# Amortized Analysis

CS31005: Algorithms-II

Autumn 2020

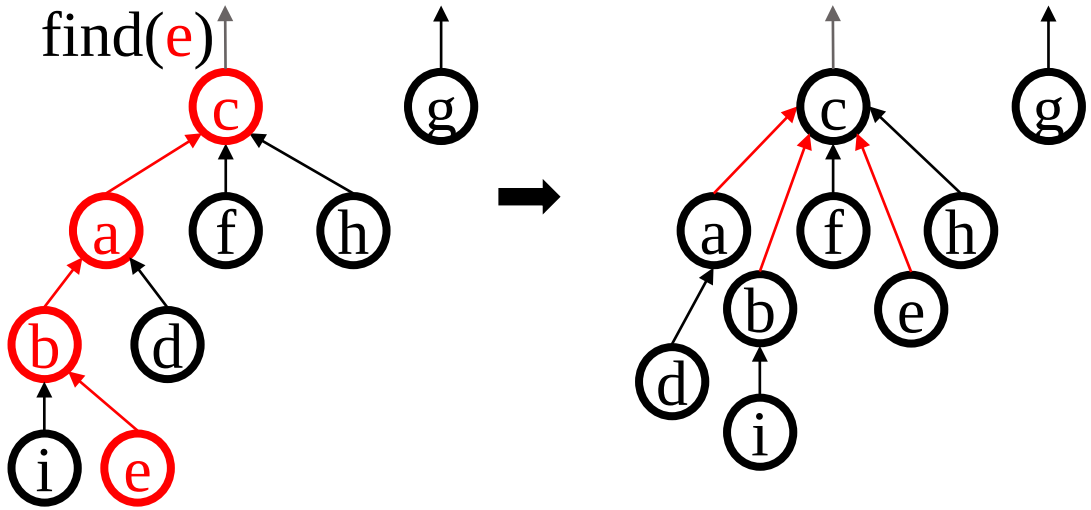
IIT Kharagpur

# Motivation

- You have seen many data structures, each with a set of operations defined on them
- You have done time complexity analysis
  - Worst case time complexity – worst case time for one execution of an operation
    - Ex. Insert in a heap, search in a BST,….
  - Average case time complexity – average time of one execution of an operation over all inputs
    - This is usually done by imposing a probability distribution on the inputs (usually uniform distribution, meaning all inputs are equally likely), and calculating the expected time of execution of the operation over all inputs

- But sometimes, we are interested in considering a set of operations together
- The set may consist of multiple executions of same or different operations
- Trivial worst case complexity analysis will multiply the worst case time of each operation with the number of that operation and sum everything up
- This may not be tight
  - Operations may depend on each other in some ways to affect their time
  - All executions of the same operation in the sequence may not incur the worst case time
- Question: Can we find the average time of an operation in a sequence of operations?

- Example: think of the Disjoint Set (Union-Find) data structure you have studied earlier
  - Represents a set of disjoint sets, each set with a unique id (usually the id of one of the elements in the set
  - Two operations
    - find(x) – returns the id of the set the element x belongs to
    - union(x, y) – merges the set containing element x and the set containing element y into a single set
- Forest implementation of disjoint set
  - Represent each set as a rooted tree, with id of the set = element at root
  - find(x) – traverse the path from x to root to find root and return
    - Also does path compression while doing find() so that subsequent calls take less time
  - union(x, y) – add one root as child of the other root
    - Different techniques as to who goes as child of who
- Read from text if you have forgotten




- find(e) also does path compression, so subsequent calls to say find(b) will take less time
- So if you have a given sequence of finds and unions, can you find the average time per operation (its amortized time) in the set?
- Example: For Kruskal's MST algo, you have a sequence of 2|E| find operations (for each vertex of an edge when you try to check if the edge will create a cycle or not) and (|V|-1) union operations (one for each edge you add to the MST)

# Amortized Analysis

- Considers a sequence of operations on a given data structure
- Computes the average cost over a sequence of operations
  - Guarantees the avg. performance of each operation in the worst case for that sequence of operations
- Note the difference from average case complexity analysis
  - Unlike average case complexity analysis, no involvement of probability
  - Guarantee the average performance of each operation among the sequence in worst case, even if some operations in the sequence are costlier than others
- This is somewhat confusing to understand at first, so let us look at examples

# Examples we will look at: Multipop Stack

- A stack that allows more than one element at in a single pop
- Main operations on a multipop stack

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

POP(S): pops the top of stack S and return the popped object

MULTIPOP(S, k): pops the top k objects of stack S

```
{
  While not STACK-EMPTY(S) and k≠0
     do   POP(S)
          k ← k-1
}
```

- PUSH and POP can be implemented in O(1) time
- MULTIPOP (S, k) can be implemented in O(k) time
- Consider a sequence of $n$ PUSH, POP, MULTIPOP
  - Total number of operations is $n$
  - The $n$ operations can be any mix of PUSH, POP, MULTIPOP
- A simple time complexity analysis
  - The worst case time for a MULTIPOP in the sequence is $O(n)$, since the stack size is at most $n$.
  - Hence the worst case time of the sequence is $O(n^2)$
- This is correct, but not tight
  - One obvious issue with this analysis: all $n$ operations cannot be MULTIPOP!

# Another Example: Binary Counter Increment

- Simple k-bit binary counter, with A[0] as lowest order bit and A[k-1] as highest order bit

```
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;
    }
```

| A[7] | A[6] | A[5] | A[4] | A[3] | A[2] | A[1] | A[0] |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

- Consider a sequence of $n$ increments
- Simple time complexity analysis
  - Worst case time for one Increment operation is $O(k)$ (when A contains all 1s)
  - Hence time for a sequence of $n$ executions is $O(nk)$ (Assuming initial value of 0)
- Again, this is correct, but not tight
  - Note that not every bit changes in every increment
  - Can this be used to find a better bound?

# Techniques for Amortized Analysis

- Aggregate analysis
  - First find total cost of *n* operations, then divide by n to find amortized cost
- Accounting method
  - Assign each type of operation an (different) amortized cost
  - Overcharge some operations
  - Store the overcharge as credit on specific objects
  - Use the credit for compensation for some later operations
- Potential method
  - Same as accounting method
  - But store the credit as "potential energy" as a whole on the data structure (not on specific operations)
- We will apply each of these methods on multipop stack and binary counter examples

# Aggregate Analysis

- Find the worst case time $T(n)$ for a sequence of $n$ operations
- Amortized cost (average cost) per operation is then $T(n)/n$

# Aggregate Method; Multipop Stack

- Consider a sequence of $n$ PUSH, POP, MULTIPOP operations on an initially empty stack
- MULTIPOPS are just a sequence of POPs, so analysis can consider only number PUSH and POPs (either a direct POP or a POP within a MULTIPOP)
- Each element pushed can be popped at most once (either in a direct POP or a POP within a MULTIPOP)
- So the no. of POP operations (including the ones inside MULTIPOP) $\leq$ number of PUSH
- So the total time is at most $O(n)$ (since PUSH and POP are each O(1))
- Hence the average cost of an operation is $O(n)/n = O(1)$
- We say that the amortized cost of a PUSH, POP, or MULTIPOP in a sequence of $n$ such operations is O(1)

# Aggregate Method: Binary Counter

- Total time is bounded by the total number of bit flips
- But each bit does not flip on each increment
  - A[0] flips every time (total $n$ times)
  - A[1] flips every other time (total $n/2$ times)
  - A[2] flips every fourth time (total $n/4$ times)
  - ….
  - for $i$=0,1,…,$k$-1, A[$i$] flips $n/2^i$ times
- Thus total number of bit flips is

$$\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} 1/2^i = 2n$$

So the total running time is $< 2n$

So amortized cost for an increment operation is O(1)

| 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 |

# Accounting Method

- Basic Idea
  - Assign differing charges to different operations
  - The amount of the charge is called the amortized cost of that operation
  - Amortized cost of an operation can be more or less than actual cost
  - When amortized cost > actual cost, the difference is saved in specific objects as credits
  - The credits can be used by later operations whose amortized cost < actual cost
  - As a comparison, in aggregate analysis, all operations have same amortized costs
    - We do not assign an amortized cost at the beginning for any operation, we get the total cost and divide by the number of operation, irrespective of what those operations are

- Conditions to be satisfied by assigned amortized costs
  - Suppose actual cost is $c_i$ for the $i$-th operation in the sequence, and amortized cost is $c_i$'
  - For any sequence of operations, the total amortized cost should be an upper bound of total actual cost

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

    - Otherwise the worst case time obtained using amortized analysis is not really the worst case!
  - The total credits at any point should be non-negative

$$\sum_{i=1}^{t} c_i' - \sum_{i=1}^{t} c_i \geq 0 \text{ for any } 0 < t \leq n$$

    - Otherwise, total amortized cost after t operations is less than the total actual cost, so if the subsequent operations do not give enough credit to make up for it, the condition of amortized cost being upper bound of actual cost will get violated

# Accounting Method: Multipop Stack

- Actual cost of operations: PUSH = 1, POP = 1, MULTIPOP = min (|S|, k)
- Set amortized cost (charge) for operations as: PUSH = 2, POP = 0, MULTIPOP = 0
- Intuition
  - A POP cannot happen without a PUSH
  - While pushing, pay cost 1 for the actual cost of PUSH, and leave a credit of 1 for a POP (direct or within MULTIPOP) in case it is popped later
    - If not popped, the credit stays, remember we only need an upper bound on the actual cost
  - So a POP later does not need to pay anything, it has already been paid by the PUSH
    - PUSH is overcharged (more than 1), POP/MULTIPOP is undercharged (less than 1)

- For a sequence of $n$ PUSH, POP, MULTIIPOP operations, easy to see the conditions hold
  - After any step, amount of total credit never becomes negative
    - If no. of POP (direct or in a MULTIPOP) = no. of PUSH, credit is 0
    - If no. of POP < no. of PUSH, credit is > 0
  - Total amortized cost ≥ total actual cost
- So total actual cost is bounded by total amortized cost = $2n$
- So average cost per operation in the sequence = O(1)

# Accounting Method: Binary Counter

- Two types of flip operations: set to 1 (from 0) and set to 0 (from 1)
- Assign amortized cost of \$2 to "set to 1" flip (overcharge), assign \$0 to "set to 0" flip (undercharge)
  - Intuition: whenever a bit is set, use \$1 to pay the actual cost, and store another \$1 on the bit as credit (to be used for it to be set to 0 later)
  - If it is not set to 0, stays as credit
- When a bit is set to 0, the stored \$1 pays the cost
- Satisfies the conditions
  - Total credit at any time = no. of 1's in the bit pattern, which is never negative
  - Total amortized cost ≥ total actual cost (easy to see, as a bit cannot be set to 0 unless it has been set to 1 earlier)
- At most one bit is set to 1 in each operation, so the amortized cost of an operation is at most \$2
- Thus, total amortized cost of $n$ operations is $O(n)$, and average is $O(1)$

# Potential Method

- Similar idea as in Accounting method
  - Use prepaid work to pay for later work
- Difference
  - Store the prepaid work as potential energy or potential, instead of credit
  - The potential is associated with the data structure as a whole rather than with specific objects within the data structure

- Initial data structure $D_0$
- $n$ operations, resulting in $D_0, D_1, \ldots, D_n$ with costs $c_1, c_2, \ldots, c_n$
- A potential function $\Phi$: $\{D_i\}$ → R (real numbers)
  - $\Phi(D_i)$ is called the potential of $D_i$
  - So the potential of the data structure changes as operations are done on it
- Amortized cost $c_i'$ of the $i$-th operation is:

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

- Total amortized cost

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

- We want the total amortized cost to be an upper bound of the total actual cost
  - So we need $\Phi(D_n) \geq \Phi(D_0)$
  - But this has to be true for any n, so we need $\Phi(D_i) \geq \Phi(D_0)$ for any $i$
  - Define $\Phi(D_0)=0$,and so $\Phi(D_i) \geq 0$, for all $i$
- If the potential change is positive (i.e., $\Phi(D_i) - \Phi(D_{i-1})>0$), then $c_i'$ is an overcharge (so store the increase as potential)
- Otherwise, undercharge (discharge the potential to pay the extra actual cost)

# Potential Method: Multipop Stack

- Assign Potential = number of elements in stack
- So $\Phi(D_0)=0$, and $\Phi(D_i) \geq 0$
- Amortized cost of stack operations
  - PUSH
    - Potential change = (|S|+1) – |S| = 1
    - Amortized cost = actual cost + potential change = 1 + 1 = 2
  - POP
    - Potential change = (|S| – 1) – |S| = -1
    - Amortized cost = actual cost + potential change = – 1 + 1 = 0
  - MULTIPOP(S, k): let k' = min(|S|, k)
    - Potential change = – k'
    - Amortized cost = actual cost + potential change = k' + (– k') = 0

- Total amortized cost per operation is O(1)
- Hence  total amortized cost for $n$ operations is O($n$)
- So average cost per operation is O(1)
- Total amortized cost is an upper bound of total actual cost

# Potential Method: Binary Counter

- Assign potential of the k-bit counter after i-th increment $\Phi(D_i) = b_i$ , the number of 1's in the counter
- So $\Phi(D_i) \geq 0$ for any i (no. of 1's can never be negative in the counter)
- Computing the amortized cost of an increment
  - Suppose the $i$-th operation resets $t_i$ bits
  - Actual cost $c_i$ of the operation is at most $t_i + 1$
    - $= t_i$ for the overflow case when all bits are already 1
  - If $b_i = 0$, then the $i$-th operation resets all $k$ bits, so $b_{i-1} = t_i = k$
    - Again this is the overflow case, so previous value must be all 1's
  - If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$
  - In either case, $b_i \leq b_{i-1} - t_i + 1$

| 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 |

- So potential change is $\Phi(D_i) - \Phi(D_{i-1}) \leq b_{i-1} - t_i + 1 - b_{i-1} = 1 - t_i$
- So amortized cost is: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq t_i + 1 + 1 - t_i = 2$

- The total amortized cost of $n$ increments is $O(n)$
- Thus average cost per increment is $O(1)$