**CS2040S: Data Structures and Algorithms**

Discussion Group Problems for Week 6

*For: February 17–February 21*

# 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.

# Problems

## Problem 1. Trees Review

The diagram below depicts a BST.

65

52

39

23

58

57

55

60

70

68

66

72

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

**Ans:** When we are trying to delete the node with key 70, since the node has two children, we start by looking for the successor of node 70. In this case, the successor node would be 72. Then, we replace node 70 with node 72. The BST property still holds since all keys in the left sub-tree < key < all keys in the right sub-tree.

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

**Ans:** If this was an AVL tree, the tree would not be height-balanced. Specifically, node 72, which is now at where node 70 once was, would be left heavy. Thus, to rebalance the tree, we would need to do a right rotation on node 72. This would result in the following AVL tree.

A diagram of a network

AI-generated content may be incorrect.

As can be seen in the tree, node 65 would not be height balanced and is left heavy. Its left child, node 52, is right heavy. Thus, we would need to left rotate node 52, and right rotate node 65 afterwards. The final configuration of the tree is as shown and is height balanced.

A diagram of a network

AI-generated content may be incorrect.

**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.

**Ans:** All roots of the subtrees in the original tree are maximally imbalanced. This is due to the fact that in order for an AVL tree to have the minimum possible of nodes given its height *h*, it needs to have two subtrees that are of height *h – 1* and *h – 2*.

**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?

**Ans:** We can store the balance factor instead of the height in the node. Balance factor refers to the difference between the heights of its left and right subtrees. For AVL trees, the balance factor can only have values -1, 0 or +1. Since the balance factor can only be of 3 possible values, 2 bits is enough to store this information.

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

**Ans:** Given a pre-order traversal result of a binary search tree T, our algorithm to reconstruct the original tree can be as such:

1. Construct the root node using the first element in the array. This will be the root of its subtree.
2. Recursively construct on all the elements in the array that are smaller than its own key, i.e. call construct and pass in the Array[1, …, n] whereby n is the last element smaller than the current key. This will construct the left subtree of the current key. Set this tree to be the left subtree of the root node.
3. Recursively construct on all the elements in the array that are bigger than its own key, i.e. call construct and pass in the Array[n + 1, …, size – 1] whereby n is the first element that is bigger than the current key in the array. This will construct the right subtree of the current key. Set this tree to be the right subtree of the root node.
4. Return the current node.

For example, using the array [65, 52, 39, 23, 58, 57, 55, 59, 60, 61, 70, 68, 66, 72] as example, the first element in the array is 65, which is the root node of the tree. The last element in the array that is smaller than 65 is 61. Thus, elements 52 to 61 would make up the left subtree of 65. The first element that is larger than 65 would be 70 and the last element in the array is 72. Thus, elements 70 to 72 would make up the right subtree of 65. Thus, we can recursively construct on the left subtree and right subtree to construct the original BST.

**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.

We can make use of a stack to simulate the recursive call stack.

The algorithm is as follows:

1. Create an empty stack and set the current node to the root
2. Traverse the left subtree
   1. Push the current node to the stack
   2. Move to the left child
   3. Repeat until you reach a node with no left child
3. Visit the node
   1. Pop the node from the stack
   2. Process the node’s value (e.g. print it out or add to the resulting list)
4. Traverse to the right subtree
   1. Move to the right child of the popped node
5. Repeat the process until the stack is empty and the current node is null

Pseudocode:

1. Create an empty stack S
2. Initialise the current node as root
3. While the stack is not empty and the current node is not null:
   1. If the current node is not null:
      1. Push the current node to the stack
      2. Set the current node to the left child
   2. Else:
      1. Pop the node from the stack
      2. Process the current node (e.g. print or add to resulting list)
      3. Set the current node to the right child

## 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?

**Ans:** 1 bite left. In the worst-case, the winning plate of chicken rice is the first plate that is placed on the table. Then, we would have to compare with the other (n – 1) plates of chicken rice, which would result in (n – 1) bites of chicken rice being eaten from the first (winning) plate. Thus, there would only be 1 bite left on the winning 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!)

**Ans:** We can use a tournament tree to determine the winning plate. We compare each plate in pairs. Among each pair of plates, the winning plate goes on to compete with other winning plates within their original pairs. This process repeats until there is only one winning plate. Thus, there is bites of chicken rice left on the winning plate. Consumed total?

Thought process: There are n plates and n bites initially. On each level, only n/2 plates proceed forward to compete with other winning plates. Thus, we can keep dividing n by 2 until n becomes 1 (i.e. there is only 1 winning plate). The number of times we can dividing n by 2 is .

**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.)

**Ans:** We can make use of QuickSelect. Since where are n plates of chicken rice, we wish to obtain the plate of chicken rice in terms of taste. We start by randomly selecting a plate of chicken rice to be used as the pivot. Then we compare the pivot with the other plates of chicken rice, and sort them into 2 buckets, chicken rice that has a poorer taste and chicken rice that tasted better. After doing this, we are also able to know the index position of the plate of chicken rice that was used as the pivot. If the median plate index is smaller than the pivot’s index, we recurse on the buckets of chicken rice that has a poorer taste. Else, we would recurse on the other bucket of chicken rice that tastes nicer and set the median index to now be the plate, where k is the index of the initial pivot plate of chicken rice. In expectation, the median chicken rice plate would have bites of chicken rice left. Consumed total?

## 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*,...,Ak*), such that each set has the following properties:

* 1. All the land IDs in set *Aj* should be less than or equal to the land ID of faction in *Aj*+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*,*∞) 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!

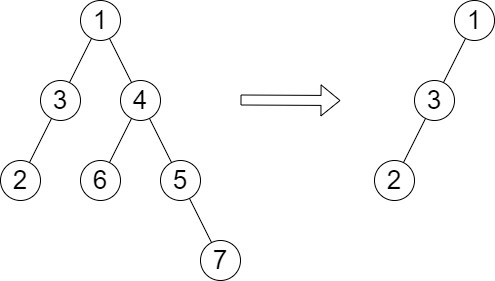
## Problem 5. (Optional) Height of Binary Tree After Subtree Removal Queries

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:

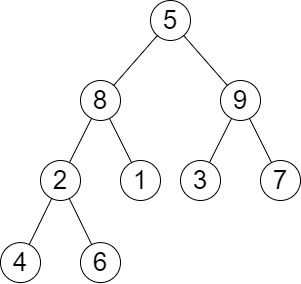


Input: Tree data structure as left diagram above, queries = [4]

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).