Amortized Analysis and Heaps Intro

David Kauchak
cs302
Spring 2013

Admin

- Homeworks 4 and 5 back soon
- How are the homeworks going?

Extensible array

Sequential locations in memory in linear order

Elements are accessed via index

- Access of particular indices is O(1)

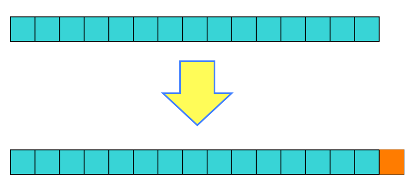
Say we want to implement an array that supports *add* (i.e. *addToBack*)

- ArrayList or Vector in Java
- arrays in Python, perl, Ruby, …

How can we do it?

Extensible array

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

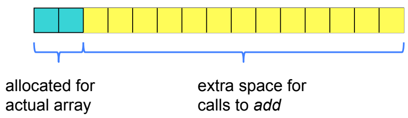


Running time: $\Theta(n)$

Extensible array

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

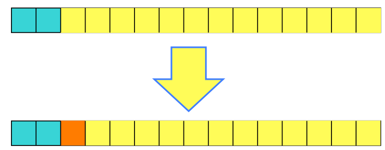
For example: new ArrayList(2)



Extensible array

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

Adding an item:

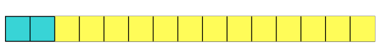


Running time: Θ(1)   Problems?

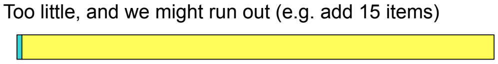
Extensible array

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)

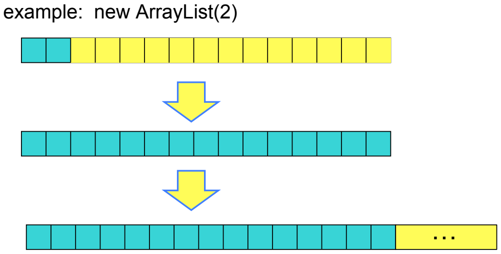


Too much, and we waste lots of memory   Ideas?

Extensible array

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

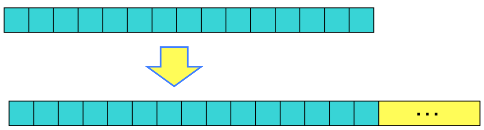
For example: new ArrayList(2)



Extensible array

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

For example: new ArrayList(2)

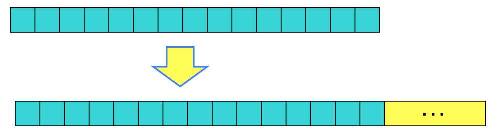


Running time: Θ(n)

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?

Extensible array

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

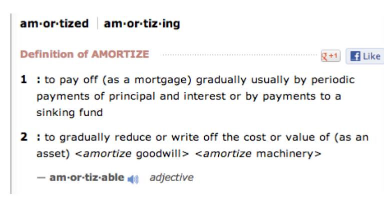
How else might we talk about runtime?

What is the ***average*** running time of *add* in the ***worst case***?

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

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*

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

Amortized analysis

Many approaches for calculating the amortized analysis

- we'll just look at the counting/aggregate method
- book has others

aggregate method

- figure out the big-O runtime for a sequence of *n* calls
- divide by *n* to get the average run-time per call

Amortized analysis

What is the aggregate cost of *n* calls?

Let's assume it's O(1) and then prove it

Base case: size 1 array, add an element: O(1)

Inductive case: assume n-1 calls are O(1), show that *n*th call is O(1)

Two cases:

- array need to be doubled
- array does need to be doubled

Amortized analysis

What is the aggregate cost of *n* calls?

Case 1: doesn't need doubling

- just add the element into the current array
- O(1)

Case 2: need doubling

- O(n) operation to copy all the data over
- Overall cost of n-insertions:
  - *n-1*O(1) + O(n) = O(n)*
- Amortized cost: *O(n)/n* = ***O(1)***

We amortize (spread) the cost of the O(n) operation over all of the previous O(1) operations

Amortized analysis

Another way we could have done the analysis would be to calculate the total cost over *n* operations

Assume we start with an empty array with 1 location. What is the cost to insert n items?

$$\text{total\_cost(n)} = \text{basic\_cost(n)} + \text{double\_cost(n)}$$

$$\text{basic\_cost(n)} = O(n) \qquad \text{double\_cost(n)} \le 1+2+4+8+16+\ldots+n = 2n$$

$$\text{total\_cost(n)} = O(n)$$

amortized *O(1)*

Amortized analysis vs. worse case

What is the worse case of *add*?

- Still O(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

Extensible arrays

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

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

- No!
- Why?

Amortized analysis

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

$$\text{total\_cost(n)} = \text{basic\_cost(n)} + \text{double\_cost(n)}$$

$$\text{basic\_cost(n)} = O(n) \qquad \text{double\_cost(n)} = k+2k+3k+4k+5k+\ldots+n$$

$$= \sum_{i=1}^{n/k} ki$$

$$= k\sum_{i=1}^{n/k} i$$

$$= k\frac{\frac{n}{k}\left(\frac{n}{k}+1\right)}{2} = O(n^2)$$

Amortized analysis

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

$$\text{total\_cost(n)} = O(n) + O(n^2)$$
$$= O(n^2)$$

amortized *O(n)!*

Another set data structure

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

Arrays can easily 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

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

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

- Worse case runtime?

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

Worse 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)

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

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

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

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

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

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

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

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

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

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

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

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?

Insert running time

Worse 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?

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 $A_{\log n}$?

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?

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:

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!

Binary heap

A binary tree where the value of a parent is greater than or equal to the value of its children

Additional restriction: all levels of the tree are **complete** except the last

Max heap vs. min heap

Binary heap - operations

Maximum(S) - return the largest element in the set

ExtractMax(S) – Return and remove the largest element in the set

Insert(S, val) – insert val into the set

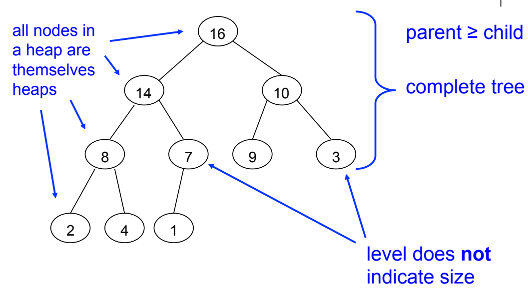
IncreaseElement(S, x, val) – increase the value of element x to val

BuildHeap(A) – build a heap from an array of elements

Binary heap

How can we represent a heap?

Binary heap - references



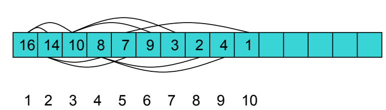
Binary heap - array

PARENT(i)
  return $\lfloor i/2 \rfloor$

LEFT(i)
  return $2i$

RIGHT(i)
  return $2i + 1$

Binary heap - array

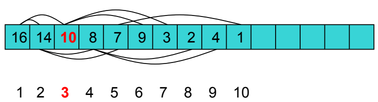


PARENT(i)
  return $\lfloor i/2 \rfloor$

LEFT(i)
  return $2i$

RIGHT(i)
  return $2i + 1$

Binary heap - array



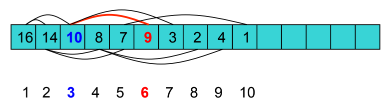
Left child of A[3]?

PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

Binary heap - array



Left child of A[3]?

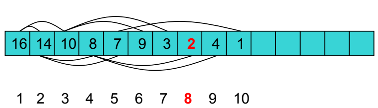
2*3 = 6

PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

Binary heap - array



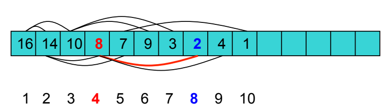
Parent of A[8]?

PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

Binary heap - array



Parent of A[8]?

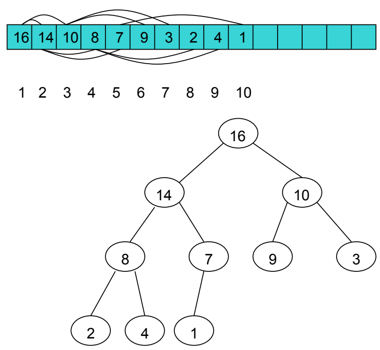
$\lfloor 8/2 \rfloor = 4$

PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

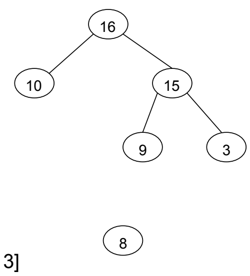
RIGHT(i)
return $2i + 1$

Binary heap - array



Identify the valid heaps

[15, 12, 3, 11, 10, 2, 1, 7, 8]



[20, 18, 10, 17, 16, 15, 9, 14, 13]