

# Laboratory Manual

**Subject:** Data Structures Lab

**Semester:** III2025-26

**Class:** S. Y. B.Tech Computer Engineering

**Experiment No.: 01**

---

## Title

Implementation of Menu Driven Program for Linked List Operations

---

## Aim

To implement menu-driven program for linked list operations.

---

## Software/Hardware Requirements

- Operating System: Windows/DOS
  - Compiler: Turbo C / GCC
  - System: Any standard PC
- 

A **linked list** is a linear collection of data elements called **nodes**, where each node contains two parts:

1. **DATA** → Stores information (can be integer, character, or structure).
2. **NEXT** → Pointer to the next node in sequence.

The last node contains a special value **NULL**, indicating the end of the list.

A linked list is a **self-referential data type** since each node contains a pointer to another node of the same type.

In memory representation, a special pointer variable **START** is used to store the address of the first node.

- If **START = NULL**, then the list is empty.
- We traverse the list by repeatedly following the **NEXT** pointers.

Example structure of a node in C:

```
struct node {  
    int data;  
    struct node *next;  
};
```

A linked list supports various operations such as **insertion, deletion, and traversal (display)**.

---

## Operations on Singly Linked List

### 1. Insertion

Insertion can be done in three ways:

- At the beginning of the list
- At the end of the list
- At a specific location in the list

### 2. Deletion

Deletion can be done in three ways:

- From the beginning of the list
- From the end of the list
- A specific node from the list

### 3. Display

To display a linked list:

1. Check if the list is empty (`head == NULL`).
  2. If empty → print “**List is Empty!**”.
  3. If not empty → start from the first node (`temp = head`) and traverse until `NULL`.
  4. Print each node’s data with an arrow (`-->`).
  5. End with **NULL**.
- 

## Algorithms

### Algorithm 1: Insertion at Beginning

1. Start.
2. Create a new node.

- 
3. Assign data to the node.
  4. Set `newnode->next = head`.
  5. Update `head = newnode`.
  6. Stop.
- 

### **Algorithm 2: Insertion at End**

1. Start.
  2. Create a new node.
  3. Assign data to the node, set `newnode->next = NULL`.
  4. If `head == NULL`, set `head = newnode`.
  5. Else, traverse to the last node and set `last->next = newnode`.
  6. Stop.
- 

### **Algorithm 3: Insertion at Specific Location**

1. Start.
  2. Create a new node.
  3. Traverse the list to  $(\text{position}-1)$ th node.
  4. Update pointers:
    - o `newnode->next = temp->next`
    - o `temp->next = newnode`
  5. Stop.
- 

### **Algorithm 4: Deletion from Beginning**

1. Start.
  2. If `head == NULL`, print “**List is Empty**”.
  3. Else, store `temp = head`.
  4. Update `head = head->next`.
  5. Free `temp`.
  6. Stop.
- 

### **Algorithm 5: Deletion from End**

1. Start.
2. If `head == NULL`, print “**List is Empty**”.
3. Else, traverse list to second last node.
4. Free last node and set `secondlast->next = NULL`.
5. Stop.

---

### **Algorithm 6: Deletion of Specific Node**

1. Start.
  2. If `head == NULL`, print “**List is Empty**”.
  3. Else, traverse the list to find the node with given value.
  4. Update pointer of previous node: `prev->next = current->next`.
  5. Free current node.
  6. Stop.
- 

### **Algorithm 7: Display Linked List**

1. Start.
2. If `head == NULL`, print “**List is Empty**”.
3. Else, set `temp = head`.
4. While `temp != NULL`:
  - o Print `temp->data --->`.
  - o Move `temp = temp->next`.
5. Print **NULL**.
6. Stop.