CS 600.226: Data Structures

Michael Schatz

Sept 21 2018

Lecture 10. Stacks and JUnit

Agenda

1. ***Review HW2***
2. ***Introduce HW3***
3. ***Recap on Stacks***
4. ***Queues***
5. ***Deques***

Assignment 2: Due Friday Sept 21 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment02/README.md

Assignment 2: Arrays of Doom!

Out on: September 14, 2018
Due by: September 21, 2018 before 10:00 pm
Collaboration: None
Grading:
Functionality 65%
ADT Solution 20%
Solution Design and README 5%
Style 10%

Overview

The second assignment is mostly about arrays, notably our own array specifications and implementations, not just the built-in Java arrays. Of course we also once again snuck a small ADT problem in there...

Note: The grading criteria now include **10% for programming style**. Make sure you use Checkstyle with the correct configuration file from Github!

Agenda

1. ***Review HW2***
2. ***Introduce HW3***
3. ***Recap on Stacks***
4. ***Queues***
5. ***Dequeues***

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Assignment 3: Assorted Complexities

Out on: September 21, 2018
Due by: September 28, 2018 before 10:00 pm
Collaboration: None
Grading:

- Functionality 60% (where applicable)
- Solution Design and README 10% (where applicable)
- Style 10% (where applicable)
- ***Testing 10% (where applicable)***

Overview

The third assignment is mostly about sorting and how fast things go. You will also write yet another implementation of the Array interface to help you analyze how many array operations various sorting algorithms perform.

Note: The grading criteria now include 10% for unit testing. This refers to JUnit 4 test drivers, not some custom test program you hacked. The problems (on this and future assignments) will state whether you are expected to produce/improve test drivers or not.

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Problem 1: Arrays with Statistics (30%)

Your first task for this assignment is to develop a new kind of `Array` implementation that keeps track of how many read and write operations have been performed on it. Check out the `Statable` interface first, reproduced here in compressed form (be sure to use and read the full interface available in github):

```
public interface Statable {
    void resetStatistics();
    int numberOfReads();
    int numberOfWrites();
}
```

This describes what we expect of an object that can collect statistics about itself. After a `Statable` object has been "in use" for a while, we can check how many read and write operations it has been asked to perform. We can also tell it to "forget" what has happened before and start counting both kinds of operations from zero again.

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Problem 2: All Sorts of Sorts (50%)

You need to write classes implementing BubbleSort and InsertionSort for this problem. Just like our example algorithms, your classes have to implement the SortingAlgorithm interface.

All of this should be fairly straightforward once you get used to the framework. Speaking of the framework, the way you actually "run" the various algorithms is by using the PolySort.java program we've provided as well. You should be able to compile and run it without yet having written any sorting code yourself.

Here's how:

```
$ java PolySort 4000 <random.data

 Algorithm            Sorted?  Size        Reads       Writes      Seconds

 Null Sort            false    4,000       0           0           0.000007
 Gnome Sort           true     4,000       32,195,307  8,045,828   0.243852
 Selection Sort       true     4,000       24,009,991  7,992       0.252085
```

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

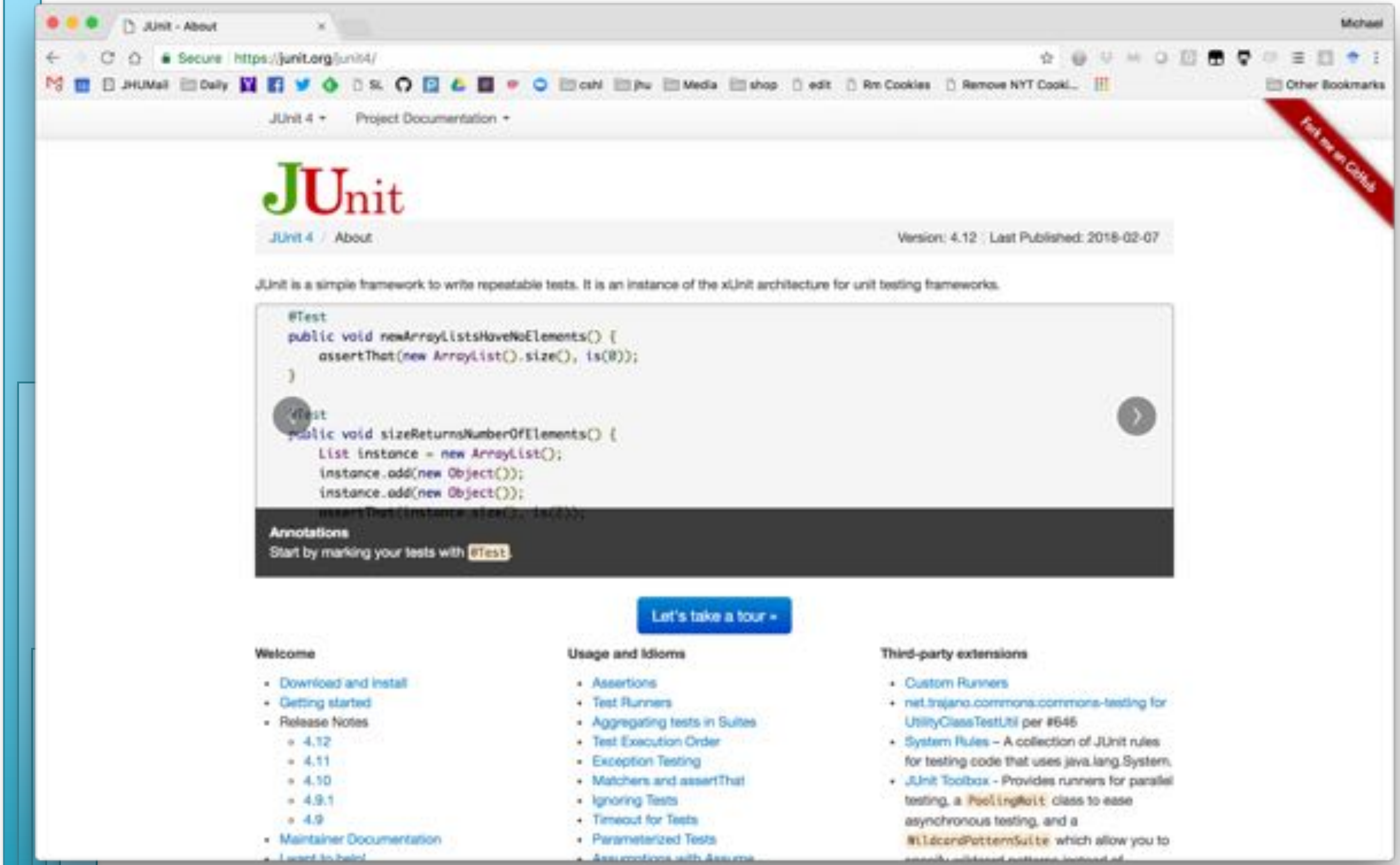
Problem 3: Analysis of Selection Sort (20%)

Your final task for this assignment is to analyze the following selection sort algorithm theoretically (without running it) in detail (without using O-notation).

Here's the code, and you must analyze exactly this code (the line numbers are given so you can refer to them in your writeup for this problem):

```
 1: public static void selectionSort(int[] a) {
 2:     for (int i = 0; i < a.length - 1; i++) {
 3:         int min = i;
 4:         for (int j = i + 1; j < a.length; j++) {
 5:             if (a[j] < a[min]) {
 6:                 min = j;
 7:             }
 8:         }
 9:         int t = a[i]; a[i] = a[min]; a[min] = t;
10:     }
11: }
```

Introducing JUnit



JUnit 4 / About

Version: 4.12 | Last Published: 2018-02-07

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().size(), is(0));
}

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

Annotations

Start by marking your tests with `@Test`.

Welcome

- Download and install
- Getting started
- Release Notes
  - 4.12
  - 4.11
  - 4.10
  - 4.9.1
  - 4.9
- Maintainer Documentation
- I want to help!

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

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

TestSimpleArray.java

@BeforeClass causes the method to be run once before any of the test methods in the class

```
import org.junit.Test;
import org.junit.BeforeClass;
import static org.junit.Assert.assertEquals;

public class TestSimpleArray {
    static Array<String> shortArray;

    @BeforeClass
    public static void setupArray() throws LengthException {
       shortArray = new SimpleArray<String>(10, "Bla");
    }

    @Test
    public void newArrayLengthGood() throws LengthException {
        assertEquals(10, shortArray.length());
    }

    @Test
    public void newArrayInitialized() throws LengthException, IndexException {
        for (int i = 0; i < shortArray.length(); i++) {
            assertEquals("Bla", shortArray.get(i));
        }
    }

    @Test(expected=IndexException.class)
    public void IndexDetected() throws IndexException {
        shortArray.put(shortArray.length(), "Paul");
    }
}
```

Check the results with assertEquals, or listing the expected exception

Running JUnit

```
// Step 0: Download junit-4.12.jar and hamcrest-core-1.3.jar
// Jar files are bundles of java classes ready to run

// Step 1: Compile your code as usual and checkstyle
$ javac –Xlint:all SimpleArray.java
$ check SimpleArray.java

// Step 2: Compile tests, but not checkstyle for these :)
$ javac -cp .:junit-4.12.jar -Xlint:all TestSimpleArray.java

// Step 3: Run Junit on your TestProgram. Notice that
org.junit.runner.JUnitCore is the main code we run, and
TestSimpleArray is just a parameter to it
$ java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar \
     org.junit.runner.JUnitCore TestSimpleArray
JUnit version 4.12
...
Time: 0.011

OK (3 tests)

// Hooray, everything is okay!
```

Hint: save commands to a file!
chmod +x tester.sh
./tester.sh

-cp sets the class path. This tells Java where to find the relevant code needed for compiling and running

Guidelines

1. ***Every Method should be tested for correct outputs***
    - Try simple and complex examples (different lengths of arrays, etc)
    - Private methods can be tested implicitly, but the entire public interface should be evaluated

2. ***Every exception and error condition should also be tested***
    - This is how the ADT contract will be enforced

3. ***Write the test cases first, that way you will know when you are done***

Stacks

Stacks

Stacks are very simple but surprisingly useful data structures for storing a collection of information

- Any number of items can be stored, but you can only manipulate the top of the stack:
    - ***Push***: adds a new element to the top
    - ***Pop***: takes off the top element
    - ***Top***: Lets you peek at top element's value without removing it from stack

Many Applications

- In hardware call stack
- Memory management systems
- Parsing arithmetic instructions:

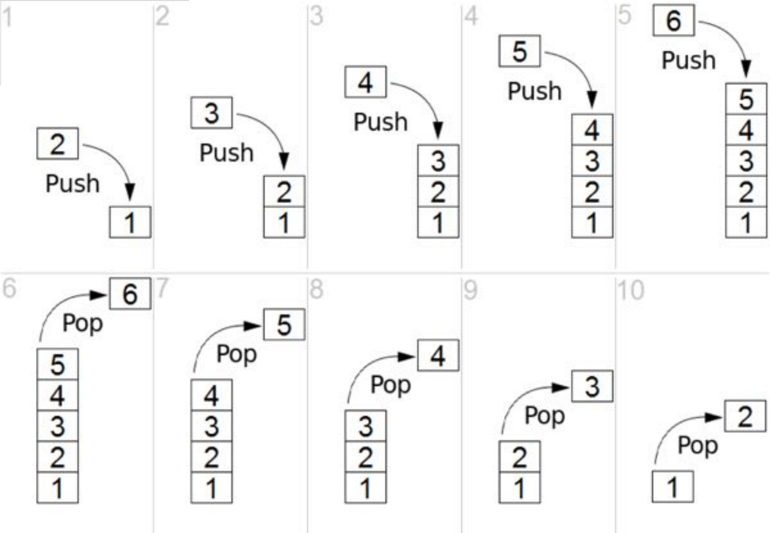
  ((x+3) / (x+9)) * (42 * sin(x))
- Back-tracing, such as searching within a maze

Stack Interface

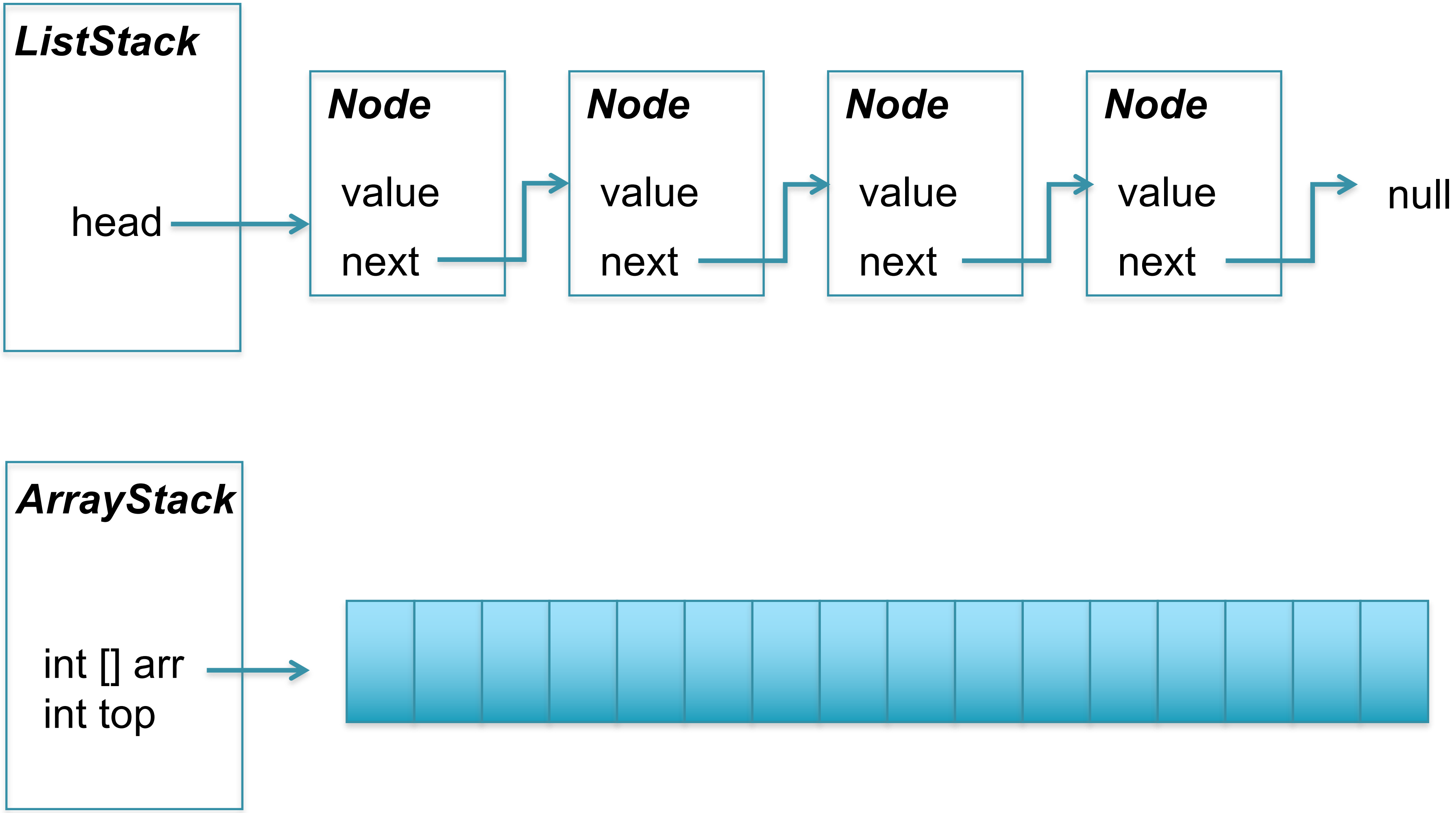
```
public interface Stack<T> {
    // checks if empty
    boolean empty();

    // peeks at top value without removing
    T top() throws EmptyException;

    // removes top element
    void pop() throws EmptyException;

    // adds new element to top of stack
    void push(T t);
}
```

How would you implement this interface?

Why?



ListStack vs ArrayStack



ListStack

head

Node

value

next

Node

value

next

Node

value

next

Node

value

next

null

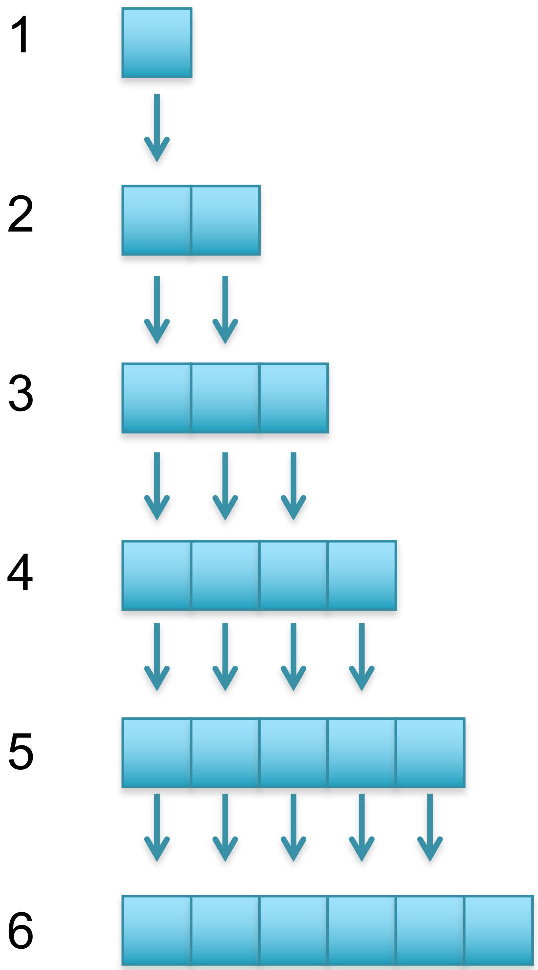
ArrayStack

int [] arr

int top

ArrayStack Growing

If the array size starts at 1, how expensive will it be to grow to 1M if we copy one element at a time?



1M push()s will require a total of

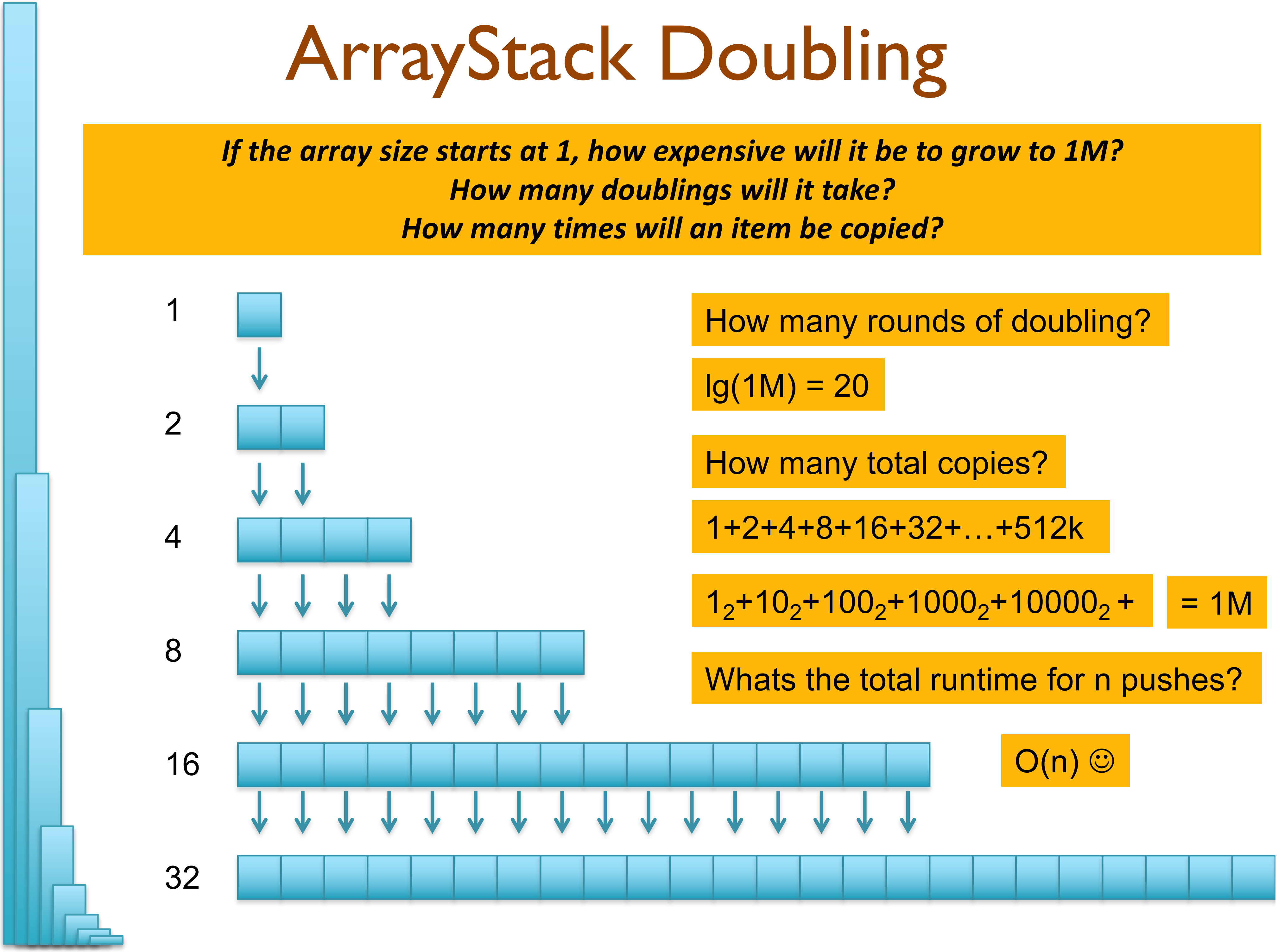
1+2+3+4+5+6+…+999,999 copies

= 0.5MM steps!

O(n^2) performance ☹

ArrayStack Doubling

If the array size starts at 1, how expensive will it be to grow to 1M?
How many doublings will it take?
How many times will an item be copied?



How many rounds of doubling?

lg(1M) = 20

How many total copies?

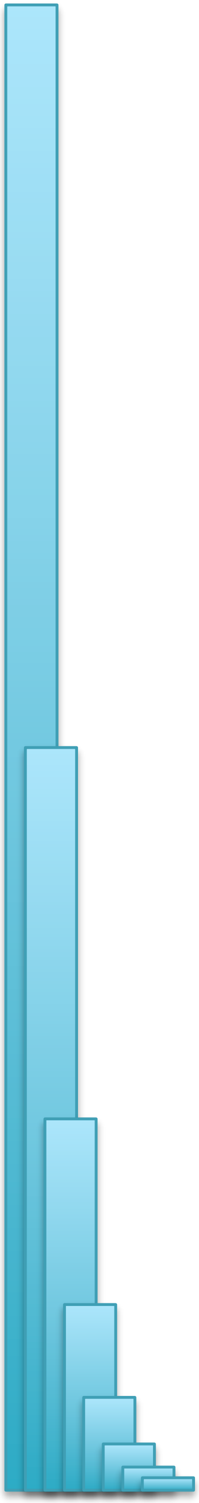
1+2+4+8+16+32+…+512k

$1_2+10_2+100_2+1000_2+10000_2+$ = 1M

Whats the total runtime for n pushes?

O(n) ☺

Sums of Powers of Two



1	2^0	0000 0000 0000 0000 0001
+ 2	$+ 2^1$	+ 0000 0000 0000 0000 0010
+ 4	$+ 2^2$	+ 0000 0000 0000 0000 0100
+ 8	$+ 2^3$	+ 0000 0000 0000 0000 1000
+ 16	$+ 2^4$	+ 0000 0000 0000 0001 0000
+ 32	$+ 2^5$	+ 0000 0000 0000 0010 0000
+ 64	$+ 2^6$	+ 0000 0000 0000 0100 0000
…	…	…
+ 524,288	$+ 2^{19}$	+ 1000 0000 0000 0000 0000
1,048,576-1	$2^{20} - 1$	1111 1111 1111 1111 1111
1,048,575		

Amortized Analysis

The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n

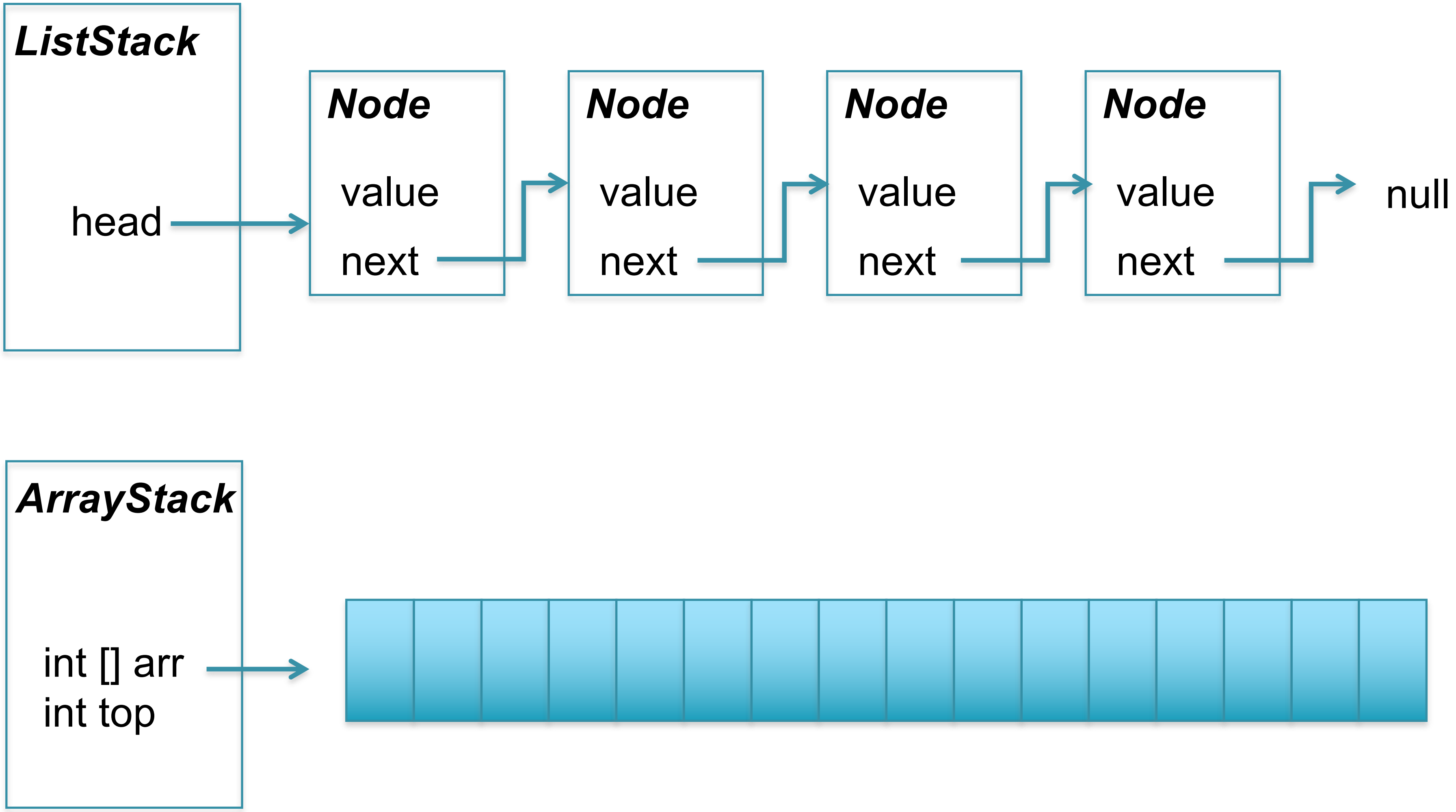
Example: If we have 100 operations at cost 1, followed by one operation at cost 100, the amortized cost per operation is 200/101 < 2.
Note the worst case operation analysis yields 100

Amortized cost analysis is helpful because many important data structures occasionally incur a large cost as they perform some kind of rebalancing or improvement of their internal state, but those expensive operations cannot occur too frequently. In this case, amortized analysis can give a much tighter bound on the true cost of using the data structure than a standard worst-case-per-operation bound.

Note that even though the definition of amortized cost is simple, analyzing it will often require some thought.

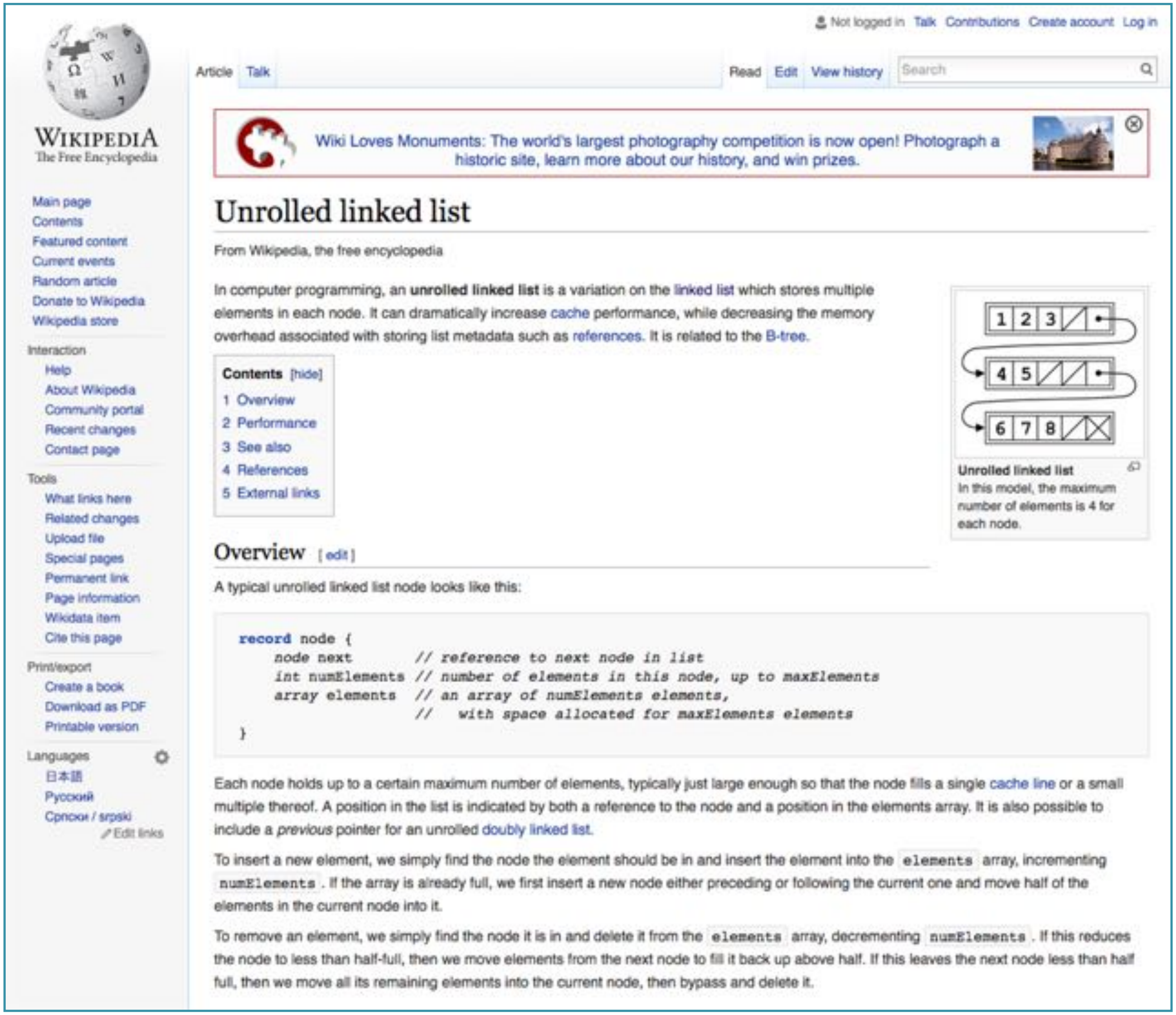
http://www.cs.cmu.edu/afs/cs/academic/class/15451-s10/www/lectures/lect0203.pdf

ListStack vs ArrayStack



Would you ever use lists and stacks together?

Unrolled Linked List



Wiki Loves Monuments: The world's largest photography competition is now open! Photograph a historic site, learn more about our history, and win prizes.

Unrolled linked list

From Wikipedia, the free encyclopedia

In computer programming, an **unrolled linked list** is a variation on the linked list which stores multiple elements in each node. It can dramatically increase cache performance, while decreasing the memory overhead associated with storing list metadata such as references. It is related to the B-tree.

Unrolled linked list
In this model, the maximum number of elements is 4 for each node.

Overview [edit]

A typical unrolled linked list node looks like this:

```
record node {
    node next       // reference to next node in list
    int numElements // number of elements in this node, up to maxElements
    array elements  // an array of numElements elements,
                    //   with space allocated for maxElements elements
}
```

Each node holds up to a certain maximum number of elements, typically just large enough so that the node fills a single cache line or a small multiple thereof. A position in the list is indicated by both a reference to the node and a position in the elements array. It is also possible to include a *previous* pointer for an unrolled doubly linked list.

To insert a new element, we simply find the node the element should be in and insert the element into the `elements` array, incrementing `numElements`. If the array is already full, we first insert a new node either preceding or following the current one and move half of the elements in the current node into it.

To remove an element, we simply find the node it is in and delete it from the `elements` array, decrementing `numElements`. If this reduces the node to less than half-full, then we move elements from the next node to fill it back up above half. If this leaves the next node less than half full, then we move all its remaining elements into the current node, then bypass and delete it.

Queues

Stacks versus Queues

LIFO: Last-In-First-Out

Add to top +
Remove from top

FIFO: First-In-First-Out

Add to back +
Remove from front

Queue Applications

Whenever a resource is shared among multiple jobs:

- accessing the CPU
- accessing the disk
- Fair scheduling (ticketmaster, printing)

Whenever data is transferred asynchronously (data not necessarily received at same rate as it is sent):

- Sending data over the network
- Working with UNIX pipes:
  - ./slow | ./fast | ./medium

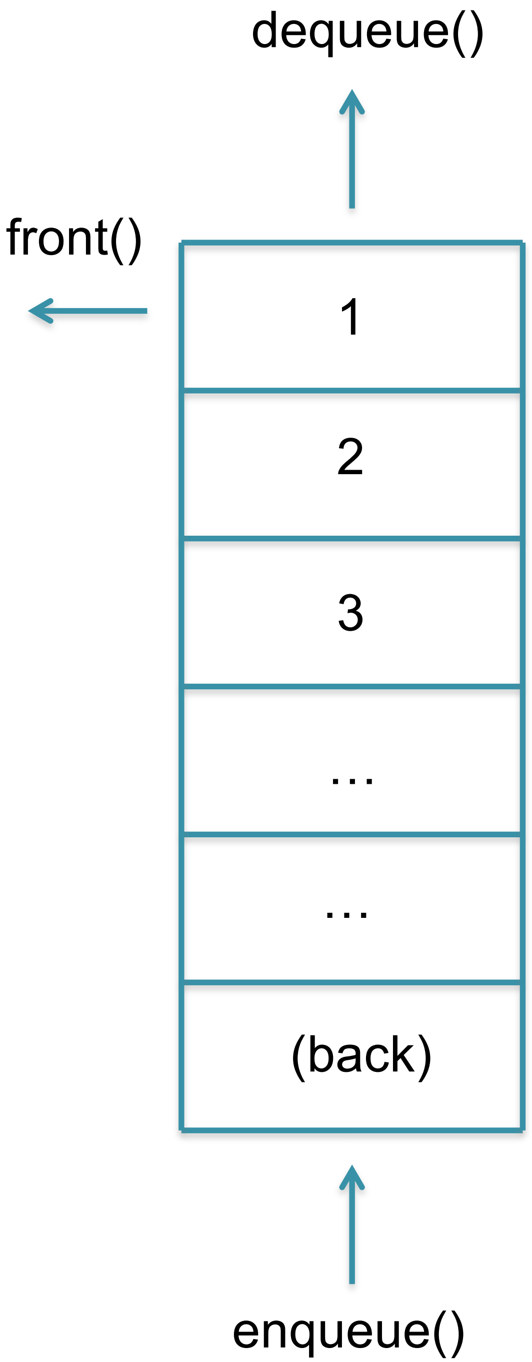
Also many applications to searching graphs (see 3-4 weeks)

FIFO: First-In-First-Out

Add to back +
Remove from front

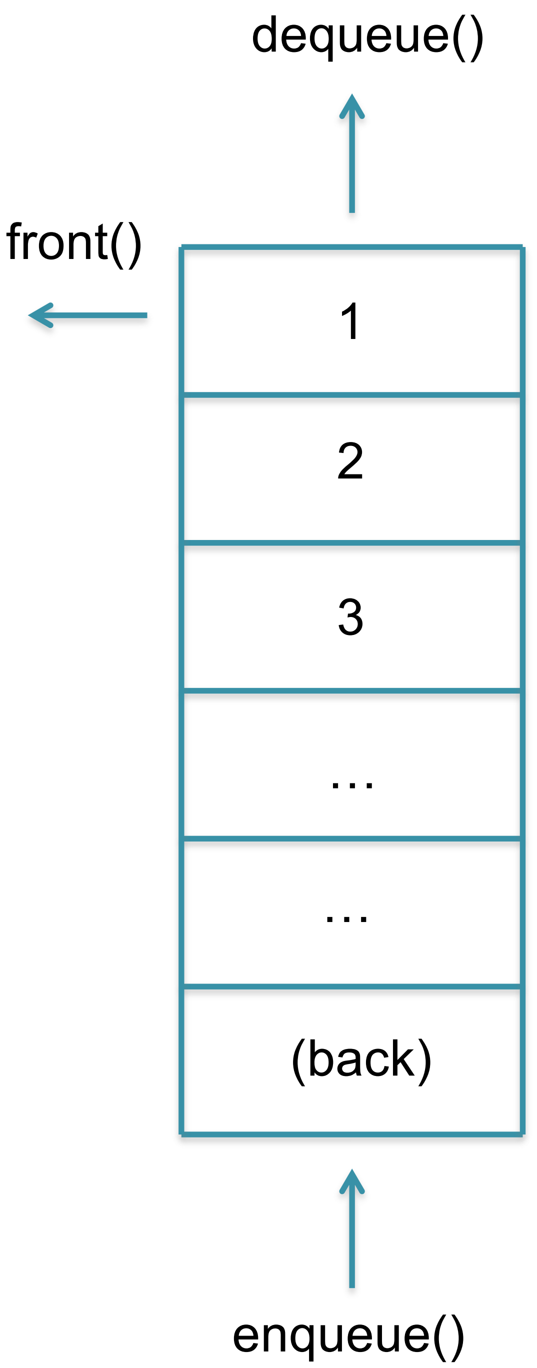
Queue ADT

Bonus question on Assignment 4



Queue Interface

```
public interface Queue<T> {
    /**
     * Test if queue is empty.
      * @return True if the queue is empty
     */
    boolean empty();

    /**
     *  Access front element of queue.
      *  @return Top element of the queue.
     *  @throws EmptyException for empty queue.
     */
    T front() throws EmptyException;

    /**
     *  Remove element at front of queue.
      *  @throws EmptyException for empty queue.
     */
    void dequeue() throws EmptyException;

    /**
     *  Insert new element at back of queue.
     *  @param t Element to enqueue.
     */
    void enqueue(T t);
}
```



dequeue()

front()

1

2

3

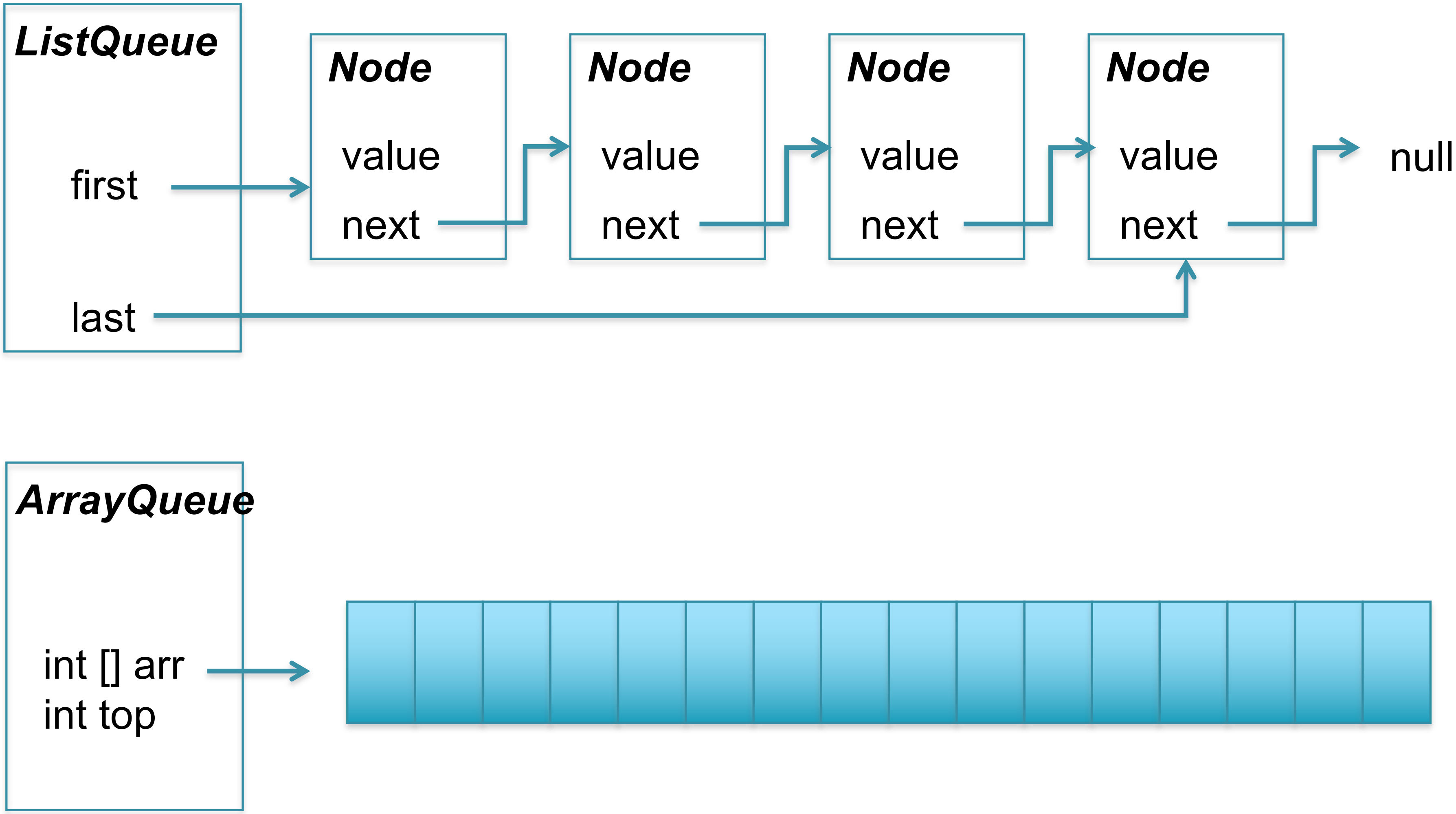
…

…

(back)

enqueue()

ListQueue vs ArrayQueue



ListQueue

first

last

Node

value

next

Node

value

next

Node

value

next

Node

value

next

null

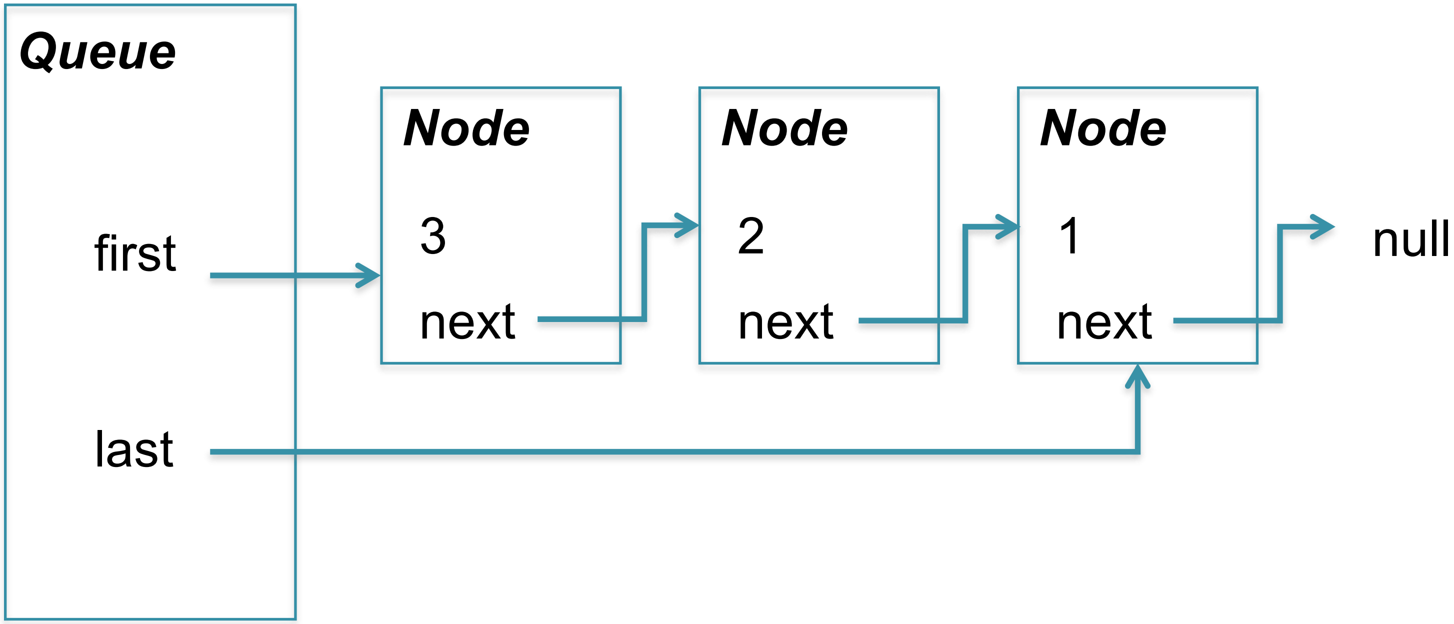
ArrayQueue

int [] arr

int top

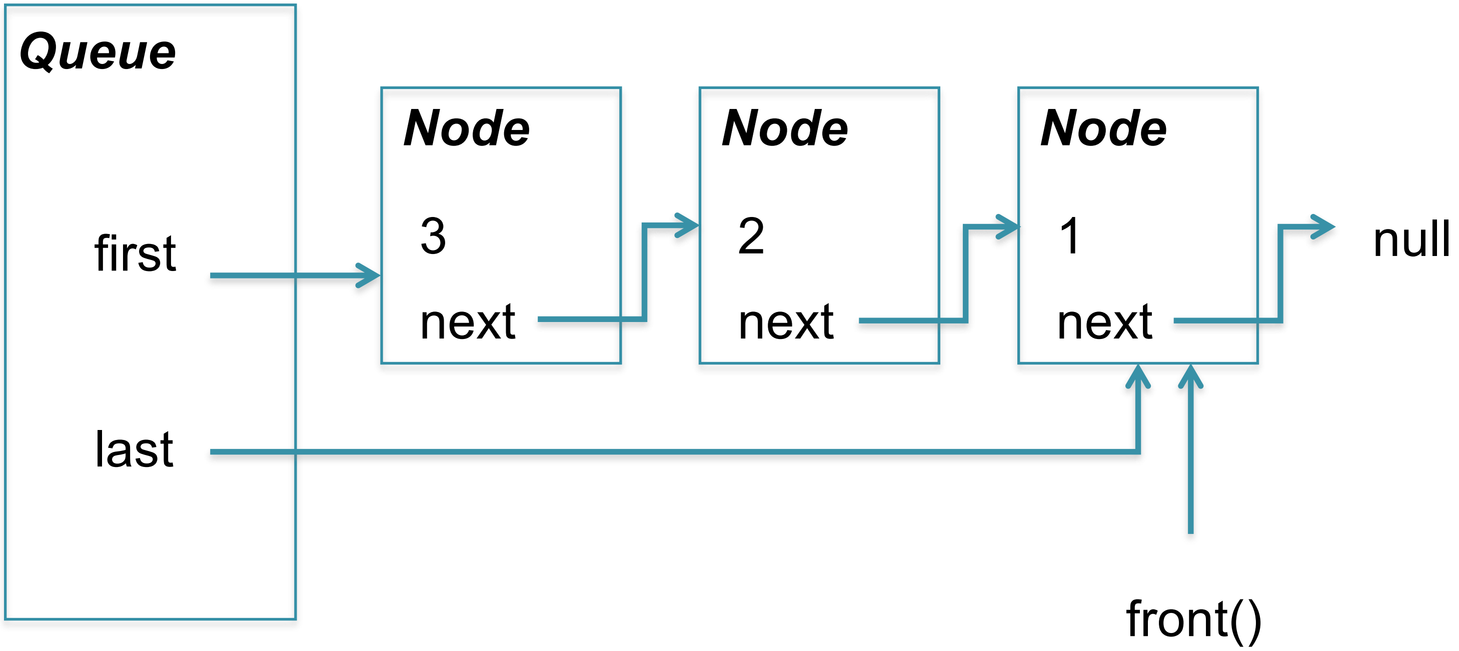
Many of the same tradeoffs as ListStack vs ArrayStack

List Queue



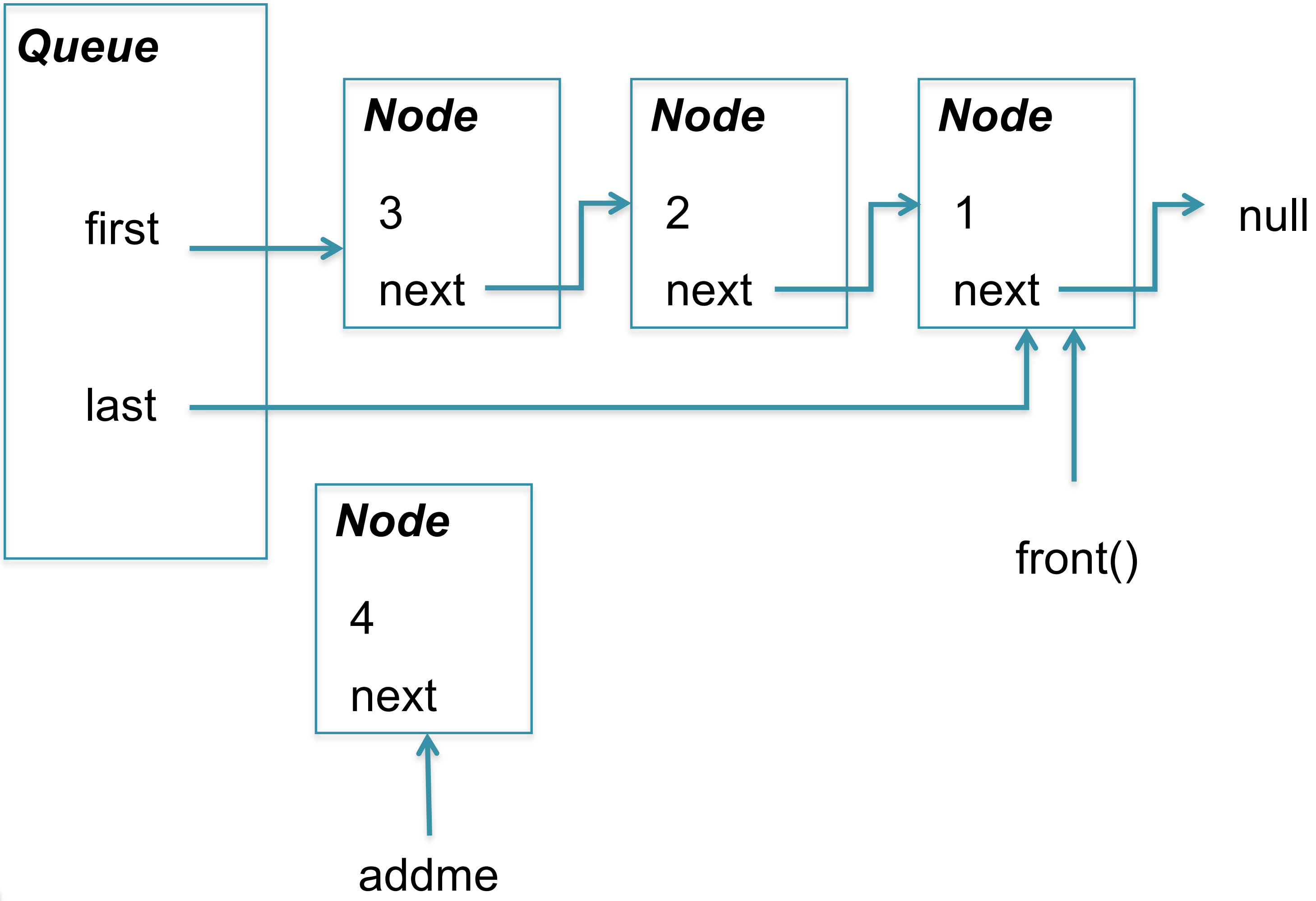
Where should we enqueue?
Is the next node to dequeue 1 or 3?

Enqueue first, Dequeue last

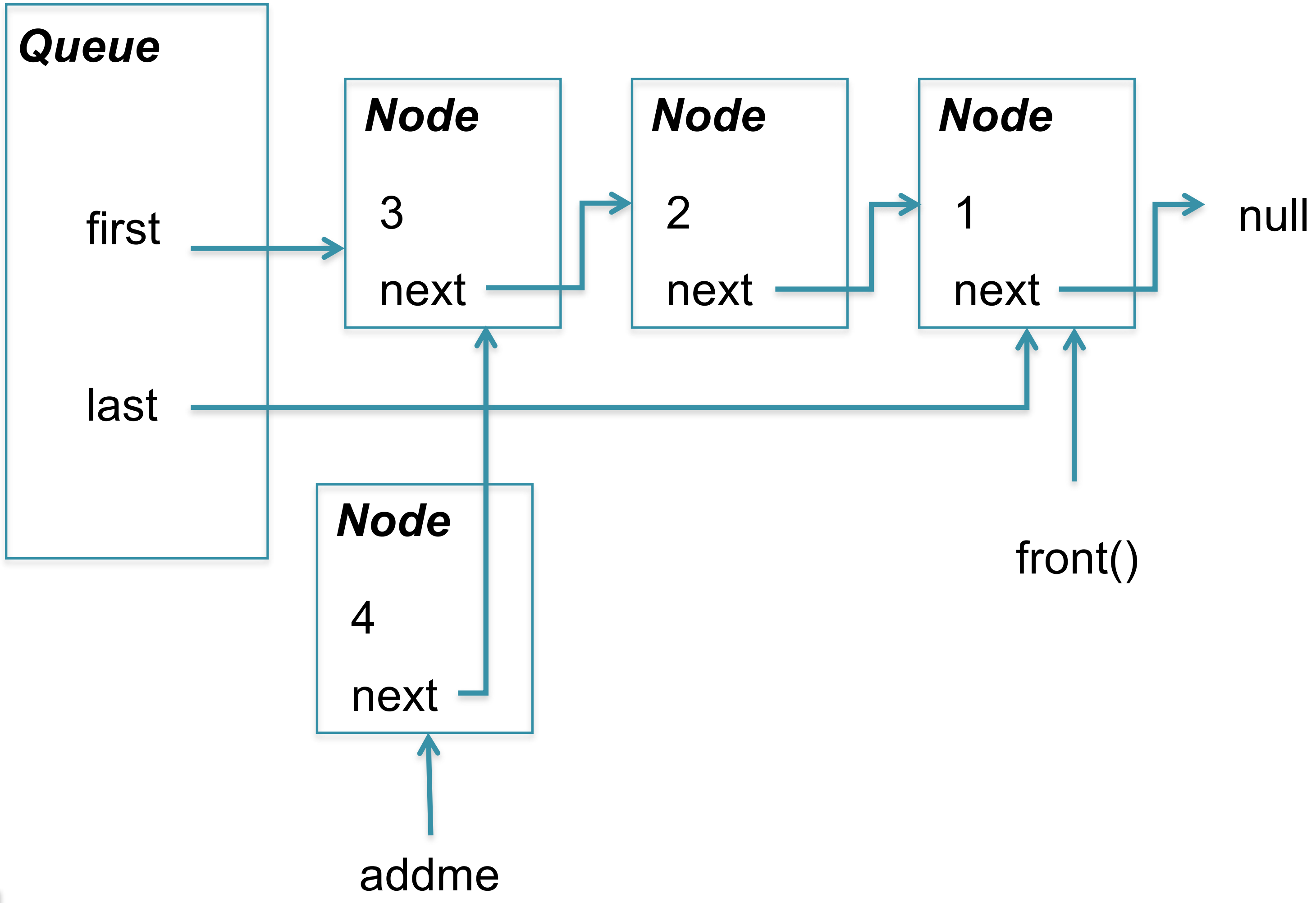


How to add a new element at first?

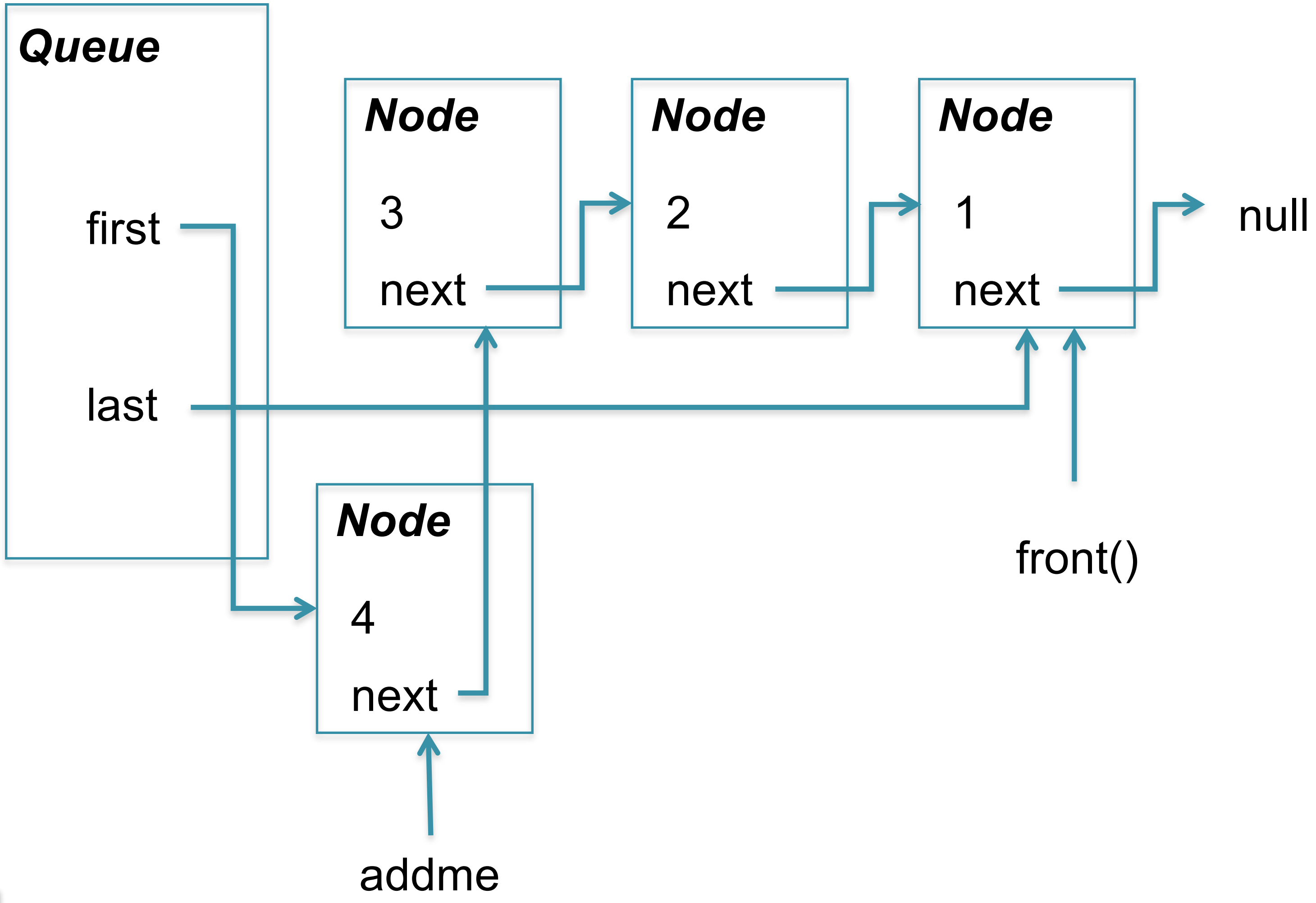
Enqueue first, Dequeue last



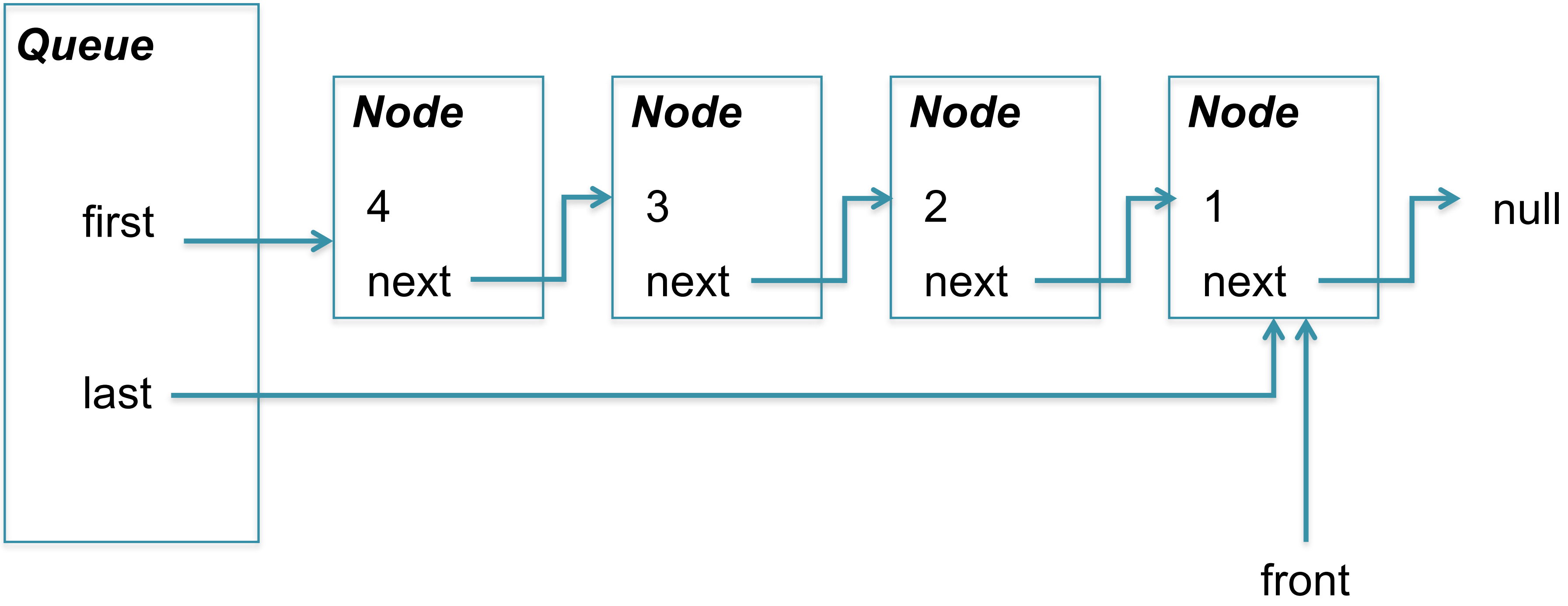
Enqueue first, Dequeue last



Enqueue first, Dequeue last

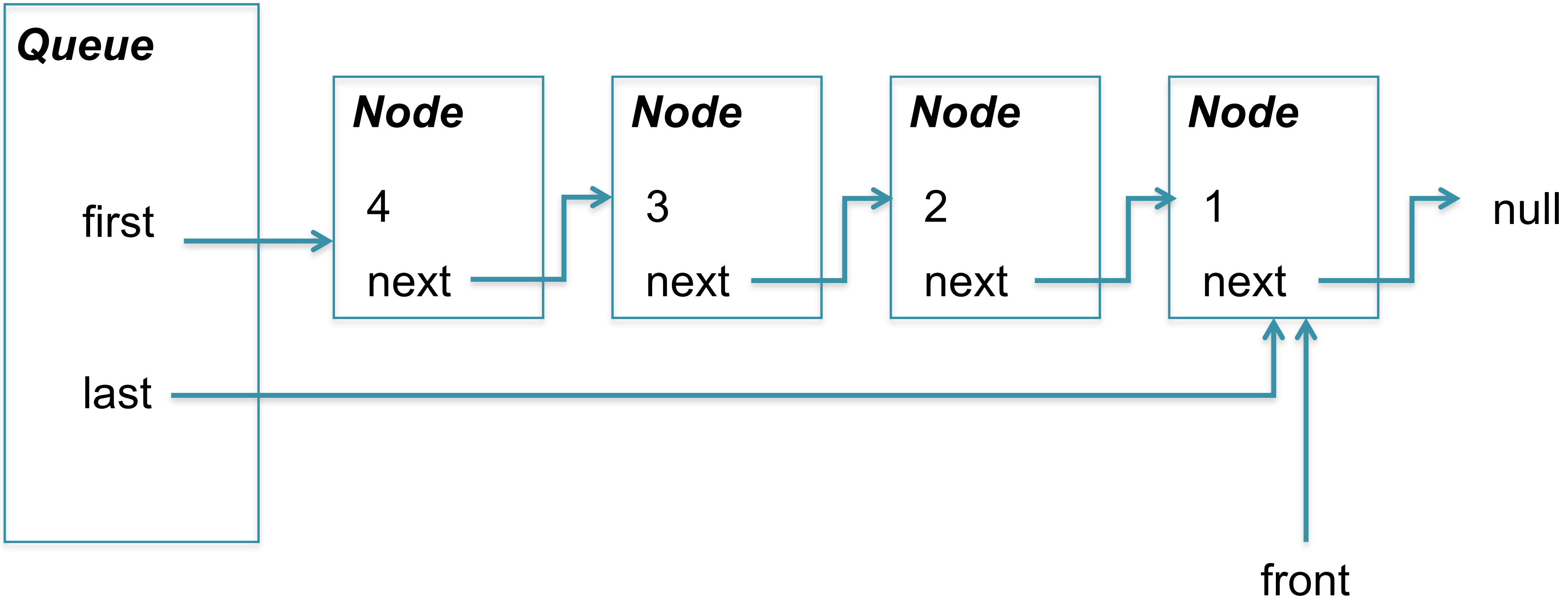


Enqueue first, Dequeue last



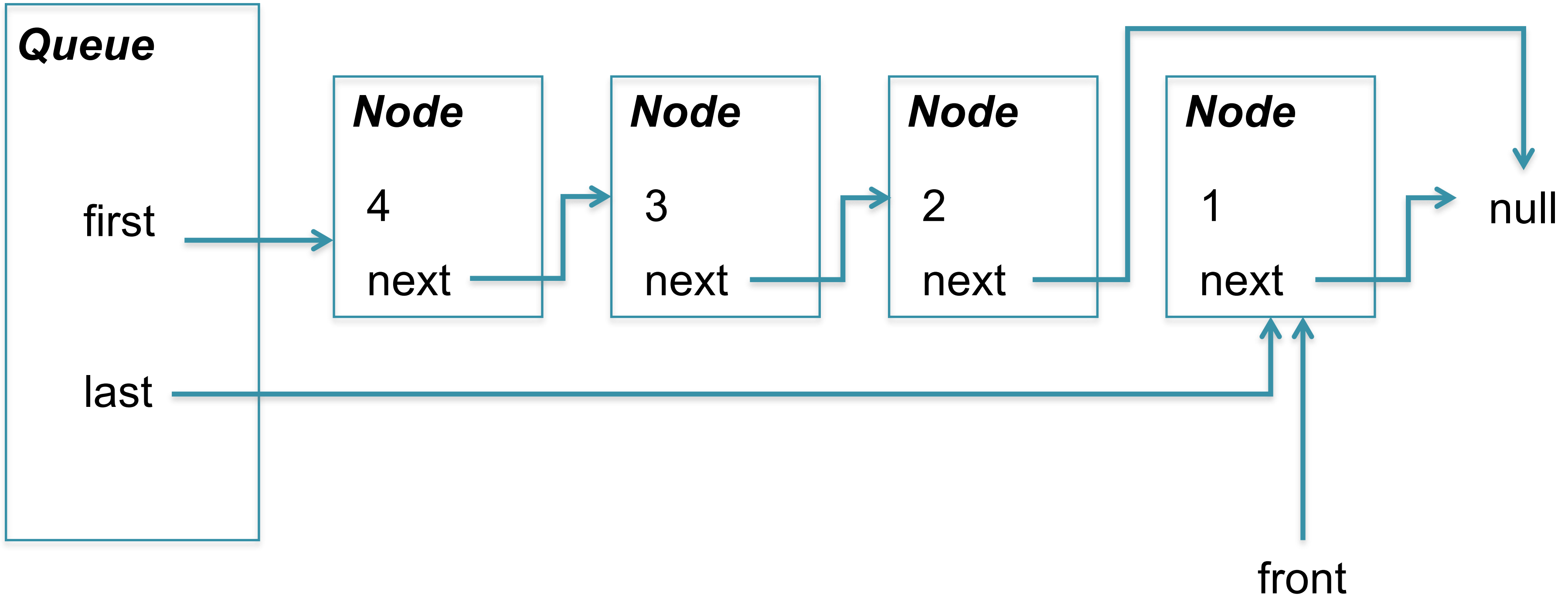
Enqueue is an O(1) operation ☺

Enqueue first, Dequeue last

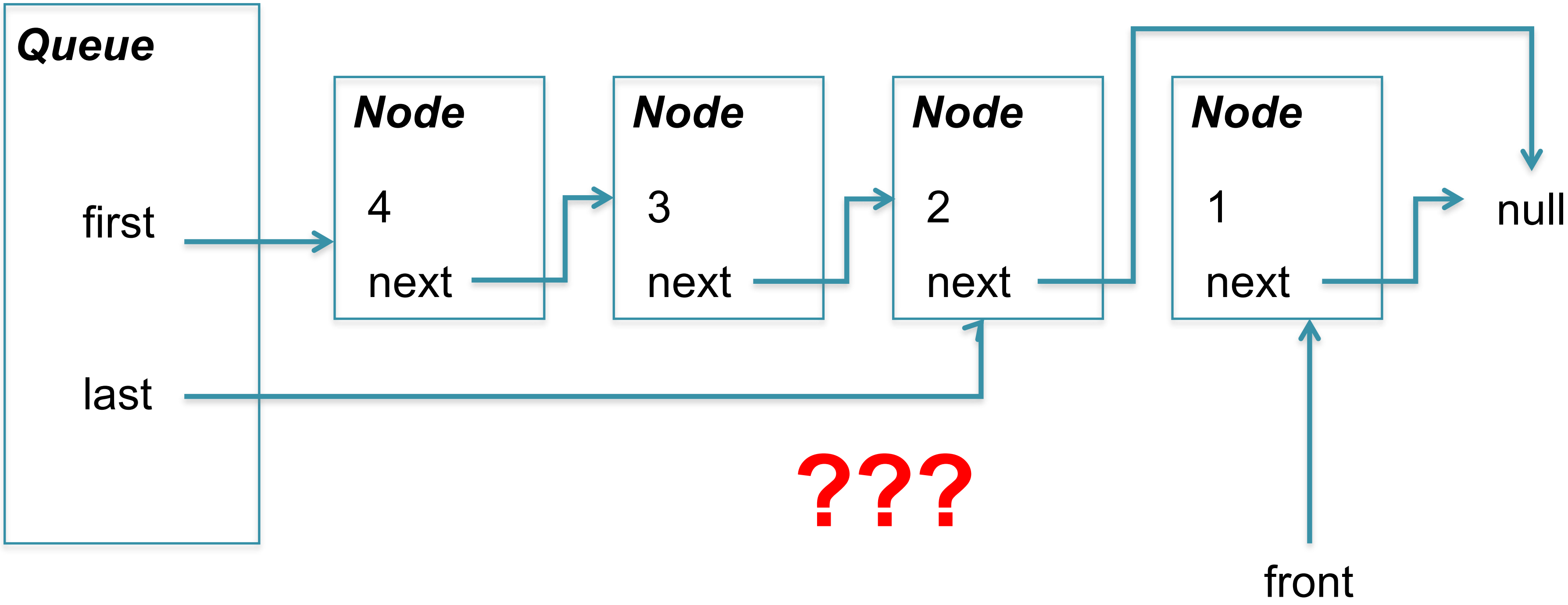


How to remove an element at front?

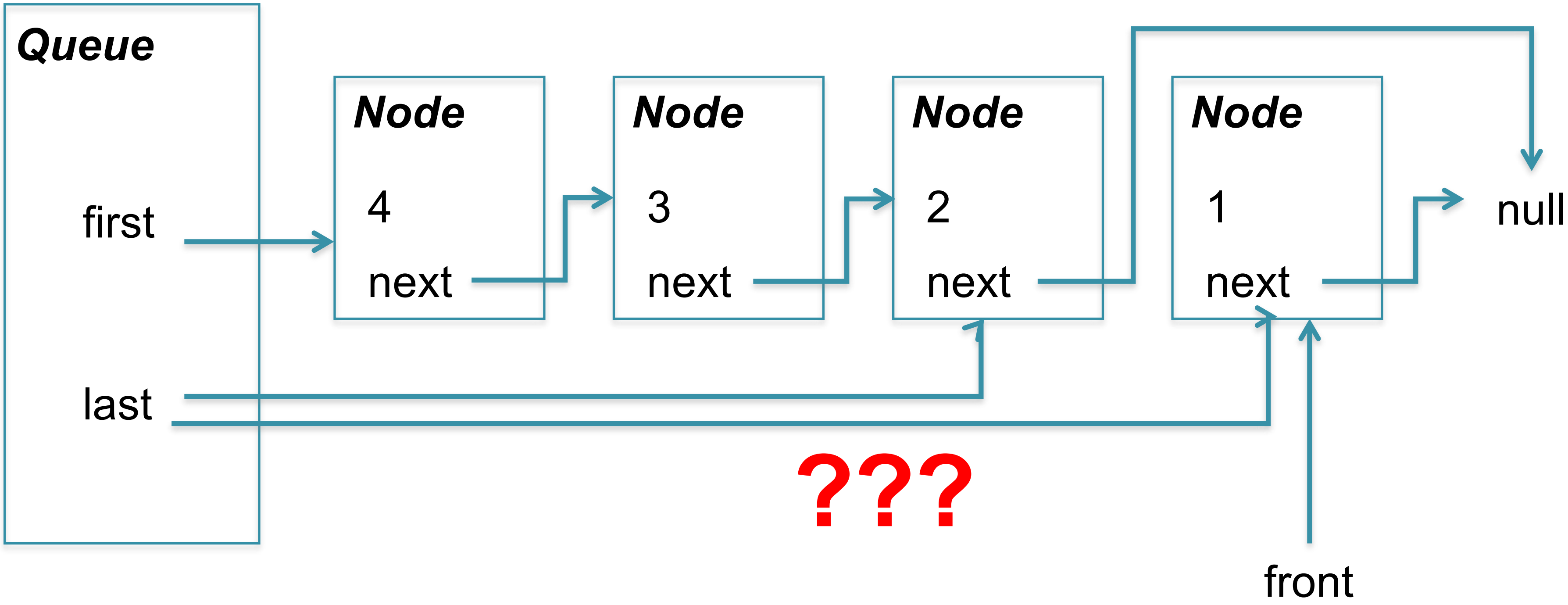
Dequeue at last



Dequeue at last

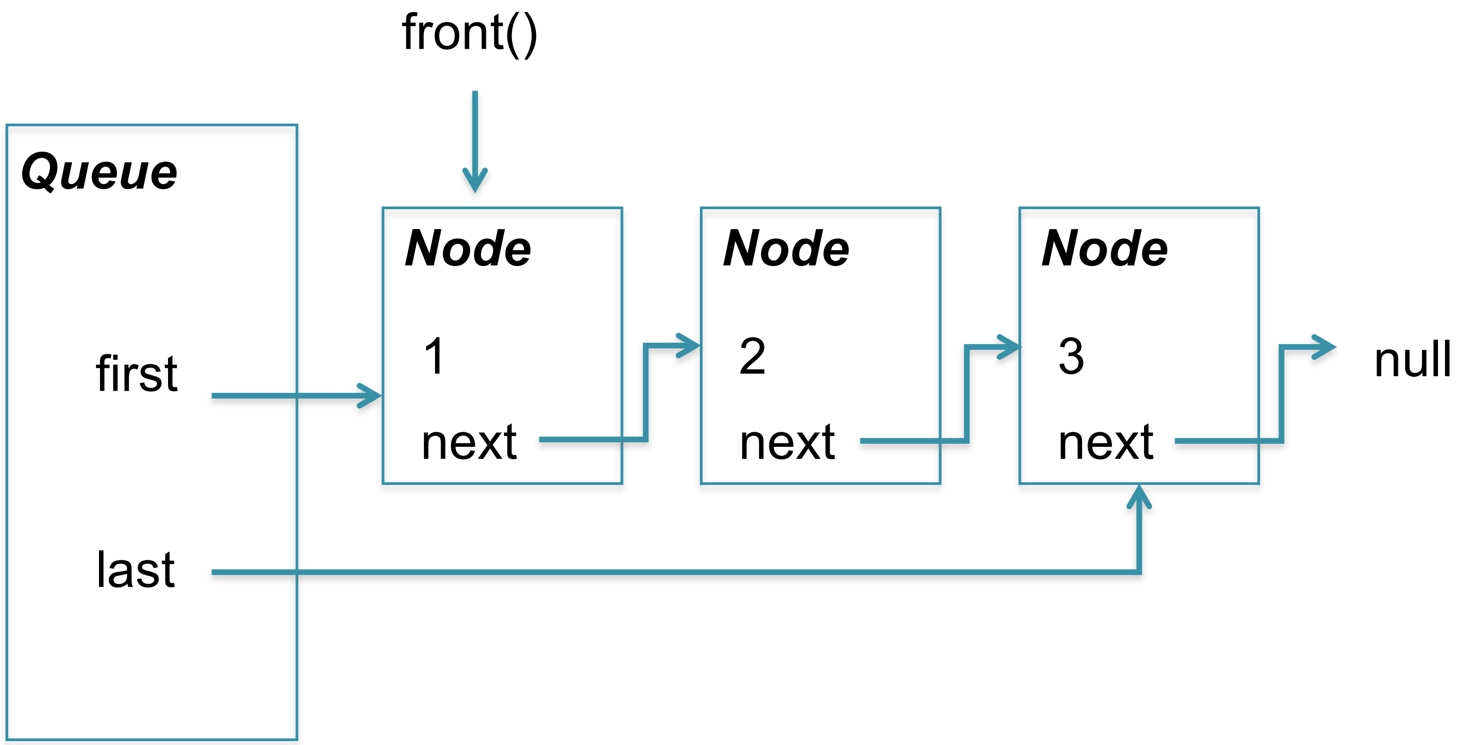


Dequeue at last



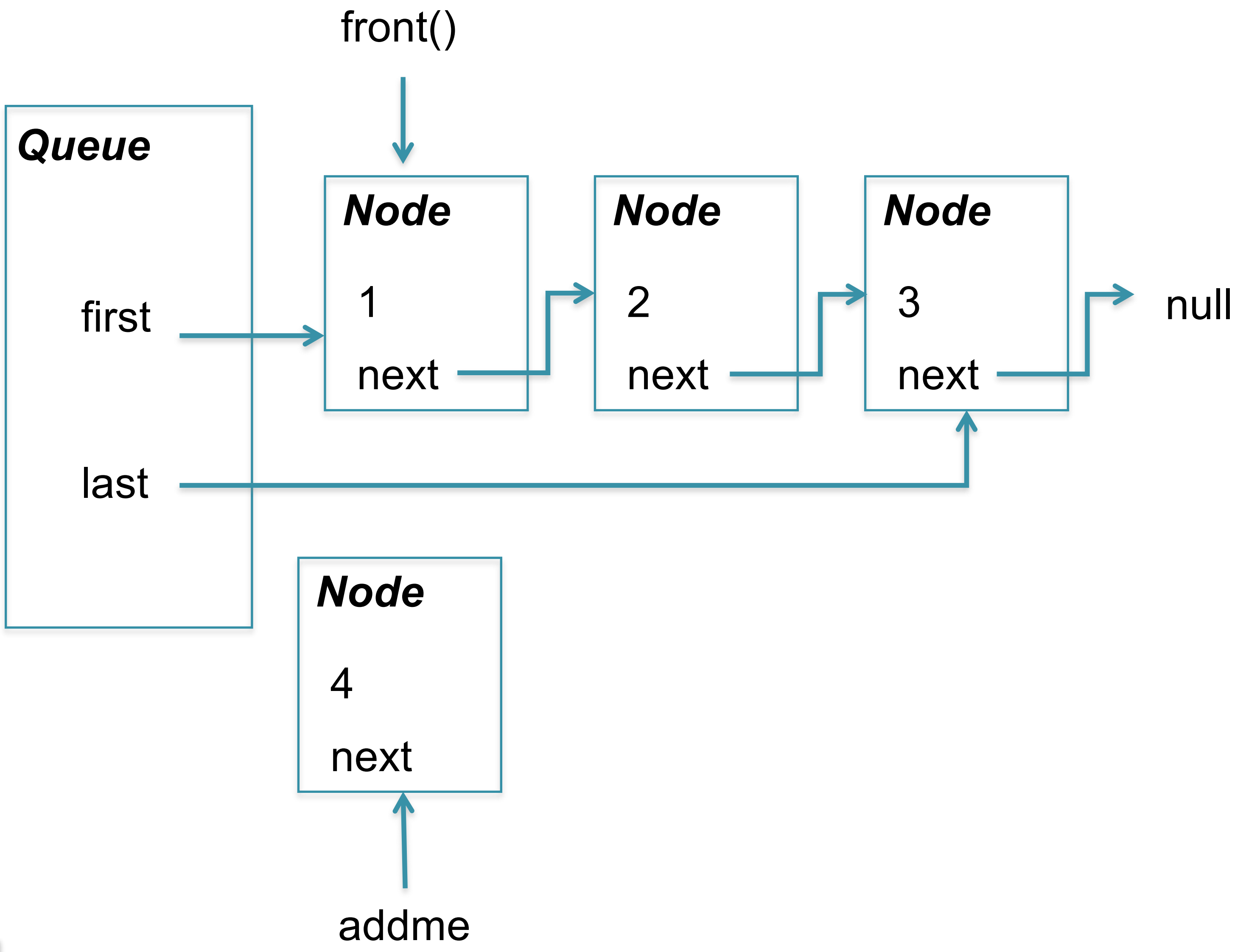
Oops, just made dequeue an O(n) operation
How might you address this?

Enqueue last, Dequeue first

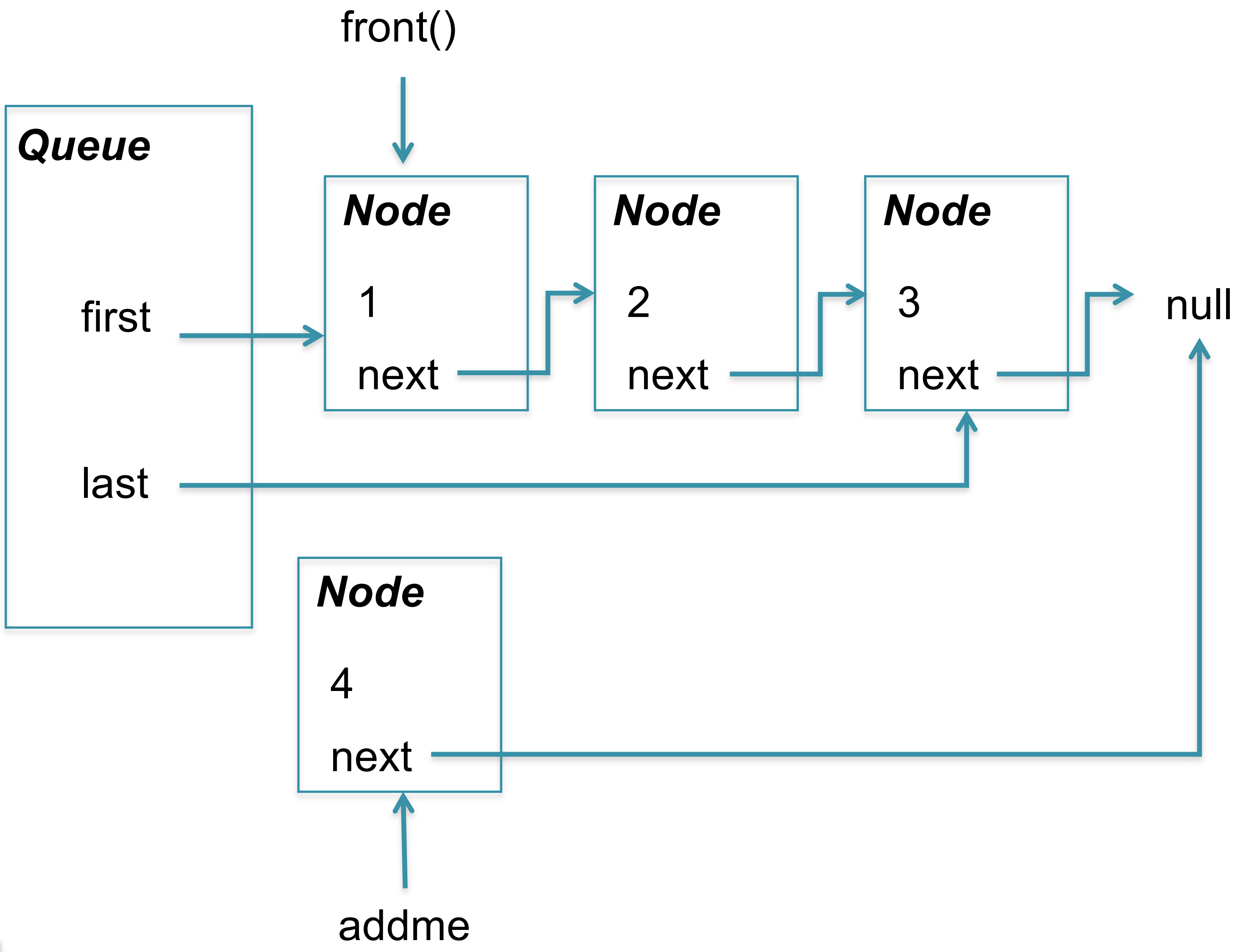


Lets try inserting at last and removing from first

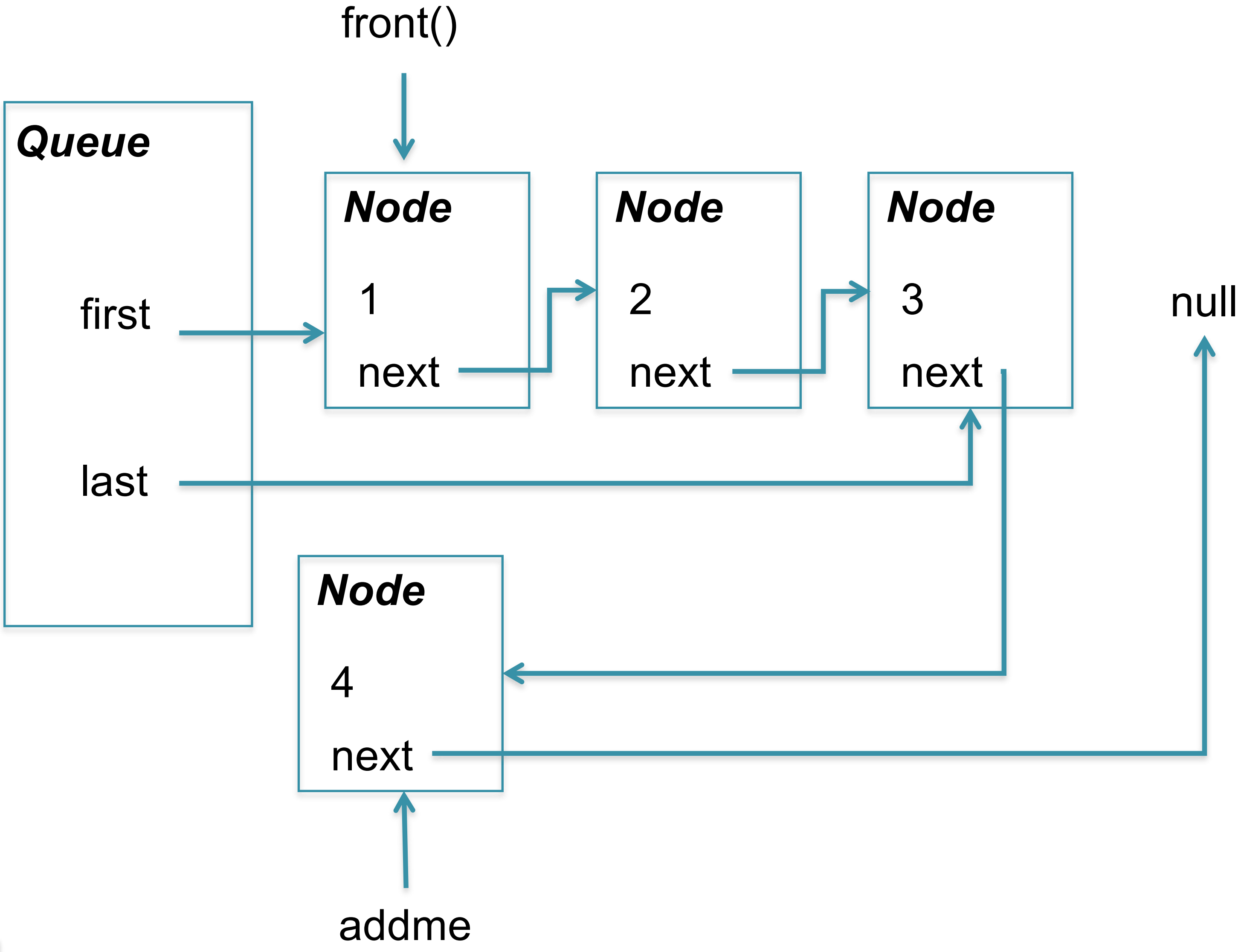
Enqueue last, Dequeue first



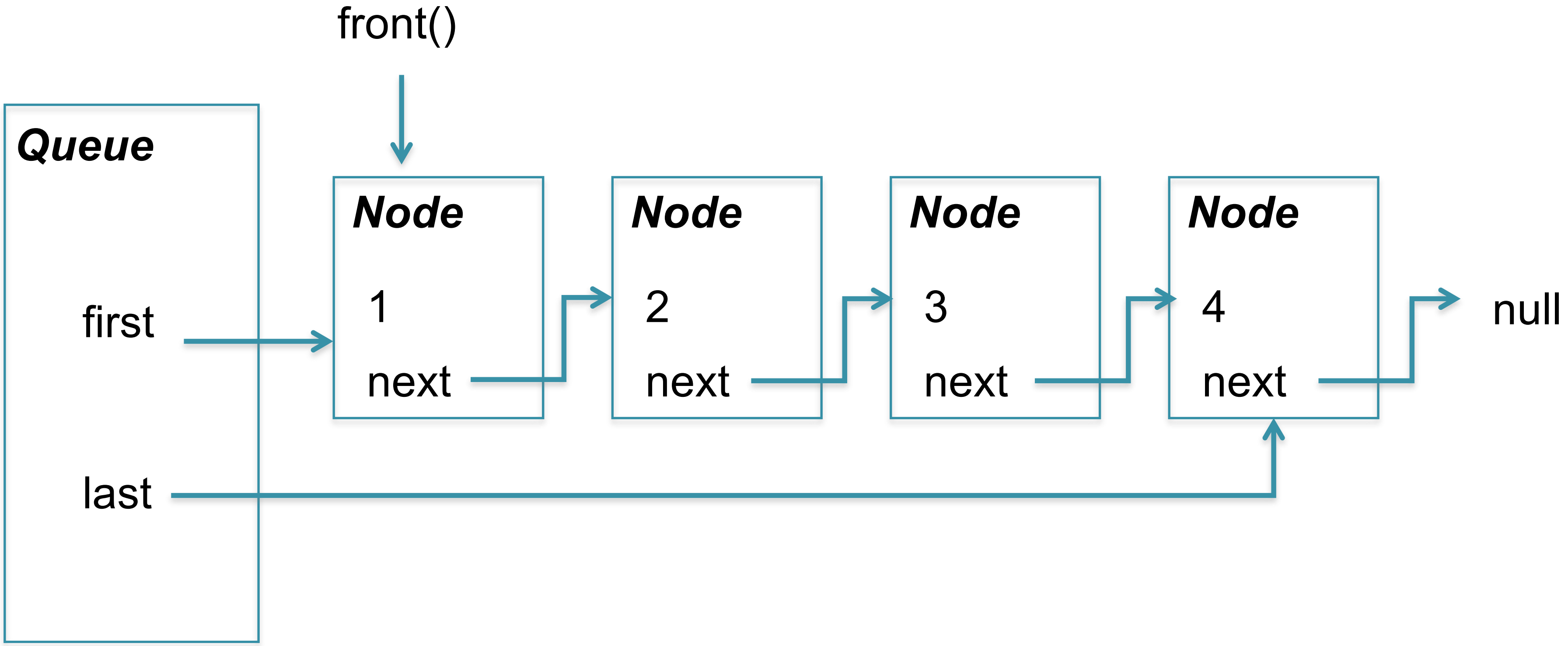
Enqueue last, Dequeue first



Enqueue last, Dequeue first

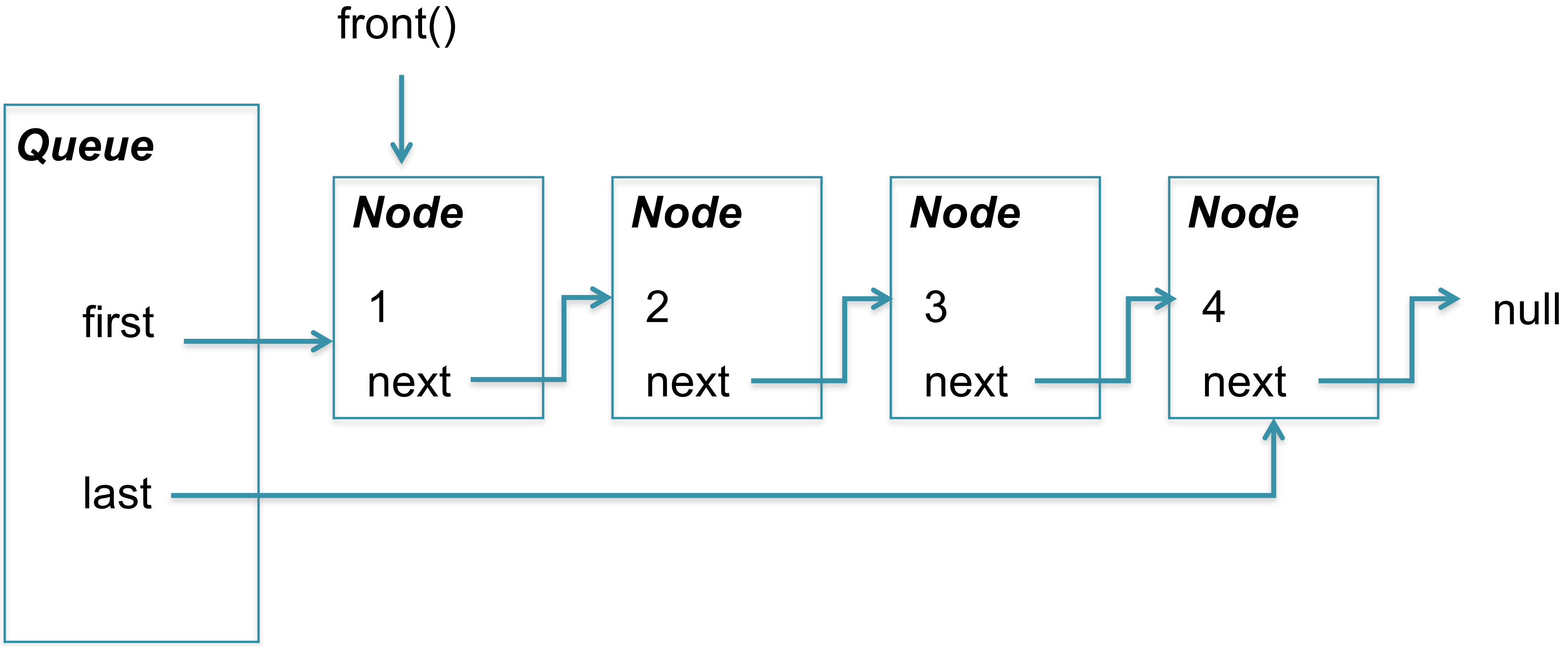


Enqueue last, Dequeue first



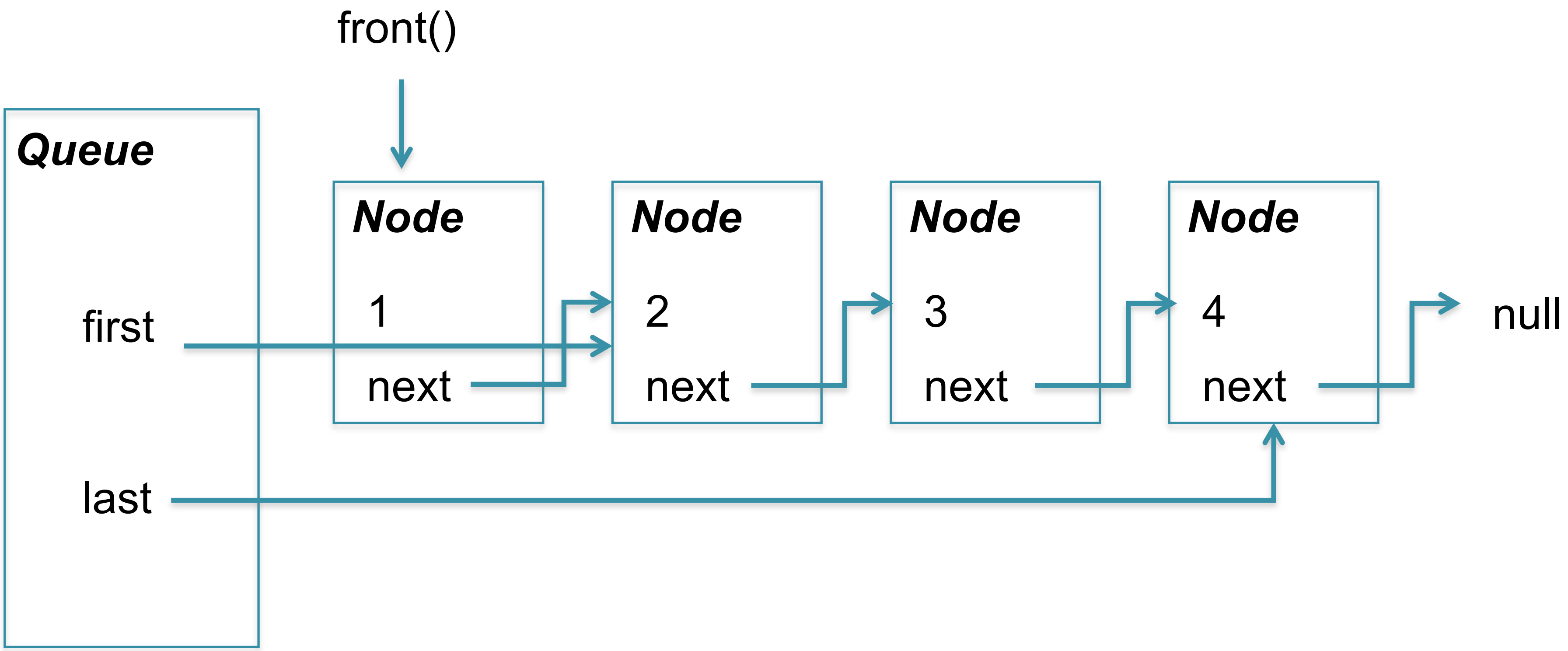
Enqueue is an O(1) operation ☺

Enqueue last, Dequeue first

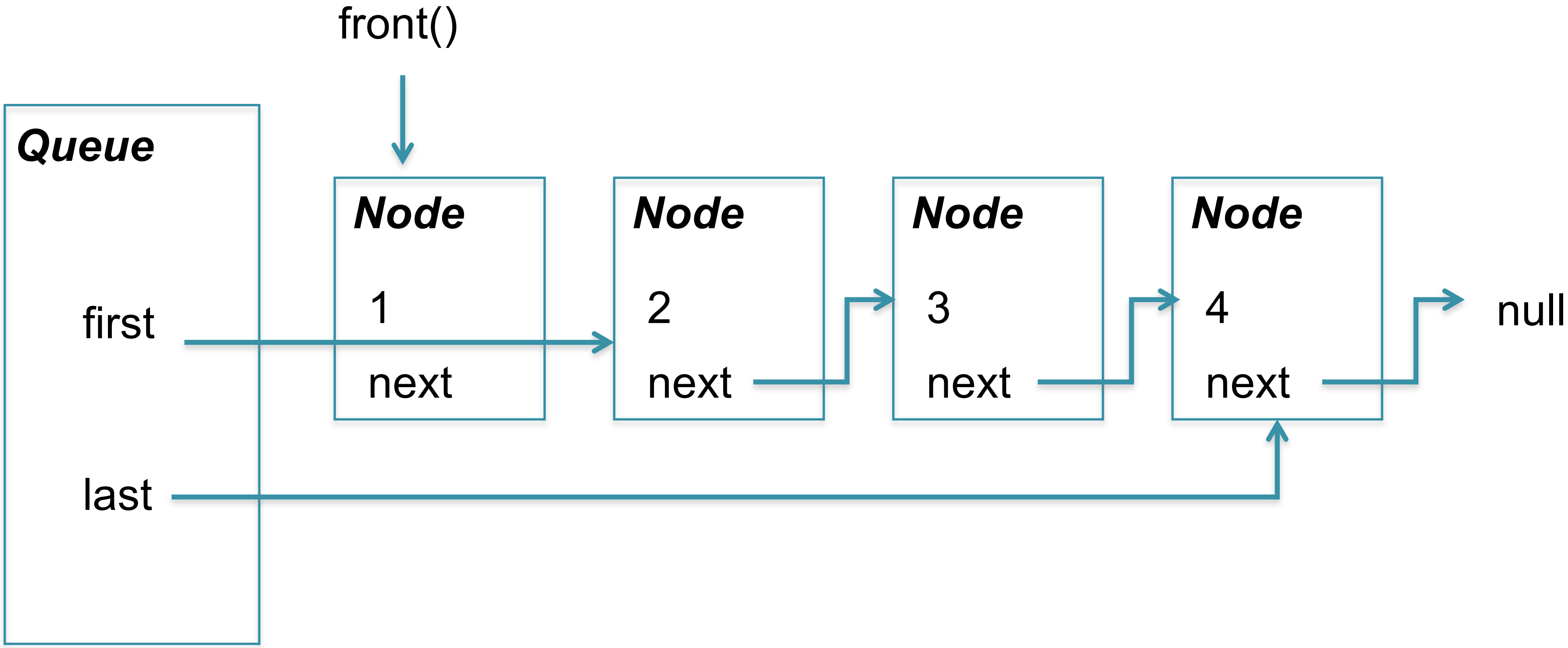


Now try dequeueing at first

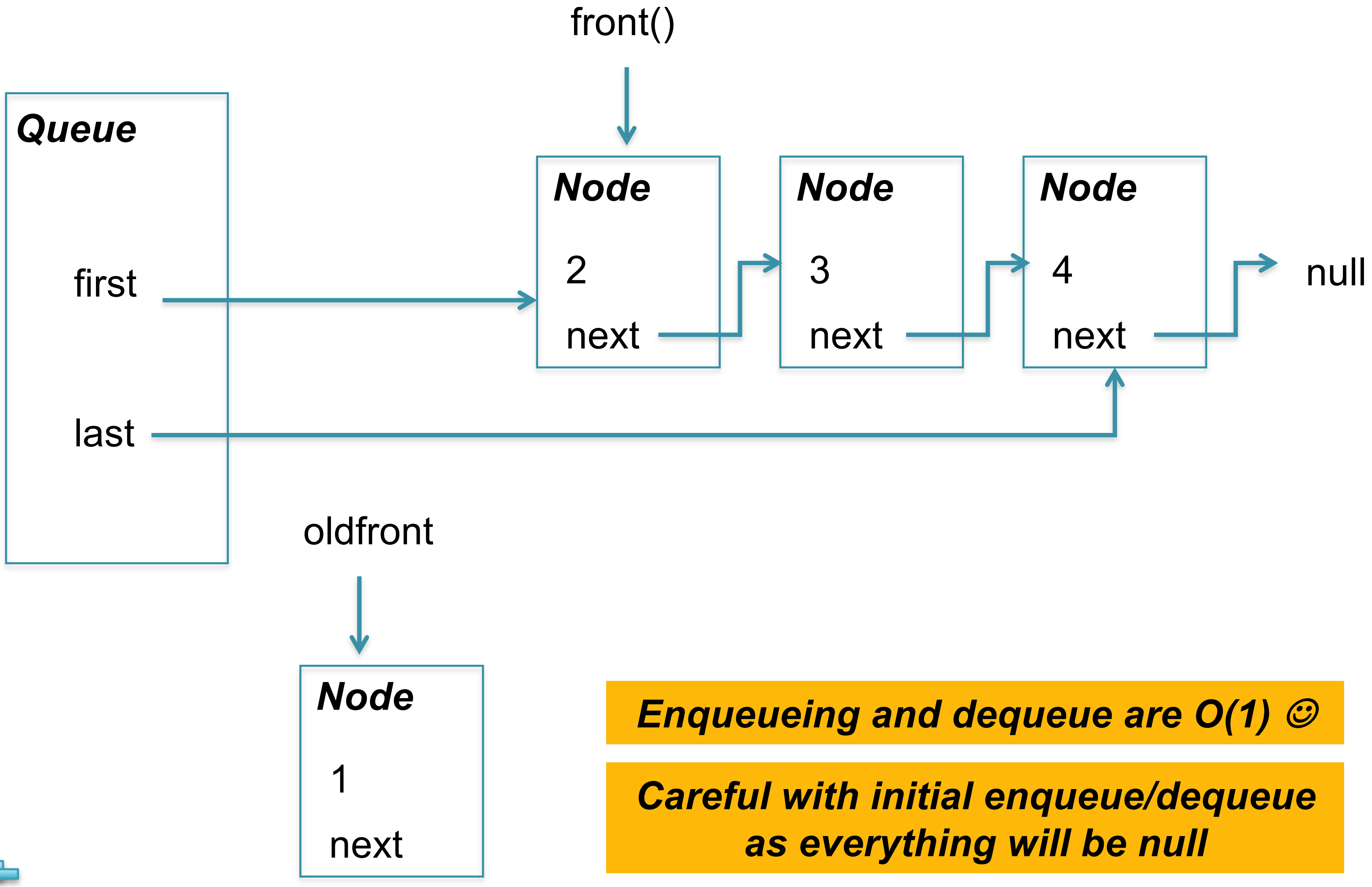
Enqueue last, Dequeue first



Enqueue last, Dequeue first



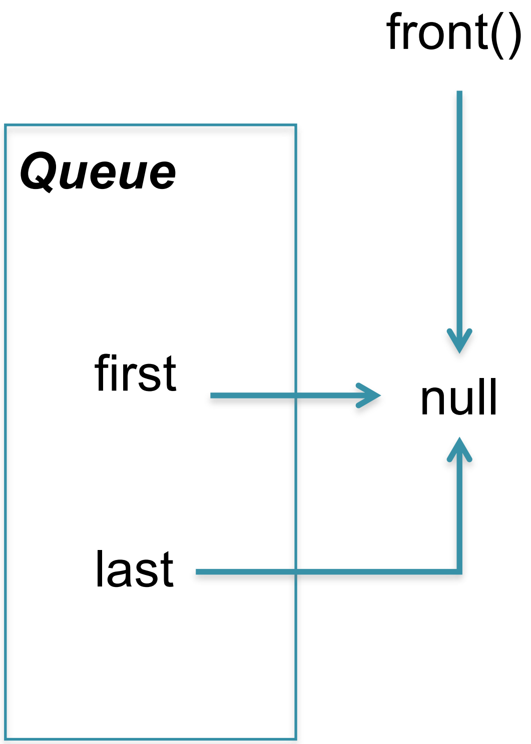
Enqueue last, Dequeue first



Enqueueing and dequeue are O(1) ☺

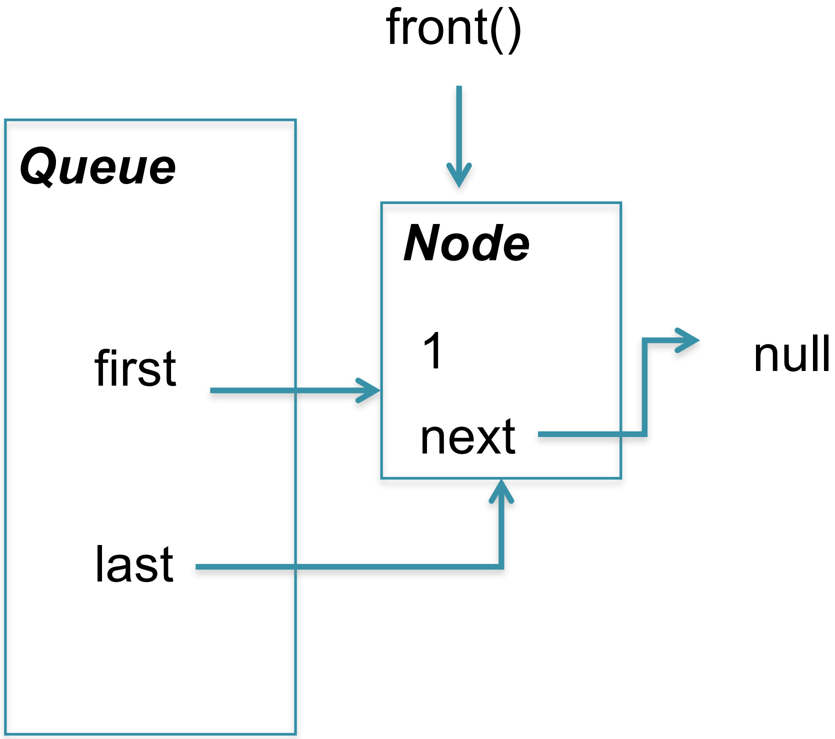
Careful with initial enqueue/dequeue as everything will be null

Enqueue sequence



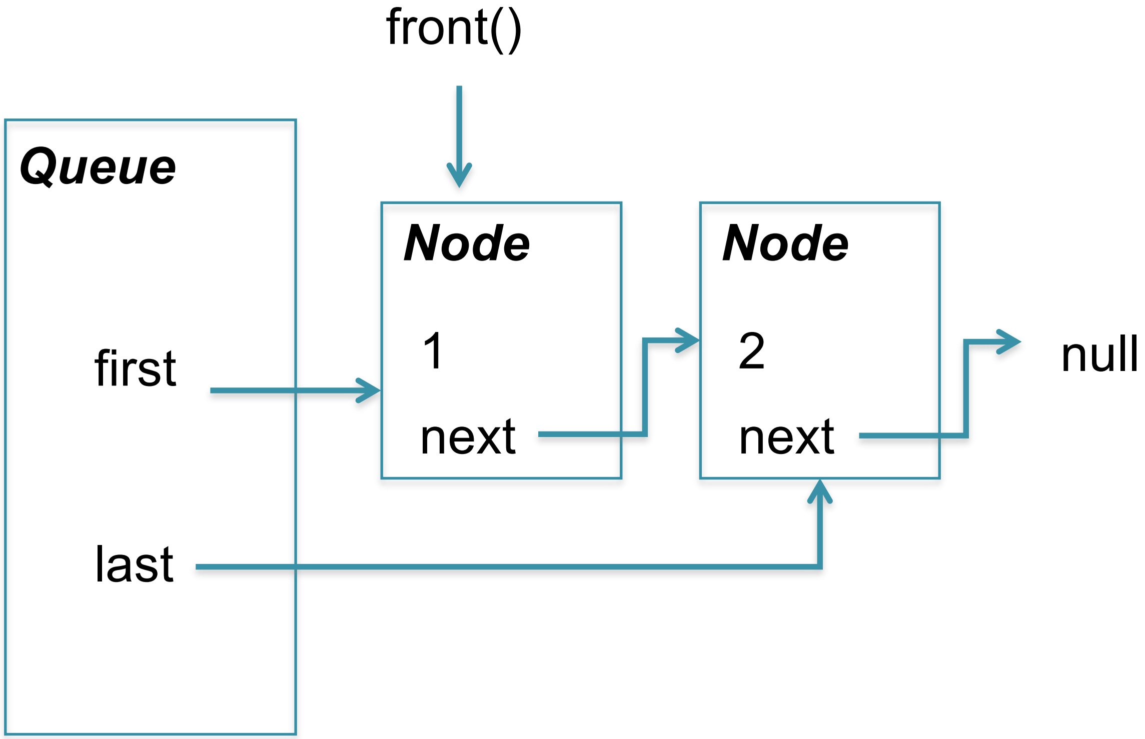
queue.enqueue(1);

Enqueue sequence



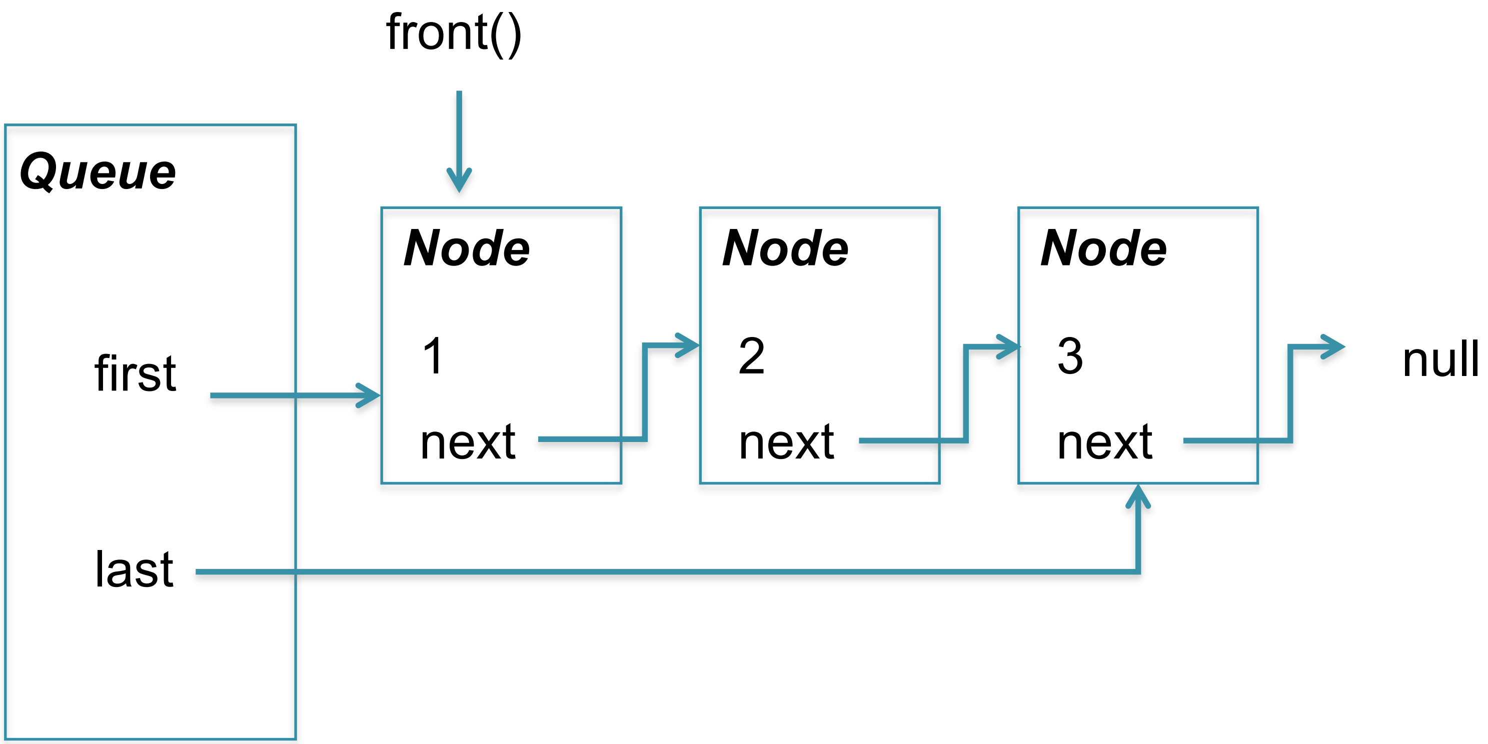
queue.enqueue(1);
queue.enqueue(2);

Enqueue sequence



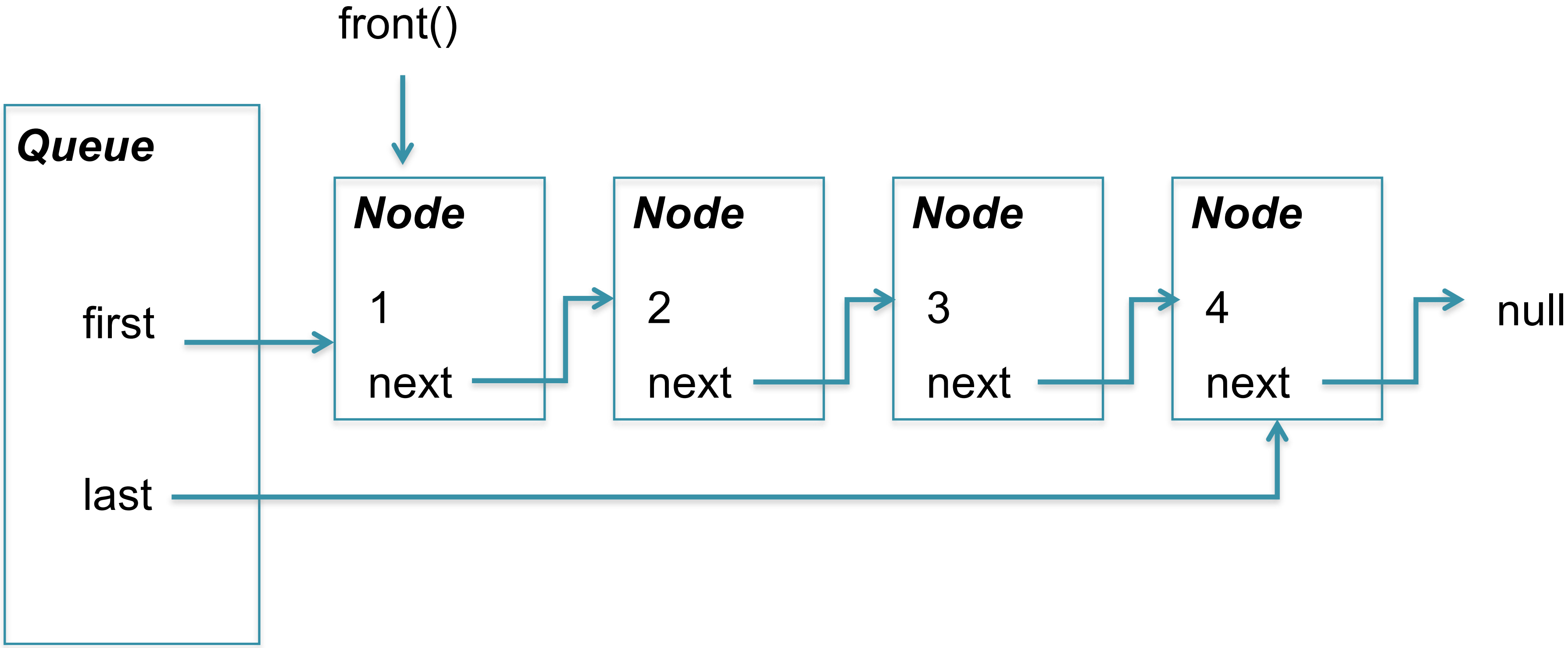
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);

Enqueue sequence



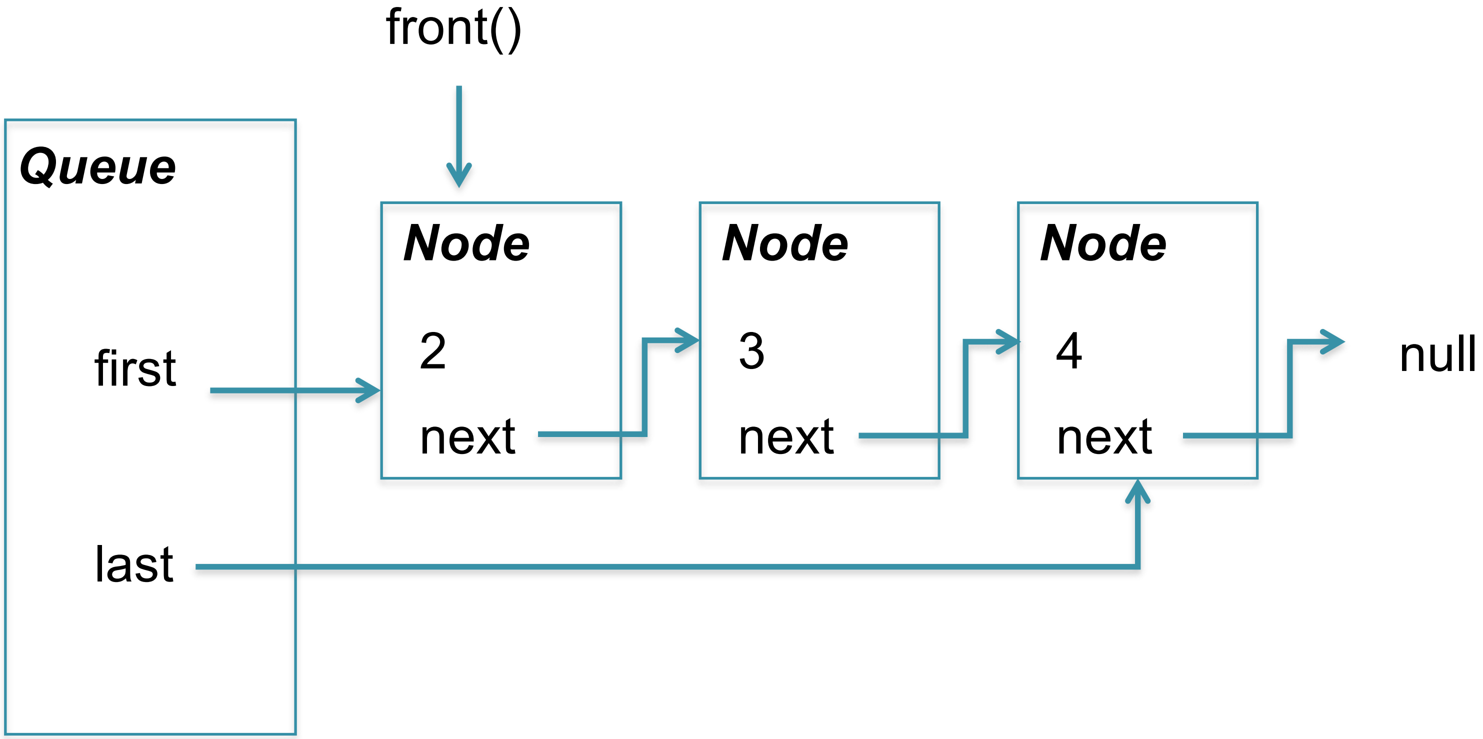
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
queue.enqueue(4);

Enqueue sequence



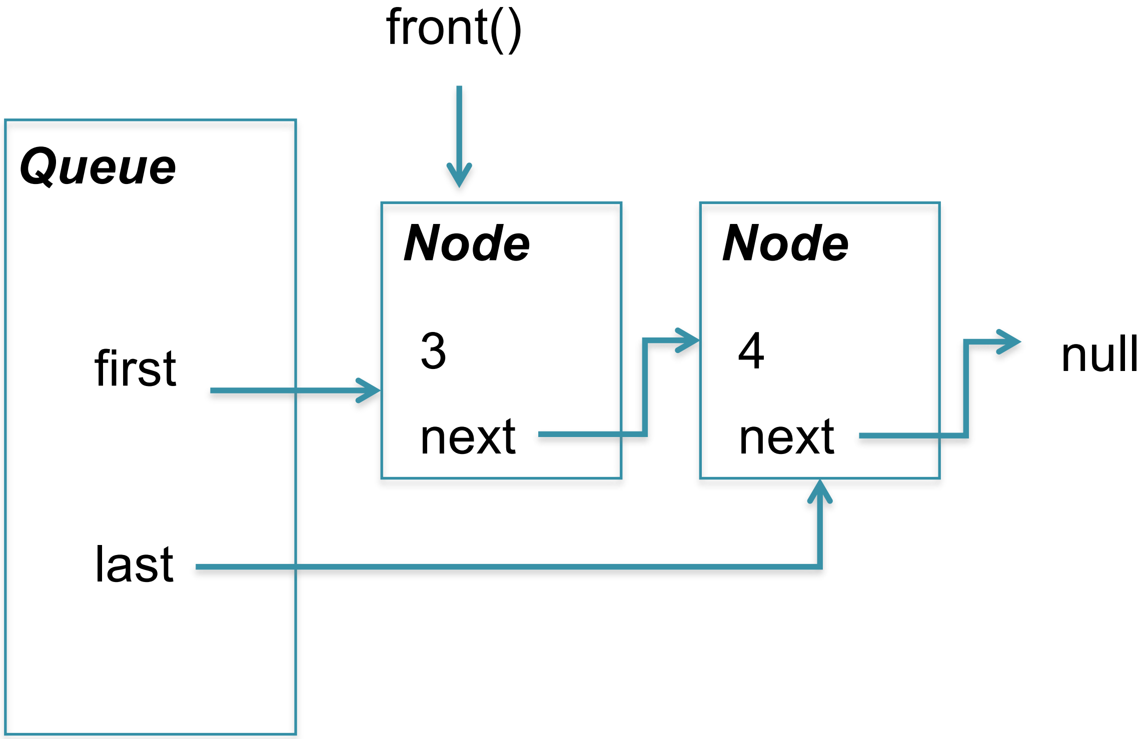
All 4 items queued up and ready to be dequeued starting with 1

Dequeue sequence



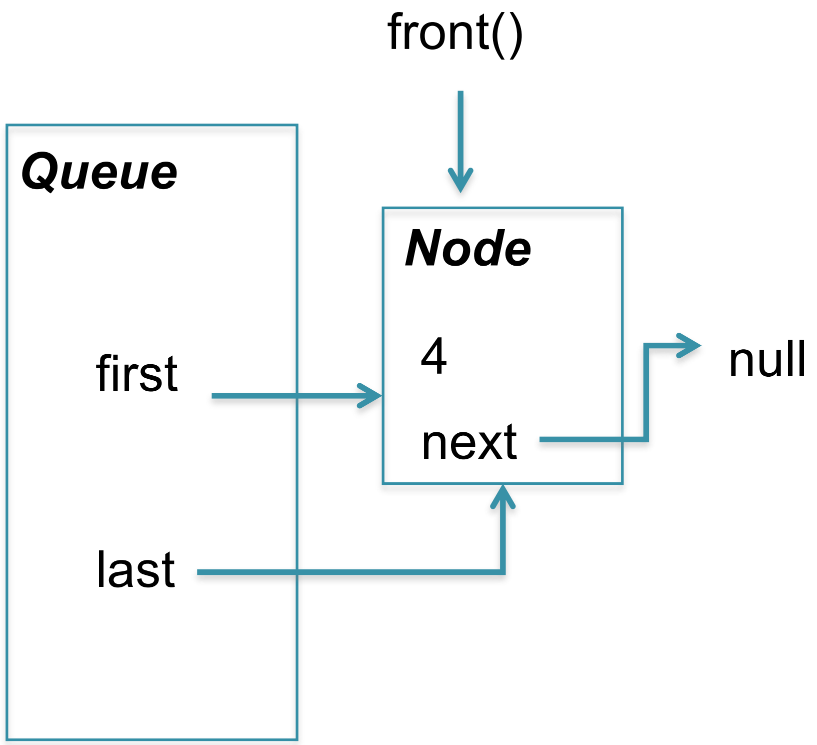
queue.dequeue();
queue.dequeue();

Dequeue sequence



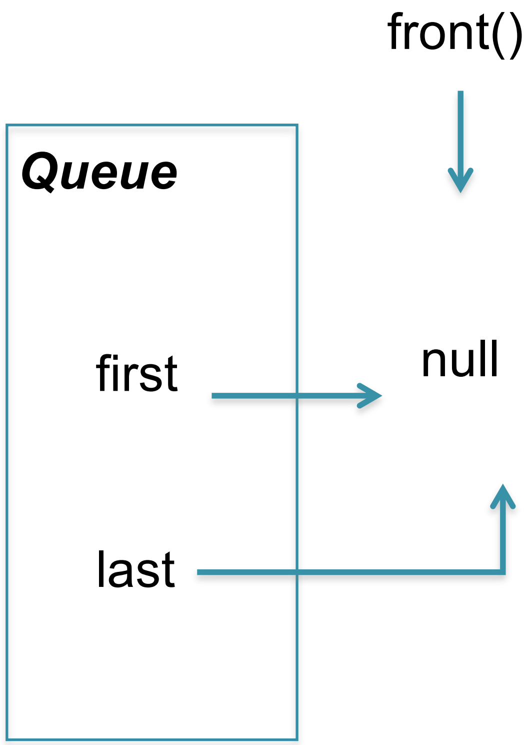
queue.dequeue();
queue.dequeue();
queue.dequeue();

Dequeue sequence



```
queue.dequeue();
queue.dequeue();
queue.dequeue();
queue.dequeue();
```

Dequeue sequence



front()

Queue

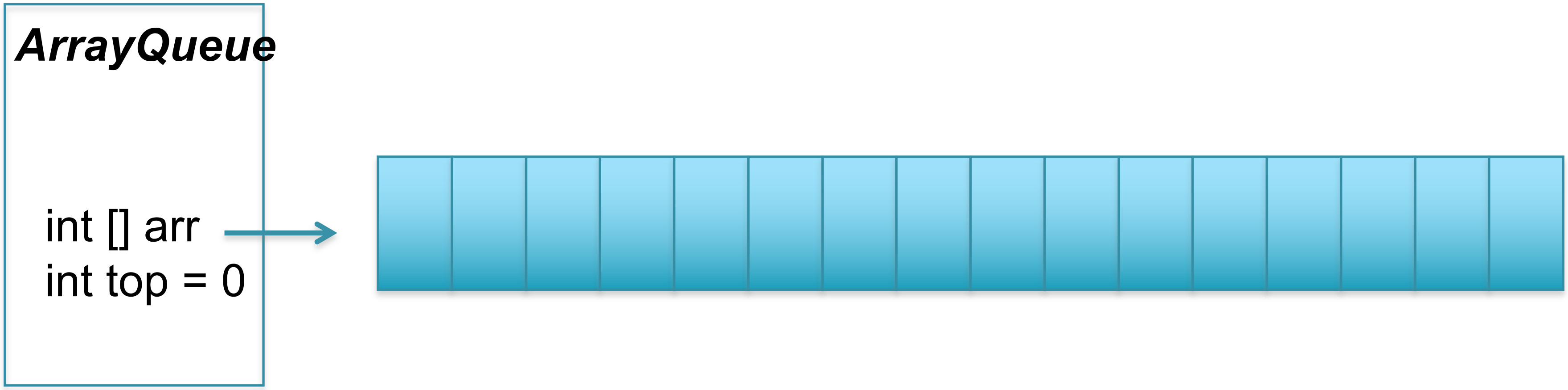
first

null

last

```
queue.dequeue();
queue.dequeue();
queue.dequeue();
queue.dequeue();
```

ArrayQueue



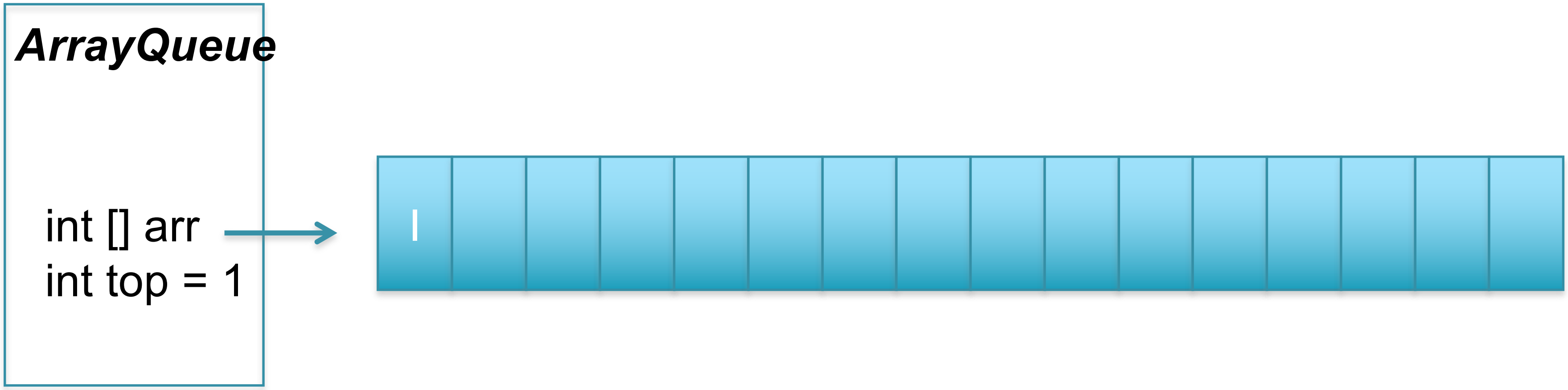
ArrayQueue

int [] arr

int top = 0

Enqueue into the first open slot,
use array doubling to grow array as needed

ArrayQueue



ArrayQueue

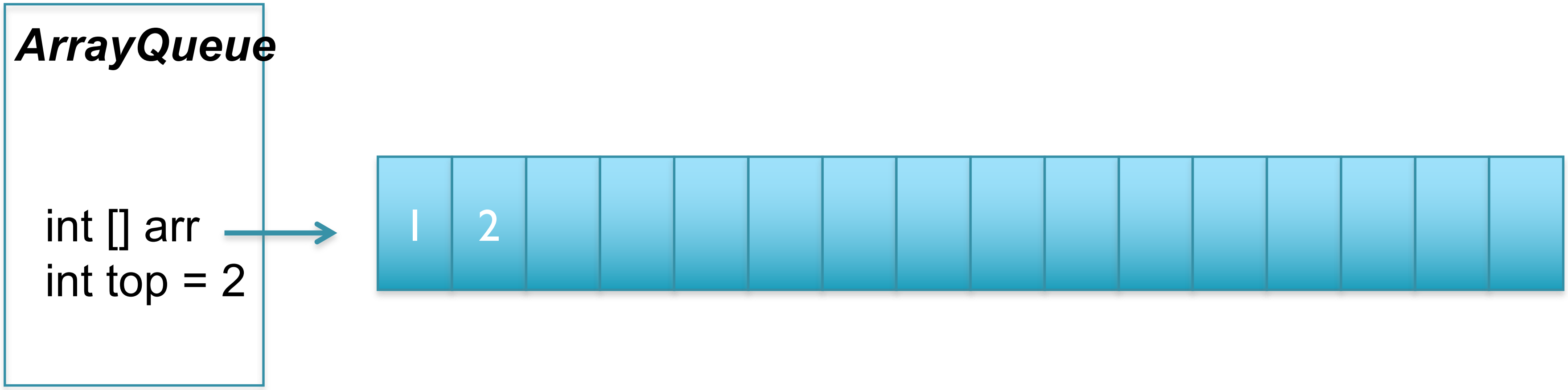
int [] arr

int top = 1

1															

queue.enqueue(1);

ArrayQueue

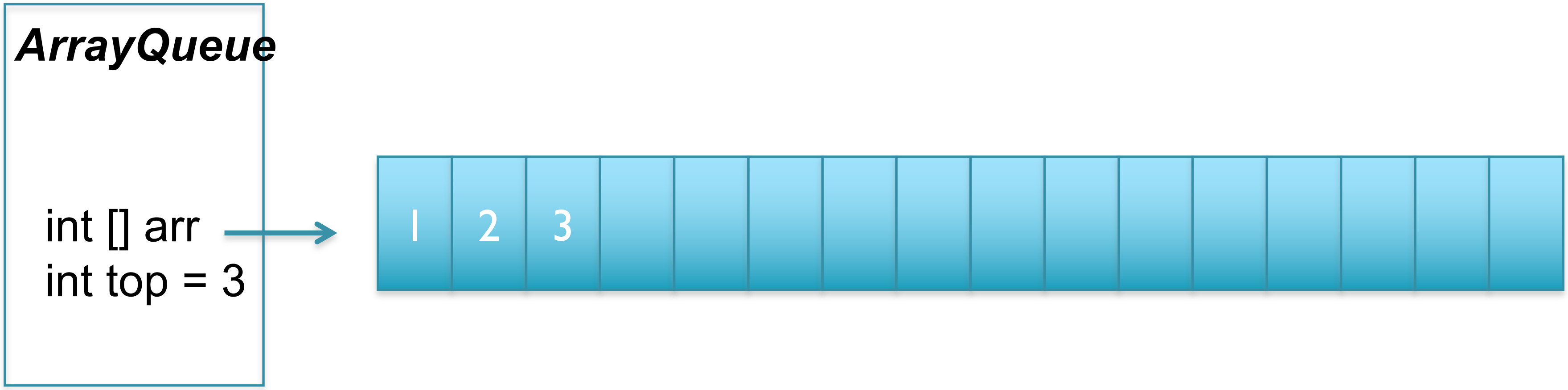


ArrayQueue

int [] arr
int top = 2

```
queue.enqueue(1);
queue.enqueue(2);
```

ArrayQueue



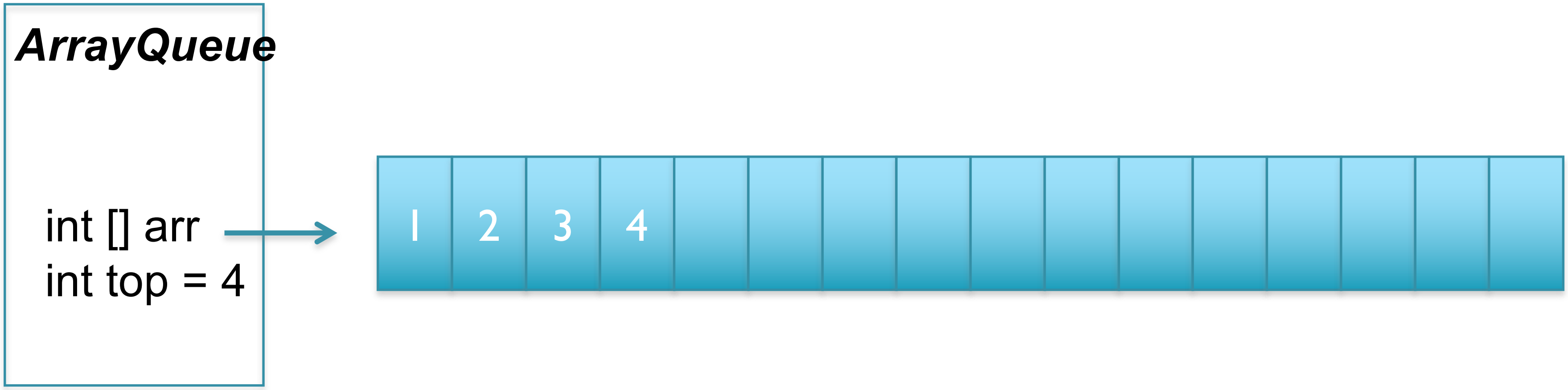
ArrayQueue

int [] arr
int top = 3

1 2 3

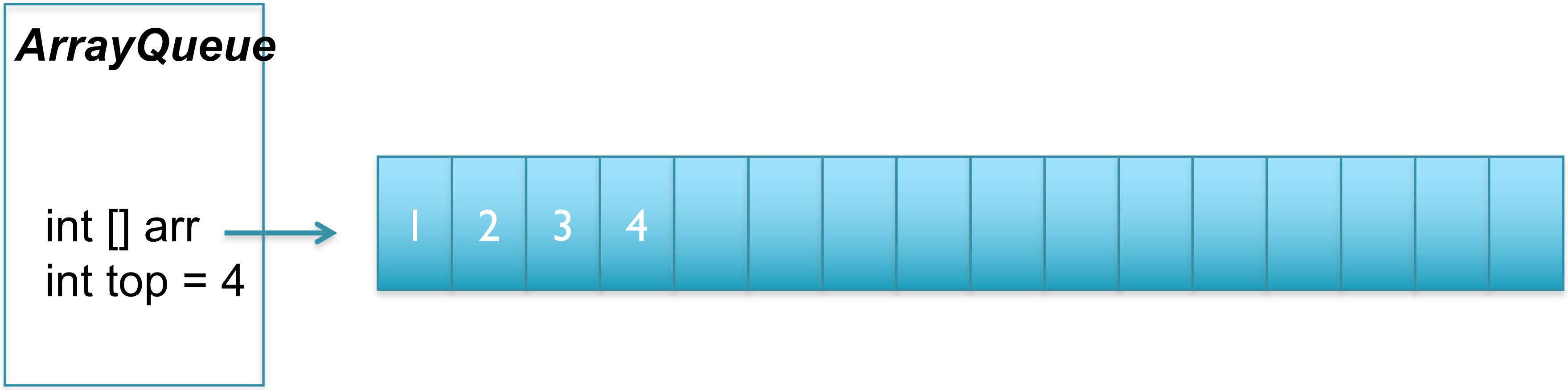
```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
```

ArrayQueue



```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
queue.enqueue(4);
```

ArrayQueue



ArrayQueue

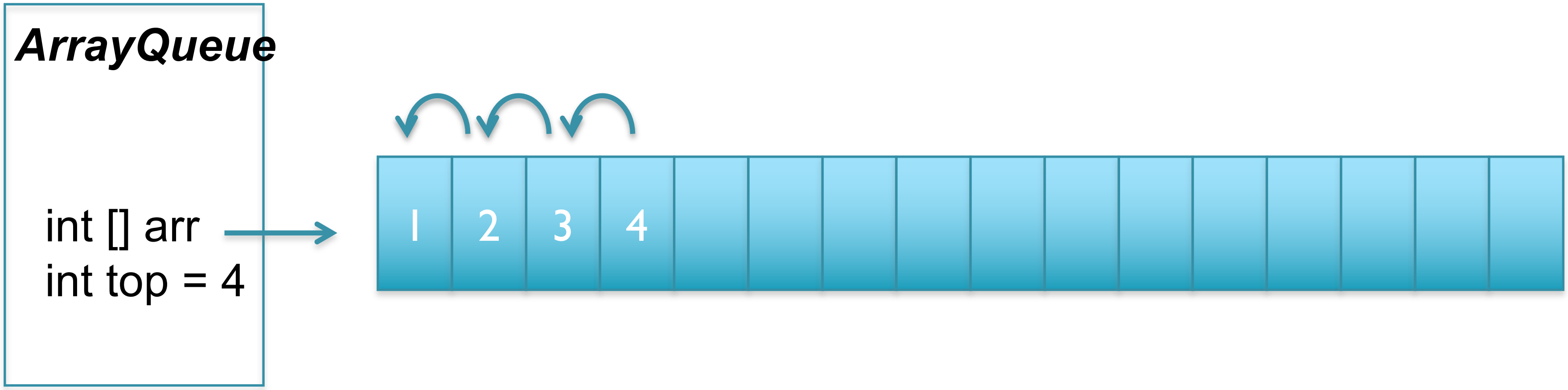
int [] arr

int top = 4

1	2	3	4												

queue.dequeue()

ArrayQueue

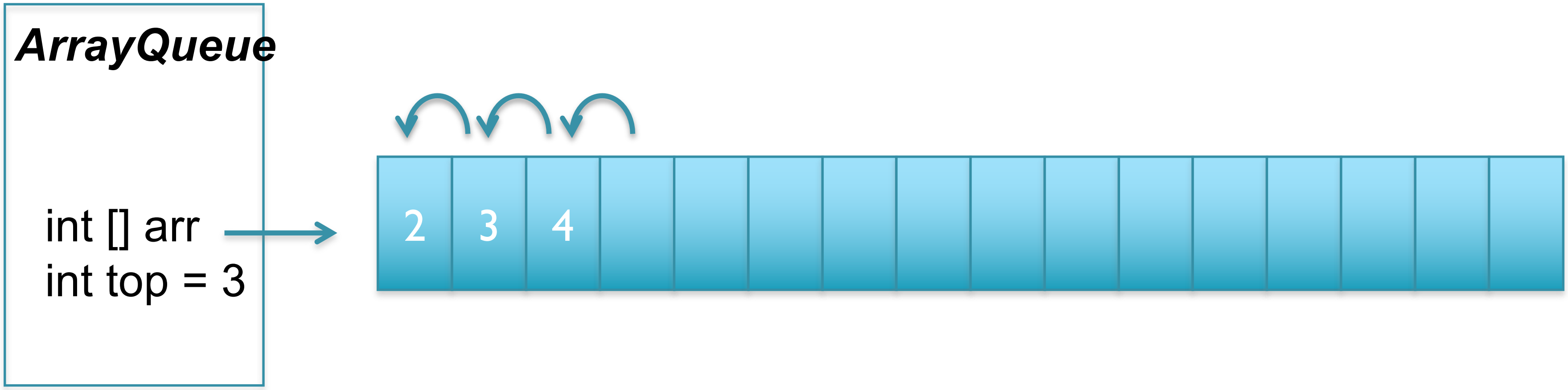


ArrayQueue

int [] arr
int top = 4

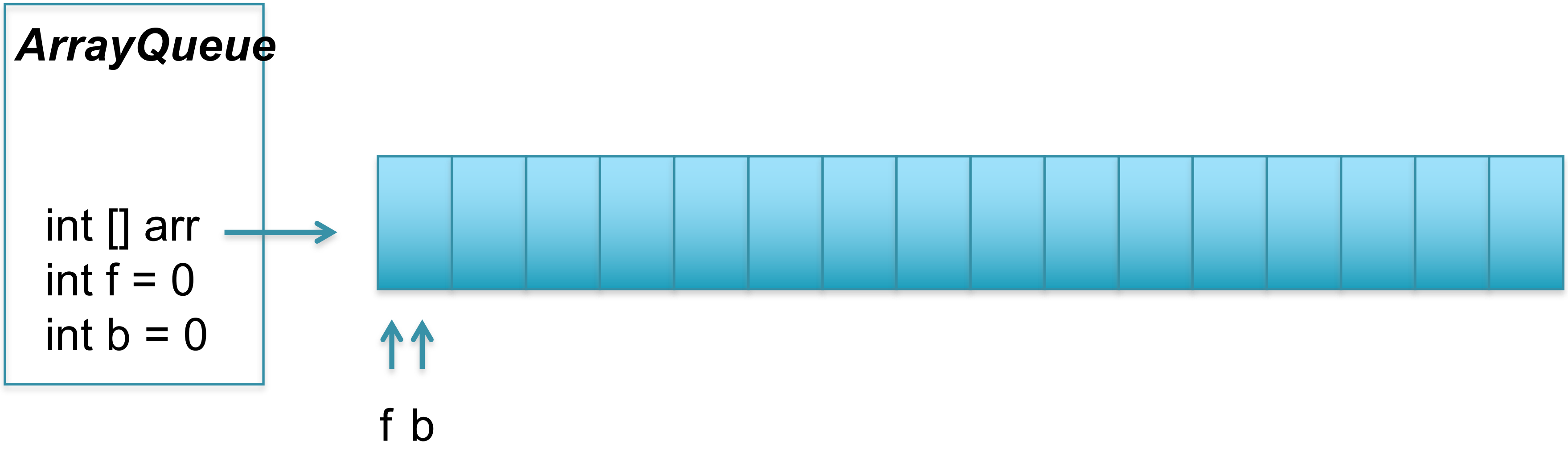
queue.dequeue()

ArrayQueue



Whats wrong with copying?
How could we fix it?

ArrayQueue



ArrayQueue

int [] arr

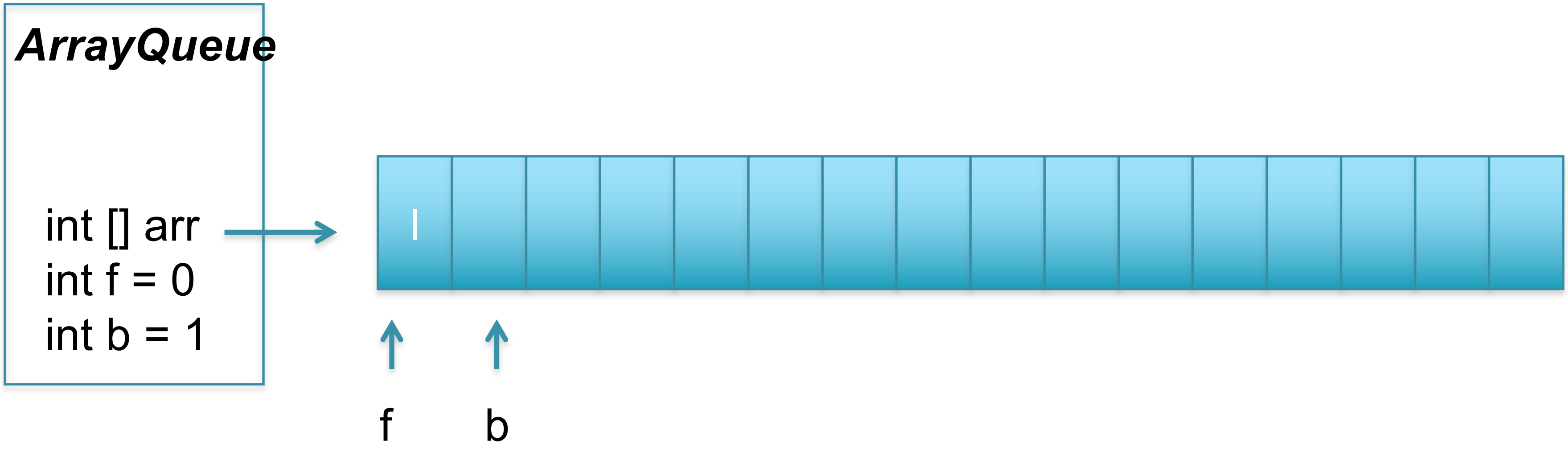
int f = 0

int b = 0

Use a separate index for the front and back of the queue

We know the queue is empty when f == b

ArrayQueue

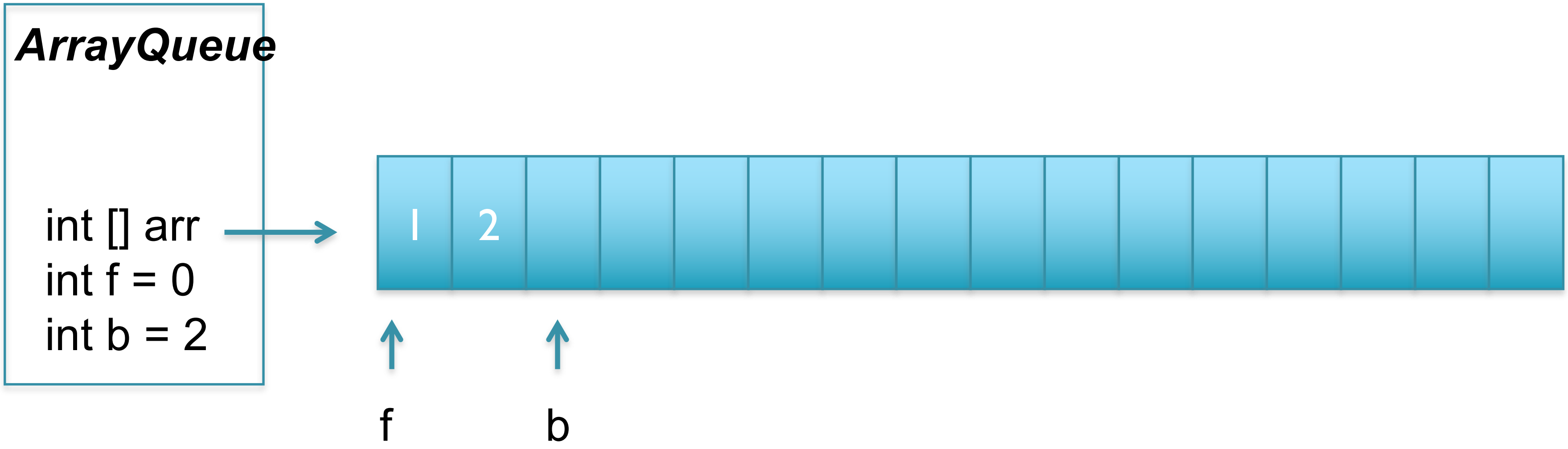


ArrayQueue

int [] arr
int f = 0
int b = 1

```
queue.enqueue(1);
```

ArrayQueue

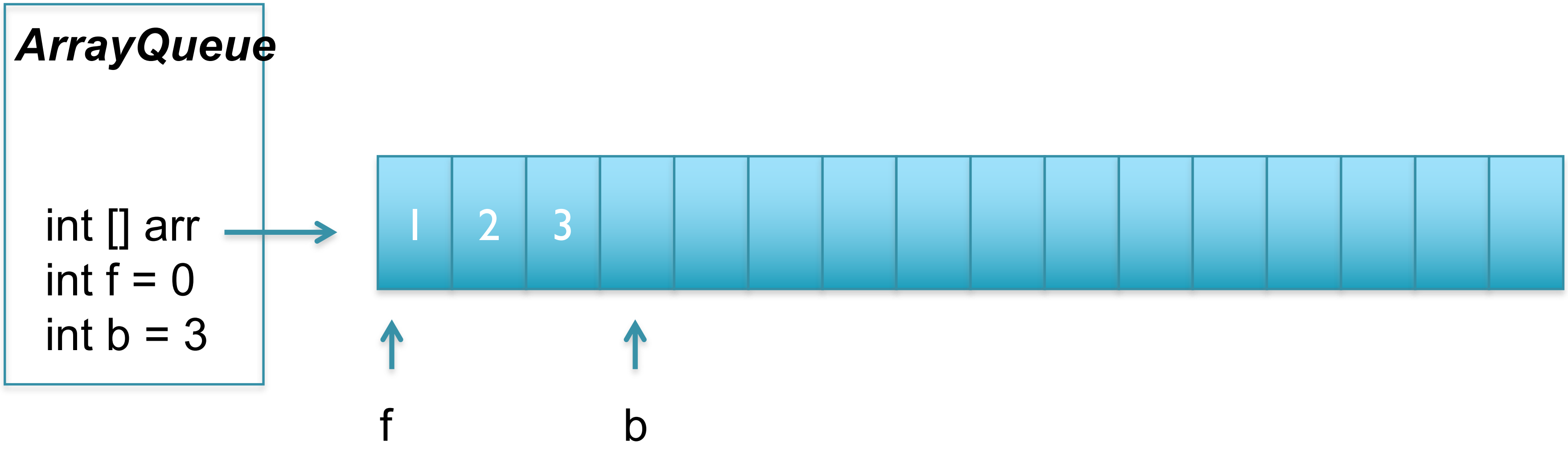


ArrayQueue

int [] arr
int f = 0
int b = 2

```
queue.enqueue(1);
queue.enqueue(2);
```

ArrayQueue

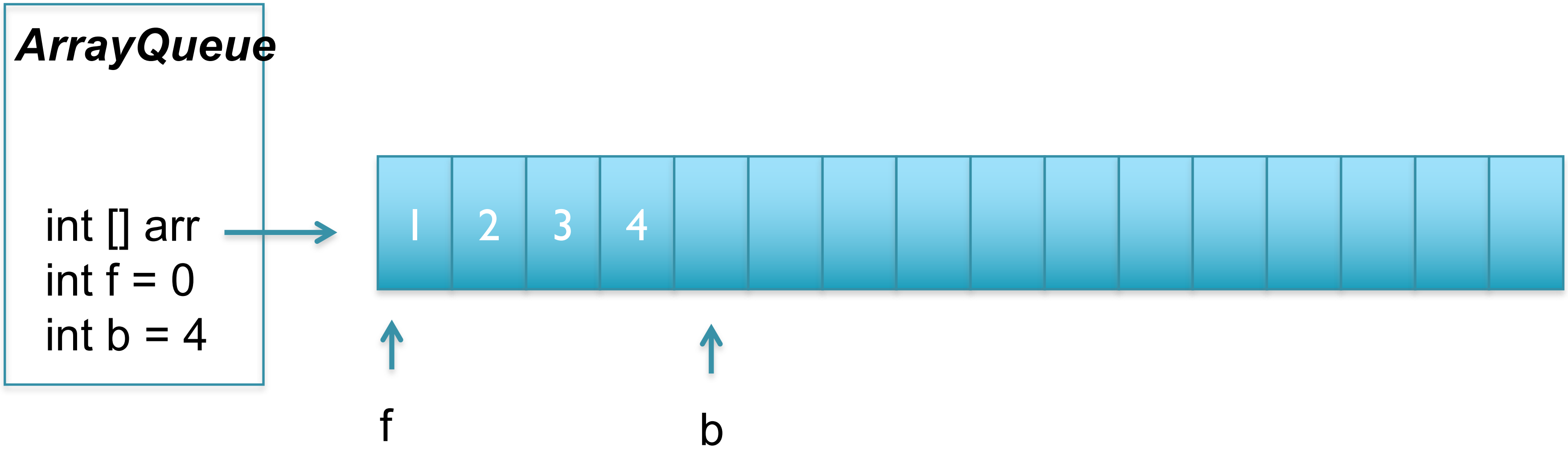


ArrayQueue

```
int [] arr
int f = 0
int b = 3
```

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
```

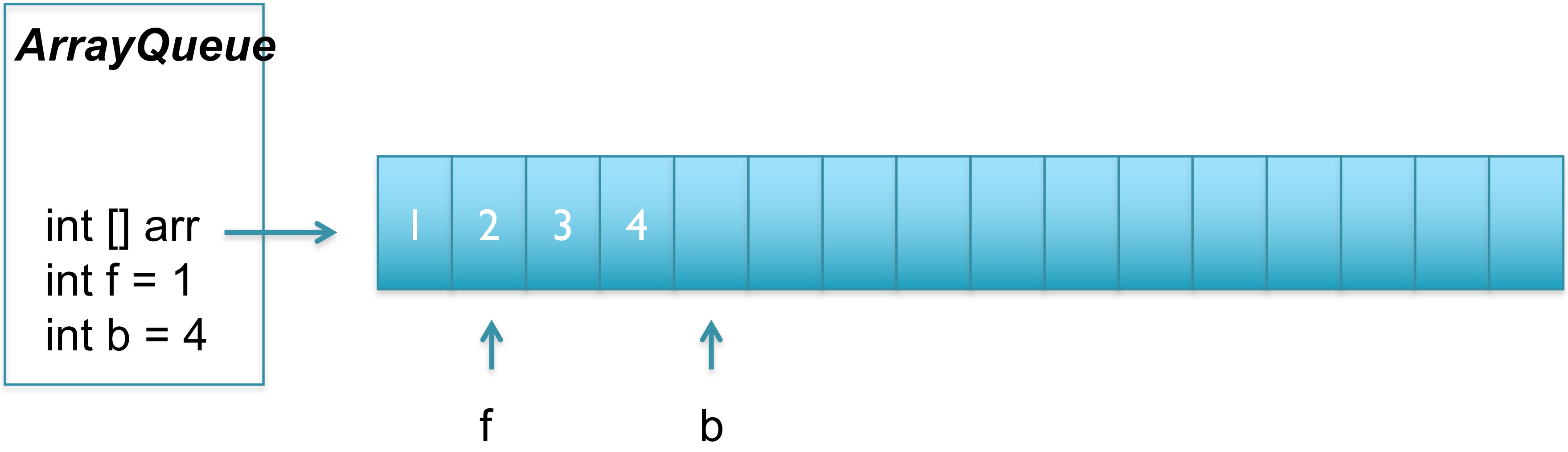
ArrayQueue



```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
queue.enqueue(4);
```

Notice: queuelen = b - f

ArrayQueue



ArrayQueue

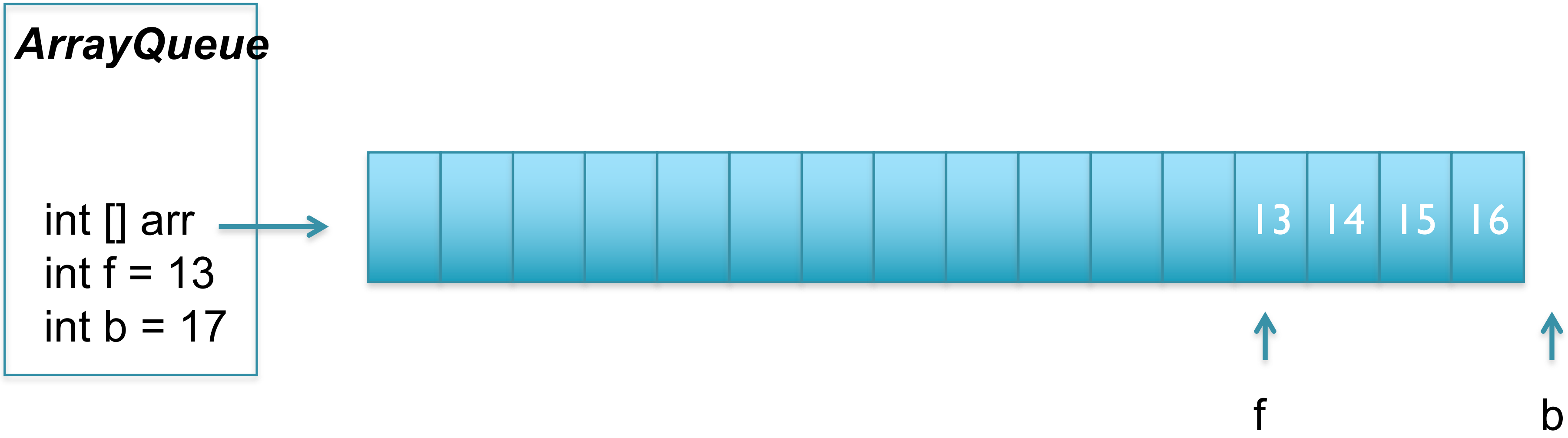
int [] arr
int f = 1
int b = 4

queue.dequeue()

Hooray, enqueue and dequeue are O(1) ☺

We don’t even need to clear out the old front of the list

ArrayQueue

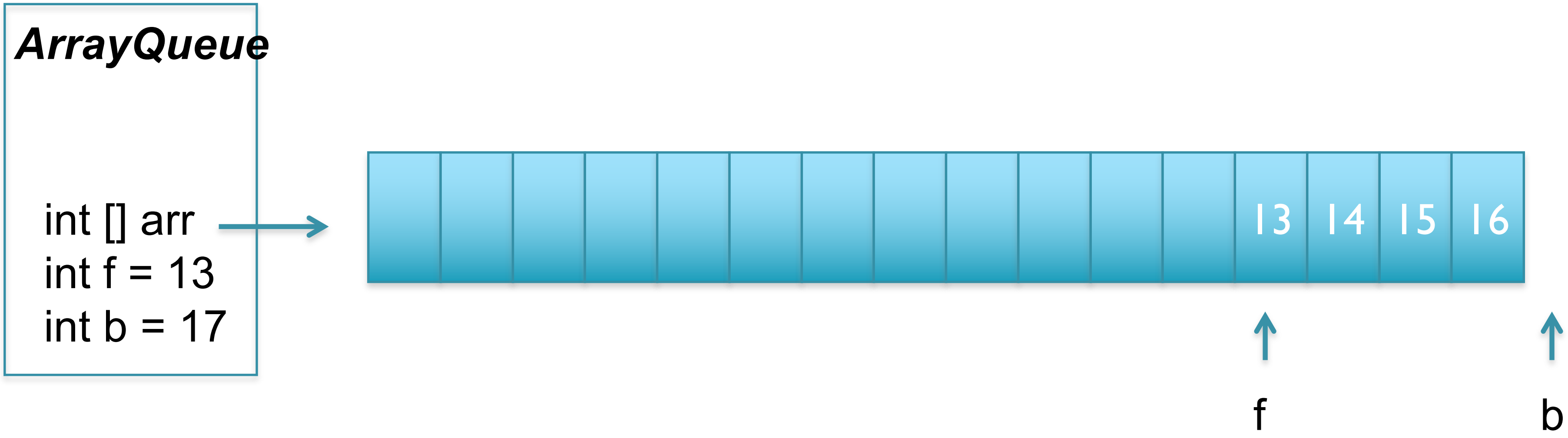


ArrayQueue

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

What happens when we get to the end of the array?
Queuelen = 17-13 = 4 ☺

ArrayQueue



ArrayQueue

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

13 14 15 16

f

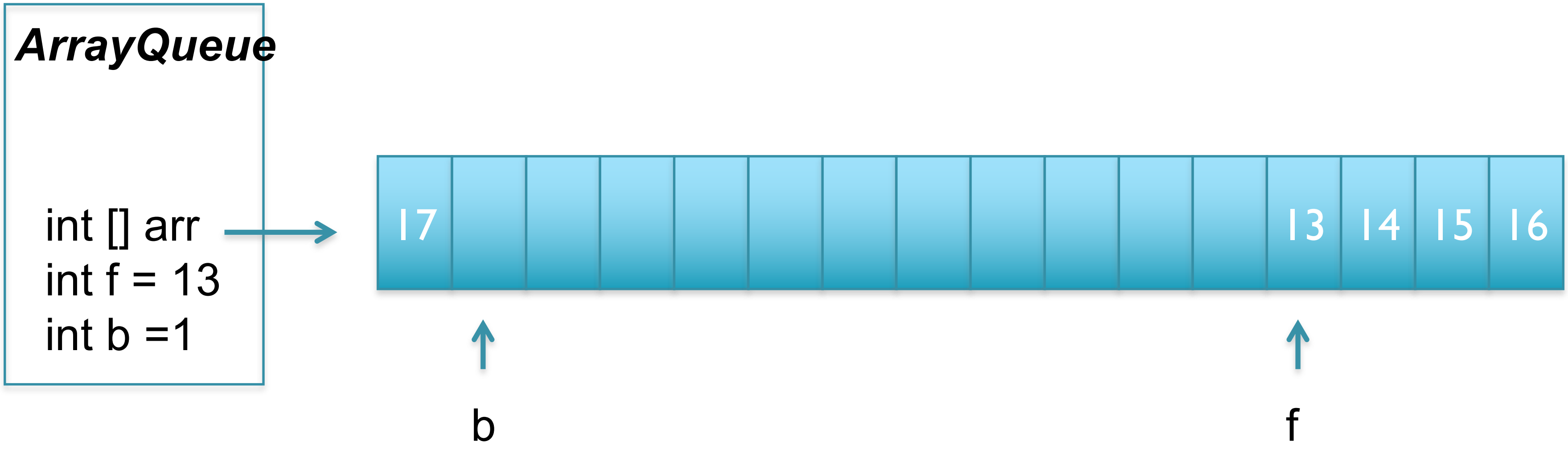
b

queue.enqueue(17);

Should we double the array?

Nah, the array is mostly empty. Lets use it up first!

ArrayQueue



ArrayQueue

int [] arr
int f = 13
int b =1

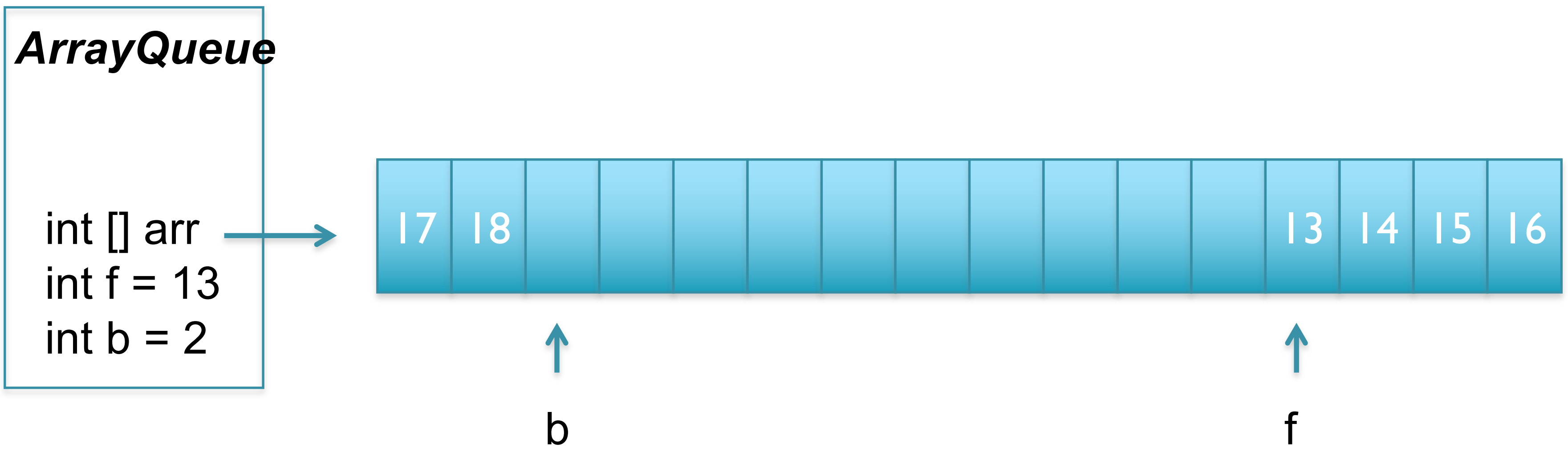
17												13	14	15	16

b

f

queue.enqueue(17);

ArrayQueue



ArrayQueue

int [] arr
int f = 13
int b = 2

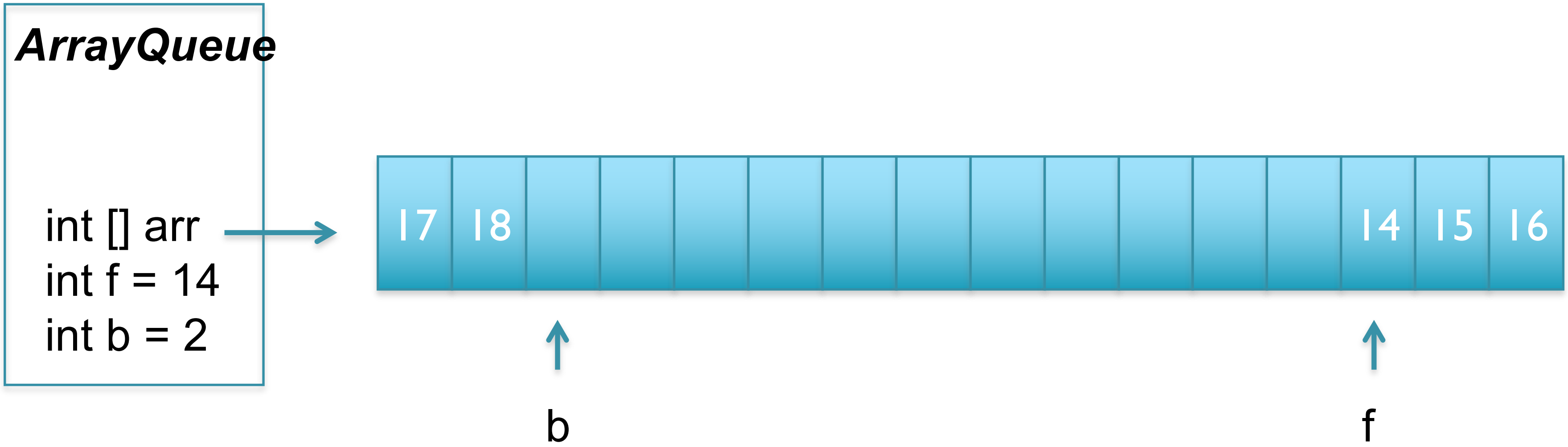
17	18											13	14	15	16

b

f

```
queue.enqueue(17);
queue.enqueue(18);
```

ArrayQueue



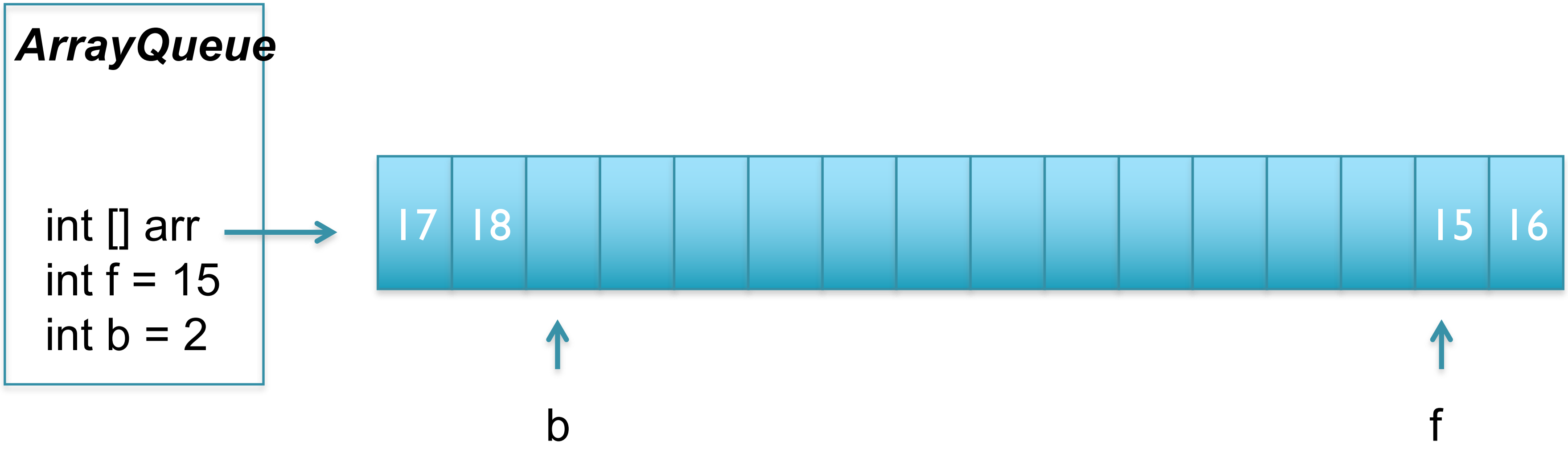
ArrayQueue

int [] arr
int f = 14
int b = 2

17	18												14	15	16

queue.dequeue()

ArrayQueue



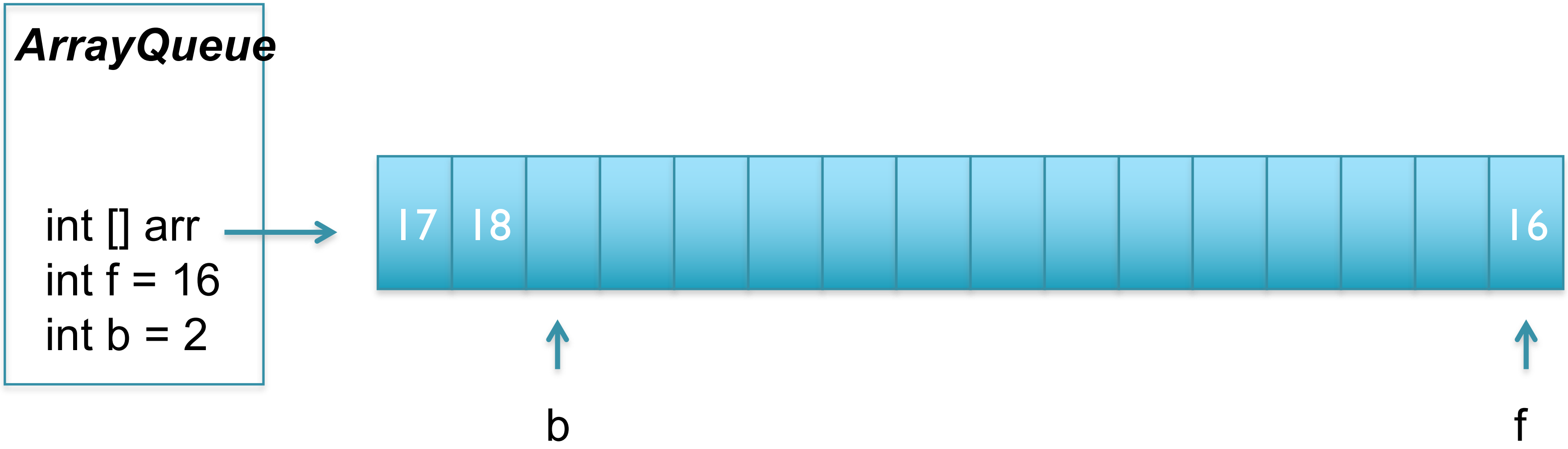
ArrayQueue

int [] arr
int f = 15
int b = 2

17	18													15	16

queue.dequeue()
queue.dequeue()

ArrayQueue



ArrayQueue

int [] arr
int f = 16
int b = 2

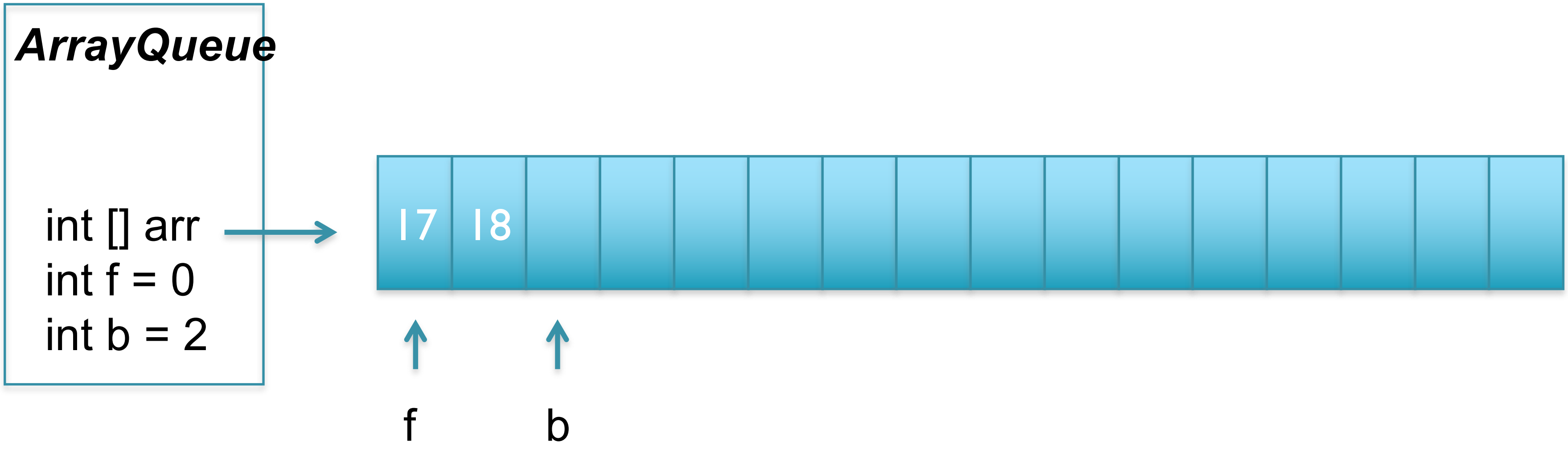
17	18														16

b

f

queue.dequeue()
queue.dequeue()
queue.dequeue()

ArrayQueue



ArrayQueue

int [] arr
int f = 0
int b = 2

How can we implement the wrap around?

queue.dequeue()
queue.dequeue()
queue.dequeue()
queue.dequeue()

Modular Arithmetic

m = a % b means to set m to be the remainder when dividing a by b
m is guaranteed to fall between 0 and b

ShowMod.java

```
public class ShowMod {
  public static void main(String [] args){
    System.out.println("i\ti%2\ti%5\ti%10\ti%16");
    for (int i = 0; i < 20; i++) {
      System.out.println(i + "\t" +
                         i % 2 + "\t" +
                         i % 5 + "\t" +
                         i % 10 + "\t" +
                         i % 16);
    }
  }
}
```

```
$ java Mod
i       i%2     i%5     i%10    i%16
0       0       0       0       0
1       1       1       1       1
2       0       2       2       2
3       1       3       3       3
4       0       4       4       4
5       1       0       5       5
6       0       1       6       6
7       1       2       7       7
8       0       3       8       8
9       1       4       9       9
10      0       0       0       10
11      1       1       1       11
12      0       2       2       12
13      1       3       3       13
14      0       4       4       14
15      1       0       5       15
16      0       1       6       0
17      1       2       7       1
18      0       3       8       2
19      1       4       9       3
```

back = (back + 1) % arr.length

How do we compute length or know when it is full?

Use a separate counter. When array is totally full, double the size and copy into new array starting at 0

Stacks versus Queues

LIFO: Last-In-First-Out

Add to top +
Remove from top

FIFO: First-In-First-Out

Add to back +
Remove from front

Stacks versus Queues

LIFO: Last-In-First-Out

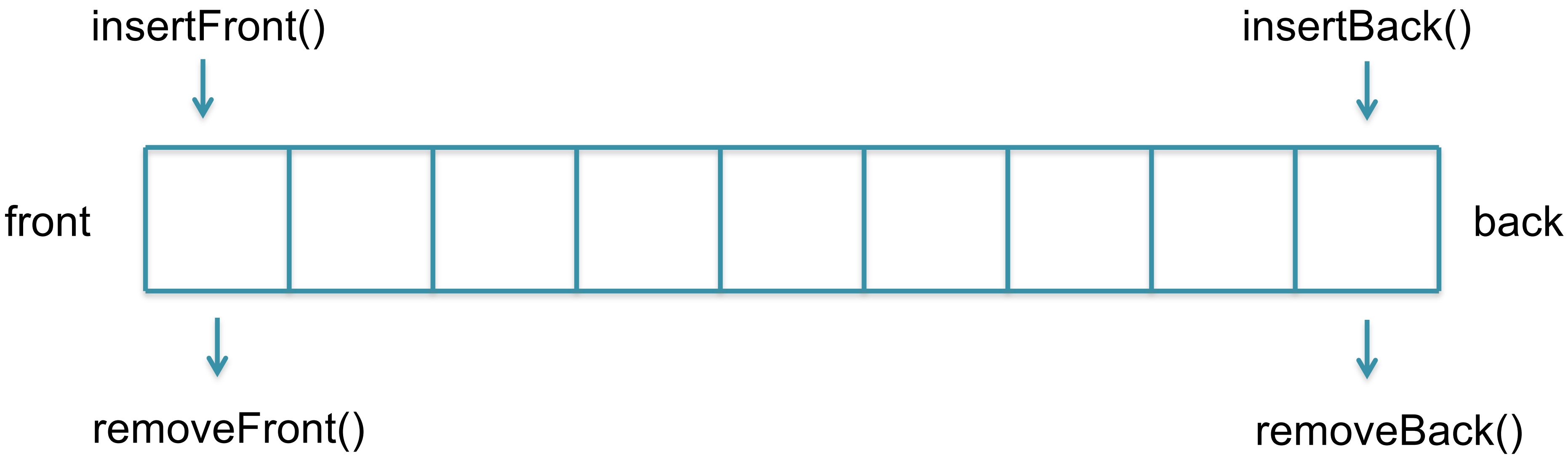
Add to top +
Remove from top

FIFO: First-In-First-Out

Add to back +
Remove from front

Dequeues
aka Doubled-Ended Queue
aka Deques
aka “Decks”

Dequeues



Dynamic Data Structure used for storing sequences of data

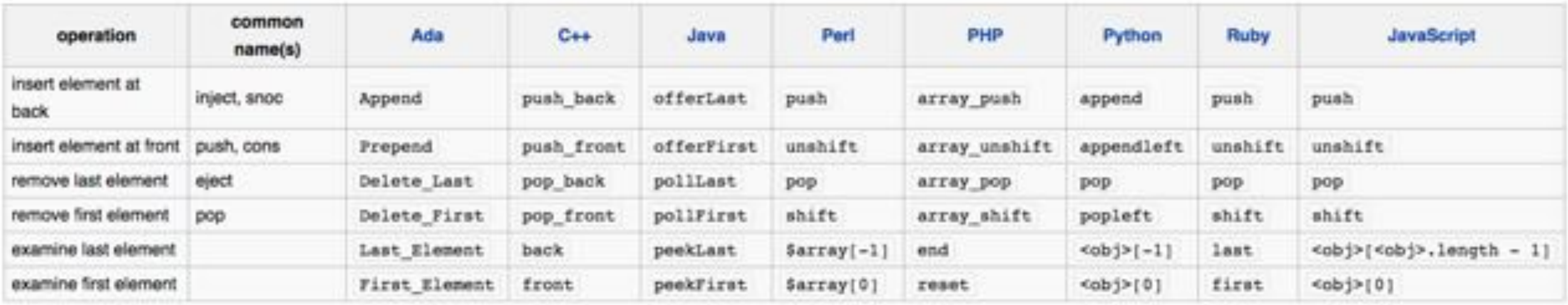
- Insert/Remove at either end in O(1)

- If you exclusively add/remove at one end, then ***it becomes a stack***

- If you exclusive add to one end and remove from other, then ***it becomes a queue***

- Many other applications:
  - browser history: deque of last 100 webpages visited

Dequeue Support



operation	common name(s)	Ada	C++	Java	Perl	PHP	Python	Ruby	JavaScript
insert element at back	inject, snoc	`Append`	`push_back`	`offerLast`	`push`	`array_push`	`append`	`push`	`push`
insert element at front	push, cons	`Prepend`	`push_front`	`offerFirst`	`unshift`	`array_unshift`	`appendleft`	`unshift`	`unshift`
remove last element	eject	`Delete_Last`	`pop_back`	`pollLast`	`pop`	`array_pop`	`pop`	`pop`	`pop`
remove first element	pop	`Delete_First`	`pop_front`	`pollFirst`	`shift`	`array_shift`	`popleft`	`shift`	`shift`
examine last element		`Last_Element`	`back`	`peekLast`	`$array[-1]`	`end`	`<obj>[-1]`	`last`	`<obj>[<obj>.length - 1]`
examine first element		`First_Element`	`front`	`peekFirst`	`$array[0]`	`reset`	`<obj>[0]`	`first`	`<obj>[0]`

Many common programming languages have builtin support

- Offers most flexibility on how users may choose to use data structure
  - Stack or queue from one data structure

- This is what you should use for “production” code …
  - … but still useful to implement your own so you fully understand the limitations ☺

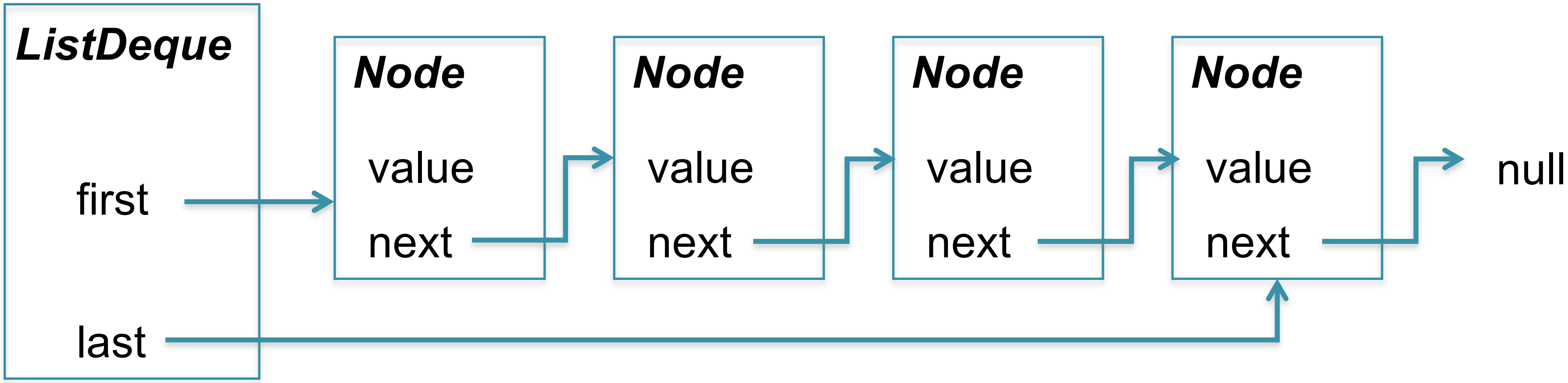
Dequeue Interface

```
public interface Dequeue<T> {
    boolean empty();
    int length();

    T front() throws EmptyException;
    T back() throws EmptyException;

    void insertFront(T t);
    void insertBack(T t);

    void removeFront() throws EmptyException;
    void removeBack() throws EmptyException;
}
```

How would you implement the underlying storage?

Why?

ArrayDeque



ListDeque

first

last

Node

value

next

Node

value

next

Node

value

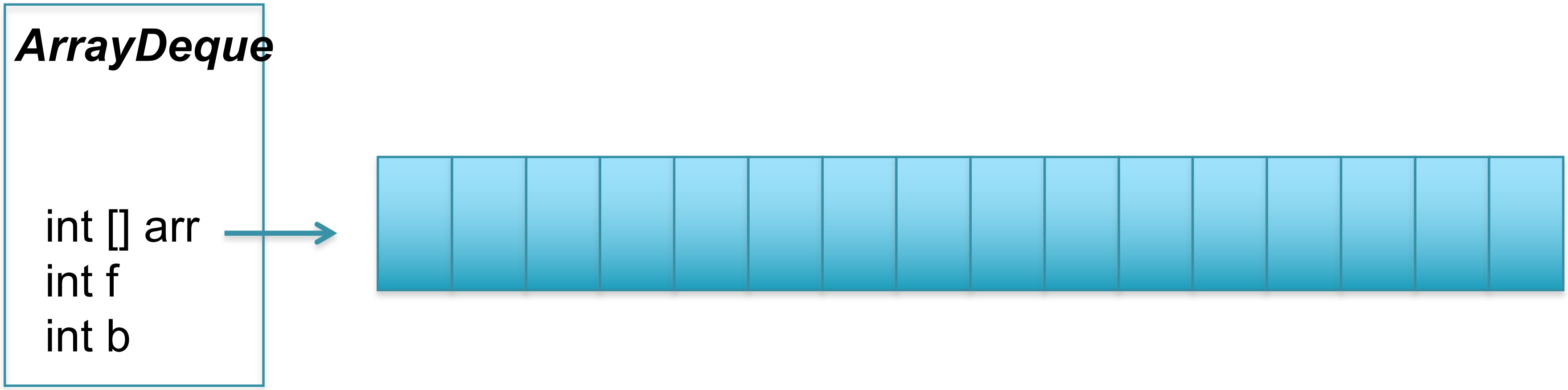
next

Node

value

next

null



ArrayDeque

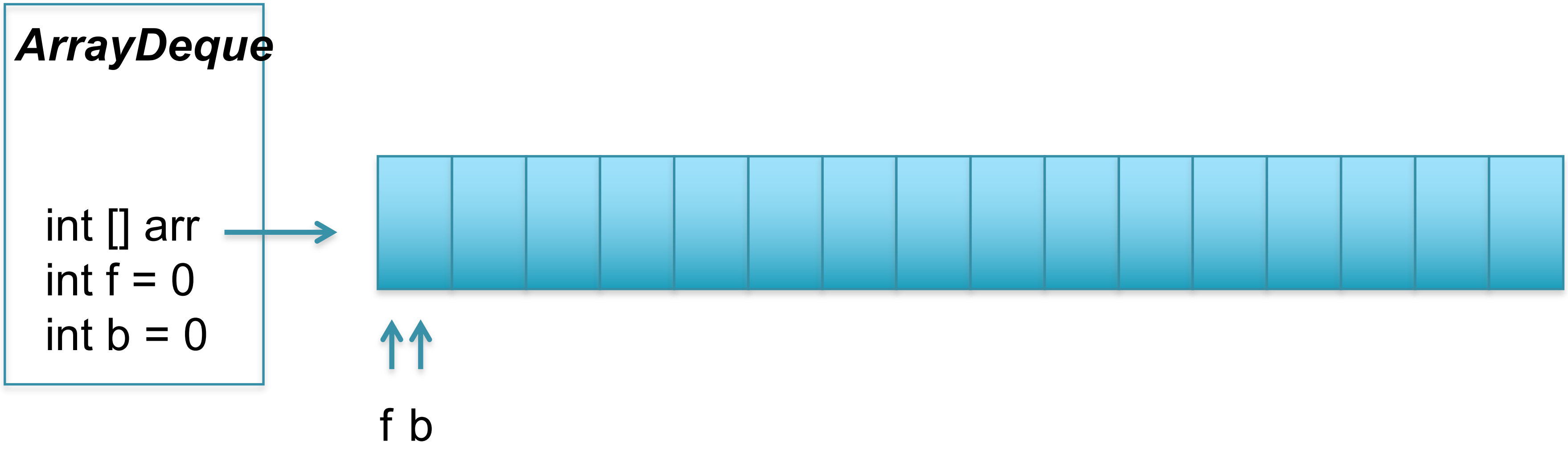
int [] arr

int f

int b

Many of the same tradeoffs as ListQueue vs ArrayQueue

ArrayDequeue



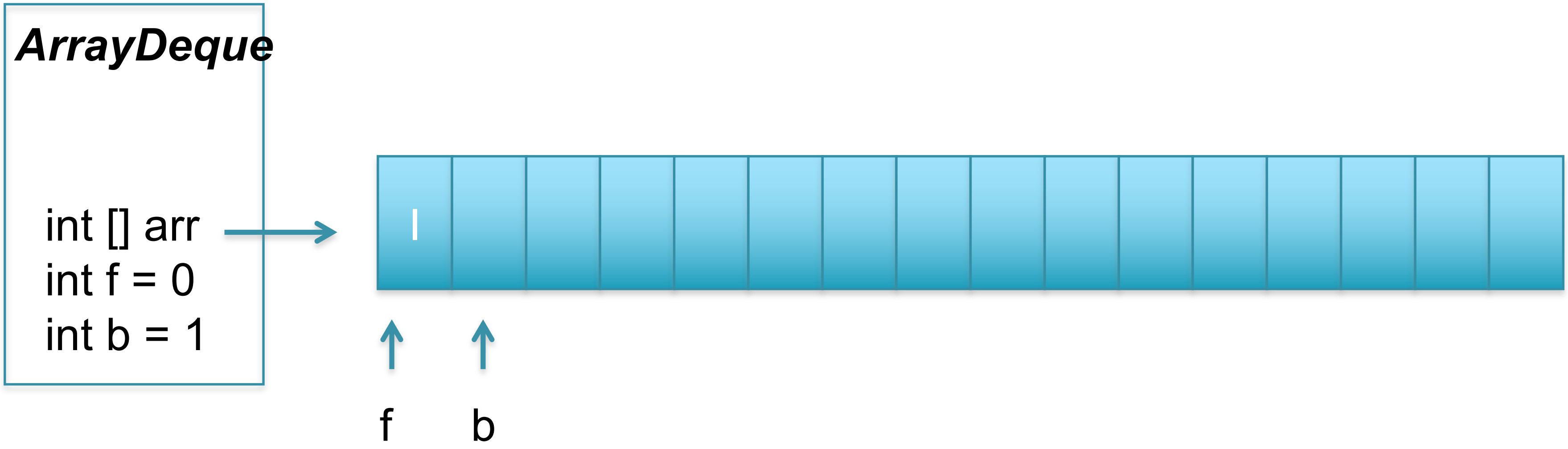
ArrayDeque

int [] arr
int f = 0
int b = 0

f b

deque = new ArrayDequeue();

ArrayDequeue

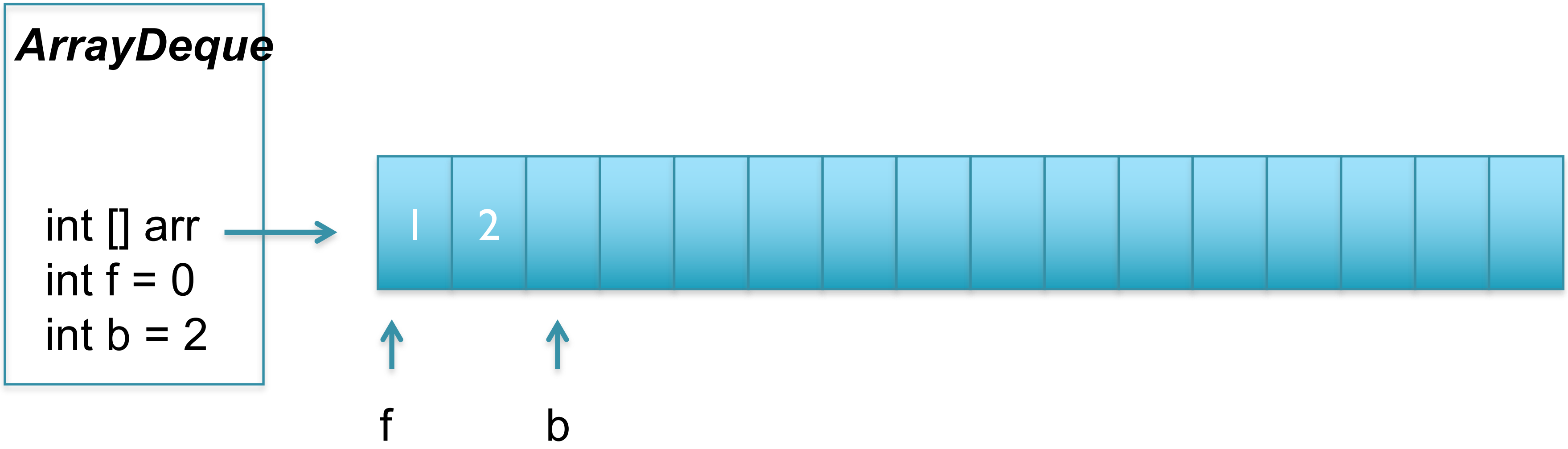


ArrayDeque

int [] arr
int f = 0
int b = 1

```
deque = new ArrayDequeue();
deque.insertBack(1);
```

ArrayDequeue

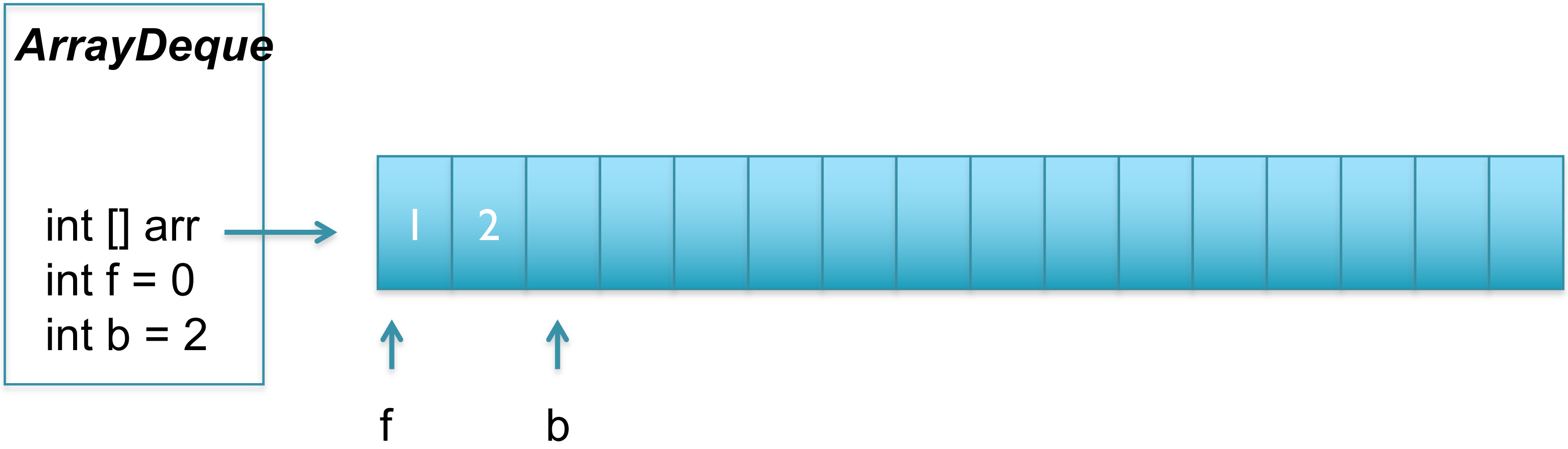


ArrayDeque

int [] arr
int f = 0
int b = 2

```
deque = new ArrayDequeue();
deque.insertBack(1);
deque.insertBack(2);
```

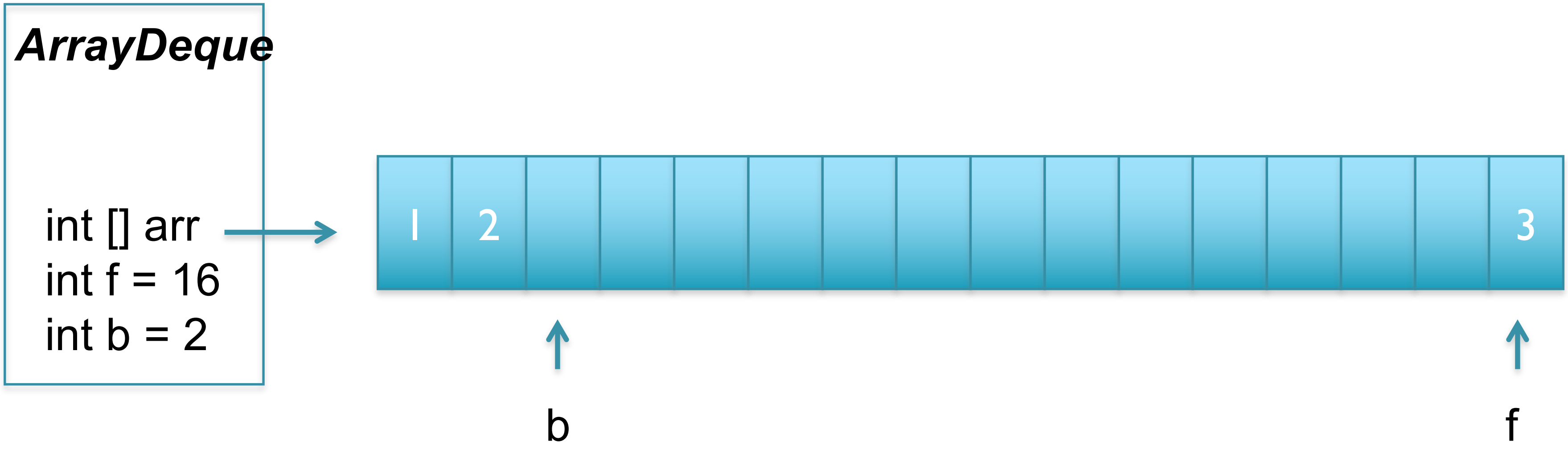
ArrayDequeue



```
deque = new ArrayDequeue();
deque.insertBack(1);
deque.insertBack(2);
```

deque.insertFront(3);

ArrayDequeue

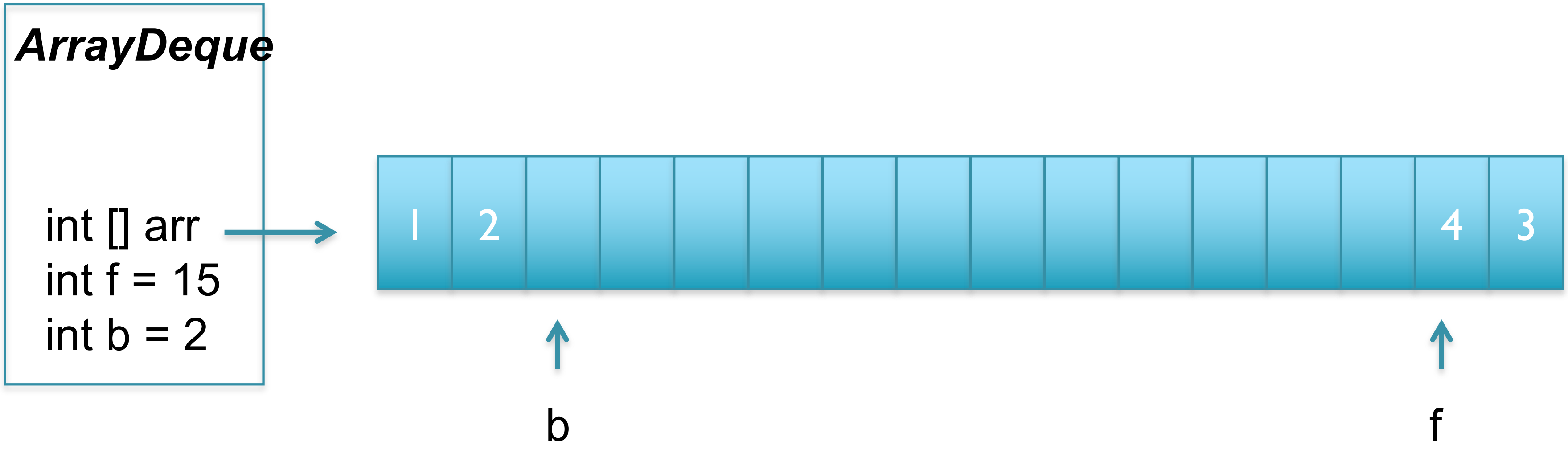


ArrayDeque

int [] arr
int f = 16
int b = 2

deque = new ArrayDequeue();
deque.insertBack(1);
deque.insertBack(2);
deque.insertFront(3);

ArrayDequeue



ArrayDeque

int [] arr
int f = 15
int b = 2

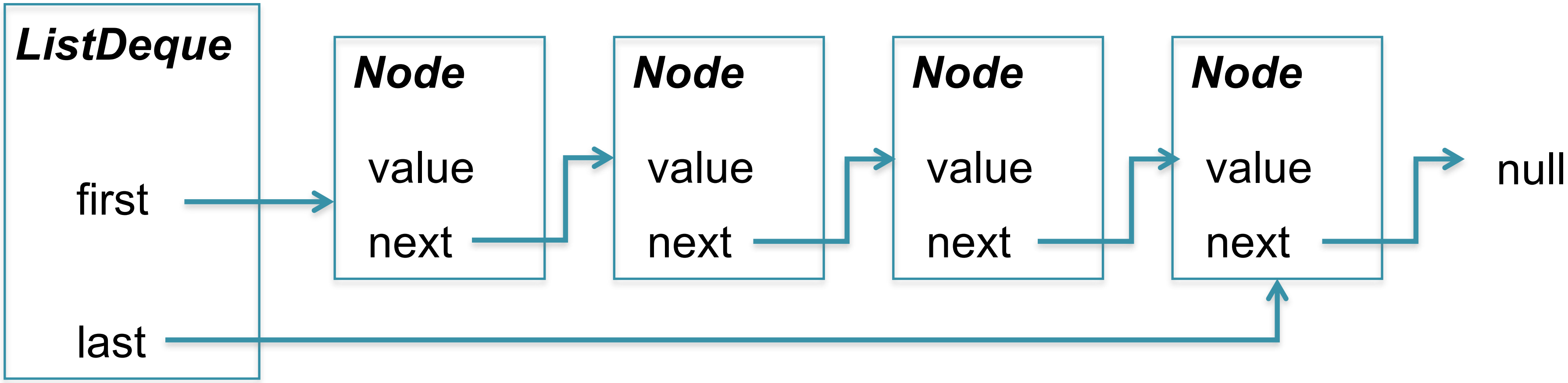
Inserting at front usually means subtract, but gets tricky when we wrap around.

f = (f-1) % arr.length; // depends on how this is implemented for negative numbers

f = (f -1+arr.length) % arr.length; // does the right thing

deque = new ArrayDequeue();
deque.insertBack(1);
deque.insertBack(2);
deque.insertFront(3);
deque.insertFront(4);

ListDequeue



Hint: This wont quite work as shown

Will discuss next time ☺

Next Steps

1. Work on HW3

2. Check on Piazza for tips & corrections!