Quick sort an array and a linked list

Algorithms and data structures ID1021

Johan Montelius

Fall term 2023

Introduction

So you have realized that sorting data sets could be very important and why not then implement a sorting algorithm called *quick sort* - after all, if it's called quick sort it must be good.

Quick sort works in a similar way to merge sort in that it divides the data set into two parts and then sort the to parts separately. The difference is that when merge sort simply divides the set into two equal parts and does some merging in the end, quick sort does some more work when dividing but nothing when the two sets are joined together.

To make it a bit more interesting you should implement the quick sort algorithm both for an array and for a linked list. It turns out that the linked list version is simpler to implement but working with linked lists is always more expensive . . . or, is it?

Quick sort - the algorithm

The quick sort algorithm is very easy to describe; given a sequence of items we divide them into two sequences, one with all the *smaller* and one with all the *larger*. You of course wonder what smaller and larger exactly means but it turns out that is not that important; the only important thing is that you don't end up with all the items in the same sequence.

Now assume that you can sort the two sequences separately, then you will end up with one sorted sequence with the smaller items and a sorted sequence with the larger items. If this is the case we simply combine the sequences into one sorted sequence; this can be done blindfolded since none of the smaller items should occur among the larger items and vice versa.

The quick sorting algorithm is of course used recursively so the only thing left is to make sure that we can sort a very short sequence. A sequence of just one item is of course sorted so that means that we can sort any sequence of two items - divide it and sort the two one items sequences. If you can sort

any two item sequence you can sort any three item sequence and so forth so by induction you can sort sequences of arbitrary length.

The algorithm is easily explained but to get all the corner cases implemented is slightly tricky, especially if you're sorting an array of items.

An array of items

You first task is to sort an array of let's say `int` values. Implement a static method, `sort`, that takes an array and sorts the items between, and including, two indices. If the indices are the same then of course your fine and you don't have to do anything but if they are not you have to move items around to create the two sequences. This rearrangement of items should be done in-place i.e. you should not use a temporary array to store the smaller or larger items.

Implement a static method `partition` that takes an array and two indices, *min* and *max*, that rearrange the items so all smaller items are in the beginning and all larger items in the end. The method should return an index *mid* that holds an item that divides the two segments.

The question is of course, what do we mean by smaller and larger. One solution is to simply select the first item as a *pivot* element and treat all item smaller or equal to this as smaller and the rest as larger.

The trick to implement the partitioning is to have two indices, i, and j that start in each end of the sequence, i from *min* and j from *max*. As long as the item at j is larger to the pivot element and j is still greater than i, you decrement j. Then you do the same thing with i and increment it as long as items are smaller or equal to the pivot element.

When you have done this you could be in either of two situations; either i is smaller than j which means that they both point to element that are misplaced and should be swapped. Alternatively i and j are equal and we are almost done. The last thing we need to do is to swap the pivot element at index *min* with the element at index j (which is an element that is smaller or equal to the pivot element).

The sequence now looks like follows: from *min* to $j-1$ we have all items smaller or equal to the pivot item, the pivot item is at index j and, from $j+1$ to *max* we have all items greater than the pivot item. If you now return the index j you have everything that you need to sort the two sub-sequences.

A linked list of items

Once you have the sorting of an array done, let's see how easy it is to sort a linked list. The idea is of course the same but now we do not have to keep track of indices.

Use a simple `Node` structure that holds apart from a reference to the *next node*, a simple `int` *value*. A linked list of nodes is then sorted by first partitioning the list into the smaller and larger nodes. When you do this you should use the original node structures i.e. you should not construct two new lists with new nodes - you will construct two empty lists and add the nodes of the list being sorted to either. (read that again).

Since you are now changing the next references as you traverse the list you have to be careful not ending up in some circular structure.

Once you have the partitioning done correctly you can sort the two lists recursively. The final step is then to *append* the list with the sorted larger elements but don't forget to place the *pivot node* in between.

The simple way to implement an append operation is of course to run down the first list and update the last next pointer. Doing this is rather expensive so you might want to represent a linked list with two references: one to the first element and one the last. If you do this then the append operation becomes a constant time operation.

Benchmark

Run some benchmarks to estimate the run time complexity of your implementation. Convincingly show the correlation between the expected big-O complexity and the execution time of your implementation.

Compare the cost of sorting an array and a linked list. Since the sorting time is depending on what the sequence looks like to start with you should use the same random sequences in your comparison.