# Assignment 2

# Name: Neron Parmar

# Student ID: 171690217

# File: Part B & Part C

# Date: 11/3/23

# All the work in the assignment is done by my own and no part of my assignment is shared with anyone.

# **Part B:**

1. **At Initial State:**

* At initial it will be completely empty

|  |  |
| --- | --- |
|  |  |

**HEAD**

1. **After Inserting the 5:**

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| **5** |  |

**HEAD**

1. **After Inserting the value 3:**

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| **5** |  |

|  |  |
| --- | --- |
| **3** |  |

**HEAD**

1. **After Inserting the value 8:**

* Before inserting the value, it will sort the values.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| **3** |  |

|  |  |
| --- | --- |
| **5** |  |

|  |  |
| --- | --- |
| **8** |  |

**HEAD**

1. **After Inserting the value 2:**

* Again, it will follow a similar process, it will sort the values.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| **2** |  |

|  |  |
| --- | --- |
| **3** |  |

|  |  |
| --- | --- |
| **5** |  |

|  |  |
| --- | --- |
| **8** |  |

**HEAD**

1. **After Inserting te value 6:**

* Again, as following the previous steps of sorting and inserting values.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| **2** |  |

|  |  |
| --- | --- |
| **3** |  |

|  |  |
| --- | --- |
| **5** |  |

|  |  |
| --- | --- |
| **6** |  |

|  |  |
| --- | --- |
| **8** |  |

**HEAD**

1. **So now removing the value 3 from the second element position:**

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| **2** |  |

|  |  |
| --- | --- |
| **5** |  |

|  |  |
| --- | --- |
| **6** |  |

**HEAD**

|  |  |
| --- | --- |
| **8** |  |

* So, as we remove the value 3 from the second element position all the values will start shifting and shuffling its position and give the sorted list like 2, 5, 6, 8 as we had removed the value 3 from the list.

# **PART C - ANALYSIS:**

Here’s the analysis for the code of all the four functions:

1. **def insert(self, data):**

* In this function the code begins with creating a new node with the given data, which is an O(1) operation.
* Then it checks if the list is not empty, it will enter a loop to fine the appropriate position to insert the new node and this loop can potentially iterate through the entire list, which is where the O(n) time complexity arise.
* Then the code checks and handles the three cases for insertion that is at the end, at the beginning or in the middle of the list. Each of these cases involves updating the pointers, which is still an O(1) operation.
* However, in a short way the insert function has a overall time complexity of O(n) because of the loop that is traversing the list to fins the correct position of the insertion.

1. **def remove(self, data):**

* This remove function iterates through the list using a while loop that checks each node one by one to see if it contains the data to be removed.
* If seeing in the worst-case scenarios, where the data to be removed is located at the end of the list or is not present in the list, the function will traverse all the nodes.
* Within the loop, the function performs constant-time operations for each node, but the number of iterations is directly proportional to the number of the nodes present in the list.
* Therefore, the time complexity of the remove function is O(n) where n is the number of the node in the list.

1. **def is\_present(self, data):**

* In the is\_present function, it checks the presence of the specific data value within the linked list. The it iterates through the list by starting at the head and examining each node.
* In the worst case scenario, it may need to traverse the entire list to confirm if the data is present in the list or not.
* The time complexity of the is\_present function is O(n) where the n is the number of the node in the list.

1. **def \_\_len\_\_(self):**

* In this \_\_len\_\_ function, it counts the number of the nodes in the list by iterating through the linked list, starting with the head and moving through the next pointers until it reaches the end of the list.
* The time complexity of this function is O(n) where n is the number of the node in the linked list.