Analysis of Algorithm

Time Complexity | Space Complexity

Big O notation

Constant time complexity algo
⇒ algo takes time independent of I/p size.

Find sum of 3 numbers

1. Read 3 numbers (no1, no2, no3). — 1
2. Sum = no1 + no2 + no3 — 1
3. Print Sum. — 1
4. Stop.

Total = 3

If its a constant then time complexity = $\boxed{O(1)}$

Find sum of N numbers

$y-x+1$ ← [x y] → closed range

$y-x$ ← [x y) → open at y

1. Read N. —— 1
2. Create space for N numbers (nums) — 1
3. for i = 0 to (n-1) ⇒ loop. how many times? n
   - 3.1 Read nums[i] —— 1] n times
4. Sum = 0 —— 1
5. for i = 0 to (n-1) ⇒ loop. how many times? n
   - 5.1 Sum = Sum + nums[i] —— 1] n times
6. Print Sum. —— 1
7. Stop.

Total: $1+1+n+1+n+1 = \cancel{2}n+\cancel{4}$

1. Ignore constants

⇒ $O(n)$ ← linear time complexity.

Find time complexity of following

```
for i = 0 to (n-2)
    for j = (i+1) to (n-1)
        .. do something...
```

Sum of first N natural numbers $= \frac{n(n+1)}{2}$

i	j	how many times inner loop (j) runs?
0	$1 \quad 2 \quad \dots \quad (n-1)$	$(n-1)$
1	$2 \quad 3 \quad \dots \quad (n-1)$	$(n-2)$
2	$3 \quad 4 \quad \dots \quad (n-1)$	$(n-3)$
$\vdots$	$\vdots$	$\vdots$
$(n-3)$	$(n-2) \quad (n-1)$	2
$(n-2)$	$(n-1)$	1

Total $= \frac{(n-1)(n-1+1)}{2}$

$$\frac{(n-1)(n)}{2} = \cancel{\frac{1}{2}}(n^2-n)$$

As value of n gets larger $n^2 - n \approx n^2$

(1) Ignore constants $\Rightarrow n^2 - n$

(2) Pick term with highest power of n.

$\Rightarrow O(n^2)$ ← quadratic time complexity.

Find time complexity of

for $i = 0$ to $(n/2)$

for $j = (i+1)$ to $(n-1)$

for $k = 0$ to $(n-1)$

... do something ..

Stack

- Stack is a linear data structure.
- Stack is a container of objects.

Stack operations

- LIFO – Last In First Out
- Elements are added and removed according to LIFO principle.
- Operations are performed with respect to “**top**” of stack.

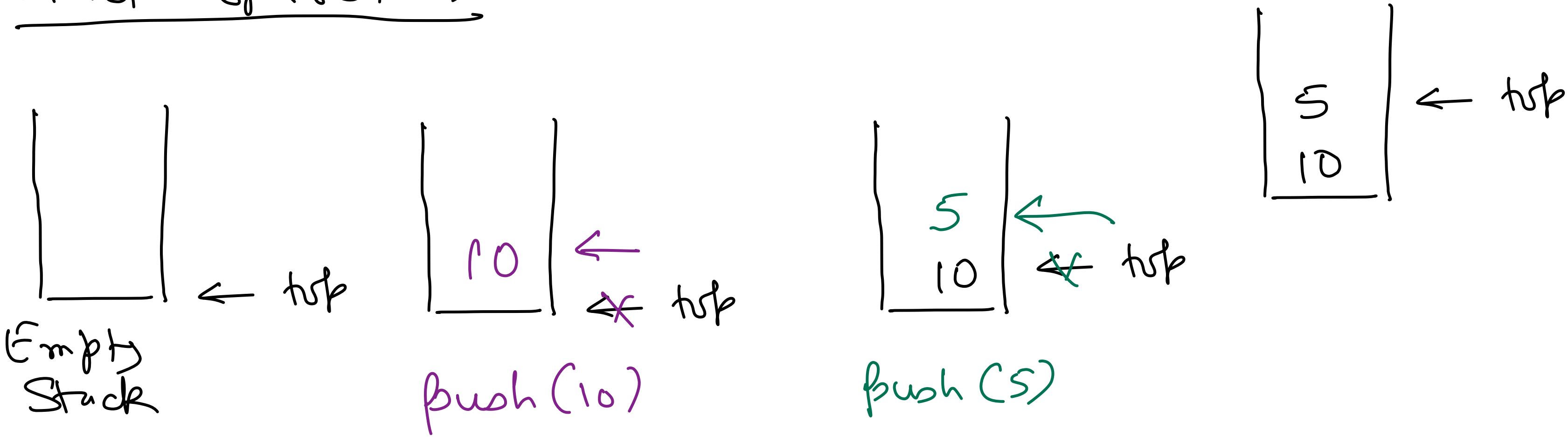
Stack as Abstract Data Type (ADT)

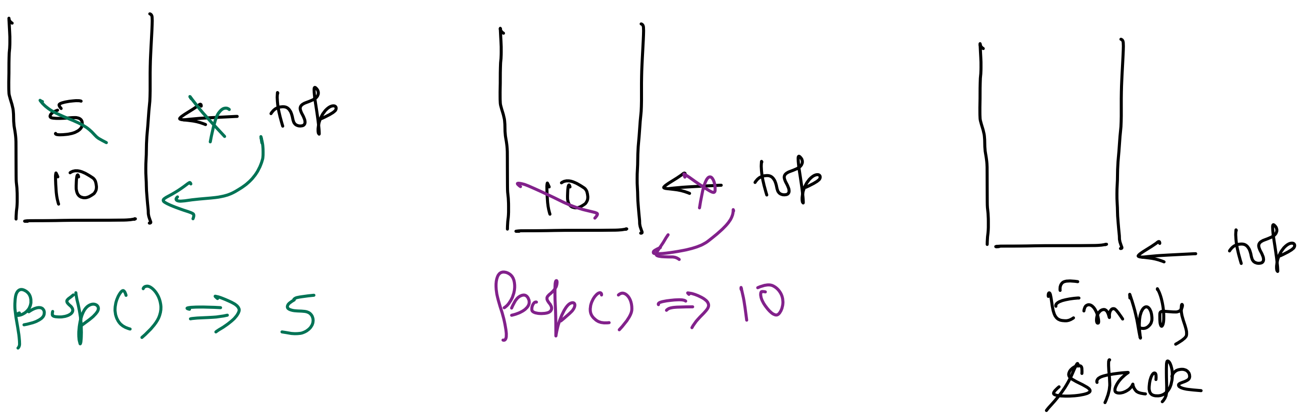
- push ← adds element to stack
- pop ← removes element from stack.
- peek ← get top element without removing it
- isEmpty
- isFull

defines what operations can be performed.

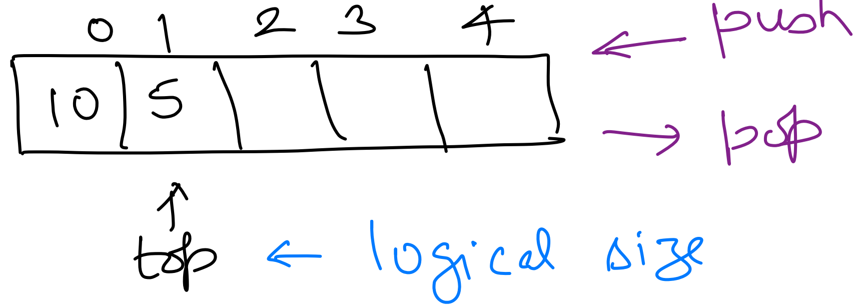
```java
public interface Stack {
    void push(int element);
    int pop();
    int peek();
    boolean isEmpty();
    boolean isFull();
}
```

Stack Operations



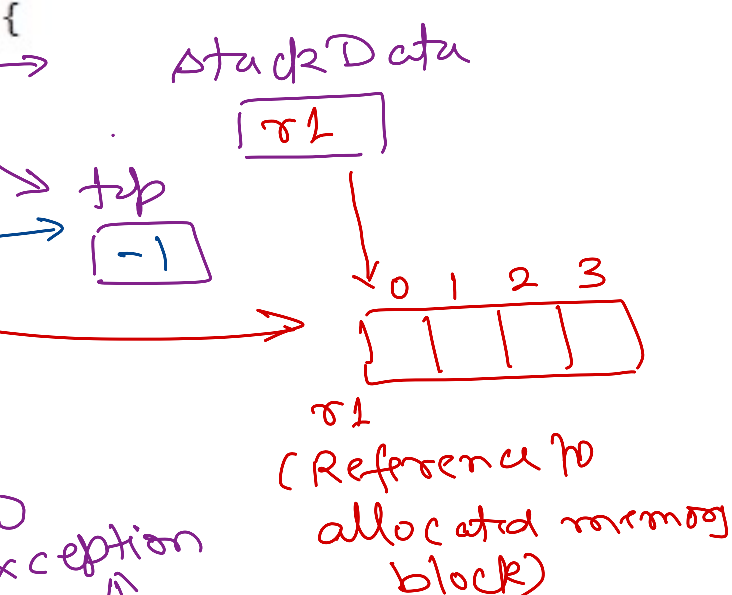


Stack using Array




```java
public class FixedSizeStack implements Stack {
    private int[] stackData;
    private int top;

    public FixedSizeStack(int n) {
        stackData = new int[n];
        top = -1;
    }
```



Throw Exception

Push(element)

if (isFull()) $\boxed{\text{return;}}$

- If stack is full then stop. → ~~if (top == stackData.length-1) return;~~
- Make space at top for new element. → ++top;
- Store new element and make it topmost element. → stackData[top] = element;

Pop()

- If stack is empty then stop. → if (isEmpty()) $\boxed{\text{return -1;}}$ → throw appropriate Exception
- Set topmost element as result. → result = stackData[top];
- Remove topmost element and make element below top, the topmost element. → --top;
- Return the result. → return result;

IsEmpty()

- If no element stored at top then return true. → if (top == -1) return true;

Else return false → return false;

IsFull()

- If no space left for new element to be stored then return true.

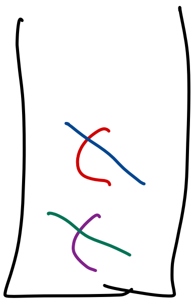
Else return false.

→ if (top == stackData.length -1) return true;

→ return false;

Check if string of paranthesis is balanced or not.

(( ))

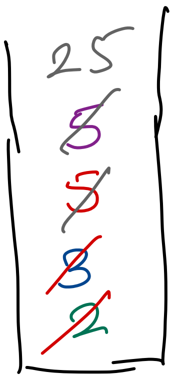


(( ) ✗

( )) ✗

[{ (})] ✗

Evaluate postfix expression



2 3 + 5 *

$\underbrace{5 * 5}_{25}$

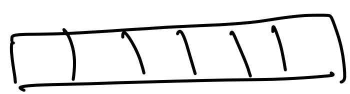
$\underbrace{2 + 3}_{5}$

Min Stack

Max Stack

Implement n - stacks in an array.

Implement two stacks in an array.



top1 → ← top2

→ push ←

← pop →

Application of stack

→ O.S. ⇒ function calls.

→ Recursive to Iterative algorithm.

→ Other algorithms.

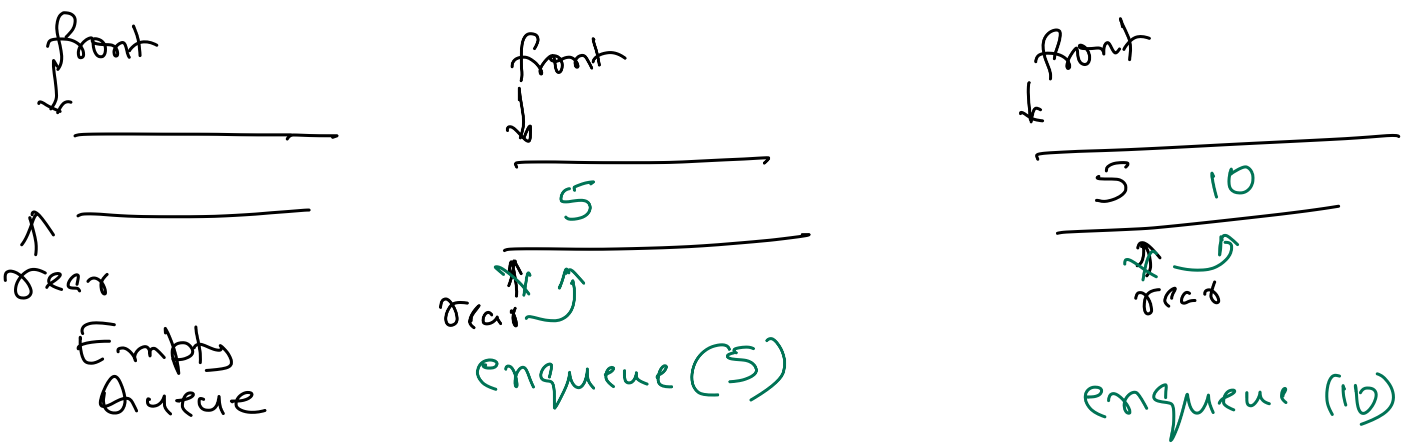
Queue

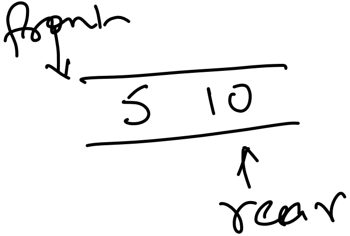
- Queue is a linear data structure.

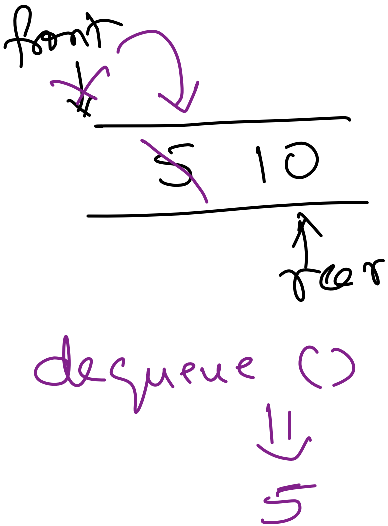
- Queue is a container of objects.

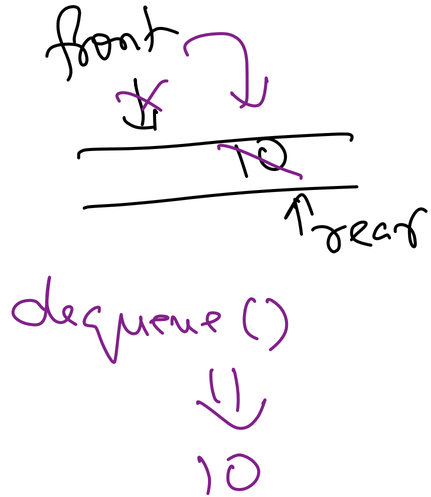
Queue operations

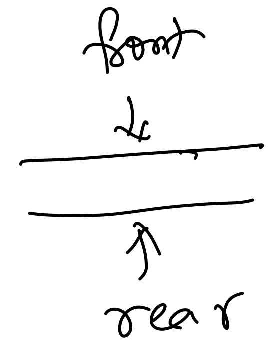
- FIFO – First In First Out
- Elements are added and removed according to FIFO principle.
- Addition of elements are performed at “**rear**” of queue.
- Elements are removed from “**front**” of queue.




front
5 10
rear


front
~~5~~ 10
rear
dequeue()
5


front
~~10~~
rear
dequeue()
10

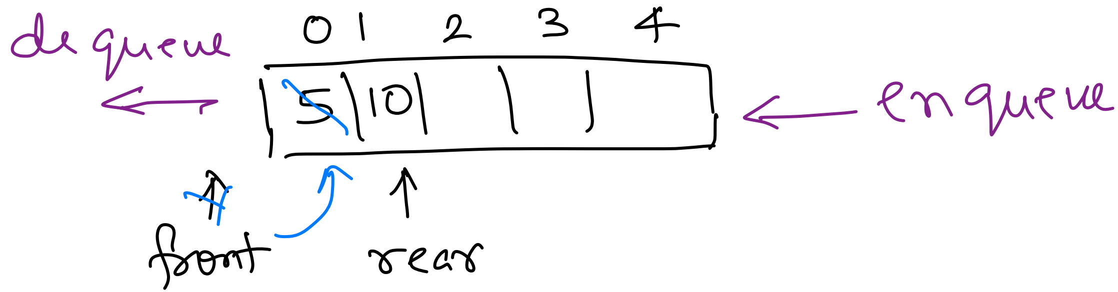

front
rear

Define Queue as ADT.

```
public interface Queue {
    void enqueue (int element);
    int dequeue ();
    boolean isEmpty ();
    boolean isFull ();
}
```

Queue using Array

front -> -1
rear -> -1




```
public class ... implements Queue {
    private int [] queueData;
    private int front;
    private int rear;
    public ... (int n) {
        queueData = new int [n];
        front = -1; rear = -1;
    }
```

Enqueue(element)
- If queue is full then stop. → if (isFull()) return; ⇒ throw Exception
- Make space at rear for new element. → ++rear;
- Store new element and make it the rear element. → queueData[rear] = element;

Dequeue()
- If queue is empty then stop. → if (isEmpty()) return -1; ⇒ throw Exception
- Move the front towards rear. → ++front;
- Remove and return the front element as result.
↳ return queueData[front];

IsEmpty()
- If no elements stored in queue then return true. → if (front == rear) return true;
Else return false. → return false;

IsFull()
- If no space left for new element to be stored then return true.
Else return false.
↳ if (rear == queueData.length-1) return true;
↳ return false;

Linear queue suffers from the problem that queue can be empty and full at the same time.

0	1	2
~~5~~	~~10~~	~~20~~

front → ~~-1~~ ~~0~~ ~~1~~ 2

rear → ~~-1~~ ~~0~~ ~~1~~ 2

enqueue (5)

enqueue (10)

enqueue (20)

isFull() ⇒ TRUE

dequeue () → 5

dequeue () → 10

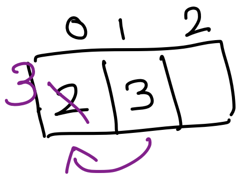
dequeue () → 20

isEmpty () ⇒ TRUE

isFull() ⇒ TRUE

Solution

① In dequeue, after removing the element, shift all remaining elements of queue to left by one place.



dequeue() → 2

front → −1

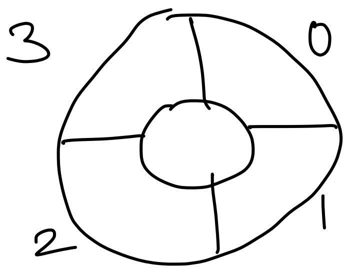
rear → ~~1~~ 0

② In dequeue, we check if queue is empty and full at the same time. If yes, we reset front & rear.

③ Circular queue.

Circular Queue

- Last position of Circular Queue is connected back to first position. Making a circle.



$N = 4$

front → 0

rear → ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~

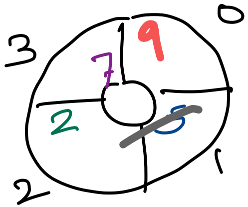
⇓

0

Incrementing front and rear is always a MOD N operation.

$$rear = (rear + 1) \% N$$

```
if (rear == N)
    rear = 0;
```



$N = 4$

front = ~~0~~ 1

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

enqueue(5)

enqueue(2)

enqueue(7)

~~enqueue(3)~~ throw queue full exception

dequeue() => 5

enqueue(9)

isFull()

```
if ((rear+1)%N == front)
    return true;
```

Enqueue(element)
- If queue is full then stop.
- Make space at rear for new element. → rear = (rear +1) % queueData.length;
- Store new element and make it the rear element.

Dequeue()
- If queue is empty then stop.
- Move the front towards rear. → front = (front +1) % queueData.length;
- Remove the front element as result.
- Return result.

IsEmpty()
- If no elements stored in queue then return true.
Else return false.

IsFull()
- If no space left for new element to be stored then return true.
Else return false. ↳ if ((rear +1) % queueData.length == front)
return true;

Applications of queue

1. O.S. ⇒ Schedular.
2. Simulation.
3. Other algorithms.

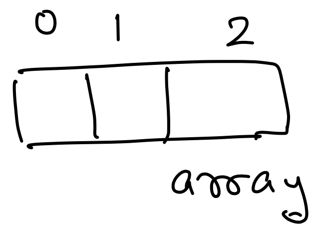
---

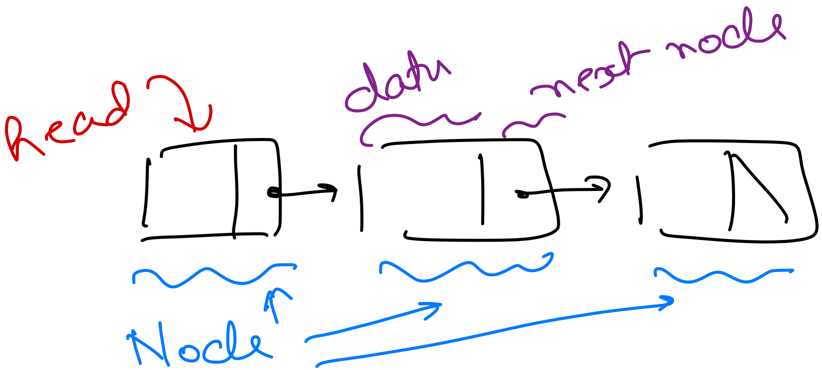
1. Implement Queue using Stack.
2. Implement Stack using Queue.

Linear Data Structures

Linked List

- Need for a linked list?





Properties of Linked List

- Stores data as a chain of nodes.
- Each node contains data and a pointer to the next node in the chain.
- First node of linked list is pointed by “head”.
  When list is empty, head do not point to any node.
- Last node of list points to no node.

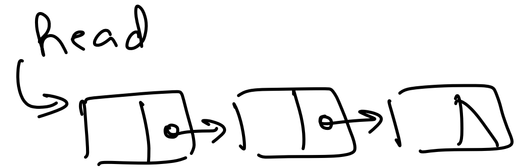
Pros and Cons of Linked List

- Advantages
  - o Can dynamically grow or shrink is size.
  - o Efficient in insertion and deletion of elements.

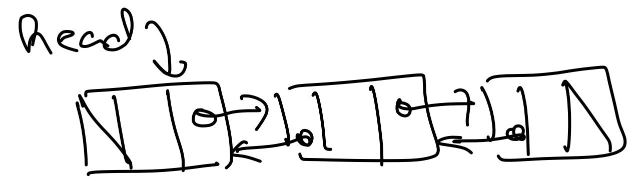
- Disadvantages
  - o Lookup OR Random access is inefficient.

Types of Linked List

- Single linked list (Uni-directional).
  One node keeps track of one neighbour node only.



- Doubly linked list (Bi-directional).
  Each node keeps track of two of its neighbours.



- Circular linked list.

Singly Linked List

Traversal

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

array traversal

for i = 0 to (n-1)

... arr[i] . .

linked list traversal

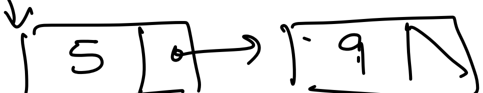
1. Empty list

head → empty

list is empty. Do nothing.

2. Non-empty list

head ↘



⇐ Non-empty list.

1 current

2 current

current $\xrightarrow{3}$ empty

Singly LinkedList Traversal

- If list is empty then stop.
- Set current to first node of list.
- while (current is not empty) do
  - Process current node.
  - Set current to current node’s next.
- Stop.

Head $\rightarrow$ empty

$\uparrow$

current

Singly LinkedList Traversal (Optimised)

- Set current to first node of list.
- while (current is not empty) do
  - Process current node.
  - Set current to current node’s next.
- Stop.