

CS151 Intro to Data Structures

Stacks

JUnit

Queues

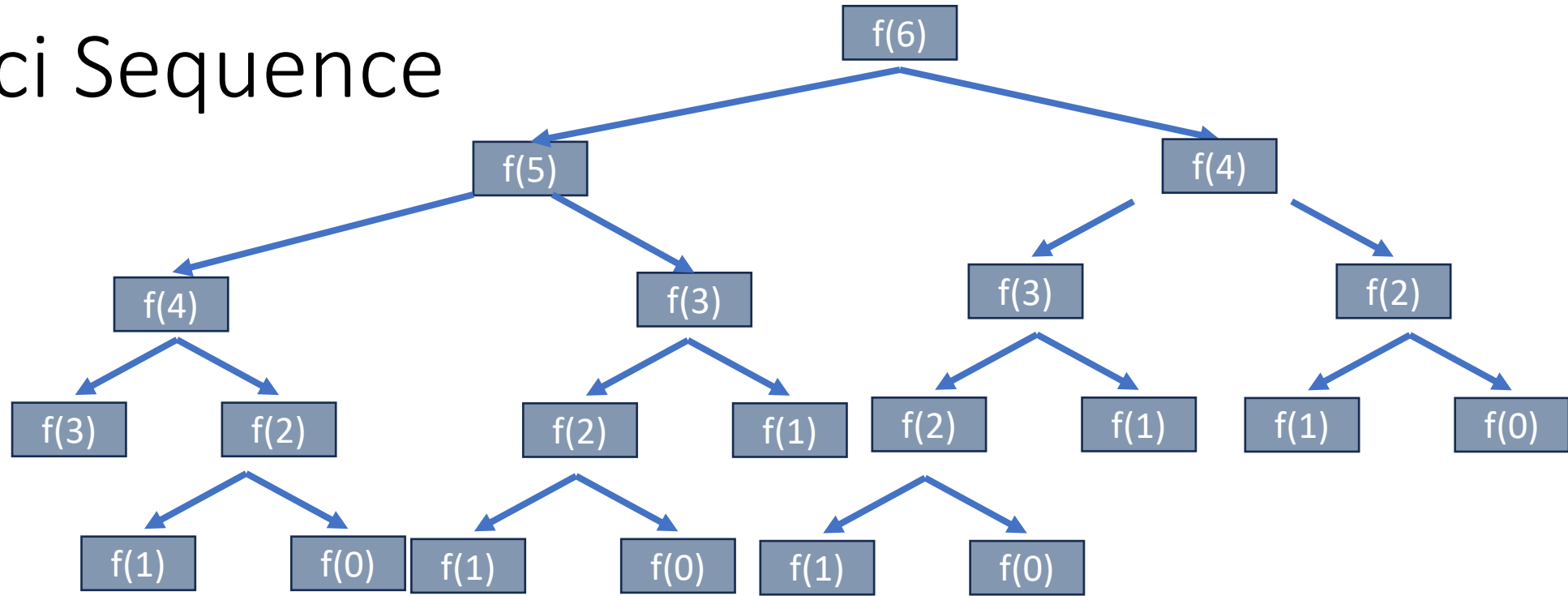
Announcements

- HW02 due ~~Thursday October 5th~~
Friday October 6th
- Lab checkoff, deadline is when corresponding HW is due

Outline

- Stacks
- Junit
- Queues

Fibonacci Sequence



```
public static int fib(int n) {  
    if (n <= 1) {  
        return 1; }  
    return fib(n-1) + fib(n-2);  
}
```

Stacks

Simple and surprisingly useful data structure

Can store any number of items

User can only interact with the top of the stack:

- Push: add a new element to the top
- Pop: take off the top element
- Top/Peek: view the top element without removing it

Stacks - Applications

Hardware call stack

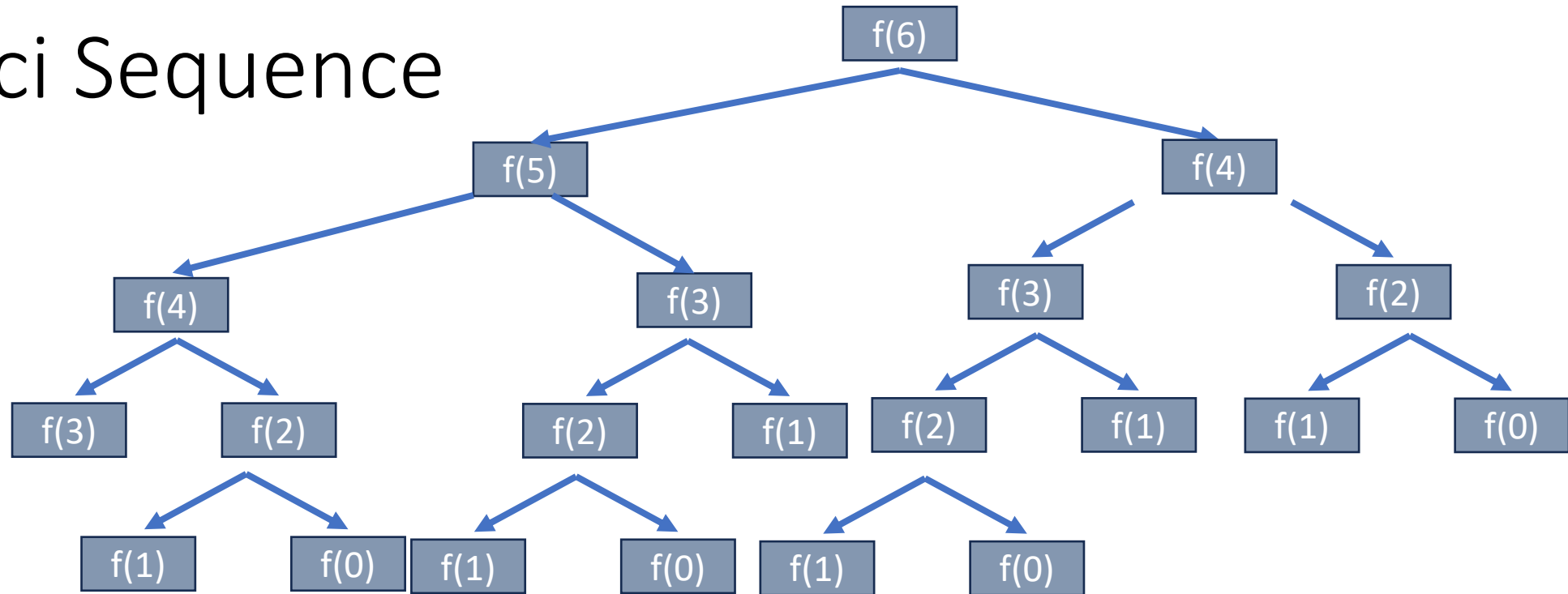
Memory Management

Parsing arithmetic instructions:

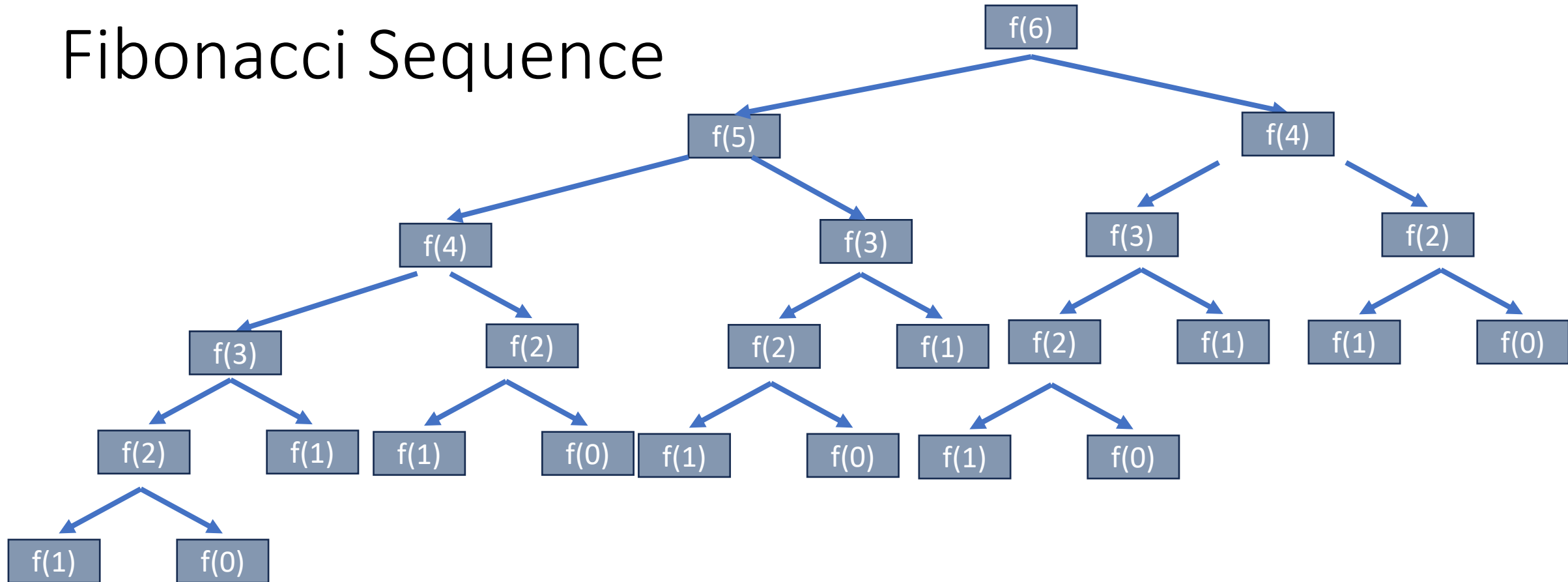
$$((x*2) + (4 + x)) * (3 * \cos(x))$$

Back-tracing (e.g. searching in a maze)

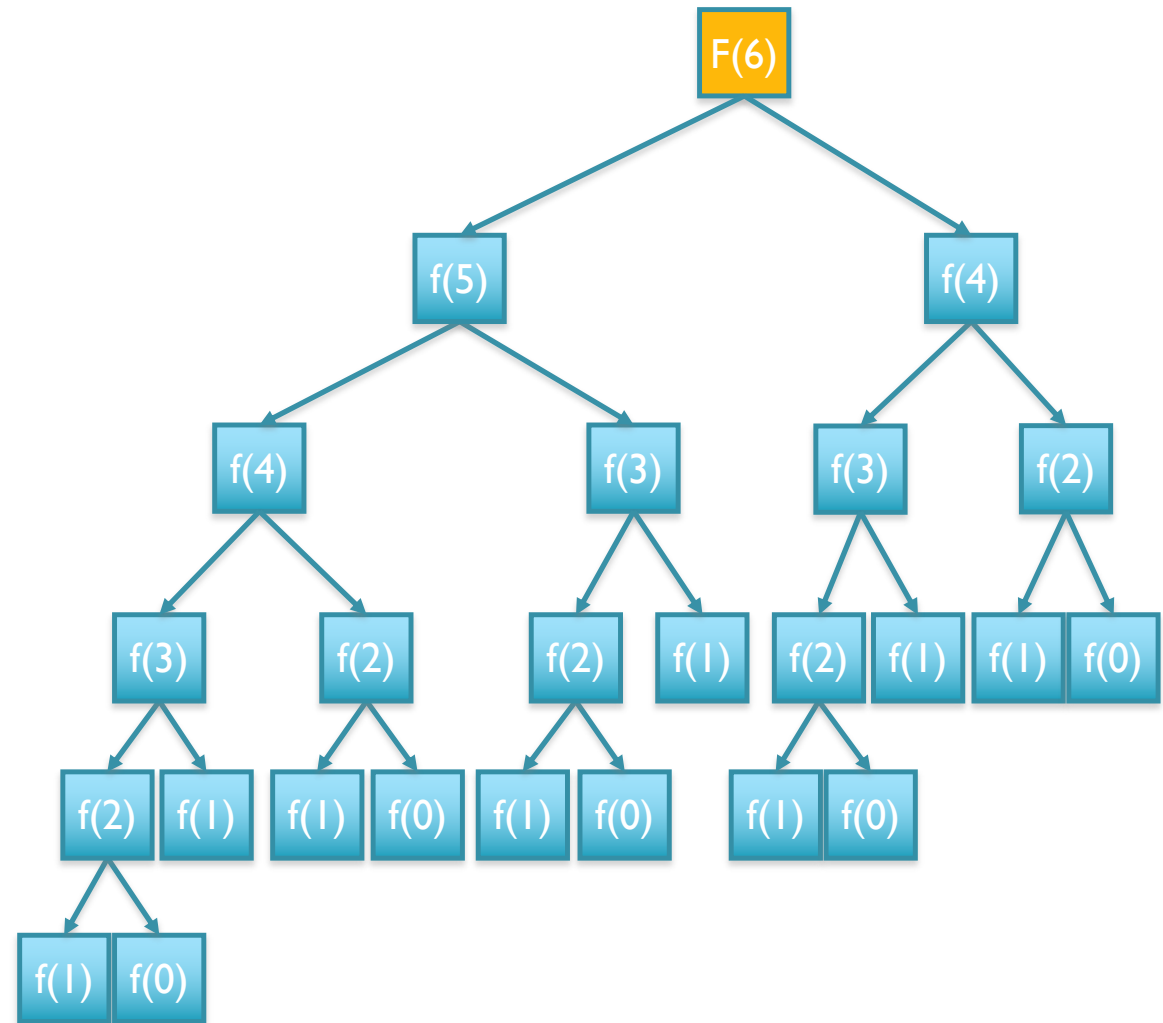
Fibonacci Sequence



Fibonacci Sequence

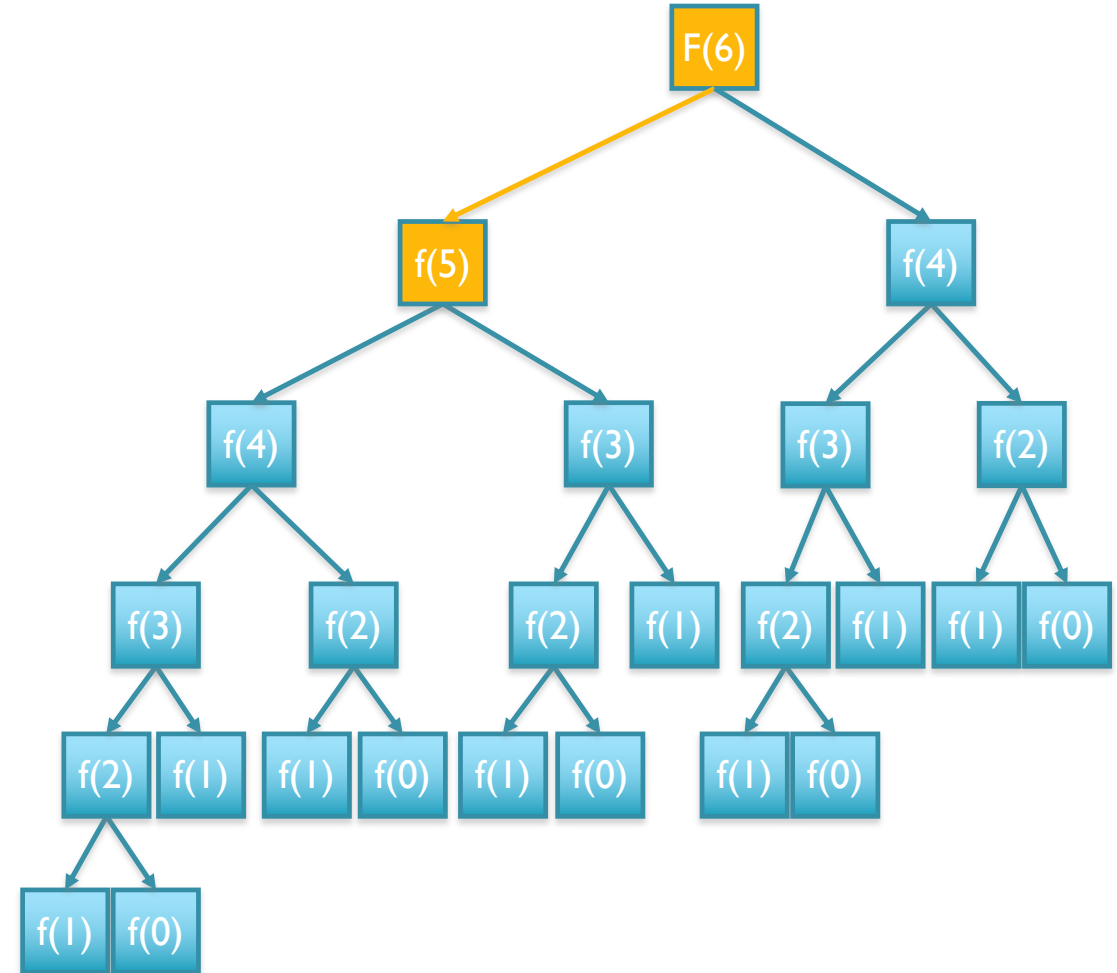


Fibonacci Sequence



$F(6)$
 $a = F(5)$

Fibonacci Sequence



$F(5)$
 $a = F(4)$

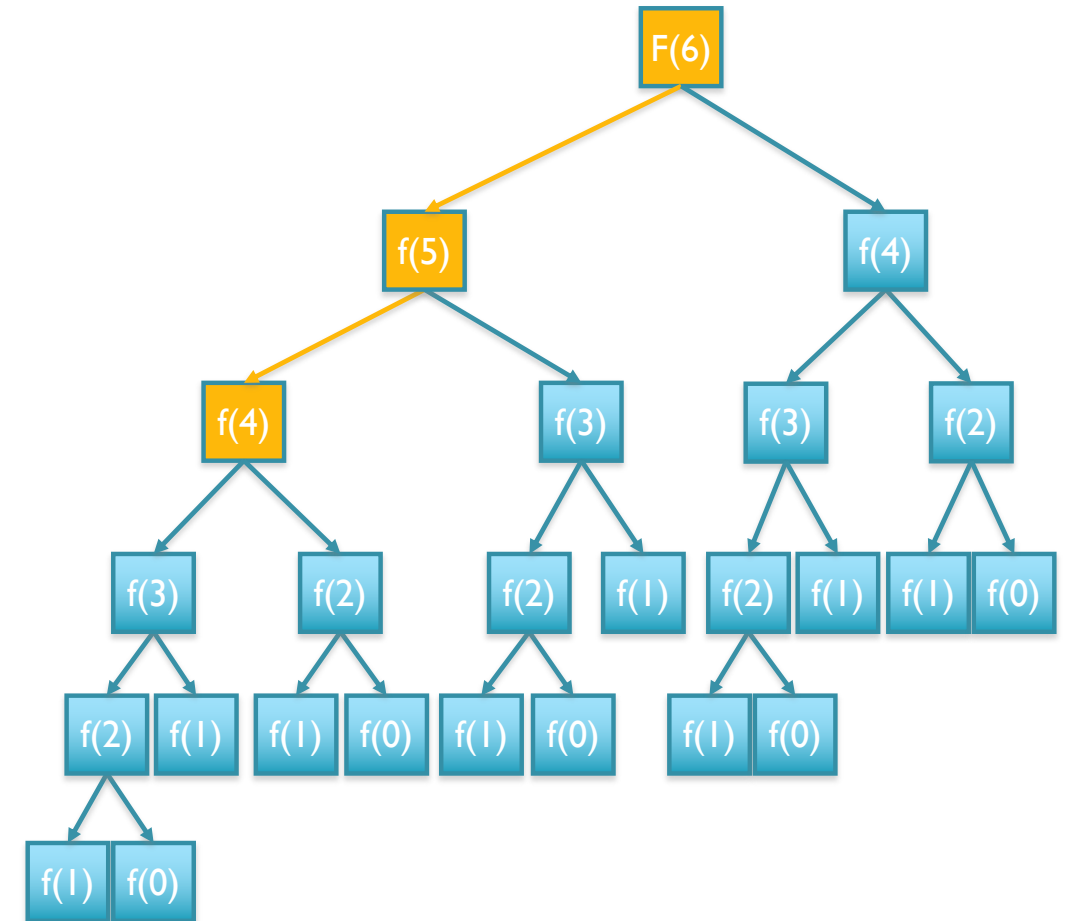
$F(6)$
 $a = F(5)$

Fibonacci Sequence

$F(4)$
 $a = F(3)$

$F(5)$
 $a = F(4)$

$F(6)$
 $a = F(5)$



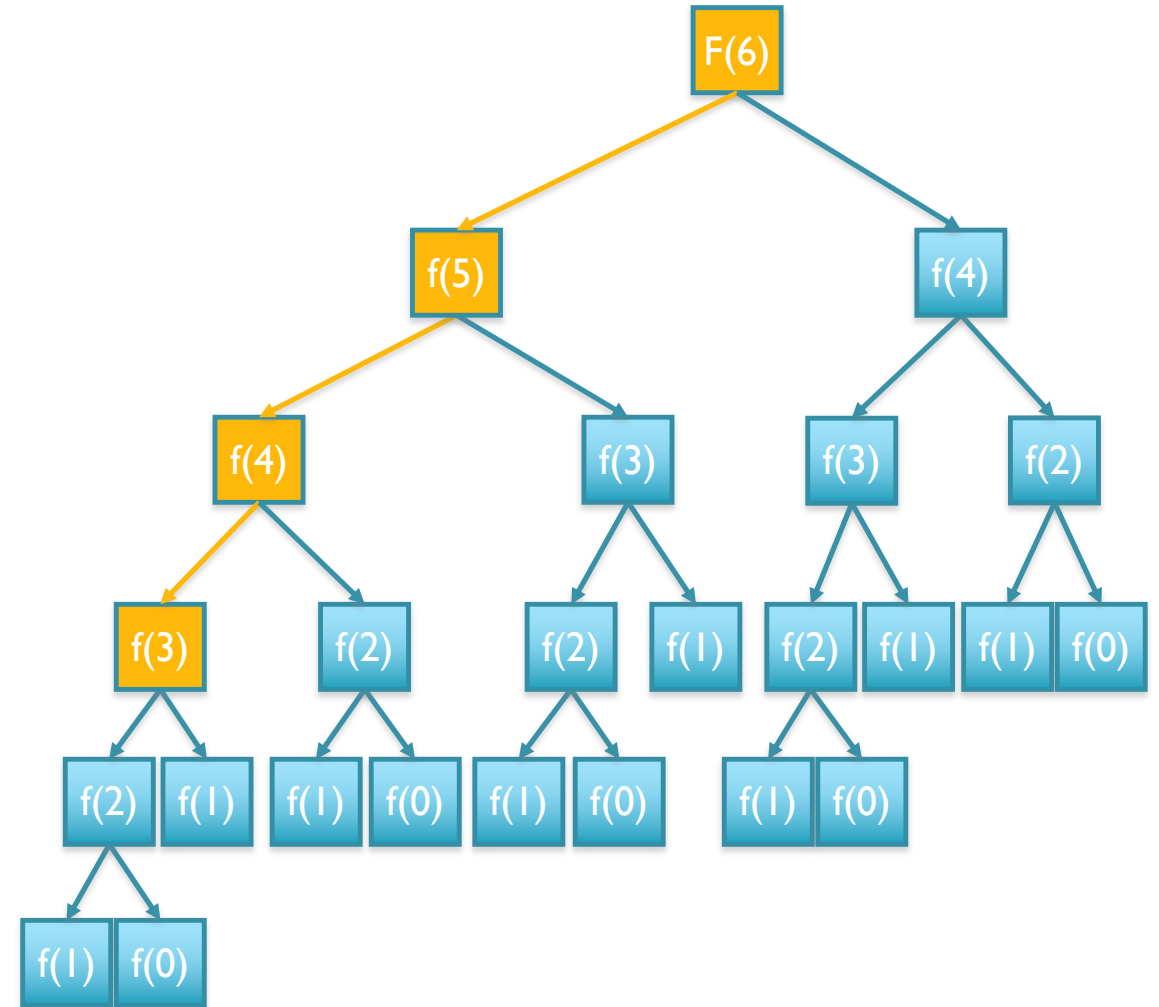
Fibonacci Sequence

$F(3)$
 $a = F(2)$

$F(4)$
 $a = F(3)$

$F(5)$
 $a = F(4)$

$F(6)$
 $a = F(5)$



Fibonacci Sequence

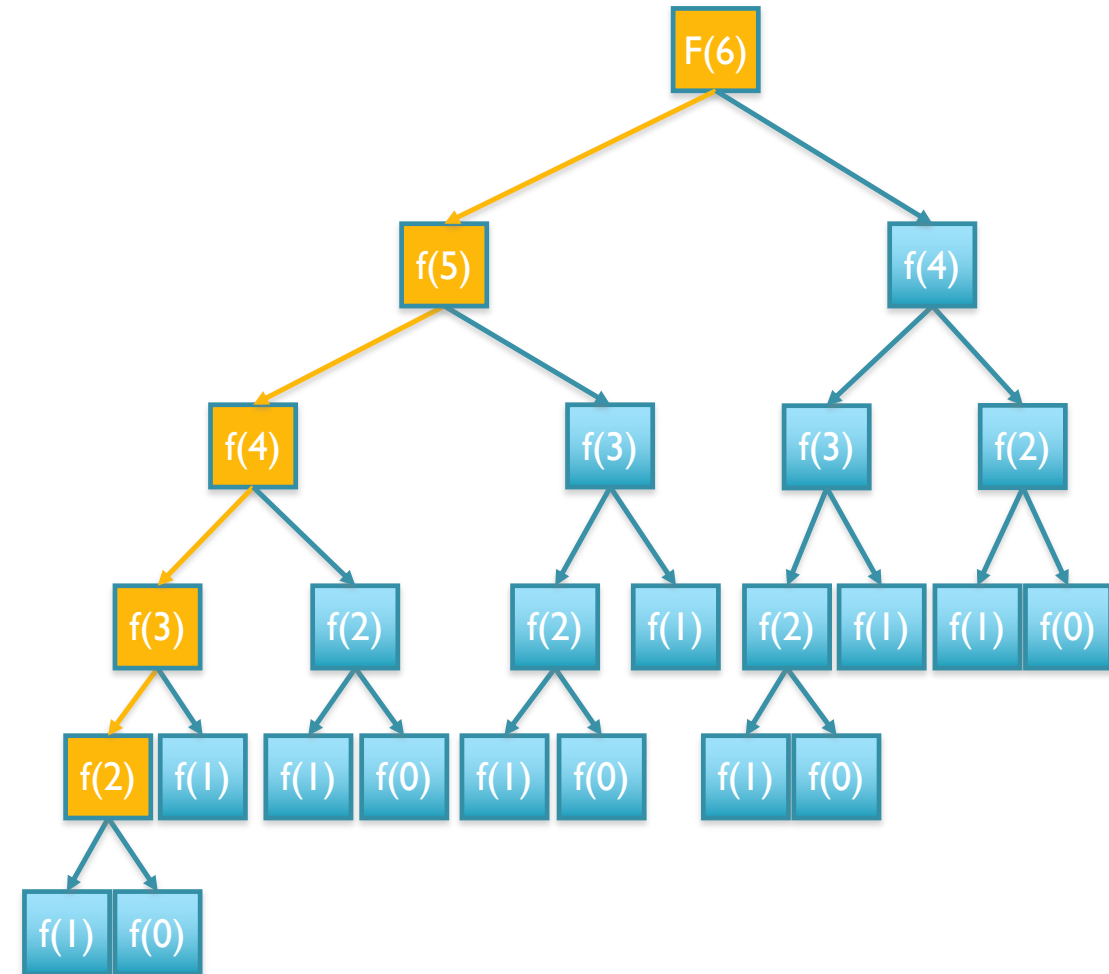
$F(2)$
 $a = F(1)$

$F(3)$
 $a = F(2)$

$F(4)$
 $a = F(3)$

$F(5)$
 $a = F(4)$

$F(6)$
 $a = F(5)$



Fibonacci Sequence

F(1)
return 1

F(2)
a = F(1)

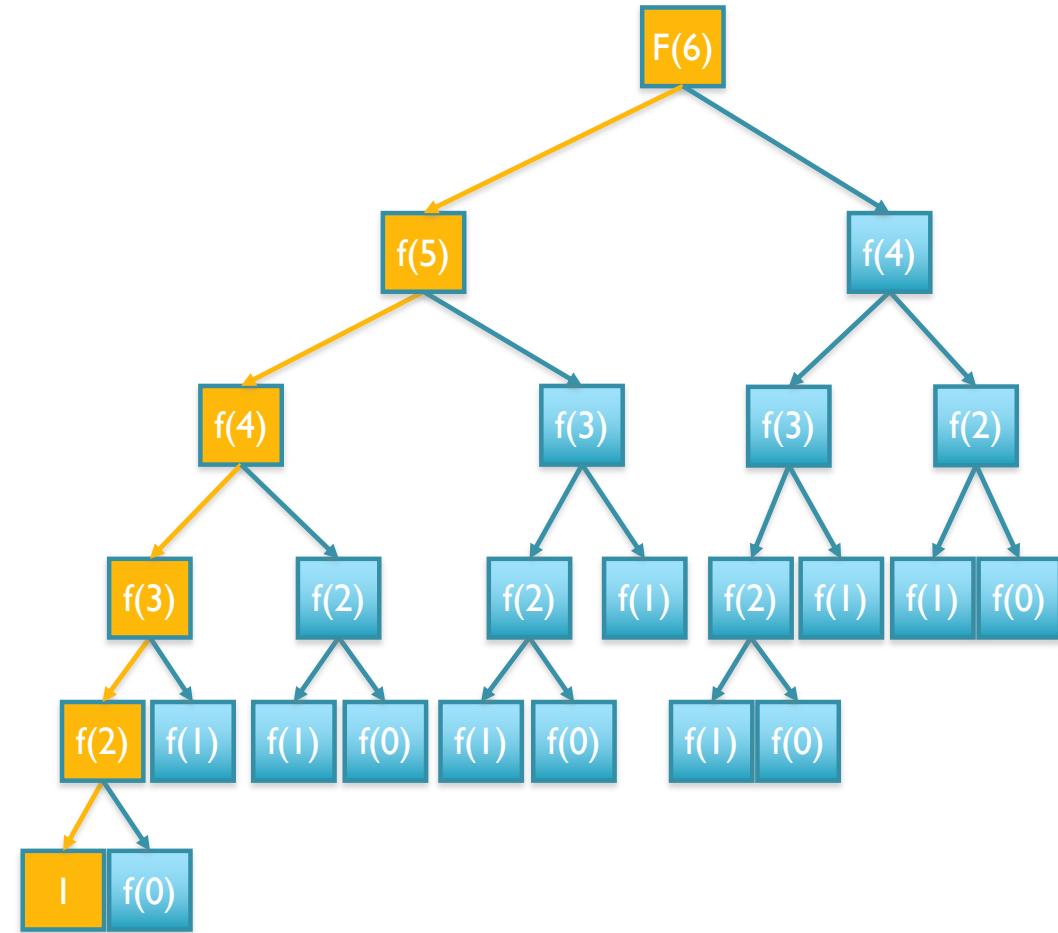
F(3)
a = F(2)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)

```
public static int fib(int n) {  
    if (n <= 1) {  
        return 1;  
    }  
    return fib(n-1) + fib(n-2);  
}
```



Fibonacci Sequence

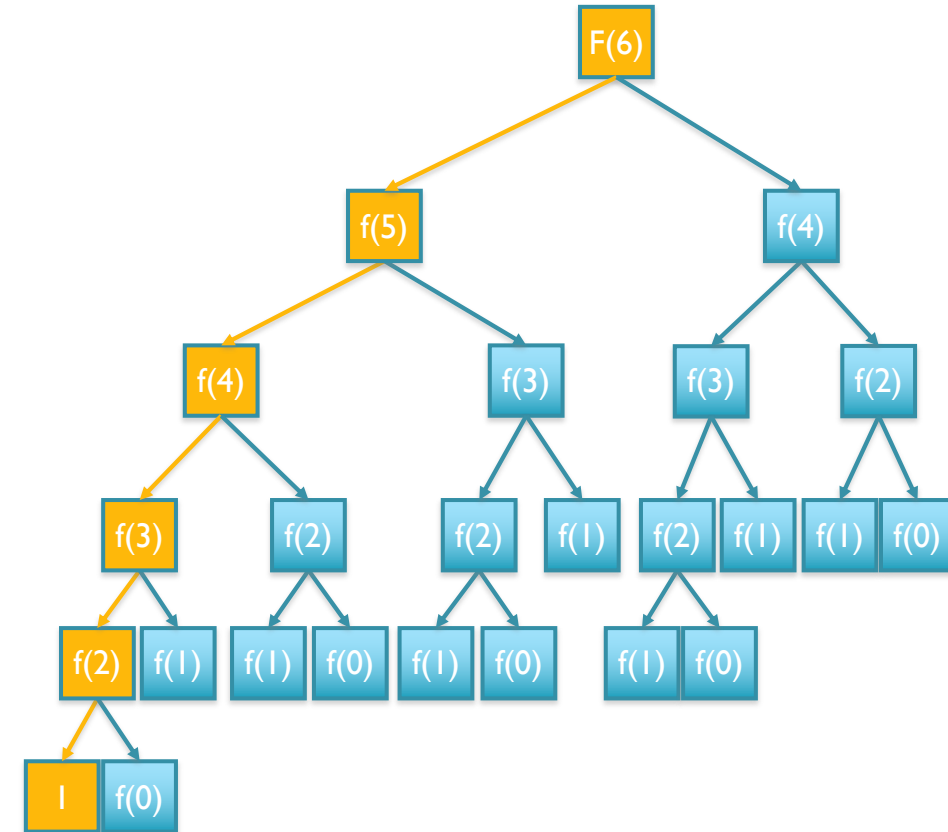
$F(2)$
 $b = F(0)$

$F(3)$
 $a = F(2)$

$F(4)$
 $a = F(3)$

$F(5)$
 $a = F(4)$

$F(6)$
 $a = F(5)$



Fibonacci Sequence

```
public static int fib(int n) {  
    if (n <= 1) {  
        return 1;  
    }  
    return fib(n-1) + fib(n-2);  
}
```

F(0)
return 1

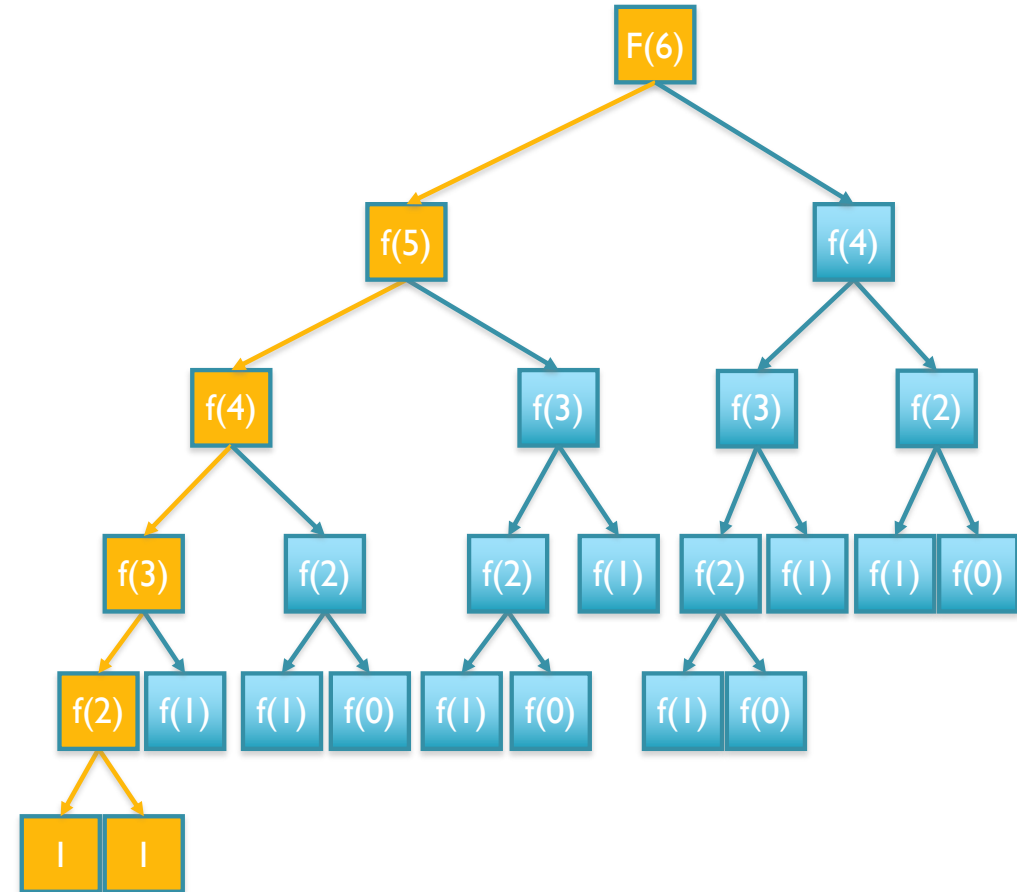
F(2)
b = F(0)

F(3)
a = F(2)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence

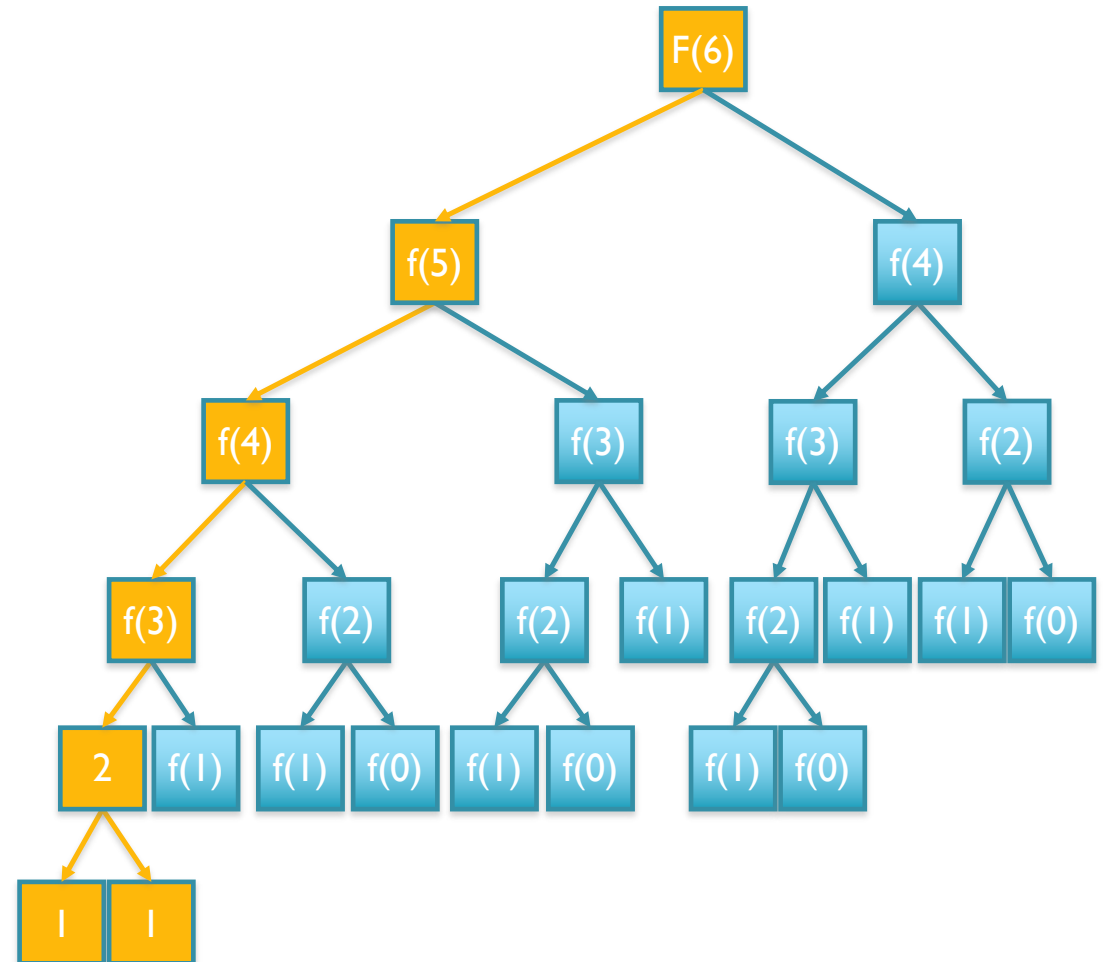
$F(2)$
return 2

$F(3)$
 $a = F(2)$

$F(4)$
 $a = F(3)$

$F(5)$
 $a = F(4)$

$F(6)$
 $a = F(5)$



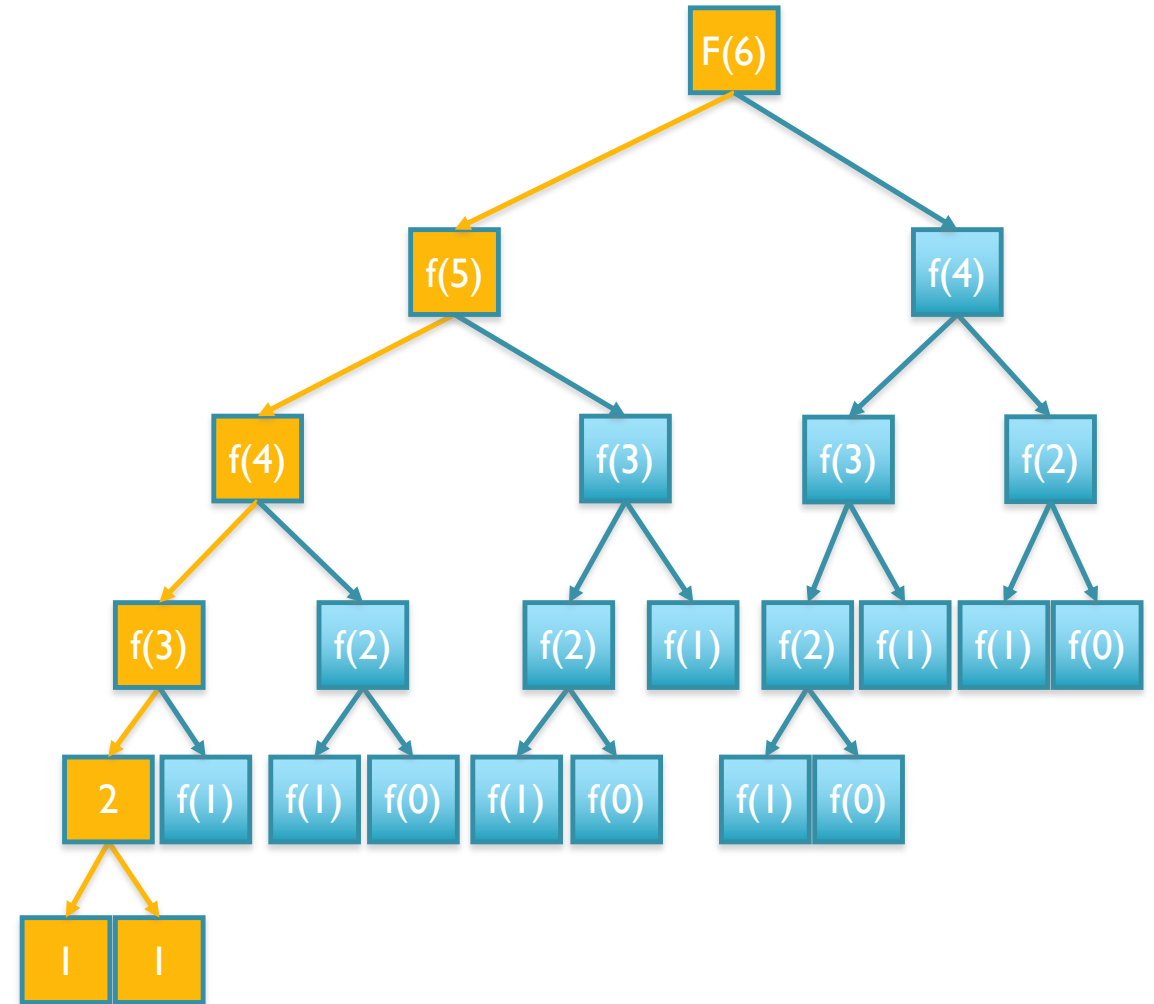
Fibonacci Sequence

$F(3)$
 $a = F(2)$

$F(4)$
 $a = F(3)$

$F(5)$
 $a = F(4)$

$F(6)$
 $a = F(5)$



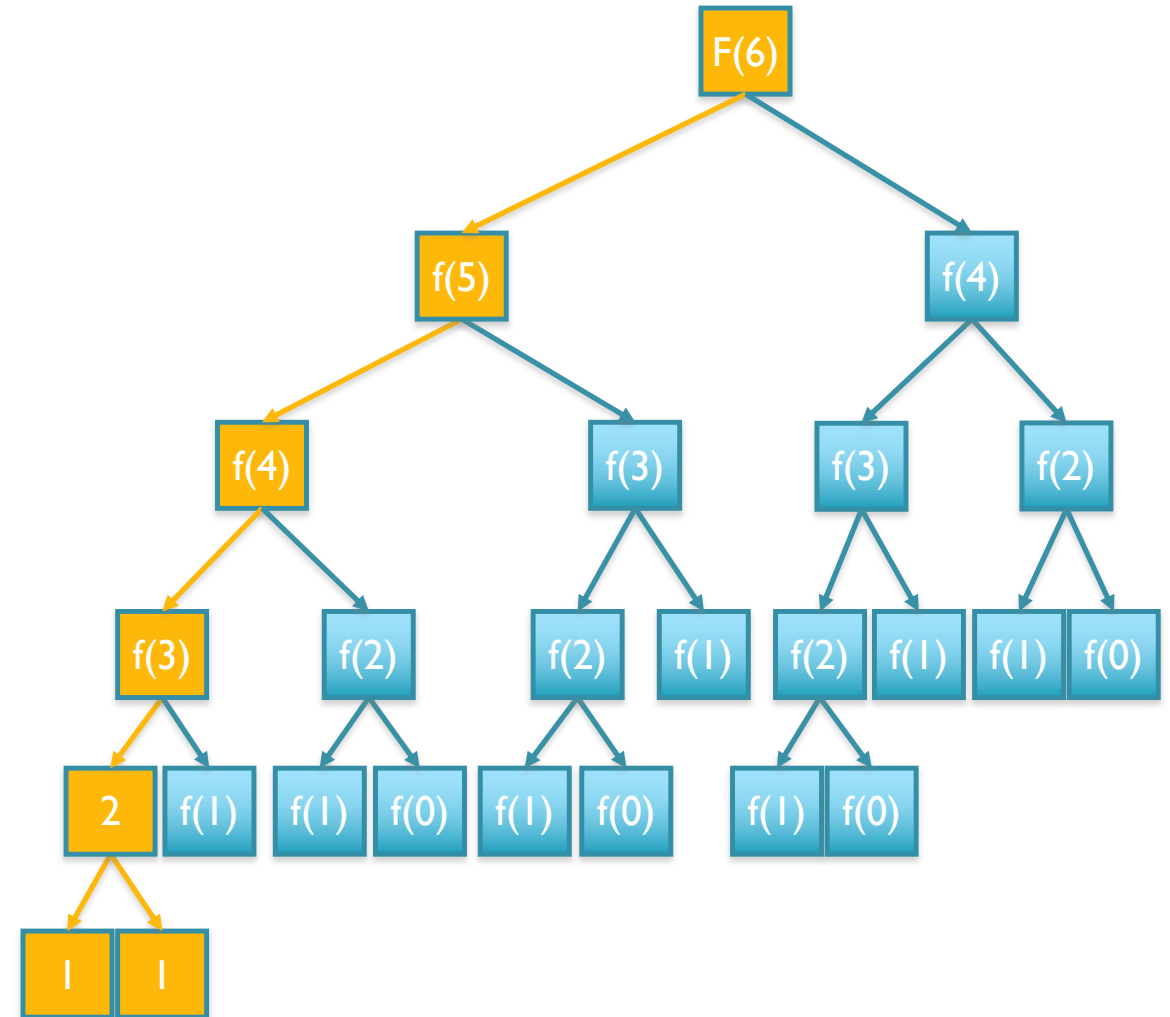
Fibonacci Sequence

$F(3)$
 $b = F(1)$

$F(4)$
 $a = F(3)$

$F(5)$
 $a = F(4)$

$F(6)$
 $a = F(5)$



Fibonacci Sequence

```
public static int fib(int n) {  
    if (n <= 1) {  
        return 1;  
    }  
    return fib(n-1) + fib(n-2);  
}
```

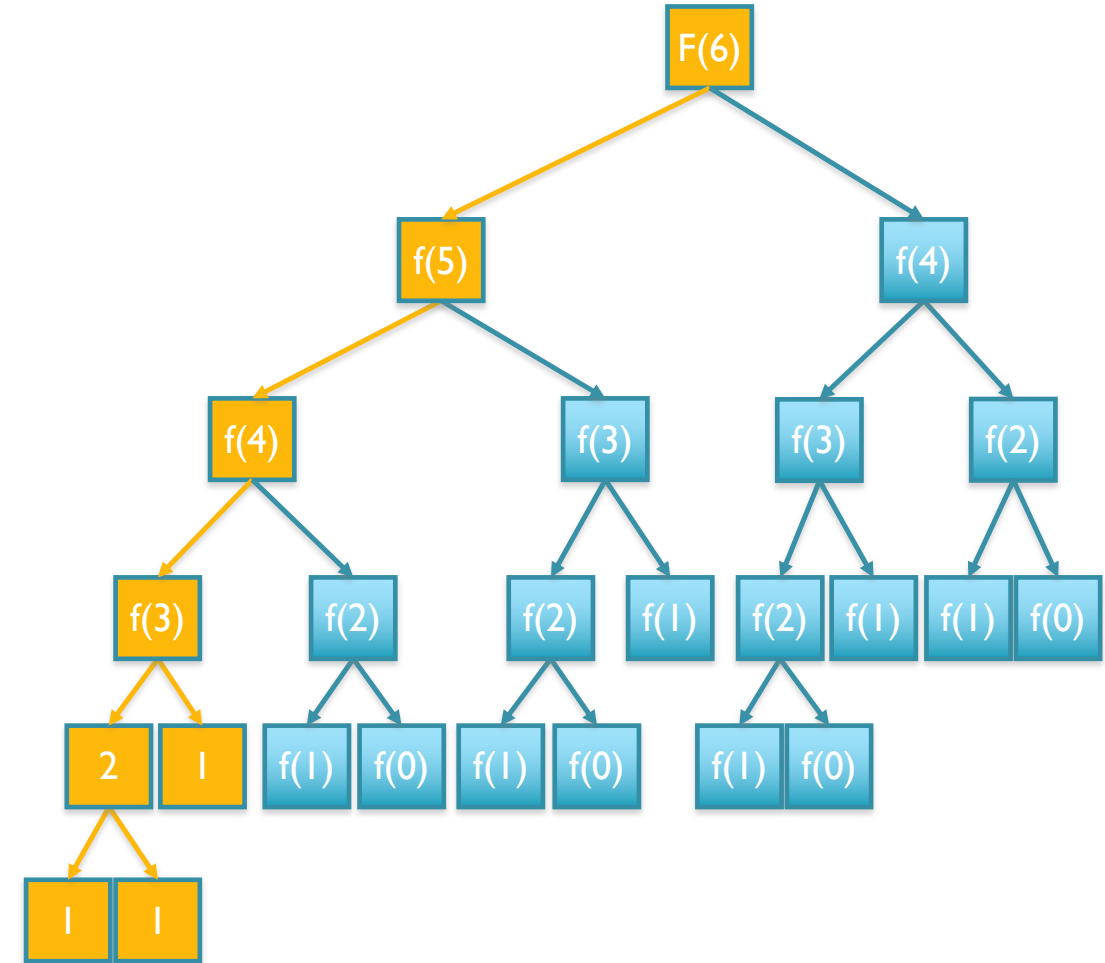
$F(1)$
return 1

$F(3)$
b = $F(1)$

$F(4)$
a = $F(3)$

$F(5)$
a = $F(4)$

$F(6)$
a = $F(5)$



Stack Applications

- Reversing
- Matching
 - `()(()){([()]}`
 - `((()(()))){([()]}`
 - `()`
 - `{[]}`
 - `)(()){([()]}`
 - `(`

```
for each symbol s:
    if s = ( or [ or {
        push(s)
    if s = ) or ] or }
        t = pop()
        if s doesn't match t
            reject
if stack is empty
    accept
else
    reject
```

Stack Applications

Postfix notation

- 5 6 * 2 +
 $5 * 6 + 2$
- 3 4 5 * -
 $3 - 4 * 5$
- 3 4 - 5 *
 $(3 - 4) * 5$

Evaluating postfix expressions with a stack

- operands – push
- operator – pop top two operands, perform operation and push results back on

Stack Applications

Evaluating postfix expressions with a stack

- operands – push
- operator – pop top two operands, perform operation and push results back on
- 15 7 1 1 + - / 3 * 2 1 1 + + -
- $((15/(7-(1+1))) * 3) - (2+(1+1))$

Testing Stack Implementation

new stack is empty

Pushing "hello", then top return "hello"

Push, pop, then stack should be empty

JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.

```
@Test
public void newArrayListsHaveNoElements() {
    assertThat(new ArrayList<Integer>().size(), is(0));
}
```

```
@Test
public void sizeReturnsNumberOfElements() {
    List<Object> instance = new ArrayList<Object>();
    instance.add(new Object());
    instance.add(new Object());
    assertThat(instance.size(), is(2));
}
```

Annotations

Start by marking your tests with `@Test`.

Let's take a tour »

Welcome

- [Download and install](#)
- [Getting started](#)
- [Release Notes](#)
 - [4.13.2](#)
 - [4.13.1](#)
 - [4.13](#)
 - [4.12](#)
 - [4.11](#)
 - [4.10](#)
 - [4.9.1](#)
 - [4.9](#)
- [Maintainer Documentation](#)
- [I want to help!](#)
- [Latest JUnit Questions on StackOverflow](#)
- [JavaDocs](#)
- [Frequently asked questions](#)
- [Wiki](#)
- [Licence](#)

Usage and Idioms

- [Assertions](#)
- [Test Runners](#)
- [Aggregating tests in Suites](#)
- [Test Execution Order](#)
- [Exception Testing](#)
- [Matchers and assertThat](#)
- [Ignoring Tests](#)
- [Timeout for Tests](#)
- [Parameterized Tests](#)
- [Assumptions with Assume](#)
- [Rules](#)
- [Theories](#)
- [Test Fixtures](#)
- [Categories](#)
- [Use with Maven](#)
- [Multithreaded code and Concurrency](#)
- [Java contract test helpers](#)
- [Continuous Testing](#)

Third-party extensions

- [Custom Runners](#)
- [net.trajano.commons:commons-testing for UtilityClassTestUtil](#) per #646
- [System Rules](#) – A collection of JUnit rules for testing code that uses `java.lang.System`.
- [JUnit Toolbox](#) - Provides runners for parallel testing, a `PoolingWait` class to ease asynchronous testing, and a `WildcardPatternSuite` which allow you to specify wildcard patterns instead of explicitly listing all classes when you create a suite class.
- [junit-quickcheck](#) - QuickCheck-style parameter suppliers for JUnit theories. Uses [junit.contrib's version of the theories machinery](#), which respects generics on theory parameters.

Why use Junit over asserts?

Modularize tests

- Large projects will have as much testing as program code

Run all test cases every time

- When an assert fails, program throws an Exception and stops
- Can get all feedback at once

Future class and jobs will expect familiarity with testing frameworks

Using JUnit

Import Test Annotation Framework

```
import org.junit.Test;
```

Write tests using @Test annotation

```
@Test
public void testEmpty() {
    ArrayStack<String> stack = new ArrayStack<String>(10);
    assertTrue(stack.isEmpty());
}
```

Testing Guidelines

Test every method for correct outputs:

- Try simple and complex examples

Every exception and error condition should be tested too

Write test cases first, then implement

- Will make it easy to know when you are done

Queues

Stack Property

First-in Last-out (FILO)

Applications:

browser history (Ctrl+H)

Undo (Ctrl+Z)

Applications where we don't want FILO:

Queuing system

Cash register

Scheduling tasks

First-in First-out

The first item in, is the first item out

Add-to the back, remove from the front

This is a **Queue**

Inserting – enqueue

Removing - dequeue

Queue Interface

```
public interface Queue<E> {  
    int size();  
    boolean isEmpty();  
    E first();  
    void enqueue(E e);  
    E dequeue();  
}
```

- **null is returned from dequeue() and first() when queue is empty**

Example

<i>Operation</i>	<i>Output</i>	<i>Q</i>

Example

Operation
enqueue(5)



Output *Q*



Example

Operation
enqueue(5)



Output *Q*
— (5)



Example

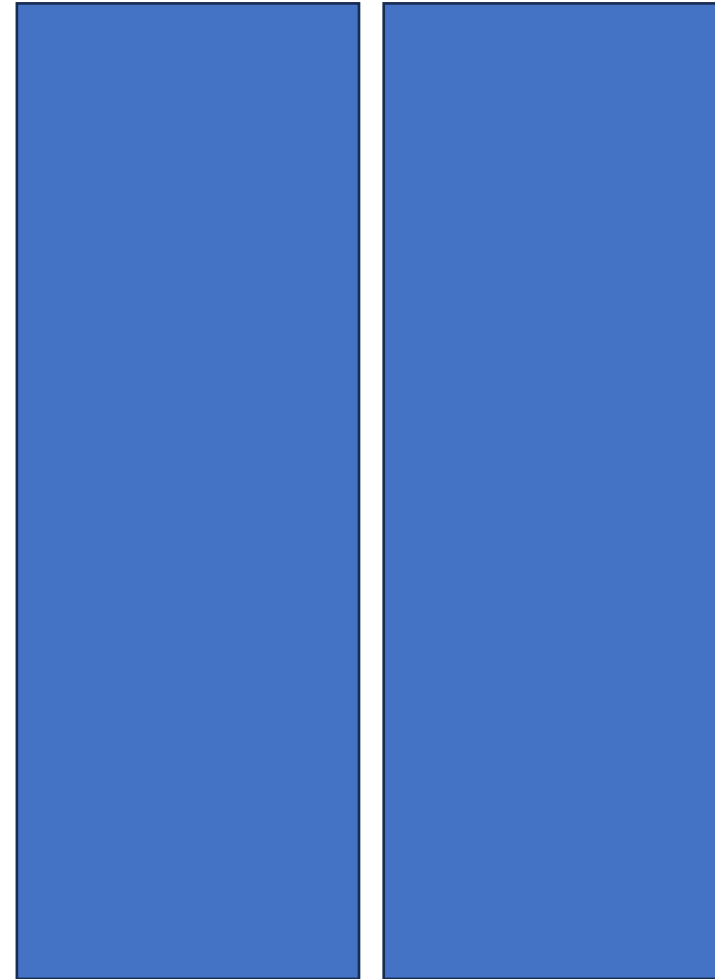
Operation
enqueue(5)
enqueue(3)

Output *Q*
— (5)
— (5, 3)



Example

<i>Operation</i>	<i>Output</i>	<i>Q</i>
enqueue(5)	—	(5)
enqueue(3)	—	(5, 3)
dequeue()		
enqueue(7)		
dequeue()		
first()		
dequeue()		
dequeue()		
isEmpty()		
enqueue(9)		
enqueue(7)		
size()		
enqueue(3)		
enqueue(5)		
dequeue()		



Skip slide

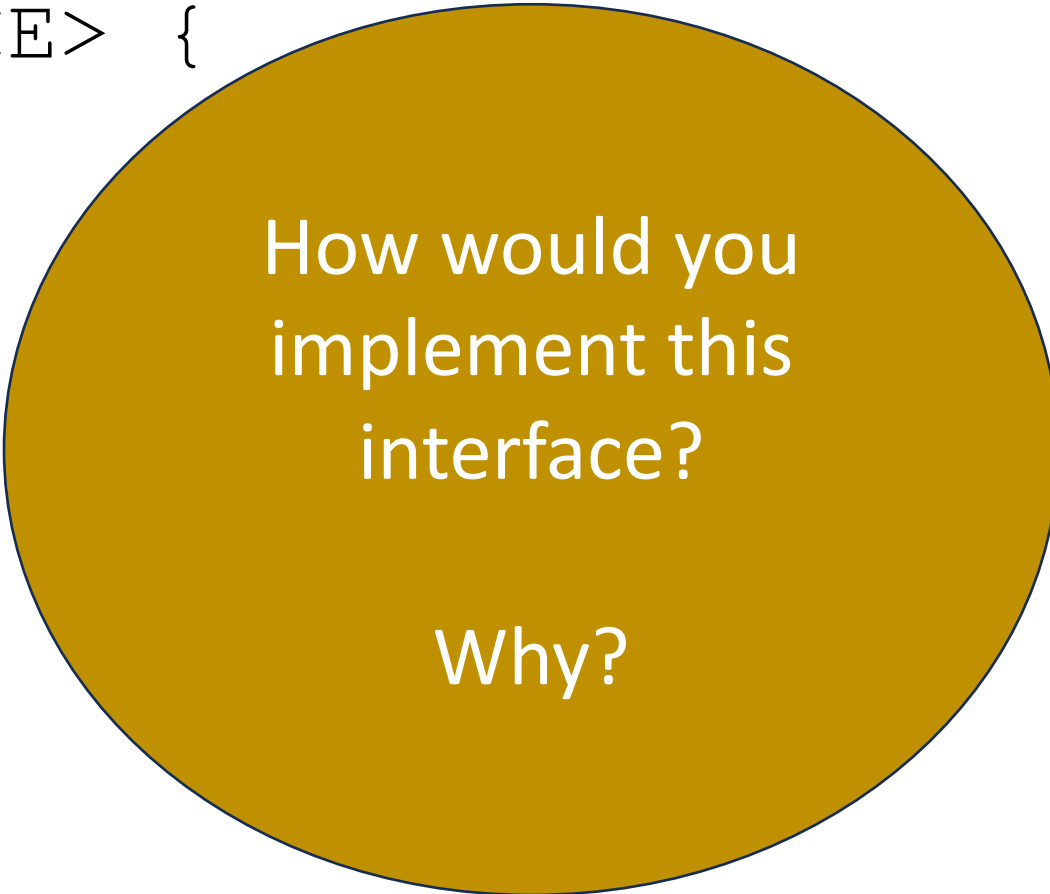
Skip slide

Example

<i>Operation</i>	<i>Output</i>	<i>Q</i>
enqueue(5)	—	(5)
enqueue(3)	—	(5, 3)
dequeue()	5	(3)
enqueue(7)	—	(3, 7)
dequeue()	3	(7)
first()	7	(7)
dequeue()	7	()
dequeue()	<i>null</i>	()
isEmpty()	<i>true</i>	()
enqueue(9)	—	(9)
enqueue(7)	—	(9, 7)
size()	2	(9, 7)
enqueue(3)	—	(9, 7, 3)
enqueue(5)	—	(9, 7, 3, 5)
dequeue()	9	(7, 3, 5)

Queue Interface

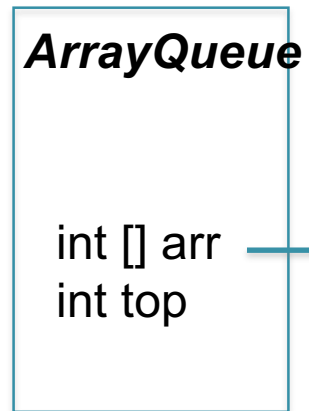
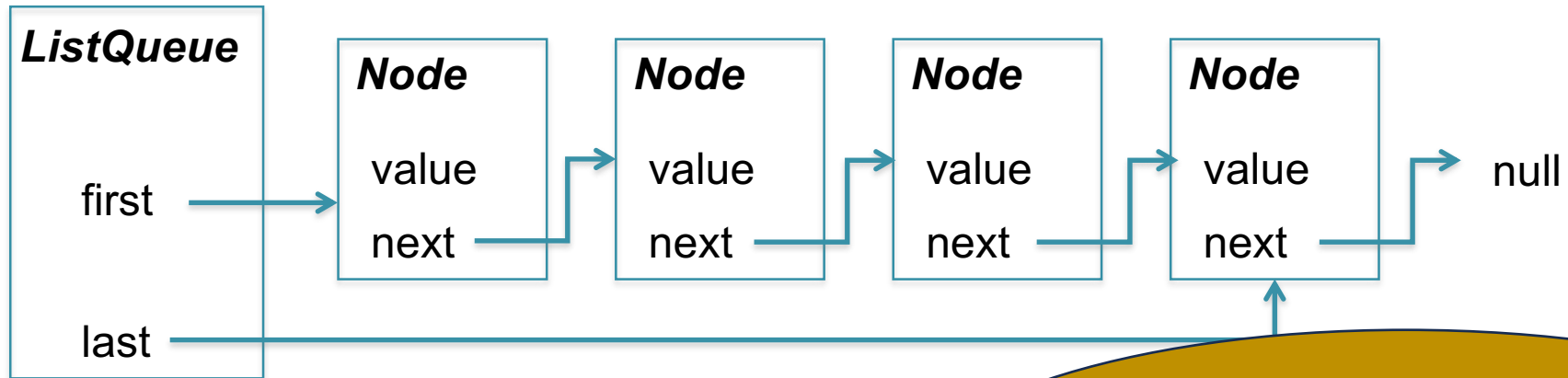
```
public interface Queue<E> {  
    int size();  
    boolean isEmpty();  
    E first();  
    void enqueue(E e);  
    E dequeue();  
}
```



How would you
implement this
interface?

Why?

ListQueue vs ArrayQueue

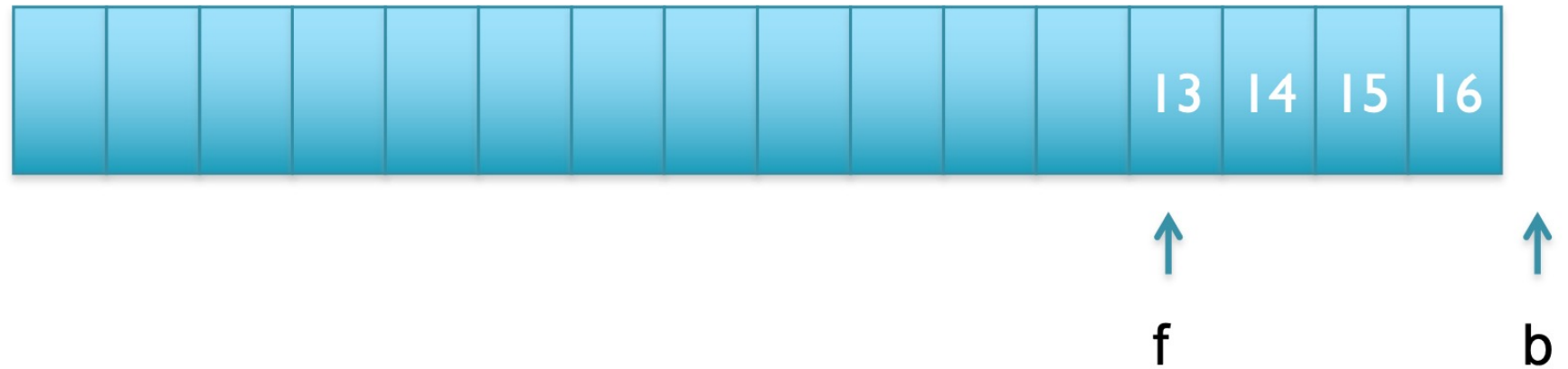


Many of the same tradeoffs as ListStack vs ArrayStack

ArrayQueue

ArrayQueue

```
int [] arr  
int f = 13  
int b = 17
```



What should we do if
we insert?

Comparison to `java.util.Queue`

Our Queue ADT	Interface <code>java.util.Queue</code>	
	throws exceptions	returns special value
<code>enqueue(<i>e</i>)</code>	<code>add(<i>e</i>)</code>	<code>offer(<i>e</i>)</code>
<code>dequeue()</code>	<code>remove()</code>	<code>poll()</code>
<code>first()</code>	<code>element()</code>	<code>peek()</code>
<code>size()</code>	<code>size()</code>	
<code>isEmpty()</code>	<code>isEmpty()</code>	