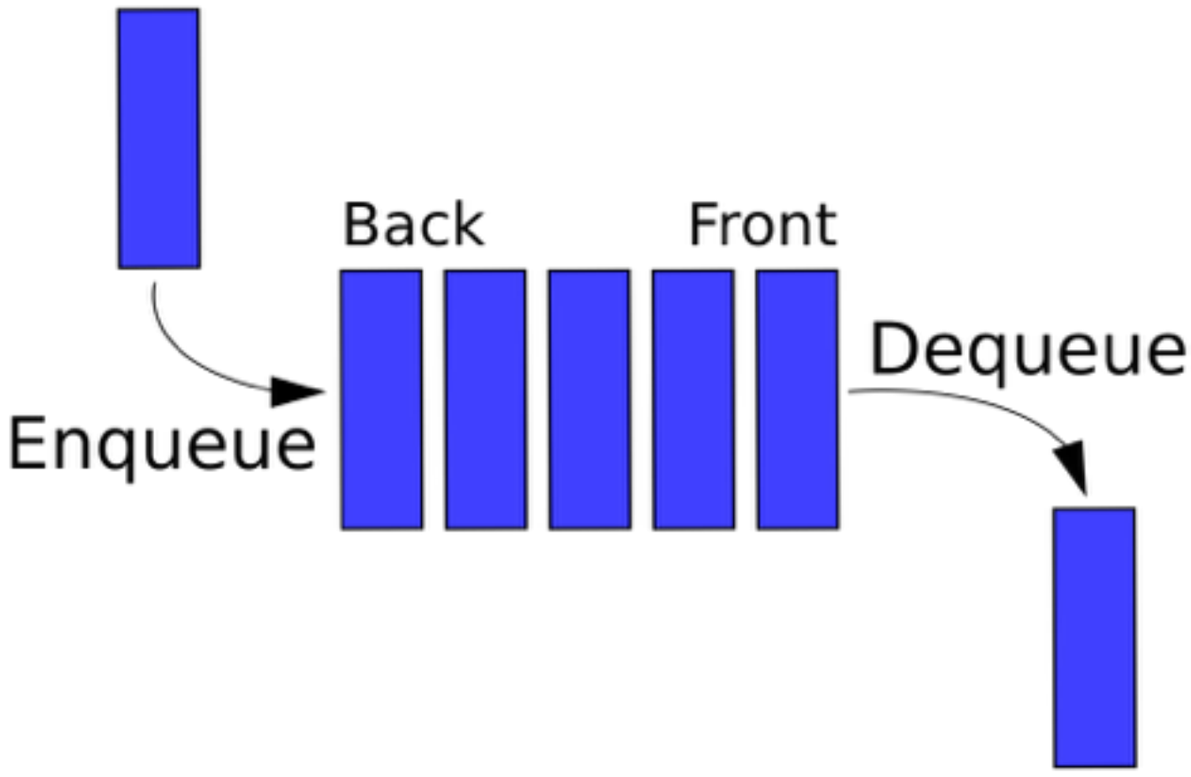
Queues

What is a queue?

- A queue is a container of objects that are inserted and removed according to a **First In First Out** (FIFO) principle.



- Items are added to the back of the queue (**Enqueue**)

- Removal happens from the front (**Dequeue**)

Operations (linear queues)

Operation	Description
add(item)	Add an item to the end of the list
remove()	Remove the first item in the list
peek()	Return the top of the stack (next item to be returned) — optional
isEmpty()	Return true if and only if the stack is empty —optional

Implementation

- There are several efficient implementations of FIFO queues.
- An efficient implementation is one that can perform the operations—enqueuing and dequeuing—in O(1) time.

Implementation

A queue can be implemented with a linked list, with the restriction that items are added and removed from opposite sides.

```
class Queue:
    def __init__(self):
        self.length = 0
        self.head = None

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

    def insert(self, cargo):
        node = Node(cargo)
        if self.head is None:
            # If list is empty the new node goes first
            self.head = node
        else:
            # Find the last node in the list
            last = self.head
            while last.next:
                last = last.next
            # Append the new node
            last.next = node
        self.length += 1

    def remove(self):
        cargo = self.head.cargo
        self.head = self.head.next
        self.length -= 1
        return cargo
```

Implementation

Problem: with this singly-linked list we are traversing the entire list (O(n)) each time we add insert a new node

```
class Queue:
    def __init__(self):
        self.length = 0
        self.head = None

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

    def insert(self, cargo):
        node = Node(cargo)
        if self.head is None:
            # If list is empty the new node goes first
            self.head = node
        else:
            # Find the last node in the list
            last = self.head
            while last.next:
                last = last.next
            # Append the new node
            last.next = node
        self.length += 1

    def remove(self):
        cargo = self.head.cargo
        self.head = self.head.next
        self.length -= 1
        return cargo
```

Implementation

A better implementation would have a self.last attribute to keep track of the last element in the list

```
class ImprovedQueue:
    def __init__(self):
        self.length = 0
        self.head = None
        self.last = None

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

    def insert(self, cargo):
        node = Node(cargo)
        if self.length == 0:
            # If list is empty, the new node is head and last
            self.head = self.last = node
        else:
            # Find the last node
            last = self.last
            # Append the new node
            last.next = node
            self.last = node
        self.length += 1

    def remove(self):
        cargo = self.head.cargo
        self.head = self.head.next
        self.length -= 1
        if self.length == 0:
            self.last = None
        return cargo
```

Other kinds of queues

- **Circular queues**
(also known as a circular buffer)

- **Priority queues**

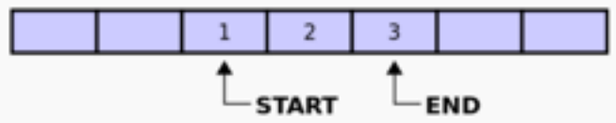
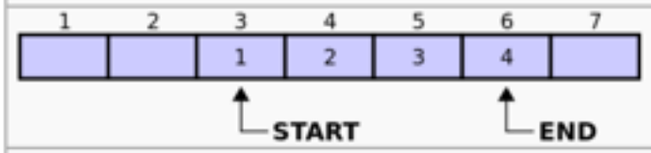
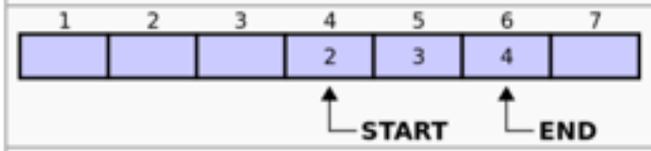
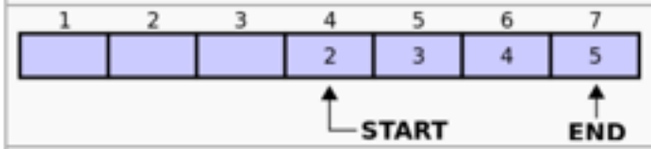
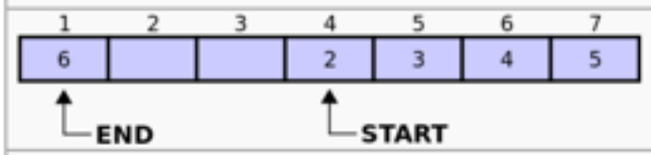
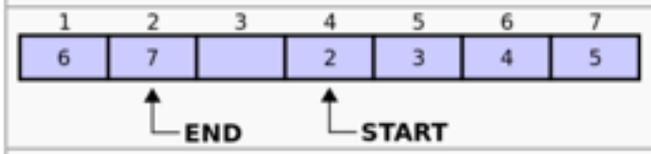
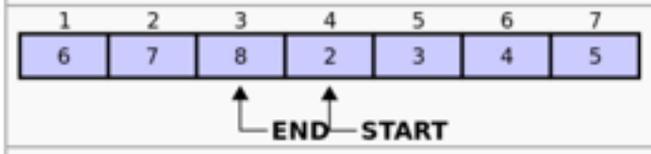
- **Deques (double-ended queues)**

Circular queues
(also known as a circular buffer)

- fixed space allocated to it
- last element points to the first one in the queue, forming a circle
- reuses memory space, so won't overwrite data or run out of space (within limits)
- limited to amount of buffer in array

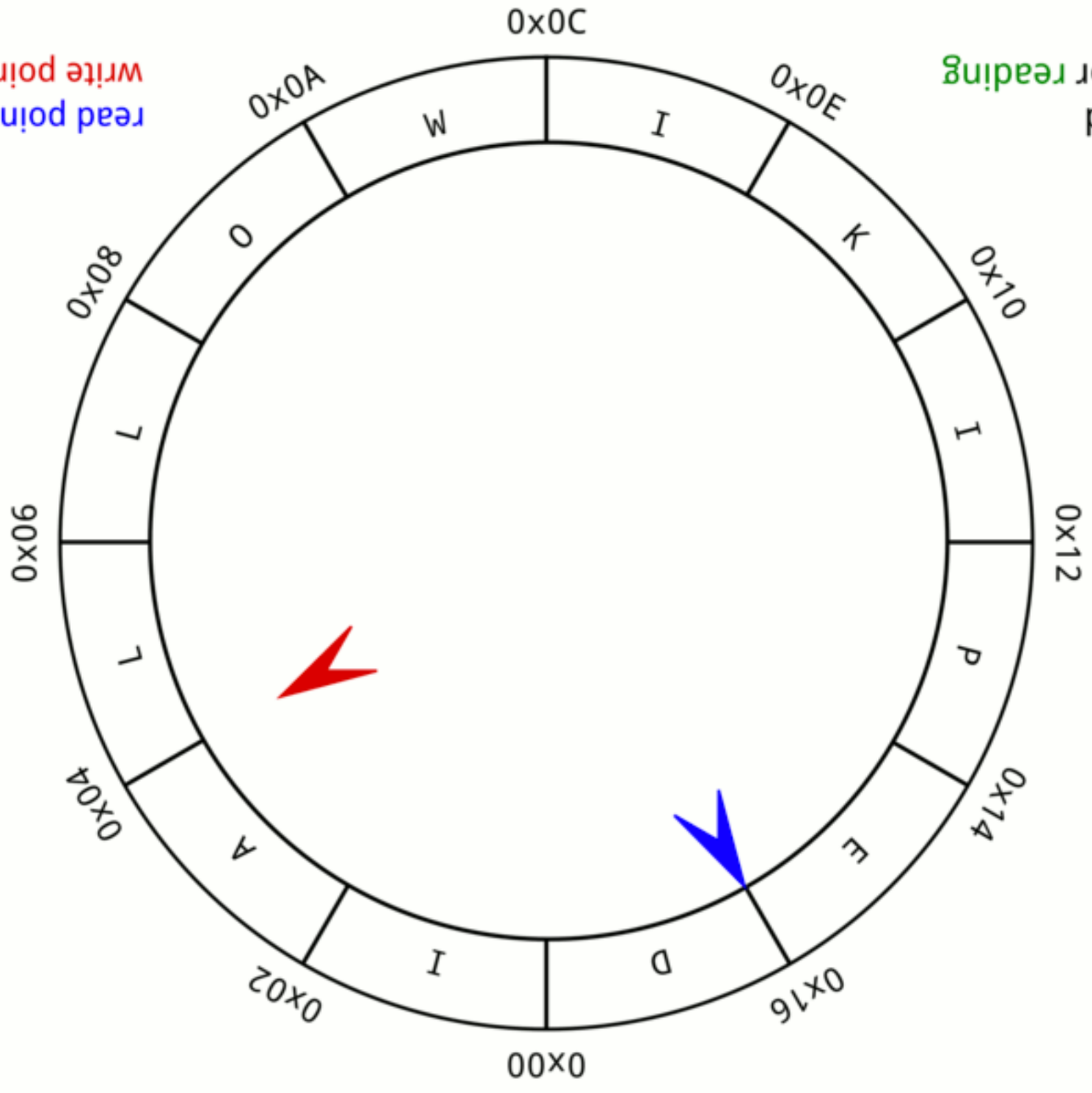
Circular queues generally have 3 pointers:

- one to the buffer in memory
- one to the start of the data
- one to the end

Diagram	StartPtr	EndPtr
	3	5
Data is specified as being between the StartPtr and the EndPtr		
	3	6
Adding another element to the queue moves the EndPtr up one		
	4	6
Deleting an element from the front of the queue is achieved by increasing the StartPtr by one		
	4	7
Adding another element to the queue moves the EndPtr up one, it is now at the end of the buffer		
	4	1
Adding another element to the queue makes the EndPtr loop around to the free space at the beginning of the buffer		
	4	2
Adding another element to the queue moves the EndPtr up one, it is now closing in the on the StartPtr		
	4	3
Adding another element to the queue moves the EndPtr up one, it is now closing in the on the StartPtr. If we tried to add another element to the circular queue we couldn't without first removing something from the front of the queue		

keyboard
processor reading

write pointer 0x04
read pointer 0x16



Priority queues

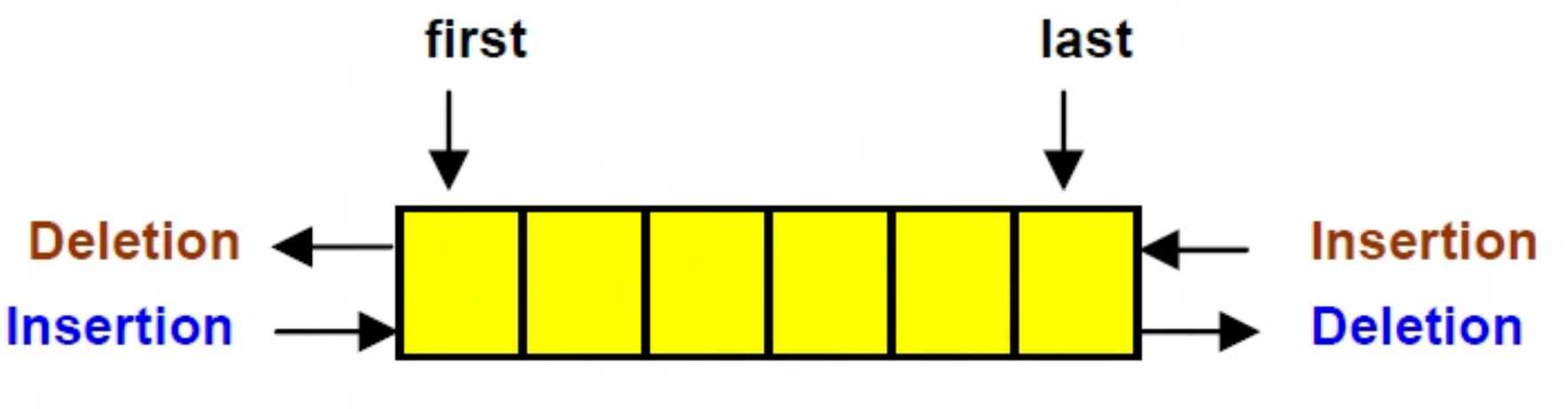
Order of removal is determined by some priority measure rather than order of addition

Example of a priority queue

Process	Priority
Web browser	Normal
Media player	Below normal
OS security	High
Scheduled virus scanner	Low

Deques

- Elements can be added and removed from both ends
- can be implemented using a doubly-linked list



Common applications of queues

Buffers:

any example where you want things to happen in the order that they were added, but the computer cannot keep up to speed. e.g. the keyboard buffer example used earlier, or a print queue.

Breadth-first search:

use a queue to store a list of nodes to process. Each time we process a node, we add its adjacent nodes to the back of the queue. This ensures that we process nodes in the order they are viewed