# Unit 5
# Stack

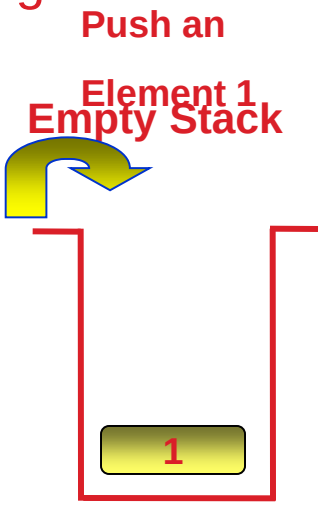
Prof. Pujashree Vidap

- In this session, you will learn to:
  - Identify the features of a stack
  - Implement stacks
  - Apply stacks to solve programming problems

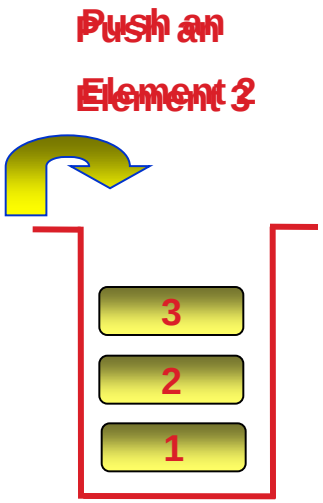
- What is a Stack ?

- A stack is a collection of data items that can be accessed at only one end, called top.
- Items can be inserted and deleted in a stack only at the top.
- The last item inserted in a stack is the first one to be deleted.
- Therefore, a stack is called a Last-In-First-Out (LIFO) data structure.
- It is also known as restricted linear list.

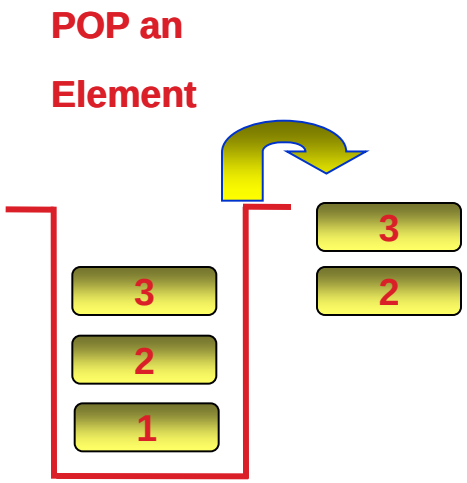
- There are two basic operations that are performed on stacks:
  - PUSH
  - POP
- **PUSH**: It is the process of inserting a new element on the top of a stack.



- **PUSH**: It is the process of inserting a new element on the top of a stack.



- **POP**: It is the process of deleting an element from the top of a stack.



- Elements in stacks are inserted and deleted on a _____ basis.

- Answer:
  - LIFO

- List down some real life examples that work on the LIFO principle.
- Answer:
  - **Pile of books**: Suppose a set of books are placed one over the other in a pile. When you remove books from the pile, the topmost book will be removed first. Similarly, when you have to add a book to the pile, the book will be placed at the top of the pile.
  - **Pile of plates**: The first plate begins the pile. The second plate is placed on the top of the first plate and the third plate is placed on the top of the second plate, and so on. In general, if you want to add a plate to the pile, you can keep it on the top of the pile. Similarly, if you want to remove a plate, you can remove the plate from the top of the pile.
  - **Bangles in a hand**: When a person wears bangles, the last bangle worn is the first one to be removed.

# ADT Stack

```
class Stack
{
finite ordered list  with 0 or more elements.
int top;
public:
Stack();//create an empty stack and initializes top;
void push( ele);//to insert element into stack;
void pop();//to remove element from stack;
int top();//returns value of top
Isempty();//check wheather stack is empty or not
Isfull();
}
```

- IMPLEMENTATION OF STACK

  Stack can be implemented using

  1) arrays (STATIC IMPLEMENTATION)

  2) linked lists.(DYNAMIC IMPLEMENTATION)

- To implement a stack using an array:
  - Declare an array:

    ```
    int Stack[5]; // Maximum size needs to be specified in
    // advance
    ```

  - Declare a variable, **top to hold** the index of the topmost element in the stacks:

    ```
    int top;
    ```

  - Initially, when the stack is empty, set:

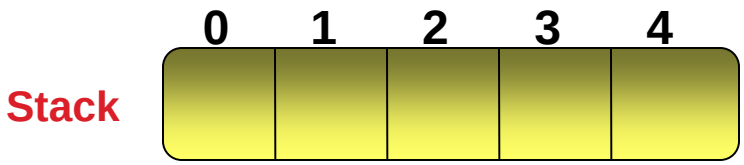
    top = –1

- Let us now write an algorithm for the PUSH operation.

**Initially:**

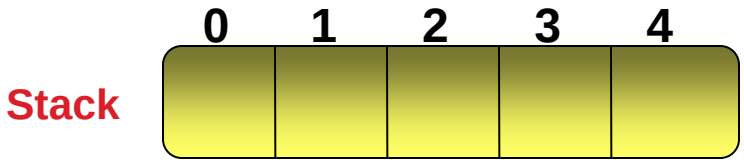
**top = – 1**

**PUSH an element 3**



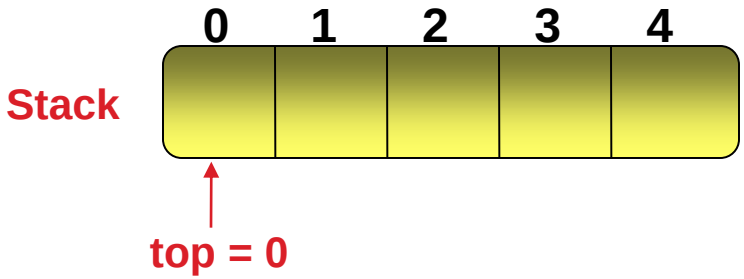
**top = – 1**

**PUSH an element 3**

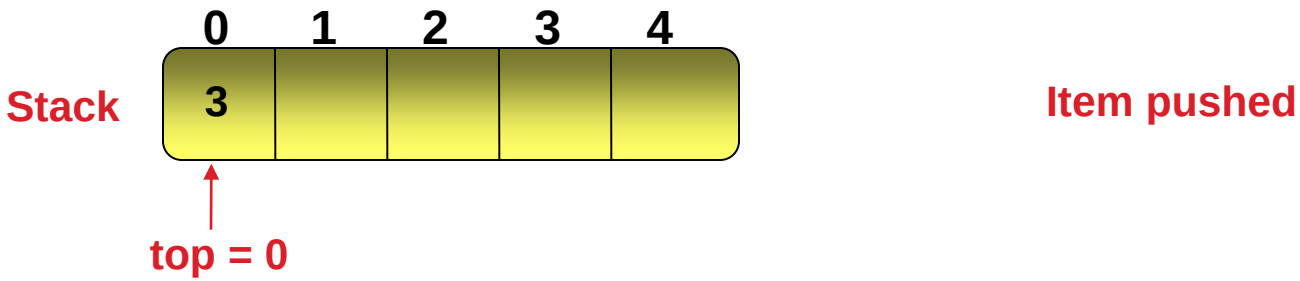


**top = 0**

**PUSH an element 3**

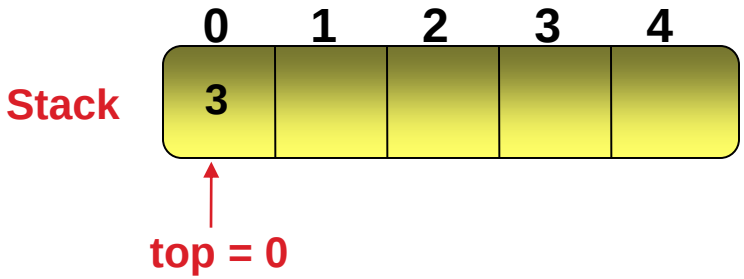


PUSH an element 3

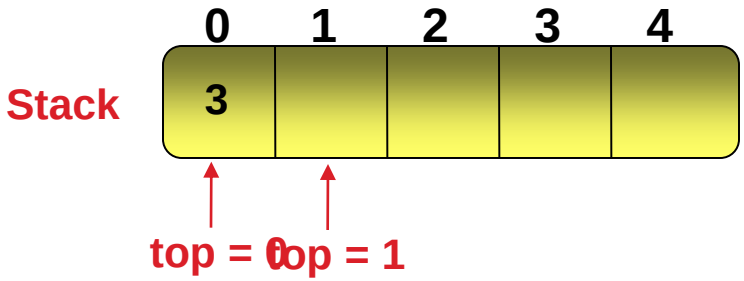



Item pushed

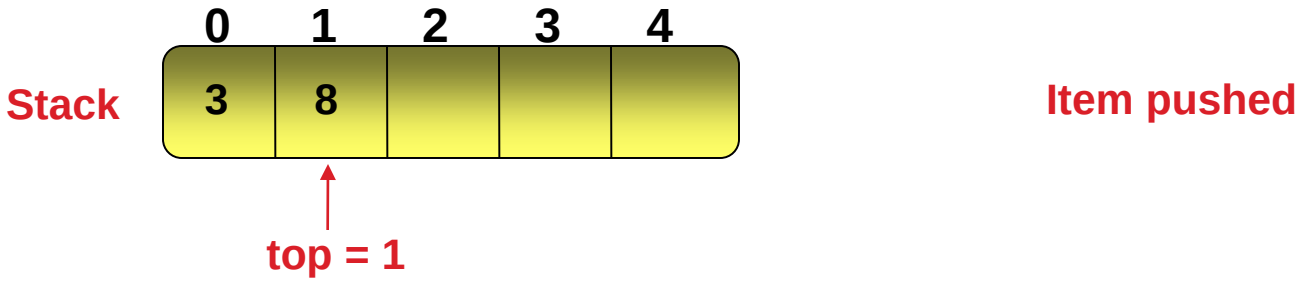
PUSH an element 8



PUSH an element 8

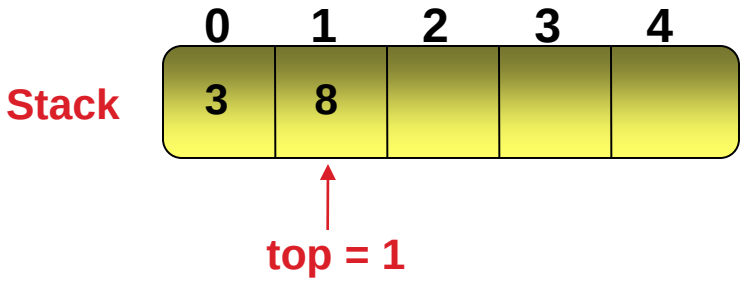


PUSH an element 8

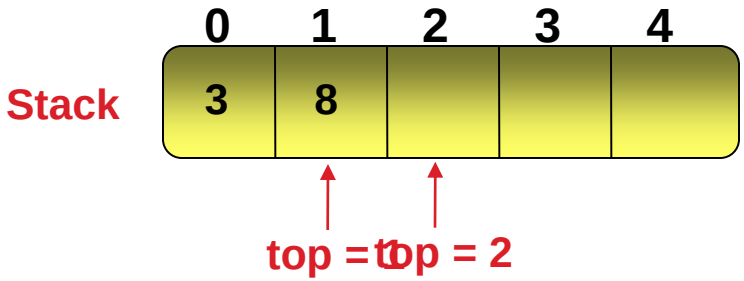



Item pushed

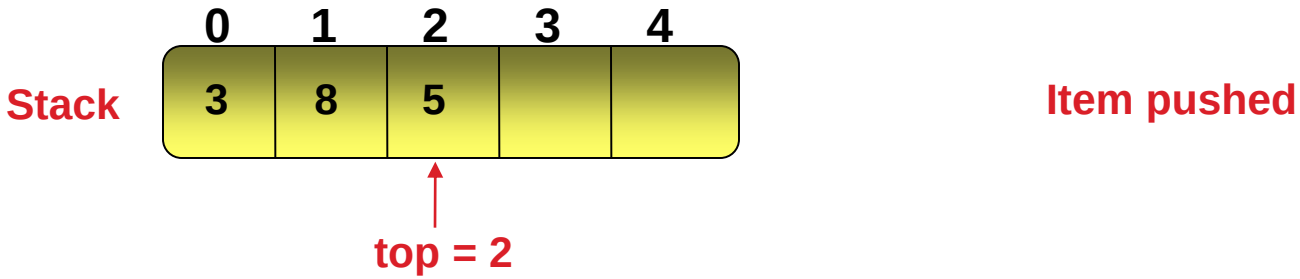
PUSH an element 5



PUSH an element 5

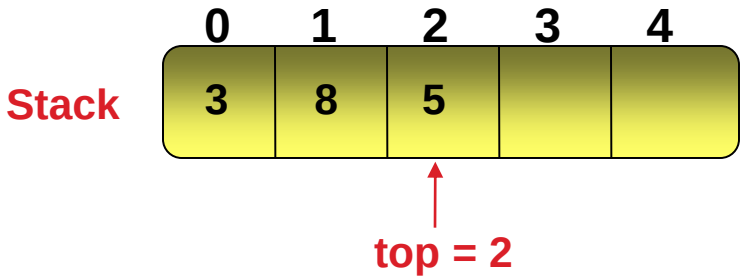


PUSH an element 5

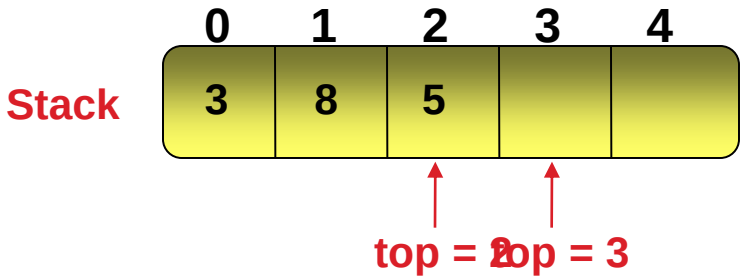



Item pushed

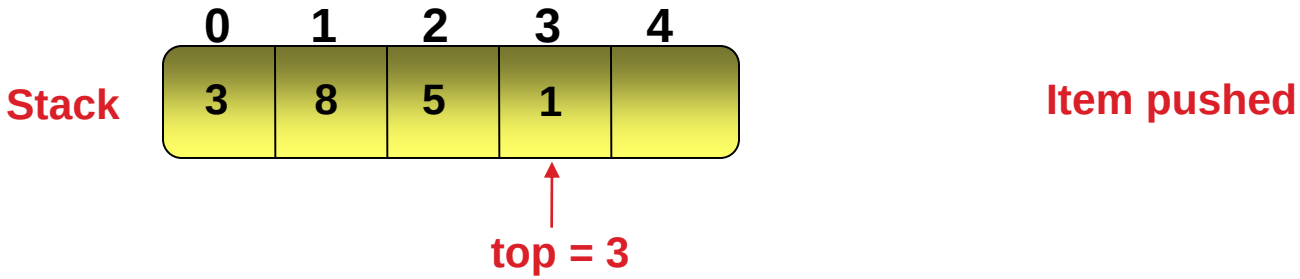
PUSH an element 1



PUSH an element 1

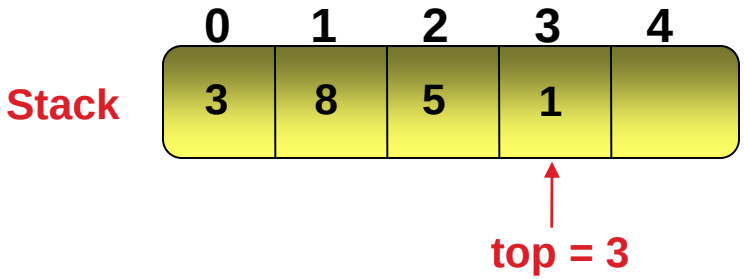


PUSH an element 1

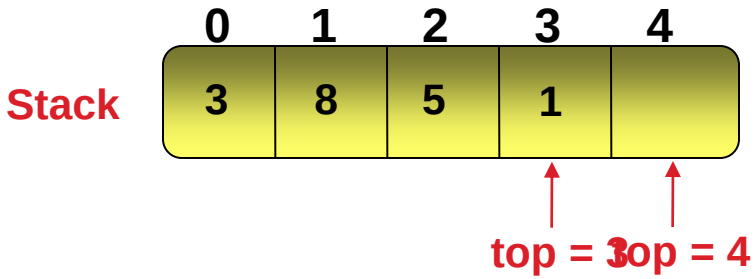



Item pushed

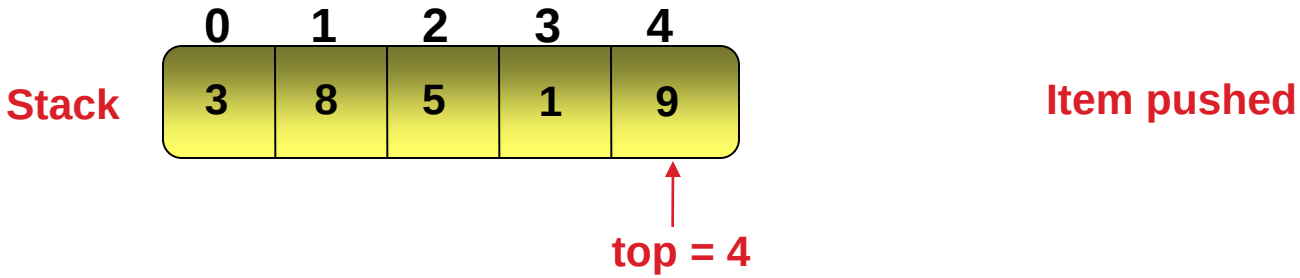
PUSH an element 9



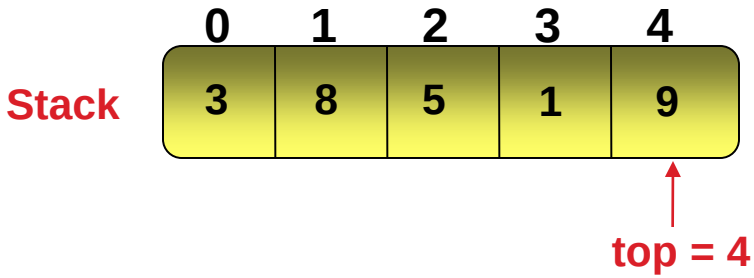
PUSH an element 9



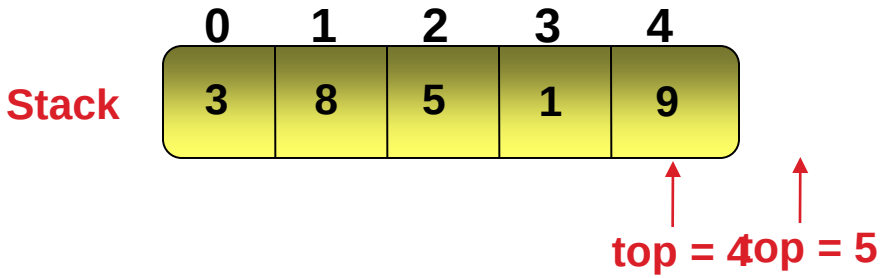
PUSH an element 9



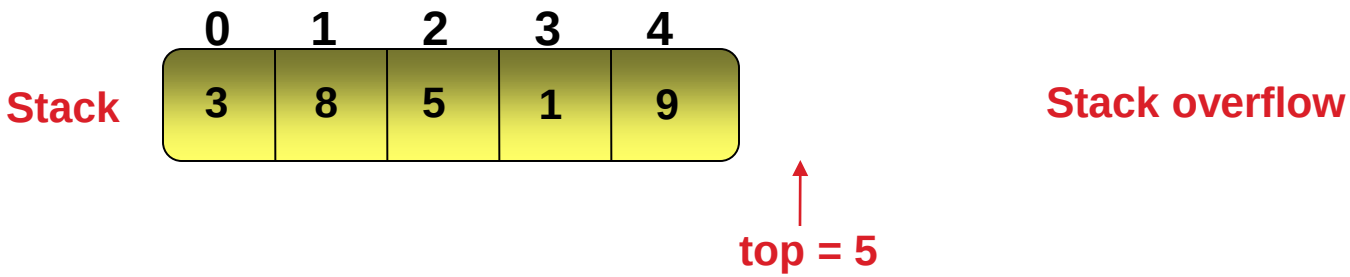
PUSH an element 2



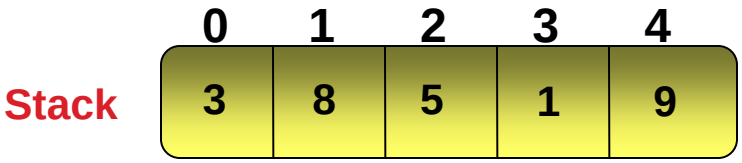
PUSH an element 2



PUSH an element 2



- The stack has been implemented in an array of size 5.
- Therefore, you cannot store more than 5 elements in the stack.
- To avoid the stack overflow, you need to check for the stack full condition before pushing an element into the stack.



# ALGORITHM TO PUSH AN ELEMENT INTO STACK

TOP=-1 TO REPRESENT STACK IS EMPTY

Algorithm PUSH(STACK[N],TOP,ITEM)

{

//STACK is an array of size N ,ITEM is element to be inserted, TOP represents position.

2. if (TOP == N-1)

    1.1 Print " Stack Overflow "

3. Else

    3.1TOP=TOP+1

    3.2 Enter ITEM

    3.3STACK[TOP] = ITEM

}

- **Write an algorithm to implement the POP operation on a stack.**

**Algorithm POP(STACK[N],TOP,ITEM)**

**{**

**1. if( TOP == – 1)**

**1.1 Print “Stack Empty”**

**2. else**

**2.1 ITEM = STACK[TOP]**

**2.2 TOP = TOP-1**

**Return ITEM**

**}**

- In a stack, data can be stored and removed only from one end of the stack called the _____ of the stack.
  - Answer:
    - top

- Some of the applications of stacks are:
  - Implementing function calls
  - Maintaining the UNDO list for an application
  - Checking the nesting of parentheses in an expression
  - Evaluating expressions
  - Simulating recursion
  - Backtracking algorithms.
  - Reversing a string.

## Implementing function calls:

- Consider an example. There are three functions, F1, F2, and F3. Function F1 invokes F2 and function F2 invokes F3, as shown.

```
                void F1()
                {
                        int x;
                        x = 5;
                        F2();
1100                    print(x);
1101            }
1102    void F2(int x)
1103    {
                        x = x + 5;
                        F3(x);
                        print(x);
1120    }
1121    void F3(int x)
1122    {
                        x = x × 2;
                        print x;
        }
1140
1141
```

Assuming these instructions at the given locations in the memory.

```
     void F1()
     {
              int x;
              x = 5;
              F2();
1100          print(x);
1101 }
1102 void F2(int x)
1103 {
              x = x + 5;
              F3(x);
              print(x);
1120 }
1121 void F3(int x)
1122 {
              x = x × 2;
              print x;
     }
1140
1141
```

The execution starts from function F1

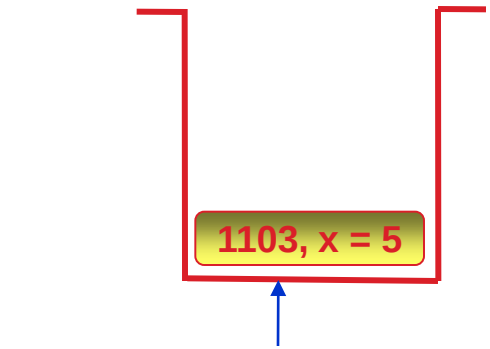
```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2();
1103         print(x);
     }
     void F2(int x)
     {
1120         x = x + 5;
1121         F3(x);
1122         print(x);
     }
     void F3(int x)
     {
1140         x = x × 2;
1141         print x;
     }
```

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2();
1103         print(x);
     }
     void F2(int x)
     {
1120         x = x + 5;
1121         F3(x);
1122         print(x);
     }
     void F3(int x)
     {
1140         x = x × 2;
1141         print x;
     }
```

x = 5

```
     void F1()
     {
1100          int x;
1101          x = 5;
1102          F2();
1103          print(x);
     }
     void F2(int x)
     {
1120          x = x + 5;
1121          F3(x);
1122          print(x);
     }
     void F3(int x)
     {
1140          x = x × 2;
1141          print x;
     }
```

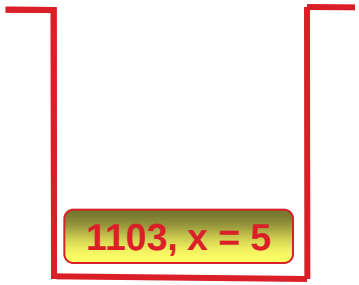
x = 5




Address and the local variable of F1

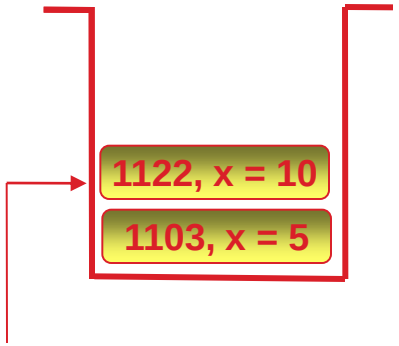
```
     void F1()
     {
1100          int x;
1101          x = 5;
1102          F2();
1103          print(x);
     }
     void F2(int x)
     {
1120          x = x + 5;
1121          F3(x);
1122          print(x);
     }
     void F3(int x)
     {
1140          x = x × 2;
1141          print x;
     }
```

x = 5 10



```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2();
1103         print(x);
     }
     void F2(int x)
     {
1120         x = x + 5;
1121         F3(x);
1122         print(x);
     }
     void F3(int x)
     {
1140         x = x × 2;
1141         print x;
     }
```

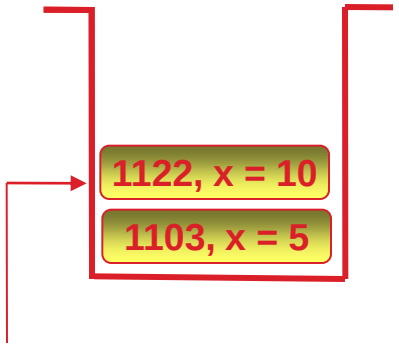
x = 10




Address and the local variable of F2

```
     void F1()
     {
1100          int x;
1101          x = 5;
1102          F2();
1103          print(x);
     }
     void F2(int x)
     {
1120          x = x + 5;
1121          F3(x);
1122          print(x);
     }
     void F3(int x)
     {
1140          x = x × 2;
1141          print x;
     }
```

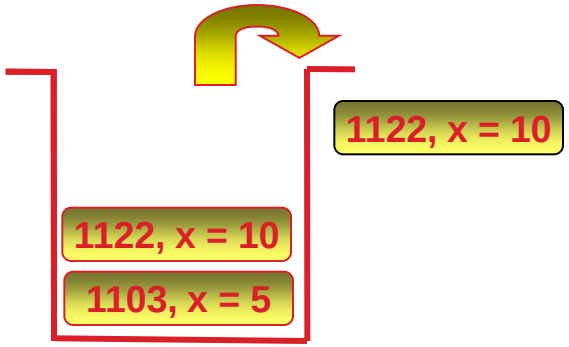
x = 20




Address and the local variable of F2

```
     void F1()
     {
1100          int x;
1101          x = 5;
1102          F2();
1103          print(x);
     }
     void F2(int x)
     {
1120          x = x + 5;
1121          F3(x);
1122          print(x);
     }
     void F3(int x)
     {
1140          x = x * 2;
1141          print x;
     }
```

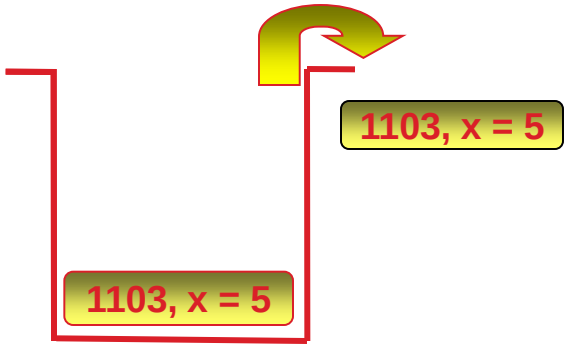
x = 20




20

```
     void F1()
     {
1100          int x;
1101          x = 5;
1102          F2();
1103          print(x);
     }
     void F2(int x)
     {
1120          x = x + 5;
1121          F3(x);
1122          print(x);
     }
     void F3(int x)
     {
1140          x = x * 2;
1141          print x;
     }
```

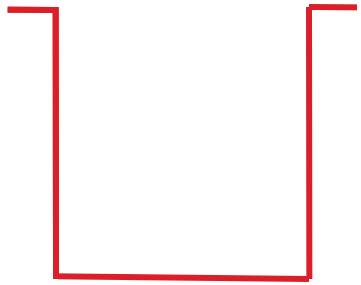
x = 5 10




20 10

```
     void F1()
     {
1100          int x;
1101          x = 5;
1102          F2();
1103          print(x);
     }
     void F2(int x)
     {
1120          x = x + 5;
1121          F3(x);
1122          print(x);
     }
     void F3(int x)
     {
1140          x = x × 2;
1141          print x;
     }
```

x = 5



20 10 5

## Maintaining the UNDO list for an application:

- Consider that you made some changes in a Word document. Now, you want to revert back those changes. You can revert those changes with the help of an UNDO feature.
- The UNDO feature reverts the changes in a LIFO manner. This means that the change that was made last is the first one to be reverted.
- You can implement the UNDO list by using a stack.

## Checking the nesting of parentheses in an expression

You can do this by checking the following two conditions:

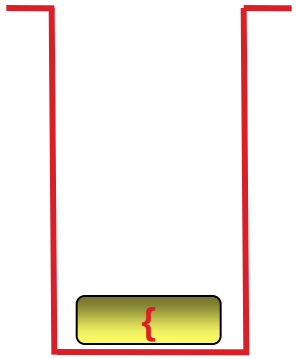
- The number of left parenthesis should be equal to the number of right parenthesis.
- Each right parenthesis is preceded by a matching left parenthesis.

- You need to develop a method to check if the parentheses in an arithmetic expression are correctly nested.
- How will you solve this problem?
- You can solve this problem easily by using a stack.

{(a + b) × (c + d) ]}

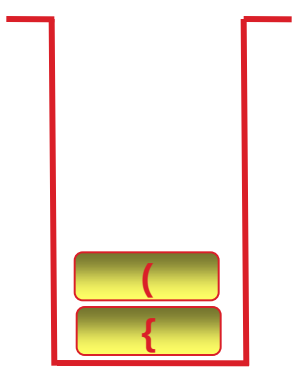
- Consider an example.
- Suppose the expression is:

  {(a + b) × (c + d) ]}
- Scan the expression from left to right.
- The first entry to be scanned is ‘{’, which is a left parenthesis.
- Push it into the stack.



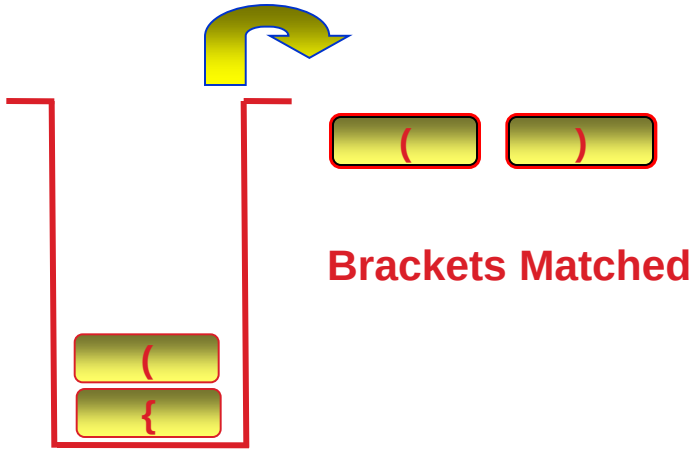
{(a + b) × (c + d) ]}

- The next entry to be scanned is ‘(’, which is a left parenthesis.
- Push it into the stack.
- The next entry is ‘a’, which is an operand. Therefore, it is discarded.
- The next entry is ‘+’, which is an operator. Therefore, it is discarded.
- The next entry is ‘b’, which is an operand. Therefore, it is discarded.



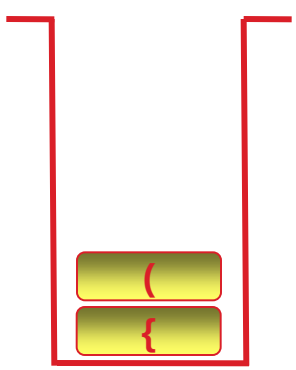
{(a + b) × (c + d) ]}

- The next entry to be scanned is ‘)’, which is a right parenthesis
- POP the topmost entry from the stack.
- Match the two brackets.



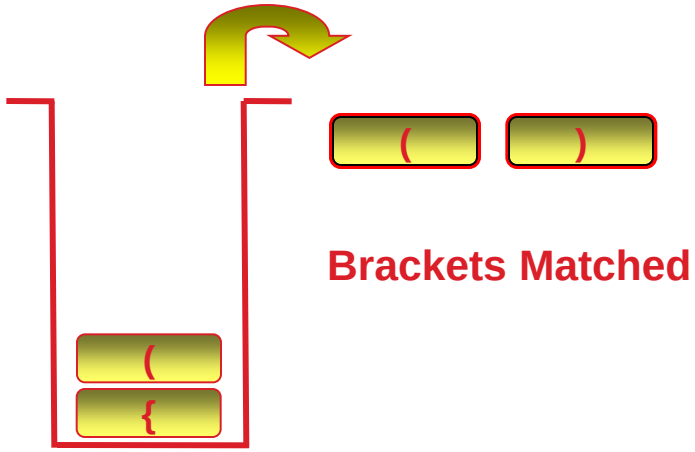
{(a + b) × (c + d) ]}

- The next entry to be scanned is ‘×’, which is an operator. Therefore, it is discarded.
- The next entry to be scanned is ‘(’, which is a left parenthesis
- Push it into the stack
- The next entry to be scanned is ‘c’, which is an operand. Therefore it is discarded
- The next entry to be scanned is ‘+’, which is an operator. Therefore it is discarded
- The next entry to be scanned is ‘d’, which is an operand. Therefore it is discarded



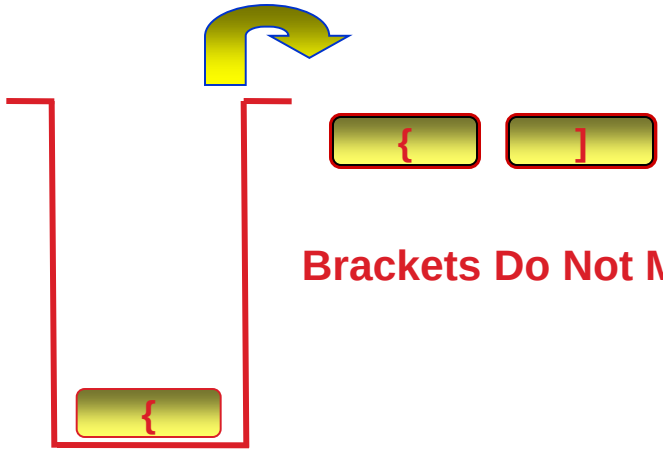
{(a + b) × (c + d) ]}

- The next entry to be scanned is ‘)’, which is a right parenthesis.
- POP the topmost element from the stack.
- Match the two brackets.



{(a + b) × (c + d) ]}

- The next entry to be scanned is ‘]’, which is a right parenthesis.
- POP the topmost element from the stack.
- Match the two brackets.




**Brackets Do Not Match**

**The Expression is INVALID**

# Algorithm for Balanced Symbol Checking

- Make an empty stack
- read symbols until end of file
  - if the symbol is an opening symbol push it onto the stack
  - if it is a closing symbol do the following
    - if the stack is empty report an error
    - otherwise pop the stack. If the symbol popped does not match the closing symbol report an error
- At the end of the file if the stack is not empty report an error

## Evaluating an expression by using stacks:

- Stacks can be used to solve complex arithmetic expressions.
- The evaluation of an expression is done in two steps:
  - Conversion of the infix expression into a postfix expression.
  - Evaluation of the postfix expression.

In this session, you learned that:

- A stack is a collection of data items that can be accessed at only one end, called top. The last item inserted in a stack is the first one to be deleted.
- A stack is called a LIFO data structure.
- There are two operations that can be performed on stacks. They are:
  - **PUSH**
  - **POP**
- Stacks can be implemented by using both arrays and linked lists
- Stacks are used in many applications. Some of the application domains of stacks are as follows:
  - Implementing function calls
  - Maintaining the UNDO list for an application
  - Checking the nesting of parentheses in an expression
  - Evaluating expressions

# Expression evaluation

- Infix expressions
  - An operator appears between its operands
    - Example: a + b
    - Prefix expressions
    - An operator appears before its operands
    - Example: + a b
- Postfix expressions
  - An operator appears after its operands
    - Example: a b +

- To convert a fully parenthesized infix expression to a prefix form
  - Move each operator to the position marked by its corresponding open parenthesis
  - Remove the parentheses
  - Example
    - Infix expression: ( ( a + b ) * c )
    - Prefix expression: * + a b c

- To convert a fully parenthesized infix expression to a postfix form
  - Move each operator to the position marked by its corresponding closing parenthesis
  - Remove the parentheses
  - Example
    - Infix form: ( ( a + b ) * c )
    - Postfix form: a b + c *

- Prefix and postfix expressions
  - Never need
    - Precedence rules
    - Association rules
    - Parentheses
  - Have
    - Simple grammar expressions
    - Straightforward recognition and evaluation algorithms

## Evaluating an expression by using stacks:

- Stacks can be used to solve complex arithmetic expressions.
- The evaluation of an expression is done in two steps:
  - Conversion of the infix expression into a postfix expression.
  - Evaluation of the postfix expression.

# Various convertion

- Infix to postfix
- Infix to prefix
- Prefix to infix
- Prefix to postfix
- Postfix to infix
- Postfix to prefix

# Infix to postfix algorithm

```
for (each character ch in the infix expression){
  switch(ch){
    case operand:     // append operand to end of PE
      postfixExp = postfixExp + ch
      break
    case '(':         // save '(' on stack
      aStack.push(ch)
      break
    case ')':         // pop stack until matching '('
      while (top of stack is not '('){
        postfixExp = postfixExp + (top of aStack)
        aStack.pop()
      }  // end while
      aStack.pop()    // remove the '('
      break
    case operator:    // process stack operators of
                      // greater precedence
      while (!aStack.isEmpty() and
              top of stack is not '(' and
              precedence(ch) <= precedence(top of aStack)){
        postfixExp = postfixExp + (top of aStack)
        aStack.pop()
      } // end while
      aStack.push(ch)    // save new operator
      break
  } // end switch
} // end for
```

```
// append to postfixExp the operators remaining on the stack
while(!aStack.isEmpty()){
  postfixExp = postfixExp + (top of aStack)
  aStack.pop()
} // end while
```

# Convert infix to postfix

- 4-1%8+6%(3+1)*2+(4*(4+(2-5^2)))
- 5*(6+2)-12/4

- Postfix : 418%631+%2*44252^-+*++-
- Value : 75

# Evaluating postfix expression

- Start with an empty stack.
- We scan P from left to right.
- While (we have not reached the end of P)
- If an operand is found push it onto the stack End-If
- If an operator is found
  - Pop the stack and call the value A
  - Pop the stack and call the value B
  - Evaluate B op A using the operator just found.
  - Push the resulting value onto the stack
- End-If

End-While Pop the stack (this is the final value)

| Precedence | Operator | Type | Associativity |
|---|---|---|---|
| 15 | ()<br>[]<br>. | Parentheses<br>Array subscript<br>Member selection | Left to Right |
| 14 | ++<br>-- | Unary post-increment<br>Unary post-decrement | Right to left |
| 13 | ++<br>--<br>+<br>-<br>!<br>~<br>( type ) | Unary pre-increment<br>Unary pre-decrement<br>Unary plus<br>Unary minus<br>Unary logical negation<br>Unary bitwise complement<br>Unary type cast | Right to left |
| 12 | *<br>/<br>% | Multiplication<br>Division<br>Modulus | Left to right |
| 11 | +<br>- | Addition<br>Subtraction | Left to right |
| 10 | <<<br>>><br>>>> | Bitwise left shift<br>Bitwise right shift with sign extension<br>Bitwise right shift with zero extension | Left to right |
| 9 | <<br><=<br>><br>>=<br>instanceof | Relational less than<br>Relational less than or equal<br>Relational greater than<br>Relational greater than or equal<br>Type comparison (objects only) | Left to right |
| 8 | ==<br>!= | Relational is equal to<br>Relational is not equal to | Left to right |
| 7 | & | Bitwise AND | Left to right |
| 6 | ^ | Bitwise exclusive OR | Left to right |
| 5 | \| | Bitwise inclusive OR | Left to right |
| 4 | && | Logical AND | Left to right |
| 3 | \|\| | Logical OR | Left to right |
| 2 | ? : | Ternary conditional | Right to left |
| 1 | =<br>+=<br>-=<br>*=<br>/=<br>%= | Assignment<br>Addition assignment<br>Subtraction assignment<br>Multiplication assignment<br>Division assignment<br>Modulus assignment | Right to left |

# Polish notation

- Its notation form for expressing arithmetic, logical and algebraic equations.
- Here operators are placed left side of operands
- Its free of brackets and parenthesis
- Lesser ambiguity
- Prefix & postfix

# Multiple stack

- Several stack can be stored in a single array.
- Suppose declare an array of int  a[15]
- We have to create 3 stack of same size.
- Lower bound of each  array can be stored in array c
- Position of top  of each stack can be stored in b[i].

```
for(i=0;i<n; i++)
     c[i]=((max/n)*i));
      b[i]=c[i]-1;
```

- Algorithm push(stackno,int h)

```
{ i=stackno
If(b[i]==c[i+1]-1 &&i!=n-1)
  print “overflow”
else if(i==n-1 &&b[i]==max-1)
  Print ”overflow”
 else
{
b[i]=b[i]+1
 a[b[i]]=h;
}
```

```
Algorithm pop(stackno)
{i=stackno
if(b[i]=c[i]-1)
  write “underflow”
else
{
ele=a[b[i]]
Write “element deleted ele”
b[i]=b[i]-1
}
```

- Algorithm Display(stackno)

```
i=stack no;
if(b[i]=c[i]-1)
Print("no element')
else
For(j=c[i];j<=b[i];j++)
 write a[j]
```

# Linked stack

- Push-insertion at beginning
- Pop-deletion from top
- Display

Write the pseudo code for the same.