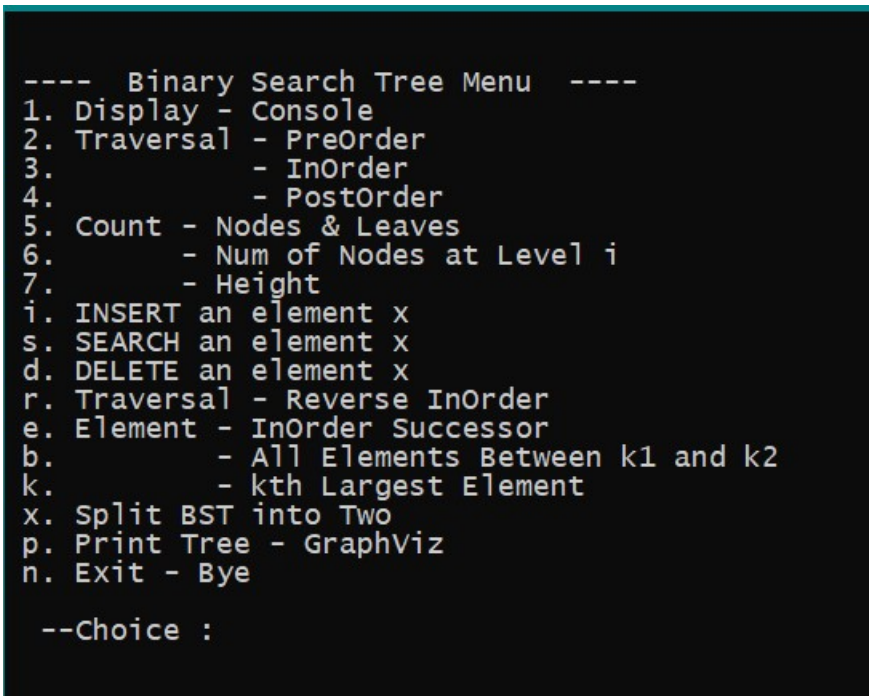
DS LAB CS 513

Assignment 01 (BST)

Roll No: 214101058



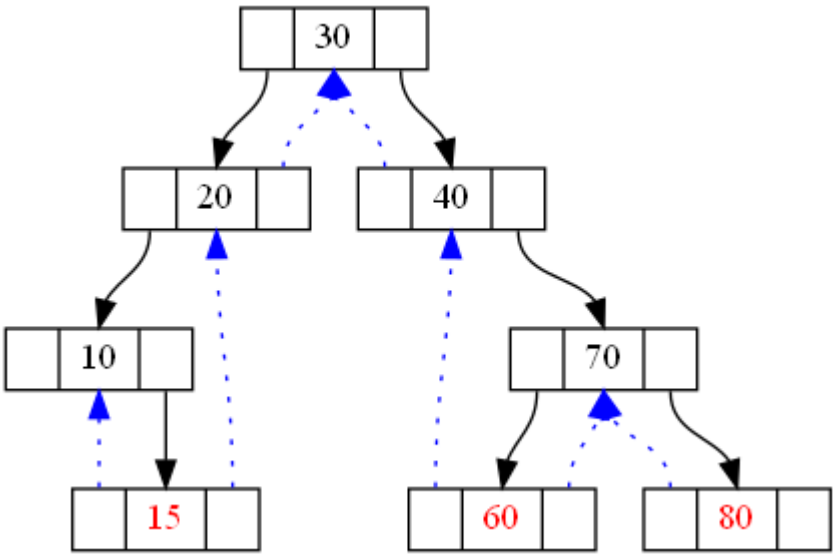
1. **Insert(x)** – insert an element x (if not present) into the BST. If x is present, throw an exception.

   Logic:

   - After taking element x as an input, we are finding its location in a tree also location of its parent going down the tree from the root node.
   - If it exists, then we are throwing an exception and not inserting it.
   - If it doesn't exist, then we get its parent node location through which we will insert.
   - Creating New Node for insertion and initialising with appropriate value.
   - isLeftThread (isRightThread) will be TRUE for a node which having Left pointer (Right Pointer) pointing to their inorder thread location.
   - Then we are inserting the element x using parent pointer at its appropriate location.
   - We are also maintaining count of Number of Nodes in its Left Subtree and Right Subtree for each node.

   Test Case: Insertion: 30, 20, 40, 70, 80, 60, 10, 15.

Output:



Test Case: Insertion: 30, 20, 40, 70, 80, 60, 10, 15. Again 20.

Output:

```
i. INSERT an element x
s. SEARCH an element x
d. DELETE an element x
r. Traversal - Reverse InOrder
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : i

--------
Enter Infomartion (int) for new BST Node .. : 15
       Duplicate Values Detected
```

2. **Search(x)** -- search an element x, if found return its reference, otherwise return NULL.

   Logic:

   - After taking input element that is needed to search, we are finding its location down the tree from the root node.
   - If it is equal to current node key value, return its reference.
   - It is less than current node key value, go to left subtree.
   - Else go to right subtree.

- Until leaf node is reached, or No reference found.

Test Case: Element = 40 (present)

Output:

```
s. SEARCH an element x
d. DELETE an element x
r. Traversal - Reverse InOrder
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : s

--------
Enter a Node value to SEARCH : 40

Search Successful for Element (40).
       Its parent value and address: 30 , 0xc81ad0
       Its location: 40 , 0xc81b30
```

Test Case: Element = 15 (not present)

Output:

```
s. SEARCH an element x
d. DELETE an element x
r. Traversal - Reverse InOrder
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : s

--------
Enter a Node value to SEARCH : 35

Element (35) Not Found.
```

3. Delete(x) -- delete an element x, if the element x is not present, throw an exception.

Logic:

- Using Search function, search the position of element x to be deleted.
- Depending on the node to delete and its number of children we are calling three cases.

  case B: Node having both the children.

case C: Node having single child.

case A: Node having no children.

- Case A: It (node) will simply delete leaf node and update pointers of parent appropriately.
- Case C: It will find its child node and update it with parent appropriately, then if it has left subtree, we will update its predecessor right pointer to its successor, else its successor left pointer to its predecessor.
- Case B: We find its successor to, replace it with current node and then proceed to delete that node according to case A or case B.
- Simultaneously updating left subtree or right subtree nodes count in the tree from root node to the node being deleted.

Test Case: Element = 15 (case A)

Output:

```
d. DELETE an element x
r. Traversal - Reverse InOrder
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : d

--------
Enter a Node value to DELETE : 15

        Element Deleted

                          80
                70
                          60
           40
Root->:  30
           20
                10
```

Test Case: Element = 20 (Case C)

Output:

```
d. DELETE an element x
r. Traversal - Reverse InOrder
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : d

--------
Enter a Node value to DELETE : 20

        Element Deleted

                          80
                70
                          60
           40
Root->:  30
           10
```

Test Case: Element = 70 (Case B)

Output:

```
d. DELETE an element x
r. Traversal - Reverse InOrder
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : d

--------
Enter a Node value to DELETE : 70

        Element Deleted

                80
                      60
           40
Root->:  30
           10
```

Test Case: Tree Empty

Output:

```
d. DELETE an element x
r. Traversal - Reverse InOrder
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : d

--------
Enter a Node value to DELETE : 15

                => Tree empty
```

Test Case: Not Present

Output:

```
d. DELETE an element x
r. Traversal - Reverse InOrder
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : d

--------
Enter a Node value to DELETE : 35

               =>  Item not present in tree
```

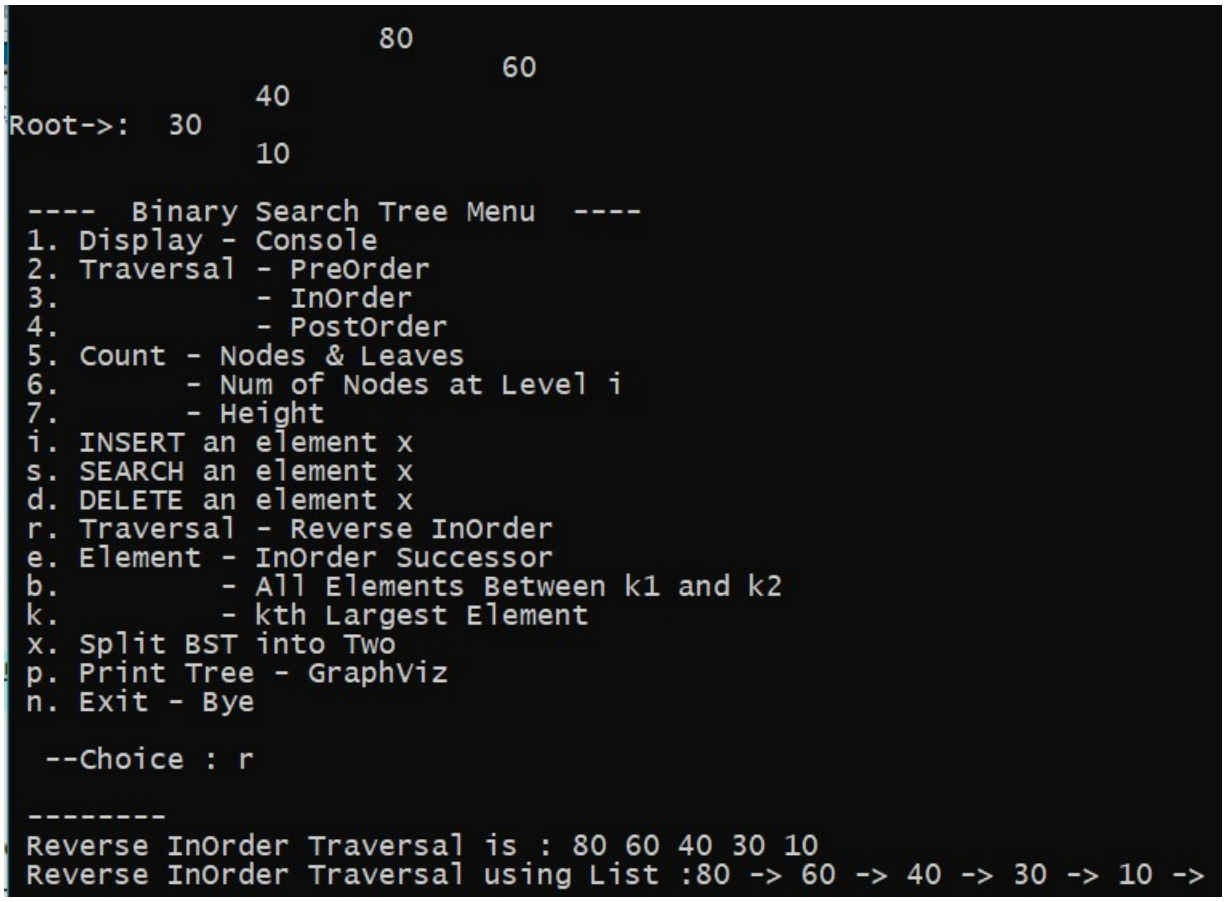
4. reverseInorder() -- returns a singly linked list containing the elements of the BST in max to min order. You should not use any extra stack and use threading for non-recursive implementation.

   Logic:

   - Go to Right most element from Root Node.
   - Print its value, save it into Linked List object at the end and finds its inorder Predecessor. Repeat until pointer is NULL.
   - For inorder predecessor, we go to rightmost child of left subtree of the node.

   Test Case: Elements = 30, 10, 40, 70, 80, 60

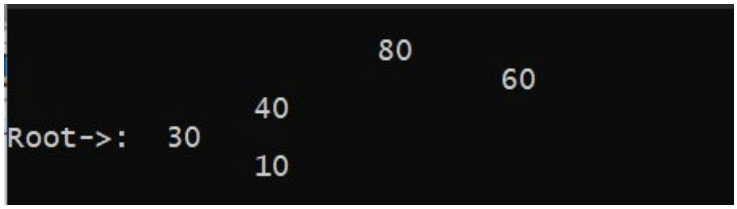
Output:



5. successor(ptr) -- returns the key value of the node which is the inorder successor of the x, where x is the key value of the node pointed by ptr.

Logic:

- If right link is thread to inorder successor we return that pointer.
- Else, we go to leftmost child of the right subtree of the node.



Test Case: Element = 40(present)

Output:

```
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : e

--------
Enter a Node value to find its inorder successor : 40

       Its inorder successor is: 60
```

Test Case: Element =18 (not present)

Output:

```
e. Element - InOrder Successor
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : e

--------
Enter a Node value to find its inorder successor : 80

       Its inorder successor does not exit:
```

6. **split(k)** -- split the BST into two BSTs T1 and T2, where keys(T1) <= k and keys(T2) > k. Note that, k may / may not be an element of the tree. Your code should run in O(h) time, where h is the height of the tree. Print the inorder of both the BSTs T1 and T2 using recursive/non-recursive implementation within this function.

   Logic:

   - First, we clone an existing tree already created in the program using copy constructor. Function creates clone by creating node, copying key and left and right pointers and copying threads link by maintaining mapping from given tree node to clone.
   - We maintain three pointers, fptr, secptr, and cutptr.
   - Using secptr we travel left (or right) based on the k value and secptr key value. If it needs to travel left, then we make fptr pointing to secptr, and secptr to the left child.

Else for right, we make fptr pointing to secptr, and secptr to the right child till leaf node.

We repeat it till leaf node is reached or we required to change direction (from travelling left to right, vice versa).

- Once out of above loop (because secptr key falls beyond k), then we need to create a cut in the link, using fptr based on the direction we are travelling above.
- Change the direction (since we need to link nodes back).

  If we were travelling in left, then we need to update right pointer of last node where we made a cut.

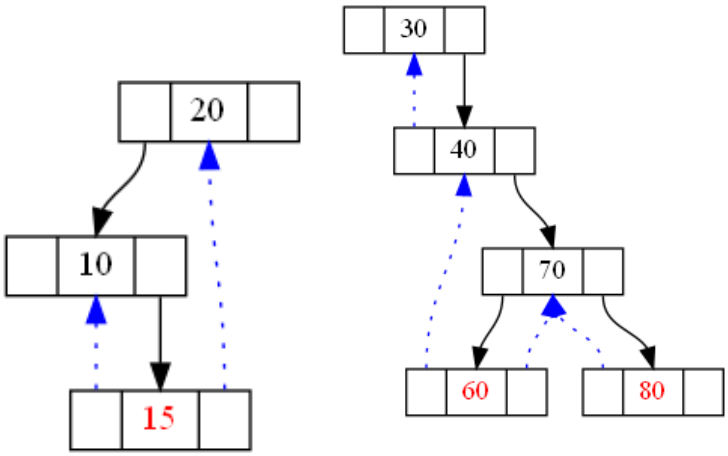
  If we were travelling in right, then we need to update left pointer of last node where we made a cut.
- If it is a first cut, then we initialize the root 2 to the secptr.

  Else we update the cutptr left (or right) pointer to the secptr. Also changing its thread values to show it is not a thread.
- Updating cutptr to current fptr and fptr to NULL.
- Printing their inorder.

Test Case: k = 20(present, left subtree)

Output:

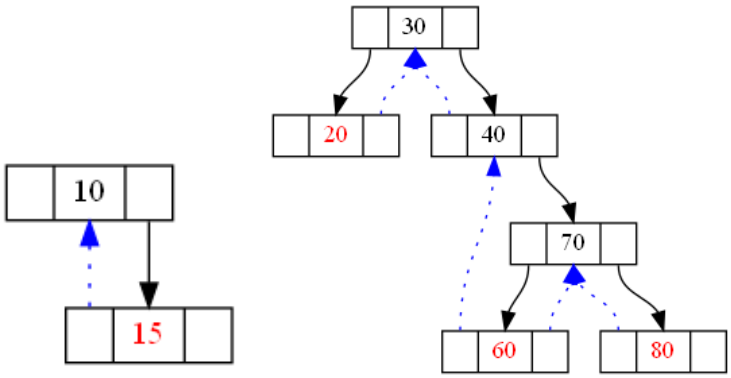
```
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : x

--------
Split BST into Two
Enter a value of k to make a split : 20

                        80
                70
                        60
            40
Root->:  30
            20
                        15
                10
Tree 1 inOrder : 30 40 60 70 80
Tree 2 inOrder : 10 15 20
Printing Tree(s):
        printed.
        printed.
```



Test Case: Element = (not present, left subtree)

Output:

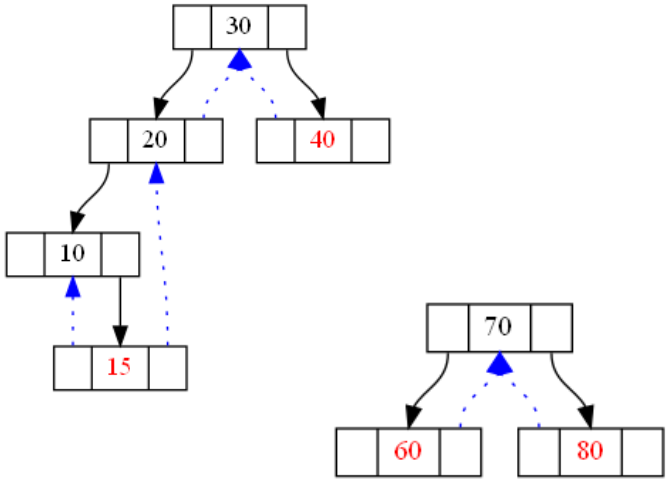
```
 --Choice : x

--------
Split BST into Two
Enter a value of k to make a split : 18

                        80
                70
                        60
            40
Root->:  30
            20
                        15
                10
Tree 1 inOrder : 20 30 40 60 70 80
        printed.
Tree 2 inOrder : 10 15
```



Test Case: Element = 40 (present, right subtree)

Output:

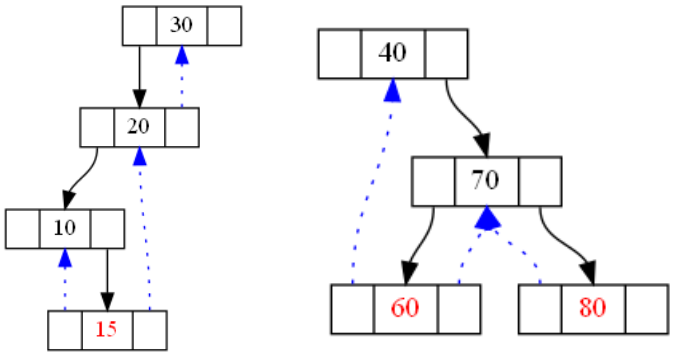
```
Split BST into Two
Enter a value of k to make a split : 40
30 40 60 70 80
                        80
                70
                        60
            40
Root->:  30
            20
                        15
                10
Tree 1 inOrder : 10 15 20 30 40
        printed.
Tree 2 inOrder : 60 70 80
        printed.
Printing Tree(s):
```



Test Case: Element =35 (not present, right subtree)

Output:

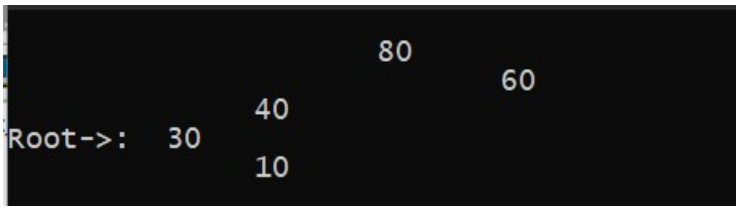
```
 --Choice : x

--------
Split BST into Two
Enter a value of k to make a split : 35
20 30 40 60 70 80
                        80
                70
                        60
            40
Root->:  30
            20
                        15
                10
Tree 1 inOrder : 10 15 20 30
        printed.
Tree 2 inOrder : 40 60 70 80
```



7. **allElementsBetween(k1,k2)** -- returns a singly linked list (write your own class) that contains all the elements (k) between k1 and k2, i.e., k1 <= k <= k2. Your code should run in O(h + N) time, where N is the number of elements that appears between k1 and k2 in the BST.

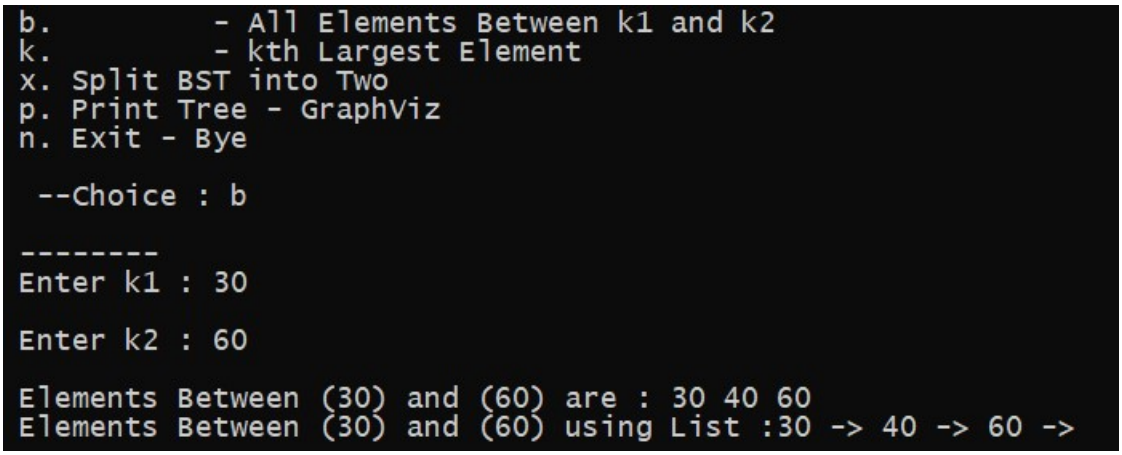
   Logic:

   - We find location of k1 or k2 in the tree, or else their parent location if they don't exist. (O(h))
   - Using location of k1, we travel from that node using inorder successor until we reach k2. (O(k))



Test Case: k1 = 30, k2 = 60(present)

Output:



Test Case: k1 = 15, k2 = 65 (not present in tree)

Output:

```
b.         - All Elements Between k1 and k2
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : b

--------
Enter k1 : 15

Enter k2 : 65

Elements Between (15) and (65) are : 10 30 40 60
Elements Between (15) and (65) using List :10 -> 30 -> 40 -> 60 ->
```

8. **kthElement(k)** -- finds the k-th largest element in the BST and prints the key value. Your code should run in O(h) time.

   Logic:

   - Using count of elements in its left subtree or right subtree, we travel down from root.
   - If total count is greater than k value we return NULL.
   - If right count of node plus its root is equal to k then we return that node.
     Else if it is greater then we travel to left subtree subtracting right count and 1 from k value.
   - Else to right subtree.

   Test Case: k = 3th(bounded)

   Output:

```
k.         - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : k

--------
Enter value of k to find kth largest : 3

        3th Largest Element is : 40
```

   Test Case: k = 15th(unbounded)

Output:

```
k.          - kth Largest Element
x. Split BST into Two
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : k

--------
Enter value of k to find kth largest : 15

        Value of k is greater than total nodes
```

9. **printTree()** -- Your program should print the BST (in a tree like structure) [use graphviz]

Logic:

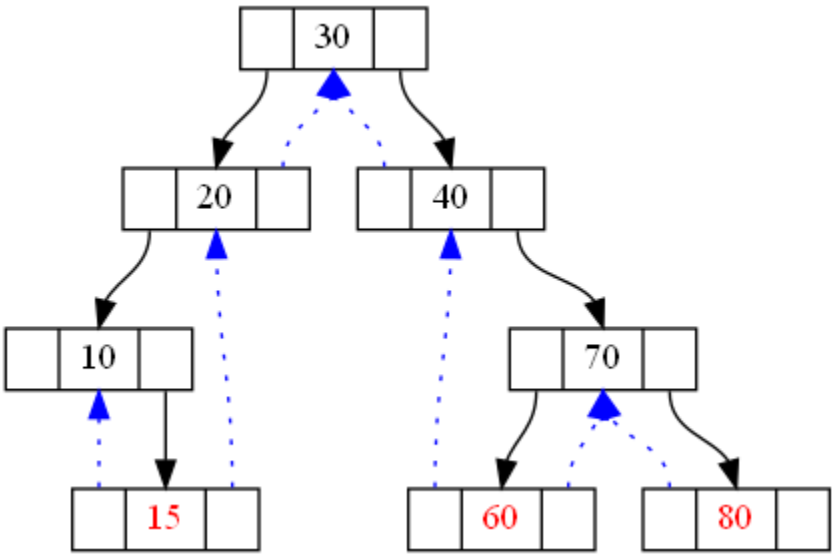
- It travels level order in the tree using BFS traversal and simultaneously write to the file (.gv) in the appropriate format for the graphviz to read.
- Using graphiz command "*dot -Tpng a_print_bst.gv -o a_print_bst.png*" in the console to generate png file.

Test Case: Element = 30, 20, 10, 40, 70, 80, 60, 15

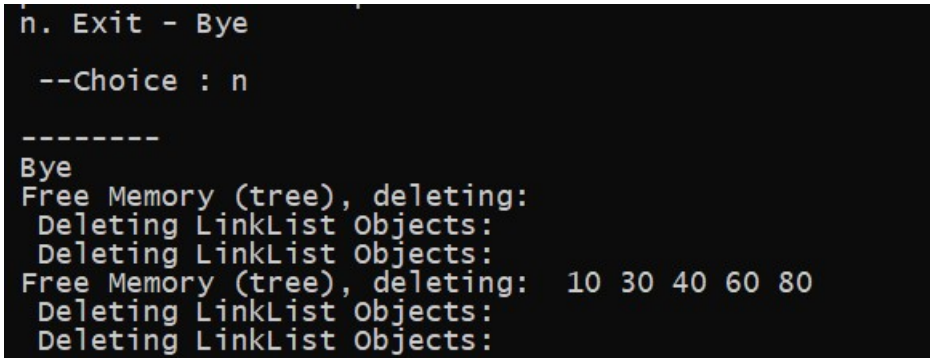
Output:

```
p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : p

--------
Printing Tree:
       printed.
```

dot -Tpng a_print_thbst.gv -o a_print_bst.png



10. EXIT


n. Exit - Bye
--Choice : n

Bye
Free Memory (tree), deleting:
Deleting LinkList Objects:
Deleting LinkList Objects:
Free Memory (tree), deleting: 10 30 40 60 80
Deleting LinkList Objects:
Deleting LinkList Objects: