Amortized Analysis of Algorithms

Adnan YAZICI
Dept. of Computer Engineering
Middle East Technical Univ.
Ankara - TURKEY

Amortized Analysis of Algorithms

- *Worst-case analysis* is sometimes overly pessimistic.
- Amortized analysis of an algorithm involves computing the maximum total number of all operations on the various data structures.
- Amortized cost applies to each operation, even when there are several types of operations in the sequence.
- In *amortized algorithms*, time required to perform a sequence of data structure operations is averaged over all the successive operations performed. That is, a large cost of one operation is spread out over many operations (amortized), where the others are less expensive.
- Therefore, *amortized anaysis* can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though one of the single operations might be very expensive.

Amortized Analysis of Algorithms

- *Amortized time analysis* provides more accurate analysis.
- These situations arise fairly often in connection with dynamic sets and their associated operations.

- *Example*: Time needed to get a cup of coffee in a common coffee room. Once in a while, you have to start a fresh brew when you find the pot empty. It is quick in amortized sense since a long time is required only after several cups have been obtained quickly.

  Operations: - get a cup of coffee (quick)
  - brew a fresh pot (time consuming)

Amortized Analysis of Algorithms

- Amortized analysis differs from average-case analysis in that probability is not involved in amortized analysis.
- Rather than taking the *average over all possible inputs*, which requires an assumption on the probability distribution of instances, in amortized analysis we take the *average over successive calls*.
- In *amortized analysis* the times taken by the various calls are highly *dependent*, whereas in *average-case analysis* we implicitly assume that each call is *independent* from the others.

Amortized Analysis of Algorithms

- Suppose we have an ADT and we want to analyze its operation using *amortized time analysis*. Amorized time analysis is based on the following equation, which applies to each individual operation of this ADT.

  amortized cost = actual cost + accounting cost

- The creative part is to design a system of *accounting costs* for individual operations that achives the two goals:

1. In any legal sequence of operations, beginning from the creation of the ADT object being analyzed, the sum of the *accounting cost* is nonnegative.
2. Although the *actual cost* may fluctuate widely from one individual operation to the next, it is feasible to analyze the *amortized cost* of each operation.

Amortized Analysis of Algorithms

- If these two goals are achived, then the *total amortized cost* of a sequence of operations (always starting from the creation of the ADT object) is an upper bound on the *total actual cost*.
- Intuitively, the sum of the *accounting costs* is like a savings account.
- The main idea for designing a system of *accounting costs* is that "normal" individual operations should have a positive accounting cost, while the unusually expensive individual operations receive a negative accounting cost.
- Working out how big to make the positive charges to accounting costs often requires creativity, and may involve a degree of trial and error to arrive at some amount that is *reasonaly small*, yet *large enough* to prevent the "accounting balance" from going negative.

Amortized Analysis of Algorithms

There exists three common techniques used in amortized analysis:

- **Aggeregate method**
- **Accounting trick**
- **The potential function method**

Aggeregate method:

- We show that a sequence of n operations take worst-case time T(n) in total. In the worst case, the ave. cost, or amortized cost, per operation is therefore T(n) / n.
- In the aggregate method, all operations have the same amortized cost.
- The other two methods, the accounting tricky and the potential function method, may assign different amortized costs to diferent types of operations.

Amortized Analysis of Algorithms

Example: *Stack operations:*

Push(S,x): pushes object x onto stack S

Pop(S): pops the top of the stack S and returns the poped object

Multipop(S,k): Removes the k top objects of stack S

The action of Multipop on a stack S is as follows:

Multipop (S,k)

while not STACK-EMPTY(S) and k $\neq$ 0

do POP(s)

k $\leftarrow$ k –1

23			top
34			
14			
10			
22	22		
50	50	-	

- The top 4 objects are popped by Multipop(S,4), whose result is shown in second column.

Amortized Analysis of Algorithms

- The *worst-case* cost of a Multipop operation in the sequence is O(n), hence a sequence of n operations costs $O(n^2)$, (since we may have O(n) *Multipop* operations costing O(n) each and the stack size is at most n.)
- Although this analysis is correct, but not tight.
- Using the *aggregate method* of amortized analysis, we can obtain a tighter upper bound that considers the entire sequence of n operations.

Amortized Analysis of Algorithms

- In fact, although a single *Multipop* operation can be expensive, any sequence of n *Push, Pop,* and *Multipop* operations on an initially empty stack can cost at most O(n). Why?
- Because each object can be poped at most once for each time it is pushed. Therefore, the number of times that *Pop* can be called on a nonempty stack, including calls within *Multipop*, is at most the number of *Push*, which is at most n. For any value of n, any sequence of n *Push*, *Pop*, and *Multipop* operations takes a total of O(n) time.
- The amortized cost of an operation is the average: O(n)/n = O(1).

Accounting trick

- Different charges to different operations are assigned. Some operations are charged more or less than they actually cost.
- When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as *credit*.
- *Credit* can be used later on to help pay for operations whose amortized cost is less than their actual cost.
- One must choose the amortized costs of operations carefully. The *total credit* in the data structure should never become negative, otherwise the total amortized cost would not be an upper bound on the total actual cost.

Amortized Analysis of Algorithms

Example-1: *stack operations:*

The actual costs of the operations were,

Push	1,
Pop	1,
Multipop	min(k,s),

- where *k* is the argument supplied to *Multipop* and *s* is the stack size when it is called.

We assign the following amortized costs:

Push	2,
Pop	0,
Multipop	0.

- Here all three amortized costs are O(1), although in general the amortized costs of the operations under consideration may differ asymptotically.

Amortized Analysis of Algorithms

- We shall now show that we can pay for any sequence of stack operations by charging the amortized costs.
  - For *Push* operation we pay the actual cost of the push 1 token and are left with a credit of 1 token out of 2 tokens charged, which we put on top of the plate.
  - When we execute a *Pop* operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. Thus, by charging the *Push* operation a little bit more, we needn't charge the *Pop* operation anything.
  - We needn't charge the *Multipop* operation anything either. We have always charged at least enough up front to pay for the *Multipop* operations.
- Thus, 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 *total actual cost*.

Amortized Analysis of Algorithms

- **Example -2:** *Accounting scheme for Stack with array doubling:*
  - Say the actual cost of *push* or *pop* is 1 when no resizing of the array occurs, and
  - The actual cost of *push* is 1 + nt, for some constant t, if it involves doubling the array size from n to 2n and copying n elements over the new array.
  - So, the worst-case actual time for *push* is Θ(n). However, the amortized analysis gives a more accurate picture.
    - The accounting cost for a *push* that does not require array doubling is 2t,
    - The accounting cost for a *push* that requires doubling the array from n to 2n is – nt + 2t,
    - Pop is 0.

Amortized Analysis of Algorithms

- The coefficient of 2 in the accounting costs is chosen to be large enough, from the time the stack is created, the sum of the accounting costs can never be negative. To see this informally, when the account balance – net sum of accounting costs - grows to 2nt (doubling occurs from size n to 2n), then the first negaive charge will reduce it to nt + 2t. Therefore, this is a valid accounting scheme for the Stack ADT.
- With some experimentation we can convince ourselves that any coefficient less than 2 will lead to eventual bankruptcy in the worst case.
- Amortized cost = actual cost + accounting cost = 1 + nt + (-nt + 2t) = 1 + 2t.
- With this accounting scheme, the amortized cost of each individual push operation is 1 + 2t, whether it causes array doubling or not and the amortized cost of each pop operation is 1. Thus we can say that both push and pop run in the worst-case amortized time that is in $\Theta(1)$.
- More complicated data structures often require more complicated accounting schemes, which require more creativity to think up.

Amortized Analysis of Algorithms

- **The potential function method**
  - The potential is associated with the data structure as a whole rather than with specific objects within the data structure.
  - The potential method works as follows:
    - We start with an initial data structure D_0 on which n operations are performed.
    - For each i = 1,2,...,n, we let c_i be the actual cost of the i^{th} operation and D_i be the data structure that results after applying the i^{th} operation on data structure D_{i-1}.

Amortized Analysis of Algorithms

- A potential function Φ maps each data structure D_i to a real number.
- $\Phi(D_i)$ is potential associated with data structure D_i.
- The amortized cost ac_i of the i^{th} operation with respect to *potential function* Φ is defined by

  $$ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

- The *amortized cost* of each operation is therefore its actual cost(c_i) plus the increase in potential ($\Phi(D_i) - \Phi(D_{i-1})$) caused by i^{th} operation.
- So, the total amortized cost of the n operations is

  $$\sum_{1\leq i\leq n} ac_i = \sum_{1\leq i\leq n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$
  $$= \sum_{1\leq i\leq n} c_i + \Phi(D_n) - \Phi(D_0).$$

- Here we used telescoping series;

  for any sequence $a_0, a_1, ..., a_n$, $\sum_{1\leq k\leq n} (a_k - a_{k-1}) = (a_n - a_0)$.

Amortized Analysis of Algorithms

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

- If we can define a *potential function* Φ so that $\Phi(D_n) \ge \Phi(D_0)$, then the *total amortized cost,* $\sum_{1 \le i \le n} ac_i$, is an upper bound on the *total actual cost* needed to perform a sequence of operations.
- It is often convenient to define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \ge 0$, $\forall i$.
- The challenge in applying this technique is to figure out the *proper potential function*.
- Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs.

Amortized Analysis of Algorithms

Example: Suppose that the process to be analysed modifies a database and its efficiency each time it is called depends on the current state of that database. We associate a notion of "cleanliness", known as the *potential function* of the database.

Formally, we introduce the following parameters:

- Φ: an integer-valued potential function of the state of the database. Larger values of Φ correspond to dirtier states.
- Φ_0: the value of Φ on the initial state; it represents our standard of cleanliness.
- Φ_i: the value of Φ on the database after the i^th call on the process, and
- c_i: the *actual time* needed by that call.
- ac_i: the *amortized time*, which is *actual time* (required to carry out the i^th call on the process *plus* the *increase in potential* caused by that call.

Amortized Analysis of Algorithms

- So, the amortized time taken by that call is:

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

- Let T_n denote the total time required for the first n calls on the process, and denote the total amortized time by aT_n.

  $aT_n = \Sigma_{1\leq i\leq n}\, ac_i = \Sigma_{1\leq i\leq n}\, (c_i + \Phi_i - \Phi_{i-1}) = \Sigma_{1\leq i\leq n}\, c_i + \Sigma_{1\leq i\leq n}\, \Phi_i - \Sigma_{1\leq i\leq n}\, \Phi_{i-1}$

  $= T_n + \Phi_n + \Phi_{n-1} +.. \Phi_1 - \Phi_{n-1} - ...- \Phi_1 - \Phi_0$

  $= T_n + \Phi_n - \Phi_0$

  Therefore, $aT_n = T_n + (\Phi_n - \Phi_0)$.

- The significance of this is that $T_n \leq aT_n$ holds for all n provided Φ_n never becomes smaller than Φ_0. In other words, the total amortized time is always an upper bound on the total cost actual time needed to perform a sequence of operations, as long as the database is never allowed to become “cleaner” than it was initially.
- This shows that overcleaning can be harmful!!
- This approach is interesting when the actual time varies significantly from one call to the next, whereas the amortized time is nearly invarient.

Amortized Analysis of Algorithms

Example: *stack operations:*

- We define Φ on a stack to be the number of objects in the stack.
- The stack D_i that results after the i^{th} operation has nonnegative potential, since the number of objects in the stack is never negative. Thus,

  $$\Phi(D_i) \geq 0 = \Phi(D_0).$$

- The total amortized cost of n operations w.r.t Φ therefore represents an upper bound on the actual cost.

Amortized Analysis of Algorithms

The amortized costs of the various stack operations are as follows:

- If the i^{th} operation on a stack containing s objects is a *Push* operation, then the potential difference is

  $\Phi(D_i) - \Phi(D_{i-1}) = (s+1) – s = 1.$

- The amortized cost of this *Push* operation is

  $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$

- If i^{th} operation is *Pop* on the stack containing an object that is popped off the stack. The actual cost of the *Pop* operation is 1, and the potential difference is

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

  Thus, the amortized cost of this *Pop* operation is

  $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 – 1 = 0.$

Amortized Analysis of Algorithms

- Therefore, the amortized cost of the each of the three operations is O(1), and thus the total amortized cost of a sequence of n operations is O(n).
- Suppose that i^{th} operation on the stack is *Multipop(S,k)* and k = min(k,s) objects are popped off the stack. The actual cost of the operation is k, and the potential difference is

  $\Phi(D_i) - \Phi(D_{i-1}) = -k.$
- Thus, the amortized cost of this *Multipop* operation is

  $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k – k = 0.$

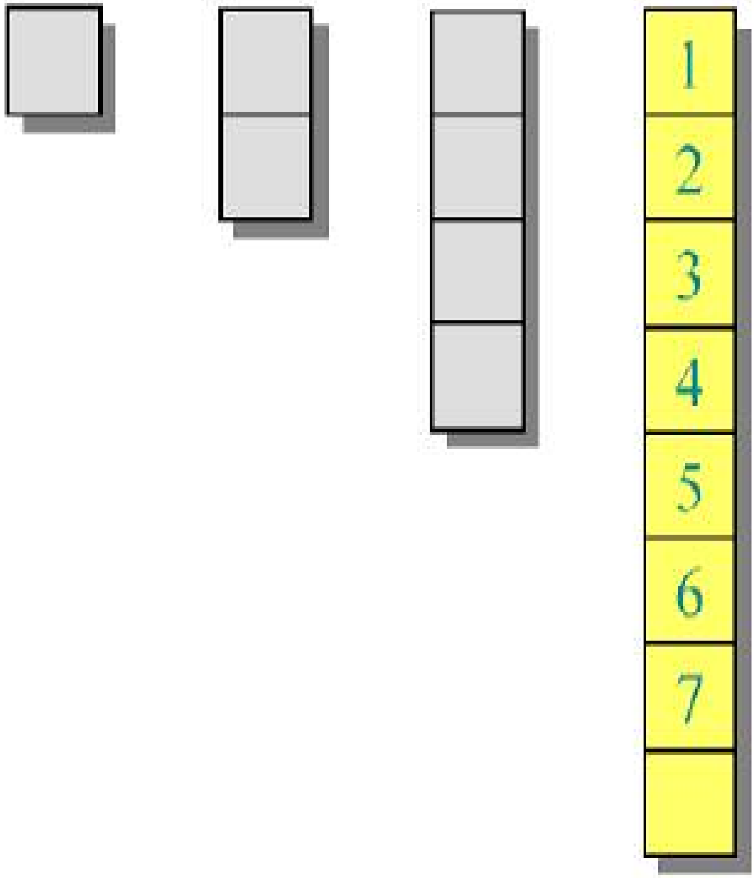
How large should a hash table be?

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

Amortized Analysis of Algorithms

Example:

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



Worst-case analysis

- Consider a sequence of n insertions. The worst-case time to execute one insertion is $O(n)$. Therefore, the worst-case time for n insertions is n.

$$n \cdot O(n) = O(n^2).$$

- **WRONG!** In fact, the worst-case cost for n insertions is only $O(n) \ll O(n^2)$.
- Let's see why.

Amortized Analysis of Algorithms

- If we analyze a sequence of n *Table-Insert* operations on an initially empty table, what is the actual cost c_i of the i^{th} operation?

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise (if there is room in the current table)} \end{cases}$$

- The total cost of *n Table-Insert* operations is; therefore,

$$\sum_{1\le i\le n} c_i \le n + \sum_{0\le j\le \lfloor \lg n \rfloor} 2^j < n + 2n = 3n$$

- there are at most n operations that cost 1 and the costs of the remaining operations for a geometric series. Since the total cost of n operations is 3n, the amortized cost of a single operation is 3.

Tighter analysis

Let c_i = the cost of the i th insertion

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

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

$$\text{Cost of } n \text{ insertions} = \sum_{i=1}^{n} c_i$$

$$\leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j$$

$$\leq 3n$$

$$= \Theta(n).$$

Thus, the average cost of each dynamic-table operation is $\Theta(n)/n = \Theta(1)$.

Amortized Analysis of Algorithms

IDEA: View the bank account as the potential energy (*à la* physics) of the dynamic set.

Framework:

- Start with an initial data structure D_0.
- Operation i transforms D_{i-1} to D_i.
- The cost of operation i is c_i.
- Define a ***potential function*** $\Phi : \{D_i\} \rightarrow \mathbb{R}$, such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i.
- The ***amortized cost*** $\hat{c}_i$ with respect to Φ is defined to be $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Understanding potentials

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i.

The amortized costs bound the true costs

The total amortized cost of n operations is

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

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

$$\geq \sum_{i=1}^{n} c_i$$ since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$.

Example: *Dynamic Tables*

- Assume that T is an object representing the table.
- The field *table[T]* contains a pointer to the block of storage representing the table.
- The field *num[T]* contains the number of items in the table
- The field *size[T]* is the total number of slots in the table.
- Initially, the table is empty: *num[T]* = *size[T]* = 0.

Amortized Analysis of Algorithms

Dynamic Tables

Table insertion: If only insertions are performed, the load factor of a table is always at least ½, thus the amount of wasted space never exceeds half the total space in the table.

```
Table-Insert (T, x)
If size [T] = 0
    Then allocate table[T] with 1 slot
If num[T] = size [T]
    Then allocate new-table with 2*size[T] slots
        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
```

Amortized Analysis of Algorithms

- To use the *potential function* method to analyze a sequence of n *Table-Insert* operations, we start by defining a *potential function* Φ that is 0 immediately after an expansion, but builds to the table size by the time the table is full, so that the next expansion can be paid for by the potential.
- The potential function $\Phi(T) = 2*num[T] - size[T]$ is one possibility.
  - Immediately after the expansion, we have $num[T] = size[T]/2$, and thus $\Phi(T)$ *is 0* (as desired).
  - Immediately before the expansion, we have $num[T]=size[T]$, thus $\Phi(T)=num[T]$, thus the potential can pay for an expansion if an item is inserted (as desired).
  - The inial value of the potential is 0, since the table is always at least half full, $num[T] \geq size[T]$, which imples that $\Phi(T)$ is always nonnegative. Thus, the sum of the amortized costs of n *Table-Insert* operations is an upper bound on the sum of the actual costs (as desired).

Amortized Analysis of Algorithms

- If the i^{th} *Table-Insert* operation does not trigger an expansion, then $size_i = size_{i-1}$ and the amortized cost of the operation is

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

  $= 1 + (2*num_i – size_i) – (2*num_{i-1} – size_{i-1})$

  $= 1 + (2*num_i – size_i) – (2*(num_i – 1) - size_i)$

  $= 3.$

- If the i^{th} *Table-Insert* operation does trigger an expansion, then $size_i / 2 = size_{i-1} = num_i – 1$ and the amortized cost of the operation is

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

  $= num_i + (2*num_i – size_i) – (2*num_{i-1} – size_{i-1})$

  $= num_i + (2*num_i–(2*num_i–2)) – (2*(num_i – 1)–(num_i–1))$

  $= num_i + 2 - (num_i–1)$

  $= 3.$

Amortized Analysis of Algorithms

Table expansion and contraction:

- The improvement on the natural strategy for expansion and contraction (doubling the table size for both cases which may result an immediate expansion and contraction on the table size whose n sequence of them would be $\Theta(n^2)$, where amortized cost of an operation would be $\Theta(n)$) is to allow the load factor of the table to drop below ½.
- The load factor, denoted as $\alpha(T)$, is the no. of items stored in the table divided by the size (no. of slots) of the table; that is,

  $\alpha(T) = num[T] / size[T]$.
- Specifically, we continue to double the table size when an item is inserted into a full table, but halve the table size when a deletion causes the table to become less than ¼ full rather than ½ full as before.

Amortized Analysis of Algorithms

- We can now use the potential method to analyze the cost of a sequence of n *Table-Insert* and n *Table-delete* operations.
- We start by defining a potential function Φ that is 0 immediately after an expansion or contraction and builds as the load factor increases to 1 or decreases to ¼.
- We use the potential function as
- $$\Phi(T) = \begin{cases} 2*num[T] - size[T] & \text{if } \alpha(T) \geq 1/2, \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2. \end{cases}$$

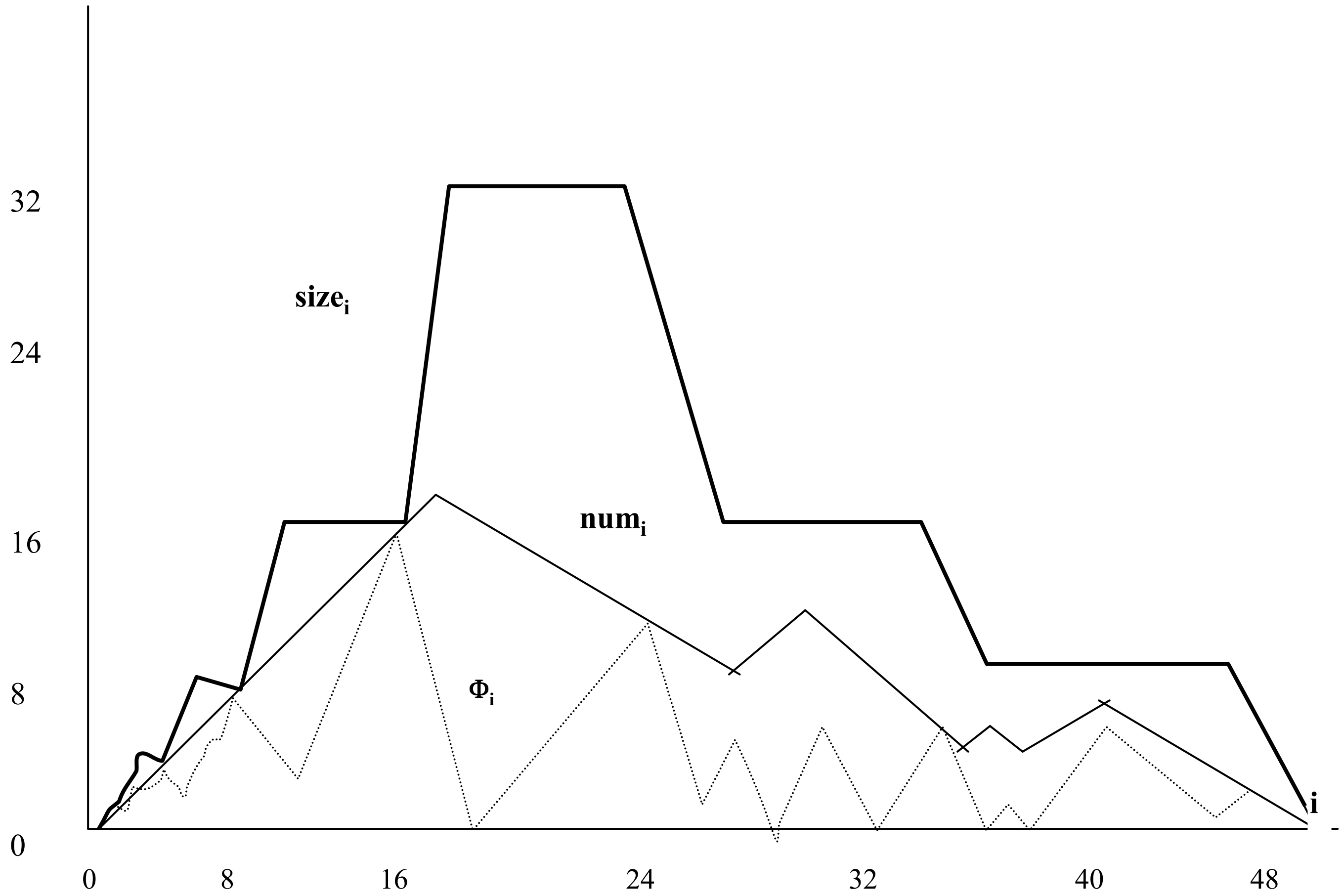
Amortized Analysis of Algorithms

$$\Phi(T) = \begin{cases} 2*num[T] - size[T] & \text{if } \alpha(T) \geq ½, \\ size[T]/2 - num[T] & \text{if } \alpha(T) < ½. \end{cases}$$

- Observe that when the load factor is ½, the potential is 0 (since we have $num[T] = size[T]/2$, and thus $\Phi(T)$ *is* 0 (as desired)).
- When $\alpha(T)$ is 1, we have $num[T] = size[T]$, which implies $\Phi(T) = num[T]$, thus the potential can pay for an expansion if an item is inserted (as desired).
- When the load factor is $1/4$, we have $size[T] = 4*num[T]$, which implies $\Phi(T) = num[T]$, thus the potential can pay for an contraction if an item is deleted (as desired).
- Observe that the potential of an empty table is 0 and the potential is never negative. Thus, the total amortized cost of a sequence of operations w.r.t Φ is an upper bound on their actual cost (as desired).

Amortized Analysis of Algorithms

The figure below illustrates how the potential behaves for a sequence of operations.



Amortized Analysis of Algorithms

- Initially, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$, and $\Phi_0 = 0$.
- We start with the case in which the i^{th} operation is *Table-Insert*.
- If $\alpha_{i-1} \geq ½$, the analysis is identical to that for table expansion before, whether the table expands or not, the amortized cost, ac_i, of the *Table-insert* operation is at most 3.
- If $\alpha_{i-1} < ½$, the table cannot expand as a result of the operation, since expansion occurs only when $\alpha_{i-1}=1$. If $\alpha_i < ½$ as well, then the amortized cost of the i^{th} operation is

$$ac_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) = 0.$$

Since $size_i = size_{i-1}$ and $num_{i-1} = num_i - 1$.

Amortized Analysis of Algorithms

• If $\alpha_{i-1} < ½$ but $\alpha_i \geq ½$, then

$$ac_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + (2*num_i - size_i) - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + (2*(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1})$$
$$= 3*num_{i-1} - 3/2 size_{i-1} + 3$$
$$= 3*\alpha_{i-1}*size_{i-1} - 3/2 size_{i-1} + 3$$
$$< 3/2*size_{i-1} - 3/2 size_{i-1} + 3 = 3.$$

Since $size_i = size_{i-1}$, $num_{i-1} + 1 = num_i$, and $\alpha_{i-1} = num_{i-1}/size_{i-1}$.

• Thus, the amortized cost of a Table-insert operation is at most 3.

Amortized Analysis of Algorithms

We now turn to the case in which the i[th] operation is *Table-delete*.

• In this case, $num_i = num_i - 1$. If $\alpha_{i-1} < ½$, then we must consider whether the *Table-delete* operation causes a contraction.

• If it does not, then $size_i = size_{i-1}$ and the amortized cost of the operation is

$$ac_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + (size_i /2 - num_i) - (size_{i-1} /2 - num_{i-1})$$
$$= 1 + (size_i /2 - num_i) - (size_i /2 - num_i + 1).$$
$$= 2.$$

Amortized Analysis of Algorithms

- If $\alpha_{i-1} < ½$ and the i[th] operation does trigger a contraction, then the actual cost of the operation is $c_i = num_i + 1$, since we delete one item and move num_i items.
- We have $size_i /2 = size_{i-1}/4 = num_i + 1$, and the amortized cost of the operation is

$$
\begin{aligned}
ac_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i /2 - num_i) – (size_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) – \\
&\qquad ((2*num_i + 2) - (num_i + 1)) \\
&= 1.
\end{aligned}
$$

Amortized Analysis of Algorithms

• When the *i*th operation is a *Table-delete* and $\alpha_{i-1} \geq ½$, the amortized cost is also bounded above by a constant.

In summary, since the *amortized cost* of each operation is bounded above by a constant, the actual time for any sequence of n operations on a *dynamic table* is *O(n)*.

Conclusions

• Amortized costs can provide a clean abstraction of data-structure performance.

• Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest.

• Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.