# Explanations of Question 4

My approach to solve the question comes from the property of the binary search trees' inorder printing i.e., if we print the values of the BST in inorder then the values will be in sorted order.

So, if we traverse the tree in inorder and instead of printing we just modify the values of node in the tree.

For modifying the value of the tree, I used one global variable named PREFIX_SUM which stores the value of prefix sum of inorder traversal.

For max sales, I used another global variable named MAX_SALE which stores the sum of the modified value of nodes of the tree.

For constructing the tree from the level order, I used the range of the value that children of a particular node can take.

## Implementation of ListToBST function

```
Tree ListToBST(const int *arr, int len)
{
    Tree T = NULL;
    Queue Q = initiallizeQueue();
    T = makeNode(arr[0], -10000001, 10000001);
    Push(Q, T);
    Tree temp;
    for (int i = 1; i < len; i++)
    {
        temp = Pop(Q);
        if (temp->max > arr[i] && temp->min < arr[i]){
            if (arr[i] < temp->key){
                if (temp->leftChild != NULL){
                    i--;
                }else{
                    temp->leftChild = makeNode(arr[i],temp->min,temp->key);
                    Inject(Q,temp,0);
                    Push(Q,temp->leftChild);
                }
            }else{
                temp->rightChild = makeNode(arr[i],temp->key,temp->max);
                Push(Q, temp->rightChild);
            }
        }else{
            i--;
        }
    }
    return T;
}
```

In implementation of the ListToBST function, I used the data structure circular deque which I implemented in question 2.

I used this data structure because it can work as a stack as well as queue. The functions of the deque are as follows.

Push -: it pushes the element at rear of the queue.

Pop -: it pops the element at the front of the queue.

Inject -: it pushes the element at the front of the queue.

***The time complexities of the above functions (Push, Pop and Inject) is O (1) or constant time.***

The element of the queue is of TreeNode pointer.

The structure of the TreeNode struct is as follows.

```
struct TreeNode
{
    long long int key;
    int min;
    int max;
    Tree leftChild;
    Tree rightChild;
};
```

Here key is the value of the node, min and max are the values such that the childrens of a node can be in the range (min, max) excluding min and max.

## makeNode function

```
Tree makeNode(long long int e, int min, int max)
{
    Tree T = (Tree)malloc(sizeof(struct TreeNode));
    T->key = e;
    T->min = min;
    T->max = max;
    T->leftChild = NULL;
    T->rightChild = NULL;
    return T;
}
```

***Time complexity of the makenode function is O (1) or constant time.***

It takes the arguments e, min, max and creates a TreeNode and assigns the respective values and shown in above picture.

Now in ListToBST function first it creates a TreeNode T with first element of the array and creates a queue Q & pushes Tree T to the queue Q. The min and max value of the T is given as the 1 less than the minimum value and 1 greater than the maximum value of the node that is given in the question, let call them MIN and MAX i.e., MIN = -10000001, MAX = 10000001. This because the value that the childrens of the root node can take is in between MIN and MAX.

Now using a for loop, it iterates over every other element in the array. Inside the for loop, first it pops the element in queue Q and stores that in a variable 'temp',

Now it checks the current value of the array if it can be the child of the node (by checking the value is in range by min and max). If arr[i] can be the leftchild (i.e., arr[i] is less than key value of temp) and leftchild of temp is NULL (if leftchild is not NULL then we just decrement i since the node temp already has it's leftchild and the arr[i] is less than the key value of node so the arr[i] can not be the right child of the temp) then is create the node having key value = arr[i], min value = min value of temp, max value = key value of temp and assign this node to the leftchild of temp. This is because now the childrens of this node can have values from min value of temp to key value of temp so that the property of BST be satisfied. Now we Push the leftchild of temp to the queue Q and Inject the temp to the queue Q so that it checks if in next iteration it finds rightchild or not. If arr[i] can be the rightchild of the temp then it creates the node having key value = arr[i], min value = key value of temp, max value = max value of temp and assigns this node to right child of the temp. The reason

is same as before, the property of the BST be satisfied. Now we Push the rightchild of the temp to queue Q. Here no need push the temp because no matter it has its leftchild or not, since the values is in level order it can not get a leftchild in array after rightchild.

If arr[i] can not be the left or right child of the temp, then we just decrement i so that in the next iteration it again checks for the same value as this value is not inserted in the tree.

We repeat this for every other element in the array arr and when loop ends, we just return T which is pointing to the root of the Tree.

Since in the function the for loop is iterating through the length of the input array. Due to the i-- thing the loop will not exactly run length of input array times but still is it will be linear time as O(n+k) is basically O(n).

***Hence the time complexity of the ListToBST function is O(n).***

**Implementation of modify BST.**

```
void ModifyBST(Tree T)
{
    Queue Q = initiallizeQueue();
    Inject(Q, T, 1);
    while (Q->numOfElements != 0)
    {
        int flag = Q->front->flag;
        Tree T = Pop(Q);
        if (flag == 0)
        {
            long long int temp = T->key;
            T->key += PREFIX_SUM;
            MAX_SALE += T->key;
            PREFIX_SUM += temp;
        }
        else
        {
            if (T->rightChild != NULL)
                Inject(Q, T->rightChild, 1);

            Inject(Q, T, 0);

            if (T->leftChild != NULL)
                Inject(Q, T->leftChild, 1);
        }
    }
}
```

It takes a Tree as an argument and modify the values of nodes of the Tree as the following logic.

The inorder traversal of the question is Iterative. For this I used data structure deque and Node struct of the data structure deque have a flag variable. The use of that flag variable is to insure any TreeNode is pushed only twice in the Q and to get the info of, whether it should be printed or pushed again.

To use the deque as a stack the functions used are as follows.

Inject -: it inserts the element at the front of the queue same as push operation of the stack.

Pop -: it removes the front element of the queue same as stack.

In the function, first a queue Q is created, and the root node of the tree is injected in the queue Q with flag 1. Now a while loop is running with the condition no. of elements in Q is not equal to 0. In every iteration, it stores the flag of the top (or front) in a flag variable and then pops the front element and stores that in a variable T. If the flag is 0 then it means that it was once gone in the flag 1 and it's childrens (if it has) are already pushed in the Q with flag 1. Hence, we store the key of the T in a temp variable and now we modify the value of the key of T as (PREFIX_SUM + temp). Now MAX_SALE variable is modified to MAX_SALE + new value of the key of treeNode T and PREFIX_SUM is updated to PREFIX_SUM + temp (previous value of the key of treeNode T).

If flag is 1 then it'd childrens are not yet pushed to Q so it injects the rightchild (if exists) with flag 1 and then treeNode T with flag 0 and at last leftchild (if exists) with flag 1 this is because now the leftchild will be on the front of the Q and will be popped first and followed by T and then rightchild and this is what inorder traversal is.

This process will continue until the Q in not empty (i.e., no of elements in Q = 0).

*In this way every node will be pushed & popped twice in the Q and hence the time complexity of this function is O(4n) that basically O(n).* ***Hence the time complexity of this function is O(n)***.

**Implementation of Level Order Printing.**

```
void printLevelOrder(Tree T)
{
    Queue Q = initiallizeQueue();
    Push(Q, T);
    while (Q->numOfElements != 0)
    {
        Tree temp = Pop(Q);
        if (temp->leftChild != NULL && temp->rightChild != NULL)
        {
            Push(Q, temp->leftChild);
            Push(Q, temp->rightChild);
        }
        else if (temp->leftChild != NULL)
        {
            Push(Q, temp->leftChild);
        }
        else if (temp->rightChild != NULL)
        {
            Push(Q, temp->rightChild);
        }
        printf("%lld ", temp->key);
    }
}
```

This function takes root node of the tree and prints the values of the nodes of this tree in level order. The idea is use a queue and enqueue the root node in starting and then dequeue the front element from the queue and print the key value of that and enqueue the leftchild first and then right child so that the leftchild will dequeued first.

In the function first a queue Q is created, and the root node of the Tree is pushed. Now a while loop runs with the condition no. of elements in the Q is not equal to 0. Inside the loop it pops the

front element in Q and if it has left or right child then push that at the rear in the Q in order of leftchild first and then right child.

*Since every element is pushed and popped once hence the time complexity of the function is O(2n) that is basically O(n).*

***Hence the time complexity of the function is O(n).***