CompSci 100
Prog Design and Analysis II

October 21, 2010
Prof. Rodger
Announcements

• Apt-five due today, apt-six due next week
• 20 Questions assignment out, due Nov 2.
Trees: no data structure lovelier?
Plan of Action: re trees

• Trees from top to bottom
  – Why trees are useful, tradeoffs in types of trees
  – How trees are implemented
  – Where trees are used, data structures and algorithms

• We’ll concentrate on binary trees
  – A tree can be empty
  – A tree consists of a (root (subtree) (subtree))
  – Analyzing tree functions with *recurrences*

• Other types of trees from a 30,000 ft view
From doubly-linked lists to binary trees

- Re-imagine prev/next, no longer linear
  - Similar to binary search, everything less goes left, everything greater goes right
  - How do we search?
  - How do we insert?
Binary Trees

• Search and insert: toward the best of both worlds
  – Linked list: efficient insert/delete, inefficient search
  – ArrayList: efficient (binary) search, but shift

• Binary trees: efficient insert, delete, and search
  – trees used in many contexts, not just for searching,
    • Game trees, collisions, ...
    • Cladistics, genomics, quad trees, ...
  – search in $O(\log n)$ like sorted array
    • Average case, worst case can be avoided!
  – insertion/deletion $O(1)$ like list, once location found
A TreeNode by any other name...

• What does this look like?
  – What does the picture look like?

```java
public class TreeNode {
    TreeNode left;
    TreeNode right;
    String info;
    TreeNode(String s,
              TreeNode llink, TreeNode rlink){
        info = s;
        left = llink;
        right = rlink;
    }
}
```
Printing a search tree in order

- **When is root printed?**
  - After left subtree, before right subtree.

```java
void visit(TreeNode t){
    if (t != null) {
        visit(t.left);
        System.out.println(t.info);
        visit(t.right);
    }
}
```

- **Inorder traversal**
Tree traversals

• Different traversals useful in different contexts
  – Inorder prints search tree in order
    • Visit left-subtree, process root, visit right-subtree
  – Preorder useful for reading/writing trees
    • Process root, visit left-subtree, visit right-subtree
  – Postorder useful for destroying trees
    • Visit left-subtree, visit right-subtree, process root
Barbara Liskov

• First woman to earn PhD from CompSci dept
  – Stanford
• Turing award in 2008
  ● OO, SE, PL, Clu

“It's much better to go for the thing that's exciting. But the question of how you know what's worth working on and what's not separates someone who's going to be really good at research and someone who's not. There's no prescription. It comes from your own intuition and judgment.”
Basic tree terminology

• Binary tree is a structure:
  – empty
  – root node with left and right subtrees

• Tree Terminology
  – parent and child: A is parent of B, E is child of B
  – leaf node has no children, internal node has 1 or 2 children
  – path is sequence of nodes (edges), N₁, N₂, ... Nₖ
    • Nᵢ is parent of Nᵢ₊₁
  – depth (level) of node: length of root-to-node path
    • level of root is 1 (measured in nodes)
  – height of node: length of longest node-to-leaf path
    • height of tree is height of root
Tree functions

• Compute height of a tree, what is complexity?
  
  ```java
  int height(Tree root) {
    if (root == null) return 0;
    else {
      return 1 + Math.max(height(root.left),
                           height(root.right));
    }
  }
  ```

• Modify function to compute number of nodes in a tree, does complexity change?
  – What about computing number of leaf nodes?
Balanced Trees and Complexity

• A tree is height-balanced if
  – Left and right subtrees are height-balanced
  – Left and right heights differ by at most one

boolean isBalanced(Tree root){
  if (root == null) return true;
  return
    isBalanced(root.left) && isBalanced(root.right) &&
    Math.abs(height(root.left) - height(root.right)) <= 1;
}
What is complexity?

• Assume trees are “balanced” in analyzing complexity
  – Roughly half the nodes in each subtree
  – Leads to easier analysis

• How to develop recurrence relation?
  – What is T(n)?
  – What other work is done?

• How to solve recurrence relation
  – Plug, expand, plug, expand, find pattern
  – A real proof requires induction to verify correctness
Recurrences

• If $T(n) = T(n-1) + O(1)$... where do we see this?
$T(n) = T(n-1) + O(1)$

  *true for all $X$ so, $T(n-1) = T(n-2) + O(1)$
$T(n) = [T(n-2) + 1] + 1 = T(n-2) + 2$
  $= [T(n-3) + 1] + 2 = T(n-3) + 3$

• True for 1, 2, so eureka! We see a pattern
$T(n) = T(n-k) + k$, true for all $k$, let $n=k$
$T(n) = T(n-n) + n = T(0) + n = n$

• We could solve, we could prove, or remember!
Recognizing Recurrences

- Solve once, re-use in new contexts
  - T must be explicitly identified
  - n must be some measure of size of input/parameter
- T(n) is for quicksort to run on an n-element array

\[
\begin{align*}
T(n) &= T(n/2) + \Omega(1) & \text{binary search} & \Theta(1) \\
T(n) &= T(n-1) + \Omega(1) & \text{sequential search} & \Theta(1) \\
T(n) &= 2T(n/2) + \Omega(1) & \text{tree traversal} & \Theta(1) \\
T(n) &= 2T(n/2) + \Omega(n) & \text{quicksort} & \Theta(n) \\
T(n) &= T(n-1) + \Omega(n) & \text{selection sort} & \Theta(n)
\end{align*}
\]

- Remember the algorithm, re-derive complexity