CMPSCI 187: Programming With Data Structures

Lecture #35: Selecting and Searching
David Mix Barrington
5 December 2012

Selecting and Searching

- The Selection Problem
- Finding Maxima or Minima
- Finding Maxima *and* Minima
- The Quick Select Algorithm
- A Review of Searching Methods
- O(1) Search -- Dedicated Slots for Keys
- Introduction to Hashing

The Selection Problem

- The **selection problem** is to take a collection of elements from some ordered type and an index k, and to return the k'th smallest element in the collection. Sorting the collection is equivalent to solving the selection problem for all possible indices, but we will consider the difficulty of individual selection problems for unsorted collections. If k = 1 we are finding the **minimum**, if k = n the **maximum**, and if k = n/2 the **median** element.

- As with comparison-based sorting, we will measure the difficulty of selection by the **number of comparisons** needed to determine the k'th smallest object. Clearly we don't need more than O(n log n) for any k, since that is enough to sort the collection completely.

- The median element would be the best possible pivot for Quick Sort, but even with the best selection algorithms the extra effort to find the median outweighs the advantage from using it.

Finding Maxima or Minima

- Finding the maximum or minimum can be likened to a sports tournament, where we have n teams and want to find an undisputed champion. We could hold a **binary tournament** (as in college basketball) where we group the items into pairs, compare each pair, group the winners from each pair into new pairs, compare them, and so on, halving the number of remaining competitors each time until there is only one.

- A **ladder tournament** compares two items, compares the winner against a third, the winner of that against a fourth, and so on until all items have been compared. Both the binary and ladder are **single elimination tournaments**.

- It's easy to see that either of these methods uses exactly n - 1 comparisons, because every item *except for* the eventual winner has been the loser in exactly one comparison. And n - 1 comparisons are necessary, since if there were two "undefeated" items, either might be the true maximum and thus the winner crowned by the algorithm would be wrong in the worst case.

Finding Maxima *and* Minima

- Suppose we want to conduct comparisons to determine both the maximum and the minimum of the same collection. Clearly we could use n - 1 comparisons to find the maximum and another n - 1 to find the minimum, for 2n - 2 total.

- But there is a better way. Group the n elements into pairs (assume that n is even) and compare each pair. The maximum must be among the n/2 winners and the minimum among the n/2 losers. We can use one single elimination tournament to find the maximum with n/2 - 1 comparisons, and another to find the minimum with another n/2 - 2, making n/2 + (n/2 - 1) + (n/2 - 2) = 3n/2 - 2 total comparisons.

- In fact this number of 3n/2 - 2 cannot be improved using only comparisons. Proving this takes a more sophisticated **adversary argument**. We can design a system to provide answers to any algorithm's comparisons, so that until 3n/2 - 2 have been used, there are either two undefeated items or two winless items, and the algorithm cannot give an answer correct in the worst case.

The Quick Select Algorithm

- What if we want the k'th smallest item for arbitrary k? The basic idea of Quick Sort gives us an algorithm called Quick Select, which is the best known in the average case. (Like Quick Sort, in the worst case it needs $O(n^2)$ time.)

- Suppose I pick a pivot and compare it against all other items, determining that it is the p'th smallest element. If $k < p$, I now need to select the k'th smallest item from the elements smaller than the pivot. If $k = p$, of course, I am done. If $k > p$, I need to select the $(k - p - 1)$'st element from those larger.

- If my pivot were always in the middle, I would spend at most $(n - 1) + (n/2 - 1) + (n/4 - 1) + ... + 1$ comparisons, which sums to about $2n$. In CMPSCI 311, you may show that the average-case time is $O(n)$. But note that if the pivot is always the maximum or minimum of the remaining elements, you use $O(n^2)$.

- Also in CMPSCI 311, you may see a more sophisticated algorithm that always selects the k'th smallest element with $O(n)$ comparisons, for any k.

A Review of Searching Methods

- We've seen a number of ways to store a collection, and for each we have looked at the time needed to **find** a given element. This is the fundamental operation behind the `contains`, `get`, and `remove` operations.

- An unsorted list requires us to make a **linear search**, taking O(n) time. With a sorted list, we could use **binary search**, taking O(log n) time, as long as we have random access to the elements.

- In order to get O(log n) time in a collection where we can easily insert and delete, we looked at **binary search trees**, which allow insertion, deletion, and finding in O(log n) time each *if they are balanced*. (There are self-balancing BST's, which you will see in CMPSCI 311.)

- Can we do better? Could we insert, delete, and find all in O(1) time?

O(1) Searching -- Dedicated Slots for Keys

- DJW give the example of a small company where the employee ID's range from 0 to 99 and the HR department knows everyone's ID without having to look it up. They can keep the employee record objects in an array, and find it by using the ID as an index.

- The general problem of finding an object from a **key** is simplified if the **key space** is small, as in this instance. If we can afford an array with one index for each key, all our operations become easy. We test whether a key is in use by checking that key's slot for a null entry. We add a key by replacing a null entry with a real one. We delete an key by replacing that entry with a null one.

- The problem is, of course, that our key space is usually uncomfortably large. UMass keeps student records by eight-digit ID's, so using this system would require an array with 100,000,000 entries, only a small fraction of which would be used -- fewer than a million students have *ever* attended UMass.

Introduction to Hashing

- **Hashing** is a technique to simulate the dedicated-slot method in the general situation where only a small fraction of the possible keys are used.

- In the general hashing situation we have a large set of keys and a smaller set of indices for our array. We choose a **hash function**, which takes any key and produces an index. The simplest hash function uses the integer remainder operator % -- if we have m different indices, then for any key k the hash function produces the index k % m, which is in the range from 0 through m - 1.

- Suppose for a moment that on the subset of the keys that we use, our hash function is what CMPSCI 250 will call a one-to-one function. No two different keys in use are ever mapped to the same index. In that case, we can use the array just as we used the dedicated-slot array, inserting, deleting, and finding elements in O(1) time with an array of size m instead of the size of the key space.

- Of course in the worst case we cannot avoid **collisions**, where two different keys in use are hashed the same index. We'll deal with this in the next lecture.