Midterm Review
Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
Data Structures So Far

- **Array List**
  - (Extendable) Array

- **Node List**
  - Singly or Doubly Linked List

- **Stack**
  - Array
  - Singly Linked List

- **Queue**
  - Array
  - Singly or Doubly Linked List

- **Priority Queue**
  - Unsorted doubly-linked list
  - Sorted doubly-linked list
  - Heap (array-based)

- **Adaptable Priority Queue**
  - Sorted doubly-linked list with location-aware entries
  - Heap with location-aware entries

- **Tree**
  - Linked Structure

- **Binary Tree**
  - Linked Structure
  - Array
Topics on the Midterm

- **Data Structures & Object-Oriented Design**
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
Data Structures & Object-Oriented Design

- Definitions
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- Abstract Data Types & Interfaces
- Casting
- Generics
- Pseudo-Code
Software Engineering

Software must be:

- Readable and understandable
  - Allows correctness to be verified, and software to be easily updated.
- Correct and complete
  - Works correctly for all expected inputs
- Robust
  - Capable of handling unexpected inputs.
- Adaptable
  - All programs evolve over time. Programs should be designed so that re-use, generalization and modification is easy.
- Portable
  - Easily ported to new hardware or operating system platforms.
- Efficient
  - Makes reasonable use of time and memory resources.
Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant $\approx 1$
  - Logarithmic $\approx \log n$
  - Linear $\approx n$
  - N-Log-N $\approx n \log n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
  - Exponential $\approx 2^n$

- In a log-log chart, the slope of the line corresponds to the growth rate of the function.
Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
Some Math to Review

- Summations
- Logarithms and Exponents
- Existential and universal operators
- Proof techniques
- Basic probability

- **existential and universal operators**

  \[ \exists g \forall b \text{ Loves}(b, g) \]

  \[ \forall g \exists b \text{ Loves}(b, g) \]

- **properties of logarithms:**

  \[ \log_b(xy) = \log_b x + \log_b y \]

  \[ \log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y \]

  \[ \log_b x^a = a \log_b x \]

  \[ \log_b a = \frac{\log_x a}{\log_x b} \]

- **properties of exponentials:**

  \[ a^{(b+c)} = a^b a^c \]

  \[ a^{bc} = (a^b)^c \]

  \[ a^b / a^c = a^{(b-c)} \]

  \[ b = a^{\log_a b} \]

  \[ b^c = a^{c \log_a b} \]
Definition of “Big Oh”

\[ f(n) \in O(g(n)) \]

\[ \exists c, n_0 > 0 : \forall n \geq n_0, f(n) \leq cg(n) \]
Arithmetic Progression

- The running time of \textit{prefixAverages1} is $O(1 + 2 + \ldots + n)$

- The sum of the first $n$ integers is $\frac{n(n + 1)}{2}$
  
  - There is a simple visual proof of this fact

- Thus, algorithm \textit{prefixAverages1} runs in $O(n^2)$ time
Relatives of Big-Oh

big-Omega

- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

big-Theta

- $f(n)$ is $\Theta(g(n))$ if there are constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for $n \geq n_0$
The time complexity of an algorithm is the \textit{largest} time required on \textit{any} input of size \( n \). (Worst case analysis.)

- \( O(n^2) \): For any input size \( n \geq n_0 \), the algorithm takes no more than \( cn^2 \) time on every input.

- \( \Omega(n^2) \): For any input size \( n \geq n_0 \), the algorithm takes at least \( cn^2 \) time on at least one input.

- \( \Theta(n^2) \): Do both.
Time Complexity of a Problem

The time complexity of a problem is the time complexity of the \textit{fastest} algorithm that solves the problem.

- $O(n^2)$: Provide an algorithm that solves the problem in no more than this time.
  - Remember: for every input, i.e. worst case analysis!
- $\Omega(n^2)$: Prove that no algorithm can solve it faster.
  - Remember: only need one input that takes at least this long!
- $\Theta(n^2)$: Do both.
Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- **Linear Data Structures**
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
Arrays
Arrays

- Array: a sequence of indexed components with the following properties:
  - **array size is fixed** at the time of array’s construction
    - `int[] numbers = new int [10];`
  - **array elements are placed contiguously** in memory
    - Address of any element can be calculated directly as its offset from the beginning of the array
  - Consequently, array components **can be efficiently inspected or updated** in O(1) time, using their indices
    - `randomNumber = numbers[5];`
    - `numbers[2] = 100;`
Arrays in Java

- Since an array is an object, the name of the array is actually a **reference** (pointer) to the place in memory where the array is stored.

  - reference to an object holds the **address** of the actual object

- Example [arrays as objects]

  ```java
  int[] A={12, 24, 37, 53, 67};
  int[] B=A;
  B[3]=5;
  ```

- Example [cloning an array]

  ```java
  int[] A={12, 24, 37, 53, 67};
  int[] B=A.clone();
  B[3]=5;
  ```
Example

Example  [ 2D array in Java = array of arrays]

```java
int[][] nums = new int[5][4];
int[][] nums;
nums = new int[5][];
for (int i=0; i<5; i++) {
    nums[i] = new int[4];
}
```

A 5x4 integer array

```
nums

  2  8  1  6
  1  6  5  3
  3  2  6  4
  2  9  7  2
  9  3  1  5
```
Array Lists
The Array List ADT (§6.1)

- The **Array List** ADT extends the notion of array by storing a sequence of arbitrary objects.

- An element can be accessed, inserted or removed by specifying its rank (number of elements preceding it).

- An exception is thrown if an incorrect rank is specified (e.g., a negative rank).
The Array List ADT

```java
public interface IndexList<E> {

    /** Returns the number of elements in this list */
    public int size();

    /** Returns whether the list is empty. */
    public boolean isEmpty();

    /** Inserts an element e to be at index I, shifting all elements after this. */
    public void add(int I, E e) throws IndexOutOfBoundsException;

    /** Returns the element at index I, without removing it. */
    public E get(int i) throws IndexOutOfBoundsException;

    /** Removes and returns the element at index I, shifting the elements after this. */
    public E remove(int i) throws IndexOutOfBoundsException;

    /** Replaces the element at index I with e, returning the previous element at i. */
    public E set(int I, E e) throws IndexOutOfBoundsException;

}
```
Performance

- In the array based implementation
  - The space used by the data structure is $O(n)$
  - `size`, `isEmpty`, `get` and `set` run in $O(1)$ time
  - `add` and `remove` run in $O(n)$ time

- In an `add` operation, when the array is full, instead of throwing an exception, we could replace the array with a larger one.

- In fact `java.util.ArrayList` implements this ADT using extendable arrays that do just this.
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of $n$ add(o) operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 = 2n - 1$$

geometric series

- Thus $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$!

Recall:
$$\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$
Stacks

Chapter 5.1
The Stack ADT

- The **Stack** ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser

**Main stack operations:**
- `push(object)`: inserts an element
- `pop()`: removes and returns the last inserted element

**Auxiliary stack operations:**
- `top()`: returns the last inserted element without removing it
- `size()`: returns the number of elements stored
- `isEmpty()`: indicates whether no elements are stored
Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm `size()`
```java
return t + 1
```

Algorithm `pop()`
```java
if isEmpty() then
    throw EmptyStackException
else
    t ← t - 1
    return S[t + 1]
```
Queues

Chapters 5.2-5.3
Array-Based Queue

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the front and rear
  - $f$  index of the front element
  - $r$  index immediately past the rear element
- Array location $r$ is kept empty

![Normal configuration](image1)

![Wrapped-around configuration](image2)
Queue Operations

- We use the modulo operator (remainder of division)

Algorithm `size()`
return \((N - f + r) \mod N\)

Algorithm `isEmpty()`
return \((f = r)\)

Note: \(N - f + r = (r + N) - f\)
Linked Lists

Chapters 3.2 – 3.3
A singly linked list is a concrete data structure consisting of a sequence of nodes.

Each node stores:
- element
- link to the next node

Diagram:

- Node A
- Node B
- Node C
- Node D
- Next pointer
- Element
- Node
Running Time

- Adding at the head is $O(1)$
- Removing at the head is $O(1)$
- How about tail operations?
Doubly Linked List

- Doubly-linked lists allow more flexible list management (constant time operations at both ends).

- Nodes store:
  - element
  - link to the previous node
  - link to the next node

- Special trailer and header (sentinel) nodes
Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
Iterators

- An **Iterator** is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.

- You get an Iterator for a collection by calling its iterator method.

- Suppose **collection** is an instance of a **Collection**. Then to print out each element on a separate line:

  ```java
  Iterator<E> it = collection.iterator();
  while (it.hasNext())
    System.out.println(it.next());
  ```
The Java Collections Framework (Ordered Data Types)
Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
Linear Recursion Design Pattern

➢ Test for base cases

- Begin by testing for a set of base cases (there should be at least one).
- Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.

➢ Recurse once

- Perform a single recursive call. (This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.)
- Define each possible recursive call so that it makes progress towards a base case.
Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.

- Example 1: The Fibonacci Sequence
Formal Definition of Rooted Tree

- A rooted tree may be empty.
- Otherwise, it consists of
  - A root node \( r \)
  - A set of subtrees whose roots are the children of \( r \)
Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
Tree Terminology

- **Root**: node without parent (A)
- **Internal node**: node with at least one child (A, B, C, F)
- **External node (a.k.a. leaf)**: node without children (E, I, J, K, G, H, D)
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc.
- **Descendant of a node**: child, grandchild, grand-grandchild, etc.
- **Siblings**: two nodes having the same parent
- **Depth of a node**: number of ancestors (excluding self)
- **Height of a tree**: maximum depth of any node (3)
- **Subtree**: tree consisting of a node and its descendants
Position ADT

- The **Position** ADT models the notion of place within a data structure where a single object is stored.

- It gives a unified view of diverse ways of storing data, such as:
  - a cell of an array
  - a node of a linked list
  - a node of a tree

- Just one method:
  - `object element():` returns the element stored at the position
Tree ADT

- We use positions to abstract nodes

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()

- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)

- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)

- Update method:
  - object replace(p, o)
  - Additional update methods may be defined by data structures implementing the Tree ADT

- Additional update methods may be defined by data structures implementing the Tree ADT
Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner.
- In a preorder traversal, a node is visited before its descendants.

Algorithm `preOrder(v)`

- `visit(v)`
- `for each child w of v
  `preOrder(w)`

1. Make Money Fast!
   - 2. Motivations
     - 3. 1.1 Greed
     - 4. 1.2 Avidity
   - 5. 2. Methods
     - 6. 2.1 Stock Fraud
     - 7. 2.2 Ponzi Scheme
     - 8. 2.3 Bank Robbery
   - 9. References
In a postorder traversal, a node is visited after its descendants.

Algorithm `postOrder(v)`

for each child `w` of `v`

`postOrder(w)`

`visit(v)`
Properties of Proper Binary Trees

- **Notation**
  - $n$ number of nodes
  - $e$ number of external nodes
  - $i$ number of internal nodes
  - $h$ height

- **Properties:**
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2(n + 1) - 1$
BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT.

- Additional methods:
  - position $\text{left}(p)$
  - position $\text{right}(p)$
  - boolean $\text{hasLeft}(p)$
  - boolean $\text{hasRight}(p)$

- Update methods may be defined by data structures implementing the BinaryTree ADT.
Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
Priority Queue ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - `insert(k, x)` inserts an entry with key k and value x
  - `removeMin()` removes and returns the entry with smallest key
- Additional methods
  - `min()` returns, but does not remove, an entry with smallest key
  - `size()`, `isEmpty()`
- Applications:
  - Process scheduling
  - Standby flyers
Entry ADT

- An **entry** in a priority queue is simply a key-value pair

- Methods:
  - **key()**: returns the key for this entry
  - **value()**: returns the value for this entry

- As a Java interface:

```java
/**
 * Interface for a key-value pair entry
 */

public interface Entry {
    public Object key();
    public Object value();
}
```
Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation.
- A generic priority queue uses an auxiliary comparator.
- The comparator is external to the keys being compared.
- When the priority queue needs to compare two keys, it uses its comparator.
- The primary method of the Comparator ADT:
  - `compare(a, b)`: Returns an integer $i$ such that
    - $i < 0$ if $a < b$
    - $i = 0$ if $a = b$
    - $i > 0$ if $a > b$
    - an error occurs if $a$ and $b$ cannot be compared.
Sequence-based Priority Queue

- Implementation with an unsorted list
  4 5 2 3 1

- Performance:
  - `insert` takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - `removeMin` and `min` take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list
  1 2 3 4 5

- Performance:
  - `insert` takes $O(n)$ time since we have to find the right place to insert the item
  - `removeMin` and `min` take $O(1)$ time, since the smallest key is at the beginning

Is this tradeoff inevitable?
Heaps

- Goal:
  - $O(\log n)$ insertion
  - $O(\log n)$ removal

- Remember that $O(\log n)$ is almost as good as $O(1)$!
  - e.g., $n = 1,000,000,000 \Rightarrow \log n \approx 30$

- There are min heaps and max heaps. We will assume min heaps.
Min Heaps

A min heap is a binary tree storing keys at its nodes and satisfying the following properties:

- **Heap-order**: for every internal node $v$ other than the root
  - $key(v) \geq key(parent(v))$

- **(Almost) complete binary tree**: let $h$ be the height of the heap
  - for $i = 0, \ldots, h - 1$, there are $2^i$ nodes of depth $i$
  - at depth $h - 1$
    - the internal nodes are to the left of the external nodes
    - Only the rightmost internal node may have a single child

- The last node of a heap is the rightmost node of depth $h$
After the insertion of a new key $k$, the heap-order property may be violated.

Algorithm **upheap** restores the heap-order property by swapping $k$ along an upward path from the insertion node.

**Upheap** terminates when the key $k$ reaches the root or a node whose parent has a key smaller than or equal to $k$.

Since a heap has height $O(\log n)$, **upheap** runs in $O(\log n)$ time.
Downheap

- After replacing the root key with the key $k$ of the last node, the heap-order property may be violated.
- Algorithm downheap restores the heap-order property by swapping key $k$ along a downward path from the root.
- Note that there are, in general, many possible downward paths – which one do we choose?
Downheap

- We select the downward path through the **minimum-key** nodes.
- Downheap terminates when key \( k \) reaches a leaf or a node whose children have keys greater than or equal to \( k \).
- Since a heap has height \( O(\log n) \), downheap runs in \( O(\log n) \) time.
Array-based Heap Implementation

- We can represent a heap with \( n \) keys by means of an array of length \( n + 1 \).
- Links between nodes are not explicitly stored.
- The cell at rank 0 is not used.
- The root is stored at rank 1.
- For the node at rank \( i \):
  - the left child is at rank \( 2i \)
  - the right child is at rank \( 2i + 1 \)
  - the parent is at rank \( \text{floor}(i/2) \)
  - if \( 2i + 1 > n \), the node has no right child
  - if \( 2i > n \), the node is a leaf
We can construct a heap storing $n$ keys using a bottom-up construction with $\log n$ phases.

In phase $i$, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys.

Run time for construction is $O(n)$. 

Bottom-up Heap Construction
Adaptable Priority Queues
Additional Methods of the Adaptable Priority Queue ADT

- **remove(e):** Remove from $P$ and return entry $e$.

- **replaceKey(e,k):** Replace with $k$ and return the old key; an error condition occurs if $k$ is invalid (that is, $k$ cannot be compared with other keys).

- **replaceValue(e,x):** Replace with $x$ and return the old value.
Location-Aware Entries

- A locator-aware entry identifies and tracks the location of its (key, value) object within a data structure
List Implementation

- A location-aware list entry is an object storing:
  - key
  - value
  - position (or rank) of the item in the list
- In turn, the position (or array cell) stores the entry
- Back pointers (or ranks) are updated during swaps
Heap Implementation

- A location-aware heap entry is an object storing
  - key
  - value
  - position of the entry in the underlying heap
- In turn, each heap position stores an entry
- Back pointers are updated during entry swaps
Performance

- Times better than those achievable without location-aware entries are highlighted in red:

<table>
<thead>
<tr>
<th>Method</th>
<th>Unsorted List</th>
<th>Sorted List</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>min</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeMin</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>replaceKey</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>replaceValue</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps