Queues

* The abstract queue (or simply queue) is specified as a linearly ordered collection of data elements with enqueue, dequeue and peek operations

1. The enqueue operation inserts an element into the collection. The linear ordering of elements are determined by the time they are inserted. The front element is the earliest inserted element, and the rear element is the last inserted element.
2. The dequeue operation removes (deletes) the front element
3. The peek operation gets the first element

* The queue has the characteristic of First in First Out (FIFO), meaning that it deletes the first inserted element
* A queue data structure is an implementation of the abstract queue
* Queue can be implemented by arrays or linked lists with two variables front and rear representing the front and rear positions of the queue. Deleting element at the front position, inserting element at the rear position
* A queue is said to be empty if there is no element in the queue. When a queue is empty, deletion can not be done such a situation is called underflow
* The length of a queue is the number of elements in the queue. When the length reaches the maximum length that a queue is allowed, then insertion cannot be done such a situation is called an overflow

**Array based queues**

* A simple array queue uses an array to store queue elements, and the front, rear variables represent the index positions that deletions and insertions are done respectively.
  + A simple array queue is created by creating an array of MAX (given) size and front =-1 and rear =-1
  + Enqueue: When rear <MAX-1, inserting an element is done by setting rear = rear+1 and front =0 if front =-1, and set the new value of the rear position
  + Dequeue: when the queue is not empty, deleting an element from the queue is done by setting front = front +1
  + Peek: when the queue is not empty, get the element at the front position
* Underflow (empty): front = rear = -1 or front>rear
* Overflow: rear = MAX-1

**Enqueue algorithm: insert an element into a queue**

Input queue[MAX], front, rear, value

1. If rear = MAX-1, then;

Print OVERFLOW

Goto step 4

1. If front ==-1 and rear ==-1

Front = rear = 0

Else

Rear++

1. Queue[rear] = value
2. Stop

Time: O(1), space: O(1)

**Dequeue algorithm: delete front element**

Input: queue[Max], front, rear

1. If front =-1

Print UNDERFLOW

Goto step 2

1. If front<rear

Front++

Else set front = rear = -1

1. Return val, stop

Time: O(1), space : O(1)

**Peek algorithm: get front element**

Input: queue[Max], front, rear

1. If front = -1

Print UNDERFLOW

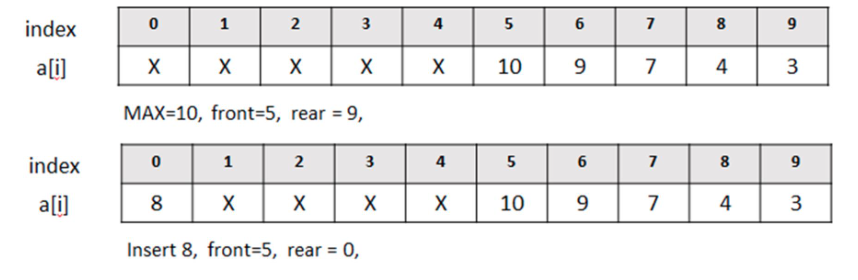
Return -1

Else return front

Time: O(1), space: O(1)

**Queue by circular arrays**

* Circular array queues fix the drawback of the simple array queue by adding new elements to the available spaces on the “left side” when rear reaches MAX-1 and front >0



* Underflow: rear = front = -1
* Overflow: Front = 0 and rear = Max-1, or rear = front-1
* When there is only one element, namely when front = rear !=-1, the deletion will set front = rear = -1

**Enqueue algorithm: insert an element in a circular array queue**

Input: queue[Max], front, rear, value

1. If (front==0 && rear ==MAX-1)||rear ==front-1

Print OVERFLOW

Goto step 4

1. If front ==-1

Front = rear=0

Else if rear == MAX-1 and front !=0

Rear = 0

Else

Rear = rear+1

1. Queue[rear] = value
2. Stop

Time: O(1), space: O(1)

**Dequeue algorithm: delete from a circular array queue**

Input: queue[MAX], front, rear

1. If front = -1

Print “underflow”

Goto step 4

1. Value = queue[front]
2. If front == rear

Front = rear = -1

Else if front == MAX -1

Front = 0

Else

Front = front + 1

1. Stop

Time: O(1), space: O(1)

**Drawbacks of array queues**

* The array (circular) queues have two drawbacks:
  + The length of the queue is bounded by the length of its host array
  + It wastes space if the length of the queue is much shorter than the length of the host array
* Linked list queues fix these drawbacks of array and circular queues

**Queue by linked list**

* A linked list queue uses a singly linked list to represent queue with two pointers front and rear to represent the front and rear positions. The front pointer points to the first node, the rear pointer points to the last node.
  + Enqueue: firstly create a node containing the data value, secondly insert the node after the rear node, thirdly update front and rear
  + Dequeue: delete the first node and update front and rear
  + Peek: operation returns the data value in the front node

**Enqueue algorithm: insert an element value**

Input: front, rear, value

1. Create newNode = new\_node(value)
2. If front==NULL

Front = newNode

Rear = newNode

Goto step 4

1. Rear->next = newNode;

Rear=newNode

1. Output front and rear

Time: O(1), space: O(1)

**Peek algorithm: get front node**

Input: front, rear

1. Output front

Time: O(1), space: O(1)

**Dequeue algorithm: delete the front node**

Input: Front, rear

1. If front == NULL

Print “underflow”

Goto step 6

1. Ptr = front
2. If front == rear

Front = NULL

Rear = NULL

Goto step 5

1. Front = front->next
2. Free ptr
3. Output front and rear

Time: O(1), space: O(1)

**Deques**

* A deque (double ended queue) is a queue in which elements can be inserted or deleted at either end
* In a deque, two pointers are maintained, left and right which represent the two ends of the deque
* A deque can be implemented either using a circular array or a circular doubly linked list

**Priority queues**

* A priority queue is a queue in which each element is assigned a priority. The priority of elements is used to determine the order in which these elements will be processed. The general rule of processing elements of a priority queue:
  + An element with higher priority is processed before an element with lower priority
  + Two elements with same priority are processed on a first come first serve basis (FCFS)
* Priority queues can be represented by linked lists or arrays

**Queue vs Priority Queue**

* A general queue can be viewed as a priority queue by taking the time stamp of inserting an element as the priority

**Priority queue as ordered items:** a simple priority queue uses a priority value and the time stamp of insertion to order items

* Each item is associated with a value pair (priority, time-stamp). The elements are ordered by (priority, time-stamp). Namely, A appears before item B if (priority(A), time-stamp(A)< Priority(B), time-stamp(B)).
* From this view, the concepts of priority queue can be generalized by associating items with a priority vector (p1, p2, …, pk), then use this vector to order items in increasing ordering of the priority vector.

**Application of priority queue**

* Priority queues are used in operating systems to manage execution ordering the processes.

**Linked list representation of priority queues**

* When a priority queue is implemented using a linked list, then every node of the list contains three parts:
  + Data
  + Priority
  + Address of the next node
* If we are using a sorted linked list, then elements having higher priority will precede the elements with lower priority

**Analysis of priority queue representations**

* Linked list representation: similar to inserting after a node in singly linked list, the priority queue inserting algorithm has O(n) time. The deletion is done at the front of the linked list and has time and space complexity of O(1).
* Array representation: the insertion operation also has time complexity of O(n)
* Heap representation: time complexity O(log n) for both insertion and deletion operation.
* Special case: if there are a few priority values, each priority value corresponds to an individual queue. Then order all elements from high priority queue to low priority queue.

**Applications of queues**

* Whenever an algorithm needs to remember data items and a FIFO is required for item processing, a queue data structure can be an option.

1. Queues are often used to store intermediate data within an algorithm for FIFO retrieval. Ex breadth-first search
2. Queues are used as waiting lists for a single shared resource like printer, disk, CPU, network device.
3. Queues are used in OS for handling interrupts. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.
4. Queues are used to transfer data asynchronously ex. Pipes, file IO, sockets
5. Queues are used in simulations.