# CS2302 Lab 4 by Cesar Lopez

# Introduction

Viewing the problems given to us, we designed solutions for the Balance tree questions given. The balance tree was unique and something new to us students because we did not learn it in our previous CS classes and was very fun to work with. A balance tree is like a Binary Search tree, except nodes can hold multiple values based on the size we choose for our tree. The problems given were Height of tree, return a sorted list, minimum and max element in tree, number of elements at height D, print the elements at height D, return the number of nodes that are full, return the number of leaves that are full, and find the depth of given k.

# Problem 1 Design and Solution

To begin with this problem, I did not struggle as much as I thought I would, but I did overthink this problem a bit. The way I chose to tackle this problem was to iterate only to the left. Although I do believe that a tree can be bigger in height on the left side of the tree, I just personally assumed It would be even on all of it. The base case that I chose for my code was to check if we reached a leaf, if true, we return 1, therefore if we reach where the end of a tree is, we return 1, even if the tree is only a leaf. The next part that I did was make a counter and set it equal to 1 and we create a return statement that returns count + tree.child[0], which allows us to go to the left. After finishing and submitting my work, there are a couple things that I would change to help my code. To begin, I did not really need a count and could have just returned 1 + tree.child[0], and it would have done the same as my solution.

# Problem 2 Design and Solution

This problem required us to put our Balanced tree into a sorted list which was a bit troublesome are first. The base case caused no issues, we checked if the node we were at was a leaf, and if it was, we would use a for loop to go through the node and append element from 0 to the length. After this base case is where I had a couple issues. The first way I tried to tackle this problem was to use a for loop to allow us to iterate through the tree while appending after our recursive calls, the problem with this was that we would not be able to append our tree head. The next solution I did, which I used, was to iterate the for loop through T.item and append after our recursive calls, which worked, but not for the whole tree, which lead to me putting an extra recursive call that would go to T.child[-1]. The reason I had to add that extra call was because after printing my list, I noticed that I was missing all of the right side of the tree, that extra call would allow me to go to the right and allowed me to get a sorted list.

# Problem 3 Design and Solution

Finding min was not hard because the base cases were easy to create. The first base case checked if d was equal to 0. If d was equal to zero, we would return T.item[0] because 0 held our minimum value. Our next base case checked if we were at a leaf, if we did reach a leaf, we could no longer iterate through the tree, therefore we return T.item[0], which Is the smallest value in the tree. The last was our recursive call that call the method name and T was updated to T.child[0], which iterates to the left where our minimum values are, and we also remove 1 from d because that allows us to go down 1 level with our calls.

# Problem 4 Design and Solution

Finding max was very similar to finding min except instead of using T.item[0], we use T.item[-1] because -1 gives us the last value in the list, the largest. Our first case checked if d was equal to 0 and we would return T.item[-1] if true. Our next base case checked if T was a leaf and would return T.item[-1] if true. Lastly, our recursive call would call the method and update T to T.item[-1], which iterates to the right where the large values are, and we subtract 1 from d because that allows us to reach the base case if d is 0.

# Problem 5 Design and Solution

This problem wanted us to give the number of nodes at level d. First we create a hold value named count to hold the number of nodes. Our first base case checks if d is equal to 0, and if it is we use a for loop to iterate through T.item and update count by 1 for every node there is, and then we return count. Our next case checks if T is a leaf, and if it is, we return 0. And next we have our recursive call which is a for loop that iterates through the length of T.child, and updates count with recursive calls using T.child[for loop], and d-1. This update count to every count that we had before or 0 if we reach a leaf. After finishing updating count, we return count.

# Problem 6 Design and Solution

This problem requires us to print the nodes at depth d. First we created a base case that would check if d was equal to 0, and if true, we would use a for loop to iterate through T.item and print T.item[for loop]. Next was our recursive case which would use a for loop to iterate through T.item[for loop] and update d by -1.

# Problem 7 Design and Solution

This problem required us to find the number of full nodes that were in our balance tree. The first approach that we chose to take was to create a base case that checked if len(T.item) was equal to T.MAX\_ITEMS, and if true, we would return 1. If T was a leaf, we would return 0 because a leaf would turn into more nodes if it was full. Next, we create a number holder that holds the amount of max item nodes. We use a for loop to iterate through our nodes using recursive calls with T.child[for loop] and we add that to our hold. Lastly, we return our hold.

# Problem 8 Design and Solution

Here we were asked to provide the amount of T.leafs that are full. For this problem we used a nested if statement because first we checked if T is a leaf, and next we checked if T.item is equal to T.MAX\_ITEMS and then we return 1 if both are true. After, we create a hold that hold the amount of leaves that are full. Next, we have a for loop to iterate through the tree that allows us to get to our leaves with recursive calls and it adds up to our hold value. After that we return our hold.

# Problem 9 Design and Solution

This problem was the most time consuming because of how difficult it was and the amount of problem solving that had to be done. The first way I solved this was to iterate and return +1 for every level and return when found. The problem was that we could not return -1 if not found. The solution we finally ended with was to iterate through T.item and if we found T.item[for loop] is equal to k, we return 0, that is because it can be found in the first level which is 0. Next we check if T.isLeaf and if true, we return -1 because it was not found. Next we check if k is located towards the right side by comparing T.item[-1] and seeing if K was larger which made us iterate to the right. We created a hold value named depth and we set it equal to the right recursive call. We also check if it is equal to -1, and if it is, we return -1, and if not we return depth+1. Next recursive case we use a for loop to iterate through the rest of the tree, except of course the right most side. Here we check if K<T.item[for loop], which calls a recursive call if true. Then we check If our hold is 0 and if true we return depth+1. Our last case check is depth is less than 0 and if it is, we return -1, if not we return depth+1.

Graph is based on the size of the tree and uses all the problems given, this saved time instead of having lots of small graphs based on every problem.

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class. C.L