Jason Starovoytov

Christina Mulu

2/12/2026

CSC 310 D01

A1 part 1

1. We need to find the total cost of expensive operations, inexpensive operations, add them, and divide by n (n operations).

   To find the total cost of expensive operations, we need to find the sum of the all of the powers of 2 <= n. If we look at the example of n = 32, we would need to find the sum of 1 + 2 + 4 + 8 + 16 + 32. The sum of that is 63. With that, we can see that the largest possible cost is <= 2n – 1. In terms of O, the cost of this would be O(n).

   The total cost of inexpensive operations is a lot easier to get. The most possible amount of cheap operations is n. Since it costs O(1), n * O(1) = O(n).

   Putting it all together, we have:

   O(n) + O(n) = O(n)

   O(n) / n = O(1)

2. Amortized cost = actual cost + deposits – withdrawals

   So, if we give the PUSH and POP operations an Amortized cost of 2, (the actual cost is 1 + 1 for deposit. We do not need to withdrawal any credit). Once we reach k operations, we already have a build up of k credits (since every operation makes a deposit of 1); therefore, we can make a withdrawal from our deposits. In the end, this means the average time complexity of one operation is O(1). Since we have n operations, the amortized complexity is O(n).

3.

$$\varphi = \log(n!)$$

$n$ = number of values in Heap

$$\text{insert} = \log(n+1) + \log((n+1)!) - \log(n!)$$

$$\log\left(\frac{\cancel{1}\cdot\cancel{2}\cdot\cancel{3}\cdot\ldots\cdot\cancel{n}\cdot\boxed{n+1}}{\cancel{1}\cdot\cancel{2}\cdot\cancel{3}\cdot\ldots\cdot\cancel{n}}\right)$$

$$\log(n+1) + \log(n+1) = \boxed{O(\log(n))}$$

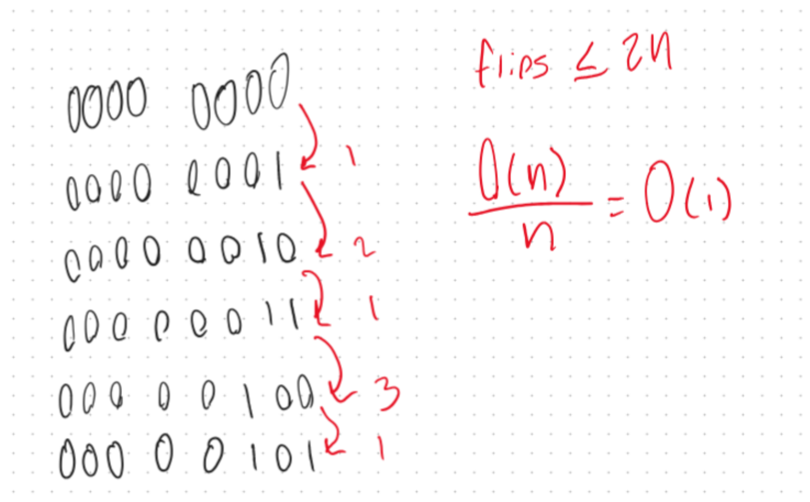
$$\text{delete} = O(\log(n)) + \log((n-1)!) - \log(n!)$$

$$O(\log(n)) - \log(n) = \boxed{O(1)}$$

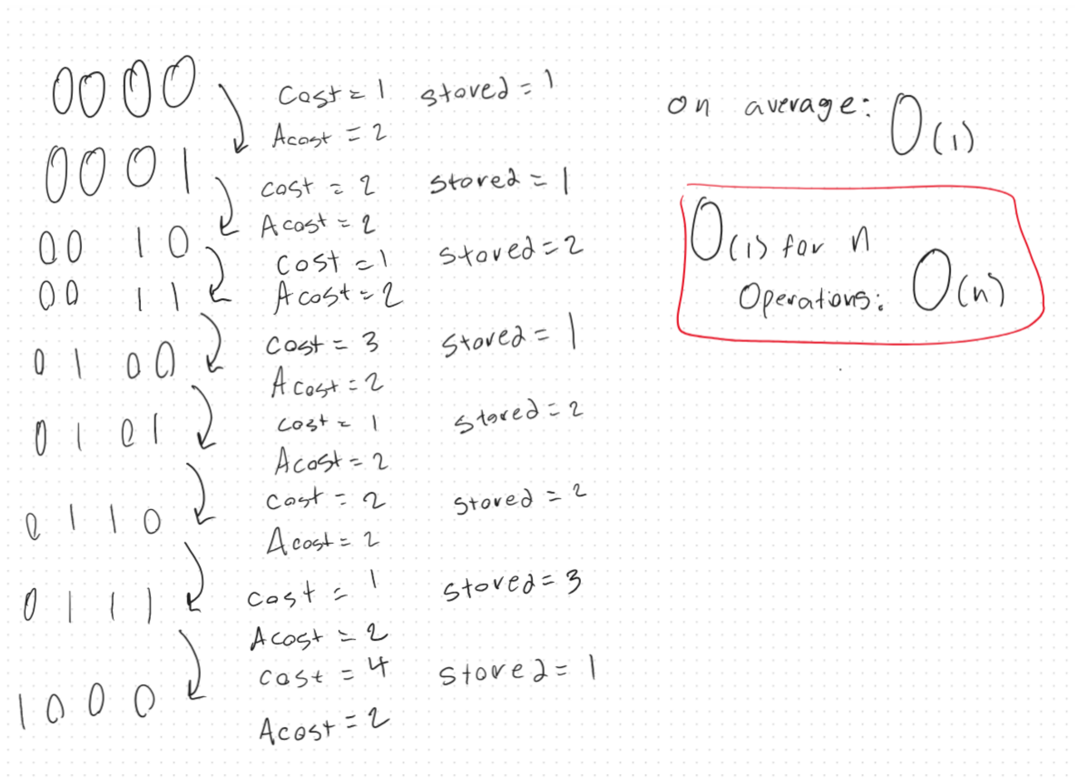
4. While it might take longer for certain operations like insert or delete (only in certain cases. For instance, when the heap must be fixed after a deletion), it is not always the case. When there are cheaper operations, we can use amortized analysis to “store credit”

for operations that are more expensive, and yet to come. One example of this is doubling the heap size. If you have n operations, and want to double it, it would take O(n) time. If you split that work between the n operations from before, that becomes O(1) time.

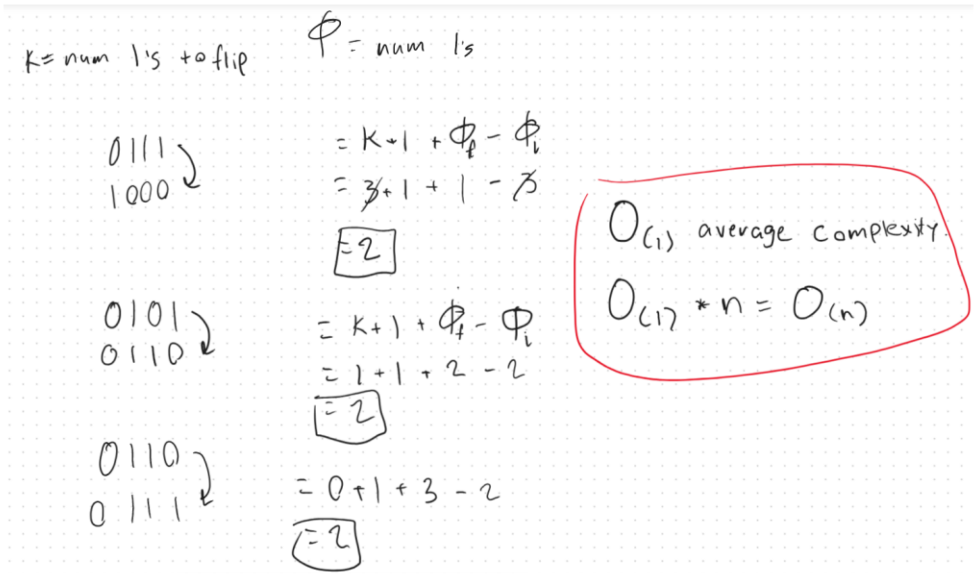
5. Aggregate:



Accounting:



Potential:



K= num 1's to flip

$\Phi$ = num 1's

$$0111 \to 1000$$

$$= K+1 + \Phi_f - \Phi_i$$

$$= \cancel{3}+1 + 1 - \cancel{3}$$

$$\boxed{=2}$$

$$0101 \to 0110$$

$$= K+1 + \Phi_f - \Phi_i$$

$$= 1+1 + 2 - 2$$

$$\boxed{=2}$$

$$0110 \to 0111$$

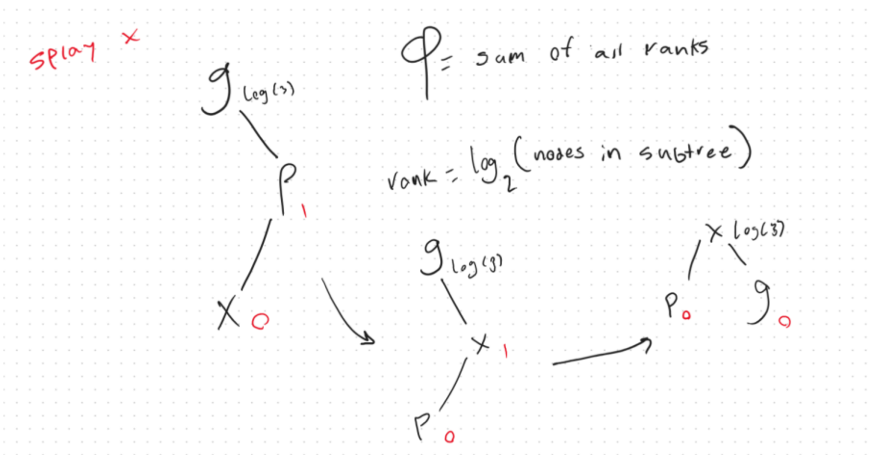
$$= 0+1 + 3 - 2$$

$$\boxed{=2}$$

$O(1)$ average complexity

$O(1) * n = O(n)$

6. Splay trees can use amortized analysis to prove a time complexity of O(log(n)). If you have a skewed tree, accessing the bottom most element will take O(n) time. That time can be “paid” for by giving us a more balanced tree. If you make the Potential function = the sum of all ranks, with a rank defined as = log2( size of subtree ), then you can see that the more our splay tree is skewed, the greater potential it has. We can use this greater potential to “pay” for expensive moves, and in turn, it allows the tree to be more balanced, causing less expensive operations.


splay x
g log(3)
p 1
x 0
$\Phi$ = sum of all ranks
rank = $\log_2$(nodes in subtree)
g log(3)
x 1
p 0
x log(3)
p 0
g 0