Data Structures with C++ : CS189

Lecture 9-1: Search and Sort

Note on Class Structure

- Conceptually, this should be Thursday and the test should be Tuesday
  - This is an algorithm and the test splits data structures and algorithms
- But I hate when tests are on the first class in a week
  - Feels like an ambush, especially when someone misses the week before

Sorting

Sorting - The Classic Algorithm

- In this "Data Structures and Algorithms" class, we've done a bunch of Data Structures
- The first algorithm we'll look at is the "Hello World" of algorithms
- We want: To make a set of data be in order
  - That's the overall topic, there are actually around 30 ways to do it

Insertion Sort

- "Look at each item in our right hand, and flip through our left hand to find its correct spot."
  - For each of these variations, I use the analogy of my left hand being sorted and my right hand being unsorted, and homework assignments as the data
- This is actually what humans do
  - Grab the top one on the right
  - Look at the name
  - Flip through the left pile and insert it in the right spot

Insertion Sort: O(n^2)

- For each piece of data, we may have to compare it to each piece of data
  - A nested loop is n^2
- In Big O notation, we don't actually care that each entry on average only needs to search half of the left hand
  - n * n/2 + n is still O(n^2)
    - That's right times left plus insertion overhead
  - Worst case is a pre-sorted array where every right actually does compare against every left

Selection Sort

- "Find the lowest item on the right, and make it the next item on the left"
  - Not very human. Search the pile, find "Bob" is next, pick that one up with the left.
- Still have every entry comparing against every entry
  - Still ignore all the smaller constants when you say this too is O(n^2)
  - Small benefit over Insertion in that the left hand doesn't ever have to be moved around

Shell Sort

- Insertion Sort is slow because if you pick what will be the last entry, it has to flip through all of left before it finds out it is last.
- "Instead of checking every entry on the left, flip more than one so you move along quicker."
  - Very human. With 50 items on the left, you flip through big chunks to find the spot
  - How far? 701, 301, 132, 57, 23, 10, 4, 1. Try to jump 701 forward, then 301, etc
  - Not kidding. Supercomputer figured out this was the best and got O(n^(4/3))

Heap Sort

- "A priority queue always gives you the lowest item. Just throw all the data in one and pop it back out."
- You have n items to sort. PQ's promise O(log n) performance.
  - That gives O(n log n)
  - There is a tree inside, sorta. It's a Heap
- O(n log n) is actually the best we can get. Other algorithms just give better O(n log n)
  - And if you had to write it from scratch, it is super easy since PQ is in STL. Push push push pop pop pop

Merge Sort

- "Split the data in half. Sort each half recursively and then combine the two halves back together."
  - Recursive. Base case is one side has only one element.
- When you hear "half" think "tree" and therefore "log n"
- Recombining halves is n, not n^2, since you only look at the front of the two halves
- O(n log n) again. n items compare against log n items since size shrinks

Quick Sort

- "Mergesort is neat, but instead of just splitting in the middle, let's think about where to split first."
  - The best O(n log n) you can get, so everyone (and STL) just uses this one
- Big question is how do you make that decision? (Called picking the pivot.)
  - First? Last? Random? Average of three random picked entries? Average of five?
- While "MergeSort with a pivot" is the definition of QSort, there are many different implementations
  - That's the homework.

One Version of Quick Sort

- Pick first item as Pivot
- Compare Pivot to item to right
- If item to right is smaller, swap it with Pivot
- If item is greater or equal, swap with the last item we haven't touched this round
  - So future items that are greater aren't sent to the end, they are sent to end-1, then end-2, etc
- When the Pivot has processed everything, Quick Sort the two sides
  - Extra bonus, the pivot will never move again

Other Versions of Quick Sort

- First, you can pick a different pivot
- If you are worried about presorted data since that is the worst case, you can make the first step be to shuffle the data
- You can change how you do the swapping
  - I think the way I have here is easy to understand, but other methods will have fewer swaps
    - Me: "Pick a pivot, then move everything smaller to the left and everything bigger to the right."
- You can change the base case
  - Instead of stopping when one side is 1 item, stop recursing if it is 4, and then just switch to Insertion or Selection Sort

Functors Reminder

- All of the searches above do comparisons between data
- So if you want to sort a bunch of Student objects, you need to write a functor again
  - A struct inside Student
  - Overloads the parenthesis operator
  - Pass one in to the sorter so it knows what "in order" even means
  - Don't just override less than, because then you can't sort in different ways later

Searching

Linear Search

- Now that everything is in order, the second classic algorithm is searching for something inside
  - All of these only work if the data is already sorted
- Simplest is "Start at the lowest and keep checking until you get to the end"
- O(n), because, well, obvious. You touch every element

Binary Search

- Exactly the same as our tree's Contains, if we first flattened the tree in to a vector
- "Look at the middle (root). If the item we want is smaller, look left. Otherwise look right."
  - When you think tree or halving, think O(log n)

M-ary Search

- "Binary" means two. "M" can be anything
- Instead of tree nodes having left and right, make them have Many pointers out
  - Then do a slightly more complicated version of Contains
- The result is taking a tall binary tree and making it shorter but wider
  - Big O doesn't change, since the shallower tree is offset by the extra effort of picking which pointer out to use

Caching

- Keep track of how often things are searched for
  - Most recent? Most often? Highest priority?
- Build a side list that only holds a small number of items. Keep the "best" data in there based on your criteria above
- Every time you go to search, search the side list first
  - If it is there (a hit), you have O(1). Constant time to search exactly ten items
  - On a miss, you just wasted a little time and now you have to go to the tree anyway
  - O(log n + 10) doesn't change Big O.

End

Study->Get A->Graduate->Job->Money->Win!