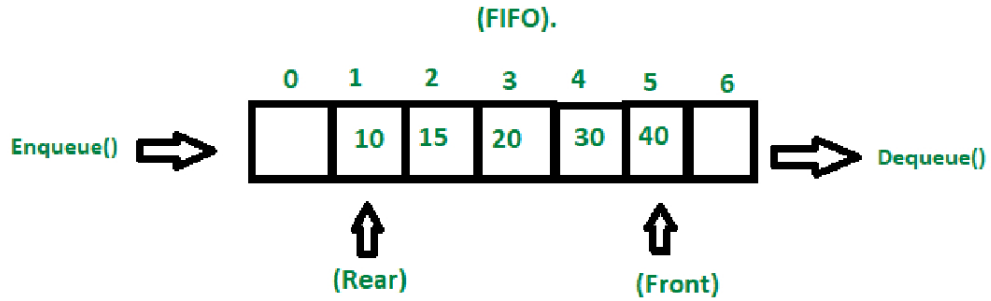
FIFO Principle of Queue:

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue. See the below figure.



Characteristics of Queue:

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

Operations associated with queue are:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity : O(1)
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity : O(1)
- **Front:** Get the front item from queue – Time Complexity : O(1)

- **Rear:** Get the last item from queue – Time Complexity : O(1)

Insertion operation: enqueue()

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

Algorithm

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 rear pointer to point the next empty space.
5 – Add data element to the queue location, where the rear is pointing.
6 – return success.
7 – END

Deletion Operation: dequeue()

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

Algorithm

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

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.

Algorithm

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

The isFull() Operation

The isFull() operation verifies whether the stack is full.

Algorithm

1 – START
2 – If the count of queue elements equals the queue size, return true
3 – Otherwise, return false
4 – END

The isEmpty() operation

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1 – START
2 – If the count of queue elements equals zero, return true
3 – Otherwise, return false
4 – END

A **priority queue** is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.
In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value. For example, if you add an element with a high priority value to a priority

queue, it may be inserted near the front of the queue, while an element with a low priority value may be inserted near the back.

There are several ways to implement a priority queue, including using an array, linked list, heap, or binary search tree. Each method has its own advantages and disadvantages, and the best choice will depend on the specific needs of your application.

Priority queues are often used in real-time systems, where the order in which elements are processed can have significant consequences. They are also used in algorithms to improve their efficiencies, such as **Dijkstra's algorithm** for finding the shortest path in a graph and the A* search algorithm for pathfinding

.

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.

Operations of a Priority Queue:

A typical priority queue supports the following operations:

1) Insertion in a Priority Queue

When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct place then it will be compared with the parent node. If the element is not in the correct order, the elements are swapped. The swapping process continues until all the elements are placed in the correct position.

2) Deletion in a Priority Queue

As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the heap invariant.

3) Peek in a Priority Queue

This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

Types of Priority Queue:

1) Ascending Order Priority Queue

As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements

in a priority queue arranged in ascending order like 4,6,8,9,10. Here, 4 is the smallest number, therefore, it will get the highest priority in a priority queue and so when we dequeue from this type of priority queue, 4 will remove from the queue and dequeue returns 4.

-

2) Descending order Priority Queue

The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future. The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.

Implementing Priority Queue

So now we will design our very own minimum priority queue using python list and object oriented concept.

Below are the algorithm steps:

1. Node: The Node class will be the element inserted in the priority queue. You can modify the Node class as per your requirements.
2. insert: To add a new data element(Node) in the priority queue.
    - If the priority queue is empty, we will insert the element to it.
    - If the priority queue is not empty, we will traverse the queue, comparing the priorities of the existing nodes with the new node, and we will add the new node before the node with priority greater than the new node.

- If the new node has the highest priority, then we will add the new node at the end.

3. delete: To remove the element with least priority.
4. size: To check the size of the priority queue, in other words count the number of elements in the queue and return it.
5. show: To print all the priority queue elements
   .

```
class Node:

  def __init__(self, info, priority):
    self.info = info
    self.priority = priority

# class for Priority queue
class PriorityQueue:

  def __init__(self):
    self.queue = list()
    # if you want you can set a maximum size for the queue

  def insert(self, node):
    # if queue is empty
    if self.size() == 0:
      # add the new node
      self.queue.append(node)
    else:
      # traverse the queue to find the right place for new node
      for x in range(0, self.size()):
        # if the priority of new node is greater
        if node.priority >= self.queue[x].priority:
          # if we have traversed the complete queue
          if x == (self.size()-1):
            # add new node at the end
```

```
            self.queue.insert(x+1, node)
          else:
            continue
        else:
          self.queue.insert(x, node)
          return True

  def delete(self):
    # remove the first node from the queue
    return self.queue.pop(0)

  def show(self):
    for x in self.queue:
      print str(x.info)+" - "+str(x.priority)

  def size(self):
    return len(self.queue)

pQueue = PriorityQueue()
node1 = Node("C", 3)
node2 = Node("B", 2)
node3 = Node("A", 1)
node4 = Node("Z", 26)
node5 = Node("Y", 25)
node6 = Node("L", 12)

pQueue.insert(node1)
pQueue.insert(node2)
pQueue.insert(node3)
pQueue.insert(node4)
pQueue.insert(node5)
pQueue.insert(node6)
pQueue.show()
print("--------")
```

```
pQueue.delete()
pQueue.show()
```