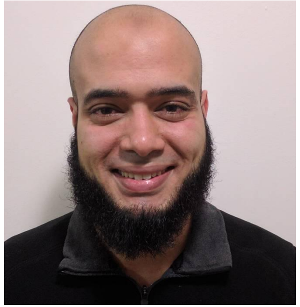
Data Structures

Self-balancing binary search tree

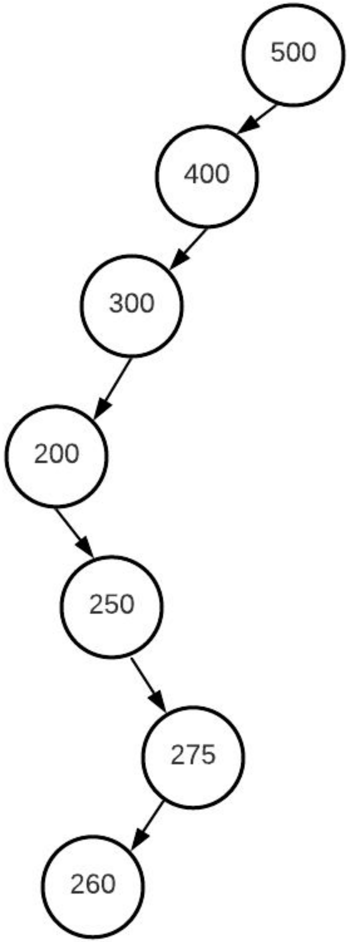
Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



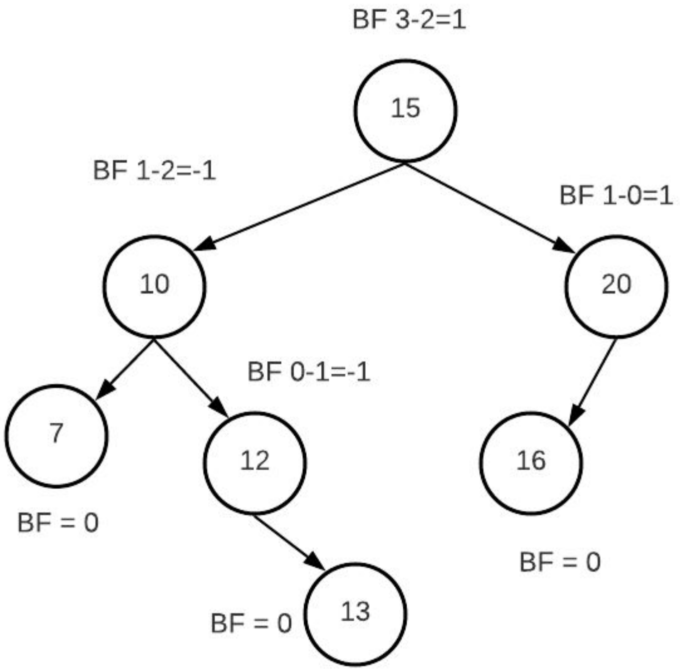
Recall: Degenerate BST

- Each node has 1 child
- From performance perspective, it is like a linkedlist, making several operations O(n)
- In general, trees with height much far from log(n) are not efficient!



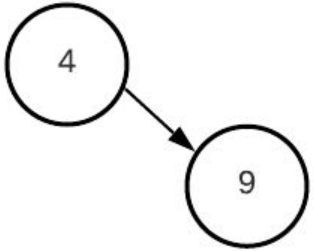
Recall: Balanced BST

- **Difference** between **heights** of left subtree and right subtree is not more than 1.
  - Height(left) - Height(right)
  - **For visualization**, assume height(leaf) = 1
- Let's compute this difference for each node (**Balance factor**)
  - -7 means = right height greater with 7
  - -1, 0, 1 ⇒ BBST
  - |bf| > 1 ⇒ imbalanced tree



Consider height and balance_factor()

- In last line of insert() function we call: update_height()




```cpp
class AVLTree {
private:
    int data { };
    int height {};
    AVLTree* left { };
    AVLTree* right { };

    int ch_height(AVLTree* node) {  // child height
        if (!node)
            return -1;           // -1 for null
        return node->height;     // 0 for leaf
    }
    void update_height() {  // call in end of insert function
        height = 1 + max(ch_height(left), ch_height(right));
    }
    int balance_factor() {
        return ch_height(left) - ch_height(right);
    }
```

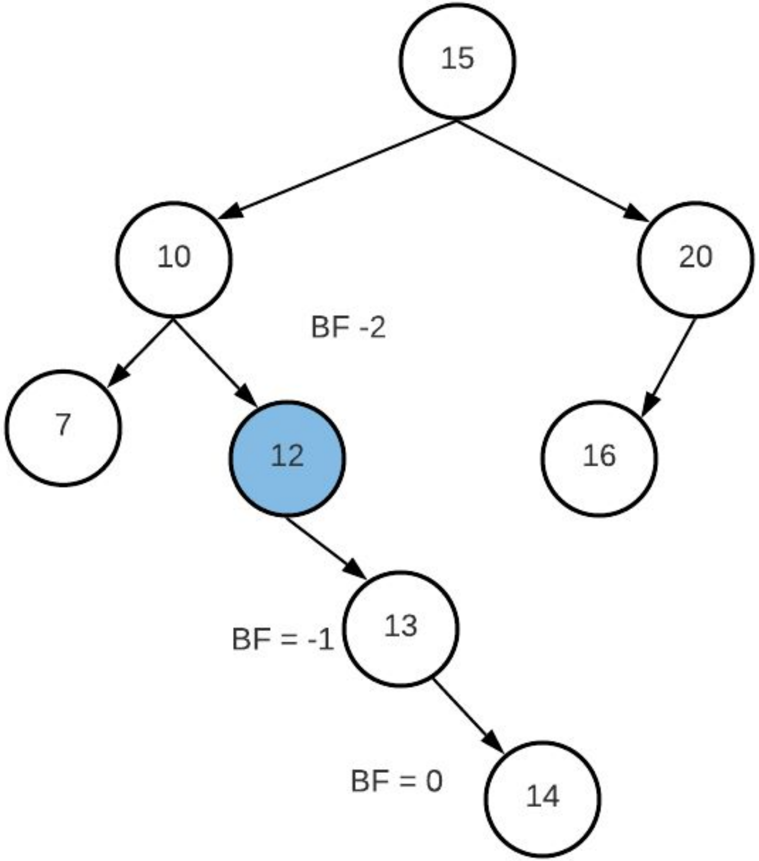
```cpp
class AVLTree {
private:
	struct BinaryNode {
		int data { };
		int height { };
		BinaryNode* left { };
		BinaryNode* right { };

		BinaryNode(int data) :
				data(data) {
		}

		int ch_height(BinaryNode* node) {   // child height
			if (!node)
				return -1;            // -1 for null
			return node->height;    // 0 for leaf
		}
		int update_height() {   // call in end of insert function
			return height = 1 + max(ch_height(left), ch_height(right));
		}
		int balance_factor() {
			return ch_height(left) - ch_height(right);
		}
	};
```

Maintaining BBST

- Insertion in BST depends on input order ⇒ will generate imbalance trees
- **Self-balancing** BST trees follow **change and fix** approach to keep it BBST
  - E.g. Insert a new element in BST
  - Is imbalance BST?
  - Yes ⇒ **Fix** the tree to have again |BF| <= 1
- There are several such trees that maintains tree as BBST
  - **AVL Trees**: one of oldest and simplest ways
  - Red-Black Trees, Splay Trees, Treaps
- This section is focusing on AVL Trees

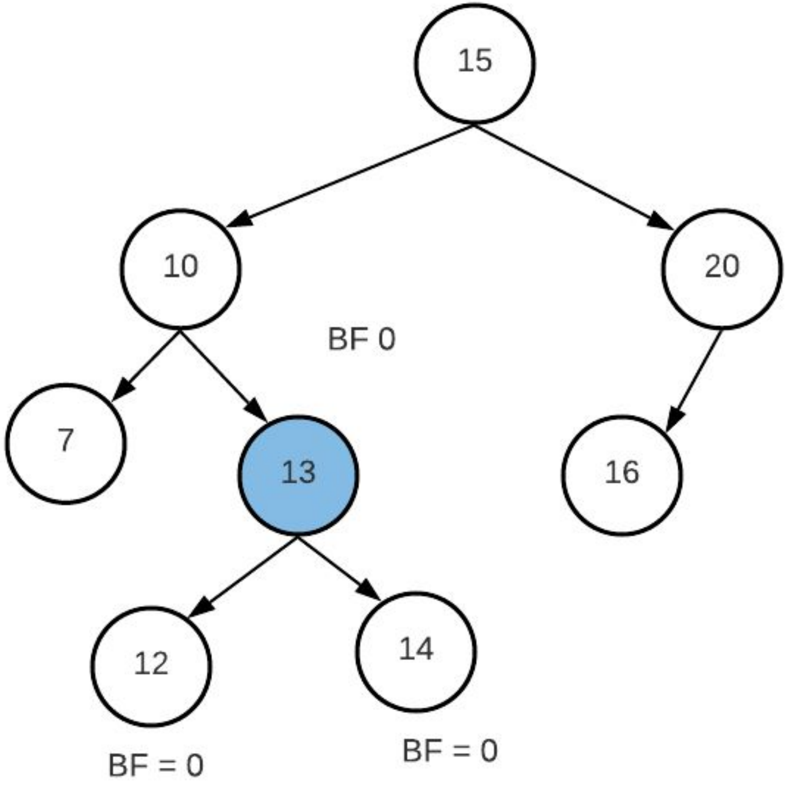
Change (insert 14)

- Let's insert 14
- Now, recompute Balance Factor
- Recursively we reach leave nodes then go back to parents and so on
- Once you detect |BF| > 1, this subtree is not BST
- Think for 2 min how to restructure subtree (12) to make it balanced BST?



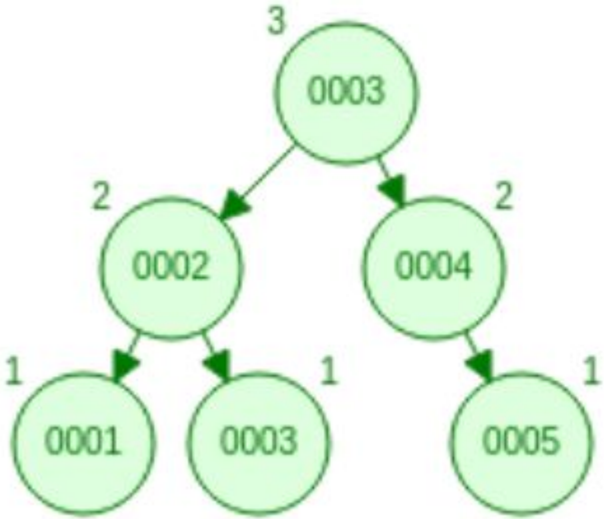
Fix (Tree Rotation)

- If we pushed node 13 **up** and node 12 **down** left, this subtree is fixed
  - Observe, we did not change other subtrees
  - Observe, tree remains BBST
- This kind of systematic change is called **tree rotation**
- If after insertion, in **bottom up** style we kept fixing corrupted sub-trees ⇒ BBST



AVL Demo

- Many web sites provide online demo for insertion/deletion in AVL
- Try to insert almost sorted numbers and watch the tree **restructuring**



"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."