Q.1) Write a C program to implement INSERT, DELETE and TRAVERSE operations on a Singly linked list.

Algorithm:

Insertion at the Beginning of Singly Linked List:

- Create a new node.
- Set the new node's data.
- Set the new node's next pointer to the current head.
- Set the head pointer to the new node.

Insertion at the End of Singly Linked List:

- Create a new node.
- Set the new node's data.
- Set the new node's next to NULL.
- If the list is empty (head is NULL), set head to the new node.
- If the list is not empty, traverse to the last node.
- Set the last node next to point to the new node.

Insertion at a Specific Position:

- Create a new node.
- Set the new node's data.
- Traverse the list to the node just before the desired position.
- Set the new node's next to the node at the desired position (i.e., the node currently at the next position).
- Set the previous node's next to the new node.

Deletion at the Beginning of Singly Linked List:

- If head == NULL, the list is empty. Print an error message or return.
- Set head = head->next to remove the current head.
- Use free(head) to release the memory of the old head node.

Deletion at the End of Singly Linked List:

- If head == NULL, the list is empty. Print an error message or return.
- If head->next == NULL, it means the list has only one node, so set head = NULL and free the memory.
- Otherwise, traverse the list to find the second-to-last node (i.e., where node->next->next == NULL).

- Set second_to_last->next = NULL to remove the last node.
- Use free(last_node) to release the memory of the removed node.

Deletion at a Specific Position:

- If head == NULL, the list is empty. Print an error message or return.
- If the position to delete is 0 (i.e., the first node), set head = head->next.
- Otherwise, traverse the list to find the node just before the desired position.
- Set previous_node->next = previous_node->next->next to unlink the node at the desired position.
- Use free(node_to_delete) to release the memory of the deleted node.

Traversing into Singly Linked List:

- Initialize a pointer current to the head of the list.
- While current != NULL, print current->data and move to the next node (current = current->next).
- Stop when current == NULL, indicating the end of the list.

<u>Example:</u>

Consider an empty linked list: head -> NULL, where head is a pointer to the first node.

Insertion at the Beginning:

- Insert 104 , List becomes: head -> 104 -> NULL
- Insert 103, List becomes: head -> 103 -> 104 -> NULL
- Insert 102, List becomes: head -> 102 -> 103 -> 104 -> NULL
- Insert 101, List becomes : head -> 101 -> 102 -> 103 -> 104 -> NULL

Insertion at the End: Insert 106 → List becomes: 101 -> 102 -> 103 -> 104 -> 106

Insertion at Position 5: Insert 105 at position 5 → List becomes: 101 -> 102 -> 103 -> 104 -> 105 -> 106

Deletion at the Beginning: Delete the first node (101) → List becomes: 102 -> 103 -> 104 -> 105 -> 106

Deletion at the End: Delete the last node (106) → List becomes: 102 -> 103 -> 104 -> 105
Deletion at Position 3: Delete node at position 3 (104) → List becomes: 102 -> 103 -> 105

Traversing into singly Linked List:

- Input: A singly linked list with nodes: head -> [102] -> [103] -> [105] -> NULL

- Output: Traverse and print all nodes in the list: 1 2 3

Program:

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

// Define the node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert at the beginning

void insertAtBeginning(struct Node** head, int value) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->next = *head;  // Point to the current head

    *head = newNode;  // Update head to the new node

    printf("Inserted %d at the beginning\n", value);
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;  // If the list is empty, new node becomes the head
        printf("Inserted %d at the end\n", value);
        return;
    }

    struct Node* temp = *head;
```

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

    temp->next = newNode;  // Set the last node's next to the new node
    printf("Inserted %d at the end\n", value);
}
// Function to insert at a specific position
void insertAtPosition(struct Node** head, int value, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        printf("Inserted %d at position %d\n", value, position);
        return;
    }
    struct Node* temp = *head;
    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position is out of bounds.\n");
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    printf("Inserted %d at position %d\n", value, position);
}

void deleteAtBeginning(struct Node** head) {   // Function to delete at the beginning
    if (*head == NULL) {
        printf("The list is empty.\n");
        return;
```

```
    }
    struct Node* temp = *head;
    *head = (*head)->next;  // Move head to the next node
    free(temp);  // Free the old head
    printf("Deleted the first node\n");
}
void deleteAtEnd(struct Node** head) {    // Function to delete at the end
    if (*head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);  // Only one node, free it
        *head = NULL;
        printf("Deleted the last node\n");
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL && temp->next->next != NULL) {
        temp = temp->next;  // Traverse to second-to-last node
    }

    free(temp->next);  // Free the last node
    temp->next = NULL;  // Set second-to-last node's next to NULL
    printf("Deleted the last node\n");
}

// Function to delete at a specific position
void deleteAtPosition(struct Node** head, int position) {
    if (*head == NULL) {
        printf("The list is empty.\n");
        return;
    }
```

```
    if (position == 1) {
        struct Node* temp = *head;
        *head = (*head)->next;
        free(temp);  // Free the first node
        printf("Deleted the node at position %d\n", position);
        return;
    }
    struct Node* temp = *head;
    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL || temp->next == NULL) {
        printf("Position out of bounds.\n");
        return;
    }
    struct Node* nodeToDelete = temp->next;
    temp->next = temp->next->next;
    free(nodeToDelete);  // Free the node at the desired position
    printf("Deleted the node at position %d\n", position);
}
void printList(struct Node *head) { // Function to print the list
    if (head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    printf("Current Linked List: ");
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

void traverseList(struct Node* head) {   // Function to traverse the list
    if (head == NULL) {
```

```
        printf("The list is empty.\n");
        return;
    }
    printf("Traverse and print all nodes in the list: ");
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}
int main() {
    struct Node* head = NULL; // Start with an empty list
    printList(head);
    // Insertion at the Beginning
    insertAtBeginning(&head, 104);  // Insert 104
    insertAtBeginning(&head, 103);  // Insert 103
    insertAtBeginning(&head, 102);  // Insert 102
    insertAtBeginning(&head, 101);  // Insert 101
    printList(head);
    printf("\n");
    // Insertion at the End
    insertAtEnd(&head, 106);  // Insert 106
    printList(head);
    printf("\n");
    // Insertion at Position 5
    insertAtPosition(&head, 105, 5);  // Insert 105 at position 5
    printList(head);
    printf("\n");
    deleteAtBeginning(&head);  // Delete the first node (101)
    printList(head);
    printf("\n");
    deleteAtEnd(&head);  // Delete the last node (106)
    printList(head);
    printf("\n");
```

```
    deleteAtPosition(&head, 3);  // Delete node at position 3 (104)
    printList(head);
    printf("\n");
    // Traversing the final linked list
    traverseList(head);  // Output: Traverse and print all nodes in the list: 102 103 105
    return 0;
}
```

Output:

```
user@DolindraBahadurRaut:~/DSA$ gcc singlyLinkedList.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
The list is empty.
Inserted 104 at the beginning
Inserted 103 at the beginning
Inserted 102 at the beginning
Inserted 101 at the beginning
Current Linked List: 101 -> 102 -> 103 -> 104 -> NULL
```

```
Inserted 106 at the end
Current Linked List: 101 -> 102 -> 103 -> 104 -> 106 -> NULL

Inserted 105 at position 5
Current Linked List: 101 -> 102 -> 103 -> 104 -> 105 -> 106 -> NULL

Deleted the first node
Current Linked List: 102 -> 103 -> 104 -> 105 -> 106 -> NULL
```

```
Deleted the first node
Current Linked List: 102 -> 103 -> 104 -> 105 -> 106 -> NULL

Deleted the last node
Current Linked List: 102 -> 103 -> 104 -> 105 -> NULL

Deleted the node at position 3
Current Linked List: 102 -> 103 -> 105 -> NULL

Traverse and print all nodes in the list: 102 103 105
user@DolindraBahadurRaut:~/DSA$
```

Conclusion:

Hence, we have successfully implemented a C program to perform the insert, delete, and traverse operations on a singly linked list.

Q.2) Write a program for linked list implementation of stack.

Answer:

Algorithm:

Push:

- Create a new node.
- Set the node's next to the current top.
- Update top to point to the new node.

Pop:

- If the stack is not empty, store the data of the top node.
- Update top to point to the next node.
- Free the memory of the previous top node.
- Return the stored data.

Example:

Push(100)

Push(200)

Pop( )

Program:

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

   int data;

   struct Node* next;

};

struct Node* top = NULL;    // Pointer to the top of the stack

void push(int value) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   newNode->data = value;

   newNode->next = top; // Point new node to the current top

   top = newNode; // Update the top to new node

   printf("%d pushed to stack\n", value);

}

int pop() {
   if (top == NULL) {
      printf("Stack is empty\n");
      return -1;
   }
   int poppedValue = top->data;

   struct Node* temp = top;

   top = top->next; // Move top pointer to next node
```

```
    free(temp); // Free the old top node
    return poppedValue;
}
void display() {
    if (top == NULL) {
        printf("Stack is empty\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    push(100);
    push(200);
    display(); // Display the stack
    printf("%d popped from stack\n", pop());
    display(); // Display the stack after pop
    return 0;
}
```


Output:

```
user@DolindraBahadurRaut:~/DSA$ gcc question2.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
100 pushed to stack
200 pushed to stack
Stack: 200 100
200 popped from stack
Stack: 100
```

Conclusion:

Hence, we successfully gave a C program for the linked list based implementation of stack.