COMP6651- Algorithm Design - Fall 2019

Lecture 6: Amortized Complexity

Professor B. Jaumard

CSE, Concordia University, Canada

October 11, 2019

Outline

Amortized Analysis

- Average cost (or amortized cost) of an operation over a sequence of operations.
- Probabilistic analysis:
  - Assume a probabilistic distribution of all possible inputs.
  - Average case running time: average over all possible inputs for one algorithm (operation).
  - The average-case complexity (expected time complexity)
- Amortized analysis:
  - No probabilities, it is a deterministic analysis.
  - Average performance on a sequence of operations, even if some operations are more expensive than others.
  - Guarantee average performance of each operation among the sequence of operations in the worst case.

Analysis Methods and Problems

- Analysis Methods
  - Aggregate Analysis
  - Accounting Method
  - Potential Method
- Three Problems:
  - Stack with MULTIPOP, can pop several objects at once.
  - Binary Counter, uses the INCREMENT operation and counts up starting from 0.
  - Dynamic tables

Aggregate Analysis: Outline

- An algorithm using a sequence of n operations takes $T(n)$ time in total for the *worst case*.
  - Average cost or amortized cost per operation is $T(n)/n$.
- Amortized cost assigns an identical amortized cost to every operation even when there are several types of operations in the sequence.
  - The two other methods may assign different amortized costs to different types of operations.

Stack Operations

Two fundamental stack operations, each of which takes $O(1)$ time:

- PUSH(S, x): pushes object x onto stack S.
- POP(S): pops the top of stack S and returns the popped object.

Let us consider the cost of each operation to be 1.

Total cost of a sequence of n PUSH and POP operations is n $\Rightarrow$ running time for n operations is $\Theta(n)$.

MULTIPOP Stack Operation

- Add the stack operation MULTIPOP(S, k)
  - removes the k top objects of stack S
  - or pops the entire stack if it contains fewer than k objects.

MULTIPOP(S, k)

while not STACK-EMPTY(S) and $k \neq 0$
  do POP(S)
    $k \leftarrow k - 1$

Running Time of the MULTIHOP Stack Operation: Analysis A

- Assuming that a POP operation costs 1 unit, then, a single MULTIPOP operation will cost:

$$\min\{k, \text{length}(S)\}$$

- Consider a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack.
  - Stack size is at most $n \Rightarrow$ the worst case of a MULTIPOP operation is $O(n)$.

  $\Rightarrow$ a sequence of n operations costs $O(n^2)$, since we could have n MULTIPOP operations costing $O(n)$ each.

Running Time of the MULTIHOP Stack Operation: Analysis B

- Analysis A overestimates the cost of n PUSH, POP, and MULTIPOP operations.
- Each object can be popped at most once for each time it is pushed.

$\Rightarrow$ number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations: at most n.

$\Rightarrow$ any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time.

$\Rightarrow$ average cost of an operation is $O(n)/n = O(1)$.

Incrementing a binary counter(1/6)

- Implementing a k-bit binary counter that counts upward from 0.
- We use an array A[0...$k-1$] of bits, where length[A] $= k$, as the counter.
- A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest order in $A[k-1]$, so that $x = \sum_{i=0}^{k-1} A[i].2^i$,
  for $x = 5 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2$ can be represented by 101.

Incrementing a binary counter(2/6)

INCREMENT(A)

```
i ← 0
while i < length[A] and A[i] = 1
   do A[i] ← 0
      i ← i + 1
if i < length[A]
   then A[i] ← 1
```

- If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into position $i + 1$ on the next iteration of the loop.
- Else ($A[i] = 0$), then adding 1 flips the bit 0 to 1 in position i.
- The cost of each INCREMENT operation is linear in the number of flipped bits.

Incrementing a binary counter(3/6)

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

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 never more than twice the total number of INCREMENT operations.

Incrementing a binary counter(4/6)

- Analysis 1
  - A single execution of INCREMENT: $\Theta(k)$ in worst case, if array A contains all 1's
  - $\Rightarrow$ a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in worst case.

Incrementing a binary counter(5/6)

- Analysis 2
  - Tightening the analysis yields a worst-case cost of $O(n)$ for a sequence of n INCREMENT operations, since not all bits flip each time INCREMENT is called.
  - A[0] flips each time INCREMENT is called, $A[1]$ flips each $\lfloor n/2 \rfloor$ times, $A[2]$ flips each $\lfloor n/4 \rfloor$ times, $A[i]$ flips each $\lfloor n/2^i \rfloor$ times, in a sequence of n INCREMENT operations on an initially zero counter.
  - Total number of flips is:

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \times \overbrace{\sum_{i=0}^{\infty} \frac{1}{2^i}}^{\textit{Geometric progression}} = 2n$$

Incrementing a binary counter (6/6)

The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$.

The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Outline of the Accounting Method

- Assign different charges to different operations, with some charges more or less than their actual charge
- Some are charged more than the actual cost
- Some are charged less than the actual cost
- The charged amount is called amortized cost
  - The objective is to simplify the complexity analysis
- Accounting method is very different from the aggregate analysis, in which all operations have the same amortized cost.

The Accounting Method

Everything must balance out at the end: for a given operation i the total of amortized charges denoted by $\hat{c}_i$ must be at least as high as the actual cost denoted by c_i:

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

Allow for an easier analysis: by over-charging for some operations that are easy to count, no need to charge anything to operations that are less easy to count.

Accounting Method: Stack Operations

Assigned costs in the previous method :
PUSH ... 1
POP ... 1
MULTIPOP ... $\min\{k, s\}$

Assign the following amortized costs:
PUSH ... 2
POP ... 0
MULTIPOP ... 0

Accounting Method: Stack Operations

Consider any sequence of n PUSH, POP and MULTIPOP operations on an initially empty stack.
Each PUSH is over-charged by 1. Thus, we "pre-pay" for the eventual POP or MULTIPOP.

- By charging the PUSH a little bit more we do not need to charge the POP operation anything.
- Moreover we do not need to charge the MULTIPOP operation anything neither.

For any sequence of n PUSH, POP and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the actual cost.

Accounting Method: Incrementing a Binary Counter

- Let the cost of setting a bit from 0 to 1 be two (2).
- $\Rightarrow$ we overcharge by 1 to set it (from 0 to 1) and we completely prepay the eventual change of the bit from 1 to 0.

INCREMENT(A)

$i \leftarrow 0$
while $i < length[A]$ and $A[i] = 1$
    do $A[i] \leftarrow 0$
        $i \leftarrow i + 1$
if $i < length[A]$
    then $A[i] \leftarrow 1$

Accounting method: Incrementing a binary counter

INCREMENT(A)

1 $i \leftarrow 0$
2 **while** $i < \mathit{length}[A]$ and $A[i] = 1$
3 **do** $A[i] \leftarrow 0$
4 $i \leftarrow i + 1$
5 **if** $i < \mathit{length}[A]$
6 **then** $A[i] \leftarrow 1$

- The cost of resetting the bits within the **while** loop is paid.
- At most one bit is set to 1 in line 6 of INCREMENT, (which is prepaid) and therefore the amortized cost of **an** INCREMENT is at most 2 dollars $= O(2)$.
- The amount of credit is never negative
- For n INCREMENT operations, the amortized cost is $O(n)$, which bounds the total cost.

The Potential Method : Principle

In the potential method of amortized analysis we represent the prepaid work as "potential energy" that can be released later for the cost of future operations.

The potential is associated with the data structure as a whole rather than with specific objects (recall the accounting method).

The Potential Method: Definition (1/2)

- Data structure D.
- $c_i =$ actual cost of the i-th operation
- $D_i =$ data structure after applying the i-th operation to D_{i-1}. Initially, we have D_0.
- A potential function Φ maps the data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i

Amortized cost $\hat{c}_i$ of the ith operation with respect to potential function Φ:

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

The Potential Method: Definition (2/2)

Total amortized cost of the n operations:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$
$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

If $\Phi(D_n) \geq \Phi(D_0)$ then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ is an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

If $\Phi(D_i) \geq \Phi(D_0)$ for all i then we guarantee, as in the accounting method, that we pay in advance.
Convenient to define $\Phi(D_0) = 0$ and then to show that $\Phi(D_i) \geq 0$.

The Potential Method : Stack Operations

- Define the potential to be the number of objects in the stack: $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$
- Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has a potential function $\Phi(D_i) \geq 0 = \Phi(D_0)$

Amortized cost $\hat{c}_i$ for a PUSH at ith operation, on a stack containing s objects:

- $\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \;\Rightarrow\; \hat{c}_i = 1 + 1 = 2$

Recall $c_i = 1$ actual cost of a PUSH.

The Potential Method : Stack Operations

Amortized cost $\hat{c}_i$ of a MULTIPOP at ith operation:

- Suppose that $k' = \min\{k, s\}$ objects are popped off the stack.
- Actual cost is k'
- $\Phi(D_i) - \Phi(D_{i-1}) = -k'$
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$

Similarly, the amortized cost of an ordinary POP operation is 0.

Incrementing a binary counter

$$b_i = \text{potential of the counter after the } i\text{th INCREMENT operation}$$
$$= \text{number of 1's in the counter after the } i\text{th operation.}$$

Amortized cost of an INCREMENT operation:

- Suppose that the ith INCREMENT resets t_i bits.
- Actual cost c_i of the operation is at most $t_i + 1$ since in addition to resetting t_i bits it sets at most one bit to 1.
- $\begin{cases} \text{if } b_i = 0, & \text{the } i\text{th operation resets all the } k \text{ bits: } b_{i-1} = t_i = k \\ \text{if } b_i > 0 & b_i = b_{i-1} - t_i + 1 \end{cases}$
- In either cases: $b_i \leq b_{i-1} - t_i + 1$

Potential difference: $\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$.

Amortized cost $\hat{c}_i$ of an INCREMENT operation:
$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) \leq 2$.

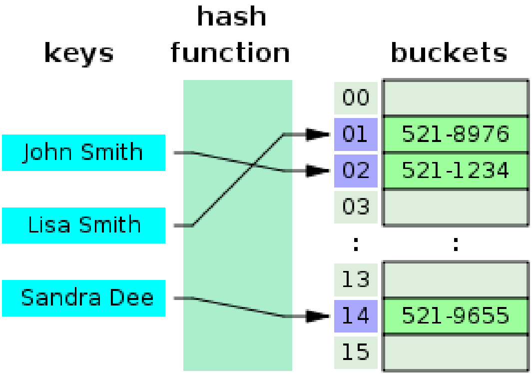
Incrementing a Binary Counter

- If the counter starts at zero: $\Phi(D_0) = 0$.
- Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost
- Amortized cost of n INCREMENT operations is $2n$.

$\Rightarrow$ an upper bound on the actual cost of n INCREMENT operations is $O(n)$.

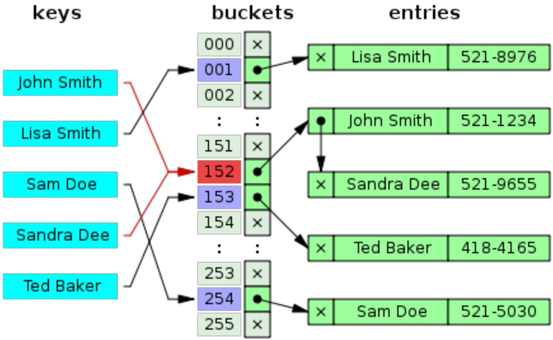
Dynamic Tables

Hash Tables



hash function
keys
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
:
13
14
15
521-8976
521-1234
:
521-9655

Hash Tables: Collision Resolution



How Large Should a Hash Table Be?

- **Goal:** Make the table as small as possible, but large enough so that it will not overflow (or otherwise it becomes inefficient)
- **Problem:** What if we do not know the proper size in advance?
- **Solution:** Dynamic tables
- **Idea:** Whenever the table overflows, "grow" it by allocating a new, larger table. Move all items from the from the old table into the new one, and free the storage for the old table.

Dynamic Table Parameters

Load Factor: NUM/ SIZE, where

- NUM = # stored items
- SIZE = allocated size

If SIZE = 0, then NUM = 0

Never allow Load Factor > 1

Keep Load Factor > constant fraction

Dynamic Tables - Table Extension

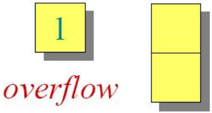
If a table T implemented as an array becomes too small, we can :

- allocate a new larger table T',
- copy the old table T into T', and
- delete T.

We call it **table expansion**.

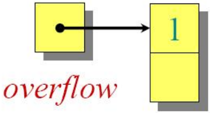
Example of a Dynamic Table

1. INSERT
2. INSERT



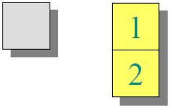
Example of a Dynamic Table

1. INSERT
2. INSERT



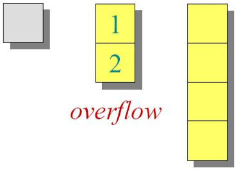
Example of a Dynamic Table

1. INSERT
2. INSERT



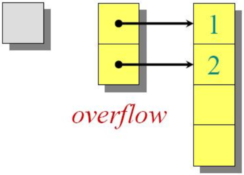
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



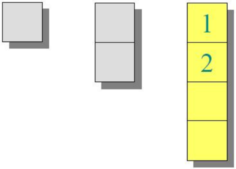
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



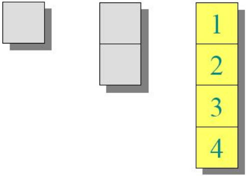
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



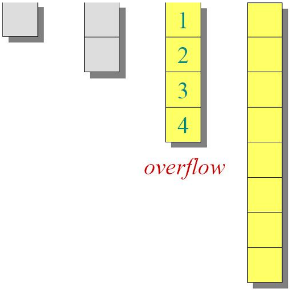
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT
4. INSERT



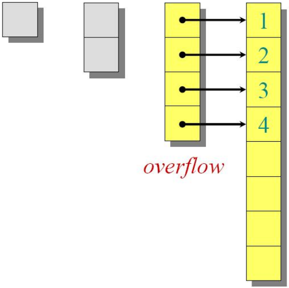
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



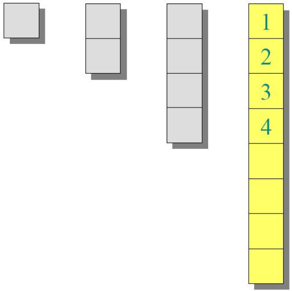
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



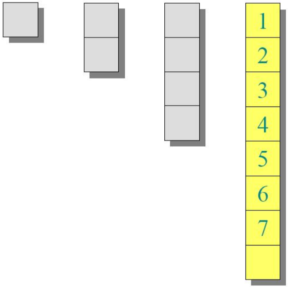
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Dynamic Tables - Table Contraction

Similarly if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size :

- allocate a new smaller table T',
- copy the old table T into T', and
- delete T.

We call it table **contraction**.

Table expansion

When an item is inserted in a table that is full, we can **expand** the table by allocating a new table with more slots than the old table.

A common heuristic:

- When the table is full, double the existing table size.
- The load factor of a table is at least $1/2$ and
- The amount of wasted space never exceeds half the total space in the table.

The **load factor** is defined as the ratio between the number of items $\text{NUM}[T]$ stored in the table and the size $\text{SIZE}[T]$ of the table.

Table expansion

TABLE-INSERT(T,x)

```
if (SIZE[T] == 0)
   then allocate table[T] with 1 slot
      SIZE[T] ← 1
if NUM[T] = SIZE[T]
   then allocate new-table of size 2 × SIZE[T]
      insert all items in table[T] into new-table
      free table[T]
      table[T] ← new-table
      size[T] ← 2 × SIZE[T]
insert x into table[T]
NUM[T] ← NUM[T] + 1
```

- We next analyze the run-time of TABLE-INSERT(T,x) in terms of the number of basic insertions.
- **Assumption.** Creation and deletion of an array: both of constant time regardless of the size of array.

Table expansion

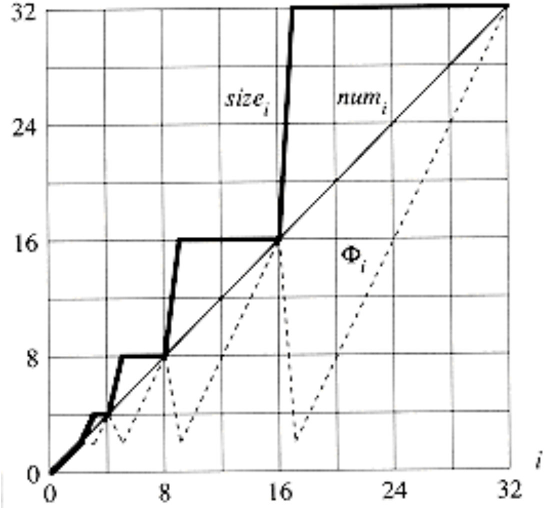


Table expansion

Cost of n TABLE-INSERT operations

Simple analysis. Copying is costly $\rightsquigarrow$ cost of one TABLE-INSERT could be n (in worst case having to expand the table when it is full) $\rightsquigarrow$ cost of n TABLE-INSERT is $O(n^2)$.

Amortized aggregate analysis. Cost c_i of the i'th insertion is:

$$c_i = \begin{cases} 1 & \text{not full} \\ i & \text{if full: have } i-1 \text{ in the table at the start of the } i\text{th operation.} \\ & \text{Have to copy all } i-1 \text{ existing items, then insert } i\text{th item } \rightsquigarrow i, \end{cases}$$

In the course of n TABLE-INSERT operations, the ith operation causes an expansion only when $i-1$ is an exact power of 2

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

Table expansion

Cost of *n* TABLE-INSERT operations

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

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j = n + \frac{2^{\lfloor \log_2 n \rfloor + 1} - 1}{2-1} \leq 3n - 1 < 3n$$

Amortized aggregate analysis. Since the total cost of n TABLE-INSERT operations is $3n$, the amortized cost of a single operation is $3 = O(1)$.

Accounting Analysis

- 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 moving another item that has already been moved once when the table expands
- Suppose we have just expanded, $\text{SIZE}[T] = m$ before next expansion, $\text{SIZE}[T] = 2m$ after expansion
- Assume that the expansion used up all the credit, so that there's no credit store after the expansion
- Will expand again after another m expansions
- Each insertion will put 1$ on one of the m items that were in the table just after expansion and will put $1 on the item inserted
- Have $2m of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion, with no credit left over!

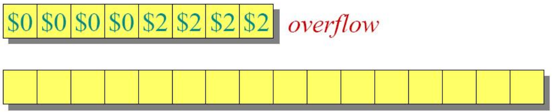
Accounting Analysis

Charge an amortized cost of $\hat{c} = \$3$ for the ith insertion

- $1 pays for the immediate insertion
- $2 is stored for later table doubling

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:

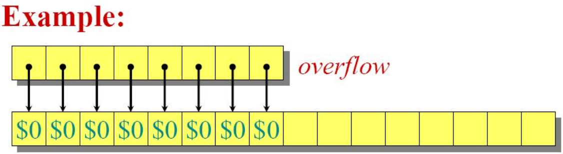


Accounting Analysis

Charge an amortized cost of $\hat{c} = \$3$ for the ith insertion

- $1 pays for the immediate insertion
- $2 is stored for later table doubling

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.



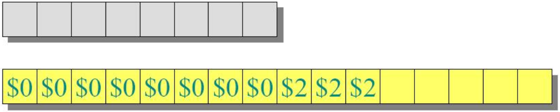
Accounting Analysis

Charge an amortized cost of $\hat{c} = \$3$ for the ith insertion

- $1 pays for the immediate insertion
- $2 is stored for later table doubling

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:



Potential function : Table expansion

Let $\Phi(T) = 2 \times \text{NUM}[T] - \text{SIZE}[T]$ such that:

- Immediately after an expansion, we have $\text{NUM}[T] = \text{SIZE}[T]/2$, and thus $\Phi(T) = 0$.
- Immediately before an expansion, we have $\text{NUM}[T] = \text{SIZE}[T]$, and thus $\Phi(T) = \text{NUM}[T]$

The initial value of $\Phi(T)$ is zero, and since the table is at least half full
$\text{NUM}[T] \geq \text{SIZE}[T]/2 \Rightarrow \Phi(T) \geq 0$

Thus sum of the **amortized costs** of n TABLE-INSERT is an **upper bound** on the sum of the **actual cost**.

Potential function : Table expansion

Let

- NUM_i = number of items stored after the *i*th operation
- SIZE_i = size of the table after the *i*th operation.
- Φ_i = potential after the *i*th operation.
- Initially: $\text{NUM}_0 = \text{SIZE}_0 = \Phi_0 = 0$

If the *i*th TABLE-INSERT does not trigger an expansion, then $\text{SIZE}_i = \text{SIZE}_{i-1}$ ($c_i = 1$) and the amortized cost $\hat{c}_i$:

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \times \text{NUM}_i - \text{SIZE}_i) - (2 \times (\text{NUM}_i - 1) - \text{SIZE}_i) \\
&= 1 - (-2) \\
&= 3
\end{aligned}$$

Potential Function: Table Expansion

If the ith TABLE-INSERT does trigger an expansion, then we have

- $\text{SIZE}_i = 2 \times \text{SIZE}_{i-1}$ and $\text{SIZE}_{i-1} = \text{NUM}_{i-1} = \text{NUM}_i - 1$
- $c_i = \text{NUM}_i$ because we have to copy the items to the new table

Amortized cost $\hat{c}_i$:

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= \text{NUM}_i + (2 \times \text{NUM}_i - \text{SIZE}_i) - (2 \times \text{NUM}_{i-1} - \text{SIZE}_{i-1}) \\
&= \text{NUM}_i + (2 \times \text{NUM}_i - 2(\text{NUM}_i - 1)) - (2(\text{NUM}_i - 1) - (\text{NUM}_i - 1)) \\
&= \text{NUM}_i + 2 - (\text{NUM}_i - 1) \\
&= 3
\end{aligned}$$

Notice how the potential builds to pay the expansion of the table.

Table expansion and contraction

Contraction:

- If the load of the table is too small, shrink it.
- We cannot use the same strategy for table contraction and table expansion:
- If we contract when the load is 1/2, we could keep on expanding and contracting repeatedly, which is very costly.

Good strategy:

- Shrink the table size to half size when the load is $1/4$.
- This way, we avoid frequent expansion and contraction within a sequence of a few instructions.
- Again, we can show that operations remain $O(1)$ using amortized analysis.

Table Expansion and Contraction