Searching Algorithms

Locating an element from within a data structure

Alwin Tareen

Searching

The process of searching

- Often, programmers work with large amounts of data stored in arrays.
- It may be necessary to determine whether an array contains a value that matches a certain key value.
- The process of locating a key value in an array is called **searching**.

Linear Search

A description of the linear search algorithm

- The linear search algorithm begins searching at the beginning of the array.
- It compares the search key value(the value that you are looking for) with the first item in the array.
- If there is a match, then it stops the search.
- If there is not a match, then the second item in the array is examined.
- The process is repeated until a match is found, or the end of the array is reached.

Linear Search, Determine `key` Existence

- The following `Java` program performs a linear search on an array of integers.
- If the key value is found in the array, then the method returns `true`, otherwise it returns `false`.

The BooleanSearch Class

```
public class BooleanSearch
{
    public static boolean linearSearch(int[] arr, int key)
    {
        for (int i = 0; i < arr.length; i++)
        {
            if (arr[i] == key)
            {
                return true;
            }
        }
        return false;
    }
```

The BooleanSearch Class, Continued

```
    public static void main(String[] args)
    {
        int[] samples = {93,24,85,72,30,23,15,36};
        int query = 30;
        boolean result = linearSearch(samples, query);
        System.out.println("Found search key? " + result);

        query = 57;
        result = linearSearch(samples, query);
        System.out.println("Found search key? " + result);
    }
}
```

Linear Search, Determine `key` Location Index

- Sometimes, it is useful to know where in the array the item was found. In other words, the specific index.
- A return value of `-1` indicates that the searched item was not found in the array.

The LocationSearch Class

```
public class LocationSearch
{
    public static int linearSearch(int[] arr, int key)
    {
        for (int i = 0; i < arr.length; i++)
        {
            if (arr[i] == key)
            {
                return i;
            }
        }
        return -1;
    }
```

The LocationSearch Class, Continued

```
    public static void main(String[] args)
    {
        int[] samples = {93,24,85,72,30,23,15,36};
        int query = 30;
        int result = linearSearch(samples, query);
        System.out.println("Search key at: " + result);

        query = 57;
        result = linearSearch(samples, query);
        System.out.println("Search key at: " + result);
    }
}
```

Linear Search with `Strings`

- A linear search can also be used to search for a `String` within an array.
- However, you must remember to compare the `Strings` using `.equals()`

The StringSearch Class

```
public class StringSearch
{
   public static int linearSearch(String[] arr, String key)
   {
       for (int i = 0; i < arr.length; i++)
       {
           if (arr[i].equals(key))
           {
               return i;
           }
       }
       return -1;
   }
```

The StringSearch Class, Continued

```
    public static void main(String[] args)
    {
        String[] samples = {"cat", "dog", "mouse", "bird"};
        String query = "mouse";
        int result = linearSearch(samples, query);
        System.out.println("Search key at: " + result);

        query = "lizard";
        result = linearSearch(samples, query);
        System.out.println("Search key at: " + result);
    }
}
```

Linear Search Algorithm Analysis

Determining the time efficiency

- Computer scientists talk about the **efficiency** of an algorithm in terms of its best, average, and worst case runtime.
- The **best case** occurs when the data is organized in such a way that the algorithm works at its peak performance, or fastest.
- The **average case** occurs when the data is organized in such a way that the algorithm works at its average speed.
- The **worst case** occurs when the algorithm is least efficient, or works at its slowest speed.

Linear Search Algorithm Analysis

Time efficiency of the worst case

- The **worst case** is often the one that most people examine when analyzing an algorithm, because it gives you the best guaranteed performance of the algorithm.

Analyzing the time efficiencies

- Assume that `n` represents the size of the array to be searched.
- The linear search algorithm has the following time efficiencies(see next slide).

Linear Search Algorithm Analysis

Best case

- This is when the item to be searched is the first item in the array.

Worst case

- When the item to be searched is at the end of the array, or it is not in the array.
- The search makes `n` comparisons before it determines that the item is not in the array.

Average case

- When the item to be searched is in a random location. The search makes on average `n/2` comparisons before it locates the item.

Binary Search

A more efficient search algorithm

- Linear search works well for arrays that are fairly small, such as a few hundred elements.
- As the array in question gets very large, say, millions of elements, then the efficiency of the linear search degrades.
- If the array of elements is in sorted order, then there is a much better algorithm: **binary search**.

The Binary Search Algorithm

A description of binary search

- Before a binary search can be performed, the data must be in sorted order, either ascending or descending.
- The basic idea is to examine the element at the array's midpoint on each pass through the search loop.
- If the current element matches the target, then we return its position.

The Binary Search Algorithm

A description of binary search, continued

- If the current element is less than the target, then we search the part of the array to the right of the midpoint(containing the positions of the greater items).
- Otherwise, we search the part of the array to the left of the midpoint(containing the positions of the lesser items).
- On each pass through the loop, the current leftmost position or the current rightmost position is adjusted to track the position of the array being searched.

The Binary Search Algorithm Analysis

Determining the time efficiency

- Assume that `n` represents the size of the array to be searched.

Best case

- The key is found on the first try.

Average case

- You would need about half the comparisons of the worst case.

The Binary Search Algorithm Analysis

Worst case

- The key is not in the array, or it is at either end of a subarray.
- In such a case, the `n` elements must be divided by 2 until there is just one element remaining, and then that last element must be tested.
- This equals $\log_2 n$ comparisons.
- Therefore, in the worst case, the algorithm takes roughly $\log_2 n$ units of time.
- For example, if the size of the array is 8, then the number of comparisons needed is 3, since $\log_2 8 = 3$.

The BinarySearch Class

```
public class BinarySearch
{
    public static int binarySearch(int[] arr, int key)
    {
        int left = 0;
        int right = arr.length - 1;
        while (left <= right)
        {
            int midpoint = (left + right)/2;
            if (arr[midpoint] == key)
                return midpoint;
            else if (arr[midpoint] < key)
                left = midpoint + 1;
            else
                right = midpoint - 1;
        }
        return -1;
    }
}
```

Searching Algorithms: End of Notes