19.2 Linear Search

Looking up a phone number, finding a website via a search engine and checking the definition of a word in a dictionary all involve searching large amounts of data. This section and Section 19.4 discuss two common search algorithms—one that’s easy to program yet relatively inefficient (linear search) and one that’s relatively efficient but more complex to program (binary search).

Linear Search Algorithm

The **linear search algorithm** searches each element in an array sequentially. If the search key does not match an element in the array, the algorithm tests each element, and when the end of the array is reached, informs the user that the search key is not present. If the search key is in the array, the algorithm tests each element until it finds one that matches the search key and returns the index of that element.

As an example, consider an array containing the following values

```
34  56  2  10  77  51  93  30  5  52
```

and a program that’s searching for 51. Using the linear search algorithm, the program first checks whether 34 matches the search key. It does not, so the algorithm checks whether 56 matches the search key. The program continues moving through the array sequentially, testing 2, then 10, then 77. When the program tests 51, which matches the search key, the program returns the index 5, which is the location of 51 in the array. If, after checking every array element, the program determines that the search key does not match any element in the array, it returns a sentinel value (e.g., `-1`).

Linear Search Implementation

Class `LinearSearchTest` (Fig. 19.2) contains `static` method `linearSearch` for performing searches of an `int` array and `main` for testing `linearSearch`.

```
// Fig. 19.2: LinearSearchTest.java
// Sequentially searching an array for an item.
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.Scanner;

public class LinearSearchTest {
   // perform a linear search on the data
   public static int linearSearch(int data[], in
      // loop through array sequentially
      for (int index = 0; index < data.length; i
         if (data[index] == searchKey) {
            return index; // return index of int
            }
        }

        return -1; // integer was not found
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        SecureRandom generator = new SecureRandom(

        int[] data = new int[10]; // create array

        for (int i = 0; i < data.length; i++) { //
            data[i] = 10 + generator.nextInt(90);
        }

        System.out.printf("%s%n%n", Arrays.toStrin

        // get input from user
        System.out.print("Please enter an integer
        int searchInt = input.nextInt();

        // repeatedly input an integer; -1 termina
        while (searchInt != -1) {
            int position = linearSearch(data, searc

            if (position == -1) { // not found
                System.out.printf("%d was not found%
            }
            else { // found
                System.out.printf("%d was found in p
                    searchInt, position);
            }

            // get input from user
            System.out.print("Please enter an integ
            searchInt = input.nextInt();
        }
    }
}
```

```
[59, 97, 34, 90, 79, 56, 24, 51, 30, 69]
Please enter an integer value (-1 to quit): 79
79 was found in position 4
Please enter an integer value (-1 to quit): 61
61 was not found
Please enter an integer value (-1 to quit): 51
51 was found in position 7
Please enter an integer value (-1 to quit): -1
```

Fig. 19.2

Sequentially searching an array for an item.

Method `linearSearch`

Method `linearSearch` (lines 9–18) performs the linear search. The method receives as parameters the array to search (`data`) and the `searchKey`. Lines 11–15 loop through the elements in the array `data`. Line 12 compares each with `searchKey`. If the values are equal, line 13 returns the *index* of the element. If there are *duplicate* values in the array, linear search returns the index of the *first* element in the array that matches the search key. If the loop ends without finding the value, line 17 returns `-1`.

Method `main`

Method `main` allows the user to search an array. Lines 24–28 create an array of `10 int`s and populate it with random `int`s from `10`–`99`. Then, line 30 displays the array's contents using `Arrays static` method `toString`, which returns a `String` representation of the array with the elements in square brackets (`[` and `]`) and separated by commas.

Lines 33–34 prompt the user for and store the search key. Line 38 calls method `linearSearch` to determine whether `searchInt` is in the array `data`. If it's not, `linearSearch` returns `-1` and the program notifies the user (line 41). If `searchInt` is in the array, `linearSearch` returns the position of the element, which the program outputs in lines 44–45. Lines 49–50 get the next search key from the user.

Generating Arrays of Random Values with Java SE 8 Streams

8

Recall from Section 17.14 that you can use `SecureRandom` method `ints` to generate streams of random values. In Fig. 19.2, lines 24–28 can be replaced by

---

```
int[] data = generator.ints(10, 10, 91).toArray();
```

The first argument to `ints` (`10`) is the number of elements in the stream. The second and third arguments indicate that the random values `ints` produces will be in the range `10` up to, but not including, `91` (that is, `10`–`90`). On the resulting `IntStream`, we call `toArray` to get an `int` array containing the `10` random values. In this chapter's subsequent examples, we'll use this streams-based technique, rather than `for` statements to populate the arrays. We'll still use `for` statements to enable us to create visual outputs showing how the searching and sorting algorithms work.