Contents

CS513: BST Assignment

Prateekshya Priyadarshini

M.Tech CSE

1 Approaches

1.1 Keeping a dummy node for linked list

For making addition of nodes easier, one dummy node is created while creating the linked list object. When returning the linked list, the dummy node is first removed and then the list is being returned.

1.2 Using level order traversal for printing the tree

Since the order of nodes mentioned in the graphviz file is important, we need to do a traversal of the tree starting from the root such that the root should be mentioned at the top. This can be done using either preorder traversal or level order traversal. Here level order traversal is being considered and a separate queue is implemented for that purpose.

1.3 Three global variables for printing the tree

1. **fileCount**
   This variable is initialized to 0. It gets incremented everytime a new graphviz file is created. So that we can create different files each time. For example-graph0.gv,graph1.gv,graph2.gv etc.

2. **fileType**
   This variable stores 0 for windows OS, 1 for linux OS. More options can be added as per requirement. This variable helps us to decide which type of commands file we have to create. If we are in windows, we need to add all the commands needed to convert graphviz files to png files into a batch file. Similary in linux we have to add all of them to a shell script.

3. **color**
   For reference, after splitting the original tree, both the splitted trees are also being printed. To distinguish between these files, all type of trees which are considered to be the original ones are printed in black, the splitted trees are printed in two different colors i.e. blue and green.

1.4 Extra variables in Tree Node structure

To make the tree threaded, there are two boolean variables. They store $true$ if the respective pointer is threaded and $false$ if the pointer points to a legit child.
To get the k^{th} largest element in $O(h)$ time; where h is the height of the tree; there are two integer variables. They store the count of the nodes present in the left subtree and right subtree respectively.

1.5 Deleting the nodes with two children using successor replacement strategy

When a node with two children is being deleted, it can be replaced with either with its predecessor or with its successor. Here successor replacement strategy is chosen.

1.6 Creating batch files or shell scripts

Since the commands to generate a .png file from a .gv file are not straight forward for a beginner, this program generates batch file in windows and shell script in linux which contains all those commands. Every time a new tree is printed, these files get executed and the respective images get generated. For reference, the images won't be deleted or replaced till the program is being executed.

1.7 Storing the entire execution process

The entire execution process is stored in **output.txt**. This file can be referred to check what went wrong, which images refer to which trees etc.

2 Files, Functions and Their Logic

2.1 nodes.h

This header file contains node structures for linked list, queue and binary search tree.

2.2 datastructures.h

This header file contains linked list class, queue class and binary search tree class along with their respective function prototypes.

2.3 list.h

This header file contains the below mentioned functions for linked list.

2.3.1 void addNode(TreeNode * treeNode)

This function takes a node of type TreeNode and adds it's data to the end of the linked list. Time complexity is constant.

2.3.2 void trim()

This function removes the dummy node from the linked list. Time complexity is constant.

2.4 queue.h

This header file contains the below mentioned functions for queue.

2.4.1 void enqueue(TreeNode * node)

This function takes a node of type TreeNode and adds the entire node to the queue at the rear end. Time complexity is constant.

2.4.2 TreeNode * dequeue()

This function removes and returns the front element of the queue. Time complexity is constant.

2.4.3 bool isEmpty()

This function checks whether the queue is empty or not. If the size is zero then it returns true otherwise returns false. Time complexity is constant.

2.4.4 int getSize()

This functions returns the current size of the queue. Time complexity is constant.

2.5 tbst.h

This header file contains the below mentioned functions for threaded binary search tree.

2.5.1 void insert(int x)

This function is called from the main function. It only inserts the node if the root is null. Otherwise it calls another insertion function (Refer void insertIntoTree(TreeNode * root, int x)) which uses recursion. It also throws exception if the given value is already present in the tree. The time complexity of this function including the recursion is $O(h)$ where h is the height of the tree.

2.5.2 TreeNode * search(int x)

This function is used to search a value in the tree. A pointer is moved from root till leaf according to the given value. If the value is found in the path, then the reference is returned otherwise null is returned. The time complexity of this function is $O(h)$ where h is the height of the tree.

2.5.3 void deleteX(int x)

This function deletes a given node. First it searches for the given value. If the value is not present then it throws an exception. Otherwise its parent is searched. There are three possible cases.

1. **Deleting a leaf node**
   This part calls another function (Refer TreeNode * deleteLeafNode(TreeNode * node, TreeNode * parent)) which deletes the leaf node and returns the deleted node.

2. **Deleting a node having single child**
   This part calls another function (Refer TreeNode * deleteNodeWithSingleChild(TreeNode * node, TreeNode * parent)) which deletes the required node and returns the deleted node.

3. **Deleting a node having two children**
   This part finds out the successor of the current node and the parent of the successor node. Since the successor node can never have a left subtree, not even a left child, so it can be deleted using the functions which delete leaf nodes or nodes having single child. The functions also return the node which is being deleted. So using those functions the successor can be detached from the tree. After that, it is being replaced with the node to be deleted. All the thread information and subtree count information are maintained properly.

The time complexity for this function is roughly $O(h)$ where h is the height of the tree.

2.5.4 List reverseInOrder()

This function uses the threads to find out the reverse inorder traversal. It returns a linked list containing all the values in reverse inorder. No extra stack is being used.

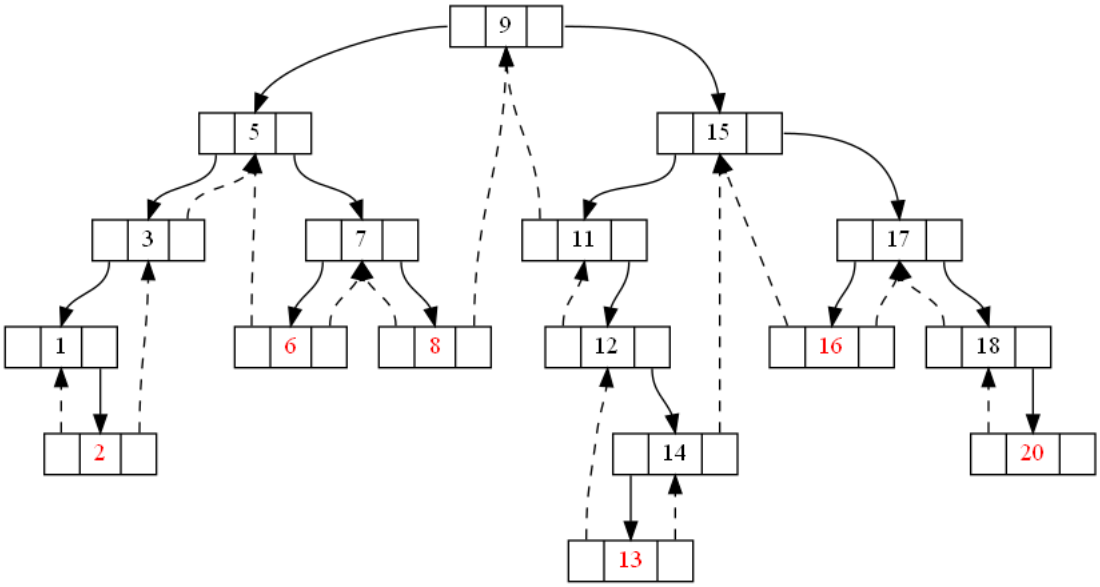


Figure 1: Example tree : red values indicate leaf nodes

2.5.5 int successor(TreeNode * xNode)

This function find out the successor value of a given node. If the given node is the largest node in the tree, it throws an exception. If the given node is null, then also it throws an exception. The time complexity of this function is $O(h)$ where h is the height of the tree.

2.5.6 TreeNode * split(int k)

This function splits the tree around a given value k such that all the values in one tree are less than or equal to k and the values in the other tree are greater than k. Then it attaches those trees to a dummy node and returns the dummy node.
The logic of splitting is simple. Maintain two pointers which point to a parent and a child at the same time. If the parent and the child are at two opposite sides of k, that means they will belong to different trees. We need to cut the tree form that point and move the pointers downwards. On the next cut, we need to hold the part being cut using the null pointer of the previous cut. Let's understand this with an example at **Figure 1**. Suppose we are splitting this tree in **Figure 1** around $k = 13$. First pointer will point to root i.e. 9. Now we shall compare k and root. Since k is greater than root, second pointer will point to the right child of root i.e. 15. We can clearly see that 9 and 15 belong to different trees. So we need to make the right child of first pointer *null*. After which the tree will look like **figure 2**. (Of course we have two root pointers one of whom will point to the original root and the other will point to the second node when the first split occurs.) Here blue and red nodes are first and second nodes respectively. We would have deleted the thread, but since we know 11 and 9 will remain in the same tree, there is no need to do that here. Since this was the first split, first root will point to 9 and the other will point to 15. The subtree counts are being maintained properly.
We shall now move first pointer to second and second to the next level. Since 13 is less than 15, second pointer will move to left i.e. 11. Now 15 and 11 also belong to different trees. So we need to make another cut.
But the important thing here is, since we are now at a tree which is supposed to contain all values greater than k, this means the previous cut was made from a tree which was supposed to contain all the values less than or equal to k. Also if we are cutting some part of the current tree where first and

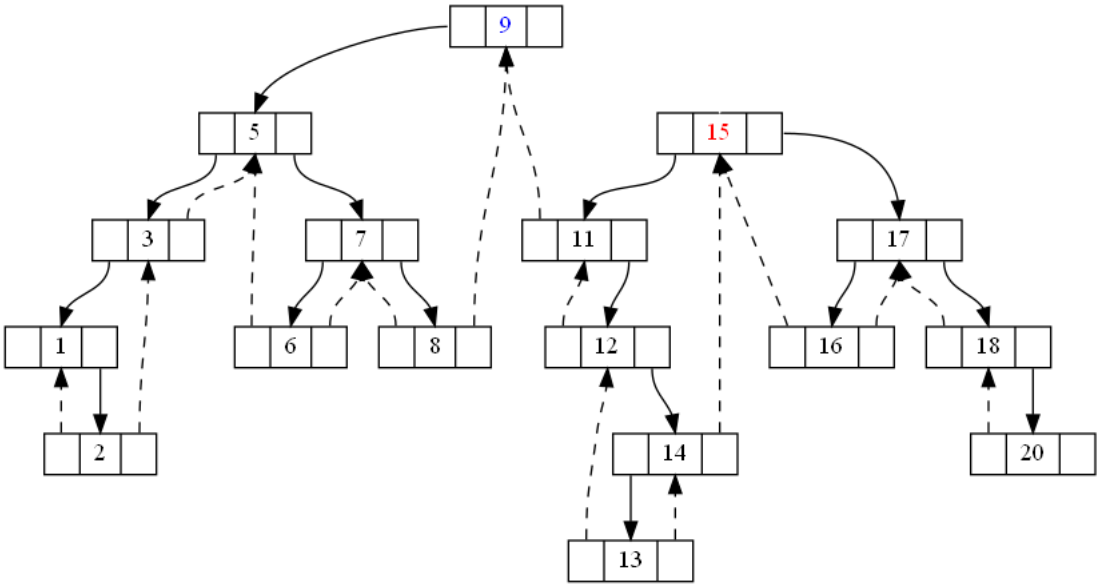


Figure 2: Example tree : after removing the link from first node

second pointers are present, then there is no way that the part which will be cut i.e. the part being pointed by the second pointer will have value greater than k. That means, the tree from which the previous cut was made and the part which is being cut now, belong to the same tree. That means here, the tree containing 9 (this was the node from which the first cut was made) and the part containing 11 (being cut now), belong to the same tree. Also we have come to the right of 9 and made the right pointer null. So we can attach 11 to the right of 9. **Figure 3** shows the process. We didn't delete the thread of 15 for the same reason. However, in the program, this condition is being checked and the threads are being removed wherever required. Now we shall move first pointer to 11. Since 13 is greater than 11, we shall move second pointer to right. Now both first and second pointers i.e. 11 and 12 belong to the same tree, so we need not do anything and we can directly move forward.
Now first will point to 12 and second will point to 14. They both belong to different trees. So we need to 14. The previous cut was done from 15. We can again see that, the node from which previous cut was made i.e. 15 and the node which is being cut now i.e. 14 belong to the same tree. We had made the left pointer of 15 *null*. Now it can point to 14.
This process will continue till first pointer reaches a leaf node. And the resultant trees will look similar to what is shown in **Figure 4**. The time complexity is $O(h)$ since we are moving from root to leaf path.

2.5.7 List inOrder()

Since we need to print the inorder traversals of the splitted trees, this function is created. It returns a linked list containing all the values of a tree in inorder.

2.5.8 List allElementsBetween(int k1, int k2)

This function returns a linked list containing all elements between k_1 and k_2 in inorder. The logic here is to spot k_1 or a value just greater than k_1, k_2 or a value just smaller than k_2. We can store those two nodes in $k_1 node$ and $k_2 node$ respectively. Then we can traverse from $k_1 node$ to $k_2 node$ node using the threads which will take roughly $O(h + N)$ where N is the number of nodes from k1 to k_2.
So we traverse from root to leaf using a pointer *temp* and we keep updating $k_1 node$ when *temp* is less

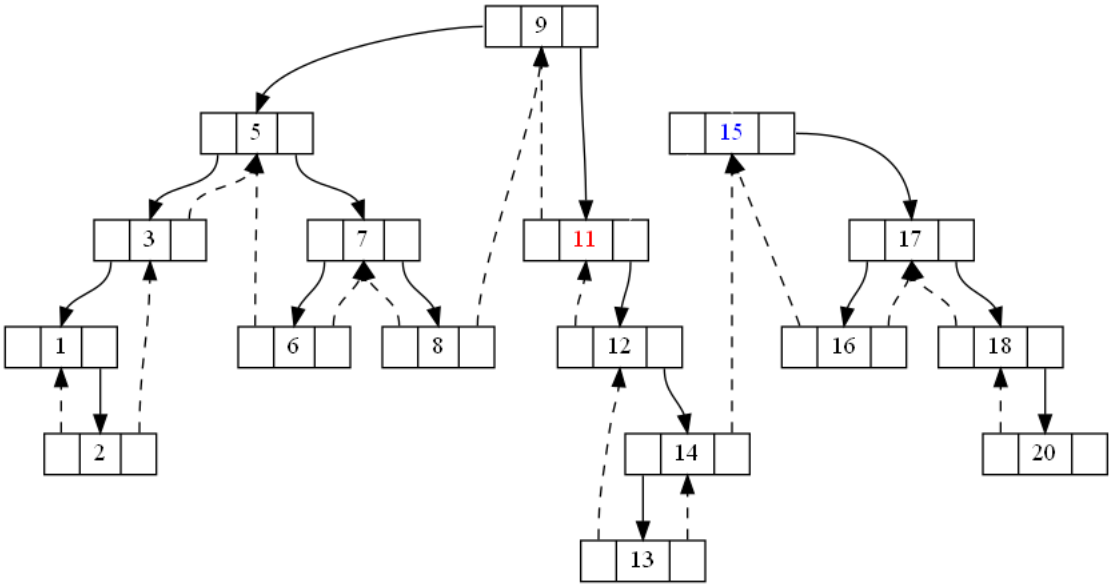


Figure 3: Example tree : moving a part from one tree to other

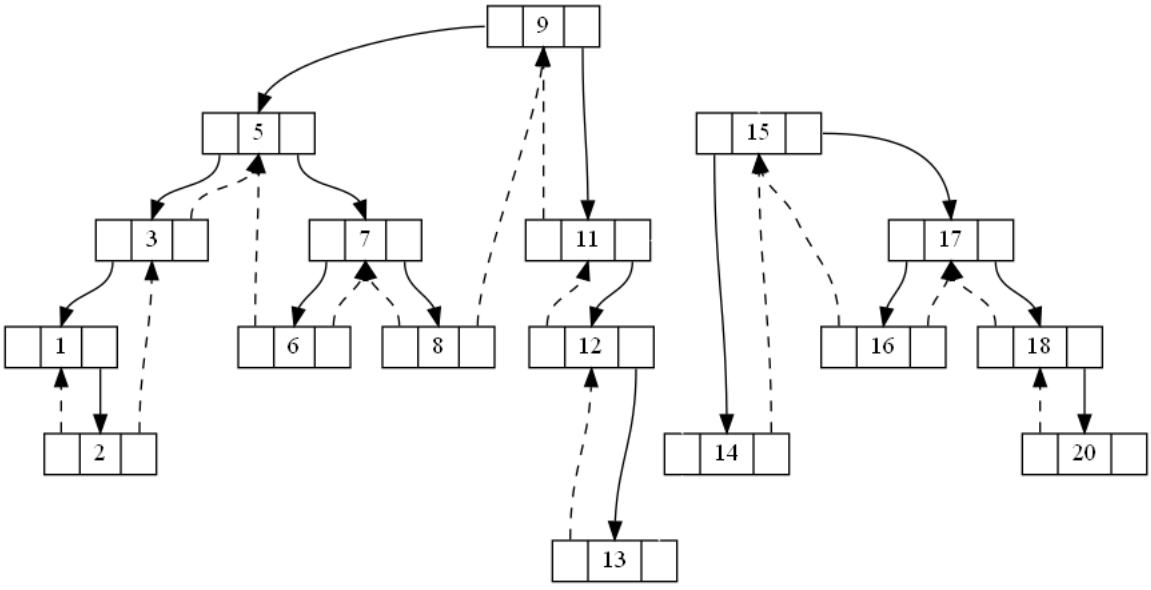


Figure 4: Example tree : two different trees splitted around k

than or equal to current value of $k_1 node$ node and $temp$ is also greater than or equal to k_1. Similarly we put slightly opposite conditions for $k_2 node$ to spot k_2.

2.5.9 int kthElement(int k)

This function finds out the k^{th} largest element from the tree. Since we are storing the subtree counts in each node, if k is greater than the $leftSubtreeCount + rightSubtreeCount + 1$ of root, we can straight away throw an exception. Otherwise we shall compare k with the right subtree count. If k is greater than $rightSubtreeCount + 1$, we can subtract this count from k and repeat this process on left subtree, else we can move right. When $rightSubtreeCount + 1$ and k become equal we return the current node data. This takes $O(h)$ time where h is the height of the tree.

2.5.10 void printTree()

This function does a level order traversal of the tree and generates two files. One is a graphviz(.gv) file and the other file depends on the operating system we are using. For windows it generates a batch file and for linux it generates a shell script.

2.5.11 void insertIntoTree(TreeNode * root, int x)

This function is a standard implementation of a recursive insert function along with proper thread and subtree count management. This runs in $O(h)$ time where h is the height of a tree.

2.5.12 void reduceSubtreeCount(TreeNode * node)

This function reduces the respective subtree count from root till the parent of a given node in $O(h)$ time where h is the height of a tree.

2.5.13 bool isLeaf(TreeNode * node)

This function returns true if the given node is a leaf otherwise returns false.

2.5.14 TreeNode * deleteLeafNode(TreeNode * node, TreeNode * parent)

This function deletes a leaf node and updates the threads of its parent. Then it returns the node which got deleted by resetting all the tree parameters.

2.5.15 TreeNode * deleteNodeWithSingleChild(TreeNode * node, TreeNode * parent)

This function replaces the parent's left or right pointer (the same pointer which points to the node to be deleted) with the child of the node to be deleted. The threads are properly managed. Then it returns the node which got deleted by resetting all the tree parameters.

2.5.16 TreeNode * leftMost(TreeNode * node)

This function finds the leftmost node of a given node in $O(h)$ where h is the height of the tree.

2.5.17 TreeNode * rightMost(TreeNode * node)

This function finds the rightmost node of a given node in $O(h)$ where h is the height of the tree.

2.5.18 TreeNode * getParent(TreeNode * node)

This function finds the parent node of a given node in $O(h)$ where h is the height of the tree.

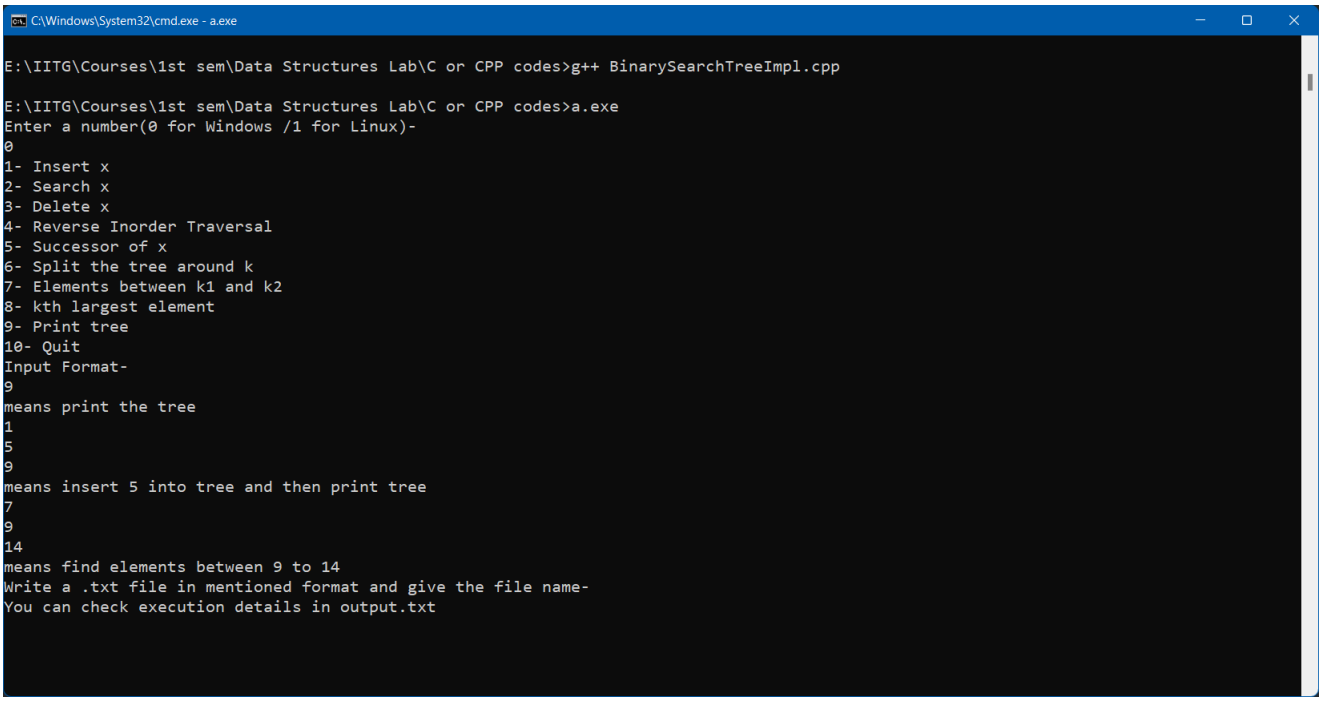


Figure 5: Sample prompt

2.5.19 TreeNode * copyNodes(TreeNode * root, unorderedmap map)

This is a recursive function called by the copy constructor which creates a replica of the given tree.

2.6 functions.h

This file contains some utility functions which are being used multiple times like printing a linked list or printing a long message.

3 How to create test cases

3.1 User prompt

After executing the program, a prompt will ask the user to enter a number indicating the current operating system. Then a prompt will be printed mentioning the sequence of operations and some small example test cases. It will ask the user to write a file name. So here the user needs to create a .txt file with a sequence of integers and save it in the same directory. Then the same file name should be given in the prompt. A sample prompt is shown in **Figure 5**. If the last instruction is not to quit the execution, after executing the current .txt file the same prompt will reappear for further execution. A new .txt file can be created and the name can be given. For quitting the execution the last instruction in the .txt file should be 10. Also, "quit" or "exit" or similar keywords can be given instead of file name to quit the execution. Currently added keywords are - quit, exit, stop, return. More can be added at line number - 64 in BinarySearchTreeImpl.cpp
After splitting the original tree, a prompt will appear asking to replace the current tree with any of the splitted trees and the user has to take the proper action.

3.2 Example testcase

1
9
1
5
1
15
1
3
1
7
1
11
1
17
9
6
10
2
15
5
4
7
9
16
10

The above sequence can be written as a test case in an input file, which indicates the following.
1 (Insert the next value)
9
1 (Insert the next value)
5
1 (Insert the next value)
15
1 (Insert the next value)
3
1 (Insert the next value)
7
1 (Insert the next value)
11
1 (Insert the next value)
17
9 (Print the tree)
6 (Split the tree around next value)
10
2 (Search the next value)
15
5 (Find successor of the last searched value)
4 (Get reverse inorder list)
7 (Find the elements between next value and its next value i.e. 9 and 16)
9

16
10 (Quit)

4 Instructions to execute the code

1. Be careful while executing the program.
2. If you're using Dev C++, Follow the steps to support *unordered_map*. Go to tools–compiler option–general tab–tick mark option (add the following commands when calling compiler)–add -std=c++11 there. Then build and execute the project.
3. In linux or GNU windows compiler, open the terminal or command prompt and type "g++ BinarySearchTreeImpl.cpp" and hit enter.
4. Then for linux type "./a.out". For windows type "a.exe" or "BinarySearchTreeImpl.exe" (One of them should work). Hit enter.
5. Now the prompt will be displayed. Enter 0 or 1 for windows or linus respectively.
6. Write a .txt file according to the given format and enter the file name there. If you give wrong sequence, it can destroy the execution.
7. If you give an option to split the tree, there will be another prompt to replace your current tree with any of the splitted ones. Give proper option. Wrong choice can lead to destruction of the execution.
8. The entire execution process can be visualized in output.txt.
9. After splitting or printing a tree, you can view the images in the same directory. The file names are written in output.txt.
10. You can give multiple .txt files one after another. For example- in the first run, you can insert a few nodes and find kth largest element. It will print the result and ask you to give another file name. You can now split the tree around the kth largest element and may be discard the higher half and replace the original root with the lower half and then continue execution.
11. 10 is used for quit. Do not write 10 unnecessarily inside the .txt file. It is better to keep it at the end.
12. You can run the batch file or shell script to manually generate the images.

5 Testing

The input files, output files, graphviz files, png files, console sequence everything is attached. Those files can be referred for better understanding. The sequence is-

5.1 input1.txt

1. Insert $18, 10, 30, 6, 14, 22, 34, 2, 12, 16, 24, 32, 36, 4, 28,$
   $60, 26, 38, 50, 44, 42, 46, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 20, 25$ in that order.
2. Print the current tree.
3. Delete 25.

4. Print the current tree.
5. Delete 60.
6. Print the current tree.
7. Delete 18.
8. Print the current tree.
9. Delete 30.
10. Print the current tree.
11. Print reverse inorder.
12. Search for 46.
13. Get its successor.
14. Search for 50.
15. Get its successor.
16. Find 9^{th} largest element.

It prints the 9^{th} largest element which is 28 in this case. Then according to that input2.txt got created.

5.2 input2.txt

1. Split around 28.

Then it displays a prompt asking whether to replace the root. Let's replace the original root with the tree having values less than or equal to 28 i.e. enter 1.

5.3 input3.txt

1. Print the current tree.
2. Find nodes between 8 to 25.
3. Find 15^{th} largest number.

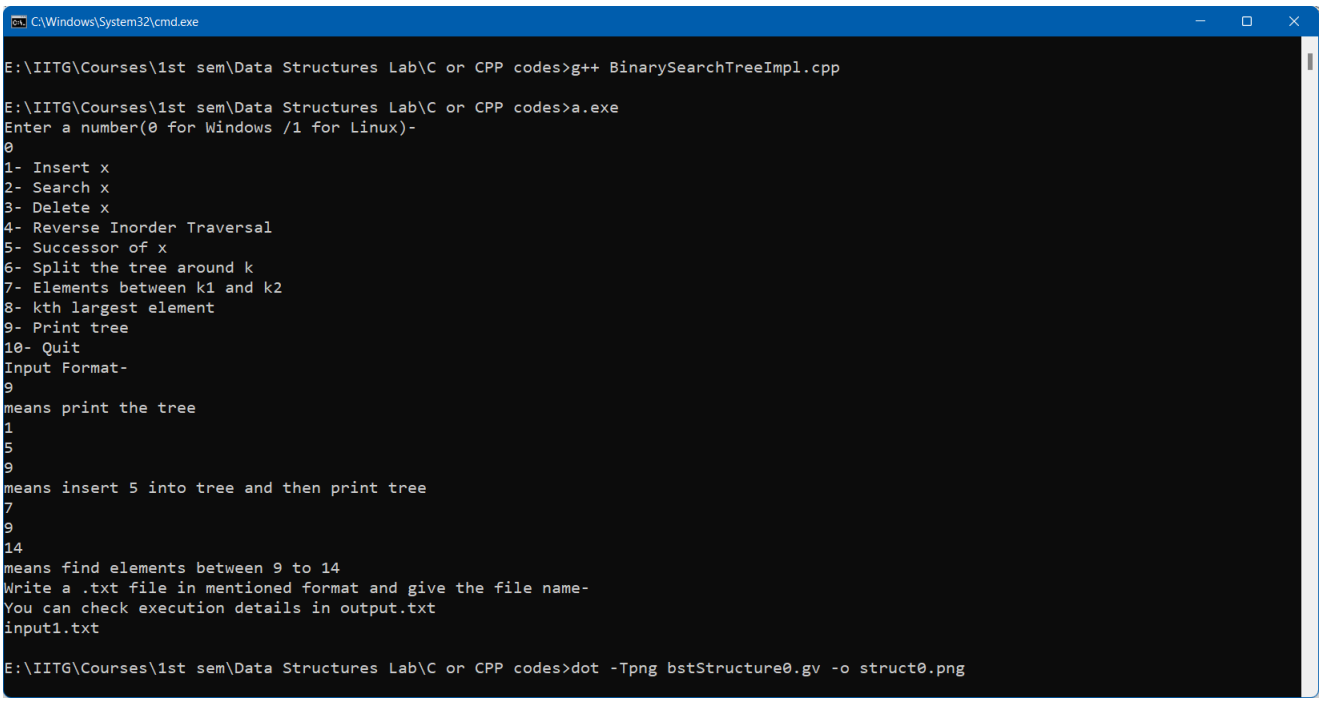
It prints the 15^{th} largest element which is 10 in this case. Then according to that input4.txt got created.

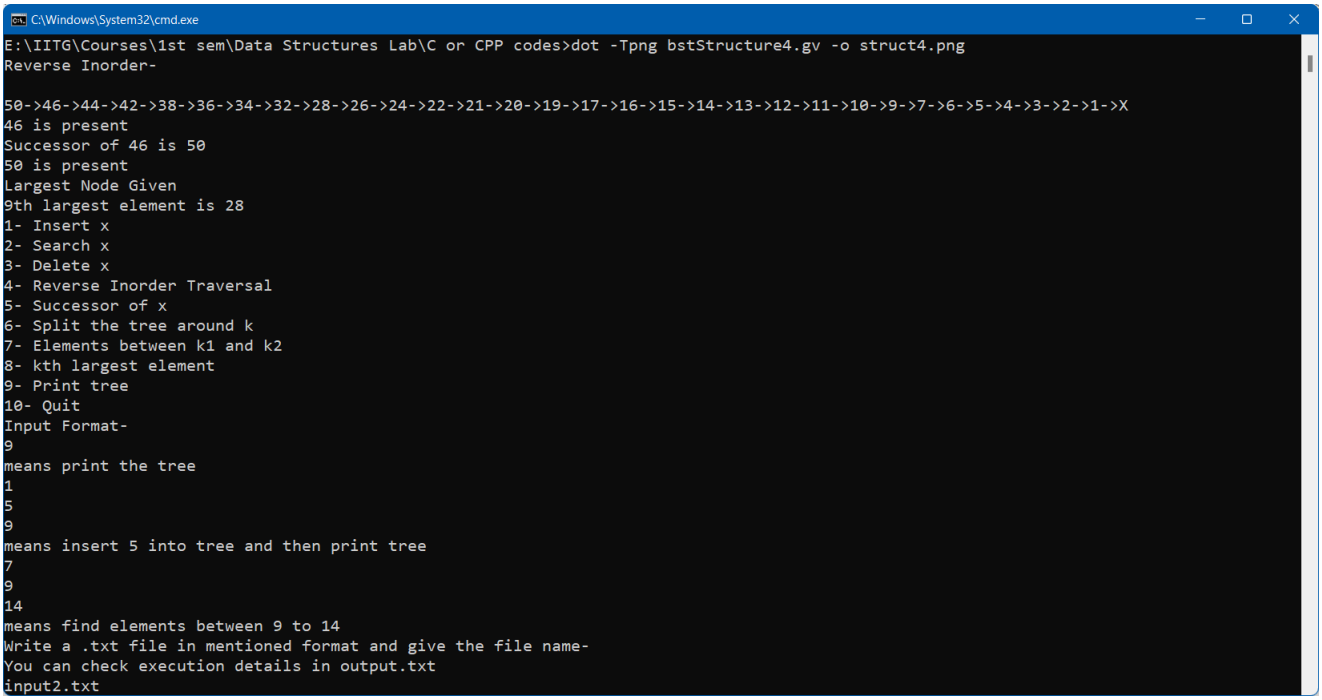
5.4 input4.txt

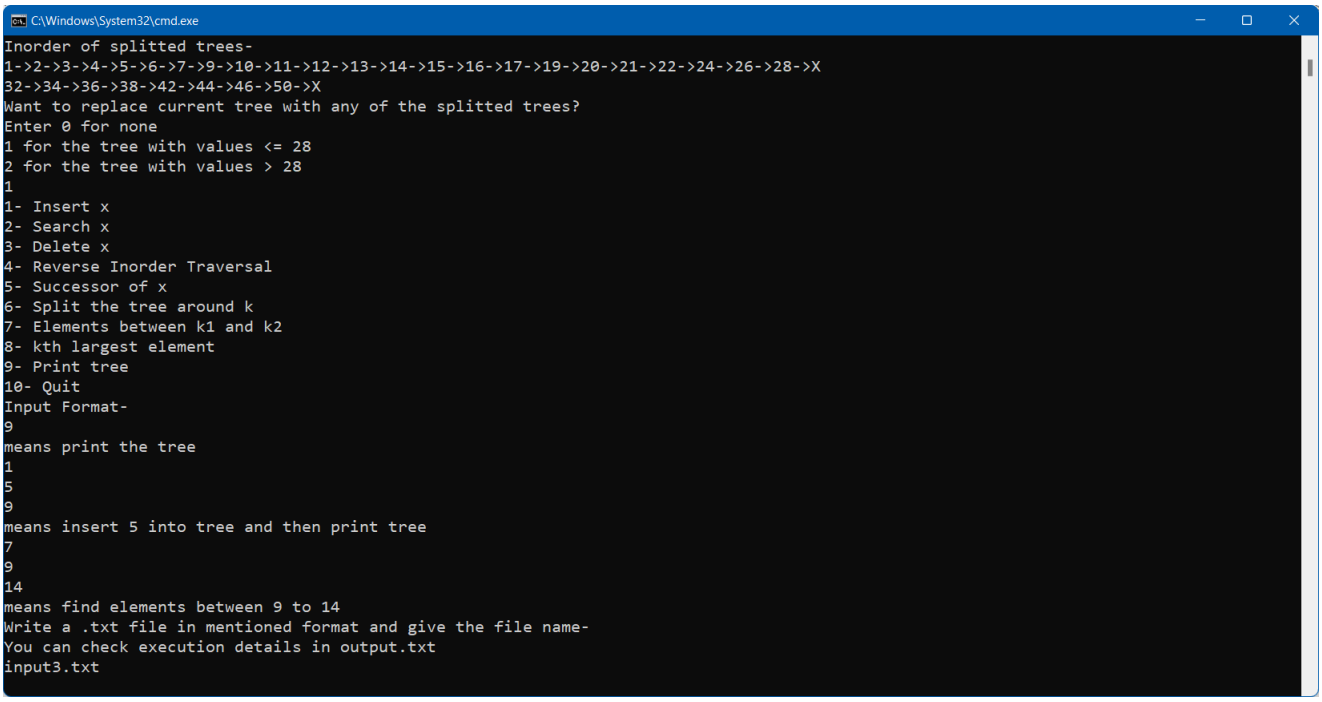
1. Split around 10.
2. Quit.

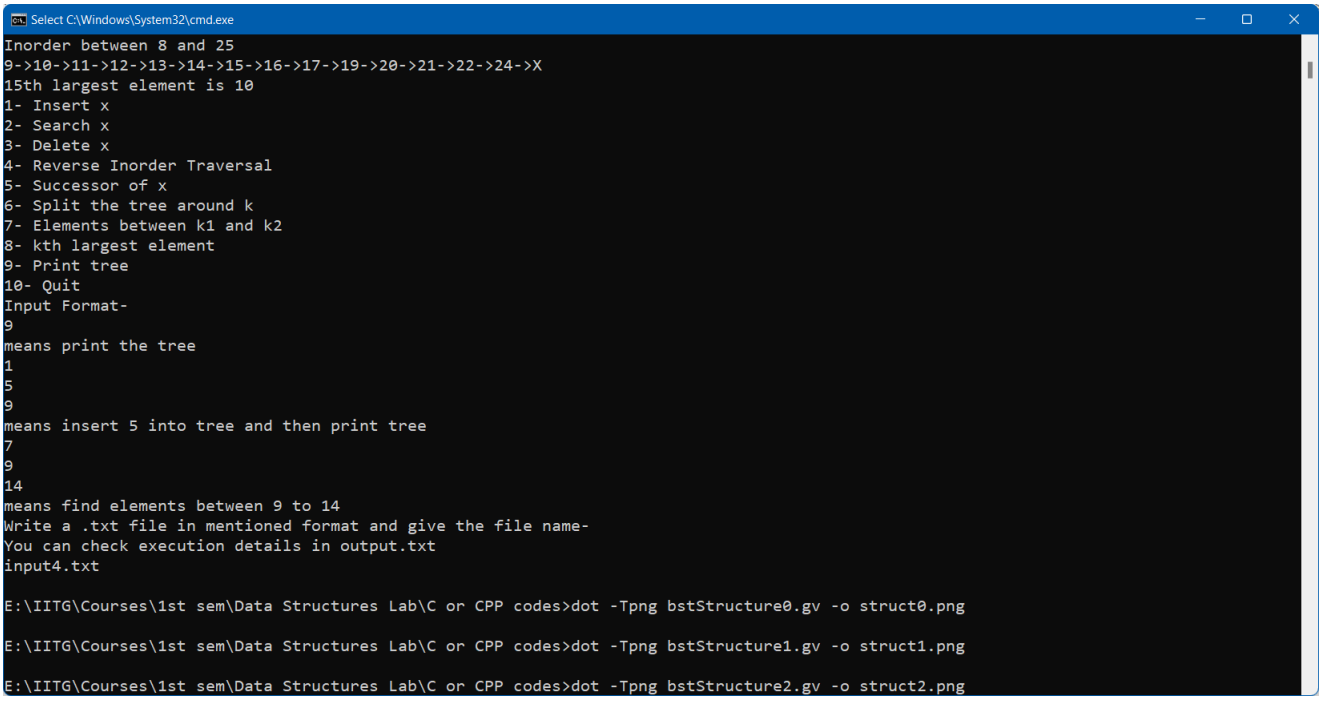
Entering 0 in prompt this time not to replace the root.

5.5 Console sequence





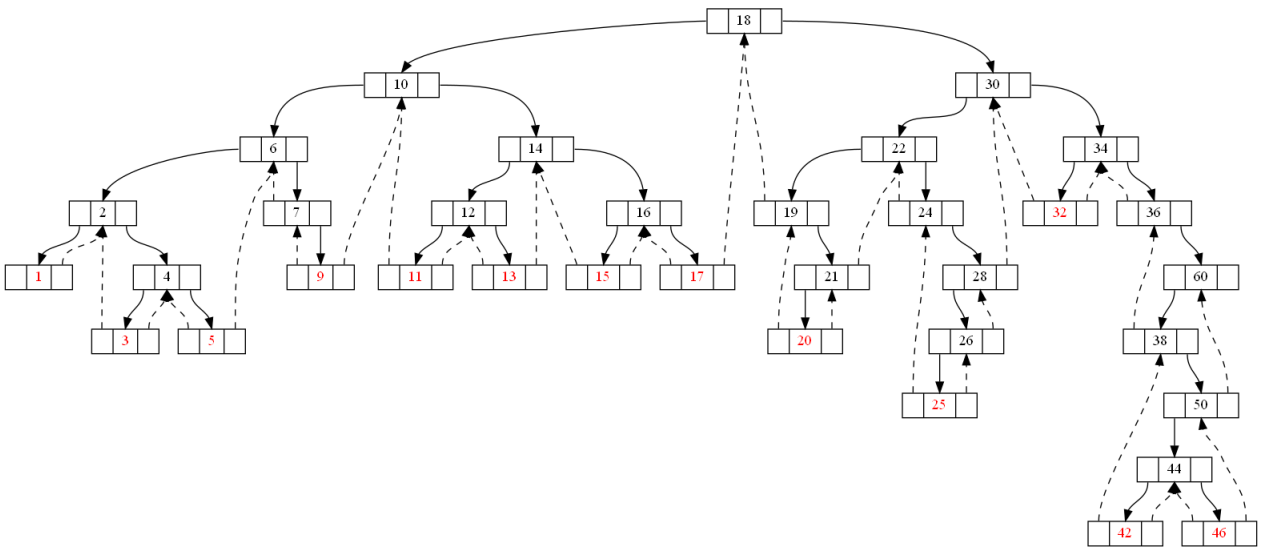

C:\Windows\System32\cmd.exe
Inorder of splitted trees-
1->2->3->4->5->6->7->9->10->11->12->13->14->15->16->17->19->20->21->22->24->26->28->X
32->34->36->38->42->44->46->50->X
Want to replace current tree with any of the splitted trees?
Enter 0 for none
1 for the tree with values <= 28
2 for the tree with values > 28
1
1- Insert x
2- Search x
3- Delete x
4- Reverse Inorder Traversal
5- Successor of x
6- Split the tree around k
7- Elements between k1 and k2
8- kth largest element
9- Print tree
10- Quit
Input Format-
9
means print the tree
1
5
9
means insert 5 into tree and then print tree
7
9
14
means find elements between 9 to 14
Write a .txt file in mentioned format and give the file name-
You can check execution details in output.txt
input3.txt


Select C:\Windows\System32\cmd.exe
Inorder between 8 and 25
9->10->11->12->13->14->15->16->17->19->20->21->22->24->X
15th largest element is 10
1- Insert x
2- Search x
3- Delete x
4- Reverse Inorder Traversal
5- Successor of x
6- Split the tree around k
7- Elements between k1 and k2
8- kth largest element
9- Print tree
10- Quit
Input Format-
9
means print the tree
1
5
9
means insert 5 into tree and then print tree
7
9
14
means find elements between 9 to 14
Write a .txt file in mentioned format and give the file name-
You can check execution details in output.txt
input4.txt
E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure0.gv -o struct0.png
E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure1.gv -o struct1.png
E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure2.gv -o struct2.png

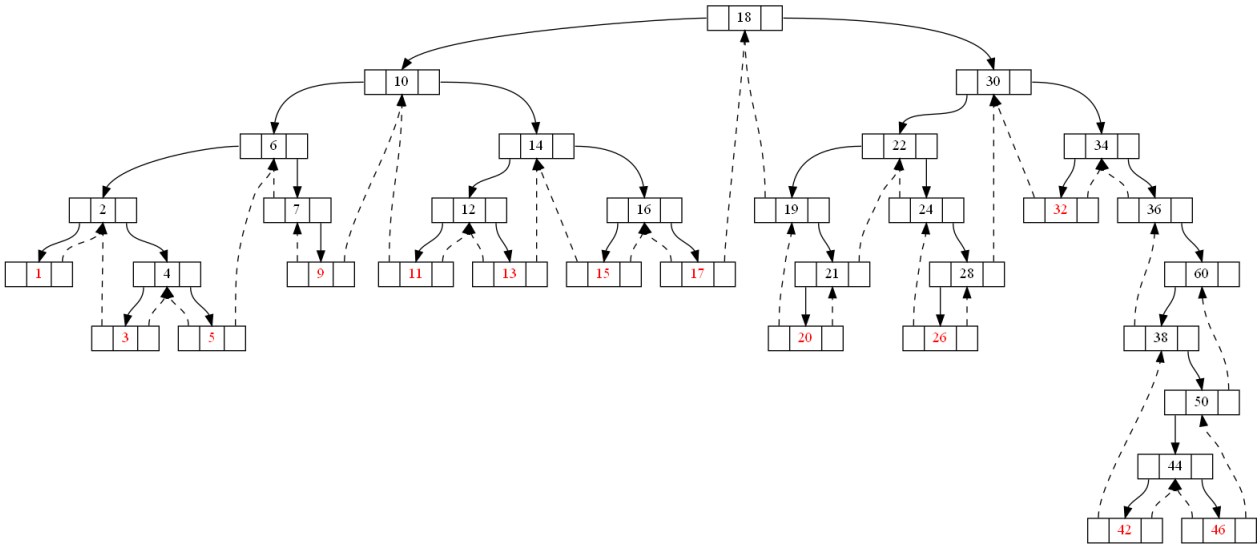
```
Select C:\Windows\System32\cmd.exe

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure1.gv -o struct1.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure2.gv -o struct2.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure3.gv -o struct3.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure4.gv -o struct4.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure5.gv -o struct5.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure6.gv -o struct6.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure7.gv -o struct7.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure8.gv -o struct8.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>dot -Tpng bstStructure9.gv -o struct9.png
Inorder of splitted trees-
11->12->13->14->15->16->17->19->20->21->22->24->26->28->X
1->2->3->4->5->6->7->9->10->X
Want to replace current tree with any of the splitted trees?
Enter 0 for none
1 for the tree with values <= 10
2 for the tree with values > 10
0
Quit

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes>
```

5.6 Trees sequence

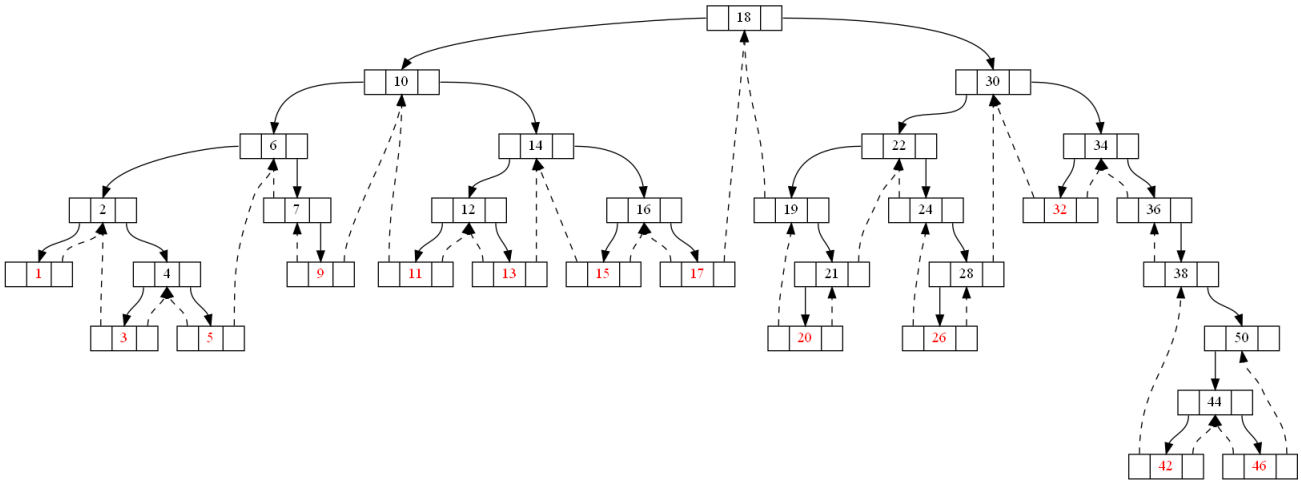
5.6.1 Initial tree



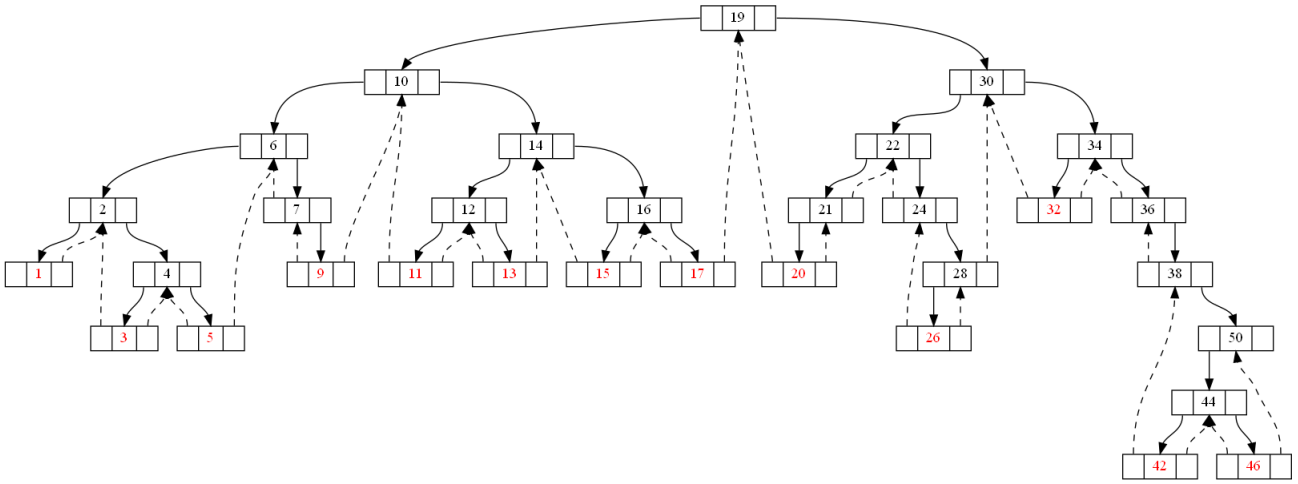
5.6.2 After deleting 25



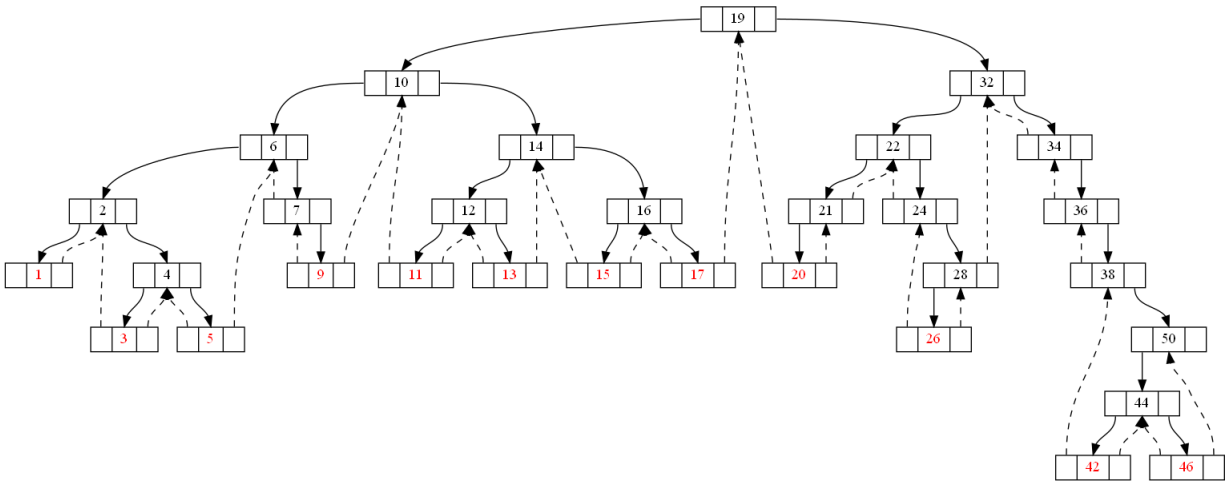
5.6.3 After deleting 60



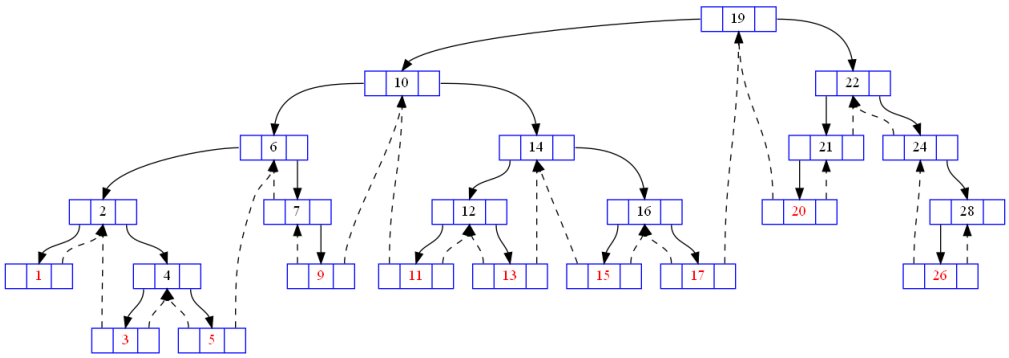
5.6.4 After deleting 18

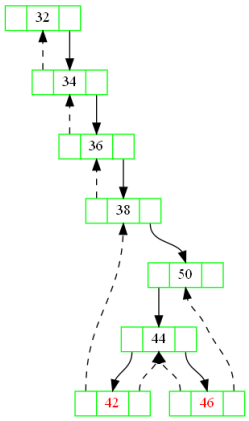


5.6.5 After deleting 30

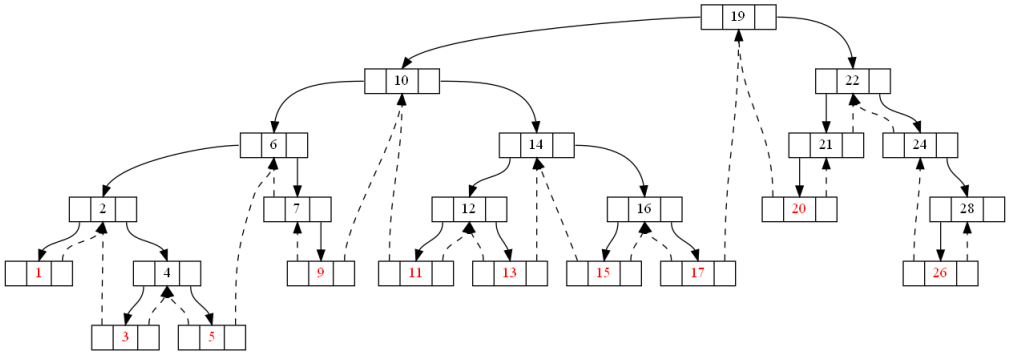


5.6.6 After splitting around 28





5.6.7 After replacing with tree having values ≤ 28



5.6.8 After splitting around 10