Data Structures : Algorithms and Applications(Lab)

EXPERIMENT No: 6

AIM: Linked List implementation of Stack/queue in real life application.

THEORY:

A linked list is a dynamic data structure that consists of nodes where each node contains data and a reference (or link) to the next node in the sequence. This structure allows for efficient memory allocation and deallocation, making it ideal for implementing stack and queue operations.

In the context of stacks and queues:

- Stack: A stack follows the Last In First Out (LIFO) principle, where the most recently added element is the first to be removed. This is useful in various applications, such as parsing expressions, backtracking algorithms, and undo mechanisms in applications. By using a linked list, elements can be added or removed without the need for resizing an array, thus optimizing memory usage.
- Queue: A queue operates on the First In First Out (FIFO) principle, where the first element added is the first to be removed. Queues are essential in scenarios such as scheduling tasks, managing resources, and handling requests in a system. Like stacks, linked lists allow dynamic memory management, enabling efficient enqueue (adding to the end) and dequeue (removing from the front) operations without the constraints of a fixed-size array.

CODE:

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

// Node structure for linked list stack
struct Node {
   char data;
   struct Node* next;
};

// Function to push an element onto the stack
void push(struct Node** top, char data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = *top;
   *top = newNode;
}
```

```
// Function to pop an element from the stack
char pop(struct Node** top) {
   if (*top == NULL) {
      return '\0'; // Return a null character if the stack is empty
   }
   struct Node* temp = *top;
   char popped = temp->data;
   *top = (*top)->next;
   free(temp);
   return popped;
}

// Function to get the top element of the stack without removing it
char peek(struct Node* top) {
   return (top != NULL) ? top->data : '\0';
}

// Function to check if the stack is empty
int isEmpty(struct Node* top) {
   return top == NULL;
}

// Function to perform "READ" operation
void READ(struct Node* Undo) {
   struct Node* temp = Undo;
   // Print the Undo stack elements
   while (temp != NULL) {
      printf("%c", temp->data);
      temp = temp->next;
   }
   printf("\n");
}

// Function to perform the queries on the document
void QUERY(char Q[][10], int N) {
   struct Node* Undo = NULL; // Undo stack
   struct Node* Redo = NULL; // Redo stack

   // Traverse all the queries
   for (int i = 0; i < N; i++) {
      if (strcmp(Q[i], "UNDO") == 0) {
         char X = pop(&Undo); // Perform UNDO operation
```

```
            if (X != '\0') {
                push(&Redo, X); // Push to REDO stack
            }
        } else if (strcmp(Q[i], "REDO") == 0) {
            char X = pop(&Redo); // Perform REDO operation
            if (X != '\0') {
                push(&Undo, X); // Push to UNDO stack
            }
        } else if (strcmp(Q[i], "READ") == 0) {
            READ(Undo); // Perform READ operation
        } else {
            // WRITE operation
            char character = Q[i][6]; // Get the character after "WRITE "
            push(&Undo, character); // Push character to Undo stack
        }
    }
}

// Driver Code
int main() {
    char Q[][10] = { "WRITE A", "WRITE B", "WRITE C", "UNDO", "READ", "REDO", "READ" };
    int N = sizeof(Q) / sizeof(Q[0]); // Total count of queries
    QUERY(Q, N); // Pass both arguments to QUERY
    return 0;
}
```

OUTPUT:

```
BA
CBA


=== Code Execution Successful ===
```

CONCLUSION:

In conclusion, the linked list implementation of stacks and queues provides a flexible and dynamic approach to managing data structures, allowing for efficient memory usage and easy manipulation of elements. This method proves invaluable in real-life applications, such as undo/redo functionalities in software, where maintaining order and accessibility of data is crucial.