CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

November 20, 2017

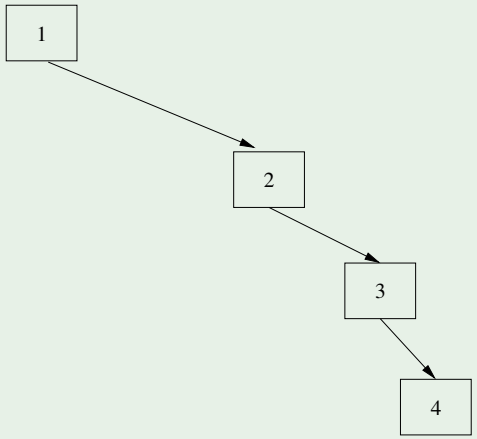
Outline

Outline

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 can be efficiently enforced each time a node is inserted or deleted.
- The worst case height is about $1.44 \log(n)$.
- The performance of the insert, delete, and find operations is $\Theta(\log n)$.

Basic Facts

The AVL Tree property

An AVL tree is a binary search tree (so it has the Binary Search Property), which has the additional AVL Tree Property that for every node, the depths of its left and right subtrees will differ by at most one level.

Basic Facts

The AVL Tree property

An AVL tree is a binary search tree (so it has the Binary Search Property), which has the additional AVL Tree Property that for every node, the depths of its left and right subtrees will differ by at most one level.

Inventors

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

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

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

AVL Insertion: Overview

```
  void AVLTree::insert(int value)
{
   _root = _insertRec(_root, value);
}
```

AVL Trees: Insertion

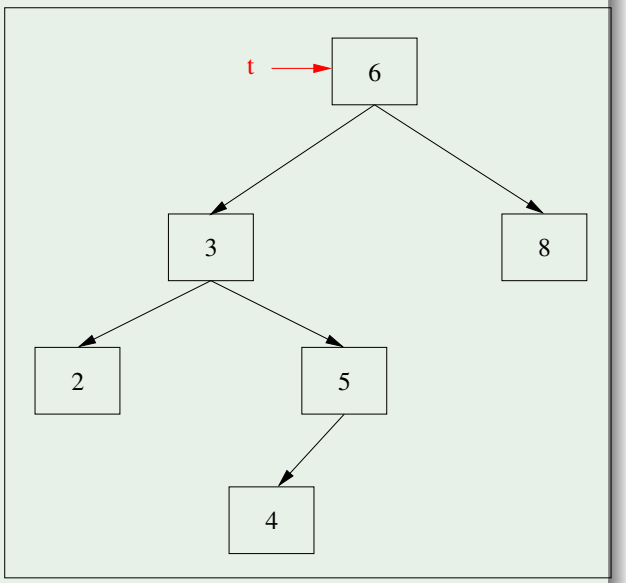
AVL Rebalancing

```
  TreeNode *AVLTree::_insertRec(TreeNode* t, int value)
{
   if (t == NULL)
      t = new TreeNode(value, NULL, NULL);
   else if (value < t->_item)
   {
      t->_left = _insertRec(t->_left, value);
      if (getHeight(t->_left) - getHeight(t->_right) == 2)
//rebalance?
      { // inserted into which subtree of left child?
         if (value < t->_left->_item)
            t = _leftSingleRotate(t); // left subtree
         else
            t = _rightLeftRotate(t); // right subtree
      }
   }
```

AVL Trees: Insertion

AVL Rebalancing: Double Rotation

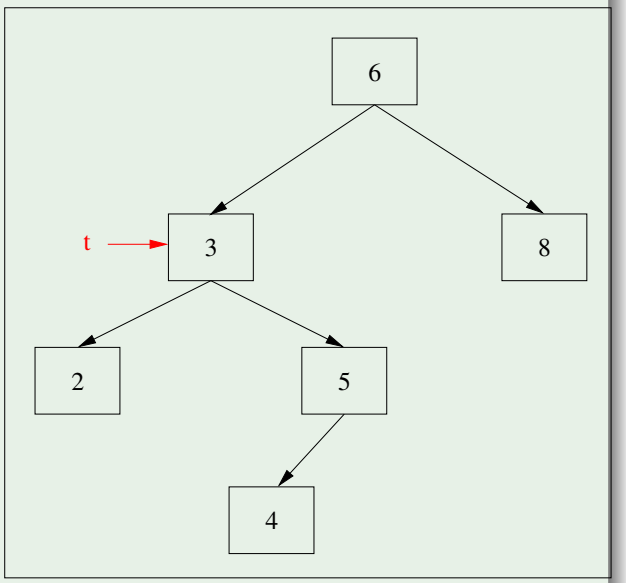
```
TreeNode *AVLTree::
        _rightLeftRotate(TreeNode *t)
{
  t->_left = _rightSingleRotate(t->_left);
  t = _leftSingleRotate(t);
  return t;
}
```



AVL Trees: Insertion

AVL Right Subtree Insertion: Rebalancing at node t

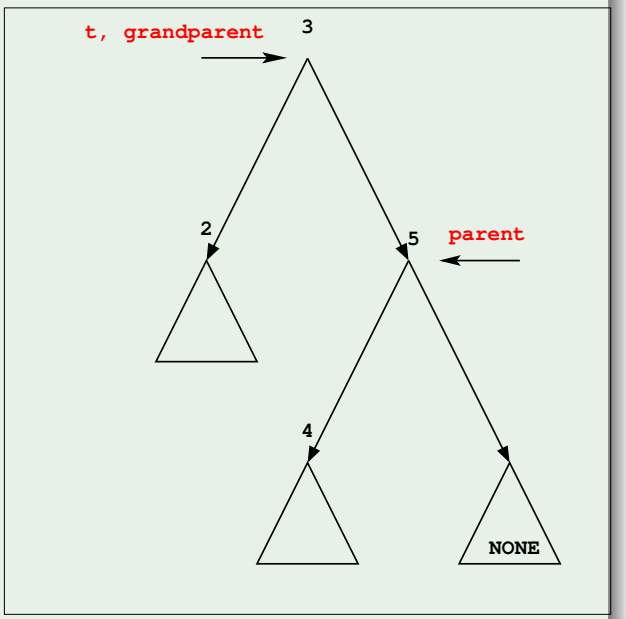
```
  TreeNode *AVLTree::
          _rightSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_right;
    grandparent->_right = parent->_left;
    parent->_left = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```



AVL Trees: Insertion

AVL Right Subtree Insertion: Rebalancing at node t

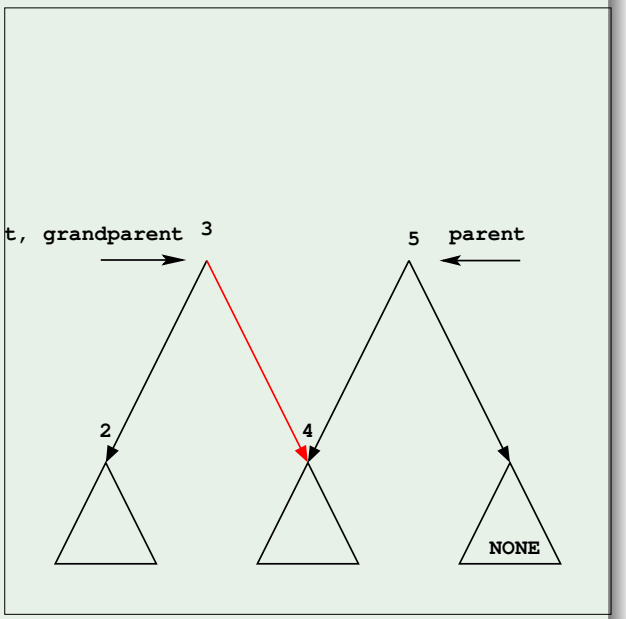
```
  TreeNode *AVLTree::
          _rightSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_right;
    grandparent->_right = parent->_left;
    parent->_left = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```



AVL Trees: Insertion

AVL Right Subtree Insertion: Rebalancing at node t

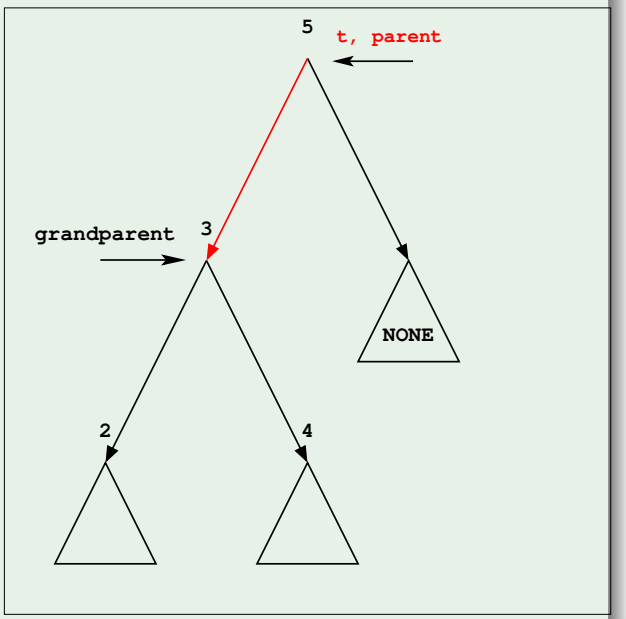
```
  TreeNode *AVLTree::
          _rightSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_right;
    grandparent->_right = parent->_left;
    parent->_left = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```



AVL Trees: Insertion

AVL Right Subtree Insertion: Rebalancing at node t

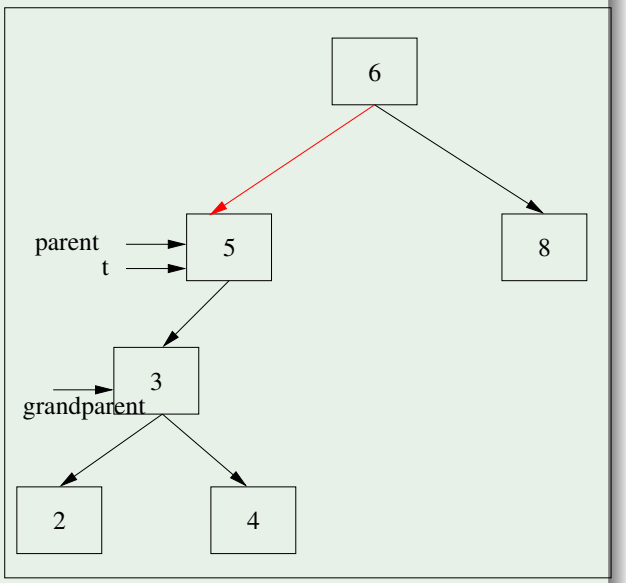
```
  TreeNode *AVLTree::
          _rightSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_right;
    grandparent->_right = parent->_left;
    parent->_left = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```



AVL Trees: Insertion

AVL Right Subtree Insertion: Rebalancing at node t

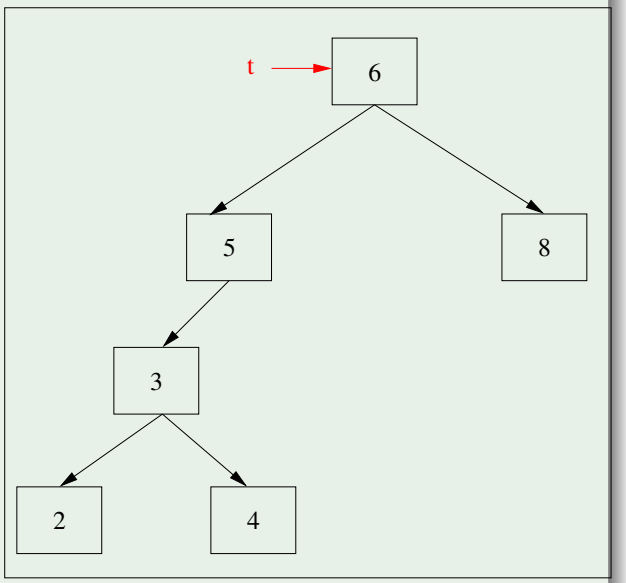
```
  TreeNode *AVLTree::
          _rightSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_right;
    grandparent->_right = parent->_left;
    parent->_left = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```



AVL Trees: Insertion

AVL Left Subtree Insertion: Rebalancing at node t

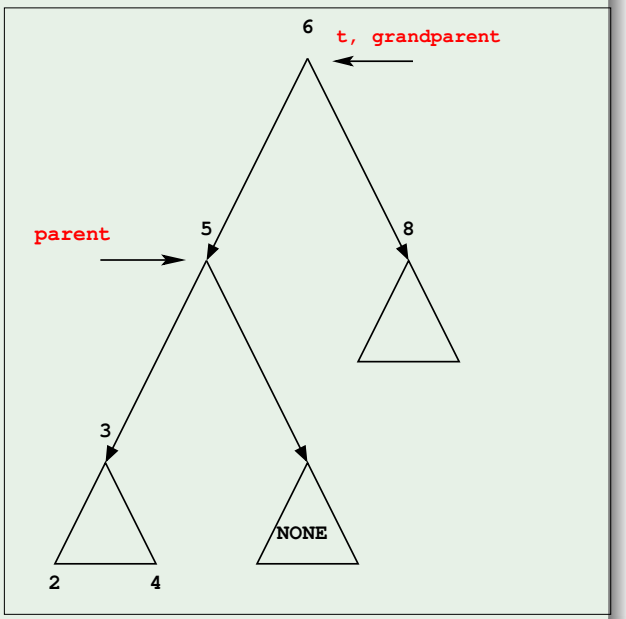
```
 TreeNode *AVLTree::
         _leftSingleRotate(TreeNode *t)
{
   TreeNode *grandparent = t;
   TreeNode *parent = t->_left;
   grandparent->_left = parent->_right;
   parent->_right = grandparent;
   t = parent;
   // adjust heights of grandparent,
parent
   return t;
}
```



AVL Trees: Insertion

AVL Left Subtree Insertion: Rebalancing at node t

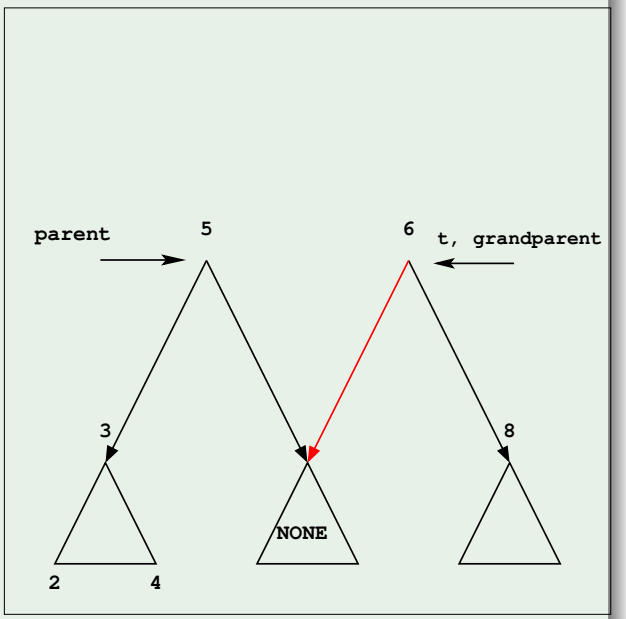
```
  TreeNode *AVLTree::
          _leftSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_left;
    grandparent->_left = parent->_right;
    parent->_right = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```



AVL Trees: Insertion

AVL Left Subtree Insertion: Rebalancing at node t

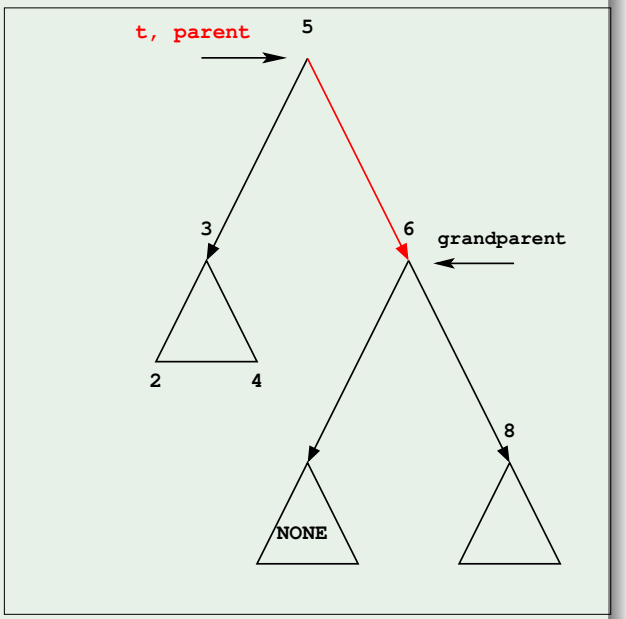
```
  TreeNode *AVLTree::
          _leftSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_left;
    grandparent->_left = parent->_right;
    parent->_right = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```



AVL Trees: Insertion

AVL Left Subtree Insertion: Rebalancing at node t

```
  TreeNode *AVLTree::
          _leftSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_left;
    grandparent->_left = parent->_right;
    parent->_right = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```



AVL Trees: Insertion

AVL Left Subtree Insertion: Rebalancing at node t

```
  TreeNode *AVLTree::
          _leftSingleRotate(TreeNode *t)
{
    TreeNode *grandparent = t;
    TreeNode *parent = t->_left;
    grandparent->_left = parent->_right;
    parent->_right = grandparent;
    t = parent;
    // adjust heights of grandparent,
parent
    return t;
}
```