



Lab 8

Strategy 3 for Memory Allocation

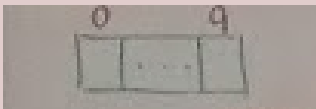
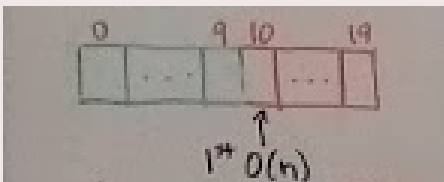
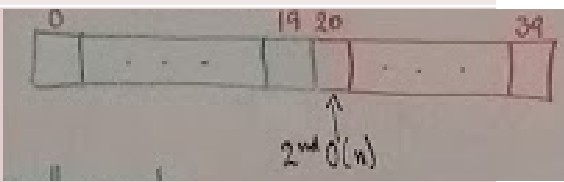
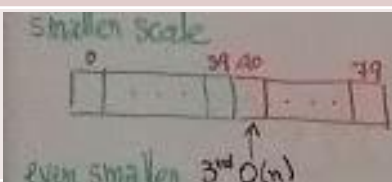
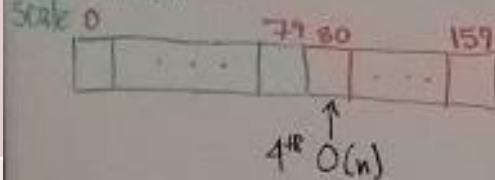
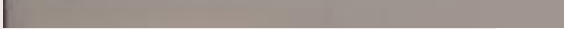
- How amortization works! -

How amortization works!

- ▶ The table shown on the next slide illustrates the effect of using the **third strategy** on the time efficiency of the function `append (...)`
 - ▶ Remember, in the third strategy, we implement the function `append (...)` as follows:
 - ▶ When the array is full (i.e., when `len == reserved`), we allocate memory for our array by calling `realloc(2 * reserved)`, hence doubling the size of our array every time our array runs out of space
- ▶ The table explains why the time efficiency of the function `append (...)` is ***amortized* $O(1)$**

How amortization works!

Called
"reserved"
in Lab 8

Array size (n) (when we start appending)	Time efficiency of each append	Ratio: # of append's in $O(1)$ Total # of append's	Our array as it grows
10	Each first 10 points (indices 0 to 9) appended in $O(1)$	$\frac{10 \times O(1)}{10 \text{ appends}} = 1.0$	
20	11 th point (index 10) appended in $O(n)^*$ + each next 9 points appended in $O(1)$	$\frac{19 \times O(1)}{20 \text{ appends}} = 0.95$ ($19 \times O(1) + 1 \times O(n)$)	
40	21 th point (index 20) appended in $O(n)^*$ + each next 19 points appended in $O(1)$	$\frac{38 \times O(1)}{40 \text{ appends}} = 0.95$ ($38 \times O(1) + 2 \times O(n)$)	
80	41 th point (index 40) appended in $O(n)^*$ + each next 39 points appended in $O(1)$	$\frac{77 \times O(1)}{80 \text{ appends}} = 0.9625$ ($77 \times O(1) + 3 \times O(n)$)	
160	81 th point (index 80) appended in $O(n)^*$ + each next 79 points appended in $O(1)$	$\frac{156 \times O(1)}{160 \text{ appends}} = 0.975$ ($156 \times O(1) + 4 \times O(n)$)	
320	161 th point (index 160) appended in $O(n)^*$ + each next 159 points appended in $O(1)$	$\frac{315 \times O(1)}{320 \text{ appends}} = 0.984375$ ($315 \times O(1) + 5 \times O(n)$)	

* realloc(...)
is called
and it may
have to
copy old
array into
new array
so: $O(n)$

Explaining the columns in our table

- Column 1 lists the size of the array when we start appending **n** elements (points) to the array
 - Note that we often express the number of elements as **n** in such complexity analysis
 - Note that, in Lab 8, the size of the array is stored in the variable `reserved`
 - In our explanation, the size of array starts at 10 and doubles on each row. Note that in our code (t1.c, t2.c), the size of the array starts at 0, not 10.
- Column 2 computes the time efficiency of appending each of the **n** elements (points) to the array
 - For example, when the size of array is 10, i.e., when the array has 10 cells (have a look at the diagram of the array), appending each of the first 10 points to the array takes $O(1)$ to do (i.e., constant time)
 - The 11th point (on the next row) will, however, take $O(n)$ time since a call to `realloc` will need to be made and the worst case scenario of `realloc` is to allocate a new memory space of 20 cells for the whole array and copy all existing 10 points to this new memory location $\rightarrow O(n)$
 - Note that `realloc` may be able to simply add 10 cells to the already allocated 10 cells in the array (i.e., best case scenario), but we cannot be sure of this so we perform our analysis using the **worst case scenario**
 - Once 20 cells have been allocated and the 11th point (at index 10) is appended to array, each of the next 9 points (indices 11 to 19) is appended to the array in $O(1)$ time
- Column 3 computes the ratio of the number of points appended in $O(1)$ over the total number of points appended to the array
 - As you can see this ratio, after moving slightly away from 1 (0.95 for $n = 20$, 0.95 for $n = 40$), starts increasing towards 1 again (0.9625 for $n = 80$, 0.975 for $n = 160$ and 0.984375 for $n = 320$) as the number of points appended to the array increases

Conclusion:

- As the size of the array increases to infinity, the number of append's done in $O(1)$ increases (demonstrated by the ratio moving towards 1)
 - We therefore say that the time efficiency of the function `append(...)` is **amortized $O(1)$**
 - The term “amortized” is used to express the fact that the effect of the $O(n)$ time efficiency required to append a small number of elements (which increases slowly as n goes to infinity) is spread over an increasing* number of elements requiring $O(1)$ time to append
 - This reduces the effect of the $O(n)$ time efficiency
- *Note: This increasing number of elements requiring $O(1)$ to append increases more rapidly than the increasing number of elements requiring $O(n)$ to append