## Topics

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;

}

}

Here's an example usage of the Node class

Node<Integer> node1 = new Node<>(10);

Node<Integer> node2 = new Node<>(20);

Node<Integer> node3 = new Node<>(30);

node1.setNext(node2);

node2.setNext(node3);

System.out.println(node1.getData()); // Output: 10

System.out.println(node1.getNext().getData()); // Output: 20

System.out.println(node1.getNext().getNext().getData()); // Output: 30

1. Implement DoublyLinkedList Class

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.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.setPrev(newNode);

head = newNode;

}

size++;

}

public void addLast(T data) {

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

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.setPrev(tail);

tail.setNext(newNode);

tail = newNode;

}

size++;

}

public void removeFirst() {

if (isEmpty()) {

return;

} else if (size == 1) {

head = null;

tail = null;

} else {

head = head.getNext();

head.setPrev(null);

}

size--;

}

public void removeLast() {

if (isEmpty()) {

return;

} else if (size == 1) {

head = null;

tail = null;

} else {

tail = tail.getPrev();

tail.setNext(null);

}

size--;

}

public void displayForward() {

Node<T> current = head;

while (current != null) {

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

current = current.getNext();

}

System.out.println();

}

public void displayBackward() {

Node<T> current = tail;

while (current != null) {

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

current = current.getPrev();

}

System.out.println();

}

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

public T getData() {

return data;

}

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;

}

}

}

1. Implement Basic Methods of DoublyLinkedList

* isEmpty()

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

public boolean isEmpty() {

return size == 0;

}

// Other methods...

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

// Getter and setter methods...

}

}

You can use this method to check if a DoublyLinkedList object is empty before performing other operations on it. For example:

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

System.out.println(list.isEmpty()); // Output: true

list.addFirst(10);

System.out.println(list.isEmpty()); // Output: false

list.removeFirst();

System.out.println(list.isEmpty()); // Output: true

* size()

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

public int size() {

return size;

}

// Other methods...

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

// Getter and setter methods...

}

}

You can use this method to retrieve the current size of a DoublyLinkedList object. For example:

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

System.out.println(list.size()); // Output: 0

list.addFirst(10);

list.addLast(20);

System.out.println(list.size()); // Output: 2

list.removeFirst();

System.out.println(list.size()); // Output: 1

* first()

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

public T first() {

if (isEmpty()) {

return null;

}

return head.getData();

}

// Other methods...

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

public T getData() {

return data;

}

// Getter and setter methods...

}

}

* last()

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

public T last() {

if (isEmpty()) {

return null;

}

return tail.getData();

}

// Other methods...

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

public T getData() {

return data;

}

// Getter and setter methods...

}

}

* addFirst()

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

public void addFirst(T data) {

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

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.setNext(head);

head.setPrev(newNode);

head = newNode;

}

size++;

}

// Other methods...

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

// Getter and setter methods...

}

}

* addLast()

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

public void addLast(T data) {

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

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.setPrev(tail);

tail.setNext(newNode);

tail = newNode;

}

size++;

}

// Other methods...

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

// Getter and setter methods...

}

}

* removeFirst()

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

public T removeFirst() {

if (isEmpty()) {

return null;

}

Node<T> removedNode = head;

if (head == tail) { // Only one element in the list

head = null;

tail = null;

} else {

head = head.getNext();

head.setPrev(null);

}

removedNode.setNext(null);

size--;

return removedNode.getData();

}

// Other methods...

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

// Getter and setter methods...

}

}

* removeLast()

public class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

public T removeLast() {

if (isEmpty()) {

return null;

}

Node<T> removedNode = tail;

if (head == tail) { // Only one element in the list

head = null;

tail = null;

} else {

tail = tail.getPrev();

tail.setNext(null);

}

removedNode.setPrev(null);

size--;

return removedNode.getData();

}

// Other methods...

private static class Node<T> {

private T data;

private Node<T> prev;

private Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

// Getter and setter methods...

}

}

## 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<T> findMiddleNode() {

Node<T> slow = header.getNext();

Node<T> fast = header.getNext();

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

slow = slow.getNext();

fast = fast.getNext().getNext();

}

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 int size() {

int count = 0;

Node<T> currentNode = header.getNext();

while (currentNode != trailer) {

count++;

currentNode = currentNode.getNext();

}

return count;

}

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

public boolean equals(DoublyLinkedList<T> otherList) {

if (otherList == this) {

return true;

}

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

return false;

}

Node<T> currentNode = header.getNext();

Node<T> otherNode = otherList.header.getNext();

while (currentNode != trailer) {

if (!currentNode.getData().equals(otherNode.getData())) {

return false;

}

currentNode = currentNode.getNext();

otherNode = otherNode.getNext();

}

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 DoublyLinkedList<T> concatenate(DoublyLinkedList<T> listM) {

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

if (this.isEmpty()) {

concatenatedList.copyFrom(listM);

return concatenatedList;

} else if (listM.isEmpty()) {

concatenatedList.copyFrom(this);

return concatenatedList;

}

Node<T> lastNodeL = this.trailer.getPrev();

Node<T> firstNodeM = listM.header.getNext();

lastNodeL.setNext(firstNodeM);

firstNodeM.setPrev(lastNodeL);

this.trailer.setNext(listM.header);

listM.header.setPrev(this.trailer);

concatenatedList.header = this.header;

concatenatedList.trailer = listM.trailer;

concatenatedList.size = this.size() + listM.size();

return concatenatedList;

}

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;

private int size;

public DoublyLinkedList() {

sentinel = new Node<>(null);

sentinel.setNext(sentinel);

sentinel.setPrev(sentinel);

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public void addFirst(T data) {

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

newNode.setNext(sentinel.getNext());

newNode.setPrev(sentinel);

sentinel.getNext().setPrev(newNode);

sentinel.setNext(newNode);

size++;

}

public void addLast(T data) {

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

newNode.setPrev(sentinel.getPrev());

newNode.setNext(sentinel);

sentinel.getPrev().setNext(newNode);

sentinel.setPrev(newNode);

size++;

}

public T removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty");

}

Node<T> firstNode = sentinel.getNext();

T data = firstNode.getData();

sentinel.setNext(firstNode.getNext());

firstNode.getNext().setPrev(sentinel);

firstNode.setPrev(null);

firstNode.setNext(null);

size--;

return data;

}

public T removeLast() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty");

}

Node<T> lastNode = sentinel.getPrev();

T data = lastNode.getData();

sentinel.setPrev(lastNode.getPrev());

lastNode.getPrev().setNext(sentinel);

lastNode.setPrev(null);

lastNode.setNext(null);

size--;

return data;

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

Node<T> currentNode = sentinel.getNext();

while (currentNode != sentinel) {

sb.append(currentNode.getData()).append(" ");

currentNode = currentNode.getNext();

}

return sb.toString().trim();

}

private class Node<E> {

private E data;

private Node<E> prev;

private Node<E> next;

public Node(E data) {

this.data = data;

prev = null;

next = null;

}

public E getData() {

return data;

}

public Node<E> getPrev() {

return prev;

}

public void setPrev(Node<E> prev) {

this.prev = prev;

}

public Node<E> getNext() {

return next;

}

public void setNext(Node<E> next) {

this.next = next;

}

}

}

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 addFirst(T data) {

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

if (isEmpty()) {

newNode.setNext(newNode);

newNode.setPrev(newNode);

} else {

newNode.setNext(head);

newNode.setPrev(head.getPrev());

head.getPrev().setNext(newNode);

head.setPrev(newNode);

}

head = newNode;

size++;

}

public void addLast(T data) {

if (isEmpty()) {

addFirst(data);

} else {

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

newNode.setNext(head);

newNode.setPrev(head.getPrev());

head.getPrev().setNext(newNode);

head.setPrev(newNode);

size++;

}

}

public T removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty");

}

T data = head.getData();

if (size == 1) {

head = null;

} else {

Node<T> newHead = head.getNext();

newHead.setPrev(head.getPrev());

head.getPrev().setNext(newHead);

head.setNext(null);

head.setPrev(null);

head = newHead;

}

size--;

return data;

}

public T removeLast() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty");

}

T data = head.getPrev().getData();

if (size == 1) {

head = null;

} else {

Node<T> tail = head.getPrev();

Node<T> newTail = tail.getPrev();

newTail.setNext(head);

head.setPrev(newTail);

tail.setNext(null);

tail.setPrev(null);

}

size--;

return data;

}

public void rotate() {

if (!isEmpty()) {

head = head.getNext();

}

}

public void rotateBackward() {

if (!isEmpty()) {

head = head.getPrev();

}

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

if (!isEmpty()) {

Node<T> currentNode = head;

do {

sb.append(currentNode.getData()).append(" ");

currentNode = currentNode.getNext();

} while (currentNode != head);

}

return sb.toString().trim();

}

private class Node<E> {

private E data;

private Node<E> prev;

private Node<E> next;

public Node(E data) {

this.data = data;

prev = null;

next = null;

}

public E getData() {

return data;

}

public Node<E> getPrev() {

return prev;

}

public void setPrev(Node<E> prev) {

this.prev = prev;

}

public Node<E> getNext() {

return next;

}

public void setNext(Node<E> next) {

this.next = next;

}

}

}

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

public class DoublyLinkedList<T> implements Cloneable {

private Node<T> header;

private Node<T> trailer;

private int size;

// Rest of the class implementation...

@Override

public DoublyLinkedList<T> clone() {

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

Node<T> currentNode = header.getNext();

while (currentNode != trailer) {

clonedList.addLast(currentNode.getData());

currentNode = currentNode.getNext();

}

return clonedList;

}

// Rest of the class implementation...

private static class Node<E> {

// Node implementation...

}

}