Lince Rumainum
Student ID: 112722337
CS 2413 Data Structures
Project 2

For Project 2, we have chosen the approach to keep the list to be sorted as we read the file. This approach is appropriate since we are using a linked list. Unlike the array, in linked list, when we are inserting a new data into the sorted list, we do not have to move any data from its location when it is between two existing data or shift all data when the new data is the minimum value in the list. That has made the runtime for inserting data in linked list to be efficient. Although we have to do linear search to find the location to insert a new data that is larger than the first but less than the last existing data, we can easily append data when the new data is larger than the last node without doing a linear search through the linked list since we have a node that keep track of the last data (current maximum data). We also could efficiently insert a new minimum data by making a new node and rearrange where each node pointing to since we have the node where the list started (current minimum data). Since we have access to what each node pointing to next, we can keep track the chaining of the linked list.

There are several functions inside the linked list design that enable us to insert and append non-duplicate data. We are also can get the first data in the list or find a specific data in that list and when data that is in the list become unwanted then we can either remove or replace that data. There are also functions that help user keep track of how many data is present in the list and going through that list. Knowing the size of the list and having enumerator helps to create an array that can be used for sorting the data and searching from the sorted column other than the record ID number. Having an enumerator helps with traversing through the list node by node and help to see if the list is empty or at the end of the list by peeking into the list and see if there a next node exists.

From creating the linked list and reading the data the insertion time and merge time for already sorted list is as efficient as Big O (1). By storing the first and last node of the linked list and keeping track of what does it pointed to next, we are able to efficiently insert and merge data into the linked list. The append insertion time of the chosen approach of this project is Big O(1) while the run time to search and then inserting the data that is between first and last is Big O(n) due to the linear search behavior of the linked list but still have a Big O (1) for inserting the data since we just change where the node point to next and we do not have to move any data before or after that inserted node when we use a linked list.

Merging the two files for the chosen approach is a Big O (n) for however *n*-numbers of data being added. Although, sorting the list is essentially the same as the insertion time since we are keeping the list to be sorted as we read in each new data and we do keep track of the first and the last nodes of the link are. The chosen approach is effective since the merging and sorting happen simultaneously with sorting time of Big O (1).

On the other hand, he insertion time of the alternative approach (Approach B) is the same as the chosen approach, Big O (1), due to the nature of a linked list. Although, the data might need to be re-inserted and pointing to a different node later. The merge time of Approach B is also going to the same as the approach, a Big O (n), given that the same number of data is being handled for both approaches. Since the list is being kept unsorted, each data can be appended one by one until the last one. The difference between the chosen approach and the Approach B happens in sorting time. When the list of data needs to be sorted from unsorted lists, the best method to use is using a merge sort, where the Big O runtime complexity is n log n.