Course Wrapup
CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:
Aaron Johnston       Ethan Knutson       Nathan Lipiarski
Amanda Park         Farrell Fileas       Sam Long
Anish Velagapudi    Howard Xiao         Yifan Bai
Brian Chan          Jade Watkins        Yuma Tou
Elena Spasova       Lea Quan

Announcements

- Final is cancelled, but HW8 is still due
  - ... and there’s no late days

- Please fill out your TA nominations!
  - [https://www.cs.washington.edu/students/ta/bandes](https://www.cs.washington.edu/students/ta/bandes)

- Lecture eval: [https://uw.iasystem.org/survey/219337](https://uw.iasystem.org/survey/219337)
Announcements

❖ Section Evals:

- AA: [https://uw.iasystem.org/survey/221482](https://uw.iasystem.org/survey/221482)
- AB: [https://uw.iasystem.org/survey/221455](https://uw.iasystem.org/survey/221455)
- AC: [https://uw.iasystem.org/survey/221537](https://uw.iasystem.org/survey/221537)
- AD: TBD
- AE: [https://uw.iasystem.org/survey/221470](https://uw.iasystem.org/survey/221470)
- AF: [https://uw.iasystem.org/survey/221507](https://uw.iasystem.org/survey/221507)
- AG: [https://uw.iasystem.org/survey/221496](https://uw.iasystem.org/survey/221496)
- AH: [https://uw.iasystem.org/survey/221521](https://uw.iasystem.org/survey/221521)
Two Key Skills

❖ In Software Engineering, two important skills to have are:
   ▪ Identifying the requirements (ie, selecting an ADT)
   ▪ Making tradeoffs (ie, selecting the data structure for that ADT)

❖ So let’s review the ADTs’ functionality and the performance characteristics of each data structure
Intuitively ...

❖ Think of the ADTs and data structures you’ve learned this quarter as a cookbook

▪ ADTs are the chapters/category: Soups, Salads, Cookies, Cakes, etc
  • High-level descriptions of a category of functionality
  • You don’t serve a soup when guests expect a cookie!

▪ Data structures (and algorithms) are the recipes: chocolate chip cookies, snickerdoodles, etc
  • Step-by-step, concrete descriptions of an item with specific characteristics
  • Understand your tradeoffs before replacing carrot cake with a wedding cake

❖ When you go out into the world, your two key skills are:

▪ Figure out which category is required
▪ Choose the specific recipe within that category which best fits the situation
Lecture Outline

❖ ADT and Data Structure Review

❖ Algorithms Review
How to Review These Structures

❖ For each ADT:
  ▪ What behavior does the ADT actually allow?
  ▪ What is unique about this ADT?

❖ For each data structure:
  ▪ How easy is it to implement?
  ▪ What is the runtime for each of its core operations?
  ▪ What is its memory utilization?
List Functionality

**List ADT.** A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index.
- A list has a size defined as the number of elements in the list.
- Elements can be added to the front, back, *or any index in the list.*
- Optionally, elements can be removed from the front, back, *or any index in the list.*

- Possible Implementations:
  - ArrayList
  - LinkedList
## List Performance Tradeoffs

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFront</td>
<td>linear</td>
<td>constant</td>
</tr>
<tr>
<td>removeFront</td>
<td>linear</td>
<td>constant</td>
</tr>
<tr>
<td>addBack</td>
<td>constant*</td>
<td>linear</td>
</tr>
<tr>
<td>removeBack</td>
<td>constant</td>
<td>linear</td>
</tr>
<tr>
<td>get(idx)</td>
<td>const</td>
<td>linear</td>
</tr>
<tr>
<td>put(idx)</td>
<td>linear</td>
<td>linear</td>
</tr>
</tbody>
</table>

* constant for most invocations
Stack and Queue Functionality

**Stack ADT.** A collection storing an ordered sequence of elements.
- A stack has a size defined as the number of elements in the stack.
- Elements can only be added and removed from the top ("LIFO")

- Possible Implementations:
  - ArrayStack, LinkedStack

**Queue ADT.** A collection storing an ordered sequence of elements.
- A queue has a size defined as the number of elements in the queue.
- Elements can only be added to one end and removed from the other ("FIFO")

- Possible Implementations:
  - ArrayQueue, LinkedQueue
# Stack and Queue Performance Tradeoffs

- **Stack (LIFO):**

<table>
<thead>
<tr>
<th></th>
<th>ArrayStack</th>
<th>LinkedStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>constant*</td>
<td>constant</td>
</tr>
<tr>
<td>pop</td>
<td>constant</td>
<td>constant</td>
</tr>
</tbody>
</table>

* constant for most invocations

- **Queue (FIFO):**

<table>
<thead>
<tr>
<th></th>
<th>Array Queue (v2)</th>
<th>LinkedQueue (v2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>constant*</td>
<td>constant</td>
</tr>
<tr>
<td>dequeue</td>
<td>constant</td>
<td>constant</td>
</tr>
</tbody>
</table>

* constant for most invocations
Deque Functionality

Deque ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index.
- A deque has a size defined as the number of elements in the deque.
- Elements can be added to the front or back.
- Optionally, elements can be removed from the front or back.

Possible Implementations:

- ArrayDeque, LinkedDeque
## Deque Performance Tradeoffs

<table>
<thead>
<tr>
<th>Method</th>
<th>CircularArrayDeque</th>
<th>LinkedDeque</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFirst</td>
<td>constant*</td>
<td>constant</td>
</tr>
<tr>
<td>removeFirst</td>
<td>constant</td>
<td>constant</td>
</tr>
<tr>
<td>addLast</td>
<td>constant*</td>
<td>constant</td>
</tr>
<tr>
<td>removeLast</td>
<td>constant</td>
<td>constant</td>
</tr>
</tbody>
</table>

* constant for most invocations
Set and Map Functionality

**Set ADT.** A collection of values.
- A set has a size defined as the number of elements in the set.
- You can add and remove values.
- Each value is accessible via a “get” or “contains” operation.

**Map ADT.** A collection of keys, each associated with a value.
- A map has a size defined as the number of elements in the map.
- You can add and remove (key, value) pairs.
- Each value is accessible by its key via a “get” or “contains” operation.

❖ **Possible Implementations:**
- Unbalanced BST
- LLRB Tree
- B-Tree (eg, 2-3 Tree)
- Hash Tables
- Tries
## Set and Map Performance Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th>Find</th>
<th>Add</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resizable Separate Chaining Hash Table</strong> <em>(worst case)</em></td>
<td>$Q \in \Theta(N)$</td>
<td>$Q \in \Theta(N)$</td>
<td>$Q \in \Theta(N)$</td>
</tr>
<tr>
<td><strong>Resizable Separate Chaining Hash Table</strong> <em>(best/average cases)</em></td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)^*$</td>
<td>$\Theta(1)^*$</td>
</tr>
<tr>
<td>LLRB Tree</td>
<td>$h \in \Theta(\log N)$</td>
<td>$h \in \Theta(\log N)$</td>
<td>$h \in \Theta(\log N)$</td>
</tr>
<tr>
<td>B-Tree</td>
<td>$h \in \Theta(\log N)$</td>
<td>$h \in \Theta(\log N)$</td>
<td>$h \in \Theta(\log N)$</td>
</tr>
<tr>
<td>BST</td>
<td>$h \in \Theta(N)$</td>
<td>$h \in \Theta(N)$</td>
<td>$h \in \Theta(N)$</td>
</tr>
<tr>
<td>LinkedList</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>Trie</td>
<td>$\Theta(1)^*$</td>
<td>$\Theta(1)^*$</td>
<td>$\Theta(1)^*$</td>
</tr>
</tbody>
</table>
Priority Queue Functionality

Priority Queue ADT. A collection of values.
• A PQ has a size defined as the number of elements in the set.
• You can add values.
• You cannot access or remove arbitrary values, only the max value.

❖ Possible Implementations:
  ▪ Balanced BST with “max” pointer
  ▪ Binary Heap
  ▪ (and a ton of others we didn’t discuss)

❖ Don’t forget you also know Floyd’s buildHeap!
### Priority Queue Performance Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th>Balanced BST (worst case)</th>
<th>Binary Heap (worst case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$O(\log N)$</td>
<td>$O(\log N)**$</td>
</tr>
<tr>
<td>max</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeMax</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

* If we keep a pointer to the largest element in the BST  
** Average case is constant
Multidimensional Data

Key Operations:
- Range Searching: What are all the objects inside this (rectangular) region?
- Nearest Neighbour: What is the closest object to a specific point (this is often the k-nearest in machine learning)

Spatial Partitioning: Dividing space into non-overlapping subspaces, allowing us to prune the search space
- Uniform partitioning
- Quadtree
- k-d Tree
Graph Functionality

**Graph ADT.** A collection of vertices and the edges connecting them.

- We can query for vertices connected to, or edges leaving from, a vertex v
- Edges are specified as pairs of vertices
- We can add/remove edges from the graph

Possible Implementations:
- Adjacency Matrix
- Edge Set
- Adjacency List
**Graph Performance Tradeoffs**

<table>
<thead>
<tr>
<th></th>
<th>getAllEdgesFrom(v)</th>
<th>hasEdge(v, w)</th>
<th>getAllEdges()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency Matrix</td>
<td>$\Theta(V)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(V^2)$</td>
</tr>
<tr>
<td>Edge Set</td>
<td>$\Theta(E)$</td>
<td>$\Theta(E)$</td>
<td>$\Theta(E)$</td>
</tr>
<tr>
<td>Adjacency List</td>
<td>$O(V)$</td>
<td>$\Theta(\text{degree}(v))$</td>
<td>$\Theta(E + V)$</td>
</tr>
</tbody>
</table>
Disjoint Sets ADT

Disjoint Sets ADT. A collection of elements and sets of those elements.

- An element can only belong to a single set.
- Each set is identified by a unique id.
- Sets can be combined/connected/unioned.

Possible Implementations:
- WeightedQuickUnion
- WeightedQuickUnion with Path Compression
## Disjoint Sets Performance Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th>find</th>
<th>union excludes find(s)</th>
<th>union includes find(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickFind</td>
<td>$\Theta(1)$</td>
<td>$\Theta(N)$</td>
<td>N/A</td>
</tr>
<tr>
<td>QuickUnion</td>
<td>$h \in O(N)$</td>
<td>$\Theta(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>WeightedQuickUnion</td>
<td>$h \in \Theta(\log N)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(\log N)$</td>
</tr>
<tr>
<td>WQU + Path Compression</td>
<td>$h \in O(1)^*$</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
</tr>
</tbody>
</table>

* amortized
Lecture Outline

❖ ADT and Data Structure Review

❖ Algorithms Review
tl;dr

- Dijkstra’s is great for \textit{all-pairs shortest path}

- A* is great for \textit{single-pair shortest path}
  - But you need to be careful about picking a good heuristic
How to Review These Algorithms

❖ For each algorithm, which situations apply?
  ▪ If we used this algorithm, what do we learn about our data?
  ▪ In what state does the data need to be in, if we wanted to use it?

❖ For each algorithm, what are the pros/cons of ...
  ▪ Its ease of implementation?
  ▪ Its time complexity?
  ▪ Its space complexity?
Graph Algorithms

❖ Graphs Traversals:
  ▪ BFS
  ▪ DFS
    • Pre- and Post-Order Traversals
    • For trees, also add In-order Traversals

❖ Shortest Paths:
  ▪ Dijkstra’s
  ▪ A* Search

❖ Minimum Spanning Trees
  ▪ Prim’s
  ▪ Kruskal’s

❖ Topological Sort
Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best-Case Time</th>
<th>Worst-Case Time</th>
<th>Space</th>
<th>Stable?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionSort</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(1))</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>In-Place HeapSort</td>
<td>(\Theta(N))</td>
<td>(\Theta(N \log N))</td>
<td>(\Theta(1))</td>
<td>No</td>
<td>Slow in practice</td>
</tr>
<tr>
<td>MergeSort</td>
<td>(\Theta(N \log N))</td>
<td>(\Theta(N \log N))</td>
<td>(\Theta(N))</td>
<td>Yes</td>
<td>Fastest stable sort</td>
</tr>
<tr>
<td>In-Place InsertionSort</td>
<td>(\Theta(N))</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(1))</td>
<td>Yes</td>
<td>Best for small or partially-sorted input</td>
</tr>
<tr>
<td>Naïve QuickSort</td>
<td>(\Theta(N \log N))</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(N))</td>
<td>Yes</td>
<td>(&gt;=2x) slower than MergeSort</td>
</tr>
<tr>
<td>Dual-Pivot QuickSort</td>
<td>(\Omega(N))</td>
<td>(O(N^2))</td>
<td>(\Theta(1))</td>
<td>No</td>
<td>Fastest comparison sort</td>
</tr>
</tbody>
</table>
## RadixSorts

<table>
<thead>
<tr>
<th></th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountingSort</td>
<td>$\Theta(N+R)$</td>
<td>$\Theta(N+R)$</td>
</tr>
<tr>
<td>LSD RadixSort</td>
<td>$\Theta(LN + LR)$</td>
<td>$\Theta(N + R)$</td>
</tr>
</tbody>
</table>
| MSD RadixSort    | Best: $\Theta(N + R)$  
Worst: $\Theta(LN + LR)$ | $\Theta(N + LR)$ |
tl;dr

- THANK YOU for a wonderful quarter!