Queues and Stacks

Atul Prakash

Downey: Chapter 15 and 16

Queues

- Queues occur in real life a lot.
  - Queues at checkout
  - Queues in banks
- In software systems:
  - Queue of requests at a web servers

Properties of Queues

- Queue is a FIFO data structure. First-in-first-out. Basic operations:
  - enqueue(element): add element to the end of the queue
  - dequeue() -> returns the element from the front of the queue, removing it
  - isEmpty() -> returns true if queue empty
  - Some queues provide additional operations, such as peeking at the front

Wrapping Lists to Create Queues

- Can use lists as the underlying structure.
  - Create empty queue
  - boolean Q.isEmpty(): true if queue is empty
  - Q.enqueue(T elem): add element at the end of the list
  - T Q.dequeue(): remove element from the front of the list and returns it.

Design Outline

```
List a;
```

```
isEmpty() {
   if a.size() == 0 return true;
   else return false;
}

enqueue(T element) {
      a.add(element);
}

dequeue() {
     T result = a.get(0);
     a.remove(result);
     return result;
}
```

```
Queue() {
    a = new List()
}
```

What we just did

```
List a;
```

```
isEmpty() {
   if a.size() == 0 return true;
   else return false;
}

enqueue(T element) {
        a.add(element);
}

dequeue() {
      T result = a.get(0);
      a.remove(result);
      return result;
}
```

```
Queue() {
    a = new List()
}
```

Queue.java

```
T data[];
```

```
size() {
    ....
}

add(T element)
{
        ....
}

remove() {
    ....


 T get(int i) {

 }
```

List.java

Discussion

- Lists can do everything that queues can
- So, why implement a queue class? Why not directly use lists?

Wrapping

- We created lists by using an array and providing functions such as add(), remove(), get(), size(), etc.
- We created queues by using lists as the starting point
- We are sometimes restricting functionality by doing this, but also making the data structure safer and easier to use for the intended task

Queue using ArrayList a (pseudo-code)

- enqueue(o): same as a.add(o)
- a.dequeue(): three steps using ArrayList API
  - result = a.get(0)
  - a.remove(0) or a.remove(result), depending on the interface of arraylist a
  - return result

Performance using Array-based Lists

- enqueue operation:
  - a[size] = element; size++. Performance: O(1)
  - But, if array fills up, expensive. all n elements have to be copied to a new, larger array. Cost: O(n) in copying.
  - Think about if you double every time array fills up, what is the average cost per insertion? Suppose array size is 100.
    - First 100 insertions are cheap (constant time). Next one costs 100 operations. Then, the next 100 will be cheap. Next one costs 200. Then, the next 200 will be cheap, next one costs 400, etc.
      - One expensive operation makes subsequent ones cheap. Average performance per insert is at most one element copy + one element update. It is still constant time.

deque Operation

- save A[0], shift all elements left, and decrement size
- Have to shift everything to the left.
  - Cost: O(size)

Overall Cost

- With array-based lists, adding elements at the end and removing from the front:
  - Enqueue: O(n) worst-case. O(1): average
  - Dequeue: O(n)

Mini-exercise

- Create a stack by wrapping a list. Three operations:
  - isEmpty()
  - push(T element): push an element on top of the stack
  - T pop(): remove and return the top element from the stack
- Just need to give pseudo-code of Stack.java

Discussion

- Stacks: What is the cost of push and pop() in terms of big-Oh?
- Queues: The cost of dequeue is O(n) and the cost of enqueue is O(1). Can we make them both O(1)?

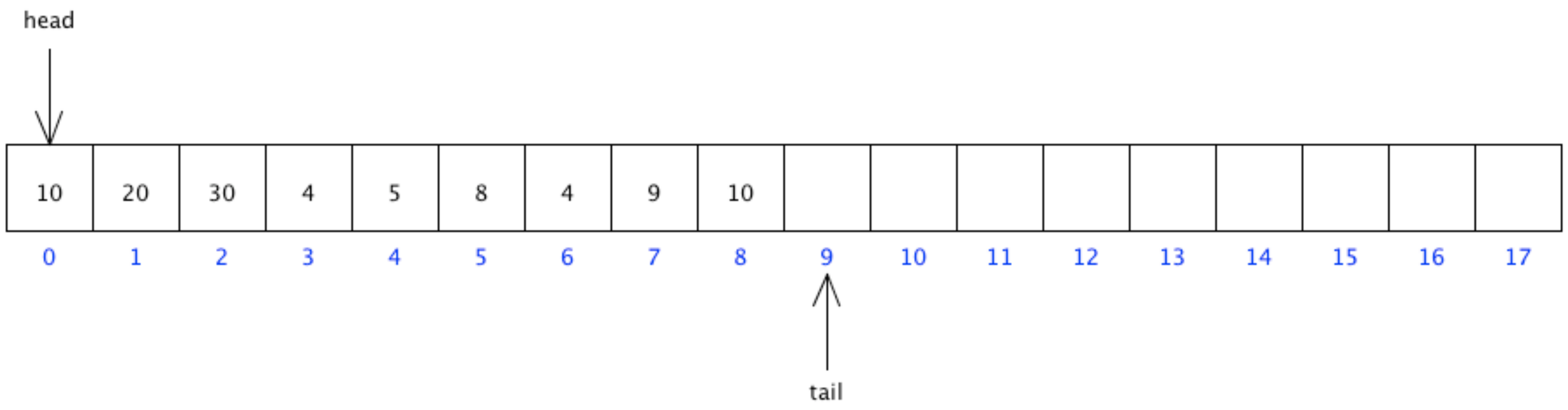
Making Queues Efficient

- Yes, we can make both operations O(1)
  - Using arrays with a data structure called circular queues
  - Using linked lists (more on that later)

Circular Queues: Basic Idea

- Right now, our implementation is as follows:
  - Arrays used to implement lists
  - Lists used to implement queues
- It turns out that we can implement queues directly on top of arrays more efficiently.

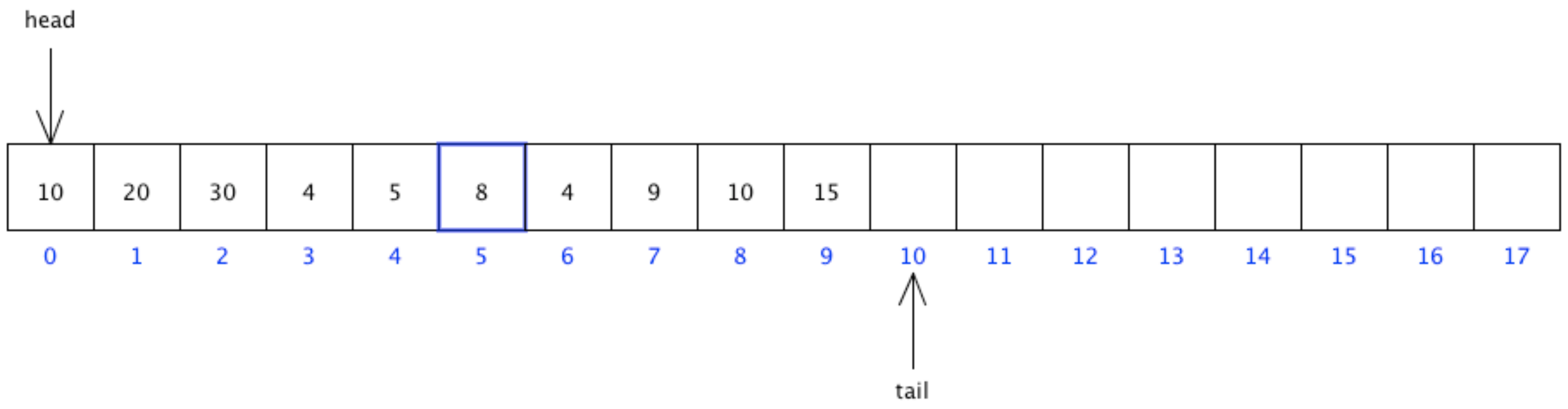
Circular queue



Keep a head pointer to the the front of the queue.
Tail pointer to the next empty slot.

Best to imagine the array to be CIRCULAR.
Position 0 follows 17.

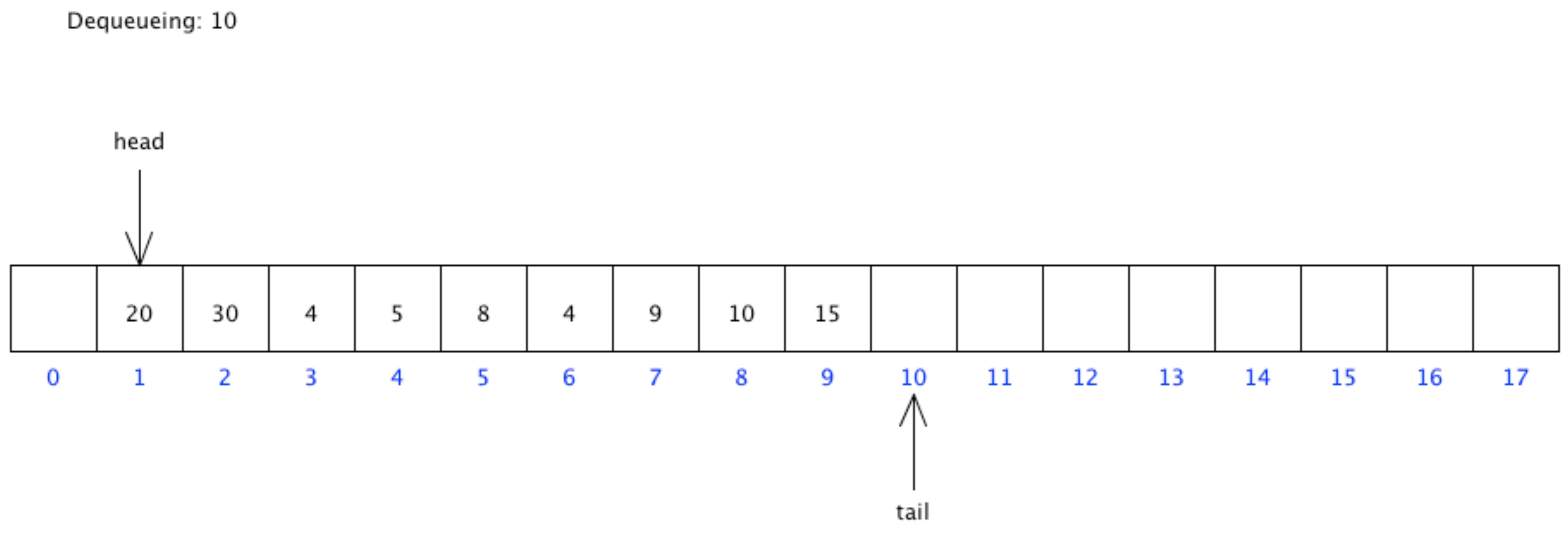
Insertion



We added 15 to the queue. tail advances. Basic code (without error checking for full queue)

```
A[tail] = elem;
tail = (tail + 1) % size.
```

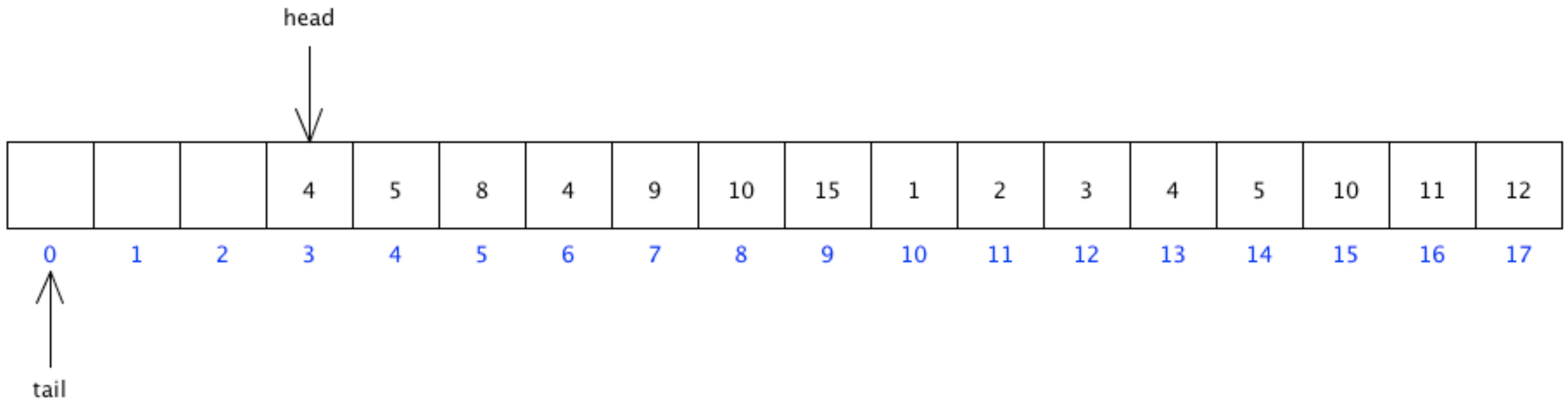
Deletion



head advances on deletion. Basic code without error checking for empty queue:

```
result = A[head];
head = (head + 1) mod size;
return result
```

Wrap around

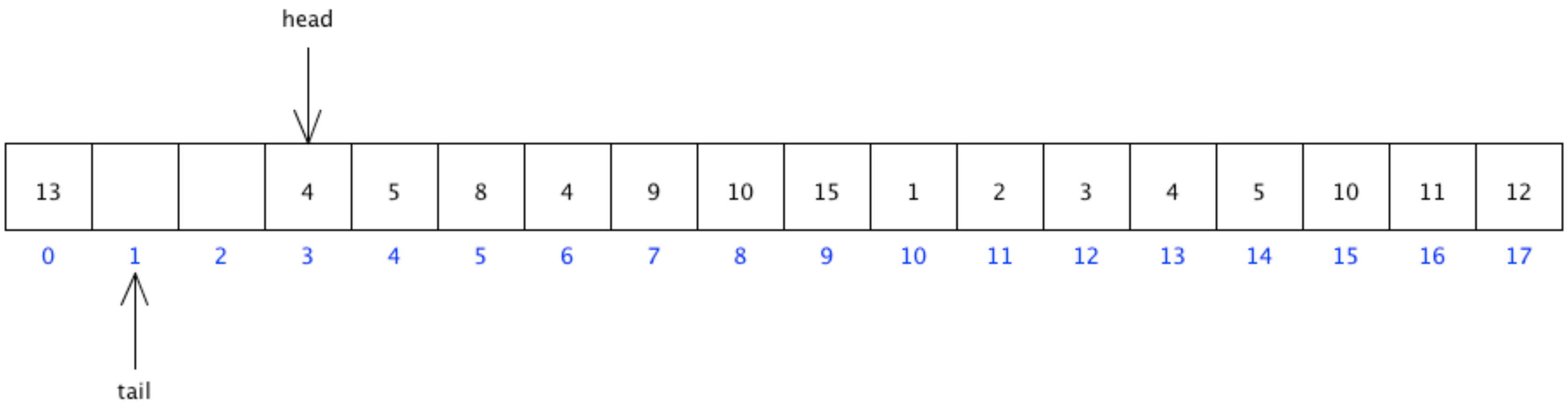


After some deletions and insertions, we can get the above situation.

Queue: 4, 5, 8, …., 12.

Arithmetic on head/tail must be mod 18

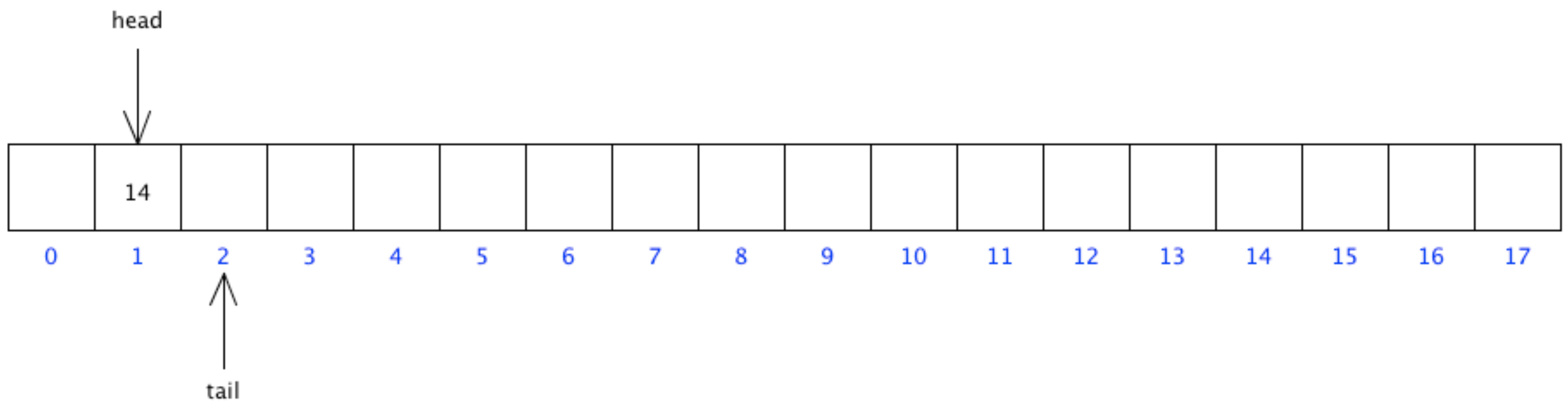
Inserts after wrap-around



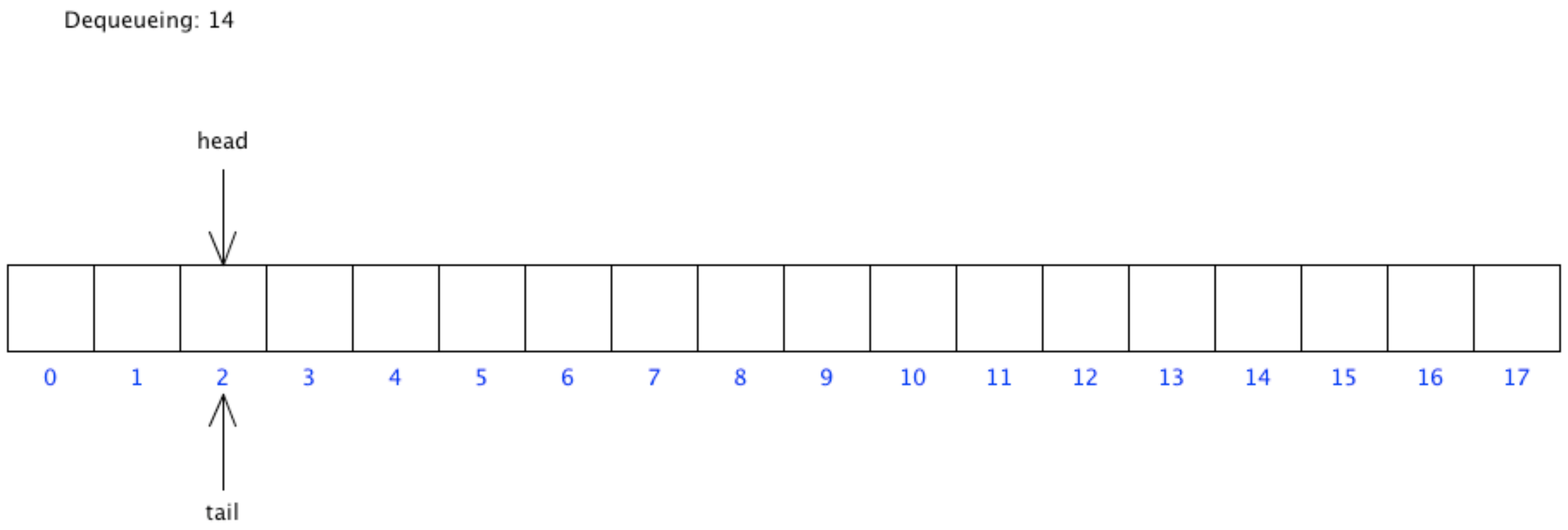
Queue: 4, 5, 8, 4… 11, 12, 13

Empty Queue

What happens when we dequeue the last element from the queue?



Empty queue situation

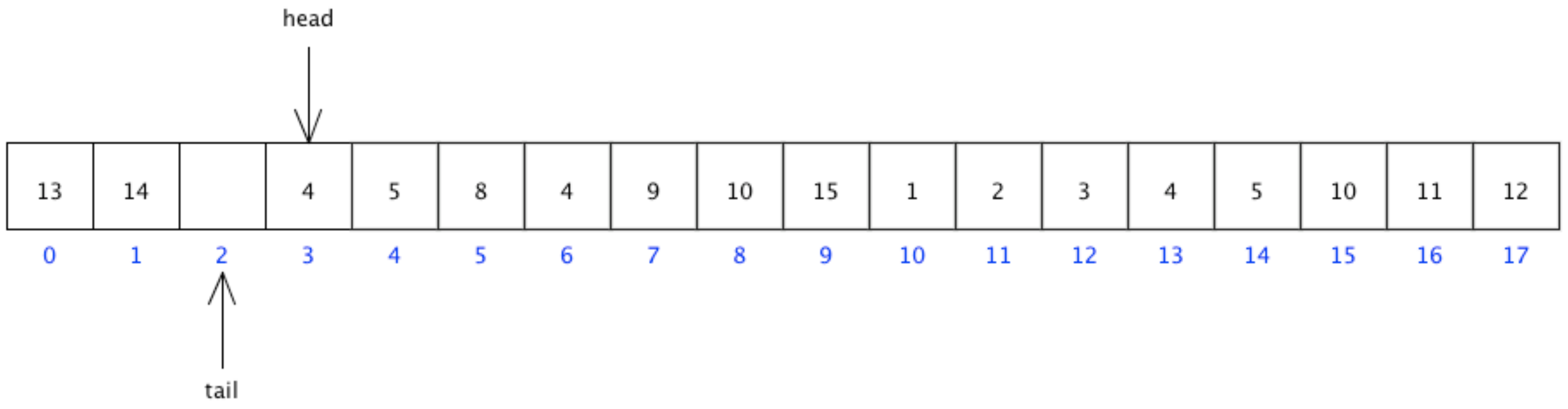


Rule: (head == tail) means the queue is empty.

Initialization of a circular queue:
head = 0; tail = 0
would be a reasonable choice.

Full Queue

- What happens on adding to the following queue?



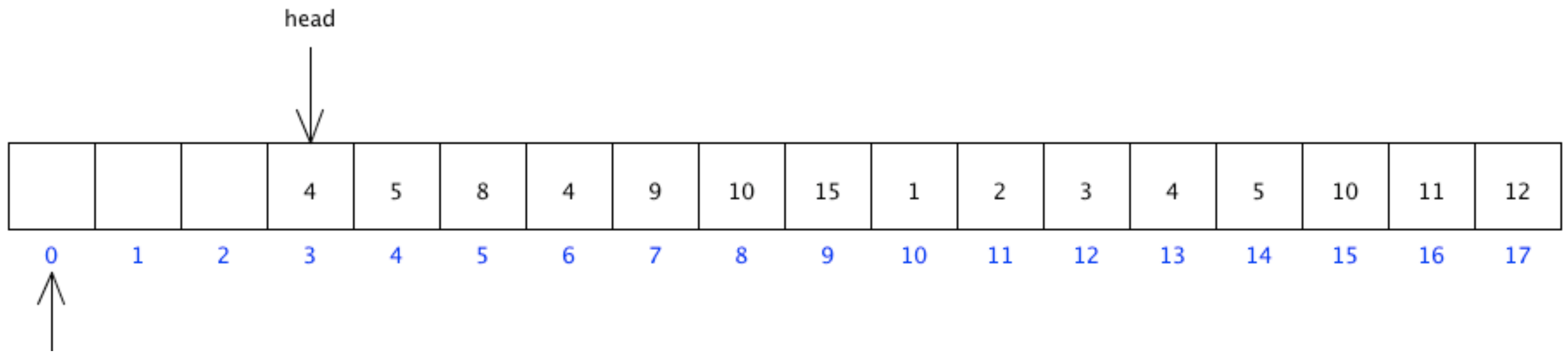
Use the last slot?

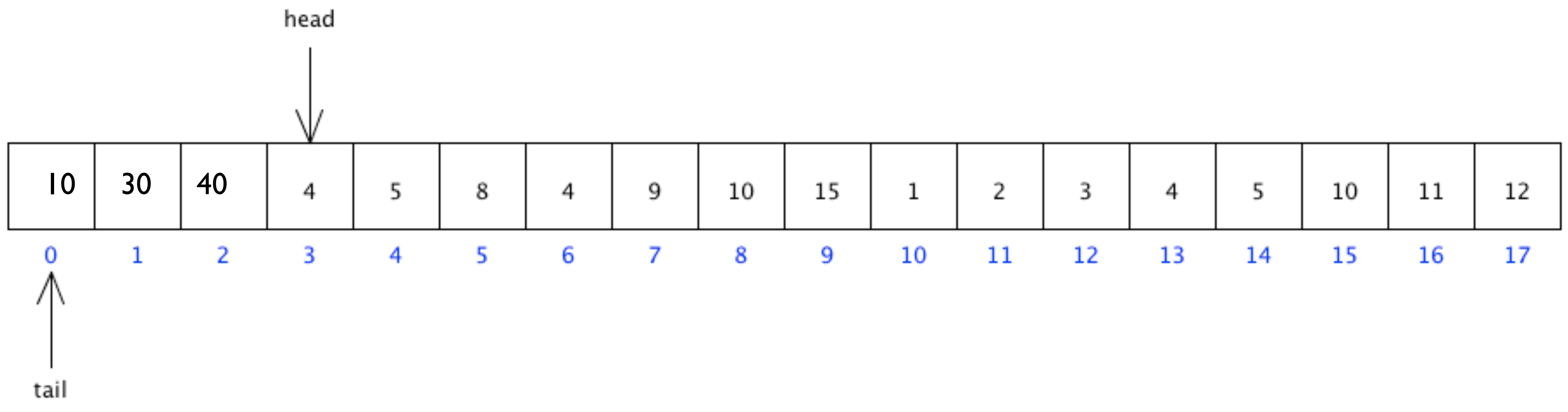
- head will become same as tail.
- How do we know whether the queue is full or empty?

Note

- If the queue contains values, we can't really use the values to determine whether the queue is empty or full.

Equivalent queues

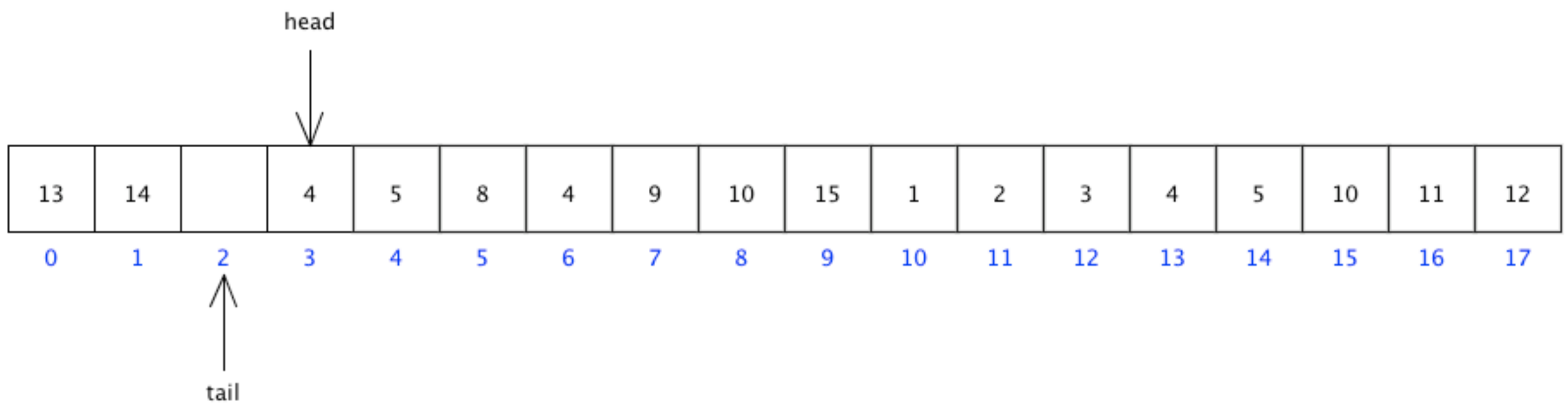




There is always some value in "empty" slots in the array.

Avoiding the trap

- Always leave one slot empty. This is a full circular queue



Rule: if (tail + 1) mod size equals head, then the queue is full.

Cost of insertion/ deletion

- Insertion cost: add at tail, advance tail. O(1). (assuming no overflow allowed).
- Deletion cost: remove from head, advance head. O(1).

Discussion

- Circular queues, as described, are of bounded size, limited by array size.
- What if we wanted unbounded-length queues?