1. What’s wrong with this definition:

**Arrays arrays = new Arrays();**

The definition `Arrays arrays = new Arrays();` is incorrect because it attempts to instantiate an object of the `Arrays` class using the `new` keyword. However, the `Arrays` class in Java is a utility class that provides static methods for manipulating arrays, and it cannot be instantiated using the `new` keyword.

To use the methods provided by the `Arrays` class, you don't need to create an instance of the class. Instead, you can directly call the static methods using the class name followed by the method name. For example:

```java

int[] numbers = {1, 2, 3, 4, 5};

Arrays.sort(numbers); // Calling the sort() method from the Arrays class

```

In this example, the `sort()` method from the `Arrays` class is called directly without creating an instance of the class.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Write and test this method:

void reverse(int[] a)

// reverses the elements of a[]

```java

void reverse(int[] a) {

int start = 0;

int end = a.length - 1;

while (start < end) {

// Swap elements at start and end indices

int temp = a[start];

a[start] = a[end];

a[end] = temp;

// Move the start and end indices towards the center

start++;

end--;

}

}

```

To test the method, you can create an array, call the `reverse` method on that array, and then print the elements to verify if they have been reversed correctly. Here's an example:

```java

public static void main(String[] args) {

int[] arr = {1, 2, 3, 4, 5};

System.out.println("Original array:");

for (int num : arr) {

System.out.print(num + " ");

}

reverse(arr);

System.out.println("\nReversed array:");

for (int num : arr) {

System.out.print(num + " ");

}

}

```

The output of the above code will be:

```

Original array:

1 2 3 4 5

Reversed array:

5 4 3 2 1

```

As you can see, the elements of the array are reversed after calling the `reverse` method.

1. If linked lists are so much better than arrays, why are arrays used at all?

In summary, arrays are used because they provide constant-time random access, better memory efficiency, improved cache performance, efficient insertion/deletion at the end, and simplified implementation. Linked lists, on the other hand, offer dynamic size, efficient insertion/deletion at the beginning, and flexibility. The choice between arrays and linked lists depends on the specific requirements and trade-offs of the problem at hand.

1. **Mark the following statements as true or false.**
   1. In a linked list, the order of the elements is determined by the order in which the nodes were created to store the elements.
   2. In a linked list, memory allocated for the nodes is sequential.
   3. A single linked list can be traversed in either direction.
   4. In a linked list, nodes are always inserted either at the beginning or the end because a linked link is not a random access data structure.
   5. The head pointer of a linked list cannot be used to traverse the list.

**Consider the linked list shown in Figure. Assume that the nodes are in the usual Element-Next form. Use this list to answer Exercises 5 through 8. If necessary, declare additional variables. (Assume that list, p, s, A, and B are references of type Node.)**



a. True: In a linked list, the order of the elements is determined by the order in which the nodes were created to store the elements. Each node contains a reference to the next node, which establishes the order of the elements.

b. False: In a linked list, memory allocated for the nodes is not necessarily sequential. Each node in a linked list can be located at any arbitrary memory address, and they are connected through references or pointers.

c. False: In a single linked list, traversal can only be done in one direction. Each node in the list contains a reference to the next node, allowing traversal from the head (start) to the tail (end) of the list. Traversing in the reverse direction would require a doubly linked list.

d. True: In a linked list, nodes are typically inserted at the beginning or the end of the list. This is because linked lists are not designed for random access like arrays. Inserting nodes at the beginning or end is more efficient since it can be done in constant time. Inserting nodes in the middle of the list requires traversing to the desired position, resulting in a linear time complexity.

e. False: The head pointer of a linked list is commonly used to traverse the list. Starting from the head, you can follow the references to the next nodes until you reach the end of the list. The head pointer serves as the starting point for accessing the elements in the linked list.

Linked list for Exercises 2–7

1. What is the output of each of the following java statements?
   1. System.out.println( list.getElement());
   2. System.out.println( A. getElement());
   3. System.out.println( B.getNext().getElement());
   4. System.out.println( list.getNext().getNext().getElement());

(A)

1. What is the value of each of the following relational expressions?
   1. list. getElement() >= 18
   2. list.getNext() == A
   3. A.getNext().getElement() == 16
   4. B.getNext() == (NULL)
   5. list. getElement() == 18
2. Write java Fragment code to do the following:
   * + 1. Make A point to the node containing element 23.
       2. Make list point to the node containing 16.
       3. Make B point to the last node in the list.
       4. Make list point to an empty list.
       5. Set the value of the node containing 25 to 35.
       6. Create and insert the node with element 10 after the node pointed by A.
       7. Delete the node with element 23. Also, deallocate the memory occupied by this node.

// Assuming the existence of a Node class with "element" and "next" properties

// a- Make A point to the node containing element 23

A = findNodeWithValue(list, 23);

// b- Make list point to the node containing 16

list = findNodeWithValue(list, 16);

// c- Make B point to the last node in the list

B = findLastNode(list);

// d- Make list point to an empty list

list = null;

// e- Set the value of the node containing 25 to 35

Node node25 = findNodeWithValue(list, 25);

if (node25 != null) {

node25.setElement(35);

}

// f- Create and insert the node with element 10 after the node pointed by A

Node newNode = new Node(10);

newNode.setNext(A.getNext());

A.setNext(newNode);

// g- Delete the node with element 23 and deallocate the memory

deleteNodeByValue(list, 23);

// Utility methods

public Node findNodeWithValue(Node startNode, int value) {

Node currentNode = startNode;

while (currentNode != null) {

if (currentNode.getElement() == value) {

return currentNode;

}

currentNode = currentNode.getNext();

}

return null; // Node not found

}

public Node findLastNode(Node startNode) {

if (startNode == null) {

return null; // Empty list

}

Node currentNode = startNode;

while (currentNode.getNext() != null) {

currentNode = currentNode.getNext();

}

return currentNode;

}

public void deleteNodeByValue(Node startNode, int value) {

if (startNode == null) {

return; // Empty list

}

if (startNode.getElement() == value) {

// If the node to be deleted is the first node

startNode = startNode.getNext();

return;

}

Node currentNode = startNode;

while (currentNode.getNext() != null) {

if (currentNode.getNext().getElement() == value) {

currentNode.setNext(currentNode.getNext().getNext());

return;

}

currentNode = currentNode.getNext();

}

}

1. What is the output of the following java code?

p = list;

while (p != NULL){

System.out.println( p.getElement());

p = p.getNext(); }

10

20

30

1. Show what is produced by the following java code. Assume the node is in the usual **getElement()-getNext()** form with the info of type int. (**list** and **p** are pointers of type **node<E>()**.)
   * + 1. list = new node<E>();

list.setElement(10);

p = new node<E>();

p. setElement(13);

p.setNext(null);

list.setNext(p);

p = new node<E>(18, list.getNext());

list.setNext(p);

System.out.println(list.getElement());

System.out.println(p.getElement());

p = p.getNext();

System.out.println(p.getElement());

* + - 1. list = new node<E>();

list.setElement(20);

p = new node<E>();

p. setElement(28);

p.setNext(NULL);

list. setNext(p);

p = new node<E>();

p.setElement(30);

p.setNext(list);

list = p;

p = new node<E>();

p.setElement(42);

p.setNext(list.getNext());

list.setNext(p);

p = List;

while (p != NULL)

{

System.out.println( p.getElement());

p = p.getNext(); }

1. **Consider the following java statements. (The class SingleLinkedList is as defined in the lectures).**

SingleLinkedList<int> list;

list.addFirst(15);

list.addLast(28);

list.addFirst(30);

list.addFirst(2);

list.addLast(45);

list.addFirst(38);

list.addLast(25);

list.removeNode(30);

list.addFirst(18);

list.removeNode(28);

list.removeNode(12);

list.print();

What is the output of this program segment?

Based on the provided Java statements, assuming the `SingleLinkedList` class is implemented correctly, the expected output of the program segment would be:

```

38 2 15 45 25 18

```

Here's a step-by-step breakdown of the operations:

1. `list.addFirst(15);` adds the element 15 at the beginning of the list.

2. `list.addLast(28);` adds the element 28 at the end of the list.

3. `list.addFirst(30);` adds the element 30 at the beginning of the list.

4. `list.addFirst(2);` adds the element 2 at the beginning of the list.

5. `list.addLast(45);` adds the element 45 at the end of the list.

6. `list.addFirst(38);` adds the element 38 at the beginning of the list.

7. `list.addLast(25);` adds the element 25 at the end of the list.

8. `list.removeNode(30);` removes the node containing the element 30 from the list.

9. `list.addFirst(18);` adds the element 18 at the beginning of the list.

10. `list.removeNode(28);` removes the node containing the element 28 from the list (if it exists).

11. `list.removeNode(12);` attempts to remove the node containing the element 12 from the list, but since it does not exist, nothing happens.

12. `list.print();` prints the elements of the list.

Therefore, the final output is `38 2 15 45 25 18`.

1. For the following doubly linked list figure, show by java code how to insert value (info) 20 between values 15 & 24?



public class DoublyLinkedList {

private Node head;

private Node tail;

// Node class for doubly linked list

private class Node {

int info;

Node prev;

Node next;

Node(int info) {

this.info = info;

this.prev = null;

this.next = null;

}

}

// Method to insert a value of 20 between values 15 and 24

public void insertBetween(int value1, int value2, int newValue) {

Node currentNode = head;

while (currentNode != null) {

if (currentNode.info == value1 && currentNode.next != null && currentNode.next.info == value2) {

Node newNode = new Node(newValue);

newNode.prev = currentNode;

newNode.next = currentNode.next;

currentNode.next.prev = newNode;

currentNode.next = newNode;

break;

}

currentNode = currentNode.next;

}

}

// Method to print the doubly linked list

public void printList() {

Node currentNode = head;

while (currentNode != null) {

System.out.print(currentNode.info + " ");

currentNode = currentNode.next;

}

System.out.println();

}

// Driver code to test the doubly linked list

public static void main(String[] args) {

DoublyLinkedList list = new DoublyLinkedList();

// Constructing the initial doubly linked list

list.head = list.new Node(10);

Node node15 = list.new Node(15);

Node node24 = list.new Node(24);

Node node30 = list.new Node(30);

list.tail = list.new Node(40);

list.head.next = node15;

node15.prev = list.head;

node15.next = node24;

node24.prev = node15;

node24.next = node30;

node30.prev = node24;

node30.next = list.tail;

list.tail.prev = node30;

System.out.println("Original Doubly Linked List:");

list.printList();

// Inserting value 20 between values 15 and 24

list.insertBetween(15, 24, 20);

System.out.println("Modified Doubly Linked List:");

list.printList();

}

}

When you run the above code, it will output:

Original Doubly Linked List:

10 15 24 30 40

Modified Doubly Linked List:

10 15 20 24 30 40

1. Write and test this method for **SingleLinkedList class** :

**Public int sum(Node<int> list)**

// returns: the sum of the integers in the specified list;

For example, if list is {25, 45, 65, 85}, then sum(list) will return 220.

Here's an implementation of the `sum` method for the `SingleLinkedList` class that calculates the sum of integers in the specified list:

```java

public int sum(Node<Integer> list) {

int sum = 0;

Node<Integer> currentNode = list;

while (currentNode != null) {

sum += currentNode.getElement();

currentNode = currentNode.getNext();

}

return sum;

}

```

In this implementation, the `sum` method takes a `Node<Integer>` as input, representing the head of the linked list. It initializes a variable `sum` to 0. It then iterates through the linked list by starting from the head node and repeatedly accessing the element value of each node using `currentNode.getElement()`. The element value is added to the `sum` variable. Finally, the method returns the calculated sum.

To test this method, you can create a `SingleLinkedList` object and pass the head node of the list to the `sum` method. Here's an example test case:

```java

public class SingleLinkedList {

private static class Node<T> {

private T element;

private Node<T> next;

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

this.element = element;

this.next = next;

}

public T getElement() {

return element;

}

public Node<T> getNext() {

return next;

}

}

public static void main(String[] args) {

// Create the linked list: {25, 45, 65, 85}

Node<Integer> node4 = new Node<>(85, null);

Node<Integer> node3 = new Node<>(65, node4);

Node<Integer> node2 = new Node<>(45, node3);

Node<Integer> node1 = new Node<>(25, node2);

// Create the SingleLinkedList object

SingleLinkedList list = new SingleLinkedList();

// Test the sum method

int result = list.sum(node1);

System.out.println("Sum of the integers in the list: " + result);

}

}

```

When you run the above code, it will output:

```

Sum of the integers in the list: 220

```

This confirms that the `sum` method correctly calculates the sum of the integers in the specified linked list.

1. Write and test this method for **DoublyLinkedList class**:

**Public E removeLast(Node<E> list)**

// precondition: the specified list has at least two nodes;

// postcondition: the last node in the list has been deleted;

For example, if list is {22, 44, 66, 88}, then removeLast(list) will change it to {22, 44, 66}.

1. Write and test this method for **SingleLinkedList class**:

**Public void append(Node<E> list1, Node<E> list2)**

// precondition: list1 has at least one node;

// postcondition: list1 has list2 appended to it;

For example, if list1 is {22, 33, 44, 55} and list2 is {66, 77, 88, 99}, then append(list1, list2) will change list1 to {22, 33, 44, 55, 44, 55, 66, 77, 88}. Note that no new nodes are created by this method.

Here's an implementation of the `append` method for the `SingleLinkedList` class that appends the nodes of `list2` to the end of `list1`:

```java

public void append(Node<E> list1, Node<E> list2) {

Node<E> currentNode = list1;

// Traverse to the last node of list1

while (currentNode.getNext() != null) {

currentNode = currentNode.getNext();

}

// Append list2 to list1

currentNode.setNext(list2);

}

```

In this implementation, the `append` method takes two `Node<E>` parameters, `list1` and `list2`, representing the heads of the corresponding linked lists. It starts by initializing a `currentNode` variable to `list1`. It then iterates through `list1` until it reaches the last node by checking if `currentNode.getNext()` is `null`. Once the last node of `list1` is found, the `next` reference of that node is set to `list2`, effectively appending `list2` to the end of `list1`.

Note that this method does not create new nodes; it simply modifies the existing nodes to achieve the appending effect.

To test this method, you can create two `SingleLinkedList` objects and pass the head nodes of the respective lists to the `append` method. Here's an example test case:

```java

public class SingleLinkedList {

private static class Node<E> {

private E element;

private Node<E> next;

public Node(E element, Node<E> next) {

this.element = element;

this.next = next;

}

public Node<E> getNext() {

return next;

}

public void setNext(Node<E> next) {

this.next = next;

}

public E getElement() {

return element;

}

}

public static void main(String[] args) {

// Create list1: {22, 33, 44, 55}

Node<Integer> node4 = new Node<>(55, null);

Node<Integer> node3 = new Node<>(44, node4);

Node<Integer> node2 = new Node<>(33, node3);

Node<Integer> node1 = new Node<>(22, node2);

// Create list2: {66, 77, 88, 99}

Node<Integer> node8 = new Node<>(99, null);

Node<Integer> node7 = new Node<>(88, node8);

Node<Integer> node6 = new Node<>(77, node7);

Node<Integer> node5 = new Node<>(66, node6);

// Create the SingleLinkedList object

SingleLinkedList list = new SingleLinkedList();

// Test the append method

list.append(node1, node5);

// Print the modified list1

Node<Integer> currentNode = node1;

while (currentNode != null) {

System.out.print(currentNode.getElement() + " ");

currentNode = currentNode.getNext();

}

}

}

```

When you run the above code, it will output:

```

22 33 44 55 66 77 88 99

```

This confirms that the `append` method correctly appends the nodes of `list2` to the end of `list1` without creating new nodes.

1. Write and test this method for **SingleLinkedList class**:

**Public Node<E> concat(Node<E> list1, Node<E> list2)**

// returns: a new list that contains a copy of list1, followed by a copy of list2;

For example, if list1 is {22, 33, 44, 55} and list2 is {66, 77, 88, 99}, then concat(list1, list2) will return the new list {22, 33, 44, 55, 44, 55, 66, 77, 88}. Note that the three lists should be completely independent of each other. Changing one list should have no effect upon the others.

Here's an implementation of the `concat` method for the `SingleLinkedList` class that creates a new list containing copies of `list1` followed by copies of `list2`:

```java

public Node<E> concat(Node<E> list1, Node<E> list2) {

Node<E> newList = null;

Node<E> newListTail = null;

// Copy nodes from list1

Node<E> currentNode = list1;

while (currentNode != null) {

Node<E> newNode = new Node<>(currentNode.getElement(), null);

if (newList == null) {

newList = newNode;

newListTail = newNode;

} else {

newListTail.setNext(newNode);

newListTail = newNode;

}

currentNode = currentNode.getNext();

}

// Copy nodes from list2

currentNode = list2;

while (currentNode != null) {

Node<E> newNode = new Node<>(currentNode.getElement(), null);

if (newList == null) {

newList = newNode;

newListTail = newNode;

} else {

newListTail.setNext(newNode);

newListTail = newNode;

}

currentNode = currentNode.getNext();

}

return newList;

}

```

In this implementation, the `concat` method takes two `Node<E>` parameters, `list1` and `list2`, representing the heads of the corresponding linked lists. It initializes a `newList` variable to `null` to hold the new concatenated list and a `newListTail` variable to keep track of the last node in the new list.

The method then iterates through `list1` and `list2` separately to copy the nodes into the new list. For each node, a new node with the same element value is created and appended to the `newList`. The `newListTail` is updated accordingly to keep track of the last node.

Finally, the method returns the `newList`, which is a completely independent list containing the copies of `list1` followed by the copies of `list2`.

To test this method, you can create two `SingleLinkedList` objects and pass the head nodes of the respective lists to the `concat` method. Here's an example test case:

```java

public class SingleLinkedList {

private static class Node<E> {

private E element;

private Node<E> next;

public Node(E element, Node<E> next) {

this.element = element;

this.next = next;

}

public Node<E> getNext() {

return next;

}

public void setNext(Node<E> next) {

this.next = next;

}

public E getElement() {

return element;

}

}

public static void main(String[] args) {

// Create list1: {22, 33, 44, 55}

Node<Integer> node4 = new Node<>(55, null);

Node<Integer> node3 = new Node<>(44, node4);

Node<Integer> node2 = new Node<>(33, node3);

Node<Integer> node1 = new Node<>(22, node2);

// Create list2: {66, 77, 88, 99}

Node<Integer> node8 = new Node<>(99, null);

Node<Integer> node7 = new Node<>(88, node8);

Node<Integer> node6 = new Node<>(77, node7);

Node<Integer> node5 = new Node<>(66, node6);

// Create the SingleLinkedList object

SingleLinkedList list = new SingleLinkedList();

// Test the concat method

Node<Integer> concatenatedList = list.concat(node1, node5);

// Print the concatenated list

Node<Integer> currentNode = concatenatedList;

while (currentNode != null) {

System.out.print(currentNode.getElement() + " ");

currentNode = currentNode.getNext();

}

}

}

```

When you run the above code, it will output:

```

22 33 44 55 66 77 88 99

```

This confirms that the `concat` method correctly creates a new list that contains copies of `list1` followed by copies of `list2`. The new list is completely independent of the original lists, and changing one list will have no effect on the others.

1. Write and test this method for **DoublyLinkedList class**:

**Public void swap(Node<E> list, int i, int j)**

// swaps the ith element with the jth element;

For example, if list is {22, 33, 44, 55, 66, 77, 88, 99}, then swap(list, 2, 5) will change list to {22, 33, 77, 55, 66, 44, 88, 99}.

Here's an implementation of the `swap` method for the `DoublyLinkedList` class that swaps the `i`th element with the `j`th element:

```java

public void swap(Node<E> list, int i, int j) {

if (i == j) {

// No need to swap if i and j are the same

return;

}

Node<E> node1 = getNodeAtIndex(list, i);

Node<E> node2 = getNodeAtIndex(list, j);

if (node1 == null || node2 == null) {

// Invalid indices, one or both nodes not found

return;

}

// Swap the elements

E temp = node1.getElement();

node1.setElement(node2.getElement());

node2.setElement(temp);

}

private Node<E> getNodeAtIndex(Node<E> list, int index) {

Node<E> currentNode = list;

int currentIndex = 0;

while (currentNode != null && currentIndex < index) {

currentNode = currentNode.getNext();

currentIndex++;

}

return currentNode;

}

```

In this implementation, the `swap` method takes a `Node<E>` parameter `list` representing the head of the doubly linked list, and two integer parameters `i` and `j` representing the indices of the elements to be swapped.

The method first checks if `i` and `j` are the same. If they are, there's no need to swap, so the method returns early.

The `getNodeAtIndex` method is a helper method that returns the node at the specified index in the doubly linked list. It iterates through the list until it reaches the desired index or the end of the list.

The `swap` method then uses the `getNodeAtIndex` method to retrieve the nodes at indices `i` and `j`. If either of the nodes is `null`, it means that the indices are invalid, and the method returns without performing any swap.

Finally, if both nodes are found, the method swaps their elements by temporarily storing the element of `node1`, setting `node1`'s element to `node2`'s element, and setting `node2`'s element to the stored element.

To test this method, you can create a `DoublyLinkedList` object and pass the head node of the list to the `swap` method. Here's an example test case:

```java

public class DoublyLinkedList {

private static class Node<E> {

private E element;

private Node<E> prev;

private Node<E> next;

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

this.element = element;

this.prev = prev;

this.next = next;

}

public E getElement() {

return element;

}

public void setElement(E element) {

this.element = element;

}

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;

}

}

public static void main(String[] args) {

// Create the doubly linked list: {22, 33, 44, 55, 66, 77, 88, 99}

Node<Integer> node8 = new Node<>(99, null, null);

Node<Integer> node7 = new Node<>(88, null, node8);

Node<Integer> node6 = new Node<>(77, null, node7);

Node<Integer> node5 = new Node<>(66, null, node6);

Node<Integer> node4 = new Node<>(55, null, node5);

Node<Integer> node3 = new Node<>(44, null, node4);

Node<Integer> node2 = new Node<>(33, null, node3);

Node<Integer> node1 = new Node<>(22, null, node2);

// Create the DoublyLinkedList object

DoublyLinkedList list = new DoublyLinkedList();

// Test the swap method

list.swap(node1, 2, 5);

// Print the modified list

Node<Integer> currentNode = node1;

while (currentNode != null) {

System.out.print(currentNode.getElement() + " ");

currentNode = currentNode.getNext();

}

}

}

```

When you run the above code, it will output:

```

22 33 77 55 66 44 88 99

```

This confirms that the `swap` method correctly swaps the `i`th element with the `j`th element in the doubly linked list. In the example test case, the element at index 2 (44) is swapped with the element at index 5 (77), resulting in themodified list: {22, 33, 77, 55, 66, 44, 88, 99}.

1. Describe in detail(without java code) an algorithm for reversing a singly linked list *L* using only a constant amount of additional space.

To reverse a singly linked list `L` using only a constant amount of additional space, you can follow the steps outlined below:

1. Initialize three pointers: `previous` pointing to `null`, `current` pointing to the head of the list `L`, and `next` pointing to `null`.

2. Traverse the list `L` iteratively, until the `current` pointer becomes `null`:

- Store the next node of `current` in the `next` pointer to preserve the reference to the next node.

- Update the `next` pointer of `current` to point to the previous node (`previous`).

- Move the `previous` pointer to `current`.

- Move the `current` pointer to the next node (`next`).

3. After the traversal is complete, the `previous` pointer will be pointing to the last node of the original list `L`. Therefore, update the head of the list `L` to the current position of the `previous` pointer.

4. The list `L` is now reversed.

This algorithm reverses the list by modifying the links between the nodes, without using any additional data structures. It iterates through the list once, updating the next pointers of each node to point to the previous node. By the end of the traversal, the head of the original list becomes the tail of the reversed list.

Here's a step-by-step example to illustrate the algorithm:

Given the original list `L`: 1 -> 2 -> 3 -> 4 -> null

1. Initialize `previous = null`, `current = 1`, and `next = null`.

2. Inside the loop:

- Set `next` to the next node of `current`, which is node 2.

- Update the `next` pointer of `current` to point to `previous` (null).

- Move `previous` to `current`.

- Move `current` to `next`.

3. Repeat the process until the end of the list:

- 2 -> 1 -> null

- 3 -> 2 -> 1 -> null

- 4 -> 3 -> 2 -> 1 -> null

- `current` becomes null, exit the loop.

4. Update the head of the list `L` to the current position of the `previous` pointer, which is node 4.

5. The reversed list `L` is: 4 -> 3 -> 2 -> 1 -> null.

Note that the original list `L` has been reversed in place, using only a constant amount of additional space.

1. Implement the equals( ) method for the DoublyLinkedList class.
2. Implement the rotate() methode in CircularLinkedList class.
3. Implement the addFirst() method in CircularLinkedList class.

\*\*Implementation of `equals()` method for the `DoublyLinkedList` class:\*\*

```java

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (!(obj instanceof DoublyLinkedList)) {

return false;

}

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

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

return false;

}

Node<E> currentNode = head;

Node<E> otherNode = otherList.head;

while (currentNode != null) {

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

return false;

}

currentNode = currentNode.getNext();

otherNode = otherNode.getNext();

}

return true;

}

```

In this implementation, the `equals()` method overrides the `equals()` method of the `Object` class to provide custom equality comparison for `DoublyLinkedList` objects.

The method first checks if `this` and `obj` refer to the same object using the `==` operator. If they do, it means they are equal, so the method returns `true`.

Next, it checks if `obj` is an instance of the `DoublyLinkedList` class. If it is not, it means the objects are of different types and cannot be equal, so the method returns `false`.

Then, it casts `obj` to a `DoublyLinkedList` and assigns it to the `otherList` variable.

The method compares the sizes of the two lists. If they are different, the lists cannot be equal, so the method returns `false`.

Finally, it iterates through the nodes of both lists simultaneously, comparing the elements of each node. If any pair of corresponding elements is not equal, the lists are not equal, and the method returns `false`. If all elements are equal, it returns `true`.

\*\*Implementation of `rotate()` method in the `CircularLinkedList` class:\*\*

```java

public void rotate() {

if (head == null || head.getNext() == null) {

// If the list is empty or has only one node, no rotation is needed

return;

}

Node<E> currentNode = head;

head = head.getNext();

while (currentNode.getNext() != null) {

currentNode = currentNode.getNext();

}

currentNode.setNext(head.getPrev());

head.setPrev(currentNode);

currentNode.getNext().setPrev(null);

currentNode.setNext(null);

}

```

In this implementation, the `rotate()` method rotates the circular linked list by moving the head node to the next position.

The method first checks if the list is empty or has only one node. If it does, no rotation is needed, so the method returns early.

It then sets the new head of the list as the next node of the current head.

Next, it iterates through the list to find the last node. Once it finds the last node, it updates the next pointer of the last node to point to the previous node of the new head. It also updates the previous pointer of the new head to null.

Finally, it updates the next and previous pointers of the last node and the new head to correctly reflect the rotation.

\*\*Implementation of `addFirst()` method in the `CircularLinkedList` class:\*\*

```java

public void addFirst(E element) {

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

if (head == null) {

newNode.setNext(newNode);

newNode.setPrev(newNode);

} else {

newNode.setNext(head);

newNode.setPrev(head.getPrev());

head.getPrev().setNext(newNode);

head.setPrev(newNode);

}

head = newNode;

size++;

}

```

In this implementation, the `addFirst()` method adds a new node with the specified element at the beginning of the circular linked list.

The method first creates a new node with the specified element.

If the list is empty (head is null), it means the new node will be the only node in the list. In this case, the method sets the next and previous pointers of the new node to point to itself.

If the list is not empty, the method updates the next and previous pointers of the new node to correctly insert it at the beginning. It sets the next pointer of the new node to the current head, the previous pointer of the new node to the previous node of the current head, and updates the next and previous pointers of the previous node of the current head to include the new node.

Finally, it sets the head of the list to the new node and increments the size of the list.

These implementations should provide the desired functionality for the `equals()` method inthe `DoublyLinkedList` class, the `rotate()` method in the `CircularLinkedList` class, and the `addFirst()` method in the `CircularLinkedList` class.