Apply to be a Meiklejohn!

Meiklejohn applications due February 14 at 5:00pm

tinyurl.com/meikapply

Announcements

- Sections have started!
- Clinic started last week!
  - Tue & Wed 6-8pm in CIT 227 (Motorola)
- Let us know if you don't receive graded Homework **1** via email by Thursday
- **Homework 2** due **Friday 5:00pm**
- **Seamcarve** due **Monday 11:59pm**

Expanding Stacks & Queues

CS16: Introduction to Data Structures & Algorithms

Spring 2019

From Lecture 02

What is Running Time?

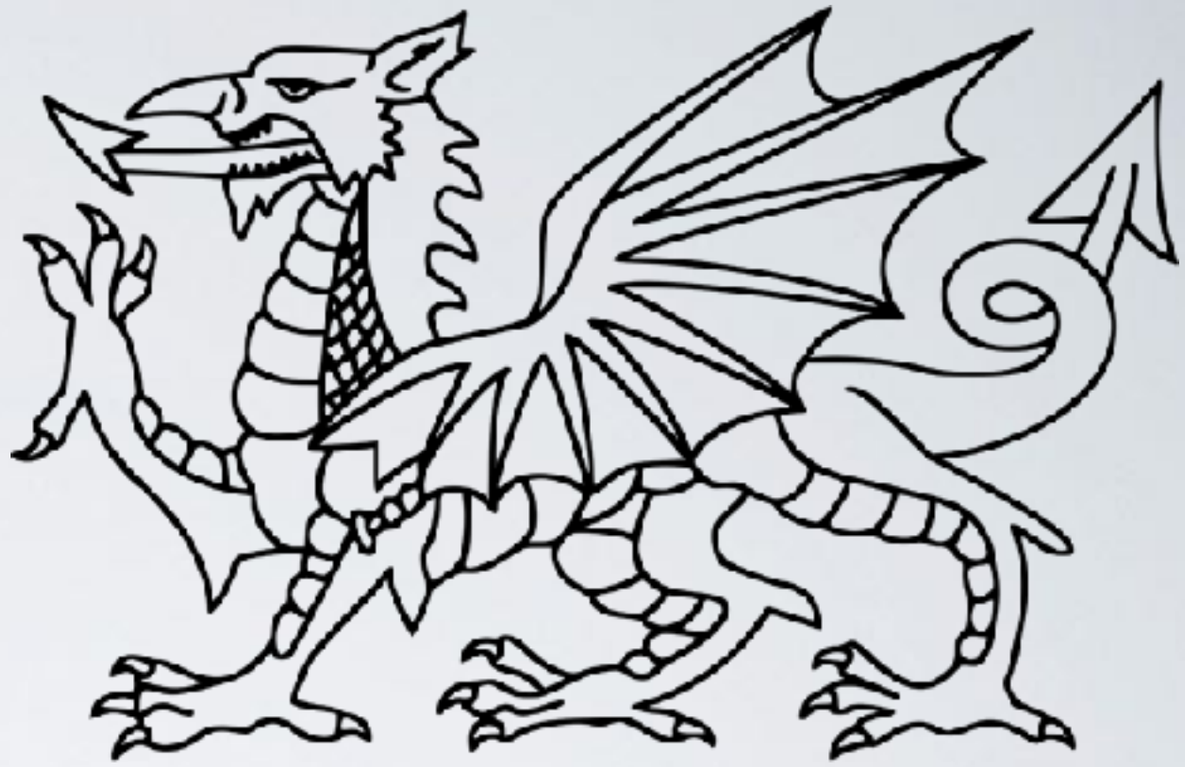
Asymptotic worst-case running time
=
Number of elementary operations
on worst-case input
as a function of input size n
when n tends to infinity

In CS "running time" usually means asymptotic worst-case running time…but not always!

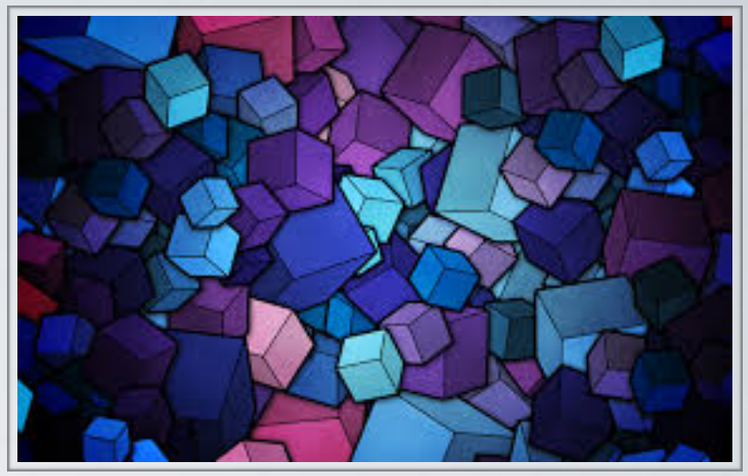
we will learn about other kinds of running times

Outline

- Abstract data types
- Stacks
  - Capped-capacity
  - Expandable
- Amortized analysis
- Queues
  - Expandable queues



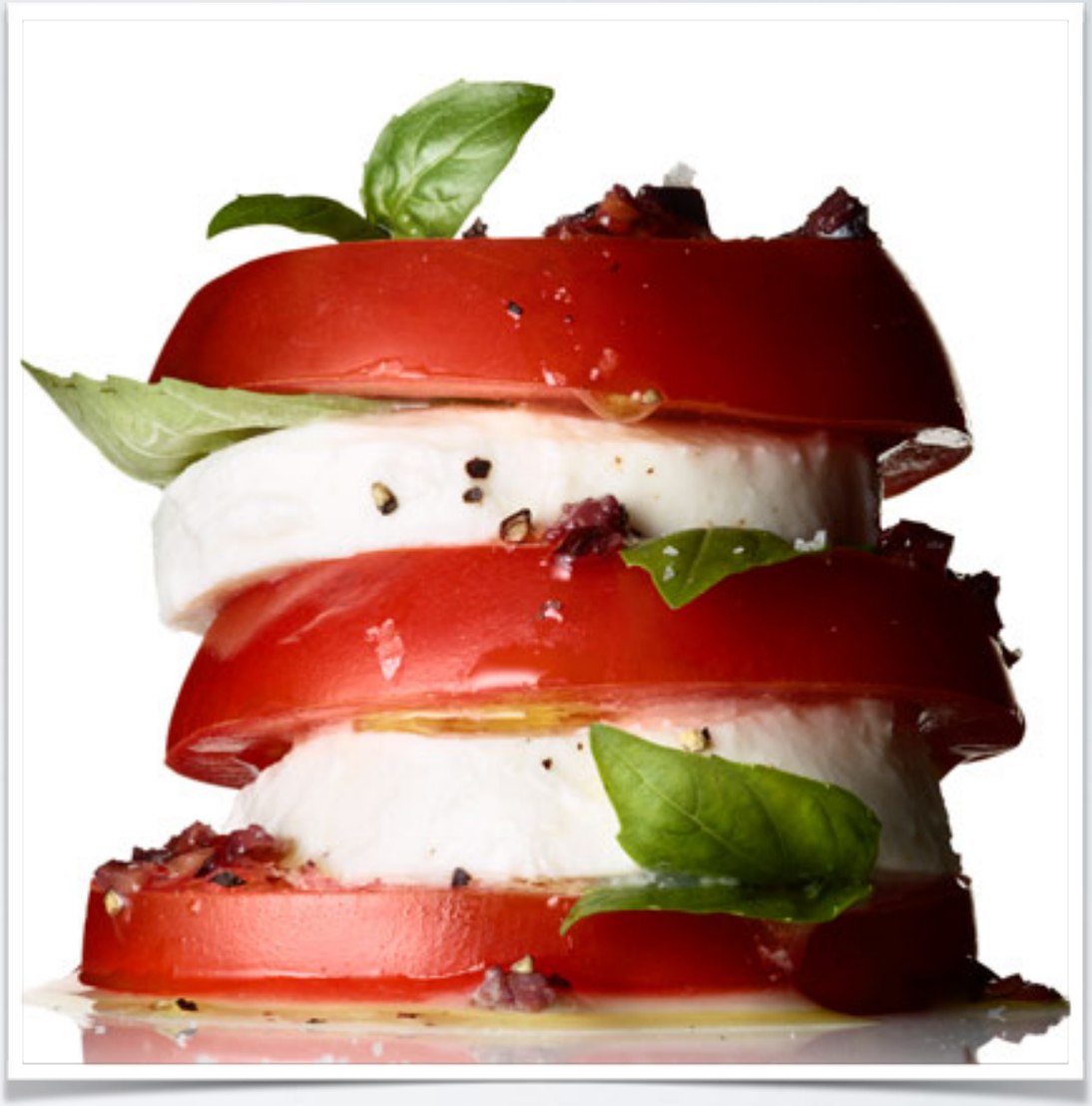
Abstract Data Types



- Abstraction of a data structure
- Specifies "functionality"
  - type of data stored
  - operations it can perform
- Like a Java interface
  - Specifies name & purpose of methods
  - But not implementations

Stacks

- Stores
  - arbitrary objects
- Operations
  - **Push**: adds object
  - **Pop**: returns last object
  - LIFO: last-in first-out
- Implemented
  - Linked list, array, …



Stack ADT

- **push**(object):
  - inserts object
- *object* **pop**( ):
  - returns and removes last inserted object
- *int* **size**( ):
  - returns number objects in stack
- *boolean* **isEmpty**( ):
  - returns TRUE if empty; FALSE otherwise

Capped-capacity Stack

- Array-based Stack
  - Store objects in array
  - keep pointer to last inserted object
- Problem?
  - Size of stack bounded by size of array :-(

Capped-capacity Stack

```
Stack( ):
  data = array of size 20
  count = 0
```

```
function isEmpty( ):
  return count == 0
```

```
function size( ):
      ?????
```

```
function push(object):
      ?????
```

```
function pop( ):
      ?????
```

Activity #1

Capped-capacity Stack

```
Stack( ):
  data = array of size 20
  count = 0
```

```
function isEmpty( ):
  return count == 0
```

```
function size( ):
      ?????
```

```
function push(object):
      ?????
```

```
function pop( ):
      ?????
```

Activity #1

2 min

Capped-capacity Stack

```
Stack( ):
  data = array of size 20
  count = 0
```

```
function isEmpty( ):
  return count == 0
```

```
function size( ):
      ?????
```

```
function push(object):
      ?????
```

```
function pop( ):
      ?????
```

Activity #1

1 min

Capped-capacity Stack

```
Stack( ):
  data = array of size 20
  count = 0
```

```
function isEmpty( ):
  return count == 0
```

```
function size( ):
      ?????
```

```
function push(object):
      ?????
```

```
function pop( ):
      ?????
```

Activity #1

0 min

Capped-capacity Stack

```
Stack( ):
  data = array of size 20
  count = 0
```

```
function push(object):
  if count < 20:
    data[count] = object
    count++
  else:
    error(“overfull”)
```

O(1)

O(1)

```
function size( ):
  return count
```

O(1)

```
function isEmpty( ):
  return count == 0
```

```
function pop( ):
  if count == 0:
    error(“empty stack”)
  else:
    count--
    return data[count]
```

O(1)

Expandable Stack

- Capped-capacity stack is fast
  - but not useful in practice
- How can we design an *uncapped* Stack?
- Strategy #1: **Incremental**
  - increase size of array by constant **c** when full
- Strategy #2: **Doubling**
  - double size of array when full

Arrays can't be resized!
Can only be copied

Expandable Stack

```
Stack( ):
   data = array of size 20
   count = 0
   capacity = 20
```

What is the runtime?

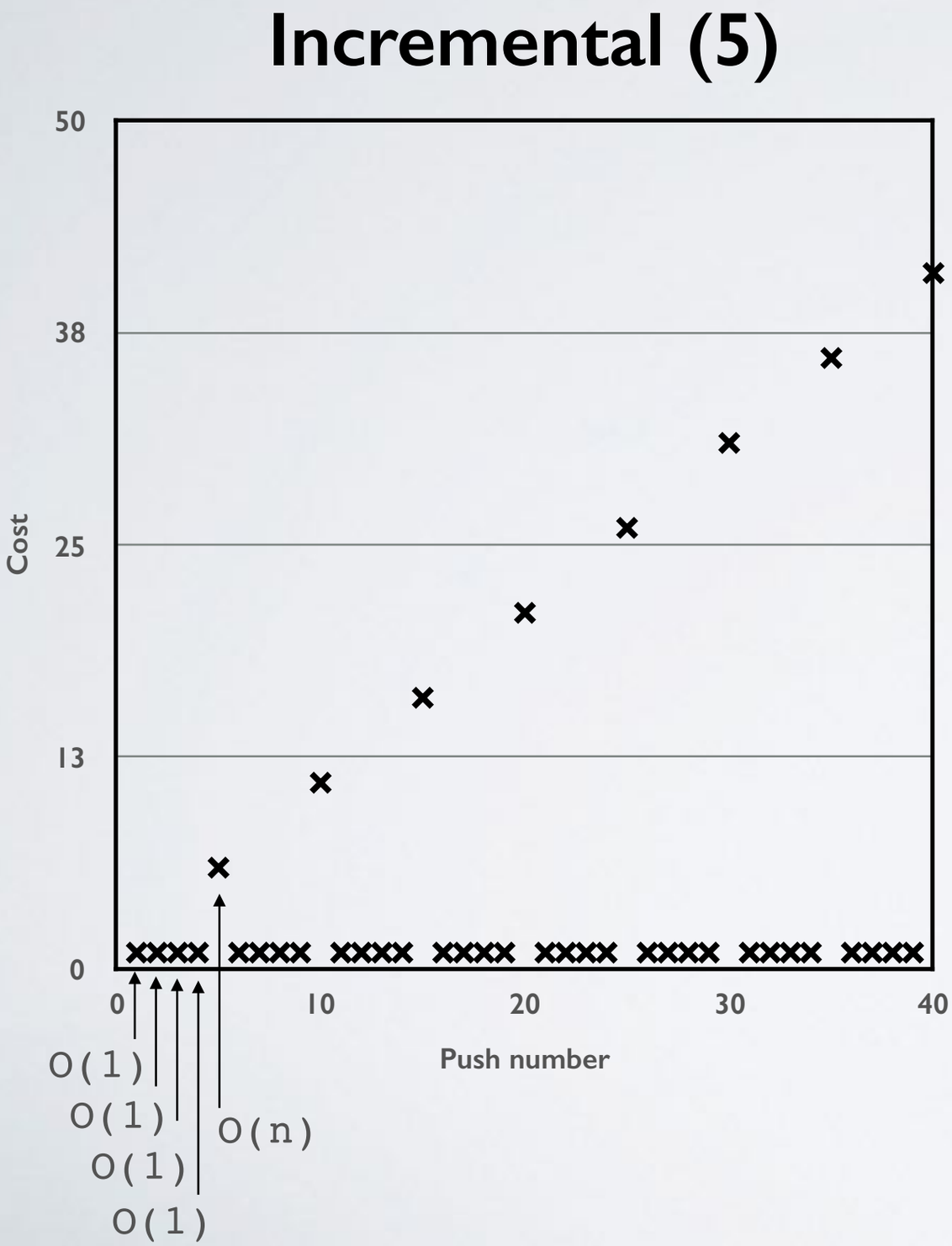
```
function push(object):
   data[count] = object
   count++
   if count == capacity
      new_capacity = capacity + c /* incremental */
                   = capacity * 2 /* doubling */
      new_data = array of size new_capacity
      for i = 0 to capacity - 1
         new_data[i] = data[i]
      capacity = new_capacity
      data = new_data
```

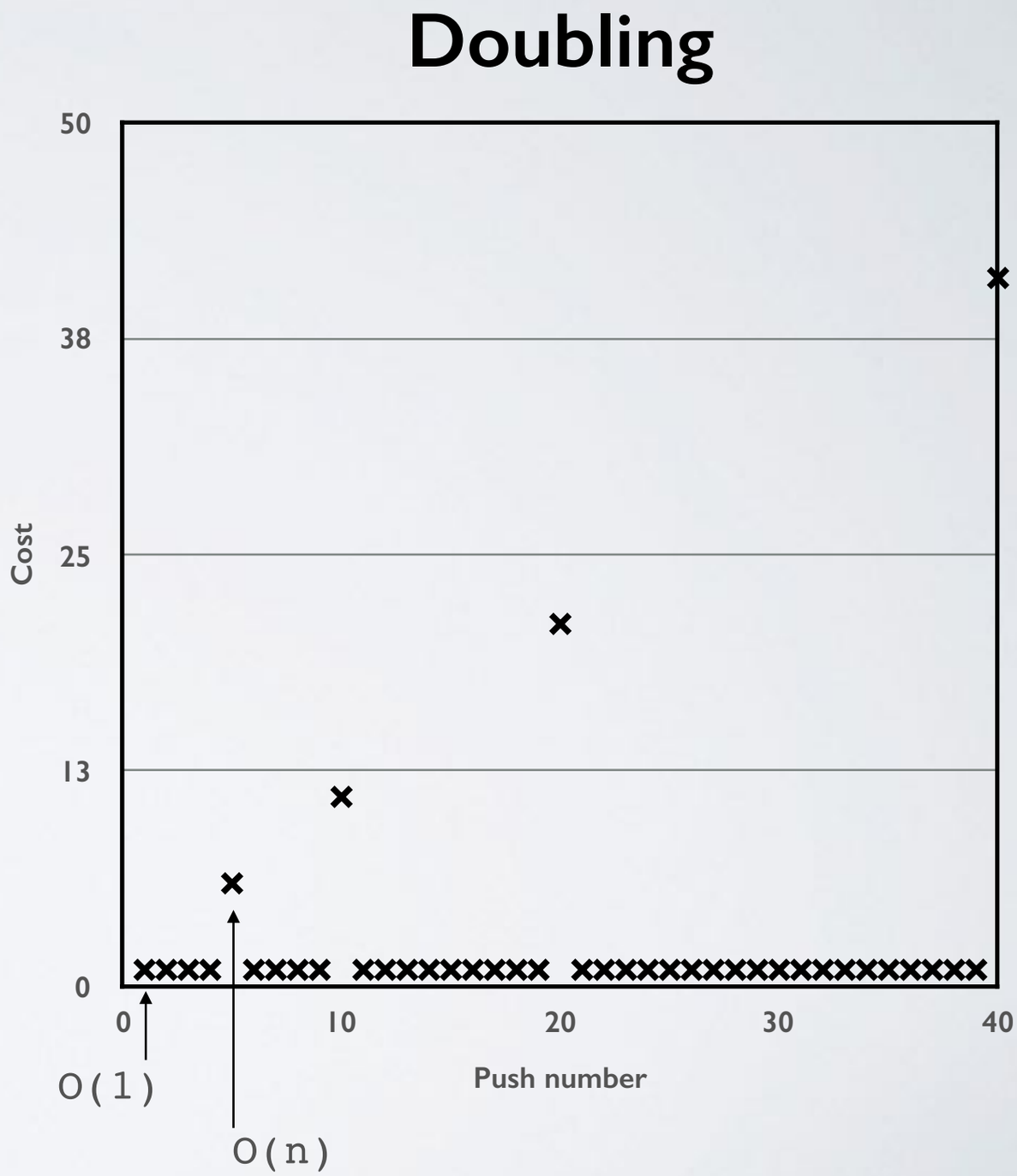
Expandable Stack

```
function push(object):
  data[count] = object
  count++
  if count == capacity
    new_capacity = capacity + c /* incremental */
                 = capacity * 2 /* doubling */
    new_data = array of size new_capacity
    for i = 0 to capacity - 1
      new_data[i] = data[i]
    capacity = new_capacity
    data = new_data
```

- Runtime when not expanding is **O(1)** & runtime when expanding is **O(n)**
- When does it expand?
  - after **n** pushes, where **n** is capacity of array

Incremental & Doubling





Incremental & Doubling

- What is the running time of incremental?
  - `O(1)` or `O(n)`?
- What is the running time of doubling?
  - `O(1)` or `O(n)`?
- It depends…

What's going on?

Expandable Stack

```
Stack( ):
   data = array of size 20
   count = 0
   capacity = 20
```

Run time depends on count which depends on *previous* pushes

```
function push(object):
   data[count] = object
   count++
   if count == capacity
      new_capacity = capacity + c /* incremental */
                   = capacity * 2 /* doubling */
      new_data = array of size new_capacity
      for i = 0 to capacity - 1
         new_data[i] = data[i]
      capacity = new_capacity
      data = new_data
```

Incremental & Doubling

- What is the running time of incremental?
  - `O(1)` or `O(n)`?
- What is the running time of doubling?
  - `O(1)` or `O(n)`?
- It depends…

Measure cost on *sequence* of calls not a single call!

Towards Amortized Analysis

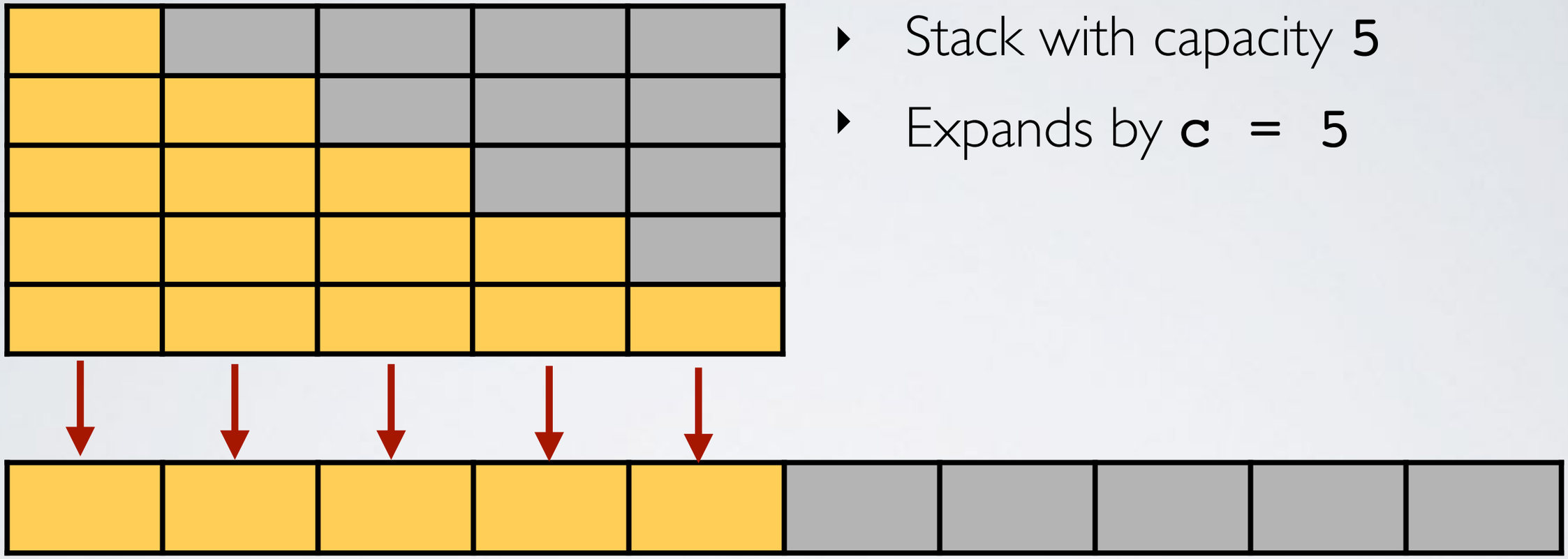
- For certain algorithms better to measure
  - **total** running time on **sequence** of calls
  - instead of running time on single call
    - `S(n)`: total #calls on **sequence** of `n` calls
    - **Not runtime on a single input of size n**
    - Usually the case for data structure operations
- ex: Stack
  - `S(n)`: cost push `#1` + cost push `#2` + … + cost push `#n`

Amortized Analysis

- Instead of reporting **total** cost of sequence
  - report cost of sequence **per call**

$$\frac{S(n)}{n}$$

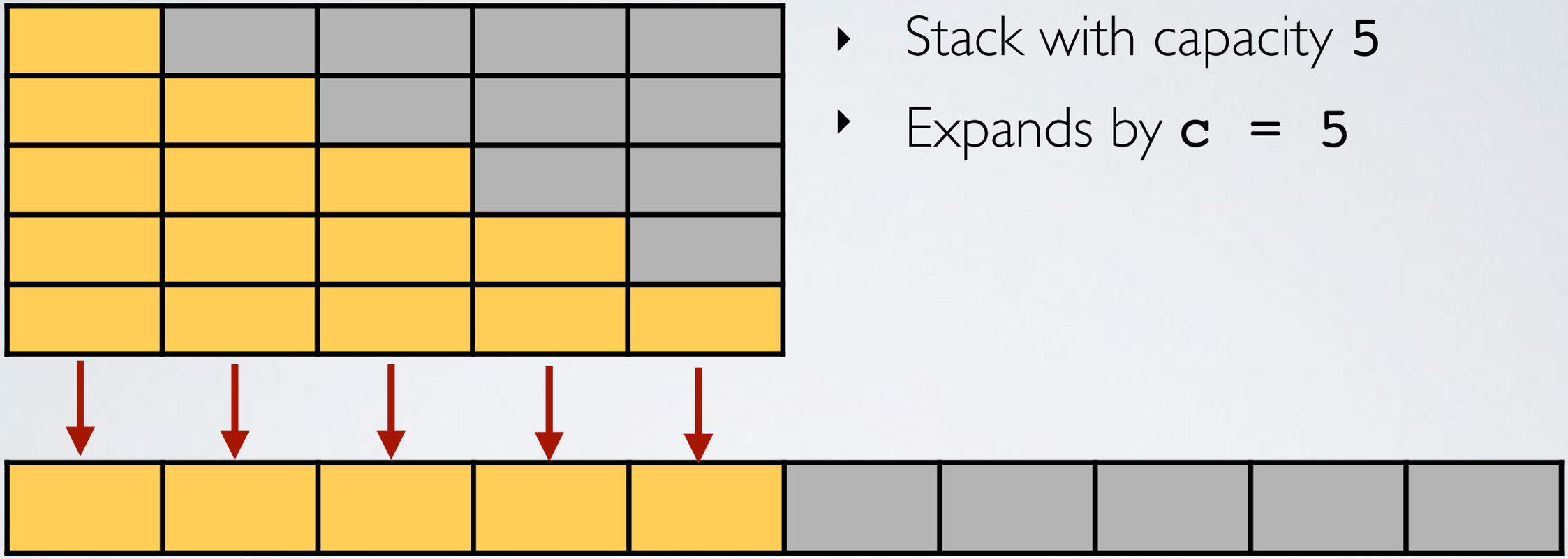
Amortized Analysis of Incremental



- Stack with capacity `5`
- Expands by `c = 5`

- 5th push brings to capacity
  - Objects copied to new array of size `5+c = 10`
  - Total cost per push over 5 pushes?

Amortized Analysis of Incremental



- Stack with capacity `5`
- Expands by `c = 5`

$$\frac{S(n)}{n} \leq \frac{5+c}{5} \leq \frac{5+5}{5} = 2$$

Is each push O(1)?

Cost of 5 store ops

Cost of expansion

Amortized Analysis of Incremental

- What if we push `5` more objects?
- `O(1)` until 10th push brings to capacity
  - then all `10` objects copied to new array
  - of size `10+c = 15`

$$\frac{S(n)}{n} = \frac{10 + c + 2c}{10} \leq \frac{10 + 5 + 10}{10} = 2.5$$

Cost of 10 pushes

Cost of 1st expansion

Cost of 2nd expansion

Amortized Analysis of Incremental

$$\frac{S(n)}{n} = \frac{S(10)}{10} = \frac{10 + c + 2c}{10} = \frac{10 + 5 + 10}{10} = 2.5$$

$$\frac{S(n)}{n} = \frac{S(15)}{15} = \frac{15 + c + 2c + 3c}{15} = \frac{15 + 5 + 10 + 15}{15} = 3$$

$$\boxed{\frac{S(n)}{n} = \frac{S(20)}{20} = ?}$$

Activity #2

Amortized Analysis of Incremental

$$\frac{S(n)}{n} = \frac{S(10)}{10} = \frac{10 + c + 2c}{10} = \frac{10 + 5 + 10}{10} = 2.5$$

$$\frac{S(n)}{n} = \frac{S(15)}{15} = \frac{15 + c + 2c + 3c}{15} = \frac{15 + 5 + 10 + 15}{15} = 3$$

$$\boxed{\frac{S(n)}{n} = \frac{S(20)}{20} = ?}$$

Activity #2

1 min

Amortized Analysis of Incremental

$$\frac{S(n)}{n} = \frac{S(10)}{10} = \frac{10 + c + 2c}{10} = \frac{10 + 5 + 10}{10} = 2.5$$

$$\frac{S(n)}{n} = \frac{S(15)}{15} = \frac{15 + c + 2c + 3c}{15} = \frac{15 + 5 + 10 + 15}{15} = 3$$

$$\boxed{\frac{S(n)}{n} = \frac{S(20)}{20} = ?}$$

Activity #2

0 min

Amortized Analysis of Incremental

$$\frac{S(n)}{n} = \frac{S(10)}{10} = \frac{10 + c + 2c}{10} = \frac{10 + 5 + 10}{10} = 2.5$$

$$\frac{S(n)}{n} = \frac{S(15)}{15} = \frac{15 + c + 2c + 3c}{15} = \frac{15 + 5 + 10 + 15}{15} = 3$$

$$\frac{S(n)}{n} = \frac{S(20)}{20} = \frac{20 + c + 2c + 3c + 4c}{20} = \frac{20 + 5 + 10 + 15 + 20}{20} = 3.5$$

- So on and so forth…
- Looks linear…

Amortized Analysis of Incremental

n pushes w/o exp.

cost of exp. # n/c

$$S(n) = n + c + 2c + 3c + \cdots + \frac{n}{c} \cdot c$$

$$= n + c \cdot \left(1 + 2 + \cdots + \frac{n}{c}\right)$$ ←----**factoring out c**

$$= n + c \cdot \frac{1}{2} \cdot \left(\frac{n}{c}\left(\frac{n}{c} + 1\right)\right)$$ ←----**using:**

$$(1 + 2 + \cdots + k) = \frac{k \cdot (k+1)}{2}$$

$$= n + \frac{n^2/c + n}{2}$$ ←---- **distributing & simplifying:**

$$= O(n^2)$$

$$\frac{S(n)}{n} = O(n)$$

Amortized Analysis of Incremental

- Summary
  - Total cost of **n** pushes: $S(n) = O(n^2)$
  - Amortized cost of **n** pushes: $S(n)/n = O(n)$

Amortized Analysis of Doubling

- ex: doubling stack with initial capacity `5`?
  - pushes are `O(1)` until 5th push
  - then linear in capacity

cost of pushes w/o exp

$$\frac{S(n)}{n} = \frac{S(5)}{5} = \frac{5+5}{5} = 2$$ ← **cost of exp**

$$\frac{S(n)}{n} = \frac{S(10)}{10} = \frac{10+5+10}{10} = 2.5$$ ← **cost of exp #2**

$$\frac{S(n)}{n} = \frac{S(20)}{20} = \frac{20+5+10+20}{20} = 2.75$$ ← **cost of exp #3**

Amortized Analysis of Doubling

cost of n pushes

cost of last exp

cost of second to last exp

$$S(n) = n + n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}}$$

cost of exp #1

$$= n + n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right)$$

$$< n + n \cdot 2$$

using: $\lim_{k \to \infty} \sum_{i=0}^{k} \frac{1}{2^i} = 2$

$$= 3n$$

Assume:
c=2
$n=2^k$

$$\frac{S(n)}{n} = O(1)$$

Amortized Analysis

- Summary for Incremental
  - Total cost of `n` pushes: $S(n) = O(n^2)$
  - Amortized cost of `n` pushes: $S(n)/n = O(n)$
- Summary for Doubling
  - Total cost of `n` pushes: $S(n) = O(n)$
  - Amortized cost of n pushes: $S(n)/n = O(1)$

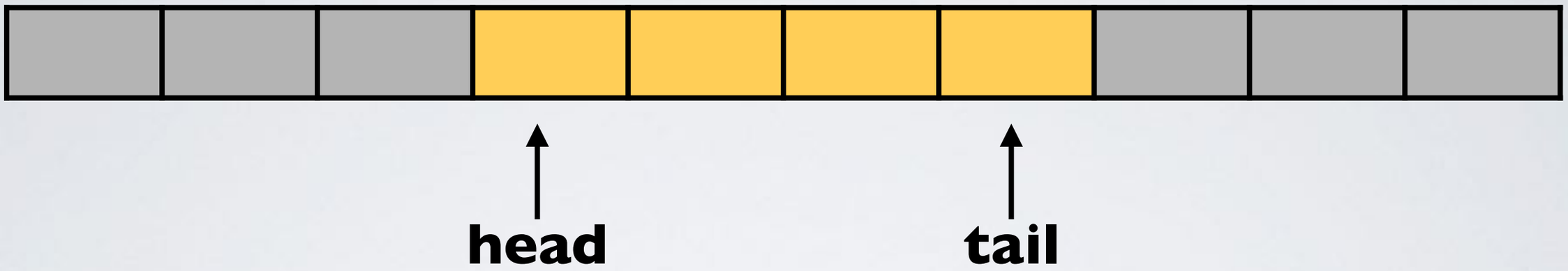
Way to Think about Amortized

- Each fast operation adds some credit
- Need enough credits to execute slow operation

Queue ADT

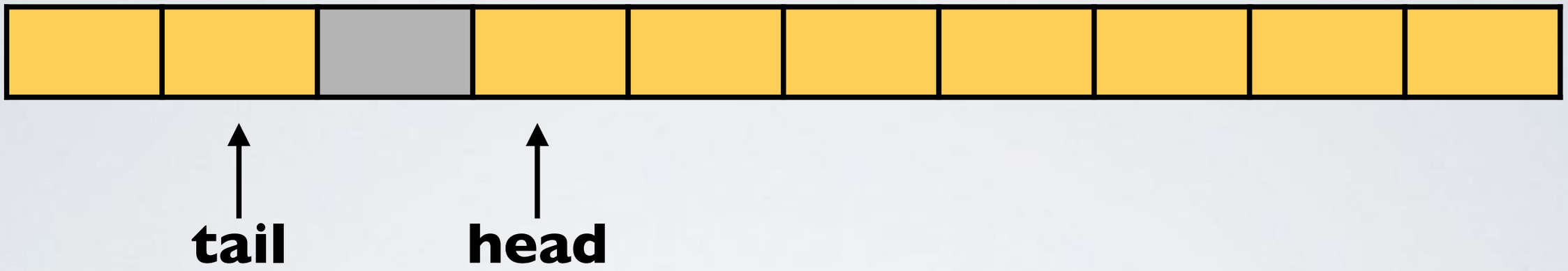
- **enqueue**(object):
  - inserts object
- *object* **dequeue**( ):
  - returns and removes first inserted object
- *int* **size**( ):
  - returns number objects in queue
- *boolean* **isEmpty**( ):
  - returns TRUE if empty; FALSE otherwise

Expandable Queue



- Can be implemented with expandable array
  - need to keep track of head and tail
- What happens when tail reaches end?
  - Is the queue full?
- So when should we expand array?

Expandable Queue



- Wrap around until array is completely full
- When expanding re-order objects properly

Expandable Queue

```
function enqueue(object):
  if size == capacity
    double array and copy contents
    reset head and tail pointers
  data[tail] = object
  tail = (tail + 1) % capacity
  size++
```

```
function dequeue( ):
  if size == 0
    error(“queue empty”)
  element = data[head]
  head = (head + 1) % capacity
  size--
  return element
```

$$\frac{S(n)}{n} = O(1)$$