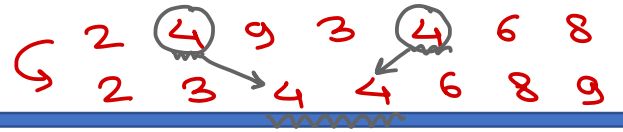
Data Structure & Algorithms

Sunbeam Infotech

Nilesh Ghule

Stack and Queue



- Stack & Queue are utility data structures.
- Can be implemented using array or linked lists.
- Usually time complexity of stack & queue operations is O(1).
- Stack is Last-In-First-Out structure.
- Stack operations
  - push()
  - pop()
  - peek()
  - isEmpty()
  - isFull()*
- Simple queue is First-In-First-Out structure.
- Queue operations
  - push() enque()
  - pop() deque()
  - peek()
  - isEmpty()
  - isFull()*
- Queue types
  - Linear queue
  - Circular queue
  - Deque
  - Priority queue

ADT
1. Array
   - get ele at pos
   - set ele at pos
   - sort, search
   - slice, ...
2. Queue

Stack / Queue using Linked List

- Stack can be implemented using linked list.
  - add first
  - delete first
  - is empty

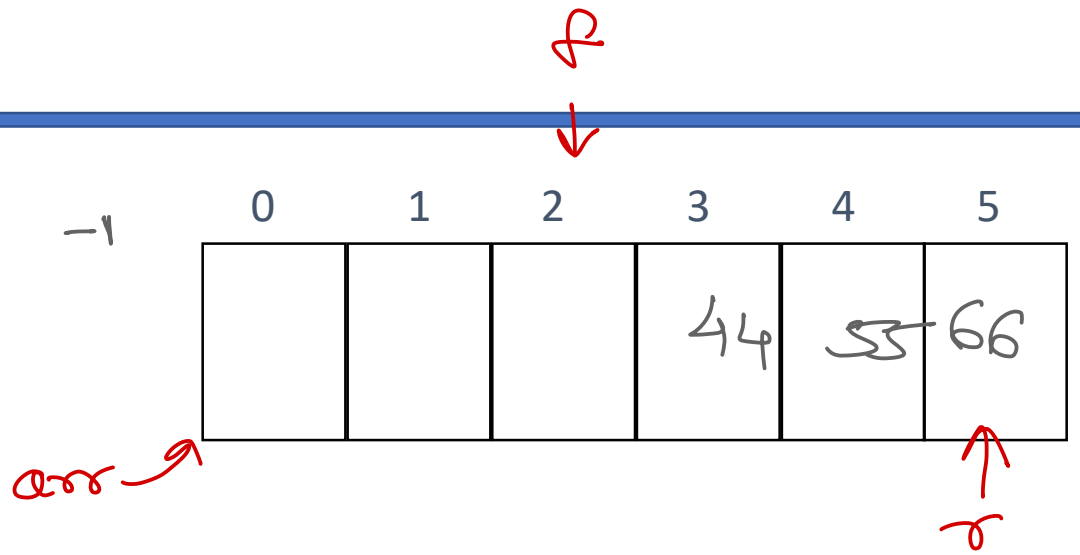
- Queue can be implemented using linked list.
  - add last
  - delete first
  - is empty

Linear Queue → FIFO

deque
pop
front

enque
push
rear

init:
f = -1
r = -1



push:
```
r++;
arr[r] = val;
```

pop:
```
f++;
```

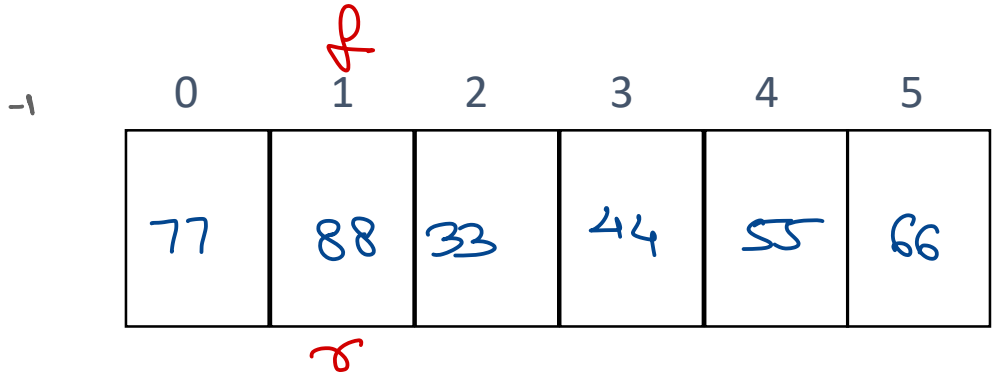
isEmpty:
```
f == r
```

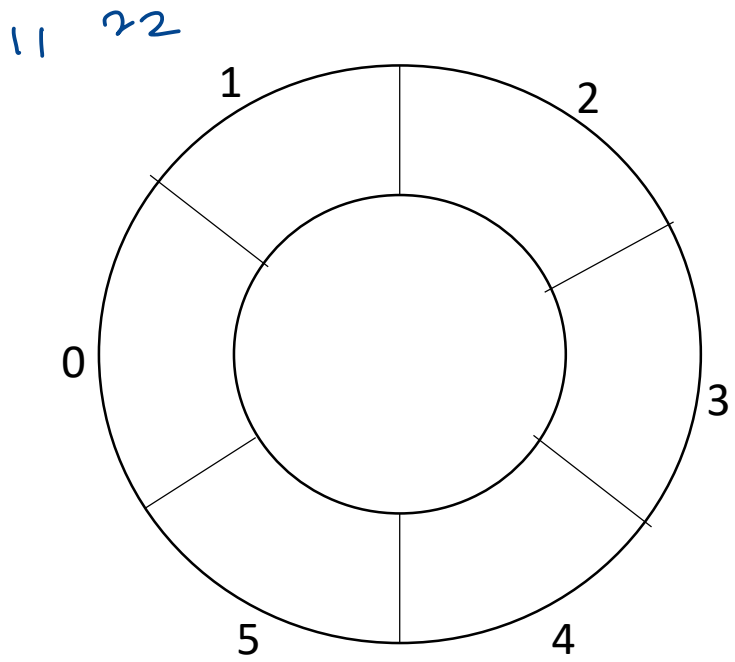
isFull:
```
r == size-1
```

Peek:
```
return arr[f+1];
```

Circular Queue

- In linear queue (using array) when *rear* reaches last index, further elements cannot be added, even If space is available due to deletion of elements from *front*. Thus space utilization is poor.
- Circular queue allows adding elements at the start of array if *rear* reaches last index and space is free at the start of the array.
- Thus *rear* and *front* can be incremented in circular fashion i.e. 0, 1, 2, 3, …, n-1. So they are said to be circular queue.
- However queue full and empty conditions become tricky.





Circular Queue

q full → f == -1 && r == size-1

push

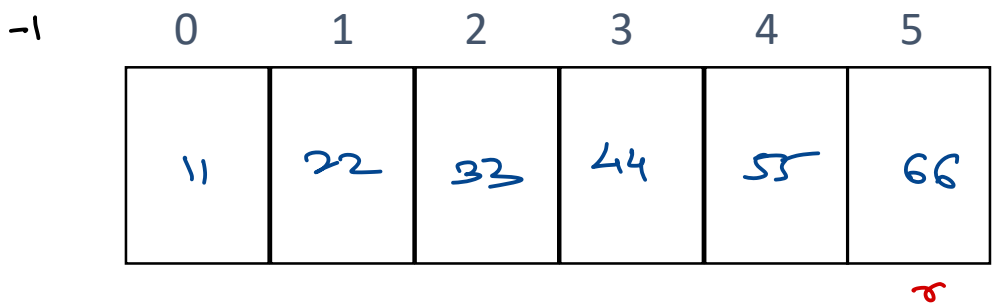
```
r++;
if(r==size)
    r=0;
arr[r] = val;
```

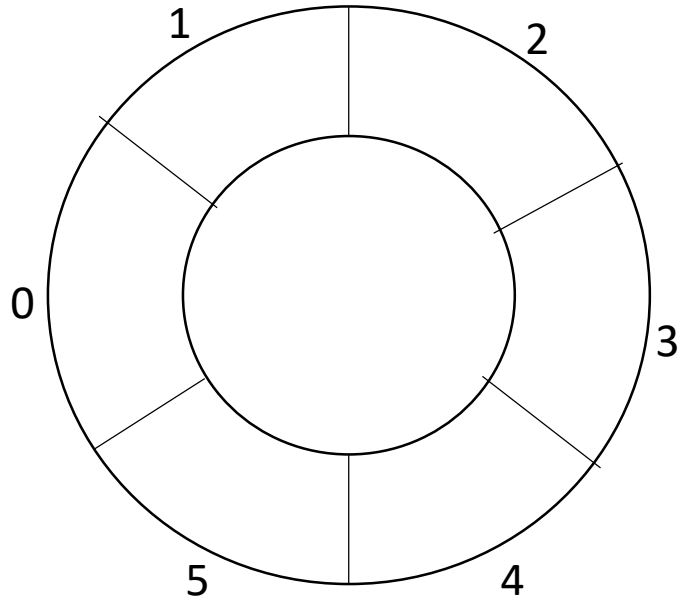
push

```
r = (r+1) % size;
arr[r] = val;
```

r/f

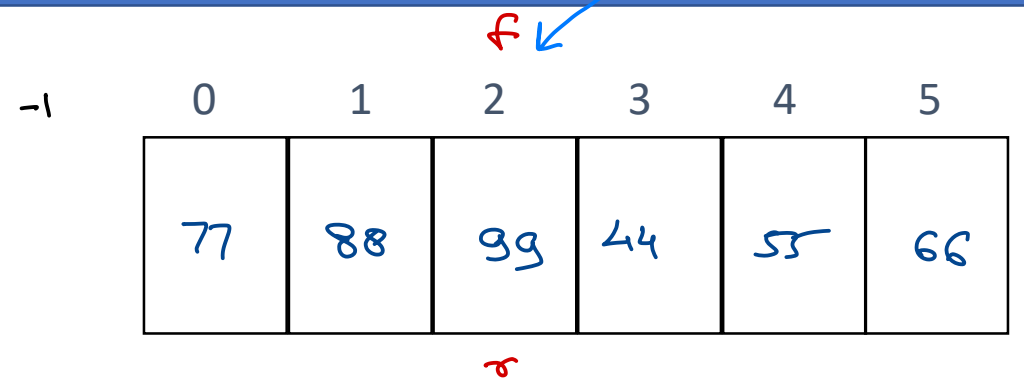
	+1	%c
-1	0	0
0	1	1
1	2	2
2	3	3
3	4	4
4	5	5
5	6	0

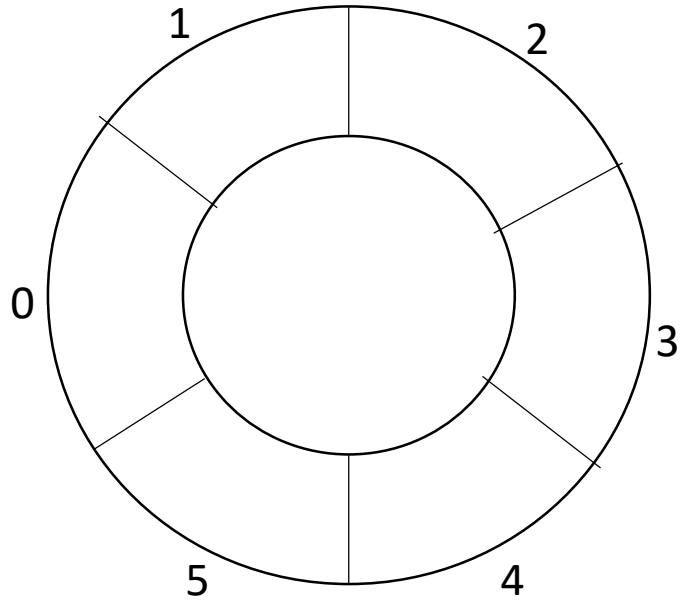




Circular Queue

q full : f == r && f != -1





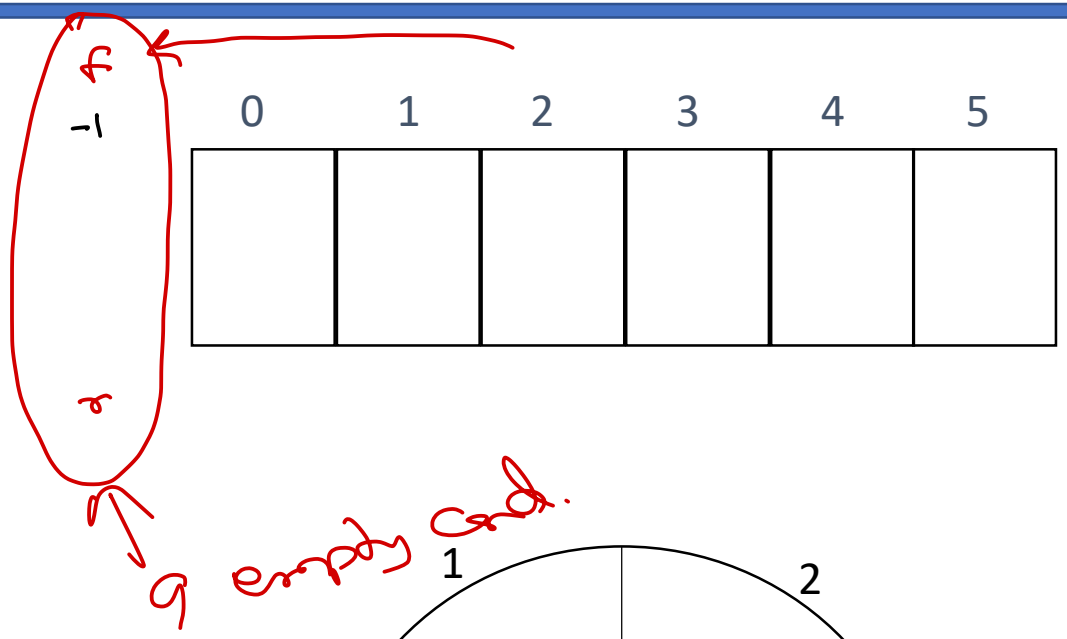
Circular Queue

q empty: f == r

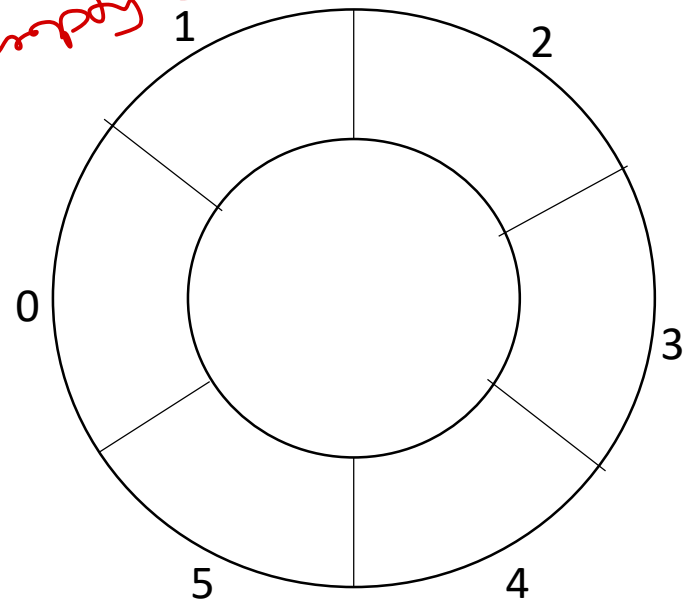
pop:

```
f = (f+1) % size;
if(f == r)   <- check if ele deleted was last ele.
{
    f = -1;
    r = -1;
}
```

tick

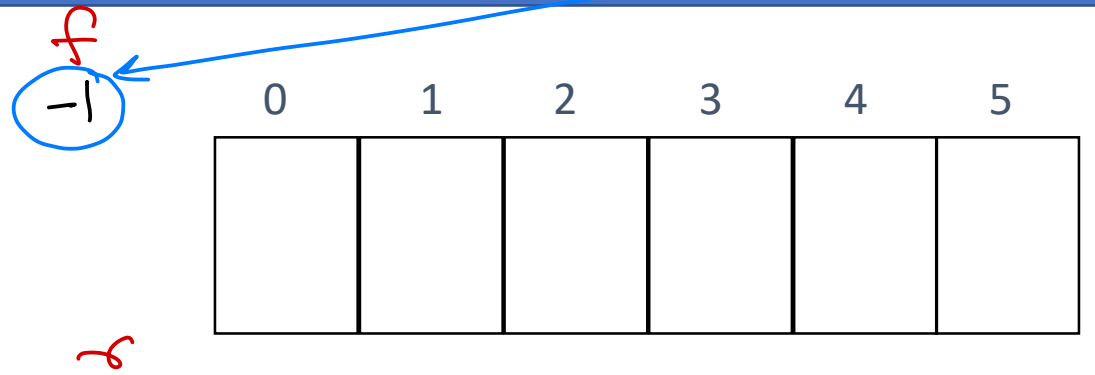


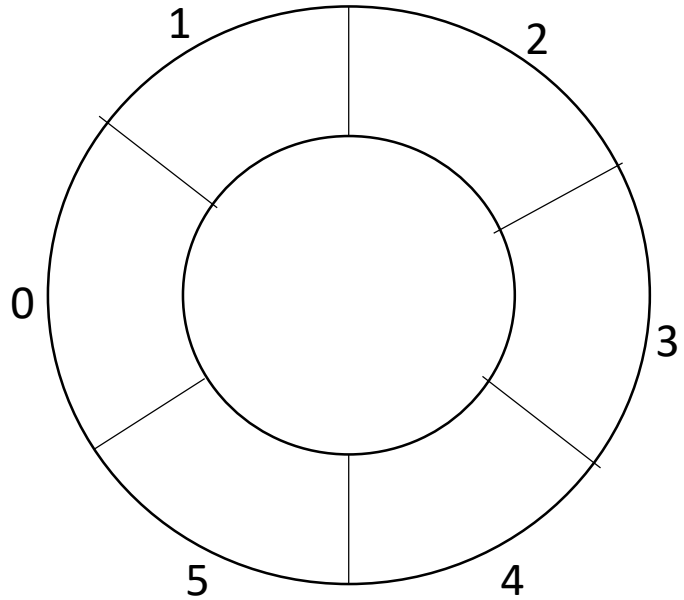
q empty case.



Circular Queue

q empty: f == r && f == -1





Circular Queue

init:

```
r = -1
f = -1
```

push:

```
r = (r+1) % size;
arr[r] = val;
```

pop:

```
f = (f+1) % size;
if(f == r){
    f = -1;
    r = -1;
}
```

peek:

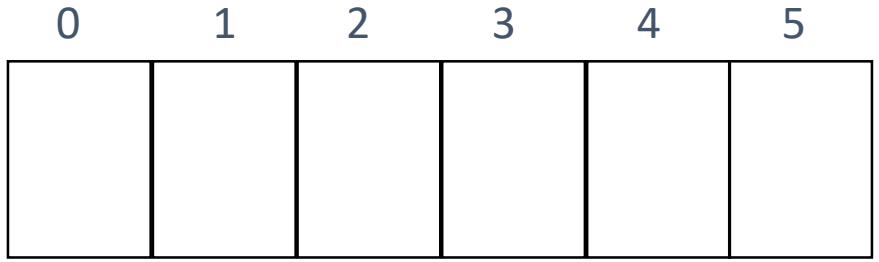
```
i = (f+1) % size;
return arr[i];
```

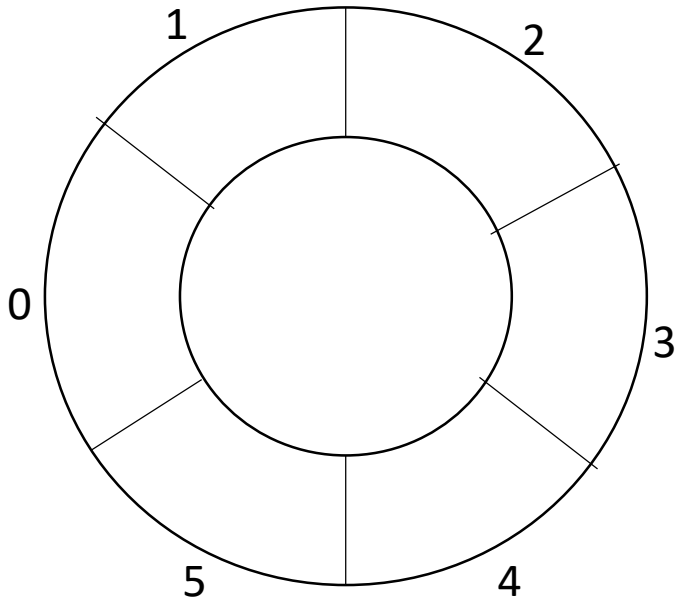
is full:

```
(f == -1 && r == size-1)
OR
(f == r && f != -1)
```

is empty:

```
(f == r && f == -1)
```

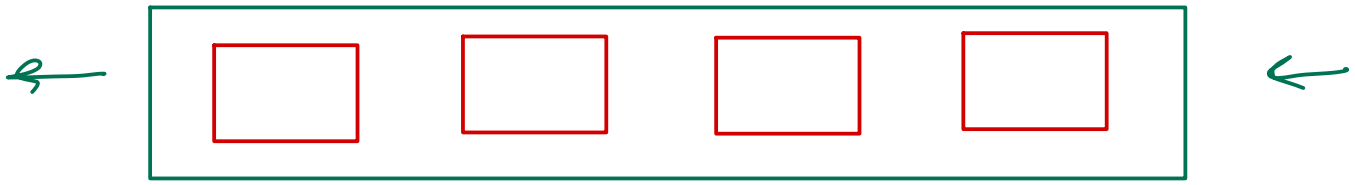




DeQueue → java.util.Deque → 1.6

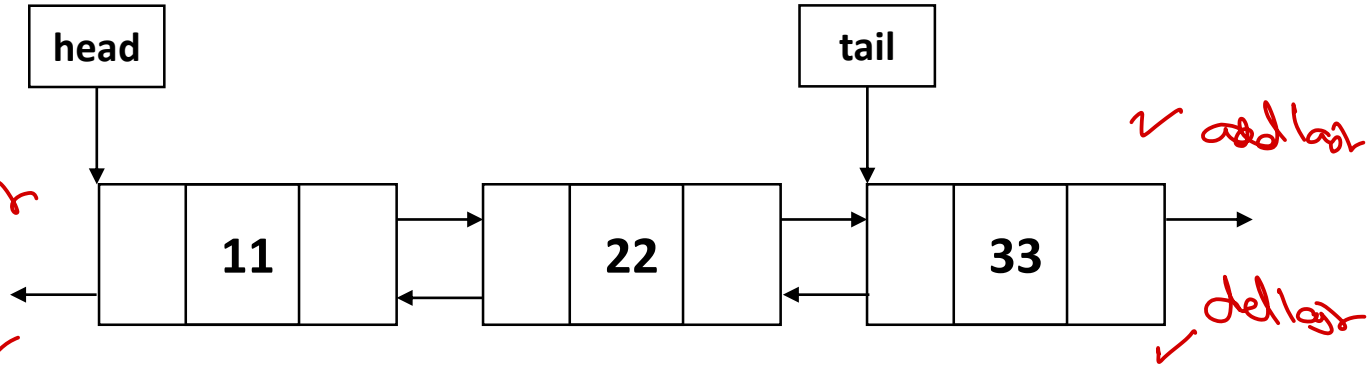
- In double ended queue, values can be added or deleted from front end or rear end.

→ appln specific queue.



input restricted ← rear

front ← output restricted



addfirst

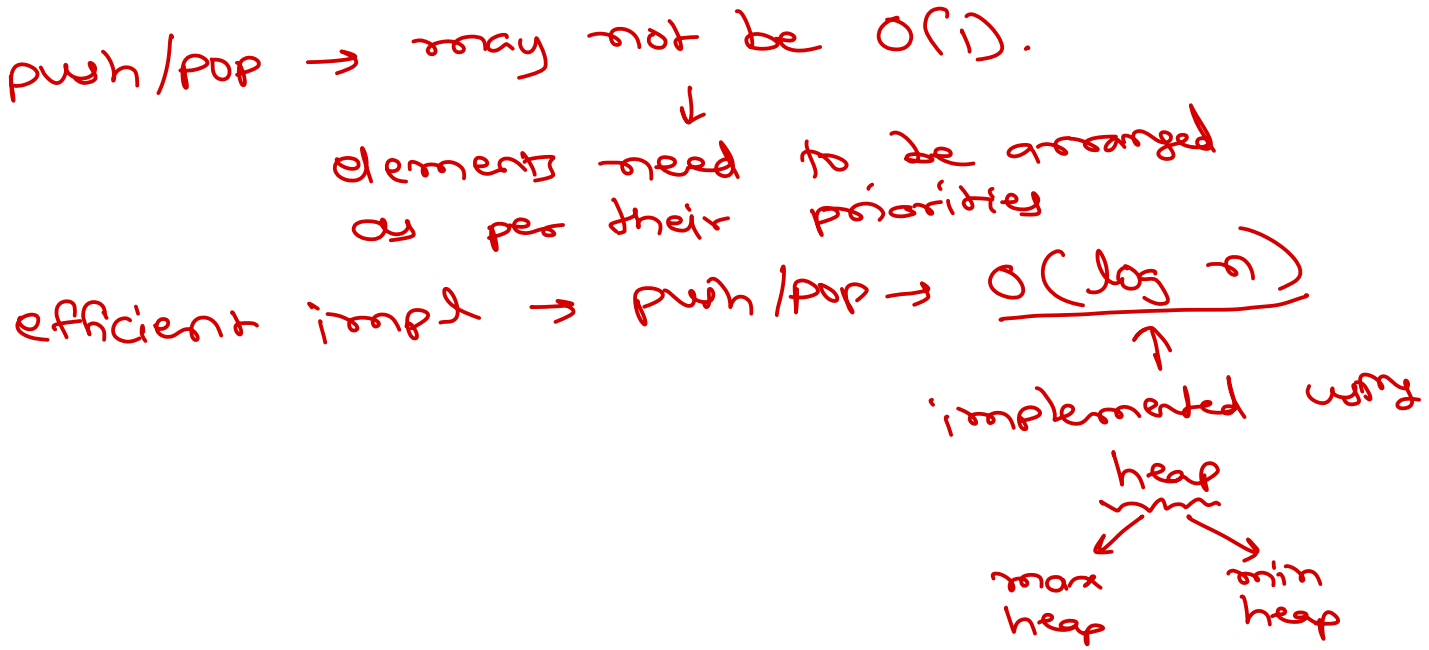
del first

addlast

dellast

Priority queue

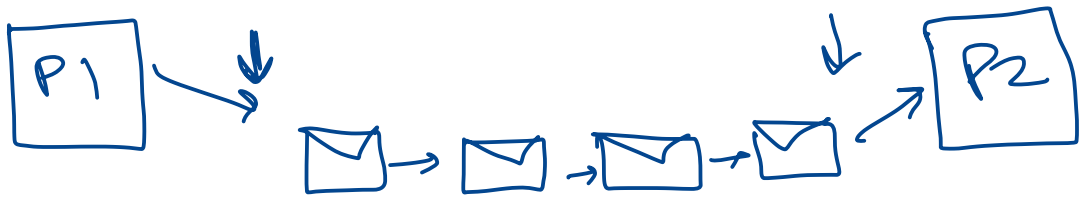
- In priority queue, element with highest priority is removed first.



applications

real life : queue at ticket counter.

1. CPU scheduling
   - FCFS, SJF, RR, ...
     - FCFS ↑ FIFO
     - SJF, RR ↑ priority queue
2. page replacement
   - FIFO, LRU, ...
3. message queue (IPC)



4. processes wait for IO devices in queue
   - ↓ waiting queue.

Stack → push/pop op from same end(top). - LIFO

arr

init:

top = -1

peek:

return arr[top];

22

pop:

33

top--;

44

push:

top++;

arr[top] = val;

55

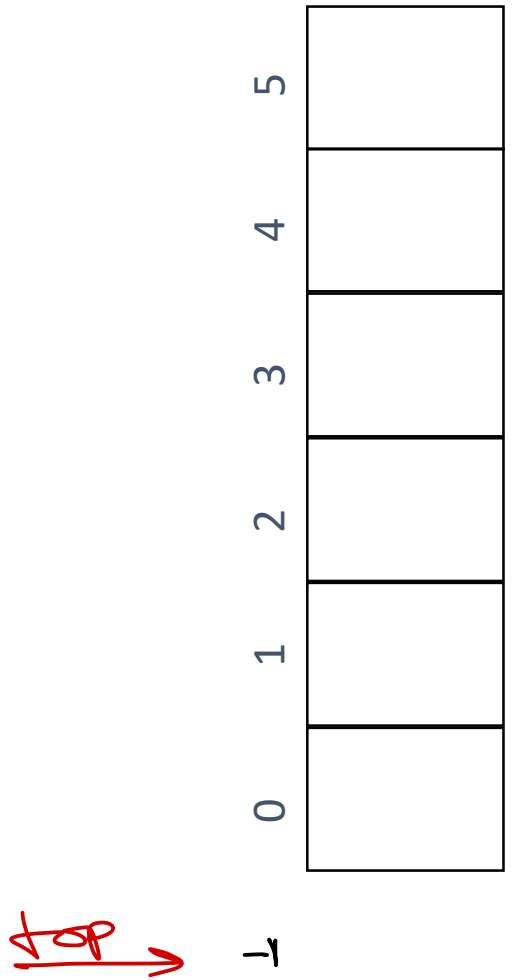
isempty:

top == -1

11

isFull:

top == size-1



Stack / Queue in Java collections

- class java.util.Stack<E>
  - E push(E);
  - E pop();
  - E peek();
  - boolean isEmpty();

- interface java.util.Queue<E>
  - boolean offer(E e); push
  - E poll(); pop
  - E peek();
  - boolean isEmpty();

LinkedList<>

infix, postfix, prefix

5 + 9 - 4 * (8 - 6 / 2) + 1 $ (7 - 3)

(6) (7) (5) (2) (1) (8) (4) (3)

* notations for math expression.

a + b → infix ← human → parenthesis

+ a b → prefix
a b + → postfix
} Computer → no parenthesis

precedence

1. ( )
2. $
3. * / L → R
4. + - L → R

a) 5 + 9 - 4 * (8 - 6 2 /) + 1 $ (7 - 3)

b) 5 + 9 - 4 * 8 6 2 / - + 1 $ (7 - 3)

c) 5 + 9 - 4 * 8 6 2 / - + 1 $ 7 3 -

d) 5 + 9 - 4 * 8 6 2 / - + 1 7 3 - $

e) 5 + 9 - 4 8 6 2 / - * + 1 7 3 - $

f) 5 9 + - 4 8 6 2 / - * + 1 7 3 - $

g) 5 9 + 4 8 6 2 / - * - + 1 7 3 - $

h) 5 9 + 4 8 6 2 / - * - 1 7 3 - $ +

(6) (7) (5) (2) (1) (8) (4) (3)

5 + 9 - 4 * (8 - 6 / 2) + 1 $ (7 - 3)

+ - + 5 9 * 4 - 8 / 6 2 $ 1 - 7 3

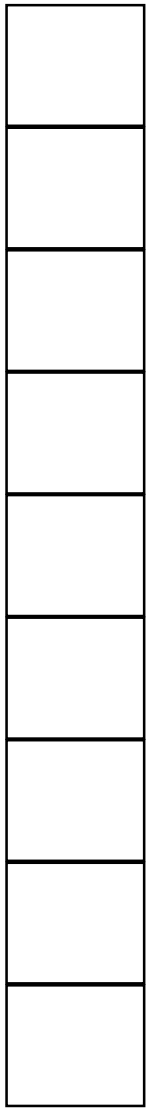
Infix to Postfix

5 + 9 - 4 * (8 - 6 / 2) + 1 $ (7 - 3)

stack of operators

- 5 + 9 – 4 * (8 – 6 / 2) + 1 $ (7 – 3)

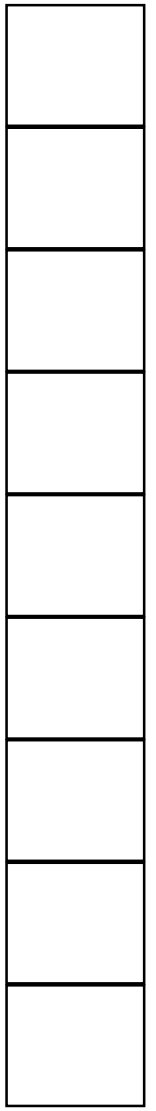
5 9 + 4 8 6 2 / - * - 1 7 3 - $ +



① traverse infix expr left to right.
② if operand found, append to postfix result.
③ if operator found, push on stack.
   * if priority of topmost op in stack >= priority of cur op, then pop & append to postfix.
④ when all symbols from infix completed, pop all ops from stack & append to postfix.
⑤ if opening (found, push on stack.
⑥ if closing) found, pop all ops from stack & append to postfix, until (is found. also pop & discard (.

Infix to Prefix

- 5 + 9 – 4 * (8 – 6 / 2) + 1 $ (7 – 3) = -5

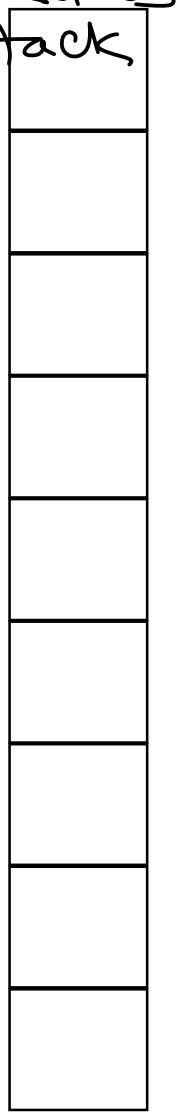


Postfix Evaluation

5 9 + 4 8 6 2 / - * - 1 7 3 - $ +

- 5 9 + 4 8 6 2 / - * - 1 7 3 - $ +

operand stack



① parse postfix from left to right;

② if operand, push on stack.

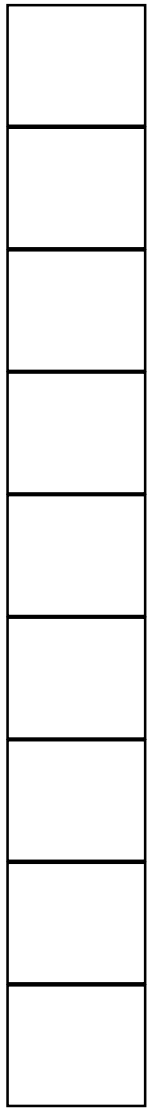
③ if operator, pop two operands, calc result & push it on stack.

first popped is 2nd operand
& second popped is 1st operand

④ when all symbols are done, pop final result from stack & return. -5

Prefix Evaluation

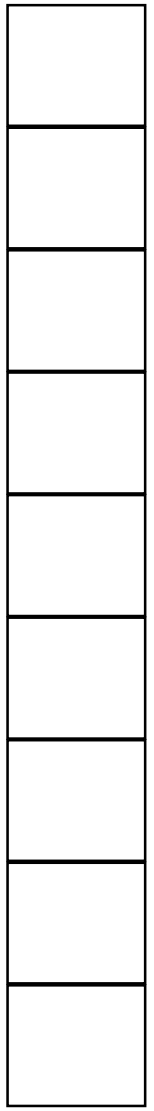
- + - + 5 9 * 4 – 8 / 6 2 $ 1 – 7 3



Postfix to Infix

- While there are input symbol left
- Read the next symbol from input.
- If the symbol is an operand , Push it onto the stack.
- Otherwise, the symbol is an operator.
- If there are fewer than 2 values on the stack
- Show Error
- Else
- Pop the top 2 values from the stack.
- Put the operator, with the values as arguments and form a string.
- Encapsulate the resulted string with parenthesis.
- Push the resulted string back to stack.
- If there is only one value in the stack
- That value in the stack is the desired infix string.
- If there are more values in the stack
- Show Error

- a b c - + d e – f g – h + / *



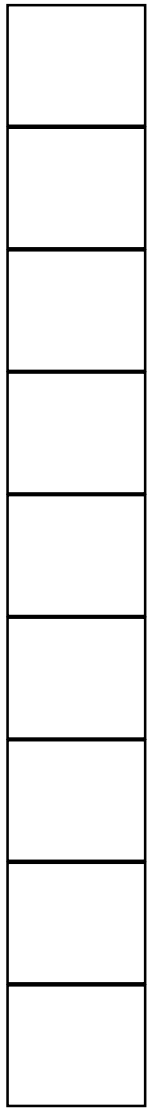
Prefix to Postfix

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
- Create a string by concatenating the two operands and the operator after them.
- string = operand1 + operand2 + operator
- And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

- * + A B – C D

Parenthesis Balancing

- 5 + ([9 – 4] * (8 – { 6 / 2 }))



Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>