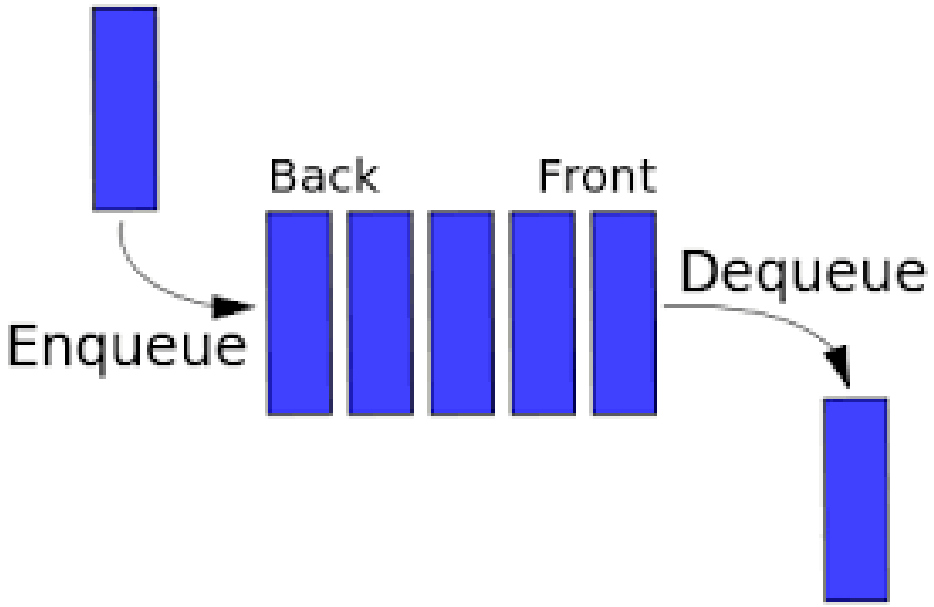
Queue

We have started to learn secondary data structures made using primary data structures. The first secondary data structure we have learned is Stack. The idea of Stack was Last In First Out (LIFO) and we implemented that using a linked list. Stack helps us to solve multiple problems more efficiently. Similarly, we can create a new Data Structure that will follow different rules. One example of a new data structure is Queue

A queue is a new data structure that follows **First In First Out (FIFO)** while inserting and retrieving data. We are very familiar with the idea of queue in our daily lives. To illustrate, if you are waiting in line to buy a bus ticket, the counter will serve the customer who has come first and a new customer will wait behind the last person waiting in line. This idea is also important for our computers. One of the main use of queue in computers is resource management. For example, think of an office scenario where 10 people use the same printer. So if everyone requests to print simultaneously, the printer will print the first request and store the other requests in a queue. While executing any printing operation if a new print request comes it will store it in the same queue and execute the command in the First In First Out manner. In addition, when we execute 100 programs in our 8-core CPU, the processor also maintains a queue for command execution (Though the scenario is a bit more complex which you will learn in Operating Systems).

Queue Operations:



Queue

A queue has three main operations (like a stack). These operations are:

1. **Enqueue:** It stores a new object in the queue at the end. If the queue is full, this operation throws a QueueOverflow exception.
2. **Dequeue:** It removes the first object from the queue. If the queue is empty, this operation throws a QueueUnderflow exception.
3. **Peek**: This operation shows the first item in the queue. This operation does not remove any items from the queue. If the queue is empty, this operation throws a QueueUnderflow exception.

Queue Implementation:

As we can see, to maintain the queue, we need two pointers, one for item retrieval and one for item insertion. As a result, we will maintain a front and a back for a queue. The idea is whenever we dequeue an item or peek at an item, we will use the front pointer and to enqueue an item we will use the back pointer. An example of a linked list-based queue is given below.

This is the node class.

```
class Node:
  def __init__(self, elem, next):
    self.elem = elem
    self.next = next
```

```
class Queue:

  def __init__(self):
    self.front = None
    self.back = None

  def enqueue(self, elem):
    if self.front == None: # This means first item
      self.front = Node(elem, None)
      self.back = self.front # Initially front and back are same
    else:
      n = Node(elem, None)
      self.back.next = n # Inserting at the last
      self.back = self.back.next # Moving back at the end

  def dequeue(self):
    if self.front == None: # For empty Queue
      print("Queue Underflow Exception")
    else:
      dequeued_item = self.front.elem
      self.front = self.front.next # Moving the front pointer
      return dequeued_item # Returning the dequeued item

  def peek(self):
    if self.front == None: # For empty Queue
      print("Queue Underflow Exception")
    else:
      return self.front.elem # Returning the front item
```

Note that here, we are maintaining a front (for dequeue and peek) and a back (for enqueue). As the size of the linked list is not fixed, we are not maintaining the QueueOverflow exception in the given code.

Array Based Queue: If we want to create a queue with fixed size then array based queue is a good approach. Here we will also use a front index and a back index. Initially both will start from the same index. After that, after every enqueue the back index will move one index further. After every dequeue the front index will move one index further. To keep indexing convenient we will use a circular array to implement the queue. To determine the queue overflow exception, we will maintain a size variable.

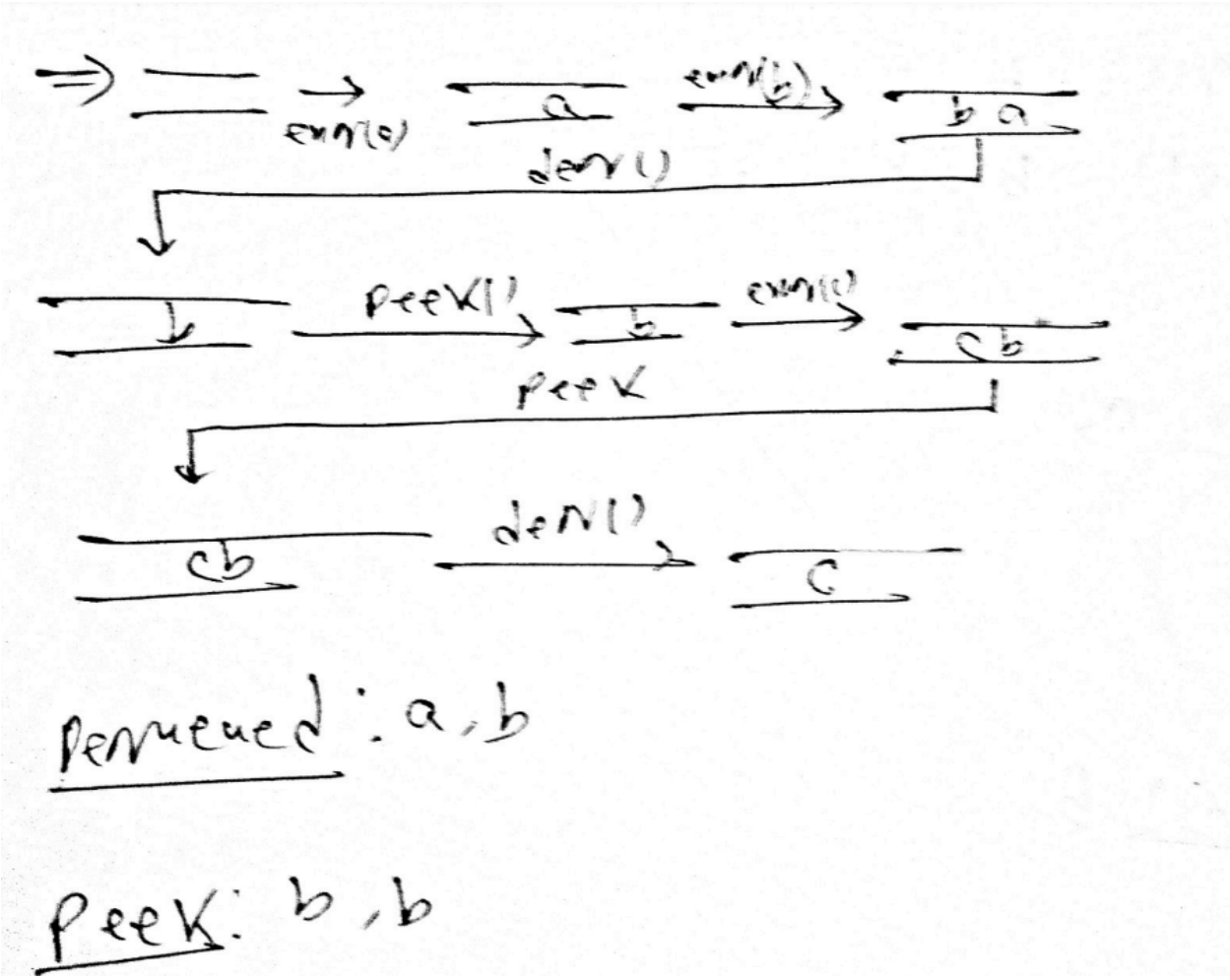
Array based queue are also known as buffer. Buffer is used in many applications such as web server request management.

```
class ArrayQueue:
  def __init__(self):
    self.queue = [None] * 10 # Queue with size 10
    self.front = 0 # Initializing at index 0
    self.back = 0 # Initializing at index 0
    self.size = 0 # no elements

  def enqueue(self, elem):
    if self.size == len(self.queue):
      print("Queue Overflow")
    else:
      self.queue[self.back] = elem
      self.back = (self.back + 1) % len(self.queue)
      self.size += 1

  def dequeue(self):
    if self.size == 0:
      print("Queue Underflow")
    else:
      dequeued_item = self.queue[self.front]
      self.queue[self.front] = None
      self.front = (self.front + 1) % len(self.queue)
      self.size -= 1
      return dequeued_item

  def peek(self):
    if self.size == 0:
      print("Queue Underflow")
    else:
      return self.queue[self.front]
```

Queue Simulation:

You need to do the following operations on a queue:

enqueue a, enqueue b, dequeue, peek, enqueue c, peek, dequeue.



Another type of simulation:

You need to perform the following operations on a array based queue where the length of array is 4 and starting index of front and back is 3.

enqueue a, enqueue b, dequeue, peek, enqueue c, peek, dequeue.