CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

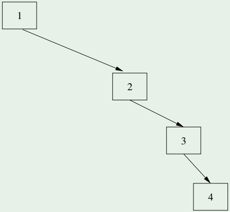
Outline

1. Section 13.3: Balanced Binary Search Trees
   - Balanced Binary Search Trees
   - AVL Trees
   - Conclusion
   - In-Class work

Improving The Worst-Case Performance for BSTs

The Worst Case Scenario

- In the worst case, a binary search tree looks like a linked list, with all the links going the same way.
- The performance of the important methods (find, insert,



delete) is $\Theta(n)$.

Improving The Worst-Case Performance for BSTs

Goal: Keeping Any BST "Balanced"

- Ideally, to prevent a BST from becoming too unbalanced, it would be filled so that as many nodes as possible have left and right subtrees. This would be equivalent to being a complete binary tree.
- This is impractical, since it would take too long to rearrange the nodes for the tree to keep this shape every time a new node gets added or deleted.

Improving The Worst-Case Performance for BSTs

A Workable Compromise

- We will only insist that, for a BST to be "balanced", any node will have the property that the depths of its left and right subtrees will differ by one level at most *(this solutions was developed by G.M. Adelson-Velskii and E.M. Landis in 1960s)*.
- This can be efficiently enforced each time a node is inserted or deleted.
- The worst case height is about $1.44 \lg(n)$.
- The performance of the insert, delete, and find operations is $\Theta(\lg n)$.

Basic Facts

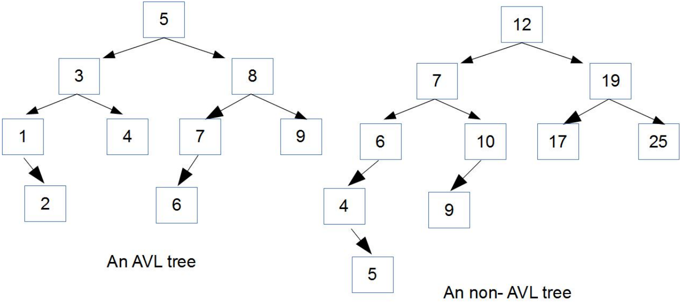
The AVL Tree property

An AVL tree is a binary search tree, with the additional property: for every node, the depths of its left and right subtrees can differ by at most one level.

Inventors

Such a tree is called an AVL Tree after its two co-inventors, Adelson-Velskii and Landis.

Basic Facts



AVL Trees: Insertion

Normal BST Insertion

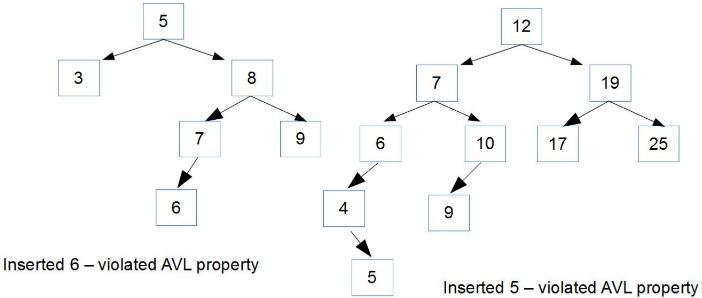
- A value gets inserted into a BST by comparing its value with the current node (starting with the root).
- If the value is less, it changes the current node to the left subtree if it exists.
- If the value is greater, it changes the current node to the right subtree if it exists.
- If the value is equal, an error has occurred: value is already in the tree.
- The new node is made a leaf when the subtree on that side doesn't exist.

AVL Trees: Insertion

When the AVL property is violated:

- we are **inserting a node into an existing leaf node**, and at nodes:

5 (Tree1) and 6(Tree2) AVL property is violated



Note that nodes **have a subtree with a depth at least two**.

AVL Trees: Insertion

AVL Insertion: Overview

- The height of each subtree is saved as a new attribute of every `TreeNode` object.
- Perform the insertion to the proper subtree, say, the left subtree.
- If the left subtree height is now 2 more than the right subtree, rebalance the tree at the current node.
- Similarly for the right subtree.
- Height of the current node = max(height left subtree, height right subtree)+1.

AVL Trees: Insertion

Modification of TreeNode class: Python

```
class TreeNode:
   def __init__(self, data=None, left=None, right=None, height=0):
      self.item = data
      self.left = left
      self.right = right
      self.height = height

   def get_height(t):
      if t is None:
         return -1
      else:
         return t.height
```

AVL Trees: Insertion

Modification of TreeNode class: C++

```
class TreeNode {
   friend int getHeight(TreeNode *t);

public:
   TreeNode(int item, TreeNode* left = NULL, TreeNode* right =
NULL);
   ~ TreeNode();

private:
   int _item;
   TreeNode* _left;
   TreeNode* _right;
   int _height;
};
```

AVL Trees: Insertion

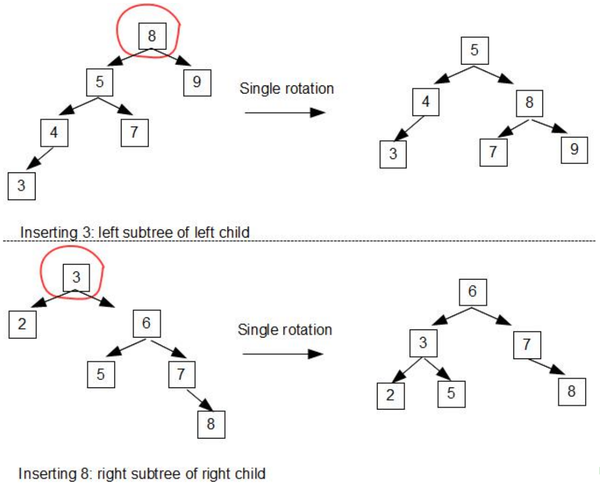
AVL property failure and rotations

Depending on the "direction" of insertion (after which AVL property fails) there are different rotations that re-balance the tree:

- inserting into the left subtree of the left child
  $\rightarrow$ single rotation
- inserting into the right subtree of the right child
  $\rightarrow$ single rotation
- inserting into the left subtree of the right child
  $\rightarrow$ double rotation
- inserting into the right subtree of the left child
  $\rightarrow$ double rotation

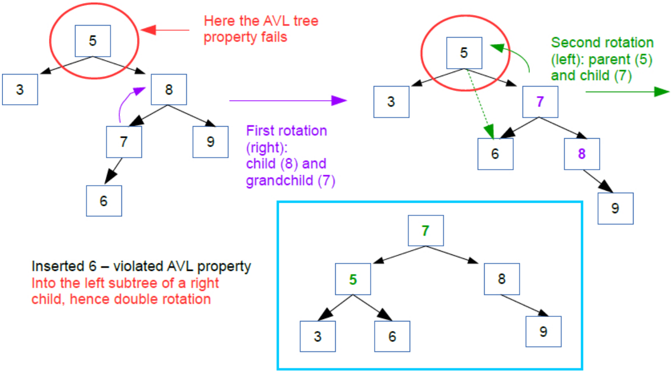
AVL Trees: Insertion

Single Rotations 2 cases:



AVL Trees: Insertion

Double rotation example:



AVL Trees: Insertion implementation

AVL Insertion : in Python

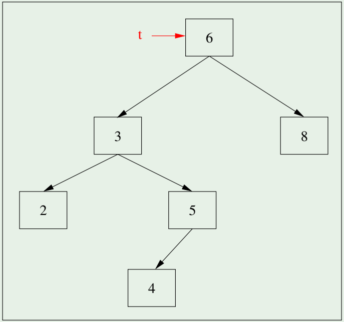
```
def insert(self, value):
    self.root = self._insert_help(self.root, value)
 def _insert_help(self, t, value):

    if t is None:
       t = TreeNode(value)
    elif value < t.item:# inserting into the left subtree
       t.left = self._insert_help(t.left, value)
       if get_height(t.left) - get_height(t.right) == 2:
          if value < t.left.item:  #left subtree of left child
             t = self._left_single_rotate(t)
          else:  t = self._right_left_rotate(t)
    else:  # exercise for reader, inserting into the right subtree
    t.height = max(get_height(t.left), get_height(t.right)) + 1
    return t
```

AVL Trees: Insertion implementation

AVL Rebalancing: Double Rotation

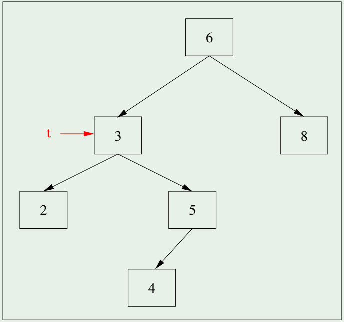
```
def _right_left_rotate(self, t):
  t.left =
self._right_single_rotate(t.left)
  t = self._left_single_rotate(t)
  return t
```



AVL Trees: Insertion implementation

AVL Right Subtree Insertion: Rebalancing at node t

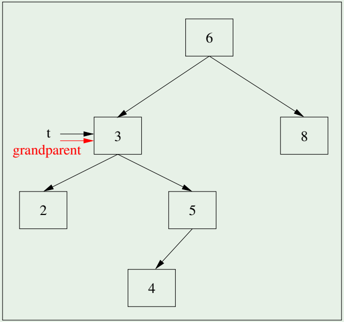
```
  def _right_single_rotate(self,
t):
   grandparent = t
   parent = t.right
   grandparent.right =
parent.left
   parent.left = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Right Subtree Insertion: Rebalancing at node t

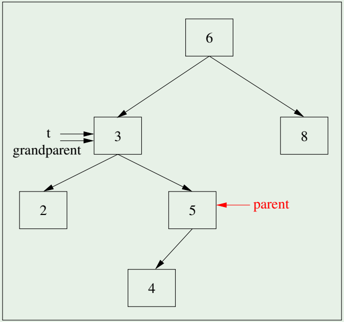
```
  def _right_single_rotate(self,
t):
   grandparent = t
   parent = t.right
   grandparent.right =
parent.left
   parent.left = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Right Subtree Insertion: Rebalancing at node t

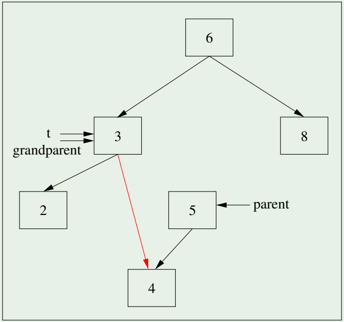
```
  def _right_single_rotate(self,
t):
   grandparent = t
   parent = t.right
   grandparent.right =
parent.left
   parent.left = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Right Subtree Insertion: Rebalancing at node t

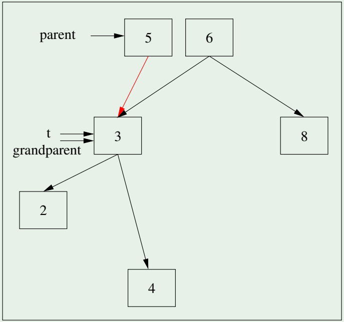
```
  def _right_single_rotate(self,
t):
   grandparent = t
   parent = t.right
   grandparent.right =
parent.left
   parent.left = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Right Subtree Insertion: Rebalancing at node t

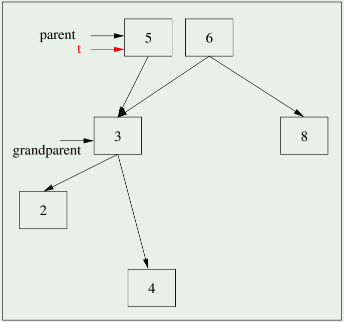
```
  def _right_single_rotate(self,
t):
   grandparent = t
   parent = t.right
   grandparent.right =
parent.left
   parent.left = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Right Subtree Insertion: Rebalancing at node t

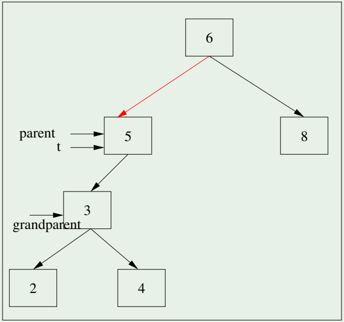
```
def _right_single_rotate(self, t):
    grandparent = t
    parent = t.right
    grandparent.right = parent.left
    parent.left = grandparent
    t = parent
    # adjust heights of grandparent, parent
    return t
```



AVL Trees: Insertion implementation

AVL Right Subtree Insertion: Rebalancing at node t

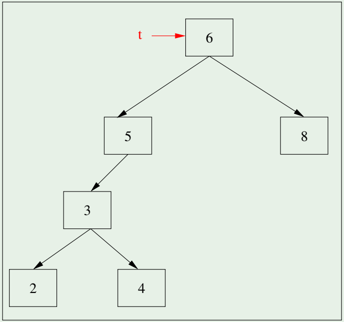
```
  def _right_single_rotate(self,
t):
    grandparent = t
    parent = t.right
    grandparent.right =
parent.left
    parent.left = grandparent
    t = parent
    # adjust heights of
grandparent, parent
    return t
```



AVL Trees: Insertion implementation

AVL Left Subtree Insertion: Rebalancing at node t

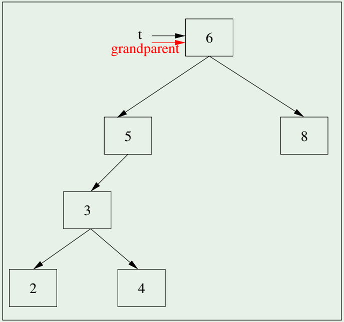
```
  def _left_single_rotate(self,
t):
   grandparent = t
   parent = t.left
   grandparent.left =
parent.right
   parent.right = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Left Subtree Insertion: Rebalancing at node t

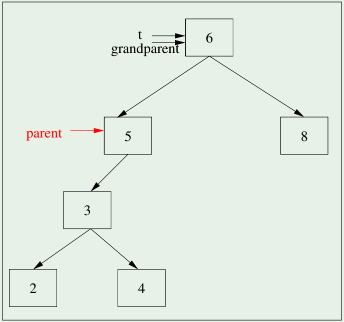
```
  def _left_single_rotate(self, t):
    grandparent = t
    parent = t.left
    grandparent.left = parent.right
    parent.right = grandparent
    t = parent
    # adjust heights of grandparent, parent
    return t
```



AVL Trees: Insertion implementation

AVL Left Subtree Insertion: Rebalancing at node t

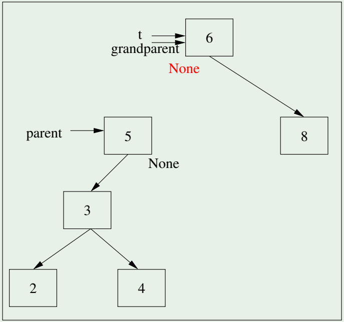
```
  def _left_single_rotate(self,
t):
   grandparent = t
   parent = t.left
   grandparent.left =
parent.right
   parent.right = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Left Subtree Insertion: Rebalancing at node t

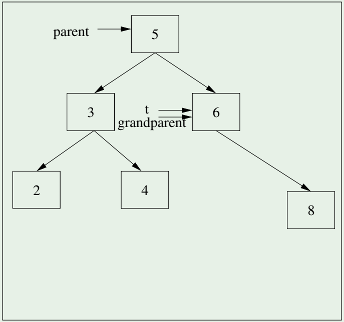
```
  def _left_single_rotate(self,
t):
    grandparent = t
    parent = t.left
    grandparent.left =
parent.right
    parent.right = grandparent
    t = parent
    # adjust heights of
grandparent, parent
    return t
```



AVL Trees: Insertion implementation

AVL Left Subtree Insertion: Rebalancing at node t

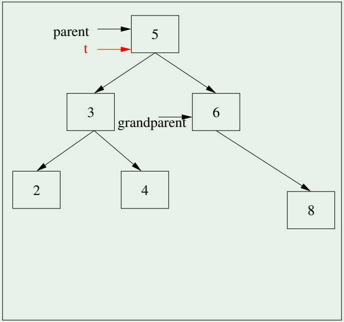
```
  def _left_single_rotate(self,
t):
   grandparent = t
   parent = t.left
   grandparent.left =
parent.right
   parent.right = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Left Subtree Insertion: Rebalancing at node t

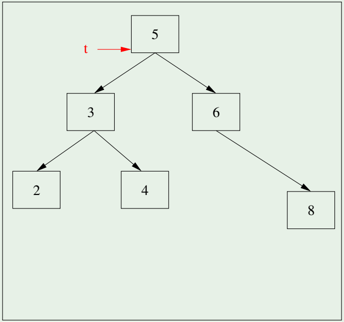
```
  def _left_single_rotate(self,
t):
   grandparent = t
   parent = t.left
   grandparent.left =
parent.right
   parent.right = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



AVL Trees: Insertion implementation

AVL Left Subtree Insertion: Rebalancing at node t

```
  def _left_single_rotate(self,
t):
   grandparent = t
   parent = t.left
   grandparent.left =
parent.right
   parent.right = grandparent
   t = parent
   # adjust heights of
grandparent, parent
   return t
```



What is done and what is not done

in Python:

The code of AVLTree is not complete. Two rotations are absent, conversion to list and generator are not defined.

Deletion operation is absent (and not even discussed in class)

in C++:

The code of AVLTree is complete.

AVL trees

Practicing on re-balancing

see CSI33-AVL-In-Class-Practice.pdf and avl_handout-CSVirginiaEDU.pdf