CMPT 435 - Spring 2019 - Dr. Labouseur

Assignment Two – Sorting, Searching and Hashing

Timothy Polizzi
Timothy.Polizzi1@Marist.edu

March 13, 2019

1 Sorts

For this assignment there were a number of sorts used to show the different amounts of time it takes for specific sorts to run. These sorts all had different run times and numbers of comparisons. A chart of my specific comparisons is listed below.

1.1 Comparisons

Sort Name	Comparisons	Asymptotic Running time
Selection Sort	221445	$\Theta(n^2)$
Insertion Sort	114975	$\Theta(n^2)$
Merge Sort	5413	$\Theta(n \log n)$
Quick Sort	6470	$\Theta(n \log n)$

1.2 Selection Sort

For selection sort, the absolute best and worst case scenarios ($\Omega()$ or $O()$) are both actually n^2. This is because the internal loop will always run through the whole list for as many items as are in the list, which ends up being the number of items times the number of items, or n^2.

1.3 Insertion Sort

For insertion sort, there is a bit more optimization. The sort has the ability to check if a given item in the list is in sorted order, which means that its best case is $\Omega(n)$ rather than a constant n^2. Even though its best case is n, the average and worst cases stay at n^2, which means it is not all that much faster, but it can check for items that are already sorted meaning it can cut down on more time.

1.4 Merge Sort

Merge sort is a distinct step up in terms of speed and comparisons, running at a blazing hot (relatively) pace of $n \log n$ no matter what case the code is running in. Due to the nature of divide and conquer, any problem no matter what case can be done with the same complexity.

1.5 Quick Sort

Quick sort is exactly like its name suggests: quick. It utilizes a pivot to make its divide and conquer in place allowing for less data usage than merge sort and usually in comparable time. But this comes at a costs: a chance of loosing the speed lottery. The worst case of Quick Sort is a stunningly slow n^2, while its average and best case scenarios are at the speedy pace of $n \log n$.

2 Searches

After sorting comes searching, where you only need to get one item from a long list of things. To do this two different searching algorithms: linear search and binary search.

2.1 Comparisons

Search Name	Average Comparisons	Asymptotic Running time
Linear Search	347	$\Theta(n)$
Binary Search	3	$\Theta(\log n)$

2.2 Linear Search

Linear search is just about the most basic and straight forward way to search. You simply iterate through a list until you find the item you are looking for. This understandably makes the best case scenario a constant time if the first item is what you are looking for, while the average and worst case are both n.

2.3 Binary Search

Divide and conquer once again takes the cake in terms of quickly doing things and doing them especially efficiently. In this particular case though, there is a prerequisite: the list must be sorted before it can be searched. This causes a bit more time than a basic linear search as it requires an additional step, but on a pre-sorted list will leave linear search in the dust with a worst case of $\log n$ which also happens to be its average case as well. With a best case of constant as well, it makes it the faster sort.

3 Hashing (with chaining)

Hashing doesn't need a whole lot of comparisons in order to find things, running only about 4 comparisons a search for my hash table. This is in an average of $\Omega(n)$ which is also it's worst case but can be as good as constant time for look-ups. This is because you could theoretically have all of your items have only one hash and essentially just be a linked list, which is far slower. A hash table tends not to have that however and will generally be much faster.