# ADVANCED ANALYSIS OF ALGORITHMS

Dr. Qaiser Abbas

Department of Computer Science & IT,

University of Sargodha, Sargodha, 40100, Pakistan

qaiser.abbas@uos.edu.pk

## AMORTIZED ANALYSIS

- Not just consider one operation, but a **sequence of operations** on a given data structure.
- **Average cost** over a sequence of operations.
- Probabilistic analysis:
  - Average case running time: **average over all possible inputs** for one algorithm (operation).
  - If using probability, called **expected running time**.
- Amortized analysis:
  - No involvement of probability
  - **Average performance on a sequence of operations**, even some operation is expensive.
  - Guarantee average performance of each operation among the **sequence in worst case**.

## THREE METHODS OF AMORTIZED ANALYSIS

- Aggregate analysis:
  - **Total cost of *n* operations/*n***
- Accounting method:
  - **Assign each type of operation** an (different) amortized cost
  - **overcharge** some operations,
  - store the overcharge as **credit** on specific objects,
  - then use the credit for compensation **for some later operations**.
- Potential method:
  - Same as accounting method
  - But store the credit as "**potential energy**" and as a whole.

## AGGREGATE ANALYSIS (STACK OPERATIONS)

```
MULTIPOP($S, k$)
while not STACK-EMPTY($S$) and $k > 0$
    POP($S$)
    $k = k - 1$
```

- Consider a sequence of ***n*** **PUSH, POP, MULTIPOP**.
  - The worst-case cost for single MULTIPOP in the sequence is $O(n)$, since the stack size is at most $n$.
  - Thus, the cost of the whole sequence is $O(n^2)$. Correct, but not tight.

## AGGREGATE ANALYSIS (STACK OPERATIONS)

- In fact, a sequence of $n$ operations on an initially empty stack cost at most $O(n)$. Why?

- Each object can be POP only once (including in MULTIPOP) for each time it is PUSHed. **#POPs are at most #PUSHs**, which is at most **n**.
- Thus, the average cost of an operation is $\boldsymbol{O(n)/n = O(1)}$.
- Amortized cost in aggregate analysis is defined to be average cost.

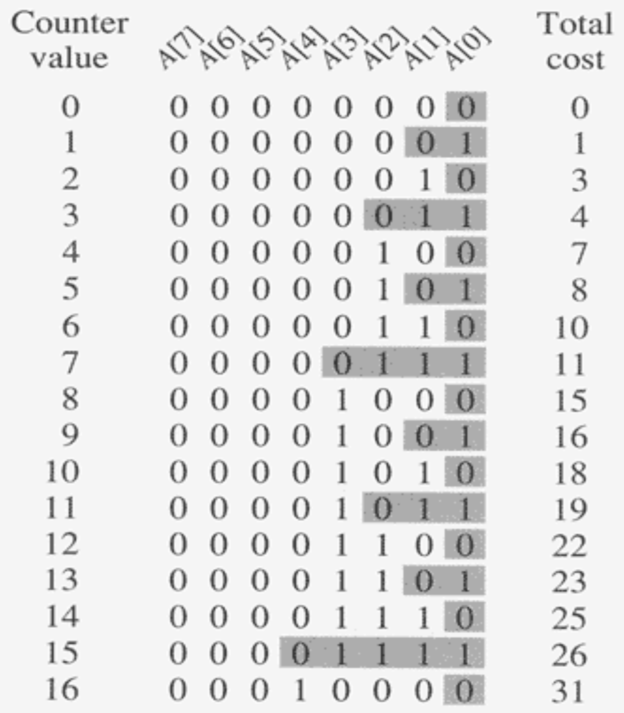
# AGGREGATE ANALYSIS (BINARY COUNTER)

```
INCREMENT(A)
1  $i = 0$
2  while $i < A.length$ and $A[i] == 1$
3      $A[i] = 0$
4      $i = i + 1$
5  if $i < A.length$
6      $A[i] = 1$
```

- Single execution of INCREMENT takes $O(k)$ in the worst case (when A contains all 1s) and k is total bits in A
- Sequence of $n$ executions takes $O(nk)$ in worst case (suppose initial counter is 0).
- This bound is correct, but not tight. The tight bound is $O(n)$ for $n$ executions.

# AGGREGATE ANALYSIS (BINARY COUNTER)

Observation: The running time determined by #flips but not all bits flip each time INCREMENT is called.



| 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 |

**Figure 17.2** An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is never more than twice the total number of INCREMENT operations.

A[0] flips every time, total $n$ times.
A[1] flips every other time, $\lfloor n/2 \rfloor$ times.
A[2] flips every forth time, $\lfloor n/4 \rfloor$ times.
….
for $i$=0,1,…,$k$-1, A[$i$] flips $\lfloor n/2^i \rfloor$ times.

Thus total #flips is $\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$
$< n\sum_{i=0}^{\infty} 1/2^i$
$=2n$.

## AMORTIZED ANALYSIS (DYNAMIC TABLE)

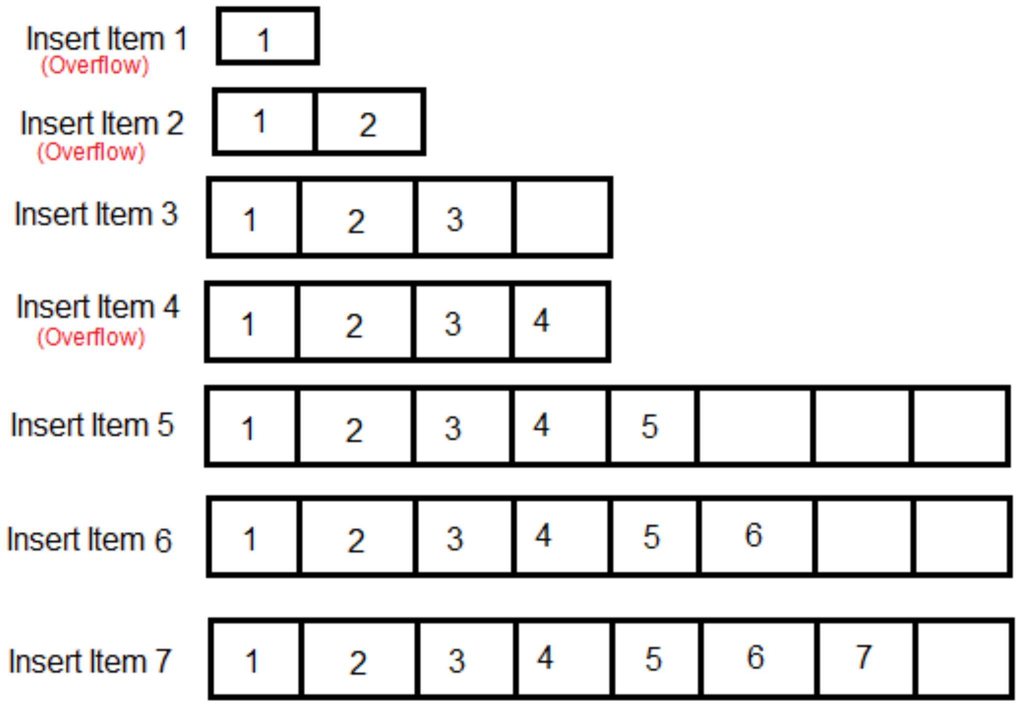
- Let us consider an example of a simple hash table insertions.
- How do we decide table size?
- There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.

## AMORTIZED ANALYSIS (DYNAMIC TABLE)

- The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase the size of table whenever it becomes full. Following are the steps to follow when table becomes full.
  - Allocate memory for a larger table of size, typically twice the old table.
  - Copy the contents of old table to new table.
  - Free the old table.
- If the table has space available, we simply insert new item in available space.

# AMORTIZED ANALYSIS (DYNAMIC TABLE)

Initially table is empty and size is 0

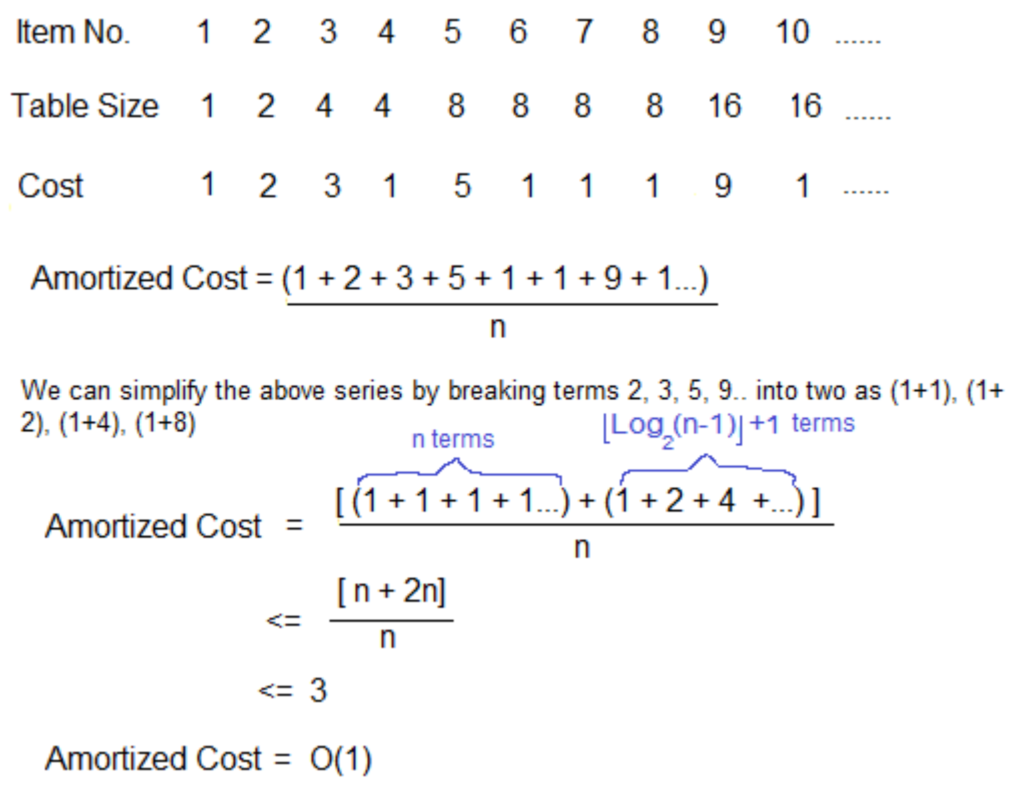


Next overflow would happen when we insert 9, table size would become 16

## AMORTIZED ANALYSIS (DYNAMIC TABLE)

- **What is the time complexity of n insertions using the previous scheme?**
  - If we use simple analysis, the worst-case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n * O(n) which is O($n^2$). This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take Θ(n) time.

## AMORTIZED ANALYSIS (DYNAMIC TABLE)



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Item No. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ...... |
| Table Size | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 | ...... |
| Cost | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9 | 1 | ...... |

$$\text{Amortized Cost} = \frac{(1+2+3+5+1+1+9+1\ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1+1+1+1\ldots)}^{\text{n terms}} + \overbrace{(1+2+4+\ldots)}^{\lfloor \text{Log}_2(n-1)\rfloor+1 \text{ terms}}]}{n}$$

$$<= \frac{[n+2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

- Using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in vectors in C++, ArrayList in Java.

## AMORTIZED ANALYSIS (ACCOUNTING METHOD)

- Idea:
  - Assign **differing charges** to different operations.
  - The amount of the charge is called **amortized cost.**
  - Amortized cost is **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.
- In aggregate analysis, all operations have same amortized costs but here different.

# ACCOUNTING METHOD (CONT.)

- Suppose **actual cost is $c_i$** for the *i*th operation in the sequence, and **amortized cost is $c_i'$** and then $\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$ **should hold**.
- **Average cost (per operation)** should be small using amortized cost, and total amortized cost is an upper bound of total actual cost as holds for all sequences of operations above.
- Total credit is $\sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$, which should be **nonnegative**. Moreover, $\sum_{i=1}^{t} c_i' - \sum_{i=1}^{t} c_i \geq 0$ for any ***t*>0**.

## ACCOUNTING METHOD (STACK OPERATIONS)

- Actual costs:
  - PUSH :1, POP :1, MULTIPOP: min($s$,$k$).
- Let assign the following amortized costs:
  - PUSH:2, POP: 0, MULTIPOP: 0.
- Similar to a stack of plates in a cafeteria.
  - Suppose $1 represents a unit cost.
  - When pushing a plate, use one dollar to pay the actual cost of the push and leave one dollar on the plate as credit.
  - Whenever POPing a plate, the one dollar on the plate is used to pay the actual cost of the POP. (same for MULTIPOP).
  - By charging PUSH a little more, do not charge POP or MULTIPOP.
- The total amortized cost for $n$ PUSH, POP, MULTIPOP is $O(n)$, thus $O(1)$ for average amortized cost for each operation.
- Conditions hold: total amortized cost ≥total actual cost, and amount of credits never becomes negative.

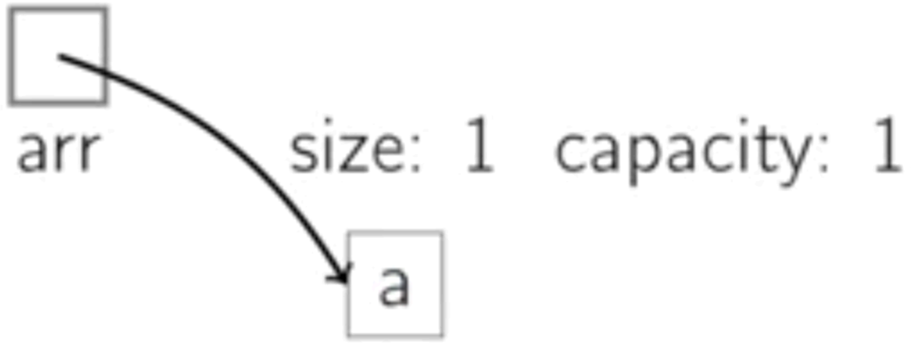
## ACCOUNTING METHOD (BINARY COUNTER)

- Let \$1 represent each unit of cost (i.e., the flip of one bit).
- Charge an amortized cost of \$2 to set a bit to 1.
- Whenever a bit is set, use \$1 to pay the actual cost, and store another \$1 on the bit as credit.
- When a bit is reset, the stored \$1 pays the cost.
- At most, one bit is set 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 $O(1)$.

# ACCOUNTING METHOD (HASH TABLES)

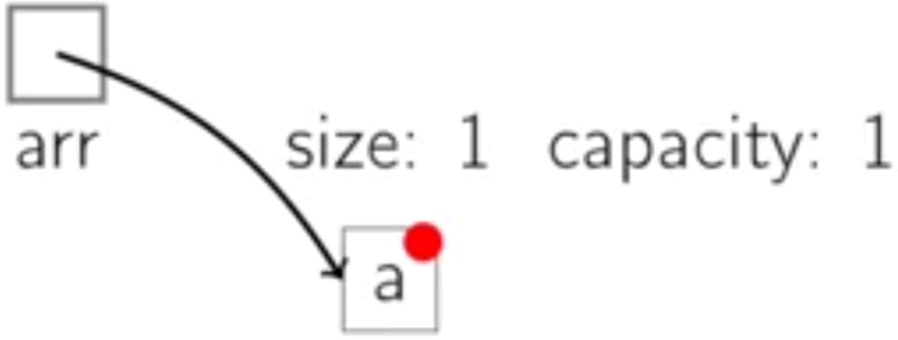
- Charge 3 for each insertion:
  - 1 token for each raw insertion
  - Resize needed: To pay for moving the elements, use the token that's present on each element that needs to move.
  - Place one token on newly inserted element, and one token capacity/2 elements prior.

# ACCOUNTING METHOD (HASH TABLES)



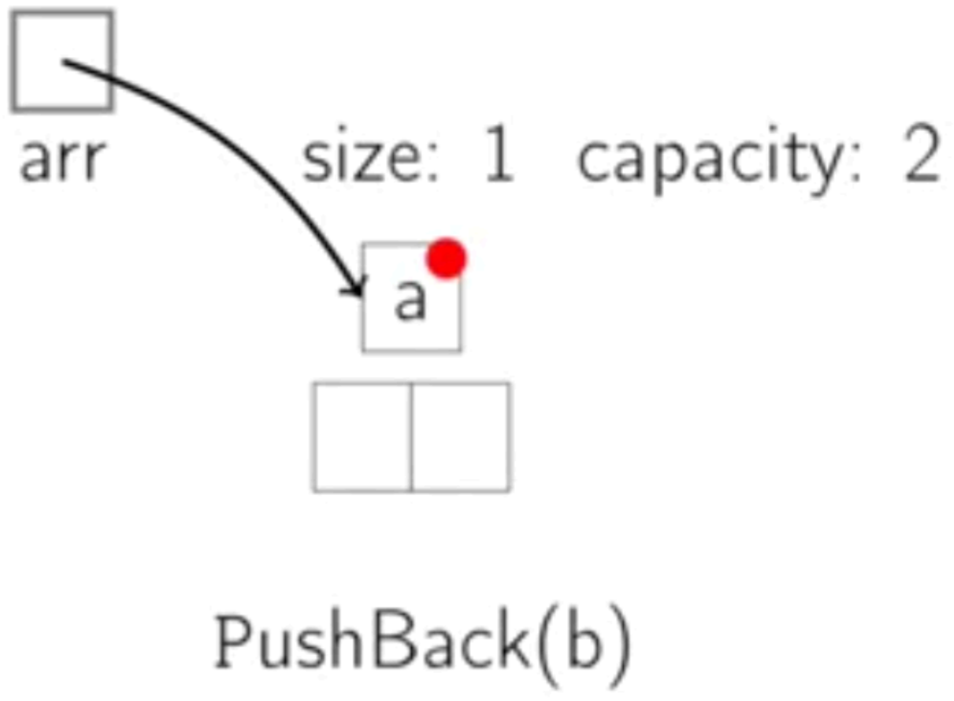
PushBack(a)

## ACCOUNTING METHOD (HASH TABLES)

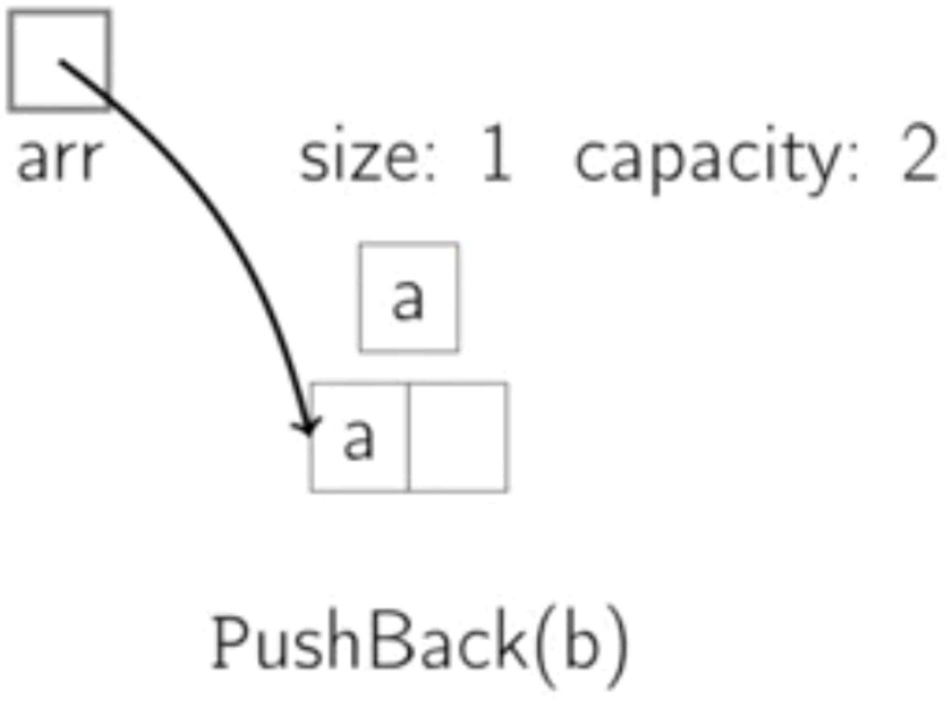


PushBack(a)

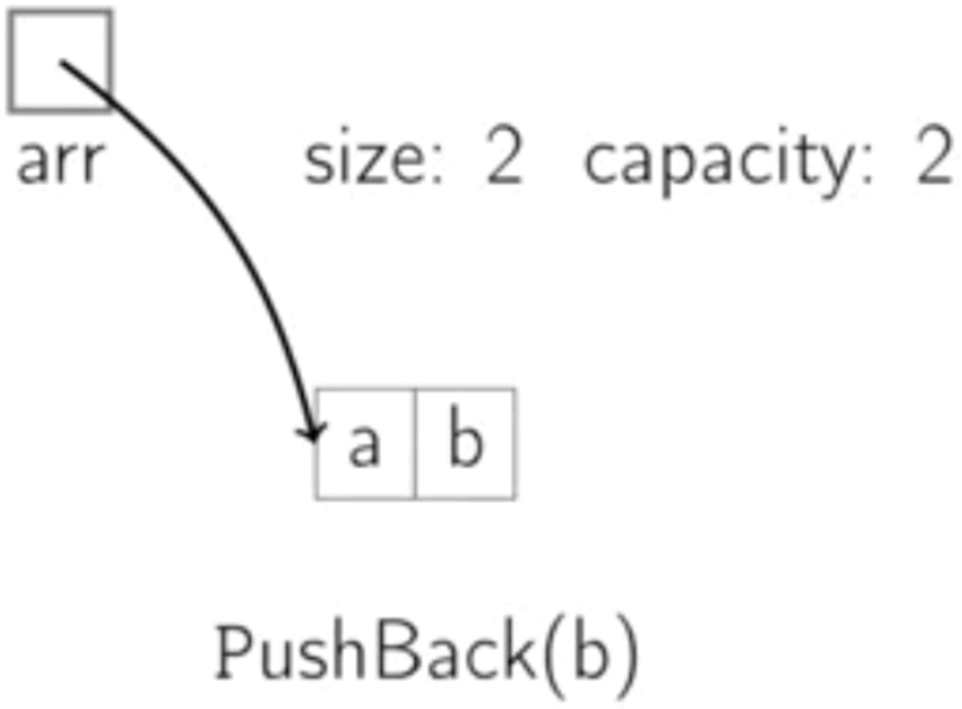
## ACCOUNTING METHOD (HASH TABLES)

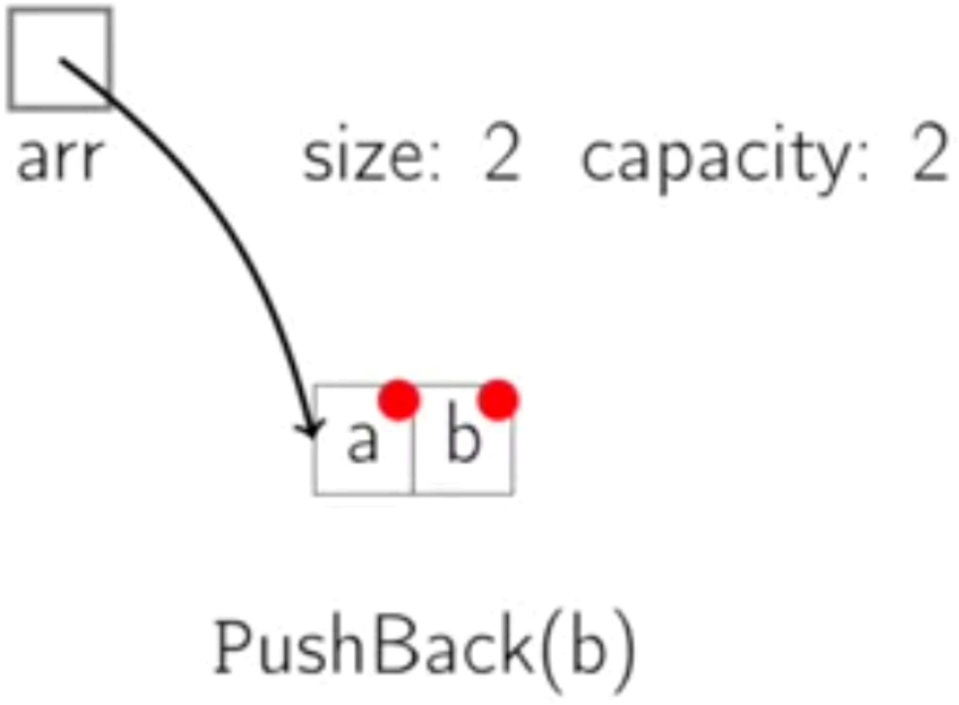


## ACCOUNTING METHOD (HASH TABLES)

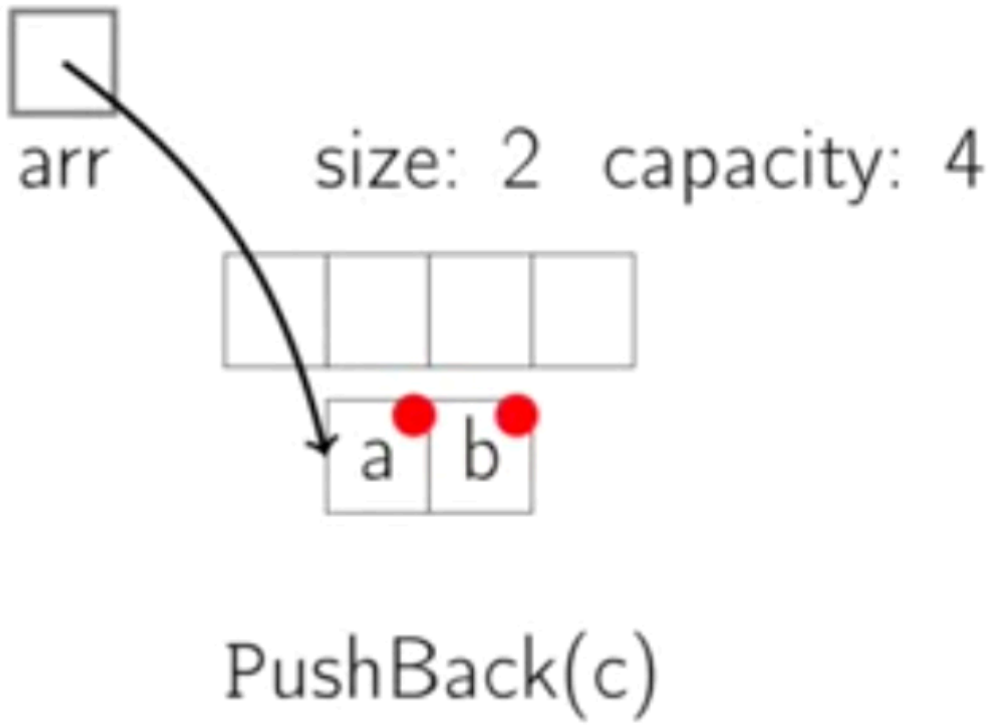


## ACCOUNTING METHOD (HASH TABLES)



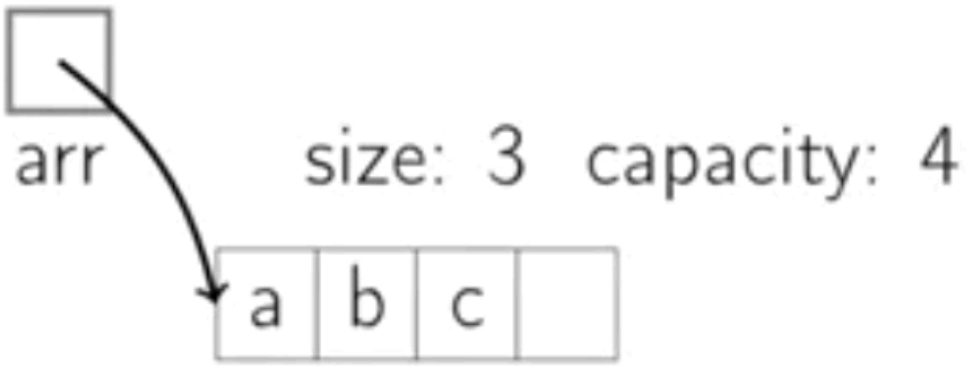


## ACCOUNTING METHOD (HASH TABLES)



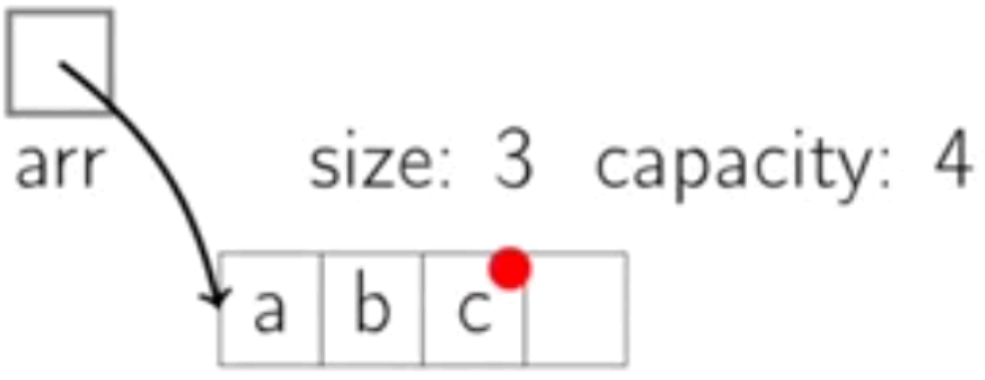
# ACCOUNTING METHOD (HASH TABLES)

## ACCOUNTING METHOD (HASH TABLES)



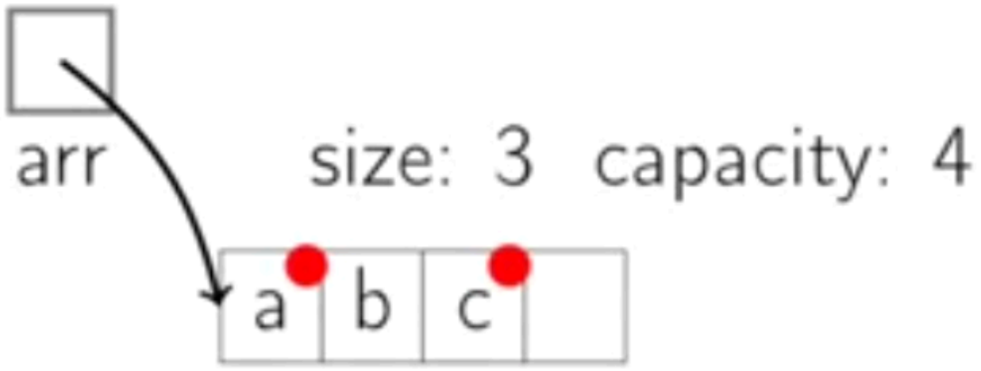
PushBack(c)

## ACCOUNTING METHOD (HASH TABLES)



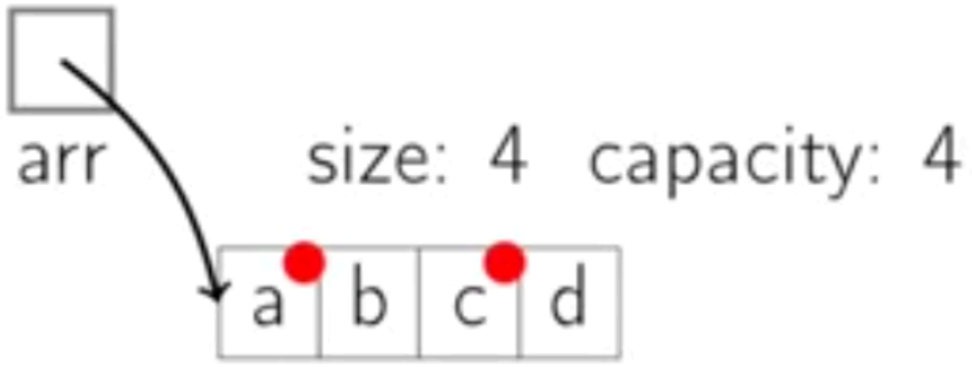
PushBack(c)

# ACCOUNTING METHOD (HASH TABLES)



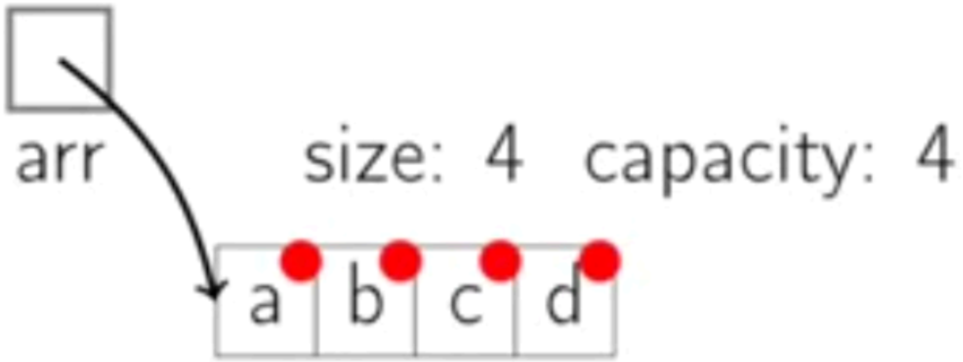
PushBack(c)

## ACCOUNTING METHOD (HASH TABLES)



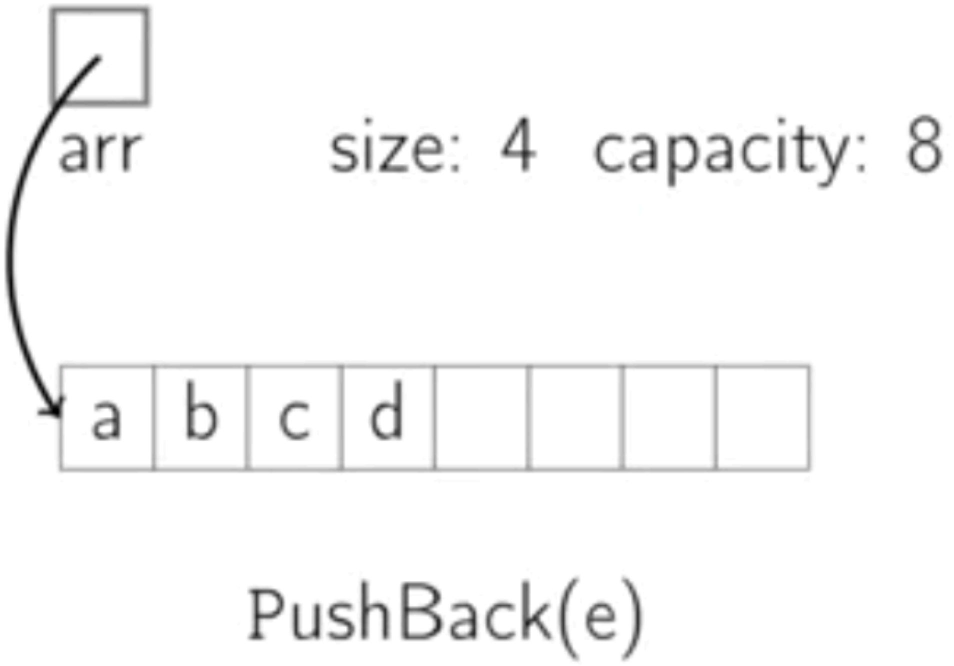
PushBack(d)

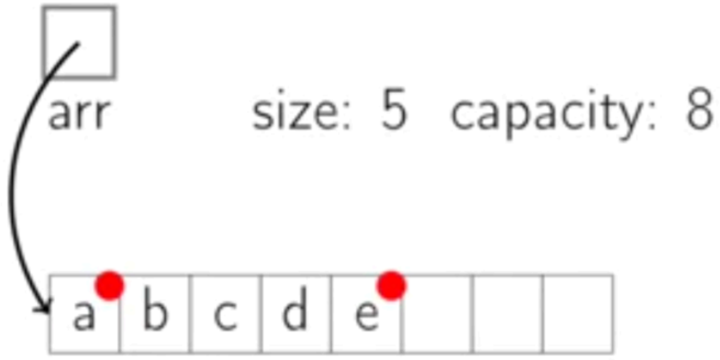
# ACCOUNTING METHOD (HASH TABLES)

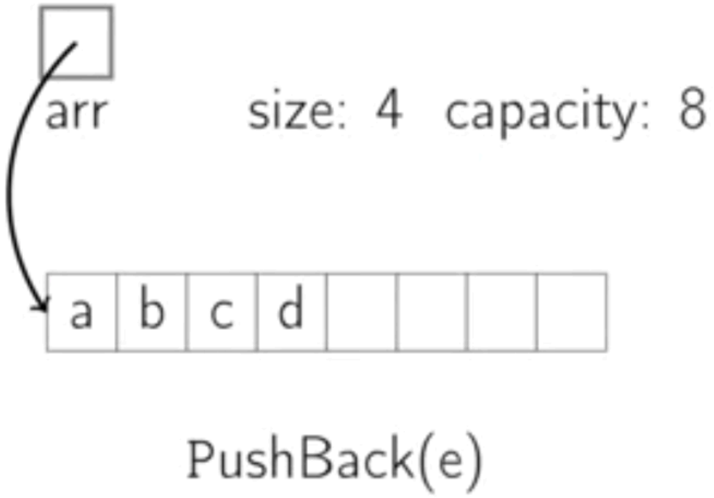


PushBack(d)

## ACCOUNTING METHOD (HASH TABLES)







## ACCOUNTING METHOD (HASH TABLES)

- O(1) amortized cost for each PushBack.

## POTENTIAL METHOD

- Same as accounting method: something prepaid is used later.
- Different from accounting method
  - The prepaid work not as credit, but as "potential energy", or "potential".
  - The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

# POTENTIAL METHOD (CONT.)

- 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 $\Phi : \{D_i\} \rightarrow$ R (real numbers)
- $\Phi(D_i)$ is called the potential of $D_i$.
- 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)
- $\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$

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

## POTENTIAL METHOD (CONT.)

- If $\Phi(D_n) \geq \Phi(D_0)$, then total amortized cost is an upper bound of total actual cost. But we do not know how many operations, so $\Phi(D_i) \geq \Phi(D_0)$ is required for any $i$.
- It is convenient to 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 actual cost).

## POTENTIAL METHOD: STACK OPERATION

- Potential for a stack is the number of objects in the stack. So $\Phi(D_0)=0$, and $\Phi(D_i) \geq 0$
- Amortized cost of stack operations:
  - PUSH:
    - Potential change: $\Phi(D_i) - \Phi(D_{i-1}) = (s+1)-s = 1$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+1=2$.
  - POP:
    - Potential change: $\Phi(D_i) - \Phi(D_{i-1}) = (s-1) - s = -1$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+(-1)=0$.
  - MULTIPOP($S$,$k$): $k'=\min(s,k)$
    - Potential change: $\Phi(D_i) - \Phi(D_{i-1}) = -k'$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k'+(-k')=0$.
- So amortized cost of each operation is $O(1)$, and total amortized cost of $n$ operations is $O(n)$.
- Since total amortized cost is an upper bound of actual cost, the worse case cost of $n$ operations is $O(n)$.

# POTENTIAL METHOD: BINARY COUNTER

- Define the potential of the counter after the *i*th INCREMENT is $\Phi(D_i) = b_i$, the number of 1's. clearly, $\Phi(D_i) \geq 0$.
- Let us compute amortized cost of an operation
  - Suppose the *i*th operation resets $t_i$ bits.
  - Actual cost $c_i$ of the operation is at most $t_i + 1$.
  - If $b_i=0$, then the *i*th operation resets all $k$ bits, so $b_{i-1}=t_i=k$.
  - If $b_i>0$, then $b_i=b_{i-1}-t_i+1$
  - In either case, $b_i \leq b_{i-1}-t_i+1$.
  - 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 $n$ operations is $O(n)$.
- Thus worst case cost is $O(n)$.

## POTENTIAL METHOD (HASH TABLE)

- Potential function $\Phi$ on states of a data structure
  - $\Phi(h_0) = 0$, where $h_0$ is the initial state of the data structure.
  - $\Phi(h_t) \geq 0$ for all states $h_t$ of the data structure.

## POTENTIAL METHOD (HASH TABLE)

- Sequence of $n$ operations taking actual times $c_0, c_1, c_2, \ldots, c_{n-1}$ and producing data structures $h_1, h_2, \ldots, h_n$ starting from $h_0$.
- The total amortized time is the sum of the individual amortized times:

$$(c_0 + \Phi(h_1) - \Phi(h_0)) + (c_1 + \Phi(h_2) - \Phi(h_1)) + \ldots + (c_{n-1} + \Phi(h_n) - \Phi(h_{n-1}))$$
$$= c_0 + c_1 + \ldots + c_{n-1} + \Phi(h_n) - \Phi(h_0)$$
$$= c_0 + c_1 + \ldots + c_{n-1} + \Phi(h_n)$$

- Amortized time for a sequence of operations overestimates of the actual time by $\Phi(h_n)$, which by assumption is always positive.
- Thus, the total amortized time is always an upper bound on the actual time.

# POTENTIAL METHOD (HASH TABLE)

- For dynamically arrays, we can use the potential function

$\Phi(h) = 2n - m$

  - $n$ is the current number of elements and $m$ is the current length of the array.

- If we start with an array of length 0, and allocate an array of length 1 when the first element is added, and thereafter double the array size whenever we need more space, we have $\Phi(h_0) = 0$ and $\Phi(h_t) \geq 0$ for all $t$.
- The latter inequality holds because the number of elements is always at least half the size of the array.

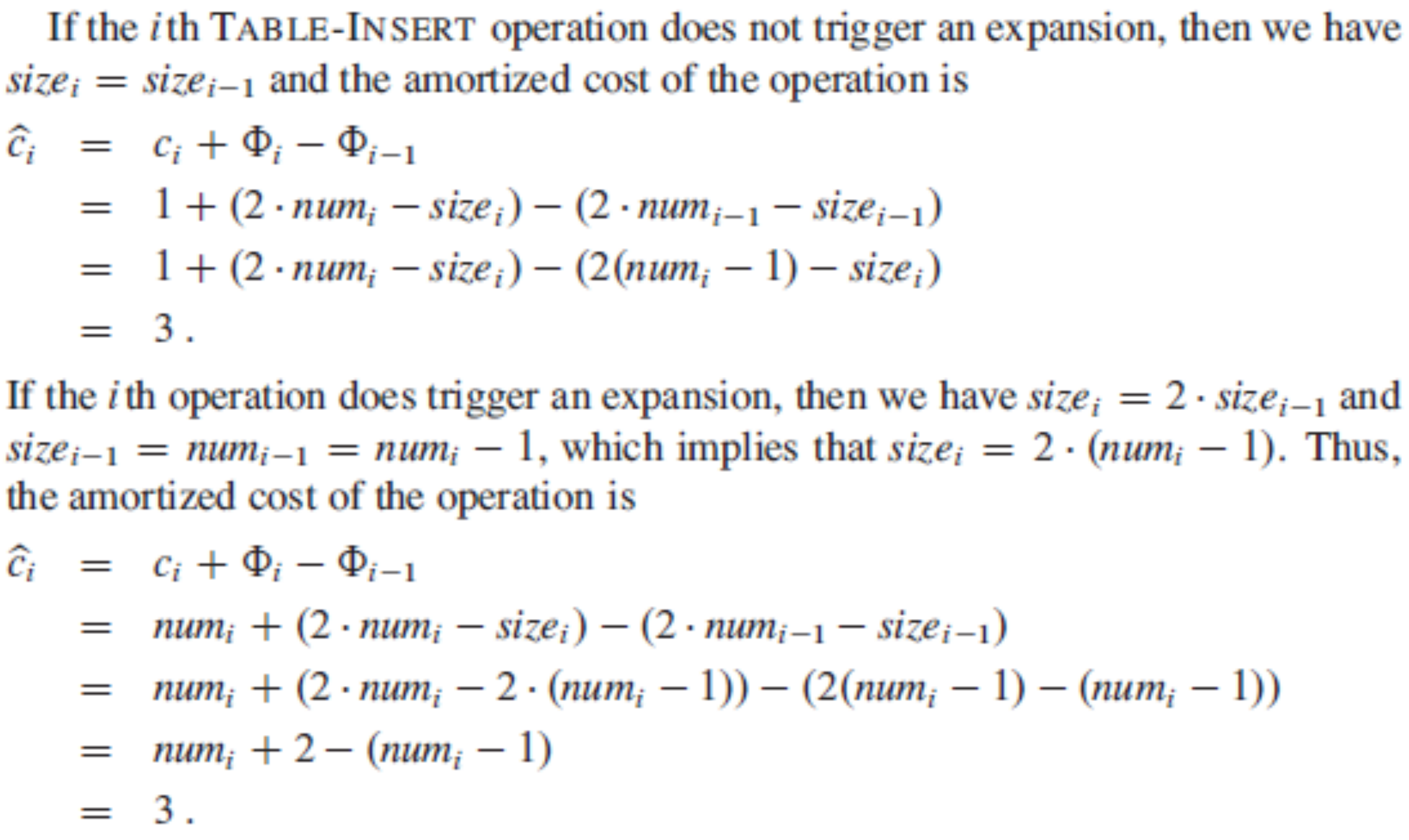
# POTENTIAL METHOD (HASH TABLE)

- Now we would like to show that adding an element takes amortized constant time. There are two cases.
  - If $n < m$, then the actual cost is 1, $n$ increases by 1, and $m$ does not change. The table does not expand and suppose that $n_i$=$num_i$ and $m_i$=$size_i$.
  - If $n = m$, then the array is doubled, so the actual time is $n + 1$. The table expands and suppose that $n_i$=$num_i$ and $m_i$=$size_i$.
- In both cases, the amortized time is O(1).
- The key to amortized analysis with the physicist's method is to define the right potential function.

## POTENTIAL METHOD (DYNAMIC TABLE)

- **Potential function**
- $\Phi(T) = 2 \cdot T.num - T.size$
- Initially, $T.num = T.size = 0 \Rightarrow \Phi = 0$.
- Just after expansion, $T.num = T.size / 2, \Rightarrow \Phi(T) = 0$.
- Just before expansion, $T.num = T.size \Rightarrow \Phi(T) = T.num \Rightarrow$ have enough potential to pay for moving all items.
- Need $\Phi \geq 0$, always.
- Always have
  - $size \geq num \geq ½\ size \Rightarrow 2 \cdot num \geq size \Rightarrow \Phi \geq 0$ .

# POTENTIAL METHOD (DYNAMIC TABLE)



If the $i$th TABLE-INSERT operation does not trigger an expansion, then we have $size_i = size_{i-1}$ and the amortized cost of the operation is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\
&= 3 .
\end{aligned}$$

If the $i$th operation does trigger an expansion, then we have $size_i = 2 \cdot size_{i-1}$ and $size_{i-1} = num_{i-1} = num_i - 1$, which implies that $size_i = 2 \cdot (num_i - 1)$. Thus, the amortized cost of the operation is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3 .
\end{aligned}$$

## POTENTIAL METHOD DYNAMIC TABLE

- Continue reading Dynamic Tables from the textbook

# ASSIGNMENT # 4

***17.1-2***

Show that if a DECREMENT operation were included in the $k$-bit counter example, $n$ operations could cost as much as $\Theta(nk)$ time.

***17.2-3***

Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of $n$ INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter. (*Hint:* Keep a pointer to the high-order 1.)

***17.3-3***

Consider an ordinary binary min-heap data structure with $n$ elements supporting the instructions INSERT and EXTRACT-MIN in $O(\lg n)$ worst-case time. Give a potential function $\Phi$ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works.