Lecture 1: AVL Trees

Authors: Refat Othman and Diaeddin Rimawi

What is an AVL Tree?

An **AVL tree** (named after **Adelson-Velsky and Landis**) is a self-balancing binary search tree (BST). For every node in the AVL tree, the height difference (also called the **balance factor**) between its left and right subtrees is **at most** 1**.

- **Balanced tree**: A tree where, for each node, the height difference between the left and right subtrees is → 1.
- **Balance Factor** = height(left subtree) - height(right subtree)

Why AVL Trees?

In regular BSTs, if data is inserted in a sorted manner, the tree can become skewed and degenerate into a linked list. AVL trees ensure that the height remains → log(n), ensuring optimal performance for search, insert, and delete operations.

Rotations in AVL Trees

To maintain balance after insertion or deletion, the AVL tree may need to perform **rotations**. There are four possible cases:

Case 1: Single Right Rotation (Left-Left case)

Example 1 (Basic):

Insert: 60 → 50 → 20

Step-by-step:

```
Insert 60:
    60

Insert 50:
    60
   /
  50

Insert 20:
    60
   /
  50
```

```
   /
  20
```

After right rotation at 60:

```
     50
    /  \
   20   60
```

Example 2 (Complex):

Insert: 70 → 60 → 80 → 50 → 40 (left-side insertions that create imbalance)

Step-by-step:

```
Insert 70:
    70

Insert 60:
    70
   /
  60

Insert 80:
    70
   /  \
  60   80

Insert 50:
    70
   /  \
  60   80
 /
50

Insert 40:
    70
   /  \
  60   80
 /
50
/
40
```

Unbalanced at 70 → Left-Left case → Right rotation at 70

```
      60
     /  \
   50    70
  /        \
40          80
```

Case 2: Single Left Rotation (Right-Right case)

Example 1 (Basic):

Insert: 80 → 90 → 100

Step-by-step:

```
Insert 80:
    80

Insert 90:
    80
      \
       90

Insert 100:
    80
      \
       90
         \
         100
```

After left rotation at 80:

```
    90
   /  \
  80  100
```

Example 2 (Complex):

Insert: 50 → 60 → 40 → 70 → 80

Step-by-step:

```
Insert 50:
    50

Insert 60:
    50
```

```
      \
       60

Insert 40:
    50
   /  \
  40   60

Insert 70:
    50
   /  \
  40   60
         \
          70

Insert 80:
    50
   /  \
  40   60
         \
          70
            \
             80
```

Tree is imbalanced after inserting 80, requiring a left rotation at node 50 to restore balance

After left rotation at 50:

```
     60
    /  \
  50    70
 /        \
40         80
```

Case 3: Double Rotation (Right-Left case)

Example 1 (Basic):

Insert: 60 → 80 → 70

Step-by-step:

```
Insert 60:
    60

Insert 80:
    60
      \
```

```
      80

Insert 70:
    60
     \
      80
     /
   70
```

Step 1: Right rotation on 80:

```
    60
     \
      70
        \
         80
```

Step 2: Left rotation on 60:

```
   70
  /  \
 60   80
```

Example 2 (Complex):

Insert: 30 → 10 → 60 → 40 → 70 → 35

Step-by-step:

```
Insert 30:
    30
```

```
Insert 10:
    30
   /
 10
```

```
Insert 60:
    30
   /  \
 10   60
```

```
Insert 40:
    30
   /  \
 10   60
      /
    40
```

```
Insert 70:
    30
   /  \
 10   60
      /  \
    40    70
```

```
Insert 35:
    30
   /  \
 10   60
      /  \
    40    70
   /
 35
```

Unbalanced at 30 → Right-Left case

Step 1: Right rotation on 60:

```
    30
   /  \
 10   40
        /  \
      35    60
              \
               70
```

Step 2: Left rotation on 30:

```
       40
      /  \
    30    60
   /  \     \
 10  35     70
```

Case 4: Double Rotation (Left-Right case)

Example 1 (Basic):

Insert: 60 → 20 → 40

Step-by-step:

```
Insert 60:
    60

Insert 20:
    60
   /
  20

Insert 40:
    60
   /
  20
    \
     40
```

Step 1: Left rotation on 20:

```
    60
   /
  40
 /
20
```

Step 2: Right rotation on 60:

```
   40
  /  \
 20   60
```

Example 2 (Complex):

Insert: 50 → 70 → 30 → 20 → 40 → 45

Step-by-step:

```
Insert 50:
    50
```

```
Insert 70:
    50
      \
       70
```

```
Insert 30:
    50
   /  \
 30    70
```

```
Insert 20:
    50
   /  \
 30    70
 /
20
```

```
Insert 40:
    50
   /  \
 30    70
 / \
20  40
```

```
Insert 45:
    50
   /  \
 30    70
 / \
20  40
        \
         45
```

Unbalanced at 50 → Left-Right case

Step 1: Left rotation on 30’s right child (node 40):

```
       50
      /  \
```

```
      40    70
      / \
     30  45
    /
   20
```

Here, node 40 becomes the new right child of 30, and 45 becomes its right child.

Step 2: Right rotation on node 50:

```
       40
      /  \
    30    50
   /     /  \
  20    45   70
```

Scenarios for Deleting a Value from an AVL Tree

Deleting a node from an AVL tree follows the same logic as BST deletion but is followed by rebalancing. There are three main scenarios:

1. Deleting a Leaf Node

Example: Insert: 40 → 30 → 50

```
      40
     /  \
    30   50
```

Delete 50:

```
      40
     /
    30
```

No imbalance — tree remains balanced.

2. Deleting a Node with One Child

Example: Insert: 40 → 30 → 60 → 50

```
      40
     /  \
    30   60
```

```
         /
       50
```

Delete 60:

```
      40
     /  \
    30    50
```

No imbalance — tree remains balanced.

3. Deleting a Node with Two Children

Example: Insert: 50 → 30 → 70 → 20 → 40 → 60 → 80

```
          50
         /  \
       30    70
      /  \   /  \
    20   40 60   80
```

Delete 30:

- Replace 30 with its in-order predecessor (20) or successor (40)

```
          50
         /  \
       40    70
      /     /  \
    20    60   80
```

No imbalance — tree remains balanced.

4. Deleting a Node That Causes Imbalance (and Requires Rotation)

Example: Insert: 50 → 30 → 70 → 20 → 40 → 60 → 80 → 10

```
          50
         /  \
       30    70
      /  \   /  \
    20   40 60   80
   /
  10
```

Delete 80:

```
        50
       /  \
     30    70
    /  \   /
  20   40 60
 /
10
```

Delete 60:

```
        50
       /  \
     30    70
    /  \
  20   40
 /
10
```

Now the tree is unbalanced at node 50:

- Left subtree height = 3 (via 30 → 20 → 10)
- Right subtree height = 1 (node 70)

Right rotation is needed at node 50

Step 1: Left rotation on node 30's right child is not needed (no Right-Left case). Step 2: Perform Right rotation at 50:

```
       30
      /  \
    20    50
   /     /  \
 10     40   70
```

Tree is balanced now.

AVL Tree Code in Python

```
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
```

```
        self.right = None
        self.height = 1

class AVLTree:
    def get_height(self, node):
        return node.height if node else 0

    def get_balance(self, node):
        return self.get_height(node.left) - self.get_height(node.right) if node else 0

    def rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = max(self.get_height(y.left), self.get_height(y.right)) + 1
        x.height = max(self.get_height(x.left), self.get_height(x.right)) + 1
        return x

    def rotate_left(self, x):
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        x.height = max(self.get_height(x.left), self.get_height(x.right)) + 1
        y.height = max(self.get_height(y.left), self.get_height(y.right)) + 1
        return y

    def rebalance(self, node):
        balance = self.get_balance(node)
        if balance > 1:
            if self.get_balance(node.left) < 0:
                node.left = self.rotate_left(node.left)
            return self.rotate_right(node)
        if balance < -1:
            if self.get_balance(node.right) > 0:
                node.right = self.rotate_right(node.right)
            return self.rotate_left(node)
        return node

    def insert(self, root, key):
        if not root:
            return AVLNode(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)
        root.height = max(self.get_height(root.left), self.get_height(root.right)) + 1
        return self.rebalance(root)

    def find_min(self, node):
        while node.left:
```

```python
            node = node.left
        return node

    def delete(self, root, key):
        if not root:
            return root
        elif key < root.key:
            root.left = self.delete(root.left, key)
        elif key > root.key:
            root.right = self.delete(root.right, key)
        else:
            if not root.left:
                return root.right
            elif not root.right:
                return root.left
            temp = self.find_min(root.right)
            root.key = temp.key
            root.right = self.delete(root.right, temp.key)
        root.height = max(self.get_height(root.left), self.get_height(root.right)) + 1
        return self.rebalance(root)
```

AVL vs. BST

Example Insertion Sequence: 60, 50, 20, 80, 90, 70, 55, 10, 40, 35

- The resulting AVL tree remains balanced after each insertion through rotations.
- The resulting plain BST becomes unbalanced and degraded into a less efficient form.

Visualizations for each step can be added using a diagramming tool or whiteboard illustrations in class.

Lecture 2: AVL Tree Exercises

Exercise 1: Validate AVL Tree

Given a binary tree, check whether it satisfies the AVL property (balance factor → 1 for every node).

Exercise 2: Range Search in AVL Tree

Implement a function `range_search(root, low, high)` that returns all values in the AVL tree between `low` and `high` (inclusive). Demonstrate how the function uses the AVL property to efficiently skip unnecessary branches. Test the function with a tree containing: 40, 20, 60, 10, 30, 50, 70.

Expected output for `range_search(root, 25, 65)` should be `[30, 40, 50, 60]`.

Exercise 3: Find k-th Smallest Element in AVL Tree

Write a function that returns the k-th smallest element in the AVL tree.

Test your implementation by inserting the following values: [50, 30, 70, 20, 40, 60, 80], and:

- Return the 4th smallest element (expected output: 50)