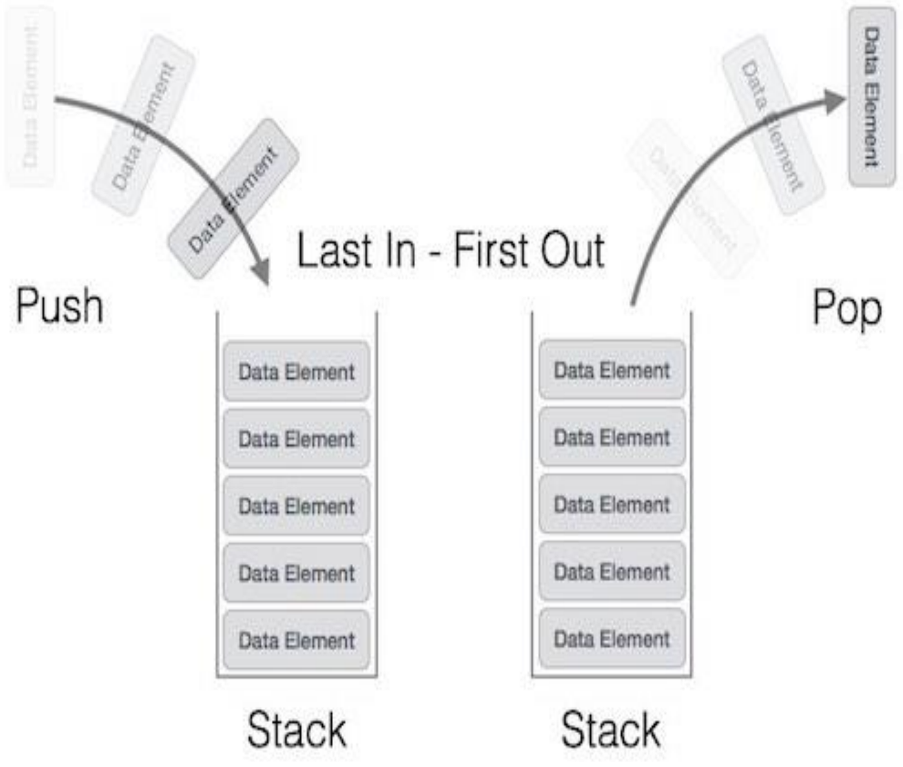
DATA STRUCTURES

UNIT III:

Stacks: Introduction to stacks: properties and operations, implementing stacks using arrays and linked lists, Applications of stacks in expression evaluation, backtracking, reversing list etc.

Introduction to Stack and properities

- A stack is a linear data structure consisting of a set of homogeneous elements and is based on the principle of last in first out (LIFO).
- In stack, adding and removing an element are performed at a single position called top.
- The element which is entered at last will be coming first (LIFO)
- The push operation adds an element to the stack while the pop operation removes an element from the top position.
- The stack concept is used in programming and memory organization in computers.



Stack ADT

Abstract Data Type Stack
{
Instances: A Stack contains elements of same type arranged in sequential order. All operations take place at a single end that is top of the stack and it follows principal LIFO.
Operations
push() – Insert an element at one end of the stack called top.

pop() – Remove and return the element at the top of the stack, if it is not empty.

peek () – Return the element at the top of the stack without removing it, if the stack is not empty.

size () – Return the number of elements in the stack.

stackempty () – Return true if the stack is empty, otherwise return false.

stackfull () – Return true if the stack is full, otherwise return false.
}

Stack Operations

1. Algorithm for PUSH ()

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

```
Step 1: If TOP >= SIZE – 1 then
            Write "Stack is Overflow"
Step 2: TOP = TOP + 1
Step 3: STACK [TOP] = X
```

2. Algorithm for POP ()

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

```
Step 1: If TOP = -1 then
            Write "Stack is Underflow"
Step 2: Return STACK [TOP]
Step 3: TOP = TOP – 1
```

3. Algorithm for PEEK ()

The Peek operation is employed when it is necessary to return the value of the topmost stack element without erasing it. This operation first determines whether the Stack is empty, i.e., TOP = NULL; if it is, then the value will be returned; otherwise, an appropriate notice will be displayed.

```
Algorithm:
Step-1: If TOP = NULL
PRINT "Stack is Empty"
Goto Step 3
Step-2: Return Stack[TOP]
Step-3: END
```

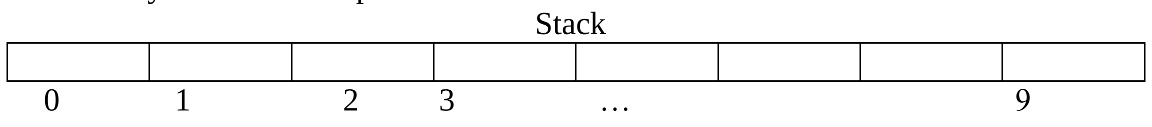
Representation of Stack in the memory

Stack is special case of an ordered list, i.e. it is ordered list with some restrictions on the way in which we perform various operations on a list. It follows the principal LIFO. Stack can represent in two methods.
(1) Using Array
(2) Using Linked List

Stack using an Array: The first method to represent stack is using an array. To do this we need to define an array if some maximum size. Moreover, we need an integer variable top which will keep track of the top of the stack as more and more elements are inserted into and deleted from the stack. The declarations in C are as follows:
Declaration 1:

```
#define SIZE 10
int stack[SIZE], top=-1
```

In the above declaration stack is nothing but an array of integers. and most recent index of that array will act as a top.



Initially
Top= -1

The stack is of size 10. As we insert the numbers, the top will get incremented. The elements will be placed from 0th position in the stack. At the most we can store 10 elements in the stack, so at the most last element can be at (SIZE-1) position. i.e. index 9.
Declaration 2:

```
#define SIZE 10
Struct stack
{
    int s[SIZE];
    int top=-1;
}st;
```

In the above declaration stack is declared as structure. Here top and stack are associated with each other by putting them together in a structure.

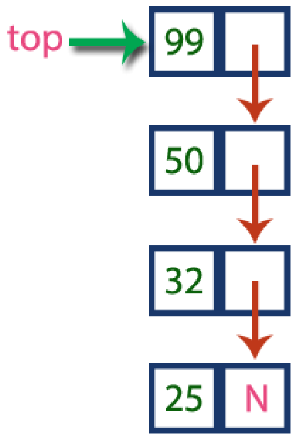
Linked Representation Of Stack

❑ The drawback of using array method is fixed size, but in this method it's removed.

❑The storage requirement of linked representation of the stack with n elements is O(n) and the typical time requirement for the operation is O(1).

❑In a linked stack, every node has two parts : one that stores data and another that stores the address of the next node.

❑The START pointer of the linked list is used as TOP.



Implementing stack using array

```
#define MAXSIZE 10
#include<stdio.h>
int a[MAXSIZE],top=-1;
void main()
{
int ch, e;
void push();
int pop();
void peek();
clrscr();
do
{
printf("\n1.PUSH\n");
printf("\n2.POP\n");
printf("\n3.PEEK\n");
printf("\n4.EXIT\n");
printf("\nENTER YOUR CHOICE:\n");
scanf("%d", &ch);
switch(ch)
{
case 1:
if(isfull())
printf("STACK FULL\n");
else
{
printf("\nEnter Element to be pushed:\n");
scanf("%d”, &e);
push(e);
}
break;
case 2:
if(isempty ())
printf("\nSTACK EMPTY\n");
else
{
printf("\nThe Poped Element=%d",pop());
}
break;
case 3:
if(isempty())
printf("\nSTACK IS EMPTY\n");
else
peek();
break;
case 4:
exit(0);
}
}while(ch!=4);
}
void push(int x)
{
top=top+1;
a[top]=x;
}
int pop()
{
return(a[top--]);
}
void peek()
{
int j;If(lsempty())
Printf("stack is empty");
Else

{
for(j=top;j>=0;j--)
printf("%4d",a[j]);
}
}
int isfull()
{
if(top==MAXSIZE-1)
return 1;
else
return 0;
}
int isempty()
{
if(top==-1)
return 1;
else
return 0;
}
```

Implementing stack using Linked List

```
#include<stdio.h>
#include<conio.h>
struct Node
{
int data;
struct Node *next;
}*top = NULL;
void push(int);
void pop();
void display();
void main()
{
int choice, value;
clrscr();
printf("\n:: Stack using Linked List ::\n");
while(1){
printf("\n****** MENU ******\n");
printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
printf("Enter your choice: ");
scanf("%d",&choice);
switch(choice){
case 1: printf ("Enter the value to be insert: ");
scanf("%d", &value);
push(value);
break;
case 2: pop(); break;
case 3: display(); break;
case 4: exit(0);
default: printf("\nWrong selection!!! Please try again!!!\n");
}}}
void push(int value)
{
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
if(top == NULL)
newNode->next = NULL;
else
newNode->next = top;
top = newNode;
printf("\nInsertion is Success!!!\n");
}
void pop()
{
if(top == NULL)
printf("\nStack is Empty!!!\n");
else{
struct Node *temp = top;
printf("\nDeleted element: %d", temp->data);
top = temp->next;
free(temp);
}
}
void display()
```

```
{
if(top == NULL)
printf("\nStack is Empty!!!\n");
else{
struct Node *temp = top;
while(temp->next != NULL){
printf("%d--->",temp->data);
temp = temp -> next;
}
printf("%d--->NULL",temp->data);
}
}
```

Applications of Stack

- To find the reverse of a given string
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
- Forward and backward feature in web browsers
- To convert infix expression to postfix expression
- To Evaluate postfix expression
- Used in recursion
- Used in Function calls
- Used in Depth First Search Algorithm
- Decimal to Binary Conversion

1. **Reverse a String:** To reverse a string Stack can be used. The simple mechanism is to push all the characters of a string onto the stack and then pop all the characters from the stack and print them. For example

If the input string is

P	**R**	**O**	**G**	**R**	**A**	**M**	**\0**

Then push all the characters onto the stack till '\0' encountered.

M
A
R
G
O

R
P

Stack

Now if we pop each character from the stack and print it we get,

M	**A**	**R**	**G**	**O**	**R**	**P**

2. **Factorial Calculation using Stack:** Factorial can be computed using stack. The simple algorithm is:

Step 1:	Enter the number for which the factorial is to be computed – say n
Step 2:	push the numbers from 1 to n onto the stack
Step 3:	initialize prod=1
Step 4:	Perform the multiplication with prod by popping the element each time.
Step 5:	Store the multiplication in prod variable
Step 6:	Repeat Steps 4 and 5 for n elements.
Step 7:	Finally display the factorial value.

3. **Algorithm to convert Infix to Postfix with example**

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

```
1. Push "("onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is
   empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered, then:
     1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which
        has the same precedence as or higher precedence than operator.
     2. Add operator to Stack.
        [End of If]
6. If a right parenthesis is encountered, then:
     1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a
        left parenthesis is encountered.
     2. Remove the left Parenthesis.
        [End of If]
        [End of If]
7. END
```

Convert the infix expression '(a+b) * c/d + (e+f)' to postfix expression.

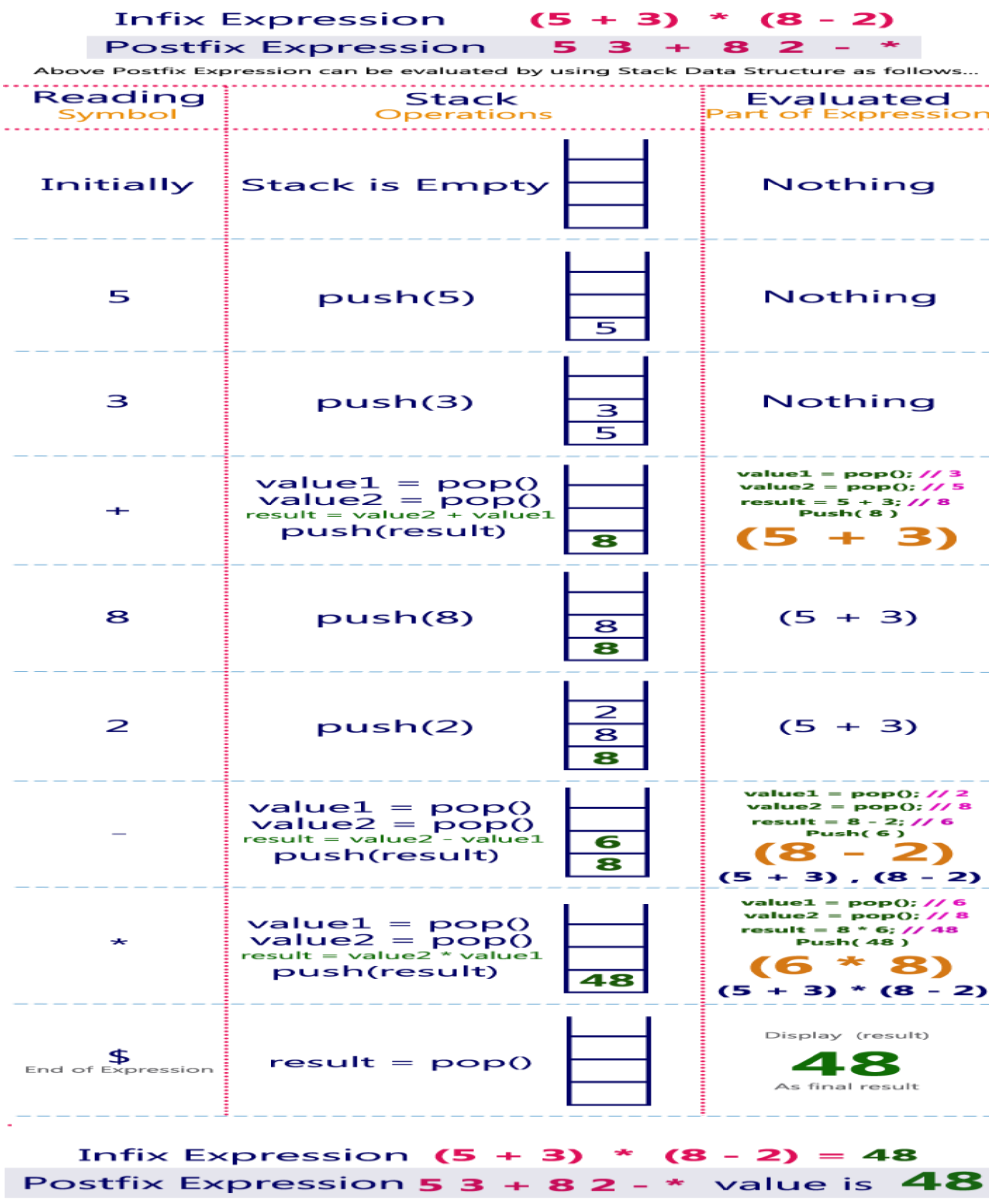
Input Character	**Stack**	**Postfix Expression**
	(	
(	((	
a	((	a
+	((+	a
b	((+	ab
)	(	ab+
*	(*	ab+
c	(*	ab+c
/	(/	ab+c*
d	(/	ab+c*d
+	(+	ab+c*d/
(	(+(	ab+c*d/
e	(+(	ab+c*d/e
+	(+(+	ab+c*d/e
f	(+(+	ab+c*d/ef
)	empty	ab+c*d/ef++

4. Algorithm to Evaluate postfix expression with example

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

- Read all the symbols one by one from left to right in the given Postfix Expression
- If the reading symbol is operand, then push it on to the Stack.
- If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.

➢ Finally! Perform a pop operation and display the popped value as final result.



Infix Expression (5 + 3) * (8 - 2)
Postfix Expression 5 3 + 8 2 - *
Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)	5	Nothing
3	push(3)	3 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)	8 8	(5 + 3)
2	push(2)	2 8 8	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		6
		8
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
		48	value1 = pop(); // 6
			value2 = pop(); // 8
			result = 8 * 6; // 48
			Push(48)
			(6 * 8)
			(5 + 3) * (8 - 2)

$ End of Expression	result = pop()		Display (result) 48 As final result

Infix Expression (5 + 3) * (8 - 2) = 48
Postfix Expression 5 3 + 8 2 - * value is 48

5. Backtracking Algorithms

Backtracking algorithms, such as depth-first search (DFS) in graph theory, depend on the stack data structure to systematically explore paths and make choices at each step. As the algorithm progresses, the current path and choices are pushed onto the stack. If a dead-end is encountered, the algorithm backtracks by popping elements from the stack, allowing it to explore alternative paths and find solutions to problems like pathfinding and puzzles.

6. Reversing a List

The idea of this approach is simple: traverse all the nodes one by one from the starting node of the given linked list and push all the values of the nodes into the auxiliary stack.

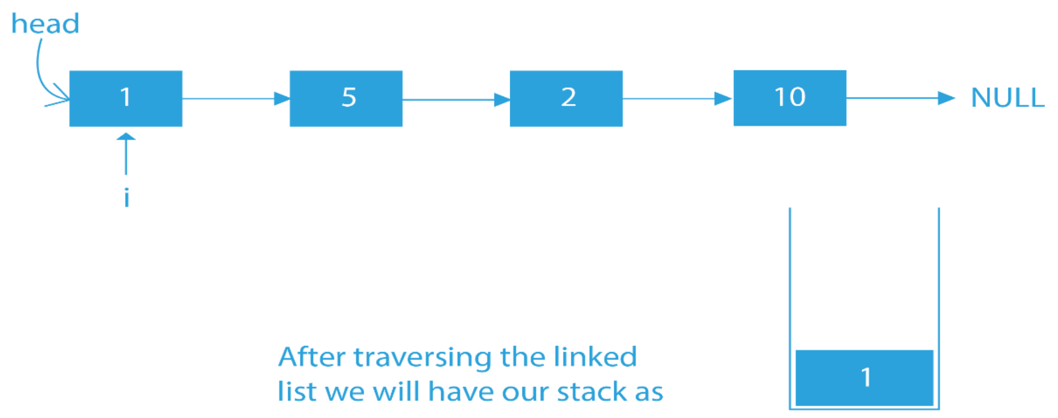
We know that the stack Data Structure follows the LIFO (last in first out) order. So, if we extract the content of the stack, we get the last element first, then the second last element and so on, i.e. we get the data in reverse order.

Finally, we change the linked list data from starting with the reversed data from the stack one by one for all nodes.

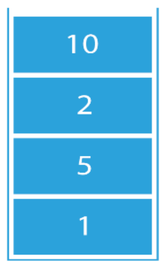
Algorithm:

1. Check if the linked list is empty or has a single node. If yes, then no need to reverse it and simply return from the function. Otherwise, follow the below steps.
2. Declare an empty stack.
3. Iterate through the linked list and push the values of the nodes into the stack.
4. Now, after the iteration is complete, the linked list is stored in reverse order in the stack.
5. Now, start popping the elements from the stack and iterating through the linked list together, and change the values in the nodes by the values we get from the stack.
6. Once the stack is empty, we will have the required reversed linked list.

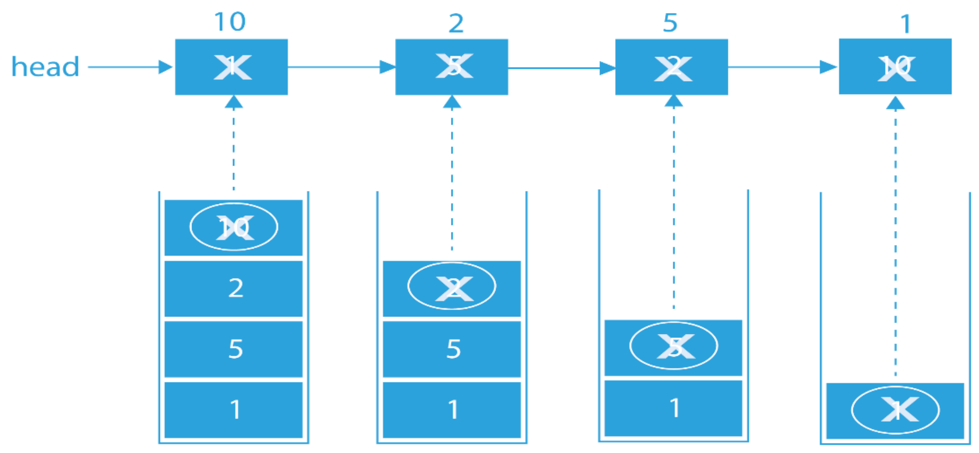
Let's understand the above approach with an example:

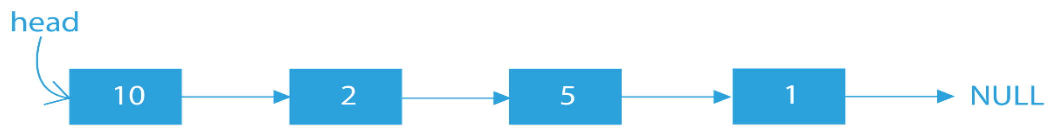


After traversing the linked list we will have our stack as



Now iterate the linked list again, replace the value in nodes with stack's top element. POP that element from stack after using it


10
2
5
1
head
2
5
1
5
1
1


head
10
2
5
1
NULL