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, ...
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 $i^{th}$ element of array.
- **WRITE($A, i, value$):** set $i^{th}$ element of array to $value$.

Assumptions.

- Can `malloc` an uninitialized array of length $n$ in $O(1)$ time.
- Given an array, can read or write $i^{th}$ element in $O(1)$ time.

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


- $A[i]$ stores the current value for READ (if initialized).
- $k =$ number of initialized entries.
- $C[j] =$ index of $j^{th}$ initialized entry 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.

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

$k = 4$

**Appetizer**

**INIT** ($A, n$)

$k \leftarrow 0.$

$A \leftarrow \text{MALLOC}(n).$

$B \leftarrow \text{MALLOC}(n).$

$C \leftarrow \text{MALLOC}(n).$

**READ** ($A, i$)

**IF** (INITIALIZED ($A[i]$))

**RETURN** $A[i].$

**ELSE**

**RETURN** 0.

**WRITE** ($A, i, value$)

**IF** (INITIALIZED ($A[i]$))

$A[i] \leftarrow value.$

**ELSE**

$k \leftarrow k + 1.$

$A[i] \leftarrow value.$

$B[i] \leftarrow k.$

$C[k] \leftarrow i.$

**INITIALIZED** ($A, i$)

**IF** ($1 \leq B[i] \leq k$) and ($C[B[i]] = i$)

**RETURN** true.

**ELSE**

**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, $C[B[i]] = i$.

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

$k = 4$

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

Pf. $\Leftarrow$

- 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 $i$. $\blacksquare$

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

$k = 4$

Amortized Analysis

- binary counter
- multipop stack
- dynamic table
Amortized analysis

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

Amortized analysis. Determine worst-case running time of a sequence of data structure operations as a function of the input size.

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, ...

AMORTIZED COMPUTATIONAL COMPLEXITY

ROBERT ENDRE TARJAN†

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
- multipop stack
- dynamic table

Chapter 17
**Binary counter**

**Goal.** Increment a $k$-bit binary counter (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>0</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>0</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>0</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>0</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>

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

**Goal.** Increment a $k$-bit binary counter (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>0000000000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0000000001</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0000000010</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0000000100</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0000001000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0000010000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0000100000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0000000011</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0000000101</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0000001001</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0000010010</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0000100100</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0001000000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0000011000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0000101100</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0001010100</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0001110100</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

**Aggregate method.** Sum up sequence of operations, weighted by their cost.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>0 0 0 0 0 1 0 1 1 0 0 0 0 0 1 1</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>0 0 0 1 0 1 0 0 1 0 0 0 0 1 1 0</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 0</td>
<td>22</td>
</tr>
<tr>
<td>13</td>
<td>0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 0</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>0 0 0 1 1 1 1 0 0 0 1 0 0 0 1 1</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>0 0 0 1 1 1 1 0 0 0 1 0 0 0 1 1</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>31</td>
</tr>
</tbody>
</table>

**Figure 17.2** An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is always less than twice the total number of INCREMENT operations.
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
  \[
  \sum_{j=0}^{k-1} \lfloor \frac{n}{2^j} \rfloor < n \sum_{j=0}^{\infty} \frac{1}{2^j} = 2n \quad \blacksquare
  \]

**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 zero credits.

**Key invariant.** The total number of credits in the data structure \( \geq 0 \).
\[
\sum_{i=1}^{\hat{c}_i} - \sum_{i=1}^{c_i} \geq 0
\]
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 zero credits.

Key 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 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 two 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 two credits (use one and save one in bit $j$).
• Flip bit $j$ from 1 to 0: pay for it with the one 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>
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 two credits (use one and save one in bit $j$).
- Flip bit $j$ from 1 to 0: pay for it with the one 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>
```
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 two credits (use one and save one in bit \( j \)).
- Flip bit \( j \) from 1 to 0: pay for it with the one 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 total amortized cost is at most \( 2n \)).
- The invariant is maintained. \( \Rightarrow \) number of credits in each bit \( \geq 0 \). \( \blacksquare \)
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.
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 \( i^{\text{th}} \) operation.
- \( \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \) amortized cost of \( i^{\text{th}} \) 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>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 $D_i$. 

<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 $c_i \leq t_i + 1$.  
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  \[\leq c_i + 1 - t_i\]
  \[\leq 2. \quad \blacksquare\]
Famous potential functions

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

**Splay trees.** \( \Phi(T) = \sum_{x \in T} \left\lfloor \log_2 \text{size}(x) \right\rfloor \)

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

**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
- multipop stack
- dynamic table
Multipop stack

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

- **PUSH(S, x):** push object x onto stack S.
- **POP(S):** remove and return the most-recently added object.
- **MULTIPOP(S, k):** remove the most-recently added k objects.

```
MULTIPOP (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:
- \( \text{PUSH}(S, x) \): push object \( x \) onto stack \( S \).
- \( \text{POP}(S) \): remove and return the most-recently added object.
- \( \text{MULTIPOP}(S, k) \): remove the most-recently added \( k \) objects.

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

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

![Diagram of a stack with elements 1, 4, 1, 3 and a pointer labeled as 'top'.]
Multipop stack: aggregate method

Goal. Support operations on a set of elements:

- \texttt{PUSH}(S, x): push object \( x \) onto stack \( S \).
- \texttt{POP}(S): remove and return the most-recently added object.
- \texttt{MULTIPOP}(S, k): remove the most-recently added \( k \) objects.

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

Pf.

- An object is popped at most once for each time it is pushed onto stack.
- There are \( \leq n \) \texttt{PUSH} operations.
- Thus, there are \( \leq n \) \texttt{POP} operations (including those made within \texttt{MULTIPOP}).
Multipop stack: accounting method

Credits. One credit pays for a push or pop.

Accounting.
• \( \text{PUSH}(S, x) \): charge two credits.
  - use one credit to pay for pushing \( x \) now
  - store one credit to pay for popping \( x \) at some point in the future
• No other operation is charged a credit.

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

Pf. The algorithm maintains the invariant that every object remaining on the stack has 1 credit \( \Rightarrow \) number of credits in data structure \( \geq 0 \).
Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of objects 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 $n$ PUSH, POP, and MULTIPOP operations takes $O(n)$ time.

Pf. [Case 1: push]
- Suppose that the $i^{th}$ operation is a 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 objects 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 $n$ 
$\text{PUSH}$, $\text{POP}$, and $\text{MULTIPOP}$ operations takes $O(n)$ time.

Pf. [Case 2: pop]
- Suppose that the $i^{th}$ operation is a $\text{POP}$.
- 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 objects 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 $n$ PUSH, POP, and MULTIPOP operations takes $O(n)$ time.

**Pf.** [Case 3: multipop]
- Suppose that the $i^{th}$ operation is a MULTIPOP of $k$ objects.
- The actual cost $c_i = k$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k = 0$. □
Amortized Analysis

- binary counter
- multipop stack
- dynamic table

Section 17.4
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.** A single **INSERT** or **DELETE** takes \( O(n) \) time. □

overly pessimistic
upper bound
Dynamic table: insert only

- Initialize table to be size 1.
- **INSERT:** if table is full, first copy all items to a table of twice the size.

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

**Cost model.** Number of items that are copied.
Dynamic table: insert only (aggregate method)

**Theorem.** [via aggregate method] Starting from an empty dynamic table, any sequence of \( n \) 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 \( n \) INSERT operations is:

\[
\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{[\lg n]} 2^j
\]

\[
< n + 2n
\]

\[
= 3n \quad \blacksquare
\]
**Dynamic table: insert only (accounting method)**

WLOG, can assume the table fills from left to right.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

![Dynamic table 1](image)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

![Dynamic table 2](image)

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

![Dynamic table 3](image)
Dynamic table: insert only (accounting method)

Accounting.
- **INSERT**: charge 3 credits (use 1 credit to insert; save 2 with new item).

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

**Pf.** The algorithm maintains the invariant that there are 2 credits with each item in right half of table.
- When table doubles, one-half of the items in the table have 2 credits.
- This pays for the work needed to double the table. □
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 \( \Phi(D_i) = 2 \text{size}(D_i) - \text{capacity}(D_i) \).

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]
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 $\Phi(D_i) = 2 \text{size}(D_i) - \text{capacity}(D_i)$.

Case 1. [does not trigger expansion] $\text{size}(D_i) \leq \text{capacity}(D_{i-1})$.

- Actual cost $c_i = 1$.
- $\Phi(D_i) - \Phi(D_{i-1}) = 2$.
- Amortized costs $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 = 3$.

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

- Actual cost $c_i = 1 + \text{capacity}(D_{i-1})$.
- $\Phi(D_i) - \Phi(D_{i-1}) = 2 - \text{capacity}(D_i) + \text{capacity}(D_{i-1}) = 2 - \text{capacity}(D_{i-1})$.
- Amortized costs $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 = 3$. □
Dynamic table: doubling and halving

Thrashing.
- Initialize table to be of fixed size, say 1.
- \textbf{INSERT}: if table is full, expand to a table of twice the size.
- \textbf{DELETE}: if table is $\frac{1}{2}$-full, contract to a table of half the size.

Efficient solution.
- Initialize table to be of fixed size, say 1.
- \textbf{INSERT}: if table is full, expand to a table of twice the size.
- \textbf{DELETE}: if table is $\frac{1}{4}$-full, contract to a table of half the size.

Memory usage. A dynamic table uses $O(n)$ memory to store $n$ items.

\textbf{Pf.} Table is always at least $\frac{1}{4}$-full (provided it is not empty). □
Dynamic table: insert and delete (accounting method)

### Insert

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
</table>

### Delete

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
</table>

### Resize and Delete

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>
Dynamic table: insert and delete (accounting method)

**Accounting.**
- **INSERT:** charge 3 credits (1 credit for insert; save 2 with new item).
- **DELETE:** charge 2 credits (1 credit to delete, save 1 in emptied slot).

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

**Pf.** The algorithm maintains the invariant that there are 2 credits with each item in the right half of table; 1 credit with each empty slot in the left half.
  - When table doubles, each item in right half of table has 2 credits.
  - When table halves, each empty slot in left half of table has 1 credit.  ■
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 $\alpha(D_i) = \frac{\text{size}(D_i)}{\text{capacity}(D_i)}$.

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

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

...