**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #3**

Due: October 8th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we were asked to implement a *SortedList* class that would operate like the *List* class, but its elements would be in ascending order at all times. To do this, we must be familiar with the behavior of nodes and implement ten different methods in both classes. The main objective of this lab is to recognize how the very nature of being sorted or unsorted influences the running times of the methods of either class by taking note of their big-O running times.

**Proposed Solution Design and Implementation**

**Operation #1 (Print(*self*)):**

This operation was actually provided for by Dr. Fuentes for the *List* class, and it is no different for the *SortedList* class. The idea in both methods is to start with a reference to the head node of the given list before traversing through it using the reference node’s next pointer, and simply print out each node as it’s referenced.

**Operation #2 (Insert(*self, i*)):**

For this operation, whereas the method in the *List* class simply traverses to the list’s next empty slot and fills it with the given element *i*, the method in *SortedList* class must be able to know when to stop traversing the list in order to place *i* in its proper place, thus maintaining the order of the sorted list. To note, *i* is placed into a node *i* such that it can be inserted into the list.

To do this, the method first checks if either the list is empty, in which case *i* is simply made head and tail, or if *i* is less than the value held by the head node’s data, in which case it is pointed to the head node and then assigned as the new head node. If neither of these special cases are true, then the method will begin to traverse the list. From here, it will continue to traverse until either the tail node is reached, in which case *i* is made the new tail, or the current node’s next node has data greater than *i*, in which case *i* is pointed towards that next node and the current node is now pointed towards *i*, thus inserting the list in its proper place.

In the *List* class, this method always has a running time of O(1), as insertion requires either a single assignment of its head and tail pointers to *i* if the list is empty, or just an assignment of its former tail’s next pointer and its new tail pointer to *i*. In the *SortedList* class, this method will have a running time of O(*n*), where *n* is the length of the list, as the worst case is if *i* is greater than every element in the sorted list, and must thus be placed at the end of the list only after the entire list has been traversed.

**Operation #3 (Delete(*self, i*)):**

For this operation, the method in the *List* class must always traverse the entire list and check every node along the way in order to determine if its data is equal to *i* in order to remove it by changing its preceding node’s next pointer to its own next pointer. This means the method will always have a running time of O(*n*), where *n* is the length of the list.

Similarly, the method in the *SortedList* class can, in its worst case, traverse the entire list in order to reach the node(s) to be deleted, as in this case, *i* would be equal to the greatest value in the list; however, what stops this from always being the case is that in a sorted list, traversal can be stopped once a value greater than *i* has been reached, as that would imply that a node with data equal to *i* does not exist. What’s more, in the best case, since the list is sorted, we can determine before traversal even begins if *i* is outside the range of the list by comparing it to the head and tail nodes. If this is true, the method can simply do nothing and end. That said, the big-O running time of the method is of course still O(*n*).

**Operation #4 (Merge(*self, M*)):**

For this operation, in the *List* class, all the method need do to merge the current list with another list *M* is point its tail’s next pointer to the head node of *M* and then update its tail pointer to that the tail node of *M*. This is because the resulting list need not be sorted, and results in a running time of O(1).

The method in the *SortedList* class, on the other hand, is ultimately more complex given the requirement of the resulting list being sorted. That said, the method benefits from *M*, in this instance, referring to its own sorted list, thus accelerating the entire procedure. In this method, we start with the head node of *M* and traverse through the list, and along the way, use the previously established *Insert()* method to place each of *M*’s node’s data values in their proper positions into the current list. Though simple to code, this does unfortunately take a method of O(*n*) running time, where *n* is the length of the current list, and refer to it *n* times, where *n* is the length of list *M*, thus resulting in a total running time of O(*n*2).

**Operation #5 (IndexOf(*self, i*)):**

For this operation, in the *List* class, the method must again traverse, at worst, the entire list in search of a node with data equal to *i* in order to return its index in the list. To keep track of the index, an integer variable is incremented each time the list is traversed from one node to the next until such a node is found, at which point traversal can end and the integer variable representing the index can be returned. The unsorted nature of the list again lends to this method have a running time of O(*n*), where *n* is the length of the list.

As for the method in the *SortedList* class, it is actually much the same, except again the method can bypass executing anything entirely by checking to see if *i* is out of the range of the list, utilizing the nature of sorted lists; however, in its worst case, the method could be asked to return the index of its next to last node, thus requiring traversal through the entire list, resulting in a running time of O(*n*), where *n* is the length of the list.

**Operation #6 (Clear(*self, i*)):**

For this operation, the method is exactly the same in both the *List* and *SortedList* classes. Since the aim of the method is to remove all elements from the list, be it sorted or unsorted, we can simply set the head and tail pointers equal to None, thus resulting in a running time of O(1).

**Operation #7 (Min(*self*)):**

For this operation, the method in the *List* class is in much the same vein as the *Delete()* and *IndexOf()* method mentioned previously in that the list must be traversed entirely in order to find the minimum value of the list. Like the *IndexOf()* method in particular, a variable is created to store the current minimum value as the list is traversed, first being set to positive infinity so that even a single number within the list can be determined as larger than said variable and thus deemed the current minimum value. In this fashion, the list is traversed, making comparisons between every node’s data and the current minimum value until no smaller value is found and the minimum value is returned, thus resulting in a running time of O(*n*), where *n* is the length of the list.

The equivalent method in the *SortedList* class is much simpler, as, again utilizing the nature of a sorted list, it can easily be determined that the head node of the list is the smallest value, and can thus be returned with no traversal necessary. The only case to check is if the list is empty, in which case positive infinity can be returned to indicate as much. The lack of any procedure thus leads to a constant running time of O(1).

**Operation #8 (Max(*self*)):**

For this operation, the method in the *List* class is practically the same as the previously mentioned *Min()* method. Again, in an unsorted list, the method must traverse the entire list and keep track of the largest value with a variable first set to negative infinity. This results in a running time of O(*n*), where *n* is the length of the list.

Again, the equivalent method in the *SortedList* class is much simpler, utilizing the nature of a sorted list to determine that the tail node contains the largest value of the list, and can thus be returned unless the list is empty, in which case, negative infinity is returned to indicate this. The running time of this method is constant, or O(1).

**Operation #9 (HasDuplicates(*self*)):**

For this operation, the method in the *List* class must compare each of the list’s node to each and every other node in the list. To do this, the list is traversed once with one reference to the head, and at each step of the traversal, the list is traversed in its entirety with a second reference to the head. At each step of this second traversal, a comparison is made between the current nodes of the first and second traversals, and if said nodes have equal data but are not referencing the same node, then a boolean value of *True* can be returned; however, if this condition is not met, the list will be traversed in its entirety *n*2 times, where *n* is the length of the list, returning a boolean value of *False* and thus having a big-O running time of O(*n*2).

The equivalent method in *SortedList* class can again utilize the nature of a sorted list for a more streamlined process. The list is traversed simply once, and at each step of the traversal, the current node (which starts at the head node) is compared to its next node, and, if these two nodes have equal data, a boolean value of *True* can be returned. This is because in a sorted list, any duplicates will be placed next to one another, one being pointed to by the other’s next pointer. If the entire list is traversed and this condition is never met, then a boolean value of *False* can be returned. The big-O running time of this method is O(*n*), where *n* is the length of the list.

**Operation #10 (Select(*self, k*)):**

For this operation, it is best to go over the method in the *SortedList* class first. Essentially, the method must return the *k*th smallest element in the list, so in a sorted list, we can simply traverse the list *k* times and then return the current node. The only case to consider is if the end of the list is reached before traversal has occurred *k* times, in which case, positive infinity can be returned to indicate this. In this instance, the big-O running time will actually be constant, or O(1), as the amount of traversal necessary is mandated not by the length of the list, but by the value of *k*, which, since it is provided as a parameter, is treated as a constant within the method.

To recreate this method in the *List* class, it is an inevitability that the list must first be sorted before the *k*th smallest element can be determined. Thus, the simplest way to do this is to make a *SortedList()* copy of the current list, and then use the same method as described above. This copy is made by creating a new *SortedList()* and using the *Insert()* method of the *SortedList* class to insert the values of each node of the current list. This extra step to the procedure causes the big-O running time to be O(*n*), where *n* is the length of the current list, and since the method in a sorted list is constant, the total running time becomes O(*n*).

**Experimental Results**

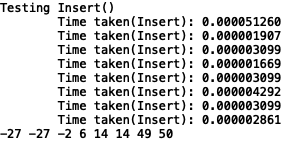
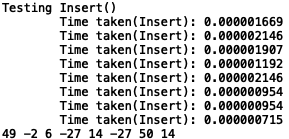
**Operation #1:**

For this operation, the lists in their states after the other methods are used will be printed after each other operation is tested.

**Operation #2:**

For this operation, I will insert eight variables, including duplicates, both to show the differences between the equivalent methods in the two classes as well as to find average running times for the methods in both classes.

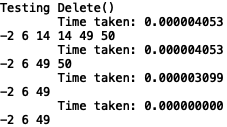
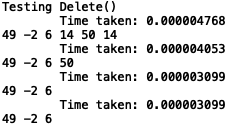
SortedList(): List():

****

**Operation #3:**

For this operation, I will delete the duplicates and the largest elements from the lists, thus testing the ability of the methods to delete duplicates and finding both the worst case running times and the overall average running times.

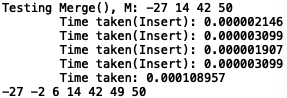
SortedList(): List():

**Operation #4:**

For this operation, I will create new lists *M* to test the methods in both classes. Of note is that the method in the *SortedList* class uses the *Insert()* method, thus taking more time overall. From this, the overall running time of each method can be found, and the resulting lists continuing to be sorted or unsorted is also illustrated.

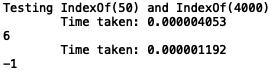
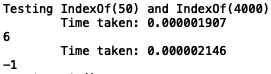
SortedList(): List():



**Operation #5:**

For this operation, I will find the index of 50 (which happens to be the last element in both lists) and the index of 4000 (an element not present in either list). This will showcase their worst running times and the capability of SortedList to recognize that the sought for element is out of bounds.

SortedList(): List():

**Operation #6:**

For this operation, I will test to see that the methods do clear the lists of all elements, as (perhaps not clearly) indicated by the blank lines printed out.

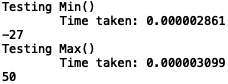
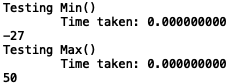
SortedList(): List():

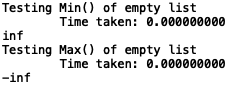
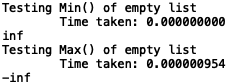
 

**Operation #7:**

For this operation, I will test to find the minimum values of the lists before and after they’ve been cleared to show that empty lists can be handled as well as to show that for a sorted list, the method requires almost no time at all.

SortedList(): List():

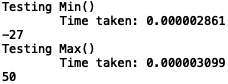
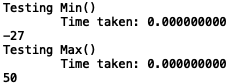


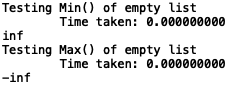
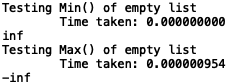


**Operation #8:**

For this operation, I will test to find the maximum values of the lists before and after they’ve been cleared to show that empty lists can be handled as well as to show that for a sorted list, the method requires almost no time at all.

SortedList(): List():





**Operation #9:**

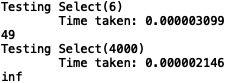
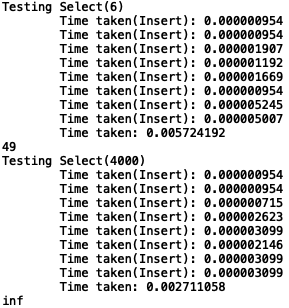
For this operation, I will check to see if the lists have duplicates after they’re first filled with elements (thus resulting *True*) and after those duplicates have been deleted and lists with no duplicates are merged into the current lists (thus resulting *False*).

SortedList(): List():



**Operation #10:**

For this operation, I will test to see that the methods can find the 6th and 4,000th smallest elements of their respective lists to show that the added process of sorting for the *List* class adds far more running time and that a call to these methods for an index out of bounds can be handled.

SortedList(): List():

**Tables:**

**Big-O Running Times**

|  |  |  |
| --- | --- | --- |
| **Function** | **SortedList** | **List** |
| **Print()** | O(*n*) | O(*n*) |
| **Insert(*i*)** | O(*n*) | O(1) |
| **Delete(*i*)** | O(*n*) | O(*n*) |
| **Merge(*M*)** | O(*n*2) | O(1) |
| **IndexOf(*i*)** | O(*n*) | O(*n*) |
| **Clear(*i*)** | O(1) | O(1) |
| **Min()** | O(1) | O(*n*) |
| **Max()** | O(1) | O(*n*) |
| **HasDuplicates()** | O(*n*) | O(*n*2) |
| **Select(k)** | O(1) | O(*n*) |

**Experimental Running Times**

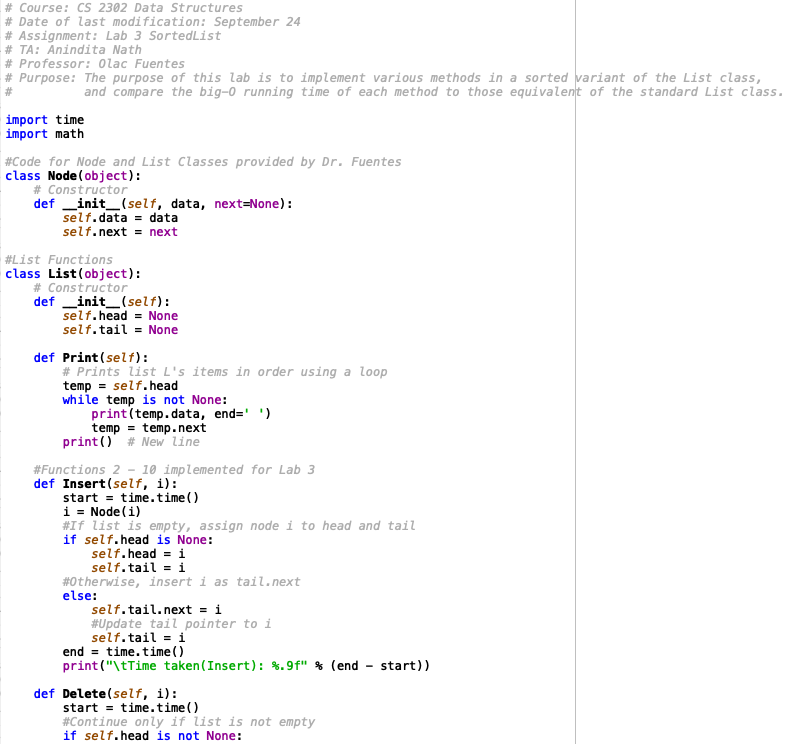
|  |  |  |
| --- | --- | --- |
| **Function** | **SortedList(n x 10-6 ms)** | **List(n x 10-6 ms)** |
| **Insert(*i*) (on average)** | 8.722 | 1.460 |
| **Delete(*i*) (on average)** | 2.801 | 3.754 |
| **Merge(*M*)** | 108.957 | 0.000 |
| **IndexOf(*i*)** | 2.622 | 2.026 |
| **Clear(*i*)** | 1.907 | 0.954 |
| **Min() (non-empty)** | 0.000 | 2.861 |
| **Max() (non-empty)** | 0.000 | 3.099 |
| **HasDuplicates()**  **(on average)** | 2.980 | 13.470 |
| **Select(k) (on average)** | 5.245 | 4,217.625 |

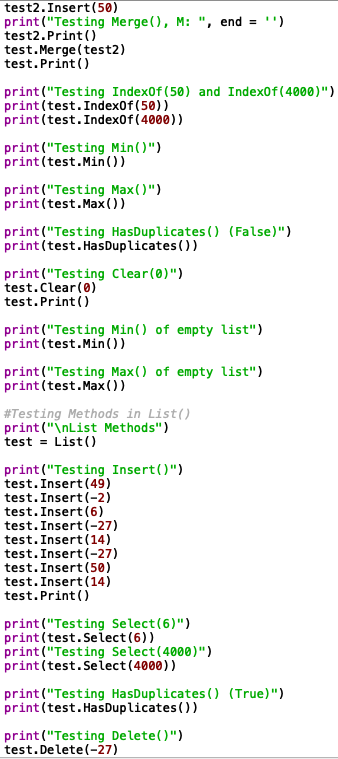
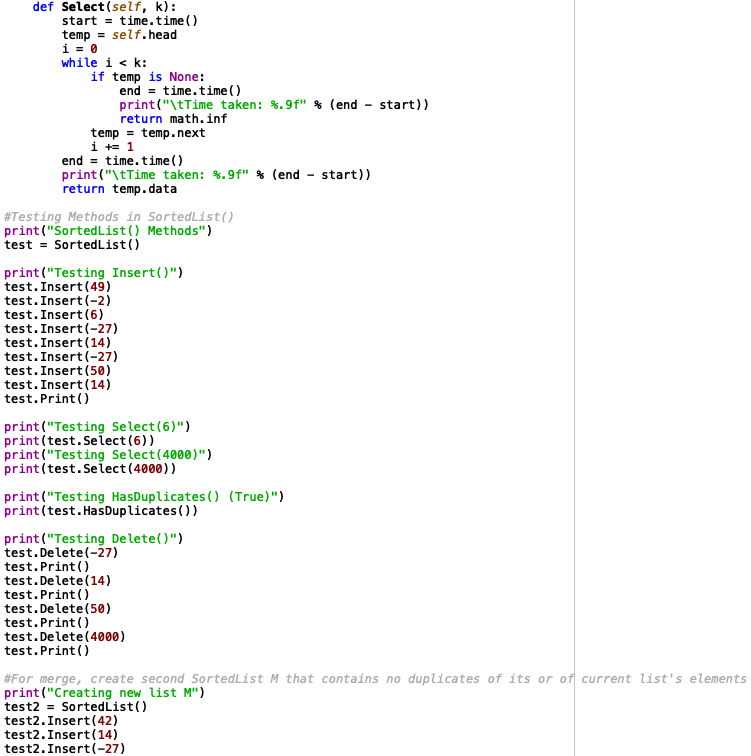
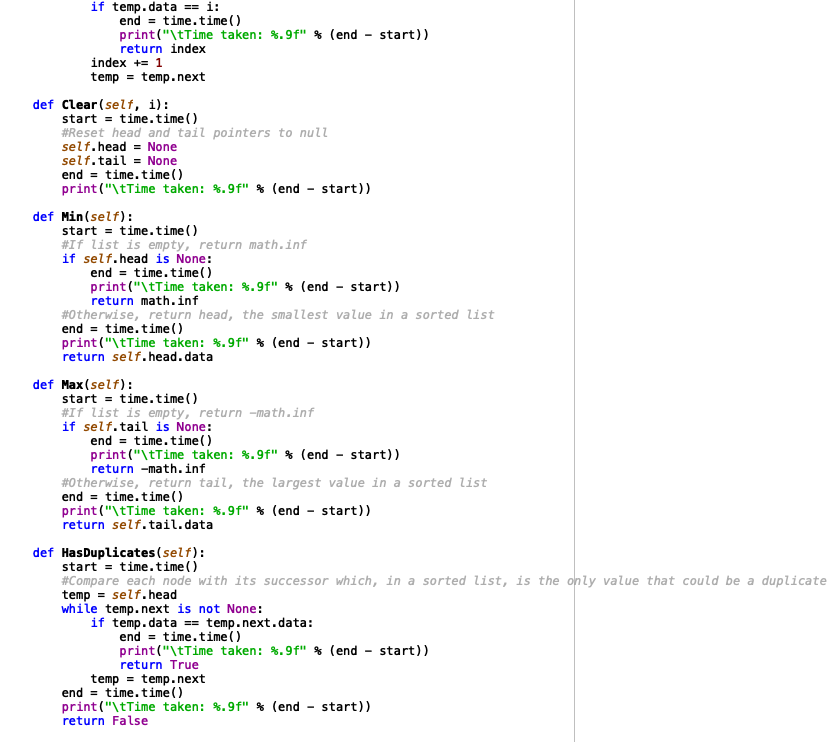
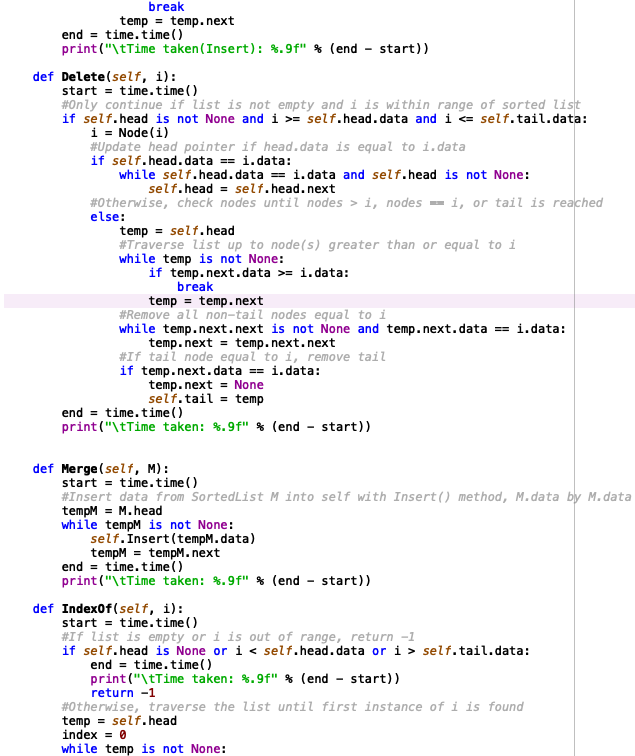
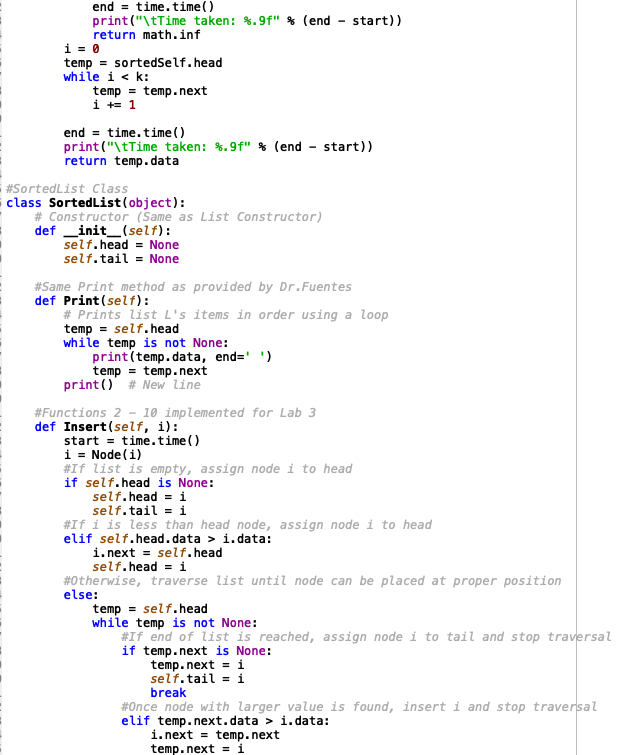
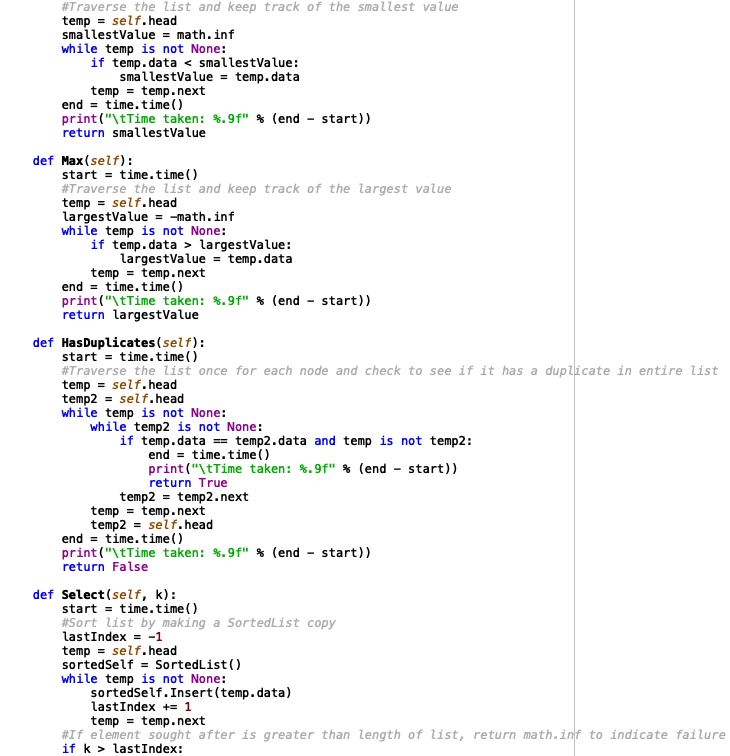
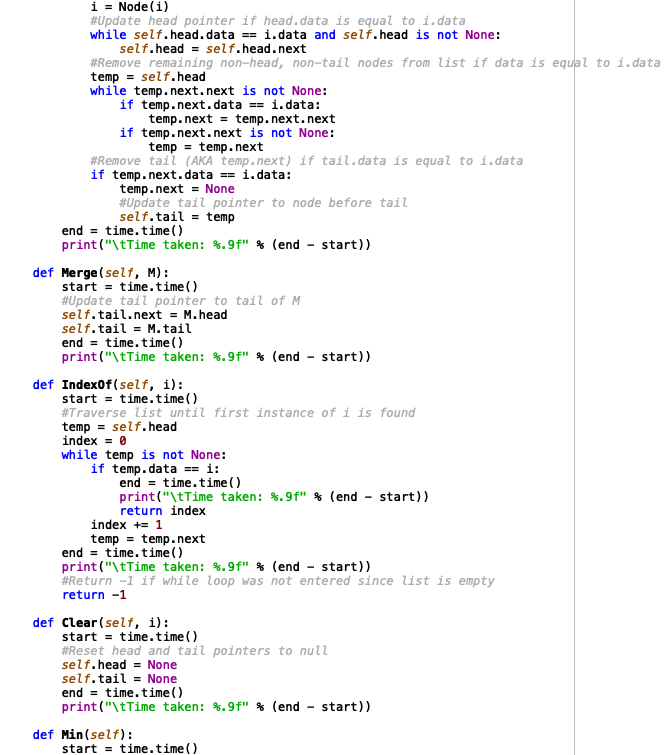
As the results indicate, the *SortedList()* methods *Insert()* and *Merge()* took more time than their *List()* counterparts, again because of the nature of having to maintain the sortedness of the list. Conversely, the *Delete(), Min(), Max(), HasDuplicates(),* and *Select()* methods of the *List()* class took longer (in some cases much longer) than their *SortedList()* counterparts, again because of the lists being unsorted, and thus requiring more time to find specific values or occurrences. The *IndexOf()* method was roughly of the same running time given the similarity of the methods in both classes. The *Clear()* method takes oddly longer for the *SortedList()* class, despite both methods having the exact same coding.

**Conclusion**

From this lab, I learned how to implement methods within a class that handles nodes. Notably, I was able to better appreciate the opportunities and struggles such a class presents when faced with either inputting or extracting data from its list, and how that list being sorted makes inputting a somewhat longer process due to needing to maintain sortedness, but makes extracting information much simpler since patterns in the list can be assumed.

**Appendix**

****

****

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