CMSC 420: Data Structures[1]
Spring 2001
Dave Mount

[1]Copyright, David M. Mount, 2001, Dept. of Computer Science, University of Maryland, College Park, MD, 20742. These lecture notes were prepared by David Mount for the course CMSC 420, Data Structures, at the University of Maryland, College Park. Permission to use, copy, modify, and distribute these notes for educational purposes and without fee is hereby granted, provided that this copyright notice appear in all copies.

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:

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?

Information Retrieval: How does a search engine like Google store the contents of the Web so that the few pages that are most relevant to a given query can be extracted quickly?

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 (say because one is a subtype of the other).

Computer Graphics: You are designing a virtual reality system for an architectural building walk-through. Given the location of the viewer, what portions of the architectural design are visible to the viewer?

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

[1]Copyright, David M. Mount, 2001

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 that you will need to design and implement your own data structures to solve your own specific problems in data storage and retrieval.

Course Overview: In this course we will consider many different abstract data types, and we will consider many different data structures for storing each type. Note that there will generally be many possible data structures for each abstract type, and there will not generally be a "best" one for all circumstances. It will be important for you as a designer of data structures 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. 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.

Algorithmics: 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.

There is another aspect of complexity, that we will not discuss at length (but needs to be considered) and that is the complexity of programming. Some of the data structures that we will discuss will be quite simple to implement and others much more complex. The issue of which data structure to choose may be dependent on issues that have nothing to do with run-time issues, but instead on the software engineering issues of what data structures are most flexible, which are easiest to implement and maintain, etc. These are important issues, but we will not dwell on them excessively, since they are really outside of our scope.

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. 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 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 $T(n) \in O(n^3)$.

Informally, the statement $T(n) \in O(n^3)$ 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. (It is not the most standard one, but is good enough for most uses and is the easiest one to apply.)

Definition: $T(n) \in O(f(n))$ if $\lim_{n\to\infty} T(n)/f(n)$ is either zero or a constant (but not ∞).

For example, we said that the function above $T(n) \in O(n^3)$. Using the definition we have

$$\begin{aligned} \lim_{n\to\infty} \frac{T(n)}{f(n)} &= \lim_{n\to\infty} \frac{13n^3 + 42n^2 + 2n\log n + 3\sqrt{n}}{n^3} \\ &= \lim_{n\to\infty} \left(13 + \frac{42}{n} + \frac{2\log n}{n^2} + \frac{3}{n^{2.5}}\right) \\ &= 13. \end{aligned}$$

Since this is a constant, we can assert that $T(n) \in O(n^3)$.

The O notation is good for putting an upper bound on a function. Notice that if $T(n)$ is $O(n^3)$ it is also $O(n^4)$, $O(n^5)$, etc. since the limit will just go to zero. We will try to avoid getting bogged down in this notation, but it is important to know the definitions. To get a feeling what various growth rates mean here is a summary.

$T(n) \in O(1)$: Great. This means your algorithm takes only constant time. You can't beat this.

$T(n) \in O(\log\log n)$: Super fast! For all intents this is as fast as a constant time.

$T(n) \in O(\log n)$: Very good. This is called *logarithmic* time. 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).

$T(n) \in O((\log n)^k)$: (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 $O(\log^k n)$.

$T(n) \in O(n^p)$: (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.

$T(n) \in O(n)$: 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. (In data structures the goal is usually to avoid this though.)

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

$T(n) \in O(n^2)$: *Quadratic* time. Okay if n is in the thousands, but rough when n gets into the millions.

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

$T(n) \in O(2^n), O(n^n), O(n!)$: *Exponential* time. Algorithms taking this much time are only practical for the smallest values of n (e.g. $n \le 10$ or maybe $n \le 20$).