Searching

Contents

- Introduction
- Searching in an Array
- The Comparable Interface

Introduction

Searching is a very common problem, both in life and in computer applications. It is so important that many data structures and algorithms have been designed specifically to make searching as easy and efficient as possible.

In this set of notes we will review the three techniques for searching for a value in an array. Other sets of notes will present several different data structures designed to support efficient searches.

Searching in an Array

Recall that there are two basic approaches to searching for a given value `val` in an array: **sequential search** and **binary search**.

Sequential search involves looking at each value in turn (i.e., start with the value in `array[0]`, then `array[1]`, etc.). The algorithm quits and returns true if the current value is `val`; it quits and returns false if it has looked at all of the values in the array without finding `val`.

If the values are in **sorted** order, then the algorithm can sometimes quit and return false without having to look at all of the values in the array; in particular, if the current value is ***greater*** than `val` then there is no point in looking at the rest of the values in the array (`val` is definitely not there).

The worst-case time for a sequential search in an array of size N is always O(N), even when the array is sorted (though the average-case time should be better for a sorted array than for a non-sorted array).

When the values are in sorted order, a better approach than sequential search is to use

binary search. The algorithm for binary search starts by looking at the middle item `mid`. If `mid` is the value `val` that we're searching for, the algorithm quits and returns true. Otherwise, it uses the relative ordering of `mid` and `val` to eliminate half of the array (if `val` is less than `mid`, then it can't be to the right of `mid` in the array; similarly, if it is greater than `mid`, it can't be to the left of `mid`). Once half of the array has been eliminated, the algorithm starts again by looking at the middle item in the remaining half. It quits when it finds `val` or when the entire array has been eliminated.

The worst-case time for binary search in an array of size N is proportional to $\log_2 N$ (pronounced "log base 2 of N"). The log base 2 of N is the number of times N can be divided in half before there is nothing left. Similarly, the log base 3 of N would be the number of times N can be divided in thirds before there is nothing left. Using Big-O notation, log base anything of N is written O(log N), since the difference between "log base 2 of N" and "log base some other number of N" is just a constant factor.

The Comparable Interface

As discussed above, binary search only works if the values in the array are in sorted order. Values can be arranged in sorted order only if, given two values `a` and `b`, we can answer the question: Is `a` less than `b`? That question can be answered for all numeric values (e.g., `int`, `float`, `double`); we can compare two numeric variables `x` and `y` using the "less-than" operator: `if (x < y) ...`

What about Java `Object`s? Since `Integer`s and `Double`s represent numeric values, it makes sense to be able to compare them; similarly, there is a natural ordering on strings, so it makes sense to be able to compare two `String` objects, too. For `Integer`s and `Double`s, you could convert them to the corresponding `int` / `double` values and use the less-than operator, but Java gives you an easier option (one that also works for `String`s): `Integer`s, `Double`s, and `String`s all implement the `Comparable` interface, which means that they have a `compareTo` method. The `compareTo` method allows you to compare two `Integer`s, two `Double`s, or two `String`s to see if one is less than, equal to, or greater than the other. The `compareTo` method returns a negative value to mean "less than", it returns zero to mean "equal to", and it returns a positive value to mean "greater than".

For example, if variables `S1` and `S2` are both `String`s, then `S1.compareTo(S2)` returns an `int` value that is:

- negative if `S1` is less than `S2`;
- zero if `S1` is equal to `S2`;

- positive if `S1` is greater than `S2`.

When you define a new class you should think about whether it makes sense to ask whether one instance of the class is less than another instance. If the answer is yes, you should define a `compareTo` method and you should make your class implement the `Comparable` interface. If you do that, it will be possible to pass instances of your class to methods that require `Comparable` parameters (for example, Java provides some sorting methods that require `Comparable` parameters).

To make your class implement the `Comparable` interface you must:

1. include `implements Comparable<C>` after the class name in the file that defines the class, and
2. define a `compareTo` method with the following signature:

```
int compareTo(C other)
```

For example, we could define a `Name` class with two fields for the first and last names. It makes sense to consider one `Name` smaller than another if the first one would come first in the phonebook (i.e., its last name comes first in alphabetical order, or the last names are the same and its first name comes first in alphabetical order). Here's a (partial) definition of the `Name` class, including just the fields and the `compareTo` method:

```
public class Name implements Comparable<Name> {
    private String firstName, lastName;

    public int compareTo(Name other) {
        int tmp = lastName.compareTo(other.lastName);
        if (tmp != 0) {
            return tmp;
        }
        return firstName.compareTo(other.firstName);
    }
}
```

Back

Next: Continue with Trees