And Deques

Queues

Outline

- Queues
- Deques
- Adapters
- Priority Queues

Queues

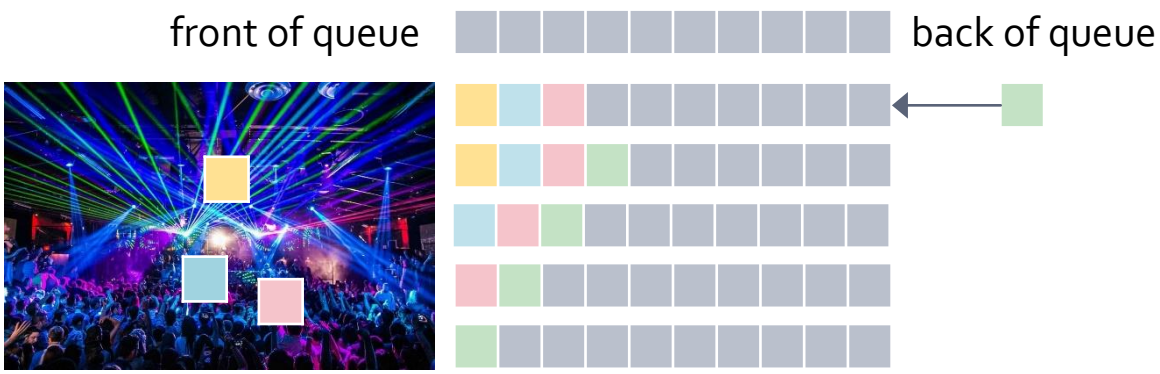
Print Queues

- Assume we want to store data for files to be printed by a shared printer
  - The printer print files in the order in which they are received
    - A *fair* algorithm
- To maintain the print queue we create two classes
  - Request class
  - Container class to store requests
    - FIFO (First In First Out)
    - The ADT is a *queue*

Print Request
Student ID
Time
File name

Queues

- In a queue items are inserted at the back and removed from the front
  - As an aside *queue* is just the British (i.e. correct☺) word for a line (or line-up)
- Queues are **FIFO** (First In First Out) data structures – *fair* data structures



What Can You Use a Queue For?

- Server requests
  - Instant messaging servers queue up incoming messages
  - Database requests
    - Why might this be a bad idea for all such requests?
- Print queues
- Operating systems often use queues to schedule CPU jobs
- Various algorithm implementations

Queue Operations

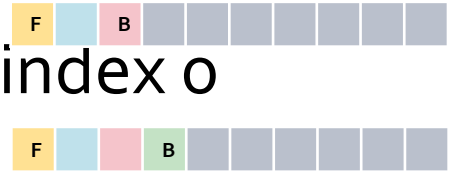
- A queue should implement at least the first two of these operations:
  - *insert* – insert item at the back of the queue
  - *remove* – remove an item from the front
  - *peek* – return the item at the front of the queue without removing it
- Like stacks, it is assumed that these operations will be implemented efficiently
  - That is, in constant time

Implementing a Queue

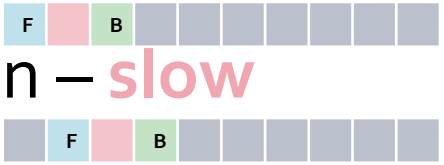
with an Array

Array Implementation

- Consider using an array as the underlying structure for a queue, we could
  - Make the back of the queue the current size of the array, much like the stack implementation
  - Initially make the front of the queue index 0
  - Inserting an item is easy

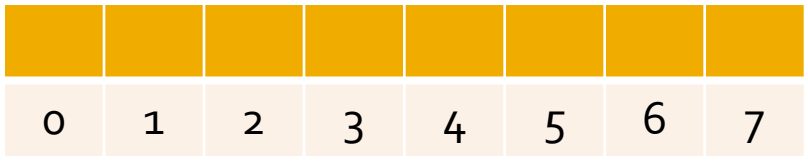


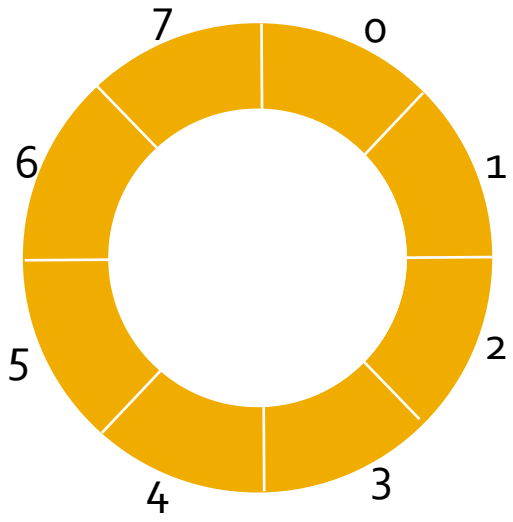
- What happens when items are removed?
  - Either move all remaining items down – slow
  - Or increment the front index – wastes space



Circular Arrays

- **Neat trick**: use a *circular array* to insert and remove items from a queue in constant time
- The idea of a circular array is that the end of the array "wraps around" to the start of the array





Also used in hash tables

The mod Operator

- The *mod* operator (%) calculates remainders:
  - `1%5 = 1, 2%5 = 2, 5%5 = 0, 8%5 = 3`
- The *mod* operator can be used to calculate the front and back positions in a circular array
  - Thereby avoiding comparisons to the array size
  - The back of the queue is:
    - `(front + num) % queue.length`
    - where *num* is the number of items in the queue
  - After removing an item the front of the queue is:
    - `(front + 1) % queue.length;`

Array Queue Example

Insert at (front + num) % length

42		3	13	7	11
0	1	2	3	4	5

6 @ (0 + 0) % 6 = 0

4 @ (0 + 1) % 6 = 1

3 @ (0 + 2) % 6 = 2

13 @ (0 + 3) % 6 = 3

7 @ (0 + 4) % 6 = 4

11 @ (1 + 4) % 6 = 5

42 @ (2 + 4) % 6 = 0

```
Queue q();
q.insert(6); //front = 0
q.insert(4); //front = 0
q.insert(3); //front = 0
q.insert(13); //front = 0
q.insert(7); //front = 0
q.remove(); //front = 1
q.insert(11); //front = 1
q.remove(); //front = 2
q.insert(42); //front = 2
```

Implementing a Queue

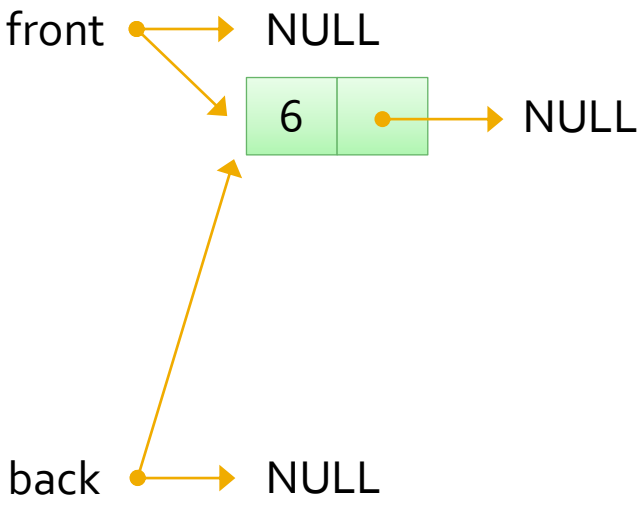
With a Linked List

Linked List Implementation

- Removing items from the front of the queue is straightforward
- Items should be inserted at the back of the queue in constant time
  - So we must avoid traversing through the list
  - Use a second node pointer to keep track of the node at the back of the queue
    - Requires a little extra administration

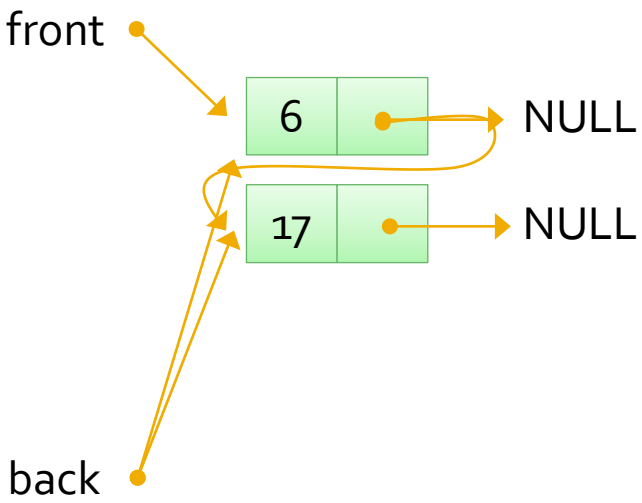
List Queue Example

```
Queue q;
q.insert(6);
```



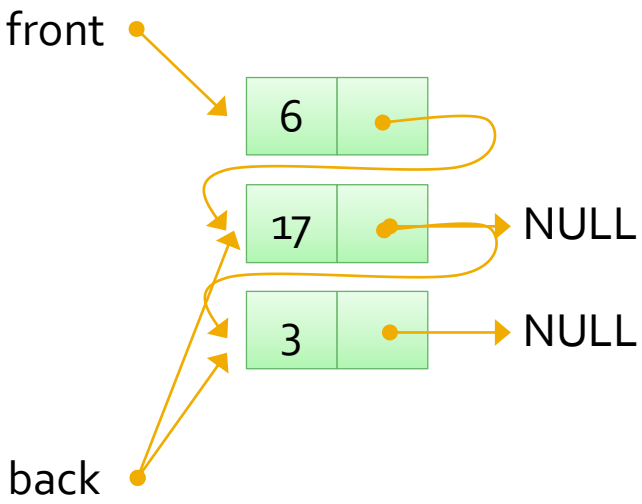
List Queue Example

```
Queue q;
q.insert(6);
q.insert(17);
```

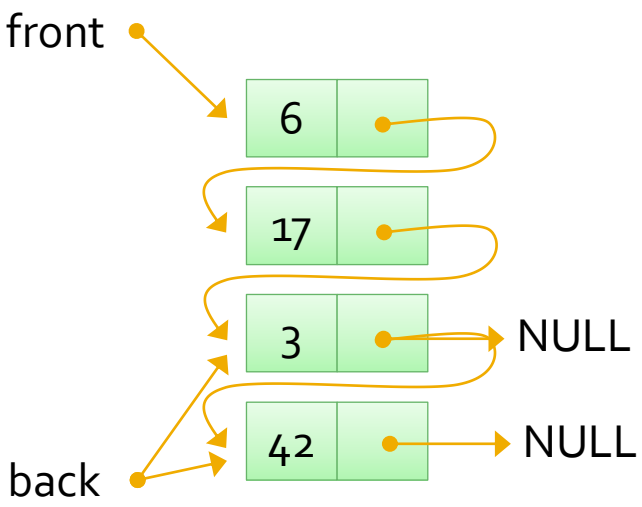


List Queue Example

```
Queue q;
q.insert(6);
q.insert(17);
q.insert(3);
```

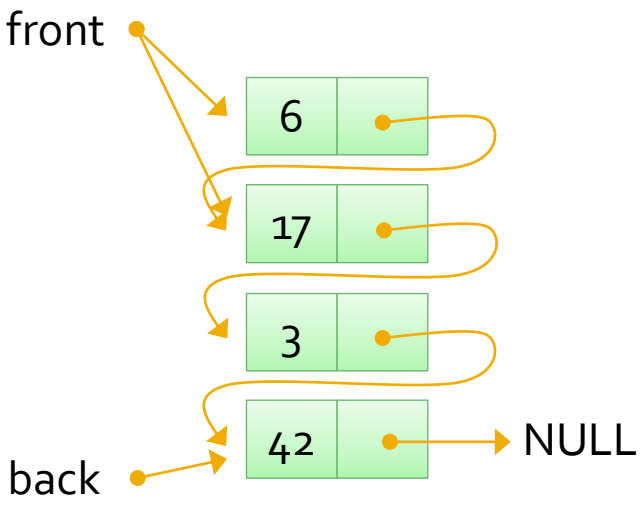


List Queue Example



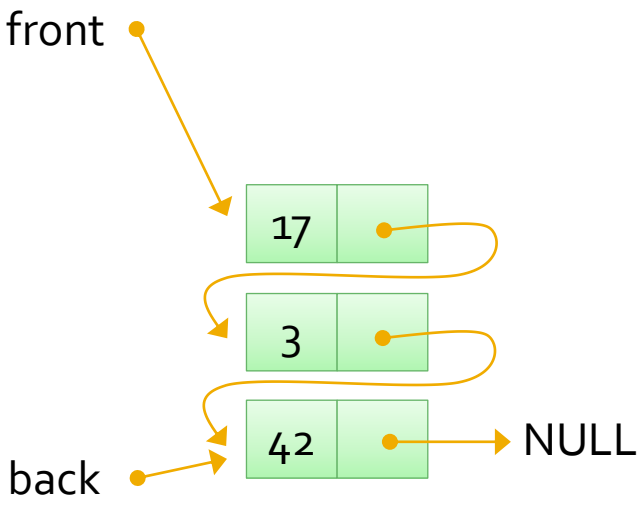

```
Queue q;
q.insert(6);
q.insert(17);
q.insert(3);
q.insert(42);
```

List Queue Example




```
Queue q;
q.insert(6);
q.insert(17);
q.insert(3);
q.insert(42);
q.remove();
```

List Queue Example




```
Queue q;
q.insert(6);
q.insert(17);
q.insert(3);
q.insert(42);
q.remove();
```

Other Simple Data Structures

Deques

- A deque is a double ended queue
  - That allows items to be inserted and removed from either end
  - Its pronounced *deck*, not *deke*
    - Also not to be confused with de-queue, queue removal
- Deque implementations
  - Circular array, similar to the queue implementation
  - Linked List
    - Singly linked list implementations are not efficient

So use a doubly linked list

Deque Methods

- Deque insertion and removal methods
  - insertFront
  - insertBack
  - removeFront
  - removeBack
- A deque could be used to implement both stacks and queues
  - Queue – use insertBack and removeFront
  - Stack – use insertFront and removeFront

Hang on a moment …

How should we design a Deque implementation of a stack?

Implementing one Class with Another

- Assume that we need to create a Stack class and already have a working and tested Deque class
- We could rewrite the Deque class
  - So it only inserts and removes from the front
  - And rename the methods to comply with the Stack interface
- Rewrite all of the modules that are using the Stack
  - They call *insertFront* and *removeFront* instead of *push* and *pop*
- Or write a class that implements a Stack but **uses** a Deque object to do so
- An adapter is a *design pattern*
  - A solution to a common design problem

Requires refactoring and re-testing

Bad!

Referred to as an Adapter class

Adapter Design Pattern

```
class Deque
{
    // …
    void insertFront(int x);
    void insertBack(int x);
    int removeFront();
    int removeBack();
    // …
};
```

Existing Deque class

The **adaptee**

```
class Stack
{
public:
    // …
    void push(int x);
    int pop();
    // …
private:
    Deque dq;
    // …
};
```

The Stack class is the **adapter**, also known as a **wrapper** class

Note that the Deque is part of its implementation so is *private*

Stack methods just call the methods of the Deque object

Referred to as **delegation**

```
void Stack::push(int x);
{
    dq.insertFront(x);
}

int Stack::pop();
{
    return dq.removeFront();
}
```

Priority Queues Introduction

- Items in a priority queue are given a priority
  - Could be numerical or something else
  - The highest priority item is always removed first
- Can items be inserted and removed *efficiently* from a priority queue?
  - Using an array, or
  - Using a linked list?
- Note that items are not removed based on the order in which they are inserted
- We will return to priority queues later in the course

Uses

system requests

Dijkstra's algorithm