Stacks and Queues

by Raghav Goel

Stacks

A stack is a container that stores elements in a **last-in first-out (LIFO)** order.

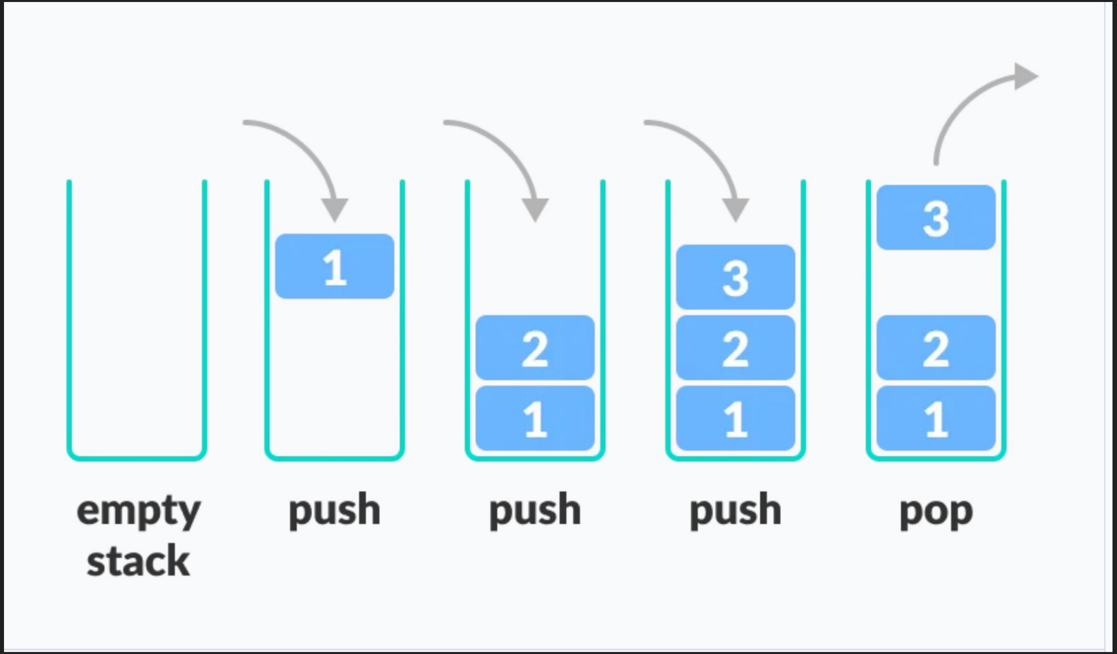


Image credits:-
https://www.programiz.com/dsa/stack

Syntax

Declaration :-

```
stack<Type> st; // a stack which can store elements of datatype Type
```

Type → the Type of elements that have to be stored inside the stack

Example :-

```
stack<int> st1; // this stack can store integers inside it

// stacks can store complicated Types also, such as

stack<pair<int, string>> st2;
```

Methods / Functions

void push(Type val) { … } → pushes val to the top of the stack

Type pop() { … } → pops out the top element of the stack

Type top() { … } → returns the value of the top element of the stack

bool empty() { … } → return true if the stack is empty else false

int size() { … } → returns the size of the stack

Methods / Functions

Time Complexity of all these methods is O(1).

Note: pop() and top() will throw exception if the stack is empty which can result in runtime error

Queues

A queue is a container that stores elements in a **first-in first-out (FIFO)** order.

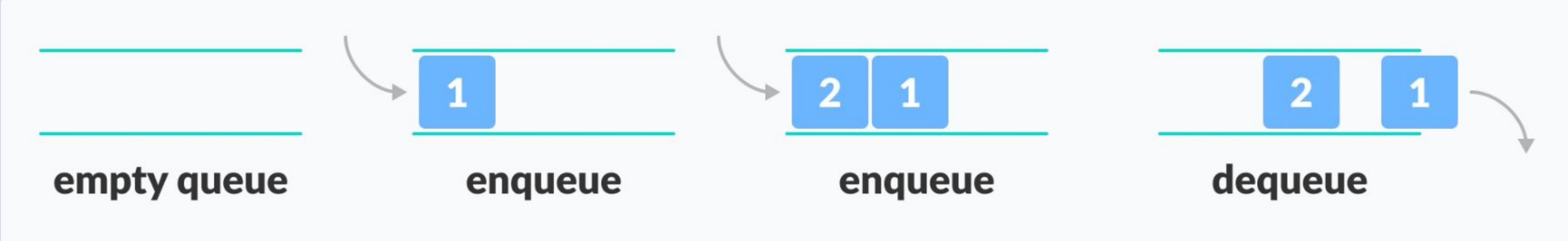


Image credits :-
https://www.programiz.com/dsa/queue

Syntax

Declaration :-

```
queue<Type> q; // a queue which can store elements of datatype Type
```

Type → the Type of elements that have to be stored inside the queue

Example :-

```
queue<int> q1; // this queue can store integers inside it

// queues can also store complicated Types, such as

queue<pair<int, string>> q2;
```

Methods / Functions

void push(Type val) { … } → pushes val to the back of the queue

Type pop() { … } → pops out the element from the front of the queue

Type front() { … } → returns the value of the element at the front of the queue

Type back() { … } → returns the value of the element at the back of the queue

empty() and size() functions are same as in stack

Methods / Functions

Time Complexity of all these functions is O(1).

Note: pop(), front() and back() will throw exception if the queue is empty which can result in runtime error.

Deque

Deque stands for Double Ended Queues

They are not allocated contiguous memory locations.

Unlike queues, they allow both insertion and deletion at both ends.

Although vector also allows us for insertion and deletion at both ends, these operations on deques are more efficient.

But the random access in deque is marginally slower than vectors.

Syntax

Declaration :-

```
deque<Type> deq;
```

Type → the Type of elements that have to be stored inside the deque

Example :-

```
deque<int> deq1;

deque<pair<int,int>> deq2;
```

Methods / Functions

- push_back(val) and push_front(val)
- pop_back() and pop_front()
- front() and back()
- size()
- empty()

All of these methods work in O(1) time complexity.

Problems

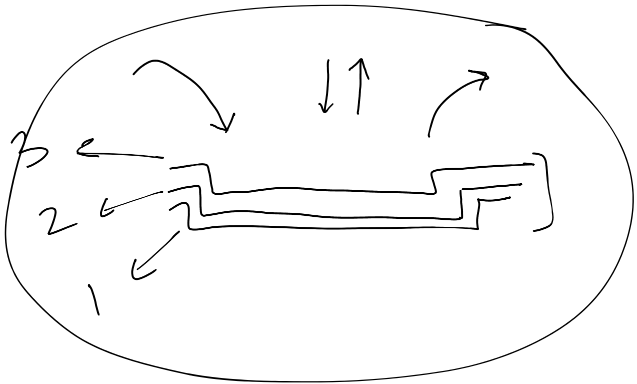
- https://leetcode.com/problems/implement-queue-using-stacks/
- https://leetcode.com/problems/implement-stack-using-queues/
- https://leetcode.com/problems/daily-temperatures/ (Monotonic Stack)

Resources

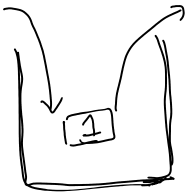
Search on Google Chrome :)

Monotonic Stacks and Queues - https://cp-algorithms.com/data_structures/stack_queue_modification.html

THANKS FOR WATCHING

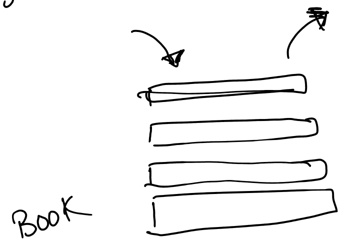


Last in First Out



LIFO

stack data structure



vector → data structure

you can store some elements of some specific type

vector<int> vec;

- push_back()
- pop_back()
- back()

stack <int> st;

- push()
- pop()
- top()
- size()
- empty()

array

fast but
not so
convenient

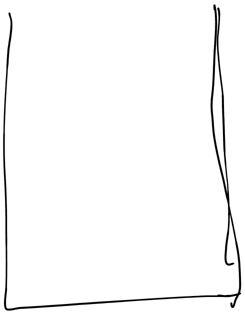
stacks

some specific

vectors

good
enough
and
fast

empty()



push (10) → void   O(1)

push (20) → void   O(1)

pop ( ) → void   O(1)

top ( ) → 10   O(1)

empty() → false   O(1)

size() → 1

pop()
top()   pop() → exception
↓
runtime error

vector <int> v = {10};

v. pop_back();

cout<< v.empty() << endl;

→ ✓true / false ✓

exception handling X

v. front()   v. back()

v. pop_back ();

tunnel

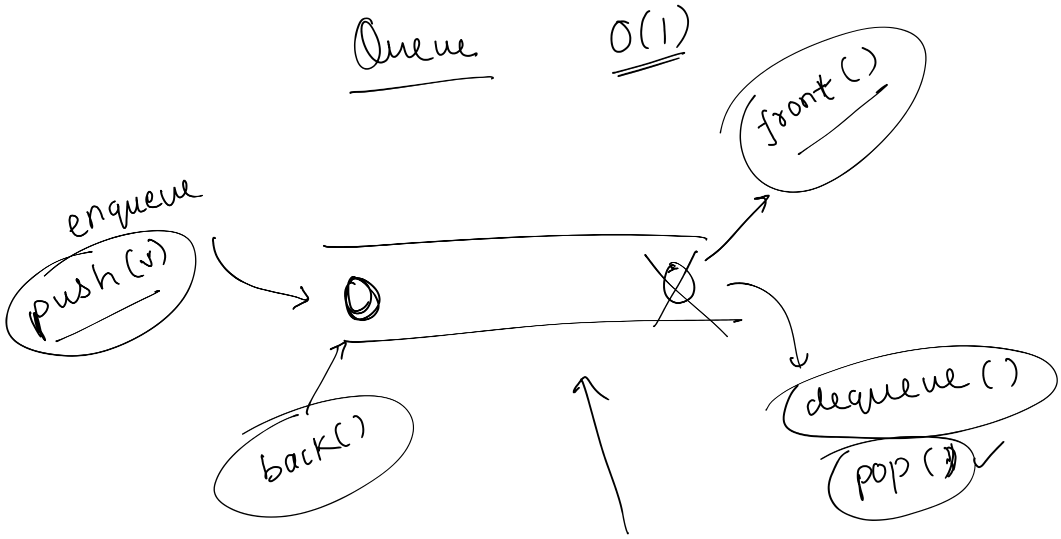
queue at bank

indexing X

Queue   O(1)

t()

Queue   O(1)



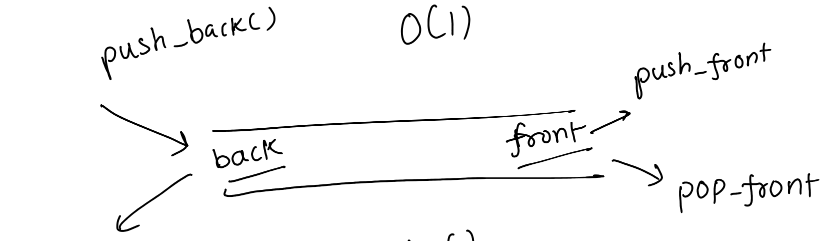
size()
empty()

queue   (data structure)

dequeue()   method

deque()   data structure

Deque



empty()
size()

indexing   deque <int> deq;

⋮

slower?   deq[2] >>> vector

O(1)

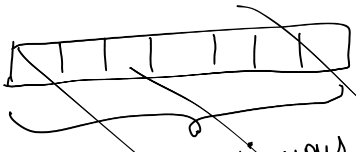
vector ✓

stack ✓

✓

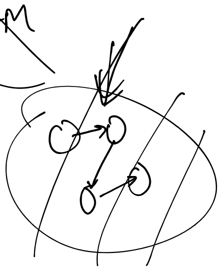
queue

deque ✓



contiguous memory being allocated in RAM

linked list



arrays

push (v)

pop ( )

top( )

size( )

empty( )

int top = 1 ;

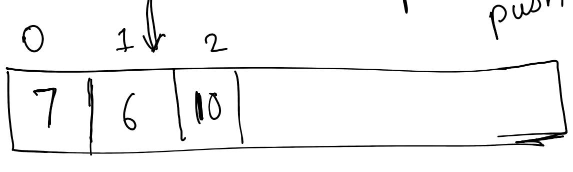
push (7)

push (6)

push (10)

pop( )

arr[ ] =



top( ) → 6

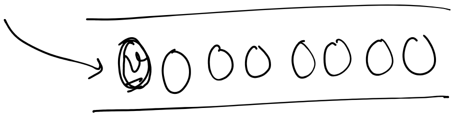
empty( ) → true / false

size( ) → top +1

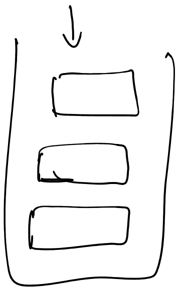
Implement a Stack using 1 queues

push(v) pop() top() empty()

O(1) O(n) O(1) O(1)

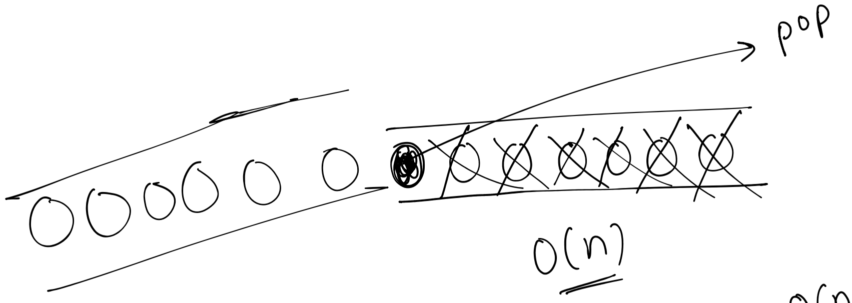


O(n)



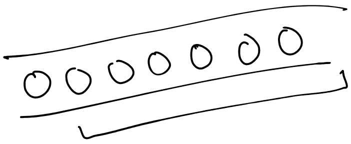
aux queue

q.back()



O(n)

7-1=6



Daily Temperatures

0 1 2 3 4 5 6 7

[73, 74, 75, 71, 69, 72, 76, 73]

for every index, find out where is ~~greater~~ element on the right for every index, find out where is the next greater element on the right

0 1 2 3 4 5 6 7

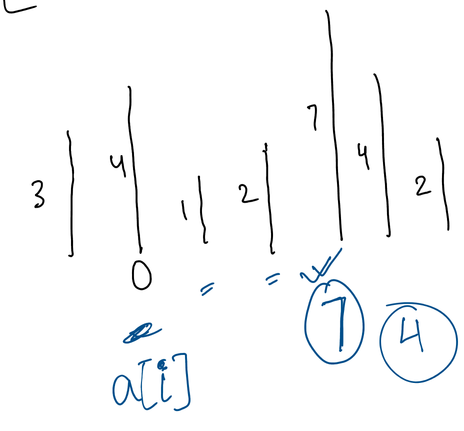
[1 , 1 , 4 , 2 ,]

7 6 5 4 3 2 1

~~$O(N^2)$~~

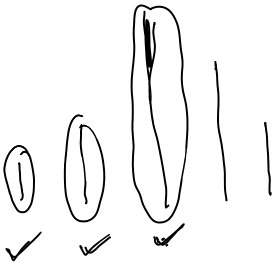
```
for (i=0 ; i<n; i++ ){
    for (j=i+1  ; j<n; j++){
        a[j] > a[i]
    }
}
```

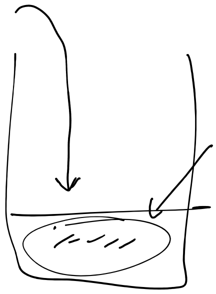
[3 , 4, 1 , 2, 7 , 4 , 2]



1 2 7 8

[7 , 1, 2, 7, 4 , 2, 8]





[1, 4, 1, 2, 7, 4, 2, 8]

←

~~queue~~

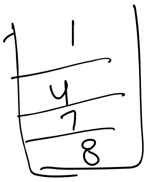
~~deque~~

data structure

[1, 4, 7, 8]

stack ✓

vector ✓



monotonic stack

push on the left

pop from the left

check the leftmost element

vectors are iterable

```
for ( auto it : vec){
    . - - -
}
```

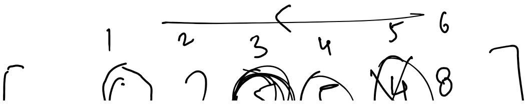
vectors

deque

stacks and queues

answer[6] = 0

[5] = 1



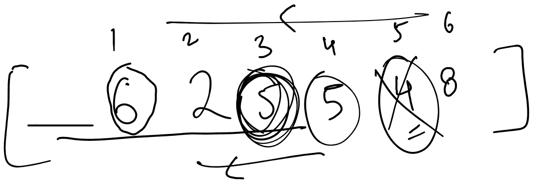
6 - 5 = 1

6 - 4 = 2

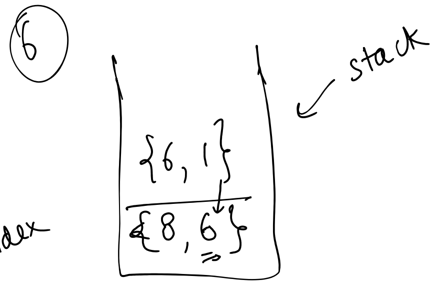
6 - 3 = 3

3 - 2 = 1

answer[6] -
answer[5] = 1
answer[4] = 2
answer[3] = 3
answer[2] = 1



6 - 3 = 3
3 - 2 = 1



pair<int,int>
↓ value
↓ index

intuition

1-2 hours

~~code~~ ~~impleme~~

Q1. implement stacks using arrays

Q2. " queues " "

Q3. implementing stack using queues

Q4. " queues " stacks (follow up)

Q5. daily temperatures

next greater on left
" " " right
" smaller " left
" " " right

stacks / Queues