CSE 326: Data Structures

Introduction

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.
• Make the transformation from programmer to computer scientist

Goals

• You will understand
  – what the tools are for storing and processing common data types
  – which tools are appropriate for which need
• So that you can
  – make good design choices as a developer, project manager, or system customer
• You will be able to
  – Justify your design decisions via formal reasoning
  – Communicate ideas about programs clearly and precisely

Goals

“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

Linus Torvalds, 2006

Goals

“Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won’t usually need your flowcharts; they’ll be obvious.”

Fred Brooks, 1975

Data Structures

“Clever” ways to organize information in order to enable efficient computation

– What do we mean by clever?
– What do we mean by efficient?
Picking the best Data Structure for the job

- The data structure you pick needs to support the operations you need
- Ideally it supports the operations you will use most often in an efficient manner
- Examples of operations:
  - A List with operations insert and delete
  - A Stack with operations push and pop

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

Terminology examples

- A stack is an abstract data type supporting push, pop and isEmpty operations
- A stack data structure could use an array, a linked list, or anything that can hold data
- One stack implementation is java.util.Stack; another is java.util.LinkedList

Concepts vs. Mechanisms

- Abstract
- Pseudocode
- Algorithm
  - A sequence of high-level, language independent operations, which may act upon an abstracted view of data.
- Abstract Data Type (ADT)
  - A mathematical description of an object and the set of operations on the object.
- Concrete
- Specific programming language
- Program
  - A sequence of operations in a specific programming language, which may act upon real data in the form of numbers, images, sound, etc.
- Data structure
  - A specific way in which a program’s data is represented, which reflects the programmer’s design choices/goals.

Why So Many Data Structures?

Ideal data structure:
“fast”, “elegant”, memory efficient
Generates tensions:
- time vs. space
- performance vs. elegance
- generality vs. simplicity
- one operation’s performance vs. another’s

The study of data structures is the study of tradeoffs. That’s why we have so many of them!

Today’s Outline

- Introductions
- Administrative Info
- What is this course about?
- Review: Queues and stacks
First Example: Queue ADT

- FIFO: First In First Out
- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty

Circular Array Queue Data Structure

Linked List Queue Data Structure

Circular Array vs. Linked List

- Too much space
- Kth element accessed "easily"
- Not as complex
- Could make array more robust
- Can grow as needed
- Can keep growing
- No back looping around to front
- Linked list code more complex

Second Example: Stack ADT

- LIFO: Last In First Out
- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty

Stacks in Practice

- Function call stack
- Removing recursion
- Balancing symbols (parentheses)
- Evaluating Reverse Polish Notation
Data Structures
Asymptotic Analysis

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 be correct
  – Which should I use?

Recursive algorithm for sum
• Write a recursive function to find the sum of the first n integers stored in array v.

```plaintext
sum(integer array v, integer n) returns integer
if n = 0 then
  sum = 0
else
  sum = nth number + sum of first n-1 numbers
return sum
```

Proof by Induction
• Basis Step: The algorithm is correct for a base case or two by inspection.
• Inductive Hypothesis (n=k): Assume that the algorithm works correctly for the first k cases.
• Inductive Step (n=k+1): Given the hypothesis above, show that the k+1 case will be calculated correctly.

Program Correctness by Induction
• Basis Step:
  `sum(v, 0) = 0. √`
• Inductive Hypothesis (n=k):
  Assume `sum(v, k)` correctly returns sum of first k elements of v, i.e. `v[0]+v[1]+…+v[k-1]+v[k]`
• Inductive Step (n=k+1):
  `sum(v, n)` returns
  `v[k]+sum(v, k-1) = (by inductive hyp.)`

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
Comparing Two Algorithms

GOAL: Sort a list of names

"I'll buy a faster CPU"
"I'll use C++ instead of Java – wicked fast!"
"Ooh look, the –O4 flag!"
"Who cares how I do it, I'll add more memory!"
"Can't I just get the data pre-sorted??"

• What we want:
  – Rough Estimate
  – Ignores Details

• Really, independent of details
  – Coding tricks, CPU speed, compiler optimizations, …
  – These would help any algorithms equally
  – Don’t just care about running time – not a good enough measure

Big-O Analysis

• Ignores "details"
• What details?
  – CPU speed
  – Programming language used
  – Amount of memory
  – Compiler
  – Order of input
  – Size of input … sorta.

Analysis of Algorithms

• Efficiency measure
  – how long the program runs \(T(n)\)
  – how much memory it uses \(S(n)\)

• Why analyze at all?
  – Decide what algorithm to implement before actually doing it
  – Given code, get a sense for where bottlenecks must be, without actually measuring it

Asymptotic Analysis

One detail we won't ignore: problem size, # of input elements

• Complexity as a function of input size \(n\)
  \[T(n) = 4n + 5\]
  \[T(n) = 0.5n \log n - 2n + 7\]
  \[T(n) = 2^n + n^3 + 3n\]

• What happens as \(n\) grows?

Why Asymptotic Analysis?

• Most algorithms are fast for small \(n\)
  – Time difference too small to be noticeable
  – External things dominate (OS, disk I/O, …)

• BUT \(n\) is often large in practice
  – Databases, internet, graphics, …

• Difference really shows up as \(n\) grows!
Exercise - Searching

```c
bool ArrayFind(int array[], int n, int key) {
    // Insert your algorithm here
    return false;
}
```

What algorithm would you choose to implement this code snippet?

Analyzing Code

- Constant time
- Sum of times
- Larger branch plus test
- Sum of iterations
- Cost of function body
- Solve recurrence relation

Linear Search Analysis

```c
bool LinearArrayFind(int array[], int n, int key) {
    for(int i = 0; i < n; i++) {
        if(array[i] == key)
            return true;
    }
    return false;
}
```

Best Case: 3
Worst Case: 2n+1

Binary Search Analysis

```c
bool BinArrayFind(int array[], int low, int high, int key) {
    if(low > high) return false;
    int mid = (high + low) / 2;
    if(key == array[mid])
        return true;
    else if(key < array[mid])
        return BinArrayFind(array, low, mid-1, key);
    else
        return BinArrayFind(array, mid+1, high, key);
}
```

Best case: 4
Worst case: log n

Solving Recurrence Relations

1. Determine the recurrence relation. What is/are the base case(s)?
2. “Expand” the original relation to find an equivalent general expression in terms of the number of expansions.
3. Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case.

Data Structures

Asymptotic Analysis
Linear Search vs Binary Search

<table>
<thead>
<tr>
<th>Best Case</th>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case</td>
<td>3n + 2</td>
<td>4 log n + 4</td>
</tr>
</tbody>
</table>

So ... which algorithm is better? What tradeoffs can you make?

Fast Computer vs. Slow Computer

Fast Computer vs. Smart Programmer (round 1)

Fast Computer vs. Smart Programmer (round 2)

Asymptotic Analysis

- Asymptotic analysis looks at the order of the running time of the algorithm
  - A valuable tool when the input gets “large”
  - Ignores the effects of different machines or different implementations of an algorithm
- Intuitively, to find the asymptotic runtime, throw away the constants and low-order terms
  - Linear search is $T(n) = 3n + 2 \in O(n)$
  - Binary search is $T(n) = 4 \log n + 4 \in O(\log n)$

Remember: the fastest algorithm has the slowest growing function for its runtime

Asymptotic Analysis

- Eliminate low order terms
  - $4n + 5 \Rightarrow 0.5 n \log n + 2n + 7 \Rightarrow n^3 + 2^n + 3n \Rightarrow$
- Eliminate coefficients
  - $4n \Rightarrow 0.5 n \log n \Rightarrow n \log n^2 \Rightarrow$
Properties of Logs

• \( \log AB = \log A + \log B \)

Proof:
\[
A = 2^{\log_2 A}, \quad B = 2^{\log_2 B}
\]
\[
AB = 2^{\log_2 A} \cdot 2^{\log_2 B} = 2^{\log_2 A + \log_2 B}
\]
\[
\therefore \log AB = \log A + \log B
\]

• Similarly:
  – \( \log(A/B) = \log A - \log B \)
  – \( \log(A^p) = p \log A \)

• Any log is equivalent to log-base-2

Order Notation: Intuition

Although not yet apparent, as \( n \) gets “sufficiently large”, \( f(n) \) will be “greater than or equal to” \( g(n) \)

Definition of Order Notation

• Upper bound:
  \( T(n) = O(f(n)) \) Big-O
  Exist positive constants \( c \) and \( n' \) such that
  \( T(n) \leq c f(n) \) for all \( n \geq n' \)

• Lower bound:
  \( T(n) = \Omega g(n) \) Omega
  Exist positive constants \( c \) and \( n' \) such that
  \( T(n) \geq c g(n) \) for all \( n \geq n' \)

• Tight bound:
  \( T(n) = \Theta f(n) \) Theta
  When both hold:
  \( T(n) = O(f(n)) \)
  \( T(n) = \Omega g(n) \)

Definition of Order Notation

\( O(f(n)) \): a set or class of functions

\( g(n) \in O(f(n)) \) iff there exist positive consts \( c \) and \( n_0 \) such that:
\( g(n) \leq c f(n) \) for all \( n \geq n_0 \)

Example:
\( 100n^2 + 1000 \leq 5(n^3 + 2n^2) \) for all \( n \geq 19 \)

So \( g(n) \in O(f(n)) \)

Some Notes on Notation

• Sometimes you’ll see
  \( g(n) = O(f(n)) \)

• This is equivalent to
  \( g(n) \in O(f(n)) \)

• What about the reverse?
  \( O(f(n)) = g(n) \)
Big-O: Common Names

- constant: $O(1)$
- logarithmic: $O(\log n)$ ($\log_2 n, \log n^2 \in O(\log n)$)
- linear: $O(n)$
- log-linear: $O(n \log n)$
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$ ($k$ is a constant)
- exponential: $O(c^n)$ ($c$ is a constant $> 1$)

Meet the Family

- $O(f(n))$ is the set of all functions asymptotically less than or equal to $f(n)$
- $o(f(n))$ is the set of all functions asymptotically strictly less than $f(n)$
- $\Omega(f(n))$ is the set of all functions asymptotically greater than or equal to $f(n)$
- $\omega(f(n))$ is the set of all functions asymptotically strictly greater than $f(n)$
- $\Theta(f(n))$ is the set of all functions asymptotically equal to $f(n)$

Meet the Family, Formally

- $g(n) = O(f(n))$ iff
  - There exist $c$ and $n_0$ such that $g(n) \leq c f(n)$ for all $n \geq n_0$
  - $g(n) = o(f(n))$ iff $g(n)/f(n) \to 0$ as $n \to \infty$
  - $g(n) = \omega(f(n))$ iff $g(n)/f(n) \to \infty$ as $n \to \infty$

Big-Omega et al. Intuitively

<table>
<thead>
<tr>
<th>Asymptotic Notation</th>
<th>Mathematics Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O$</td>
<td>$\leq$</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>$=$</td>
</tr>
<tr>
<td>$o$</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$\omega$</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>

Pros and Cons of Asymptotic Analysis

- Running time may depend on actual data input, not just length of input
- Distinguish
  - Worst Case
    - Your worst enemy is choosing input
  - Best Case
  - Average Case
    - Assumes some probabilistic distribution of inputs
  - Amortized
    - Average time over many operations

Perspective: Kinds of Analysis
Types of Analysis

Two orthogonal axes:

- **Bound Flavor**
  - Upper bound (O, o)
  - Lower bound (Ω, ω)
  - Asymptotically tight (Θ)

- **Analysis Case**
  - Worst Case (Adversary)
  - Average Case
  - Best Case
  - Amortized

\[ 16n^3 \log_8(10n^2) + 100n^2 = O(n^3 \log n) \]

- Eliminate low-order terms
  - \[ 16n^3 \log_8(10n^2) + 100n^2 \]
  - \[ n^3 \log_8(10) \]
  - \[ 2n^3 \log_8(n) \]
  - \[ n^3 \log(n) \]

- Eliminate constant coefficients
  - \[ 16n^3 \log_8(10n^2) \]
  - \[ 2n^3 \log(n) \]
  - \[ n^3 \log(n) + 100n^2 \]
  - \[ n^3 \log_8(10) \]
  - \[ n^3 \log_8(n^2) \]
  - \[ n^3 \log(n^2) \]
  - \[ n^3 \log_8(n) \]
  - \[ n^3 \log_8(2) \]
  - \[ n^3 \log(n) / 3 \]
  - \[ n^3 \log(n) \]