BINARY SEARCH TREES AND AVL TREES

SEARCH TREES

Search trees are data structures that support the dynamic set operations *search, minimum, maximum, predecessor, successor, insert* and *delete.*

A **BINARY SEARCH TREE (BST)** is a special search tree that satisfies the following binary search tree property:

"The value of **every node** is GREATER THAN or EQUAL to the values of the nodes in its **left subtree** and is LESS THAN OR EQUAL to the values of the nodes in its right subtree."

Note: This definition means that duplicates may exist. (Meaning the BST of integers can contain more than one "3" etc.)

ADVANTAGE: *Searching using a BST can be done more efficiently (compared to searching using a normal binary tree*

BST OPERATIONS (A Quick Glance)

- **Searching:** Start at root, if value is less than the root, search the left subtree, if greater than, search right subtree.
- **Minimum:** The leftmost node without a child [The lower, left most node]
- **Maximum:** The rightmost node without a right child [The lower rightmost node]
- **Successor:** The value next to the value of the node in the sorted (increasing) order of all the nodes
- **Predecessor:** The value before the node in the sorted order of all the nodes

Note: Finding the predecessor and successor of a node is not that straightforward if the values are not unique.

- **Inserting** (more on this later)
- **Deleting** (more on this later)

Watch This:
https://www.youtube.com/watch?v=dYWz6wqvEEs

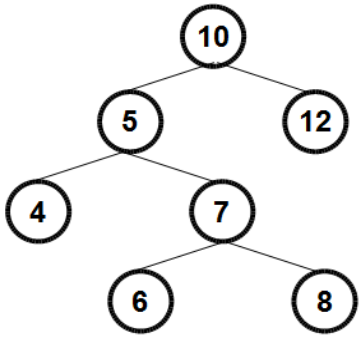
Read:
Module 6 (Page 57) of CMSC 204 Manual

BST OPERATIONS IN DETAIL

Let's have a quick recap of what **Binary Search Trees** are.

Once again, **Binary Search Trees (BSTs)** are search trees wherein every node is **greater than or equal to** the nodes in its **left** subtree and is **less than or equal to** the nodes in its **right subtree**.

The BST at the right demonstrates this property. Note that if we wish to print out the contents of the BST in sorted order, we would have to use the **inorder traversal** to do so.



BSTs have certain advantages over normal binary trees. One of which is searching which is done more efficiently since there's no need to examine **every** node in the tree when searching for a value.

We may perform several operations on BSTs. These operations include *search, minimum, maximum, successor, predecessor, insert* and *delete.*

SEARCHING A BST

Searching a BST is done recursively:

1. Start at the root. If the value is the root, return the root.
2. If the value is less than the root, search the left subtree (recursively).
3. If the value is greater than the root, search the right subtree (recursively).
4. Do this process recursively until you find the value in the tree. If you reach a leaf and you have not yet found the value, it means that the value is not in the tree.

SEARCHING THE MINIMUM AND MAXIMUM IN A BST

The **minimum** value in a BST is found in the lower leftmost node or the left most node without a left child. Therefore, you must *keep searching the left subtree recursively until you arrive at a node whose left pointer points to null.*

Meanwhile, the **maximum** value of a BST is found in the lower rightmost node without a right child. You have to *keep searching the right subtree recursively until you reach a node who's right pointer points to null*.

SUCCESSOR AND PREDECESSOR OF A NODE

When the values of the node are printed in increasing order, the **predecessor** of a node is the node containing the value right before the value of the node. The **successor** on the other hand, is the node containing the value right after the value of the node. For example, a BST contains the values 1 2 3 4 5 6 7 8. The predecessor of the **node 5** would be the node containing the value 4 and its successor would be the node containing the value 6.

If all the values in a BST are **unique** (there's no repeated value), the **predecessor of a node** is simply the *maximum* of its *left subtree.* The **successor of a node,** meanwhile, is simply the *minimum* of its *right subtree.*

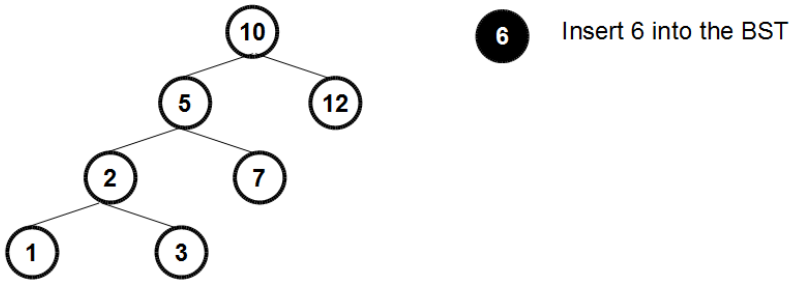
INSERTING A NODE INTO A BST

When you insert a node into a BST, you cannot just insert it anywhere. You have to make sure that when the node is inserted, the property that makes a BST unique still holds.

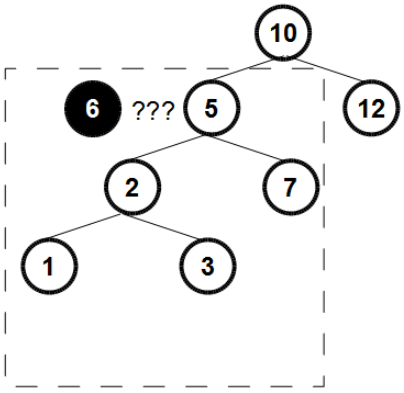
To insert a node into a BST:

1. Determine the position where the node is to be inserted. Look for it's position in the left subtree if the value is less than the node, look for the value at the right subtree if the value is greater than the node.
2. Insert that node as either the left or right child.

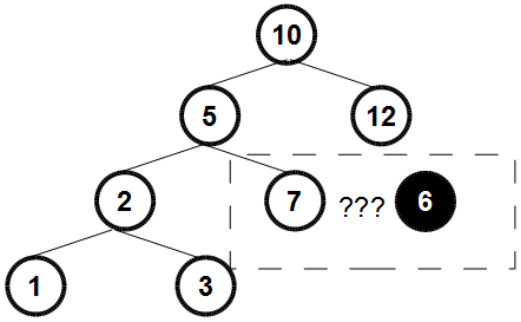
Example:



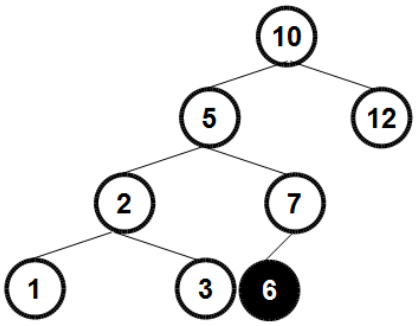
We first compare 6 with the root (10). Since 6 is less than 10, we proceed to searching for its position in the left subtree.



Now, we compare 6 with 5 and we see that it is greater than 5. We then proceed to search for its position, this time in the right subtree.



6 is less than 7 so we search for its position in 7's left subtree. However, since 7's left subtree is null, we can therefore insert 6 as it's **left child.**



Doing the insertion in the linked list implementation is relatively easy. All you need is a simple while loop with two branches of if statements inside the while loop block. No recursion necessary.

DELETING A NODE FROM A BST

Deleting a node from a BST is a bit more complicated, compared to inserting a node into a BST. You have to be careful that you don't end up destroying the BST when you perform the deletion.

In deletion, there are three possible scenarios.

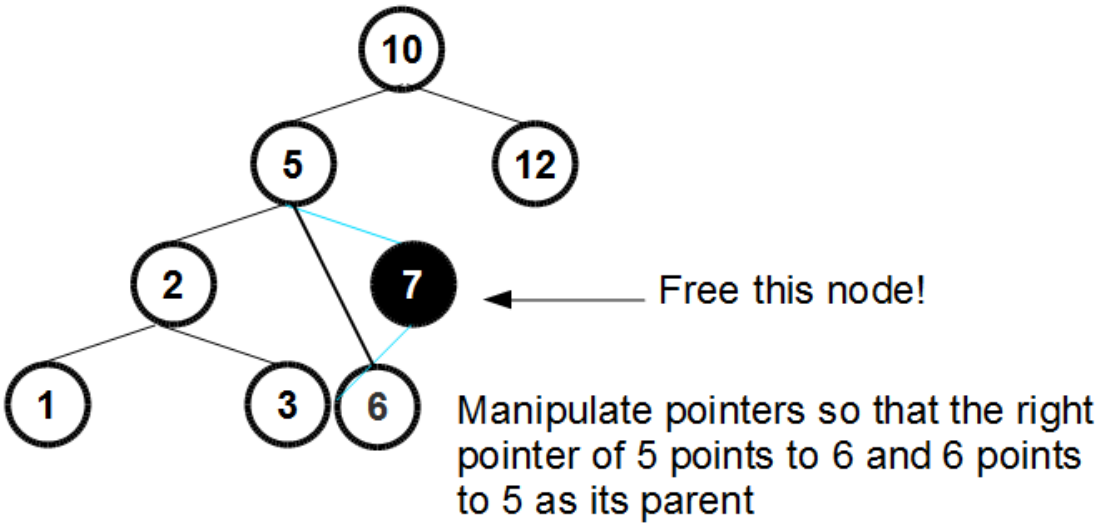
1. The node to be deleted has no child
2. The node to be deleted has only one child
3. The node to be deleted has two children

The node to be deleted has no child.

Simply free the node, but make sure that its parent's left pointer should eventually point to NULL (if it's the left child) or the parent's right pointer should eventually point to NULL (if it's the right child).

The node to be deleted has only one child.

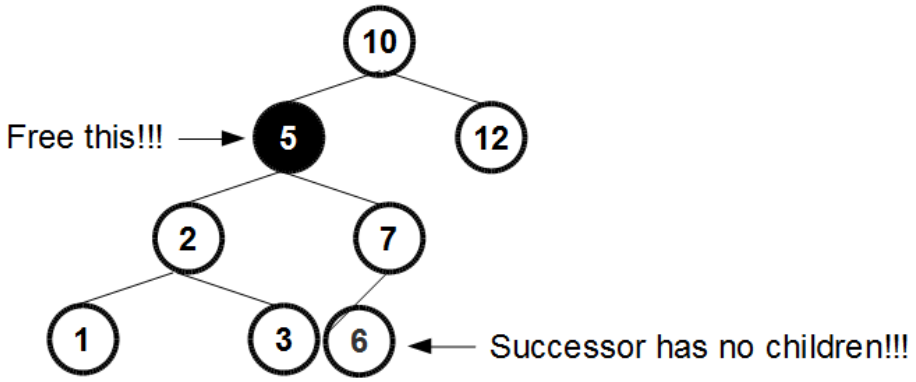
The node to be deleted is “spliced out”. In other words, the “grandparent” becomes the parent of the deleted node's child.



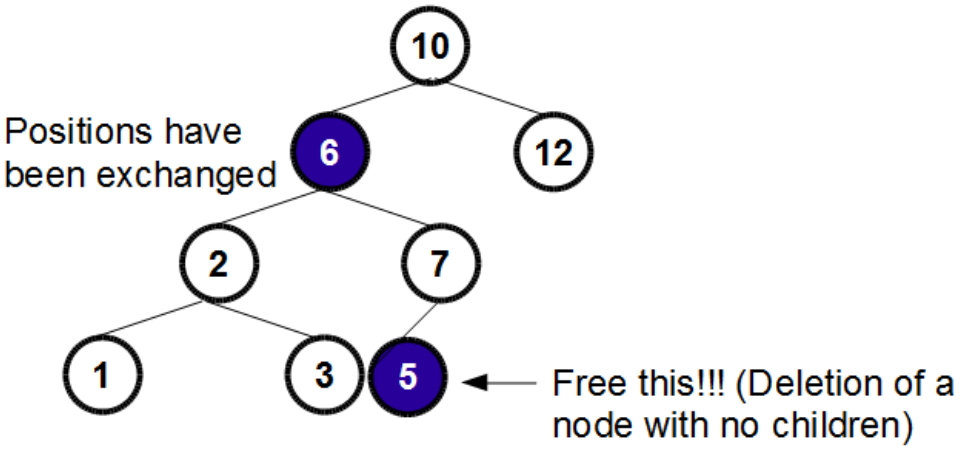
The node to be deleted has two children.

If the node to be deleted has two children, the node has to be replaced by its **successor.**

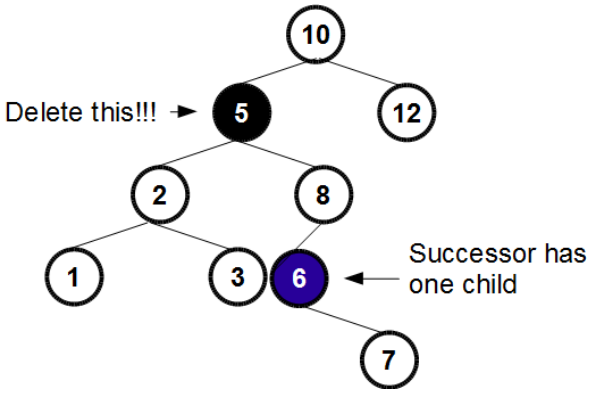
Case 1: The successor has no children.

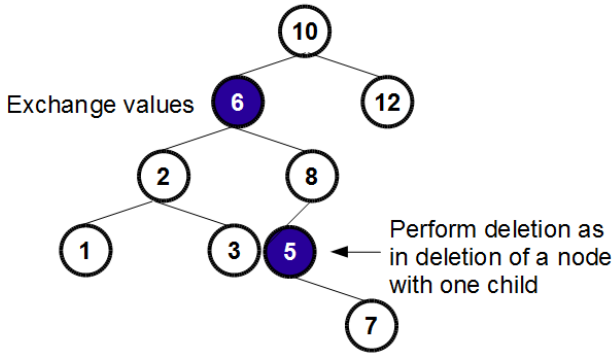


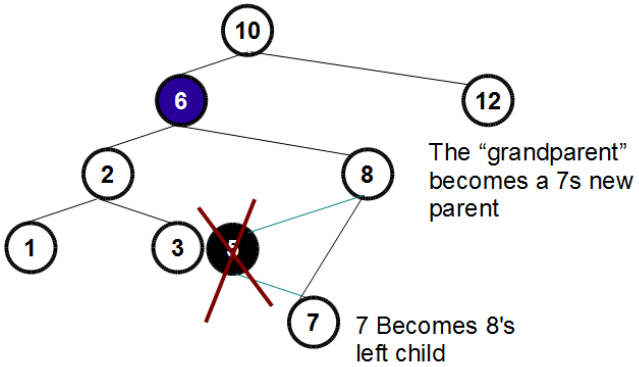
Simply exchange the values of the two nodes and then free the node that now contains 5.



Case 2: The successor has 1 child. (Note that a successor always has **at the very most** one child, which is a **right child**)







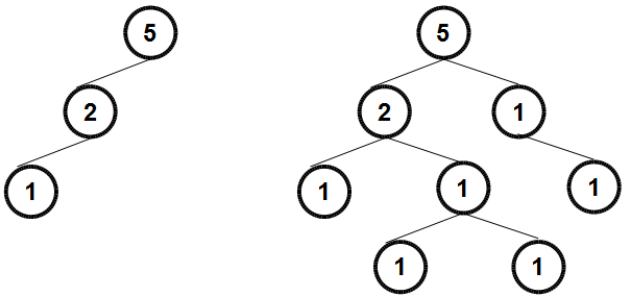
So how would you translate all of these into C##???

Try it out: ***http://visualgo.net/bst.html***

See in C#: ***https://msdn.microsoft.com/en-us/library/ms379572(v=vs.80).aspx***

AVL TREES

AVL (Adelson-Velskii and Landis) trees are binary search trees that are height balanced, that is, both subtrees of every node in the tree differ by no more than one in height.



Why AVL Trees???

"Binary search trees produce good times for the operations search, insert, minimum and maximum, predecessor and successor, and delete when the binary tree is BALANCED. However, it is possible to produce a BST that is not balanced."

Two Russian mathematicians *Adelson-Velskii* and *Landis* formulated a method that ensured that a BST would always be balanced.

Watch:
AVL TREES
https://www.youtube.com/watch?v=lQp431QjrWI

IMPLEMENTATION

We need to add an additional field to the node structure which would **monitor the height of the tree rooted at a certain node,** so that we can **ensure that the tree remains height balanced**.

TRANSFORMING NON-AVL INTO AVL TREES

When we insert a node to an AVL tree, it is possible that we would end up with a **NON-AVL TREE.** Therefore, we have to perform some **transformations** on that tree so that we would eventually have a tree that satisfies the AVL Tree property.

NOTE: We can also use these transformations to transform a BST that was never an AVL tree to begin with into an AVL tree.

PSEUDOCODES FOR AVL TREE OPERATIONS

Single right rotate tree rooted at node A

```
B = leftchild(A)
temp = rightchild(B)
rightchild(B) = A
leftchild(A) = temp
```

Single left rotate tree rooted at node A

```
B = rightchild(A)
temp = leftchild(B)
leftchild(B) = A
rightchild(A) = temp
```

Double left rotate tree rooted at node A (Right-left Rotate)

```
Single right rotate tree rooted at rightchild(A)
Single left rotate tree rooted at A
```

Double right rotate tree rooted at node A (Left-right Rotate)

```
Single left rotate tree rooted at leftchild(A)
Single right rotate tree rooted at A
```

Try it out: http://visualgo.net/bst.html (Click AVL Tree Link)

Look for C# AVL code to see how it is implemented! Don't forget to try it out!

Exercise What You've Learned:

On paper, perform the following sets of insertions.

For every number, start with an empty AVL tree. Insert the nodes one by one and perform the necessary rotations that would keep the AVL tree property. Show the **step-by-step** illustration of **each insertion and each rotation.**

1. **Insert the values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 (use single rotations)**
2. **Insert the values 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 (use single rotations)**
3. **Insert the values 7, 8, 6, 9, 5, 10, 4, 11, 3, 12, 2, 13, 1, 14 (use double rotations)**
4. **Insert the values 7, 8, 9, 10, 5, 5, 4, 3, 11, 12, 1, 2, 13, 13 (use double rotations)**

Takeaway Thoughts and Questions:

1. *What new knowledge about BSTs and AVL Trees is the most interesting for you?*
2. *What will help you in understanding BSTs and AVL Trees more?*