EXPLANATION FOR Q4. KING AND CRISIS

```c
#include <stdio.h>
#include <stdlib.h>
int MAX = 10000005, MIN = -10000005;
typedef struct TreeNode *Tree;
typedef struct TreeNode *PtrToNode;
struct TreeNode
{
    long long int Key;
    PtrToNode Left;
    PtrToNode Right;
    int Min;
    int Max;
};
```

Global variables MIN and MAX are set to -(1e7+5) and 1e7+5 respectively since the range of values inputted lie between -1e7 to +1e7, so all values in the inputted tree will lie between MAX and MIN. We will see the use of MIN and MAX in the ListToBST() function.

struct TreeNode contains information about a node of the BST, where Key is the value of the Node, Left and Right store pointers to the left and right child of the node respectively. Min and Max store the minimum and maximum possible value for the key respectively, derived from the Min and Max of its Parent node.

Tree represents the pointer to the Root of the Tree, while PtrToNode is used as the pointer to any one node in the Tree.

```c
typedef struct QueInfo
{
    int elementCount;
    int elementMax;
    int front;
    int rear;
    PtrToNode QArr[1000000];
} QueInfo;
typedef QueInfo *Queue;

long long int PrefixSum, MaxSale;
```

struct QueInfo contains the information about a queue:

elementCount:	Current number of elements in the queue
elementMax:	Maximum number of elements the queue can hold
front:	Index pointing to the element to be next dequed.
Rear:	Index pointing to the element last enqued.

Qarr: Array of elements in the Queue

We have set the size of QArr to 1e6 elements since that is the maximum number of tree nodes that can be enqued.

PrefixSum stores the sum of all the elements encountered till now in the ModifyBST function, whereas MaxSale stores the sum of all values that PrefixSum has stored during its use.

```
Queue Initialize()
{
    Queue Q = (Queue)malloc(sizeof(QueInfo));
    Q->elementMax = 1000000;
    Q->front = 0;
    Q->rear = -1;
    Q->elementCount = 0;
    return Q;
}
int isFull(Queue head)
{
    if (head->elementCount >= head->elementMax)
        return 1;
    return 0;
}
int isEmpty(Queue head)
{
    if (head->elementCount < 1)
        return 1;
    return 0;
}
void Enque(Queue head, PtrToNode val)
{
    if (!isFull(head))
    {
        head->rear = (head->rear + 1) % (head->elementMax);
        head->QArr[head->rear] = val;
        head->elementCount += 1;
    }
}
PtrToNode Deque(Queue head)
{
    if (isEmpty(head))
        return NULL;
    PtrToNode e = head->QArr[head->front];
    head->front = (head->front + 1) % (head->elementMax);
    head->elementCount -= 1;
    return e;
}
```

These are the functions for Enqueing and Dequeing elements from the Queue.

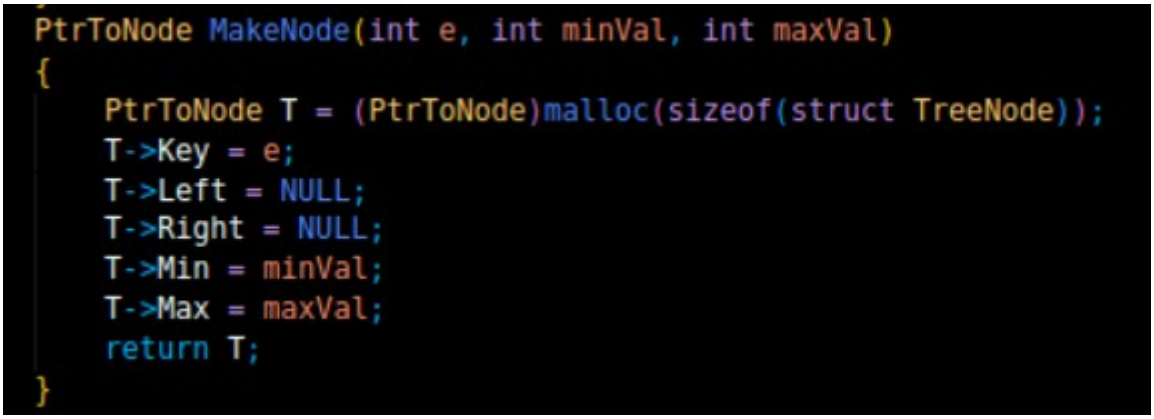
Queue Initialize() is used to create a new queue with a maximum size of 1e6. Time Complexity: O(n), n=1e6. When we malloc the size of the Queue, all 1e6 QArr elements are initialized to 0 (may be compiler dependent). This initialization causes n operations to happen.

int isEmpty(Queue head) is used to check if the queue is empty.
Time Complexity: O(1). head->elementCount can be accessed and compared to 1 in constant time.

int isFull(Queue head) is used to check if the queue is full.
Time Complexity: O(1). head->elementCount and head->elementMax can be accessed and compared to each other in constant time.

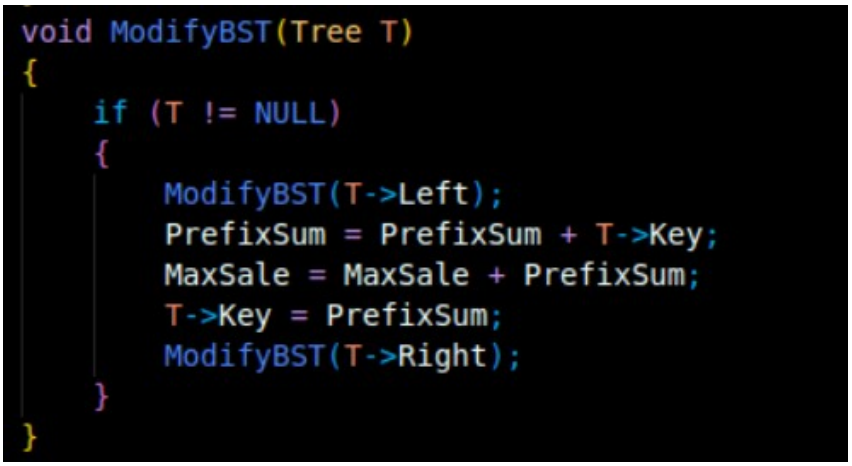
void Enque(Queue head, PtrToNode val) is used to add a new node val to the rear of the queue and hence update rear and elementCount.
Time Complexity: O(1). The queue rear can be accessed in constant time as array elements can be accessed in O(1). Updating rear and elementCount also takes contant time.

PtrToNode Deque(Queue head) is used to remove a node from the front of the queue and hence update front and elementCount.
Time Complexity: O(1). The queue front can be accessed in constant time as array elements can be accessed in O(1). Updating rear and elementCount also takes contant time.



```
PtrToNode MakeNode(int e, int minVal, int maxVal)
{
    PtrToNode T = (PtrToNode)malloc(sizeof(struct TreeNode));
    T->Key = e;
    T->Left = NULL;
    T->Right = NULL;
    T->Min = minVal;
    T->Max = maxVal;
    return T;
}
```

PtrToNode MakeNode(int e, int minVal, int maxVal) is used to create a new node. T is a pointer to this node with the Key as e, set Left and Right to NULL, as in a top down method of node insertion, T is presently a leaf node. Min and Max of T are set to minVal and maxVal given in the function call, and T is returned.
Time Complexity: O(1). We allocate space of the size of struct TreeNode and initialize Key, Left, Right, Min and Max, each of wich happens in constant time.



```
void ModifyBST(Tree T)
{
    if (T != NULL)
    {
        ModifyBST(T->Left);
        PrefixSum = PrefixSum + T->Key;
        MaxSale = MaxSale + PrefixSum;
        T->Key = PrefixSum;
        ModifyBST(T->Right);
    }
}
```

void ModifyBST(Tree T) takes a BST as input. It is a recursive function that calls itself for its left child. After that it calculates the PrefixSum as the sum of the PrefixSum of the previously encountered node and the Key of the present node. Then it calculates MaxSale as the sum of MaxSale till the last node and the PrefixSum calculated for this node. Its stores the PrefixSum calculated as the Key of the Node. Then ModifyBST() calls itself for the right child. The recursive calls happen until after the leaf node is reached, after which T->Left and T->Right are NULL, thus terminating it.

Due to the recursive nature of it, ModifyBST(T->Left) is called until the leftmost leaf node is reached (after which T->NULL stops the recursive call). So the function starts from the leftmost node of the Tree, calculates its PrefixSum and MaxSale (both of which are initialised as 0 in the main() function, so PrefixSum and MaxSale are the Key of the leftmost node itself).

The function first recursively traverses the left subtree, then updates PrefixSum by adding the key value of the current node, then updating the MaximumSale value by adding the present PrefixSum, and finally updating the key value of the current node with the PrefixSum so that it becomes the sum of all the elements lesser than or equal to it. The function then recursively traverses the right subtree. This is done because, in a BST, all elements smaller than an element appear to the left of it. An inorder traversal thus gives us the tree elements sorted in ascending order, thus traversing it in inorder, left subtree, root and then right subtree, allows us to access the smallest (leftmost) node first and a node is accessed only after all nodes smaller than it (left to it) are accessed, thus allowing us to calculate the values of all elements equal or lesser than the key by maintaing a PrefixSum variable.

Time Complexity: O(n), n=number of elements in the tree. Since we recursively access each element only one time (from left to right) and perform only addition and assignment operations after it, which take O(1) for each element, and there are n elements to access, the complexity is O(n).

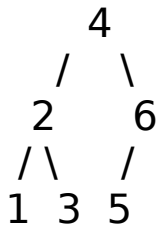
```
void Levelorder(Tree T)
{
    if (T == NULL)
        return;
    Queue Q = Initialize();
    Enque(Q, T);
    while (!isEmpty(Q))
    {
        PtrToNode Curr = Deque(Q);
        if (Curr->Left != NULL)
            Enque(Q, Curr->Left);
        if (Curr->Right != NULL)
            Enque(Q, Curr->Right);
        printf("%lld ", Curr->Key);
    }
}
```

void Levelorder(Tree T) is an iterative function that prints the Tree T in a top-down way, i.e., first the root is printed, then the children of the roots are printed with the left node being printed before the right one, then the children of the children of the root are printed from leftmost to rightmost and so on until all the nodes are printed.

If T is NULL nothing is to be printed. Then we initialize a Queue of TreeNode, where we enque the root of the tree. Then, until the queue is empty, we first deque the TreeNode at front, enque its left and right children (if they exist) and then print its Key.

Time Complexity: O(n), n=number of elements in the tree. Every element is enqued and dequed only once, both of which take O(1) time, and there are n elements. The loop runs until the Queue is empty, which happens by n deques, i.e., O(n). Each element is also printed once, which takes O(1) time too. Thus, the time complexity is O(n).

The working of this algorithm is explained using the following tree:



Assume for this explanation, that 1 to 6 refer to the PtrToNode of the tree elements, not the Key itself.

Enque: 4

Queue: 4

Q is not empty.
Deque() gives us 4.
4->Left is 2 and not NULL.
Enque(2)
4->Right is 6 and not NULL.
Enque(6)
Print(4->Key)

Queue: 2 6

Q is not empty.
Deque() gives us 2.
2->Left is 1 and not NULL.
Enque(1)
2->Right is 3 and not NULL.
Enque(3)
Print(2-Key)

Queue: 6 1 3

Q is not empty.
Deque() gives us 6.
6->Left is 5 and not NULL.
Enque(5)
6->Right is NULL.
No Enque
Print(6->Key)

Queue: 1 3 5

Q is not empty.
Deque() gives us 1.
1->Left is NULL.
No Enque
1->Right is NULL.
No Enque
Print(1->Key)

Queue: 3 5

Q is not empty.

Deque() gives us 3.
3->Left is NULL.
No Enque
3->Right is NULL.
No Enque
Print(3->Key)

Queue: 5

Q is not empty.
Deque() gives us 5.
5->Left is NULL.
No Enque
5->Right is NULL.
No Enque
Print(5->Key)

Queue:

Q is empty.
Terminate loop

Output console:

2 4 6 1 3 5

Level order of tree:

2 4 6 1 3 5

This function ensures the output is in LevelOrder as the children of a node are added to the queue after it is dequed, ensuring top-down printing, and the left node is added before the right node ensuring that at any level, it is dequed before the right node is, and thus gets printed before a node to the right of it.

```
PtrToNode ListToBST(int n, int *arr)
{
    Tree T = MakeNode(arr[0], MIN, MAX);
    Queue Q = Initialize();
    Enque(Q, T);
    for (int i = 1; i < n;)
    {
        PtrToNode Parent = Deque(Q);
        if (arr[i] > Parent->Min && arr[i] < Parent->Max)
        {
            if (arr[i] < Parent->Key && arr[i]>Parent->Min)
            {
                Parent->Left = MakeNode(arr[i], Parent->Min, Parent->Key);
                Enque(Q, Parent->Left);
                i++;
                if (i >= n)
                    break;
            }
            if (arr[i] > Parent->Key && arr[i]<Parent->Max)
            {
                Parent->Right = MakeNode(arr[i], Parent->Key, Parent->Max);
                Enque(Q, Parent->Right);
                i++;
                if (i >= n)
                    break;
            }
        }
    }
    return T;
}
```

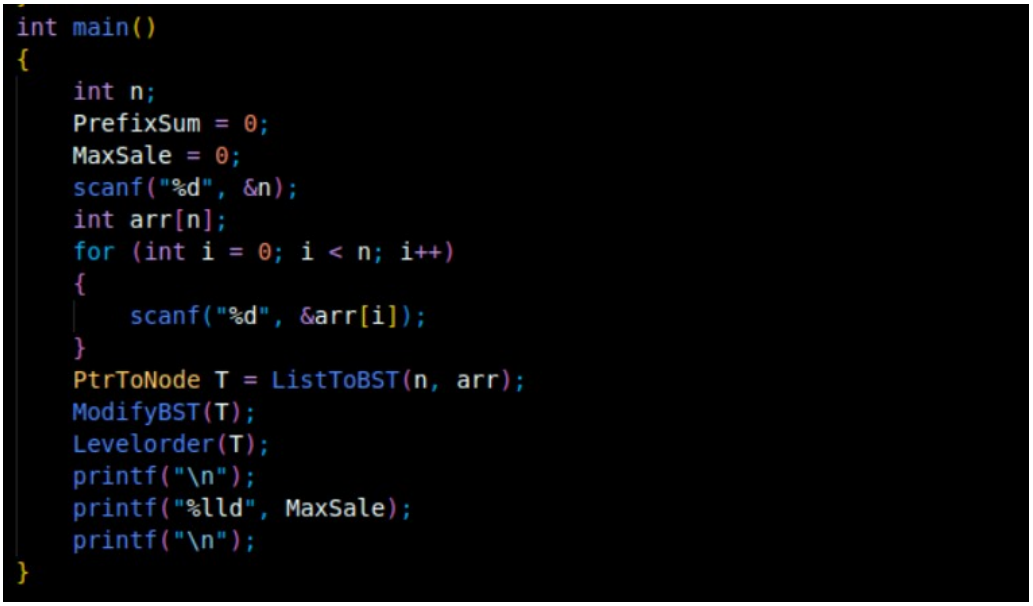
PtrToNode ListToBST(int n, int *arr) takes in an array and its length as its input. This array is the list of all elements in the tree inputted by the user in a Level Order form.

It is used to construct a BST iteratively from a given array in Level order. This is done by creating a new node with the first element of the array as the Key of the root of the tree. MIN and MAX are the minimum and maximum values that Key could take, which have been initialized according to the range of input values specified in the question.

It then initializes a new queue and enques the root node. It iterates over the remaining elements of the list, starting from the second element. For each element, it dequeues Parent (variable name), the node at the front from the queue, checks if the element currently being looped over can be inserted as a left child node of Parent, which is possible if it lies in the range [Parent->Min, Parent->Key). If possible, we set it as the left child of Parent by creating a new node with Max set to Parent->Key and Min of Parent retained as its Min, and increment i to now be the index of the next element. If it cannot be the left child, i is not incremented and we and check if the present element in the list can be the or right child node of Parent, which is possible if it lies in the range (Parent->Key, Parent->Max]. If possible, we set it as the right child of Parent by creating a new node with Min set to Parent->Key and Max of Parent retained as its Max, and increment i to now be the index of the next element, after which it loops over and over until all the elements in the list are inserted. It is possible

that Parent is a leaf node, in which case the current element in the array will not match either of the conditions required to be the Left or the Right child of it.
This logic follows from the level order traversal of a tree, where the left node appears before the right node and the parent of each node appears before its children in a top down form of traversal.

Time Complexity: O(n), n=number of elements in the tree. We are simply running a loop from the second to the last element in the array of tree elements. The operations inside the loop are dequeing and then enqueing based on if-else statements, which have an O(1) complexity. Thus the overall complexity is O(n).



```
int main()
{
    int n;
    PrefixSum = 0;
    MaxSale = 0;
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    PtrToNode T = ListToBST(n, arr);
    ModifyBST(T);
    Levelorder(T);
    printf("\n");
    printf("%lld", MaxSale);
    printf("\n");
}
```

int main() is used to first take the input of the number of elements in the BST from the user. Then we run a loop from i=0 to n-1 where we take input of the i^{th} element of the tree in level order traversal. We then store the tree created by by passing the inputted array and n to ListToBST() in T. We pass T which modifies the tree in-place according to the question. It also updates the value of MaxSale. Then we print this modified tree in level order along MaxSale, which is the sum of the keys of all the modified nodes calculated by adding up all the values stored by PrefixSum.

Time Complexity: O(n), n=number of elements in the tree. We loop over 0 to n and input all the n elements into an array, and each input into arr[i] takes O(1) time as array elements can be accessed by their index in contant time. After that we call ListToBST, ModifyBST() and Levelorder(), all of which are functions happening in O(n). Rest of the initialization and printing MaxSale takes O(1). So overall the main() function has a complexity of O(n).

Thus the program, overall takes O(n) time to run, since all functions are of a maximum of O(n): input, list to tree conversion, modifying the BST and printing it in Level order, and they do not occur in a nested fashion but serially.