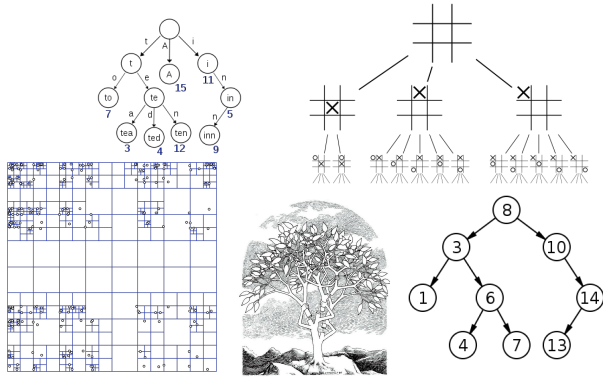


Trees: no data structure lovelier?

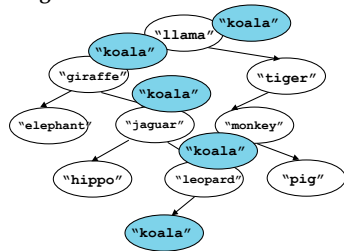


Binary Trees

- 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*

From doubly-linked lists to binary trees

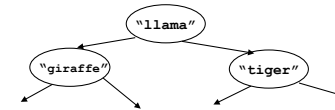
- Re-imagine prev/next, no longer linear
 - > Similar to binary search, everything less goes left, everything greater goes right
 - > How do we search?
 - > How do we insert?



A TreeNode by any other name...

- What does this look like?
 - > What does the picture look like?

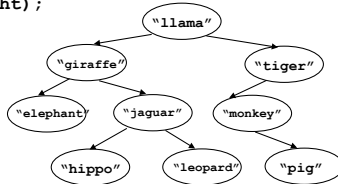
```
public class TreeNode
{
    TreeNode left;
    TreeNode right;
    String info;
    TreeNode(String s,
             TreeNode llink, TreeNode rlink) {
        info = s;
        left = llink;
        right = rlink;
    }
}
```



Printing a search tree in order

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

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



- *Inorder traversal*

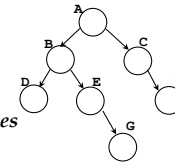
CPS 100, Fall 2009

9.5

Basic tree definitions

- 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), N_1, N_2, \dots, N_k
 - N_i is parent of N_{i+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

CPS 100, Fall 2009

9.6

Insertion and Find? Complexity?

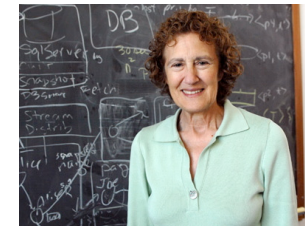
- Search for a value in a search tree, start at root?
 - Can do this both iteratively and recursively, contrast to printing which is very difficult to do iteratively
 - How is insertion similar to search?
- What is complexity of print? Of insertion?
 - Is there a worst case for trees?
 - Do we use best case? Worst case? Average case?
- How do we define worst and average cases
 - For trees? For vectors? For linked lists? For arrays of linked-lists?

CPS 100, Fall 2009

9.7

Barbara Liskov

- First woman to earn PhD from compsci dept
 - Stanford
- Turing award in 2008
 - OO, SE, PL



"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."

CPS 100, Fall 2009

9.8

See SetTiming code

- What about *ISimpleSet* interface
 - How does this compare to java.util?
 - Why are we looking at this, what about Java source?
- What does a simple implementation look like?
 - What are complexity repercussions: add, contains
 - What about iterating?
- Scenarios where linked lists better?
 - Consider N adds and M contains operations
 - Move to front heuristic?

What does contains look like?

```
public boolean contains(E element) {
    return myList.indexOf(element) >= 0;
}
public boolean contains(E element){
    returns inlist(myHead, element);
}
private boolean inlist(Node list,E element)
{
    if (list == null) return false;
    if (list.info.equals(element))
        return true;
    return contains(list.next,element);
}
```

What does (tree) contains look like?

```
public boolean contains(E element){
    returns intree(myRoot, element);
}
private boolean intree(TreeNode root,
    E element) {
    if (root == null) return false;
    if (list.info.equals(element))
        return true;
    if (element.compareTo(root.info) <= 0){
        return contains(root.left,element);
    }
    else
        return contains(root.right,element);
}
```

What does insertion look like?

- Simple recursive insertion into tree (accessed by root)
 - root = insert("foo", root);

```
TreeNode insert(TreeNode t, String s) {
    if (t == null) t = new Tree(s,null,null);
    else if (s.compareTo(t.info) <= 0)
        t.left = insert(t.left,s);
    else
        t.right = insert(t.right,s);
    return t;
}
```

Notes on tree insert and search

- Note: in each recursive *insert* call, the parameter *t* in the called clone is either the left or right pointer of some node in the original tree
 - Why is this important?
 - Why must the idiom `t = treeMethod(t, ...)` be used?
- When good trees go bad, what happens and why?
 - Insert alpha, beta, gamma, delta, epsilon, ...
 - Where does gamma go?
 - Can we avoid this case? Yes!
 - What to prefer? Long and stringy or short and bushy

Removal from tree?

- For insertion we can use iteration (see BSTSet)
 - Look below, either left or right
 - If null, stop and add
 - Otherwise go left when \leq , else go right when $>$
- Removal is tricky, depends on number of children
 - Straightforward when zero or one child
 - Complicated when two children, find successor
 - See set code for complete cases
 - If right child, straightforward
 - Otherwise find node that's left child of its parent (why?)

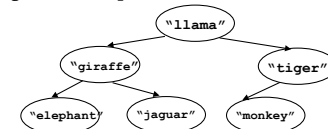
Tree functions

- Compute height of a tree, what is complexity?

```
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?

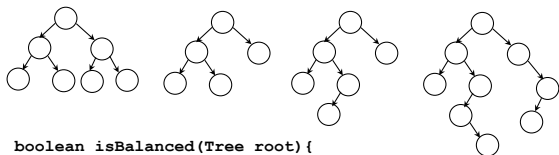
Tree traversals

- 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



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 euraka! We see a pattern

$$T(n) = T(n-k) + k, \text{ true for all } k, \text{ let } n=k$$

$$T(n) = T(n-n) + n = T(0) + n = n$$

- 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

$$T(n) = T(n/2) + O(1) \quad \text{binary search} \quad O(\log n)$$

$$T(n) = T(n-1) + O(1) \quad \text{sequential search} \quad O(n)$$

$$T(n) = 2T(n/2) + O(1) \quad \text{tree traversal} \quad O(n)$$

$$T(n) = 2T(n/2) + O(n) \quad \text{quicksort} \quad O(n \log n)$$

$$T(n) = T(n-1) + O(n) \quad \text{selection sort} \quad O(n^2)$$

- Remember the algorithm, re-derive complexity