Amortized Analysis

David Kauchak
cs140
Spring 2024

1

Admin

Group work today and tomorrow

Assignment 3

2

Extensible array

Sequential locations in memory in linear order

Elements are accessed via index

- Access of particular indices is O(1)

Say we want to implement an array that supports *add* (i.e. *append*)

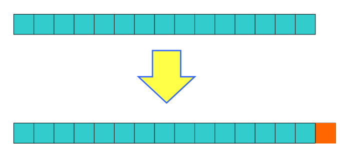
- ArrayList in Java
- lists in Python, perl, Ruby, …

How can we do it?

3

Extensible array

Idea 1: Each time we call *add*, create a new array one element larger, copy the data over and add the element



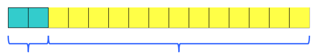
Running time: $\Theta(n)$

4

Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

For example: new ArrayList(2)

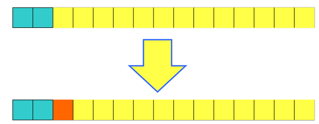


5

Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

Adding an item:



Running time: $\Theta(1)$ Problems?

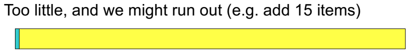
6

Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

How much extra space do we allocate?

Too little, and we might run out (e.g. add 15 items)



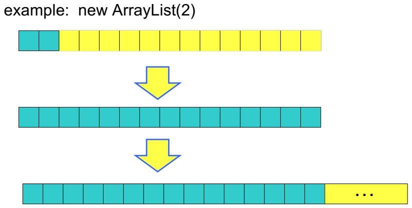
Too much, and we waste lots of memory Ideas?

7

Extensible array

Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

For example: new ArrayList(2)

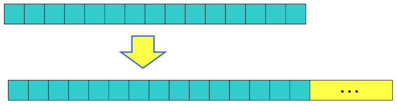


8

Extensible array

Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

For example: new ArrayList(2)



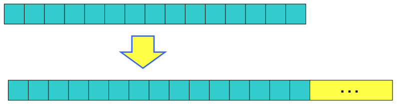
Running time: Θ(n)

9

Extensible array

Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

For example: new ArrayList(2)



How much extra memory should we allocate?

10

Extensible array

Challenge: most of the calls to *add* will be O(1)

How else might we talk about runtime?

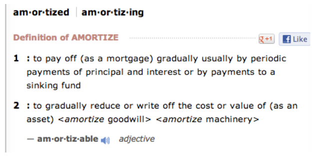
What is the **average worst-case** running time of *a sequence of adds*?

- Note this is different than the *average-case* running time

11

Amortized analysis

What does "amortize" mean?



am·or·tized | **am·or·tiz·ing**

Definition of AMORTIZE

1 **:** to pay off (as a mortgage) gradually usually by periodic payments of principal and interest or by payments to a sinking fund

2 **:** to gradually reduce or write off the cost or value of (as an asset) <*amortize* goodwill> <*amortize* machinery>

— **am·or·tiz·able** *adjective*

12

Amortized analysis

There are many situations where the worst case running time is bad

However, if we average the operations over *n* operations, the average time is more reasonable

This is called *amortized* analysis

- This is different than average-case running time, which requires probabilistic reasoning about input
- The worse case running time doesn't change

13

What are the costs?

Assume we start with an array of size 1 and double each time

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost:

Count: 1) inserting element and 2) copying elements

17

What are the costs?

Assume we start with an array of size 1 and double each time

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost: 1 2 3 1 5 1 1 1 9 1

Count: 1) inserting element and 2) copying elements

18

What are the costs?

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

basic cost: 1 1 1 1 1 1 1 1 1 1

double cost: 0 1 2 0 4 0 0 0 8 0

Count: 1) inserting element and 2) copying elements

19

What are the costs?

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
basic cost:	1	1	1	1	1	1	1	1	1	1
double cost:	0	1	2	0	4	0	0	0	8	0

What is the sum of basic cost for n operations?

What is the sum of the copy cost for n operations?

20

Amortized analysis

More generally:

total_cost(n) = basic_cost(n) + double_cost(n)

basic_cost(n) = n

double_cost(n) = 1 + 2 + 4 + … + n/2 + n = 2n

total_cost(n) = 3n

over n operations:
amortized *O(1)*

21

Amortized analysis vs. worse case

What is the worse case of *add*?

- Still θ(n)
- If you have an application that needs it to be O(1), this implementation **will not work!**

amortized analysis give you the cost of *n* operations (i.e., average cost) **not** the cost of any individual operation

22

Extensible arrays

What if instead of doubling the array, we increase the array by a fixed amount (call it *k*) each time

Is the amortized run-time still O(1)?

- No!
- Why?

23

Amortized analysis

Consider the cost of *n* insertions for some constant *k*

total_cost(n) = basic_cost(n) + double_cost(n)

basic_cost(n) = n

$$\text{double\_cost(n)} = k+2k+3k+4k+5k+\ldots+n$$
$$= \sum_{i=1}^{n/k} ki$$
$$= k\sum_{i=1}^{n/k} i$$
$$= k\frac{\frac{n}{k}\left(\frac{n}{k}+1\right)}{2} = \Omega(n^2)$$

24

Amortized analysis

Consider the cost of *n* insertions for some constant *k*

$$total\_cost(n) = n + \Omega(n^2)$$
$$= \Omega(n^2)$$

amortized *Ω(n)!*

25

Accounting method

Each operation has an amount we charge

(this will become the amortized run-time)

if the actual cost of the operation **is less than** the charge, put the excess in the bank

can never have < 0 in the bank

if the actual cost of the operation **is more than** the charge, get the extra needed from the bank

Key idea: charge more for low-cost operations and save that up to offset the cost of expensive operations

26

Insertion: 1 2 3 4 5 6 7 8 9 10

size: 1 2 4 4 8 8 8 8 16 16

cost: 1 2 3 1 5 1 1 1 9 1

bank:

How much should we pay for each insert?

28

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

Try insert: 2

29

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

Try insert: 2

How much is left?

30

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1									

Try insert: 2

31

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1									

Try insert: 2

How much is left?

32

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1								

Try insert: 2

33

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1	0							

Try insert: 2

34

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1	0	1						

Try insert: 2

35

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1	0	1						

Try insert: 2

How much is left?

36

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	1	1	0	1						

Try insert: 2

-2!!

37

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

Try insert: ??

38

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:										

Try insert: 3

How much is left?

39

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2									

Try insert: 3

40

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3								

Try insert: 3

41

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3							

Try insert: 3

42

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3							

Try insert: 3

43

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3	5						

Try insert: 3

44

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3	5	3	5	7	9		

Try insert: 3

45

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3	5	3	5	7	9	3	

Try insert: 3

46

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3	5	3	5	7	9	3	

Try insert: 3

Will this work??

47

Insertion:	1	2	3	4	5	6	7	8	9	10
size:	1	2	4	4	8	8	8	8	16	16
cost:	1	2	3	1	5	1	1	1	9	1
bank:	2	3	3	5	3	5	7	9		

last copy happened here

Try insert: 3

1: pay for our operation
Getting ready for the copy:
1: pay for our copy
1: pay to copy from an item first half

48

Accounting method

Insert pay 3 = O(1)!

Particularly useful when there are multiple operations

49

Another set data structure

We want to support fast lookup and insertion (i.e. faster than linear)

Arrays can easily be made to be fast for one or the other

- fast search: keep list sorted
  - O(n) insert
  - O(log n) search
- fast insert: extensible array
  - O(1) insert (amortized)
  - O(n) search

50

Another set data structure

Idea: store data in a collection of arrays

- array *i* has size 2^i
- an array is either full or empty (never partially full)
- each array is stored in sorted order
- no relationship between arrays

51

Another set data structure

Which arrays are full and empty are based on the number of elements

- specifically, binary representation of the number of elements
- 4 items = 100 = A2-full, A1-empty, A_0-empty
- 11 items = 1011 = A_3-full, A_2-empty, A_1-full, A_0-full

A_0: [5]
A_1: [4, 8]
A_2: empty
A_3: [2, 6, 9, 12, 13, 16, 20, 25]

Lookup: binary search through each array

- Worst case runtime?

52

Another set data structure

A_0: [5]
A_1: [4, 8]
A_2: empty
A_3: [2, 6, 9, 12, 13, 16, 20, 25]

Lookup: binary search through each array

Worst case: all arrays are full

- number of arrays = number of digits = log n
- binary search cost for each array = O(log n)
- O(log n log n)

53

Another set data structure

Insert(A, item)

- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

54

Insert 5

A_0: empty

Insert

- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

55

Insert 5

A_0: [5]

Insert

- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

56

Insert 6

A_0: [5]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

57

Insert 6

A_0: empty
A_1: [5, 6]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

58

Insert 12

A_0: empty
A_1: [5, 6]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

59

Insert 12

A_0: [12]
A_1: [5, 6]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

60

Insert 4

A_0: [12]
A_1: [5, 6]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

61

Insert 4

A_0: empty
A_1: empty
A_2: [4, 5, 6, 12]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

62

Insert 23

A_0: empty
A_1: empty
A_2: [4, 5, 6, 12]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

63

Insert 23

A_0: [23]
A_1: empty
A_2: [4, 5, 6, 12]

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

64

Another set data structure

Insert
- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

running time?

65

Insert running time

Worst case
- merge at each level
- 2 + 4 + 8 + … + n/2 + n = O(n)

There are many insertions that won't fall into this worse case

What is the amortized worse case for insertion?

66

insert: amortized analysis

Consider inserting *n* numbers
- how many times will A_0 be empty?
- how many times will we need to merge with A_0?
- how many times will we need to merge with A_1?
- how many times will we need to merge with A_2?
- …
- how many times will we need to merge with $A_{\log n}$?

67

insert: amortized analysis

Consider inserting *n* numbers

	times
how many times will A_0 be empty?	n/2
how many times will we need to merge with A_0?	n/2
how many times will we need to merge with A_1?	n/4
how many times will we need to merge with A_2?	n/8
…	
how many times will we need to merge with $A_{\log n}$?	1

cost of each of these steps?

68

insert: amortized analysis

- Consider inserting *n* numbers

	times	cost
how many times will A_0 be empty?	n/2	O(1)
how many times will we need to merge with A_0?	n/2	2
how many times will we need to merge with A_1?	n/4	4
how many times will we need to merge with A_2?	n/8	8
…		
how many times will we need to merge with $A_{\log n}$?	1	n

total cost:

69

insert: amortized analysis

- Consider inserting *n* numbers

	times	cost
how many times will A_0 be empty?	n/2	O(1)
how many times will we need to merge with A_0?	n/2	2
how many times will we need to merge with A_1?	n/4	4
how many times will we need to merge with A_2?	n/8	8
…		
how many times will we need to merge with $A_{\log n}$?	1	n

total cost: log n levels * O(n) each level
O(n log n) cost for *n* inserts
O(log n) amortized cost!

70