دانشگاه آزاد اسلامی واحد تبریز

نام درس: طراحی و تحلیل الگوریتم‌های پیشرفته

بخش: تحلیل استهلاک

نام استاد: دکتر مسعود کارگر

Amortized analysis

- Main goals of the lecture:
    - to understand what is **amortized analysis**, when is it used, and how it differs from the average-case analysis;
    - to be able to apply the techniques of the **aggregate analysis**, the **accounting method**, and the **potential method** to analyze operations on simple data structures.

Sequence of operations

- ## The problem:
  - ### We have a data structure
  - ### We perform a sequence of operations
    - #### Operations may be of different types (e.g., insert, delete)
    - #### Depending on the state of the structure the actual cost of an operation may differ (e.g., inserting into a sorted array)
  - ### Just analyzing the worst-case time of a single operation may not say too much
  - ### We want the average running time of an operation (but from the worst-case sequence of operations!).

درس : طراحی و تحلیل الگوریتم‌های پیشرفته   استاد : دکترمسعودکارگر   دانشگاه آزاداسلامی واحد تبریز

Binary counter example

- Example data structure: a binary counter
  - Operation: Increment
  - Implementation: An array of bits A[0..k–1]

```
Increment(A)
1 i ← 0
2 while i < k and A[i] = 1 do
3   A[i] ← 0
4   i ← i + 1
5 if i < k then A[i] ← 1
```

- How many bit assignments do we have to do in the **worst-case** to perform Increment(A)?
  - But usually we do much less bit assignments!

Analysis of binary counter

- How many bit-assignments do we do on average?
  - Let's consider a sequence of n Increment's
  - Let's compute the sum of bit assignments:
    - A[0] assigned on each operation: n assignments
    - A[1] assigned every two operations: n/2 assignments
    - A[2] assigned every four ops: n/4 assignments
    - A[i] assigned every 2^i ops: $n/2^i$ assignments

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

Thus, a single operation takes 2n/n = 2 = O(1)
time **amortized** time

Aggregate analysis

- **Aggregate** analysis – a simple way to do amortized analysis
  - Treat all operations equally
  - Compute the worst-case running time of a sequence of n operations.
  - Divide by n to get an amortized running time

Another look at binary counter

- Another way of looking at it (proving the amortized time):
  - To assign a bit, I have to use one dollar
  - When I assign "1", I use one dollar, plus I put one dollar in my "savings account" associated with that bit.
  - When I assign "0", I can do it using a dollar from the savings account on that bit
  - How much do I have to pay for the Increment(A) for this scheme to work?
    - Only one assignment of "1" in the algorithm. Obviously, two dollars will always pay for the operation

Accounting method

- Principles of the **accounting method**
    - 1. Associate credit accounts with different parts of the structure
    - 2. Associate amortized costs with operations and show how they credit or debit accounts
        - Different costs may be assigned to different operations
    - Requirement (c – real cost, c' – amortized cost):

$$\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$$

    - This is equivalent to requiring that the sum of all credits in the data structure is non-negative
        - What would it mean not satisfy this requirement?
    - 3. Show that this requirement is satisfied

Stack example

- Start with an empty stack and consider a sequence of n operations: Push, Pop, and Multipop(k).
  - What is the worst-case running time of an operation from this sequence?
  - 1. Let's associate an account with each element in the stack
  - 2. After pushing an element, put a dollar into the account associated with it,
    - then Pop and Multipop can work only using money in the accounts (amortized cost 0)
    - Push has amortized cost 2
  - 3. The total credit in the structure is always ≥ 0
  - Thus, the amortized cost of an operation is O(1)

Potential method

- We can have one account associated with the whole structure:
  - We call it a **potential**
  - It's a function that maps a state of the data structure after operation i to a number: $\Phi(D_i)$

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

  - The main step of this method is defining the potential function
    - Requirement: $\Phi(D_n) - \Phi(D_0) \geq 0$
  - Once we have Φ, we can compute the amortized costs of operations

Binary counter example

- How do we define the potential function for the binary counter?
  - Potential of A: b_i – a number of "1"s
  - What is $\Phi(D_i) - \Phi(D_{i-1})$, if the number of bits set to 0 in operation i is t_i?
  - What is the amortized cost of Increment(A)?
    - We showed that $\Phi(D_i) - \Phi(D_{i-1}) \le 1 - t_i$
    - Real cost $c_i = t_i + 1$
    - Thus,

$$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le (t_i + 1) + (1 - t_i) = 2$$

Potential method

- We can analyze the counter even if it does not start at 0 using potential method:
    - Let's say we start with b_0 and end with b_n "1"s
    - Observe that:
    $$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} c_i' - \Phi(D_n) + \Phi(D_0)$$

    - We have that: $c_i' \le 2$

    - This means that:
    $$\sum_{i=1}^{n} c_i \le 2n - b_n + b_0$$
    - Note that $b_0 \le k$. This means that, if k = O(n) then the total actual cost is O(n).

Dynamic table

- It is often useful to have a dynamic table:
  - The table that expands and contracts as necessary when new elements are added or deleted.
    - **Expands** when insertion is done and the table is already full
    - **Contracts** when deletion is done and there is "too much" free space
  - Contracting or expanding involves **relocating**
    - Allocate new memory space of the new size
    - Copy all elements from the table into the new space
    - Free the old space
  - Worst-case time for insertions and deltions:
    - Without relocation: O(1)
    - With relocation: O(m), where m – the number of elements in the table

Requirements

- Load factor
    - num – current number of elements in the table
    - size – the total number of elements that can be stored in the allocated memory
    - Load factor α = num/size
- It would be nice to have these two properties:
    - Amortized cost of insert and delete is constant
    - The load factor is always above some constant
        - That is the table is not too empty

Naïve insertions

- Let's look only at insertions: Why not expand the table by some constant when it overflows?
    - What is the amortized cost of an insertion?
    - Does it satisfy the second requirement?

Aggregate analysis

- The "right" way to expand – double the size of the table
  - Let's do an aggregate analysis
  - The cost of i-th insertion is:
    - i, if i–1 is an exact power of 2
    - 1, otherwise
  - Let's sum up…
  - The total cost of n insertions is then < 3n
  - Accounting method gives the intuition:
    - Pay $1 for inserting the element
    - Put $1 into element's account for reallocating it later
    - Put $1 into the account of another element to pay for a later relocation of that element

Potential function

- What potential function do we want to have?
    - $\Phi_i = 2\text{num}_i - \text{size}_i$
    - It is always non-negative
    - Amortized cost of insertion:
        - Insertion triggers an expansion
        - Insertion does not trigger an expansion
    - Both cases: 3

Deletions

- Deletions: What if we contract whenever the table is about to get less than half full?
  - Would the amortized running times of a sequence of insertions and deletions be constant?
  - Problem: we want to avoid doing reallocations often without having accumulated "the money" to pay for that!

درس : طراحی و تحلیل الگوریتم‌های پیشرفته   استاد : دکترمسعودکارگر   دانشگاه آزاداسلامی واحد تبریز

Deletions

- Idea: delay contraction!
  - Contract only when num = size/4
  - Second requirement still satisfied: $\alpha \geq$ ¼
- How do we define the potential function?

$$\Phi = \begin{cases} 2 \cdot num - size & \text{if } \alpha \geq 1/2 \\ size/2 - num & \text{if } \alpha < 1/2 \end{cases}$$

- It is always non-negative
- Let's compute the amortized running time of deletions:
  - $\alpha <$ ½ (with contraction, without contraction)