Grading

• Grades will be assigned using the University's standard scale for A to (eek) F
  o A = 100%-95%; A- = 94%-90%; etc.
  o Homework + Exams = 200 points available
• Curved Grades
  o Minimum overall average will be adjusted to at least 85%
  o Curve will be applied at end of course to total score
• Homework
  o Programming Assignments (5*10 = 50 pts)
  o Written Assignments ("Homework") (5*10=50 pts)
  o The Never-Late Rule
• Exams (2*50=100 pts)
• Bonus Points Opportunities (no penalty)
  ▪ In-Class Pop Quiz (5*2=10pts)
  ▪ Homework "extra" problems (5*2=10pts)
Course Resources

- Primary website: [http://mathcs.emory.edu/~jsmit52/cs323](http://mathcs.emory.edu/~jsmit52/cs323)
  - Schedule & General Information
  - Links to downloadable texts & lecture materials
  - Locations of restricted materials
    - Copyright caveat
    - Building our own "textbook"
  - Homework submission via SOSS:
    - [http://simon.library.emory.edu/submission/login.php](http://simon.library.emory.edu/submission/login.php)

- Course programming server: simon.library.emory.edu
  - Ubuntu Linux
  - Usual set of user tools (vim, grep, etc.)
  - Login/file upload via ssh and scp
    - "ssh loginId@simon.library.emory.edu"
    - "scp myfile.py loginId@simon.library.emory.edu:myDir/"
  - Python version 2.6.x installed
    - You **must** upload and test your file on that server before submitting it for grading
Course Goals

Description of the course in Emory's OPUS database:

"This is an advanced course in computer science, studying ideas and algorithms for processing data in a computer. The course will involve studying such data structures as lists, trees, heaps, hash tables, graphs, and their applications in the construction of efficient algorithms. Searching and sorting algorithms will be also discussed."

Our working goal is to improve our problem solving capabilities:

- Understand the impact of data structures on program design
- Become familiar with essential data structures commonly used in programming
- Evaluate performance of algorithms
- Improve estimation skills
- Improve program design skills (i.e., efficiency)

Learning to stop and think about a problem before jumping in to solve it may be the most important approach you can learn from this course

Creating a solutions checklist may be the 2nd most important thing you learn -- or possibly the most important
A Review of Basic Data Structures

Quiz 1: Things You (Probably) Learned in CS-1 & CS-2

1. What is an abstract data type ("ADT") - also called built-in data type?
2. List at least three common ADTs.
3. What is the purpose of an ADT?
4. What is precision?
5. Why don’t you directly specify the number of bytes to reserve in memory to store data?
6. How does a hash differ from an array and a linked list?
7. What's the difference between an algorithm and an ADT?
8. What's the difference between an algorithm and a program?
9. What's the difference between an array and a linked list?
10. What's the difference between a stack and a queue?
Abstract Data Types

• Role of Abstract Data Types
  o Data & Memory
  o Data Types & Memory
• Abstract Data Type Groups
  o Integers
  o Signed
  o Unsigned
  o Floating Point
  o Characters
  o Boolean
• User-Defined Data Types
  o Memory Allocation
  o Constructs
Data & Memory

1. The fastest memory is CPU cache memory
2. Next is Main Memory (“RAM”)
3. Distant 3rd in speed is persistent storage, mainly because of the need to move between the various buses

- Size of data has obvious impact on even simple access
- Add complexity of code and the impact can be serious…
- If the computational result will be reused, caching can make a big difference
- Consider a Fibonacci sequence, a simple recursion relationship:
  - F(n) = (sum of n, n-1, n-2…)
  - 0, 1, 1, 2, 3, 5, 8, 13, etc.
  - Nested loops? No way. Extra var is the obvious solution
  - Storing results in an array makes later retrieval of any nth item trivial & fast
### Memory Used by Data Types: Java

Chart from *Data Structures Demystified* (Keogh & Davidson, 2004)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Type Size in Bits</th>
<th>Range of Values</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>−128 to 127</td>
<td>Integers</td>
</tr>
<tr>
<td>short 16</td>
<td>16</td>
<td>−32,768 to 32,767</td>
<td>Integers</td>
</tr>
<tr>
<td>int 32</td>
<td>32</td>
<td>−2,147,483,648 to 2,147,483,647</td>
<td>Integers</td>
</tr>
<tr>
<td>long 64</td>
<td>64</td>
<td>−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
<td>Integers</td>
</tr>
<tr>
<td>char 16 (Unicode)</td>
<td>16 (Unicode)</td>
<td>65,536 (Unicode)</td>
<td>Characters</td>
</tr>
<tr>
<td>float 32</td>
<td>32</td>
<td>3.4e-038 to 3.4e+038</td>
<td>Floating-point</td>
</tr>
<tr>
<td>double 64</td>
<td>64</td>
<td>1.7e-308 to 1.7e+308</td>
<td>Floating-point</td>
</tr>
<tr>
<td>boolean 1</td>
<td>1</td>
<td>0 or 1</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Note: Booleans are usually stored as a special case of *int* and so in practice use 32 or 64 bits, depending on the system.
Memory & Untyped Languages

• Allocation on-the-fly
• Can switch the allocation in mid-stream, too.
• Consider this Perl snippet:
  – $title, $word: chars
  – $quant: int
  – Ops on lines 5, 9?
  – $title dynamically changed to int on line 13
• Unexpected operations can produce unexpected results or program failure

```
#!/usr/bin/perl
my ($title, $quant, $new);
$title = "mary";
$quant = 4;
$new = $title + $quant;
print "Title is $title\n";
print "Quantity is $quant\n";
print "Adding T + Q = $new \n";
$new = $title/3;
print "Dividing Title by 3 = $new \n";
my $word = "word";
print "Title is still $title and word is still $word\n";
$title = $title + $word;
print "$title\n";
print "Appending word to title: $title. $word\n";
exit;
```
Arrays

• A simple, often-used construct
  – Declaration is by number of elements
    • `char LastNames[n];`
  – Typically “for \(j = 0\) to \(n-1\)” loop
    • zero-based addressing during array assignment

• Note memory reserved by declaring array size
  – Guarantees sequential addressing in storage
Array Characteristics

• An array is a way to reference a series of memory locations using the same name. Each memory location is represented by an array element. An array element is similar to one variable except it is identified by an index value instead of a name. An index value is a number used to identify an array element. (Keogh & Davidson)

• Some Pros
  – Directly/randomly addressable items, equivalent to individual variables
  – Variables are stored sequentially in memory (in statically-typed langs.)
    • Sequential access of
    • $array[1], $array[2], $array[3] is typically faster than $var1, $var2, $var3
  – Simple code in nearly all languages
  – Rapid data reorganization (arrays of pointers, e.g. to hash)
  – Easily multi-dimensional (arrays of arrays) (also stored contiguously)

• Some Cons
  – Item deletion problematic: waste memory or rewrite the array?
  – Increasing array size also troublesome (if statically typed)
Linked Lists

• Basic construct
  – Doubly-Linked and Singly-Linked Forms
  – Example Doubly-Linked Node:
  – Singly-linked has only the Next() pointer
  – Front & back of list have Null values to indicate beginning, end of list
  – Some languages (Java, e.g.) offer a LinkedList class
    • But what’s underneath it? May depend on language, library

• Pros
  – Dynamic data/record management
  – Runtime modification of size efficient, simple, i.e.,
  – Supports dynamic Inserts and Deletes

• Cons
  – Direct addressing, out-of-order traversal are problematic
  – More complicated code

• Uses of Linked Lists
  – Ordered sets of multi-element information such as customer record
  – Priority Queues
typedef struct Node {
    struct Node(int data) {
        this->data = data; next = NULL;
    }
    int data;
    struct Node* next;
} NODE;

Figure 6-3: A doubly linked list contains next and previous members, and a single linked list contains only a next member.
Linked List Modifications

- Adding a new node requires Temp() variables to track prev() and next()
- Node deletion has similar requirements
- Number of steps depends on whether it is singly- or doubly-linked
Stacks

• The LIFO Concept
  – Last In, First Out Operations
  – Therefore usual operations are:
    • Push() and Pop()
    • IsFull() and IsEmpty()
  – Not designed for random-access use

• Building stacks using arrays
  – An array stores values in memory; a stack tracks which array element is at the top of the stack. When a value is popped off the stack, the value remains in memory because the value is still assigned to an array element. Popping it only changes the array element that is at the top of the stack. (Keogh & Davidson)
  – Stack is a user-defined data structure
  – First item in array is usually **bottom item in stack**
  – Growth? Not efficient with an array underneath the stack

• Stacks can also be built using Linked Lists (see homework)
  – Growth, deletion much easier
Queues

• A FIFO construct: First In, First Out
  – Typical operations are:
    • Enqueue() and Dequeue()
    • IsFull() and IsEmpty()

• Can be built using arrays or linked lists
  – Similar issues regarding growth, decrease in size

• Simple queues
  – Typically built using an Array structure
  – Operates like an array (first item in is at the top)
  – But purpose is to retain order, rather than randomly access items
  – Items remain in order

    --VS --

• Priority queues
  – Often built using Linked List structure
  – Items have a “priority” value, so can “jump to front of line”
  – Processing of items occurs according to priority values first
Simple Queue

Take a look at Figure 5-1 and you’ll see how an array and a queue are different and yet are linked together to organize data. The array is pictured as a block of elements. The queue is pictured as a circle. The empty boxes are where values are stored in the queue, and the numbers correspond to the index of the array that is associated with the queue. To the right of the circle are three values. The front and back values store the index of the front and back of the queue. The size value is the number of elements in the queue, which is 8 in this example.

Figure 5-1: The queue is different from the array used to store data that appears in the queue.

Keogh & Davidson
Hashes

• Basic construct
  – Specialized built-in data type in most languages
    • What’s underneath the Hash structure varies by language, library
  – Unique “key-value” pairs: bits of the key are scrambled into a hash value
  – Typical Operations: get(), put(), find(), has(), remove(), getsize()

• Hash is often used as a "Dictionary" data type
  – Quick look-up of key, value
  – “Automatic” de-duplication

• Advantages of hashes
  – Dynamic inserts and deletes
  – Random access: hashed key returns the value
  – Hash of hashes, hash of arrays, hash of linked lists
  – The “multi-tool” of dictionary-based programming

• Disadvantages of hashes
  – Not an automatic FIFO, LIFO kind of structure
  – Collisions may occur if keys compute to the same hash value
Some Basic Concepts

• Data Structure
• Program
• Algorithm

• Is there a difference?
• Why (or when) should we care about memory allocation?
• Why should we care about algorithms?
Analysis of Algorithms

• Order of Complexity:
  – Data structures are populated with $n$ objects
  – As $n$ grows, operations on the data structure take longer
  – How much longer?

• Function of “$n$”
  – Linear?
  – Exponential?
  – Quadratic?

• It obviously depends on the algorithm…

• It can depend on the data set:
  – If all members of $n$ are odd, then $O(n^2)$ but
  – If all members of $n$ are even, then $O(n \log n)$
## Rates of Growth

<table>
<thead>
<tr>
<th>n</th>
<th>constant</th>
<th>logarithmic</th>
<th>linear</th>
<th>quadratic</th>
<th>cubic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>O(1)</td>
<td>O(log N)</td>
<td>O(N)</td>
<td>O(N log N)</td>
<td>O(N^2)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>1,024</td>
<td>1</td>
<td>10</td>
<td>1,024</td>
<td>10,240</td>
<td>1,048,576</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1</td>
<td>20</td>
<td>1,048,576</td>
<td>20,971,520</td>
<td>10^{12}</td>
</tr>
</tbody>
</table>
Rates of Growth (Graphed)

- Linear -- $O(n)$
- Quadratic -- $O(n^2)$
- Cubic -- $O(n^3)$
- Logarithmic -- $O(\log n)$
- Exponential -- $O(2^n)$
- Square root -- $O(\sqrt{n})$

From: http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/index.html
Expected Performance

• Most algorithms have Upper and Lower bounds

• Big-Oh ($O$) is the Upper bound, i.e., worst-expected case
  – $T(n)$ is in $O(f(n))$ IFF:
    • Positive constants $k$, $n_0$
    • $|T(n)| \leq |k(f(n))|$ for all $n \geq n_0$

• Omega ($\Omega$) is the Lower bound, i.e., best-case
  – $T(n)$ in $\Omega(f(n))$
    • Positive constants $k$, $n_0$
    • $|k(f(n))| \leq |T(n)|$ for all $n \geq n_0$

• There is also Theta ($\theta$), “Tight bound”
  – Both Upper and Lower bounds have same performance expectation
  – I.E., $T(n)$ is in $\theta(f(n))$ IFF:
    • $T(n)$ is in $O(f(n))$ AND
    • $T(n)$ in in $\Omega(f(n))$