CSC263H
Data Structures and Analysis

Prof. Bahar Aameri & Prof. Marsha Chechik

Winter 2024 – Week 6

Amortized Analysis: Motivation

- The claim that average-case run time of hash operations is in $\Theta(1)$ is based on the assumption that α is in $\Theta(1)$.
  That is, $\frac{n}{m} = c$ where c is some constant number.

- If a hash table has many more elements than buckets, α will NOT be in $\Theta(1)$ anymore.

- **Solution**: Increase the size of the table when the number of elements in the table gets too large compared to the size of the table.
  (typically when $\alpha = 2$ for chaining and $\alpha = 0.75$ for open addressing).
  Double the size of the old table (typically), and rehash all the elements into the new table.

Issue: Rehashing all elements in the table is NOT constant time.
Intuitive Answer: Rehashing happens very rarely, so on average the cost of *HashInsert* remains in $\Theta(1)$.

Need a formal approach to prove this claim! $\rightarrow$ Amortized Analysis

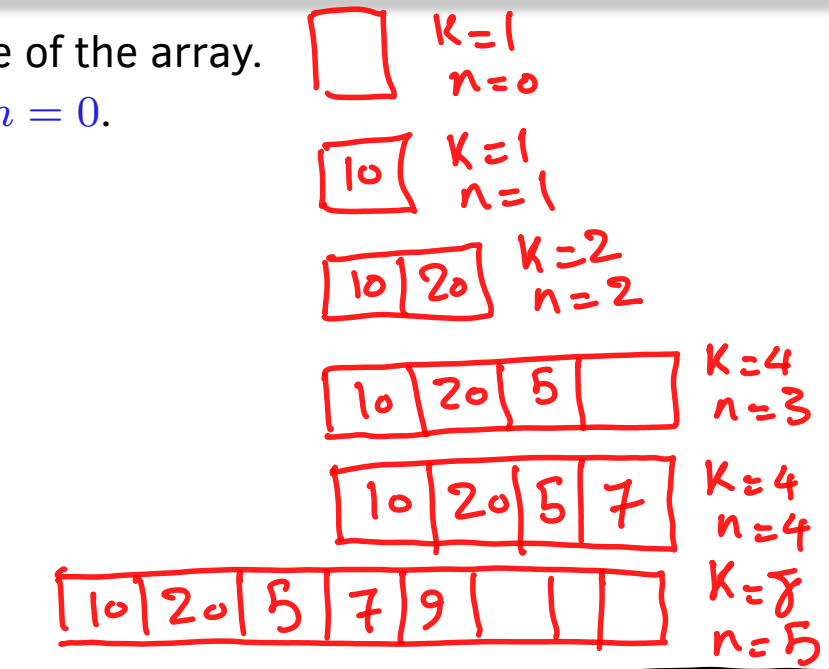
Amortized analysis of hash tables are a bit complicated for this course.
Instead we consider a simplified version of hash table resizing: **Dynamic Arrays**

Dynamic Arrays

- *Append*(A_1, x): store element x in the first free position of array A_1.
  If A_1 is full, create new array A_2 **twice** the size of A_1, copy over all elements in A_1 to A_2, then append x.

n: number of elements stored in the array. k: size of the array.
Assumption: start with an array with $k = 1$ and $n = 0$.

Append(A_1, x):
1. **if** $n == k$:
2. Create a new array A_2 with size $2 \times k$
3. Copy all the elements from A_1 to A_2.
4. $k = k \times 2$
5. $n = n + 1$
6. $A_2[n] = x$
7. **else**:
8. $n = n + 1$
9. $A_1[n] = x$



Worst-case run time of *Append*:

Suppose we perform a sequence of m *Append* operations.
What is the **total** cost of this sequence of operations in the worst-case?

$T(i)$: worst-case run time of the $i-$th operation in the sequence.

Total cost of the sequence $= \sum_{i=1}^{m} T(i)$

Let $T'(i) = T(i) - 1 \Rightarrow T'(i) = \begin{cases} i & \text{if } i \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$

$$\sum_{i=1}^{m} T(i) = \sum_{i=1}^{m} 1 + T'(i)$$

$$= m + \sum_{i=1}^{m} T'(i)$$

$$= m + \sum_{j=0}^{\lfloor \lg_2 m \rfloor} 2^j$$

$$1 \leq m \leq 2^{\lg_2 m}$$

$$2^0 \leq m \leq 2^{\lg_2 m}$$

Largest power of 2 before m is $2^{\lfloor \lg_2 m \rfloor}$

$$= \underbrace{m}_{\in \Theta(m)} + \underbrace{2^{\lfloor \lg_2 m \rfloor + 1} - 1}_{\in \Theta(m)}$$

$$\Rightarrow T_m^{sq} = \sum_{i=1}^{m} T(i) \in \Theta(m)$$

$$\Rightarrow \frac{T_m^{sq}}{m} \in \Theta(1)$$

Amortized Analysis: General Idea

Worst-case sequence complexity:

- **total cost** of performing a **sequence** of m operations in the **worst case**.
- Denoted by T_m^{sq}, where m is the number of operations in the sequence.
- **Cost of an operation**: Number of **steps** executed by the operation.

Amortized sequence complexity of a sequence of m operations:

$$\frac{\text{worst-case sequence complexity}}{m} = \frac{T_m^{sq}}{m}$$

Basically, it is the **average** cost per operation.

Amortized sequence complexity for **Dynamic Arrays**: $\Theta(1)$

- **Amortized Analysis**: a **worst-case** analysis of a **sequence of operations**.
- Often we use amortize analysis to show that the **average** cost of an operation is **small**, even though a **single** operation within the sequence might be **expensive**.
- Amortized complexity can be interpreted as average worst-case complexity of **each** operation in a sequence of operations.

Do NOT confuse Amortized Analysis with Average-case run time!

- **Average-case Analysis**: considers an **individual** operation; uses **probability** over the space of all possible inputs.
- **Amortized Analysis**: considers **multiple** operations; involves **NO probability**.

Amortized Analysis: Aggregate Method

Aggregate Method:

- Find the **total cost** of performing a **sequence** of operations in the **worst case**.
  No overstimate or underestimate
- Divide the **total cost** by the **number of operations** in the sequence.

Aggregate Method: Example

Consider a different implementation for *Append*:
Initially, the array has size 10 and is empty.
When the array gets full, allocate a new array with room for **10 more** elements (instead of doubling the size).
Calculate the amortized sequence complexity for this implementation.

n: number of elements stored in the array.
k: size of the array.
Assumption: start with an array with $k = 10$ and $n = 0$.

Append(A_1, x) :
1. **if** $n == k$:
2. Create a new array A_2 with size $k + 10$
3. Copy all the elements from A_1 to A_2.
4. $k = k + 10$
5. $n = n + 1$
6. $A_2[n] = x$
7. **else**:
8. $n = n + 1$
9. $A_1[n] = x$

number of Appends in the sequence	number of array extensions (so far)	number of copying (so far)	number of array insert
1	0	0	1
2	0	0	2
⋮	⋮	⋮	⋮
10	0	0	10
11	1	10	11
12	1	10	12
⋮	⋮	⋮	⋮
20	1	10	20
21	2	10+20	21
⋮			
m	$\lfloor \frac{m-1}{10} \rfloor$	$10+20+30+\cdots+10\lfloor \frac{m-1}{10} \rfloor$	m

$$T_m^{sq} = m + \sum_{i=1}^{\lfloor \frac{m-1}{10} \rfloor} 10 \times i = m + 10 \sum_{i=1}^{\lfloor \frac{m-1}{10} \rfloor} i$$

$$= \underbrace{m}_{\in \Theta(m)} + 10 \underbrace{\frac{\lfloor \frac{m-1}{10} \rfloor \left(\lfloor \frac{m-1}{10} \rfloor + 1 \right)}{2}}_{\in \Theta(m^2)}$$

$$\Rightarrow T_m^{sq} \in \Theta(m^2) \Rightarrow \frac{T_m^{sq}}{m} \in \Theta(m)$$

Amortized Analysis: Accounting Method

- Let c_i denote the **actual** cost of i-th operation (in the worst-case) in a sequence of m operations. Then

$$T_m^{sq} = \sum_{i=1}^{m} c_i$$

- Aggregate method is straightforward when we can easily identify a worst-case sequence and c_i for all operations in such a sequence.

- **Problem**: In some cases, it is not easy to calculate the exact value for $\sum_{i=1}^{m} c_i$.

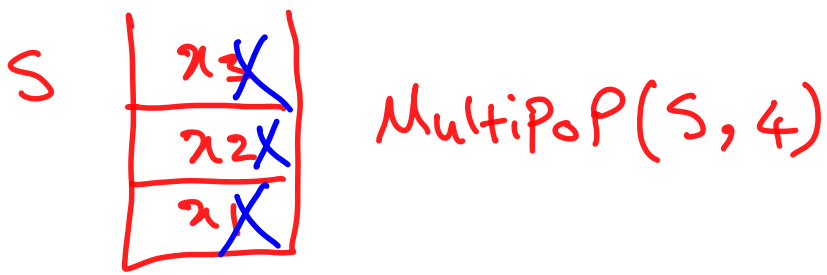
Example: Augmented Stack

Push(S, x): push object x onto stack S.

Pop(S): pop the top of stack S and return the popped object. Calling *Pop* on an empty stack generates an error.

MultiPop(S, k): remove the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects.

MultiPop(S, k) :
1. **while not** *StackEmpty*(S) **and** $k > 0$:
2. *Pop*(S)
3. $k = k - 1$



Push **actual** cost: 1
Pop **actual** cost: 1
MultiPop **actual** cost: $\min(n, k)$

Amortized Analysis: Accounting Method

Problem: In some cases, it is not easy to calculate the exact value for $\sum_{i=1}^{m} c_i$.

Solution: Instead of considering the actual cost of each operation, assign an **amortized cost** a_i to each operation so that:

- calculation of $\sum_{i=1}^{m} a_i$ is *straightforward*;
- we can prove that for all sequences of operations the **total amortized** cost of a sequence of operations provides an **upper-bound** on the **total actual** cost of the sequence. That is,

total amortized cost $\sum_{i=1}^{m} a_i \geqslant \sum_{i=1}^{m} c_i$ total actual cost

- Then, since $T_m^{sq} = \sum_{i=1}^{m} c_i$, we have

$$\sum_{i=1}^{m} a_i \geqslant T_m^{sq}$$

Accounting Method: Intuition

How should **amortized costs** be assigned?

$\sum a_i - \sum c_i \geq 0 \Rightarrow$

- Think of it as maintaining a bank account which is used to pay for the number of steps each operation executes.

$\sum a_i \geq \sum c_i$

- Some operations are charged **more** than their **actual** cost.
  The surplus is deposited as **credit** into the bank account for later use.
- Some operations are charged **less** than their **actual** cost.
  The deficit is paid for by the savings (**credit**) in the bank account.
- The amortized cost of each operation must be set *large enough* so that the **balance** in the bank account always remains **non-negative**, but *small enough* so that no operation is charged significantly more than its actual cost.
- **Important:** The extra amortized charges to an operation do **not** mean that the operation really takes that much time. It is just a method of accounting that makes the analysis easier.

Accounting Method: Example

Using Accounting Method to Analyze Amortized Complexity of Augmented Stacks:

Instead of charging the actual cost when an operation is executed, we charge for *Pop* and *MultiPop* operations **in advance**. That is, whenever we push an element, we pay the cost of popping it too.

In other words, we **amortize** the cost of *Pop* and *MultiPop* operations over *Push* operations. More specifically, for each execution of *Push*(S, x), we charge 2:

- Charge 1 for the cost of pushing x;
- Charge 1 for the cost of popping x (which may or may not occur later on).

Accounting Method: Stack Example

Step 1: Find appropriate amortized costs.

Push **amortized** cost: 2
Pop **amortized** cost: 0
MultiPop **amortized** cost: 0

Step 2: Show that for all $k \in \mathbb{Z}^+$, and all possible sequences $\sum_{i=1}^{k} a_i \geqslant \sum_{i=1}^{k} c_i$.
Proof:
Suppose we use a dollar bill to represent each unit of cost.
When we push an object x, we use 1 dollar to pay the actual cost of the push and x is left with a credit of 1 dollar.
When a *Pop* or *MultiPop* is executed on x, we pay its cost using the credit stored for x.
It is not possible to pop an element that has not been pushed!
So the total credit of stack objects is non-negative for every possible sequence.
Total credit is equal to the difference between total amortized cost and total actual cost.
That is, for any sequence of k *Push*, *Pop* and *MultiPop* operations we have

$$(\sum_{i=1}^{k} a_i - \sum_{i=1}^{k} c_i) \geqslant 0,$$

meaning that $\sum_{i=1}^{k} a_i \geqslant \sum_{i=1}^{k} c_i$ always holds.

Step 3: Find an upper-bound for T_m^{sq} based on amortized costs:

$$T_m^{Sq} = \sum_{i=1}^{m} c_i \leq \sum_{i=1}^{m} a_i \leq \sum_{i=1}^{m} 2 = 2m$$

$$\frac{T_m^{Sq}}{m} \leq \frac{2m}{m} \Rightarrow \frac{T_m^{Sq}}{m} \in O(1)$$

Amortized sequence complexity for **Stacks**: $O(1)$

Accounting Method: Example

Dynamic Arrays

- *Append*(A_1, x): store element x in the first free position of array A_1.
  If A_1 is full, create new array A_2 **twice** the size of A_1, copy over all elements in A_1 to A_2, then append x.
- *Delete*: remove element in last occupied position.

Use accounting method to analyze amortized sequence complexity of modified Dynamic Arrays.
Idea: We must pay for the cost of **copying** an element during an extension in advance.

Can we charge 2 for Append, 1 for Delete?

Initial  $n=0, K=1$

Append (X_1)  $n=1, K=1$

Append (X_2)  $n=2, K=2$

Append (X_3)  $n=3, K=4$

Append (X_4)  $n=4, K=4$

Append (X_5)  $n=5, K=8$

charge 2, pay 1, credit 1

charge 2, pay 1+1, credit

charge 2, pay 2+1, credit

charge 2, pay 1, credit 1

charge 2, pay 4+1, credit

2 is sufficient to copy element only once and we need to copy multiple times!

Accounting Method: Example

Issue: When we charge 2 for *Append*, only elements in the second half of the array have credit for copy in the next extension.

Intuitive Solution: Make each element in the ***second half*** responsible for paying to copy both itself and one of the elements in the first half.
To have sufficient credit for this, we need to **charge 3** for each *Append*(A, x):

- Spend 1 for the cost of inserting x into the array;
- Save 2 for the *future cost of copying x and one of the elements in the first half* of A in the next extension (which may or may not happen).

That is, if we **charge 3** for each *Append* (and 1 for *Delete*), each element in the **second half** must always have a **credit of at least 2**.

Accounting Method: Example

Claim: Assuming that we start from an empty dynamic array of size 1, and charge 3 for each *Append* and 1 for each *Delete*, each element in the second half of the array has a credit of at least 2. (**Credit Invariant**)

Proof: Induction over the number of operations in the sequence.

Base Case: Initially, the array has size 1 and no elements.
If the first operation is a *Delete*, the array remains empty and so the statement is vacuously true. If the first operation is an *Append*, use 1 to pay for storing the new element, and keep 2 as credit with the only element in the array.

Induction Hypothesis: Assume that the invariant is true after a sequence of length m.

Suppose that at this point the length of the array is k, and there are $r \geq 0$ elements in the second half.

If the next operation is a *Delete*, then at most 1 item from the second half is removed but the credit remains the same (because we charge and use 1 unit for *Delete* and the credit for the removed item remains in the array, even though the item itself is removed). So the credit invariant holds.

Accounting Method: Example

If the next operation is an *Append*:

- If the size of the array does not need to be changed, simply use 1 to pay for storing the new element, and keep 2 as credit with that new element. Since new elements are only added in the second half of the array, the credit invariant is maintained.

- If the size of the array needs to be changed, then this means that the array is full. Since the number of elements in the first half of the array is the same as the number of elements in the second half of the array, and since we have 2 credit on each element in the second half, we have enough money to pay for the cost of copying all the elements into the new array of twice the size. Afterwards, we use the 3 as before, to pay for storing the new element and keep 2 credit on that new element. And as before, the invariant is maintained.

Accounting Method: Example

The base case of the inductive proof shows that for arrays of size 1 the credit is always non-negative.
Suppose after m operations the size of the array is larger than 1, meaning that at least one extension has happened and at least one element exists in the second half after the extension (this is based on the description of *Append*).
Given the credit invariant we proved, if after m operations there's at least one item in the second half of the array, then the total credit in the array is at least 2.
If there's no item in the second half, then at least one *Delete* operation has occurred in the sequence which removed the item(s) in the second half. But the credit of the removed item(s) remains in the array and is never used later in the sequence. So the total credit is positive in this case.
Hence, the number of credits in the array never becomes negative. That is, the total amortized cost for the sequence is always greater than or equal to the total actual cost of the sequence.

Since for all operations the amortized cost is less than or equal to 3, we have:

$$T_m^{sq} = \sum_{i=1}^{m} c_i \leq \sum_{i=1}^{m} a_i \leq \sum_{i=1}^{m} 3 = 3m$$

Thus, the amortized complexity of each operation (i.e., $\frac{T_m^{sq}}{m}$) in any sequence is upper-bounded by 3 which is a constant.

Step 1: Find appropriate **amortized** costs for all operation.

Step 2: Show that for all $k \in \mathbb{Z}^+$, and all possible sequences $\sum_{i=1}^{k} a_i \geqslant \sum_{i=1}^{k} c_i$.

- We might need to identify and prove a **credit invariant** to show that the above statement holds.
- After trying to prove the credit invariant and/or the inequality, you might realize that the amortized costs you considered are insufficient.
  You then must adjust the costs accordingly and re-do Step 2.

Step 3: Find an upper-bound for T_m^{sq} based on total amortized costs.

Accounting Method: Summary

- **Amortized** cost of some operations might be more than their **actual** cost.
- **Amortized** cost of some operations might be less than their **actual** cost.
- When an operation's amortized cost **exceeds** its actual cost, we assign the difference to the data structure as credit.
- Credit will be used to pay for later operations whose amortized cost is less than their actual cost.
- Comping up with amortized costs is **problem-dependent**, and there could be *more than one* correct charging scheme for a given set of operations
- The actual cost and the amortized cost of an operation are **not** necessarily in the **same order**.
  Example in the Augmented Stack example, the actual cost of *MultiPop* is NOT constant, but we can cover the actual costs by charging constant amortized costs.
- There are also cases where we have to assign **non-constant amortized costs** to operations.

After Lecture

- After-lecture Readings: Chapter 7 in the course notes.
- Optional Readings: CLRS Sections 17.1, 17.2.
- Exercise of Chapter 7 in the course notes, Problem 17.1-2 in the CLRS.