Willia Hardy
November 14, 2018
Data Structures Project
Student ID: 2905326

Project 3 Report

Manifest

- insertsort.h
  - code from book to implement insertion sort
- quicksort.h
  - code from book with slight modification to implement quick sort
- mergesort.h
  - code from book to implement merge sort
- heapsort.h
  - code from book to implement heap sort
- sorting.cpp
  - Main project code. Includes function definitions and implementation
- sorting.dat
  - File that is being read from, sorted, and data that is merged
- CMakeLists.txt/a.out
  - File used to compile and run the project
  - I used the "cmake .." command and make in order to create the executable
  - However, I had to use the g++ command to compile the project using current c++11 features. (**g++ -std=c++11 sorting.cpp**)

Summary of Project

This project was intended to use several sorting algorithms to determine the runtime of each algorithm based on a given dataset and random integers. The dataset given was in the sorting.dat file and the random integers were generated using for loops. The runtime was recorded and tested on the large dataset using 10,000, 100,000, and 1,000,000 integers. The user also had the opportunity to enter any number of integers to run the large dataset function on.

Answers to Questions

1. For each sorting algorithm, explain the difference in runtimes for the different type of data. That is, why do you think the randomized, ascending, descending data yields different/similar runtimes for the merge sort algorithm; is there an inherent reason with the way the code works? Answer this question for each algorithm; i.e. for heap sort, merge sort, quick sort and insertion sort. Explain please.
   a. Heap sort
      i. Heap sort yielded similar runtimes for ascending and descending data because heap sort did not have to divide its data into two regions since the data was already sorted. Since the randomized integers were not already sorted, it took a lot longer because the data had to be divided into a sorted and not sorted region to execute the algorithm. The runtimes were also similar for the increasing and decreasing data set because the data was already in a complete binary tree. It had to work harder for the decreasing data because the binary

tree has a condition of the root being the smallest element instead of the largest element.

b. Merge Sort
    i. The merge sort had similar runtimes for the ascending and descending data because merge sort does not care if the data is smallest to largest, or largest to smallest. The runtime is the same and the data was already sorted. It took longer to run the randomized integers because the data was not necessarily sorted, and it had to perform the division of equal halves and sort it.
c. Quick Sort
    i. None of the runtimes for quicksort were similar because the algorithm will sort the elements until the "less than" operation is achieved. The only set that achieve that without doing extra work was the ascending order set. All the elements were less than the smallest element.
d. Insertion sort
    i. Insertion sort took the longest amount of time out of all of the algorithms because it had to make (n – 1) number of passes and the n was either 10,000, 100,000, or 1,000,000 which were not small numbers. The only time the runtime was reasonable was when the insertion sort sorted data in increasing order. At that point, the algorithm did not have much to do because the data was already sorted using the same "less than" operation principle that quick sort has. The insertion sort also took too long to run the descending data so that runtime is inconclusive.

2. Explain the difference in runtimes between the insertion sort algorithm and quick sort. Why do you think there are differences/similarities?
    a. Insertion sort and quick sort did not have similar runtimes when using the randomized, ascending, or descending integers. However, they did have similar characteristics of how the runtimes compared for the ascending and descending data. It was noticeably higher between the quicksort ascending and descending and the insertion sort ascending and descending. I think that this is the case because whether the data is sorted or not, if the data is not in ascending order, each of the algorithms work to get the data in ascending order.
3. Explain the differences/similarities in runtimes between the heap sort, merge sort and quick sort algorithms. Why do you think there are differences/similarities?
    a. Heap sort and merge sort had very similar runtimes when the data was in ascending or descending order. This makes sense to me because both algorithms divide the data into two sets. This division into two data sets should take the same amount of time whether is it merged or constructed into a binary tree. There were no similarities when the randomized integers were sorted between these two algorithms. I believe this to be true because merge sort is much faster than heap sort.
4. Table with Runtimes:

Willia Hardy
November 14, 2018
Data Structures Project
Student ID: 2905326

number of integers N	runtime									theoretical Big-Oh runtime		
	randomized integers			presorted in increasing order			presorted in decreasing order			random order	increasing order	decrease order
	10,000	100,000	1,000,000	10,000	100,000	1,000,000	10,000	100,000	1,000,000			
heap sort	0.007733	0.041343	0.454837	0.002546	0.031863	0.369988	0.002442	0.031702	0.360166	n log n	n log n	n log n
merge sort	0.003298	0.040016	0.456315	0.002322	0.026406	0.317562	0.002215	0.02684	0.299739	n log n	n	n
quick sort (no cutoff)	0.001763	0.020347	0.220819	0.000566	0.005997	0.075718	0.00101	0.011773	0.140433	n^2	n log n	n^2
insertion sort	0.207284	19.4976	1949.35	0.000134	0.001241	0.01315	0.430763	38.9456	n/a	n^2	n^2	n^2