DOCUMENTATION FOR ASSIGNMENT-I

Table of Contents

Basic points to consider

I have created a class BST, which contains the following attributes:

- **value:** it is an int variable which will store the value of the BST node.
- **left:** it is a pointer which points to the left node. It is of type BST*
- **right:** it is a pointer which points to the right node. It is of type BST*
- **rstval:** it is an int variable which stores the total number of nodes in the node's right subtree
- **lthread:** It is a boolean variable which is true if the node has a left thread.
- **rthread:** It is a boolean variable which is true if the node has a right thread.

The functions of BST class are all public, and discussed in the document.

Apart from this, I have also created two structs:

- **LLNode:** It is a struct which stores an integer value, and a pointer to another LLNode. It is used to print/store reverseInorder() and to print/store allElementsBetween(k1,k2).
- **LLNode2:** It is a struct which stores a BST pointer, and a pointer to another LLNode2. It is used to store parents of node to be deleted from the parents' right subtree (Useful in finding kth maximum element).

In the main() of the program, I have provided a textual user interface which can be used to call the following functions and get relevant outputs.

1. insert(x):

Logic:

Insertion is done by creating a new node with value x, and left and right child as NULL.

Our first task is to find the location for insertion. This will happen in a typical BST fashion, i.e we will compare the value of x to our current node's value.

If during traversal, the element which is to be inserted is already present (i.e. x==value of current node), it will give a message saying "Element already present".

If x<value of current node, we will recursively enter in the left subtree of the current node, and call insert function with node as node->right. In this case if a node has no left child, we traverse to its left (node = NULL), and node is inserted there. Then if the parent has a left thread, it will be deleted and the parent's left will point to the new node.
Vice versa will occur if x>value of current node.

The proper location is obtained when node = NULL. There are the following 3 cases that can occur afterwards, and the adjustment of threads are to be made accordingly.

1. **Tree is empty**

Here, we just simply assign the root to the created node.

2. **Node inserted as left child**

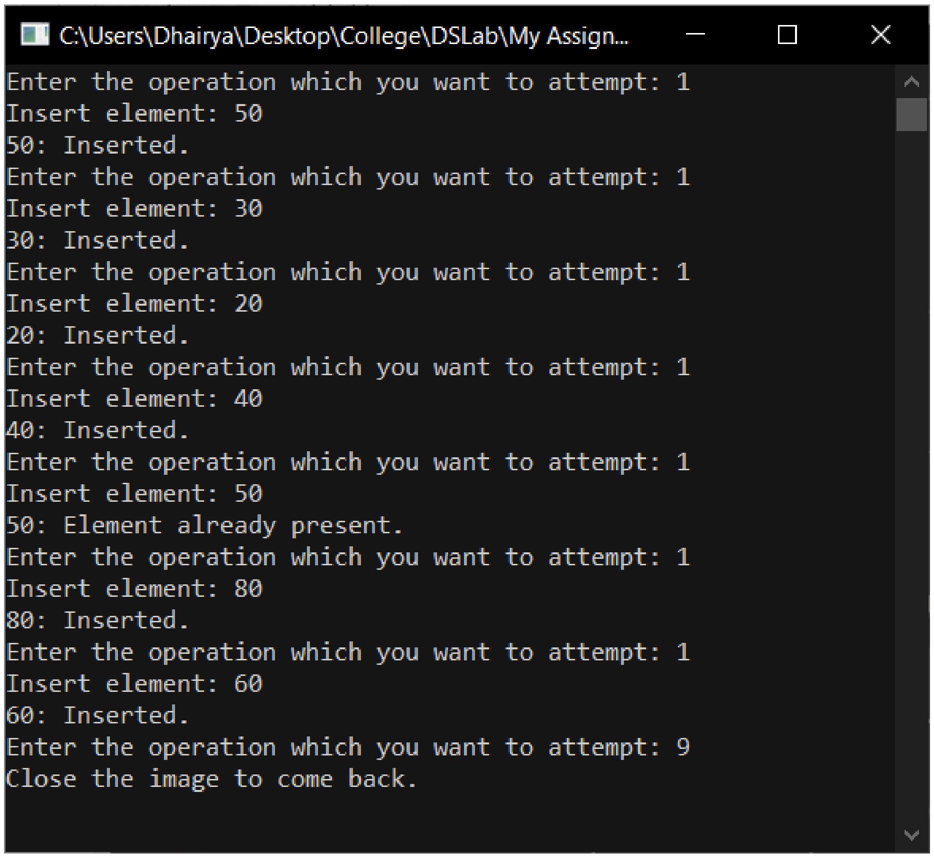
Here, we see if the parent has a lthread or not. If they do, we assign it to the new node. Along with this, we make the rthread of the new node point to their parent.

3. **Node inserted as right child**

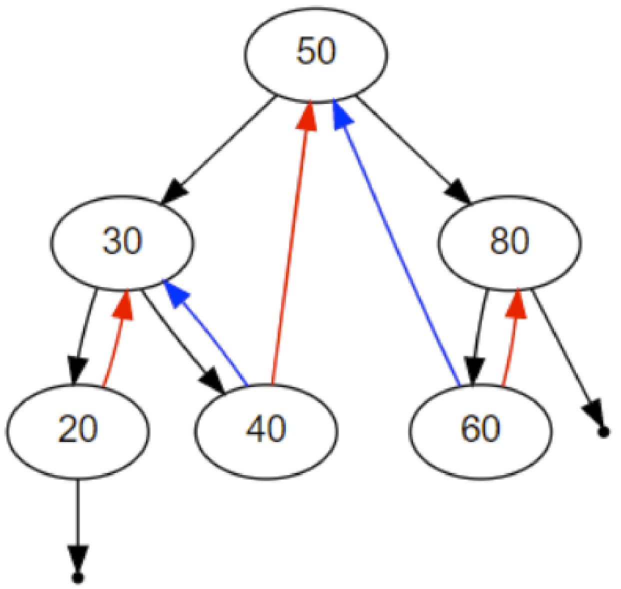
Here, we see if the parent has a rthread or not. If they do, we assign it to the new node. Along with this, we make the lthread of the new node point to their parent.

At the end of the function, it returns the value of the node inserted. If node is already present, it returns that node.

Test Cases:



```
Enter the operation which you want to attempt: 1
Insert element: 50
50: Inserted.
Enter the operation which you want to attempt: 1
Insert element: 30
30: Inserted.
Enter the operation which you want to attempt: 1
Insert element: 20
20: Inserted.
Enter the operation which you want to attempt: 1
Insert element: 40
40: Inserted.
Enter the operation which you want to attempt: 1
Insert element: 50
50: Element already present.
Enter the operation which you want to attempt: 1
Insert element: 80
80: Inserted.
Enter the operation which you want to attempt: 1
Insert element: 60
60: Inserted.
Enter the operation which you want to attempt: 9
Close the image to come back.
```



RED: rthread, BLUE: lthread

search(x)

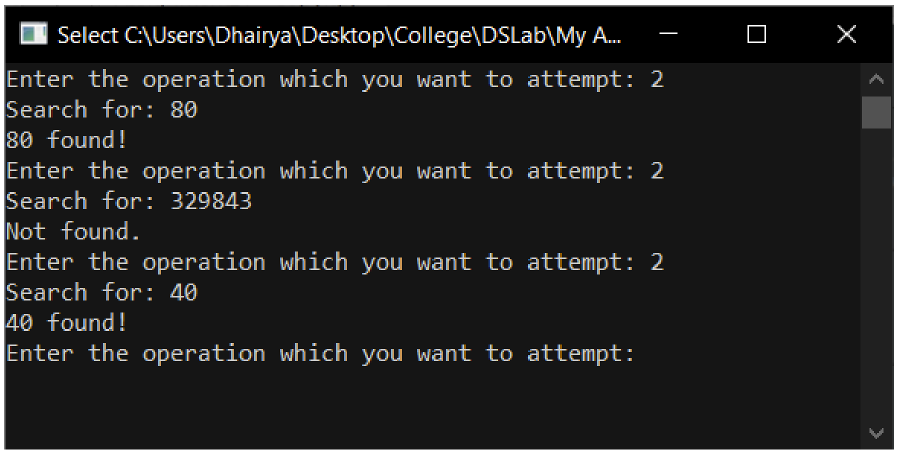
Logic:

The procedure of traversal is the same as the above function. We can basically consider it as being the case of the above function where the element to be inserted already existed.

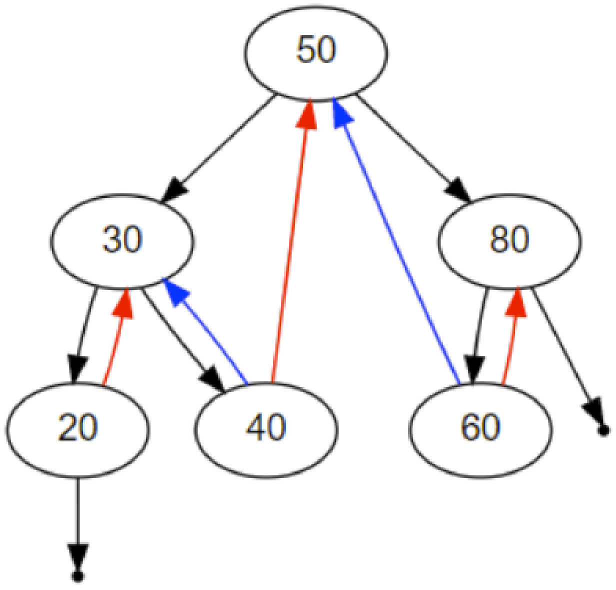
I.e. we will compare the values of x and current node. If x < current->value, we will recursively call the left subtree. If x > current->value, we will recursively call the right subtree. If x == current->value, we will display “Element found!” message and return that node.

If the tree is empty (root == NULL) or the element is not found during traversal (a leaf node is not equal to x), we will return NULL and display “Element not found”.

Test Case:



```
Enter the operation which you want to attempt: 2
Search for: 80
80 found!
Enter the operation which you want to attempt: 2
Search for: 329843
Not found.
Enter the operation which you want to attempt: 2
Search for: 40
40 found!
Enter the operation which you want to attempt:
```



RED: rthread, BLUE: lthread

delete(x)

Logic:

To delete a node, we need to first locate it. We could do that with search(), but we also need to do fair amount of adjustments to its threads, so the parent node is necessary. Hence, we will use the same traversal technique as used in search, but will also keep track of the parent.

When we find the node, we will see if it has 2 children, 1 child or 0 children (leaf).

1. **Node has 0 children(leaf)**

If it is the only node in the tree, we will make root 0.
Otherwise if it was the left child of its parent, we will assign lthread of node to the parent.
Else, we will assign the rthread of node to the parent.

2. **Node has 1 child**

We will first check if the node has a left or right child.
If only the left child exists, we will assign the temp node variable to it. Otherwise we will assign the right child the temp node variable.

Now we will check if the parent is null. If it is, the temp node becomes the root.
Else if the node is the right child of the parent, temp becomes the right child of the parent.
Likewise if node is the left child.

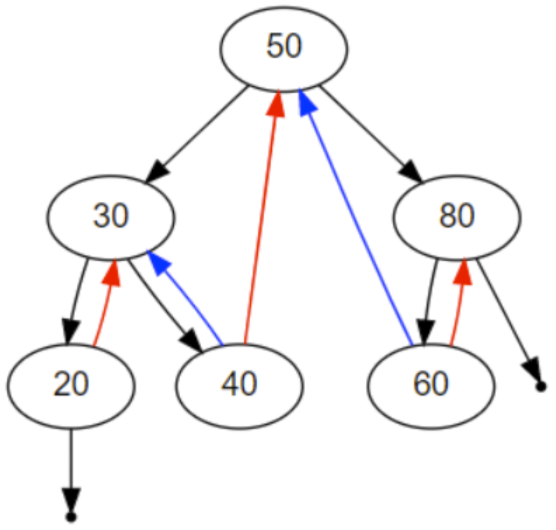
Now if the node has left child, we will assign the right thread to its child. Else we'll assign the left thread to its child.

3. **Node has 2 children**

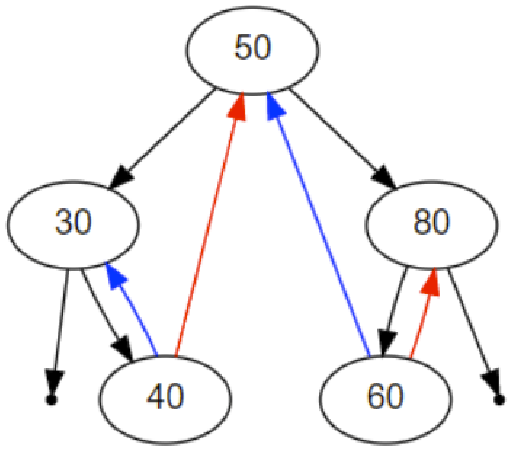
For this, we will find the inorder successor of the node, and its parent. We will then copy the value of inorder successor to the node. Now, we will delete the inorder successor, by seeing how many children it has and acting accordingly.

This function always returns the root of the whole tree.

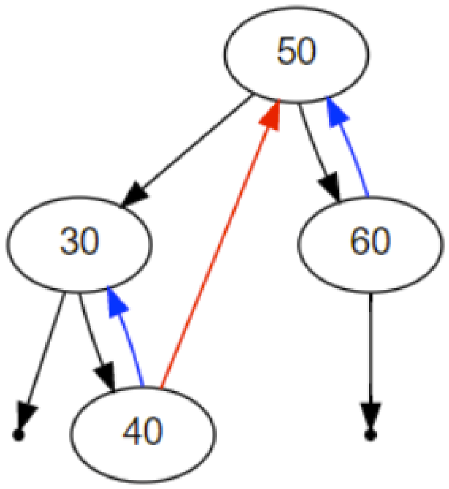
Test Case:



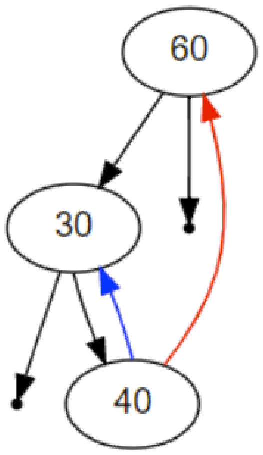
RED: rthread, BLUE: lthread



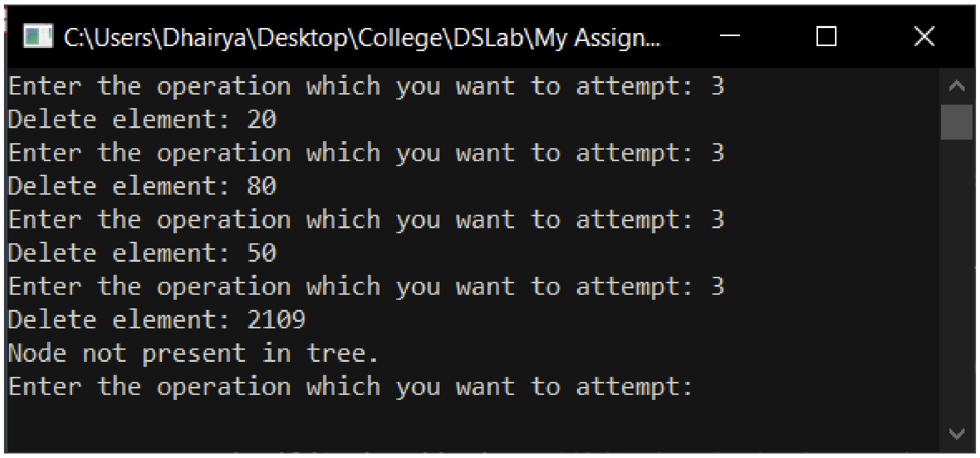
RED: rthread, BLUE: lthread



RED: rthread, BLUE: lthread



RED: rthread, BLUE: lthread



```
Enter the operation which you want to attempt: 3
Delete element: 20
Enter the operation which you want to attempt: 3
Delete element: 80
Enter the operation which you want to attempt: 3
Delete element: 50
Enter the operation which you want to attempt: 3
Delete element: 2109
Node not present in tree.
Enter the operation which you want to attempt:
```

reverseInorder()

Logic:

This function requires an output into a linked list. So first thing is we defined a linked list structure (LLNode), and the return type of this function will be that structure. Along with the structure, we defined two functions: append() which adds a new node with a given value after the last element, and printout() which prints the linked list. In the function, we will give a pointer to this structure.

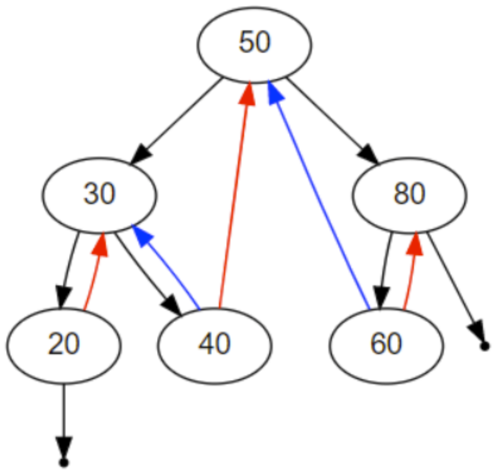
To do a reverse inorder traversal, we will first go to the rightmost element, by traversing rightwards (node = node->right) till the leaf node occurs. I have created a function rightmost() for this, which returns the rightmost element of a tree with given node as root. We will append the value of that node to the linked list.

Then we will check if a node has a lthread or not. If it does, we will go to the node pointed by lthread, and append its value to the linked list.

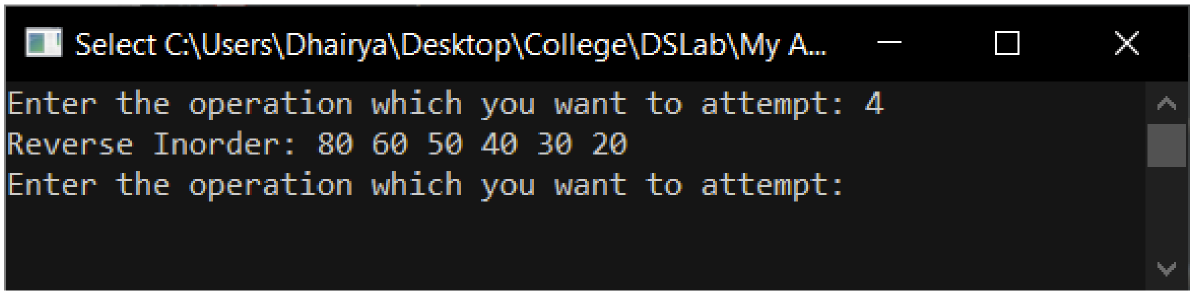
If the node does not have a right thread, we will find its inorder predecessor (rightmost node of the left subtree). We will pass current->left to the rightmost() function to obtain that. We will also append that node's value to the linked list

When the current node becomes NULL (as the leftmost node will not have any lthread), we will terminate the loop. Then, we will printout() the linked list obtained, and return it.

Test Cases:

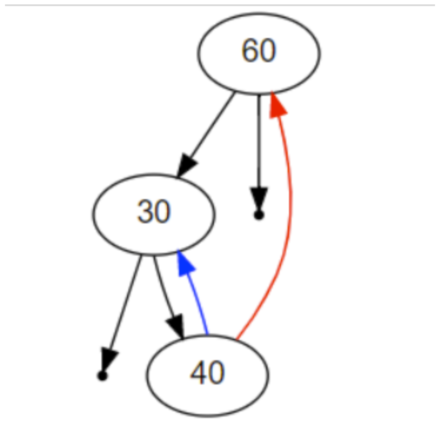


RED: rthread, BLUE: lthread

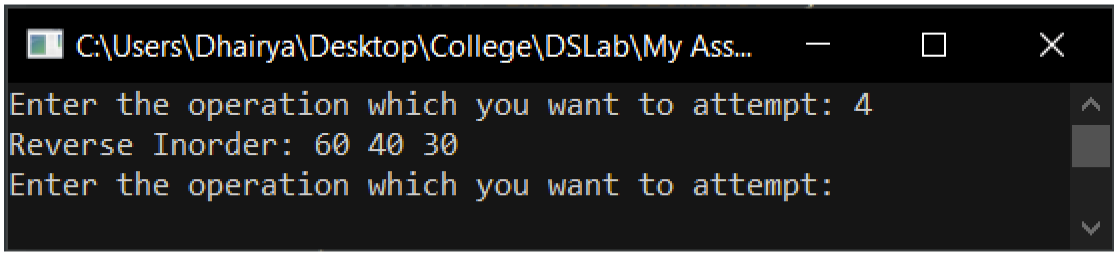


```
Select C:\Users\Dhairya\Desktop\College\DSLab\My A...
Enter the operation which you want to attempt: 4
Reverse Inorder: 80 60 50 40 30 20
Enter the operation which you want to attempt:
```

Case2:



RED: rthread, BLUE: lthread



```
C:\Users\Dhairya\Desktop\College\DSLab\My Ass...
Enter the operation which you want to attempt: 4
Reverse Inorder: 60 40 30
Enter the operation which you want to attempt:
```

successor(node)

Logic:

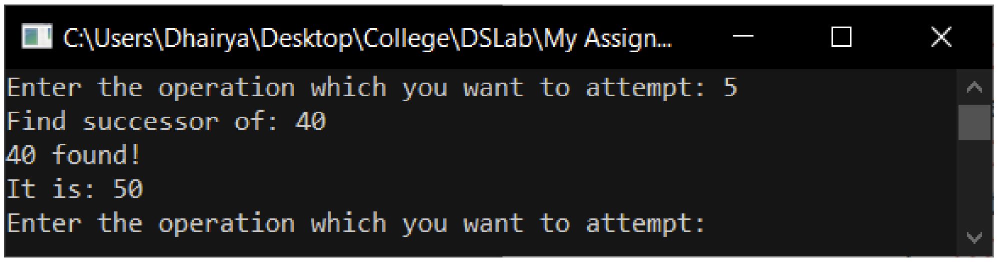
Here, we have to return the inorder successor of the node which is passed in the argument. Since I am taking input as number x, I have to first search(x), and the node it returns is then passed into successor().

So, we will check if the node has rthread==true. If it does, we will go to that node (node = node>right) and return that node.

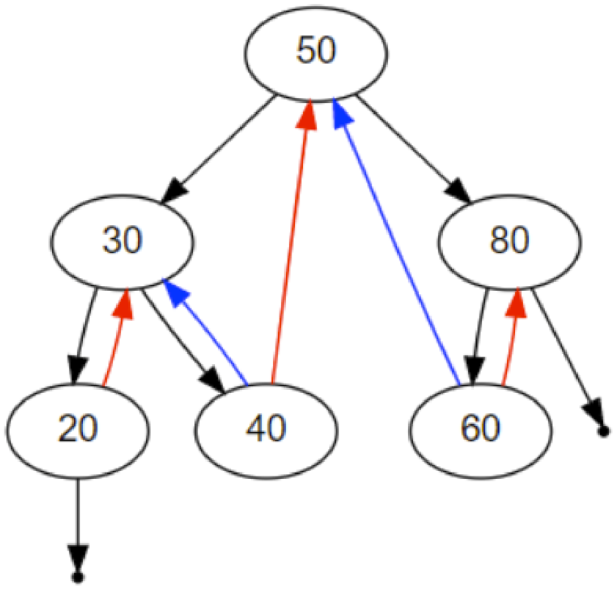
If for the given node rthread==false, we will go to the node which is the leftmost node of its right subtree. For that, we will return the value of the node returned by leftmost(node->right).

We saw rightmost() in the above function, leftmost works on the exact same principle, but instead of node->right, we will traverse to node->left till leaf occurs.

Test Cases:



```
Enter the operation which you want to attempt: 5
Find successor of: 40
40 found!
It is: 50
Enter the operation which you want to attempt:
```



RED: rthread, BLUE: lthread

allElementsBetween(k1,k2)

Logic:

This function requires an output into a linked list. So the return type of this function will be the linked list structure (LLNode) which we saw in reverseInorder() function application.

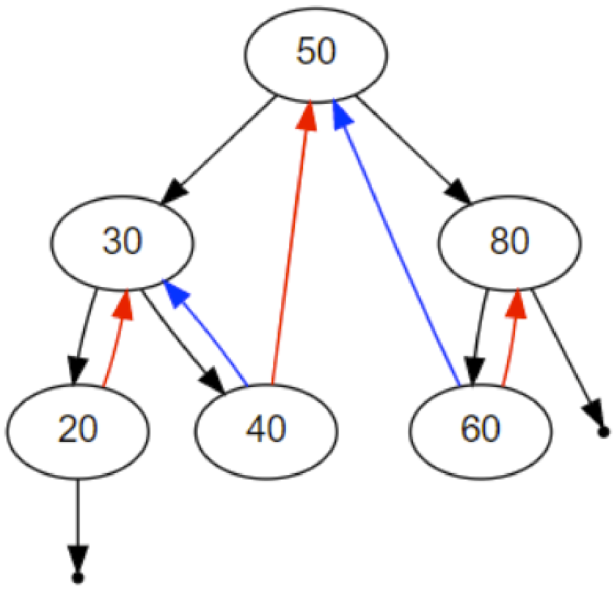
Then I have saved the return value of search(k1) and search(k2) into two nodes node1 and node2 respectively.

First we check if either of node1 and node2 is NULL, or k1>k2. If this happens, it means the input is invalid, and we give this message and return NULL.

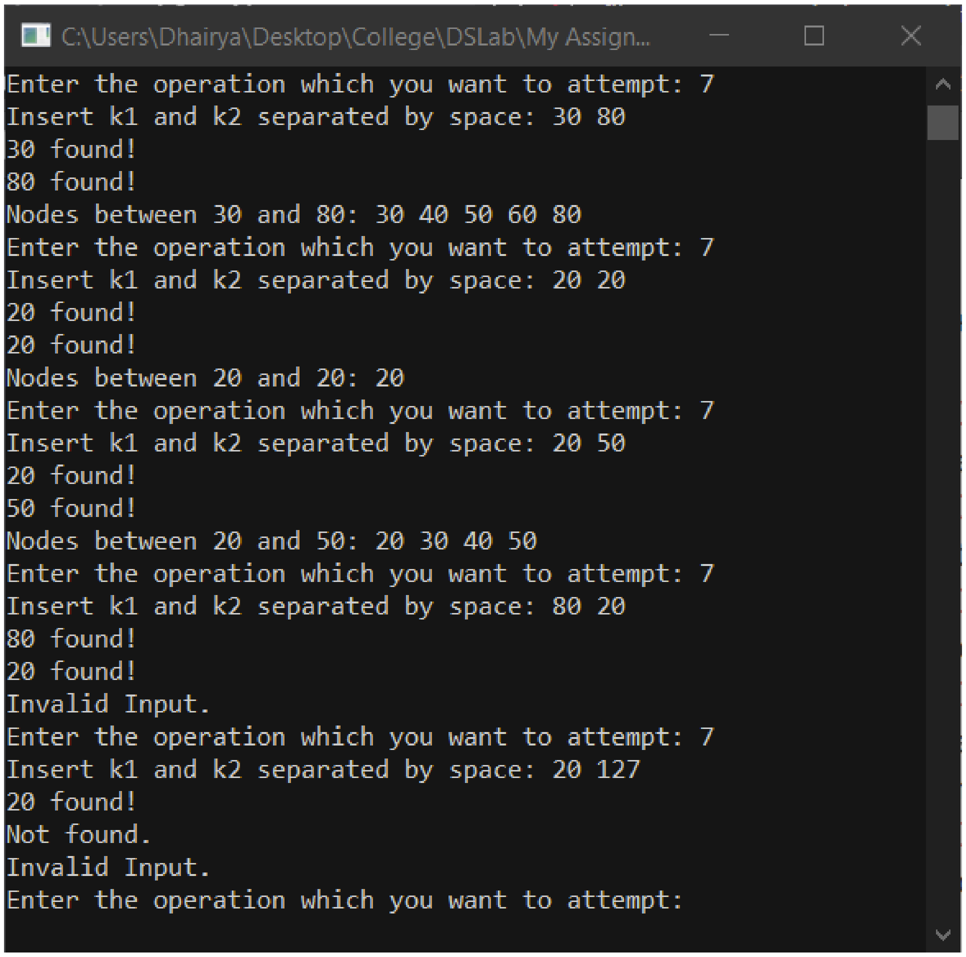
Now, we will check if node1 == node2. If it is not, we will append that node's value into the linked list.
We then find the inorder successor of that node. We could have used the function successor(), but the append function would not have been implemented in it. So we use the same logic, but not call that function. Then, node1 = inorder successor of node2.

We will repeat the above steps till node1 == node2. Then, we will append value of node2 to the list, because due to while() condition it won't be appended automatically. Then we call printout() on the linked list, and return it.

Test Case:



RED: rthread, BLUE: lthread



```
Enter the operation which you want to attempt: 7
Insert k1 and k2 separated by space: 30 80
30 found!
80 found!
Nodes between 30 and 80: 30 40 50 60 80
Enter the operation which you want to attempt: 7
Insert k1 and k2 separated by space: 20 20
20 found!
20 found!
Nodes between 20 and 20: 20
Enter the operation which you want to attempt: 7
Insert k1 and k2 separated by space: 20 50
20 found!
50 found!
Nodes between 20 and 50: 20 30 40 50
Enter the operation which you want to attempt: 7
Insert k1 and k2 separated by space: 80 20
80 found!
20 found!
Invalid Input.
Enter the operation which you want to attempt: 7
Insert k1 and k2 separated by space: 20 127
20 found!
Not found.
Invalid Input.
Enter the operation which you want to attempt:
```

kthElement(k)

Logic:

For this function, we maintained a rank of each node. That is basically the count of the number of nodes present in the right subtree of a node. This is denoted by rstval in my code.

rstval was incremented for a node during the insert() function by me whenever an element was inserted into its right subtree.

During deletion, while traversing to find the node to delete, all parents of that node were stored in a linked list if that node was in the right subtree of their parents. This linkedlist was of type LLNode2, and a structure was defined for it. Whenever delete happened, all the rstvals of parents were decremented by 1 using function rstvalAdjustment().

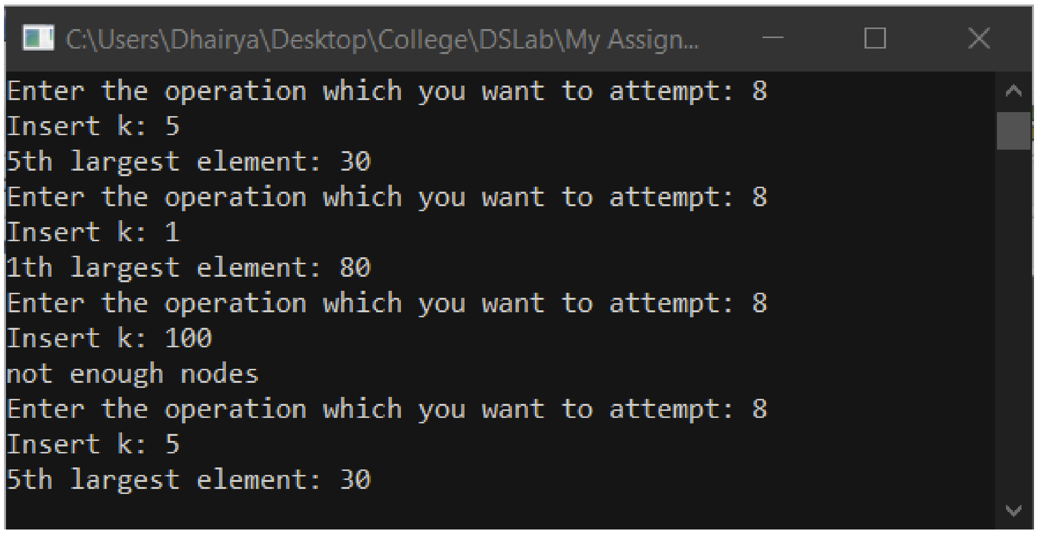
Coming to the actual function, We would traverse from the root. We had to calculate the rstval+1 of the node at each step, and compare it with k.

If it was greater than k, it meant that traversal had to be done in the left subtree. So node = node->left, and k = k-(rstval+1). This was done because we skipped all the nodes from the right subtree of the node, plus the node itself.
If it was less than k, we would simply go to the right subtree.

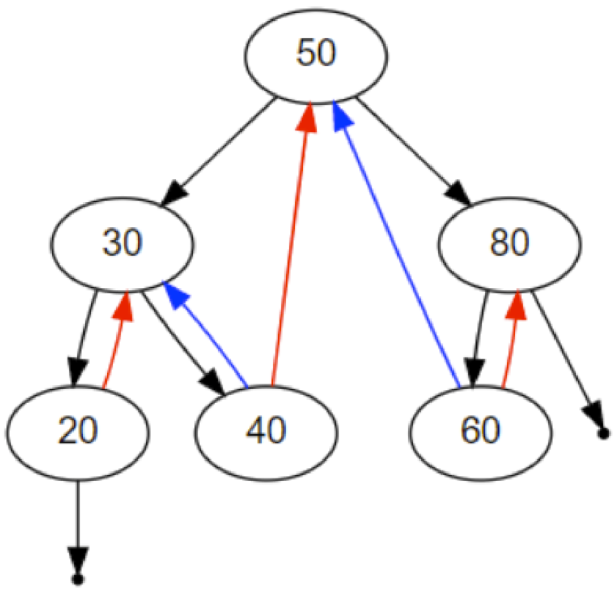
This loop ends when k == rstval+1 of a node. That node is the kth largest element of the tree. We will return that node.

The loop also ends when node equals NULL. That means there aren't enough nodes and we return NULL.

Test Case:



```
Enter the operation which you want to attempt: 8
Insert k: 5
5th largest element: 30
Enter the operation which you want to attempt: 8
Insert k: 1
1th largest element: 80
Enter the operation which you want to attempt: 8
Insert k: 100
not enough nodes
Enter the operation which you want to attempt: 8
Insert k: 5
5th largest element: 30
```



RED: rthread, BLUE: lthread

printTree()

Logic:

Here, we needed the use of graphviz software. So we had to basically make a graphviz compatible file by writing data out of the cpp file.

For this, we used 3 functions:

1. **printTree_null**
2. **printTree_main**
3. **printTree**

The first function is just used to write the command which makes the input argument node connect to a dot through an arrow.

The second function is basically creating all the edges.
It first sees if the left of the current node (starting from root) is NULL or not. If not, it checks if lthread is true or not. If lthread is not true, it gives command to draw edge from current node to current->left node, and then calls the function again with argument current->left. If lthread is true, it draws a blue arrow from current to current->left.
If node->left was NULL, it calls the printTree_null() function.

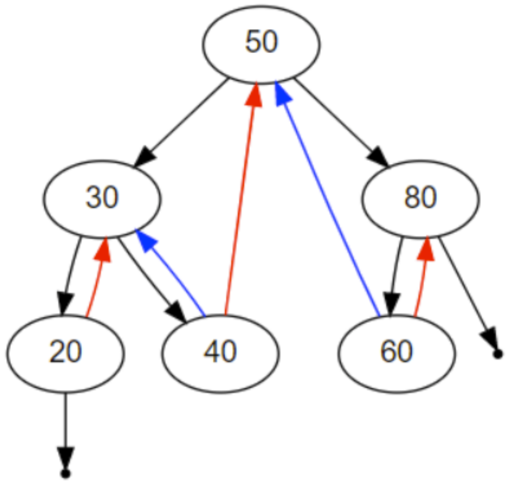
After this completes, it does the same, starting from root but with rthread and current->right. But if rthread is true, it draws a red arrow.

The second function also has a static int nullc. This is passed with post-increment to the printTree_null() function, so that every null node is named uniquely.

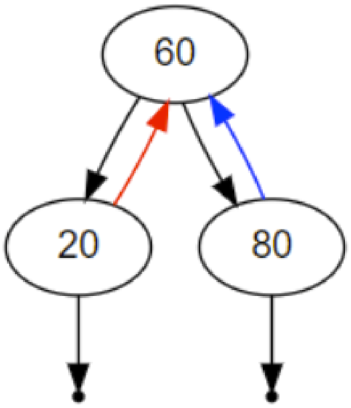
The third function is basically a driver function.
It first creates and opens a file in write mode.
It then gives some starting commands in the graph file.
If root is NULL, it does not input anything in the graph file. Else if root has left and right NULL, it just feeds a single node value into the file. Else, it calls the printTree_main() function.
It then gives ending commands to the file, and closes the file.

After closing, we open the cmd, convert the .gv file to .svg image, and open the image using the system() function.

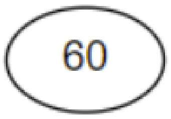
Test Cases:



RED: rthread, BLUE: lthread



RED: rthread, BLUE: lthread



RED: rthread, BLUE: lthread