CS 146: Data Structures and Algorithms
Binary Search Trees

- A binary search tree has these properties for each of its nodes:
  - All the values in the node's left subtree is less than the value of the node itself.
  - All the values in the node's right subtree is greater than the value of the node itself.
AVL Trees

- An AVL tree is a binary search tree (BST) with a balance condition.
  - Named after its inventors, Adelson-Velskii and Landis.

- For each node of the BST, the heights of its left and right subtrees can differ by at most 1.
  - Remember that the height of a tree is the length of the longest path from the root to a leaf.
  - The height of the root = the height of the tree.
  - The height of an empty tree is -1.
AVL Trees, cont’d

Figure 4.30  Smallest AVL tree of height 9
Balancing AVL Trees

- We need to rebalance the tree whenever the balance condition is violated.
  - We need to check after every insertion and deletion.

Figure 4.29 Two binary search trees. Only the left tree is AVL.
Balancing AVL Trees, *cont’d*

- Assume the tree was balanced before an insertion.
- If it became unbalanced due to the insertion, then the inserted node must have caused some nodes between itself and the root to be unbalanced.
- An unbalanced node must have the height of one of its subtrees exactly 2 greater than the height its other subtree.
Balancing AVL Trees, *cont’d*

- Let the deepest unbalanced node be \( \alpha \).
- Any node has at most two children.
- A new height imbalance means that the heights of \( \alpha \)'s two subtrees now differ by 2.
Therefore, one of the following had to occur:

- **Case 1 (outside left-left):** The insertion was into the left subtree of the left child of $\alpha$.
- **Case 2 (inside left-right):** The insertion was into the right subtree of the left child of $\alpha$.
- **Case 3 (inside right-left):** The insertion was into the left subtree of the right child of $\alpha$.
- **Case 4 (outside right-right):** The insertion was into the right subtree of the right child of $\alpha$.

Cases 1 and 4 are mirrors of each other, and cases 2 and 3 are mirrors of each other.
Balancing AVL Trees: Case 1

- Case 1 (outside left-left):
  Rebalance with a single right rotation.

Figure 4.31  Single rotation to fix case 1
Balancing AVL Trees: Case 1, cont’d

Case 1 (outside left-left):
Rebalance with a single right rotation.

Node A is unbalanced. **Single right rotation**: A's left child B becomes the new root of the subtree. Node A becomes the right child and adopts B's right child as its new left child.

Node 8 is unbalanced. **Single right rotation**: 8's left child 7 becomes the new root of the subtree. Node 8 is the right child.
Balancing AVL Trees: Case 4

- Case 4 (outside right-right): Rebalance with a single left rotation.

Node A is unbalanced. **Single left rotation:** A's right child C becomes the new root of the subtree. Node A becomes the left child and adopts C's left child as its new right child.
Balancing AVL Trees: Case 2

- Case 2 (inside left-right):
  Rebalance with a double left-right rotation.

![Diagram of AVL tree balancing](image)

**Figure 4.35** Left–right double rotation to fix case 2
Balancing AVL Trees: Case 2, cont’d

- **Case 2 (inside left-right):** Rebalance with a **double left-right rotation**.

Node A is unbalanced. **Double left-right rotation:** E becomes the new root of the subtree after two rotations. Step 1 is a **single left rotation** between B and E. E replaces B as the subtree root. B becomes E's left child and B adopts E's left child F as its new right child. Step 2 is a **single right rotation** between E and A. E replaces A is the subtree root. A becomes E's right child and A adopts E's right child G as its new left child.

http://www.cs.uah.edu/~rcoleman/CS221/Trees/AVLTree.htm
Balancing AVL Trees: Case 3

- Case 3 (inside right-left):
  Rebalance with a **double right-left rotation**.

![Diagram showing the process of rebalancing with a double right-left rotation.](image)
Balancing AVL Trees: Case 3, cont’d

- Case 3 (inside right-left):
  Rebalance with a double right-left rotation.

Node A is unbalanced.
**Double right-left rotation:** D becomes the new root of the subtree after two rotations. Step 1 is a single right rotation between C and C. D replaces C as the subtree root. C becomes D's right child and C adopts D's right child G as its new left child. Step 2 is a single left rotation between D and A. D replaces A is the subtree root. A becomes D's left child and A adopts D's left child F as its new right child.
AVL Tree Implementation

- Since an AVL tree is just a BST with a balance condition, it makes sense to make the AVL tree class a subclass of the BST class.

```java
public class AvlTree extends BinarySearchTree
```

- Both classes can share the same `BinaryNode` class.
The AVL Tree Node

- With so many height calculations, it makes sense to store each node's height in the node itself.

```java
public class BinaryNode {
    private int data;         // data in this node
    private int height;       // height of this node
    private BinaryNode left;  // left child
    private BinaryNode right; // right child

    // ... }
AVL Tree Implementation, cont’d

- Class `AVLTree` overrides the `insert()` and `remove()` methods of class `BinarySearchTree`.
  - Each method calls the superclass's method and wraps the result in a call to the `balance()` method.

```java
protected BinaryNode insert(int data, BinaryNode node) {
    return balance(super.insert(data, node));
}

protected BinaryNode remove(int data, BinaryNode node) {
    return balance(super.remove(data, node));
}
```
The private `AVLTree` method `balance()` checks whether the balance condition still holds, and rebalances the tree with rotations whenever necessary.
private BinaryNode balance(BinaryNode node) {
    ...
    if (height(node.getLeft()) - height(node.getRight()) > 1) {
        if (height(node.getLeft().getLeft())
            >= height(node.getLeft().getRight())) {
            node = singleRightRotation(node);
        } else {
            node = doubleLeftRightRotation(node);
        }
    } else if (height(node.getRight()) - height(node.getLeft()) > 1) {
        if (height(node.getRight().getRight())
            >= height(node.getRight().getLeft())) {
            node = singleLeftRotation(node);
        } else {
            node = doubleRightLeftRotation(node);
        }
    } else {
        ...
    }
    return node;
}
AVL Tree Implementation, cont’d

```java
private BinaryNode singleRightRotation(BinaryNode k2) {
    BinaryNode k1 = k2.getLeft();
    k2.setLeft(k1.getRight());
    k1.setRight(k2);
    k2.setHeight(Math.max(height(k2.getLeft()), height(k2.getRight())) + 1);
    k1.setHeight(Math.max(height(k1.getLeft()), k2.getHeight()) + 1);
    return k1;
}

private BinaryNode singleLeftRotation(BinaryNode k1) {
    BinaryNode k2 = k1.getRight();
    k1.setRight(k2.getLeft());
    k2.setLeft(k1);
    k1.setHeight(Math.max(height(k1.getLeft()), height(k1.getRight())) + 1);
    k2.setHeight(Math.max(height(k2.getRight()), k1.getHeight()) + 1);
    return k2;
}
```

Case 1

Case 4
private BinaryNode doubleLeftRightRotation(BinaryNode k3)
{
    k3.setLeft(singleLeftRotation(k3.getLeft()));
    return singleRightRotation(k3);
}

private BinaryNode doubleRightLeftRotation(BinaryNode k1)
{
    k1.setRight(singleRightRotation(k1.getRight()));
    return singleLeftRotation(k1);
}
Break
Assignment #3

- This assignment will give you practice with binary search trees (BST) and AVL trees.
- You are provided a `TreePrinter` class that has a `print()` method that will print any arbitrary binary tree.

A template for how it prints a tree:
Assignment #3, cont’d

- **TreePrinter** is able to print trees with height up to 5, i.e., 32 node values on the bottom row.
- An example of an actual printed tree:
Assignment #3: First Part

- The first part of the assignment makes sure that you can successfully insert nodes into, and delete nodes from, a binary search tree (BST) and an AVL tree.
Assignment #3: First Part, cont’d

- **First** generate a random BST that has height 5 and contains random values from 10 through 99.
  - You may have to generate dozens of trees until you get one that's exactly height 5.
  - Don't worry that the tree is unbalanced.
  - Print the tree.

- **Now** repeatedly delete the root of the tree.
  - Print the tree after each deletion to verify that you did the deletion correctly.
  - Stop when the tree becomes empty.
Assignment #3: First Part, cont’d

- Second, create an AVL tree node by node.
  - Generate 35 unique random integers 10-99 to insert into the tree.
  - Print the tree after each insertion to verify that you are keeping it balanced.
  - Each time you do a rebalancing, print a message indicating which rotation operation and which node.
    - Example: Double left-right rotation: 76

- As you did with the BST, repeatedly delete the root of your AVL tree.
  - Print the tree after each deletion to verify that you are keeping it balanced.
Assignment #3: First Part, cont’d

- A handy AVL tree balance checker:

```java
private int checkBalance(BinaryNode node) throws Exception {
    if (node == null) return -1;

    if (node != null) {
        int leftHeight = checkBalance(node.getLeft());
        int rightHeight = checkBalance(node.getRight());
        if ((Math.abs(height(node.getLeft()) - height(node.getRight())) > 1)
            || (height(node.getLeft()) != leftHeight)
            || (height(node.getRight()) != rightHeight)) {
            throw new Exception("Unbalanced trees.");
        }
    }

    return height(node);
}
```
Assignment #3: Second Part

- The second part of the assignment compares the performance of a BST vs. an AVL tree.
Assignment #3: Second Part

☐ First, generate \( n \) random integers.
   - \( n \) is some large number, explained on the next slide.

☐ Time and print how long it takes to insert the random integers one at a time into an initially empty BST.
   - Do not print the tree after each insertion.

☐ Time and print how long it takes to insert the same random integers one at a time into an initially empty AVL tree.
   - Do not print the tree after each insertion.
Choose a value of $n$ large enough to give you consistent timings that you can compare.

Try values of $n = 1,000\ 10,000\ 100,000\ 1,000,000$

If $T(n)$ is the time function, how does the growth of $T_{\text{BST}}(n)$ compare with the growth of $T_{\text{AVL}}(n)$?
Assignment #3: Second Part, cont’d

- **Second**, generate $k$ random integers.
  - $k$ is some large value.

- Time how long it takes to search your $n$-node BST for all $k$ random integers.
  - It doesn’t matter whether or not the search succeeds.

- Time how long it takes to search your $n$-node AVL tree for the same $k$ random integers.

- Compare the grow rates of these two time functions.
Assignment #3: Second Part, cont’d

- Third, perform a random mixture of $m$ insertions and searches on your BST and then on your $n$-node AVL tree.
  - $m$ is some large number.
  - Perform the same sequence of insertions and searches on both trees.

- Try different ratios of insertions vs. searches.

- Empirically estimate the ratio where an AVL tree has better performance than a BST for a mixture of insertions and searches.
Assignment #3, cont’d

- You can use any code from the lectures or from the textbook.

- You do not have to use parameterized generic types.
  - You can use raw (nongeneric) types, or `<Integer>`.
Assignment #3, cont’d

- You may choose a partner to work with you on this assignment.
  - Both of you will receive the same score.

- Create a zip file containing:
  - Your Java source files.
  - Any instructions on how to build and run your code.
  - Text files containing your outputs.
  - A 1- or 2-page that briefly describes your conclusions from doing this assignment.
Assignment #3, cont’d

☐ Email the zip file to me
    ▪ Subject line:  
      **CS 146 Assignment #3: Your Name(s)**

☐ If you work with a partner, both of you turn in one assignment.
    ▪ CC your partner's email address so I can “reply all” to you both.

☐ Due Friday, July 3.
Splay Trees

- Not a new type of tree, but a reimplementation of the BST insert, delete, and search methods.
  - The goal is to improve their performance.

- No single operation on a splay tree is guaranteed to have better performance.
  - But a series of $m$ operations will take $O(m \log n)$ time for a tree of $n$ nodes, whenever $m > n$.

- Not highly balanced like an AVL tree.
  - Lowering the cost of an entire series of operations is more important than keeping the tree balanced.
Whenever a splay tree node is accessed, the tree performs splaying operations that moves the accessed node to the root of the tree.

Splaying a node consists of a series of rotations. Similar to AVL tree rotations.

The goal is to move the accessed node to the root.

A side benefit is to make the tree more balanced.
Splay Trees, cont’d

- The theory is that once a node has been accessed, it will soon be accessed again.
- Future accesses are fast if the node is the root.
How is a worst-case BST created?
- When all the nodes are entered in sorted order.
- Suppose the bottom node is accessed in such a tree:

Figure 4.47 Result of splaying at node 1
Splay Trees, *cont’d*

- If a node hasn’t been accessed in a while, then the next time it’s accessed, you pay the *performance penalty* of splaying.

- But accesses of that node in the near future will be very fast.

- And so we *amortize the cost* of splaying over future operations.
Splay Trees, cont’d

Figure 4.48 Result of splaying at node 1 a tree of all left children
Splay Trees, cont’d

Figure 4.49 Result of splaying the previous tree at node 2
Splay Trees, *cont’d*

*Figure 4.50* Result of splaying the previous tree at node 3
Splay Trees, *cont’d*

**Figure 4.51** Result of splaying the previous tree at node 4
Splay Trees, cont’d

Figure 4.52  Result of splaying the previous tree at node 5
Splay Trees, cont’d

Figure 4.53  Result of splaying the previous tree at node 6
Splay Trees, cont’d

**Figure 4.54** Result of splaying the previous tree at node 7
Splay Trees, *cont’d*

*Figure 4.55* Result of splaying the previous tree at node 8
Splay Trees, cont’d

Figure 4.56 Result of splaying the previous tree at node 9