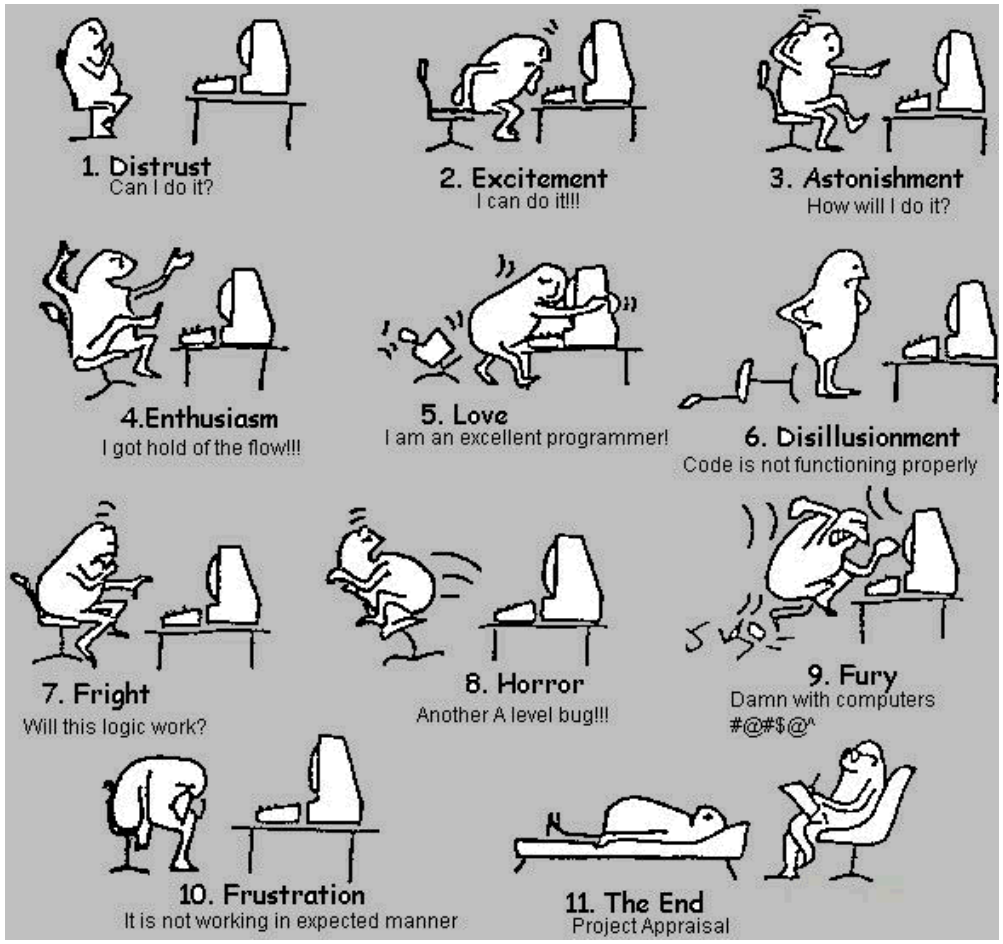
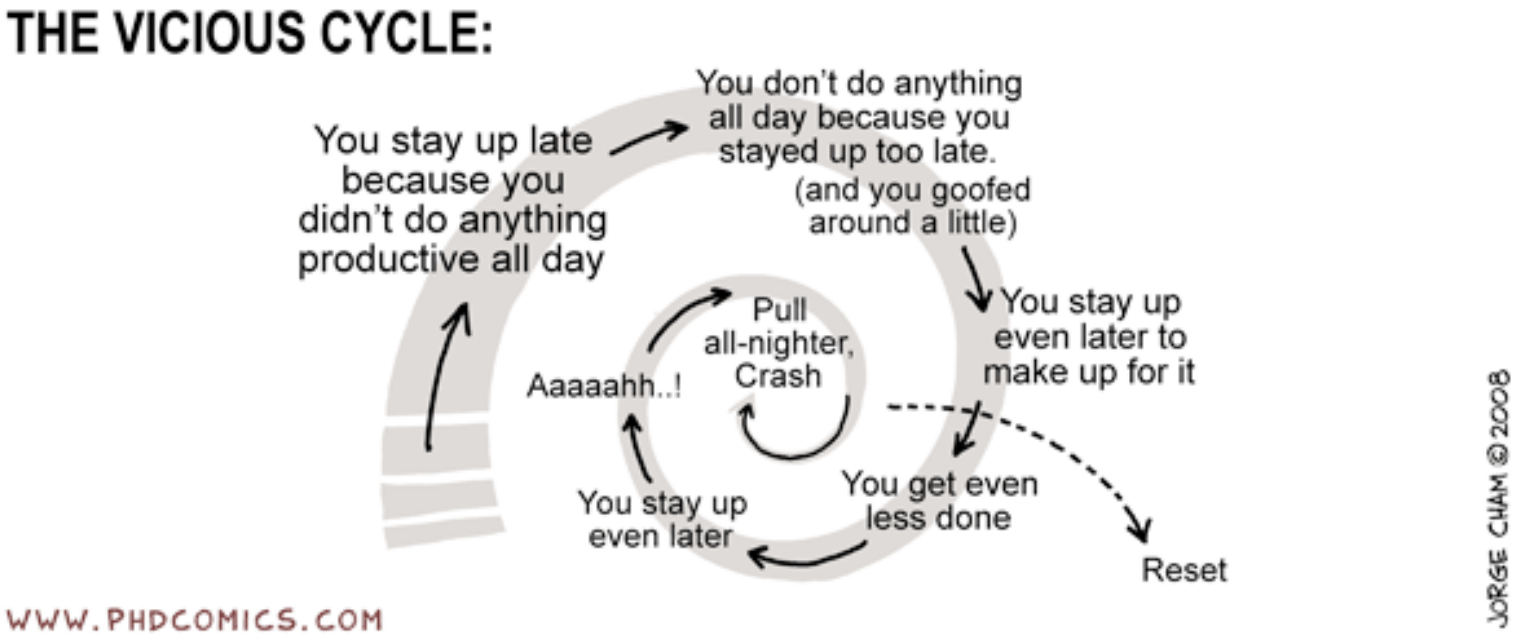
Queues

States of a Programmer



Vicious Software Development



Queue ADT

- ***Definition:*** a **queue** is a collection of objects that are inserted and removed according to the first-in-first-out (FIFO) principle.
- Objects are inserted into the **rear** of the queue.
- Objects can **ONLY** be removed from the **front** of the queue.
- Objects that have been in the queue the longest are first to be removed.
- All queue operations are O(1).
  - *All of the action occurs at the **front** or **rear** of queue.*

Queue - Examples

- Movie ticket line
- Amusement park line
- Grocery store checkout
- Access to shared resources (e.g., printer queue)
- Phone calls to large companies
- Freeway off-ramp
- Life ☺

Queue ADT

Main **queue** operations:

- ***enqueue(o)*** : insert object ***o*** at the rear of the queue.
  - STL operation - ***push(o)***
- ***dequeue( )*** : remove from the queue the object in the front.
  - STL operation - ***pop()***
  - An error occurs if the queue is empty. (*exception*)
- ***front( )*** : returns the element at the front **without** removing it.
  - STL operation - ***front()***
  - An error occurs if the queue is empty. (*exception*)

Auxiliary **queue** operations:

- ***size( )*** : returns the number of objects in a queue. Either store as a variable counter or calculate it.
- ***isEmpty( )*** : returns `true` if the stack is empty, else `false`

Naïve Array-based Queue

- Two variables keep track of the front and rear
  - ***front*** - index of the ***front*** element, initialize to 0
  - ***rear*** -index immediately past the ***rear*** element, initialize to 0
- Variable for number of objects in queue ***Q***
  - ***size***
- Variable for capacity of the queue ***Q***
  - ***N***

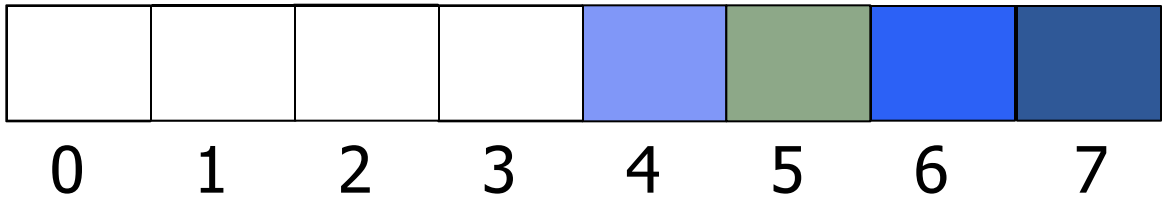
Queue Data Structure

```
class Queue
{
private:
        objectType queue[MAX_QUEUE_SIZE];
        int front;
        int rear;
        int size;
        int N;

public:
        functions for queue manipulation
        constructor sets front and rear to 0
};
```

Naïve Array-based Queue

front = 4
rear = 8

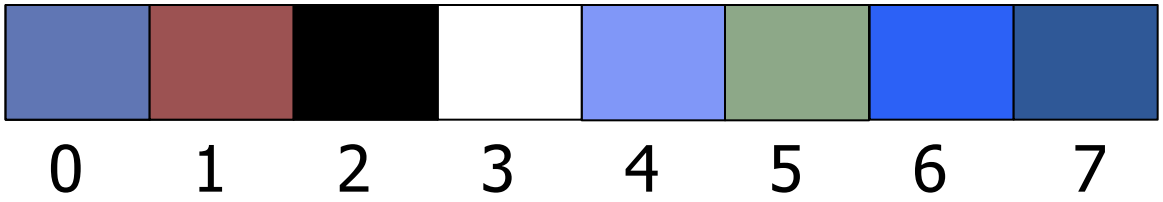


What happens on the next enqueue operation?
What are the possible solutions?

Circular Array-based Queue

- Best solution - use a circular array (wraps around)
  - ***Enqueue*** at the beginning of the array

front = 4
rear = 3



Circular Array-based Queue

- Even though there is plenty of room in the queue, *rear* is at the last cell.
- We want to be able to wrap around.
- We want to index *Q[0] to Q[N-1]* and then immediately go back to *Q[0]*.
- For ***Enqueue***:
  - *rear = (rear + 1) % N , where N=8, Q[0,1,2,…,7]*
  - *rear never points to 8 for N = 8*
  - *rear = (7+1)% 8, wraps around to 0*
- Similarly you can make *front* wrap around.

Queue ADT - Pseudocode

Algorithm *dequeue():*
    if *isEmpty()* **then**
        throw *a QueueEmptyException*
    $f \leftarrow (f + 1) \bmod N$

Algorithm *enqueue(**o**):*
    if *size()=N -1* **then**
        throw *a QueueFullException*
    $Q[r] \leftarrow \boldsymbol{o}$
    $r \leftarrow (r + 1) \bmod N$

Queue ADT - Pseudocode

Algorithm *size():*
    return $(N - f + r) \bmod N$

Algorithm *isEmpty():*
    return $(f == r)$

Algorithm *front():*
    if *isEmpty()* **then**
      throw *QueueEmptyException*
    return *Q[f]*

Circular Array-based Queue

Algorithm *size():*

return *(N – f + r) mod N*

Algorithm *isEmpty():*

return *(f==r)*

Algorithm *front():*

if *isEmpty()* **then**

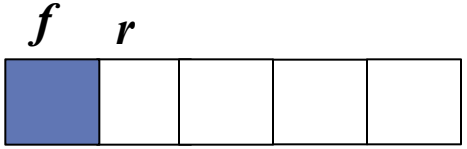
throw *a QueueEmptyException*

return *Q[f]*

size **= (N - f + r) % N**

size **= (5 - 0 + 1) % 5**

size = 1 **= (6) % 5**



Circular Array-based Queue

Algorithm *size():*

return *(N – f + r) mod N*

Algorithm *isEmpty():*

return *(f==r)*

Algorithm *front():*

if *isEmpty()* **then**

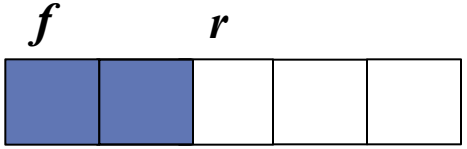
throw *a QueueEmptyException*

return *Q[f]*

***size* = (N - f + r) % N**

***size* = (5 - 0 + 2) % 5**

***size* = 2 = (7) % 5**



Circular Array-based Queue

Algorithm *size():*

return *(N – f + r) mod N*

Algorithm *isEmpty():*

return *(f==r)*

Algorithm *front():*

if *isEmpty()* **then**

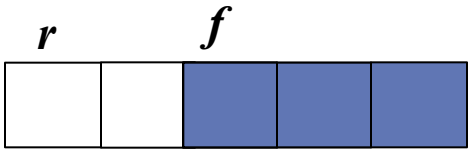
throw *a QueueEmptyException*

return *Q[f]*

***size* = (N - f + r) % N**

***size* = (5 - ? + ?) % 5**

***size* = 3 = (?) % 5**



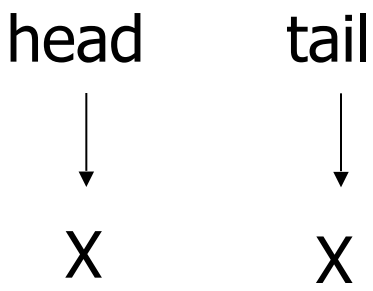
Extendable Array-based Queue

- In an *enqueue* operation, when the array is full, instead of making this an error condition, we can replace the array with a larger one
- Generally every time you increase the size of an array, you will double it in size.
- This disadvantage can also be addressed by using a linked list rather than an array as the underlying data structure.

Linked List-based Queue

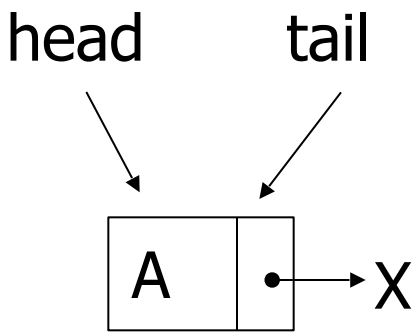
- Using a linked list -- can remove the size restrictions of an array
- Queue can grow dynamically
- Linked list with front and rear pointers
  - *front* is the same as *head*
  - *rear* is the same as *tail*
- *head* and *tail* initially point to NULL
  - Similar to array-based queue where *head* and *tail* are set to zero

Linked List-based Queue



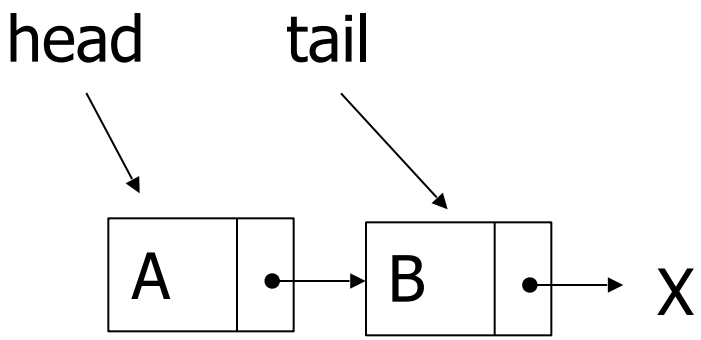
Linked List-based Queue

Enqueue



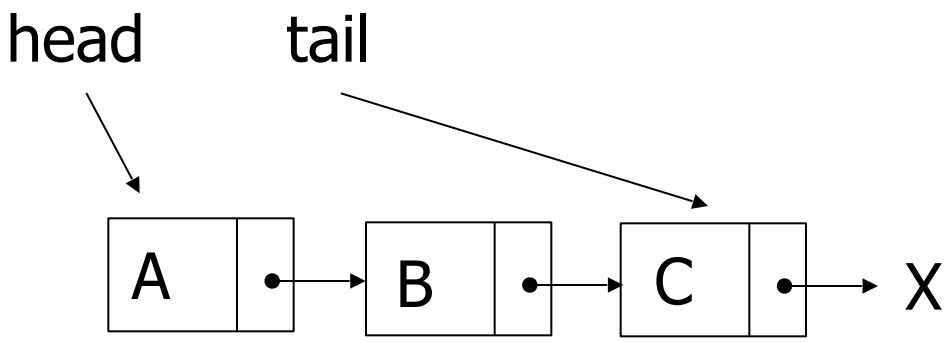
Linked List-based Queue

Enqueue



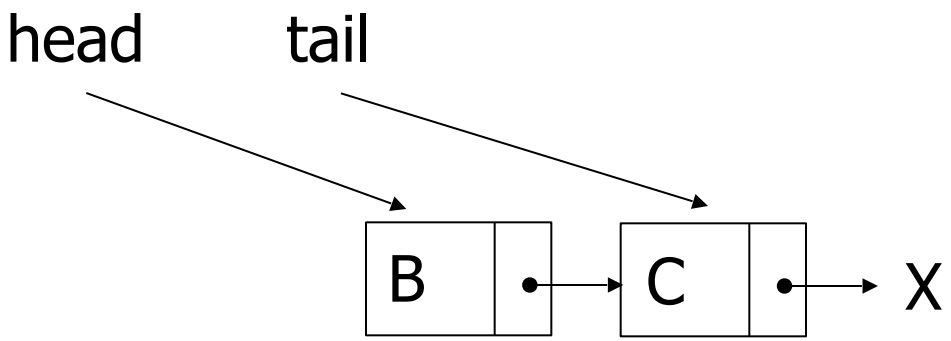
Linked List-based Queue

Enqueue



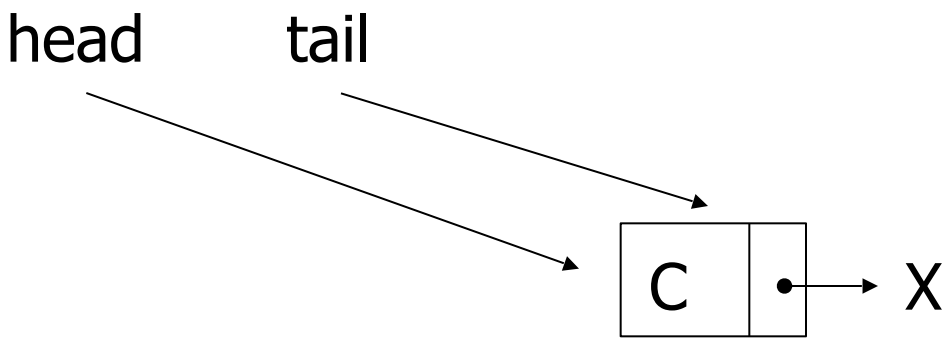
Linked List-based Queue

Dequeue



Linked List-based Queue

Dequeue



Linked List-based Queue

```
class Queue
{
private:
        Node* front;
        Node* rear;
        int numItems;
public:
        all functions to interface with queue
};
```

Linked List-based Queue

```
bool isEmpty ( )
{
   return ( rear == NULL );
}
```

```
const Object & getFront ( )
{
   return front->data;
}
```

```
void enqueue ( const Object & o)
{
   Node * newNode = new Node(o);
   if ( isEmpty ( ) )
      front = newNode;
   else
      rear->next = newNode;
   rear = newNode;
}
```

Linked List-based Queue

```
void dequeue ( )
{
        if (empty)
                error – queue empty
        else
                Node* ptr = front
                front = front->next
                if (front == NULL)
                        rear = NULL
                delete ptr
                --numItems
}
```