Queue

Queue ADT

- Like a stack, a queue (pronounced "cue") is a List ADT that holds a sequence of elements.
- **Restriction:** Items are inserted at one end while deleted from other end.
- A queue, provides access to its elements in ***first-in, first-out (FIFO)*** order.
- The elements in a queue are processed like customers standing in a grocery check-out line: the first customer in line is the first one served.

Example Applications of Queues

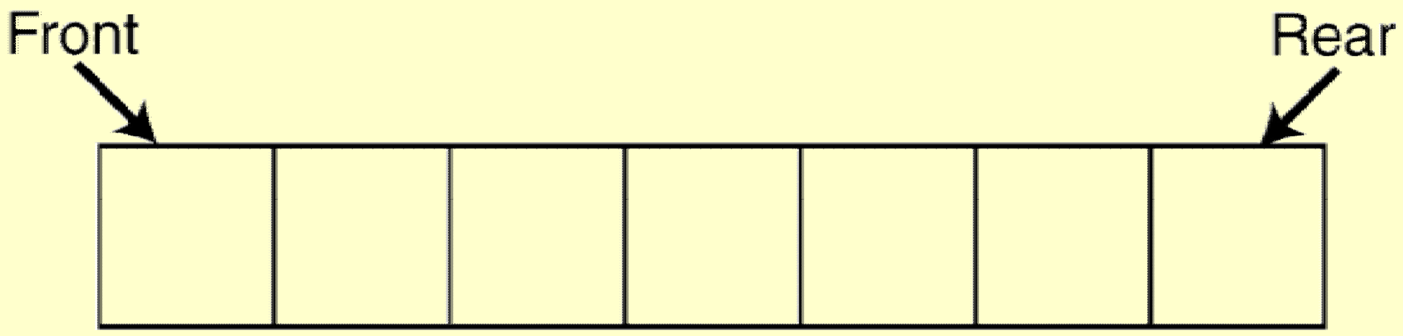
- In a multi-user system, a queue is used to hold print jobs submitted by users , while the printer services those jobs one at a time.
- Communications software also uses queues to hold information received over networks and dial-up connections. Sometimes information is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.

Static and Dynamic Queues

- Just as stacks are implemented as **arrays** or **linked lists**, so are queues.
- Dynamic queues (linked list implementation) offer the same advantages over static queues that dynamic stacks offer over static stacks.

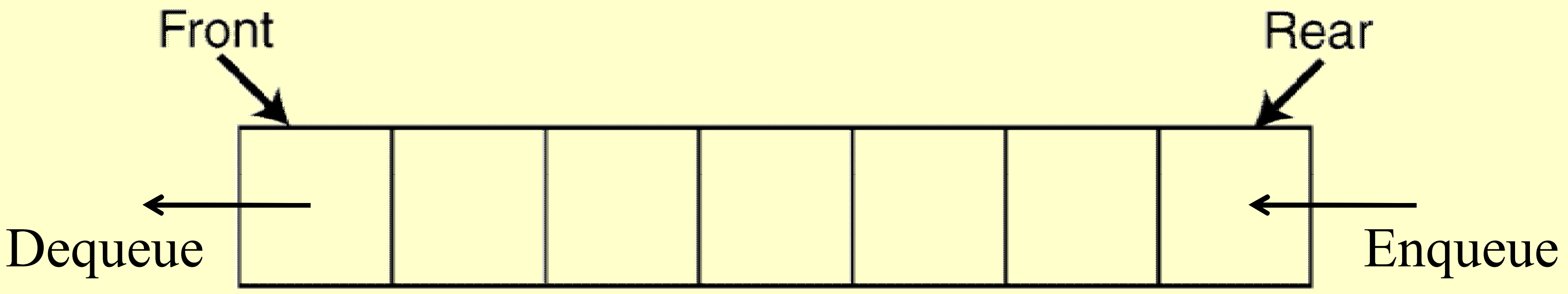
Queue Operations

- Think of queues as having a front and a rear.



Queue Operations

- The two primary queue operations are ***enqueuing*** and ***dequeuing***.
- To ***enqueue*** means to insert an element at the rear of a queue.
- To ***dequeue*** means to remove an element from the front of a queue.



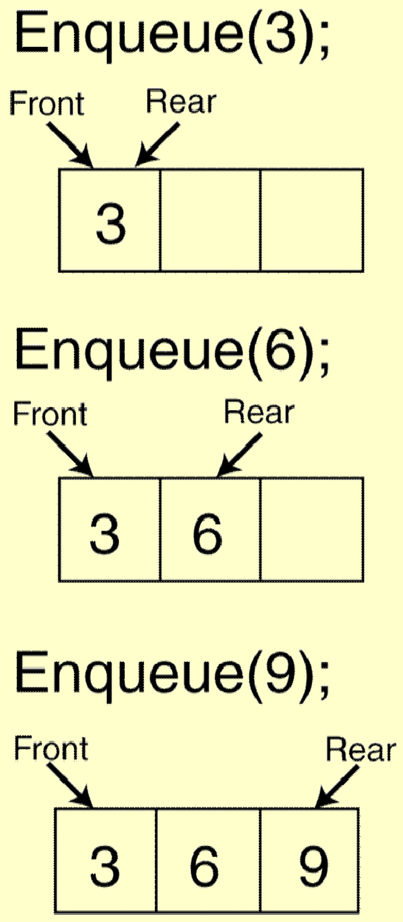
Array Implementation

- Suppose we have an empty static queue that is capable of holding a maximum of three values. With that queue we execute the following enqueue operations.

```
Enqueue(3);
Enqueue(6);
Enqueue(9);
```

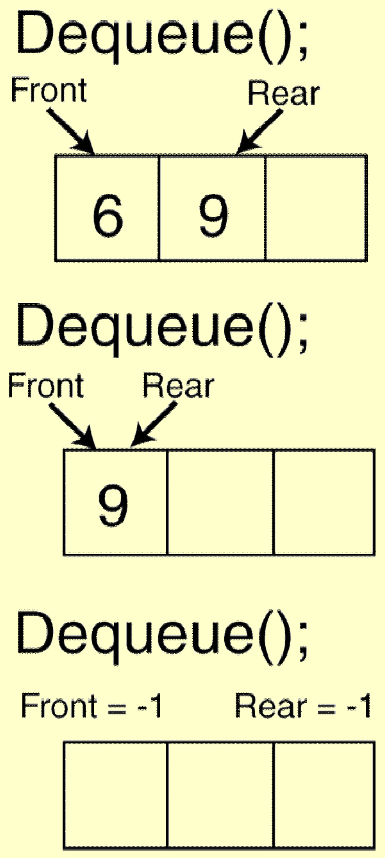
Queue Operations

- The state of the queue after each of the **enqueue** operations.



Queue Operations

- The state of the queue after each of three consecutive **dequeue** operations:

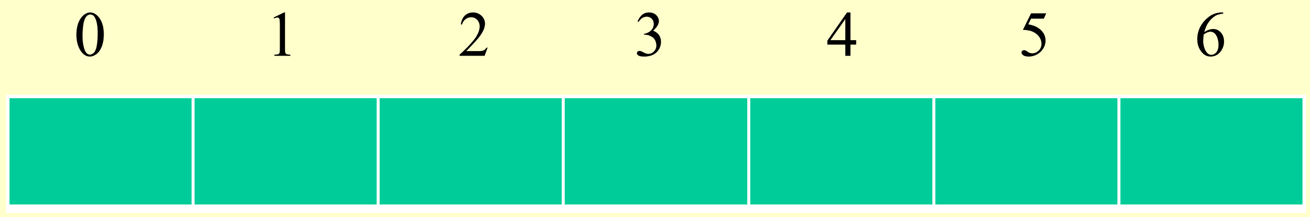


Queue Operations

- When the last deqeue operation is performed in the illustration, the queue is empty. An empty queue can be signified by setting both front and rear indices to –1.

- This method is inefficient: WHY?

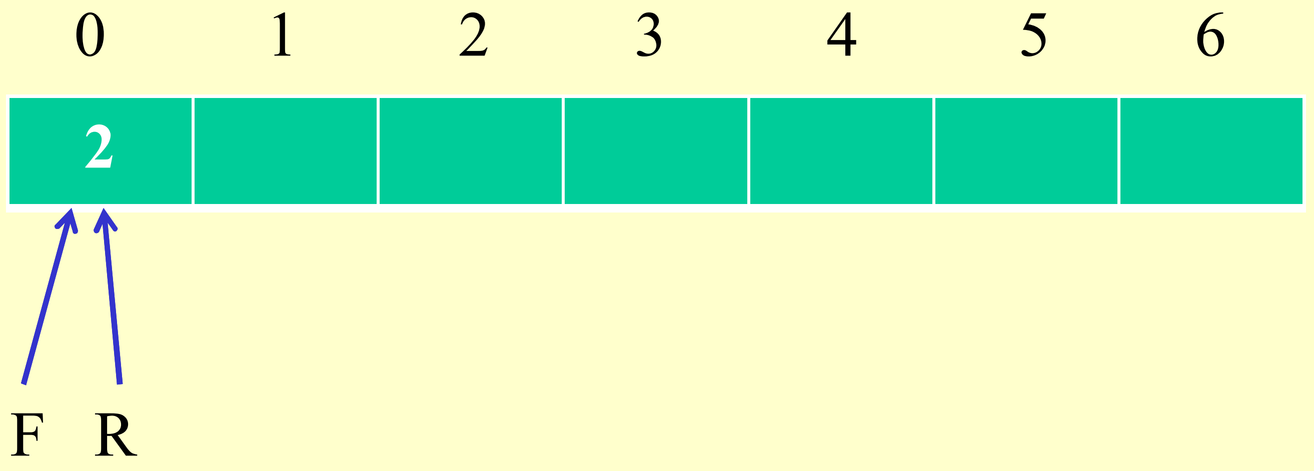
Array Implementation - Limitations



F = R = -1

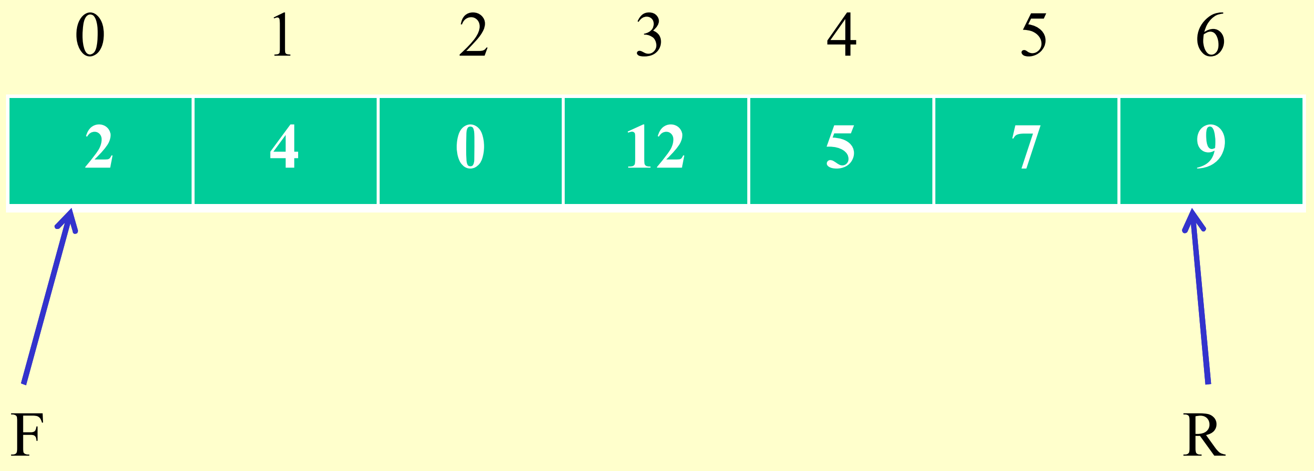
Empty Queue

Array Implementation - Limitations



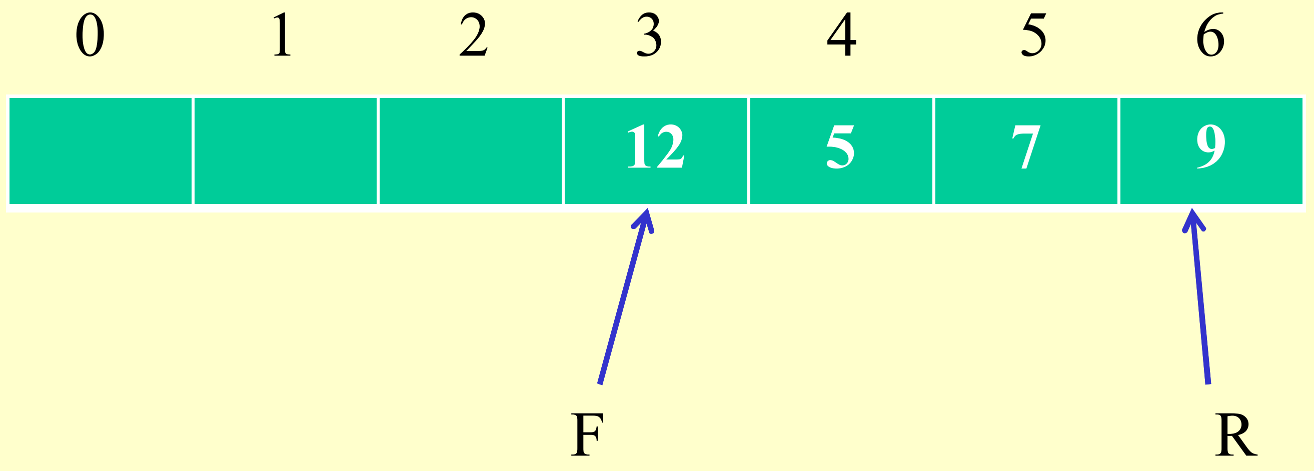
Queue with 1 element

Array Implementation - Limitations



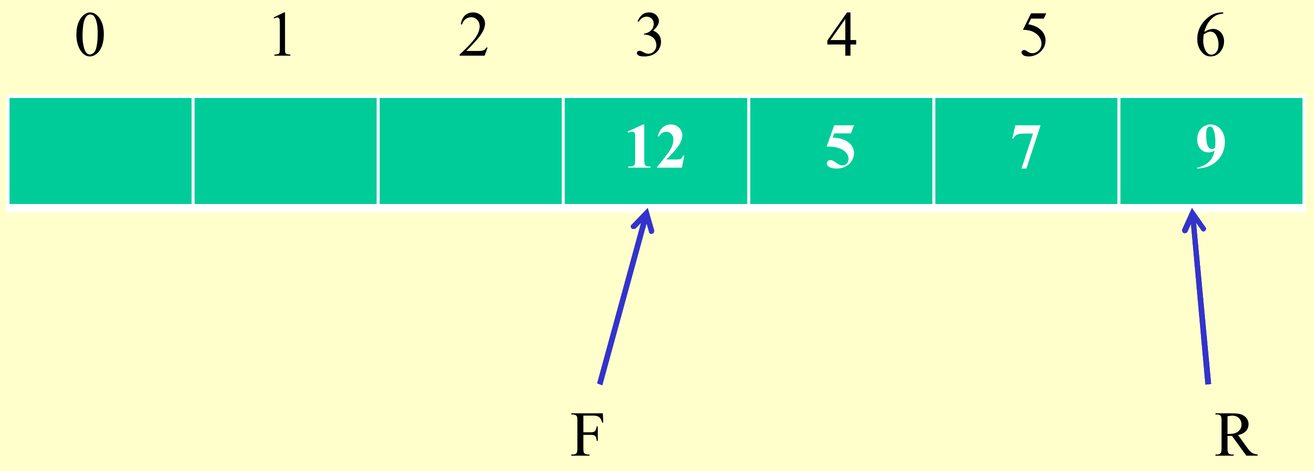
After inserting all 6 elements

Array Implementation - Limitations



After deleting 3 elements

Array Implementation - Limitations



Insert another element: Overflow even the queue is not full

Queue class

```
class Queue
{
private:
      double *queueArray;
      int maxSize;
      int front;
      int rear;
      int numItems;
public:
      Queue(int);
      ~Queue(void);
      void enqueue(double);
      double dequeue(void);
      bool isEmpty(void);
      bool isFull(void);
      void clear(void);
};
```

Constructor

```
#include <iostream.h>
#include "IntQueue.h“

//*************************
// Constructor            *
//*************************


Queue::Queue(int s)
{
      queueArray = new double[s];
      maxSize = s;
      front = -1;
      rear = -1;
      numItems = 0;
}
```

O(1)

Destructor

```
//***************************
// Destructor               *
//***************************

Queue::~Queue(void)
{
      delete [] queueArray;
}
```

O(1)

Enqueue operation

```
//*********************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//*********************************************

void Queue::enqueue(double num)
{
        if (isFull())
                cout << "The queue is full.\n";
        else
        {
                // Calculate the new rear position
                rear = rear + 1;
                // Insert new item
                queueArray[rear] = num;
                // Update item count
                numItems++;
                // if first item set front=0
                if (numItems==1)
                  front=0;
        }
}
```

O(1)

Dequeue operation

```
//********************************************
// Function dequeue removes the value at the  *
// front of the queue, and shift remaining elements*
//********************************************

double Queue::dequeue(void)
{
        if (isEmpty())
                cout << "The queue is empty.\n";
        else
        {
                // Retrieve the front item
                double num = queueArray[front];
                // Shift remaining items left
                for(int i=front,i<rear;i++)
               queueArray[i]=queueArray[i+1];
                // Update rear and item count, and return
                rear=rear-1;   numItems--;
                return(num);
                //If last item deleted, set front=-1
                if (numItems==0)
                  front=-1;
        }
}
```

O(n)

isEmpty operation

```
//***********************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.             *
//***********************************************

bool Queue::isEmpty(void)
{
      bool status;

      if (numItems==0)  //or, if (front==-1)
            status = true;
      else
            status = false;

      return status;
}
```

O(1)

isFull operation

```
//**************************************
// Function isFull returns true if the queue *
// is full, and false otherwise.            *
//**************************************

bool Queue::isFull(void)
{
      bool status;

      if (numItems == maxSize) // or, if (rear==maxSize-1)
            status = true;
      else
            status = false;

      return status;
}
```

O(1)

Clear operation

```
//********************************************
// Function clear resets the front and rear *
// indices, and sets numItems to 0.         *
//********************************************

void Queue::clear(void)
{
      front = -1;
      rear = -1;
      numItems = 0;
}
```

O(1)

Illustrating Program

```
// This program demonstrates the Queue class


void main(void)
{
      Queue queue(5);

      cout << "Enqueuing 5 items...\n";
      // Enqueue 5 items.
      for (int x = 0; x < 5; x++)
            queue.enqueue(x);

      // Attempt to enqueue a 6th item.
      cout << "Now attempting to enqueue again...\n";
      queue.enqueue(5);
```

Illustrating Program

```
      // Deqeue and retrieve all items in the queue
      cout << "The values in the queue were:\n";
      while (!queue.isEmpty())
      {
            double value;
            value=queue.dequeue();
            cout << value << endl;
      }
}
==================================================
```

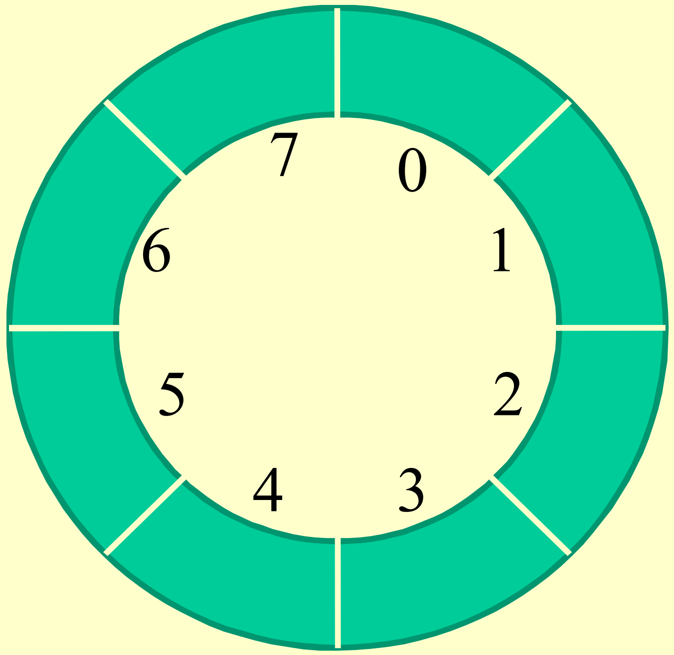
Program Output

```
Enqueuing 5 items...
Now attempting to enqueue again...
The queue is full.
The values in the queue were:
0
1
2
3
4
```

Improving efficiency

- Use circular queue

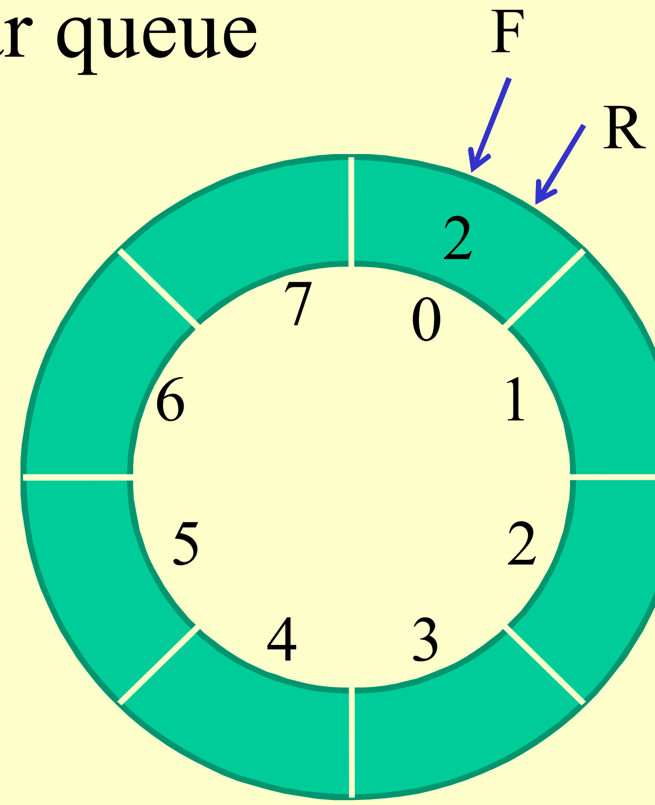
F=R= -1



Empty Queue

Improving efficiency

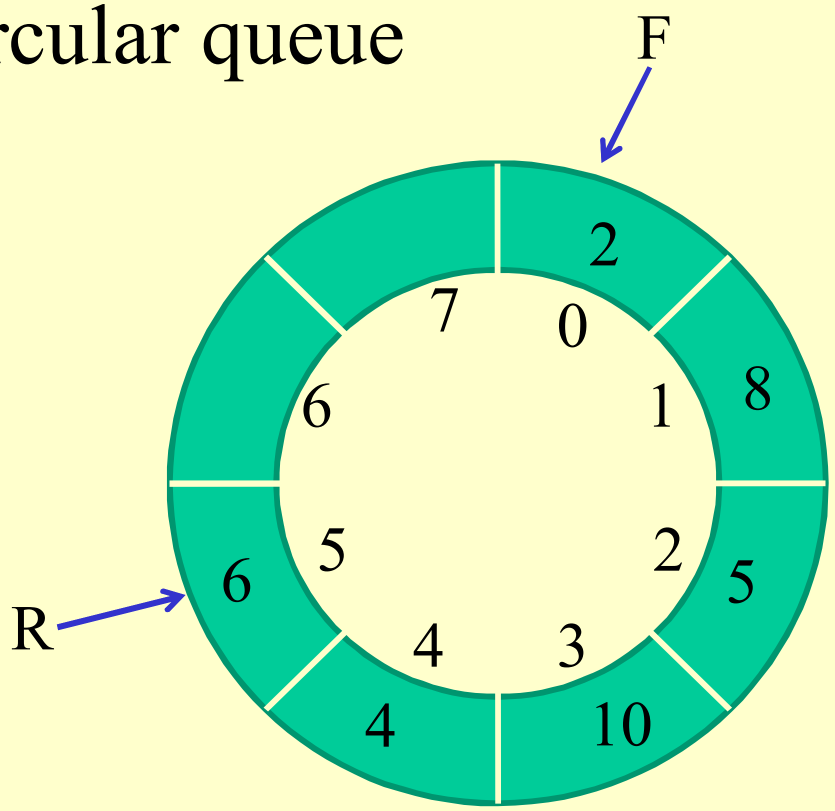
- Use circular queue



Queue with 1 element

Improving efficiency

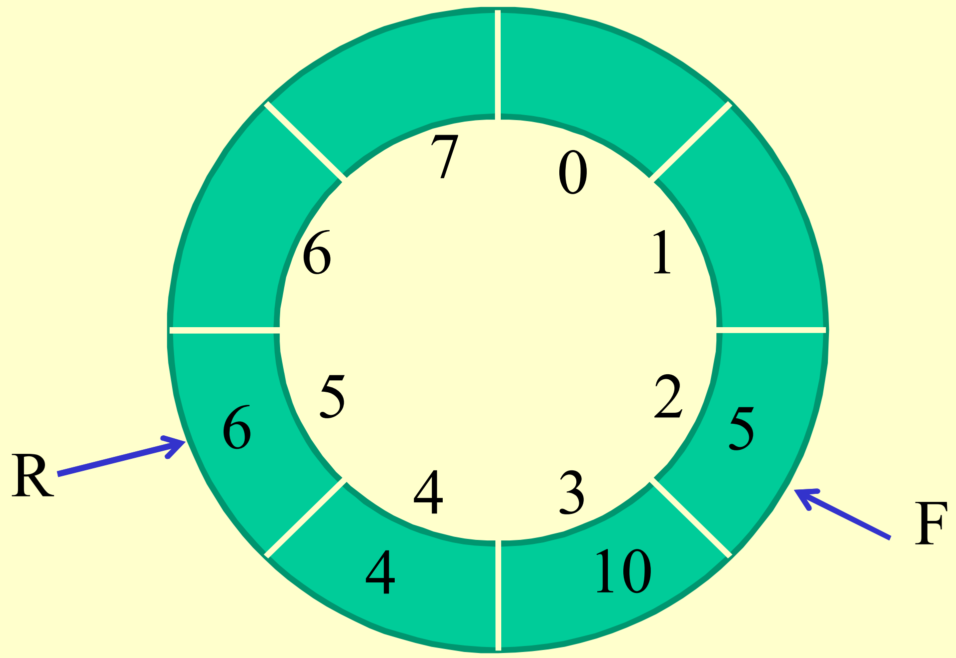
- Use circular queue



After inserting 6 items

Improving efficiency

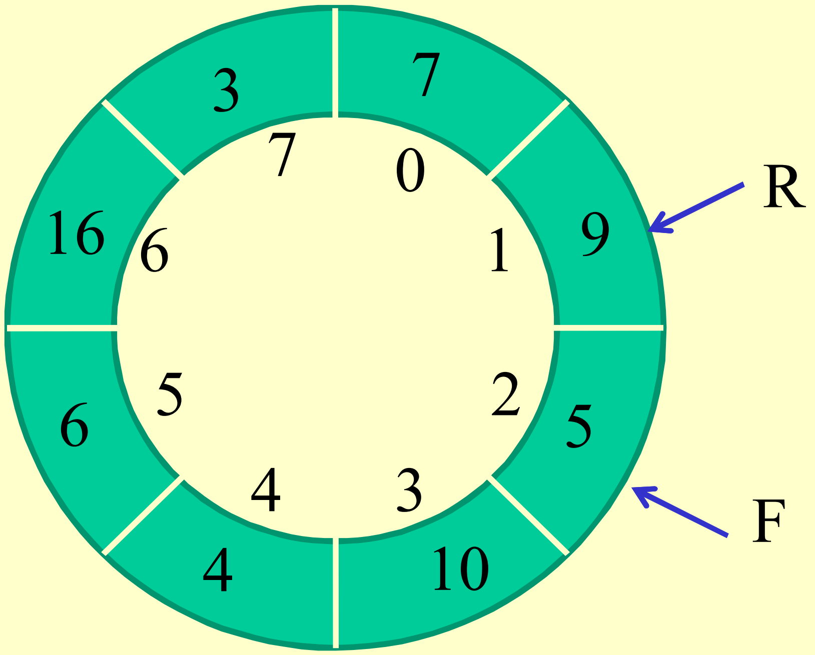
- Use circular queue



After deleting 2 items

Improving efficiency

- Use circular queue



After inserting another 4 items **(Full queue)**

Constructor

```
#include <iostream.h>
#include "IntQueue.h"

//**************************
// Constructor            *
//**************************


Queue::Queue(int s)
{
      queueArray = new double[s];
      maxSize = s;
      front = -1;
      rear = -1;
      numItems = 0;
}
```

O(1)

Destructor

```
//***************************
// Destructor               *
//***************************

Queue::~Queue(void)
{
      delete [] queueArray;
}
```

O(1)

Enqueue operation

```
//*******************************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//*******************************************************

void Queue::enqueue(double num)
{
        if (isFull())
                cout << "The queue is full.\n";
        else
        {
                // Calculate the new rear position
                rear = (rear + 1)%maxSize;
                // Insert new item
                queueArray[rear] = num;

                // Update item count
                numItems++;
                if (numItems == 1)   //First element
                   front=0;

        }
}
```

O(1)

Dequeue operation

```
//********************************************
// Function dequeue removes the value at the  *
// front of the queue, and copies it into num. *
//********************************************

double Queue::dequeue(void)
{
	if (isEmpty())
		cout << "The queue is empty.\n";
	else
	{
		// Retrieve the front item
		num = queueArray[front];
		// Move front
		front = (front + 1)%maxSize;
		// Update item count and return num
		numItems--;

		// Last element deleted
		if (numItems == 0){front = -1; rear = -1;}

		return(num);
	}
}
```

O(1)

isEmpty operation

```
//*********************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.             *
//*********************************************

bool Queue::isEmpty(void)
{
      bool status;

      if (numItems==0) //or, if (front==-1)
            status = true;
      else
            status = false;

      return status;
}
```

O(1)

isFull operation

```
//*****************************************
// Function isFull returns true if the queue *
// is full, and false otherwise.             *
//*****************************************

bool Queue::isFull(void)
{
       bool status;

       if (numItems == maxSize) //or, if (front==(rear+1)%maxSize)
              status = true;
       else
              status = false;

       return status;
}
```

O(1)

Clear operation

```
//*******************************************
// Function clear resets the front and rear *
// indices, and sets numItems to 0.         *
//*******************************************

void Queue::clear(void)
{
      front = -1;
      rear = -1;
      numItems = 0;
}
```

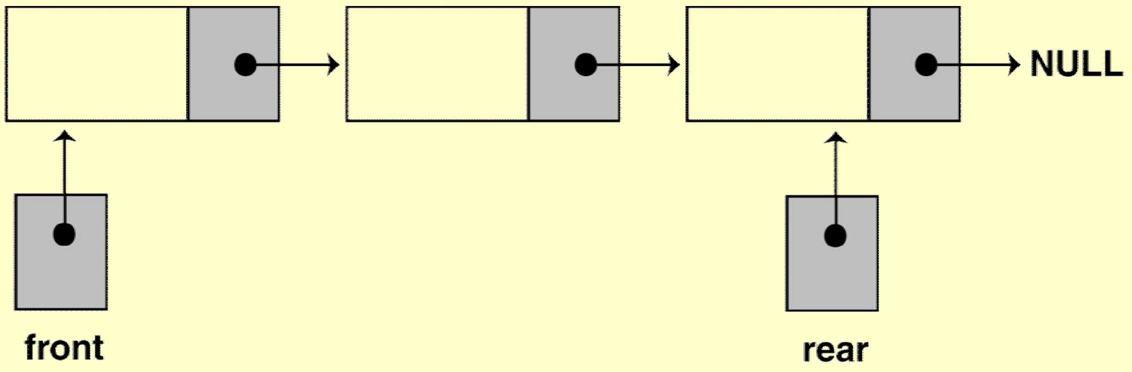
O(1)

Linked List-based Queues

- A dynamic queue starts as an empty linked list.
- With the first enqueue operation, a node is added, which is pointed to by `front` and `rear` pointers.
- As each new item is added to the queue, a new node is added to the rear of the list, and the `rear` pointer is updated to point to the new node.
- As each item is dequeued, the node pointed to by the `front` pointer is deleted, and `front` is made to point to the next node in the list.

Linked List-based Queues

- Figure shows the structure of a dynamic queue.



Queue Class

```
class Queue
{
private:
       struct QueueNode
       {
              int value;
              QueueNode *next;
       };

       QueueNode *front;
       QueueNode *rear;
       int numItems;
public:
       Queue(void);
       ~Queue(void);
       void enqueue(double);
       double dequeue(void);
       bool isEmpty(void);
       void clear(void);
};
```

Constructor and Destructor

```
//************************
// Constructor          *
//************************

Queue::Queue(void)
{
       front = NULL;
       rear = NULL;
       numItems = 0;
}

//************************
// Destructor           *
//************************

Queue::~Queue(void)
{
       clear();
}
```

O(1)

O(n)

Enqueue operation

```
//*********************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//*********************************************

void Queue::enqueue(double num)
{
       QueueNode *newNode = new QueueNode;
       newNode->value = num;
       newNode->next = NULL;
        if (isEmpty())
       {
              front = newNode;
              rear = newNode;
       }
       else
       {
              rear->next = newNode;
              rear = newNode;
       }
       numItems++;
}
```

O(1)

Dequeue operation

```
//*******************************************
// Function dequeue removes the value at the   *
// front of the queue, and copies it into num. *
//*******************************************

double Queue::dequeue(void)
{
        QueueNode *temp;

        if (isEmpty())
                cout << "The queue is empty.\n";
        else
        {
                num = front->value;
                temp = front->next;
                delete front;
                front = temp;
                numItems--;

                return(num);
        }
}
```

O(1)

isEmpty operation

```
//*************************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.             *
//*************************************************

bool Queue::isEmpty(void)
{
	bool status;

	if (numItems==0)  // or, if (front==NULL)
		status = true;
	else
		status = false;

	return status;
}
```

O(1)

Clear operation

```
//*******************************************
// Function clear dequeues all the elements  *
// in the queue.                             *
//*******************************************

void Queue::clear(void)
{
      int value;  // Dummy variable for dequeue

      while(!isEmpty())
            value = dequeue();
}
```

O(n)

Illustrating Program

```
// This program demonstrates the Queue class
#include <iostream.h>

void main(void)
{
	Queue queue;

	cout << "Enqueuing 5 items...\n";
	// Enqueue 5 items.
	for (int x = 0; x < 5; x++)
		queue.enqueue(x);

	// Deqeue and retrieve all items in the queue
	cout << "The values in the queue were:\n";
	while (!queue.isEmpty())
	{
		double value;
		value=queue.dequeue();
		cout << value << endl;
	}
}
```

Illustrating Program

Program Ouput

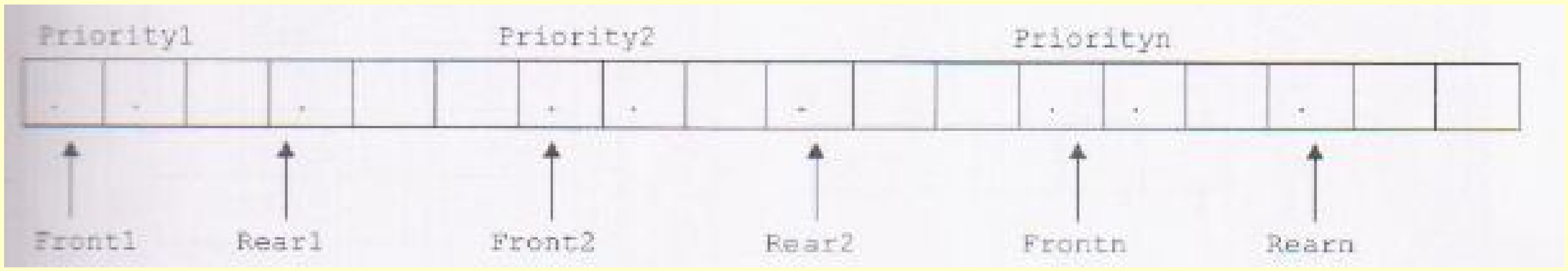
```
Enqueuing 5 items...
The values in the queue were:
0
1
2
3
4
```

PRIORITY QUEUE

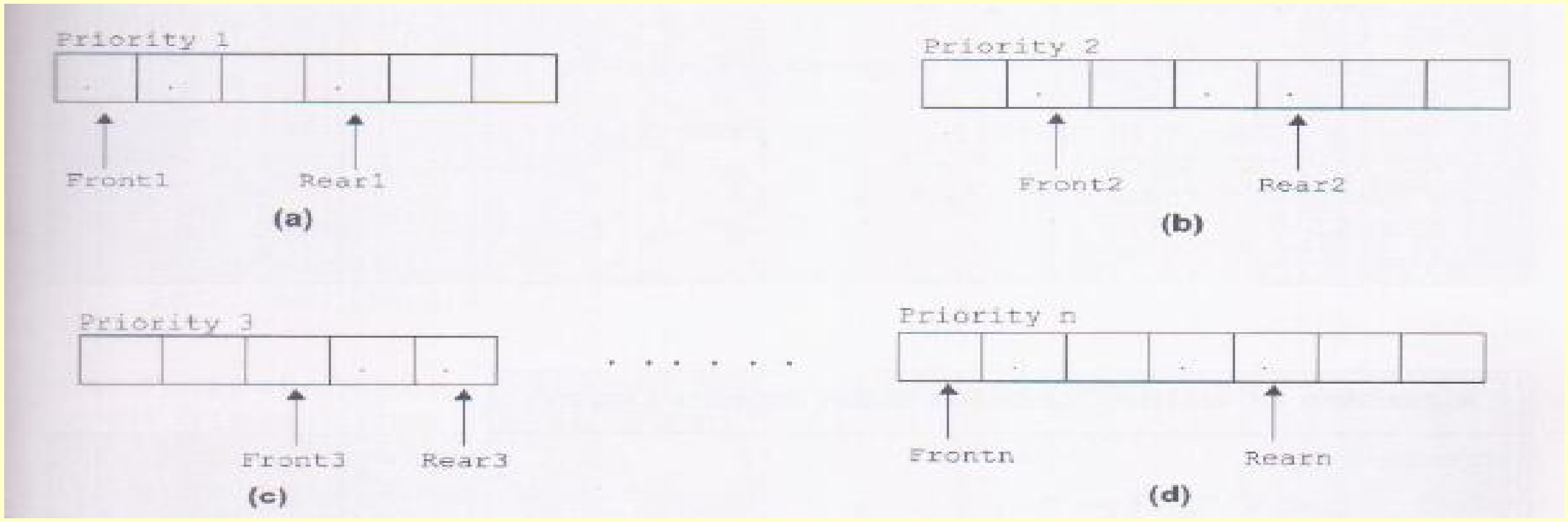
- A priority queue is a type of queue in which each element is assigned a priority.
- The elements are inserted at the rear.
- The elements are deleted according to the priority.
- While implementing a priority queue, following two rules are applied.
  - The element with higher priority is processed before any element of lower priority.
  - The elements with the same priority are processed according to the order in which they were added to the queue.
- A priority queue can be represented in many ways. Here, we are discussing **multi-queue** implementation of priority queue.

Multi-queue Implementation (Array Representation of Priority Queue)

- In multi-queue representation of priority queue, for each priority, a queue is maintained. The queue corresponding to each priority can be represented in the same array of sufficient size.
- For each queue, two variables **Front_i** and **Rear_i** are maintained.
- Observe that Front_i and Rear_i correspond to the front and rear position of queue for priority Priority_i.

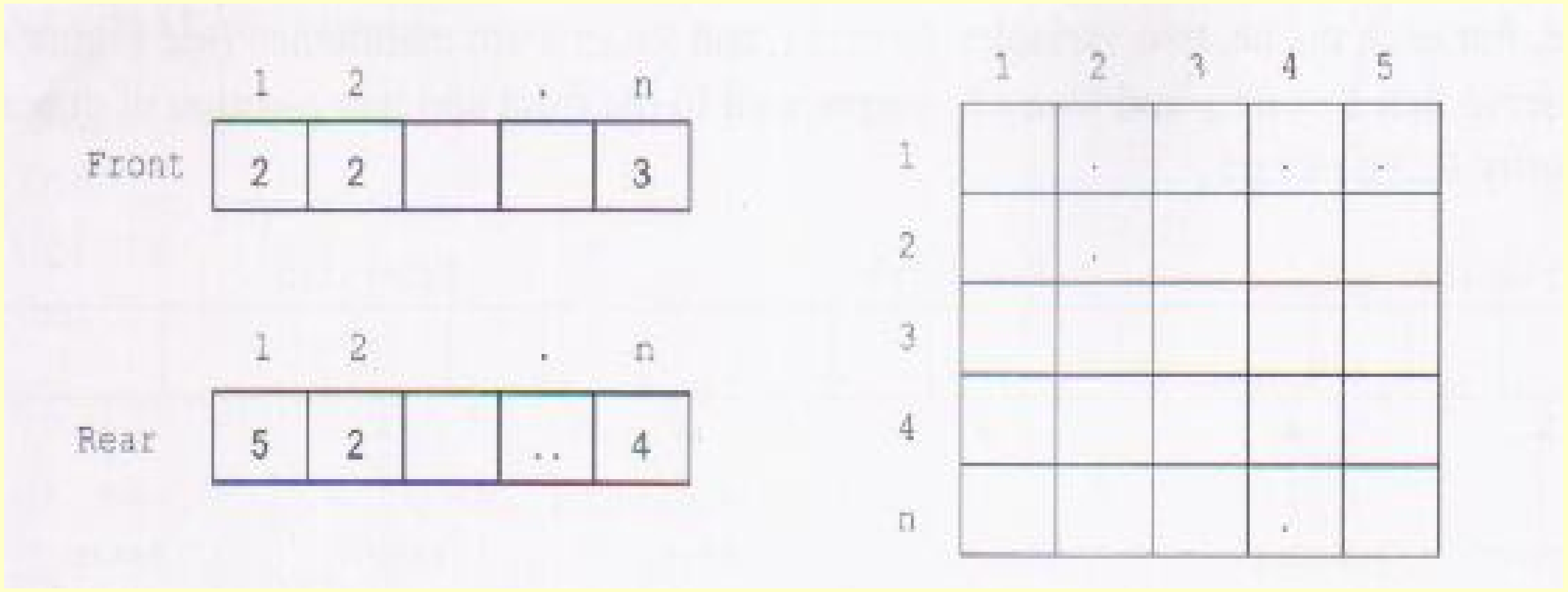


- Clearly, in this representation, shifting is required to make space for an element to be inserted.
- To avoid this shifting, an alternative representation can be used. In this representation, instead of representing queue corresponding to each priority using a single array, a separate array for each priority is maintained.
- Each queue is implemented as **a** circular array and has its own two variables, Front and Rear.
- The element with given priority number is inserted in the corresponding queue. Similarly, whenever an element is to be deleted from the queue, it must be the element from the highest priority queue.
- Note that lower priority number indicates higher priority.



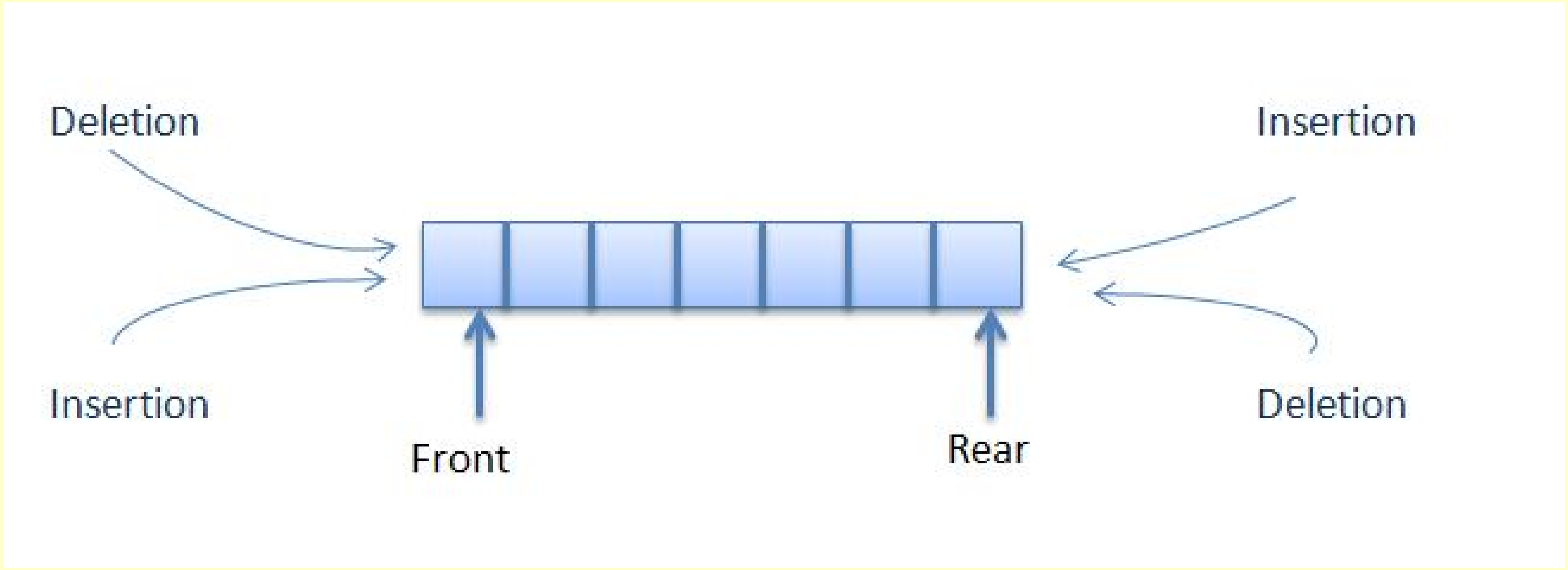
2D Array Implementation

- If the size of each queue is same, then instead of multiple one-dimensional arrays, a single two-dimensional array can be used where row i corresponds to the queue of priority i.
- In addition, two single dimensional arrays are used. One is used to keep track of front position and another to keep track of rear position of each queue.



DEQUE

- Deque (short form of **Double-Ended QUEue**) is a linear list in which elements can be inserted or deleted at either end but not in the middle.
- Pronounced as "DECK".
- That is, elements can be inserted/deleted to/from the rear or the front end.



Deque – Possible operations

- `void insertFront(double)`
- `void insertRear(double)`
- `double deleteFront(void)`
- `double deleteRear(void)`
- `bool isEmpty(void)`
- `bool isFull(void)`
- `double front(void)`
- `double rear(void)`
- `void clear(void)`

VARIATIONS OF DEQUE

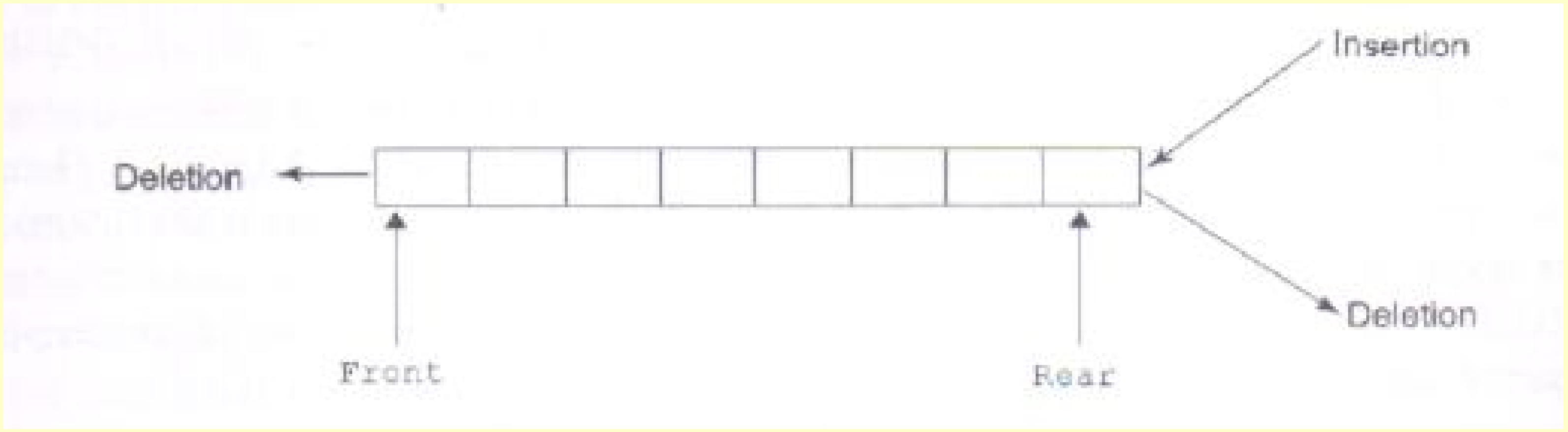
- There are two variations of a deque that are as follows:

  (1)Input restricted deque: It allows insertion of elements at one end only but deletion can be done at both ends.

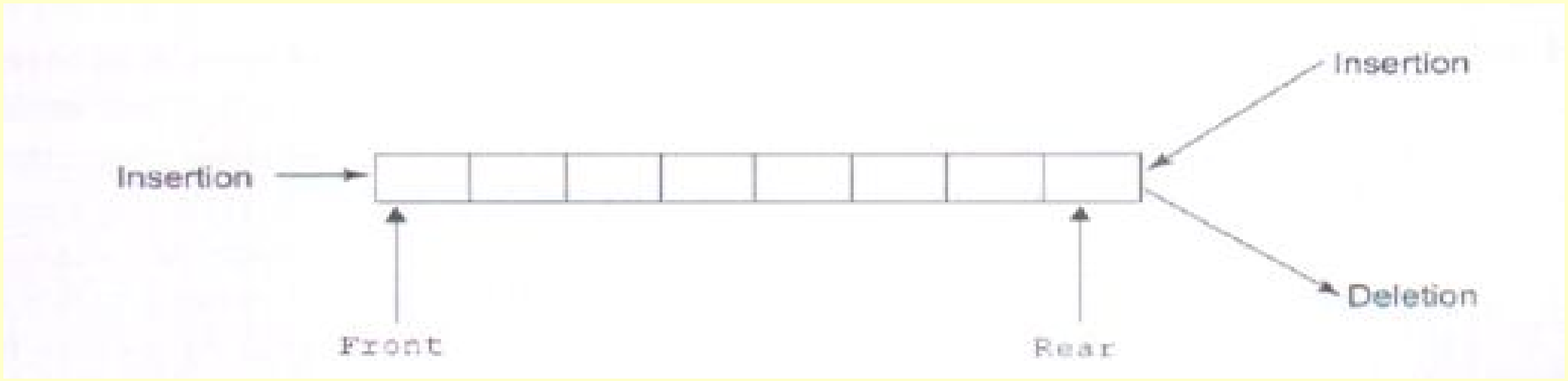
  (2)Output restricted deque: It allows deletion of elements at one end only but insertion can be done at both ends.

- The implementation of both these queues is similar to the implementation of deque. The only difference is that in input restricted queue, function for insertion in the beginning is not needed whereas in output-restricted queue, function for deletion in the beginning is not needed.

Variations of Deque

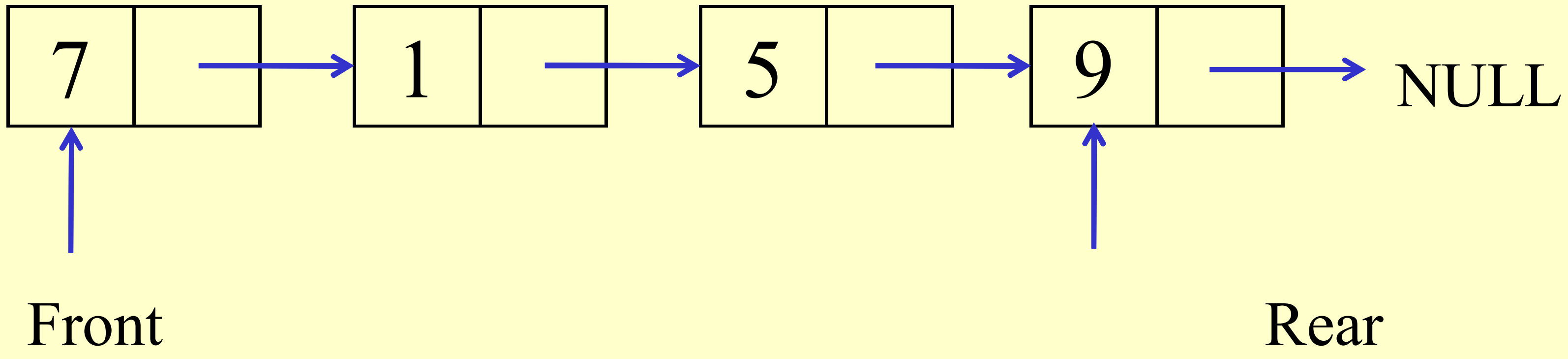


Input-restricted Deque



Output-restricted Deque

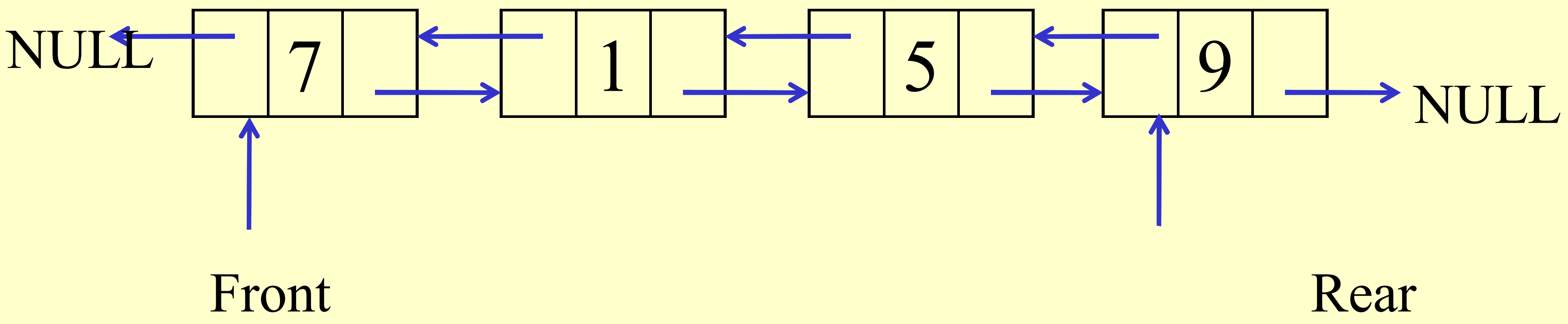
Deque using Singly-Linked List



Insertion at the Front	--	O(1)
Insertion at the Rear	--	O(1)
Deletion from the Front	--	O(1)
Deletion from the Rear	--	O(n)

Solution – Use doubly-linked List

Deque using Doubly-Linked List



Insertion at the Front	--	O(1)
Insertion at the Rear	--	O(1)
Deletion from the Front	--	O(1)
Deletion from the Rear	--	O(1)

APPLICATION OF QUEUE

- There are numerous applications of queue in computer science.
- Various real-life applications, like railway ticket reservation, banking system are implemented using queue.
- One of the most useful applications of queue is in **simulation**.
- Another application of queue is in operating system to implement various functions like **CPU scheduling** in multiprogramming environment, **device management** (printer or disk), etc.
- Besides these, there are several algorithms like **level-order traversal of binary tree**, **breadh-first search in graphs**, etc., that use queues to solve the problems efficiently.

Simulation

- Simulation is the process of modelling a real-life situation through a computer program.
- Its main use is to study a real-life situation without actually making it to occur.
- It is mainly used in areas like military, scientific research operations where it is expensive or dangerous to experiment with the real system.
- In simulation, corresponding to each object and action, there is counterpart in the program.
- The objects being studied are represented as data and action as operations on the data.
- By supplying different data, we can observe the result of the program.
- If the simulation is accurate, the result of the program represents the behaviour of the actual system accurately.

Simulation(Cont.)

- Consider a ticket reservation system having four counters.
- If a customer arrives and a counter is free then the customer will get the ticket immediately.
- However, it is not always possible that counter is free. In that case, a new customer goes to the queue having less number of customers.
- Assume that the time required to issue the ticket is t. Then the total time spent by the customer is equal to the time t (time required to issue the ticket) plus the time spent waiting in line.
- The average time spent in the line by the customer can be computed by a program simulating the customer action.
- This program can be implemented using queue, since while one customer is being serviced, others keep on waiting.

CPU Scheduling in Multiprogramming Environment

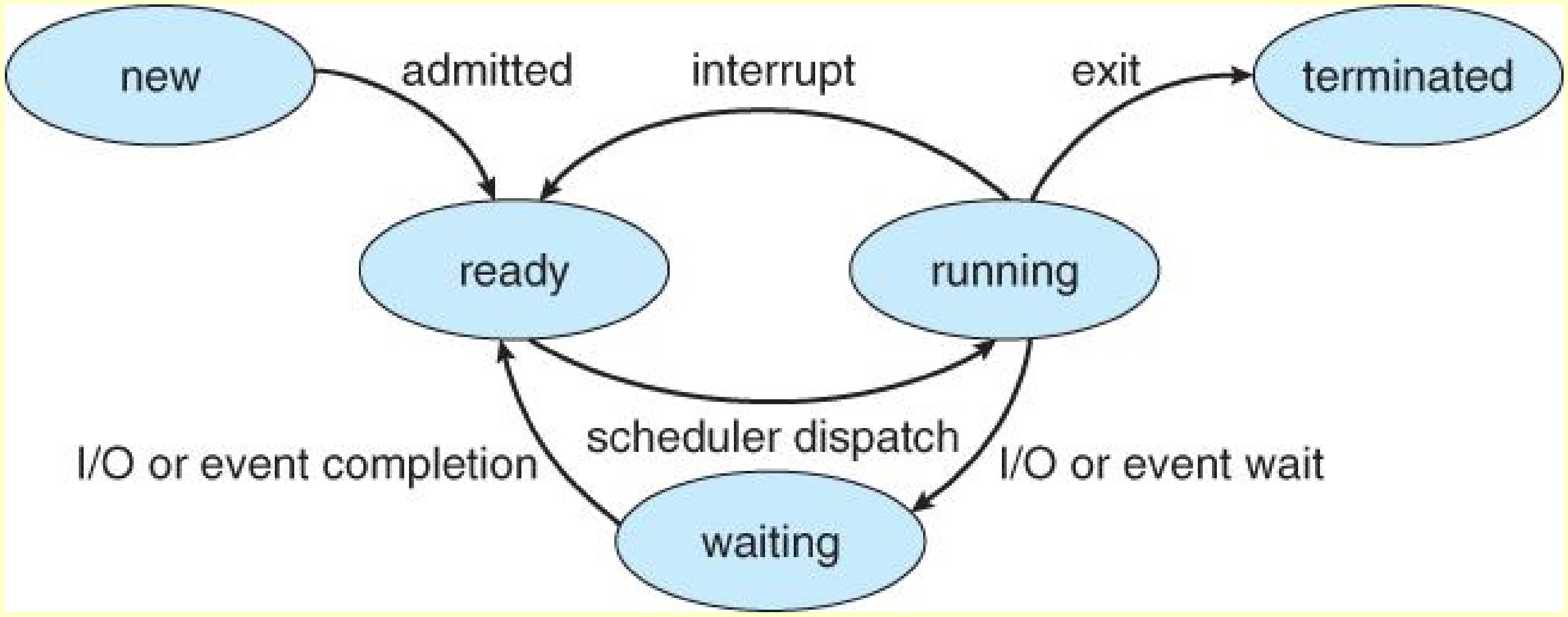
- In multiprogramming environment, multiple processes run concurrently to increase CPU utilization.

- All the processes that are residing in memory and are ready to execute are kept in a list referred as **ready queue**. It is the job of scheduling algorithm to select a process from the processes and allocate the CPU to it.

- Let us consider a multiprogramming environment where the processes are classified into three different groups, namely, *system processes, interactive processes* and b*atch processes*.

- To each group of process, some priority is associated. The system processes have the highest priority whereas the batch processes have the least priority.

Process state diagram

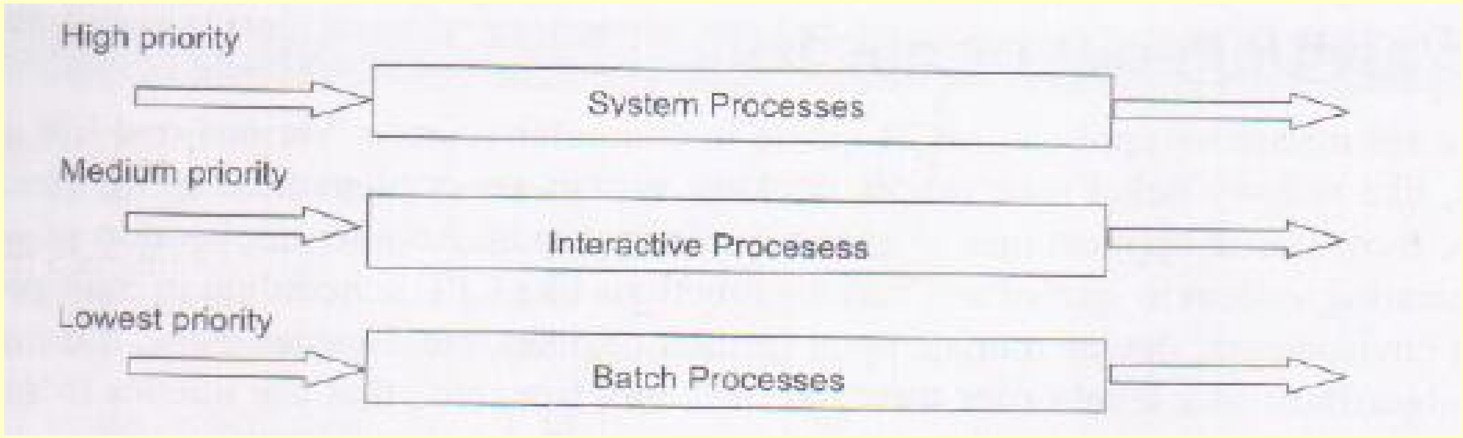


When a process is loaded in memory, it becomes ready for getting CPU for execution.

OS maintains a ready queue for all ready processes.

CPU Scheduling in Multiprogramming Environment

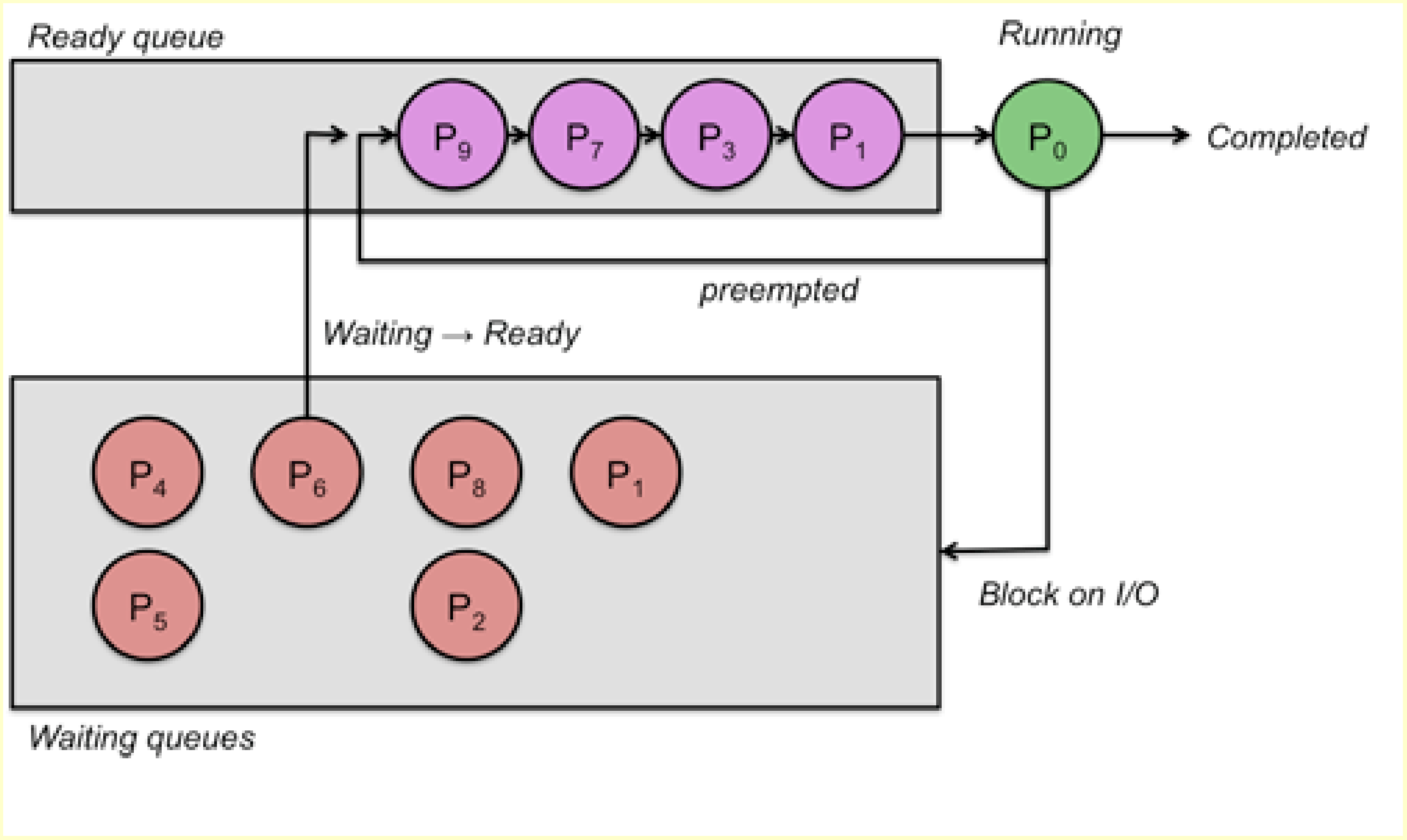
- To implement mul-tiprogramming environment, **multi-level queue** scheduling algorithm is used.
- In this algorithm, the ready queue is partitioned into multiple queues.
- The processes are assigned to the respective queues.
- The higher priority processes are executed before the lower priority processes. For example, no batch process can run unless all the system processes and interactive processes arc executed.
- If a batch process is running and a system process enters the queue, then batch process would be pre-empted to execute this system process.



Round Robin algorithm

- The round robin algorithm is one of the CPU scheduling algorithms designed for the time-sharing systems.
- In this algorithm, the CPU is allocated to a process for a small time interval called **time quantum** (generally from 10 to 100 milliseconds).
- Whenever a new process enters, it is inserted at the end of the ready queue. The CPU scheduler picks the first process from the ready queue and processes it until the time quantum elapsed.
- Then CPU switches to the next process in the queue and first process is inserted at the end of the queue if it has not been finished.
- If the process is finished before the time quantum, the process itself will release the CPU voluntarily and the process will be deleted from the ready queue.
- This process continues until all the processes are finished.
- When a process is finished, it is deleted from the queue. To implement the round robin algorithm, a **circular queue** can be used.

Round-Robin Scheduling


Ready queue
Running
P9
P7
P3
P1
P0
Completed
preempted
Waiting → Ready
P4
P6
P8
P1
P5
P2
Block on I/O
Waiting queues