STACK

A stack is a data structure that follows the **Last In, First Out** (LIFO) principle, meaning that the last element added to the stack will be the first one to be removed.

The basic operations that can be performed on a stack:

1. Push: Add an element to the top of the stack**.**
2. Pop: Remove the top element from the stack.
3. Peek (or Top): Look at the top element without removing and inserting it.
4.isEmpty: Checks if the stack has no elements.

Stacks have many applications in programming, such as:

- Evaluating postfix expressions
- Implementing recursive functions
- Parsing syntax in compilers
- Managing memory allocation
- Implementing undo/redo functionality

Implementation of stack:

- **Array based stacks:** Uses a fixed-size or dynamic array to store elements. Operations such as push, pop, and peek are performed using array indices.

- Linked list based stacks: Uses a linked list where each node contains an element and a reference to the next node. This approach allows for dynamic sizing.

- Dynamic stacks: Unlike a fixed-size stack, which has a predetermined maximum size, a dynamic stack can expand and contract based on the number of elements it contains.

- Structure and pointers stacks: In this implementation, each node in the linked list represents an element in the stack. Each node contains the data and a pointer to the next node in the stack. The stack itself is managed with a pointer to the top node.

1.EXAMPLE FOR PUSH OPERATION :

```
#include <stdio.h>
#define MAX 10
int stack[MAX];
int top = -1;
void push(int value)
 {
    if (top == MAX - 1)
 {
        printf("Stack Overflow! Cannot push %d onto the stack.\n", value);
    }
else
 {
        top++;
        stack[top] = value;
        printf("%d pushed onto the stack.\n", value);
    }
}
void display()
 {
    if (top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack elements are:\n");
        for (int i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
        }
    }
}
int main() {
    push(10);
    push(20);
    push(30);
    push(40);
    push(50);
    display();
    return 0;
}
```

OUTPUT:

10 pushed onto the stack.
20 pushed onto the stack.
30 pushed onto the stack.
40 pushed onto the stack.
50 pushed onto the stack.
Stack elements are:
50
40
30
20
10

2.EXAMPLE FOR POP OPERATION:

```
#include <stdio.h>
#define MAX 10
int stack[MAX];
int top = -1;
void push(int value) {
    if (top == MAX - 1) {
        printf("Stack Overflow! Cannot push %d onto the stack.\n", value);
    } else {
        top++;
        stack[top] = value;
        printf("%d pushed onto the stack.\n", value);
    }
}
int pop() {
    if (top == -1) {
        printf("Stack Underflow! Cannot pop from the stack.\n");
        return -1;  // Returning -1 to indicate error
    } else {
        int poppedValue = stack[top];
        top--;
        printf("%d popped from the stack.\n", poppedValue);
        return poppedValue;
    }
}
void display() {
```

```
    if (top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack elements are:\n");
        for (int i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
        }
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    push(40);
    push(50);
    printf("\nCurrent Stack:\n");
    display();
    printf("\nPopping elements:\n");
    pop();
    pop();
    printf("\nStack after popping elements:\n");
    display();
    return 0;
    }
```

OUTPUT:

```
10 pushed onto the stack.
20 pushed onto the stack.
30 pushed onto the stack.
40 pushed onto the stack.
50 pushed onto the stack.
Current Stack:
Stack elements are:
50
40
30
20
10
Popping elements:
```

```
50 popped from the stack.
40 popped from the stack.
Stack after popping elements:
Stack elements are:
30
20
10
```

3.EXAMPLE FOR PEEK OPERATION:

```
#include <stdio.h>
#define MAX 10
char stack[MAX];
int top = -1;
void push(char value) {
    if (top == MAX - 1) {
        printf("Stack Overflow! Cannot push '%c' onto the stack.\n", value);
    } else {
        top++;
        stack[top] = value;
        printf("'%c' pushed onto the stack.\n", value);
    }
}
char peek() {
    if (top == -1) {
        printf("Stack is empty! Cannot peek.\n");
        return '\0';
    } else {
        return stack[top];
    }
}
void display() {
    if (top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack elements are:\n");
        for (int i = top; i >= 0; i--) {
            printf("%c\n", stack[i]);
        }
    }
```

```c
}

int main() {
    push('A');
    push('B');
    push('C');
    push('D');
    push('E');
    printf("\nCurrent Stack:\n");
    display();
    printf("\nPeek at the top element: '%c'\n", peek());
    return 0;
}
```

OUTPUT:

```
'A' pushed onto the stack.
'B' pushed onto the stack.
'C' pushed onto the stack.
'D' pushed onto the stack.
'E' pushed onto the stack.
Current Stack:
Stack elements are:
E
D
C
B
A
Peek at the top element: 'E'.
```

1.STACK WITH ARRAY

Advantages:

- Simple to implement.
- Allows random access to elements (though not commonly used for stacks).

Disadvantages:

- Fixed size in basic implementation (unless using dynamic resizing).

Example :

```
#include<stdio.h>
#include<limits.h>
#define MAX_SIZE 20
int arr[MAX_SIZE];
int top=-1;
void push(int val)
{
  if(top==MAX_SIZE-1)
  {
    printf("stack is full\n");
    return 0;
  }
  arr[+top]=val;
}
int pop()
{
  if(top==-1)
  {
    printf("stack is empty\n");
    return INT_MIN;
  }
  return arr[top--];
}
int peek()
{
  if(top==-1)
  {
    printf("stack is empty\n");
    return INT_MIN;
  }
}
int main()
{
  push(10);
  push(20);
  push(30);
  push(40);
  push(50);
```

```
  printf("after pushing 5 elements:\n");
  printf("top element: %d\n",peek());
  printf("popped element: %d\n",pop());
  printf("popped element: %d\n",pop());
  printf("after popping 2 elements:\n");
  printf("top element is %d\n",peek());
  return 0;
}
```

OUTPUT

after pushing 5 elements:
top element: 50
popped element: 50
popped element: 40
after popping 2 elements:
top element is 30

2.STACK WITH STRUCTURE AND POINTER.

Advantages

1. **Dynamic Size:**if refers to defining data structure,whose size can change at run time.
2. **Efficient Memory Use:**Memory is allocated only when needed.there's no need to allocate a large block of memory upfront.
3. **Simple Insertions and Deletions:**Both “push” and “`pop`” operations are O(1) operations because they involve only updating pointers and do not require shifting elements.

Disadvantages

1. **Memory Overhead:** Each node requires extra memory for the pointer in addition to the data.
2. **Pointer Management:** Requires careful handling of pointers to avoid memory leaks and dangling pointers. Every “`malloc`”should be paired with a `free`, and pointer updates need to be managed carefully.
3. **Cache Performance:** Due to non-contiguous memory allocation, linked lists may suffer from poor cache locality compared to arrays. This can lead to slower access times compared to array-based implementations

Example :

```
# include<stdio.h>
#include<limits.h>
#include<stdlib.h>
#define MAX_SIZE 20
struct Stack
{
int *arr;
int top;
int size;
};
void initStack(struct Stack *stack, int size)
{
stack->arr=(int *)malloc(size * sizeof(int)) ;
stack->top = -1;
stack->size = size;
}
void push(struct Stack *stack, int val){
if(stack->top == stack->size - 1)
{
printf("stack is full\n") ;
return;
}
stack->arr[++stack->top] = val;
}
int pop(struct Stack *stack)
{
if(stack->top == -1)
{
printf("stack is empty\n") ;
return INT_MIN;
```

```
}
return stack->arr[stack->top--];
}
int peek(struct Stack *stack)
{
if(stack->top == -1)
{
printf("stack is empty\n") ;
return INT_MIN;
}
return stack->arr[stack->top--];
}
void freeStack(struct Stack*stack)
{
free(stack->arr);
}
int main()
{
struct Stack stack1, stack2;
initStack(&stack1,MAX_SIZE) ;
initStack(&stack2,MAX_SIZE) ;
push(&stack1, 10) ;
push(&stack1, 20) ;
push(&stack1, 30) ;
push(&stack1, 40) ;
push(&stack1, 50) ;
push(&stack2, 100) ;
push(&stack2, 200) ;
push(&stack2, 300) ;
printf("popped from stack 1: %d\n", pop(&stack1)) ;
printf("popped from stack 2: %d\n", pop(&stack2)) ;
printf("top of stack 2: %d\n", peek(&stack2)) ;
freeStack(&stack1) ;
freeStack(&stack2) ;
return 0;
}
```

OUTPUT:

popped from stack 1: 50
popped from stack 2: 300
top of stack 2: 200

2.STACK WITH LINKED LIST:

Advantages:

- Dynamic size.
- No need to manage capacity explicitly.

Disadvantages:

- Overhead of storing pointers.

Example :

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

struct node{
int data;
struct node *next;
};
struct node *createnode(int data){
  struct node *newnode=(struct node*)malloc(sizeof(struct node));
   if(!newnode){
    printf("memory allocation failed");
    return NULL;
  }
  newnode->data=data;
  newnode->next=NULL;
  return newnode;
}
void push(struct node **top,int data)
{
   struct node *newnode=createnode(data);//create a node funtion are used to
create a node
  if(!newnode){
```

```
    return;
  }
  newnode->next=*top;
  *top=newnode;
  printf("%d pushed to stack\n",data);
}
int pop(struct node **top)
{
  if(*top==NULL){
    printf("stack is empty\n");
    return INT_MIN;
  }
  struct node *temp=*top;
  int poppedvalue=temp->data;
  *top=temp->next;
  free(temp);
  return poppedvalue;

}
int beginning(struct node *top){
  if(top==NULL){
    printf("stack is empty\n");
    return INT_MIN;
  }
  struct node *current = top;
  while(current->next!=NULL){
    current=current->next;
  }
  return current->data;
}
int main()
{
  struct node *top=NULL;
  push(&top,10);
  push(&top,20);
  push(&top,30);
  push(&top,40);
  push(&top,50);
  printf("popped element is %d\n",pop(&top));
```

```
printf("bottom element is %d\n", beginning(top));
printf("popped element is %d\n",pop(&top));
printf("popped element is %d\n",pop(&top));
printf("top element is %d\n", beginning(top));
return 0;
}
```

OUTPUT:

50 pushed to stack
40 pushed to stack
30 pushed to stack
20 pushed to stack
10 pushed to stack
popped from stack: 10
popped from stack: 20
top of stack: 30