Lecture 12: March 22, 2001

Today's Lecture:

- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method
- Binary Counter
- Table Update

Amortized Analysis

Scenario: Maintaining data structure over a sequence of n operation.

Operation cost: Cost per operation may be large (e.g. $\Theta(n)$).

Total cost: But total cost may not be as much as $n\times$ (worst-case cost of one operation).

Amortized analysis: How to do tight analysis in such scenarios?

Examples: (1) HEAPIFY vs. BUILD-HEAP.
(2) PARTITION.

Three techniques

Today we will describe three techniques that can be used to analyze such scenarios.

- **Aggregate method**
- **Accounting method**
- **Potential method**

- Motivate by the "Binary Counter" example.
- Explain the differences.
- Move to "Dynamic Table" problem.

Binary Counter

Data Structure: k-bit array, counting numbers from 0 to $2^k - 1$.

Operation: Increment.

Cost: # of bits flipped.

(Toy problem! Motivates the ideas clearly!)

Example:

00000	
00001	Cost $= 1$
00010	Cost $= 2$
00011	Cost $= 1$
00100	Cost $= 3$

Notice: Costs vary! Max cost $= \Theta(\log n)$.

Aggregate method

Idea: Count the total cost for n operations directly.

Technique: Ad-hoc. Possibly try to count from a different way.

Example

$$
\begin{array}{lll}
0\ 0\ 0\ 0\ 0 & & \\
0\ 0\ 0\ 0\ 1 & & \text{Cost} = 1 \\
0\ 0\ 0\ 1\ 0 & & \text{Cost} = 2 \\
0\ 0\ 0\ 1\ 1 & & \text{Cost} = 1 \\
0\ 0\ 0\ 0\ 0 & & \text{Cost} = 3 \\
0\ 0\ 0\ 0\ 1 & & \text{Cost} = 1 \\
\quad\ \vdots & & \\
1\ 1\ 1\ 1\ 1 & & \text{Cost} = 5 \\
\uparrow\ \uparrow\ \uparrow\ \uparrow\ \uparrow & & \\
1\ 2\ 4\ 8\ 16 & \leftarrow & \text{Cost}
\end{array}
$$

Aggregate method (contd).

- Count per row varies. Hard to sum up.
- Instead count by columns.
- I.e., count how many times ith bit is flipped.
- Add the costs.

For a sequence of n increments
ith l.s.b. flipped $\lfloor n/2^i \rfloor$ times.

$$
\begin{aligned}
\text{Total Cost } &= \sum_{i=0}^{\lfloor \log_2 n \rfloor} \lfloor n/2^i \rfloor \\
&\leq \sum_{i=0}^{\lfloor \log_2 n \rfloor} n/2^i \\
&\leq n \sum_{i=0}^{\infty} 1/2^i \\
&= 2n
\end{aligned}
$$

Thus total cost $= O(n)$.
Amortized cost per insertion $= O(n)/n = O(1)$.

Amortized cost

Definition: Average cost of an operation over a sequence of n operations, maximized over all n and all sequences.

Warning:

Not average case analysis!

No assumptions about input sequence.

Amortization is still a worst-case principle!

In aggregate method, counted the total to bound the amortized cost. Usually do this the other way.

Accounting Method

- Data structure comes with a “bank account”.
- Every operation allotted a fixed \$ cost (its amortized cost).
- If actual cost less than allotted amount, deposit extra \$'s into bank.
- If actual cost more than allotted amount, withdraw from bank to pay for the operation.
- Prove: Always have a non-negative balance.
- Conclude: Sequence of n operations costs at most n times the amortized cost!

Example: Binary Counter

- Amortized Cost of flipping $0 \to 1 = 2\$$.
- Amortized Cost of flipping $1 \to 0 = 0\$$.

- When flipping 0 to 1: Pay 1$ for operation, and put 1$ in bank account (earmarked to the newly set bit).
- When flipping 1 to 0: Withdraw from the bank to pay for the operation.
- Invariant: Every 1 in the counter has 1$ earmarked for it in the bank account. So always have money to pay for the operation.
- Finally: Increment makes only one $0 \to 1$ flip. So amortized cost of increment = 2.

Potential Method

- In this method, we associate a potential energy with every data structure.
- Potential energy is "Potential to do damage".
- Amortized cost = actual cost + new potential - old potential.
- I.e., Must pay to increase the potential of the data structure.
- If operation has large actual cost but reduces the potential a lot, then amortized cost is low.
- How to find potential: Look for what makes a data structure bad.

Potential functions

- Basic rules:
- Must always be non-negative.
- Must start at zero.
- Implies a sequence of n operations cost at most n times the amortized cost.

Proof: Suppose n operations modify data structure from D_0 to D_1 to $\ldots D_n$.
Let $\Phi(D)$ be the potential of data structure D.
Let c_i be actual cost of ith operation.
Let $\hat{c}_i$ be amortized cost of ith operation.

Then $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.
Summing we get.

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) \geq \sum_{i=1}^{n} c_i\end{aligned}$$

Example: Binary Counter

Data structure is bad if it has a lot of 1's in it.
Let $\Phi(\text{Counter}) = \#$ 1's in Counter.
Potential increase on increment
= # $(0 \to 1)$ flips - # $(1 \to 0)$ flips
= 1 - # $(1 \to 0)$ flips

Thus amortized cost of increment
= Actual cost + Potential increase
= (1 + # $(1 \to 0)$ flips + (1 - # $(1 \to 0)$ flips
= 2

Thus amortized cost = 2, and cost of n increments is at most $2n$.

Example: Dynamic tables. *(cont'd)*

- Idea: "Grow" table with system call to allocate more memory. Reinsert old items.

Sequence of n Inserts:

$$\begin{aligned} \text{Worst-case cost} &= \Theta(n) \\ \text{Total cost} &\ll n \cdot \Theta(n) = \Theta(n^2) \\ &= \Theta(n) \end{aligned}$$

Let c_i = cost of i^{th} Insert

$$= \begin{cases} i & \text{if } i-1 = 2^l \\ 1 & \text{if } otherwise \end{cases}$$

i	*Size*	*Cost*
1	1	1
2	2	$1+1$
3	4	$1+2$
4	4	1
5	8	$1+4$
6	8	1
7	8	1
8	8	1
9	16	$1+8$
10	16	1

Aggregate Analysis:

$$\begin{aligned} \text{Cost of } n \text{ inserts} &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=1}^{\lg n} 2^j \\ &\leq 3 \cdot n \end{aligned}$$

Accounting analysis

- Charge each operation a fictitious amortized amount.
- Amount not immediately used is stored in bank.
- Later operations use bank reserve.
- Balance must not go negative.

Must have

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \forall n$$

Dynamic table:

Charge $\underbrace{\$3}_{\text{amortized cost } \hat{c}_i}$ for ith Insert.

\$1 pays for immediate Insert \$2 stored for later.

When table doubles:

\$1 reinserts item \$1 reinserts old item

0 0 0 0	2 2 2 2

0 0 0 0 0 0 0 0	2 2 2

Therefore, balance is never negative.

Amortized costs are upper bound on true costs.

Advantage of accounting over aggregate:

Each operation can be assigned a specific amortized cost.

Note: Different amortized costs may work.

Expansion and Contraction

Inserts and Deletes

Table overflows $\Rightarrow$ double

Table $< \frac{1}{2}$ full $\rightarrow$ halve (causes thrashing)

$\rightarrow \frac{1}{4}$

Accounting analysis:

$$\hat{c}_i = \begin{cases} 3 & \text{if } Insert \\ 2 & \text{if } Delete \end{cases}$$

Always can pay $\Rightarrow$ n operations cost $O(n)$

Potential analysis:

Stored work viewed as potential energy of data structure.

Most flexible and powerful.

Framework: Start with data structure D_0
operation i transforms D_{i-1} to D_i
cost of operation i is c_i

Idea: Define potential function $\Phi : \{D_i\} \to \Re$ such that $\Phi(D_0) = 0$ and $\Phi(D_1) \geq 0 \;\; \forall i$.

Amortize cost $\hat{c}_i$ defined by:

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{potential\ difference\ \Delta\Phi}$$

- If $\Delta\Phi > 0$, then $\hat{c}_i > c_i$, and operation i stores work in data structure.
- If $\Delta\Phi < 0$, then $\hat{c}_i < c_i$, and operation i delivers up work from data structure to help pay for c_i.

Total amortized cost of n operations is

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \underbrace{\Phi(D_n)}_{\geq 0} - \underbrace{\Phi(D_0))}_{0} \\ &\geq \sum_{i=1}^{n} c_i\end{aligned}$$

So amortized costs upper bound true costs.

Key: Find useful potential function.

Example: Table doubling (Inserts only)

Defn: $\Phi(D_i) = 2i - 2^{\lceil \lg i \rceil}$ (Assume $2^{\lceil \lg 0 \rceil} = 0$)

Note: $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0 \;\; \forall \;\; i$

ANOTHER TABLE of 0's and 2's.
$2(i - 2_{\lceil \lg i \rceil - 1})$
$2(6 - 2_{3-1}) = 4$

Amortized cost of ith Insert

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

$$= \begin{cases} \left.\begin{array}{l} i + (2i - 2^{\lceil \lg i \rceil}) \\ \quad -(2(i-1) - 2^{\lceil \lg(i-1) \rceil}) \end{array}\right\} & \text{if } (i-1) = 2^l \\ \left.\begin{array}{l} 1 + (2i - 2^{\lceil \lg i \rceil}) \\ \quad -(2(i-1) - 2^{\lceil \lg(i-1) \rceil}) \end{array}\right\} & \text{if } otherwise \end{cases}$$

- Case 1: $2^{\lceil \lg i \rceil} = 2 \cdot 2^{\lceil \lg(i-1) \rceil} = 2(i-1)$
  $\Rightarrow \hat{c}_i = i + 2 - 2(i-1) + (i-1)$
  $= 3$
- Case 2: $2^{\lceil \lg i \rceil} = 2^{\lceil \lg(i-1) \rceil}$
  $\Rightarrow \hat{c}_i = 1 + 2i - 2(i-1)$
  $= 3$

n Inserts cost $\Theta(n)$ in worst case.

More complicated Φ for Deletes (see book).