A **doubly linked list** allows a greater variety of $O(1)$ time update operations including insertions and deletions at arbitrary positions within the list. We will use the term “next” and “prev” for the reference to the nodes that follows and precedes a node respectively. In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a **header** node at the beginning of the list and a **trailer** node at the end of the list. These “dummy” nodes are known as **sentinels** (or guards) and they do not store elements of the primary sequence. A doubly linked list with such sentinels is shown below.

When using sentinel nodes, an empty list is initialized so that the next field of the header points to the trailer and the prev field of the trailer points to the header, the remaining fields of the sentinels are irrelevant. Every insertion into the doubly linked list will take place between a pair of existing nodes. Similarly, during deletion, the two neighbors of the node to be deleted are linked directly to each other, thereby bypassing the original node.

We provide a preliminary implementation of a doubly linked list, in the form of a class named *_DoublyLinkedBase* in the attached file **DoblyLinkedBase.py**. In this file you need to complete the following tasks.
Question 1a: [1 Mark] __len__(self): This function should be able to return the number of elements in the list.
Question 1b: [1 Mark] is_empty(self): This function should be able to return true if list is empty.
Question 1c: [4 Marks] _insert_between(self, e, predecessor, successor): This function should be able to add element e between two existing nodes and return new node.
Question 1d: [4 Marks] _delete_node(self, node): This function should be able to delete nonsentinel node from the list and return the nodes element/data.