Functions Logic and Program Output

Utility Functions:-

- treeNode* pred(treeNode* ptr) - Utility function for predecessor,

Given a pointer, it returns pointer to node that is predecessor of current pointer. If leftThread is true, it returns the left ptr node , else returns the rightmost child in left subtree.

It will be used in various functions.

- treeNode* succ(treeNode* ptr) - For a given pointer, returns its successor pointer . If rightThread is true, just returns the right pointer node else returns the leftmost child in right subtree

  Used in various functions.

DeleteNode function uses the below 3 utility functions:-

- treeNode* delete0(treeNode* _root ,treeNode* parent , treeNode* ptr) - This is called when node to delete has no child.

- treeNode* delete1(treeNode* _root , treeNode* parent , treeNode* ptr) - This is called when node to delete has 1 child either left or right.

- treeNode* delete2(treeNode* _root ,treeNode* parent , treeNode* ptr)- This is called when node to delete has 2 children.

Split function uses the below 2 utility functions: -

- vector<treeNode*> split_utility(treeNode* _root , int k, int* visitd)- This function finds and returns 2 root nodes which has element value just <=k ans >k respectively in O(h) time.

  Working is explained in comments briefly.

- void print_inorder(treeNode* _root) - This simply prints tree in Inorder way. To check output after splitting the tree.

Main functions: -

- treeNode* **insert**(treeNode* _root, int _val)- This adds a node to the BST, also changes left and right threads bools for parent of node inserted. Returns root pointer.

- treeNode* **search**(treeNode* _root , int _val)- Searches key value _val in the BST and returns the node ptr else nullptr.

- treeNode* **deleteNode**(treeNode* _root , int _val)-   This call all 3 delete functions accordingly. First finds the node to delete and its parent. Then based on children the node has , calls the utility function. It also manages left/right threads bool for parent of node deleted. Returns root ptr.

- llist* **reverseInorder**(treeNode* _root) - This uses pred() function to get predecessor ptr of each node. We go to the rightmost child then use pred() till we reach the smallest element. Returns a linked list containing the reverse inorder elements.

- int **successor**(treeNode* ptr)- This uses succ() utility function and returns element value of predecessor.

- llist* **allElementsBetween**(treeNode* _root , int k1, int k2) -

First, I found the nodes start and end.

start->val is just greater than or equal to k1 . end->val is just less than or equal to k2.

Then created a linked list to store the elements from start to end. Iterated through both tree and LinkedList simultaneously storing the values in LinkedList.

Returns the linked list.

- int **kthLargest**(treeNode* _root, int k)- Uses Morris traversal algorithm. We keep track of nodes in right subtree and use this to find kth largest. Refer comments, it explains in detail.

- vector<treeNode*> **split**(treeNode* _root , int k)-

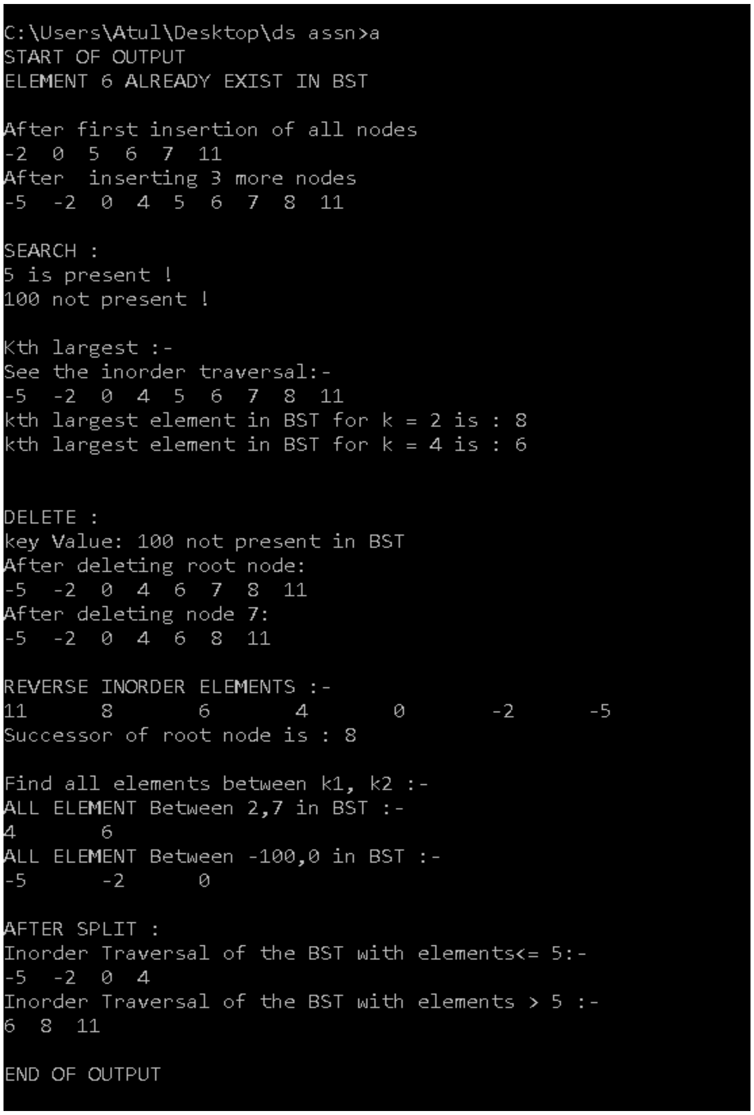
First it calls split_utility to get the 2 root nodes,
then we remove any connected threads in O(h) time.

Then calls print_inorder to print 2 trees in O(n) time.

- void **printTree**(treeNode* _root, const string& name,const string& label) - Prints tree inorder , just for showing output.

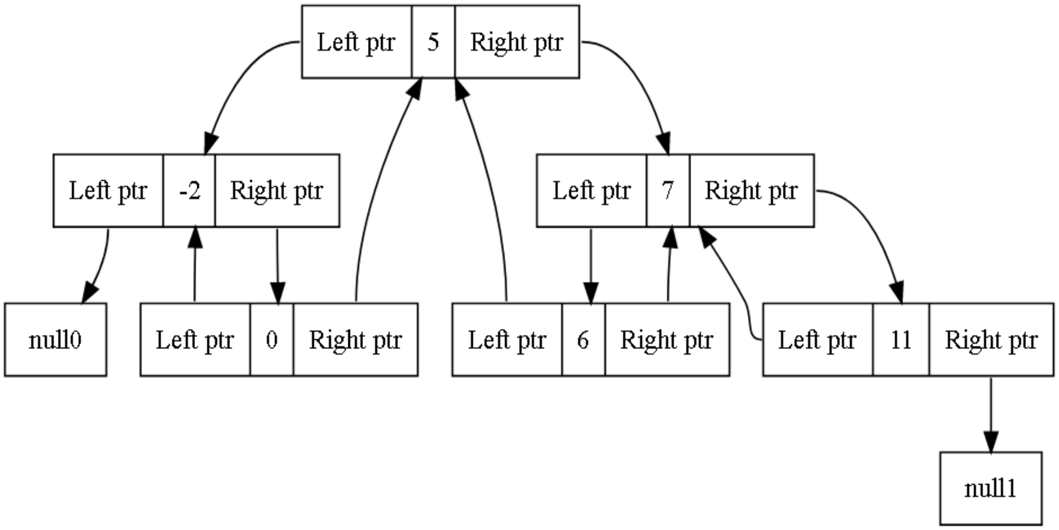
OUTPUT :

This is the command output:



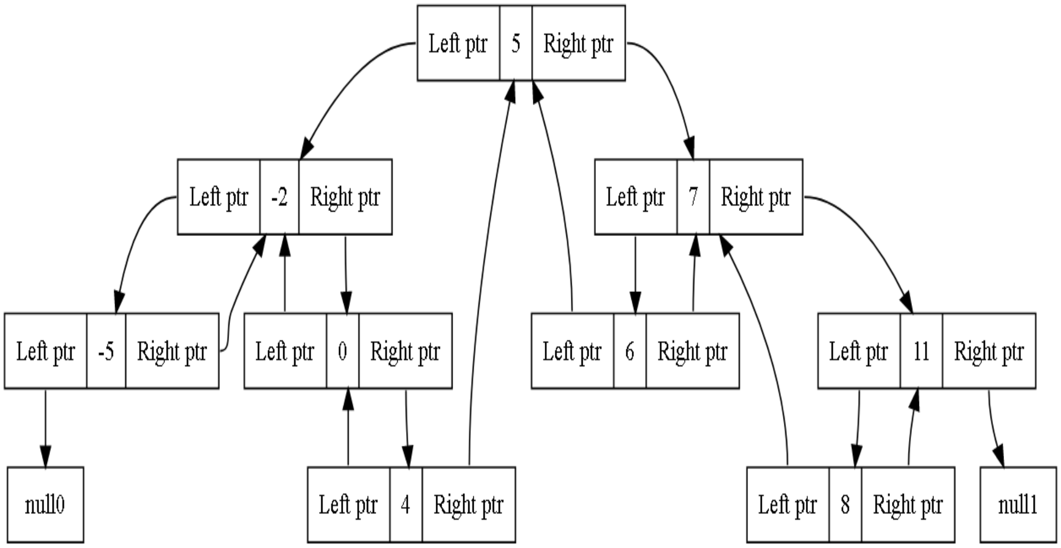
```
C:\Users\Atul\Desktop\ds assn>a
START OF OUTPUT
ELEMENT 6 ALREADY EXIST IN BST

After first insertion of all nodes
-2  0  5  6  7  11
After  inserting 3 more nodes
-5  -2  0  4  5  6  7  8  11

SEARCH :
5 is present !
100 not present !

Kth largest :-
See the inorder traversal:-
-5  -2  0  4  5  6  7  8  11
kth largest element in BST for k = 2 is : 8
kth largest element in BST for k = 4 is : 6


DELETE :
key Value: 100 not present in BST
After deleting root node:
-5  -2  0  4  6  7  8  11
After deleting node 7:
-5  -2  0  4  6  8  11

REVERSE INORDER ELEMENTS :-
11      8       6       4       0       -2      -5
Successor of root node is : 8

Find all elements between k1, k2 :-
ALL ELEMENT Between 2,7 in BST :-
4       6
ALL ELEMENT Between -100,0 in BST :-
-5      -2      0

AFTER SPLIT :
Inorder Traversal of the BST with elements<= 5:-
-5  -2  0  4
Inorder Traversal of the BST with elements > 5 :-
6  8  11

END OF OUTPUT
```

TREES FROM GRAPHVIZ:



After first insertion of all nodes

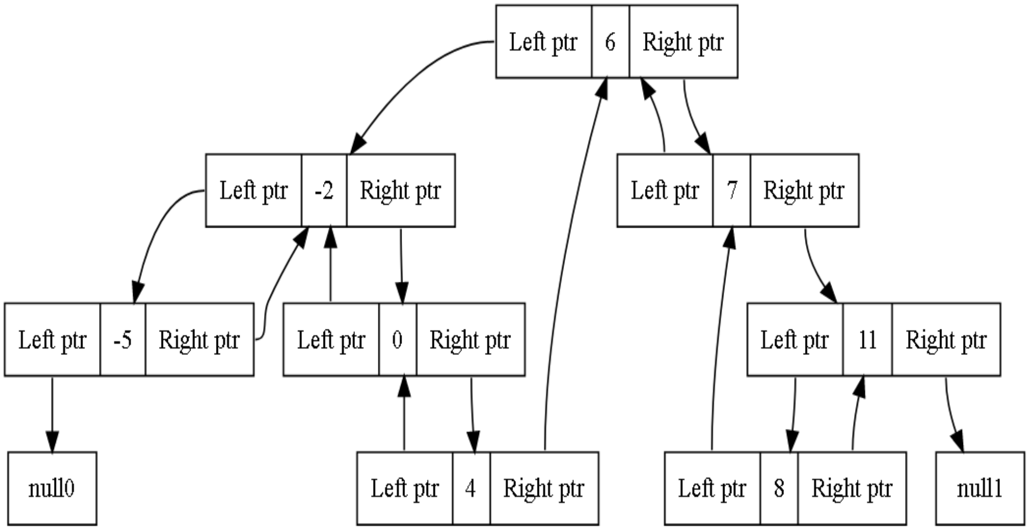
First inserted few elements, notice the left ptr of 6 points to pred(6) and right pointer of 0 points to succ(0) = 5



After inserting 3 more nodes

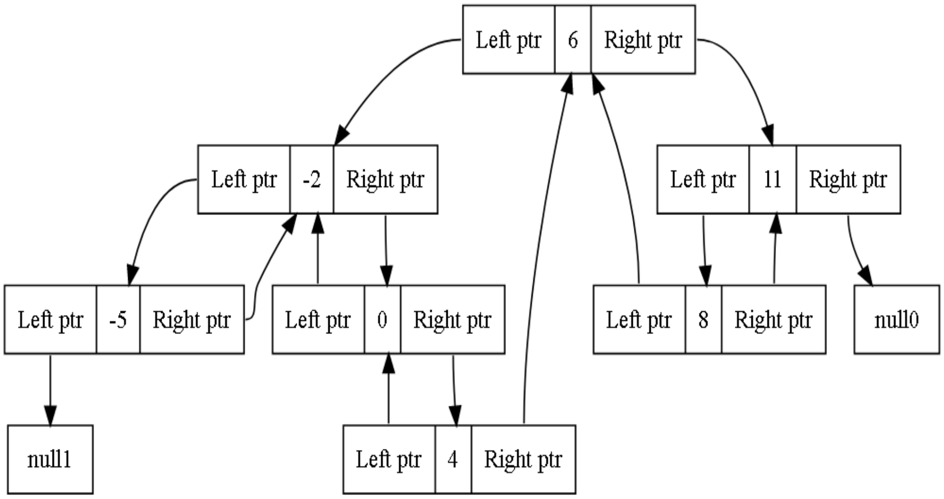
After inserting 3 more nodes: -5,4,8. Now the right ptr of 0 is not pointing to succ(0) but to its right child 4. Similar for left ptr of 11.

Now we delete root node 5.



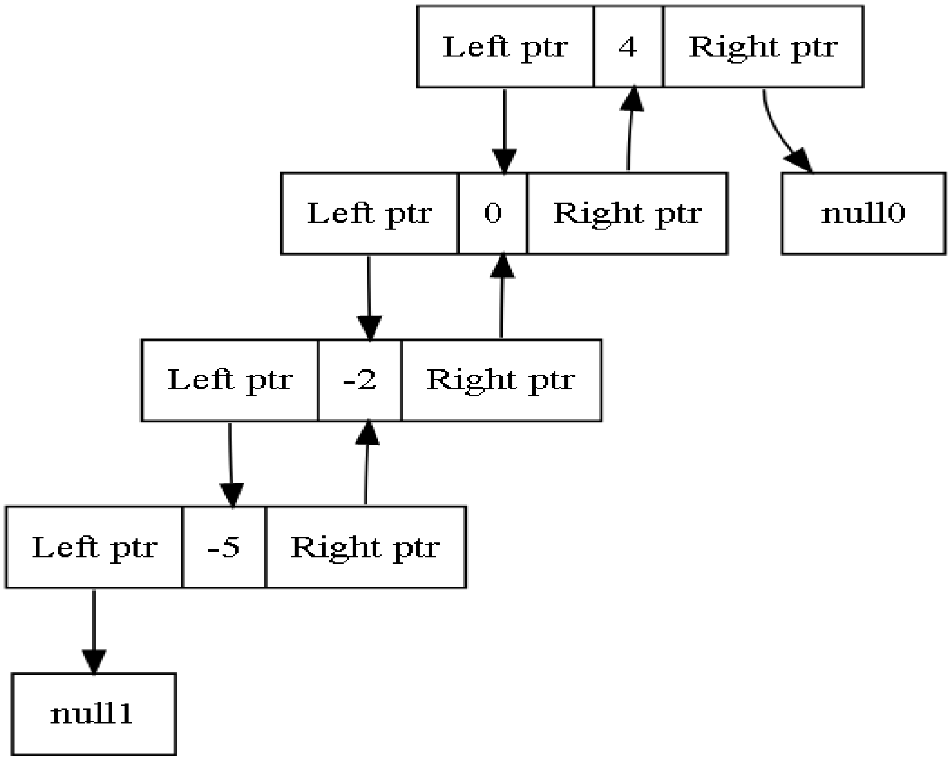
After deleting root node

After deleting root node 5 , notice how pointers connected to 5 are now connected to/from 6. After this node 7 is deleted, given below.



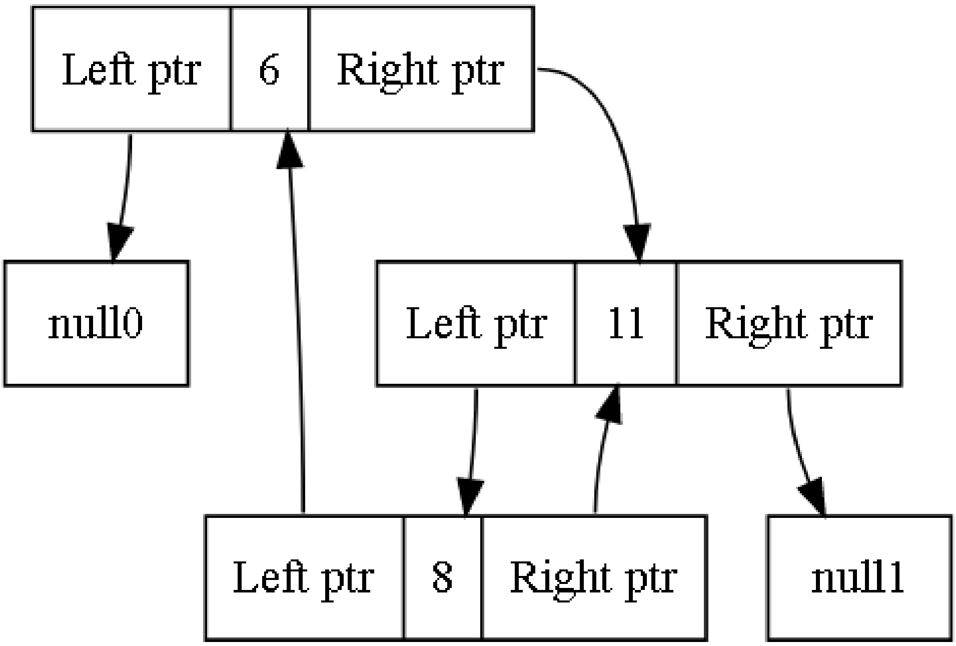
After deleting node 7

In the above tree, we perform split operation to get below 2 trees. First contains elements less than or equal to 5. Second tree has elements greater than 5.



Smaller elements tree After splitting main tree

Notice 4 is just less than 5 so it becomes the root of smaller element tree. And 6 is just greater than 5 so it becomes root of larger element trees.



Bigger elements tree After splitting main tree

Notice the left ptr of 6 and right ptr of 4 in main tree are connected to other element and in split trees, these connections are removed.