Home Work

Q1 the answer:  There's nothing inherently wrong with the code. It initializes an object of the **Arrays** class. However, if the **Arrays** class does not exist, the code will produce a compilation error.

Q2 the answer: void reverse(int[] a) {

// reverses the elements of a[]

int left = 0;

int right = a.length - 1;

while (left < right) {

// swap the elements at the left and right indices

int temp = a[left];

a[left] = a[right];

a[right] = temp;

// move the left and right indices towards each other

left++;

right--;

}

}

THE TEST IS:

public static void main(String[] args) {

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

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

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

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

}

reverse(arr);

System.out.println("\nReversed array:");

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

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

}

}

Q3: the answer

1- Arrays have some advantages over linked lists, particularly when it comes to accessing or searching elements. This is because accessing an element in an array is faster because we can directly calculate its index using the base address and the size of each element.

2- Linked lists, on the other hand, allow for efficient insertion and deletion of elements, particularly at the beginning or end of the list. However, accessing elements in a linked list can be slow, especially for large lists, because we need to traverse the list from the beginning or end, depending on the required element.

3- Another factor to consider is memory usage. Arrays use a single block of memory, which is allocated and deallocated as a unit. On the other hand, linked lists use a separate block of memory for each element, which can lead to higher memory usage if there are a large number of elements in the list.

4- Finally, arrays are simpler to implement and use compared to linked lists. This can make them more suitable for smaller tasks or projects where the focus is not on manipulating large or complex data structures.

Q4: **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.

the answer:

a. True b. False c. False d. False e. False

the answerQ5:

a. System.out.println( list.getElement()); Output: 5 This statement prints the element of the first node in the list, which is 5.

b. System.out.println( A. getElement()); Output: 7 This statement prints the element of the node A, which is 7.

c. System.out.println( B.getNext().getElement()); Output: 1 This statement prints the element of the node following node B, which is 1.

d. System.out.println( list.getNext().getNext().getElement()); Output: 3 This statement prints the element of the third node in the list, which is 3.

Please note that these outputs are based on the assumption that the list, p, s, A, and B are correctly pointing to the nodes in the linked list.

Q6 the answer:

a. list.getElement() >= 18 Output: False This expression checks if the element of the first node in the list is greater than or equal to 18.

b. list.getNext() == A Output: True This expression checks if the node following the first node in the list is the same as node A.

c. A.getNext().getElement() == 16 Output: False This expression checks if the element of the node following node A is 16.

d. B.getNext() == (NULL) Output: True This expression checks if the node following node B is the NULL node.

e. list.getElement() == 18 Output: False This expression checks if the element of the first node in the list is 18.

the answerQ7:

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

class LinkedList {

Node head;

public LinkedList() {

this.head = null;

}

public void insert(int data) {

Node newNode = new Node(data);

if (this.head == null) {

this.head = newNode;

} else {

Node temp = this.head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public void delete(int key) {

Node temp = head, prev = null;

if (temp != null && temp.data == key) {

head = temp.next;

return;

}

while (temp != null && temp.data != key) {

prev = temp;

temp = temp.next;

}

if (temp == null) {

return;

}

prev.next = temp.next;

}

public Node search(int key) {

Node current = head;

while (current != null) {

if (current.data == key) {

return current;

}

current = current.next;

}

return null;

}

}

public class Main {

public static void main(String[] args) {

LinkedList list = new LinkedList();

list.insert(25);

list.insert(23);

list.insert(16);

list.insert(10);

list.insert(5);

Node A = list.search(16);

Node B = list.search(25);

list.delete(23);

list.insert(10);

}

}

The answer Q8:

 The output of the given Java code is the elements of the linked list, one per line. The while loop iterates through the linked list until it reaches the end (NULL). The output would be the same as if the linked list were printed using the built-in Java method, System.out.println(list).

The answer Q9:

The provided Java code can be separated into two sections. The first section creates a new linked list and assigns specific values to each node. The second section rearranges the linked list in a specific way.

Section a: The code in section a creates a linked list with three nodes. The nodes are filled with the following integer values: 10, 13, and 18.

Section b: The code in section b creates a new linked list and assigns specific values to each node. The nodes are filled with the following integer values: 20, 28, 30, and 42. After creating the linked list, the code rearranges the linked list in the following way:

20 <-> 28 <-> 30 <-> 42

This means that the next node of the node containing the value 20 is the node containing the value 28. The next node of the node containing the value 28 is the node containing the value 30. The next node of the node containing the value 30 is the node containing the value 42. The last node, which contains the value 42, points to NULL.

After rearranging the linked list, the code prints the values of each node in the linked list.

Output:

20 28 30 42

These values are printed by the following code in section b:

while (p != NULL) { System.out.println(p.getElement()); p = p.getNext(); }

The variable "p" is used to traverse the linked list from the first node to the last node. The code prints the value of each node using the getElement() method and updates the value of "p" to refer to the next node in the linked list using the getNext() method. The loop continues until "p" refers to the last node in the linked list, which is NULL.

The answer Q10:

18, 38, 2, 15, 45, 25 The SingleLinkedList is implemented using a LinkedList. The program first creates a LinkedList named list. Then it adds various numbers to the beginning or end of the list using the addFirst() and addLast() methods. After adding the numbers, the program removes the node with the value 30 using the removeNode() method. It also removes the nodes with the values 28 and 12. Finally, the program prints the remaining nodes in the list using the print() method.

The output of the program is a list of the remaining nodes in the list. Since the removeNode() method does not exist in the SingleLinkedList class, it cannot remove any node from the list. Therefore, the program simply adds all the numbers to the list and then prints the entire list.

The answer Q11:

public class SingleLinkedList

{

public class Node<T>

{

public T Value { get; set; }

public Node<T> Next { get; set; }

}

public int Sum(Node<int> list)

{

int sum = 0;

Node<int> current = list;

while (current != null)

{

sum += current.Value;

current = current.Next;

}

return sum;

}

}

You can test this method with the following code:

class Program

{

static void Main(string[] args)

{

SingleLinkedList.Node<int> list = new SingleLinkedList.Node<int> { Value = 25 };

list.Next = new SingleLinkedList.Node<int> { Value = 45 };

list.Next.Next = new SingleLinkedList.Node<int> { Value = 65 };

list.Next.Next.Next = new SingleLinkedList.Node<int> { Value = 85 };

SingleLinkedList linkedList = new SingleLinkedList();

int sum = linkedList.Sum(list);

Console.WriteLine("The sum of the integers in the specified list is: " + sum);

}

}

The answer Q13:

 Write and test this method for DoublyLinkedList class:

public E removeLast(Node<E> list) {

if (list == null || list.next == null) {

throw new IllegalArgumentException("The list has less than two nodes");

}

// Store a reference to the second-last node

Node<E> secondLastNode = list;

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

secondLastNode = secondLastNode.next;

}

// Store the value of the last node

E removedValue = secondLastNode.next.data;

// Change the 'next' pointer of the second-last node to 'null'

secondLastNode.next = null;

// Return the value of the removed node

return removedValue;

}

For example, if list is {22, 44, 66, 88}, then removeLast(list) will change it to {22, 44, 66}.

You can test this method with the following test case:

public static void main(String[] args) {

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

dll.addLast(22);

dll.addLast(44);

dll.addLast(66);

dll.addLast(88);

Integer removedValue = dll.removeLast(dll.head);

System.out.println("Removed value: " + removedValue); // Output: Removed value: 88

// dll.head now points to {22, 44, 66}

}

The answer Q12:

public class Node<T>

{

public T Data { get; set; }

public Node<T> Next { get; set; }

}

public class SingleLinkedList

{

public Node<int> Head { get; set; }

public int Sum(Node<int> list)

{

int sum = 0;

while (list != null)

{

sum += list.Data;

list = list.Next;

}

return sum;

}

}

class Program

{

static void Main(string[] args)

{

SingleLinkedList list = new SingleLinkedList();

list.Head = new Node<int> { Data = 25 };

list.Head.Next = new Node<int> { Data = 45 };

list.Head.Next.Next = new Node<int> { Data = 65 };

list.Head.Next.Next.Next = new Node<int> { Data = 85 };

int sum = list.Sum(list.Head);

Console.WriteLine($"The sum of the integers in the list is: {sum}");

}

}

Q14:

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

if (list1 == null) {

throw new IllegalArgumentException("list1 cannot be null.");

}

// Find the last node of list1

Node<E> currentNode = list1;

while (currentNode.next != null) {

currentNode = currentNode.next;

}

// Append list2 to list1

currentNode.next = list2;

}

// Define the node class

class Node<E> {

E value;

Node<E> next;

public Node(E value) {

this.value = value;

}

}

// Define the SingleLinkedList class

class SingleLinkedList<E> {

Node<E> head;

// ... other methods

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

// Implementation as mentioned above

}

}

// Example usage

SingleLinkedList<Integer> list1 = new SingleLinkedList<>();

list1.head = new Node<>(22);

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

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

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

SingleLinkedList<Integer> list2 = new SingleLinkedList<>();

list2.head = new Node<>(66);

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

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

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

list1.append(list1.head, list2.head);

// Verify the result

Node<Integer> currentNode = list1.head;

while (currentNode != null) {

System.out.println(currentNode.value);

currentNode = currentNode.next;

}

Q15:

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

if (list1 == null) {

return copyList(list2); // Create a copy of list2 if list1 is empty

}

Node<E> newList = copyList(list1); // Create a copy of list1

// Find the last node of the new list

Node<E> currentNode = newList;

while (currentNode.next != null) {

currentNode = currentNode.next;

}

// Append a copy of list2 to the new list

currentNode.next = copyList(list2);

return newList;

}

// Helper method to create a copy of a linked list

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

if (head == null) {

return null;

}

Node<E> newHead = new Node<>(head.value);

Node<E> currentNode = head.next;

Node<E> newCurrentNode = newHead;

while (currentNode != null) {

newCurrentNode.next = new Node<>(currentNode.value);

currentNode = currentNode.next;

newCurrentNode = newCurrentNode.next;

}

return newHead;

// Define the node class

class Node<E> {

E value;

Node<E> next;

public Node(E value) {

this.value = value;

}

}

// Define the SingleLinkedList class

class SingleLinkedList<E> {

Node<E> head;

// ... other methods

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

// Implementation as mentioned above

}

}

// Example usage

SingleLinkedList<Integer> list1 = new SingleLinkedList<>();

list1.head = new Node<>(22);

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

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

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

SingleLinkedList<Integer> list2 = new SingleLinkedList<>();

list2.head = new Node<>(66);

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

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

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

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

// Verify the result

Node<Integer> currentNode = concatenatedList;

while (currentNode != null) {

System.out.println(currentNode.value);

currentNode = currentNode.next;

}

Q16:

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

if (i == j) {

return; // No need to swap if the indices are the same

}

Node<E> node1 = getNodeAtIndex(list, i);

Node<E> node2 = getNodeAtIndex(list, j);

if (node1 == null || node2 == null) {

throw new IndexOutOfBoundsException("Invalid index provided.");

}

// Swap the values of the nodes

E temp = node1.value;

node1.value = node2.value;

node2.value = temp;

}

// Helper method to get the node at a specific index

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

Node<E> currentNode = list;

int currentIndex = 0;

while (currentNode != null && currentIndex < index) {

currentNode = currentNode.next;

currentIndex++;

}

return currentNode;

}

// Define the node class

class Node<E> {

E value;

Node<E> previous;

Node<E> next;

public Node(E value) {

this.value = value;

}

}

// Define the DoublyLinkedList class

class DoublyLinkedList<E> {

Node<E> head;

// ... other methods

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

// Implementation as mentioned above

}

}

// Example usage

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

list.head = new Node<>(22);

list.head.next = new Node<>(33);

list.head.next.previous = list.head;

list.head.next.next = new Node<>(44);

list.head.next.next.previous = list.head.next;

list.head.next.next.next = new Node<>(55);

list.head.next.next.next.previous = list.head.next.next;

list.head.next.next.next.next = new Node<>(66);

list.head.next.next.next.next.previous = list.head.next.next.next;

list.head.next.next.next.next.next = new Node<>(77);

list.head.next.next.next.next.next.previous = list.head.next.next.next.next;

list.head.next.next.next.next.next.next = new Node<>(88);

list.head.next.next.next.next.next.next.previous = list.head.next.next.next.next.next;

list.head.next.next.next.next.next.next.next = new Node<>(99);

list.head.next.next.next.next.next.next.next.previous = list.head.next.next.next.next.next.next;

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

// Verify the result

Node<Integer> currentNode = list.head;

while (currentNode != null) {

System.out.println(currentNode.value);

currentNode = currentNode.next;

}

Q17:

To reverse a singly linked list `L` using only a constant amount of additional space, you can follow the steps outlined in the algorithm below:

* Initialize three pointers: `previous`, `current`, and `next`. Set `previous` to `null` and `current` to the head of the original list `L`.
* Iterate through the linked list by moving the `current` pointer until it reaches the end of the list (i.e., `current` becomes `null`).
* Inside the loop, store the reference to the next node of the `current` node in the `next` pointer. This is necessary to prevent losing the reference to the remaining part of the list after reversing the link.
* Reverse the link of the `current` node by pointing its `next` pointer to the `previous` node.
* Move the `previous` pointer to the `current` node.
* Move the `current` pointer to the next node (using the reference stored in the `next` pointer).
* Repeat steps 3 to 6 until the end of the list is reached (i.e., `current` becomes `null`).
* After reaching the end of the list, set the head of the reversed list to the `previous` node, which will now be the first node of the reversed list.
* Return the head of the reversed list.

Q18:

public boolean equals(Object other) {

// If the object is compared with itself, return true

if (this == other) {

return true;

}

// If the object is not an instance of DoublyLinkedList, return false

if (!(other instanceof DoublyLinkedList)) {

return false;

}

// Typecast the object to DoublyLinkedList

DoublyLinkedList linkedList = (DoublyLinkedList) other;

// If the size of the two linked lists is not the same, return false

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

return false;

}

// Get the heads of the two linked lists

Node head1 = this.head;

Node head2 = linkedList.head;

// Traverse the linked lists and compare each corresponding pair of nodes

while (head1 != null && head2 != null) {

if (head1.data != head2.data) {

return false;

}

head1 = head1.next;

head2 = head2.next;

}

// If we have reached this point, the two linked lists are equal

return true;

}

Q19:

public void rotate() {

// If the circular linked list is empty, do nothing

if (head == null) {

return;

}

// Get the tail of the circular linked list

Node tail = head.prev;

// Rotate the circular linked list by moving the tail to the head

head = tail;

head.next = null;

// Set the next pointer of the new tail to the new head

tail.next = head;

// Update the tail pointer of the circular linked list

head.prev = tail;

}

Q20:

public class CircularLinkedList<E> {

private Node<E> head;

private int size;

private static class Node<E> {

E data;

Node<E> next;

public Node(E data) {

this.data = data;

this.next = null;

}

}

// ... other methods

public void addFirst(E element) {

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

if (head == null) {

head = newNode;

head.next = head; // Point to itself for circularity

} else {

Node<E> lastNode = head;

while (lastNode.next != head) {

lastNode = lastNode.next;

}

newNode.next = head;

head = newNode;

lastNode.next = head;

}

size++;

}

}