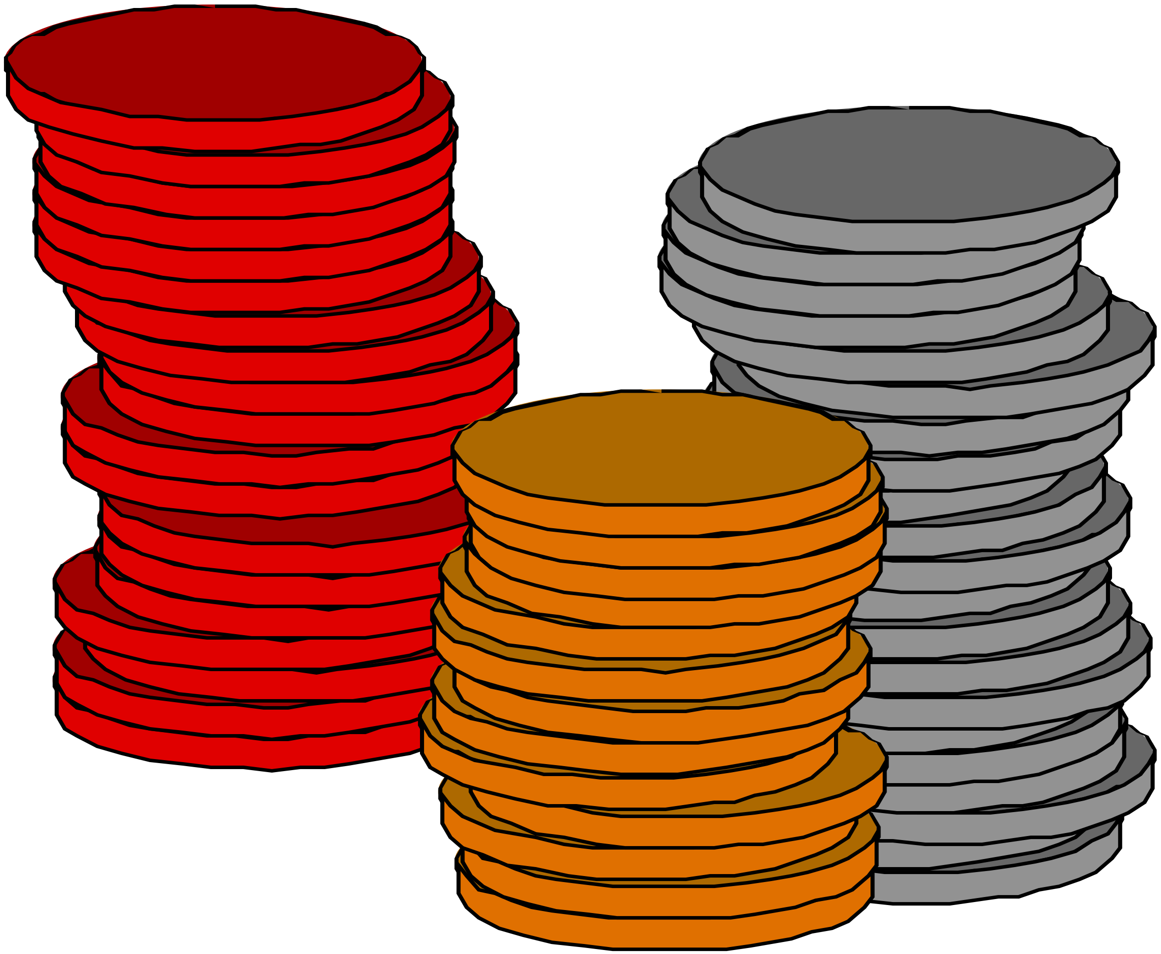
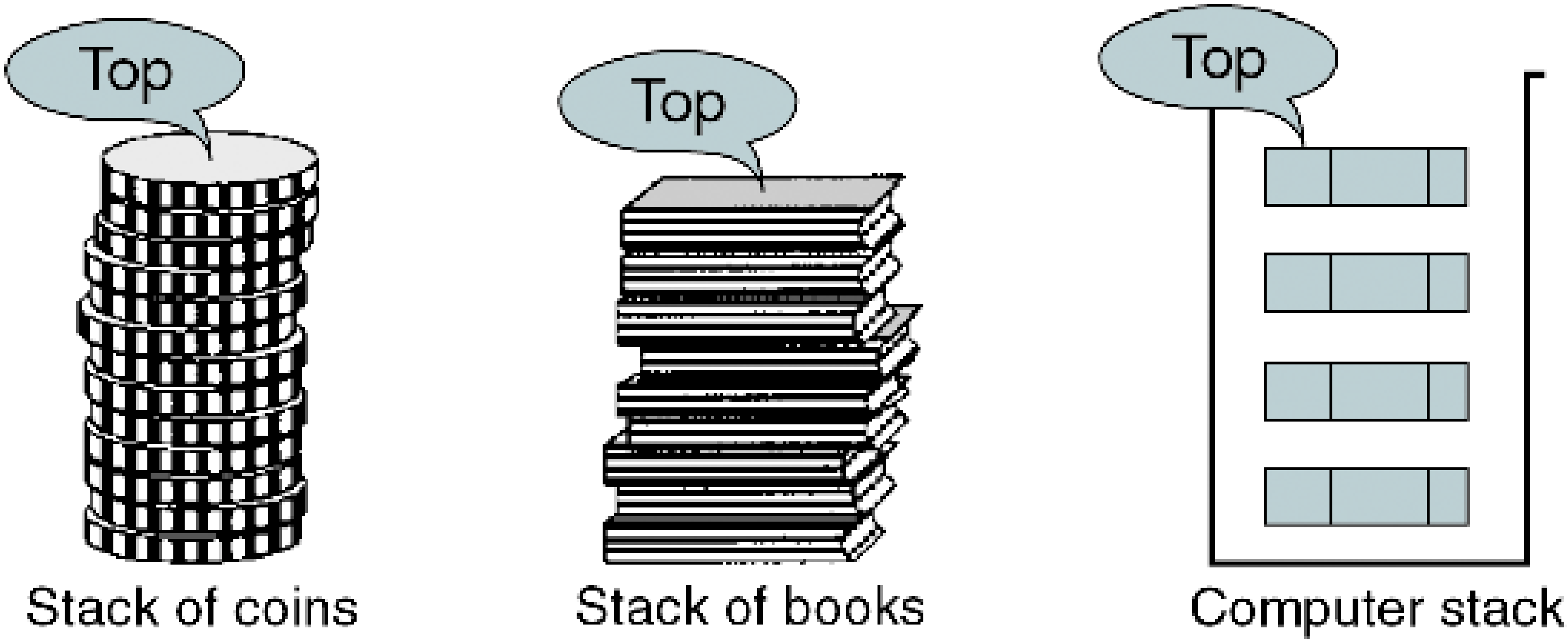
Stacks and Queues

STACK



STACK

- A **stack** is an ordered list in which insertions and deletions are made at one end called the top.
- A stack is also known as a **Last-In-First-Out** (**LIFO**) list. The last element inserted is the first one to be removed



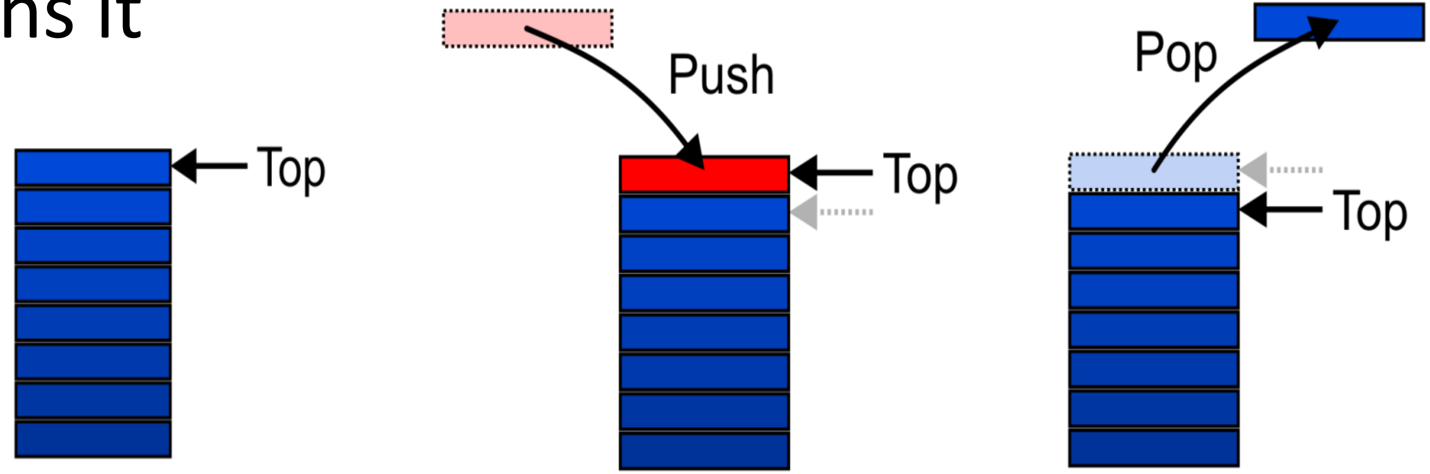
STACK Operations

push(object)

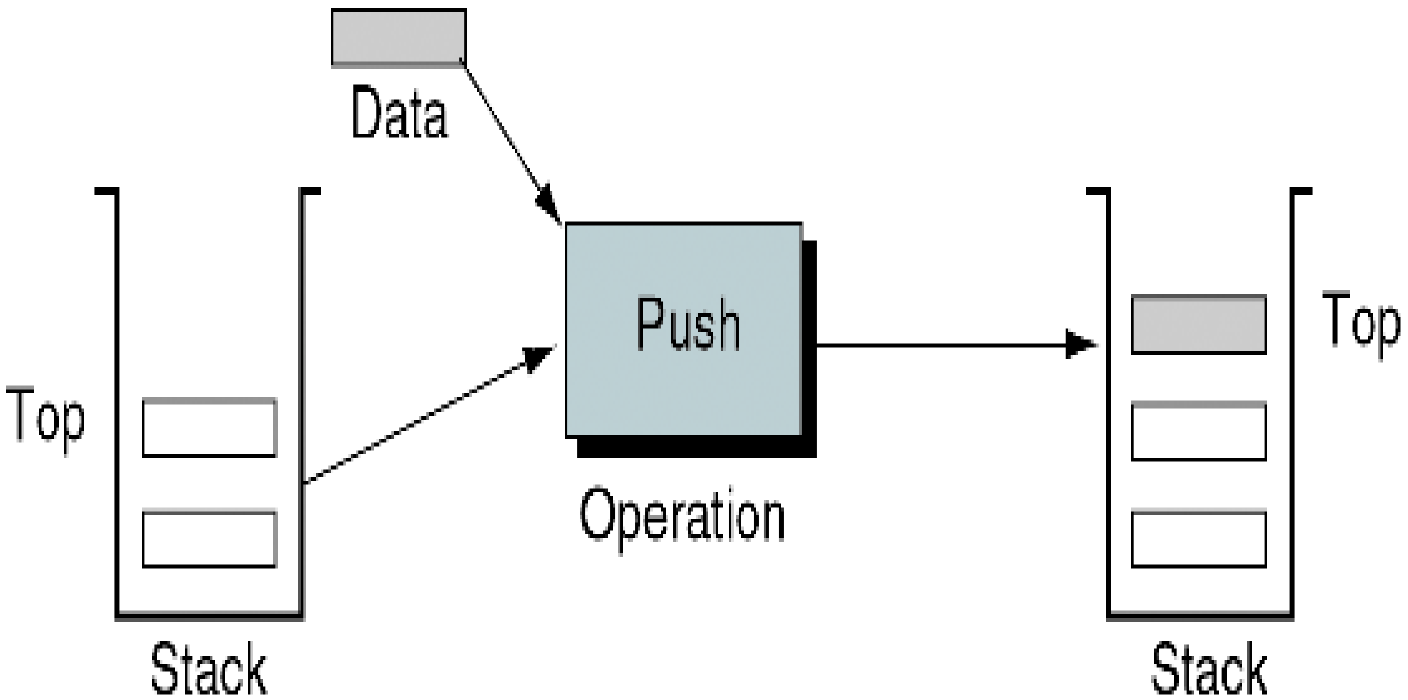
– Adds the object to the top of the stack; the item pushed is also returned as the value of **push**

object = pop()

– Removes the object at the top of the stack and returns it

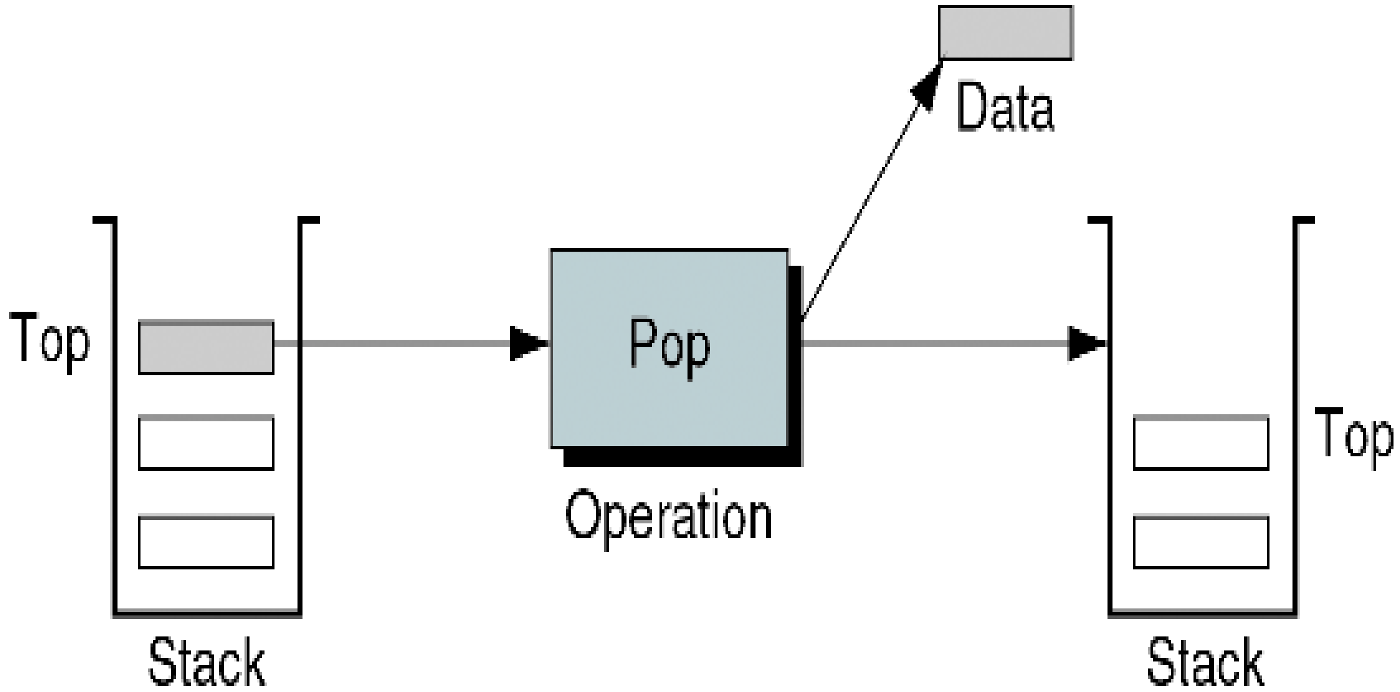


STACK Operation - PUSH



Push Stack Operation

STACK Operation - POP



Pop Stack Operation

Applications of Stacks

- Parsing
- Recursive Function
- Calling Function
- Expression Evaluation
- Expression Conversion
  - Infix to Postfix
  - Infix to Prefix
  - Postfix to Infix
  - Prefix to Infix

STACK implementation

- Stacks structures are usually implemented using
  - Arrays or
  - Linked lists.

Implementing Stacks : Array

- Advantages
  - best performance
- Disadvantage
  - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack

STACK Operation - PUSH

Algorithm PUSH_A (ITEM)

Input : The new item ITEM to be pushed onto it

Output : A stack with newly pushed ITEM at the TOP position

Data Structure : An array A with TOP as the pointer

Steps :

1. Top = -1
2. If TOP >= SIZE-1 then
    1. Print “Stack is full”
3. Else
    1. TOP = TOP+1
    2. A[TOP]=ITEM
4. Endif
5. Stop

STACK Operation - POP

Algorithm POP_A ( )

Input : A stack with elements

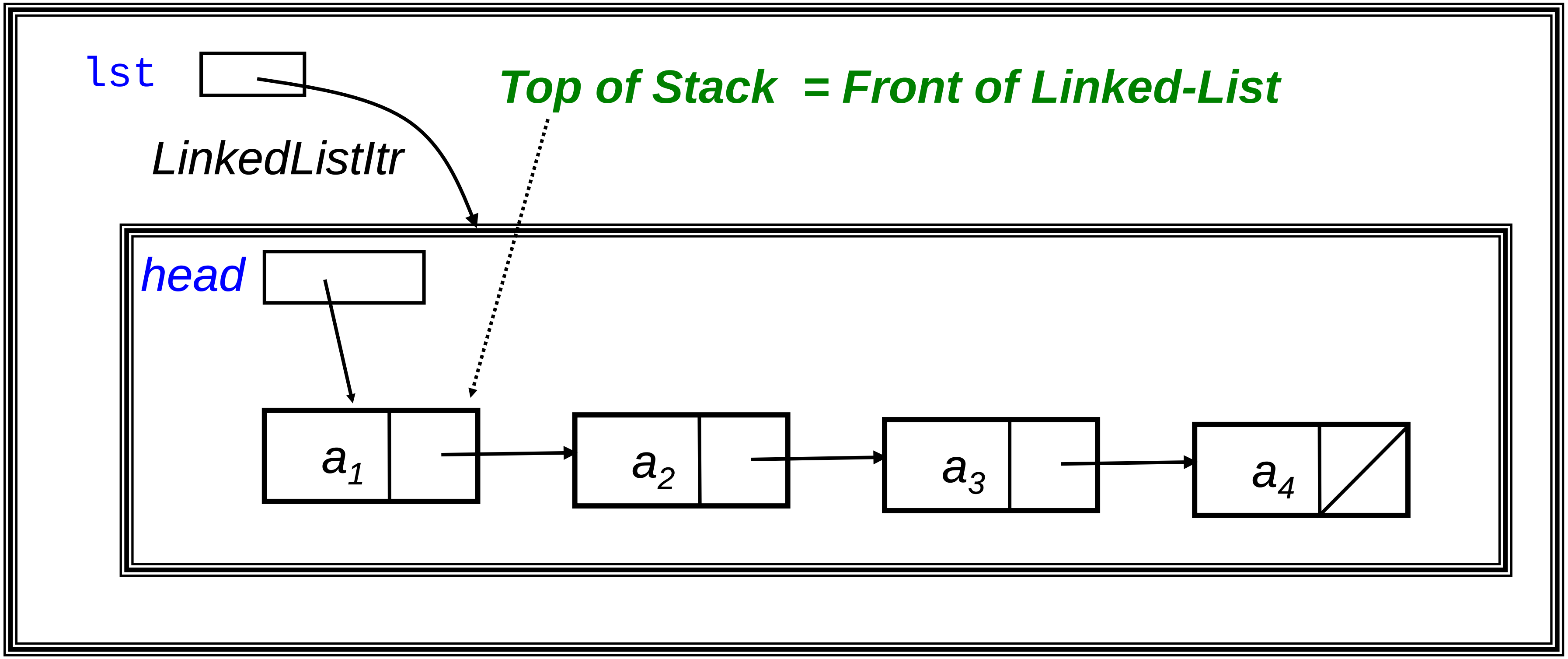
Output : Removes an ITEM from the top of the stack if it is not empty

Data Structure : An array A with TOP as the pointer

Steps :

1. If TOP < 0
    1. Print “Stack is empty”
2. Else
    1. ITEM = A[TOP]
    2. TOP = TOP-1
3. Endif
4. Stop

Implementing Stacks:Linked Lists



Algorithm PUSH_L (ITEM)

Input : The new item ITEM to be pushed onto it
Output : A single linked list with a newly inserted node with data content ITEM
Data Structure : singly linked list, header : STACK_HEAD

Steps :

1. New = GETNODE(NODE)
2. New.DATA = ITEM
3. New.LINK = STACK_HEAD
4. STACK_HEAD = New
5. Stop

STACK - POP

Algorithm POP_L ( )

Input : The new item ITEM to be poped from the stack
Output : A single linked list with a newly inserted node with data content ITEM
Data Structure : single linked list, header : STACK_HEAD

Steps :

1. If STACK_HEAD = NULL
    1. Print “Stack is empty”
    2. Exit
2. Else
    1. ITEM = STACK_HEAD.ITEM
    2. Ptr = STACK_HEAD
    3. STACK_HEAD = STACK_HEAD.LINK
    4. DELETE (Ptr)
3. Endif
4. Stop

Why postfix representation of the expression?

- Infix expressions are readable and solvable by humans because of easily distinguishable order of operators, but compiler doesn't have integrated order of operators.
- Hence to solve the Infix Expression compiler will scan the expression multiple times to solve the sub-expressions in expressions orderly which is very inefficient.
- To avoid this traversing, Infix expressions are converted to Postfix expression before evaluation.

Algorithm to Convert Infix Expression to Postfix using Stack

Step 1 : **Scan** the **Infix Expression** from **left to right**.

Step 2 : **If** the **scanned character** is an **operand**, **append** it with final Infix to **Postfix** string.

Step 3 : **Else**,

Step 3.1 : **If** the **precedence order** of the **scanned(incoming**) **operator** is **greater** than the **precedence** order of the **operator in** the **stack** (or the stack is **empty** or the stack contains a **'('** or **'['** or **'{'**), **PUSH it** on **stack**.

Algorithm to Convert Infix Expression to Postfix using Stack

Step 3.2 : **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.)

Step 4 : **If** the scanned **character** is an **'('** or **'['** or **'{', PUSH** it to the **stack**.

Algorithm to Convert Infix Expression to Postfix using Stack

Step 5 : **If** the scanned **character** is an **‘)’**or **‘]’** or **‘}’, POP** the stack and **output** it **until** a **‘(‘** or **‘[‘** or **‘{‘** respectively is **encountered**, **and discard both the parenthesis**.

Step 6 : **Repeat** steps **2-6** until **infix** expression is **scanned**.

Step 7 : **Print** the **output**

Step 8 : Pop and output from the stack until it is not empty.

Example : Convert Infix Expression to Postfix using Stack

Infix Expression : 3+4*5/6

Stack :

Output : 3

Stack : +

Output : 3

Stack : +

Output : 3 4

Stack : + *

Output : 3 4

Stack : + *

Output : 3 4 5

Stack : + /

Output : 3 4 5 *

Stack : + /

Output : 3 4 5 * 6

Stack :

Output : 3 4 5 * 6 / +

Convert an expression, I = ((6+2)*5–8/4)

Character scanned	Status of Stack	Postfix expression 'P'
(	(	
(	((	
6	((	6
+	((+	6
2	((+	6 2
)	(	6 2 +
*	(*	6 2 +
5	(*	6 2 + 5
-	(-	6 2 + 5 *
8	(-	6 2 + 5 * 8
/	(- /	6 2 + 5 * 8
4	(- /	6 2 + 5 * 8 4
)	(-	6 2 + 5 * 8 4 /
	(	6 2 + 5 * 8 4 / -

Evaluate a postfix expression

1. Read the tokens from the postfix string one at a time from left to right.
2. Initialize an empty stack
3. If the token is an **operand**, **PUSH** the operand into the stack
4. If the token is an **operator**, then **POP** the top two elements from the stack and apply the operator on the poped out elements. The result of this operation is pushed back into the stack.
5. After all the tokens are read, only one element is present in the stack and that is the result.

Evaluate a postfix expression

Infix is 3*4 + 2*5

Postfix is 3 4 * 2 5 * +

3 4 * 2 5 * +

* is the first operator 3 4 * is replaced by 12

12 2 5 * +

13 2 5 * is replaced by 10

12 10 +

12 10 + is replaced by 22

22

Evaluate the following postfix expression using a stack:

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

Evaluate a postfix expression

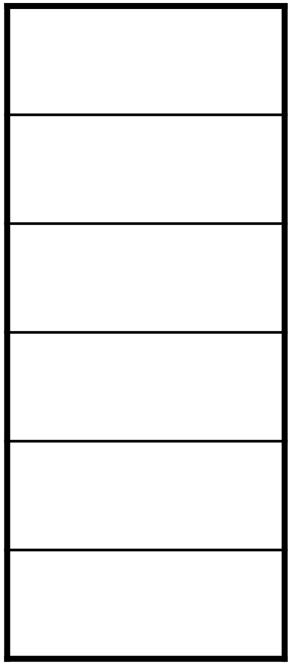
Evaluate the following postfix expression using a stack:

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$

Evaluate a postfix expression

1 2 3 + 4 5 6 × − 7 × + − 8 9 × +

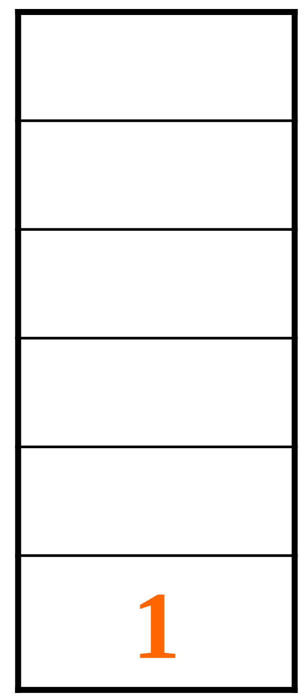
PUSH 1 onto the stack



Evaluate a postfix expression

2 3 + 4 5 6 × − 7 × + − 8 9 × +

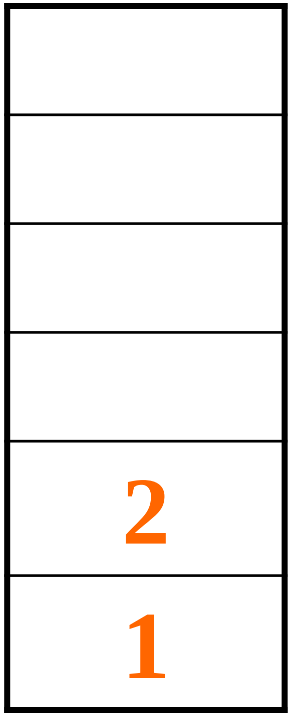
PUSH 2 onto the stack



Evaluate a postfix expression

3 + 4 5 6 × – 7 × + – 8 9 × +

PUSH 3 onto the stack



Evaluate a postfix expression

+ 4 5 6 × – 7 × + – 8 9 × +

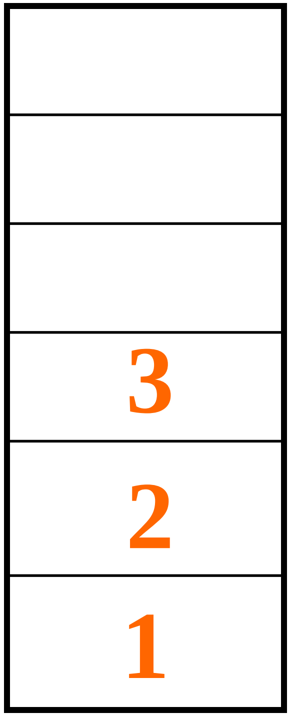
Next character is an operator +.

So, POP 3 and 2,

Apply the operator +

Now

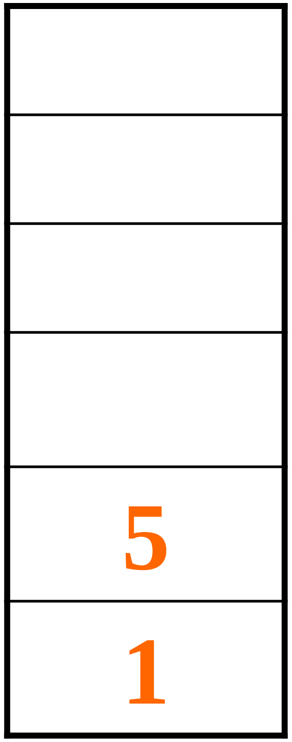
PUSH result of 2 + 3 = 5 to Stack



Evaluate a postfix expression

4 5 6 × – 7 × + – 8 9 × +

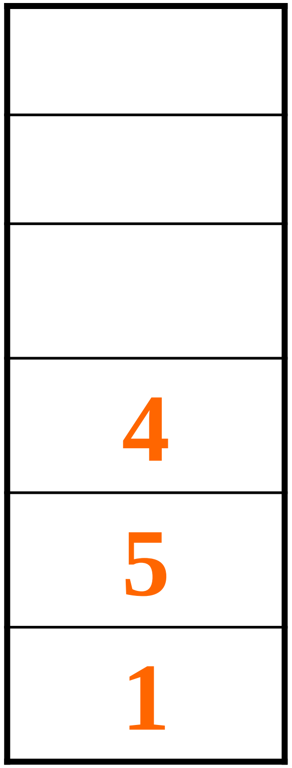
Push 4 onto the stack



Evaluate a postfix expression

5 6 × – 7 × + – 8 9 × +

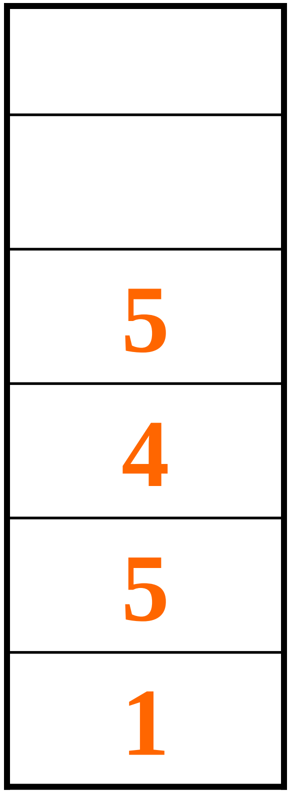
Push 5 onto the stack



Evaluate a postfix expression

$$6 \times - 7 \times + - 8\ 9 \times +$$

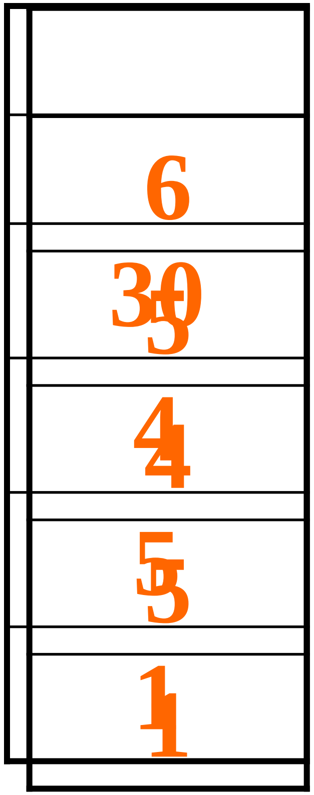
Push 6 onto the stack



Evaluate a postfix expression

$x \; - \; 7 \; \times \; + \; - \; 8 \; 9 \; \times \; +$

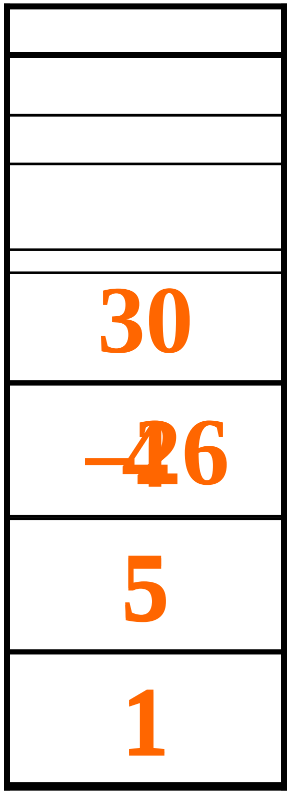
Pop 6 and 5 and push 5 × 6 = 30



Evaluate a postfix expression

– 7 × + – 8 9 × +

Pop 30 and 4 and push 4 – 30 = –26



Evaluate a postfix expression

$$7 \;\times\; + \;-\; 8 \;\; 9 \;\times\; +$$

Push 7 onto the stack

7
–26
5
1

Evaluate a postfix expression

× + – 8 9 × +

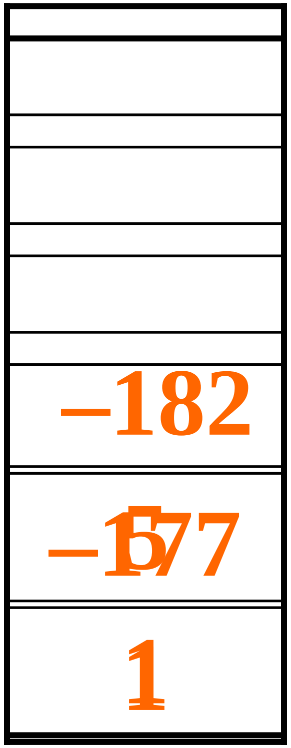
Pop 7 and –26 and push –26 × 7 = –182

–182
5
1

Evaluate a postfix expression

$+ \quad - \quad 8 \quad 9 \quad \times \quad +$

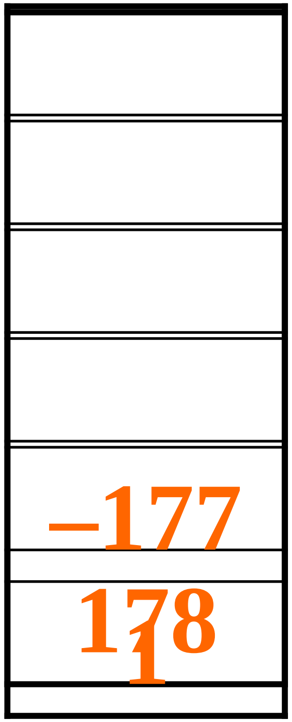
Pop -182 and 5 and push $-182 + 5 = -177$



Evaluate a postfix expression

– 8 9 × +

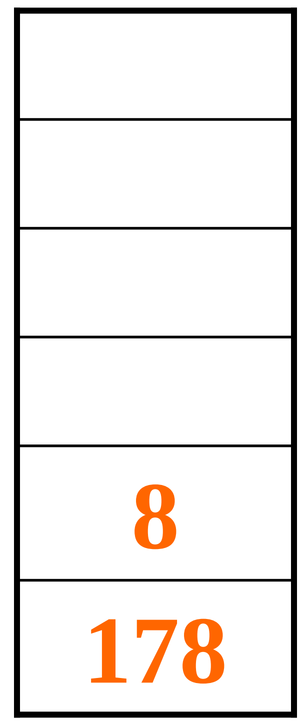
Pop –177 and 1 and push 1 – (–177) = 178



Evaluate a postfix expression

8 9 × +

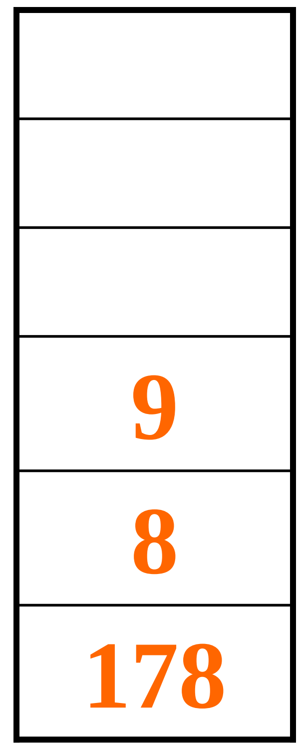
Push 8 onto the stack



Evaluate a postfix expression

9 × +

Push 9 onto the stack



Evaluate a postfix expression

× +

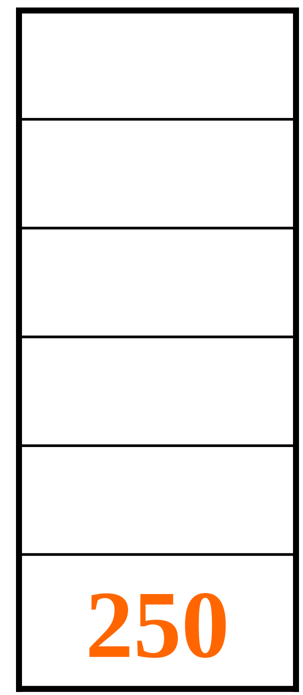
Pop 9 and 8 and push 8 × 9 = 72

72
178

Evaluate a postfix expression

+

Pop 72 and 178 and push 178 + 72 = 250



Evaluate a postfix expression

Thus

$$\mathbf{1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +}$$

evaluates to the value : 250

Queues

Linear list.

One end is called **front**.

Other end is called **rear**.

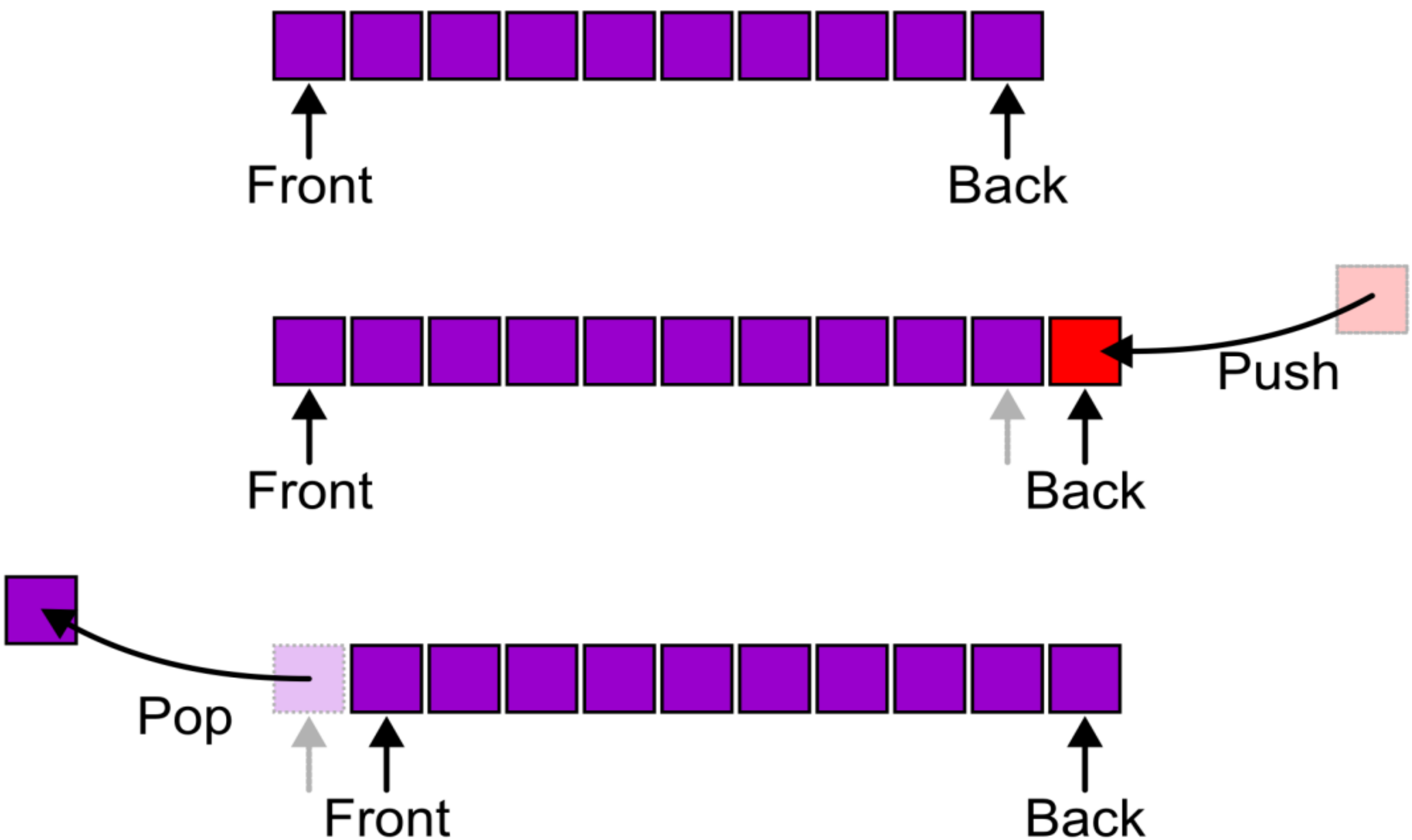
Additions are done at the **rear** only.

Removals are made **from** the **front** only.

Queue

First-In–First-Out (FIFO) or

First Come First Serve (FCFS) data structure



Queue

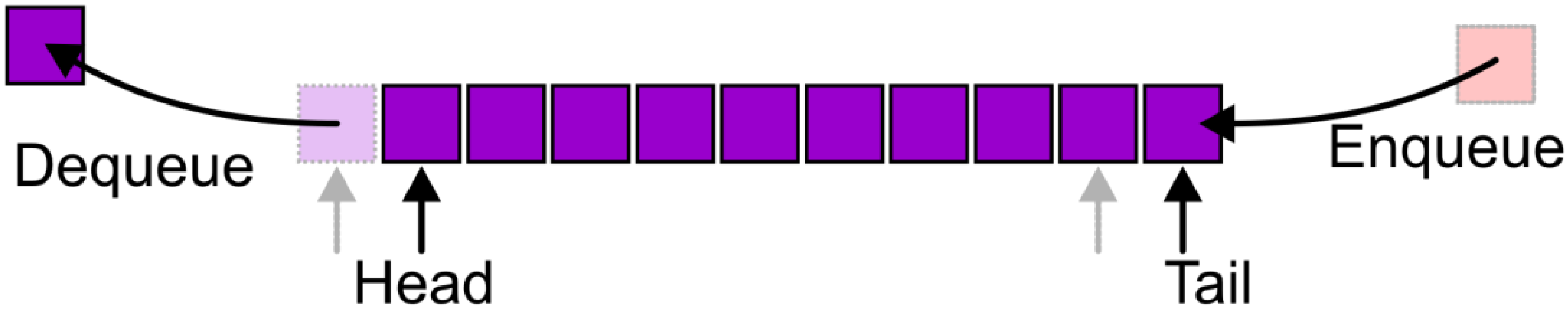
Alternative terms may be used for the four operations on a queue, including:

ENQUEUE(PUSH),

DEQUEUE(POP),

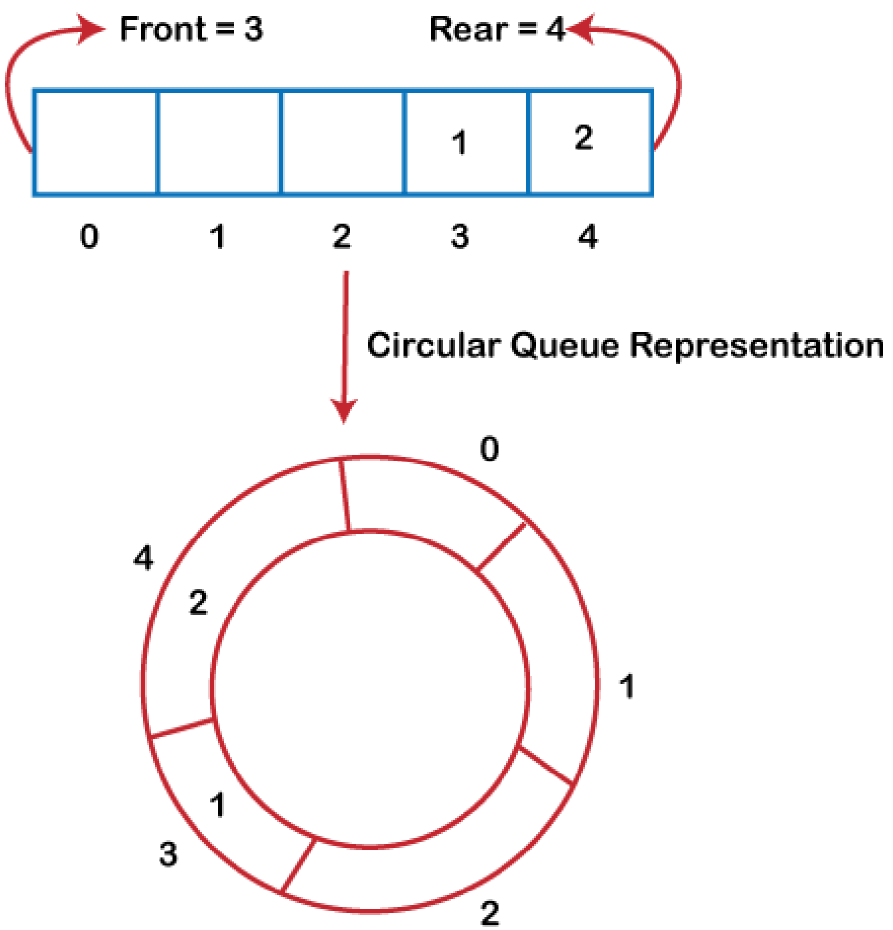
HEAD(FRONT),

TAIL (BACK)

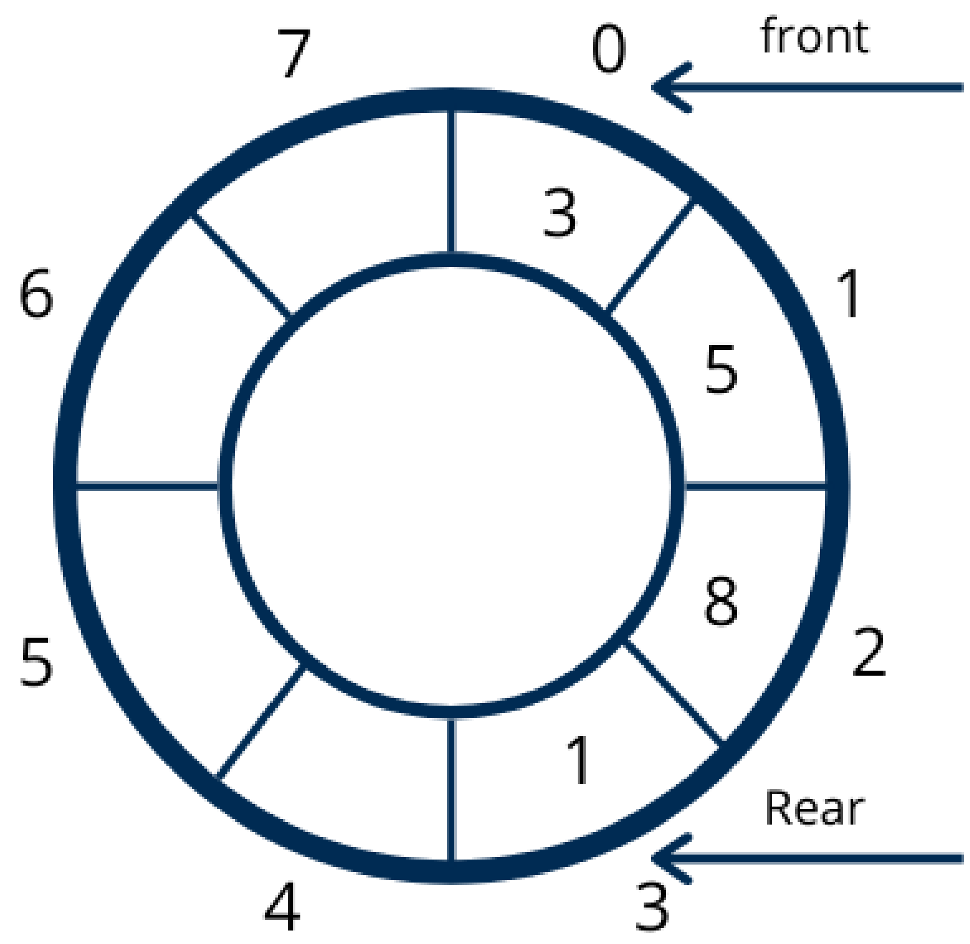


Circular Queue

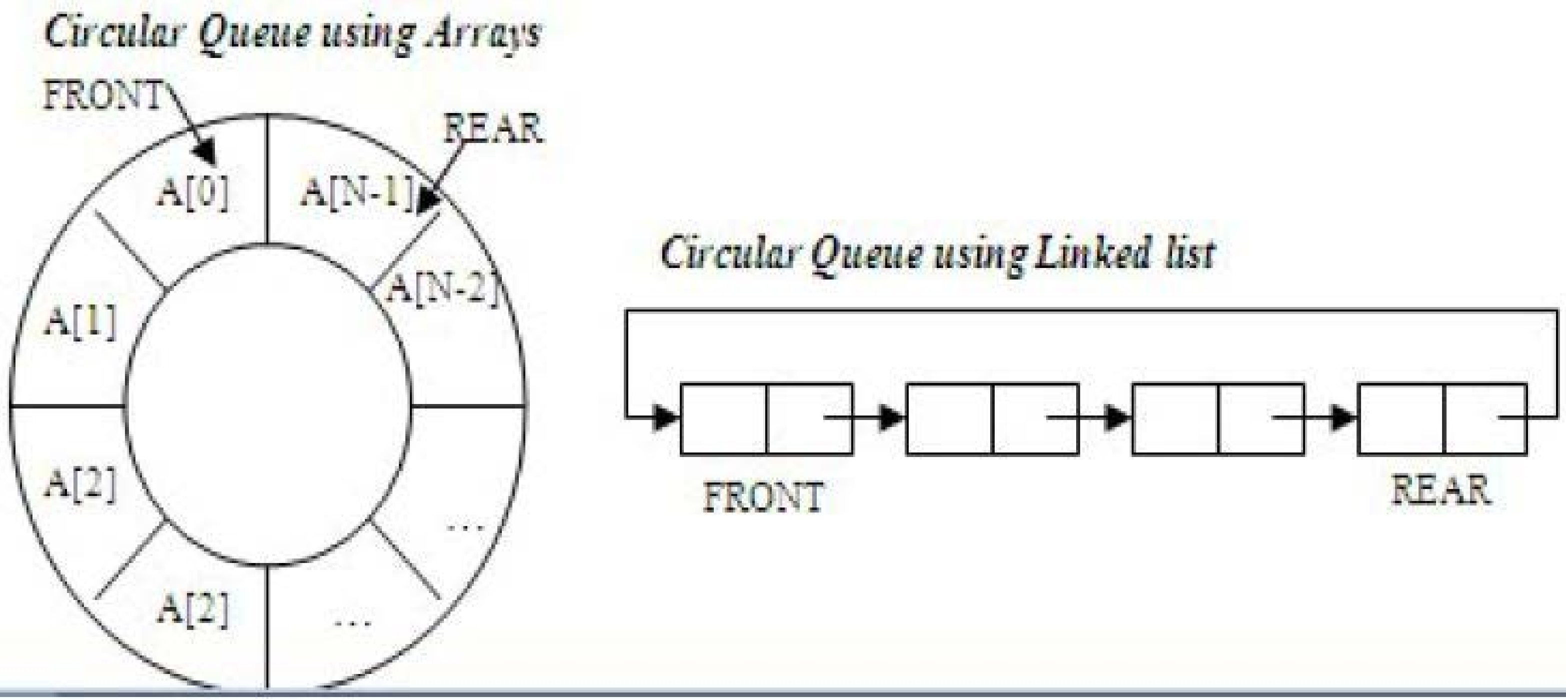
In a Circular queue the last element is connected to the first element of the queue forming a circle.



Circular Queue

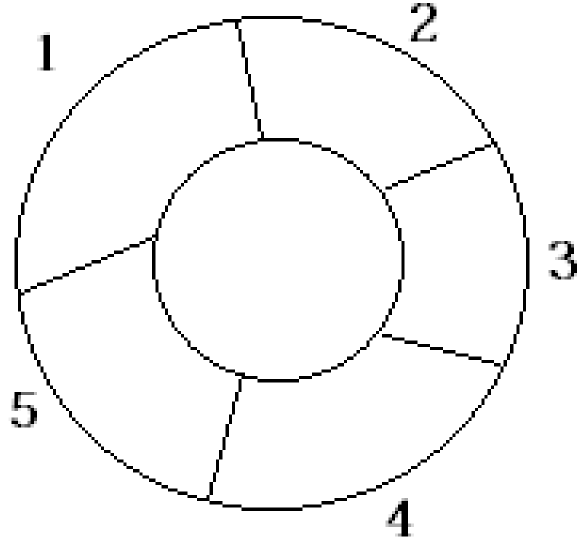


Circular Queue

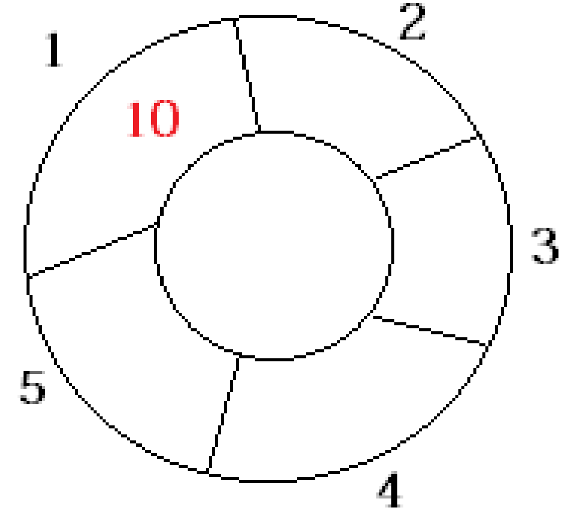


Circular Queue : Insert & Delete

1. Initially Rear=0, Front=0

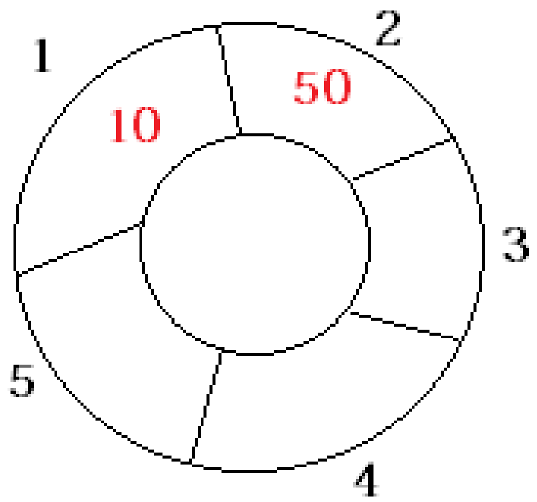


2.Insert 10, Rear =1 , Front =1

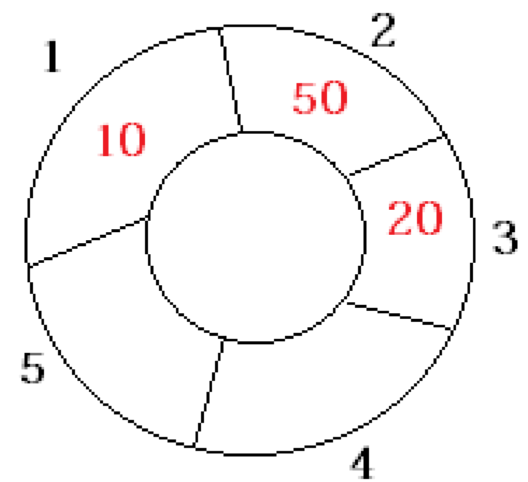


Circular Queue : Insert & Delete

3. Insert 50, Front=1, Rear = 2

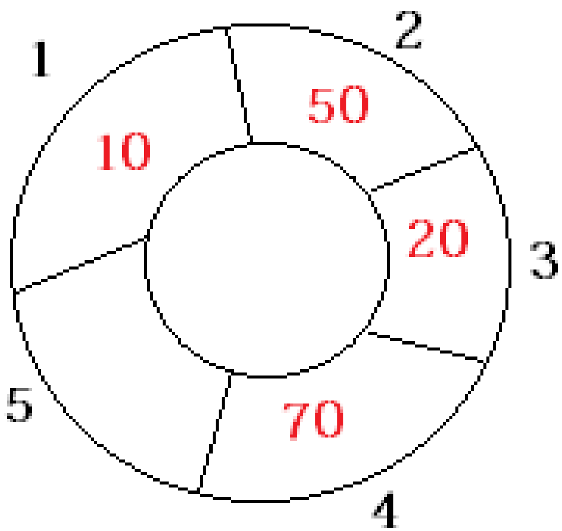


4. Insert 20 , Front = 1, Rear = 3

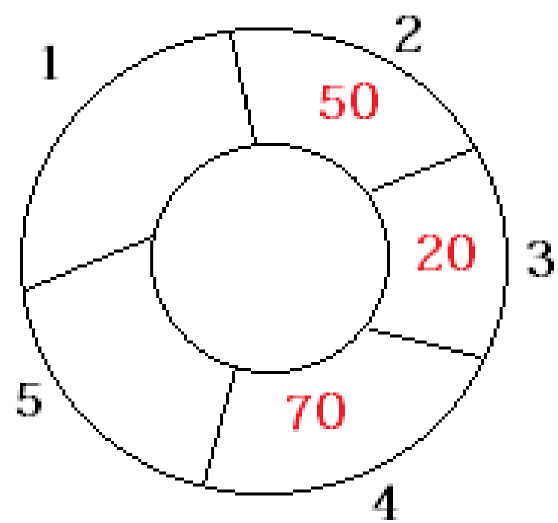


Circular Queue : Insert & Delete

5. Insert 70, Front =1 , Rear = 4

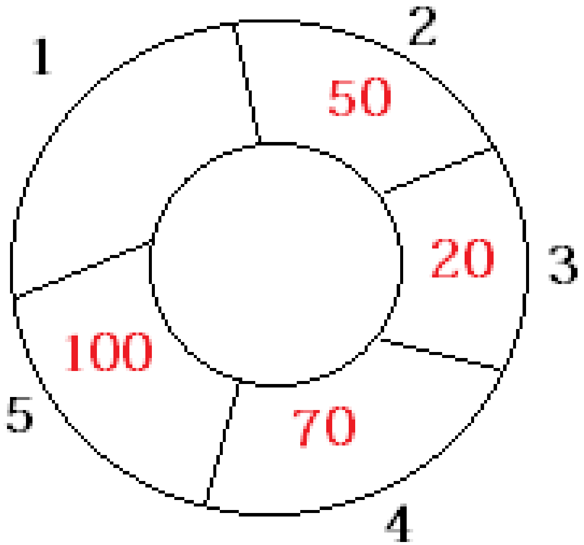


6. Delete Front, Front = 2, Rear = 4

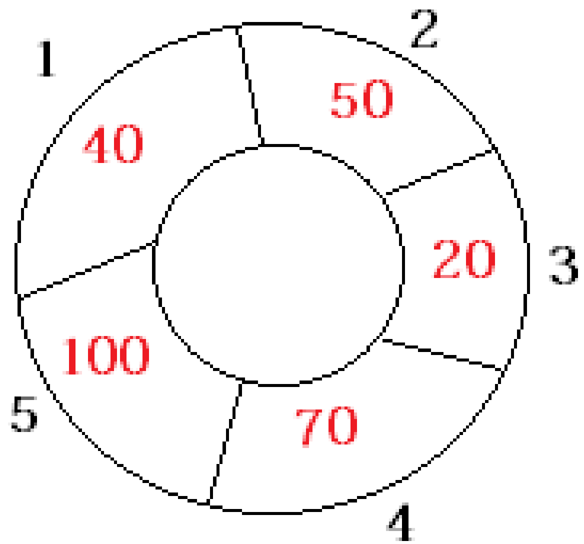


Circular Queue : Insert & Delete

7.Insert 100, Front =2 , Rear = 5



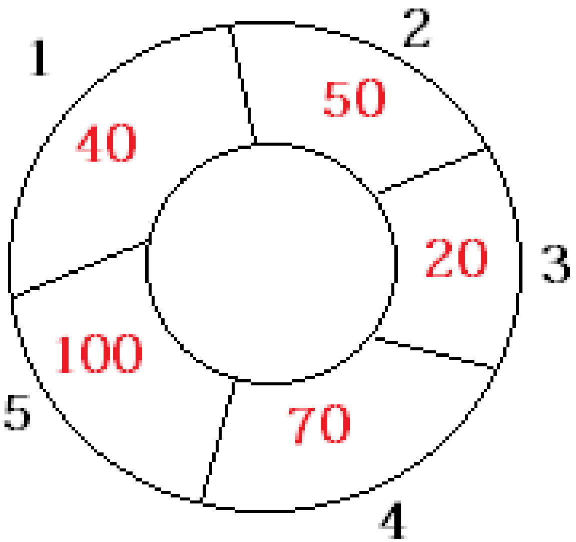
8. Insert 40, Front = 2, Rear = 1



Circular Queue : Insert & Delete

9. Insert 150, Front = 2, Rear = 1

"Queue Overflow"



10. Delete Front, Front=3, Rear = 1