CMSC 420: Lecture 1
Course Introduction and Background

Algorithms and Data Structures: The study of data structures and the algorithms that manipulate them is among the most fundamental topics in computer science. Most of what computer systems spend their time doing is *storing*, *accessing*, and *manipulating* data in one form or another. Some examples from computer science include:

Information Retrieval: List the 10 most informative Web pages on the subject of "how to treat high blood pressure?" Identify possible suspects of a crime based on fingerprints or DNA evidence. Find movies that a Netflix subscriber may like based on the movies that this person has already viewed. Find images on the Internet containing both kangaroos and horses.

Geographic Information Systems: How many people in the USA live within 25 miles of the Mississippi River? List the 10 movie theaters that are closest to my current location. If sea levels rise 10 meters, what fraction of Florida will be under water?

Compilers: You need to store a set of variable names along with their associated types. Given an assignment between two variables we need to look them up in a symbol table, determine their types, and determine whether it is possible to cast from one type to the other).

Networking: Suppose you need to multicast a message from one source node to many other machines on the network. Along what paths should the message be sent, and how can this set of paths be determined from the network's structure?

Computer Graphics: Given a virtual reality system for an architectural building walk-through, what portions of the building are visible to a viewer at a particular location? (Visibility culling)

In many areas of computer science, much of the content deals with the questions of how to store, access, and manipulate the data of importance for that area. In this course we will deal with the first two tasks of storage and access at a very general level. (The last issue of manipulation is further subdivided into two areas, manipulation of numeric or floating point data, which is the subject of numerical analysis, and the manipulation of discrete data, which is the subject of discrete algorithm design.) An good understanding of data structures is fundamental to all of these areas.

What is a *data structure*? Whenever we deal with the representation of real world objects in a computer program we must first consider a number of issues:

Modeling: the manner in which objects in the real world are modeled as abstract mathematical entities and basic data types,

Operations: the operations that are used to store, access, and manipulate these entities and the formal meaning of these operations,

Representation: the manner in which these entities are represented concretely in a computer's memory, and

Algorithms: the algorithms that are used to perform these operations.

Note that the first two items above are essentially mathematical in nature, and deal with the "what" of a data structure, whereas the last two items involve the implementation issues and the "how" of the data structure. The first two essentially encapsulate the essence of an *abstract data type* (or ADT). In contrast the second two items, the concrete issues of implementation, will be the focus of this course.

For example, you are all familiar with the concept of a *stack* from basic programming classes. This a sequence of *objects* (of unspecified type). Objects can be inserted into the stack by *pushing* and removed from the stack by *popping*. The pop operation removes the last unremoved object that was pushed. Stacks may be implemented in many ways, for example using arrays or using linked lists. Which representation is the fastest? Which is the most space efficient? Which is the most flexible? What are the tradeoffs involved with the use of one representation over another? In the case of a stack, the answers are all rather mundane. However, as data structures grow in complexity and sophistication, the answers are far from obvious.

In this course we will explore a number of different data structures, study their implementations, and analyze their efficiency (both in time and space). One of our goals will be to provide you with the *tools* and *design principles* that will help you to design and implement your own data structures to solve your own data storage and retrieval problems efficiently.

Course Overview: In this course we will consider many different abstract data types and various data structures for implementing each of them. There is not always a single "best" data structure for a given task. For example, there are many common sorting algorithms: Bubble-Sort is easy to code but slow, Quick-Sort is very fast but not stable, Merge-Sort is stable but needs additional memory, and Heap-Sort needs no additional memory but is hard to code (relative to Quick-Sort). It will be important for you, as a designer of the data structure, to understand each structure well enough to know the circumstances where one data structure is to be preferred over another.

How important is the choice of a data structure? There are numerous examples from all areas of computer science where a relatively simple application of good data structure techniques resulted in massive savings in computation time and, hence, money.

Perhaps a more important aspect of this course is a sense of how to design new data structures, how to implement these designs, and how to evaluate how good your design is. The data structures we will cover in this course have grown out of the standard applications of computer science. But new applications will demand the creation of new domains of objects (which we cannot foresee at this time) and this will demand the creation of new data structures. It will fall on the students of today to create these data structures of the future. We will see that there are a few important elements which are shared by all good data structures. We will also discuss how one can apply simple mathematics and common sense to quickly ascertain the weaknesses or strengths of one data structure relative to another.

Our Approach: We will consider the design of data structures from two different perspectives: *theoretical* and *practical*. Our theoretical analysis of data structures will be similar in style to the approach taken in algorithms courses (such as CMSC 351). The emphasis will be on deriving asymptotic (so called, "big-O") bounds on the space, query time, and cost of operations for a given data structure. On the practical side, you will be writing programs to implement a number of classical data structures. This will acquaint you with the skills needed to develop clean designs and debug them.

The remainder of the lecture will review some material from your earlier algorithms course, which hopefully you still remember.

Algorithmics: *Review material. Please be sure you are familiar with this.*

It is easy to see that the topics of algorithms and data structures cannot be separated since the two are inextricably intertwined. So before we begin talking about data structures, we must begin with a quick review of the basics of algorithms, and in particular, how to measure the relative efficiency of algorithms. The main issue in studying the efficiency of algorithms is the amount of resources they use, usually measured in either the *space* or *time* used. There are usually two ways of measuring these quantities. One is a mathematical analysis of the general algorithm being used, called an *asymptotic analysis*, which can capture gross aspects of efficiency for all possible inputs but not exact execution times. The second is an *empirical analysis* of an actual implementation to determine exact running times for a sample of specific inputs, but it cannot predict the performance of the algorithm on all inputs. In class we will deal mostly with the former, but the latter is important also.[1]

For now let us concentrate on running time. (What we are saying can also be applied to space, but space is somewhat easier to deal with than time.) Given a program, its running time is not a fixed number, but rather a function. For each input (or instance of the data structure), there may be a different running time. Presumably as input size increases so does running time, so we often describe running time as a function of input/data structure size n, denoted $T(n)$. We want our notion of time to be largely machine-independent, so rather than measuring CPU seconds, it is more common to measure basic "steps" that the algorithm makes (e.g. the number of statements executed or the number of memory accesses). This will not exactly predict the true running time, since some compilers do a better job of optimization than others, but its will get us within a small constant factor of the true running time most of the time.

Even measuring running time as a function of input size is not really well defined, because, for example, it may be possible to sort a list that is already sorted, than it is to sort a list that is randomly permuted. For this reason, we usually talk about *worst case* running time. Over all possible inputs of size n, what is the maximum running time. It is often more reasonable to consider *expected case* running time where we average over all inputs of size n. When dealing with *randomized algorithms* (where the execution depends on random choices), it is common to focus on the worst case over all inputs (of a given size) and expected case over all random choices. Another example that is often used in data structure design is called *amortized analysis*, where the average is taken over a series of operation. Any one operation might be costly, but the overall average in any long sequence cannot be. We will usually do worst-case analysis, except where it is clear that the worst case is significantly different from the expected case.

Review of Asymptotics: There are particular bag of tricks that most algorithm analyzers use to study the running time of algorithms. For this class we will try to stick to the basics. The

[1]Of course, there is another aspect of complexity, that we will not discuss at length (but needs to be considered) and that is the software-engineering issues regarding the complexity of programming a correct implementation.

first element is the notion of asymptotic notation. Suppose that we have already performed an analysis of an algorithm and we have discovered through our worst-case analysis that

$$T(n) \;=\; 13n^3 + 42n^2 + 2n\log n + 3\sqrt{n}.$$

(This function was just made up as an illustration.) Unless we say otherwise, assume that logarithms are taken base 2. When the value n is small, we do not worry too much about this function since it will not be too large, but as n increases in size, we will have to worry about the running time. Observe that as n grows larger, the size of n^3 is much larger than n^2, which is much larger than $n\log n$ (note that $0 < \log n < n$ whenever $n > 1$) which is much larger than $\sqrt{n}$. Thus the n^3 term dominates for large n. Also note that the leading factor 13 is a constant. Such constant factors can be affected by the machine speed, or compiler, so we may ignore it (as long as it is relatively small). We could summarize this function succinctly by saying that the running time grows "roughly on the order of n^3", and this is written notationally as $T(n) = O(n^3)$. Informally, the statement $T(n) = O(n^3)$ means, "when you ignore constant multiplicative factors, and consider the leading (i.e. fastest growing) term, you get n^3". This intuition can be made more formal, however. We refer you back to your algorithms course for how to do this.

Digression about Notation: Let us pause for a moment to explain the "=" in the assertion that "$T(n) = O(n^3)$". It is not being used in the usual mathematical sense. By definition, "$a = b$" implies "$b = a$". The use of "=" in asymptotic notation just a lazy way of saying $T(n)$ "is" on the order of n^3. It would *not* make sense to express this as $O(n^3) = T(n)$. A more proper way of writing this expression would be "$T(n) \in O(n^3)$", that is $T(n)$ is a member of the set of functions whose asymptotic growth rate is at most n^3. But, most of the world simply uses the "=" notation, and so shall we.

Common Complexity Classes: To get a feeling what various growth rates mean here is a summary. In data structures, our objective is usually to answer a query in time that is less than the size of the data set n, so in this class we will be primarily interested in the following complexity classes, which are collectively called *sublinear.*

$T(n) = O(1)$: Great. This means your algorithm takes only *constant time.* You can't beat this. (Example: Popping a stack.)

$T(n) = O(\alpha(n))$: The function α is the inverse of the famous *Ackerman's function.* Ackerman's function grows *insanely* rapidly, and hence its inverse grows *insanely* slowly. How slowly? If n is less than the number of atoms in the visible universe, then $\alpha(n) \leq 5$. Thus, $\alpha(n)$ is a constant "for all practical purposes". However, formally it is *not* a constant. In the limit, as n tends to infinity, $\alpha(n)$ also tends to infinity. It just gets there extremely slowly! Believe or not, there are actually a number of data structures whose running times are $O(\alpha(n))$, but they are *not* $O(1)$. (Example: Disjoint set union-find.)

$T(n) = O(\log\log n)$: Super fast! For most practical purposes, this is as fast as a constant time. (Example: Van Emde Boas trees.)

$T(n) = O(\log n)$: Very good. This is called *logarithmic* time, and is the "gold standard" for data structures based on making binary comparisons. It is the running time of binary search and the height of a balanced binary tree. This is about the best that can be achieved for data structures based on binary trees. Note that $\log 1000 \approx 10$ and $\log 1,000,000 \approx 20$ (log's base 2). (Example: Binary search.)

$T(n) = O((\log n)^k)$: (where k is a constant). This is called *polylogarithmic* time. Not bad, when simple logarithmic time is not achievable. We will often write this as $O(\log^k n)$. (Example: Orthogonal range searching, that is, counting the number of points in a d-dimensional axis-parallel rectangle.)

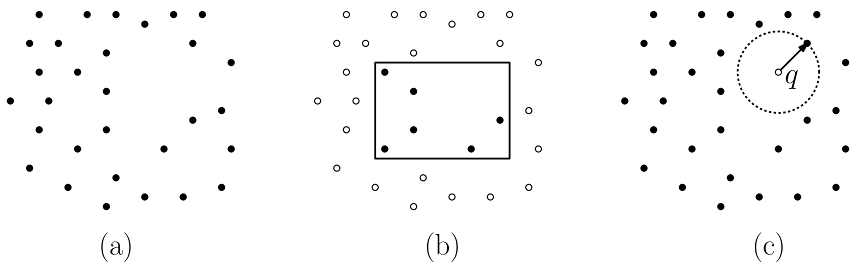

Fig. 1: (a) A point set, (b) orthogonal range search query, and (c) nearest-neighbor query.

$T(n) = O(n^p)$: (where $0 < p < 1$ is a constant). An example is $O(\sqrt{n})$. This is slower than polylogarithmic (no matter how big k is or how small p), but is still faster than linear time, which is acceptable for data structure use. (Example: Nearest neighbor searching in d-dimensional space.)

In an algorithms course, it is more common to focus on running times that grow at least linearly. These are described below.

$T(n) = O(n)$: This is called *linear* time. It is about the best that one can hope for if your algorithm has to look at all the data. (Example: Enumerating the elements of a linked list.)

$T(n) = O(n \log n)$: This one is famous, because this is the time needed to sort a list of numbers by means of comparisons. It arises in a number of other problems as well. (Example: Sorting, of course.)

$T(n) = O(n^2)$: *Quadratic* time. Okay if n is in the thousands, but rough when n gets into the millions. (Example: 3Sum: Given a list of n numbers (positive and negative), do any three sum to zero?)

$T(n) = O(n^k)$: (where k is a constant). This is called *polynomial* time. Practical if k is not too large. (Example: Matrix multiplication.)

$T(n) = O(2^n), O(n^n), O(n!)$: *Exponential* time. Algorithms taking this much time are only practical for the smallest values of n (e.g. $n \leq 10$ or maybe $n \leq 20$). (Example: Your favorite NP-complete problem…as far as anyone knows!)