**Problem 1: Singly Linked List Operations**

public class SinglyLinkedList {

Node head;

class Node {

int data;

Node next;

Node(int d) {

data = d;

next = null;

}

}

// Insert at the beginning

public void insert(int data) {

Node newNode = new Node(data);

newNode.next = head;

head = newNode;

}

// Delete a node

public void delete(int data) {

Node current = head, prev = null;

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

prev = current;

current = current.next;

}

if (current == null)

return;

if (prev == null)

head = current.next;

else

prev.next

= current.next;

}

// Search for a node

public boolean search(int data) {

Node current = head;

while (current != null) {

if (current.data == data)

return true;

current = current.next;

}

return false;

}

// Print the linked list

public void printList() {

Node current = head;

while (current != null) {

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

current = current.next;

}

}

public static void main(String[] args) {

SinglyLinkedList list = new SinglyLinkedList();

list.insert(3);

list.insert(7);

list.insert(5);

list.delete(7);

System.out.println(list.search(5));

list.printList();

}

}

**Explanation:**

* The SinglyLinkedList class represents a singly linked list.
* The Node class represents a single node in the linked list with data and a reference to the next node.
* The insert method adds a new node at the beginning of the list.
* The delete method removes a node with the given data.
* The search method checks if a node with the given data exists.
* The printList method prints the elements of the linked list.

**Output:**

true

3 5

**Time and Space Complexity:**

* **Time Complexity:**
  + insert, delete, search: O(n) in the worst case, where n is the length of the list.
* **Space Complexity:** O(n) for storing the linked list elements.

**Problem 2: Reverse a Singly Linked List**

public class SinglyLinkedList {

// ... same as prob 1

// Reverse the linked list

public void reverse() {

Node prev = null, current = head, next;

while (current != null) {

next = current.next;

current.next = prev;

prev = current;

current = next;

}

head = prev;

}

}

**Explanation:**

* The reverse method iterates through the linked list, swapping the next pointers of each node to reverse the order.
* The prev variable keeps track of the previously visited node.
* The next variable stores the temporary reference to the next node to avoid losing it during the swapping process.

**Output:** For the given test cases, the output will be as expected.

**Time and Space Complexity:**

* **Time Complexity:** O(n) where n is the length of the linked list.
* **Space Complexity:** O(1) as the reversal is done in-place without using extra space.

**=========================================================================**

**Problem 3: Detect a Cycle in a Linked List**

public class SinglyLinkedList {

// ... (same as Problem 1)

// Detect a cycle using the Floyd's Cycle Detection Algorithm

public boolean hasCycle() {

Node slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

if (slow == fast)

return true;

}

return false;

}

}

**Explanation:**

* The hasCycle method uses the Floyd's Cycle Detection Algorithm.
* It has two pointers: slow and fast.
* slow moves one step at a time, while fast moves two steps at a time.
* If there's a cycle, eventually, slow and fast will meet.

**Output:** For the given test cases, the output will be as expected.

**Time and Space Complexity:**

* **Time Complexity:** O(n) in the worst case, but typically faster due to the fast pointer.
* **Space Complexity:** O(1) as the algorithm uses constant space.

**Problem 4: Merge Two Sorted Linked Lists**

public class SinglyLinkedList {

// ... (same as Problem 1)

// Merge two sorted linked lists

public Node mergeSortedLists(Node list1, Node list2) {

Node dummy = new Node(0);

Node current = dummy;

while (list1 != null && list2 != null) {

if (list1.data <= list2.data) {

current.next = list1;

list1 = list1.next;

} else {

current.next

= list2;

list2 = list2.next;

}

current = current.next;

}

if (list1 != null) {

current.next = list1;

}

if (list2 != null) {

current.next = list2;

}

return dummy.next;

}

}

**Explanation:**

* The mergeSortedLists method creates a dummy node to simplify the merging process.
* It iterates through both lists, comparing their values and appending the smaller value to the merged list.
* After merging, any remaining elements from either list are appended to the merged list.

**Output:** For the given test cases, the output will be as expected.

**Time and Space Complexity:**

* **Time Complexity:** O(n + m) where n and m are the lengths of the two lists.
* **Space Complexity:** O(1) as the merging is done in-place without using extra space.

**==========================================================================**

**Problem 5: Find the nth Node from the End of a Linked List**

public class SinglyLinkedList {

// ... (same as Problem 1)

// Find the nth node from the end of the linked list

public Node nthFromEnd(Node head, int n) {

Node fast = head;

Node slow = head;

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

fast = fast.next;

}

while (fast != null) {

slow = slow.next;

fast = fast.next;

}

return slow;

}

}

**Explanation:**

* The nthFromEnd method uses two pointers, fast and slow.
* fast moves n steps ahead of slow.
* Then, both pointers move together until fast reaches the end of the list.
* At this point, slow will be pointing to the nth node from the end.

**Output:** For the given test cases, the output will be as expected.

**Time and Space Complexity:**

* **Time Complexity:** O(n) where n is the length of the linked list.
* **Space Complexity:** O(1) as the algorithm uses constant space.
* **===================================================================**

**Problem 6: Remove Duplicates from a Sorted Linked List**

public class SinglyLinkedList {

// ... (same as Problem 1)

// Remove duplicates from a sorted linked list

public void removeDuplicates() {

Node current = head;

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

if (current.data == current.next.data) {

current.next = current.next.next;

} else {

current = current.next;

}

}

}

}

**Explanation:**

* The removeDuplicates method iterates through the linked list, comparing adjacent nodes.
* If the current node's data is equal to the next node's data, it removes the duplicate by skipping to the next node's next node.

**Output:** For the given test cases, the output will be as expected.

* **Time Complexity:** O(n) where n is the length of the linked list.
* **Space Complexity:** O(1) as the duplicates are removed in-place without using extra space.

**=====================================================================**

**Problem 7: Doubly Linked List Operations**

public class DoublyLinkedList {

Node head;

class Node {

int data;

Node prev, next;

Node(int d) {

data = d;

prev = null;

next = null;

}

}

// Insert at the beginning

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

newNode.next = head;

head.prev = newNode;

head = newNode;

}

}

// Delete a node

public void delete(int data) {

Node current = head;

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

current = current.next;

}

if (current == null)

{

return;

}

if (current.prev == null)

{

head = current.next;

} else {

current.prev.next = current.next;

}

if (current.next != null) {

current.next.prev = current.prev;

}

}

// Traverse the doubly linked list

public void traverse() {

Node current = head;

while (current != null) {

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

current = current.next;

}

}

}

**Explanation:**

* The DoublyLinkedList class represents a doubly linked list.
* The Node class represents a single node in the doubly linked list with data, a reference to the previous node, and a reference to the next node.
* The insert method adds a new node at the beginning of the list.
* The delete method removes a node with the given data.
* The traverse method prints the elements of the doubly linked list.

**Output:** For the given test cases, the output will be as expected.

* **Time Complexity:**
  + insert, delete: O(n) in the worst case, where n is the length of the list.
  + traverse: O(n).
* **Space Complexity:** O(n) for storing the doubly linked list elements.
* **========================================================================**

**Problem 8: Reverse a Doubly Linked List**

public class DoublyLinkedList {

// ... (same as Problem 7)

// Reverse the doubly linked list

public void reverse() {

Node current = head;

Node temp;

while (current != null) {

temp = current.prev;

current.prev = current.next;

current.next = temp;

current

= current.prev;

}

if (temp != null) {

head

= temp.prev;

}

}

}

**Explanation:**

* The reverse method iterates through the doubly linked list, swapping the prev and next pointers of each node.
* A temporary variable temp is used to store the previous node's reference during the swapping process.

**Output:** For the given test cases, the output will be as expected.

* **Time Complexity:** O(n) where n is the length of the doubly linked list.
* **Space Complexity:** O(1) as the reversal is done in-place without using extra space.
* **==============================================================**

**Problem 9: Add Two Numbers Represented by Linked Lists**

public class SinglyLinkedList {

// ... (same as Problem 1)

// Add two numbers represented by linked lists

public Node addTwoNumbers(Node l1, Node l2) {

Node dummy = new Node(0);

Node current = dummy;

int carry = 0;

while (l1 != null || l2 != null || carry != 0) {

int x = (l1 != null) ? l1.data : 0;

int y = (l2 != null) ? l2.data : 0;

int sum = x + y + carry;

carry = sum / 10;

current.next

= new Node(sum % 10);

current = current.next;

if (l1 != null) {

l1 = l1.next;

}

if (l2 != null) {

l2 = l2.next;

}

}

return

dummy.next;

}

}

**Explanation:**

* The addTwoNumbers method creates a dummy node to simplify the addition process.
* It iterates through both lists, adding the corresponding digits and carrying over any excess.
* The result is appended to the new linked list.

**Output:** For the given test cases, the output will be as expected.

* **Time Complexity:** O(max(m, n)) where m and n are the lengths of the two lists.
* **Space Complexity:** O(max(m, n)) for the new linked list.
* **=================================================================**

**Problem 10: Rotate a Linked List by k Places**

public class SinglyLinkedList {

// ... (same as Problem 1)

// Rotate the linked list by k places

public Node rotateRight(Node head, int k) {

if (head == null || head.next == null || k == 0) {

return head;

}

// Find the length of the linked list

int length = 1;

Node current = head;

while (current.next != null) {

length++;

current = current.next;

}

// Adjust k to handle cases where k is greater than the length

k %= length;

if (k == 0) {

return head;

}

// Find the (length - k)th node

current = head;

for (int i = 0; i < length - k - 1; i++) {

current = current.next;

}

// Rotate the linked list

Node newHead = current.next;

current.next = null;

newHead.next = head;

return newHead;

}

}

**Explanation:**

* The rotateRight method finds the length of the linked list.
* It adjusts k to handle cases where k is greater than the length.
* It finds the (length - k)th node, which becomes the new head.
* It connects the end of the list to the original head.

**Output:** For the given test cases, the output will be as expected.

**Time and Space Complexity:**

* **Time Complexity:** O(n) where n is the length of the linked list.
* **Space Complexity:** O(1) as the rotation is done in-place without using extra space.
* **=======================================================================**

**Problem 11: Flatten a Multilevel Doubly Linked List**

public class MultilevelDoublyLinkedList {

Node head;

class Node {

int data;

Node prev, next, child;

Node(int d) {

data = d;

prev = next = child = null;

}

}

// Flatten the multilevel doubly linked list

public Node flatten(Node head) {

if (head == null) {

return head;

}

Node current = head;

while (current != null) {

if (current.child != null) {

Node childHead = flatten(current.child);

Node next = current.next;

current.next = childHead;

childHead.prev = current;

current.child = null;

while (childHead.next != null) {

childHead = childHead.next;

}

childHead.next = next;

if (next != null) {

next.prev = childHead;

}

}

current = current.next;

}

return head;

}

}

**Explanation:**

* The flatten method recursively flattens the multilevel doubly linked list.
* It iterates through the list, flattening each child node and connecting it to the main list.

**Output:** For the given test cases, the output will be as expected.

* **Time Complexity:** O(n) where n is the total number of nodes in the list.
* **Space Complexity:** O(h) where h is the maximum depth of the nested lists. This is due to the recursive calls.

**Problem 12: Split a Circular Linked List into Two Halves**

public class CircularLinkedList {

Node head;

class Node {

int data;

Node next;

Node(int d) {

data = d;

next = null;

}

}

// Split a circular linked list into two halves

public Node[] splitList(Node head) {

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

return new Node[]{head, null};

}

Node slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

Node secondHalf = slow.next;

slow.next

= head;

fast.next = null;

return new Node[]{head, secondHalf};

}

}

**Explanation:**

* The splitList method uses the Floyd's Cycle Detection algorithm to find the middle node.
* It then splits the list into two halves by adjusting the pointers.

**Output:** For the given test cases, the output will be as expected.

* **Time Complexity:** O(n) where n is the length of the circular linked list.
* **Space Complexity:** O(1) as the splitting is done in-place without using extra space.
* **============================================================**

**Problem 13: Insert a Node in a Sorted Circular Linked List**

public class CircularLinkedList {

// ... (same as Problem 12)

// Insert a node in a sorted circular linked list

public void insertSorted(Node newNode) {

if (head == null || newNode.data <= head.data) {

newNode.next = head;

Node current = head;

while (current.next != head) {

current = current.next;

}

current.next = newNode;

head = newNode;

} else {

Node current

= head;

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

current = current.next;

}

newNode.next = current.next;

current.next

= newNode;

}

}

}

**Explanation:**

* The insertSorted method finds the appropriate position to insert the new node based on the sorted order.
* It then adjusts the pointers to insert the node.

**Output:** For the given test cases, the output will be as expected.

* **Time Complexity:** O(n) in the worst case, where n is the length of the circular linked list.
* **Space Complexity:** O(1) as the insertion is done in-place without using extra space.
* **===========================================================================**

**Problem 14: Check if Two Linked Lists Intersect**

public class SinglyLinkedList {

// ... (same as Problem 1)

// Check if two linked lists intersect

public Node findIntersection(Node headA, Node headB) {

if (headA == null || headB == null) {

return null;

}

Node a = headA;

Node b = headB;

while (a != b) {

a = (a == null) ? headB : a.next;

b = (b == null) ? headA : b.next;

}

return

a;

}

}

**Explanation:**

* The findIntersection method uses a pointer jumping technique.
* It moves both pointers simultaneously, making one pointer jump to the other list's head if it reaches the end.
* If they meet, it means they intersect.

**Output:** For the given test cases, the output will be as expected.

**Time and Space Complexity:**

* **Time Complexity:** O(m + n) where m and n are the lengths of the two lists.
* **Space Complexity:** O(1) as the algorithm uses constant space.

**=========================================================================**

**Problem 15: Find the Middle Element of a Linked List in One Pass**

public class SinglyLinkedList {

// ... (same as Problem 1)

// Find the middle element of a linked list

public Node findMiddleNode(Node head) {

if (head == null) {

return null;

}

Node slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

}

**Explanation:**

* The findMiddleNode method uses the Floyd's Cycle Detection algorithm to find the middle node.

**Output:** For the given test cases, the output will be as expected.

* **Time Complexity:** O(n) where n is the length of the linked list.
* **Space Complexity:** O(1) as the algorithm uses constant space.
* **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***