Review for Exam

1 Topics

In this course we have covered a number of different data structures. The fundamental concept behind all data structures is that of arranging data in a way that permits a given set of queries or accesses to be performed efficiently. The aim has been at describing the general principles that lead to good data structures as opposed to giving a "cookbook" of standard techniques. In addition we have discussed some of the mathematics needed to prove properties and analyze the performance of these data structures (both theoretically and empirically). Also, we have covered some major techniques for algorithm development (greedy, divide-and-conquers, dynamic programming, backtracking, branch-and bound etc.) These are the main topics that have been covered:

Basics : Asymptotic notation, induction proofs, summations and recurrences.

Lists, stacks, queues : Representations, traversals, implementations, performance.

Trees : General Trees, binary representation of trees, standard traversals (pre-, in-, postorder).

Search Trees : Binary search trees, Optimal trees, AVL trees (guarantee $O(\log n)$ time for insertions, deletions and finds by height balancing. Use single and double rotations to restore balance).

Hashing : Hashing is considered the simplest and most efficient technique (from a practical perspective) for performing dictionary operations. The operations Insert, Delete, and Find can be performed in $O(1)$ expected time. The main issues here are designing a good hash function which distributes the keys in a nearly random manner, and a good collision resolution method to handle the situation when keys hash to the same location. Hashing will not replace trees, because trees provide so many additional capabilities (e.g. find all the keys in some range).

Priority Queues : We discussed binary heaps. Binary heaps supported the operations of Insert and Delete-Min. The most important aspect of binary heaps is the fact that since the tree in which they are stored is so regular, we can store the tree in an array and do not need pointers.

Disjoint Set Union/Find : We discussed a data structure for maintaining partitions of sets. This data structure can support the operations of merging two sets together (Union) and determining which set contains a specific element (Find).

Directed/undirected Graphs : Representations. Shortest paths, Dijkstra's Algorithm: Single source, nonnegative edge weights. Running time $O(E \log V)$. Bellman-Ford Algorithm: Single source, arbitrary edge weights (no negative cost cycles). Running time $O(VE)$. Floyd-Warshall Algorithm: All-pairs shortest paths, arbitrary edge weights (no negative cost cycles). Running time $O(V^3)$. Transitive closure. Topological sort. Strong components. Traversals: depth-first, breadth-first. Minimum spanning trees – Prim's algorithm, Kruskal's algorithm. Articulation points and graph matching.

Algorithm analysis : Correctness and performance of algorithms (asymptotic).

Algorithm Design Techniques : Divide-and-Conquer, Dynamic Programming, Brute-force, Greedy, Backtracking, Minimax and Alpha-beta Prunning, Branch-and-Bound, local search.

Sorting : simple sorting schemes (bubblesort, insertion sort, selection sort, counting sort); heapsort, quicksort, radix sort, bucket sort – algorithms and performance. Criteria for selecting a sort algorithm.

Data Structures for External Storage : sequential files, sorting files(merge Sort, m-way sort, polyphase sort). Alternative file organizations. Indexes: m-way search trees, B trees and B^+ trees (shallow, generalizing 2-3 trees (discussed in lecture 4), used for database indexing).

2 How to use this information

Before you design any data structure or algorithm, first isolate the key mathematical elements of the task at hand – i.e. the **mathematical model**. Identify what the mathematical objects are, and what operations you are performing on them. (E.g. Dictionary: insertion, deletion, and finding of numeric keys.)

Recursive Subdivision : A large number of data structures (particularly tree based struc- tures) have been defined recursively, by splitting the underlying domain using a key value. The algorithms that deal with them are often most cleanly conceptualized in recursive form.

(Strong) Induction : In order to prove properties of recursively defined objects, like trees, the most natural mathematical technique is induction. In particular, strong induction is important when dealing with trees.

Balance : The fundamental property on which virtually all good data structures rely is a notion of information balancing. Balance is achieved in many ways (rebalancing by rotations, using hash functions to distribute information randomly, balanced merging in UNION-FIND trees).

Asymptotic analysis : Before fine-tuning your algorithm or data structure's performance, first be sure that your basic design is a good one. A half-hour spent solving a recurrence to analyze a data structure may tell you more than 2 days spent prototyping and profiling.

Empirical analysis : Asymptotics are not the only answer. The bottom line: the algorithms have to perform fast on my data for my application. Prototyping and measuring performance parameters can help uncover hidden inefficiencies in your data structure.

Build up good tools : Modern languages like C++ help you to build up classes of data structures and objects.

In some sense, the algorithms you have learned here are hardly ever immediately applicable to your later work (unless you go on to be an algorithm designer) because real world problems are always messier than these simple abstract problems. However, there are some important lessons to take out of this class.

Develop a clean mathematical model : Most real-world problems are messy. An important first step in solving any problem is to produce a simple and clean mathematical formulation. For example, this might involve describing the problem as an optimization problem on graphs, sets, or strings.

Create good rough designs : Before jumping in and starting coding, it is important to begin with a good rough design. If your rough design is based on a bad paradigm (e.g. exhaustive enumeration, when depth-first search could have been applied) then no amount of additional tuning and refining will save this bad design.

Prove your algorithm correct : Many times you come up with an idea that seems promising, only to find out later that it does not work. Try to prove that your algorithm is correct. If you cannot see why it is correct, chances are that it is not correct at all. Use general paradigms in designing algorithms: Rather than designing an algorithm from scratch ask yourself whether the design paradigms you know can be used to solve your problem. Is there a divide-and-conquer solution? greedy solution? etc.

Can it be improved? : Once you have a solution, try to come up with a better one. Is there some reason why a better solution does not exist? If your solution is exponential time, then maybe your problem is NP-hard.

Prototype to generate better designs : We have attempted to analyze algorithms from an asymptotic perspective, which hides many of details of the running time (e.g. constant factors), but give a general perspective for separating good designs from bad ones. After you have isolated the good designs, then it is time to start prototyping and doing empirical tests to establish the real constant factors.

Still too slow? : If your problem has an unacceptably high execution time, you might consider an approximation algorithm. The world is full of heuristics, good and bad. You should develop a good heuristic, and if possible, prove a ratio bound for your algorithm. If you cannot prove a ratio bound, run many experiments to see how good the actual performance is.

There is still much more to be learned about algorithm design, but we have covered a great deal of the basic material. One direction is to specialize in some particular area, e.g. string pattern matching, computational geometry, parallel algorithms, randomized algorithms, or approximation algorithms. It would be easy to devote an entire semester to any one of these topics. Another direction is to gain a better understanding of average-case analysis, which we have largely ignored. Still another direction might be to study numerical algorithms (as covered in a course on numerical analysis), or to consider general search strategies such as simulated annealing. Finally, an emerging area is the study of algorithm engineering, which considers how to design algorithms that are both efficient in a practical sense, as well as a theoretical sense.

3 What is tested in the exam

- Knowledge of data structures(e.g. what is a complete binary search tree; give an example; show the result of inserting given values into the given tree,...)
- Knowledge of algorithms (e.g. give pseudocode (or C code) of the insertion sort)

- Understanding big-Oh notation(e.g. what is the time complexity of this algorithm)
- Given a problem, suggest which algorithms and data structures are appropriate for solving it
- In the practical part, also implement and test your algorithm

4 How to revise

- Straightforward knowledge questions: lecture notes, course handouts and any recommended textbook
- Choosing appropriate data structures and algorithms: use knowledge and other properties (dynamic vs. static) of different data structures
- For every algorithm explained, practice training it on some example (Draw a graph and do Dijkstra's algorithm on it. Draw a B+tree and add some new elements on it)
- Browse your assignment programs.

