**LAB 3**

**DANIEL CRUZ**

**Introduction**

The purpose of this lab is to provide practice with working with linked list. Specifically, this lab focuses on creating a specialized sorting linked list to demonstrate how to write certain algorithms when working with linked lists. In an extension to this, this lab also involves determining and analyzing the run-time complexity of generic linked lists and this specialized sorting linked list.

**Proposed Solution Design and Implementation**

In each of the assigned class functions, I focused on writing algorithms that limits the number of possible times the full list may have to be traversed. As such, I tried to limit most functions on traversing the list once. Since the list had to be sorted from least to greatest, I insured any insertion functions maintained this rule. As long as all sorting was done in insertion, all other functions could take advantage of using a sorted linked list. In the following I will describe the algorithm for each function:

Print: Starting from the head, a traversing pointer will move until it reaches the end of the list, printing the data of every node.

Insert: Insert is in charge of ensuring the list is sorted. If the list is empty, i, the value to be inserted, is set as both the head and the tail to initiate the list. If the head is less than i, i is the smallest element and thus is the new head. Likewise, if i is greater than tail, then i is the greatest element and will become the new tail. Otherwise, the list is traversed until the next node is greater than i. i will be inserted right before this node.

Delete: i is the value to be deleted. Nothing is done if the list is empty, i is less than the head, or i is greater than tail. If i is the head check if the head is the only value in the clear. If it is, set both head and tail to none, other delete the head only. Otherwise, traverse list. If the value is the tail, set the tail as the node before the tail. Otherwise link the node previous to the value to the node after the value.

Merge: For all nodes in M, another sorted list, call Insert() to merge with this list.

IndexOf: i is value to be searched. Return -1 if list is empty, i is less than head, or i is greater than tail. Otherwise traverse list and find the first node that contains i. return a counter that counts how many nodes that have been searched, this is the index of that value.

Clear: set head and tail to None. This clears all nodes in the list.

Min: Returns math.inf is the list is empty. Otherwise returns head, this will always be the smallest value.

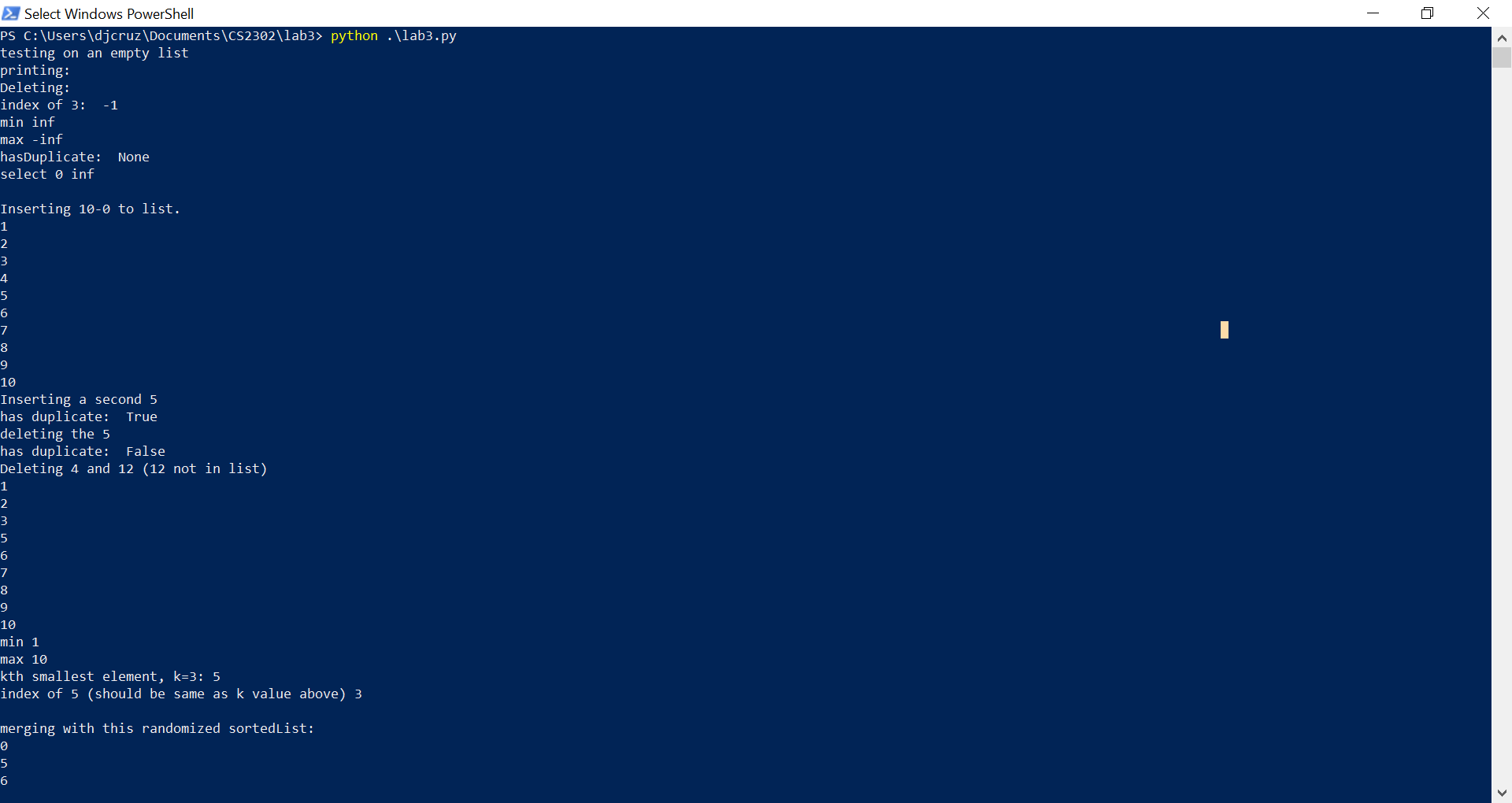
Max: Returns -math.inf is the list is empty. Otherwise returns tail, this will always be the largest value.

HasDuplicated: Returns False if the list is empty. Checks if two node pairs are the same value. This is the only way duplicates can exist in the list. If it did not find any pairs will traversing the whole list, returns False.

Select: If the list is empty, returns math.inf. Otherwise traverses up to k nodes. If the list is shorter than k nodes, return math.inf, otherwise returns the data at the kth node.

**Experimental Results**

The first result is the output of the testbench. In this testbench each function was tested with both empty and nonempty lists.





Each function was tested with an empty list to test this edge case. After clearing the list, min and max was tested to check if the head and tail was set to none. As show, each function works as expected and knows how to react to an empty list.

Because this linked list is specialized to always have a sorted list, some functions have longer runtime complexities. The following is a comparison of runtimes of an optimized linked list and the sortedList:

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

Interesting enough, the sorted list has slightly more constant runtime functions. Due to the sorted nature of this list, min and max run at constant time in expense to an insert function of O(n). The results in a net gain of one more O(1) of a normal list however.   
I have attempted to prove these runtimes are accurate through experimental results, however the data values received is not necessarily applicable due to the speed of these functions being relatively quick even for large values of n. Here is the data graphed and the raw values are included in the lab3 folder as a csv file. Merge and print do show the expected n increase, however the other functions of O(n) seemed to have run too quickly to be captured by the test function, even at high n.

**Conclusion**

This lab proved a way to practice working with linked lists. Specifically seeing the runtime complexity of certain functions accentuated the pros and cons with linked lists. In most cases, insertion and reorganizing the list is trivial while searching involves traversing the whole list. Working with a specialized sorting list and seeing its runtime complexities also showed how certain applications of a linked list can change its efficiency in certain algorithms.