**Sorting Project: Code Documentation**

Brandon Michelsen

**Introduction:**

Sorting algorithms are a very important aspect of software development. Sorting algorithms have many practical uses. For example, when comparing two data sets, it is much easier to compare elements when they are sorted. Another application would be when you are trying to search data. You could sort the data, and then use binary search to lower the time it takes to search through the data set.

There are many sorting algorithms in existence, each with its own benefits and limitations. The two I chose to implement in my program were insertion sort and quick sort. The rest of this section will describe how both of these algorithms work from a theoretical standpoint.

Insertion sort is a rather simple algorithm to implement. It is comparison based, meaning it compares each element in the data set with the rest of the data set. The algorithm works in a similar fashion to how you would sort playing cards. First, the data set is split into sorted and unsorted portions – starting with the very first element being the sorted potion and the rest of the data set being the unsorted portion. The algorithm then compares the first element in the unsorted portion of the data set with the sorted portion – in the first case, this means comparing the very first element with the second element. If the element in the sorted portion is greater than the element in the unsorted portion, the unsorted element is inserted into the sorted portion before the checked sorted element. If the sorted element is less than or equal to the unsorted element, the elements stay in their positions. The sorted portion indicator then moves forward, and the next element in the unsorted portion is compared with the newly-grown sorted portion. This process continues until there are no more elements in the unsorted portion.

Quick sort is a bit more difficult to implement. This is because it makes use of recursive function calls. Quick sort algorithms begin by first choosing a pivot, which is simply a data item in the list that will be used for comparisons. To avoid the worst-case running timing for this algorithm, the pivot is usually chosen either at random or by taking the median of the first, middle, and last elements in the data set. The pivot must fit the following requirements: all the data elements to the left of it when it is in its final position must be less than it, and all the elements to the right must be greater than it. Once the pivot is chosen, it is inserted at the end of the list until the data is ready for it to be inserted into its final position. The rest of the data is then compared to the pivot, being swapped around until it is ready for the pivot to be inserted into its proper place. The way the data is compared to the pivot is as follows: starting from the far left, the data is compared to the pivot; if the data is greater than the pivot, the loop progresses through the data set, and the last location in the set that the loop was at is stored; if the data is less than or equal to the pivot, the data is swapped with the last location where there was no swap, the last location where there was no swap progresses and the loop progresses through the data set; finally, once the data reaches the pivot, the pivot is swapped with the last known location where there was no swap. Looking at the data after running this process, it can be seen the pivot is in its final position – all elements to the left of the pivot are less than it, and all elements to the right are greater than the pivot. This process is then recursively called for the portion of the data set to the left of the pivot and the portion of the data set to the right of the pivot. Once all the recursive calls have finished, it will be found that the data is sorted.

**Analysis:**

This next section describes the time complexity of the two algorithms, along with an expected runtime and measured runtime for different lengths of data sets. Tables and charts are included to make the analysis more tangible.

First of all, we have the insertion sort algorithm. The best-case runtime of this algorithm O(n). However, this is only with the ideal data, as both the average-case runtime and the worst-case runtime are O(n2).

The chart and graph below show the measured runtimes for insertion sort using varying data set sizes ranging from 10 to 1,000,000 (note: the runtimes are in microseconds and the graph only shows the range from 10 to 100,000):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data Set Size:** | **10** | **100** | **1000** | **10000** | **100000** | **1000000** |
| **Times (microseconds):** | 7 | 439 | 43988 | 4564099 | 4.7E+08 | Took too long for proper measurement |

The data in the graph above does not seem very intuitive, but if the data is plotted in Desmos along with the f(x) = x2 function, it can be seen that it is in the O(n2) range.

As can be seen, insertion sort runs fairly poorly, increasing quadratically with the amount of data.

As for quick sort, the time complexity is much better. For the best-case runtime, quick sort is O(n\*log n). It is the same for the average-case runtime. The worst-case runtime is O(n2).

The chart and graph below show the measured runtimes for different data set sizes ranging from 10 to 1,000,000 (note: the runtimes are in microseconds):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data Set Size:** | **10** | **100** | **1000** | **10000** | **100000** | **1000000** |
| **Times (microseconds):** | 11 | 132 | 2071 | 24278 | 341290 | 4028369 |

Again, the graph is a little bit non-intuitive. However, graphing the runtimes in Desmos along with the function f(x) = x\*log (x) will show that the algorithm’s runtime fits within O(n\*log n) time.

With quick sort, we can certainly see that there is better runtime compared to insertion sort. The very first indicator is that we were able to get a valid runtime for 1,000,000 data elements. The graph is able to fit within O(n\*log n), which is a much better runtime than insertion sort.

**Programmer’s Guide**

This section describes how both the insertion sort and quick sort algorithms are meant to be used.

**Insertion Sort Function:**

|  |
| --- |
| **Function: insertionSort** |
| template<typename T> |
| **Return Type:** |
| void |
| **Parameters:** |
| LinkedList<T>& list |

The insertionSort function is a sorting function implementing the Insertion Sort algorithm. It is templated with a comparable generic data type (type T). It is structured to sort a LinkedList object that stores data of type T (from the LinkedList class in linked-list.h). The algorithm checks for empty lists internally. The return type is void. The runtime complexity is O(n2) for the average and worst cases.

**Function Parameters**

|  |  |
| --- | --- |
| Parameter | Description |
| list | A reference to the list to be sorted, list stores data of type T (LinkedList<T>&) |

**Implementation details:**

The algorithm does not check if the entered data type for T is comparable. The programmer must ensure that the data type used with this function is of a comparable type.

**Quick Sort Function:**

|  |
| --- |
| **Function: quickSort** |
| template<typename T> |
| **Return Type:** |
| Void |
| **Parameters:** |
| LinkedList<T>& list  Node<T>\* low  Node<T>\* high |

The quickSort function is a sorting function implementing the Quicksort algorithm. It is templated with a comparable generic data type (type T). The function sorts a LinkedList object that stores data of type T (from the LinkedList class in linked-list.h). The function also accepts parameters for value representing the low end of the list and the high end of the list. These two values represent the partition of the list that is being sorted and are of type Node<T>\* (from the Node class in node.h). The function is recursive, as it calls itself for each new partition of the list until the data is sorted.

The function checks for empty lists internally. The return type is void. The runtime complexity is O(n\*log n) for the average case and O(n2) for the worst case.

**Function Parameters**

|  |  |
| --- | --- |
| Parameter | Description |
| list | A reference to the list to be sorted, stores data of type T (LinkedList<T>&) |
| low | A pointer of the low end (far left) of the list to be sorted (Node<T>\*) |
| high | A pointer to the high end (far right) of the list to be sorted (Node<T>\*) |

**Implementation Details:**

The function does not check if the entered data type for T is comparable. The programmer must ensure that the entered data type is comparable.

The quickSort function also makes use of the quickSortPartition function, which is documented below:

**Quick Sort Partition** **Function:**

|  |
| --- |
| **Function: quickSortPartition** |
| template<typename T> |
| **Return Type:** |
| Node<T>\* |
| **Parameters:** |
| LinkedList<T>& list  Node<T>\* low  Node<T>\* high |

The quickSortPartition function is a helper function for the quickSort function. It is templated to use comparable generic data types (type T). The function is used internally by the quickSort function to properly partition the list. The function takes a parameter for the list to partition as well as pointers to the low and high boundaries of the partition. The return type is of Node<T>\*.

**Function Parameters:**

|  |  |
| --- | --- |
| Parameter | Description |
| list | A reference to the list to be partition, stores data of type T (LinkedList<T>&) |
| low | A pointer of the low end (far left) of the list to be partitioned (Node<T>\*) |
| high | A pointer to the high end (far right) of the list to be partitioned (Node<T>\*) |

**Implementation Details:**

The function returns a pointer to a Node. It is up to the programmer to check for a null pointer.

**Usage Examples:**

Below are the examples of how to use the insertionSort and quickSort functions. In each example, ensure you have the location of the linked-list.h and node.h files properly included into your project.

**insertionSort Example:**



**quickSort Example (quickSortPartition is included in this example):**

