Lecture 10: Arrays and In-Place Sorting
10:00 AM, Feb 14, 2018

Contents

1 Interface Types Allow Changing Data Structures .......................................................... 1

2 Arrays ................................................................................................................................... 1
   2.1 An Example of Arrays ................................................................................................. 2
   2.2 Traversing Arrays ........................................................................................................ 2
   2.3 Summary: Arrays vs Lists ......................................................................................... 3

3 Summary: Different Kinds of Loops ................................................................................... 3

4 In-place Quicksort ............................................................................................................... 3
   4.1 Quicksort, functional implementation ................................................................. 4
   4.2 Quicksort, in-place version ....................................................................................... 4
   4.3 Partitioning ............................................................................................................... 5

Objectives

By the end of this lecture, you will know:

- Using interfaces to enable changing data structures
- What arrays are and how they differ from lists
- The quicksort algorithm
- How to sort a list in constant space

1 Interface Types Allow Changing Data Structures

The voting application code from the last lecture needed a list of votes as one of its fields. Which type makes more sense for this field? LinkList or IList? The advantage of using the interface here is that we could use one of several different list implementations depending on the performance needs of our code. Some list implementations need certain operations to run faster, or in less space, or with or without sharing modifications. The IList type lets us pick which implementation to use for a given use of the application (if we took the initial list as a constructor input), relying only on the common interface methods.
2 Arrays

Today, we want to think about a different goal for list implementations – a version that uses less space than the lists we have seen so far.

Think about how a LinkedList is laid out in memory: we have a bunch of Node objects linked together through their next fields. The nodes can be scattered in memory, and the next fields need space in addition to the data. What if instead we put the items in consecutive memory locations, as in the following picture:

The consecutive-location data structure is called an array. An array has a set number of slots for elements (which can be changed if needed later), in consecutive memory locations. This saves space, while also making it much faster to get to items based on their positions. For example, if you ask for the fifth element in the array, the underlying language implementation can compute the exact memory location rather than traverse next references through a linked list. Roughly, the location is

\[
\text{arrayLocation} + (\text{posWanted} \times \text{perElementSpace})
\]

where perElementSpace is the amount of space Java needs to store the reference to each item (the item objects can be anywhere in memory).

Arrays exist in nearly all programming languages. They are provided by default in Java, so you don’t have to import anything to use them.

2.1 An Example of Arrays

The following code creates an array of 5 strings, representing names:

```java
String[] names = new String[5]
```

Square brackets are used for arrays. The notation type[] means “an array whose items are of the given type”. You can put any type that you want into an array, but an individual array can only hold one type of value (the type can be an interface, which would allow some variation in the concrete classes supported).
We also use square brackets to store items or to access items by position. Positions are numbered starting from zero. An array of five items thus has positions 0 through 4.

names[0] = "Kathi"; \ stores a name in the first position of the array
names[3] \ accesses the fourth name in the array

2.2 Traversing Arrays

Iterators are not defined on arrays. Thus, you can’t use for loops of the style for (Integer n : anArray). You can use the style of for loop shown for the seats problem on the homework, where you manually maintain the index (for (int i = 0; i < 5; i++)).

2.3 Summary: Arrays vs Lists

Here’s a table summarizing the differences between arrays and (linked) lists.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Array</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory layout</td>
<td>consecutive elements</td>
<td>elements scattered in memory</td>
</tr>
<tr>
<td>time to access an element</td>
<td>constant</td>
<td>linear</td>
</tr>
<tr>
<td>time to add an element</td>
<td>constant if position exists (else must extend array)</td>
<td>constant if inserting at an end</td>
</tr>
<tr>
<td>space used</td>
<td>proportional to number of elements declared up front</td>
<td>proportional to number of elements in list; also needs space for next references</td>
</tr>
<tr>
<td>time to remove element</td>
<td>depends on position, since must shift remaining elements over to keep items consecutive</td>
<td>linear</td>
</tr>
<tr>
<td>Iteration</td>
<td>position-based for loop</td>
<td>iterator-based for or while</td>
</tr>
</tbody>
</table>

Sometimes, you want both the convenience of iterators and constant-time access to elements. In that case, you can use a Java class called ArrayList. They take a bit more space than arrays, and they can only be used with objects (as opposed to plain int or bools). They are often a reasonable choice if you use each of insert, delete, and access with high frequency.

3 Summary: Different Kinds of Loops

We have seen three different kinds of loops across the last two lectures:

- **while** loops have you continue a computation until some condition has occurred. Here, we saw **while** loops for traversing a list, but we could have also been repeating other tasks (such as asking a user to type in numbers, stopping when they enter the word “done”).

- Iterator-style **for** loops visit each element in a list (or other collection of data) exactly once, letting you perform some aggregating computation over each element.

- Position-based **for** loops let you repeat a computation on each item in some formula-defined sequence, such as all the numbers from 0 to the size of an array, or all even numbers.
Usually, which type of loop you use is determined by a combination of the data structure you are using (which style of for-loop) and the computation to perform (for-vs-while).

4 In-place Quicksort

We brought up arrays in the hopes of having a list implementation that uses less space. How would we do operations like sorting on array-based lists? The sorting algorithms you have seen to date involve building up new or intermediate lists as part of the algorithm. What if we had to sort in constant space (so no creating additional lists as you did in insertion or merge sort)? Instead, you would have to sort within the array you already have, somehow moving elements around within the array. Such algorithms are called in-place algorithms.

For today’s demonstration we will use one of the most widely-used algorithms: quicksort. We’ll get the intuition of quicksort by first looking at a functional implementation, then we’ll do an in-place version using arrays.

4.1 Quicksort, functional implementation

The idea of quicksort is straightforward: given a list to sort, we choose any one element, then partition the remaining elements into two pieces: those smaller than the pivot and those larger than the pivot. We sort the pieces recursively, then append the sorted-smaller items, then the pivot, then the sorted larger items. Here’s the code in Racket:

```
(define (quicksort L)
  (cond
   [(empty? L) empty]
   [else (append (quicksort (smaller-than (first L) (rest L)))
                (list (first L))
                (quicksort (larger-than (first L) (rest L))))]))

(define (smaller-than x L)
  (filter (lambda (y) (< y x)) L))

(define (larger-than x L)
  (filter (lambda (y) (>= y x)) L))
```

4.2 Quicksort, in-place version

Today we will discuss an implementation of this very same algorithm, but using mutable arrays rather than immutable lists. That is, we won’t create a new array with each recursive call; instead we will continually modify the same array, until it is sorted.

Here is an example of quicksorting an array, in-place:

- Initial Array:
  
  | 33 | 157 | 18 | 155 | 32 | 17 | 51 |
• Step 1: Choose 51 as the pivot element.

• Step 2: Move the elements less than 51 to the left part of the array, and move the elements greater than or equal to 51 to the right part of the array:

<table>
<thead>
<tr>
<th>33</th>
<th>18</th>
<th>32</th>
<th>17</th>
<th>155</th>
<th>157</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>R</td>
<td>R</td>
<td>pivot</td>
</tr>
</tbody>
</table>

• Step 3: Put the pivot element in between:

<table>
<thead>
<tr>
<th>33</th>
<th>18</th>
<th>32</th>
<th>17</th>
<th>51</th>
<th>157</th>
<th>155</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>pivot</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

and then recursively sort the left and right parts of the array:

<table>
<thead>
<tr>
<th>17</th>
<th>18</th>
<th>32</th>
<th>33</th>
<th>51</th>
<th>155</th>
<th>157</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>pivot</td>
<td>R</td>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>

• Final (Sorted) Array:

<table>
<thead>
<tr>
<th>17</th>
<th>18</th>
<th>32</th>
<th>33</th>
<th>51</th>
<th>155</th>
<th>157</th>
</tr>
</thead>
</table>

Interestingly, when we sort an array in place (and, more generally, when you do any sort of in-place operation), we need not return anything. The effect of quicksorting an array in place is not to produce a new sorted array; rather, it is to modify the given unsorted array.

The most interesting part of in-place quicksort implementation is the partitioning scheme. How would we go about moving all the elements less than the pivot to the left part of the array, and all the elements greater than or equal to the pivot to the right part of the array, in place (and efficiently)? There are a couple of approaches; we will show one of them.

4.3 Partitioning

Use two indices: left, which is initialized to index the first element of the array, and right, which is initialized to index the last element of the array. The array is then traversed (sans the pivot element) by moving the left index to the right and the right index to the left, while maintaining the following properties:

• All data stored at indices less than left have values less than the pivot

• All data stored at indices greater than or equal to right have values greater than or equal to the pivot.

Here is a simple iterative algorithm that does this:

1. Increment left if it indexes a cell whose value is less than the pivot value, otherwise leave left in place.

2. Decrement right if it indexes a cell whose value is greater than or equal to the pivot, otherwise leave right in place.

3. If a[left] ≥ pivot > a[right], then swap a[left] and a[right], and then increment left and decrement right.
4. Repeat until \( \text{left} > \text{right} \), at which point the entire array has been processed.

For example, consider the following array in which the pivot is 33. Initially, \( \text{left} \) indexes 51 and \( \text{right} \) indexes 32.

<table>
<thead>
<tr>
<th>51</th>
<th>31</th>
<th>155</th>
<th>18</th>
<th>181</th>
<th>157</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>right</td>
</tr>
</tbody>
</table>

Because \( 51 \geq 33 > 32 \), swap 51 and 32, then increment \( \text{left} \) and decrement \( \text{right} \).

<table>
<thead>
<tr>
<th>32</th>
<th>31</th>
<th>155</th>
<th>18</th>
<th>181</th>
<th>157</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>right</td>
</tr>
</tbody>
</table>

Now, since \( 31 < 33 \), increment \( \text{left} \), and since \( 157 \geq 33 \), decrement \( \text{right} \).

<table>
<thead>
<tr>
<th>32</th>
<th>31</th>
<th>155</th>
<th>18</th>
<th>181</th>
<th>157</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>left</td>
</tr>
</tbody>
</table>

Since \( 155 > 33 \), \( \text{left} \) cannot be further incremented, unless there is first a swap. But \( \text{right} \) can be decremented, since \( 181 \geq 33 \).

<table>
<thead>
<tr>
<th>32</th>
<th>31</th>
<th>155</th>
<th>18</th>
<th>181</th>
<th>157</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>right</td>
</tr>
</tbody>
</table>

And now there should be a swap, since \( 155 \geq 33 > 18 \). After swapping, increment \( \text{left} \) (\( 18 < 33 \)) and decrement \( \text{right} \) (\( 155 > 33 \)).

<table>
<thead>
<tr>
<th>32</th>
<th>31</th>
<th>155</th>
<th>18</th>
<th>181</th>
<th>157</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>right</td>
<td></td>
<td></td>
<td></td>
<td>left</td>
</tr>
</tbody>
</table>

At this point, \( \text{left} \) exceeds \( \text{right} \), meaning the entire array has been processed. Observe: the values to the left of \( \text{left} \) are less than the pivot, while values to the right of \( \text{right} \) are greater than or equal to the pivot.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)