CSC263

sean.ryan

January 2023

1 Review

Abstract Data Types	**Data Structures**
specification	implementation
objects	data
operations	algorthims

Analysis - Runtime/complexity

- Worst-Case — Upper Bounds - O
- Best-Case — Lower Bounds - Ω
- ??? — Tight Bounds - Θ

When analyzing an algorithm, we are counting by steps. Steps are represented by any constant time operations, such that

$$t_A(x) = Number\ of\ constant\ operations$$

To be able to prove an upper bound, you need to compare two functions: usually, it's $t_A(x)$ compared to runtime. but runtime can have different values depending on the input size, and the order of the input, and that's where worst and best-case scenarios come into play. For a worst-case analysis, you take the largest possible value of runtime for a given input size, and the best case takes the smallest. can we use this fact in proving the tight bound?

Average-case running time
for each n, $s_n = all\ inputs\ of\ size\ n$ if we consider inputs to be random, S_n = a sample space.

we'll want a discrete probability distribution on S_n. For each x$\in$S$_n$, pr(x)

$$t(x) = steps\ on\ input\ x$$

Where x is a random variable.

Average-case: T(n) = E(t) = $\Sigma_{x \in S_n} t(x) * Pr(x)$

EX: LinSearch(L, x):

number L is a linked list (precondition).

z =L.head (the first node)

while z != None and z.data != x:

    z = z.next

return z

Average-case runtime? S_n = ? with probability distribution? (aka Pr(x) t(x)?

Here, we need an exact expression for t(x). There is a lot of different ways to count steps, but they would all be within a constant factor of each other. The trick we're going to do here is *choose some key operations s.t counting only these operation is within a constant factor of total time then set t(x) = number of these key operations*

For LinSearch, the key operation is `z.data != x`

S_n =? what about S_3 =?

Consider the following two inputs: $input_1 = ([1, 2, 3], 2)$ and $input_2 = (["a", "b", "c"], "b")$

these two inputs would take the same amount of steps, therefore the specific elements on the input don't matter. INSIGHT: Only need one input for each possible value of t. $S_3 = ([1, 2, ..., n], 1), ([1, 2, ..., n], 2), ([1, 2, ..., n], n), ([1, 2, ..., n], 0)$

(note, you must include 0 or a value which represents the function failing, as it must be incorporated into the runtime. If you didn't, then you would be estimating the runtime assuming every input is valid.

Assuming every probability was $1/n + 1$ then the probability decreases as n gets larger.

$$T(n) = \Sigma L, i) \in S_n t(L, i) * Pr(L, i) = (1/n + 1)$$

2 Priority Queues, Data structures, & Heaps

Priority Queues note: priority is in an ascending order; the higher the number, the higher the priority

- collection of elements
- each elem. x has priority (x.priority)

- Operations are the following:
  - insert(q, x): add q to x (note: x.priority can be non-unique)
  - max(q): return an element with the highest priority (q is unchanged)
  - extract-max(q): remove and return elem with the highest priority (changes q)

Data structures note: all worst-case time

unsorted array/list

- insert() – O(1) (strong suit)
- max() – $\Omega(n)$
- extract-max() – $\Omega(n)$

unsorted array/list

- insert() – (can use binary search, still) $\Omega(n)$
- max() – O(1) (strong suit)
- extract-max() – O(1) (strong suit)

lets see how we can compare these runetimes to max heaps

max-heaps, intuition: partially sorted

- structure: "almost-complete" binary tree, every level full, except maybe last. on the last level, all leaves are as far left as possible
- Implementation detail: in practice when you have an almost complete binary tree, the way that this is stored in memory is as an array, level by level. It isnt traversed like a normal tree, but is stored by level.
- navigation: for each node at index i, then the parent(i) = floor(i/2), left(i) = 2i, right(i) = 2i + 1
- note: since heaps are stored in an array, it will usually have more slots then items in the heap. therefore you need to keep track of the size. (heap.size = n)
- MAX HEAP ORDER: this is not a binary search tree. every non-leaf node stores an elem with priority ¿= the priorities of the elements in the node's children. NO PARTICULAR ORDERING REQUIRED AMONG THE CHILDREN