**PROGRAM-1**

***Q. Design and implement a stack using ARRAY IMPLEMENTATION and demonstrate its working with necessary inputs. Display the appropriate messages in case of exceptions*.**

* **Theory:**

Using linked lists to implement a stack is another typical method. The elements in a linked list-based stack are kept in nodes that each include an element and a pointer to the next node in the list.

We make a new node and update its pointer to point to the current top of the stack in order to add an element to the top of the stack. The top pointer is then modified to point at the new node. Simply delete the node at the top of the linked list and change the top pointer to point to the subsequent node in the list to remove an element from the top of the stack.

Due to linked lists' ability to dynamically grow and contract as necessary, one benefit of employing one is that it can manage any number of elements. Array-based stacks, on the other hand, have a set size and may result in overflow or underflow issues if the stack fills up or empties out.

Since we only need to change the pointers of a small number of nodes in the linked list, employing a linked list-based stack also enables constant-time insertions and removals at the top of the stack. Compared to an array-based stack, which may need to move all of the remaining array elements after an insertion or deletion, this may be more effective.

In general, linked list-based stacks can be a practical and effective technique to design stacks, particularly when the number of components in the stack is ambiguous or variable. They can dynamically grow and shrink as necessary and allow for constant-time insertions and deletions at the top of the stack.

**ALGORITHM**

1. Start with an empty stack.
2. Create a new node to represent the top of the stack.
3. Set the data field of the new node to the first element of the stack.
4. Set the next field of the new node to NULL.
5. Set the top of the stack to point to the new node.

**PUSH OPERATION**

step 1: A new node can be made to symbolise the element.

step 2: Change the new node's data field to point to the element that will be added.

step 3: Set the new node's next field to point to the stack's current top.

step 4: Change the stack's top node to point at the new node.

**POP OPERATION**

Step 1: Determine whether the stack is empty (top should be NULL); if so, move on.

step 2: the second step is to direct a temporary node to the top of the stack.

step 3: Set the top of the stack to point to the list's third node.

step 4: Release the RAM that was assigned to the temporary node.

step 5: Return the data from the temporary node.

**DISPLAY OPERATION**

step 1: Beginning from the top, go down the stack.

step 2: Print each node's data as it is accessed.

step 3: Keep going until the stack's end (NULL) is reached.

**CODE**

#include<stdio.h>

#include<stdlib.h> //header files are included

//----------------------------------------------------

void push(); // user defined function for push operation

void pop(); // user defined function for pop operation

void display(); // user defined function for display operation

//----------------------------------------------------

struct node // structure created calling it as node

{

int val; // integer datatype called val

struct node \*next; // pointer next is taken

};

struct node \*head; //head pointer taken

//----------------------------------------------------

//main function starts from here

void main()

{

int choice=0;

printf("-------------STACKS USING LINKED LIST--------------- \n");

while(choice!=4) // while operator used to create loop for menu driven program

{

printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - \n");

printf("Choose the operation to be performed:\n");

printf("1.Push\n2.pop\n3.display\n4.exit\n\n"); //menu printed with options

scanf("%d",&choice);

switch(choice) //switch case

{

case 1: push();

break;

case 2: pop();

break; // all four options in different cases(to call)

case 3: display();

break;

case 4: exit(0); //to exit the loop

break;

default: printf("enter valid choice \n");

}

}

}

//------------------------------------------------

//push function defined

void push()

{

int val;

struct node \*ptr= (struct node \*)malloc(sizeof(struct node)); //dynamic memory allocation

if(ptr==NULL)

{

printf("cannot be inserted \n\n"); //in case ptr pointer is null

}

else

{

printf("enter the value: ");

scanf("%d",&val);

if(head==NULL) //if head is null

{

ptr->val=val; //ptr pointing to val

ptr->next=NULL; //ptr pointing to next

head=ptr; //head equals ptr

}

else

{

ptr->val=val; //ptr pointing to val

ptr->next=head; //ptr pointing to next equals head

head=ptr;

}

printf("item pushed\n\n"); //item pushed

}

}

//------------------------------------------------

//pop function defined

void pop()

{

int item;

struct node \*ptr; //ptr pointer

if(head==NULL)

{

printf("underflow\n"); //if head is null

}

else

{

item=head->val; //item is equal to head pointing to val

ptr=head; //ptr pointer equals to head pointer

head=head->next;

free(ptr); //ptr pointer is freed

printf("item popped\n\n"); //item popped

}

}

//------------------------------------------------

//display function defined

void display()

{

int i;

struct node \*ptr; //ptr pointer given

ptr=head; //ptr equals head

if(ptr==NULL)

{

printf("stack is empty\n"); // in case ptr equals null

}

else

{

printf("the elements in the stack are \n");

while(ptr!=NULL) //while pointer ptr is not equal to null

{

printf("%d\n",ptr->val); //print the value pointed by ptr

ptr=ptr->next; //ptr points to next

}

}

}

//end of program

//--------------------------------------------------

**“SCREENSHOTS OF OUTPUT”**



