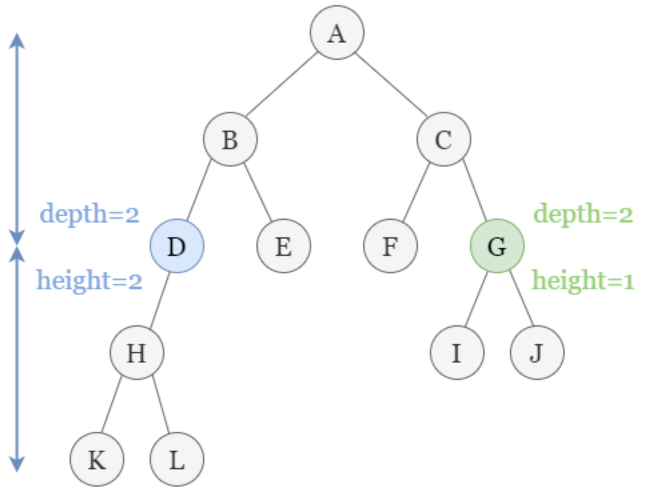
1. Removing the subtree rooted at node with value 3. The height of the tree becomes 3 (The path $5- > 8- > 2- > 4$).
2. Removing the subtree rooted at node with value 2. The height of the tree becomes 2 (The path $5- > 8- > 1$).
3. Removing the subtree rooted at node with value 4. The height of the tree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

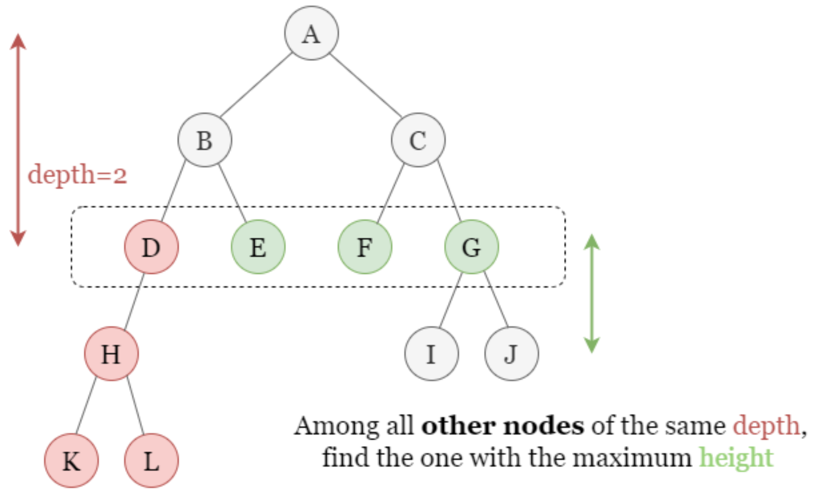
1. The **depth** of a node is the number of edges in the simple path from the root to the given node.

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 $O(n)$.


A
B
C
D
E
F
G
H
I
J
K
L
depth=2
Among all **other nodes** of the same depth,
find the one with the maximum height

**Solution: (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.