Midterm TA evals coming up!

Please please complete the eval when you hear about it.

Your feedback will be used to help your TA improve this semester.
A3 due tonight

262 groups submitted
~215 to go

max: 24 hours used    average: 4.2 hours    mean: 4.0

Histogram: [inclusive:exclusive)
(e.g. 63 people took at least 2 but less than 3 hours)

[0:1): 3    [07:08): 20

We wrote a Java program to extract the times and produce this table. Later, we will share it with you.

These assignments are not meant to kill you! If you are taking an inordinate amount of time, seek help!
Readings and homework

Textbook, Chapter 23, 24

Homework: A thought problem (draw pictures!) In A1, you had a binary tree! Given two such trees, how would you determine whether they had a person in common?
Tree overview

**Tree**: recursive data structure:
A tree is a set of nodes that is either

- empty OR
- a node with a value and a list of trees (called its children)

**Binary tree**: tree in which each node has two children:
a left child and a right child
Binary trees were in A1!

You have seen a binary tree in A1.

A PhD object phd has one or two advisors. Here is an intellectual ancestral tree!

```
        phd
       /   \
      ad1   ad2
     /   /  \
    ad1 ad2 ad1
```
Tree terminology

$M$: root of this tree
$G$: root of the left subtree of $M$
$B, H, J, N, S$: leaves (their set of children is empty)
$N$: left child of $P$; $S$: right child
$P$: parent of $N$
$M$ and $G$: ancestors of $D$
$P, N, S$: descendents of $W$
$J$ is at depth 2 (i.e. length of path from root = no. of edges)
$W$ is at height 2 (i.e. length of longest path to a leaf)
A collection of several trees is called a ...?
Tree terminology

*Two views of G.*

G is a node of a tree.

G is the root of a (sub)tree, that is, we can talk about tree G or the tree rooted at G.

*Same idea:*

X is a node of a linked list.

Linked list X (Linked list whose first node is X)
Class for binary tree node

class TreeNode<T> {
    private T datum;
    private TreeNode<T> left, right;

    /** Constructor: one node tree with datum x */
    public TreeNode (T x) { datum= x; }

    /** Constr: Tree with root value x, left tree l, right tree r */
    public TreeNode (T x, TreeNode<T> l, TreeNode<T> r) {
        datum= x; left= l; right= r;
    }
}

more methods: getDatum, setDatum, getLeft, setLeft, etc.
Binary versus general tree

In a binary tree, each node has exactly two pointers: to the left subtree and to the right subtree:

- One or both could be null, meaning the subtree is empty (remember, a tree is a set of nodes)

In a general tree, a node can have any number of child nodes

- Very useful in some situations ...
- ... one of which may be in an assignment!
Class for general tree nodes

```java
class GTreeNode {
    1. private Object datum;
    2. private GTreeCell[] siblings;
    3. appropriate getters/setters
}
```

Parent contains an array of its children
class GTreeNode {
1. private Object datum;
2. private GTreeCell left;
3. private GTreeCell sibling;
4. appropriate getters/setters
}

- Parent points only to its leftmost child
- Each child has pointer to its next sibling.
Use of trees: Represent expressions

In textual representation:
Parentheses show hierarchical structure

In tree representation:
Hierarchy is explicit in the structure of the tree

We’ll talk more about expression and trees on Thursday
Recursion on trees

Trees are defined recursively. So recursive methods can be written to process trees in an obvious way.

Base case

- empty tree (null)
- leaf

Recursive case

- solve problem on left / right subtrees
- put solutions together to get solution for full tree
Searching a binary tree. The tree is a parameter

```java
/** Return true iff \(x\) is the datum in a node of tree \(t\)*/
public static boolean treeSearch(Object x, TreeNode t) {
    if (t == null) return false;
    if (t.datum.equals(x)) return true;
    return treeSearch(x, t.left) || treeSearch(x, t.right);
}
```

• Analog of linear search in lists:
  given tree and an object, find out if
  object is stored in tree
• Easy to write recursively, harder to write iteratively
public class TN{
    private TN lft;  private TN rgt;
    /** Return size of this tree */
    public int size() {
        return 1 + (lft == null ? 0 : lft.size()) +
                (rgt == null ? 0 : rgt.size());
    }
    /** Return size of tree t –note: t could be null (empty tree)*/
    public static int size(TN t) {
        if (t == null) return 0;
        return 1 + size(t.lft) + size(t.rgt);
    }
}
**Binary Search Tree (BST)**

**BST:** All left descendents of each node have a smaller value than that node’s value. All right descendents of each node have a larger value than that node’s value.

```java
/** Return true iff x is the datum in a node of tree t. 
   * Precondition: t is a BST */
public static boolean treeSearch (Object x, TreeNode t) {
    if (t==null) return false;
    if (t.datum.equals(x)) return true;
    if (t.datum.compareTo(x) > 0)
        return treeSearch(x, t.left);
    return treeSearch(x, t.right);
}
```
Building a BST

- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree
- This can be done using either recursion or iteration

- Example
  - Tree uses *alphabetical order*
  - Months appear for insertion in *calendar order*
What can go wrong?

A BST makes searches very fast, unless...

- Nodes are inserted in increasing order
- In this case, we’re basically building a linked list (with some extra wasted space for the left fields, which aren’t being used)

BST works great if data arrives in random order
Printing contents of BST

Because of ordering rules for a BST, it’s easy to print the items in alphabetical order

- Recursively print left subtree
- Print the node
- Recursively print right subtree

```java
/** Print BST t in alpha order */
private static void print(TreeNode t) {
    if (t == null) return;
    print(t.lchild);
    System.out.print(t.datum);
    print(t.rchild);
}
```
Tree traversals

“Walking” over whole tree is a **tree traversal**

- Done often enough that there are standard names

Previous example:
- **inorder traversal**
  - Process left subtree
  - Process root
  - Process right subtree

Note: Can do other processing besides printing

**Other standard kinds of traversals**

- **preorder traversal**
  - Process root
  - Process left subtree
  - Process right subtree

- **postorder traversal**
  - Process left subtree
  - Process right subtree
  - Process root

- **level-order traversal**
  - Not recursive uses a queue. We discuss later
Some useful methods

/** Return true iff node t is a leaf */
public static boolean isLeaf(TreeNode t) {
    return t != null && t.left == null && t.right == null;
}

/** Return height of node t (postorder traversal) */
public static int height(TreeNode t) {
    if (t == null) return -1; // empty tree
    if (isLeaf(t)) return 0;
    return 1 + Math.max(height(t.left), height(t.right));
}

/** Return number of nodes in t (postorder traversal) */
public static int nNodes(TreeNode t) {
    if (t == null) return 0;
    return 1 + nNodes(t.left) + nNodes(t.right);
}
Useful facts about binary trees

Max # of nodes at depth $d$: $2^d$

If height of tree is $h$
- min # of nodes: $h + 1$
- max # of nodes in tree:
  $2^0 + \ldots + 2^h = 2^{h+1} - 1$

Complete binary tree
- All levels of tree down to a certain depth are completely filled
Assignment A4: Collision detection

Detect whether two shapes share a common pixel (or block)

A shape consists of LOTS of blocks (like pixels). If each shape has 1,000 blocks, brute force checking for a common block takes worst-case time proportional to $1,000^2 = 1,000,000$. 

(attempt to show the blocks)
If their bounding boxes don’t overlap, the shapes can’t have a block in common.

Each Shape object has a field that contains its bounding box. Can check whether two bounding boxes overlap in constant time.

But, if bounding boxes overlap, still have to look for a block that is common to both, and there may not be one! Need data structure to make that task efficient.
Assignment A4: Idea: Use a Binary search tree!

Below, BT stands for BlockTree

This object is a node of a binary search tree, and it and its subtrees describe a bunch of blocks (pixels)
Assignment A4: Leaf of the binary search tree

Below, BT stands for BlockTree

<table>
<thead>
<tr>
<th>BT@1</th>
<th>BT</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>ptr to the block that this leaf describes</td>
</tr>
</tbody>
</table>

A leaf contains one block

For this shape, might have 1,000 leaves!
White space is not part of image
Assignment A4: internal node of the BST

Below, BT stands for BlockTree

Doesn’t contain a block!

Behind the node, there are two child nodes:
- left child
- right child

Two fields of BT:
- left: ptr to left subtree
- right: ptr to right subtree

All blocks whose center is \( \leq \) midpt go in left subtree.
All blocks whose center is \( > \) midpt go in right subtree.

Bounding box for all blocks in left or right subtree is longer than it is tall.
Assignment A4: internal node of the BST

Below, BT stands for BlockTree

Node

Left child
- Blocks with horiz center \( \leq \) midpt

Right child
- Blocks with horiz center \( > \) midpt

Left child
- Blocks with vertical center \( \leq \) midpt
- Bounding box for blocks in tree

Right child
- Blocks with vertical center \( > \) midpt

Midpt
- Bounding box for blocks in tree
Assignment A4: the Block Tree: a BST

You will write the constructor of BlockTree, which constructs the BST. It will be recursive-like.

You will write a method that uses the BlockTree ---i.e. the BST--- to determine whether two shapes have a block in common. That method will be recursive.
Assignment A4: Building a bounding box

```java
public static BoundingBox findBBox(Iterator<Block> iter)
```

This method is supposed to construct and return a BoundingBox (which represents a rectangle) for the blocks given by iter.

WHAT THE HECK IS AN ITERATOR?

We posted an explanation in Piazza A4 FAQ note @472
Class BoundingBox

Class BoundingBox contains methods whose bodies you must write. This is one of the first things to work on.

If you don’t implement the methods correctly, nothing will work!

Think about how you can use a Junit testing class to test these.
Advice

This assignment is fun and illuminating. You will learn a lot from it.

It is harder than A3! You need time to ponder, to ask questions, to get answers. You have to start early!

Start reading now (if you haven’t done so already). Get BoundingBox finished and tested soon.

Make use of the Piazza, especially A4 FAQ note @472.
In some applications, it is useful to have trees in which nodes can reference their parents.

Analog of doubly-linked lists.
Things to think about

What if we want to delete data from a BST?

A BST works great as long as it’s **balanced**

How can we keep it balanced? *This turns out to be hard enough to motivate us to create other kinds of trees*
Tree Summary

- A tree is a recursive data structure
  - Each node has 0 or more successors (children)
  - Each node except the root has at exactly one predecessor (parent)
  - All nodes are reachable from the root
  - A node with no children (or empty children) is called a leaf

- Special case: binary tree
  - Binary tree nodes have a left and a right child
  - Either or both children can be empty (null)

- Trees are useful in many situations, including exposing the recursive structure of natural language and computer programs
Suffix tree (we won’t test on these)
A suffix tree for a string $s$ is a tree such that

- each edge has a unique label, which is a nonnull substring of $s$
- two edges leaving the same node have labels beginning with different characters
- catenation of labels along any path from root to a leaf gives a suffix of $s$
- all suffixes are represented by some path
- the leaf of the path is labeled with the index of the first character of the suffix in $s$

Suffix trees can be constructed in linear time
Suffix trees (we won’t test on these)

- Useful in string matching algorithms (e.g. longest common substring of 2 strings)
- Most algorithms linear time
- Used in genomics (human genome is ~4GB)
Huffman trees (we won’t test on these)

Fixed length encoding
197*2 + 63*2 + 40*2 + 26*2 = 652

Huffman encoding
197*1 + 63*2 + 40*3 + 26*3 = 521
## Huffman compression of “Ulysses”

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
<th>Huffman Code</th>
<th>Length</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>242125</td>
<td>00100000</td>
<td>3</td>
<td>110</td>
</tr>
<tr>
<td>'e'</td>
<td>139496</td>
<td>011000101</td>
<td>3</td>
<td>000</td>
</tr>
<tr>
<td>'t'</td>
<td>95660</td>
<td>01110100</td>
<td>4</td>
<td>1010</td>
</tr>
<tr>
<td>'a'</td>
<td>89651</td>
<td>011000001</td>
<td>4</td>
<td>1000</td>
</tr>
<tr>
<td>'o'</td>
<td>88884</td>
<td>01101111</td>
<td>4</td>
<td>0111</td>
</tr>
<tr>
<td>'n'</td>
<td>78465</td>
<td>01101110</td>
<td>4</td>
<td>0101</td>
</tr>
<tr>
<td>'i'</td>
<td>76505</td>
<td>01101001</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>'s'</td>
<td>73186</td>
<td>01110011</td>
<td>4</td>
<td>0011</td>
</tr>
<tr>
<td>'h'</td>
<td>68625</td>
<td>01101000</td>
<td>5</td>
<td>11111</td>
</tr>
<tr>
<td>'r'</td>
<td>68320</td>
<td>01110010</td>
<td>5</td>
<td>11110</td>
</tr>
<tr>
<td>'l'</td>
<td>52657</td>
<td>01101100</td>
<td>5</td>
<td>10111</td>
</tr>
<tr>
<td>'u'</td>
<td>32942</td>
<td>01110101</td>
<td>6</td>
<td>111011</td>
</tr>
<tr>
<td>'g'</td>
<td>26201</td>
<td>01100111</td>
<td>6</td>
<td>101101</td>
</tr>
<tr>
<td>'f'</td>
<td>25248</td>
<td>01100110</td>
<td>6</td>
<td>101100</td>
</tr>
<tr>
<td>'p'</td>
<td>21361</td>
<td>00101110</td>
<td>6</td>
<td>011010</td>
</tr>
<tr>
<td>'p'</td>
<td>20661</td>
<td>01110000</td>
<td>6</td>
<td>011001</td>
</tr>
</tbody>
</table>
Huffman compression of “Ulysses”

... 

- '7' 68 00110111 15 111010101001111
- '/\' 58 00101111 15 111010101001110
- 'X' 19 01011000 16 0110000000100011
- '&\' 3 00100110 18 011000000010001010
- '%' 3 00100101 19 0110000000100010111
- '+' 2 00101011 19 01100000001000101110

original size 11904320
compressed size 6822151
42.7% compression
BSP Trees (we won’t test on these)

- BSP = Binary Space Partition (not related to BST!)
- Used to render 3D images composed of polygons
- Each node $n$ has one polygon $p$ as data
- Left subtree of $n$ contains all polygons on one side of $p$
- Right subtree of $n$ contains all polygons on the other side of $p$
- Order of traversal determines occlusion (hiding)!