Design of Information Structures - Lab 4

January 28, 2019

1 Search Techniques

This is the fourth lab of the Design of Information Structures course. The focus of this lab will be the development of data structures which allow us to efficiently search for a piece of data amongst a large array or list. The principle techniques we will use today will include a linear search, binary search and hashing. The following descriptions talk about searching for a known value in a list, which may sound redundant, but is in fact a frequent requirement. The key thing to remember is we are often searching for a key in some lookup table – imagine searching for a known name in a phonebook to find the number, for example..

1.1 An Introduction to Comparators in Java

The task of comparing two elements in Java is very common. The designers of Java have included a special interface called `java.lang.Comparable` which indicates that objects of a particular class can be compared to one another. The `Comparable` interface states that the class should implement a method `compareTo(Object o)` which will compare the object `o` to an object of the class and return either a value less than 1 if the item is smaller than the object, 0 if the object is the same and greater than 1 if the value is greater.

Before we proceed we should check how this works by running a small test program. Copy the files for this laboratory from the cs126 web site (download file `lab4.zip`).

You should compile the `CompareToTest` class and then run this. Check that the `compareTo` method works as you would expect.

If you intend to create a class which is likely to need frequent comparisons then you should consider using the `Comparable` interface since Java provides a large number of library methods to sort and process Objects which implement `Comparable`.

You can find out more about Comparable at:
`http://java.sun.com/javase/6/docs/api/java/lang/Comparable.html`

1.2 Linear Search

A linear search is a search which starts from the beginning of the data structure and proceeds to search each element in turn until the desired element is found or the end of the structure is reached.

Exercise 1 *The first programming task of today's lab is to take the* `SortedArrayList` *class provided and implement the* `findLinear(E element)` *which will find* `str` *within the Vector. You should count the number of comparisons needed to find the value in the list.*

Exercise 2 *Add more elements to the* `SortedArrayList` *and re-run your* `findLinear` *method to find a few of the elements. How does the number of comparisons scale with the number of elements in the list? Is the order of the data important to the number of comparisons?*

Exercise 3 *Now compile the* `SortedArrayList` *class. The current* `add()` *method just calls the super (ArrayList) add method. This means that elements are not currently added in their correct order, but in the order they were added to the list. Fix this, to make sure that the elements are added in their correct (ascending) order.*

Exercise 4 *Write a program to test that the data inserted into the* `SortedArrayList` *class is kept in order. You should add a few items to the* `SortedArrayList` *and then use the* `toString()` *method (implemented in* `ArrayList`*) to check the data is being kept in order.*

Exercise 5 *Perform a linear search through the* `SortedArrayList`*. Count the number of comparisons. Does the ordering of the data affect the number of comparisons needed to find the data?*

Milestone 1) You have now completed the first milestone. You should get this signed off by your lab tutor.

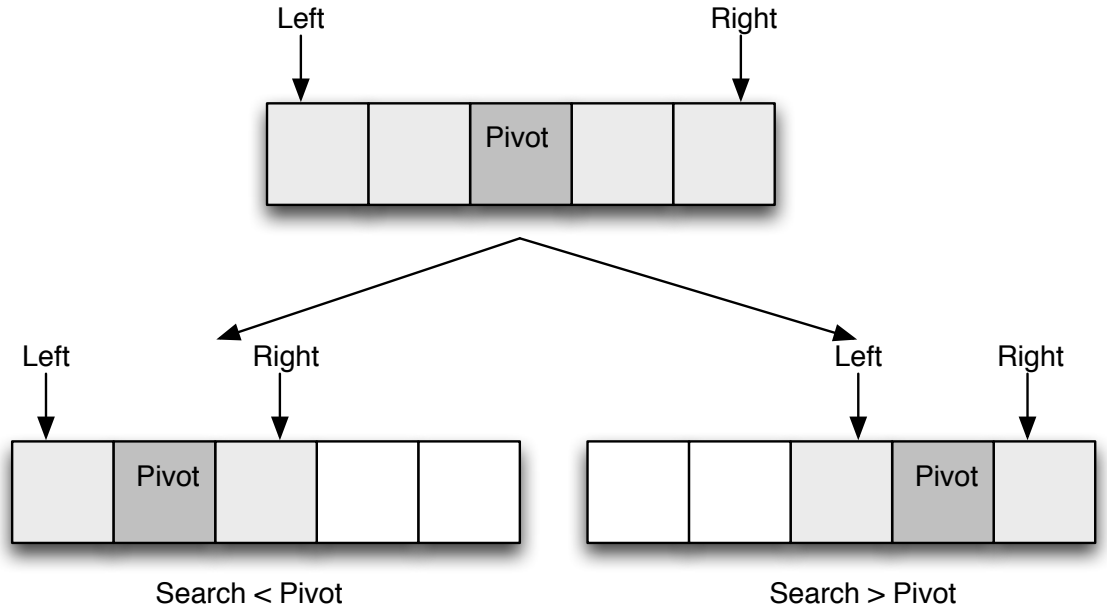

Figure 1: **Binary Search Algorithm**

1.3 Binary Search

A Binary Search is a special technique for searching which relies on the fact that the data is ordered. The search proceeds by finding a special element called the pivot. This is the element half way between the start and end of the list. The pivot element is compared to the data being searched for. If the target data is found at the pivot the search is complete. If this is not the case, we know our target data is ordered either before or after our pivot and so we need only search the appropriate sublist. The search is performed recursively until the item being searched for is found.

For example, if we have a list of integers from 0 to 6 and we search for the value 1 we would perform the binary search as follows - the pivot would be the index $(0+6)/2 = 3$. We would check for the value 1 at index 3. The result of the comparison would be that the data being searched for is less than the pivot so we would search the sublist which is less than the pivot. We would repeat the above search between 0 and 3, halving the search space for each comparrison performed.

Exercise 6 *Create a method called* `findBinary(E element)` *which performs a binary search on the* `SortedArrayList` *class. Ensure you also add code to count the number of comparisons the binary search technique uses.*

Exercise 7 *Add over 10 items to the* `SortedArrayList` *and then attempt to search for*

a number of these items. How many comparisons are needed on average? Is this method faster than the linear search technique developed earlier? In what cases would you use the linear search method? In which cases would you use the binary search method?

Milestone 2) You have now completed Milestone 2. You should get this signed off by your lab tutor.

---

2 Hashing

We have seen two search techniques for storing and retrieving data. Both of these techniques require us to search through a potentially large number of entries before we find the one we are looking for. Wouldn't it be much faster is we could go directly to a small subset of the data to find the item we are looking for? It turns out that it is possible to reduce the number of elements we search through by using a third technique called hashing.

Hashing involves generating a 'code' for a piece of data and then using this code to place the item at a location in memory. An example hash function is given below:

```
public int generateHashCode(String s) {
    int code = 0;

    for(char next_character: s.toCharArray()) {
        code += next_character;
    }

    return code;
}
```

Exercise 8 *Use the code provided above to develop a small tester program. You should generate hashcodes for two or three Strings. Are these codes different? Will these codes always be different or will some be the same?*

2.1 Storing Data using Hashcodes

Now that we can generate hash codes for data we need to consider how exactly we can use these to help us store data more efficiently. One technique is to have an array of say size n. If we want to store data efficiently we can then generate the hash code and use

modulo arithmetic to find a location to place the data. We can find the location to store the data in our by using the following code:

```
int location = hash(key) % n;
```

This technique is very efficient provided we have an efficient hashing function which generate a good distribution of data. However, there is a problem with this approach. What happens if two pieces of data need to be stored at the same location in our array? When this happens, we say that a collision has occurred - i.e. two pieces of data require the same location in memory. The hash function given maps the anagrams 'straw' and 'warts' to the same hash code, for example.

2.1.1 Overcoming Collisions

One way in which we can overcome the problem of collisions is instead of having just an array of data we will use an array of lists of data. We can then use the hashcode to find which list we should search for the item. This helps us to reduce the total number of items we need to search since if we develop a good hash function each list will contain a smaller number of elements that the total data stored in the data structure. This type of data storage structure is known as a hash table.

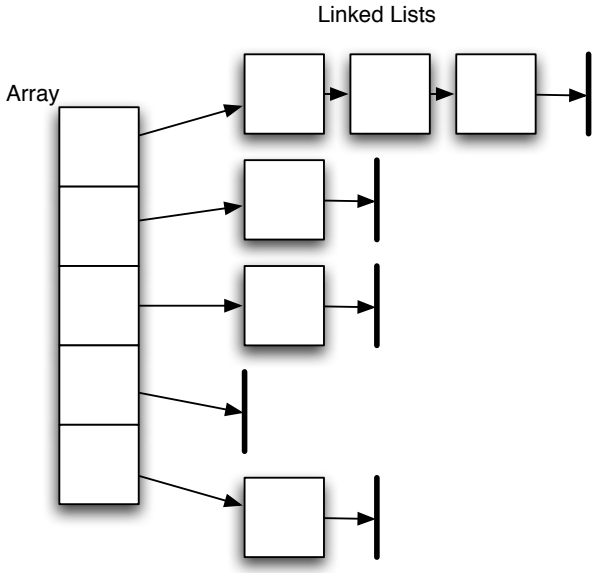

Figure 2: **Hashtable**

Since we are now storing multiple items at each 'location' in the array (in the form of a list) we need some way of remembering which key is associated with which piece of data.

We can link these two pieces of information together by developing a Key Value Pair - i.e. a pair of data items, one key and one value.

Exercise 9 *Load up the* `HashMap.java` *file and analyse how the data is added to the hash table. You will need to look at* `KeyValuePair.java`, `KeyValuePairLinkedList.java` *and other files. You will need to explain to your lab tutor how these classes work together to form the hash table structure.*

Exercise 10 *Add code to count the number of comparisons needed to find a piece of data in the hash table. You will need to add this code to the appropriate files.*

Exercise 11 *Add a large number of Strings to the hash table (say 20 or so). How are the values distributed in the hash table? What is the average number of comparisons needed to find values in the table?*

Milestone 3) You should contact you lab tutor and get this signed off.

Notes: The design an implementation of efficient hashing functions is still a subject of study in algorithmics. The analysis often involves studying the algorithm behaviour under varying types of data and using probability to work out how the data will be structured. It is likely that during your time at Warwick you will have the opportunity to investigate these techniques further.