Hyperiondev

TASK

Algorithms II — Collections Framework

Visit our website

Introduction

WELCOME TO THE ALGORITHMS II — COLLECTIONS FRAMEWORK TASK!

In this task, you will be looking at searching algorithms, as well as some other useful algorithms that are used within the Java Collections Framework.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with a code reviewer. You can also schedule a call or get support via email.

Our expert code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

SEARCHING ALGORITHMS

There are two main algorithms for searching through elements in code, namely **linear search** and **binary search.** Linear search is closest to how we, as humans, search for something. If we are looking for a particular book on a messy bookshelf, we will look through all of them until we find the one we're looking for, right? This works very well if the group of items are not in order, but it can be quite time-consuming. Binary search, on the other hand, is much quicker but it only works with an ordered collection. Let us look and these two in more detail. If you want to have a look at visual representations of any of the searching methods above, have a look **here**.

Linear Search

As mentioned, linear search is used when we know that the elements are not in order. We start by knowing what element we want, and we go through the list comparing each element to the known element. The process stops when we either find an element that matches the known element, or we get to the end of the list. Linear search is O(n) and is executed below (Dalal, 2004):

```
public int sequentialSearch(int item, int[] list) {
    // if index is still -1 at the end of this method, the item is not
    // in this array.
    int index = -1;
    // loop through each element in the array. if we find our search
    // term, exit the loop.
    for (int i=0; i<list.length; i++) {
        if (list[i] == item) {
            index = i;
            break;
        }
    }
    return index;
}
```

Output:

```
List: [10, 2, 23, 14]
Sorted List: [2, 10, 14, 23]
```

Binary Search

Binary search works if the list is in order. If we go back to our book example, if we were looking for *To Kill a Mockingbird* and the books were organised in

alphabetical order, we could simply go straight to 'T', refine to 'To' and so on until we found the book. This is how binary search works. We start in the middle of the list and determine if our sought-for element is smaller than (on the left of) or bigger than (on the right of) that element. By doing this we instantly eliminate half of the elements in the list to search through! We continue to halve the list until either we find our sought-for element, or until there are no elements found that equal to our sought-for element. Let's look at an example:

We are looking for the number 63 in the following list:

3	10	63	80	120	6000	7400	8000

In the list above, the midpoint is between 80 and 120, so the value of the midpoint will be 100 (*(120+80) ÷ 2 = 100*). 63 is less than 100, so we know 63 must be in the left half of the list:

3	10	63	80	~~120~~	~~6000~~	~~7400~~	~~8000~~

Let's half what's left. The midpoint is now 37.5, which is smaller than 63, so our element must be on the right side.

~~3~~	~~10~~	63	80	~~120~~	~~6000~~	~~7400~~	~~8000~~

We're left with our last two elements. The midpoint of 63 and 80 is 71.5, which is greater than 63, so our element must be on the left side.

~~3~~	~~10~~	63	~~80~~	~~120~~	~~6000~~	~~7400~~	~~8000~~

And it is! We've found our element.

Binary search is O(log n) (UCT, 2014) and is written below (Dalal, 2004):

```
public int binarySearch(int item, int[] list) {
    // if index = -1 when the method is finished, we did not find the
    // search term in the array
    int index = -1;

    // set the starting and ending indexes of the array; these will
    // change as we narrow our search
```

```
    int low = 0;
    int high = list.length-1;
    int mid;

    // Continue to search for the search term until we find it or
    // until our ''low'' and ''high'' markers cross
    while (high >= low) {
        mid = (high + low)/2; // calculate the midpoint of the current array
        if (item < list[mid]) { // value is in lower half, if at all
            high = mid - 1;
        }
        else if (item > list[mid]) {
            // value is in upper half, if at all
            low = mid + 1;
            }
        else {
                // found it! break out of the loop
            index = mid;
            break;
        }
    }
    return index;
}
```

It is worth noting that the Java Collections Framework uses the binary search algorithm in the form of `Collections.binarySearch()`. This means that you need to sort your list before you search it. See the example below:

```
import java.util.*;

public class BinarySearch{
   public static void main(String[] args) {

      List<String> list = new ArrayList<>();
      list.add("Java");
      list.add("Programming");
      list.add("Is");
      list.add("Fun");

      // Sort the list into alphabetical order
      Collections.sort(list);
      System.out.println("Sorted List: " + list);

      int index = Collections.binarySearch(list, "Java");
      System.out.println("'Java' in List is at " + index);
```

```
        index = Collections.binarySearch(list, "Python");
        System.out.println("'Python' in List is at " + index);
    }
}
```

Output:

```
Sorted List: [Fun, Is, Java, Programming]
'Java' in List is at 2
'Python' in List is at -5
```

Note what the output prints as 'Python's' index: -5. This is because if you were to insert this element into the list, it would have the index of 4. It sounds strange but it is actually calculated using *(-(insertion_index) - 1)*, which is *(-(-5) -1)*.

SHUFFLING

The shuffle algorithm does the complete opposite of what the sort algorithm does. While the sort algorithm puts the elements of a list in some sort of order, the shuffle algorithm destroys all traces of order that may be present in the list. In other words, the shuffle algorithm gives us a random permutation of the elements in a list. This is useful if you would like to create a program that implements a game of chance, such as a card game. It can also be useful for generating test cases.

There are two shuffle methods provided in the Java Collections Framework:

- `shuffle(List list)` — takes in a List object as an argument and uses a default source of randomness
- `shuffle(List list, Random rnd)` — requires a **Random** object to use as a source of randomness

The following code is used to shuffle a number of words in a list:

```
import java.util.*;

public class Shuffle {
   public static void main(String[] args) {

    List<String> list = new ArrayList<>();
    list.add("Java");
    list.add("Programming");
    list.add("Is");
    list.add("Fun");
```

```
    System.out.println("Original List: " + list);

    Collections.sort(list);
    System.out.println("Sorted List: " + list);

    Collections.shuffle(list);
    System.out.println("Shuffled List: " + list);

    Collections.shuffle(list);
    System.out.println("Shuffled List: " + list);
   }
}
```

Output:

```
Original List: [Java, Programming, Is, Fun]
Sorted List: [Fun, Is, Java, Programming]
Shuffled List: [Programming, Is, Fun, Java]
Shuffled List: [Is, Fun, Java, Programming]
```

Notice that the two calls of shuffle give you two different shuffled versions of the list.

COMPOSITION

The Collections Framework contains two algorithms to test some aspect of the composition of one or more collections:

- `int frequency(Collection c, Object o)` — returns the number of times a specific object occurs in the given collection.
- `boolean disjoint(Collection c1, Collection c2)` — determines whether the two specified Collections contain no elements in common.

FINDING EXTREME VALUES

The Collections Framework contains two algorithms, min and max, that allow you to find the minimum or maximum element in a Collection.

- `max(Collection c)` — Takes in a collection and returns the maximum element according to the elements' natural ordering.
- `min(Collection c)` — Takes in a collection and returns the minimum element according to the elements' natural ordering.

Compulsory Task 1

Answer the following questions:

- Design a class called Course. The class should contain:
  - The data fields courseName (String), numberOfStudents (int) and courseLecturer (String).
  - A constructor that constructs a Course object with the specified courseName, numberOfStudents and courseLecturer.
  - The relevant get and set methods for the data fields.
  - A toString() method that formats that returns a string that represents a course object in the following format: (courseName, courseLecturer, numberOfStudents)
- Create a new ArrayList called courses1, add 5 courses to it and print it out.
- Sort the List according to the numberOfStudents and print it out.
- Swap the element at position 1 of the List with the element at position 2 and print it out.
- Create a new ArrayList called courses2.
- Using the addAll method add 5 courses to the courses2 List and print it out.
- Copy all of the courses from courses1 into courses2.
- Add the following two elements to courses2:
  - (Java 101, Dr. P Green, 55)
  - (Advanced Programming, Prof. M Milton, 93)
- Sort the courses in courses2 alphabetically according to the course name and print it out.
- Search for the course “Java 101” in courses2 and print out the index of the course in the List.
- Use the disjoint function to determine whether courses1 and courses2 have any elements in common and print out the result.

- In courses2, find the course with the most students and the course with the least students and print each out.

Compulsory Task 2

Answer the following question:

- Create a Java file called **chores.java**. Inside, write a Java lot-drawing program that allows the user to enter a list of names and a list of chores (equal lengths).
- The program should assign each person a random chore.
- Print out the pairs (i.e. name + chore)

Completed the task(s)?

Ask an expert to review your work!

Review work

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References:

Adamchik, V. (2009). Sorting. Retrieved 25 February 2020, from https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html

Dalal, A. (2004). Searching and Sorting Algorithms. Retrieved 25 February 2020, from http://www.cs.carleton.edu/faculty/adalal/teaching/f04/117/notes/searchSort.pdf

University of Cape Town. (2014). Sorting, searching and algorithm analysis — Object-Oriented Programming in Python 1 documentation. Retrieved 25 February 2020, from https://python-textbok.readthedocs.io/en/1.0/Sorting_and_Searching_Algorithms.html