CS2302 - Data Structures

Spring 2019

Lab No.2 Report

Introduction

For this lab we are making use of linked lists in and different sorting algorithms such as, bubble sort, merge sort and quick sort. The objective is to better understand the linked list data structure, how its pointers work and how the different sorting algorithms also work. Closely observing the different running times of them when implemented in a different data structure.

Design & Implementation

To find how to implement the different sorting algorithms in linked lists we have to first see how they work and then try to implement them in a way that works without indexes.

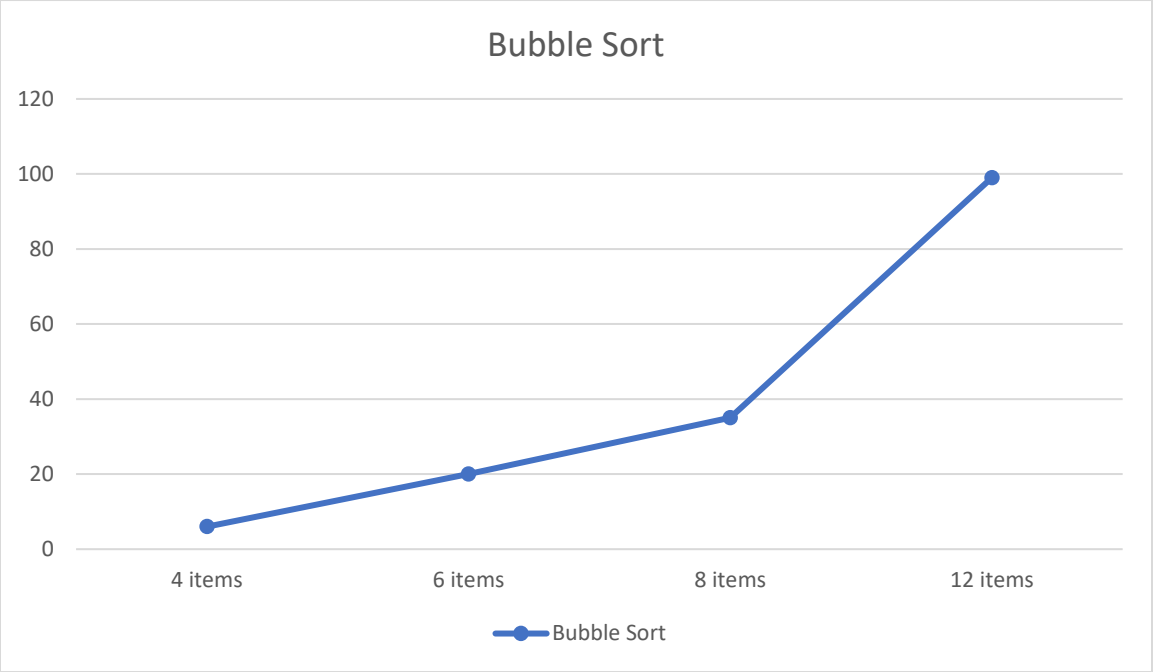
Problem #1: Bubble Sort

The simplest form of sort. The method first takes into account if the list is empty or if it only has one element in it. If any of them are true then just return the list.

The same ruleset for an array is followed in the linked list. If the function sees a number that is higher than the next one they will be swapped. This is accomplished with a variable that stores the current item, if the check is true then the current number will be the next one and the next one will be the previously stored item.

The function follows the check of bigger numbers using a while loop to make its comparisons, and it will continue until there is no longer a swap of numbers made. This is accomplished with a variable that sees if there were any swaps made during the while loop. If there were no swaps then exit the while loop and the list will be sorted.

Here are the different number of comparisons made from the Bubble Sort method.



As be seen the method looks like a parabola, denoting its $O(n^2)$ running time.

Problem #2

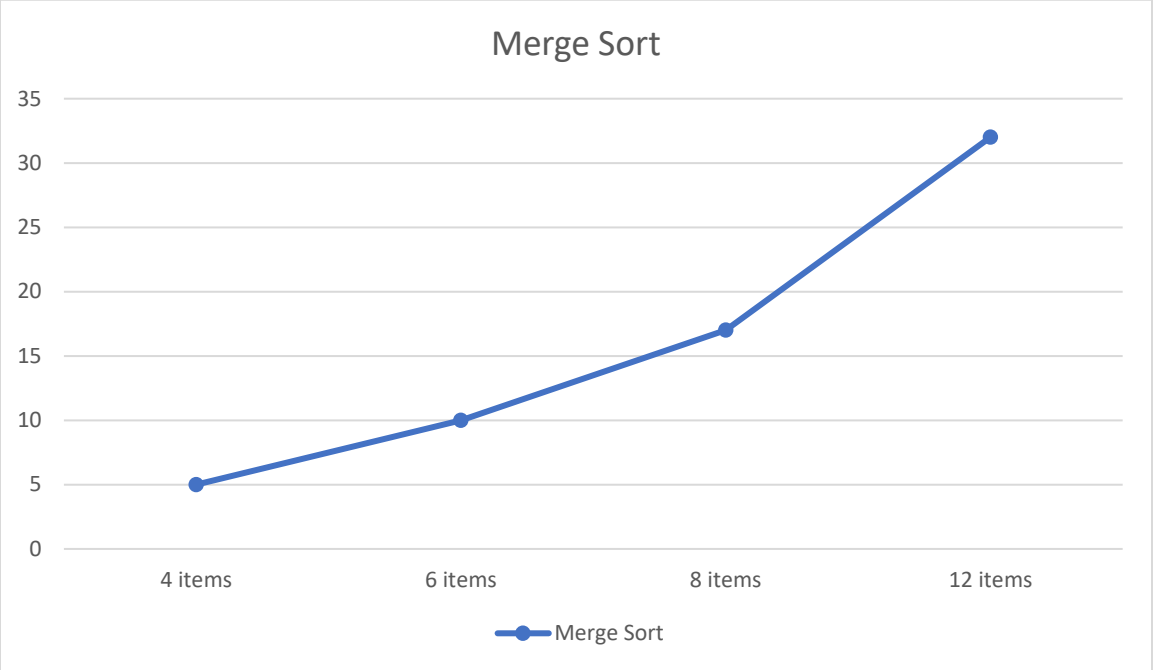
For merge sort we will use 3 different methods, one that receives the list, another one that splits it and finally another one that unites the broken apart lists that are of length 1.

First the method receives the list in $mergeSort(L)$ the function then checks if the List is empty or if it only contains 1 item. If it does then return the head of this list. If not then partition the given list by its half.

For partitioning the list we will have a fast and a slow pointer. The slow pointer will be assigned to the head of our list to be partitioned and the fast pointer will be assigned to the next node after the head. The fast pointer will always move 2 nodes while the slow pointer will only move 1. After the fast pointer becomes null then the slow pointer will be pointing at the center of the list. The left list will become the head of the original up until the slow pointer and the right list will be whatever is after the slow pointer.

After both lists are split then we will call the method that unites them but first we will continue breaking up the lists by their halves until only length 1 lists remain.

Different number of comparisons by the Merge Sort:



The method continues growing, like bubble sort the running time will continue rising but in a much slower manner. Proving its running time of $O(n * \log n)$

Problem #3

The method for Quick Sort is much harder to implement in a singly linked list since we don't have backwards traversal available. This comes as a problem as many implementations of Quick Sort use a pivot point to compare all the numbers after the pivot point and change their position in respect to it.

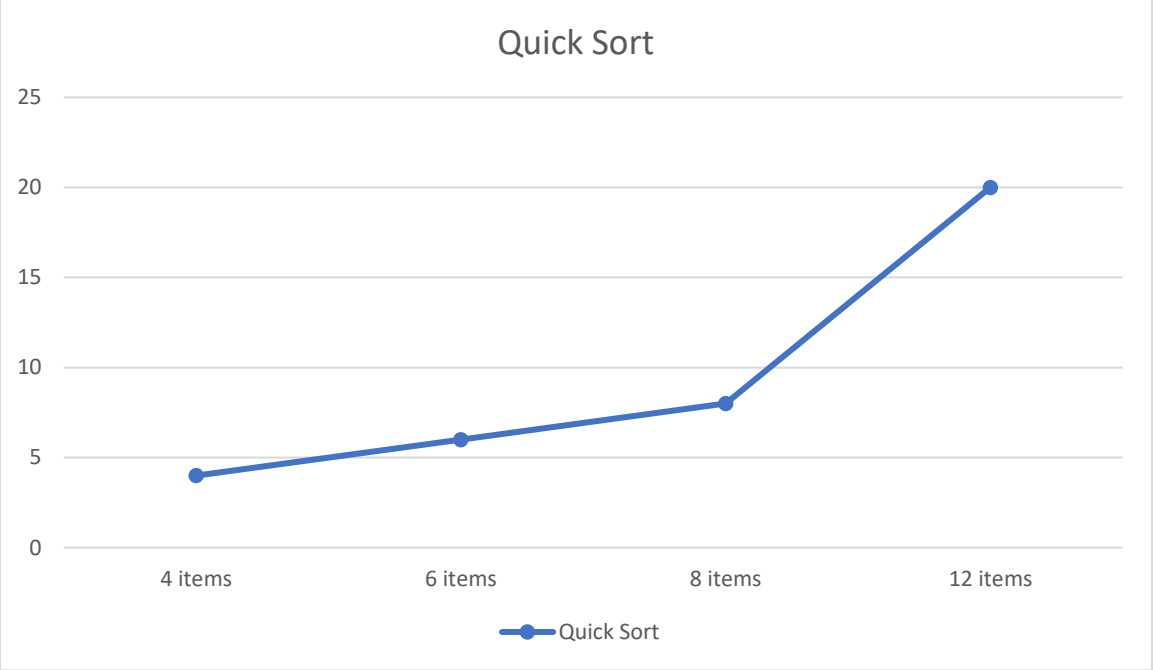
For this sort we will have 2 methods, one that receives the linked lists and unites them at a later point, after using a second method to split the linked lists following their pivot points. This is used because we know the pivot point is already in its correct position and we need to continue doing the same for the list created to the right of the pivot point and to the left of it.

The method used for the partition of the linked lists receives the head of the list and returns the head of the new partitioned list.

The method will use 3 pointers for the partitioning of the lists, one that is the pivot point that will be the head of the list, another one for traversing the list as an iterator and the last one will tell us the last swapped node. Every number smaller or equal to the pivot will be placed at the end of the list and in this way we will have all the numbers smaller than the pivot on a side of the list and those bigger on the other side.

Keep repeating until the lists are sorted and then return them to the Quick Sort method. Use an iterator to unite the partitioned lists without any comparisons made since the lists are already sorted and finally return the sorted list.

Different number of comparisons by Quick Sort:



It can be seen how the number of comparisons is much more lower than the previous methods. Still, my method is not perfect and sometimes will perform extra comparisons since the lists are not sorted correctly. Proving its best case running time of $O(n * logn)$ and its worst running time of $O(n\text{^}2)$ still the method will perform even worse at some points.

Design

The code provides a user interface that asks for the length of the desired linked list and populating it with random numbers from 1 to 101. The random ordered list is provided and then the sorted list, with its median and the number of operations performed also being displayed.

Conclusion

It is hard to get a grasp in how linked lists work, since they are commonly influenced by pass by references and the pointers system is somewhat complicated to understand at first glance. The sorting algorithms are simple enough when arrays are involved but become increasingly difficult when trying to implement them in linked lists. They also prove to perform poor when running time are concerned.