DATA STRUCTURES I, II, III, AND IV

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

Appelizer

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.

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, ...

Appelizer

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

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

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

Pf. $
\begin{itemize}
  \item Suppose $A[i]$ is the $j^{th}$ entry to be initialized.
  \item Then $C[j] = i$ and $B[i] = j$.
  \item Thus, $C[B[i]] = i$.
\end{itemize}$

<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>
<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>

$k = 4$


---

**Amortized Analysis**

- binary counter
- multipop stack
- dynamic table

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 2/4/16 5:09 AM
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, ...

Binary counter

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

Representation. \( a_j = f^j \) least significant bit of counter.

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.

| Counter value | $0000000000000000$ | $0000000000000001$ | $0000000000000010$ | $0000000000000011$ | $0000000000000100$ | $0000000000000101$ | $0000000000000110$ | $0000000000000111$ | $0000000000001000$ | $0000000000001001$ | $0000000000001010$ | $0000000000001011$ | $0000000000001100$ | $0000000000001101$ | $0000000000001110$ | $0000000000001111$ | $0000000000010000$ | $0000000000010001$ | $0000000000010010$ | $0000000000010011$ | $0000000000010100$ | $0000000000010101$ | $0000000000010110$ | $0000000000010111$ | $0000000000011000$ | $0000000000011001$ | $0000000000011010$ | $0000000000011011$ | $0000000000011100$ | $0000000000011101$ | $0000000000011110$ | $0000000000011111$ |
|---------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|

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

**Pf.** At most $k$ bits flipped per increment.  

Binary counter: aggregate method

Starting from the zero counter, in a sequence of $n$ INCREMENT operations:

- Bit 0 flips $n$ times.
- Bit 1 flips $\lceil n/2 \rceil$ times.
- Bit 2 flips $\lceil n/4 \rceil$ times.
- ...

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

**Pf.**

- Bit $j$ flips $\lceil n/2^j \rceil$ times.
- The total number of bits flipped is
  \[ \sum_{j=0}^{\log_2 n} \frac{n}{2^j} < n \sum_{j=0}^{\infty} \frac{1}{2^j} = 2n \]

**Remark.** Theorem may be false if initial counter is not zero.

Aggregate method (brute force)

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

| Counter value | $0000000000000000$ | $0000000000000001$ | $0000000000000010$ | $0000000000000011$ | $0000000000000100$ | $0000000000000101$ | $0000000000000110$ | $0000000000000111$ | $0000000000001000$ | $0000000000001001$ | $0000000000001010$ | $0000000000001011$ | $0000000000001100$ | $0000000000001101$ | $0000000000001110$ | $0000000000001111$ | $0000000000010000$ | $0000000000010001$ | $0000000000010010$ | $0000000000010011$ | $0000000000010100$ | $0000000000010101$ | $0000000000010110$ | $0000000000010111$ | $0000000000011000$ | $0000000000011001$ | $0000000000011010$ | $0000000000011011$ | $0000000000011100$ | $0000000000011101$ | $0000000000011110$ | $0000000000011111$ |
|---------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|

| Total cost    | $0$               | $1$               | $2$               | $3$               | $4$               | $5$               | $6$               | $7$               | $8$               | $9$               | $10$              | $11$              | $12$              | $13$              | $14$              | $15$              | $16$              | $17$              | $18$              | $19$              | $20$              | $21$              | $22$              | $23$              | $24$              | $25$              | $26$              | $27$              | $28$              | $29$              | $30$              | $31$              |
|---------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|

**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.
- $\tilde{c}_i$ = amortized cost of $i^{th}$ operation = amount we charge operation $i$.
- When $\tilde{c}_i > c_i$, we store credits in data structure $D_i$ to pay for future ops; when $\tilde{c}_i < c_i$, 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 $\geq 0$.

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

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

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: $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$. $lacklozenge$

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 $j$).
- 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>

- -

<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.

**Proof.**
- 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 \( \geq 0 \). ■

---

**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.

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

\[
\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\
= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \\
\geq \sum_{i=1}^{n} c_i. \quad \square
\]
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>

Theorem. Starting from the zero counter, a sequence of $n$ INCREDENT 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} \text{operation sets one bit to 1 (unless counter resets to zero)}
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  \hspace{1cm} \leq c_i + 1 - t_i
  \hspace{1cm} \leq 2. \hspace{1cm} \blacksquare

Famous potential functions

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

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

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

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

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

---

**Multipop stack: aggregate method**

**Goal.** Support operations on a set of elements:
- \textsc{PUSH}(S,x): push object \(x\) onto stack \(S\).
- \textsc{POP}(S): remove and return the most-recently added object.
- \textsc{MULTIPOP}(S,k): remove the most-recently added \(k\) objects.

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

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

---

**Exceptions.** We assume \textsc{POP} throws an exception if stack is empty.
Multipop stack: accounting method

Credits. One credit pays for a push or pop.

Accounting.
- \text{PUSH}(S, x): 
  - charge two credits.
- \text{PUSH}(S, x): 
  - 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 \( n \) \text{PUSH}, \text{POP}, and \text{MULTIPOP} operations takes \( O(n) \) time.

Pf. [Case 1: push]
- Suppose that the \( i \)th operation is a \text{PUSH}.
- The actual cost \( c_i = 1 \).
- The amortized cost \( \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2 \).

Pf. [Case 2: pop]
- Suppose that the \( i \)th operation is a \text{POP}.
- The actual cost \( c_i = 1 \).
- The amortized cost \( \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0 \).

Pf. [Case 3: multipop]
- Suppose that the \( i \)th operation is a \text{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

**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. 

**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.

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

Starting from empty table, the cost of a sequence of $n$ **INSERT** operations is:

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j$$

$$\leq n + 2n$$

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

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

1 2 3 4

1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

**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: 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: doubling and halving

Thrashing.
- Initialize table to be of fixed size, say 1.
- **INSERT:** if table is full, expand to a table of twice the size.
- **DELETE:** if table is ½-full, contract to a table of half the size.

Efficient solution.
- Initialize table to be of fixed size, say 1.
- **INSERT:** if table is full, expand to a table of twice the size.
- **DELETE:** if table is ¼-full, contract to a table of half the size.

Memory usage. A dynamic table uses $O(n)$ memory to store $n$ items.
**Pf.** Table is always at least ¼-full (provided it is not empty).

---

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

...