1. What’s wrong with this definition:

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

1. For a single-dimensional array of integers:

int[] arrays = new int[10];

1. For a two-dimensional array of strings:

String[][] arrays = new String[5][5];

1. For an array using the Array class:

Object arrays = Array.newInstance(Object.class, 10);

1. Write and test this method:

void reverse(int[] a)

// reverses the elements of a[]

public class ArrayReverse {

public static 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 index one step forward

start++;

// Move the end index one step backward

end--;

}

}

public static void main(String[] args) {

// Test the reverse method

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

System.out.println("Original Array: " + Arrays.toString(arr));

reverse(arr);

System.out.println("Reversed Array: " + Arrays.toString(arr));

}

}

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

**Arrays are still used in various scenarios despite the advantages of linked lists. Here are some reasons why arrays are used:**

**1. Random Access: Arrays provide constant-time random access to elements based on their indices. This allows for efficient retrieval and modification of elements at specific positions. Linked lists, on the other hand, require traversing the list from the beginning to access an element at a specific position, which takes linear time.**

**2. Memory Efficiency: Arrays have a more memory-efficient representation compared to linked lists. Linked lists require additional memory to store the links between nodes, while arrays store only the elements themselves. This can result in better memory utilization, especially when dealing with large data sets.**

**3. Sequential Access Efficiency: Arrays are efficient for iterating over all elements sequentially. The contiguous memory layout of arrays allows for better cache utilization and faster iteration compared to linked lists, where elements are scattered in different memory locations.**

**4. Simplicity and Ease of Use: Arrays have a straightforward and intuitive structure that is easy to understand, implement, and work with. They have a fixed size, which can simplify certain scenarios where a fixed-size collection is required.**

**5. Performance Considerations: Arrays can offer better performance in cases where the size of the collection is known in advance and doesn't change frequently. Arrays have better locality of reference and cache performance due to their contiguous memory layout. Linked lists, on the other hand, excel in scenarios where frequent insertion and deletion of elements are required.**

**6. Language Limitations: Some programming languages have limited or no built-in support for linked lists, making arrays the default choice for storing and manipulating collections of elements.**

**7. Interoperability and Compatibility: Arrays are a fundamental data structure in many programming languages, libraries, and frameworks. They have well-established conventions, support in standard libraries, and compatibility with various tools and APIs.**

**It's important to note that the choice between arrays and linked lists depends on the specific requirements of the problem at hand. While linked lists offer advantages like dynamic size, efficient insertion and deletion, and flexibility, arrays provide benefits in terms of random access, memory efficiency, simplicity, and performance. Each data structure has its strengths and weaknesses, and the selection depends on factors such as the nature of the problem, operations to be performed, memory constraints, and performance considerations.**

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.

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, establishing the order.

b. False: In a linked list, memory allocated for the nodes is not necessarily sequential. Each node can be dynamically allocated in different memory locations, and they are linked through pointers or references.

c. False: In a singly linked list, traversal is only possible in the forward direction. Each node has a reference to the next node, allowing traversal from the head to the tail. To traverse in the reverse direction, a doubly linked list is required.

d. True: In a linked list, nodes are commonly inserted at the beginning or the end because linked lists are not optimized for random access. Inserting in the middle of a linked list requires traversing the list to find the insertion point, which can be less efficient than other data structures.

e. False: The head pointer of a linked list is used to traverse the list. It points to the first node, and by following the references to subsequent nodes, the list can be traversed from beginning to end. The head pointer is essential for accessing and navigating the linked 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

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());

To answer the question, I need to have information on the specific implementation of the `list`, `A`, and `B` variables in the provided Java code. Without that information, I can't determine the output of the statements accurately.

However, I can provide you with a general understanding of what the statements would do and the potential outcomes:

a. `System.out.println(list.getElement());`: This statement would call the `getElement()` method on the `list` object and print the returned value. The output would depend on the implementation of the `getElement()` method and the content of the `list`.

b. `System.out.println(A.getElement());`: This statement would call the `getElement()` method on the `A` object and print the returned value. The output would depend on the implementation of the `getElement()` method and the content of the `A` object.

c. `System.out.println(B.getNext().getElement());`: This statement would call the `getNext()` method on the `B` object, retrieve the next node, and then call the `getElement()` method on that node. The output would depend on the implementation of the `getNext()` and `getElement()` methods and the content of the `B` object and its next node.

d. `System.out.println(list.getNext().getNext().getElement());`: This statement would call the `getNext()` method on the `list` object twice to retrieve the next two nodes, and then call the `getElement()` method on the second retrieved node. The output would depend on the implementation of the `getNext()` and `getElement()` methods and the content of the `list` object and its subsequent nodes.

To obtain accurate output for these statements, you would need to examine the code implementation of the `list`, `A`, and `B` variables and ensure that they are properly instantiated and populated with relevant data.

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

To determine the value of each of the relational expressions, I need to have information on the specific implementation of the `list`, `A`, and `B` variables in the provided code. Without that information, I can't accurately determine the values of the expressions.

However, I can explain the general behavior of these relational expressions:

a. `list.getElement() >= 18`: This expression compares the value returned by `getElement()` method of `list` with 18 using the greater-than-or-equal-to operator (`>=`). The result would depend on the implementation of `getElement()` and the content of `list`. If the value returned by `getElement()` is greater than or equal to 18, the expression would evaluate to `true`; otherwise, it would evaluate to `false`.

b. `list.getNext() == A`: This expression compares the reference returned by `getNext()` method of `list` with the reference of `A` using the equality operator (`==`). The result would depend on the implementation of `getNext()` and the content of `list` and `A`. If the references point to the same object, the expression would evaluate to `true`; otherwise, it would evaluate to `false`.

c. `A.getNext().getElement() == 16`: This expression first calls the `getNext()` method on `A` to retrieve the next node, then calls the `getElement()` method on that node, and finally compares the returned value with 16 using the equality operator (`==`). The result would depend on the implementation of `getNext()` and `getElement()`, as well as the content of `A` and its next node. If the value returned by `getElement()` is equal to 16, the expression would evaluate to `true`; otherwise, it would evaluate to `false`.

d. `B.getNext() == (NULL)`: This expression compares the reference returned by `getNext()` method of `B` with the null reference using the equality operator (`==`). The result would depend on the implementation of `getNext()` and the content of `B`. If the reference returned by `getNext()` is null, indicating the absence of a next node, the expression would evaluate to `true`; otherwise, it would evaluate to `false`.

e. `list.getElement() == 18`: This expression compares the value returned by `getElement()` method of `list` with 18 using the equality operator (`==`). The result would depend on the implementation of `getElement()` and the content of `list`. If the value returned by `getElement()` is equal to 18, the expression would evaluate to `true`; otherwise, it would evaluate to `false`.

To obtain accurate values for these expressions, you would need to examine the code implementation of `list`, `A`, and `B` and ensure that they are properly instantiated and populated with relevant data.

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 appropriate methods and attributes

// 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 node10 = new Node(10);

node10.setNext(A.getNext());

A.setNext(node10);

// g. Delete the node with element 23 and deallocate the memory occupied by this node

Node previousNode = findPreviousNode(list, A);

if (previousNode != null) {

previousNode.setNext(A.getNext());

} else {

list = A.getNext();

}

A = null; // Deallocate memory occupied by the node

// Helper method to find a node with a specific value

private Node findNodeWithValue(Node head, int value) {

Node current = head;

while (current != null) {

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

return current;

}

current = current.getNext();

}

return null;

}

// Helper method to find the last node in the list

private Node findLastNode(Node head) {

Node current = head;

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

current = current.getNext();

}

return current;

}

// Helper method to find the previous node of a given node

private Node findPreviousNode(Node head, Node target) {

Node current = head;

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

current = current.getNext();

}

return current;

}

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

p = list;

while (p != NULL){

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

p = p.getNext(); }

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(); }

list = new node<>();

list.setElement(10);

p = new node<>();

p.setElement(13);

p.setNext(null);

list.setNext(p);

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

list.setNext(p);

System.out.println(list.getElement()); // Output: 10

System.out.println(p.getElement()); // Output: 18

p = p.getNext();

System.out.println(p.getElement()); // Output: 13

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?

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?

// Assume the doubly linked list is represented by a class called DoublyLinkedList

// with appropriate methods and attributes.

// Traverse the list to find the node with the value 15

Node currentNode = list.getHead();

while (currentNode != null && currentNode.getValue() != 15) {

currentNode = currentNode.getNext();

}

// Create a new node with the value 20

Node newNode = new Node(20);

// Adjust the links of the new node

newNode.setPrevious(currentNode);

newNode.setNext(currentNode.getNext());

// Update the links of the previous and next nodes

currentNode.getNext().setPrevious(newNode);

currentNode.setNext(newNode);



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

public int sum(Node<Integer> list) {

int sum = 0;

Node<Integer> currentNode = list;

while (currentNode != null) {

sum += currentNode.getElement();

currentNode = currentNode.getNext();

}

return sum;

}.

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}.

public E removeLast(Node<E> list) {

Node<E> currentNode = list;

// Traverse to the last node

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

currentNode = currentNode.getNext();

}

// Update the previous node's next reference

currentNode.getPrevious().setNext(null);

// Return the element of the removed node

return currentNode.getElement();

}

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.

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();

}

// Update the next reference of the last node of list1 to point to the head of list2

currentNode.setNext(list2);

}

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> newHead = null;

Node<E> tail = null;

// Copy list1

Node<E> currentNode = list1;

while (currentNode != null) {

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

if (newHead == null) {

newHead = newNode;

tail = newNode;

} else {

tail.setNext(newNode);

tail = newNode;

}

currentNode = currentNode.getNext();

}

// Copy list2

currentNode = list2;

while (currentNode != null) {

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

if (newHead == null) {

newHead = newNode;

tail = newNode;

} else {

tail.setNext(newNode);

tail = newNode;

}

currentNode = currentNode.getNext();

}

return newHead;

}

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

if (i == j) {

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

return;

}

// Find the nodes at positions i and j

Node<E> nodeI = getNodeAtPosition(list, i);

Node<E> nodeJ = getNodeAtPosition(list, j);

if (nodeI == null || nodeJ == null) {

// Invalid indices, one or both nodes not found

return;

}

// Swap the elements of nodes i and j

E temp = nodeI.getElement();

nodeI.setElement(nodeJ.getElement());

nodeJ.setElement(temp);

}

private Node<E> getNodeAtPosition(Node<E> list, int position) {

Node<E> currentNode = list;

int currentPosition = 0;

while (currentNode != null && currentPosition < position) {

currentNode = currentNode.getNext();

currentPosition++;

}

return currentNode;

}

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 a constant amount of additional space, you can follow these steps:**

**1. Initialize three pointers: `prev` (initialized as `null`), `current` (initialized as the head of the list), and `next` (initialized as `null`).**

**2. Iterate through the list `L` using the `current` pointer until it reaches the end of the list (i.e., `current` becomes `null`).**

**3. Inside each iteration:**

**a. Store the next node of the current node in the `next` pointer. This is necessary to keep track of the next node before modifying the `next` reference of the current node.**

**b. Set the `next` reference of the current node to the previous node (`prev`). This step effectively reverses the `next` pointer of the current node, pointing it to the previous node instead of the next node.**

**c. Move the `prev` pointer to the current node.**

**d. Move the `current` pointer to the next node (stored in the `next` pointer).**

**4. After the iteration, the `prev` pointer will be pointing to the new head of the reversed list.**

**5. Update the head of the list `L` to the `prev` pointer, making the new head the former tail of the original list.**

**6. The list `L` is now reversed.**

**This algorithm reverses the list by iteratively modifying the `next` pointer of each node, effectively reversing the direction of the pointers. It only uses three pointers (`prev`, `current`, and `next`) to keep track of the necessary nodes, and no additional data structures are required.**

**The time complexity of this algorithm is O(n), where n is the number of nodes in the list `L`, as it iterates through the list once. The space complexity is O(1), as it uses a constant amount of additional space regardless of the size of the list.**

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

**public boolean equals(DoublyLinkedList<E> otherList) {**

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

**return false;**

**}**

**Node<E> currentNodeThis = this.getHead();**

**Node<E> currentNodeOther = otherList.getHead();**

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

**if (!currentNodeThis.getElement().equals(currentNodeOther.getElement())) {**

**return false;**

**}**

**currentNodeThis = currentNodeThis.getNext();**

**currentNodeOther = currentNodeOther.getNext();**

**}**

**return true;**

**}**

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

**public void rotate() {**

**if (isEmpty() || getHead().getNext() == getHead()) {**

**// No need to rotate if the list is empty or has only one element**

**return;**

**}**

**Node<E> lastNode = getHead();**

**// Traverse to the last node of the list**

**while (lastNode.getNext() != getHead()) {**

**lastNode = lastNode.getNext();**

**}**

**// Move the head to the next node**

**setHead(getHead().getNext());**

**// Make the last node the new head by updating its next pointer**

**lastNode.setNext(getHead());**

**}**

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

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

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

**if (isEmpty()) {**

**// If the list is empty, make the new node the head and point its next pointer to itself**

**setHead(newNode);**

**newNode.setNext(newNode);**

**} else {**

**// If the list is not empty, insert the new node at the beginning**

**Node<E> lastNode = getHead();**

**// Traverse to the last node of the list**

**while (lastNode.getNext() != getHead()) {**

**lastNode = lastNode.getNext();**

**}**

**// Insert the new node at the beginning**

**newNode.setNext(getHead());**

**lastNode.setNext(newNode);**

**setHead(newNode);**

**}**

**setSize(size() + 1);**

**}**