Amortized Analysis

Definition

- The time required to perform a sequence of data structure operations is averaged over all the operations performed.
- Average performance of each operation in the worst case.
- For all n, a sequence of n operations takes worst time $T(n)$ in total. The amortize cost of each operation is T(n)/n
- No probability is involved
- An amortized analysis guarantees the average performance of each operation in the worst case.

Three Techniques

- aggregate analysis
- accounting method
- potential method

The aggregate analysis

- Stack operation

  - PUSH(S, x)

  - POP(S)

  - MULTIPOP(S, k)

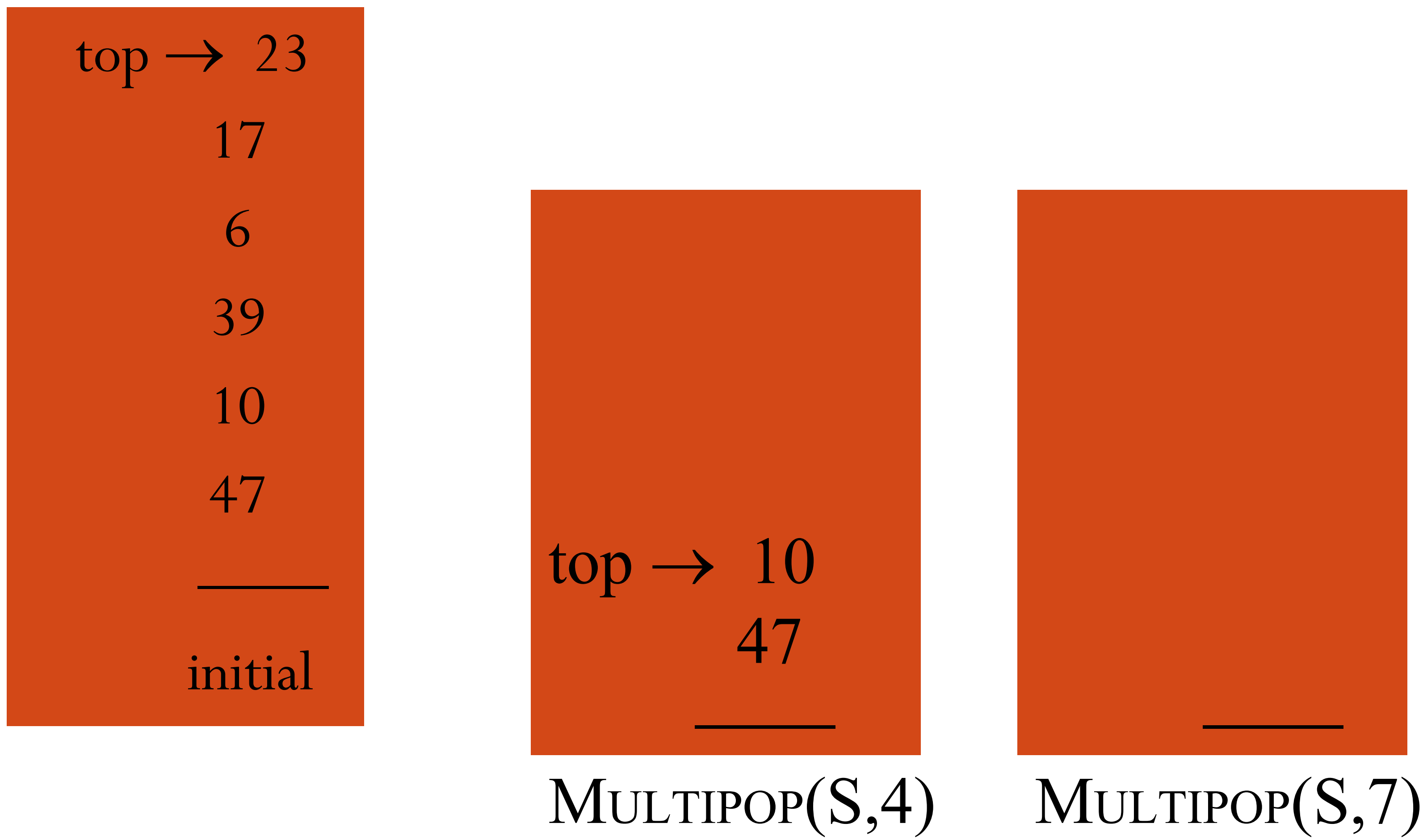
MULTIPOP(S, k)

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

  do POP(S)

    $k \leftarrow k - 1$

Action of MULTIPOP on a stack S



- PUSH(S, x) / POP(S), each runs in O(1) time.
- Actual running time for a sequence of n PUSH,POP operations is $\Theta(n)$.
- MULTIPOP(S, k), pops the top k objects of stack

  Total cost : min(s, k) s : stack size

- Analyze a sequence of n PUSH, POP, and
- MULTIPOP operation on an initially empty stack.

PUSH : O(1)

POP : O(1)

MULTIPOP : O(n) (the stack size is at most n)

- Total cost of n operations: O(n*n)
- We can get a better bound. (see the next)

- Each object can be popped at most once for each time it is pushed.
- Number of times the POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operatons, which is at most n.
- Total cost of any seq of n operations: O(n), better bound !
- The amortized cost of an operation is O(n)/n= O(1)

- Analysis a sequence of n PUSH, POP, and MULTIPOP operation on an **initially empty stack**.
- $O(n^2)$
- $O(n)$ (better bound)
- The amortize cost of an operation is $O(n)/n=1$

.

INCREMENT

```
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
```

Incremental of a binary counter

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

Analysis:

- O(n k) (k is the word length)
- Amortize Analysis:

$$\sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$= 2n$$

$$\Rightarrow \text{the amortize cost is } \frac{O(n)}{n} = O(1)$$

The accounting method

- We assign different charges to different operations, with some operations charged more or less than the actually cost. The amount we charge an operation is called its **amortized cost**.
- When an operation's amortized cost exceeds its actually cost, the difference is assign to specific object 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.
- If we want analysis with amortized costs to show that in the worst case the average cost per operation is small, the total amortized cost of a sequence of operations must be an **upper bound** on the total actual cost of the sequence.

- If the amortized cost > actual cost, the difference is treated as credit.
- Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost.

- If we denote the actual cost of the ith operation by c_i and the amortized cost of the ith operation by $\hat{c}_i$, we require

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

for all sequence of n operations.

- The total credit stored in the data structure is the difference between the total actual cost, or

.

$$\sum_{i=1}^{n} \hat{C}_i - \sum_{i=1}^{n} C_i$$

Stack operation

PUSH	1	PUSH	2
POP	1	POP	0
MULTIPOP	$\min\{k,s\}$	MULTIPOP	0

Amortize cost: O(1)

Incrementing a binary counter

$0 \rightarrow 1$	1	$0 \rightarrow 1$	2
$1 \rightarrow 0$	1	$1 \rightarrow 0$	0

- Each time, there is exactly one 0 that is changed into 1.
- The number of 1's in the counter is never negative!
- Amortized cost is at most $2 = O(1)$.

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.
- Can release potential to pay for future operations.
- Most flexible of the amortized analysis methods.

- D_0: initial data structure
- D_i: the data structure of the result after applying the *i*-th operation to the data structure D_{i-1}.
- C_i: actual cost of the *i*-th operation.
- $\hat{c}_i$: amotized cost of the *i*-th operation.

- A potential function Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i.
- The amortized cost $\hat{c}_i$ of the i-th operation with respect to potential Φ is defined by $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

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

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

Dynamic tables (Tutorial-I)

- A nice use of amortized analysis.
- Scenario
  - Have a table—maybe a hash table.
  - Don't know in advance how many objects will be stored in it.
  - 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.
- Details of table organization not important.

Table expansion

- Consider only TABLE-INSERT
- TABLE-DELETE (discuss later)
- Each time we actually insert an item into the table, it's an *elementary insertion*.

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 with 2 · size[T] slots
        insert all items in table[T] in new-table
        free table[T]
        table[T] ← new-table
        size[T] ← 2 · size[T]
insert x into table[T]
    num[T] ← num[T] + 1
```

Item No.	1	2	3	4	5	6	7	8	9	10	......
Table Size	1	2	4	4	8	8	8	8	16	16	......
Cost	1	2	3	1	5	1	1	1	9	1	......

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1\ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1\ldots)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + \ldots)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

Aggregate method:

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

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

Analysis

- *Running time:* Charge 1 per elementary insertion. Count only elementary insertions, since all other costs together are constant per call.
- c_i = actual cost of ith operation
  - If not full, $c_i = 1$.
  - If full, have i - 1 items in the table at the start of the ith operation. Have to copy all i - 1 existing items, then insert ith item $\Rightarrow c_i = i$.
- n operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for n operations. (?)

- amortized cost = 3