Data Structures and Algorithms

Spring 2008-2009

Outline

Outline

1. BST Trees (contd.)
   - AVL Trees

The bad news about BSTs...

- ✗ Problem with BSTs is that there is nothing to stop them going badly skewed
- ✓ An *AVL* (Adelson-Velskii and Landis) tree is a BST with a height balance condition
- (There exist other balancing strategies based on the "weight" of a tree)
- Cannot just insist that the root be balanced:
- Also, cannot insist that *every* node have an equal height left and right subtree:
- → AVL requirement is that the height of every left and right subtree can differ by at most 1
- Height information is kept in `tree_node` object
- Insertions and deletions now become harder

The bad news about BSTs...

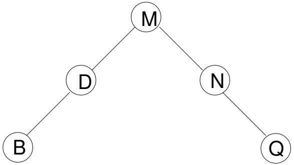
- ✗ Problem with BSTs is that there is nothing to stop them going badly skewed
- ✓ An *AVL* (Adelson-Velskii and Landis) tree is a BST with a height balance condition
- (There exist other balancing strategies based on the "weight" of a tree)
- Cannot just insist that the root be balanced:
- Also, cannot insist that *every* node have an equal height left and right subtree:
- ➜ AVL requirement is that the height of every left and right subtree can differ by at most 1
- Height information is kept in `tree_node` object
- Insertions and deletions now become harder

The bad news about BSTs...

- ✗ Problem with BSTs is that there is nothing to stop them going badly skewed
- ✓ An *AVL* (Adelson-Velskii and Landis) tree is a BST with a height balance condition
- (There exist other balancing strategies based on the "weight" of a tree)
- Cannot just insist that the root be balanced:
- Also, cannot insist that *every* node have an equal height left and right subtree:
- ➜ AVL requirement is that the height of every left and right subtree can differ by at most 1
- Height information is kept in `tree_node` object
- Insertions and deletions now become harder

The bad news about BSTs...

- ✗ Problem with BSTs is that there is nothing to stop them going badly skewed
- ✓ An *AVL* (Adelson-Velskii and Landis) tree is a BST with a height balance condition
- (There exist other balancing strategies based on the "weight" of a tree)
- Cannot just insist that the root be balanced:

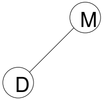


- Also, cannot insist that *every* node have an equal height left and right subtree:

➜ AVL requirement is that the height of every left and right subtree can differ by at most 1

The bad news about BSTs...

- ✗ Problem with BSTs is that there is nothing to stop them going badly skewed
- ✓ An *AVL* (Adelson-Velskii and Landis) tree is a BST with a height balance condition
- (There exist other balancing strategies based on the “weight” of a tree)
- Cannot just insist that the root be balanced:
- Also, cannot insist that *every* node have an equal height left and right subtree:



- ➜ AVL requirement is that the height of every left and right subtree can differ by at most 1
- Height information is kept in `tree_node` object

The bad news about BSTs...

- ✗ Problem with BSTs is that there is nothing to stop them going badly skewed
- ✓ An *AVL* (Adelson-Velskii and Landis) tree is a BST with a height balance condition
- (There exist other balancing strategies based on the “weight” of a tree)
- Cannot just insist that the root be balanced:
- Also, cannot insist that *every* node have an equal height left and right subtree:
- ➜ AVL requirement is that the height of every left and right subtree can differ by at most 1
- Height information is kept in `tree_node` object
- Insertions and deletions now become harder

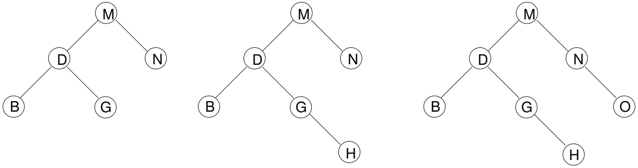
The bad news about BSTs...

- ✗ Problem with BSTs is that there is nothing to stop them going badly skewed
- ✓ An *AVL* (Adelson-Velskii and Landis) tree is a BST with a height balance condition
- (There exist other balancing strategies based on the "weight" of a tree)
- Cannot just insist that the root be balanced:
- Also, cannot insist that *every* node have an equal height left and right subtree:
- ➜ AVL requirement is that the height of every left and right subtree can differ by at most 1
- Height information is kept in `tree_node` object
- Insertions and deletions now become harder

The bad news about BSTs...

- ✗ Problem with BSTs is that there is nothing to stop them going badly skewed
- ✓ An *AVL* (Adelson-Velskii and Landis) tree is a BST with a height balance condition
- (There exist other balancing strategies based on the "weight" of a tree)
- Cannot just insist that the root be balanced:
- Also, cannot insist that *every* node have an equal height left and right subtree:
- → AVL requirement is that the height of every left and right subtree can differ by at most 1
- Height information is kept in `tree_node` object
- Insertions and deletions now become harder

AVL Tree Example



AVL Tree | Not an AVL Tree | AVL Tree

- In practise an AVL tree on n nodes has height of about $O(\log(n+1)) + 0.25$
- Compare to *best* BST (CBBST) which has height $k = \log(n+1) - 1$
- The worst the height can be is $1.44 \log n$ – found by reasoning on what is the minimum number of nodes that a tree of height, h, can have; we will return to this problem later

AVL Trees: Insertions

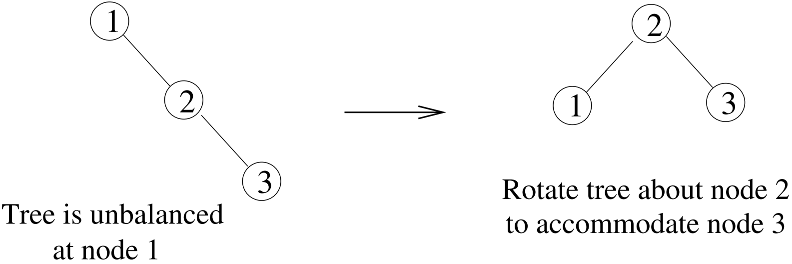
- When we insert a node in to an AVL tree we need to ensure that the tree remains balanced after the insertion
- Consider inserting values 1 to 7 in to an empty AVL
- Inserting values 1 and 2 are easy, but 3...
- Inserting 6:

AVL Trees: Insertions

- When we insert a node in to an AVL tree we need to ensure that the tree remains balanced after the insertion
- Consider inserting values 1 to 7 in to an empty AVL
- Inserting values 1 and 2 are easy, but 3...
- Inserting 6:

AVL Trees: Insertions

- When we insert a node in to an AVL tree we need to ensure that the tree remains balanced after the insertion
- Consider inserting values 1 to 7 in to an empty AVL
- Inserting values 1 and 2 are easy, but 3...

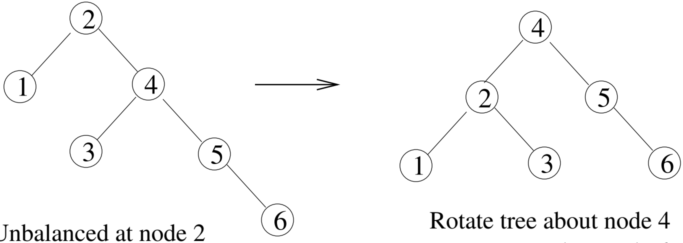


- Inserting 6:

AVL Trees: Insertions

- When we insert a node in to an AVL tree we need to ensure that the tree remains balanced after the insertion
- Consider inserting values 1 to 7 in to an empty AVL
- Inserting values 1 and 2 are easy, but 3...

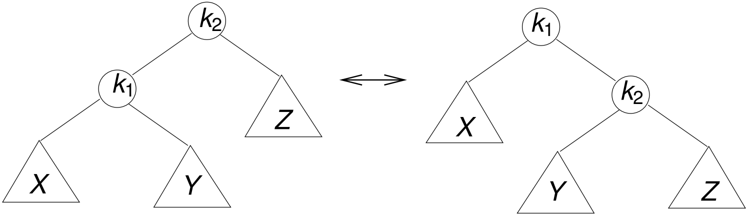
- Inserting 6:



Unbalanced at node 2

Rotate tree about node 4
to accommodate node 6

AVL Trees: Single Rotations



- If the height of k_2's left subtree is two more than the right subtree then there is a left imbalance and we perform a **left** rotation, giving the picture on the right
- If the height of k_2's right subtree is two more than the left subtree then there is a right imbalance and we perform a **right** rotation, giving the picture on the left

AVL Trees: Insertions (contd.)

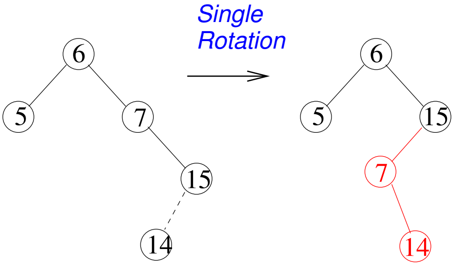
- After inserting 7, suppose we want to insert $15, 14, \ldots, 8$
- Inserting 14:

✗ A single rotation will not repair the balancedness of the tree so we need to perform a deeper twist

- This is called a *double* rotation
- There are two types of double rotation: *right-left* rotations and *left-right* rotations (think in terms of **imbalanced** side)
- A double rotation is the composition of two single rotations

AVL Trees: Insertions (contd.)

- After inserting 7, suppose we want to insert 15, 14, . . . , 8
- Inserting 14:



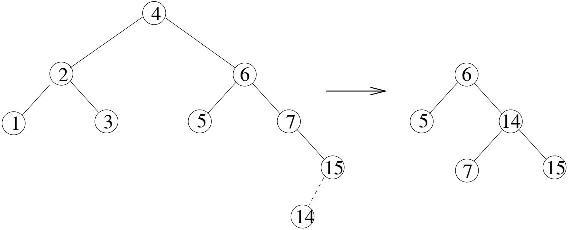
✗ A single rotation will not repair the balancedness of the tree so we need to perform a deeper twist

- This is called a *double* rotation
- There are two types of double rotation: *right-left* rotations and *left-right* rotations (think in terms of **imbalanced** side)
- A double rotation is the composition of two single rotations

AVL Trees: Insertions (contd.)

- After inserting 7, suppose we want to insert 15, 14, . . . , 8
- Inserting 14:

✗ A single rotation will not repair the balancedness of the tree so we need to perform a deeper twist

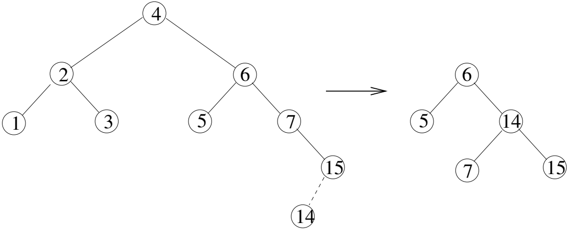


- This is called a *double* rotation
- There are two types of double rotation: *right-left* rotations and *left-right* rotations (think in terms of **imbalanced** side)
- A double rotation is the composition of two single rotations

AVL Trees: Insertions (contd.)

- After inserting 7, suppose we want to insert 15, 14, . . . , 8
- Inserting 14:

✗ A single rotation will not repair the balancedness of the tree so we need to perform a deeper twist



- This is called a *double* rotation
- There are two types of double rotation: *right-left* rotations and *left-right* rotations (think in terms of **imbalanced** side)
- A double rotation is the composition of two single rotations

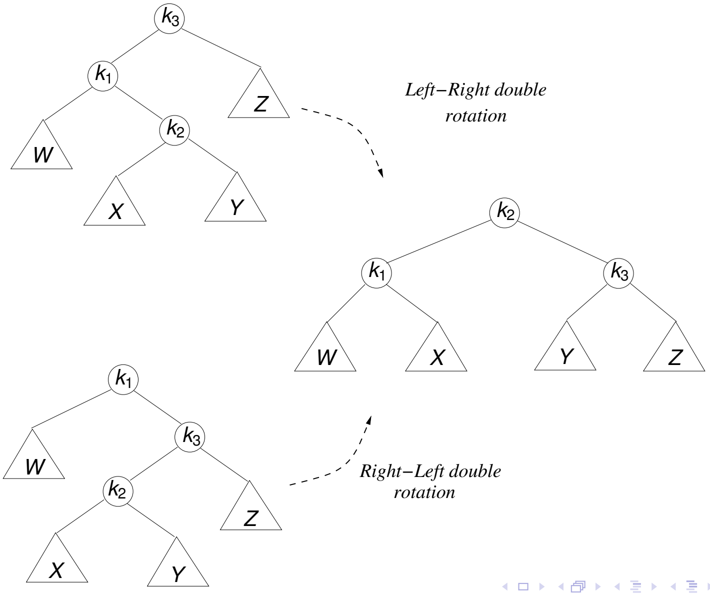
AVL Trees: Rotations (contd.)

- A left-right rotation is the composition of a right rotation on the subtree rooted at k_1 followed by a left rotation on the subtree rooted at k_3 (upper case on picture over)
- A right-left rotation is the composition of a left rotation on the subtree rooted at k_3 followed by a right rotation on the subtree rooted at k_1 (lower case on picture over)

AVL Trees: Rotations (contd.)

- A left-right rotation is the composition of a right rotation on the subtree rooted at k_1 followed by a left rotation on the subtree rooted at k_3 (upper case on picture over)
- A right-left rotation is the composition of a left rotation on the subtree rooted at k_3 followed by a right rotation on the subtree rooted at k_1 (lower case on picture over)

AVL Trees: Rotations (contd.)



AVL Tree: Deletions

- Just as with BSTs, deletions are similar to, but more tricky than, insertions
- The *actual* deletion is simply that described earlier for the BST
- However, shifting the right subtree's leftmost descendant to maintain the "inorderedness" may cause an imbalance
- Restoring the balance (through rotations) may cause imbalance further up the tree

➜ While it will never take more than *one* single or double rotation to rebalance a tree after an insertion, we may need $\lfloor \frac{h}{2} \rfloor$ rotations to rebalance a tree of height h after a deletion