Lab 3 – Sorted List

Jonatan M. Contreras

University of Texas at El Paso

**Introduction**

In lab three, a SortedList() class was implemented.. This class is a linked list implementation that is always sorted in ascending order. This sortedList() class was to implement 10 different functions, which are: Print(self), Insert(self, i), Delete(self, i), Merge(self, M), IndexOf(self, i), Clear(self, i), Min(self), Max(self), HasDuplicates(self), and Select(self, k). The purpose of each function is briefly discussed per function section in the Proposed Solution Design and Implementation section. We also implemented these functions on a regular List() class and compared running times for each class’s functions.

**Proposed Solution Design and Implementation**

*Part 1: SortedList() Functions:*

Print(self):

The purpose of the Print(self) function is to print the contents of the sorted linked list. I implemented it such that a dummy variable takes the value of the head of the list if the head is not empty and traverses the list. While at each node, the function prints the data of that node. If the head is empty, it prints “empty list.” Since Print(self) loops through the entire linked list, it runs at O(N).

Insert(self, i):

Insert(self, i) inserts an element i into the sorted linked list and it must preserve the ascending order nature of the SortedList() class. If the head is None, the element i becomes the head and tail. Else, the element i is compared to the head. If the element is less than the head, it is swapped with the head and the old head becomes the new head’s next. In the third case where the element is greater than the head, a dummy variable iterates through the list, comparing the next Node’s data to element i, looking for the appropriate place for i to be placed. The iteration stops right before the appropriate insertion location and places that Node in that spot. Since the function iterates through the list to add an element, the worst-case scenario of the function is going to be when an element needs to be inserted right before the tail. This causes the running time to be O(N).

Delete(self, i):

Delete(self, i) finds every instance of the element i in the sorted linked list and removes it. For this function, a dummy variable iterates through the linked list, comparing the dummy variable’s next.data to i. If they are equal, the dummy variable’s next.next becomes the dummy’s variable’s next, removing that element instance of the element i. It ends by checking to see if the head has the same data as i, and if it does, changes the head to the head’s next. In the worst case scenario, Delete(self, i) iterates through the whole list once. This causes it to run in O(N) time.

Merge(self, M):

Merge(self, M) merges a sorted linked list M into the current sorted linked list. Merge(self, M) creates a temporary SortedList() variable that is empty. Two dummy variables, one for each sorted list’s head, then iterate through both lists at the same time. While doing this, the function compares the data from each current Node and places the smallest element into the temporary SortedList() by making it the tail’s next. It then moves the iterator of the list that had its element inserted into the temporary list forward. If the lists are not of equal length, one list will be iterated through completely before the other, as well as when one list consistently has values that are less than the values in the other list. So after this loop, there is another loop that finishes iterating through the other list, inserting its elements as the next of the tail. The reason this is possible is because as we iterate through the two lists, it is in ascending order. This means that the current node we are on will always be larger than the tail of the temporary list. The function finishes by making the current linked list’s head and tail that of the temporary linked list.

The function iterates through both current and M list, placing the elements as the temporary sorted list’s tail’s next. Thus the running time is O(M) or O(N) depending on which list is bigger.

IndexOf(self, i):

IndexOf(self, i) finds the index of the first instance of the element i in the sorted linked list. IndexOf(self, i) assigns a dummy to the head of the list and iterates through the list, comparing the current Node’s data to i. A counter keeps track of the index. If there is a match, it exits the loop and returns the counter. In the worst-case scenario, the function needs to loop through the entire list. Thus, it runs in O(N) time.

Clear(self, i):

Clear(self, i) makes the current list an empty list. Clear(self, i) sets the head and tail to None. This runs in constant time, or O(1).

Min(self):

Min(self) returns the smallest element in the list. Since the list is sorted, the smallest element is the head of the list. Thus, Min(self) returns the head of the list. This is in constant time, or O(1).

Max(self):

Max(self) returns the largest element in the list. Since the list is sorted, the largest element is the tail of the list. Thus, Max(self) returns the tail of the list. This is in constant time, or O(1).

HasDuplicates(self):

This function returns True if there are multiple Nodes with the same data or False otherwise. HasDuplicates(self) assigns a dummy variable with the current list’s head and iterates through the list, comparing the dummy variable’s next element’s data to the dummy variable’s data. If they are equal, the function returns true. Else, the dummy variable is assigned with the next Node. If the iteration goes through the whole list without returning True, it returns False.

In order to complete this process, the list needs to be completely iterated through. This means that the function runs in O(N) time.

Select(self, k):

Select(self, k) returns the kth smallest element in the sorted list. Since the list is already sorted, Select(self, k) only needs to iterate through the list up until the kth element. Thus, a dummy variable is assigned to the head of the list and iterates through the list. Using a counter, the function keeps track of the current index and compares it to k. Once counter is equal to k – 1 (due to indices beginning at 0), the function returns the data at that current Node. Since the function, in the worst case scenario, needs to iterate through the whole list, the running time is O(N).

*Part 2: List() Functions:*

Print(self):

The Print(self) function is the same as the SortedList() function., It runs at O(N).

Insert(self, i):

The Insert(self, i) function simply appends an element to the end of the List() by making the element i the current List’s() tail’s next. This is done in constant time so the running time is O(1).

Delete(self, i):

The Delete(self, i) function is the same as the SortedList’s() Delete(self, i) function. The running time is O(N).

Merge(self, M):

Since Merge(self, M) does not need maintain any order in the List() that is merging M, Merge(self, M) uses the Insert(self, i) function to simply append every element of M to the end of the current list. The running time is O(M), m being the size of M.

IndexOf(self, i):

The IndexOf(self, i) function for List() class is the same as the IndexOf(self, i) function for the SortedList() class. The running time is O(N)

Clear(self):

The Clear(self) function simply sets the head and tail to none. The running time is O(1).

Min(self):

The Min(self) function declares the first node of the List() as the min. Min(self) then uses this newly declared min to compare to the rest of the list. If it encounters a new min, it swaps the previous min for the new min. Since it iterates through the entire list in order to find the min of the list, the running time is O(N).

Max(self):

The Max(self) function declares the first node of the List() as the max. Max(self) then iterates through the list, using this max to compare to the rest of the values. If it finds a new value that is greater than the current max, it swaps that new max out with the old max. Since the function needs to iterate through the whole list to find the max, Max(self) runs at O(N).

HasDuplicates(self):

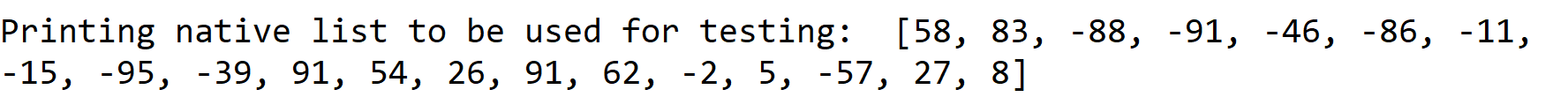
Since the list is not sorted, duplicates will not necessarily be next to each other as is the case with the SortedList(). Thus, every element has to be compared to the rest of the elements, looking for possible duplicates. Thus, HasDuplicates(self) takes the first value of the list and compares it to every remaining value in the last. If a match occurs, it returns True. If not, it moves on to the second element in the list and compares it to the remaining elements in the list. Since the function is comparing the n elements in the List() n times, the running time is O(N2)

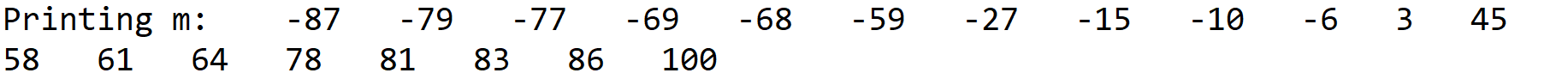
Select(self, k):

Since List() is not sorted, the list must first be sorted in order to find the kth smallest element in the list. Thus, Select(self, k) for List() creates a temporary SortedList() and inserts the current List() into the SortedList(). Once this SortedList() is created, the Select(self, k) iterates through the list until it finds the kth element. Since the SortedList’s() Insert function has a runtime of O(N) and we’re inserting N elements, this part of the function runs at N2 time. Finding the kth element runs at N, making the running time of this function O(N3).

**Experimental Results**

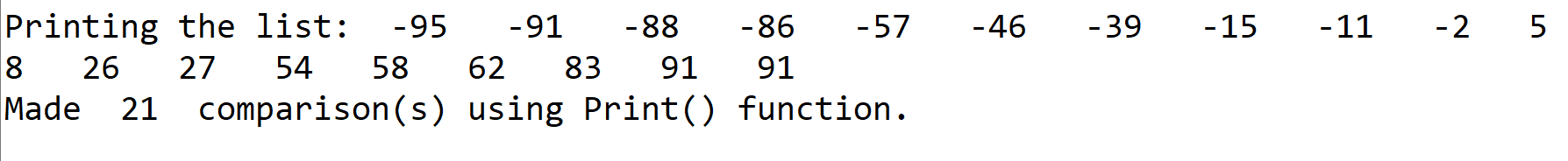
*Part 1.1: Function Output for SortedList()*



This list will be used for all functions in experiments, for both UnsortedList() and List().

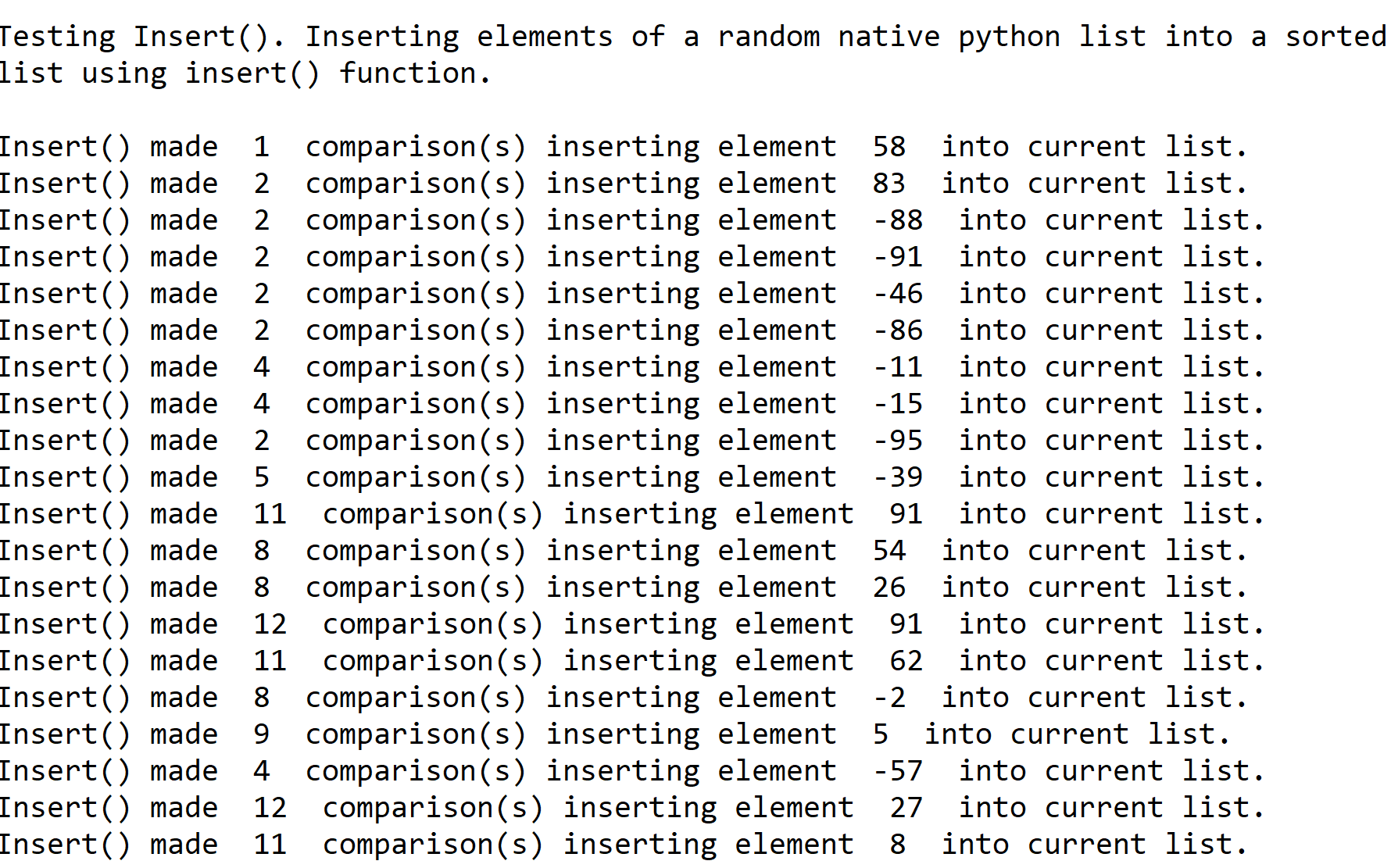
This will be the list used as m for the Merge() function.

Print(self):



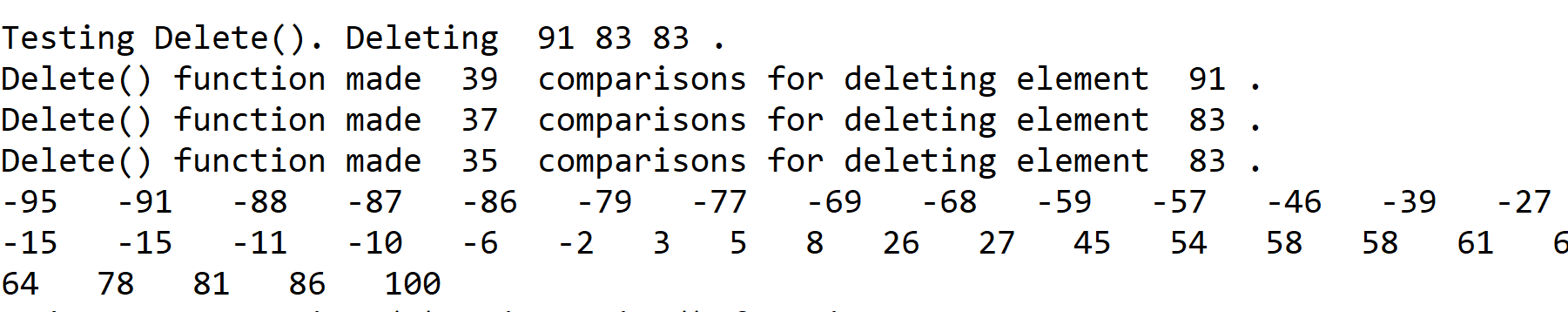
Print makes an additional comparison when checking if the head is None.

Insert(self, i):



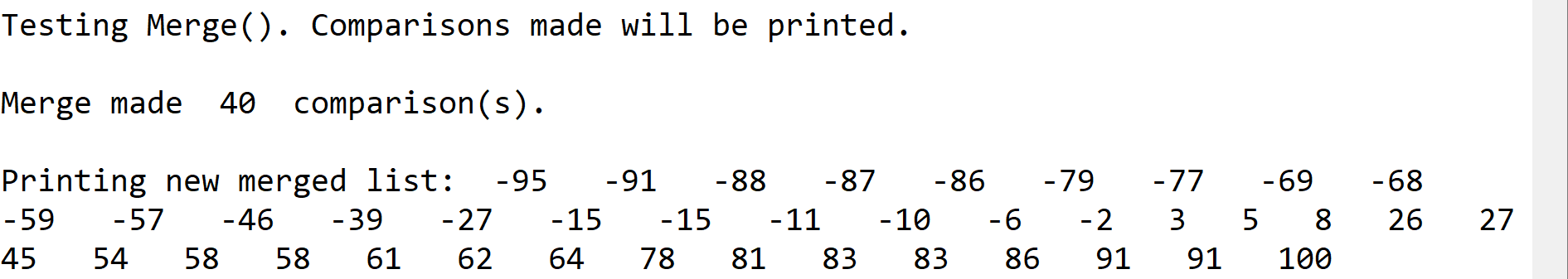
As the element gets larger, it makes more comparisons because it traverses the list further to find the appropriate place of insertion.

Delete(self, i):

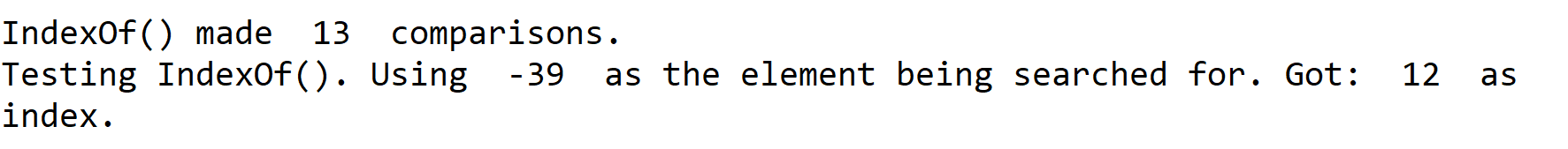


Delete makes n-1 comparisons every time it deletes another element. (Note: Delete() was tested after forthcoming functions so the elements deleted here are present in the coming tests.)

Merge(self, M):

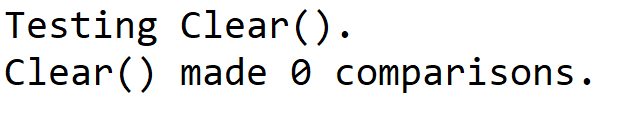


IndexOf(self, i):



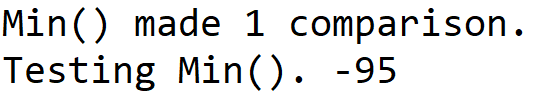
The list that was used for IndexOf can be seen in the Merge() output.

Clear(self, i):



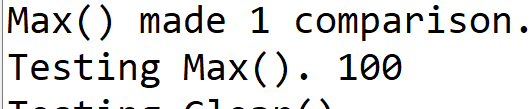
Clear() makes no comparisons. It only sets tail and head to None.

Min(self):



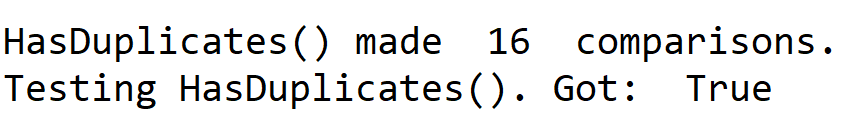
Min() only checks to see if the head is not None.

Max(self):



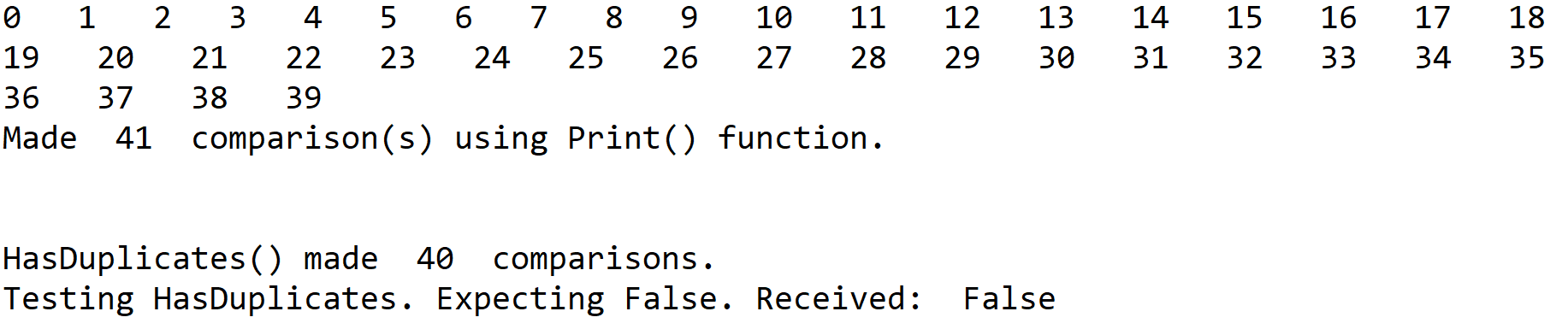
Max() only checks to see if the head is not None.

HasDuplicates(self):

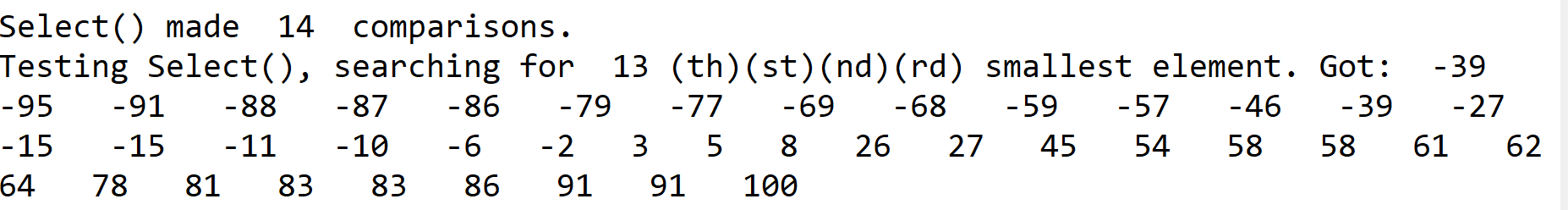


HasDuplicates() only got 16 elements in before finding a duplicate.

I made a temporary list with no duplicates to test HasDuplicates() on this list. The list consists of the elements 0-39. This was the output:



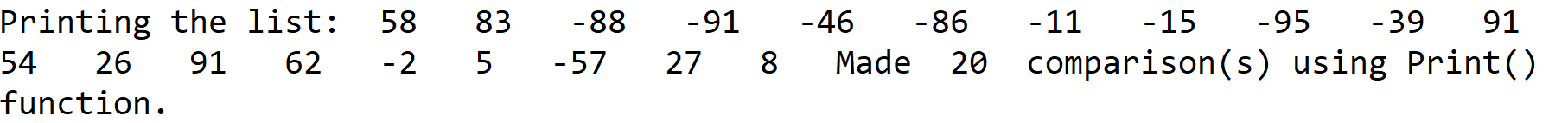
Select(self, k):



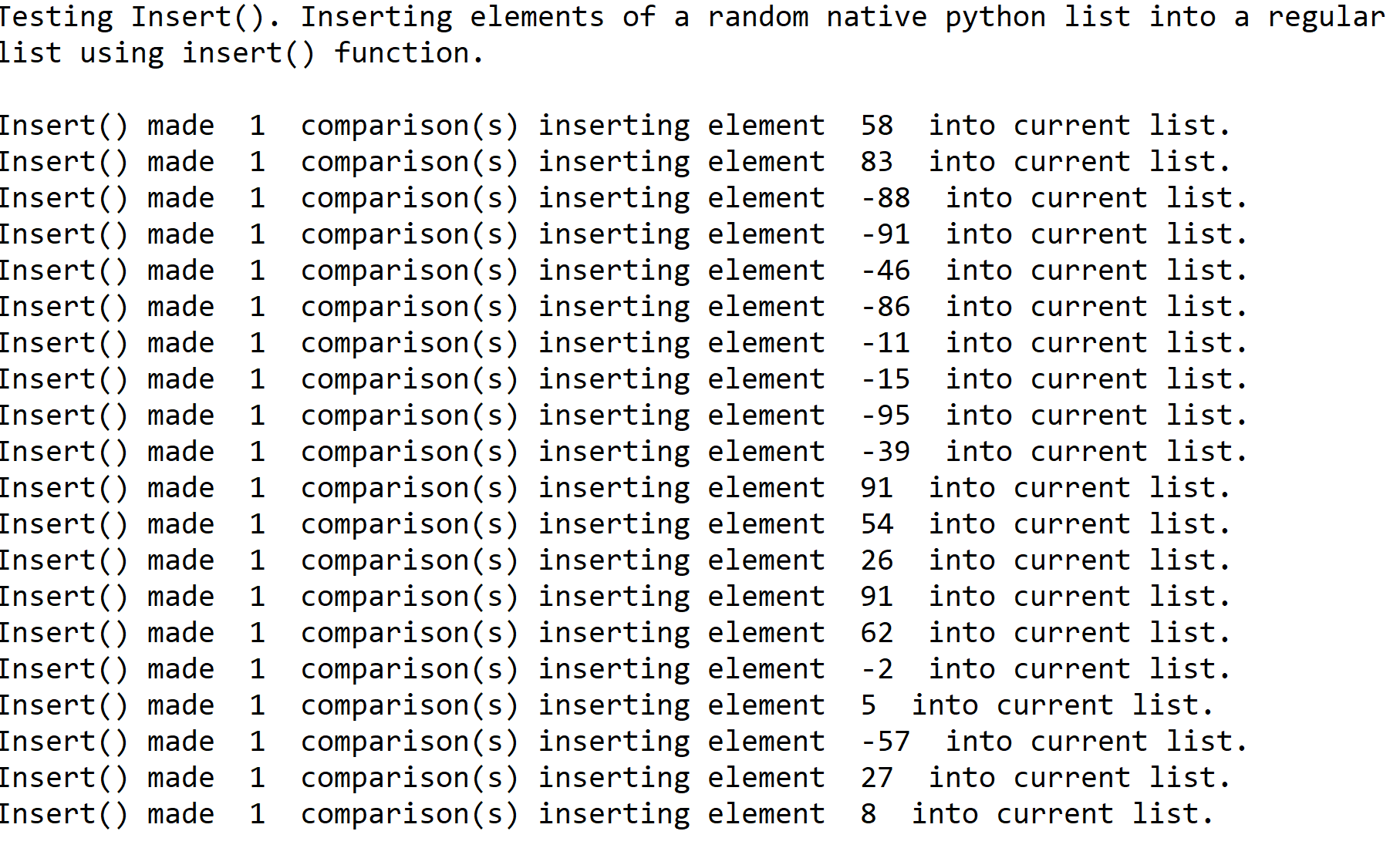
Select made 13 comparisons to find the 13th smallest element plus 1 for checking if the head is not None.

*Part 1.2: Functions Output for List()*

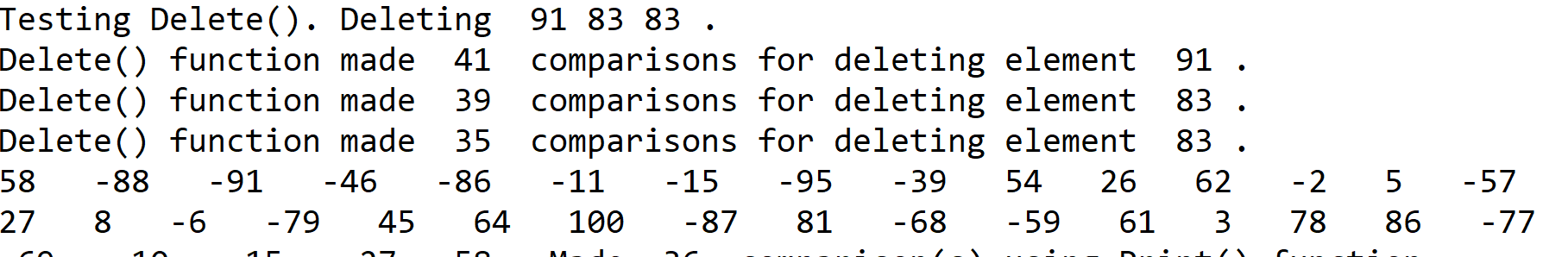
Print(self):



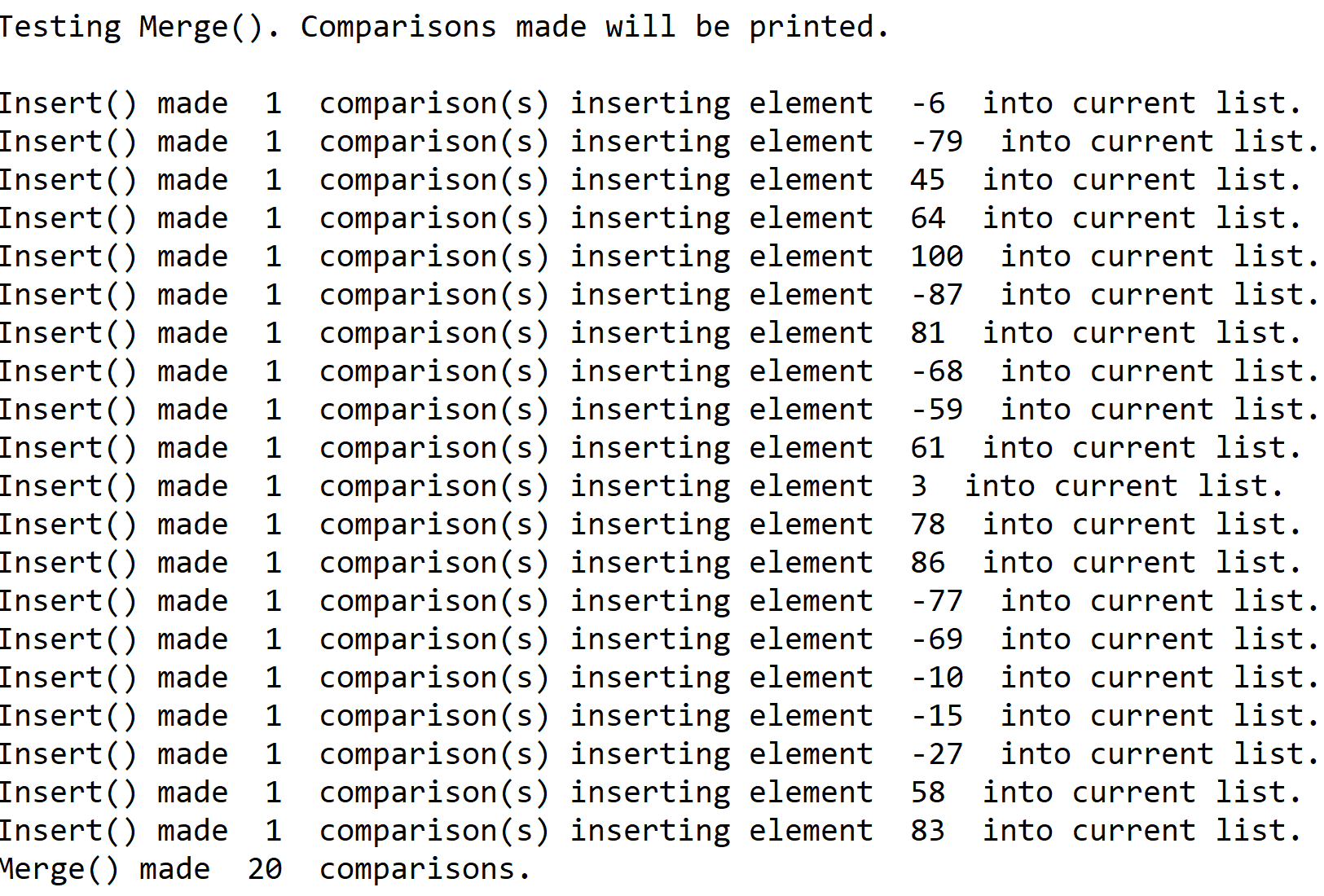
Insert(self, i):



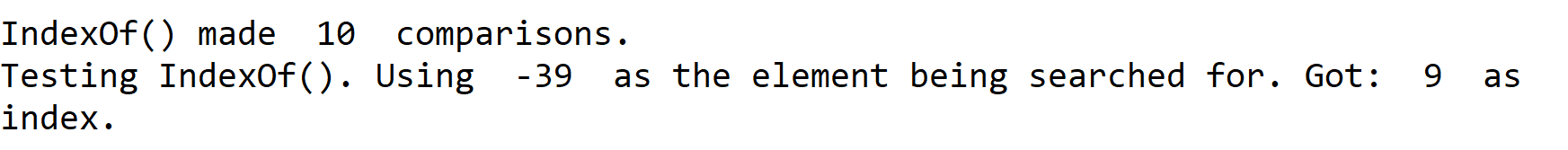
Delete(self, i):



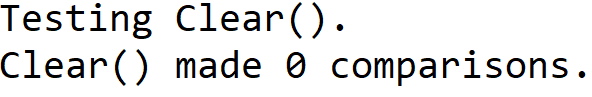
Merge(self, M):



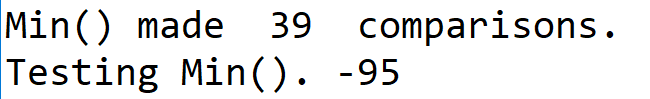
IndexOf(self, i):



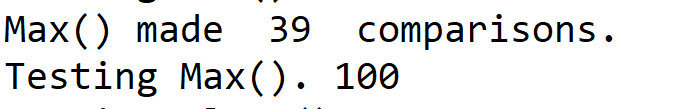
Clear(self, i):



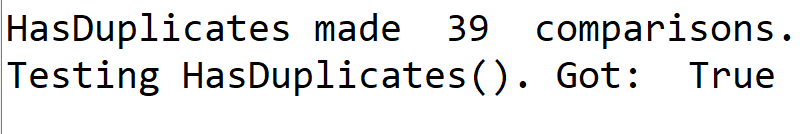
Min(self):

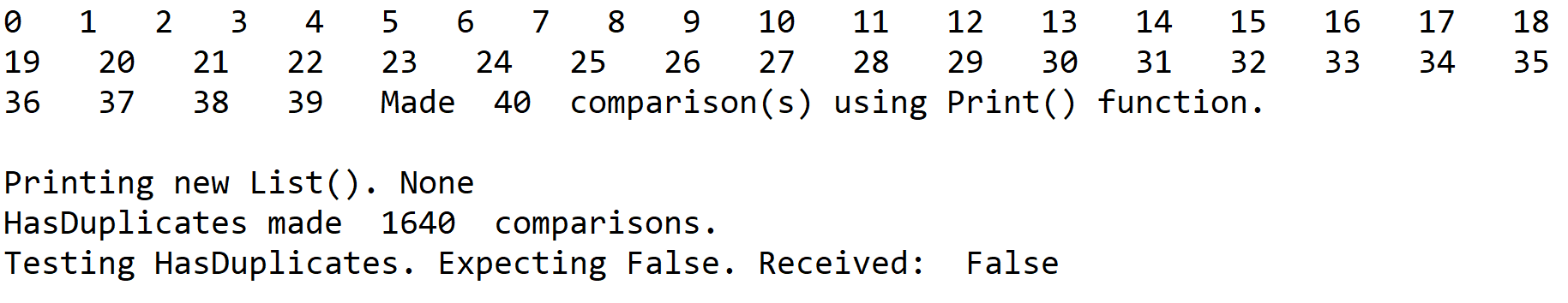


Max(self):

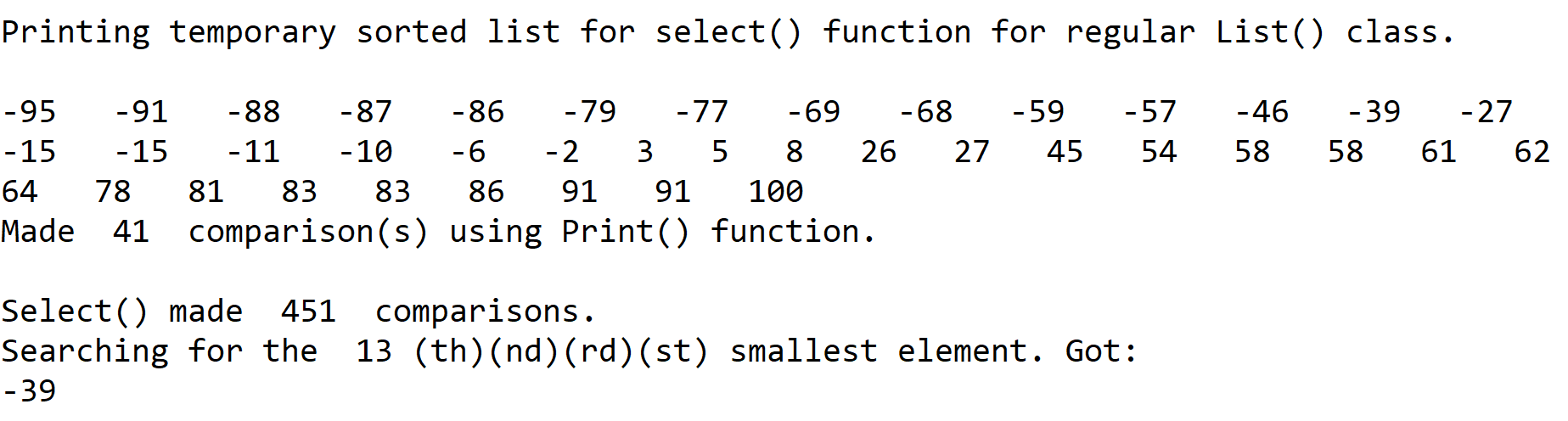


HasDuplicates(self):





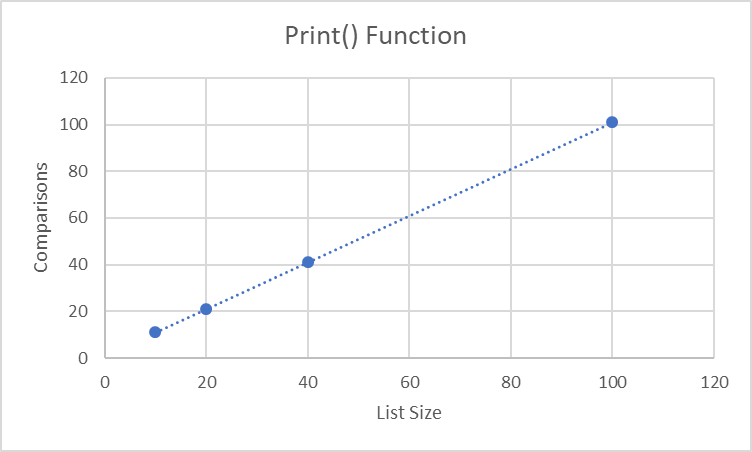
Select(self, k):

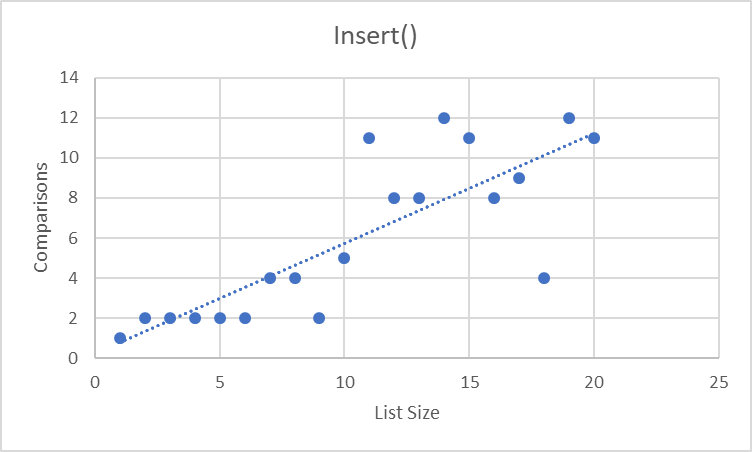


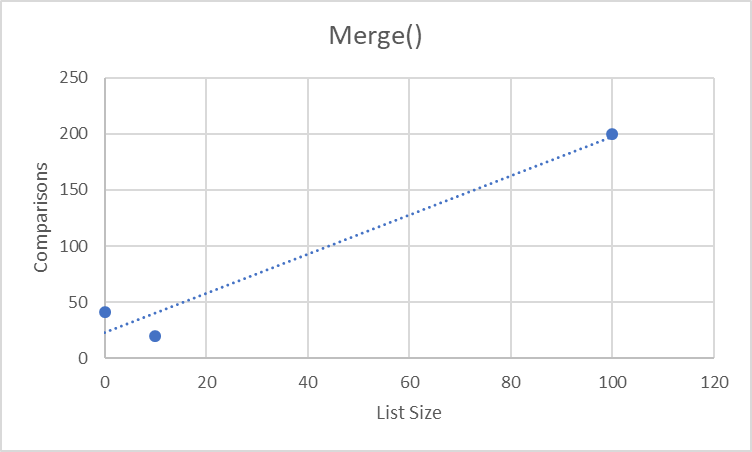
*Part 2: Running Times Table*

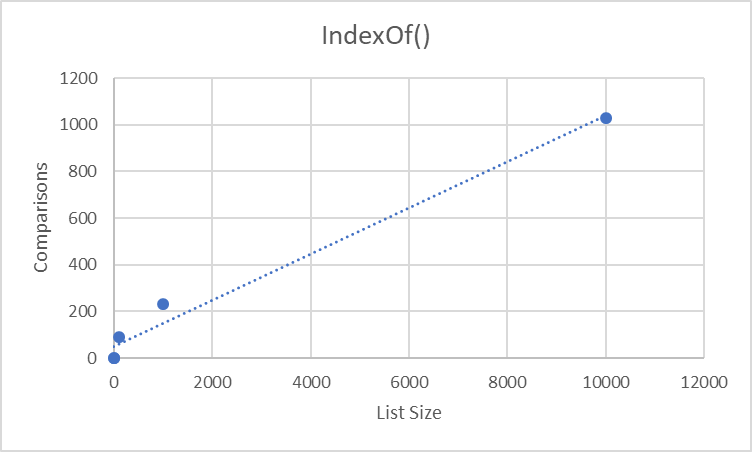
|  |  |  |
| --- | --- | --- |
| Function | SortedList | List |
| Print() | O(N) | O(N) |
| Insert(i) | O(N) | O(1) |
| Delete(i) | O(N) | O(N) |
| Merge(M) | max(O(M), O(N)) | max(O(M), O(N)) |
| IndexOf(i) | O(N) | O(N) |
| Clear(i) | O(1) | O(1) |
| Min() | O(1) | O(N) |
| Max() | O(1) | O(N) |
| HasDuplicates() | O(N) | O(N2) |
| Select(k) | O(N) | O(N3) |

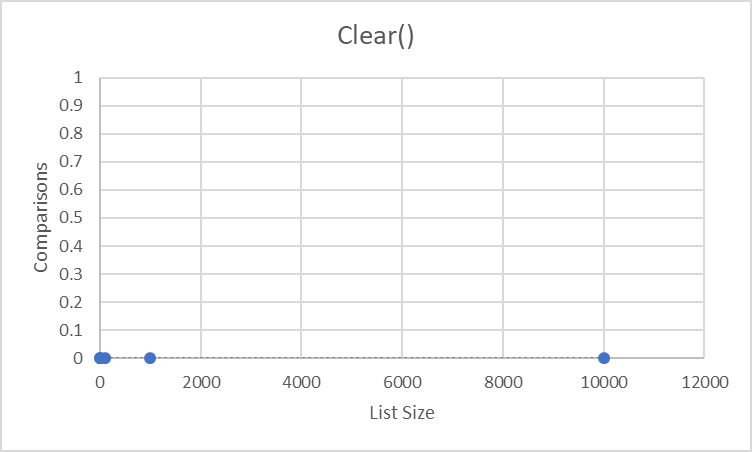
*Part 3.1: Graphs of SortedList() Functions*

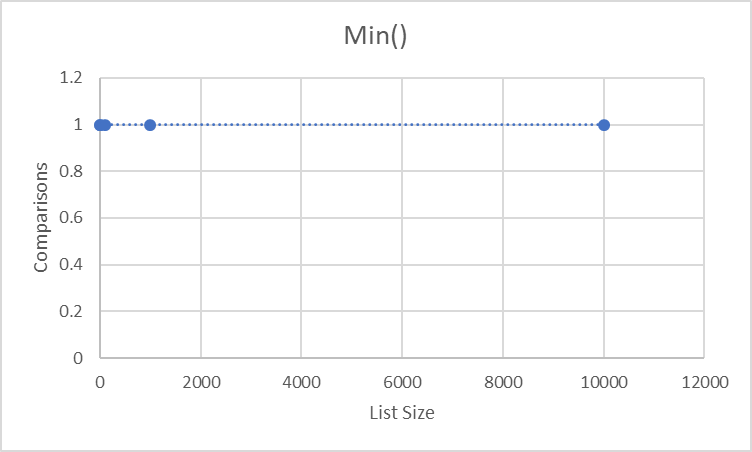
**

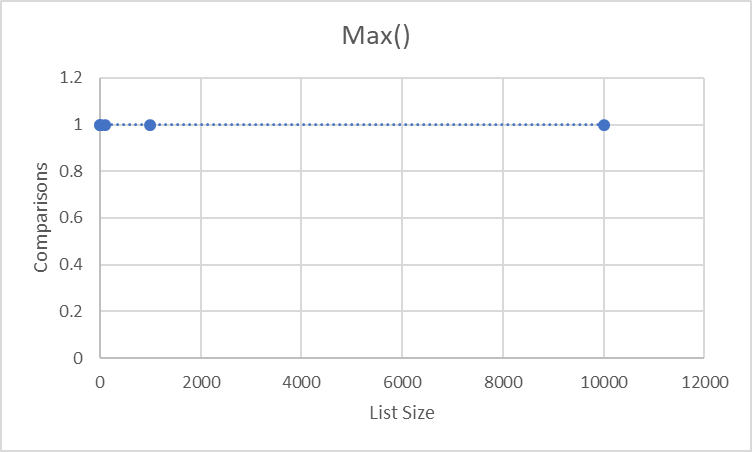


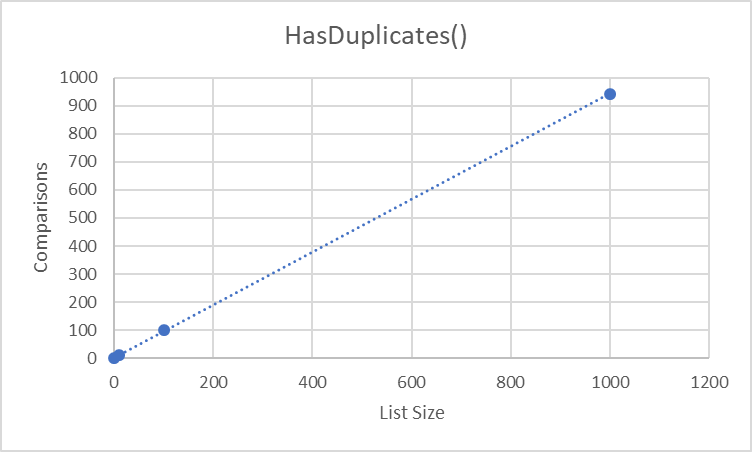


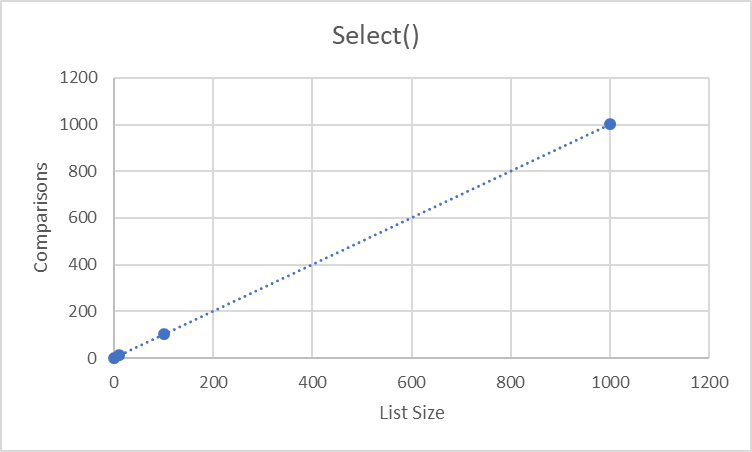




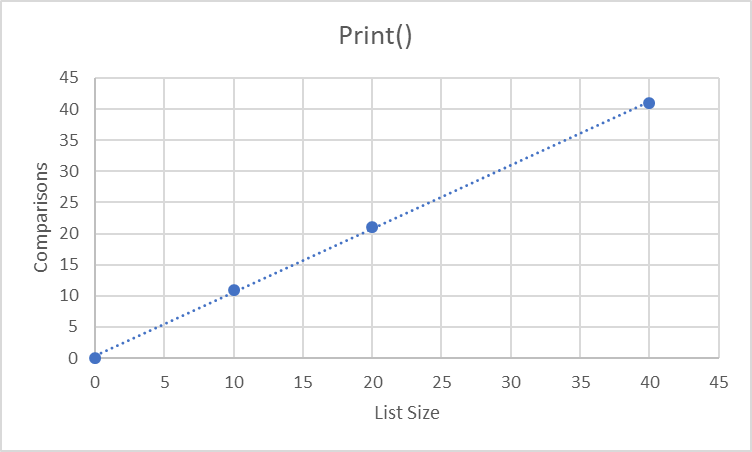


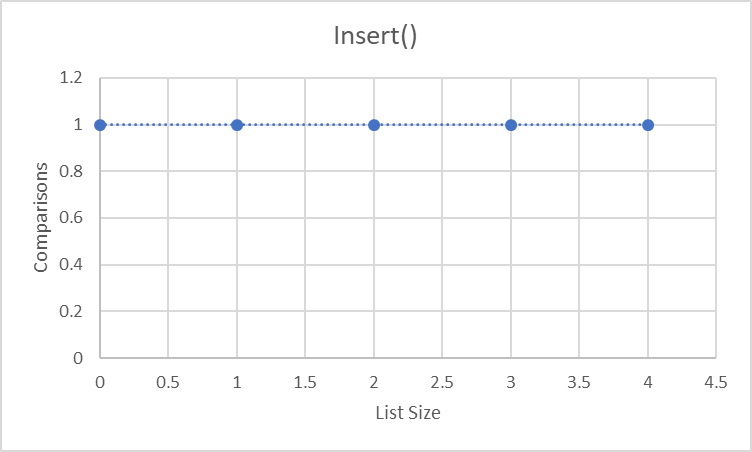


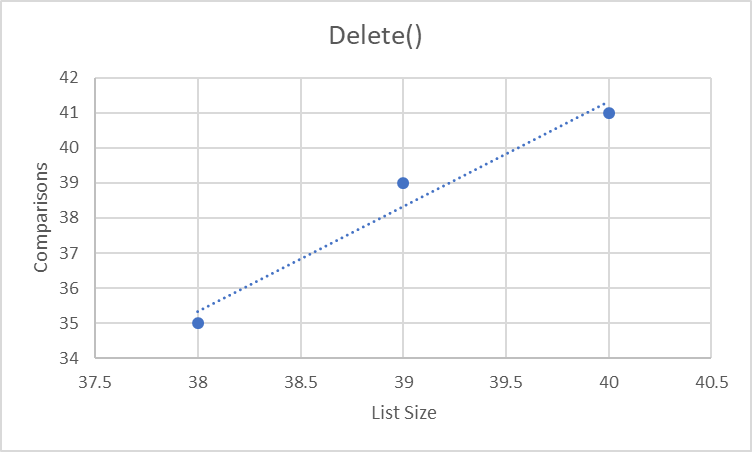


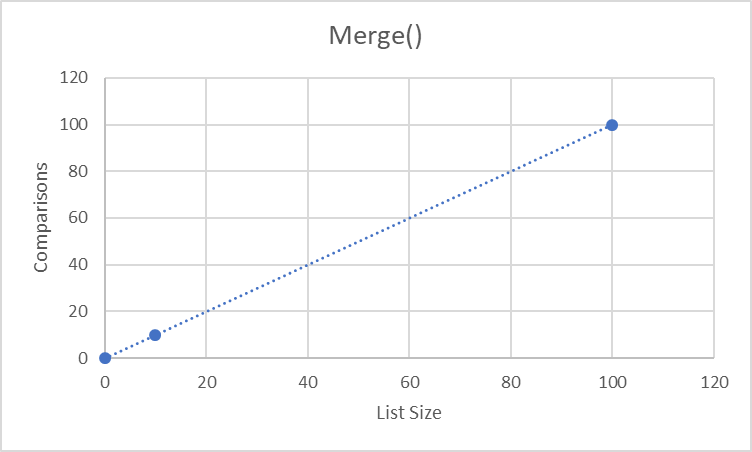


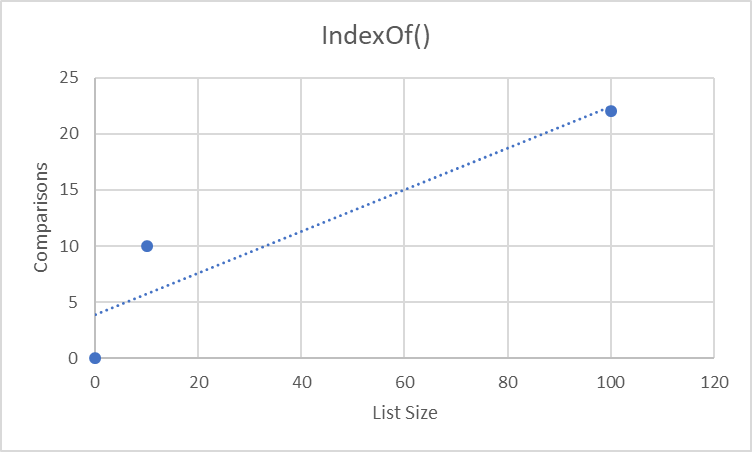
*Part 3.2: Graphs of List() Functions*

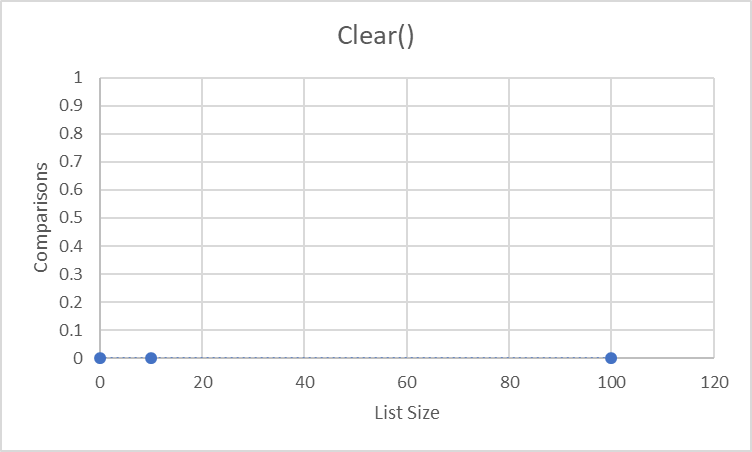
**

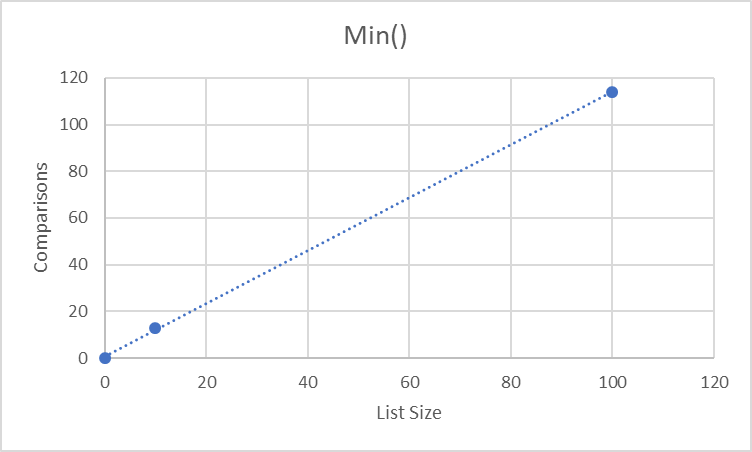


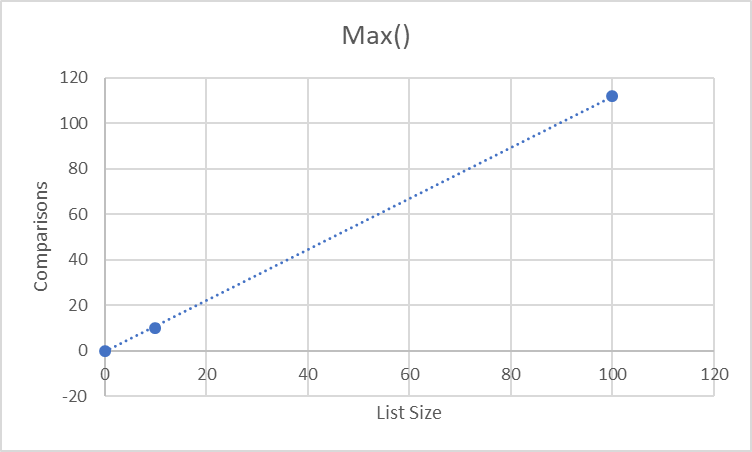


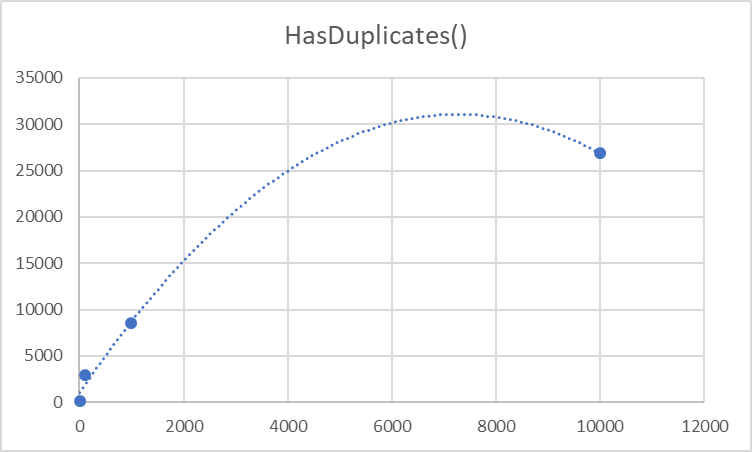


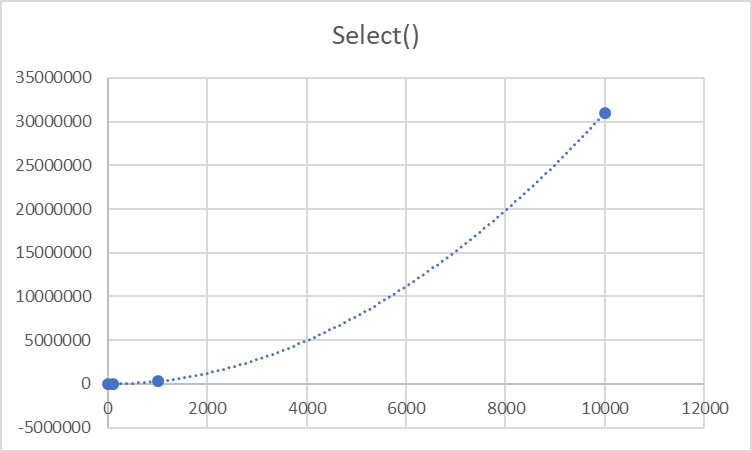








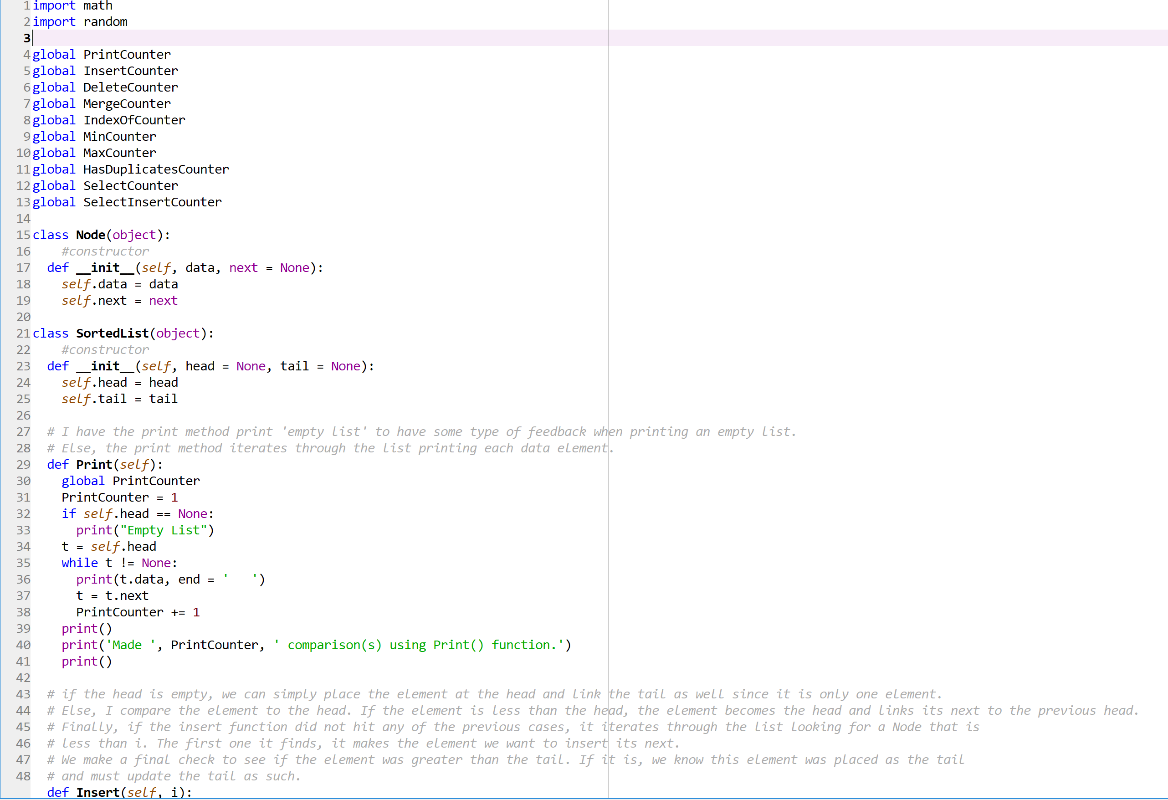


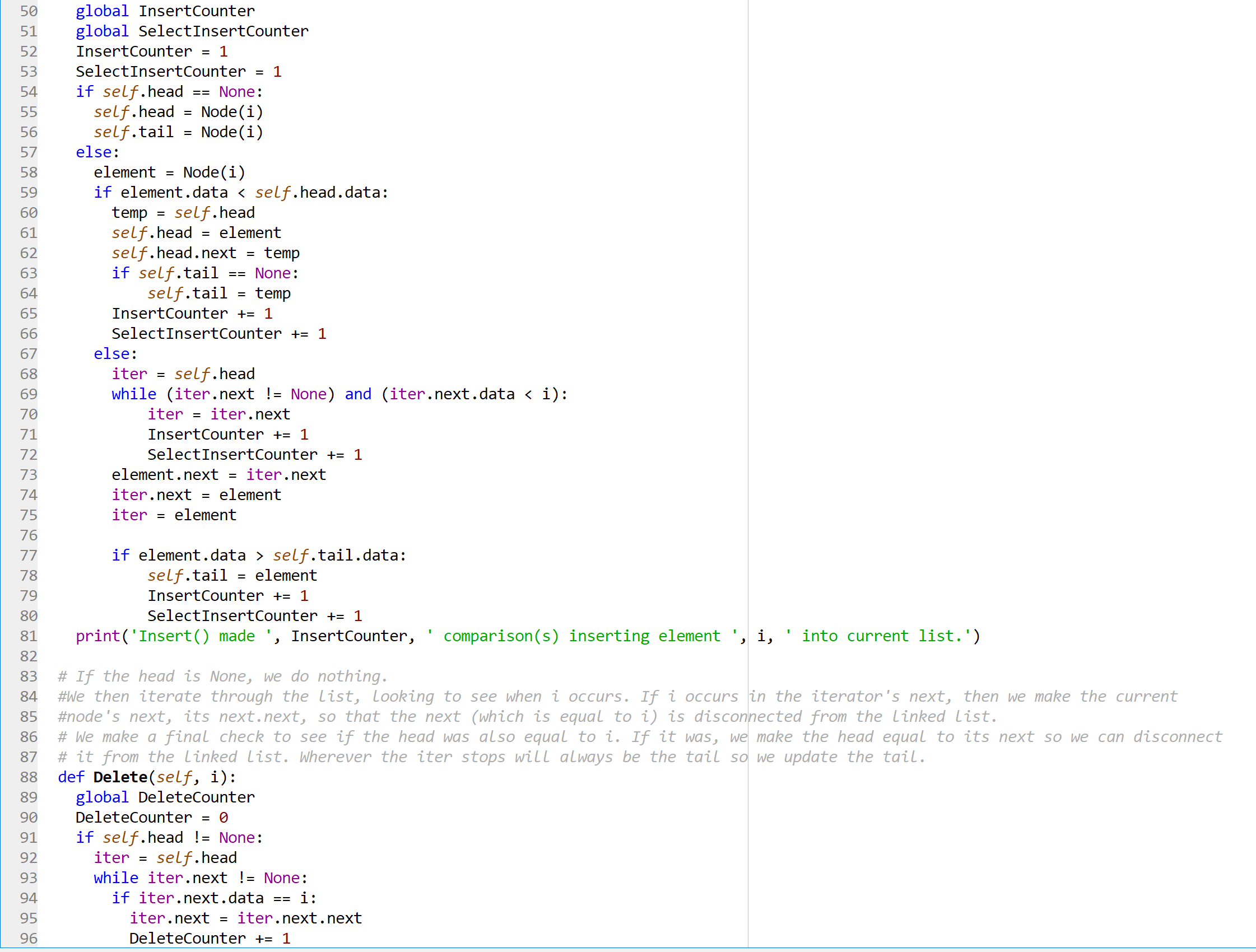


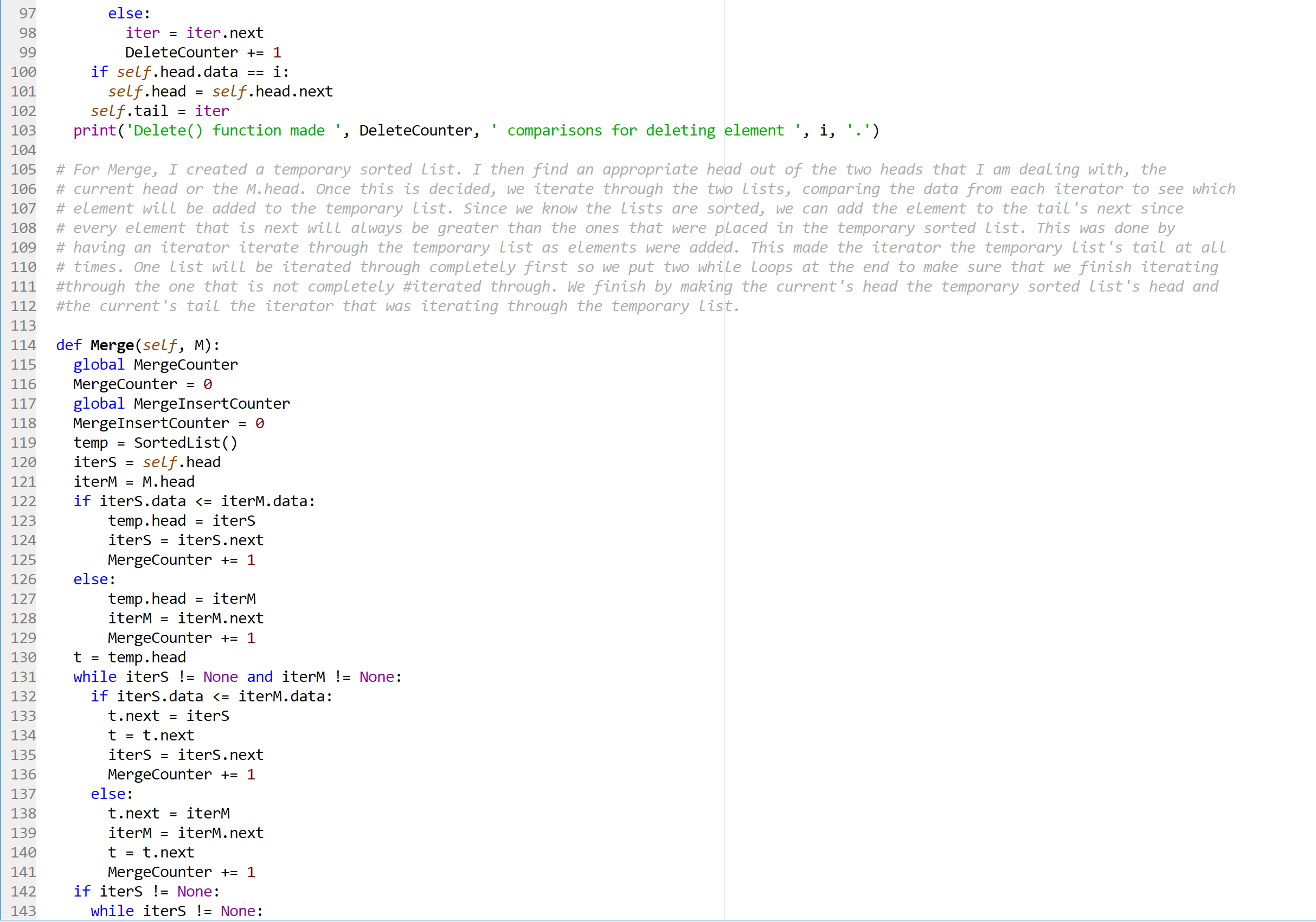
**Conclusion**

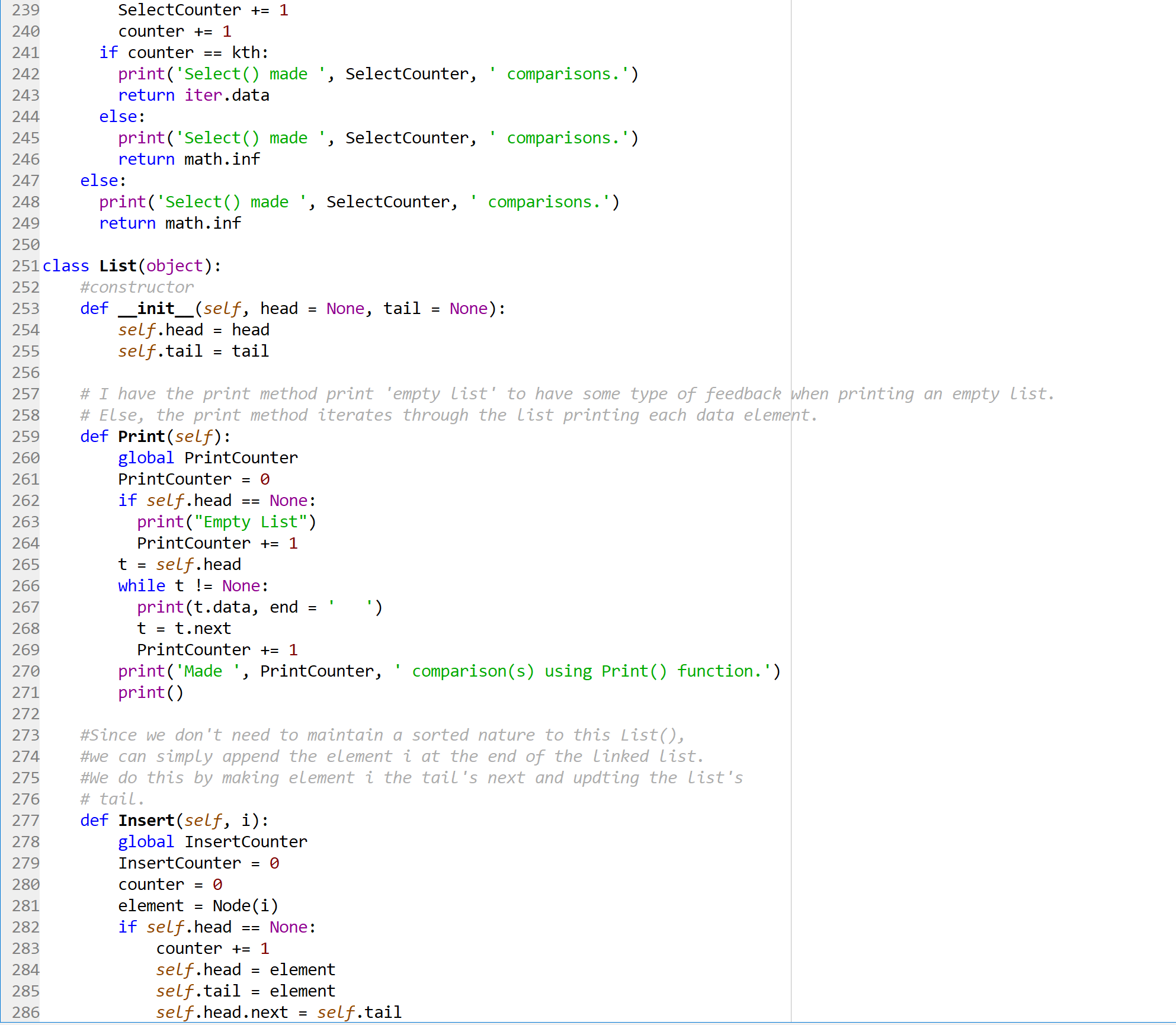
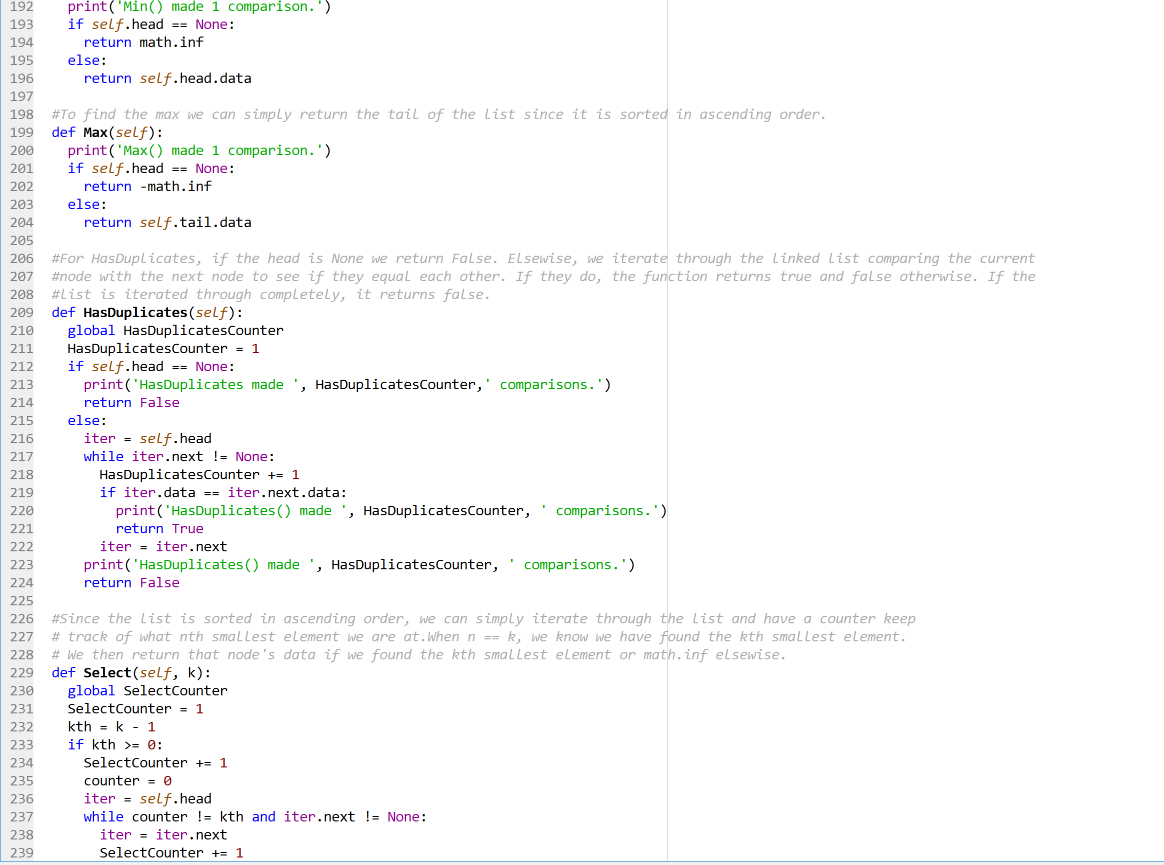
I learned that the running time of a function will depend on its implementation in its general class. For some functions, the SortedList() version was faster than the List() version, but for others, the inverse was true. This means that it is important to understand these nuances and its implications for your specific applications before deciding on a method of implementation. This will allow your applications that use the functions to have the best chance of success.

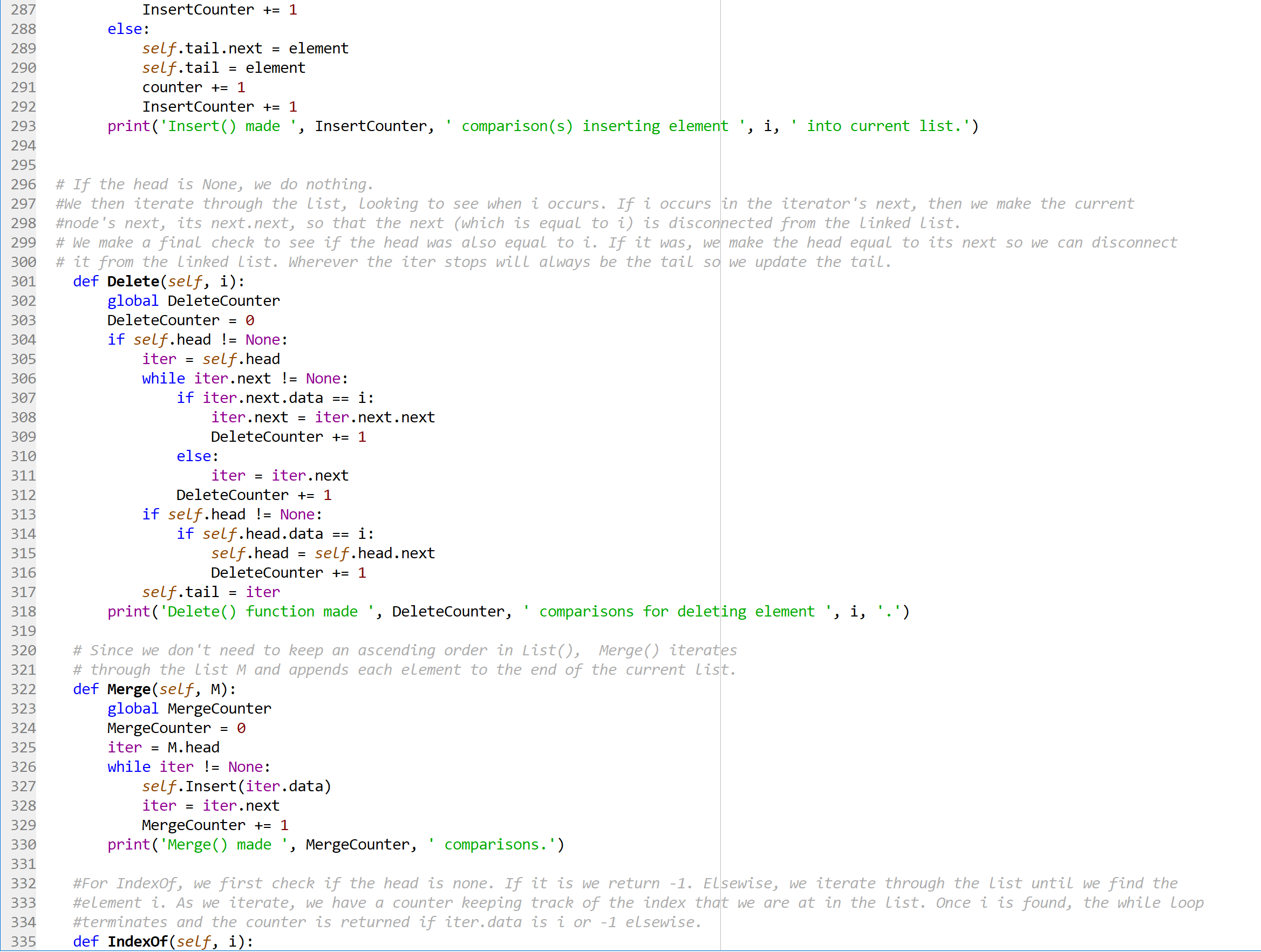
**Appendix – Source Code**

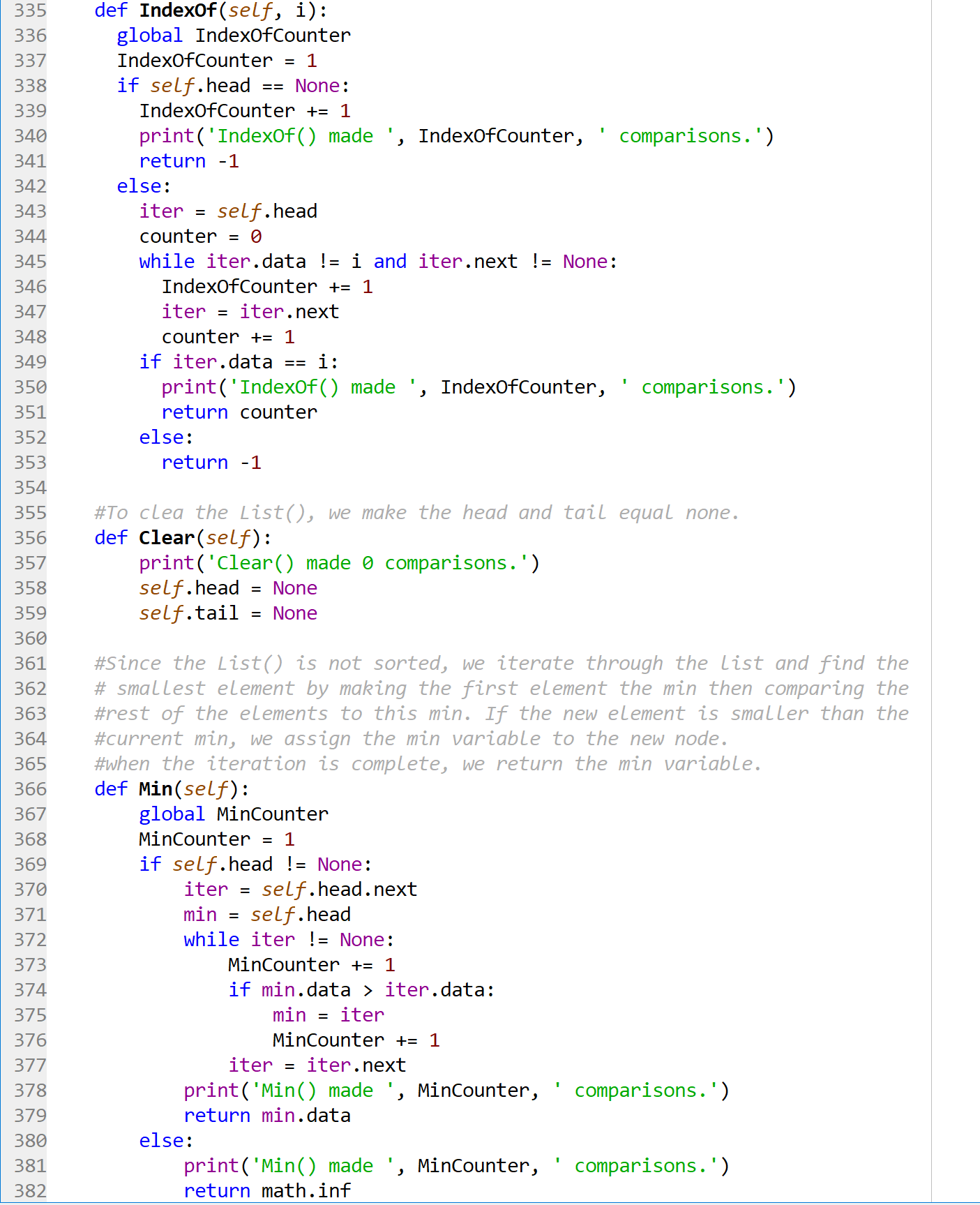
****

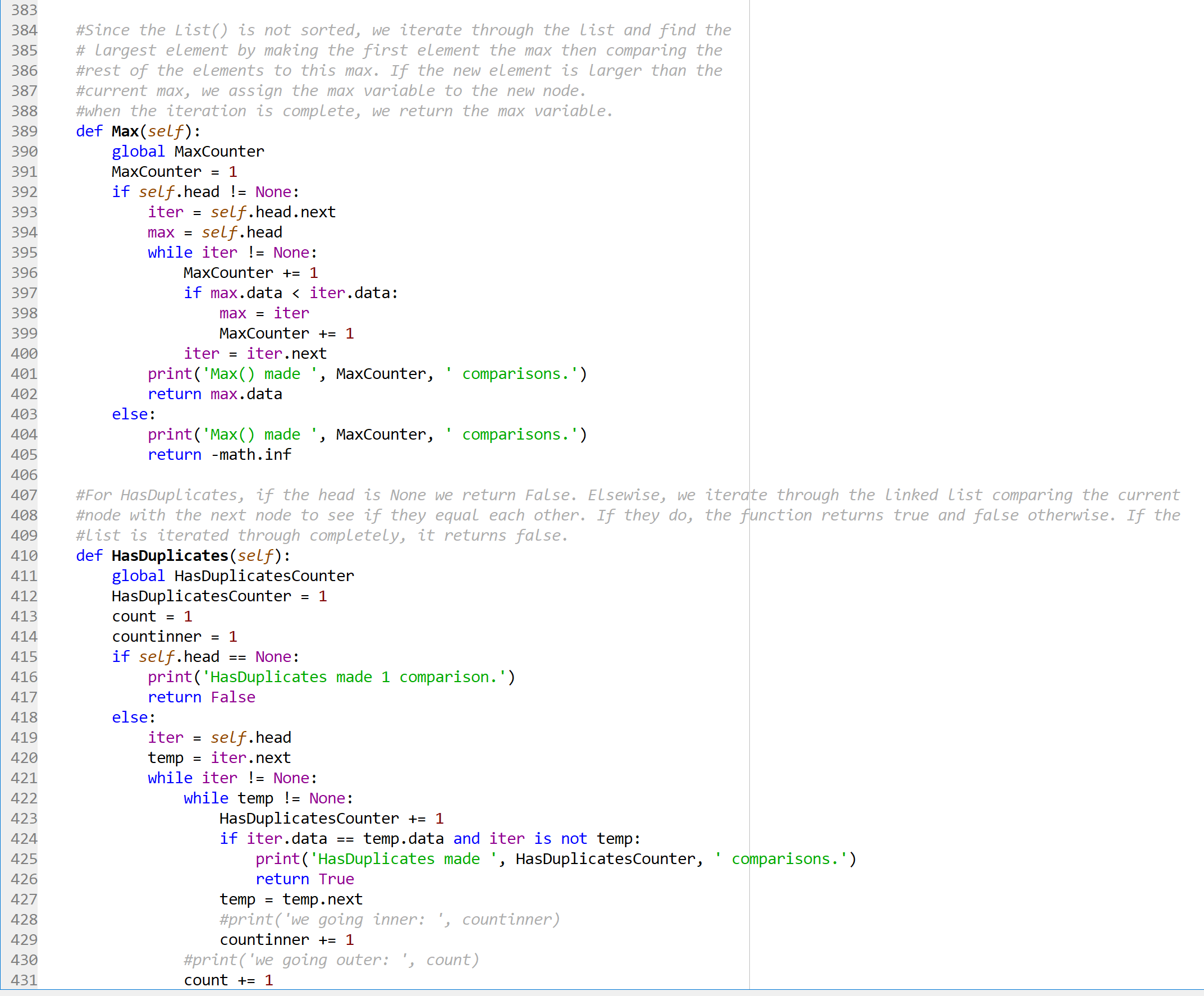
****

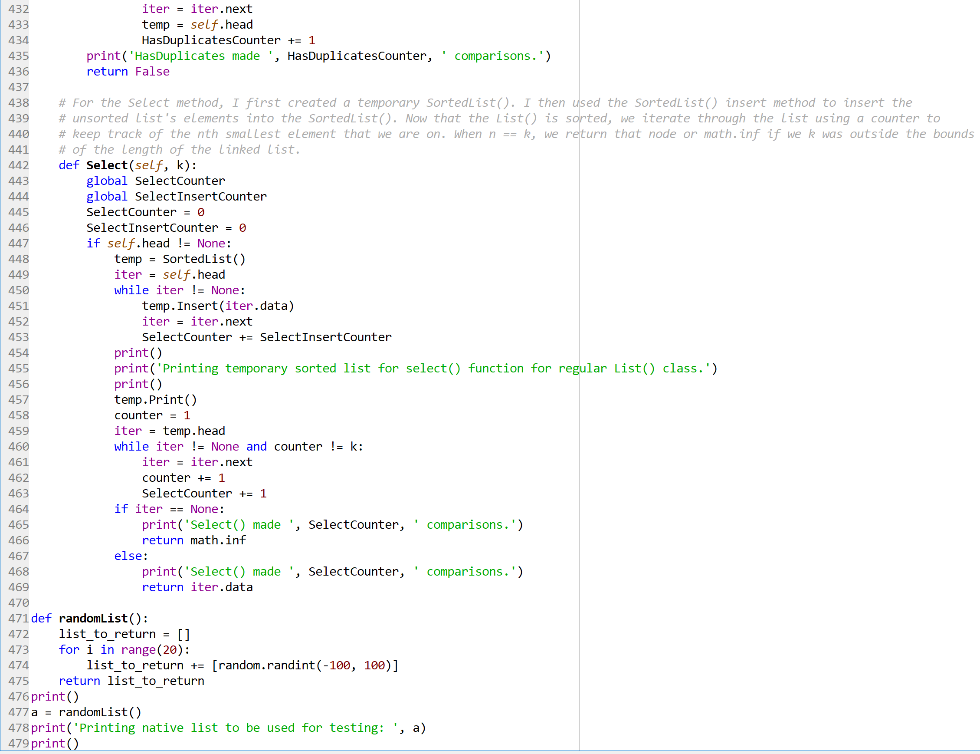
****

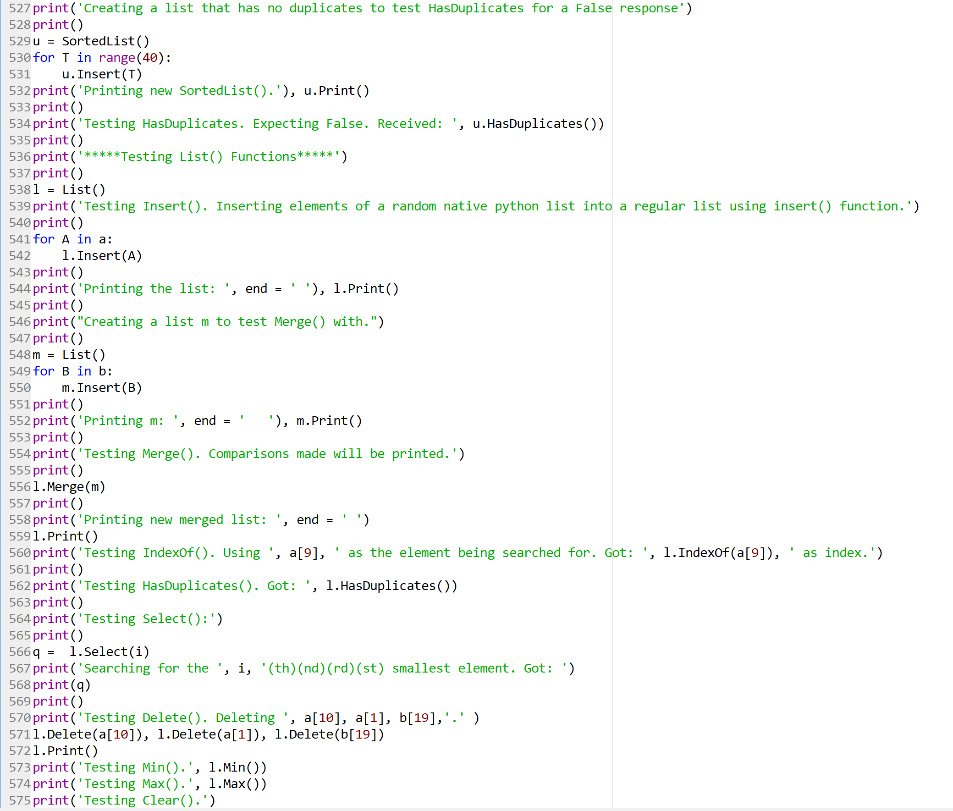
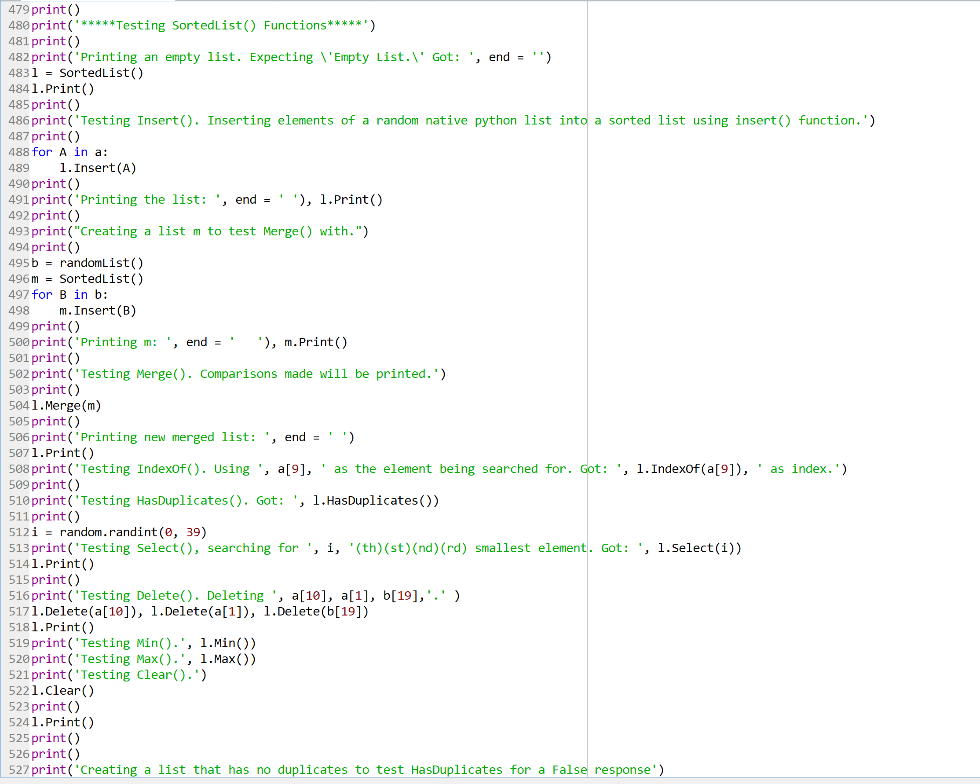
****

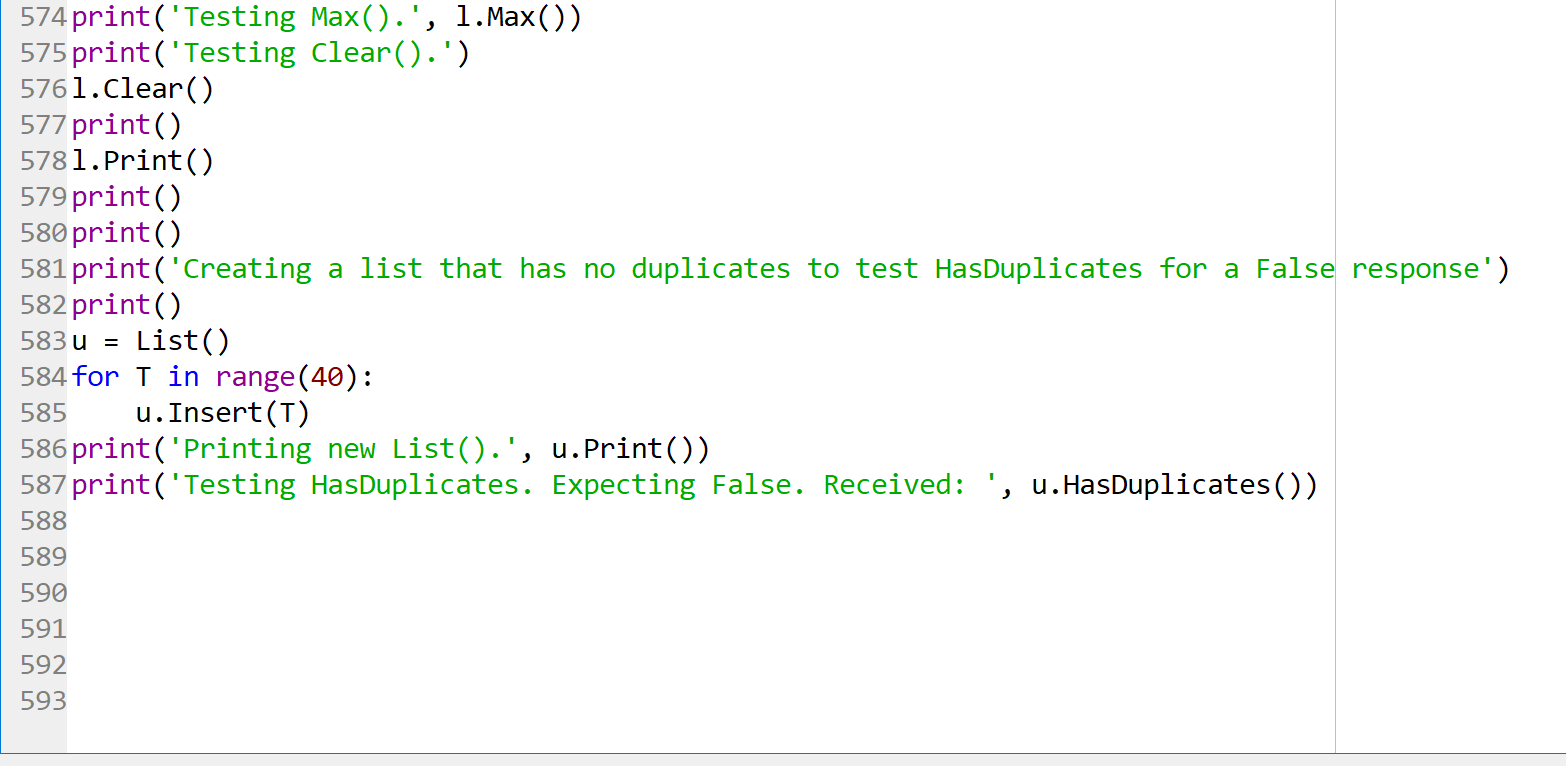
****

****

****

****

****

****

**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.

Jonatan Contreras