PFTWBH

- More examples of recursion and alternatives
  - Blob counting and applications of neighborhood search/Flood Fill
  - Using Queue to solve without recursion

- Review some concepts from Recitation/Discussion
  - Recursive code analyzed

- Understanding the DNA Assignment
- Binary Search trees via Doubly Linked Lists

Blob Counting, Flood Fill

- Flood a region with color
  - Erase region, make transparent
  - How do find the region?

- Finding regions, blobs, edges, ...
  - See blob counting code
  - What is a blob?

- Recursion helps, but necessary?
  - Performance, clarity, ...
  - Ease of development

BlobCount or edge detection or ...

- How do we find images? Components? Paths?
  - [https://git.cs.duke.edu/201fall16/blobstuff/tree/master/src](https://git.cs.duke.edu/201fall16/blobstuff/tree/master/src)

Running Blobs and BlobModel

- Initialize with number of random pixels/points
  - Could also read a file of scanned gels from a lab!
  - Use Random with no seed to get different runs

- Count and label blobs of minimal size
  - Blobs contiguous horizontal/vertical connection
  - Large blobs not as plentiful as small blobs
  - Run example and look for different sizes

- We'll dissect code to understand algorithm and implementation
Counting Blobs, Flood Fill

  - See animation on page!
- The general idea is to "color" a pixel, then visit adjacent pixels and color them
  - Repeat same process for adjacent pixels
- The core of recursion - similar task, smaller problem
  - One pixel done, many to go

---

Details and Idioms in blob code

- Method `blobFill` has four parameters
  - `(row, column)` of where search starts
  - Character being searched for (initially * or blob)
  - Character to fill with on success (e.g., count '2' or '4')
    - Mark for visualization
    - Mark to ensure we don’t search again!
- If `(row, column)` is part of blob, count it and ask neighbors for their counts
  - They’re part of blob (if never visited before)
- Return total of yourself and neighbors
  - Key to recursion: do one thing and ask for help

---

Ideas behind blob fill code

- Ask your neighbors
  - Return blob size
  - Ensure no re-counts
  - Sum and return
- What do neighbors do?
  - Same thing!
  - Colors indicate calls

---

Examine code in Eclipse/Git

- `int size = blobFill(5, 7, '*', '8')`
  - Start at `(row, col) == (5,7)`, look for '*', fill with '8'

```java
protected int blobFill(int row, int col, char lookFor, char fillWith) {
    int size = 0;
    if (inRange(row, col)) {
        if (myGrid[row][col] != lookFor)
            return 0;
        myGrid[row][col] = fillWith; // mark pixel
        size = 1; // count this pixel, then scout for neighbors
        size += blobFill(row - 1, col, lookFor, fillWith) +
                blobFill(row + 1, col, lookFor, fillWith) +
                blobFill(row, col - 1, lookFor, fillWith) +
                blobFill(row, col + 1, lookFor, fillWith);
    }
    return size;
}
```
Blob questions

- What changes if diagonal cells are adjacent?
  - Conceptually and in code

- How do we find blob sizes in a range?
  - Not bigger than X, but between X and Y

- How would we number blobs by size rather than by when they’re found? Bigger blobs labeled with one kind of character, smaller with different
  - Do we have the tools to do this in existing code?

- Can we avoid recursion and do this iteratively?

Iterative Blob Counting

- Conceptually: Use a queue of elements in one blob
  - Enqueue the first element of a blob
  - Put its neighbors on the queue (one step away)
  - Dequeue element, enqueue neighbors if in blob
    - But do not enqueue element seen before

- Programmatically: re-use most of BlobModel
  - Change two methods, use inheritance
  - More on this later, see @Override in code
  - These methods use Iterative Version

BlobModel and IterativeBlobModel

- Iterative inherits some methods, use those
  - Including findBlobs, addView

- Override methods, see findBlobs calling blobFill
  - Where is blobFill method? It depends

- Basic technique of determining which method to call at runtime
  - Cornerstone of object oriented programming
  - Studied more in later courses

Inheritance concepts

- Parent/super class can be extended by subclassing
  - Possible to use methods from parent class, subs have them!
  - Can override parent methods, change behavior
  - Possible to do both! Call super.doThis();

- Often you don't have access to parent .java file
  - Can subclass, use methods, extend/override them
  - Do NOT have access to private data fields
  - DO have access to protected data fields
Inheritance in General and in Java

- Hard to do OO design, leave for later courses
  - But get an idea now as to power and utility

- There is a base class, in Java it's Object. Everything inherits from Java
  - Object has implementations, override them, e.g., toString, hashCode, equals, more

- Determine of method calls happens at runtime, the actual type is determined at runtime rather than at compile time (think about this)

floodFill and blobCounting

- In recursive version: four recursive calls
  - Use result of each, sum to get total
  - If we want to include diagonals? Eight calls!

- In iterative version: four neighbors enqueued
  - See the offset/delta code as short-hand, could have used this in recursive version as well

- Instance variable myQueue – should it be local?
  - Is it used in more than one method?
  - Does it maintain state across more than one call?

WOTO questions


Sometimes recursion helps, sometimes not so much, but practice is good

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.”
How do we verify? How do we analyze?

- [1,2,2,3,3,4,4,4,4,...,7,7,7,7,7,7,7]

Correctness and Performance

```java
public Node createGaussList(int n){
    if (n == 1) return new Node(1,null);
    Node first = createGaussList(n-1);
    Node last = first;
    while (last.next != null){
        last = last.next;
    }
    last.next = createNlist(n);
    return first;
}
```

Doubly Linked Lists

- Why do we have some lists with nodes to previous and next nodes?
  - Easier to write code, don't need before, current, after...
  - See examples in Recitation
  - Used in Java through Java 7 for HashMap
  - Still used in LinkedList, easy traversal from front or back

- See DNA LinkStrand with singly linked list

Trees: no data structure lovelier?

Plan of Action: re trees

- Trees from top to bottom (seen some already)
  - 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
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
  - Not just for searching, used in many contexts,
    - Game trees, collisions, ...
    - Cladistics, genomics, quad trees, ...
  - search in $O(\log n)$ like sorted array
    - Average case. Note: worst case can be avoided!
  - insertion/deletion $O(1)$, 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
Good Search Trees and Bad Trees

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

Review: 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), $N_1, N_2, ..., N_k$
    - $N_i$ 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

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

```java
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 "balanced" in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis

- How to develop recurrence relation?
  - What is T(n)? Time `func` executes on n-node tree
  - What other work? Express recurrence, solve it

- How to solve recurrence relation
  - Plug, expand, plug, expand, find pattern
  - Proof requires induction to verify correctness

Recurrence relation

- Let T(n) be time for `height` to execute (n-node tree)
  - T(n) = T(n/2) + T(n/2) + O(1)
  - T(n) = 2T(n/2) + 1
  - T(n) = 2(2T(n/4) + 1) + 1
  - T(n) = 4T(n/4) + 2 + 1
  - T(n) = 8T(n/8) + 4 + 2 + 1, eureka! Pattern!
  - T(n) = 2^kT(n/2^k) + 2^k-1 why is this true?
  - T(n) = nT(1) + O(n) is O(n), if we let n=2^k

- Let T(n) be time for `isBalanced` on n-node tree
  - T(n) = 2T(n/2) + O(n), why? Solution?

https://git.cs.duke.edu/201fall16/set-examples/tree/master

- What about `ISimpleSet` interface
  - How does this compare to java.util?
  - What about Java source? Can we look at it?

- 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 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?
  - The idiom t = treeMethod(t, ...) 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/stringy or short/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 <=, 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?)