CSCI 132:
Basic Data Structures and Algorithms

Queues (Linked List implementation)

Reese Pearsall

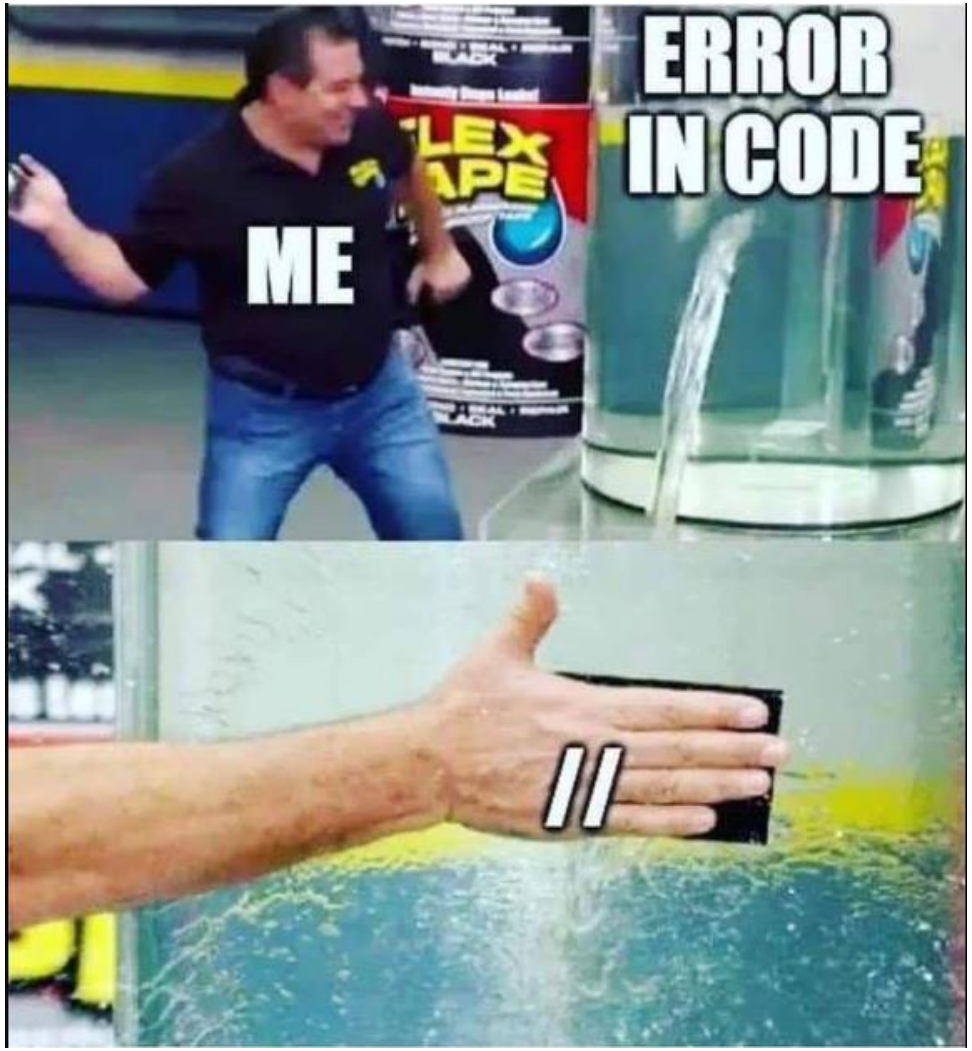
Spring 2023

Other Announcements

Lab 8 due tomorrow @ 11:59 PM

Program 3 due Sunday 4/2

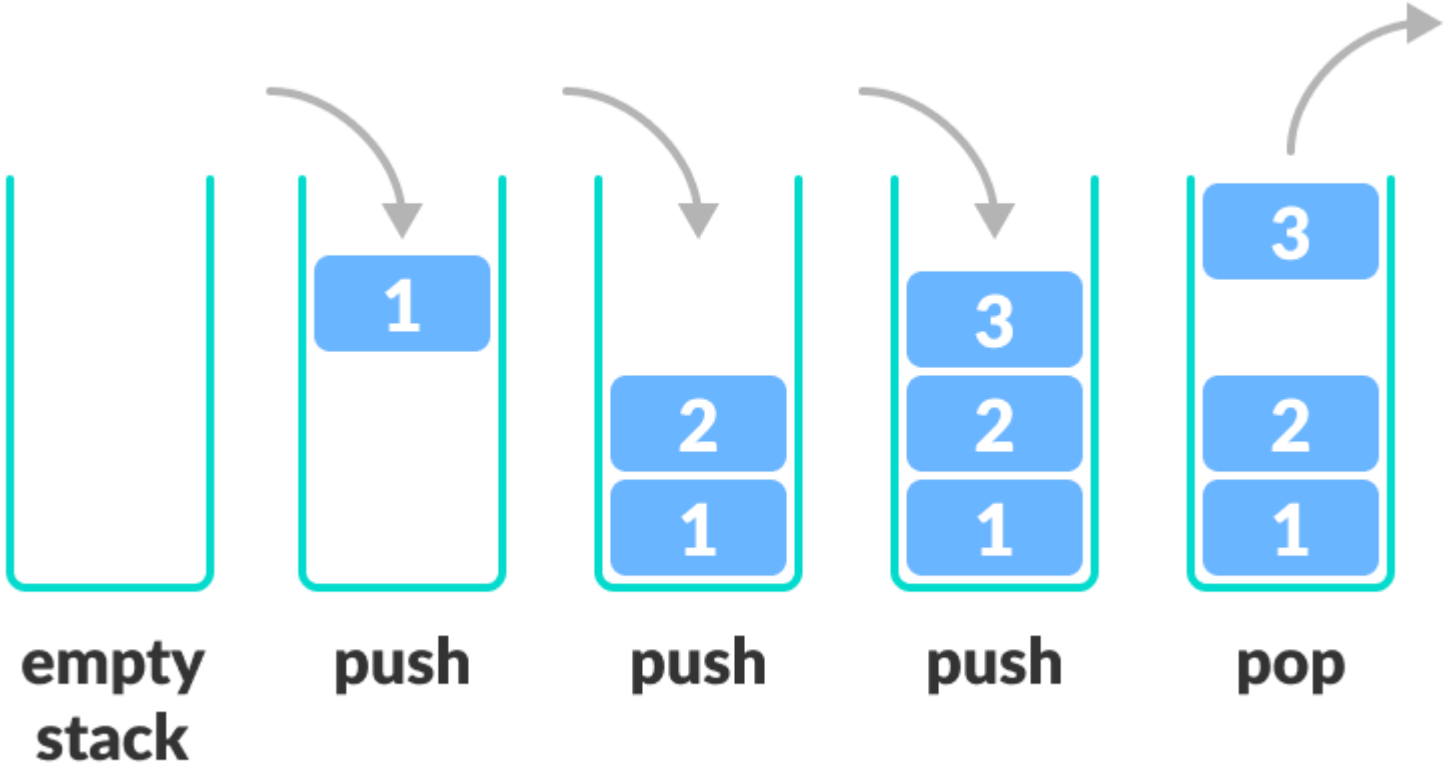
Friday's lecture will likely be a help session for Program 3



A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle

We can:

- Add an element to the top of the stack (**push**)
- Remove the top element (**pop**)



```
public void pop() {
    if(this.size == 0) {
        return;
    }
    else {
        this.data.removeFirst();
        this.top_of_stack =
        this.data.getFirst();
        this.size--;
    }
}
```

Stack has no data!!

If `pop()` is called on a stack with one element, our program might crash

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
       this.data.get(i).printInfo();
    }
}
```

Running time?

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) { O(n)
       this.data.get(i).printInfo();
    }
}
```

Running time?

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
        this.data.get(i).printInfo();
    }
}
```

O(n)

O(???)

The running time of this operation depends how the `get()` operation is implemented

Running time?

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
        this.data.get(i).printIn
    }
}
```

O(n)

Running time?

```
public T get(int index)
{
  checkBoundsExclusive(index);
  return getEntry(index).data;
}
```

```
Entry<T> getEntry(int n)
{
  Entry<T> e;
  if (n < size / 2)
    {
      e = first;
      // n less than size/2, iterate from start
      while (n-- > 0)
        e = e.next;
    }
  else
    {
      e = last;
      // n greater than size/2, iterate from end
      while (++n < size)
        e = e.previous;
    }
  return e;
}
```

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
        this.data.get(i).printIn
    }
}
```

O(n)

Running time?

```
public T get(int index)
{
  checkBoundsExclusive(index);
  return getEntry(index).data;
}
```

```
Entry<T> getEntry(int n)
{
  Entry<T> e;
  if (n < size / 2)
    {
      e = first;
      // n less than size/2, iterate from start
      while (n-- > 0)
        e = e.next;
    }
  else
    {
      e = last;
      // n greater than size/2, iterate from end
      while (++n < size)
        e = e.previous;
    }
  return e;
}
```

O(N/2)

O(N/2)

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
        this.data.get(i).printIn
    }
}
```

O(n)

Running time?

```
public T get(int index)
{
  checkBoundsExclusive(index);
  return getEntry(index).data;
}
```

```
Entry<T> getEntry(int n)
{
  Entry<T> e;
  if (n < size / 2)
    {
      e = first;
      // n less than size/2, iterate from start
      while (n-- > 0)
        e = e.next;
    }
  else
    {
      e = last;
      // n greater than size/2, iterate from end
      while (++n < size)
        e = e.previous;
    }
  return e;
}
```

O(N)

O(N)

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
        this.data.get(i).printIn
    }
}
```

O(n)

Running time?

```
public T get(int index)
{
  checkBoundsExclusive(index);
  return getEntry(index).data;
}
```

```
Entry<T> getEntry(int n)
{
  Entry<T> e;
  if (n < size / 2)
    {
      e = first;
      // n less than size/2, iterate from start
      while (n-- > 0)
        e = e.next;
    }
  else
    {
      e = last;
      // n greater than size/2, iterate from end
      while (++n < size)
        e = e.previous;
    }
  return e;
}
```

Total running time of get(): **O(N)**

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
       this.data.get(i).printInfo();
    }
}
```

O(n)

O(n)

Running time?

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) { O(n)
       this.data.get(i).printInfo();  O(n)
    }
}
```

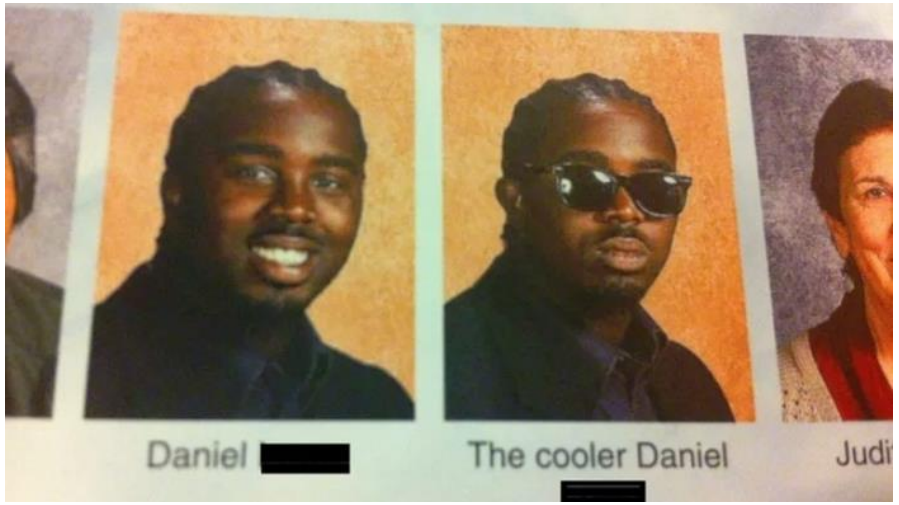
Running time?

$O(n^2)$

Can we do better?

Old Way O(n^2)

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
       this.data.get(i).printInfo();
    }
}
```



New Way

```
for(String each: this.data) { O(n)
    System.out.println(each); O(1)
}
```

Running time?

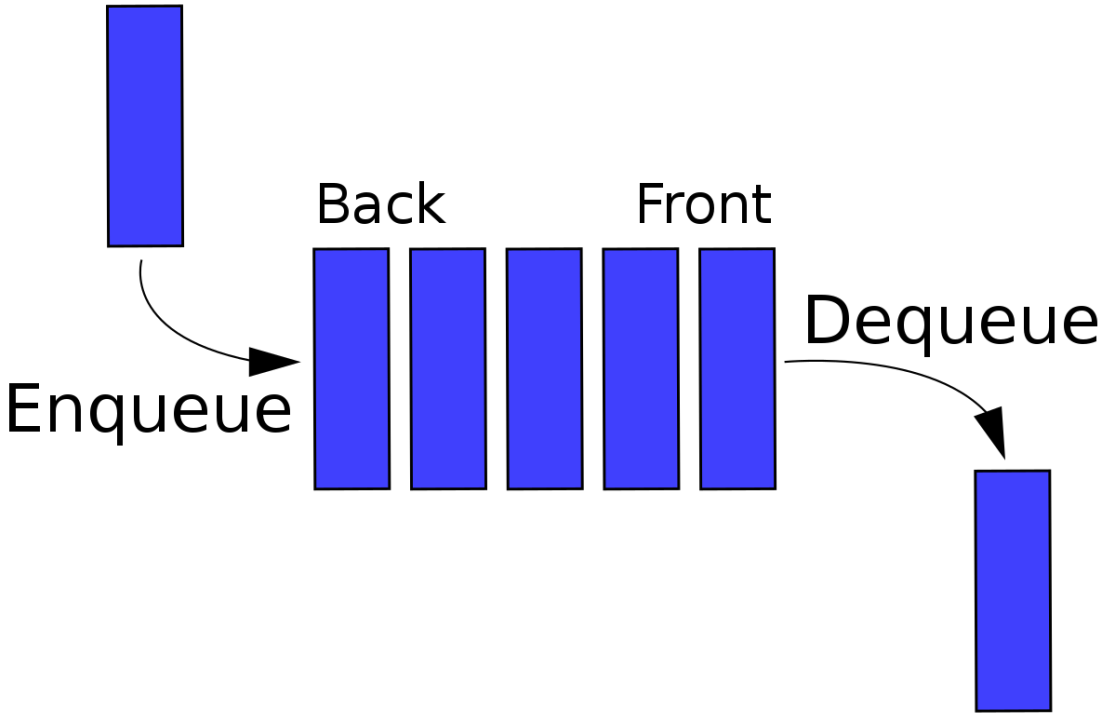
O(n)

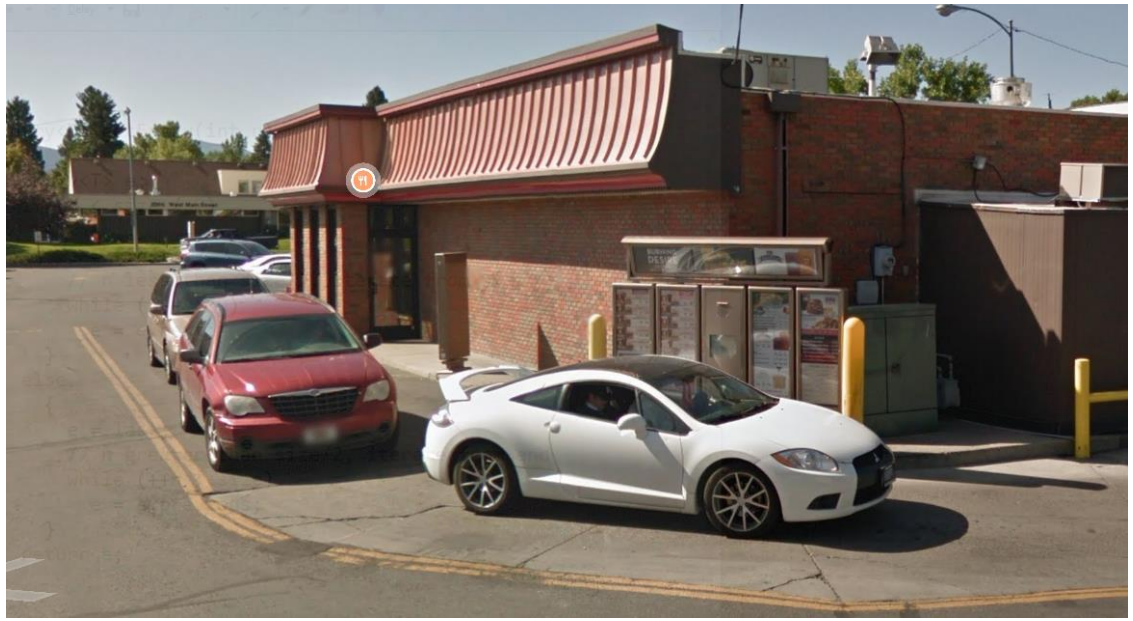
Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Stack Runtime Analysis

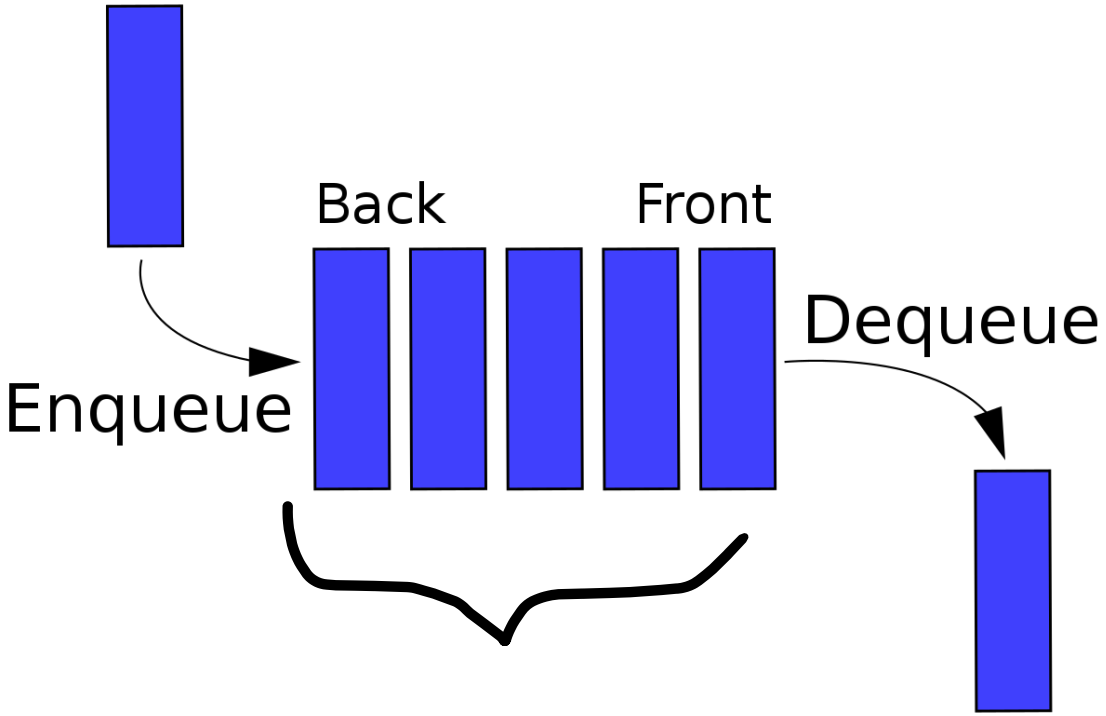
Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion





A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion

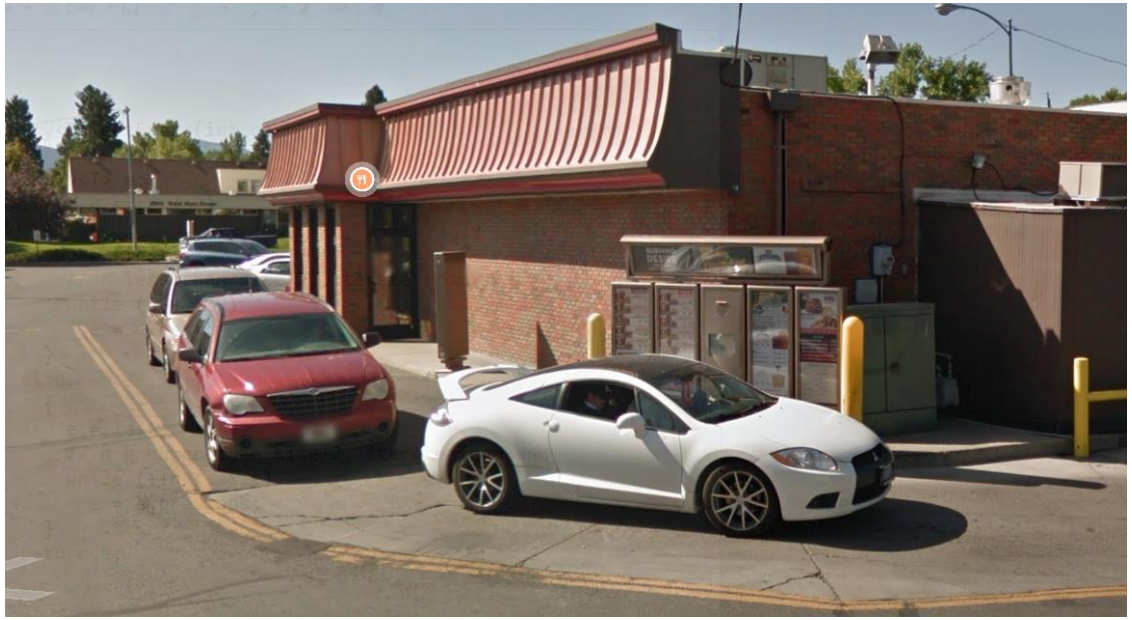


Once again, we need a data structure to hold the data of the queue

- Linked List (today)
- Array (tomorrow)

Elements get added to the **`Back`** of the Queue.

Elements get removed from the **`Front`** of the queue



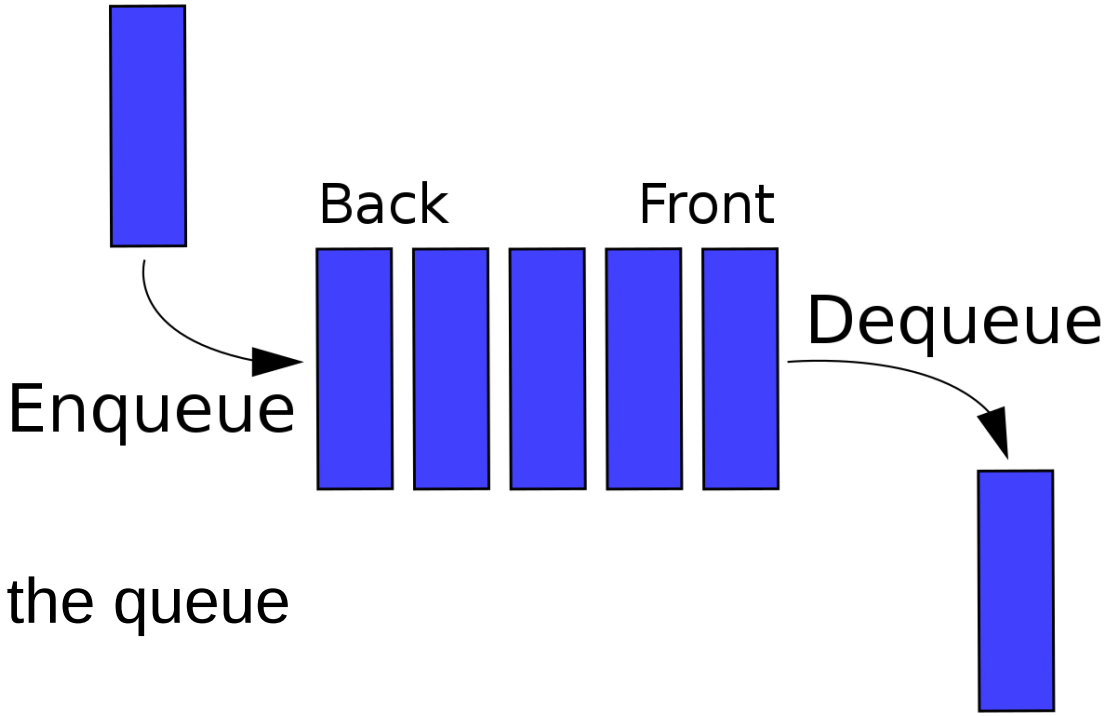
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion

The Queue ADT has the following methods:

Enqueue- Add new element to the queue

Dequeue- Remove element from the queue

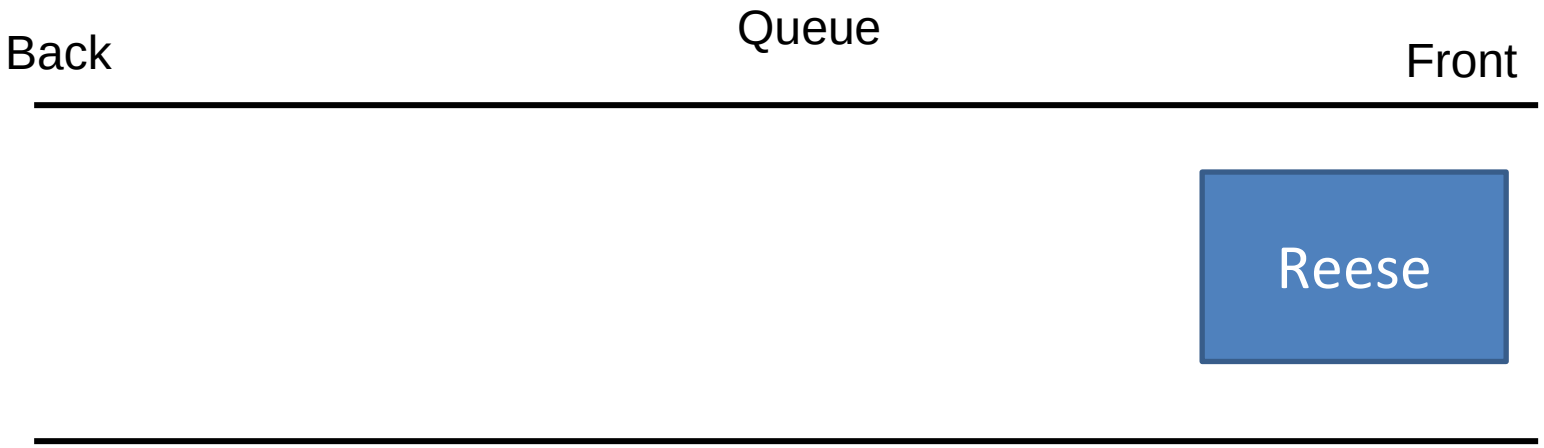
** Always remove the front-most element



Peek()- Return the element that is at the front of the queue

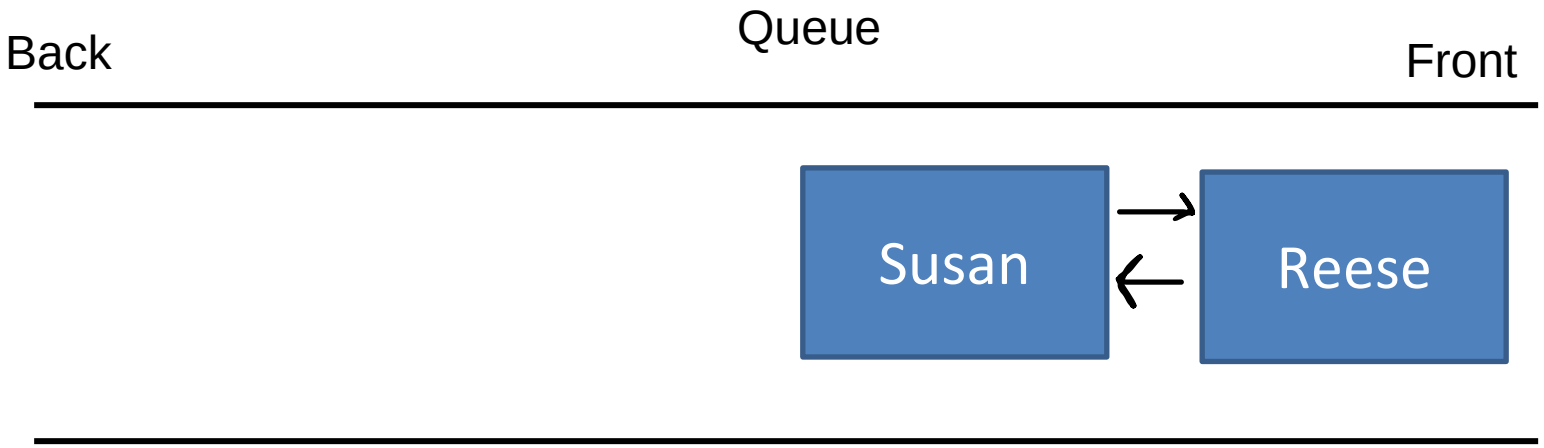
IsEmpty() – Returns true if queue is empty, returns false is queue is not empty

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



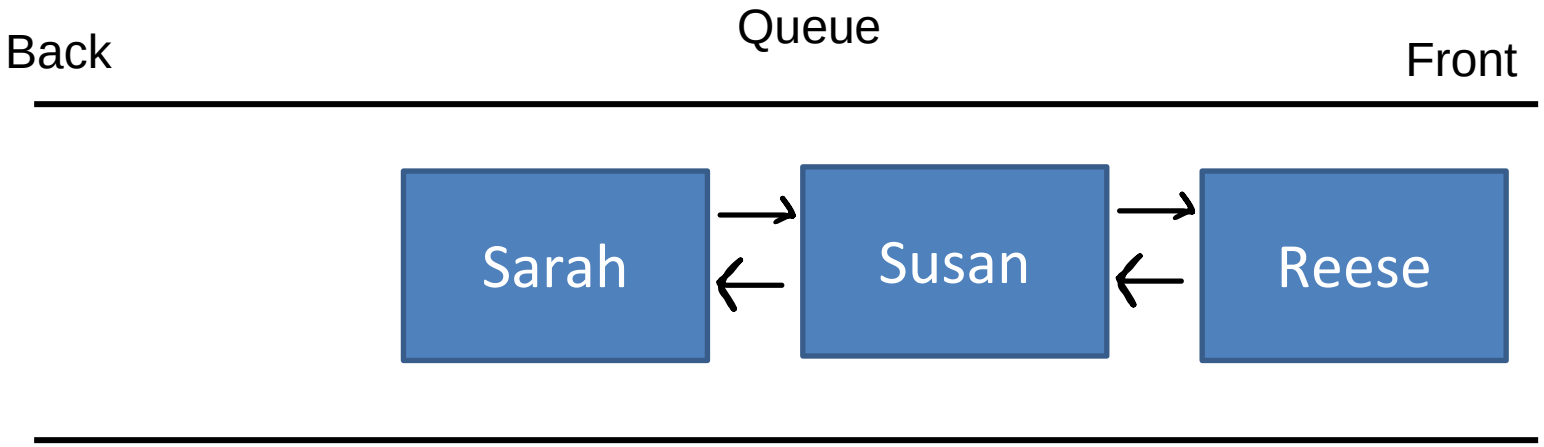
queue.enqueue(“Reese”);

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



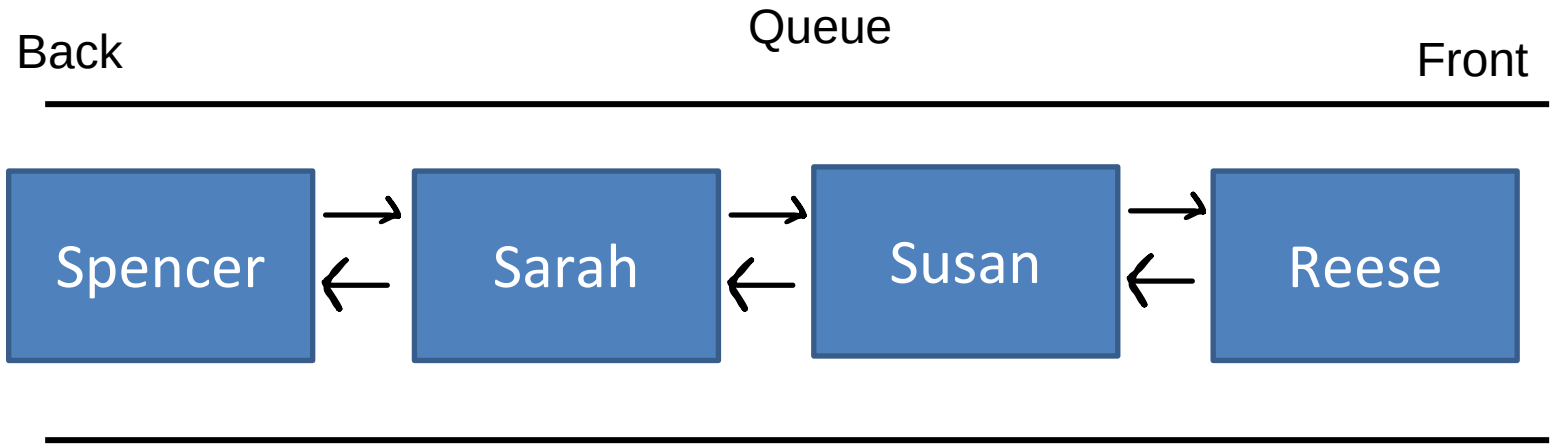

```
queue.enqueue("Reese");
queue.enqueue("Susan");
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



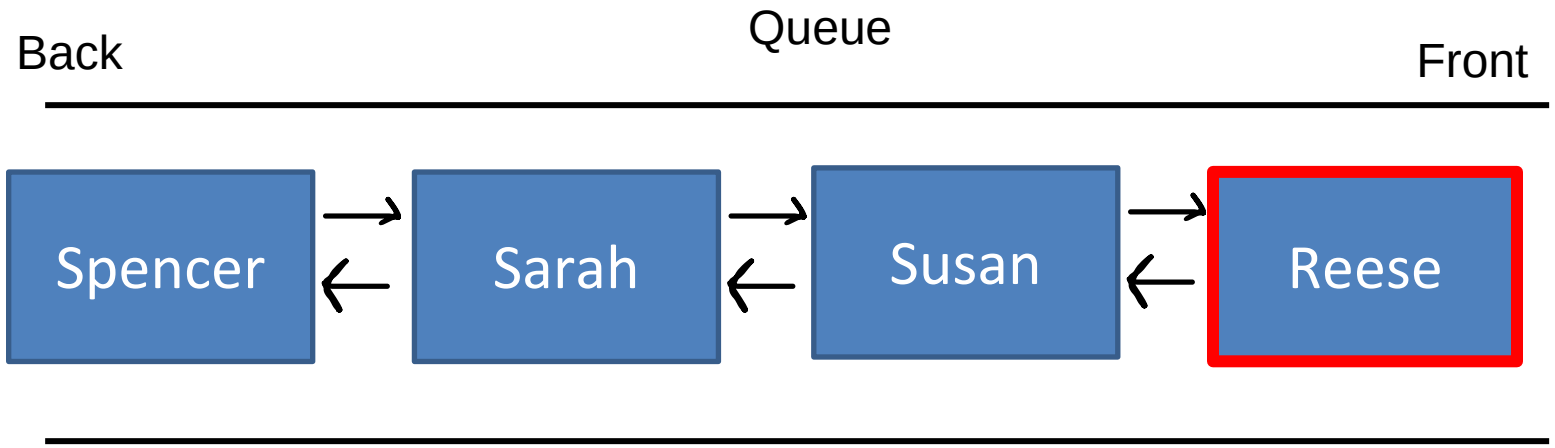

```
queue.enqueue("Reese");
queue.enqueue("Susan");
queue.enqueue("Sarah");
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
queue.enqueue("Reese");
queue.enqueue("Susan");
queue.enqueue("Sarah");
queue.enqueue("Spencer");
```

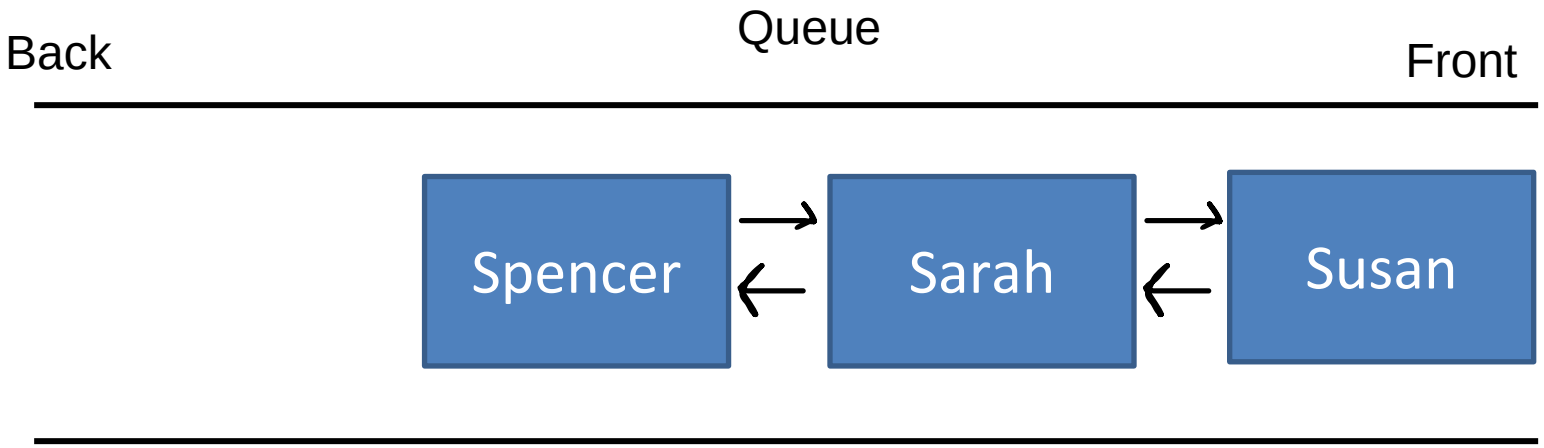
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
queue.enqueue("Reese");
queue.enqueue("Susan");
queue.enqueue("Sarah");
queue.enqueue("Spencer");
```

```
queue.deque()
```

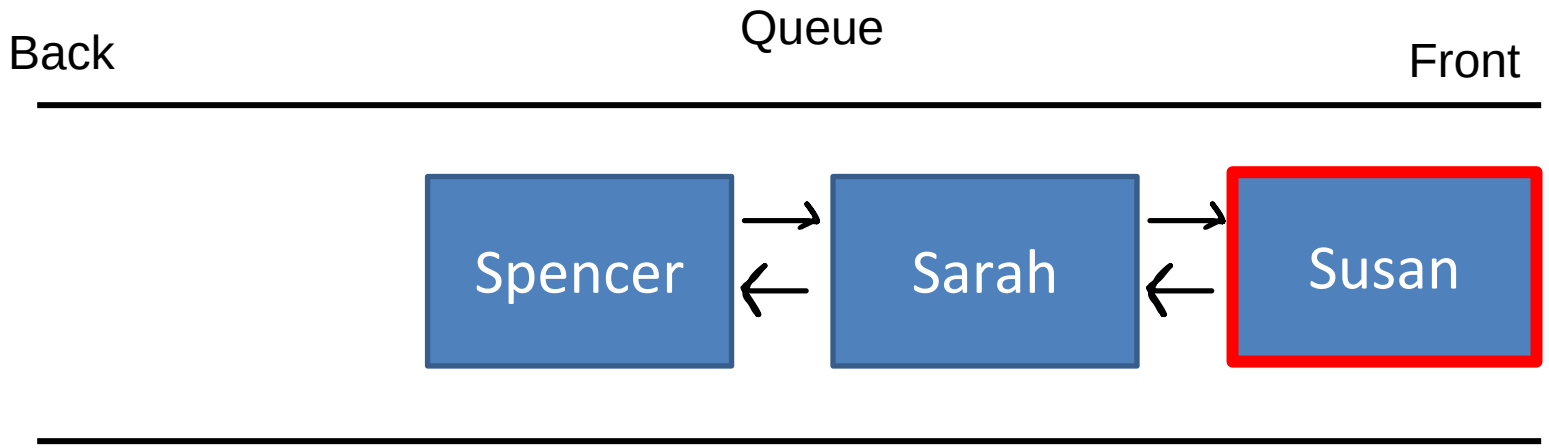
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
queue.enqueue("Reese");
queue.enqueue("Susan");
queue.enqueue("Sarah");
queue.enqueue("Spencer");
```

```
queue.deque()
```

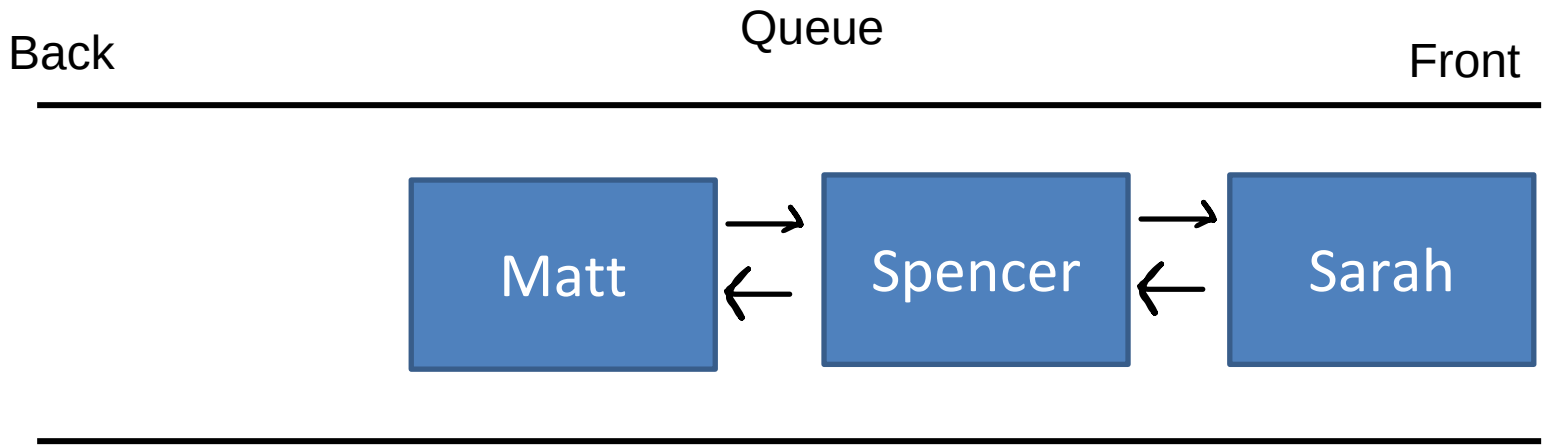
A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion




```
queue.enqueue("Reese");
queue.enqueue("Susan");
queue.enqueue("Sarah");
queue.enqueue("Spencer");
```

```
queue.deque()
queue.deque()
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion

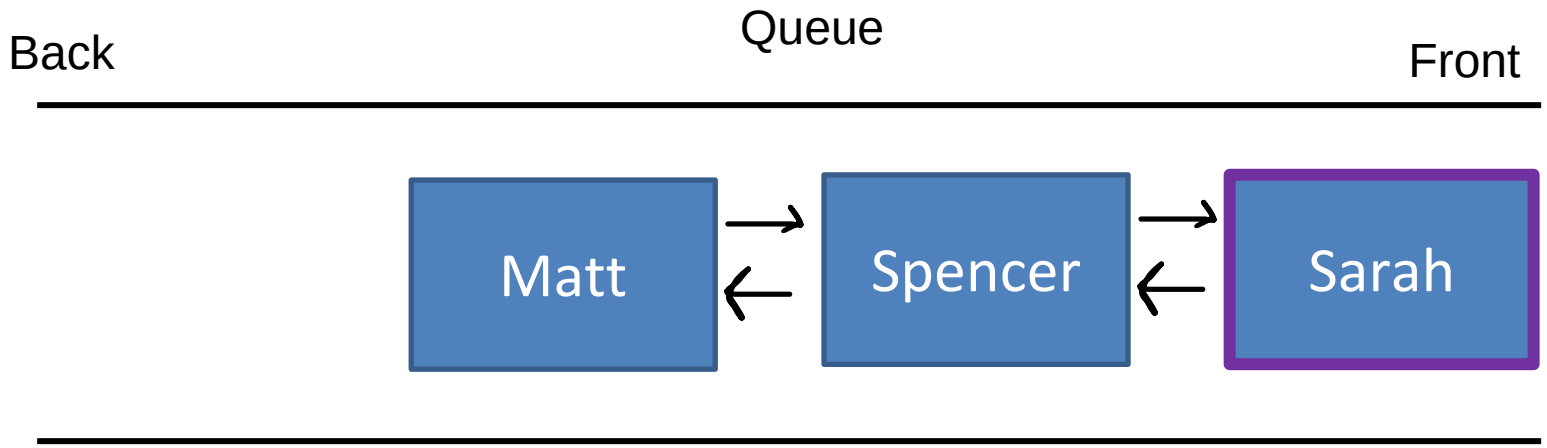



```
queue.enqueue("Reese");
queue.enqueue("Susan");
queue.enqueue("Sarah");
queue.enqueue("Spencer");
```

```
queue.deque()
queue.deque()

queue.enqueue("Matt");
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion

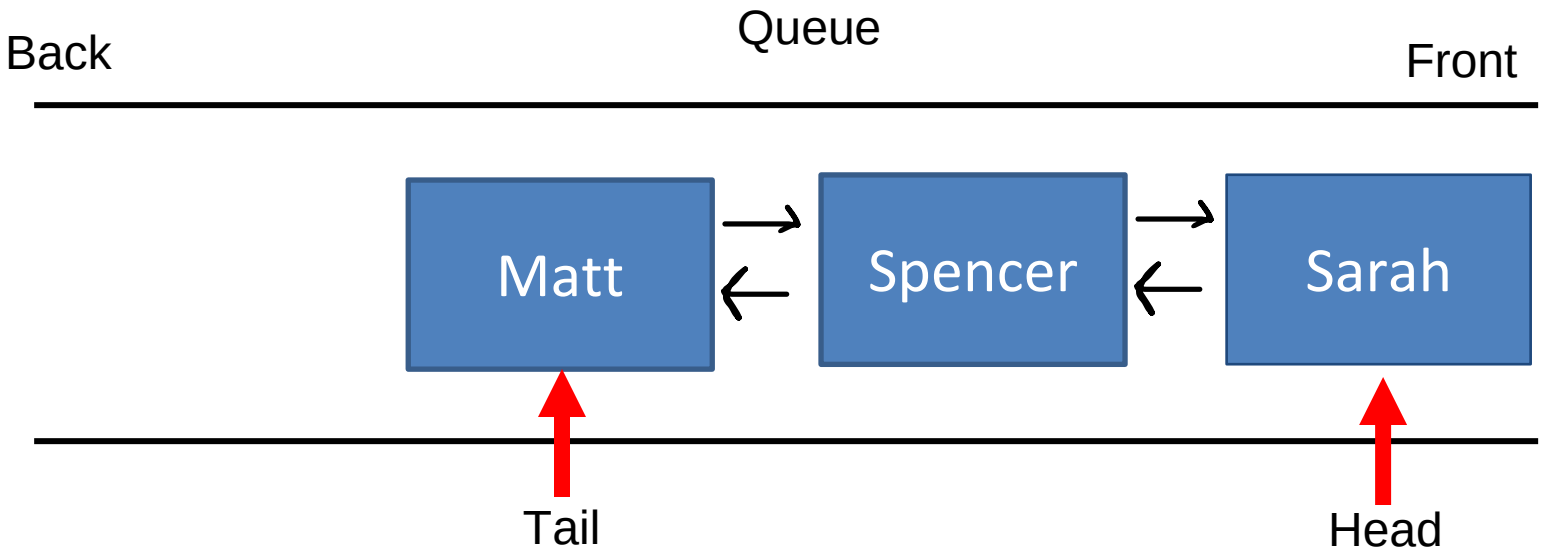



```
queue.enqueue("Reese");
queue.enqueue("Susan");
queue.enqueue("Sarah");
queue.enqueue("Spencer");
```

```
queue.deque()
queue.deque()

queue.enqueue("Matt");
queue.peek()
    → "Sarah"
```

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion

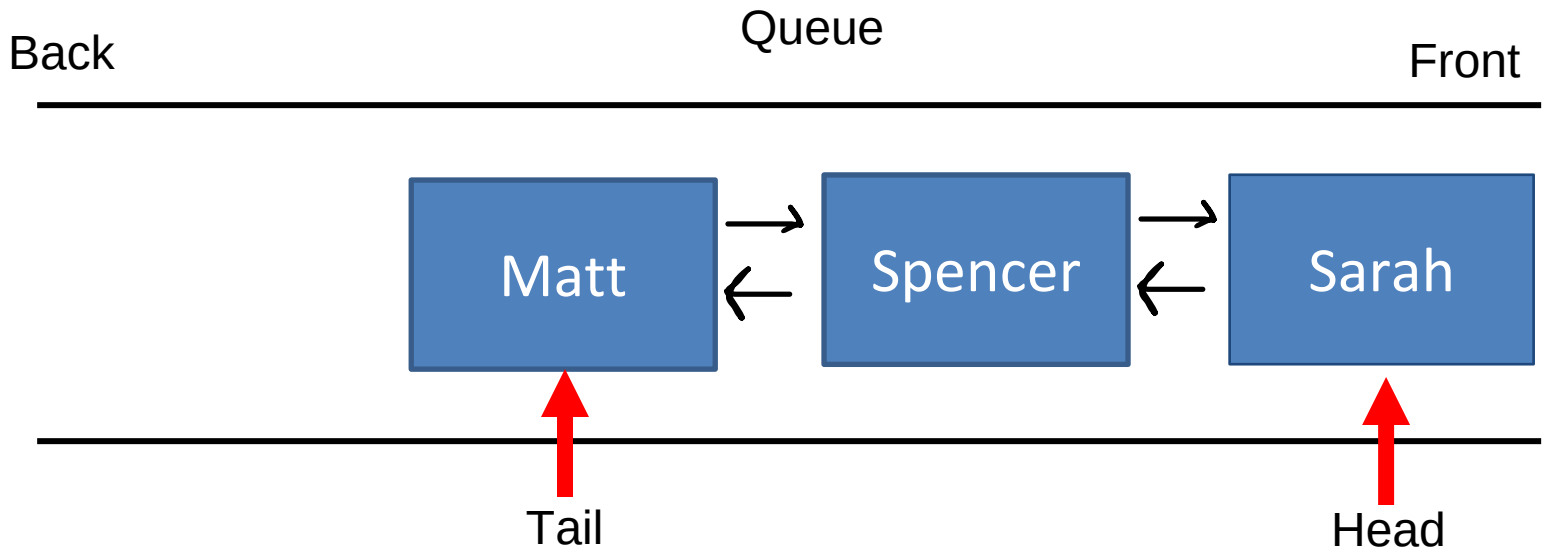


Linked List Implementation

When we enqueue, we add the element to ???

When we dequeue, we remove the element from ???

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion

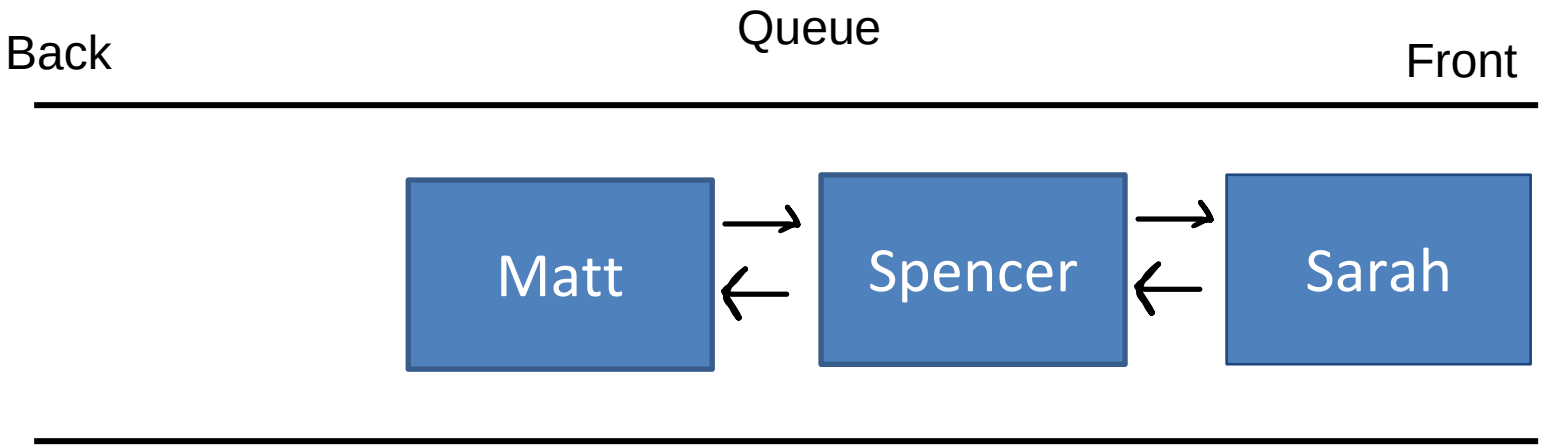


Linked List Implementation

When we enqueue, we add the element to the end of the linked list

When we dequeue, we remove the element from the beginning of the linked list

A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



As we use our queue, we need to keep track of a few things

- The **size** of the queue
- The **front** of the queue (not when we use LLs)
- The **rear** of the queue (not when we use LLs)