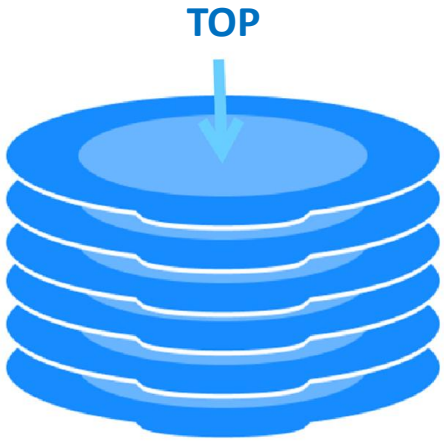
Data Structure & Algorithms

Stack

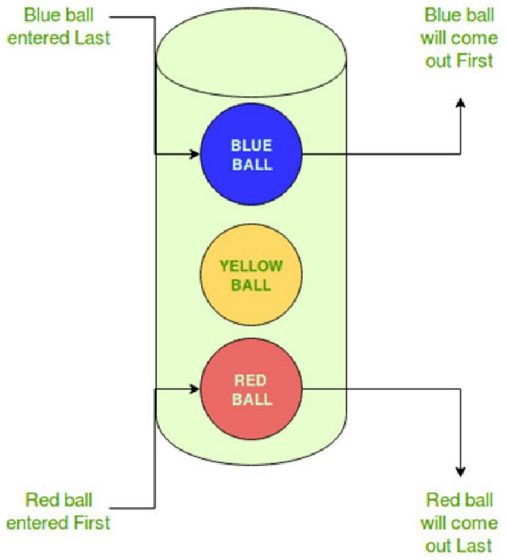
What is Stack?

- A Stack is a linear data structure which follows a particular order in which the operations are performed. The order is LIFO(Last In First Out).
- A good example of a stack is when we put some plates over one another, the plate which is at the top is the first one to be removed and the plate which has been placed at the bottommost position remains in the stack for the longest period of time.
- The top of the stack is **where your next plate goes when putting, and from where you take a plate.**



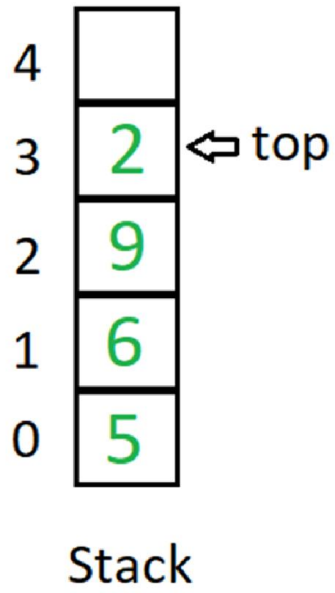
LIFO

- **LIFO** is an abbreviation for **last in, first out**. It is a method for handling data structures where the **first element** is processed last and the **last element** is processed first.
- Take a look at this example:



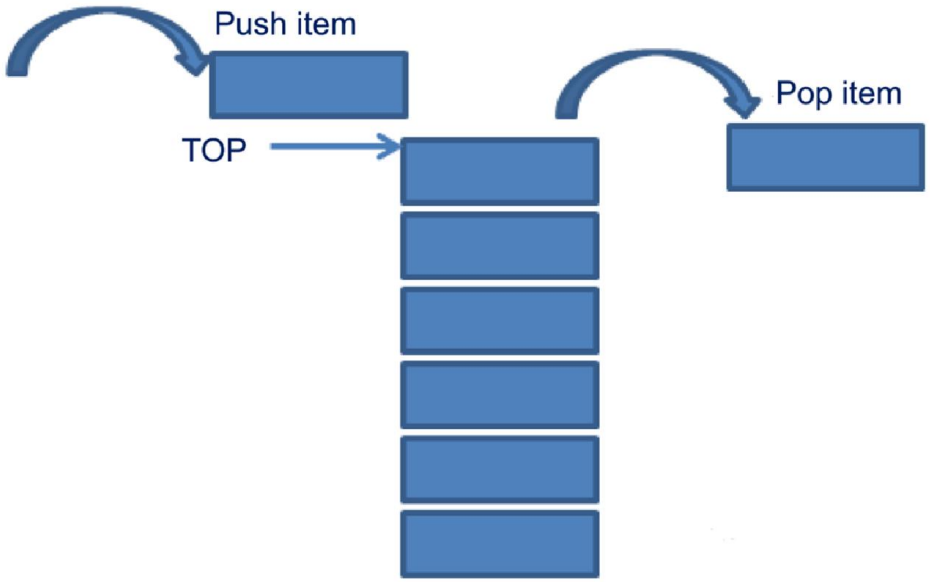
Implementation of Stack

- We can use an array to store the components of a Stack:



Operations on Stack

- Mainly there are 3 basic operations performed in a stack:
  1. Push
  2. Pop
  3. Peek or Top



Push Operation

Push operation refers to **inserting** an element in the stack. Since there's only one position at which the new element can be inserted — Top of the stack, the new element is inserted at the top of the stack, just as a plate being put at the top of other plates:

- Step 1 − Check if the stack is full.
- Step 2 − If the stack is full, produce overflow error and exit.
- Step 3 − If the stack is not full, increment top pointer to point the next empty space.
- Step 4 − Add data element to the stack location, where the top is pointing.
- Step 5 − return success

Push Operation - Algorithm

```
int items[maxsize];
int top;
int Push (int item){
  if(top == maxsize){
      print("stack is full!");
      return -1;   //Error!
  }

   top++;
   items[top] = item;
   return 0;    //Success!
}
```

Pop Operation

Pop operation refers to the removal of an element. Again, since we only have access to the element at the top of the stack, there's only one element that we can remove. We just remove the top of the stack.

- **Note:** We can also choose to return the value of the popped element back.

- Step 1 − Check if the stack is empty.
- Step 2 − If the stack is empty, produce underflow error and exit.
- Step 3 − If the stack is not empty, access the data where top is pointing.
- Step 4 − Decrement top pointer to point to the next available data element.
- Step 5 − Return success. (Return the popped item)

Pop Operation - Algorithm

```
int items[maxsize];
int top;
int Pop (){
  if(top <= 0){
      print("Stack is empty!");
      return -1;   //Error!
  }

  return items[top--]; //Success!
}
```

Top or Peek Operation

Top or Peek operation allows the user to see the element on the top of the stack. The stack is not modified in any manner in this operation.

- **Note:** The difference between this operation and pop operation is that in this operation **the top item will not be removed!**

- Step 1 – Check if the stack is empty.
- Step 2 – If the stack is empty, produce underflow error and exit.
- Step 3 – If the stack is not empty, return the data where top is pointing.

Top or Peek Operation - Algorithm

```
int items[maxsize];
int top;
int Peek (){
  if(top <= 0){
      print("Stack is empty!");
      return -1;   //Error!
  }

  return items[top]; //Success!
}
```

Stack - Application

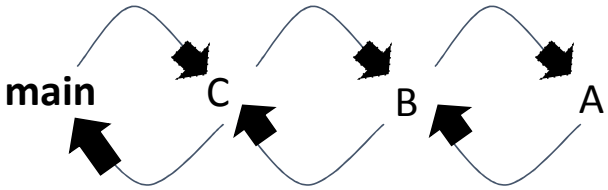
- One of the main usage of stack is in function call.

```
void A() {
 printf("3");
}

void B() {
  A();
  printf("2");
}

void C() {
  B();
  printf("1");
}

void main() {
  C();
}
```



- So, it's better to have a stack:

Memory address of B
Memory address of C
Memory address of main

Stack - Application

Expression conversion is the most important application of stacks. Given an infix expression, it can be converted to both prefix and postfix notations.

- Infix notation ($\boldsymbol{a}$ **operator** $\boldsymbol{b}$): For example, $\boldsymbol{a} + \boldsymbol{b}$ is an infix notation.
- Prefix notation (**operator** $\boldsymbol{a}\ \boldsymbol{b}$): $+\ \boldsymbol{a}\ \boldsymbol{b}$ is the equivalent prefix notation of $\boldsymbol{a} + \boldsymbol{b}$.
- Postfix notation ($\boldsymbol{a}\ \boldsymbol{b}$ **operator**): $\boldsymbol{a}\ \boldsymbol{b}\ +$ is the equivalent postfix notation of $\boldsymbol{a} + \boldsymbol{b}$.

Infix notation is the notation commonly used in arithmetical and logical statements. Consider another infix example, $A + B * C$. The operators $+$ and $*$ still appear between the operands, but there is a problem. Which operands do they work on? Does the $+$ work on A and B or does the $*$ take B and C?

The expression seems **ambiguous**.

What is the solution?

- One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a fully parenthesized expression. This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations.

- Another way is to convert infix expression to prefix or postfix expression. There is no ambiguity in prefix and postfix notations.

Infix Expression	Prefix Expression	Postfix Expression
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 * D	+ * A B * C D	A B * C D * +

Infix to Postfix conversion using Stack

Steps:

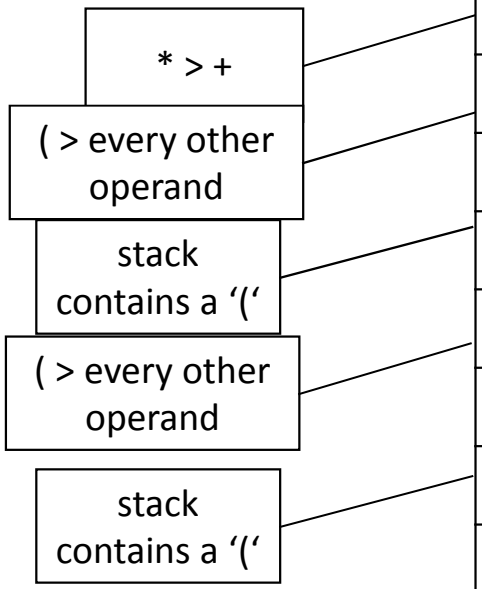
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
    1. If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a ‘(‘), **push** it.
    2. Else, **pop** all the operators from the stack which are greater than or equal to (in precedence) than that of the scanned operator. After doing that, Push the scanned operator to the stack. (If you encounter parenthesis while popping , then stop there and push the scanned operator in the stack) .

Infix to Postfix conversion using Stack (cont.)

4. If the scanned character is a ‘(‘, **push** it onto the stack.
5. If the scanned character is a ‘)’, **pop** the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Example

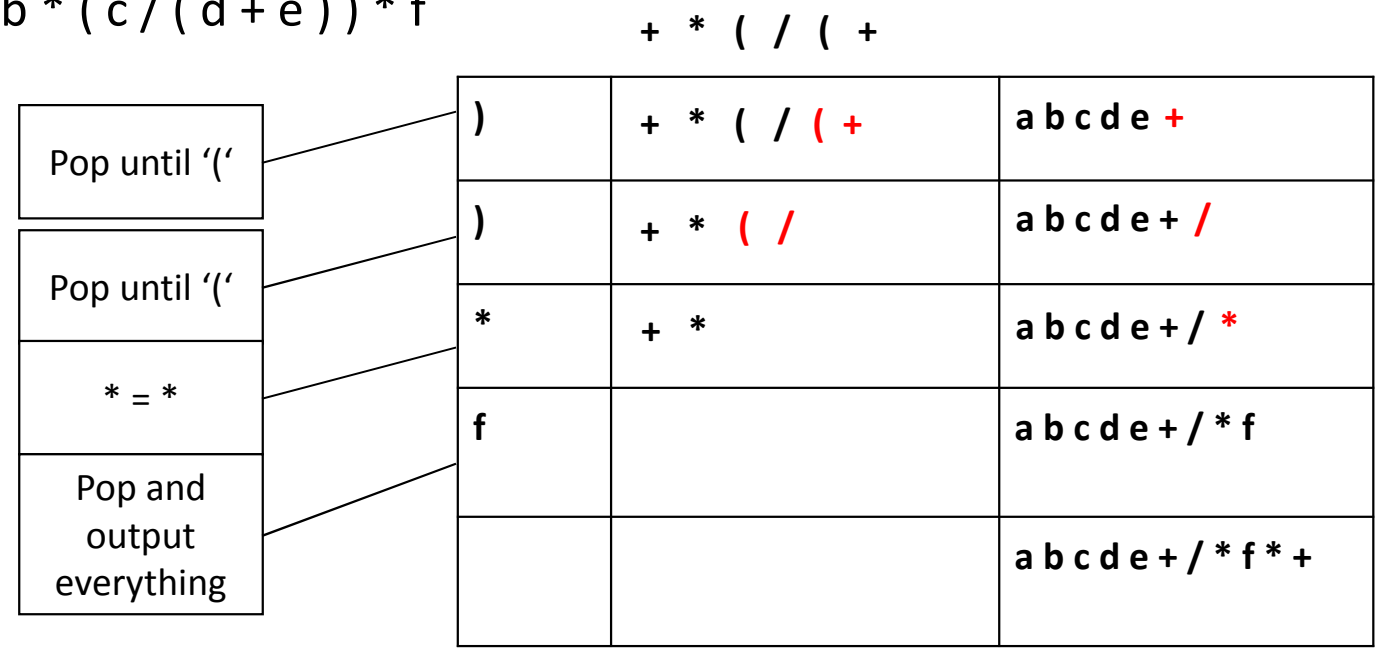
- Expression: a + b * (c / (d + e)) * f



input	stack	output
a		a
+	+	
b	+	a b
*	+ *	
(	+ * (	
c	+ * (	a b c
/	+ * (/	
(	+ * (/ (	
d	+ * (/ (	a b c d
+	+ * (/ (+	
e	+ * (/ (+	a b c d e

Example (cont.)

- Expression: a + b * (c / (d + e)) * f



Stack - Application

Another application of stack is the Evaluation of the Postfix Expressions.

Steps:

1. Create a stack to store operands (or values).
2. Scan the given expression and do the following for every scanned element
    1. If the element is a number, **push** it onto the stack
    2. If the element is an operator, **pop** operands for the operator from stack.

    Evaluate the operator and push the result back to the stack
3. When the expression is ended, the number in the stack is the final answer

Example

- Expression: 5 6 2 + * 12 4 / —

input	stack	output
5	5	
6	5 6	
2	5 6 2	
+	5 8	
*	40	
12	40 12	
4	40 12 4	
/	40 3	
-	37	37

Stack - Application - Checking parentheses balance

Checking for **balanced parentheses** is a very old and classic problem in the field of computer science.

The **valid parentheses problem** involves checking that:

- all the parentheses are matched, i.e., every opening parenthesis has a corresponding closing parenthesis.
- the matched parentheses are in the correct order, i.e., an opening parenthesis should come before the closing parenthesis.

Example:

{ ([ ]) } ( ) [{ }] => The parentheses are valid

{] [ ) => The parentheses are invalid

Checking parentheses balance

Steps:

1. If there is an opening bracket, **push** is onto the stack.
2. If there is a closing bracket, check the top of the stack.
    1. If the top of the stack contains the opening bracket match of the current closing bracket, then pop and move ahead in the string.
    2. If the top of the stack is not the opening bracket match of the current closing bracket, the parentheses are not balanced. In that case, break from the loop.
    3. If the stack is empty, the parentheses are not balanced.
3. After traversing, if the stack is not empty, then the parentheses are not balanced.

Example