CMPT 225: Data Structures & Programming
– Unit 20 –
Midterm Review

Dr. Jack Thomas

Simon Fraser University

Spring 2021

The March 8th Midterm

- **Monday, March 8th, 11:30am to 12:30pm.**
- One attempt, one hour, MUST be completed within this time.
- Completed on Canvas.
- If you can't complete it at the given time, **NOTIFY ME ASAP**!

Format

- Three types of questions:
  - **Very Short Answer Questions**: Answers should be a sentence or two.
  - **Short Answer Questions**: A paragraph (or equivalent).
  - **Code Questions**: Questions that involve coding. Highly recommend you open the IDE to a blank project before you begin the midterm so you can code there and then copy-paste your answer over.

Academic Integrity

- The midterm is **open book**, meaning you're free to consult your notes, course material, or even the open internet.
- You **may NOT cooperate with anyone** to complete your midterm, especially other students.
- Any source you use outside of course material **must be cited** – looking code up is fine, lifting code directly will be treated as plagiarism.

Content

- The midterm will cover everything up to the end of **Heaps** (**unit 16**), meaning no APQs, Maps, Hash Tables, or Ordered Maps.
- We won't be taking questions directly from the assignments, labs, or textbooks, but they may be similar.
- The **exam will be cumulative**, so don't go forgetting everything as soon as the test is over!

How To Study For The Midterm

1. Attend this review (good job!)
2. Consult your notes.
3. Check the slides
4. Watch the recordings.
5. Go through your code and the sample solutions.

Object-Oriented Programming

- A **paradigm** for organizing code into discrete "objects", each complete and self-contained.
- The **Four Principles**
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy

Object-Oriented Programming

- **Inheritance**, how objects are organized into a **hierarchy** who inherit **fields** and **methods** from predecessors
- **Polymorphism** allows multiple related objects to fulfill the same purpose (e.g. Chihuahuas and Daschunds are both dogs).
  - **Overriding** a function inherited from a **superclass** with a new version in the **subclass**.
  - **Overloading** a function with another version that has a different **signature**.

Arrays

- A **primitive data structure** whose size is fixed at declaration, reserving a contiguous block of memory.
- Built-in quite deeply to Java, and most other programming languages too.
- Arrays can also be **multi-dimensional** – you can have arrays of arrays.

Arrays

- Remember **Insertion Sort**? An algorithm for sorting an array.

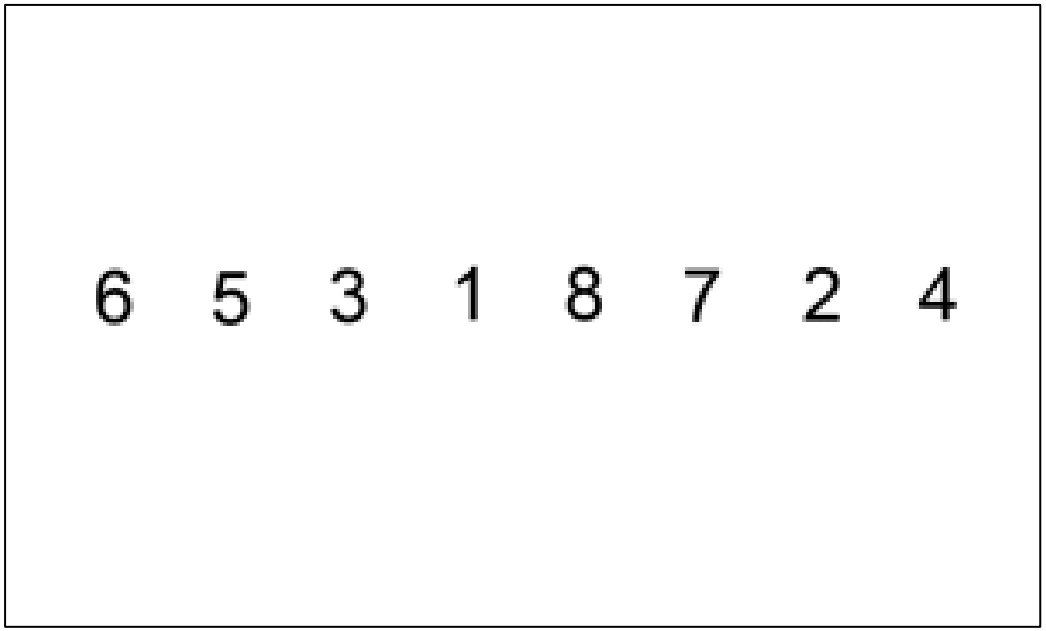


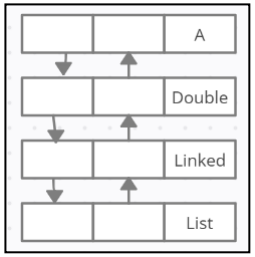
Image credit: https://upload.wikimedia.org/wikipedia/commons/0/0f/Insertion-sort-example-300px.gif

Lists

- An alternative primitive data structure made of **nodes**, each of which stores an **element** of data.
- Typically made of a **list class** that stores a **head** node (and possibly a **tail** node) and tracks the **length** of the list, along with functions for **inserting** and **removing** from the list.
- Java provides a List interface, while **LinkedList** is a good all-purpose list data structure we'll reuse a lot.

Lists

- Nodes in a **singly-linked** list store a link to a next node, to create a linear sequence, while **doubly-linked** lists store two links (next and previous).



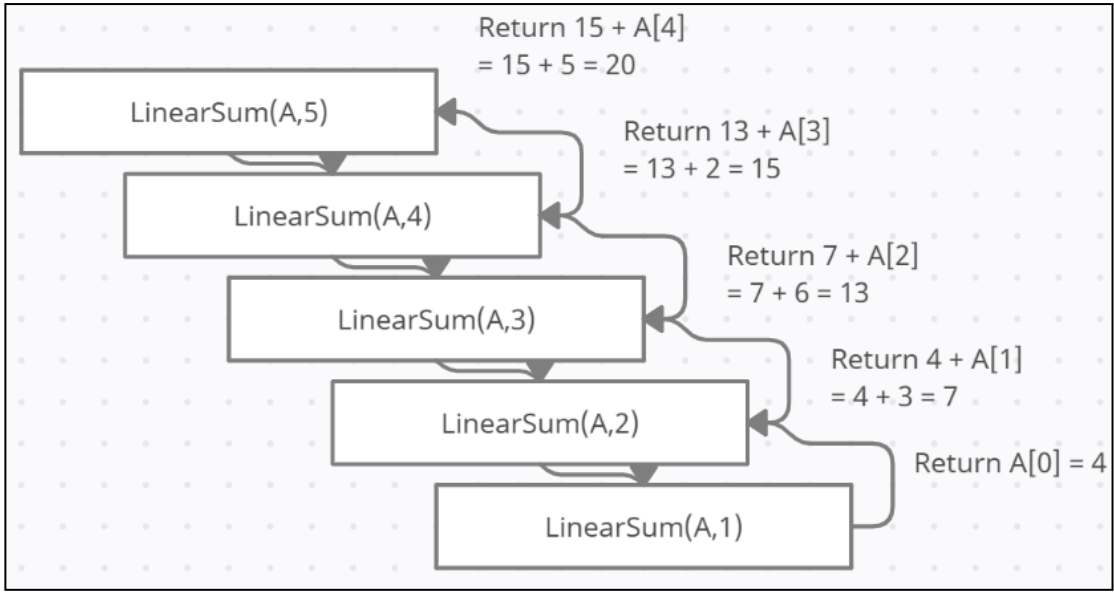
- **Sentinel Nodes** are special blank nodes we can include at the front (header) and back (trailer) of doubly-linked lists to make some algorithms easier to implement.
- **Circular lists?**

Recursion

- When **a function calls itself**, pausing the current instance and starting a new one.
- Each call ends with the function either **recursively calling itself again** (usually on a different set of data) or reaching a **base case** that returns something and allows the stack of recursive calls to start resolving themselves.

Recursion

- Recursion Tracing is a way of visualizing a recursive process.



- Linear Recursion only calls itself once per call, Binary Recursion calls itself twice per call, Multiple Recursion goes further.

Analysis

- Remember what it means for a program to not only work, but to be good – it doesn't just solve the problem, it does it **optimally**.
- Our analysis tools are for figuring out how much **space** our structures take up and how much **time** it takes our algorithms to run on them.
- Time is usually the bigger factor, which either leads to **experimental studies** or analyzing the number of **primitive operations** in our algorithms.

Analysis

- The Seven Important Functions
  1. Constant (1)
  2. Logarithmic (log n)
  3. Linear (n)
  4. N-Log-N (n log n, usually with a log base of 2)
  5. Quadratic (n^2)
  6. Cubic / Polynomial (n^3 or more)
  7. Exponential (x^n)
- They're important because they describe different **rates of growth**.

Analysis

- **Asymptotic Analysis** measures performance by how the run-time of a function grows as the number of inputs grows.
- **Big-Oh Notation** tries to match the tightest-fitting function to the worst-case time performance, like O(n) for a function with a loop that runs once per input.
- There's also **Big-Omega** (best-case) and **Big-Theta** (the actual growth rate).

Stacks

- The first of our more advanced data structures, new elements are “pushed” on to the top of the stack, and then “popped” back off of the top.
- Could be based on a list or an array, since it defines how data is accessed, not stored.
- Java includes a standard Stack class built-in.

Stack: The ADT

- A Stack stores a set of objects.
- Follows FILO (first-in-last-out).
- Standard Stack operations include:
  - **Push**: Add an element to the top of the Stack.
  - **Pop**: Remove the top element.
  - **Top**: Return what's on top of the stack without removing it.
  - **Size**: How many things are on the Stack?
  - **Empty**: Is the stack empty? Yes or no.

Queues

- Cousin of the Stack, except you add elements to the back and take them from the front.

- Java doesn’t have a built-in Queue class, but does have a Queue interface, and you can use a LinkedList as a Queue pretty easily.

Queue: The ADT

- A Queue stores a set of objects.
- Follows (FIFO) (first-in-first-out).
- Standard Queue operations include:
  - **Enqueue**: Add an element to the back of the queue.
  - **Dequeue**: Remove and return the element at the front of the queue.
  - **Front**: Return what's at the front of the queue without removing it.
  - **Size**: How many things are in the queue?
  - **isEmpty**: Is the queue empty? Yes or no.

Deques

- Double-Ended Queues, essentially both a Queue and a Stack.
- Java has a Deque interface, there's also an ArrayDeque class, and a doubly-linked list is a good basis if you're implementing one.

Deque: The ADT

- A Deque stores a set of objects.
- Follows neither FIFO nor FILO.
- Standard Deque operations include:
  - **addFirst**: Inserts a new element at the head.
  - **addLast**: Inserts a new element at the tail
  - **removeFirst**: Removes and returns the element at the head.
  - **removeLast**: Removes and returns the element at the tail.
  - **getFirst**: Returns (but doesn't remove) the element at the head.
  - **getLast**: Returns (but doesn't remove) the element at the tail.
  - **Size**: How many things are in the queue?
  - **isEmpty**: Is the queue empty? Yes or no.

Adapter Design Pattern

- **Design Patterns as best practices** for solving programming problems.
- The **Adapter** (or **Wrapper**) is a design for an object that looks like one data structure but is based on another, to allow two structures to interface.
- In **Java**, you can **extend** the class you're emulating to make it official, while storing the other class as a variable and using its functions to fill out the functions of the class you're extending.

Array Lists

- The more advanced version of Array, as a full data structure like Queue or Stack.
- Based on the Sequence, the more formal name for linear data structures, and accessed by an index.
- There’s a standard Java version, which also handles resizing itself by doubling its capacity whenever add() pushes it over.

The Array List ADT

- A linear sequence of data elements, organized along and accessed by its index.
- Essentially the full data structure version of what arrays do.
- Standard methods include:
  - **Get**: Returns the element at a given index.
  - **Set**: Replaces the element at a given index with a given element, returns the old element.
  - **Add**: Adds a new element at the given index and increases the size.
  - **Remove**: Removes the element at a given index and decreases the size.
  - **Size**: Returns the number of elements stored in the Array List.
  - **isEmpty**: Returns whether the Array List is empty.

General Trees

- The beginning of the **non-linear data structures**, unlike the sequence-based ones.
- A **Tree** is made of **vertices** (our nodes) connected to at least one other vertex by an **edge** (our links). These terms are used while visualizing Trees.
- Trees **may not have cycles or disconnected vertices**, all connections are **one-to-one**, which means there's **only one path** from any vertex to any other vertex.

General Trees

- Tree Terminology:
  - Vertex
  - Adjacent neighbours
  - Degree
  - Leaves (external nodes)
  - Internal Nodes
  - Distance

Tree: The ADT

- A data structure storing a non-linear set of data elements.
- These elements are organized into a hierarchy.
- Methods of a Tree include:
  - **Element**: Returns the object stored in a given node.
  - **Root**: Returns the root of a Tree.
  - **Parent**: Returns the parent of a given node.
  - **Children**: Returns a collection of the nodes that are children of a given node.
  - **isInternal**: Tests whether a node is internal.
  - **isExternal**: Tests whether a node is external (a leaf).
  - **isRoot**: Tests whether a node is the root.

General Trees

- **Rooted Trees** are a common type of Tree that have a special Root node, with the rest of the nodes descending “down” from it.
- Nodes now have a **parent**, which is the neighbouring node that leads back toward the root, and **children**, which is any other neighbouring node.
- This creates a ton of other family relations (grandchildren/grandparents/siblings/descendants/ancestors)

General Trees

- There is **no general Tree class or interface in Java**, but some based on specific Tree variants.
- You can implement your own basic tree pretty easily, however – very similar to building your own list.
- How your add, remove, get, sort, and search methods work (and their efficiency) **varies greatly based on the variant**.

General Trees

- Most Tree methods rely on **traversals**, which is how you navigate a Tree. Again, depends a lot on variant (how many children, whether there's a root, etc).
- **Pre-order traversals** work down through a node's children to the leaves. **Post-order traversals** work up through the node's parents to the root.

Binary Trees

- A **constrained version of a rooted Tree**, where each node may have only two, one, or zero children.
- Has a **number of properties** surrounding the height of the tree, depth of any one node, number of nodes total, and number of internal and external nodes.

Binary Tree ADT

- A **subtype of the Tree** data structure which limits nodes to a maximum of two ordered children.
- Includes all of the methods and properties of the general Tree.
- Binary Trees include the following methods:
  - **Left**: Returns the left child.
  - **Right**: Returns the right child.
  - **hasLeft**: Confirms whether there's a left child.
  - **hasRight**: Confirms whether there's a right child.

Binary Trees

- Allows for **In-Order traversals**, which first visits a node's left child, then the node itself, then the node's right child.
- The generalized traversal is the **Euler Tour traversal**, which tours around every node, with the other three traversals being sub-types of the Euler.

Priority Queues

- **Key-based** data structures use a key paired with a value within an entry for retrieving and storing data, instead of the entry's position within the structure.
- **Priority Queues** are like **Queues** that return the **entry with the highest priority** (smallest key) instead of the oldest entry.

The Priority Queue ADT

- A **data structure** for storing **entries** containing data **values** and **keys**.
- **Based on keys** included with each entry **rather than their positions** in the queue.
- Standard methods include:
  - **Insert**: Adds a given key and value to the Priority Queue, and returns their combined entry.
  - **removeMin**: Removes and returns an entry of P with the smallest key. (Sometimes called poll, from queue)
  - **Min**: Returns but does not remove an entry of P with the smallest key. (Sometimes called peek, from queue)
  - The usual generic methods from Queue as well, like isEmpty() and size().

Priority Queues

- Java has a **standard PriorityQueue class**, which accepts a **Comparator** you can define for which of two entries has the smaller key.
- Priority Queue's performance depends on the underlying structure.

Heaps

- A data structure that combines **non-positional and non-linear properties** to always store the entry with the smallest key at the root.
- Based on a **Complete Binary Tree**, where every node may have zero, one, or two children and the order of where the next node must be attached follows a strict pattern.
- The location for the next node is called the **last node**, which essentially requires each “row” of the heap to be filled up from left to right before a new row may be started.

Heaps

- The **Heap-Ordering Property** says that the key stored at each node must be greater than or equal to the key stored by that node's parent.
- When a new entry is added, we do **Up-Heap Bubbling** to see how high it must climb the Tree to maintain Heap Ordering.
- When the root is removed, we do **Down-Heap Bubbling** to make one of the root's children the new root and restore the order.

Heaps

- Heaps **don't have an ADT**, they're the name for Complete Binary Trees who obey the Heap Ordering property.
- The standard Java **PriorityQueue** class is **based on a Heap**.
- In terms of efficiency, **Heaps balance the time for adding and removing** from Priority Queues to O(log n), compared to O(1)/O(n) for unsorted lists and O(n)/O(1) for sorted lists.

Recap – Ooooh, We're Halfway There!

- The Midterm is on **Monday at 11:30am** and must be submitted by 12:30pm – you have one hour!
- It's on **Canvas**, with a mix of theory and coding questions, so **open up your IDE**.
- It covers everything up to the end of **unit 16** on heaps.
- It's **open book**, but **no cooperating with others** or lifting solutions directly from the internet. ***Cite any sources used***.
- I'll be available on **Discord** and in the **virtual lecture room** if you need me!