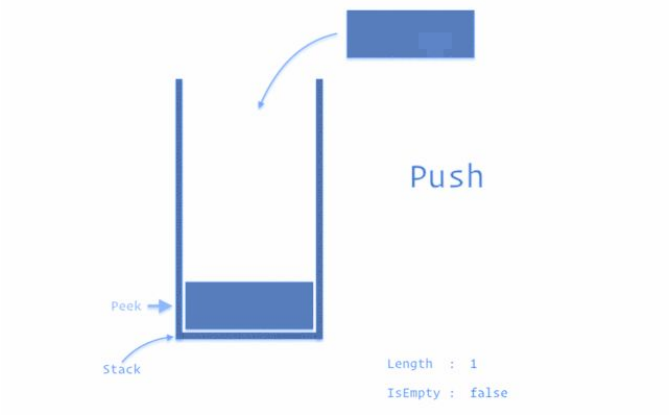
Stacks, Queues and Monotonicity

Pre-requisites

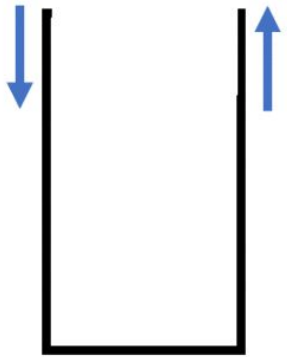
- Linear data structures array/list
- Linked List

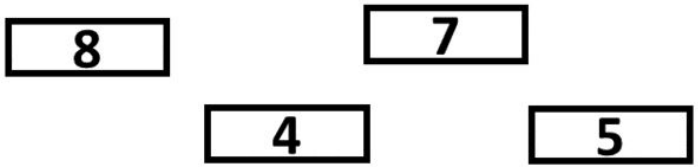
Stacks



Introduction To Stack

Stack data structure is a linear data structure that accompanies a principle known as **LIFO** (Last In First Out) or FILO (First In Last Out).





Real Life Example

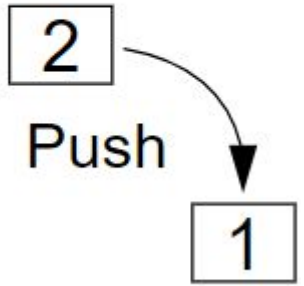
Stack of plates

Stack of books

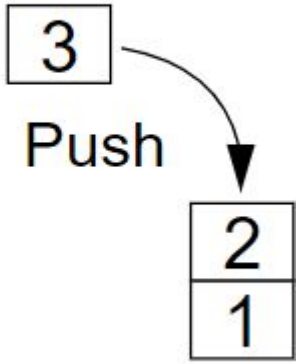
Stack Operations

Push operation

- Add an element to the top of the stack



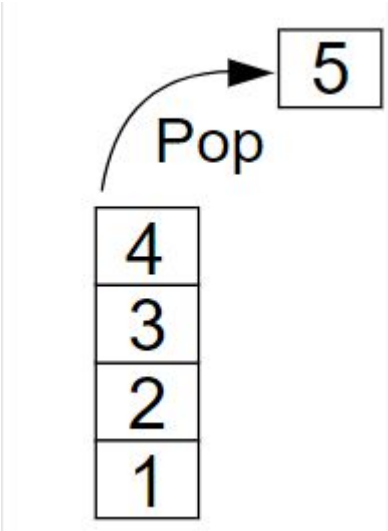
Stack Operations



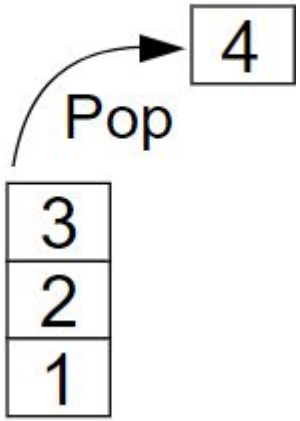
Stack Operations

Pop operation

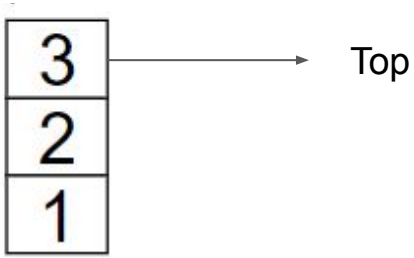
- Remove element from the top of the stack



Stack Operations



Stack Operations



Peek operation

- returning the top element of a stack.

Is empty()

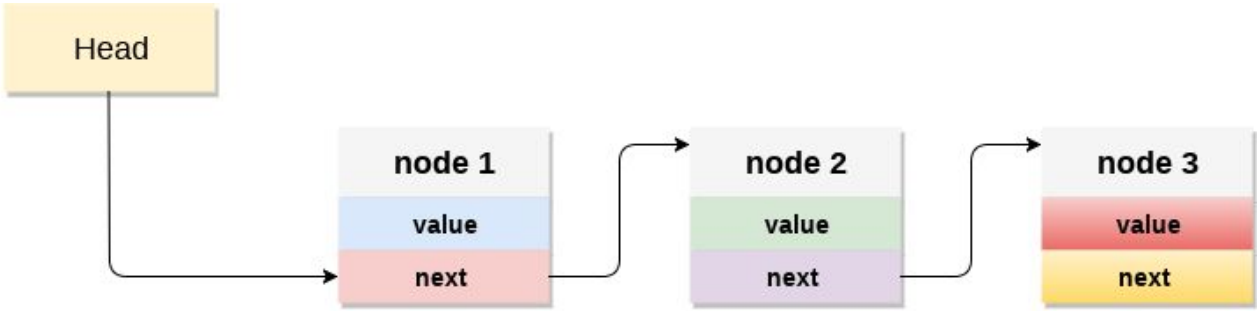
- Check if the list is empty or not.
- If it's empty return True else False.

Practice

Problem

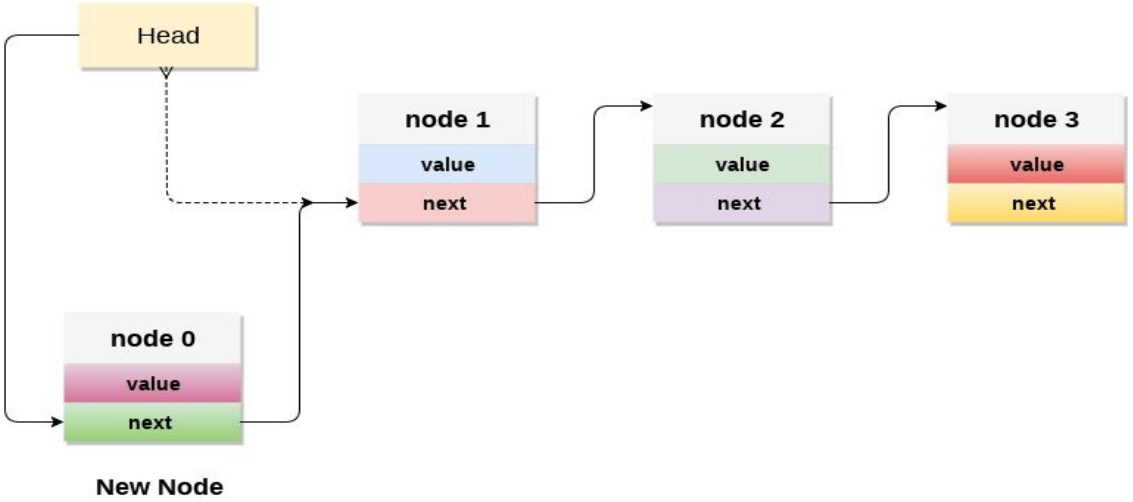
How can we implement Stack?

Implementing stack using linked list



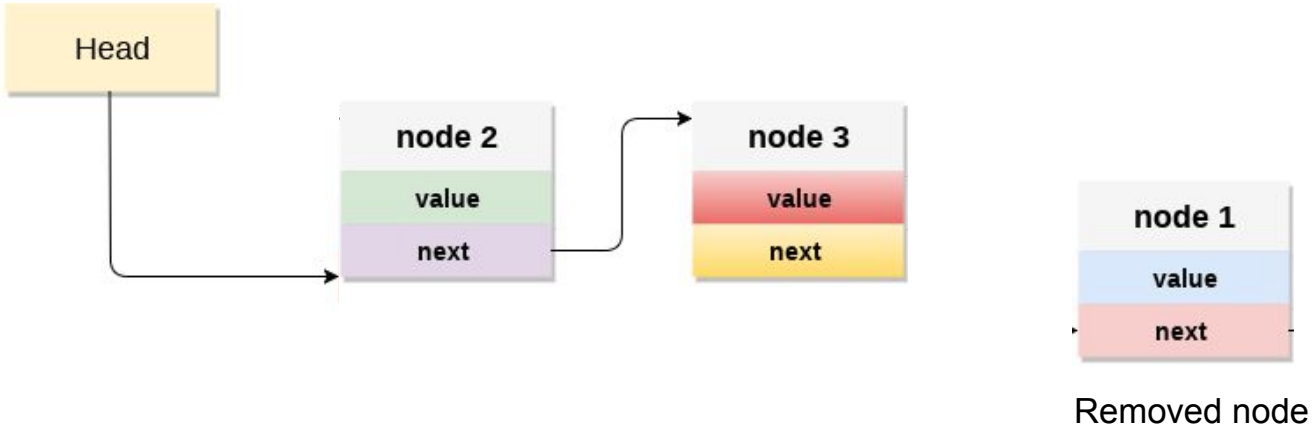
Push operation

- Initialise a node
- Update the value of that node by data i.e. node.value = data
- Now link this node to the top of the linked list i.e. node.next = head
- And update top pointer to the current node i.e. head = node



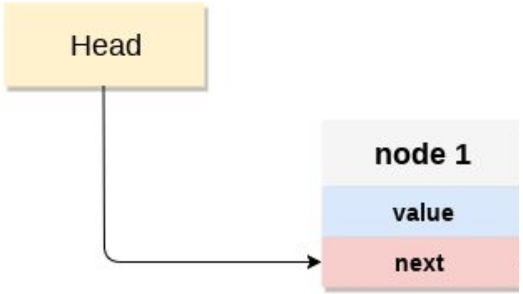
POP operation

- First Check whether there is any node present in the linked list or not, if not then return
- Otherwise make pointer let say temp to the top node and move forward the top node by 1 step
- Now free this temp node



Top operation

- Check if there is any node present or not, if not then return.
- Otherwise return the value of top node of the linked list which is the node at **Head**



Implementation

```
class Node:
     def __init__(self, data):
          self.data = data
          self.next = None

class Stack:
     def __init__(self):
          self.head = None

     def isempty(self):
          if self.head == None:
               return True
          else:
               return False

     def peek(self):
          if self.isempty():
               return None
          else:
               return self.head.data
```

```
def push(self, data):

     if self.head == None:
          self.head = Node(data)
     else:
          new_node = Node(data)
          new_node.next = self.head
          self.head = new_node

def pop(self):

     if self.isempty():
          return None
     else:
          popped_node = self.head
          self.head = self.head.next
          popped_node.next = None
          return popped_node.data
```

Pair Programming

Problem

Time and space complexity

- Push
  - Time complexity - ___?
- Pop
  - Time complexity - ___?
- Peek
  - Time complexity - ___?
- isEmpty()
  - Time complexity - ___?

Time and space complexity

- Push
  - Time complexity - O(1)
- Pop
  - Time complexity - O(1)
- Peek
  - Time complexity - O(1)
- isEmpty()
  - Time complexity - O(1)

Applications of Stack

Practice

Problem

Reflection: Stack can help you simulate deletion of elements in the middle in O(1) time complexity

Pair Programming

Problem

Reflection: stack can help you defer decision until some tasks are finished.

The bottom of the stack waits on the top of stack until they are processed

Common PitFalls

- Popping from empty list
  - This will throw index out of range error
- Null pointer exception if we are using linked list

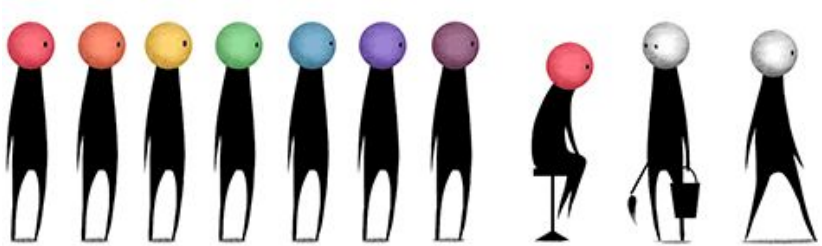
Runtime Error

```
IndexError: list index out of range
    if i == open_close[stack[-1]]:
Line 6 in isValid (Solution.py)
    ret = Solution().isValid(param_1)
Line 32 in _driver (Solution.py)
    _driver()
Line 43 in <module> (Solution.py)
```

Common PitFalls

- Stack overflow
    - May be not in python but In other programming language
    - Pushing to a full stack

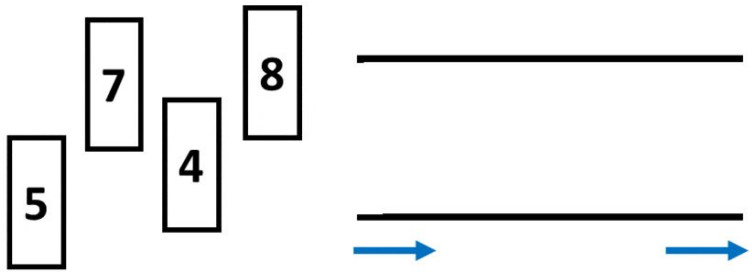
Queues



Introduction

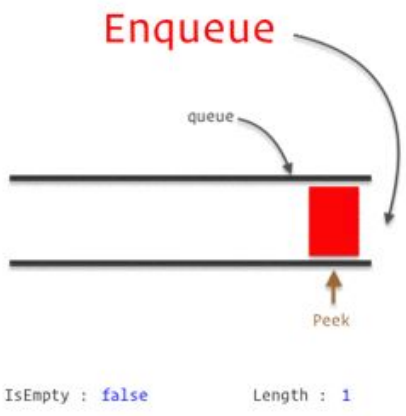
A collection whose elements are added at one end (the **rear**) and removed from the other end (the **front**)

- Uses **FIFO** data handling



Real Life Example

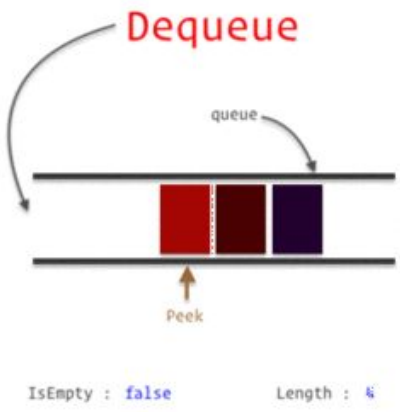
Queue Operations



Enqueue (Append)

- Add an element to the tail of a queue
- **F**irst **I**n

Queue Operations



Dequeue (Popleft)

- Remove an element from the head of the queue
- **F**irst **O**ut

Practice

Problem

Implementing Queue

Using an array to implement a queue is significantly harder than using an array to implement a stack. **Why**?

What would the time complexity be?

Implementing Queue with List

```
def __init__(self):
   self.queue = []
   self.headIndex = 0

def append(self, value: int):
   self.queue.append(value)

def pop(self) -> int:
   if self.headIndex < len(self.queue):
        val = self.queue[self.headIndex]
        self.headIndex += 1
        return val
```

Implementing Queue

- Either linked list or list can be used with careful considerations
- In practice, prefer to use built-in or library implementations like `deque()`
- Internally, `deque()` is implemented as a linked list of nodes

```
.pop()
.append()
.popleft()
.appendleft()
```

Implementation (built-in)

```
from collections import deque

# Initializing a queue

queue = deque()

# Adding elements to a queue

queue.append('a')

queue.append('b')

# Removing elements from a queue

print(queue.popleft())

print(queue.popleft())

# Uncommenting queue.popleft()

# will raise an IndexError

# as queue is now empty
```

We can also use it the other way around by using;

- `.appendleft()`
- `.pop()`

Time and space complexity

- Append
  - Time complexity - ____?
- Popleft
  - Time complexity - ____?
- Peek
  - Time complexity - ____?
- isEmpty()
  - Time complexity - ____?

Time and space complexity

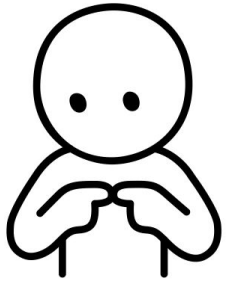
- Append
  - Time complexity - O(1)
- Popleft
  - Time complexity - O(1)
- Peek
  - Time complexity - O(1)
- isEmpty()
  - Time complexity - O(1)

Applications of Queue

Practice

Problem

Reflection: Queue helps solve problems that need access to the “first something”



Common Pitfalls

Africa To Silicon Valley

Africa To Silicon Valley

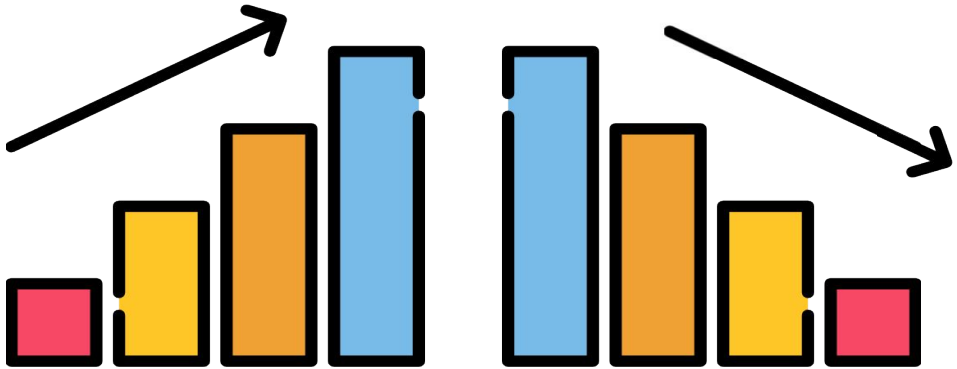
Not handling edge cases

- Popping from an empty queue
  - ```
    if queue:

        queue.popleft()
    ```

- Appending to a full queue
  - ```
    if len(queue) < capacity:

        queue.append(val)
    ```

Check point

Monotonicity



Practice

Problem

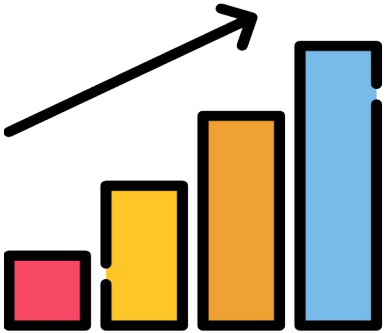
Basic Concepts

- A stack whose elements are **monotonically** increasing or decreasing.
- Useful when we're looking for the next larger/smaller element
- For a mono-decreasing stack:
    - we need to pop smaller elements before pushing.
    - it keeps tightening the result as lexicographically greater as possible. (Because we keep popping smaller elements out and keep greater elements).

Practice

Problem

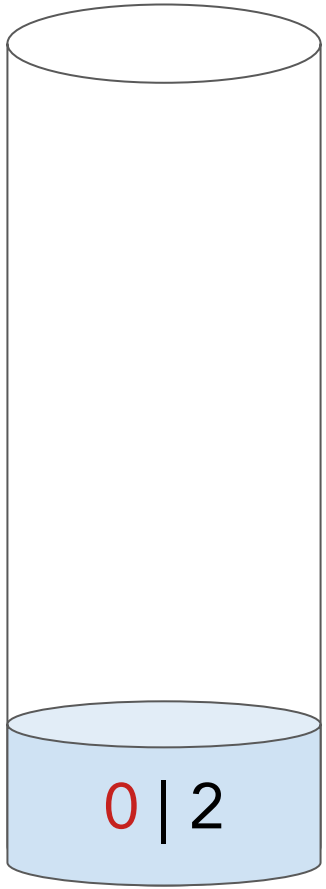
Monotonic Stack Application



- It gives you how far a value spans as a **maximum** or **minimum** in the given array.

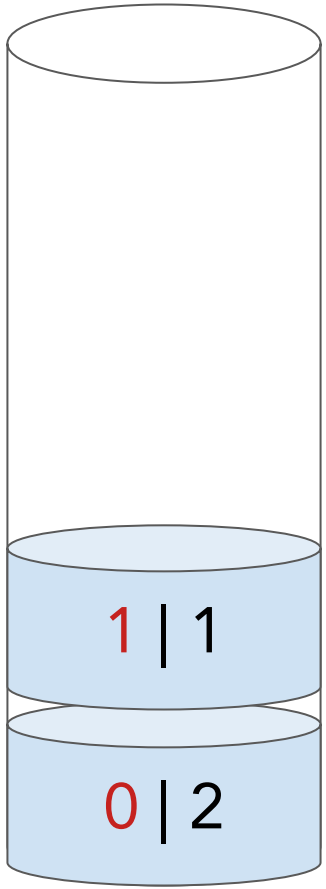
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



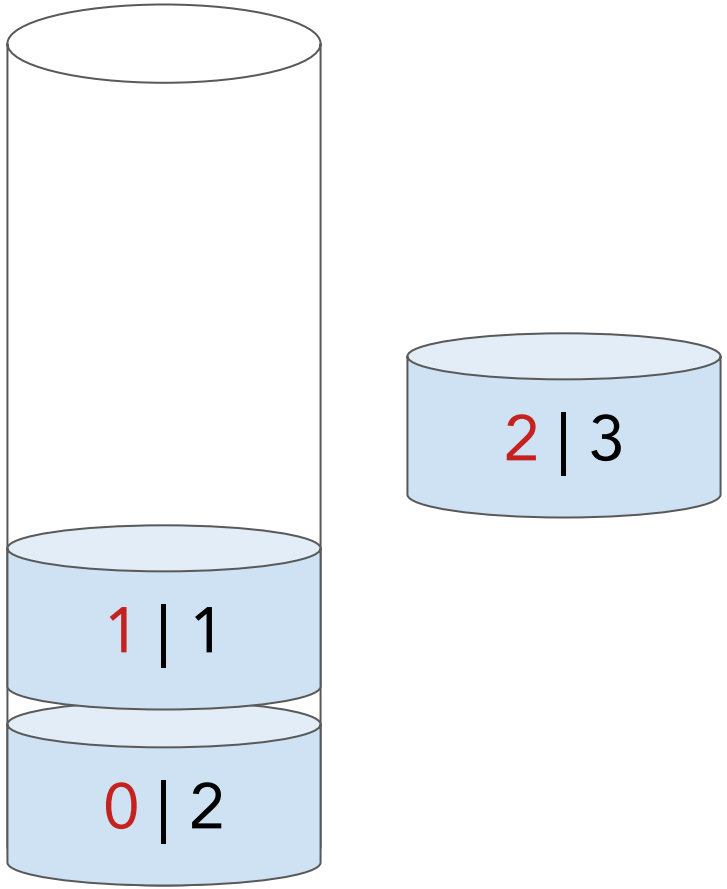
[2, 1, 3, 5, 6, 1]
 0 1 2 3 4 5

Monotonically decreasing stack



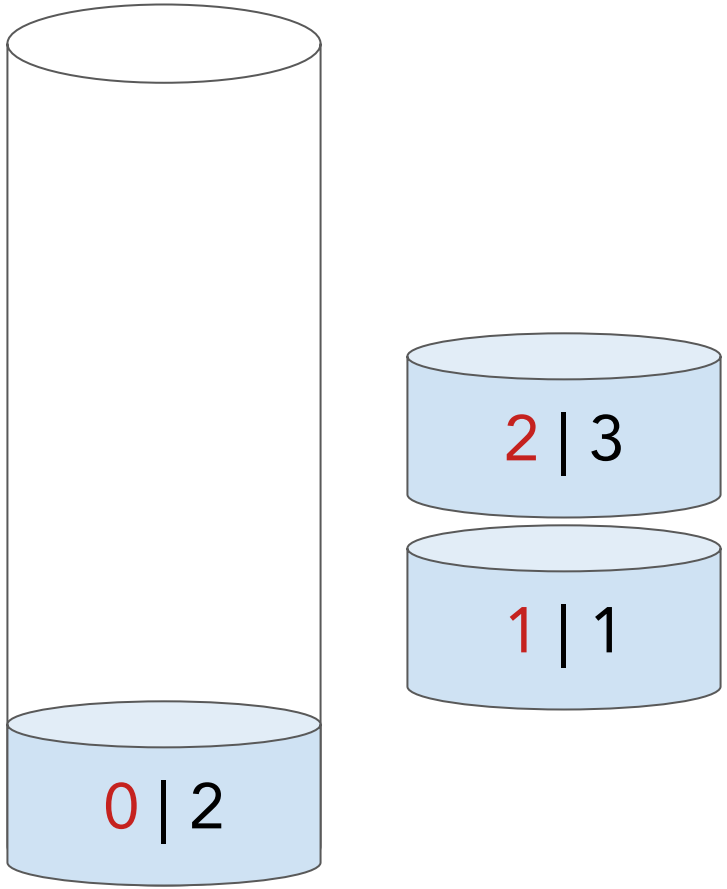
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



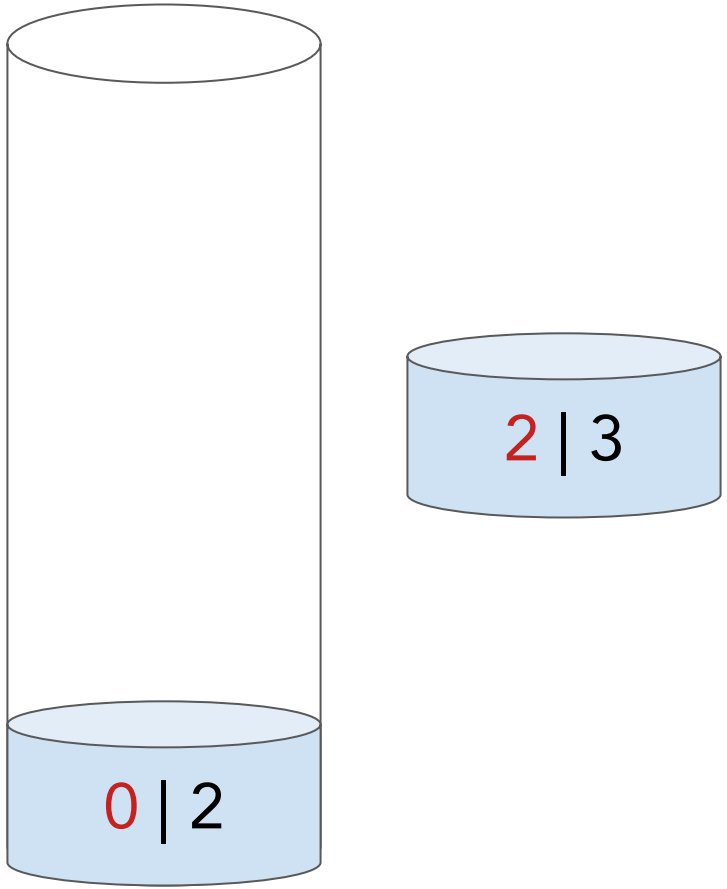
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



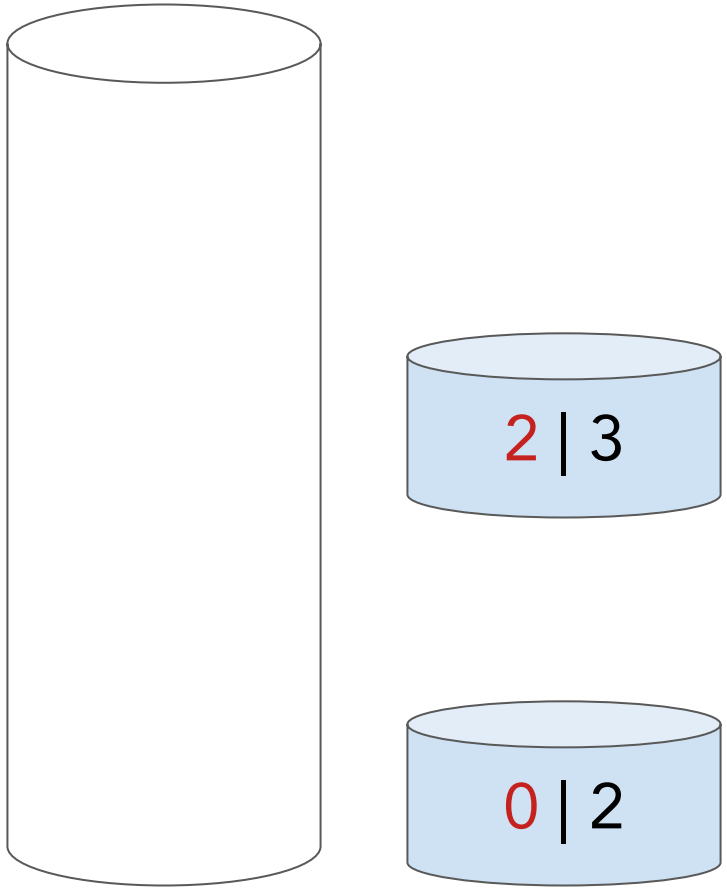
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack

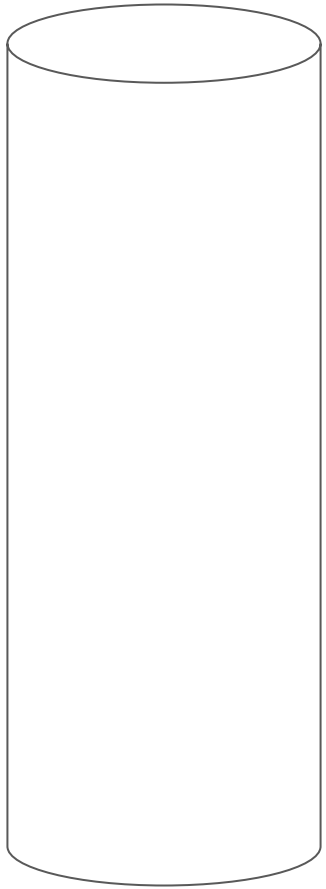


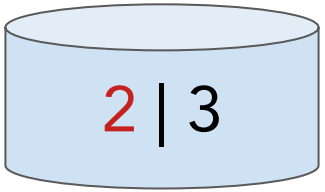
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

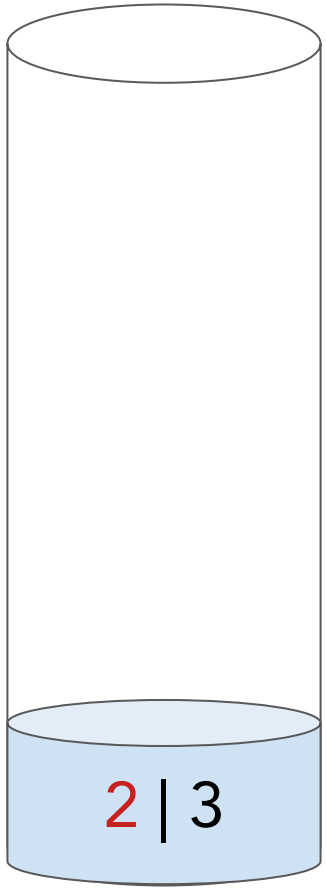




Monotonically decreasing stack

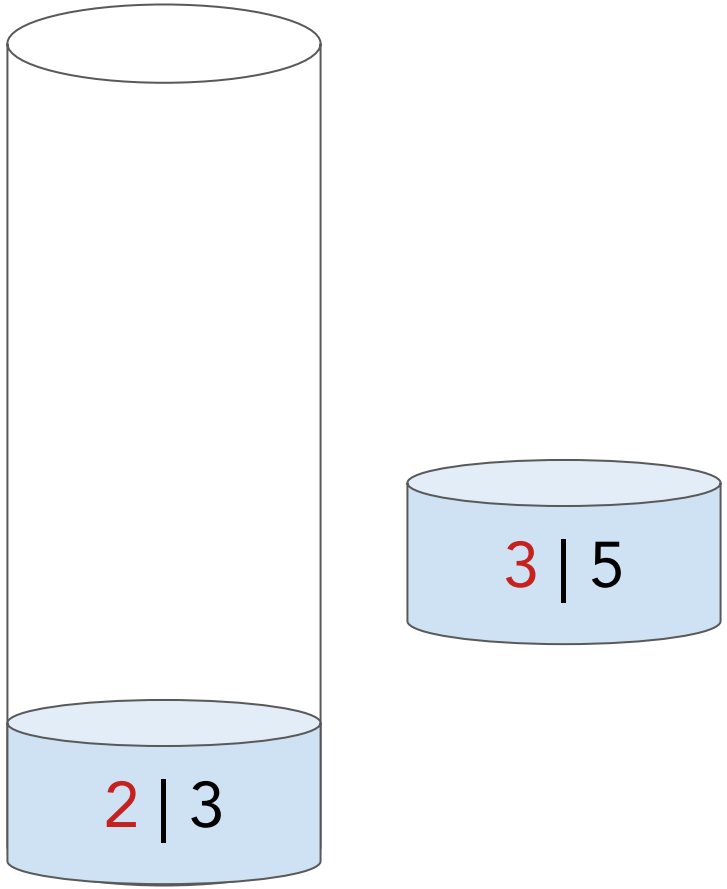
[2, 1, 3, 5, 6, 1]
 0 1 2 3 4 5

Monotonically decreasing stack



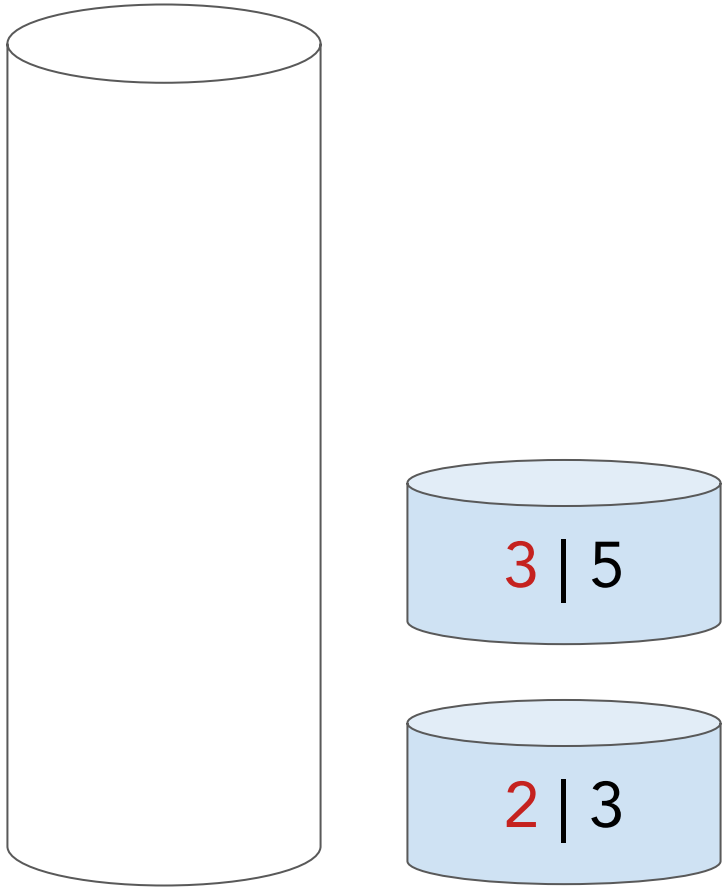
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



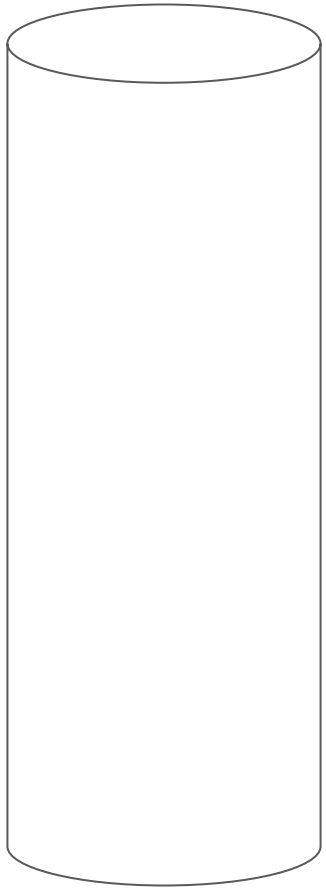
[2, 1, 3, 5, 6, 1]
 0 1 2 3 4 5

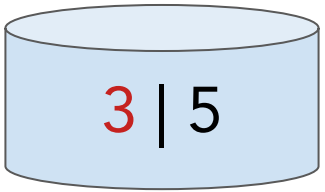
Monotonically decreasing stack



[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

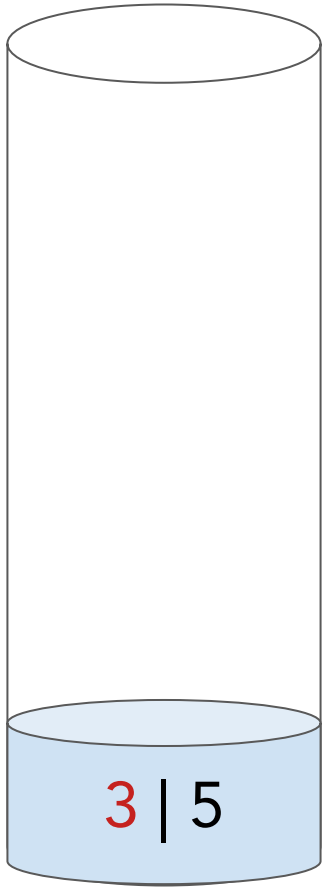
Monotonically decreasing stack





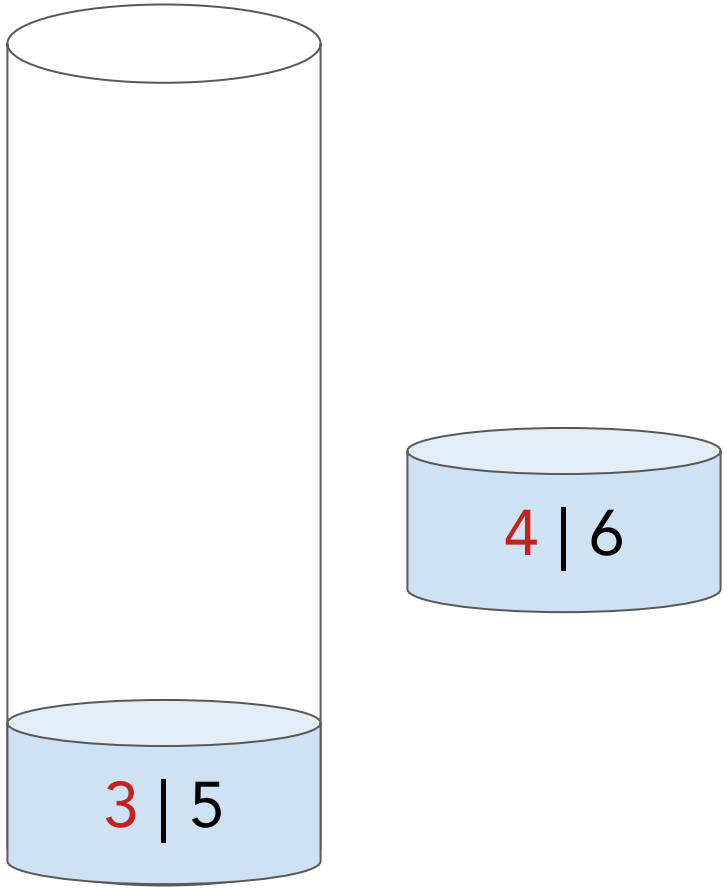
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



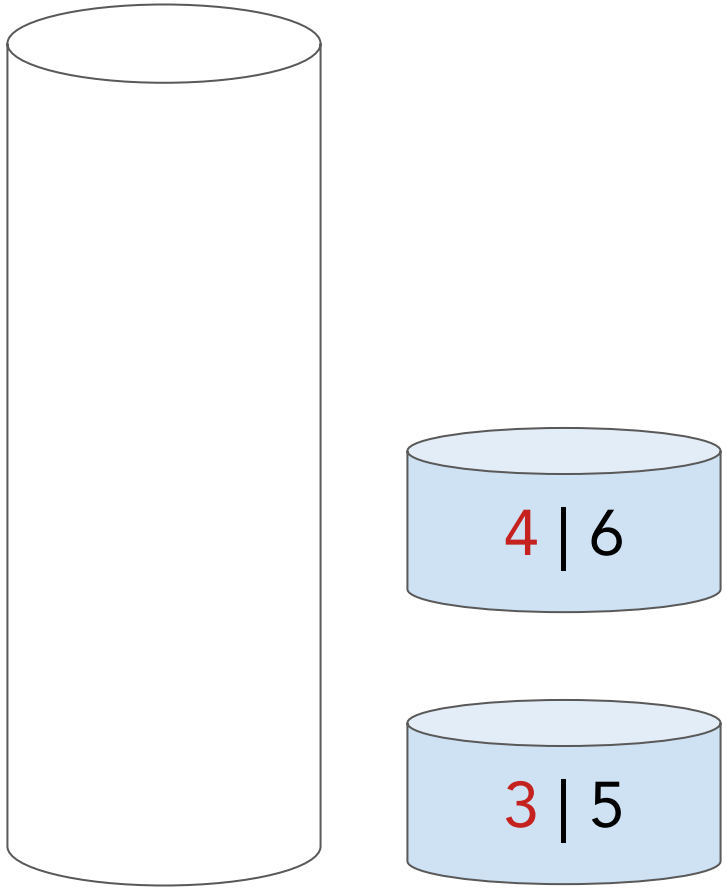
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



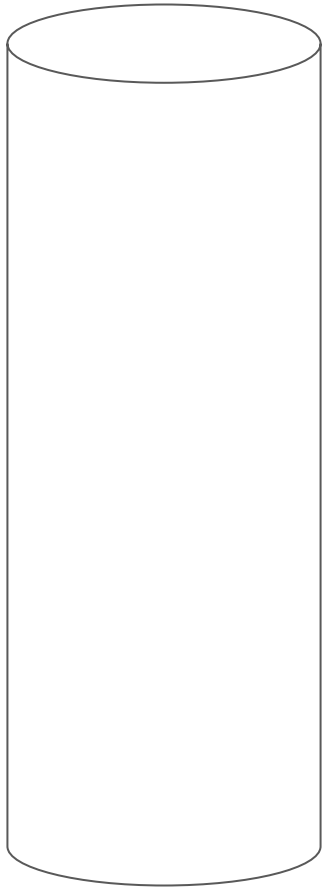
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

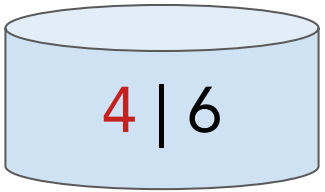
Monotonically decreasing stack



[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

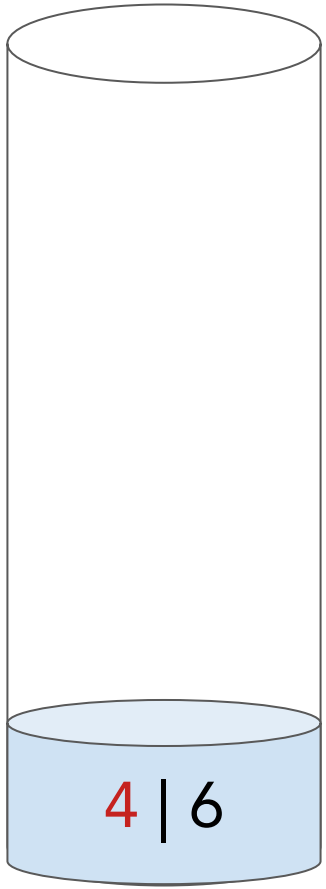
Monotonically decreasing stack





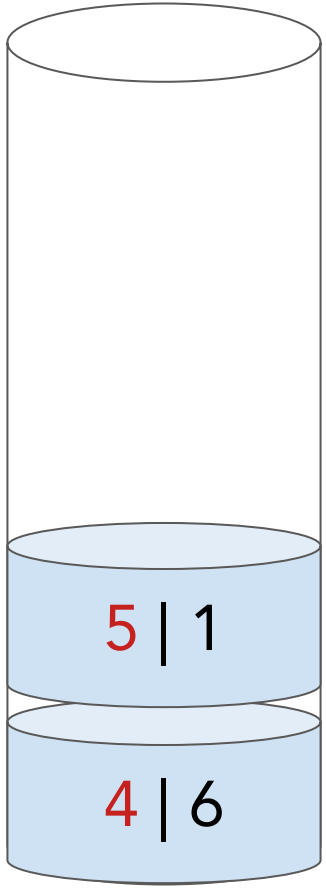
[2, 1, 3, 5, 6, 1]
 0 1 2 3 4 5

Monotonically decreasing stack



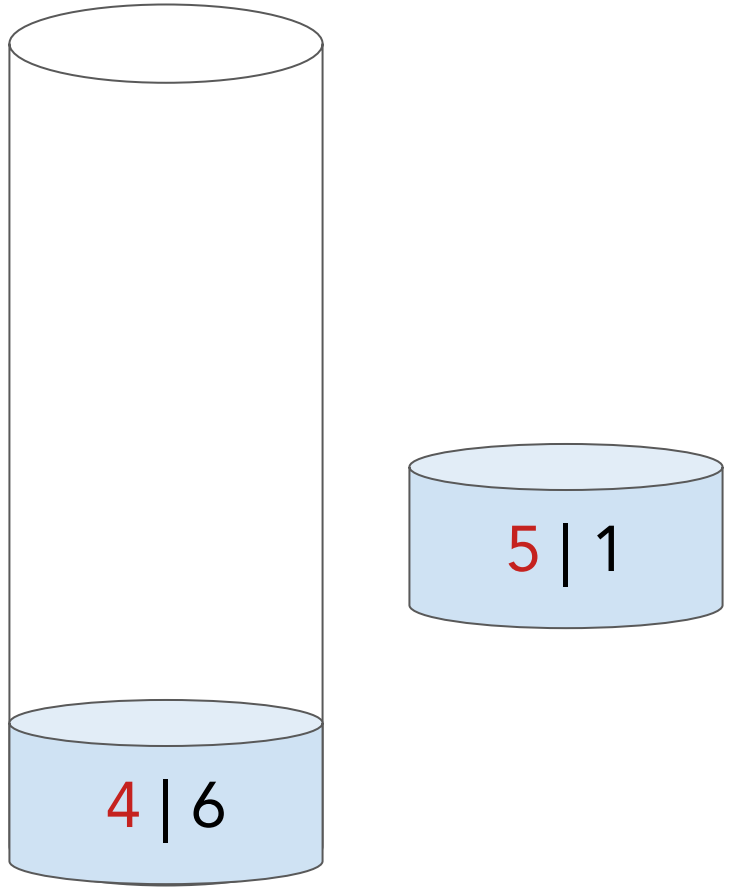
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



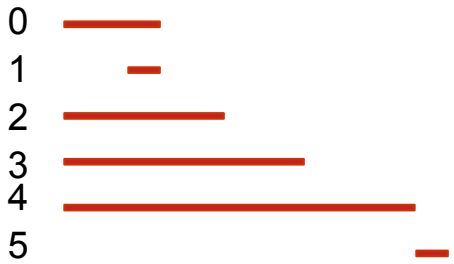
[2, 1, 3, 5, 6, 1]
0 1 2 3 4 5

Monotonically decreasing stack



[2, 1, 3, 5, 6, 1]

0 1 2 3 4 5



Index Spans:

1: 1 <= ... < 2

0: 0 <= ... < 2

2: 0 <= ... < 3

3: 0 <= ... < 4

5: 4 <= ... < 5

Monotonic Queue

- A queue whose elements are monotonically increasing or decreasing.
- For a mono-decreasing Queue:
    - To push an element e, starts from the rear element, we pop out elements less than e.

Pair Programming

Problem

Time and Space Complexity

- The time and space complexity for monotonic stack and queue operations are the same as stack and queue operations.
    - **Why**?

Pitfalls & Opportunities

- Be careful of how to handle equality
    - Should we pop elements in the monotonic stack/queue that are equal?
- Check if stack/queue is empty before accessing/removing
- For greater problems, usually use a monotonically increasing stack
- For smaller problems, usually use a monotonically decreasing stack
- For problems with a circular list, iterate through the list twice.

Practice Questions

Stacks

- Valid Parentheses
- Simplify Path
- Evaluate Reverse Polish Notation
- Score of parenthesis
- Backspace String Compare

Queues

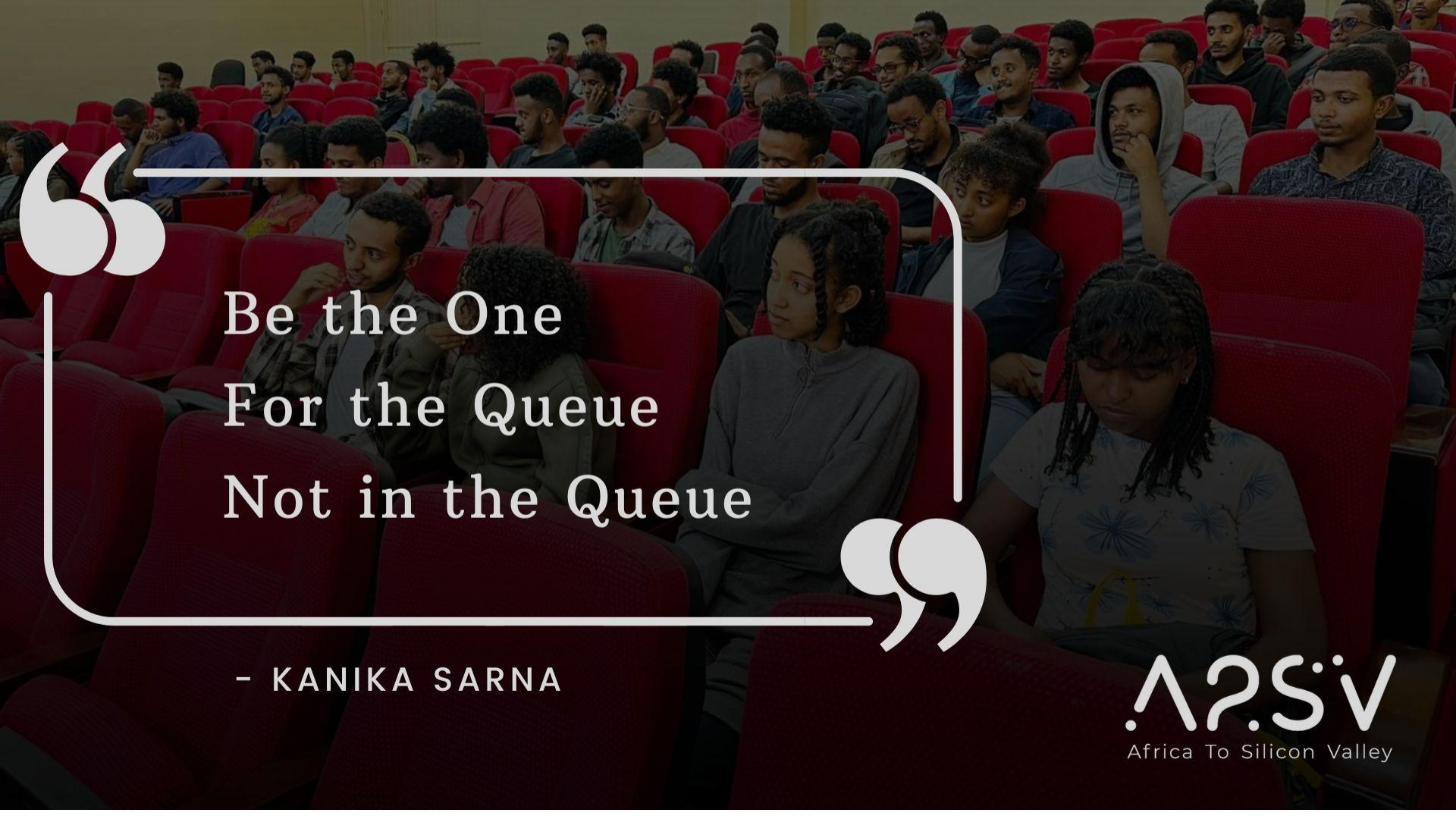
- Number of recent calls
- Find consecutive integers
- Design Circular Deque
- Implement Queue using Stack
- Shortest subarray with sum at least K

Monotonic

- Car Fleet
- Remove duplicates
- Sum of subarray minimum
- Remove k digits
- 132 Pattern

Resources

A comprehensive guide and template for monotonic stack based problems


Be the One
For the Queue
Not in the Queue
– KANIKA SARNA
A2SV
Africa To Silicon Valley