CS151 Intro to Data Structures

Checkstyle

Stacks

JUnit

Announcements

- HW02 due ~~Tuesday October 3rd~~
  Thursday October 5th

- Lab checkoff, deadline is when corresponding HW is due

HW02 Recommendations

Each file should still be read only once

- Create only one new `Scanner` per file
- No resetting the `Scanner`s

Totals are computed only once and stored, not over and over again

All relevant stats for one name is stored in a single `Name` object

Print yearly stats in the order of the files given

HW02 Recommendations

Parse the year from the filename

Don’t store the percentages

- Computing and storing totals cost no additional loop
- Computing and storing percentages do

Do you feel like you are writing the same code twice a lot?

- Try making it into a method and call that twice, instead of repeating the lines

HW02 Recommendations

Command-line Arguments:

Array passed into `Main`: `String[] args`

```
for(int i=0; i<args.length; i++) {
  System.out.println(args[i]);
}
```

Catch the exceptions!

Quit the program

```
System.exit(0);
```

Outline

- Checkstyle
- Stacks
- JUnit

CS 151 - Data Structures
Style and Design Guide

Now that you are no longer a new programmer, you are expected to pay attention to your code organization. Your programming should adhere to the standards explained here. While some of the rules are due to convention, many more are pearls of wisdom distilled from solid software-engineering principles and all of them are essential for making your code readable and/or easy to maintain and modify/reuse. This document explains guiding principles. For more specific formatting rules please refer to the `formatting guidelines`.

Intentional Coding

CS151 Code Formatting Standards and Guidelines

Naming Conventions

- Use meaningful names! For example, if your program needs a variable to represent the radius of a circle, call it `radius`, *not* `r` and *not* `rad`.
- Use single letter variables for simple loop indices *only*.
- The use of very obvious, common, meaningful abbreviations is permitted. For example, "number" can be abbreviated as "num" as in `numStudents`.
- Variable, instance variables, and method names in Java generally are written in camelCase, starting with a lower-case letter and putting the first letter of subsequent words in uppercase.
- Class names are written in PascalCase, starting with a capital letter.
- Constants (`static final`) are written in `ALL_CAPS`.

Whitespace

The most-readable programs are written with prudent use of whitespace (including both blank lines and

Checkstyle

```
(base) apoliak@mani:~/handouts/cs151/class-examples-f23$ java -jar checkstyle-8.16-all.jar -c cs151_
checks.xml lec08/HelloWorld.java
Starting audit...
[ERROR] /home/apoliak/handouts/cs151/class-examples-f23/lec08/HelloWorld.java:1: Missing a Javadoc c
omment. [JavadocType]
[ERROR] /home/apoliak/handouts/cs151/class-examples-f23/lec08/HelloWorld.java:3: 'method def modifie
r' has incorrect indentation level 2, expected level should be 4. [Indentation]
[ERROR] /home/apoliak/handouts/cs151/class-examples-f23/lec08/HelloWorld.java:3:3: Missing a Javadoc
 comment. [JavadocMethod]
[ERROR] /home/apoliak/handouts/cs151/class-examples-f23/lec08/HelloWorld.java:4:1: File contains tab
 characters (this is the first instance). [FileTabCharacter]
[ERROR] /home/apoliak/handouts/cs151/class-examples-f23/lec08/HelloWorld.java:6: 'method def rcurly'
 has incorrect indentation level 0, expected level should be 4. [Indentation]
Audit done.
Checkstyle ends with 5 errors.
```

```
java -jar checkstyle-8.16-all.jar -c \
cs151_checks.xml lec08/HelloWorld.java
```

Checkstyle

Checkstyle:
https://raw.githubusercontent.com/BMC-CS-151/class-examples-f23/main/cs151_checks.xml

Jar:
https://github.com/checkstyle/checkstyle/releases/tag/checkstyle-8.16

```
java -jar checkstyle-8.16-all.jar -c \
cs151_checks.xml lec08/HelloWorld.java
```

java -jar checkstyle-8.16-all.jar

```
98          /**
99           * Loops over the files specified checking them for errors. The exit code
100          * is the number of errors found in all the files.
101          *
102          * @param args the command line arguments.
103          * @throws IOException if there is a problem with files access
104          * @noinspection UseOfSystemOutOrSystemErr, CallToPrintStackTrace, CallToSystemExit
105          * @noinspectionreason UseOfSystemOutOrSystemErr - driver class for Checkstyle requires
106          *      usage of System.out and System.err
107          * @noinspectionreason CallToPrintStackTrace - driver class for Checkstyle must be able to
108          *      show all details in case of failure
109          * @noinspectionreason CallToSystemExit - driver class must call exit
110          **/
111         public static void main(String... args) throws IOException {
```

https://github.com/checkstyle/checkstyle/blob/617ee09942f9da4a69538dc154fcce3d57334f39/src/main/java/com/puppycrawl/tools/checkstyle/Main.java#L98-L111

Checkstyle Requirements

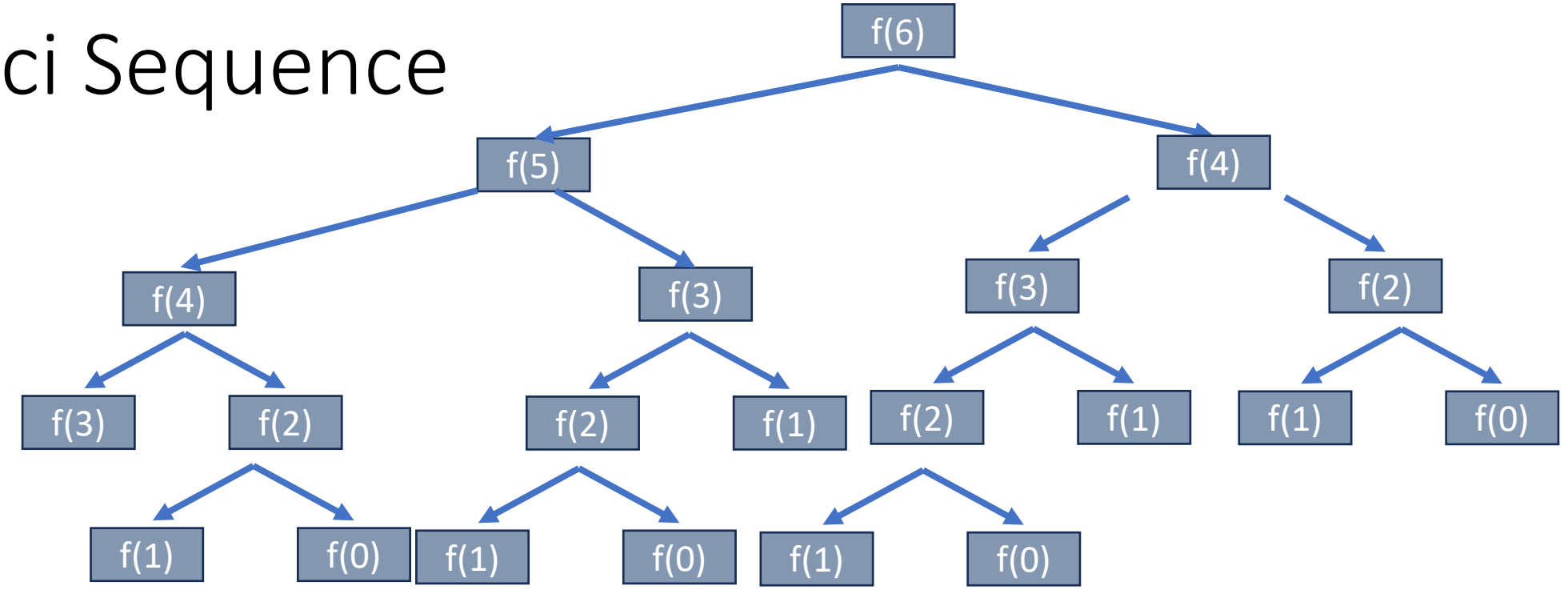
HW04 onwards, all code must comply with the checkstyle

Future courses require it

Employers often require it

Makes working collaboratively much easier

Fibonacci Sequence




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

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

h
Return val to g line 501
x=56
k=56
val = 57

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

Call Stack

Keeps track of the local variables and return location for the current function

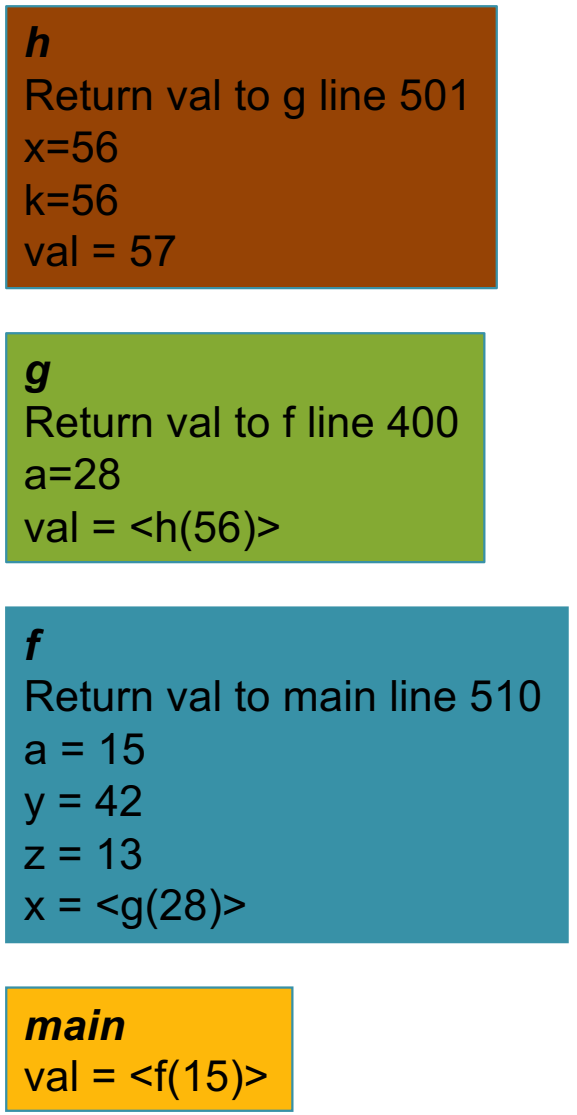
Allows program to jump from function to function without loosing track of where the program should resume

stack frame: records the information for each function call:

- local variables
- address of where to resume processing after this function is complete.

computer only needs to ***add or remove items from the very top of the stack***,

stacks are a very useful data structure for ***Last-In-First-Out (LIFO)*** processing



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

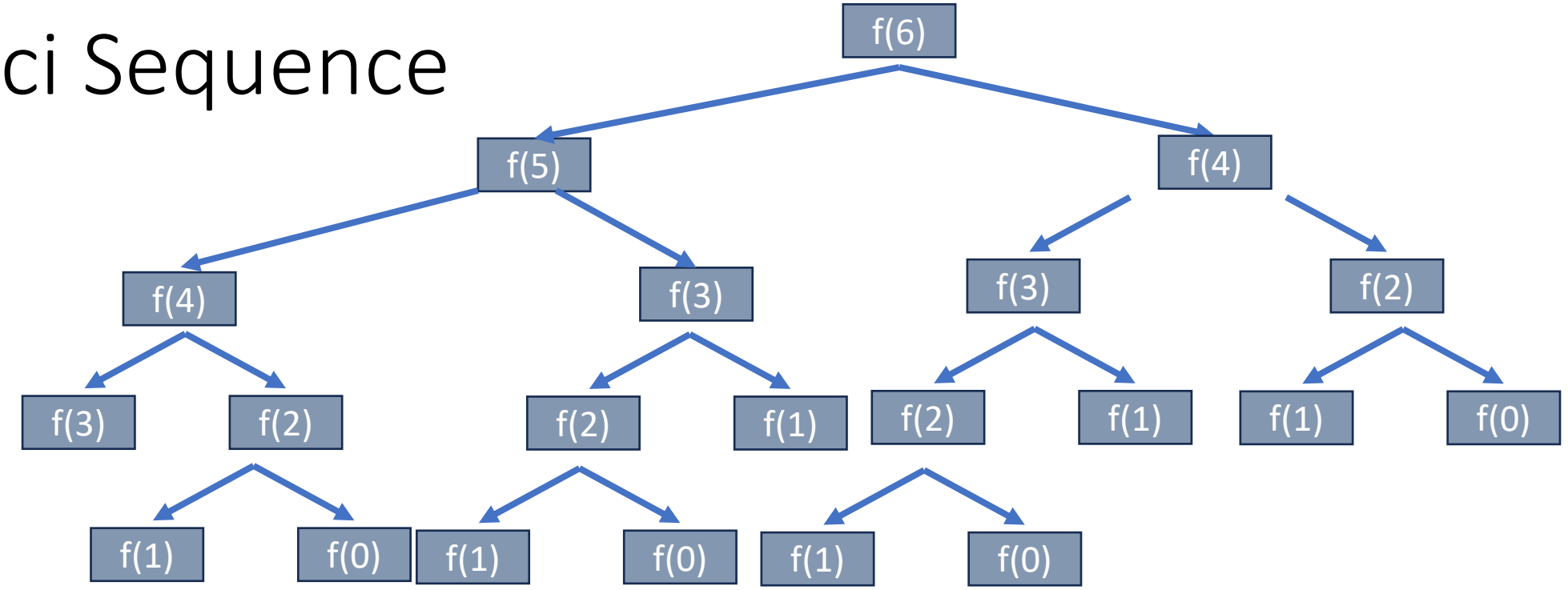
Memory Management

Parsing arithmetic instructions:

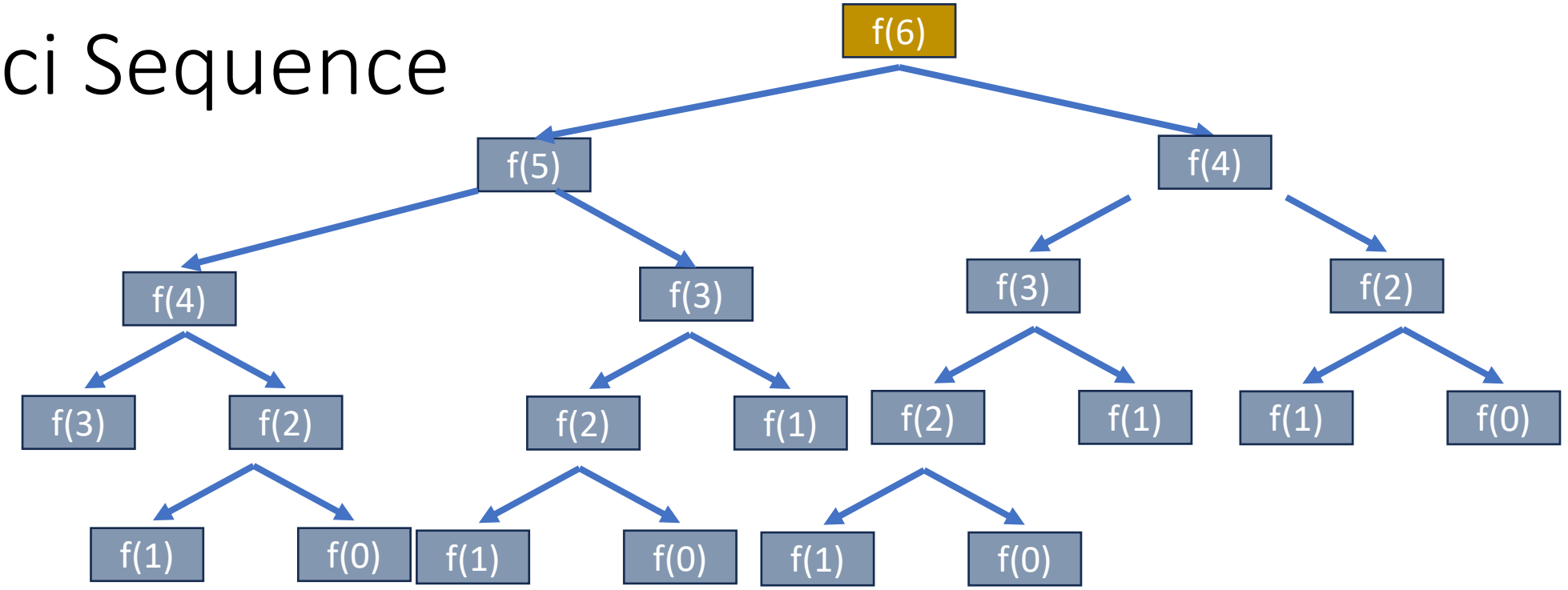
((x*2) + (4 + x)) * (3 * cos(x))

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

Fibonacci Sequence

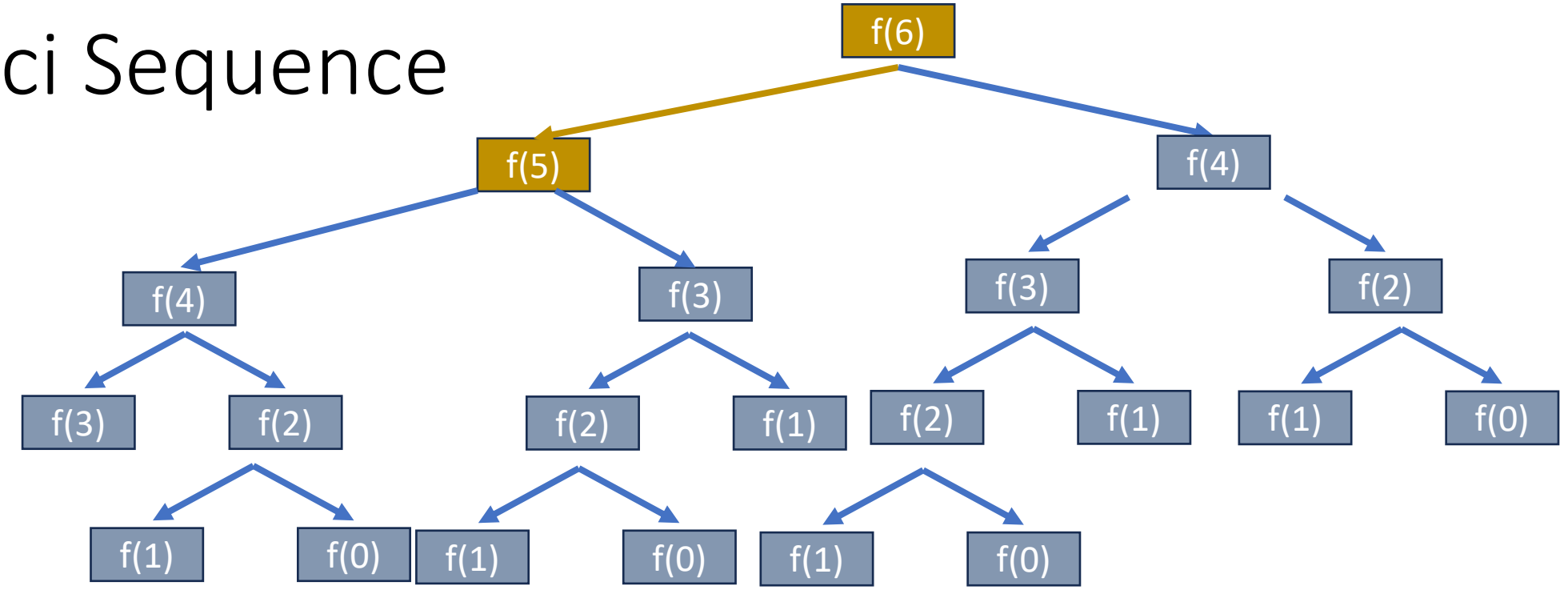


Fibonacci Sequence



f(6)
a = f(5)

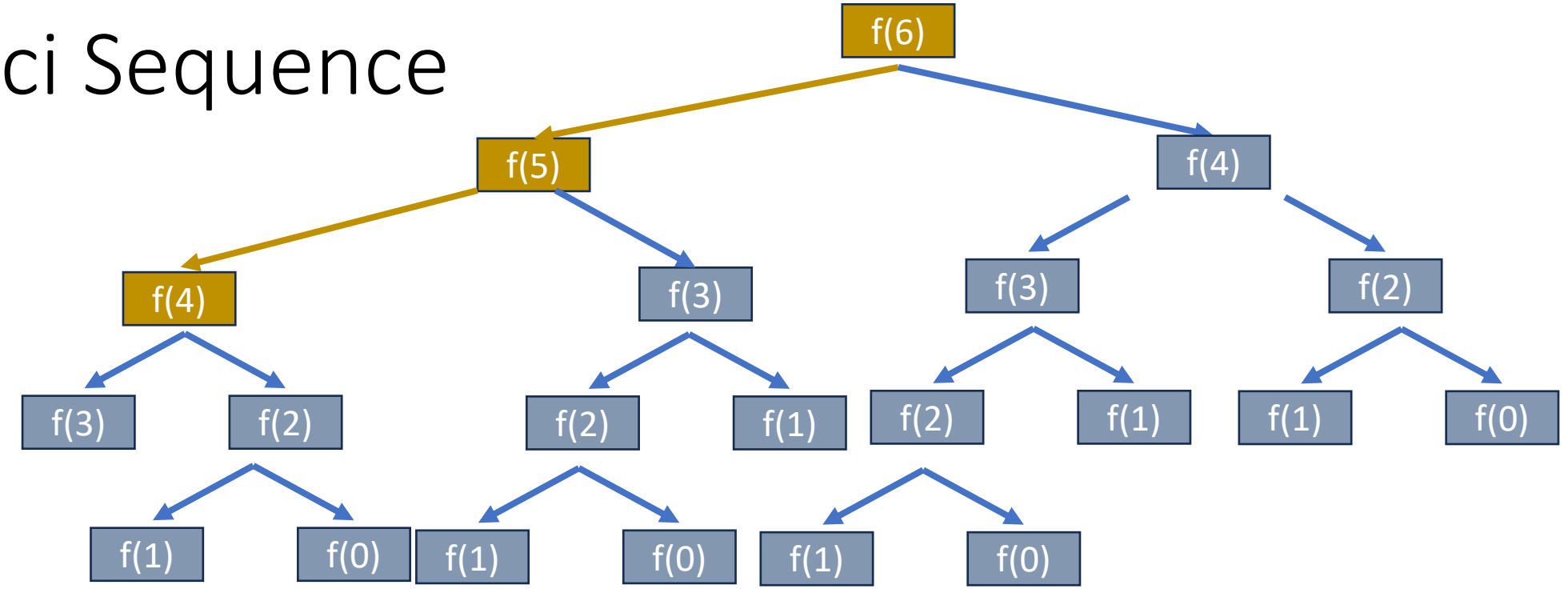
Fibonacci Sequence



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)

StackADT

adt Stack

uses Any, Boolean

defines Stack<T: Any>

operations

preconditions

axioms

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:
        push:
        pop:
        top:
        empty:
preconditions
axioms
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:   ---> Stack<T>
        push:
        pop:
        top:
        empty:
preconditions
```

```
axioms
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:   ---> Stack<T>
        push: Stack<T> x T ---> Stack<T>
        pop:
        top:
        empty:
preconditions
```

```
axioms
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:   ---> Stack<T>
        push: Stack<T> x T ---> Stack<T>
        pop:    Stack<T> ---> Stack<T>
        top:
        empty:
preconditions

axioms
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:   ---> Stack<T>
        push: Stack<T> x T ---> Stack<T>
        pop:    Stack<T> ---> Stack<T>
        top:    Stack<T> ---> T
        empty:
preconditions
```

```
axioms
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:  ---> Stack<T>
        push: Stack<T> x T ---> Stack<T>
        pop:   Stack<T> ---> Stack<T>
        top:   Stack<T> ---> T
        empty: Stack<T> ---> Boolean
preconditions
```

```
axioms
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:  ---> Stack<T>
        push: Stack<T> x T ---> Stack<T>
        pop:   Stack<T> ---> Stack<T>
        top:   Stack<T> ---> T
        empty: Stack<T> ---> Boolean
preconditions
        pop(s): not empty(s)
```

```
axioms
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:  ---> Stack<T>
        push: Stack<T> x T ---> Stack<T>
        pop:   Stack<T> ---> Stack<T>
        top:   Stack<T> ---> T
        empty: Stack<T> ---> Boolean
preconditions
        pop(s): not empty(s)
        top(s): not empty(s)
axioms
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:  ---> Stack<T>
        push: Stack<T> x T ---> Stack<T>
        pop:   Stack<T> ---> Stack<T>
        top:   Stack<T> ---> T
        empty: Stack<T> ---> Boolean
preconditions
        pop(s): not empty(s)
        top(s): not empty(s)
axioms
        empty(new())
```

StackADT

```
adt Stack
uses Any, Boolean
defines Stack<T: Any>
operations
        new:  ---> Stack<T>
        push: Stack<T> x T ---> Stack<T>
        pop:   Stack<T> ---> Stack<T>
        top:   Stack<T> ---> T
        empty: Stack<T> ---> Boolean
preconditions
        pop(s): not empty(s)
        top(s): not empty(s)
axioms
        empty(new())
        not empty(push(s, t))
        top(push(s, t)) = t
        pop(push(s, t)) = s
```

Stack Example

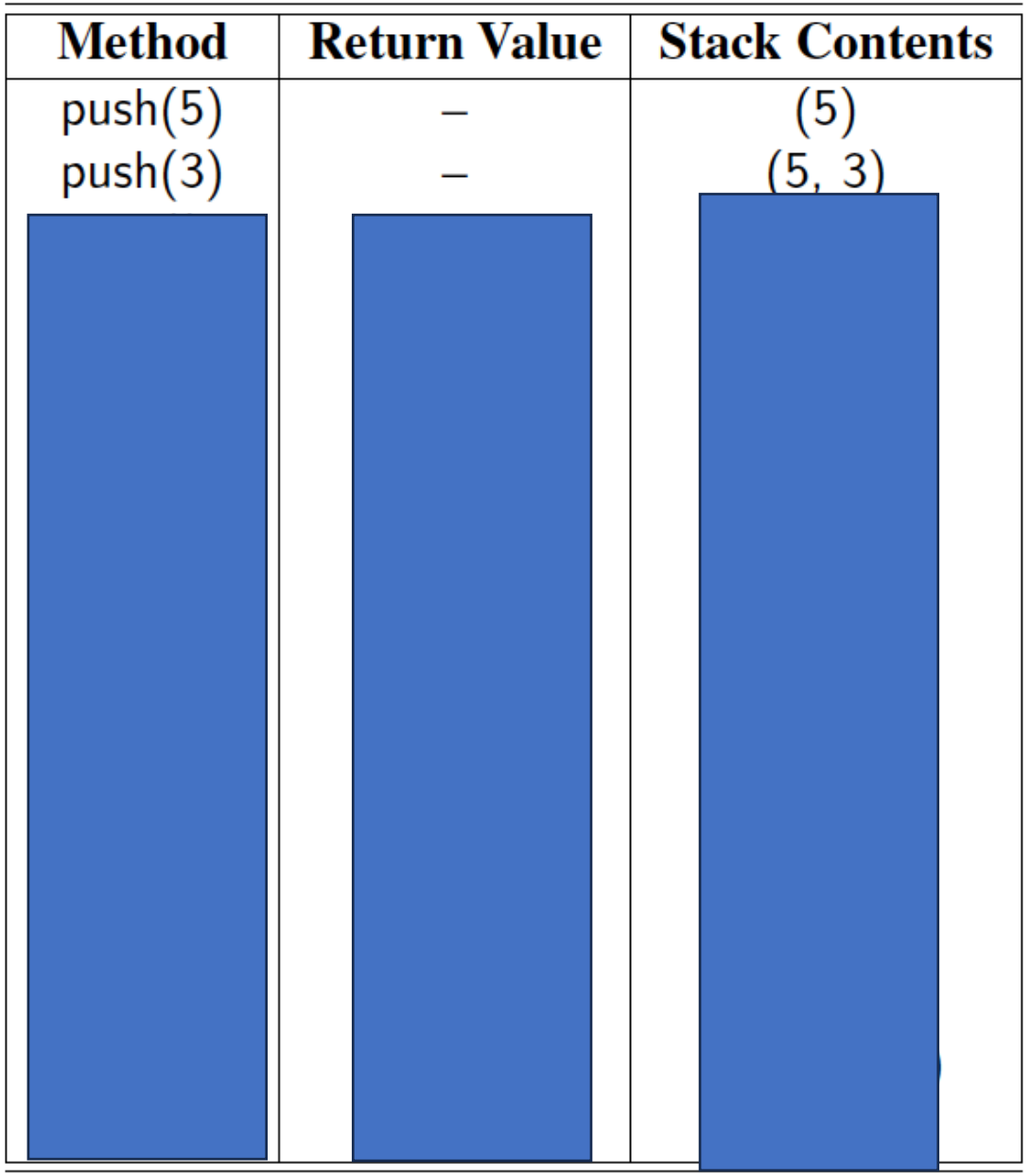
Method	Return Value	Stack Contents
push(5)		

Stack Example

Method	Return Value	Stack Contents
push(5)	–	(5)

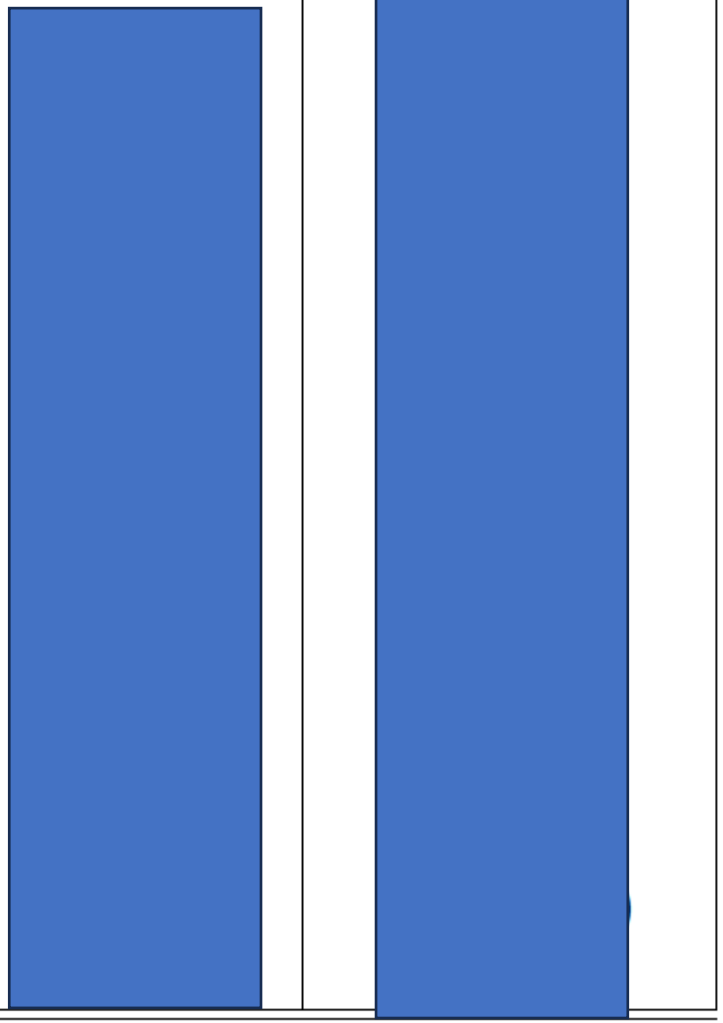
Stack Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)



Stack Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )		
pop( )		
isEmpty( )		
pop( )		
isEmpty( )		
pop( )		
push(7)		
push(9)		
top( )		
push(4)		
size( )		
pop( )		
push(6)		
push(8)		
pop( )		



Empty slide skip

Empty slide skip

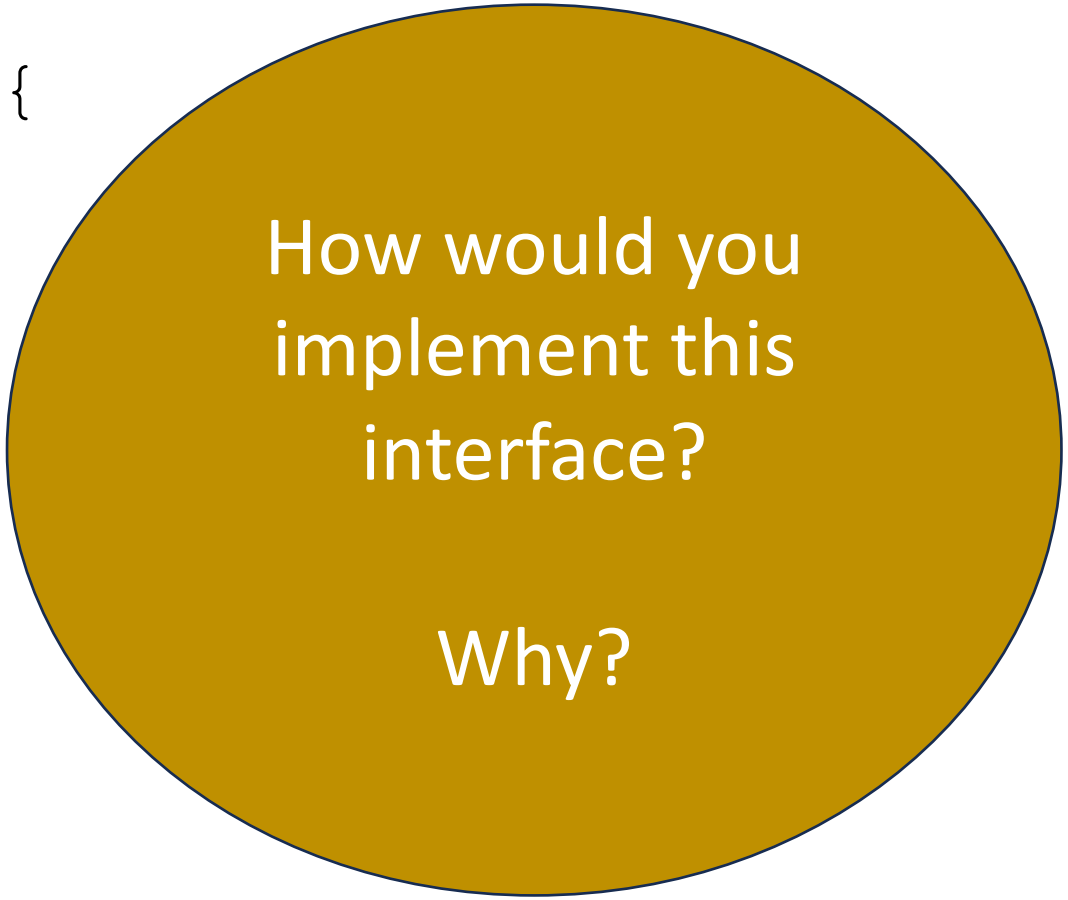
Empty slide skip

Stack Example

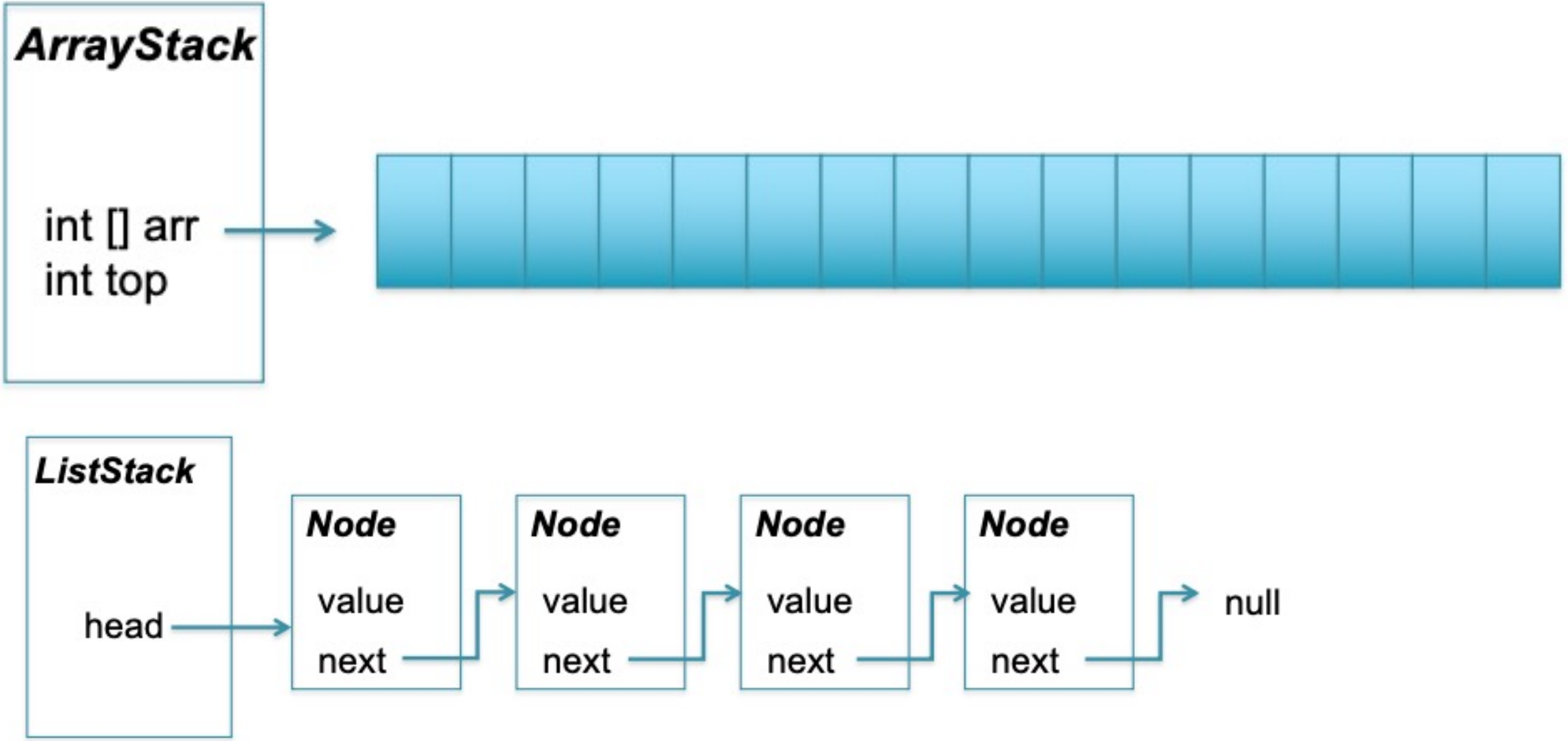
Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Stack Interface

```
public interface Stack<E> {
   int size();
   boolean isEmpty();
   E pop();
   E top();
   void push(E element);
}
```



ArrayStack vs ListStack



ArrayStack vs ListStack

Pros:

- Minimal overhead to store items:
  - Just put them in the array
- Simple implementation:
  - Pop just update one number
  - Top – quick look up

Cons:

- push() might be impossible or very slow (if copying array)

Pros:

- Unbounded
- Push, pop, top are all constant time
- Simple implementation

Cons:

- More memory intensive (need to dedicate memory for node references)
- Take computer organization course!

Performance

Let n be the number of objects in the stack

Space complexity is
- O(n)

Runtime Complexity:
- Top: O(1)
- Pop: O(1)
- Push: depends on implementation

Amortized Analysis

- Given: an array of size 100
- We insert 100 items into the array
- Insert another item:
  - Create new array
  - Copy over previous 100 elements
  - Insert 101th element

Amortized Analysis

Whats the worst case of how many operations inserting takes?

- How many operations did we perform?

201 operations

- We insert 100 items into the array

100 operations

- Insert another item:
  - Create new array
  - Copy over previous 100 elements
  - Insert 101th element

100 operations

1 operations

Amortized Analysis

- How many operations did we perform?

- We insert 100 items into the array

- Insert another item:
  - Create new array
  - Copy over previous 100 elements
  - Insert 101th element

Whats the worst case here of how many operations inserting takes?

O(n)

Amortized Analysis

Is O(n) the typical cost of inserting here?

What was the typical cost?

O(1)

Amortized cost per operation for a sequence of *n* operations is the total cost of the operations divided by *n*

Similar to an average

https://www.cs.cmu.edu/afs/cs/academic/class/15451-s10/www/lectures/lect0203.pdf

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

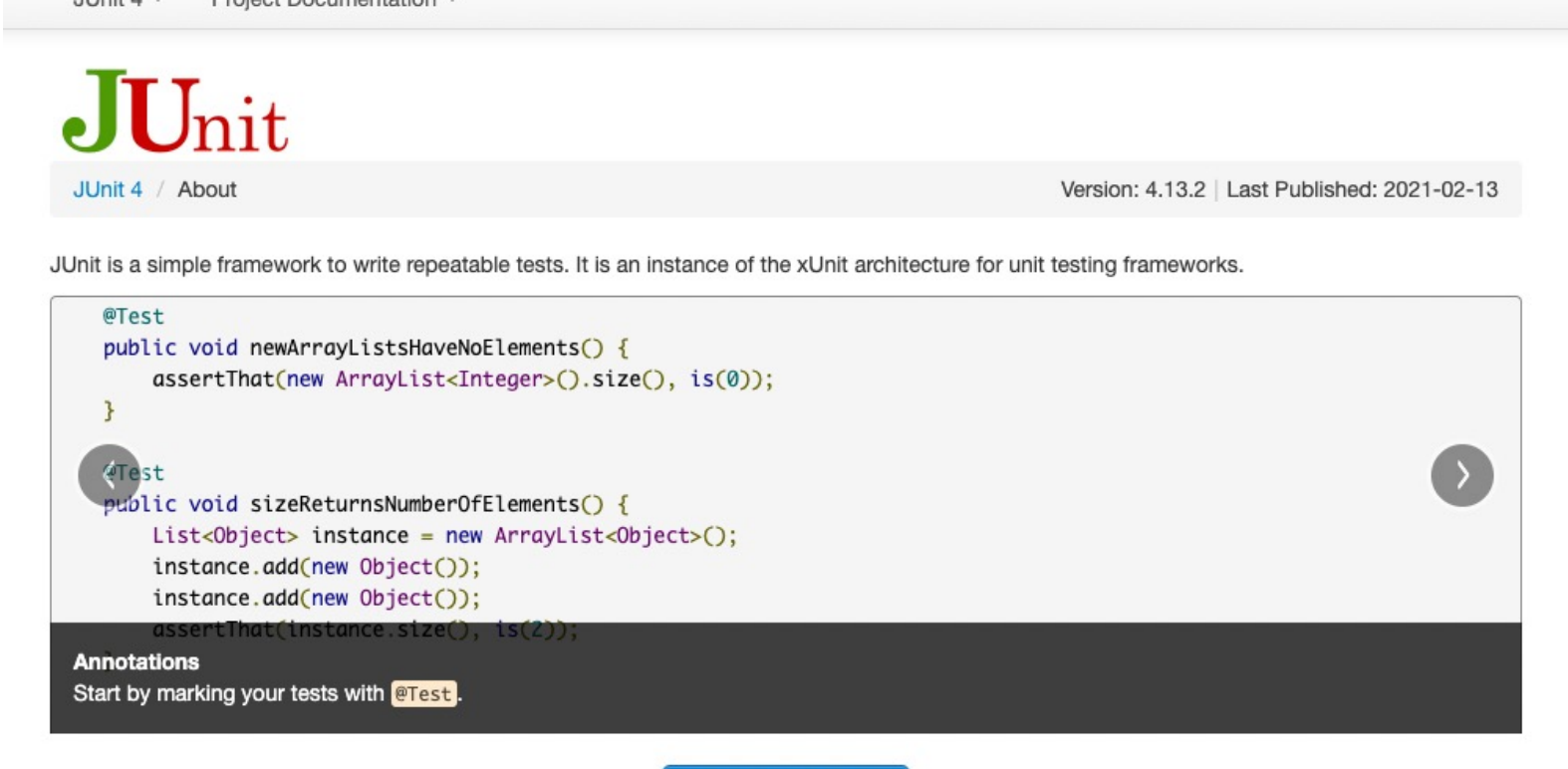
- 15 7 1 1 + – / 3 * 2 1 1 + + –
- ((15/(7–(1+1)))*3)–(2+(1+1))

Testing Stack Implementation

new stack is empty

Pushing ”hello”, then top return “hello”

Push, pop, then stack should be empty


JUnit
JUnit 4 / About
Version: 4.13.2 | Last Published: 2021-02-13
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
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
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 complext 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