|  |
| --- |
| **Experiment No : 3** |

|  |  |
| --- | --- |
| **Name : Manjiri Chavande** | **DSE SY COMPS** |
| **Division: B** | **Batch : A** |

|  |
| --- |
| * **Problem Statement** |

1. Create a Linked List ADT using the Struct 'Node'.
2. Detect origin of linked list cycle

|  |
| --- |
| * **Theory** |

**A linked list** is a linear data structure that consists of a sequence of elements, where each element points to the next one in the sequence.

1. **Nodes:** The fundamental building blocks of a linked list are nodes. Each node contains two parts: the data (or value) and a reference (or pointer) to the next node in the sequence.
2. **Head:** The first node in a linked list is known as the "head." It serves as the entry point to the list.
3. **Tail:** The last node in a linked list is called the "tail." It is the node whose next reference points to null, indicating the end of the list.
4. **Singly Linked List:** In a singly linked list, each node points only to the next node in the sequence, forming a unidirectional chain.
5. **Dynamic Size:** Linked lists can easily grow or shrink in size during runtime as nodes can be added or removed without the need for contiguous memory allocation.
6. **Random Access:** Unlike arrays, linked lists do not support direct or random access to elements; you must traverse from the head to find a specific node.
7. **Efficient Insertion and Deletion:** Linked lists excel at insertion and deletion operations, as they

**Detecting the origin of a cycle** in a linked list is a critical algorithmic problem known as Floyd's Cycle Detection Algorithm or the "Tortoise and Hare" algorithm:

1. **Purpose:** The goal is to determine if a linked list contains a cycle (loop) and, if so, to find the node where the cycle begins.
2. **Approach:** Two pointers, often called "slow" and "fast," traverse the list. The slow pointer advances one step at a time, while the fast pointer moves two steps at a time.
3. **Cycle Detection:** If there is a cycle, the fast pointer will eventually catch up to the slow pointer within the cycle. This is due to the relative speed difference between them.
4. **Identification:** Once the two pointers meet inside the cycle, they are at the same distance from the cycle's origin. To find the origin, reset the slow pointer to the head and move both pointers one step at a time until they meet again; this meeting point is the cycle's origin.

|  |
| --- |
| * **Algorithm** |

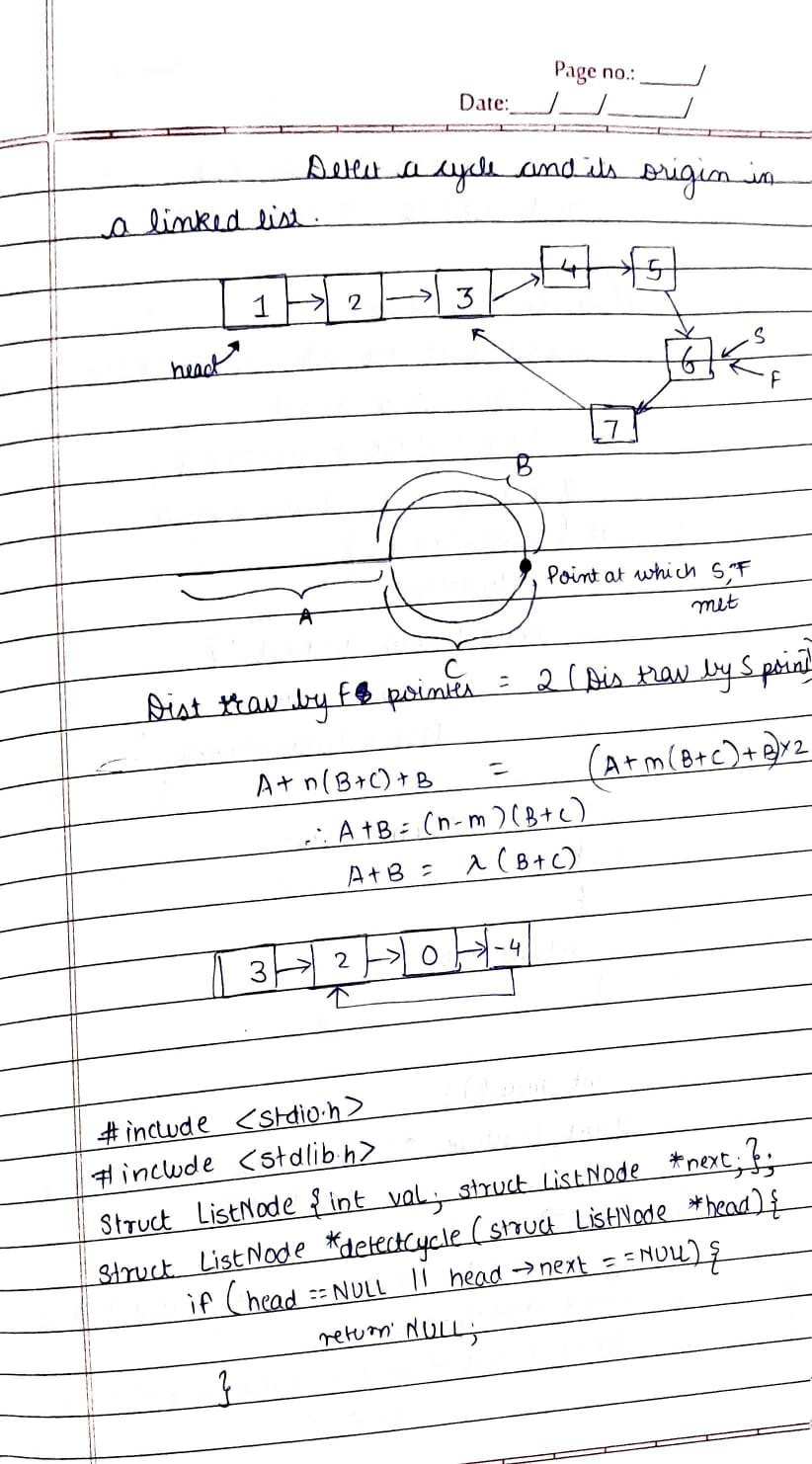
***Algorithm for Linked List Operations***

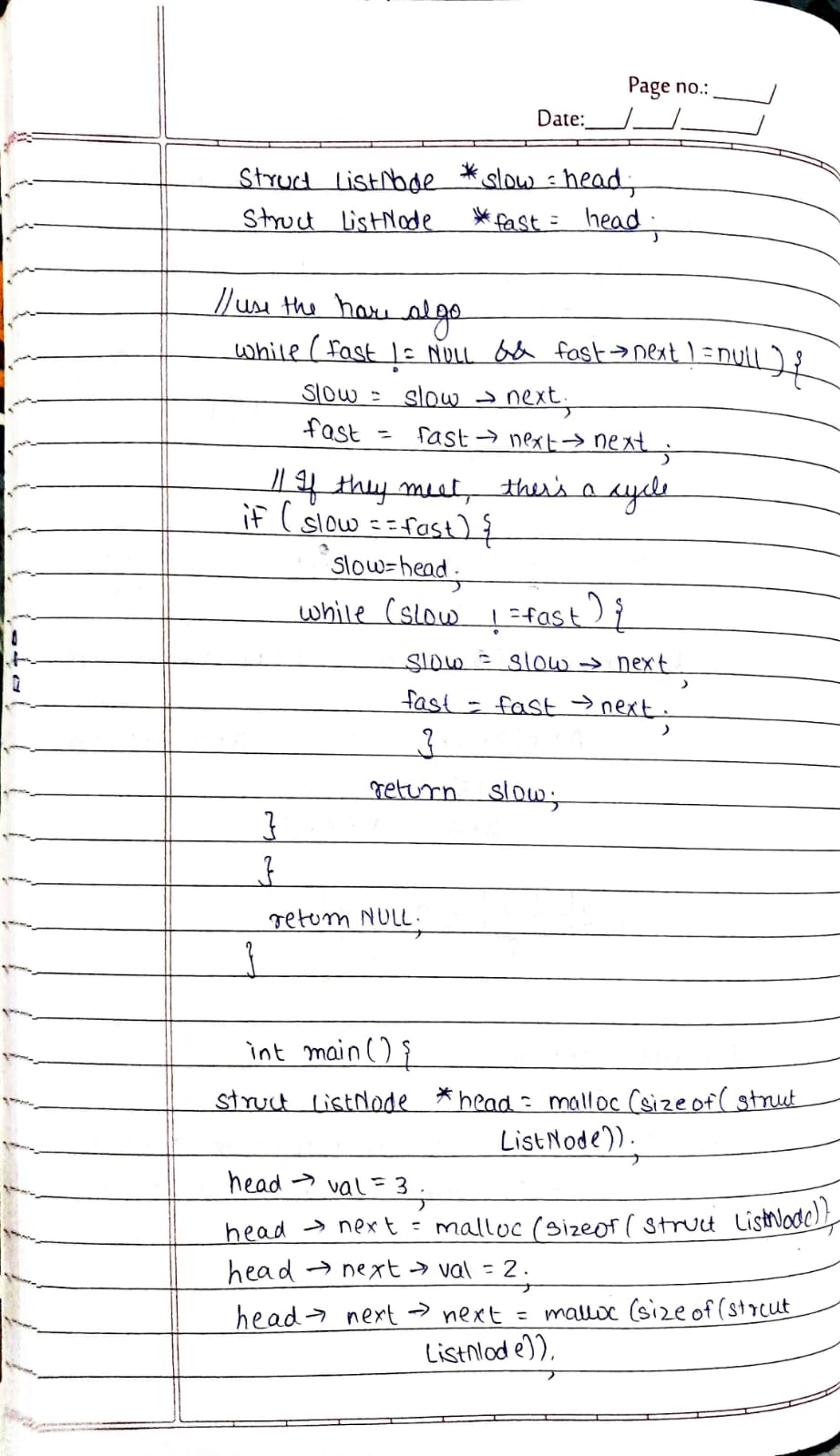
1. **Create Node (create\_node):**
   * Create a new node with the given data.
   * Set its 'next' pointer to NULL.
   * Return the new node.
2. **Insert Node at Position (insert\_at\_pos):**
   * Create a new node with the provided data.
   * If **pos** is 0, make the new node the new head node and point its 'next' to the old head.
   * Otherwise, traverse the list to find the node at **pos - 1**.
   * Insert the new node between the found node and the next node.
3. **Delete Node at Position (delete\_at\_pos):**
   * If **pos** is 0 and the list is not empty, point the head to the second node and free the old head.
   * Otherwise, traverse to find the node at **pos - 1**.
   * Adjust pointers to bypass the node at **pos** and free it.
4. **Delete by Value (delete\_by\_value):**
   * Traverse the list to find the first node with the given value.
   * Adjust pointers to bypass the found node and free it.
5. **Get Node at Position (get\_node\_at\_pos):**
   * Start from the first node and traverse **pos** nodes.
   * Return the data of the node at **pos**.
   * If the position is not found, return NULL.
6. **Find First (find\_first):**
   * Start from the first node and traverse until a node with the given value is found.
   * Return the first node with the specified value.
   * If not found, return NULL.
7. **Display (display):**
   * Traverse the list from the head.
   * Print each node's data.
8. **Free Linked List (free\_linkedlist):**
   * Start from the head and traverse the list.
   * Free the memory of each node.
9. **Reverse Linked List (reverse):**
   * Initialize three pointers: **prev**, **current**, and **next**.
   * Start from the head's next node.
   * For each node, update the 'next' pointer to reverse the list.
   * Set the head's next pointer to the new head (formerly the last node).

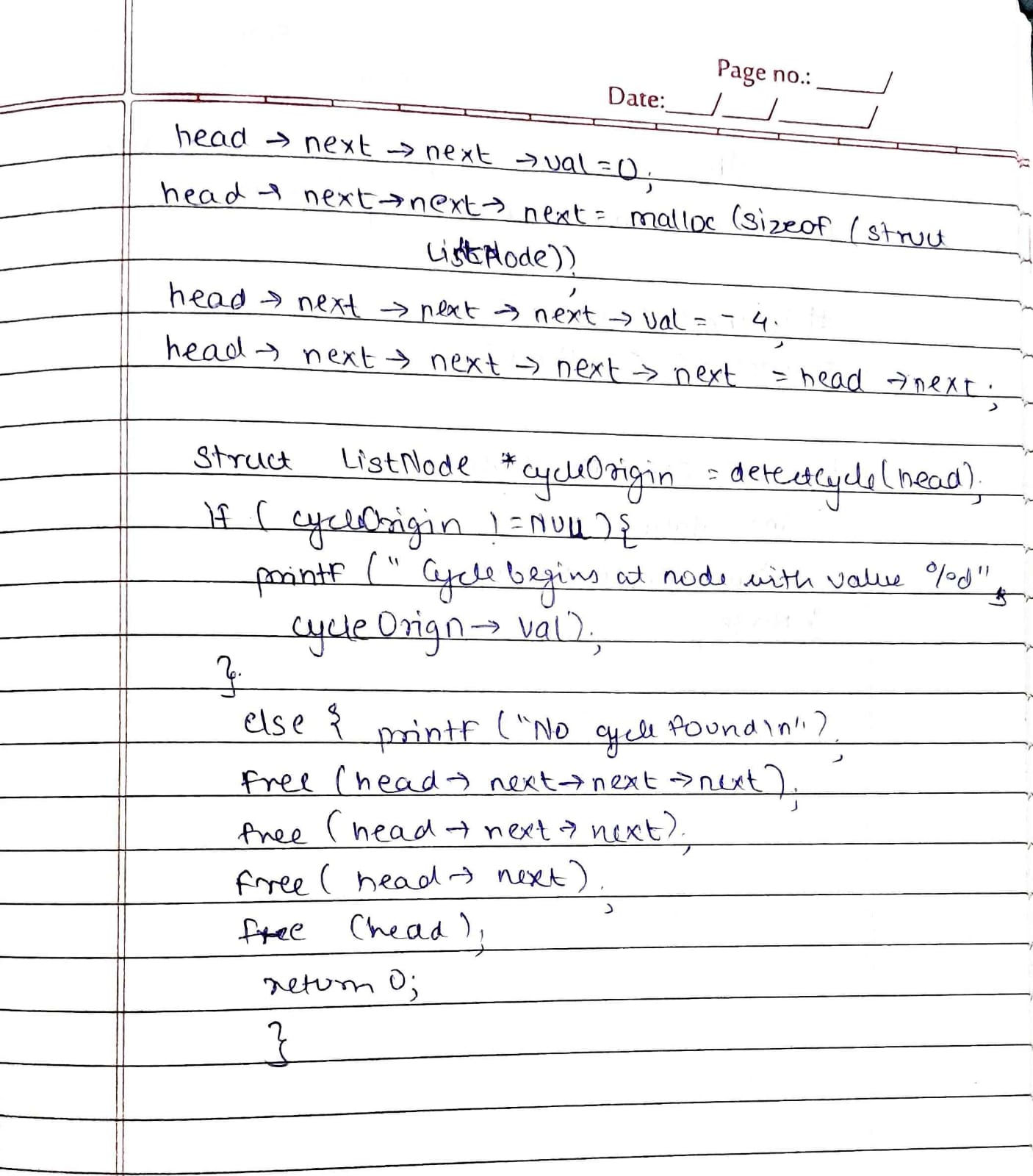
***Algorithm for Detecting Origin of Linked List Cycle***

1. **Initialize Pointers:**
   * Initialize two pointers, **slow** and **fast**, both pointing to the head of the linked list.
   * Check if the list is empty or has only one node. If true, return NULL since there's no cycle.
2. **Detect Cycle (Tortoise and Hare Algorithm):**
   * Use the Tortoise and Hare algorithm to detect a cycle in the linked list.
   * While both **fast** and **fast->next** are not NULL (meaning they have not reached the end of the list):
     + Move the **slow** pointer one step at a time.
     + Move the **fast** pointer two steps at a time.
     + If **slow** and **fast** meet at the same node, there's a cycle in the list.
3. **Find Cycle Origin:**
   * Reset one of the pointers, e.g., **slow**, to the head of the list.
4. **Move Both Pointers to Find Origin:**
   * Move both **slow** and **fast** pointers one step at a time until they meet again.
   * The node where they meet is the origin of the cycle.
5. **Return Origin Node:**
   * Return the node where the cycle begins.
   * Print the value of the node where the cycle begins if a cycle is detected; otherwise, print "No cycle found."
6. **Memory Cleanup:**
   * Free the allocated memory for the linked list nodes.

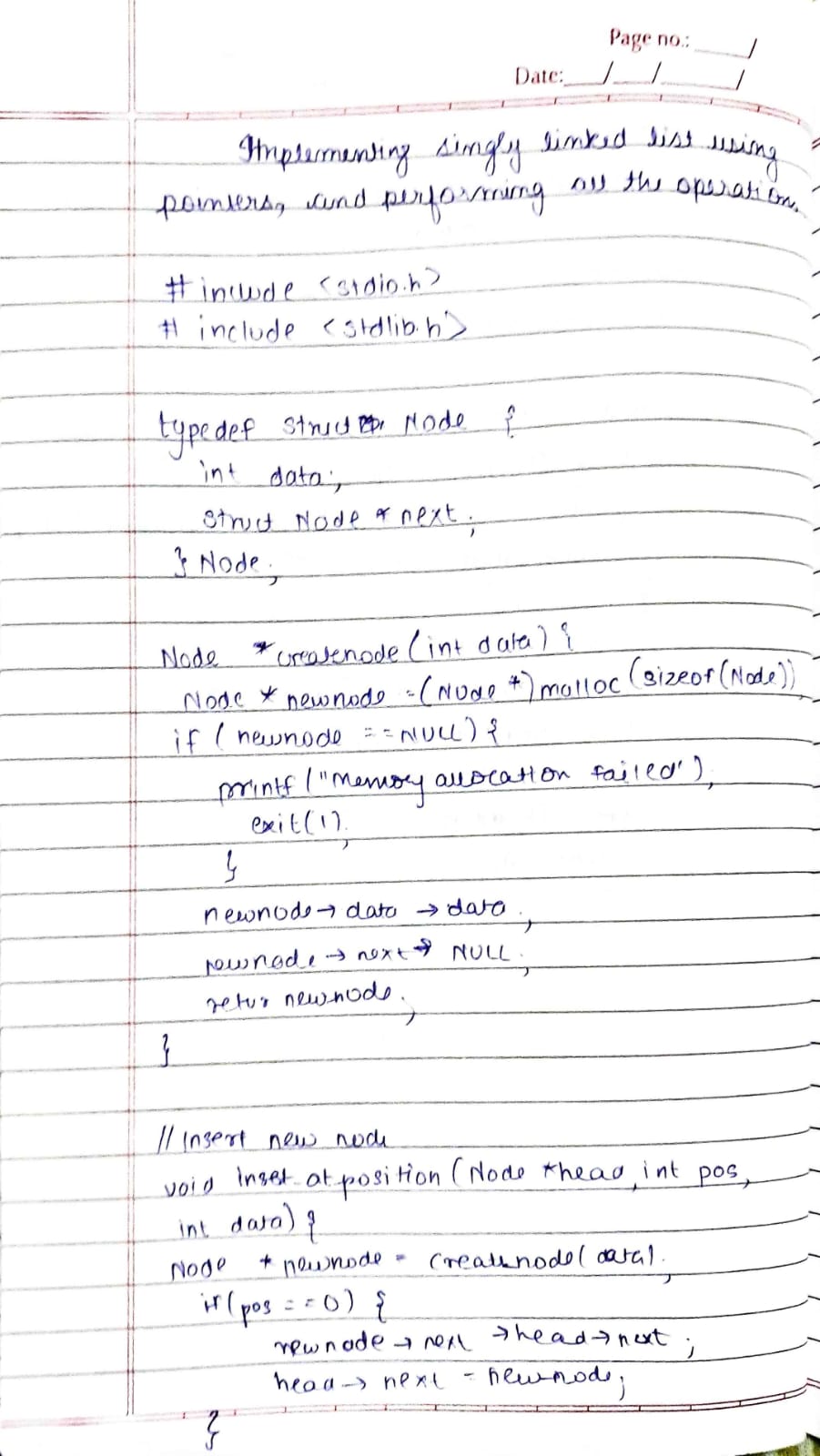
|  |
| --- |
| * **Solution** |

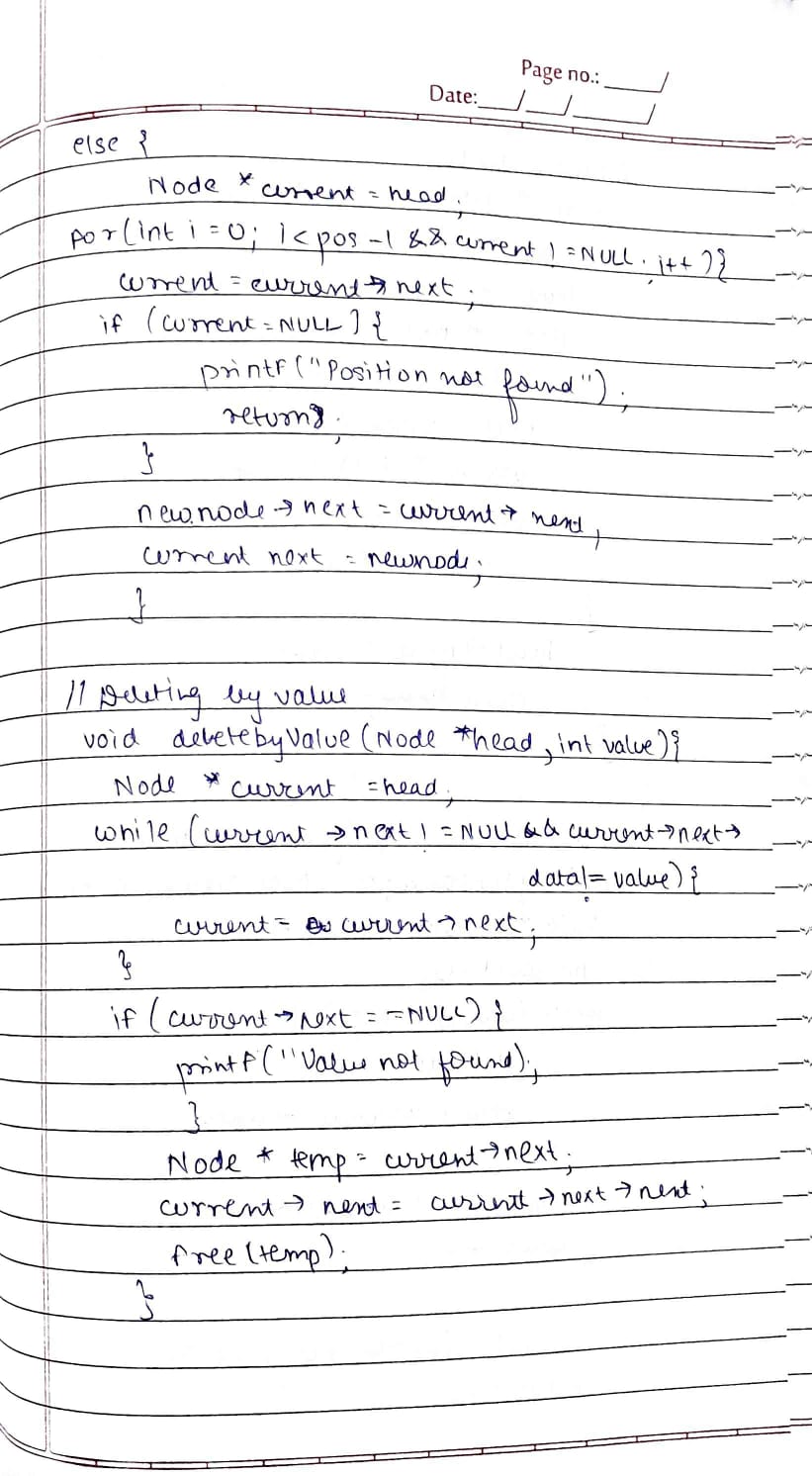


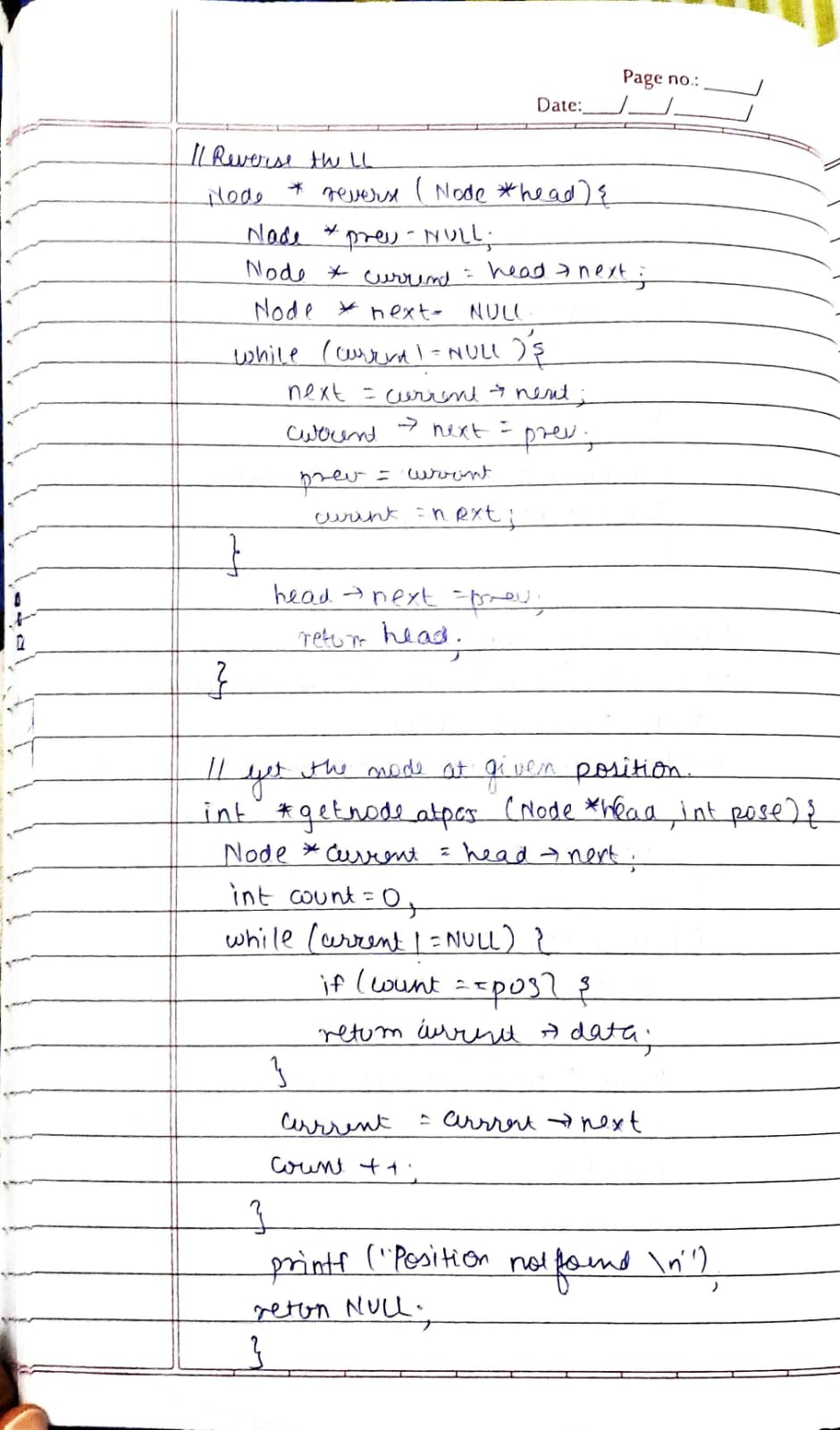


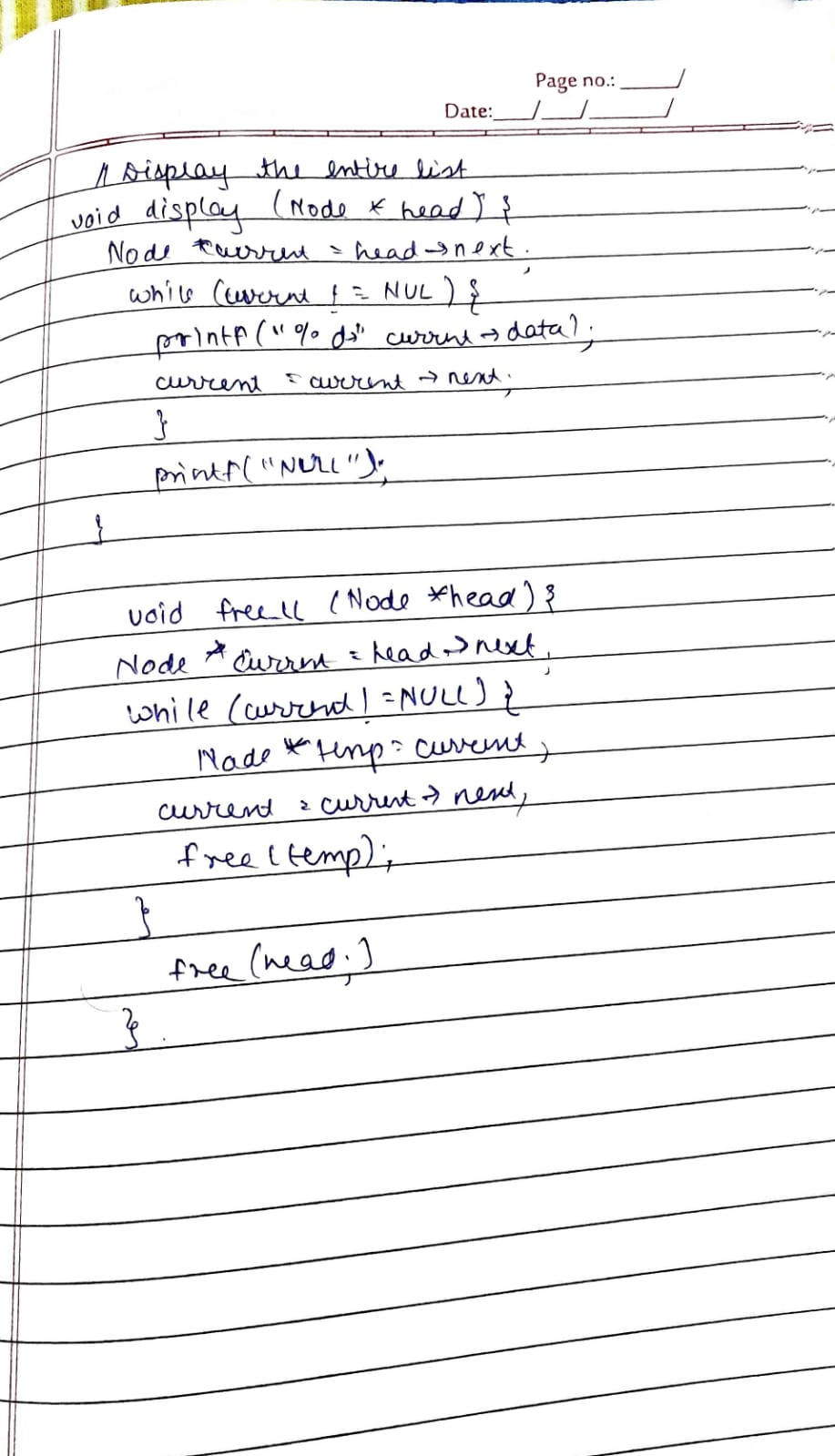


1.2









|  |
| --- |
| * **Code /Output and test cases** |

***/\****

***\* File: linkedlist.c***

***\* Author: Manjiri Chavande***

***\* Email: cmanjiri1912@gmail.com***

***\* Created: September 18, 2023***

***\* Description: i.  Create a Linked List ADT using the Struct 'Node'.***

***\*/***

***#include* <stdio.h>**

***#include* <stdlib.h>**

**typedef struct Node**

**{**

**int data;**

**struct Node \*next;**

**} Node;**

***// Create a new linked list node with 'data'***

**Node \*create\_node(int data)**

**{**

**Node \*new\_node = (Node \*)malloc(sizeof(Node));**

***if* (new\_node == NULL)**

**{**

**printf("Memory allocation failed.\n");**

**exit(1);**

**}**

**new\_node->data = data;**

**new\_node->next = NULL;**

***return* new\_node;**

**}**

***// Insert a new node at the given position in the linked list***

**void insert\_at\_pos(Node \*head, int pos, int data)**

**{**

**Node \*new\_node = create\_node(data);**

***if* (pos == 0)**

**{**

**new\_node->next = head->next;**

**head->next = new\_node;**

**}**

***else***

**{**

**Node \*current = head;**

***for* (int i = 0; i < pos - 1 && current != NULL; i++)**

**{**

**current = current->next;**

**}**

***if* (current == NULL)**

**{**

**printf("Position not found.\n");**

**free(new\_node);**

***return*;**

**}**

**new\_node->next = current->next;**

**current->next = new\_node;**

**}**

**printf("Node with data %d inserted successfully\n", data);**

**}**

***// Delete a node at the given position in the linked list***

**void delete\_at\_pos(Node \*head, int pos)**

**{**

***if* (pos == 0 && head->next != NULL)**

**{**

**Node \*temp = head->next;**

**head->next = head->next->next;**

**free(temp);**

**}**

***else***

**{**

**Node \*current = head;**

***for* (int i = 0; i < pos - 1 && current != NULL; i++)**

**{**

**current = current->next;**

**}**

***if* (current->next == NULL)**

**{**

**printf("Position not found.\n");**

***return*;**

**}**

**Node \*temp = current->next;**

**current->next = current->next->next;**

**free(temp);**

**}**

**}**

***// Delete the first occurrence of a node with the given value in the linked list***

**void delete\_by\_value(Node \*head, int value)**

**{**

**Node \*current = head;**

***while* (current->next != NULL && current->next->data != value)**

**{**

**current = current->next;**

**}**

***if* (current->next == NULL)**

**{**

**printf("Value not found.\n");**

***return*;**

**}**

**Node \*temp = current->next;**

**current->next = current->next->next;**

**free(temp);**

**}**

***// Get the node at the given position in the linked list***

**int \*get\_node\_at\_pos(Node \*head, int pos)**

**{**

**Node \*current = head->next;**

**int count = 0;**

***while* (current != NULL)**

**{**

***if* (count == pos)**

**{**

***return* current->data;**

**}**

**current = current->next;**

**count++;**

**}**

**printf("Position not found.\n");**

***return* NULL;**

**}**

***// Find the first node with the given value in the linked list***

**Node \*find\_first(Node \*head, int value)**

**{**

**Node \*current = head->next;**

***while* (current != NULL)**

**{**

***if* (current->data == value)**

**{**

***return* current;**

**}**

**current = current->next;**

**}**

**printf("Value not found.\n");**

***return* NULL;**

**}**

***// Display the entire linked list***

**void display(Node \*head)**

**{**

**Node \*current = head->next;**

***while* (current != NULL)**

**{**

**printf("%d -> ", current->data);**

**current = current->next;**

**}**

**printf("NULL\n");**

**}**

***// Deallocate the memory used by the linked list***

**void free\_linkedlist(Node \*head)**

**{**

**Node \*current = head->next;**

***while* (current != NULL)**

**{**

**Node \*temp = current;**

**current = current->next;**

**free(temp);**

**}**

**free(head);**

**}**

***// Reverse the linked list***

**Node \*reverse(Node \*head)**

**{**

**Node \*prev = NULL;**

**Node \*current = head->next;**

**Node \*next = NULL;**

***while* (current != NULL)**

**{**

**next = current->next;**

**current->next = prev;**

**prev = current;**

**current = next;**

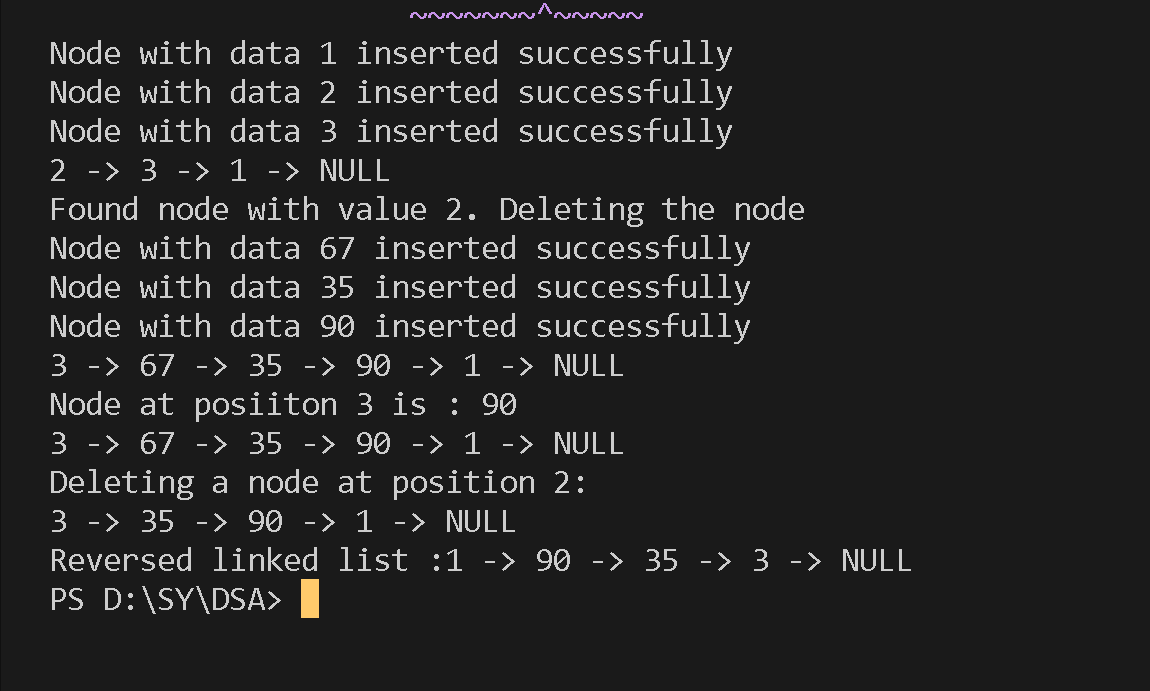
**}**

**head->next = prev;**

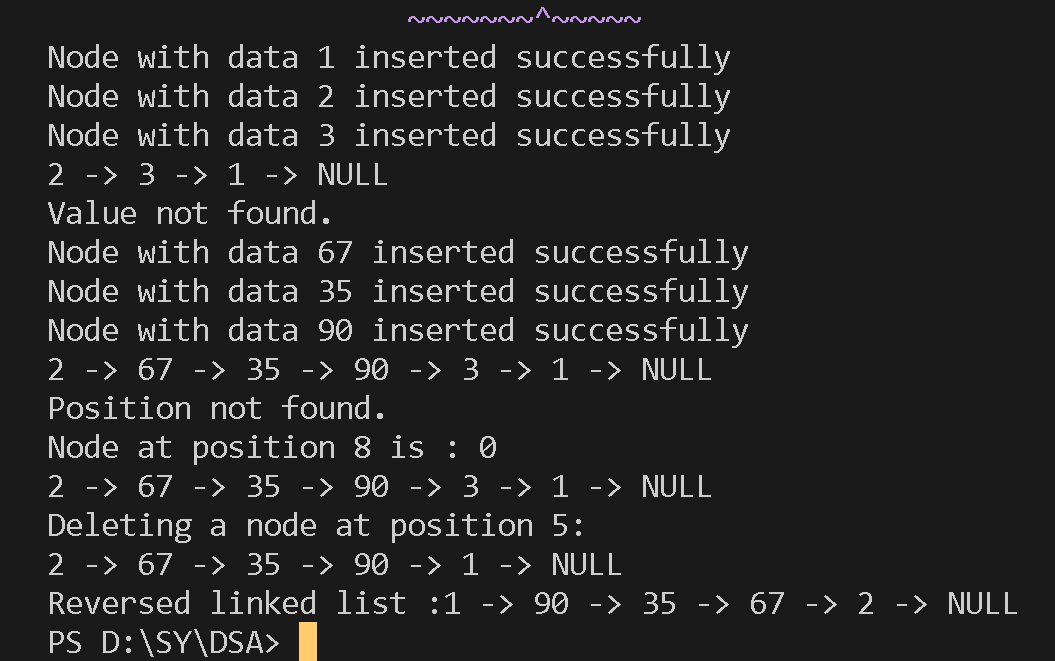
***return* head;**

**}**

* 1. Expected /Ideal Output



* 1. Output when we add the incorrect position values for a) finding the element by position b) deleting a node by position.



***/\****

***\* File: detectOrigin.c***

***\* Author: Manjiri Chavande***

***\* Email: cmanjiri1912@gmail.com***

***\* Created: September 18, 2023***

***\* Description: ii.  Detect origin of linked list cycle***

***\*/***

***#include* <stdio.h>**

***#include* <stdlib.h>**

***// Define the structure for a linked list node***

**struct ListNode {**

**int val;**

**struct ListNode \*next;**

**};**

***// Function to detect the origin of a linked list cycle***

**struct ListNode \*detectCycle(struct ListNode \*head) {**

***if* (head == NULL || head->next == NULL) {**

***// No cycle if there are less than two nodes***

***return* NULL;**

**}**

**struct ListNode \*slow = head;**

**struct ListNode \*fast = head;**

***// Use the Tortoise and Hare algorithm***

***while* (fast != NULL && fast->next != NULL) {**

**slow = slow->next; *// Move slow pointer by one step***

**fast = fast->next->next; *// Move fast pointer by two steps***

***// If they meet, there's a cycle***

***if* (slow == fast) {**

***// Reset one pointer to the head***

**slow = head;**

***// Move both pointers one step at a time until they meet again***

***while* (slow != fast) {**

**slow = slow->next;**

**fast = fast->next;**

**}**

***return* slow; *// Return the node where the cycle begins***

**}**

**}**

***return* NULL;**

**}**

**int main() {**

**struct ListNode \*head = malloc(sizeof(struct ListNode));**

**head->val = 3;**

**head->next = malloc(sizeof(struct ListNode));**

**head->next->val = 2;**

**head->next->next = malloc(sizeof(struct ListNode));**

**head->next->next->val = 0;**

**head->next->next->next = malloc(sizeof(struct ListNode));**

**head->next->next->next->val = -4;**

**head->next->next->next->next = head->next; *// Create a cycle***

**struct ListNode \*cycleOrigin = detectCycle(head);**

***if* (cycleOrigin != NULL) {**

**printf("Cycle begins at node with value %d\n", cycleOrigin->val);**

**} *else* {**

**printf("No cycle found\n");**

**}**

***// Clean up memory***

**free(head->next->next->next);**

**free(head->next->next);**

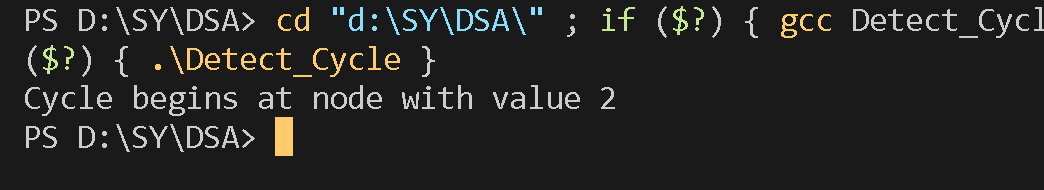
**free(head->next);**

**free(head);**

***return* 0;**

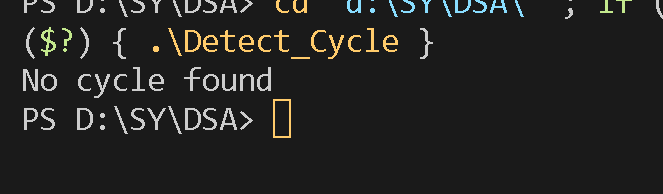
**}**

Expected Output: The origin is 2nd node.



When there is no cycle present

**head->next->next->next->next = NULL;**



|  |
| --- |
| * **Conclusion** |

Creating a Linked List Abstract Data Type (ADT) using the 'Node' struct allows you to build and manipulate linked lists efficiently .Detecting the origin of a linked list cycle is essential for identifying the point where a cycle begins, providing a critical tool for solving cycle-related problems in linked lists. This is typically achieved using algorithms like the Tortoise and Hare algorithm.