Week 3 - Monday

COMP 2100

Last time

- What did we talk about last time?
- Computing Big Oh
- Logarithms

Questions?

Project 1

Bitmap Manipulator

Assignment 2

Quiz Post Mortem

Big Oh, Big Omega, Big Theta

Formal definition of Big Oh

- Let $f(n)$ and $g(n)$ be two functions over integers
- $f(n)$ is $O(g(n))$ if and only if
  - $f(n) \leq c \cdot g(n)$ for all $n > N$
  - for **some** positive real numbers c and N
- In other words, past some arbitrary point, with some arbitrary scaling factor, $g(n)$ is always bigger

All three are useful measures

- **O** establishes an upper bound
  - $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leq cg(n)$ for all $n \geq N$
- **Ω** establishes a lower bound
  - $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq cg(n)$ for all $n \geq N$
- **Θ** establishes a tight bound
  - $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1, c_2 and N such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N$

Complexity practice

- Give a tight bound for $n^{1.1} + n \log n$
- Give a tight bound for 2^{n+a} where a is a constant
- Give functions f_1 and f_2 such that $f_1(n)$ and $f_2(n)$ are $O(g(n))$ but $f_1(n)$ is not $O(f_2(n))$

Mathematical approaches

- If you can model a segment of code as a series of numbers, a few equations might help you make sense of them
  - Arithmetic series: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ is $\theta(n^2)$
  - Geometric series: $\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}$
    - and the bound depends on *r*
  - Harmonic series: $\sum_{i=1}^{n} \frac{1}{i}$ is $\theta(\log n)$

ADTs

Types

- From a formal perspective, a type is a set of data values and the operations you can perform on them

Type	Values	Operations
`int`	Integers from `-2147483648` to `2147483647`	`+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `>>>`, `|`, `&`
`double`	Floating points numbers	`+`, `-`, `*`, `/`, `%`
`String`	All possible Java `String` objects	`+`, `length()`, `charAt()`, `substring()`, etc.
`Wombat`	All possible `Wombat` objects	`toString()`, `eat()`, etc.

ADTs

- So, you have a type with operations
- Do you need to know how those operations are implemented to be able to use them?
- No!
  - In fact, in OOP (including Java), the data is usually hidden from you
- Enter the **Abstract Data Type** (**ADT**)
  - It does *something!*
  - We aren't necessarily concerned with implementation

Interfaces

- The idea of a Java interface has a strong connection to an ADT
- Let's look at the **`List<E>`** interface
- Some of its methods:
  - **`boolean add(E element)`**
  - **`void add(int index, E element)`**
  - **`void clear()`**
  - **`E get(int index)`**
  - **`int size()`**
  - **`boolean remove(Object o)`**

List implementations

- There are lots of different ways of keeping a list of data
- The **`List`** ADT doesn't care how we do it
- And there are lots of implementations that Java provides:
  - **`ArrayList`**
  - **`LinkedList`**
  - **`Stack`**
  - **`Vector`**
- You can use whichever you think best suits your task in terms of efficiency

Bags

- A **bag** is an ADT that is iterable but otherwise only has one real operation
- **Add**
  - Put an element in the bag
- It's a collection of things in no particular order
- A bag is also called a multiset
- The book talks about bags partly because it's hard to imagine a simpler ADT

Lists

- The **list** ADT is not entirely standardized
  - Some lists allow insertion at the beginning, end, or at arbitrary locations
  - Some lists allow elements to be retrieved from an arbitrary location
- Let's focus on a list that allows the following operations
- **Add**
  - Insert element at the end of the list
- **Add at index**
  - Insert element at an arbitrary location
- **Get**
  - Retrieve element from arbitrary location

Stacks

- A **stack** is an ADT with three main operations
- **Push**
  - Add an item to the top of the stack
- **Pop**
  - Remove an item from the top of the stack
- **Top**
  - Retrieve the item at the top of the stack
- Stacks are often implemented with a dynamic array or a linked list

Queues

- A **queue** is an ADT with three main operations
- **Enqueue**
  - Add an item to the back of the queue
- **Dequeue**
  - Remove an item from the front of the queue
- **Front**
  - Retrieve the item at the front of the queue
- Queues are also often implemented with a dynamic array or a linked list

List Implementation

List implementation

- We're not going to implement the bag ADT since it's very limited
- Instead, we'll focus on the following methods from our list ADT (and a couple of other useful ones)
  - Constructor
  - Add: Insert element at the end of the list
  - Get: Retrieve element from arbitrary location
  - Size: Get the current number of elements stored
- For now, we'll implement the list with a dynamic array that holds generic objects of type `E`
- This is essentially what you've been doing for Assignment 1

Array backed list

```
public class ArrayList<E> {
   private E[] array;
   private int size;

   public ArrayList() {}
   public int  size() {}
   public void add(E element) {}
   public E get(int index) {}
   boolean remove(Object o) {}
}
```

Constructor Implementation

Size Implementation

Get Implementation

Add Implementation

Remove implementation

Upcoming

Next time…

- Stacks
- Keep reading section 1.3

CAREER JUMPSTART EVENT

Engineering & Computer Science

THURSDAY, SEPTEMBER 12TH FROM 4:45PM-7PM

Otterbein University @ The Point

Come and network with alumni and recruitment partners and learn how to be successful with your field.

OTTERBEIN
UNIVERSITY

SCAN the QR CODE to REGISTER

Reminders

- Keep reading section 1.3
- Start on Assignment 2
  - Due Friday by midnight
- Keep working on Project 1
  - Due Friday, September 20 by midnight