## Topics

1. Implement Node Class
2. Implement DoublyLinkedList Class
3. Implement Basic Methods of DoublyLinkedList

* isEmpty()
* size()
* first()
* last()
* addFirst()
* addLast()
* removeFirst()
* removeLast()

public class Node<T> {

private T element;

private Node<T> prev;

private Node<T> next;

public Node(T element, Node<T> prev, Node<T> next) {

this.element = element;

this.prev = prev;

this.next = next;

}

public T getElement() {

return element;

}

public void setElement(T element) {

this.element = element;

}

public Node<T> getPrev() {

return prev;

}

public void setPrev(Node<T> prev) {

this.prev = prev;

}

public Node<T> getNext() {

return next;

}

public void setNext(Node<T> next) {

this.next = next;

}

}

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

head = null;

tail = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public Node<T> first() {

return head;

}

public Node<T> last() {

return tail;

}

public void addFirst(T element) {

Node<T> newNode = new Node<>(element, null, head);

if (isEmpty()) {

tail = newNode;

} else {

head.setPrev(newNode);

}

head = newNode;

size++;

}

public void addLast(T element) {

Node<T> newNode = new Node<>(element, tail, null);

if (isEmpty()) {

head = newNode;

} else {

tail.setNext(newNode);

}

tail = newNode;

size++;

}

public T removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("The list is empty.");

}

T removedElement = head.getElement();

head = head.getNext();

if (head == null) {

tail = null;

} else {

head.setPrev(null);

}

size--;

return removedElement;

}

public T removeLast() {

if (isEmpty()) {

throw new NoSuchElementException("The list is empty.");

}

T removedElement = tail.getElement();

tail = tail.getPrev();

if (tail == null) {

head = null;

} else {

tail.setNext(null);

}

size--;

return removedElement;

}

}

## Homework

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.”

public Node findMiddleNode() {

Node slow = header.next;

Node fast = header.next;

while (fast != trailer && fast.next != trailer) {

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

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

class DoublyLinkedListNode:

def \_\_init\_\_(self, data):

self.data = data

self.prev = None

self.next = None

class DoublyLinkedList:

def \_\_init\_\_(self):

self.head = None

self.tail = None

def size(self):

count = 0

current = self.head

while current is not None:

count += 1

current = current.next

return count

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

class DoublyLinkedListNode:

def \_\_init\_\_(self, data):

self.data = data

self.prev = None

self.next = None

class DoublyLinkedList:

def \_\_init\_\_(self):

self.head = None

self.tail = None

def equals(self, other\_list):

if not isinstance(other\_list, DoublyLinkedList):

return False

current\_self = self.head

current\_other = other\_list.head

while current\_self is not None and current\_other is not None:

if current\_self.data != current\_other.data:

return False

current\_self = current\_self.next

current\_other = current\_other.next

# If one list has more elements than the other,

# they are not equal.

if current\_self is not None or current\_other is not None:

return False

return True

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

class DoublyLinkedListNode:

def \_\_init\_\_(self, data):

self.data = data

self.prev = None

self.next = None

class DoublyLinkedList:

def \_\_init\_\_(self):

self.header = DoublyLinkedListNode(None)

self.trailer = DoublyLinkedListNode(None)

self.header.next = self.trailer

self.trailer.prev = self.header

def concatenate(self, other\_list):

if other\_list.header.next == other\_list.trailer:

return self # List M is empty, return List L

last\_L = self.trailer.prev

first\_M = other\_list.header.next

last\_L.next = first\_M

first\_M.prev = last\_L

self.trailer.prev = other\_list.trailer.prev

other\_list.trailer.prev.next = self.trailer

other\_list.header.next = self.header.next

self.header.next.prev = other\_list.header

return self

1. 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.

public class DoublyLinkedList {

private Node sentinel;

public DoublyLinkedList() {

sentinel = new Node();

sentinel.next = sentinel;

sentinel.prev = sentinel;

}

}

1. 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 CircularDoublyLinkedListNode:

def \_\_init\_\_(self, data):

self.data = data

self.prev = None

self.next = None

class CircularDoublyLinkedList:

def \_\_init\_\_(self):

self.current = None

def is\_empty(self):

return self.current is None

def insert(self, data):

new\_node = CircularDoublyLinkedListNode(data)

if self.is\_empty():

new\_node.prev = new\_node

new\_node.next = new\_node

self.current = new\_node

else:

new\_node.prev = self.current

new\_node.next = self.current.next

self.current.next.prev = new\_node

self.current.next = new\_node

def delete(self):

if not self.is\_empty():

if self.current.next == self.current:

self.current = None

else:

self.current.next.prev = self.current.prev

self.current.prev.next = self.current.next

self.current = self.current.next

def get\_current(self):

if not self.is\_empty():

return self.current.data

else:

return None

def rotate(self):

if not self.is\_empty():

self.current = self.current.next

def rotate\_backward(self):

if not self.is\_empty():

self.current = self.current.prev

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

@Override

public DoublyLinkedList clone() {

DoublyLinkedList newList = new DoublyLinkedList();

Node current = header.next;

while (current != trailer) {

newList.addLast(current.data);

current = current.next;

}

return newList;

}