CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Midterm Review
Midterm Procedure

- Exam is during normal class time. Same time, same place.
- Seating is assigned randomly
  - Wait outside the room until instructed to enter
  - Immediately place all bags/electronics at the front of the room
- At your seat you should have:
  - Writing utensil
  - UB ID card
  - One 8.5x11 cheatsheet (front and back) if desired
  - Summation/Log rules will be provided
## Overview

<table>
<thead>
<tr>
<th>Analysis Tools/Techniques</th>
<th>ADTs</th>
<th>Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymptotic Analysis, (Unqualified) Runtime Bounds</td>
<td>Seq</td>
<td>Array</td>
</tr>
<tr>
<td>Amortized Runtime</td>
<td>Seq, Buffer</td>
<td>ArrayBuffer</td>
</tr>
<tr>
<td>Recursive analysis, divide and conquer, Average/Expected Runtime</td>
<td>Seq</td>
<td>Linked Lists</td>
</tr>
<tr>
<td></td>
<td>Stack, Queue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Graphs</td>
<td>EdgeList, AdjacencyList, Adjacency Matrix</td>
</tr>
</tbody>
</table>
Analysis Tools and Techniques
Limit Tests

**Case 1:** \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \)  
(f grows faster; g is better)

**Case 2:** \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \)  
(g grows faster; f is better)

**Case 3:** \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{some constant} \)  
(f and g “behave” the same)
Recap of Runtime Complexity

**Big-Θ — Tight Bound**
- Growth functions are in the same complexity class
- If \( f(n) \in \Theta(g(n)) \) then an algorithm taking \( f(n) \) steps is "exactly" as fast as one that takes \( g(n) \) steps.

**Big-O — Upper Bound**
- Growth functions in the same or smaller complexity class.
- If \( f(n) \in O(g(n)) \), then an algorithm that takes \( f(n) \) steps is at least as fast as one taking \( g(n) \) (but it may be even faster).

**Big-Ω — Lower Bound**
- Growth functions in the same or bigger complexity class
- If \( f(n) \in \Omega(g(n)) \), then an algorithm that takes \( f(n) \) steps is at least as slow as one that takes \( g(n) \) steps (but it may be even slower)
Common Runtimes (in order of complexity)

Constant Time: $\Theta(1)$

Logarithmic Time: $\Theta(\log(n))$

Linear Time: $\Theta(n)$

Quadratic Time: $\Theta(n^2)$

Polynomial Time: $\Theta(n^k)$ for some $k > 0$

Exponential Time: $\Theta(c^n)$ (for some $c \geq 1$)
Formal Definitions

\( f(n) \in O(g(n)) \) iff exists some constants \( c, n_0 \) s.t.

\[ f(n) \leq c \times g(n) \] for all \( n > n_0 \)

\( f(n) \in \Omega(g(n)) \) iff exists some constants \( c, n_0 \) s.t.

\[ f(n) \geq c \times g(n) \] for all \( n > n_0 \)

\( f(n) \in \Theta(g(n)) \) iff \( f(n) \in O(g(n)) \) and \( f(n) \in \Omega(g(n)) \)
Amortized Runtime

If $n$ calls to a function take $O(T(n))$...

We say the **Amortized Runtime** is $O(T(n) / n)$

The **amortized runtime** of `append` on an **ArrayBuffer** is: $O(n/n) = O(1)$

The **unqualified runtime** of `append` on an **ArrayBuffer** is: $O(n)$
What guarantees do you get?

If \( f(n) \) is a Tight Bound
The algorithm always runs in \( cf(n) \) steps

If \( f(n) \) is a Worst-Case Bound
The algorithm always runs in at most \( cf(n) \)

If \( f(n) \) is an Amortized Worst-Case Bound
\( n \) invocations of the algorithm always run in \( cnf(n) \) steps

If \( f(n) \) is an Average Bound
...we don't have any guarantees
Inductive Proofs

Solve for $T(n)$

**Approach:**

1. Generate a hypothesis
2. Prove your hypothesis for the base case
3. Prove the hypothesis for the recursive case *inductively*
Abstract Data Types (ADTs)

- The specification of what a data structure can do

Diagram:
- Read everything
- Read "nth" element
- Update "nth" element
Abstract Data Types (ADTs)

- The specification of what a data structure can do

Usage is governed by **what** we can do, not **how** it is done

What's in the box? ...we don't know, and in some sense...we don't care
# Abstract Data Type vs Data Structure

<table>
<thead>
<tr>
<th>ADT</th>
<th>Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>The interface to a data structure</td>
<td>The implementation of one (or more) ADTs</td>
</tr>
<tr>
<td>Defines <strong>what</strong> the data structure can do</td>
<td>Defines <strong>how</strong> the different tasks are carried out</td>
</tr>
<tr>
<td>Many data structures can implement the same ADT</td>
<td>Different data structures will excel at different tasks</td>
</tr>
</tbody>
</table>
Abstract Data Type vs Data Structure

**ADT**

*The interface to a data structure*

*Defines what the data structure can do*

*Many data structures can implement the same ADT*

---

**Data Structure**

*The implementation of one (or more) ADTs*

*Defines how the different tasks are carried out*

*Different data structures will excel at different tasks*

---

Think about the Linked List we are implementing for PA2.

The internal structure and the mental model of our sequence are very different.
## Seq Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>apply(i)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>update(i, val)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>insert(i, val)</td>
<td>Θ(n)</td>
<td>O(n)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>remove(i, val)</td>
<td>Θ(n)</td>
<td>Θ(n-i), O(n)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>append(i)</td>
<td>Θ(n)</td>
<td>O(n), Amortized Θ(1)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>
**Queues vs Stacks (ADTs)**

*Queue* First in, First Out (FIFO)

*Stacks* Last in, First Out (LIFO / FILO)
Recap

Stacks: Last In First Out (LIFO)
- Push (put item on top of the stack) \( \Theta(1) \) (or amortized \( O(1) \))
- Pop (take item off top of stack) \( \Theta(1) \)
- Top (peek at top of stack) \( \Theta(1) \)

Queues: First in First Out (FIFO)
- Enqueue (put item on the end of the queue) \( \Theta(1) \) (or amortized \( O(1) \))
- Dequeue (take item off the front of the queue) \( \Theta(1) \)
- Head (peek at the item in the front of the queue) \( \Theta(1) \)
A (Directed) Graph ADT

Two type parameters (Graph[V, E])
- V: The vertex label type
- E: The edge label type

Vertices
- ...are elements (like Linked List Nodes)
- ...store a value of type V

Edges
- ...are also elements
- ...store a value of type E
Edge List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(m)$
- vertex.incidentEdges: $O(m)$
- vertex.edgeTo: $O(m)$
- Space Used: $O(n) + O(m)$
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(\text{deg}(\text{vertex}))$
- vertex.incidentEdges: $O(\text{deg}(\text{vertex}))$
- vertex.edgeTo: $O(\text{deg}(\text{vertex}))$
- Space Used: $O(n) + O(m)$
Adjacency Matrix Summary

- addEdge, removeEdge: $O(1)$
- addVertex, removeVertex: $O(n^2)$
- vertex.incidentEdges: $O(n)$
- vertex.edgeTo: $O(1)$
- Space Used: $O(n^2)$
DFS vs BFS

DFS (LIFO order...Stacks)

BFS (FIFO order...Queues)

BACK Edge($v, w$): $w$ is an ancestor of $v$ in the discovery tree

CROSS Edge($v, w$): $w$ is at the same or next level as $v$
# DFS Traversal vs BFS Traversal

<table>
<thead>
<tr>
<th>Application</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning Trees</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Connected Components</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Paths/Connectivity</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cycles</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shortest Paths</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Articulation Points</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>