DATA STRUCTURES I, II, III, AND IV

I. Amortized Analysis
II. Binary and Binomial Heaps
III. Fibonacci Heaps
IV. Union–Find

Lecture slides by Kevin Wayne
http://www.cs.princeton.edu/~wayne/kleinberg-tardos
Data structures

Static problems. Given an input, produce an output.
Ex. Sorting, FFT, edit distance, shortest paths, MST, max-flow, ...

Dynamic problems. Given a sequence of operations (given one at a time), produce a sequence of outputs.
Ex. Stack, queue, priority queue, symbol table, union–find, ....

Algorithm. Step-by-step procedure to solve a problem.
Data structure. Way to store and organize data.
Ex. Array, linked list, binary heap, binary search tree, hash table, ...
**Appetizer**

**Goal.** Design a data structure to support all operations in $O(1)$ time.
- **INIT($n$):** create and return an *initialized* array (all zero) of length $n$.
- **READ($A, i$):** return element $i$ in array.
- **WRITE($A, i, value$):** set element $i$ in array to $value$.

**Assumptions.**
- Can **malloc** an uninitialized array of length $n$ in $O(1)$ time.
- Given an array, can read or write element $i$ in $O(1)$ time.

**Remark.** An array does **INIT** in $\Theta(n)$ time and **READ** and **WRITE** in $\Theta(1)$ time.
**Appetizer**

**Data structure.** Three arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer $k$.

- $A[i]$ stores the current value for READ (if initialized).
- $k =$ number of initialized entries.
- $C[j] =$ index of $j^{th}$ initialized element for $j = 1, \ldots, k$.
- If $C[j] = i$, then $B[i] = j$ for $j = 1, \ldots, k$.

**Theorem.** $A[i]$ is initialized iff both $1 \leq B[i] \leq k$ and $C[B[i]] = i$.

**Pf.** Ahead.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B[\ ]$</td>
<td>?</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>?</td>
<td>2</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>$C[\ ]$</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

$k = 4$

Appetizer

**INIT** \((A, n)\)
---
\[
k \leftarrow 0.
\]
\[
A \leftarrow \text{MALLOC}(n).
\]
\[
B \leftarrow \text{MALLOC}(n).
\]
\[
C \leftarrow \text{MALLOC}(n).
\]

**READ** \((A, i)\)
---
\[
\text{IF} \ (\text{IS-INITIALIZED} \ (A[i]))
\]
\[
\text{RETURN} \ A[i].
\]
\[
\text{ELSE}
\]
\[
\text{RETURN} \ 0.
\]

**WRITE** \((A, i, value)\)
---
\[
\text{IF} \ (\text{IS-INITIALIZED} \ (A[i]))
\]
\[
A[i] \leftarrow \text{value}.
\]
\[
\text{ELSE}
\]
\[
k \leftarrow k + 1.
\]
\[
A[i] \leftarrow \text{value}.
\]
\[
B[i] \leftarrow k.
\]
\[
C[k] \leftarrow i.
\]

**IS-INITIALIZED** \((A, i)\)
---
\[
\text{IF} \ (1 \leq B[i] \leq k) \text{ and } (C[B[i]] = i)
\]
\[
\text{RETURN} \ true.
\]
\[
\text{ELSE}
\]
\[
\text{RETURN} \ false.
\]
**Theorem.** $A[i]$ is initialized iff both $1 \leq B[i] \leq k$ and $C[B[i]] = i$.

**Pf.** $\Rightarrow$

- Suppose $A[i]$ is the $j^{th}$ entry to be initialized.
- Then $C[j] = i$ and $B[i] = j$.
- Thus, $C[B[i]] = i$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B[i]$</td>
<td>?</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>?</td>
<td>2</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C[i]$</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

$k = 4$

Theorem. \( A[i] \) is initialized iff both \( 1 \leq B[i] \leq k \) and \( C[B[i]] = i \).

Pf. \( \iff \)

- Suppose \( A[i] \) is uninitialized.
- If \( B[i] < 1 \) or \( B[i] > k \), then \( A[i] \) clearly uninitialized.
- If \( 1 \leq B[i] \leq k \) by coincidence, then we still can’t have \( C[B[i]] = i \) because none of the entries \( C[1..k] \) can equal \( i \).

\[ k = 4 \]

\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}

Amortized Analysis

- binary counter
- multi-pop stack
- dynamic table

Lecture slides by Kevin Wayne
http://www.cs.princeton.edu/~wayne/kleinberg-tardos
Amortized analysis

**Worst-case analysis.** Determine worst-case running time of a data structure operation as function of the input size $n$.

Amortized analysis. Determine worst-case running time of a sequence of $n$ data structure operations.

**Ex.** Starting from an empty stack implemented with a dynamic table, any sequence of $n$ push and pop operations takes $O(n)$ time in the worst case.
Amortized analysis: applications

- Splay trees.
- Dynamic table.
- Fibonacci heaps.
- Garbage collection.
- Move-to-front list updating.
- Push–relabel algorithm for max flow.
- Path compression for disjoint-set union.
- Structural modifications to red–black trees.
- Security, databases, distributed computing, ...

**Abstract.** A powerful technique in the complexity analysis of data structures is amortization, or averaging over time. Amortized running time is a realistic but robust complexity measure for which we can obtain surprisingly tight upper and lower bounds on a variety of algorithms. By following the principle of designing algorithms whose amortized complexity is low, we obtain "self-adjusting" data structures that are simple, flexible and efficient. This paper surveys recent work by several researchers on amortized complexity.

**AMS(MOS) subject classifications.** 68C25, 68E05
Amortized Analysis

- binary counter
- multi-pop stack
- dynamic table
## Binary counter

**Goal.** Increment a $k$-bit binary counter (mod $2^k$).

**Representation.** $A[j] = j^{\text{th}}$ least significant bit of counter.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0/1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1/0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Cost model.** Number of bits flipped.
Binary counter

**Goal.** Increment a $k$-bit binary counter $(\text{mod } 2^k)$.

**Representation.** $A[j] = j^{th}$ least significant bit of counter.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Theorem.** Starting from the zero counter, a sequence of $n$ INCREMENT operations flips $O(nk)$ bits. **overly pessimistic upper bound**

**Pf.** At most $k$ bits flipped per increment. ■
Aggregate method (brute force)

**Aggregate method.** Analyze cost of a sequence of operations.

<table>
<thead>
<tr>
<th>Counter value</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>Total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>31</td>
</tr>
</tbody>
</table>
Binary counter: aggregate method

Starting from the zero counter, in a sequence of \( n \) INCREMENT operations:
- Bit 0 flips \( n \) times.
- Bit 1 flips \( \lfloor n/2 \rfloor \) times.
- Bit 2 flips \( \lfloor n/4 \rfloor \) times.
- ...

Theorem. Starting from the zero counter, a sequence of \( n \) INCREMENT operations flips \( O(n) \) bits.

Pf.
- Bit \( j \) flips \( \lfloor n/2^j \rfloor \) times.
- The total number of bits flipped is 
  \[
  \sum_{j=0}^{k-1} \left\lfloor \frac{n}{2^j} \right\rfloor < n \sum_{j=0}^{\infty} \frac{1}{2^j} = 2n \; \blacksquare
  \]

Remark. Theorem may be false if initial counter is not zero.
Accounting method (banker’s method)

Assign (potentially) different charges to each operation.
- \(D_i\) = data structure after \(i^{th}\) operation.
- \(c_i\) = actual cost of \(i^{th}\) operation.
- \(\hat{c}_i\) = amortized cost of \(i^{th}\) operation = amount we charge operation \(i\).
- When \(\hat{c}_i > c_i\), we store credits in data structure \(D_i\) to pay for future ops; when \(\hat{c}_i < c_i\), we consume credits in data structure \(D_i\).
- Initial data structure \(D_0\) starts with 0 credits.

Credit invariant. The total number of credits in the data structure \(\geq 0\).

\[ \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0 \]

Our job is to choose suitable amortized costs so that this invariant holds.

Can be more or less than actual cost.

initial data structure \(D_0\) starts with 0 credits.
Accounting method (banker’s method)

Assign (potentially) different charges to each operation.

• $D_i = \text{data structure after } i^{th} \text{ operation.}$
• $c_i = \text{actual cost of } i^{th} \text{ operation.}$
• $\hat{c}_i = \text{amortized cost of } i^{th} \text{ operation} = \text{amount we charge operation } i.$
• When $\hat{c}_i > c_i,$ we store credits in data structure $D_i$ to pay for future ops; when $\hat{c}_i < c_i,$ we consume credits in data structure $D_i.$
• Initial data structure $D_0$ starts with 0 credits.

Credit invariant. The total number of credits in the data structure $\geq 0.$

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

Theorem. Starting from the initial data structure $D_0,$ the total actual cost of any sequence of $n$ operations is at most the sum of the amortized costs.

Pf. The amortized cost of the sequence of $n$ operations is: $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i.$

Intuition. Measure running time in terms of credits (time = money).
Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit $j$ from 0 to 1: charge 2 credits (use one and save one in bit $j$).
Binary counter: accounting method

**Credits.** One credit pays for a bit flip.

**Invariant.** Each 1 bit has one credit; each 0 bit has zero credits.

**Accounting.**
- Flip bit \( j \) from 0 to 1: charge 2 credits (use one and save one in bit \( j \)).
- Flip bit \( j \) from 1 to 0: pay for it with the 1 credit saved in bit \( j \).

**increment**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

![Increment diagram]
Binary counter: accounting method

**Credits.** One credit pays for a bit flip.

**Invariant.** Each 1 bit has one credit; each 0 bit has zero credits.

**Accounting.**
- Flip bit $j$ from 0 to 1: charge 2 credits (use one and save one in bit $j$).
- Flip bit $j$ from 1 to 0: pay for it with the 1 credit saved in bit $j$.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

![Chip](image1.png) ![Chip](image2.png)
Binary counter: accounting method

Credits. One credit pays for a bit flip.
Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit \( j \) from 0 to 1: charge 2 credits (use one and save one in bit \( j \)).
- Flip bit \( j \) from 1 to 0: pay for it with the 1 credit saved in bit \( j \).

Theorem. Starting from the zero counter, a sequence of \( n \) INCREMENT operations flips \( O(n) \) bits.

Pf.

- Each INCREMENT operation flips at most one 0 bit to a 1 bit, so the amortized cost per INCREMENT \( \leq 2 \).
- Invariant \( \Rightarrow \) number of credits in data structure \( \geq 0 \).
- Total actual cost of \( n \) operations \( \leq \) sum of amortized costs \( \leq 2n \). □

accounting method theorem
Potential method (physicist’s method)

**Potential function.** $\Phi(D_i)$ maps each data structure $D_i$ to a real number s.t.:

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each data structure $D_i$.

**Actual and amortized costs.**

- $c_i = \text{actual cost of } i^{th} \text{ operation.}$
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \text{amortized cost of } i^{th} \text{ operation}.$

Our job is to choose a potential function so that the amortized cost of each operation is low.
Potential method (physicist’s method)

**Potential function.** $\Phi(D_i)$ maps each data structure $D_i$ to a real number s.t.:
- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each data structure $D_i$.

**Actual and amortized costs.**
- $c_i =$ actual cost of $i^{th}$ operation.
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) =$ amortized cost of $i^{th}$ operation.

**Theorem.** Starting from the initial data structure $D_0$, the total actual cost of any sequence of $n$ operations is at most the sum of the amortized costs.

**Pf.** The amortized cost of the sequence of operations is:

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

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i \quad \blacksquare$$
**Binary counter: potential method**

**Potential function.** Let $\Phi(D) =$ number of 1 bits in the binary counter $D$.
- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

**Increment**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Binary counter: potential method

**Potential function.** Let $\Phi(D) =$ number of 1 bits in the binary counter $D$.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

**increment**

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Binary counter: potential method

Potential function. Let $\Phi(D) = \text{number of 1 bits in the binary counter } D$.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$. 

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Binary counter: potential method

**Potential function.** Let $\Phi(D) =$ number of 1 bits in the binary counter $D$.
- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

**Theorem.** Starting from the zero counter, a sequence of $n$ INCREMENT operations flips $O(n)$ bits.

**Pf.**
- Suppose that the $i^{th}$ INCREMENT operation flips $t_i$ bits from 1 to 0.
- The actual cost $c_i \leq t_i + 1$. \hspace{1cm}
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  $\leq c_i + 1 - t_i$ \hspace{1cm}
  potential decreases by 1 for $t_i$ bits flipped from 1 to 0
  and increases by 1 for bit flipped from 0 to 1
  \hspace{1cm}$\leq 2$.
- Total actual cost of $n$ operations $\leq$ sum of amortized costs $\leq 2n$. □

potential method theorem
Famous potential functions

**Fibonacci heaps.** \( \Phi(H) = 2 \text{trees}(H) + 2 \text{marks}(H) \)

**Splay trees.** \( \Phi(T') = \sum_{x \in T} \left\lfloor \log_2 \text{size}(x) \right\rfloor \)

**Move-to-front.** \( \Phi(L) = 2 \text{inversions}(L, L^*) \)

**Preflow–push.** \( \Phi(f) = \sum_{v : \text{excess}(v) > 0} \text{height}(v) \)

**Red–black trees.** \( \Phi(T) = \sum_{x \in T} w(x) \)

\[
w(x) = \begin{cases} 
0 & \text{if } x \text{ is red} \\
1 & \text{if } x \text{ is black and has no red children} \\
0 & \text{if } x \text{ is black and has one red child} \\
2 & \text{if } x \text{ is black and has two red children}
\end{cases}
\]
**Amortized Analysis**

- binary counter
- multi-pop stack
- dynamic table

Section 17.4
**Multipop stack**

**Goal.** Support operations on a set of elements:

- **PUSH**($S, x$): add element $x$ to stack $S$.
- **POP**($S$): remove and return the most-recently added element.
- **MULTI-POP**($S, k$): remove the most-recently added $k$ elements.

**MULTI-POP**($S, k$)

FOR $i = 1$ TO $k$

**POP**($S$).

**Exceptions.** We assume **POP** throws an exception if stack is empty.
**Multipop stack**

**Goal.** Support operations on a set of elements:
- $\text{PUSH}(S, x)$: add element $x$ to stack $S$.
- $\text{POP}(S)$: remove and return the most-recently added element.
- $\text{MULTI-POP}(S, k)$: remove the most-recently added $k$ elements.

**Theorem.** Starting from an empty stack, any intermixed sequence of $n$ $\text{PUSH}$, $\text{POP}$, and $\text{MULTI-POP}$ operations takes $O(n^2)$ time.

**Pf.**
- Use a singly linked list.
- $\text{POP}$ and $\text{PUSH}$ take $O(1)$ time each.
- $\text{MULTI-POP}$ takes $O(n)$ time. □

![Linked list diagram](image)
Multipop stack: aggregate method

**Goal.** Support operations on a set of elements:
- \textbf{PUSH}$(S, x)$: add element $x$ to stack $S$.
- \textbf{POP}$(S)$: remove and return the most-recently added element.
- \textbf{MULTI-POP}$(S, k)$: remove the most-recently added $k$ elements.

**Theorem.** Starting from an empty stack, any intermixed sequence of $n$ \textbf{PUSH}, \textbf{POP}, and \textbf{MULTI-POP} operations takes $O(n)$ time.

**Pf.**
- An element is popped at most once for each time that it is pushed.
- There are $\leq n$ \textbf{PUSH} operations.
- Thus, there are $\leq n$ \textbf{POP} operations (including those made within \textbf{MULTI-POP}).
Multipop stack: accounting method

Credits. 1 credit pays for either a \texttt{PUSH} or \texttt{POP}.

Invariant. Every element on the stack has 1 credit.

Accounting.
- \texttt{PUSH}(S, x): charge 2 credits.
  - use 1 credit to pay for pushing \( x \) now
  - store 1 credit to pay for popping \( x \) at some point in the future
- \texttt{POP}(S): charge 0 credits.
- \texttt{MULTIPOP}(S, k): charge 0 credits.

Theorem. Starting from an empty stack, any intermixed sequence of \( n \) \texttt{PUSH}, \texttt{POP}, and \texttt{MULTIPOP} operations takes \( O(n) \) time.

Pf.
- Invariant \( \Rightarrow \) number of credits in data structure \( \geq 0 \).
- Amortized cost per operation \( \leq 2 \).
- Total actual cost of \( n \) operations \( \leq \) sum of amortized costs \( \leq 2n \). \( \blacksquare \)
Multipop stack: potential method

Potential function. Let $\Phi(D) =$ number of elements currently on the stack.
- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

Theorem. Starting from an empty stack, any intermixed sequence of $n$ \textsc{Push}, \textsc{Pop}, and \textsc{Multi-Pop} operations takes $O(n)$ time.

Pf. [Case 1: push]
- Suppose that the $i^{th}$ operation is a \textsc{Push}.
- The actual cost $c_i = 1$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$. 
Multipop stack: potential method

**Potential function.** Let \( \Phi(D) \) = number of elements currently on the stack.

- \( \Phi(D_0) = 0 \).
- \( \Phi(D_i) \geq 0 \) for each \( D_i \).

**Theorem.** Starting from an empty stack, any intermixed sequence of \( n \) \textsc{Push}, \textsc{Pop}, and \textsc{Multi-Pop} operations takes \( O(n) \) time.

**Pf.** [Case 2: \textsc{pop}]

- Suppose that the \( i^{th} \) operation is a \textsc{Pop}.
- The actual cost \( c_i = 1 \).
- The amortized cost \( \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0 \).
Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

Theorem. Starting from an empty stack, any intermixed sequence of $n$ \texttt{PUSH}, \texttt{POP}, and \texttt{MULTI-POP} operations takes $O(n)$ time.

Pf. [Case 3: multi-pop]

- Suppose that the $i^{th}$ operation is a \texttt{MULTI-POP} of $k$ objects.
- The actual cost $c_i = k$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k = 0$. $\blacksquare$
**Multipop stack: potential method**

**Potential function.** Let $\Phi(D) =$ number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

**Theorem.** Starting from an empty stack, any intermixed sequence of $n$ \textsc{push}, \textsc{pop}, and \textsc{multi-pop} operations takes $O(n)$ time.

**Pf.** [putting everything together]

- Amortized cost $\hat{c}_i \leq 2$. \hspace{1cm} $\leftarrow 2$ for push; $0$ for pop and multi-pop
- Sum of amortized costs $\hat{c}_i$ of the $n$ operations $\leq 2n$.
- Total actual cost $\leq$ sum of amortized cost $\leq 2n$. $\blacksquare$
Amortized Analysis

- binary counter
- multi-pop stack
- dynamic table
Dynamic table

**Goal.** Store items in a table (e.g., for hash table, binary heap).

- Two operations: **INSERT** and **DELETE**.
  - too many items inserted ⇒ **expand** table.
  - too many items deleted ⇒ **contract** table.
- Requirement: if table contains \( m \) items, then space = \( \Theta(m) \).

**Theorem.** Starting from an empty dynamic table, any intermixed sequence of \( n \) **INSERT** and **DELETE** operations takes \( O(n^2) \) time.

**Pf.** Each **INSERT** or **DELETE** takes \( O(n) \) time. •

**overly pessimistic upper bound**
Dynamic table: insert only

- When inserting into an empty table, allocate a table of capacity 1.
- When inserting into a full table, allocate a new table of twice the capacity and copy all items.
- Insert item into table.

<table>
<thead>
<tr>
<th>insert</th>
<th>old capacity</th>
<th>new capacity</th>
<th>insert cost</th>
<th>copy cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>16</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
</tbody>
</table>

Cost model. Number of items written (due to insertion or copy).
Dynamic table: insert only (aggregate method)

**Theorem.** [via aggregate method] Starting from an empty dynamic table, any sequence of \( n \) \texttt{INSERT} operations takes \( O(n) \) time.

**Pf.** Let \( c_i \) denote the cost of the \( i^{th} \) insertion.

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

Starting from empty table, the cost of a sequence of \( n \) \texttt{INSERT} operations is:

\[
\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n \quad \blacksquare
\]
Dynamic table demo: insert only (accounting method)

**Insert.** Charge 3 credits (use 1 credit to insert; save 2 with new item).

**Invariant.** 2 credits with each item in right half of table; none in left half.

**insert N**

capacity = 16

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th></th>
</tr>
</thead>
</table>

![Dynamic table diagram with inserted tokens](image-url)
Dynamic table: insert only (accounting method)

**Insert.** Charge 3 credits (use 1 credit to insert; save 2 with new item).

**Invariant.** 2 credits with each item in right half of table; none in left half.

**Pf.** [induction]
- Each newly inserted item gets 2 credits.
- When table doubles from $k$ to $2k$, $k/2$ items in the table have 2 credits.
  - these $k$ credits pay for the work needed to copy the $k$ items
  - now, all $k$ items are in left half of table (and have 0 credits)

**Theorem.** [via accounting method] Starting from an empty dynamic table, any sequence of $n$ INSERT operations takes $O(n)$ time.

**Pf.**
- Invariant $\implies$ number of credits in data structure $\geq 0$.
- Amortized cost per INSERT = 3.
- Total actual cost of $n$ operations $\leq$ sum of amortized cost $\leq 3n$. □
Dynamic table: insert only (potential method)

**Theorem.** [via potential method] Starting from an empty dynamic table, any sequence of $n$ INSERT operations takes $O(n)$ time.

**Pf.** Let $\Phi(D_i) = 2 \text{ size}(D_i) - \text{capacity}(D_i)$.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

\[ \Phi(D_i) = 2 \text{ size}(D_i) - \text{capacity}(D_i) \]

\[ \Phi(D_i) \geq 0 \quad \text{for each } D_i \]

\[ \text{size} = 6 \]
\[ \text{capacity} = 8 \]
\[ \Phi = 4 \]
Dynamic table: insert only (potential method)

**Theorem.** [via potential method] Starting from an empty dynamic table, any sequence of \( n \) INSERT operations takes \( O(n) \) time.

**Pf.** Let \( \Phi(D_i) = 2 \text{ size}(D_i) - \text{capacity}(D_i) \).

- \( \Phi(D_0) = 0 \).
- \( \Phi(D_i) \geq 0 \) for each \( D_i \).

**Case 0.** [first insertion]

- Actual cost \( c_1 = 1 \).
- \( \Phi(D_1) - \Phi(D_0) = (2 \text{ size}(D_1) - \text{capacity}(D_1)) - (2 \text{ size}(D_0) - \text{capacity}(D_0)) \)
  \[= 1.\]
- Amortized cost \( \hat{c}_i = c_i + (\Phi(D_1) - \Phi(D_0)) \)
  \[= 1 + 1\]
  \[= 2.\]
Dynamic table: insert only (potential method)

**Theorem.** [via potential method] Starting from an empty dynamic table, any sequence of $n$ INSERT operations takes $O(n)$ time.

**Pf.** Let $\Phi(D_i) = 2 \text{size}(D_i) - \text{capacity}(D_i)$.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

**Case 1.** [no array expansion] $\text{capacity}(D_i) = \text{capacity}(D_{i-1})$.

- Actual cost $c_i = 1$.
- $\Phi(D_i) - \Phi(D_{i-1}) = (2 \text{size}(D_i) - \text{capacity}(D_i)) - (2 \text{size}(D_{i-1}) - \text{capacity}(D_{i-1}))$
  
  \[ = 2. \]
- Amortized cost $\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$
  
  \[ = 1 + 2 \]
  
  \[ = 3. \]
Dynamic table: insert only (potential method)

Theorem. [via potential method] Starting from an empty dynamic table, any sequence of $n$ INSERT operations takes $O(n)$ time.

Pf. Let $\Phi(D_i) = 2 \text{size}(D_i) - \text{capacity}(D_i)$.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

Case 2. [array expansion] $\text{capacity}(D_i) = 2 \text{capacity}(D_{i-1})$.

- Actual cost $c_i = 1 + \text{capacity}(D_{i-1})$.
- $\Phi(D_i) - \Phi(D_{i-1}) = (2 \text{size}(D_i) - \text{capacity}(D_i)) - (2 \text{size}(D_{i-1}) - \text{capacity}(D_{i-1}))$
  $= 2 - \text{capacity}(D_i) + \text{capacity}(D_{i-1})$
  $= 2 - \text{capacity}(D_{i-1}).$
- Amortized cost $\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$
  $= 1 + \text{capacity}(D_{i-1}) + (2 - \text{capacity}(D_{i-1}))$
  $= 3.$
Dynamic table: insert only (potential method)

**Theorem.** [via potential method] Starting from an empty dynamic table, any sequence of $n$ INSERT operations takes $O(n)$ time.

**Pf.** Let $\Phi(D_i) = 2 \text{size}(D_i) - \text{capacity}(D_i)$.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each $D_i$.

[putting everything together]
- Amortized cost per operation $\hat{c}_i \leq 3$.
- Total actual cost of $n$ operations $\leq$ sum of amortized cost $\leq 3n$. $\blacksquare$
Dynamic table: doubling and halving

Thrashing.
- **INSERT**: when inserting into a full table, double capacity.
- **DELETE**: when deleting from a table that is \( \frac{1}{2} \)-full, halve capacity.

Efficient solution.
- When inserting into an empty table, initialize table size to 1; when deleting from a table of size 1, free the table.
- **INSERT**: when inserting into a full table, double capacity.
- **DELETE**: when deleting from a table that is \( \frac{1}{4} \)-full, halve capacity.

Memory usage. A dynamic table uses \( \Theta(n) \) memory to store \( n \) items.

**Pf.** Table is always between 25% and 100% full. □
**Dynamic table demo: insert and delete (accounting method)**

**Insert.** Charge 3 credits (1 to insert; save 2 with item if in right half).

**Delete.** Charge 2 credits (1 to delete; save 1 in empty slot if in left half).

**Invariant 1.** 2 credits with each item in right half of table.

**Invariant 2.** 1 credit with each empty slot in left half of table.

delete M

capacity = 16

| A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

![Image of tokens]
Dynamic table: insert and delete (accounting method)

**Insert.** Charge 3 credits (1 to insert; save 2 with item if in right half).

**Delete.** Charge 2 credits (1 to delete; save 1 in empty slot if in left half).

**Invariant 1.** 2 credits with each item in right half of table.

**Invariant 2.** 1 credit with each empty slot in left half of table.

**Theorem.** [via accounting method] Starting from an empty dynamic table, any intermixed sequence of \( n \) INSERT and DELETE operations takes \( O(n) \) time.

**Pf.**
- Invariants \( \Rightarrow \) number of credits in data structure \( \geq 0 \).
- Amortized cost per operation \( \leq 3 \).
- Total actual cost of \( n \) operations \( \leq \) sum of amortized cost \( \leq 3n \). □
Dynamic table: insert and delete (potential method)

**Theorem.** [via potential method] Starting from an empty dynamic table, any intermixed sequence of $n$ INSERT and DELETE operations takes $O(n)$ time.

**Pf sketch.**

- Let $\alpha(D_i) = \text{size}(D_i) / \text{capacity}(D_i)$.

- Define $\Phi(D_i) = \begin{cases} 2 \text{size}(D_i) - \text{capacity}(D_i) & \text{if } \alpha(D_i) \geq 1/2 \\ \frac{1}{2} \text{capacity}(D_i) - \text{size}(D_i) & \text{if } \alpha(D_i) < 1/2 \end{cases}$

- $\Phi(D_0) = 0, \Phi(D_i) \geq 0$. [a potential function]
- When $\alpha(D_i) = 1/2, \Phi(D_i) = 0$. [zero potential after resizing]
- When $\alpha(D_i) = 1, \Phi(D_i) = \text{size}(D_i)$. [can pay for expansion]
- When $\alpha(D_i) = 1/4, \Phi(D_i) = \text{size}(D_i)$. [can pay for contraction]

...