King and the Crisis

Moida Praneeth Jain 2022101093

Structures

struct node

This structure represents each node of a tree. The value of the node is stored in `val`, the pointer to the left node is stored in `left` and the pointer to the right node is stored in `right`. In addition to these, it also holds two integers `min` and `max`. Any children of the current node have to lie in this range for the tree to be a BST. This information shows whether another node could be a child of this node or not.

struct deq_node

This structure represents each node of a circular deque. It is a deq of the struct nodes above. The tree node is stored in `n`, the pointer to the next deq_node is stored in `next`, the pointer to the previous deq_node is stored in `prev` and the length of the circular deque is stored in `length`.

Functions

create_range_node: O(1)

This function is used to create a new tree node with no children. It takes in `val`, `min` and `max`, and returns a pointer to a tree node with these attributes. Since there are no loops in this process, the time complexity is constant, i.e, O(1).

ListToBst: O(n)

This function takes in a list (array of size n) and constructs a binary search tree from it. If the size of the array is 0, `NULL` tree is returned. If it is not 0, a deq is initialized. The root of the tree can have children having any value, so the min of the root is `INT_MIN` and the max of the root is `INT_MAX`. Such a node is initialized and enqued onto the deq. Now we set up a while loop to loop through the array of integers. During every iteration, we will deque a node from the deq. Then, we will check if the current integer in the array `arr[i]` fits within the range of the dequed node. Since the input is a level order traversal of the BST, enqueing every node that has been created ensures that the tree is constructed in the correct order. If `arr[i]` is less than the current node's value, then it must go to the left of it, provided it fits within the range, i.e, its value is greater than the `min` of the current node. If it meets this condition, then this node's minimum value would be same as `min`, and its maximum value would be the `val` of the current node, since it is to the left of the current node, hence all values must be smaller than it. This node is created and inserted, and then enqued onto the deq. Similarly, the case of `arr[i]` being greater than the current node's value and less than the `max` of the current node is handled. In case `arr[i]` fails to meet both these conditions, the current node does not have it as its child, and hence is dequed, and the index of the array is not incremented. Since each node can only be enqued and dequed once, each node is visited not more than twice, i.e, 2n. The time complexity is linear, i.e, O(n).

level_order: O(n)

This function takes a tree and prints it in level order. If the tree is `NULL`, then nothing is printed. If it is not `NULL`, a deq is initialized and the root node is enqued. Now we iterate until the deq is empty. In each iteration, we deque the head and print its value. If it has a left child, it is enqued, and if it has a right child, then it is enqued after that. This ensures that each level is printed from left to right. The current level is printed before the next level because all the nodes of the next level have been enqued after the nodes of current level. Since each node appears only once, the time complexity is linear, i.e, O(n).

ModifyBST: O(n)

We need to find the sum of all values smaller than or equal to the value of each node, and set this sum as each node's value. To do this, we can traverse the BST **inorder**, and take the cumulative sum while traversing it. Since the inorder traversal of a binary search tree is in increasing order, the sum until that point would include all and only those values which are smaller than the node's value. On adding this sum with the node's value, we get the desired modification to the BST. To find out the maximum sale, we can sum up each node's value after it has been modified. Since each node appears only once in an inorder traversal, the time complexity is linear, i.e, O(n).

free_tree: O(n)

This function takes in a tree and recursively frees it. First the left and right subtrees are freed so that the root can be safely freed without losing access to its subtrees. Since each node is freed exactly once, the time complexity is linear, i.e, O(n).

push: O(1)

This function takes in the head of a deq and a node of the tree, and then enques the node onto the deq. We can access the previous node of the head directly, and then insert the newly created node in constant time, i.e, O(1).

pop: O(1)

This function takes in the head of a deq and deques a node of the tree, and then returns it. We can access the node to deque directly as it is the next of the head, and then remove it from the deq and return it constant time, i.e, O(1).

is_empty: O(1)

This function takes in the head of a deq and returns 1 if the deq is empty and 0 if the deq is not empty. This is checked with the length of the deque, hence it occurs in constant time, i.e, O(1).

free_deq: O(n)

This function takes in the head of a deq and iterates through it while freeing each node. Since each node appears exactly once, the time complexity is linear, i.e, O(n).