QUEUE
(FIFO)

Presented By
Avani Sharma
Assistant Professor
JECRC Foundation

Course Outcomes

- Apply the concepts of data structure, data type and array data structure and analyze the algorithms and determine their time complexity.
- Apply various data structure such as stacks, Linked List, queues, trees and graphs to solve various computing problems using C-programming language.
- Implement standard algorithms for searching & sorting and identify when to choose which technique.
- Apply the data structure that efficiently models the information in a problem.
-

Introduction of Queue

- Queue is a linear data structure
- In a Queue data structure elements are inserted from the end and deleted from the start that's why it is called **FIFO** (First In First Out) representation.
- Start of queue is called **Front** and End of the queue is called **Rear.** So we can say, Insert from the rear and delete from the end.

Examples

- Remove a person from the queue

Bus Stop Queue

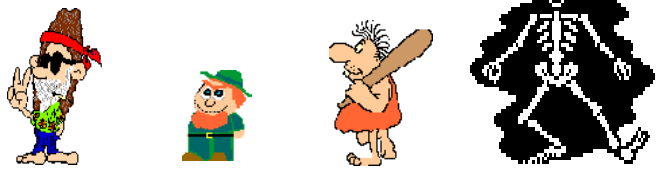
front

rear

Bus Stop Queue

front rear

Bus Stop Queue



front rear

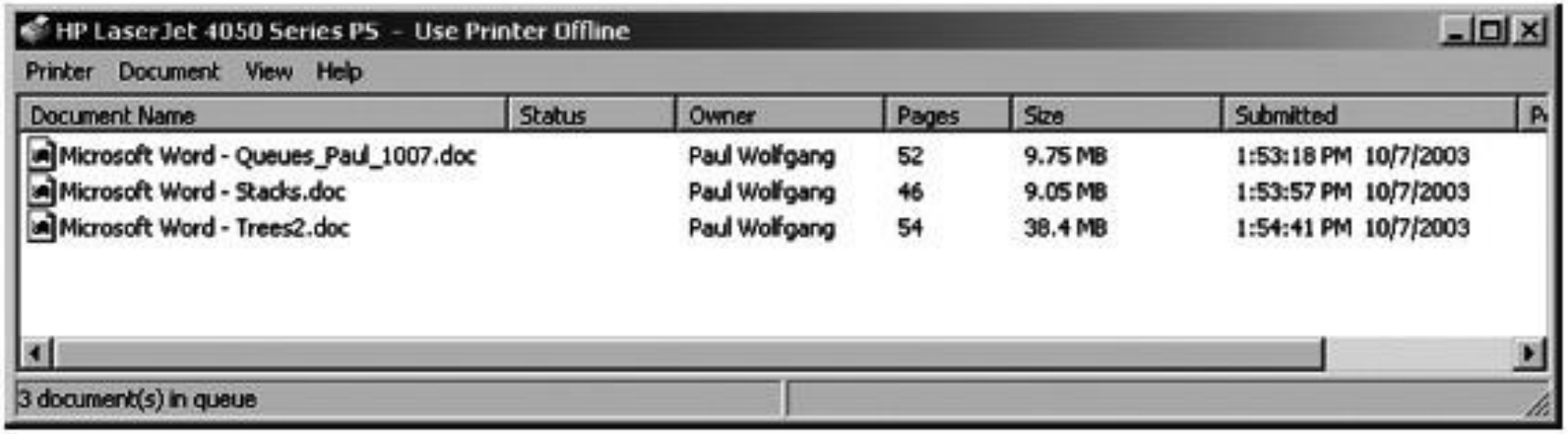
- Add a person to the queue
- A queue is a FIFO (First-In, First-Out) list.

Print Queue

Print queue: printing is much slower than the process of selecting pages to print and so a queue is used

FIGURE 6.2

A Print Queue in the Windows Operating System

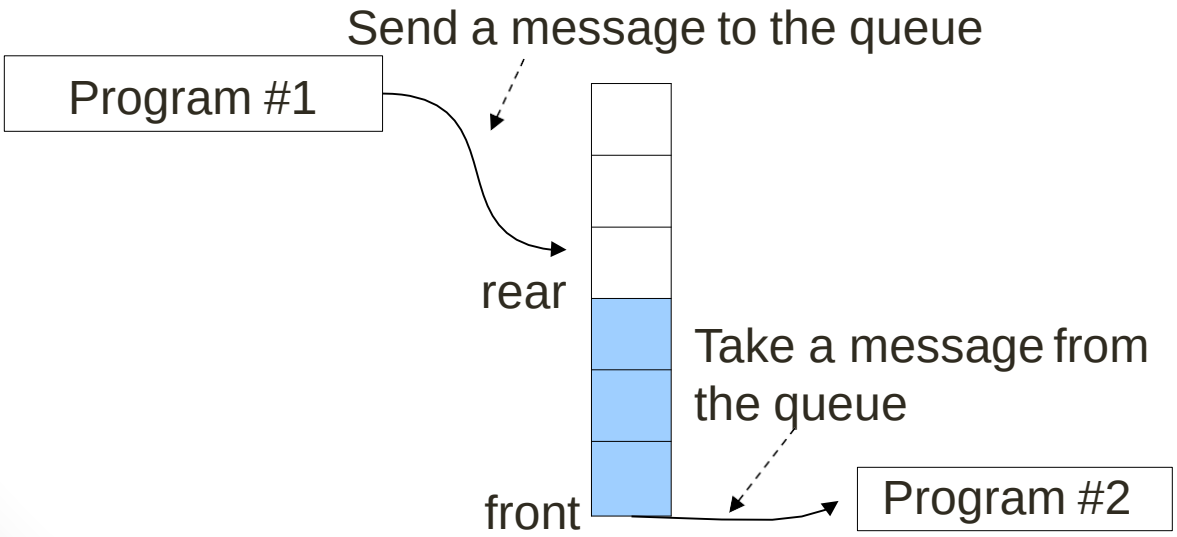


Why not Stack for print?

- Stacks are last-in, first-out (LIFO)
- The most recently selected document would be the next to print
- Unless the printer stack is empty, your print job may never get executed if others are issuing print jobs.

Message queues in an operating system

- There are times that programs need to communicate with each other. Unix operating system provides message queue as one of the mechanisms to facilitate communication.



Exercise

- What would be the contents of queue Q after the following code is executed and the following data are entered in the queue?

```
loop (not end of file)
   read number
   if (number not 0)
        enqueue(Q, number)
   else
        dequeue(Q)
   end if
end loop
```

Data:

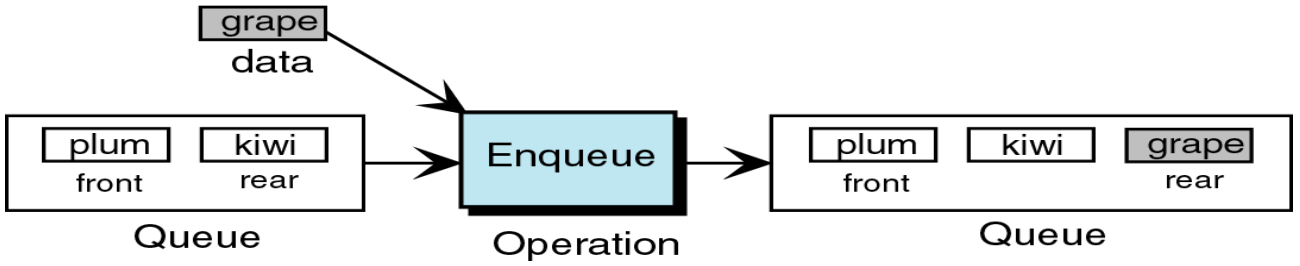
5, 7, 12, 4, 0, 4, 8, 67, 23, 0, 44, 33, 6, 0

Output :

4,4,8,67,23,44,33,
6,

Queue Operations

- **Enqueue**: Insertion of an element at rear of queue
- **Dequeue:** deletion of an element at the front of queue
- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.



Queue Operations

***peek():** This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

Algorithm:

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek() function in C programming language:

```
int peek() {
   return queue[front];
}
```

Queue Operations

***isfull() (Overflow Condition)**

Algorithm:

```
begin procedure isfull
   if rear equals to MAXSIZE
      return true
   else
      return false
   endif
end procedure
```

Implementation of isfull() function in C programming language:

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

Queue Operations

*isempty() (Underflow Condition)

Algorithm:

```
begin procedure isempty
 if front is less than MIN  OR front is greater than rear
     return true
   else
     return false
   endif
end procedure
```

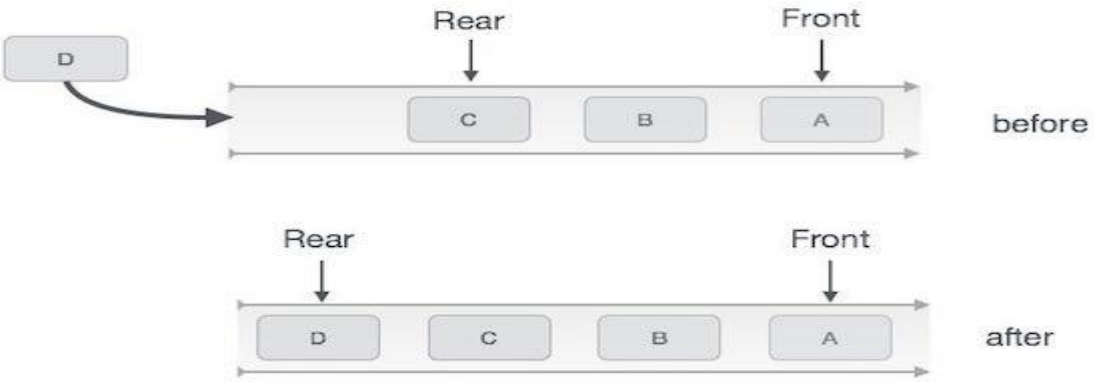
Implementation of peek() function in C programming language:

```
bool isempty() {
   if(front <0 || front > rear)
      return true;
   else
      return false;
}
```

Queue Operations

***Enqueue Operation:**

- **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 Enqueue

Queue Operations

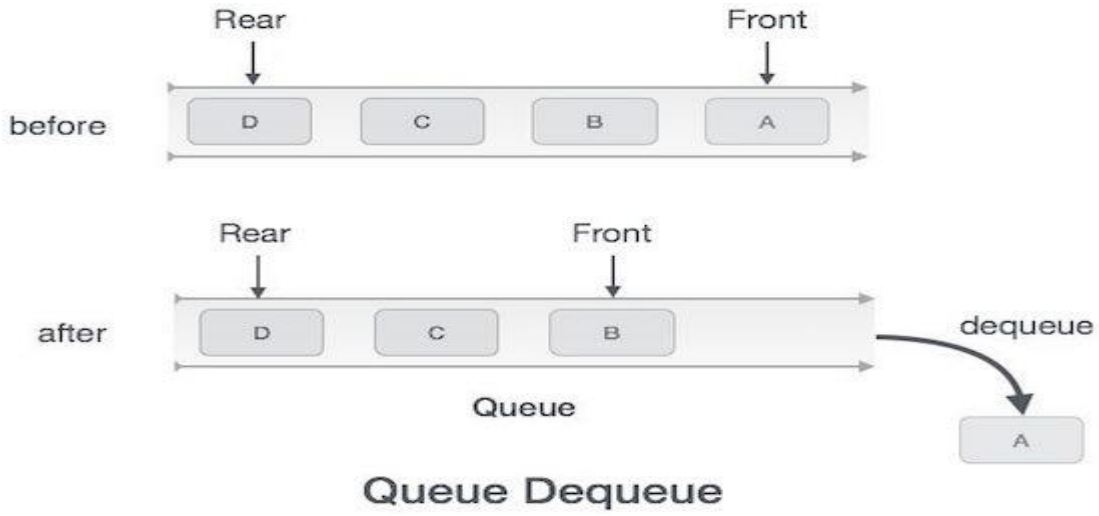
*__Enqueue Operation:__

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

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Queue Operations

*Dequeue Operation:

- **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 Dequeue

Queue Operations

*__Dequeue Operation:__

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

   int data = queue[front];
   front = front + 1;

   return data;
}
```

Queue Implementation: Using Array

```
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;

void main()
{ int value, choice;
  clrscr();
  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
   case 1: printf("Enter the value to be insert: ");
                scanf("%d",&value);
                enQueue(value);
                break;
   case 2: deQueue();
                break;
   case 3: display();
                break;
   case 4: exit(0);
   default: printf("\nWrong selection!!! Try again!!!");
    }
  }}
```

Queue Implementation: Using Array

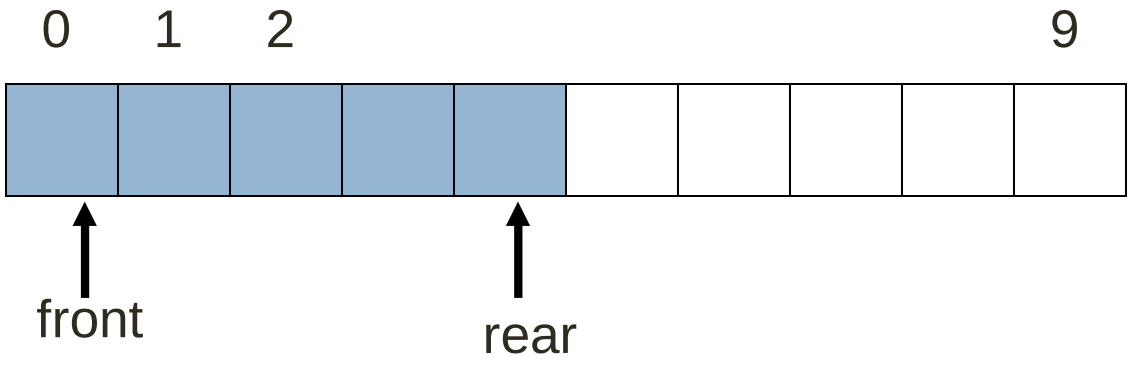
```
void enQueue(int value){
  if(rear == SIZE-1)
    printf("\nQueue is Full!!! Insertion is not
      possible!!!");
  else
  {
    rear++;
    queue[rear] = value;
    printf("\nInsertion success!!!");
   }
}
```

```
void deQueue(){
  if(front < 0 || front > rear)
    printf("\nQueue is Empty!!! Deletion is not
      possible!!!");
  else{
    printf("\nDeleted : %d", queue[front]);
    front++;
     }
}
```

```
void display(){
  if(rear == -1)
    printf("\nQueue is Empty!!!");
  else{
    int i;
    printf("\nQueue elements are:\n");
    for(i=front; i<=rear; i++)
        printf("%d\t",queue[i]);
  }
}
```

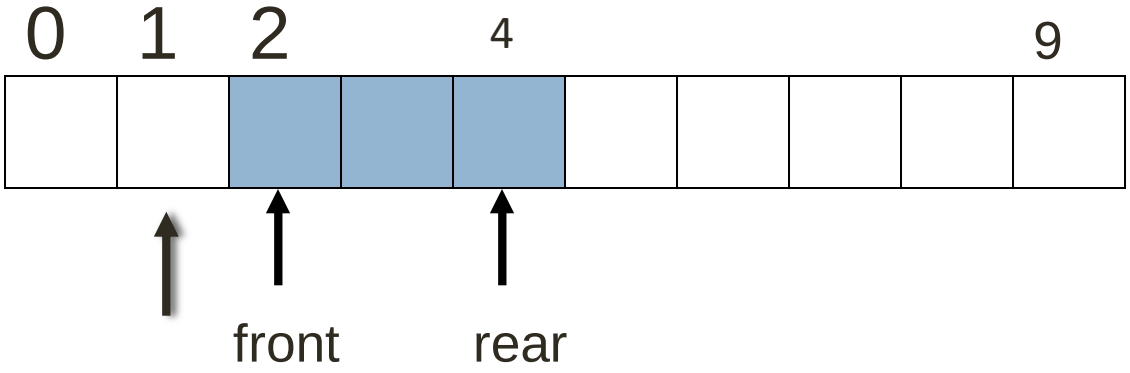
Queue Implementation Issues

- Consider a queue implemented in an array of size 10
- After 5 enqueue operations, the shaded portion of the array has been filled:



Queue Implementation Issues

- After two dequeue operations, if array contents are not shifted, we have:



- The queue capacity is now reduced to 8 items

Queue Implementation Issues

- Option 1: using an underlying array: enhance the enqueue operation to increase the size of the underlying array whenever there is an item in the final array location:
- Dequeue is a fast operation
- Enqueue is fast except when the array needs replacing could use a large amount of space: once an item is removed, its array location is never reused

Queue Implementation Issues

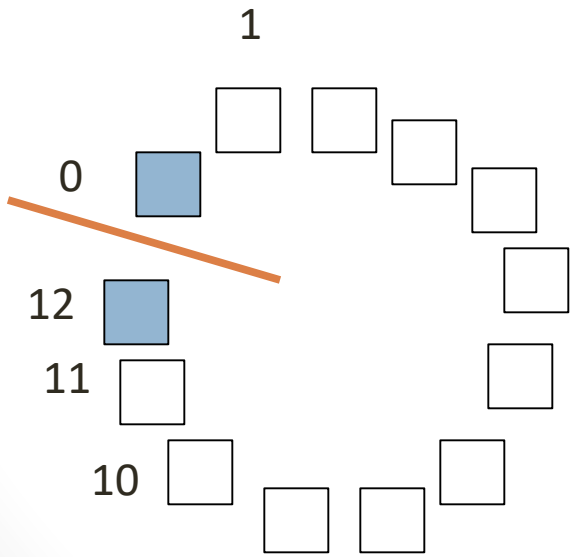
- Option 2
  - Shift all the remaining items at each dequeue operation

Circular Queues

- We can implement fixed-capacity queues using arrays so that the queue capacity never decreases: use circular queues
- Concept can be extended so that the underlying array can be replaced by a larger array if necessary
- **A circular queue is a type of queue in which the last position is connected to the first position to make a circle.**

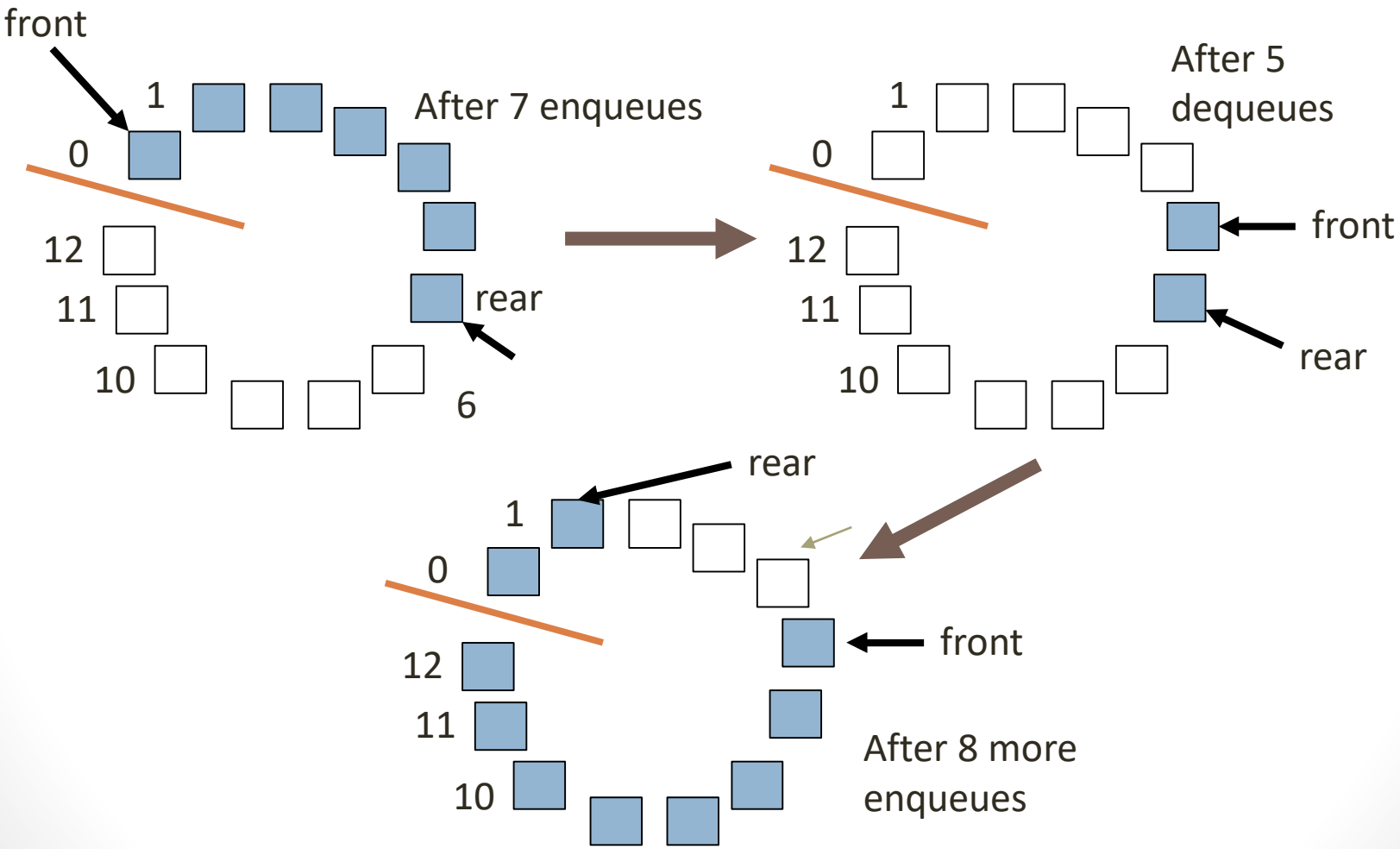
Circular Queues

- Treat the underlying array as if it is circular: in an n-element array, the location "before" index 0 is index n-1; the location "after" index n-1 is index 0



- Use modular arithmetic to compute next and previous indices

Circular Queues



Circular Queues: Operations

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value):** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

  Steps:

  1. Check whether queue is Full
     if((rear == SIZE-1 && front == 0) || (front == rear+1))
  2. If it is full then display Queue is full. If queue is not full then step 3
  3. Check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

Circular Queues: Operations

- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

  Steps:

  1. Check whether queue is Empty means check (front==-1).
  2. If it is empty then display Queue is empty. If queue is not empty then step 3
  3. Check if (front==rear) if it is true then set front=rear= -1 else step 4
  4. Check if (front==size-1), if it is true then set front=0 and return the element.

Circular Queues: Applications

- CPU scheduling
- Memory management
- Traffic Management

1

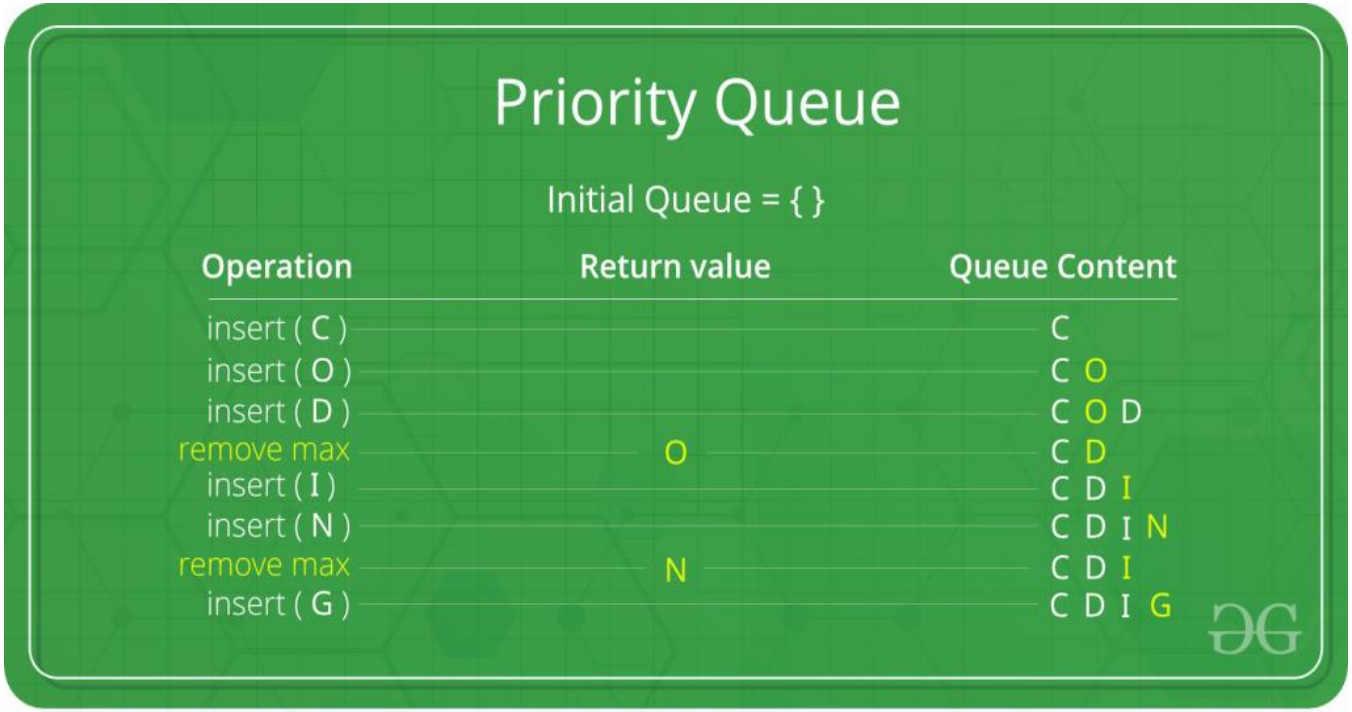
Priority Queues

Priority Queue is an extension of queue with following properties:

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Priority Queues

In the below priority queue, element with maximum ASCII value will have the highest priority.



Priority Queue: Operations

In the below priority queue, element with maximum ASCII value will have the highest priority.

- **insert(item, priority):** Inserts an item with given priority.
- **getHighestPriority():** Returns the highest priority item.
- **deleteHighestPriority():** Removes the highest priority item.
- **Peek** – get the element at front of the queue.
- **isFull** – check if queue is full.
- **isEmpty** – check if queue is empty.

Priority Queue Implementation

- Using Array
- Using Linked List
- Using Heap

Priority Queue Implementation: Linked List

Algorithm :

PUSH(HEAD, DATA, PRIORITY)
Step 1: Create new node with DATA and PRIORITY
Step 2: Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.
Step 3: NEW -> NEXT = HEAD
Step 4: HEAD = NEW
Step 5: Set TEMP to head of the list
Step 6: While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY
Step 7: TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: NEW -> NEXT = TEMP -> NEXT
Step 9: TEMP -> NEXT = NEW
Step 10: End

Priority Queue Implementation: Linked List

Algorithm :

POP(HEAD)
Step 1: Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.
Step 2: Free the node at the head of the list
Step 3: End

PEEK(HEAD):
Step 1: Return HEAD -> DATA
Step 2: End

Priority Queue Implementation: Linked List

```
typedef struct node {
    int data;
    int priority; // Lower values indicate higher priority
    struct node* next;
} Node;


Node* newNode(int d, int p)
{
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = d;
    temp->priority = p;
    temp->next = NULL;
    return temp;
}
```

Priority Queue Implementation: Linked List

```
int peek(Node** head)
{
   return (*head)->data;
}

// Removes the element with the highest priority form the list

void pop(Node** head)
{
   Node* temp = *head;
   (*head) = (*head)->next;
   free(temp);
}
```

Priority Queue Implementation: Linked List

```
int peek(Node** head)
{
   return (*head)->data;
}

// Removes the element with the
// highest priority form the list

void pop(Node** head)
{
   Node* temp = *head;
   (*head) = (*head)->next;
   free(temp);
}
```

Priority Queue Implementation: Linked List

```
void push(Node** head, int d, int p)
{
    Node* start = (*head);
    Node* temp = newNode(d, p); // Create new Node
    // Special Case: The head of list has lesser priority than new node. So insert new
    node before head node and change head node.
    if ((*head)->priority < p) {
        temp->next = *head; // Insert New Node before head
        (*head) = temp; }
    else {
        // Traverse the list and find a position to insert new node
        while (start->next != NULL &&
               start->next->priority < p) {
            start = start->next;
        }
        // Either at the ends of the list or at required position
        temp->next = start->next;
        start->next = temp; }}
```

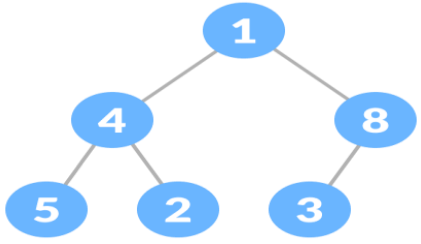
Priority Queue Implementation: Linked List

```
int isEmpty(Node** head)
{
    return (*head) == NULL;
}
int main()
{
    // Create a Priority Queue 7->4->5->6
    Node* pq = newNode(4, 1);
    push(&pq, 5, 2);
    push(&pq, 6, 3);
    push(&pq, 7, 0);
    while (!isEmpty(&pq)) {
        printf("%d ", peek(&pq));
        pop(&pq);
    }
    return 0;
}
```

Priority Queue using Heap

- Heap data structure is a **complete binary tree** that satisfies **the heap property**. It is also called as **a binary heap**.

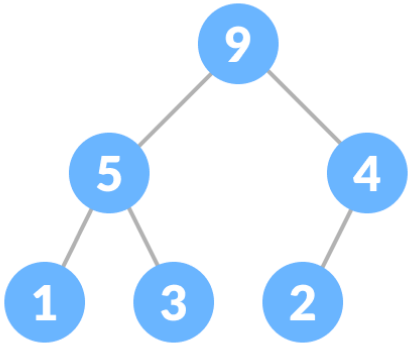
- A complete binary tree is a special binary tree in which
- every level, except possibly the last, is filled
- all the nodes are as far left as possible



Priority Queue using Heap

Heap Property is the property of a node in which

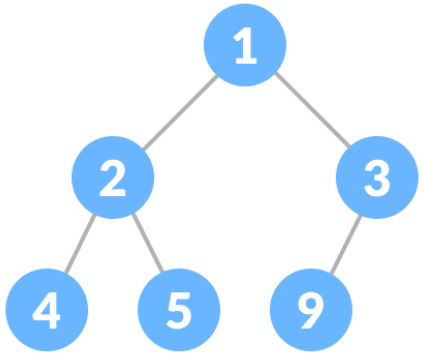
- (for max heap) key of each node is always greater than its child node/s and the key of the root node is the largest among all other nodes;



Priority Queue using Heap

Heap Property is the property of a node in which

- (for min heap) key of each node is always smaller than the child node/s and the key of the root node is the smallest among all other nodes.

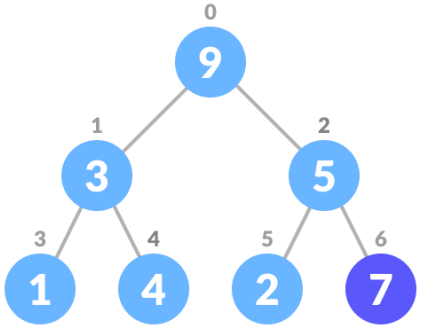


Priority Queue using Heap

Inserting an Element from the Priority Queue

Inserting an element into a priority queue (max-heap) is done by the following steps.

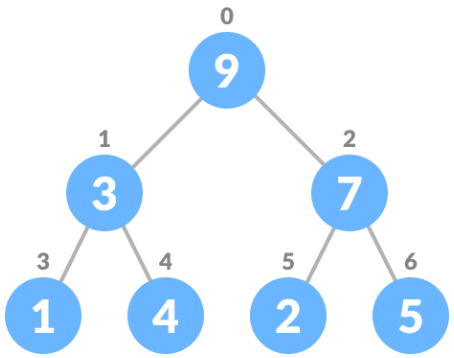
a) Insert the new element at the end of the tree



Priority Queue using Heap

Inserting an Element from the Priority Queue

b) Heapify the tree.



Priority Queue using Heap

Inserting an Element from the Priority Queue

If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

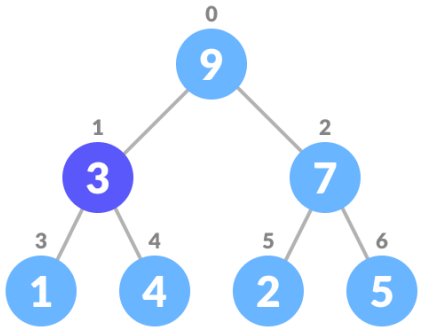
heapify the array

Priority Queue using Heap

Deleting an Element from the Priority Queue

Deleting an element from a priority queue (max-heap) is done as follows:

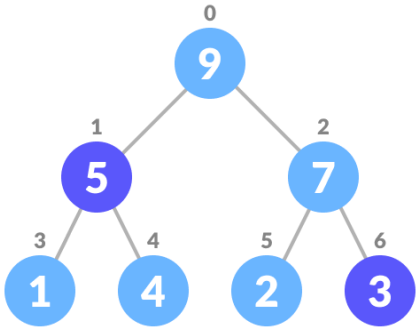
a) Select the element to be deleted.



Priority Queue using Heap

Deleting an Element from the Priority Queue

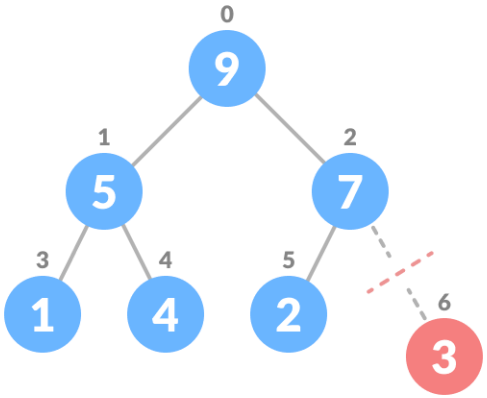
b) Swap it with the last element.



Priority Queue using Heap

Deleting an Element from the Priority Queue

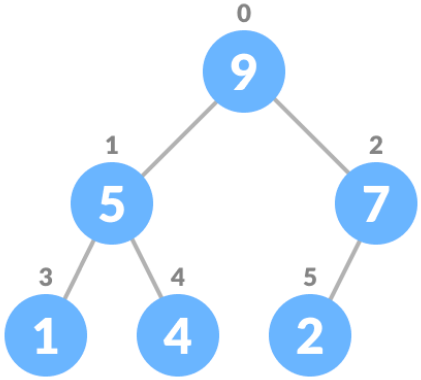
c) Remove the last element



Priority Queue using Heap

Deleting an Element from the Priority Queue

d) Heapify the tree



Priority Queue using Heap

Deleting an Element from the Priority Queue

Algorithm

If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array

Priority Queue: Applications

- CPU Scheduling
- Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc.
- All queue applications where priority is involved.

Implementation of queue using stacks

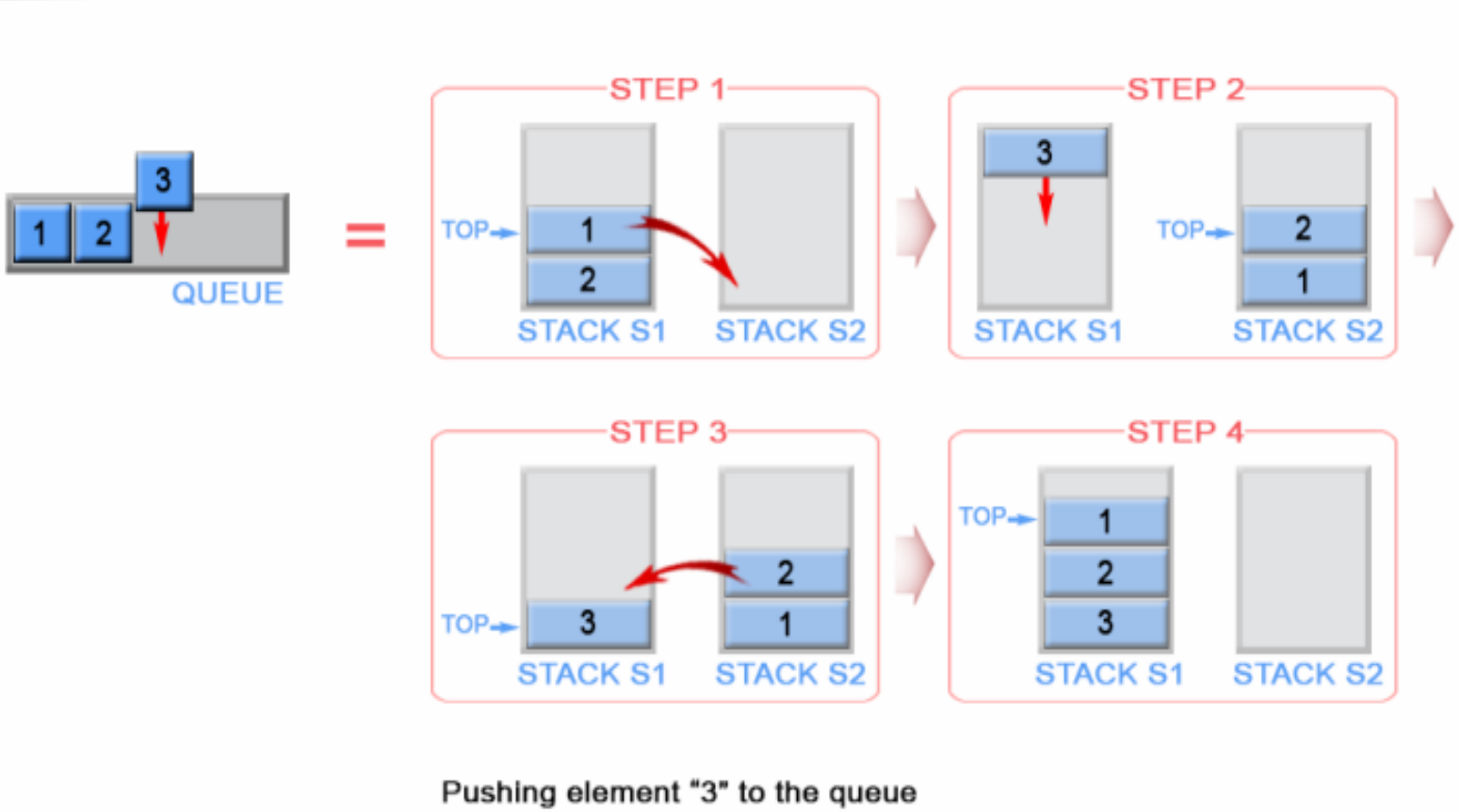
Implementation of queue can be done using two stacks. Queue using stack can be done in two ways.

- By making the enqueue (insertion) operation costly.
- By making the dequeue (deletion) operation costly.

Queue using stacks by making enqueue operation costly

Implementing queue using stacks by making the enqueue operation costly is discussed below.

- Initialize stack1 and stack2 making **top1 = top2 = -1**.
- To perform an enqueue operation,
  - Push all the elements from stack1 to stack2.
  - Push the new element to stack2.
  - Pop all the elements from stack2 to stack1.
- To perform dequeue operation,
  - Pop and return the element from stack1.

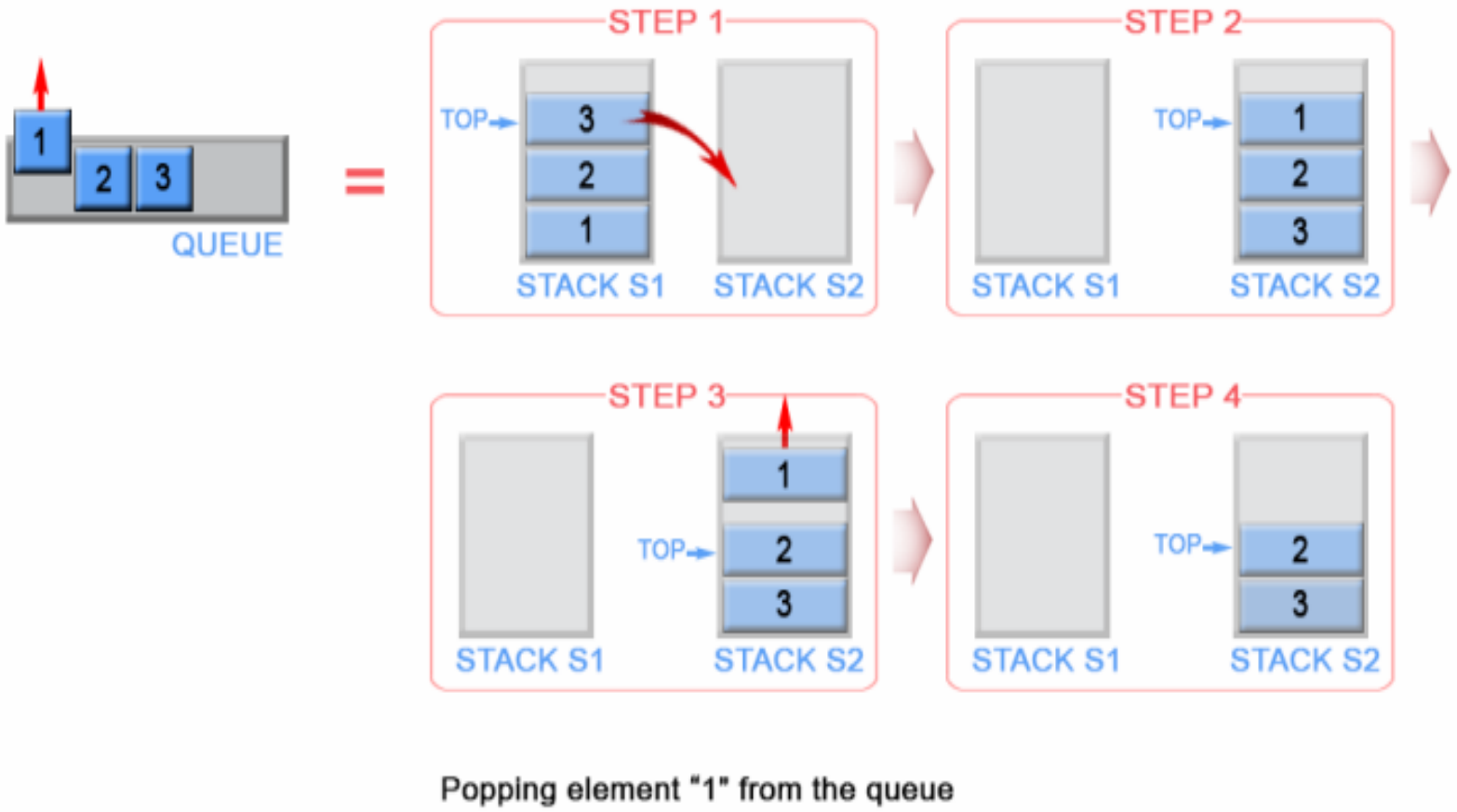


Pushing element "3" to the queue

Queue using stacks by making dequeue operation costly

Implementing queue using stacks by making the dequeue operation costly is discussed below.

- Initialize stack1 and stack2 making **top1 = top2 = -1.**
- To perform an enqueue operation,
  - Push the element to be added to stack1.
- To perform dequeue operation,
  - Push all the elements from stack1 to stack2.
  - Pop and return the element from the stack2.



Popping element "1" from the queue

Queues vs. Stacks

- Stacks are a LIFO container =>store data in the reverse of order received
- Queues are a FIFO container =>store data in the order received
- Stacks then suggest applications where some sort of reversal or unwinding is desired.
- From an ordering perspective, then, Queue are the “opposite” of stacks
- Easy solution to the palindrome problem

Application of Queue: Round Robin Algorithm

- Round Robin Scheduling is a scheduling algorithm used by the system to schedule CPU utilization.

- There exist a fixed time slice associated with each request called **the quantum**. The job scheduler saves the progress of the job that is being executed currently and moves to the next job present in the queue when a particular process is executed for a given time quantum.

Application of Queue: Round Robin Algorithm

ROUND ROBIN SCHEDULING ALGORITHM

- We first have **a queue** where the processes are arranged in first come first serve order.
- A quantum value is allocated to execute each process.
- The first process is executed until the end of the quantum value. After this, an interrupt is generated and the state is saved.
- The CPU then moves to the next process and the same method is followed.
- Same steps are repeated till all the processes are over.

Application of Queue: Round Robin Algorithm

Round Robin Example:

Process	Duration	Order	Arrival Time
P1	3	1	0
P2	4	2	0
P3	3	3	0

Suppose time quantum is 1 unit.

P1	P2	P3	P1	P2	P3	P1	P2	P3	P2

0 10

P1 waiting time : 4

P2 waiting time: 6

P3 waiting time: 6

The average waiting time(AWT) : (4+6+6)/3=5.33