## Topics

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

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

class Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

private Node head;

private Node tail;

private int size;

public DoublyLinkedList() {

head = null;

tail = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public int first() {

if (isEmpty()) throw new NoSuchElementException("List is empty");

return head.data;

}

public int last() {

if (isEmpty()) throw new NoSuchElementException("List is empty");

return tail.data;

}

public void addFirst(int element) {

Node newNode = new Node(element);

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.next = head;

head.prev = newNode;

head = newNode;

}

size++;

}

public void addLast(int element) {

Node newNode = new Node(element);

if (isEmpty()) {

head=newNode;

tail=newNode;

} else {

tail.next=newNode;

newNode.prev=tail;

tail=newNode;

}

size++;

}

public void removeFirst() {

if (isEmpty()) throw new NoSuchElementException("List is empty");

if (head == tail) {

head=null ;

tail=null ;

} else {

head=head.next ;

head.prev=null ;

}

size--;

}

public void removeLast() {

if(isEmpty()) throw new NoSuchElementException("List is empty");

if(head==tail){

head=null ;

tail=null ;

}else{

tail=tail.prev ;

tail.next=null ;

}

size--;

}

}

## 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.”

class Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

private Node head;

private Node tail;

public DoublyLinkedList() {

head = null;

tail = null;

}

public void addLast(int element) {

Node newNode = new Node(element);

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

}

public boolean isEmpty() {

return head == null;

}

public Node middle() {

Node slow = head;

Node fast = head;

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

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 Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

private Node head;

private Node tail;

public DoublyLinkedList() {

head = null;

tail = null;

}

public boolean isEmpty() {

return head == null;

}

public int size() {

Node temp = head;

int size = 0;

while (temp != null) {

size++;

temp = temp.next;

}

return size;

}

// Other methods...

}

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

class Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

private Node head;

private Node tail;

public DoublyLinkedList() {

head = null;

tail = null;

}

public boolean isEmpty() {

return head == null;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

DoublyLinkedList that = (DoublyLinkedList) o;

Node current1 = this.head;

Node current2 = that.head;

while (current1 != null && current2 != null) {

if (current1.data != current2.data) return false;

current1 = current1.next;

current2 = current2.next;

}

return current1 == null && current2 == null;

}

}

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 Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

private Node head;

private Node tail;

public DoublyLinkedList() {

head = null;

tail = null;

}

public boolean isEmpty() {

return head == null;

}

public void addLast(int element) {

Node newNode = new Node(element);

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

}

public void concatenate(DoublyLinkedList M) {

if (!M.isEmpty()) {

if (this.isEmpty()) {

this.head = M.head;

this.tail = M.tail;

} else {

this.tail.next = M.head;

M.head.prev = this.tail;

this.tail = M.tail;

}

M.head = null;

M.tail = null;

}

}

}

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.

class Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

private Node sentinel;

public DoublyLinkedList() {

sentinel = new Node(-1); // Sentinel node with dummy data

sentinel.next = sentinel;

sentinel.prev = sentinel;

}

public boolean isEmpty() {

return sentinel.next == sentinel; // List is empty if the next node is the sentinel itself

}

public void addFirst(int element) {

Node newNode = new Node(element);

newNode.next = sentinel.next;

newNode.prev = sentinel;

sentinel.next.prev = newNode;

sentinel.next = newNode;

}

public void addLast(int element) {

Node newNode = new Node(element);

newNode.prev = sentinel.prev;

newNode.next = sentinel;

sentinel.prev.next = newNode;

sentinel.prev = newNode;

}

public void removeFirst() {

if (isEmpty()) throw new NoSuchElementException("List is empty");

sentinel.next = sentinel.next.next;

sentinel.next.prev = sentinel;

}

public void removeLast() {

if(isEmpty()) throw new NoSuchElementException("List is empty");

sentinel.prev = sentinel.prev.prev;

sentinel.prev.next = 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 Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class CircularDoublyLinkedList {

private Node head;

public CircularDoublyLinkedList() {

head = null;

}

public boolean isEmpty() {

return head == null;

}

public void addFirst(int element) {

Node 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;

head = newNode;

}

}

public void addLast(int element) {

Node 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;

}

}

public void removeFirst() {

if (isEmpty()) throw new NoSuchElementException("List is empty");

if (head.next == head) {

head = null;

} else {

head.next.prev = head.prev;

head.prev.next = head.next;

head = head.next;

}

}

public void removeLast() {

if(isEmpty()) throw new NoSuchElementException("List is empty");

if (head.next == head) {

head = null;

} else {

head.prev.prev.next = head;

head.prev = head.prev.prev;

}

}

public void rotate() {

if (!isEmpty()) {

head = head.next;

}

}

public void rotateBackward() {

if (!isEmpty()) {

head = head.prev;

}

}

}

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

class Node implements Cloneable {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone();

}

}

class DoublyLinkedList implements Cloneable {

private Node head;

private Node tail;

public DoublyLinkedList() {

head = null;

tail = null;

}

public boolean isEmpty() {

return head == null;

}

@Override

protected Object clone() throws CloneNotSupportedException {

DoublyLinkedList newList = (DoublyLinkedList) super.clone();

if (!isEmpty()) {

newList.head = (Node) this.head.clone();

newList.tail = (Node) this.tail.clone();

}

return newList;

}