Lecture Notes on Nov/16

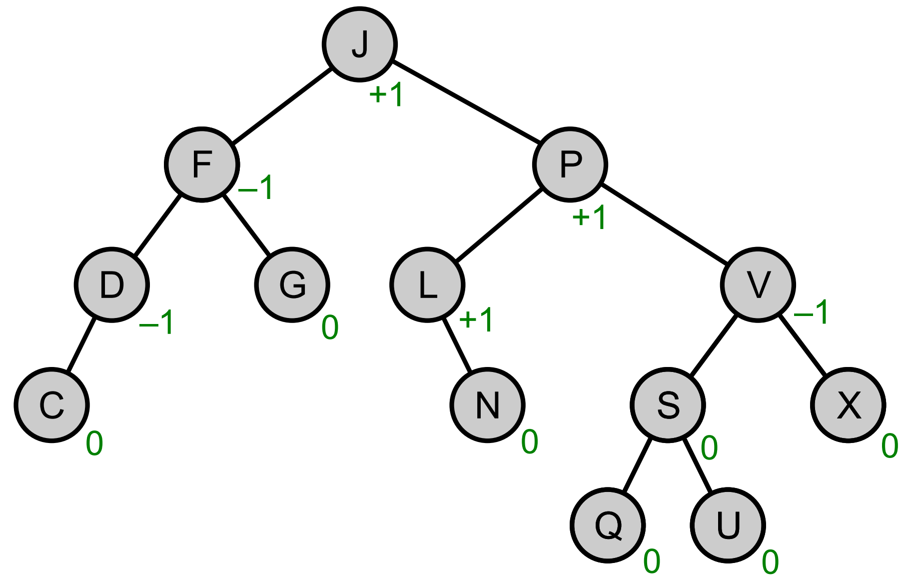
Adelson-Velsky and Landis (AVL) Tree

ECE217 Data Structure and Algorithms

Instructor: Dr. Shayan (Sean) Taheri

AVL Trees

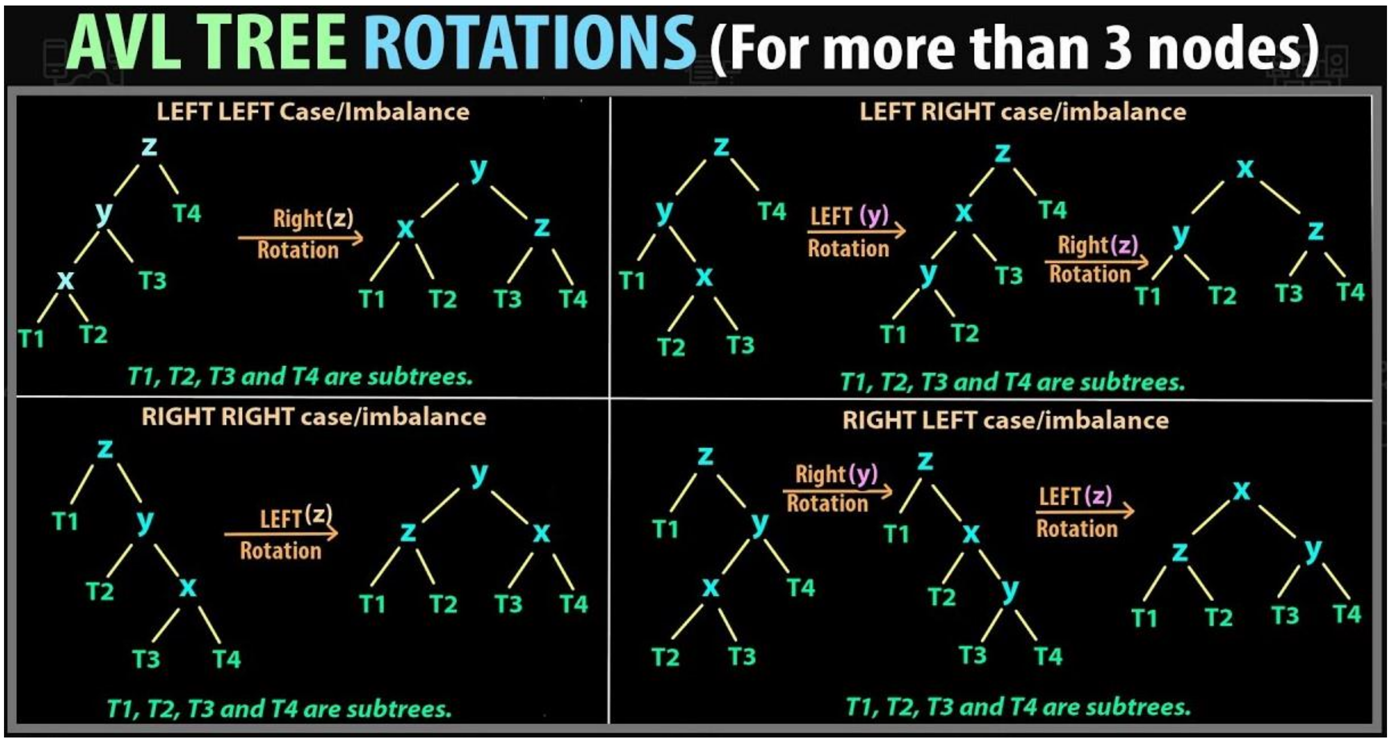
- **Introduction to AVL Tree**
- **Searching in AVL Tree**
- **Insertion in AVL Tree**
- **Rotations in AVL Tree**
- **Deletion in AVL Tree**



AVL Tree with Balance Factors (Green)

- **Animation showing the insertion of several elements into an AVL tree.**
- **It includes left, right, left-right and right-left rotations.**

AVL Trees (Cont.)



AVL tree

Type	Tree
Invented	1962
Invented by	Georgy Adelson-Velsky and Evgenii Landis

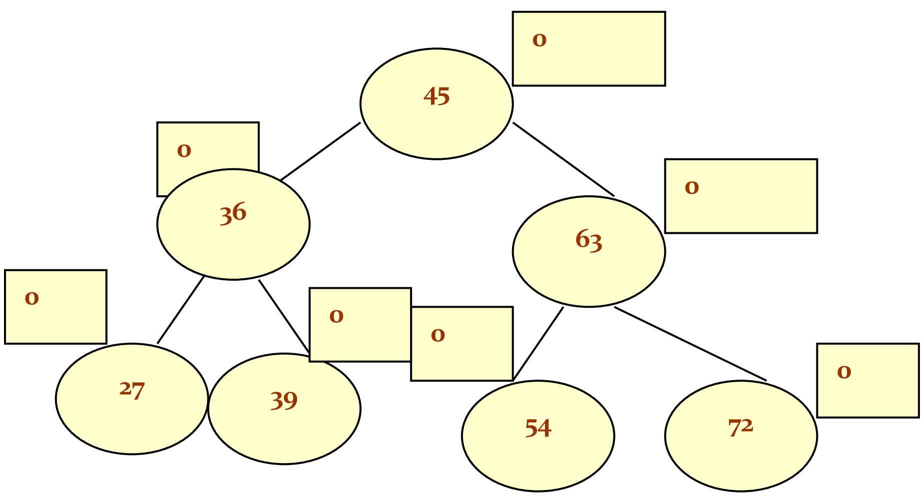
Time complexity in big O notation

Algorithm	Average	Worst case
Space	$\Theta(n)$	$O(n)$
Search	$\Theta(\log n)$ [1]	$O(\log n)$ [1]
Insert	$\Theta(\log n)$ [1]	$O(\log n)$ [1]
Delete	$\Theta(\log n)$ [1]	$O(\log n)$ [1]

AVL Trees

- AVL tree (a.k.a. **height-balanced tree**) is a self-balancing binary search tree in which the heights of the two sub-trees of a node may differ by at most one.
- AVL Tree Height → **O(logn)** = Average Time for search, insertion and deletion.
- **Balance factor = Height (left sub-tree) – Height (right sub-tree)**
- A binary search tree in which every node has a balance factor of **-1, 0, or +1** is said to be height balanced.

Balanced AVL Tree → Balance Factor = 0



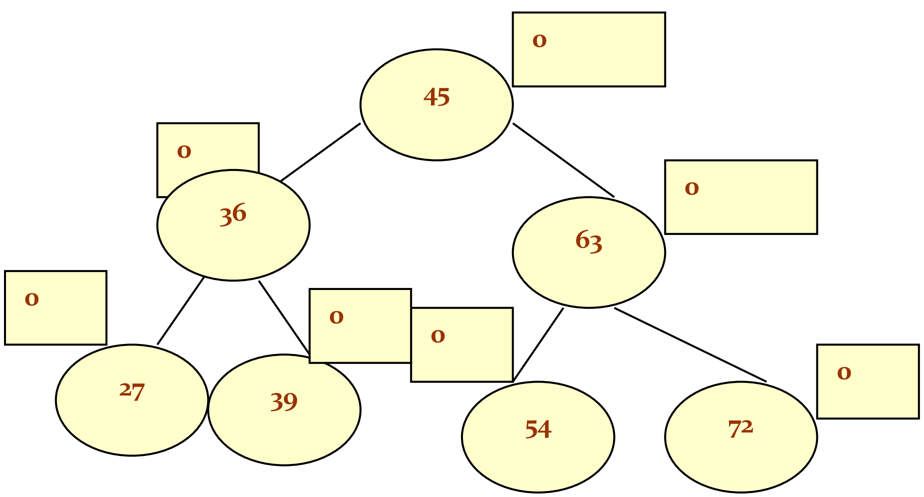
Left Heavy AVL Tree → Balance Factor = +1 (Exercise)

Right Heavy AVL Tree → Balance Factor = -1 (Exercise)

Searching for a Node in an AVL Tree

- Since an AVL tree is also a variant of binary search tree, searching is also done in the same way as it is done in case of a binary search tree.
- The operation does not modify the structure of the tree, no special provisions need to be taken.

Searching in Balanced AVL Tree



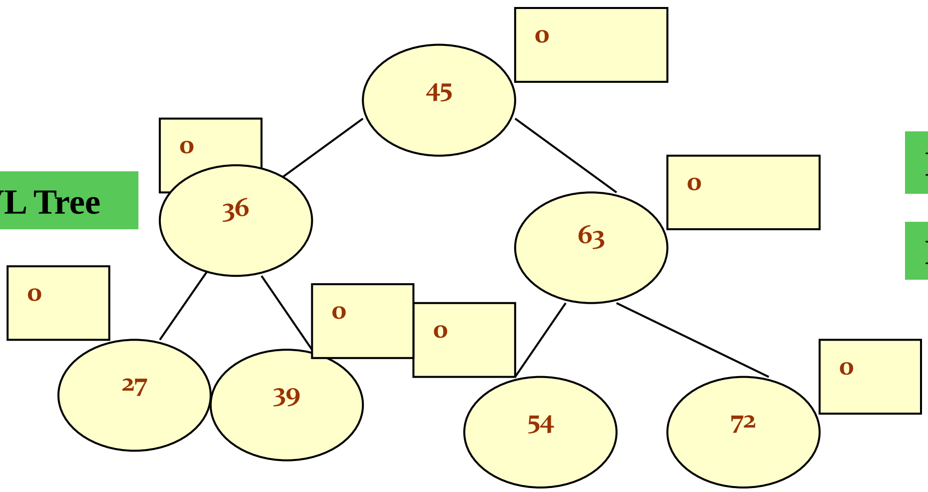
Searching in Left Heavy AVL Tree (Exercise)

Searching in Right Heavy AVL Tree (Exercise)

Inserting a Node in an AVL Tree

- The new node is always inserted as the leaf node.
- But the step of insertion is usually followed by an additional step of rotation.
- Rotation is done to restore the balance of the tree, if the balance factor of every node is not equal to **-1, 0, or +1**.
- The nodes whose balance factors will change are those which lie on the path between the root of the tree and the newly inserted node.
- **Critical node is the nearest ancestor node on the path from the root to the newly inserted node whose balance factor is neither -1, 0 nor 1**.
    - **Creating an "unbalanced sub-tree"**.

Inserting in Balanced AVL Tree

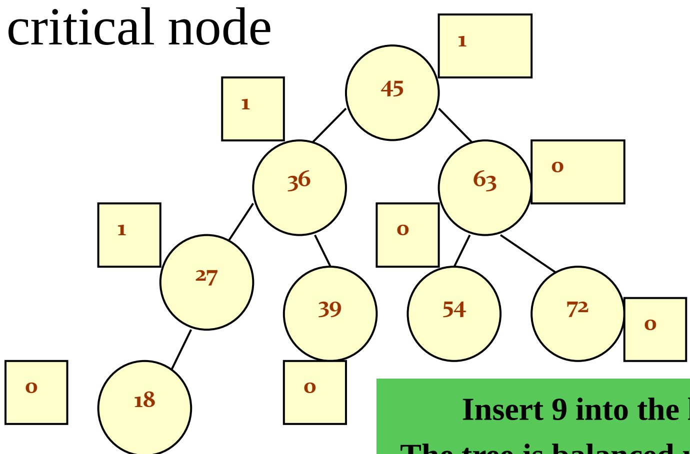


Inserting in Left Heavy AVL Tree (Exercise)

Inserting in Right Heavy AVL Tree (Exercise)

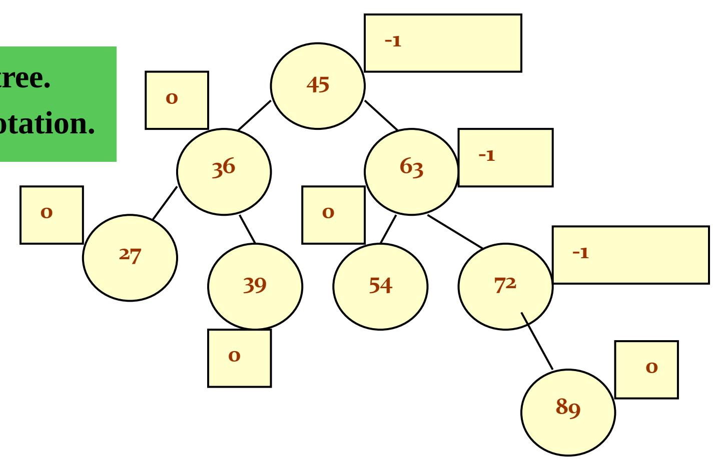
Rotations to Balance AVL Trees

- **Task 1**: Finding the **critical node**.
- **Task 2**: Determining one of four types of rebalancing rotation to be done.
  - It depends on the position of the inserted node with reference to the critical node.
  - **LL Rotation**: The new node is inserted in the left sub-tree of the left sub-tree of the critical node
  - **RR Rotation**: The new node is inserted in the right sub-tree of the right sub-tree of the critical node
  - **LR Rotation**: The new node is inserted in the right sub-tree of the left sub-tree of the critical node
  - **RL Rotation**: The new node is inserted in the left sub-tree of the right sub-tree of the critical node



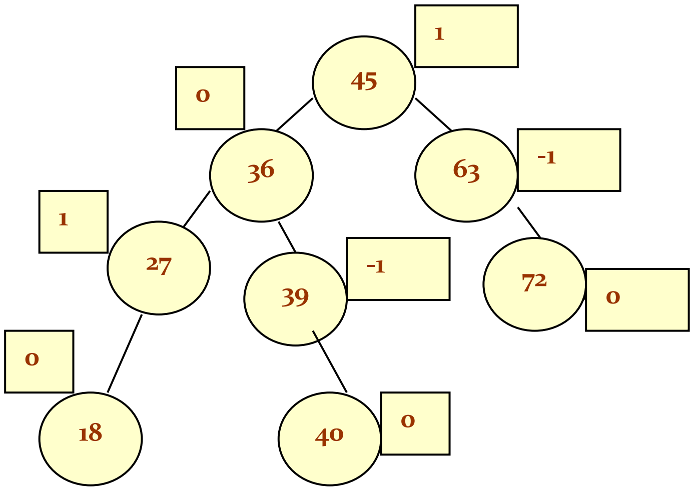
Insert 9 into the left AVL tree.
The tree is balanced using LL rotation.

Insert 91 into the right AVL tree.
The tree is balanced using RR rotation.

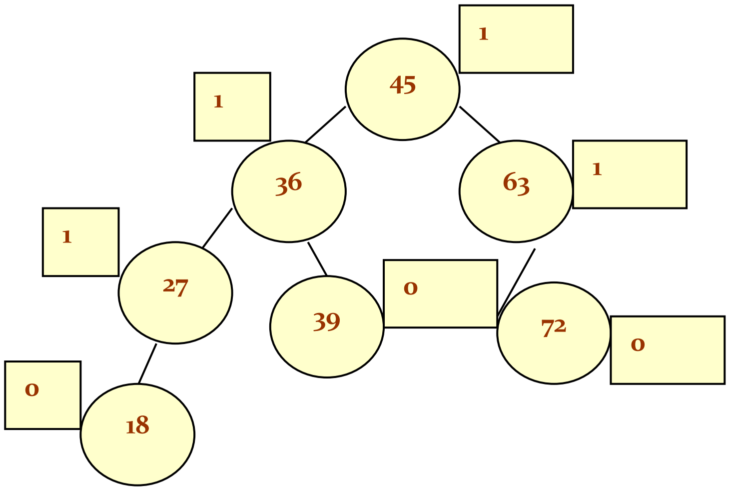


Deleting a Node from an AVL Tree

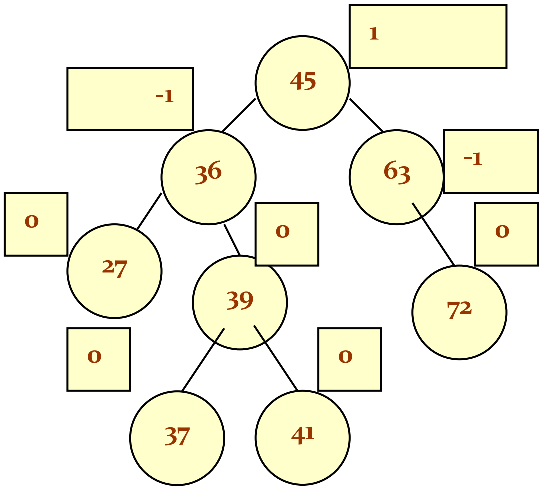
- There are two classes of rotation that can be performed on an AVL tree after deleting a given node for rebalancing: **R Rotation** and **L Rotation**.
- If the node to be deleted is present in the left sub-tree of the critical node, then **L Rotation** is applied else if node is in the right sub-tree, **R Rotation** is performed.
- **L Rotation Variations**: **L-1, L0, and L+1** rotations.
- **R Rotation Variations**: **R-1, R0, and R+1** rotations.



Left AVL Tree:
R0 Rotation – Deleting "72"



Center AVL Tree:
R+1 Rotation – Deleting "72"



Right AVL Tree:
R-1 Rotation – Deleting "72"

AVL Tree Operations in C++ Language

```cpp
// Adelson-Velsky and Landis (AVL) Tree Operations in C++ Language
// Instructor: Dr. Shayan (Sean) Taheri

#include <iostream>
using namespace std;

// AVL Tree Node Class
class Node {
  public:
    int key;
    Node *left;
    Node *right;
    int height;
};

// Function Declaration of "Getting Maximum Value"
int max(int a, int b);

// Height Calculation for AVL Tree
int height(Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

// Function Definition of "Getting Maximum Value"
int max(int a, int b) {
  return (a > b) ? a : b;
}
```

AVL Tree Operations in C++ Language (Cont.)

```cpp
// Node Creation for AVL Tree
Node *newNode(int key) {
  Node *node = new Node();
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
}

// Rotate to Right Side for AVL Tree
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  return x;
}
```

AVL Tree Operations in C++ Language (Cont.)

```
// Rotate to Left Side for AVL Tree
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  return y;
}

// Getting Balance Factor of Each AVL Tree Node
int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
      height(N->right);
}
```

AVL Tree Operations in C++ Language (Cont.)

```
// AVL Node Insertion
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // (1) Updating Balance Factor of Each AVL Tree Node
  // (2) Balancing AVL Tree
  node->height = 1 + max(height(node->left),
             height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
      return rightRotate(node);
    } else if (key > node->left->key) {
      node->left = leftRotate(node->left);
      return rightRotate(node);
    }
  }
  if (balanceFactor < -1) {
    if (key > node->right->key) {
      return leftRotate(node);
    } else if (key < node->right->key) {
      node->right = rightRotate(node->right);
      return leftRotate(node);
    }
  }
  return node;
}
```

AVL Tree Operations in C++ Language (Cont.)

```
// Getting AVL Tree Node with Minimum Value
Node *nodeWithMimumValue(Node *node) {
  Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}

// AVL Node Deletion
Node *deleteNode(Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    if ((root->left == NULL) ||
      (root->right == NULL)) {
      Node *temp = root->left ? root->left : root->right;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      Node *temp = nodeWithMimumValue(root->right);
      root->key = temp->key;
      root->right = deleteNode(root->right,
                   temp->key);
    }
  }

  if (root == NULL)
    return root;
```

AVL Tree Operations in C++ Language (Cont.)

```cpp
  // (1) Updating Balance Factor of Each AVL Tree Node
  // (2) Balancing AVL Tree
  root->height = 1 + max(height(root->left),
                         height(root->right));
  int balanceFactor = getBalanceFactor(root);
  if (balanceFactor > 1) {
    if (getBalanceFactor(root->left) >= 0) {
      return rightRotate(root);
    } else {
      root->left = leftRotate(root->left);
      return rightRotate(root);
    }
  }
  if (balanceFactor < -1) {
    if (getBalanceFactor(root->right) <= 0) {
      return leftRotate(root);
    } else {
      root->right = rightRotate(root->right);
      return leftRotate(root);
    }
  }
  return root;
}

// Printing AVL Tree
void printTree(Node *root, string indent, bool last) {
  if (root != nullptr) {
    cout << indent;
    if (last) {
      cout << "R----";
      indent += "   ";
    } else {
      cout << "L----";
      indent += "|  ";
    }
    cout << root->key << endl;
    printTree(root->left, indent, false);
    printTree(root->right, indent, true);
  }
}
```

AVL Tree Operations in C++ Language (Cont.)

```
// Driver Code
int main() {

    // Task 1: Create an AVL Tree Node

    // Task 2: Execute AVL Tree Operations on the Created Tree

}
```

Assignment

- Reading Assignment:
  - Data Structures Using C by Reema Thareja, Oxford University Press; 2nd Edition.
    - Chapter 10. Efficient Binary Trees (Starting Page: 298).
  - AVL Tree in Wikipedia.
- **Assignment 2 – Part B** Deadline: **November/21/2022**.

Questions?