NC STATE UNIVERSITY

Queues

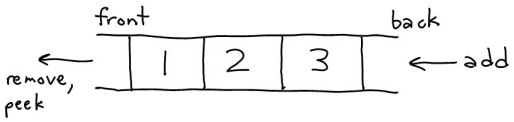
Reading:
Zybooks: 15.3 & 15.4

NC STATE UNIVERSITY

Queues

- **Queue**: retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.





- Basic queue operations:
  - **Add** (enqueue): Add an element to the back.
  - **Remove** (dequeue): Remove the front element.

Queues in Computer Science

- Operating Systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Programming with `Queues`

`add(`**`value`**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`element()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);        // front [42, -3, 17] back
System.out.println(q.remove());    // 42
```

- **IMPORTANT**: When constructing a queue you must use a new `LinkedList` object instead of a new `Queue` object. Why?

NC STATE UNIVERSITY

More About `java.lang.Queue`

- Provides insertion, extraction, and inspection operations
  - One set throws exceptions if precondition isn't met
  - One set returns a special value if cannot meet post condition

	Throws Exception	Returns Special Value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

8

NC STATE UNIVERSITY

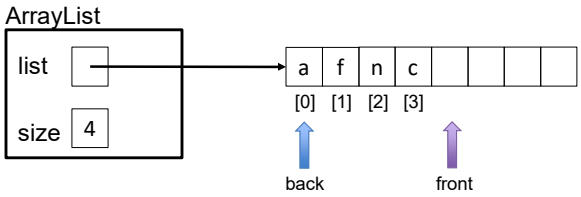
Queue Tradeoffs

- What are the tradeoffs in implementing a queue as an array vs a linked list?
  - Identify the front and the back of the queue

9

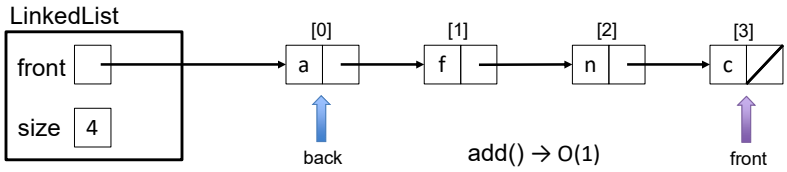
NC STATE UNIVERSITY

Implementation Tradeoffs
Front is size; Back is 0



add() → **O(n)**

remove() → O(1)



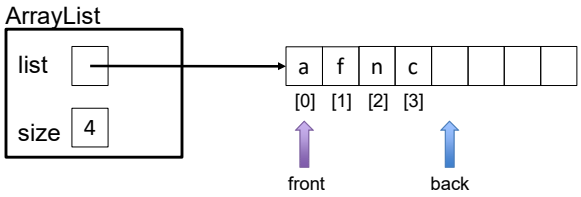
add() → O(1)

remove() → **O(n)**

10

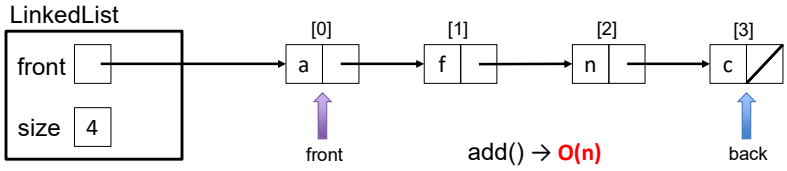
NC STATE UNIVERSITY

Implementation Tradeoffs
Front is 0; Back is size



add() → O(1)

remove() → **O(n)**



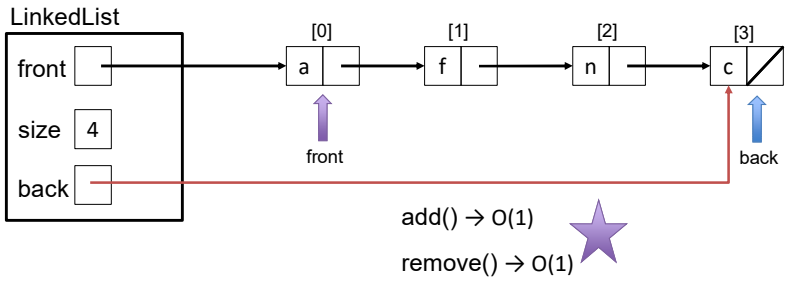
add() → **O(n)**

remove() → O(1)

11

NC STATE UNIVERSITY

Implementation Tradeoffs
Front is 0; Back is size; Additional Reference



The additional reference does NOT work when front is size and back is 0. To remove, we need a reference to the node *before* the one we want to remove. We can't traverse the list backwards (not the way it's currently implemented), so we can't move the back reference to prior nodes.

NC STATE UNIVERSITY

Queue Idioms

- As with stacks, must pull contents out of queue to view them.

```
while (!q.isEmpty()) {
    do something with q.remove();
}
```

- Another idiom: Examining each element exactly once.

```
int size = q.size();
for (int i = 0; i < size; i++) {
    do something with q.remove();
    (including possibly re-adding it to the queue)
}
```

  - Why do we need the `size` variable?

Exercise

What is the output produced when the following method is passed various queues:

```
public static void mystery3(Queue<Integer> q) {
   int size = q.size();
   for (int i = 0; i < size; i++) {
      int n = q.remove();
      if (n > 0) {
         q.add(-n);
      }
   }
   System.out.println(q);
}
```


Mixing stacks and queues

- We often mix stacks and queues to achieve certain effects.
  - Example: Reverse the order of the elements of a queue.

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(1);
q.add(2);
q.add(3);                    // [1, 2, 3]

Stack<Integer> s = new Stack<Integer>();
while (!q.isEmpty()) {       // Q -> S
    s.push(q.remove());
}
while (!s.isEmpty()) {       // S -> Q
    q.add(s.pop());
}
System.out.println(q);       // [3, 2, 1]
```

Delegation Example - `Queue`

- Consider `java.util.Queue`
- How many methods does the `Queue` interface have?
  - `Queue` defines 6 methods
  - `Queue` inherits 19 methods from `Collection`
  - `Queue` inherits 1 method from `Iterable`

- What is the main functionality of a `Queue`?
- Some of the methods in `java.util.Queue` allow for functionality that a "pure" `Queue` wouldn't allow
  - Remove an arbitrary element

A "Pure" `Queue`

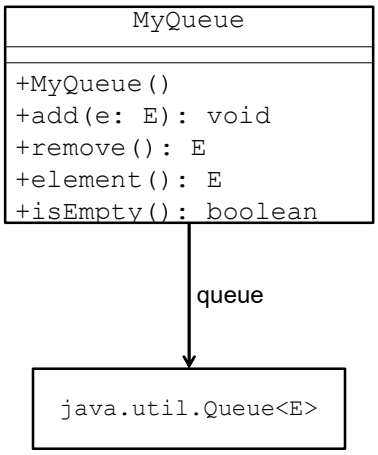
- Why would we want a "pure" `Queue`?
  - One that only allows for `add()`, `remove()`, `peek()`, and `isEmpty()`
- How could we get a "pure" `Queue`?
  - Implement `Queue` and override methods we don't want with empty methods / throw exception
  - Create our own `Queue`
  - Use delegation (via an association relationship)
    - Write a class that has an instance of the `Queue (as LinkedList)`
    - Provide the interface you want, and then pass those tasks on to your `Queue` instance

NC STATE UNIVERSITY

Delegation Example

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

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



NC STATE UNIVERSITY

Exercise

- From Sedgewick's and Wayne's *Introduction to Programming in Java*
- This is based on the Josephus problem from antiquity. There are n people in a population and there's only 1 item of something everyone wants (like chocolate cake). They arrange themselves in a circle (as positions numbered from 0 to N-1) and proceed around the circle, removing every Mth person from consideration until only one person is left. Legend has it that Josephus figured out where to sit to get the chocolate cake. Write a `Queue` client `Josephus` that takes N and M from the command line and prints out the order in which people are removed from consideration (and thus would show Josephus where to sit in the circle).

```
% java Josephus 7 2
1 3 5 0 4 2 6
```