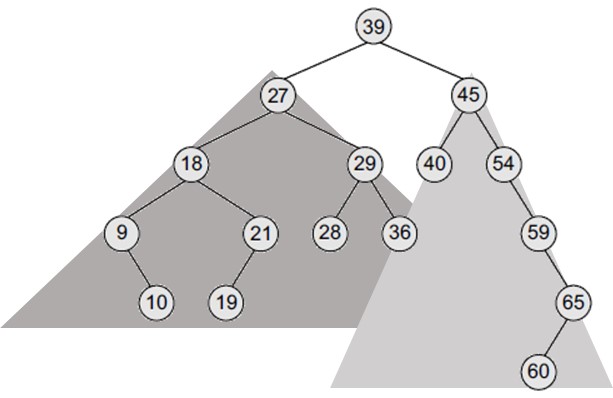
# Lab Tutorial for Week 7: BST (Binary Search Tree) and AVL Tree Implementation

## Contents

1. Tutorial w7a: Linked BST
2. Tutorial w7b: Linked AVL Tree (optional)

## Tutorial w7a: Linked BST

### A binary search tree (BST) is binary trees in which the nodes are arranged in an order. In a BST, all the nodes in the left sub-tree have a value less than that of the root node, and all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.



The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the root node. The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65.

Recursively, each of the sub-trees also obeys the binary search tree constraint. For example, in the left sub-tree of the root node, 27 is the root and all elements in its left sub-tree (9, 10, 18, 19, 21) are smaller than 27, while all nodes in its right sub-tree (28, 29, and 36) are greater than the root node’s value.

Since each node will have maximum two children (successors), the structure for the node is like below:

struct node

{

int data;

node \*left;

node \*right;

};

The key operations for a BST include insert a new node, traversal (pre-, in- and post-order), delete a node and delete the tree. In the class BST below, the pointer to the node is made public to allow recursive call for traversal and deletion functions.

class BST

{

public:

node \*root; // pointer to the root node

public:

// constructor

BST();

// key operations for a BST

void insertElement(int);

void preorderTraversal(node \*);

void inorderTraversal(node \*);

void postorderTraversal(node \*);

void deleteElement(int);

node \*deleteTree(node \*);

// utility operations for a BST

node \*findSmallestElement(node \*);

node \*findLargestElement(node \*);

int totalNodes(node \*);

int totalExternalNodes(node \*);

int totalInternalNodes(node \*);

int Height(node \*);

};

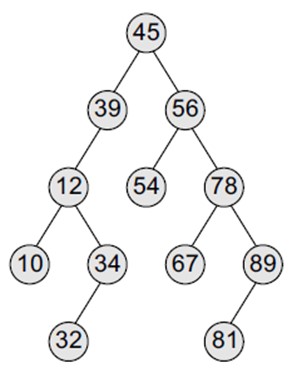
### Practice I: TESTING A BST

1. Open the starter file (LinkedBST-startercode.zip) in your IDE.
2. This file contains a linked implementation of a BST.
3. Try to understand how the code works, notably the insertion function and the recursive traversal functions.
4. Test your Tree with the following scenarios:
   1. **Create a BST**.

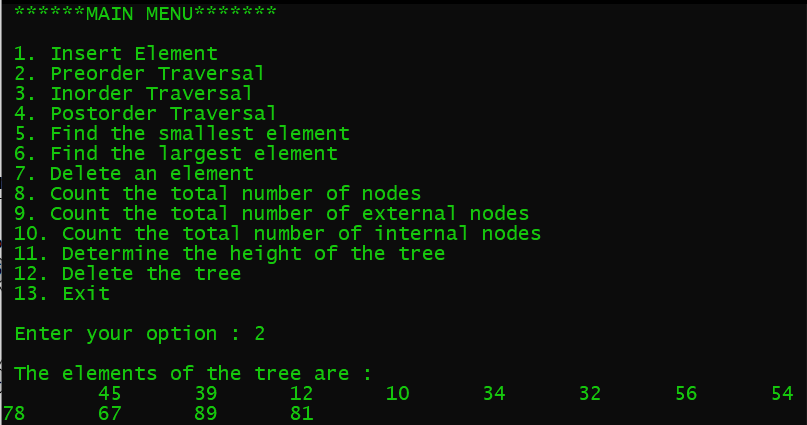
Create a binary search tree using the following data elements:

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

Make sure that you after you inserted all the data, you will have the following BST:



* 1. **Pre-order Traversal**. When you traverse pre-order, it will print the nodes sequence below:



* 1. **In-order Traversal**. When you traverse in-order, it will print the nodes sequence below:

10 12 32 34 39 45 54 56 67 78 81 89

* 1. **Post-order Traversal**. When you traverse post-order, it will print the nodes sequence below:

10 32 34 12 39 54 67 81 89 78 56 45

1. Please, test all the function with this BST and make you have these results:

|  |  |
| --- | --- |
| **Test functions** | **Result** |
| Find the smallest element | 10 |
| Find the largest element | 89 |
| Count the total number of nodes | 12 |
| Count the total number of external nodes (leaves) | 5 |
| Count the total number of internal nodes | 7 |
| Determine the height of the tree | 5 |

### Questions:

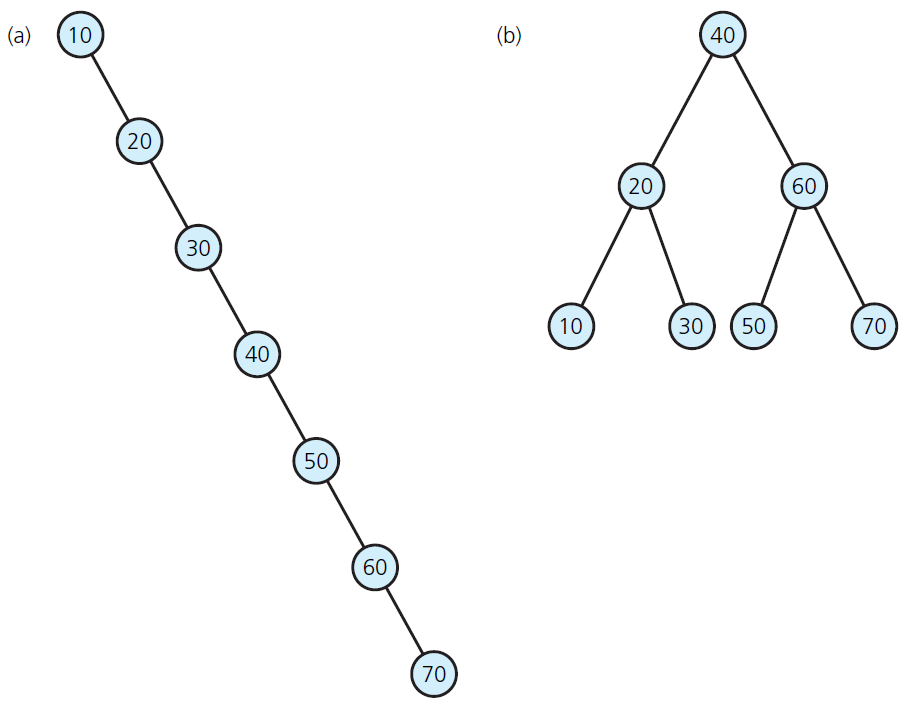
### Why height is one the most important property of BST? What does height indicate?

### What happen if you have the same number of nodes but structured/layout differently?

**Discussion**: What affects the height of a binary search tree?

The height of the tree is quite sensitive to the order in which you insert or remove items.

* For example, consider a binary search tree that contains the items 10, 20, 30, 40, 50, 60, and 70. If you inserted the items into the tree in ascending order, you would obtain a binary search tree of maximum height (a). This is called a Skewed Tree.
* If, on the other hand, you inserted the items in the order 40, 20, 60, 10, 30, 50, 70, you would obtain a balanced binary search tree of minimum height (b)



### Practice II: IMPLEMENTING A SEARCH FUNCTION FOR A BST

1. Just now, you may have realized that when you have skewed tree, searching a BST in very inefficient. It’s just like linear searching, right?
2. Now, your task is to add a search function for the BST (searchElement). Its prototype is like below.

bool searchElement(node \*, int);

1. Please implement the search function for the BST (*see the hints at the back pages of this tutorial*)
2. Save the completed source file and make sure it’s compiled.
3. Now, test your search function by first create a skewed BST just like in the picture (a) above.
4. Create the tree in the picture (b) above as well, and also test your search function to search for the same element.

Did you notice the difference in the running time?

The skewed tree as exemplified in the picture (a) represents the WORST-CASE for the search function of a BST, while the balanced BST in the picture (b) represents the BEST-CASE for the search function of a BST, while the balanced BST.

Since you cannot guarantee to always work with a balanced BST, you may start wondering how a Tree can be made perfectly balanced all the time?

If you’re curious about this, please go on the second practice.

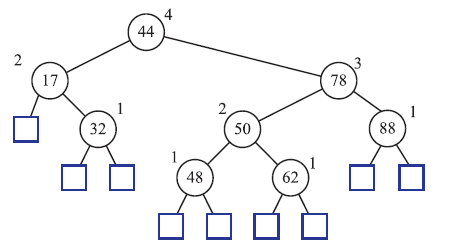
## Tutorial w7b: Implementing AVL Tree (optional)

In the previous section, the worst-case performance of BST search function is linear time, which is no better than the performance of list- and array-based search implementations

The simple correction is to add a rule to the binary search tree definition that maintains a logarithmic height for the tree. The rule we consider in this section is the following height-balance property, which characterizes the structure of a binary search tree T in terms of the heights of its internal nodes that the height of a node v in a tree is the length of the longest path from v to an external node):

Height-Balance Property: For every internal node v of T, the heights of the children of v differ by at most 1.

Any binary search tree T that satisfies the height-balance property is said to be an AVL tree, named after the initials of its inventors, Adelson-Velskii and Landis.



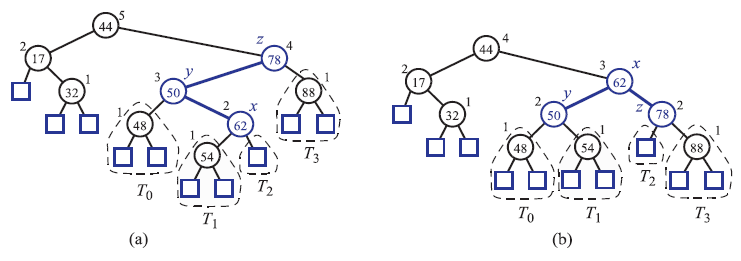
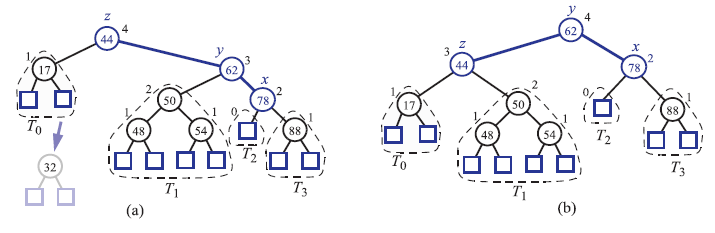
The above picture shows an example of an AVL tree. The keys of the entries are shown inside the nodes, and the heights of the nodes are shown next to the nodes.

### Practice:

1. Open the starter file (LinkedAVL-startercode.zip) in your IDE.
2. This file contains a linked implementation of an AVL Tree.
3. Try to understand how the code works, notably the insertion function and rebalancing (rotation) functions.
4. Create a tester function (main function) that does the following testing:
   1. **Create** the AVL tree shown above.
   2. **Insert** of an entry with key 54 in the AVL tree. Draw the resulting tree and you will notice after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; How to rebalance it? Run the rebalance function and observe the resulting tree. Does it satisfy the condition of an AVL tree now?
   3. Now **removal** of the entry with key 32 from the AVL tree. Draw the resulting tree and you will notice that after removing the node storing key 32, the root becomes unbalanced; How do you make it balanced? (*hints: a (single) rotation restores the height-balance property*)

### Questions:

1. Rose claims that the order in which a fixed set of entries is inserted into an AVL tree does not matter—the same AVL tree results every time. Give a small example that proves she is wrong.
2. Are the rotations in Figures (I) dan (II) below single or double rotations?
3. Draw the AVL tree resulting from the insertion of an entry with key 52 into the AVL tree of Figure II-b.
4. Draw the AVL tree resulting from the removal of the entry with key 62 from the AVL tree of Figure II-b.

(I) Insertion operation in AVL Tree: insert 44

(II) Deletion operation in AVL Tree: delete 32

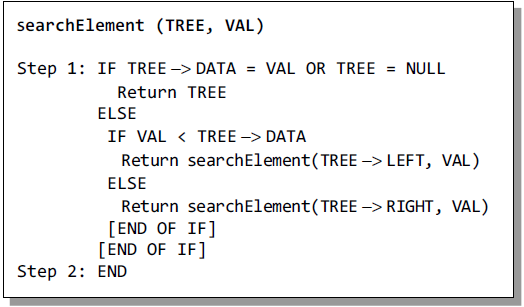
---end of Tutorial Week 7---

# Hints for Implementing the Search function in a BST

The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node. The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displaying an appropriate message.

However, if there are nodes in the tree, then the search function checks to see if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node.

The algorithm to search for an element in the binary search tree is presented below:



In Step 1, we check if the value stored at the current node of TREE is equal to VAL or if the current node is NULL, then we return the current node of TREE. Otherwise, if the value stored at the current node is less than VAL, then the algorithm is recursively called on its right sub-tree, else the algorithm is called on its left sub-tree.