**C-DAC Mumbai Date 01/10/2024**

**Subject: Algorithm and Data Structure**

**Assignment 3**

**Solve the assignment with following thing to be added in each question.**

-Program

-Flow chart

-Explanation

-Output

-Time and Space complexity

Submission Date: 3/10/2024

**1. Implement a singly linked list with basic operations: insert, delete, search.**

**Solution—**

**package** assign4;

**import** java.util.Scanner;

**class** SLL{

Node head;

**static** **class** Node {

**int** data;

Node next;

Node(**int** d){

data=d;

next=**null**;

}

}

**void** insert(**int** data) {

Node n=**new** Node(data);

n.next=head;

head =n;

}

**void** delete(**int** data) {

Node temp=head, prev=**null**;

**if**(temp!=**null** && temp.data==data)

{ head=temp.next;

**return**; }

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

prev=temp;

temp=temp.next;

}

prev.next=temp.next;

}

**boolean** search(**int** data) {

Node temp=head;

**while**(temp!=**null**)

{ **if** (temp.data==data)

**return** **true**;

temp=temp.next;

}

**return** **false**;

}

**void** display() {

Node n=head;

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

System.***out***.print(n.data+"-->");

n=n.next;

}

System.***out***.println();

}

}

**public** **class** A4\_Q1 {

**static** Scanner *sc*=**new** Scanner(System.***in***);

**static** SLL *ll*=**new** SLL();

**static** **void** insert() {

System.***out***.println("Enter the data for new element:");

*ll*.insert(*sc*.nextInt());

}

**static** **void** search() {

System.***out***.println("Enter the element:");

System.***out***.println(*ll*.search(*sc*.nextInt()));

}

**static** **void** delete() {

System.***out***.println("Enter the element to be deleted:");

*ll*.delete(*sc*.nextInt());

}

**static** **void** display() {

*ll*.display();

}

**static** **int** menuList() {

System.***out***.println("1.Enter an element");

System.***out***.println("2.Search an element");

System.***out***.println("3.Delete an element");

System.***out***.println("4.Display");

System.***out***.println("0.Exit");

System.***out***.print("Enter choice:");

**int** c=*sc*.nextInt();

**return** c;

}

**public** **static** **void** main(String[] args) {

**int** choice;

**while**((choice =*menuList*())!=0) {

**switch**(choice) {

**case** 1:

*insert*();

**break**;

**case** 2:

*search*();

**break**;

**case** 3:

*delete*();

**break**;

**case** 4:

*display*();

}

}

}

}

* **Test Case 1**:  
  Input: Insert 3 → Insert 7 → Insert 5 → Delete 7 → Search 5  
  Output: List = [3, 5], Found = True
* **Test Case 2**:  
  Input: Insert 9 → Insert 4 → Delete 4 → Search 10  
  Output: List = [9], Found = False

**2. Reverse a singly linked list.**

* **Test Case 1**:  
  Input: List = [1, 2, 3, 4, 5]  
  Output: List = [5, 4, 3, 2, 1]
* **Test Case 2**:  
  Input: List = [10, 20, 30]  
  Output: List = [30, 20, 10]
* **package** assign4;
* **class** LL{
* Node head;
* **static** **class** Node {
* **int** data;
* Node next;
* Node(**int** d){
* data=d;
* next=**null**;
* }
* }
* **void** insert(**int** d) {
* Node n=**new** Node(d);
* **if**(head==**null**)
* {
* head=n;
* **return**;
* }
* n.next=head;
* head=n;
* }
* Node reverse() {
* Node n=head;
* Node current=head;
* Node next,prev=**null**;
* **while**(current!=**null**) {
* next=current.next;
* current.next=prev;
* prev=current;
* current=next;
* }
* head=prev;
* **return** n;
* }

* **void** display() {
* Node n=head;
* **while**(n!=**null**)
* { System.***out***.print(n.data+",");
* n=n.next;
* }System.***out***.println();
* }





* }
* **public** **class** A4\_Q2 {
* **public** **static** **void** main(String[] args) {
* LL l=**new** LL();
* l.insert(1);
* l.insert(2);
* l.insert(3);
* l.insert(4);
* l.insert(5);
* l.display();
* l.reverse();
* l.display();
* }
* }

**3. Detect a cycle in a linked list.**

* **Test Case 1**:  
  Input: List = [1 → 2 → 3 → 4 → 5 → 3 (cycle)]  
  Output: Cycle Detected
* **Test Case 2**:  
  Input: List = [6 → 7 → 8 → 9]  
  Output: No Cycle
* **package** assign4;
* **import** assign4.LL.Node;
* **class** LL1{
* Node head;
* **static** **class** Node{
* **int** data;
* Node next;
* Node(**int** d){
* data=d;
* next=**null**;
* }
* }
* **void** insert(**int** d) {
* Node n=**new** Node(d);
* **if**(head==**null**)
* {
* head=n;
* **return**;
* }
* Node c;
* Node temp=head;
* **while**(temp.next!=**null**)
* { **if**(temp.data==d)
* {
* n.next=temp;
* }
* temp=temp.next;}
* temp.next=n;
* }
* **void** cycle() {
* Node slow=head,fast=head;
* **while**(fast!=**null** && fast.next!=**null**) {
* slow=slow.next;
* fast=fast.next.next;
* **if**(slow==fast)
* {System.***out***.println("cycle detected");
* **return**;}}
* System.***out***.println("no cycle");
* }
* }
* **public** **class** A4\_Q3 {
* **public** **static** **void** main(String[] args) {
* LL1 l=**new** LL1();
* l.insert(1);
* l.insert(2);
* l.insert(3);
* l.insert(4);
* l.insert(5);
* l.insert(3);
* l.cycle();
* }
* }

**4. Merge two sorted linked lists.**

* **Test Case 1**:  
  Input: List1 = [1, 3, 5], List2 = [2, 4, 6]  
  Output: Merged List = [1, 2, 3, 4, 5, 6]
* **Test Case 2**:  
  Input: List1 = [10, 15, 20], List2 = [12, 18, 25]  
  Output: Merged List = [10, 12, 15, 18, 20, 25]
* **package** assign4;
* **class** mergeLL{
* Node head;
* **static** **class** Node {
* **int** data;
* Node next;
* Node(**int** d){
* data=d;
* next=**null**;
* }
* }
* **void** insert(**int** d) {
* Node n=**new** Node(d);
* **if**(head==**null**)
* {
* head=n;
* **return**;
* }
* Node temp=head;
* **while**(temp.next!=**null**)
* {
* temp=temp.next;}
* temp.next=n;
* }
* **static** Node merge(Node l1, Node l2) {
* **if**(l1==**null**)
* **return** l2;
* **if**(l2==**null**)
* **return** l1;
* **if**(l1.data<l2.data)
* {l1.next=*merge*(l1.next,l2);
* **return** l1;
* }
* **else**
* l2.next=*merge*(l1,l2.next);
* **return** l2;
* }
* **void** display(Node no) {
* Node n=no;
* **while**(n!=**null**)
* { System.***out***.print(n.data+",");
* n=n.next;
* }System.***out***.println();
* }
* }
* **public** **class** A4\_Q4 {
* **public** **static** **void** main(String[] args) {
* mergeLL l1=**new** mergeLL();
* mergeLL l2=**new** mergeLL();
* l1.insert(1);
* l1.insert(3);
* l1.insert(5);
* l2.insert(2);
* l2.insert(4);
* l2.insert(6);
* mergeLL.Node n=mergeLL.*merge*(l1.head,l2.head);
* l2.display(n);
* }
* }

**5. Find the nth node from the end of a linked list.**

* **Test Case 1**:  
  Input: List = [10, 20, 30, 40, 50], n = 2  
  Output: 40
* **Test Case 2**:  
  Input: List = [5, 15, 25, 35], n = 4  
  Output: 5
* **package** assign4;
* **class** nLL{
* Node head;
* **static** **class** Node {
* **int** data;
* Node next;
* Node(**int** d){
* data=d;
* next=**null**;
* }
* }
* **void** insert(**int** d) {
* Node n=**new** Node(d);
* **if**(head==**null**)
* {
* head=n;
* **return**;
* }
* Node temp=head;
* **while**(temp.next!=**null**)
* {
* temp=temp.next;}
* temp.next=n;
* }
* **int** find(**int** no) {
* Node temp=head;
* **int** count=0;
* **if**(head==**null**)
* **return** -1;
* **while**(temp!=**null**)
* {count++;
* temp=temp.next;
* }
* temp=head;
* **int** length=count;
* **for**(**int** i=1; i<=length-no;i++ ) {
* temp=temp.next;}
* **return** temp.data;
* }
* }
* **public** **class** A4\_Q5 {
* **public** **static** **void** main(String[] args) {
* nLL l=**new** nLL();
* l.insert(10);
* l.insert(15);
* l.insert(25);
* l.insert(35);
* l.insert(50);
* System.***out***.println(l.find(2));
* }
* }

**6. Remove duplicates from a sorted linked list.**

* **Test Case 1**:  
  Input: List = [1, 1, 2, 3, 3, 4]  
  Output: List = [1, 2, 3, 4]
* **Test Case 2**:  
  Input: List = [7, 7, 8, 9, 9, 10]  
  Output: List = [7, 8, 9, 10]
* **package** assign4;
* **class** dLL{
* Node head;
* **static** **class** Node {
* **int** data;
* Node next;
* Node(**int** d){
* data=d;
* next=**null**;
* }
* }
* **void** insert(**int** d) {
* Node n=**new** Node(d);
* **if**(head==**null**)
* {
* head=n;
* **return**;
* }
* Node temp=head;
* **while**(temp.next!=**null**)
* {
* temp=temp.next;}
* temp.next=n;
* }
* **public** **void** removeDuplicates() {
* Node current = head;
* // Traverse the list
* **while** (current != **null** && current.next != **null**) {
* // Compare current node with the next node
* **if** (current.data == current.next.data) {
* // Skip the next node (duplicate node)
* current.next = current.next.next;
* } **else** {
* // Move to the next node
* current = current.next;
* }
* }
* }
* **void** display() {
* Node n=head;
* **while**(n!=**null**)
* { System.***out***.print(n.data+",");
* n=n.next;
* }System.***out***.println();
* }

* }
* **public** **class** A4\_Q6 {
* **public** **static** **void** main(String[] args) {
* dLL l=**new** dLL();
* l.insert(1);
* l.insert(1);
* l.insert(2);
* l.insert(3);
* l.insert(3);
* l.insert(9);
* l.insert(4);
* l.removeDuplicates();
* l.display();
* }
* }

**7. Implement a doubly linked list with insert, delete, and traverse operations.**

* **Test Case 1**:  
  Input: Insert 10 → Insert 20 → Insert 30 → Delete 20  
  Output: List = [10, 30]
* **Test Case 2**:  
  Input: Insert 1 → Insert 2 → Insert 3 → Delete 1  
  Output: List = [2, 3]

class DoublyLinkedList {

// Node class to define a doubly linked list node

class Node {

int data;

Node prev;

Node next;

Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

// Head and tail pointers

private Node head;

private Node tail;

// Constructor to initialize an empty list

public DoublyLinkedList() {

this.head = null;

this.tail = null;

}

// Insert a node at the end of the list

public void insert(int data) {

Node newNode = new Node(data);

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

if (head == null) {

head = newNode;

tail = newNode;

} else {

// Link the new node to the tail and update the tail

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

}

// Delete a node with a specific value

public void delete(int data) {

if (head == null) {

System.out.println("List is empty, nothing to delete.");

return;

}

Node current = head;

// Traverse the list to find the node to delete

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

current = current.next;

}

// If the node is not found

if (current == null) {

System.out.println("Node with data " + data + " not found.");

return;

}

// If the node to be deleted is the head

if (current == head) {

head = current.next;

if (head != null) {

head.prev = null;

}

}

// If the node to be deleted is the tail

if (current == tail) {

tail = current.prev;

if (tail != null) {

tail.next = null;

}

}

// If the node to be deleted is in the middle

if (current.prev != null) {

current.prev.next = current.next;

}

if (current.next != null) {

current.next.prev = current.prev;

}

System.out.println("Node with data " + data + " deleted.");

}

// Traverse the list in forward order

public void traverseForward() {

if (head == null) {

System.out.println("List is empty.");

return;

}

Node current = head;

System.out.print("Forward Traversal: ");

while (current != null) {

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

current = current.next;

}

System.out.println();

}

// Traverse the list in reverse order

public void traverseBackward() {

if (tail == null) {

System.out.println("List is empty.");

return;

}

Node current = tail;

System.out.print("Backward Traversal: ");

while (current != null) {

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

current = current.prev;

}

System.out.println();

}

// Helper method to check if the list is empty

public boolean isEmpty() {

return head == null;

}

}

public class Main {

public static void main(String[] args) {

DoublyLinkedList dll = new DoublyLinkedList();

dll.insert(10);

dll.insert(20);

dll.insert(30);

dll.insert(40);

dll.traverseForward(); // Output: 10 20 30 40

dll.traverseBackward(); // Output: 40 30 20 10

dll.delete(30);

dll.traverseForward(); // Output: 10 20 40

dll.traverseBackward(); // Output: 40 20 10

dll.delete(10);

dll.traverseForward(); // Output: 20 40

dll.delete(40);

dll.traverseForward(); // Output: 20

dll.delete(50); // Node with data 50 not found.

}

}

**8. Reverse a doubly linked list.**

* **Test Case 1**:  
  Input: List = [5, 10, 15, 20]  
  Output: List = [20, 15, 10, 5]
* **Test Case 2**:  
  Input: List = [4, 8, 12]  
  Output: List = [12, 8, 4]

class DoublyLinkedList {

// Node class to define a doubly linked list node

class Node {

int data;

Node prev;

Node next;

Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

// Head and tail pointers

private Node head;

private Node tail;

// Constructor to initialize an empty list

public DoublyLinkedList() {

this.head = null;

this.tail = null;

}

// Insert a node at the end of the list

public void insert(int data) {

Node newNode = new Node(data);

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

if (head == null) {

head = newNode;

tail = newNode;

} else {

// Link the new node to the tail and update the tail

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

}

// Traverse the list in forward order

public void traverseForward() {

if (head == null) {

System.out.println("List is empty.");

return;

}

Node current = head;

System.out.print("Forward Traversal: ");

while (current != null) {

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

current = current.next;

}

System.out.println();

}

// Reverse the doubly linked list

public void reverse() {

if (head == null) {

System.out.println("List is empty.");

return;

}

Node current = head;

Node temp = null;

// Traverse the list and swap next and prev pointers for each node

while (current != null) {

// Swap next and prev pointers

temp = current.prev;

current.prev = current.next;

current.next = temp;

// Move to the next node (which is now the previous node)

current = current.prev;

}

// After the loop, temp will be the previous node of the current (which is null now)

if (temp != null) {

head = temp.prev; // Update head to the new first node

}

System.out.println("Doubly Linked List reversed.");

}

}

public class Main {

public static void main(String[] args) {

DoublyLinkedList dll = new DoublyLinkedList();

dll.insert(10);

dll.insert(20);

dll.insert(30);

dll.insert(40);

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

dll.traverseForward(); // Output: 10 20 30 40

dll.reverse();

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

dll.traverseForward(); // Output: 40 30 20 10

}

}

**9. Add two numbers represented by linked lists.**

* **Test Case 1**:  
  Input: List1 = [2 → 4 → 3], List2 = [5 → 6 → 4] (243 + 465)  
  Output: Sum List = [7 → 0 → 8]
* **Test Case 2**:  
  Input: List1 = [9 → 9 → 9], List2 = [1] (999 + 1)  
  Output: Sum List = [0 → 0 → 0 → 1]
* **package** assign4;
* **public** **class** addLL {
* Node head;
* **static** **class** Node{
* **int** data;
* Node next;
* Node(**int** d)
* {
* data=d;
* next=**null**;
* }

* }
* **void** insert(**int** d) {
* Node n=**new** Node(d);
* **if**(head==**null**)
* {
* head=n;
* **return**;
* }
* n.next=head;
* head=n;
* }
* **int** adddigit(Node n1) {
* Node temp=n1;
* **int** sum=0;
* **while**(temp!=**null**) {
* sum=sum\*10+temp.data;
* temp=temp.next;
* }
* System.***out***.println(sum);
* **return** sum;
* }
* **int** addResult(Node n1, Node n2) {
* **int** sum1=adddigit(n2);
* **int** sum2=adddigit(n1);
* **return** sum1+sum2;
* }
* **void** rLL(Node n1, Node n2){
* **int** result=addResult(n1,n2);
* System.***out***.println(result);
* addLL l=**new** addLL();
* **while**(result>0) {
* **int** rem=result%10;
* result=result/10;
* l.insert(rem);
* }
* l.display();
* }
* **void** display() {
* Node n=head;
* **while**(n!=**null**)
* { System.***out***.print(n.data+"-->");
* n=n.next;
* }System.***out***.println();
* }
* **public** **static** **void** main(String[] args) {
* addLL l1=**new** addLL();
* addLL l2=**new** addLL();
* l1.insert(3);
* l1.insert(4);
* l1.insert(2);
* l2.insert(4);
* l2.insert(6);
* l2.insert(5);
* l1.rLL(l1.head,l2.head);
* }
* }

**10. Rotate a linked list by k places.**

* **Test Case 1**:  
  Input: List = [10, 20, 30, 40, 50], k = 2  
  Output: List = [30, 40, 50, 10, 20]
* **Test Case 2**:  
  Input: List = [5, 10, 15, 20], k = 3  
  Output: List = [20, 5, 10, 15]

**public** **void** rotate(**int** k) {

* **if** (head == **null** || head.next == **null** || k == 0) {
* **return**;
* }
* // Compute the length of the list
* ListNode current = head;
* **int** length = 1; // since we are starting with the head
* **while** (current.next != **null**) {
* current = current.next;
* length++;
* }
* // If k is greater than the length, rotate only the remainder steps
* k = k % length;
* **if** (k == 0) {
* **return**; // No need to rotate if k is a multiple of the length
* }
* // Connect the last node to the head to make it circular
* current.next = head;
* // Find the new tail, which is (length - k)th node
* **for** (**int** i = 0; i < length - k; i++) {
* current = current.next;
* }
* // Set the new head to be the next node after the new tail
* head = current.next;
* // Break the circular connection
* current.next = **null**;

**11. Flatten a multilevel doubly linked list.**

* **Test Case 1**:  
  Input: List = [1 → 2 → 3, 3 → 7 → 8, 8 → 10 → 12]  
  Output: Flattened List = [1 → 2 → 3 → 7 → 8 → 10 → 12]
* **Test Case 2**:  
  Input: List = [1 → 2 → 3, 2 → 5 → 6, 6 → 7 → 9]  
  Output: Flattened List = [1 → 2 → 5 → 6 → 7 → 9 → 3]

class FlattenMultilevelDoublyLinkedList {

// Node class to define a multilevel doubly linked list node

class Node {

int data;

Node prev;

Node next;

Node child; // Pointer to a child node (which can be another doubly linked list)

Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

this.child = null;

}

}

// Head pointer

private Node head;

// Constructor to initialize an empty list

public FlattenMultilevelDoublyLinkedList() {

this.head = null;

}

// Insert a node at the end of the list

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

newNode.prev = current;

}

}

// Set the child of a specific node (used to create multilevel structure)

public void setChild(Node parent, Node child) {

parent.child = child;

}

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

public Node findNode(int data) {

Node current = head;

while (current != null) {

if (current.data == data) {

return current;

}

current = current.next;

}

return null;

}

// Traverse and print the list in forward order

public void traverse() {

Node current = head;

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

while (current != null) {

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

current = current.next;

}

System.out.println();

}

// Method to flatten the multilevel doubly linked list

public void flatten() {

if (head == null) {

return;

}

Node current = head;

// Traverse the list

while (current != null) {

// If current node has a child, flatten the child list

if (current.child != null) {

Node child = current.child;

// Find the tail of the child list

Node childTail = child;

while (childTail.next != null) {

childTail = childTail.next;

}

// Link the tail of the child list to the next node of the current node

childTail.next = current.next;

if (current.next != null) {

current.next.prev = childTail;

}

// Link the current node to the child

current.next = child;

child.prev = current;

// Remove the child pointer (it's no longer needed)

current.child = null;

}

// Move to the next node

current = current.next;

}

}

public static void main(String[] args) {

FlattenMultilevelDoublyLinkedList list = new FlattenMultilevelDoublyLinkedList();

// Create a main list

list.insert(1);

list.insert(2);

list.insert(3);

list.insert(4);

// Create child list for node 3

Node node3 = list.findNode(3);

FlattenMultilevelDoublyLinkedList childList1 = new FlattenMultilevelDoublyLinkedList();

childList1.insert(7);

childList1.insert(8);

list.setChild(node3, childList1.head);

// Create child list for node 8

Node node8 = childList1.findNode(8);

FlattenMultilevelDoublyLinkedList childList2 = new FlattenMultilevelDoublyLinkedList();

childList2.insert(11);

childList2.insert(12);

childList1.setChild(node8, childList2.head);

System.out.println("Original Multilevel List:");

list.traverse(); // Only prints the top-level list

System.out.println("Flattening the list...");

list.flatten();

System.out.println("Flattened List:");

list.traverse(); // Should print all nodes in a single level

}

}

**12. Split a circular linked list into two halves.**

* **Test Case 1**:  
  Input: Circular List = [1 → 2 → 3 → 4 → 5 → 6 → (back to 1)]  
  Output: List1 = [1 → 2 → 3], List2 = [4 → 5 → 6]
* **Test Case 2**:  
  Input: Circular List = [10 → 20 → 30 → 40 → (back to 10)]  
  Output: List1 = [10 → 20], List2 = [30 → 40]

class CircularLinkedList {

// Node class to define a circular linked list node

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

// Head pointer of the circular linked list

private Node head;

// Constructor to initialize an empty circular linked list

public CircularLinkedList() {

this.head = null;

}

// Insert a node at the end of the circular linked list

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

newNode.next = head; // Point to itself

} else {

Node current = head;

while (current.next != head) {

current = current.next;

}

current.next = newNode;

newNode.next = head; // New node points to head

}

}

// Split the circular linked list into two halves

public CircularLinkedList[] split() {

if (head == null) {

return new CircularLinkedList[]{null, null}; // Both halves are null

}

CircularLinkedList[] result = new CircularLinkedList[2];

Node slow = head;

Node fast = head;

// Use fast and slow pointers to find the middle

while (fast.next != head && fast.next.next != head) {

slow = slow.next; // Move slow by 1

fast = fast.next.next; // Move fast by 2

}

// Create two new circular linked lists

result[0] = new CircularLinkedList();

result[1] = new CircularLinkedList();

// First half starts from head to slow

result[0].head = head;

result[1].head = slow.next; // Second half starts from slow.next

// Break the circular links

slow.next = result[0].head; // End the first half

Node current = result[1].head;

while (current.next != head) { // Find the last node of the second half

current = current.next;

}

current.next = result[1].head; // End the second half

return result;

}

// Helper method to traverse and print the circular linked list

public void traverse() {

if (head == null) {

System.out.println("List is empty.");

return;

}

Node current = head;

do {

System.out.print(current.data + " -> ");

current = current.next;

} while (current != head);

System.out.println("(back to head)");

}

public static void main(String[] args) {

CircularLinkedList list = new CircularLinkedList();

list.insert(1);

list.insert(2);

list.insert(3);

list.insert(4);

list.insert(5);

System.out.println("Original Circular Linked List:");

list.traverse(); // Expected: 1 -> 2 -> 3 -> 4 -> 5 -> (back to head)

// Split the circular linked list

CircularLinkedList[] halves = list.split();

// Print the two halves

System.out.println("First Half:");

halves[0].traverse(); // Expected: 1 -> 2 -> 3 -> (back to head)

System.out.println("Second Half:");

halves[1].traverse(); // Expected: 4 -> 5 -> (back to head)

}

}

**13. Insert a node in a sorted circular linked list.**

* **Test Case 1**:  
  Input: Circular List = [10 → 20 → 30 → 40 → (back to 10)], Insert 25  
  Output: Circular List = [10 → 20 → 25 → 30 → 40 → (back to 10)]
* **Test Case 2**:  
  Input: Circular List = [5 → 15 → 25 → (back to 5)], Insert 10  
  Output: Circular List = [5 → 10 → 15 → 25 → (back to 5)]

class CircularLinkedList {

// Node class to define a circular linked list node

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

// Head pointer of the circular linked list

private Node head;

// Constructor to initialize an empty circular linked list

public CircularLinkedList() {

this.head = null;

}

// Insert a node in a sorted circular linked list

public void insertSorted(int data) {

Node newNode = new Node(data);

// If the list is empty, create a single-node circular list

if (head == null) {

newNode.next = newNode; // Point to itself

head = newNode;

return;

}

Node current = head;

Node prev = null;

// Case 1: Insert before the head (smallest element) or after the tail (largest element)

if (data <= head.data) {

// Find the tail node (which points to the head)

while (current.next != head) {

current = current.next;

}

// Insert the new node before the current head

current.next = newNode; // Tail's next points to new node

newNode.next = head; // New node points to old head

head = newNode; // Update head to the new node

return;

}

// Case 2: Insert in the middle or at the end of the list

current = head;

while (current.next != head && current.next.data < data) {

current = current.next;

}

// Insert the new node after 'current' node

newNode.next = current.next;

current.next = newNode;

}

// Helper method to traverse and print the circular linked list

public void traverse() {

if (head == null) {

System.out.println("List is empty.");

return;

}

Node current = head;

do {

System.out.print(current.data + " -> ");

current = current.next;

} while (current != head);

System.out.println("(back to head)");

}

public static void main(String[] args) {

CircularLinkedList list = new CircularLinkedList();

list.insertSorted(10);

list.insertSorted(20);

list.insertSorted(5); // Should become the new head

list.insertSorted(15);

list.insertSorted(2); // Should become the new head

// Expected order: 2 -> 5 -> 10 -> 15 -> 20 -> (back to head)

System.out.println("Circular Linked List:");

list.traverse();

}

}

**14. Check if two linked lists intersect, and find the intersection point if they do.**

* **Test Case 1**:  
  Input: List1 = [1 → 2 → 3 → 4 → 5], List2 = [6 → 7 → 4 → 5]  
  Output: Intersection Point = 4
* **Test Case 2**:  
  Input: List1 = [10 → 20 → 30 → 40], List2 = [15 → 25 → 35]  
  Output: No Intersection

class LinkedListIntersection {

// Node class to define a singly linked list node

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

// Head pointers of two linked lists

private Node head1;

private Node head2;

// Constructor to initialize both lists as empty

public LinkedListIntersection() {

this.head1 = null;

this.head2 = null;

}

// Insert a node at the end of a list (list1 or list2)

public Node insert(Node head, int data) {

Node newNode = new Node(data);

if (head == null) {

return newNode;

}

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

return head;

}

// Function to find the length of a linked list

public int getLength(Node head) {

int length = 0;

Node current = head;

while (current != null) {

length++;

current = current.next;

}

return length;

}

// Function to find the intersection point of two linked lists

public Node getIntersectionNode(Node head1, Node head2) {

if (head1 == null || head2 == null) {

return null;

}

// Get the lengths of both linked lists

int length1 = getLength(head1);

int length2 = getLength(head2);

// Calculate the difference in lengths

int difference = Math.abs(length1 - length2);

// Move the pointer of the longer list by the difference

if (length1 > length2) {

for (int i = 0; i < difference; i++) {

head1 = head1.next;

}

} else {

for (int i = 0; i < difference; i++) {

head2 = head2.next;

}

}

// Traverse both lists and check for intersection

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

if (head1 == head2) { // Intersection found

return head1;

}

head1 = head1.next;

head2 = head2.next;

}

// No intersection

return null;

}

// Helper function to print a list

public void printList(Node head) {

Node current = head;

while (current != null) {

System.out.print(current.data + " -> ");

current = current.next;

}

System.out.println("null");

}

public static void main(String[] args) {

LinkedListIntersection list = new LinkedListIntersection();

// Creating first linked list: 1 -> 2 -> 3 -> 4 -> 5

list.head1 = list.insert(list.head1, 1);

list.head1 = list.insert(list.head1, 2);

list.head1 = list.insert(list.head1, 3);

list.head1 = list.insert(list.head1, 4);

list.head1 = list.insert(list.head1, 5);

// Creating second linked list: 10 -> 9 -> (intersection at 3 -> 4 -> 5)

list.head2 = list.insert(list.head2, 10);

list.head2 = list.insert(list.head2, 9);

// Create intersection by pointing to node 3 from list1

list.head2.next.next = list.head1.next.next; // Intersection at node 3

// Print the lists

System.out.println("List 1:");

list.printList(list.head1); // Output: 1 -> 2 -> 3 -> 4 -> 5 -> null

System.out.println("List 2:");

list.printList(list.head2); // Output: 10 -> 9 -> 3 -> 4 -> 5 -> null

// Find the intersection point

Node intersection = list.getIntersectionNode(list.head1, list.head2);

if (intersection != null) {

System.out.println("The intersection point is at node with data: " + intersection.data);

} else {

System.out.println("No intersection found.");

}

}

}

**15. Find the middle element of a linked list in one pass.**

* **Test Case 1**:  
  Input: List = [1, 2, 3, 4, 5]  
  Output: Middle = 3
* **Test Case 2**:  
  Input: List = [11, 22, 33, 44, 55, 66]  
  Output: Middle = 44

**package** assign4;

**public** **class** A4\_Q15 {

Node head;

**static** **class** Node{

**int** data;

Node next;

Node(**int** d)

{

data=d;

next=**null**;

}

}

**void** insert(**int** d) {

Node n=**new** Node(d);

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

{

head=n;

**return**;

}

Node temp=head;

**while**(temp.next!=**null**)

{

temp=temp.next;}

temp.next=n;

}

**int** middle() {

Node slow=head, fast=head;

**while**(fast!=**null** && fast.next!=**null**) {

slow=slow.next;

fast=fast.next.next;

}

**return** slow.data;

}

**public** **static** **void** main(String[] args) {

A4\_Q15 a=**new** A4\_Q15();

a.insert(1);

a.insert(2);

a.insert(3);

a.insert(4);

a.insert(5);

a.insert(66);

System.***out***.println(a.middle());

}

}