Zura Mestiashvili
Professor Chun Wai Liew
CS 150 : Data Structures & Algorithms
10/02/2016

Lab_05 Report

Introduction

With this experiment we are trying to evaluate performance of two sorting algorithms - *QuickSort* and *MergeSort*. We apply both of them on different data containers - *ArrayList*("Java Platform SE 8.") and *LinkedList*("Java Platform SE 8."). Both containers contain only String values, and our aim is to sort this data. Program tries to perform this experiment by adding different numbers of String values to each container, and then applying sorting algorithms on both containers. We are trying to find out which sorting algorithm is better in specific case. For instance, *QuickSort* can be take less time for sorting *LinkedList* than for sorting the *ArrayList,* while *MergeSort* may work absolutely differently - taking more time while sorting *ArrayList.* We are also trying to find out in which cases are sorting algorithms profitable and when to use them. My hypothesis is that *QuickSort* will take less time for sorting *ArrayList* than *LinkedList*, as its algorithm is based on swapping values, while working on *LinkedList* we will need more time to get specific element and then replace it with another. Although, QuickSort is slow for *LinkedLists*, *MergeSort* will have almost the same performance for both containers, as its algorithms is not based on swapping elements. Average complexity for both sorting algorithms is O(nlogn), while worst case for *QuickSort* is *O(n^2)* and for *MergeSort O(nlogn)*, though in some cases *QuickSort* can be more efficient than *MergeSort* as *QuickSort* has a smaller constant than the other and *MergeSort* requires extra memory(Stackoverflow. *Why is quicksort better than mergesort.*).

Approach

As we are trying to compare two sorting algorithms, which have to sort data, we can use interface, defining the methods each sorting algorithm should implement. For this we use *SortedList Interface* that declares six functions: *addElement, clearAL, getSize, getElement, setElement* - which are mainly setters and getters, as our internally stored container should be private we need to get and set data somehow, those methods serve this purpose, and another method is *printData*.

In addition, as we are testing our experiment for different kinds of data containers we can use abstract classes, stating which methods have to be implemented in children classes that use different data containers(ArrayLists or LinkedLists)*. *QuickSortData* and *MergeSortData* are these abstract classes. Finally, we will have four children classes: *ArrayListQuickSortData, LinkedListQuickSortData, ArrayListMergeSortData and LinkedListMergeSortData*. Therefore, the whole structure of the program will be the following:

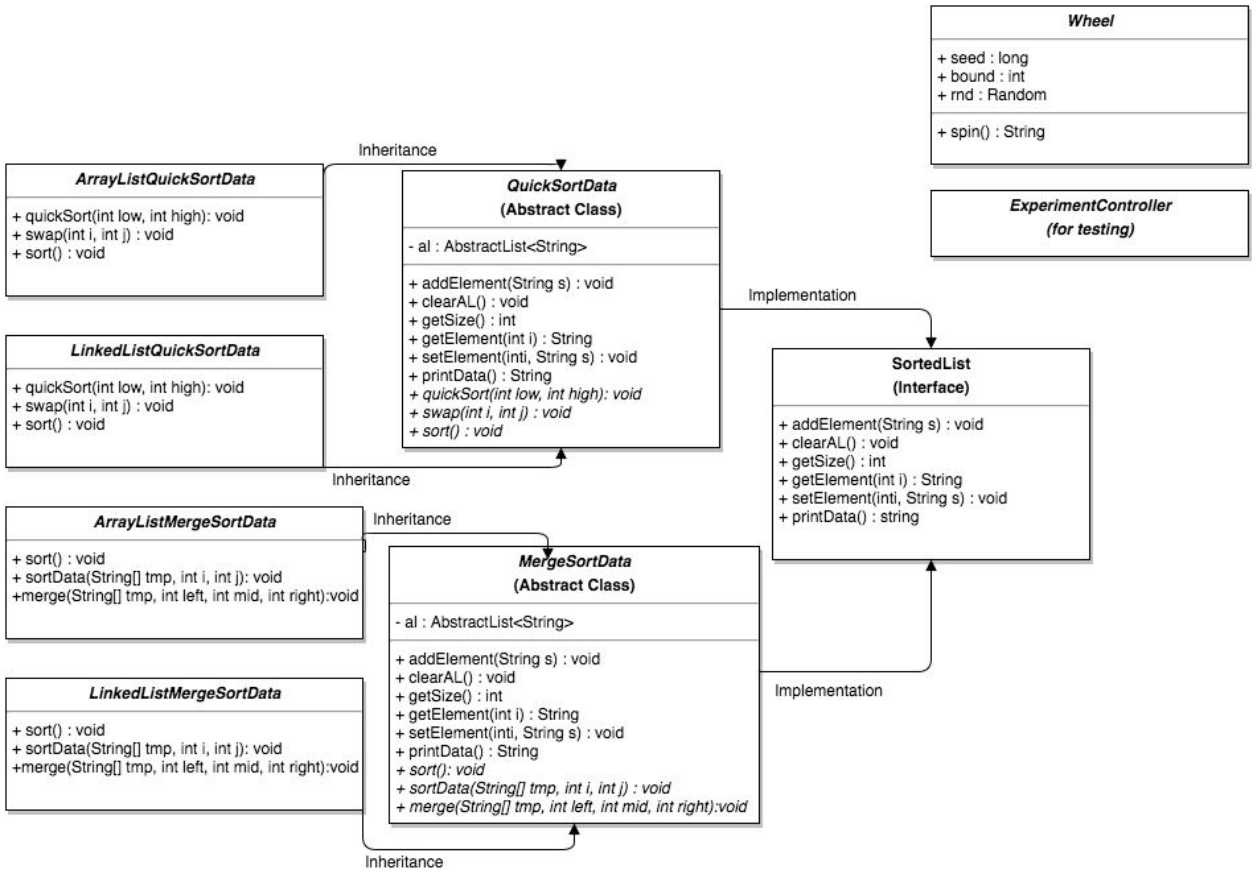

Figure 1. Class Diagram. Italic letters indicate abstract classes. '+' sign indicates variables or methods which are public, while '-' sign indicates private ones

We use *Wheel* class to generate new *String* values and fill our containers. *ExperimentController* is used for testing purposes. For quickSort we use partitioning algorithm(Weiss, Mark Allen. *Data Structures & Problem Solving Using Java, page 413*).

*The only reason I define methods in *ArrayListQuickSortData, LinkedListQuickSortData, ArrayListMergeSortData and LinkedListMergeSortData* is because the lab assignment description states that *QuickSortData* and *MergeSortData* should be abstract classes. If I define methods: *quickSort, swap* and *sort* in *QuickSortData* and *sort, sortData* and *merge* in *MergeSortData* it will be totally fine, the program will still work by the virtue of inheritance but there will be no point in declaring *QuickSortData* and *MergeSortData* as abstract classes. That is why I define those methods in those four children classes, otherwise I would prefer using inheritance and putting common functionality in *QuickSortData* and *MergeSortData*.

Methods

I ran this experiment with two different ways. First of all, I created a constant which would be the limit of number of items container could have. I created loop, and for each iteration of the loop I would increase number of items of container by 500. My initial value was 1000. For instance, if constant were 5, then I would apply sorting algorithms to containers with 1000 elements at first, 1500 -at second, 2000 - at third, 2500 - at fourth and 3000 - at fifth iteration. I used this variant of testing twice for constant 100. Following graphs represent the average values of the results of these test cases:

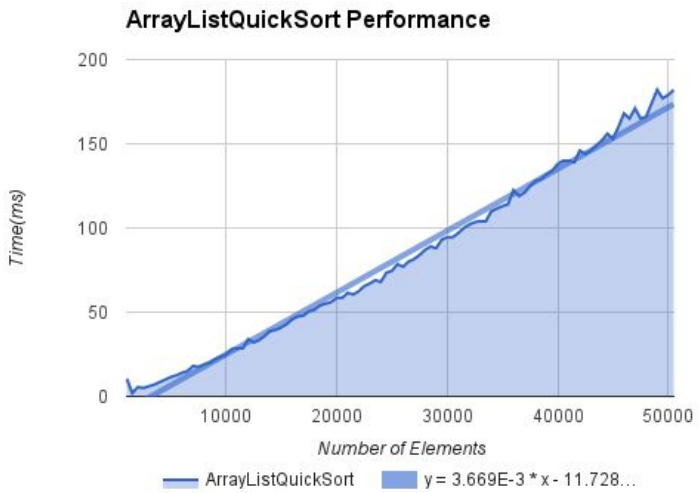

Figure 2. Results of applying QuickSort to ArrayList

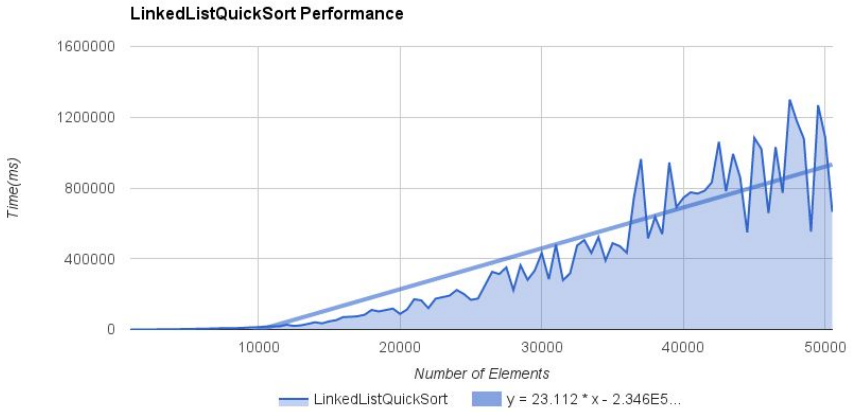

Figure 3. Results of applying QuickSort to LinkedList

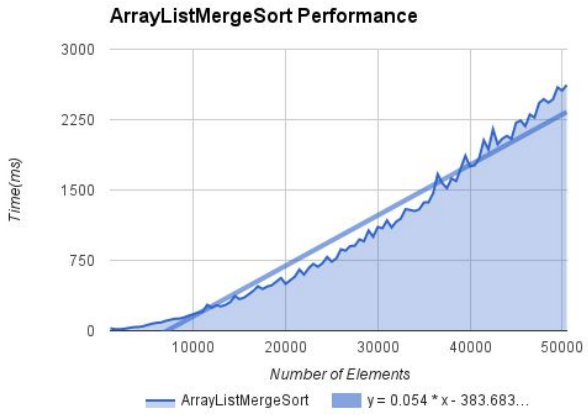

Figure 4. Results of applying MergeSort on ArrayList

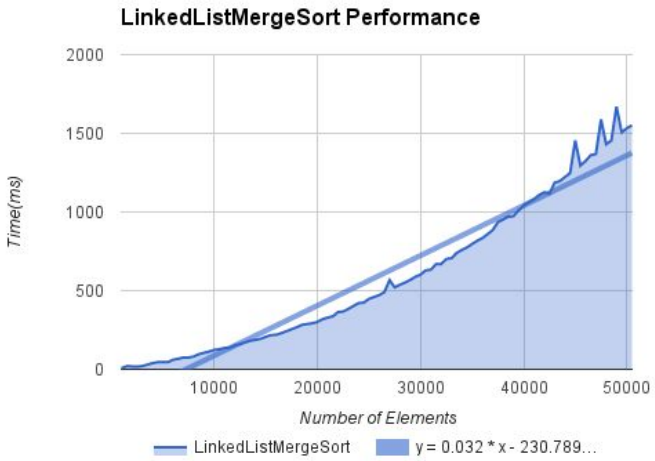

Figure 5. Results of applying MergeSort on LinkedList

Another way that I used is creating constant again and for each iteration multiplying initial value of number of elements over 2. For instance, if initial value is 100, and constant 12, program will run sorting algorithms on containers with: 100 * 2^0, 100*2^1, 100*2^2 ….. 100*2^11 elements. I ran this test for constant 12 twice and got the following results:

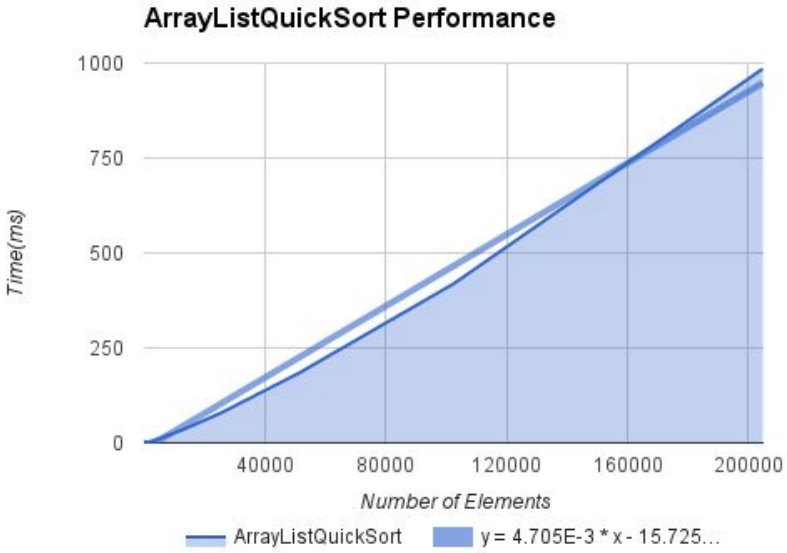

Figure 6. Performance of QuickSort on ArrayList for bigger number of items

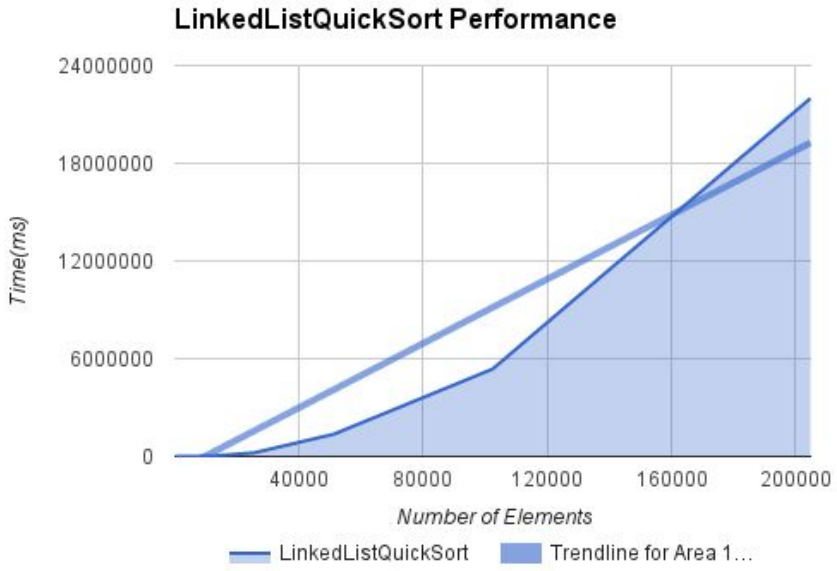

Figure 7. Performance of QuickSort on LinkedList for bigger number of items

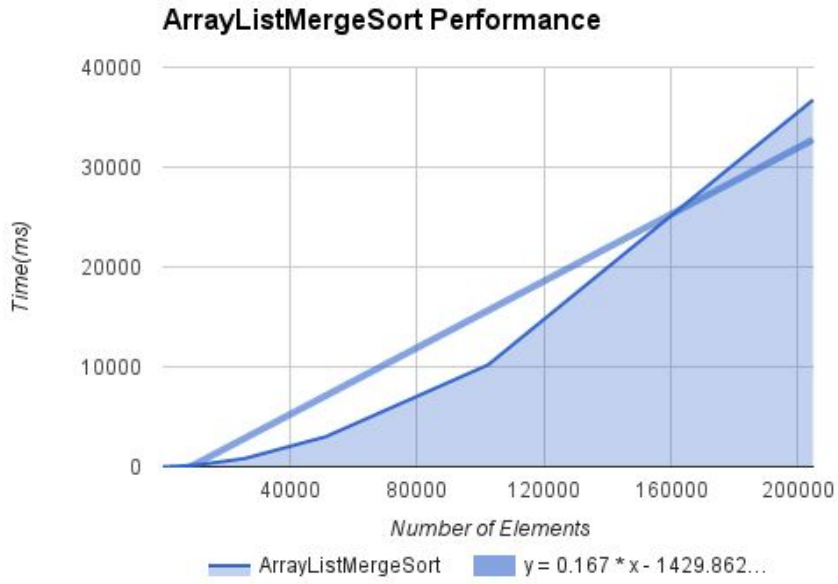

Figure 8. Performance of MergeSort on ArrayList for bigger number of items

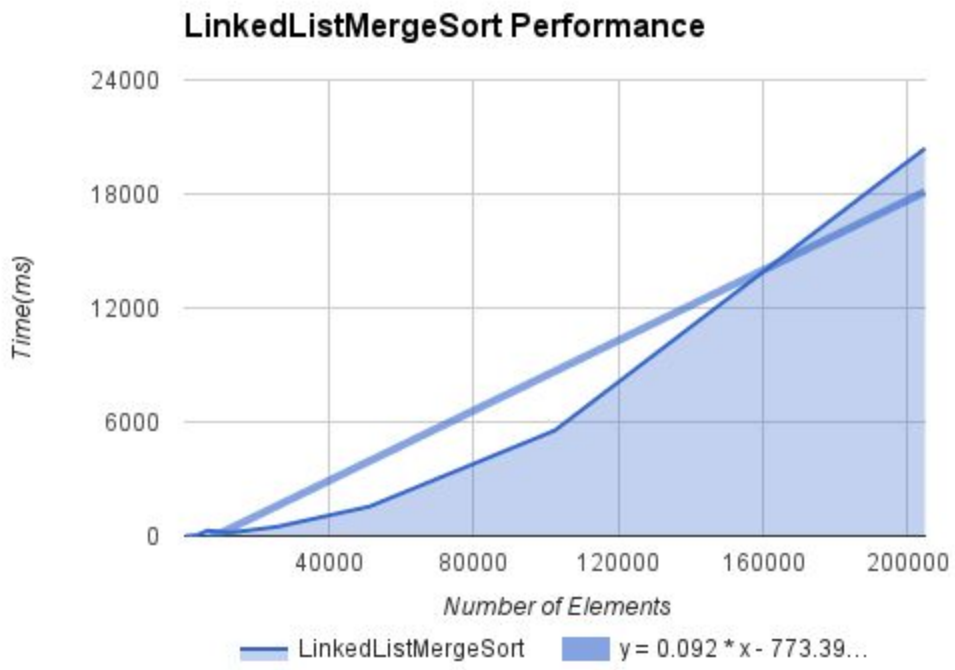

Figure 9. Performance of MergeSort on LinkedList for bigger number of items

While first way of testing sorting algorithms provides more data(approximately 10 times bigger data than the second one), second way provides results for bigger numbers(such as 200000, while first way's maximum number of elements is just 50 000).

Data and Analysis

Both methods provide similar results, though Figures 6, 7, 8, 9 seem more accurate, as Figures 2,3,4 and 5 at some points have values that are not appropriate for that specific argument(in my opinion this is mainly caused by other processes running on the computer during this experiment). Both Figure 2 and 6 show that performance of QuickSort is better on ArrayList - for 50000 element container running time is under 250ms, while for LinkedList, as Figures 3 and 7 show the same size container needs approximately 1,000,000ms. Although, QuickSort takes so much time for LinkedLists, as Figures 5 and 9 show MergeSort needs more time to sort LinkedList than to sort the same size ArrayList(Figures 4 and 8). While for 70,000 element ArrayList takes about 10,000ms(Figure 8), same size LinkedList takes just 6,000 ms to be sorted with MergeSort algorithm. While comparing QuickSort and MergeSort performances on ArrayList(Figure 2 and Figure 4) we can see that QuickSort is more efficient than MergeSort;

Sorting 50,000 element ArrayList takes approximately 10 times less for QuickSort than for MergeSort. For LinkedList it is absolutely different. As we can see from Figures 7 and 9 it takes approximately 1000 times more for QuickSort than for MergeSort to sort LinkedList.

Conclusion

In conclusion, answer to the question which sorting algorithm is better depends on the case. As we found QuickSort is more efficient for sorting ArrayLists, while MergeSort is more efficient for LinkedLists, and can sort it even faster that it can sort ArrayList(Figures 8 and 9). As I states in my hypothesis *QuickSort* takes less time for sorting *ArrayList* than sorting *LinkedList,* though my hypothesis about MergeSort is different from reality. As we figured out, MergeSort takes less time for sorting LinkedLists than for sorting ArrayLists, it does not have the same performance for both data structures. Although on Figures 2,4 and 5 graphs looks close to the linear one, as we look at Figures 6,7,8 and 9 we will see that it is different from linear, as we stated average complexity of both containers is *O(nlogn),* which is illustrated better on Figures 6,7,8 and 9 as they represent results of bigger size containers.

References:

1. Liew, Chun W. *CS150 Lab 5 Description*. Retrieved September 25, 2016 from http://www.cs.lafayette.edu/~liew/courses/cs150/lab/labs/lab05g/
2. Stackoverflow. *Why is quicksort better than mergesort.* (n.d.). Retrieved October 2, 2016, from http://stackoverflow.com/questions/70402/why-is-quicksort-better-than-mergesort
3. Weiss, Mark Allen. *Data Structures & Problem Solving Using Java*. Reading, MA: Pearson College Div, 4th edition, 2009. Print.
4. "Java Platform SE 8." *Java Platform SE 8*. Retrieved October 1, 2016 from https://docs.oracle.com/javase/8/docs/api/