**Practical No. 10**

**Aim:** Write a menu driven program to perform following insertion operations in a single linked list:

i. Insertion at beginning

ii. Insertion at end

iii. Insertion after a given node

iv. Traversing a linked list

**Theory**

A **singly linked list** is a dynamic linear data structure where each element (node) contains data and a pointer to the next node. Unlike arrays, linked lists allow efficient insertion and deletion without shifting elements because nodes can be linked/unlinked by changing pointers. Common operations:

* **Insertion at beginning**: New node becomes the head. O(1).
* **Insertion at end**: Traverse to tail and link new node. O(n).
* **Insertion after a given node**: Locate the target node then insert by pointer update. O(n) to find node.
* **Traversal**: Visit each node from head to tail to display data. O(n).

**Algorithm**

Main loop (menu-driven):

1. Start.
2. Display menu: 1) Insert at beginning 2) Insert at end 3) Insert after a given node 4) Traverse 5) Exit.
3. Read user choice.
4. If choice is 1:
   * Read value.
   * Create new node.
   * Make new node->next = head; head = new node.
5. If choice is 2:
   * Read value.
   * Create new node with next = NULL.
   * If head is NULL, head = new node; else traverse to last node and last->next = new node.
6. If choice is 3:
   * Read target value (after which to insert) and new value.
   * Traverse to find target node. If found, insert new node after it; else print "not found".
7. If choice is 4:
   * Traverse from head and print node data. If empty, print "List is empty".
8. If choice is 5: Exit.
9. Repeat until exit.
10. Stop.

**Program (C)**

#include <stdio.h> // For input/output functions (printf, scanf)

#include <stdlib.h> // For malloc and free

// Define the node structure for singly linked list

struct Node {

int data; // To store integer data of the node

struct Node \*next; // Pointer to the next node in the list

};

// Head pointer for the list, initialized to NULL (empty list)

struct Node \*head = NULL; // Global so all functions can access the list

// Function to create a new node with given value and return its pointer

struct Node\* createNode(int value) {

struct Node \*newNode = (struct Node\*) malloc(sizeof(struct Node)); // Allocate memory

if (newNode == NULL) { // Check for allocation failure

printf("Memory allocation failed\n"); // Inform user

return NULL; // Return NULL to indicate failure

}

newNode->data = value; // Store the value in node's data

newNode->next = NULL; // Initialize next to NULL (no link yet)

return newNode; // Return pointer to newly created node

}

// Insert a node at the beginning of the list

void insertAtBeginning(int value) {

struct Node \*newNode = createNode(value); // Create node for the value

if (newNode == NULL) return; // If allocation failed, exit function

newNode->next = head; // New node points to current head

head = newNode; // Update head to new node

printf("%d inserted at the beginning.\n", value); // Confirmation

}

// Insert a node at the end of the list

void insertAtEnd(int value) {

struct Node \*newNode = createNode(value); // Create node for the value

if (newNode == NULL) return; // If allocation failed, exit

if (head == NULL) { // If list is empty

head = newNode; // New node becomes the head (only node)

} else {

struct Node \*temp = head; // Temporary pointer to traverse the list

while (temp->next != NULL) // Move to the last node

temp = temp->next;

temp->next = newNode; // Link last node to new node

}

printf("%d inserted at the end.\n", value); // Confirmation

}

// Insert a node after the first node that contains target value

void insertAfterNode(int target, int value) {

struct Node \*temp = head; // Start searching from head

while (temp != NULL && temp->data != target) // Find node with data == target

temp = temp->next;

if (temp == NULL) { // If target not found

printf("Node with value %d not found.\n", target); // Inform user

return; // Exit without inserting

}

struct Node \*newNode = createNode(value); // Create new node

if (newNode == NULL) return; // If allocation failed, exit

newNode->next = temp->next; // Link new node to target's next node

temp->next = newNode; // Link target node to new node

printf("%d inserted after %d.\n", value, target); // Confirmation

}

// Traverse the list and print elements

void traverseList() {

if (head == NULL) { // If list is empty

printf("List is empty.\n"); // Inform user

return; // Nothing to traverse

}

struct Node \*temp = head; // Start from head

printf("Linked list elements: ");

while (temp != NULL) { // Visit nodes until end

printf("%d ", temp->data); // Print current node data

temp = temp->next; // Move to next node

}

printf("\n"); // Newline after printing list

}

// Function to free all nodes (cleanup) before program exit

void freeList() {

struct Node \*temp = head; // Start from head

while (temp != NULL) { // While nodes remain

struct Node \*next = temp->next; // Save pointer to next node

free(temp); // Free current node

temp = next; // Move to next node

}

head = NULL; // Ensure head is NULL after cleanup

}

// Main function: menu-driven interface

int main() {

int choice, value, target; // Variables to read menu choice and values

while (1) { // Infinite loop, will break on choice 5

// Display menu options to user

printf("\n--- Singly Linked List Menu ---\n");

printf("1. Insert at beginning\n");

printf("2. Insert at end\n");

printf("3. Insert after a given node\n");

printf("4. Traverse\n");

printf("5. Exit\n");

printf("Enter your choice: ");

if (scanf("%d", &choice) != 1) { // Read user choice and validate

printf("Invalid input. Exiting.\n"); // If input invalid, exit

break;

}

switch (choice) { // Handle user's choice

case 1: // Insert at beginning

printf("Enter value to insert: ");

scanf("%d", &value); // Read value to insert

insertAtBeginning(value); // Call insertion at beginning

break;

case 2: // Insert at end

printf("Enter value to insert: ");

scanf("%d", &value); // Read value to insert

insertAtEnd(value); // Call insertion at end

break;

case 3: // Insert after a given node

printf("Enter target value after which to insert: ");

scanf("%d", &target); // Read target value

printf("Enter value to insert: ");

scanf("%d", &value); // Read new node value

insertAfterNode(target, value); // Call insert-after function

break;

case 4: // Traverse list

traverseList(); // Display list elements

break;

case 5: // Exit program

freeList(); // Clean up allocated memory

printf("Exiting program.\n"); // Inform user

return 0; // Exit main with success

default: // Invalid choice handling

printf("Invalid choice. Please enter 1-5.\n");

}

}

freeList(); // In case loop breaks unexpectedly, clean up

return 0; // End program

}

**Output**

--- Singly Linked List Menu ---

1. Insert at beginning

2. Insert at end

3. Insert after a given node

4. Traverse

5. Exit

Enter your choice: 1

Enter value to insert: 10

10 inserted at the beginning.

--- Singly Linked List Menu ---

1. Insert at beginning

2. Insert at end

3. Insert after a given node

4. Traverse

5. Exit

Enter your choice: 2

Enter value to insert: 30

30 inserted at the end.

--- Singly Linked List Menu ---

1. Insert at beginning

2. Insert at end

3. Insert after a given node

4. Traverse

5. Exit

Enter your choice: 1

Enter value to insert: 5

5 inserted at the beginning.

--- Singly Linked List Menu ---

1. Insert at beginning

2. Insert at end

3. Insert after a given node

4. Traverse

5. Exit

Enter your choice: 3

Enter target value after which to insert: 10

Enter value to insert: 20

20 inserted after 10.

--- Singly Linked List Menu ---

1. Insert at beginning

2. Insert at end

3. Insert after a given node

4. Traverse

5. Exit

Enter your choice: 4

Linked list elements: 5 10 20 30

--- Singly Linked List Menu ---

1. Insert at beginning

2. Insert at end

3. Insert after a given node

4. Traverse

5. Exit

Enter your choice: 5

Exiting program.

**Conclusion**

* Implemented a menu-driven program for singly linked list operations: insertion at beginning, insertion at end, insertion after a given node, and traversal.
* Learnt how dynamic memory allocation (malloc) is used to create nodes and how pointer manipulation links nodes together.
* Observed pointer updates needed for each insertion type and importance of handling edge-cases (empty list, target node not found).
* Cleaned up memory at exit using free, which is good practice to avoid memory leaks.