Unit 2: Queues and Linked List

Queues:

A **queue** is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the delete operation is first performed on that.

Characteristics of Queue:

- Queue can handle multiple data.
- We can access both ends.
- They are fast and flexible.

Applications of Queue:

Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kinds 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.
- Queue can be used as an essential component in various other data structures.

Basic Operations in Queue

Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.

The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the queue.

Queue uses two pointers − **front** and **rear**. The front pointer accesses the data from the front end (helping in dequeuing) while the rear pointer accesses data from the rear end (helping in enqueueing).

Queue Insertion Operation: Enqueue()

The *enqueue()* is a data manipulation operation that is used to insert elements into the queue. The following algorithm describes the enqueue() operation in a simpler way.

1. START
2. Check if the queue is full.
3. If the queue is full, produce overflow error and exit.
4. If the queue is not full, increment the rear pointer to point to the next empty space.
5. Add a data element to the queue location, where the rear is pointing.
6. return success.
7. END

Queue Deletion Operation: dequeue()

The *dequeue()* is a data manipulation operation that is used to remove elements from the queue. The following algorithm describes the dequeue() operation in a simpler way.

1. START
2. Check if the queue is empty.
3. If the queue is empty, produce an underflow error and exit.
4. If the queue is not empty, access the data where the front is pointing.
5. Increment front pointer to point to the next available data element.
6. Return success.
7. END

Queue - The peek() Operation

The peek() is an operation which is used to retrieve the frontmost element in the queue, without deleting it. This operation is used to check the status of the queue with the help of the pointer.

1. START
2. Return the element at the front of the queue
3. END

Front: Getting the front element of the queue without removing it.
Rear: Getting the rear element of the queue without removing it.
IsEmpty: Checking if the queue is empty.
IsFull: Checking if the queue is full (applicable to a fixed-size array-based implementation)

How to implement Queue using Array?

To implement a queue using an array,

- create an array arr of size **n** and
- take two variables **front** and **rear** both of which will be initialized to -1 which means the queue is initially empty.
- Element
    - rear is the index up to which the elements are stored in the array and
    - front is the index of the first element of the array.

Now, some of the implementations of queue operations are as follows:

- **Enqueue:** Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If *rear* < *n* which indicates that the array is not full then store the element at arr[rear] and increment rear by 1 but if rear == n then it is said to be an Overflow condition as the array is full.
- **Dequeue:** Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. *rear* > *0*. Now, the element at arr[front] can be deleted but all the remaining elements have to shift to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.
- **Front:** Get the front element from the queue i.e. arr[front] if the queue is not empty.
- **Display:** Print all elements of the queue. If the queue is non-empty, traverse and print all the elements from the index front to rear.

How to implement Queue using Stacks?

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly): This method makes sure that the oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

enQueue(q, x):

- While stack1 is not empty, push everything from stack1 to stack2.
- Push x to stack2 (assuming size of stacks is unlimited).
- Push everything back to stack1.

Here time complexity will be O(n)

deQueue(q):

- If stack1 is empty then error
- Pop an item from stack1 and return it

Method 2 (By making deQueue operation costly): In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q, x)
 1) Push x to stack1 (assuming size of stacks is unlimited).
Here time complexity will be O(1)

deQueue(q)
 1) If both stacks are empty then error.
 2) If stack2 is empty
     While stack1 is not empty, push everything from stack1 to stack2.
 3) Pop the element from stack2 and return it.
Here time complexity will be O(n)

Method 2 is definitely better than method 1.

Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty.

Applications of Queues-
Round Robin Algorithm
Circular Queues,
DeQueue
Priority Queues.

Let's delve into these concepts, which are fundamental in computer science, particularly in the context of scheduling algorithms and data structures.

Round Robin Algorithm

The Round Robin (RR) algorithm is a preemptive scheduling algorithm used primarily in operating systems and networks. It's designed to allocate a fixed time slot or quantum to every process in the queue, in a cyclic order. One of the main advantages of the Round Robin algorithm is its fairness, ensuring that no process is starved, as each gets an equal share of the CPU time in turns. It's widely used in time-sharing systems. The key challenge in implementing an efficient Round Robin scheduler lies in choosing the optimal time quantum; too short leads to excessive context switching, and too long may cause it to behave like FIFO (First-In, First-Out), diminishing its responsiveness benefits.

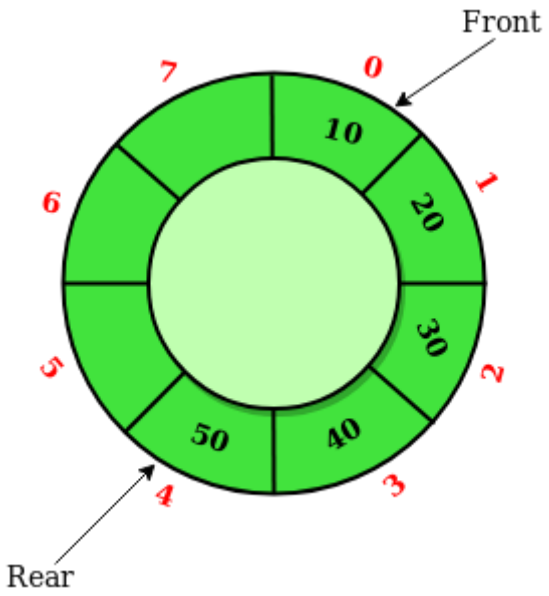
Circular Queues

What is a Circular Queue?

> A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle. It efficiently

utilizes the storage space by reusing the positions of elements that have been dequeued. Circular queues are especially useful in situations where the queue needs to be reset frequently, as it avoids the overhead of shifting elements. This structure is often used in buffering data streams, where a fixed-size buffer is continuously reused.

The operations are performed based on FIFO (First In First Out) principle. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

Operations on Circular Queue:

- **Front:** Get the front item from the queue.
- **Rear:** Get the last item from the 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 the rear position.
    - Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
    - If it is full then display Queue is full.
        - If the queue is not full then, insert an element at the end of the queue.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.
    - Check whether the queue is Empty.
    - If it is empty then the display Queue is empty.
        - If the queue is not empty, then get the first element and remove it from the queue.

Implement Circular Queue using Array:

1. Initialize an array queue of size **n**, where n is the maximum number of elements that the queue can hold.
2. Initialize two variables front and rear to -1.
3. **Enqueue:** To enqueue an element **x** into the queue, do the following:
    - Increment rear by 1.
        - If **rear** is equal to n, set **rear** to 0.
    - If **front** is -1, set **front** to 0.
    - Set queue[rear] to x.
4. **Dequeue:** To dequeue an element from the queue, do the following:
    - Check if the queue is empty by checking if **front** is -1.
        - If it is, return an error message indicating that the queue is empty.
    - Set **x** to queue[front].
    - If **front** is equal to **rear**, set **front** and **rear** to -1.
    - Otherwise, increment **front** by 1 and if **front** is equal to n, set **front** to 0.
    - Return x.

Applications of Circular Queue:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic systems, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

DeQueue (Double-Ended Queue)

Deque is a more generalized version of a linear queue. As you know, the linear queue has some restrictions while performing the insertion and deletion of elements. The insertion in a linear queue must happen from the rear end and deletion from the front end. But, in deque, you can perform both insertion and deletion operations at both of its ends. That's why it is called a Double-Ended Queue (Deque).

This flexibility makes it a powerful data structure, useful in scenarios where elements need to be frequently added or removed from either end, such as in certain caching algorithms. DeQueues can be implemented using various underlying structures like arrays or linked lists, each with its trade-offs in terms of performance for different operations.

Applications of Deque in Data Structure

- Palindrome Checker
- Multiprocessor Scheduling

Operations on Deque

Four basic operations are performed on deque, they are as follows:

- Insertion at Rear
- Insertion at Front
- Deletion at Front
- Deletion at Rear

Along with these primary operations, you can also perform isEmpty(), isFull() and Peek() operations. These operations are called supportive queue operations.

Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.

Output restricted Queue

In the output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

Priority Queues

A Priority Queue is an abstract data type in which each element is associated with a priority, and elements are served based on their priority (not just their order in the queue). If two elements have the same priority, they are served according to their order in the queue. Priority queues are commonly implemented with heaps, which allows for efficient retrieval of the highest (or lowest) priority element. This structure is essential in numerous applications, such as scheduling tasks, algorithms like Dijkstra's shortest path, and anytime we need to dynamically fetch elements by their priority rather than just their insertion order.

The hospital emergency queue is an ideal real-life example of a priority queue. In this queue of patients, the patient with the most critical situation is the first in a queue, and the patient who doesn’t need immediate medical attention will be the last. In this queue, the priority depends on the medical condition of the patients.

Properties of Priority Queue

So, a priority Queue is an extension of the queue with the 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.

How is Priority assigned to the elements in a Priority Queue?

In a priority queue, generally, the value of an element is considered for assigning the priority.

For example, the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority. The reverse case can also be used i.e., the element with the lowest value can be assigned the highest priority. Also, the priority can be assigned according to our needs.

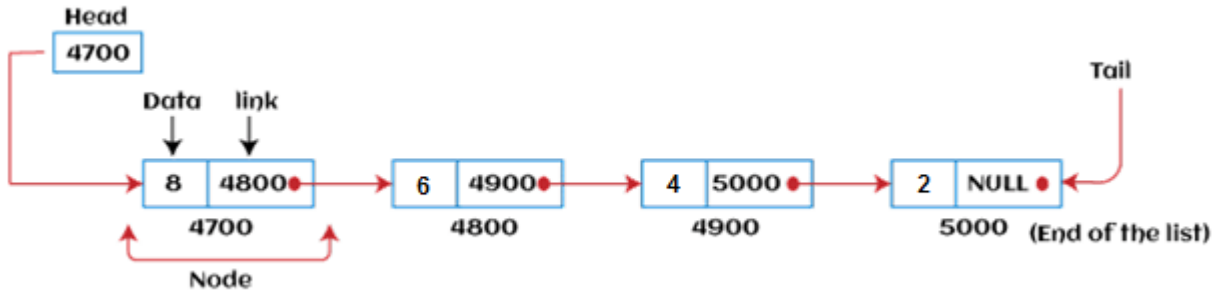
Each of these concepts—Round Robin Algorithm, Circular Queues, DeQueues, and Priority Queues—plays a crucial role in different areas of computer science and software engineering, particularly in the design and implementation of efficient algorithms, operating systems, and application systems. Understanding these concepts is fundamental for designing systems that require efficient data handling and scheduling capabilities.

Linked list

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked list

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



Till now, we have been using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages that must be known to decide the data structure that will be used throughout the program.

Now, the question arises why we should use linked list over array?

Why use linked lists over arrays?

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be continuously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- In the linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and is limited to the available memory space.

How to declare a linked list?

It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type **structure.**

The declaration of node in singly linked list is given as follows -

```
struct node

{

    int data;

    struct node *next;
```

}

In the above declaration, we have defined a structure named as **node** that contains two variables, one is **data** that is of integer type, and another one is **next** that is a pointer which contains the address of the next node.

Now, let's move towards the types of linked list.

Types of Linked list

Linked list is classified into the following types -

- **Singly-linked list -** Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
- **Doubly linked list -** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).

```
struct node

{

        Struct node *prev; // link to previous node

        int data;

        struct node *next;  // link to next node

}
```

- **Circular singly linked list -** In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked lists as well as circular doubly linked lists.
- **Circular doubly linked list -** Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.