# Explanation - Q4

Vyakhya Gupta
2022101104

**Structs defined:**

1. Node- It is the Tree node containing long long int key, and pointers to right and left nodes of the Tree. (ptrnode, Tree: Node*)
2. Qnode- It is the element of Queue. Contains a pointer to a Tree node and a pointer to the next element. (ptrq: Qnode*)
3. qn- It is the Queue struct storing front and rear (pointers to the first and last elements of the linked list queue). (Queue: qn*)
4. qinfo- It is the element of second queue which stores a pointer to a tree node, pointer to the next element in queue and long long ints min and max- which represent the range of values the children of that node can take.
5. infoq- It is the Queue struct that stores pointers to the front and rear of the qinfo linked list. (Range_queue: infoq*)

**Functions:**

1. **makenode** takes a long long int as argument and returns the pointer to a tree node which it mallocs and assigns the key as e, and left and right children as NULL.
2. **ListToBST** takes an array with level order traversal of a tree and its size as arguments and returns a pointer to the root tree node by creating a Binary Search Tree. First T points to the node created by creating a node with the first element of the array. Then a Range_queue rq is initialised for storing elements and their ranges as we insert them into the tree. The first element of the Range_Queue rq is malloced, with treenode as T (root node). Two variables max and min store the maximum and minimum elements in the array, so that the range of the root node is assigned as (min-1,max+1). This queue node is enqueued onto rq.
   Then the function iterates through the array containing level order traversal of the tree, and creates queue nodes for each element.

   If the value of the element does not lie within the range permissible for the children of the front element of the queue, the front element is dequeued and i is decremented. (This is because the given array is in level order traversal, so once we reach an element whose range doesn't satisfy, we can be sure that the particular node will not have any more children. i is decremented because we need to recheck the next queue node for the same array element.)

Else, if the element satisfies the range, we need to check whether to insert at left or right child position. If arr[i] is less than the front node of the queue's key value, but the front node's left child is already assigned, we dequeue that node, decrement i, and continue. (This would mean that the element is also less than the front element's left child and being a level order array, the right child must be NULL.) If the left child is not already assigned, we assign the node of arr[i] as the left child of the queue's front. The min of this node is the same as parent (rq->front->min), but the max now becomes the key value of the front element. Then this queue element is enqueued.

Since it is a level order traversal, as soon as the right child of a node is assigned, it no longer has any scope for further children nodes, and is thus dequeued.

When the loop is done, rq is freed and the head root node T is returned.

3. **ModifyBST:** Takes a binary search tree as input and modifies the tree such that each Node's Value gets updated to the sum of all the values smaller than or equal to the current node.
   static long long int is used to store the current sum (as the function is recursive, static is used so that the value of the sum doesn't get reinitialized to 0 in every function call). The base case is when the node becomes NULL, the function returns. It follows an inorder logic, which first modifies the left subtree, then the root (where it adds the key value of the root to pref_sum variable and updates the key to pref_sum) , and then the right subtree.

4. **enqueue:** void function which takes a Queue and tree node pointer as input and inserts the tree node at the rear.
5. **dequeue:** removes the tree node pointer from the front of the queue and returns a pointer to that node.
6. **enqueue2:** void function takes a Range_queue and qinfo pointer as input and inserts qinfo node pointer at the rear.
7. **dequeue2:** removes the first element from Range_queue.
8. **print_tree:** takes a tree as input, prints it in level order and returns the sum of all values in the tree. A variable sum is initialized to zero for storing total sum. A queue of pointers to tree nodes is malloced and the first node is enqueued. Then it enters a while loop until the queue becomes empty (all nodes are printed). The key of the node is printed as it is dequeued in every iteration, and added to sum. Then it checks if the left and right children are not NULL and enqueues them onto the queue. Thus, as each node is printed

and dequeued, its children get enqueued. This ensures that we go level by level while traversing through the tree.

9. **main:** scans array size and array in level order fashion, creates Tree from list, modifies and prints the tree and sm by calling respective functions.

**Time Complexities:**

ListToBST: O(n), since in the worst case scenario, a node will go through the loop atmost twice if it doesn't satisfy the range, or if the left child is already assigned, and the for loop runs from 1 to n. The checks in if-else statements as well as the enqueue-dequeue operations inside the loop take O(1) time.

ModifyBST: O(n), since each node is visited only once in the entire traversal and the tree contains n nodes.

print_tree: O(n), since each node is visited only once in the entire traversal while printing in level order.