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"

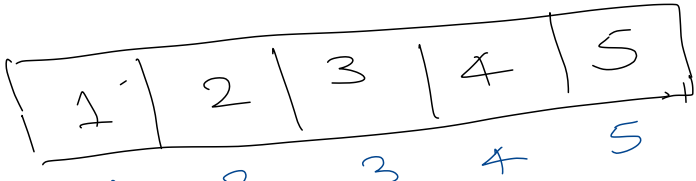
→ Loop — sequence of statements that are repeated number of times.

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

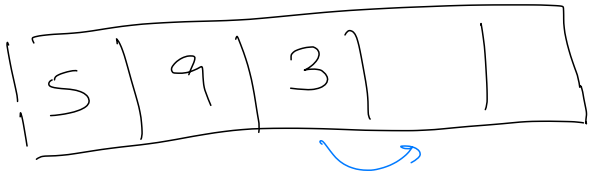
same type.

Benefit :- Random access.

Fast

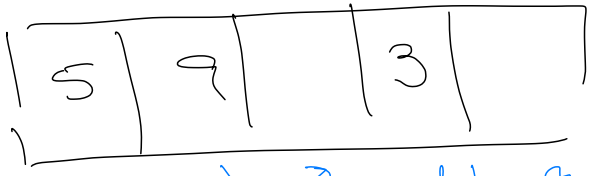
start location + i * size of array element type.

Insert / Delete is inefficient in array.

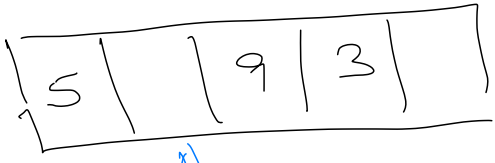


Insert between 5 & 9, 1.

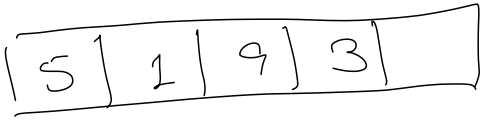
Shift 3 to right



Shift 9 to right



Insert 1



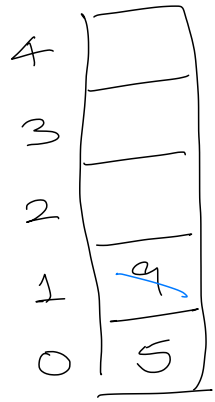
Stack

↓

LIFO

Last In
First Out

top ← add
& delete
done with
respect to top.



top → ~~1~~ ~~0~~ ~~-1~~

Push(5)

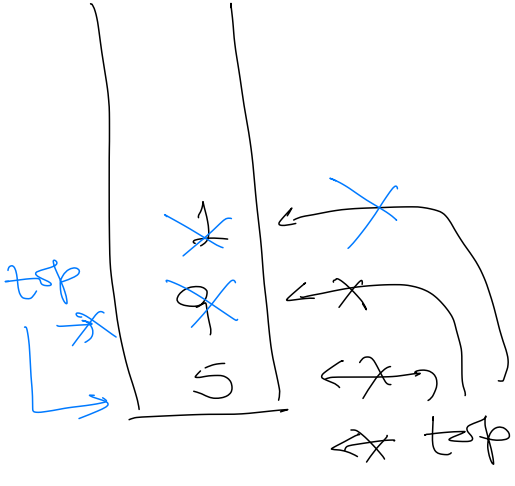
Push(9)

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.

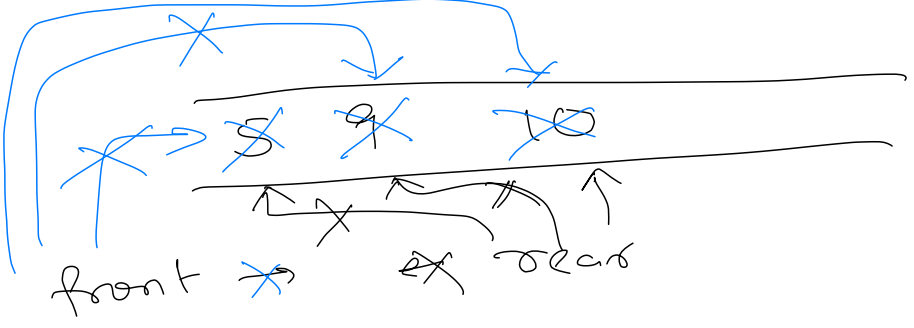
```
interface StackInf {
    void Push (int);
    int Pop ();
    boolean IsFull ();
    boolean IsEmpty ();
}
```

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.

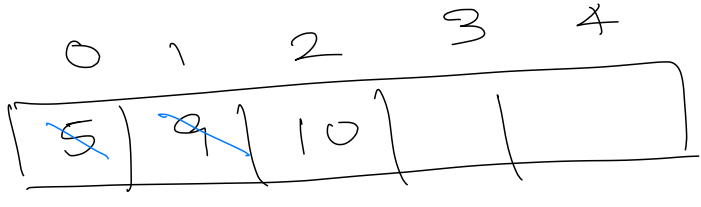
Delete Q() → 10

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.



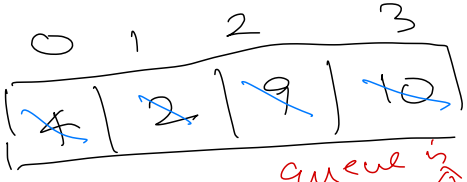
front → ~~-1~~ 0 1

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

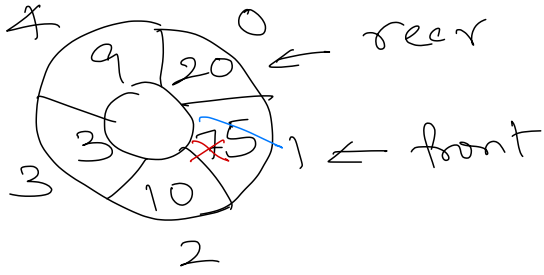
front → ~~-1~~ 0 ~~1~~ ~~2~~ 3
rear → ~~-1~~ 0 ~~1~~ 2 3

↑ 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

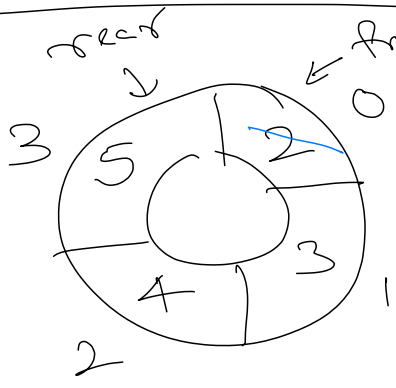
Add Q(7) rear → ~~0~~ ~~1~~

(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

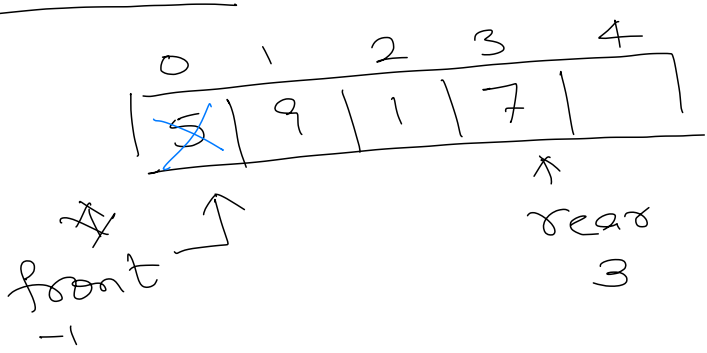
Add Q (2)
Add Q (3)
Add Q (4)
Add Q (5)

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

Delete Q() → 2

Add Q (10) ✗ Queue is full.

Linear Queue



Delete Q ( )

→ Move front to next element.

→ Remove element from front.

→ [Shift all elements of queue to left by 1 place.]

↓

This is done to avoid condition that queue is empty as well as full.

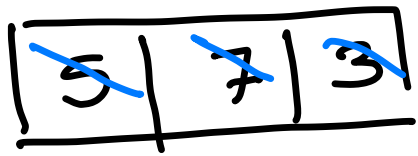
Inefficient way.

OR

→ if queue is empty AND queue is full then

→ Reset front and rear

front = -1, rear = -1.



Add Q(5)
Add Q(7)
Add Q(3)

front → -1 0 1 2
rear → -1 0 1 2

⇐ Queue is full.

Queue is empty.

Delete Q() ⇒ 5
Delete Q() ⇒ 7
Delete Q() ⇒ 3

---

class C1 {

data member
member functions F1();

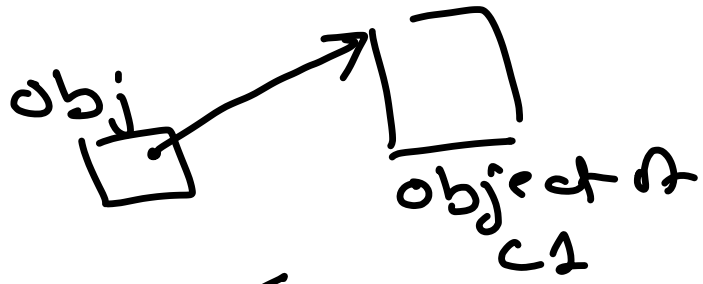
}

C1 obj;

↳ variable that stores reference to object of class.

```
obj.F1();
```
← Null Pointer Exception.

```
obj = new C1();
```




```
obj.F1();
```
✓

```
class Stack {
    public void Push(int)
    public int Pop() {...}
        :
}
```

Defining Stack ⇔ ADT

```
interface StackInt {
    public void Push(int);
    public int Pop();
        :
}
```

```
class StackUsingArray
      implements StackIntf
{
   :
}
```

```
Reverse(int [] elements,
        StackIntf stack)
   stack.Push(..);
     :
   stack.Pop();
}
```


```
Obj = new StackUsingArray();
Reverse(arr, Obj);
```

---

main()

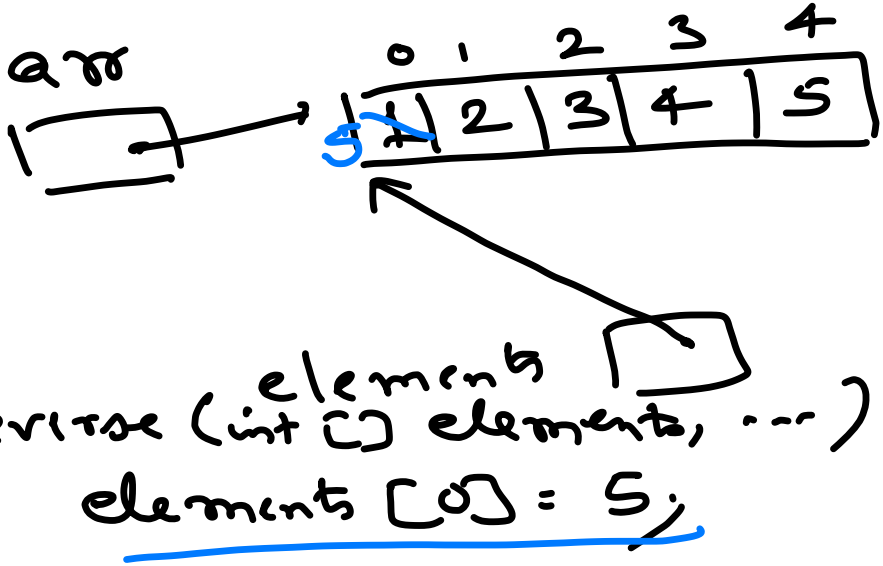
```
int [] arr = {1, 2, .. , 5};
```

Reverse (arr, ...);

⋮

}

OR

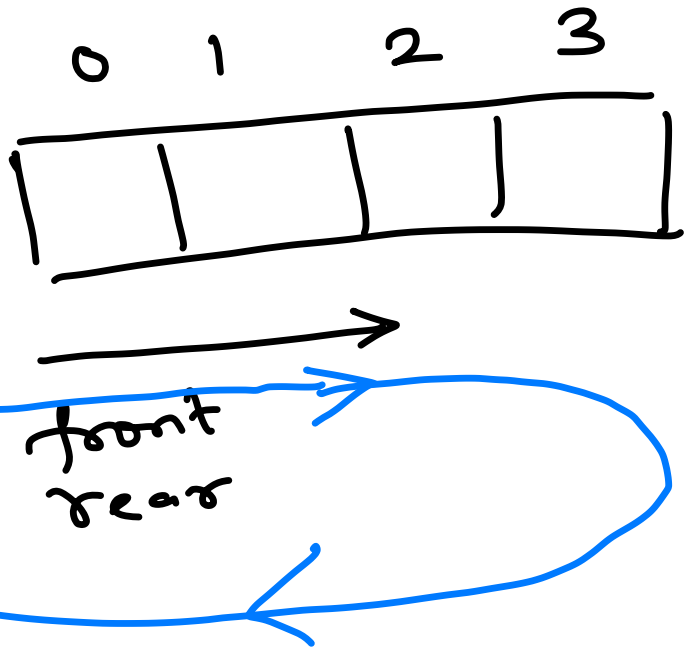


Reverse (int [] elements, ...) {

elements [0] = 5;

}

```
int [] arr;
arr = new int [5];
arr [0] = 1;   arr [2] = 3;
arr [1] = 2;   arr [3] = 4;
               arr [4] = 5;
```

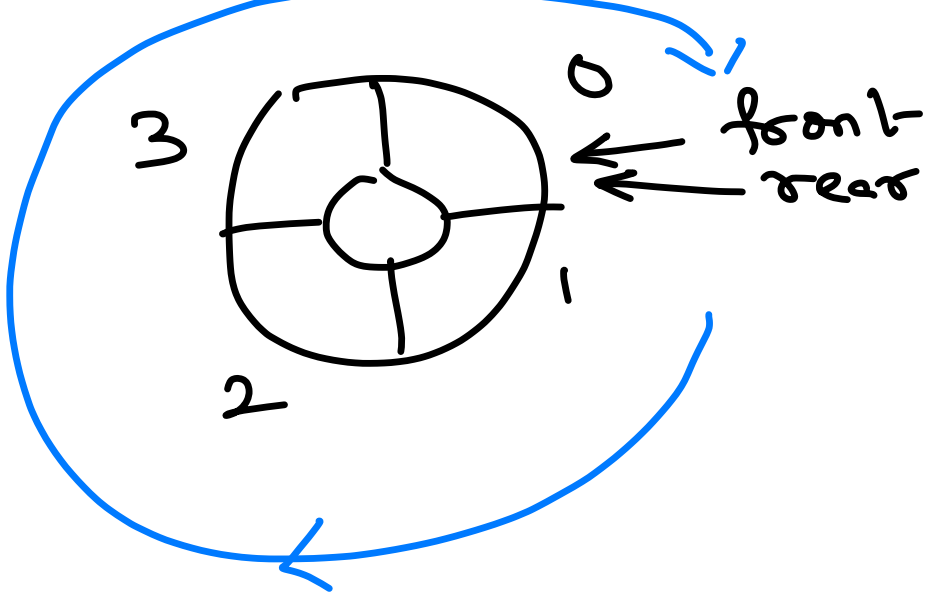



```
++rear;
if (rear == n)
    rear = 0;
```

OR

```
rear = (rear + 1) % n;
```

0..(n-1)



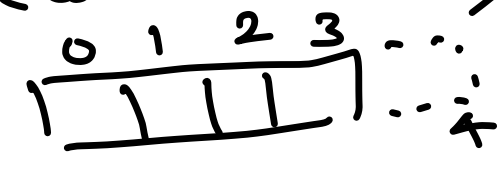
front equals rear ← Empty

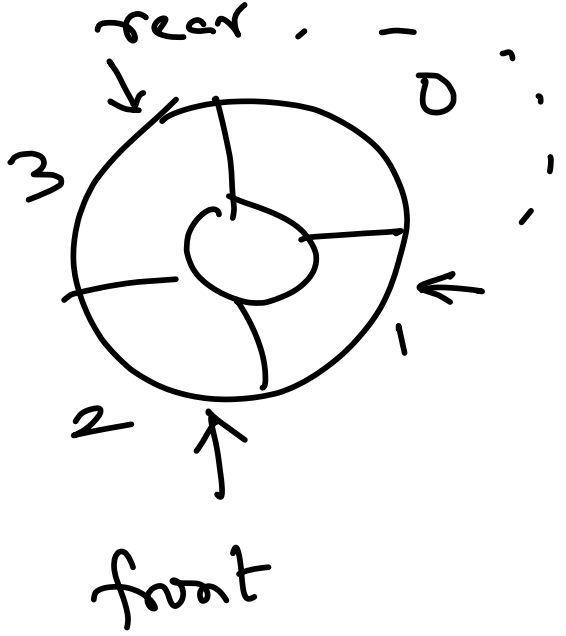
rear just before front ← Full

⇓

(rear+1) % n equals front

new int [4];





---

① Check for balanced parenthesis.

( ) ( ) ✓

([ )] ✗

) (✗

([ ]) ✓

Hint: Use stack.

boolean IsBalanced (String str)

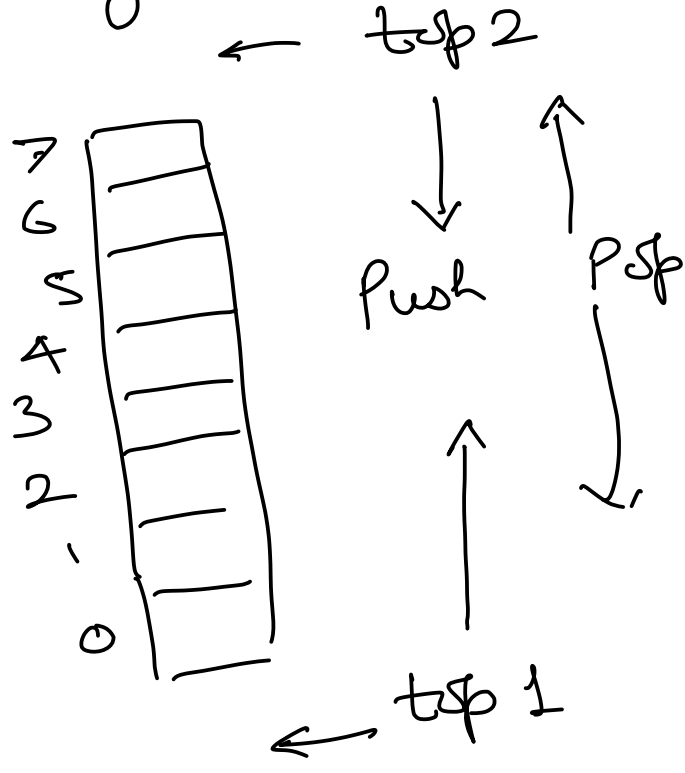
* ② Implement Stack using Queue.

* ③ Implement Queue using Stack.

Hint: ② will need two queues.

③ will need two stacks.

④ Implement 2 Stacks in a single array.



Linked List

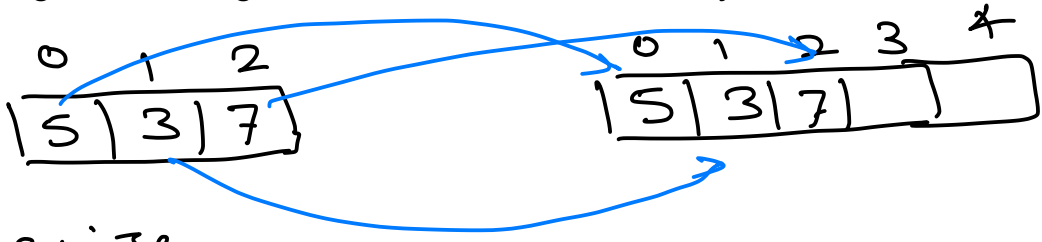
Array : Need?

When we need to store multiple elements.
And do same processing on those elements.

Properties of array:
- Data structure that stores multiple elements, all of the same type.
- All elements of an array are stored sequentially in memory, one after another.

Advantages of array:
- Efficient lookup OR Random access.
- Efficient in adding or removing elements at the end of array.

Disadvantages of array"
- Fixed size. Resizing of array is inefficient.
- Inserting and deleting of elements, in middle of array is inefficient.



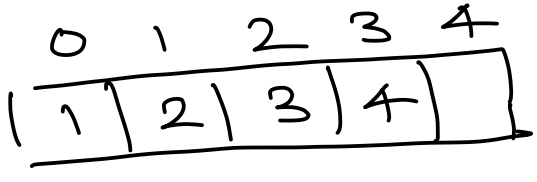
Resize
→ Create a larger array
→ Copy values from existing array to new one.
→ Release old memory.

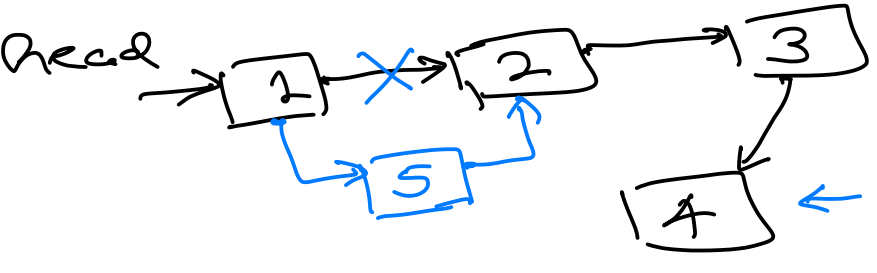
Initial array size = 1000 → Copy 1000 elem.

Resized array size = 2000 → Copy 2000 elements

Resized array size = 4000

Linked list → Do not store elements next to each other.





Properties of linked list
- Stores data as a chain of nodes. → where data is stored.
- Each node contains data and a pointer to next node in chain.
- We need to know where first node is of list - head.

Advantages of linked list
- Can easily grow / shrink in size.
- Efficient in insertion and deletion of elements.

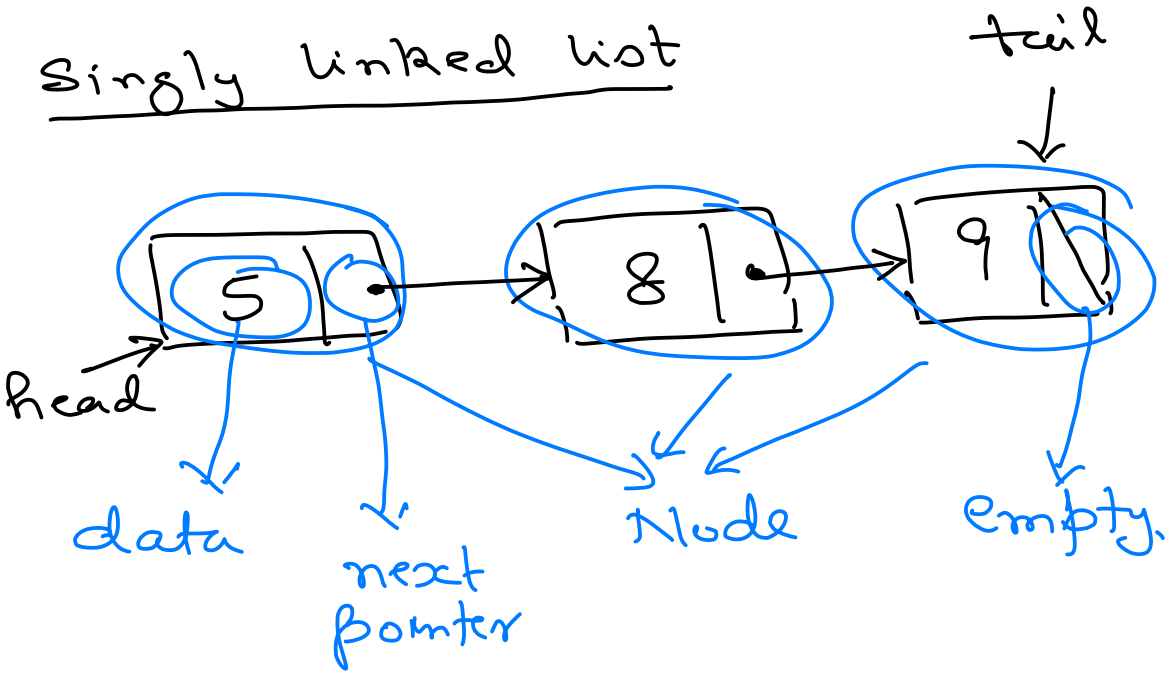
Disadvantages of linked list
- Random access is inefficient.

Types of linked list
- Singly linked list (Uni-directional)
- Doubly linked list (Bi-directional)
- Circular list.

One node keeps track of one neighbour only.

Each node keeps track of both of its neighbours

Singly linked list



Traversal

Starting from first element, access each element one at a time, till the last element.

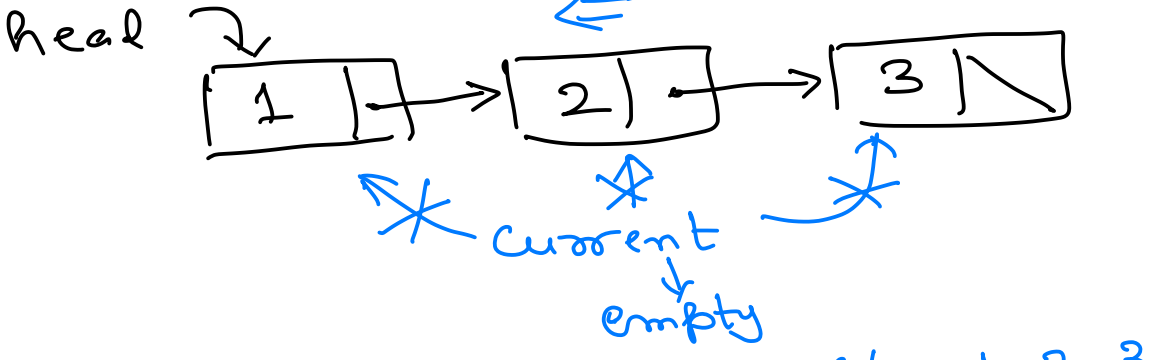
```
for ( i = 0; i < elements.length; ++i)
    stack.Push (elements[i]);
```

Traversal.

① What if list is empty?
② What if list is not empty?

head → empty ⇐ list is empty.

⇐ list is not empty.



o/p: 1 2 3

Traversal of singly list
- if list is empty them Stop.
- Set current to first node of list.
- while (current is not empty) do
  - Process current node.
  - Move current to current node's next.
- Stop.

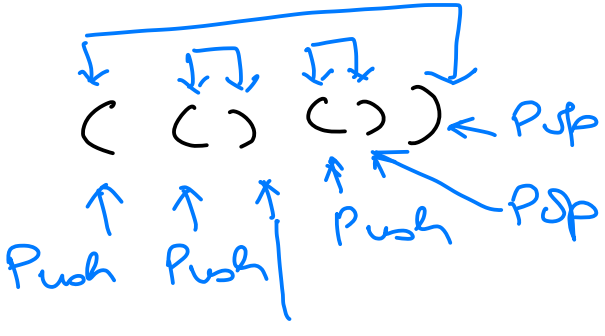
Traversal of singly list (Optimised)
- Set current to first node of list.
- while (current is not empty) do
  - Process current node.
  - Move current to current node's next.
- Stop.

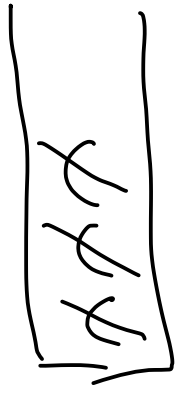
① Check for balanced parenthesis.

( ) ✓)(× (×) ×

((( ))) ✓ (( ) ( )) ✓

(( ) ×





Pop
→ Check if opening parent. poped out is of same type as closing.

→ When I/p is over and stack is empty then Report Success.

{) ×

IsBalanced(expr)

- While expr is not over, get a char from it.
  - if char is '(' then Push char on stack.
    Else // char is ')'
    - if stack is empty then
      - Report failure
      - Stop
    - Pop a value from stack
    - if char is ')' and value is NOT '(' then
      - Report failure
      - Stop
- if stack is not empty then
  - Report failure
  - Stop
- Report success
- Stop

Unit Test → TDD
Test Driven Development

JUnit

(2) Implement queue using Stacks.

↳ Push()
↳ Pop()

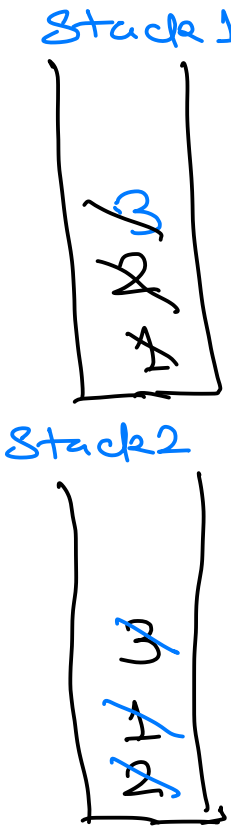
Add Q (1)
Delete Q () ⇒ 1

Add Q (1)
Add Q (2)
Delete Q () ⇒ 1
Delete Q () ⇒ 2

Add Q (1)
Add Q (2)
Delete Q () ⇒ 1
Add Q (3)
Delete Q () ⇒ 2
Delete Q () ⇒ 3

Add Q()
↳ Push()

Delete Q()
↳ Pop()



Add Q(1)
↳ Push(1) in Stack1

Add Q(2)
↳ Push(2) in Stack1

Delete Q() ⇒ 1
→ if Stack2 is empty
→ One by one Pop from Stack1 and Push in stack2.
↳ Pop() from Stack2

Delete Q() ⇒ 2
↳ Pop() from Stack2

Add Q(3)
↳ Push(3) in Stack1

Delete Q() ⇒ 3
↳ if stack2 is empty
}
↳ Pop() from stack2

Add Q (1) → Push (1) in Stack 1

Add Q (2) → Push (2) in Stack 1

Delete Q ()

Add Q (3)

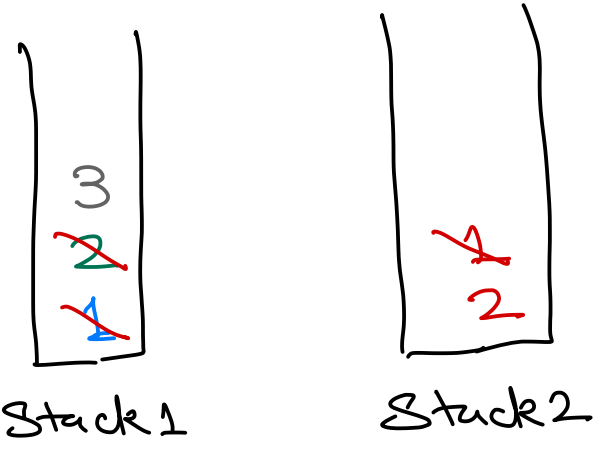
↓

Push (3) in Stack 1

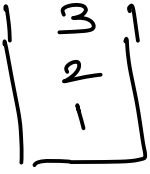
1

→ if stack 2 is empty then
→ One by one pop from stack 1 and push in stack 2

→ Pop from stack 2



Store ⇒ 1 2 3



Get from Stack ⇒ 3 2 1

and Put in another stack

Get from another stack ⇒ 1 2 3

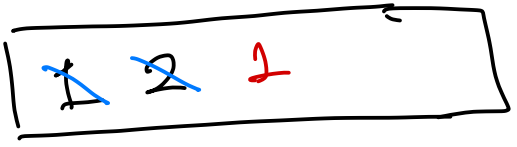
(3) Stack using queues.

Push (1) → Add Q(1) in Queue 1

Push (2) → Add Q(2) in Queue 1

Pop ( ) ⇒ 2

→ Move all elements from Queue 1 to Queue 2, except the last one.



Queue 1

elem → 1 2

Queue 2

→ Last element removed from queue 1 is result

→ Put back all elements from queue 2 to queue 1

→ Return result

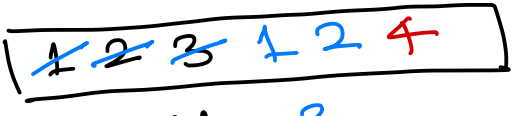
Push (1) → Add Q(1)

Push (2) → Add Q(2)

Push (3) → Add Q(3)

Pop() → 3

Push (4) → Add Q(4)

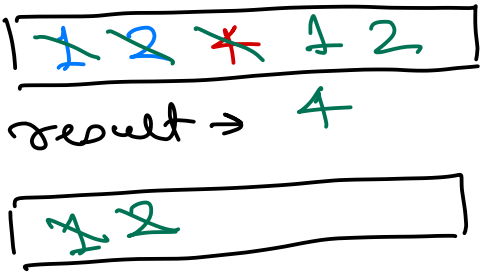


Queue 1

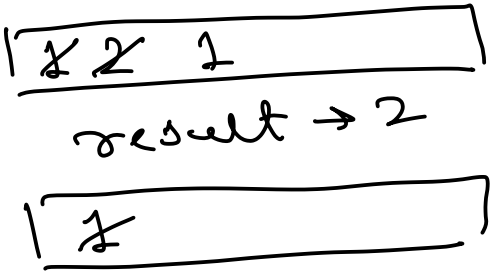
result → 3

Queue 2

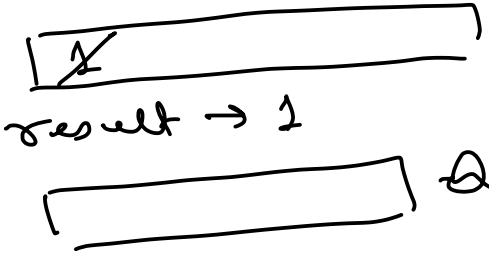
pop() → 4



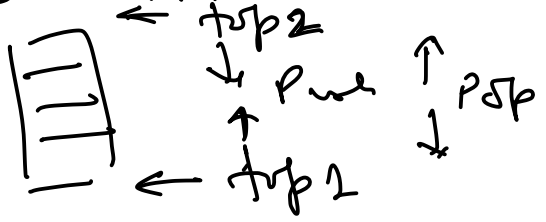
pop() → 2



pop() → 1



④ Implement two stacks



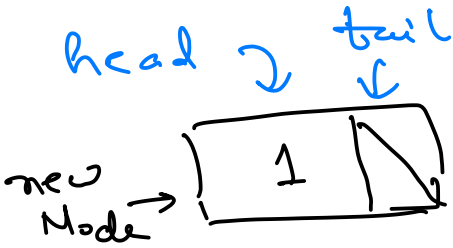
Push(stackno, elem)

Linked List

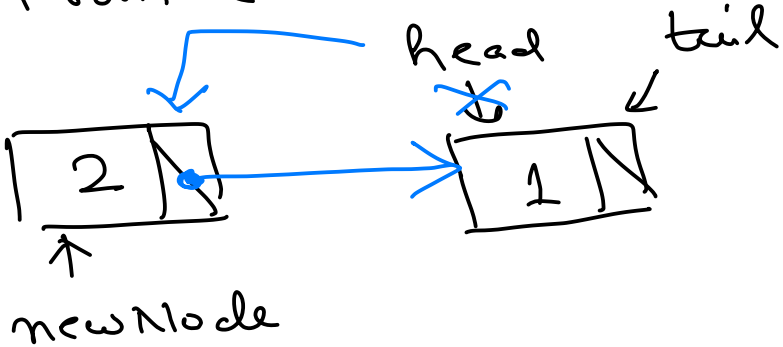
Create a linked list

- Add new element at front
- Add new element at end of list
- Insert in a list ⇒ Creating Sorted list.

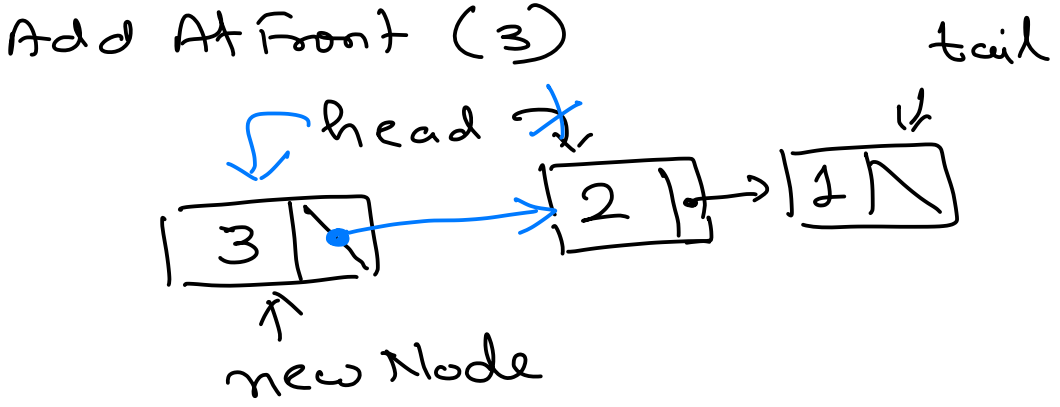
Add At Front (1)

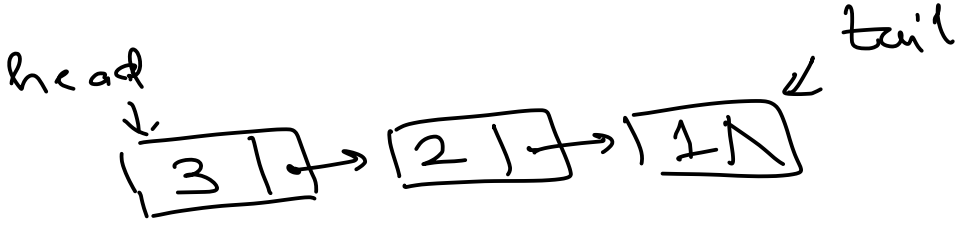


Add At Front (2)



AddAtFront(element)
- Make space for new elements, say newNode.
- Store element in newNode's data.
- Set newNode's next to empty.
- if list is empty then
  - Set head and tail to newNode.
  - Stop.
- Set newNode's next to head.
- Set head to newNode.
- Stop.





Dynamically allocate memory for Node.

```
Class Node {
public int data;
public Node next;
}

Node head;
Node tail;
```