**LABORATORY EXERCISE #3**

**Group Members:**

* Caranto, Marjoy M.
* Diamante, Danielle Carl
* Layo, Margaret Sydney
* Legaspi, Marian May
* Red, Francen Venisse
* Relator, Mary Ruth
* Tinawin, John Reydo

**CYS:** BSCPE 2-7

**Guide Questions:**

1. What are the advantages of using a linked list over an array?
   * A linked list offers several advantages over an array. Unlike arrays, linked lists do not require a fixed size, allowing them to grow or shrink dynamically as needed, which makes them more memory-efficient when the size of the data is unpredictable. Additionally, linked lists make it easier to insert or delete elements, as these operations do not require shifting elements like in arrays. Instead, you simply update the pointers of the nodes. This flexibility makes linked lists particularly useful for applications where frequent insertions and deletions are required. However, it is important to note that accessing elements in a linked list is slower compared to arrays, as you need to traverse the list sequentially to reach a specific element.
2. How does a linked list manage memory differently from arrays?
   * A linked list manages memory differently from arrays because arrays use a contiguous block of memory for all their elements and have a fixed size, meaning they reserve a certain amount of memory beforehand, even if some of it isn’t used. This setup allows for efficient random access, since the address of any element can be quickly calculated using its index and the base address. However, resizing an array is difficult since you need to create a new, larger array and copy all existing elements into it. A linked list, on the other hand, uses a dynamic memory allocation approach. Each element, or in this case a node, is stored separately in memory and linked together using pointers. This makes linked lists more flexible, allowing easy insertion or deletion of nodes without reorganizing the entire structure. However, this flexibility comes with a downside, as linked lists require extra memory for storing pointers and accessing elements takes more time compared to arrays.
3. What happens if we delete the head node in a singly linked list?
   * If we delete the head node in a singly linked list, the head pointer goes to the second node in the list. As seen in the delete node() method from exercise 1, it checks if the node deleted is the head and if it is, it moves the head pointer to the next node. This means that the old head node loses its reference and gets removed from the memory. If the head happens is the only node in the list, deleting it makes the list empty since next node would be None.
4. How can we modify this code to create a doubly linked list?
   * We can modify the code to create a doubly linked list by adding a prev pointer in the Node class so that each node can link to the previous node as well as the next one. The insert method should be updated to set both the next and prev pointers when adding a new node, ensuring proper connections in both directions. The delete method should also be adjusted to fix the forward and backward links after removing a node to maintain the structure of the list. Additionally, we can optionally traverse the list backward using the prev pointer.