ELEC 278

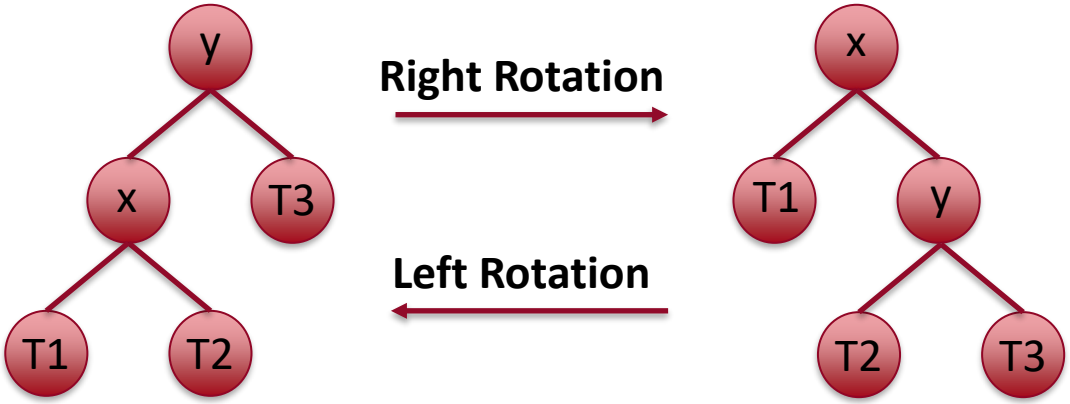
Tutorial 9

2024-25 Fall

Rebalancing an AVL Tree

- To ensure that an AVL tree remains AVL after each deletion or insertion, we must add some re-balancing to the standard BST delete operation

keys(left) < key(root) < keys(right)



keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

BST is not violated anywhere

Rebalancing an AVL Tree

Right Rotate

```
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
height(y->right))+1;
    x->height = max(height(x->left),
height(x->right))+1;

    // Return new root
    return x;
}
```

Left Rotate

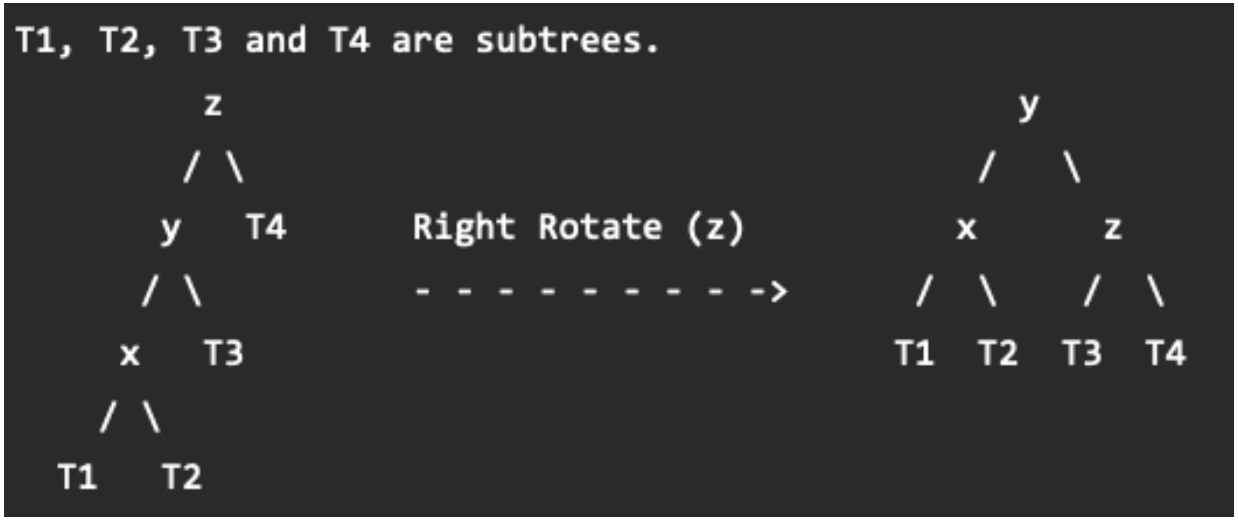
```
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    //  Update heights
    x->height = max(height(x->left),
height(x->right))+1;
    y->height = max(height(y->left),
height(y->right))+1;

    // Return new root
    return y;
}
```

Deletion in an AVL Tree Process

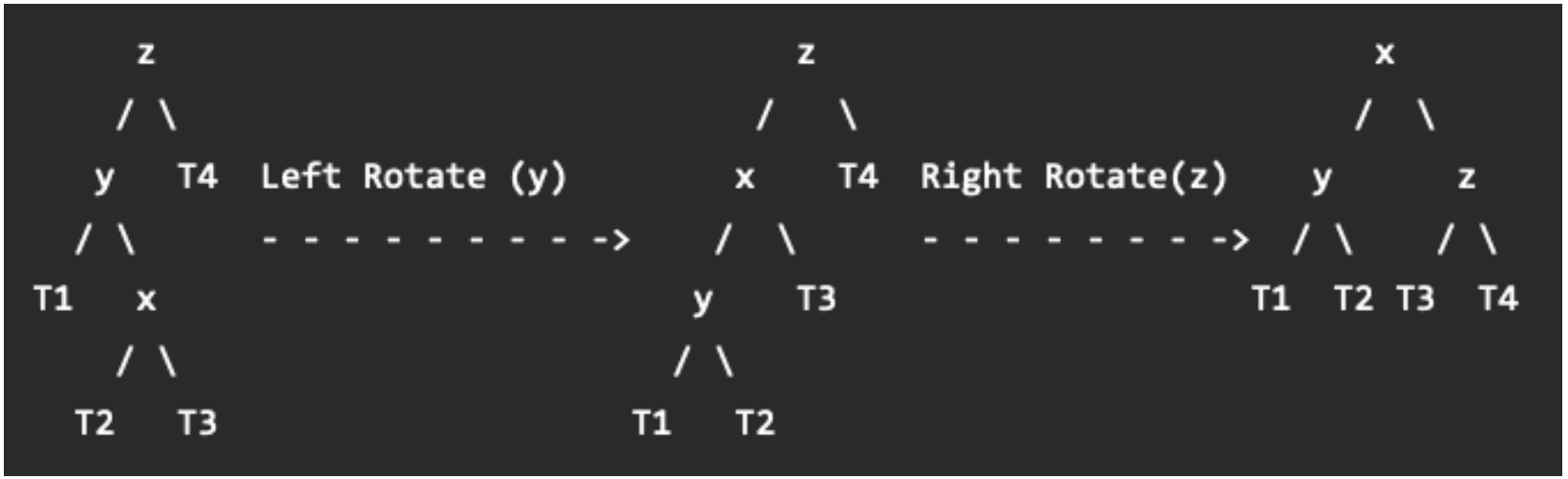
Let w be the node to be deleted:

1. Perform standard BST delete for w.
2. Travel up and find the first unbalanced node.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with the first unbalanced node. Following are the possible 4 arrangements:
   a) Left Left Case
   b) Left Right Case
   c) Right Right Case
   d) Right Left Case

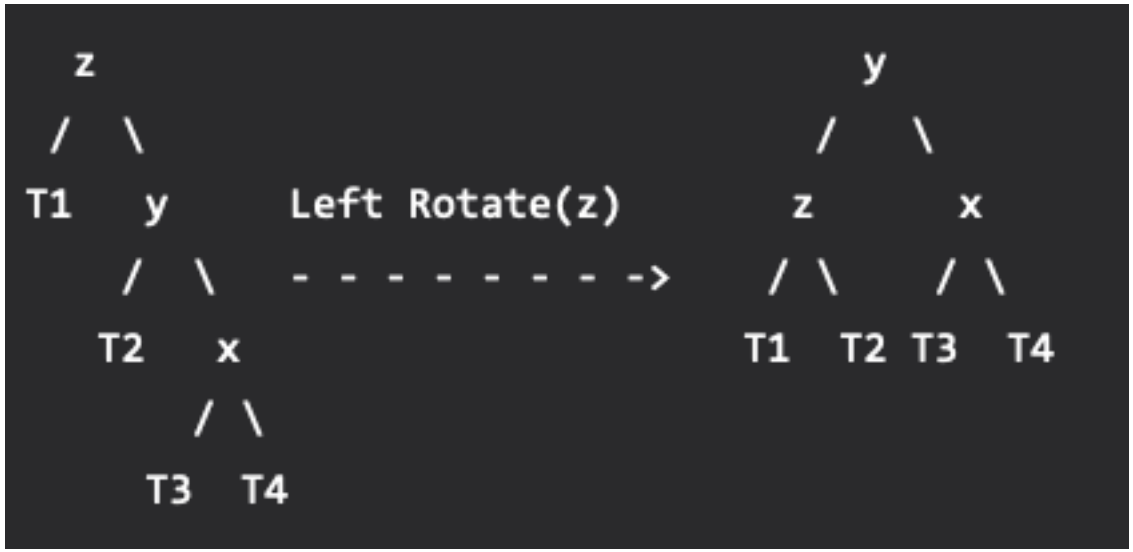
Rebalancing an AVL Tree (Left Left Case)



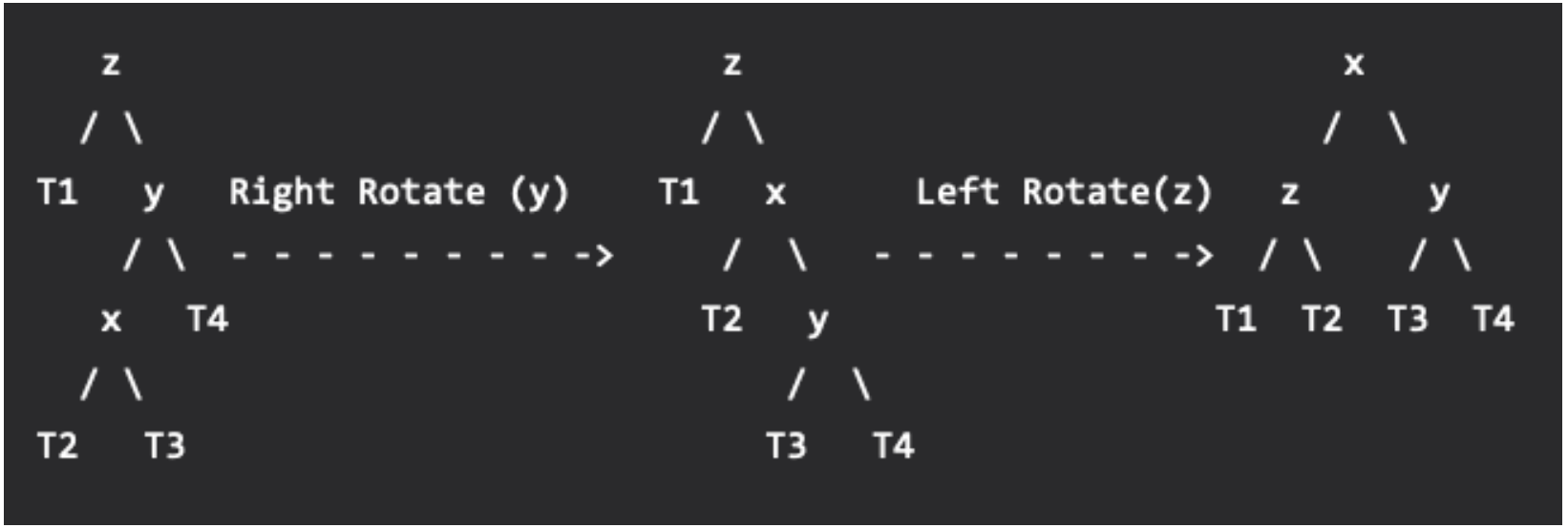
Rebalancing an AVL Tree (Left Right Case)



Rebalancing an AVL Tree (Right Right Case)



Rebalancing an AVL Tree (Right Left Case)



Deletion in an AVL Tree (Code)

STEP 1: PERFORM STANDARD BST DELETE

```
node* delete_node(node* root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = delete_node(root->left, key);
  else if (key > root->key)
    root->right = delete_node(root->right, key);
  else {
    // ...
```

```
    // ...
    // If the node is with only one child or no child
    if (root->left == NULL) {
      node* temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      node* temp = root->left;
      free(root);
      return temp;
    }
    // If the node has two children
    node* temp = next_min_node(root->right);
    // Place the inorder successor in position of the
node to be deleted
    root->key = temp->key;
    // Delete the inorder successor
    root->right = delete_node(root->right, temp->key);
  }
  if (root == NULL) {
      return root;
  }
… continued
```

Deletion in an AVL Tree (Code)

STEP 2: UPDATE HEIGHT OF THE CURRENT NODE

```
root->height = 1 + max(height(root->left),height(root->right));
```

```
int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}
```

Deletion in an AVL Tree (Code)

STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether this node became unbalanced)

```
int balance = getBalance(root);

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);
    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0)
    {
        root->left =  leftRotate(root->left);
        return rightRotate(root);
    }
    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);
    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}
```

```
int getBalance(struct Node *N)
{
   if (N == NULL)
      return 0;
   return height(N->left) –
         height(N->right);
}
```

Complexity and Pros

- The time complexity of AVL delete remains the same as that of BST delete, which is O(h), where h is the tree's height.
- Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL delete is O(Log n).

Advantages Of AVL Trees

- It is always height balanced
- Height is never greater than logN, where N is the number of nodes.
- It outperforms the binary search tree in terms of search performance.
- It has self balancing capabilities

Red-Black Trees

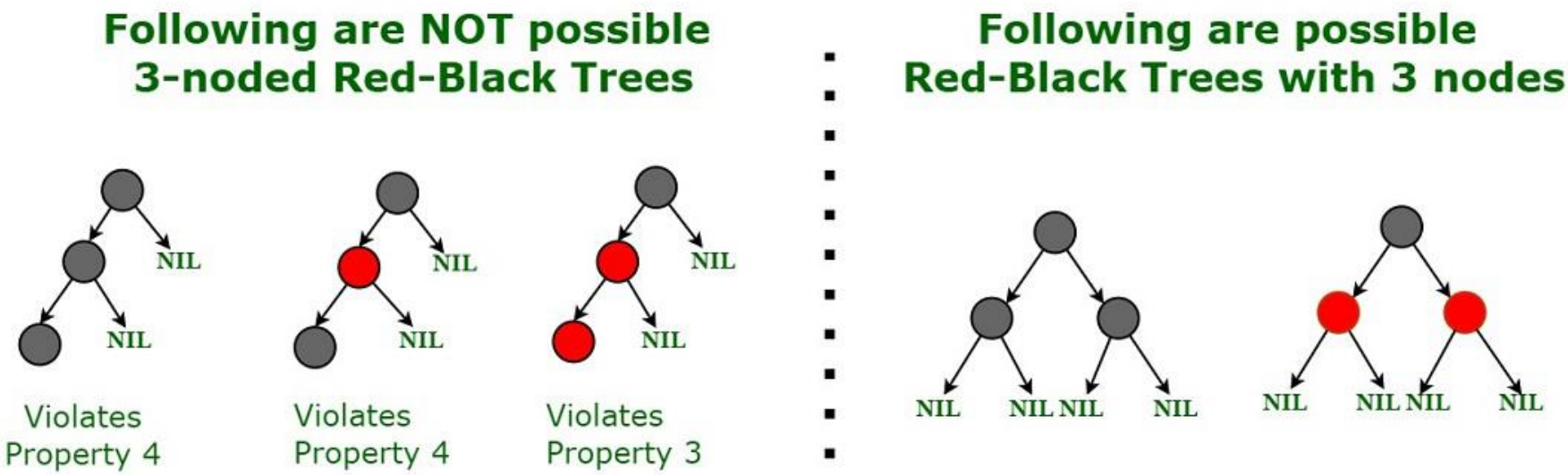
- A self-balancing BST where each node has an additional attribute: a color, which can be either red or black.
- The primary objective is to maintain balance during insertions and deletions

Properties of a Red-Black Tree

1. **Node Color**: Each node is either red or **black**.
2. **Root Property**: The root of the tree is always **black**.
3. **Red Property**: Red nodes cannot have red children (no two consecutive red nodes).
4. **Black Property**: Every path from a node to its descendant null nodes (leaves) has the same number of **black** nodes.
5. **Leaf Property**: All leaves (NIL nodes) are **black**.

These properties ensure that the longest path from the root to any leaf is no more than twice as long as the shortest path, maintaining the tree's balance and efficient performance.

Red-Black Trees



All Possible Structure of a 3-noded Red-Black Tree

Source: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

Red-Black Trees - Insertion

1. BST Insert: Insert the new node like in a standard BST.
2. Fix Violations:
   2. If the parent of the new node is **black**, no properties are violated.
   3. If the parent is **red**, the tree might violate the Red Property, requiring fixes.

<u>Fixing Violations During Insertion</u>
After inserting the new node as a **red** node, we might encounter several cases depending on the colors of the node's parent and uncle (the sibling of the parent):
• **Case 1: Uncle is Red**: Recolor the parent and uncle to **black**, and the grandparent to **red**. Then move up the tree to check for further violations.
• **Case 2: Uncle is Black**:
   - **Sub-case 2.1: Node is a right child**: Perform a left rotation on the parent.
   - **Sub-case 2.2: Node is a left child**: Perform a right rotation on the grandparent and recolor appropriately.

Red-Black Trees - Deletion

1. BST Deletion: Remove the node using standard BST rules.

2. Fix Double Black:

2. If a black node is deleted, a “double black” condition might arise, which requires specific fixes.

<u>Fixing Violations During Deletion</u>

When a black node is deleted, or a red node is replaced by a black node, we handle the double black issue based on the sibling’s color and the colors of its children:

- **Case 1: Sibling is Red**
  - Recolor the sibling and the parent, and perform a rotation.
- **Case 2: Sibling is Black with Black Children**
  - Recolor the sibling to red and move the problem up to the parent.
- **Case 3: Sibling is Black with at least one Red Child**
  - Rotate and recolor to fix the double-black issue.

Demos

Let’s compare them in action:

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