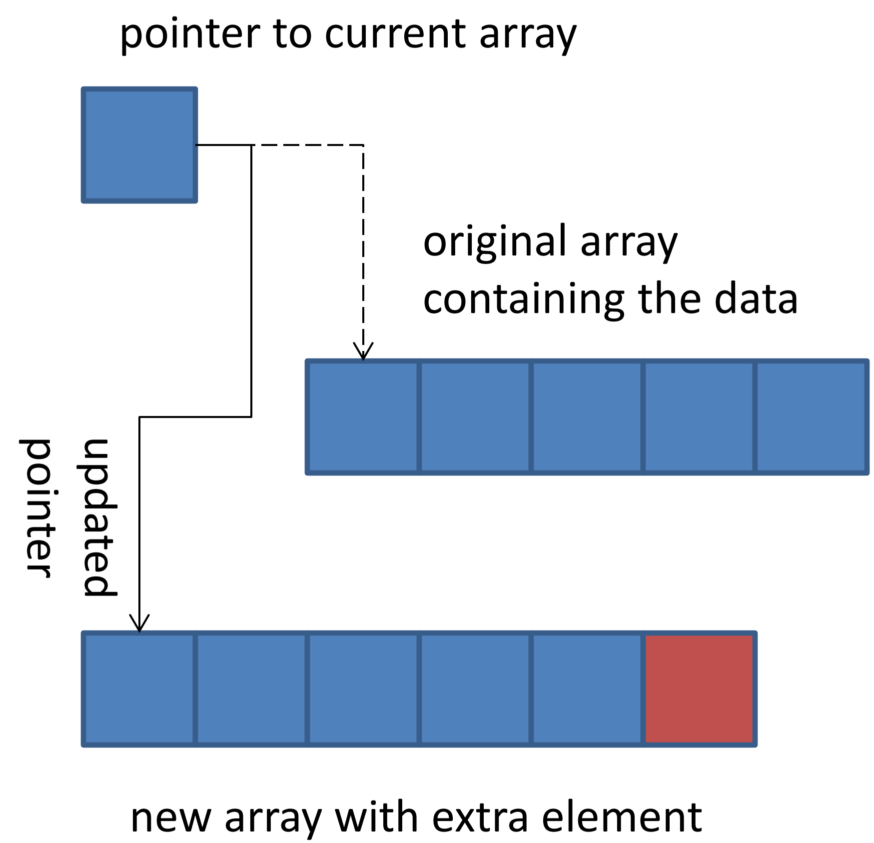
Lecture 16
Dynamic tables
and amortized analysis

EECS-214

Dynamic arrays

- Just an object with a pointer to an array
- The array stores the real data
- When you need to change the size
  - Make a whole new array
  - Copy the data
  - Change the pointer



Dynamic array in C#

```
public class DynamicArray
  {
    object[] realArray = new object[0];

    // Add an element at end
    void Add(object newValue)
    {
      object[] newArray
        = new object[realArray.Length + 1];
      for (int i = 0; i < realArray.Length; i++)
        newArray[i] = realArray[i];
      newArray[realArray.Length] = newValue;
      realArray = newArray;
    }

    … other methods …
}
```

Adding 10 elements

Step	Elements copied	Total copies
a = new DynamicArray()	0	0
a.Add(1)	0	0
a.Add(2)	1	1
a.Add(3)	2	3
a.Add(4)	3	6
a.Add(5)	4	10
a.Add(6)	5	15
a.Add(7)	6	21
a.Add(8)	7	28
a.Add(9)	8	36
a.Add(10)	9	45

$O(n^2)$ copies!

Aggregate analysis

- Cost of a **sequence** of operations
- In this case
  - We grow the array **every time**
  - We grow by **just enough space**
- So
  - Each add requires $\Theta(n)$ element copies
  - **Sequence** requires $\Theta(n^2)$ copies

Adding more than we need

- What if we
  - **Don’t** grow the array every time
    - Start with **extra space**
    - Only grow when we **run out** of extra space
  - **Add extra** when we do grow

- Question: **how much extra** should we add?

HOW ABOUT IF WE GROW BY A LOT LIKE 1000 ELEMENTS?

Growing by 1000 elements at a time

- Then if we add n elements total
  - Instead of growing n times
  - We **grow $\mathbf{0.001}\boldsymbol{n}$ times**
  - Which is **still grow $\mathbf{O}(\boldsymbol{n})$ times**
- And we still do $\mathrm{O}(n)$ work each time we grow
- So the total work is **still $\mathbf{O}\left(\boldsymbol{n}^{\mathbf{2}}\right)$**

WHAT IF WE KEEP INCREASING THE AMOUNT THAT WE GROW BY?

Increasing the growth factor

- If we **increase the growth factor** each time
- Then we need to **grow less frequently** as time goes on
- So it **might cancel out** …

Doubling the array size

```
public class DynamicArray
  {
     object[] realArray = new object[2];
     int Count = 0;

     // Add an element at end
     void Add(object newValue)
     {
        count++;
        if (Count == realArray.Length) {
            object[] newArray = new object[2*realArray.Length];
            for (int i = 0; i < realArray.Length; i++)
                newArray[i] = realArray[i];
            realArray = newArray;
        }
        realArray[count-1] = newValue;
     }
}
```

Adding 10 elements

Step	Elements copied	Total copies
a = new DynamicArray()	0	0
a.Add(1)	0	0
a.Add(2)	0	0
a.Add(3)	2	2
a.Add(4)	0	2
a.Add(5)	4	6
a.Add(6)	0	6
a.Add(7)	0	6
a.Add(8)	0	6
a.Add(9)	8	14
a.Add(10)	0	14

n calls < 2n copies

O(n) copies!

Aggregate analysis (handwavy version)

- Cost of a **sequence** of operations
- In this case
  - We grow with exponentially **decreasing frequency**
  - But by exponentially **increasing amounts**
- So
  - **Most** adds requires **zero copies**
    - Some require $\Theta(n)$ copies
    - But they occur with exponentially decreasing frequency
  - **Sequence** requires $\Theta(n)$ total copies

Aggregate analysis (formal version)

- A sequence of n **add operations** only **recopies** on the 3rd, 5th, 9th, adds, etc.
  - That is, **when we're doing the $2^i + 1$st add** for some integer i
  - Each of those copies **2^i elements**
  - So the **total** number of copies is

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} 2^i = 2^{\lfloor \log_2 n \rfloor + 1} - 2 < 2^{\lfloor \log_2 n \rfloor + 1}$$

$$= 2 \times 2^{\lfloor \log_2 n \rfloor} \leq 2 \times 2^{\log_2 n} = 2n = \boldsymbol{\Theta(n)}$$

Amortized analysis

(using the aggregate method)

- Basic idea
  - If an **arbitrary sequence** of n operations takes time $\mathrm{O}(\mathrm{f}(n))$
- And we **really mean** arbitrary – **any sequence** of **any length** of those operations
  - Then we can **pretend** that each individual operation takes $\mathrm{O}\left(\frac{\mathrm{f}(n)}{n}\right)$ time

- We say it takes $\mathrm{O}\left(\frac{\mathrm{f}(n)}{n}\right)$ **amortized time**

- In this case, insertion takes $\mathrm{O}\left(\frac{n}{n}\right) = \mathrm{O}(1)$ **amortized time**

What’s the difference between?

- Saying an algorithm takes $O(n)$ **average time**
- And saying it takes $O(n)$ **amortized time**?

Average vs. amortized complexity

- **Average-case complexity**
  - Averages over **possible inputs** to a single call
- **Amortized complexity**
  - Averages over **operations in a sequence of calls**

Methods for amortized analysis

- **Aggregate** method (what we just did)
  - Amortized cost = cost of sequence/#operations
  - Only works for sequences of operations with the same cost
- **Accounting** method
  - More general
  - **Guess** the right **amortized cost for each procedure**
    - **Overcharge** some operations to pay for later operations
    - Excess charges get assigned to particular parts of the data structure as "**credit**"
  - Show the sum of actual costs of any series of operations can't exceed the sum of the amortized costs
- **Potential** method
  - Define a "**potential energy**" Φ of a data structure
  - Show that **no sequence of operations** can decrease Φ **below its initial value**
  - Amortized cost of an operation = **real cost + change in Φ**

Using the potential method

- Let A be our dynamic array
- Define $\boldsymbol{\Phi(A) \stackrel{\text{def}}{=} (2 \times A.\text{Count}) - A.\text{realArray.Length}}$
- Amortized **cost of Add** is $\boldsymbol{O(1)}$ + **change in Φ**

Great! What's the change in Φ?

Case 1: **Array isn't full**

- Then we **increase A.Count by 1**
- And **don't change A.realArray.Length**
- Amortized cost = real cost + potential change
  - Real cost =
    - cost of checking if we need to grow $O(1)$
    - + cost of incrementing Count $O(1)$
    - + storing new item $O(1)$
    - = $O(1)$ total
- Amortized cost = $O(1) + \Delta\Phi = O(1) + 2 = O(1)$

Great! What's the change in Φ?

Case 2: **Array is full** (have to **expand** the array)

- Let $\boldsymbol{i}$ be the **number of items** in the array **before insertion**
- Before expansion
  - A.Count=A.realArray.Length=i
  - $\boldsymbol{\Phi = 2i - i = i}$
- After expansion
  - A.realArray.Length=$2i$
  - $\boldsymbol{\Phi = 2i - 2i = 0}$
- $\boldsymbol{\Delta\Phi = -i}$ (we loose potential)
- Cost = $O(1) + i \text{ copies} + \Delta\Phi = O(1) + i - i = O(1)$
- So **amortized cost is $\boldsymbol{O(1)}$ either way**

Shrinking the array

- What if we want to support both Add and Remove (from the end)
  - And we want it to shrink the table automatically if it's too big

- Analysis is hairier
  - See section 18.4 in book, if you're curious

- Take home message:
  - **Double** the array size **when full**
  - **Halve** it when the array is only **¼ full**

Applying to other kinds of tables

- Which of these data structures can you use this technique on?
  - Stacks
  - Queues
  - Hash tables
  - Binary heaps

- **All of them**!
- All can support growing the tables in $\mathbf{O(1)}$ **amortized time**.

Amortized hash table

- Same idea
  - **Double size** when load factor α exceeds some threshold $\alpha_{\max}$
  - **Halve size** when $\alpha \leq \frac{\alpha_{\max}}{4}$

- Now α is in a **fixed range**: $\frac{\alpha_{\max}}{4} \leq \alpha \leq \alpha_{\max}$

- So for a chained hash table, we get:

  Amortized cost = real cost + $\Delta\Phi$

$$= \Theta(\alpha_{\max}) + \text{cost of copying} + \Delta\Phi$$
$$= \Theta(\alpha_{\max})$$

FINALLY! REAL CONSTANT-TIME PERFORMANCE FOR HASH TABLES!

Well, almost …

Let's unpack what we just showed

- The **average-case**, **amortized** time
  - Averaging over all **operations in a sequence**
  - But also all **possible keys** and hash table **contents**
- For insertion or lookup
- In a **chained** hash table with dynamic expansion
  - You can show it for open-coding too
- Is $\Theta(1)$

True $\Theta(1)$ performance (average case)

- It's only useful in niche applications
- But you can actually make hash tables with **true $\Theta(1)$ average case** performance
  - Not amortized
  - I.e. they'll never pause to copy a lot of data

- How do you do that?

Basic idea

- We got the amortized time bound by **charging insertions for the copies that happened later**
- What we want to do is to actually do the copying **during the insertion**
- How do we do that?

Make two hash tables!

- The hash table really has two hash tables inside it
  - The **current** table
  - And the **future** one that's twice as big
- Each time you **add a new item**
  - **Put it in the current table**
    - $\Theta(1)$ time
  - **Copy two items** in the current table to the future table
    - Also $\Theta(1)$ time, so $\Theta(1)$ time total

- By the time the **current one fills**, all its data has been copied to the future table
  - So **make the future table the current table**
  - And **make a new future table** that's twice as big

Again,

YOU CAN USE THE SAME TRICK FOR ARRAYS, STACKS, ETC.

What do you need to know about this for the quiz?

- Nothing for Quiz 2
- For Quiz 3
  - Amortized analysis lets you **average out** the costs of operations in a **sequence**
    - Think of operations having uniform costs rather than variable
    - Done by "**overcharging**" early operations to pay for later ones
  - Tables and sequence data structures can be made **self-expanding** in $\boldsymbol{\Theta(1)}$ **amortized time**
    - By **doubling** size on overflow