Welcome to

CSE 326
Data Structures

Staff

- Instructor
  - Tami Tamir
tami@cs.washington.edu
  - office hours: Friday 11:30-12:30 or by appointment
- TA’s
  - Matt Cary (cary@cs.washington.edu)
  - Xu Miao (xm@cs.washington.edu)

See web-page for office hours.

Web Page

- All info is on the web page for CSE 326
- also known as

CSE 326 E-mail List

- Subscribe by going to the class web page.
- E-mail list is used for posting announcements by instructor and TAs.
- It is your responsibility to subscribe. It might turn out to be very helpful for assignments hints, corrections etc.
Textbook

- *Data Structures and Algorithm Analysis in Java (or in C++), by Weiss*
- See Web page for errata and source code

Grading

- Dry assignments 25% - submit in singles
- Wet assignments (programming projects) 25% - can submit in pairs.
- Midterm 20%
  ‣ Friday, Feb 6, 2004
- Final 30%
  ‣ Group I: 8:30-10:20 a.m. Thursday, Mar. 18, 2004
  ‣ Group II: 2:30-4:20 p.m. Tuesday, Mar. 16, 2004

Class Overview

- Introduction to many of the basic data structures used in computer software
  ‣ Understand the data structures
  ‣ Analyze the algorithms that use them
  ‣ Know when to apply them
- Practice design and analysis of data structures.
- Practice using these data structures by writing programs.

Goal

- You will understand
  ‣ what the tools are for storing and processing common data types
  ‣ which tools are appropriate for which need
- So that you will be able to
  ‣ make good design choices as a developer, project manager, or system customer
Course Topics

- Introduction to Algorithm Analysis
- Lists, Stacks, Queues
- Search Algorithms and Trees
- Hashing and Heaps
- Sorting
- Disjoint Sets
- Graph Algorithms

Data Structures: What?

- Need to organize program data according to problem being solved
- Abstract Data Type (ADT) - A data object and a set of operations for manipulating it
  › List ADT with operations insert and delete
  › Stack ADT with operations push and pop
- Note similarity to Java classes
  › private data structure and public methods

Data Structures: Why?

- Program design depends crucially on how data is structured for use by the program
  › Implementation of some operations may become easier or harder
  › Speed of program may dramatically decrease or increase
  › Memory used may increase or decrease
  › Debugging may be become easier or harder

Reading

- Chapters 1 and 2, *Data Structures and Algorithm Analysis in Java*, by Weiss
  › Most of Chapter 2 will be seen in class next week.
Terminology

- **Abstract Data Type (ADT)**
  › Mathematical description of an object with set of operations on the object. Useful building block.
- **Algorithm**
  › A high level, language independent, description of a step-by-step process
- **Data structure**
  › A specific family of algorithms for implementing an abstract data type.
- **Implementation of data structure**
  › A specific implementation in a specific language

Algorithm Analysis: Why?

- **Correctness:**
  › Does the algorithm do what is intended.
- **Performance:**
  › What is the running time of the algorithm.
  › How much storage does it consume.
- **Different algorithms may correctly solve a given task**
  › Which should I use?

Evaluating an algorithm

Mike: My algorithm can sort 10^6 numbers in 3 seconds.
Bill: My algorithm can sort 10^6 numbers in 5 seconds.

Mike: I’ve just tested it on my new Pentium IV processor.
Bill: I remember my result from my undergraduate studies (1985).

Mike: My input is a random permutation of 1..10^6.
Bill: My input is the sorted output, so I only need to verify that it is sorted.

Program Evaluation / Complexity

- **Processing time is surely a bad measure!!!**
- **We need a ‘stable’ measure, independent of the implementation.**

  * A complexity function is a function \( T: \mathbb{N} \rightarrow \mathbb{N} \).
  \( T(n) \) is the number of operations the algorithm does on an input of size \( n \).

  * We can measure three different things.

    - **Worst-case complexity**
    - **Best-case complexity**
    - **Average-case complexity**
The RAM Model of Computation

- Each simple operation takes 1 time step.
- Loops and subroutines are not simple operations.
- Each memory access takes one time step, and there is no shortage of memory.

For a given problem instance:
- Running time of an algorithm = \# RAM steps.
- Space used by an algorithm = \# RAM memory cells

useful abstraction ⇒ allows us to analyze algorithms in a machine independent fashion.

Big O Notation

- Goal:
  › A stable measurement independent of the machine.

- Way:
  › Ignore constant factors.

  f(n) = O(g(n)) if c \cdot g(n) is upper bound on f(n)

  ⇔ There exist c, N, s.t. for any n ≥ N, f(n) ≤ c \cdot g(n)

  Consider large inputs (asymptotic behavior)

  Ignore constants

Ω, Θ Notation

- f(n) = Ω(g(n)) if c \cdot g(n) is lower bound on f(n)

  ⇔ There exist c, N, s.t. for any n ≥ N, f(n) ≥ c \cdot g(n)

- f(n) = Θ(g(n)) if f(n) = O(g(n)) and f(n) = Ω(g(n))

  ⇔ There exist c₁, c₂, N, s.t. for n ≥ N,

  c₁ \cdot g(n) ≤ f(n) ≤ c₂ \cdot g(n)
**Ω, Θ Notation**

**Examples:**

- $4x^2 + 100 = O(x^2)$
- $4x^2 + 100 = \Omega(x^2)$
- $4x^2 + 100 = \Theta(x^2)$
- $4x^2 - 100 = O(x^2)$
- $123400 = O(1)$

**Growth Rates**

- Even by ignoring constant factors, we can get an excellent idea of whether a given algorithm will be able to run in a reasonable amount of time on a problem of a given size.
- The “big O” notation and worst-case analysis are tools that greatly simplify our ability to compare the efficiency of algorithms.

**Practical Complexity**

Graph showing growth rates for various functions, including:
- $f(n) = n$
- $f(n) = \log(n)$
- $f(n) = n \log(n)$
- $f(n) = n^2$
- $f(n) = n^3$
- $f(n) = 2^n$
**Big O Fact**

- A polynomial of degree k is $O(n^k)$
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
  - Let $a = \max_i \{ b_i \}$
  - $f(n) \leq an^k + an^{k-1} + \ldots + an + a$
  - $\leq kan^k \leq cn^k$ (for $c=ka$).

**Iterative Algorithm for Sum**

- Find the sum of the first `num` integers stored in an array `v`.

```java
sum(v[]): integer array, num: integer): integer{
  temp_sum: integer ;
  temp_sum := 0;
  for i := 0 to num - 1 do
    temp_sum := v[i] + temp_sum;
  return temp_sum;
}
```

Note the use of pseudocode
Programming via Recursion

• Write a recursive function to find the sum of the first num integers stored in array v.

```java
sum (v[ ]: integer array, num: integer): integer {
    if (num = 0) then
        return 0
    else
        return (v[num-1] + sum(v,num-1));
}
```

Pseudocode

• In the lectures algorithms will be presented in pseudocode.
  › This is very common in the computer science literature
  › Pseudocode is usually easily translated to real code.
  › This is programming language independent

• Pseudocode should also be used for homework (dry ones)

Review: Induction

• Suppose
  › S(k) is true for fixed constant k
    • Often k = 0
  › S(n) implies S(n+1) for all n >= k
• Then S(n) is true for all n >= k

Proof By Induction

• Claim: S(n) is true for all n >= k
• Base:
  › Show S(n) is true for n = k
• Inductive hypothesis:
  › Assume S(n) is true for an arbitrary n
• Step:
  › Show that S(n) is then true for n+1
Induction Example: Geometric Closed Form

• Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$

  › Basis: 1. show that $a^0 = (a^{0+1} - 1)/(a - 1)$:
    $a^0 = 1 = (a^1 - 1)/(a - 1)$. 2. Show true for $n=2$.

  › Inductive hypothesis:
    • Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$

  › Step (show true for $n+1$):
    $a^0 + a^1 + \ldots + a^n + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$
    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+2} - 1)/(a - 1)$

Program Correctness by Induction

• Basis Step: $\text{sum}(v,0) = 0$.

• Inductive Hypothesis ($n=k$): Assume $\text{sum}(v,k)$ correctly returns sum of first $k$ elements of $v$, i.e. $v[0]+v[1]+\ldots+v[k-1]$

• Inductive Step ($n=k+1$): $\text{sum}(v,n)$ returns $v[k]+\text{sum}(v,k)$ which is the sum of first $k+1$ elements of $v$.

Algorithms vs Programs

• Proving correctness of an algorithm is very important
  › a well designed algorithm is guaranteed to work correctly and its performance can be estimated

• Proving correctness of a program (an implementation) is fraught with weird bugs
  › Abstract Data Types are a way to bridge the gap between mathematical algorithms and programs