# CS218- Data Structures
# Week 13

Muhammad Rafi
December 01, 2020

## Agenda

- BST – Pros n Cons
- BST Motivation
- AVL Tree
- Different Rotations

## Binary Search Tree (BST)

- Pros
  - max, min, search, insertion, and deletion all operations can be performed in O(log h), where h is the height of a BST.
- Cons
  - BST structures very much dependents on the order of keys inserted into the BST.

## Binary Search Tree (BST)

- Motivation
  - Can we do some thing to retains the balance of a BST ?
    - Reconstruct the BST when your accounting of access suggest some bad performance
    - Allow a little out of balance and keep the operations cost to at least O(log (h+1)). Follow the insertion of BST.
    - Restrict the tree to be balance all the time.
- Solution
  - AVL Trees
  - Red Black Trees

# AVL Trees

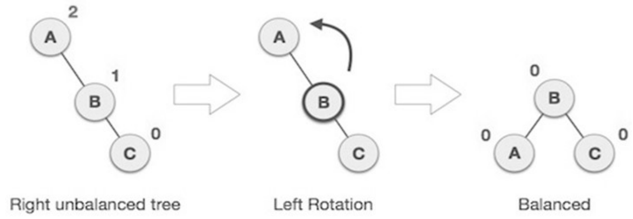
- Adelson-Velskii and Landis (AVL) trees (height-balanced trees) – it is a kind of self balancing tree.
- In an AVL tree, the heights of the two child subtrees of any node differ by at most one.
- If at any time they differ by more than one, rebalancing is done to restore this property.
- Idea is to keep the tree almost balance all the time to gives O(log h+1) operations.

# AVL Trees

- Insertion in AVL Tree
  - The insertion is same as BST but after this we try to restrict the condition on balance factor.
  - The absolute (height of left subtree – the height of right subtree) should not be greater than 1.
  - If the insertion violated the restriction – the tree is rearranged to obey the conditions.

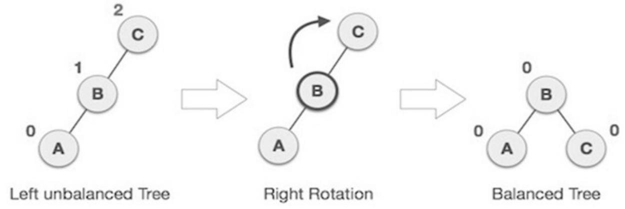
# AVL Trees

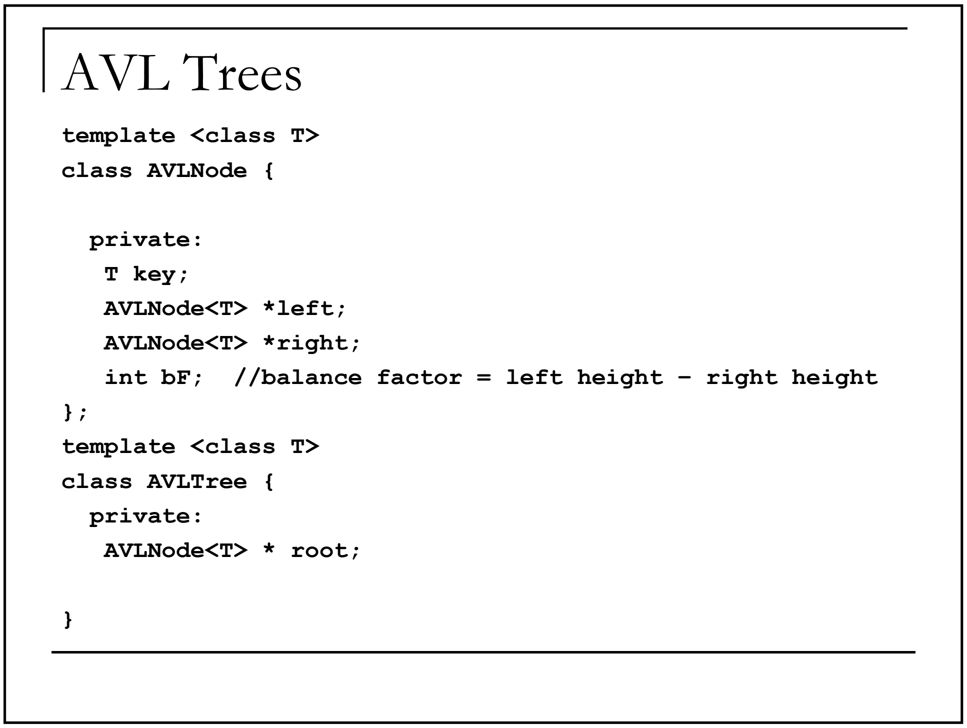
- Left Rotation
  - if a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation



# AVL Trees

- Right Rotation
  - AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



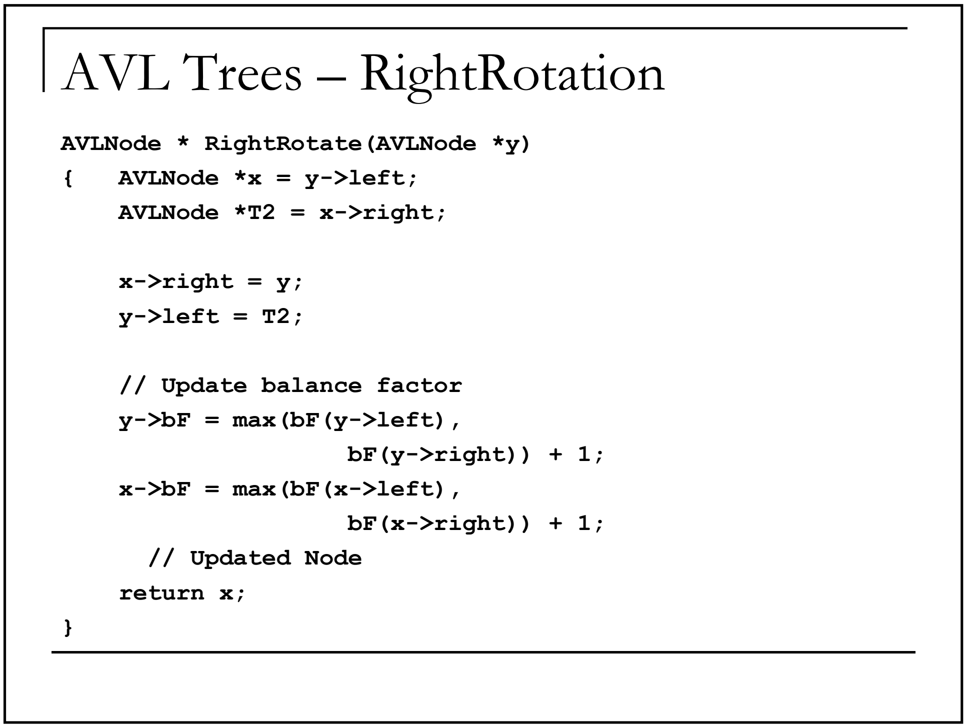


## AVL Trees

```
template <class T>
class AVLNode {

  private:
   T key;
   AVLNode<T> *left;
   AVLNode<T> *right;
   int bF;  //balance factor = left height – right height
};
template <class T>
class AVLTree {
  private:
   AVLNode<T> * root;

}
```



## AVL Trees – RightRotation

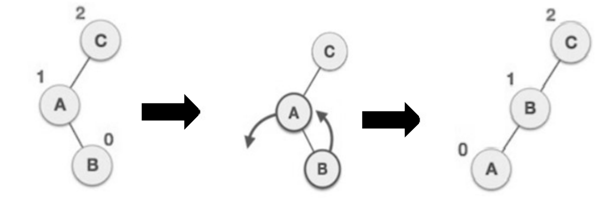
```
AVLNode * RightRotate(AVLNode *y)
{   AVLNode *x = y->left;
    AVLNode *T2 = x->right;

    x->right = y;
    y->left = T2;

    // Update balance factor
    y->bF = max(bF(y->left),
                bF(y->right)) + 1;
    x->bF = max(bF(x->left),
                bF(x->right)) + 1;
      // Updated Node
    return x;
}
```

# AVL Trees

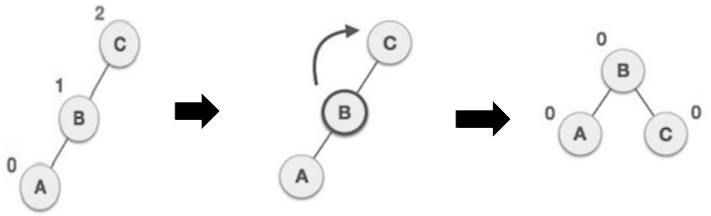
- Double Rotation (Left-Right Rotation)
  - A left-right rotation is a combination of left rotation followed by right rotation.
  - The result of this double rotation is perfectly balance tree.



First left Rotation

# AVL Trees

- Double Rotation (Left-Right Rotation)
  - A left-right rotation is a combination of left rotation followed by right rotation.
  - The result of this double rotation is perfectly balance tree.

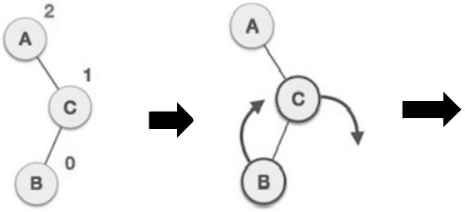


Second Right Rotation

## AVL Trees

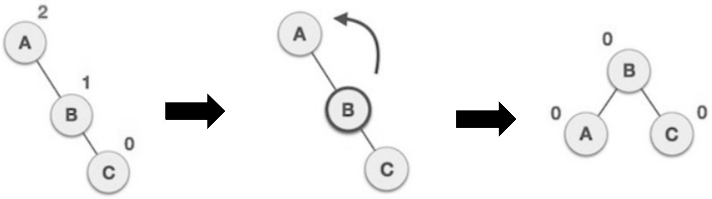
- Double Rotation (Right-Left Rotation)
  - A right-left rotation is a combination of right rotation followed by left rotation.
  - The result of this double rotation is perfectly balance tree.

First right Rotation

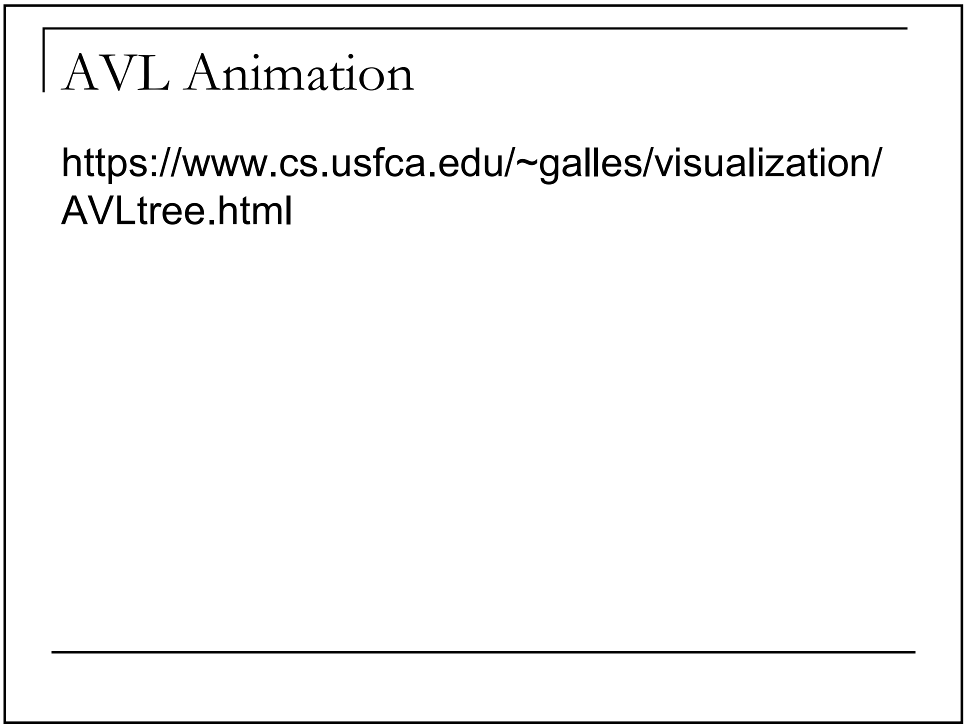


## AVL Trees

- Double Rotation (Right-left Rotation)
  - A right-left rotation is a combination of right rotation followed by left rotation.
  - The result of this double rotation is perfectly balance tree.



Second Left Rotation



# AVL Animation

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html