Data Structures: Lab 4

Following the American Psychological Association's Guidelines

Markham M. Shofner

Johns Hopkins University

Data Structures: Lab 4

Analysis Document

The primary data structures I used in this lab were: a simple Stack class and a simple StackNode class. The rest of the logic lived in the primary SortComparison class, where the sorting methods, and their supporting subroutines lived. The Stack and StackNode were used to implement the iterative version of QuickSort, whose recursive implementation can be mimicked by using a stack that tracks start and end values for separate chunks of the data (instead of allowing recursion to naturally manage state). The functions were based on their standard patterns [e.g. – QuickSort() heavily leaning on Partition() and HeapSort() heavily leaning on MaxHeapPercolateDown()], so no major changes in the way that logic was handled were needed.

In terms of process, this lab required the greatest number of hours thus far, but I also invested the most hours ahead of time - so ended up writing a more mature and less rushed program than any of the other labs in this course. Working in chunks and allowing the problem(s) time to percolate in my mind increased the efficiency with which I was able to solve the touchier aspects of the code once I actually started typing. I also spent more time upfront rereading the book sections, rewatching the relevant lectures, and sketching/planning a course of attack, which reduced the amount of throwaway code and wheel-spinning that had cropped up in some of my earlier lab work. I kicked the work off by building out the heap, using the zyBook HeapSort pseudocode. I leaned on all the zyBook presented pseudocode to set up the bones of each sorting function. The only function that required some particularly intensive tweaking was the iterative implementation of QuickSort, as the zyBook implementation was recursive. Recursion naturally makes more sense to me, and seems more elegant, for a QuickSort – so I built out the recursive method based on the zyBook pseudocode, and then messed around for a while with a stack and pointers before landing on an option the properly and iteratively QuickSorted the data based heavily on the Stack class. In the end, iteration is a great solution because it is easier to debug, and also takes up substantially less space on the computer. If I had written the QuickSort recursively, it would have been easier initially, but may have crashed when messing around with larger datasets. Also, I didn't give writing HeapSort recursively a shot, but that may have taken up a large amount of space, both cognitively and from a computer memory/function perspective.

Here is a conditionally formatted excel table of runtimes (in nanoseconds).

	50	500	1000	2000	5000	
QuickSort1	7.434E+05	2.934E+06	5.546E+06	1.017E+07	2.036E+07	**Random**
QuickSort1	7.570E+05	1.840E+06	3.145E+06	6.575E+06	1.395E+07	**Ordered**
QuickSort1	7.900E+05	1.924E+06	3.588E+06	7.018E+06	1.924E+07	**Reversed**
QuickSort2	7.183E+05	3.066E+06	5.777E+06	1.075E+07	2.053E+07	**Random**
QuickSort2	7.601E+05	1.973E+06	3.225E+06	6.542E+06	1.499E+07	**Ordered**
QuickSort2	7.218E+05	1.970E+06	3.547E+06	6.964E+06	1.896E+07	**Reversed**
QuickSort3	6.699E+05	3.032E+06	5.686E+06	1.084E+07	2.061E+07	**Random**

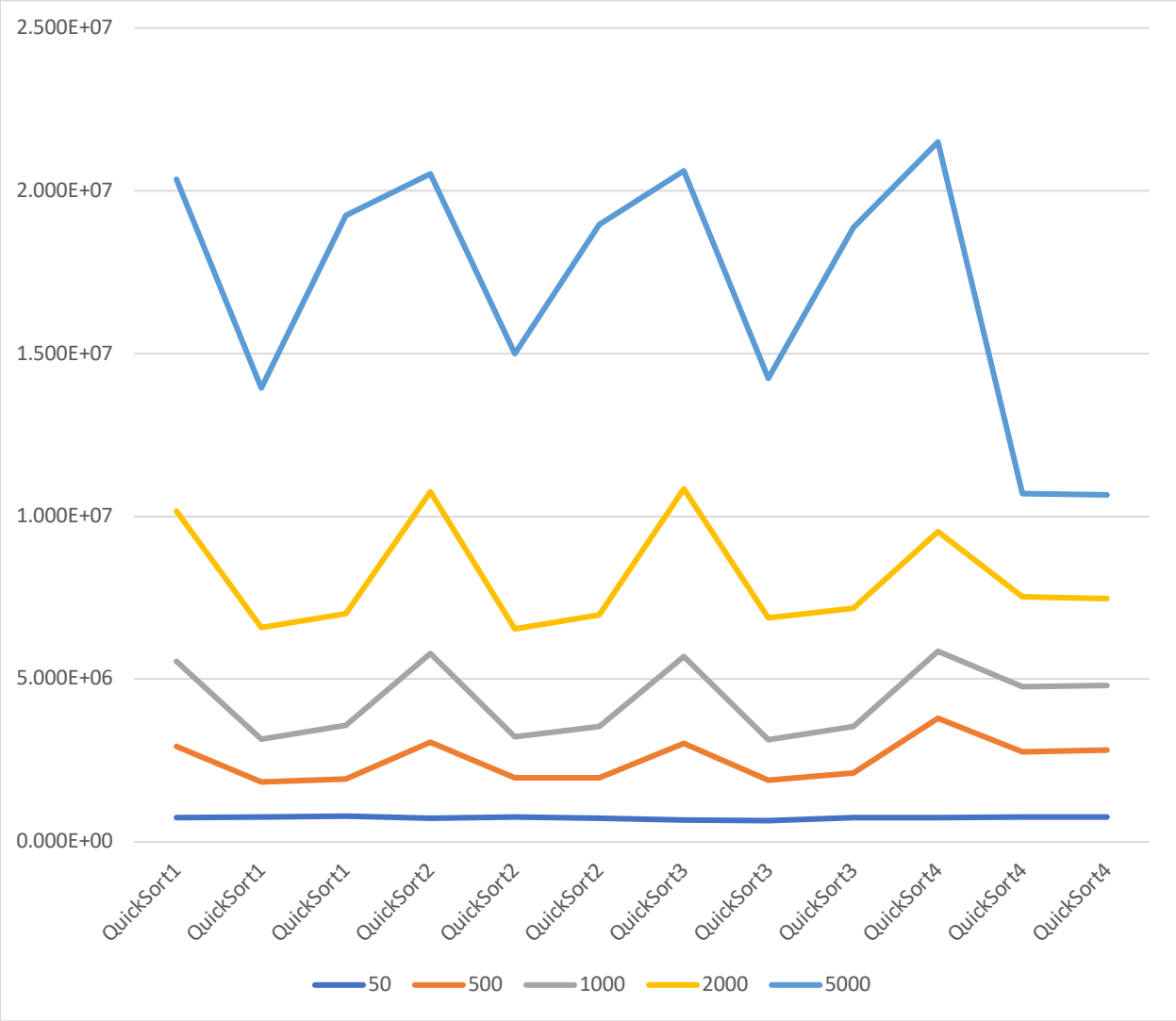
QuickSort3	6.511E+05	1.900E+06	3.138E+06	6.885E+06	1.423E+07	**Ordered**
QuickSort3	7.373E+05	2.116E+06	3.535E+06	7.182E+06	1.886E+07	**Reversed**
QuickSort4	7.451E+05	3.796E+06	5.854E+06	9.528E+06	2.150E+07	**Random**
QuickSort4	7.618E+05	2.759E+06	4.758E+06	7.536E+06	1.069E+07	**Ordered**
QuickSort4	7.609E+05	2.814E+06	4.794E+06	7.476E+06	1.066E+07	**Reversed**
HeapSort	2.613E+04	3.637E+05	5.478E+05	8.229E+05	1.445E+06	**Random**
HeapSort	2.753E+04	4.160E+05	5.410E+05	7.843E+05	1.424E+06	**Ordered**
HeapSort	2.612E+04	3.997E+05	5.449E+05	7.787E+05	1.421E+06	**Reversed**

The below table was pulled out of Excel as an image, since Word was not able to handle the conditional formatting applied. It offers an alternative way to view the same data above, but with a potentially more intuitive (depending on the viewer) way to view trends in the data. Highlights include HeapSort being O(nlogn) and faster across the board than any of the four QuickSort options, while also being resilient to the order, randomness, or reverse-order of the data, in terms of impact on speed. Also, QuickSort seems to play much better with ordered data than random or reverse-ordered.

	50	500	1000	2000	5000	
QuickSort1	7.434E+05	2.934E+06	5.546E+06	1.017E+07	2.036E+07	**Random**
QuickSort1	7.570E+05	1.840E+06	3.145E+06	6.575E+06	1.395E+07	**Ordered**
QuickSort1	7.900E+05	1.924E+06	3.588E+06	7.018E+06	1.924E+07	**Reversed**
QuickSort2	7.183E+05	3.066E+06	5.777E+06	1.075E+07	2.053E+07	**Random**
QuickSort2	7.601E+05	1.973E+06	3.225E+06	6.542E+06	1.499E+07	**Ordered**
QuickSort2	7.218E+05	1.970E+06	3.547E+06	6.964E+06	1.896E+07	**Reversed**
QuickSort3	6.699E+05	3.032E+06	5.686E+06	1.084E+07	2.061E+07	**Random**
QuickSort3	6.511E+05	1.900E+06	3.138E+06	6.885E+06	1.423E+07	**Ordered**
QuickSort3	7.373E+05	2.116E+06	3.535E+06	7.182E+06	1.886E+07	**Reversed**
QuickSort4	7.451E+05	3.796E+06	5.854E+06	9.528E+06	2.150E+07	**Random**
QuickSort4	7.618E+05	2.759E+06	4.758E+06	7.536E+06	1.069E+07	**Ordered**
QuickSort4	7.609E+05	2.814E+06	4.794E+06	7.476E+06	1.066E+07	**Reversed**
HeapSort	2.613E+04	3.637E+05	5.478E+05	8.229E+05	1.445E+06	**Random**
HeapSort	2.753E+04	4.160E+05	5.410E+05	7.843E+05	1.424E+06	**Ordered**
HeapSort	2.612E+04	3.997E+05	5.449E+05	7.787E+05	1.421E+06	**Reversed**

In the below chart, HeapSort has been taken out for clarity, so that we may more accurately analyze the QuickSort options. The chart is imperfect, in terms of formal data discussion, but it does a few things well. Firstly, these periodic peaks in the lines are related to the randomly sorted data files – so, across the board, randomly sorted data requires *more* time than either ordered or reverse-ordered data. Secondly, the major valleys are related to the ordered data files – so, across the board, ordered data requires *less* time than either random or reverse-ordered data for QuickSort. Lastly, the pivot choice in Option4 (where instead of using the first index, we calculate a midpoint pivot) apparently equalizes the time for ordered and reverse-ordered data

(which makes sense), and leaves both still substantially below the time requirements for randomly ordered data.



Enhancements

- Recursive quicksort an option
- Each data file can be consumed with delimiters being spaces, carriage returns, or new lines [32, 13, 10 in ASCII respectively].

General notes

- I started running a few of the time trials with `System.out.println()` statements still in the code, and after removing those the speed improved *drastically*, almost comically, (Roughly 10x faster, and there weren't even that many println() statements… Apparently they are very expensive)
- For QuickSort options 2 and 3, where insertion sort is done at 50 and 100 elements, for input data of size 50 (random, ordered, and reverse) since the input size is smaller than the partition size, we are effectively measuring the speed of an insertion sort.

References

zyBooks. 605.202: Data Structures. Module 7 – Trees.

zyBooks. 605.202: Data Structures. Module 9 – Huffman Encoding and Sorting.

zyBooks. 605.202: Data Structures. Module 10 – Sorting and Searching Sorted Data.