1. What’s wrong with this definition:

**Arrays = new Arrays();**

**المشكلة في هذا التعريف هي استخدام الكلمة "Arrays" لتعريف الكائن. في لغة Java، "Arrays" هو اسم الفئة المستخدمة لتوفير وظائف مرتبطة بالمصفوفات، ولا يمكن استخدامها لإنشاء كائن جديد من الفئة.**

**بدلاً من ذلك، يجب استخدام الكلمة الرئيسية "new" مع اسم الفئة التي ترغب في إنشاء كائن منها. في هذه الحالة، يمكن استخدامها كالتالي:**

**```java**

**int[] array = new int[5];**

**```**

**في هذا المثال، تم إنشاء كائن جديد من الفئة "int[]" (وتعني أنها مصفوفة من الأعداد الصحيحة) باستخدام الكلمة الرئيسية "new"، وتم تحديد حجم المصفوفة بـ 5.**

1. Write and test this method:

void reverse(int[] a)

// reverses the elements of a[]

يمكنك تنفيذ واختبار الطريقة التالية لعكس عناصر المصفوفة:

```java

void reverse(int[] a) {

int start = 0;

int end = a.length - 1;

while (start < end) {

// قم بتبديل العناصر

int temp = a[start];

a[start] = a[end];

a[end] = temp;

// انتقل إلى العنصر التالي

start++;

end--;

}

}

```

لتجربة الطريقة، يمكنك استخدام الشفرة التالية:

```java

public class Main {

public static void main(String[] args) {

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

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

printArray(array);

reverse(array);

System.out.println("Reversed Array:");

printArray(array);

}

static void printArray(int[] a) {

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

System.out.print(a[i] + " ");

}

System.out.println();

}

}

```

نتوقع أن يتم طباعة المصفوفة بالترتيب العكسي بعد استدعاء الدالة reverse().

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

**على الرغم من أن قوائم الروابط لها بعض الفوائد على المصفوفات، إلا أن المصفوفات لا تزال تستخدم لأسباب معينة. هنا بعض الأسباب التي تجعل استخدام المصفوفات مفيدًا:**

**1. سهولة الوصول إلى العناصر: في المصفوفات، يمكن الوصول إلى العناصر مباشرة بواسطة الفهرس. بينما في قوائم الروابط، يتطلب الوصول إلى عنصر معين البحث عنه من البداية. هذا يجعل المصفوفات أكثر كفاءة في بعض الحالات.**

**2. تخزين البيانات المتتابعة: إذا كانت البيانات المخزنة تتبع تسلسلًا متصلًا ولا يحتاج إلى إدراج أو حذف متكرر، فإن المصفوفات توفر طريقة بسيطة لتخزين هذه البيانات والوصول إليها.**

**3. قدرة التنبؤ بالحجم: في المصفوفات، يمكن تحديد حجم المصفوفة مسبقًا، بينما في قوائم الروابط يمكن أن تنمو وتنقص حسب الحاجة. إذا كانت الحجم المتوقع للبيانات معروفًا مسبقًا وثابتًا، فإن استخدام المصفوفات يمكن أن يكون أكثر كفاءة من حيث إدارة الذاكرة.**

**4. تنفيذ بعض العمليات بشكل أسرع: في بعض الحالات، المصفوفات يمكن أن توفر أداءًا أفضل لتنفيذ بعض العمليات مثل الاسترجاع العشوائي للعناصر أو الحلقات التكرارية.**

**بالإضافة إلى ذلك، تعتمد الاختيار بين المصفوفات وقوائم الروابط على الاحتياجات والمتطلبات الخاصة بالتطبيق المحدد. قد يكون استخدام قوائم الروابط أكثر ملاءمة في بعض الحالات مثل الإدراج والحذف المتكرر، في حين يمكن أن تكون المصفوفات أكثر ملاءمة في حالات أخرى مثل الوصول السريع إلى العناصر.**

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.)**



Linked list for Exercises 2–7

False

False

False

True

False

1. What is the output of each of the following java statements?
   1. System.out.println( list.getElement());

18

* 1. System.out.println( A. getElement());

32

* 1. System.out.println( B.getNext().getElement());

25

* 1. System.out.println( list.getNext().getNext().getElement());

23

1. What is the value of each of the following relational expressions?
   1. list. getElement() >= 18

True

* 1. list.getNext() == A

True

* 1. A.getNext().getElement() == 16

False

* 1. B.getNext() == (NULL)

False

* 1. list. getElement() == 18

True

1. 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.

يمكنك استخدام الشفرة التالية لتنفيذ المهام المطلوبة:

أفترض أن لديك الفئة التالية لتمثيل العقدة:

```java

class Node {

int element;

Node next;

}

```

a. جعل A يشير إلى العقدة التي تحتوي على العنصر 23:

```java

A = findNodeWithValue(list, 23);

```

b. جعل list تشير إلى العقدة التي تحتوي على العنصر 16:

```java

list = findNodeWithValue(list, 16);

```

c. جعل B يشير إلى العقدة الأخيرة في القائمة:

```java

B = getLastNode(list);

```

d. جعل list يشير إلى قائمة فارغة:

```java

list = null;

```

e. تعيين قيمة العقدة التي تحتوي على 25 إلى 35:

```java

Node node25 = findNodeWithValue(list, 25);

if (node25 != null) {

node25.element = 35;

}

```

f. إنشاء وإدراج العقدة التي تحتوي على العنصر 10 بعد العقدة المشار إليها بواسطة A:

```java

Node newNode = new Node();

newNode.element = 10;

newNode.next = A.next;

A.next = newNode;

```

g. حذف العقدة التي تحتوي على العنصر 23 وإعادة الذاكرة المخصصة لهذه العقدة:

```java

Node previousNode = findPreviousNode(list, 23);

if (previousNode != null) {

Node nodeToDelete = previousNode.next;

previousNode.next = nodeToDelete.next;

nodeToDelete.next = null; // إعادة الذاكرة المخصصة

}

```

يرجى ملاحظة أنه يجب تنفيذ بعض الوظائف المساعدة مثل `findNodeWithValue()` و `getLastNode()` و `findPreviousNode()` لتحقيق الوظائف المذكورة. هذه الوظائف تعتمد على النظام الخاص بك والتنظيم الداخلي للقائمة المتصلة الخاصة بك.

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

p = list;

while (p != NULL){

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

p = p.getNext(); }

18

32

23

16

43

87

25

44

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());

10

18

13

* + - 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(); }

30

42

20

28

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?

18

38

2

15

45

25

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



1-Node<integer>newest=newNode<>(20,first.next,last.prer);

2-firset.next.setNext(newest);

3-Last.prev.set.prev(newest);

4-connt ++;

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.

يمكنك استخدام الشفرة التالية لتنفيذ الطريقة `sum` في فئة `SingleLinkedList`:

```java

public int sum(Node<Integer> list) {

int sum = 0;

Node<Integer> current = list;

while (current != null) {

sum += current.getElement();

current = current.getNext();

}

return sum;

}

```

ثم يمكنك اختبار الطريقة باستخدام القائمة المتصلة التي تم توفيرها في المثال:

```java

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

Node<Integer> node1 = new Node<>(45);

Node<Integer> node2 = new Node<>(65);

Node<Integer> node3 = new Node<>(85);

list.setNext(node1);

node1.setNext(node2);

node2.setNext(node3);

int result = sum(list);

System.out.println(result); // الناتج: 220

```

في هذا المثال، تم إنشاء قائمة متصلة تحتوي على الأعداد 25 و 45 و 65 و 85. تم استدعاء الطريقة `sum` مع القائمة كوسيط، ومن ثم تم طباعة النتيجة التي يجب أن تكون 220.

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}.

يمكنك استخدام الشفرة التالية لتنفيذ الطريقة `removeLast` في فئة `DoublyLinkedList`:

```java

public E removeLast(Node<E> list) {

Node<E> current = list;

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

current = current.getNext();

}

Node<E> previous = current.getPrevious();

previous.setNext(null);

return current.getElement();

}

```

تحتاج هذه الطريقة إلى قائمة متصلة مزدوجة تحتوي على عقدتين على الأقل كشرط مسبق. ستقوم الطريقة بإزالة العقدة الأخيرة في القائمة المتصلة المزدوجة.

ثم يمكنك اختبار الطريقة باستخدام القائمة المتصلة المزدوجة التي تم توفيرها في المثال:

```java

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

Node<Integer> node1 = new Node<>(44);

Node<Integer> node2 = new Node<>(66);

Node<Integer> node3 = new Node<>(88);

list.setNext(node1);

node1.setPrevious(list);

node1.setNext(node2);

node2.setPrevious(node1);

node2.setNext(node3);

node3.setPrevious(node2);

Integer removedElement = removeLast(list);

System.out.println("Removed Element: " + removedElement); // الناتج: 88

System.out.println("Updated List: " + list); // الناتج: {22, 44, 66}

```

في هذا المثال، تم إنشاء قائمة متصلة مزدوجة تحتوي على الأعداد 22 و 44 و 66 و 88. تم استدعاء الطريقة `removeLast` مع القائمة كوسيط، ومن ثم تم طباعة العنصر المحذوف والقائمة المتصلة المحدثة. يجب أن تظهر القائمة المحدثة بعد إزالة العقدة الأخيرة كـ {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.

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

Node<Integer> node1 = new Node<>(33);

Node<Integer> node2 = new Node<>(44);

Node<Integer> node3 = new Node<>(55);

list1.setNext(node1);

node1.setNext(node2);

node2.setNext(node3);

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

Node<Integer> node4 = new Node<>(77);

Node<Integer> node5 = new Node<>(88);

Node<Integer> node6 = new Node<>(99);

list2.setNext(node4);

node4.setNext(node5);

node5.setNext(node6);

append(list1, list2);

Node<Integer> current = list1;

while (current != null) {

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

current = current.getNext();

}

// Output: 22 33 44 55 66 77 88 99

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.

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

Node<E> current = list1;

Node<E> newList = null;

Node<E> newListCurrent = null;

while (current != null) {

Node<E> newNode = new Node<>(current.getElement());

if (newList == null) {

newList = newNode;

newListCurrent = newList;

} else {

newListCurrent.setNext(newNode);

newListCurrent = newListCurrent.getNext();

}

current = current.getNext();

}

current = list2;

while (current != null) {

Node<E> newNode = new Node<>(current.getElement());

newListCurrent.setNext(newNode);

newListCurrent = newListCurrent.getNext();

current = current.getNext();

}

return newList;

}

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

Node<Integer> node1 = new Node<>(33);

Node<Integer> node2 = new Node<>(44);

Node<Integer> node3 = new Node<>(55);

list1.setNext(node1);

node1.setNext(node2);

node2.setNext(node3);

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

Node<Integer> node4 = new Node<>(77);

Node<Integer> node5 = new Node<>(88);

Node<Integer> node6 = new Node<>(99);

list2.setNext(node4);

node4.setNext(node5);

node5.setNext(node6);

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

Node<Integer> current = concatenatedList;

while (current != null) {

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

current = current.getNext();

}

// Output: 22 33 44 55 66 77 88 99

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}.

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

Node<E> current = list1;

Node<E> newList = null;

Node<E> newListCurrent = null;

while (current != null) {

Node<E> newNode = new Node<>(current.getElement());

if (newList == null) {

newList = newNode;

newListCurrent = newList;

} else {

newListCurrent.setNext(newNode);

newListCurrent = newListCurrent.getNext();

}

current = current.getNext();

}

current = list2;

while (current != null) {

Node<E> newNode = new Node<>(current.getElement());

newListCurrent.setNext(newNode);

newListCurrent = newListCurrent.getNext();

current = current.getNext();

}

return newList;

}

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

Node<Integer> node1 = new Node<>(33);

Node<Integer> node2 = new Node<>(44);

Node<Integer> node3 = new Node<>(55);

list1.setNext(node1);

node1.setNext(node2);

node2.setNext(node3);

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

Node<Integer> node4 = new Node<>(77);

Node<Integer> node5 = new Node<>(88);

Node<Integer> node6 = new Node<>(99);

list2.setNext(node4);

node4.setNext(node5);

node5.setNext(node6);

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

Node<Integer> current = concatenatedList;

while (current != null) {

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

current = current.getNext();

}

// Output: 22 33 44 55 66 77 88 99

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

**@Override**

**public boolean equals(Object obj) {**

**if (this == obj) {**

**return true; // Same object reference**

**}**

**if (!(obj instanceof DoublyLinkedList)) {**

**return false; // Different classes**

**}**

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

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

**return false; // Different sizes**

**}**

**Node<E> currentNode = this.first;**

**Node<E> otherNode = otherList.first;**

**while (currentNode != null) {**

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

**return false; // Different elements**

**}**

**currentNode = currentNode.getNext();**

**otherNode = otherNode.getNext();**

**}**

**return true; // All elements are equal**

**}**

1. Implement the rotate() methode in CircularLinkedList class.

**public void rotate() {**

**if (isEmpty() || size() == 1) {**

**return; // لا يوجد عنصر أو عنصر واحد فقط في القائمة المتدورة**

**}**

**Node<E> newLast = first; // العقدة الجديدة التي ستصبح العقدة الأخيرة**

**first = first.getNext(); // تحديث العقدة الأولى لتصبح العقدة التالية**

**last = last.getNext(); // تحديث العقدة الأخيرة لتصبح العقدة التالية**

**newLast.setNext(null); // فصل العقدة الجديدة من القائمة المتدورة**

**last.setNext(first); // ربط العقدة الأخيرة الجديدة بالعقدة الأولى لإكمال الدورة**

**last = newLast; // تحديث العقدة الأخيرة لتصبح العقدة الجديدة**

**}**

1. Implement the addFirst() method in CircularLinkedList class.

**public void addFirst(E element) {**

**Node<E> newNode = new Node<>(element); // إنشاء عقدة جديدة**

**if (isEmpty()) {**

**first = newNode; // في حالة عدم وجود عناصر في القائمة المتدورة، يتم جعل العقدة الجديدة العقدة الأولى والعقدة الأخيرة**

**last = newNode;**

**newNode.setNext(newNode); // العقدة الوحيدة تشير إلى نفسها**

**} else {**

**newNode.setNext(first); // تعيين الرابط القادم للعقدة الجديدة لتكون العقدة الأولى الحالية**

**first = newNode; // تحديث العقدة الأولى لتصبح العقدة الجديدة**

**last.setNext(first); // ربط العقدة الأخيرة بالعقدة الأولى لإكمال الدورة**

**}**

**}**