Course Wrapup

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

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

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
- Lecture eval: https://uw.iasystem.org/survey/219337

Announcements

- Section Evals:
  - AA: https://uw.iasystem.org/survey/221482
  - AB: https://uw.iasystem.org/survey/221455
  - AC: https://uw.iasystem.org/survey/221537
  - AD: TBD
  - AE: https://uw.iasystem.org/survey/221470
  - AF: https://uw.iasystem.org/survey/221507
  - AG: https://uw.iasystem.org/survey/221496
  - AH: 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

	ArrayList	LinkedList
addFront	linear	constant
removeFront	linear	constant
addBack	constant*	linear
removeBack	constant	linear
get(idx)	const	linear
put(idx)	linear	linear

* 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):

	ArrayStack	LinkedStack
push	constant*	constant
pop	constant	constant

* constant for most invocations

- Queue (FIFO):

	Array Queue (v2)	LinkedQueue (v2)
enqueue	constant*	constant
dequeue	constant	constant

* 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

	CircularArrayDeque	LinkedDeque
addFirst	constant*	constant
removeFirst	constant	constant
addLast	constant*	constant
removeLast	constant	constant

* 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

	Find	Add	Remove
Resizing Separate Chaining Hash Table *(worst case)*	$Q \in \Theta(N)$	$Q \in \Theta(N)$	$Q \in \Theta(N)$
Resizing Separate Chaining Hash Table *(best/average cases)* +	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)^*$
LLRB Tree	$h \in \Theta(\log N)$	$h \in \Theta(\log N)$	$h \in \Theta(\log N)$
B-Tree	$h \in \Theta(\log N)$	$h \in \Theta(\log N)$	$h \in \Theta(\log N)$
BST	$h \in \Theta(N)$	$h \in \Theta(N)$	$h \in \Theta(N)$
LinkedList	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$
Trie	$\Theta(1)^*$	$\Theta(1)^*$	$\Theta(1)^*$

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

	Balanced BST (worst case)	Binary Heap (worst case)
add	O(log N)	O(log N)**
max	O(1)*	O(1)
removeMax	O(log N)	O(log N)

** 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 top 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

	getAllEdgesFrom(v)	hasEdge(v, w)	getAllEdges()
Adjacency Matrix	$\Theta(V)$	$\Theta(1)$	$\Theta(V^2)$
Edge Set	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$
Adjacency List	$O(V)$	$\Theta(\text{degree}(v))$	$\Theta(E + V)$

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

	find	union *excludes find(s)*	union *includes find(s)*
QuickFind	Θ(1)	Θ(N)	N/A
QuickUnion	h ∈ O(N)	Θ(1)	O(N)
WeightedQuickUnion	h ∈ Θ(log N)	Θ(1)	Θ(log N)
WQU + Path Compression	h ∈ O(1)*	O(1)*	O(1)*

* amortized

Lecture Outline

- ADT and Data Structure Review
- **Algorithms Review**

tl;dr

- Dijkstra's is great for *all-pairs shortest path*

- A* is great for *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 ue 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

	Best-Case Time	Worst-Case Time	Space	Stable?	Notes
SelectionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	No	
In-Place HeapSort	$\Theta(N)$	$\Theta(N \log N)$	$\Theta(1)$	No	Slow in practice
MergeSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Yes	Fastest stable sort
In-Place InsertionSort	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Yes	Best for small or partially-sorted input
Naïve QuickSort	$\Theta(N \log N)$	$\Theta(N^2)$	$\Theta(N)$	Yes	>=2x slower than MergeSort
Dual-Pivot QuickSort	$\Omega(N)$	$O(N^2)$	$\Theta(1)$	No	Fastest comparison sort

RadixSorts

	Time Complexity	Space Complexity
CountingSort	Θ(N+R)	Θ(N+R)
LSD RadixSort	Θ(LN + LR)	Θ(N + R)
MSD RadixSort	Best: Θ(N + R) Worst: Θ(LN + LR)	Θ(N + LR)

tl;dr

- THANK YOU for a wonderful quarter!