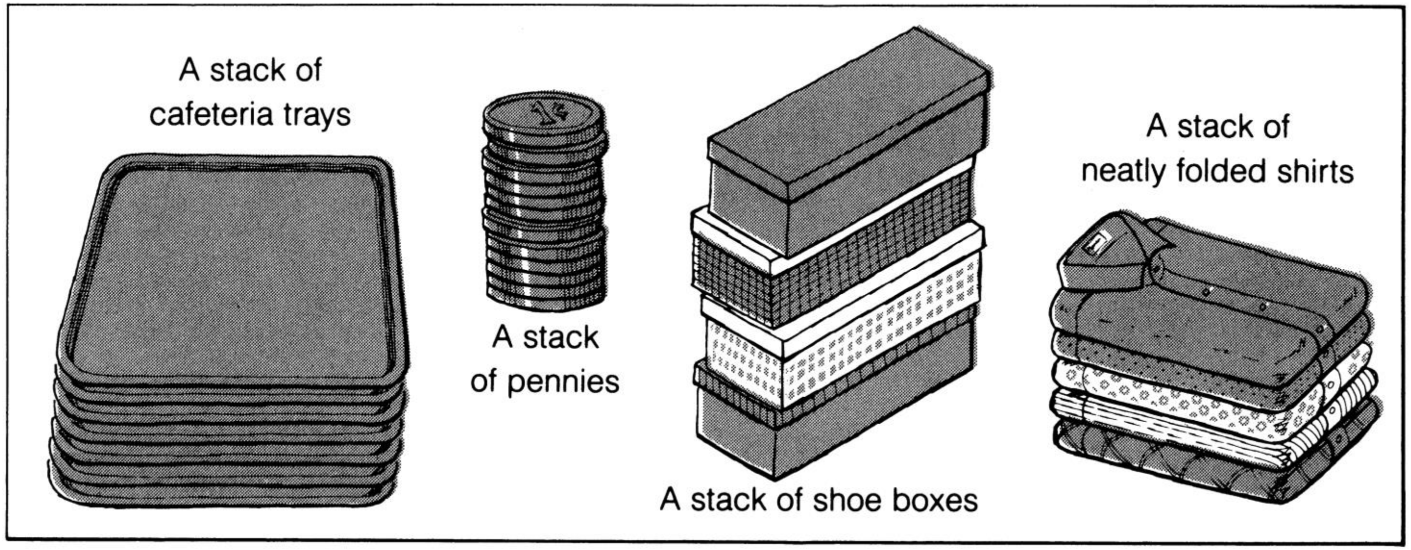
Array, **Stacks**, and Queues

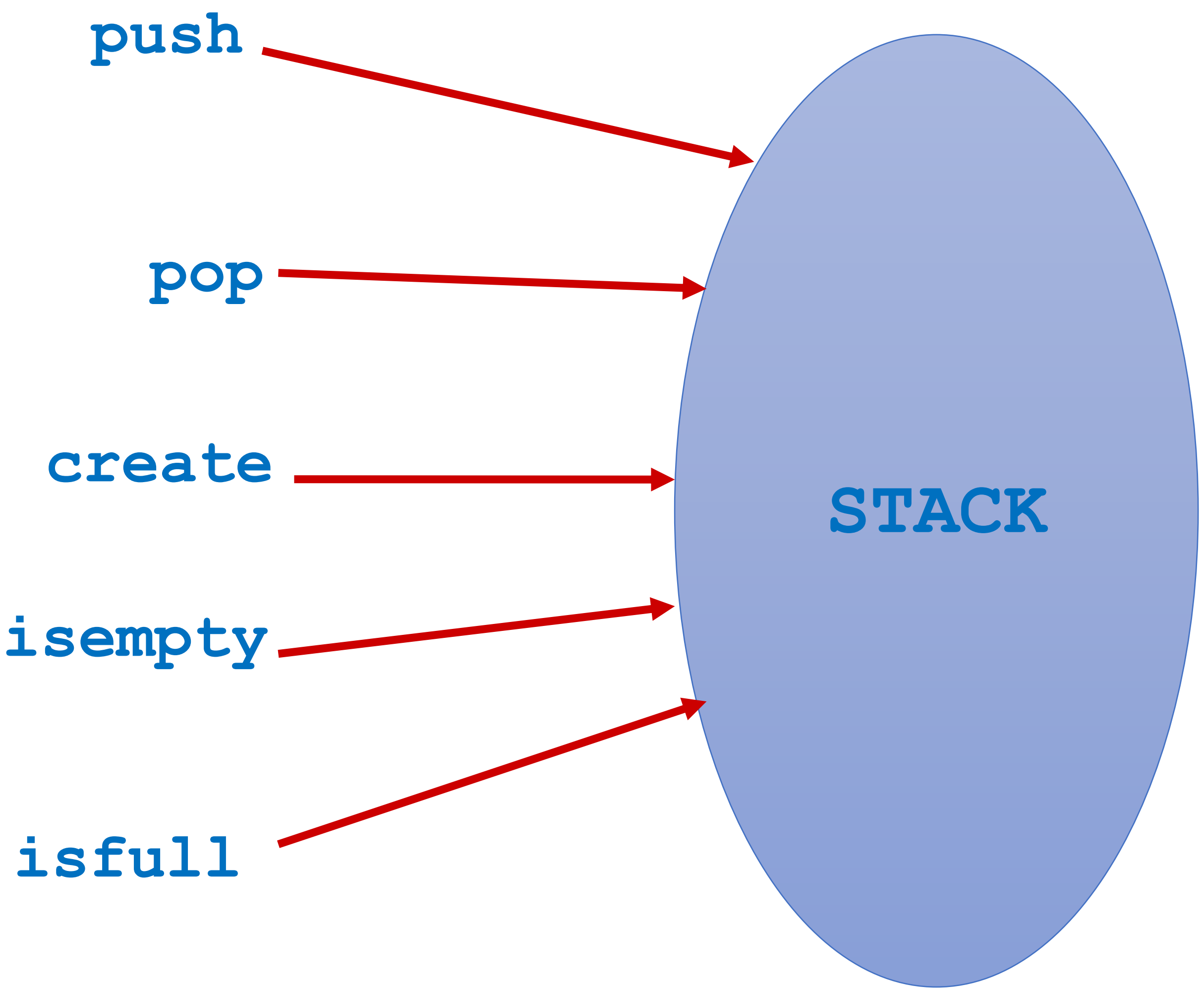
Stacks: Stack, operations, implementation using arrays, recursion, ToH, linked list and Stack Applications: Infix to postfix expression conversion, Evaluation of Postfix expressions, balancing the symbols.

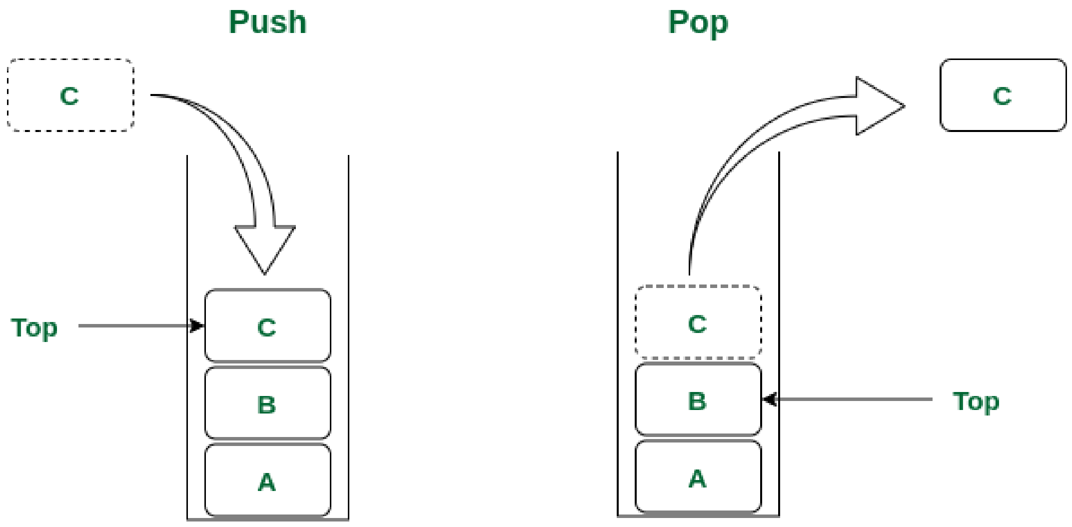
Stack

Basic Idea

- Stacks is a linear data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion and deletion operations from one end of the stack data structure, that called top.
- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.







Stack Data Structure

Stack Operations

1. Push – add an element to a stack

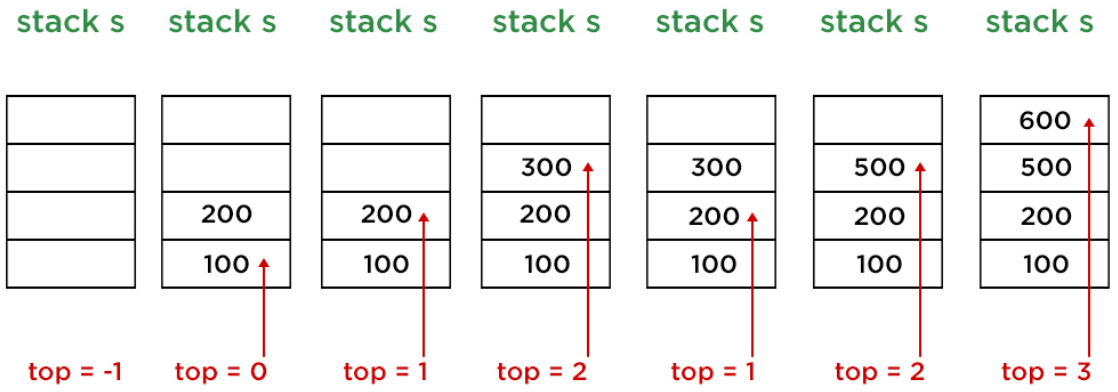
2. Pop -- remove the top element from stack, return or not return the data of top element

3. Peek – get the data of top element of stack, return the value of the top element

All operations work at the top of a stack

Array Representation of Stacks

- Use an element array of MAX size to represent a stack.
- Use a variable TOP to represent the index/or address of the top element of the stack in the array. It is this position from where the element will be added or removed
- TOP = -1 indicates that the stack is empty
- TOP = MAX -1 indicates that the stack is full



Push Operation

- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack.
- First check if **<u>TOP==MAX-1.</u>**
  If true, then it means the stack is full and no more insertions can further be added, an OVERFLOW message is printed.
- If not true, increase TOP by 1, then add the element at TOP position

A	B	C	D	E					
0	1	2	3	TOP = 4	5	6	7	8	9

A	B	C	D	E	F				
0	1	2	3	4	TOP =5	6	7	8	9

Pop Operation

- The pop operation is used to delete the topmost element from the stack.
- First check if **<u>TOP == -1</u>**.
  If true then it means the stack is empty so no more deletions can further be done, an UNDERFLOW message is printed.
  If not true, get the value of the top element, decrease TOP by one.

A	B	C	D	E					
0	1	2	3	TOP = 4	5	6	7	8	9

A	B	C	D						
0	1	2	TOP = 3	4	5	6	7	8	9

Peek Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- The peek operation first checks if the stack is empty or contains some elements.
- If TOP == -1, then an appropriate message is printed else the value is returned.

A	B	C	D	E					
0	1	2	3	TOP = 4	5	6	7	8	9

Here Peek operation will return E, as it is the value of the topmost element of the stack.

Algorithms for Push and Pop Operations

```
Algorithm to PUSH an element in a stack
Step 1: IF TOP = MAX-1, then
          PRINT "OVERFLOW"
          Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

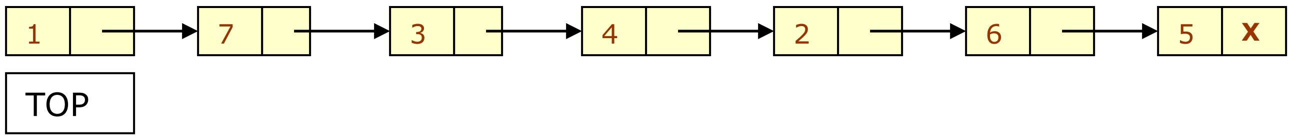
```
Algorithm to POP an element from a stack
Step 1: IF TOP = NULL, then
        PRINT "UNDERFLOW"
        Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Algorithm for Peep Operation

```
Algorithm for Peep Operation

Step 1: IF TOP = NULL, then
        PRINT "STACK IS EMPTY"
        Go TO Step 3
           [END OF IF]
Step 2: RETURN STACK[TOP]
Step 3: END
```

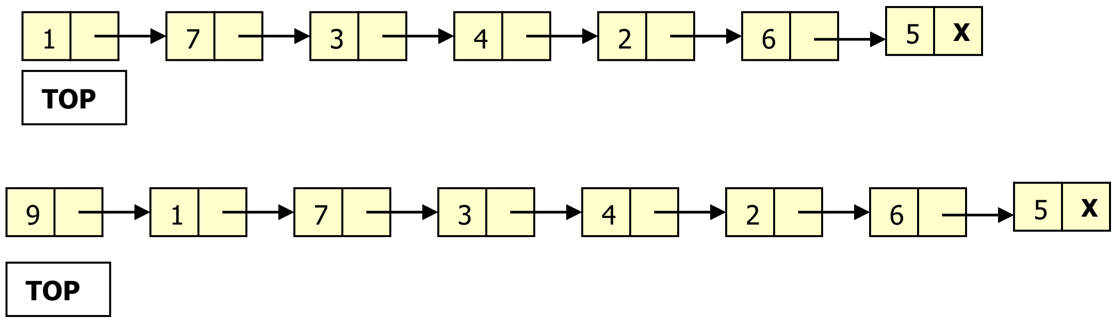
Linked List Representation of Stacks

- 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.
- If TOP is NULL then it indicates that the stack is empty.



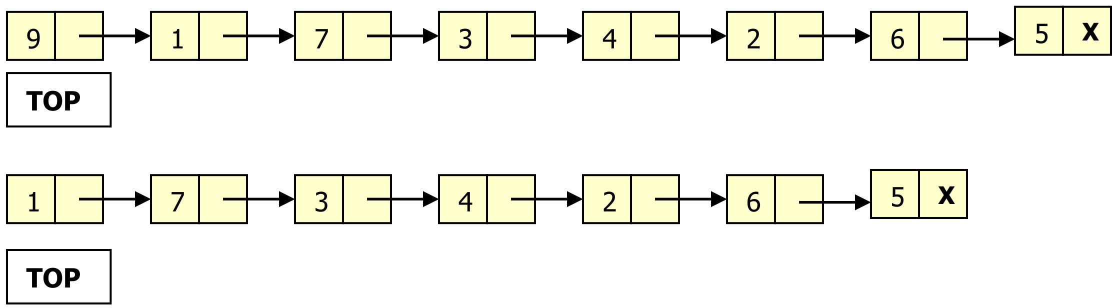
Push Operation on a Linked Stack

```
Algorithm to PUSH an element in a linked stack
Step 1: Allocate memory for the new node and name it as New_Node
Step 2: SET New_Node->DATA = VAL
Step 3: IF TOP = NULL, then
             SET New_Node->NEXT = NULL
             SET TOP = New_Node
        ELSE
             SET New_node->NEXT = TOP
             SET TOP = New_Node
        [END OF IF]
Step 4: END
```



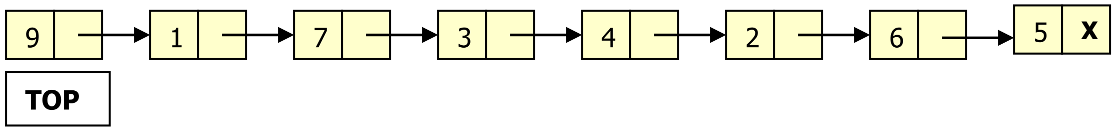
Pop Operation on a Linked Stack

```
Algorithm to POP an element from a stack

Step 1: IF TOP = NULL, then
             PRINT "UNDERFLOW"
                   Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP ->NEXT
Step 4: FREE PTR
Step 5: END
```



Peek Operation on a Linked Stack

```
Algorithm to PEEK an element from a stack

Step 1: IF TOP = NULL, then
             PRINT "UNDERFLOW"
                    Goto Step 5
        [END OF IF]
Step 2: RETURN TOP->data
Step 3  END
```



Declaration

```
#define MAXSIZE 100

struct lifo
{
    int st[MAXSIZE];
    int  top;
};
typedef struct lifo stack;
stack s;
```

ARRAY

```
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo stack;
stack *top;
```

LINKED LIST

Stack Creation

```
void create (stack *s)
{
    s->top = -1;

    /* s->top points to
       last element
       pushed in;
       initially -1 */
}
```

ARRAY

```
void create (stack **top)
{
    *top = NULL;

    /* top points to NULL,
       indicating empty
       stack              */
}
```

LINKED LIST

Pushing an element into stack

```
void push (stack *s, int element)
  {
   if (s->top == (MAXSIZE-1))
   {
     printf ("\n Stack overflow");
         exit(-1);
     }
     else
     {
         s->top++;
         s->st[s->top] = element;
     }
  }
```

ARRAY

```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *)malloc (sizeof(stack));
    if (new == NULL)
    {
       printf ("\n Stack is full");
       exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

LINKED LIST

Popping an element from stack

```
int pop (stack *s)
  {
     if (s->top == -1)
     {
        printf ("\n Stack underflow");
        exit(-1);
     }
     else
     {
        return (s->st[s->top--]);
     }
  }
```

ARRAY

```
int pop (stack **top)
{
   int t;
   stack *p;
   if (*top == NULL)
   {
      printf ("\n Stack is empty");
      exit(-1);
   }
   else
   {
      t = (*top)->value;
      p = *top;
      *top = (*top)->next;
      free (p);
      return t;
   }
}
```

LINKED LIST

Checking for stack empty

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Checking for Stack Full

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Example: A Stack using an Array and Linked list

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;

main() {
  stack A, B;
  create(&A);
  create(&B);
  push(&A,10);
  push(&A,20);
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf ("%d %d", pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf ("\n B is empty");
  return;
}
```

```
#include <stdio.h>
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo stack;

main() {
  stack *A, *B;
  create(&A);
  create(&B);
  push(&A,10);
  push(&A,20);
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf ("%d %d", pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(B))
    printf ("\n B is empty");
  return;
}
```

```c
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{ top=-1;
printf("\n Enter the size of STACK[MAX=100]:"); scanf("%d",&n);
printf("\n\t STACK OPERATIONS USING ARRAY"); printf("\n\t----------------------
---------"); printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
do
{
printf("\n Enter the Choice:"); scanf("%d",&choice);
switch(choice)
{ case 1:
{
push();
break;
}
case 2:
{
pop();
break;
}
case 3:
{
display();
break;
}
case 4:
{
printf("\n\t EXIT POINT ");
break;
}
default:
{ printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
}
}
}
while(choice!=4);
return 0;
}
```

```c
void push()
{
if(top>=n-1)
{
printf("\n\tSTACK is over flow");
}
else
{
printf(" Enter a value to be
pushed:");
scanf("%d",&x);
top++;
stack[top]=x;
}
}
```

```c
void pop()
{
if(top<=-1)
{
printf("\n\t Stack is under flow");
}
else
{
printf("\n\t The popped elements is
%d",stack[top]);
top--;
}
}

void display()
{
if(top>=0)
{
printf("\n The elements in STACK \n");
for(i=top; i>=0; i--)
printf("\n%d",stack[i]);
printf("\n Press Next Choice");
}
else
{
printf("\n The STACK is empty");
}
}
```

```python
class StackUsingArray:
    def __init__(self):
        self.stack = []

    #Method to append the element in the stack at top position.
    def push(self, element):
        self.stack.append(element)

    #Method to Pop last element from the top of the stack
    def pop(self):
        if(not self.isEmpty()):
            lastElement = self.stack[-1] #Save the last element to return
            del(self.stack[-1]) #Remove the last element
            return lastElement
        else:
            return("Stack Already Empty")

    #Method to check if stack is empty or not
    def isEmpty(self):
        return self.stack == []

    def printStack(self):
        print(self.stack)
```

```python
if __name__ == "__main__":
    s = StackUsingArray()
    print("*"*5+" Stack Using Array "+5*"*")
    while(True):
        el = int(input("1 for Push\n2 for Pop\n3 to check if it is Empty\n4 to print Stack\n5 to exit\n"))
        if(el == 1):
            item = input("Enter Element to push in stack\n")
            s.push(item)
        if(el == 2):
            print(s.pop())
        if(el == 3):
            print(s.isEmpty())
        if(el == 4):
            s.printStack()
        if(el == 5):
            break
```

Applications of Stacks

- Direct applications:
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Validate XML

- Indirect applications:
  - Auxiliary data structure for algorithms
  - Component of other data structures

Applications of Stacks

1.Reversing a list

2.Parentheses checker

3.Conversion of an infix expression into a postfix expression

4.Evaluation of a postfix expression

5.Conversion of an infix expression into a prefix expression

6.Evaluation of a postfix expression

7.Recursion

8.Tower of Hanoi

Infix and Postfix

Infix and Postfix Notations

- Infix: operators placed between operands:

  A+B*C

- Postfix: operands appear before their operators:-

  ABC*+

- There are no precedence rules to learn in postfix notation, and parentheses are never needed

Infix to Postfix

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

A + B * C → (A + (B * C)) → (A + (B C *)) → A B C * +

A + B * C + D → ((A + (B * C)) + D) → ((A + (B C*))+ D) → ((A B C *+) + D) → A B C * + D +