Midterm Review
When and Where

• When
  • Thursday, November 2 between 4:30pm and 6:20pm.

• Where
  • M3 1006, MC 2017.
Midterm Focus

• Concepts you learn in the course
  • Less on memorizing
  • More on understanding and applying
Coverage

• Abstraction, ADT
• Array, Python List
• Analysis of Algorithms
• Linked Lists
• Searching
• Sorting
• Stack
• Queue
• Hash Table
Materials

• Course Slides
  • available under “Lectures” on course website

• Chalkboard Notes

• Assignments

• Need more help?
  • Office hours
    • Monday 9:15-11:15am, DC2129
    • Wednesday 1:30-2:30pm, Friday 2:00-3:00pm, MC4065
Abstractions

- Used by computer scientists to help manage complex problems.
  - **abstraction**
    - a mechanism for separating the properties of an object, and
    - restricting the focus to those relevant in the current context.

- Focus on the “what” not the “how”.

- Abstraction means capture the relevant (or important) aspects
Advantages

• Several advantages of working with ADTs:
  • Focus on solving the problem at hand. Focus on functionality, instead of implementation.
  • Help to reduce logical errors from misuse of the data type by limiting controlling the use of implementation details.
  • Can easily change the implementation w/o affecting the use of the ADT.
  • Easier to manage and divide larger problems into smaller parts.
Array -vs- Python list

• Similar:
  • Both are sequences
  • composed of multiple sequential elements
  • can be accessed by position

• Two major differences:
  • Arrays only have 3 operations:
    • array creation
    • reading a specific element
    • writing a specific element
  • The size of an array is fixed
When to use Arrays?

- Arrays are best suited to problems where:
  - maximum number of elements is known up front.
    - array size is fixed
    - the list has extra space that can be wasteful.
  - only a limited number of operations are needed.
    - arrays have 3 operations.
    - the list can manage the items in the container.
Implementing the 2-D Array

- There are various approaches that can be used to implement a 2-D array.
  - Use a 1-D array of 1-D arrays.
  - Use a single 1-D array with the elements arranged by row or column.
Computing Resources

• space or memory
  • Typically dictated by the problem instance

• execution time (running time)
  • actual time required for the program to compute its result
  • the starting time and ending time wrt the system used
Example Algorithm (v.1)

• How many addition operations?

\[ T(n) = 2n(n) = 2n^2 \]

```python
rowSum = Array(n)
totalSum = 0

for i in range(n):
    rowSum[i] = 0

for i in range(n):
    for j in range(n):
        rowSum[i] = rowSum[i] + matrix[i,j]

    totalSum = totalSum + matrix[i,j]
```
**Big-O Definition**

- The order of magnitude: Big-O notation.
- Then, the algorithm has a **time-complexity** of or executes “on the order of” $f(n)$
  - We use the notation: $O(f(n))$
  - Big-O is intended for large values of $n$.

---

$f(n)$ indicates the rate of growth at which the run time increases as the input size increases.
Classes of Algorithms

- Many algorithms have a time-complexity selected from a common set of functions.

<table>
<thead>
<tr>
<th>f()</th>
<th>Common Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
</tr>
<tr>
<td>\log n</td>
<td>logarithmic</td>
</tr>
<tr>
<td>n</td>
<td>linear</td>
</tr>
<tr>
<td>n \log n</td>
<td>log linear</td>
</tr>
<tr>
<td>\text{n}^2</td>
<td>quadratic</td>
</tr>
<tr>
<td>\text{n}^3</td>
<td>cubic</td>
</tr>
<tr>
<td>a^n</td>
<td>exponential</td>
</tr>
</tbody>
</table>
Node Definition

- The nodes are constructed from a simple storage class:

```python
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None
```
Linked Structure

- Constructed using a collection of objects called nodes.
- Each node contains data and at least one reference or link to another node.
- **Linked list** – a linked structure in which the nodes are linked together in linear order.
Linked List

- Most nodes have no name; they are referenced via the link of the preceding node.
- **head reference** – the first node must be named or referenced by an external variable.
  - Provides an entry point into the linked list.
  - An empty list is indicated by a null head reference.
Singly Linked List

- A linked list in which
  - each node contains a single link field and
  - allows for a complete linear order traversal from front to back.
- Several common operations can be performed.
Building Linked Lists

- There are more ways to build a linked list than simply prepending nodes to the front.
  - Appending nodes
  - Sorted linked lists
Doubly Linked List

- A linked list in which each node contains a data component(s) and two links:
  - one pointing the next node and
  - one pointing to the preceding node.
Circular Linked List

- Another variation of the linked list in which the nodes form a continuous circle.
  - Allows for a complete traversal from any initial node.
  - Used with round-robin type applications.
  - The external reference can point to any node in the list. Common to reference “end” of the list.
Searching

- The process of selecting particular information from a collection of data based on specific criteria.
  - Can be performed on different data structures.
  - **sequence search** – search within a sequence.
  - **search key** (or key) – identifies a specific item.
  - **compound key** – consists of multiple parts.
Binary Search

- The linear search has a linear time-complexity.
  - We can improve the search time if we modify the search technique itself.
  - Use a **divide and conquer** strategy.
  - Requires a sorted sequence.
Sorting

- The process of arranging a collection of items such that each item and its successor satisfy a prescribed relationship.
  - items can be simple types or complex types
  - **sort key** – values on which items are ordered.
  - items arranged in ascending or descending order.
Sorting Algorithms

- Can be divided into two categories:
  - **comparison sorts**
    - items are arranged by performing pairwise logical comparisons between two sort keys.
  - **distribution sorts**
    - distributes the sort keys into intermediate groups based on individual key values.
Bubble Sort

- A simple solution to the sorting problem.
- Arranges the items by
  - iterating over the sequence multiple times.
  - larger values bubble to the top (or end).
Bubble Sort Example

- First complete iteration of the inner loop.
Selection Sort

- Improves on the bubble sort.
- Works in a fashion similar to what a human may use to sort a sequence.
- Instead of swapping many items,
  - requires a search to select the smallest item.
  - makes a single swap after each pass
Selection Sort Example
Insertion Sort

- Another commonly studied algorithm.
- Arranges the items by
  - iterating over the sequence one complete time.
  - inserts each unsorted item into its proper place.
- To position an item:
  - find the correct spot within the sorted sequence
  - open the slot by shifting the items down one position
Insertion Sort Example

![Insertion Sort Example Diagram]
The efficiency of some algorithms can be improved when working with sorted sequences. eg. Binary Search
- For non-static collections, it would be inefficient to re-sort a sequence for each add/remove.
- Better to maintain a sorted sequence.

Static collection: no items will be added/removed to the sequence
Maintaining a Sorted List

- To maintain a sorted list, new items must be inserted into their proper position.
  - Can not simply be appended at the end.
  - Must locate the proper position and use `insert()`.
Merging Sorted Lists

- Sometimes it may be necessary to merge two sorted lists into a new list.
  - For example

```python
listA = [ 2, 8, 15, 23, 37 ]
listB = [ 4, 6, 15, 20 ]
newList = mergeSortedLists( listA, listB )
print( newList )
```

creates a new merged list

```
[2, 4, 6, 8, 15, 15, 20, 23, 37]
```
Merge Sort

- Uses a divide and conquer strategy to sort the keys stored in a sequence.
  - Keys are recursively divided into smaller and smaller subsequences.
  - Subsequences are merged back together.
Merge Sort – Divide

- Starts by splitting the original sequence in the middle to create two subsequences of approximately equal size.
Merge Sort – Conquer

- After the sequences are split, they are merge back together, two at a time to create sorted sequences.
Quick Sort

- Uses a divide and conquer strategy to sort the keys stored in a sequence.
  - Partitions the sequence by dividing it into two segments based on a pivot key.
  - Uses virtual subsequences without the need for temporary storage.

- Quick sort is a recursive algorithm.
Quick Sort – Efficiency

- The quick sort algorithm:
  - has an average case time of $O(n \log n)$
- It does not require additional storage.
- Commonly used in language libraries.
  - Earlier versions of Python used quick sort.
  - Current versions use a hybrid that combines the insertion and merge sort algorithms.
Radix Sort

- A fast distribution sorting algorithm.
  - Special purpose sorting algorithm.
  - Orders keys by examining individual key components instead of comparing them.
  - When used with integers, individual key digits are compared from least to most significant.
  - *aka* bin sort
    - dates back to the days of card readers.
Distribute the Keys

- The process starts by distributing the keys among the various bins.
  - Based on the digits in the ones column.
  - Stored in the order they occur in the sequence.
What if we need to sort keys stored in an unsorted linked list?

- Many of the algorithms used with sequences can be used.
- Instead of swapping values, nodes are unlinked and relinked as necessary.
Stacks

- A restricted access container that stores a linear collection.
  - Very common for solving problems in computer science.
  - Provides a last-in first-out (LIFO) protocol.
The Stack ADT

- A **stack** stores a linear collection of items with access limited to a last-in first-out order.
- Adding and removing items is restricted to the top of the stack.

- Stack()
- isEmpty()
- length()
- pop()
- peek()
- push(item)
Stack Implementation

- Several common ways to implement a stack:
  - Python list
    - easiest to implement
  - Linked list
    - better choice when a large number of push and pop operations are performed.
Stack Applications

- Many applications encountered in computer science requires the use of a stack.
  - Balanced delimiters
  - Postfix expressions
Types of Expressions

- Three different notations can be used:
  - infix: \( A + B \times C \)
  - prefix: \( + A \times B C \)
  - postfix: \( A B C \times + \)
Queue

- A restricted access container that stores a linear collection.
  - Very common for solving problems in computer science that require data to be processed in the order in which it was received.
  - Provides a **first-in first-out (FIFO)** protocol.

- New items are added at the **back** while existing items are removed from the **front** of the queue.
The Queue ADT

- A *queue* stores a linear collection of items with access limited to a first-in first-out order.
  - New items are added to the back.
  - Existing items are removed from the front.

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue()</td>
</tr>
<tr>
<td>isEmpty()</td>
</tr>
<tr>
<td>length()</td>
</tr>
<tr>
<td>enqueue( item )</td>
</tr>
<tr>
<td>dequeue()</td>
</tr>
</tbody>
</table>
Queue Implementation

- Several common ways to implement a stack:
  - Python list
    - easiest to implement
  - Circular array
    - fast operations with a fixed size queue.
  - Linked list
    - reduces memory wastes by eliminating the extra capacity created with a vector.
Priority Queues

- Some applications require the use of a queue in which items are assigned a priority.
  - higher priority items are dequeued first.
  - items with equal priority still follow FIFO.

- Two types:
  - **bounded** – limited range of priorities.
  - **unbounded** – unlimited range.
A priority queue is a queue in which each item is assigned a priority and items with a higher priority are removed before those with lower priority.

- Integer values are used for the priorities.
- Smaller integers have a higher priority.

**Priority Queue ADT**

- `PriorityQueue()`
- `BpriorityQueue( numLevels )`
- `isEmpty()`
- `length()`
- `enqueue( item, priority )`
- `dequeue()`
Hashing

- The process of mapping a search key to a limited range of array indices.
  - The goal of providing direct access to the keys.
  - **hash table** – the array containing the keys.
  - **hash function** – maps a key to an array index.
Probing

- If two keys map to the same table entry, we must resolve the collision to find another available slot.
  - **linear probe** – simplest approach which examines the table entries in sequential order.
Modified Linear Probe

- We can improve the linear probe by changing the step size to some fixed constant.

\[
\text{slot} = (\text{home} + i \times c) \mod M
\]

- Suppose we set \( c = 3 \) to build the hash table.

<table>
<thead>
<tr>
<th>( h(765) ) =&gt; 11</th>
<th>( h(579) ) =&gt; 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(431) ) =&gt; 2</td>
<td>( h(226) ) =&gt; 5</td>
</tr>
<tr>
<td>( h(96) ) =&gt; 5</td>
<td>( h(903) ) =&gt; 6</td>
</tr>
<tr>
<td>( h(142) ) =&gt; 12</td>
<td>( h(388) ) =&gt; 11</td>
</tr>
</tbody>
</table>

![Hash Table Diagram]
Quadratic Probing

- A better approach for reducing primary clustering.
  \[ \text{slot} = (\text{home} + i^2) \mod M \]

- Increases the distance between each probe in the sequence.

- Example:

  \[
  \begin{align*}
  h(765) &\Rightarrow 11 & h(579) &\Rightarrow 7 \\
  h(431) &\Rightarrow 2 & h(226) &\Rightarrow 5 &\Rightarrow 6 \\
  h(96) &\Rightarrow 5 & h(903) &\Rightarrow 6 &\Rightarrow 7 &\Rightarrow 10 \\
  h(142) &\Rightarrow 12 & h(388) &\Rightarrow 11 &\Rightarrow 12 &\Rightarrow 2 &\Rightarrow 7 &\Rightarrow 1 \\
  \end{align*}
  \]
Double Hashing

- When a collision occurs, a second hash function is used to build a probe sequence.
  \[ \text{slot} = (\text{home} + i \times \text{hp(key)}) \mod M \]

- Step size remains a constant throughout the probe.
- Multiple keys that have the same home position, will have different probe sequences.
Rehashing

- We can start with a small table and expand it as needed.
  - Similar to the approach used with the vector.
- **load factor** – the ratio between the number of keys and the size of the table.
  - A hash table should be expanded before the load factor reaches 80%.
Separate Chaining

- We can eliminate collisions altogether if we store the keys outside the table.
  - chains – use linked lists to store keys that map to the same entry.
  - The hash table becomes an array of linked lists.
  - After mapping the key to an entry in the table, the linked list is searched for the key.
Questions During the Exam

• If you need some clarification during the exam, raise your hand and a proctor will bring you a question form to fill out. The form will be taken to an instructor, who will either answer your question or decline to answer if it is inappropriate.

• If you are unsure of something, it may be less time-consuming to simply state your assumptions about the question instead of asking an instructor to clarify.

• To ensure that no student is unfairly advantaged, proctors will not answer questions individually during the exam.

• If you feel that a question contains an error, raise your hand. A proctor will bring the concern to the instructors and an announcement will made if necessary.