UNIT - 2
DATA STRUCTURE
QUESTION BANK

1) Define Linked list?

Ans). A **Linked List** is a **linear data structure** that consists of a **sequence of elements (nodes)**, where each node contains two parts:

1. **Data:** The value stored in the node.
2. **Pointer (Next):** A reference to the next node in the sequence.

2) What are the different types of linked list?

Ans). In **data structures**, linked lists are categorized into the following main types:

1. **Singly Linked List (SLL)**
2. Doubly Linked List (DLL)
3. Circular Linked List (CLL)
4. Circular Doubly Linked List

3) Define the Singly list?

A **Singly Linked List** consists of nodes where:

- Each node contains **two parts**:
  Data → Stores the value.
  Next pointer → Points to the next node.
- The **last node** points to NULL, indicating the end of the list.
- **Traversal is one-way** (from head to tail).

Structure and Example

[Data|Next] → [Data|Next] → [Data|Next] → NULL

4) Define the Doubly linked list?

A **Doubly Linked List** consists of nodes where:

- Each node contains **three parts**:
  - **Prev pointer** → Points to the previous node.
  - **Data** → Stores the value.
  - **Next pointer** → Points to the next node.
- Allows **bi-directional traversal**.

Structure and Example

NULL ← [Prev|Data|Next] ↔ [Prev|Data|Next] → NULL

5) Define the Circular Linked list?

A **Circular Linked List** is a variation where:

- The **last node** points back to the **first node**, forming a circle.
- Can be either **singly circular** or **doubly circular**.

Structure and Example

[Data|Next] → [Data|Next] → [Data|Next] → (points back to first node)

6) Singly linked list advantages and dis advantages?

Advantages:

7) Simple and easy to implement.
8) Efficient for **insertion and deletion** operations.

Disadvantages:

9) Cannot traverse backward.
10) Accessing a specific node requires traversal from the beginning.

7) Doubly Linked List advantages and dis advantages?

Advantages:

- **Bi-directional traversal** (both forward and backward).
- Easier to **delete nodes** (as it contains a reference to the previous node).

Disadvantages:

- Uses **extra memory** for storing an additional pointer.
- Slightly more complex compared to singly linked lists.

8) Circular linked list advantage and dis advantages?

Advantages:

- Can be traversed **continuously** in a loop.
- Useful for **round-robin scheduling**.

Disadvantages:

- More complex to implement.
- Risk of infinite loops if not handled properly.

9). How can represent in singly linked list?

Representation of Singly Linked List in Data Structure

In a **Singly Linked List (SLL)**:

- Each **node** contains two parts:
    - **Data** → Holds the value of the node.
    - **Next Pointer** → Points to the **next node** in the list.
- The **last node** points to NULL, indicating the end of the list.

10). How can represent in doubly linked list?

Representation of Doubly Linked List in Data Structure

A **Doubly Linked List (DLL)** is a **linear data structure** where:

- Each **node** contains **three parts**:
    1. **Prev pointer** → Points to the **previous node**.
    2. **Data** → Holds the value.
    3. **Next pointer** → Points to the **next node**.
- The **first node's previous pointer** is NULL (start of the list).
- The **last node's next pointer** is NULL (end of the list).
- It allows **bi-directional traversal**.

11). How can represent in circular linked list?

Representation of Circular Linked List in Data Structure

A **Circular Linked List (CLL)** is a **linear data structure** where:

- Each **node** contains **two parts**:
    1. **Data** → Holds the value.
    2. **Next pointer** → Points to the **next node** in the list.
- The **last node's next pointer** points to the **first node**, forming a **circular connection**.
- It can be:
    - **Singly Circular Linked List** → Only next pointer exists.
    - **Doubly Circular Linked List** → Both next and prev pointers exist.

12). What are the operation of doubly linked list?

A **Doubly Linked List (DLL)** supports various operations, including:

- **Insertion**
- **Deletion**
- **Traversal**
- **Searching**
- **Reversing the list**

13). Difference between array and linked list?

Array	Linked List
A **fixed-size** data structure that stores elements in **contiguous memory locations**.	A **dynamic-size** data structure consisting of **nodes** with data and pointers.
static allocation: Memory is allocated at **compile-time**.	**Dynamic allocation**: Memory is allocated at **runtime**.
Fixed size → Cannot be easily resized.	**Dynamic size** → Can grow or shrink in memory.
Direct access using the index → O(1) time complexity.	**Sequential access** → Traverses nodes one by one O(n).
Faster → Random access is possible.	**Slower** → Sequential access only.

14). Applications of linked list?

1.Dynamic Memory Management
2. Implementing Stack and Queue
3. Implementing Graphs
4. Polynomial Manipulation
5. Circular Lists in Gaming

15). illustrate the use of the linked list?

Ans). To **illustrate** the use of linked lists, let's go through **real-world scenarios** and visualize how they work.

1). Linked List in Memory Management (Operating System).
2). Music Playlist Navigation.
3). Web Browser History.
4). Stack Implementation Using Linked List.
5). Graph Representation Using Linked List.

16). How can insert the elements in doubly linked list?

Ans). Insertion of Elements in Doubly Linked List (DLL)

In a **doubly linked list**, each node contains:

- **Data**
- A pointer to the **next node**
- A pointer to the **previous node**

Insertion operations in a **Doubly Linked List** can be performed in three ways:

1. **At the beginning**
2. **At the end**
3. **At a specific position.**

17). Difference between singly linked list and doubly linked list?

Singly Linked list	Doubly Linked list
A list where each node contains **data** and a pointer to the **next node**.	A list where each node contains **data**, a pointer to the **next node**, and a pointer to the **previous node**..
Each node contains: data → next.	Each node contains: data → next → prev
One-directional → Can be traversed only forward**.**	Two-directional → Can be traversed forward and backward.
Requires less memory → Only one pointer per node.	Requires more memory → Two pointers per node.
Simpler to implement due to only one pointer.	More complex to implement due to two pointers.

18). Difference between array and circular list?

Array	Circular Linked list
A collection of **contiguous memory locations**.	A linked list where the **last node points to the first node**, forming a **circular structure**.
Static memory allocation → Fixed size.	**Dynamic** memory allocation → Flexible size.
Requires **continuous memory** blocks.	Nodes can be scattered in **different memory locations**.
Linear traversal → Moves from the first to the last element.	**Circular** traversal → Moves continuously in a loop.
The last element points to **NULL**.	The last node points to the **first node**.

5- & 10-MARKS QUESTIONS

DATA STRUCTURE
UNIT – 2

1). Application of Linked list?
Ans). Here are some of the key applications:

1.Dynamic Memory Allocation

- **Linked lists** are used for dynamic memory management, where memory is allocated or deallocated during program execution.
- In **C language**, malloc () and free () functions internally use linked lists to manage the heap memory.

2. Implementation of Stack and Queue

- **Stacks** can be implemented using a **singly linked list**.
- **Queues** are often implemented using a **doubly linked list** for efficient insertion and deletion operations at both ends.
- **Dequeue (Double-ended queue)** can be easily implemented using doubly linked lists.

3.Managing Directories and File Systems

- **Operating systems** use linked lists to manage files and directories.
- Each file or directory is represented as a node containing the name and the address of the next file or directory.
- **File Allocation Table (FAT)** in operating systems uses linked lists for memory allocation.

4. Implementing Hash Tables

- **Chaining in hash tables** uses linked lists to handle collisions.
- When multiple keys hash to the same index, a linked list is used to store the keys at that index.

5. Undo and Redo Operations

- Applications like **text editors** use linked lists to implement **undo and redo** operations.
- Each change is stored as a node in the linked list, allowing you to traverse back and forth.

6.Graph Representation

- **Adjacency lists** in graphs use linked lists to store the neighbours of each node.
- This is an efficient way to represent sparse graphs.

7.Polynomial Arithmetic

- Linked lists are used to represent and manipulate **polynomials**.
- Each node represents a term with coefficients and exponents.

2) Explain the Singly linked list?
Ans). A **Singly Linked List (SLL)** is a type of linked list where:

- Each **node** contains **two parts**:
  - **Data**: The actual value stored in the node.
  - **Pointer (Next)**: The address of the next node in the sequence.
- The **last node** points to NULL, indicating the end of the list.

Structure of a Singly Linked List
[Data | Next] → [Data | Next] → [Data | Next] → NULL

Example:
10 → 20 → 30 → 40 → NULL
0 is the first node (head), pointing to the next node (20).
20 points to **30**, and so on.
40 is the last node, pointing to NULL.

Basic Operations on Singly Linked List

Insertion
Insertion in a singly linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

🔷 **Insertion at the Beginning**

- Create a new node.
- Point the new node's next to the current head.
- Update the head to the new node.

Example:
Original List: 20 → 30 → 40 → NULL
Insert: 10 at the beginning
New List: 10 → 20 → 30 → 40 → NULL
Insertion at the End

- Create a new node.
- Traverse the list to reach the last node.
- Point the last node's next to the new node.

Example:'
Original List: 10 → 20 → 30 → NULL
Insert: 40 at the end
New List: 10 → 20 → 30 → 40 → NULL

Insertion at a Specific Position
Create a new node.
Traverse the list until the desired position.
Point the new node's next to the current node at the position.
Update the previous node's next to the new node.

Example: Insert 25 at position 2 in the list 10 → 20 → 30 → NULL

New List: 10 → 20 → 25 → 30 → NULL.

Deletion
Deletion in a singly linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Deletion at the Beginning

- Update the head to point to the next node.
- Free the memory of the original head.

Example:
Original List: 10 → 20 → 30 → NULL
Delete the first node
New List: 20 → 30 → NULL

Deletion at the End

- Traverse the list to find the **second last node**.
- Update the next of the second last node to NULL.

Example:
Original List: 10 → 20 → 30 → NULL
Delete the last node
New List: 10 → 20 → NULL

Deletion at a Specific Position

- Traverse the list to reach the node before the target node.
- Update the next pointer of the previous node to skip the target node.

Example: Delete node at position 2 in 10 → 20 → 30 → 40 → NULL
New List: 10 → 20 → 40 → NULL

Advantages of Singly Linked List

1. **Dynamic Size:** It allows dynamic memory allocation, making it easy to add or remove nodes.
2. **Efficient Insert/Delete:** Insertion and deletion are faster compared to arrays, especially at the beginning.
3. **Memory Utilization:** It uses memory efficiently by allocating space only when required.

Disadvantages of Singly Linked List

1. **No Backward Traversal:** You can only traverse the list in **one direction**.
2. **Extra Memory Usage:** Each node uses extra memory to store the pointer.
3. **Random Access Not Possible:** You must traverse the list sequentially to access a node, unlike arrays that allow direct access by index.

3) Explain the Doubly linked list?
A **Doubly Linked List (DLL)** is a type of linked list where:

- Each **node** contains **three parts**:
  - **Data**: The actual value stored in the node.
  - **Pointer to the next node (next)**.
  - **Pointer to the previous node (prev)**.
- It allows **traversal in both directions** (forward and backward).

Structure of a Doubly Linked List
[Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ NULL
Example:
NULL ← 10 ↔ 20 ↔ 30 ↔ 40 → NULL

- **10** is the head node with prev = NULL and next pointing to **20**.
- **20** points back to **10** and forward to **30**.
- **40** is the last node with next = NULL.

Basic Operations on Doubly Linked List:
Insertion
Insertion in a doubly linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Insertion at the Beginning

- Create a new node.
- Point the next of the new node to the current head.
- Update the prev of the current head to the new node.
- Update the head pointer to the new node.

Example:
Original List: 20 ↔ 30 ↔ 40 → NULL
Insert: 10 at the beginning
New List: 10 ↔ 20 ↔ 30 ↔ 40 → NULL

Insertion at the End

- Create a new node.
- Traverse the list to reach the **last node**.
- Point the next of the last node to the new node.
- Set the prev of the new node to the last node.
- Update the next of the new node to NULL.

Example:
Original List: 10 ↔ 20 ↔ 30 → NULL
Insert: 40 at the end
New List: 10 ↔ 20 ↔ 30 ↔ 40 → NULL

Insertion at a Specific Position

- Create a new node.
- Traverse the list to the **target position**.
- Update the pointers:
    - Set the next of the new node to the current node at the position.
    - Set the prev of the new node to the previous node.
    - Adjust the prev and next pointers of the surrounding nodes.

Example: Insert 25 at position 2 in the list 10 ↔ 20 ↔ 30 → NULL
New List: 10 ↔ 20 ↔ 25 ↔ 30 → NULL

Deletion

Deletion in a doubly linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Deletion at the Beginning

- Update the head pointer to the **next** node.
- Set the prev pointer of the new head to NULL.
- Free the memory of the removed node.

Example:
Original List: 10 ↔ 20 ↔ 30 → NULL
Delete the first node
New List: 20 ↔ 30 → NULL

Deletion at the End

- Traverse the list to the **last node**.

- Update the next of the **second last node** to NULL.
- Free the memory of the removed node.

Example:
Original List: 10 ↔ 20 ↔ 30 → NULL
Delete the last node
New List: 10 ↔ 20 → NULL

Deletion at a Specific Position

- Traverse the list to the **target position**.
- Adjust the pointers:
    - Set the next of the previous node to the next of the target node.
    - Set the prev of the next node to the prev of the target node.
- Free the memory of the removed node.

Example: Delete node at position 2 in 10 ↔ 20 ↔ 30 ↔ 40 → NULL
New List: 10 ↔ 20 ↔ 40 → NULL

Advantages of Doubly Linked List

1. **Bidirectional Traversal:** You can traverse the list in **both directions** (forward and backward).
2. **Efficient Deletion:** Easier to delete a node compared to singly linked list.
3. **Insertion and Deletion:** Faster compared to arrays, especially in the middle.
4. **No Need for Previous Pointer:** Unlike singly linked list, you can directly access the previous node.

Disadvantages of Doubly Linked List

1. **Extra Memory Usage:** Requires extra memory for the prev pointer in each node.
2. **More Complex Operations:** Insertion and deletion involve more pointer adjustments.
3. **Higher Memory Consumption:** Compared to singly linked lists, DLL uses more memory.

4) Explain the Circular linked list?

A **Circular Linked List (CLL)** is a variation of the linked list where:

- **The last node** points back to the **first node**, forming a circle.
- Unlike singly and doubly linked lists, circular linked lists **do not have NULL pointers** at the end.
- It can be **singly** or **doubly circular**:
    - **Singly Circular Linked List:** Each node points to the next node, and the last node points back to the head.
    - **Doubly Circular Linked List:** Each node has two pointers (next and prev), and the last node points back to the head, while the head's prev points to the last node.

Structure of a Circular Linked List:
[Data | Next] → [Data | Next] → [Data | Next] → (Back to Head)

Example:
10 → 20 → 30 → 40 → (Back to 10)

- **10** is the head, pointing to **20**.
- **40** is the last node, pointing back to **10**, forming a circular structure.

Doubly Circular Linked List
[Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ (Circular Connection)
Example:
NULL ←→ 10 ←→ 20 ←→ 30 ←→ 40 ←→ (Back to 10)
10 is the head node with prev pointing to **40** and next pointing to **20**.
40 is the last node, pointing back to **10**, forming a circle.

Basic Operations on Circular Linked List:

Insertion

Insertion in a circular linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Insertion at the Beginning

- Create a new node.
- Point the new node's next to the current head.
- Traverse to the **last node** and update its next to point to the new node.
- Update the head to the new node.

Example:

Original List: 20 → 30 → 40 → (Back to 20)

Insert: 10 at the beginning

New List: 10 → 20 → 30 → 40 → (Back to 10)

Insertion at the End

- Create a new node.
- Traverse the list to reach the **last node**.
- Point the next of the last node to the new node.
- Set the next of the new node to the head, forming a circle.

Example:

Original List: 10 → 20 → 30 → (Back to 10)

Insert: 40 at the end

New List: 10 → 20 → 30 → 40 → (Back to 10)

Insertion at a Specific Position

- Create a new node.
- Traverse the list to the **target position**.
- Adjust the next pointers:
    - Set the next of the new node to the next node at the position.
    - Set the next of the previous node to the new node.

Example: Insert 25 at position 2 in the list 10 → 20 → 30 → (Back to 10)

New List: 10 → 20 → 25 → 30 → (Back to 10)

Deletion

Deletion in a circular linked list can occur at:

- **Beginning**
- **End**
- **Specific Position**

Deletion at the Beginning

- Update the head to the **next node**.
- Traverse to the **last node** and update its next to the new head.
- Free the memory of the removed node.

Example:
Original List: 10 → 20 → 30 → (Back to 10)
Delete the first node
New List: 20 → 30 → (Back to 20)

- Traverse the list to reach the **second last node**.
- Update the next of the second last node to point to the head.
- Free the memory of the removed node.

Example:
Original List: 10 → 20 → 30 → (Back to 10)
Delete the last node
New List: 10 → 20 → (Back to 10)

Deletion at a Specific Position

- Traverse the list to the **target position**.
- Adjust the pointers:
    - Set the next of the previous node to the next node of the target node.
- Free the memory of the removed node.

Example: Delete node at position 2 in 10 → 20 → 30 → 40 → (Back to 10)
New List: 10 → 20 → 40 → (Back to 10)

Advantages of Circular Linked List

1. **Efficient Circular Traversal:** You can traverse the entire list from any point, making it useful for applications that require continuous looping.
2. **Dynamic Size:** No need to define the size in advance.
3. **Efficient Insertion/Deletion:** Insertions and deletions are efficient, especially at the **beginning** and **end**.

Disadvantages of Circular Linked List

1. **Complex Traversal:** Traversal requires special handling to avoid infinite loops.
2. **More Complex Implementation:** Requires extra logic to manage the circular link.
3. **No direct access:** Unlike arrays, random access is not possible.

5). Comparing Arrays and linked list?

Array	Linked list
A collection of elements stored in **contiguous memory locations**.	A collection of **nodes** where each node contains data and a pointer/reference to the next (or previous) node.
Static memory allocation (fixed size).	**Dynamic memory allocation (size can grow or shrink).**
Direct/Random access (O(1) time complexity).	**Sequential access** (O(n) time complexity).
Inserting/deleting elements requires shifting subsequent elements (O(n) time complexity).	**Efficient insertion/deletion (O(1) if position is known).**
May lead to **wasted memory** if the size is over-allocated.	Memory efficient with no unused space (but requires extra space for pointers).
Fixed size (defined at declaration).	Flexible size (grows and shrinks dynamically).
Elements are stored in **contiguous** memory locations.	Nodes are scattered across memory with **pointers** linking them.

6). write a program to reverse an array in c language

```
#include <stdio.h>

int main() {
   int n, i, temp;

   // Input the size of the array
   printf("Enter the size of the array: ");
   scanf("%d", &n);

   int arr[n];

   // Input array elements
   printf("Enter %d elements of the array:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   // Reversing the array using two-pointer technique
   for (i = 0; i < n / 2; i++) {
      temp = arr[i];
      arr[i] = arr[n - i - 1];
      arr[n - i - 1] = temp;
   }

   // Display the reversed array
   printf("\nReversed array:\n");
   for (i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");

   return 0;
};
```

OUTPUT:
Enter the size of the array: 5
Enter 5 elements of the array:
1 2 3 4 5

Reversed array:
5 4 3 2 1

7) write algorithm for insert and delete a node from doubly linked list ?
Ans : Algorithm for Insertion and Deletion in a Doubly Linked List:
1. Insertion in a Doubly Linked List
There are three cases for inserting a node:

- **At the beginning**

- **At the end**
- **At a specific position**

Algorithm for Inserting at the Beginning

1. **Create a new node**.
2. Set new_node → next = head.
3. If the list is not empty:
    - Set head → prev = new_node.
4. Update head = new_node.

Time Complexity: O(1)

Algorithm for Inserting at the End

1. **Create a new node**.
2. If the list is empty:
    - Set head = new_node.
3. Otherwise:
    - Traverse the list to the last node.
    - Set last → next = new_node.
    - Set new_node → prev = last.

Time Complexity: O(n)

Algorithm for Inserting at a Specific Position

1. **Create a new node**.
2. If inserting at the beginning:
    - Follow the **beginning insertion algorithm**.
3. Otherwise:
    - Traverse the list to the desired position (pos).
    - Set:
        - new_node → next = current → next
        - new_node → prev = current
    - Update the pointers:
        - current → next → prev = new_node
        - current → next = new_node

Time Complexity: O(n)

2. Deletion in a Doubly Linked List

There are three cases for deleting a node:

- **Delete the first node**
- **Delete the last node**
- **Delete a node at a specific position**

Algorithm for Deleting the First Node

1. **Check if the list is empty:**
    - If head == NULL, return.
2. Set temp = head.
3. Update head = head → next.
4. If head != NULL:
    - Set head → prev = NULL.
5. Free the temp node.

Time Complexity: O(1)

Algorithm for Deleting the Last Node

1. **Check if the list is empty:**
    - If head == NULL, return.
2. Traverse to the last node.
3. Set second_last → next = NULL.
4. Free the last node.

Time Complexity: O(n)

Algorithm for Deleting a Node at a Specific Position

1. **Check if the list is empty**:
    - If head == NULL, return.
2. Traverse the list to the desired position (pos).
3. Update the pointers:
    - current → prev → next = current → next
    - current → next → prev = current → prev
4. Free the current node.

Time Complexity: O(n)

8) Specify the use of Header node in a linked list?

Use of Header Node in a Linked List

A **Header Node** is a special node at the **beginning of a linked list** that does not contain any actual data but acts as a placeholder or reference point. It typically stores **meta-information** (e.g., list size) or serves as a pointer to the first node in the list.

Purpose of Header Node

1. **Simplifies List Operations:**
    - Insertion and deletion operations become simpler as you don't need to handle special cases for the first node separately.
    - The header node acts as a consistent starting point.
2. **Improved Traversal Efficiency:**
    - The header node provides a single reference point, making traversal and searching easier and more structured.
3. **Storage of Meta-Information:**
    - It can store the **size of the list**, making size-related operations efficient.
    - It may also contain a pointer to the last node, improving the efficiency of tail operations.
4. **Consistency in List Operations:**
    - The header node makes **empty list handling** easier by providing a consistent reference point, even when the list has no data nodes.

Types of Header Nodes

1. **Dummy Header Node:**
    - A node with no significant data, used purely for simplifying operations.
    - It always points to the first data node.
2. **Sentinel Header Node:**
    - A special marker node used to indicate the **end of the list** (in circular linked lists).
    - It simplifies the termination condition in traversals.

Advantages of Using a Header Node

- Simplifies **insertion and deletion** operations by avoiding special cases for the first node.
- Improves **consistency** and readability of the linked list operations.
- Makes **list management** easier by storing meta-data like size or end pointers.

9). create a doubly linked list by inserting following elements in a list 13,45,23,20,25

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a doubly linked list node
struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->prev = NULL;
   newNode->next = NULL;
   return newNode;
}

// Function to insert a node at the end of the doubly linked list
void insertEnd(struct Node** head, int data) {
   struct Node* newNode = createNode(data);

   if (*head == NULL) {
      *head = newNode;  // If the list is empty, make the new node the head
      return;
   }

   struct Node* temp = *head;

   // Traverse to the last node
   while (temp->next != NULL) {
      temp = temp->next;
   }

   // Insert the new node at the end
   temp->next = newNode;
   newNode->prev = temp;
}

// Function to display the doubly linked list
void display(struct Node* head) {
   struct Node* temp = head;

   printf("\nDoubly Linked List: ");
   while (temp != NULL) {
      printf("%d ", temp->data);
```

```
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    struct Node* head = NULL;

    // Inserting the elements into the doubly linked list
    insertEnd(&head, 13);
    insertEnd(&head, 45);
    insertEnd(&head, 23);
    insertEnd(&head, 20);
    insertEnd(&head, 25);

    // Display the doubly linked list
    display(head);

    return 0;
};
```

Explanation

1. **Node Structure Definition:**
    - data: To store the value of the node.
    - prev: Pointer to the previous node.
    - next: Pointer to the next node.
2. **Functions Used:**
    - createNode(int data) → Allocates memory for a new node and assigns the value.
    - insertEnd(struct Node** head, int data) → Inserts the node at the **end** of the doubly linked list.
    - display(struct Node* head) → Prints the doubly linked list.
3. **Insertion Process:**
    - The program inserts 13 → 45 → 23 → 20 → 25 sequentially.
    - The nodes are linked together in both directions using next and prev pointers.

Output :

Doubly Linked List: 13 45 23 20 25

10). explain the insertion operation in single linked list. How nodes are inserted after a specified node?

Insertion Operation in a Singly Linked List

The **insertion operation** in a singly linked list involves adding a new node into the list. The operation can be performed in three cases:

1. **At the beginning**
2. **At the end**
3. **After a specified node**

1. Inserting a Node at the Beginning

- **Steps:**
    1. Create a new node.
    2. Assign the next pointer of the new node to the current head.

3. Update the head pointer to the new node.

- **Time Complexity:** O(1)

Inserting a Node at the End

- **Steps:**
  1. Create a new node.
  2. Traverse the list until you reach the last node.
  3. Set the next pointer of the last node to the new node.
  4. Set the next of the new node to NULL.
- **Time Complexity:** O(n)

Inserting a Node After a Specified Node

- This is the most common type of insertion where you add a new node **after a given node**.

Algorithm for Inserting After a Specified Node

1. **Create a New Node**
   - Allocate memory for the new node.
   - Assign the data value to the new node.
2. **Traverse the List**
   - Start from the head and find the specified node.
3. **Insert the New Node**
   - Set:
     - new_node → next = specified_node → next
     - specified_node → next = new_node

Time Complexity: O(n) in the worst case (if the specified node is at the end). **Space Complexity:** O(1) for the new node.

C Program to Insert After a Specified Node in a Singly Linked List:

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a singly linked list node
struct Node {
   int data;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}

// Function to insert a node at the end
void insertEnd(struct Node** head, int data) {
   struct Node* newNode = createNode(data);

   if (*head == NULL) {
      *head = newNode;  // If the list is empty, make the new node the head
      return;
```

```
    }

    struct Node* temp = *head;

    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = newNode;
}

// Function to insert a new node after a specified node
void insertAfter(struct Node* prev_node, int data) {
    if (prev_node == NULL) {
        printf("The given previous node cannot be NULL\n");
        return;
    }

    struct Node* newNode = createNode(data);

    newNode->next = prev_node->next;
    prev_node->next = newNode;
}

// Function to display the linked list
void display(struct Node* head) {
    struct Node* temp = head;

    printf("\nSingly Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    struct Node* head = NULL;

    // Insert elements at the end
    insertEnd(&head, 10);
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    insertEnd(&head, 40);

    printf("Original List:");
    display(head);
```

```
    // Inserting after the second node (after 20)
    printf("\nInserting 25 after 20...\n");
    insertAfter(head->next, 25);  // Inserts 25 after 20

    display(head);

    return 0;
};
```

Explanation

1. **Structure Definition:**
    - Each Node contains:
        - data: To store the value.
        - next: Pointer to the next node.
2. **Functions Used:**
    - createNode(int data) → Creates a new node.
    - insertEnd() → Adds a node at the end of the list.
    - insertAfter() → Adds a node **after a specified node**.
    - display() → Prints the list.
3. **Insertion After a Specified Node:**
    - insertAfter(head->next, 25) → Inserts 25 **after the second node** (20).

Output

Singly Linked List: 10 -> 20 -> 30 -> 40 -> NULL

Inserting 25 after 20...
Singly Linked List: 10 -> 20 -> 25 -> 30 -> 40 -> NULL