Queues

What is a queue?, implementing queues as linked lists, add and remove methods, working with Queue<E> and LinkedList<E>, creating queues through composition

What is a queue?

A **queue** is another special kind of list with a restricted set of operations.

- A queue can add new items only at the back, which is called the **rear** of the queue.
- A queue can remove only its first item, which is at the **front** of the queue. Removing an item from the queue returns the front item.

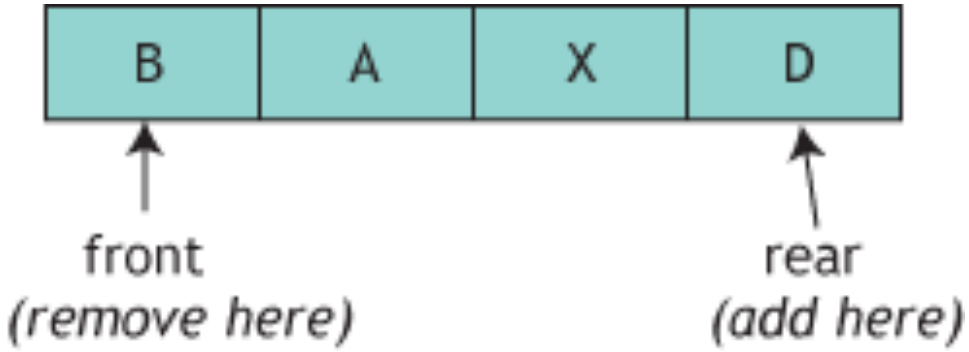
Queues are called first-in-first-out lists, or **FIFO** lists.

Queue operations are meant to be simple and efficient. All pure queue operations should have complexity O(1).

Queues are like…

- Lines in grocery stores
- Lines at tollbooths
- Printer queues

You always remove the one off the front. You always add new ones to the rear.



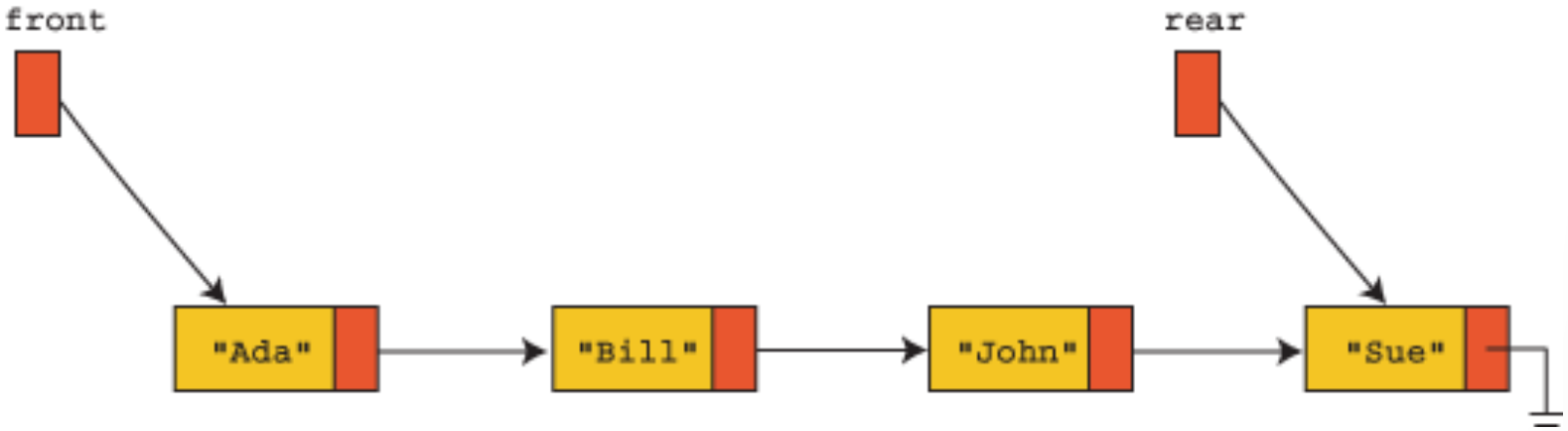
Standard queue operations

The standard queue operations include:

- **add** (or **enqueue**): New elements are added only at the rear of the queue.
- **remove** (or **dequeue**): A queue can remove only its first element, which is at the front of the queue. Removing an element from the queue returns that element.
- **isEmpty**: A queue can tell whether it is empty.
- **peek**: A queue can tell what its front element is without removing it.

Implementing queues via linked lists

- Need fast access to the front and to the rear.
- Solution: Use two pointers.



Setting up the queue code

```
public class MyQueue<E> {
   Node front;  // Points to the front of the queue
   Node rear;   // Points to the rear of the queue

   // The old Node class goes here

   public MyQueue () {
      front = null;
      rear = null;
   }

   public boolean isEmpty() {
      return front == null;
   }
}
```

Implementing add

Take care adding to empty queues.

```
public void add(E data){
   if (front == null) {  // Empty queue
      front = new Node(data, null);
      rear = front;  // Now the queue has one item.
   }
   else { // Not empty queue
      rear.next = new Node(data, null);
      rear = rear.next;
   }
}
```

Implementing remove

Take special care when the queue becomes empty.

```
public E remove() {
   if (front == null)  // Empty queue
      return null;     // Or throw NoSuchElementException
   // Not an empty queue. Take the element from the front.
   E toGo = front.data;
   front = front.next;
   if (front == null)  // Happens with a 1-item queue.
      rear = null;
   return toGo;
}
```

Using a queue to reverse a stack

```
PureStack<String> stack = new PureStack<String>();
stack.push("A");
stack.push("B");
stack.push("C");  // ["C", "B", "A"]

MyQueue<String> queue = new MyQueue<String>();

while (!stack.isEmpty()) {
   String s = stack.pop();
   queue.add(s);
}                 // queue is ["C", "B", "A"]

while (!queue.isEmpty()) {
   String s = queue.remove();
   stack.push(s);
}                 // stack is {"A', "B", "C"]
```

top

front rear

top

Understanding java.util.Queue

java.util. Queue<E> from the Java collection framework is an interface.

Methods declared in Queue<E> are:

- **boolean add(E x)**. Adds x to the queue. Throws IllegalStateException if there's no room.
- **void offer(E x)**. Same as add() but will not throw an exception.
- **E element()**. Returns but does not remove the front queue element. Throws NoSuchElementException if empty.
- **E peek()**. Same as element(), but return null if the queue is empty.
- **E remove()**. Removes the front element and returns it. Throws NoSuchElementException if empty.
- **E poll()**. Same as remove() but returns null if the queue is empty.

Using java.util.Queue<E>

Several collection classes implement Queue<E>, including LinkedList<E>. Sample code:

```
Queue<String> waitList = new LinkedList<String>();
waitList.add("Al");
waitList.add("Bob");
waitList.add("Mary"); // ["Al","Bob","Mary"]

System.out.println(waitlist.peek()).   // Al
```

- **`waitList`** can use all the methods defined in Queue<E> directly.
- If you cast **`waitList`** to LinkedList<String>, you can also use all the LinkedList<E> methods. In that case, you no longer have a pure queue.

Creating a queue using composition and delegation

```
public class PureQueue<E> {
   private java.util.Queue<E> q;

   public PureQueue()         { q = new LinkedList<E>(); }
   public void add(E e)       { q.add(e); }
   public E remove()          { return q.remove(); }
   public E peek()            { return q.peek(); }
   public boolean isEmpty()   { return q.isEmpty(); }
}
```