Data Structures

- Specialized format for organizing, processing, retrieving & storing data
- Different structures have different characteristics.

  eg: some take more memory but can be accessed very quickly.
  - some have quicker inserts, some quicker removals

Data

- Stored in memory (Typically hexadecimal address) / easier representation
- 1 Byte = 8 bits, 1 nibble = 4 bits = 1 hex digit

Fixed Array

- static memory allocation: size must be predefined & cannot be altered.
- Array elements are stored in consecutive memory locations

my_arr → 0X01 (reference)

← contigious memory allocation.

* Buffer overflow → Amount of data in buffer exceeds storage capacity
  Extra data overflows to adjacent memory locations, corrupting data there.

* Buffer : Region in memory used to store data temporarily while its being moved

Fixed Array complexity

Insertion

Random (any index) : O(n) : Need to shift all other elements
Front : O(n) ——— downwards.
Back : O(1) : No shifting required.

Delete :

Random (any index) : O(n) : Need to shift back.
Front : O(n) ———
Back : O(1)

Search :

Unsorted : O(n) : check every element
sorted : O(log n) : Divide & conquer. (Binary Search Algo)

Access Time :

O(1) : By providing index, can access value instantaneously.

Binary Search Algorithm.

- only works for sorted structure.
- start search at middle., if middle = desired (done)
  if middle < desired (discard left half)
  else discard right half
  repeat

Circular array

- more efficient than fixed array (insert & delete @ front & back more efficient)
  from O(n) → O(1)
  fixed arr circular arr.
- Algorithm slightly more complex
- using modulo operation & front & back markers
- insert & delete random & search unsorted still O(n)

Modulo pattern:

0 % 3 = 0	0 % 5 = 0
1 % 3 = 1	1 % 5 = 1
2 % 3 = 2	2 % 5 = 2
3 % 3 = 0	3 % 5 = 3
4 % 3 = 1	4 % 5 = 4
5 % 3 = 2	5 % 5 = 0
⋮	

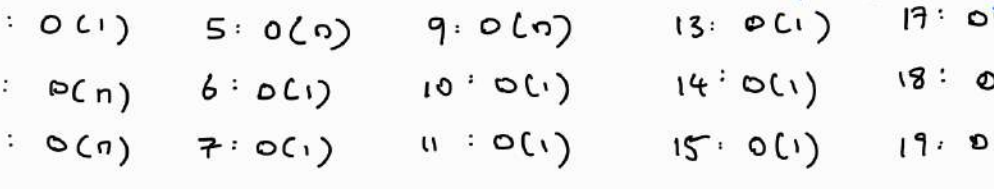
Pattern: Numbers returned will always be between 0 & n-1 where n is mod number.
(same idx as array)

eg: if array size = 7,
then use mod 7, n = 7

Circular Array

current idx of front marker.

Front marker : Front = (f - 1) % <Arr_size> ← % Arr_size to bounce back to idx 9 whenever idx < 0.
10 in this eg. ∵ (f - 1)



Back marker : Back = (b + 1) % <Arr_size> ← % Arr_size to bounce back to idx 0 whenever idx > 9. ∵ (b + 1)
10 in this eg.

Modulo of negative Numbers:

-6 % 5 = 4

1. Take the next number smaller than -6 divisible by 5
   i.e -10 ← smaller than -6 & divisible by 5
2. Subtract number from original:
   i.e -6 - (-10) = 4 //

Dynamic Arrays

- Array size can grow according to use.

* can implement dynamic + circular array (best of both world)

Typical flow:

1. create initial array with predetermined size.
2. fill up array till full
3. create new array, twice size of initial array
4. copy contents from initial arr to new arr.
5. Fill up new arr. & repeat once full.

⟨1, 2, 4, 8, 16, 32, 64 ...

↓ don't have to resize array frequently as array size doubles higher.

* cannot simply increase size of initial array since memory was pre allocated. & adjacent memory might already been used.

O(n): due to copying old array content to new array

1 : O(1)	5 : O(n)	9 : O(n)	13 : O(1)	17 : O(n)	21 : O(1)
2 : O(n)	6 : O(1)	10 : O(1)	14 : O(1)	18 : O(1)	22 : O(1)
3 : O(n)	7 : O(1)	11 : O(1)	15 : O(1)	19 : O(1)	23 : O(1)
4 : O(1)	8 : O(1)	12 : O(1)	16 : O(1)	20 : O(1)	24 : O(1)

insertion: O(log n)