‏What’s wrong with this definition:

‏Arrays arrays = new Arrays();

‏2- Write and test this method:

‏void reverse(int[] a)

‏// reverses the elements of a[]

void reverse(int[] a) {

for (int i = 0; i < a.length / 2; i++) {

int temp = a[i];

a[i] = a[a.length - 1 - i];

a[a.length - 1 - i] = temp;

}

}

```

This method takes an integer array `a` as input and reverses the order of its elements. Here's a step-by-step breakdown of how it works:

1. We loop through the first half of the array using the variable `i`.

2. For each iteration, we swap the element at index `i` with the corresponding element at index `a.length - 1 - i`.

3. Once we've swapped all elements in the first half of the array, the array is now reversed!

Here's an example usage of the `reverse()` method:

```java

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

System.out.println("Original array: " + Arrays.toString(a));

reverse(a);

System.out.println("Reversed array: " + Arrays.toString(a));

```

This code would output the following:

```

Original array: [1, 2, 3, 4, 5]

Reversed array: [5, 4, 3, 2, 1]

```

‏3- If linked lists are so much better than arrays, why are arrays used at all?

Arrays are still used in certain scenarios despite the advantages of linked lists for several reasons:

1. Random Access: Arrays allow for direct access to any element based on its index, which means the time complexity for accessing an element is O(1). In contrast, linked lists require traversing the list from the beginning to access a specific node, resulting in a time complexity of O(n) in the worst case.

2. Efficient Memory Allocation: Arrays have a more efficient memory allocation mechanism compared to linked lists. Linked lists use additional memory to store references or pointers, which can consume more memory compared to arrays, especially for small-sized data structures.

3. Efficient Cache Usage: Arrays exhibit better cache locality and improved cache usage compared to linked lists. This is because elements in an array are stored contiguously in memory, allowing for better utilization of CPU cache and faster memory access during iteration.

4. Simplicity and Ease of Use: Arrays have a simpler data structure and are easier to implement and use compared to linked lists, which require additional logic for maintaining pointers and handling node connections.

5. Performance in Certain Operations: Arrays outperform linked lists in some operations, such as searching for an element that requires direct access. Additionally, arrays often have a smaller constant factor overhead compared to linked lists.

While linked lists offer certain advantages like dynamic resizing and efficient element insertion and deletion, arrays still have their place and are used when their specific benefits are required.

‏4- Mark the following statements as true or false.

‏a. In a linked list, the order of the elements is determined by the order in which the nodes were created to store the elements.

‏b. In a linked list, memory allocated for the nodes is sequential.

‏c. A single linked list can be traversed in either direction.

‏d. 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.

‏e. 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.)

‏Linked list for Exercises 2–7

B

‏5- What is the output of each of the following java statements?

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

‏b. System.out.println(  A. getElement());

‏c. System.out.println(  B.getNext().getElement());

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

‏6- What is the value of each of the following relational expressions?

‏a. list. getElement() >= 18

‏b. list.getNext() == A

‏c. A.getNext().getElement() == 16

‏d. B.getNext() == (NULL)

‏e. list. getElement() == 18

الجواب :

a. list.getElement() >= 18

- This expression will evaluate to a boolean value, true or false, depending on whether the value returned by the method `getElement()` of the object `list` is greater than or equal to 18.

b. list.getNext() == A

- This expression will evaluate to a boolean value, true or false, depending on whether the object returned by the method `getNext()` of the object `list` is the same as the object `A`.

c. A.getNext().getElement() == 16

- This expression will evaluate to a boolean value, true or false, depending on whether the value returned by the method `getElement()` of the object returned by the method `getNext()` of the object `A` is equal to 16.

d. B.getNext() == null

- This expression will evaluate to a boolean value, true or false, depending on whether the object returned by the method `getNext()` of the object `B` is null.

e. list.getElement() == 18

- This expression will evaluate to a boolean value, true or false, depending on whether the value returned by the method `getElement()` of the object `list` is equal to 18.

‏7- Write java Fragment code to do the following:

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

‏b- Make list point to the node containing 16.

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

‏d- Make list point to an empty list.

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

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

‏g- Delete the node with element 23. Also, deallocate the memory occupied by this node.

 الجواب:

import java.util.LinkedList;

public class FragmentCode {

public static void main(String[] args) {

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

list.add(12);

list.add(23);

list.add(16);

list.add(25);

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

LinkedList.Node<Integer> A = list.getNodeAtIndex(list.indexOf(23));

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

LinkedList.Node<Integer> node16 = list.getNodeAtIndex(list.indexOf(16));

LinkedList.Node<Integer> list = new LinkedList<>();

list.add(node16.getData());

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

LinkedList.Node<Integer> B = list.getTail();

// d - Make list point to an empty list

list.clear();

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

LinkedList.Node<Integer> node25 = list.getNodeAtIndex(list.indexOf(25));

node25.setData(35);

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

LinkedList.Node<Integer> newNode = new LinkedList.Node<>(10);

list.insertAfter(A, newNode);

// g - Delete the node with element 23, deallocate its memory

LinkedList.Node<Integer> node23 = list.getNodeAtIndex(list.indexOf(23));

list.remove(node23);

}

}

```

‏8- What is the output of the following java code?

‏p = list;

‏while (p != NULL){

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

‏p = p.getNext();  }

 الجواب : error

‏9- 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>().)

‏a- 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());

‏b- 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();     }

الجواب  a- The code creates a linked list with two nodes. The first node has an element of 10 and points to the second node. The second node has an element of 13 and points to null. Then, a new node is created with an element of 18 and the next pointer is set to the next node of the first node. The next pointer of the first node is then updated to point to the new node. The following output will be produced:

10

18

13

b- The code creates a linked list with three nodes. The first node has an element of 20 and points to the second node. The second node has an element of 28 and points to null. Then, a new node is created with an element of 30 and points to the first node. The variable "list" is updated to point to the new node. Another new node is created with an element of 42 and the next pointer is set to the next node of the second node. The next pointer of the second node is then updated to point to the new node. The following output will be produced:

30

20

42

‏10- 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?

 الجواب :error

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

الجواب:

```java

// Definition of the doubly linked node

class Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

Node head;

// Insert the value 20 between the values 15 and 24

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

// Create the new node

Node newNode = new Node(newValue);

// Find the node containing the value 15

Node currentNode = head;

while (currentNode != null && currentNode.data != value1) {

currentNode = currentNode.next;

}

// Identify the node containing the value 24

Node nextNode = currentNode.next;

// Link the new node between the two nodes

newNode.prev = currentNode;

newNode.next = nextNode;

currentNode.next = newNode;

if (nextNode != null) {

nextNode.prev = newNode;

}

}

// Print the doubly linked list

void printList() {

Node currentNode = head;

while (currentNode != null) {

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

currentNode = currentNode.next;

}

}

}

You can test the solution by creating a doubly linked list and inserting the value 20 between the values 15 and 24 as follows:

```java

public class Main {

public static void main(String[] args) {

DoublyLinkedList list = new DoublyLinkedList();

list.head = new Node(15);

Node secondNode = new Node(24);

list.head.next = secondNode;

secondNode.prev = list.head;

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

list.printList();

list.insertBetween(15, 24, 20);

System.out.println("\n\nAfter inserting the value 20: ");

list.printList();

}

}

The list will be printed with the input value 20 inserted between the values 15 and 24.

‏12- 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.

 الجواب:

‏```java

‏class Node<T> {

‏ T data;

‏ Node<T> next;

‏ public Node(T data) {

‏ this.data = data;

‏ this.next = null;

}

}

‏class SingleLinkedList<T extends Number> {

‏ private Node<T> head;

‏ public int sum(Node<T> list) {

‏ int sum = 0;

‏ Node<T> currentNode = list;

‏ while (currentNode != null) {

‏ sum += currentNode.data.intValue();

‏ currentNode = currentNode.next;

}

‏ return sum;

}

}

‏public class Main {

‏ public static void main(String[] args) {

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

‏ Node<Integer> head = new Node<>(25);

‏ Node<Integer> secondNode = new Node<>(45);

‏ Node<Integer> thirdNode = new Node<>(65);

‏ Node<Integer> fourthNode = new Node<>(85);

‏ head.next = secondNode;

‏ secondNode.next = thirdNode;

‏ thirdNode.next = fourthNode;

‏ int sum = list.sum(head);

‏ System.out.println("Sum: " + sum);

}

}

```

‏This solution defines a `SingleLinkedList` class with a `sum` method that takes a `Node` as a parameter and returns the sum of the integers in the specified list. In the `sum` method, it iterates over the list, adding the integer values to the `sum` variable. Finally, it returns the sum.

‏In the `Main` class, a `SingleLinkedList` object is created and a list is created with values 25, 45, 65, and 85. The `sum` method is called on the head of the list, and the sum is printed to the console.

‏The output will be:

```

‏Sum: 220

```

‏13- 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}. الجواب:

class Node<E> {

E data;

Node<E> prev;

Node<E> next;

public Node(E data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList<E> {

private Node<E> head;

private Node<E> tail;

public E removeLast(Node<E> list) {

if (list.next == null) {

return null;

}

Node<E> currentNode = list;

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

currentNode = currentNode.next;

}

Node<E> lastNode = currentNode.next;

currentNode.next = null;

lastNode.prev = null;

return lastNode.data;

}

}

public class Main {

public static void main(String[] args) {

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

Node<Integer> head = new Node<>(22);

Node<Integer> secondNode = new Node<>(44);

Node<Integer> thirdNode = new Node<>(66);

Node<Integer> fourthNode = new Node<>(88);

head.next = secondNode;

secondNode.prev = head;

secondNode.next = thirdNode;

thirdNode.prev = secondNode;

thirdNode.next = fourthNode;

fourthNode.prev = thirdNode;

Integer removedData = list.removeLast(head);

System.out.println("Removed Data: " + removedData);

Node<Integer> currentNode = head;

while (currentNode != null) {

System.out.println(currentNode.data);

currentNode = currentNode.next;

}

}

}

```

This solution defines a `DoublyLinkedList` class with a `removeLast` method that takes a `Node` as a parameter and removes the last node in the list. In the `removeLast` method, it iterates over the list to find the second-to-last node, and then removes the last node by updating the `next` and `prev` references. It returns the data of the removed node.

In the `Main` class, a `DoublyLinkedList` object is created and a list is created with values 22, 44, 66, and 88. The `removeLast` method is called on the head of the list, and the removed data is printed to the console. Finally, the updated list is printed.

The output will be:

```

Removed Data: 88

22

44

66

```

‏14- 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.

 الجواب :

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

// Find the last node of list1

Node<E> currentNode = list1;

while (currentNode.next != null) {

currentNode = currentNode.next;

}

// Append list2 to list1

currentNode.next = list2;

}

```

To test this method, you can create two instances of `Node` representing `list1` and `list2`, and then call the `append` method passing these instances:

```java

Node<Integer> list1 = new Node<>(22);

list1.next = new Node<>(33);

list1.next.next = new Node<>(44);

list1.next.next.next = new Node<>(55);

Node<Integer> list2 = new Node<>(66);

list2.next = new Node<>(77);

list2.next.next = new Node<>(88);

list2.next.next.next = new Node<>(99);

append(list1, list2);

```

After calling `append(list1, list2)`, the `list1` variable will now contain the appended linked list: `{22, 33, 44, 55, 66, 77, 88, 99}`.

‏15- 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.

 الجواب

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

// Create a new list to store the concatenated elements

Node<E> newList = new Node<>(list1.data);

Node<E> currentNode = newList;

Node<E> current = list1.next;

// Copy all elements from list1 to the new list

while (current != null) {

currentNode.next = new Node<>(current.data);

currentNode = currentNode.next;

current = current.next;

}

// Copy all elements from list2 to the new list

current = list2;

while (current != null) {

currentNode.next = new Node<>(current.data);

currentNode = currentNode.next;

current = current.next;

}

return newList;

}

```

To test this method, you can create the two instances of `Node` representing `list1` and `list2`, and then call the `concat` method passing these instances. For example:

```java

Node<Integer> list1 = new Node<>(22);

list1.next = new Node<>(33);

list1.next.next = new Node<>(44);

list1.next.next.next = new Node<>(55);

Node<Integer> list2 = new Node<>(66);

list2.next = new Node<>(77);

list2.next.next = new Node<>(88);

list2.next.next.next = new Node<>(99);

Node<Integer> concatenatedList = concat(list1, list2);

```

After calling `concat(list1, list2)`, the `concatenatedList` variable will now contain the concatenated linked list: `{22, 33, 44, 55, 66, 77, 88, 99}`. Note that `list1` and `list2` remain unchanged and are independent of `concatenatedList`.

‏16- 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}.

 الحواب

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

if (list == null) {

return; // Nothing to swap if the list is empty

}

Node<E> currentNode = list;

Node<E> nodeA = null;

Node<E> nodeB = null;

int counter = 0;

// Traverse the list to find the ith and jth nodes

while(currentNode != null) {

if (counter == i) {

nodeA = currentNode;

}

if (counter == j) {

nodeB = currentNode;

}

currentNode = currentNode.next;

counter++;

}

// Swap the data between the ith and jth nodes

if (nodeA != null && nodeB != null) {

E temp = nodeA.data;

nodeA.data = nodeB.data;

nodeB.data = temp;

}

}

```

To test this method, you can create an instance of `DoublyLinkedList` representing the list and then call the `swap` method passing the list and the indices. For example:

```java

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

list.add(22);

list.add(33);

list.add(44);

list.add(55);

list.add(66);

list.add(77);

list.add(88);

list.add(99);

System.out.println("Before swap: " + list); // Output: Before swap: {22, 33, 44, 55, 66, 77, 88, 99}

list.swap(list.head, 2, 5);

System.out.println("After swap: " + list); // Output: After swap: {22, 33, 77, 55, 66, 44, 88, 99}

```

After calling the `swap` method with indices 2 and 5, the list will be updated to `{22, 33, 77, 55, 66, 44, 88, 99}` as shown in the output.

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

الجواب:

1. Initialize three pointers: current, previous, and next.

2. Set the current pointer to the head of the linked list.

3. Set the previous pointer to null.

4. Iterate through the linked list until the current pointer becomes null.

5. Inside the loop, do the following:

- Set the next pointer to the next node after the current node.

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

- Update the previous pointer to point to the current node.

- Move the current pointer to the next node.

6. After the loop finishes, the previous pointer will be pointing to the last node of the original linked list (which will now be the first node of the reversed list).

7. Update the head pointer of the linked list to point to the node referenced by the previous pointer.

8. The linked list is now reversed.

This algorithm reverses the linked list by changing the direction of the pointers. Since it only uses three pointers and doesn't create any additional data structures, it uses a constant amount of additional space.

‏18- Implement the equals( ) method for the DoublyLinkedList class.

‏19- Implement the rotate() methode in CircularLinkedList class.

‏20- Implement the addFirst() method in CircularLinkedList class.

‏ What’s wrong with this definition:

Arrays arrays = new Arrays();

الجواب :

‏public boolean equals(Object obj) {

‏ // Check if the object reference is pointing to the same instance

‏ if (this == obj) {

‏ return true;

}

‏ // Check if the object is of the same type

‏ if (!(obj instanceof DoublyLinkedList)) {

‏ return false;

}

‏ // Typecast the object to DoublyLinkedList

‏ DoublyLinkedList otherLinkedList = (DoublyLinkedList) obj;

‏ // Check if the size of the two lists is different

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

‏ return false;

}

‏ // Iterate through the lists and compare each element

‏ Node currentThis = this.head;

‏ Node currentOther = otherLinkedList.head;

‏ while (currentThis != null) {

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

‏ return false;

}

‏ currentThis = currentThis.next;

‏ currentOther = currentOther.next;

}

‏ // If all elements are equal, return true

‏ return true;

}

```

‏19- Here is an implementation of the `rotate()` method in the `CircularLinkedList` class:

‏```java

‏public void rotate() {

‏ if (head != null && head.next != null) {

‏ Node lastNode = head;

‏ // Traverse to the last node

‏ while (lastNode.next != head) {

‏ lastNode = lastNode.next;

}

‏ // Make the last node the new head

‏ head = head.next;

‏ // Connect the last node to the new head

‏ lastNode.next = head;

}

}

```

‏20- Here is an implementation of the `addFirst()` method in the `CircularLinkedList` class:

‏```java

‏public void addFirst(T data) {

‏ Node newNode = new Node(data);

‏ if (head == null) {

‏ head = newNode;

‏ newNode.next = newNode;

‏ } else {

‏ newNode.next = head;

‏ Node lastNode = head;

‏ // Traverse to the last node

‏ while (lastNode.next != head) {

‏ lastNode = lastNode.next;

}

‏ lastNode.next = newNode;

‏ head = newNode;

}

}

```

‏Regarding the line `Arrays arrays = new Arrays();`, the issue is that `Arrays` is actually a class in the Java standard library, and you cannot create an instance of it using the `new` keyword. To create a new array, you should use the array initializer syntax:

‏```java

‏int[] array = new int[5];

```

‏Alternatively, if you are looking to create an ArrayList, you can use:

‏```java

‏ArrayList<Object> arrayList = new ArrayList<>();

```