1. Implement Node Class

public class Node<T> {

private T data;

private Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

public Node<T> getNext() {

return next;

}

public void setNext(Node<T> next) {

this.next = next;

}

}

1. Implement DoublyLinkedList Class

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 void addFirst(T data) {

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

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.setNext(head);

head.setPrevious(newNode);

head = newNode;

}

size++;

}

public void addLast(T data) {

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

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.setPrevious(tail);

tail.setNext(newNode);

tail = newNode;

}

size++;

}

public void removeFirst() {

if (isEmpty()) {

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

}

head = head.getNext();

if (head == null) {

tail = null;

} else {

head.setPrevious(null);

}

size--;

}

public void removeLast() {

if (isEmpty()) {

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

}

tail = tail.getPrevious();

if (tail == null) {

head = null;

} else {

tail.setNext(null);

}

size--;

}

public void printForward() {

Node<T> current = head;

while (current != null) {

System.out.print(current.getData() + " ");

current = current.getNext();

}

System.out.println();

}

public void printBackward() {

Node<T> current = tail;

while (current != null) {

System.out.print(current.getData() + " ");

current = current.getPrevious();

}

System.out.println();

}

private static class Node<T> {

private T data;

private Node<T> previous;

private Node<T> next;

public Node(T data) {

this.data = data;

this.previous = null;

this.next = null;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

public Node<T> getPrevious() {

return previous;

}

public void setPrevious(Node<T> previous) {

this.previous = previous;

}

public Node<T> getNext() {

return next;

}

public void setNext(Node<T> next) {

this.next = next;

}

}

public static void main(String[] args) {

DoublyLinkedList<Integer> list = new DoublyLinkedList<>();

list.addFirst(3);

list.addFirst(2);

list.addFirst(1);

list.printForward(); // Output: 1 2 3

list.printBackward(); // Output: 3 2 1

list.addLast(4);

list.addLast(5);

list.printForward(); // Output: 1 2 3 4 5

list.printBackward(); // Output: 5 4 3 2 1

list.removeFirst();

list.removeLast();

list.printForward(); // Output: 2 3 4

list.printBackward(); // Output: 4 3 2

}}

1. Implement Basic Methods of DoublyLinkedList

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

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 T first() {

if (isEmpty()) {

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

}

return head.getData();

}

public T last() {

if (isEmpty()) {

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

}

return tail.getData();

}

public void addFirst(T data) {

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

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.setNext(head);

head.setPrevious(newNode);

head = newNode;

}

size++;

}

public void addLast(T data) {

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

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.setPrevious(tail);

tail.setNext(newNode);

tail = newNode;

}

size++;

}

public void removeFirst() {

if (isEmpty()) {

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

}

head = head.getNext();

if (head == null) {

tail = null;

} else {

head.setPrevious(null);

}

size--;

}

public void removeLast() {

if (isEmpty()) {

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

}

tail = tail.getPrevious();

if (tail == null) {

head = null;

} else {

tail.setNext(null);

}

size--;

}

private static class Node<T> {

private T data;

private Node<T> previous;

private Node<T> next;

public Node(T data) {

this.data = data;

this.previous = null;

this.next = null;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

public Node<T> getPrevious() {

return previous;

}

public void setPrevious(Node<T> previous) {

this.previous = previous;

}

public Node<T> getNext() {

return next;

}

public void setNext(Node<T> next) {

this.next = next;

}

}

}

## 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 class DoublyLinkedList {

private Node header;

private Node trailer;

public Node findMiddle() {

Node slow = header.next;

Node fast = header.next;

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

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

private class Node {

private int data;

private Node prev;

private Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

}

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 {

private Node header;

private Node trailer;

public int size() {

int counter = 0;

Node cur = header.next;

while (cur != trailer) {

counter++;

cur = cur.next;

}

return counter;

}

}

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

public class DoublyLinkedList {

private Node header;

private Node trailer;

public boolean equals(DoublyLinkedList otherList) {

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

return false;

}

Node curThis = header.next;

Node curOther = otherList.header.next;

while (curThis != trailer) {

if (curThis.data != curOther.data) {

return false;

}

curThis = curThis.next;

curOther = curOther.next;

}

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′.

public class DoublyLinkedList {

private Node header;

private Node trailer;

public DoublyLinkedList concatenate(DoublyLinkedList listx) {

DoublyLinkedList concatenatedList = new DoublyLinkedList();

concatenatedList.header = this.header;

concatenatedList.trailer = this.trailer;

this.trailer.prev.next = listM.header.next;

listM.header.next.prev = this.trailer;

listx.trailer.prev.next = concatenatedList.trailer;

concatenatedList.trailer.prev = listM.trailer.prev;

Node cur = listx.header.next;

while (cur != listx.trailer) {

current.prev = concatenatedList.trailer.prev;

current.next = concatenatedList.trailer;

concatenatedList.trailer.prev.next = current;

concatenatedList.trailer.prev = current;

cur = curr.next;

}

return concatenatedList;

}

}