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. 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

40

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.

// Assuming the existence of the Node class with the appropriate methods and attributes.

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

Node A = null; // Initialize A to null

Node currentNode = list; // Start with the head of the list

while (currentNode != null) {

if (currentNode.getElement() == 23) {

A = currentNode; // Found the node with element 23, assign it to A

break;

}

currentNode = currentNode.getNext(); // Move to the next node

}

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

currentNode = list; // Start with the head of the list

while (currentNode != null) {

if (currentNode.getElement() == 16) {

list = currentNode; // Found the node with element 16, assign it to list

break;

}

currentNode = currentNode.getNext(); // Move to the next node

}

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

Node B = list; // Start with the head of the list

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

B = B.getNext(); // Move to the next node until B reaches the last node

}

// d- Make list point to an empty list.

list = null; // Set list to null, effectively emptying the list

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

currentNode = list; // Start with the head of the list

while (currentNode != null) {

if (currentNode.getElement() == 25) {

currentNode.setElement(35); // Found the node with element 25, update its value to 35

break;

}

currentNode = currentNode.getNext(); // Move to the next node

}

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

Node newNode = new Node(10); // Create a new node with element 10

if (A != null) {

newNode.setNext(A.getNext()); // Set the next pointer of the new node to the next node of A

A.setNext(newNode); // Set the next pointer of A to point to the new node

}

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

if (list != null && list.getElement() == 23) {

list = list.getNext(); // If the head of the list has element 23, move the head to the next node

} else {

currentNode = list; // Start with the head of the list

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

if (currentNode.getNext().getElement() == 23) {

currentNode.setNext(currentNode.getNext().getNext()); // Skip the node with element 23

break;

}

currentNode = currentNode.getNext(); // Move to the next node

}

}

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

* 1. System.out.println( list.getElement()); 18
  2. System.out.println( A. getElement()); 32
  3. System.out.println( B.getNext().getElement()); 25
  4. System.out.println( list.getNext().getNext().getElement()); 23

**5-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. **False**
  2. In a linked list, memory allocated for the nodes is sequential. **False**
  3. A single linked list can be traversed in either direction. **False**
  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. **True**
  5. The head pointer of a linked list cannot be used to traverse the list. **False**

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

القوائم المرتبطة والمصفوفات لها خصائص ومفاضلات مختلفة، مما يجعلها مناسبة لسيناريوهات مختلفة. أثناء وجود قوائم مرتبطة لها مزايا معينة، لا تزال الصفائف تستخدم على نطاق واسع لعدة أسباب: 1. الوصول العشوائي: توفر الصفائف الوصول العشوائي المستمر إلى العناصر. يمكنك الوصول إلى أي عنصر في صفيف مباشرة بواسطة فهرسها. في المقابل، تتطلب القوائم المرتبطة عبور من الرأس إلى الموضع المرغوب، والذي يأخذ وقتا خطيا. إذا كانت عمليات الوصول العشوائي المتكرر أو الفهرسة هي متطلبات أساسية، فهي صفائف أكثر كفاءة. 2. الذاكرة المتجاورة: مخصصات المصفوفات عناصر في مواقع الذاكرة المتجاورة. يسمح تخطيط الذاكرة هذا بتخزين مؤلف وحدة المعالجة المركزية فعالة ويمكن أن يؤدي إلى أداء أفضل في السيناريوهات حيث تكون محلية الذاكرة حاسمة، مثل التكرار على الصفيف بأكملها أو أداء عمليات مكثفة على حسابيا. القوائم المرتبطة، من ناحية أخرى، تخزين العناصر في العقد المنفصلة مع المؤشرات، والتي يمكن أن تؤدي إلى أنماط وصول الذاكرة المتناثرة وانخفاض كفاءة ذاكرة التخزين المؤقت. 3. كفاءة الذاكرة: تتطلب القوائم المرتبطة ذاكرة إضافية لتخزين المؤشرات / المراجع التي ترتبط العقد معا. في المقابل، يتمتع المصفوفات بتصميم ذاكرة أكثر إحكاما نظرا لأنهم يخزنون فقط العناصر نفسها. إذا كان استخدام الذاكرة مصدر قلق، فيمكن أن تكون الصفائف أكثر كفاءة في الذاكرة. 4. التسلسل والاستثمار: غالبا ما تكون المصفوفات أسهل لتسلسل وتخزينها ككتلة واحدة في الذاكرة أو على القرص. تتطلب القوائم المرتبطة، مع طبيعتها الديناميكية ومواقع الذاكرة المتناثرة، تقنيات التسلسل أكثر تعقيدا. 5. البساطة والقدرة على التنبؤ: يتمتع المصفوفات بنية بيانات أبسط ويمكن التنبؤ بها مقارنة بالقوائم المرتبطة. لديهم حجم ثابت يحدد في وقت الإبداع وتقديم عمليات الفهرسة مباشرة. القوائم المرتبطة، مع طبيعتها والمؤشرات الديناميكية، وإدخال تعقيد إضافي من حيث إدارة الذاكرة والاجتياز. 6. دعم اللغة والمكتبة: المصفوفات هي هياكل بيانات أساسية تدعمها جميع لغات البرمجة والمكتبات تقريبا. لديهم دعم واسع المستوى أو مستوى لغة للمكتبة للعمليات مثل الفرز والبحث والتلاعب. قد تحتوي القوائم المرتبطة، في حين أنها لا تزال مدعومة، وظيفة مدمجة محدودة أو تتطلب تطبيقات مخصصة إضافية. 7. قابلية التشغيل البيني: غالبا ما تستخدم المصفوفات عند التفاعل مع الأنظمة أو المكتبات الأخرى التي تتوقع بيانات بتنسيق الصفيف.

7-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 class DoublyLinkedList<E> {

// ... other class members ...

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

if (list == null) {

throw new IllegalArgumentException("List cannot be null.");

}

if (i < 0 || j < 0) {

throw new IllegalArgumentException("Indices should be non-negative.");

}

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

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

if (nodeA == null || nodeB == null) {

throw new IllegalArgumentException("Invalid indices.");

}

// Swap the elements

E temp = nodeA.data;

nodeA.data = nodeB.data;

nodeB.data = temp;

}

// Helper method to get a node at a given index

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

int currentIndex = 0;

Node<E> currentNode = list;

while (currentNode != null) {

if (currentIndex == index) {

return currentNode;

}

currentNode = currentNode.next;

currentIndex++;

}

return null;

}

// ... other class members ...

private static class Node<E> {

E data;

Node<E> next;

Node<E> prev;

public Node(E data) {

this.data = data;

}

}

}

public class Main {

public static void main(String[] args) {

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("Original List: " + list);

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

System.out.println("Swapped List: " + list);

}

Output:

Original List: {22, 33, 44, 55, 66, 77, 88, 99}

Swapped List: {22, 33, 77, 55, 66, 44, 88, 99}

8-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 class SingleLinkedList<E> {

private Node<E> head;

private Node<E> tail;

// ... other class members ...

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

if (list1 == null) {

return copyList(list2);

}

Node<E> newList = copyList(list1);

Node<E> current = newList;

while (current.next != null) {

current = current.next;

}

current.next = copyList(list2);

return newList;

}

// Helper method to create a new list as a copy of an existing list

private Node<E> copyList(Node<E> list) {

if (list == null) {

return null;

}

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

Node<E> current = newList;

Node<E> currentList = list.next;

while (currentList != null) {

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

current = current.next;

currentList = currentList.next;

}

return newList;

}

// ... other class members ...

private static class Node<E> {

E data;

Node<E> next;

public Node(E data) {

this.data = data;

}

}

}

public class Main {

public static void main(String[] args) {

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

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

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

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

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

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

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

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

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

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

System.out.print("Concatenated List: ");

printList(concatenatedList);

}

// Helper method to print the elements of a list

private static void printList(SingleLinkedList.Node<Integer> list) {

while (list != null) {

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

list = list.next;

}

System.out.println();

}

}

Output:

Concatenated List: 22 33 44 55 66 77 88 99

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

**public class DoublyLinkedList<E> {**

**private Node<E> head;**

**private Node<E> tail;**

**// ... other class members ...**

**@Override**

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

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

**return true;**

**}**

**if (obj == null || getClass() != obj.getClass()) {**

**return false;**

**}**

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

**Node<E> currentNode = head;**

**Node<?> otherCurrentNode = otherList.head;**

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

**if (!currentNode.data.equals(otherCurrentNode.data)) {**

**return false;**

**}**

**currentNode = currentNode.next;**

**otherCurrentNode = otherCurrentNode.next;**

**}**

**return currentNode == null && otherCurrentNode == null;**

**}**

**// ... other class members ...**

**private static class Node<E> {**

**E data;**

**Node<E> prev;**

**Node<E> next;**

**public Node(E data) {**

**this.data = data;**

**}**

**}**

**}**

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

**public class CircularLinkedList<E> {**

**private Node<E> head;**

**private Node<E> tail;**

**// ... other class members ...**

**public void rotate() {**

**if (head == null || head == tail) {**

**// Empty list or only one element, no rotation needed**

**return;**

**}**

**tail.next = head; // Make the last node point to the head**

**head = head.next; // Update the head**

**tail = tail.next; // Update the tail**

**// Update the tail's next reference to point to the new head**

**tail.next = head;**

**}**

**// ... other class members ...**

**private static class Node<E> {**

**E data;**

**Node<E> next;**

**public Node(E data) {**

**this.data = data;**

**}**

**}**

**}**

11-Implement the addFirst() method in CircularLinkedList class

**public class CircularLinkedList<E> {**

**private Node<E> head;**

**private Node<E> tail;**

**// ... other class members ...**

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

**Node<E> newNode = new Node<>(data);**

**if (head == null) {**

**// If the list is empty, make the new node the head and tail**

**head = newNode;**

**tail = newNode;**

**tail.next = head; // Close the circular structure**

**} else {**

**newNode.next = head; // Make the new node point to the current head**

**head = newNode; // Update the head reference**

**tail.next = head; // Update the tail's next reference to the new head**

**}**

**}**

**// ... other class members ...**

**private static class Node<E> {**

**E data;**

**Node<E> next;**

**public Node(E data) {**

**this.data = data;**

**}**

**}**

**}**

12-What’s wrong with this definition:

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

**Int[] array = new int[10];**

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

* 1. list. getElement() >= 18 true
  2. list.getNext() == A true
  3. A.getNext().getElement() == 16 false
  4. B.getNext() == (NULL) false
  5. list. getElement() == 18 true

14-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 20 42