


branch: master ▾

 labs / lab26 / lab26.md

 sarahjkim a day ago Changed lab numbers

3 contributors

193 lines (121 sloc) | 10.619 kb

Raw Blame History

Lab 26: Sorting Part 2

A. Project 3 progress check

There will be a project progress check in lab today. All your teammates must attend. To receive full credit, you should have working code and corresponding JUnit tests for at least one of the following:

- input of a tray's contents from a file
- checking if the goal configuration matches the current tray
- `isOk` for a tray
- checking if a given block can move in a given direction

T.a.s will also ask you about your plans for the next few days. The next project 3 checkoff is on August 12; you'll be expected to demo a program that can solve at least one puzzle.

B. Do a few more sorting-related activities

Finding the k th largest element

Suppose we want to find the k th largest element in an array. We could just sort the array to do this. Finding the k th item ought to be easier, however, since it asks for less information than sorting, and indeed finding the smallest or largest requires just one pass through the collection.

You may recall the activity of finding the k th largest item (1-based) in a binary search tree augmented by size information in each node. The desired item was found as follows:

1. If the right subtree has $k-1$ items, then the root is the k item, so return it.
2. If the right subtree has k or more items, find the k th largest item in the right subtree.
3. Otherwise, let j be the number of items in the right subtree, and find the $k-j-1$ st item in the left subtree.

A binary search tree is similar to the recursion tree produced by the Quicksort algorithm. The root of the BST corresponds to the divider element in Quicksort; a lopsided BST, which causes poor search performance, corresponds exactly to a Quicksort in which the divider splits the elements unevenly. We use this similarity to adapt the Quicksort algorithm to the problem of finding the k th largest element.

Here was the Quicksort algorithm.

1. If the collection to be sorted is empty or contains only one element, it's already sorted, so return.
2. Split the collection in two by *partitioning* around a "divider" element. One collection consists of elements greater than the divider, the other of elements less or equal to the divider.
3. Quicksort the two subcollections.

4. Concatenate the sorted large values with the divider, and then with the sorted small values.

The adaptation is as follows.

1. If the collection contains only one element, that's the one we're looking for, so return it.
2. Otherwise, partition the collection as in Quicksort, and let j be the number of values greater than the divider.
3. If k (1-based) is less than or equal to j , the item we're looking for must be among the values greater than the divider, so return the result of a recursive call.
4. If $k = j$, the k th largest item is the divider, so return it.
5. Otherwise, the item we're looking for is the $k - j - 1$ largest item among the values less than or equal to the divider, so return the result of a recursive call. The worst-case running time for this algorithm, like that for Quicksort, occurs when all the dividers produce splits as uneven as possible. On the average, however, it runs in time proportional to N , where N is the number of items in the collection being searched.

Here's a brief explanation of why the algorithm runs in linear time in a good case. Partitioning is where all the work comes in. With optimal dividing, we partition the array of N elements, then a subarray of approximately $N/2$ elements, then a subarray of $N/4$ elements, and so on down to 1. Suppose that partitioning of j elements require $P*j$ operations, where P is a constant. Then the total amount of partitioning time is approximately

$$\begin{aligned} &P*N + P*N/2 + P*N/4 + \dots + P*1 \\ &= P * (N + N/2 + N/4 + \dots + 1) \\ &= P * (N + N-1) \\ &= 2*P*N - P \end{aligned}$$

which is proportional to N .

Sorting with multiple keys

So far, we've been concerned with structuring or sorting using only a single set of keys. Much more common is the situation where a key has multiple components, and we'd like the keys to be sorted using *any* of those components. We'll look at an example in the next step.

An example is the collection of files in a UNIX directory. The command

```
ls
```

lists the files sorted by name. The command

```
ls -s
```

lists the files sorted by size. The command

```
ls -t
```

lists the files sorted by the time of the last modification.

Sorting with multiple keys - An example

Consider the following example, that represents a flat (nonhierarchical) directory of file entries.

```
public class Directory {  
  
    private FileEntry [ ] myFiles;  
  
    private class FileEntry {  
        public String myName;  
        public int mySize;  
    }  
}
```

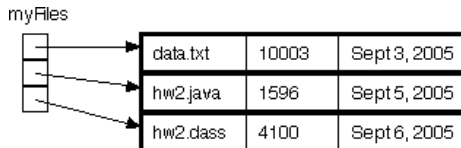
```

public GregorianCalendar myLastModifDate;
...
}

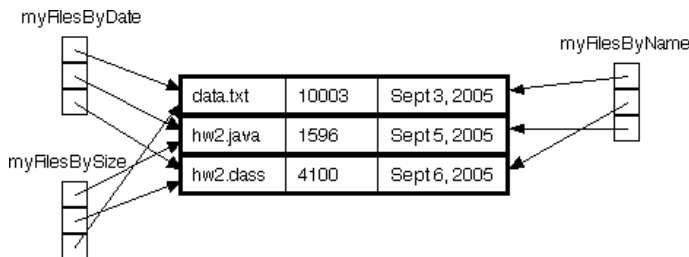
public void listByName ( ) ...
public void listBySize ( ) ...
public void listByLastModifDate ( ) ...
public void add (FileEntry f) ...
public void remove (FileEntry f) ...
}

```

One approach to supporting all three "listBy..." methods would be to have one copy of the directory's file entries—the `myFiles` array—and to sort the file entries in the array into an appropriate sequence at each call. The figure below shows an example of this approach.



Another way, which trades memory efficiency for time efficiency in the case where the directory doesn't change very often, is to have a separate array for each list order, as shown below.



Those of you with experience using data base programs may recognize this technique. Each entry in the data base typically contains a bunch of fields, and the data base program maintains index arrays that allow the entries to be listed by one field or another.

Discussion: Implementing the "add" method

Implement the "add" method.

Briefly explain how to implement the add method that uses the `myFilesByName`, `myFilesBySize`, and `myFilesByDate` as just described.

Stable sorting

A sorting method is referred to as stable if it maintains the relative order of entries with equal keys. Consider, for example, the following set of file entries:

data.txt	10003	Sept 3, 2005
hw2.java	1596	Sept 5, 2005
hw2.class	4100	Sept 5, 2005

Suppose these entries were to be sorted by date. The entries for `hw2.java` and `hw2.class` have the same date; a stable sorting method would keep `hw2.java` before `hw2.class` in the resulting ordering, while an unstable sorting method might switch them.

A stable sort will always keep the equivalent elements in the original order, even if you sort it multiple times.

Discussion: Importance of stability

Why would it matter whether or not a sorting algorithm is stable?

Describe an application where it matters if the sorting method used is stable.

Discussion: Instability of selection sort

Make the given version of selection sort stable

Here is the code for a version of selection sort that sorts an array of file entries into chronological order using the date of a file entry as the key for sorting. The `earlier` method returns true exactly when its first argument date is earlier than its second.

```
public static void selectionSort (FileEntry[] list) {
    for (int j = list.length - 1; j > 0; j--) {
        int latestPos = 0;
        for (int k = 1; k <= j; k++) {
            if (earlier (list[latestPos], list[k])) {
                latestPos = k;
            }
        }
        if (j != latestPos) {
            Date temp = list[j];
            list[j] = list[latestPos];
            list[latestPos] = temp;
        }
    }
}
```

The method is not stable. Verify this, and describe a small change in the code that makes the method stable.

Linear-time sorting with radix sort

All the sorting methods we've seen so far are comparison-based, that is, they use comparisons to determine whether two elements are out of order. One can prove that any comparison-based sort needs *at least* $O(N \log N)$ comparisons to sort N elements. However, there are sorting methods that don't depend on comparisons that allow sorting of N elements in time proportional to N . A description of one such algorithm, radix exchange sort, follows.

The radix of a number system is the number of values a single digit can take on. Binary numbers form a radix-2 system; decimal notation is radix 10. Any radix sort examines elements in passes, one pass for (say) the rightmost digit, one for the next-to-rightmost digit, and so on.

Radix exchange sort is used to sort elements of an array. Here's how it works.

1. You look at the array elements one digit at a time, starting with the high-order (most significant, leftmost) digit. You go through the array adding elements to "buckets", one bucket for each possible digit value (i.e. `bucket[0]`, `bucket[1]`, ..., `bucket[9]`). Now you've divided the array into ten pieces, `bucket[0]` to `bucket[9]`, although the order of numbers within each piece is unknown.
2. Copy the values back to the array, keeping track of the boundaries between the values in the various buckets. Now you recursively sort each of the ten buckets, only you do the division based on the next highest digit.
3. When you're down to the low-order digit, you're finished.

The number of passes through the array is the number of digits in the largest value. If this is known already, it becomes a constant, and the time for the whole sorting process is proportional to that constant times the number of array elements.

This algorithm, along with several others, is described in the [Wikipedia entry for radix sort](#).

Another radix sort (Shown in lecture) processes digits in the opposite order. A nice [demo](#) of this algorithm is available online.

If you are having issues getting the applet to run, follow the steps [here](#)

Alternatively, simply search on Youtube for a tutorial.

Why not use radix sort all the time? Fast performance requires a large number of keys that aren't too long, for which it is relatively straightforward to identify "digits".

Sorting Algorithm comparisons

		Time Complexity			Space	Stable	Comments
		Best	Worst	Avg.			
Comparison Sort	Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	For each pair of indices, swap the elements if they are out of order
	Modified Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	At each Pass check if the Array is already sorted. Best Case-Array Already sorted
	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Swap happens only when once in a Single pass
							Very small constant factor even if the complexity is $O(n^2)$.
	Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Best Case: Array already sorted Worst Case: sorted in reverse order
	Quick Sort	$O(n \cdot \lg(n))$	$O(n^2)$	$O(n \cdot \lg(n))$	$O(1)$	Yes	Best Case: when pivot divide in 2 equal halves Worst Case: Array already sorted - 1/n-1 partition
	Randomized Quick Sort	$O(n \cdot \lg(n))$	$O(n \cdot \lg(n))$	$O(n \cdot \lg(n))$	$O(1)$	Yes	Pivot chosen randomly
	Merge Sort	$O(n \cdot \lg(n))$	$O(n \cdot \lg(n))$	$O(n \cdot \lg(n))$	$O(n)$	Yes	Best to sort linked-list (constant extra space). Best for very large number of elements which cannot fit in memory (External sorting)
Non-Comparison Sort	Heap Sort	$O(n \cdot \lg(n))$	$O(n \cdot \lg(n))$	$O(n \cdot \lg(n))$	$O(1)$	No	
	Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+2^k)$	Yes	k = Range of Numbers in the list
	Radix Sort	$O(n \cdot k/s)$	$O(2^s \cdot n \cdot k/s)$	$O(n \cdot k/s)$	$O(n)$	No	
	Bucket Sort	$O(n \cdot k)$	$O(n^2 \cdot k)$	$O(n \cdot k)$	$O(n \cdot k)$	Yes	

MCN Professionals: Training & Placement division of Mindcracker Network

You might also find the [Big-O cheat sheet](#) useful when reviewing sorting algorithms and time complexities in general.

C. Homework

Readings

Tomorrow's lab will cover balanced binary search trees. Corresponding reading is

- JSS4 sections 11.4 and 11.5
- JSS3 sections 10.5 and 10.6

