Dayananda Sagar University Bangalore

Department of Computer Applications

Data Structures (21CA1203)

Dr. Rekha R. Nair

MODULE - 3

Stacks and Queues

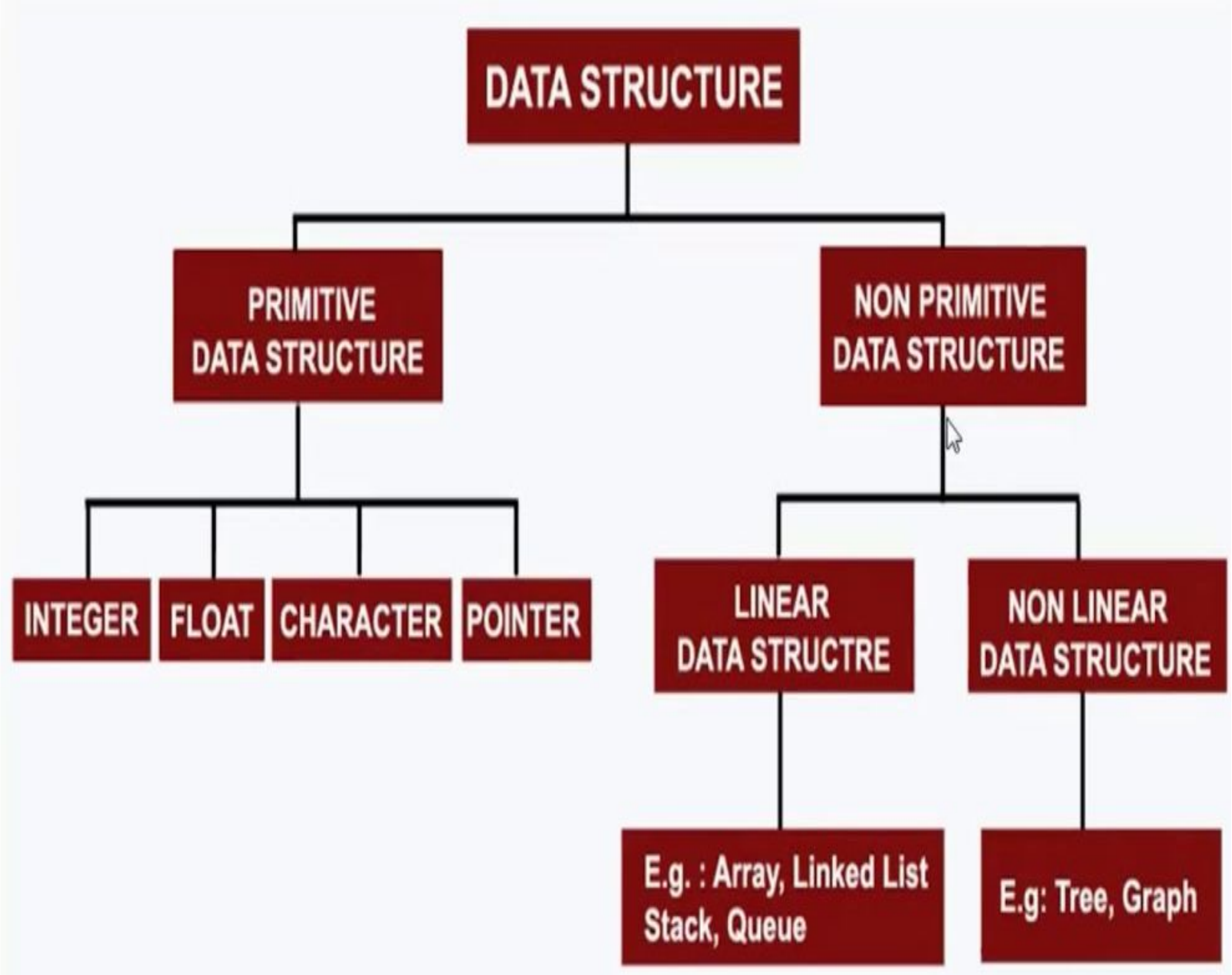
Basic Stack Operations, Representation of a Stack using Arrays, Stack Applications: Reversing list, Factorial Calculation, In-fix- to postfix Transformation, Evaluating

Arithmetic Expressions.

Queues: Basic Queues Operations, Representation of a Queue using array,

Implementation of Queue Operations using Stack, Applications of Queues-Round robin Algorithm, Enqueue, Dequeue, Circular Queues, Priority Queues.

Data Structure Types



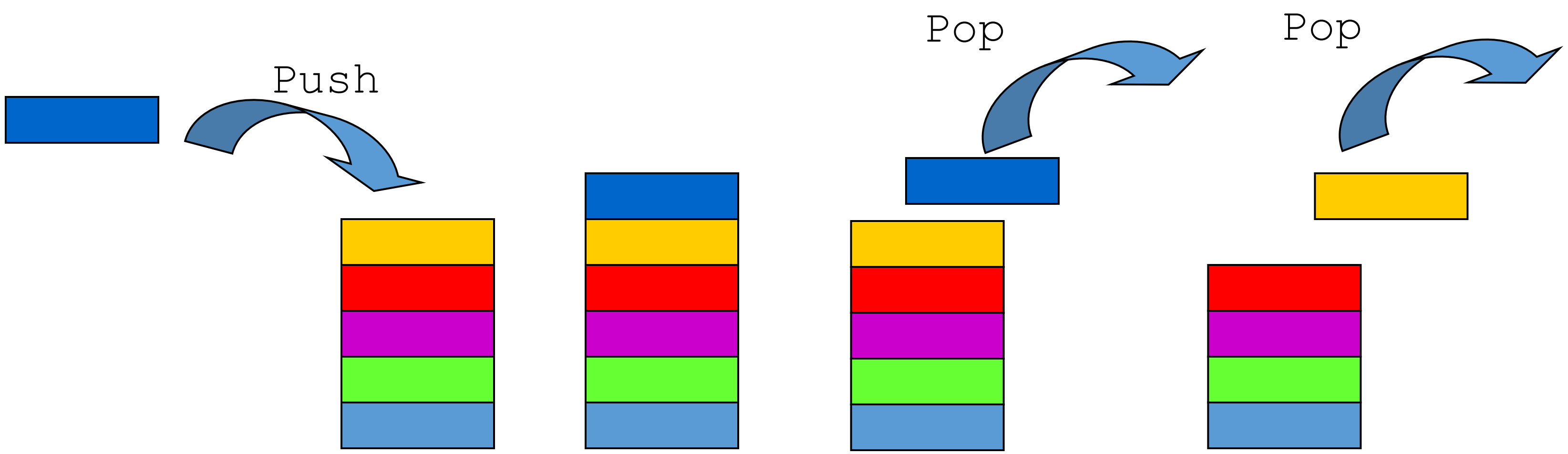
STACK

- A real-world stack, for example – a deck of cards or a pile of plates, etc.

- we can place or remove a card or plate from the top of the stack only.
- Linear Data structure

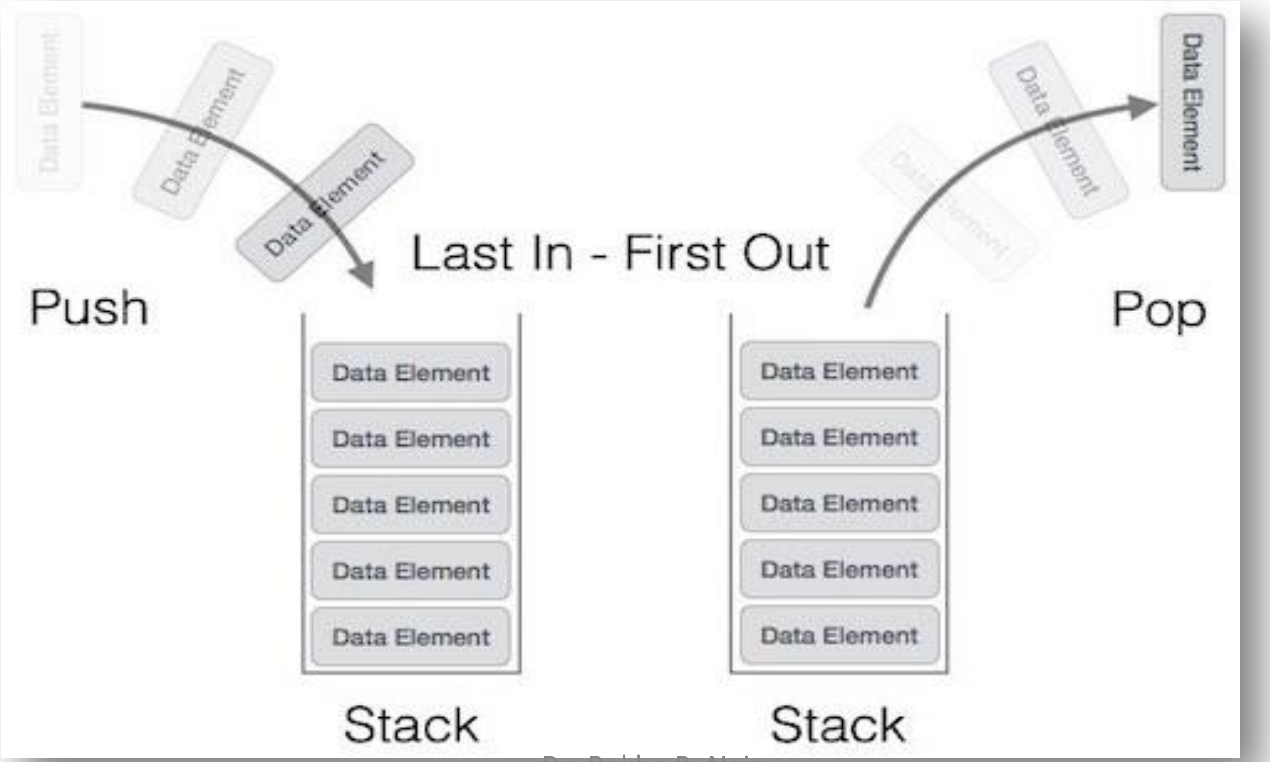
STACK

- A list for which Insert and Delete are allowed only at one end of the list (the *top*)
- Last In First Out (LIFO)



STACK

- A **stack** can be defined as a container in which insertion and deletion can be done from the one end known as the **top of the stack**.
- Can be implemented by means of Array, Structure, Pointers and Linked List.
- Stack can either be a fixed size or dynamic.



STACK – key points

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

What is this good for?

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Recursive function calls

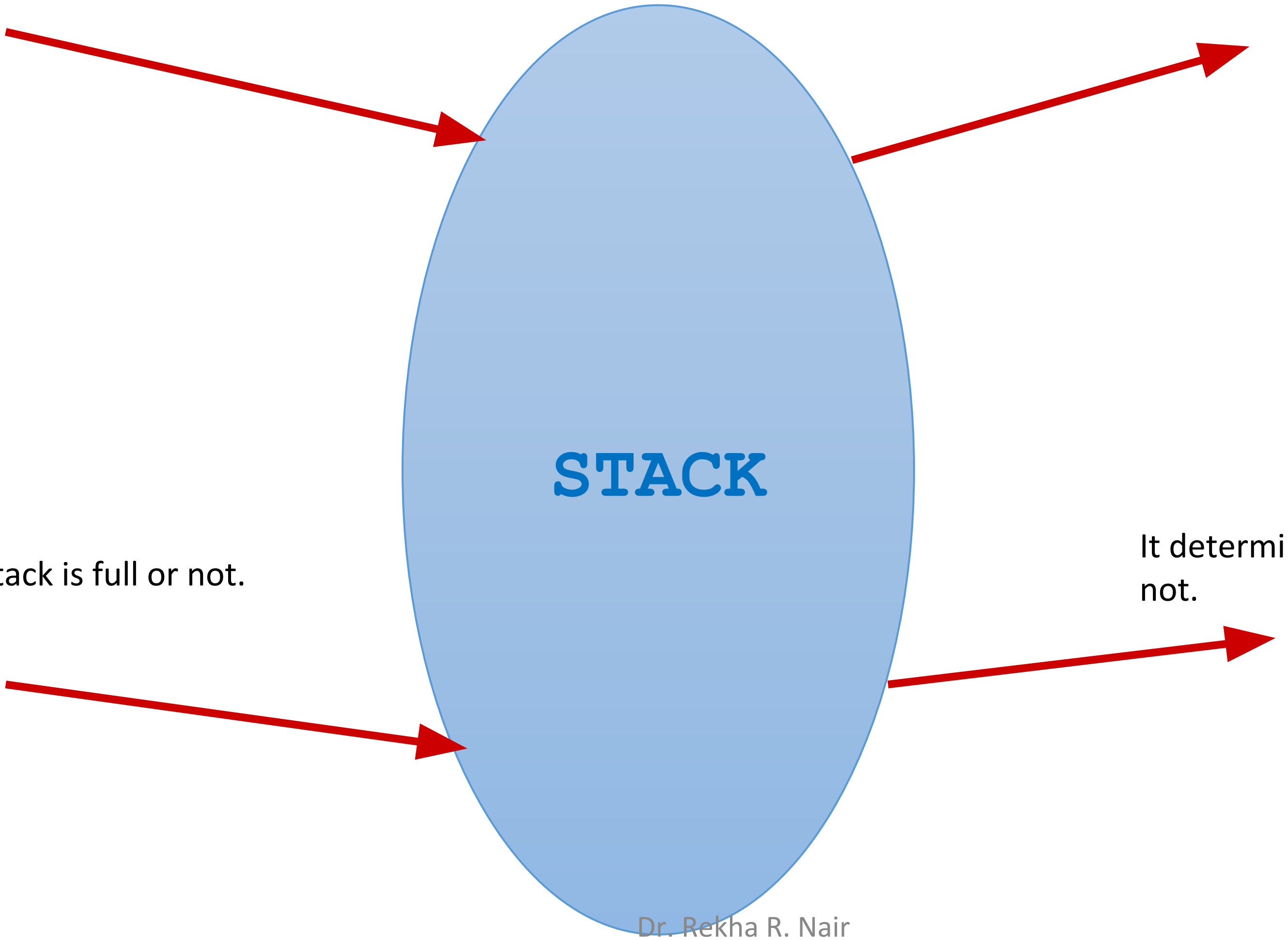
Stack Operations

When we insert an element in a stack then the operation is known as a push. If the stack is full then the ***overflow*** condition occurs

When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an ***underflow*** state.

push

pop



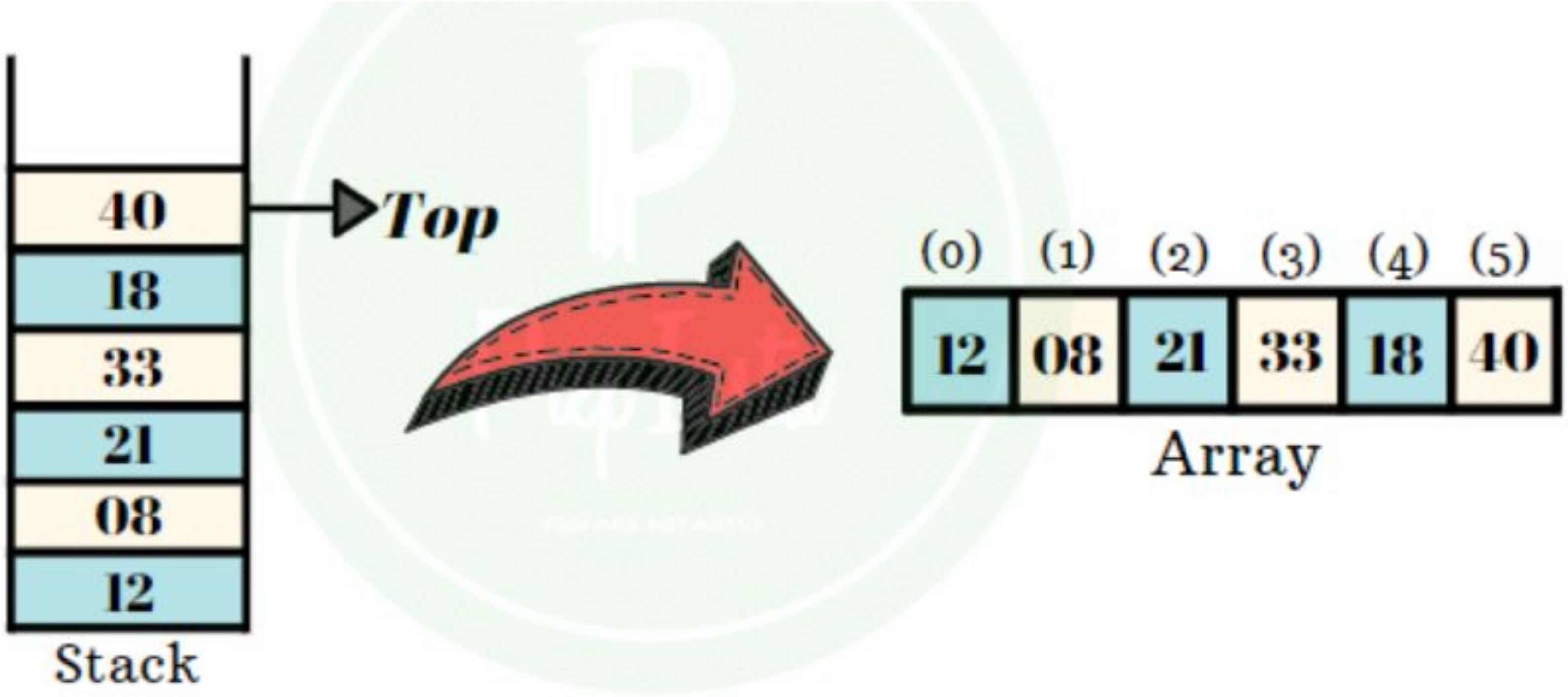
It determines whether the stack is full or not.

isfull

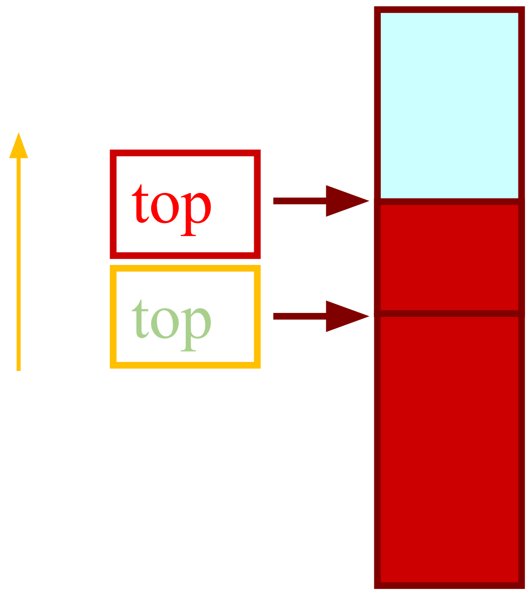
It determines whether the stack is empty or not.

isempty

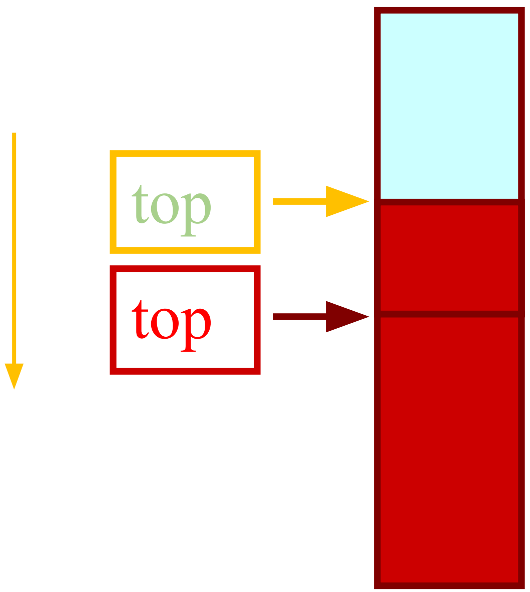
STACK Array Representation



STACK Array Representation



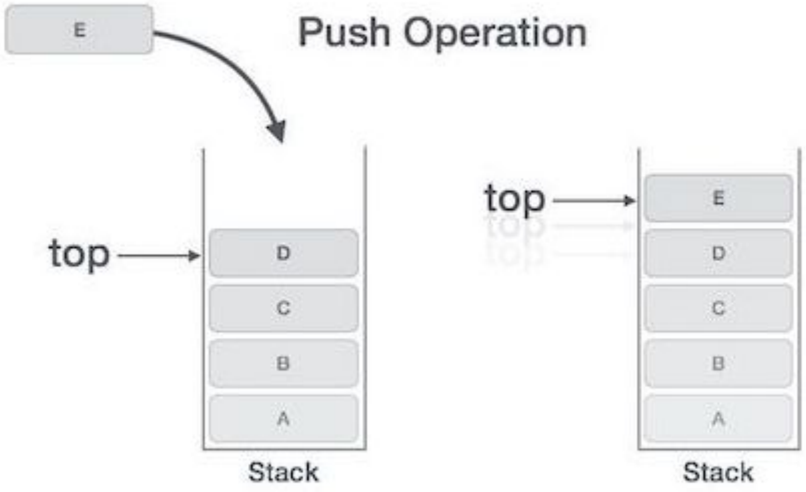
PUSH



POP

STACK – PUSH operation

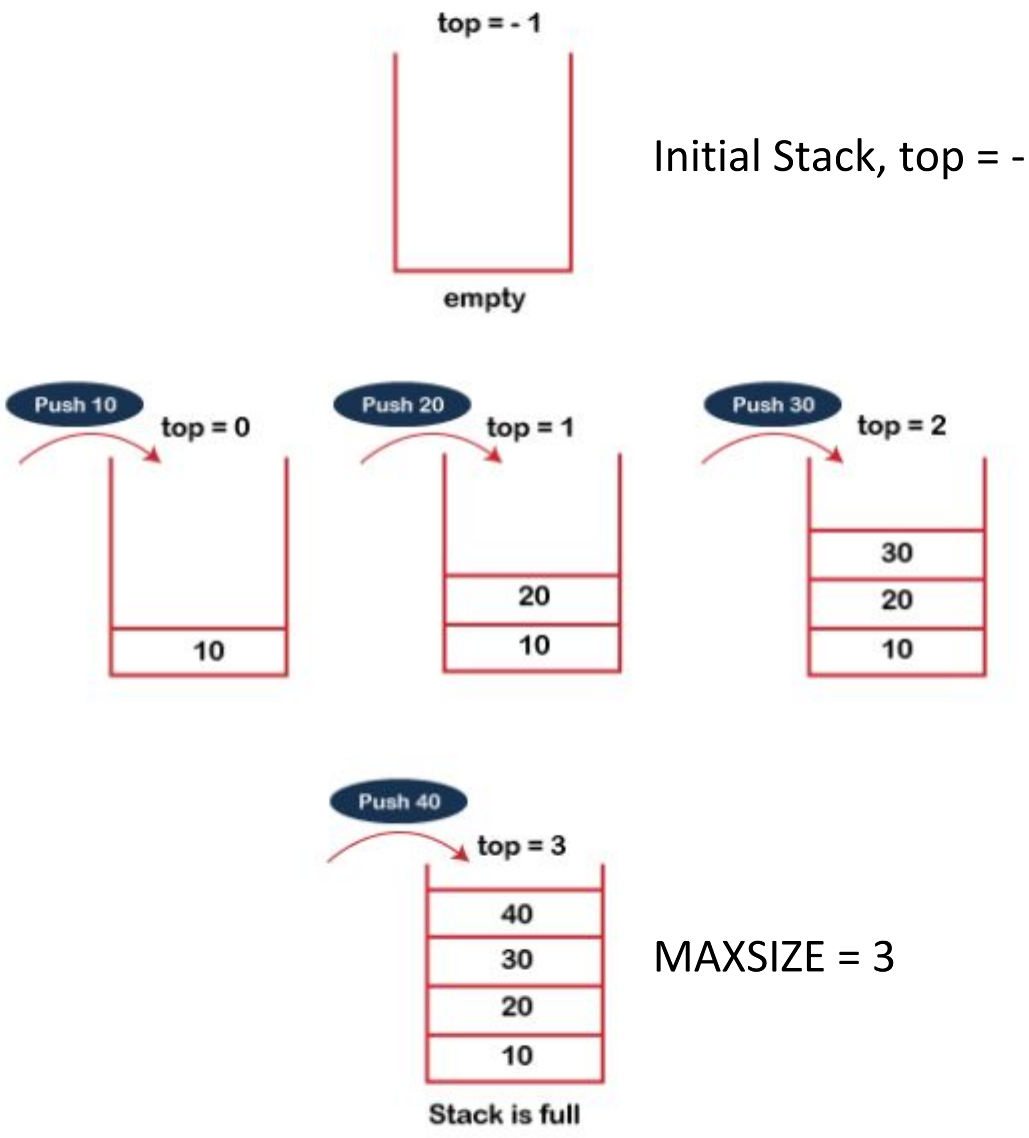
- The process of putting a new data element onto stack is known as a Push Operation.
- When stack is empty or initialised *top = -1*
- For each insertion the value of top is incremented by 1.



STACK – PUSH operation

- Steps in PUSH operation

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



Initial Stack, top = -1

MAXSIZE = 3

STACK – PUSH operation

- Algorithm

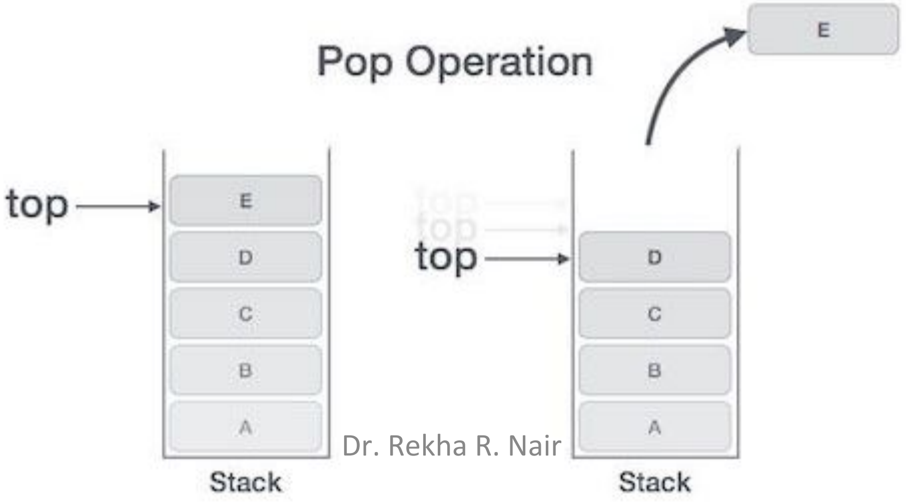
```
begin procedure push: stack, data
    if stack is full
        return null
    endif
    top ← top + 1
    stack[top] ← data
end procedure
```

- Program function

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

STACK – POP operation

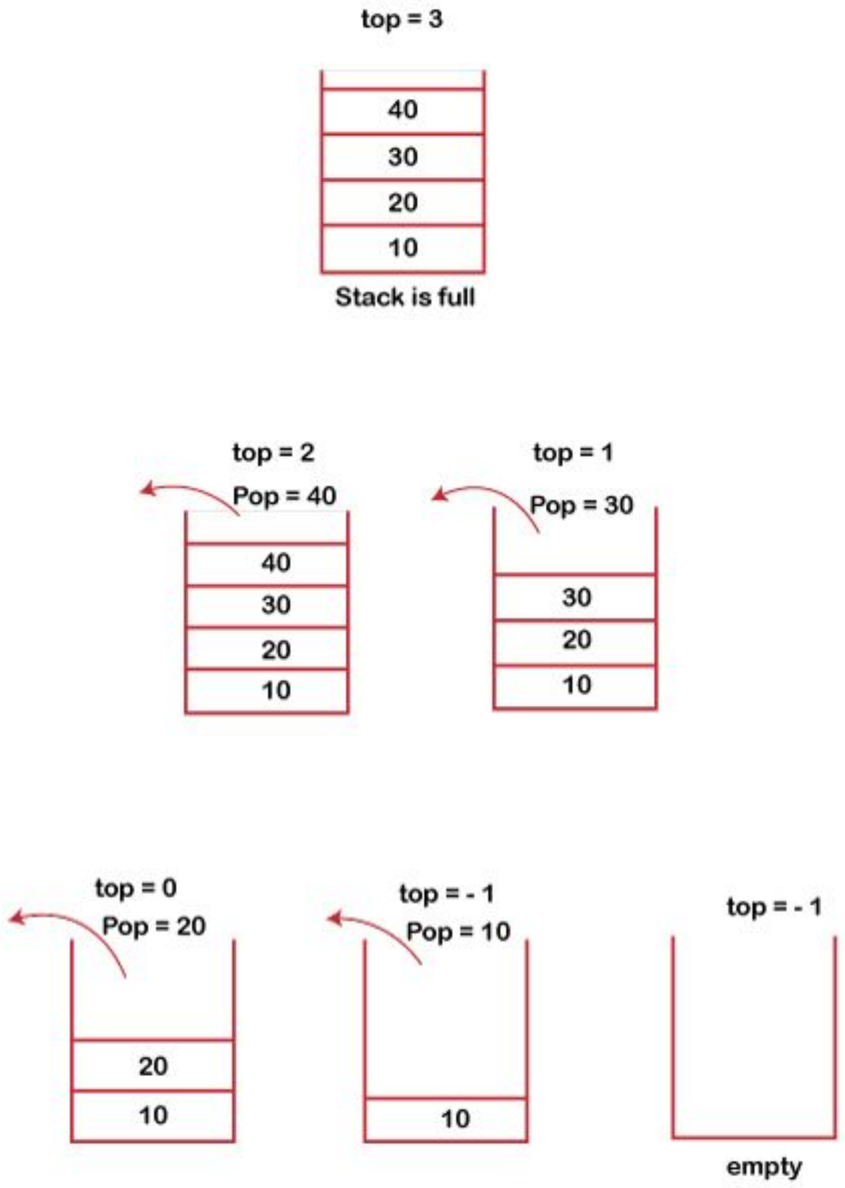
- Accessing the content while removing it from the stack, is known as a Pop Operation.
- The data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.
- For each deletion the value of *top is decremented by 1*.



STACK – POP operation

- Steps in POP operation

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



STACK – POP operation

- Algorithm

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

- Program function

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

STACK – isfull operation

- Algorithm

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

- Program function

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

STACK – isempty() operation

- Algorithm

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

- Program function

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

STACK – Display elements

```
Void display_stack()
{
    for (int i = 0; i < = top; i++)
        printf("%d",A[i]);
}
```

```c
#include <stdio.h>

int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty() {

   if(top == -1)
      return 1;
   else
      return 0;
}

int isfull() {

   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}
```

```c
int peek() {
   return stack[top];
}

int pop() {
   int data;

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

```c
int push(int data) {

   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Output

```
Element at top of the stack: 15
Elements:
15
12
1
9
5
3
Stack full: false
Stack empty: true
```

```c
int main() {
   // push items on to the stack
   push(3);
   push(5);
   push(9);
   push(1);
   push(12);
   push(15);

   printf("Element at top of the stack: %d\n" ,peek());
   printf("Elements: \n");

   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }

   printf("Stack full: %s\n" , isfull()?"true":"false");
   printf("Stack empty: %s\n" , isempty()?"true":"false");

   return 0;
}
```

Applications of stack

- String reversal
- UNDO/REDO
- Recursion
- DFS(Depth First Search)
- Backtracking
- Expression conversion
    - Infix to prefix
    - Infix to postfix
    - Prefix to infix
    - Prefix to postfix
    - Postfix to infix
- Memory management
- Evaluation of expression
- Other applications-Parenthesis matching
- -palindrome checking

```c
#include<stdio.h>
int stk[100]; // stack
int size = 100; // size of stack
int ptr = -1; // store the index of top element of the stack
// push x to stack
void push(int x) {
if (ptr == size - 1) {
printf("OverFlow \n");
}
else {
++ptr;
stk[ptr] = x;
}
}
// return top element of the stack
int top() {
if (ptr == -1) {
printf("UnderFlow \n");
return -1;
}
else {
return stk[ptr];
}
}
```

```c
// remove top element from the stack
void pop() {
if (ptr == -1) {
printf("UnderFlow \n");
}
else {
--ptr;
}
}
// check if stack is empty or not
int isempty() {
if (ptr == -1)
return 1;
else
return 0;
}
```

Arithmetic Expressions

- Arithmetic expressions have:
  - operands (variables or numeric constants).
  - Operators
    - Binary : +, -, *, / ,%
    - Unary: -
  - Priority convention:

    ***, /, %** have medium priority

    +, - have lowest priority

Operator Precedence

Operators	
( )	HIGH
+ (unary) , - (unary) , ! (NOT)	
*, /, %	
+ (addition), - (subtraction)	
<, <=, >, >=	
==, !=	
&&	
||	LOW

Infix, Prefix, Postfix

- Example: arithmetic expression a + b consists of *operands a, b* and *operator +*.

- Infix notation
  - Is format where operator is specified in **between** the two operands. **a+b**

- Prefix (Polish) notation
  - Is format where operator is specified **before** the two operands. **+ a b**

- Postfix (Reverse-Polish) notation
  - Is format where operator is specified **after** the two operands. Postfix notation is also called RPN or Reverse Polish Notation. **a b +**

WHY ??

- Why to use PREFIX and POSTFIX notations when we have simple INFIX notation?
- INFIX notations are not as simple as they seem specially while evaluating them.
- To evaluate an infix expression we need to consider Operators' Priority and Associative property
- E.g. expression 3+5*4 evaluate to 32 i.e. (3+5)*4 or to 23 i.e. 3+(5*4).
- To solve this problem Precedence or Priority of the operators were defined. Operator precedence governs evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

Postfix Notation

Expressions are converted into Postfix notation before compiler can accept and process them.

X = A / B – C + D * E – A * C

Infix => A / B – C + D * E – A * C (Operators come in-between operands)

Postfix => A B / C – D E * + A C * - (Operators come after operands)

OPERATOR	PRECEDENCE	VALUE	Association
Exponentiation ($ or ↑ or ^)	Highest	3	R → L
*, /	Next highest	2	L → R
+, -	Lowest	1	L → R

3. Conversion of Expression

1. Infix to Postfix	e.g.	A+B -> AB+
2. Infix to Prefix	e.g.	A+B -> +AB
3. Postfix to Infix	e.g.	AB+ -> A+B
4. Postfix to Prefix	e.g.	AB+ -> +AB
5. Prefix to Infix	e.g.	+AB -> A+B
6. Prefix to Postfix	e.g.	+AB -> AB+

Converting arithmetic expressions

Example: Conversion from infix arithmetic expression to prefix and postfix.

Infix Notation	Prefix Notation	Postfix Notation
A + B * C	+A * B C	A B C * +
(A+B) * C	* + A B C	A B + C *
A – B + C	+ – A B C	A B – C +
A – (B+C)	– A + B C	A B C + –

Infix to Postfix

- This application converts the infix notation of a given arithmetic expression into postfix notation.

- In an <u>infix notation</u> the operator is placed in between the operands : a+b

- In a <u>postfix notation</u> the operator is placed immediately after the operand: ab+

Infix to Postfix Conversion

B ⟶ Bracket - () or { }

O ⟶ Order or Power - 2^5, 3^7, $\sqrt{2}$

D ⟶ Division (÷)

M ⟶ Multiplication (×)

A ⟶ Addition (+)

S ⟶ Subtraction (−)

<u>The Algorithm</u>

- What are possible items in an input Infix expression
- Read an item from input infix expression
- If item is an operand append it to postfix string
- If item is "(" push it on the stack
- If the item is an operator
  - If the operator has higher precedence than the one already on top of the stack then push it onto the operator stack
  - If the operator has lower precedence than the one already on top of the stack then
    - pop the operator on top of the operator stack and append it to postfix string, and
    - push lower precedence operator onto the stack
- If item is ")" pop all operators from top of the stack one-by-one, until a "(" is encountered on stack and removed
- If end of infix string pop the stack one-by-one and append to postfix string

Example: Infix to Postfix

- Example 1: Infix notation A - B

STEPS	INFIX	OPERATOR STACK	POSTFIX
1	A	empty	A
2	-	-	A
3	B	-	AB
4		empty	**AB-**

Example: Infix to Postfix

- Example 2: Infix notation A + B - C

A	B	+	C	-

Postfix string

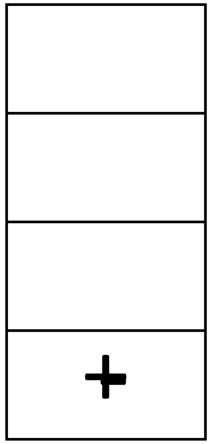
Step 1 : A, Operand, place it in postfix string

Step 2 : +, Operator, stack is empty, push to stack

Step 3 : B, Operand, place it in postfix string

Step 4 : -, Operator, - has the same precedence as +,
so Pop + and place in postfix string and
Push – to stack

Step 5 : C, Operand, place it in postfix string

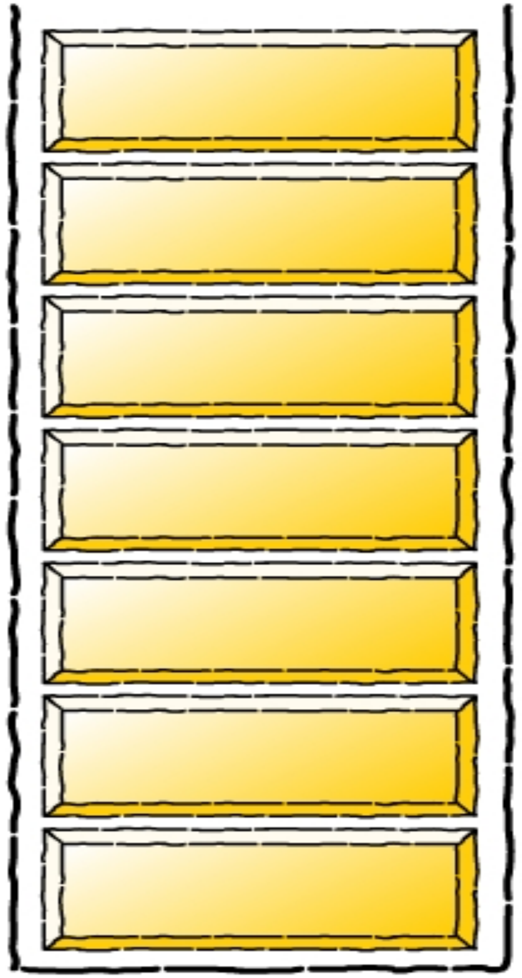


Stack

Infix exp is over, now pop all elements from stack

Infix to postfix conversion

Stack



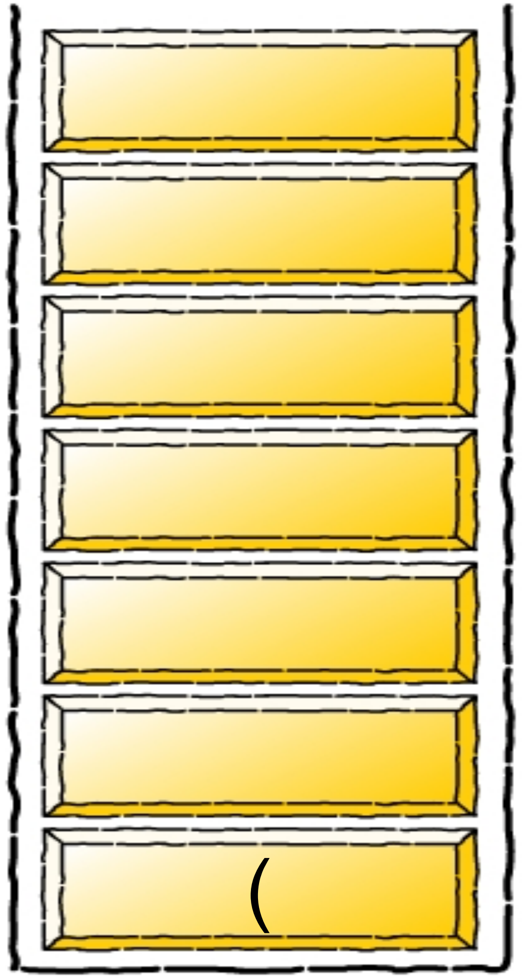
Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

Infix to postfix conversion

Stack



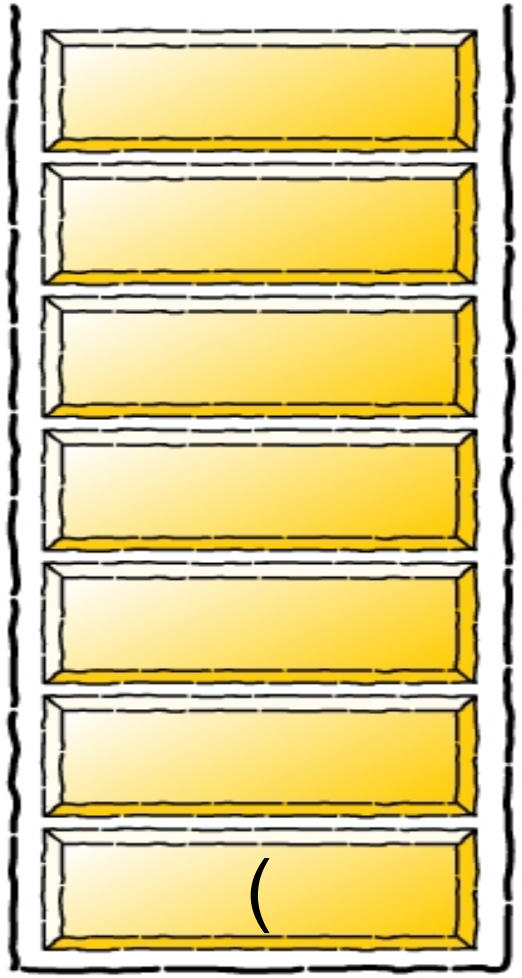
Infix Expression

a + b - c) * d – (e + f)

Postfix Expression

Infix to postfix conversion

Stack



Infix Expression

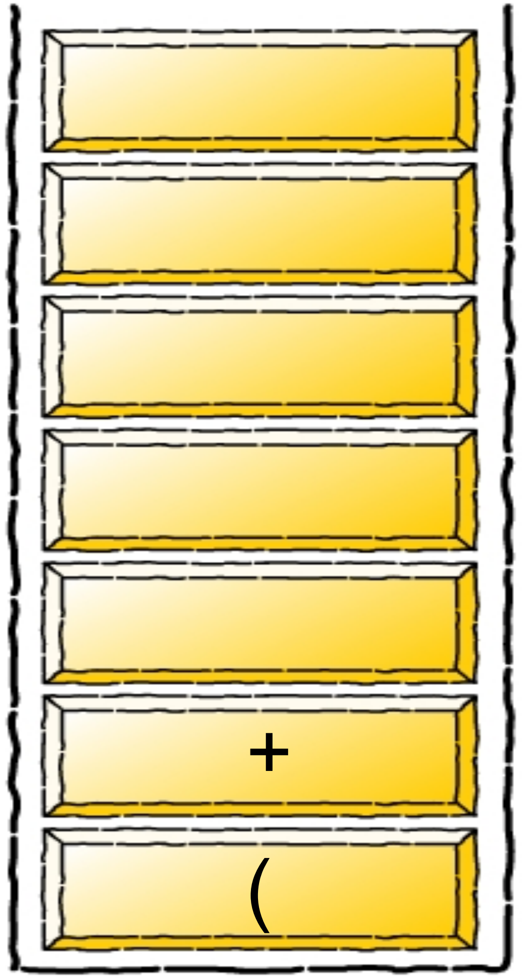
+ b - c) * d – (e + f)

Postfix Expression

a

Infix to postfix conversion

Stack



Infix Expression

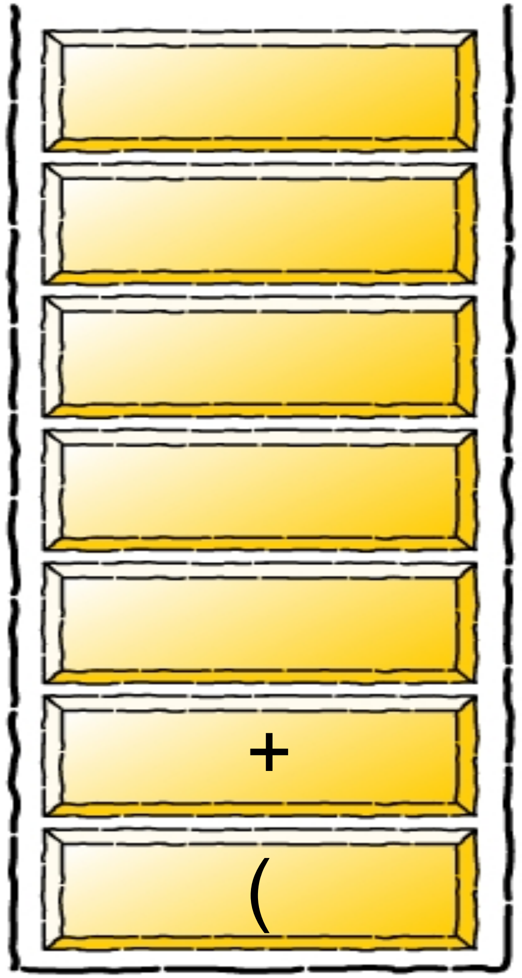
b - c) * d – (e + f)

Postfix Expression

a

Infix to postfix conversion

Stack



Infix Expression

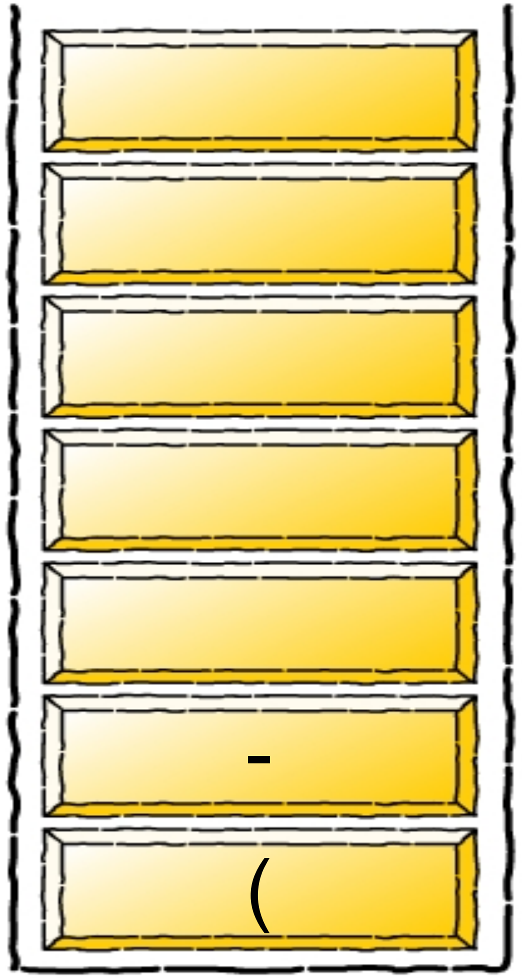
- c) * d – (e + f)

Postfix Expression

a b

Infix to postfix conversion

Stack



Infix Expression

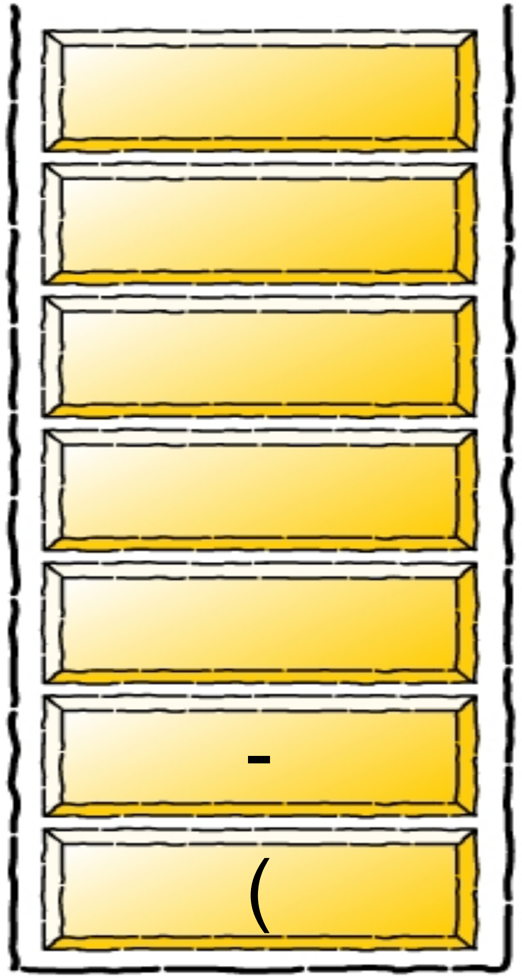
c) * d – (e + f)

Postfix Expression

a b +

Infix to postfix conversion

Stack



Infix Expression

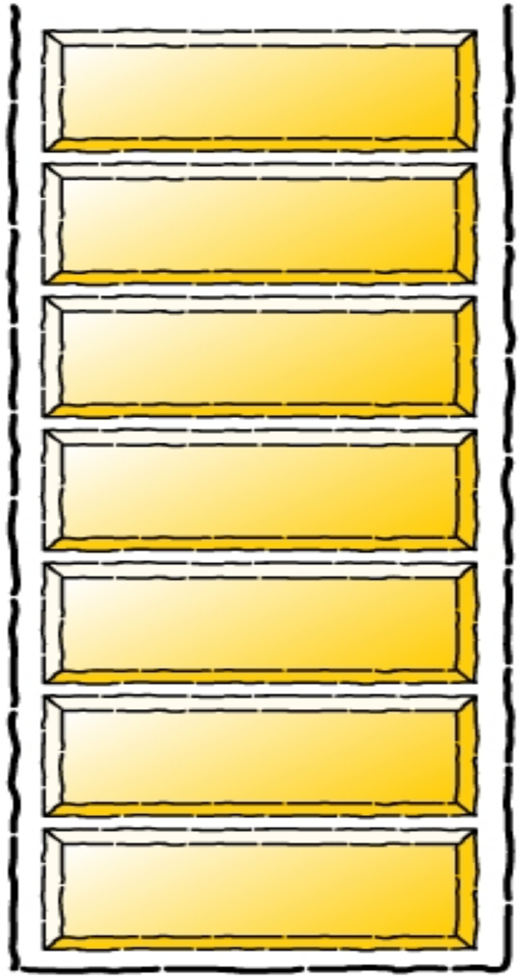
) * d – (e + f)

Postfix Expression

a b + c

Infix to postfix conversion

Stack



Infix Expression

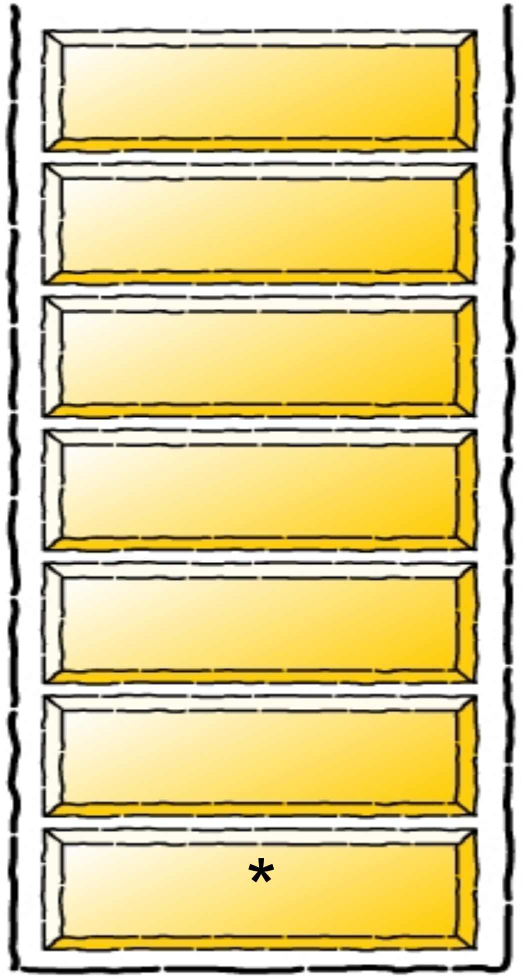
* d – (e + f)

Postfix Expression

a b + c -

Infix to postfix conversion

Stack



Infix Expression

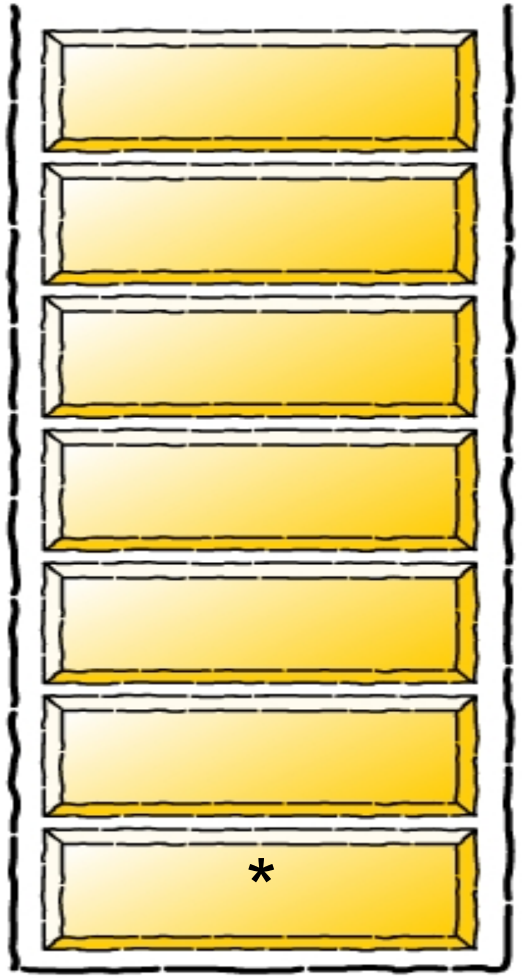
d – (e + f)

Postfix Expression

a b + c -

Infix to postfix conversion

Stack



Infix Expression

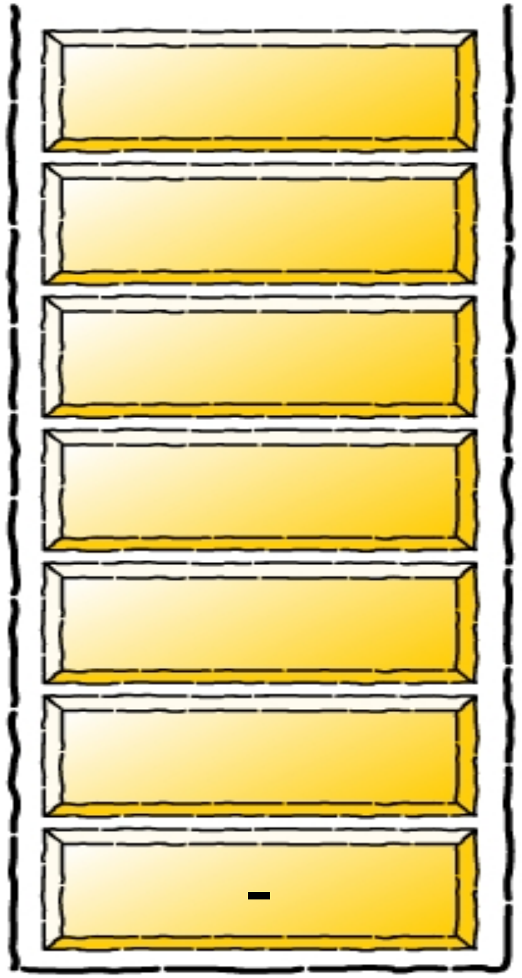
– (e + f)

Postfix Expression

a b + c - d

Infix to postfix conversion

Stack



Infix Expression

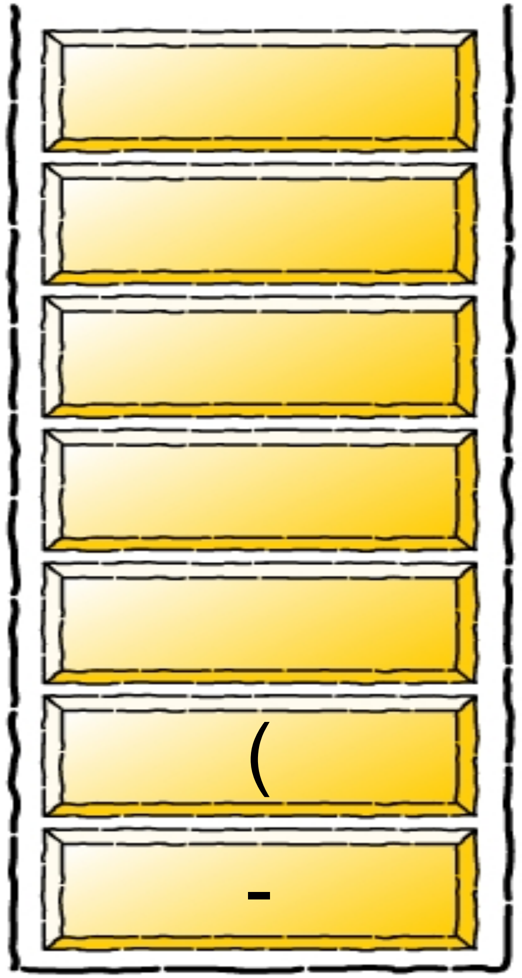
(e + f)

Postfix Expression

a b + c – d *

Infix to postfix conversion

Stack



Infix Expression

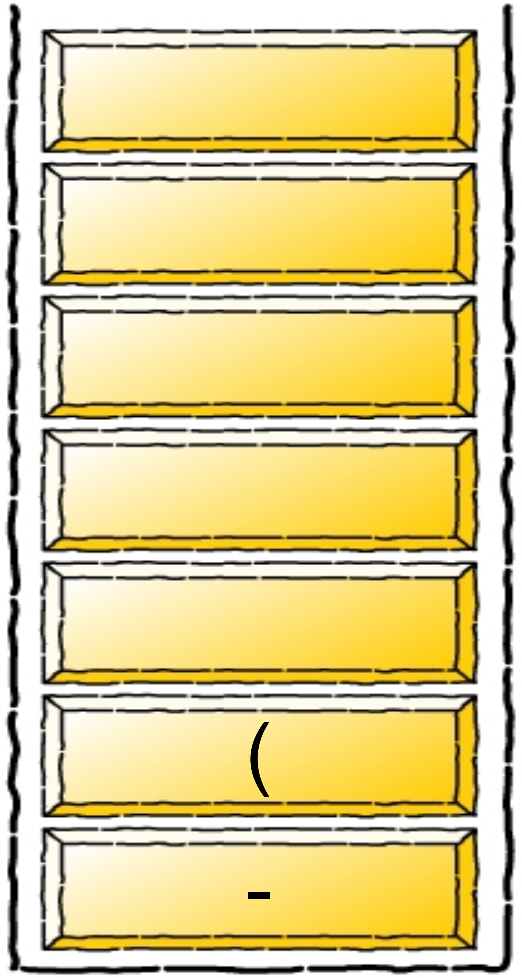
e + f)

Postfix Expression

a b + c – d *

Infix to postfix conversion

Stack



Infix Expression

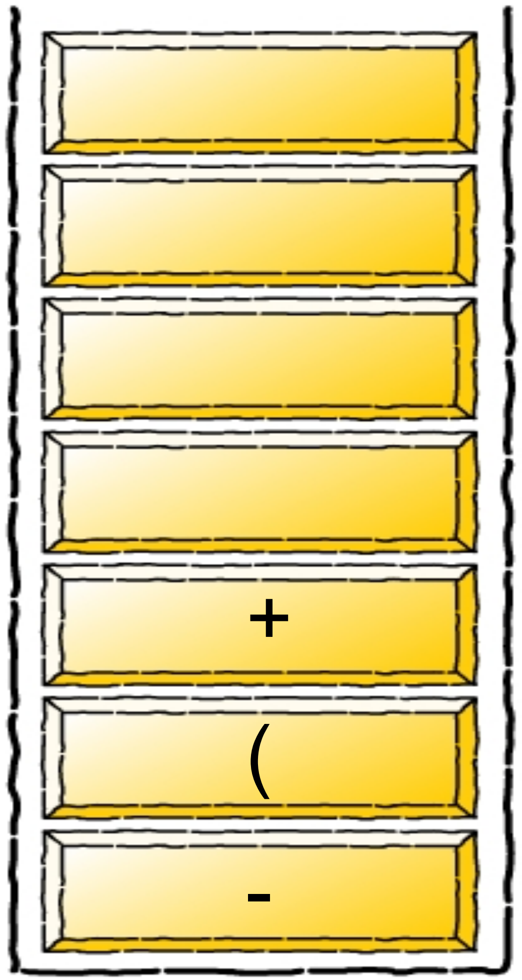
+ f)

Postfix Expression

a b + c – d * e

Infix to postfix conversion

Stack



Infix Expression

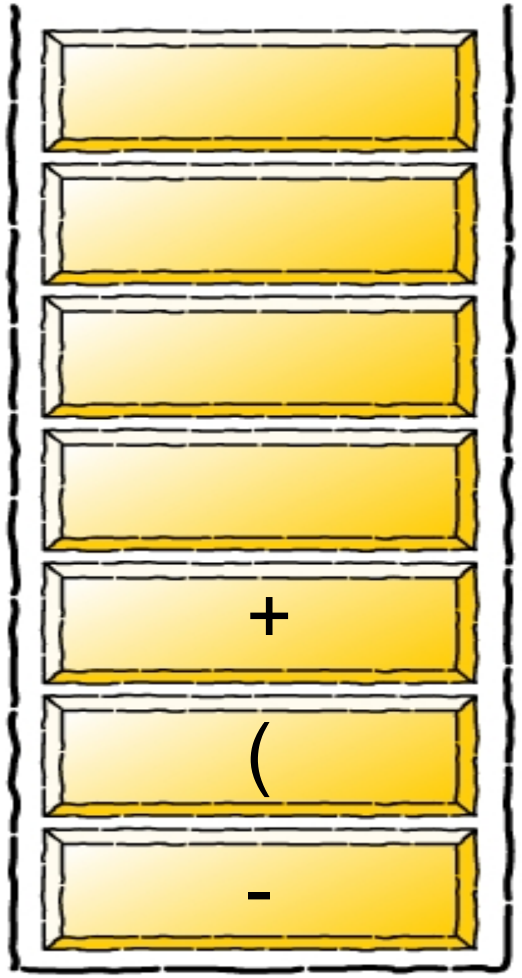
f)

Postfix Expression

a b + c – d * e

Infix to postfix conversion

Stack



Infix Expression

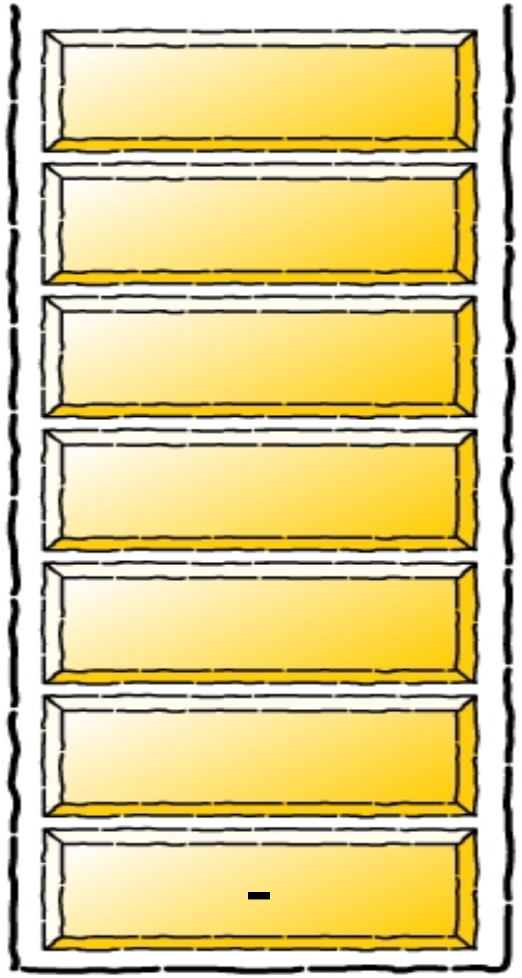
)

Postfix Expression

a b + c – d * e f

Infix to postfix conversion

Stack



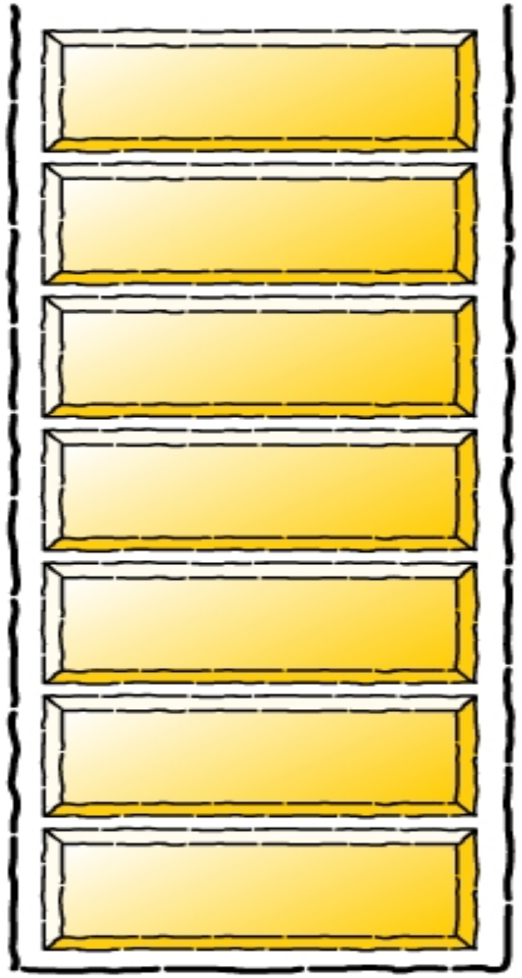
Infix Expression

Postfix Expression

a b + c – d * e f +

Infix to postfix conversion

Stack



Infix Expression

Postfix Expression

a b + c – d * e f + -

[Infix to Postfix using stack (Step by Step Animation) - YouTube]

Example of Infix to Postfix

- Example 3: Infix notation A + B * C

STEPS	INFIX	OPERATOR STACK	POSTFIX
1	A	empty	A
2	+	+	A
3	B	+	AB
4	*	+ *	AB
5	C	+ *	ABC
6		+	ABC *
7		empty	**ABC *+**

Example Infix to Postfix

- Example 4
- Convert this infix notation to postfix notation

  a + c – r / b * r

Solution

a + c – r / b * r

STEPS	INFIX	OPERATOR STACK	POSTFIX
1	a	empty	a
2	+	+	a
3	c	+	ac
4	-	-	ac +
5	r	-	ac + r
6	/	- /	ac + r
7	b	- /	ac + rb
8	*	- *	ac + rb /
9	r	- *	ac + rb / r
10		-	ac + rb / r *
11		empty	**ac + rb / r * -**

Example

Expression (Q): (A + (B * C – (D / E ^ F)))

Q	STACK	Output Postfix String P
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
)	(+	ABC*DEF^/-
)		ABC*DEF^/-+

Here, the red parenthesis defines step number 1 of the algorithm where we need to Push "(" onto STACK, and add ")" to the end of Q.

Resultant postfix expression

Convert Infix to Postfix Expression

- (A+(B-C)*D)
- A+(B*C-(D/E^F)*G)*H
- X^Y/(5*Z)+2

Answer

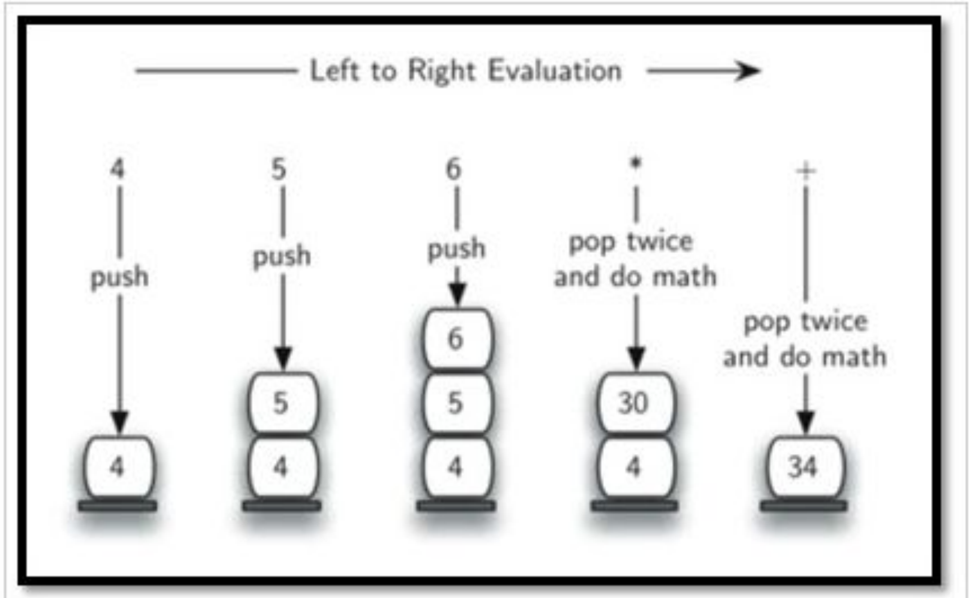
- A B C - D * +
- A B C * D E F ^ / G * - H * +
- XY^5Z*/2+

Evaluation of Postfix expression

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Evaluation of Postfix expression

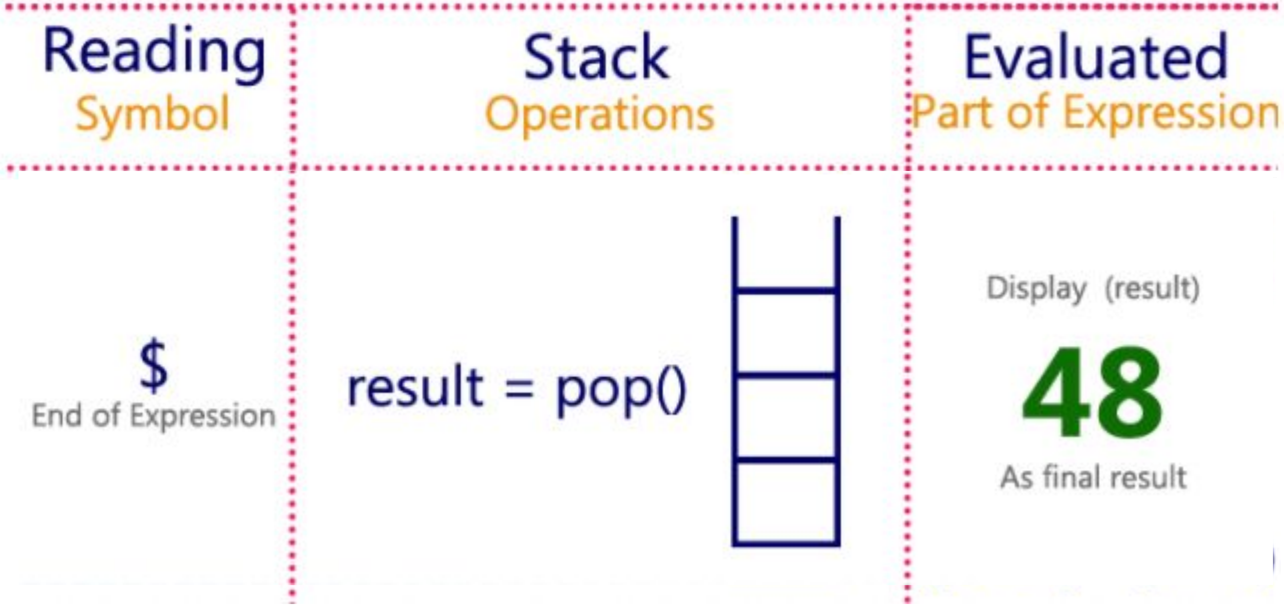
Expression: 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

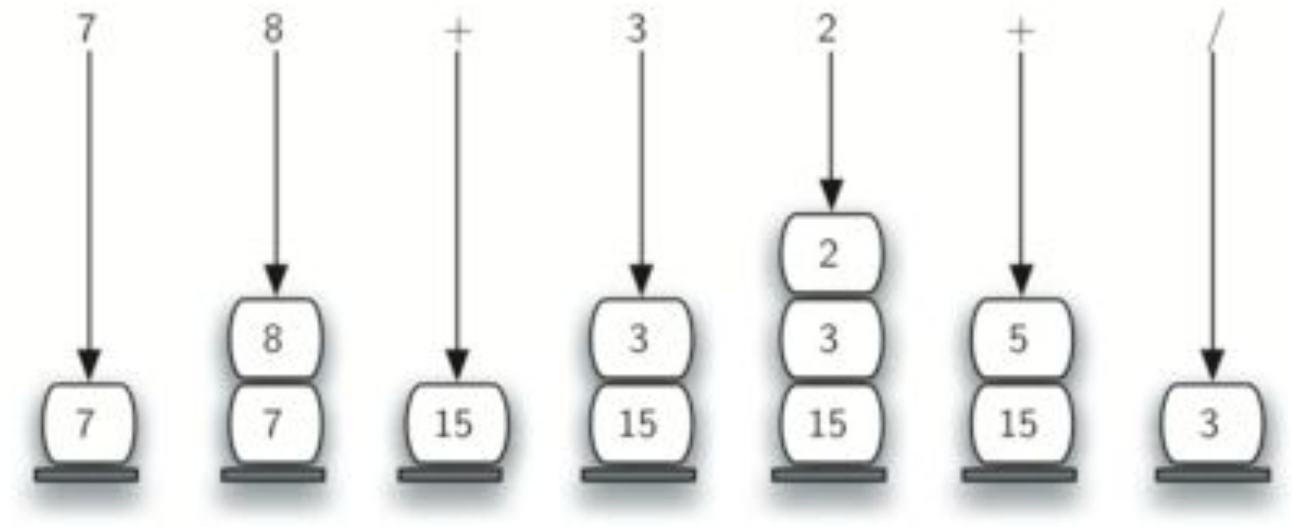
Evaluation of Postfix expression

Postfix Expression 5 3 + 8 2 - *



Evaluation of Postfix expression

- Evaluate the postfix exp 7 8 + 3 2 + /.

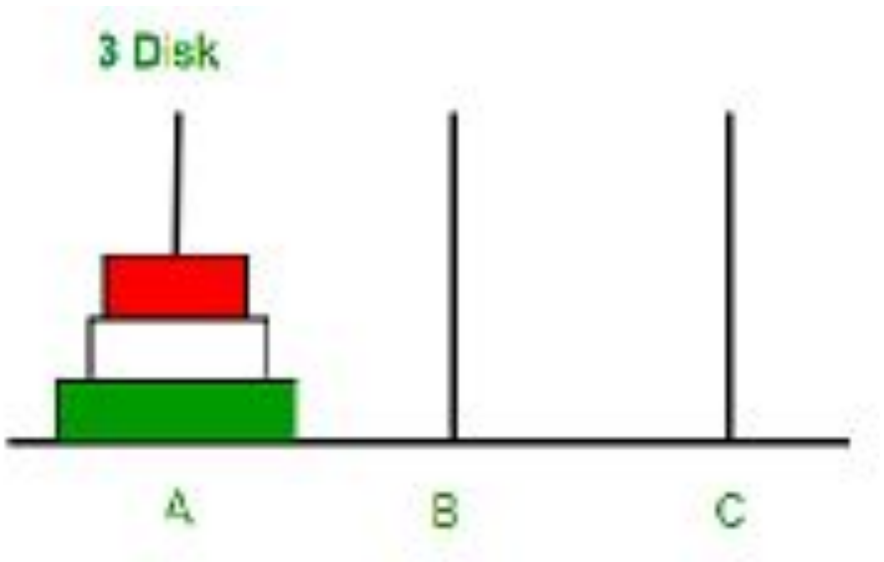


Exercise

1) 2 3 1 * + 9 –
2) 100 200 + 2 / 5 * 7 +

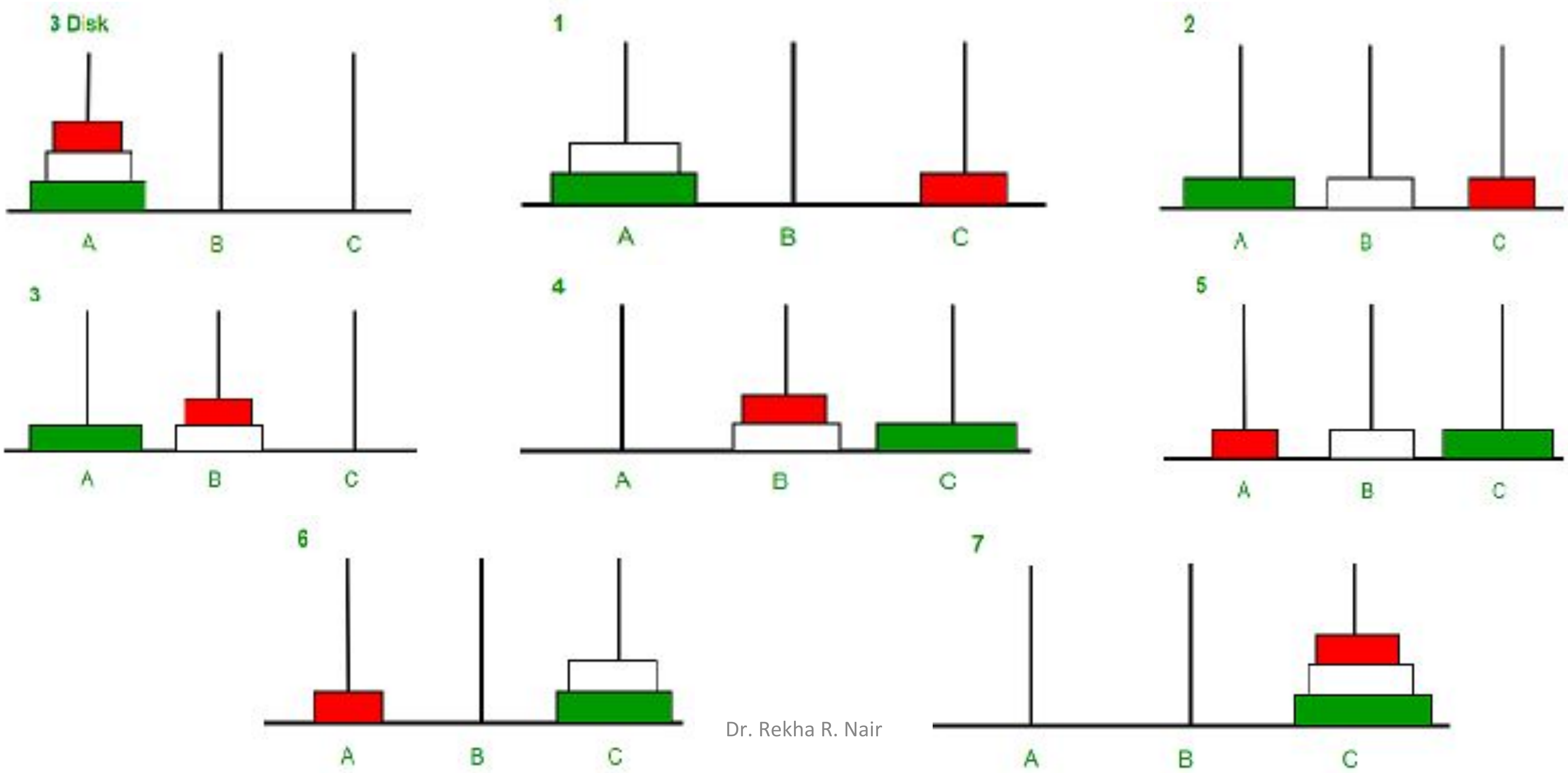
1) -4
2) 757

Tower of Hanoi

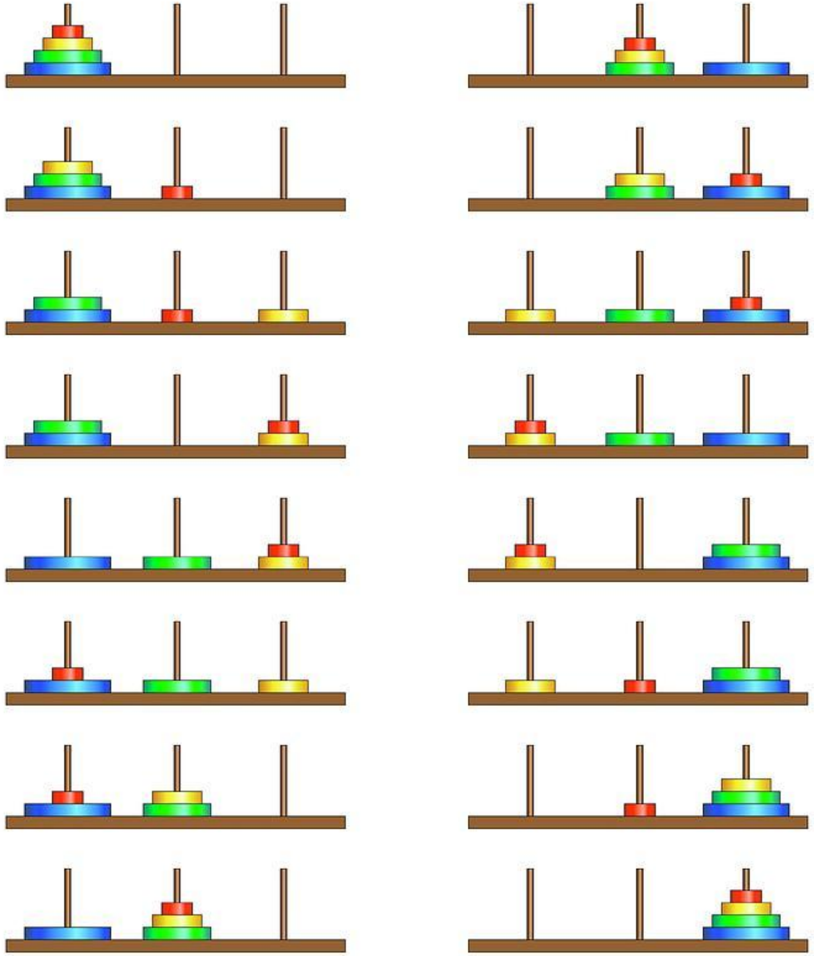


- Tower of Hanoi is a mathematical puzzle
    - Consist of three rods and n disks.
- Rules
    - Only one disk can be moved at a time.
    - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
    - No disk may be placed on top of a smaller disk.
- Objective –
    - Move all the disks from one pole (say ‘source pole’) to another pole (say ‘destination pole’) with the help of the third pole (say auxiliary pole).

Tower of Hanoi – 3 disks



Tower of Hanoi – 4 disks



- 3 disks – steps = 7
- 4 disks – steps = 15
- N disks – steps = $2^N - 1$

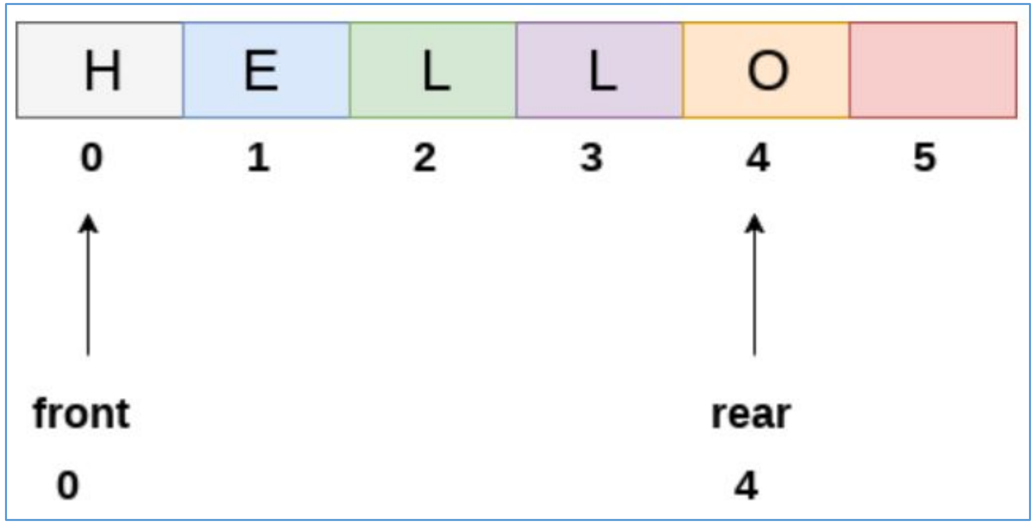
QUEUES

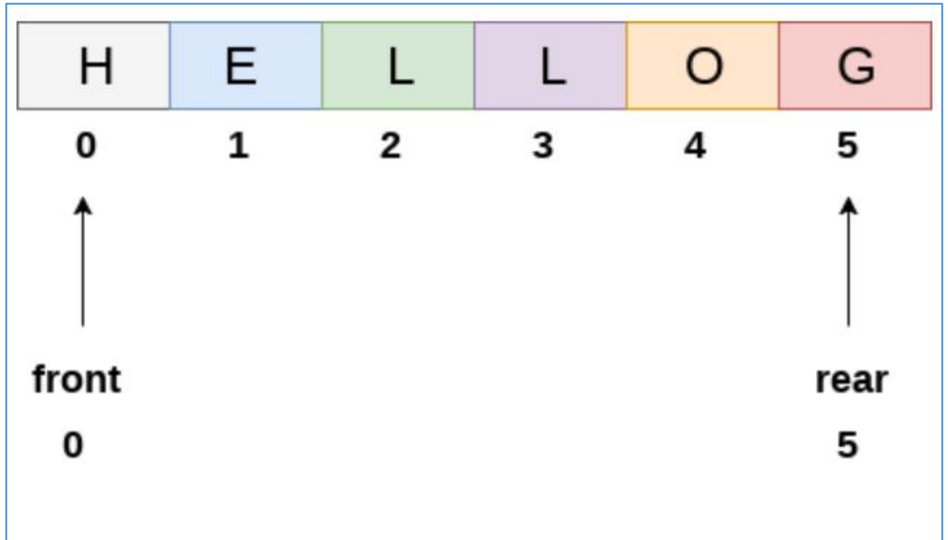
- Queue is an linear data structure, somewhat similar to Stacks.
- Unlike stacks, a queue is open at both its ends.
    - One end is always used to insert data (enqueue)
    - Other end is used to remove data (dequeue).
- First-In-First-Out (FIFO) methodology, i.e., the data item stored first will be accessed first.

QUEUE Representation

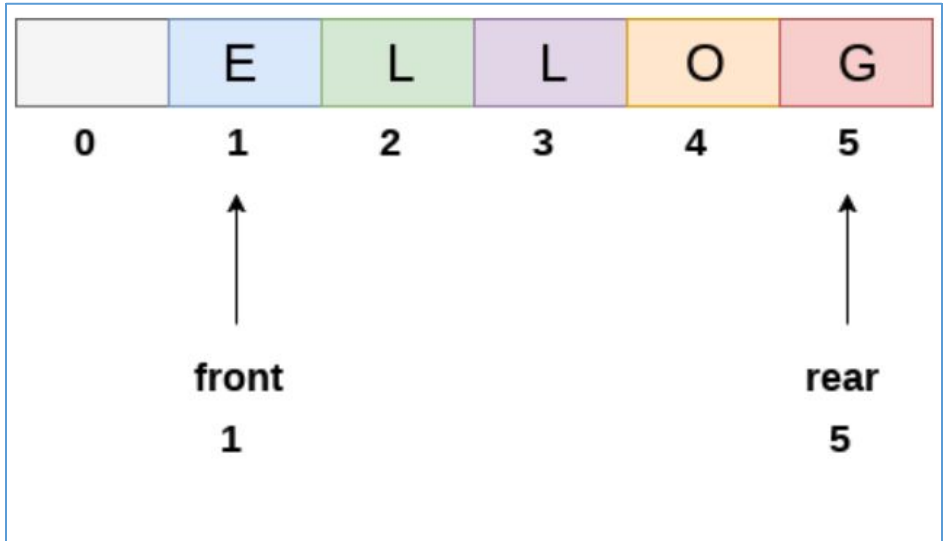
- As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures.
- For the sake of simplicity, we shall implement queues using one-dimensional array.

Array Representation





Queue after inserting an element



Queue after deleting an element

Queue Operations

Fields

- Rear – used to store position of insertion
- Front - used to store position of deletion
- Size – used to store the size of the queue
- Initial – front = rear = -1

Functions

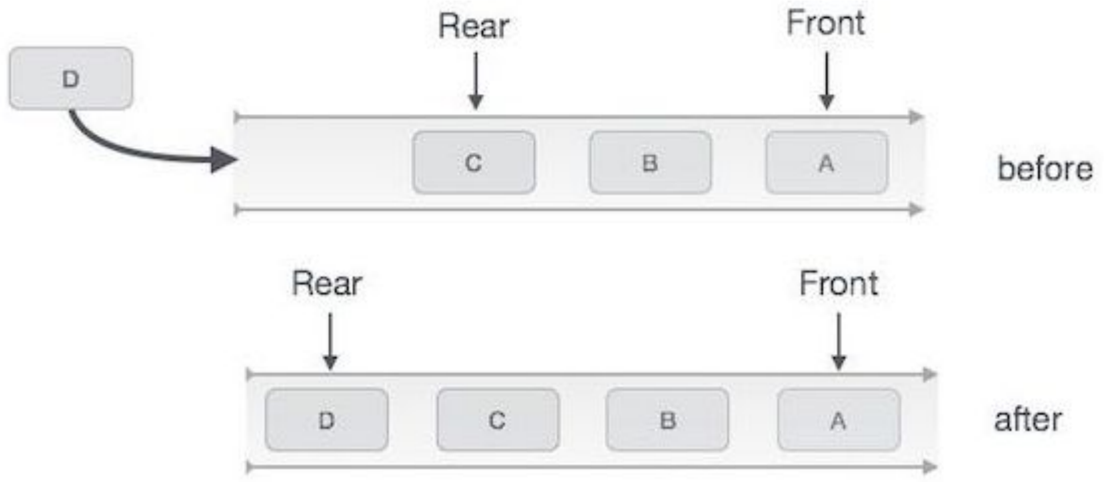
- enQueue(element) – Insert into the Queue
- deQueue() – To delete from Queue
- Isfull() – To check queue is full
- Isempty() – To check Queue is empty
- display() – display all elements in queue

Conditions

- Rear > = size – Queue is FULL
- Front = Rear = -1 – Queue is EMPTY

Queue Operation – Enqueue()

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



Queue Operation – Enqueue()

Algorithm

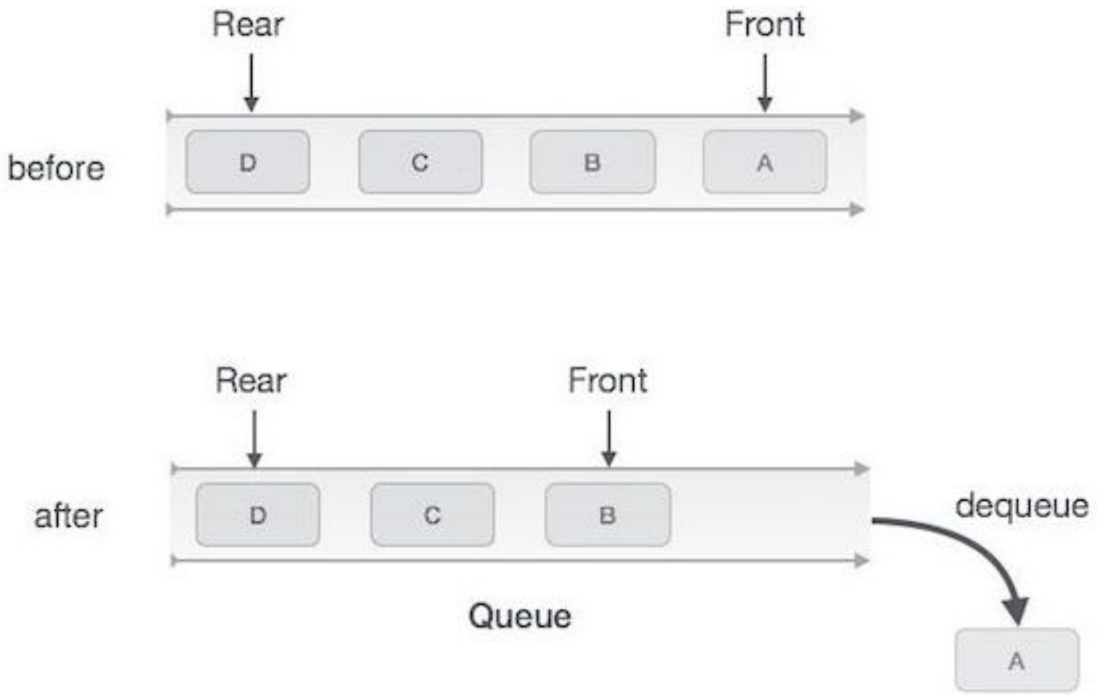
```
procedure enqueue(data)
if queue is full
   return overflow
endif
If front = -1
   front = 0;
rear ← rear + 1
queue[rear] ← data
return true
end procedure
```

Code

```
int enqueue(int data)
if(isfull())
   return 0;

If front = -1
   front = 0;
rear = rear + 1;
queue[rear] = data;

return 1;
end procedure
```

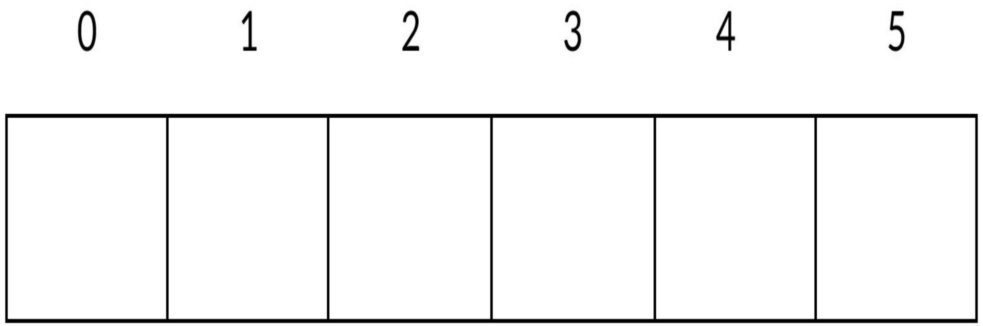
Queue Operation – dequeue()

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Operations

Front = Rear = -1



Empty Queue

Queue Operation – dequeue()

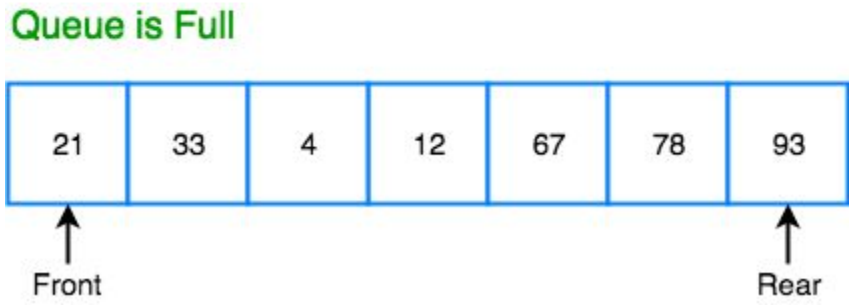
- Algorithm

```
Procedure dequeu
      if queue is empty
         return underflow

      if front < rear
         data = queue[front];
         front = front +1;
      else if front = rear
          front  =  rear = -1;
       end if
```

- Code

```
int dequeu() {
     if( isempty() )
             return 0;

      if (front < rear)
       {
          data = queue[front];
          front = front +1;
        }
      else if (front == rear)
        {
           data = queue[front];
           front = rear = -1;
         }
}
```

Queue Operation – isfull()



- Code

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

Queue Operation – isempty()

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

- Code

```
bool is empty() {
   if (front < 0 || front > rear) or (front == -1 && rear ==-1)
    {
       return true;
     }
    else
             return false;
}
```

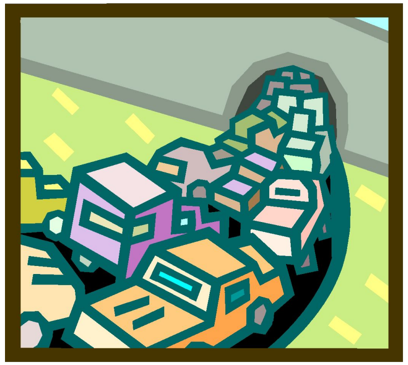
Queue Operation – display()

```
void display()
{
        if(front <= rear)
            for(i=front; i< =rear; i++)
                  printf("%d ", queue[i]);
}
```

Exercise: Queues

- Describe the output of the following series of queue operations
  - enqueue(8)
  - enqueue(3)
  - dequeue()
  - enqueue(2)
  - enqueue(5)
  - dequeue()
  - dequeue()
  - enqueue(9)
  - enqueue(1)

Applications of Queues



- Direct applications
  - Waiting lines
  - Access to shared resources (e.g., printer)

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

- Limitations
  - The maximum size of the queue must be defined *a priori* , and cannot be changed
  - Trying to enqueue an element into a full queue causes an implementation-specific exception