Data Structures and Algorithms(CSE2001)

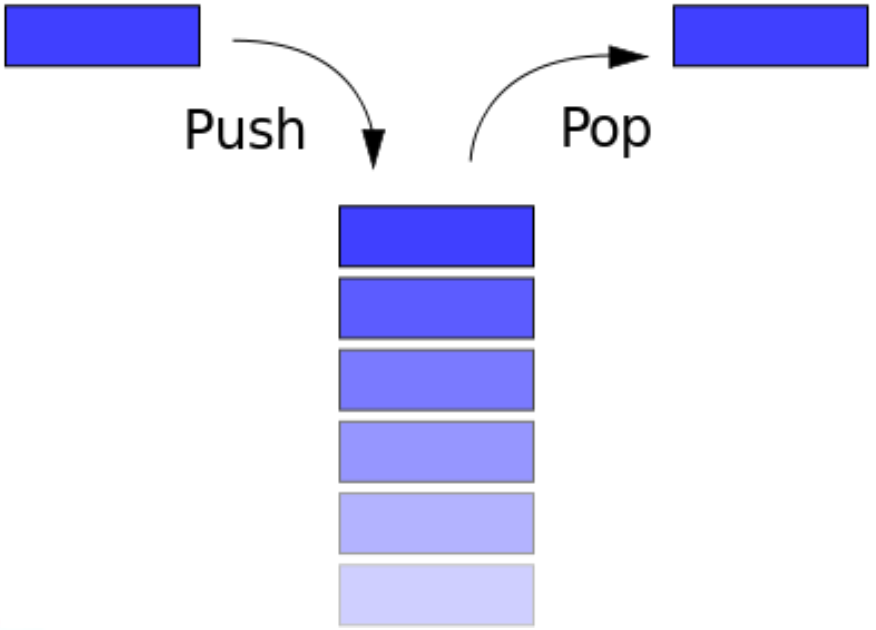
MODULE 1

Stack Linear Data Structure

Stack

What is a Stack ?

- A stack is a particular kind of abstract data type or collection in which the fundamental (or only) operations on the collection are the addition of an entity to the collection, known as push and removal of an entity, known as pop.



Examples

In Simple Words

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria
    - A stack of Books
    - A stack of Dosa
    - A stack of Money

Basic Principle or representation

- The relation between the push and pop operations is such that the stack is a Last-In-First-Out (LIFO) data structure.
- In a LIFO data structure, the last element added to the structure must be the first one to be removed.

Stack ADT

- Stack ADT operations
  - Create an empty stack
  - Destroy a stack
  - Determine whether a stack is empty
  - Add a new item
  - Remove the item that was added most recently
  - Retrieve the item that was added most recently

Operations on Stack

```
bool isEmpty() const;

    // Determines whether a stack is empty.

    // Precondition: None.

    // Postcondition: Returns true if the
    // stack is empty; otherwise returns
    // false.
```

Operations on Stack

```
bool push(StackItemType newItem);

    // Adds an item to the top of a stack.

    // Precondition: newItem is the item to
    // be added.

    // Postcondition: If the insertion is
    // successful, newItem is on the top of
    // the stack.
```

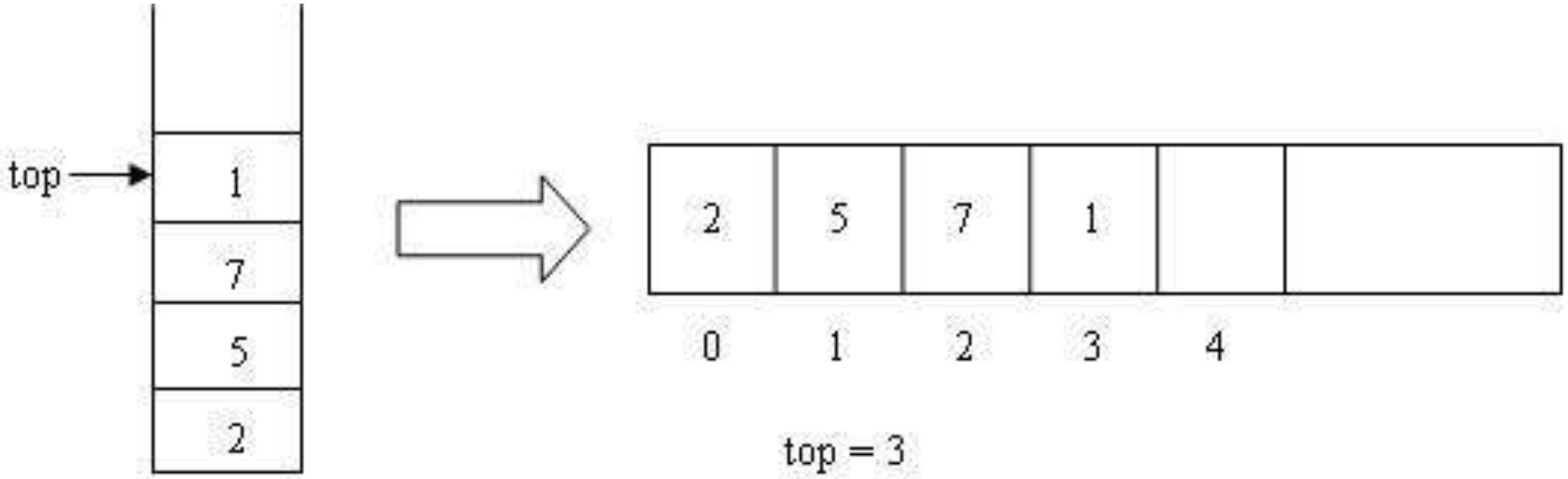
Operations on Stack

```
bool pop(StackItemType& stackTop);

    // Retrieves and removes the top of a stack

    // Precondition: None.

    // Postcondition: If the stack is not empty,
    // stackTop contains the item that was added
    // most recently and the item is removed.
    // However, if the stack is empty, deletion
    // is impossible and stackTop is unchanged.
```

Operations on Stack

```
bool getTop(StackItemType& stackTop) const;

    // Retrieves the top of a stack.

    // Precondition: None.

    // Postcondition: If the stack is not empty,
    // stackTop contains the item that was added
    // most recently. However, if the stack is
    // empty, the operation fails and stackTop
    // is unchanged. The stack is unchanged.
```

Implementation Using Arrays



The Stack Class - Public Contents

```
class  Stack
{
    public:
        Stack() { size = 10; current = -1;}
        int pop(){ return A[current--];}
        void push(int x){A[++current] = x;}
        int top(){ return A[current];}
        int isEmpty(){return ( current == -1 );}
        int isFull(){ return ( current == size-1);}
```

The Stack Class - Private Contents

```
private:

        int   object;     // The data element
        int   current;    // Index of the array
        int   size;       // max size of the array
        int   A[10];      // Array of 10 elements
};
```

Stack - a Very Simple Application

- The simplest application of a stack is to reverse a word.
- You push a given word to stack - letter by letter - and then pop letters from the stack.

Applications of Stack

- Recursion
- Decimal to Binary Conversion
- Infix to Postfix Conversion and Evaluation of Postfix Expressions
- Rearranging Railroad Cars
- Quick Sort
- Function Calls
- Undo button in various software.

outputInBinary - Algorithm

```
function outputInBinary(Integer n)
    Stack s = new Stack
    while n > 0 do
        Integer bit = n modulo 2
        s.push(bit)
        if s is full then
            return error
        end if
        n = floor(n / 2)
    end while
```

```
while s is not empty do
        output(s.pop())
    end while
end function
```

Algebraic Expressions

- 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 +

A complicated example

- Infix expressions:

```
a + b * c + ( d * e + f ) * g
```

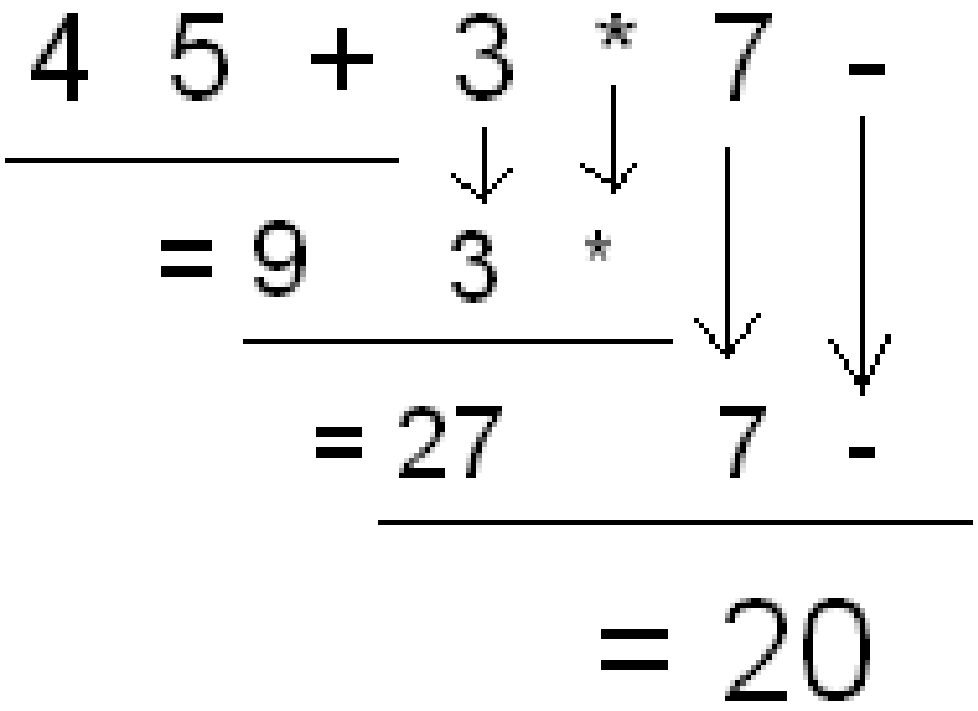
- Postfix expressions:

```
a b c * + d e * f + g * +
```

- Prefix expressions:

```
+ + a * b c * + * d e f g
```

Evaluation of Postfix Expression

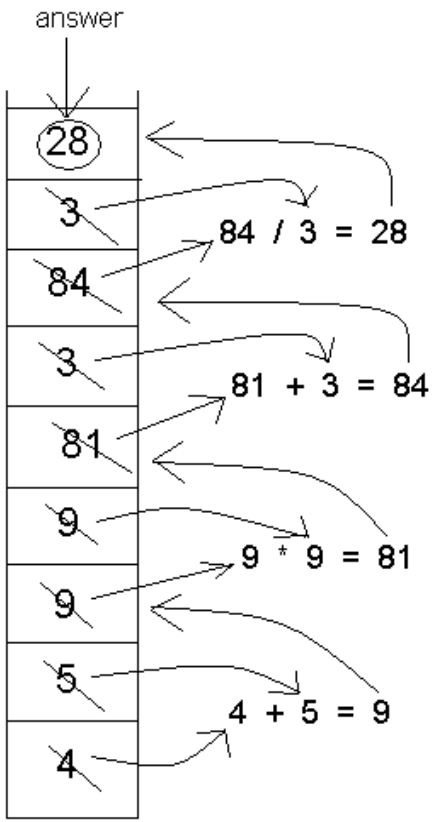


Evaluation of Postfix Expression

- The calculation: 1 + 2 * 4 + 3 can be written down like this in postfix notation with the advantage of no precedence rules and parentheses needed
- 1 2 4 * + 3 +
- The expression is evaluated from the left to right using a stack:
  - when encountering an operand: push it
  - when encountering an operator: pop two operands, evaluate the result and push it.

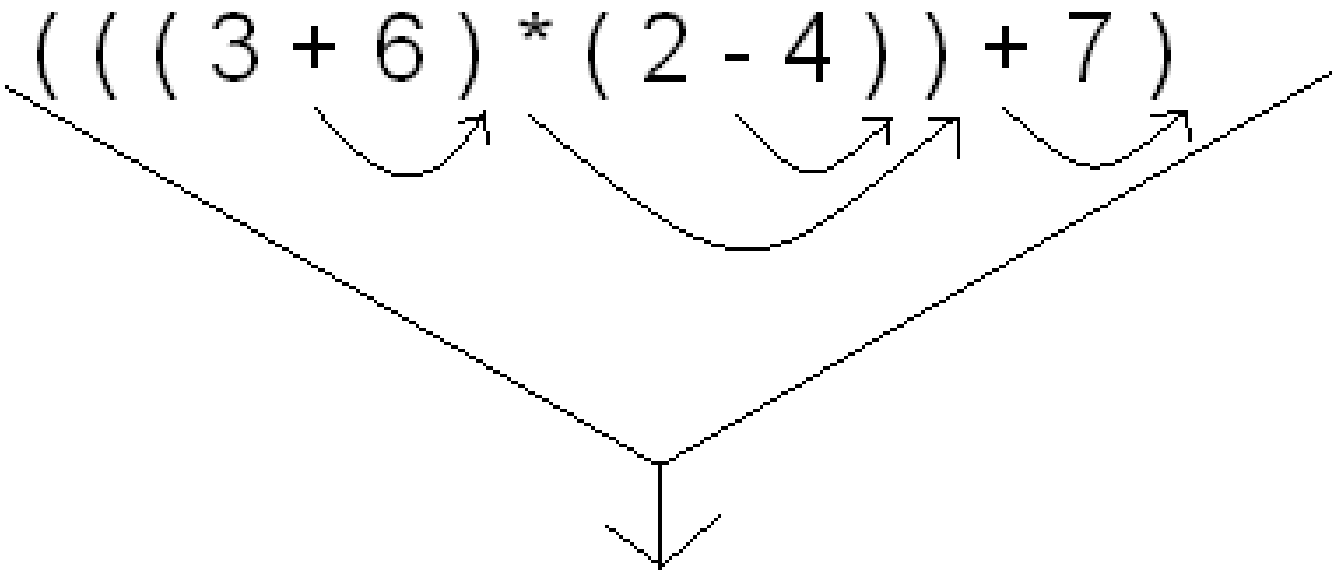
Evaluation of Postfix Expression

4 5 + 9 * 3 + 3 /



1) Push 4 onto the Stack (1 item in Stack)
2) Push 5 onto the Stack (2 items in Stack)
3) See the +, so pop 2 operands off (4 and 5) (0 items in Stack)
4) Push their result (9) onto the Stack (1 item in Stack)
5) Push 9 onto the Stack (2 items in Stack)
6) See the *, so pop 2 operands off (9 and 9) (0 items in Stack)
7) Push their result (81) onto the Stack (1 item in Stack)
8) Push 3 onto the Stack (2 items in Stack)
9) See the +, so pop 2 operands off (81 and 3) (0 items in Stack)
10) Push their result (84) onto the Stack (1 item in Stack)
11) Push 3 onto the Stack (2 items in Stack)
12) See the /, so pop 2 operands off (84 and 3) (0 items in Stack)
13) Push their result (28) onto the Stack (1 item in Stack)
14) See that there is nothing left in the expression, so pop the final element (28) off of the Stack, and you know that must be your answer. (0 items in Stack)

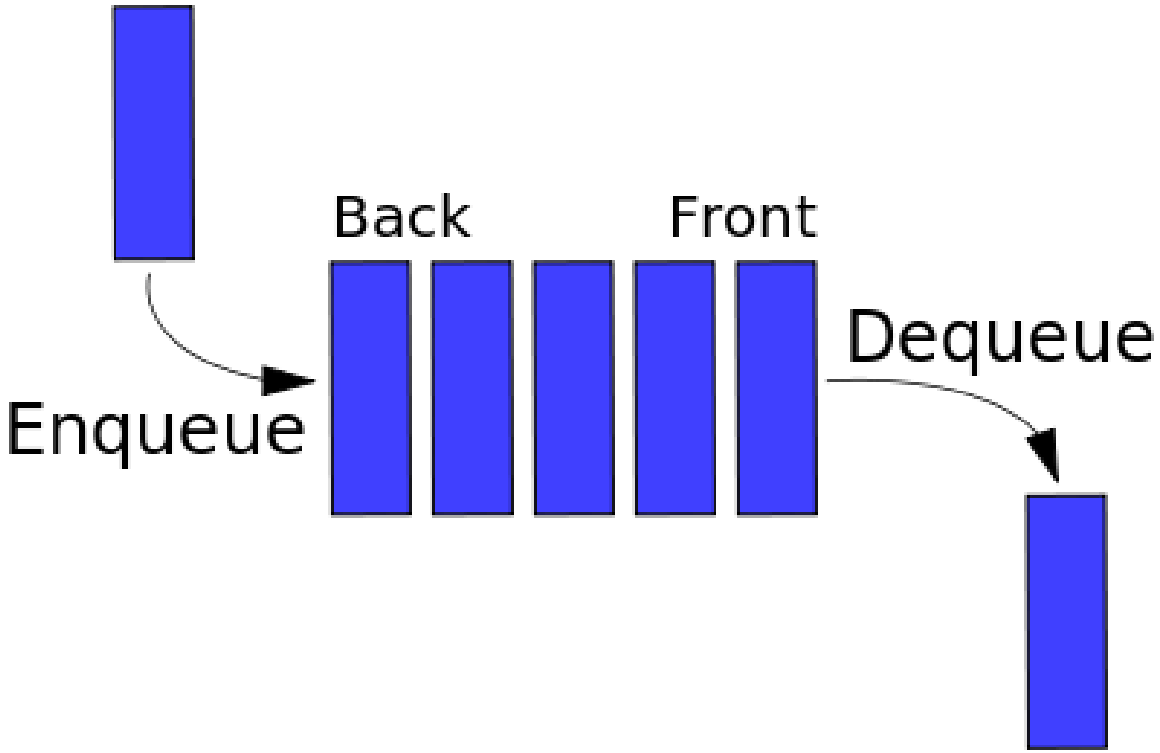
Infix to Postfix Conversion



3 6 + 2 4 - * 7 +

What's this ?

Queue



Queue

- A stack is LIFO (Last-In First Out) structure. In contrast, a queue is a FIFO (First-In First-Out) structure.
- In a FIFO data structure, the first element added to the queue will be the first one to be removed.
- A queue is a linear structure for which items can be only inserted at one end and removed at another end.

Operations on Queue

- We will dedicate once again six operations on the Queue.
  - You can add a person to the queue (enqueue),
  - Look who is the first person to be serviced (front)
  - Remove the first person (dequeue) and
  - Check whether the queue is empty (isEmpty).
  - Also there are two more operations to create and to destroy the queue.

Operations

- Queue create()
  - Creates an empty queue
- boolean isEmpty(Queue q)
  - Tells whether the queue q is empty
- enqueue(Queue q, Item e)
  - Place e at the rear of the queue q
- destroy(Queue q)
  - destroys queue q

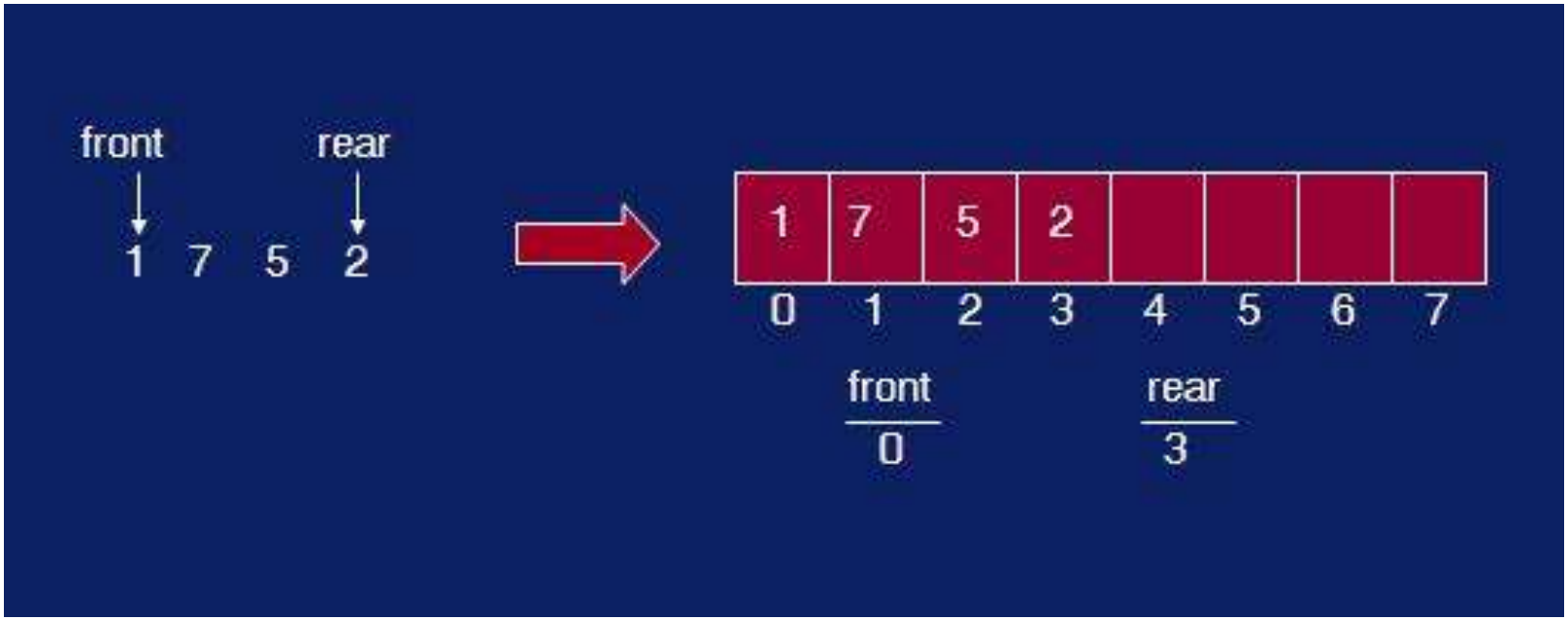
Operations

- Item front(Queue q)
  - returns the front element in Queue q without removing it
    Precondition: q is not empty
- dequeue(Queue q)
  - removes front element from the queue q
    Precondition: q is not empty

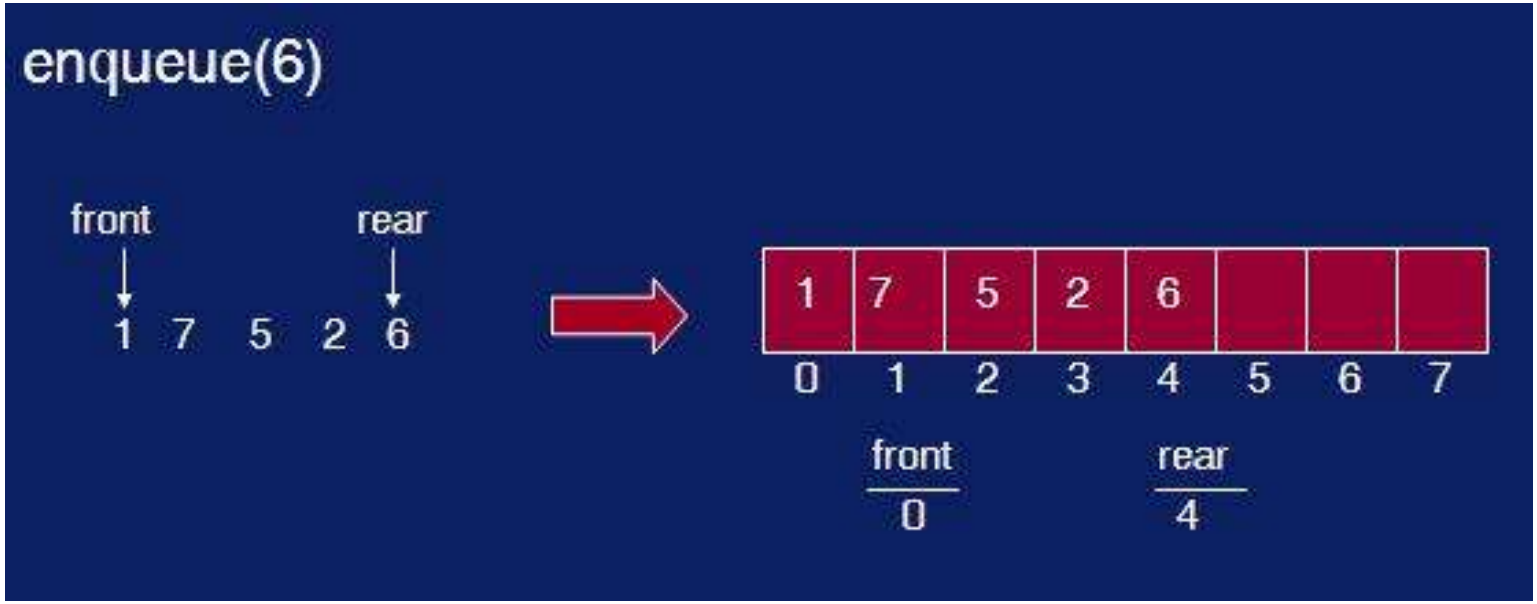
Implementation Using Arrays

- If we use an array to hold queue elements, dequeue operation at the front (start) of the array is expensive.
- This is because we may have to shift up to “n” elements.
- For the stack, we needed only one end; for queue we need both.
- To get around this, we will not shift upon removal of an element.

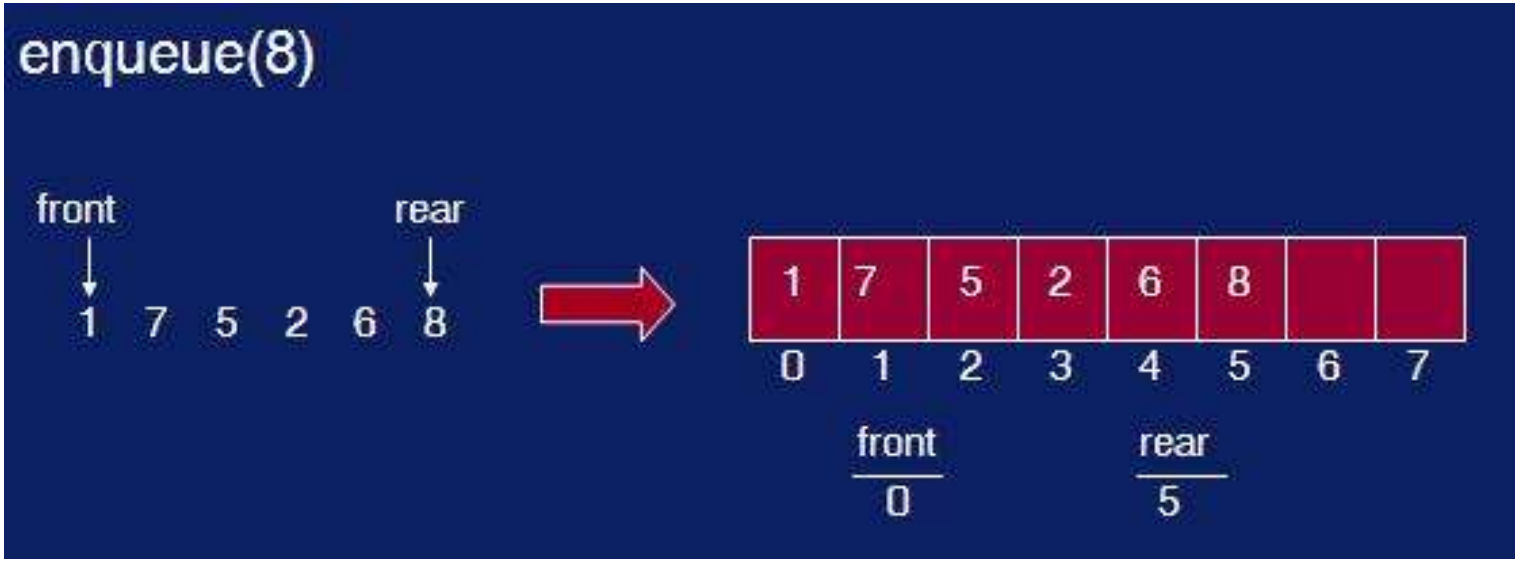
Snapshot of a Queue



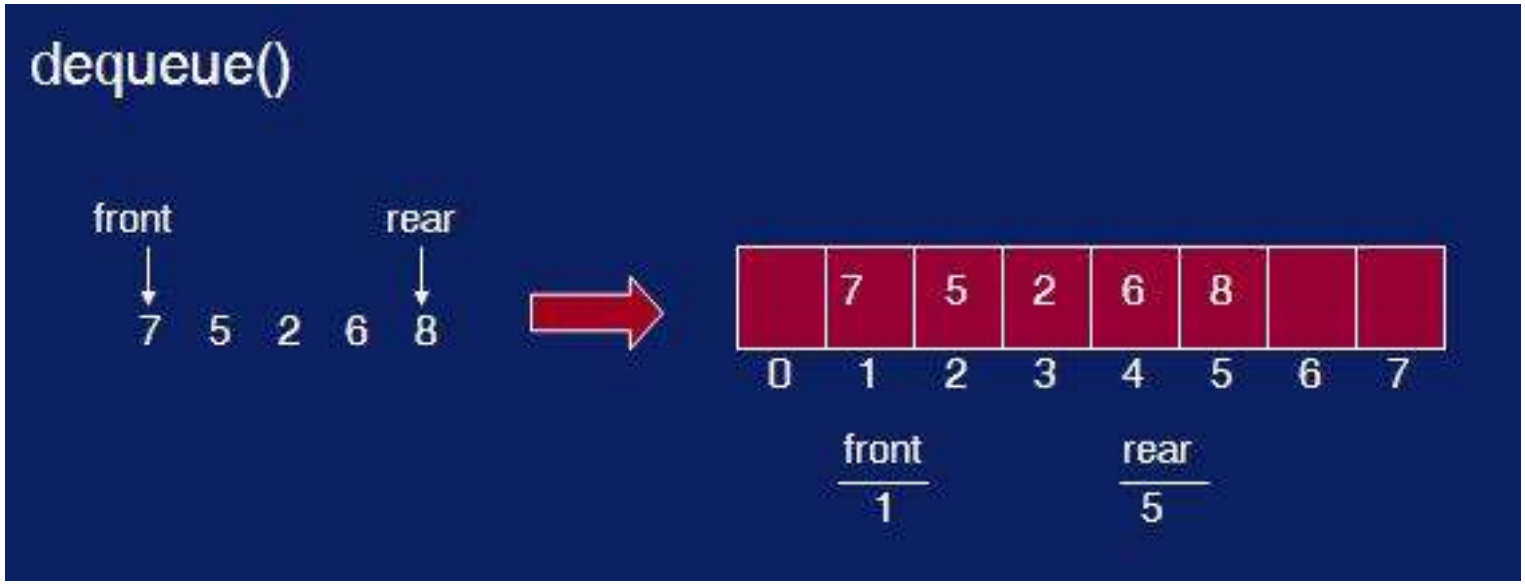
enqueue()



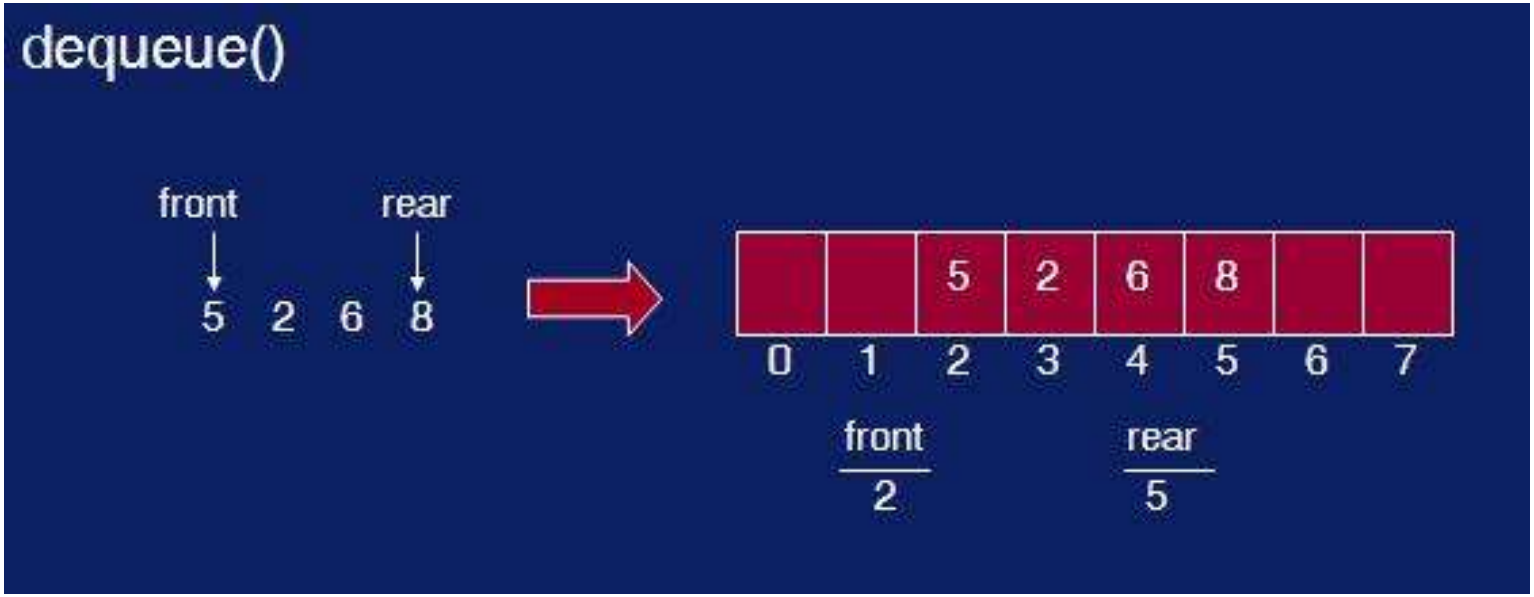
enqueue() again



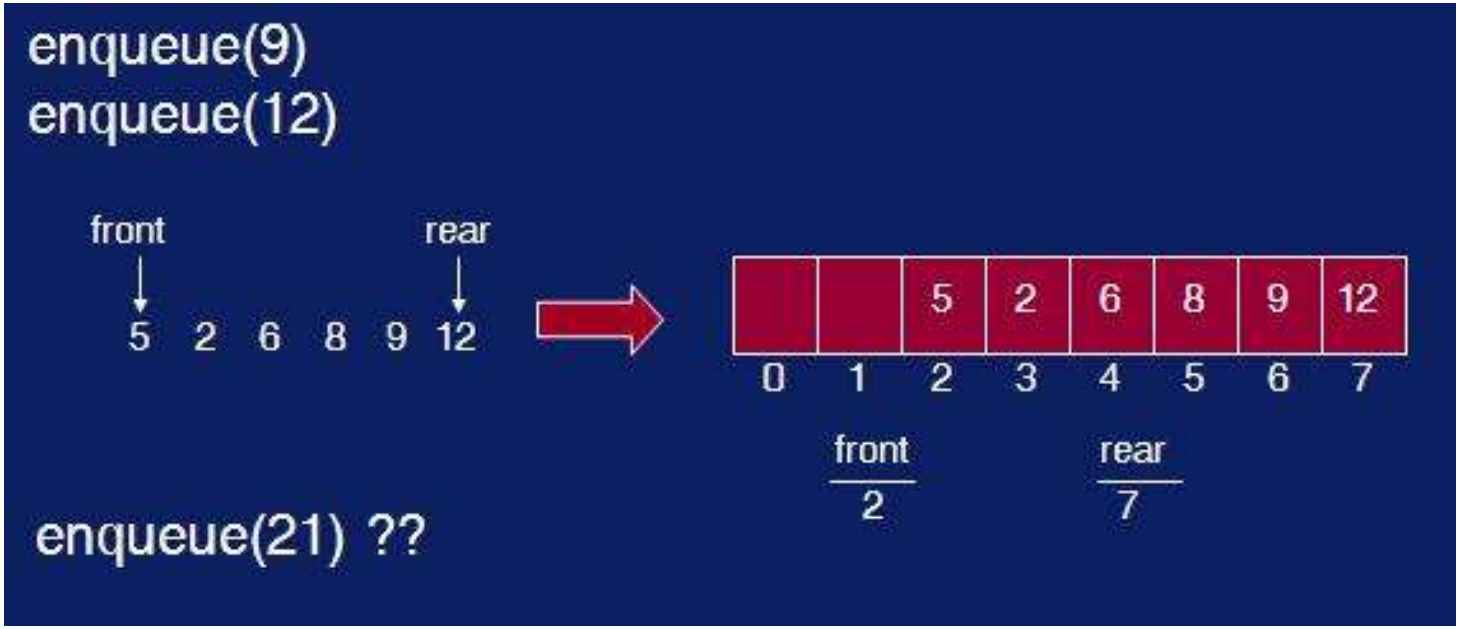
dequeue()



dequeue() again



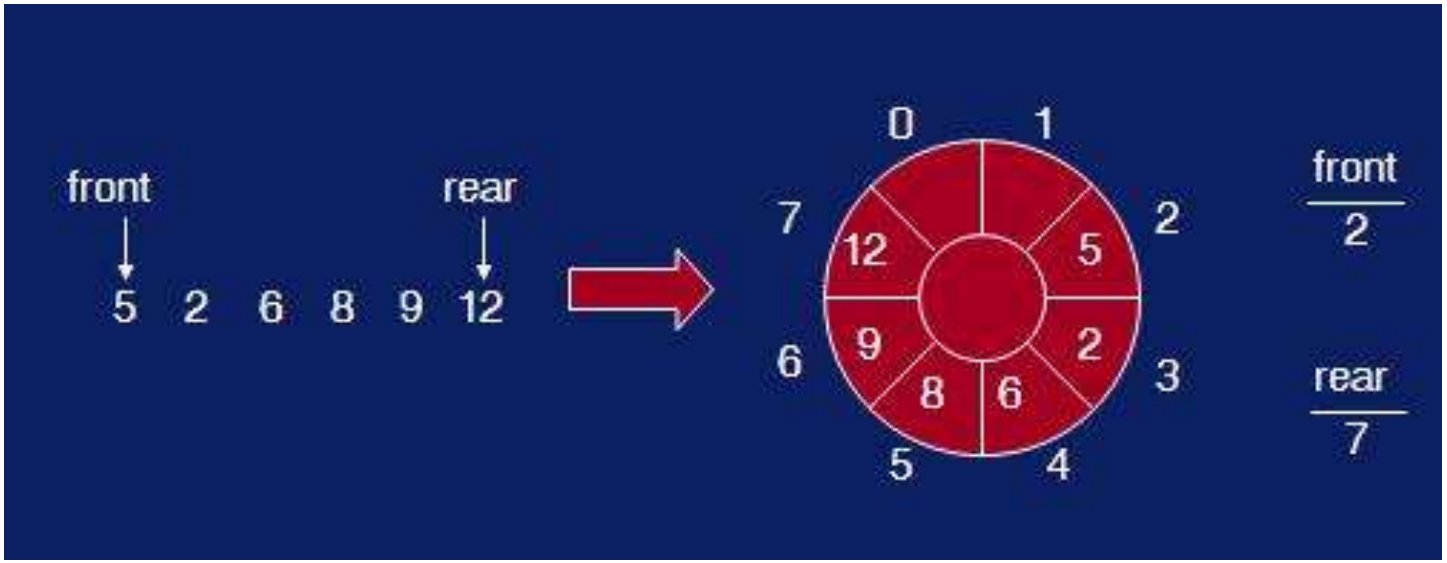
Two more enqueue()



What's Wrong ?

- We have inserts and removal running in constant time but we created a new problem.
- We cannot insert new elements even though there are two places available at the start of the array.
- Solution: allow the queue to "wrap around". Basic idea is to picture the array as a circular array.

Queue array wrap-around



enqueue(element)

```
void enqueue(int x)
{
    rear = (rear+1)%size;
    array[rear] = x;
      noElements = noElements+1;
}
```

dequeue()

```
int dequeue()
{
    int x = array[front];
      front = (front+1)%size;
      noElements = noElements-1;
      return x;
}
```

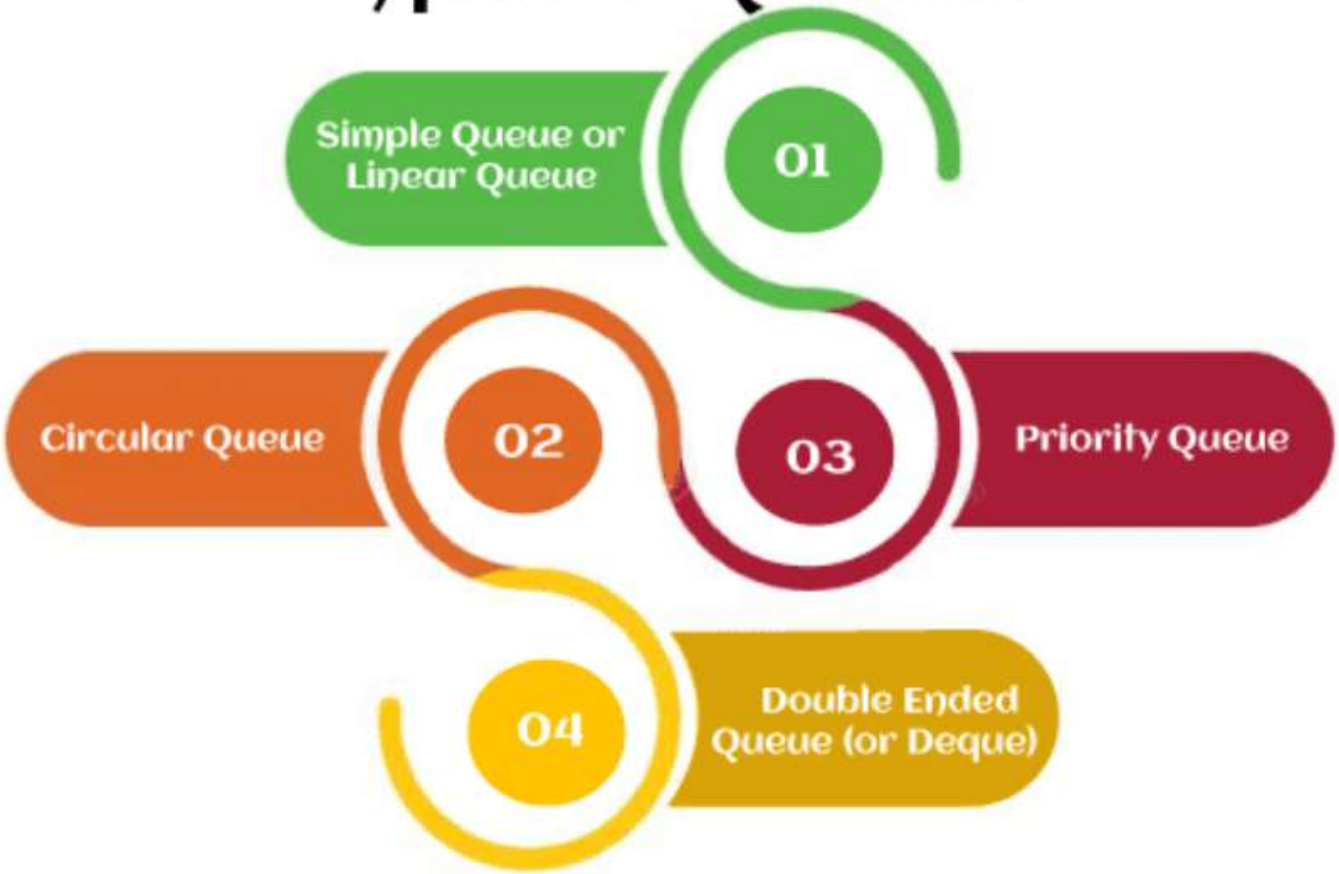
isEmpty() and isFull()

```
int isFull()
{
    return noElements == size;
}

int isEmpty()
{
      return noElements == 0;
}
```

isEmpty() and isFull()

```
int isFull()
{
    return noElements == size;
}

int isEmpty()
{
      return noElements == 0;
}
```

Types of Queues

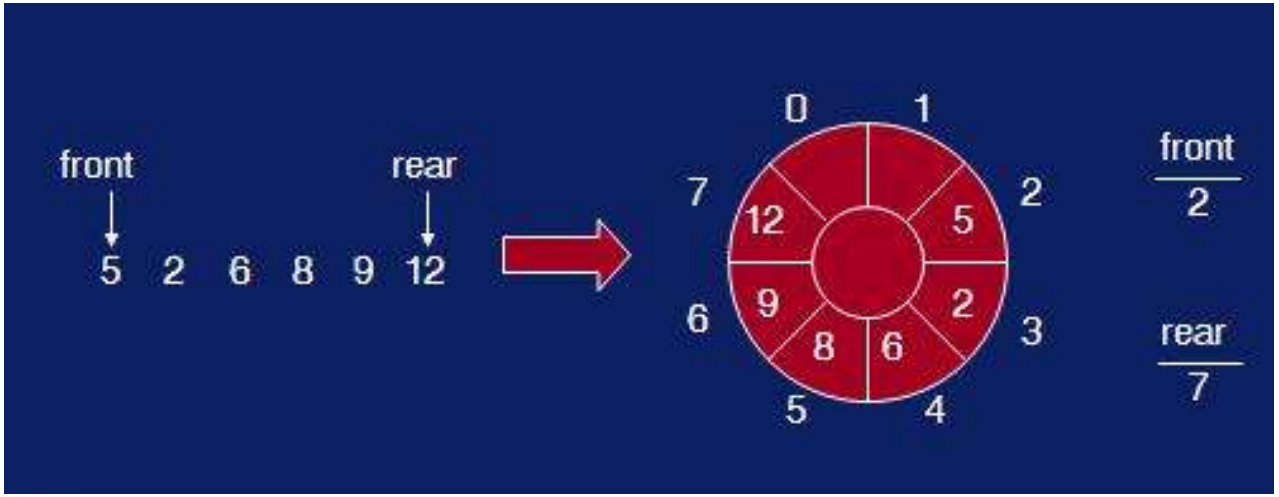


Simple Queue or Linear Queue

- In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end.

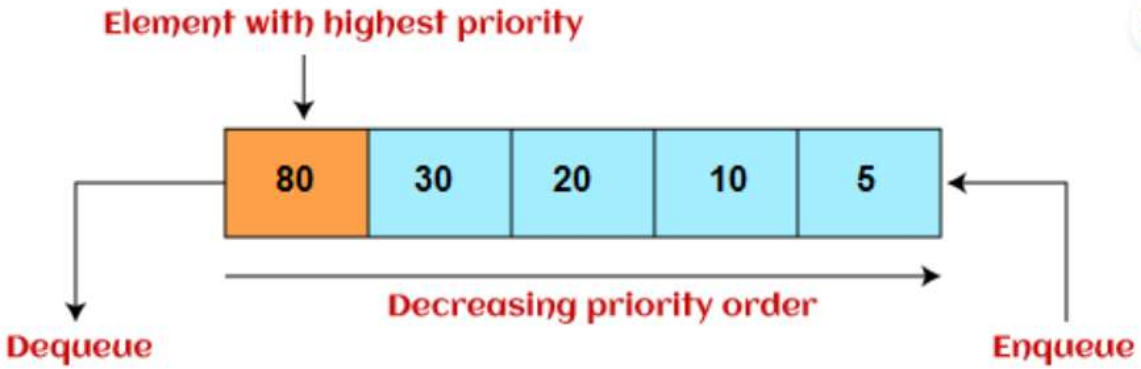
Circular Queue

- In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end.



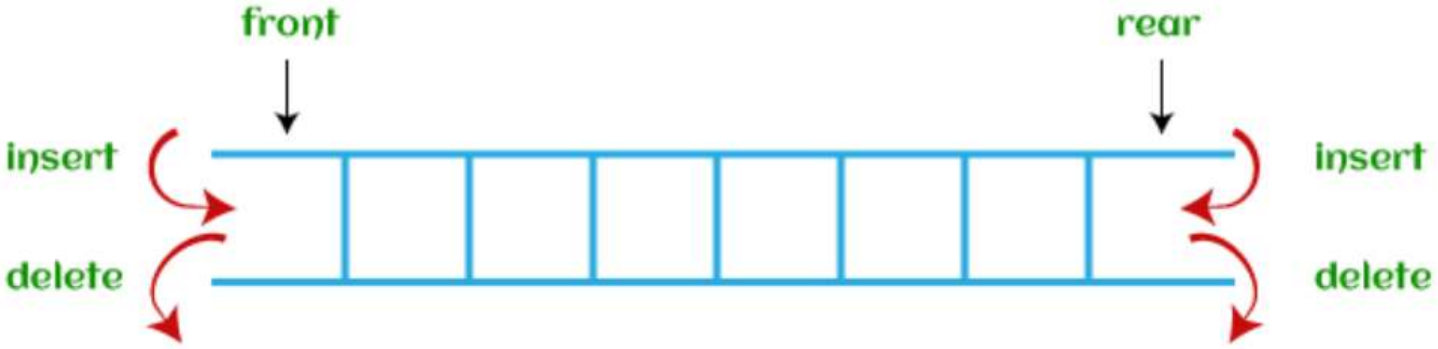
Priority Queue

- It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.



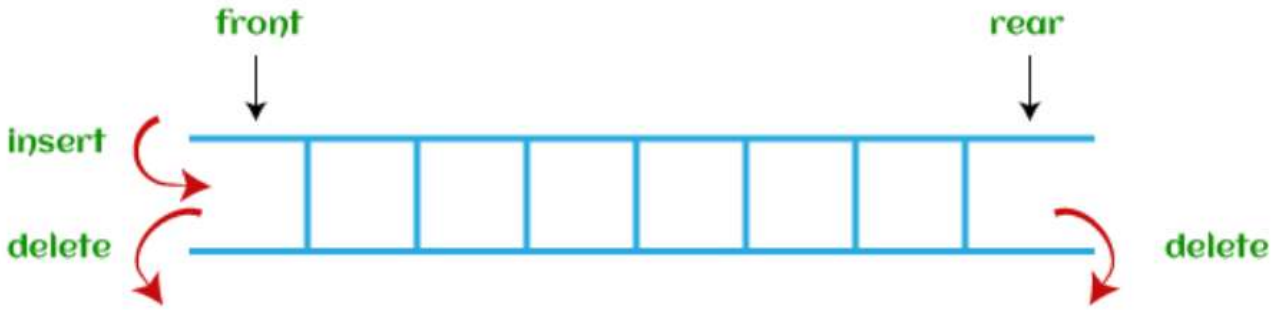
Deque (or, Double Ended Queue)

- In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue.

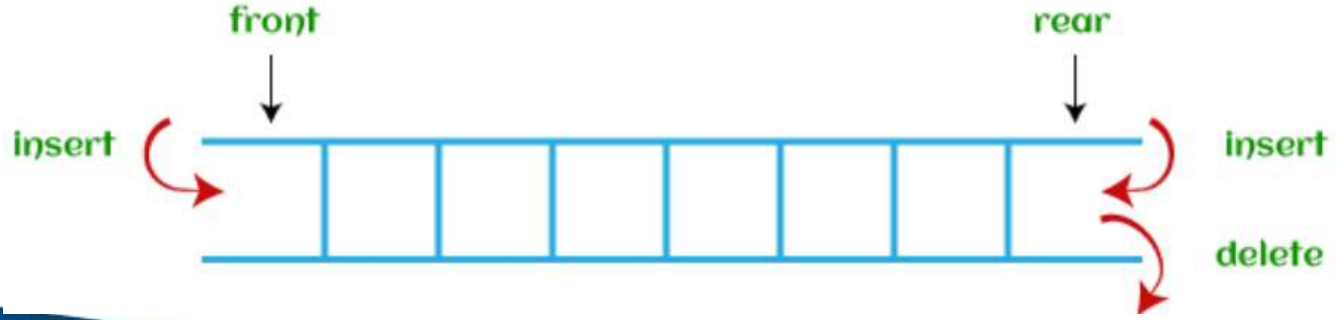


Deque (or, Double Ended Queue)

- There are two types of deque:
  - **Input restricted deque**



  - **Output restricted deque**



Applications of Queues

- Queue is used when things don't have to be processed immediatly, but have to be processed in First In First Out order like Breadth First Search.
- This property of Queue makes it also useful in following kind of scenarios.
- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Applications of Queues

- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
- Computer systems must often provide a “holding area” for messages between two processes, two programs, or even two systems. This holding area is usually called a “buffer” and is often implemented as a queue.
- Call center phone systems will use a queue to hold people in line until a service representative is free.

Applications of Queues

- Buffers on MP3 players and portable CD players, iPod playlist. Playlist for jukebox - add songs to the end, play from the front of the list.
- Round Robin (RR) algorithm is an important scheduling algorithm. It is used especially for the time-sharing system. The circular queue is used to implement such algorithms.