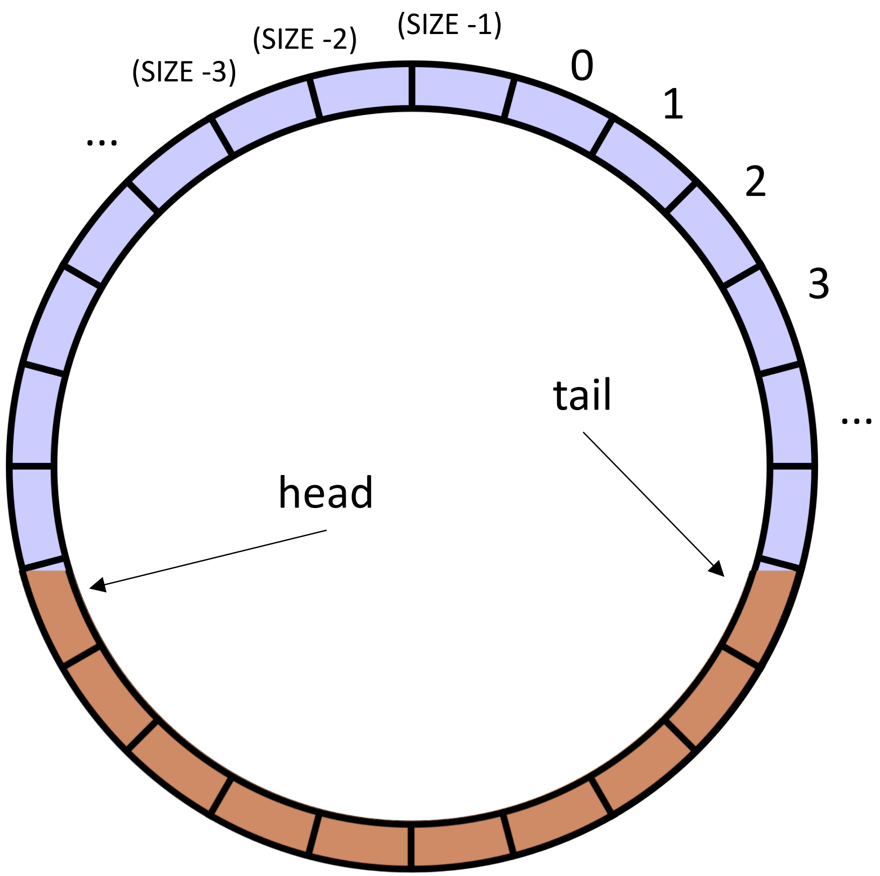
CSE113: Parallel Programming

Feb. 13, 2023

- **Topics**:
  - Input/output queues
  - Producer consumer queues
    - Synchronous
    - Circular buffer



Announcements

- HW1 grades will be out by the end of the day
  - Let us know ASAP if there are issues
- Homework 2 has a last due date today
  - We will keep an eye on Piazza and try to ask questions asked before 5 pm
- Homwork 3 will be released today by midnight
  - Due in 10 days + 4 free late days

Announcements

- Midterm out!
  - asynchronous, 1 week (no time limit)
  - Open note, open internet (to a reasonable extent: no googling exact questions or asking questions on forums)
  - do not discuss with classmates AT ALL while the test is active
  - **No late tests will be accepted.**

- **Prioritize midterm next week!**

Previous quiz

What is the relationship between linearizable (L) and sequentially consistent (SC)?

- ○ Objects can be one or the other, but not both
- ○ Objects that are L are also SC, but not the other way around
- ○ Objects that are SC are also L, but not the other way around
- ○ SC and L are the different definitions for the same concept

Previous quiz

Lock-free data structures are technically undefined because they contain data conflicts

- ○ True
- ○ False

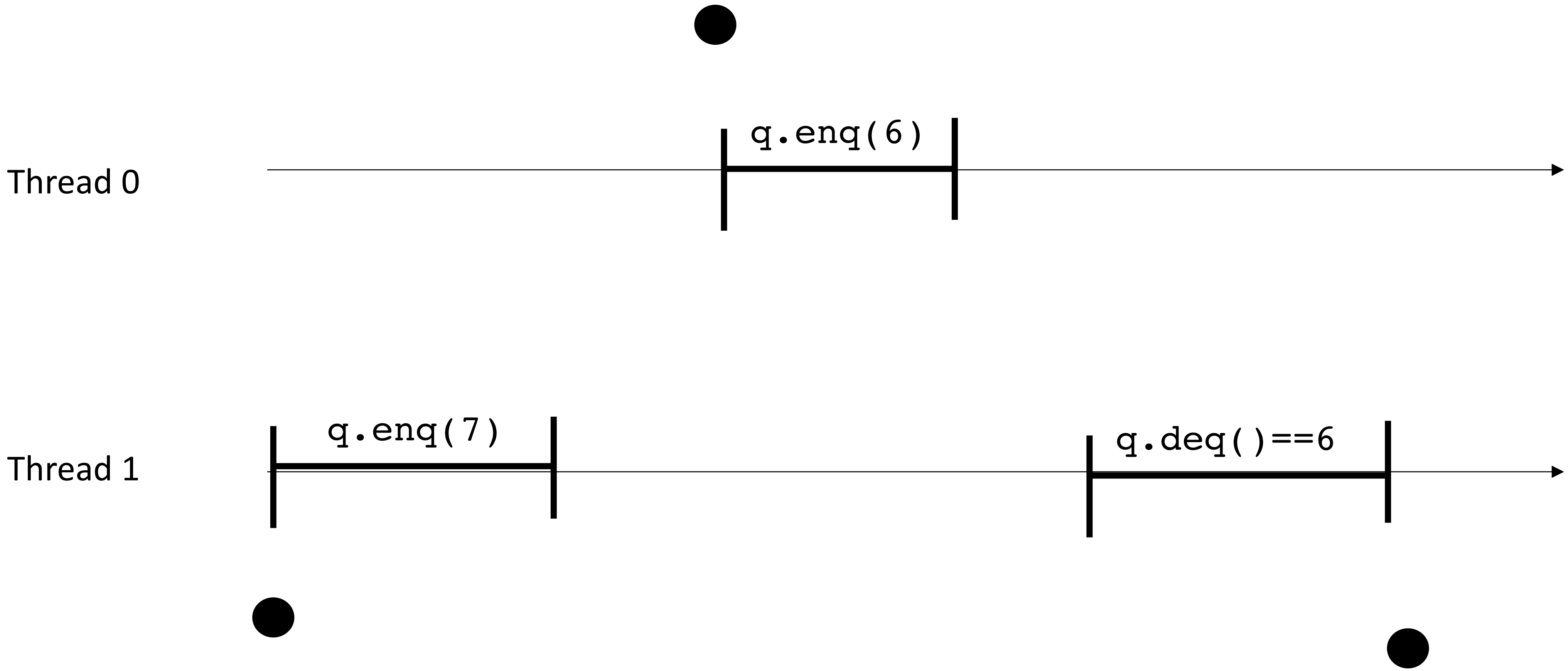
Review

Linearizability

Linearizability

each command gets a linearization point.

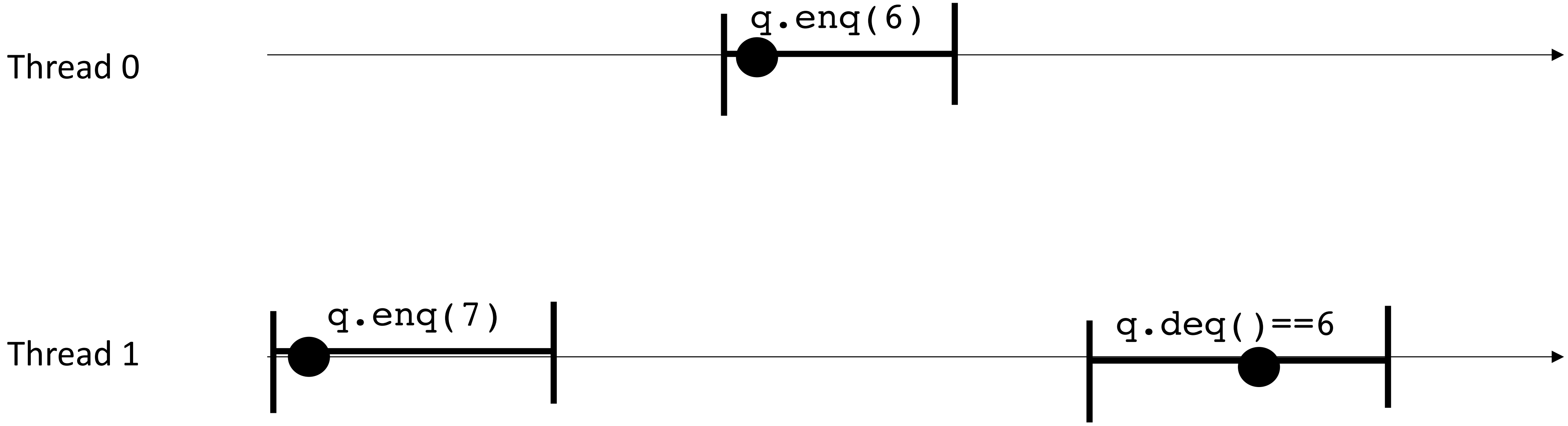
You can place the point any where between its innovation and response!



Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

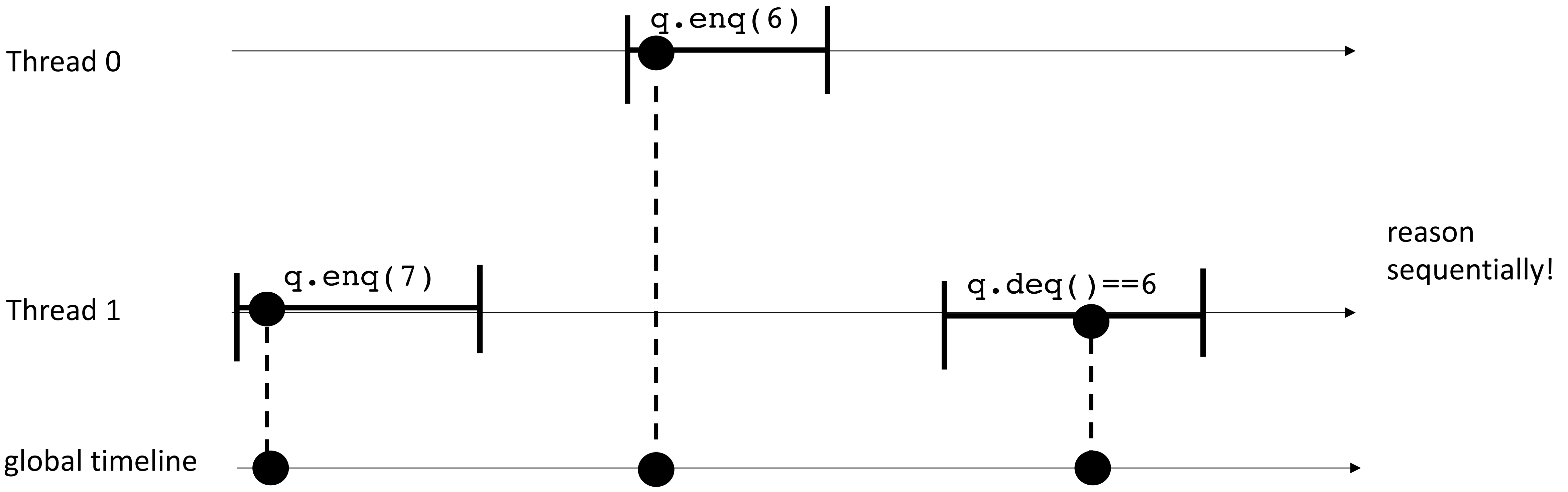


Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

Project the linearization points to a global timeline

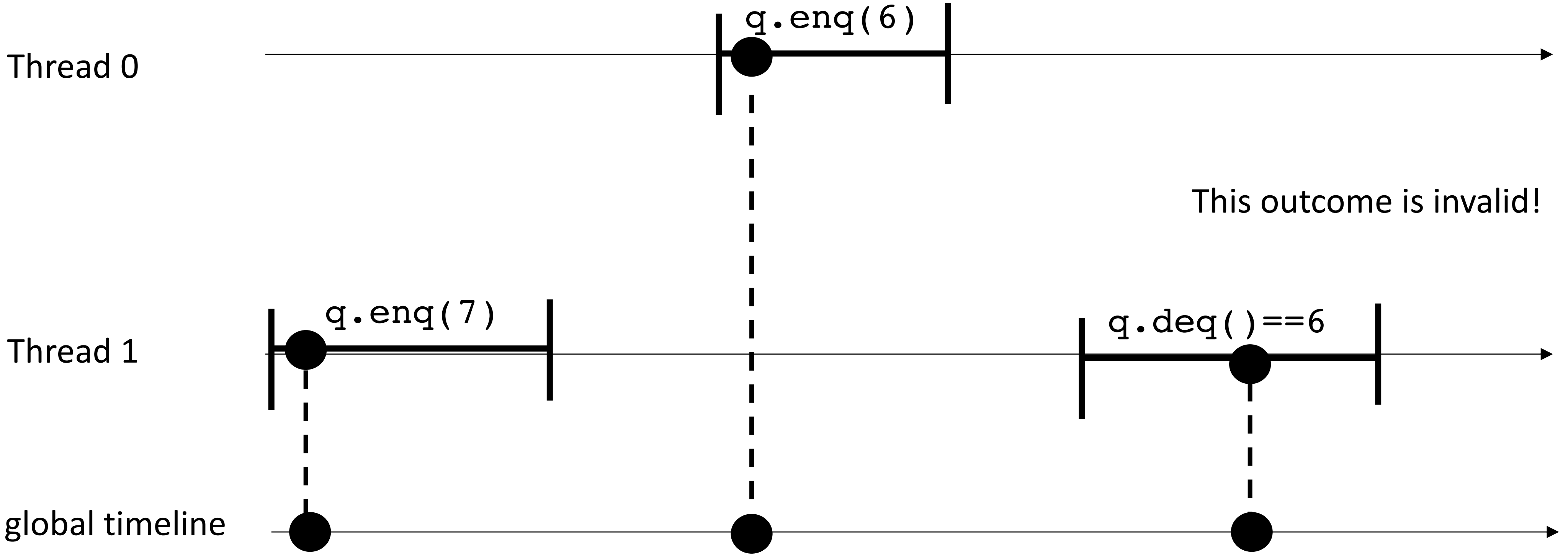


Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

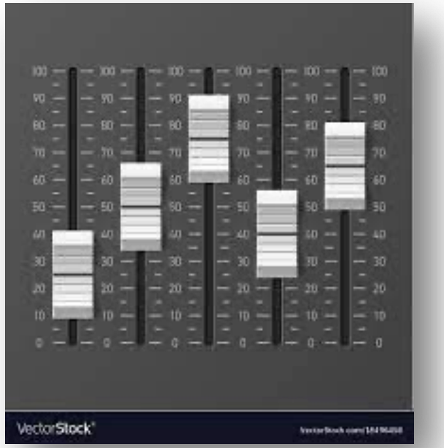
Project the linearization points to a global timeline



This outcome is invalid!

reason sequentially!

Linearizability

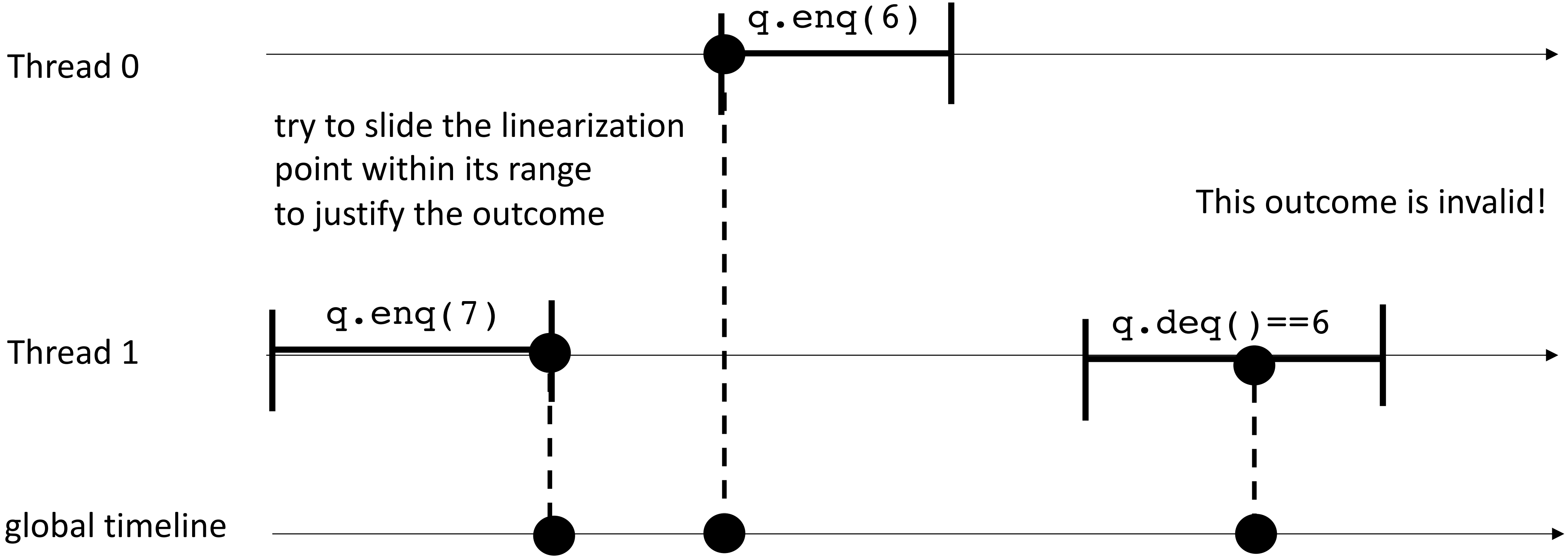


slider game!

each command gets a linearization point.

You can place the point any where between its innovation and response!

Project the linearization points to a global timeline

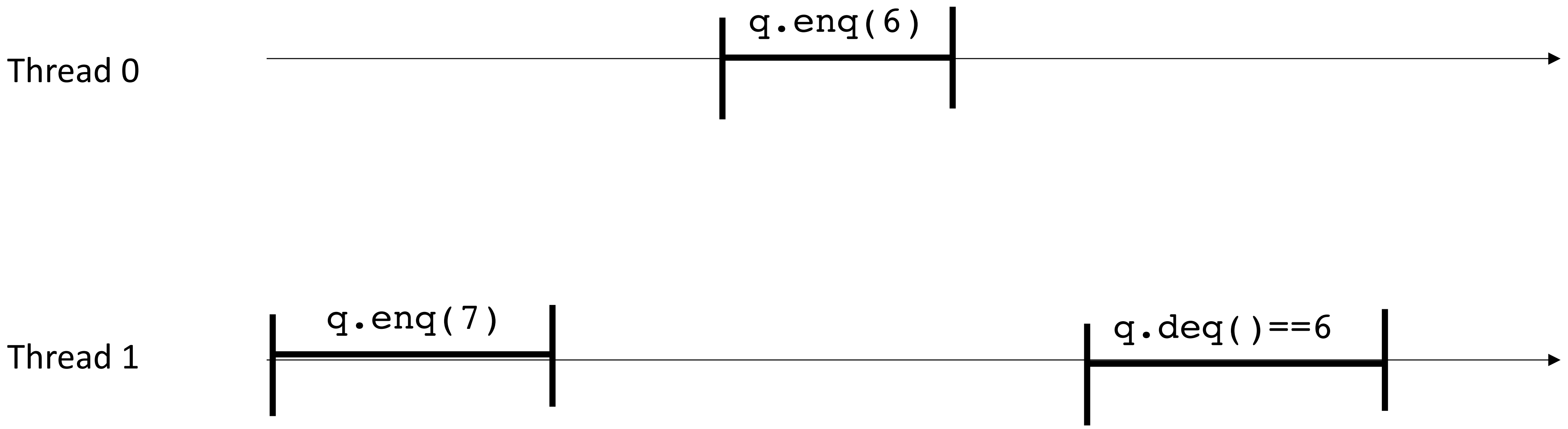


try to slide the linearization point within its range to justify the outcome

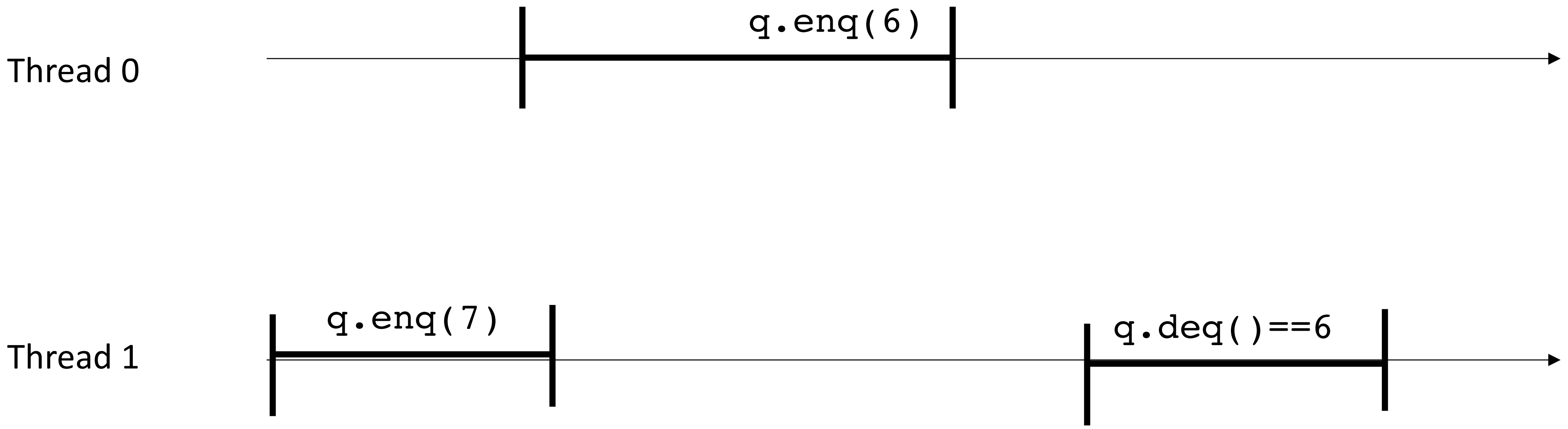
This outcome is invalid!

reason sequentially!

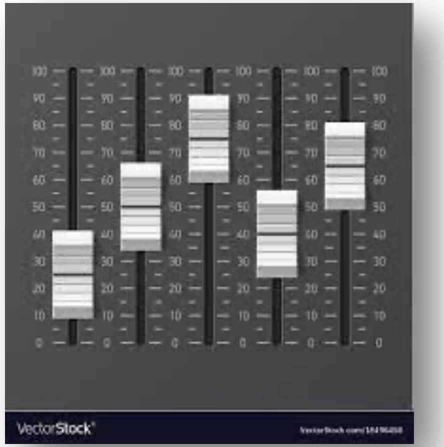
Linearizability

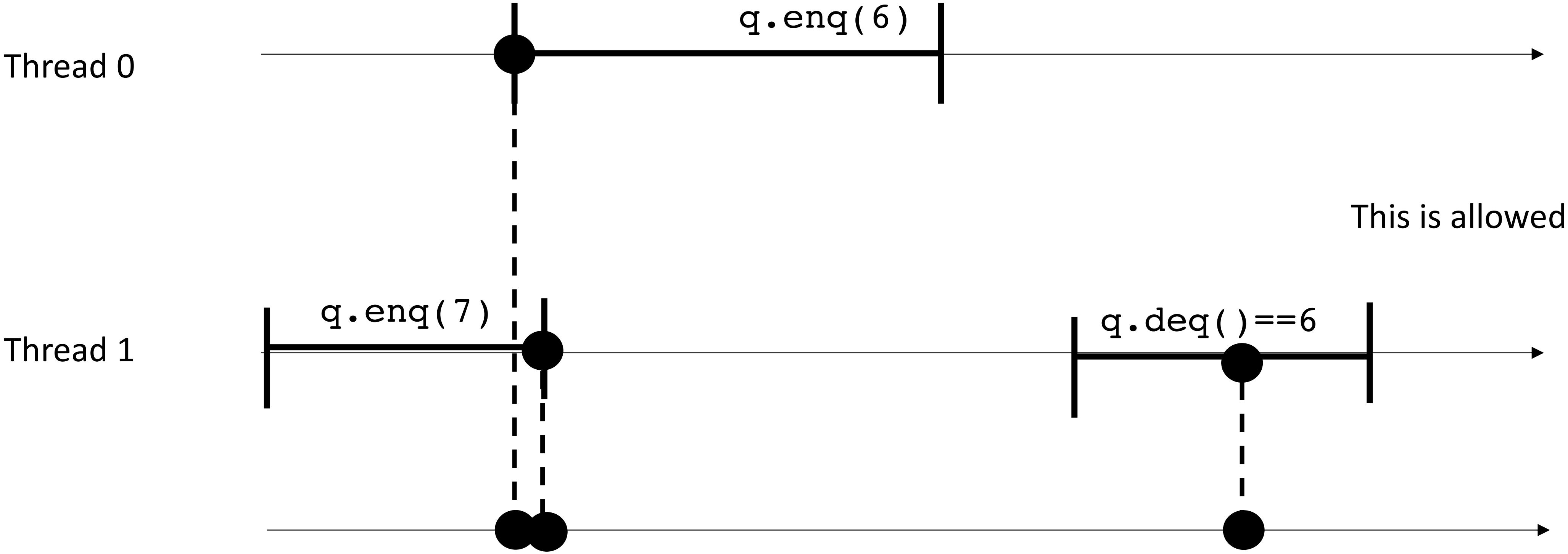


Linearizability



Linearizability

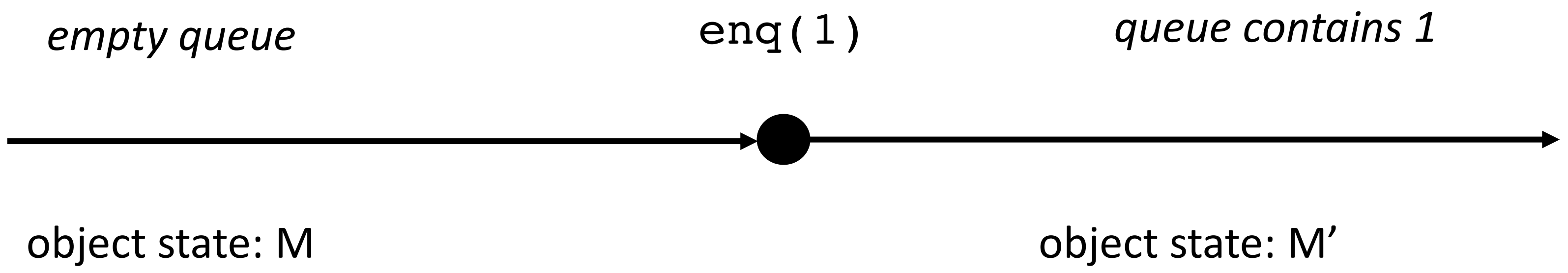




This is allowed now!

Linearizability

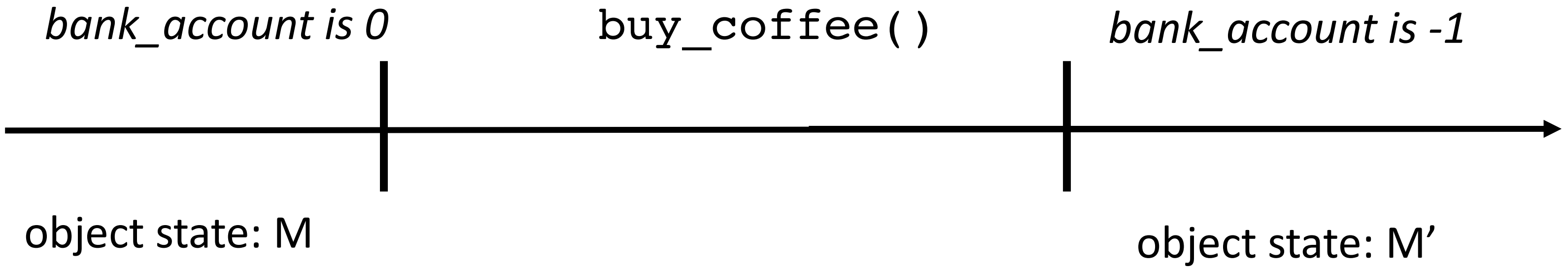
- How do we write our programs to be linearizable?
  - Identify the linearizability point
  - One indivisible region (e.g. an atomic store, atomic load, atomic RMW, or critical section) where the method call takes effect. Modeled as a point.



Linearizability

- Locked data structures are linearizable.

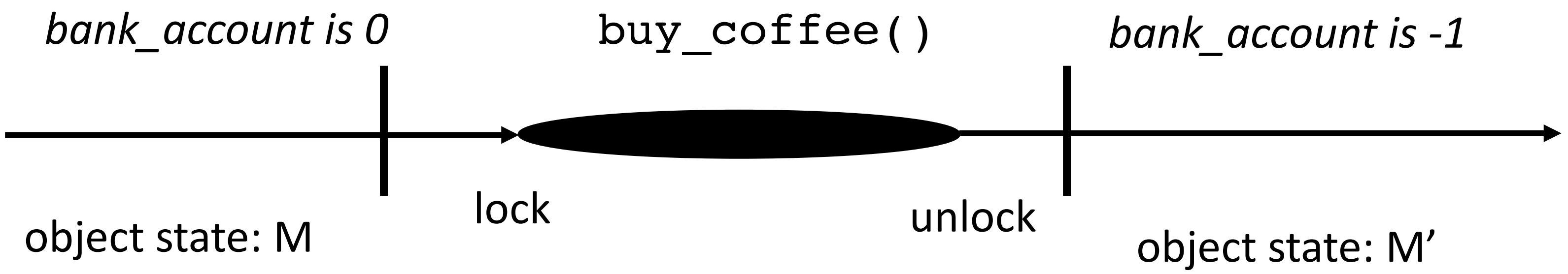
```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```



Linearizability

- Locked data structures are linearizable.

typically modeled as the point the lock is acquired or released

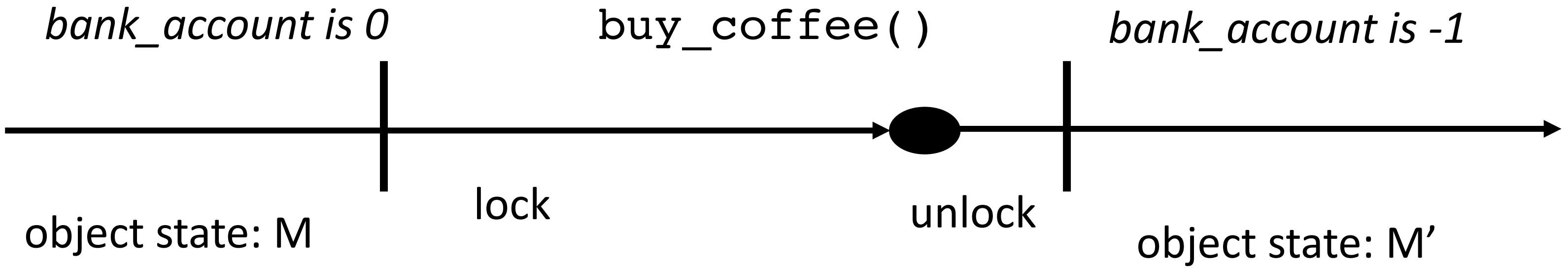



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Linearizability

- Locked data structures are linearizable.

typically modeled as the point the lock is acquired or released
lets say released.

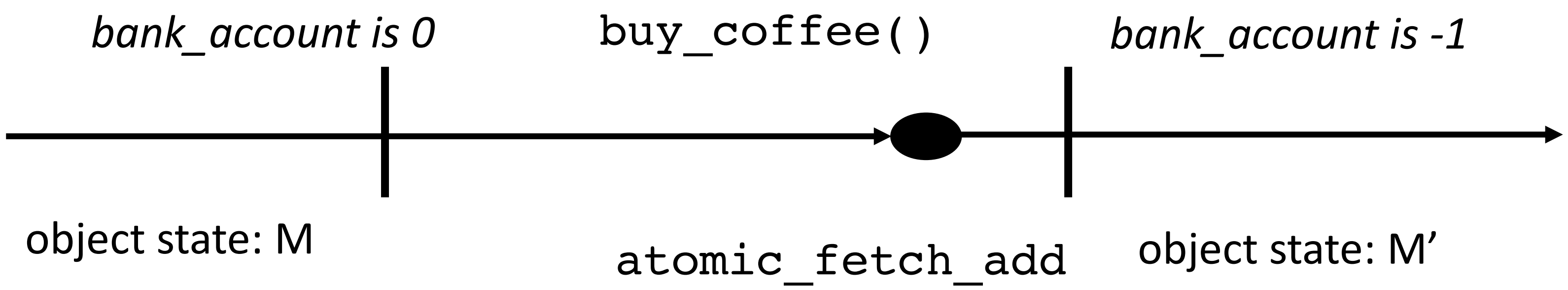



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Linearizability

- Our lock-free bank account is linearizable:
  - The atomic operation is the linearizable point

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      atomic_fetch_add(&balance, -1);
    }

    void get_paid() {
      atomic_fetch_add(&balance, 1);
    }

  private:
    atomic_int balance;
};
```



Input/Output Queues

Concurrent Queues

- List of items, accessed in a first-in first-out (FIFO) way
- *duplicates allowed*
- Methods
  - **`enq(x)`** put **`x`** in the list at the end
  - **`deq()`** remove the item at the front of the queue and return it.
  - **`size()`** returns how many items are in the queue

Concurrent Queues

- General implementation given in Chapter 10 of the book.
- Similar types of reasoning as the linked list
  - Lots of reasoning about node insertion, node deletion
  - Using atomic RMWs (CAS) in clever ways

- We will think about specialized queues
  - Implementations can be simplified!

Input/Output Queues

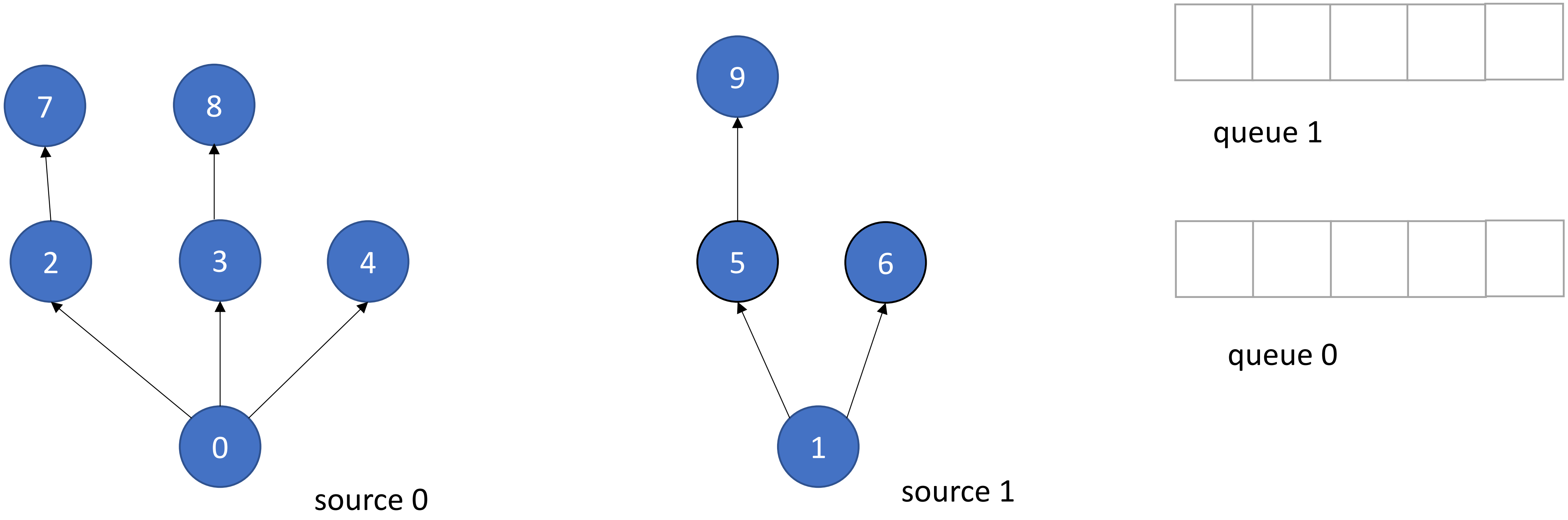
- Queue in which multiple threads read (deq), or write (enq), but not both.
- Why would we want a thing?
- Computation done in phases:
  - First phase prepares the queue (by writing into it)
  - All threads join
  - Second phase reads values from the queue.

Input/Output Queues

- Example: Information flow in graph applications:

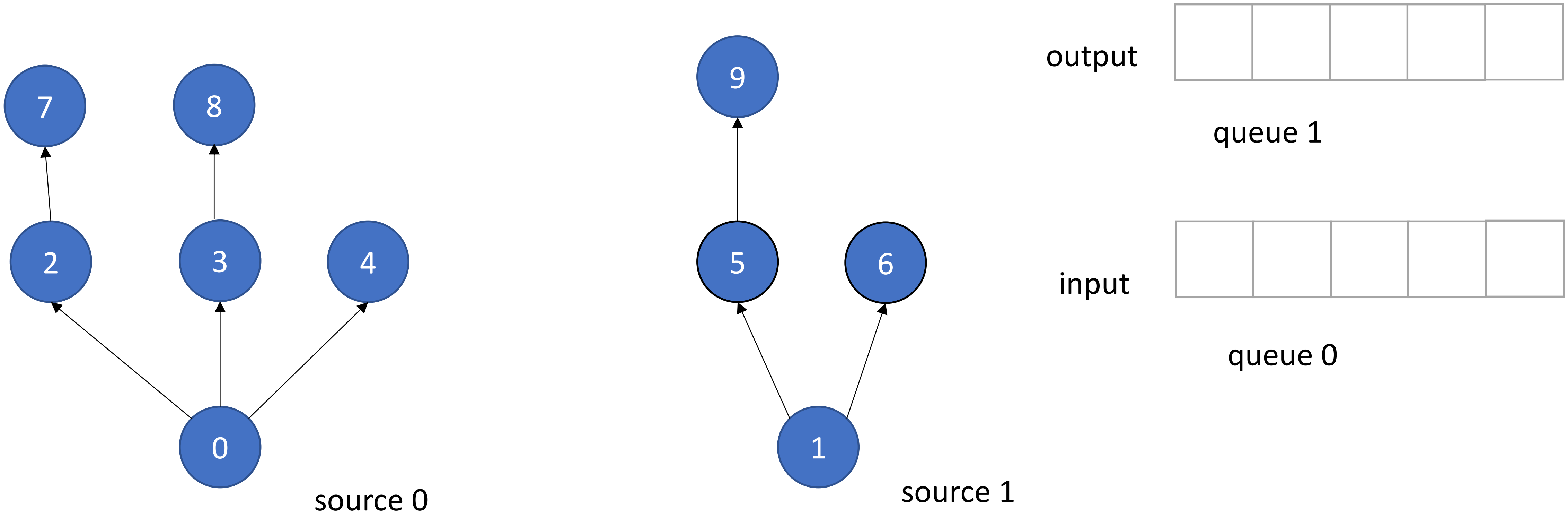
Input/Output Queues

- Example: Information flow in graph applications:



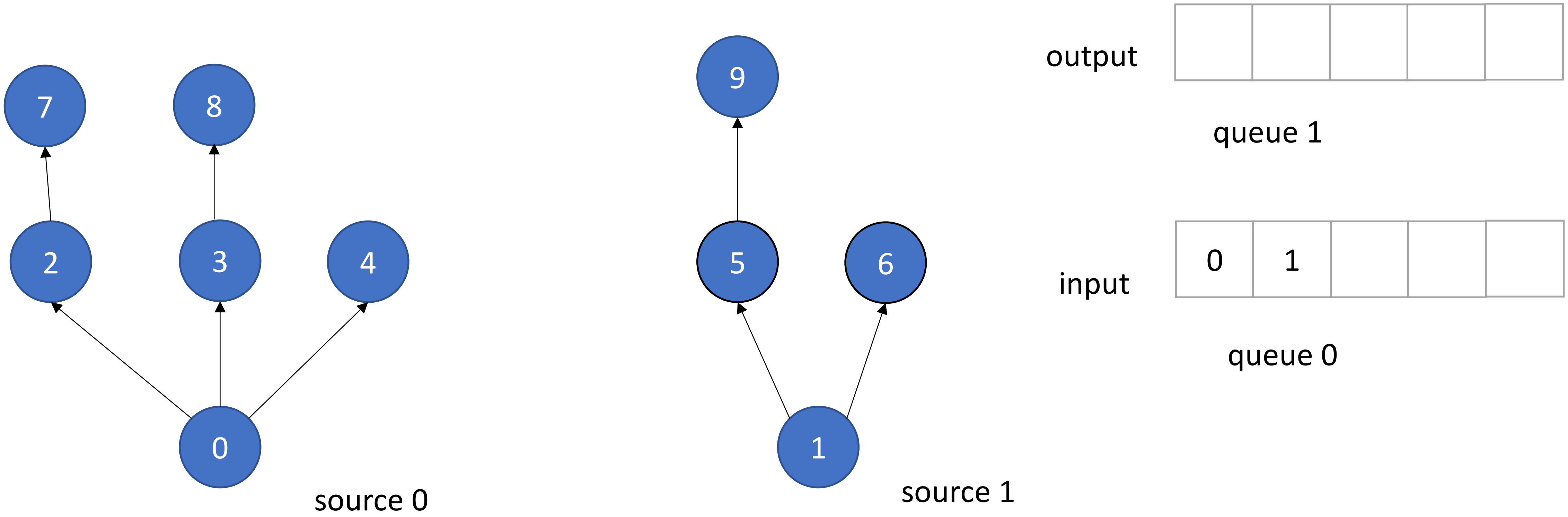
Input/Output Queues

- Example: Information flow in graph applications:



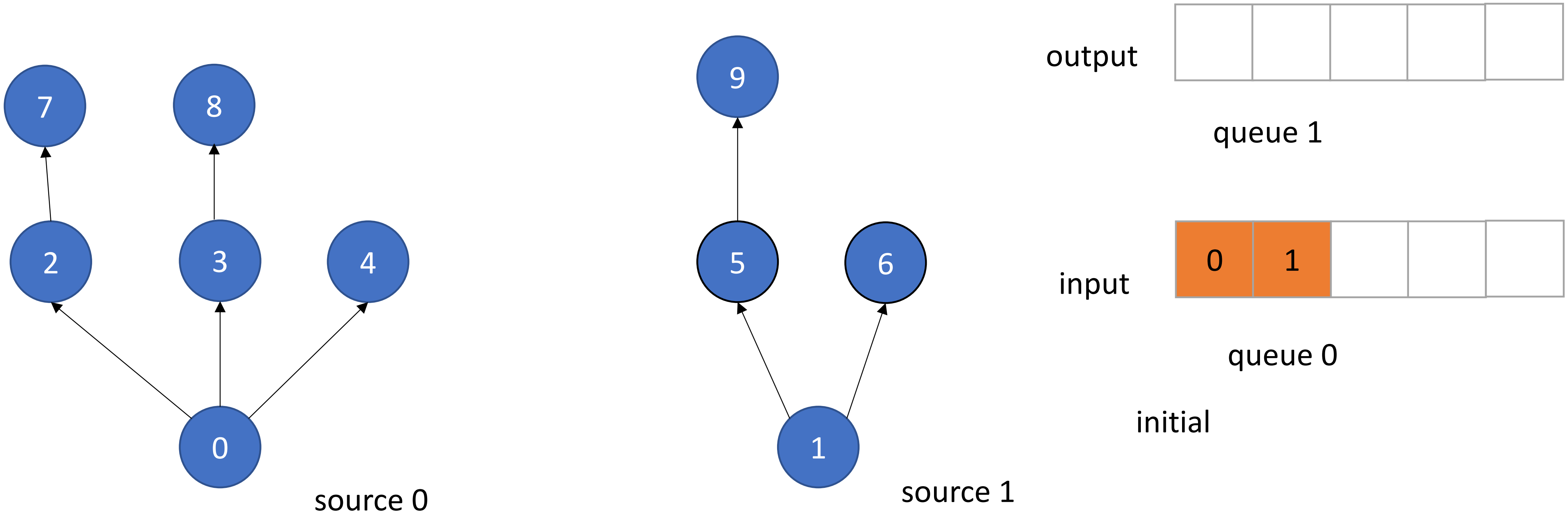
Input/Output Queues

- Example: Information flow in graph applications:



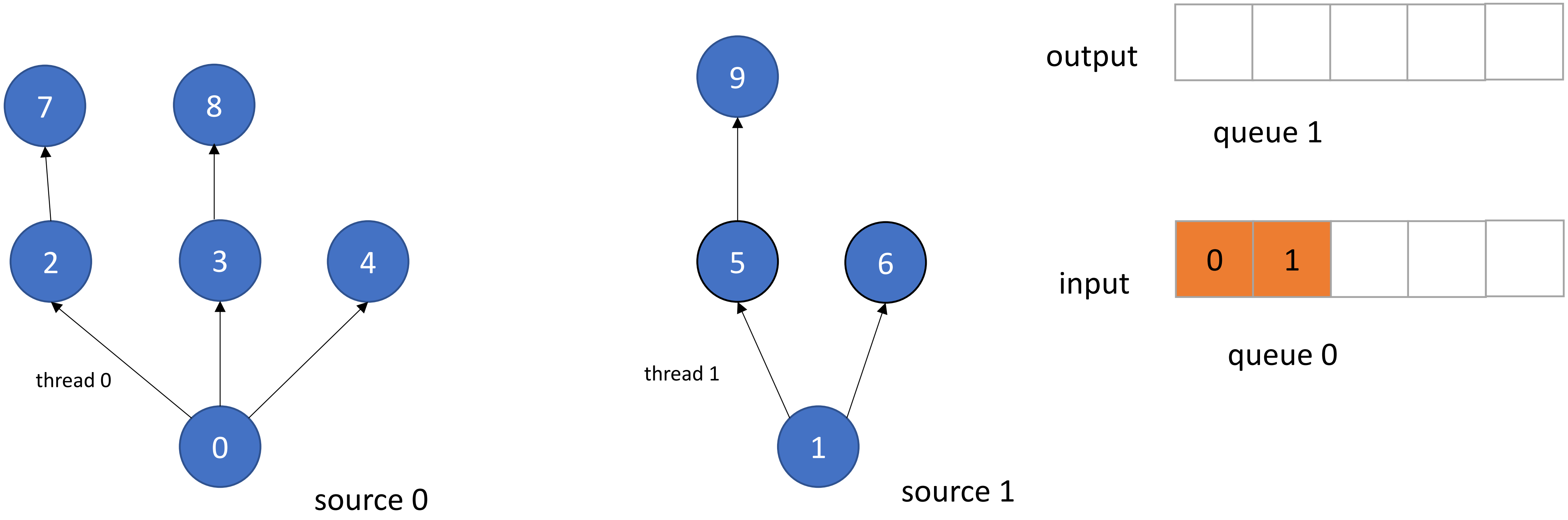
Input/Output Queues

- Example: Information flow in graph applications:



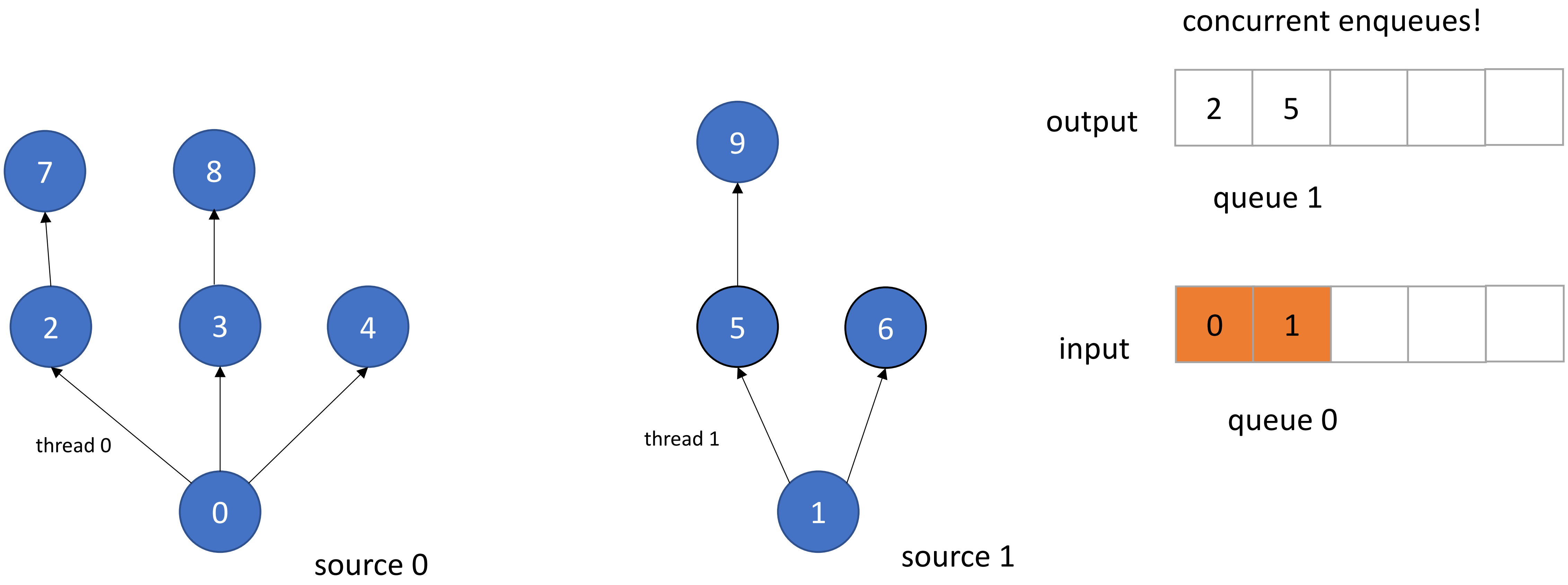
Input/Output Queues

- Example: Information flow in graph applications:



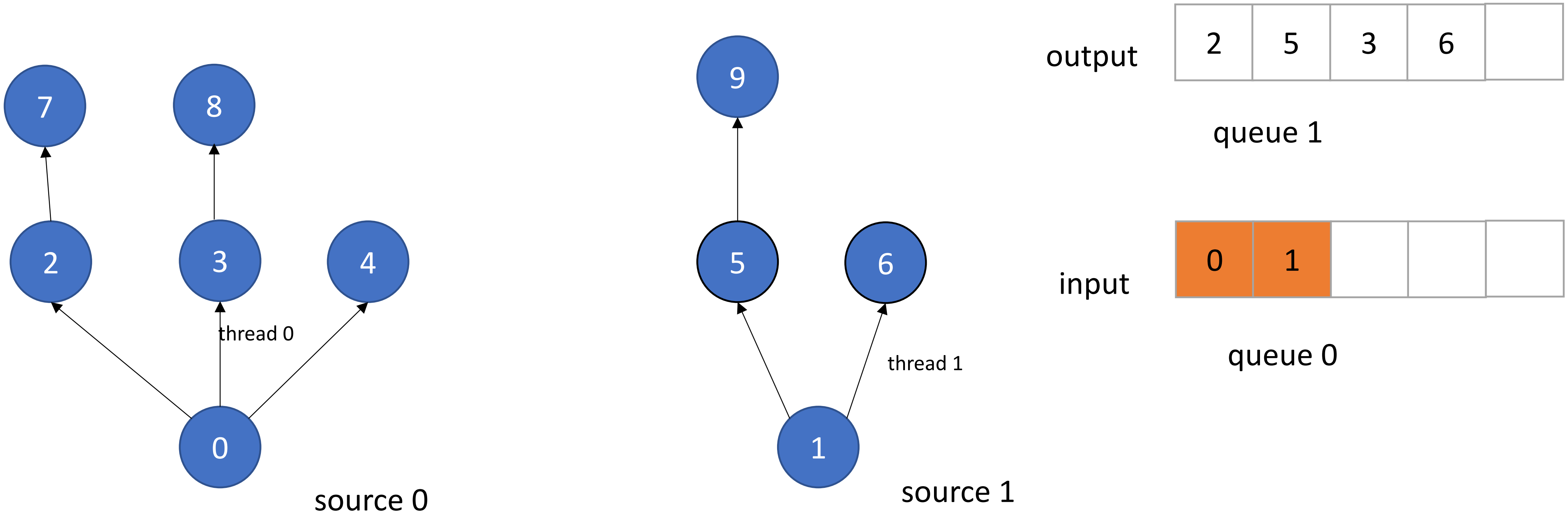
Input/Output Queues

- Example: Information flow in graph applications:



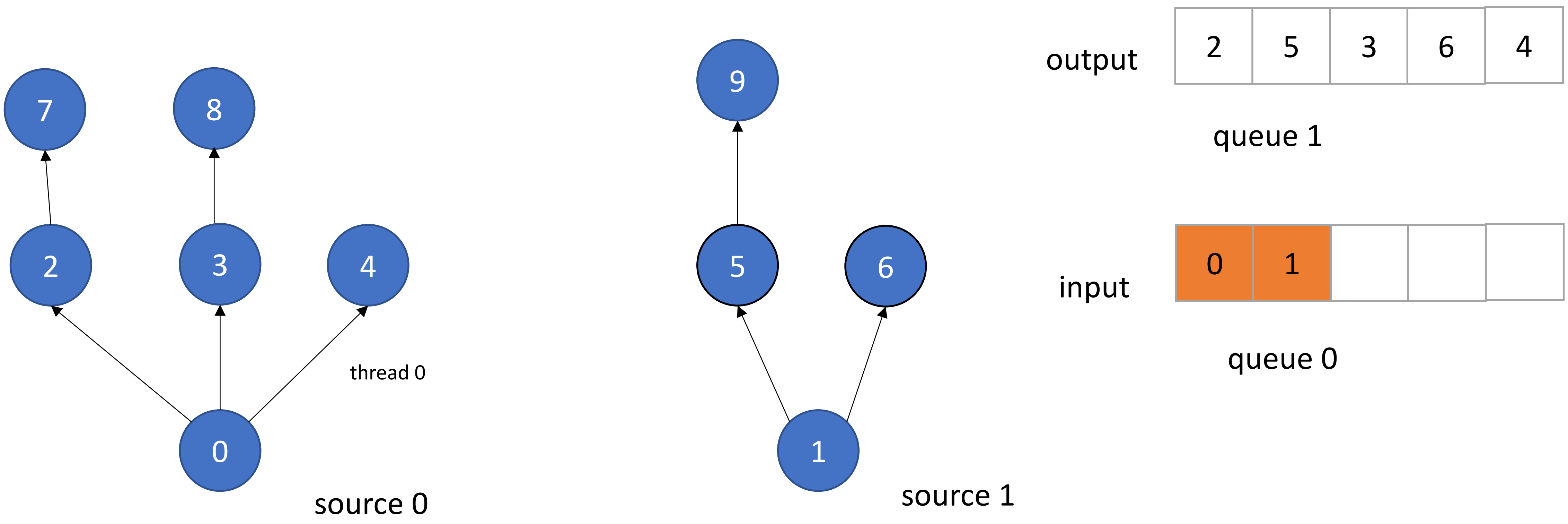
Input/Output Queues

- Example: Information flow in graph applications:



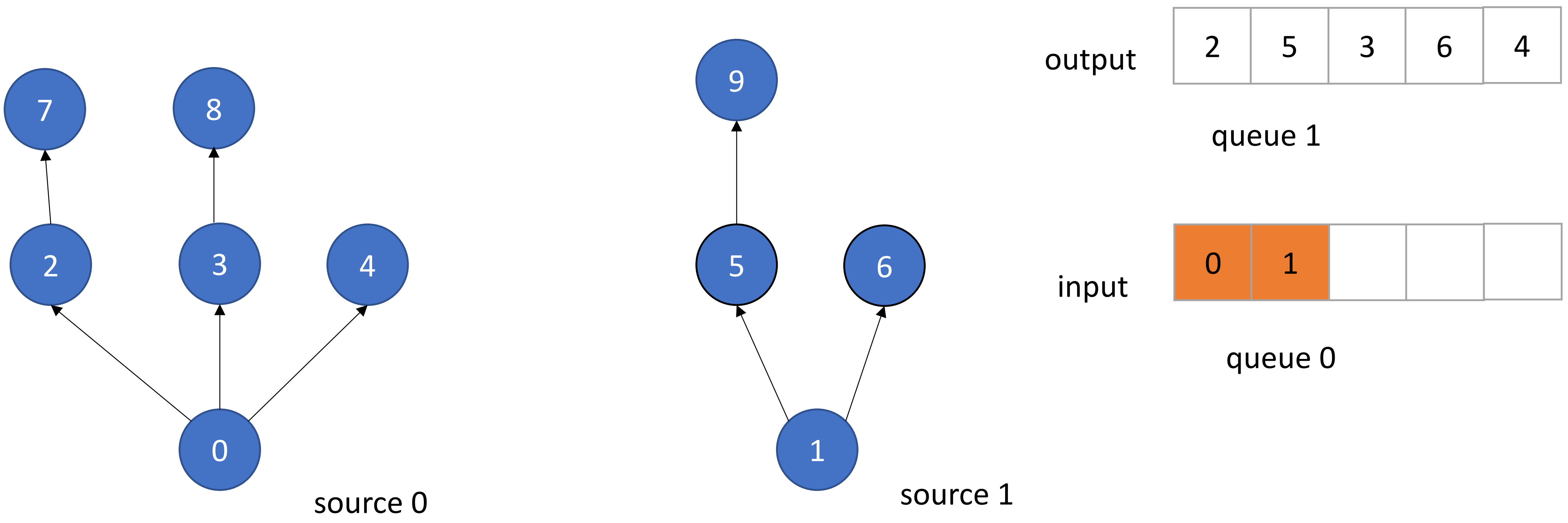
Input/Output Queues

- Example: Information flow in graph applications:



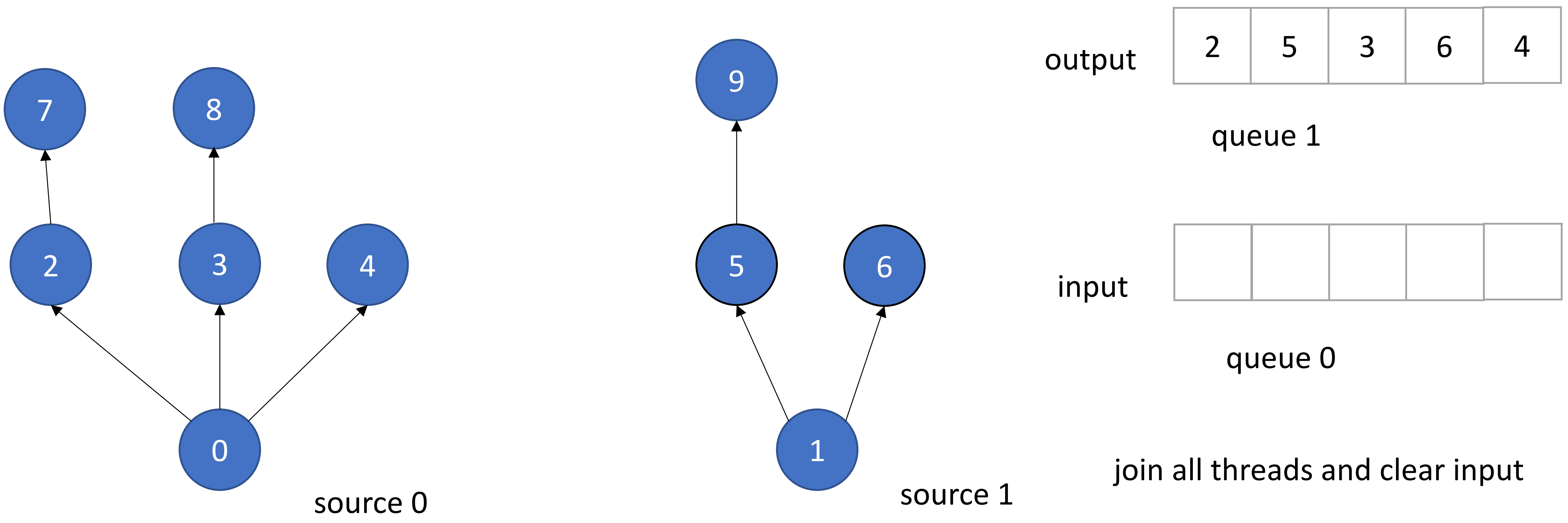
Input/Output Queues

- Example: Information flow in graph applications:



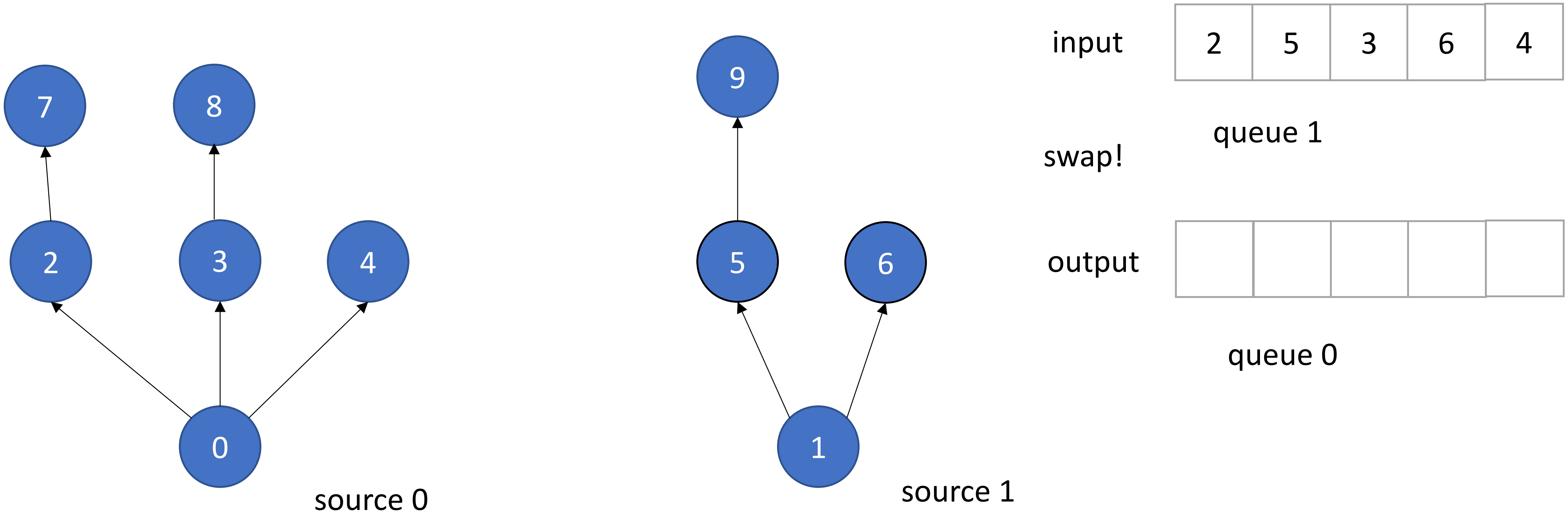
Input/Output Queues

- Example: Information flow in graph applications:



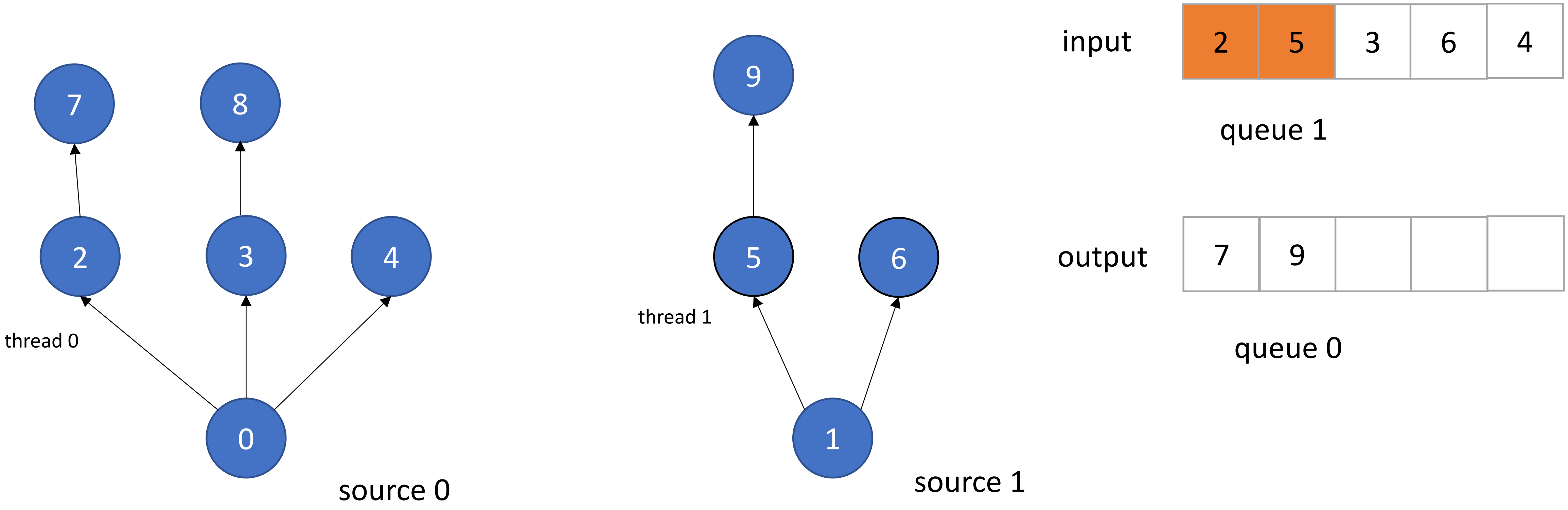
Input/Output Queues

- Example: Information flow in graph applications:



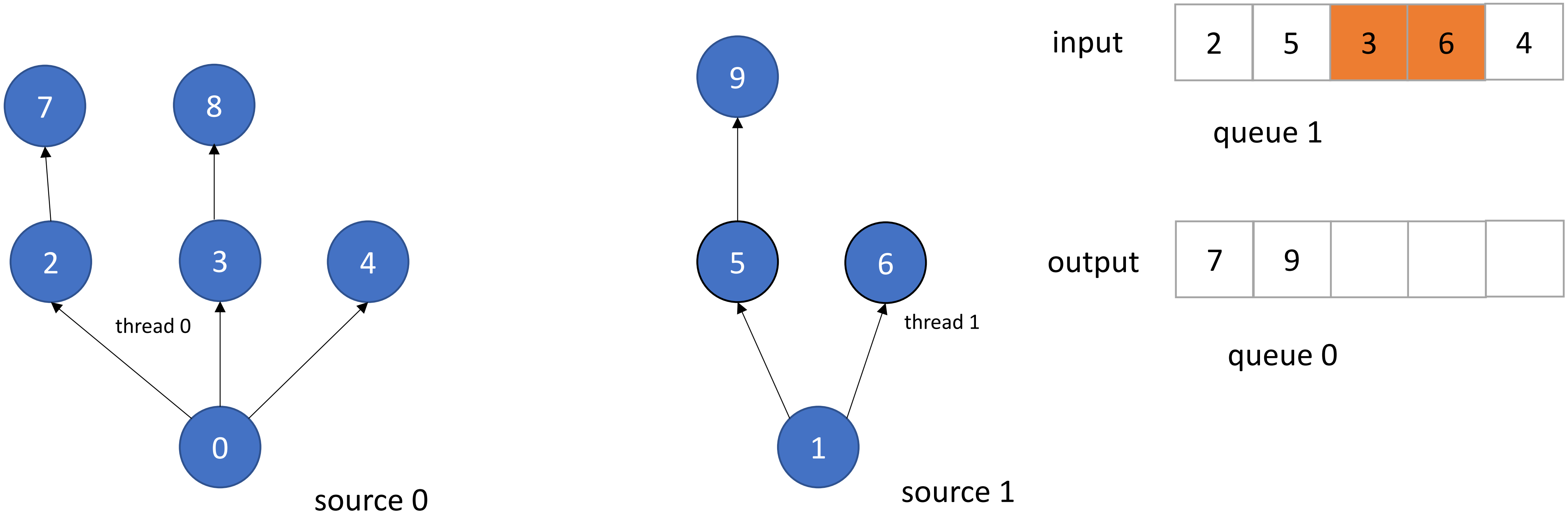
Input/Output Queues

- Example: Information flow in graph applications:



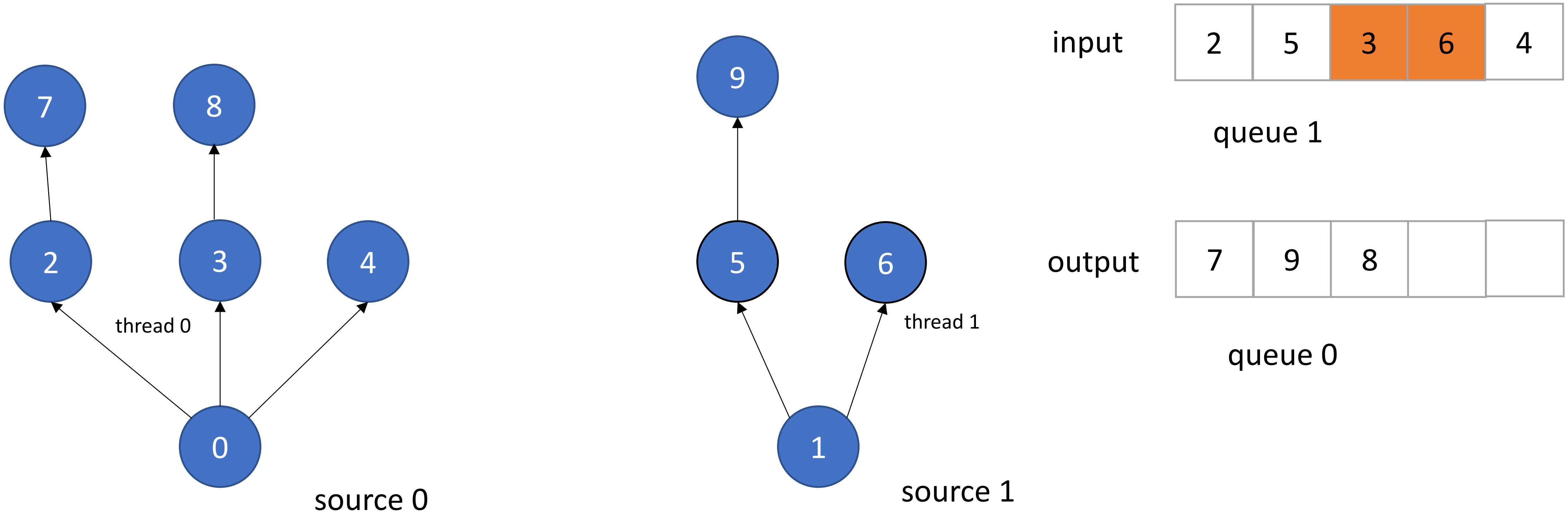
Input/Output Queues

- Example: Information flow in graph applications:



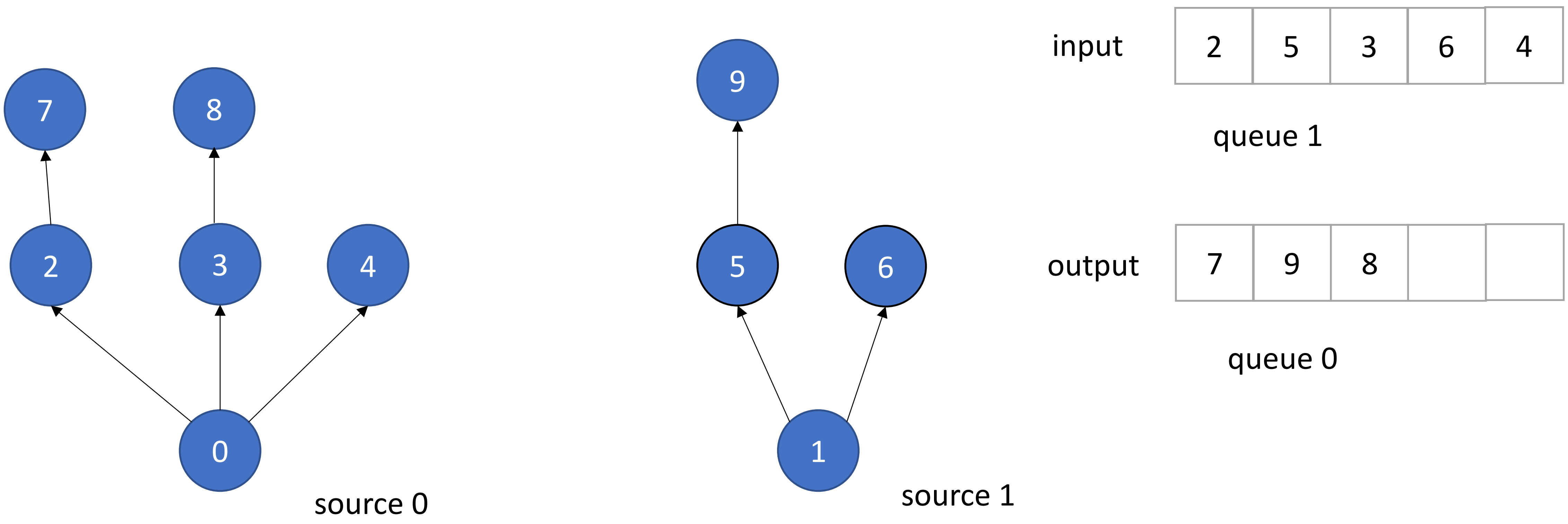
Input/Output Queues

- Example: Information flow in graph applications:



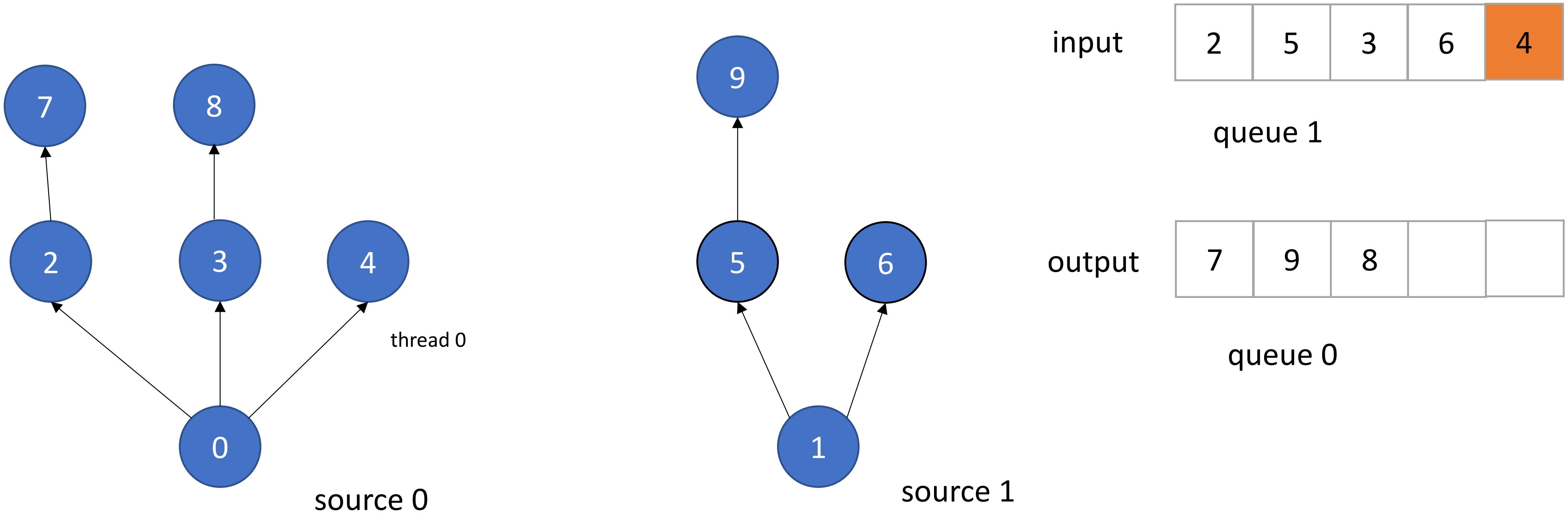
Input/Output Queues

- Example: Information flow in graph applications:



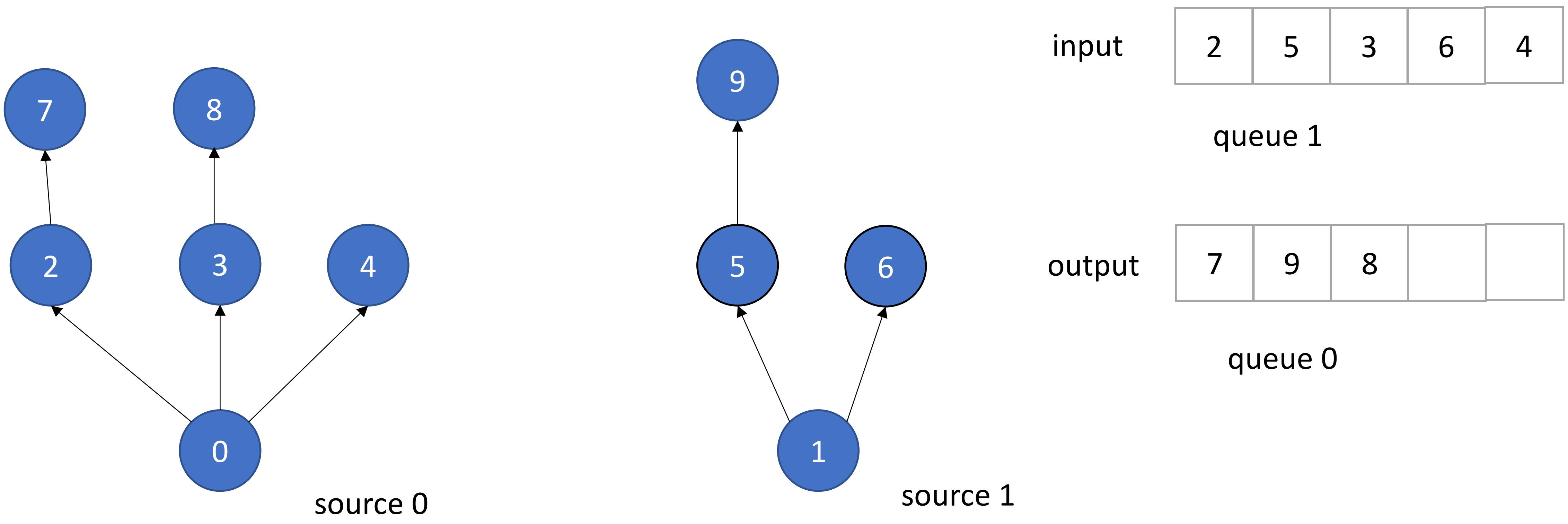
Input/Output Queues

- Example: Information flow in graph applications:



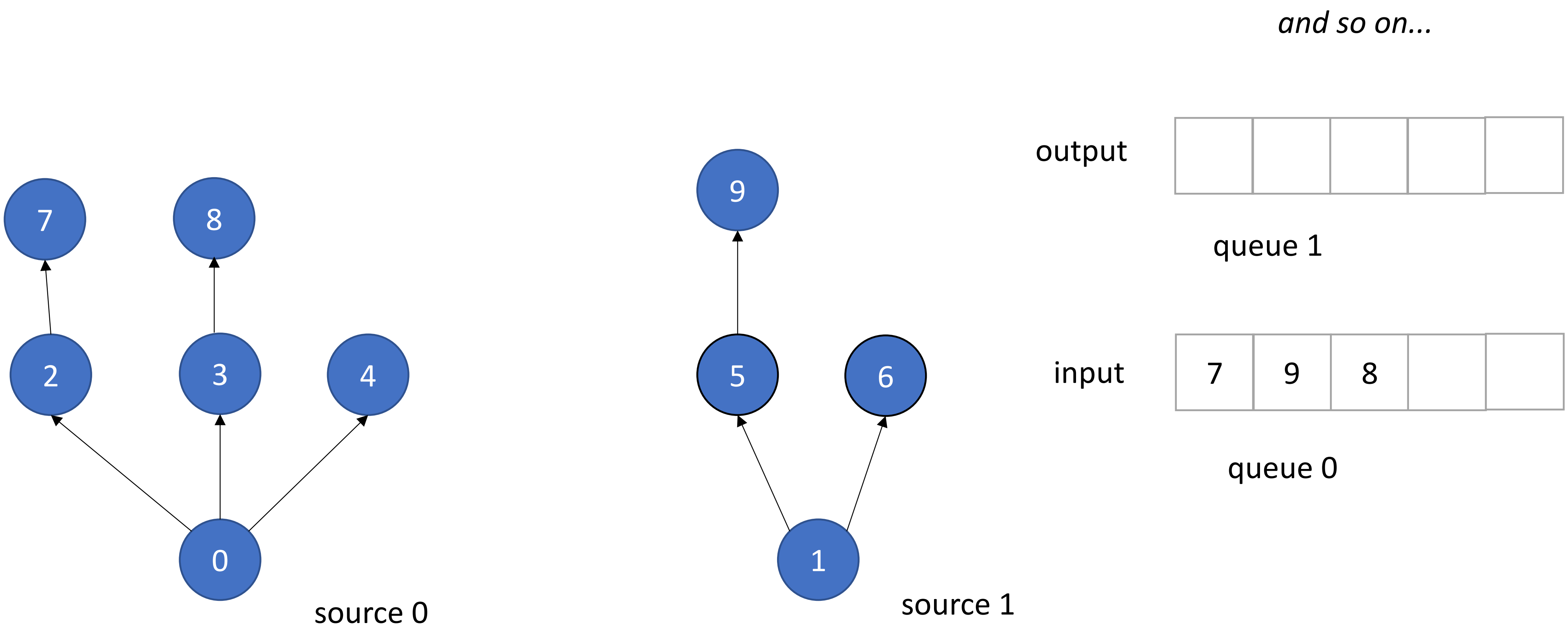
Input/Output Queues

- Example: Information flow in graph applications:



Input/Output Queues

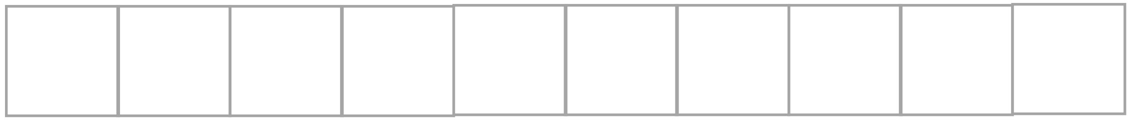
- Example: Information flow in graph applications:



Implementation

Implementation

Allocate a contiguous array

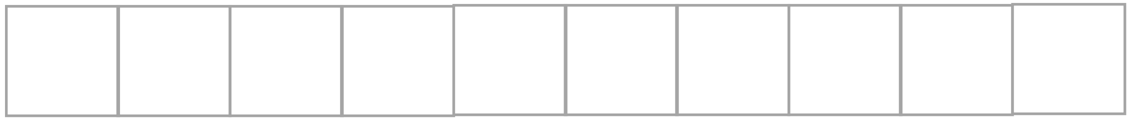


Pros:
?

Cons:
?

Implementation

Allocate a contiguous array



Pros:
+ fast!
+ we can use indexes instead of addresses

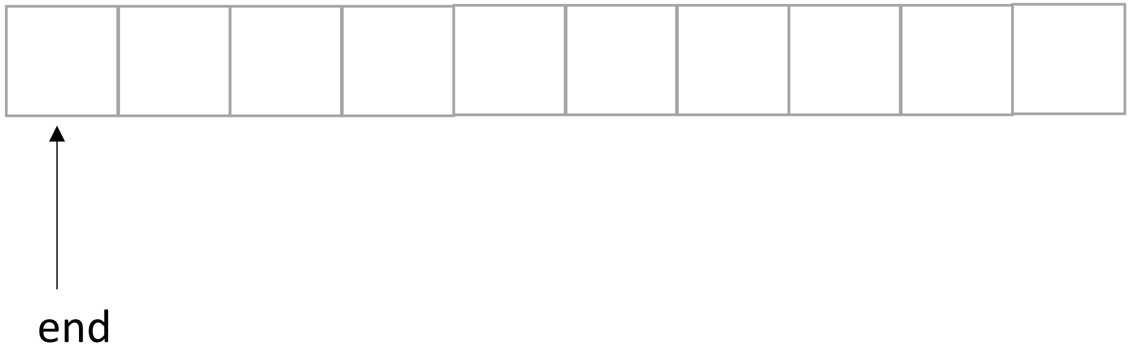
Cons:
- need to reason about overflow!

Note on terminology

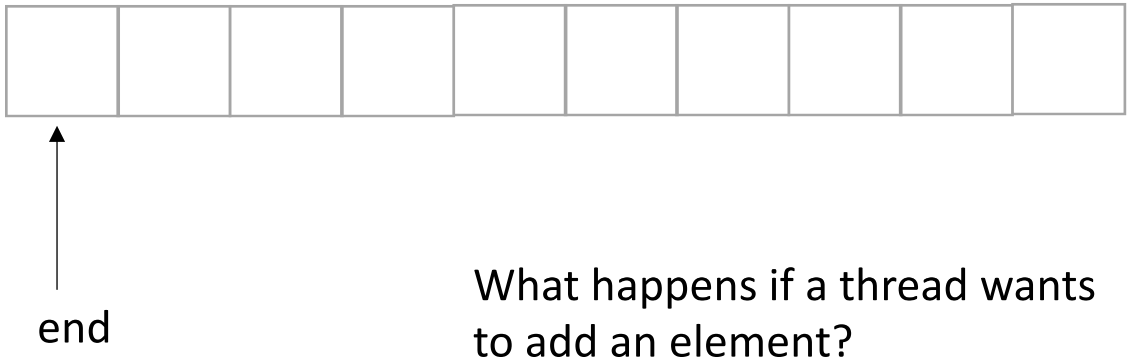
- Head/tail - often used in queue implementations, but switches when we start doing circular buffers.

- Front/end - To avoid confusion, we will use front/end for input/output queues.

Implementation

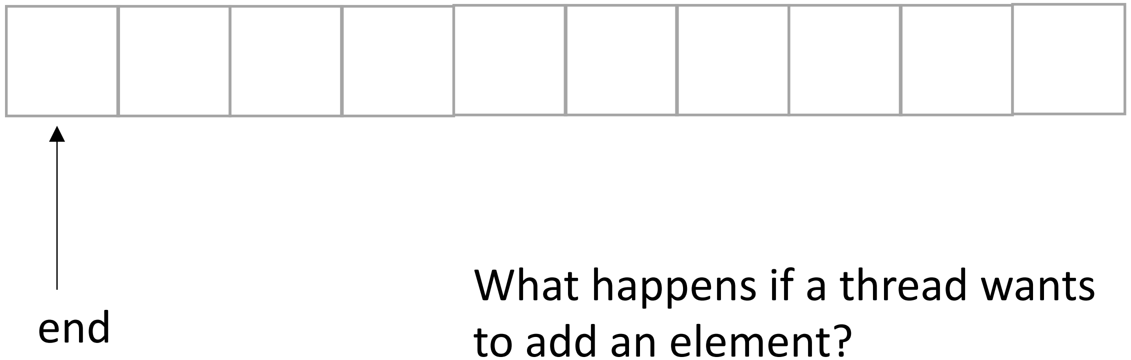


Implementation



What happens if a thread wants to add an element?

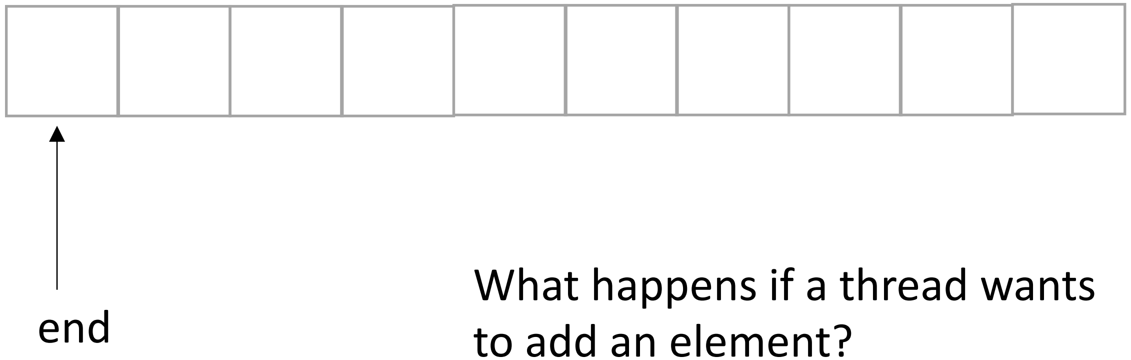
Implementation



What happens if a thread wants to add an element?

Think sequentially:

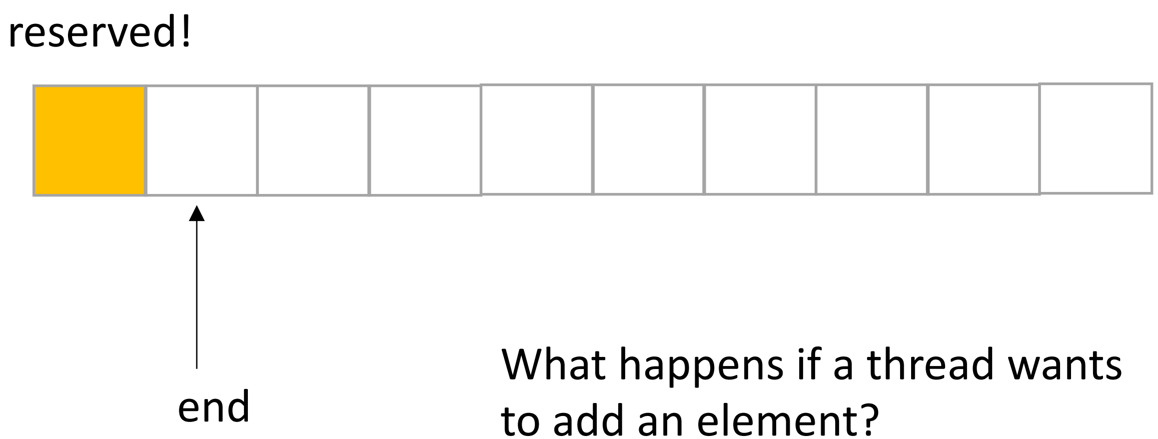
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

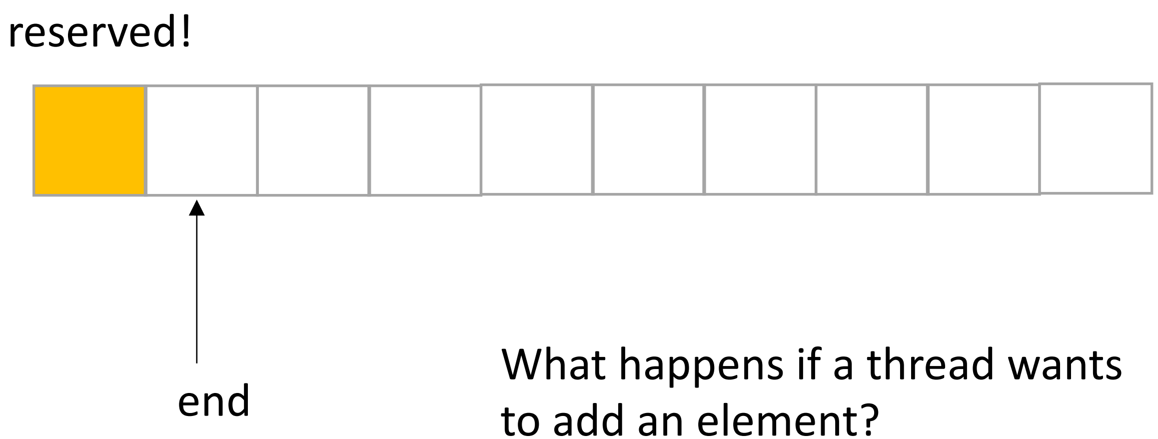
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

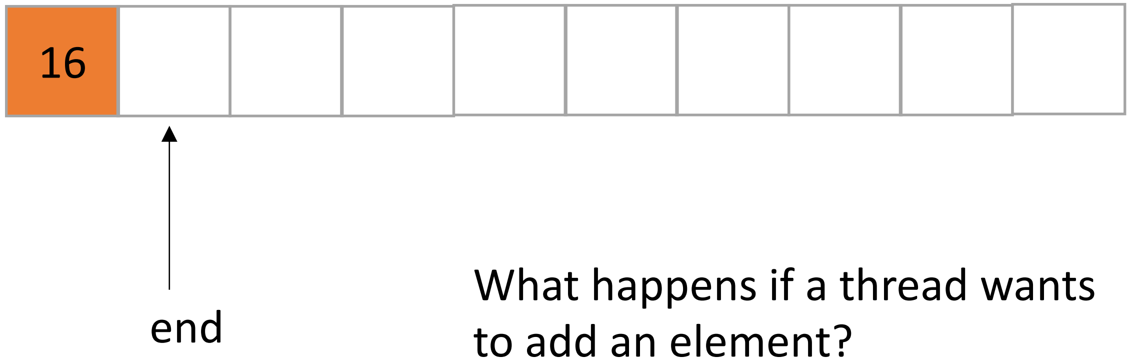
Implementation



What happens if a thread wants to add an element?

Think sequentially:
* reserve a space - increment end
* add the element

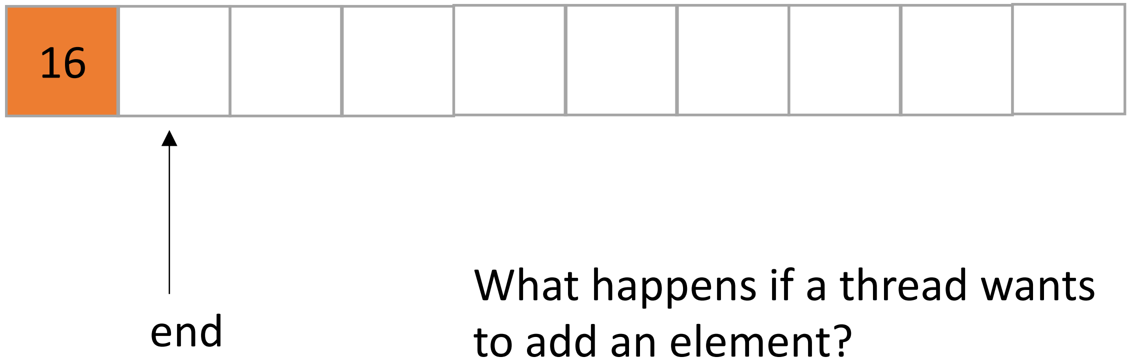
Implementation



What happens if a thread wants to add an element?

Think sequentially:
* reserve a space - increment end
* add the element

Implementation

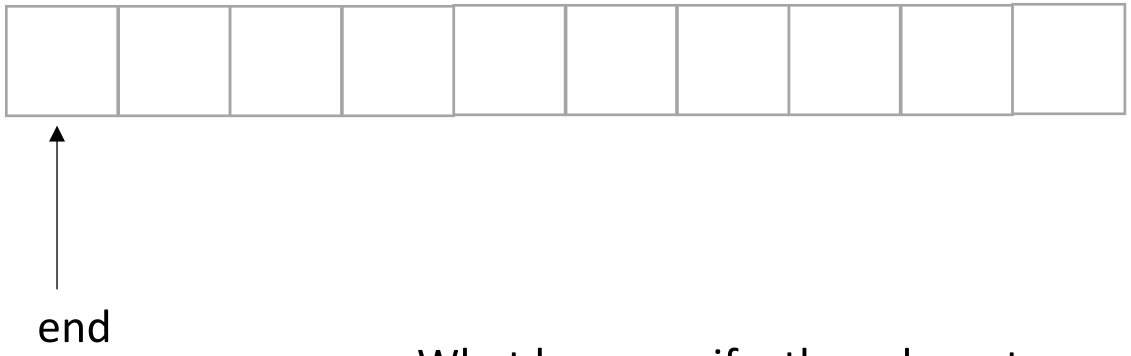


What happens if a thread wants to add an element?

Think sequentially:
* reserve a space - increment end
* add the element

done!

Implementation

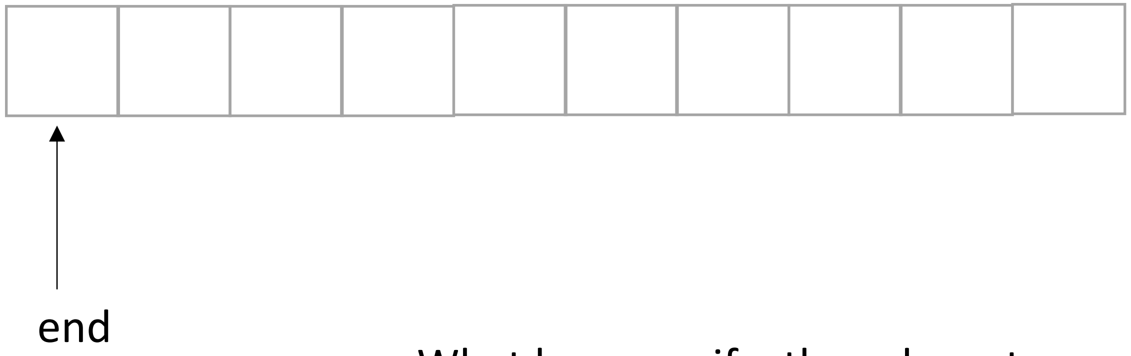


What happens if a thread wants
to add an element?

Think concurrently:

Two threads cannot reserve the same space!
We've seen this before

Implementation

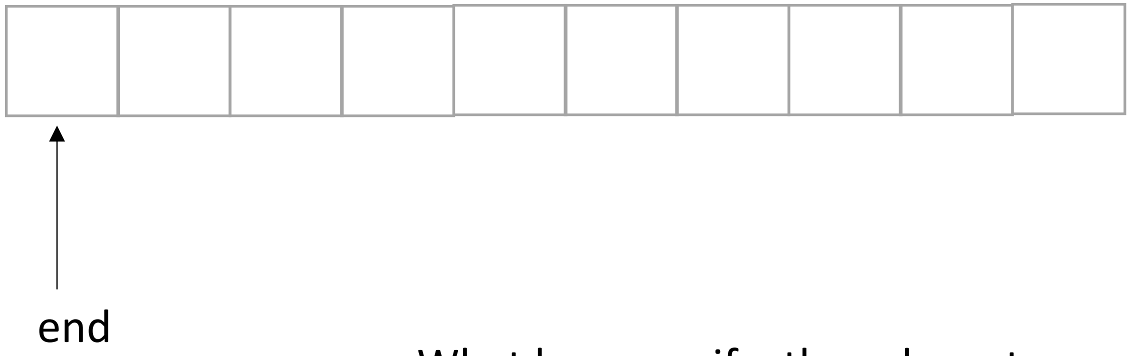


What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation



<u>Thread 0:</u>
enq(6);

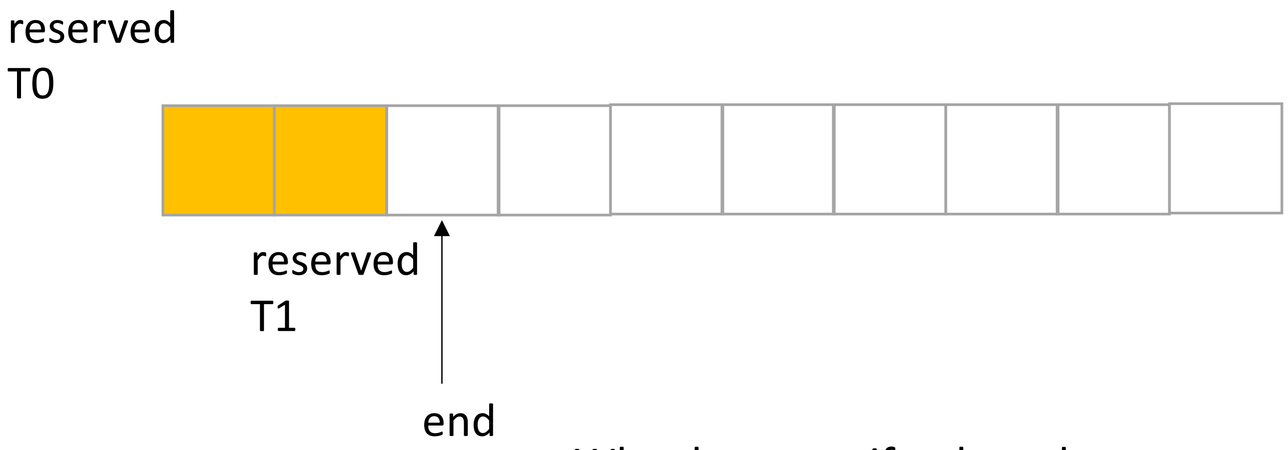
<u>Thread 1:</u>
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation



<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

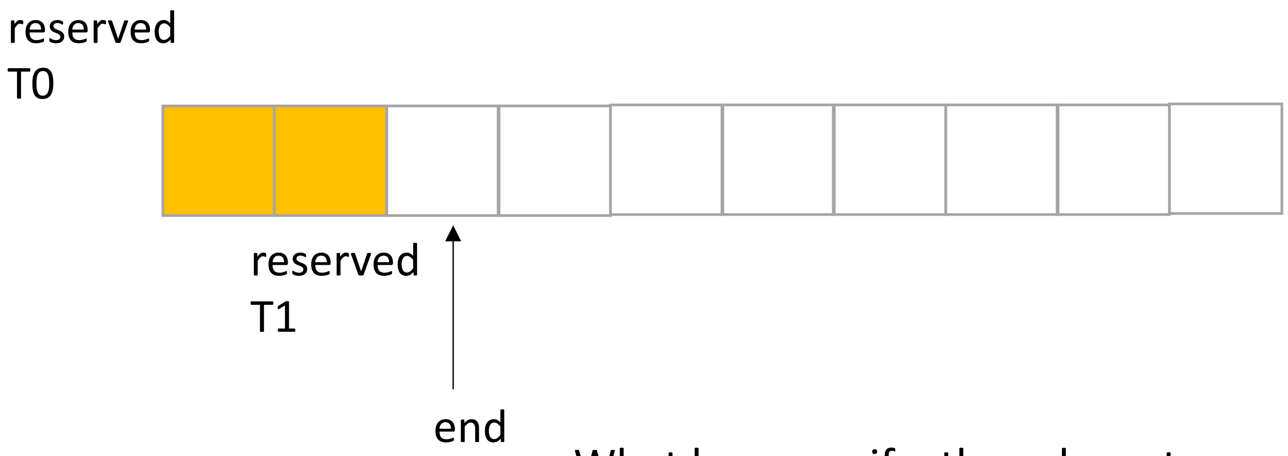
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation

does it matter which order threads add their data?



Thread 0:
enq(6);

Thread 1:
enq(7);

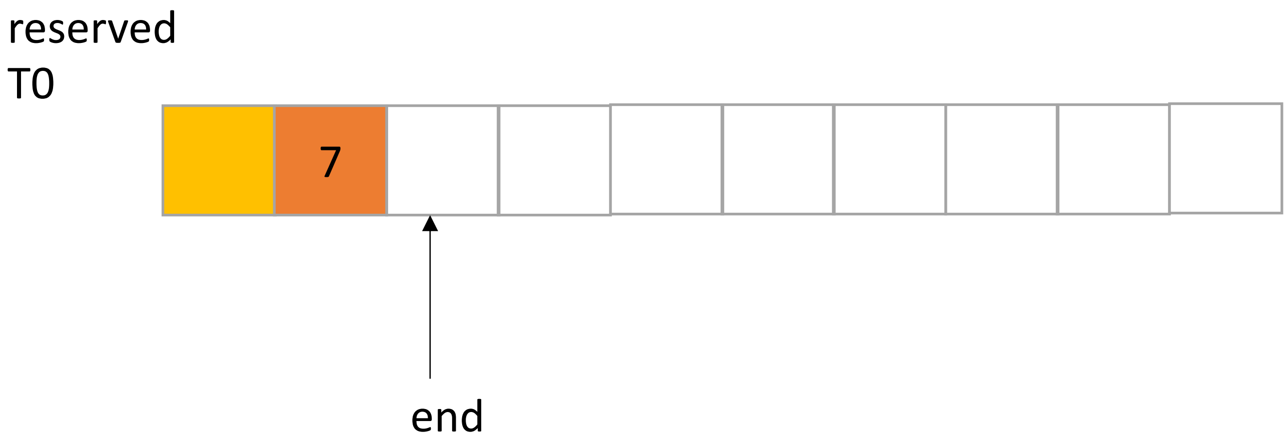
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation

does it matter which order threads add their data?



<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

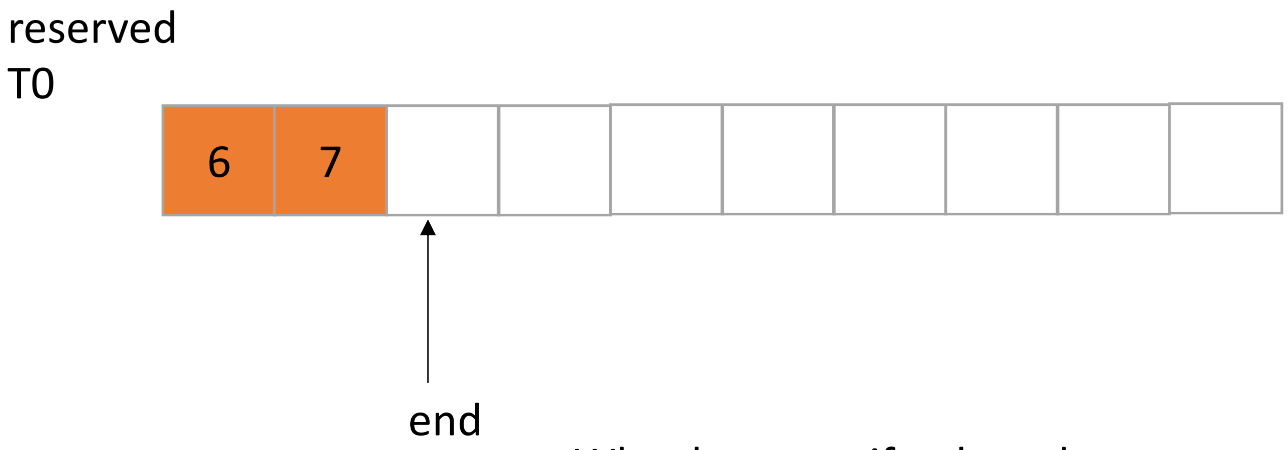
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation

does it matter which order threads add their data? No! Because there are no deqs!



<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

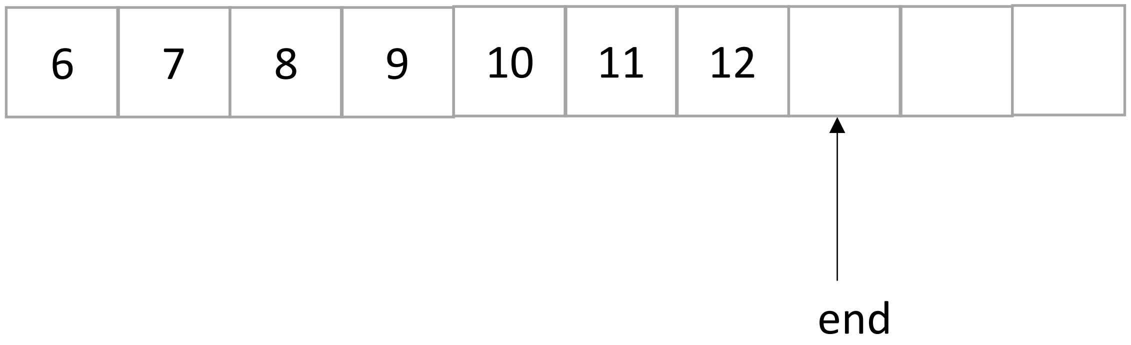
```
reserved_index = atomic_fetch_add(&end, 1);
```

```
class InputOutputQueue {
  private:
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     int size() {
        return end.load();
     }
 }
```

How to protect against overflows?

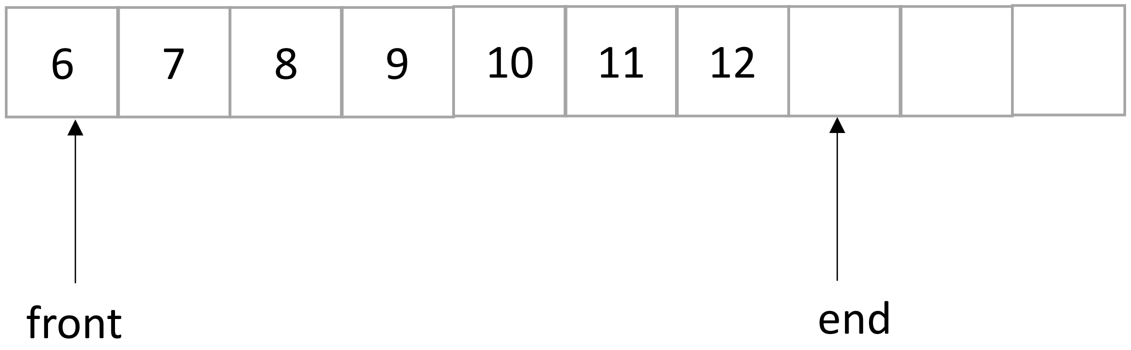
What about Input?

- Now we only do deqs



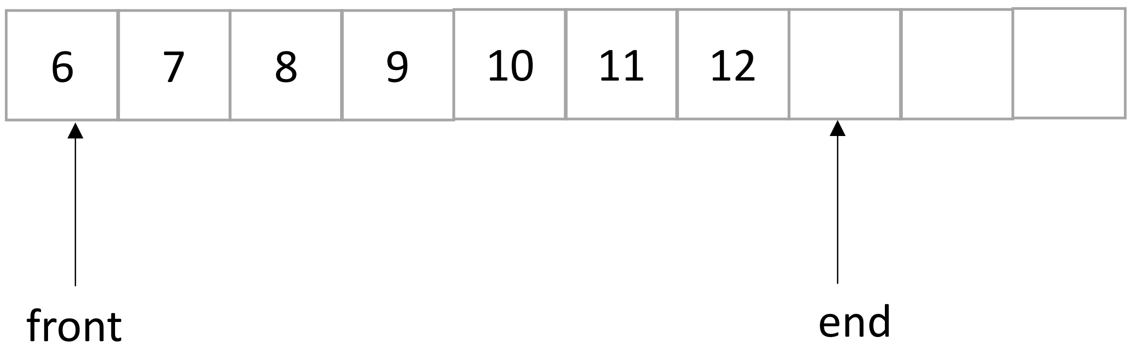
What about Input?

- Now we only do deqs



What about Input?

- Now we only do deqs



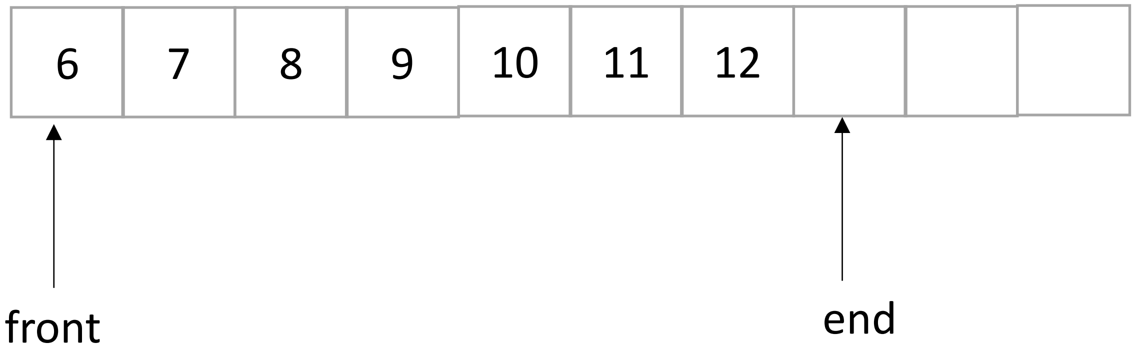
What happens if a thread wants
to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

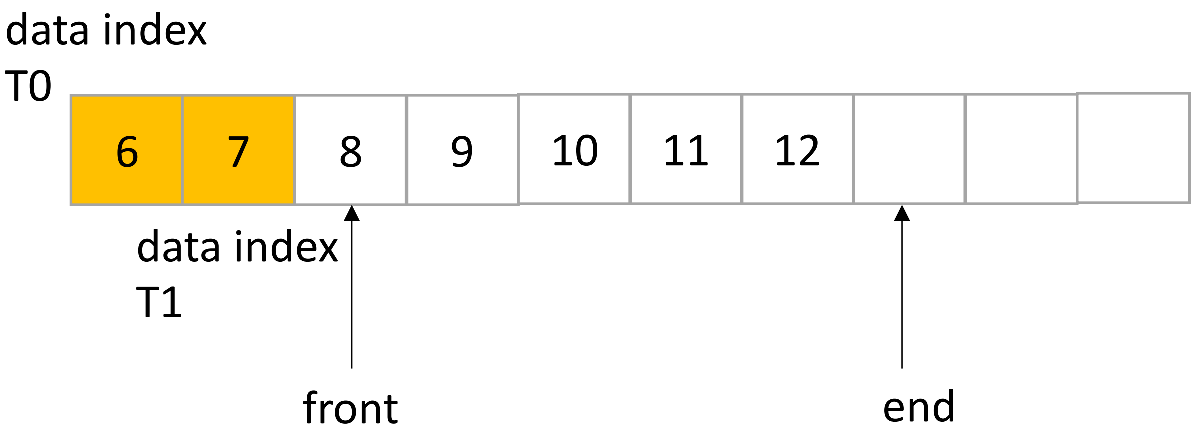
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

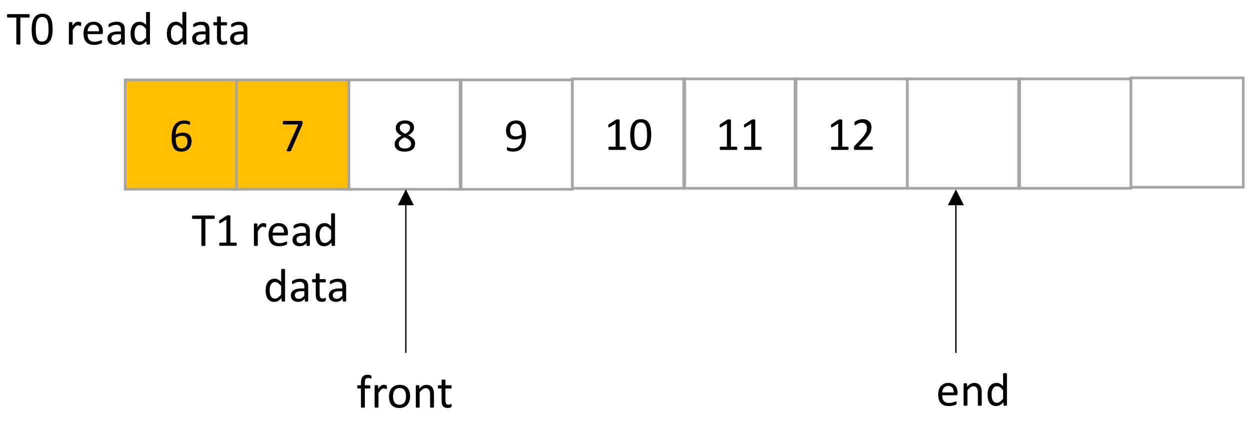
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

What about Input?

- Now we only do deqs



<u>Thread 0:</u>
deq(); // reads 6

<u>Thread 1:</u>
deq(); // reads 7

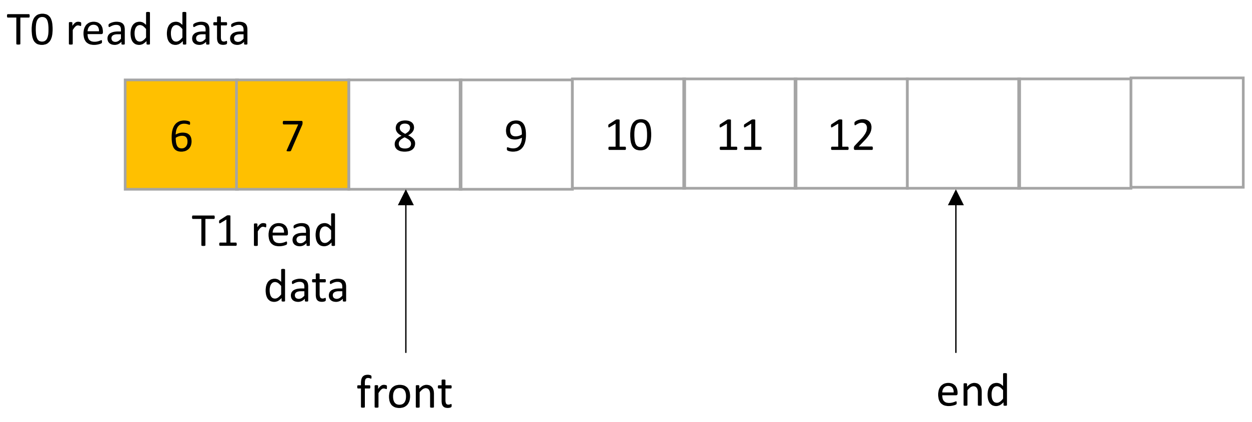
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

What about Input?

- Now we only do deqs



How to implement a stack?

Thread 0:
deq(); // reads 6

Thread 1:
deq(); // reads 7

What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return ??;
     }
 }
```

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return ??;
     }
 }
```

How about size?

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return end.load() - front.load();
     }
 }
```

how about size?

how do we reset?

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return end.load() - front.load();
     }
 }
```

how about size?

how do we reset?
Reset front and end

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return end.load() - front.load();
     }
 }
```

how about size?

how do we reset?
Reset front and end

does the list need
to be atomic?

Schedule

- Producer Consumer queues
  - **Synchronous**
  - Circular buffer

Producer Consumer Queues

- 1 enq, 1 deq
  - enq’er cannot deq
  - deq’er cannot enq
- Example: printf:
  - your program equeues values to print
  - the terminal process dequeues values and prints them

Synchronous Producer Consumer Queues

- First implementation:
  - Synchronous
  - Slow
  - Good for debugging

Synchronous Producer Consumer Queues

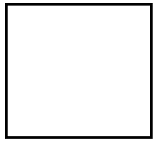
- First implementation:
  - Synchronous
  - Slow
  - Good for debugging

- enq does not return until value is deq'ed

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

`enq(7);`



<u>Consumer Thread</u>

`deq();`

Synchronous Producer Consumer Queues

Producer Thread

`enq(7);`

7

wait

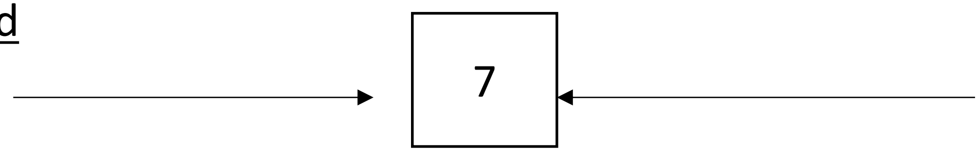
Consumer Thread

`deq();`

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

`enq(7);`

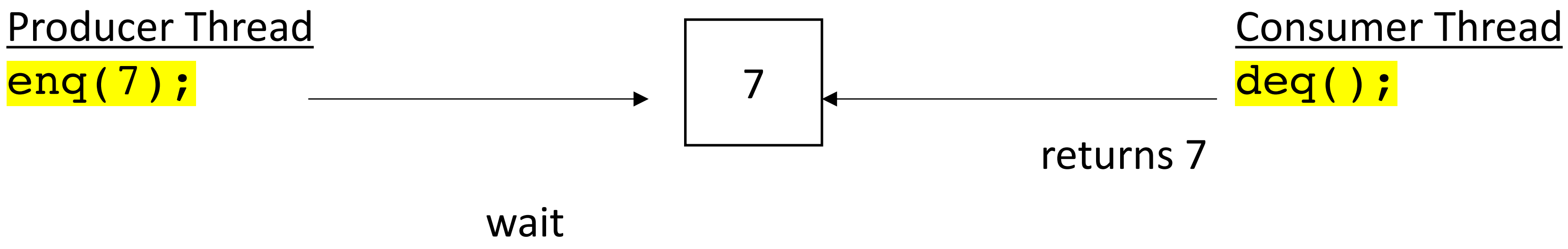


<u>Consumer Thread</u>

`deq();`

wait

Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
```

<u>Consumer Thread</u>

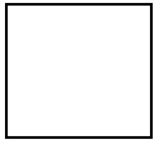
```
deq();
```

both can continue

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
sleep();
enq(7);
```



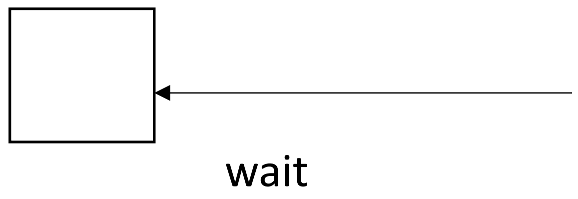
<u>Consumer Thread</u>

```
deq();
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
sleep();
enq(7);
```



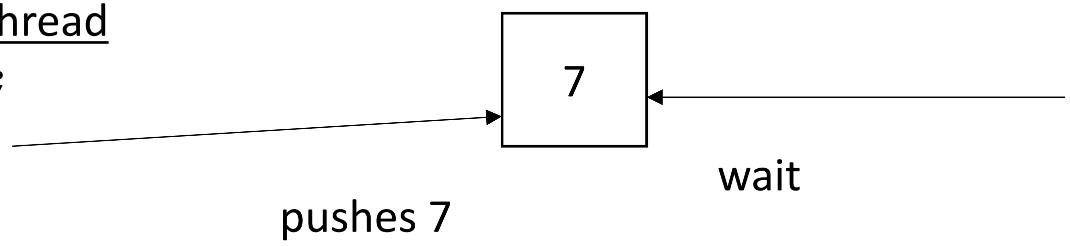
<u>Consumer Thread</u>

```
deq();
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

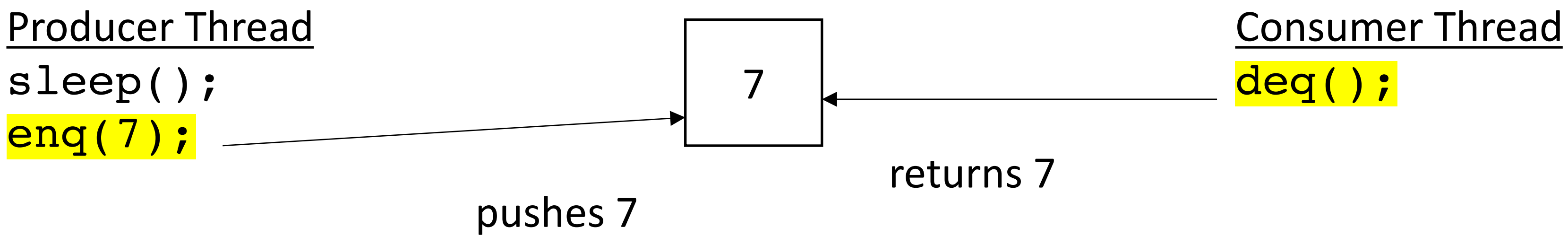
```
sleep();
enq(7);
```



<u>Consumer Thread</u>

```
deq();
```

Synchronous Producer Consumer Queues



They both can continue

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
```

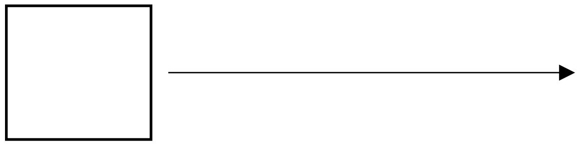
<u>Consumer Thread</u>

```
deq();
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

`enq(7);`



<u>Consumer Thread</u>

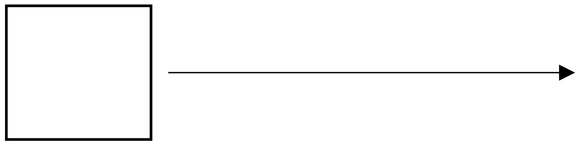
`deq();`

can the consumer just read?

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

`enq(7);`



<u>Consumer Thread</u>

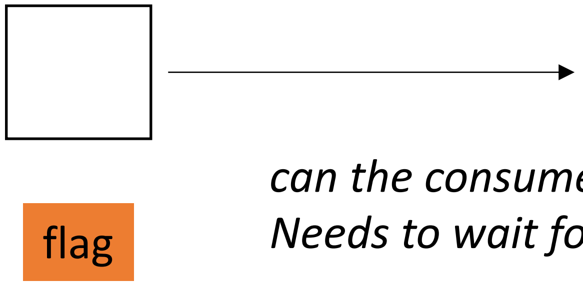
`deq();`

can the consumer just read?
Needs to wait for a value to appear

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

`enq(7);`



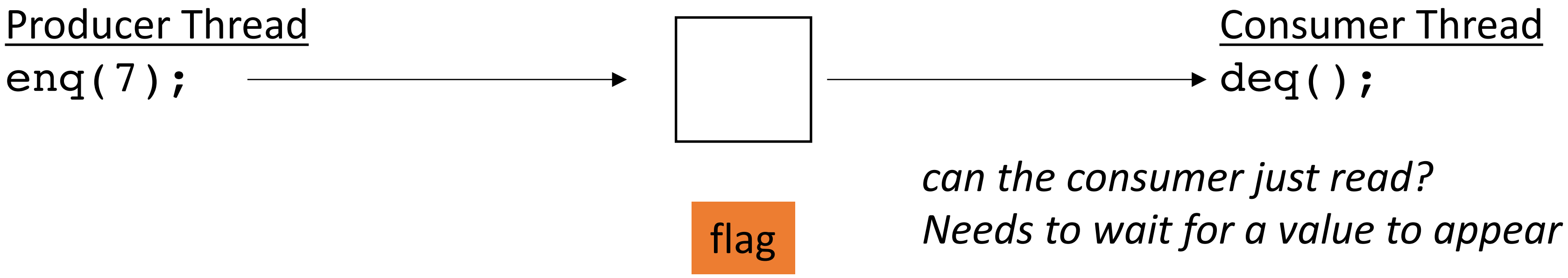
<u>Consumer Thread</u>

`deq();`

can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

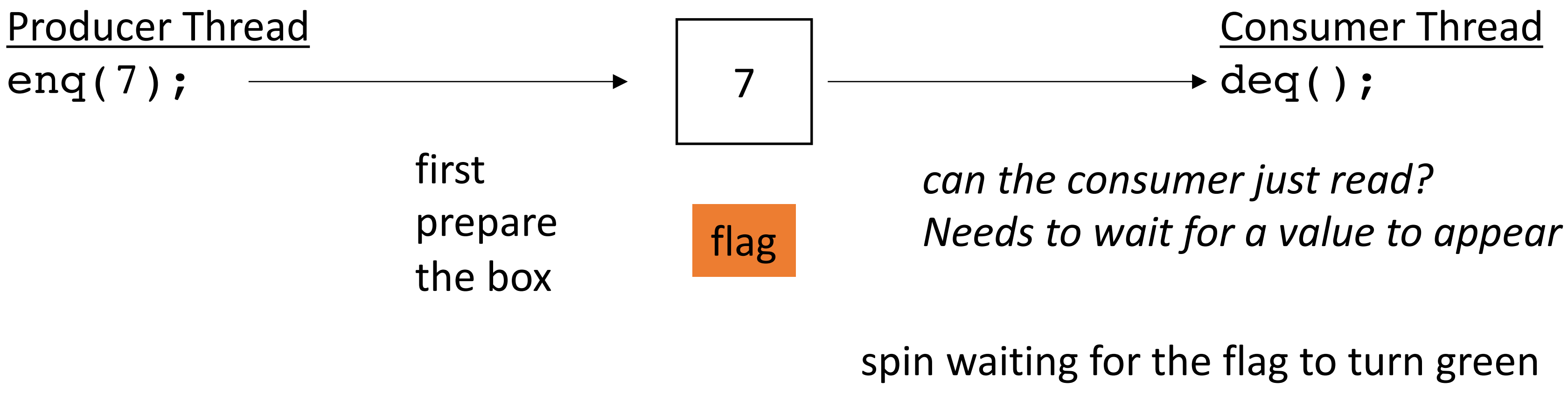
Synchronous Producer Consumer Queues



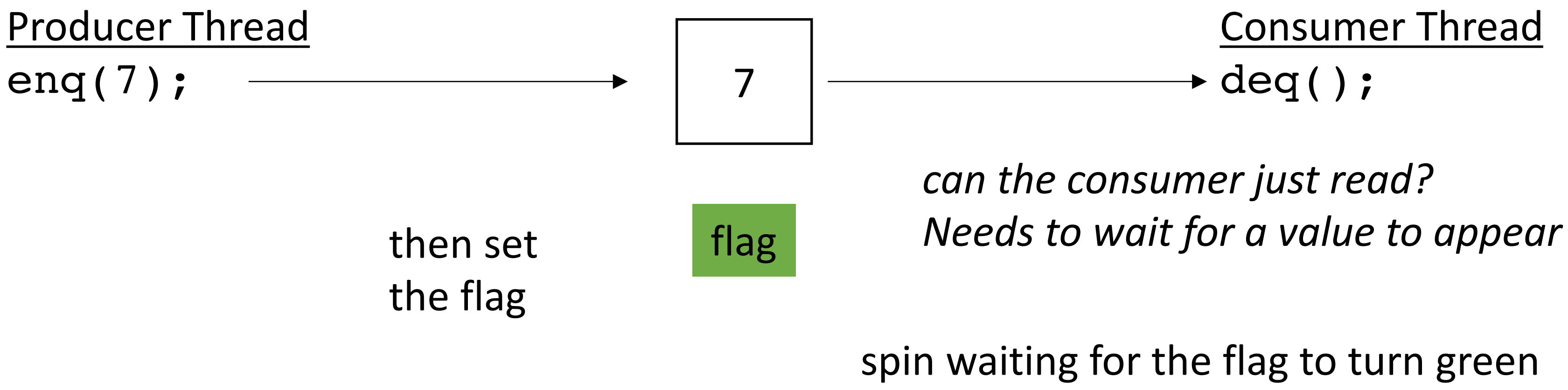
can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

now the consumer can read from the box!

<u>Producer Thread</u>
`enq(7);`

7

flag

<u>Consumer Thread</u>
`deq();`

then set
the flag

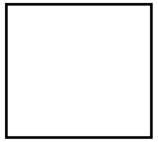
can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



flag

Consumer Thread

```
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread
```
enq(7);
```

flag

Consumer Thread
```
deq();
deq();
```

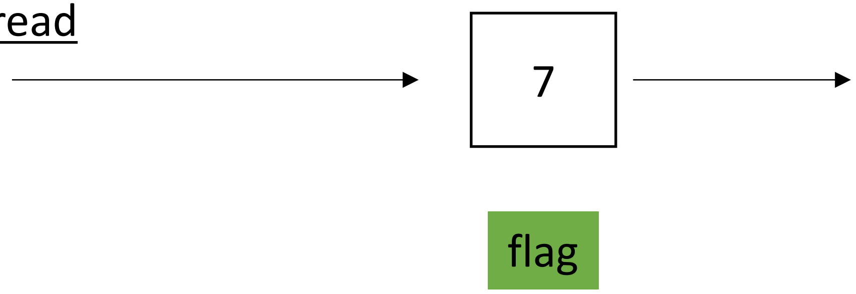
what happens when there are two deqs?

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
 }
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

`enq(7);`



<u>Consumer Thread</u>

`deq();`
`deq();`

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
}
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

`enq(7);`

7

flag

<u>Consumer Thread</u>

`deq();`

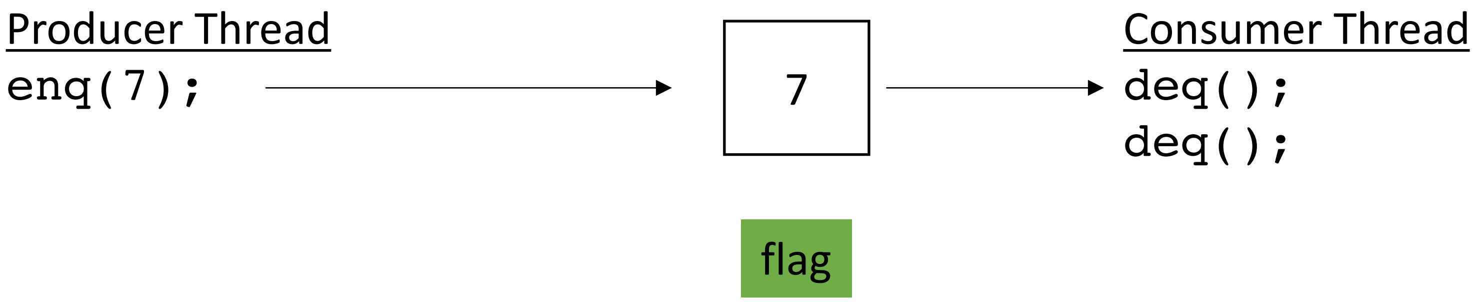
`deq();`

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
}
```

what happens in the next deq?

How to fix?

Synchronous Producer Consumer Queues

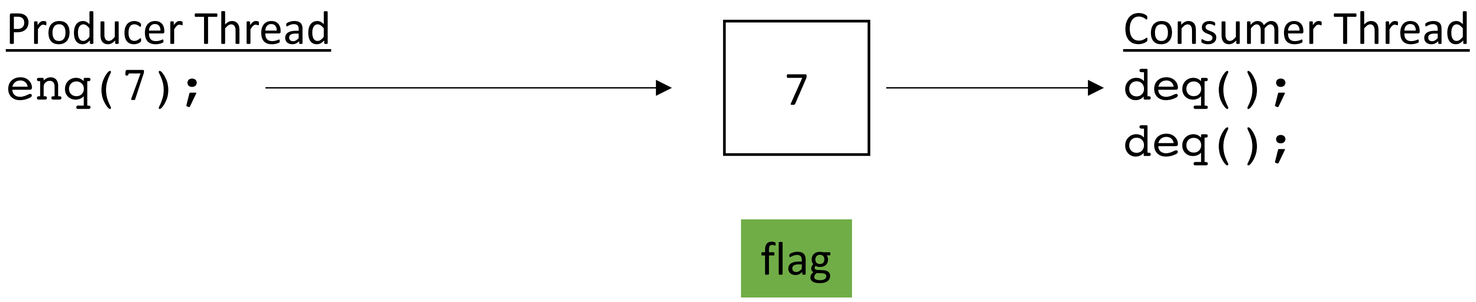



```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

what happens in the next deq?

How to fix?

Synchronous Producer Consumer Queues

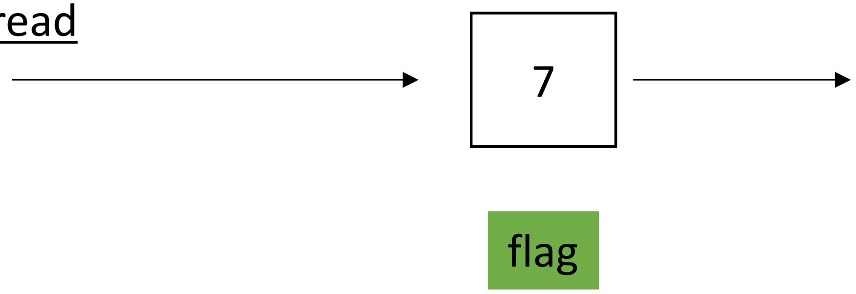



```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```

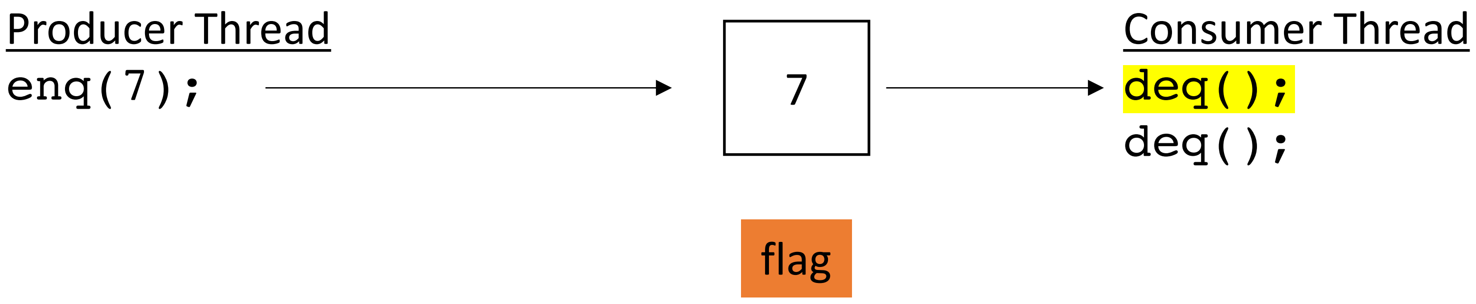


Consumer Thread

```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

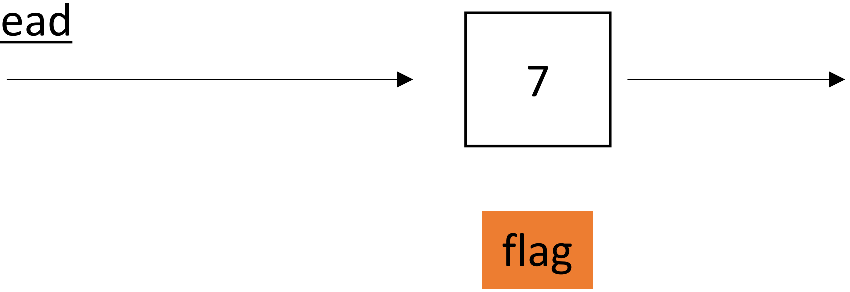



```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
```



<u>Consumer Thread</u>

```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

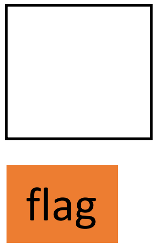
waiting like we are supposed to

Synchronous Producer Consumer Queues

reset (now with extra enq)

<u>Producer Thread</u>

```
enq(7);
enq(8);
```



<u>Consumer Thread</u>

```
deq();
deq();
```

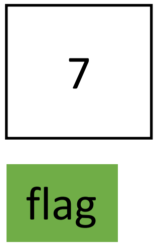
extra enq

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
enq(8);
```



<u>Consumer Thread</u>

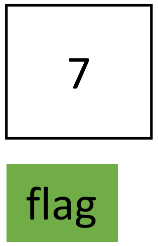
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
enq(8);
```



<u>Consumer Thread</u>

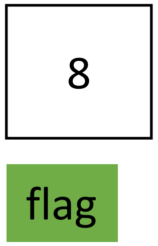
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
enq(8);
```



<u>Consumer Thread</u>

```
deq();
deq();
```

7 was dropped!

how to fix?

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```

8

flag

Consumer Thread

```
deq();
deq();
```

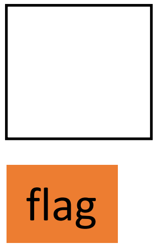
7 was dropped!

how to fix?

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

reset



Producer Thread

```
enq(7);
enq(8);
```

Consumer Thread

```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```

7

flag

Consumer Thread

```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```

7

flag

Consumer Thread

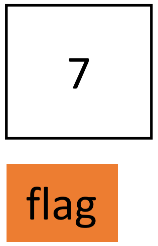
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
}
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
enq(8);
```

7

flag

<u>Consumer Thread</u>

```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
}
```

Schedule

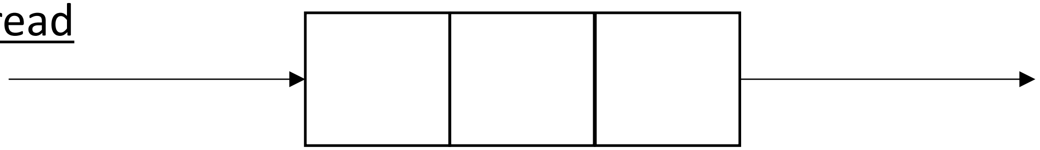
- Producer Consumer Queues
  - Synchronous
  - Circular buffer

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
```



Consumer Thread

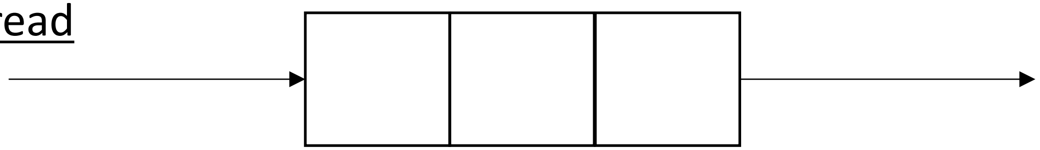
```
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
```



Consumer Thread

```
deq();
deq();
deq();
```

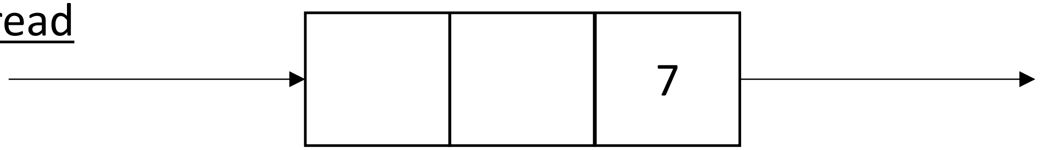
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
```



Consumer Thread

```
deq();
deq();
deq();
```

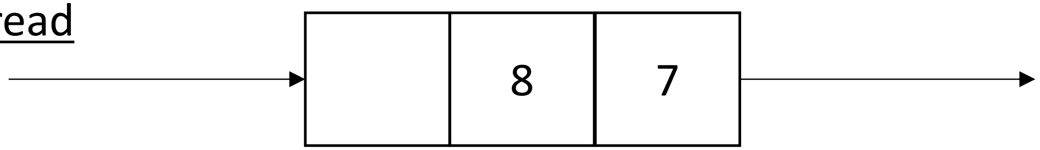
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
```



Consumer Thread

```
deq();
deq();
deq();
```

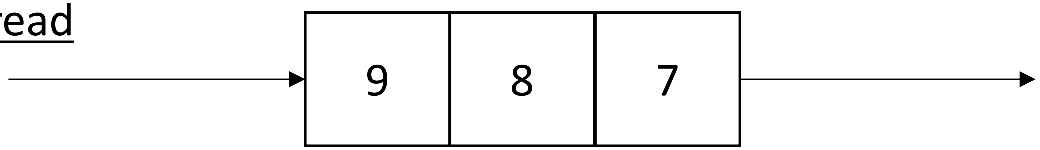
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
```



Consumer Thread

```
deq();
deq();
deq();
```

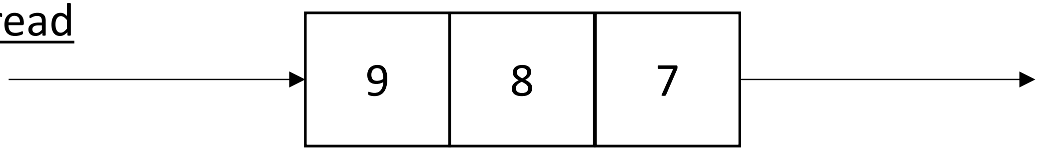
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

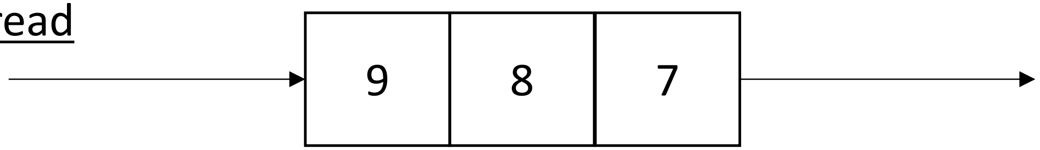
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
enq(10);
```



<u>Consumer Thread</u>

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

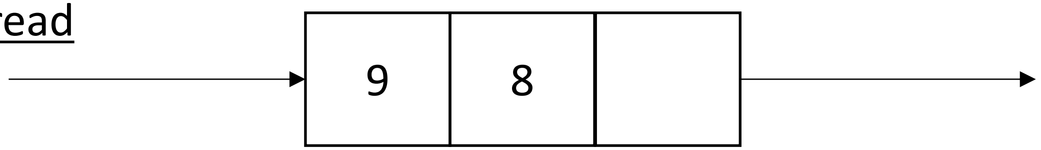
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

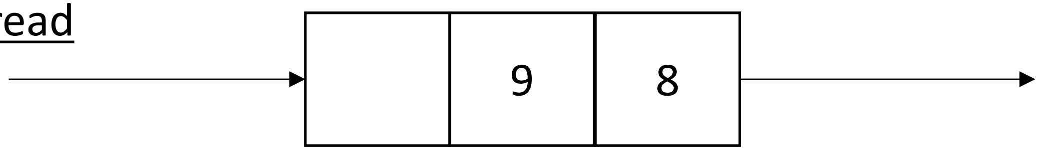
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

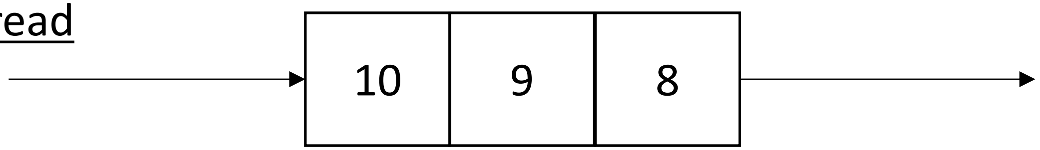
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

finishes

no waiting for producer (while there is room)

returns 7

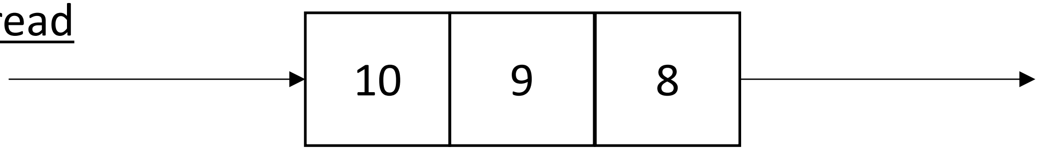
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

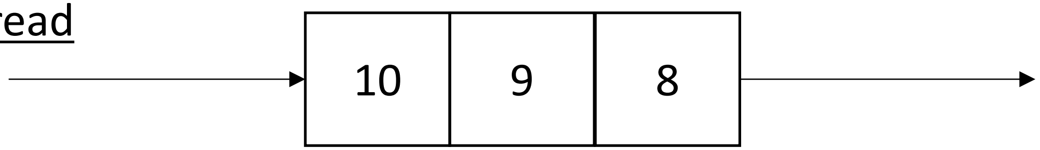
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
enq(10);
```



<u>Consumer Thread</u>

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

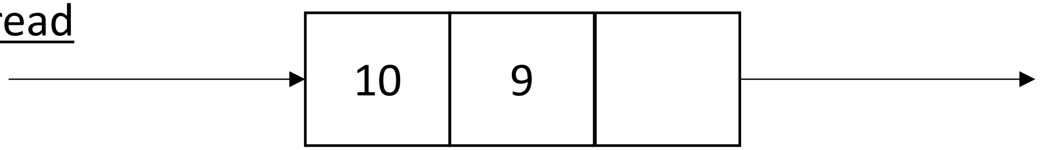
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

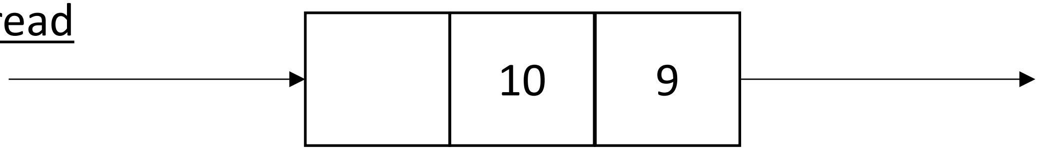
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
enq(10);
```



<u>Consumer Thread</u>

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

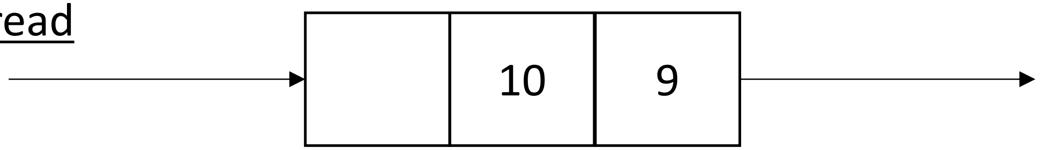
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
enq(10);
```



<u>Consumer Thread</u>

```
deq();
deq();
deq();
```

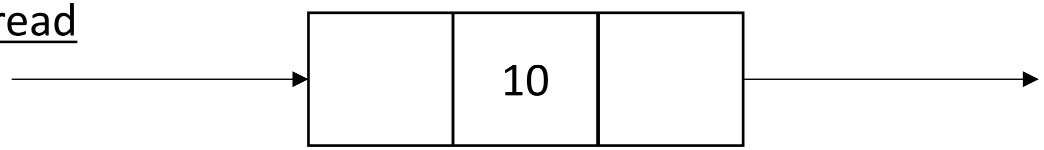
returns 9

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
enq(10);
```



<u>Consumer Thread</u>

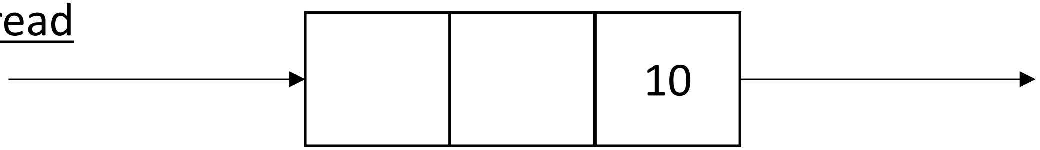
```
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

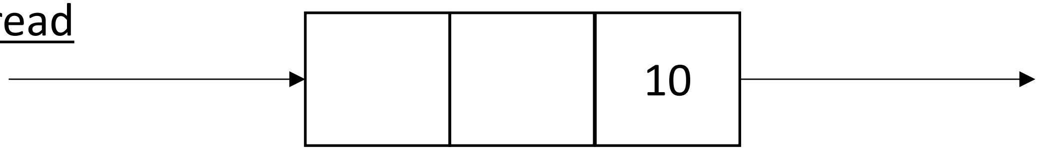
```
deq();
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

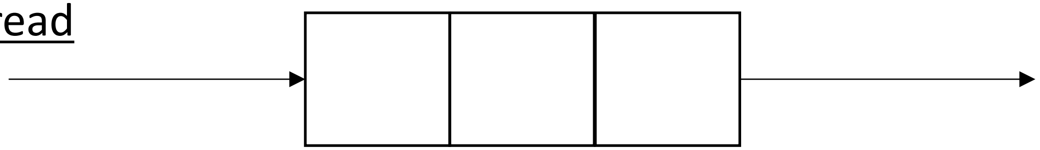
```
deq();
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
enq(10);
```



<u>Consumer Thread</u>

```
deq();
deq();
deq();
deq();
deq();
```

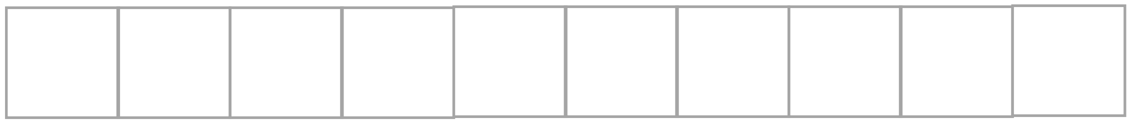
blocks when there is nothing in the queue

Producer Consumer Queues

- How do we implement it?

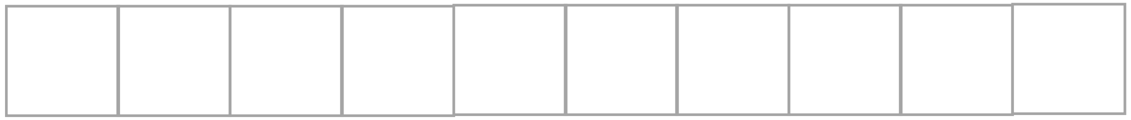
Producer Consumer Queues

- Start with a fixed size array



Producer Consumer Queues

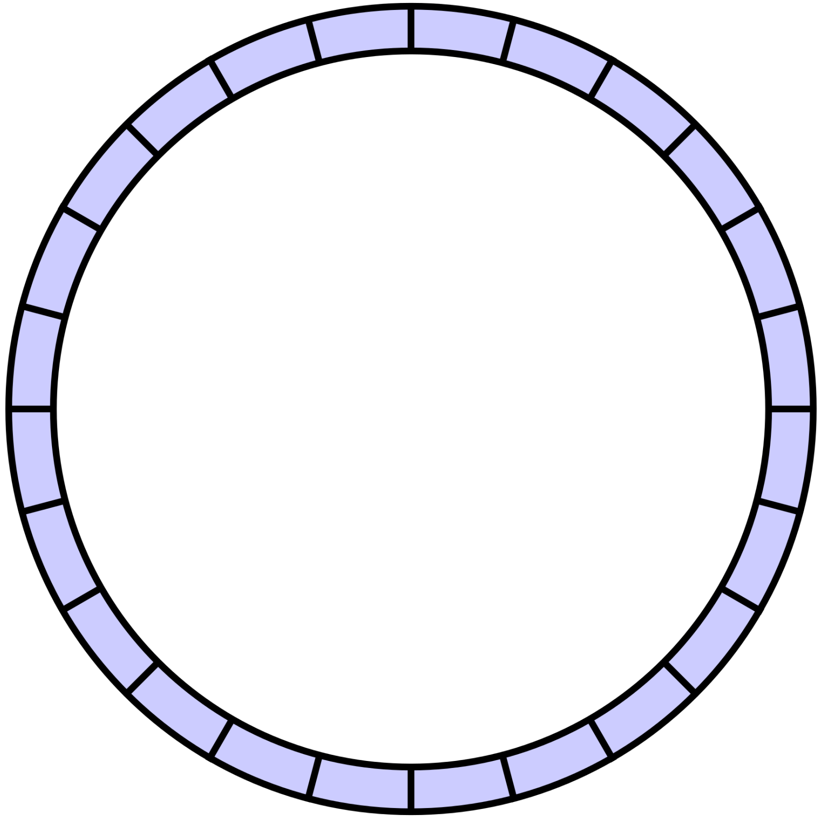
- Start with a fixed size array



We will use what is called a *circular buffer method*

Producer Consumer Queues

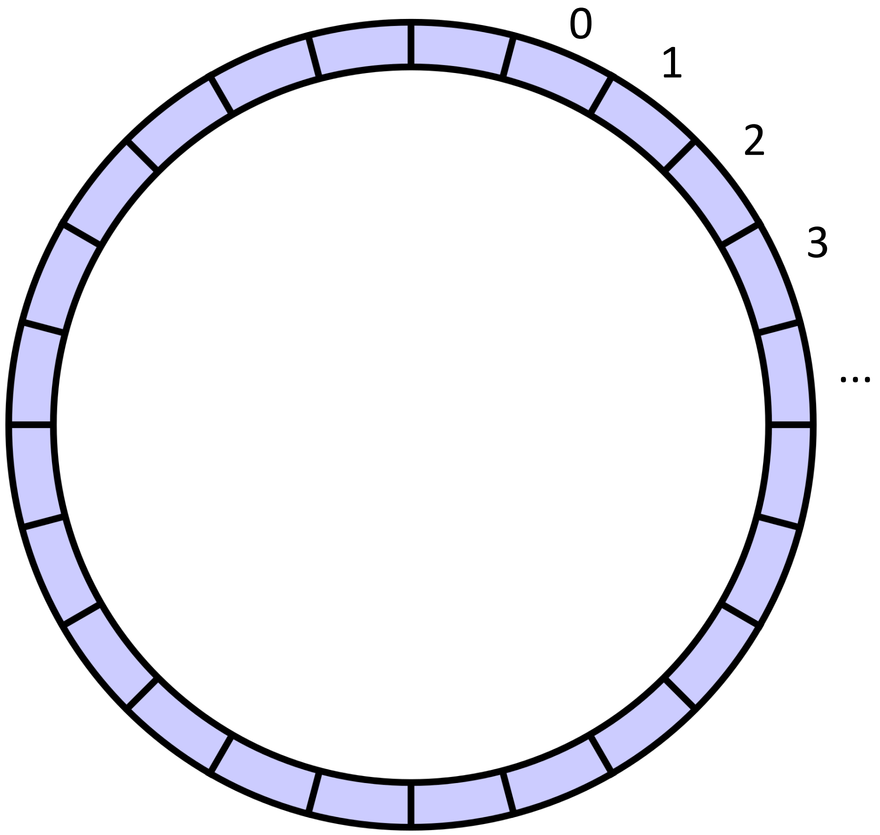
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

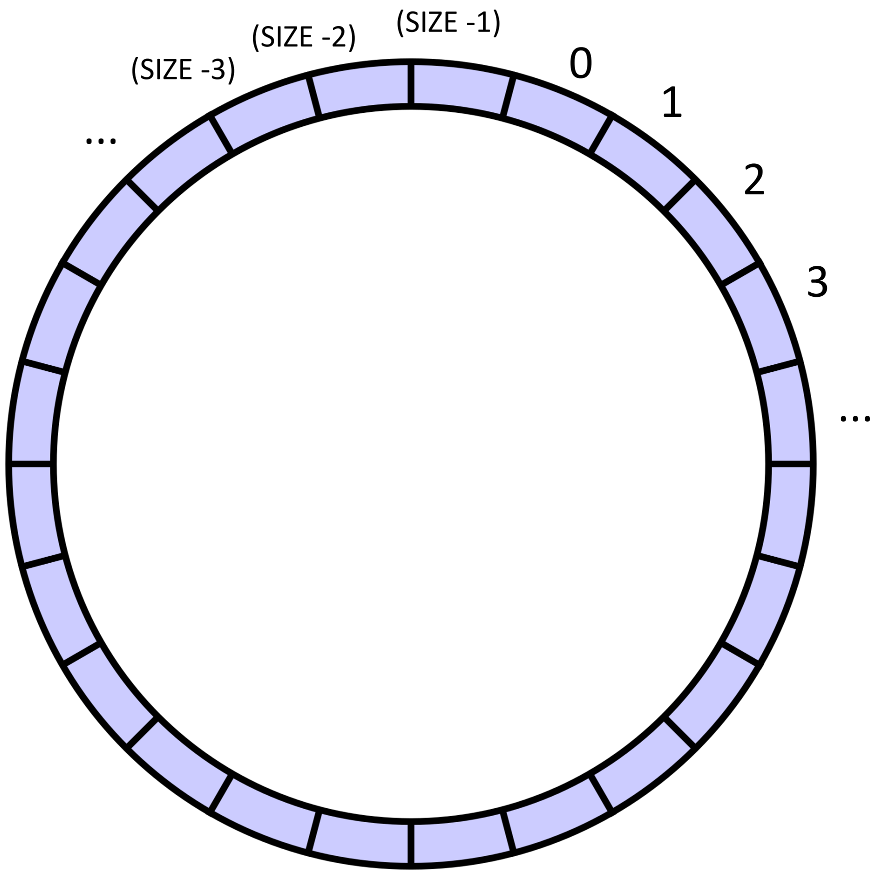
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

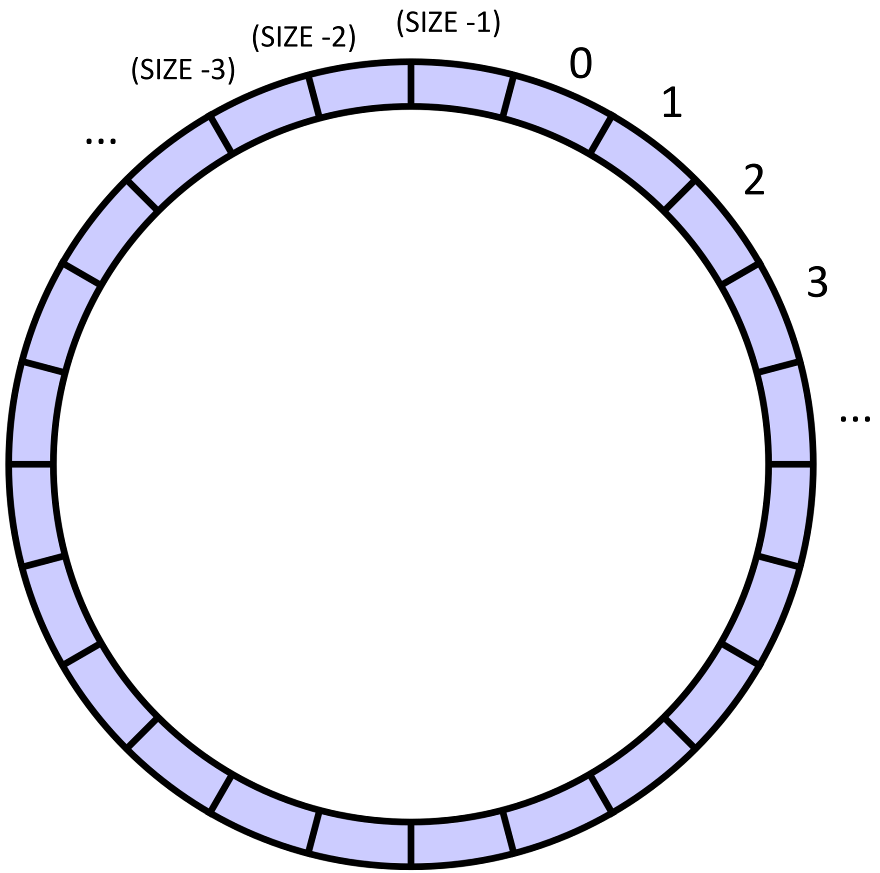


indexes will circulate in order and wrap around

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

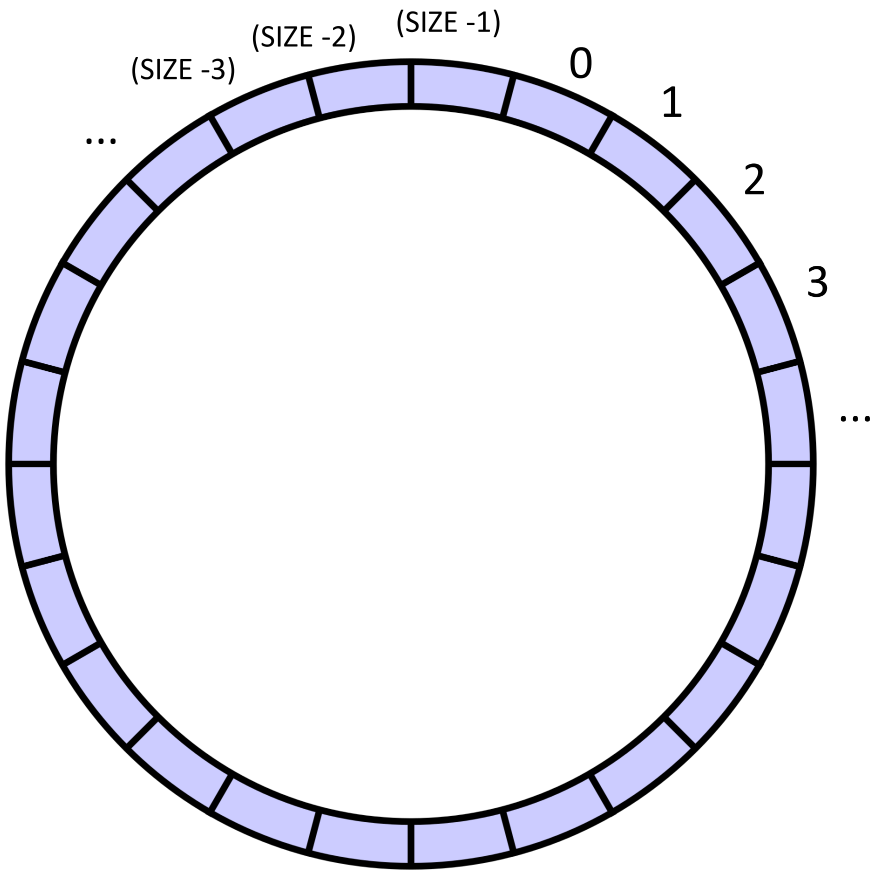
we will assume modular arithmetic:

if x = (SIZE - 1) then
x + 1 == 0;

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

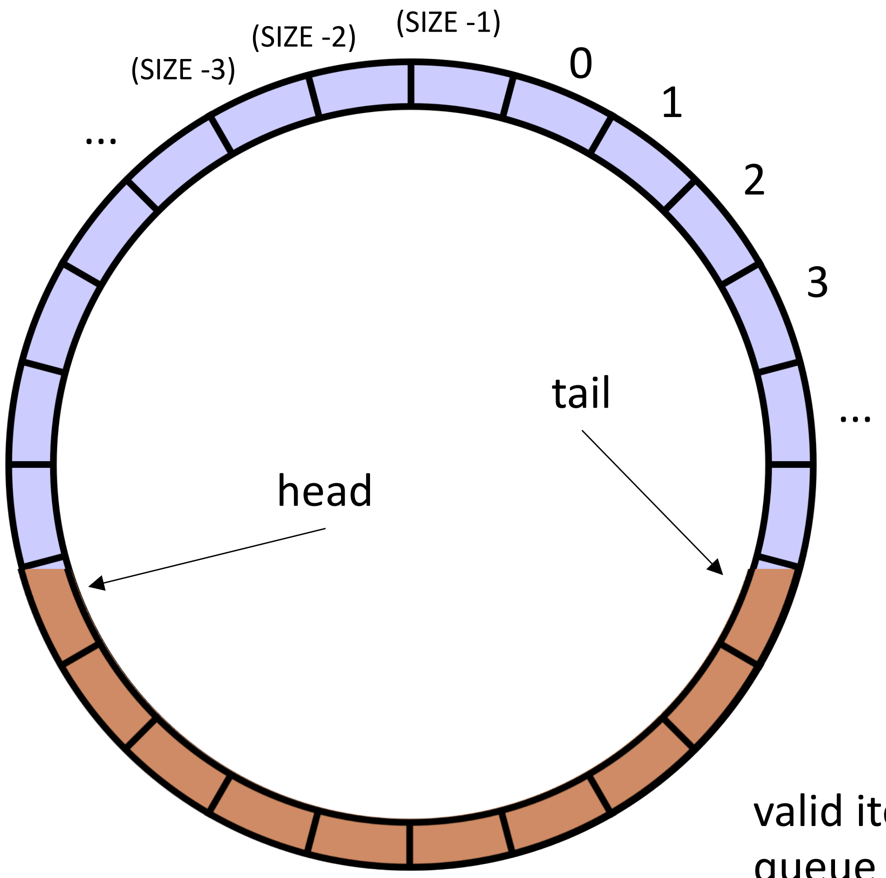
Two variables to keep track of where to deq and enq:

head and tail

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail:

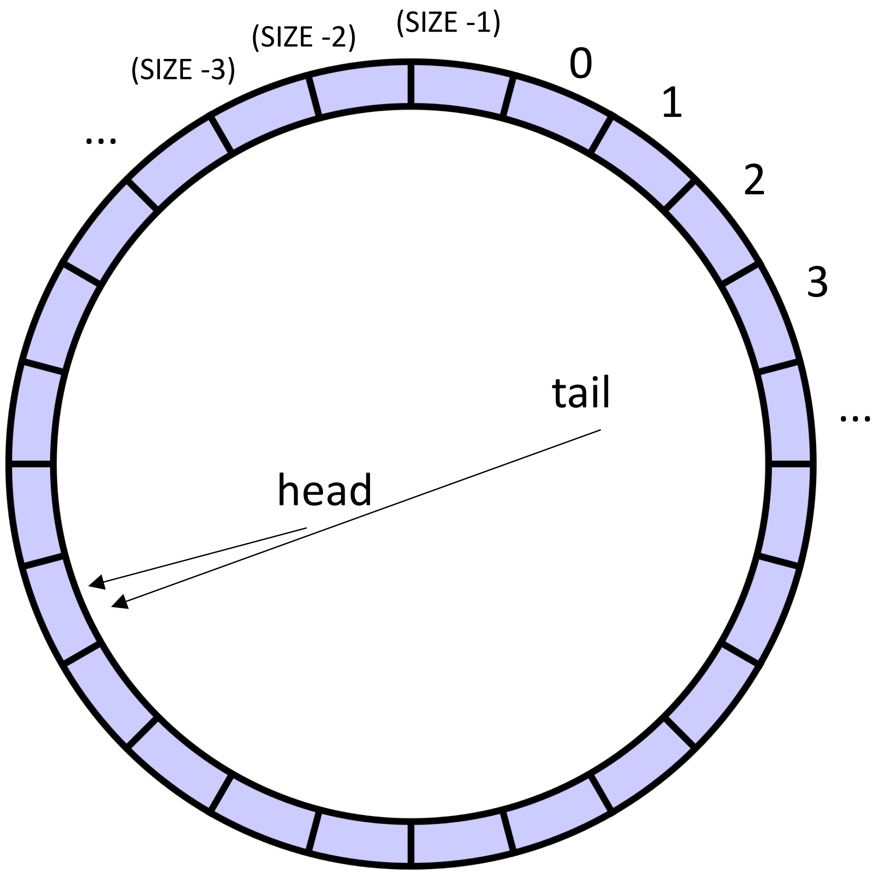
enq to the head, deq from the tail

valid items in the queue

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

conceptually it is a circle

Producer Consumer Queues

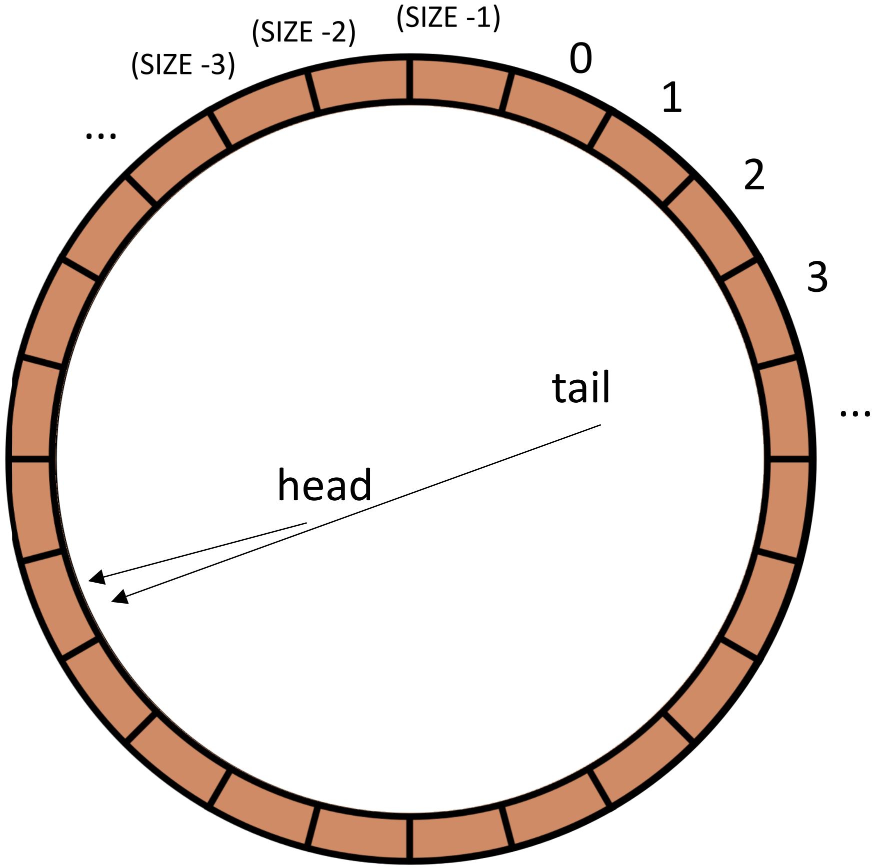
- Start with a fixed size array

Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

Full queue is when head == tail?

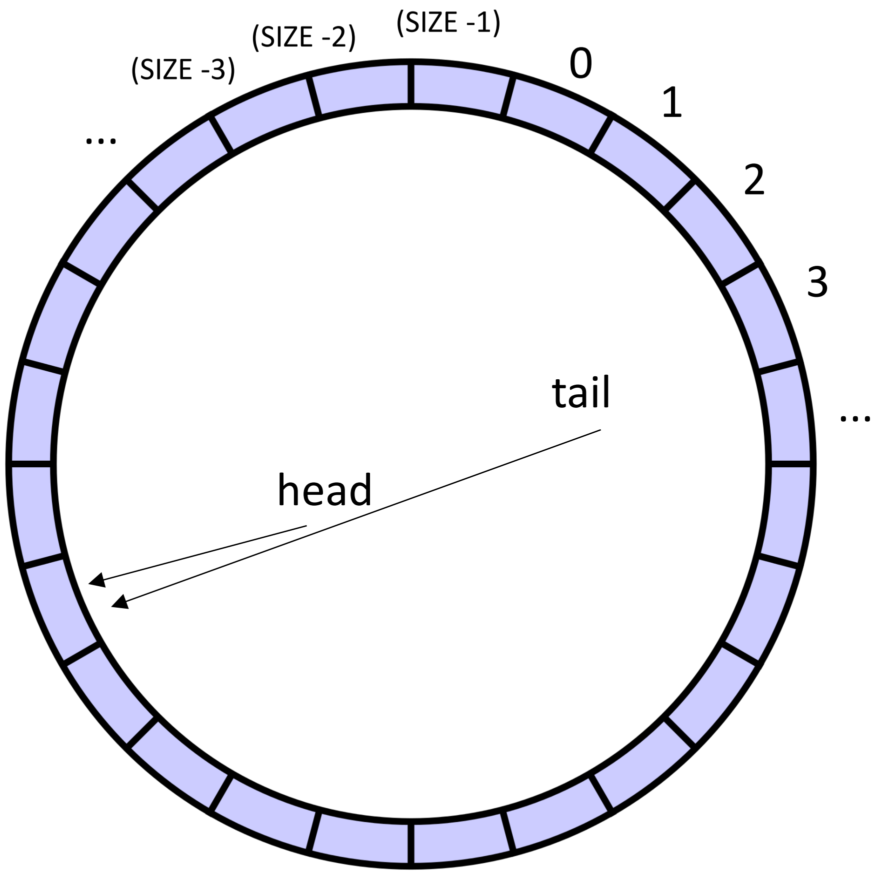


indexes will circulate in order and wrap around

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

Full queue is when head == tail?

but then how to tell full queue from empty?

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

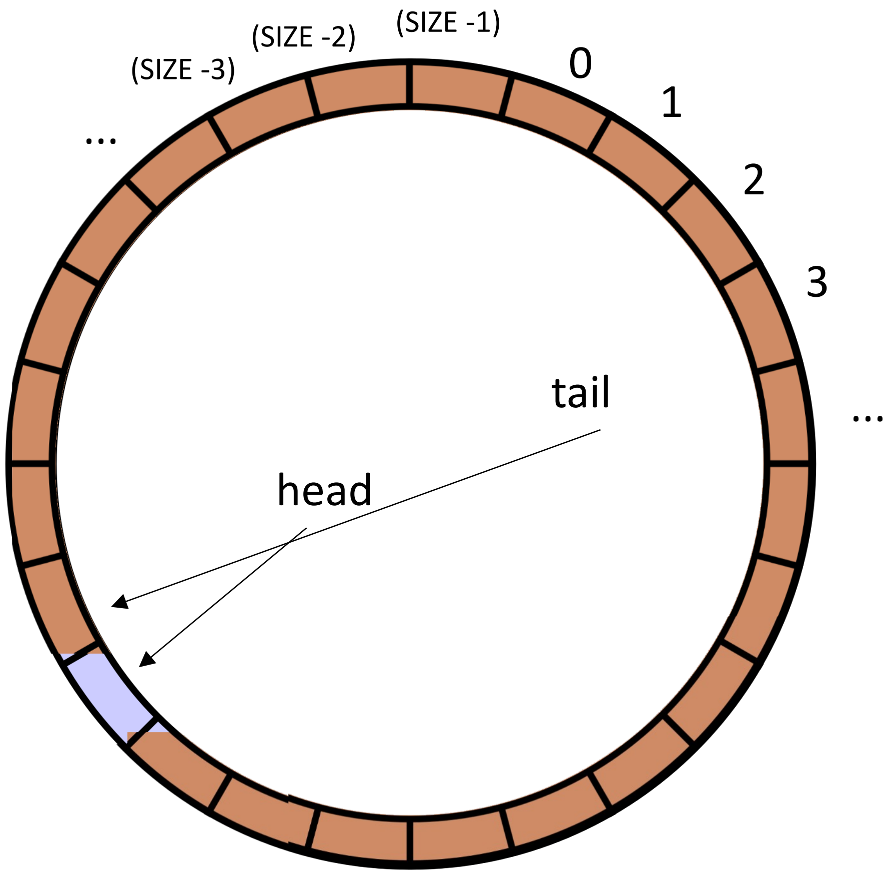
Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

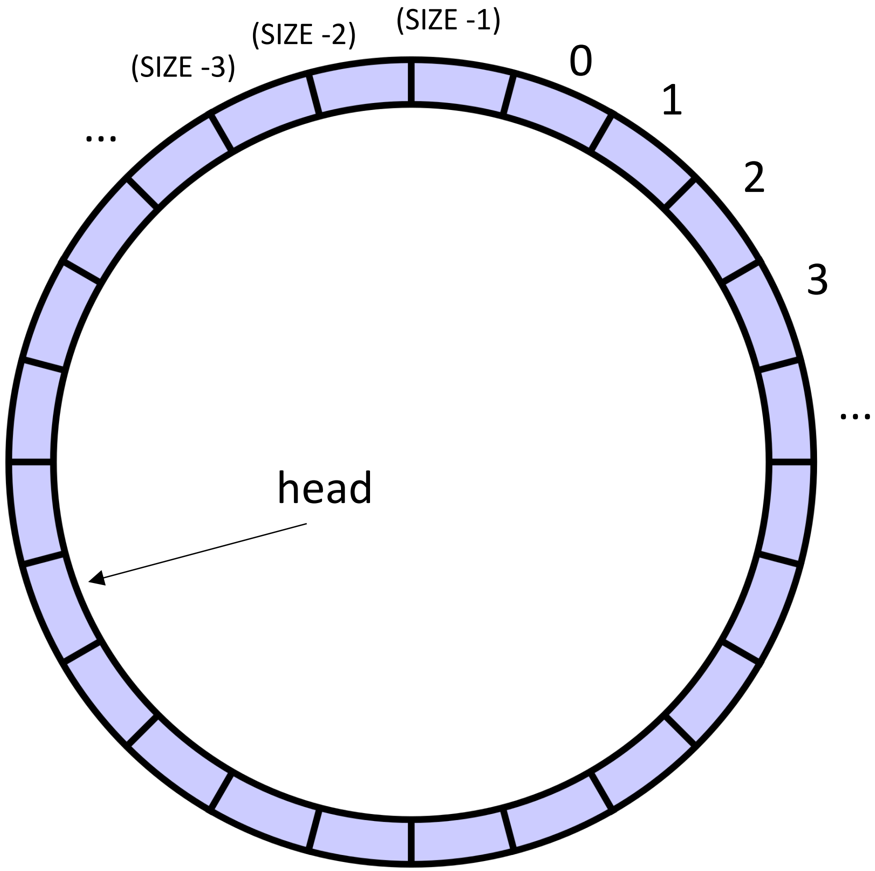
Full queue is when head + 1 == tail

conceptually it is a circle

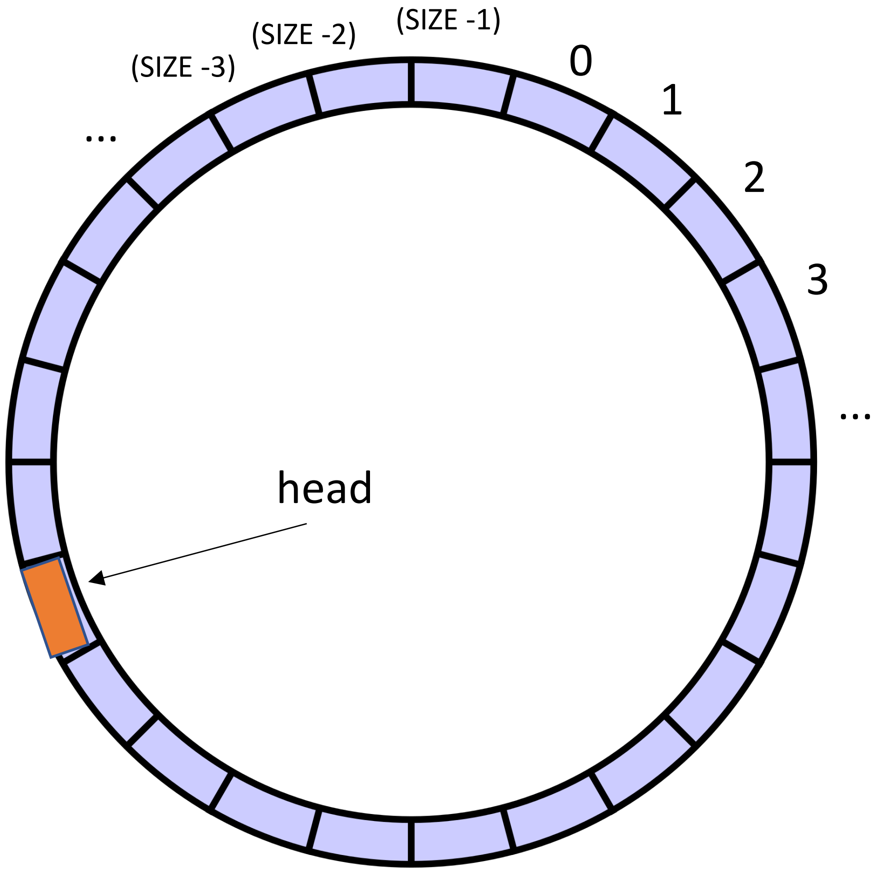


wasting one location, but its okay...

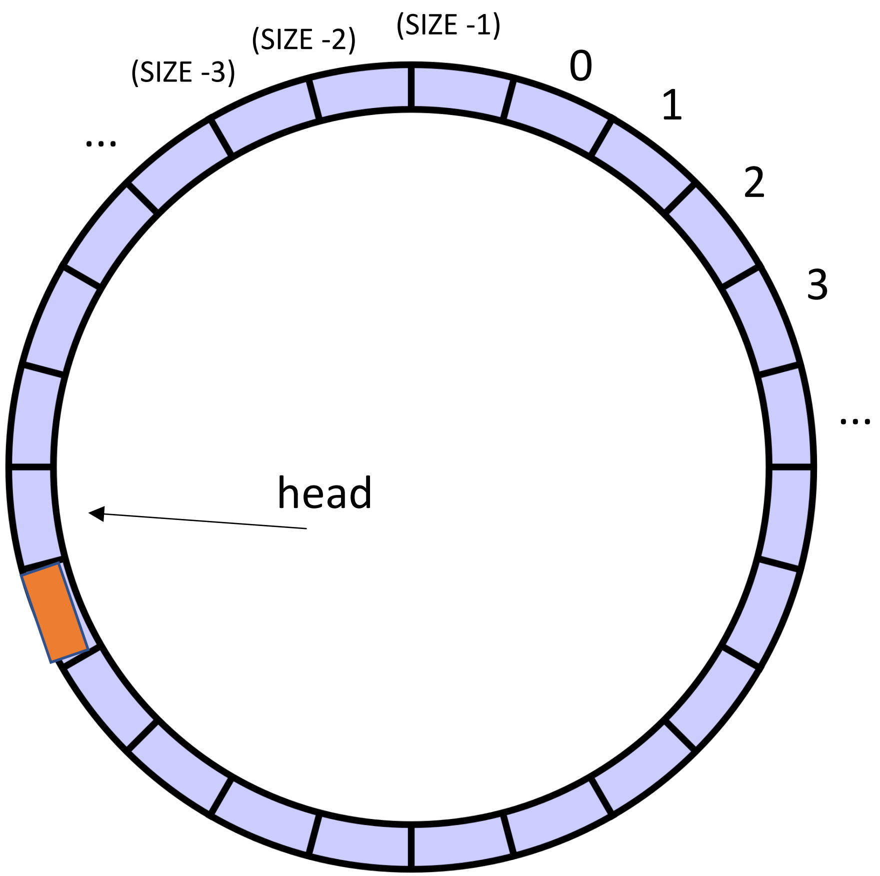
indexes will circulate in order and wrap around



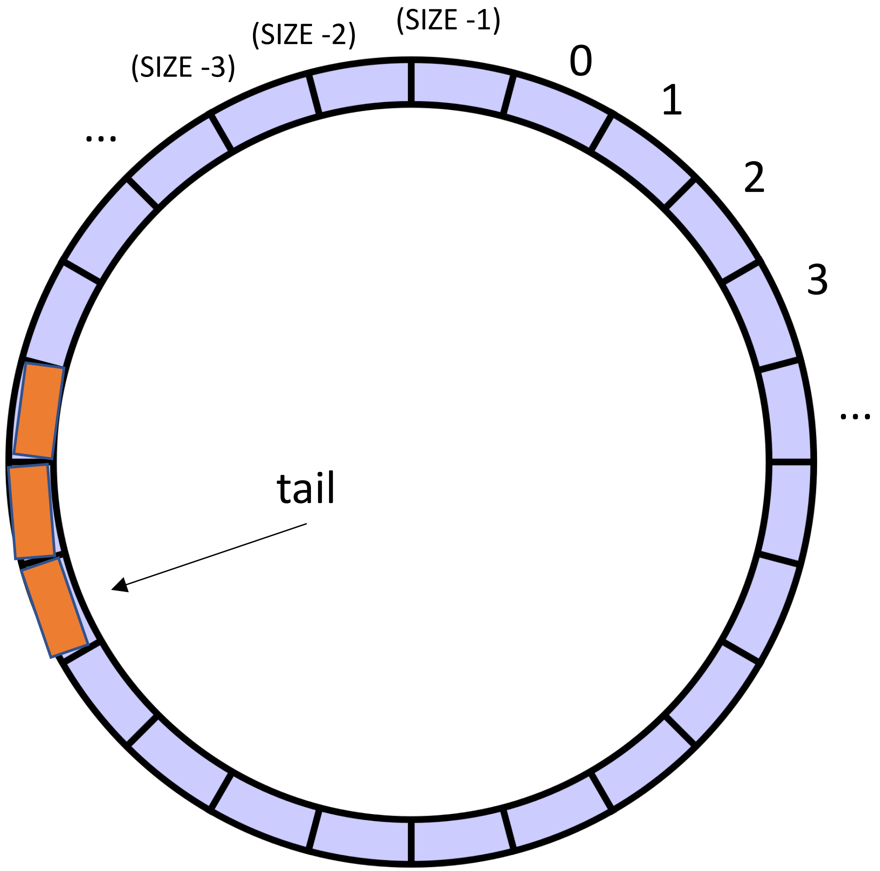

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```



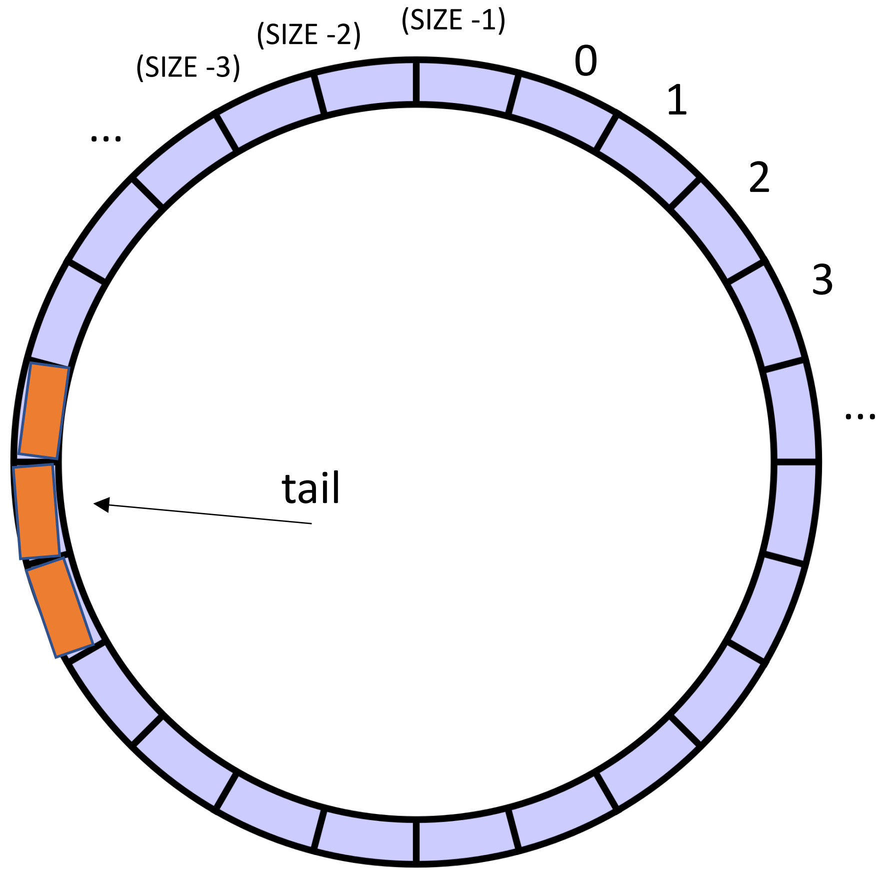

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```

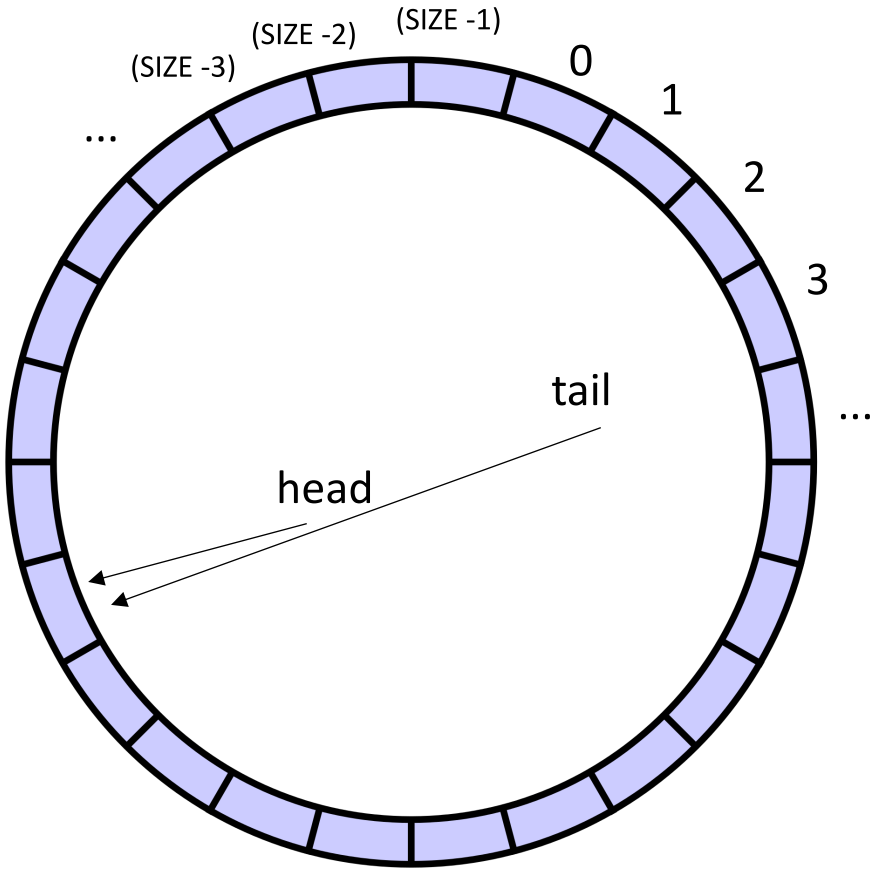



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```



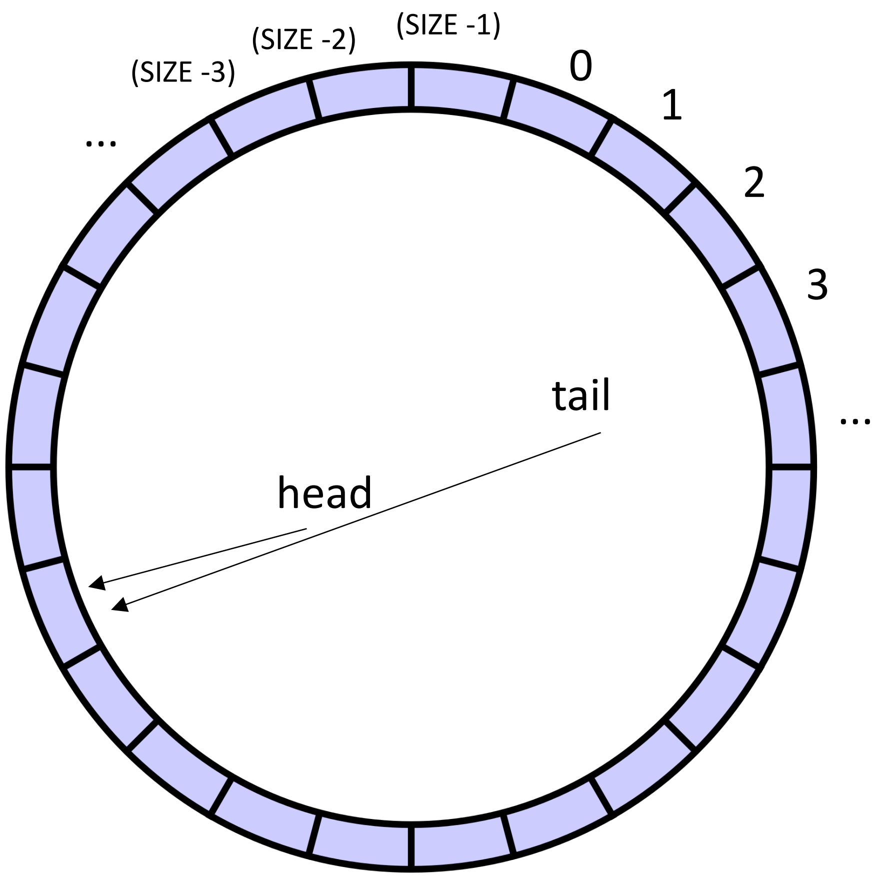

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```

This looks like the two threads don't even share head and tail! What is missing?

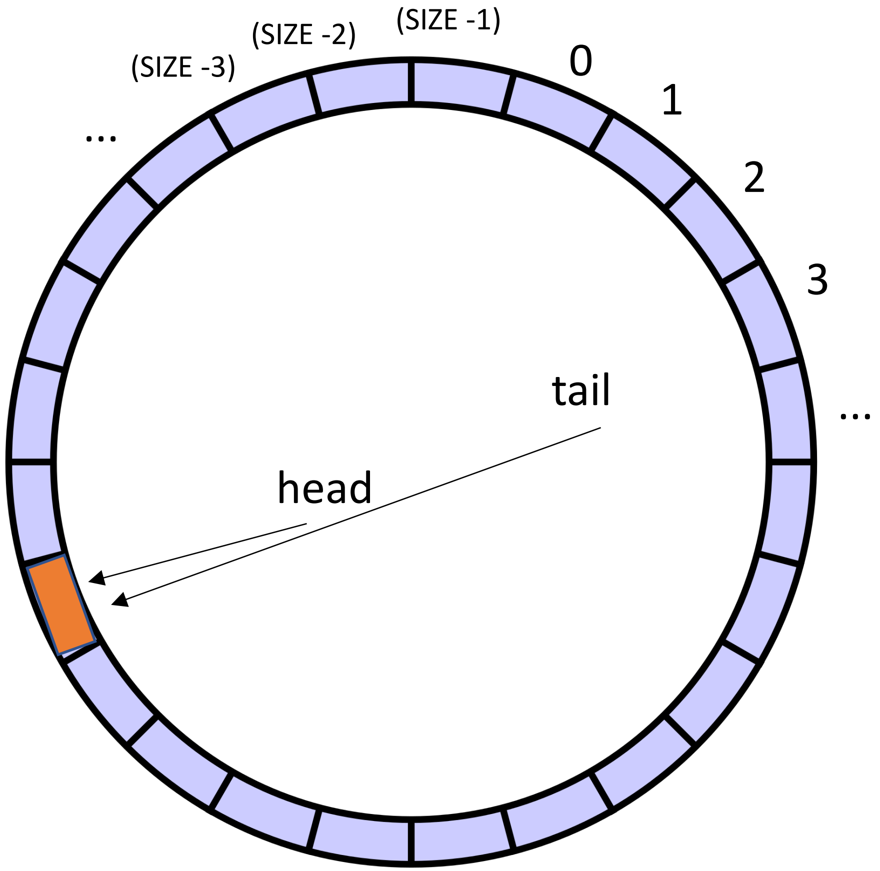



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```

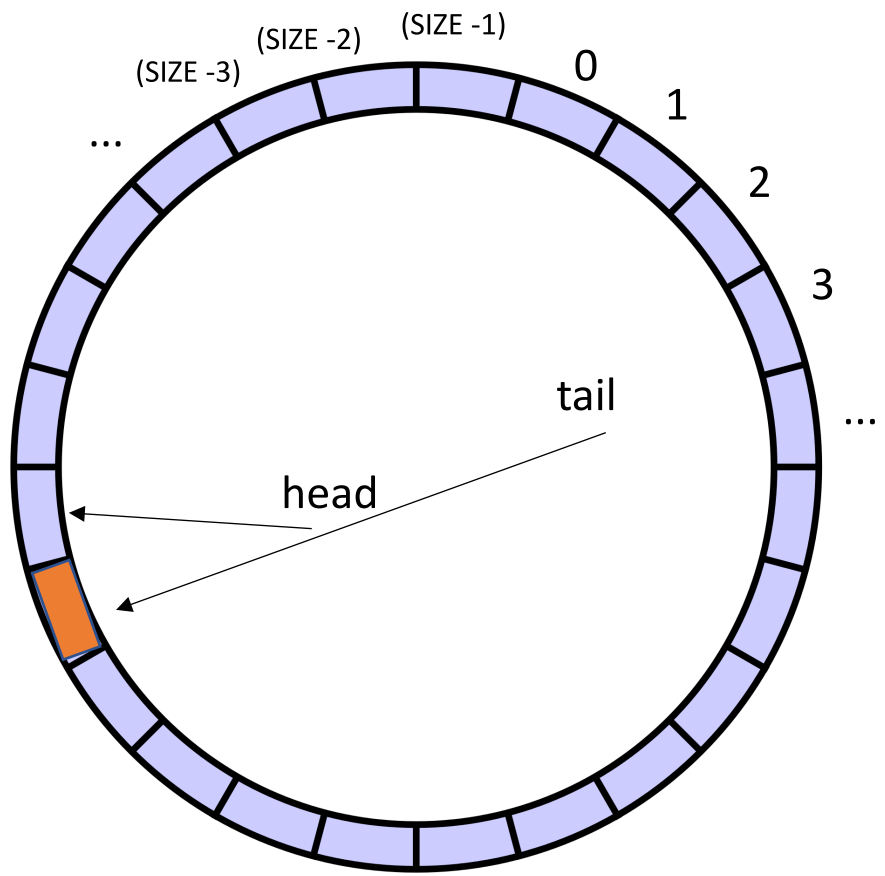
what happens if we try to dequeue here?



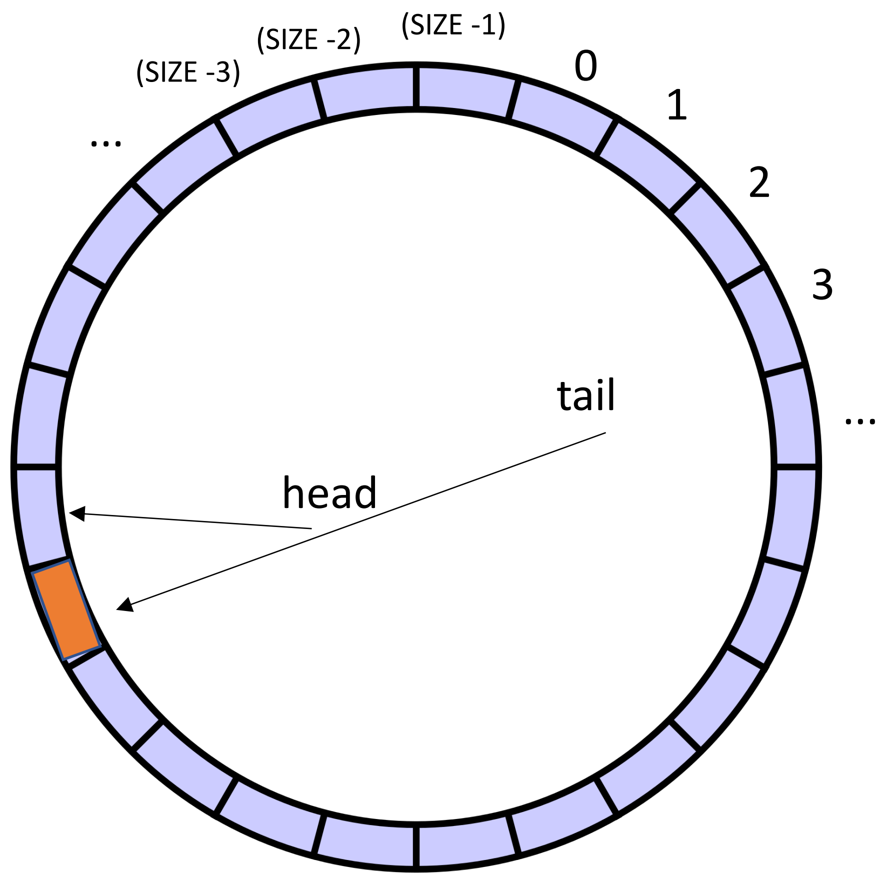

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



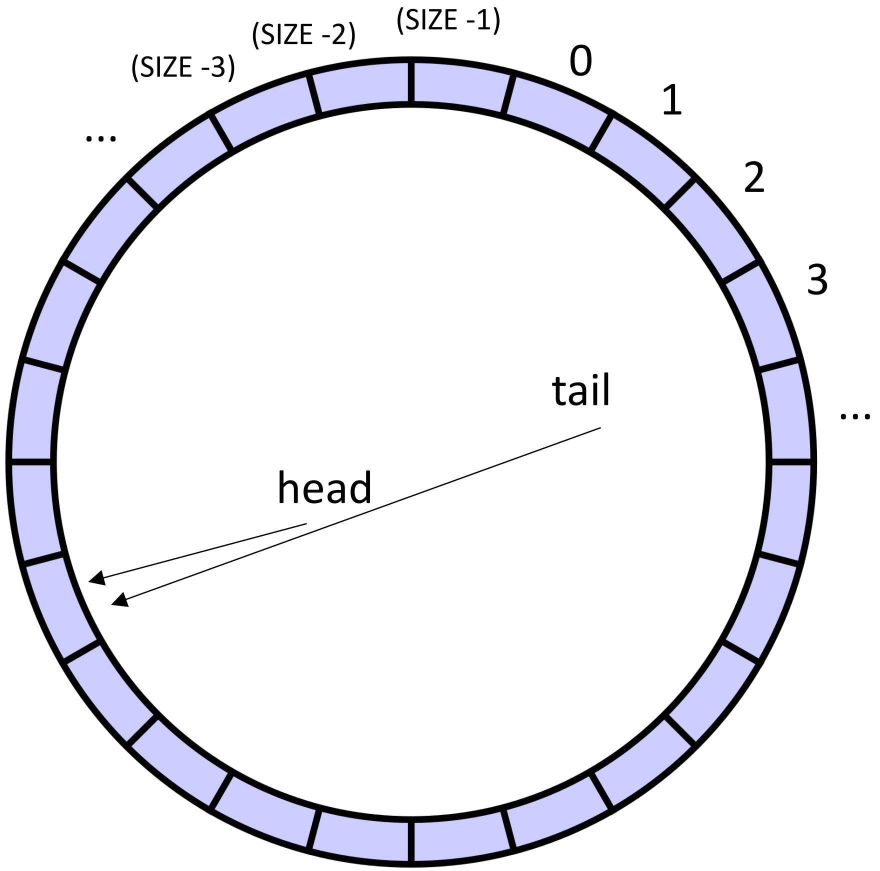

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



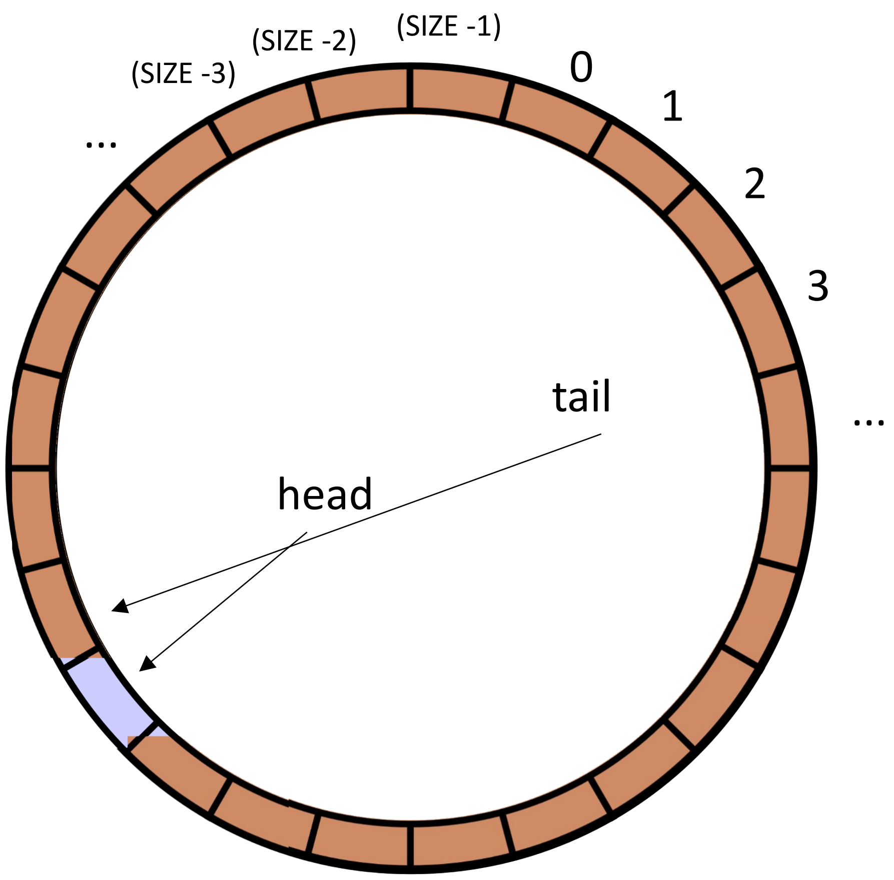

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



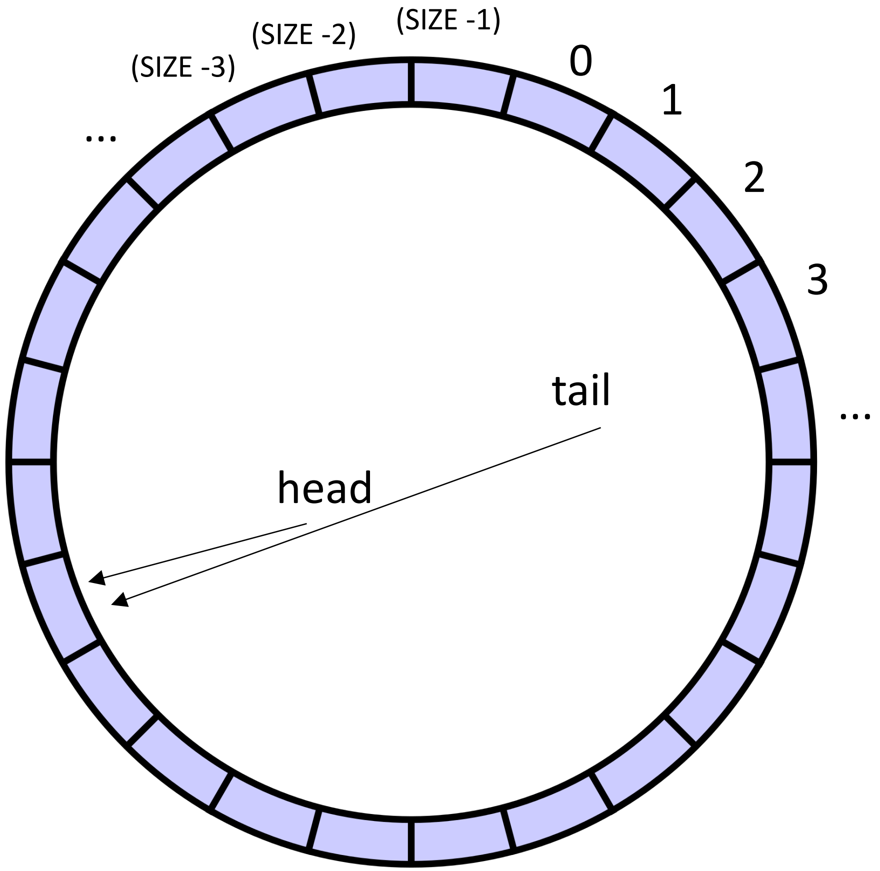

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

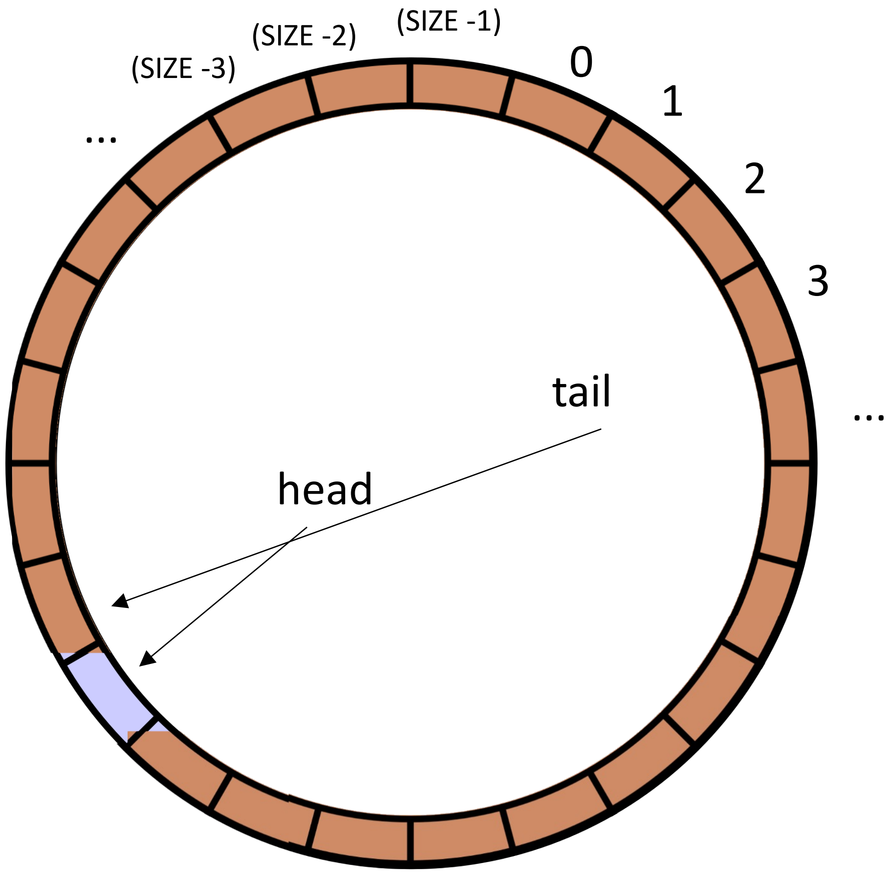
similarly for enqueue

but why can't we enqueue?




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

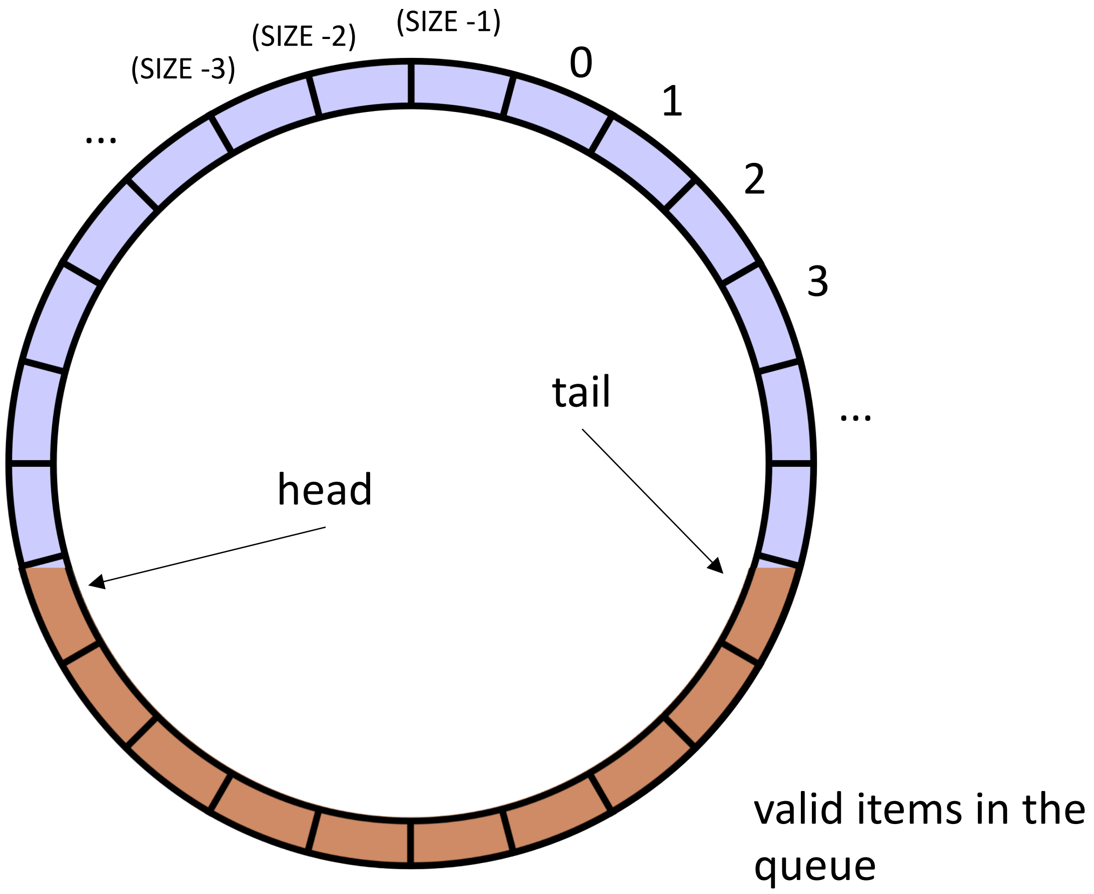
incrementing the head would make it empty!



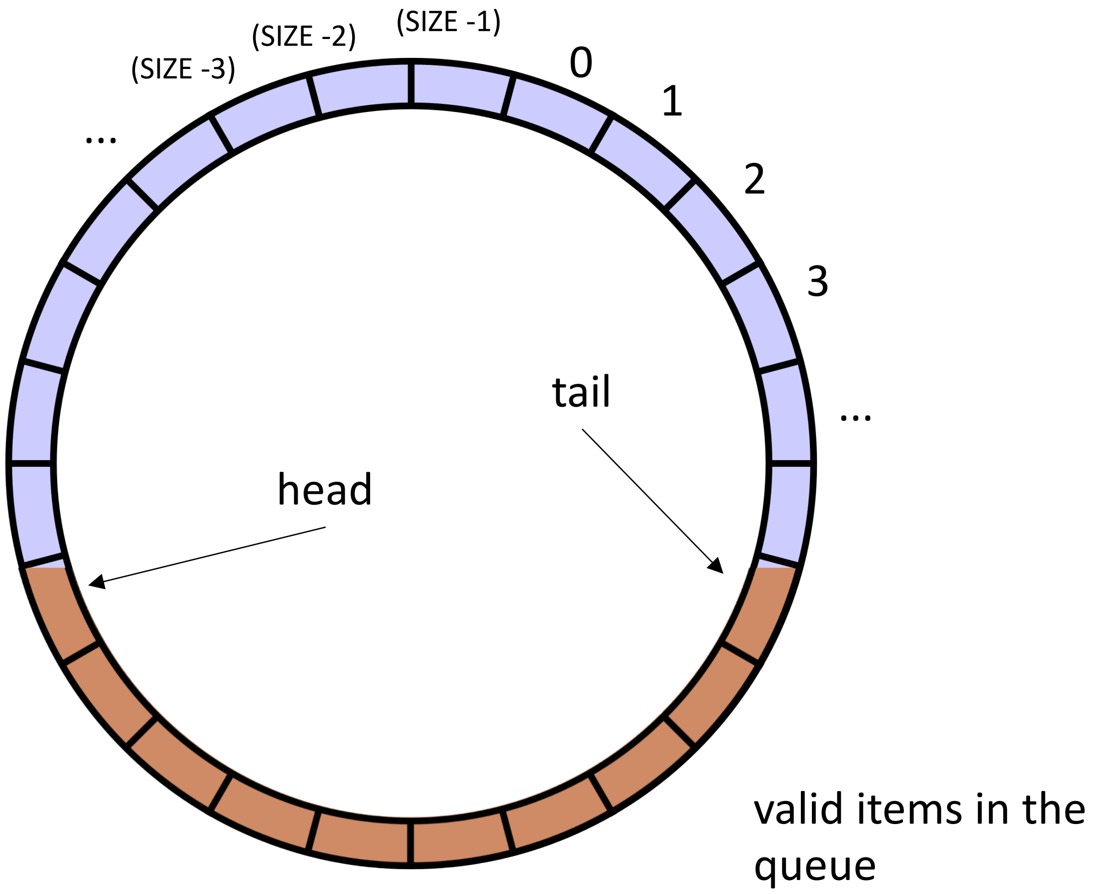

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

we need to wait for there to be room

Other questions:




```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



Other questions:

Do these need to be atomic RMWs?

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

Next week

- Work stealing and generalized concurrent objects
- Get HW 2 turned in today!
- HW 3 is out today. You can get started on Part 1
- Prepare for midterm on Monday