Keeping Order: Stacks, Queues, & Deques

Stacks

Stacks

- A **stack** is a last in, first out (LIFO) data structure
- Primary Operations:
  - push() — add item to top
  - pop() — return the top item and remove it
  - peek() — return the top item but don't remove it
  - isEmpty() — return **True** *iff* the stack is empty
- java.util.Stack is similar to our stack implementation but more fully featured
- The textbook's version of Stack is similar but has different naming.

Stacks: Why?

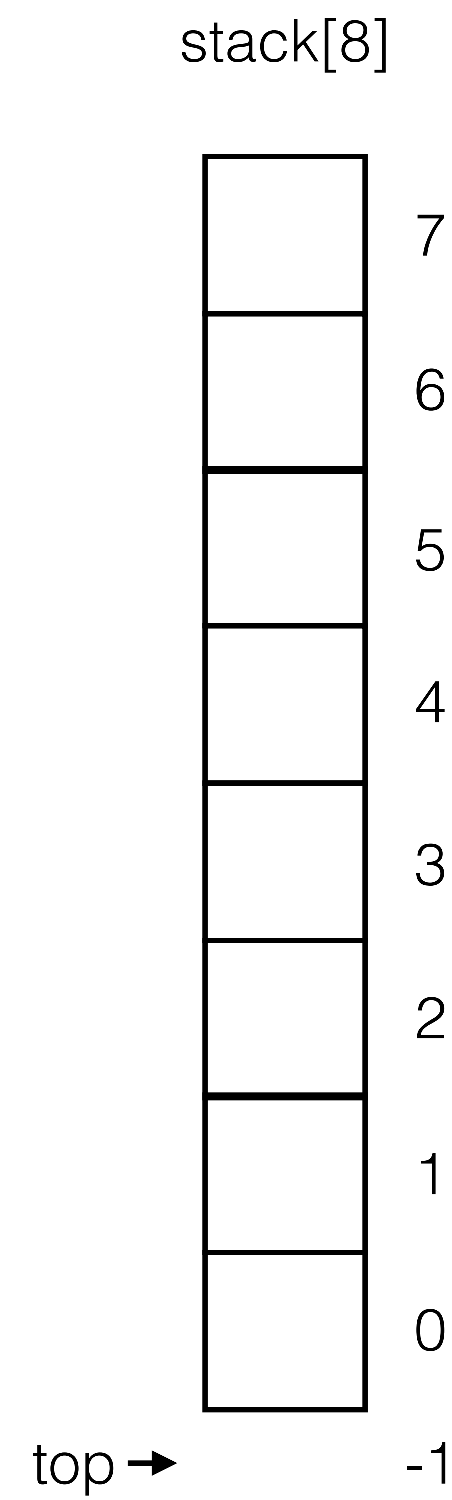
- Reversing strings/lists is easy with stacks!
  - *push* everything on in order then *pop* them all off.
- Depth-first search (DFS) — later…
  - Maze and graph searching
  - Thank back to lab 1! :)
- Matching parenthesis, braces, open/close tags for HTML, etc.
  - think about how java expects matching **{** … **}** and **(** … **)**

Demo: MatchParens.java

- Find open paren — push()
- Find close paren — check top of stack for “friend” (a.k.a. partner)

Implementing Stacks

- **Q:** How do we implement a stack?
- **Simple option:** fixed-size array.
  - instance variables (**int[] stack** and **int top**)
  - push()
    - top++
    - stack[top] = …
  - pop()
    - peek()
    - top--
- Pros: fast (operations are O(1)); space efficient
- Cons: array can fill up…

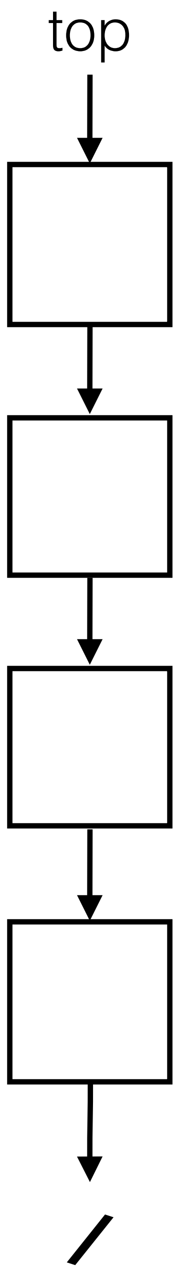


Implementing Stacks: ArrayList

- **Better option:** dynamic array.
  - instance variables (**ArrayLists<T> stack**)
  - push() — use add() to add to end
  - pop() — use remove() to remove from end
- Pros: ArrayList never becomes “full”; no need to keep track of “top” since ArrayList does this for us!
- Cons: push() could be non-constant time op. when the ArrayList has to grow to make more room.

Implementing Stacks: SLL

- **Great option:** Singly-Linked List.
  - instance variables (**Element<T> top**)
  - top of the stack is the head of the list
  - push() — adds item to front of list
  - pop() — removes item from front of list
- Pros: SLL never becomes “full”; all operations are O(1)
- Cons: need extra space for links in list but there is never wasted space for unused spaces in array/ArrayList.



Queues

Queues

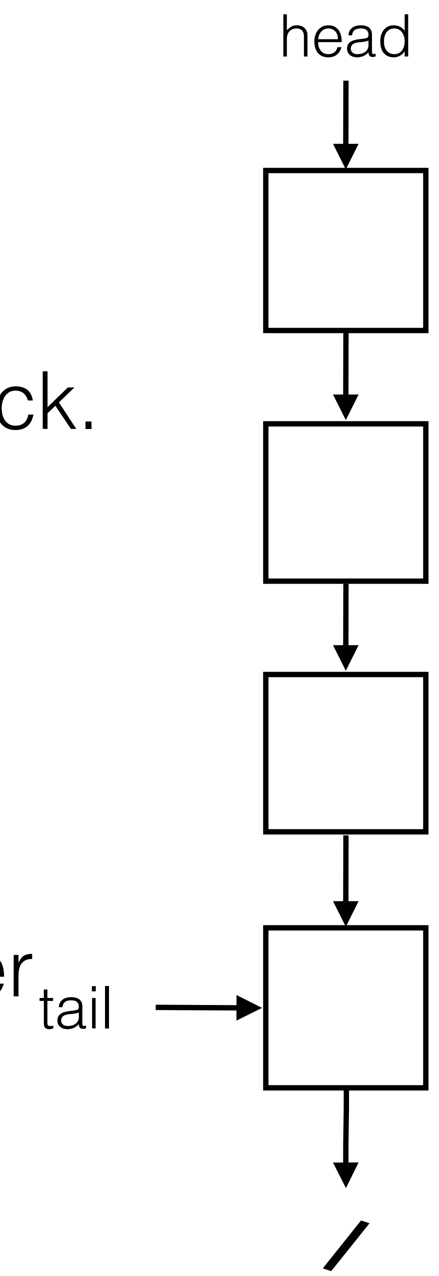
- A **queue** is a first in, first out (FIFO) data structure
- Primary Operations:
  - enqueue() — add at rear of the queue
  - dequeue() — remove and return the first item in the queue
  - front() — return the first item but don't remove it
  - isEmpty() — return **True** *iff* the queue is empty
- Java has a Queue interface but it uses non-conventional names (add()/offer(), element(), peek(), remove()/poll()).
- The 2 different alternatives for enqueue()/dequeue() are for one that uses exceptions to handle failed operations while the other doesn't

Queues: Why?

- Useful for simulations of lines (banks, toll booths, etc.)
- Useful for computer print queues
  - submit a document to be printed — enqueue()
  - print the document when it reaches the front — dequeue()
- Round-robin scheduling for threads/processes to run
  - dequeue() process — run for fixed period of time or until it blocks
  - enqueue() processes in the order they arrive to ensure fair sharing of CPU
  - processes that finish don't get enqueued again
- Used when searching (similar to stack)

Implementing Queues

- See: SLLQueue.java
- **Typical option:** Linked Lists.
  - instance variables (**Element<T> head, tail**)
  - front of the queue is the head of the list; tail is the back.
  - enqueue() — adds item to back of list
  - dequeue() — removes item from front of list
- Pros: Never becomes "full"; all operations are Θ(1).
- Cons: need extra space for links in list but there is never wasted space for unused spaces in array/ArrayList.



Implementing Queues

- Basic setup + front() & isEmpty()

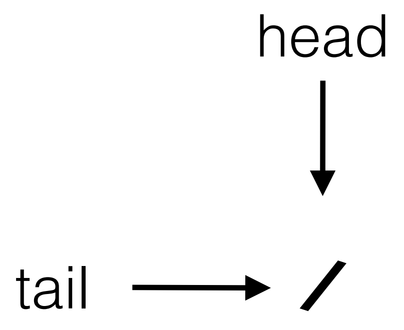
```
public class SLLQueue<T> implements SimpleQueue<T> {
    private Element<T> head;    // front of the linked list
    private Element<T> tail;    // tail of the linked list

    public SLLQueue()  {
        head = null;
        tail = null;
    }

    // ...

    public T front() throws Exception {
        if (isEmpty()) throw new Exception("empty queue");
        return head.data;
    }

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



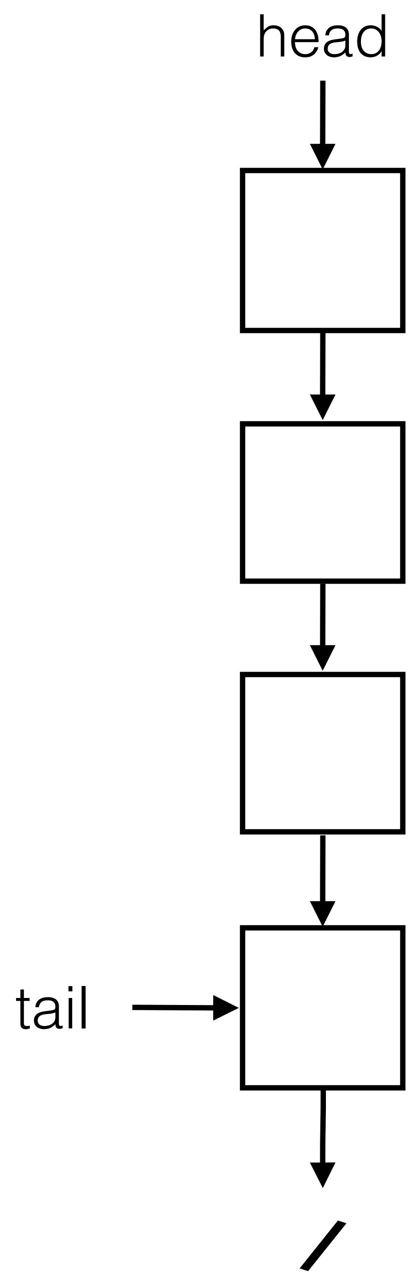
Implementing Queues

- enqueue()

```
public void enqueue(T item) {
    if (isEmpty()) {
        // first item
        head = new Element(item);
        tail = head;
    }
    else {
        tail.next = new Element(item);
        tail = tail.next;
    }
}
```

- dequeue()

```
public T dequeue() throws Exception {
    if (isEmpty()) throw new Exception("empty queue");
    T item = head.data;
    head = head.next;
    return item;
}
```



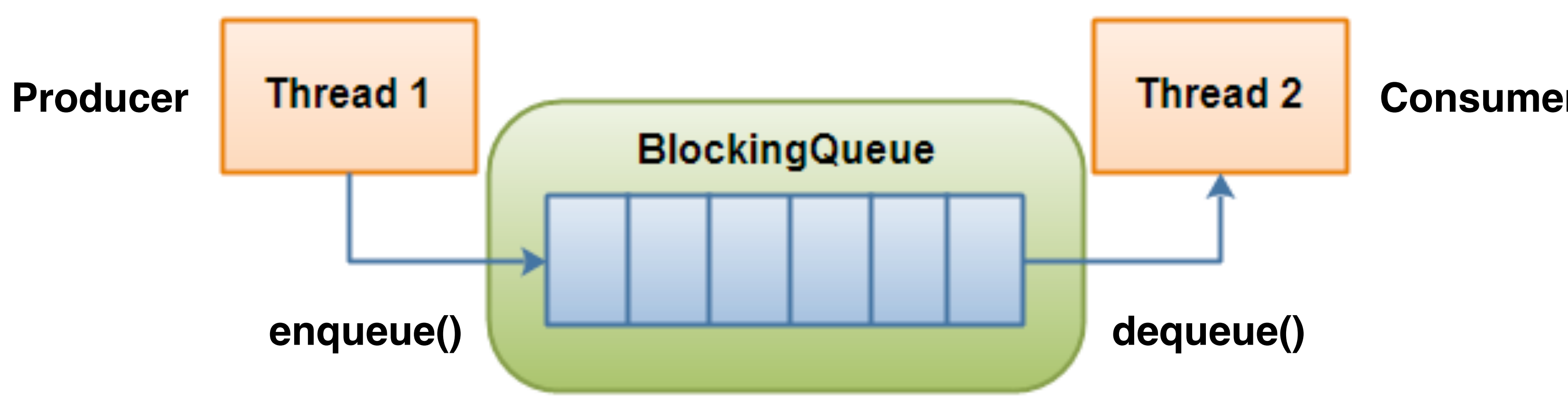
More on Queues

More on Queues

- There are *many* implementations of a Queue in Java
  - ArrayBlockingQueue,
  - ConcurrentLinkedQueue,
  - DelayQueue,
  - LinkedBlockingQueue, and
  - LinkedList
- LinkedList is probably the most common

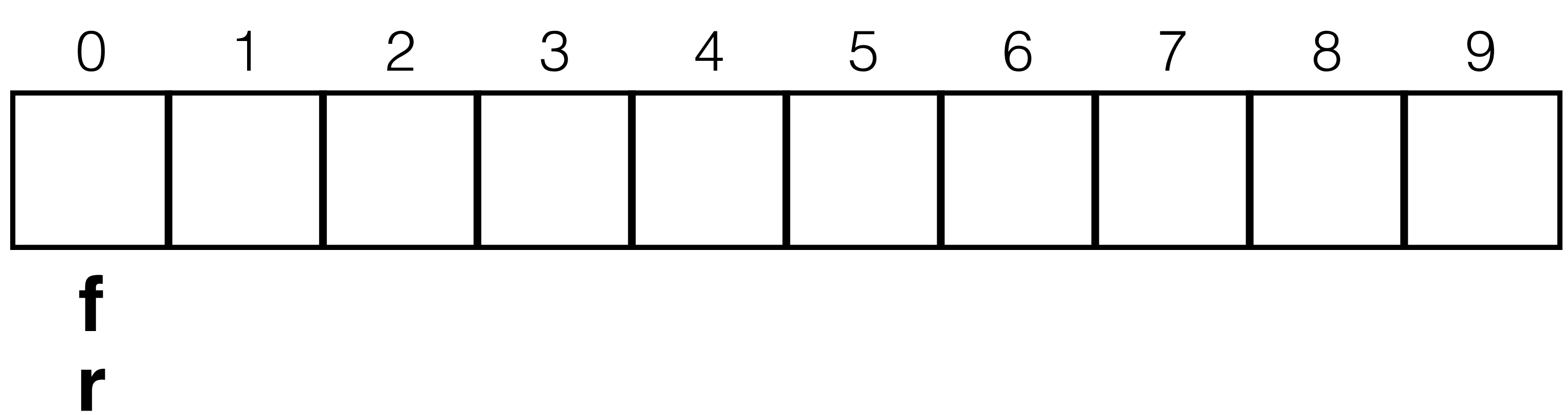
Blocking Queues

- A blocking queue is a **bounded buffer** used for interprocess communication (IPC).
- Think back to Producer/Consumer
  - we used a message box with 1 item capacity
  - with a queue, we could hold more items!
  - Producer calls enqueue(); sleep if not enough space
  - Consumer calls dequeue(); sleep if nothing to consume.
- Many ways to implement blocking queues…

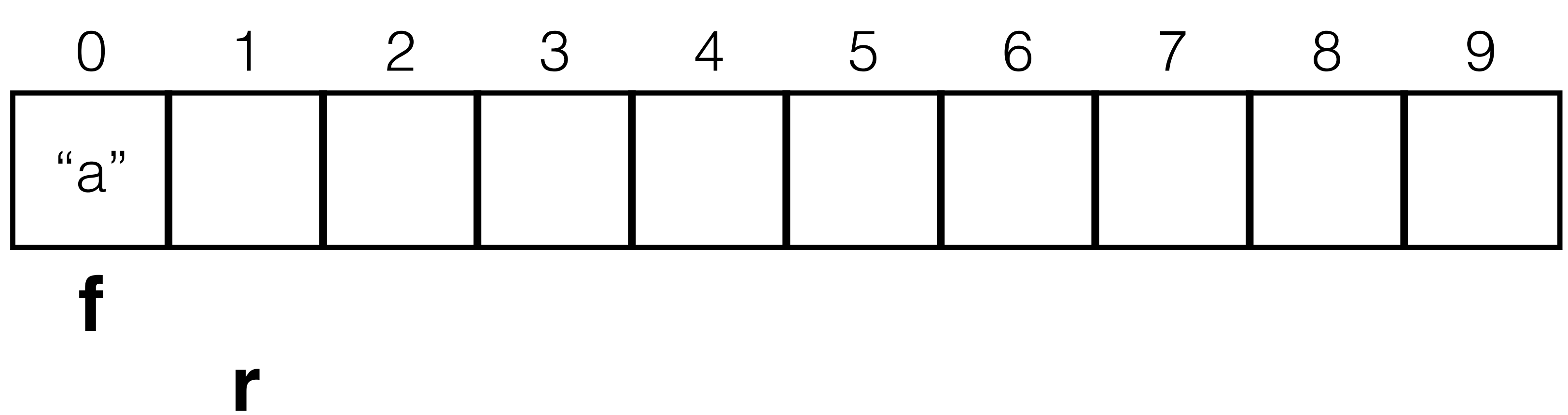


Blocking Queues

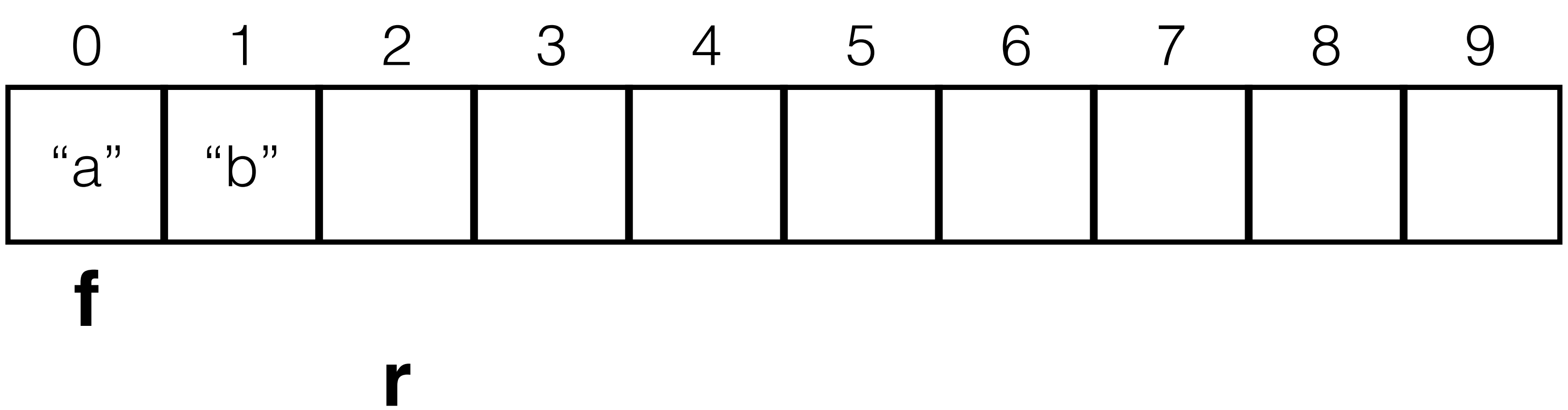
- Why not an array?
- Array-based queues might seem an odd choice…
- Why not use an ArrayList?
  - enqueue at end, dequeue from front — dequeue then requires us to shift all items left (forward)
  - enqueue at front, dequeue at end — enqueue then requires us to shift all items right (back)
- In array, keep track of indices, ***f*** and ***r*** (front and rear).
- For efficiency, when dequeuing at front, leave space empty, and “remember” that the front of the queue is now actually one space back



Blocking Queues

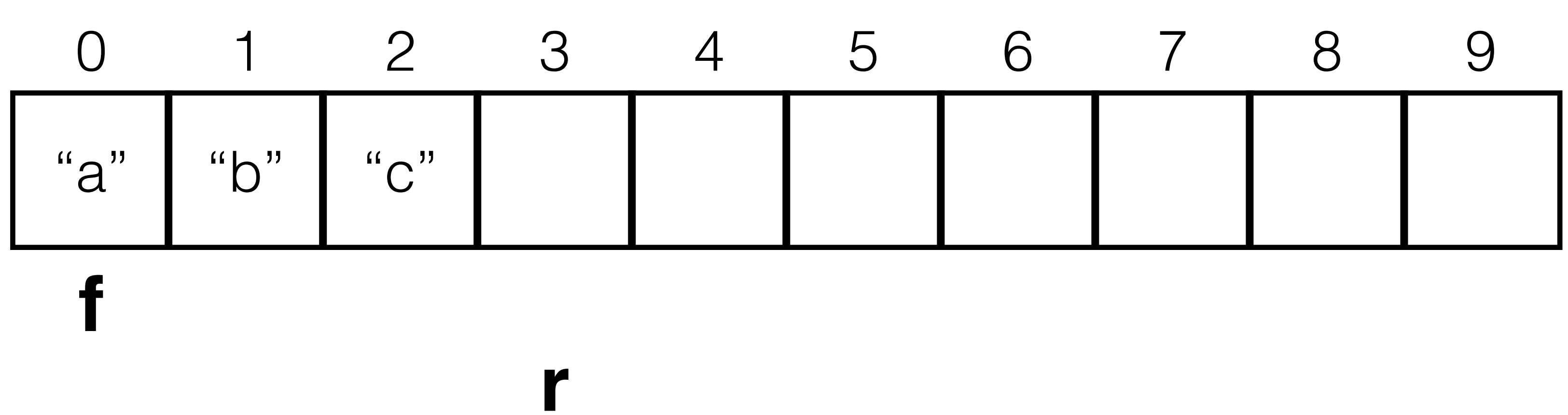


Blocking Queues



Blocking Queues

- *Skip ahead a bit…*



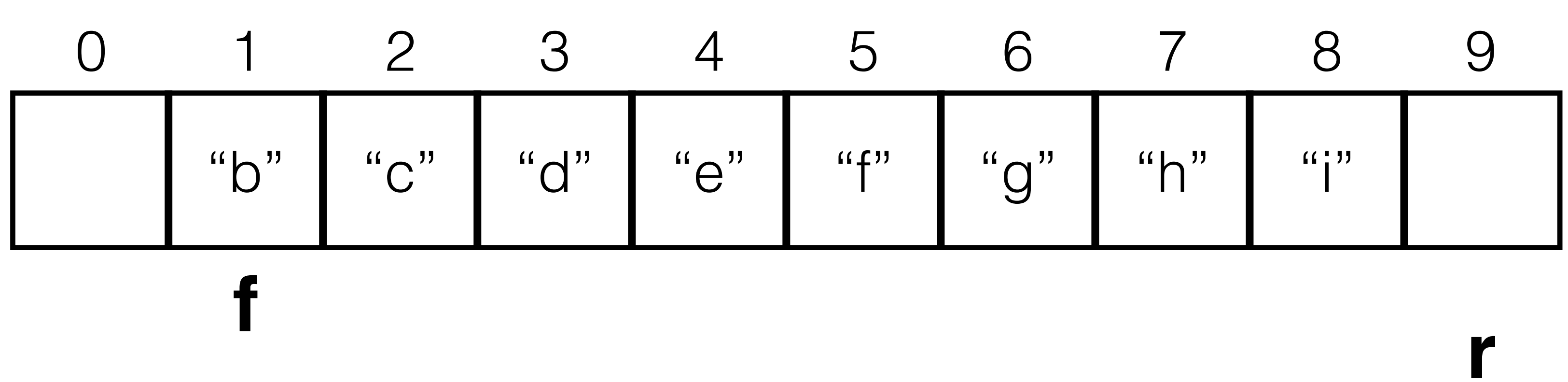
Blocking Queues

- *Now maybe some dequeue() calls are made…*

0	1	2	3	4	5	6	7	8	9
“a”	“b”	“c”	“d”	“e”	“f”	“g”	“h”	“i”	
f									**r**

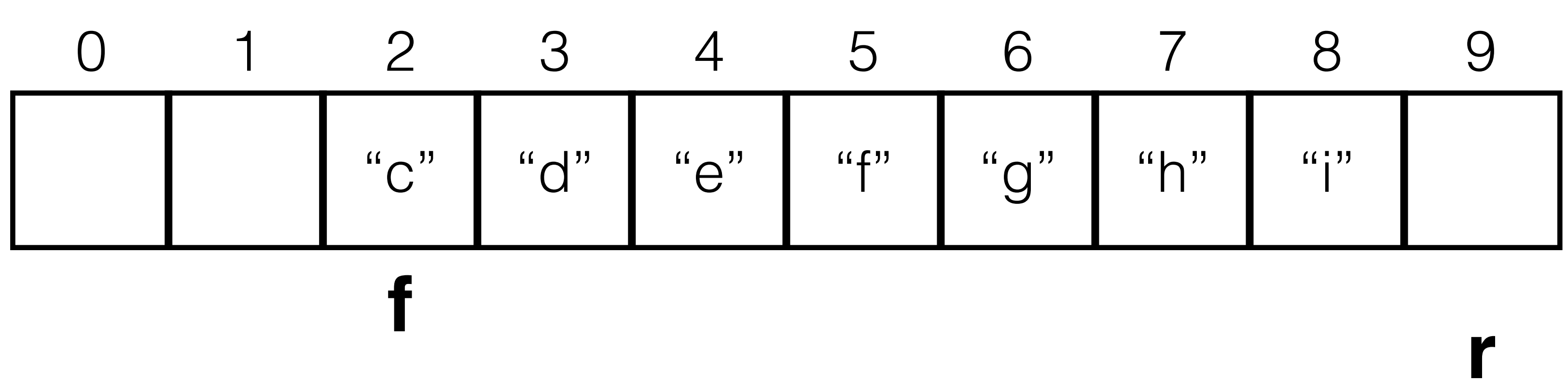
Blocking Queues

- *Now maybe some dequeue() calls are made…*



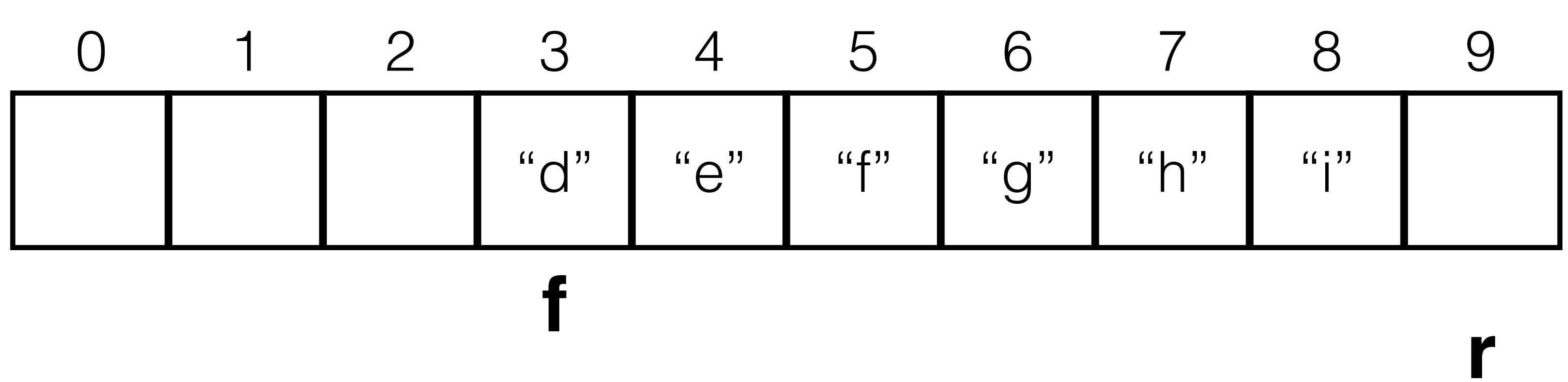
Blocking Queues

- *Now maybe some dequeue() calls are made…*



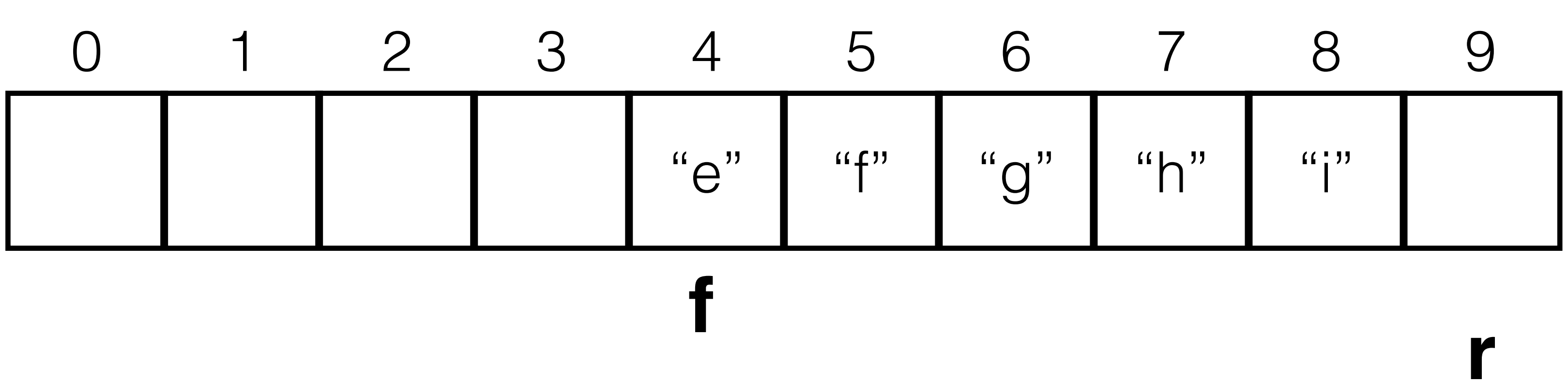
Blocking Queues

- *Now maybe some dequeue() calls are made…*



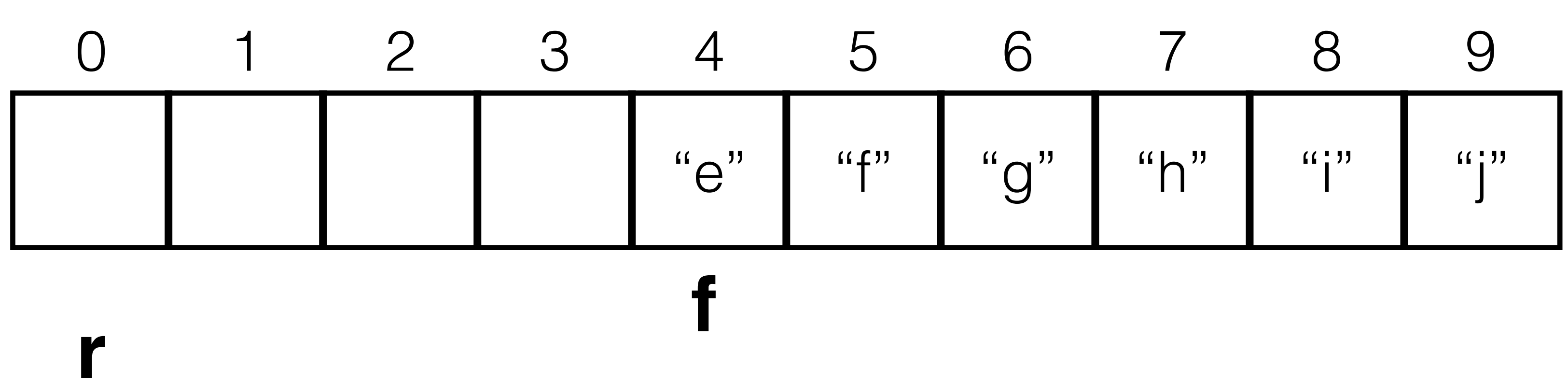
Blocking Queues

- Q: What happens when we reach the end of the array and want to enqueue()?!
  - stuck…?
  - no! There is likely free space at the front of the array…
  - wrap around!



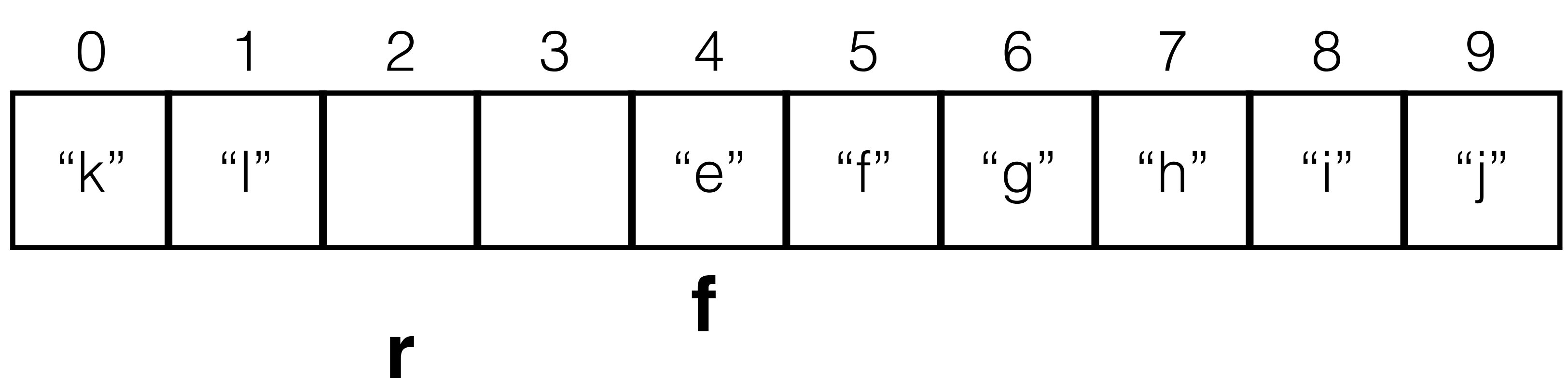
Blocking Queues

- If **N** is the size of the array, wrap by computing **r** to be: **r = (r+1) % N**
  - Same logic works for wrapping **f**.
- If ***r < f*** — this indicates that the queue has wrapped around.
- (do a couple more enqueue() ops going into next slide)



Blocking Queues

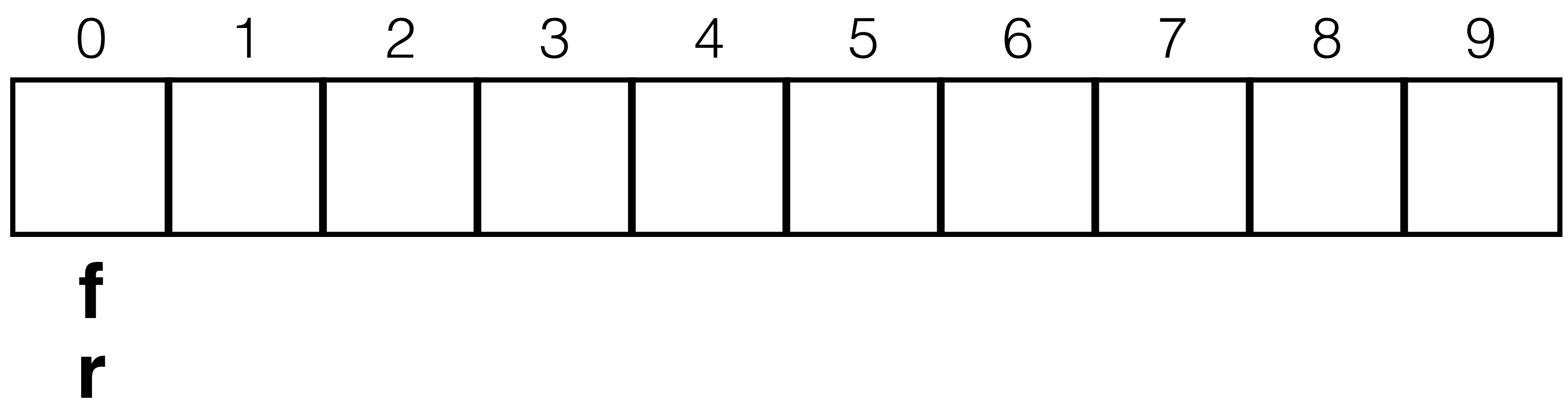
- Book shows more details about how to implement this.
- Notice that:
  - f — holds index of the first item in the queue
  - r — contains the index of the space beyond the last item in the queue (i.e., where the next new item should be added)



Blocking Queues

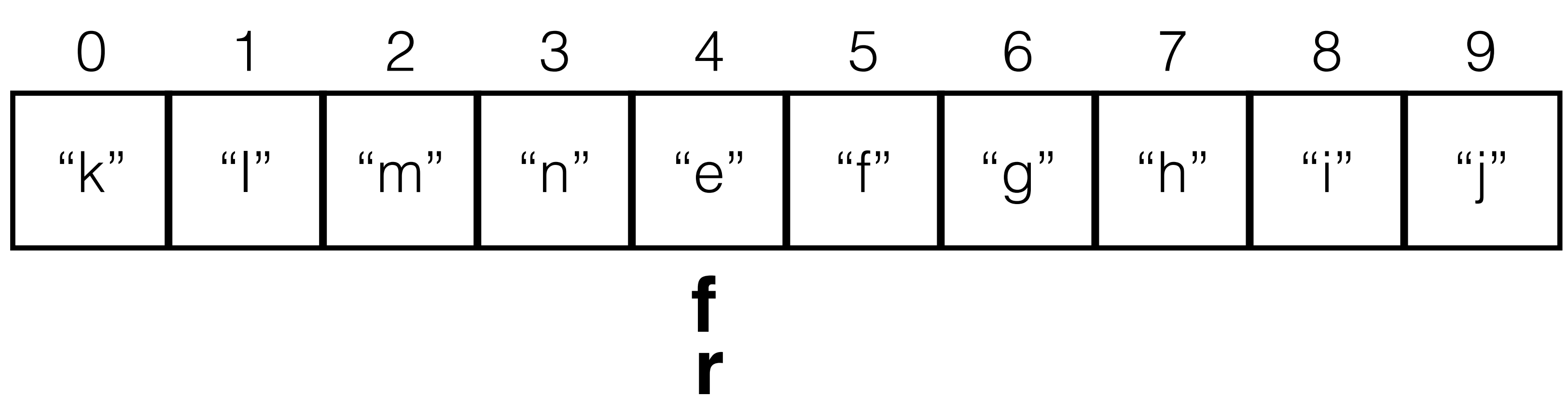
- Recall: queue is empty if: **r == f**

empty



- But this is actually the same for a full queue!
- Therefore we always need to leave one empty space *OR* keep track of the size

full



Deques

Deques

- Deque — “deck” (double-ended queue)
- Add/delete from either end
- Deque can be used as stack OR queue
- **Implementation:** DLL with head/tail pointers
    - All operations are Θ(1)
    - It is possible to use an array w/ wrapping to implement a deque (similar to how it is done w/ a queue)