Assignment 1

Write a C++ class implementing binary search tree that supports the following operations:

Understanding the data structures which are used to implement –

1. **Binary Search Tree Node class**

```
class BSTNode {
public:
    int key, rCount;
    BSTNode* left, *right;
    bool leftLinkType, rightLinkType;

    // When the node is first created, it is a leaf node. So, both links must
be THREAD
    BSTNode(int key) { this->key = key; this->rCount = 0;
left = right = NULL; leftLinkType = rightLinkType = THREAD; }
};
```

***key* –** stores integer data

left & right – these are 2 pointers which can point to the same type of the node object

leftLinkType & rightLinkType – Boolean variables which states that left and right pointers are pointing to the **CHILD** or **THREAD** to predecessor or successor.

Here in the program I have defined 2 macros named as **CHILD** and **THREAD**

```
//if it is pointing to child
#define CHILD   0 // false means link is pointing child
//if it is pointing to successor or predecessor
#define THREAD  1 //true means link is pointing thread
```

2. **Singly Linked List**

```
class LLNode {
    public:
        int data;
        LLNode *next;
        LLNode(int x){
            this->data = x;
            this->next = NULL;
        }
};
```

data – integer data value

***next* –** pointer to same type of the object

3. Threaded BST class

```
class ThreadedBST {
class ThreadedBST {
private:
    BSTNode* root;
    void eraseTreeNodes(BSTNode* node);
    BSTNode* copy_helper(BSTNode* copy_from) const;
public:
    ThreadedBST() { root = NULL; }
    ~ThreadedBST() { eraseTreeNodes(root); root = NULL; }

    ThreadedBST(const ThreadedBST& other);

    void insert(int x);
    BSTNode* search(int key);
    void deleteNode(int key);
    LLNode* reverseInorder();
    int successor(BSTNode* node);
    void split(int k);
    LLNode* allElementsBetween(int k1, int k2);
    int kthElement(int k);
    void printTree();

    BSTNode* getRoot() { return root; }
    BSTNode* min();
    BSTNode* max();
    BSTNode* predecessor(BSTNode* node);
    BSTNode* next(BSTNode* node);
    LLNode* inorder();
};
```

root- it is the pointer to the BSTNode class object

Extra function which are implemented as helper for the given function –

1) ***min()* –** returns the minimum node which is available in the Threaded Binary Search Tree

- It will traverse to the left most node of the tree and return the reference of that node

2) ***max()* --** returns the maximum node which is available in the Threaded Binary Search Tree

- It will traverse to the right most node of the tree and return the reference of that node

3) ***predecessor(ptr)* –** returns the reference of predecessor node of the node pointed by the ptr

- there are **2 cases**
    1. The ***leftLinkType*** is **THREAD** , in that case it is thread than we directly go to that node and return reference of that node
    2. The ***leftLinkType*** is **CHILD**, in that case we will traverse to the right most node of the left subtree

4) ***next(ptr)* –** returns the reference of the node which is successor node of the node pointed by ptr

- there are **2 cases**
    1. The ***rightLinkType*** is **THREAD** , in that case it is thread than we directly go to that node and return reference of that node
    2. The ***rightLinkType*** is **CHILD**, in that case we will traverse to the left most node of the right subtree

5) ***inorder()* –** returns singly linked list containing inorder of the given threaded bst

- we will start from the minimum node (by calling *min()*)
- we will traverse to all the successor and make linkedlist
- when there is no successor, we will return the generated linked list

Functions which are implemented -

1. *insert(x) --*

First, I found out the proper place at which I am going to insert the new node.

Now there are 2 cases -

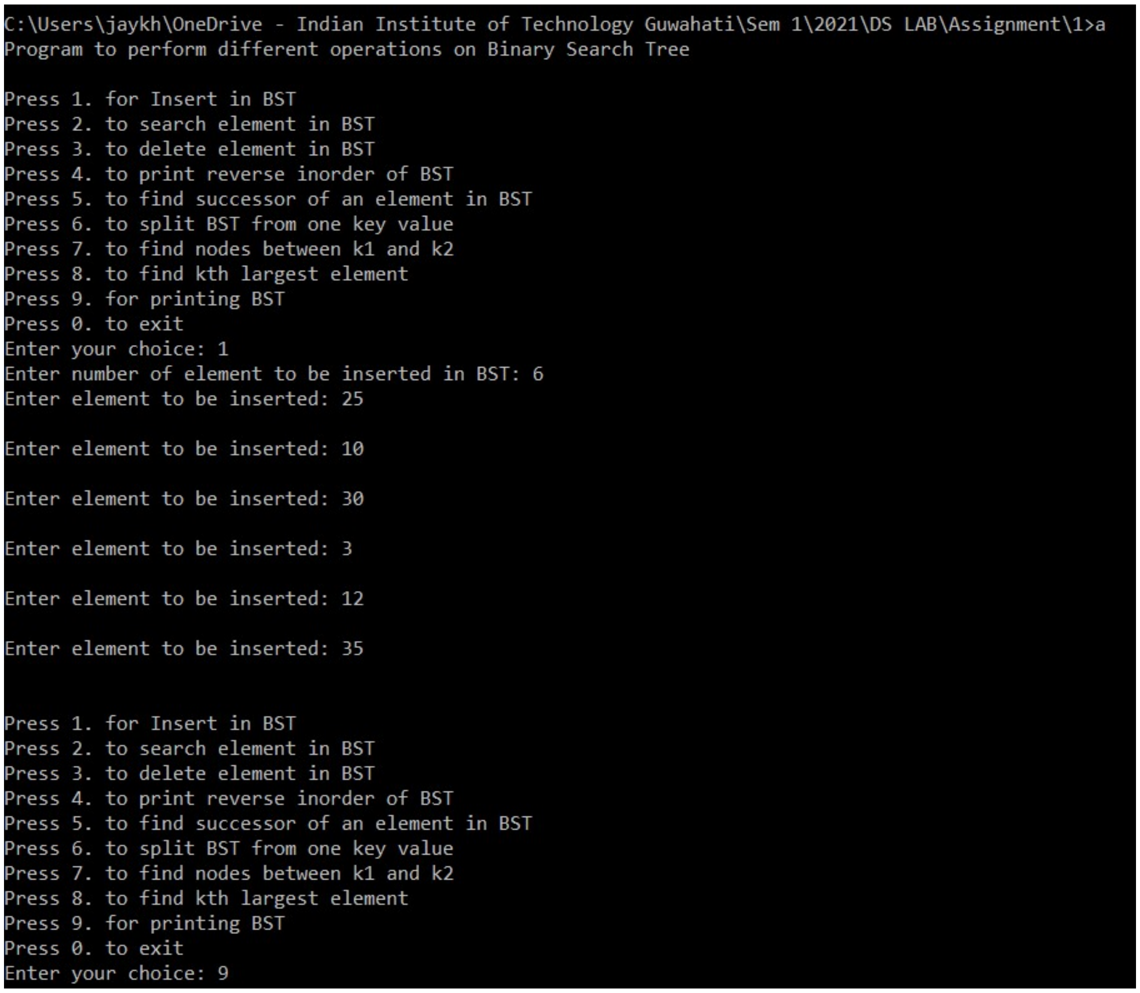
1) **Tree was initially empty so there is no node**

   In this case, when we find the *parentNode*, it will come out as *nullptr*. So, we can directly insert and assign root value to be node itself.

2) **Tree was having some nodes in prior**

   In this case, we will find the *parentNode* first and then we will insert the new node at the left or right according to key value which is being inserted.

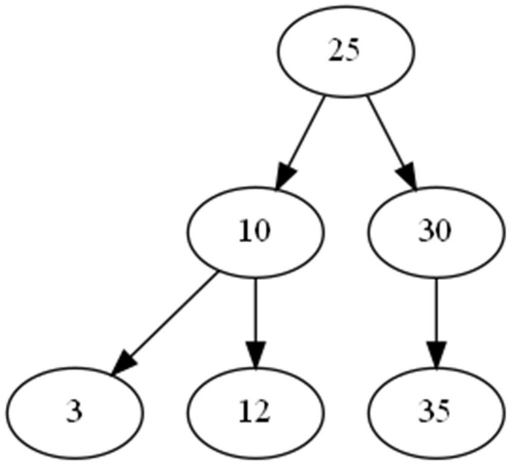
Commands:



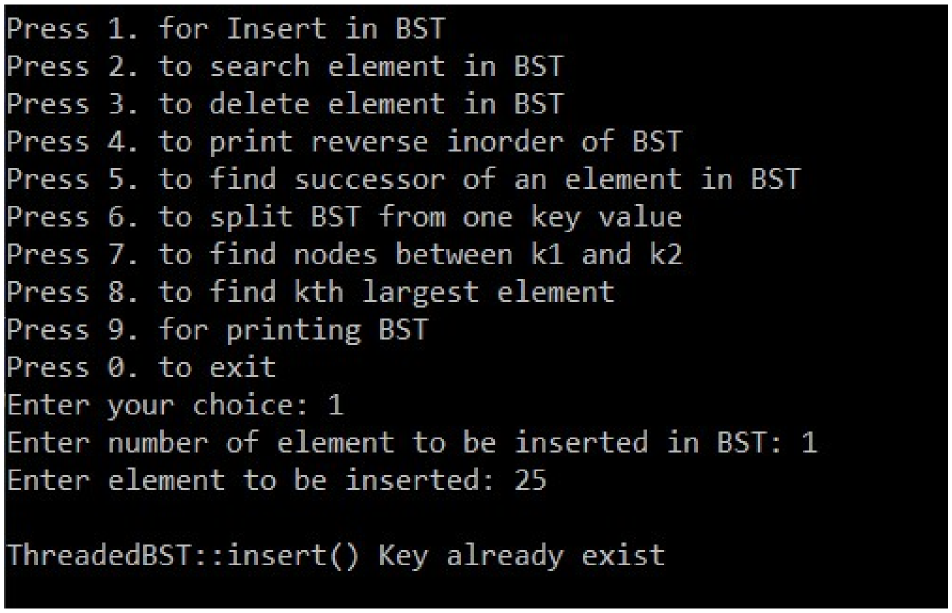
```
C:\Users\jaykh\OneDrive - Indian Institute of Technology Guwahati\Sem 1\2021\DS LAB\Assignment\1>a
Program to perform different operations on Binary Search Tree

Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 1
Enter number of element to be inserted in BST: 6
Enter element to be inserted: 25

Enter element to be inserted: 10

Enter element to be inserted: 30

Enter element to be inserted: 3

Enter element to be inserted: 12

Enter element to be inserted: 35


Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 9
```

OUTPUT:

Test case 1: We insert all the element and check the BST via printing it, below is the output.



Test case 2: Inserting the element which already exists in the BST



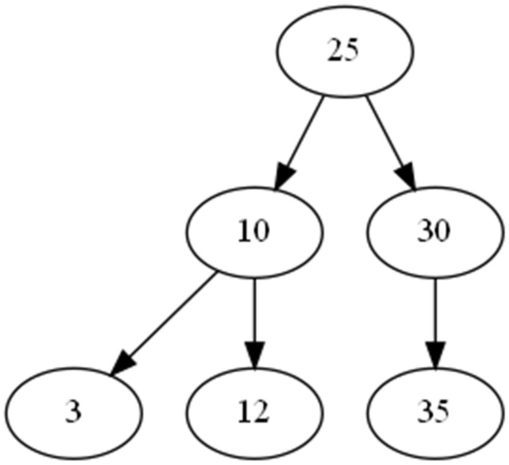
```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 1
Enter number of element to be inserted in BST: 1
Enter element to be inserted: 25

ThreadedBST::insert() Key already exist
```

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

This is simple traversal of the nodes by comparing the nodes, if node matches we found the node else value doesn't exist in a BST.

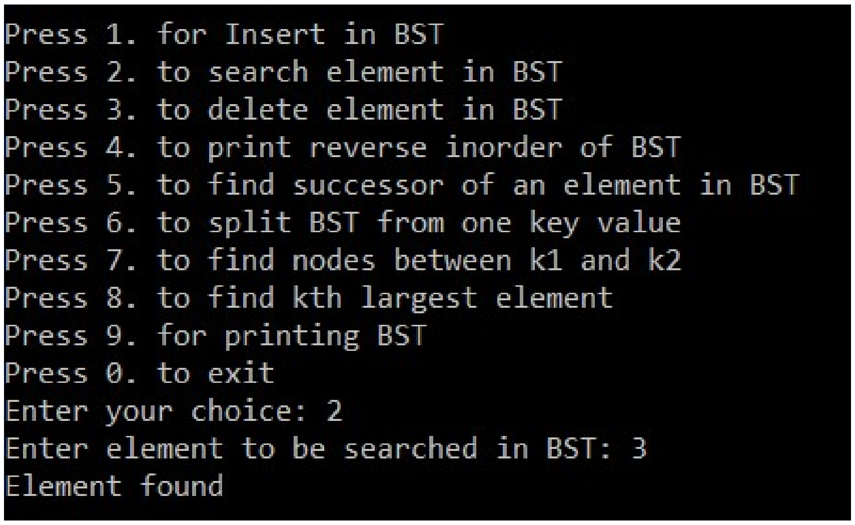
We compare the key value which is x to the root if it is less than the root value then we search to the left subtree if it is greater than the root value then we search the key in right subtree. If we find the root value equal to the key, then we return that node otherwise in the last we return *NULL*.

Input:



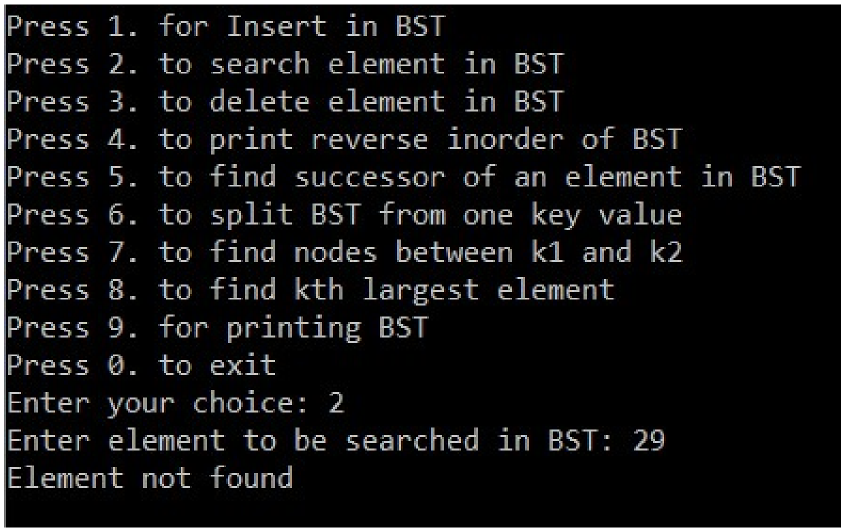
Output:

Test case 1: Element is present in the BST. i.e. x = 3



```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 2
Enter element to be searched in BST: 3
Element found
```

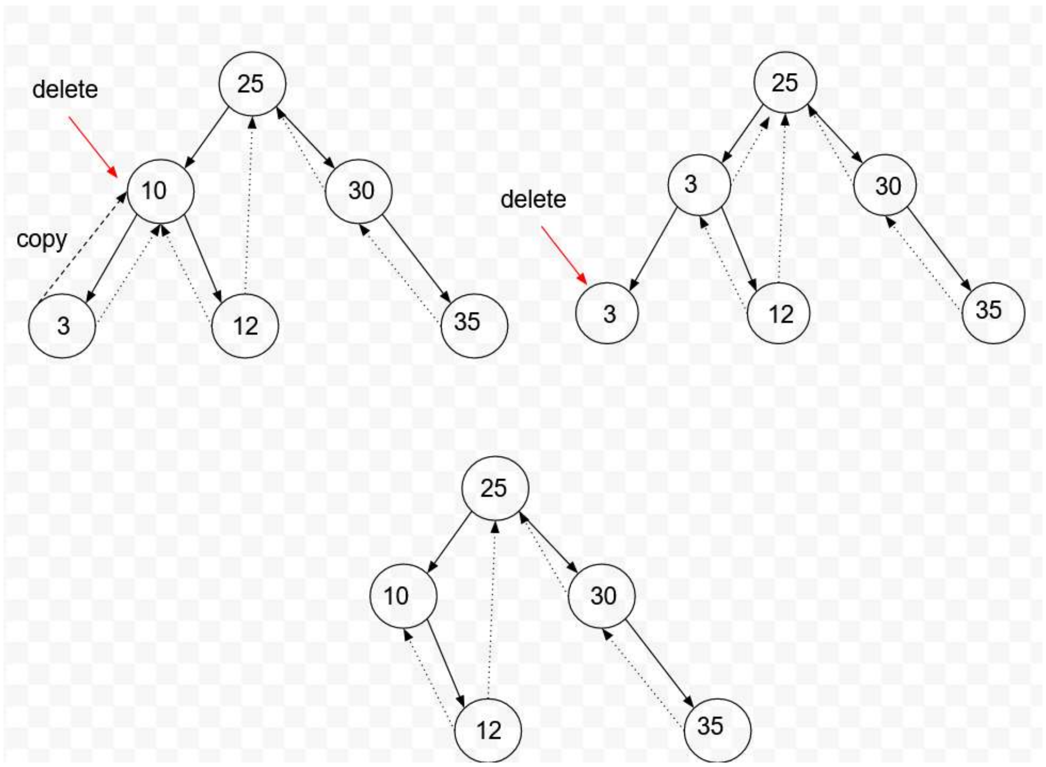
Test case 2: Element is not present in the BST



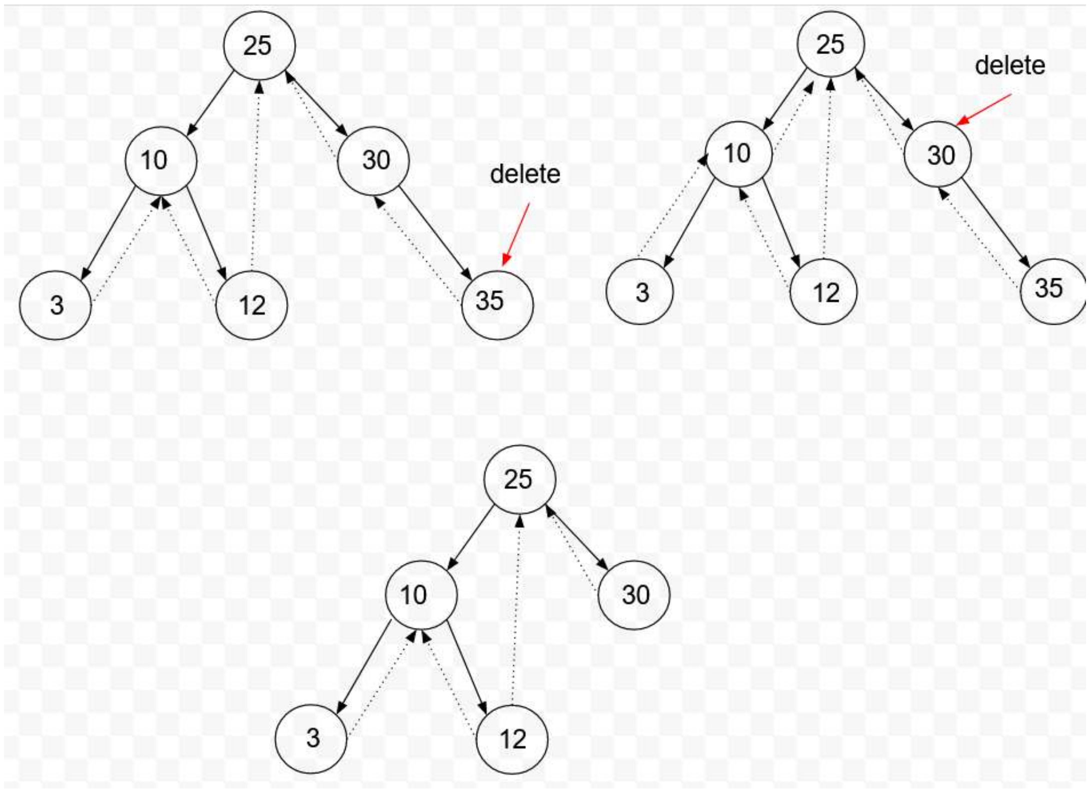
```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 2
Enter element to be searched in BST: 29
Element not found
```

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

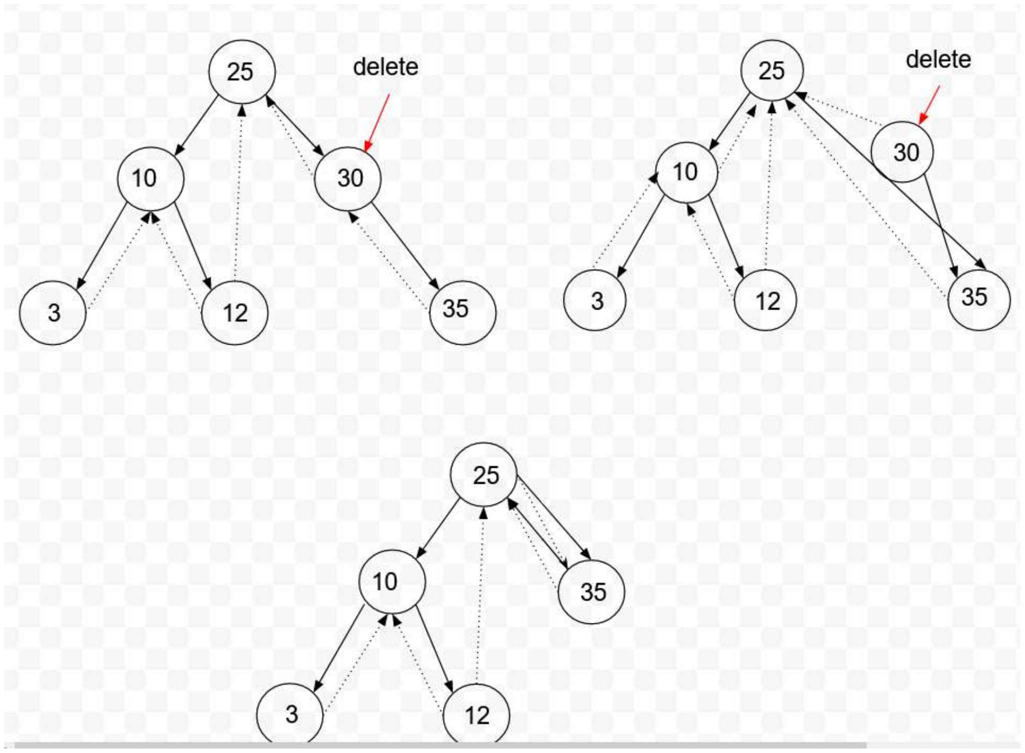
- There are 3 cases –
    1. **If the node being deleted has both child**
       In that case, we copy the predecessor value to the node being deleted and then we delete the predecessor node.



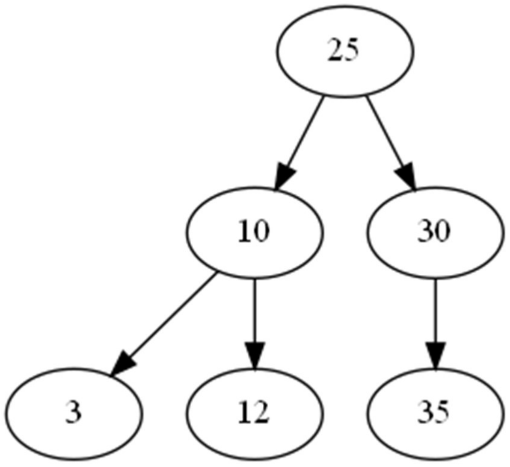
2. **If the node being deleted has no child**
   In that case, we will change the parent node's left or right child point to directly the left or right child of the node being deleted. Also, change the type of right link type to **THREAD**



3. **If the node being deleted has single child**
    - If the single child is right child then we find the minimum node in the right subtree of the node being deleted and then we change the left child of the minimum node to parent node and we make parent node to point the right child of the node being deleted.
    - In case the single child is left child then we find the maximum node in the left subtree of the node being delted and we change the right child of the maximum node to parent node (parent node is new successor of the maximum node of the left subtree) and we make parent node to point to the left child of the node being deleted.

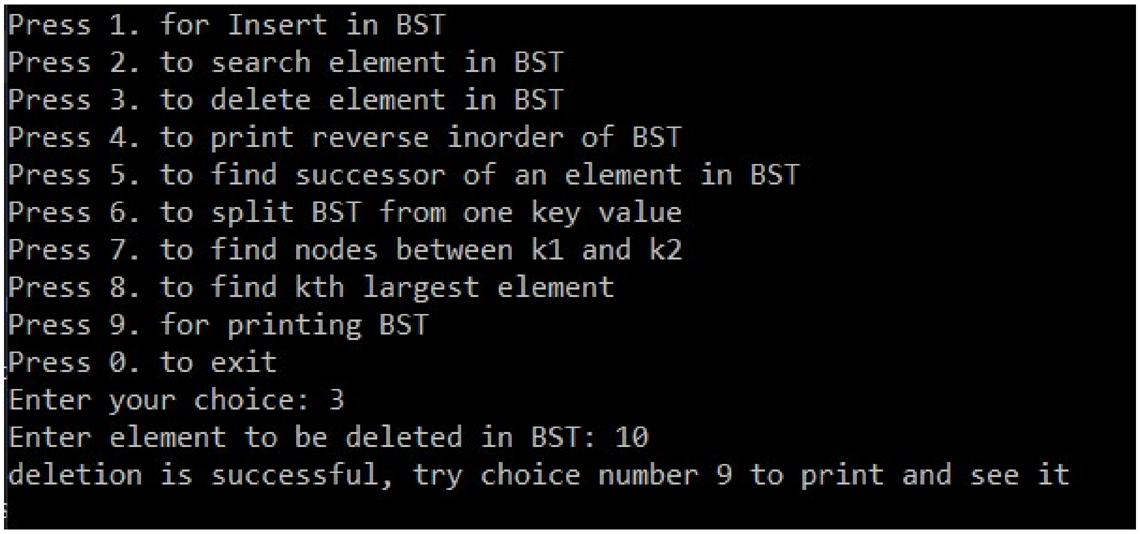


Input:

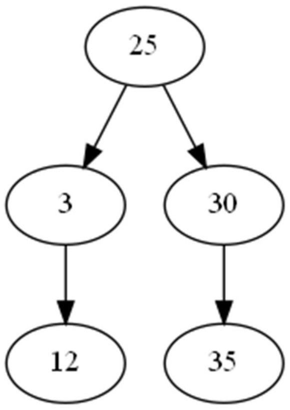


Output:

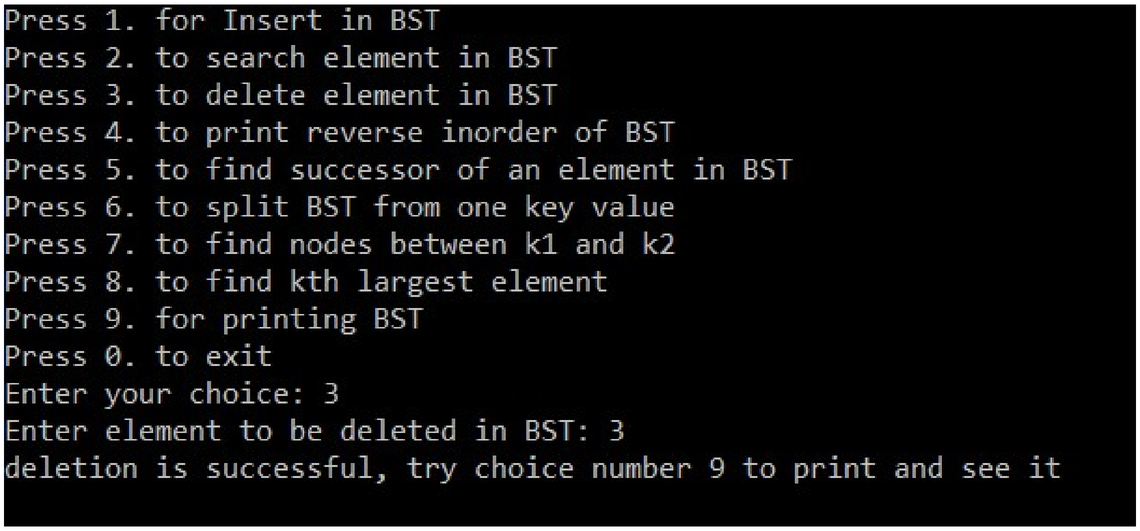
Test case 1: deleting the node which has 2 children, i.e. 10



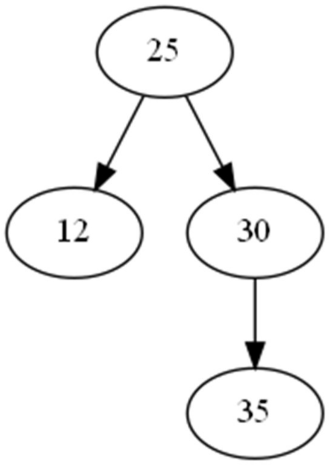
```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 3
Enter element to be deleted in BST: 10
deletion is successful, try choice number 9 to print and see it
```



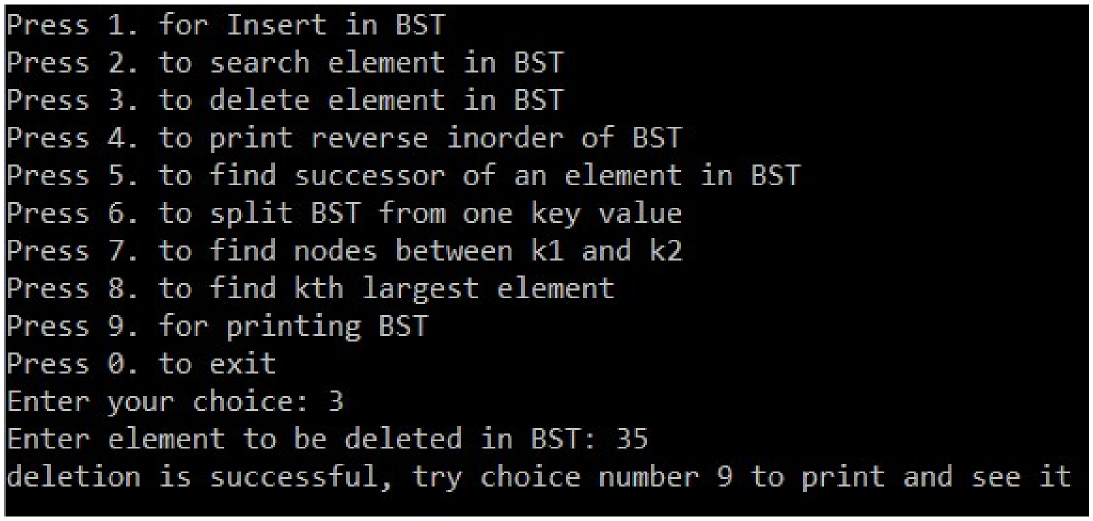
Test case 2: deleting the node which has 1 child from the above, i.e. 3



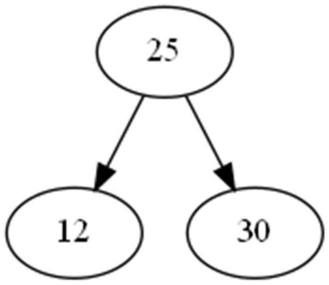
```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 3
Enter element to be deleted in BST: 3
deletion is successful, try choice number 9 to print and see it
```



Test case 3: deleting the node which has no child in the previous tree, i.e. 35



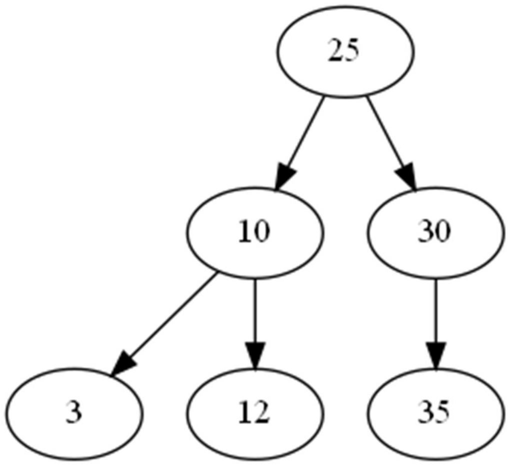
```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 3
Enter element to be deleted in BST: 35
deletion is successful, try choice number 9 to print and see it
```



4. ***reverseInorder()* --** returns a singly linked list containing the elements of the BST in max to min order.

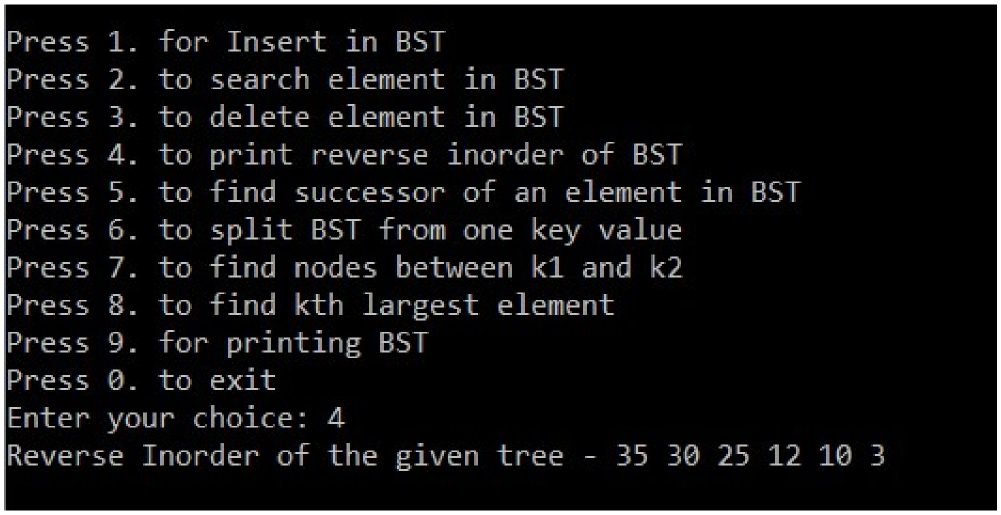
- We will start with the maximum node in the tree and we will call the predecessor function and make a linked list simultaneously.
- When the ***predecessor(cur)*** returns the NULL then we have traversed all the nodes and now we can return the singly linked list.

Input:



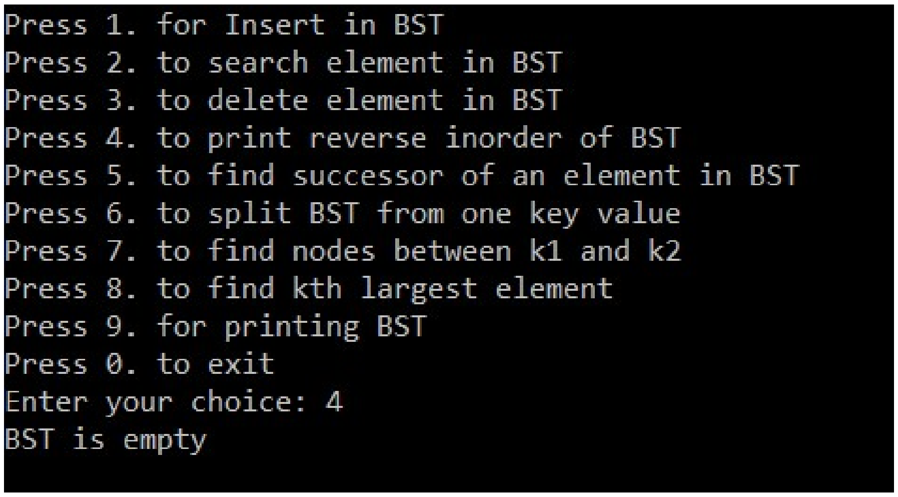
OUTPUT:

Test case 1: Non-empty tree



```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 4
Reverse Inorder of the given tree - 35 30 25 12 10 3
```

Test case 2: Empty tree

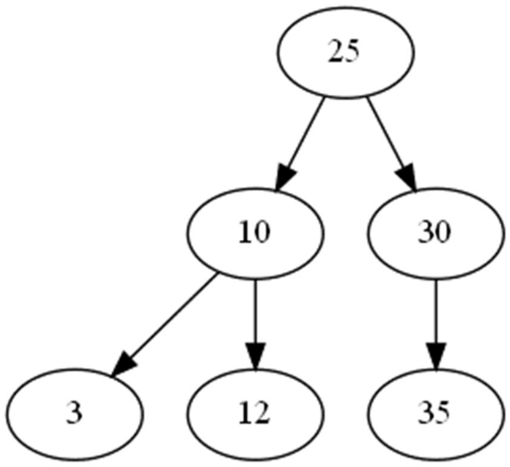


```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 4
BST is empty
```

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.

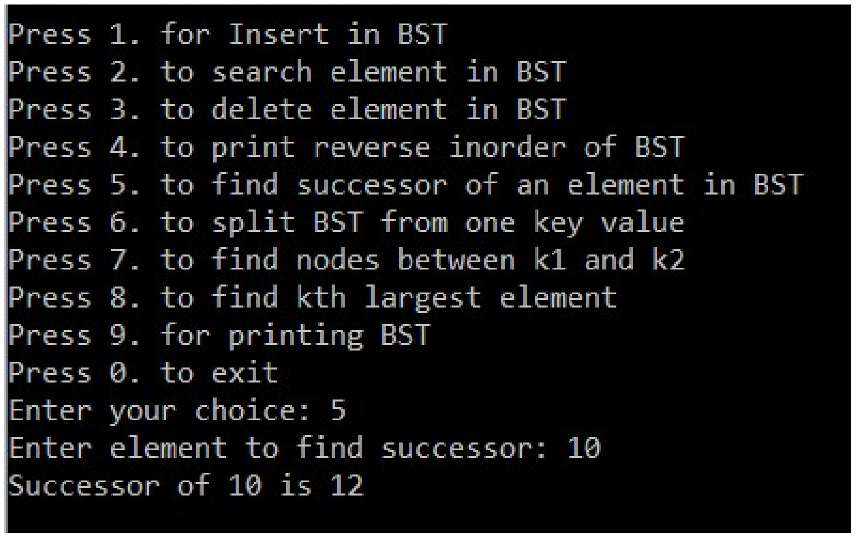
- there are **2 cases**
    3. The ***rightLinkType*** is **THREAD** , in that case it is thread than we directly go to that node and return value of that node
    4. The ***rightLinkType*** is **CHILD**, in that case we will traverse to the left most node of the right subtree and then return the value of that left most node of the right subtree

Input:



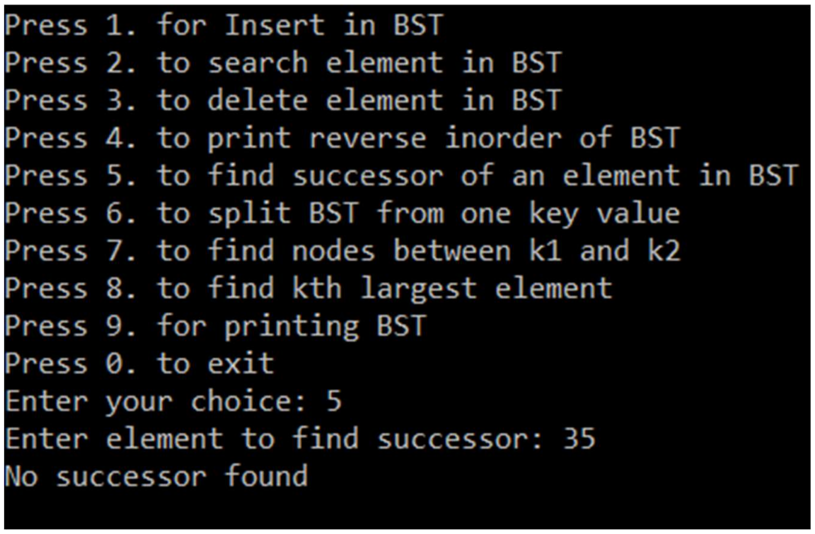
Output:

Test case 1: Node in between i.e. 10



```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 5
Enter element to find successor: 10
Successor of 10 is 12
```

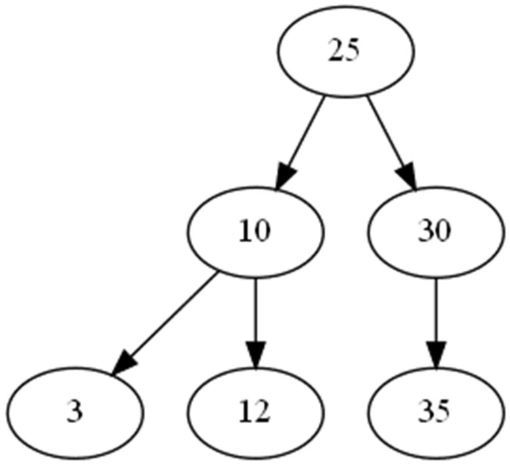
Test case 2: rightmost node, i.e. 35



```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 5
Enter element to find successor: 35
No successor found
```

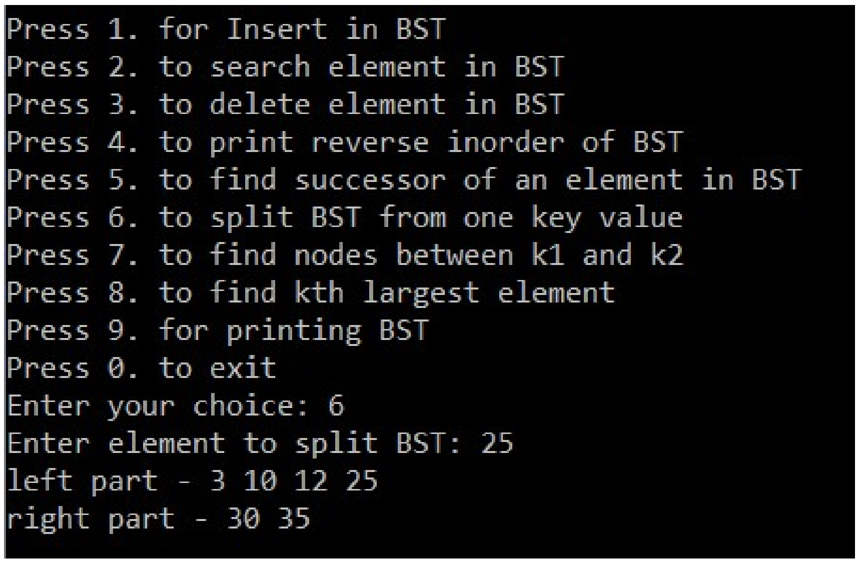
6. ***split(k)* --** split the BST into two BSTs T1 and T2, where keys(T1) <= k and keys(T2) > k. Note that, k may / may not be an element of the tree.
- First I am finding the inorder of the given tree.
- Traversing the inorder until we reach k and then using that linked list which has values from minimum node to k value creating the left tree. Likewise making the new tree for the values which are greater than k and less than the maximum node of the original tree.
- In the last I am printing the inorder of both trees.

Input tree:



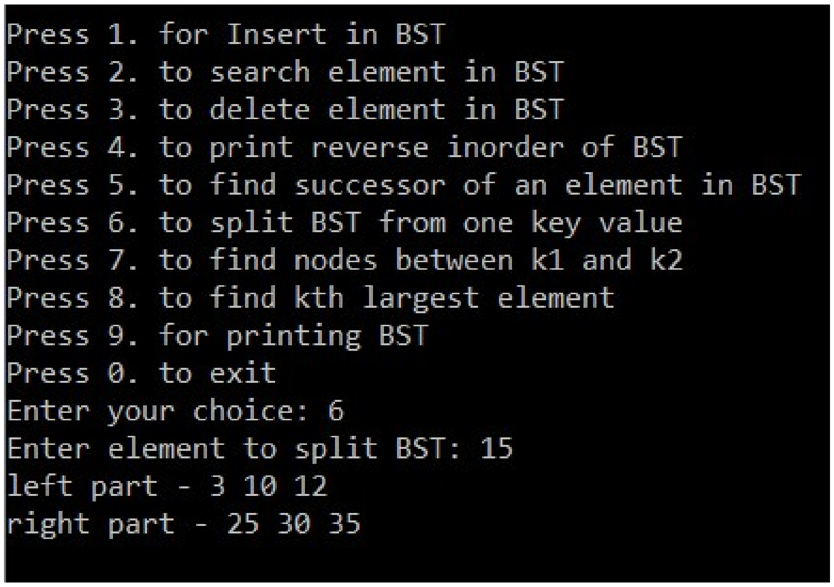
Output:

Test case 1: Entering the k value which is existing in some node of the tree, i.e. 25



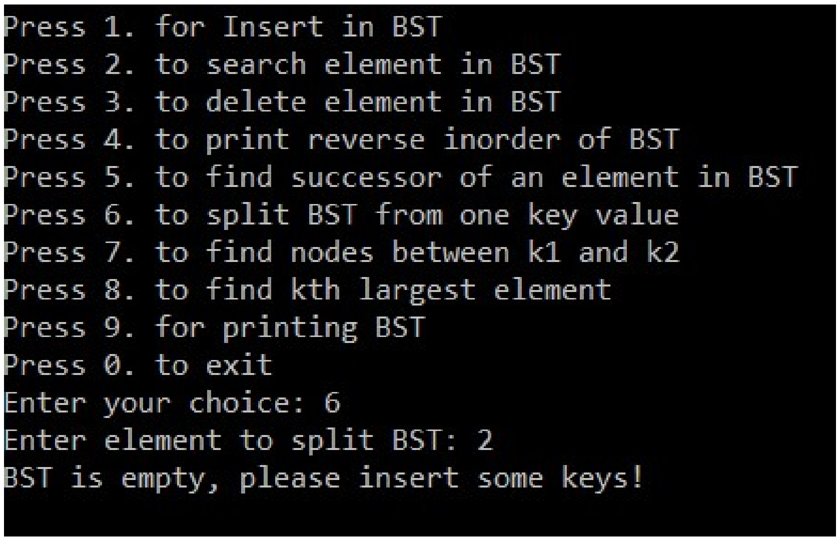
```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 6
Enter element to split BST: 25
left part - 3 10 12 25
right part - 30 35
```

Test case 2: Entering the k value which does not exist in the tree



```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 6
Enter element to split BST: 15
left part - 3 10 12
right part - 25 30 35
```

Test case 3: BST is empty

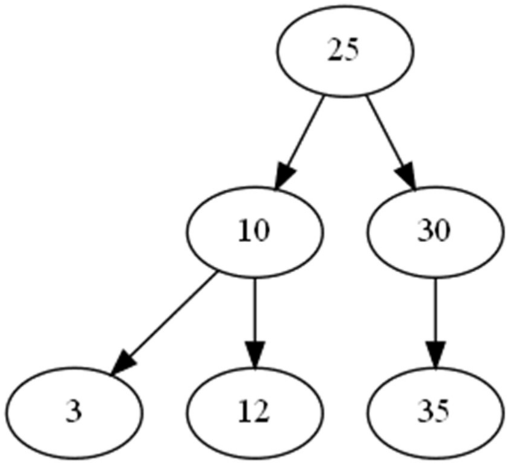


```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 6
Enter element to split BST: 2
BST is empty, please insert some keys!
```

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

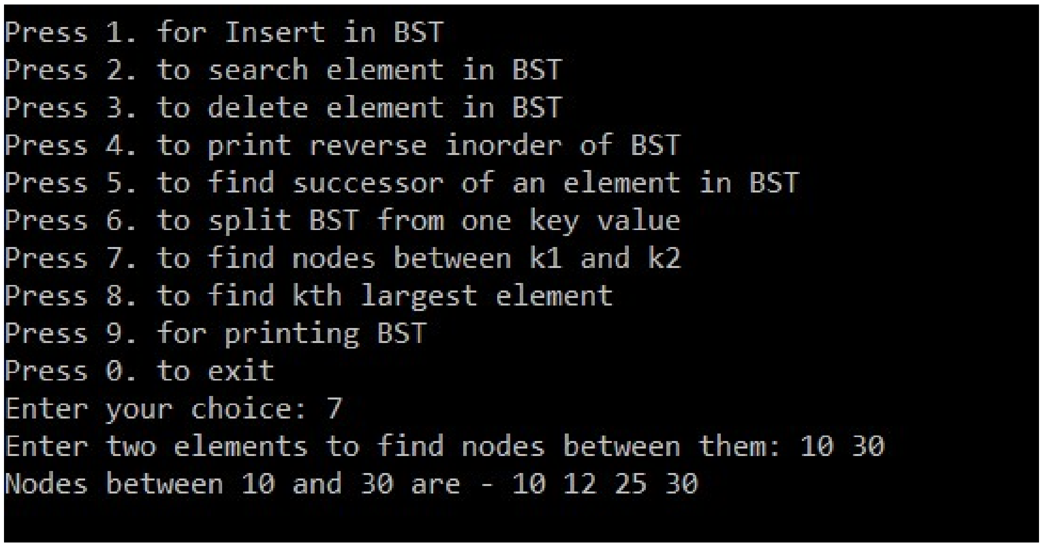
- First, I search for the value of k1 into tree. If the value exist we will get a pointer to that node. Else program will throw error that k1 value is not valid
- Now if the value exist then we will traverse the tree from k1 to k2 part using the ***successor()*** function and add all the nodes into singly linked list.
- In the last I am checking if the value is equal to k2 then only we return the linked list.

Input:



for this tree we input k1 = 10 and k2 = 30

Output:

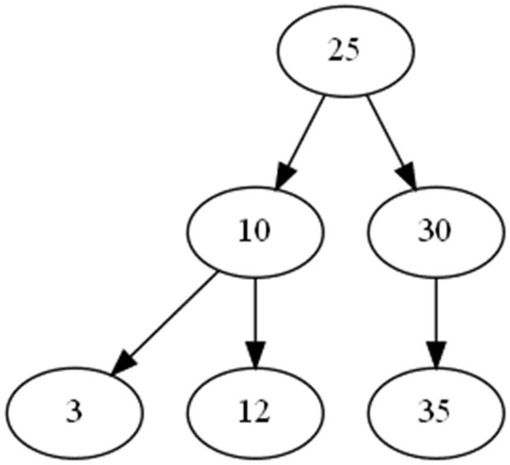


```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 7
Enter two elements to find nodes between them: 10 30
Nodes between 10 and 30 are - 10 12 25 30
```

8. ***kthElement(k)* --** finds the k-th largest element in the BST and prints the key value.

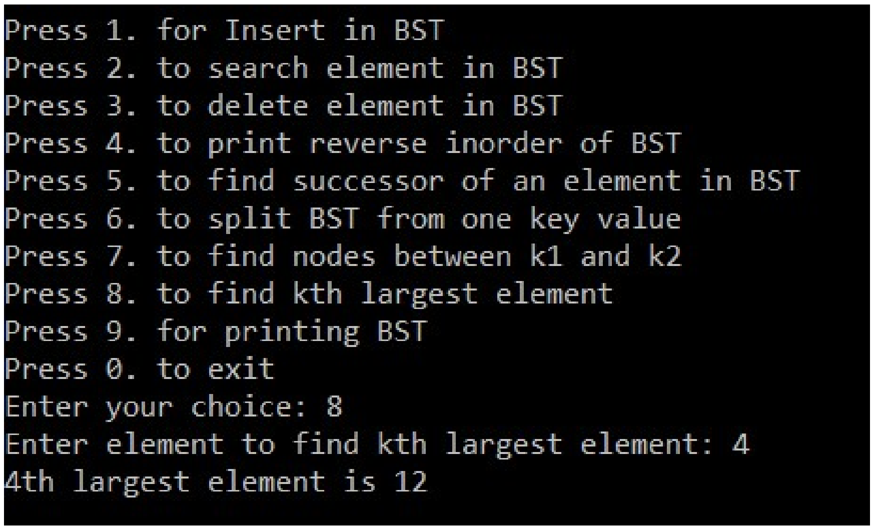
- We are using ***rCount*** variable specifically for finding the kth largest element.
- We call one utility function recursively. Utility function count the number of nodes in the right sub tree of that node. If it is greater than the k value then the largest element will be in the right subtree, else it will be in the left sub tree.
- Here we discard the whole subtree depending on the count and k value, means it will reach to kth largest element in O(h) time, h = height of the tree

Input:



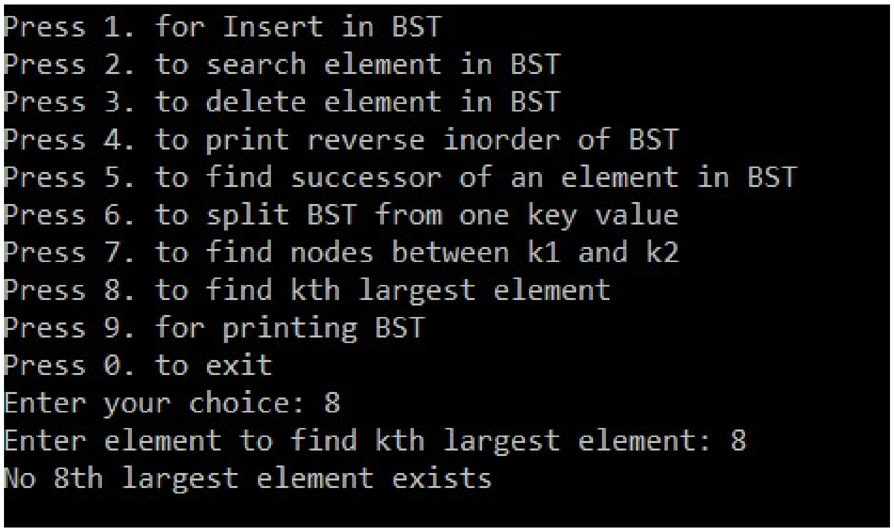
Output:

Test case 1: where the kth largest element exists in tree. i.e. k= 4



```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 8
Enter element to find kth largest element: 4
4th largest element is 12
```

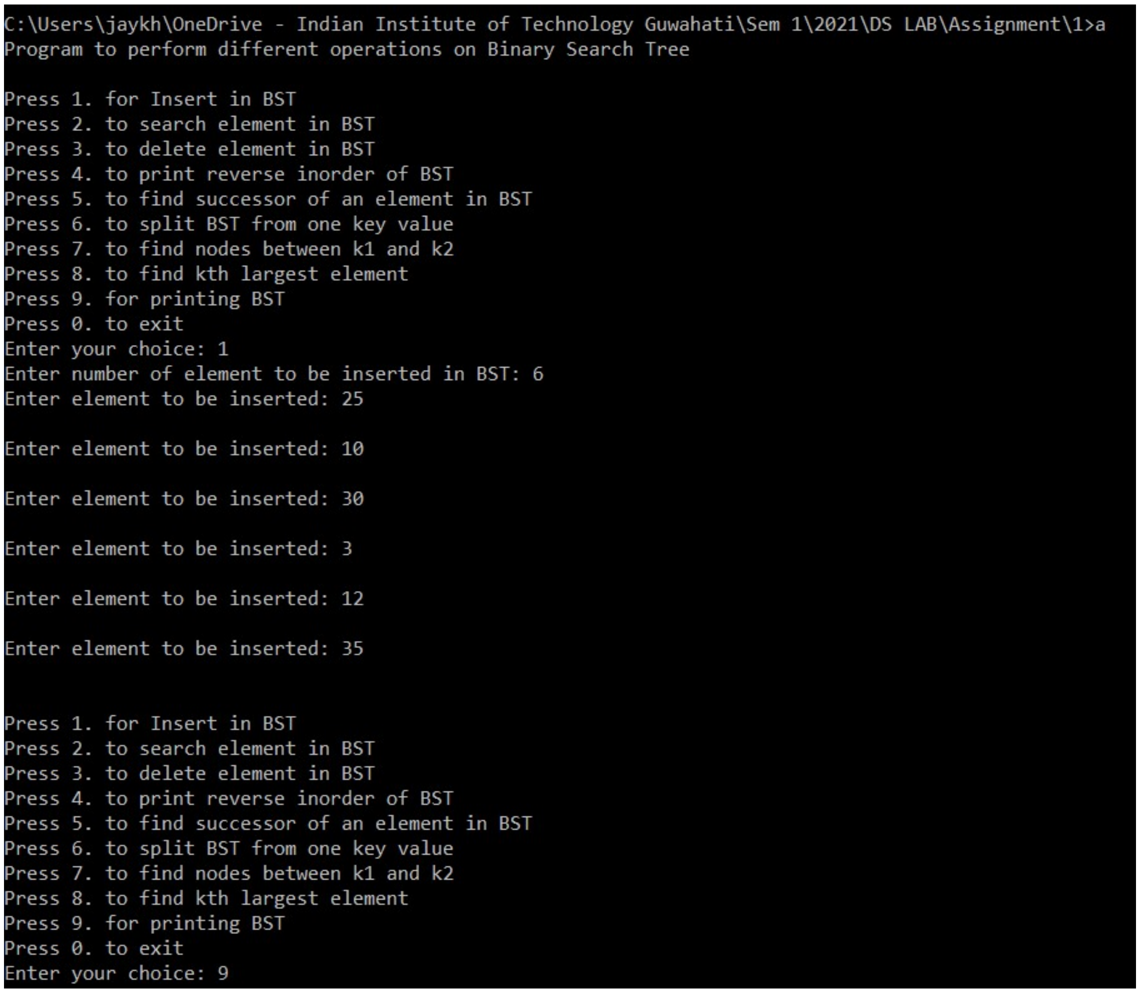
Test case 2: where the kth largest element does not exist in tree. i.e. k=8



```
Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 8
Enter element to find kth largest element: 8
No 8th largest element exists
```

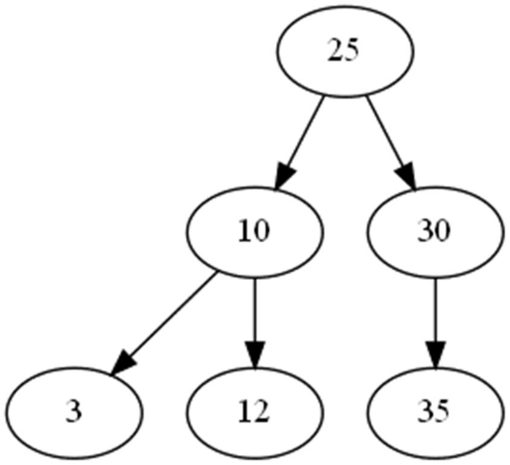
9. ***printTree()* --** Your program should print the BST
- we use the file handling and write in the ***graph.gv*** file.
- We call the ***printNode()*** function to write the left and right node and edges to the parent.
- ***printNode()*** function works recursively, it adds first the left node if it is not null then it will add the edge of itself and the parent into file ***graph.gv*** then we do same for the right child of the node.
- After completion we will have our source file graph.gv ready
- Now, we will call the ***system()*** call to execute the command "***dot -Tpng graph.gv -o graph.png"***
- It will create png file which contains our printed tree.

Input:



```
C:\Users\jaykh\OneDrive - Indian Institute of Technology Guwahati\Sem 1\2021\DS LAB\Assignment\1>a
Program to perform different operations on Binary Search Tree

Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 1
Enter number of element to be inserted in BST: 6
Enter element to be inserted: 25

Enter element to be inserted: 10

Enter element to be inserted: 30

Enter element to be inserted: 3

Enter element to be inserted: 12

Enter element to be inserted: 35


Press 1. for Insert in BST
Press 2. to search element in BST
Press 3. to delete element in BST
Press 4. to print reverse inorder of BST
Press 5. to find successor of an element in BST
Press 6. to split BST from one key value
Press 7. to find nodes between k1 and k2
Press 8. to find kth largest element
Press 9. for printing BST
Press 0. to exit
Enter your choice: 9
```

Output:


25
10
30
3
12
35