**LAB-3**

**Q1.** Write a C program to **create a doubly linked list** based on user input and then **display** the contents of the list in forward direction.

**Description:**

* Define a **struct Node** having **data, prev**, and **next** pointers.
* Dynamically allocate memory for each node.
* Maintain both **forward and backward pointers while inserting nodes**.
* After construction, traverse the list from head to end and print each node’s data.

**Hint:**  
Use a **while** or **for** loop to read **n** elements and use two pointers: one for tracking the **head**, another (**temp**) for appending new nodes.

**Q2**. Write a C program to **search for a key value** in an existing doubly linked list and report its presence and position.

**Description:**

* Take the key value from the user.
* **Traverse the list from head and compare each node’s data with the key**.
* If found, print the position (1-based index); else print "Key not found".

**Hint:**  
Maintain a counter while traversing the list. Use **while (temp != NULL)** for traversal

**Q3.** Write a C program to **insert a new node** in the following positions:

* **At the front** of the list
* **At the end** of the list
* **After a node** containing a given key
* **Before a node** containing a given key

**Description:**

* Implement separate functions for each insertion type.
* Use pointer manipulations to maintain proper **prev** and **next** links.

**Hints:**

* For **insertion at front**, update **head** and the **old head's prev**.
* For **insertion at end**, **traverse till the last node and update its next**.
* For **insertion after key**, find the key node and link the new node accordingly.
* For **insertion before key**, **handle the special case where the key is at the head**.

**Q4.** Write a C program to **delete a node** from the following positions:

* **From the beginning** of the list
* **From the end** of the list
* **After a node** containing a given key
* **Before a node** containing a given key

**Description:**

* **Implement four different deletion functions**.
* Take care of edge cases (e.g., deleting from an empty list, deleting the only node).
* Ensure proper memory deallocation using **free()**.

**Hints:**

* For **deleting at beginning**, move **head** to the next node and update **prev** to NULL.
* For **deleting at end**, find the second last node and make its **next** NULL.
* For **deletion after a key**, find the key node and remove its **next** node.
* For **deletion before a key**, find the key node and remove its **prev** node (handle special case if the node is head).