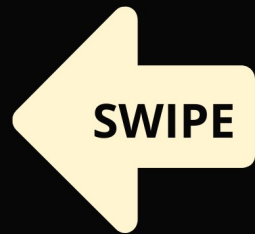




#ASLI ENGINEERING

Resizing a Hash Table



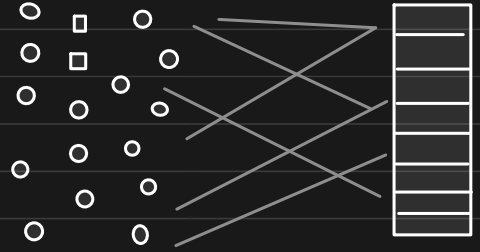
BY

ARPIT BHAYANI

Resizing a Hash Table

$$\text{Load Factor } \alpha = \frac{\# \text{ keys}}{\# \text{ slots}}$$

Performance of hash table degrades as load factor increases



and hence, at a certain threshold,
to maintain the performance of our hash table
we have to **resize**.

When should we **resize**?

Resize when load factor is "too high"

* Resizing takes time

and it depends on the underlying structure
and algorithm

We cannot be too aggressive, nor too lenient



we would waste too much
time doing resizing



performance
will degrade

A common decision,

resize when $\alpha = 0.5 \rightarrow$ array is half filled

But we can make this logic as complex as possible

How to resize?



1. create new array - $len = m_{new}$
2. copy elements from old to the new array
3. delete old array

* allocating new array and moving keys takes time

If you are implementing it in C language, 'realloc'
the most common way to do this is

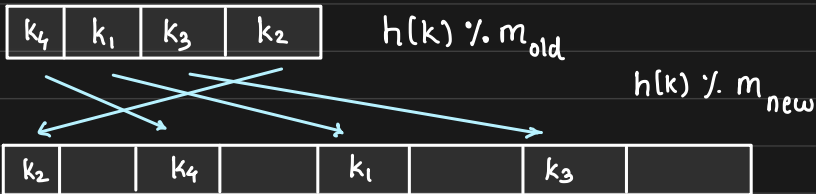
Note: Time taken to resize is $t \propto \text{keys}$

A typical strategy to resize is to always double the array

↳ during insert if we trigger resize

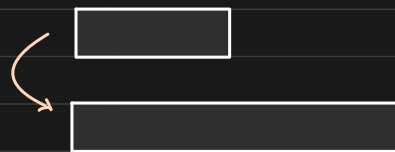
we re-allocate a new array of $2x$ size and rehash keys

* we may not be able to use 'realloc' call,
as we would need to rehash keys as per the new size $(\text{mod } m)$



Why do we always double?

It is a common practice that whenever we need to resize a hash table, we always double



↪ growing by a constant factor

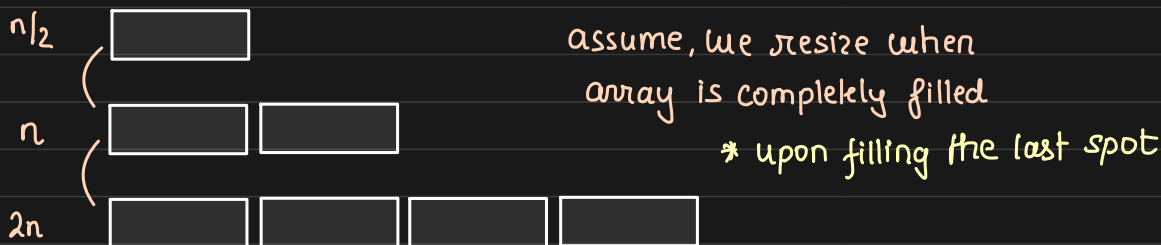
1. Say, we always increase by 1

			Operation
insert a	<div>a</div>	← allocate array and insert a	1
insert b	<div>a</div> <div>b</div>	← allocate array of size 2 copy 1 element, insert b	2
insert c	<div>a</div> <div>b</div> <div></div>	allocate array of size 3 copy 2 elements, insert c	3
	⋮		
insert z	<div>a</div> <div>b</div> <div>c</div> <div>d</div> <div>e</div> <div>f</div> <div>g</div> <div>h</div> <div>i</div> <div>...</div> <div>y</div> <div>z</div>		

- reallocate array of size 'n' 'n' operations
- copy all (n-1) elements from old to new
- insert n^{th} element

$$\text{Total ops} = 1 + 2 + \dots + n = \frac{n(n-1)}{2} = O(n^2)$$

2. Say, we double everytime





when we insert $n/2$ elements in array of size $n/2$



let's go from $n \rightarrow 2n$

To get to the next resize, we need to add $n/2$ elements

First $n/2 - 1$ takes $O(1)$, the final element

- $O(1)$ to insert 
- $O(2n)$ for resize 
- $O(n)$ for copying elements

$$\text{Total operations} = \frac{n}{2} - 1 + 1 + 2n + n = \frac{7n}{2} = O(n)$$

Why always a power of 2?

The underlying array of the hash table
has the length = power of 2

Because it is efficient, but how?

Hash table works on hash function

key $k_i \rightarrow f \rightarrow i \rightarrow i \% m \rightarrow \text{index}$



MOD operation is super-important as it bounds the integer
output from hash function to a range that fits in array

But, MOD is really expensive

internally it does division, and captures the remainder
can we do it faster?

Say, $m = 4$

hash key	mod(m)	index
1	$\% 4$	1
2	$\% 4$	2
3	$\% 4$	3
4	$\% 4$	0
5	$\% 4$	1
6	$\% 4$	2
7	$\% 4$	3

Let's use bitwise AND to get the same output

Say, $m = 4 = 2^2$, what if we AND it with $2^2 - 1 = 3$

$$\begin{array}{r} 1 \cdot 4 = 1 \quad 1 \text{ \& 3} \\ \begin{array}{r} 0001 \\ \oplus 0011 \\ \hline 0001 \end{array} \end{array}$$

$$\begin{array}{r} 2 \cdot 4 = 2 \quad 2 \text{ \& 3} \\ \begin{array}{r} 0010 \\ \oplus 0011 \\ \hline 0010 \end{array} \end{array}$$

$$\begin{array}{r} 3 \cdot 4 = 3 \quad 3 \text{ \& 3} \\ \begin{array}{r} 0011 \\ \oplus 0011 \\ \hline 0011 \end{array} \end{array}$$

$$\begin{array}{r} 4 \cdot 4 = 0 \quad 4 \text{ \& 3} \\ \begin{array}{r} 0100 \\ \oplus 0011 \\ \hline 0000 \end{array} \end{array}$$

$$\begin{array}{r} 5 \cdot 4 = \quad 5 \text{ \& 3} \\ \begin{array}{r} 0101 \\ \oplus 0011 \\ \hline 0001 \end{array} \end{array}$$

$$\text{MOD } m = \text{AND}(m-1), \quad m = 2^k$$

AND operation is significantly faster than division and hence performance gain!

- $2^k - 1$ has lower bits 1 and higher 0
- AND acts as a filter
 - ↳ zeros will keep result bounded
 - ↳ ones will percolate set bits

Shrinking a Hash Table

If keys are deleted from the hash table, then it does not make sense to keep them blocked



Hence, we shrink the table

* we grow when $\alpha \rightarrow \frac{1}{2}$

if table has n slots, it would have $n/2$ elements (max)

$n=16, e=7$

if we shrink, it would have $n/2$ slots

$n=8, e=7$ ← poor performance, immediate resize as load factor = 1

hence, it can have at max $n/4$ elements

$n=8, e=3$ ← consistent performance

But, one more insertion will cause a resize

Hence, we need to shrink when #elements are such

that few insertions after shrink would not cause a resize

$n=16, e=2$

hence we would shrink when

there are $n/8$ elements

Summary

- resizing is important to maintain performance
- resizing is costly
- hash table is always doubled
- hash table has 2^n slots for faster compute
- grow when $\alpha \rightarrow \frac{1}{2}$
- shrink when $\alpha \rightarrow \frac{1}{8}$