Assignment 5 Technical Report

2171039 Chaewon Lee

Code 2

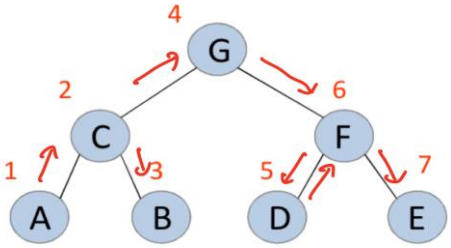
This is the code that finds a successor of node. This is implemented by the pseudo code given in the homework file.

tree_successor function gets node as an argument and returns the successor of that node. If the given node has a right child, it means that the leftmost node of its right sub-tree would be the successor of that node. To do that, the code keeps moving to the left until the left node is null. And it finally returns parent of that null node as a successor.

If the given node does not have a right child, this means that the function has to go up while finding the successor. And the successor of that node would be the most top node stretching out to the left. Therefore, when the given node is the 'right' child of the parent node, it stops moving and return that parent node as a result.

In main function, first, we must declare the parent of each node, since the TreeNode structure has a parent field. And then, because we are going to see the successor of inorder traversal, we must go to the leftmost node of the tree first. Then, it prints the current node, find the successor of the node, then set the current node as a successor. This stops when reaches the end of the node, in other words, the successor of the node is null.

- **Result**



The result of the code is same as the result of inorder traversal.

Code 3

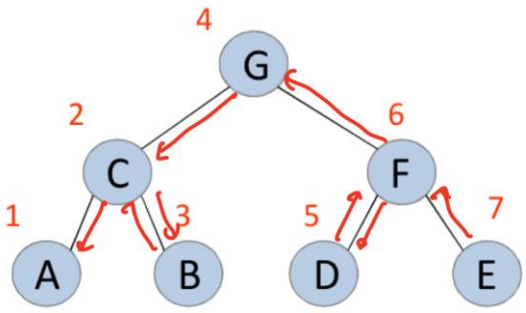
This is the code that finds a predecessor of node. The pseudo code of this process is like this.

```
Tree - predecessor (x)
{ if x→left != NULL
  { TreeNode q=x→left
    while(q→right != NULL)
      q=q→right
    return q;
  }
  y = x→parent
  while(y != NULL and x==y→left){
    x=y
    y=y→parent
  }
  return x.
```

There are two cases: if the node has a left child or not. If the node has a left child, then the predecessor of the node would be the rightmost node of left sub-tree. Therefore, until there is no right child, the code keeps moving right and returns the rightmost node. And if it does not have a left child, then the most top node stretches out to the right would be the predecessor. So, until the node is left child of parent node, it keeps going up, and return the parent node.

The code has exactly same algorithm as the pseudo code above. In tree_predecessor function, the code finds the predecessor of given node and returns it. Then, in main function, first it declares the parent of each node, then move to the rightmost of the tree, since this code is going to keep printing the predecessor of the node. Finally, the code prints the node, finds the predecessor of the node, set the node as the predecessor. When it is the beginning of the tree, in other words, the predecessor is null, then printing stops.

- **Result**



The result of the code is same as the reversed result of inorder traversal.

Code 4

This is the revised version of bst_insertion_deletion.cpp. The difference between the original code is that in deletion method when the node selected to be deleted has both child nodes. The original code finds the node's successor, but this code finds the node's predecessor. Other parts of the code are exactly same as the original code.

The algorithm to find the predecessor of the node is same as code 3. It tries to find the rightmost node of the left sub-tree. In the moment that the code finds the rightmost node, it means that the rightmost node does not have any right child. So, we have to switch the parent node of successor with the left child of successor. There would be two cases: successor is the left child of its parent, or not. We set that area that previously occupied by the successor as the successor's left child.

Main function first shows the result of inorder traversal of original tree. Then, the node is deleted by delete_node function, and the code traverses in inorder again.

- **Result**

```
Binary tree
3       7       10      12      18      22      24      26      30      35      68      99

Binary tree
3       7       10      12      22      24      26      30      35      68      99
```

```
Binary tree
3       7       10      12      18      22      24      26      30      35      68      99

Binary tree
3       10      12      18      22      24      26      30      35      68      99
```

```
Binary tree
3       7       10      12      18      22      24      26      30      35      68      99

Binary tree
3       7       10      12      18      22      24      26      30      68      99
```

You can see the code is working well in every key cases.