CS 513

DS Lab

Assignment 1

Abhijeet Padhy

Roll: 214101001

INDEX

Helper Classes/structures

LLNode

The purpose of this class is to make nodes of linked list.

Private Members:

1. int key; // stores the data of the node

Public Members:

1. LLNode *next; // stores the next pointer
2. LLNode(int val) // Constructor
3. int getKey() // function that returns key value (which is private) to outside world.

LinkedList

The purpose of this class is to create a linked list of LLNode objects.

Private Members:

1. LLNode *head; // stores the pointer of the head node of the linked list
2. LLNode *tail; // stores the pointer of the tail node of the linked list

Public Members:

1. LinkedList() // Constructor that initialises head and tail with NULL.
2. void addNode(int val) // adds a new node with value val to the linked list
3. void printList() // prints the data of all the nodes of the linked list

Splits

The purpose of this structure is to store pointers to two objects of ThreadedBST class. Using this we can return reference to two trees after we call the split function.

Members:

1. ThreadedBST *left=NULL; // reference of first tree
2. ThreadedBST *right=NULL; // reference of second tree

Tree Classes

ThreadedBST

This class stores a tree node and has methods that operate on a tree node.

Private Members:

1. ThreadedBST *leftChild; // Stores pointer of left child. It can be NULL, or it can also point as a thread to its predecessor if there is no left child.
2. bool leftThread; // It is true when leftChild is NULL or is pointing as a thread to its predecessor. It is false when leftChild is pointing to its left child node
3. int key; // Data value of a tree node
4. bool rightThread; // It is true when rightChild is NULL or is pointing as a thread to its successor. It is false when rightChild is pointing to its right child node.
5. ThreadedBST *rightChild; // Stores pointer of right child. It can be NULL, or it can also point as a thread to its successor if there is no right child.
6. int rcount; // Stores size of right subtree
7. ThreadedBST *leftMost(ThreadedBST *root); //Returns the left most tree node
8. ThreadedBST *rightMost(ThreadedBST *root); //Returns the right most tree node
9. struct Splits *splitUtil(int k);// Utility function to split a given tree
10. int printTreeUtil(FILE*); // Utility helper function for printTree
11. void printTree(char *); // Creates a pictorial image of the tree but the gv file name has to be specified in argument
12. ThreadedBST *allElementsBetweenUtil(int , ThreadedBST *); // Utility Helper function for allElementsBetween

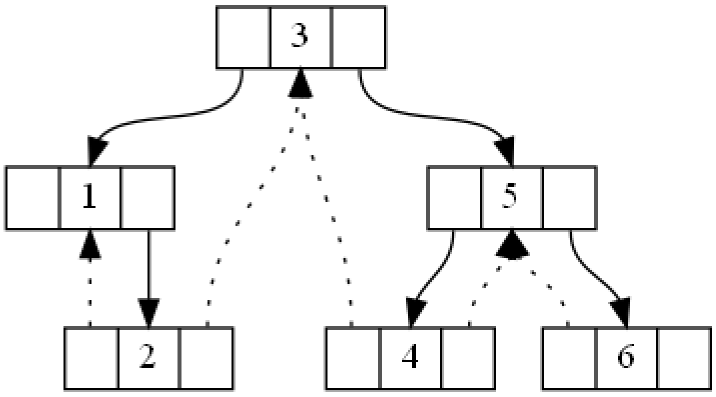
Public Members:

1. ThreadedBST(int val) // Constructor to create a new tree node having key value val
2. ThreadedBST(ThreadedBST *); // Copy constructor that creates a new Threaded BST using an existing one which is sent as an argument
3. void insert(int val); //Inserts a new tree node with value val
4. void inorder();//Prints the inorder traversal of the nodes of the tree
5. ThreadedBST *search(int val); //Searches the tree for a node with value val
6. int getKey();//Returns the key value of the root node
7. ThreadedBST *deleteElement(int val); // Delete a node of the tree with value equal to val
8. ThreadedBST *successor(ThreadedBST *); // Returns pointer to the successor node of the node passed in argument
9. LinkedList * reverseInorder();// Returns a linked list of elements which contains the reverse inorder traversal of the tree nodes
10. void printTree(); // Creates a pictorial image of the tree
11. LinkedList * allElementsBetween(int,int); // Creates a linked list of elements with values between a given range k1<=k<=k2
12. ThreadedBST * kthElement(int k); // Returns pointer to the kth largest element
13. struct Splits *split(int k); // Creates a new tree to split it into two halves and then return the two new trees
14. void clone(ThreadedBST *, LListOfTNode *list); // Function to create clone of a tree

Explanation of functions of ThreadedBST

Structure of a tree node:

The tree node contains a key value. It has a left child and a right child pointer. The left/right child pointer will point to left/right child if they exist. Otherwise they will point to predecessor/successor respectively. If left child pointer does not point to a left child, then it will have leftThread set as false. The left child pointer in this case will point to its predecessor. If there is no predecessor, it will point to NULL. Same is the case for a right child pointer, with the difference that it will point to its successor if it exists. Each node also has an rcount variable which stores the size of the right sub tree. For a newly created node, its value will be 0, as there is no right sub tree yet.



Note that node 1 does not have a predecessor, so left child pointer does not point to any node, it has NULL stored in it.

Insert function:

It has the prototype: **void insert(int val)** .

It works by comparing val with the root nodes value(which is called key). If val is equal to key, it means the node already exists and hence an exception is thrown. Hence when we call insert function, it has to be present inside try-catch block to run correctly.

If it is less than root pointer’s value (called key) and left child points to left subtree, it recursively executes for left subtree. If it does not point to a left subtree, then this is the location where it should be inseeted. So a new node is created and inserted as the left child of the root node. The predecessor of the newly inserted node is set appropriately in this step.

Similar steps are followed if val is greater than key but for the right node and right sub tree. There is one additional step here. We increment the rcount variable as well as the size of the right subtree will increase by one.

Search function:

It has the prototype: **ThreadedBST * search(int val)**

If val is equal to key, return root node. If val is less than key, recursively execute for left sub tree. If val is greater than key, recursively execute for right sub tree. If we reach a leaf node and still don't find the value, then return NULL.

Delete function:

It has the prototype: **ThreadedBST* deleteElement(int val)**

If val is less than key and left child does not point to left sub tree, then throw exception. If it points to a left sub tree, then recursively call for the left sub tree and also set the thread pointers appropriately. When we call for the left subtree, we get returned, a tree from where the node has been deleted. So we re-assign left child pointer to point to this new tree.

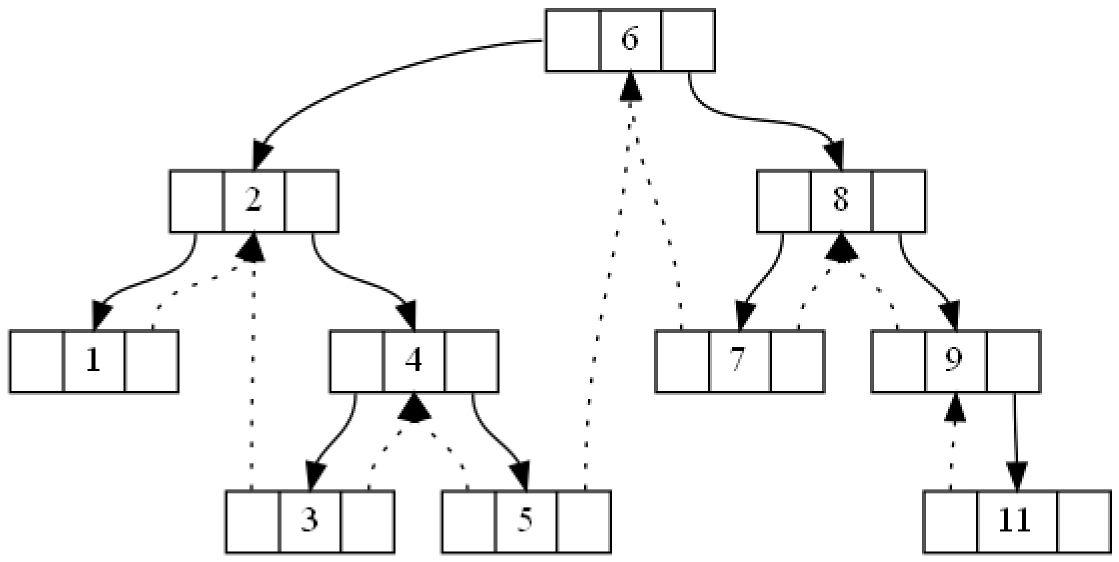
Similar things happen for right sub tree when val is greater than key.

If val is equal to key, three cases arise:

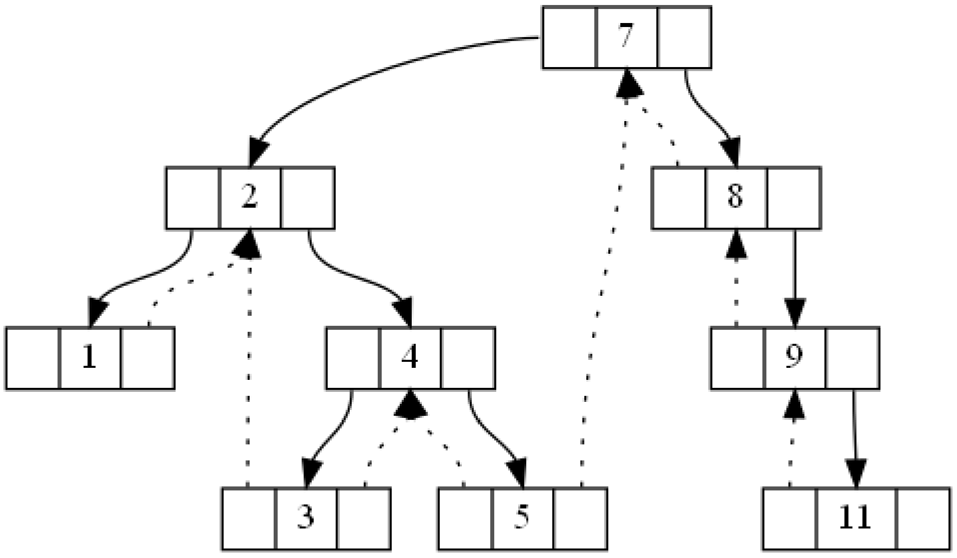
Case1: If it is a leaf node, simply delete and return NULL.

Case2: If it has only one child, delete the root and return the child.

Case3: If it has two children, we find the successor of the right child. We copy the value of the successor to the root node's key value. We then run delete operation on the right subtree for the value stored in successor. In other words, the root node is replaced by the successor and then the successor gets deleted. We then decrement the rcount value of the root node by one. When we delete the successor, case 1 or case 2 applies as it can either have a right sub tree or none at all. Note that other than the root's rcount, there is no need to modify the rcount of any other node in this step. It will be clear from the below diagram:



On deleting element 6, we get the following result:



At the end we return the new root. Any successive operation should be performed on the new root.

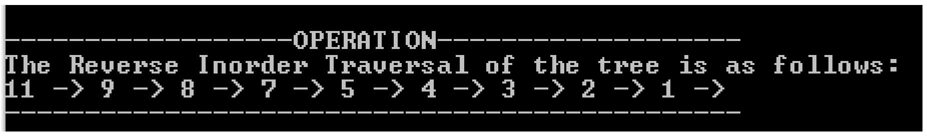
Reverse Inorder function:

It has the prototype: **LinkedList * reverseInorder();**

We first find out the right most node of the tree. We run an iteration from that node. If the left child points to a predecessor, we proceed to the predecessor. If the left child points to a left sub tree, then we go to the rightmost node of the left sub tree. We traverse in this way until next node comes to be NULL. We add all the nodes that we visit, to a linked list which we return in the end.

This approach does not make use of stack/recursion. It uses threads in the tree to find the predessor of the current node. Thus we are able to do it in an iterative way.

The reverse inorder traversal performed on the previous tree is:



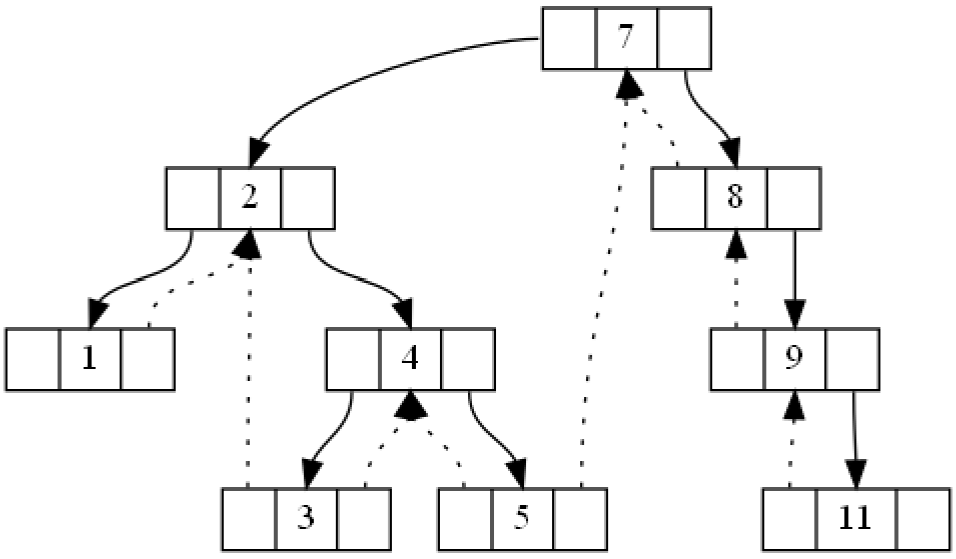
```
------------------OPERATION-------------------
The Reverse Inorder Traversal of the tree is as follows:
11 -> 9 -> 8 -> 7 -> 5 -> 4 -> 3 -> 2 -> 1 ->
----------------------------------------------
```

Successor function:

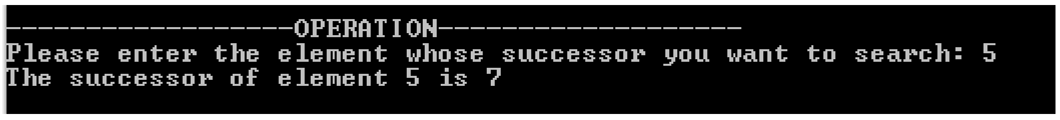
It has the prototype: **ThreadedBST * successor(ThreadedBST *root)**

It returns the successor of a node. If the right thread is true, that means right child pointer points to either NULL or its successor, we directly return the right child pointer. If the right child points to a sub tree, then we call leftmost function on the right right child. This gives the successor in this case.

For example, consider this case:

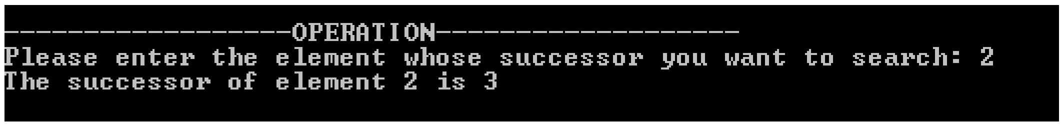


The successor of element 5 can be easily found from the right thread:



```
------------------OPERATION-------------------
Please enter the element whose successor you want to search: 5
The successor of element 5 is 7
```

But for element 2, we have to find leftmost element of the right sub tree which is 3.



```
------------------OPERATION-------------------
Please enter the element whose successor you want to search: 2
The successor of element 2 is 3
```

Split function:

It has the prototype: **struct Splits *ThreadedBST::splitUtil(int k)**

The purpose of this function is to split the tree into two halves such that all elements in the first tree have values less than or equal to k and the second tree has all the remaining elements. We return both the trees at the end.

Before we run this function, we create a clone of the tree using copy constructor. We execute the split function on the cloned tree so that original tree does not get affected.

If k is smaller than the smallest node, then first tree has NULL and second tree has the entire tree. If k is greater than the largest node, then first tree has the entire tree and second tree has NULL.

We first create an object of Splits class which stores pointer to left and right tree. Initially they contain NULL values. After the function runs, we will have the left pointer point to the left tree and the right pointer point to the right tree.

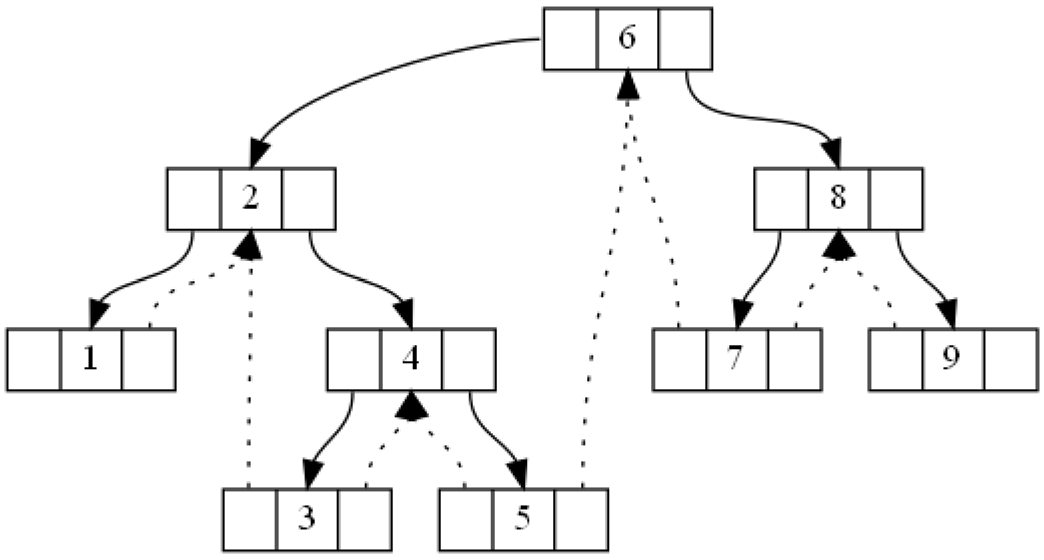
If k is equal to key, then it must be included in the left tree. The right tree now points to the right child of the current node. The left tree points to the current node. We return the left and right trees.

If k is less than key, we run recursively for left tree. What we get are two trees left and right which are already divided somewhere in the recursive step. Now we have to include the current node in the right tree because k<key and hence the current node belongs to the right tree.

If k is greater than key, we run recursively for right tree. What we get are two trees left and right which are already divided somewhere in the recursive step. Now we have to include the current node in the left tree because key<k and hence the current node belongs to the right tree.

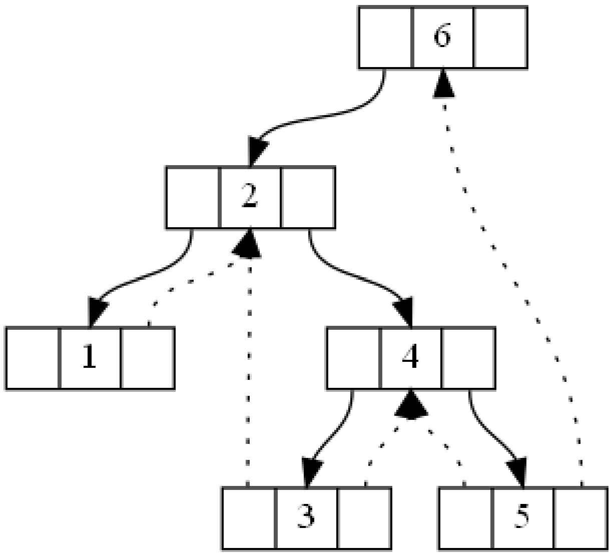
Thus at the split begins at the required node and two trees are created. Then we backtrack and keep on adding the ancestors to either the left tree or right tree based on where they belong.

Time Complexity Analysis: Since we traverse the ancestors of the required node, the time taken is in order of height of the tree(O(h)).

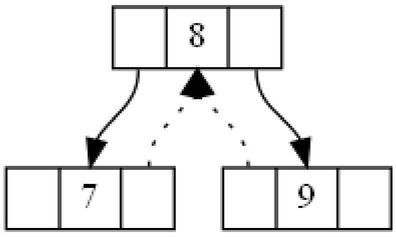


Let the value of k be 6

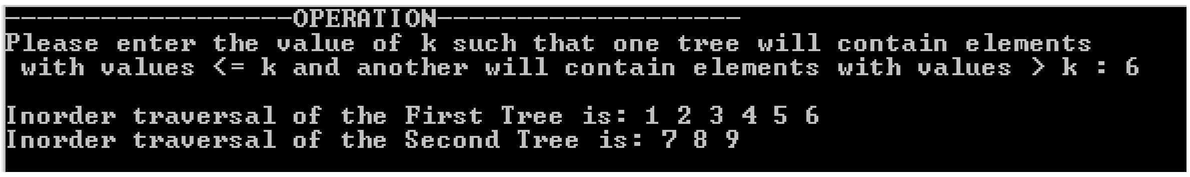
First Tree:



Second Tree:



The INORDER traversal of the first and second trees are printed on the screen. Also printTree function is run on left and right trees, so that we can verify the output in graphical way.



```
------------------OPERATION-------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 6

Inorder traversal of the First Tree is: 1 2 3 4 5 6
Inorder traversal of the Second Tree is: 7 8 9
```

allElementsBetween function:

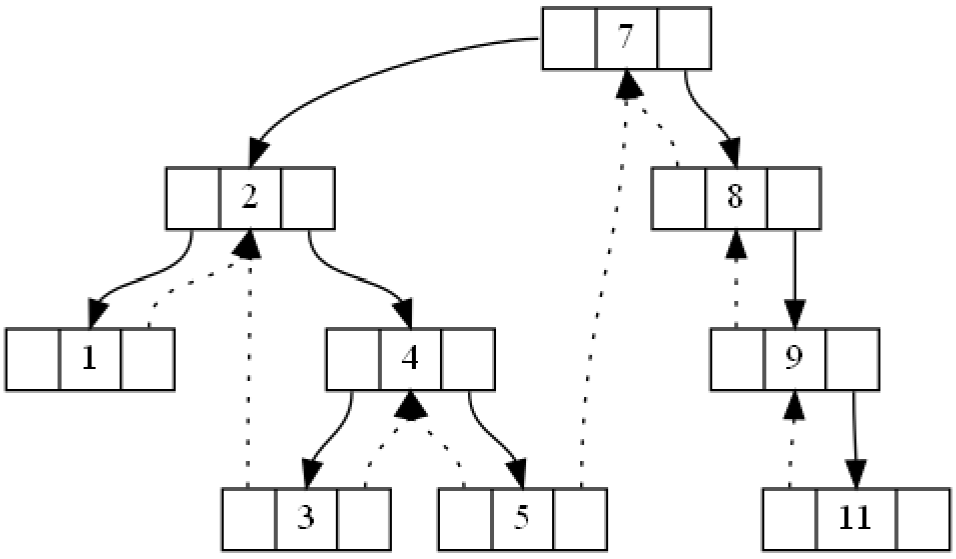
It has the prototype:

LinkedList * allElementsBetween(int k1, int k2) and

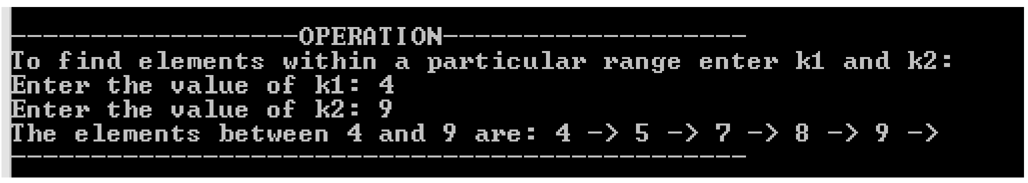
ThreadedBST * allElementsBetweenUtil(int k, ThreadedBST *root)

This function returns a singly linked list of all those elements in the tree which lie between values k1 and k2(inclusive). k1<=k<=k2 . First we run allElementsBetween function which calls allElementsBetweenUtil function. The purpose of the second function is to find that node whose key value is greater than or equal to k1. Then a linked list is created. We then traverse the tree from the node returned by the second fuction by finding succesors and keep on adding the key values to the linked list. We stop when we find the fisrt node whose value is greater than k2 or when we reach NULL.

Time Complexity Analysis: allElementsBetweenUtil takes O(h) time to find out the first element greater than or equal to k1. Then we traverse through N elements which takes O(N) time. Thus total complexity is O(h+N).



This is the list of elements between 4 and 9 inclusive. Following is the result.



```
------------------OPERATION-------------------
To find elements within a particular range enter k1 and k2:
Enter the value of k1: 4
Enter the value of k2: 9
The elements between 4 and 9 are: 4 -> 5 -> 7 -> 8 -> 9 ->
----------------------------------------------
```

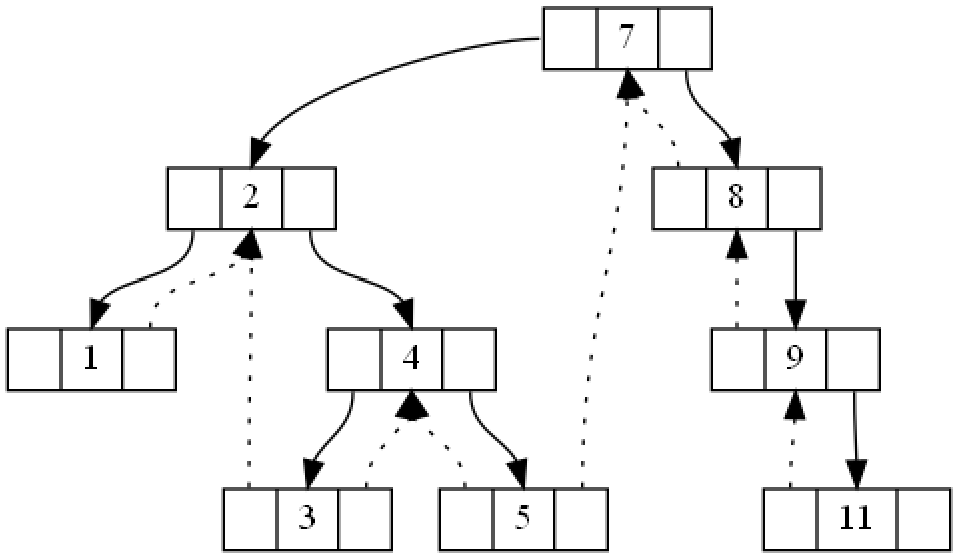
kthElement function:

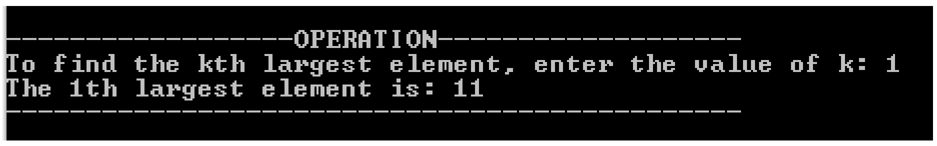
It has the prototype: **ThreadedBST * kthElement(int k);**

It finds the kth largest element in the BST and prints the key value.

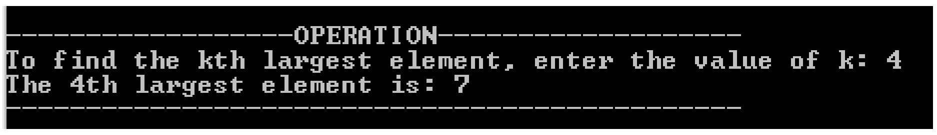
Every node has a variable called rcount which stores the size of the right sub tree. All the elements of right sub tree are greater than root node's key value. Thus the position of the root node if all the elements are sorted in descending order is rcount+1.

If k is equal to one more than rcount value of the current nood, this means current node is the kth largest element, so return it. If k is less than rcount+1, recursively run for right sub tree with the same argument. If k is greater than rcount+1, recursively run on left sub tree with argument k-rcount-1, because we have already crossed rcount+1 elements which included the root and its right sub tree.

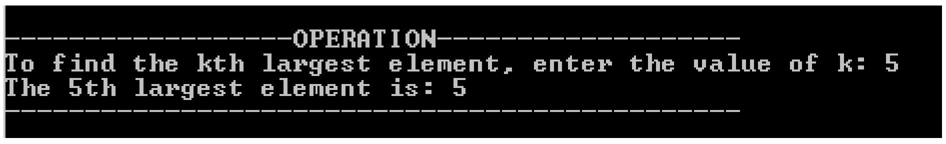




```
-------------------OPERATION--------------------
To find the kth largest element, enter the value of k: 1
The 1th largest element is: 11
------------------------------------------------
```



```
-------------------OPERATION--------------------
To find the kth largest element, enter the value of k: 4
The 4th largest element is: 7
------------------------------------------------
```



```
-------------------OPERATION--------------------
To find the kth largest element, enter the value of k: 5
The 5th largest element is: 5
------------------------------------------------
```

printTree function:

It has the prototype:

void ThreadedBST::printTree()

void ThreadedBST::printTree(char *str)

int ThreadedBST::printTreeUtil(FILE *fptr)

The purpose of this function is to generate .gv file which will be parsed by graphwiz utility to produce an image of the tree. We traverse the tree in a preorder manner. When we arrive at a node in our traversal, we add node in the .gv file. Then we recur for left and right subtrees. When we return from recursive call from a sub tree, we add an edge from the root node to the left or right sub trees' root node. At the end we have the complete .gv file. From outside of the program, run the following command to convert a .gv file to .png file which is our required image.

dot -Tpng graph.gv -o graph.png

Copy Constructor function:

It has the prototype: **ThreadedBST (ThreadedBST * root)**

This creates a new tree which is an exact clone of an existing tree. We pass the tree to be cloned in argument. Copy constructor creates a new tree object, but the child nodes are not present yet. To clone all the other nodes, we call clone method. When the clone method is called, first the current node's values are set. Then if there is to be a left sub tree, then left recursion is made. Otherwise a thread has to be created to its predecessor. But when we create this kind of thread, we need to point to a node in the new tree and not the old one. So we caanot simply copy the thread pointer from the old tree. This is the reason why we create a linked list to store all the tree node references for those tree nodes, which are already created. Thus we search for the required node which is the target of a nodes thread from the list and use reference from there. We follow a preorder traversal in this function.

The TreeAPI Class

The purpose of this class is to hide the complexity involved in a ThreadedBST class. If a user does not follow a particular way of calling the methods of an object of ThreadedBST class, there can be problems. So I have provided an API class which a developer can use to make a threaded BST without worrying about the complexity involed in ThreadedBST. Each public method of ThreadedBST has a mapping in TreeAPI class. A developer just has to create an object of TreeAPI class and has to access the methods.

Private Members:

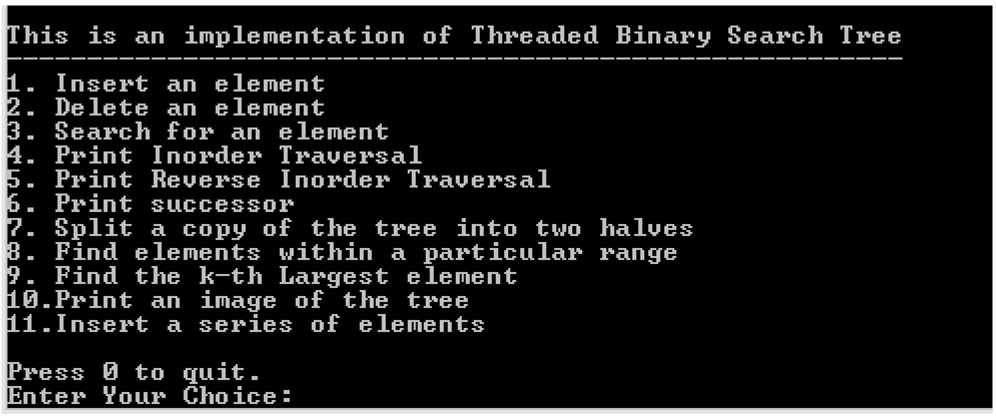
- ThreadedBST *root; //stores the root node of our BST

Public Members:

1. Tree()
2. void insert(int val)
3. ThreadedBST *search(int val)
4. void inorder()
5. LinkedList * reverseInorder()
6. struct Splits *split(int k)
7. LinkedList * allElementsBetween(int k1, int k2)
8. ThreadedBST * kthElement(int k)
9. void printTree()

Test Cases

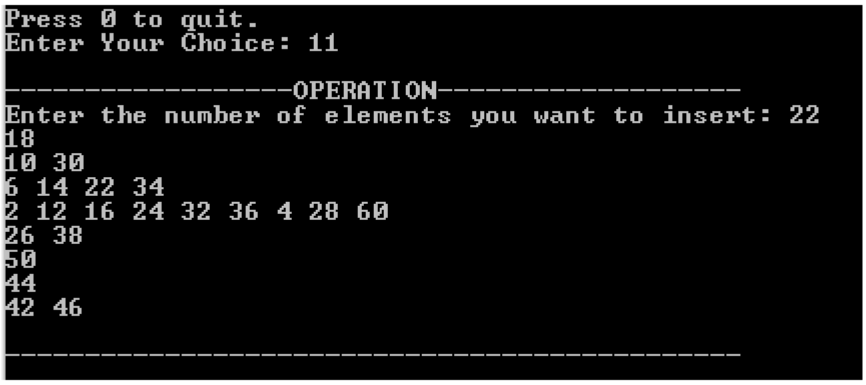
On executing the program we are shown such a kind of menu:



```
This is an implementation of Threaded Binary Search Tree
--------------------------------------------------------
1. Insert an element
2. Delete an element
3. Search for an element
4. Print Inorder Traversal
5. Print Reverse Inorder Traversal
6. Print successor
7. Split a copy of the tree into two halves
8. Find elements within a particular range
9. Find the k-th Largest element
10.Print an image of the tree
11.Insert a series of elements

Press 0 to quit.
Enter Your Choice:
```

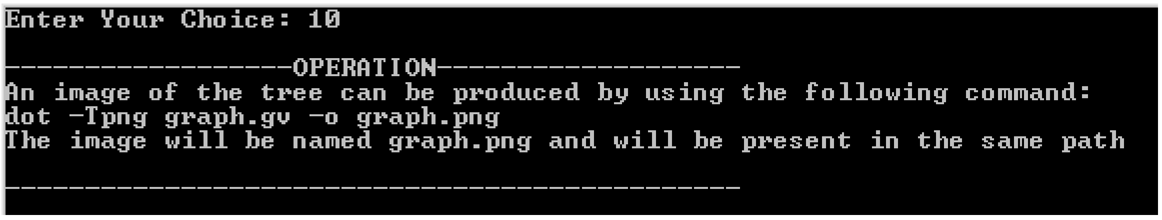
Let us now check a variety of test cases to handle our tree.

We insert a number of elements. For this we can choose option 11:



```
Press 0 to quit.
Enter Your Choice: 11

------------------OPERATION-------------------
Enter the number of elements you want to insert: 22
18
10 30
6 14 22 34
2 12 16 24 32 36 4 28 60
26 38
50
44
42 46

----------------------------------------------
```

We choose option 10 next to print our tree:

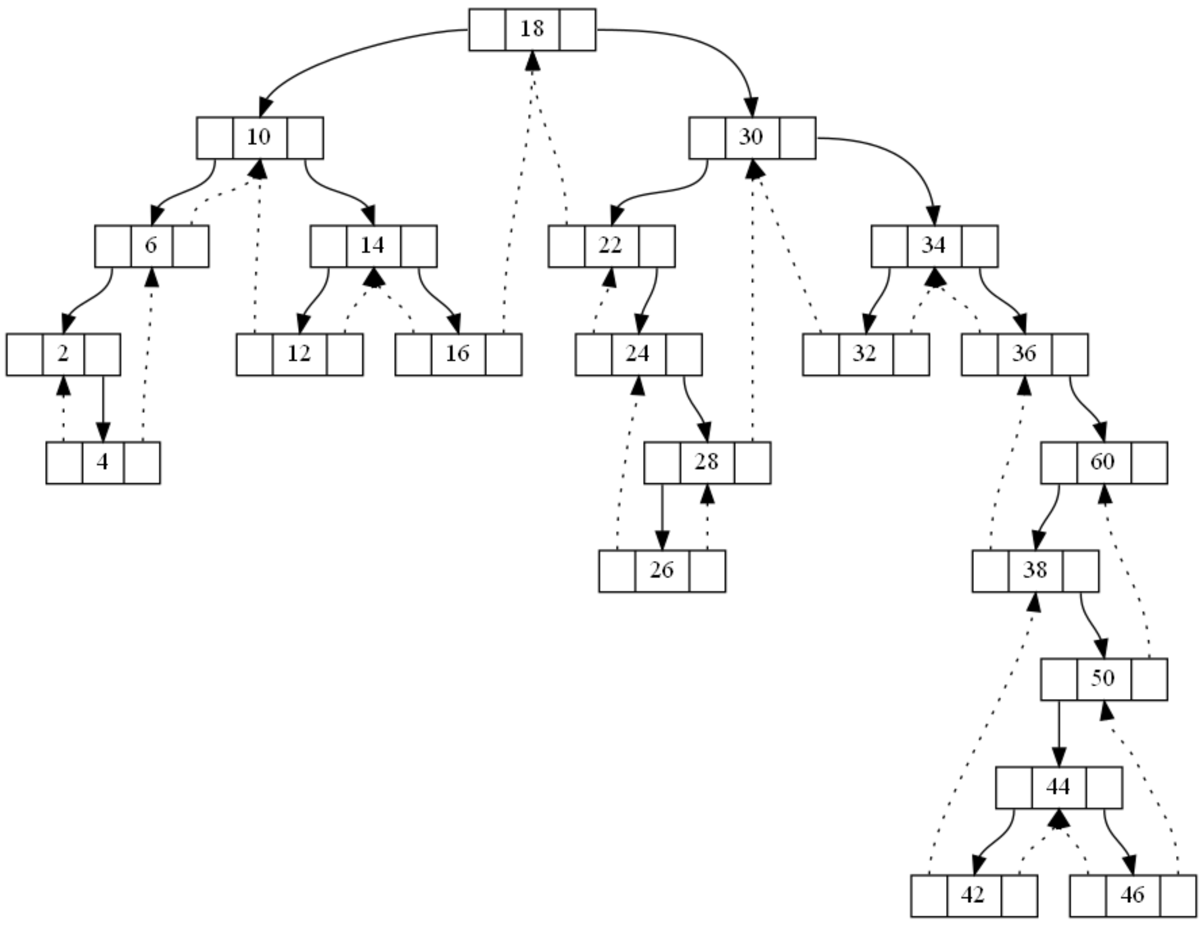


```
Enter Your Choice: 10

------------------OPERATION-------------------
An image of the tree can be produced by using the following command:
dot -Tpng graph.gv -o graph.png
The image will be named graph.png and will be present in the same path

----------------------------------------------
```

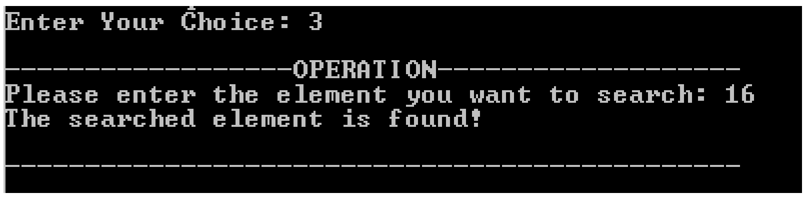
Execute the following command :

dot -Tpng graph.gv -o graph.png

And we get the below png file:

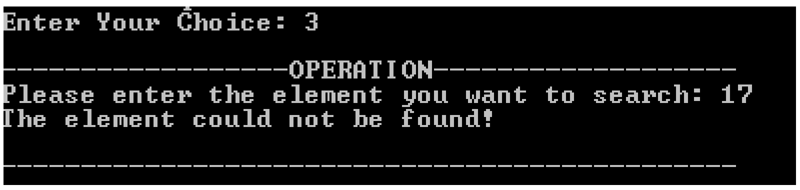


When we try to search for element 16:



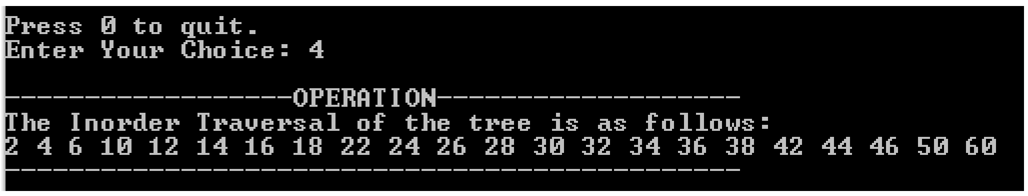
```
Enter Your Choice: 3

-------------------OPERATION--------------------
Please enter the element you want to search: 16
The searched element is found!

------------------------------------------------
```

When we try to search for element 17:



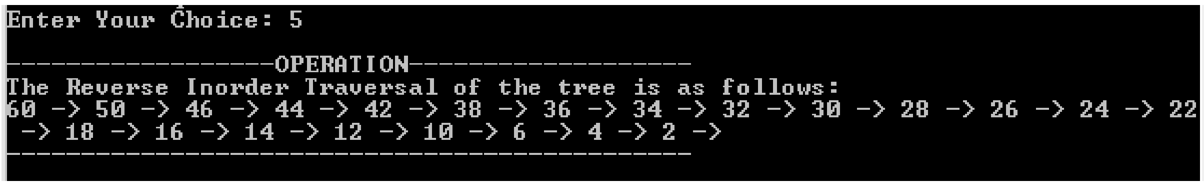
```
Enter Your Choice: 3

-------------------OPERATION--------------------
Please enter the element you want to search: 17
The element could not be found!

------------------------------------------------
```

When we print inorder traversal of the tree:



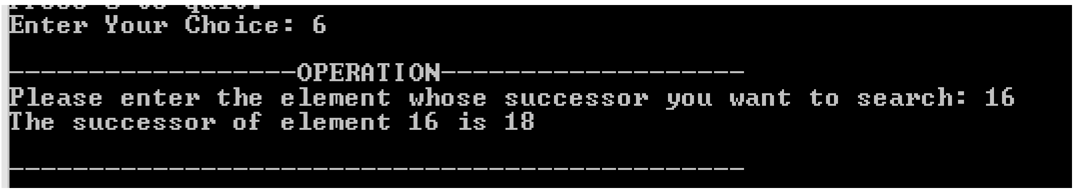
```
Press 0 to quit.
Enter Your Choice: 4

-------------------OPERATION--------------------
The Inorder Traversal of the tree is as follows:
2 4 6 10 12 14 16 18 22 24 26 28 30 32 34 36 38 42 44 46 50 60
------------------------------------------------
```

When we print the reverse inorder tree traversal:



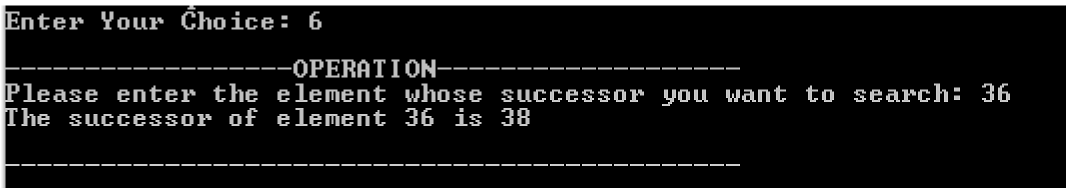
```
Enter Your Choice: 5

-------------------OPERATION--------------------
The Reverse Inorder Traversal of the tree is as follows:
60 -> 50 -> 46 -> 44 -> 42 -> 38 -> 36 -> 34 -> 32 -> 30 -> 28 -> 26 -> 24 -> 22
 -> 18 -> 16 -> 14 -> 12 -> 10 -> 6 -> 4 -> 2 ->
-------------------------------------------------
```

Successor of 16 is found to be 18



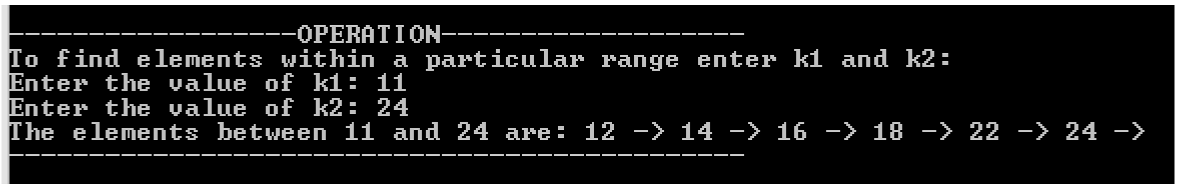
```
Enter Your Choice: 6

-------------------OPERATION--------------------
Please enter the element whose successor you want to search: 16
The successor of element 16 is 18

-------------------------------------------------
```

Successor of 36 is found to be 38



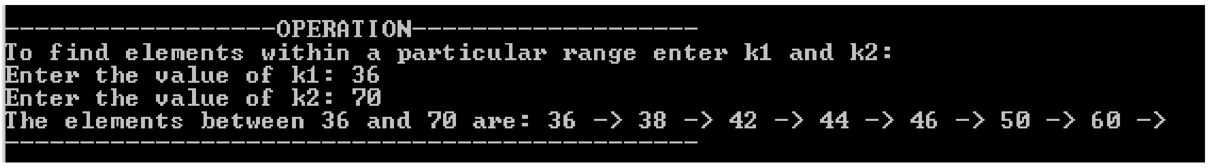
```
Enter Your Choice: 6

-------------------OPERATION--------------------
Please enter the element whose successor you want to search: 36
The successor of element 36 is 38

-------------------------------------------------
```

Range of elements between 11 and 24 are:



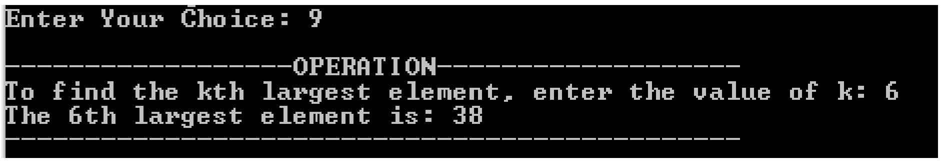
```
-------------------OPERATION--------------------
To find elements within a particular range enter k1 and k2:
Enter the value of k1: 11
Enter the value of k2: 24
The elements between 11 and 24 are: 12 -> 14 -> 16 -> 18 -> 22 -> 24 ->
-------------------------------------------------
```

Range of elements between 36 and 70 are:



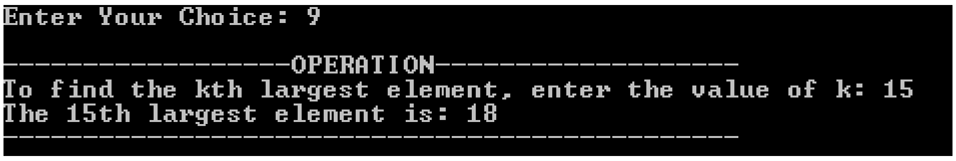
```
-------------------OPERATION--------------------
To find elements within a particular range enter k1 and k2:
Enter the value of k1: 36
Enter the value of k2: 70
The elements between 36 and 70 are: 36 -> 38 -> 42 -> 44 -> 46 -> 50 -> 60 ->
-------------------------------------------------
```

6th Largest element is 38



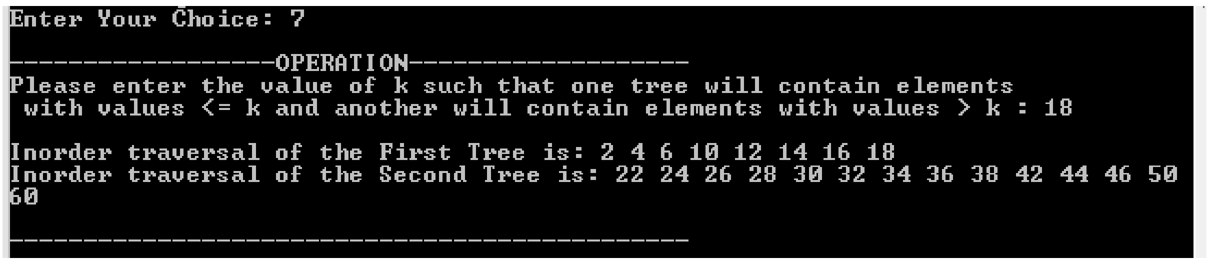
```
Enter Your Choice: 9

-------------------OPERATION--------------------
To find the kth largest element, enter the value of k: 6
The 6th largest element is: 38
-------------------------------------------------
```

15th Largest element is 18



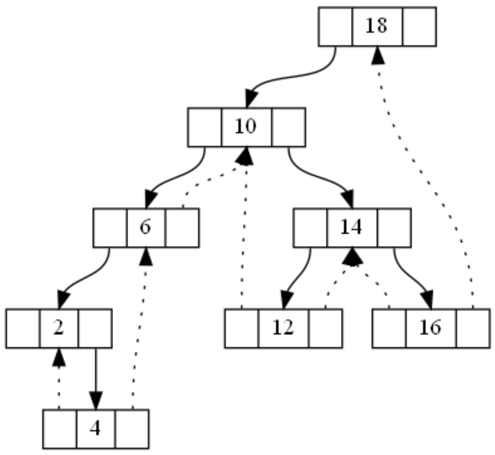
```
Enter Your Choice: 9

-------------------OPERATION--------------------
To find the kth largest element, enter the value of k: 15
The 15th largest element is: 18
-------------------------------------------------
```

Let us now proceed to split the tree. Split around 18:

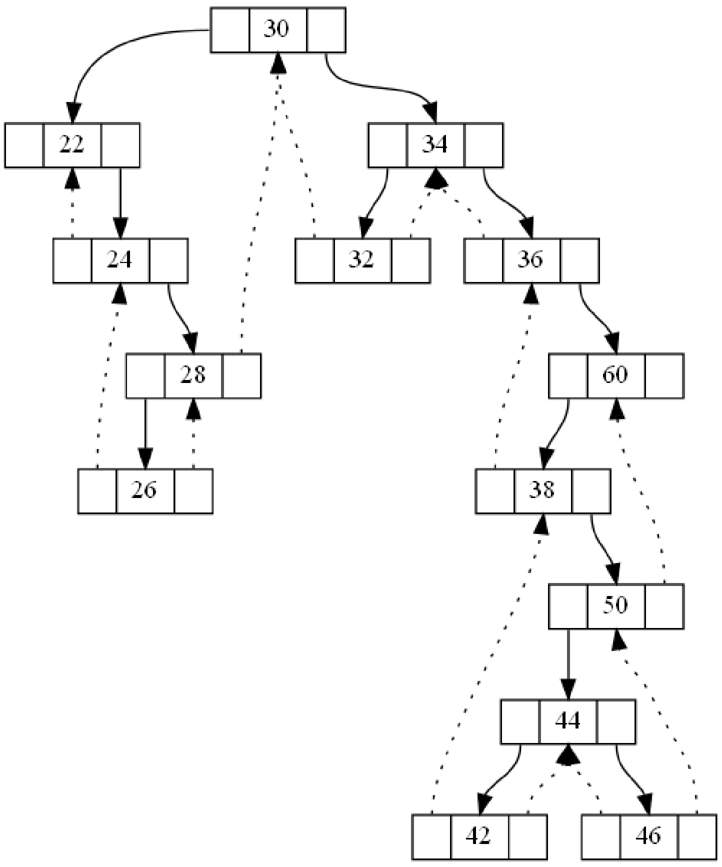


```
Enter Your Choice: 7

-------------------OPERATION--------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 18

Inorder traversal of the First Tree is: 2 4 6 10 12 14 16 18
Inorder traversal of the Second Tree is: 22 24 26 28 30 32 34 36 38 42 44 46 50
60

------------------------------------------------
```

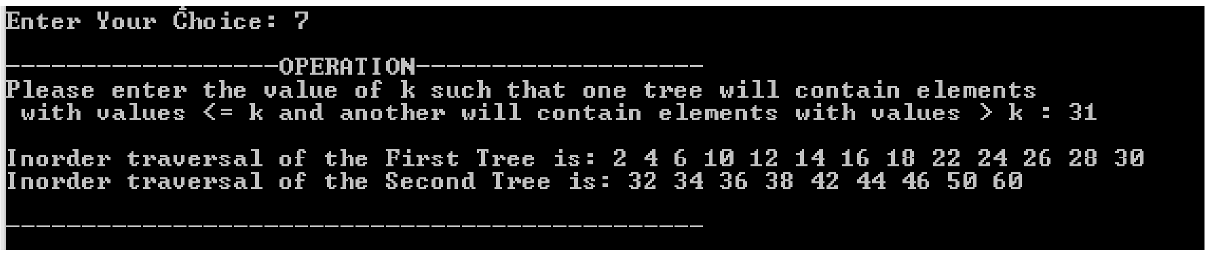
Left Tree:



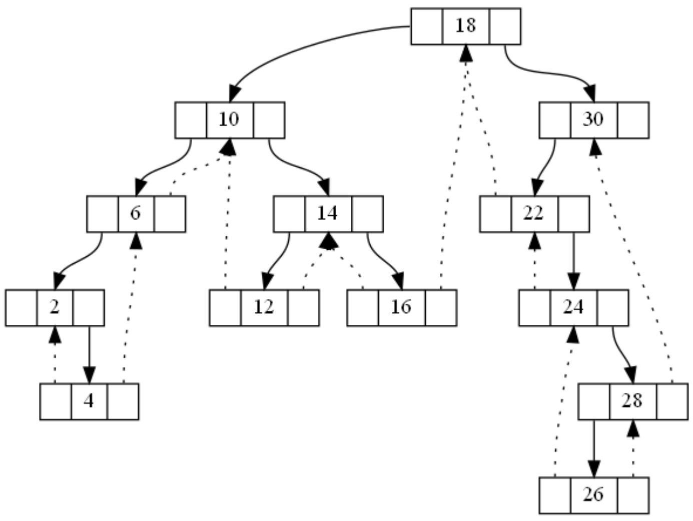
Right Tree:



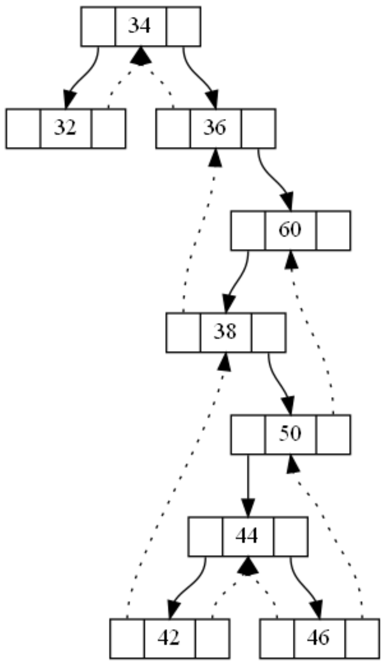
Split around 31:


```
Enter Your Choice: 7

------------------OPERATION-------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 31

Inorder traversal of the First Tree is: 2 4 6 10 12 14 16 18 22 24 26 28 30
Inorder traversal of the Second Tree is: 32 34 36 38 42 44 46 50 60

---------------------------------------------
```

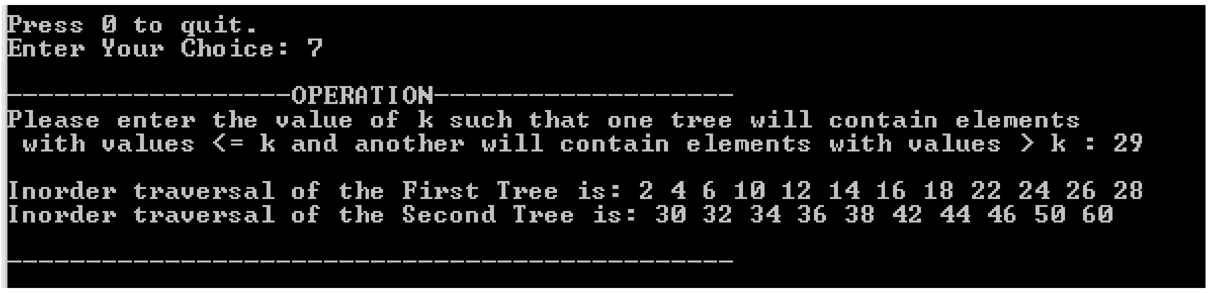
Left Tree:



Right Tree:

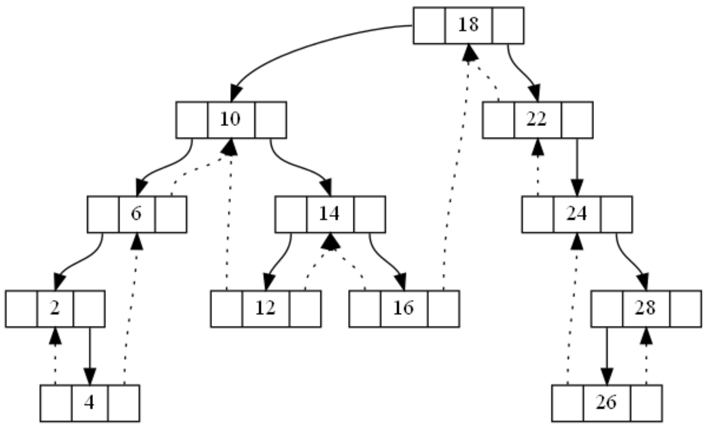


Split around 29:

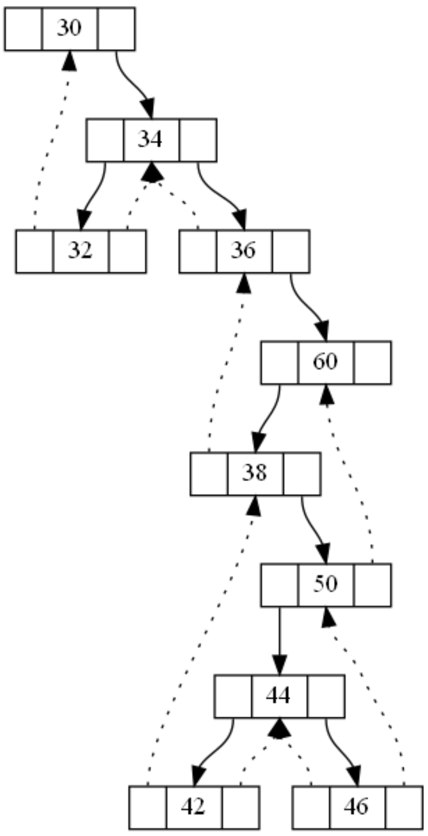


```
Press 0 to quit.
Enter Your Choice: 7

------------------OPERATION-------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 29

Inorder traversal of the First Tree is: 2 4 6 10 12 14 16 18 22 24 26 28
Inorder traversal of the Second Tree is: 30 32 34 36 38 42 44 46 50 60

----------------------------------------------
```

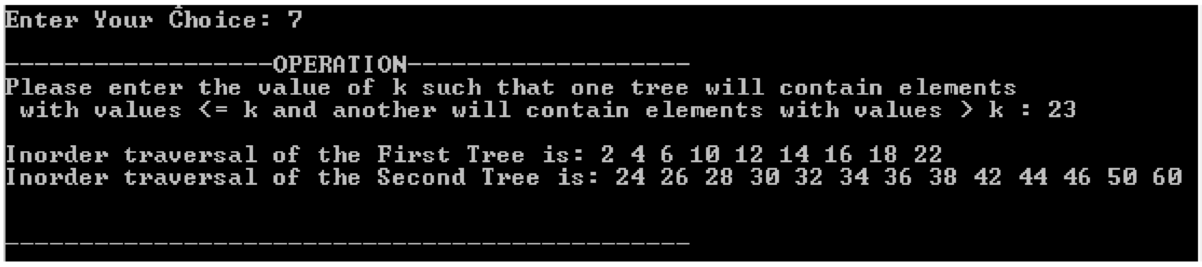
Left Tree:



Right Tree:

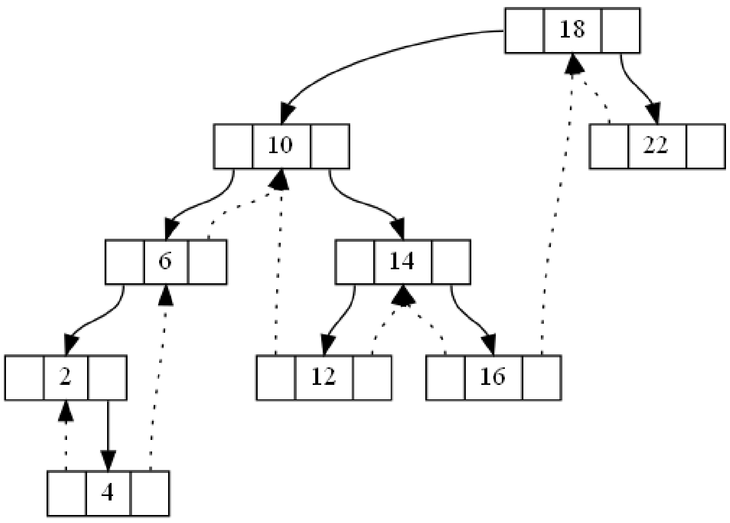


Split around 23:

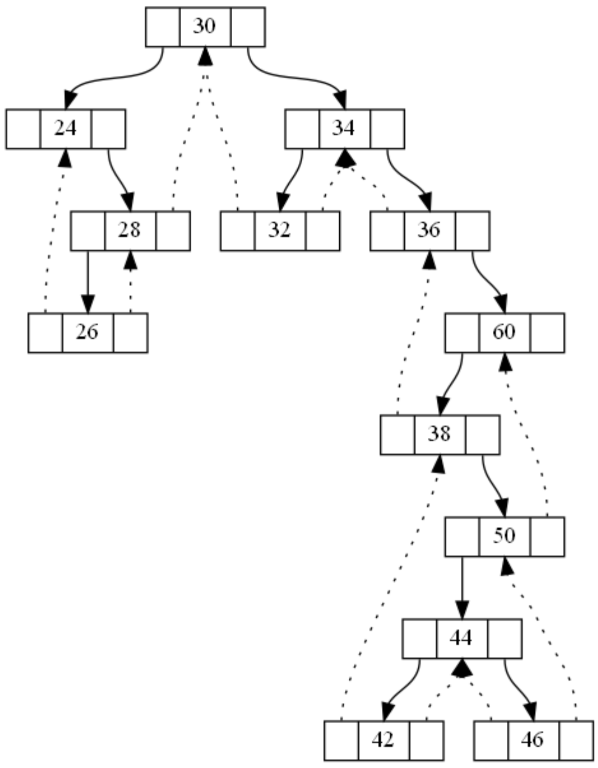


```
Enter Your Choice: 7

------------------OPERATION-------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 23

Inorder traversal of the First Tree is: 2 4 6 10 12 14 16 18 22
Inorder traversal of the Second Tree is: 24 26 28 30 32 34 36 38 42 44 46 50 60


----------------------------------------------
```

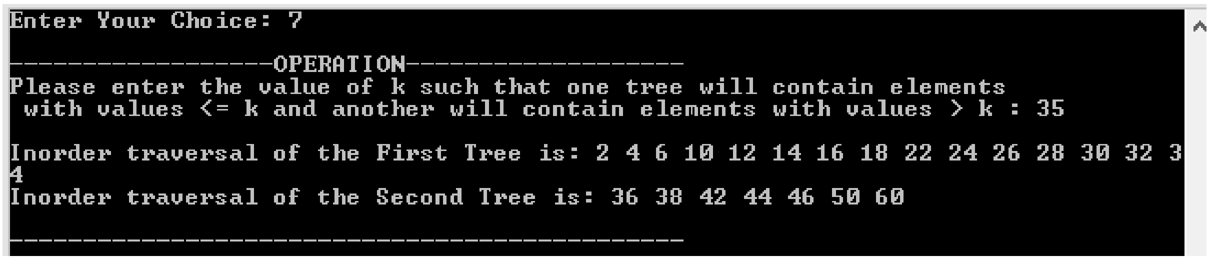
Left Tree:



Right Tree:

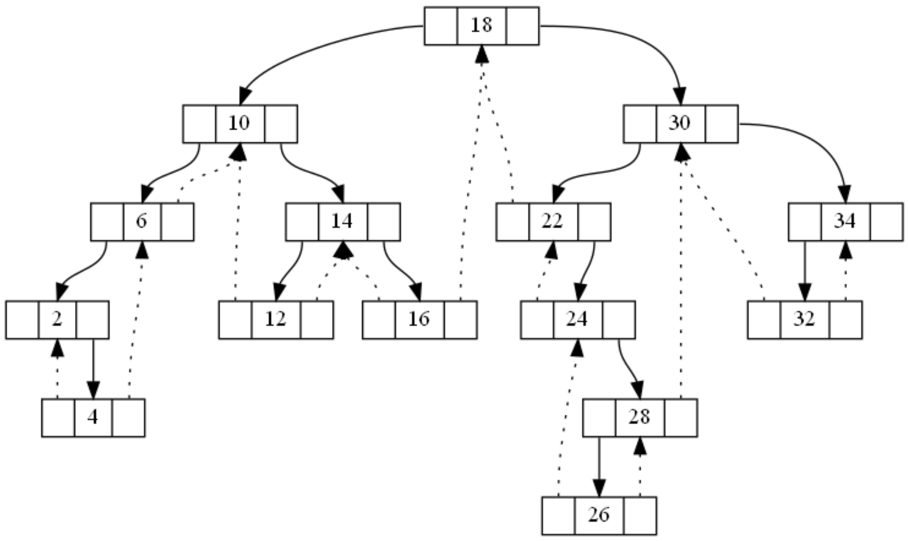


Split around 35:

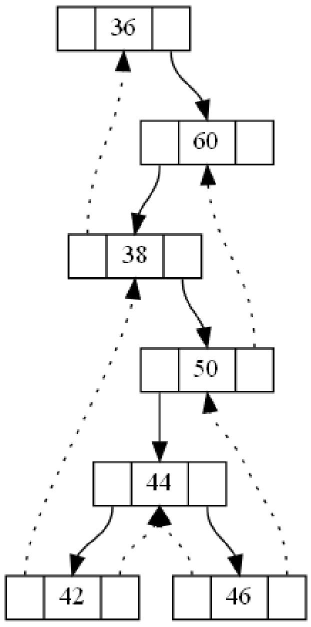


```
Enter Your Choice: 7

-----------------OPERATION------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 35

Inorder traversal of the First Tree is: 2 4 6 10 12 14 16 18 22 24 26 28 30 32 3
4
Inorder traversal of the Second Tree is: 36 38 42 44 46 50 60

----------------------------------------------
```

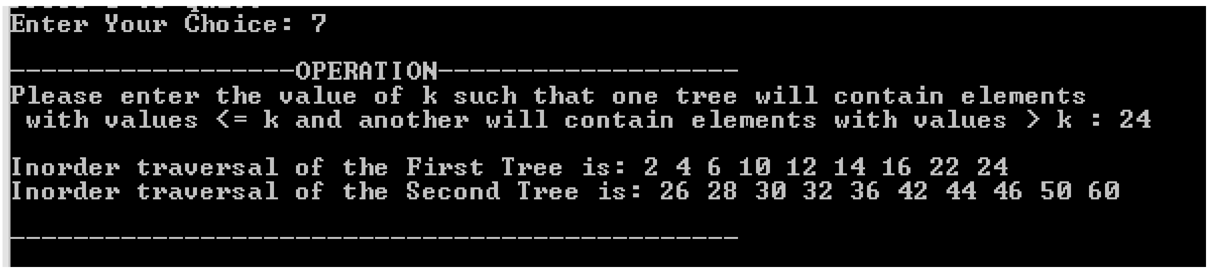
Left Tree:



Right Tree:

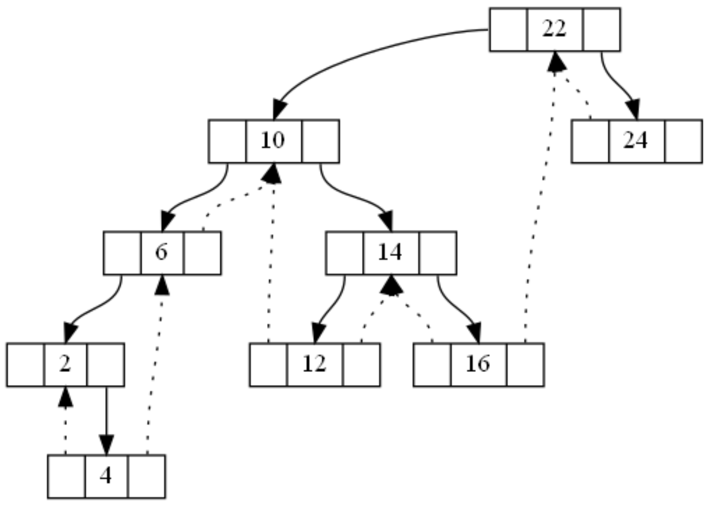


Split around 24:

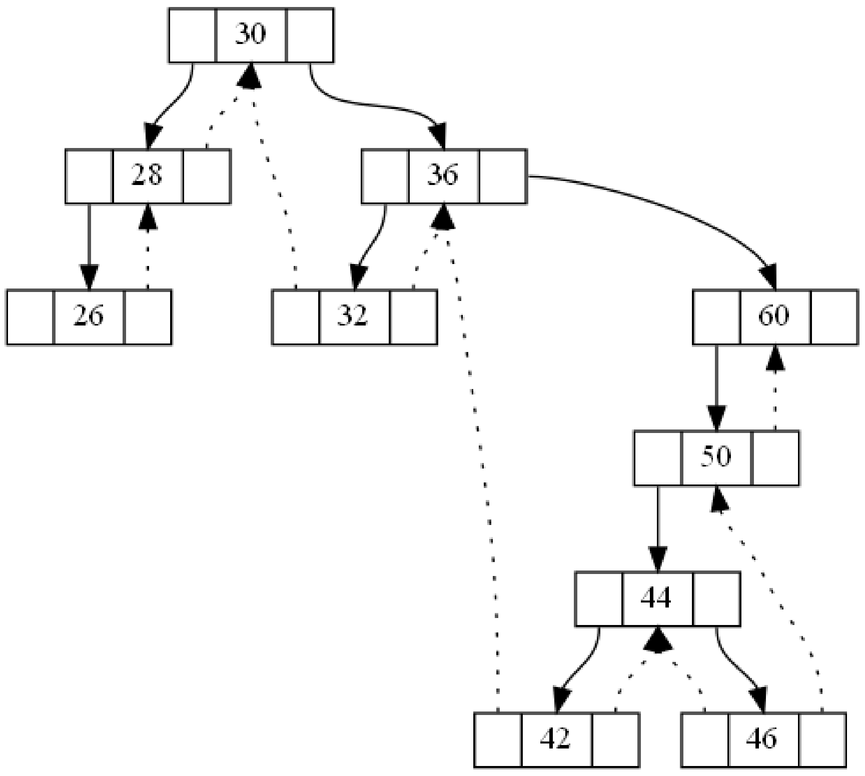


```
Enter Your Choice: 7

------------------OPERATION-------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 24

Inorder traversal of the First Tree is: 2 4 6 10 12 14 16 22 24
Inorder traversal of the Second Tree is: 26 28 30 32 36 42 44 46 50 60

----------------------------------------------
```

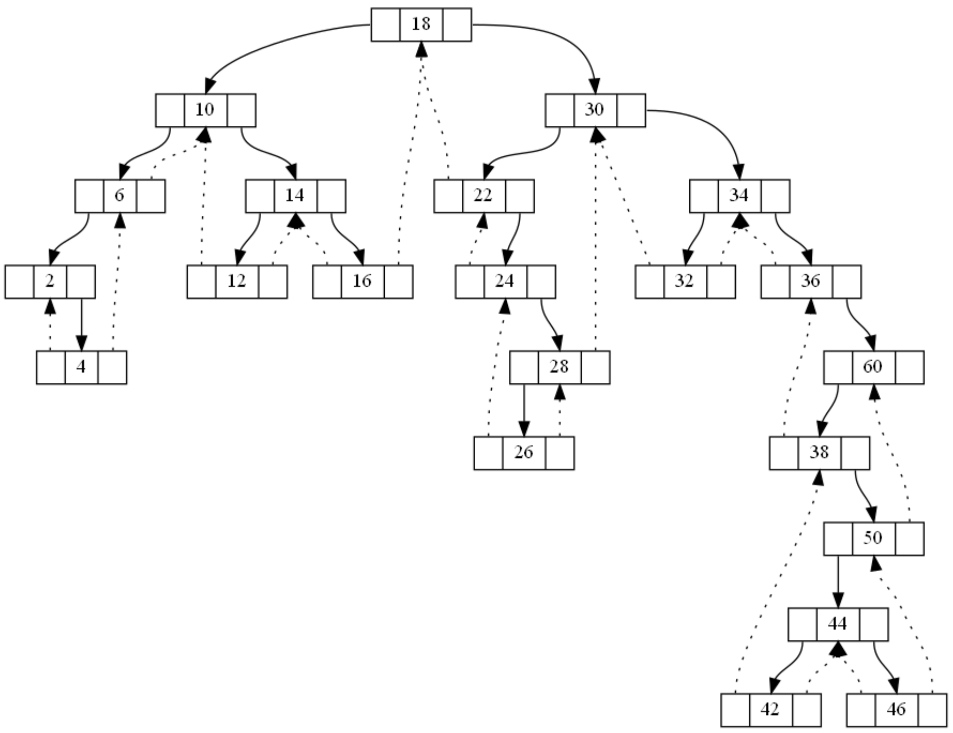
Left Tree:



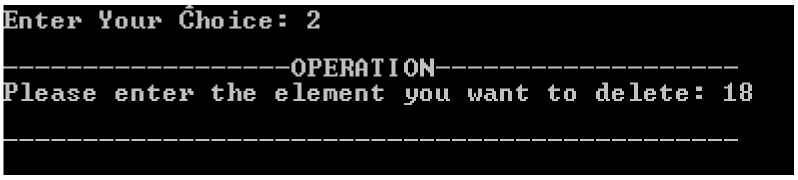
Right Tree:



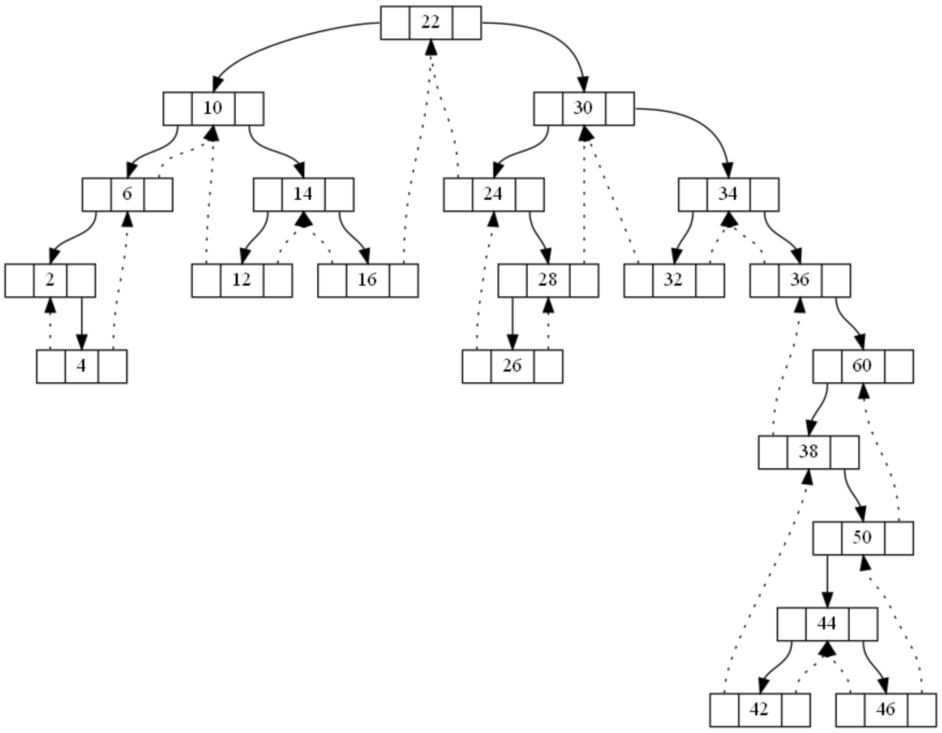
Let us now check deletion: Original Tree:



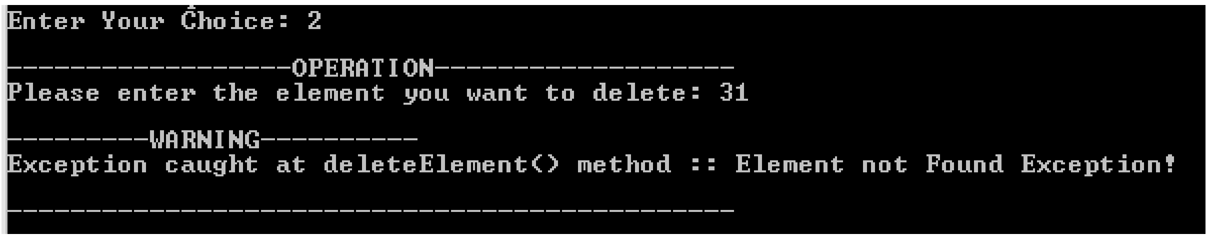
Delete 18



```
Enter Your Choice: 2

-------------------OPERATION--------------------
Please enter the element you want to delete: 18

------------------------------------------------
```

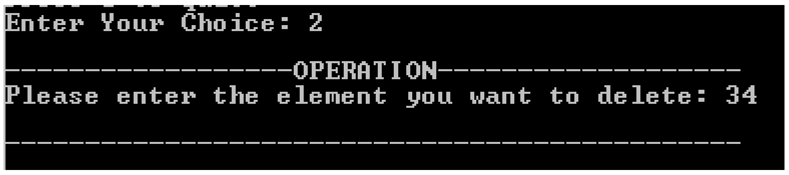


Delete 31. As 31 does not exist in the tree, there is an exception:

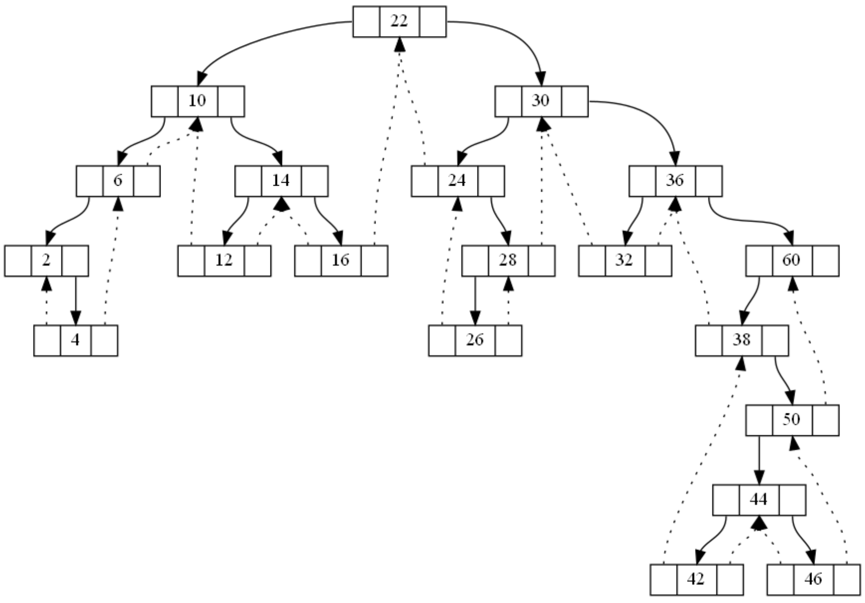


```
Enter Your Choice: 2

------------------OPERATION-------------------
Please enter the element you want to delete: 31

---------WARNING----------
Exception caught at deleteElement() method :: Element not Found Exception!

----------------------------------------------
```

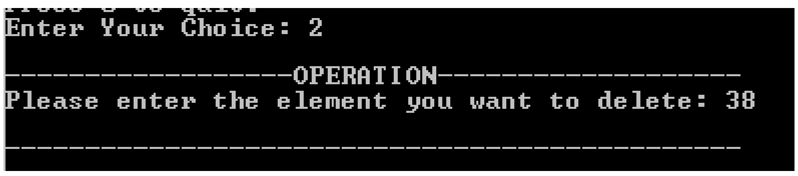
Delete34:



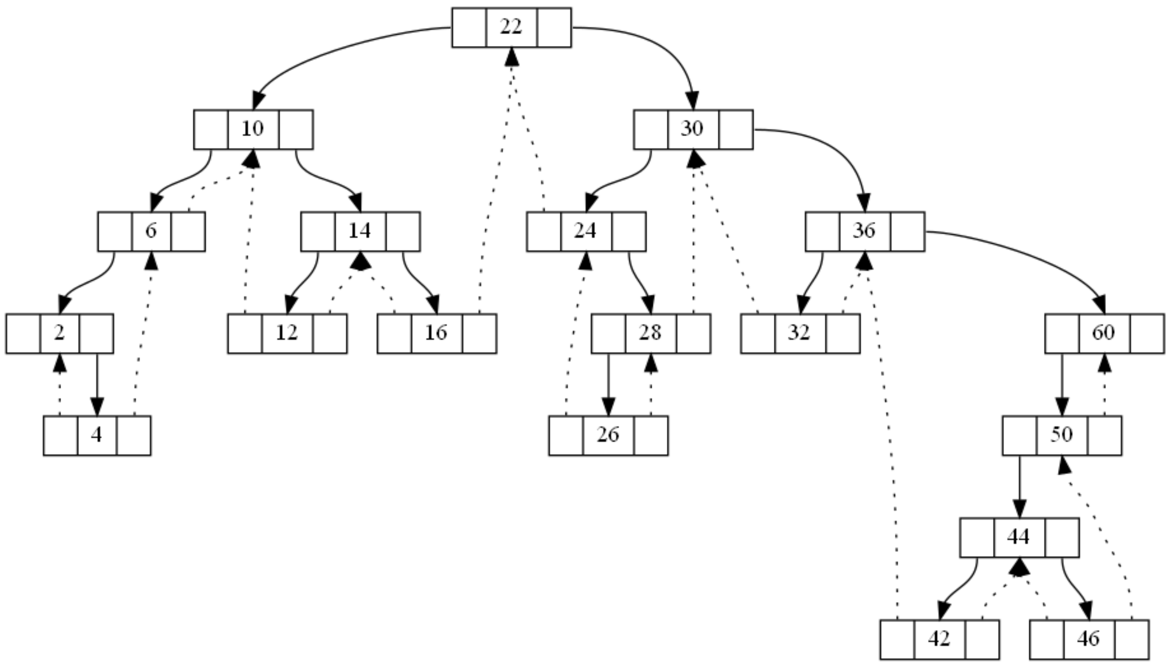
```
Enter Your Choice: 2

------------------OPERATION-------------------
Please enter the element you want to delete: 34

----------------------------------------------
```



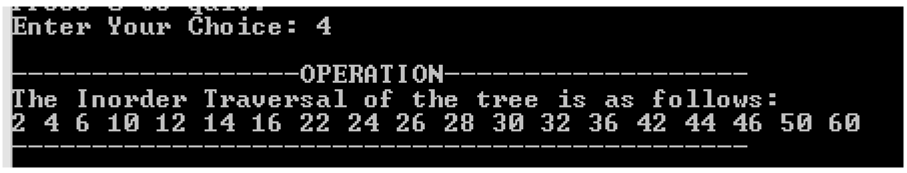
Delete 38:



```
Enter Your Choice: 2

------------------OPERATION-------------------
Please enter the element you want to delete: 38

----------------------------------------------
```

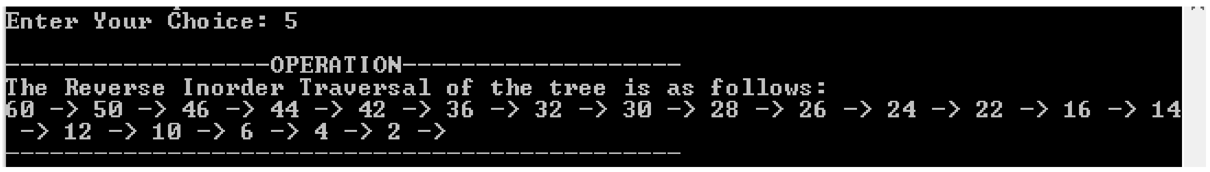


Print Inorder at this stage:



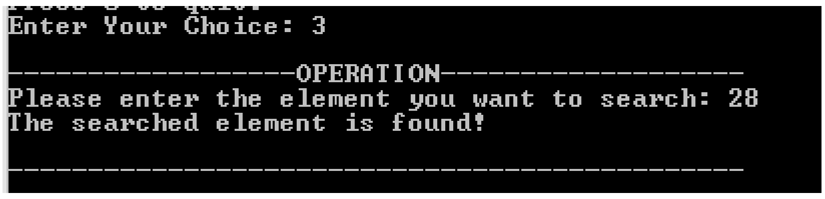
```
Enter Your Choice: 4

------------------OPERATION-------------------
The Inorder Traversal of the tree is as follows:
2 4 6 10 12 14 16 22 24 26 28 30 32 36 42 44 46 50 60
----------------------------------------------
```

Print Reverse Inorder at this stage:



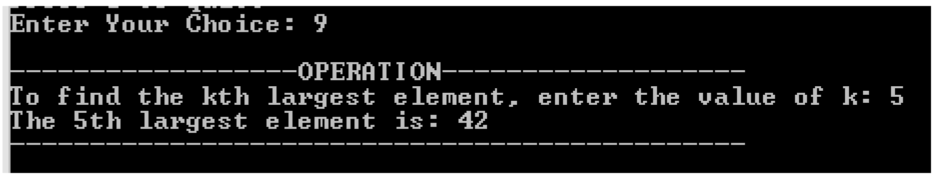
```
Enter Your Choice: 5

------------------OPERATION-------------------
The Reverse Inorder Traversal of the tree is as follows:
60 -> 50 -> 46 -> 44 -> 42 -> 36 -> 32 -> 30 -> 28 -> 26 -> 24 -> 22 -> 16 -> 14
 -> 12 -> 10 -> 6 -> 4 -> 2 ->
----------------------------------------------
```

Search for element 28:



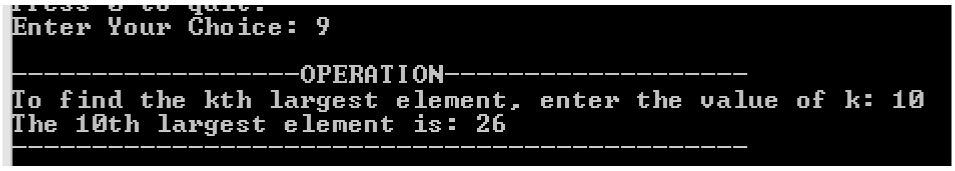
```
Enter Your Choice: 3

------------------OPERATION-------------------
Please enter the element you want to search: 28
The searched element is found!

----------------------------------------------
```

5th Largest element is 42:



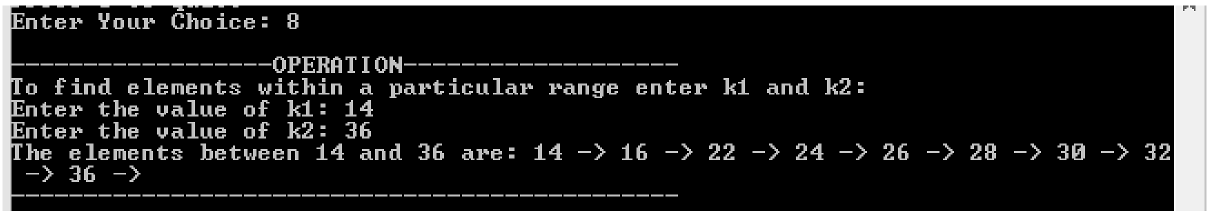
```
Enter Your Choice: 9

------------------OPERATION--------------------
To find the kth largest element, enter the value of k: 5
The 5th largest element is: 42
-----------------------------------------------
```

10th Largest element is 26:



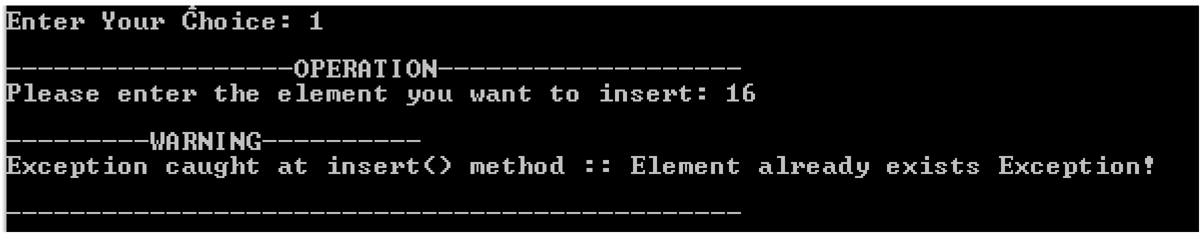
```
Enter Your Choice: 9

------------------OPERATION--------------------
To find the kth largest element, enter the value of k: 10
The 10th largest element is: 26
-----------------------------------------------
```

Find range of elements between 14 and 36:



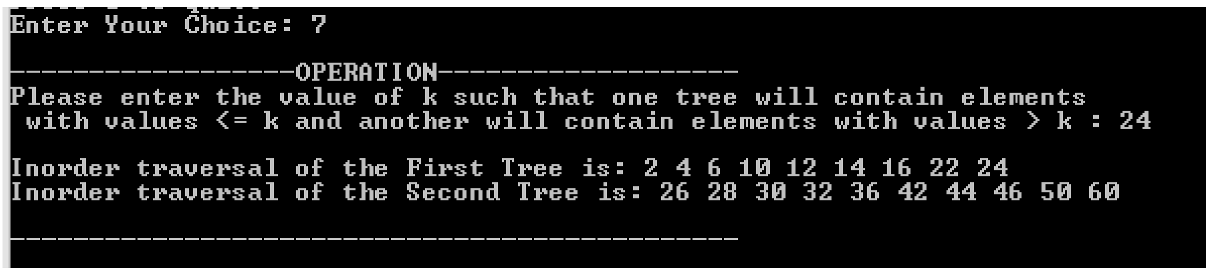
```
Enter Your Choice: 8

------------------OPERATION-------------------
To find elements within a particular range enter k1 and k2:
Enter the value of k1: 14
Enter the value of k2: 36
The elements between 14 and 36 are: 14 -> 16 -> 22 -> 24 -> 26 -> 28 -> 30 -> 32
 -> 36 ->
----------------------------------------------
```

Insert 16: Since 16 already exists, there is an exception:



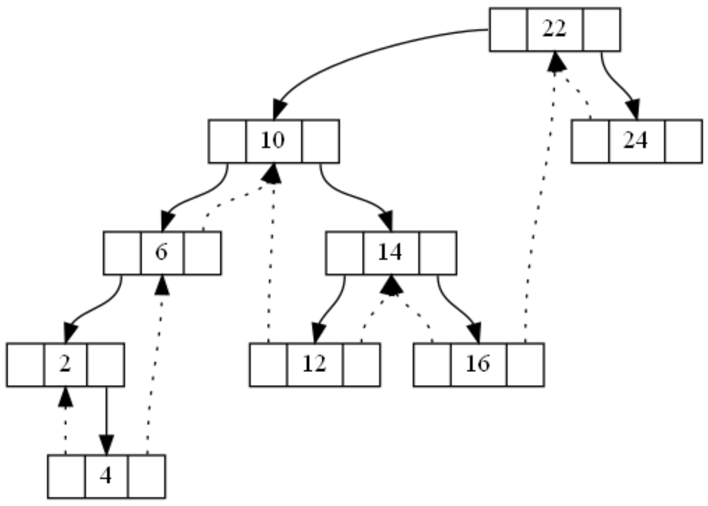
```
Enter Your Choice: 1

------------------OPERATION--------------------
Please enter the element you want to insert: 16

---------WARNING----------
Exception caught at insert() method :: Element already exists Exception!

-----------------------------------------------
```

Split around 24:

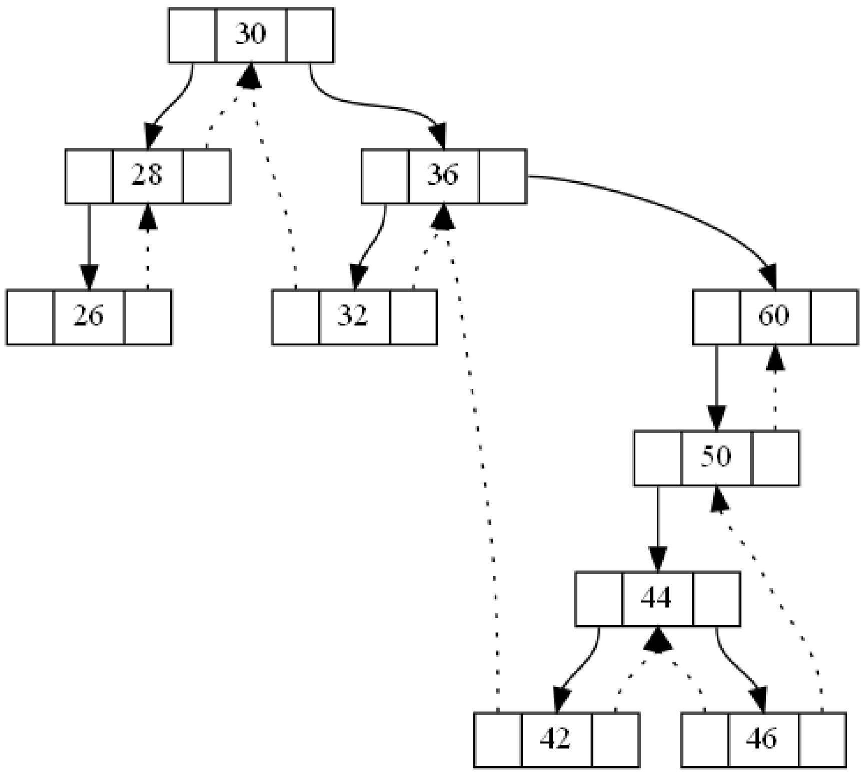


```
Enter Your Choice: 7

------------------OPERATION-------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 24

Inorder traversal of the First Tree is: 2 4 6 10 12 14 16 22 24
Inorder traversal of the Second Tree is: 26 28 30 32 36 42 44 46 50 60

----------------------------------------------
```

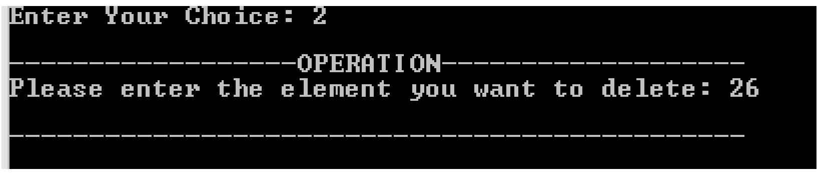
Left Tree:



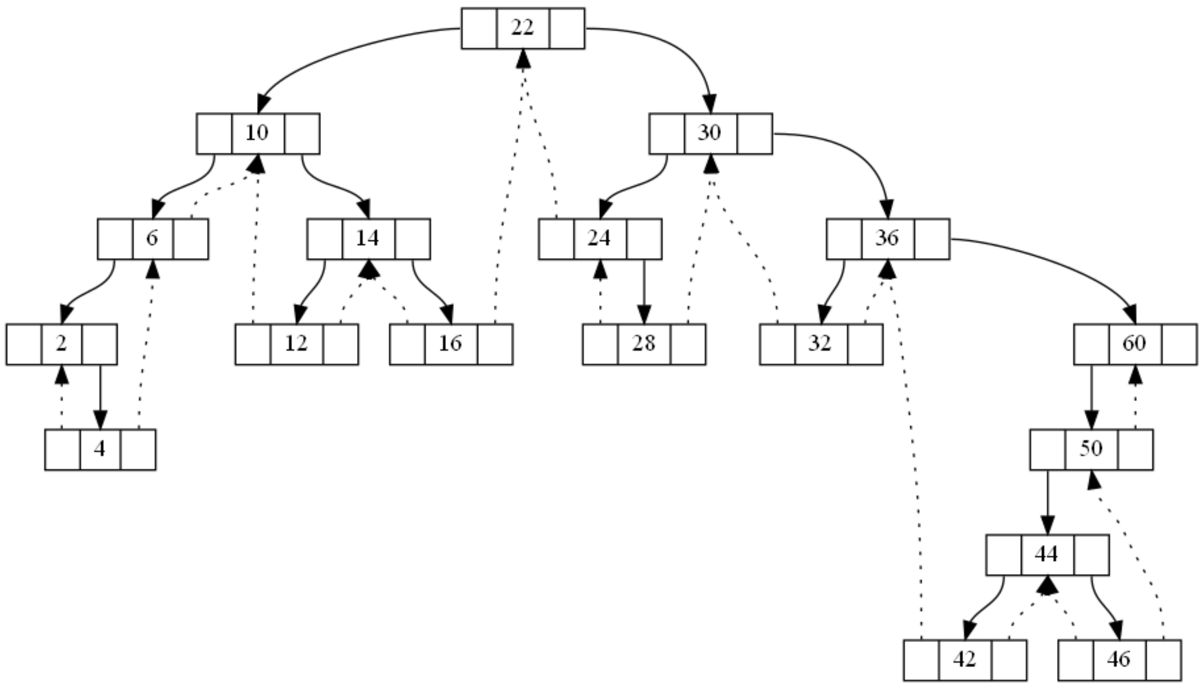
Right Tree:



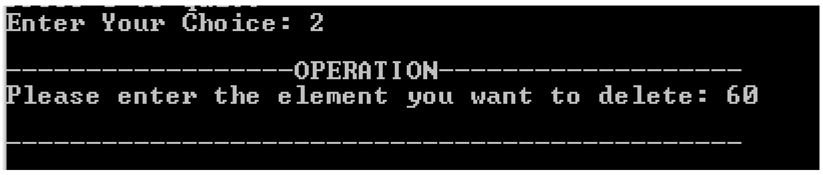
Delete 26:



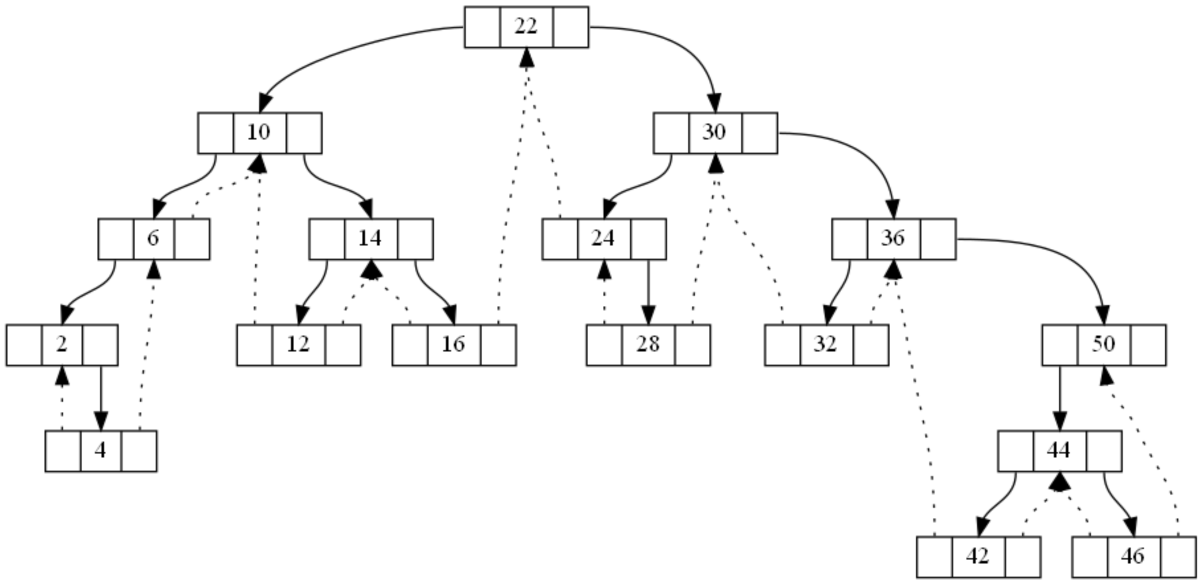
```
Enter Your Choice: 2

-------------------OPERATION--------------------
Please enter the element you want to delete: 26

------------------------------------------------
```



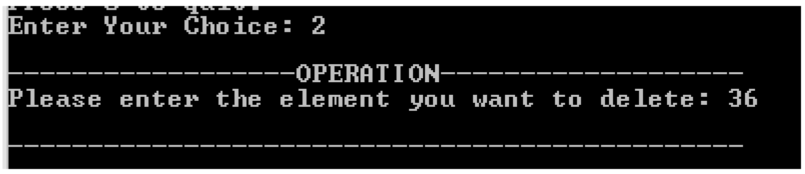
Delete 60:

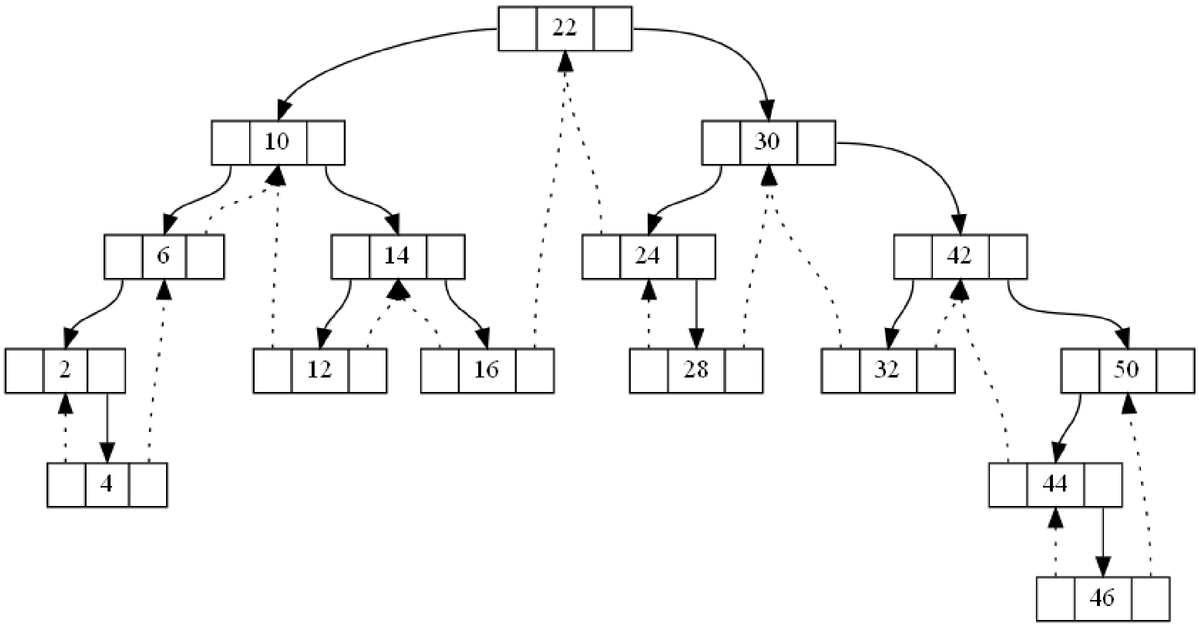


```
Enter Your Choice: 2

-------------------OPERATION--------------------
Please enter the element you want to delete: 60

------------------------------------------------
```

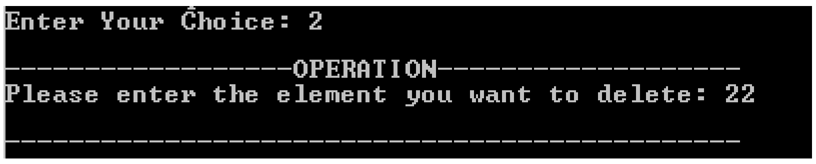


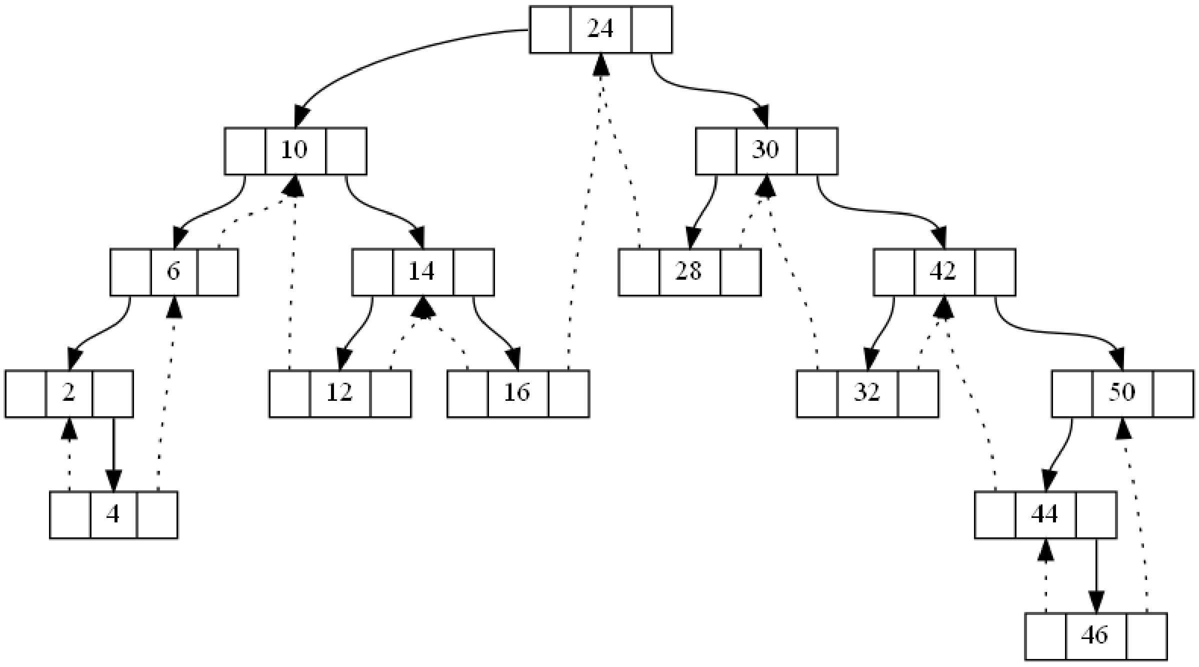
Delete 36:


```
Enter Your Choice: 2

-------------------OPERATION--------------------
Please enter the element you want to delete: 36

------------------------------------------------
```

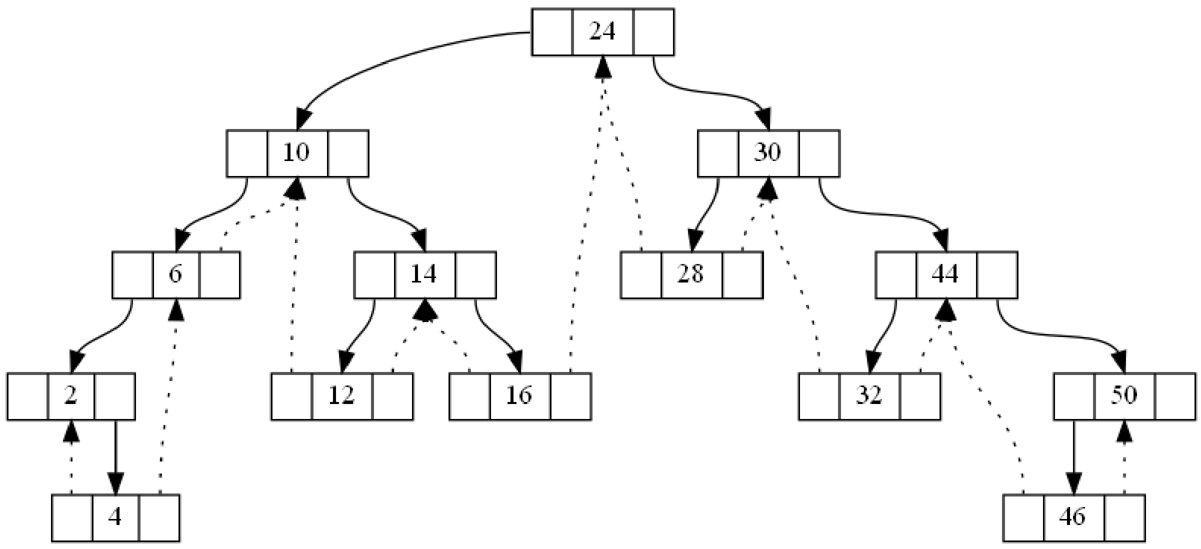


Delete 22:

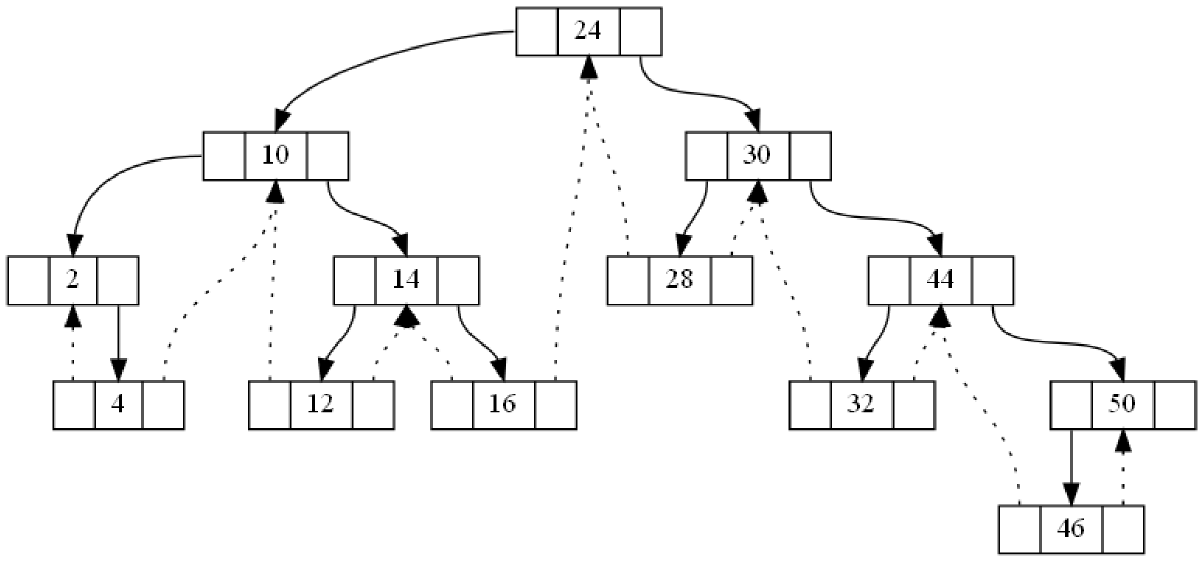

```
Enter Your Choice: 2

-------------------OPERATION--------------------
Please enter the element you want to delete: 22

------------------------------------------------
```



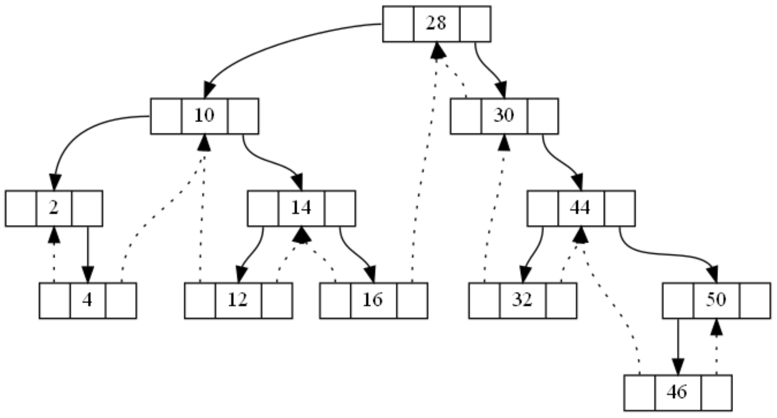
Delete 42:



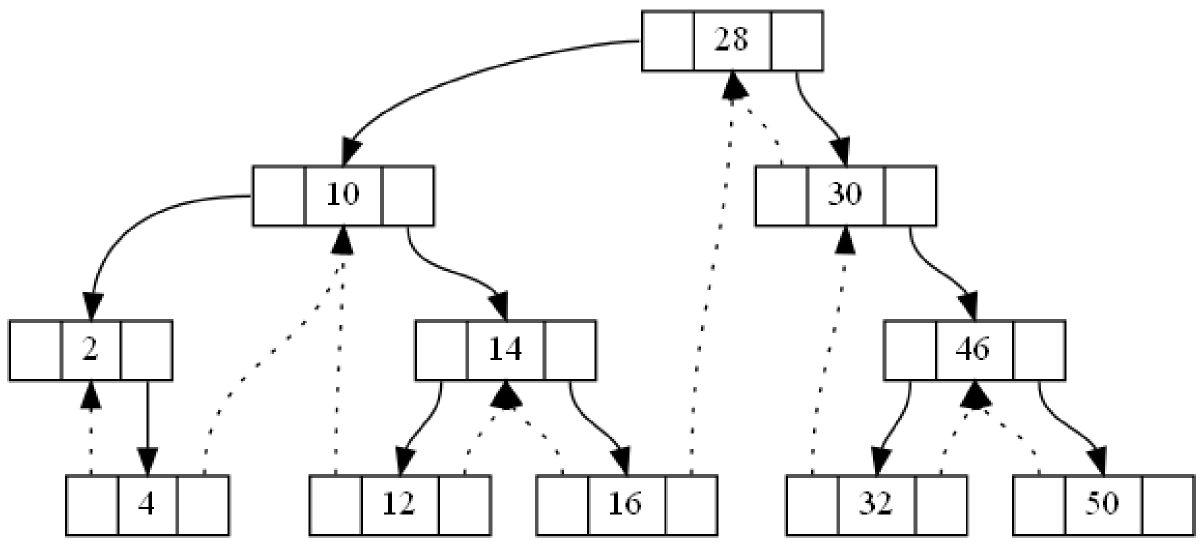
Delete 6:



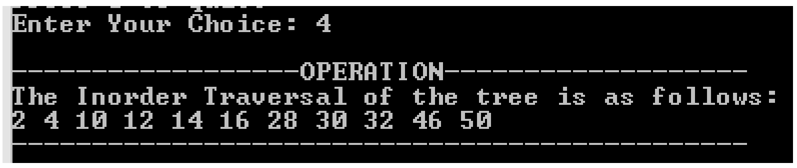
Delete 24:



Delete 44:

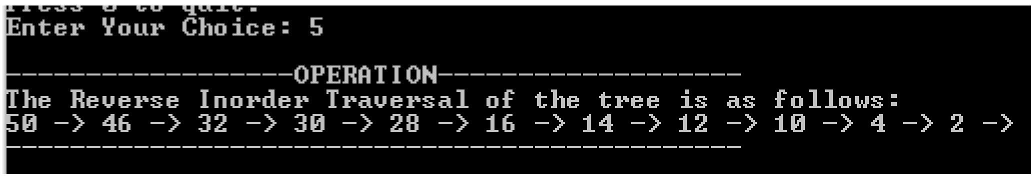


Inorder Traversal:



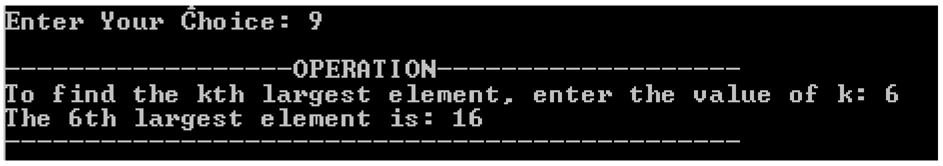
```
Enter Your Choice: 4

-------------------OPERATION--------------------
The Inorder Traversal of the tree is as follows:
2 4 10 12 14 16 28 30 32 46 50
------------------------------------------------
```

Reverse Inorder Traversal:



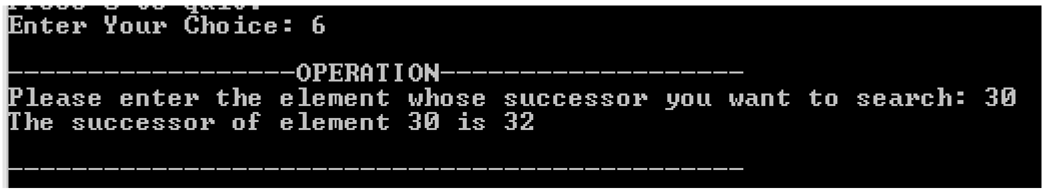
```
Enter Your Choice: 5

-------------------OPERATION--------------------
The Reverse Inorder Traversal of the tree is as follows:
50 -> 46 -> 32 -> 30 -> 28 -> 16 -> 14 -> 12 -> 10 -> 4 -> 2 ->
------------------------------------------------
```

6th Largest:



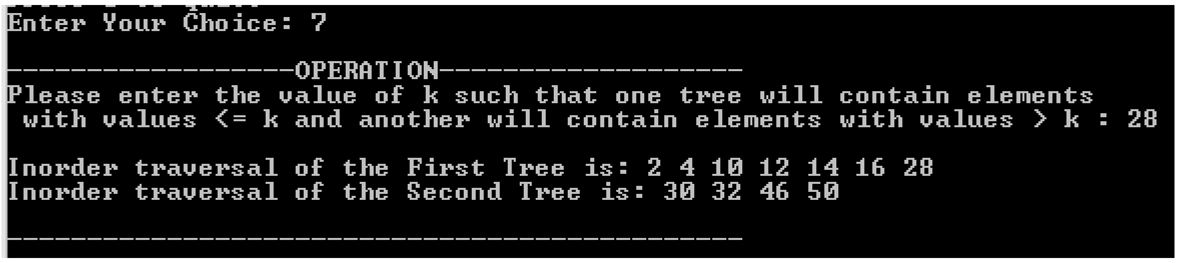
```
Enter Your Choice: 9

-------------------OPERATION--------------------
To find the kth largest element, enter the value of k: 6
The 6th largest element is: 16
------------------------------------------------
```

Successor of 30 is 32:



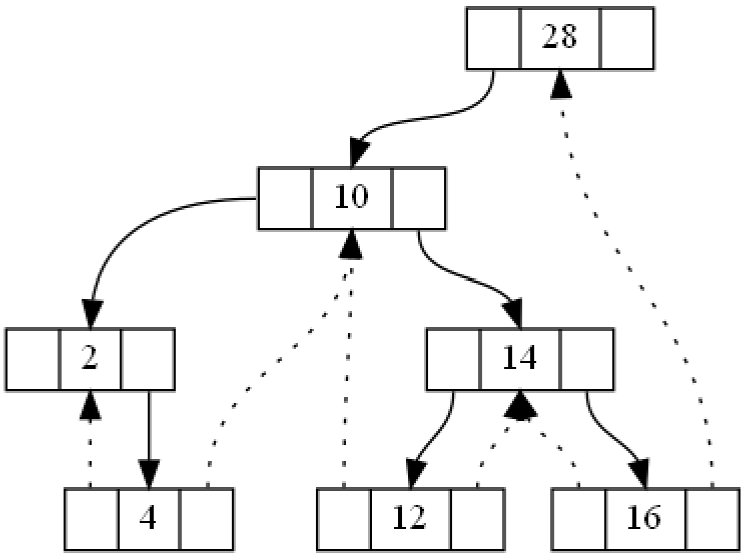
```
Enter Your Choice: 6

-------------------OPERATION--------------------
Please enter the element whose successor you want to search: 30
The successor of element 30 is 32

------------------------------------------------
```

Split around 28:

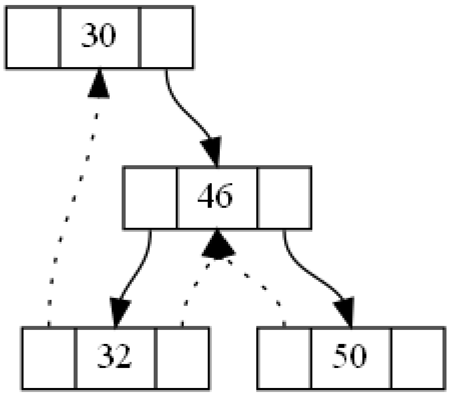


```
Enter Your Choice: 7

------------------OPERATION-------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 28

Inorder traversal of the First Tree is: 2 4 10 12 14 16 28
Inorder traversal of the Second Tree is: 30 32 46 50

----------------------------------------------
```

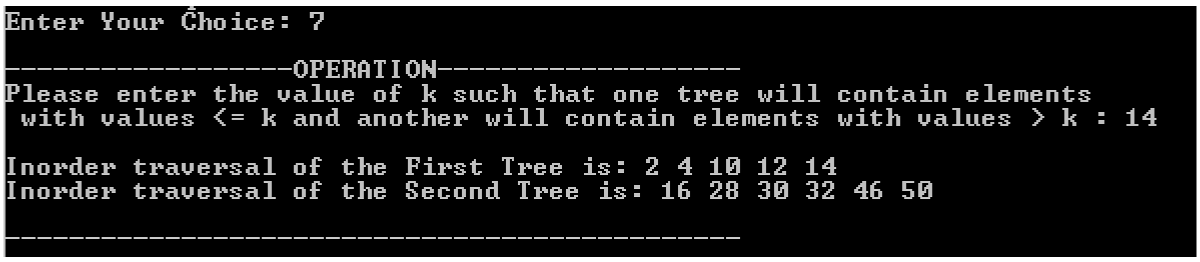
Left Tree:



Right Tree:

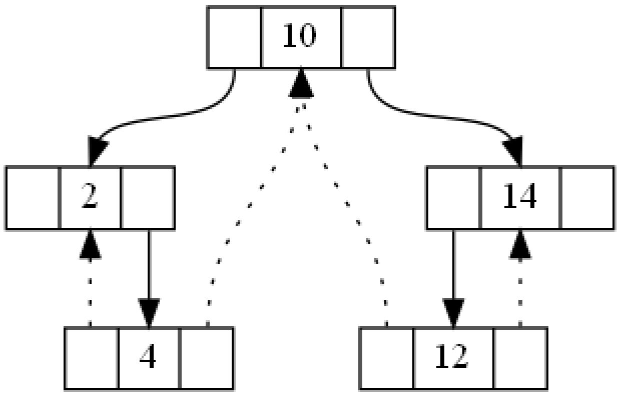


Split Around 14:

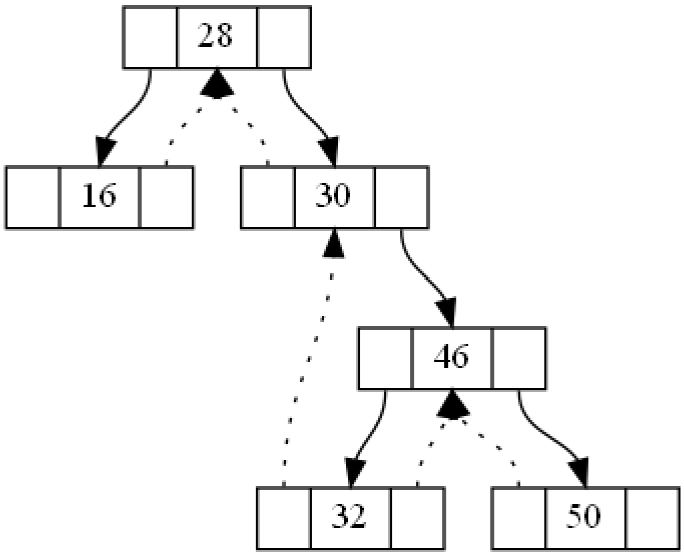


```
Enter Your Choice: 7

-------------------OPERATION--------------------
Please enter the value of k such that one tree will contain elements
 with values <= k and another will contain elements with values > k : 14

Inorder traversal of the First Tree is: 2 4 10 12 14
Inorder traversal of the Second Tree is: 16 28 30 32 46 50

------------------------------------------------
```

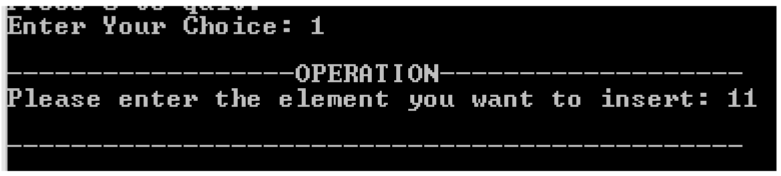
Left Tree:



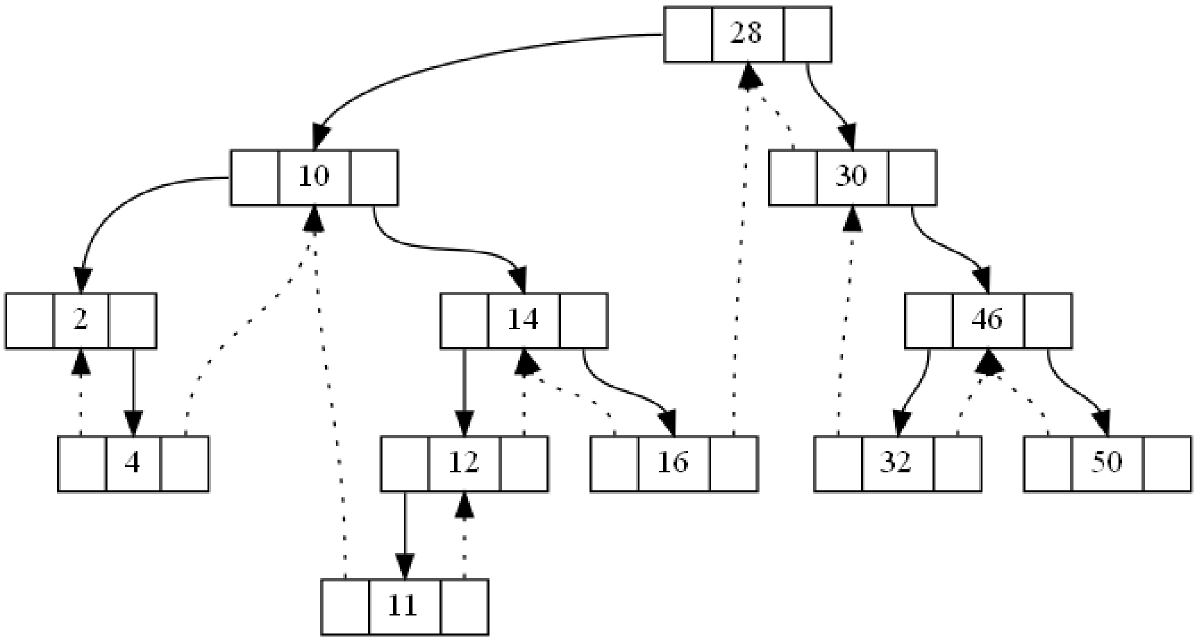
Right Tree:



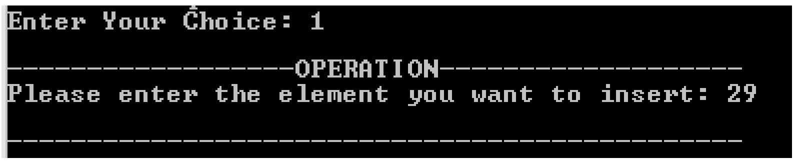
Insert 11:



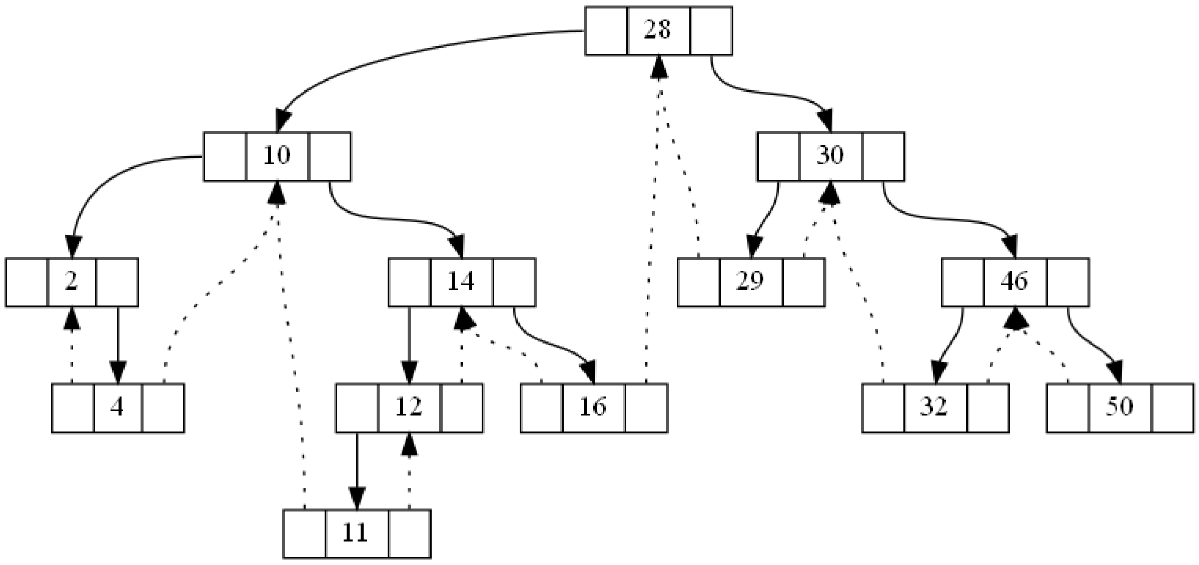
```
Enter Your Choice: 1

-------------------OPERATION--------------------
Please enter the element you want to insert: 11

------------------------------------------------
```



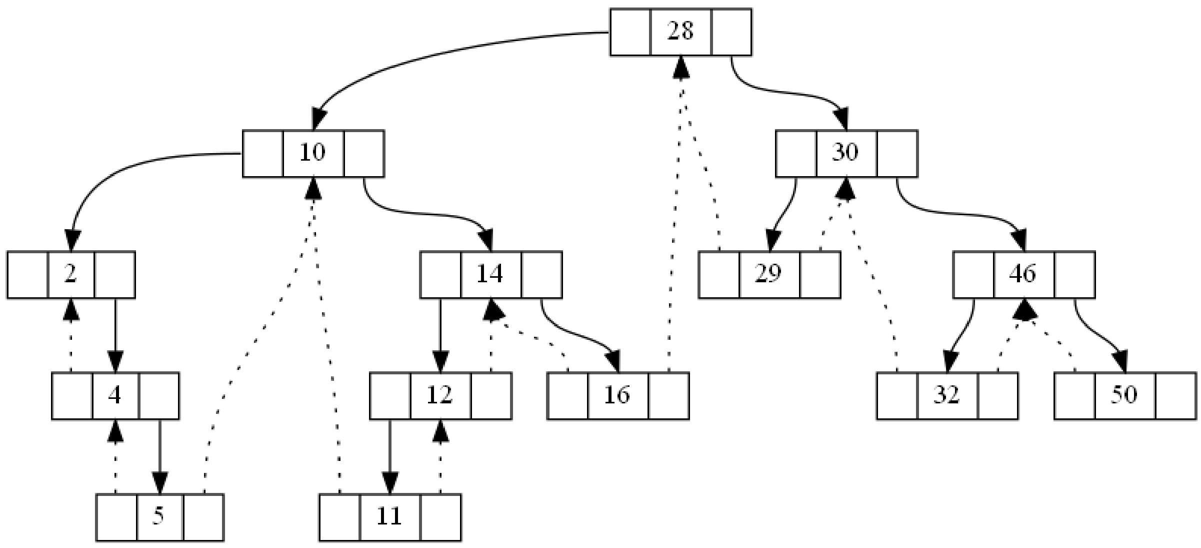
Insert 29:



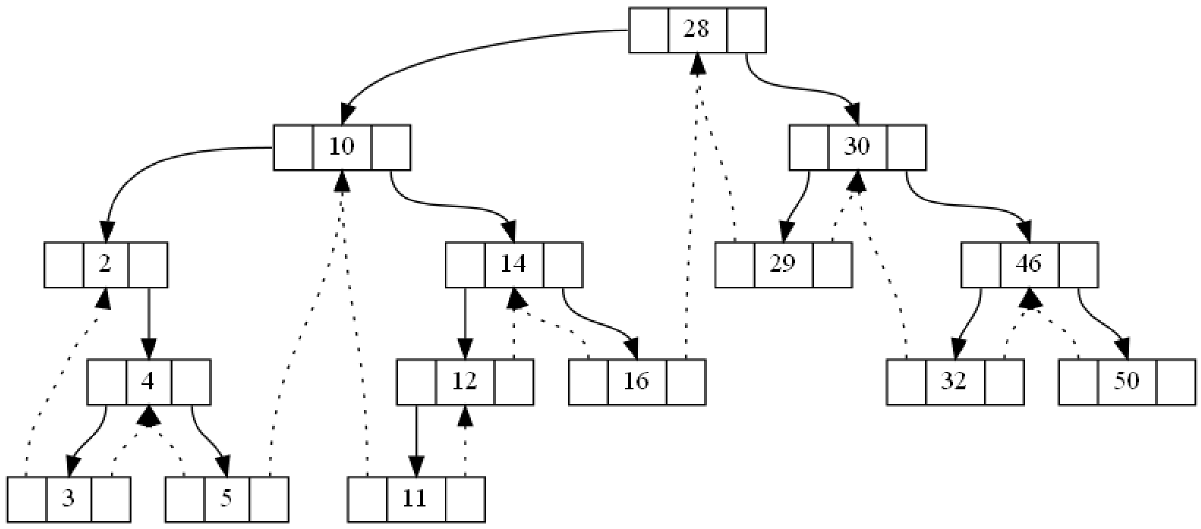
```
Enter Your Choice: 1

-------------------OPERATION--------------------
Please enter the element you want to insert: 29

------------------------------------------------
```



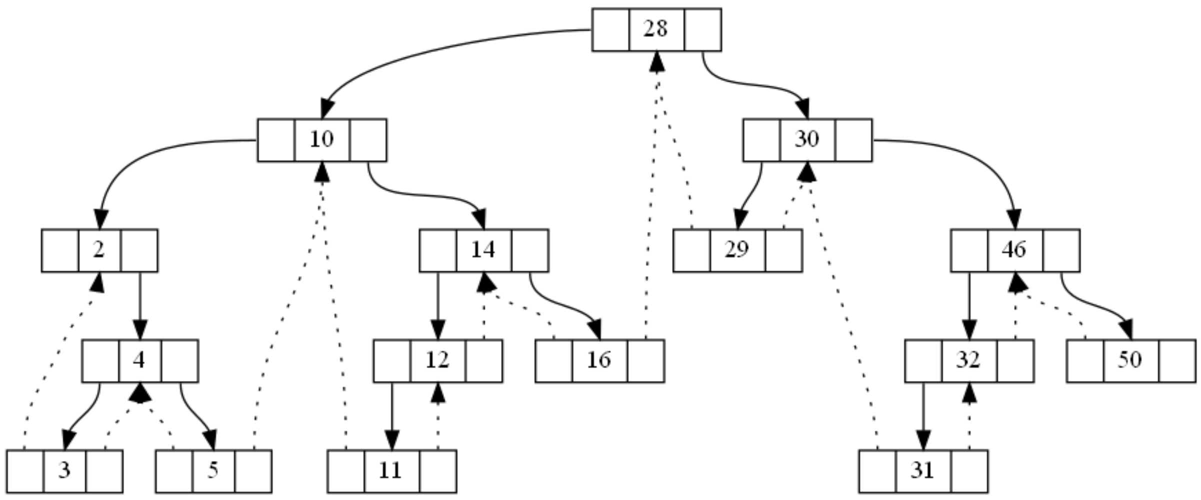
Insert 5:



Insert 3:



Insert 31:



Thank You