Experiment No. 8: Stack using Linked list

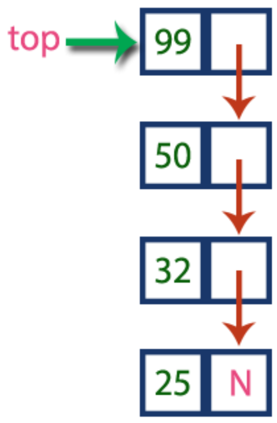
Aim: Implement Stack ADT using Linked list

Objective:

Stack can be implemented using linked list for dynamic allocation. Linked list implementation gives flexibility and better performance to the stack.

Theory:

A stack implemented using an array has a limitation in that it can only handle a fixed number of data values, and this size must be defined at the outset. This limitation makes it unsuitable for cases where the data size is unknown. On the other hand, a stack implemented using a linked list is more flexible and can accommodate an unlimited number of data values, making it suitable for variable-sized data. In a linked list-based stack, each new element becomes the 'top' element, and removal is achieved by updating 'top' to point to the previous node, effectively popping the element. The first element's "next" field should always be NULL to indicate the end of the list.



Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.

Step 2 - Define a 'Node' structure with two members data and next.

Step 3 - Define a Node pointer 'top' and set it to NULL.

Step 4 - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

push(value) - Inserting an element into the Stack

Step 1 - Create a newNode with given value.

Step 2 - Check whether stack is Empty (top == NULL)

Step 3 - If it is Empty, then set newNode → next = NULL.

Step 4 - If it is Not Empty, then set newNode → next = top.

Step 5 - Finally, set top = newNode.

pop() - Deleting an Element from a Stack

Step 1 - Check whether the stack is Empty (top == NULL).

Step 2 - If it is Empty, then display "Stack is Empty!!!

Step 3 - If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.

Step 4 - Then set 'top = top → next'.

Step 5 - Finally, delete 'temp'. (free(temp)).

display() - Displaying stack of elements

Step 1 - Check whether stack is Empty (top == NULL).

Step 2 - If it is Empty, then display 'Stack is Empty!!!' and terminate the function.

Step 3 - If it is Not Empty, then define a Node pointer 'temp' and initialize with top.

Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack. (temp → next != NULL).

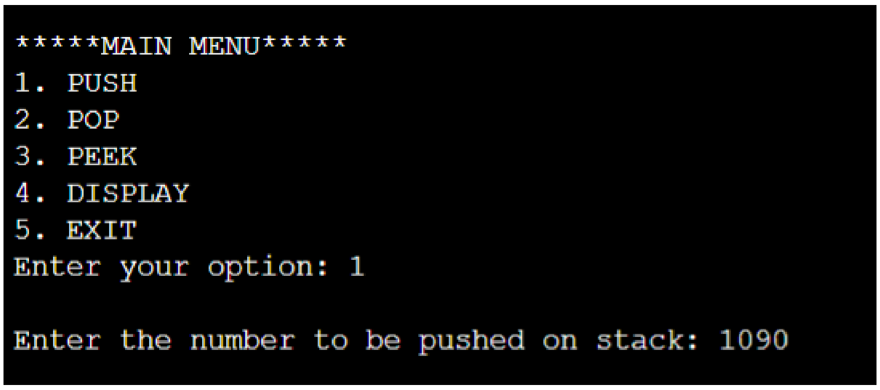
Step 5 - Finally! Display 'temp → data ---> NULL'.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>
struct stack
{
int data;
struct stack *next;
};
struct stack *top = NULL;
struct stack *push(struct stack *, int);
struct stack *display(struct stack *);
struct stack *pop(struct stack *);
int peek(struct stack *);
int main(int argc, char *argv[]) {
int val, option;
do
{
 printf("\n *****MAIN MENU*****");
 printf("\n 1. PUSH");
 printf("\n 2. POP");
 printf("\n 3. PEEK");
 printf("\n 4. DISPLAY");
 printf("\n 5. EXIT");
 printf("\n Enter your option: ");
 scanf("%d", &option);
 switch(option)
 {
 case 1:
 printf("\n Enter the number to be pushed on stack: ");
 scanf("%d", &val);
 top = push(top, val);
 break;
 case 2:
 top = pop(top);
 break;
 case 3:
 val = peek(top);
 if (val != -1)
 printf("\n The value at the top of stack is: %d", val);
 else
 printf("\n STACK IS EMPTY");
 break;
 case 4:
 top = display(top);
 break;
 }
}while(option != 5);
return 0;
}
struct stack *push(struct stack *top, int val)
{
struct stack *ptr;
ptr = (struct stack*)malloc(sizeof(struct stack));
ptr -> data = val;
if(top == NULL)
{
 ptr -> next = NULL;
 top = ptr;
}
else
{
 ptr -> next = top;
 top = ptr;
}
return top;
}
struct stack *display(struct stack *top)
{
struct stack *ptr;
ptr = top;
if(top == NULL)
printf("\n STACK IS EMPTY");
else
{
```

```
 while(ptr != NULL)
 {
 printf("\n %d", ptr -> data);
 ptr = ptr -> next;
 }
}
return top;
}
struct stack *pop(struct stack *top)
{
struct stack *ptr;
ptr = top;
if(top == NULL)
printf("\n STACK UNDERFLOW");
else
{
 top = top -> next;
 printf("\n The value being deleted is: %d", ptr -> data);
 free(ptr);
}
return top;
}
int peek(struct stack *top)
{
if(top==NULL)
return -1;
else
return top ->data;
}
```

Output:



```
*****MAIN MENU*****
1. PUSH
2. POP
3. PEEK
4. DISPLAY
5. EXIT
Enter your option: 1

Enter the number to be pushed on stack: 1090
```

Conclusion:

Write in detail about an application where stack is implemented as linked list?

Application: Infix to Postfix Conversion and Evaluation

In computer science, infix notation is the standard way of writing arithmetic expressions, where operators are placed between operands (e.g., 2 + 3 * 5). However, for efficient evaluation, postfix (or Reverse Polish Notation) is often preferred. Converting infix expressions to postfix allows for straightforward evaluation using a stack. How Linked List as a Stack is Used:

Infix to Postfix Conversion:

As the infix expression is scanned from left to right, a stack is used to keep track of operators.

When an operand is encountered, it is output to the postfix expression.

When an operator is encountered, it is pushed onto the stack after handling any operators already on the stack.

The stack ensures that operators with higher precedence are processed first.

Postfix Expression Evaluation:

After the conversion, the postfix expression is evaluated using a stack.

Operands are pushed onto the stack.

When an operator is encountered, the necessary number of operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack.

The final result is obtained from the stack.