CO518 – Algorithms, Correctness and Efficiency

Assignment 2 – 50 marks

Deadline – Week 11 (Monday, 3rd December, 23h55)

The task for this assessment is to implement a variant of mergeSort for linked lists. The linked lists in question are singly-linked lists without supervisor cells, i.e. with null representing the empty list. Moreover, the variant of mergeSort should be in two ways different from a standard mergeSort: (i) it operates bottom-up rather than top-down, and (ii) it exploits when parts of the list are already in increasing order.

Explanation:

- standard mergeSort splits a list into two parts of (near) equal length, mergeSorts them recursively, and then merges the results. It does not do the splitting when a list is of length 1, but such lists are already in sorted order. This means though that we eventually split the list into lists of length 1, and then start to merge them. Thus we can avoid all the recursive calls of mergeSort. Instead we keep a queue of sorted pieces of the list, and initialise it co contain singleton lists. We make progress by dequeuing two lists from the queue, merge them, and put the result back into the queue. When only one list is left in the queue we are done – this is the sorted list.
- however, we can do slightly better (in the best case anyway): we do not need to break lists all the way down to length 1 for the first part. We merely have to make sure that lists entering the queue are sorted.

Example: consider the list [1,4,7,2,5,8,23]. An ordinary bottom-up mergeSort based on singleton lists would give us the following sequence of queues:

[[1],[4],[7],[2],[5],[8],[23]]

[[7],[2],[5],[8],[23],[1,4]]

[[5],[8],[23],[1,4],[2,7]]

[[23],[1,4],[2,7],[5,8]]

[[2,7],[5,8],[1,4,23]]

[[1,4,23],[2,5,7,8]]

[[1,2,4,5,7,8,23]]

As you can see, the step from one queue to the next is to take the first two lists of the queue, merge them, and put the merged result at the end. It would also work if the result were placed back at the beginning, but that would just be a fancy version of insertionSort.

We do not have to break the list into singletons in the first step though. If we keep the sub-lists of our list that were already in order intact then we get this instead:

[[1,4,7],[2,5,8,23]]

[[1,2,4,5,7,8,23]]

More specifically, you need to implement the following methods: (10 marks each part)

1. A method **isSorted**() which checks whether a list is already sorted. It should return true if the list is already sorted in ascending order, and false otherwise.
2. A method **queueSortedSegments**() which breaks a list into sorted segments, add those to a (fresh) queue, and returns that queue. (At most half marks if you always break the list into singleton segments.)
3. A method **merge**(Node<T> parameter) which merges the parameter list with 'this' list. This method should only be called if both these lists are already sorted. This method should merge these two lists and return the merged result.
4. The actual **mergeSort**() method which implements the algorithm outlined here, and returns the result.
5. A static **test**(int n) method, which puts all of these things together: it should generate a random list of integers (such a method is provided), sort them, show the lists before and after sorting and report what the isSorted method has to say about those lists.

Hand-in instructions:

The deadline is Monday, 3rd December 2018 at 23:55. Submission is via the CO518 Moodle page. No late submissions will be accepted. Submit a single .zip file containing all the source code.