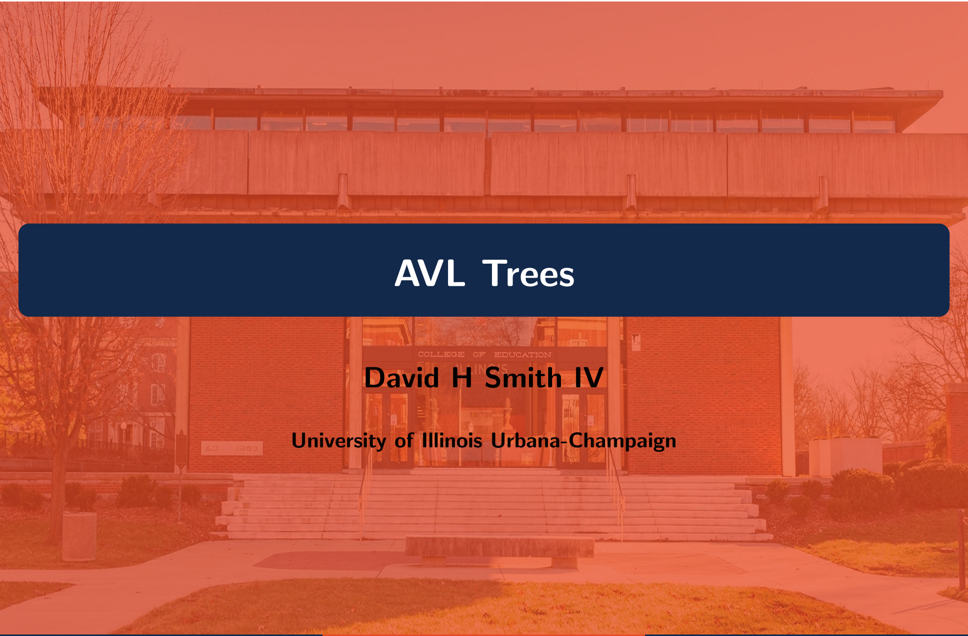


AVL Trees

David H Smith IV

University of Illinois Urbana-Champaign

Objectives

- Go over why we need to balance a BST in order to maintain $O(log_2(n))$ operations.
- Go over the relationship between height and balance factor.
- Cover how balance factor is used to determine what rotation to perform, if any, during recursive unwrapping.
- Cover the four rotations: right, left, right-left, left-right.
- **Assignment:** Transform a standard BST class into an AVL class.

BST Insert: A Nice Insertion Order

Function `Add`(*Curr*)
    if *Curr* = *NULL* **then**
        return *TreeNode(Data)*
    end
    else if *Data* > *Curr.Data* **then**
        Curr.Right ← Add(Curr.Right, Data)
    end
    else
        Curr.Left ← Add(Curr.Left, Data)
    end
    return *Curr*
return

```
BST<Integer> bst = new BST<>();
bst.add(5)
bst.add(3)
bst.add(7)
bst.add(1)
bst.add(4)
bst.add(6)
bst.add(8)
```

BST Insert: A Nice Insertion Order

Function Add(*Curr*)
  if *Curr = NULL* **then**
    return *TreeNode(Data)*
  end
  else if *Data > Curr.Data* **then**
    Curr.Right ← Add(Curr.Right, Data)
  end
  else
    Curr.Left ← Add(Curr.Left, Data)
  end
  return *Curr*
return

```
BST<Integer> bst = new BST<>();
bst.add(5)
bst.add(3)
bst.add(7)
bst.add(1)
bst.add(4)
bst.add(6)
bst.add(8)
```

Time Complexity: $O(log_2(n))$

BST Insert: A Not-So-Nice Insertion Order

```
Function Add(Curr)
    if Curr = NULL then
        return TreeNode(Data)
    end
    else if Data > Curr.Data then
        Curr.Right ← Add(Curr.Right, Data)
    end
    else
        Curr.Left ← Add(Curr.Left, Data)
    end
    return Curr
return
```

```java
BST<Integer> bst = new BST<>();
bst.add(8)
bst.add(7)
bst.add(6)
bst.add(5)
bst.add(4)
bst.add(3)
bst.add(1)
```

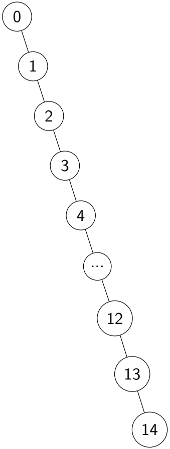
BST Insert: A Not-So-Nice Insertion Order

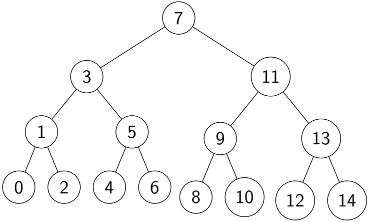
Function `Add`(*Curr*)
    if *Curr* = *NULL* **then**
        return *TreeNode(Data)*
    end
    else if *Data* > *Curr.Data* **then**
        Curr.Right ← Add(Curr.Right, Data)
    end
    else
        Curr.Left ← Add(Curr.Left, Data)
    end
    return *Curr*
return

```
BST<Integer> bst = new BST<>();
bst.add(8)
bst.add(7)
bst.add(6)
bst.add(5)
bst.add(4)
bst.add(3)
bst.add(1)
```

Time Complexity: $O(n)$, Back at linked lists!

Goal: Avoid $O(n)$ Search Trees





We want to ensure that we get the tree on the right *regardless of the order in which we insert the nodes.*

Comparing TreeNodes: AVL vs Standard BST

Standard BST TreeNode

```
static class TreeNode<E>{

    E data;
    TreeNode<E> left;
    TreeNode<E> right;

    TreeNode(data){
        this.data = data;
        left = null;
        right = null;
    }
}
```

AVL TreeNode

```
static class TreeNode<E>{

    E data;
    int height;
    TreeNode<E> left;
    TreeNode<E> right;

    TreeNode(data){
        this.data = data;
        height = 0;
        left = null;
        right = null;
    }
}
```

TreeNode for AVL adds a height attribute!

How we got here!

```
Function Preorder(curr)
    if curr is null then
        return
    end
    print(curr)
    Preorder(curr.left)
    Preorder(curr.right)
return
```

Figure 1: Binary Tree Traversal

```
Function RecursiveSearch(Curr, Val)
    if Curr = NULL OR Val = Curr.Val then
        return curr
    end
    if Val < Curr.Val then
        return RecursiveSearch(Curr.Left, Val)
    end
    else
        return RecursiveSearch(Curr.Right, Val)
    end
return
```

Figure 2: BST Search

```
Function Add(Curr)
    if Curr = NULL then
        return TreeNode(Data)
    end
    else if Data > Curr.Data then
        Curr.Right ← Add(Curr.Right, Data)
    end
    else
        Curr.Left ← Add(Curr.Left, Data)
    end
    return Curr
return
```

Figure 3: BST Add

Algorithm AVL: Insert

```
procedure Insert(CurrNode, NewNode)
    if CurrNode is Null then
        Return NewNode
    else if NewNode < CurrNode then
        CurrNode.Left = Insert(CurrNode.Left, NewNode)
    else
        CurrNode.Right = Insert(CurrNode.Right, NewNode)
    UpdateHeight(CurrNode.Height)
    Return Balance(CurrNode);
```

Figure 4: AVL Tree Add

AVL Insertion

Algorithm AVL: Insert

```
procedure INSERT(CurrNode, NewNode)
    if CurrNode is Null then
        Return NewNode
    else if NewNode < CurrNode then
        CurrNode.Left = Insert(CurrNode.Left, NewNode)
    else
        CurrNode.Right = Insert(CurrNode.Right, NewNode)
    UpdateHeight(CurrNode.Height)
    Return Balance(CurrNode);
```

1. Block (1) is *identical* to BST insert
2. The main difference between BST and AVL is in the recursive unwrap.
    1. Before returning (2) we need to update the height of each node that we traversed over.
    2. We also need to ensure that everything is balanced (3) with respect to that node.

Lets Take a Look at Height First

```
bst.add(5)
bst.add(3)
bst.add(7)
bst.add(1)
bst.add(4)
bst.add(6)
bst.add(8)
```

Update Height Equation:

$$Node.Height = Max(Node.Left.Height, Node.Right.Height) + 1$$

Balance Factor

1. **Balance Factor Equation:**
   $BF = Node.Left.Height - Node.Right.Height$
2. We use the balance factor equation to determine:
   1. When we need to rotate
   2. What type of rotation needs to be performed.
3. *Important!* If `left` or `right` are `null` we treat their height as -1.

AVL Balance Method: Right Rotations

Algorithm 1 AVL: Balance

```
procedure BALANCE(N)
    if BF(N) > 1 then
        if BF(N.left) < 0 then
            N = RotateLeftRight(N)
        else
            N = Right(N)
    else if BF(N) < -1 then
        if BF(N.right) > 0 then
            N = RotateRightLeft(N)
        else
            N = Left(N)
    Return N
```

Unbalanced with respect to left subtree (BF > 1).

AVL Balance Method: Right Rotations

Algorithm 2 AVL: Balance

```
procedure BALANCE(N)
    if BF(N) > 1 then
        if BF(N.left) < 0 then
            N = RotateLeftRight(N)
        else
            N = Right(N)
    else if BF(N) < -1 then
        if BF(N.right) > 0 then
            N = RotateRightLeft(N)
        else
            N = Left(N)
    Return N
```

AVL Balance Method: Left Rotations

Algorithm 3 AVL: Balance

```
procedure BALANCE(N)
    if BF(N) > 1 then
        if BF(N.left) < 0 then
            N = RotateLeftRight(N)
        else
            N = Right(N)
    else if BF(N) < -1 then
        if BF(N.right) > 0 then
            N = RotateRightLeft(N)
        else
            N = Left(N)
    Return N
```

Unbalanced with respect to right subtree (BF < -1).

AVL Balance Method: Left Rotations

Algorithm 4 AVL: Balance

```
procedure BALANCE(N)
    if BF(N) > 1 then
        if BF(N.left) < 0 then
            N = RotateLeftRight(N)
        else
            N = Right(N)
    else if BF(N) < -1 then
        if BF(N.right) > 0 then
            N = RotateRightLeft(N)
        else
            N = Left(N)
    Return N
```

Right Rotation

Algorithm 5 AVL: Right Rotation

```
procedure RIGHT(N)
    Tmp = N.Left
    N.Left = Tmp.Right;
    Tmp.Right = N

    UpdateHeight(N)
    UpdateHeight(Tmp)

    Return Tmp
```

Left Rotation

Algorithm 6 AVL: Left Rotation

```
procedure LEFT(N)
    Tmp = N.Right
    N.Right = Tmp.Left;
    Tmp.Left = N

    UpdateHeight(N)
    UpdateHeight(Tmp)

    Return Tmp
```

Left-Right Rotation

Algorithm 7 AVL: LeftRightRotate

```
procedure LEFTRIGHTROTATE(N)
    N.Left = Left(N.Left)
    Return Right(N)
```

Right-Left Rotation

Algorithm 8 AVL: RightLeftRotate

```
procedure RightLeftRotate(N)
    N.Right = Right(N.Right)
    Return Left(N)
```

Putting it all together...

Algorithm 9 AVL: Balance

```
procedure BALANCE(N)
    if BF(N) > 1 then
        if BF(N.left) < 0 then
            N = RotateLeftRight(N)
        else
            N = Right(N)
    else if BF(N) < -1 then
        if BF(N.right) > 0 then
            N = RotateRightLeft(N)
        else
            N = Left(N)
    Return N
```

```
bst.add(8)
bst.add(7)
bst.add(6)
bst.add(5)
bst.add(4)
bst.add(3)
bst.add(1)
```

AVL Remove: It's the same changes as Add!

Algorithm AVL: Remove

```
procedure REMOVE(CurrNode, RemovalNode)
                                                              1

    if CurrNode is Null then
        return Null

    if CurrNode < RemovalNode then
        CurrNode.Right = Remove(CurrNode.Right, RemovalNode)
    else if CurrNode > RemovalNode then
        CurrNode.Left = Remove(CurrNode.Left, RemovalNode)
    else
        if Curr.Left is Null then
            Return CurrNode.Right
        else if Curr.Right is Null then
            Return CurrNode.Left
        else
            MinNode = FindMinNode(CurrNode.Right)
            CopyData(MinNode, CurrNode)
            Curr.Right = Remove(CurrNode.Right, MinNode)

2   UpdateHeight(CurrNode)
    Return Balance(CurrNode)     3
```