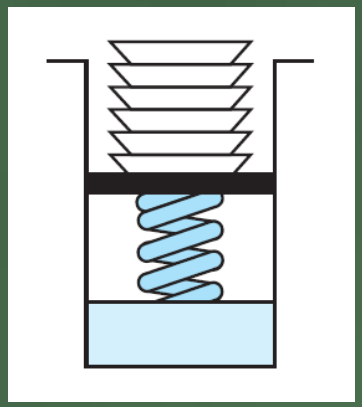
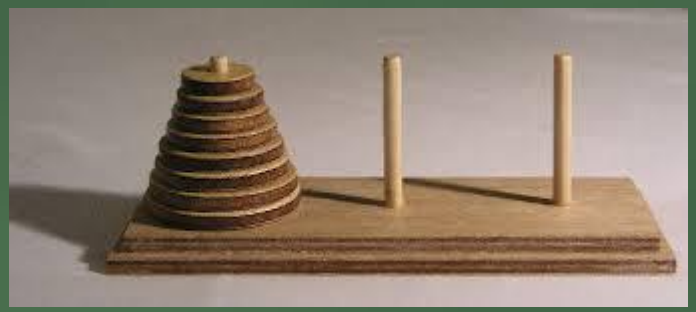
The Stack

Shikha Mehrotra

Introduction of Stack



Dinner plates



Tower of Hanoi

Pack of Tennis balls

- **Stack**
  - A list with the restriction that insertion and deletion can be performed only from one end called the top.

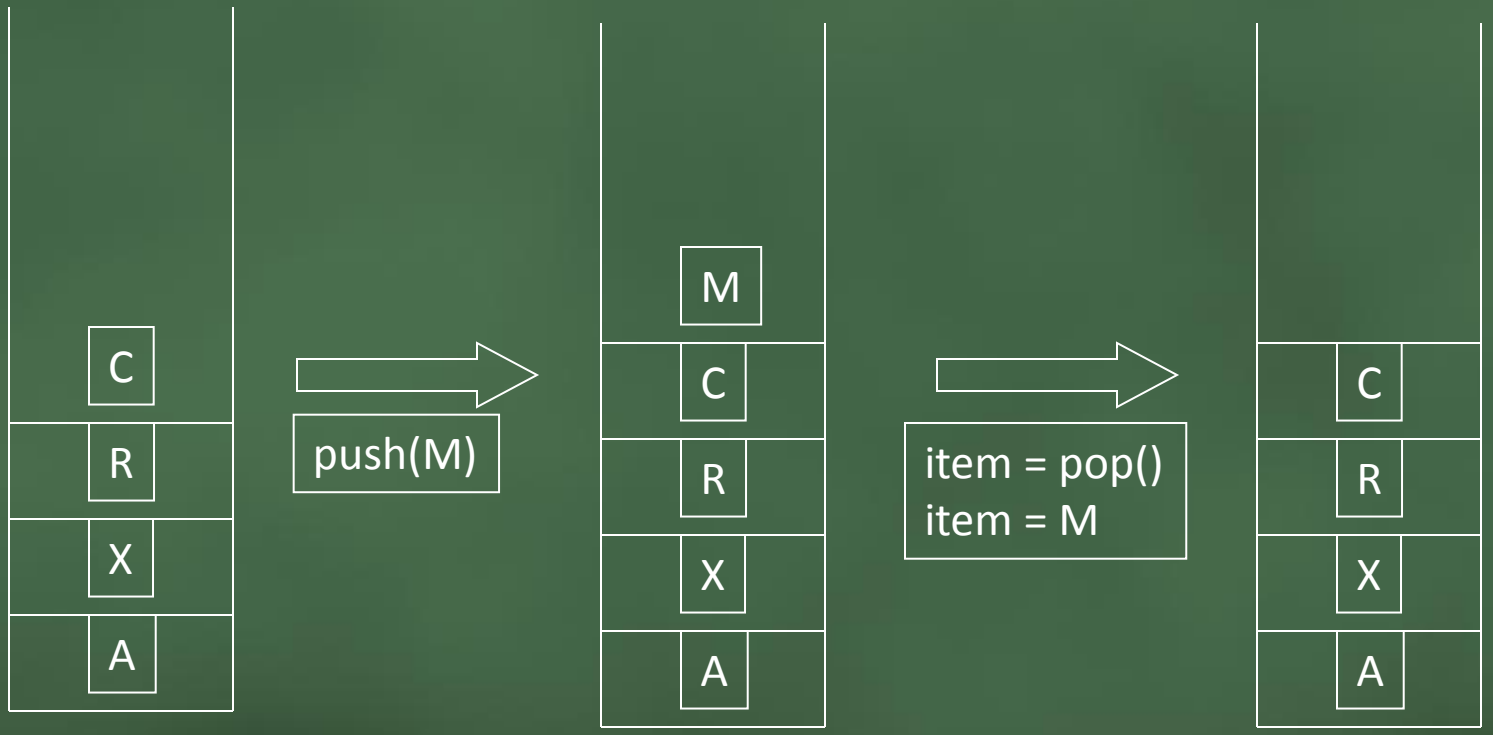
- **Stack**
- The basic implementation of stack is also called LIFO (Last In First Out)
  - It is a list like structure, but elements can be inserted or deleted from only one end. It makes stack less flexible than lists.
  - Many applications need simpler stack rather than lists.
- Only access to the stack is the top element
  - consider trays in a cafeteria
    - to get the bottom tray out, you must first remove all of the elements above

Stack Operations

- *Push*
  - the operation to place a new item at the top of the stack
- *Pop*
  - the operation to remove the top item from the stack
- Top
  - The operation that returns the top element from the stack
- isEmpty
  - The operation to check whether stack is empty or not

Stack

Watch Demo

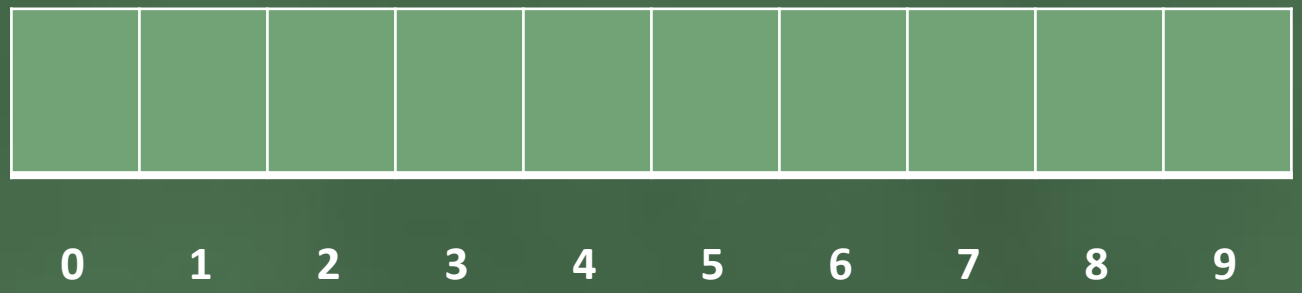


Implementing a Stack

- Different ways to implement a stack
  - Array
  - linked list

Implementation of the Stack using Array

top



int A[10]

top == -1 // empty stack

Implementation of the Stack Operations

top

- Push(int x):

0 1 2 3 4 5 6 7 8 9

```
void push(int x)
{
 if(top==MAX_SIXE -1 )
          return;

 top  = top +1
 A [top] = x
 }
```

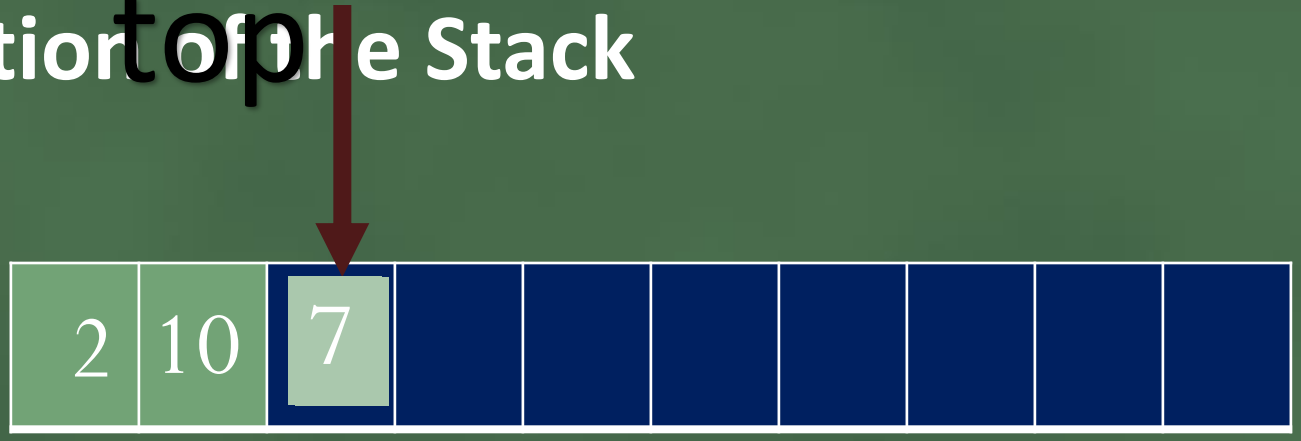
Push(2)

Push(10)

Push(5)

Implementation of the Stack Operations

- Pop():



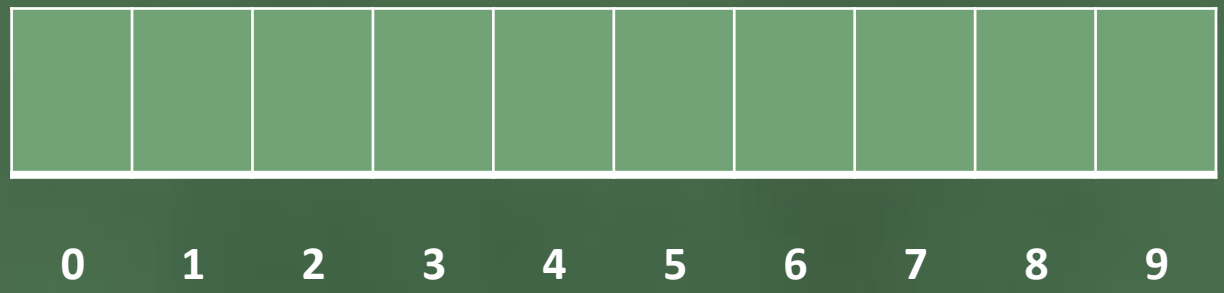

```
int pop()
{
   if(top==-1)
            return -1;
 top = top -1;
}
```

Pop()

Push (7)

Implementation of the Stack Operations

top



- IsEmpty():

```
int IsEmpty()
{
	if(top == -1)
		return 1;
	return 0;
}
```

Implementation of the Stack Operations

Top() / peek()

```
int peek()
{
   if(top==-1)
            return -1;
    return A[top];
}
```

```
#include<stdio.h>

#define MAX_SIZE 101

int A[MAX_SIZE];

int top = -1;

void Push(int x)
{
  if(top == MAX_SIZE -1)
    { // overflow case.
          printf("Error: stack overflow\n") ;
          return;
}
A[++top] = x;
}

void Pop()
{
if(top == -1) {
          printf("Error: No element to pop\n");
          return;
}
top--;
}
```

```
int Top()
{
      return A[top];
}

int IsEmpty()
{
    if(top == -1) return 1;
    return 0;
}

void Print() {
int i;
printf("Stack: ");
for(i = 0;i<=top;i++)
printf("%d ",A[i]);
printf("\n");
}


int main() {
  // Code to test the implementation.
  // calling Print() after each push or pop to see the
Push(2);Print();
Push(5);Print();
Push(10);Print();
Pop();Print();
Push(12);Print();
}
```

Stack: 2
Stack: 2 5
Stack: 2 5 10
Stack: 2 5
Stack: 2 5 12

Stack Applications

- Function Calls / recursion
- UNDO in an editor
- Compilers
    - parsing data between delimiters (Balanced Parentheses)

{ ( ) }

- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a postfix expression

• Polish Notations

- Prefix
- Postfix
- Infix
- Infix to postfix Conversion

•About Polish Notations

- Postfix, or Reverse Polish Notation (RPN) is an alternative to the way we usually write arithmetic expressions (which is called infix, or algebraic notation
  - “Postfix” refers to the fact that the operator is at the end
  - “Infix” refers to the fact that the operator is in between
  - For example, 2 2 +postfix is the same as2 + 2 Infix
  - There is also a seldom-used prefix notation, similar to postfix
- Advantages of postfix:–
  - You don’t need rules of precedence
  - You don’t need rules for right and left associativity
  - you don’t need parentheses to override the above rules
- Advantages of infix:–
  - You grew up with it
  - t’s easier to see visually what is done first

•How to evaluate postfix notation

- Going from left to right, if you see an operator, apply it to the previous two operands (numbers)
- Example:

- Equivalent infix: 2 + 10 * 4 / 5 – (9 – 3)

•Infix Notation

- Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed between the operands. For example, *A+B;* here, plus operator is placed between the two operands A and B.
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- So, computers work more efficiently with expressions written using prefix and postfix notations.

•Postfix Notation

- Postfix notation was given by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation which is better known as Reverse Polish Notation or RPN.
- In postfix notation, the operator is placed after the operands. For example, if an expression is written as *A+B* in infix notation, the same expression can be written as *AB+* in postfix notation.
- The order of evaluation of a postfix expression is always from left to right.

•Postfix Notation

- The expression (A + B) * C is written as:

  AB+C* in the postfix notation.

- A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands.

- No parenthese

- For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication.

•Prefix Notation

- In a prefix notation, the operator is placed before the operands.
- For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Prefix expressions also do not follow the rules of operator precedence, associativity, and even brackets cannot alter the order of evaluation.
- The expression (A + B) * C is written as:
  *+ABC in the prefix notation

- **Arithmetic and Logical Expressions**
- repeatedly scan through the expression !
- take parentheses and priorities of operators into account
- a + b + c * d - e / g
- a + b + (c * d) - (e / g)
- a + ((b + c) * d - e) / g
- a + b <= c && a + b <= d
- (a + b <= c) || (a + b <= d)

•The Polish Notations

- Q : How can a compiler accept an expression and produce correct code ?
- A : Tranforming the expression into a form called Polish notation

Infix form	Prefix form	Postfix form
a * b	* a b	a b *
a + b * c	+ a * b c	a b c * +
(a + b) * c	* + a b c	a b + c *

Reverse Polish notation

•Expression Evaluations : Stacks

```
5 * ( ( ( 9 + 8 ) + ( 4 * 6 ) ) - 7 )
Postfix form : 5 9 8 + 4 6 * + 7 - *

Push( 5 )
Push( 9 )
Push( 8 )
Push( Pop() + Pop() )
Push( 4 )
Push( 6 )
Push( Pop() * Pop() )
Push( Pop() + Pop() )
Push( 7 )
Push( Pop() - Pop() )
Push( Pop() * Pop() )
```

•Expression Evaluations : Stacks

5 * (((9 + 8) + (4 * 6)) - 7)

Postfix form : ***5 9 8 + 4 6 * + 7 - ****