Final Review Sheet

CSE 326: Data Structures
Autumn 2006

The final

In class, Wednesday, 12/13/06, 2:30-4:20 PM

Syllabus

§ Everything covered in the course
§ Less emphasis on material covered before the midterm
§ Closed book, closed notes!

Reading list from the textbook (for topics covered after the midterm)

§ Disjoint Sets Chapter 8
   Excluding 8.6.1

§ Sorting Chapter 7
   Excluding 7.4, average-case Q Sort anal, 7.10.5 onwards

§ Graphs Chapter 9
   Excluding 9.3.3, 9.3.4, 9.4, 9.6 onwards

§ Amortized analysis Chapter 11
   Excluding 11.4

Disjoint Set ADT

§ Operations: Find(x), Union(A,B), MakeNewSet(x)

§ Application to Maze construction

§ Can’t have an implementation that guarantees \( \Theta(1) \) worst-case time for both find() and union(); so we shoot for \( \Theta(1) \) amortized-case time

§ Union-find data structure: forest of up-trees, nifty array storage
- Union-by-size
  (union-by-height was on homework #3; what’s a potential implementation problem with union-by-height if you also want to use path compression?)

- Path compression

- Two really slow growing functions: $\log^* n$, inverse Ackermann’s function

- Analysis not covered in class

**Sorting: (A) Comparison-based**

- Worst-, best-, average-case bounds for all sorting algorithms

- $\Theta(n^2)$ sorts: Insertion sort, Selection sort
  - Simple to implement
  - Less overhead: useful when $n$ is small
  - Worst-, best- and average-case runtime?

- $\Theta(n \log n)$ sorts:
  - Using data structures we have learned: Heap sort, AVL sort (or tree sort of some kind); bounds follow from data structure analysis
  - Divide-and-conquer techniques: Merge sort, Quick sort
    We did not prove average-case bound for Quick sort in class

**Sorting: (B) In $\Theta(n)$ time**

- Bucket sort
  - Useful when the numbers are known to be in a small range, 1 to K

- Radix sort
  - Break-up the rage into smaller chunks
  - Sort from least significant to most significant using some stable sort

**Sorting: (C) External**

- Useful when too many numbers to fit in memory

- External device model
- Stage 1: sort chunks that will fit into memory
- Stage 2: repeatedly merge, switching between devices

**Sorting: (D) Lower Bounds**

- Flavors of lower bounds
  1. for an algorithm or operation on a structure
  2. for a problem
  3. for a class of algorithms for a problem

- Bound #1: Sorting by exchanging adjacent elements: $\Omega(n^2)$
  - Proof based on counting number of inversions

- Bound #2: Sorting by comparisons: $\Omega(n \log n)$
  - Proof based on decision trees

**Graphs: (A) Basics**

- Kinds: (un)directed, (un)weighted, (a)cyclic, (un)connected
- Representations: Adjacency Matrix, Adjacency List
- Natural problems with applications: Shortest path, minimal spanning network, strong connectivity, orderings, dependency graphs
- Traversals: DFS, BFS, Best-first, Topological sort order

**Graphs: (B) Shortest path algorithms**

- Problem flavors: Shortest path from s to t vs. SSSP vs. APSP
- Unweighted: BFS
- Weighted: Dijkstra’s algorithm (greedy)
  - Table of known/unknown and current cost
  - What more do you need to maintain to output path at the end?
  - Inductive proof of correctness
Negative-cost cycles: problem!

Negative-cost edges but no negative-cost cycles: mentioned in Homework #3

**Graphs: (C) Minimum spanning tree**

- Different problem than shortest paths
- Prim’s algorithm: similar to Dijkstra’s algorithm
- Kruskal’s algorithm: uses disjoint set ADT, also greedy

**Amortized analysis**

- General technique
  - Introduce Potential function such that actual time plus change in potential function doesn’t vary much over successive operations
  - \( T_{\text{actual}} + \Delta \text{Potential} = T_{\text{amortized}} \)
  - Do a telescopic sum. If net change in potential is non-negative, then sum of assumed amortized times is an upper bound on the sum of actual times

- Binomial Queue analysis: `buildBQ(n)` takes amortized time \( \Theta(n) \)
  - \( T_{\text{actual}} = C_i = \text{cost of } i^{\text{th}} \text{ insert} \)
  - Potential = \( T_i = \text{number of trees after the } i^{\text{th}} \text{ insert} \)

- Skew heap analysis: `merge()` takes amortized time \( \Theta(\log n) \)
  - Define heavy and light nodes
  - \( T_{\text{actual}} = \text{sum of lengths of right paths} \)
  - Potential = number of heavy nodes in the two trees

**Compression**

- Motivation and basics
§ Lossy vs Lossless compression

§ Huffman Trees
  o Structure
  o Decoding
  o Construction Algorithm
Topics Covered Before the Midterm

(See Midterm Review Sheet for more details)

Introduction

• Concepts vs. Mechanisms

• All Data Structures we have seen can implement all ADTs we have seen. However, they differ in efficiency.

• Simple ADTs: List, Stack, Queue

Algorithm Analysis

• Asymptotic complexity

• Two orthogonal axes:
  1. worst-case, best-case, average-case, amortized
  2. upper bound (O or o), lower bound (Ω or ω), tight bound (Θ)

• Big-Oh notation

• Proofs of correctness or complexity bounds

Priority Queue ADT

• Characterized by deleteMin() operation; usually inefficient for find(k)

• Useful for greedy applications

• Implementations include

  1. Simple stuff: array, linked lists (sorted or unsorted)
  2. Binary heap
  3. Leftist heap
  4. Skew heap
  5. Binomial Queues
6. \textit{d}-heap

\textbf{Search ADT / Dictionary ADT}

- Characterized by find($k$), insert($k$), delete($k$)
- Useful for search based applications
- Also useful for sorting based applications unless the data structure used is a hash table like structure that doesn’t organize data using ordering information
- Implementations include
  1. Simple stuff: array, linked lists (sorted or unsorted)
  2. Binary Search Tree (unbalanced)
  3. AVL Tree
  4. Splay Tree
  5. B-trees (2-3 trees, 2-3-4 trees)
  6. Hash table
     \begin{itemize}
     \item Separated chaining
     \item Open addressing
     \item Rehashing: can be used with separate chaining or open addr
     \item Extendible hashing
     \end{itemize}