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?

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: $\Theta(n)$
Extensible array

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

For example: `new ArrayList(2)`

allocated for actual array  extra space for calls to add

Adding an item:

Running time: $\Theta(1)$

Problems?

---

Extensible array

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

For example: `new ArrayList(2)`

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 more and copy.
For example: `new ArrayList(2)`

Running time: $\Theta(n)$

How much extra memory should we allocate?

---

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.

---

Amortized analysis

What does “amortize” mean?

- To spread costs over time

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), e.g., writing off the cost of machinery.

---
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

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 8 8 8 16 16
  cost:

Count: 1) inserting element and 2) copying elements

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 8 8 8 16 16
  cost:

Count: 1) inserting element and 2) copying elements

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 8 8 8 16 16
  basic cost: 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
What are the costs?

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

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

Amortized analysis

More generally:
\[ \text{total\_cost}(n) = \text{basic\_cost}(n) + \text{double\_cost}(n) \]
\[ \text{basic\_cost}(n) = n \]
\[ \text{double\_cost}(n) = 1 + 2 + 4 + \ldots + n/2 + n = 2n \]
\[ \text{total\_cost}(n) = 3n \]
over n operations:
amortized \( O(1) \)

Amortized analysis vs. worse case

What is the worse case of add?
- Still \( \Theta(n) \)
- 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 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 \( k \)

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

\[
\text{basic\_cost}(n) = n
\]

\[
\text{double\_cost}(n) = \sum_{i=1}^{n} (i + 1) = \frac{n(n + 1)}{2} = \Omega(n^2)
\]

Amortized analysis

Consider the cost of \( n \) insertions for some constant \( k \)

\[
\text{total\_cost}(n) = n + \Omega(n^2)
\]

\[
= \Omega(n^2)
\]

amortized \( \Omega(n) \)

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.

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.

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

size: 1 2 4 4 8 8 8 16 16

cost: 1 2 3 1 5 1 1 9 1

bank:

How much should we pay for each insert?
Try insert: 2

How much is left?
Try insert: 2

Try insert: 3

How much is left?
Try insert: 3

Will this work??
**Accounting method**

Insert pay 3 = $O(1)$!

Particularly useful when there are multiple operations

---

**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

---

**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, A1 empty, A0 empty
- 11 items = 1011 = A3, A2 full, A1 empty, A0 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?
Another set data structure

A₀: [5]
A₁: [4, 8]
A₂: empty
A₃: [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)

Insert(A, item)
- starting at i = 0
- current = [item]
- as long as the level i is full
  - merge current with A using merge procedure
  - store to current
  - A = empty
- i++
- A = current
Insert 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
  - $A = empty$
  - $i++$
- $A = current$

Insert 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
  - $A = empty$
  - $i++$
- $A = current$

Insert 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
  - $A = empty$
  - $i++$
- $A = current$

Insert 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
  - $A = empty$
  - $i++$
- $A = current$
2/8/24

Insert 4

A₀: [12]  
A₁: [5, 6]

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

Insert 4

A₀: empty  
A₁: empty  
A₂: [4, 5, 6, 12]

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

Insert 23

A₀: empty  
A₁: empty  
A₂: [4, 5, 6, 12]

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

Insert 23

A₀: [23]  
A₁: empty  
A₂: [4, 5, 6, 12]

Insert starting at i = 0  
  current = [item]  
  as long as the level i is full  
  merge current with A using merge procedure  
  store to current  
  A = empty  
  i++  
  A = current
Another set data structure

Insert
- starting at \( i = 0 \)
- \( current = \{\text{item}\} \)
- as long as the level \( i \) is full
  - merge current with \( A \) using merge procedure
  - store to current
  - \( A = \text{empty} \)
  - \( \ldots \)
  - \( A = \text{current} \)

running time?

Insert running time

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

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 \( A_1 \)?
- how many times will we need to merge with \( A_2 \)?
- how many times will we need to merge with \( A_3 \)?
- \( \ldots \)
- how many times will we need to merge with \( A_{\log n} \)?

\[ \text{times} \]

Consider inserting \( n \) numbers
- how many times will \( A_0 \) be empty?
  - \( n/2 \)
- how many times will we need to merge with \( A_1 \)?
  - \( n/4 \)
- how many times will we need to merge with \( A_2 \)?
  - \( n/8 \)
- \( \ldots \)
- how many times will we need to merge with \( A_{\log n} \)?
  - \( 1 \)

\[ \text{cost of each of these steps?} \]
**insert: amortized analysis**

- Consider inserting \( n \) numbers
  - how many times will \( A_0 \) be empty? \( \frac{n}{2} \) \( O(1) \)
  - how many times will we need to merge with \( A_0 \)? \( \frac{n}{2} \) \( 2 \)
  - how many times will we need to merge with \( A_1 \)? \( \frac{n}{4} \) \( 4 \)
  - how many times will we need to merge with \( A_2 \)? \( \frac{n}{8} \) \( 8 \)
  - …
  - how many times will we need to merge with \( A_{\log n} \)? \( 1 \) \( n \)

**total cost:**

\[
\log n \text{ levels} \times O(n) \text{ each level} \\
O(n \log n) \text{ cost for } n \text{ inserts} \\
O(\log n) \text{ amortized cost!}
\]