Ryan Pascal
rdp27@zips.uakron.edu
2932732

Project 3

Data structures 001

Dianne Foreback

Name

Ryan Pascal

rdp27@zips.uakron.edu

2932732

Manifest of files

1. **Project3.pdf**

Report itself

File Location: Project3/

2. **sorting.cpp**

Where main exists and execution begins.

File Location: Project3/

3. **heapsort.cpp**

heap sort cpp file.

File location: Project3/heapsort/heapsort.cpp

4. **heapsort.h**

heap sort header file

File location: Project3/heapsort/heapsort.h

5. **insertsort.cpp**

insertion sort cpp file.

File location: Project3/insertsort/insertsort.cpp

6. **insertsort.h**

insertion sort header file

File location: Project3/insertsort/insertsort.h

7. **mergesort.cpp**

merge sort cpp file

File location: Project3/mergesort/mergesort.cpp

8. **mergesort.h**

merge sort header file

File location: Project3/mergesort/mergesort.h

Ryan Pascal
rdp27@zips.uakron.edu
2932732

9. **quicksort.cpp**

quick sort cpp file

File location: Project3/quicksort/quicksort.cpp

10. **quicksort.h**

quick sort header file

File location: Project3/quicksort/quicksort.h

11. **sorting.dat**

Example data file

File Location: Project3/

Project Summary

The purpose of this project is to compare the runtime of various sorting algorithms at different input sizes. By analyzing runtime, the importance of runtime on large datasets will be emphasized.

The implementation begins by prompting the user for an input file and then a dataset size. The input file is then sorted using multiple sorting algorithms and the results are shown. Then from inputted data size datasets will be generated in random, descending, and ascending order. These datasets will then be put through the algorithms with the runtimes compared for each algorithm. The runtimes for each algorithm will then be displayed for the user to compare.

Questions

a) 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.

Heap sort

Heap sort works by building a max heap from the input data. Once the largest item is stored at the root it is replaced by the last item of the heap and the size is reduced by 1. This is repeated until the size of the heap is 1. For this reason, it makes sense that all the runtimes were very similar. For each input set the data will need to be heapified resulting in the runtimes to be similar across the board.

Merge sort

Merge sort works by dividing the input array into two halves and continues to do this until there is only one element in each half. Once that is done the halves will be merged back together but sorted. No matter what the input set is the algorithm will recursively break down the input set

Ryan Pascal
rdp27@zips.uakron.edu
2932732

into halves until there is only one element in each half. For the presorted input dataset, it makes sense it ran a little quicker because there was no need to sort the halves when merging them back together.

Quick sort

Quick sort works by picking an element as a pivot and partitions the given array around the picked pivot. All the elements smaller than the pivot will be placed on the left and all larger on the right. The best case is when a pivot is selected that divides the lists into two nearly equal pieces. If unbalanced partitions occur this results in increased times. For this reason, on the data set of increasing order and decreasing order it should run faster if the center is selected as the pivot. With these two cases the partitions will be even in size resulting in an ideal quick sort scenario. For random order depending on the input dataset it could run at different times.

Insertion sort

Insertion sort works the same way we would sort playing cards in our hands. Starting at the second element in the array it checks if its smaller than the previous element and if so swaps them. This process repeats for every element in the array. For an input dataset of increasing order, no swapping will ever be done resulting in a very quick runtime. For descending order, it will run in the worst case because every element will need to be moved.

b) Explain the difference in runtimes between the insertion sort algorithm and quick sort. Why do you think there are differences/similarities?

Quick sort run much quicker then insertion sort in all cases except ascending order. For ascending order, it makes sense it was able to run quicker due to it running in linear time. It only iterates over all the elements with no actual swapping of elements. For descending order insertion sort will run in quadratic time which is exponentially slower then quick sort which should run in O(nlogn). With the proper select of a pivot, preferably the center, in a sorted dataset it will allow for quick for to create even partitions resulting in a quick runtime. There is the possibility that quick sort runs in the same time as insertion sort if the worst case is achieved. This would come from a bad pivot continually being selected resulting in very uneven partitions.

c) Explain the differences/similarities in runtimes between the heap sort, merge sort and quick sort algorithms. Why do you think there are differences/similarities?

Overall quick sort ran faster than the other algorithms. This makes sense because with the inputted dataset, increasing and decreasing order, will select a pivot point resulting in very even partitions. Heap sort stays constant across the board and this is because regardless of the dataset it will be in about the same time. Merge sort however, will have some variations depending on how the dataset comes in. With the ascending order dataset merge sort will not need to do any sorting when merging back together the divided pieces. All of these algorithms share the same average case of O(nlogn) which shows in the results by each of these algorithms giving similar times. The biggest difference is for the ability to exceed this average case especially shown in quick sort where the times are slightly faster than the other algorithms.

Ryan Pascal
rdp27@zips.uakron.edu
2932732

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	0.00651	0.083146	1.34744	0.005809	0.0793	1.16268	0.005721	0.074004	1.0861	O(nlogn)	O(nlogn)	O(nlogn)
merge sort	0.006866	0.096518	1.25054	0.005668	0.064404	0.889768	0.005039	0.0669	0.890017	O(nlogn)	O(nlogn)	O(nlogn)
quick sort (no cutoff)	0.003043	0.036456	0.500291	0.001198	0.023384	0.225928	0.002128	0.02657	0.365784	O(n^2)	O(nlogn)	O(nlogn)
insertion sort	0.53784	51.8012	5042.64	0.000264	0.002988	0.029727	1.03171	101.495	TOO LONG	O(n^2)	O(n)	O(n^2)