Computer Science 2530
April 3, 2020

Happy Friday, April 3.

Now that we have gotten our feet wet, let's do a tiny bit of analysis of algorithms.

Lecture notes page **36B** has two exercises. See if you can understand how much time the functions in those exercises take as a function of the length of string s. Expression strchr(vowels, $s[i]$) returns a non-null pointer if character $s[i]$ occurs in string vowels, and returns a null pointer if not. Since vowels has a fixed length, it takes $\Theta(1)$ time (constant time) to do that call.

Linear search

Read page **36C** in the notes. It starts by looking at *linear search.*

Imagine that you need to look up someone's name in a telephone book. The linear search algorithm says to start at the first name and check each name in succession, name by name and page by page, until you either find the name that you are looking for or reach the end of the telephone book. That is obviously painfully slow. If there are n names in the telephone book, it takes time proportional to n to finish. That is, it takes time $\Theta(n)$.

Binary search

But telephone books are in alphabetical order, and you can do much better because of that. You can open the book to about the middle, and compare the name L that you are looking for to the name N that you see there. There are three possibilities.

1. You have been very lucky and $L = N$. You can stop the search.
2. L comes before N in alphabetical order. So you can restrict your search to the part of the phone book before N. Let's call that the remaining part to search.
3. L comes after N in alphabetical order. So you can restrict your search to the part of the phone book after N.

In cases 2 and 3, the number of names in the remaining part of the telephone book has been cut in half.

Suppose that you start with n names in the phone book. Initially, the part to search is the entire book of n names. You look at just one name and cut the number of names left in the remaining part in half. Then you look at another name, in the middle of the remaining part, and cut the total remaining in half again. That leads to the idea of ***binary search***.

If you write down the number of names left in the remaining part, it goes rougly as follows.

$$\begin{array}{l} n \\ n/2 \\ n/4 \\ n/8 \\ n/16 \\ \cdots \\ 1 \end{array}$$

Once there is only one name left, it is either the name that you are looking for or it isn't. In any event, the search is finished.

We have seen that it takes about $\log_2(n)$ halving steps to reach 1 if you start at n. If it takes a fixed amount of time to look at one name, the total time is $\Theta(\log_2(n))$.

Let's compare linear search to binary search. The first column is the time it takes to do a linear search of n values. The second column is roughly the time it takes to do a binary search of those n values, assuming that they are already in order.

n	$\log_2(n)$
10	3
100	7
1000	10
10,000	13
100,000	17
1,000,000	20

Obviously, you would rather do 20 steps than 1,000,000 steps.

Insertions and deletions

Lookups are fast in a sorted list. But insertions and deletions are another issue. You can't just insert a name into a telephone book because there is no room for it. If your telephone book is stored in an array, you need to move entries out of the way to make room for a new one. The cost of inserting an entry or removing an entry is $\Theta(n)$.

Our goal now is to discover a data structure that allows insertion, deletion and lookup all in time $\Theta(\log_2(n))$, where n is the total number of entries in a table.

That will take us into a kind of data structure called a ***binary tree***.