COMS E6998-8: Advanced Data Structures (Spring'19) Jan 24, 2019

Lecture 1: Models of Data Structures

Instructor: *Omri Weinstein* Scribes: *Sihyun Lee*

1 Course Overview

The study of data structures is one of the earliest disciplines in computer science. Since every implementation of an algorithm depends on the data structure they operate on, understanding data structures is even more fundamental than studying algorithms. This course aims to understand limits and strengths of various data structures.

The course will focus on four main types of data: integers, geometry, graphs, and strings. The three main themes throught the course would be upper bounds, lower bounds, and computational models. We will explore computational models on data structures in a complexity theoretic viewpoint. However, there are some problems that have information-theoretic lower bounds that do not rely on assumptions in complexity theory (such as P≠NP); there are some interesting breakthrough for these problems that set a low upper bound. While many problems have gaps between upper and lower bounds, some problems to be covered in class have them asymptotically equal.

2 Types of Data Structures

We focus two main types of data structures: *static* and *dynamic*. We briefly mention more special types of data structures: *offline* and *temporal*.

• **Static Data Structures** are data structures that are given all the data in advance. It assumes that all the data given to it will not be modified. Our goal is to preprocess data efficiently so that certain queries about the given dataset could be answered quickly. For these data structures, we are concerned most about the trade-off between *space* and *time* complexity.

Example 1: Graphs. We consider graphs, more specifically, road networks of n vertices and m edges. Given two points in the network, we want to preprocess data so that we can efficiently compute the distance between two points when queried. One way of storing data is to precompute all answers. This way, we can answer queries in $O(1)$ time, but to store all of the pairwise distances we will need $O(n^2)$ space. Another way is to store the entire graph. This way, we only need $O(m+n)$ space, but answering each query takes $O(m + n \log n)$ time using Dijkstra's Algorithm.

Example 2: Geometric Data Structures. The first problem we discuss is *nearest neighbor search*. Say that we are given n points $X = \{x_1, ...x_n\}$ in $\mathbb{R}^d$. Our goal is to preprocess the data into small memory so that, given a query point q, the closest point in X to q. The naive way to solve this is to store the answers for all choices of q. There are infinitely many choices, so we instead divide the entire

space into "balls" of side length r, and this approach will require $O((1/r)^d)$ space and suffers a "curse of dimensionality". Another approach is to store all points into a database and scan the entire database on each query, which takes $O(n)$ time and might not be scalable when there are large amounts of data. We will see that, *by allowing approximations*, this problem can be tackled in linear space and $O(\sqrt{n})$ time.

We will also discuss a similar problem called *orthogonal range queries*, which is similar to the above but we specify intervals in each dimension and want to estimate how much data falls into the "box" defined by the intervals. This problem turns out to be more tractable than the nearest neighbors problem because it is easier to deal with "boxes" than "balls". Problems like this has practical applications in estimating a distribution. For example, Amazon uses these types of techniques to estimate market sizes of certain segments.

• **Dynamic Data Structures** are data structures that evolve over time. The input is not fixed in advance and the data structure should allow modification (i.e. update) of the data. For dynamic data structures, we are interested in the trade-off between the *update time* and the *query time*.

Remark. Space is less of a concern, since it is already captured by the update time: if the maximum update time for a data structure is t_u we can bound the space s used by the data with $s \leq n \cdot t_u$. (Later on, when we cover data structures for strings, we will consider all three since there is one that achieves surprisingly high efficiency in all aspects.)

Obviously, dynamic data structures are much more applicable, while static data structures are easier to come up with. We will try to *dynamize* static data structures. More specifically, we will discuss *black-box* dynamization, where we provide a general framework for dynamizing without knowing the exact details of the static data structure.

Example 3: Dynamic Prefix-Sums. Say that we run a company database that stores the salary and hire year of all of its employees. On a query, we receive a year t and return the total salary of people hired on or before t. Every day people get hired, fired, or get their pay changed, so we need a dynamic data structure.

The naive solution is to group all the employees based on their hire year, and list the groups in an array. When a query is made, we iterate through the array, taking the sum within each group. Here, update takes worst case $O(n)$ time, where n is the number of years concerned.

Another solution is to create a self-balancing binary tree (e.g. a red-black tree) where the key to each node is the hire year, and each node is augmented with the list of people hired that year, the sum of their salaries, and the sum of salaries in all years on the subtree rooted on it. The update time is $O(\log n)$ since we have to traverse the path from the root to the target node and update all nodes in the path.

Given a query year t, we can easily find the sum of salaries including and before recursively: starting from the root, if the root has key greater than t, we can ignore the root and the right subtree, and we just recursively query the left subtree. Otherwise, we can return the sum of (1) the sum of salaries in the left subtree, (2) the salary of the root node, and (3) the sum of salaries in the right subtree, recursively found.

Example 4: Optimality of Prefix-Sums. In the above example, we proposed a $O(\log n)$ update time, $O(\log n)$ query time algorithm for the problem. We will soon prove that we cannot improve on this result for a 1-dimensional prefix-sum problem (where we have only one dimension that determines whether a person's salary should count or not).

What if we have to keep track of both the hire year and the birth year, and our queries ask for a sum over intervals in both information? This is a 2-dimensional prefix-sum problem, and a similar bound has been shown. However, for 3 or more dimensions, there is no result proven to be tight.

• **Other Data Structures.** An *offline* (or *batch*) data structure is a static data structure but without a preprocessing step. The online data structure mentioned above takes input step-by-step, while an offline data structure can access the entire dataset before preprocessing. Therefore, it is stronger than a static data structure; however, it is unable to make updates outside the originally given data, so it is weaker than dynamic data structures.

Temporal data structures are ones that can "time travel": a user can make queries about specific past versions of the data. Since the requirement is very demanding, not much is known about them.

3 Computational Models of Data Structures

The same way we had a discussion on Turing Machines and their variants for an evaluation of computational complexity, it is essential for us to define the exact computational models regarding data structures to make well-defined statements about time and space bounds. The conversation on different computational models of data structures have been inspired by the development of hardware and memory architectures.

• **Transdichotomous RAM Model.** Here, the memory address is considered a finite array divided into w-bit words; this model assumes that $w \geq \min\{\lg s, \lg n\}$ where s is the total memory space used and n is the amount of data that is input. This assumption enables us to index all points of the input data and the memory space.

• **Word-RAM Model.** This is the typical model for proving upper bounds and considered a realistic representation of modern computer architecture. The characteristics of the Transdichotomous RAM Model carries on, yet on each of these words, arithmetic operations such as $+, -, \times, \div$ and logical operations such as $|, \&, \hat{}$, bit-shifting, and comparing are considered constant-time operations. Depending on which operations are considered constant-time, there may be similar variants to this model. An example is the *arithmetic RAM model* where multiplication is not considered a constant-time operation since it is obviously more complicated than addition.

• **Pointer-Machine Model.** Here, a data structure is consisted of nodes with a constant fan-out of pointers. All points in memory can be accessed only by following a list of pointers. Since there is no random-access memory given here, this is considered a weaker model than any RAM Model. Non-array types of data structures that we saw in a typical data structures class fall under this category: linked lists, heaps, binary search trees. The next two lectures will cover more advanced tree models and some versions of pointer machines.

The pointer-machine model is weaker than the Word-RAM model, so when we are showing upper bounds for data structures they can be useful. Can we introduce a stronger model, so that proving a lower bound in this model in turn proves one for computers universally? We use the fact that, in any computer architecture, the CPU and memory are isolated and it takes time for the CPU to access memory.

• **Cell-Probe Model.** Here, we completely ignore the time it takes for the CPU to perform operations on memory it has already loaded. The number of loading requests is the only computational cost we consider. Formally, a static (s, t)-data structure in the Cell-Probe model defined by the following: the input data is preprocessed into a space of s different cells (i.e. words). In every query, we can load up to t words and announce the answer based on the loaded cells. For dynamic data structures, we can define a dynamic (t_u, t_q)-data structure. In every query, we can load t_q cells; in every update, we can load t_u cells. Our objective here is to find upper and lower bounds for $\max\{t_u, t_q\}$. The cell-probe model will be used in proving information-theoretic lower bounds for computation.

4 Performance Guarantees

We finally discuss some ways the performance of a data structure is evaluated. The standard guarantee we saw in an introductory data structures class is **worst-case**: we assume that an adversary can choose an input that maximizes the running time of our algorithm. However, this is sometimes not very useful when worst-case instances do not appear too often in practice.

A useful guarantee is **amortized** runtime. Here, we consider the *total* number of operations for a *sequence* of operations. A classic example is the binary counter. The worst-case runtime of incrementing a counter by 1 is $O(n)$, because if our number is all 1's we have to change all the bits except the leading 1. However, it can be shown that to increment from 0 to n, we make a total of at most $2n$ changes to the individual bits. Since n increments can be done within $2n$ operations, we can say that adding 1 to a counter is a constant-time operation. (Details can be found in Chapter 17.1 in *Introduction to Algorithms* by Cormen et al.) Amortized analysis will be a core technique for analyzing algorithms used in this course.

Another useful guarantee is **expected** runtime. This is different from amortized analysis: amortized analysis takes the average over a *sequence* of operations for a deterministic algorithm, while expected analysis takes the average over a *distribution* of random choices that a randomized algorithm makes. For example, *randomized* quicksort makes $O(n \log n)$ operations in expectation, while non-randomized quicksort makes $O(n^2)$ operations in the worst case.

We also consider **instance-optimal** data structures. The exact notion of instance-optimality will be covered later, but the intuition is that the asymptotic runtime is uniform for *any* input. For example, it is unreal to achieve instance-optimality for sorting since if we have an already sorted array it takes $O(n)$ time to check, yet any comparison-based sorting algorithm should have average-case runtime $\Omega(n \log n)$. In the next two lectures, we give examples of instance-optimal trees.

5 Advanced BSTs and Dynamic Optimality

This section is a preview of what we will see in the next two weeks. The motivating question is to find an instance-optimal integer binary search tree (BST). First, we set the **predecessor search problem** as the key problem for BSTs: given a query integer x, we want to return the greatest integer in the data structure not greater than x.

The easy way to start is to use a balanced BST. This is an efficient data structure with update and query time both $O(\log n)$. However, this is not instance-optimal: consider an extreme sequence of queries

that repeatedly sends the number in the root of the tree. We can make an adversarial argument that, in terms of worst-case analysis, we cannot make a tree that is instance-optimal compared to this sequence of queries. In the next lecture we cover Splay Trees, which is conjectured to have achieved instance optimality. The idea of this tree is to modify the tree based on the queries made.