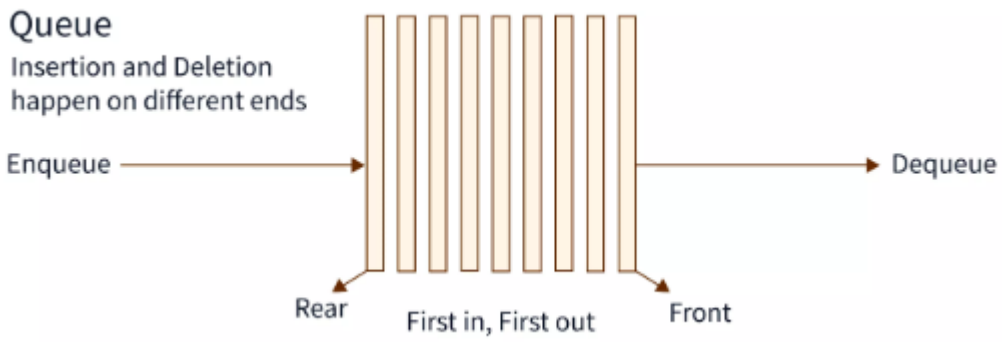
1.The Queue ADT (Abstract Data Type)

A **queue** is a data structure used to store and manage a collection of elements in a specific order. It is a linear data structure. It operates on the **First-In, First-Out (FIFO)** principle, meaning the element that is added first is the first one to be removed. This is similar to standing in a line at a checkout counter or waiting for a bus — the first person in line gets served first.



Basic Operations on a Queue

There are several basic operations that you can perform on a queue:

1. **Enqueue (Insertion)**:
    - This operation adds an element to the **rear** (or end) of the queue.
    - **Example**: queue.enqueue(10) would add the element 10 at the end of the queue.
2. **Dequeue (Removal)**:
    - This operation removes the element from the **front** of the queue.
    - **Example**: queue.dequeue() would remove the element from the front of the queue.
3. **Peek (Front)**:
    - This operation allows you to see the element at the **front** of the queue without removing it.
    - **Example**: queue.peek() would return the front element, without changing the queue.
4. **Is Empty**:
    - This operation checks if the queue is empty.
    - **Example**: queue.is_empty() returns True if the queue is empty.
5. **Size**:
    - This operation returns the number of elements currently in the queue.
    - **Example**: queue.size() returns the current size of the queue.

Why Use a Queue?

Queues are used in situations where **order** matters, and we need to process elements in the same order they were added. There are several reasons why we use queues in programming:

1. **FIFO Behavior**: Queues are perfect for situations where the first element added should be the first one to be removed.
2. **Efficient Management**: Queues allow efficient addition and removal of elements from both ends, making them ideal for tasks like scheduling, data buffering, etc.
3. **Resource Management**: In computer systems, queues help in managing resources, where each task is given attention in the order it was received.
4. **Concurrency and Multi-threading**: Queues are useful in multi-threaded environments where tasks are queued and processed by multiple workers.

Where to Use a Queue:

1. **Customer Service & Ticketing Systems:**
    - People are served in the order they arrive.
    - **Example:** Call centers or help desks, where the first customer gets served first.
2. **Print Queues:**
    - Print jobs are processed in the order they are added to the queue.
    - **Example:** Shared printers in an office setting.
3. **Job Scheduling (Operating Systems):**
    - Tasks are executed by the CPU in the order they are ready.
    - **Example:** Process scheduling in OS where processes are managed in a queue.
4. **Message Queues in Distributed Systems:**
    - Messages are queued for processing by consumers in order.
    - **Example:** Messaging systems like Kafka, RabbitMQ.
5. **Breadth-First Search (BFS) in Graphs:**
    - BFS uses a queue to explore nodes level by level.
    - **Example:** Searching through a network of connected nodes.
6. **Traffic Light Management:**
    - Vehicles wait in line at an intersection, processed in the order of arrival.
    - **Example:** Cars waiting for a green light.
7. **Real-Time Data Buffers:**
    - Data (e.g., audio, video) is buffered and processed in the order it arrives.
    - **Example:** Media streaming services.
8. **Network Packet Management:**
    - Network packets are queued and processed in sequence.

- **Example:** Internet data packets, routers, and packet-switching.

Advantages of Using a Queue

1. **FIFO Order**: Ensures that the first element to enter the queue is the first to leave.
2. **Fairness**: No element is skipped. Every element is processed in the order it arrives.
3. **Efficient in Resource Management**: Helps in handling a sequence of tasks, such as processing print jobs or scheduling CPU processes.

Disadvantages of Using a Queue

1. **Fixed Size (in some implementations)**: In some cases, queues have a fixed size, and if they become full, no new elements can be added (until space is available).
2. **Overhead**: In some queue implementations (e.g., linked lists), there may be extra overhead in terms of memory or processing time.
3. **No Random Access**: Unlike lists or arrays, queues don't allow accessing arbitrary elements in the middle. You can only access the front or rear elements.

2. Implementing a Queue: Using a Python List

Python's list can be used to implement a simple queue. However, it's not the most efficient way due to the fact that removing an element from the front of the list is an O(n) operation.

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)  # O(n) operation
        else:
            raise IndexError("Dequeue from empty queue")

    def peek(self):
        if not self.is_empty():
            return self.queue[0]
```

```
        else:
            raise IndexError("Peek from empty queue")
    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())          # Output: 1
print(q.peek())             # Output: 2
```

3. Using a Circular Array

A **circular array** is a fixed-size array where the last position is connected to the first position, making the array behave like a circle. This helps us efficiently reuse the space in the array and avoid shifting elements.

Why Circular Array for a Queue?

- A regular array implementation of a queue leads to wasted space when elements are removed, as the front moves forward but space at the beginning of the array is not reused.
- In a **circular array**, when elements are removed, the queue "wraps around" to the beginning of the array, using empty spaces and improving efficiency.

Key Operations:

- **enqueue(item)**: Adds an item to the rear of the queue.
- **dequeue()**: Removes and returns the item from the front of the queue.
- **front()**: Returns the item at the front without removing it.
- **is_empty()**: Checks if the queue is empty.
- **is_full()**: Checks if the queue is full.
- **size()**: Returns the number of elements in the queue.

How It Works :
We use two pointers, **front** and **rear**, to keep track of where to remove and add elements, respectively. When either pointer reaches the end of the array, it wraps around to the beginning.
Here's an implementation of a queue using a circular array:

```
class CircularQueue:
```

```
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0
        self.rear = 0
        self.size = 0

    def enqueue(self, item):
        if self.size == self.capacity:
            raise IndexError("Queue is full")
        self.queue[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity
        self.size += 1

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        item = self.queue[self.front]
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return item

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.queue[self.front]

    def is_empty(self):
        return self.size == 0

    def is_full(self):
        return self.size == self.capacity

    def size(self):
        return self.size

cq = CircularQueue(5)
cq.enqueue(1)
cq.enqueue(2)
print(cq.dequeue())  # Output: 1
print(cq.peek())     # Output: 2
```

4. Using a Linked List

A **Linked List** is another efficient way to implement a queue. It doesn't require shifting elements like a list-based queue, and the enqueue and dequeue operations are O(1).

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedListQueue:
    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0

    def enqueue(self, item):
        new_node = Node(item)
        if self.rear is None:
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node
        self.size += 1

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        item = self.front.data
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        self.size -= 1
        return item

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.front.data

    def is_empty(self):
        return self.size == 0

    def size(self):
        return self.size
```

```
    def print_queue(self):
        if self.is_empty():
            print("Queue is empty")
            return
        current = self.front
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Testing the queue and print_queue method
llq = LinkedListQueue()
llq.enqueue(10)
llq.enqueue(20)
llq.enqueue(30)

# Printing the queue
llq.print_queue()  # Output: 10 -> 20 -> 30 -> None

# Dequeueing an element
llq.dequeue()

# Printing the queue again
llq.print_queue()  # Output: 20 -> 30 -> None
```

---

5. Priority Queues:

A **priority queue** is a type of data structure that is similar to a regular queue, but with one key difference: in a priority queue, each element is associated with a **priority**. The element with the highest priority is always dequeued first, even if other elements have been in the queue longer.

In simpler terms, a priority queue allows you to manage tasks (or items) with different levels of urgency. For example, in a hospital, patients with critical conditions are treated first, even if they arrived later than others.

Why Do We Need a Priority Queue?

We need a priority queue when:

1. **Processing tasks with different importance**: Sometimes we need to handle tasks in a certain order based on their priority. For instance, a job scheduler might process high-priority jobs before low-priority ones.
2. **Efficient data management**: A priority queue allows us to manage a collection of items where the most urgent or important one must be accessed first, which is important in scenarios like managing real-time systems or organizing events in a simulation.
3. **Algorithms like Dijkstra's or A***: These algorithms use priority queues to find the shortest path or the most efficient solution, making them crucial in areas like navigation or network routing.

Implementing Priority Queue in Python:

In Python, you can use the queue.PriorityQueue class or the heapq module, which provides an efficient way to maintain a priority queue using a **heap**.

5.1 Priority Queue ADT (Abstract Data Type)

In a priority queue, the typical operations are:

- **Insert (enqueue)** an element with a given priority.
- **Remove (dequeue)** the element with the highest priority.
- **Peek** the highest priority element without removing it.
- **IsEmpty** to check if the queue is empty

5.2 Unbounded Priority Queue (Using a List)

An **unbounded priority queue** is a priority queue that has no fixed size limit. You can implement it using a list, where elements are inserted based on their priority.

```
import heapq

class UnboundedPriorityQueue:
    def __init__(self):
        self.pq = []

    def enqueue(self, item, priority):
        heapq.heappush(self.pq, (priority, item))

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return heapq.heappop(self.pq)[1]
    def peek(self):
```

```
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.pq[0][1]

    def is_empty(self):
        return len(self.pq) == 0

pq = UnboundedPriorityQueue()
pq.enqueue("task1", 2)
pq.enqueue("task2", 1)
pq.enqueue("task3", 3)
print(pq.dequeue())  # Output: task2
```

In the code above, heapq is used to maintain the order of priorities automatically.

5.3 Bounded Priority Queue:

A **bounded priority queue** has a fixed size limit. If the queue reaches this limit, new elements may either overwrite existing elements or be rejected.

```
import heapq

class BoundedPriorityQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.pq = []

    def enqueue(self, item, priority):
        if len(self.pq) >= self.capacity:
            raise IndexError("Queue is full")
        heapq.heappush(self.pq, (priority, item))

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return heapq.heappop(self.pq)[1]

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.pq[0][1]

    def is_empty(self):
        return len(self.pq) == 0
```

```
bpq = BoundedPriorityQueue(3)
bpq.enqueue("task1", 2)
bpq.enqueue("task2", 1)
bpq.enqueue("task3", 3)
print(bpq.dequeue())  # Output: task2
```

Bounded vs. Unbounded Priority Queue:

Unbounded Priority Queue:

An **unbounded priority queue** means that the queue has no fixed limit. You can keep adding as many elements as you want, and the system will allocate more memory as needed.

- **Characteristics**:
  - It grows dynamically as elements are added.
  - There's no fixed size limit on the number of elements.
  - Useful in situations where the number of tasks or data isn't known in advance.
- **Example**: Imagine a priority queue that manages tasks for a server. The queue will keep accepting tasks until the server shuts down, regardless of how many tasks there are.

Bounded Priority Queue:

A **bounded priority queue** has a fixed capacity. This means you can only add a specific number of elements before it becomes full. Once the limit is reached, you cannot add more elements until some are removed.

- **Characteristics**:
  - It has a fixed size limit, and adding items beyond this limit is not possible unless an item is removed.
  - Useful when there’s a constraint on memory or resources, like in embedded systems or real-time applications.
- **Example**: A bounded priority queue could be used to manage a limited number of important tasks in a system. Once the system reaches the maximum capacity of tasks, it will not accept any new tasks until some are completed and removed from the queue.

Real-World Example for Each:

1. **Unbounded Priority Queue**: Think of an emergency room where patients can keep arriving. There's no limit to the number of patients the hospital can receive, so the priority queue will handle them indefinitely based on their severity (priority).
2. **Bounded Priority Queue**: In a video game, there might be a limited number of tasks (e.g., quests) available at any time. Once the maximum number of active quests is reached, no new quests can be accepted until some existing ones are completed.

6. Applications of Priority Queues:

Priority queues are often used in computer simulations and real-life applications such as **task scheduling** or **event-driven simulations**. Below is an example of **an airline ticket counter simulation**.

Airline Ticket Counter Simulation:

In an airline ticket counter scenario, customers with higher priority (for example, those who are business class or frequent flyers) should be served before those with lower priority (economy class).

```
import heapq

class Customer:
    def __init__(self, name, priority):
        self.name = name
        self.priority = priority

    def __lt__(self, other):
        return self.priority < other.priority

class AirlineTicketCounter:
    def __init__(self):
        self.pq = []

    def add_customer(self, customer):
        heapq.heappush(self.pq, customer)

    def serve_customer(self):
        if self.is_empty():
```

```
            raise IndexError("No customers to serve")
        customer = heapq.heappop(self.pq)
        return customer.name

    def is_empty(self):
        return len(self.pq) == 0
counter = AirlineTicketCounter()
counter.add_customer(Customer("John", 2))  # priority 2
counter.add_customer(Customer("Alice", 1))  # priority 1
counter.add_customer(Customer("Bob", 3))    # priority 3

print(counter.serve_customer())  # Output: Alice
print(counter.serve_customer())  # Output: John
```