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<T> findMiddleNode() {

Node<T> slow = header.next;

Node<T> fast = header.next;

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

fast = fast.next.next;

slow = slow.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.

public class DoublyLinkedList<T> {

private Node<T> header;

private Node<T> trailer;

// ... other methods for the DoublyLinkedList class

public int size() {

int count = 0;

Node<T> current = header.next;

while (current != trailer) {

count++;

current = current.next;

}

return count;

}

private static class Node<T> {

T data;

Node<T> prev;

Node<T> next;

Node(T data, Node<T> prev, Node<T> next) {

this.data = data;

this.prev = prev;

this.next = next;

}

}

}

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

public class DoublyLinkedList<T> {

private Node<T> header;

private Node<T> trailer;

// ... other methods for the DoublyLinkedList class

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

DoublyLinkedList<?> otherList = (DoublyLinkedList<?>) obj;

if (size() != otherList.size()) {

return false;

}

Node<T> currentThis = header.next;

Node<?> currentOther = otherList.header.next;

while (currentThis != trailer) {

if (!currentThis.data.equals(currentOther.data)) {

return false;

}

currentThis = currentThis.next;

currentOther = currentOther.next;

}

return true;

}

private static class Node<T> {

T data;

Node<T> prev;

Node<T> next;

Node(T data, Node<T> prev, Node<T> next) {

this.data = data;

this.prev = prev;

this.next = next;

}

}

}

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

public DoublyLinkedList<T> concatenateLists(DoublyLinkedList<T> L, DoublyLinkedList<T> M) {

if (L.isEmpty()) {

return M;

}

if (M.isEmpty()) {

return L;

}

Node<T> lastNodeL = L.trailer.prev;

Node<T> firstNodeM = M.header.next;

lastNodeL.next = firstNodeM;

firstNodeM.prev = lastNodeL;

Node<T> lastNodeM = M.trailer.prev;

lastNodeM.next = L.trailer;

L.trailer.prev = lastNodeM;

return L;

}

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<T> {

private Node<T> sentinel;

public DoublyLinkedList() {

sentinel = new Node<>(null);

sentinel.next = sentinel;

sentinel.prev = sentinel;

}

public boolean isEmpty() {

return sentinel.next == sentinel;

}

public void add(T element) {

Node<T> newNode = new Node<>(element);

newNode.next = sentinel.next;

newNode.prev = sentinel;

sentinel.next.prev = newNode;

sentinel.next = newNode;

}

public void remove(T element) {

Node<T> current = sentinel.next;

while (current != sentinel) {

if (current.data.equals(element)) {

current.prev.next = current.next;

current.next.prev = current.prev;

break;

}

current = current.next;

}

}

public void clear() {

sentinel.next = sentinel;

sentinel.prev = sentinel;

}

// ... other methods for the DoublyLinkedList class

private static class Node<T> {

T data;

Node<T> prev;

Node<T> next;

Node(T data) {

this.data = data;

}

}

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.

public class CircularDoublyLinkedList<T> {

private Node<T> head;

private int size;

public CircularDoublyLinkedList() {

head = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public void add(T element) {

Node<T> newNode = new Node<>(element);

if (isEmpty()) {

newNode.next = newNode;

newNode.prev = newNode;

head = newNode;

} else {

newNode.next = head;

newNode.prev = head.prev;

head.prev.next = newNode;

head.prev = newNode;

}

size++;

}

public void remove(T element) {

if (isEmpty()) {

return;

}

Node<T> current = head;

do {

if (current.data.equals(element)) {

if (size == 1) {

head = null;

} else {

current.prev.next = current.next;

current.next.prev = current.prev;

if (current == head) {

head = current.next;

}

}

size--;

return;

}

current = current.next;

} while (current != head);

}

public void clear() {

head = null;

size = 0;

}

public void rotate() {

if (isEmpty()) {

return;

}

head = head.next;

}

public void rotateBackward() {

if (isEmpty()) {

return;

}

head = head.prev;

}

// ... other methods and behaviors of the CircularDoublyLinkedList class

private static class Node<T> {

T data;

Node<T> prev;

Node<T> next;

Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

}

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

public DoublyLinkedList<T> clone() {

DoublyLinkedList<T> clonedList = new DoublyLinkedList<>();

Node<T> current = header.next;

while (current != trailer) {

clonedList.add(current.data);

current = current.next;

}

return clonedList;

}