# Chapter 15: Amortized Analysis

# Chapter Outline

- The Basic Idea
- Three techniques
  - Aggregate analysis
  - Accounting method
  - Potential method
- Illustrating the techniques using 3 examples
  - stack with multipop operation
  - binary counter
  - dynamic table

# Amortized Analysis

- *Amortized analysis* is a cost analysis technique.
- It computes the average time required to perform a sequence of $n$ operations on a data structure.
- *Goal:* Show that although some individual operations may be expensive, on average the cost per operation is small.
- Often *worst case* analysis is *not tight* and the amortized cost of an operation is less than its worst case.
- Average in this context is not based on averaging over a distribution of inputs.
  - No probability is involved.
- It is about *average cost in the worst case* for a sequence of $n$ operations.

# Methods

- ***Aggregate analysis*** **– the total amount of time needed for the** $n$ **operations is computed and divided by** $n$**.**
- ***Accounting*** **– operations are assigned an amortized cost. Items of the data structure are assigned a credit.**
- ***Potential*** **– the prepaid work (money in the "bank") is represented as "potential" energy that can be released to pay for future operations.**

# Aggregate Analysis

***Basic idea*:**

- **If *n* operations together take *T*(*n*) time, then the amortized cost of an operation on average is *T*(*n*)/*n*.**

# A Stack Example

- **A stack $S$ with the following three operations:**
  - ***push*($S$, $x$): $O(1)$ each ➔ $O(n)$ for any sequence of $n$ operations.**
  - ***pop*($S$) : $O(1)$ each ➔ $O(n)$ for any sequence of $n$ operations.**
  - ***multipop*($S$, $k$) : Pop the stack $k$ times.**

    **while not *empty*($S$) and $k > 0$**

    ***Pop*($S$)**

    **$k = k - 1$**
- **Running time of *multipop*($S$, $k$):**
  - **Linear in # of pop operations with each pop costs $O(1)$.**
  - **# of iterations of *while loop* is $\min\{n, k\}$, where $n$ = # of objects on stack.**
  - **Therefore, total cost = $\min\{n, k\}$.**

# Stack: Regular Cost Analysis

- **Consider a sequence of $n$ *push*($S$, $x$), *pop*($S$) and *multipop*($S$, $k$) operations on a stack having as many as $n$ items.**
- **The following is what a regular worst-case cost analysis would do:**
  - **Worst-case cost of *multipop*( ) is $O(n)$.**
  - **Have $n$ operations.**

  **➔ The worst-case cost of the sequence is $O(n^2)$.**
- ***Question*: Notice anything problematic with the analysis?**
- ***Answer*: It's impossible to pop $n$ items $n$ times for a stack with $n$ items!**

# Stack – Aggregate Analysis

- **Each item can be popped only once for each time it is pushed.**
- **So the total number of times *pop*( ) can be called, either directly or from *multipop*, is bounded by the number of pushes.**
- **Assume that the stack is initially empty. Then the number of pushes in a sequence of $n$ operations is $\leq n$.**
- **Thus, the number of all pops (including those from multipop) is $O(n)$.**
- **So the total cost of the sequence of $n$ operations is $O(n)$.**

**➔ $O(1)$ per operation on average.**

# A Binary Counter Example

- A $k$-bit binary counter $A[0 \mathinner{..} k-1]$ of bits, where $A[0]$ is the least significant bit and $A[k-1]$ is the most significant bit.
- Counts upward from 0.
- Value of the counter is $\sum_{i=0}^{k-1} A[i] \cdot 2^i$
- Initially, counter value is 0, so $A[0 \mathinner{..} k-1] = 0$.
- To increment, add 1:

> - Flip all 1's from right to 0 until encountering the first 0.
> - Change this 0 to 1 and stop.

INCREMENT$(A, k)$

$i = 0$
**while** $i < k$ and $A[i] == 1$
  $A[i] = 0$
  $i = i + 1$
**if** $i < k$
  $A[i] = 1$

# Binary Counter: An Example

| Counter value | A[7] | A[6] | A[5] | A[4] | A[3] | A[2] | A[1] | A[0] | Total cost |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 3 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 4 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 7 |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 8 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 10 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 11 |
| 8 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 15 |
| 9 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 16 |
| 10 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 18 |
| 11 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 19 |
| 12 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 22 |
| 13 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 23 |
| 14 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 25 |
| 15 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 26 |
| 16 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 31 |

- **It shows a 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 Increment operations.**
- **The average cost per operation is 31/16 < 2.**

# Binary Counter: Regular Analysis

- With a $k$-bit binary counter, a single execution of Increment may need to flip $\Theta(k)$ bits in the worst case.
- So the total cost for executing a sequence of $n$ Increment operations is $O(nk)$ in the worst case.
  - The average per operation cost is $O(k)$.
- This bound is correct but not tight.
- We can obtain a better bound of $O(n)$ using ***aggregate analysis***.

# Binary Counter: Aggregate Analysis

- **Some observations about Increment( ):**
  - **Not all bits are flipped for each call.**
  - ***A*[0] flips each time, *A*[1] flips only every other time, and *A*[2] flips only every 4th time.**
  - **In general, *A*[*i*] flips only every $2^i$-th time.**
- **Thus, *A*[*i*] flips only $\lfloor n/2^i \rfloor$ times in a sequence of *n* Increment operations on an initially 0 counter.**
- **So the total number of flips in the sequence is:**

$$T(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

➔ $T(n) = O(n)$

➔ **The amortized cost per operation is $O(n)/n = O(1)$.**

# Accounting Method: Basic Idea

- Assign different charges to different operations.
  - Some are charged more than actual cost.
  - Some are charged less than actual cost.
- *Amortized cost = amount we charge.*
- Need to be careful with choosing the right amount to charge to each operation (see later).
- When amortized cost > actual cost, store the difference on *specific items* in the data structure as *credit*.
- Use credit later to pay for operations whose actual cost > amortized cost.

# Accounting Method: Credit

- **Need credit to never go negative.**
  - **Otherwise, have a sequence of operations for which the amortized cost is not an upper bound on actual cost.**
  - **Amortized cost would tell us nothing.**
- **Let $c_i$ = actual cost of $i$-th operation,**

  **$\hat{c}_i$ = amortized cost of $i$-th operation.**
- **For all sequences of $n$ operations, require:**

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

- **Total credit stored =** $\displaystyle\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$

# Accounting Method: Stack Example

| Operation | Actual Cost | Amortized Cost |
|---|---|---|
| push | 1 | 2 |
| pop | 1 | 0 |
| multipop | min{$n$, $k$} | 0 |

- **Intuition: When pushing an item, pay $2.**
  - **$1 pays for the *push*.**
  - **$1 is prepayment for it being popped by either *pop* or *multipop*.**
  - **Since each item on the stack has $1 credit, the credit can never go negative.**
  - **The total amortized cost in the worst case is: $2n \in O(n)$**
    - **It is an upper bound on total actual cost.**

# Accounting Method: Binary Counter Example

- **Charge \$2 to set a bit to 1.**
  - **\$1 pays for setting a bit to 1.**
  - **\$1 is prepayment for flipping it back to 0.**
  - **Have \$1 of credit for every 1 in the counter.**
  - **Therefore, credit $\geq 0$.**
- **Amortized cost of Increment:**
  - **Cost of resetting bits to 0 is paid by credit.**
  - **At most 1 bit is set to 1 in each increment operation.**
  - **Therefore, amortized cost $\leq$ \$2.**
  - **For $n$ operations, the total amortized cost in the worst case is $2n \in O(n)$.**

# Potential Method: Basic Idea

- Like the accounting method, but think of the credit as *potential* stored with the entire data structure.
  - *Accounting method* stores credit with specific items.
  - *Potential method* stores potential in the data structure as a whole.
  - Can release potential to pay for future operations.
  - It is the most flexible among the amortized analysis methods.

# Potential Method: Credit

- **Let $D_0$ = initial data structure**

  **$D_i$ = data structure after *i*-th operation**

  **$c_i$ = actual cost of *i*-th operation**

  **$\hat{c}_i$ = amortized cost of *i*-th operation**
- ***Potential function* $\Phi$ maps each data structure to a real number, i.e., the *potential* of the data structure.**
  - **$\Phi(D_i)$ is the *potential* associated with data structure $D_i$ .**
- **Define $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + \Delta\Phi(D_i)$ .**
- **The total amortized cost for a sequence of *n* operations is**

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

- **In practice, $\Phi(D_0) = 0$, $\Phi(D_i) \geq 0$ for all *i* ➔ the amortized cost is always an upper bound on actual cost.**

# Potential Method: Stack Example

- **Define potential function $\Phi$ on a stack = number of items on the stack.**
- $D_0$ **= empty ➔** $\Phi(D_0) = 0$
- **Since the number of items on a stack is always ≥ 0,** $\Phi(D_i) \geq \Phi(D_0) = 0$

| operation | actual cost | $\Delta\Phi$ | amortized cost |
|---|---|---|---|
| PUSH | 1 | $(s+1) - s = 1$<br>where $s = \#$ of objects initially | $1 + 1 = 2$ |
| POP | 1 | $(s-1) - s = -1$ | $1 - 1 = 0$ |
| MULTIPOP | $k' = \min(k, s)$ | $(s - k') - s = -k'$ | $k' - k' = 0$ |

**➔ The total amortized cost of a sequence of** $n$ **operations in the worst case is** $2n = O(n)$**.**

# Potential Method: Binary Counter (1)

- Define potential function $\Phi = b_i$ = number of 1's in the counter *after* the $i$-th Increment.
- Suppose the $i$-th operation resets $t_i$ bits to 0.
- Then the actual cost $c_i \leq t_i + 1$: reset $t_i$ bits plus set at most one bit to 1.
- If $b_i = 0$, the $i$-th operation resets all $k$ bits to 0 but no bit is set to 1, so $b_{i-1} = t_i = k \Rightarrow b_i = b_{i-1} - t_i = 0$.
  - This happens only when all $k$ bits are 1 before $i$-th operation.
- If $b_i > 0$, the $i$-th operation resets $t_i$ bits to 0 and sets one bit to 1, so $b_i = b_{i-1} - t_i + 1$.
- Either way, $b_i \leq b_{i-1} - t_i + 1$.

# Potential Method: Binary Counter (2)

- **Since $b_i \le b_{i\text{-}1} - t_i + 1$,**
  **$\Delta(D_i) = \Phi(D_i) - \Phi(D_{i\text{-}1}) = b_i - b_{i\text{-}1} \le (b_{i\text{-}1} - t_i + 1) - b_{i\text{-}1} = 1 - t_i$**
- **Thus, $\hat{c}_i = c_i + \Delta(D_i) \le (t_i + 1) + (1 - t_i) = 2$**
- **If counter starts at 0, $\Phi(D_0) = 0$.**

**➔ amortized cost of a sequence of *n* operations =**

$$\sum_{i=1}^{n} \hat{c}_i \le \sum_{i=1}^{n} 2 = 2n = O(n)$$

# Dynamic Table

- **A table is a *dynamic table* if its content can change and we can't predict its maximum size.**
- **Examples: object tables and hash tables.**
- **We consider *in-memory tables* here.**
- **When the table fills up and needs more space (*table overflow*), create a new table with a larger space, copying all contents into the new table.**
- ***Question*: Why create the new table?**
- ***Answer*: In-memory tables are usually implemented using arrays, which need contiguous space.**
- **When it gets sufficiently small, *might* want to reallocate with a smaller size.**

# Dynamic Table: Table Expansion

- **When a new insert causes a table overflow, create a new table with double the size of the old table.**

```
TABLE-INSERT(T, x)
  if T.size == 0
      allocate T.table with 1 slot
      T.size = 1
  if T.num == T.size                               // expand?
      allocate new-table with 2 · T.size slots
      insert all items in T.table into new-table   // T.num elem insertions
      free T.table
      T.table = new-table
      T.size = 2 · T.size
  insert x into T.table                            // 1 elem insertion
  T.num = T.num + 1
```

# Dynamic Table: Cost Analysis (1)

- ***Question***: **What is the worst-case cost of an insert?**
- **There are two types of inserts:**
  - ***Type 1***: **It simply inserts a single object into an existing table.**
  - ***Type 2***: **It causes the creation of a new table, copying of the old table contents to the new table, and removing the old table.**
    - **We assume that allocating memory for a new table and freeing the space for an old table takes constant time.**
- **Clearly *Type 2* is the worst-case scenario and the cost can be very high due to the copying of the old table.**
  - **We assume the cost is the number of objects to be inserted.**
- **But how about the total cost for a sequence of *n* inserts?**

# Dynamic Table: Cost Analysis (2)

- **Let $c_i$ = cost of the *i*-th insert. We have**

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

- **Example:**

| Operation | Table Size | Cost |
|---|---|---|
| `Insert(1)` | 1 | 1 |
| `Insert(2)` | 2 | 1 + 1 |
| `Insert(3)` | 4 | 1 + 2 |
| `Insert(4)` | 4 | 1 |
| `Insert(5)` | 8 | 1 + 4 |
| `Insert(6)` | 8 | 1 |
| `Insert(7)` | 8 | 1 |
| `Insert(8)` | 8 | 1 |
| `Insert(9)` | 16 | 1 + 8 |

# Dynamic Table: Aggregate Analysis

- **In general, the total cost of a sequence of *n* insert operations is**

$$T(n) = \sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j = n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1}$$

$$\le n + (2n - 1) < 3n$$

- **Per operation average cost of operation is $T(n) / n = O(1)$.**

**➔ a dynamic table has the same asymptotic cost as a fixed-size table**

  - **Both $O(1)$ per insert operation.**

# Dynamic Table: Accounting Method

- For each new object $x$ inserted, charge \$3 amortized cost.
  - \$1 for inserting $x$ into the current table $T_1$ of starting size $m$.
  - \$1 for moving $x$ to new table $T_2$ of size $2m$ after expanding $T_1$.
  - \$1 for moving another object that has been moved once to $T_2$.
    - Suppose there was no credit left after $T_1$ was created.
    - $T_1$ will expand again after another $m$ insertions.
    - Each insertion (allocate \$3, use \$1 for itself) will put \$1 credit on each of the $m$ items that were in $T_1$ when $T_1$ was created and will put \$1 credit on each new object inserted.
    - Will have \$$2m$ of credit by the time $T_1$ expands to $T_2$, when there will be $2m$ objects to move.

➔ Dynamic table has constant (amortized) cost per operation.