Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Lists II: Stacks, Queues, and Deques

""

CS2013: Programming with Data Structures

Stacks

Stack

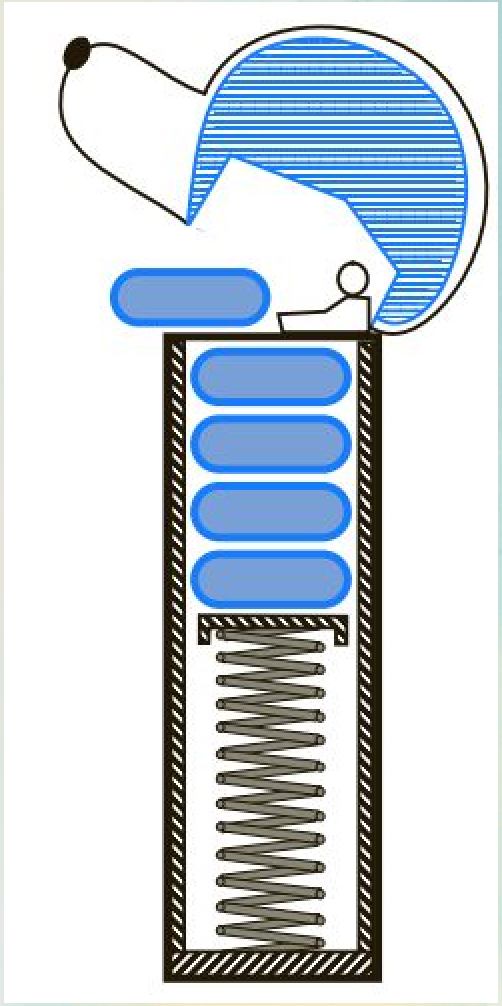
- ***stack***: a list-type data structure where items are inserted and removed according to the ***last-in, first-out (LIFO)*** principle.

- Items are inserted on the top of the stack at any time, but the only item you can access / retrieve is the item currently at the top of the stack.

- The name ***stack*** comes from the comparison to a plate dispenser in a cafeteria.
  - When you need a plate you take the top plate and the spring underneath pushes all of the plates to the top ***(pop)***.
  - When you want to put back a plate you ***push*** a plate onto the top and push the spring down to make room.

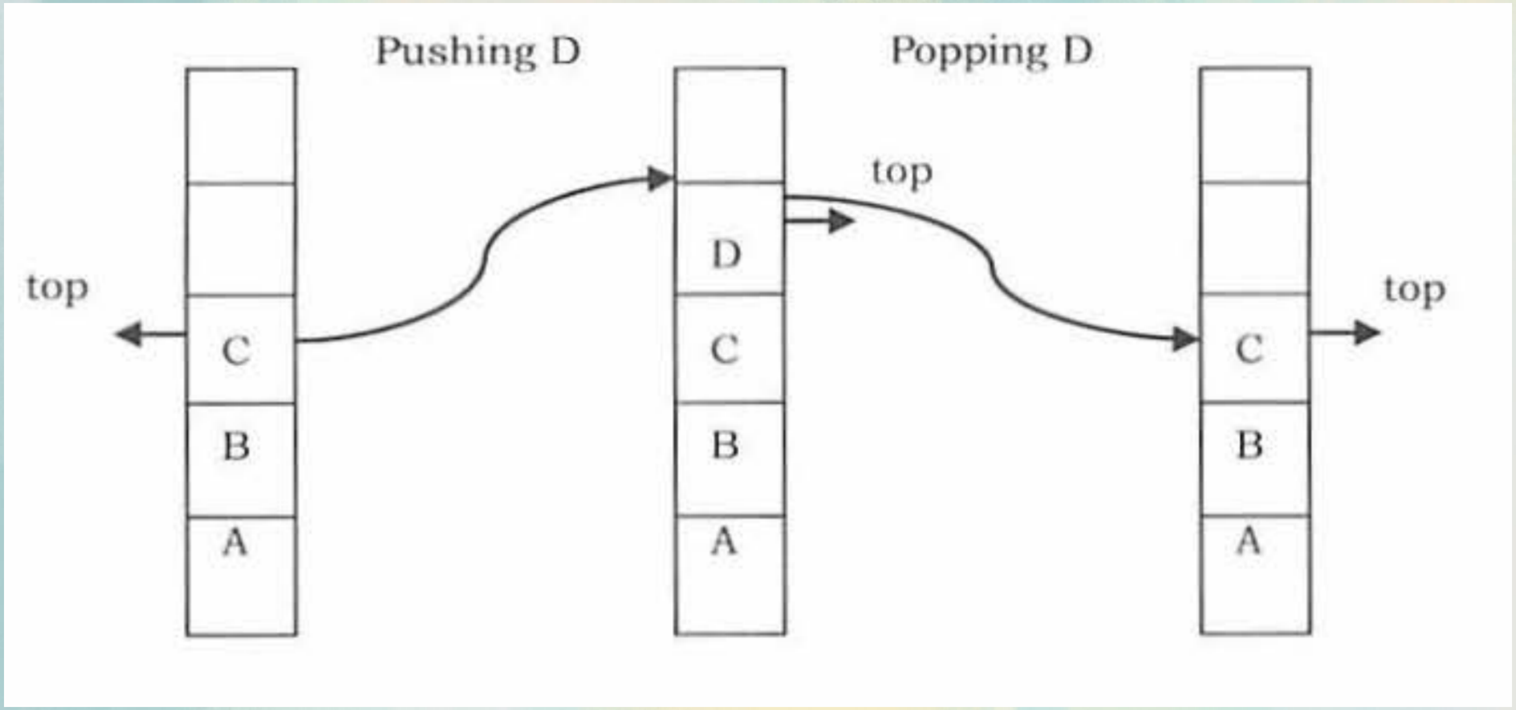
Stack

- A stack is like a PEZ candy dispenser. The candies are stored in a spring loaded dispenser and you load the candies in the top and the candies are "popped" out of the top once you want to eat one.



Stack

- Stacks have two main operations:
  - **push()**: adding an element to the top of the Stack.
  - **pop()**: removing and returning the element on the top of the Stack.
- ***underflow***: trying to pop something from an empty Stack.
- ***overflow***: trying to push something onto a Stack that is full.



Examples of Using Stacks

- Example 1: Web browsers store the addresses of recently visited sites on a stack. Each time you visit a new site the new address is "pushed" onto the stack. Every time you hit the back button the top address is "popped" off of the stack and the new top address is loaded into the browser.
- Example 2: Text editors that have an "undo" button usually keeps track of changes using a stack.
- Example 3: Balancing matching symbols such as pairs of ( ), [ ], { }, etc.
- Example 4: Checking matching HTML tags.
- Example 5: Function calls are organized using a stack, the currently running function is the function at the top of the stack.

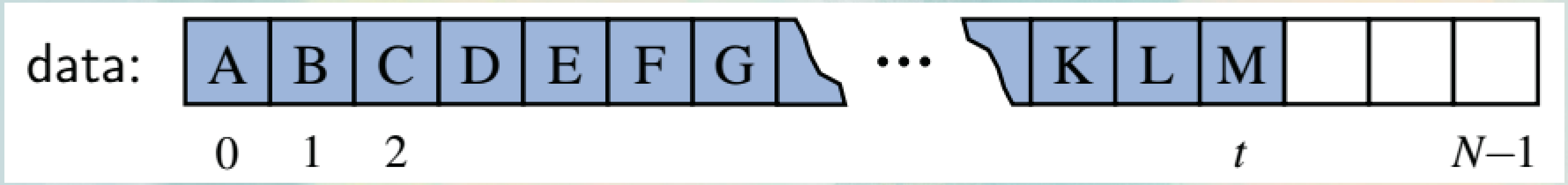
The Stack ADT

- What would be the Big O runtimes of these operations?

`Stack()`	Constructor to create an instance of a Stack.
`push(item)`	Adds the item to the top of the stack. Should throw a StackOverflowException if the Stack is full.
`pop()`	Removes and returns the top element from the stack. Returns null if the Stack is empty.
`peek()`	Return the top element of the Stack without removing it. Returns null if the Stack is empty.
`size()`	Returns the number of elements in the Stack.
`isEmpty()`	Returns a boolean indicating whether the Stack is empty.

Stack Implementation - Array

- A simple implementation would be to use an array to store the elements of the stack.
  - Let's call our array **`data`**.



- The "bottom" of the stack is at data[0] and the "top" of the stack is at data[t] where t < data.length.

```
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000;  // default array capacity
  private E[ ] data;                      // generic array used for storage
  private int t = −1;                     // index of the top element in stack
  public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
  public ArrayStack(int capacity) {       // constructs stack with given capacity
    data = (E[ ]) new Object[capacity];   // safe cast; compiler may give warning
  }
  public int size( ) { return (t + 1); }
  public boolean isEmpty( ) { return (t == −1); }
  public void push(E e) throws IllegalStateException {
    if (size( ) == data.length) throw new IllegalStateException("Stack is full");
    data[++t] = e;                        // increment t before storing new item
  }
  public E top( ) {
    if (isEmpty( )) return null;
    return data[t];
  }
  public E pop( ) {
    if (isEmpty( )) return null;
    E answer = data[t];
    data[t] = null;
    t−−;
    return answer;
  }
}
```

Null in the pop() Method

- Take a careful look at line 22 of the pop() method.
- Why do we set `data[t] = null` ?

Array-Based Implementation Drawbacks

- The size of the stack is limited by the size of the array.
  - If you choose a size that is too small you will run out of space quickly.
  - If you choose a size that is too large, you will have wasted space.

- You could implement a Stack using a Dynamic List, but what would be the drawback of this?

Stack Implementation - Singly-Linked List

- A better implementation is to use a Singly-Linked List.
  - What are some advantages?

- Should we insert elements to the beginning of the list or to the end of the list?

Stack Implementation - Singly-Linked List

- Assuming that a Singly-Linked List class already exists, how can we use it to implement our Stack?

- One way would be to make our Stack class a subclass of the Singly-Linked List class.
  - This is not the best way, why?

- A better way would be to use the ***adapter*** design pattern.
  - A ***design pattern***, is a general reusable solution to a commonly occurring problem within a given context in software design.
  - For further reading on design patterns check out the famous "Gang of Four" book; *Design Patterns: Elements of Reusable Object-Oriented Software*.

Stack Implementation - Singly-Linked List

- One way to apply the ***adapter*** pattern is to create a new class that has an instance of the class we want to ***adapt*** as a datafield.

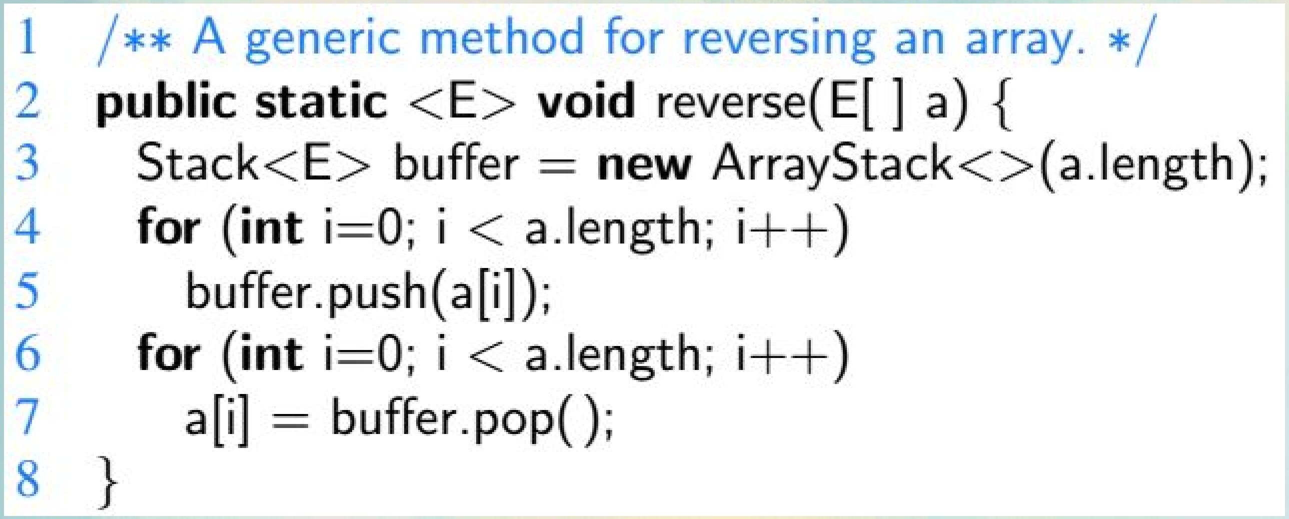
- With our Stack, we can implement the adapter pattern by defining a new Stack class that has a linked list as a private data field.

- By using this design, our class has full access to all of the methods of the LinkedList class, but we can adapt those methods to fit the requirements of a Stack.

```java
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedStack() { }                         // new stack relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(E element) { list.addFirst(element); }
  public E top() { return list.first(); }
  public E pop() { return list.removeFirst(); }
}
```

Stack Example - Reversing a List

- A Stack can be very easily used to reverse the elements of a List.
  - Iterate over the list and push the items onto the stack.
  - Iterate over the list again this time popping each value off of the stack and assigning them back to the list starting from the beginning.



```
/** A generic method for reversing an array. */
public static <E> void reverse(E[ ] a) {
  Stack<E> buffer = new ArrayStack<>(a.length);
  for (int i=0; i < a.length; i++)
    buffer.push(a[i]);
  for (int i=0; i < a.length; i++)
    a[i] = buffer.pop( );
}
```

Stack Example - Parentheses Matching

- Stacks can very easily be used to check for matching pairs of grouping symbols:
  - i.e. if you have an expression [(5 + x) - (y + z)], are all grouping symbols correctly paired?

- Other examples:
  - Correct: ( ) (( )) { ([( )]) }
  - Correct: ((( ) (( )) { ([( )])]))
  - Incorrect:) (( )) { ([( )]) }
  - Incorrect: ({ [ ]) }
  - Incorrect: (

- Can you come up with an algorithm to do this?

```java
/** Tests if delimiters in the given expression are properly matched. */
public static boolean isMatched(String expression) {
  final String opening   = "({[";              // opening delimiters
  final String closing   = ")}]";              // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>();
  for (char c : expression.toCharArray()) {
    if (opening.indexOf(c) != −1)              // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) {       // this is a right delimiter
      if (buffer.isEmpty())                    // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
        return false;                          // mismatched delimiter
    }
  }
  return buffer.isEmpty();                     // were all opening delimiters matched?
}
```

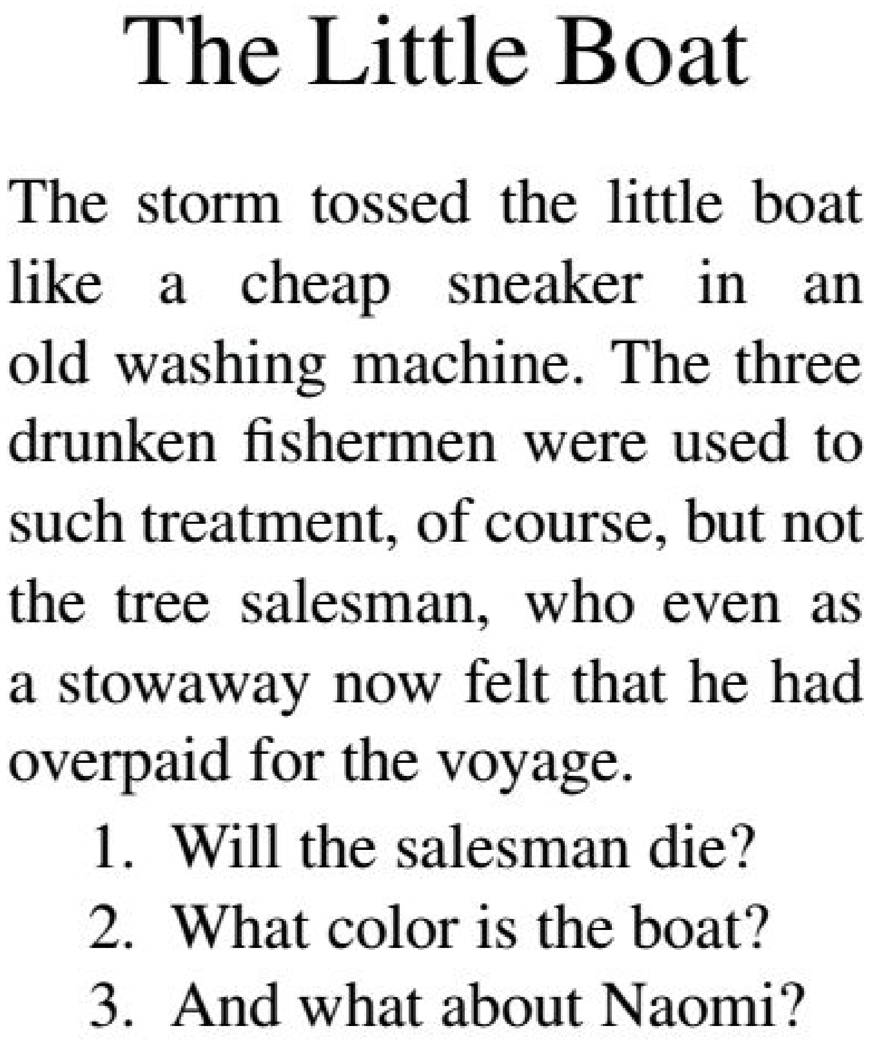
Stack Example - Matching HTML Tags

- Another real world application of a Stack is when matching the tags of markup languages such as HTML or XML.

- The same logic that you use to match grouping symbols can be applied to matching markup tags.

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine.  The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)



The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

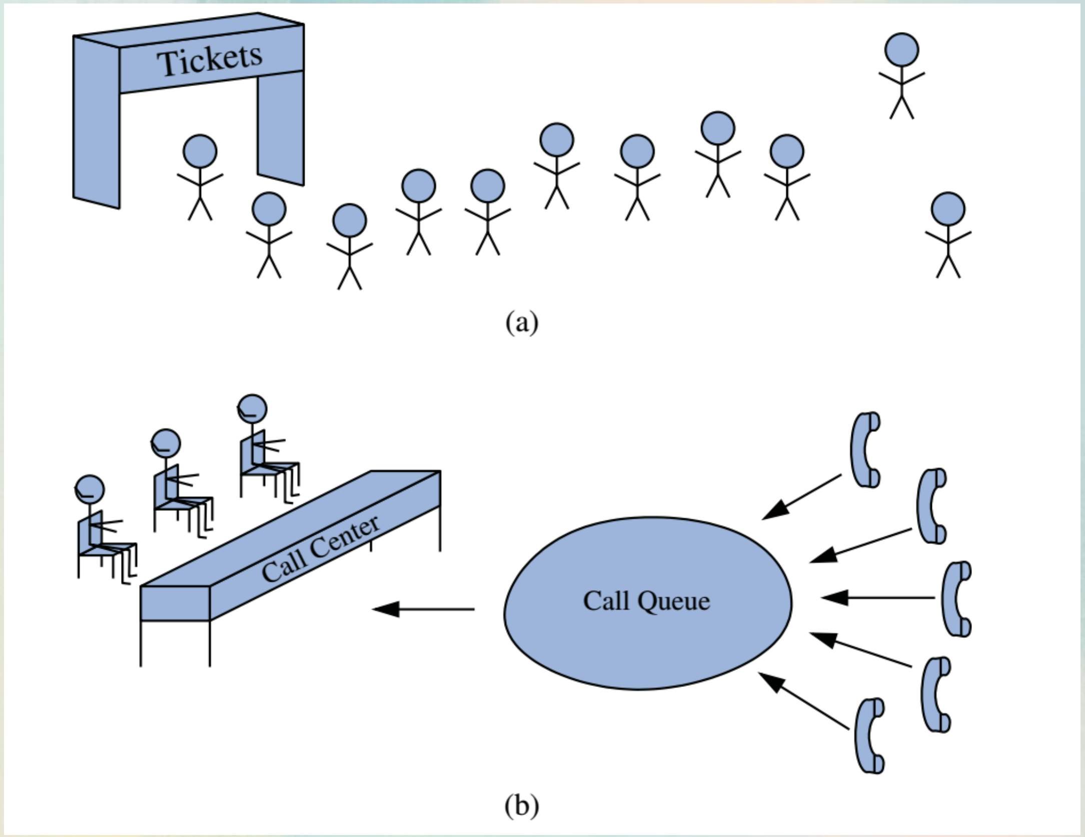
(b)

```java
/** Tests if every opening tag has a matching closing tag in HTML string. */
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer = new LinkedStack<>();
  int j = html.indexOf('<');                          // find first '<' character (if any)
  while (j != −1) {
    int k = html.indexOf('>', j+1);                   // find next '>' character
    if (k == −1)
      return false;                                   // invalid tag
    String tag = html.substring(j+1, k);              // strip away < >
    if (!tag.startsWith("/"))                         // this is an opening tag
      buffer.push(tag);
    else {                                            // this is a closing tag
      if (buffer.isEmpty())
        return false;                                 // no tag to match
      if (!tag.substring(1).equals(buffer.pop()))
        return false;                                 // mismatched tag
    }
    j = html.indexOf('<', k+1);                       // find next '<' character (if any)
  }
  return buffer.isEmpty();                            // were all opening tags matched?
}
```

Queues

Queue

- ***queue***: a collection of objects that are inserted and removed according to the ***first-in, first-out (FIFO)*** principle.
  - Elements can be inserted at anytime, but only the element that has been in the queue the longest can be removed next.
  - Usually elements enter the queue at the back, and leave the queue from the front.



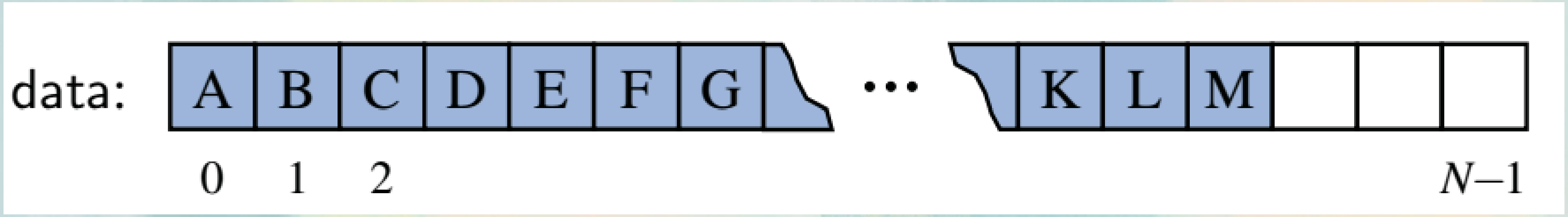
The Queue ADT

- What would be the Big O runtimes of these operations?

`Queue():`	Constructor to create an instance of a Queue.
`enqueue(item):`	Adds element *item* to the back of the queue.
`dequeue():`	Removes and returns the first element from the queue (or null if the queue is empty).
`first():`	Returns the first element of the queue, without removing it (or null if the queue is empty).
`size():`	Returns the number of elements in the queue.
`isEmpty():`	Returns a boolean indicating whether the queue is empty.

Queue Implementation - Array

- Assume elements are stored in an array such that the first element is at index 0 and and the last at index size - 1.

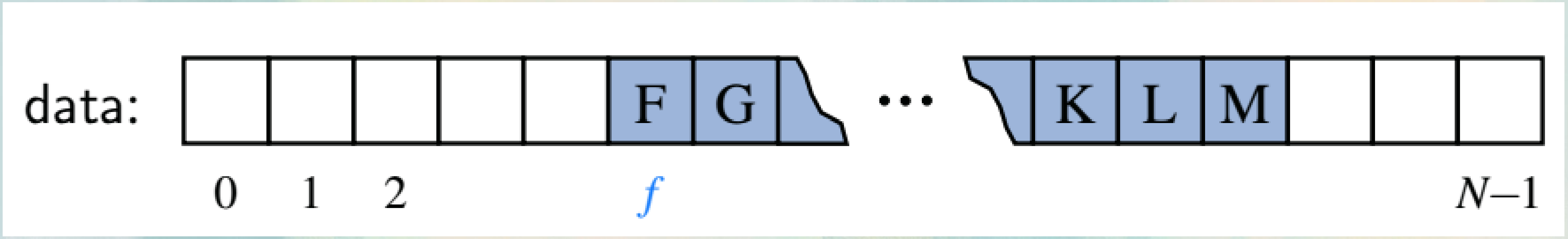


- With this implementation, enqueue() is very easy, we just keep track of the last index and keep adding to the end.

- dequeue() is a bit more challenging to implement:
  - you could just left shift every time you want to remove an item, but this means dequeue() becomes O(n).

Queue Implementation - Array

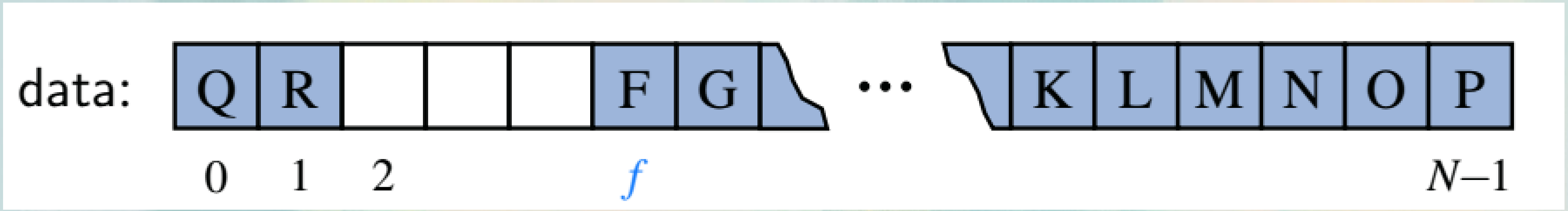
- A more efficient implementation would be to simply replace the removed item with a value of null and maintain a variable *f* to keep track of the index of the element at the front of the queue.



- This will work until *f* is travels all the way to the right in which case we have an exceptional case to deal with.

Queue Implementation - Circular Array

- An even better approach would be to use a circular array, that is an array where both the front and back indexes drift rightward and wrap around the end of the array when necessary.



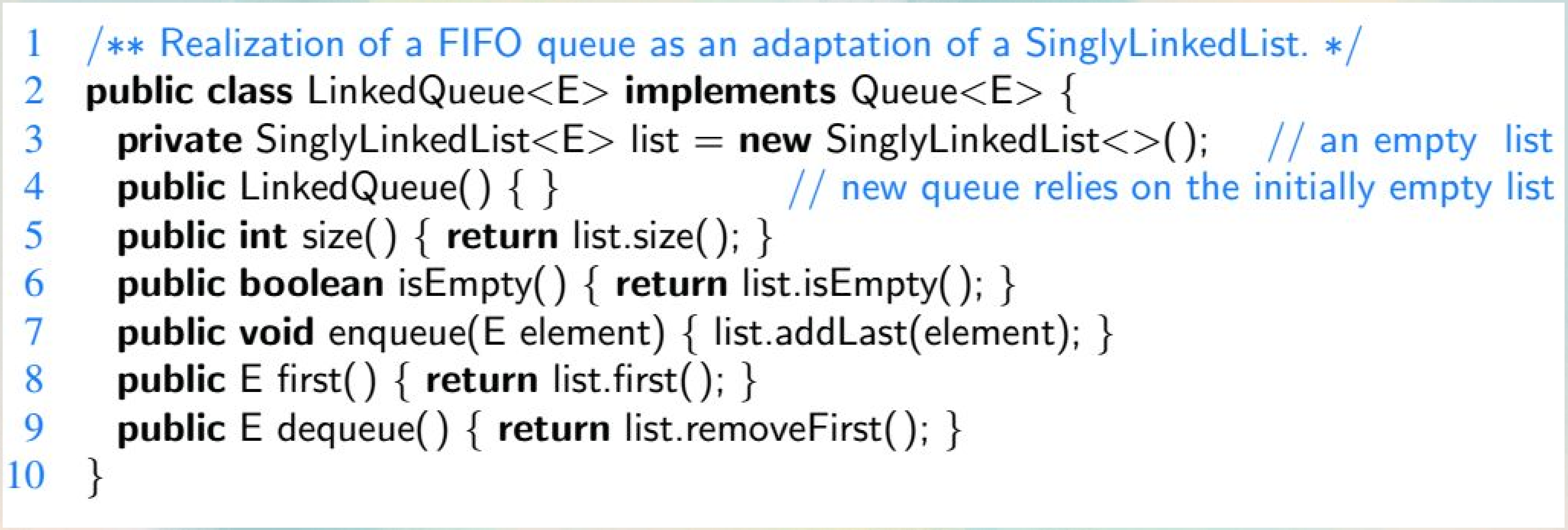
- The mod operator (%) works great for implementing a circular array.
  - when you compute f, we can use the formula:
    f = (f + 1) % N where N is size of the underlying array.

Queue - enqueue() and dequeue()

```
/** Inserts an element at the rear of the queue. */
public void enqueue(E e) throws IllegalStateException {
  if (sz == data.length) throw new IllegalStateException("Queue is full");
  int avail = (f + sz) % data.length;        // use modular arithmetic
  data[avail] = e;
  sz++;
}
```

```
/** Removes and returns the first element of the queue (null if empty). */
public E dequeue() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null;                            // dereference to help garbage collection
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```

Queue Implementation - Singly Linked List



```java
/** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty  list
  public LinkedQueue() { }                    // new queue relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
}
```

Deques

Double-Ended Queues

- double-ended queue (deque): (pronounced "deck"), a queue-like data structure that supports insertion and deletion at both the front and back of the queue.

- Example: Restaurant waitlist system. A person might be removed from the queue only to find a table is not available, so they need to be put back in the front of the queue. A person at the end may get impatient and leave, so they need to be removed from the end of the queue.

The Deque ADT

- What would be the Big O runtimes of these operations?

`Deque():`	Constructor to create an instance of a Deque.
`addFirst(item):`	Insert a new item at the front of the deque.
`addLast(item):`	Insert a new item at the back of the deque.
`removeFirst():`	Remove and return the first element.
`removeLast():`	Remove and return the last element.
`first():`	Return the first element without removing it.
`last():`	Return the last element without removing it.
`size():`	Return the number of elements.
`isEmpty():`	Return a boolean indicating whether the deque is empty.

Deque Implementations

- Circular Array:
  - Removing the first element means the front index is advanced in a circular fashion:
    - f = (f + 1) % N
  - When an element is inserted to the front, we need to make sure the front index does not go negative but wraps around to the back.
    - f = (f - 1 + N) % N, adding an additional N before the mod operation keeps the result positive.

- Doubly Linked List:
  - this is the best type of linked list to use since a deque can manipulate elements at both ends of itself.