Parul®
University

Unit 2
Stacks and queue

Prof. Shreya Dholariya, Assistant Professor
Computer Science & Engineering

Topic-1

Stacks

ADT Stack and its operations:

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.
- ADT - The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.

Image source : Google

ADT Stack and its operations:

- The abstract data type is special kind of data type, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive data types, but operation logics are hidden.

- A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

- This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

- A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

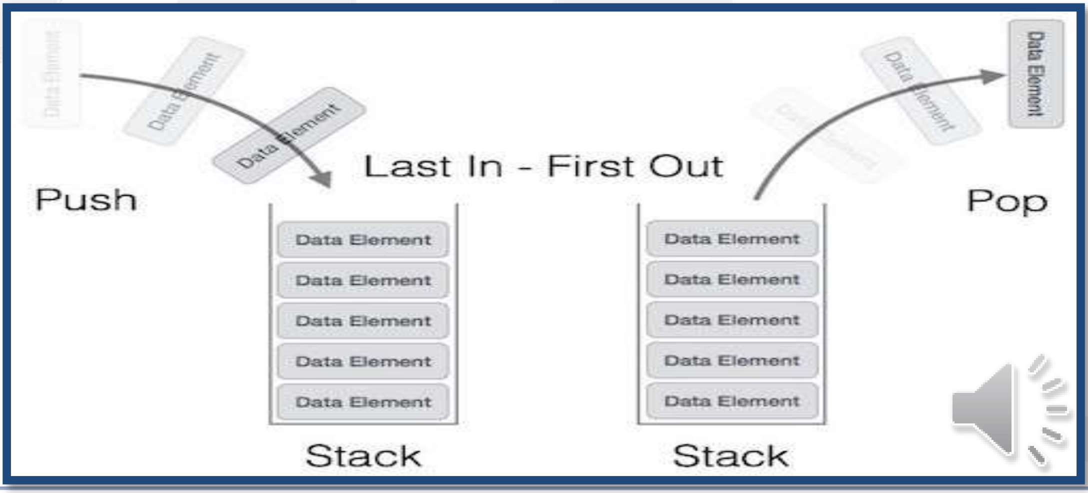


Image source : Google

Basic Operations

- Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

push() – Pushing (storing) an element on the stack.

pop() – Removing (accessing) an element from the stack.

Stack Operations

- To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

peek() − get the top data element of the stack, without removing it.

isFull() − check if stack is full.

isEmpty() − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

Push Operation

•The process of putting a new data element onto stack is known as a Push Operation.

•Push operation involves a series of steps.

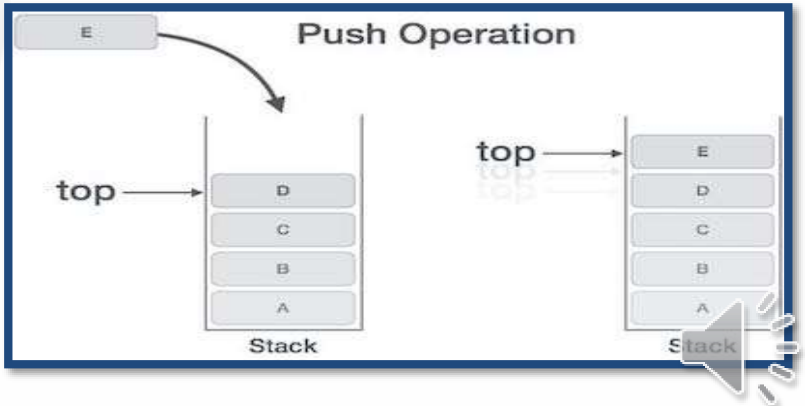
Step 1 – Checks if the stack is full.

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.



Push Function in ‘C’

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

Pop Operation

- Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of 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. But in linked-list implementation, pop() actually removes data element and deallocates memory space.
- A Pop operation may involve the following steps –

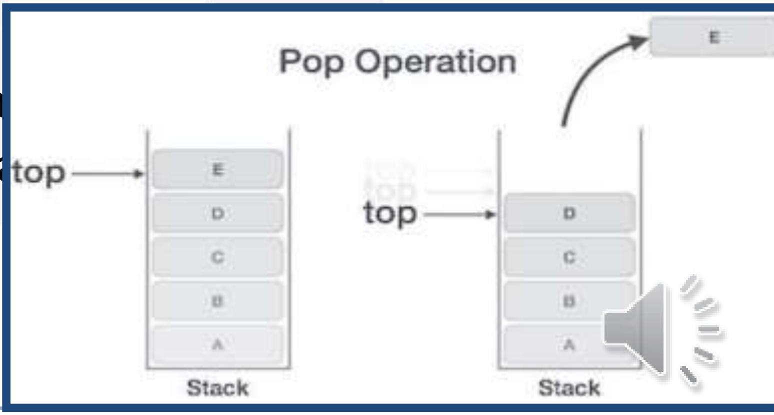
Step 1 – Checks if the stack is empty.

Step 2 – If the stack is empty, produces an error an

Step 3 – If the stack is not empty, accesses the dat
which **top** is pointing.

Step 4 – Decreases the value of top by 1.

Step 5 – Returns success.



Pop Function in ‘C’

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

Analysis of Stack Operations

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

Push Operation : O(1)
Pop Operation : O(1)
Top Operation : O(1)
Search Operation : O(n)

The time complexities for push() and pop() functions are O(1) because we always have to insert or remove the data from the **top** of the stack, which is a one step process.

Application’s Of Stack

- Recursion
- Polish Expression & their Compilation
- Reverse Polish Expression & their Compilation
- Stacks are used by compilers to check weather the contents of “string” belong to a particular language

Recursion

- Recursion is a programming Technique in which a function contains a function call to itself
- Recursion and iteration are different processes. Recursion is a process in which the function contains a function to call itself. Where iteration is a process where the group of statement is executed repeatedly.

Eg:

```
factorial(int x) {
      int f;
     if(x==1)
        return(1);
    else
      f=x*factorial(x-1);   //This is recursion(recursive call)
         return (f);
  }
```

Types Of Recursion

- Direct Recursion

```
Eg:-factorial(int x){
          factorial(int x-1); }
```

- Indirect Recursion

```
Eg:-
         int a()
          {
            b();
           }
          int b()
            {
              a();
             }
```

Tower Of Hanoi

- Tower of Hanoi basically comes from a Chinese or Japanese source

- It’s a simple game concept which can be applied in stacks. The concept is that the bigger value should not be kept upon the smaller value

- It is necessary to stack all disks onto a third tower in decreasing order of size
- The second tower may be used as temporary storage

- The conditions are
  1) only one value may be moved at a time
  2) the larger value never rest upon smaller value
- Formula for finding minimum number of moves is 2^n-1

Algorithm For Tower Of Hanoi

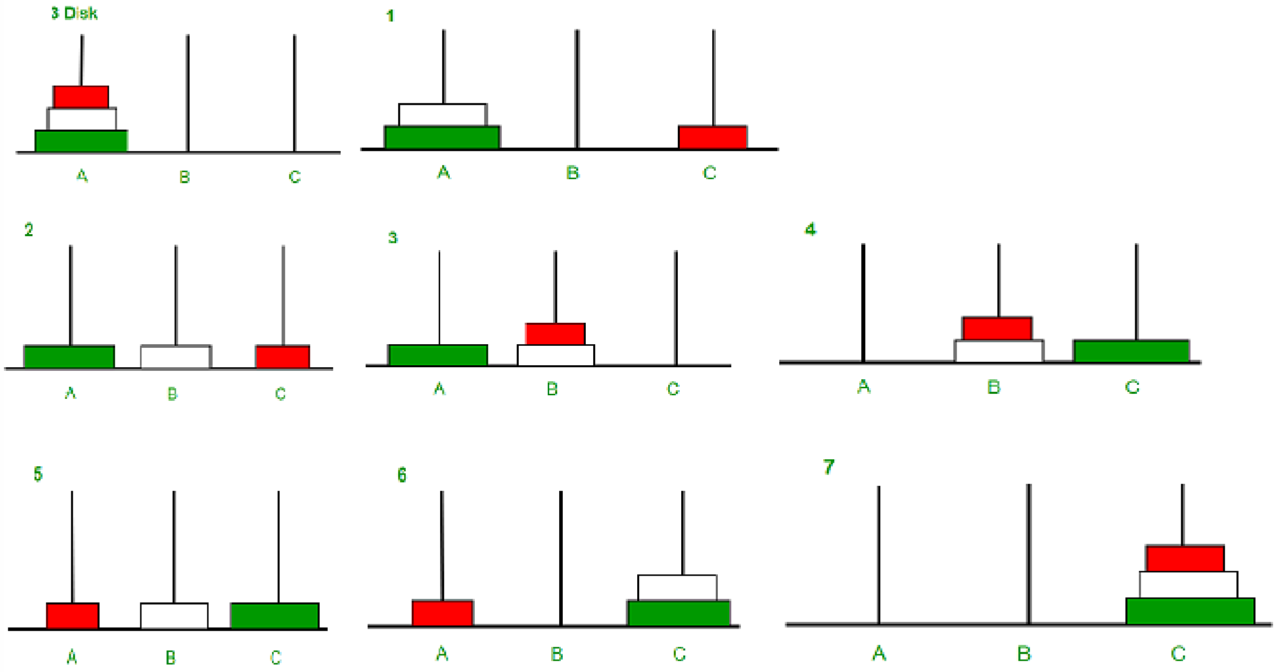
```
Step 1:- if n=1 then print"disc -1 from A->C & return"

Step 2:- [Move top n-1 disks from A->B]
                                   Tower(n-1,a,c,b)

Step 3:- print"disc-n from A->C"

Step 4:- [Move n-1 disks from B->C]
                                   tower(n-1,b,a,c)
Step 5):- return
```

Tower of Hanoi For N=3



Movement of Discs

1. A→C
2. A→B
3. C→B
4. A→C
5. B→A
6. B→C
7. A→C

Expression Parsing

- The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.

  These notations are –

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression

Infix Notation and Prefix Notation

- Infix Notation

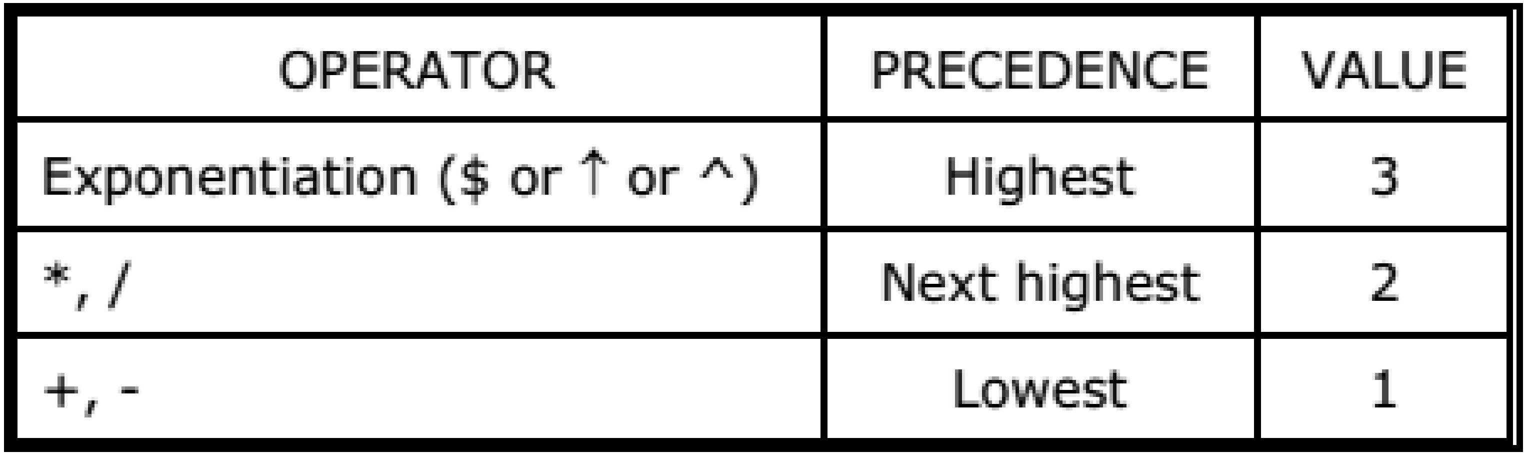
We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

- Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation and Algorithm

- This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.



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

Infix to Postfix Conversion Algorithm

We use a stack

1 - scan the expression from left to right

2 - When an operand is read, output it

3 - When an operator is read – Pop until the top of the stack has an element of lower precedence – Then push it

4 - When) is found, pop until we find the matching (

5 - (has the lowest precedence when in the stack

6 - but has the highest precedence when in the input

7 - When we reach the end of input, pop until the stack is empty

Infix to Postfix Conversion Example

Example 1 --3+4*5/6

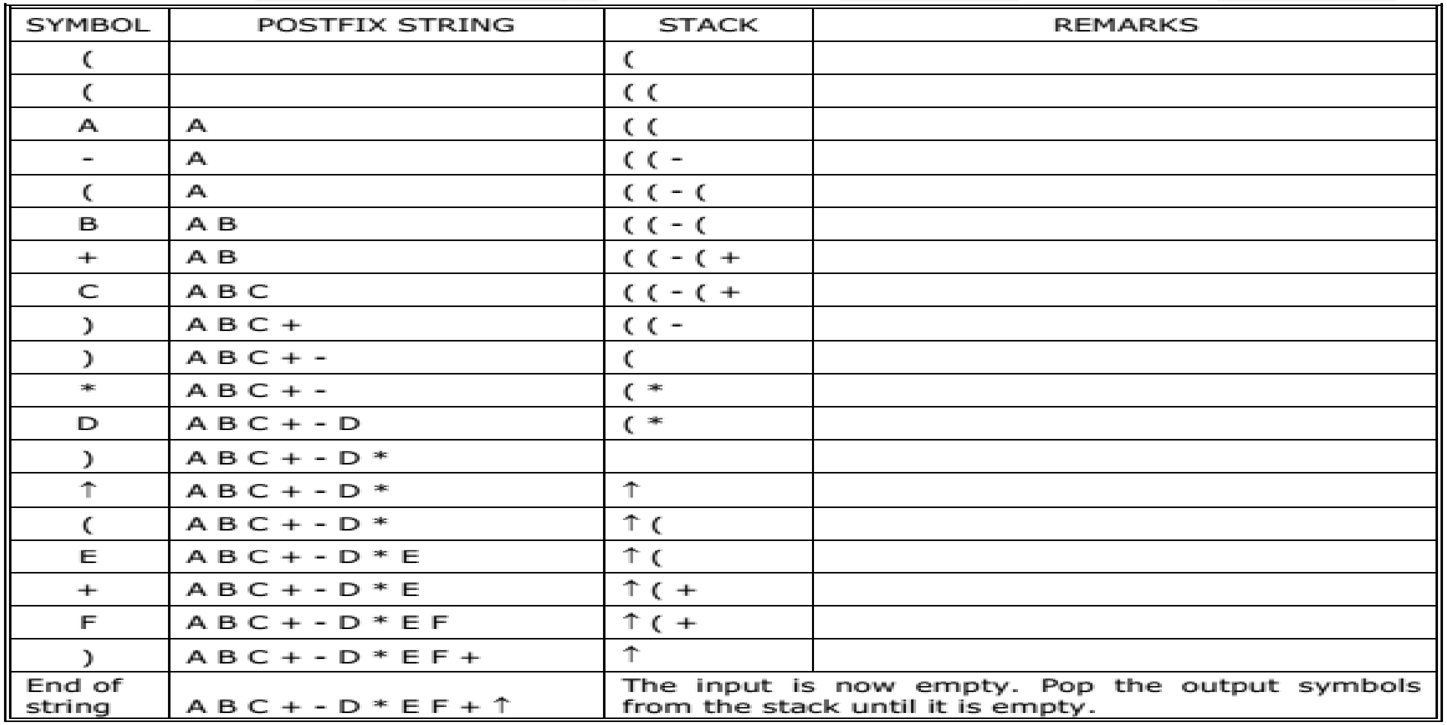
1. 3+4*5/6	2. 3+4*5/6 • Stack: • Output:	3. 3+4*5/6 • Stack: • Output: 3
4. 3+4*5/6 • Stack: + • Output: 3	5. 3+4*5/6 • Stack: + • Output: 3 4	6. 3+4*5/6 • Stack: + * • Output: 3 4

Infix to Postfix Conversion Example

7. 3+4*5/6 - Stack: + * - Output: 3 4 5	8. 3+4*5/6 - Stack: + - Output: 3 4 5 *	9. 3+4*5/6 - Stack: + / - Output: 3 4 5 *
10. 3+4*5/6 - Stack: + / - Output: 3 4 5 * 6	11. 3+4*5/6 - Stack: + - Output: 3 4 5 * 6 /	12. 3+4*5/6 - Stack: - Output: 3 4 5 * 6 / + Done!

Infix to Postfix Conversion Example 1

Example 1 : ((A-(B+C))*D)^(E+F)



SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Infix to Prefix Conversion Algorithm

1. Reverse the infix expression. And follow the steps of infix to post fix conversion algorithm .

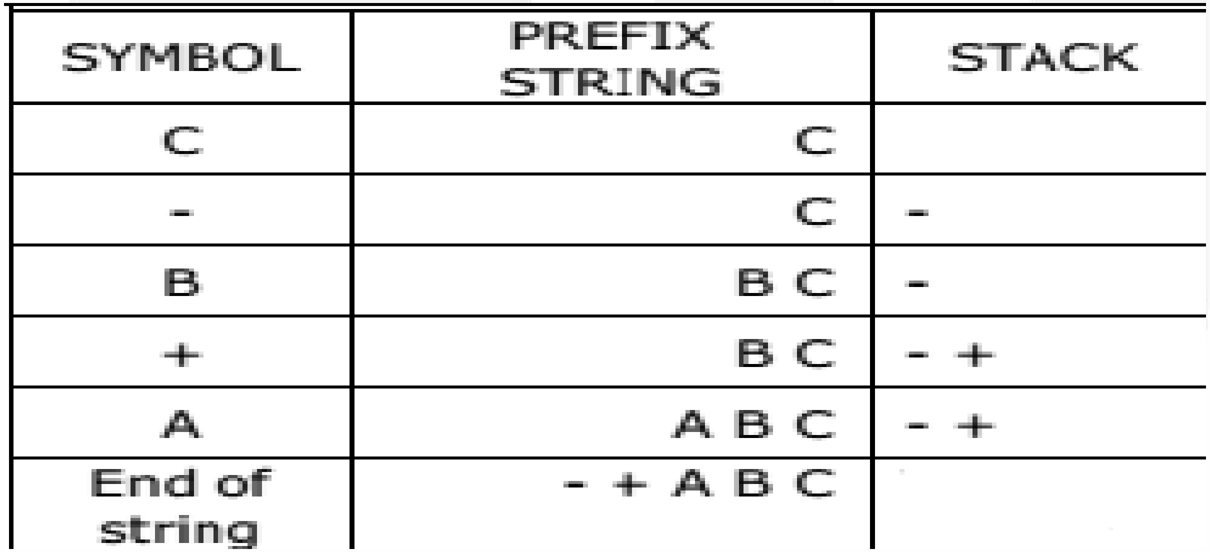
2. After getting postfix expression , reverse it and this is the prefix expression of the infix expression.

Infix to Prefix Conversion Example

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 * +
A + B + C + D	+ + + A B C D	A B + C + D +

Infix to Prefix Conversion Example

Example 1 : Infix Expression –> A+B-C



SYMBOL	PREFIX STRING	STACK
C	C	
-	C	-
B	B C	-
+	B C	- +
A	A B C	- +
End of string	- + A B C	

Evaluation of Postfix & prefix Expression

- After converting infix to postfix, we need postfix evaluation algorithm to find the correct answer.
- From the postfix expression, when some operands are found, pushed them in the stack. When some operator is found, two items are popped from the stack and the operation is performed in correct sequence. After that, the result is also pushed in the stack for future use. After completing the whole expression, the final result is also stored in the stack top.

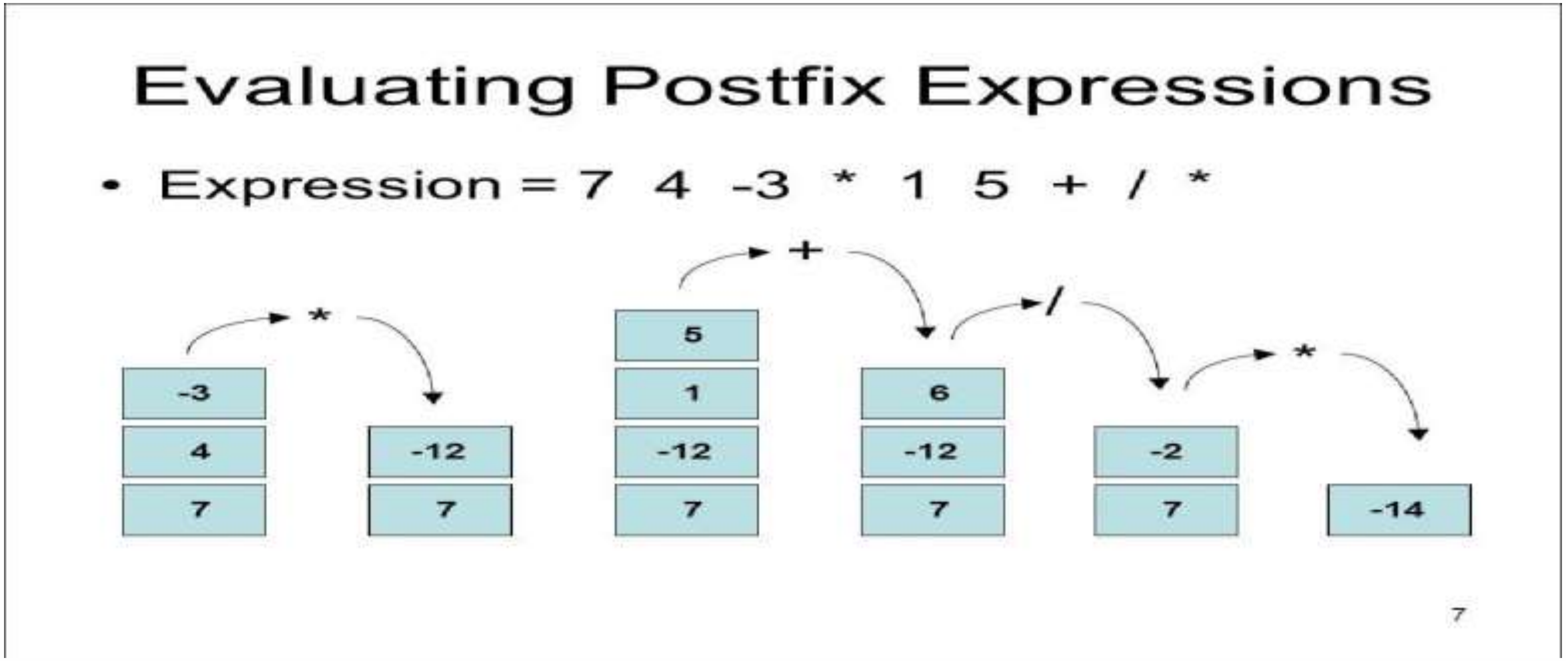
Evaluation of expression Algorithm

Input: Postfix expression to evaluate.
Output: Answer after evaluating postfix form.

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

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 expression Example



Evaluating Postfix Expressions

- Expression = 7 4 -3 * 1 5 + / *

Links for conversion of Expressions

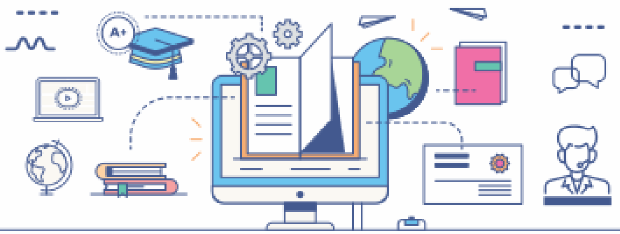
1. Infix to postfix
 https://www.youtube.com/watch?v=vq-nUF0G4fI

2. Infix to Prefix coversion
https://www.youtube.com/watch?v=UK16ttNfGSk

3.Evaluation of postfix and prefix expressions
https://www.youtube.com/watch?v=MeRb_1bddWg

Topic-2

Queues



Topics Covered

- ➢ADT Queue
- ➢Types of Queues:
  - ▪ Simple Queue
  - ▪ Circular Queue
  - ▪ Priority Queue
- ➢ Operations on each types of queues:
  - ▪ Algorithms and their analysis

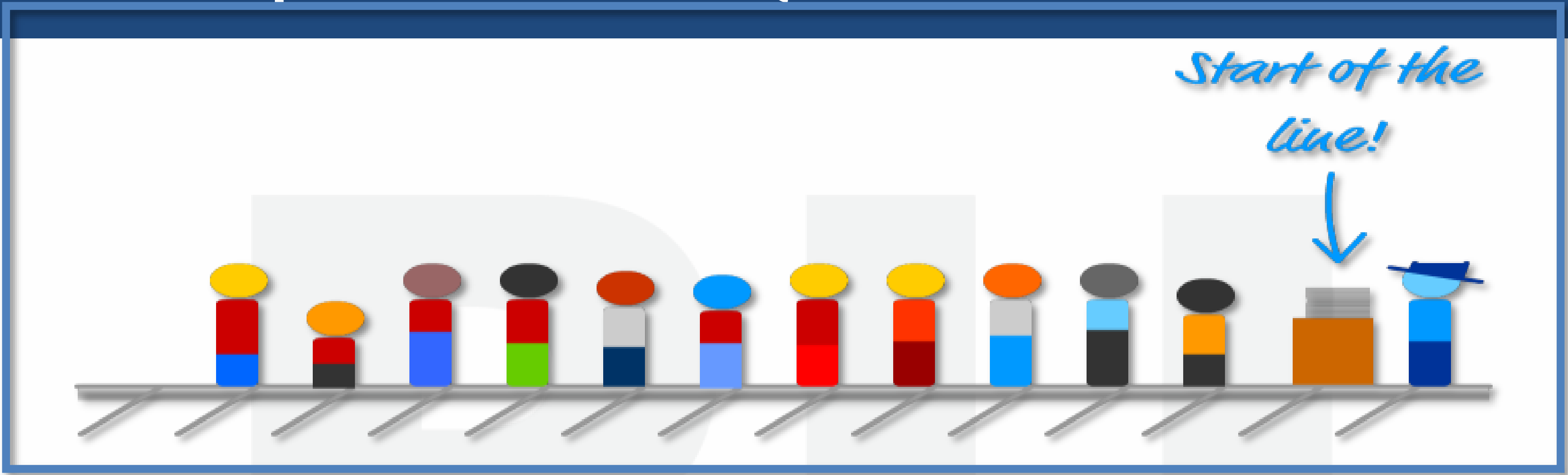
ADT(Abstract Data Type)

➢The queue abstract data type is defined by the following structure and operations:

▪Queue is a linear list of elements in which deletions can take place only at one end called the front and insertions can take place only at the other end called rear.

➢Queues are also called as FIFO (First In First Out) list.

Pictorial Representation of Queue



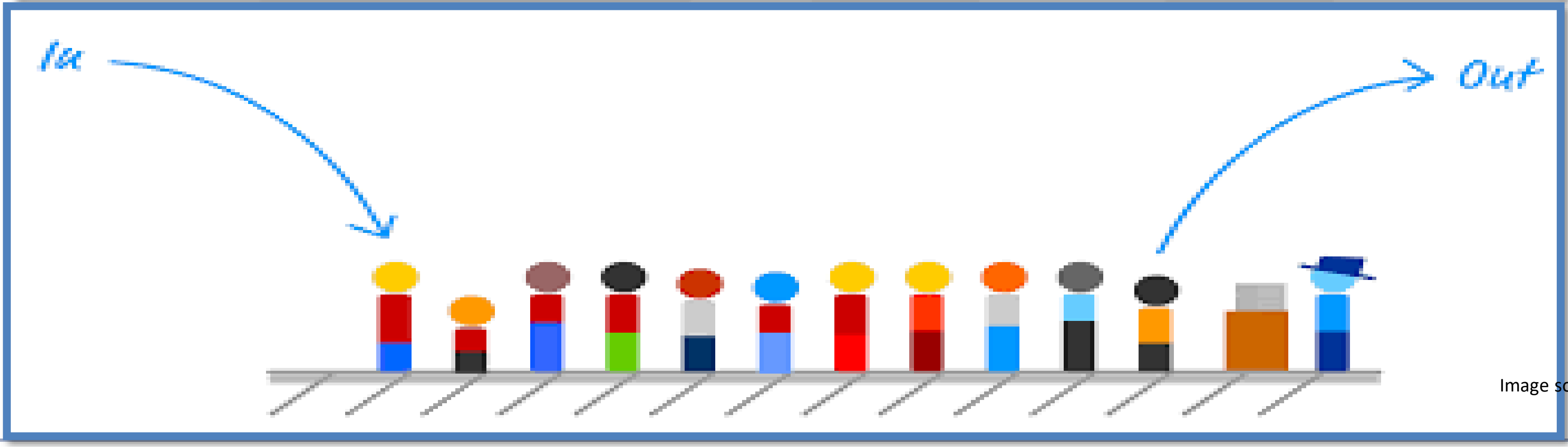


Image source : Google

Examples of Queue in day to day life

- ➢Railway reservation counter
- ➢Movie theatre ticket counter
- ➢An Escalator
- ➢A Car wash, etc.

▪Before getting the service, one has to wait in the queue. After getting the service, one leaves the queue.

▪Service is provided at one end(front/head) and people join at the other end(rear/back/tail).

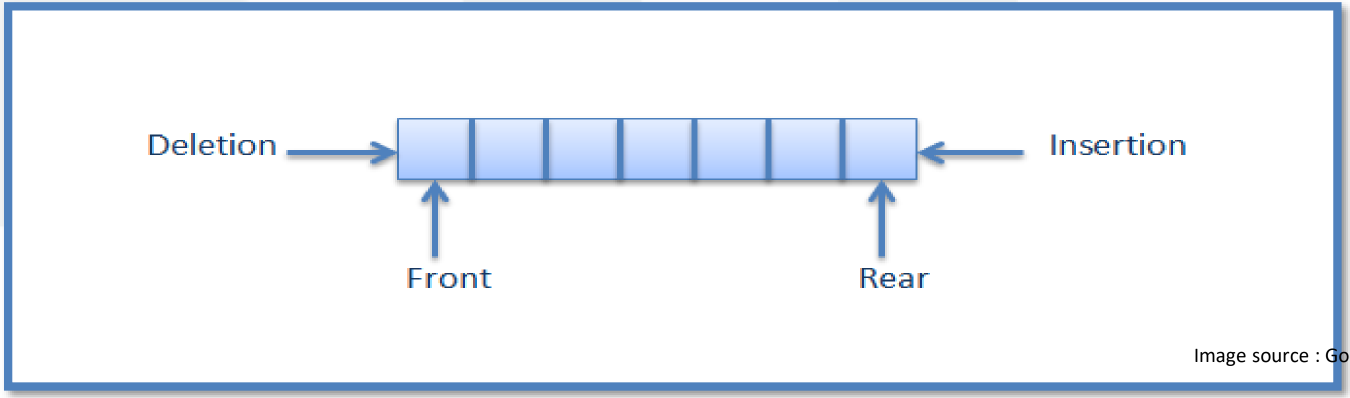


Image source : Google

Application of Queues in Computing

- ➢ Scheduling of processes (Round Robin Algorithm).
- ➢Spooling (to maintain a queue of jobs to be printed).
- ➢A queue of client processes waiting to receive the service from the server process).
- ➢An input stream.

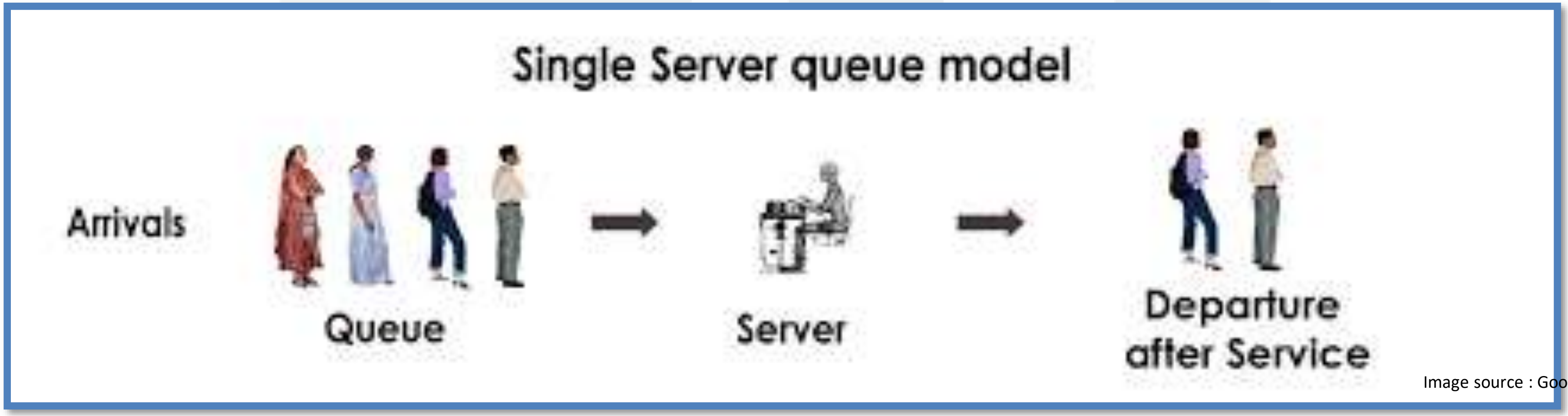


Image source : Google

Simple Queue / Linear Queue

- Simple queue defines the simple operation of queue in which insertion occurs at the rear of the list and deletion occurs at the front of the list.

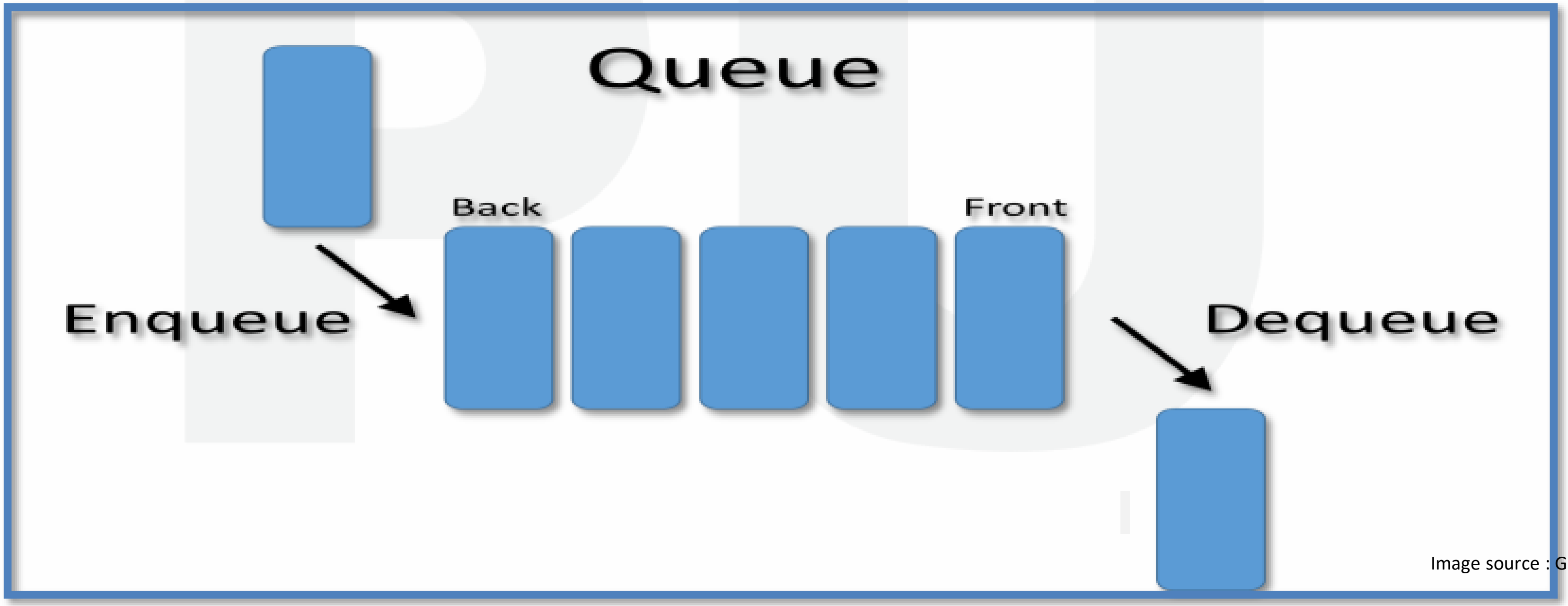
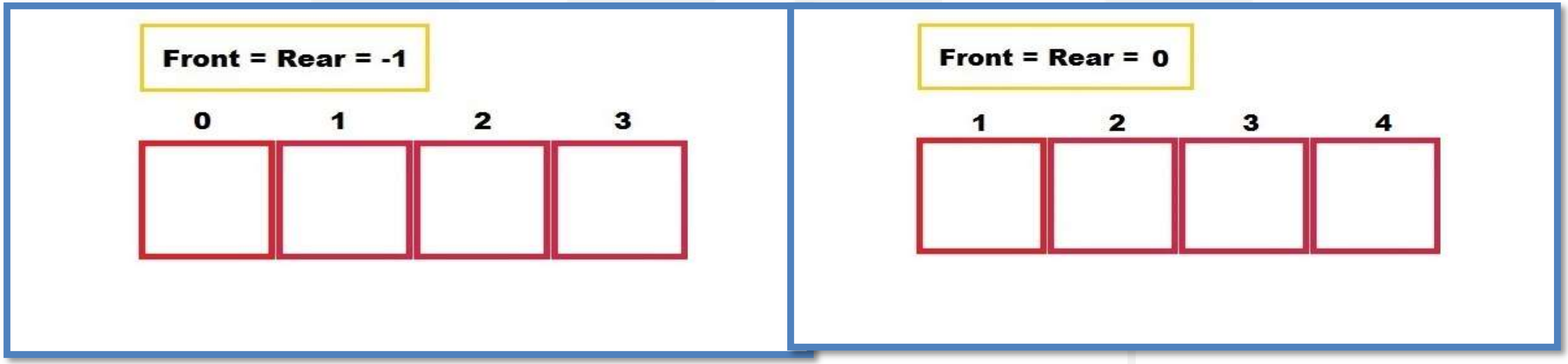


Image source : Google

Operations on a queue:

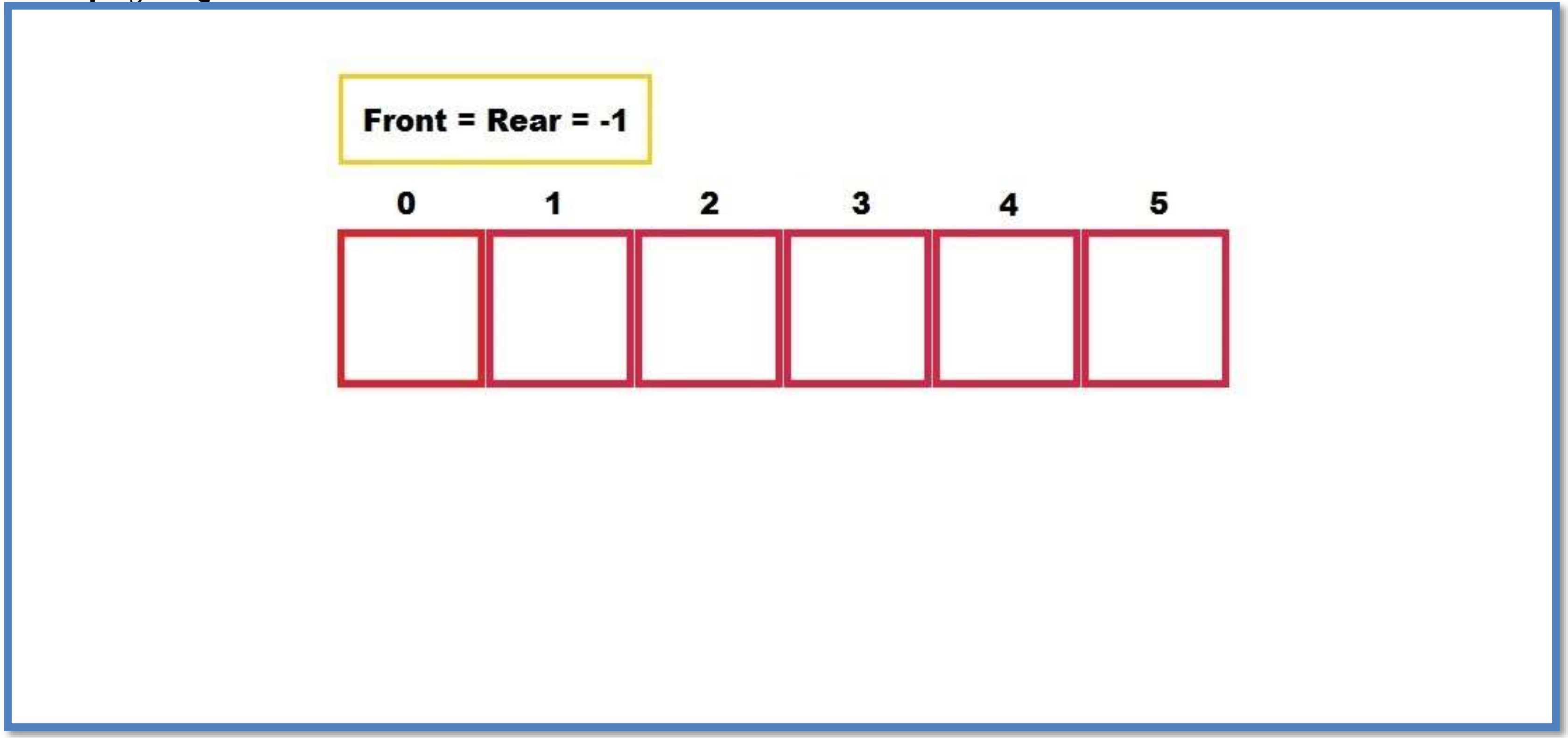
- initialize(): Initializes a queue by setting the value of front and rear to -1.
- enqueue(): Inserts an element at the rear end of queue.
- dequeue()/delete(): Deletes an element from the front end of queue.
- empty(): Returns true(1) if queue is empty and returns false(0) otherwise.
- full(): Returns true(1) if queue is full and returns false(0) otherwise.
- print(): Printing queue elements.

- ➢ We can perform initialization operation of queue by giving initial value to front and rear pointer in queue by following rules:
- ▪Initialize front and rear to 0 if the element starts from 1.
- ▪Initialize front and rear to -1 if the element starts from 0.



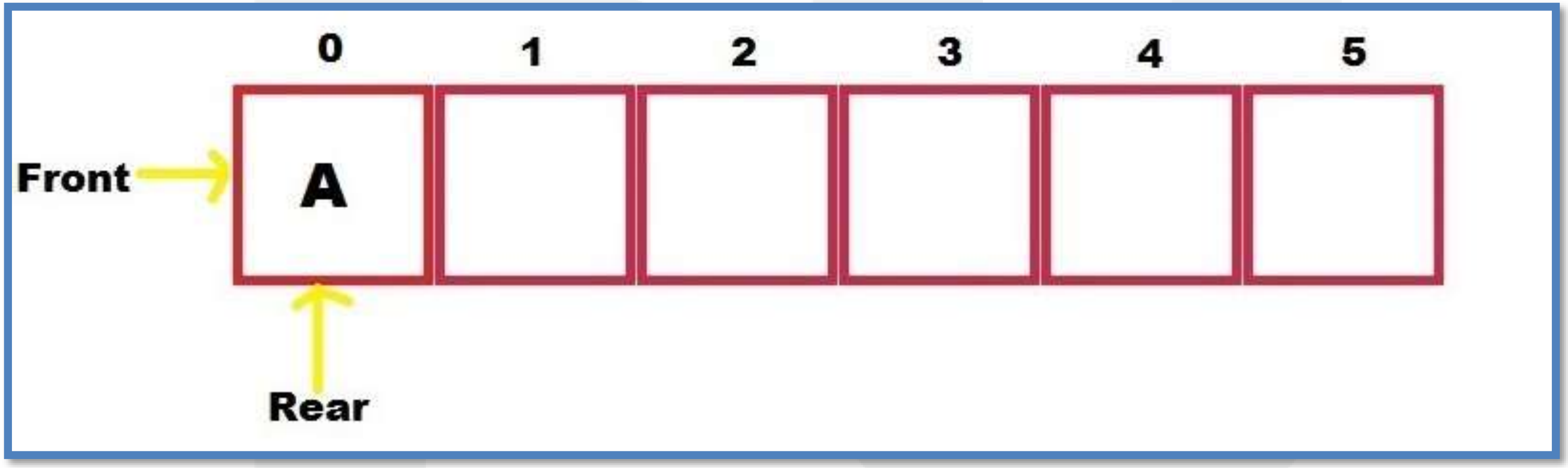
Example of Simple Queue

- Empty Queue



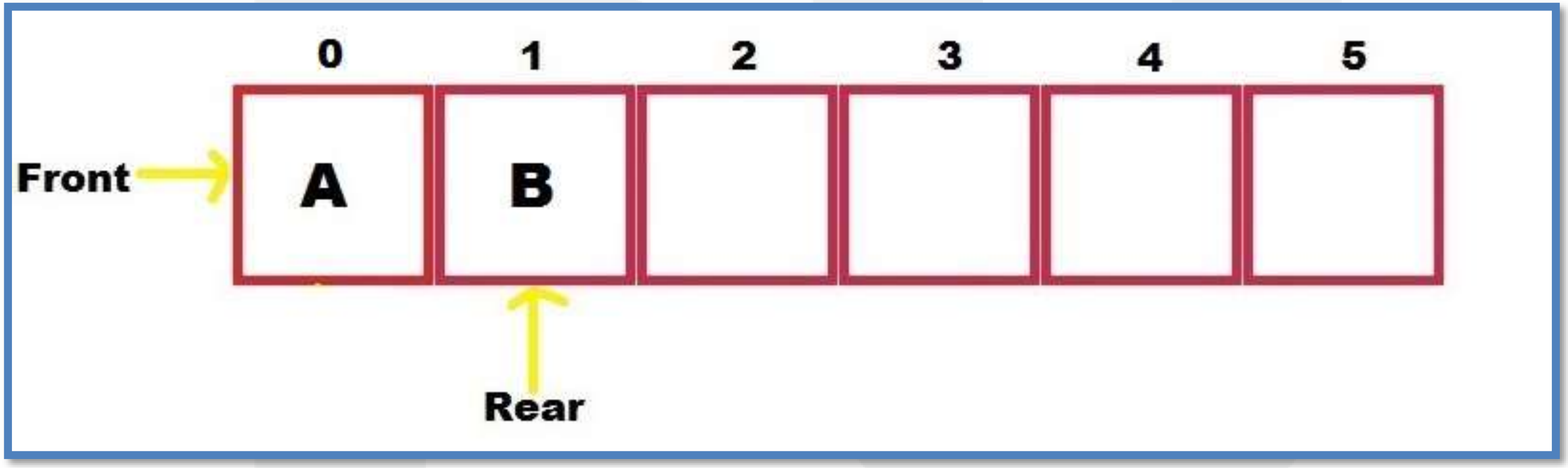
Example of Simple Queue

- Insert A



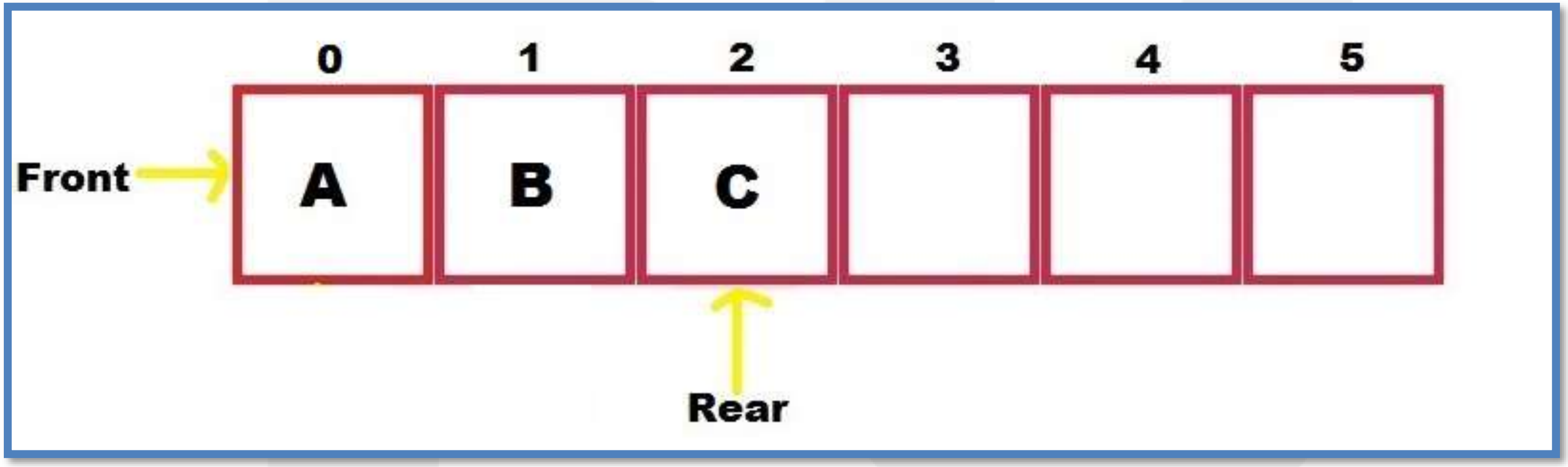
Example of Simple Queue

- Insert B



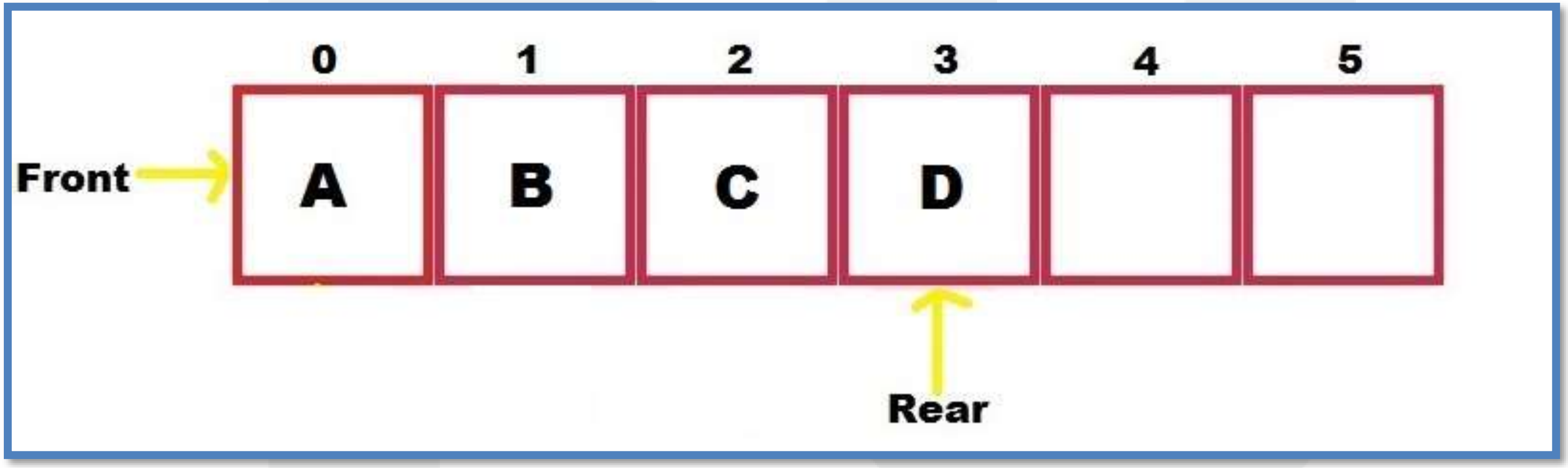
Example of Simple Queue

- Insert C



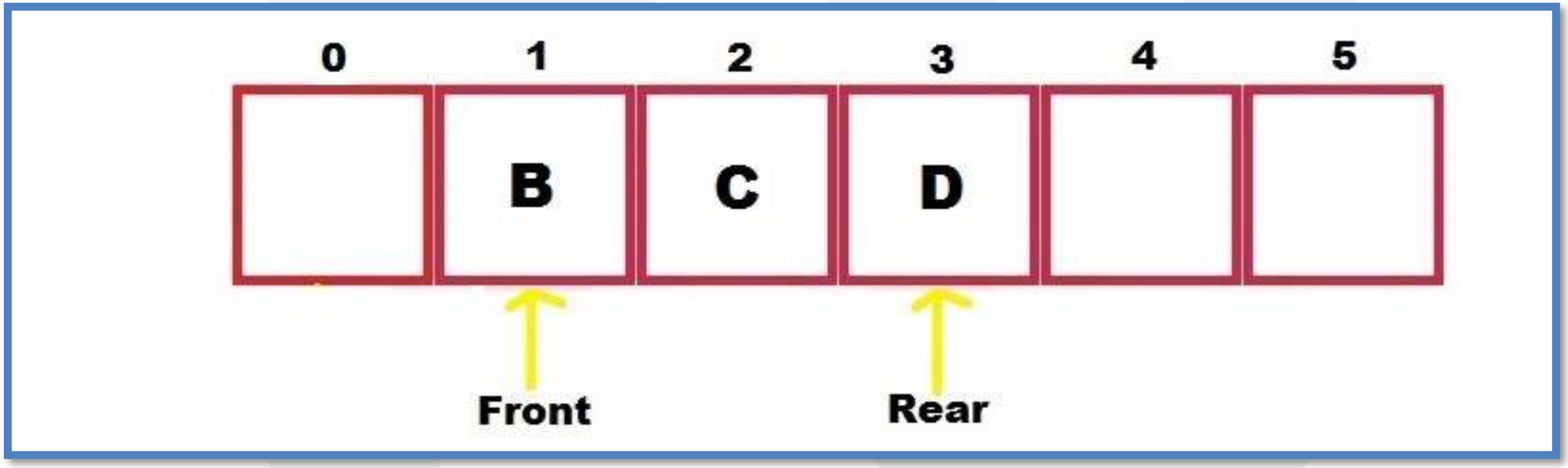
Example of Simple Queue

- Insert D



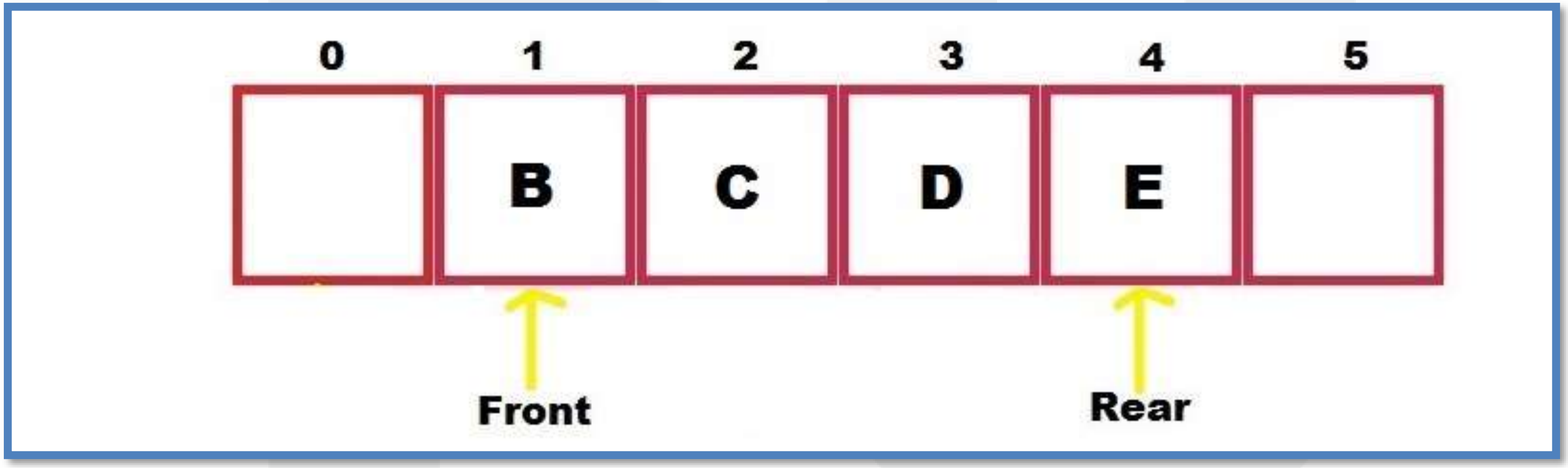
Example of Simple Queue

- Delete A



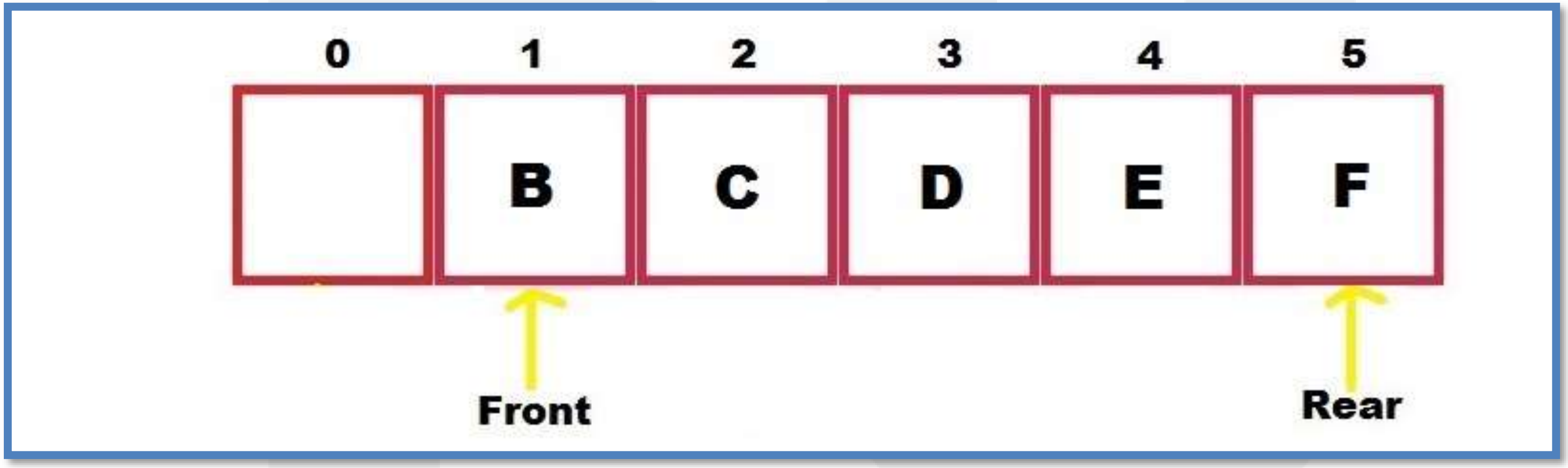
Example of Simple Queue

- Insert E



Example of Simple Queue

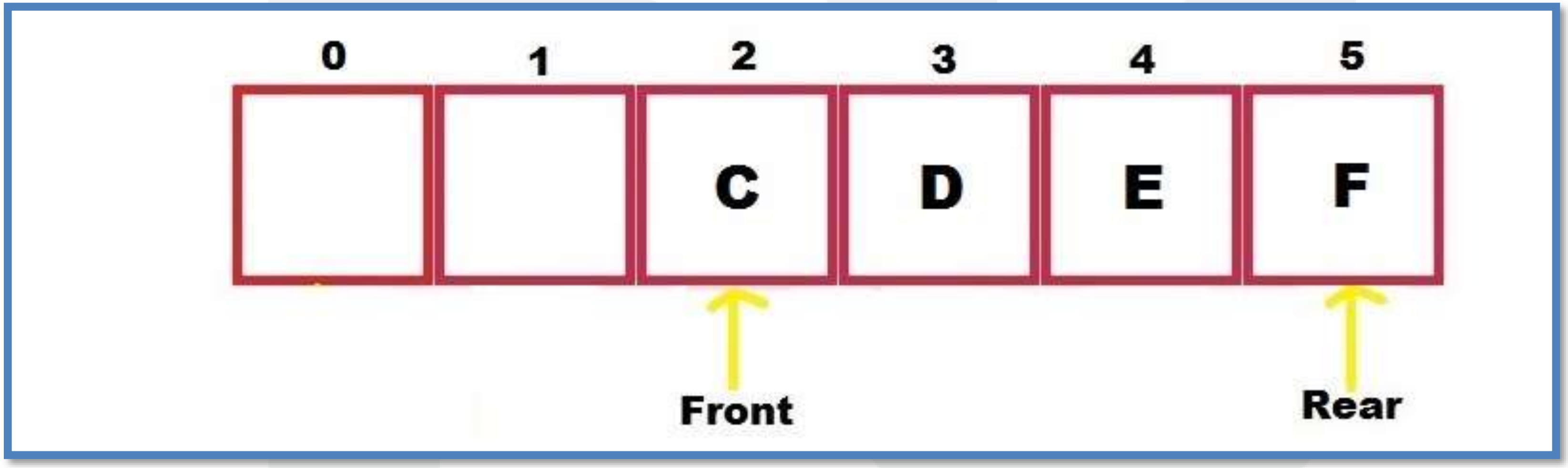
- Insert F



0	1	2	3	4	5
	B	C	D	E	F

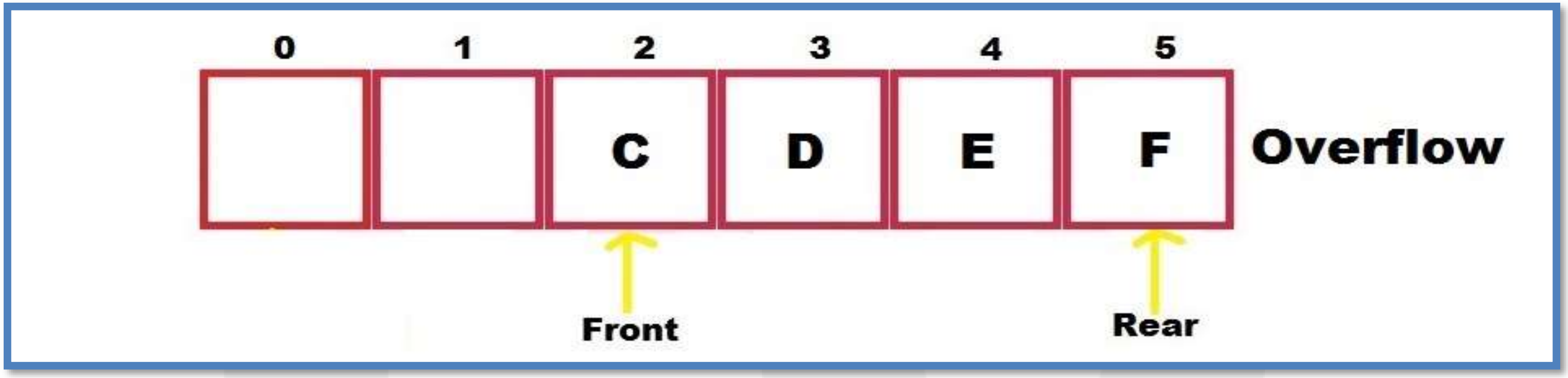
Example of Simple Queue

- Delete B



Example of Simple Queue

- Insert G



•Disadvantage of linear queue is that, in linear queue if the queue is fully occupied and the queue's initial place is empty and if want to insert other character then it is not possible even though the initial cells are empty.

Algorithms

- INSERT(Q, F, R, N, Y)

```
1. If R>=N then write: overflow and return
2. R=R+1
3. Q[R] = Y
4. If F==-1 then
       F=0 and return
```

- Here,

Q = Queue

F = Front

R = Rear

N = Size of Queue

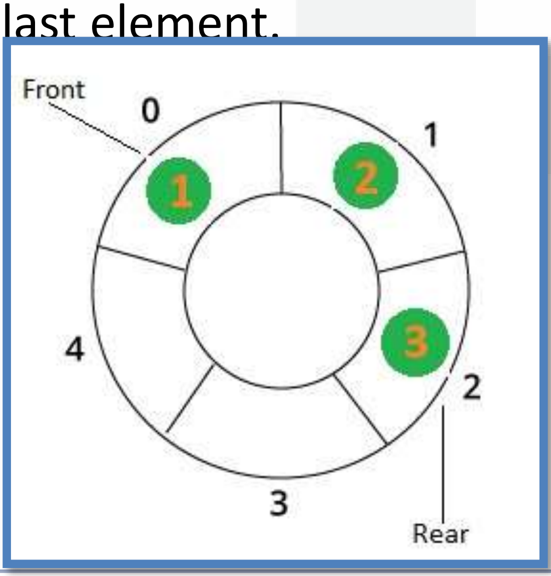
Y = Element or item that can be traced on the queue

- DELETE(Q, F, R)

```
1. If F==-1 the write: Underflow and return
2. Y = Q[F]
3. If F==R then F=-1 and R=-1 else F=F+1
4. Return [Y]
```

Circular Queue

- In a normal Queue, once queue becomes full, we can not insert the next element even if there is a space in front of queue.
- The solution to this is, whenever the rear end gets to the end of the array, it should be wrapped around to the beginning of the array.
- Now the array can be viewed as a circle where the first position will follow the last element.



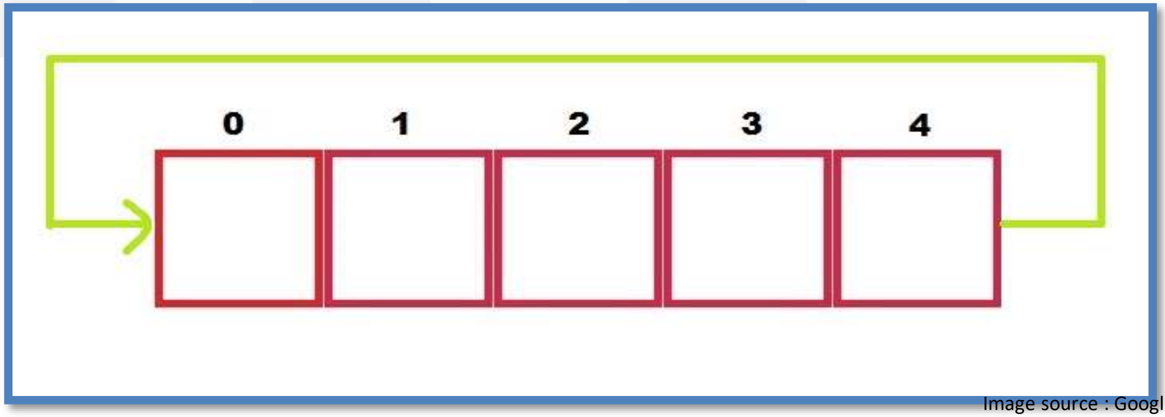
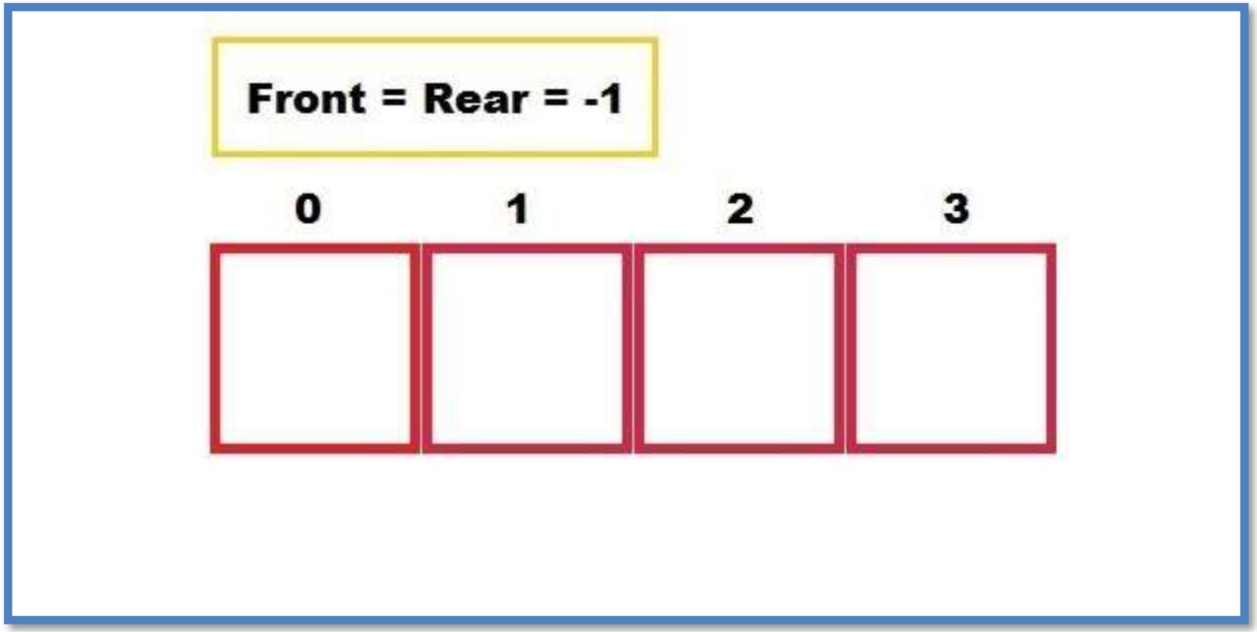


Image source : Google

Example of Circular Queue

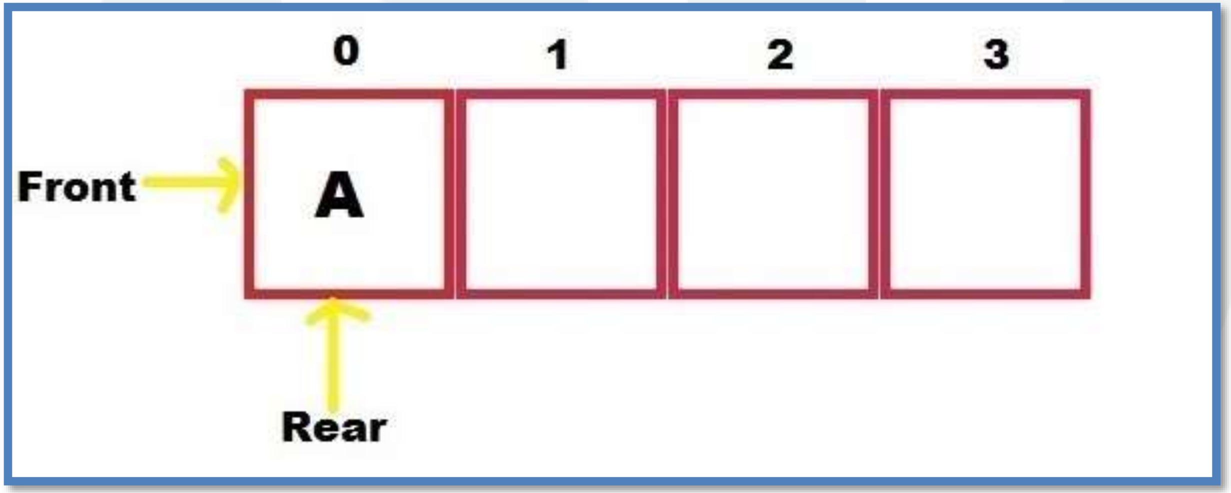
- Empty



Front = Rear = -1

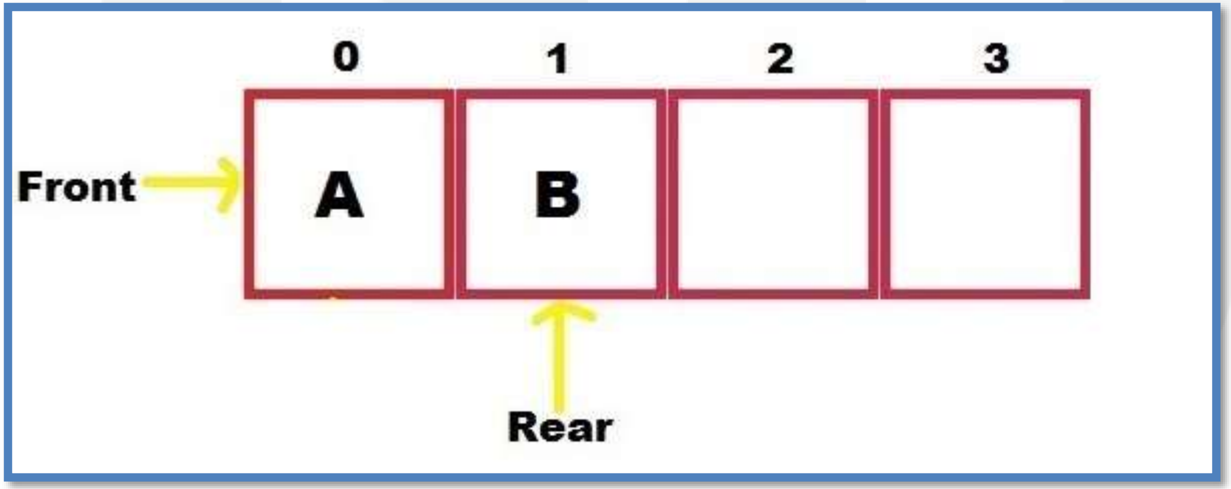
Example of Circular Queue

- Insert A



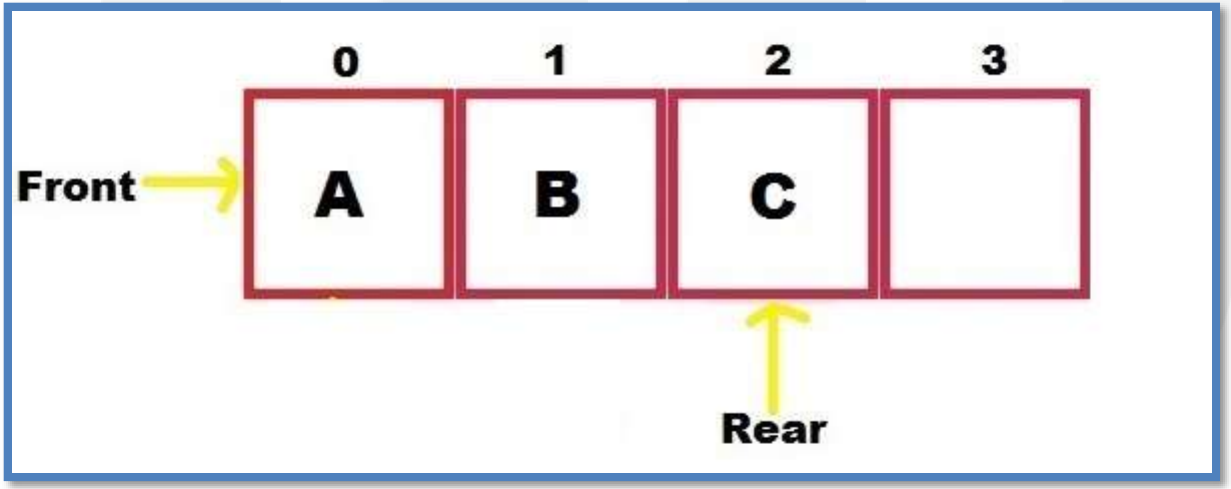
Example of Circular Queue

- Insert B



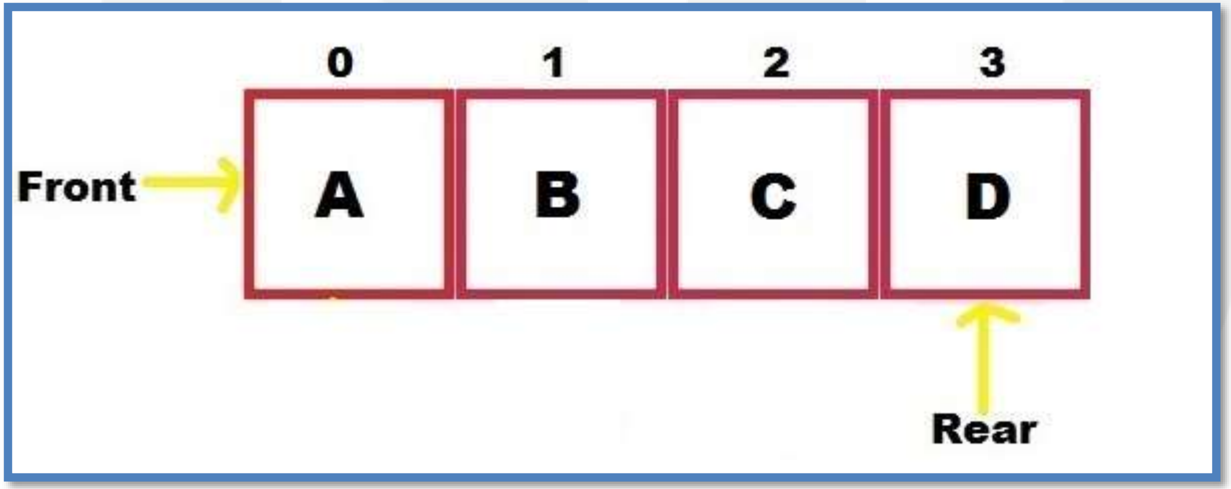
Example of Circular Queue

- Insert C



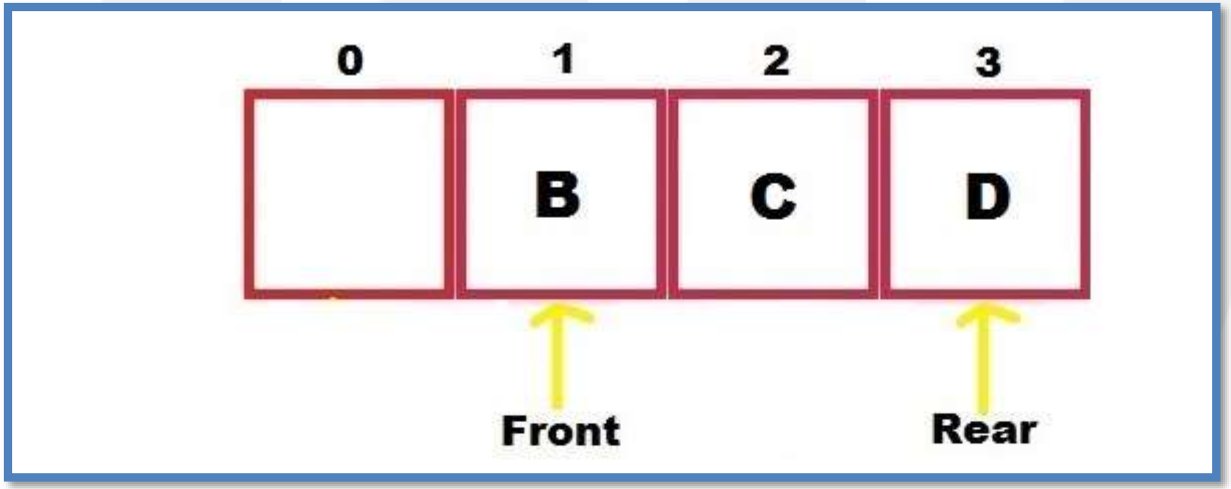
Example of Circular Queue

- Insert D



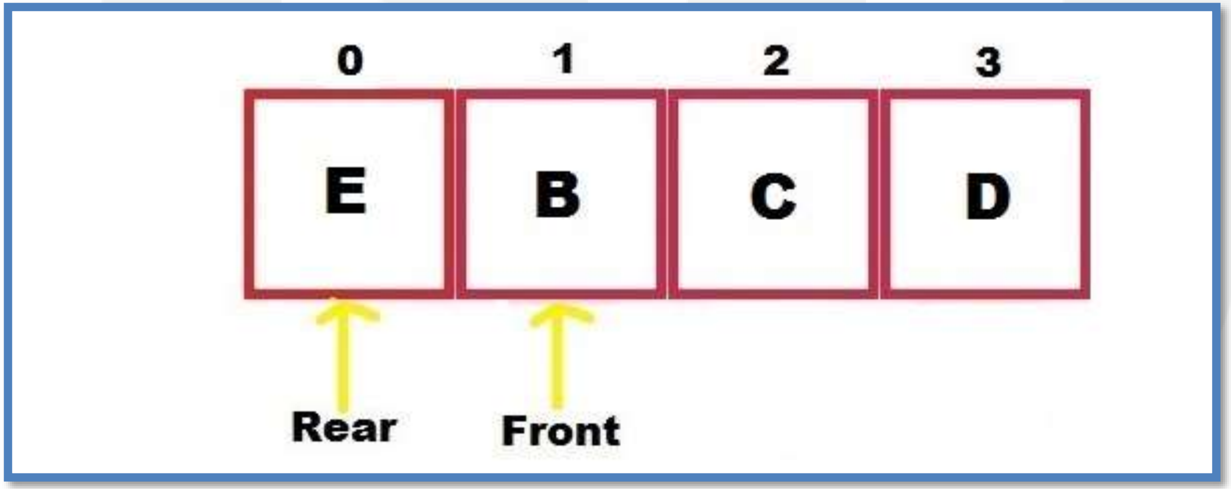
Example of Circular Queue

- Delete A



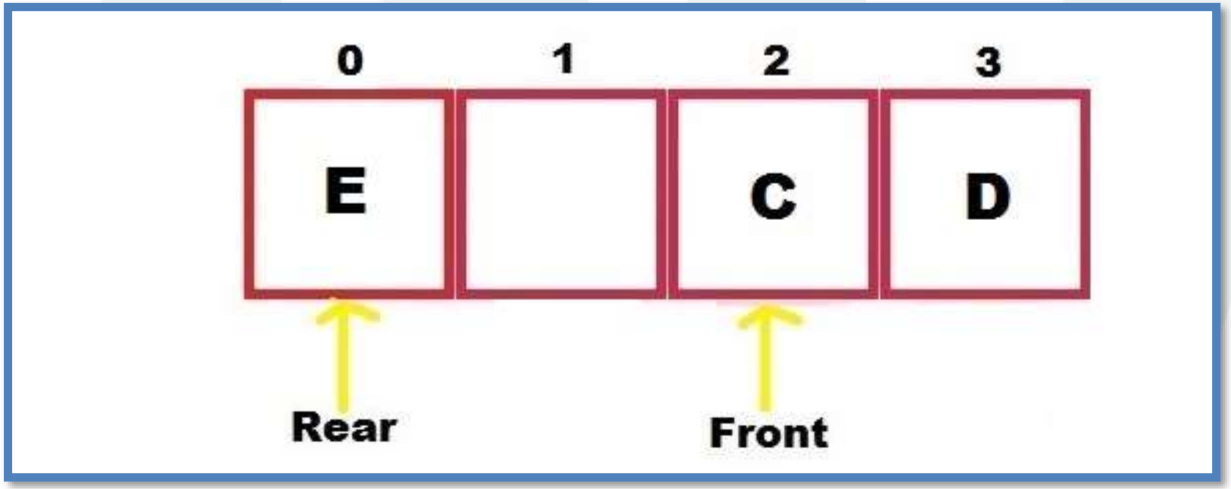
Example of Circular Queue

- Insert E



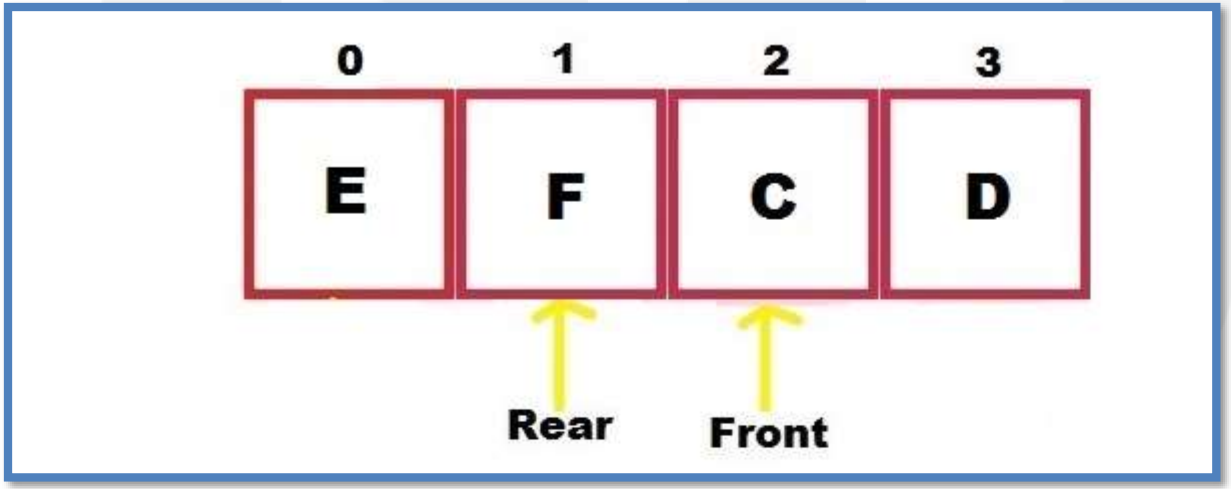
Example of Circular Queue

- Delete B



Example of Circular Queue

- Insert F



0 1 2 3

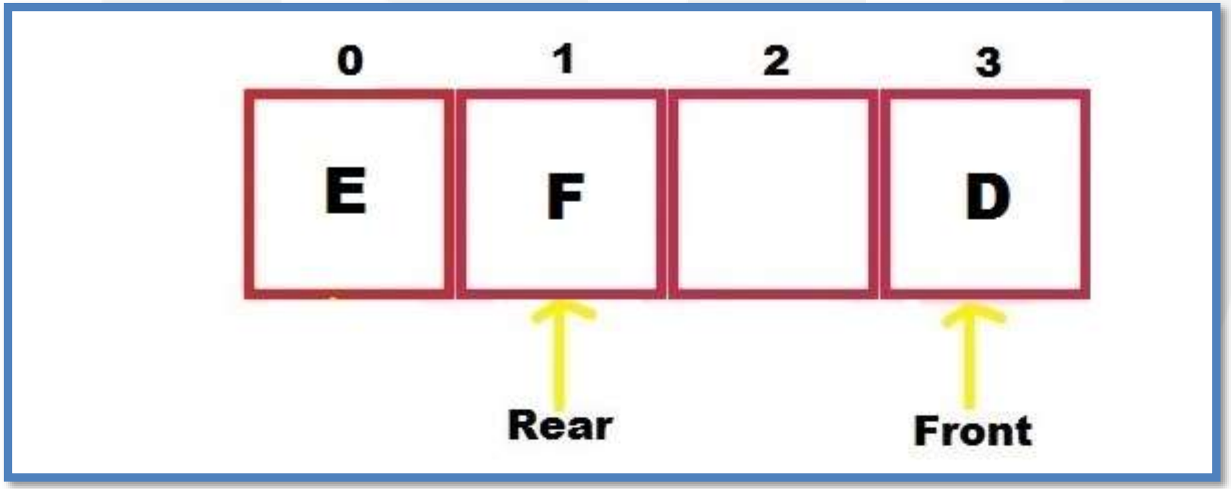
E F C D

Rear

Front

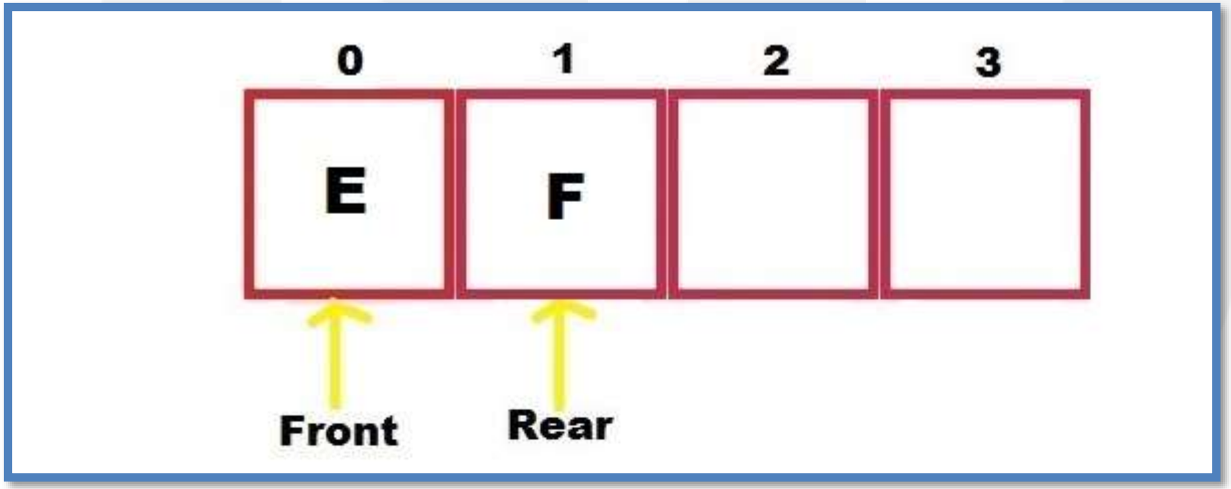
Example of Circular Queue

- Delete C



Example of Circular Queue

- Delete D



Algorithms

- CQINSERT(Q, F, R, N, Y)

```
1.  If R==N then R=0 else R=R+1
2.  If R=F-1 then write: Overflow and return
3.  Q[R] = Y
4.  If F==-1 then
        F=0 and return
```

- Here,

Q = Queue
F = Front
R = Rear
N = Size of Queue
Y = Element or item that can be traced on the queue

- CQDELETE(Q, F, R, N)

```
1.  If F==-1 the write: Underflow and return
2.  Y = Q[F]
3.  If F==R then F=-1 and R=-1 and return [Y]
4.  If F==N then F=0 else F=F+1 and return[Y]
```

Dequeue

- Dequeue is the short notation used for double ended queue.
- The usual practice is: for insertion of element we use one end called rear and for deletion of elements we use front end.
- In dequeue we make use of both the ends for insertion of the elements as well as we can use both the ends for deletion of the elements.
- In simple words, it is a linear list in which elements can be added or removed from either ends but not in the middle.

Structure of Dequeue

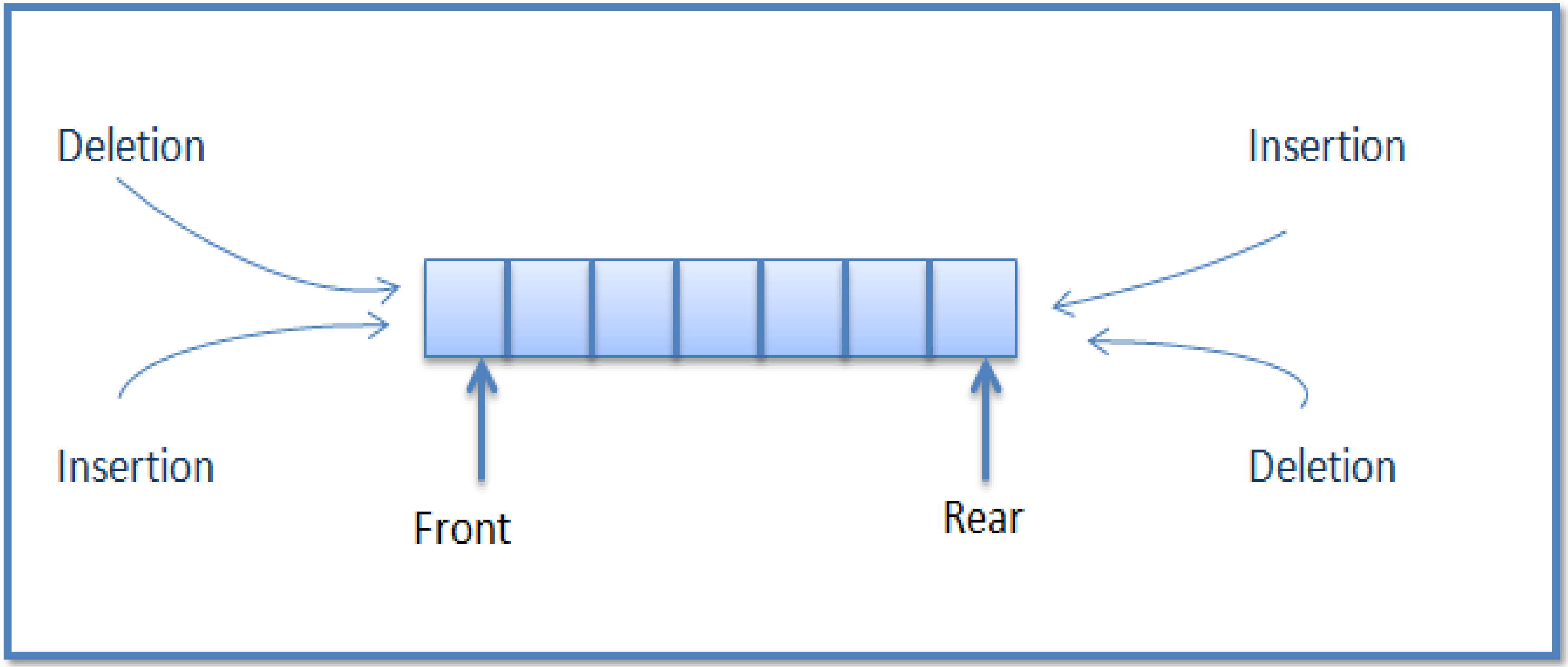
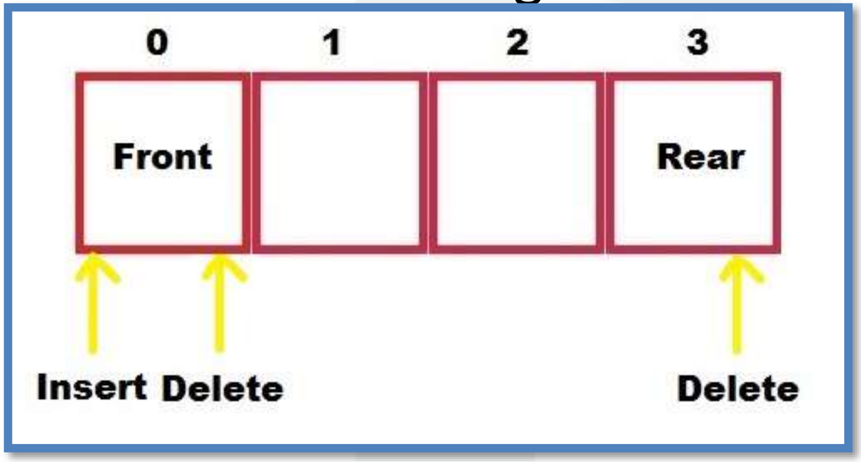


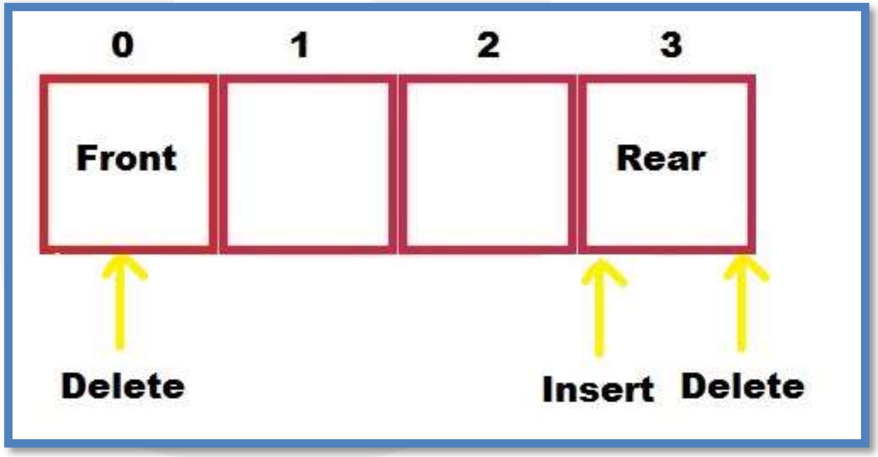
Image source : Google

Classification of Dequeue

➢Input Restricted:

▪When we use only one end for inserting an element and both the ends for deleting an element.

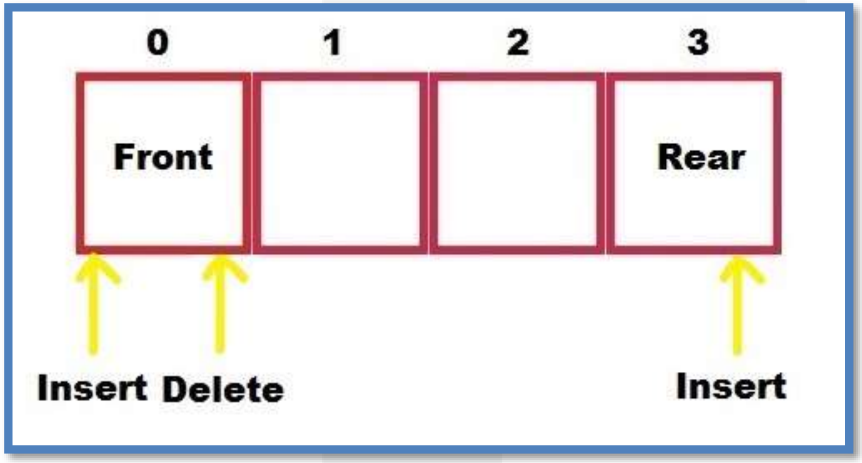


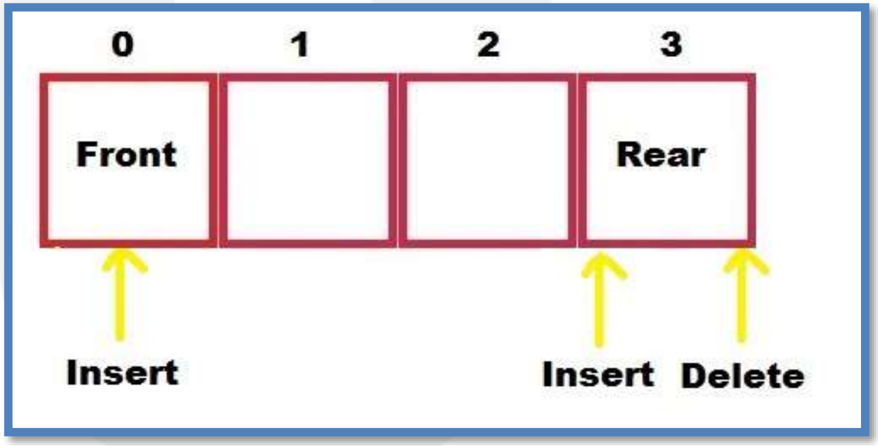


Classification of Dequeue

- Output Restricted

- When we use only one end for deleting an element and both ends for insertion of an element.





Methods to implement Dequeue

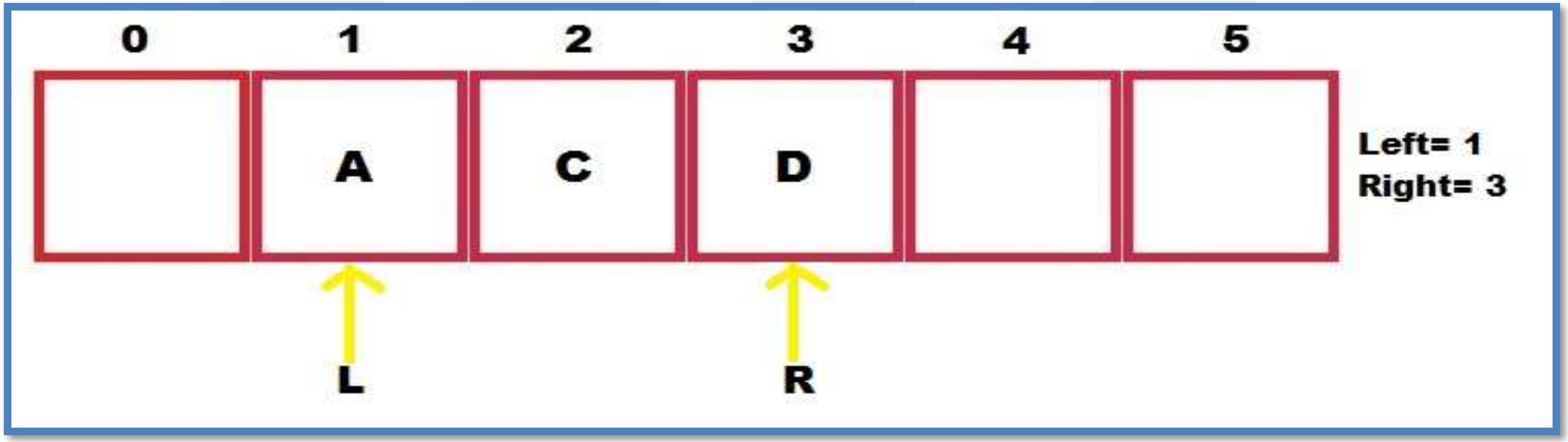
- Using circular array
- Using singly linked list
- Using doubly linked list
- Using singly circular linked list
- Using doubly circular linked list

Operations on dequeue

- initialize():Make the empty queue.
- empty(): To check whether queue is empty or not.
- full(): To check whether queue is full or not.
- enqueueF(): To insert an element to the front end of the queue.
- enqueueR(): To insert an element to the rear end of the queue.
- dequeueF(): Delete the front element.
- dequeueR(): Delete the rear element.
- print(): Print elements of queue.

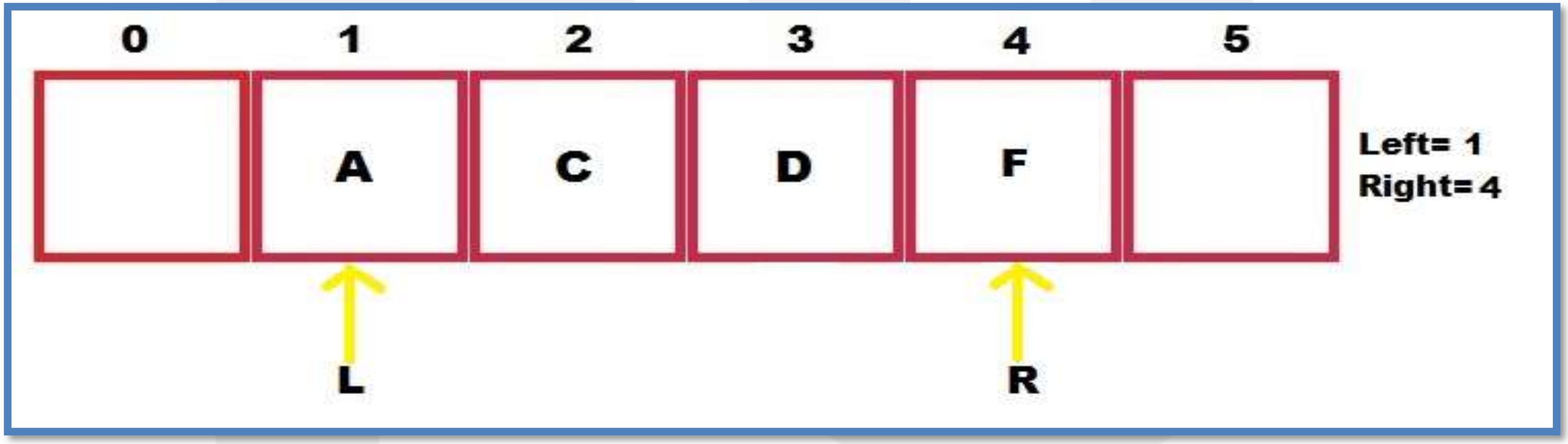
Example of Dequeue

- Consider the following dequeue of characters where dequeue is a circular array which is allocated 6 memory cells.
- Left=1 and Right =3.
- Dequeue: _,A,C,D,_,_.



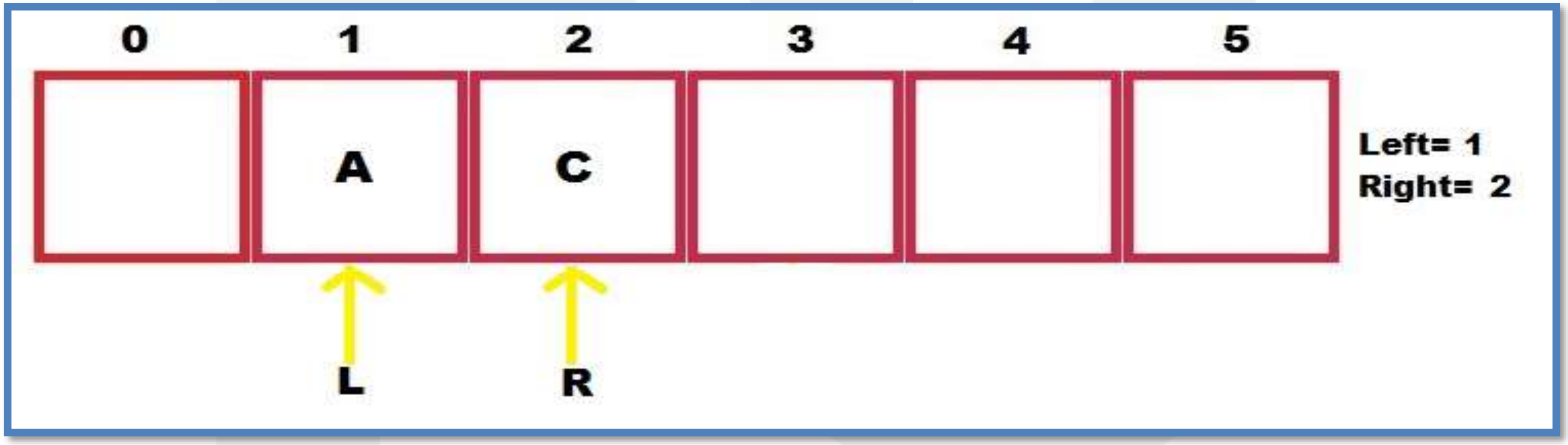
Example of Dequeue

- F is added to right of dequeue



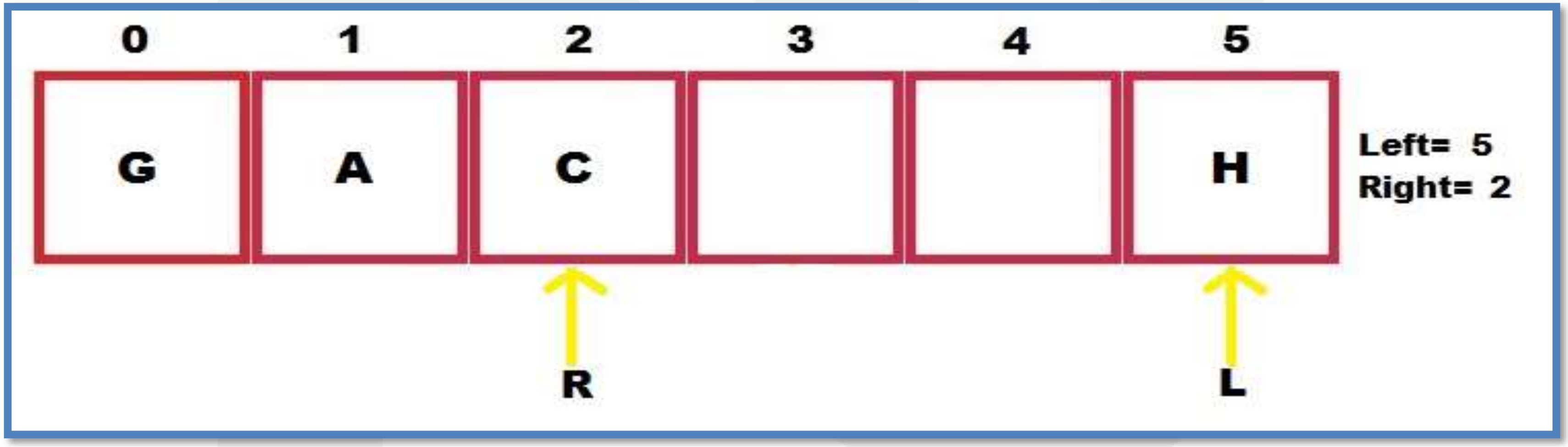
Example of Dequeue

- 2 letters are deleted from right.



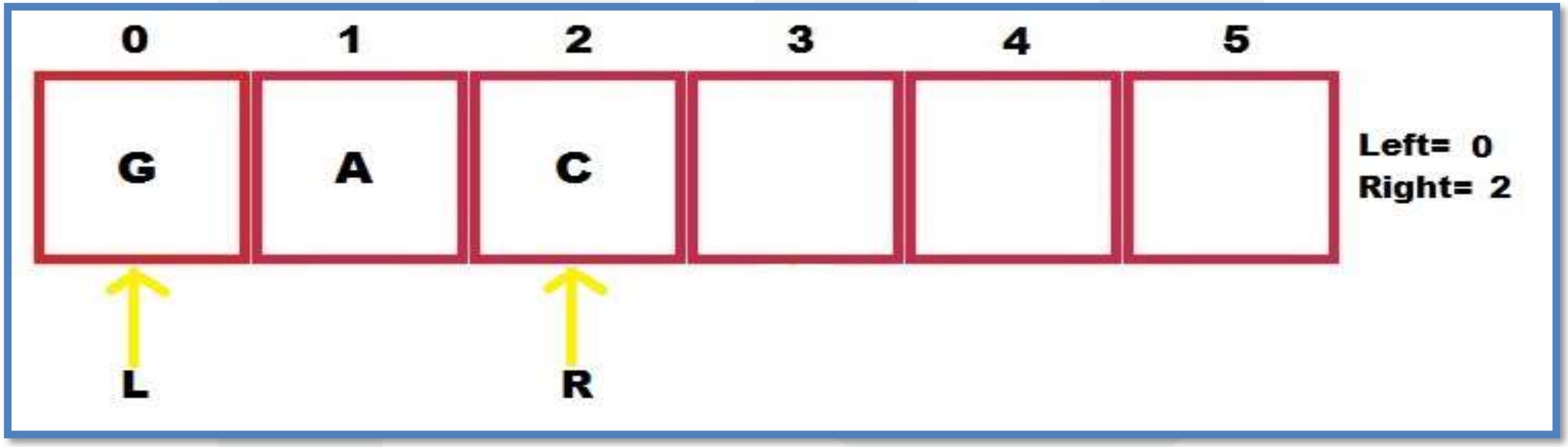
Example of Dequeue

- ➢G, H are added to left of dequeue.



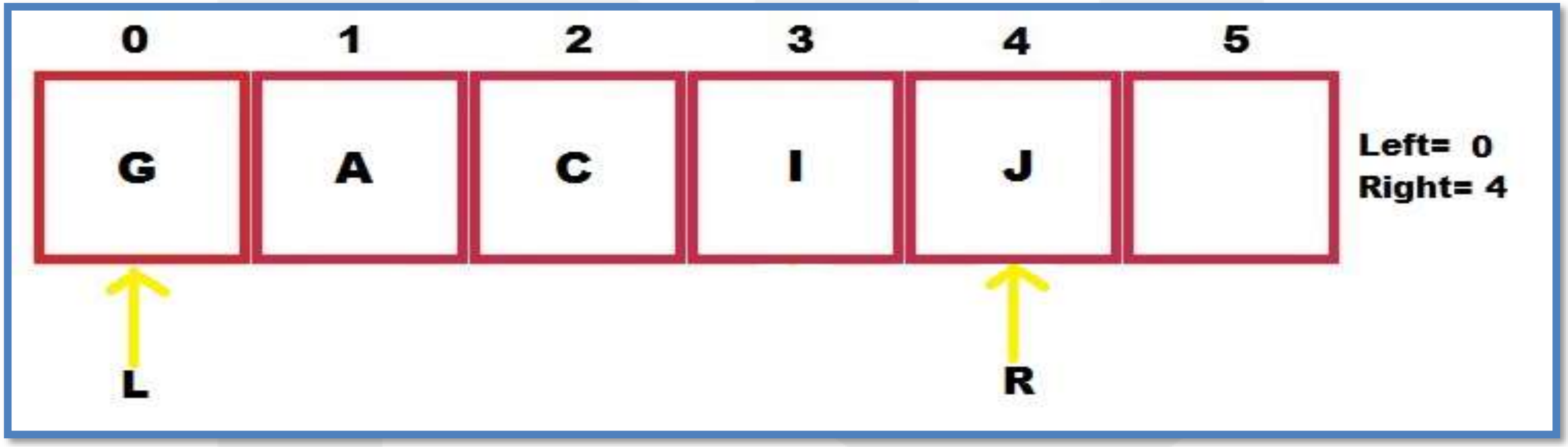
Example of Dequeue

- 1 letter is deleted from left



Example of Dequeue

- ➢I, J are added to right of dequeue

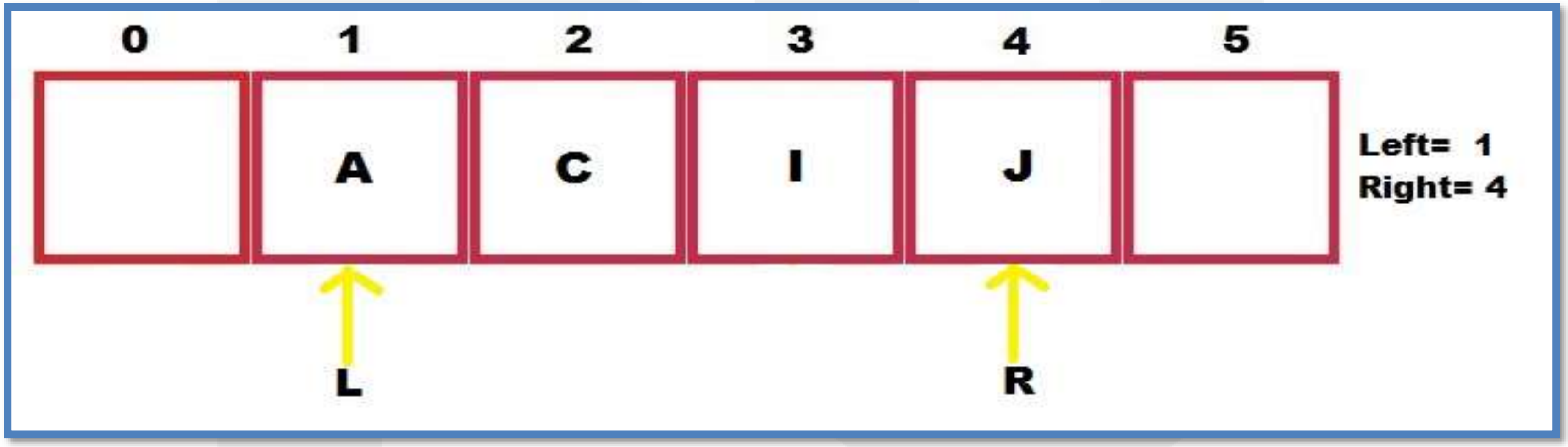


0	1	2	3	4	5
G	A	C	I	J	

Left= 0
Right= 4

Example of Dequeue

- 1 letter is deleted from left



0	1	2	3	4	5
	A	C	I	J	

Left= 1
Right= 4

Priority Queue

- ➢ It is a collection of elements such that each element has been assigned a priority and the order in which elements are processed and deleted follows following rules:
- ▪ An element with higher priority is processed before any element with lower priority.
- ▪ Two elements with same priority are processed on FCFS (First Come First Serve) basis.

Operations on priority queue

- initialize(): Make the empty queue.
- full(): Check if the queue is full or not.
- empty(): Check if the queue is empty or not.
- enqueue(): Insert an element as per its priority.
- dequeue(): Delete the element in front(as front element has the highest priority).
- print(): Print queue elements.

Types/Representation of priority queue

1) One way list representation of a priority:

- Each node in a list will contain 3 items of an information: INFO(), PRIORITY_NO() and a link.
- A node X precedes node Y in the list where:

  1. X has higher priority than Y.

  2. When both have same priority but X was added in the list before Y.

Algorithm

INSERT

- It adds an item with priority number ‘n’ to a priority queue which is maintained in memory as one-way list.
- Traverse one- way list until you find a node X whose priority number exceeds ‘n’.
- Insert ITEM in front of node X.
- If no such element is found, insert ITEM as the last element of the list.

Algorithm

DELETE

- ➢ It deletes and processes the first element in a priority queue which appears in memory as one-way list.
- Set ITEM=INFO[START]
- Delete first node from the list
- Process ITEM
- Exit

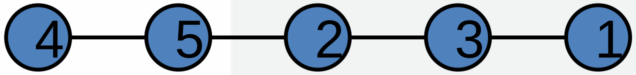
Types/Representation of priority queue

2) Array Representation of Priority Queue:

- Another way to maintain priority queue is to use a separate queue for each level of priority.
- Each such queue will appear in its own circular array and must have its own pair of pointers(rear and front).
- In each queue is allocated same amount of space, a 2D array of queue can be used instead of linear arrays.

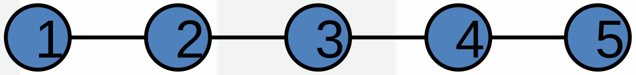
Implementation of priority queue

- Implementation with an unsorted list



- Performance:
  - insert takes ***O***(1) time since we can insert the item at the beginning or end of the sequence
  - remove take ***O***(***n***) time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list



- Performance:
  - insert takes ***O***(***n***) time since we have to find the place where to insert the item
  - remove and take ***O***(1) time, since the smallest key is at the beginning

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in