CS 2302 - Lab 3

Sofia Gutierrez

Olac Fuentes

**Introduction**

The objective of lab 3 was to create a program to implement a ‘SortedList’ class that would first contain some appropriate functions such as append, in order to initialize a linked list. The completed program was then required to have an additional ten other functions such as insert which would add a desired integer to either an empty list or an already built list, and delete, which would simply remove a desired integer from the list. The same functions were repeated for a normal ‘List’ class with minor changes in order to accommodate for the unsorted numbers within this normal list. This was done in order to compare the running times of both the ‘SortedList’ class and the ‘List’ class.

**Design and Implementation**

The program would first contain the Node class and would include the constructor needed to assign the keywords ‘data’ and ‘next’ which are vital for linked list. Next came the ‘SortedList’ class which would then initialize the keywords ‘head’ and ‘tail’ and after include its first function ‘Print’. This ‘Print’ function would simply iterate through the existing list of sorted elements and print the data individually. The class then called for the functions ‘Append’ and ‘AppendList’ which were solely needed to make it possible to add an element to a list. Then came the ‘Insert’ function which would work similar to ‘Append’ except that ‘Insert’ was used for when the user wanted to add an element to the existing list created by ‘Append’. The way in which ‘Insert’ worked was that it would first examine if the list created by ‘Append’ was empty. If it was not empty, which in this case it was not, then a Node would be created to carry the value of the desired integer and would then be compared to the already existing Nodes within the sorted list. This was done in order for the new Node to be inserted in the correct position and so that the sorted list would remain sorted. The function ‘Delete’ followed, which would iterate through the list until the data of a Node equaled that of the integer that was sought after to be removed. If the integer was not found with the list, then the list remained unchanged.

The next function to follow was ‘Merge’ which needed a whole new sorted list to be created which was done with the help of ‘Append’. The function ‘Merge’ would simply then use the function ‘Insert’ to combine both the new sorted list and the original sorted list. By using the function ‘Insert’ the list after the merging would remain sorted. The function ‘IndexOf’ followed which served its purpose to merely return the index of where ‘i’ was held, ‘i’ being an integer. To this a counter was initialized and once again the list was iterated through until the counter matched with the integer ‘i’ and the data of that index would be returned. Of course, the counter would be added by a one after the first iteration and so on, basically every time the comparison was made and not satisfied. It would return -1 if the index exceeded the length of the list. Then next came the function ‘Clear’ which would simply clear the list by setting the ‘head’ and ‘tail’ to None. Next were the functions ‘Min’ and ‘Max’ which would return the ‘head’ data (smallest element) and the ‘tail’ element (largest element) in the list. Then came the function to check for any duplicates within the list, ‘HasDuplicates’. This function worked by again iterating through the list until the whole list was checked for any duplicates. More specifically this was done by comparing the data of the current Node with the data of the next Node. Since the list was already sorted this was an easy task. Finally, the last function was implemented which was named ‘Select’. This functions purpose was to return the kth smallest element in the sorted list and would do so by again setting a counter and a while loop to iterate through the list. This counter would be compared to the data of the current Node and would return the data if the counter was equal to the integer ‘k’ which again was the kth smallest in the list. If integer k exceeded the size of the list then math.inf would be returned.

The next class, ‘List’ was implemented. The functions for this class were the same except ‘Min’, ‘Max’, and ‘Select’. This was solely because of the fact that the List class used an unsorted list instead of a sorted one and therefore the functions could not depend on the list being sorted to ease their task. After that class, the ‘main’ section was needed. Of course, this is where all the functions were called in order for the outputs to be printed. A user-friendly menu was implemented which displayed the options of which functions the user wanted to perform. After one of the functions were chosen then the outputs for both the sorted list and the unsorted list were printed.

**Experimental Results**

**Figure 1:**

|  |  |  |
| --- | --- | --- |
| **Function** | **SortedList** | **List** |
| **Print()** | O(n) | O(n) |
| **Insert(i)** | O(n) | O(n) |
| **Delete(i)** | O(n) | O(n) |
| **Merge(M)** | O(n log n) | O(n log n) |
| **IndexOf(i)** | O(1) | O(1) |
| **Clear(i)** | O(1) | O(1) |
| **Min()** | O(1) | O(n) |
| **Max()** | O(1) | O(n) |
| **HasDuplicates()** | O(n log n) | O(n log n) |
| **Select(k)** | O(1) | O(n) |

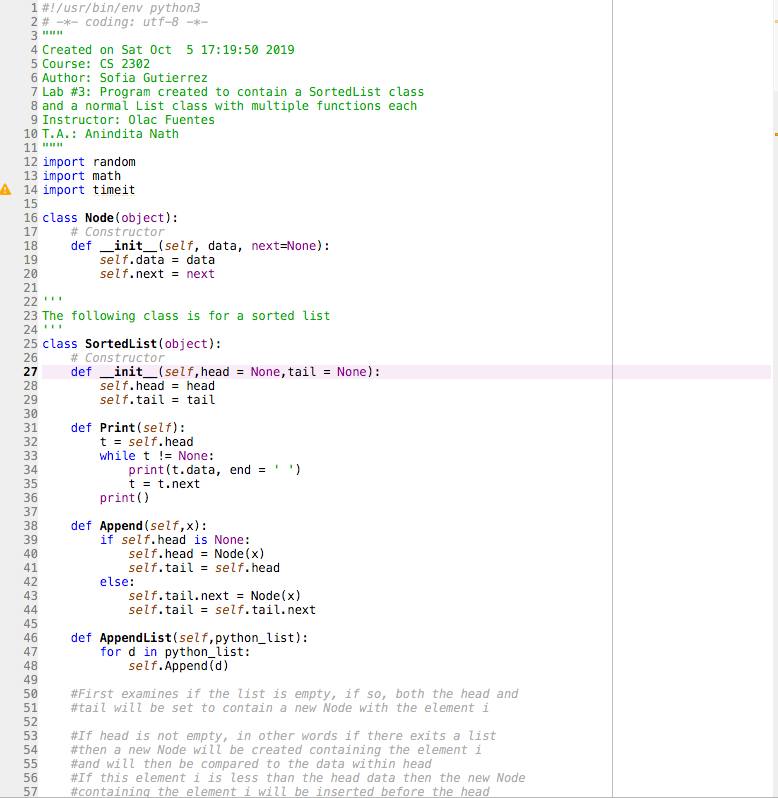
**(Figure 1) depicts a table showing the big-Oh runny times of the functions implemented for the ‘SortedList’ class and the ‘List’ class.**

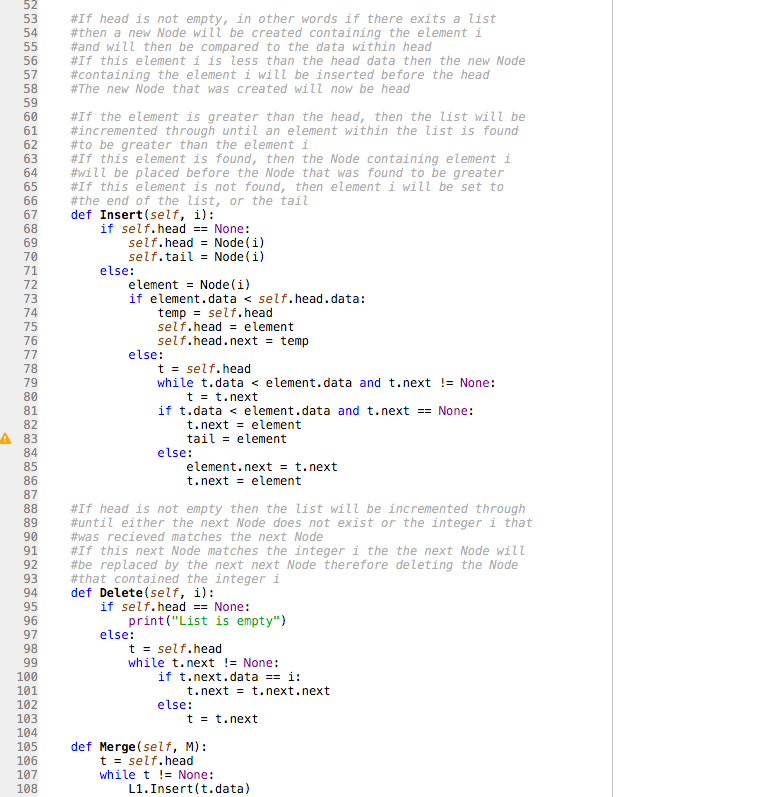
The table shows that the ‘SortedList’ class contained the same time complexities to that of the normal ‘List’ class which of course was expected since the same functions were used for both classes. The time complexities differed for the functions ‘Min’, ‘Max’, and ‘Select’ which were changed to accommodate to the unsorted list. The time cases for these functions in the ‘SortedList’ class proved to be better than the time cases for the ones in the ‘List’ class. These changes were not staggeringly different from one another considering O(1) is considered an excellent time complexity and O(n) is still considered good.

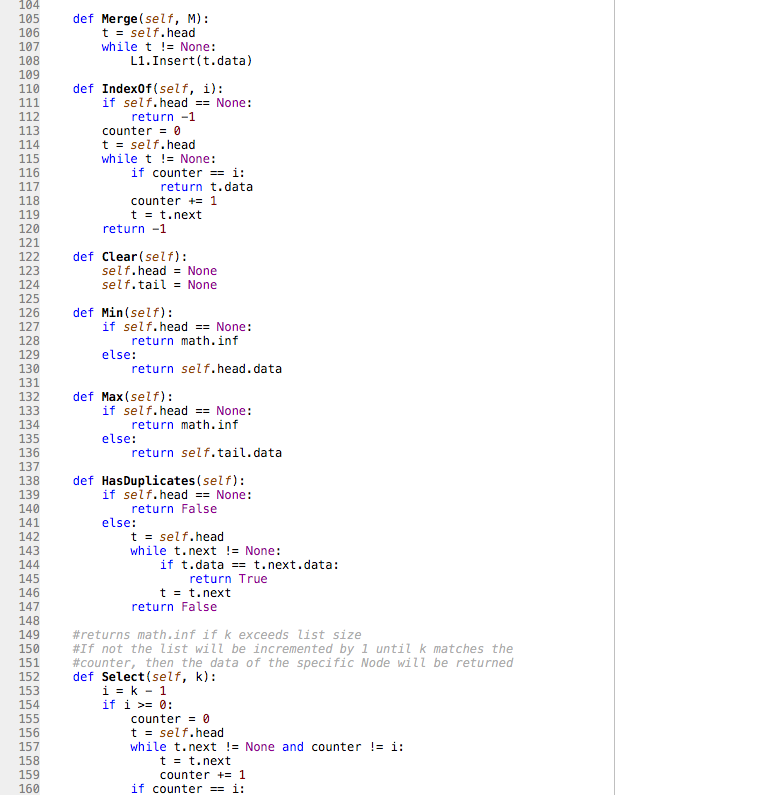
**Conclusion**

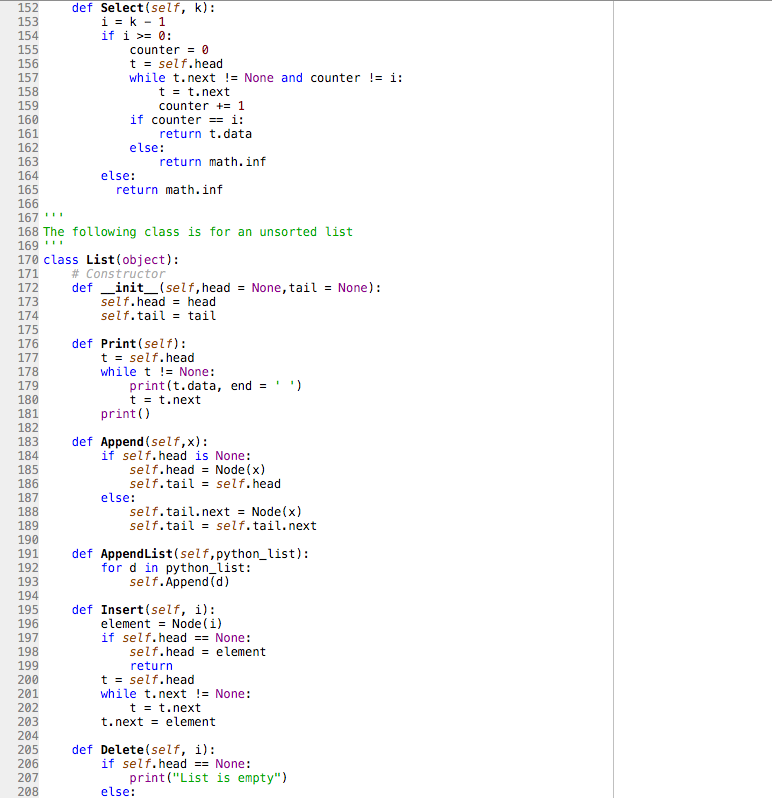
This program contains many basic functions needed to either create simple changes to a list, whether sorted or unsorted, or return data from a list. Many of them had while loops implemented which made it easy to iterate through both lists, and I learned that in order to visit every Node within any list, a while loop is most efficient. I also became more familiar with the ‘main’, an area which I was not so comfortable with before this lab. I also learned of the importance of using classes with their appropriate constructors and why they are necessary in order to create a successful program. And finally, I learned that a sorted list is most efficient when time complexity is involved.

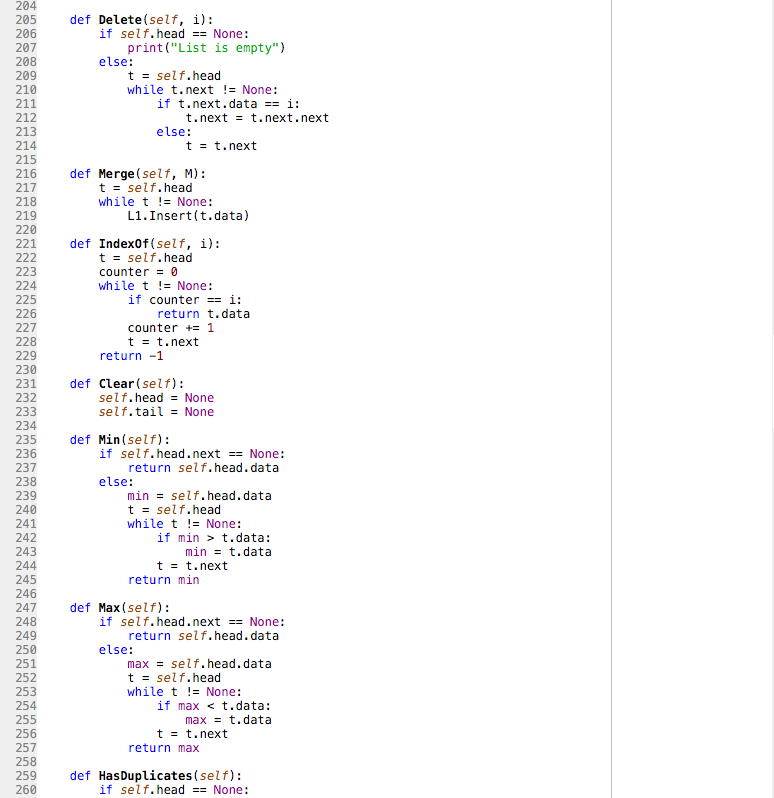
**Appendix**

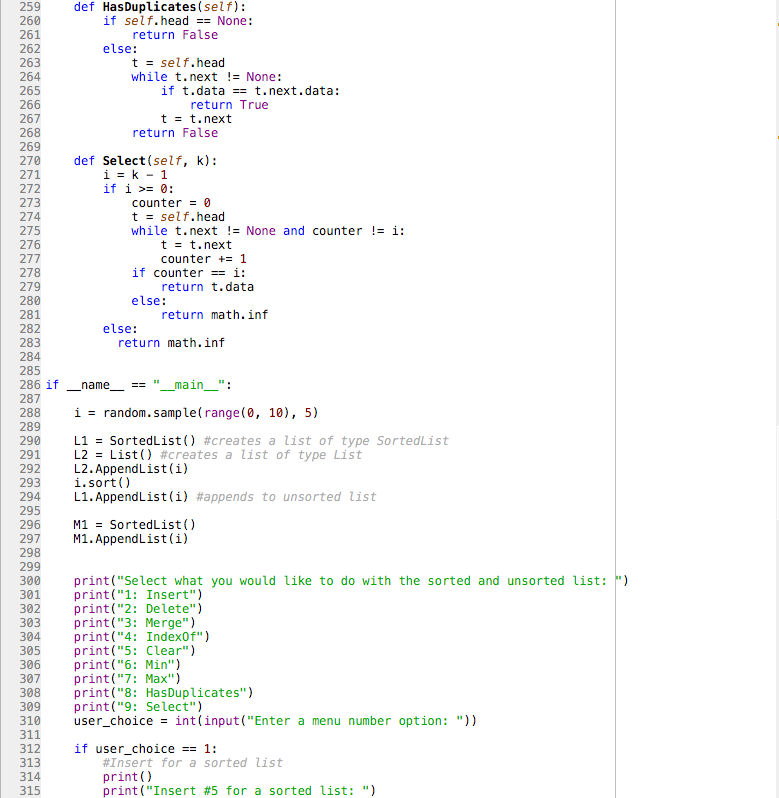




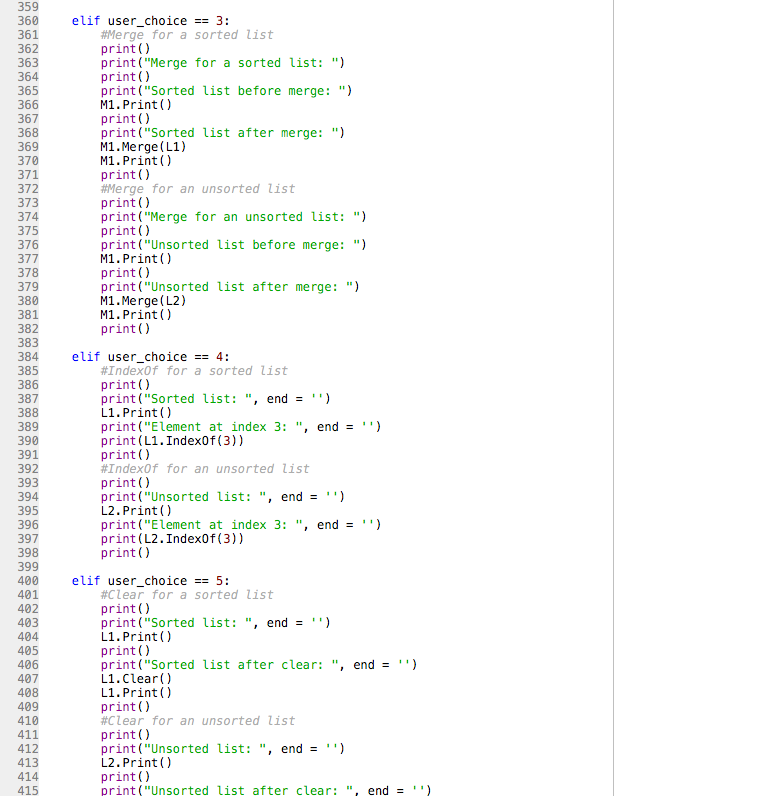


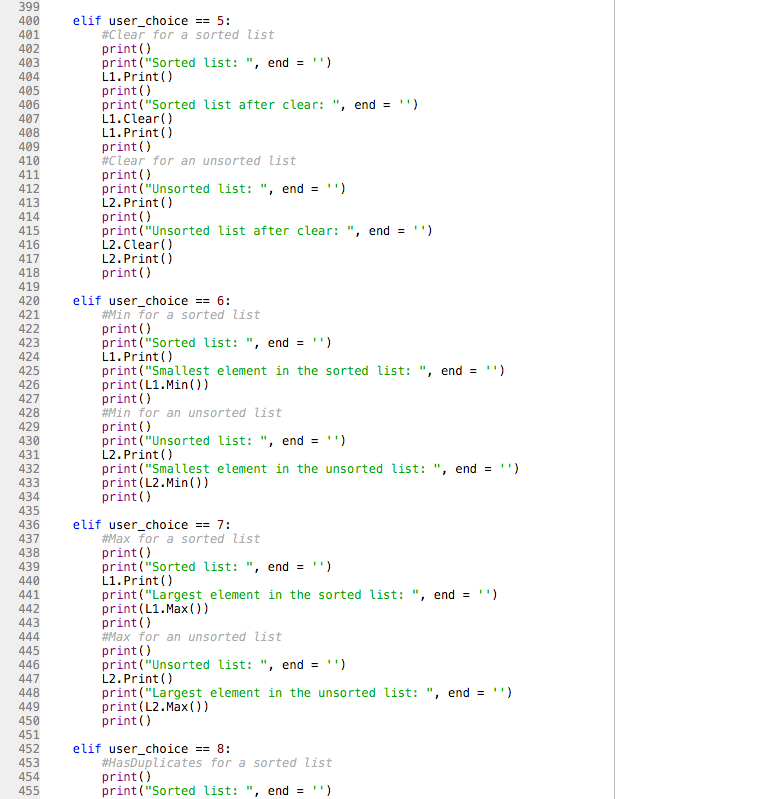


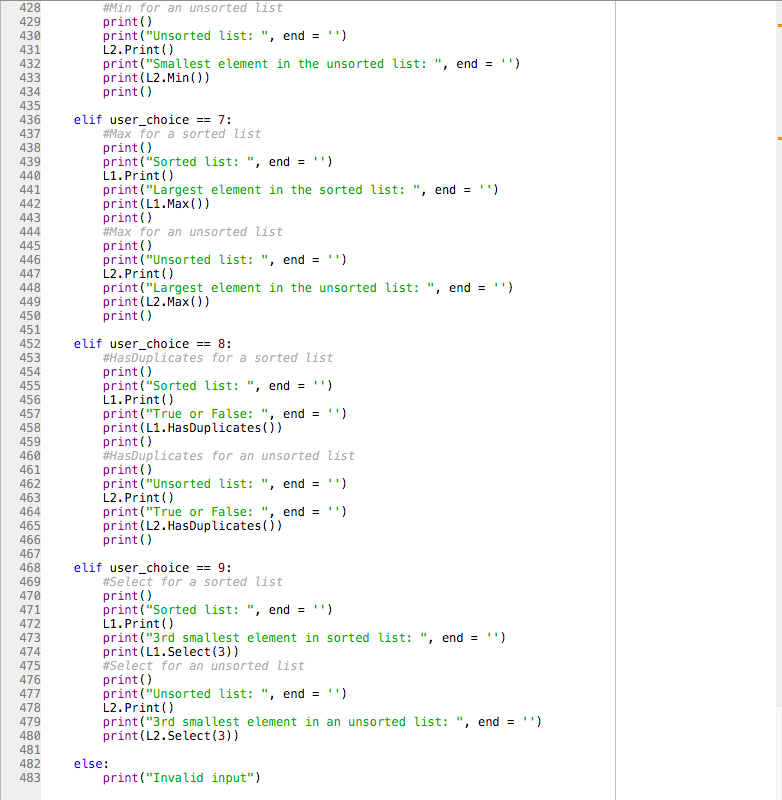












**Academic Honesty Certification**

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

Sofia Gutierrez September 9, 2019