Intro to Data Structures

Lecture #9 – ArrayLists & An Intro to Efficiency
September 16, 2014
Mark Stehlik

Outline for Today

- Hours worked in HW1/2
- HW1 ~OK! Discuss Q2
  - 1 did not compile!
  - Average was 81 (Low was 53, but 9 A’s!)
  - Let's look at some code examples (anonymous)
- Test as many boundaries/edge cases as you can,
  - empty, singleton, N
  - beginning, middle, end
- ArrayList operations and Quantifying Efficiency

ArrayList methods

- From java.util.ArrayList (the ArrayList API):
  - list.add(value); //adds value to end of *list*
  - list.add(index, value); //adds value at index
  - list.remove(value); //removes first occurrence of value
  - list.remove(index); //removes element at index (slides down)
  - list.clear(); //removes all elements, *list* is empty
  - list.contains(value); //true if value in *list*; false if not
  - list.get(index); //returns the element at index
  - list.indexOf(value); //returns first index of value; -1 if not
  - list.isEmpty(); //what do you think?
  - list.set(index, value); //sets element at index to value
  - list.size(); //returns the number of elements in *list*

Efficiency (an informal intro)

- Let's look at those ArrayList operations in terms of how many elements need to be accessed (assume the list has *n* elements)…
- But there are a number of ways to look at this:
  - best case
  - average case
  - worst case

Efficiency (more formally)

- We want to measure the performance of an algorithm/method/program
- We need a way to do this that is independent of machine, OS, programming language
- So we count how the number of operations varies wrt (as a function of) the size of the input
- What's an operation?
  - list access, comparison, whatever is innate to the algorithm being investigated

Efficiency (more formally)

- And how do we characterize these functions (operations as a function of input size, n)?
  - Takes the same amount of time regardless of n
    - "constant"
  - Takes time proportional to the size of the input (e.g., double the input, double the time [on same device])
    - linear
  - Takes time proportional to the square of the input
    - quadratic (an example?)

Efficiency (more formally)

- Big O is a notation to capture these function "families"; in increasing order of "time"
  - $O(1)$ constant
  - $O(\log n)$ logarithmic (double n → 1 more operation)
    usually involves halving the problem
  - $O(n)$ linear (double n → double the time)
  - $O(n^2)$ quadratic (triple n → 9x the time)
    usually a linear operation nested inside another linear operation
  - $O(2^n)$ exponential
  - $O(n!)$ factorial

Efficiency (more formally)

- Big O describes "bounding relationships"
  - let's look at some graphs
  - ignore constant factors, low-order terms
  - limit definition (let's get mathy):
    - $g(n) \in O(f(n))$ if limit of $g(n)/f(n) = c$ as $n \rightarrow \infty$
    - i.e., in the limit, the two functions differ by no more than a constant factor
  - think of Big O as "<=", i.e.,
    - $N^2 + 5n - 1000$ is $O(n^2)$
    - $N + 10000$ is $O(n)$; [turns out it's also $O(n^2)$ and $O(n^3)$, but we try to keep the bound as tight as possible]

Efficiency (more formally)

- As an aside, there's a whole family:
  - little o (o), which can be thought as "<"
    - limit = 0
  - big Omega (Ω), which can be thought as ">="
    - limit > 0
  - little omega (ω), which can be thought of as ">"
    - limit = ∞
  - and theta (Θ), which can be thought of as "= ="
    - $g(n) \in O(f(n))$ and $f(n) \in O(g(n))$

Efficiency (time – why it matters)

n	O(1)	$O(\log_2 n)$	O(n)	$O(n^2)$
1000	10 ns	100 ns	10 µs	10 ms
1 million	10 ns	200 ns	10 ms	3 hours
1 billion	10 ns	300 ns	10 s	300 years