Amal Al-Dubai / software Engineering / Lab6 : Doubly Linked List

1. Describe a method for finding the middle node of a doubly linked list with header and trailer sentinels by “link hopping,” and without relying on explicit knowledge of the size of the list. In the case of an even number of nodes, report the node slightly left of center as the “middle.”

```
class Node:
    def __init__(self, value=None):
        self.value, self.prev, self.next = value, None, None

class DoublyLinkedList:
    def __init__(self):
        self.header, self.trailer = Node(), Node()
        self.header.next, self.trailer.prev = self.trailer, self.header

    def append(self, value):
        node = Node(value)
        last = self.trailer.prev
        last.next = node
        node.prev, node.next = last, self.trailer
        self.trailer.prev = node

    def find_middle(self):
        left, right = self.header.next, self.trailer.prev
        while left != right and left != right.next:
            left, right = left.next, right.prev
        return left.value

# Example usage:
dll = DoublyLinkedList()
for val in [1, 2, 3, 4, 5, 6, 7]: dll.append(val)
print(dll.find_middle())  # Output: 4
```

2. Give an implementation of the size( ) method for the DoublyLinkedList class, assuming that we did not maintain size as an instance variable.

```
def size(self):
    count = 0
    current = self.header.next
    while current != self.trailer:
        count += 1
        current = current.next
    return count
```

3. Implement the equals( ) method for the DoublyLinkedList class.

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.header = Node()  # Header sentinel
        self.trailer = Node()  # Trailer sentinel
        self.header.next = self.trailer
        self.trailer.prev = self.header

    def append(self, value):
        """Add a node with the given value to the end of the list."""
        new_node = Node(value)
        last = self.trailer.prev
        last.next = new_node
        new_node.prev = last
        new_node.next = self.trailer
        self.trailer.prev = new_node

    def size(self):
        """Return the size of the list by counting nodes."""
        count = 0
        current = self.header.next
        while current != self.trailer:
            count += 1
            current = current.next
        return count

    def equals(self, other):
        """Check if this list is equal to another list."""
        if not isinstance(other, DoublyLinkedList):
            return False

        current_self = self.header.next
        current_other = other.header.next

        while current_self != self.trailer and current_other != other.trailer:
            if current_self.value != current_other.value:
                return False
            current_self = current_self.next
            current_other = current_other.next

        # Ensure both lists are fully traversed
        return current_self == self.trailer and current_other == other.trailer

# Example usage:
dll1 = DoublyLinkedList()
dll2 = DoublyLinkedList()

# Add elements to both lists
for val in [1, 2, 3]:
    dll1.append(val)
    dll2.append(val)

# Test size method
print("Size of dll1:", dll1.size())  # Output: 3
```

```
print("Size of dll2:", dll2.size())  # Output: 3

print("Are dll1 and dll2 equal?", dll1.equals(dll2))  # Output: True
dll2.append(4)
print("Are dll1 and dll2 equal after modification?", dll1.equals(dll2))  # Output: False
```

4. Give an algorithm for concatenating two doubly linked lists L and M, with header and trailer sentinel nodes, into a single list L′.

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.header = Node()  # Header sentinel
        self.trailer = Node()  # Trailer sentinel
        self.header.next = self.trailer
        self.trailer.prev = self.header

    def append(self, value):
        """Add a node with the given value to the end of the list."""
        new_node = Node(value)
        last = self.trailer.prev
        last.next = new_node
        new_node.prev = last
        new_node.next = self.trailer
        self.trailer.prev = new_node

    def display(self):
        """Display the elements of the list."""
        current = self.header.next
        elements = []
        while current != self.trailer:
            elements.append(current.value)
            current = current.next
        print(" -> ".join(map(str, elements)))

def concatenate(L, M):
    """Concatenate two doubly linked lists L and M into one list L'."""
    if M.header.next == M.trailer:  # If M is empty
        return L

    L_last = L.trailer.prev
    M_first = M.header.next
    L_last.next = M_first
    M_first.prev = L_last

    M_last = M.trailer.prev
    L.trailer.prev = M_last
    M_last.next = L.trailer

    M.header.next = M.trailer
    M.trailer.prev = M.header

    return L

# Example Usage:
```

```
L = DoublyLinkedList()
M = DoublyLinkedList()

for val in [1, 2, 3]:
    L.append(val)
for val in [4, 5, 6]:
    M.append(val)

print("List L before concatenation:")
L.display()  # Output: 1 -> 2 -> 3

print("List M before concatenation:")
M.display()  # Output: 4 -> 5 -> 6

concatenate(L, M)

print("List L after concatenation:")
L.display()  # Output: 1 -> 2 -> 3 -> 4 -> 5 -> 6

print("List M after concatenation (should be empty):")
M.display()  # Output: (empty)
```

5. Our implementation of a doubly linked list relies on two sentinel nodes, header and trailer, but a single sentinel node that guards both ends of the list should suffice. Reimplement the DoublyLinkedList class using only one sentinel node.

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.sentinel = Node()
        self.sentinel.prev = self.sentinel
        self.sentinel.next = self.sentinel

    def append(self, value):
        """Add a node with the given value to the end of the list."""
        new_node = Node(value)
        last = self.sentinel.prev
        last.next = new_node
        new_node.prev = last
        new_node.next = self.sentinel
        self.sentinel.prev = new_node

    def size(self):
        """Return the size of the list."""
        count = 0
        current = self.sentinel.next
        while current != self.sentinel:
            count += 1
            current = current.next
        return count

    def display(self):
```

```
        """Display the elements of the list."""
        current = self.sentinel.next
        elements = []
        while current != self.sentinel:
            elements.append(current.value)
            current = current.next
        print(" -> ".join(map(str, elements)))

# Example Usage:
dll = DoublyLinkedList()
dll.append(1)
dll.append(2)
dll.append(3)
dll.display()  # Output: 1 -> 2 -> 3
print("Size:", dll.size())  # Output: 3
```

6. Implement a circular version of a doubly linked list, without any sentinels, that supports all the public behaviors of the original as well as two new update methods, rotate( ) and rotateBackward.

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.prev = None
        self.next = None

class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, value):
        new_node = Node(value)
        if not self.head:
            self.head = new_node
            new_node.next = new_node
            new_node.prev = new_node
        else:
            last = self.head.prev
            last.next = new_node
            new_node.prev = last
            new_node.next = self.head
            self.head.prev = new_node

    def size(self):
        if not self.head:
            return 0
        count = 1
        current = self.head.next
        while current != self.head:
            count += 1
            current = current.next
        return count

    def display(self):
        if not self.head:
            print("List is empty")
            return
```

```
        current = self.head
        elements = []
        while True:
            elements.append(current.value)
            current = current.next
            if current == self.head:
                break
        print(" -> ".join(map(str, elements)))

    def rotate(self):
        if self.head and self.head.next != self.head:
            self.head = self.head.next

    def rotateBackward(self):
        if self.head and self.head.prev != self.head:
            self.head = self.head.prev

# Example Usage:
dll = CircularDoublyLinkedList()
dll.append(1)
dll.append(2)
dll.append(3)
dll.display()  # Output: 1 -> 2 -> 3

dll.rotate()
dll.display()  # Output: 2 -> 3 -> 1

dll.rotateBackward()
dll.display()  # Output: 1 -> 2 -> 3
```

7. Implement the clone( ) method for the DoublyLinkedList class.

```
class DoublyLinkedList:
    def clone(self):
        """Clone the current doubly linked list."""
        clone_list = DoublyLinkedList()
        current = self.header.next
        while current != self.trailer:
            clone_list.append(current.value)
            current = current.next
        return clone_list
dll = DoublyLinkedList()
dll.append(1)
dll.append(2)
dll.append(3)

cloned_list = dll.clone()
cloned_list.display()  # Output: 1 -> 2 -> 3
```