Algorithm and Data Structures

"Problem Solving is a skill that can be developed via practice"

→ Define the Problem.
- what exactly is the problem we are trying to solve?

→ Identify the problem
- How and why did this problem happen?

→ what are the possible solutions?
- The ideal solution could be one of the many possible solutions.

→ A decision is to be made.
- Any decision is usually is better than no decision.

- 80% of problems should be solved at the moment they come up.
Only 20% will need time & research.

→ Assign responsibilities to carry out the decision.
- If a team, who will do what and when.
- If alone, still decide what & when.

→ Set a schedule.
- without a schedule & deadline, its just a discussion.

Core Components of Computational Thinking.

→ Decomposition
→ Break down complex problems

into smaller, simpler problems.

→ Pattern recognition.
- Make connections between similar problems & experience

→ Abstraction
- Identify important information and ignore irrelevant details.

→ Algorithm
→ Sequential rules to follow in order to solve a problem.

Algorithm

→ A "finite sequence" of "well defined" computational steps that turns form "input" into "output"

Basic construct of algorithm

→ Linear Sequence: — statements that follow one after the other.

→ Conditional — "if then else"

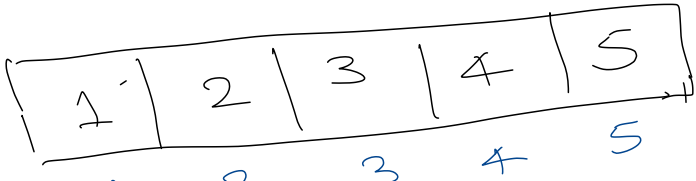
→ Loop — sequence of statements that are repeated number of times.

Data Structure

A data structure is a way to store and organise data in order to facilitate access and modifications.

Array ⇒ Sequential data structure.
Linear data structure.

Stores data in a sequential manner.



1 2 3 4 5 Index/
1 2 3 4 Subscript
0 1 2 3

All element are stored in a single memory block.

Every element of array is of

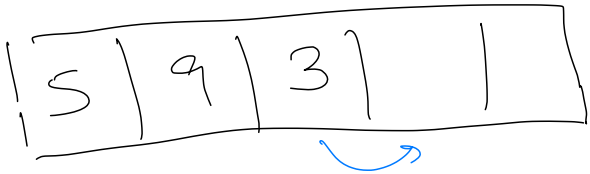
same type.

Benefit :- Random access.

Fast

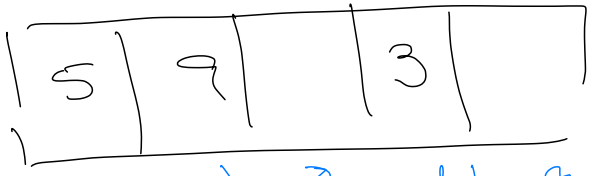
start location + i * size of array element type.

Insert / Delete is inefficient in array.

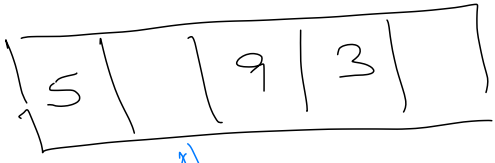


Insert between 5 & 9, 1.

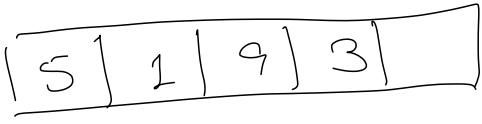
Shift 3 to right



Shift 9 to right



Insert 1



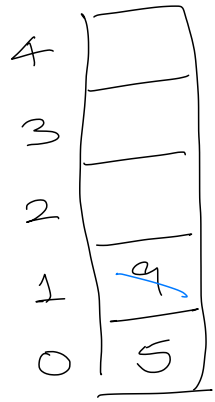
Stack

↓

LIFO

Last In
First Out

top ← add
& delete
done with
respect to top.



top → ~~1~~ ~~0~~ ~~-1~~

Push(5)

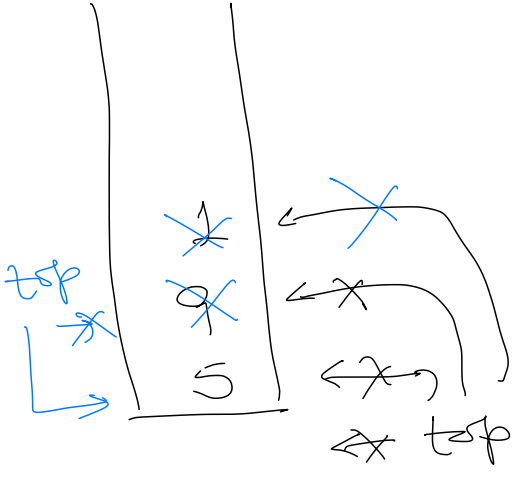
Push(9)

Pop() → 9

Push(elem) → add element to stack.

→ Make space at top.

→ Store element at top.



Push(5)
Push(9)
Push(1)

Pop() → removes element from stack.

Psp() → 2
Psp() → 9

→ Fetch element at top
→ Set top to previous element
→ Return the fetched element.

IsEmpty()
→ if top stores a element then stack is not empty
else stack is empty.

IsFull()
→ if stack has space for atleast one more element then stack is not full
else stack is full.

ADT → Abstract Data Type } Defines WHAT & NOT HOW.

Define ADT in JAVA
→ Use Keyword interface.

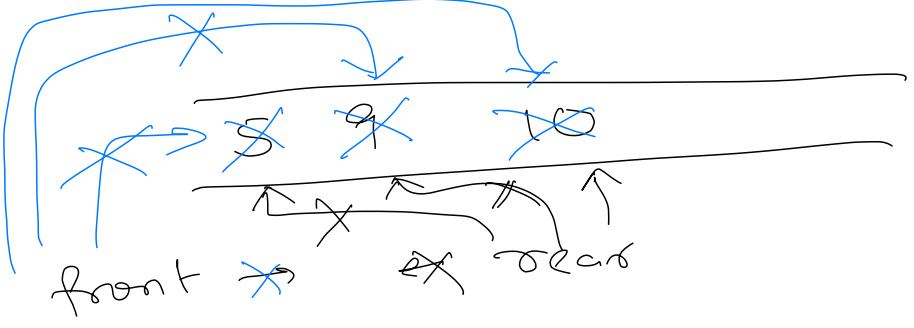
```
interface StackInf {
    void Push (int);
    int Pop ();
    boolean IsFull ();
    boolean IsEmpty ();
}
```

AddQ(elem)
- Make space at rear
- Store element at rear.

Queue
- FIFO
  - First In
  - First Out

front → Remove element from front of queue

rear → Adding of element is done at rear of queue.



AddQ(5)
AddQ(9)

Add Q(10)

Delete Q()
- Move front to next element
- Remove element from front.

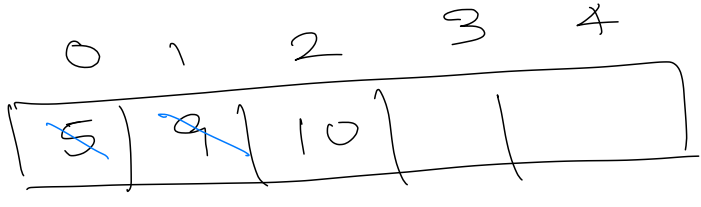
Delete Q() → 10

Delete Q() → 5

Delete Q() → 9

IsEmpty()
- If front equals rear then queue is empty
- else queue is not empty.

IsFull()
- If no space after rear then queue is full
- else queue is not full.



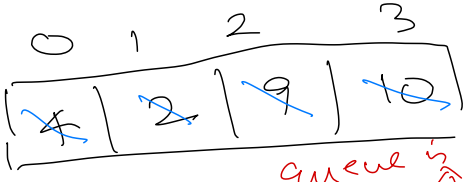
front → ~~-1~~ 0 1

rear → ~~-1~~ 0 1 2

Add Q(5)
Add Q(9)
Add Q(10)

Delete Q() → 5 Delete Q() → 9

Linear Queue → Has problem that Queue can be empty & full at the same time.



Add Q(4)
Add Q(2)
Add Q(9)
Add Q(10)

queue is empty

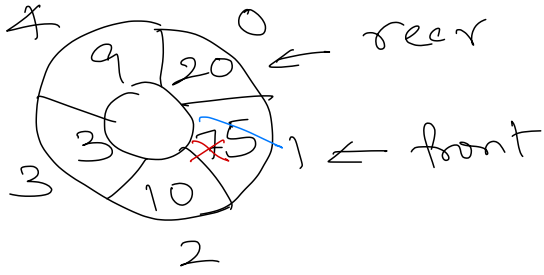
front → ~~-1~~ 0 ~~1~~ ~~2~~ 3
rear → ~~-1~~ 0 ~~1~~ 2 3

↑ queue is full.

Delete Q() → 4
Delete Q() → 2
Delete Q() → 9
Delete Q() → 10

Circular Queue

N = Array Size = 5



front → 0
rear → 0

front → -2
rear → ~~-1~~ 0 ~~1~~ ~~2~~ ~~3~~ ~~4~~ 0

rear → 4

Increment rear by 1 MOD N

rear → 5 MOD 5 ⇒ 0

↓

Value remains in range of 0 to N-1

Add Q(5) rear → 0

rear = (rear + 1) MOD N

rear → 1

Add Q(10) rear → ~~1~~ 2

DeleteQ() → front → 0 ⇒ 5

front = (front + 1) MOD N

front → 1

Add Q(3) rear → ~~2~~ 3

Add Q(9) rear → ~~3~~ 4

Add Q(20) rear → ~~4~~ (5 MOD 5) = 0

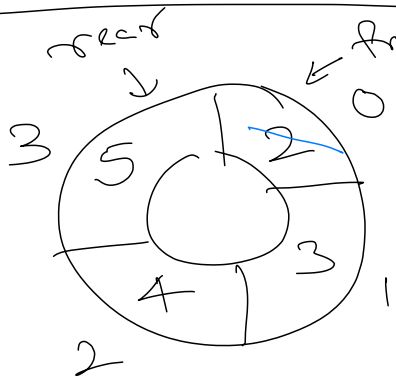
Add Q(7) rear → ~~0~~ ~~1~~

(rear + 1) MOD N equals front

↑↑

queue full in circular Q.

→ In circular queue, we end up storing max (N-1) elements.



N = 4

front = ~~-1~~ 0

rear = ~~-1~~ ~~0~~ ~~1~~ 2 3

Add Q (2)
Add Q (3)
Add Q (4)
Add Q (5)

if front equals -1 AND
rear equals last element then
queue full

Delete Q() → 2

Add Q (10) ✗ Queue is full.