AVL Trees

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

Objectives

- To know what an ***AVL tree*** is
- To understand how to ***rebalance*** a tree using the ***LL rotation***, ***LR rotation***, ***RR rotation***, and ***RL rotation***
- To know how to design the **`AVLTree`** class
- To ***insert*** elements into an AVL tree
- To implement node ***rebalancing***
- To ***delete*** elements from an AVL tree
- To implement the **`AVLTree`** class
- To test the **`AVLTree`** class
- To analyze the ***complexity*** of search, insert, and delete operations in AVL trees

(c) Paul Fodor (CS Stony Brook) & Pearson

Why AVL Tree?

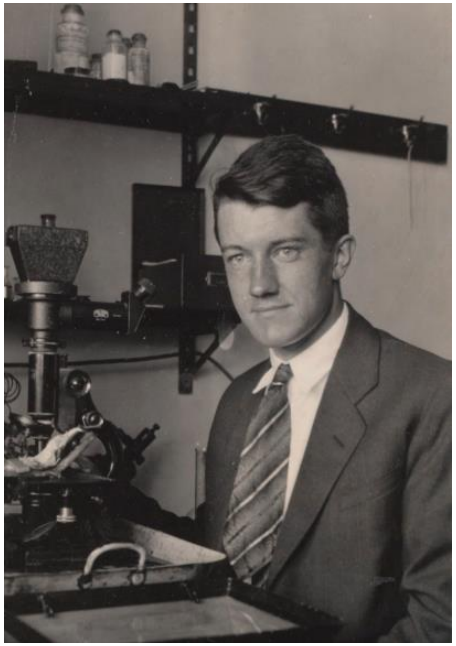
- The search, insertion, and deletion time for a binary search tree is dependent on the **height of the tree**
  - In the worst case, the height is **O(n)**
- If a tree is ***perfectly balanced***, i.e., a **complete binary tree**, its height is **log n**
- Can we maintain a perfectly balanced tree?
  - Yes, but it will be costly to do so
  - The compromise is to maintain a ***well-balanced tree***, i.e., the heights of two subtrees for every node are about the same

(c) Paul Fodor (CS Stony Brook) & Pearson

What are AVL Trees?

- ***AVL trees*** are well-balanced binary search trees were invented by two Russian computer scientists Georgy Adelson-Velsky and Evgenii Landis in 1962 at the Institute for Theoretical and Experimental Physics in Moscow
  - In an AVL tree, the difference between the heights of two subtrees for every node is **0** or **1**
  - The **maximum height** of an AVL tree is **O(log n)**

Adelson-Velsky developed Kaissa, the first computer chess champion (Stockholm 1974)



(c) Paul Fodor (CS Stony Brook) & Pearson

Balance Factor/Left-Heavy/Right-Heavy

- The process for inserting or deleting an element in an AVL tree is the same as in a regular binary search tree
  - The difference is that you may have to ***rebalance*** the tree after an insertion or deletion operation
- The ***balance factor*** of a node is the height of its right subtree minus the height of its left subtree
- A node is said to be ***balanced*** if its balance factor is -1, 0, or 1
  - A node is said to be ***left-heavy*** if its balance factor is -1
  - A node is said to be ***right-heavy*** if its balance factor is +1

(c) Paul Fodor (CS Stony Brook) & Pearson

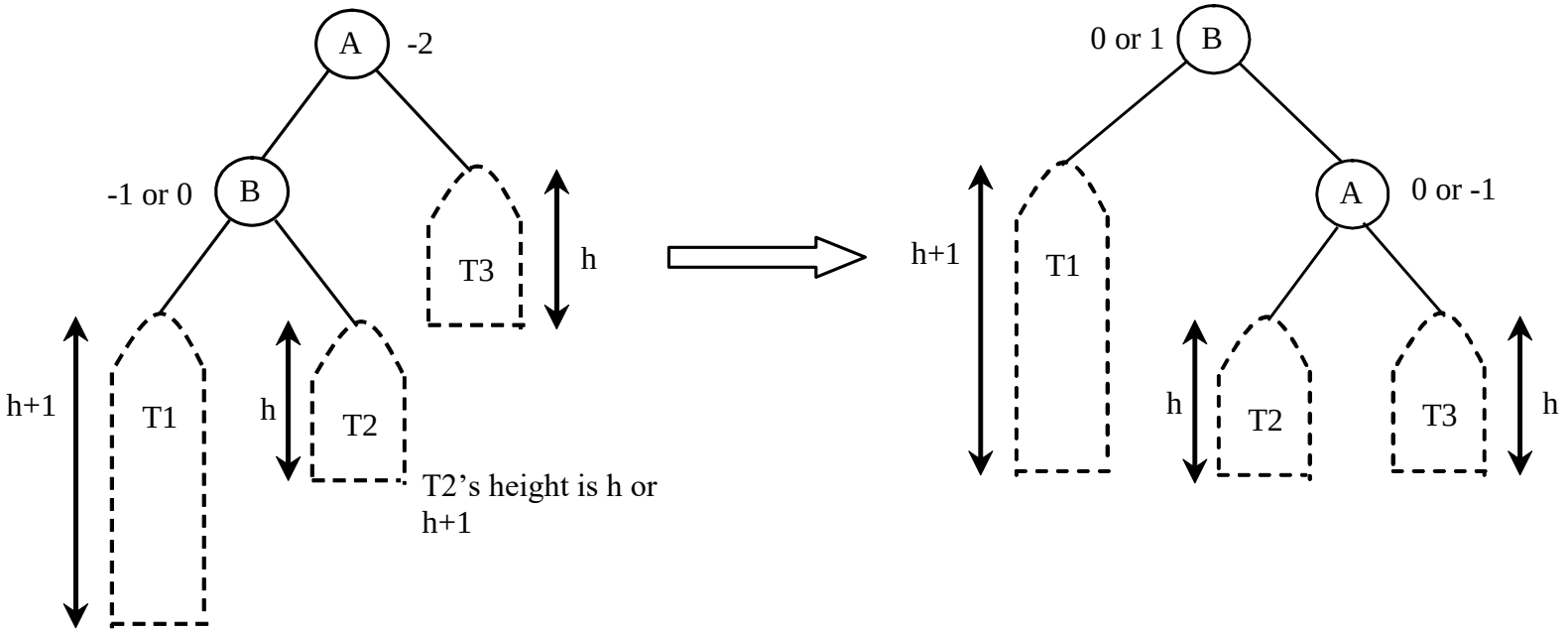
Balancing Trees

- If a node is **not balanced** after an insertion or deletion operation, you need to rebalance it:
  - The process of rebalancing a node is called a ***rotation***
- There are four possible rotations:
  - LL rotation (***left-heavy,left-heavy rotation***)
  - RR rotation (***right-heavy,right-heavy rotation***)
  - LR rotation (***left-heavy,right-heavy rotation***)
  - RL rotation (***right-heavy,left-heavy rotation***)

(c) Paul Fodor (CS Stony Brook) & Pearson

LL imbalance and LL rotation

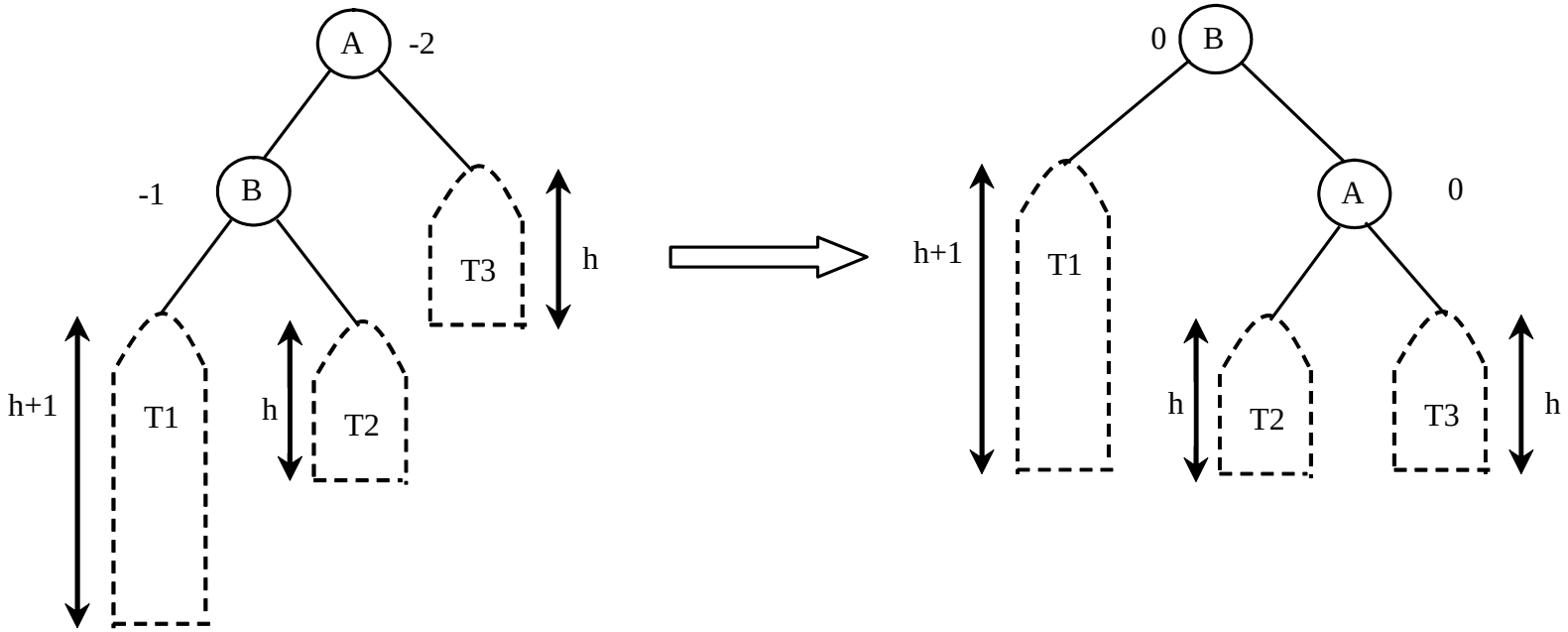
- **LL Rotation**: An ***left-heavy, left-heavy imbalance*** occurs at a node **A** if **A** has a balance factor **-2 (left-heavy)** and a left child **B** has a balance factor **-1 (left-heavy)** or **0**
  - This type of imbalance can be fixed by performing a single **right rotation** at **A**



(c) Paul Fodor (CS Stony Brook) & Pearson

LL imbalance and LL rotation

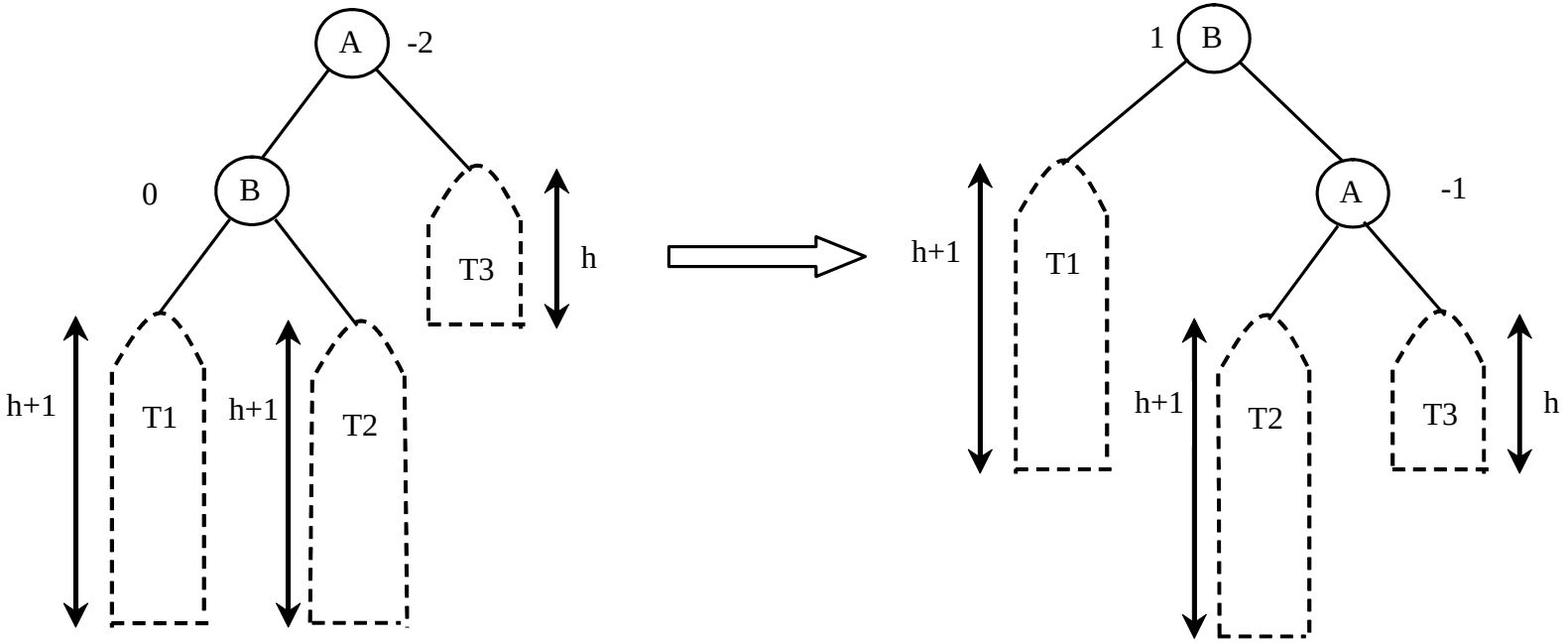
- **LL case 1:** If the left child **B** has a balance factor **-1**



(c) Paul Fodor (CS Stony Brook) & Pearson

LL imbalance and LL rotation

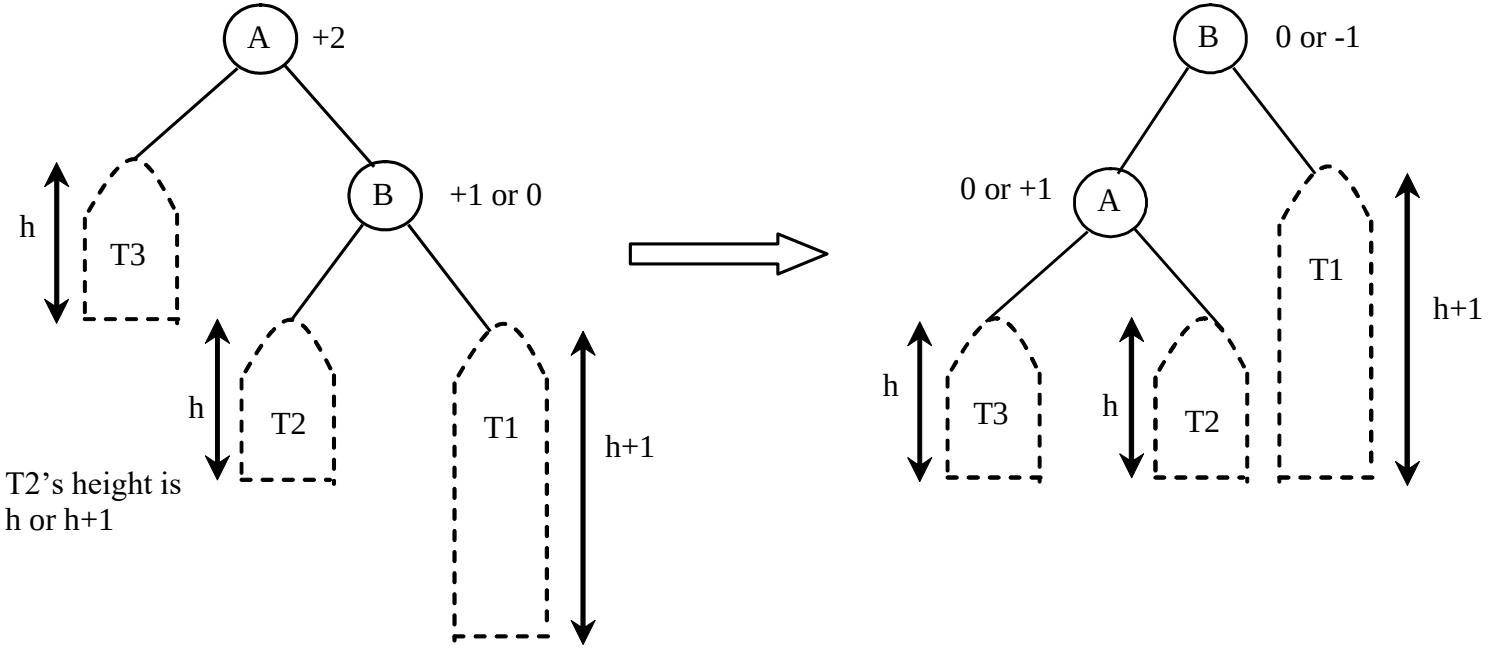
- **LL case 2:** If the left child **B** has a balance factor of **0**



(c) Paul Fodor (CS Stony Brook) & Pearson

RR imbalance and RR rotation

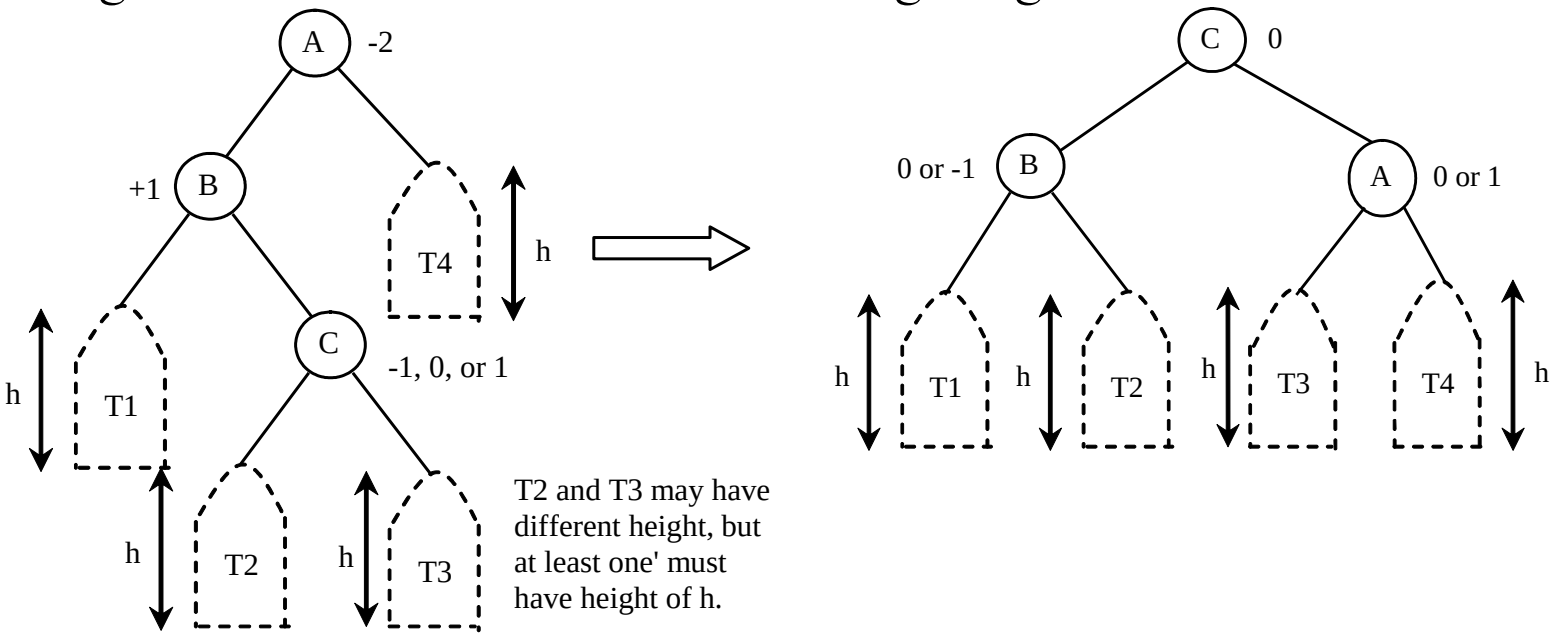
- **RR Rotation**: An ***RR imbalance*** occurs at a node **A** if **A** has a balance factor **+2 (right-heavy)** and a right child **B** has a balance factor **+1 (right-heavy)** or **0**
  - This type of imbalance can be fixed by performing a single **left rotation** at **A**:



(c) Paul Fodor (CS Stony Brook) & Pearson

LR imbalance and LR rotation

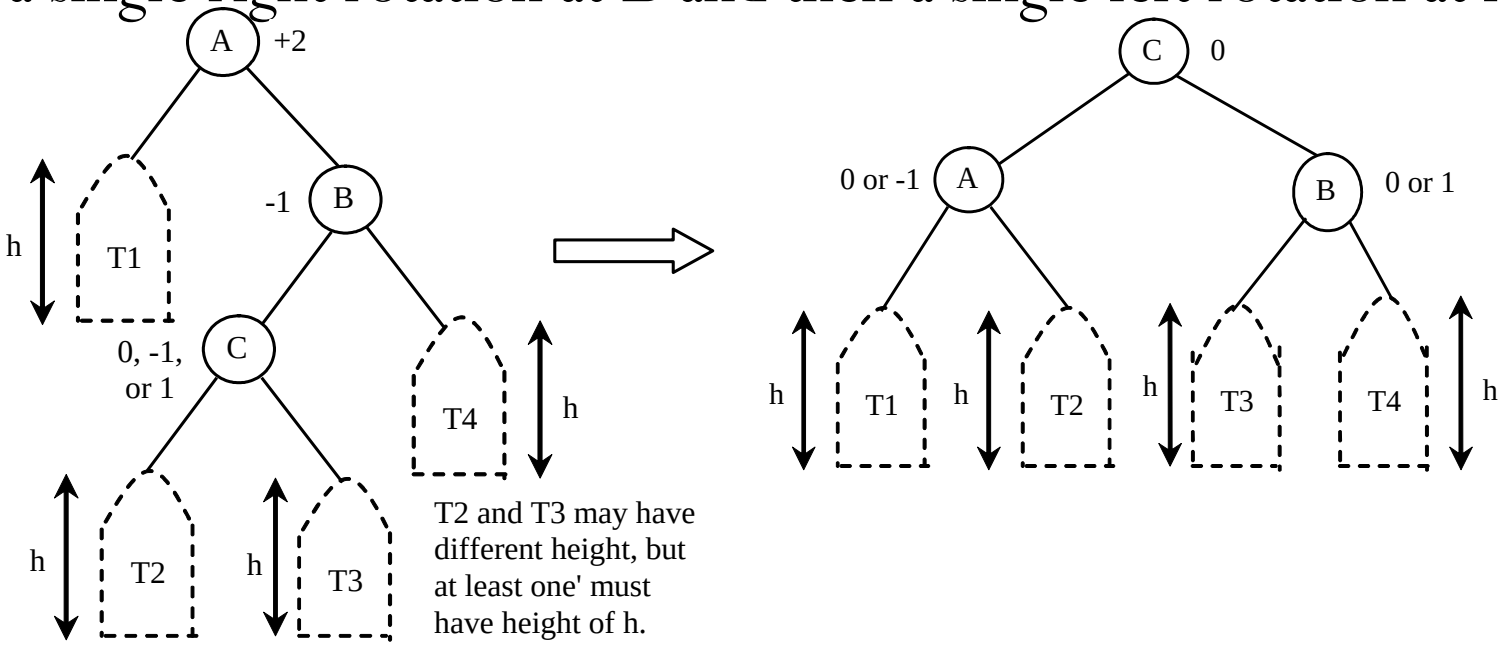
- **LR Rotation**: An ***LR imbalance*** occurs at a node **A** if **A** has a balance factor **-2 (left-heavy)** and a left child **B** has a balance factor **+1 (right-heavy)**
  - Assume **B**'s right child is **C**
  - This type of imbalance can be fixed by performing a double rotation: first a single left rotation at **B** and then a single right rotation at **A**



(c) Paul Fodor (CS Stony Brook) & Pearson

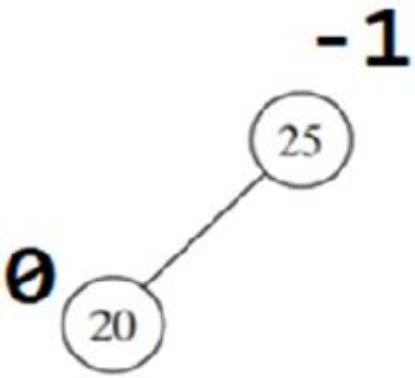
RL imbalance and RL rotation

- **RL Rotation**: An ***RL imbalance*** occurs at a node **A** if **A** has a balance factor **+2 (right-heavy)** and a right child **B** has a balance factor **-1 (left-heavy)**
  - Assume B's left child is C
  - This type of imbalance can be fixed by performing a double rotation: first a single right rotation at **B** and then a single left rotation at **A**

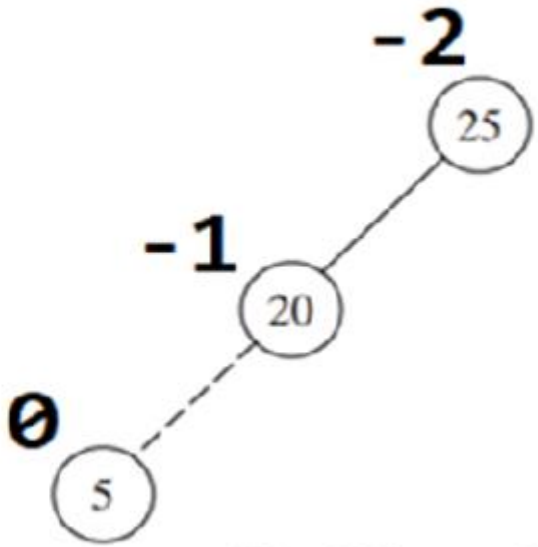


(c) Paul Fodor (CS Stony Brook) & Pearson

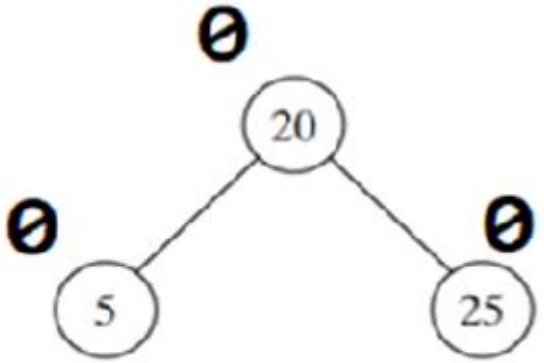
Insert 25, 20



Insert 5

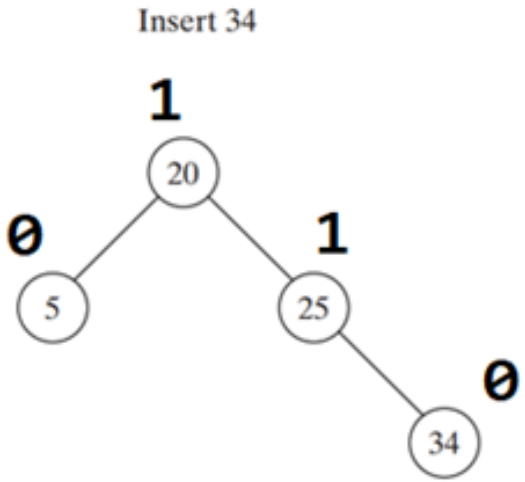


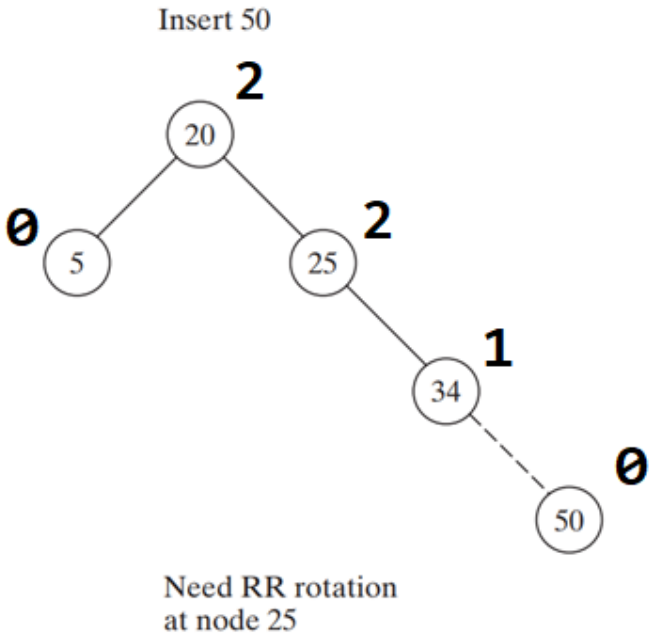
Need LL rotation
at node 25

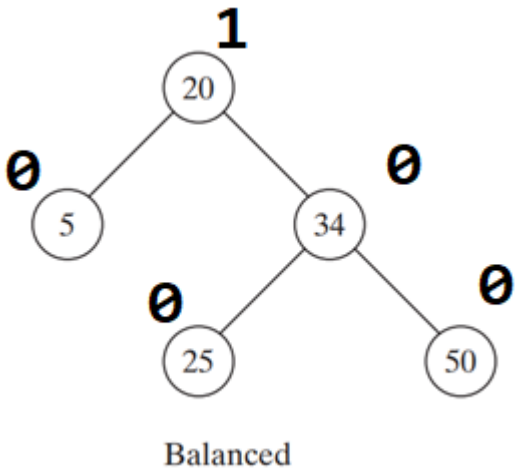


Balanced

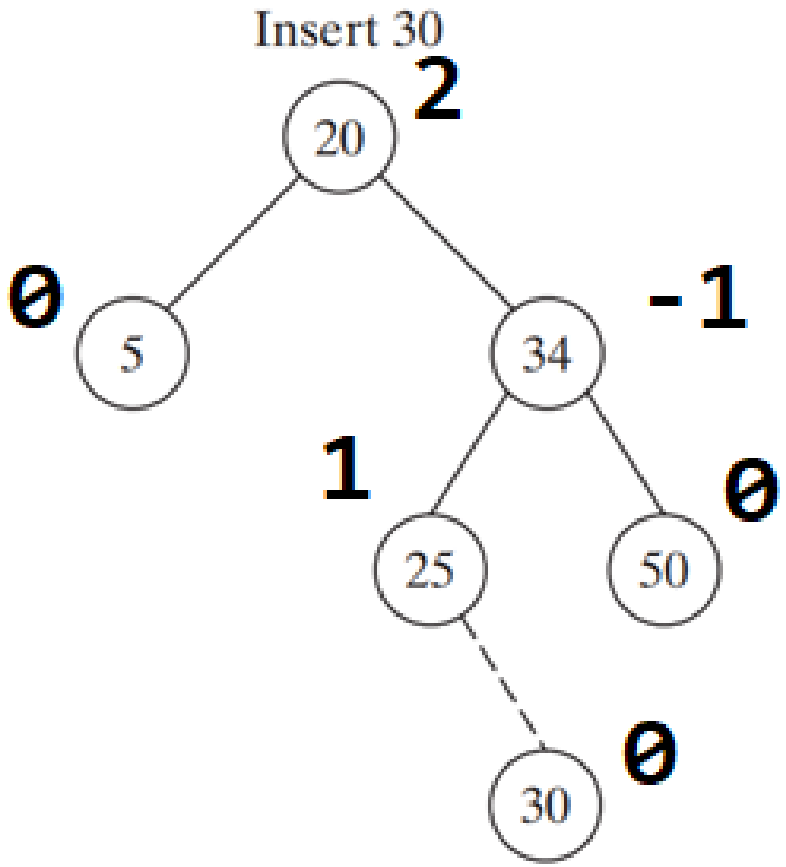
(c) Paul Fodor (CS Stony Brook) & Pearson



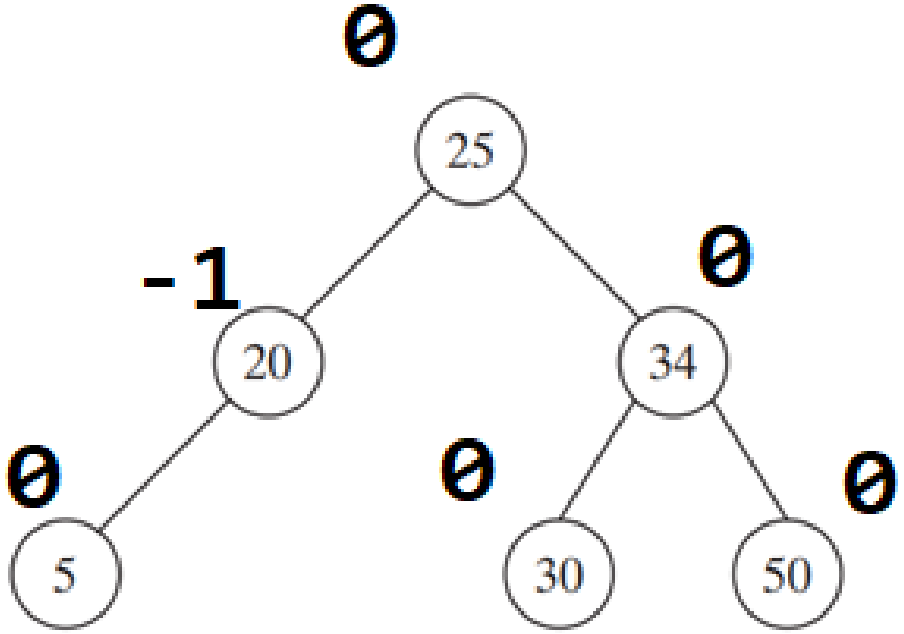




(c) Paul Fodor (CS Stony Brook) & Pearson

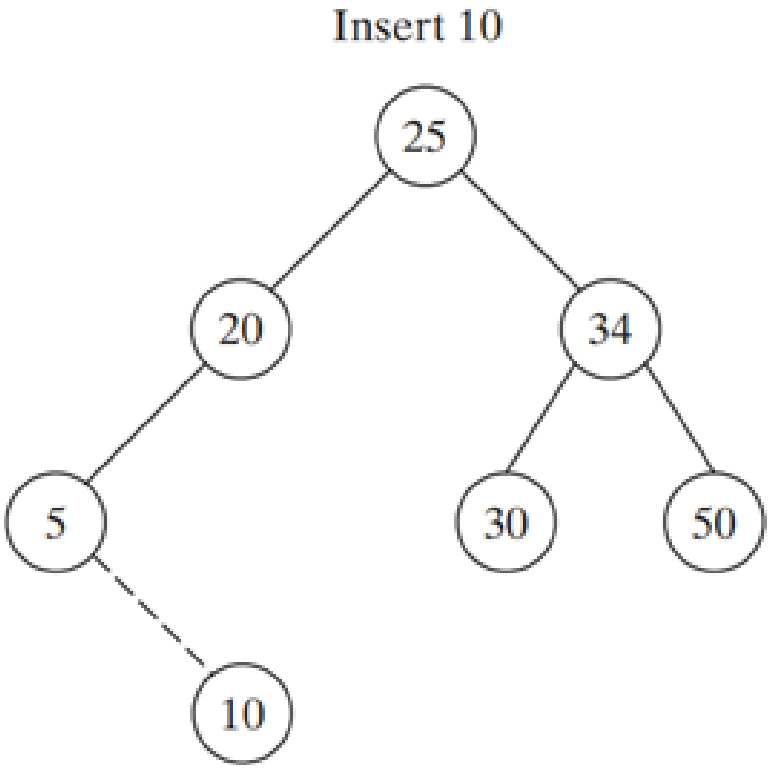


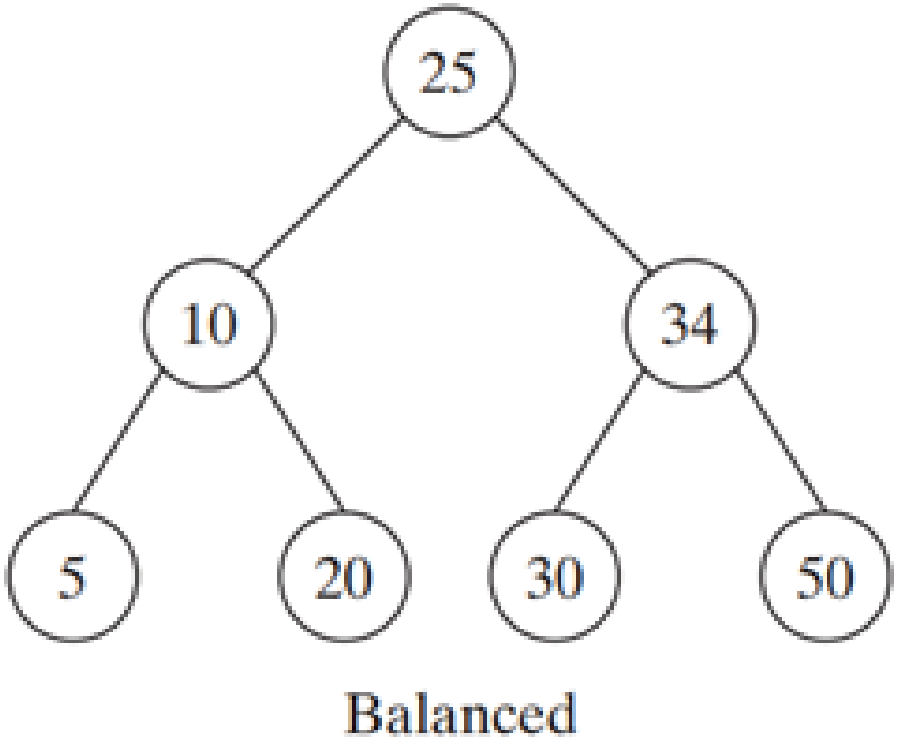
RL rotation at node 20



Balanced

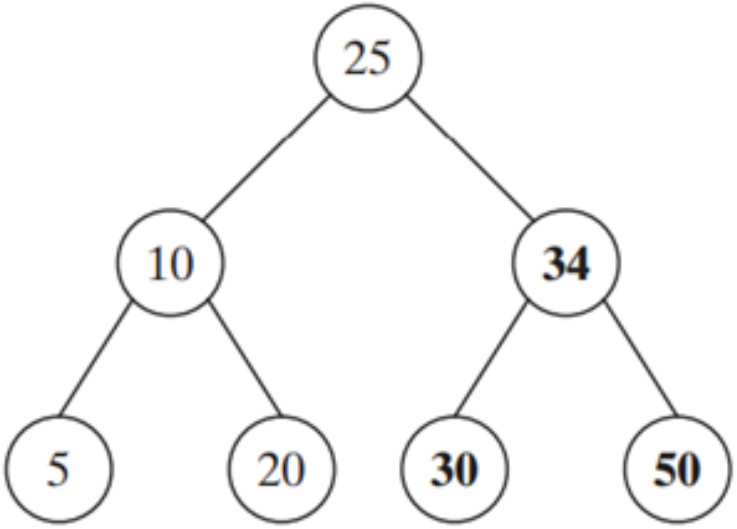
(c) Paul Fodor (CS Stony Brook) & Pearson

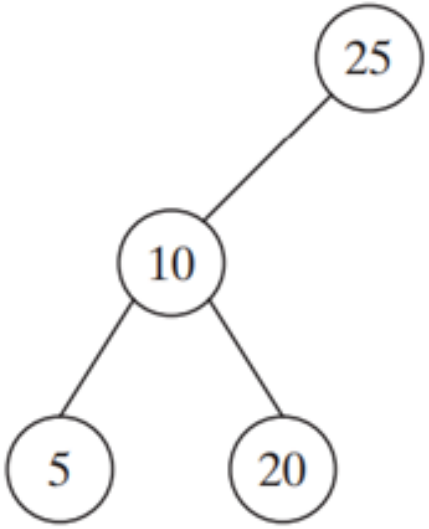




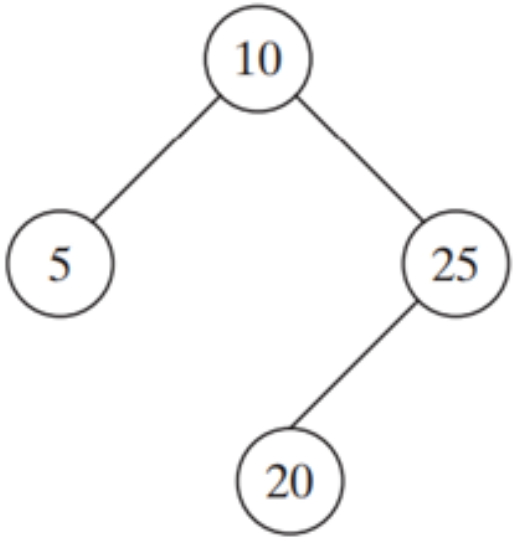
(c) Paul Fodor (CS Stony Brook) & Pearson

Delete 34, 30, 50





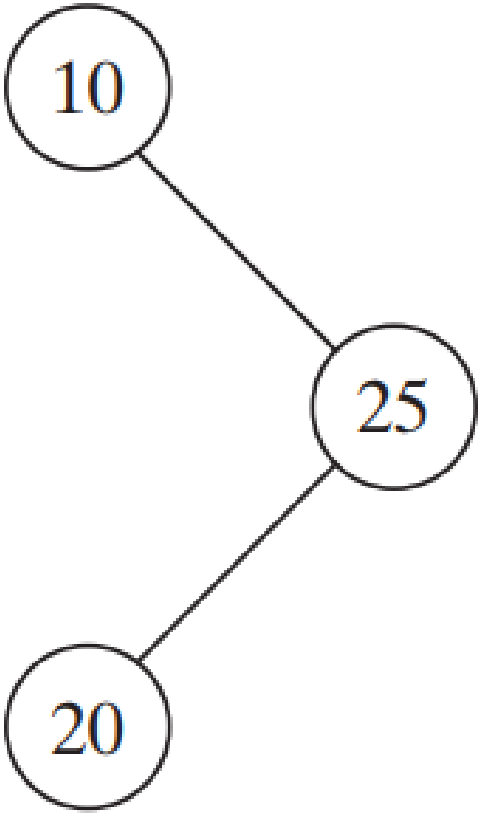
LL rotation
at node 25



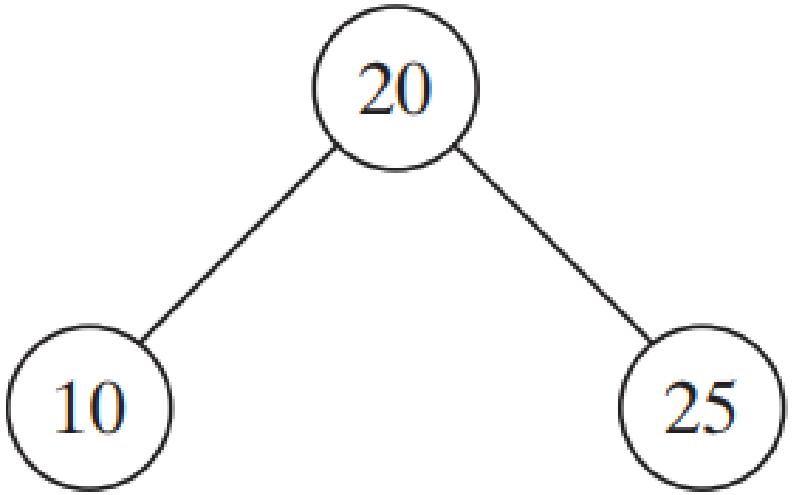
Balanced

(c) Paul Fodor (CS Stony Brook) & Pearson

After 5 is deleted



RL rotation at 10

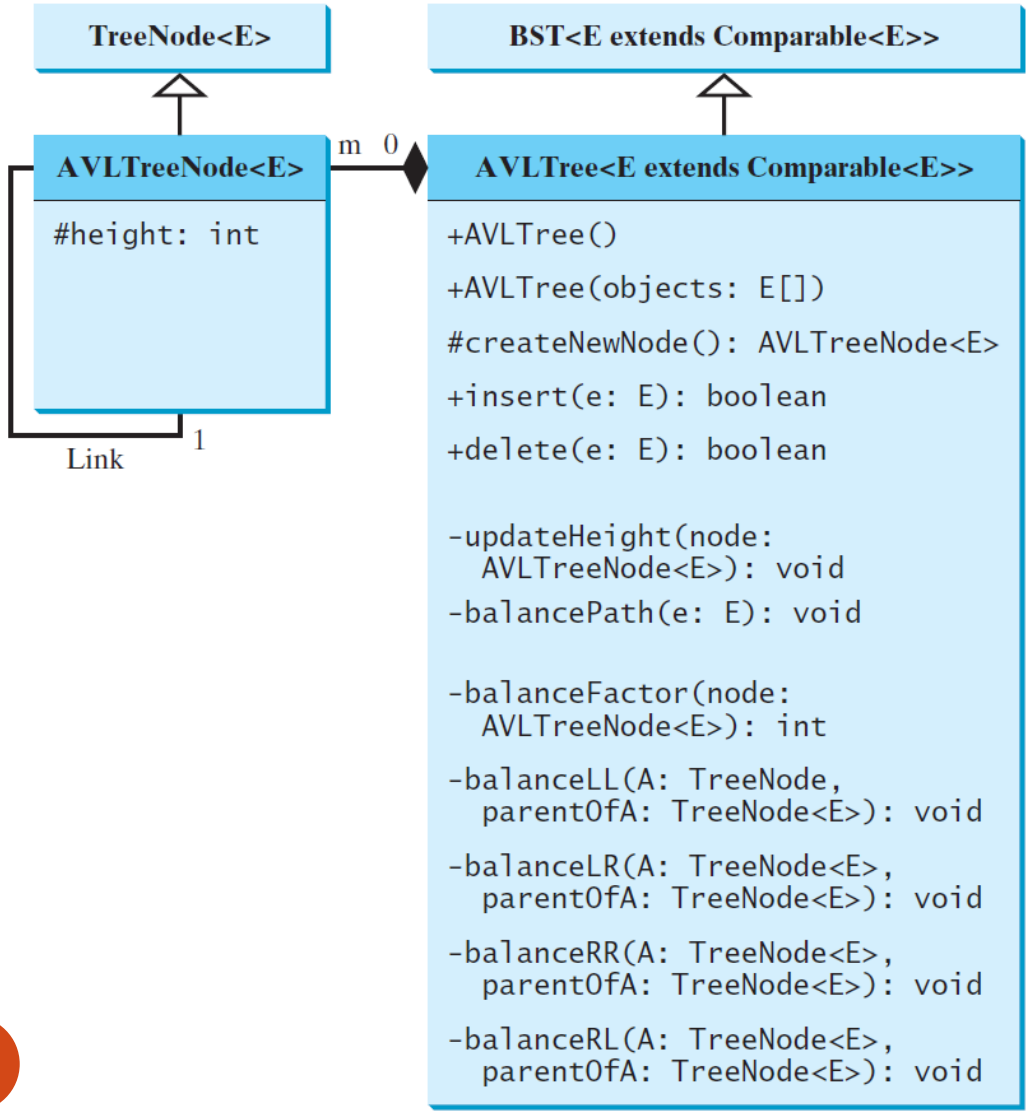


Balanced

(c) Paul Fodor (CS Stony Brook) & Pearson

Designing Classes for AVL Trees

- An AVL is a binary search tree, so we can define the **`AVLTree`** class to extend the **`BST`** class



Creates an empty AVL tree.

Creates an AVL tree from an array of objects.

Overrides this method to create an `AVLTreeNode`.

Returns true if the element is added successfully.

Returns true if the element is removed from the tree successfully.

Resets the height of the specified node.

Balances the nodes in the path from the node for the element to the root if needed.

Returns the balance factor of the node.

Performs LL balance.

Performs LR balance.

Performs RR balance.

Performs RL balance.

```
public class AVLTree<E extends Comparable<E>> extends BST<E> {
  /** Create a default AVL tree */
  public AVLTree() {
  }

  /** Create an AVL tree from an array of objects */
  public AVLTree(E[] objects) {
    super(objects);
  }

  /** AVLTreeNode is TreeNode plus height */
  protected static class AVLTreeNode<E extends Comparable<E>>
      extends BST.TreeNode<E> {
    protected int height = 0; // New data field

    public AVLTreeNode(E o) {
      super(o);
    }
  }

  @Override /** Override createNewNode to create an AVLTreeNode */
  protected AVLTreeNode<E> createNewNode(E e) {
    return new AVLTreeNode<E>(e);
  }
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
@Override /** Insert an element and rebalance if necessary */
public boolean insert(E e) {
  boolean successful = super.insert(e);
  if (!successful)
    return false; // e is already in the tree
  else {
    balancePath(e); // Balance from e to the root if necessary
  }
  return true; // e is inserted
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Balance the nodes in the path from the specified
 * node to the root if necessary
 */
private void balancePath(E e) {
  java.util.ArrayList<TreeNode<E>> path = path(e);
  for (int i = path.size() - 1; i >= 0; i--) {
    AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));
    updateHeight(A);
    AVLTreeNode<E> parentOfA = (A == root) ? null :
      (AVLTreeNode<E>)(path.get(i - 1));
    switch (balanceFactor(A)) {
      case -2:
        if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) {
          balanceLL(A, parentOfA); // Perform LL rotation
        } else {
          balanceLR(A, parentOfA); // Perform LR rotation
        }
        break;
      case +2:
        if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) {
          balanceRR(A, parentOfA); // Perform RR rotation
        } else {
          balanceRL(A, parentOfA); // Perform RL rotation
        }
    }
  }
}
```

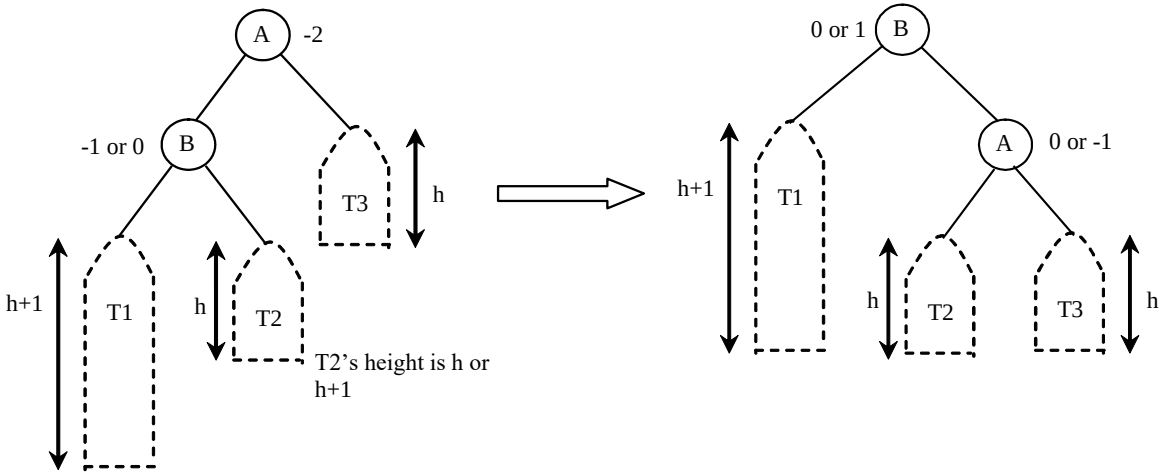

(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Update the height of a specified node */
private void updateHeight(AVLTreeNode<E> node) {
  if (node.left == null && node.right == null) // node is a leaf
    node.height = 0;
  else if (node.left == null) // node has no left subtree
    node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;
  else if (node.right == null) // node has no right subtree
    node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;
  else
    node.height = 1 +
      Math.max(((AVLTreeNode<E>)(node.right)).height,
      ((AVLTreeNode<E>)(node.left)).height);
}

/** Return the balance factor of the node */
private int balanceFactor(AVLTreeNode<E> node) {
  if (node.right == null) // node has no right subtree
    return -node.height;
  else if (node.left == null) // node has no left subtree
    return +node.height;
  else
    return ((AVLTreeNode<E>)node.right).height -
      ((AVLTreeNode<E>)node.left).height;
}
```

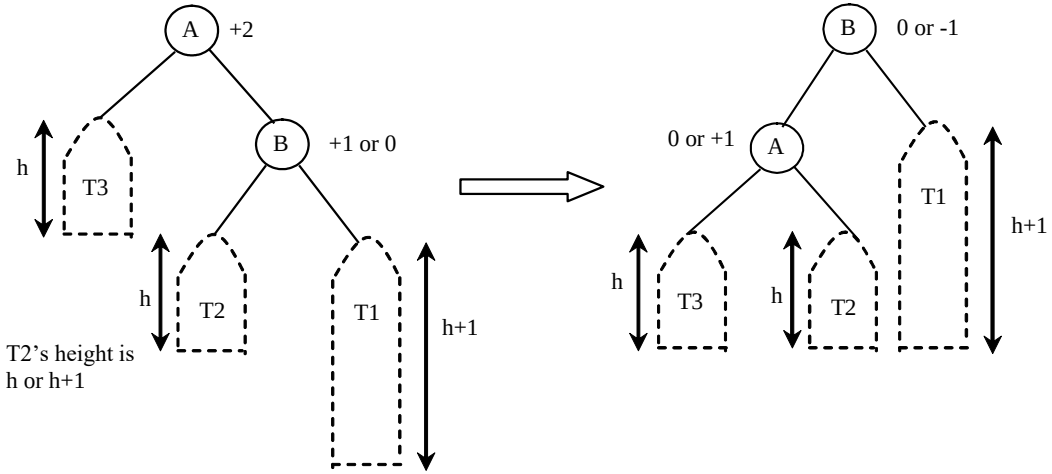

(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Balance LL */
private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.left; // A is left-heavy and B is left-heavy
  if (A == root) {
    root = B;
  } else {
    if (parentOfA.left == A) {
      parentOfA.left = B;
    } else {
      parentOfA.right = B;
    }
  }
  A.left = B.right; // Make T2 the left subtree of A
  B.right = A; // Make A the left child of B
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
}
```



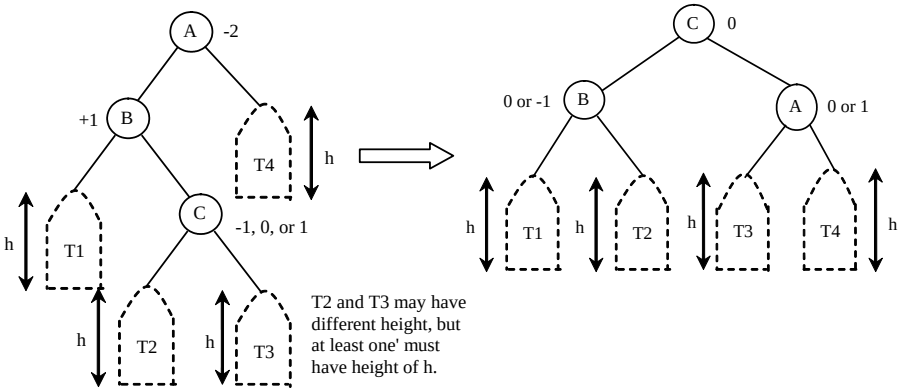
(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Balance RR */
private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy
  if (A == root) {
    root = B;
  } else {
    if (parentOfA.left == A) {
      parentOfA.left = B;
    } else {
      parentOfA.right = B;
    }
  }
  A.right = B.left; // Make T2 the right subtree of A
  B.left = A;
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
}
```



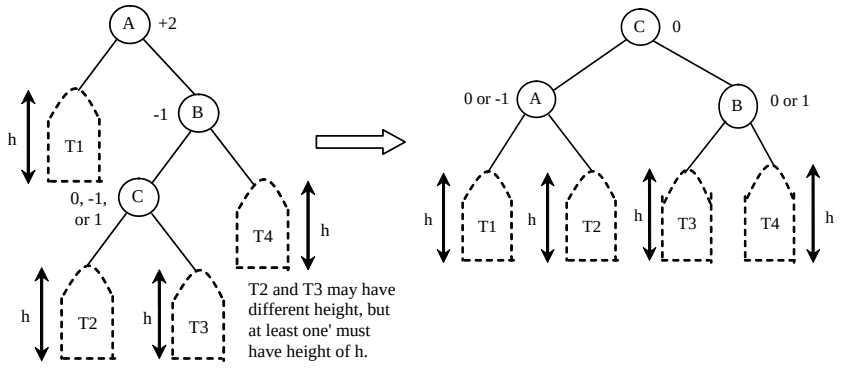
(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Balance LR */
private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.left; // we know that A is left-heavy
  TreeNode<E> C = B.right; // we know that B is right-heavy
  if (A == root) {
    root = C;
  } else {
    if (parentOfA.left == A) {
      parentOfA.left = C;
    } else {
      parentOfA.right = C;
    }
  }
  A.left = C.right; // Make T3 the left subtree of A
  B.right = C.left; // Make T2 the right subtree of B
  C.left = B;
  C.right = A;
  // Adjust heights
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
  updateHeight((AVLTreeNode<E>)C);
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Balance RL */
private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.right; // we know that A is right-heavy
  TreeNode<E> C = B.left; // we know that B is left-heavy
  if (A == root) {
    root = C;
  } else {
    if (parentOfA.left == A) {
      parentOfA.left = C;
    } else {
      parentOfA.right = C;
    }
  }
  A.right = C.left; // Make T2 the right subtree of A
  B.left = C.right; // Make T3 the left subtree of B
  C.left = A;
  C.right = B;
  // Adjust heights
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
  updateHeight((AVLTreeNode<E>)C);
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
@Override /** Delete an element from the binary tree.
 * Return true if the element is deleted successfully
 * Return false if the element is not in the tree */
public boolean delete(E element) {
  if (root == null)
    return false; // Element is not in the tree
  // Locate the node to be deleted and also locate its parent node
  TreeNode<E> parent = null;
  TreeNode<E> current = root;
  while (current != null) {
    if (element.compareTo(current.element) < 0) {
      parent = current;
      current = current.left;
    } else if (element.compareTo(current.element) > 0) {
      parent = current;
      current = current.right;
    } else
      break; // Element is in the tree pointed by current
  }
  if (current == null)
    return false; // Element is not in the tree
  // Case 1: current has no left children
  if (current.left == null) {
    // Connect the parent with the right child of the current node
    if (parent == null) {
      root = current.right;
    }
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
    else {
      if (element.compareTo(parent.element) < 0)
        parent.left = current.right;
      else
        parent.right = current.right;
      // Balance the tree if necessary
      balancePath(parent.element);
    }
  } else {
    // Case 2: The current node has a left child
    // Locate the rightmost node in the left subtree of
    // the current node and also its parent
    TreeNode<E> parentOfRightMost = current;
    TreeNode<E> rightMost = current.left;
    while (rightMost.right != null) {
      parentOfRightMost = rightMost;
      rightMost = rightMost.right; // Keep going to the right
    }
    // Replace the element in current by the element in rightMost
    current.element = rightMost.element;
    // Eliminate rightmost node
    if (parentOfRightMost.right == rightMost)
      parentOfRightMost.right = rightMost.left;
    else
      // Special case: parentOfRightMost is current
      parentOfRightMost.left = rightMost.left;
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
      // Balance the tree if necessary
      balancePath(parentOfRightMost.element);
    }
    size--;
    return true; // Element deleted
  }
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
public class TestAVLTree {
  public static void main(String[] args) {
    // Create an AVL tree
    AVLTree<Integer> tree = new AVLTree<>(new Integer[]{25, 20, 5});
    System.out.print("After inserting 25, 20, 5:");
    printTree(tree);

    tree.insert(34);
    tree.insert(50);
    System.out.print("\nAfter inserting 34, 50:");
    printTree(tree);

    tree.insert(30);
    System.out.print("\nAfter inserting 30");
    printTree(tree);

    tree.insert(10);
    System.out.print("\nAfter inserting 10");
    printTree(tree);

    tree.delete(34);
    tree.delete(30);
    tree.delete(50);
    System.out.print("\nAfter removing 34, 30, 50:");
    printTree(tree);
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
    tree.delete(5);
    System.out.print("\nAfter removing 5:");
    printTree(tree);

    System.out.print("\nTraverse the elements in the tree: ");
    for (int e: tree) {
      System.out.print(e + " ");
    }
  }

  public static void printTree(BST tree) {
    // Traverse tree
    System.out.print("\nInorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
After inserting 25, 20, 5:
Inorder (sorted): 5 20 25
Postorder: 5 25 20
Preorder: 20 5 25
The number of nodes is 3

After inserting 34, 50:
Inorder (sorted): 5 20 25 34 50
Postorder: 5 25 50 34 20
Preorder: 20 5 34 25 50
The number of nodes is 5

After inserting 30
Inorder (sorted): 5 20 25 30 34 50
Postorder: 5 20 30 50 34 25
Preorder: 25 20 5 34 30 50
The number of nodes is 6

After inserting 10
Inorder (sorted): 5 10 20 25 30 34 50
Postorder: 5 20 10 30 50 34 25
Preorder: 25 10 5 20 34 30 50
The number of nodes is 7
```

```
After removing 34, 30, 50:
Inorder (sorted): 5 10 20 25
Postorder: 5 20 25 10
Preorder: 10 5 25 20
The number of nodes is 4

After removing 5:
Inorder (sorted): 10 20 25
Postorder: 10 25 20
Preorder: 20 10 25
The number of nodes is 3

Traverse the elements in the tree:
10 20 25
```


(c) Paul Fodor (CS Stony Brook) & Pearson

AVL Tree Time Complexity Analysis

- Let **G(h)** denote the minimum number of the nodes in an AVL tree with height **h**; **G(1) = 1, G(2) = 2**
  - The minimum number of nodes in an AVL tree with height **h>2** must have two minimum subtrees: one with height **h-1** and the other with height **h-2**
  - Thus, **G(h) = G(h - 1) + G(h - 2) + 1**
    - A Fibonacci number at index **i** can be described using the recurrence relation:

    F(i) = F(i - 1) + F(i - 2)
    - Therefore, the function **G(h)** is essentially the same as **F(i)**
    - It can be proven that **h < 1.4405 log(n + 2) - 1.3277** where **n** is the number of nodes in the tree
    - Hence, the height of an AVL tree is **O(log n)**

(c) Paul Fodor (CS Stony Brook) & Pearson

AVL Tree Time Complexity Analysis

- The search, insert, and delete methods involve only the nodes along a path in the tree
- The **`updateHeight`** and **`balanceFactor`** methods are executed in a constant time for each node in the path
- The **`balancePath`** method is executed in a constant time for a node in the path
- Thus, the time complexity for the search, insert, and delete methods is **O(log n)**

(c) Paul Fodor (CS Stony Brook) & Pearson