Amortized analysis

10/25/24

Administrivia

- Exam due Sunday night
- Come and meet alumni 4pm TODAY in SMC E117

Recall: Adding to an ArrayList

Doubling the array

```
void add(T value) {
    if(num == vals.length) {
        T[] temp = (T[]) new Object[num*2];
        for(int i=0; i<num; i++)
            temp[i] = vals[i];
        vals = temp;
    }
    vals[num] = value;
    num++;
}
```

Growing the array by 1

```
void add(T value) {
    if(num == vals.length) {
        T[] temp = (T[]) new Object[num+1];
        for(int i=0; i<num; i++)
            temp[i] = vals[i];
        vals = temp;
    }
    vals[num] = value;
    num++;
}
```

Recall: Adding to an ArrayList

Doubling the array

```
void add(T value) {
    if(num == vals.length) {
        T[] temp = (T[]) new Object[num*2];
        for(int i=0; i<num; i++)
            temp[i] = vals[i];
        vals = temp;
    }
    vals[num] = value;
    num++;
}
```

Growing the array by 1

```
void add(T value) {
    if(num == vals.length) {
        T[] temp = (T[]) new Object[num+1];
        for(int i=0; i<num; i++)
            temp[i] = vals[i];
        vals = temp;
    }
    vals[num] = value;
    num++;
}
```

What are the worst-case running times of these methods?

A. O(1) and O(1) B. O(1) and O(n) C. O(n) and O(1)

D. O(n) and O(n) E. None of the above

Recall: Adding to an ArrayList

Doubling the array

```
void add(T value) {
    if(num == vals.length) {
        T[] temp = (T[]) new Object[num*2];
        for(int i=0; i<num; i++)
            temp[i] = vals[i];
        vals = temp;
    }
    vals[num] = value;
    num++;
}
```

Growing the array by 1

```
void add(T value) {
    if(num == vals.length) {
        T[] temp = (T[]) new Object[num+1];
        for(int i=0; i<num; i++)
            temp[i] = vals[i];
        vals = temp;
    }
    vals[num] = value;
    num++;
}
```

What are the worst-case running times of these methods?

A. O(1) and O(1) B. O(1) and O(n) C. O(n) and O(1)

D. **O(n) and O(n)** E. None of the above

Both have O(n) worst case time per operation, but…

- Times (in sec) for adding chars one at a time to a list of chars:

Final length	doubling	incrementing
10,000	0.001	0.015
50,000	0.007	0.721
100,000	0.012	2.743
500,000	0.054	64.746

Motivation for amortized analysis

- Suppose we have a program with the following properties:
  - Does n operations
  - Operations take different amounts of time, but always $\leq$ t

Motivation for amortized analysis

- Suppose we have a program with the following properties:
  - Does n operations
  - Operations take different amounts of time, but always $\leq$ t
- What is the running time?
  - Worst case: O(nt)

Motivation for amortized analysis

- Suppose we have a program with the following properties:
  - Does n operations
  - Operations take different amounts of time, but always $\leq$ t

- What is the running time?
  - Worst case: O(nt)
  - Average case: What does this mean? Average over what?

Motivation for amortized analysis

- Suppose we have a program with the following properties:
  - Does n operations
  - Operations take different amounts of time, but always $\leq$ t

- What is the running time?
  - Worst case: O(nt)
  - Average case: What does this mean? Average over what?
  - Amortized analysis: Looks at worst case over a sequence of operations
    - If n operations take T(n) time, then amortized time is T(n) / n
      (e.g. If T(n) is O(n), then constant amortized time per operation)

Recall: BFS

```
while q is non-empty
        u = q.dequeue()
        for each neighbor v of u
                if v is unmarked
                        mark v
                        q.enqueue(v)
```

O(V) worst case, but O(E) total for all operations

Accounting version

(yes, this example is depreciation, but it's the same idea)

- Suppose you want to run a laundromat

	Profit
• Year 1: Buy machines ($1000), Earn operating profit ($300)	-$700
• Year 2: Earn operating profit ($300)	$300
• Year 3: Earn operating profit ($300)	$300
• Year 4: Earn operating profit ($300)	$300
• Year 5: Earn operating profit ($300)	$300
• Year 6: Replace machines ($1000), Earn operating profit ($300)	-$700
• Year 7: Earn operating profit ($300)	$300
• Year 8: Earn operating profit ($300)	$300
• …	

Approach 1: Aggregate method

- Compute total time for n operations and then divide by n

Approach 2: Accounting method

- Run a “data structure store”
- Charge amortized costs to the customers
- Pay workers in the back the actual cost
- When amortized cost > actual cost, save the extra in the data structure; use this to pay for times when amortized cost < actual cost

Approach 2: Accounting method

- Run a “data structure store”
- Charge amortized costs to the customers
- Pay workers in the back the actual cost
- When amortized cost > actual cost, save the extra in the data structure; use this to pay for times when amortized cost < actual cost

NOTE: Money stored “in the data structure” does not appear in the implementation; this is just for the analysis

Approach 3: Potential method

- Accounting method except you keep a bank account for all the money
- Name comes from thinking of this as potential energy (from physics)

$\Phi(D)$ is the current balance/potential for data structure D

$\Phi(D_0) = 0$ for initial data structure D_0 (no trust funds)

$\Phi(D) \geq 0$ always (no going into debt)

amortized cost = actual cost + $\Delta\Phi$ = actual cost + $\Phi_{after} - \Phi_{before}$