

Project 5: Binary Search Trees

CSE3333

The University of Texas Rio Grande Valley

Spring 2018

Dr. Liyu Zhang

Juan Bermudez

November 28, 2018

The program was compiled using Visual Studios. The project is menu driven, and in order to execute a BST operation, the nodes need to be inserted first using choice 1. Proceed in testing the BST operations by going through the menu choices. Some difficulties were found with the Sub-BST function and the EnhancedSearch function. The Sub-BST function is working, however it needs both leaves and the parent while performing the search. If a parent node is missing a child, then the program will crash and end. The EnhancedSearch function is incomplete, however an explanation will be provided on the resulting code. For these two functions the instructions will be explained below on the testing part.

The running times for the BST functions were not implemented in this program. However, the table below will describe the theoretical running times for the main functions used. It should also be taken into consideration that for this program an array was used instead of a linked list, which may make the running time slower for some functions.

n = number of nodes, the nodes are proportional to the height of the tree based on the cost of traversal.

Function	Average Case	Best Case	Worst Case
Insert	For Best and Avg Case, if tree is balanced, $O(\log n)$		$O(n)$
Inorder	Running time is $O(n)$ because every node is visited atleast once		
Postorder			
Preorder			
SearchMaximum	Running times are $O(n)$ where n is proportional to the height of the tree		
SearchMinimum			
Search	Avg and Best case is $O(n \log n)$ for a balanced tree ($n \log n$ for n searches, 1 search is $\log n$)		$O(n^2)$ tree is not balanced
Predecessor	Running times are $O(n)$ where n is proportional to the height of the tree		
Successor			
Delete	For Best and Avg Case, if tree is balanced, $O(\log n)$		$O(n)$

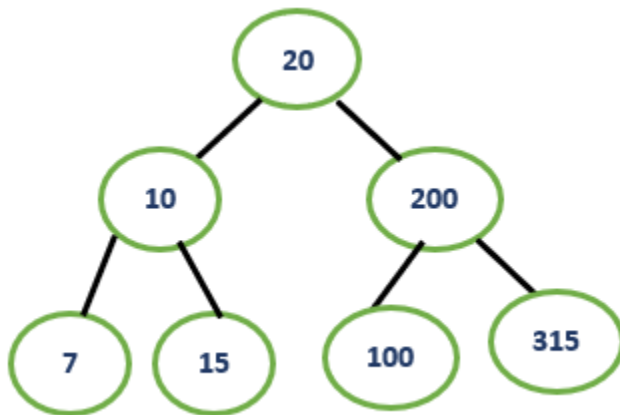
Test runs

The tests were conducted using 7 nodes. The tests were kept small to depict that the graphs were behaving accordingly with the BST functions.

Case 1 will display the Insertion of nodes into the tree:

1. Insertion:

For this test, the following tree is constructed



In the Insertion operation, the keys 20, 10, 200, 7, 15, 100, and 315 will be inserted and they will be organized according to BST behavior. 20 will be the root node. The left subtree will contain nodes smaller than the root, and the right subtree will contain nodes greater than the root. The code also does not allow for nodes to be identical, if a node is identical, then it is ignored.

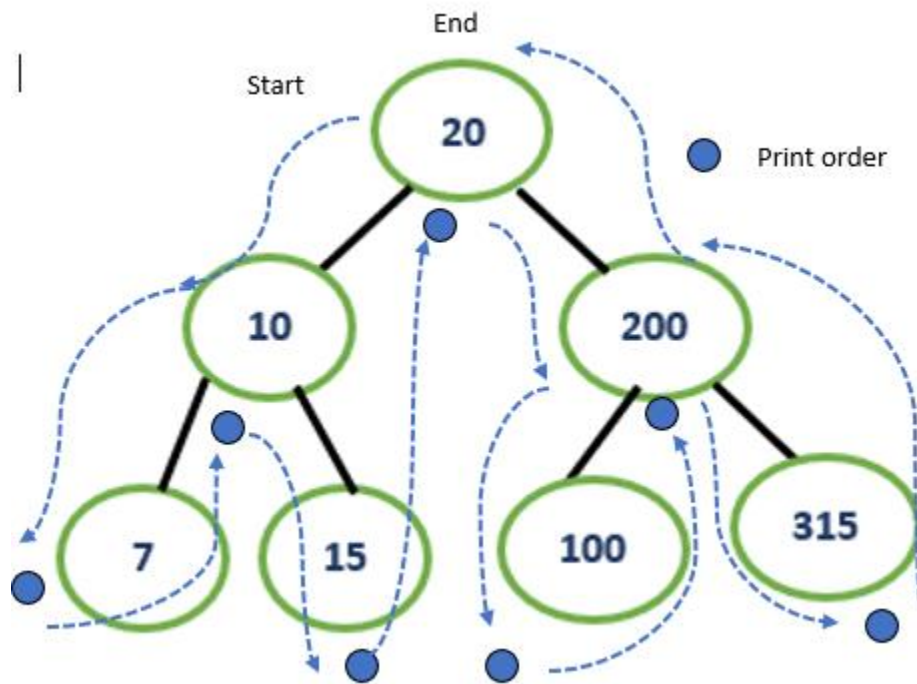
The Insertion function is tested like so:

```
Enter a choice:
1
Insert values into BST
20
10
200
7
15
100
315
```

Case 2 will display the Inorder traversal of the tree:

The Inorder function will simply follow the following sequence for any node: traverse the left subtree, visit the node, then traverse the right subtree. If at any point a node is null, then the traversal will continue with the next operation.

Inorder traversal is 7, 10, 15, 20, 100, 200, 315:



Thus, the Inorder function should print the nodes of the tree from smallest to largest, with the root node as the middle element, as shown below:

Similarly, the output for the Inorder function is shown below:

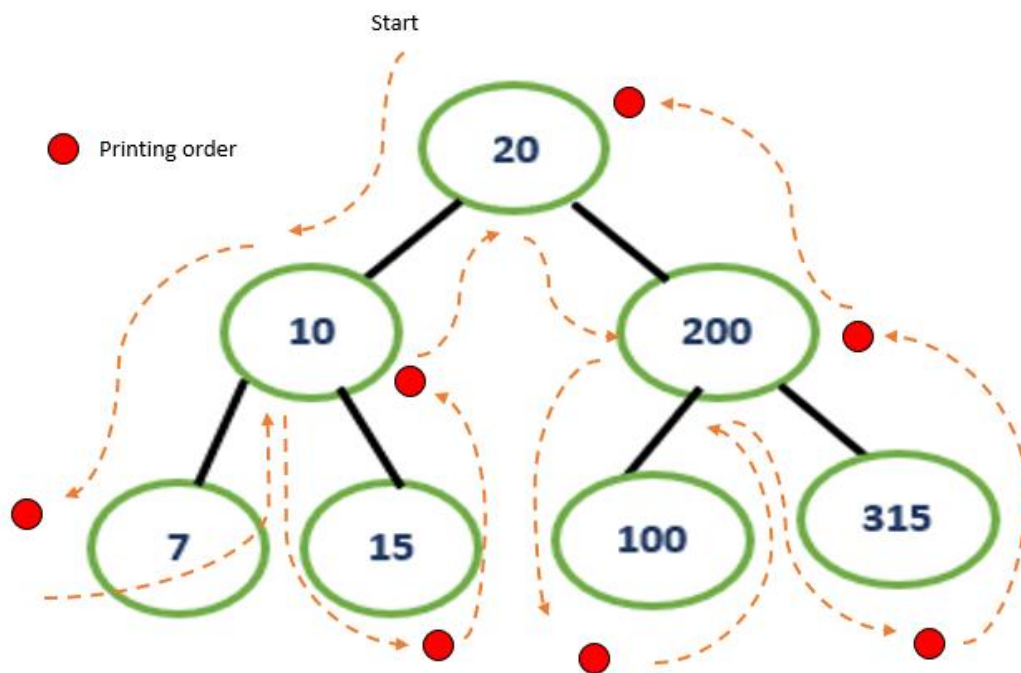
```
Enter a choice:
2

Inorder display
7 10 15 20 100 200 315
```

Case 3 will display the Postorder traversal of the tree:

The Postorder traversal will follow this sequence for any node: traverse the left subtree, traverse the right subtree, and then visit the node. If at any point a node is null, then the traversal will continue with the next operation.

Postorder Traversal is 7, 15, 10, 100, 315, 200, 20:



Similarly, the output for the Postorder traversal is shown below:

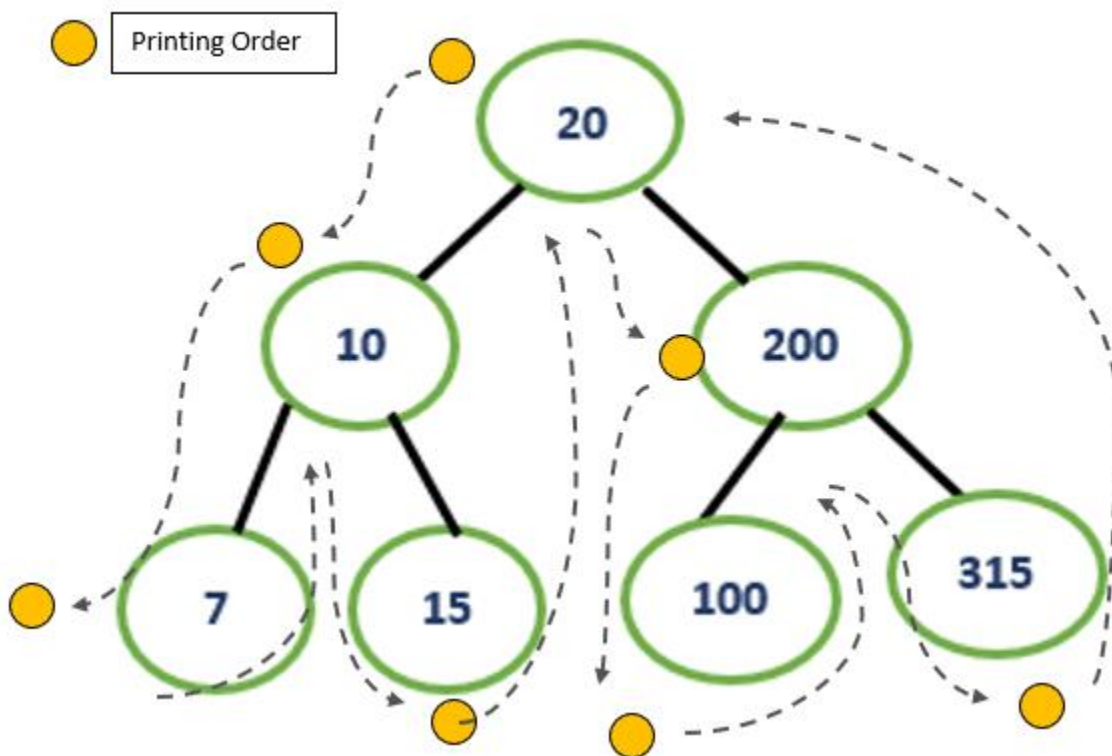
```
Enter a choice:
3

Postorder display
7 15 10 100 315 200 20
```

Case 4 will display the Preorder traversal of the tree:

The Preorder will follow the sequence for any node: Visit the node, traverse it's left subtree, and then traverse the right subtree. If at any point a node is null, then the traversal will continue with the next operation.

Preorder traversal is 20, 10, 7, 15, 200, 100, 315:



Similarly the output for the Preorder function is shown below:

```
Enter a choice:
4

Preorder display
20 10 7 15 200 100 315
```

Case 5 will display the Search Maximum function:

The Search Maximum function will traverse the right side of a tree until it reaches the maximum node, as shown in the example above, the greatest node is 315:

```
Enter a choice:
5

Search Maximum
Maximum element is 315
```

Case 6 will display the Search Minimum function:

The Search Minimum function will traverse the left side of a tree until it reaches the minimum node, as shown in the example above, the smallest node is 7:

```
Enter a choice:
6

Search Minimum
Minimum is 7
```

Case 7 will display the Search function:

The Search function will compare the key being searched for with the root node. If the key is greater than the root, the search will start at the right subtree. If the key is smaller than the root,

then the left subtree will be searched.

Search left subtree for 10, found:

```
Enter a choice:
7

Enter element to Search in BST
10

The Element 10 was Found
```

Search right subtree for 100, found:

```
Enter a choice:
7

Enter element to Search in BST
100

The Element 100 was Found
```

The key searched for,3, is not in the tree displays, not found:

```
Enter element to Search in BST
3

The element was not found
```

Case 8 will display the Predecessor function:

The Predecessor function uses inorder traversal to find a preceding value. A predecessor is the maximum value of a node's left subtree.

The predecessor of node can be proved by using the inorder arrangement. For example, 20 is behind 15, so its predecessor is 15, and 315's predecessor is 200, because it is behind it. Nothing is behind 7, so it does not have a predecessor.

7, 10, 15, 20, 100, 200, 315

The Predecessor function is tested below:

```
Predecessors of BST
The Predecessor of node 20 is 15
The Predecessor of node 10 is 7
The Predecessor of node 200 is 100
The Predecessor doesn't exists for node 7
The Predecessor of node 15 is 10
The Predecessor of node 100 is 20
The Predecessor of node 315 is 200
```

Case 9 will display the Successor function:

The successor of a node is the minimum value of a nodes right subtree. Similarly like the predecessor function, an inorder function can show proof of the successor of a node.

7, 10, 15, 20, 100, 200, 315

The successor of a node as shown below, will be the node after it. 315 will not have a predecessor because there is nothing after it.

```
Successors of BST
Successor of node 20 is 100
Successor of node 10 is 15
Successor of node 200 is 315
Successor of node 7 is 10
Successor of node 15 is 20
Successor of node 100 is 200
The Successor doesn't exists for node 315
```

Case 10 will display the delete function:

The Delete function removes a node from the tree. Here we remove 315 and 10, then display the remaining nodes inorder. If a node is not in the tree, then a display will reveal “node is

```
Enter a node to Delete
315

After deletion, the BST is:
7 10 15 20 100 200
BST Functions
```

```
Enter a node to Delete
10

After deletion, the BST is:
7 15 20 100 200
BST Functions
```

```
Enter a node to Delete
1
Element Not Found
After deletion, the BST is:
```

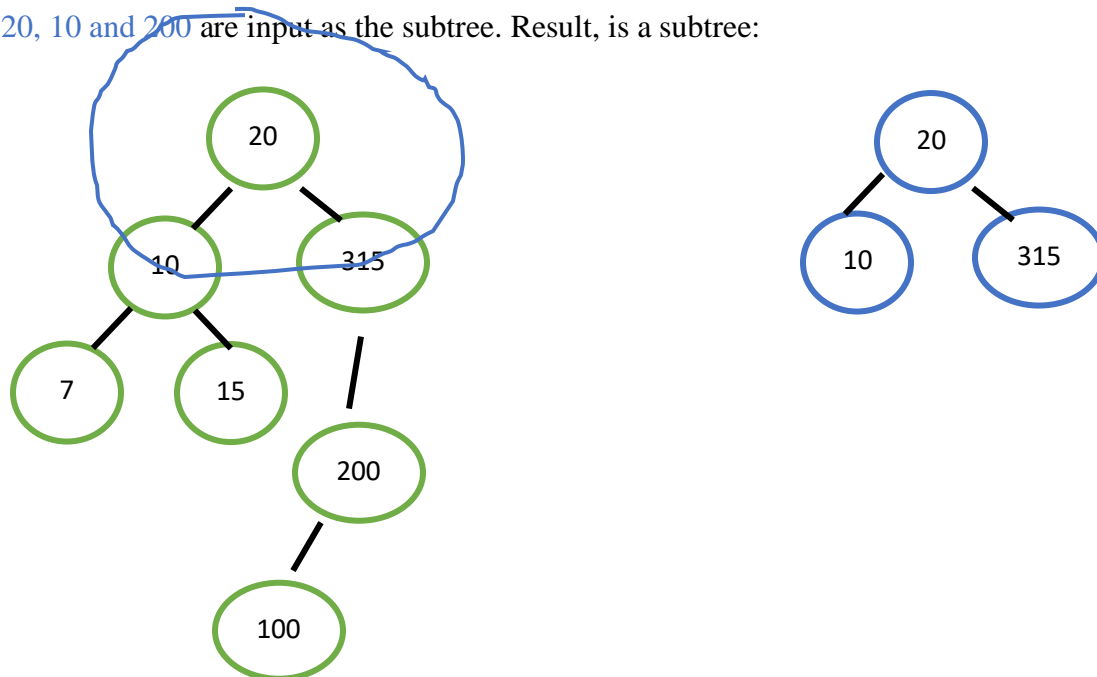
Case 11 will display the Sub-BST function:

For this case, the user will input 3 nodes into a subtree, and then the 3 nodes will be compared with the tree that was currently inserted in Case 1. This function will crash if a parent is missing a child, which is why it needs both childs. Regardless, the code will compare the left and right sides of the tree to verify if the inputted nodes are a subtree.

Example 1:

checks the head of a tree:

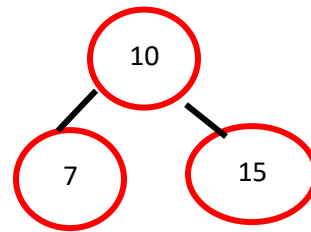
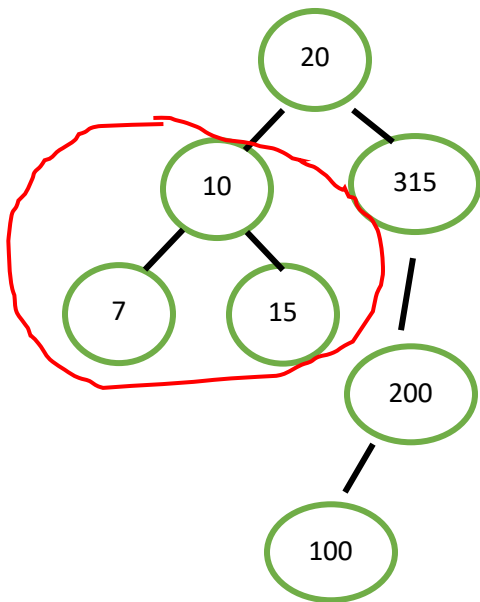
20, 10 and 200 are input as the subtree. Result, is a subtree:



```
Test if a BST is a Sub-BST of another BST
Insert values into T
20
10
315
Yes T is a subtree of Tree
```

Checks left side of the tree:

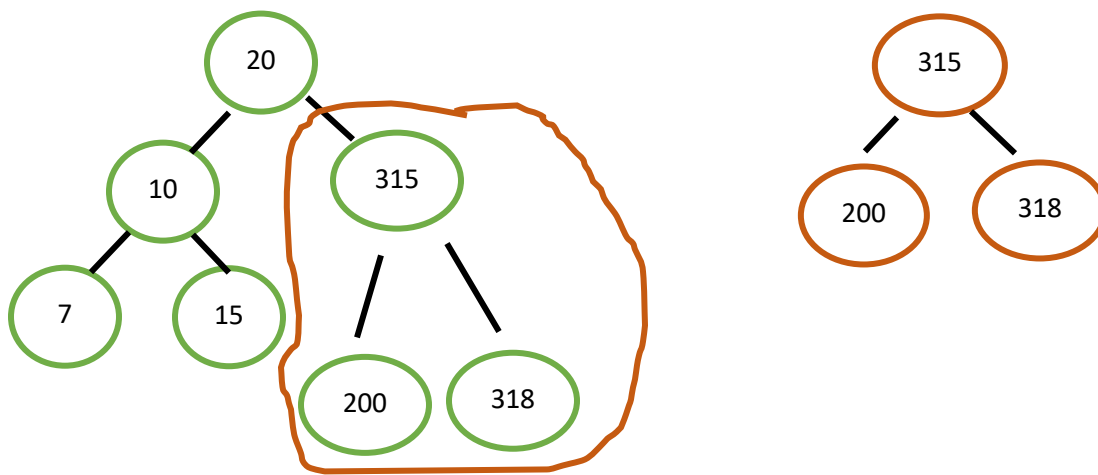
10, 7 and 15 are input as the subtree. Result, is a subtree:



```
Test if a BST is a Sub-BST of another BST
Insert values into T
10
7
15
Yes Yes T is a subtree of Tree
```

Checks right side of the tree:

If we input 315, 200, and 100, then the program will crash . However, if we add values to the right side which have parent and both its childs, then the result will be a subtree. The graph below is changed to have both its childs. **315, 318, and 200** are input as the subtree. Result, is a subtree:



```
Test if a BST is a Sub-BST of another BST
Insert values into T
315
318
200
Yes Yes T is a subtree of Tree
```

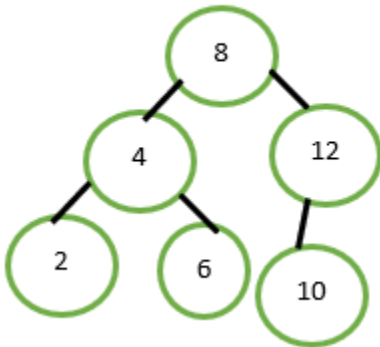
Adding nodes that are not in the tree will display, not a subtree:

```
Test if a BST is a Sub-BST of another BST
Insert values into T
1
2
3
No T is not a subtree of Tree
```

Case 12 will display the Enhanced Search function:

The code for the EnhancedSearch function is incomplete. However, the attempt was as follows by using the provided example:

1. Assume the give tree is inorder



2. Subtract the left subtree's first three nodes from the search key number:

Search for key 6 on inorder tree 2, 4, 6, 8, 10, 12 by subtracting from the first 3 nodes

$$\text{min1} = 6 - 2 = 4$$

$$\text{min2} = 6 - 4 = 2$$

$$\text{min3} = 6 - 6 = 0$$

min1, min2, and min3 are the minimal distance from the provided tree.

3. Subtract the right subtree's first three nodes from the search key number:







$$\text{min4} = 6 - 8 = \text{abs}(2)$$

$$\text{min5} = 6 - 12 = \text{abs}(6)$$

$$\text{min6} = 6 - 10 = \text{abs}(4)$$

4. Compare the distances from the left subtree and the right subtree:

The values will be compared with the smallest distance of the left subtree, in this case the distances will be compared with 4, if a distance is greater or equal to 4, then it cannot be considered as a pointer of the searched key 6.

Nodes	Distance (min)	Considered?
2	4	
4	2	
6	0	
8	2	
12	6	
10	4	

5. The pointers to be returned that are closest to search key 6 with the enhanced search will then be 4,6, and 8 as shown on the problem.

The result for the code I produced only returned the subtractions of the keys in stage 3 above, however, stage 4 which compared the results was not able to execute.

```
Enhanced Search of a BST
Insert values into Enhanced Tree
8
4
12
2
6
10
Enter element to Search in the BST, and return 3 values close to it:
6
2 4 6 -2 10 12
```

Conclusion:

The main BST functions show that the output behavior is correct and it simulates a tree that holds data. It is also safe to assume that the running times are running accordingly. The BSTs in this program also were tested using small trees to make sure the outputs were accurate. Larger BSTs can be constructed by increasing the size of the keys[] array, and the behaviors of the functions will still behave accordingly.