CS 6043 – Fall 2025
Lecture Notes
Amortized Analysis

1 Toy Example: stack with multi-pop (Chapters 17.1-17.3 in CLRS 3rd edition)

1.1 basic stack

Definition 1 (Stack)**.** A stack is a data structure S that supports

- S.push(x) – add element x to S
- S.pop() – remove most recently added element from S

Implementation : Easy to get $O(1)$ time per operation. So stacks have *worst-case* update time $O(1)$ per operation.

1.2 multi-pop stack

Definition 2 (multi-pop)**.** We now want a stack S that also supports

- S.multi-pop(k) – remove the k most recently pushed elements

Implementation: Will implement S.multi-pop(k) by just doing S.pop() for k times.

worst-case update time: It is no longer the case that every update takes $O(1)$ time, because S.multipop(k) takes $O(k)$ time. So *some* update-operations are expensive. But the average update time is still small.

1.3 Definining amortized analysis

Definition 3. worst-case vs. amortized Let D be a data structure that supports some updates. We say that D has worst-case update time T if *every* update takes time $O(D)$.

We say that D has *amortized* update time T, if, starting with an empty data structure, for any sequence of n updates, the *total* time spent by the data structure os $O(n \cdot T)$.

Remark: Note that amortized update time still has to work for all update sequences. So small amortized does NOT mean "for most update sequences the data structure is fast". Instead means "for *all* update sequences, the average update in the sequence is fast."

1.4 Amortized Analysis of multi-pop

1.4.1 Aggregate Analysis

Claim 1.1. *Amortized analysis of stack with multi-pop is* $O(1)$

Proof.
- each multi-pop is implemented as a sequence of regular pop operations.
- total # pops $\leq$ total # pushes
- So over n updates, do a total of $\leq n$ pushes and $\leq n$ pops
- So total time is $O(n)$.

□

1.4.2 Accounting Analysis

I will use \$ to denote time units. Every time the data structure is updated, it needs to spend \$ equal to the cost of the operation.

Defining the real costs c_i . I will always use c_i to defer the real cost of the ith operation in the data structure. So in the case of multi-pop stack:

- If operation i is a push then $c_i = 1$
- If operation i is a Multipop(k) then $c_i = k$

The Banks perspective : Let's say that for the ith operation, I put some money $\hat{c}_i$ in the data structure (I get to choose $\hat{c}_i$). Now, crucially, let's say I'm able to prove that the data structure is able to pay for all the operations using only money from the bank. This clearly implies that $\sum c_i \leq \sum \hat{c}_i$. The idea is then that since I chose the $\hat{c}_i$, it will be easier for me to handle on what $\sum \hat{c}_i$ is equal to, which will in turn upper bound $\sum c_i$.

Example of bank with multi-pop I'm going to define $\hat{c}_i = 2$ for push operations and $\hat{c}_i = 0$ for other operations. In other words, whenever there is a S.push(x) operation, I will add \$2 into the system.

Claim 1.2. *Given any sequence of operations on D, there is always enough money from the bank to pay for them.*

Proof. Since I add \$1 for every S.push(x), I can spend \$1 to pay for the actual push operation. Then I will put the remaining \$1 in my bank, attached to the element x. Now, when I do S.multipop(k), I remove k elements from the stack. But each of these elements has \$1 attached, so I have \$k total attached to the elements being removed, which is precisely enough to pay for the multipop. □

Final Analysis: I thus have that $\sum_{i=1}^{t} c_i \leq \sum_{i=1}^{c} \hat{c}_i \leq 2t$. So the amortized update time as is most $2t/2 = 2 = O(1)$.

1.5 Potential Function Analysis

Motivation: The above analysis requires us to keep track of amount of \$ in the bank, which requires keeping track of all updates so far. This gets messy for more complicated problems. Instead, I want to use a single function ϕ to represent the amount of money in the bank.

Definition 4. Given a data structure D with some update operations and implementation, we will define $\Phi : D \to R$ to be a function that, given any possible state of the data structure, returns some number.

Intuitively, you can think of $\Phi(D)$ as the amount of \$ in the bank for that particular state of the data structure. We will always maintain the invariant that $\phi(D)$ is a non-negative integer.

Remark 1.3. *The algorithm never computes $\Phi(D)$. It is used purely for analytic purposes.*

Example for stack with multi-pop Given any current state S of the stack, we set $\Phi(S) = \#$ elements in S.

- Note that initially S is empty and $\Phi(S) = 0$
- Note that at all times $\Phi(S) \geq 0$.

1.5.1 How to use potential function

As in the accounting analysis, we want $\hat{c}_i$ to be equal to the number of new money that enters the system. But now, the value of $\hat{c}_i$ is determine by Φ. Recall that we think of $\Phi(D_i)$ as the amount of money in the bank after i updates, then how much do we need to contribute for the ith update. Well, we need:

$$c_i + \text{change in bank} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Definition 5. Define $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Recall that $\sum c_i$ is the total amount of dollars (i.e. time units) that we spend executing updates. But since $\hat{c}_i$ corresponds to the *new* dollars we need to contribution for operation i, $\sum \hat{c}_i$ is also an upper bound on the total money spent by the data structure. We now formalize this.

Lemma 1.4. *Over a sequence of n updates, $\sum \hat{c}_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$*

Proof. $\sum \hat{c}_i = \sum_{i=1}^{n} [c_i + \Phi(D_i) - \Phi(D_{i-1})] = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} [\Phi(D_i) - \Phi(D_{i-1})] = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$ □

Corollary 1.5. *If $\Phi(D_0) = 0$ and $\Phi(D) \geq 0$ for all states of the data structure, then $\sum c_i \leq \sum \hat{c}_i$. As a result, if $\hat{c}_i \leq T$ for all i, then the data structure has amortized update time $\leq T$.*

Analysis of multi-pop stack with potential function

- Recall: $\Phi(D) = \#$ elements in stack.
- If operation i is a push, $\hat{c}_i = c_1 + [\Phi(D_i) - \Phi(D_{i-1})] = 1 + 1 = 2$
- If operation i is a multi-pop(k) then $\hat{c}_i = c_1 + [\Phi(D_i) - \Phi(D_{i-1})] = k + (-k) = 0$
- So always have $\hat{c}_i \leq 2$, so over n updates, $\sum c_i \leq \sum \hat{c}_i \leq 2n$
- So amortized update time is 2.

1.5.2 Being careful with constants

We said that $c_i = 1$ if update i is a push. But this is not quite true – really it's $O(1)$. Formally:

- For push: $c_i = O(1)$
- For multi-pop(k): $c_i = O(k)$

By definition of big-O, we can rewrite this as: there exists a constant C such that

- For push: $c_i \leq C$
- For multi-pop(k): $c_i \leq Ck$

We now define $\Phi(D) = C \cdot [\#$ elements in stack$]$.

- for push: $\hat{c}_i = c_1 + [\Phi(D_i) - \Phi(D_{i-1})] = C + C = 2C$
- For multi-pop(k): $\hat{c}_i = c_1 + [\Phi(D_i) - \Phi(D_{i-1})] = Ck - Ck = 0$
- So $\hat{c}_i \leq 2C$, so $\hat{c}_i = O(1)$, so amortized update time is $O(1)$.

2 Incrementing Binary Counters (Chapters 17.1 - 17.3 in CLRS, 3rd edition)

2.1 Defining the problem

Say that you have a binary counter with k bits, initially all 0. The largest number one can represent with k bits is is $2^k - 1$. Now, say that you continually increment the counter by 1.

Question: What is the amortized number of bit flips per counter increment? In other words, after t increments, what is the total number of times that some bit flips in the binary counter?

Observation: Every time we increment counter, exactly one 0 flips to a 1. But there might be many 1s that flip to 0. We thus have

$$c_i = 1 + [\#\text{ 1's that flip to 0}]$$

Trivial Analysis: Since we have k bits, we have $c_i \leq k$, so after t increments the total # of flips is $\leq kt$. This is true, but is a weak upper bound.

2.2 Aggregate analysis:

Let the bits of the counter be $b_k, ..., b_0$ (so b_0 is the unit bit). It is easy to check the following:

Observation 2.1. *If bit b_i flips whenever the counter is a multiple of 2^i.*

By the above observation, bit b_i flips every 2^i counter-increments, so given t counter increments, bit b_i flips at most t 2^i times. Thus, the total number of flips is at most:

$$\sum_{i=0}^{k} t/2^i \leq t \sum_{i=0}^{\infty} 1/2^i = 2t$$

2.3 Accounting Analysis

Let's say that every flip costs \$1. Imagine that every time we flip a 0 to a 1, we put an extra dollar in the bank. Thus, each 0-to-1 flip costs \$2: one for the flip, one of the bank. But now, we can process all 1-to-0 flips using our bank money, without having to contribute new \$. Since each counter-increment only causes a single 0-to-1 flip, we need to contribute \$2 per counter increment, so the amortized # flips per counter-increment is 2.

2.4 Analyzing # flips with potential function

Let C_i be the state of the counter after i increments. Following the above accounting analysis, it is easy to see that the amount of \$ in the bank is equal equal to the number of bits that are 1. Thus, for the state of the counter C, we define:

$$\Phi(C) = [\text{\# bits that are 1}]$$

.

- note: $\Phi(C_0) = 0$
- note: $\Phi(C_i) \geq 0$ for all i.

We thus have:

$$\hat{c}_i = c_i + [\Phi(C_i) - \Phi(C_{i-1})] = (1 + \#\text{ 1-to-0 flips}) + [1 - \#\text{ 1-to-0 flips}] = 2$$

So by Corollary 1.5, the amortized # flips per counter-increment is at most 2.

3 Dynamic Tables (Chapter 17.4 in CLRS)

Computer scientists use arrays as their default data structure. The problem with arrays is that you need to specify the size in advance. In particular, to initialize an array of size n, the compiler runs Memalloc(n) to find n consecutive memory slots

- The fact that the memory slots are consecutive is the reason arrays have $O(1)$ lookup.
- But now say that you want you add one more element to end of array.
- The problem is that the next memory slot might be full; the memory allocator is in fact incentivitzed to find a block of *exactly* n memory slots to minimize gaps.

Goal: A *dynamic table* is a data structure T that can support the following operations

- T.find(i) – return the ith element
- T.append(x) – adds x to end of table
- T.pop() – deletes last element in table.

Failed approaches

- A stack can do append and pop quickly, but T.find takes $O(n)$ time, where n is # elements in table.
- An array can do find quickly, but cannot append. So to execute T.append, you would need to build a new array of larger size and copy the older one into it; this takes $O(n)$ time.

Goal: dynamic table where all operations can be done in $O(1)$ time.

3.1 First step: dynamic tables without deletion

We will follow the above approach of using an array and copying into a larger array when we run of space. But with the following crucial change: to limit the number of rebuilds, whenever we construct a new array, we make sure it has extra space.

Definition 6 (Information stored in table T)**.**

T stores Two counters T.num and T.size, initially both set to 0.

T.num is always equal to the number of elements in the table

An array $T.A[1...T.size]$ of size T.size that stores all the elements in consecutive order; note that this requires T.size $\geq$ T.num

Definition 7. we say that an implementation of a dynamic table T has load factor $\alpha \geq 1$ if at all times T.size/T.num $\leq \alpha$.

- DISCUSS: why small load factor is good.
- We will show a table with load factor 2 for insertions only and 4 for append+pop.
- HW: table with load factor 1.25.

Implementation of find: Easy; to find the ith element just look in $T.A[i]$.

Implementation of append(x)

1. if T.num = T.size = 0
   - T.A $\leftarrow$ array of size 1
   - $A[i] \leftarrow x$.
   - T.size $\leftarrow$ 1.
2. If T.num $<$ T.size
   - T.A[T.num] $\leftarrow$ x

3. If T.num = T.size
   (a) T.size ← 2T.num
   (b) Initialize a new array NewArray of size T.size
   (c) Copy all the elements of A into NewArray (in that order)
   (d) Release the memory of array A
   (e) T.A ← NewArray
   (f) T.A[T.num] ← x
4. T.num++

FIGURE: 17.3 page 467

Observation 3.1. *: The above implementation has load factor* 2*.*

Worst-case update time:

- T.find always takes $O(1)$ time
- T.append takes $O(1)$ time if item 3 is not executed
- But T.append takes $O(n)$ time if T.append is executed.

3.2 Amoritzed update time of dynamic tables

Real Costs

- $c_i = \$1$ for a find operation
- $c_i = \$1$ for T.append where rebuild does not occur
- $c_i = \$T.num$ for append operation where rebuild occurs.

Goal: show that T.append has *amortized* update time $O(1)$.

Intuition – Accounting Analysis: Say that whenever we append an element, we contribute \$2 to the bank. Now, say that some append update causes us to rebuild the array A (Line 3 of T.append()). This must be because right before this append we have we have T.num = T.size. But note that when we first initialized the current array we had $T.num = T.size/2$, so we had added $T.num/2$ elements since the last rebuilt, so we have $T.num$ \$ in the bank, so we can spend those dollars copying all $T.num$ elements into a new array. We thus have all the money we need to implement the expensive rebuild operations without needing to contribute any additional money. Since we only spent \$3 per operation (\$1 for the append and \$ added to bank) The amortized cost is thus \$3, which is $O(1)$ units of time.

Potential Function Define

$$\Phi(T) = 2T.num - T.size$$

DISCUSS: how we can derive this potential function from above accounting analysis.

- Want $\phi(T)$ to decrease by at least T.num whenever we do a rebuilt (to pay for the cost of that operation)
- Want $\phi(T)$ to go up by \$ 2 whenever $T.num$ increases by 1.

Basic Observations:

- When T is empty, $\Phi(T) = 0$.
- Always have $\Phi(T) \geq 0$ by Observation 3.1.

Analyzing T.find If the kth update is T.find(i), then since find operations require \$1 and do not affect T.num or T.size we have $\hat{c}_k = c_k + \Phi(T_k) - \Phi(T_{k-1}) = c_k = \$1$.

Analyzing T.append(x) Let's analyze the cost $\hat{c}_k = c_k + \Phi(T_k) - \Phi(T_{k-1})$.

- If kth update is an append that does not trigger the rebuild of Line 3, then T.num increases by 1 and T.size remains the same, so $\Phi(T_k) - \Phi(T_{k-1}) = 2$, so $\hat{c}_k = c_k + 2 = 1 + 2 = 3\$$.

- We now focus on the case where kth update *does* trigger the rebuild of Line 3.
- Note that T_{k-1}.num = T_{k-1}.size, since otherwise we would not have triggered a rebuild.
- Thus $\Phi(T_{k-1}) = T_{k-1}$.num = T_k.num - 1
- After the rebuild, $T_k.size = 2T_k.num$, so $\Phi(T_k) = 0$.
- Thus $\Phi(T_k) - \Phi(T_{k-1}) = 1 - T_k.num$
- So $\hat{c}_k = c_k + 1 - T_k.num = T_k.num + 1 - T_k.num = \$1$.

We have thus shown that $\hat{c}_k \leq \$3$ for all k, which is $O(1)$ units of time, so by Corollary 1.5, the amortized update time is $O(1)$.

3.3 Processing Deletions

Note that after many T.pop(), we need to copy into a smaller array because otherwise we wil have $T.size >> T.num$ and the load factor will be too big.

3.3.1 Failed Attempt:

Whenever $T.num < T.size/2$, copy into a smaller array of size $T.num$. This fails.

Bad Example: Say we keep inserting elements until we have

$$T.num = n \text{ and } T.size = n$$

Now we append one more element, which forced a rebuilt: so now we have

$$T.num = n + 1 \text{ and } T.size = 2n + 2$$

Now we repeat forever: pop,append,pop,append,pop,append.

It's easy to check that *every* operation in this infinite sequence takes $O(n)$ time.

3.3.2 Better Attempt:

We need to make sure that after a rebuilt, there need to be a lot of deletions. To do this, we will increase our allowed load factor to 4. The idea is then:

- Rebuild the array when T.num > T.size, setting T.size = 2T.num
- Rebuild the array when T.num < T.size/4, again setting T.size = 2T.num

Analysis – Accounting

- Consider the situation right after a rebuild; so $T.size = 2T.num$
- The # of elements in the next rebuild will be at most $2T.num + 1$, so the next rebuild will cost at most $O(T.num)$, which is $\leq CT.num$ for some constant C.
- Note that to get to the next rebuild we would need to either double $T.num$ or divide it by 4, which requires at least $T.num/2$ appends/pops. (It's actually at least $3T.num/4$, but let's keep it simple.)
- So as long as we put in \$ $2C$ per append/pop, we will have enough money to pay for rebuild.
- So amortized update time is $O(1)$.

Potential Function DISCUSS: how to get potential function from above accounting analysis.

$$\Phi(T) = C(|2T.num - T.size|)$$

SHOW: under this potential function, $\hat{c}_i \leq 2C = O(1)$ for every operation.

- If no rebuild then $\phi(T_i) - \phi(T_{i-1}) \leq 2C = O(1)$
- If append causes rebuild then:
  - before rebuild: $\phi(T_{i-1}) = C \cdot T.num$
  - after rebuild: $\phi(T_{i-1}) = 0$

- So $\phi(T_i) - \phi(T_{i-1}) = -C \cdot T.num$
- So $\hat{c}_i = c_i - 2C \cdot T.num = C \cdot T.num - C \cdot T.num = 0$

- If pop causes rebuild then:
  - before rebuild: T.size = 4T.num, so $\phi(T_{i-1}) = 2C \cdot T.num$
  - after rebuild: $\phi(T_{i-1}) = 0$
  - So $\phi(T_i) - \phi(T_{i-1}) = -2C \cdot T.num$
  - So $\hat{c}_i = c_i - 4C \cdot T.num = C \cdot T.num - 2C \cdot T.num < 0$