Overview

- Analyze a sequence of operations on a data structure.
- We will talk about average cost in the worst case (i.e. not averaging over a distribution of inputs.)
- **Goal:** Show that although some individual operations may be expensive, on average the cost per operation is small.
- 3 methods:
  1. aggregate analysis
  2. accounting method
  3. potential method

Aggregate analysis

Stack 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):**
  ```
  while S≠Ø and k>0 do
    POP(S) k← k−1
  
  Running time of MULTIPOP?
  ```

Running time of MULTIPOP

- Linear in # of POP operations.
- Let each PUSH/POP cost 1.
- # of iterations of while loop is min(s, k), where s = # of objects on stack. Therefore, total cost = min(s, k).

Sequence of n PUSH, POP, MULTIPOP operations:
- Worst-case cost of MULTIPOP is \(O(n)\).
- Have n operations.
- Therefore, worst-case cost of sequence is \(O(n^2)\).

But:
- Each object can be popped only once per time that it is pushed.
- Have \(≤ n\) PUSHes ⇒ \(≤ n\) POps, including those in MULTIPOP. Therefore, total cost = \(O(n)\).
- Average over the n operations ⇒ \(O(1)\) per operation on average.

Binary counter

- \(k\)-bit binary counter \(A[0...k−1]\) 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 counter is: \(\sum \{A[i]⋅2^i\}\)
- Initially, counter value is 0, so \(A[0...k−1]=0\).
- To increment, add 1 (mod 2^k):
  ```
  Increment(A,k):
  i←0
  while i<k and A[i]=1 do
    A[i]←0
    i←i+1
  if i < k then
    A[i] ← 1
  
  Analysis:
  Each call could flip \(k\) bits,
  so \(n\) INCREMENTs takes \(O(nk)\) time.

---

**Example (1)**

<table>
<thead>
<tr>
<th>k=3</th>
<th>Counter</th>
<th>A</th>
<th>Value</th>
<th>2</th>
<th>10</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0 1 0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0 1 1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1 0 0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>1 0 1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>1 1 0</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>1 1 1</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>0 0 0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td></td>
<td>14</td>
</tr>
</tbody>
</table>

Cost of INCREMENT = (# of bits flipped)
Example (2)

| Bit | Flips how often | Time in 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>n INCREMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Every time</td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>1/2 of the time</td>
<td>floor(n/2)</td>
</tr>
<tr>
<td>2</td>
<td>1/4 of the time</td>
<td>floor(n/4)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>1/2 of the time</td>
<td>floor(n/2)</td>
</tr>
<tr>
<td>i+j</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Thus, total # flips = $\sum_{i=0}^{n/2} \left\lfloor \frac{n}{2} \right\rfloor + \sum_{i=0}^{n/4} \left\lfloor \frac{n}{4} \right\rfloor = n \left( \frac{1}{1-1/2} \right) = 2n$

Therefore, n INCREMENTS costs O(n).

Average cost per operation = $O(1)$.

Accounting method

Assign different charges to different operations.

- Some are charged more than actual cost.
- Some are charged less.

**Amortized cost** = amount we charge.

When amortized cost > actual cost, store the difference on specific objects in the data structure as **credit**.

Use credit later to pay for operations whose actual cost > amortized cost.

Diffs from aggregate analysis:

- In the accounting method, different operations can have different costs.
- In aggregate analysis, all operations have same cost.

**Warning:** Need credit to never go negative.

**Definition**

Let $c_i$ = cost of actual $i^{th}$ operation.

$\hat{c}_i$ = amortized cost of $i^{th}$ operation.

Then require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for all sequences of $n$ operations.

Total credit stored = $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$

**Stack**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Actual cost</th>
<th>Amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>POP</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MULTIPOP</td>
<td>min(k,s)</td>
<td>0</td>
</tr>
</tbody>
</table>

**Intuition:** When pushing an object, pay $\$2$.

- $\$1$ pays for the PUSH.
- $\$1$ is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has $\$1$, which is credit, the credit can never go negative.
- Total amortized cost, = $O(n)$, is an upper bound on total actual cost.

**Binary counter**

Charge $\$2$ to set a bit to 1.

- $\$1$ pays for setting a bit to 12.
- $\$1$ 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.
- Therefore, amortized cost $\leq \$2$.
- For $n$ operations, amortized cost = $O(n)$.

**Dynamic tables**

**Scenario**

- Have a table - maybe a hash table.
- Don’t know in advance how many objects will be stored in it.
- When it fills, must reallocate with a larger size, copying all objects into the new, larger table.
- When it gets sufficiently small, might want to reallocate with a smaller size.

**Goals**

1. $O(1)$ amortized time per operation.
2. Unused space always $\leq$ constant fraction of allocated space.

**Load factor** $\alpha = (\# \text{ items stored}) / (\text{allocated size})$

Never allow $\alpha > 1$; Keep $\alpha >$ a constant fraction $\Rightarrow$ Goal 2.
### Table expansion

Consider only insertion.
- When the table becomes full, double its size and reinsert all existing items.
- Each time we insert an item into the table, it is an elementary insertion.

**TABLE-INSERT**(T, x)

if size[T] = 0
then allocate table[T] with 1 slot
size[T] ← 1
if num[T] = size[T]
then allocate new-table with 2 · size[T] slots
insert all items in table[T] into new-table
free table[T]
size[T] ← 2 · size[T]
insert x into table[T]

### Aggregate analysis

- Charge 1 per elementary insertion.
- Count only elementary insertions (other costs = constant).
  - $c_i =$ actual cost of $i^{th}$ operation
- If not full, $c_i = 1$.
- If full, have $i-1$ items in the table at the start of the $i^{th}$ operation.

Have to copy all $i-1$ existing items, then insert $i^{th}$ item ⇒ $c_i = i$.

$n$ operations ⇒ $c_i = O(n)$ ⇒ $O(n)$ time for $n$ operations

$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} i$ if $i-1$ is power of 2

$\sum_{i=1}^{n} 1$ Otherwise

Total cost = $\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} i = \frac{2^n-1}{2-1} < n + 2n = 3n$

Amortized cost per operation = 3.

### Accounting method

Charge $3$ per insertion of x.
- $1$ pays for x’s insertion.
- $1$ pays for x to be moved in the future.
- $1$ pays for some other item to be moved.

Suppose we’ve just expanded, size=m before next expansion, size=2m after next expansion.
- Assume that the expansion used up all the credit, so that there’s no credit stored after the expansion.
- Will expand again after another $m$ insertions.
- Each insertion will put $1$ on one of the $m$ items that were in the table just after expansion and will put $1$ on the item inserted.
- Have $2m$ of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion...