**Assignment**

Q1\

المشكلة في هذا التعريف هي أنه يحاول إنشاء كائن من الفئة Arrays باستخدام الكلمة الرئيسية new. ولكن، الفئة Arrays في جافا هي فئة واجهة (interface) وليست فئة عادية يمكن إنشاء كائنات منها مباشرة.

لإنشاء مصفوفة جديدة، يجب استخدام الكلمة الرئيسية new مع نوع المصفوفة المطلوبة وعدد العناصر فيها. على سبيل المثال، لإنشاء مصفوفة من الصف العادي (class) يُمكن استخدام التعبير التالي:

```java

int[] array = new int[5];

```

وهنا يتم إنشاء مصفوفة جديدة من النوع int بحجم 5.

Q2\

يمكننا تنفيذ الطلب باستخدام الكود التالي:

public class ArrayReverse {

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);  
 }  
  
 public static void reverse(int[] a) {  
 int start = 0;  
 int end = a.length - 1;  
  
 while (start < end) {   
 int temp = a[start];  
 a[start] = a[end];  
 a[end] = temp;  
  
 start++;  
 end--;  
 }  
 }  
  
 public static void printArray(int[] a) {  
 for (int num : a) {  
 System.*out*.print(num + " ");  
 }  
 System.*out*.println();  
 }  
}

هذا الكود يتضمن طريقة `reverse` التي تقوم بعكس عناصر المصفوفة الممررة إليها. تم إنشاء مصفوفة بسيطة في الطريقة `main` وتم طباعة المصفوفة الأصلية وبعد عكس العناصر باستخدام الطريقة `reverse`، يتم طباعة المصفوفة المعكوسة.

Q3\

من المهم أن نلاحظ أن القوائم المتسلسلة والمصفوفات لها ميزات واستخدامات مختلفة، ولذلك تُستخدم المصفوفات إلى جانب القوائم المتسلسلة. إليك بعض الأسباب التي تفسر لماذا يتم استخدام المصفوفات:

الوصول العشوائي: توفر المصفوفات إمكانية الوصول العشوائي للعناصر بزمن ثابت، وهذا يعني أنه يمكن الوصول مباشرة إلى أي عنصر في المصفوفة باستخدام الفهرس. يتيح ذلك الوصول الفعال والسريع إلى العناصر عندما يكون موقعها معروفًا.

كفاءة الذاكرة: تُعد المصفوفات عمومًا أكثر كفاءة من حيث استخدام الذاكرة مقارنة بالقوائم المتسلسلة. فالقوائم المتسلسلة تحتاج إلى ذاكرة إضافية لتخزين المؤشر التالي لكل عنصر، بينما تخزن المصفوفات فقط العناصر ذاتها. يمكن أن يكون ذلك مهمًا في الحالات التي تهم استخدام الذاكرة بشكل كبير.

تخصيص ذاكرة متسلسلة: تُخزن المصفوفات في مساحات ذاكرة متسلسلة، وهذا يمكن أن يحسِّن أداء ذاكرة التخزين المؤقت ويقلل من تجزئة الذاكرة. وبالتالي، فإن المصفوفات تكون أكثر كفاءة في الوصول إلى الذاكرة، ويمكن أن تؤدي إلى أداء أفضل في بعض العمليات.

كفاءة التكرار: تكون المصفوفات فعالة في التكرار على العناصر بترتيبها، وخاصة عندما يكون التكرار في الاتجاه الأمامي ولا يتطلب إدخالًا أو حذفًا متكررًا. أما القوائم المتسلسلة، فتتطلب تمرير القائمة من البداية للوصول إلى عنصر محدد، وهذا يستغرق وقتًا خطيًا.

تنفيذ مبسط: تتمتع المصفوفات بتنفيذ أبسط بالمقارنة مع القوائم المتسلسلة. فهي تُعتبر هياكل بيانات أساسية في العديد من لغات البرمجة، وتوفر الدعم المدمج والتحسينات. ومن ناحية أخرى، تتطلب القوائم المتسلسلة هياكل بيانات أكثر تعقيدًا وتخصيص ذاكرة إضافية لك

Q4\

a. False

b. False

c. False

d. True

e. False

Q5\

بدون معرفة تفاصيل تنفيذ الشيفرة المحددة، لا يمكن تحديد النتائج الدقيقة لكل عبارة. ومع ذلك، يمكنني شرح السلوك المتوقع والنتائج المحتملة بناءً على السيناريوهات الشائعة:

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

تقوم هذه العبارة بطباعة قيمة العنصر في العقدة الحالية لكائن `18`.

ب. `System.out.println(A.getElement());`

تقوم هذه العبارة بطباعة قيمة العنصر في كائن `32`.

ج. `System.out.println(B.getNext().getElement());`

تقوم هذه العبارة بالحصول على العقدة التالية بعد `B`، ثم تقوم بطباعة قيمة العنصر في تلك العقدة. 25

د. `System.out.println(list.getNext().getNext().getElement());`

تقوم هذه العبارة بالحصول على العقدة التالية بعد العقدة الحالية لكائن `list`، ثم تحصل على العقدة التالية بعد ذلك، وأخيرًا تقوم بطباعة قيمة العنصر في تلك العقدة.23

النتائج الفعلية لكل عبارة ستعتمد على المحتوى والهيكل المحدد للقائمة المتصلة، بالإضافة إلى تنفيذ الأساليب المقابلة (`getElement()`، `getNext()`، وما إلى ذلك).

Q7\

A = list.findNode(23);  
  
  
list = new LinkedList();  
list.insertAtBeginning(16);  
  
  
B = list.getLastNode();  
  
  
list = new LinkedList();  
  
  
Node node25 = list.findNode(25);  
if (node25 != null) {  
node25.setElement(35);  
}  
  
Node node10 = new Node(10);  
list.insertAfter(A, node10);  
  
list.deleteNode(23);

Q8\

الشيفرة المذكورة تستخدم حلقة while للانتقال عبر العقد في القائمة المتصلة وطباعة قيمة العنصر في كل عقدة.

النتيجة المتوقعة هي طباعة قيم العناصر في القائمة المتصلة حتى تصل الحلقة إلى العقدة النهائية (التي يتم تمثيلها بـ `NULL`).

مثال على النتيجة المتوقعة:

Element1  
Element2  
Element3

قيمة العنصر في العقدة الأخيرة

Q9\

مخرجات :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());

النتيجة المتوقعة:

10

18

13

مخرجات :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();  
}

النتيجة المتوقعة:

30

20

42

Q10\

بناءً على الشيفرة الجافا المقدمة، وبفرض أن فئة `SingleLinkedList` مُنفذة بشكل صحيح، يتوقع أن نتيجة تنفيذ العمليات المذكورة واستدعاء الطريقة `list.print()` ستكون كالتالي:

```

18 -> 38 -> 2 -> 15 -> 45 -> 25

```

هذه النتيجة تُمثل العناصر في القائمة المتصلة بعد تنفيذ سلسلة العمليات. يتم فصل كل عنصر بواسطة سهم (`->`). يُرجى ملاحظة أن العناصر تُطبع في النظام الذي تمت به إضافتها إلى القائمة.

يرجى ملاحظة أن هذا يفترض أن فئة `SingleLinkedList` مُنفذة وفقًا للمعايير المذكورة في المحاضرات وأن الطرق `addFirst()` و `addLast()` و `removeNode()` و `print()` مُنفذة بشكل صحيح.

Q11\

لإدراج القيمة 20 بين القيم 15 و 24 في قائمة متصلة مزدوجة، ستحتاج إلى إيجاد العقدتين التي تمثل هاتين القيمتين وتنفيذ التحديثات اللازمة لربط العقدة الجديدة بشكل صحيح. فيما يلي مثال لشيفرة جافا توضح كيفية تحقيق ذلك:

```java

// نفترض أن لدينا فئة DoublyLinkedList مع الطرق المناسبة

// العثور على العقدتين ذات القيم 15 و 24

Node<Integer> node15 = list.findNode(15);

Node<Integer> node24 = list.findNode(24);

// إنشاء العقدة الجديدة بقيمة 20

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

// تحديث الروابط لإدراج العقدة الجديدة

newNode.setNext(node24);

newNode.setPrevious(node15);

node15.setNext(newNode);

node24.setPrevious(newNode);

```

في هذا المقتطف من الشيفرة، `list` يشير إلى مثيل من فئة `DoublyLinkedList` الذي يمثل القائمة المتصلة المزدوجة المعطاة. نفترض أن فئة `DoublyLinkedList` تحتوي على طرق مثل `findNode(value)` للبحث عن عقدة بقيمة محددة وطرق ضبط مناسبة للمراجع `next` و `previous` في فئة `Node`.

بتنفيذ الشيفرة أعلاه، سيتم إدراج عقدة جديدة بقيمة 20 بين العقدتين ذات القيم 15 و 24 في القائمة المتصلة المزدوجة. ستعكس القائمة المتصلة الناتجة هذا الإدراج الجديد.

Q12\

لإدراج القيمة 20 بين القيمتين 15 و 24 في قائمة الروابط المزدوجة، يمكن استخدام الخطوات التالية في الكود:

1. تحديد العقدة التي تحتوي على القيمة 15 كنقطة البداية للبحث.

2. قم بالتحرك عبر القائمة بواسطة الاستدلال على العقدة التالية بشكل متكرر حتى تصل إلى العقدة التي تحتوي على القيمة 24.

3. عندما تصل إلى العقدة التي تحتوي على القيمة 24، أنشئ عقدة جديدة تحتوي على القيمة 20.

4. قم بتحديد العقدة التي تحتوي على القيمة 15 كعقدة السابقة للعقدة الجديدة.

5. قم بتحديد العقدة التي تحتوي على القيمة 24 كعقدة التالية للعقدة الجديدة.

6. قم بتحديث الاستدلال للعقدة الجديدة كعقدة التالية للعقدة التي تحتوي على القيمة 15.

7. قم بتحديث الاستدلال للعقدة التي تحتوي على القيمة 24 كعقدة السابقة للعقدة الجديدة.

فيما يلي مثال لكيفية تنفيذ هذه الخطوات في الكود بلغة Java:

Node<Integer> currentNode = head;  
while (currentNode != null && currentNode.getData() != 15) {  
currentNode = currentNode.getNext();  
}  
  
if (currentNode != null) {  
Node<Integer> newNode = new Node<>(20);  
newNode.setPrev(currentNode);  
  
newNode.setNext(currentNode.getNext());  
  
currentNode.setNext(newNode);  
  
if (newNode.getNext() != null) {  
newNode.getNext().setPrev(newNode);  
}  
}

يرجى ملاحظة أنه يجب أن تكون لديك فئة `Node` وفئة `DoublyLinkedList` لتنفيذ القائمة المزدوجة. ويجب أن يكون لديك متغير `head` يشير إلى رأس القائمة المزدوجة.

Q13\

فيما يلي مثال لتنفيذ طريقة `removeLast` في فئة `DoublyLinkedList` التي تقوم بحذف آخر عقدة من القائمة المعطاة:

public E removeLast(Node<E> list) {  
 Node<E> current = list;  
  
 while (current.getNext().getNext() != null) {  
 current = current.getNext();  
 }  
  
 Node<E> lastNode = current.getNext();  
 E removedData = lastNode.getData();  
  
 current.setNext(null);  
  
 return removedData;

في هذه الطريقة، نبدأ بتهيئة المتغير `current` ليشير إلى العقدة الأولى في القائمة. نستخدم حلقة `while` للتكرار عبر القائمة حتى نصل إلى العقدة قبل الأخيرة. يتم ذلك عن طريق التحقق مما إذا كان `current.getNext().getNext()` يساوي `null`. تتوقف الحلقة عندما تصبح العقدة التالية بعد `current` هي العقدة الأخيرة.

بمجرد الوصول إلى العقدة قبل الأخيرة، نخزن مرجعًا إلى العقدة الأخيرة (`current.getNext()`) في متغير يسمى `lastNode`. كما نخزن بيانات العقدة الأخيرة في متغير يسمى `removedData`.

لحذف العقدة الأخيرة، نضبط المرجع `next` في العقدة قبل الأخيرة (`current`) على `null`، مما يفصلها عن العقدة الأخيرة.

أخيرًا، نعيد قيمة `removedData`.

لاختبار طريقة `removeLast` يمكنك إنشاء قائمة مثالية واستدعاء الطريقة، مثل هذا:

DoublyLinkedList<Integer> list = new DoublyLinkedList<>();  
list.addLast(22);  
list.addLast(44);  
list.addLast(66);  
list.addLast(88);  
  
Integer removedData = list.removeLast(list.getHead());  
System.out.println("Removed Data: " + removedData);  
System.out.println("Updated List: " + list);

```

سيتم إنشاء قائمة بالقيم {22, 44, 66, 88} واستدعاء طريقة `removeLast`. سيتم عرض النتيجة على الشاشة بوصفها "Removed Data: 88" و "Updated List: {22, 44, 66}".

Q14\

لعكس قائمة ربط من نوع "singly linked list" باستخدام كمية ثابتة من الذاكرة الإضافية، يمكنك اتباع الخطوات التالية:

قم بتهيئة ثلاثة مؤشرات: المؤشر "previous" ليشير إلى null، والمؤشر "current" ليشير إلى رأس القائمة "L"، والمؤشر "next" ليشير إلى null.

قم بتكرار القائمة مع تحديث المؤشرات على النحو التالي:

قم بتخزين العقدة التالية للعقدة الحالية في المؤشر "next".

ضع المؤشر "next" للعقدة الحالية بحيث يشير إلى العقدة السابقة "previous".

قم بتحديث المؤشر "previous" ليشير إلى العقدة الحالية.

قم بتحديث المؤشر "current" ليشير إلى العقدة التالية "next".

بعد الانتهاء من التكرار، سيكون المؤشر "previous" يشير إلى آخر عقدة في القائمة الأصلية "L" وهي ستكون رأس القائمة المعكوسة.

قم بتحديث رأس القائمة "L" ليكون يشير إلى العقدة التي يشير إليها المؤشر "previous".

الآن تم عكس القائمة وأصبحت القائمة "L" معكوسة.

فيما يلي توضيح خطوة بخطوة للخوارزمية:

Initial state: L: 1 -> 2 -> 3 -> 4 -> null  
  
Step 1:  
previous: null  
current: 1 -> 2 -> 3 -> 4 -> null  
next: null  
  
Step 2:  
previous: null  
current: 1 -> 2 -> 3 -> 4 -> null  
next: 2 -> 3 -> 4 -> null  
1 -> null  
  
Step 3:  
previous: 1 -> null  
current: 2 -> 3 -> 4 -> null  
next: 2 -> 3 -> 4 -> null  
1 -> null  
  
Step 4:  
previous: 2 -> 1 -> null  
current: 3 -> 4 -> null  
next: 3 -> 4 -> null  
2 -> 1 -> null  
  
Step 5:  
previous: 3 -> 2 -> 1 -> null  
current: 4 -> null  
next: 4 -> null  
3 -> 2 -> 1 -> null  
  
Step 6:  
previous: 4 -> 3 -> 2 -> 1 -> null  
current: null  
next: null  
4 -> 3 -> 2 -> 1 -> null  
  
Final state: L: 4 -> 3 -> 2 -> 1 -> null

العقدة السابقة. نظرًا لاستخدام كمية ثابتة من الذاكرة الإضافية (ثلاثة مؤشرات فقط)، فإن الخوارزمية لديها تعقيد مساحي يبلغ O(1). تعقيد الوقت للخوارزمية هو O(nلعكس قائمة مرتبطة بشكل مفرد (Singly Linked List) باستخدام كمية ثابتة من الذاكرة الإضافية، يمكن اتباع الخوارزمية التالية:

قم بتهيئة ثلاثة مؤشرات: المؤشر "previous" الذي يشير إلى null، المؤشر "current" الذي يشير إلى رأس القائمة "L"، والمؤشر "next" الذي يشير إلى null.

قم بتكرار القائمة وقم بتحديث المؤشرات على النحو التالي:

قم بتخزين العقدة التالية للعقدة الحالية في المؤشر "next".

ضع المؤشر "next" للعقدة الحالية بحيث يشير إلى العقدة السابقة "previous".

قم بتحديث المؤشر "previous" ليشير إلى العقدة الحالية.

قم بتحديث المؤشر "current" ليشير إلى العقدة التالية "next".

بعد الانتهاء من التكرار، ستكون المؤشر "previous" يشير إلى آخر عقدة في القائمة الأصلية "L" والتي ستصبح رأس القائمة المعكوسة.

قم بتحديث رأس القائمة "L" ليشير إلى العقدة التي يشير إليها المؤشر "previous".

تمت عملية العكس وأصبحت القائمة "L" معكوسة.

Q15\

لتنفيذ طريقة equals() لفئة DoublyLinkedList، يجب عليك مقارنة قائمتين متصلتين مزدوجتين والتحقق مما إذا كانت لديهما نفس العناصر بنفس الترتيب. فيما يلي مثال لتنفيذ الطريقة باستخدام لغة جافا:

public class DoublyLinkedList<T> {  
 private Node<T> head;  
 private Node<T> tail;  
 private int size;

public boolean equals(DoublyLinkedList<T> otherList) {  
 if (otherList == this) {  
 return true;  
 }  
  
 if (size != otherList.size) {  
 return false;  
 }  
  
 Node<T> currentNode = head;  
 Node<T> otherCurrentNode = otherList.head;  
  
 while (currentNode != null) {  
 if (!currentNode.data.equals(otherCurrentNode.data)) {  
 return false;  
 }  
  
 currentNode = currentNode.next;  
 otherCurrentNode = otherCurrentNode.next;  
 }  
  
 return true;  
 }  
  
 private static class Node<T> {  
 T data;  
 Node<T> prev;  
 Node<T> next;  
  
 Node(T data) {  
 this.data = data;  
 this.prev = null;  
 this.next = null;  
 }  
 }  
}

في طريقة equals():

نتحقق أولاً مما إذا كانت القوائم المراد مقارنتها هي نفس الكائن عن طريق مقارنة مراجعهما. إذا كانت هما نفس الكائن، نعيد true.

ثم نقارن أحجام القوائم المراد مقارنتها. إذا كانت مختلفة، نعيد false لأنها لا يمكن أن تكون متساوية.

نكرر القوائم المراد مقارنتها متزامنة، مقارنين بيانات كل عقدة. إذا كانت هناك أي عقدتين متوافقتين لهما بيانات مختلفة، نعيد false.

إذا وصلنا إلى نهاية التكرار دون العثور على أي اختلافات، نعيد true لأن القوائم متساوية.