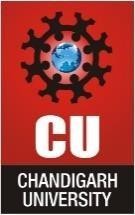
DEPARTMENT OF



COMPUTER SCIENCE & ENGINEERING

**Experiment 3**

**Student Name: Aniket Singh UID: 22BET1073**

**Branch: IT Section/Group: 22BET\_IOT-702/A**

**Semester: 6th Date of Performance: 30.01.25**

**Subject Name: AP Lab - 2 Subject Code: 22ITP-351**

**Problem A**

1. **Aim:**

To implement and understand a program that detects the presence of a cycle in a linked list using Floyd’s Cycle Detection Algorithm (Tortoise and Hare Algorithm).

# Objective:

* + To develop skills in identifying and solving problems involving cyclic linked lists.
  + To improve proficiency in implementing algorithms in Java for data structures.

# Code:

class Solution {

public boolean hasCycle(ListNode head) {

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

return false; // If empty or single node, no cycle

}

ListNode slow = head;

ListNode fast = head;

while (fast != null && fast.next != null) { // Traverse the list

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

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

if (slow == fast) {

return true; // Cycle detected

}

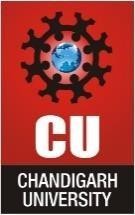
}

return false;

}

}

DEPARTMENT OF



COMPUTER SCIENCE & ENGINEERING

1. **Output:**

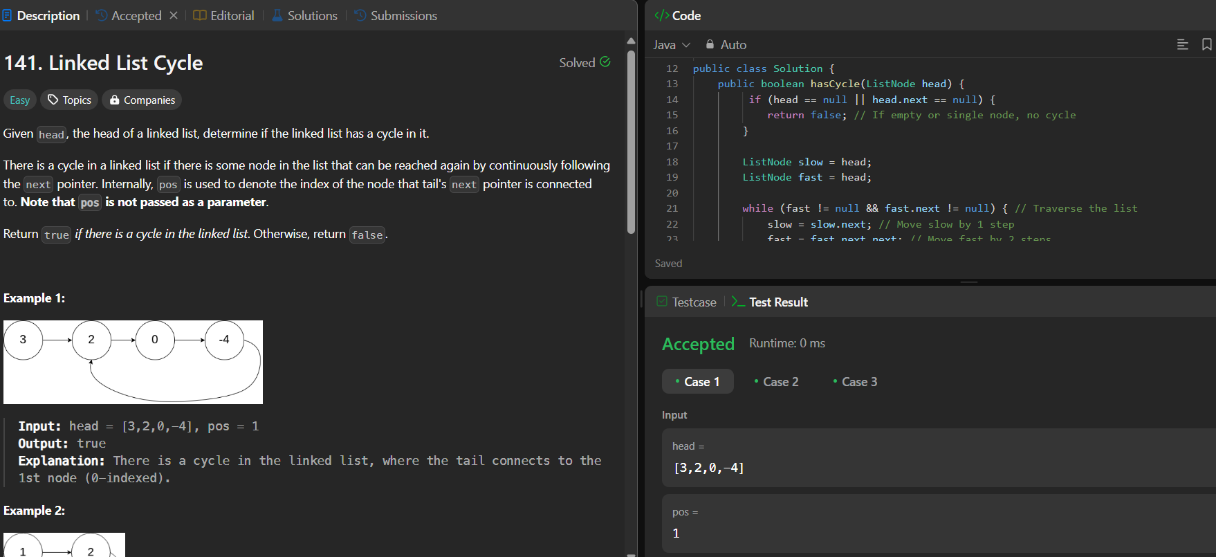


Figure Detect a cycle in a linked list

# Learning Outcomes:

* + Learn to identify and explain the use of Floyd’s Cycle Detection Algorithm for cycle detection in linked lists.
  + Learn to test and debug the code to handle different cases like linked lists with no nodes, a single node, and cyclic linked lists.

# Problem B

1. **Aim:**

To implement a program that reverses a specific portion of a singly linked list between given positions left and right.

# Objective:

* + To learn how to traverse and manipulate a singly linked list using pointers effectively.
  + To develop an understanding of reversing a sublist in-place within a linked list.

# Code:

class Solution {

public ListNode reverseBetween(ListNode head, int left, int right) {

if (head == null || left == right) {

return head; // No reversal needed

}

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy;

// Step 1: Reach the node before the `left` position

for (int i = 1; i < left; i++) {

prev = prev.next;

}

// Step 2: Reverse the sublist

ListNode current = prev.next;

ListNode next = null;

ListNode reverse = null;

for (int i = left; i <= right; i++) {

next = current.next;

current.next = reverse;

reverse = current;

current = next;

}

// Step 3: Reattach the reversed sublist

prev.next.next = current;

prev.next = reverse;

return dummy.next;

}

}

# Output:

# 

Figure Reverse a list

# Learning Outcomes:

* + Learn how to identify and explain the logic for reversing a portion of a singly linked list.
  + learn to manage edge cases like reversing at the beginning or end of the linked list.

# Problem C

1. **Aim:** To implement a program that rotates a singly linked list to the right by a given number of places, ensuring efficient handling of edge cases and large inputs.

# Objective:

# To understand the concept of list rotations and their implementation in a singly linked list.

# To practice in-place list modifications, including forming and breaking circular connections in linked lists.

# Code:

# class Solution {

# public ListNode rotateRight(ListNode head, int k) {

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

# return head; // Handle edge cases

# }

# // Step 1: Compute the length of the list

# ListNode tail = head;

# int length = 1;

# while (tail.next != null) {

# tail = tail.next;

# length++;

# }

# // Step 2: Reduce k to k % length

# k = k % length;

# if (k == 0) {

# return head; // No rotation needed

# }

# // Step 3: Form a circular linked list

# tail.next = head;

# // Step 4: Find the new tail and new head

# int stepsToNewTail = length - k;

# ListNode newTail = head;

# for (int i = 1; i < stepsToNewTail; i++) {

# newTail = newTail.next;

# }

# ListNode newHead = newTail.next;

# // Step 5: Break the circle

# newTail.next = null;

# return newHead;

# }

# }

# Output:

# 

Figure Rotate a list

# Learning Outcomes:

* + I will be able to explain and implement the logic for rotating a linked list.
  + I will gain proficiency in managing circular connections and efficiently adjusting pointers in a linked list.
  + learn to handle various edge cases in linked list operations effectively.

**Problem D**

1. **Aim:** To implement an efficient algorithm that merges kkk sorted linked lists into a single sorted linked list using priority queues or divide-and-conquer techniques.
2. **Objective:**

* To understand the concept of merging multiple sorted linked lists into one sorted list.
* To explore and implement different approaches for solving the problem, including:
* Using a min-heap (priority queue) for efficient merging.
* Using a divide-and-conquer strategy to iteratively merge pairs of lists.

1. **Code:**

class Solution {

public ListNode mergeKLists(ListNode[] lists) {

if (lists == null || lists.length == 0) return null;

// Min-Heap to store ListNodes, sorted by value

PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) -> a.val - b.val);

// Add the head of each non-null linked list into the priority queue

for (ListNode list : lists) {

if (list != null) {

minHeap.offer(list);

}

}

// Dummy node to track the merged linked list

ListNode dummy = new ListNode(-1);

ListNode current = dummy;

// Process nodes until the heap is empty

while (!minHeap.isEmpty()) {

ListNode smallest = minHeap.poll(); // Extract the smallest node

current.next = smallest; // Append it to the merged list

current = current.next; // Move the pointer

// If there's a next node, push it into the heap

if (smallest.next != null) {

minHeap.offer(smallest.next);

}

}

return dummy.next;

}

}

1. **Output:**

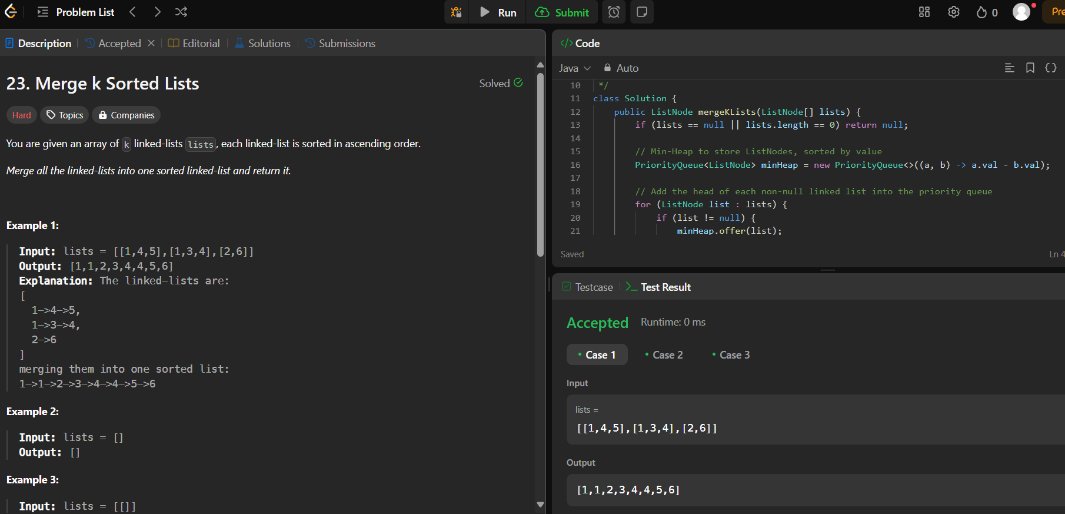
****

Figure Merge K sorted list

1. **Learning Outcome:**

* I will be able to merge multiple sorted linked lists into a single sorted list using efficient algorithms.
* I will learn the application of data structures like priority queues and their role in optimizing solutions.
* I will develop an understanding of the divide-and-conquer approach in solving recursive problems.

**Problem E**

1. **Aim:** To implement an algorithm to sort a singly linked list in ascending order using efficient sorting techniques like merge sort or quick sort.
2. **Objective:**

* To understand the challenges and strategies for sorting a linked list compared to arrays.
* To learn and implement sorting algorithms suitable for linked lists, such as merge sort, which is optimal for linked lists.

1. **Code:**

class Solution {

public ListNode sortList(ListNode head) {

if (head == null || head.next == null) return head; // Base case

// Step 1: Split the list into two halves

ListNode mid = getMid(head);

ListNode rightHead = mid.next;

mid.next = null; // Break the list into two parts

// Step 2: Recursively sort both halves

ListNode left = sortList(head);

ListNode right = sortList(rightHead);

// Step 3: Merge the two sorted halves

return merge(left, right);

}

private ListNode getMid(ListNode head) {

ListNode slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

return slow; // Middle node

}

private ListNode merge(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(-1);

ListNode current = dummy;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

current.next = l1;

l1 = l1.next;

} else {

current.next = l2;

l2 = l2.next;

}

current = current.next;

}

if (l1 != null) current.next = l1;

if (l2 != null) current.next = l2;

return dummy.next;

}

}

1. **Output:**

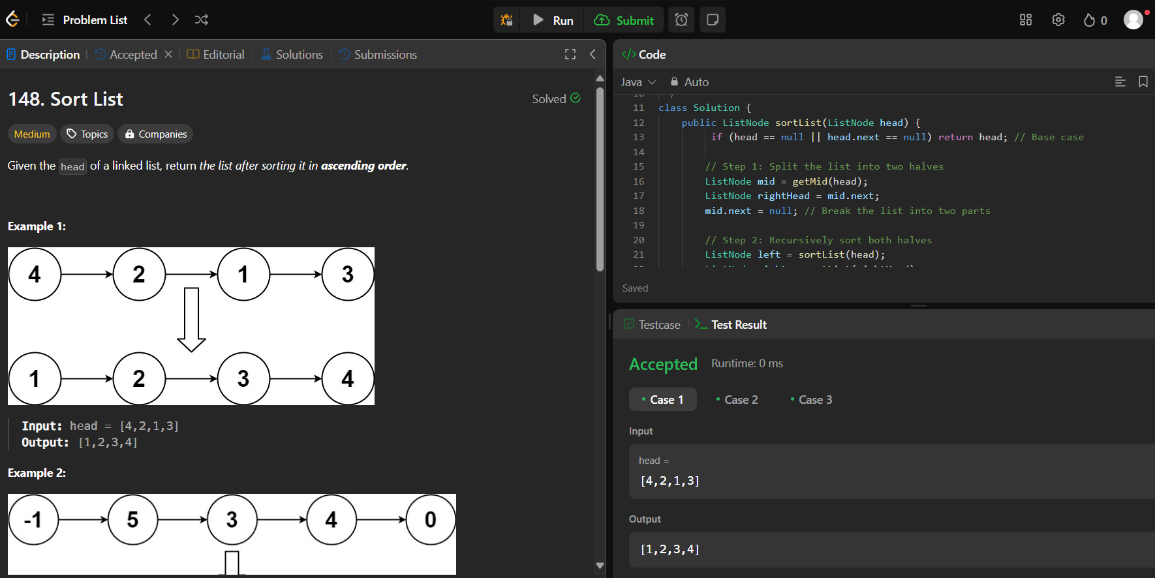
****

Figure Sort lis

1. **Learning Outcome:**

* I will understand how to sort a singly linked list using divide-and-conquer techniques.
* I will gain practical experience in implementing recursive algorithms like merge sort for linked lists.
* I will learn to manipulate pointers in a linked list efficiently to achieve sorting without creating additional structures.