


# Tutorial on Height Balanced Trees

Course: Algorithms 1  
Semester: Autumn 2019

A dark blue diagonal bar that starts from the bottom left corner and extends towards the top right corner, covering the lower half of the slide.

# Question 1: Merge Two Balanced Binary Search Trees



You are given two balanced binary search trees e.g., AVL trees. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be “m” elements in first tree and “n” elements in the other tree. Your merge function should take  $O(m+n)$  time. You can assume that the sizes of trees are also given as input.

You are given two balanced binary search trees e.g., AVL trees. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be “m” elements in first tree and “n” elements in the other tree. Your merge function should take  $O(m+n)$  time. You can assume that the sizes of trees are also given as input.

## Hint: Inorder Traversals

# Algorithm:

- Perform inorder traversal of first tree and store the traversal in one temp array `arr1[]`. This step takes  $O(m)$  time.
- Perform inorder traversal of second tree and store the traversal in another temp array `arr2[]`. This step takes  $O(n)$  time.
- The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size  $m + n$ . This step takes  $O(m+n)$  time.
- Construct a balanced tree from the merged array. This step takes  $O(m+n)$  time.

```

struct TNode* mergeTrees(struct TNode *root1, struct TNode *root2, int m, int n)
{
    // Store inorder traversal of first tree in an array arr1[]
    int *arr1 = new int[m];
    int i = 0;
    storeInorder(root1, arr1, &i);

    // Store inorder traversal of second tree in another array arr2[]
    int *arr2 = new int[n];
    int j = 0;
    storeInorder(root2, arr2, &j);

    // Merge the two sorted array into one
    int *mergedArr = merge(arr1, arr2, m, n);

    // Construct a tree from the merged array and return root of the tree
    return sortedArrayToBST (mergedArr, 0, m+n-1);
}

```

## Question 1: Merge Two Balanced Binary Search Trees

```
struct TNode* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    struct TNode *root = newNode(arr[mid]);

    /* Recursively construct the left subtree and make it
       left child of root */
    root->left = sortedArrayToBST(arr, start, mid-1);

    /* Recursively construct the right subtree and make it
       right child of root */
    root->right = sortedArrayToBST(arr, mid+1, end);

    return root;
}
```

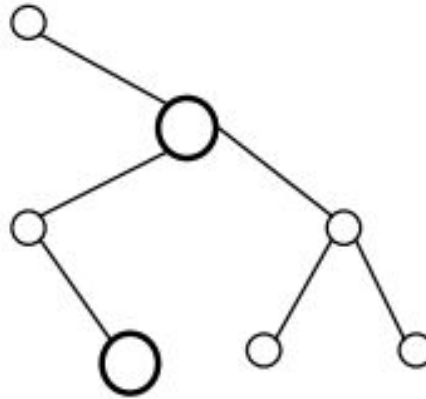
## Question 1: Merge Two Balanced Binary Search Trees

## Question 2: Loneliness-ratio





A node in a binary tree is an ***only-child*** if it has a parent node but no sibling node (Note: The root does not qualify as an only child).



A node in a binary tree is an ***only-child*** if it has a parent node but no sibling node (Note: The root does not qualify as an only-child).

The ***loneliness-ratio*** of a given binary tree  $T$  is defined as:

$$LR(T) = \frac{\text{(The number of nodes in } T \text{ that are ***only-child***)}}{\text{(The number of nodes in } T\text{)}}$$

**Prove** that for any nonempty AVL tree  $T$  we have that  $LR(T) \leq 1/2$

A node in a binary tree is an ***only-child*** if it has a parent node but no sibling node (Note: The root does not qualify as an only-child).

The ***loneliness-ratio*** of a given binary tree T is defined as:

$$LR(T) = \frac{\text{(The number of nodes in T that are ***only-child***)}}{\text{(The number of nodes in T)}}$$

**Prove** that for any nonempty AVL tree T we have that  $LR(T) \leq \frac{1}{2}$


**Hint: Which nodes in an AVL Tree can be only-children?**

## Solution:

In an AVL tree, just the leaves may be ***only-child***, and therefore for every ***only-child*** in  $T$ , there exists a unique parent node that is not an only-child.

The total number of ***only-child*** in  $T$  is at most  $n/2$ , which means that  $LR(T) \leq (n/2)/n = 1/2$ .

Question 3: Optimal sequence for  
AVL tree insertion (without any  
rotations)



Henry has been given the following array of integers to be added to an AVL tree. However, Henry doesn't understand rotations. Help him to find the sequence in which these integers should be added to an AVL tree such that no rotations are required to balance the tree.

Array = [10, 20, 15, 25, 30, 16, 18, 19]

Henry has been given the following array of integers to be added to an AVL tree. However, Henry doesn't understand rotations. Help him to find the sequence in which these integers should be added to an AVL tree such that no rotations are required to balance the tree.

Array = [10, 20, 15, 25, 30, 16, 18, 19]

**Hint 1: Sort the array and create a BST.**

Henry has been given the following array of integers to be added to an AVL tree. However, Henry doesn't understand rotations. Help him to find the sequence in which these integers should be added to an AVL tree such that no rotations are required to balance the tree.

Array = [10, 20, 15, 25, 30, 16, 18, 19]

**Hint 1: Sort the array and create a BST.**

**Hint 2: Level-order traversal of BST.**



## Solution:

- Sort the given array of integers.
- Create the AVL (BST) tree from the sorted array by using the method `sortedArrayToBST(int arr[], int start, int end)` as in Question 1.
- Now, find the level order traversal of the tree which is the required sequence. Adding numbers in this sequence will always maintain the height balance property of all the nodes in the tree.

```
void printLevelOrder(TNode *root)
{
    if (root == NULL) return;

    queue<TNode *> q;
    q.push(root);

    while (q.empty() == false)
    {
        TNode *node = q.front();
        cout << node->data << " ";
        q.pop();
        if (node->left != NULL)
            q.push(node->left);
        if (node->right != NULL)
            q.push(node->right);
    }
}
```

Question 3: Optimal sequence for AVL tree insertion

# New Topic 1: Red-Black Trees

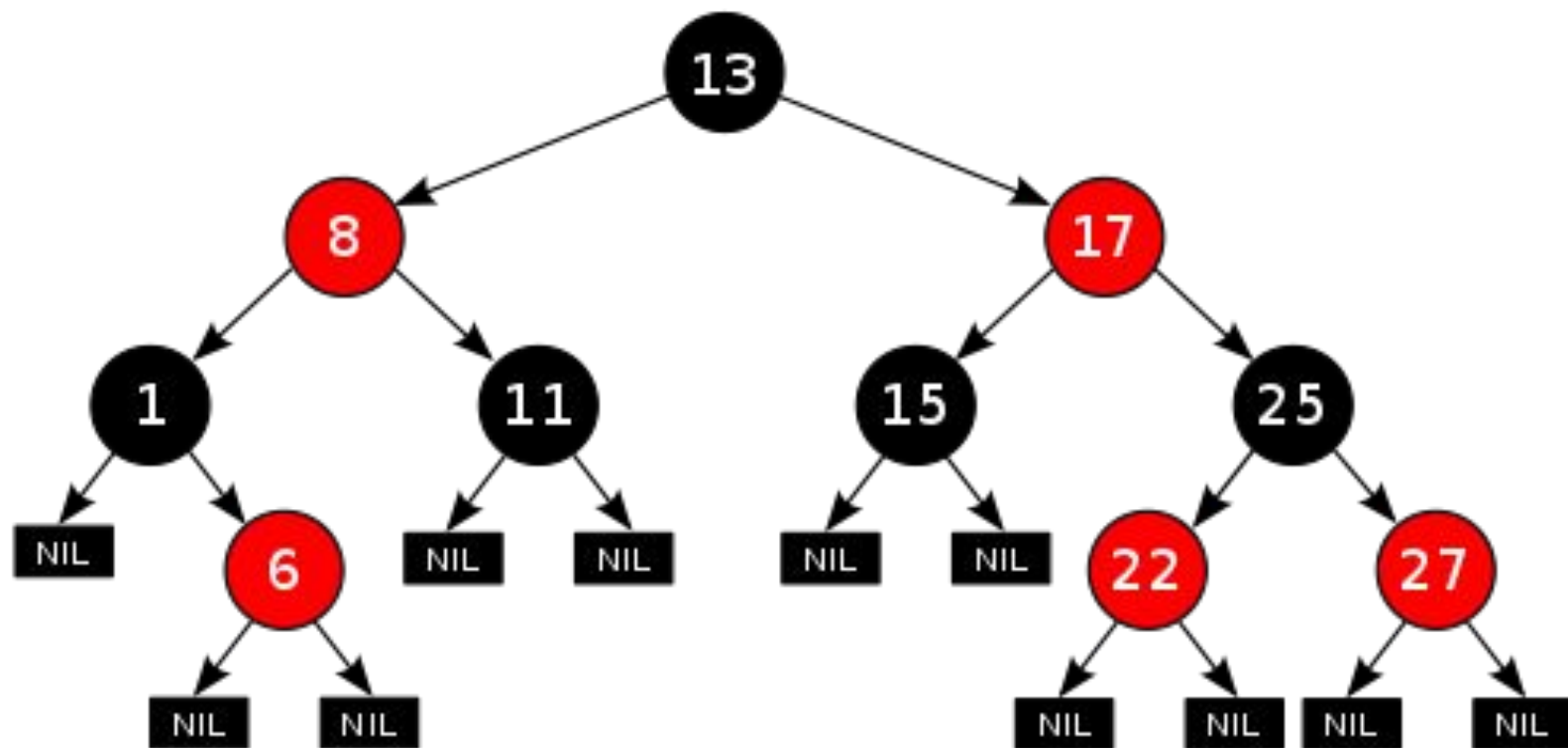


Red-black trees are an evolution of binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations.

This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantee that the deepest path in the tree is not longer than twice the shortest one (from the root to any leaf).

A red-black tree is a binary search tree with the following properties:

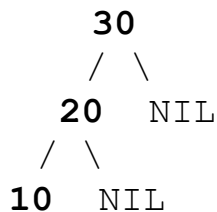
- Every node is colored with either **red** or **black**. Root of tree is always **black**.
- All leaf (nil) nodes are colored with **black**; if a node's child is missing then we will assume that it has a nil child in that place and this nil child is always colored **black**.
- Both children of a **red** node must be **black** nodes.
- Every path from a node  $n$  to a descendent leaf has the same number of **black** nodes (not counting node  $n$ ).



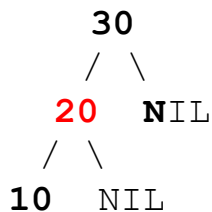
- The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion.
- So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred.
- If the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

# A chain of 3 nodes is not possible in Red-Black Trees.

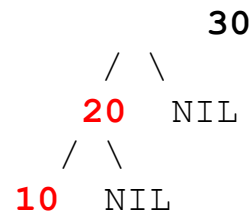
Following are **NOT** Red-Black Trees:



Violates  
Property 4

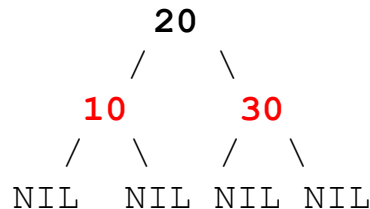
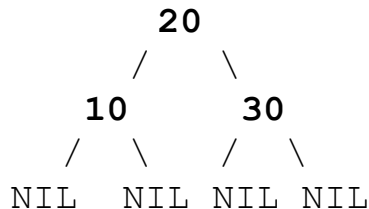


Violates  
Property 4



Violates  
Property 3

Following are different possible Red-Black Trees with above 3 keys:



Red-Black Tree: How is the balance maintained?

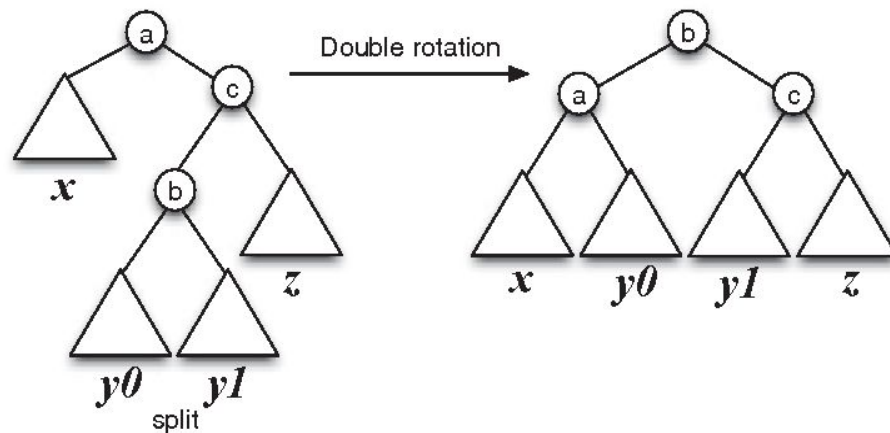
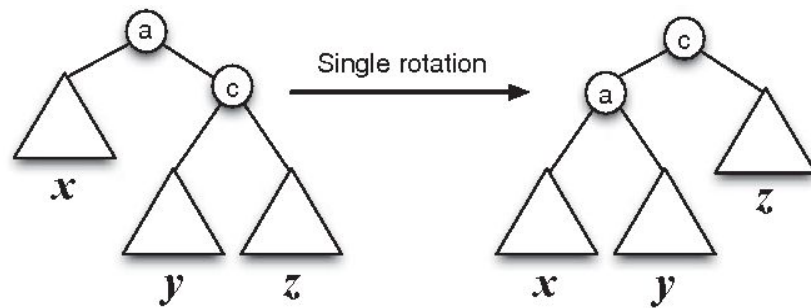


# Demo

# New Topic 2: Weight Balanced Tree



- A Weight Balanced Tree (**WBT**) is a BST that stores the size of subtrees in the nodes.
- A **node** in WBT keeps the following information:
  - Key
  - Left and Right pointers
  - Size
- Size of a node **n** is calculated as
  - $\text{Size}[n] = \text{Size}[n.\text{Left}] + \text{Size}[n.\text{Right}] + 1$
- **Weight** of **n** is calculated as  $w[n] = \text{Size}[n] + 1$
- A **WBT** is called  **$\alpha$ -weight-balanced** if for every node **n**:
  - $w[n.\text{Left}] \leq \alpha * w[n]$  and  $w[n.\text{Right}] \leq \alpha * w[n]$



## Problem:

- Consider a WBT where the number of nodes in the left subtree is at least half and at most twice the number of nodes in the right subtree.
- What is the order of maximum possible height of the Tree?

## Homework:

- Check the reference book: ***Advanced Data Structures*** [Online] by **Peter Brass**, **page no. 61-71**, Cambridge University Press.
- Try to understand the insertion and deletion operation in a WBT

## Problem:

- Consider a WBT where the number of nodes in the left subtree is at least half and at most twice the number of nodes in the right subtree.
- What is the order of maximum possible height of the Tree?
- Ans.  $O(\log_{3/2} n)$  - **Prove** it.

## Homework:

- Check the reference book: ***Advanced Data Structures*** [Online] by **Peter Brass**, **page no. 61-71**, Cambridge University Press.
- Try to understand the insertion and deletion operation in a WBT

# Question 4: Homework Problem



For the implementation of an AVL tree, each node  $v$  has an extra field  $h$ , the height of the subtree rooted at  $v$ . The height can be used in order to balance the tree.

An AVL tree with  $n$  nodes thus requires  $O(\log \log n)$  extra bits as  $h = O(\log n)$ .

- **Part 1:** How can we reduce the number of extra bits necessary for balancing the AVL tree?
- **Part 2:** Suggest an algorithm for computing the height of a given AVL tree with the representation suggested in Part 1.



## Solution for Part 1:

In addition to the regular BST fields, each node  $v$  will have 2-bits balance mark:

- $00 \equiv '/'$  :  $h(v.\text{left}) > h(v.\text{right})$
- $01 \equiv '-'$  :  $h(x.\text{left}) = h(x.\text{right})$
- $10 \equiv '\backslash'$  :  $h(x.\text{left}) < h(x.\text{right})$

## Solution for Part 2:

We will follow the path from the root to the deepest leaf by following the 'balance' property. If a sub tree is balanced to one side, the deepest leaf resides on that side.

```
calcHeight( T ):
    if T == null
        return -1
    if T.balance == '/' or T.balance == '-'
        return 1 + calcHeight( T.left )
    else
        return 1 + calcHeight( T.right )
```

Time complexity –  $O(h)$