Quote of the Week: “As I walked out the door toward the gate that would lead to my freedom, I knew if I didn't leave my bitterness and hatred behind, I'd still be in prison.”
Project 2 group evaluations

- They’re due Thursday, with Monday / Tuesday lab
- You will get a 0 on the project unless you complete this
- Please be honest and fair. These may affect your group members’ scores
Midterm 2 on Friday

- Same time, place
  - 7 - 9 pm
  - Sections 101 - 103: 144 Dwinelle
  - Sections 104 - 109: 155 Dwinelle
- Cheat sheet: one 8.5 x 11 sheet, two sides
Project 3 released on Friday

- Also a 3 - 4 group project
- Project is about speed (actual time, not theoretical)
Project 3

Write a program to solve puzzles like this:

Credit: http://magicpuzzles.org/
Motivated by a simple problem: How to figure out the steps required to solve a puzzle like this?

A tray with blocks you can slide around

Goal: move green block to bottom-right corner
Visualizing the problem

One possible move

Another possible move
It looks kinda like a tree
But wait...!

The same thing!
But wait...!
Also

Any time we make a move, we could always undo it
It looks kinda like a tree

- But it violates the rules of trees
  - No edges point back up the tree
  - No node is descended from two nodes
So it’s not a tree

- We call it a graph
Graphs

- A graph is a collection of nodes that can be connected in any which way.
Graph Traversals
To solve this problem, we must find a path through the graph from our initial tray to our goal tray.

Essentially, this boils down to iterating through our graph, starting from the initial tray, until we come across the goal tray.
Graph traversal

- How do we iterate over the nodes of a graph?
  - A graph isn’t much different from a tree, so let’s try tree traversal!
Traversing a graph like a tree

_kinda works...?

```java
Stack<Tray> fringe = new Stack<>();
fringe.push(initialTray);
while (!fringe.isEmpty()) {
    Tray currentTray = fringe.pop();
    // do stuff
    for (Tray t : currentTray.nextTrays()) {
        fringe.push(t);
    }
}
```
I’ve labeled the boards with numbers, for convenience
Let’s see the traversal in action
Step 1: Create a fringe

The fringe (a Stack)
Step 2: Put initial tray in fringe
Step 3: Take something from the fringe, make it “current”
Check if it’s the goal (it’s not), so add adjacents to fringe.
Take something, check if goal. It’s not.
Add adjacent to fringe

current

0
3
4
2

1

0
2

3
4
5
Take something, make it current
Check if goal. It’s not.

1. current
2. 0
3. 1
4. 2
5. 3
6. 4
7. 5

Check if goal. It’s not.
Add adjacent to fringe

Current:

0 → 1
0 → 2
0 → 3
0 → 4
0 → 5

1 → 0
1 → 2
1 → 3
1 → 4
1 → 5

2 → 0
2 → 1
2 → 3
2 → 4
2 → 5

3 → 1
3 → 2
3 → 4
3 → 5

4 → 1
4 → 2
4 → 3
4 → 5

5 → 2
5 → 3
5 → 4
Wait a minute!!
Traversing a graph like a tree

- We end up going in circles!

- What went wrong?
  - The rules of trees ensure that, starting from root, there is only one possible path to each node
  - But for graphs, we can keep finding the same node over-and-over again
Solution?

Recall the *fringe* is meant to be a set of nodes we’ve temporarily passed by and intend to return to later.

So, let’s not put something in the fringe if we’ve already visited it.
Graph traversal

```java
Stack<Tray> fringe = new Stack<>();
fringe.push(initialTray);
while (!fringe.isEmpty()) {
    Tray currentTray = fringe.pop();
    // do stuff
    for (Tray t : currentTray.nextTrays()) {
        if (!alreadyVisited(t)) {
            fringe.push(t);
        }
    }
}
```
What is this really?

Stack<Tray> fringe = new Stack<>();
Set<Tray> visited = new HashSet<>();
fringe.push(initialTray);
while (!fringe.isEmpty()) {
    Tray currentTray = fringe.pop();
    // do stuff
    visited.add(currentTray);
    for (Tray t : currentTray.nextTrays()) {
        if (!visited.contains(t)) {
            fringe.push(t);
        }
    }
}
Graph traversal — the full story

- The same as tree traversal
- Except we make sure to not repeat ourselves
Quiz part 1: path finding

- I claimed that finding the goal board during the traversal is essentially the same problem as figuring out the path to the goal board.

- Is it really?
public class GraphNode {
    String myItem;
    List<GraphNode> myAdjacents;
    /**
     * Prints out the items of the nodes you have
     * to follow from this node until you find a
     * you find a node with target item
     */
    public void printPathTo(String target) {
        // TODO your code here
    }
}
public void printPathTo(String target) {
    Set<GraphNode> visited = new HashSet<>();
    Stack<GraphNode> fringe = new Stack<>();
    Map<String, String> steps = new HashMap<>();
    fringe.push(this);
    while (!fringe.isEmpty()) {
        GraphNode currentNode = fringe.pop();
        if (currentNode.myItem.equals(target)) {
            break;
        }
        visited.add(currentNode);
        for (GraphNode g : currentNode.myAdjacents) {
            if (!visited.contains(g)) {
                steps.put(g.myItem, currentNode.myItem);
                fringe.push(g);
            }
        }
    }
    Stack<String> reversePath = new Stack<>();
    String currentStep = target;
    while (currentStep != null) {
        String previousStep = steps.get(currentStep);
        if (previousStep != null) {
            reversePath.push(previousStep);
        }
        currentStep = previousStep;
    }
    while (!reversePath.isEmpty()) {
        System.out.println(reversePath.pop());
    }
}
Our problem: an implicit graph

Here again is our traversal code

```java
Stack<Tray> fringe = new Stack<>();
Set<Tray> visited = new HashSet<>();
fringe.push(initialTray);
while (!fringe.isEmpty()) {
    Tray currentTray = fringe.pop();
    // do stuff
    visited.add(currentTray);
    for (Tray t : currentTray.nextTrays()) {
        if (!visited.contains(t)) {
            fringe.push(t);
        }
    }
}
```

Notice we don’t have one object that stores the entire graph of possible tray configurations
Our problem: an implicit graph

- Instead, if each tray just knows about the trays that can follow it, then we implicitly have a graph.

- We never actually have a variable of type `Graph<Tray>` that stores all the trays.
Lucky us, because…

- ...the graph of possible trays is usually **far too big** for us to store in memory at once
- Good thing we only have to look at one local part at a time
- For completeness, though: what if we wanted to store the whole explicit graph?
Digression: explicit graph representations
For **linked lists**, we had a `LinkedList` class, that stored a reference to the first node, from which could be found all the other nodes.
For trees, we had a Tree class, that stored a reference to the root, from which could be found all the other nodes.
For **graphs**, we could have a **Graph** class, that stores a reference to ???, from which could be found all the other nodes.
Graph data structure

- What would the Graph object store a reference to?
- Because a graph can have any structure, there isn’t an obvious “first” or “starting” node in general
I guess we just have to store all of them

- The Graph object will store an array, one spot for each node
Warning: strange assumption!!
But first, an assumption

- Before discussing the graph representations, I will first introduce an assumption.
- The graph does not store arbitrary objects (like Strings, Trays, etc.). Instead, it can only store the integers 0 ... N (if there are N+1 vertices).
- Wha…? Why?
- Will be justified later!
Example graph we want to represent in Java

The theoretical (conceptual) graph
The graph data structure

- The graph object will store an array, one spot for each vertex

Indices of the array match the vertices
The graph data structure

- The only thing that matters about a graph is which vertices have edges between them
The graph data structure

- So each array will contain a list of vertices that are adjacent
The graph data structure

- So each array will contain a list of vertices that are adjacent

The connections between these nodes are NOT the edges in our graph!

These are just a list of adjacent nodes
The graph data structure

This is called the adjacency list associated with node 0. It tells you the vertices adjacent to 0.
A graph is nothing more than a set of vertices and connections between them.

All the information is here.
Story of a beautiful partnership: the sequel

- A graph can be represented as an array of linked lists!
Why linked lists?

- Linked lists make it easy to append new items to the end, make it easy to **add edges**
- But what if we want to check if an edge exists? e.g. check if $0 \rightarrow 2$?
  - Must iterate through list at 0 until we find 2… not cool!
Alternative: array of arrays

- Store an array rather than a list, tells you whether it is adjacent
Alternative: array of arrays

- Commonly drawn like this:

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
```

Graph:
```
myVertices
```

0 → 1 → 2 → 3
Two graph representations

Array of adjacency lists

Adjacency matrix
Pros/cons of two graph representations

- The matrix is faster to check if there exists an edge (just index into the array)
- But, the matrix but a waste of space if there are lots of false values
Graphs of other things

- So, we can now store a graph of integers 0 … \( N \)
- What if we want to store a graph of something else? Like Strings, or tray objects?
- Easy! We’ll have a map from object to number, and back
Say we want to represent this conceptual graph.
Here's the picture

<table>
<thead>
<tr>
<th>Object</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;wug&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;wugs&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;wales&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;whales&quot;</td>
<td>3</td>
</tr>
</tbody>
</table>

Map from object to index

Adjacency structure

The conceptual graph
Graphs of other things

- Consist of **two parts:**
  - The **indexer**, which associates each item with an index
  - And the **adjacency structure**, which keeps track of all the connections
What’s this indexer thing?

- The reason we separate them out is so that graphs of all different kinds of things can have essentially the same core structure.
- This means someone could write 1 graph processing function that could work on all sorts of graphs.
Digression complete!
BREAK
Back to our problem…
Sliding block puzzle

- Yay we learned about explicit graphs~
- But these were irrelevant for our block sliding problem, which is an implicit graph
- We decided to solve this problem using traversals. Are we done?
- Not yet! We want a faster solution
A new idea

- When we have a choice which way to go…
- Shouldn’t we choose the best option?
Introducing **heuristics (your new best friend)**

- **Idea**: Choose the one that moves the block closer to the goal position. Quantify this with a number.

- This number is called a **heuristic**, just a guess.

- It might be wrong. Sometimes, moving the block toward the goal is the wrong thing to do. But it’s a reasonable guess.
Change our code!

```java
Stack<Tray> fringe = new Stack<>();
Set<Tray> visited = new HashSet<>();
fringe.push(initialTray);
while (!fringe.isEmpty()) {
    Tray currentTray = fringe.getBestItem();
    // do stuff
    visited.add(currentTray);
    for (Tray t : currentTray.nextTrays()) {
        if (!visited.contains(t)) {
            fringe.push(t);
        }
    }
}
```
Introducing the priority queue ADT

- What is this `getBestItem` method?
- This is the method of a **priority queue**, not a stack!
Priority queue

- Supports operations
  - `void add(Comparable c)` (in Java, this is `offer`)
  - `Comparable extractMin()` (in Java, this is `poll`)

Min or max priority queue?

- Would we have `extractMin` or `extractMax`?
- Tradition is `extractMin`, but it’s very easy to change to `extractMax`:
  - Just flip everything I’m about to say for the remainder of lecture
How to use priority queue

- import java.util.PriorityQueue;
- PriorityQueue<Tray> q = new PriorityQueue<Tray>();
- q.offer(new Tray());
- Tray t = q.poll();
Let’s make a priority queue!

- How?
- First idea: Use a sorted linked list of items
Priority queue with a sorted list

- `add(Comparable c)` find the correct spot for the item in the sorted list, and put it there
- `Comparable extractMin()` remove and return the first item of the list
Runtimes?

- `add(Comparable c)` find the correct spot for the item in the sorted list, and put it there: $O(N)$ time in the worst case, where there are $N$ items in the queue.

- `Comparable extractMin()` remove and return the first item of the list: $O(1)$.
Priority queue with sorted list

Surely we can do better?
Let’s make a priority queue!

- Second idea: Use a **binary search tree**
  (already kinda sorted)
Priority queue with a BST

- `add(Comparable c)` add to the BST like normal
- `Comparable extractMin()` go down to the far left, and return/remove the item there
**Runtimes?**

- **add(Comparable c)** add to the BST like normal: \(O(\log N)\) time, with \(N\) items in queue

- **Comparable extractMin()** go down to the far left, and return/remove the item there: \(O(\log N)\) time
The BST priority queue

- Both operations run in $\log$ time
- This is basically as good as it gets
- But there’s still something a little unsatisfying
The beefy BST

- The BST can find any item in log time
- But the priority queue only needs to find the best item quickly
  - The BST is *more powerful* than we need
The beefy BST

- The BST is a complicated structure (did you have fun coding AVL tree rotations?)
- Since it is complicated and more powerful than we need, we might wonder if there's a simpler data structure that does the job just as well
Simpler is better

- Simpler data structures can be faster than complicated ones, due to constant factors, even if the asymptotic runtimes are the same.
- This turns out to be the case for priority queue.
A simpler idea for a tree

- Why not just store the min item at the top of the tree? That’s the easiest place to look
Introducing the binary heap

- The binary (min) heap is a tree structure that stores items with smallest at the top, and bigger below.
Heap vs. BST

Heap
- small items
- large items

BST
- medium items
- small items
- large items
Heap invariants

- Specifically, a heap is a tree with an extra invariant:
  - Every child is bigger than (or the same as) its parent

- Notice: left-right order in heap does not matter at all! This makes it simpler than BST
Heap invariants

- Remember a BST also had an almost balanced invariant
- Heaps will also have a balance invariant, but it will be the maximally balanced property
Maximally balanced — why now?

- For BSTs, maintaining maximal balance requires a lot of work — even maintaining almost balance required some wacky rotations

- Because the heap is simpler overall, it offsets the extra work required to maintain maximal balance
Recall: maximal balance

- Was equivalent to the condition that the array tree has no holes in it
Recall: maximal balance

- We agreed the array tree would be more memory efficient if the tree was maximally balanced.
- So we’ll implement the heap with an array, usually!
Heap properties

- To sum up, a heap has additional two properties over a normal tree:
  - **The content property:** Each child is bigger than the parent
  - **The structure property:** Tree is maximally balanced
Heap operations

- Heap needs two operations
  - `add(Comparable c)`
  - `Comparable extractMin()`
Adding to a heap

- When we add, we have to make sure to maintain the properties
- The structural property is the stricter of the two policies, so let’s start with it
Adding to a heap

- There is only one possible shape for a heap with N nodes

The heap with 5 nodes always looks like this, regardless of content
Adding to a heap

- There is only one possible shape for a heap with N nodes

Not possible! Not maximally balanced!
Adding to a heap

- Hence, the shape of a heap with \( N + 1 \) nodes is completely predictable.
- A new node always appears in the next open spot (gets appended to the end of the array).
Adding to a heap

Heap with 5
Adding to a heap
Adding to a heap

- So, when we add a new item to the heap, we have no choice except to put it in the bottom right location
Adding to a heap

- Say we add 0 to this heap
We have no choice…!
Okay, but

- Now the content property is messed up
Adding to a heap

- We want to fix the content property **without** messing with the structure of the tree.
- We can do this by *swapping* the values of nodes until we’re good.
Swap!
Okay, but still not good enough...
Swap!!
Now we’re good
Bubbling up

- After appending to the end, swap the value of the node up the tree until the content property is satisfied (this is called bubbling up)

- This is the full story with add

- (Easier than an AVL tree, right?)
void add(Comparable c): append an item to the end, then swap it up the tree until okay: $O(\log N)$, potentially have to swap all the way to the top

Comparable extractMin()
Extract min

- It’s easy to figure out the value of the min item in the heap
- It’s just the root (position 1 in the array)
- But how do we take it out?
The shape of a heap with $N - 1$ nodes is completely predictable.
Adding to a heap

Heap with 6
Adding to a heap

Heap with 5
Extract min

- We have no choice except to remove the bottom right element
- But that’s not the one we want to remove! We want to remove the top!
- So: First swap the top element with the bottom, then remove
Let's remove min
First swap
Then take off
Great, but

- Now the content property is messed up
- (not again!)
We have to fix the content property without messing with the structure.

Back to swapping!

This time, we’ll swap the new top down the tree.
Which way do we swap the 3 down?

Swap so the smallest thing ends up top
Swap!!!

Diagram:

1 -> 3 -> 1

11 -> 15 -> 4 -> 2
Now we’re good
Bubbling down

- After swapping the top with the bottom, and taking off the bottom, **bubble down** the new top until it hits the correct spot
- This is the full story with extract min
- (Easier than an BST remove, right?)
Heap operations

- **void add(Comparable c)**: append an item to the end, then swap it up the tree until okay: \(O(\log N)\), potentially have to swap all the way to the top

- **Comparable extractMin()** swap top and bottom, take off bottom, bubble down new top: \(O(\log N)\), potentially have to swap all the way down
Heap vs. BST

- Heap ultimately has the same asymptotic runtimes as BST for representing a priority queue.
- But a heap is implemented with an array, which is far more memory efficient than nodes with tons of pointers.
- And the most complicated operations in a heap are swapping values at array indices, which is super fast.
Heap vs. BST: fun facts

**Heap**
- small items
- large items

**BST**
- medium items
- small items
- large items
Heap vs. BST: fun facts

- Heap is **maximally balanced**
- BST is usually **almost balanced** (well, AVL tree is)
Heap vs. BST: fun facts

- Heap is usually implemented as an **array** tree
- BST is usually implemented with **nodes**
Heap vs. BST: fun facts

- Both can get/remove the min element in log time
- Both can add new items in log time
- Heap is slightly faster for priority queue
Heap vs. BST: fun facts

- BST can find any item in the tree in log time
- Heap can find any item in the tree in...?
Heap contains

- How do we write this method for a heap?
- `boolean contains(int item)`
- Uh oh.
Does the heap contain 4?
- Does the heap contain 4?
- Good question.
Does the heap contain 4?

Good question.

Might as well start by iterating
Does the heap contain 4?
Does the heap contain 4?

Nope
Does the heap contain 4?

Now what?
If this were a BST, we could tell if we should go look down left or right

But there’s no hint at the direction in heap
This would also have been a valid heap
In general, there’s no way to know *where* an item is in a heap.

There are cute optimizations you can make, but asymptotically it’s not better than just searching through every item in the heap.
Heap vs. BST: fun facts

- BST: Can check if contains any item in log time
- Heap: Can check if contains any item in linear time (how sad!) — Heap is really only good as a priority queue
Quiz time!

This is more of a midterm review question, and doesn’t necessarily have to do with anything we just learned.
Quiz — Median maintainer

- Invent a data structure that supports the following operations:
  - `void add(int item)`
  - `int getMedian()`
- Your goal is to have as fast asymptotic runtimes as possible
Quiz — Median maintainer

Assume you have the following structures at your disposal to help:

- LinkedList
- ArrayList
- HashSet
- HashMap
- BST
- Stack
- Queue
- PriorityQueue
- Graph
Hint 1: You should be able to have `getMedian` at constant time, and add at log time
Quiz — Median maintainer

- Hint 2: Half the items are smaller than the median, and half of them are larger than the median

- Maybe have two separate collections: the items that are smaller, and the items that are larger…
Quiz — Median maintainer

Solution: Maintain two priority queues, one of which is a max priority queue, and the other of which is a min priority queue.
Main idea

Items less than median (max priority queue)

![2](2)
![1](1)
![0](0)

The median

![4](4)

Items greater than median (min priority queue)

![6](6)
![7](7)
![9](9)
If we add items here, the new median will be the smallest thing in this collection (so we want min pq)
Main idea

If we add items here, the new median will be the largest thing in this collection (so we want max pq)