Chap 17 – Amortized Analysis

17.1 Aggregate analysis

17.2 The accounting method

17.3 The potential method

17.4 Dynamic tables

Amortized Analysis

- Amortized analysis
  - Analyze a *sequence* of operations on a data structure.
  - ***Goal***

    Show that although some individual operations may be expensive, *on average* the cost per operation is small.
  - *Average* in this context means we are averaging over a sequence of operations, rather than over a distribution of inputs.
    - No probability is involved.
    - We're talking about *average cost per operation in the worst case*.

17.1 Aggregate analysis

- Aggregate analysis
  - Show that a sequence of n operations takes worst-case time $T(n)$ in total.

    In the worst case, the average cost, or **amortized cost**, per operation is therefore $T(n)/n$.
  - The amortized cost applies to each operation, regardless of the type of the operation.

17.1 Aggregate analysis

- Aggregate analysis: Stack operations
  - PUSH(S, x) Cost: $O(1)$

    POP(S) Cost: $O(1)$

    A sequence of n PUSH and POP operations takes $O(n)$ time in total.

    Amortized cost per operation is $O(1)$ – NO INTEREST!
  - Let's add a new operation:

    MULTIPOP(S, k)

    while S is not empty and $k > 0$

    POP(S)

    $k = k - 1$

    Cost: $O(\min(s, k))$, where $s =$ # of objects on stack S

17.1 Aggregate analysis

- Aggregate analysis: Stack operations
  - In a sequence of n PUSH, POP and MULTIPOP operations:
    - PUSH $O(1)$ cost
    - POP $O(1)$ cost
    - MULTIPOP $O(n)$ worst-case cost
  - PUSH, POP, PUSH, POP, PUSH, POP, ...

    Total cost: n

    PUSH, PUSH, MULTIPOP$(S, 2)$, PUSH, PUSH, MULTIPOP$(S, 2)$, ...

    Total cost: $4n/3$

    PUSH, PUSH, ..., PUSH, MULTIPOP$(S, n-1)$

    Total cost: $2n - 2$

17.1 Aggregate analysis

- Aggregate analysis: Stack operations
  - The amortized cost per operation is $O(1)$.
    - Have $\leq n$ PUSHes

      $\Rightarrow \leq n$ POPs, including those in MULTIPOP
    - Therefore, total cost $\leq 2n = O(n)$
    - Average over the n operations

      $\Rightarrow O(n)/n = O(1)$ per operation on average

17.1 Aggregate analysis

- Aggregate analysis: Binary counter
  - k-bit binary counter $A[0..k-1]$ of bits
  - Values of counter: $0..2^k - 1$
  - Count upward from 0; $A[0..k-1] = 0$ initially
  - To increment, add 1 (mod 2^k)

INCREMENT(A, k)
$i = 0$
while $i < k$ and $A[i] == 1$
    $A[i] = 0$
    $i = i + 1$
if $i < k$
    $A[i] = 1$

Cost of INCREMENT: Θ(# of bits flipped)

17.1 Aggregate analysis

- Aggregate analysis: Binary counter
  - counter value, A (2 1 0), cost of each INCREMENT, total cost

counter value	A (2 1 0)	cost of each INCREMENT	total cost
0	0 0 0		0
1	0 0 1	1	1
2	0 1 0	2	3
3	0 1 1	1	4
4	1 0 0	3	7
5	1 0 1	1	8
6	1 1 0	2	10
7	1 1 1	1	11
0	0 0 0	3	14

17.1 Aggregate analysis

- Aggregate analysis: Binary counter
  - In a sequence of n INCREMENTs on an initially zero k-bit counter, a single INCREMENT operation takes $O(k)$ time in the worst case (when the value of the counter is $2^k - 1$).
  - However, not every bit flips every time.

bit	flips how often	times in n INCREMENTs
0	every time	n
1	$1/2$ the time	$\lfloor n/2 \rfloor$
2	$1/4$ the time	$\lfloor n/4 \rfloor$
$\vdots$	$\vdots$	$\vdots$
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
$\vdots$	$\vdots$	$\vdots$

17.1 Aggregate analysis

- Aggregate analysis: Binary counter
  - For $i > \lfloor \lg n \rfloor$, bit $A[i]$ never flips at all.

    $\because \lfloor n/2^i \rfloor = 0 \Rightarrow 2^i > n \Rightarrow i > \lg n \geq \lfloor \lg n \rfloor$

    Thus,

    $$\text{total \# of flips} = \sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} 1/2^i = 2n$$

  - Therefore, total cost $= O(n)$.

    Amortized cost per operation $= O(1)$

17.2 The accounting method

- Accounting method
  - Assign different charges to different operations.
    - Some are charged more than actual cost.
    - Some are charged less.
  - Differs from aggregate analysis: In aggregate analysis, all operations have the same cost.
  - ***Amortized cost*** = amount we charge.
  - When amortized cost > actual cost, store the difference *on specific objects* in the data structure as ***credit***.
  - When actual cost > amortized cost, use credit to pay for the difference.

17.2 The accounting method

- Accounting method
  - Need credit to never go negative.
    - Otherwise, the amortized cost of such a sequence of operations is not an upper bound on the actual cost.
    - Amortized cost would tell us *nothing*.
  - Let c_i = actual cost of i^{th} operation

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

    Then, for every sequence of n operations, we need

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

17.2 The accounting method

- Accounting method: Stack operations
  - operation — actual cost — amortized cost

operation	actual cost	amortized cost	
PUSH	1	2	$O(1)$ amortized cost
POP	1	0	
MULTIPOP	$\min(k, s)$	0	

  - ***Intuition:*** When pushing an object, pay \$2.
    - \$1 pays for the PUSH
    - \$1 is prepayment for it being popped by POP or MULTIPOP
    - Each object has \$1 of credit

      $\Rightarrow$ credit ≥ 0

      $\Rightarrow$ total amortized cost $O(n)$ is an upper bound on total actual cost

17.2 The accounting method

- Accounting method: Binary counter
  - Charge \$2 to set a bit to 1.
    - \$1 pays for setting a bit to 1.
    - \$1 is prepayment for flipping it back to 0.
    - Have \$1 of credit for every 1 in the counter

      $\Rightarrow$ credit ≥ 0
  - Amortized cost of INCREMENT $= O(1)$
    - Cost of resetting bits to 0 is paid by credit.
    - At most 1 bit is set to 1.
    - Therefore, amortized cost of INCREMENT $\leq$ \$2.
    - For n operations, amortized cost $= O(n)$

17.2 The accounting method

- Aggregate analysis: Binary counter
  - counter A

value	2 1 0	c_i	$\hat{c}_i$
0	0 0 0		
1	0 0 1	1	2
2	0 1 0	2	2
3	0 1 1	1	2
4	1 0 0	3	2
5	1 0 1	1	2
6	1 1 0	2	2
7	1 1 1	1	2
0	0 0 0	3	0

Credit stored in each bit

17.3 The potential method

- Potential method
  - Like the accounting method, but think of the credit as ***potential*** stored with the entire data structure.
    - Accounting method stores credit with specific objects.
    - Potential method stores potential in the data structure as a whole.
    - Potential can be released to pay for future operations
    - Most flexible of the amortized analysis methods

17.3 The potential method

- Potential method
  - Let D_0 = initial data structure

    D_i = data structure after i^{th} operation

    c_i = actual cost of i^{th} operation

    $\hat{c}_i$ = amortized cost of i^{th} operation
  - ***Potential function***

    $\Phi : \{D_i \mid i = 0,1,2, \ldots\} \rightarrow \mathbb{R}$

    $\Phi(D_i)$ is the *potential* associated with data structure D_i
  - $\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\Delta\Phi(D_i)}$

    $\Delta\Phi(D_i)$ = in- or de-crease in potential due to i^{th} operation

17.3 The potential method

- Potential method
  - Total amortized cost

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

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

  - If we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, the amortized cost is always an upper bound on the actual cost.

    In practice

    $\Phi(D_0) = 0$, $\Phi(D_i) \geq 0$ for all i

17.3 The potential method

- Potential method: Stack operations
  - Φ = # of objects in stack
    = # of \$1 bills in accounting method
    - D_0 = empty stack $\Rightarrow \Phi(D_0) = 0$
    - # of objects in stack is always $\geq 0 \Rightarrow \Phi(D_i) \geq 0$ for all i
  - Cost of PUSH operation

$$c_i = 1$$

$$\Delta\Phi(D_i) = \Phi(D_i) - \Phi(D_{i-1})$$

$$= (s+1) - s = 1 \quad \text{where } s = \Phi(D_{i-1})$$

$$\hat{c_i} = c_i + \Delta\Phi(D_i) = 1 + 1 = 2$$

17.3 The potential method

- Potential method: Stack operations
  - Cost of POP operation

    $c_i = 1$

    $\Delta\Phi(D_i) = (s-1) - s = -1$

    $\widehat{c_i} = c_i + \Delta\Phi(D_i) = 1 - 1 = 0$
  - Cost of MULTIPOP operation

    $c_i = \min(s, k) = k'$

    $\Delta\Phi(D_i) = (s - k') - s = -k'$

    $\widehat{c_i} = c_i + \Delta\Phi(D_i) = k' - k' = 0$
  - So, amortized cost of each operation $= O(1)$

    amortized cost of a sequence of n operations $= O(n)$

17.3 The potential method

- Potential method: Binary counter
  - Φ = # of 1's in counter
    - If counter starts at 0, D_0 = all bits are 0's $\Rightarrow \Phi(D_0) = 0$
    - # of 1's in counter is always $\geq 0 \Rightarrow \Phi(D_i) \geq 0$ for all i
  - Suppose the i^{th} operation resets t_i bits to 0

    Since it sets at most one bit to 1, $c_i \leq 1 + t_i$ (1)

    CLAIM $\Delta\Phi(D_i) \leq 1 - t_i$ (2)

    By (1) and (2), we have

    $\hat{c_i} = c_i + \Delta\Phi(D_i) \leq (1 + t_i) + (1 - t_i) = 2$
  - So, amortized cost of INCREMENT $= O(1)$

    amortized cost of a sequence of n INCREMENT's $= O(n)$

17.3 The potential method

- Potential method: Binary counter
  - **CLAIM** $\Delta\Phi(D_i) \leq 1 - t_i$

    Let $\Phi(D_i) = b_i$

    Case 1: $b_i = 0$

    The i^{th} operation resets all k bits and doesn't set one, so

    $b_{i-1} = t_i = k \Rightarrow b_i = b_{i-1} - t_i \Rightarrow b_i - b_{i-1} = -t_i$

    Case 2: $b_i > 0$

    The i^{th} operation resets t_i bits and sets one, so

    $b_i = b_{i-1} - t_i + 1 \Rightarrow b_i - b_{i-1} = 1 - t_i$

    In either case,

    $\Delta\Phi(D_i) = b_i - b_{i-1} \leq 1 - t_i$

17.4 Dynamic tables

- Dynamic tables
  - *Scenario*
    - Don't know in advance how many objects will be stored in a table.
    - 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**
    - $O(1)$ amortized time per operation
    - Unused space $\leq$ constant fraction of allocated space

17.4 Dynamic tables

- Table expansion
  - ***Load factor***
    - $\alpha = num/size \leq 1$

      where num = # items stored, $size$ = allocated size
    - If $size = 0$, then $num = 0$. Call $\alpha = 1$
    - Keep $\alpha \geq$ a constant fraction $\Rightarrow$ 2nd goal
  - When the table becomes full, double its size and reinsert all existing items

    $\Rightarrow \alpha \geq 1/2$
  - Terminology: An ***elementary insertion*** inserts an item into the table.

17.4 Dynamic tables

- Table expansion

TABLE-INSERT(T, x)
if $T.size == 0$ // Initially, $T.num = T.size = 0$
allocate $T.table$ with 1 slot
$T.size = 1$
if $T.size == T.num$
allocate $newtable$ with $2 \cdot T.size$ slots
copy $T.table$ to $newtable$ free $T.table$
$T.table = newtable$ // $T.num$ elementary insertions
$T.size = 2 \cdot T.size$
Insert x into $T.table$ // one elementary insertion
$T.num = T.num + 1$

17.4 Dynamic tables

- Table expansion: Aggregate analysis
  - Consider a sequence of n TABLE-INSERT operations
  - c_i = actual cost of i^{th} operation
    - If not full, $c_i = 1$
    - If full, $c_i = i$

      $\because$ have to copy all $i-1$ existing items and insert i^{th} item

    Thus,

$$c_i = \begin{cases} i & \text{if } i-1 = 2^k \text{ for some } k \\ 1 & \text{otherwise} \end{cases}$$

17.4 Dynamic tables

- Table expansion: Aggregate analysis
  - Total cost

$$\sum_{i=1}^{n} c_i = n + \sum_{\substack{1 \le i \le n \\ i-1=2^k}} (i-1) = n + \sum_{\substack{0 \le j < n \\ j=2^k}} j$$

$$\le n + \sum_{k=0}^{\lfloor \lg n \rfloor} 2^k \quad \because k < \lg n \Rightarrow k \le \lfloor \lg n \rfloor$$

$$= n + 2^{\lfloor \lg n \rfloor + 1} - 1$$

$$\le n + 2^{\lg n + 1} - 1 = n + 2n - 1 < 3n$$

  - Therefore, **aggregate analysis** says amortized cost per TABLE-INSERT operation < 3.

17.4 Dynamic tables

- Table expansion: 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
  - Example

Just before expansion, each x_i has \$1 of credit.

The expansion uses up all the credit.

17.4 Dynamic tables

- Table expansion: Accounting method
  - Example (Cont'd)

    Thereafter, the insertion of $x_i, 5 \leq i \leq 8$, puts \$1 of credit on x_i and \$1 of credit on x_{i-4}.

When the table is full again, every object has \$1 of credit and the table is ready for further expansion.

17.4 Dynamic tables

- Table expansion: Accounting method
  - Special case

    Charge \$2 for the insertion of x_1 

    - \$1 pays for x_1's insertion
    - \$1 pays for x_1 to be moved in the future
    - No need to pay for other item to be moved
  - This agrees with the result of **aggregate analysis**, namely,

    amortized cost per TABLE-INSERT operation

    $= \text{total cost}/\#\text{ of operations}$ $\because$ aggregate analysis

    $= (2 + 3(n-1))/n$

    < 3

17.4 Dynamic tables

- Table expansion: Potential method
  - $\Phi(T) = 2 \cdot T.num - T.size$
    - T is empty $\Rightarrow num = size = 0 \Rightarrow \Phi = 0$
    - $\alpha = num/size \geq 1/2 \Rightarrow 2 \cdot num \geq size \Rightarrow \Phi \geq 0$
  - Just before expansion

    $size = num$

    $\Rightarrow \Phi = num$

    $\Rightarrow$ have enough potential to pay for moving all items
  - Just after expansion

    $size = 2 \cdot num \Rightarrow \Phi = 0$

    $\Rightarrow$ use up the potential for the expansion

17.4 Dynamic tables

- Table expansion: Potential method

  Amortized cost of i^{th} TABLE-INSERT operation

  After i^{th} TABLE-INSERT operation, let

  $num_i = num \quad size_i = size \quad \Phi_i = \Phi$

  - Case 1: No expansion

    $size_i = size_{i-1}$

    $num_i = num_{i-1} + 1$

    $c_i = 1$

    $\widehat{c_i} = c_i + \Phi_i - \Phi_{i-1}$

    $= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$

    $= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i - 1) - size_i)$

    $= 3$

17.4 Dynamic tables

- Table expansion: Potential method
  - Case 2: Expansion

    Case 2.1: $i = 1$

    $\widehat{c_1} = c_i + \Phi_i - \Phi_{i-1} = 1 + 1 - 0 = 2$

    Case 2.2: $i > 1$

    $size_i = 2 \cdot size_{i-1}$ (N.B. $size_1 = size_0 + 1$)

    $size_{i-1} = num_{i-1} = num_i - 1$

    $c_i = num_{i-1} + 1 = num_i$

    $\widehat{c_i} = c_i + \Phi_i - \Phi_{i-1}$

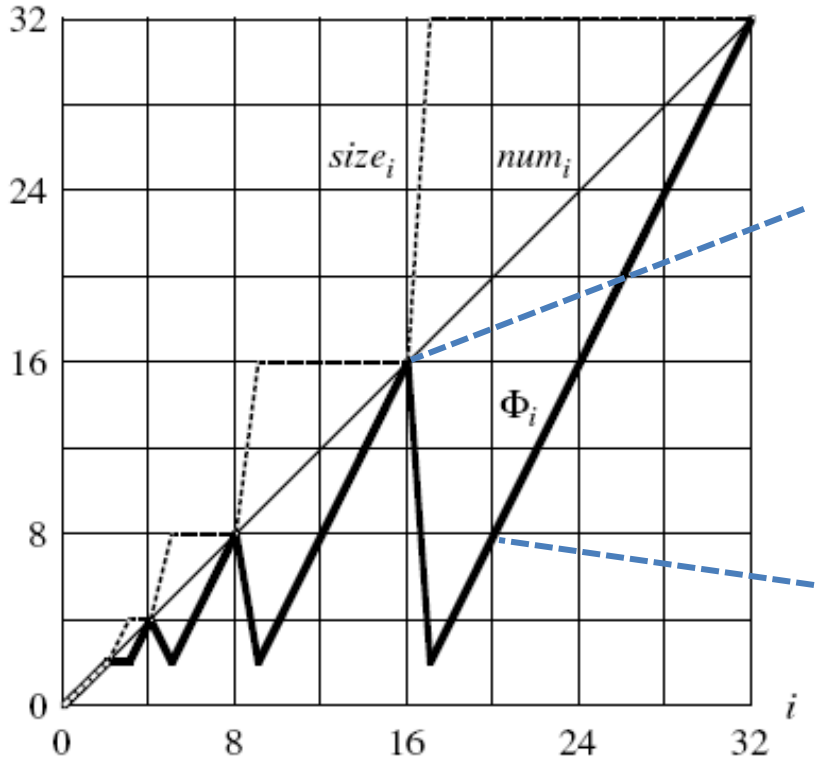
    $= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$

    $= num_i + \big(2 \cdot num_i - 2(num_i - 1)\big) - (num_i - 1)$

    $= 3$

17.4 Dynamic tables

- Table expansion: Potential method



When the table is full,

$\Phi_i = num_i$

and there is enough potential for expansion.

Afterwards, Φ_i drops to 0 and is increased by 2 for each item inserted.

17.4 Dynamic tables

- Table expansion and contraction
  - When α drops too low, contract the table
    - Allocate a new, smaller one
    - Copy all items
  - Still want
    - $\alpha \geq$ a constant fraction
    - amortized cost per operation $= O(1)$
  - "Obvious strategy"

    As before, let $\alpha \geq 1/2$

    So, halve size when deleting with $\alpha = 1/2$

    NO! After halving we have $\alpha = 1$

17.4 Dynamic tables

- Table expansion and contraction
  - Why $\alpha \geq 1/2$ doesn't work?

    Suppose the table is full, then

    insert ⇒ double; 2 deletes ⇒ halve; 2 inserts ⇒ double; …

    This is inefficient – we don't perform enough operations after expansion or contraction to pay for the next one.
  - **Solution: $\alpha \geq 1/4$**
    - Double size when inserting with $\alpha = 1$

      ⇒ after doubling, $\alpha = 1/2$
    - Halve size when deleting with $\alpha = 1/4$

      ⇒ after halving, $\alpha = 1/2$

17.4 Dynamic tables

- Table expansion and contraction
  - ***Idea***

    Make sure that we perform enough operations between consecutive expansions and contractions to build up the potential
  - $\alpha = 1/2$ potential $= 0$

    $\alpha = 1/2 \leftrightarrows 1$ potential $+2$ after each insertion

    potential -2 after each deletion

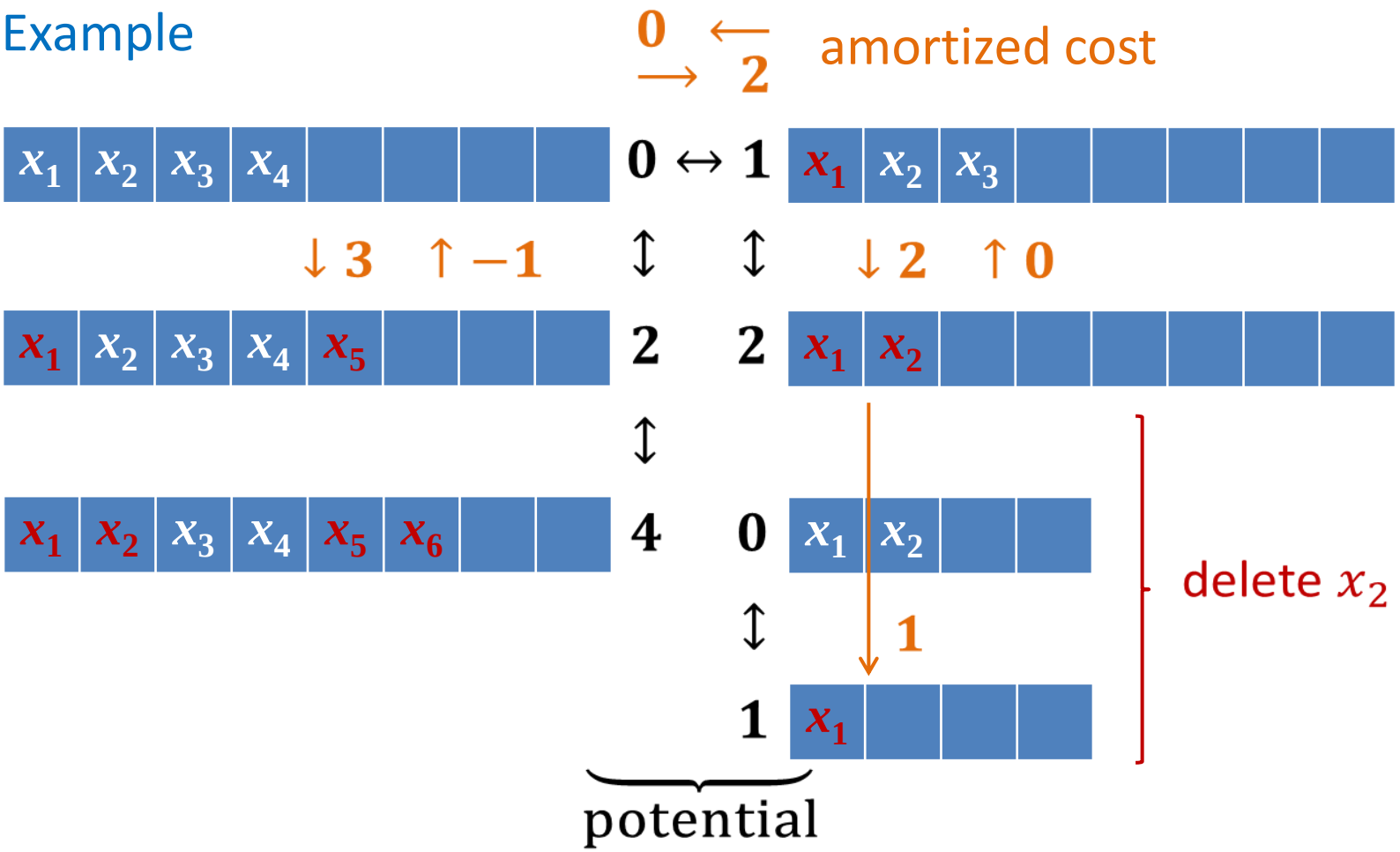
    $\alpha = 1/2 \leftrightarrows 1/4$ potential -1 after each insertion

    potential $+1$ after each deletion

17.4 Dynamic tables

- Table expansion and contraction

Example



17.4 Dynamic tables

- Table expansion and contraction: Potential method
  - Intuition
    - $\alpha = 1/2 \rightarrow 1$

      $\because num = size/2 \rightarrow size$

      $\Phi = 0 \rightarrow size$

      $\therefore \Phi$ needs to increase by 2 for each item inserted
    - $\alpha = 1/2 \rightarrow 1/4$

      $\because num = size/2 \rightarrow size/4$

      $\Phi = 0 \rightarrow size/4$

      $\therefore \Phi$ needs to increase by 1 for each item deleted
    - $\Phi = num$ when $\alpha = 1$ or $1/4 \Rightarrow$ enough potential to pay for moving all num items when we double or halve

17.4 Dynamic tables

- Table expansion and contraction: Potential method
  - Book's potential function

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size, & \alpha \geq 1/2 \\ T.size/2 - T.num & \alpha < 1/2 \end{cases}$$

    - T is empty $\Rightarrow num = size = 0 \Rightarrow \Phi = 0$
    - $\alpha \geq 1/2 \Rightarrow num \geq size/2 \Rightarrow \Phi \geq 0$
    - $\alpha < 1/2 \Rightarrow num < size/2 \Rightarrow \Phi > 0 \Rightarrow \Phi \geq 0$
  - Replacing $<$ by $\leq$ doesn't change Φ.

    In the sequel, our $\Phi(T)$ shall use $\leq$ to simplify the analysis.

17.4 Dynamic tables

- Table expansion and contraction: Potential method

  Amortized cost of i^{th} operation (may be insertion or deletion)

  Notation:

  $\alpha_i = num_i / size_i$ = load factor after i^{th} operation

  Case 1: Insertion

  - Case 1.1: $\alpha_{i-1} \geq 1/2 \Rightarrow \hat{c_1} = 2$ and $\hat{c_i} = 3, i > 1$

    Same analysis as before

  - Case 1.2: $\alpha_{i-1} < 1/2 \Rightarrow$ no expansion

    Because of its $\Phi(T)$, Book distinguishes two subcases:

    $\left.\begin{matrix} \alpha_i < 1/2 \\ \alpha_i \geq 1/2 \end{matrix}\right\}$ With our $\Phi(T)$, one case suffices: $\alpha_i \leq 1/2$

17.4 Dynamic tables

- Table expansion and contraction: Potential method
  - Case 1.2 (Cont'd)

    Moreover, Book's analysis:

    $\alpha_{i-1} < 1/2$ and $\alpha_i < 1/2 \Rightarrow \hat{c}_i = 0$
    $\alpha_{i-1} < 1/2$ and $\alpha_i \geq 1/2 \Rightarrow \hat{c}_i < 3$
    } Both $= 0$, indeed

    $\because$

    $\hat{c}_i = 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1})$

    $= 1 + (2 \cdot (num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1})$

    $= 3 \cdot num_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3$

    $= 3 \cdot \alpha_{i-1} size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3$

    $< \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 = 3$

17.4 Dynamic tables

- Table expansion and contraction: Potential method
  - Case 1.2 (Cont'd)

    A simpler analysis using our $\Phi(T)$

    Since $\alpha_{i-1} < 1/2$ and $\alpha_i \leq 1/2$, we have

$$\begin{aligned}\widehat{c_i} &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 1 - 1 = 0\end{aligned}$$

Case 2: Deletion

We shall only analyze this case using our $\Phi(T)$.

Again, the analyses are simpler than Book's.

17.4 Dynamic tables

- Table expansion and contraction: Potential method
  - Case 2.1: $\alpha_{i-1} > 1/2 \Rightarrow \alpha_i \geq 1/2 \Rightarrow$ no contraction

    $\widehat{c_i} = 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$

    $= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i + 1) - size_i)$

    $= -1$
  - Case 2.2: $\alpha_{i-1} \leq 1/2 \Rightarrow \alpha_i < 1/2$

    If no contraction

    $\widehat{c_i} = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$

    $= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1))$

    $= 2$

17.4 Dynamic tables

- Table expansion and contraction: Potential method
  - Case 2.2 (Cont'd)

    If contraction

    Since the contraction takes num_{i-1} time to delete one item and move $num_{i-1} - 1$ items, and

    $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$

    we have

    $$\begin{aligned}\widehat{c_i} &= num_{i-1} + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= num_{i-1} + (num_{i-1} - num_i) - (2 \cdot num_{i-1} - num_{i-1}) \\ &= num_{i-1} - num_i \\ &= 1\end{aligned}$$