CPSC 223
Algorithms & Data Abstract Structures

Lecture 18:
AVL Trees (cont.)

Today …

• In-place mergesort
• Midterm overview
• AVL Trees (cont.) [Ch 12: pp. 681-686]
• Heapsort exercise
Midterm Overview …

Midterm

- There will be 6-7 questions
- Open book / open notes
- Closed computer / smart phone / etc.
- Be sure you understand:
  - Answers to quizzes and exercises
  - Answers to questions on slides
- Be sure you have done the reading assignments
- Worth 10% of your grade
Topics we’ve covered since last time

• More on analysis of algorithms
  – Constant factors
  – Different examples of $O(\log n)$

• More on sorting
  – Mergesort (array-based and linked-list in-place)
  – Quicksort
  – Treesort
  – Heapsort

Topics we’ve covered since last time

• Binary trees
  – Basic terminology (root, parent, child, subtree)
  – Paths, tree height
  – Full, complete, balanced

• Binary search trees
  – Ordering constraint
  – Tree traversals
  – Lookup, Retrieve, Insert, Delete (Remove)
  – Pointer-based implementation
Topics we’ve covered since last time

• Binary search trees (cont.)
  – The costs of operations
  – Degenerate cases

• Heaps
  – Structure and ordering constraint
  – Array-based representation
  – FindMax, DeleteMax, Insert
  – Trickle down (delete) and trickle up (insert)
  – Heapsort (again)

Questions?

Topics we’ve covered since last time

• AVL trees
  – Why balanced search trees are good
  – Balance factors
  – Rotations (single, double)
  – Expect at most one question on AVL trees

Questions?
AVL Trees

AVL Trees 

[Adelson-Velskii & Landis, 1962]

• Use “tree rotations” to rebalance the tree
• Do tree rotations (if needed) after insert or delete
• Four cases:
  – Single rotation (“left-left”)
  – Single rotation (“right-right”)
  – Double rotation (“left-right”)
  – Double rotation (“right-left”)
• Traverse up the tree from inserted/deleted node
  – Only necessary if an insertion/deletion changes the balance
• Compute a “balance factor” at each node
AVL Trees

- General case for single rotation ("left-left")
  - Insertion in the left subtree of the left child of $k_2$ (subtree A)
  - Tree balanced before insertion
  - And becomes unbalanced after insertion

The rotation sets the subtree to its original height!

- We are done with insert after the rotation
  - The rotation sets the subtree (now rooted at $k_1$) to its original “balanced” height …
  - So no more rotations needed!

The rotation sets the subtree to its original height!
AVL Trees

• General case for single rotation (“right-right")
  – Insertion in the right subtree of the right child of $k_2$ (subtree A)
  – This is just the “mirror image” of the left-left case

The rotation sets the subtree to its original height!

AVL Trees -- Exercise

// return new root after left-left rotation
Node* rotateWithLeftChild(Node* k2)
{
  ...
}

// return new root after right-right subtree
Node* rotateWithRightChild(Node* k1)
{
  ...
}
AVL Trees -- Exercise

// return new root after left-left rotation
Node* rotateWithLeftChild(Node* k2)
{
    Node* k1 = k2->leftChild;
    k2->leftChild = k1->rightChild;
    k1->rightChild = k2;
    return k1;
}

AVL Trees -- Exercise

// return new root after right-right rotation
Node* rotateWithRightChild(Node* k1)
{
    Node * k2 = k1->rightChild;
    k1->rightChild = k2->leftChild;
    k2->leftChild = k1;
    return k2;
}
AVL Trees

A single rotation might not rebalance the tree …

Oops ... this (left-left) rotation didn’t help!!!
This is because we inserted into the right subtree of the left node

AVL Trees

• We sometimes need 2 rotations …
• General case for double rotation (left-right)

One of these has the inserted node
AVL Trees

- General case for double rotation (left-right) continued …

```
// return new root of left-right rotation
Node* doubleRotateWithLeftChild(Node* k3)
{
    ...
}
```

AVL Trees – Double Rotation (“left-right”)

```
// return new root of left-right rotation
Node* doubleRotateWithLeftChild(Node* k3)
{
    ...
}
```

// return new root of left-right rotation
Node* doubleRotateWithLeftChild(Node* k3)
{
    ...
}
AVL Trees – Double Rotation ("left-right")

// return new root of left-right rotation
Node* doubleRotateWithLeftChild(Node* k3)
{
    k3->leftChild = rotateWithRightChild(k3->leftChild);
    return rotateWithLeftChild(k3);
}

AVL Trees – Double Rotation ("right-left")

// return new root of right-left rotation
Node* doubleRotateWithRightChild(Node* k1)
{
    k1->rightChild = rotateWithLeftChild(k1->rightChild);
    return rotateWithRightChild(k1);
}
AVL Tree Insertion & Deletion Cost

• Cost
  – The cost for a single or double rotation is $O(1)$
  – The total cost is $O(\log n)$ since we have to traverse the tree along a path from leaf to root
  – So, $O(\log n)$ to insert + $O(\log n)$ to rebalance
  – But insertion/deletion still remains $O(\log n)$!
    • Compare to $O(n\log n)$ in our brute force approach

• For insertions
  – Once we rebalance a subtree, we are done …
  – no need to continue rebalancing
• This is *not always the case* for deletions …

AVL Tree Deletion

• Unlike insertions, for deletions we sometimes have to keep traversing up the tree after a rotation
  – Note that this doesn’t change the $O(\log n)$ deletion time
Comments on AVL Trees

• AVL vs other approaches
  – Rotations and traversals are hard to get right
  – But more importantly, the traversals create overhead
• A Red Black tree is an alternative self-balancing approach
  – In practice, (slightly) faster insertion and deletion
  – But (slightly) slower retrieval time
  – Red-black trees are often implemented in practice