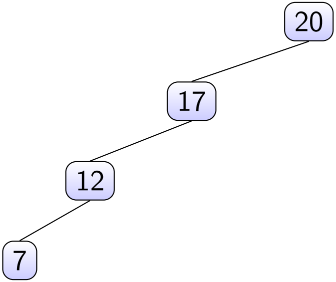
Data Structures

Self-Balancing Search Trees

CS284

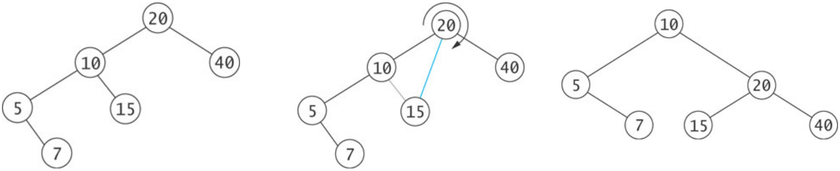
Balance is Important in BSTs

- Search in unbalanced BSTs tree are $\mathcal{O}(n)$, not $\mathcal{O}(\log n)$



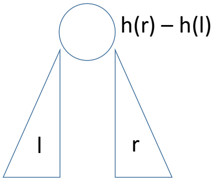
Rotation

- Operation on a binary tree that changes the relative heights of left and right subtrees, but preserves BST invariant



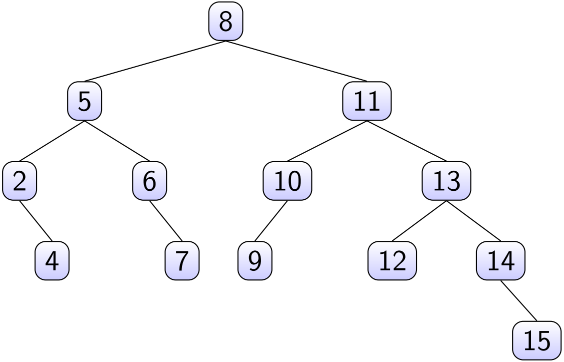
AVL Trees

- In 1962 G.M. Adelson-Velskî and E.M. Landis developed a self-balancing tree
- Definition: The balance of a node is the difference in height of its two subtrees



- AVL tree: A binary search tree in which the balance of every node is in the interval [-1,+1]

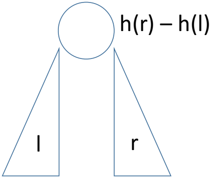
Example



- AVL tree: The balance of every node is in the interval [-1,+1]

AVL Trees

- As items are added to or deleted from a tree, the balance of each subtree from the insertion or delete point up to the root is updated

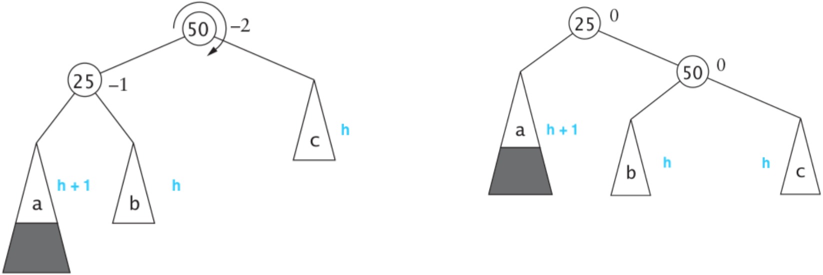


- If the balance gets out of the range -1 to +1, the tree is rotated to bring it back into balance

Before	After	Rebalance?
0	+1	No
0	-1	No
-1	0	No
+1	0	No
-1	-2	Yes
+1	+2	Yes

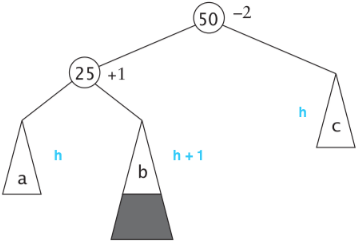
Balancing a Left-Left Tree

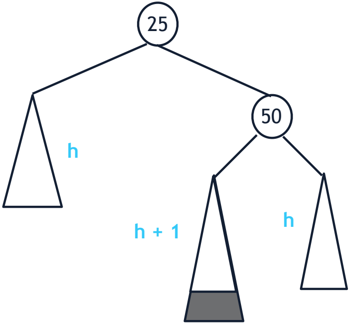
After a right rotation, the tree is balanced



Balancing a Left-Right Tree

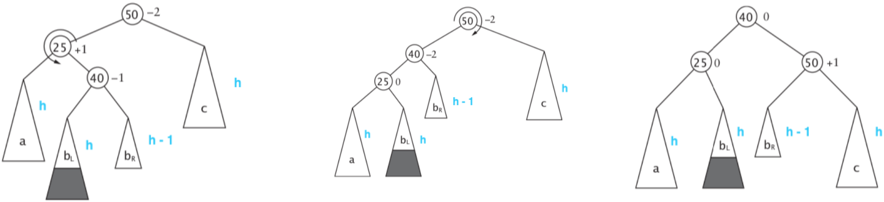
Right rotation does not work!





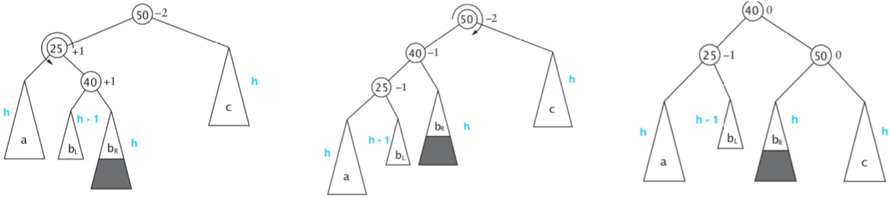
Balancing a Left-Right Tree

Left-right tree case 1: left-right-left tree, rotate left then rotate right



Balancing a Left-Right Tree

Left-right tree case 2: left-right-right tree, rotate left then rotate right



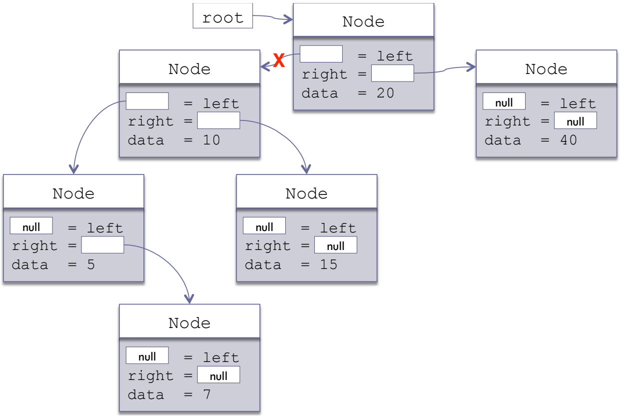
Summary of Four Kinds of Critically Unbalanced Trees

1. Right-Right (parent balance $+2$, right child balance $+1$)
   - Rotate left around parent
2. Right-Left (parent balance $+2$, right child balance -1)
   - Rotate right around child
   - Rotate left around parent
3. Left-Left (parent balance is -2, left child balance is -1)
   - Rotate right around parent
4. Left-Right (parent balance -2, left child balance $+1$)
   - Rotate left around child
   - Rotate right around parent

Note: 3 is symmetric to 1 and 4 is symmetric to 2

We can prove that rotation does not change the overall height of the tree, thus during an AVL tree insertion, we do not need to reimbalance the ancestor nodes;

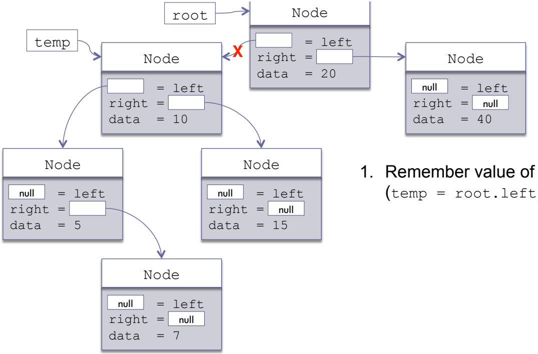
Rotation



root
Node
= left
right =
data = 20
X
Node
= left
right =
data = 10
Node
null = left
right = null
data = 40
Node
null = left
right =
data = 5
Node
null = left
right = null
data = 15
Node
null = left
right = null
data = 7

Rotation

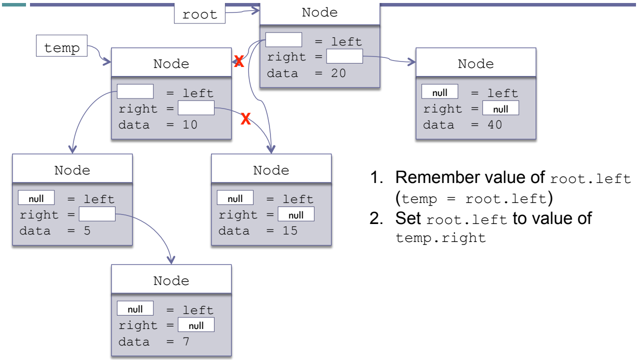
```
temp = root.left;
```



1. Remember value of `root.left` (`temp = root.left`)

Rotation

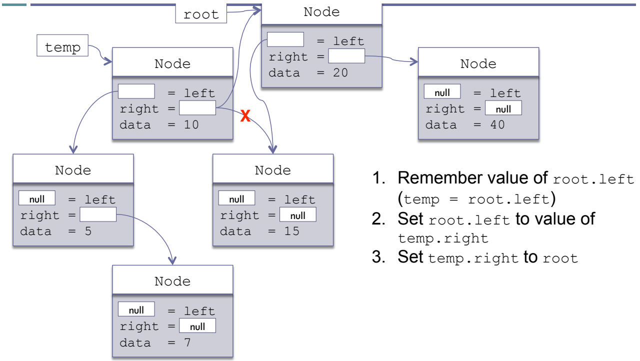
```
root.left = temp.right;
```



1. Remember value of `root.left` (`temp = root.left`)
2. Set `root.left` to value of `temp.right`

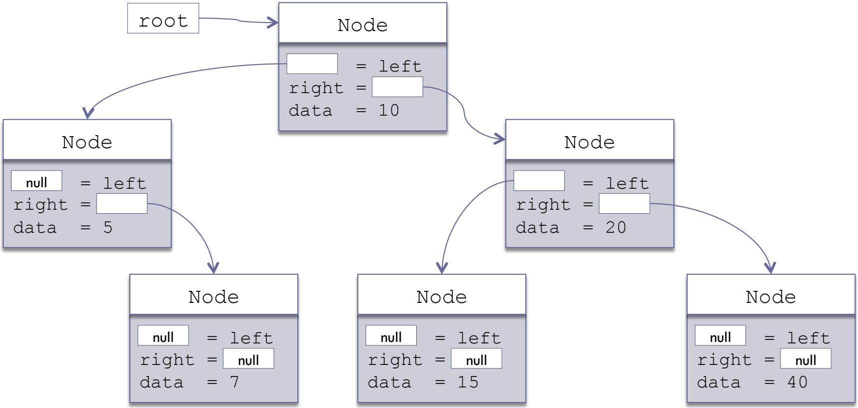
Rotation

```
temp.right = root;
```



1. Remember value of `root.left` (`temp = root.left`)
2. Set `root.left` to value of `temp.right`
3. Set `temp.right` to `root`

Rotation



root
Node
= left
right =
data = 10
Node
null = left
right =
data = 5
Node
= left
right =
data = 20
Node
null = left
right = null
data = 7
Node
null = left
right = null
data = 15
Node
null = left
right = null
data = 40

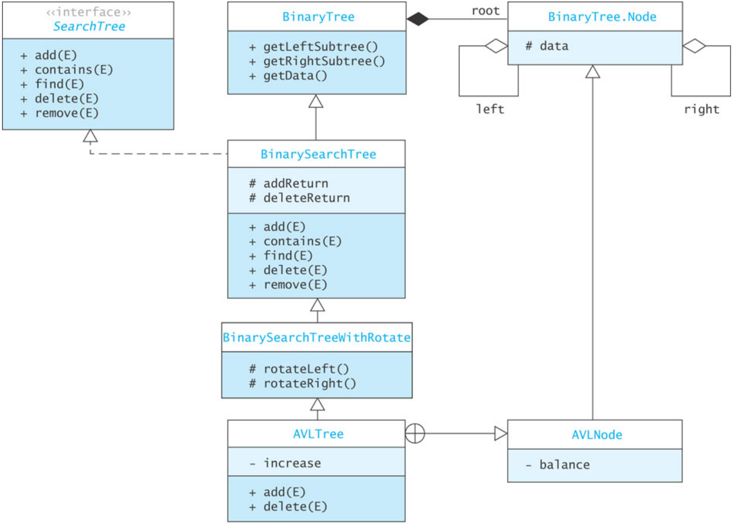
Implementing Rotation

```java
public class BinarySearchTreeWithRotate<E extends Comparable<E>>
    extends BinarySearchTree<E> {
  // Methods

  protected Node<E> rotateRight(Node<E> root) {
    Node<E> temp = root.left;
    root.left = temp.right;
    temp.right = root;
    return temp;
  }
```

- `rotateLeft` is similar

AVL Implementation



The Class AVLTree<E>

```
public class AVLTree<E extends Comparable<E>>
     extends BinarySearchTreeWithRotate<E> {

       private static class AVLNode<E> extends Node<E> {
         // defined in the upcoming slides
       }

    /** Flag to indicate that height of tree has increased. */
    private boolean increase;

    /** Flag to indicate that height of tree has decreased */
    private boolean decrease;

    // Methods
    ...
}
```

The Inner Class `AVLNode<E>`

```
class AVLNode
{
    AVLNode left, right;
    int data;
    int height;

    /* Constructor */
    public AVLNode()
    {
        left = null;
        right = null;
        data = 0;
        height = 0;
    }
    /* Constructor */
    public AVLNode(int n)
    {
        left = null;
        right = null;
        data = n;
        height = 0;
    }
}
```

Inserting into an AVL Tree

```
/* Function to insert data recursively */
public AVLNode insertNode(int target, AVLNode root)
{
    if (root == null)
        root = new AVLNode(target);
 // if the item is less than root.data
    else if (target < root.value)
    {
       // recursively insert the item in the left subtree
        root.l_child = insertNode( target, root.l_child );
        int balance = getBalance(root);
 // if after the insertion, the tree becomes unbalanced
        if( balance == -2 )
 // if x inserted into the left-left subtree, balancing a
 // left-left tree
            if( target < root.l_child.value )
                root = rotateRight( root );
            else
 // if x inserted into the left-right subtree, rotate left
 // then rotate right
                root = doubleRotation_left_right( root );
    }
```

```
 // if the item is larger than root.data
    else if( target > root.value )
    {
       // recursively insert the item in the right subtree
        root.r_child = insertNode( target, root.r_child );
        int balance = getBalance(root);
     // if after the insertion, the tree becomes unbalanced
        if( balance == 2 )

            // if x inserted into the right-right subtree,
            // balancing a right-right tree
             if( target > root.r_child.value)
                 root = rotateLeft( root );
        // if x inserted into the right-left subtree, rotate
        // right then rotate left
            else
                root = doubleRotation_right_left( root );
    }
    else
      ;  // Duplicate; do nothing
    root.height = max( height( root.l_child ),
    height( root.r_child ) ) + 1;
    // update the height of the current node t
    return root;
}
```

Left-rotate `rotateRight`

```java
/* Rotate binary tree node with left child */
private AVLNode rotateRight(AVLNode root)
{
    AVLNode tmp = root.left;
    root.left = tmp.right;
    tmp.right = root;

    root.height = max( height( root.left ),
    height( root.right ) ) + 1;
    tmp.height = max( height( tmp.left ), root.height ) + 1;
    return tmp;
}
```

search in an AVL tree

```java
public boolean search(AVLNode root, int target)
{
    boolean found = false;
    while ((root != null) && !found)
    {
        int rval = root.value;
        if (target < rval)
            root = root.l_child;
        else if (target > rval)
            root = root.r_child;
        else
        {
            found = true;
            break;
        }
        found = search(root, target);
    }
    return found;
}
```

Deletion from an AVL Tree

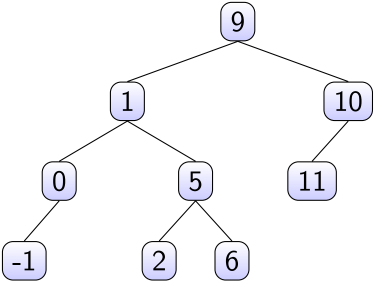
- Builds on deletion from a BST
- Similar to insertion
  - Do deletion
  - Then rebalance
- Rebalance
  - Involves both single and double rotations, as in insertion
  - In contrast to insertion, imbalance may propagate upwards: rotations at multiple nodes in the path to the root may be required

Deletion from an AVL Tree

Two part operation

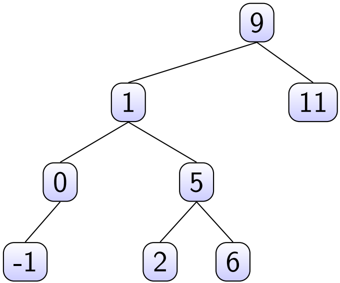
1. Apply BST deletion:
   - 0 children: just delete it
   - 1 child: Delete it, connect child to parent
   - 2 children: put successor in your place, delete successor leaf
2. Rebalance
   - Right-right: rotate left
   - Right-left: rotate right then left
   - Left-left: rotate right
   - Left-right: rotate left then right

Example – Deleting an item



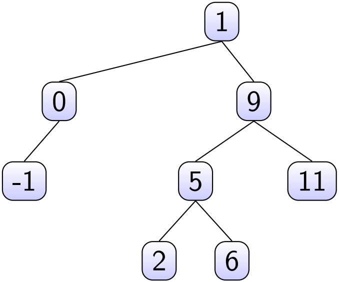
Deleting 10

Example – Deleting an item



Deleting 10: Left-left case: rotate right

Example – Deleting an item



Deleting 10: Left-left case: rotate right

Deleting an item

```
 public AVLNode deleteNode(AVLNode root, int target)
{
    if (root == null)
        return root;
    if (target < root.value)
        root.l_child = deleteNode(root.l_child, target);
    // If the key to be deleted is greater than the
    // root's key, then it lies in right subtree
    else if (target > root.value)
        root.r_child = deleteNode(root.r_child, target);
    // if key is same as root's key, then this is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if ((root.l_child == null) || (root.r_child == null))
        {
            if (root.l_child == null)
                root = root.r_child;
            else
                root = root.l_child;
        }
```

Delete

```
    else
    {
        // node with two children: Get the inorder
        // successor (smallest in the right subtree)
        AVLNode right = minValueNode(root.r_child);
        // Copy the inorder successor's data to this node
        root.value = right.value;
        // Delete the inorder successor
        root.r_child = deleteNode(root.r_child, right.value);
    }
}
if (root == null)
    return root;

root.height = max(height(root.l_child), height(root.r_child)) + 1;

int balance = getBalance(root);
```

Delete

```
 // if after the deletion, the tree becomes unbalanced
    if (balance == 2) {
        int rchild_balance = getBalance(root.r_child);
        // right-right tree
        if (rchild_balance >= 0)
            return rotateLeft(root);
        // right-left tree
        else
            return doubleRotation_right_left(root);
    }
 // if after the deletion, the tree becomes unbalanced
    if (balance == -2) {
        int lchild_balance = getBalance(root.l_child);
        // left-left tree
        if (lchild_balance <= 0)
            return rotateRight(root);
        // left-right tree
        else
            return doubleRotation_left_right(root);
    }
    return root;
}
```

Tree Balance and Rotation

AVL Trees

Implementing Rotations

Implementing AVL Trees

Search in an AVL Tree

Deletion from an AVL Tree

Performance

Performance of the AVL Tree

Good performance

- All operations $\mathcal{O}(\log n)$ because trees are always balanced.
- The height balancing adds no more than a constant factor to the speed of insert and deletion.

Cons

- Not easy to program
- Each node has an extra field
- When data to be searched is stored in disks (i.e. not in memory) other data structures are more appropriate (eg. B-Trees)