Expanding Stacks & Queues

CS16: Introduction to Data Structures & Algorithms
Spring 2020
Outline

- Abstract data types
- Stacks
  - Capped-capacity
  - Expandable
- Amortized analysis
- Queues
  - Expandable queues
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
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
- Can be implemented with
  - Linked lists, arrays, …
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
  - Stores objects in array
  - keeps pointer to last inserted object

- Problem?
  - Size of the stack is bounded by size of array :-(

7
Capped-Capacity Stack

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

isEmpty( ):  
return count == 0

size( ):  

push(object):  

pop( ):  

Activity #1
Capped-capacity Stack

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

function size():

function isEmpty():
  return count == 0

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 size( ):  
    return count  

function isEmpty( ):  
    return count == 0  

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

function pop( ):  
    if count == 0:  
        error("empty stack")  
    else:  
        count--  
        return data[count]
Expandable Stack

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

Arrays can’t be resized!
Can only be copied
Expandable Stack

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

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
```

What is the runtime?
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 (5)

Doubling

O(1)    O(1)    O(n)

O(1)    O(1)    O(n)

O(1)    O(1)    O(n)
Incremental & Doubling

- What are the worst-case runtimes?
  - incremental: $O(n)$
  - doubling: $O(n)$

- But are they really the same?
What’s going on?
Expandable Stack

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

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

Run time depends on count which depends on # of previous pushes
Incremental & Doubling

- Worst-case analysis overestimates runtime
  - for algorithms that are fast \textit{most} of the time…
  - …and slow \textit{some} of the time
- For these algorithms we need an alternative
  - Amortized analysis!

Measure cost on \textit{sequence} of calls not a single call!
Towards Amortized Analysis

- For certain algorithms it’s better to measure
  - total running time on sequence of calls
  - instead of measuring on a single call
  - \( S(n) \): total #calls on sequence of \( n \) calls
  - **Not runtime on a single input of size \( n \)**

- For a 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
Amortized Analysis of Incremental

- Stack with start capacity $c = 5$
- Expands by $e = 5$

- 5th push brings to capacity
  - Objects copied to new array of size $c + e = 5 + 5 = 10$
  - Cost per push over 5 pushes?
Amortized Analysis of Incremental

- Stack with start capacity $c = 5$
- Expands by $e = 5$

Is each push $O(1)$?

\[
S(n) = \frac{c + c}{n} = \frac{5 + 5}{5} = 2
\]
What if we push 10 objects?

\[
S(n) = \frac{c + c + e + (c + e)}{n} = 10
\]

1st batch of pushes

1st expansion

2nd batch of pushes

2nd expansion

- pushes
- expansions

\(c=5\)
\(e=5\)
Amortized Analysis of Incremental

What if we push 10 objects?

\[
S(n) = \frac{c + c + e + (c + e)}{n} = \frac{10}{10} = \frac{c + e + c + (c + e)}{10} = \frac{10 + 5 + (5 + 5)}{10} = \frac{2.5}{10}
\]

c = 5

e = 5
Amortized Analysis of Incremental Activity #2

\[
S(10) = \frac{c + e + c + (c + e)}{10} = \frac{10 + 5 + 10}{10} = \frac{25}{10} = 2.5
\]

\[
S(15) = \frac{c + e + e + c + (c + e) + (c + e + e)}{15} = \frac{15 + 5 + 10 + 15}{15} = \frac{45}{15} = 3
\]

\[
\frac{S(20)}{20} = ?
\]

[Boxes: pushes, expansions]
Amortized Analysis of Incremental

$$\frac{S(10)}{10} = \frac{c + e + c + (c + e)}{10} = \frac{10 + 5 + 10}{10} = \frac{25}{10} = 2.5$$

$$\frac{S(15)}{15} = \frac{c + e + e + c + (c + e) + (c + e + e)}{15} = \frac{15 + 5 + 10 + 15}{15} = \frac{45}{15} = 3$$

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

 pushes

 expansions

Activity #2

1 min

c=5
e=5
Amortized Analysis of Incremental Activity #2

\[
\frac{S(10)}{10} = \frac{c + e + c + (c + e)}{10} = \frac{10 + 5 + 10}{10} = \frac{25}{10} = 2.5
\]

\[
\frac{S(15)}{15} = \frac{c + e + e + c + (c + e) + (c + e + e)}{15} = \frac{15 + 5 + 10 + 15}{15} = \frac{45}{15} = 3
\]

\[
\frac{S(20)}{20} = ?
\]

**pushes**

**expansions**

Activity #2

\(c=5\)

\(e=5\)
Amortized Analysis of Incremental

\[
S(10) = \frac{c + e + c (c + e)}{10} = \frac{10 + 5 + 10}{10} = \frac{25}{10} = 2.5
\]

\[
S(15) = \frac{c + e + e + c + (c + e) + (c + e + e)}{15} = \frac{15 + 5 + 10 + 15}{15} = \frac{45}{15} = 3
\]

\[
S(20) = \frac{c + e + e + e + c + (c + e) + (c + e + e) + (c + e + e + e)}{20} \quad 20 + 5 + 10 + 15 + 20 \quad \frac{70}{20} = 3.5
\]

\(c=5\)
\(e=5\)

pushes
expansions
Amortized Analysis of Incremental

\[ S(n) = \underbrace{c + e + \ldots + e}_n + c + (c + e) + (c + 2e) + (c + 3e) + \ldots \]

\[ = n + c + (c + e) + (c + 2e) + (c + 3e) + \ldots \]

To make things simpler, let’s assume \( e = c \)

\[ = n + c + 2c + 3c + 4c + \ldots + \frac{n}{c} \cdot c \]

\# of expansions
(1 expansion per \( c \) (or \( e \)) pushes)
Amortized Analysis of Incremental

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

\[ \begin{align*}
&= n + c \cdot \left( 1 + 2 + \cdots + \frac{n}{c} \right) \\
&= n + c \cdot \frac{1}{2} \cdot \left( \frac{n}{c} \left( \frac{n}{c} + 1 \right) \right) \\
&= n + \frac{n^2/c + n}{2} \\
&= O(n^2)
\]

\[ \frac{S(n)}{n} = O(n) \]
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 Double
Amortized Analysis of Doubling

- Doubling stack with initial capacity $c=5$?

\[
\frac{S(n)}{n} = \frac{S(5)}{5} = \frac{5 + 5}{5} = 2
\]

\[
\frac{S(n)}{n} = \frac{S(10)}{10} = \frac{10 + 5 + 10}{10} = 2.5
\]

\[
\frac{S(n)}{n} = \frac{S(20)}{20} = \frac{20 + 5 + 10 + 20}{20} = 2.75
\]
Amortized Analysis of Doubling

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

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

\[ < n + n \cdot 2 \]

\[ = 3n \]

Assume:
\[ c=2 \]
\[ n=2^k \]

\[ \frac{S(n)}{n} = O(1) \]

Using:
\[ \lim_{k \to \infty} \sum_{i=0}^{k} \frac{1}{2^i} = 2 \]
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: $S(n)/n = O(1)$
Expandable Queue
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

\[
\frac{S(n)}{n} = O(1)
\]