

# LeetCode Design Questions

1. [LRU Cache](#)

Utilize a Doubly Linked List along with a HashMap of integers to nodes. Add or move nodes to the head as needed, and evict the least recently used node from the tail.

2. [MRU Cache](#)

Use Doubly Linked List, Map of Integer and Node, Add to head, evict from head (most recently used)

3. [LFU Cache](#)

Use Doubly Linked List With Freq Key, Map of Integer and Node, Map of Frequency And Node, Maintain minFreq, and update frequency-linked lists accordingly, if two nodes have same frequency remove LRU node (tail.prev or the last node)

4. [Min Stack](#)

Use two stacks, stack to track normal push and pop operations and minStack to push min elements to the stack at the end and pop them accordingly

5. [Max Stack](#)

Use TreeMap of Integer and Node, and doubly linked list, add to the list and remove accordingly

6. [Maximum Frequency Stack](#)

Use two stacks one each for count of each val and another for countToStack for obtaining list from max frequency

7. [Implement Stack using Queues](#)

Use 2 queues, push elements of second queue to first queue and swap both queues

8. [Implement Queue using Stacks](#)

Push to the stack1, reverse elements and push to the stack2

9. [Dinner Plate Stacks](#)

Use List of Stack<Integer> and TreeSet to get the first available stack and update elements

10. [How would you design a class that behaves similar to a Java Map <Date, Object> except that get\(\) returns the mapped value for not only an exact match but also for the latest Date in the Map](#)

Use TreeMap and floorKey

11. [Design a Stack With Increment Operation](#)

- Use two ArrayLists stack and inc
- Update inc[index] as inc[index] + val upon element addition
- And capacity variable
- Based on the index array update the stack elements

12. [Design HashSet](#)

- Use Node[] buckets for storing the elements
- Use Node of key and next
- Use SIZE = 1009 for prime distribution
- Using hash= key % SIZE, find the appropriate bucket and perform operations
- If key already exists in the add operation return

13. [Design HashMap](#)

- Use Node[] buckets for storing the elements
- Use Node of key, value and next
- Use SIZE = 1009 for prime distribution
- Using hash= key % SIZE, find the appropriate bucket and perform operations

#### 14. [Design Linked List](#)

- Use ListNode of val and next
- Using ListNode head and size (capacity) perform operations

#### 15. [Two Sum III - Data structure design](#)

- Use Map<Integer, Integer>
- Find the compliment and return true if the pairs are found
- If the compliment is same as num, there should be multiple nums in the map

#### 16. [Map Sum Pairs](#)

- Use TrieNode with fields TrieNode[] children and sum
- Update the sum of the child node using delta of current value and existing sum upon insertion
- Return the sum of the child node

#### 17. [All O`one Data Structure](#)

- Use Doubly Linked List with Bucket node, prev, next, count and set of strings
- Map<String, Integer> keyCountMap to track count of each string
- Map<Integer, Bucket> to get all the nodes with a given count
- Min key will be at the head.next node
- Max key will be all the tail.prev node
- Using count keep on updating the nodes

#### 18. [Time Based Key-Value Store](#)

```
Use private Map<String, TreeMap<Integer, String>> map;
```

#### 19. [Design Circular Queue](#)

- Use two pointers front and rear
- For enqueue operation increment rear as  $(\text{rear} + 1) \% \text{capacity}$ , place val at  $\text{data}[\text{rear}]$  and increment size. Check if the queue is fully occupied with max capacity
- For queue operation, increment front as  $(\text{front} + 1) \% \text{capacity}$  and decrement size and check if the queue is empty or not.

- queue[front] will be the front and queue[rear] will be the rear

## 20. [Design Circular Deque](#)

- Use two pointers front and rear
- For insertLast operation increment rear as  $(\text{rear} + 1) \% \text{capacity}$ , place val at data[rear] and increment size. Check if the deque is fully occupied with max capacity. And for insertFront increment front as  $(\text{front} - 1 + \text{capacity}) \% \text{capacity}$ .
- For deleteFront operation, increment front as  $(\text{front} + 1) \% \text{capacity}$  and decrement size and check if the queue is empty or not. And for deleteLast increment rear as  $(\text{rear} - 1 + \text{capacity}) \% \text{capacity}$ .
- queue[front] will be the front and queue[(rear - 1 + capacity) % capacity] will be the rear

## 21. [Design In-Memory File System](#)

Use TrieNode with fields String fileName, boolean isFile, StringBuilder fileContent and Map<String, TrieNode> children

## 22. [Design Video Sharing Platform](#)

PriorityQueue<Integer> userIds

Map<Integer, String>

videoidToVideo

Map<Integer, String> videoidToViews

Map<Integer, String> videoidToLikes

Map<Integer, String> videoidToDislikes

## 23. [Shuffle an Array](#)

- Use Random class
- Set original as nums.clone
- Iterate from i to n
- Set j as rand.nextInt(i + 1)
- Swap i and j
- Return shuffled array

## 24. [Snake And Ladders](#)

- Flatten the board into a 1D array by traversing from the top row to the bottom, alternating the direction (left-to-right or right-to-left) after each row.
- Initialize BFS by marking the first cell as visited and adding it to the queue.
- While traversing, if the current index equals  $n * n - 1$ , return the current number of moves.
- For each position, iterate over possible dice rolls from 1 to 6. For each roll:
- Calculate  $\text{next} = \text{current} + \text{dice}$ .
- Determine the destination using:  $\text{destination} = \text{flattenedBoard}[\text{next}] == -1 ? \text{next} : \text{flattenedBoard}[\text{next}] - 1$ .
- If the destination hasn't been visited, mark it as visited, add it to the queue, and increment the move count.

## 25. [Design Snake Game](#)

- Use `Queue<int[]> foodQueue` to track the cells where food is placed
- `Deque<int[]> snake` to track the snake moments
- `Set<String> bodySet` to track the visited cells

## 26. [Design Tic-Tac-Toe](#)

- Instead of storing the board, we maintain counters:
  - `rows[i]` = sum of moves in row  $i$
  - `cols[j]` = sum of moves in column  $j$
  - `diag` / `antiDiag` = sum on diagonals
- Player 1 adds `+1`, Player 2 adds `-1`
- If any counter hits `n` or `-n`, that player wins

## 27. [Binary Search Tree Iterator](#)

- Use Binary Tree InOrder Traversal and index will be the current index
- `next` will be `inorder.get(index++)`
- `hasNext` will be `index < inorder.size()`

## 28. [Implement Trie \(Prefix Tree\)](#)

- Use `TrieNode` of fields `TrieNode[] children` and boolean `isEndOfWord`;
- Traverse through the children and update the fields

## 29. [Design Add and Search Words Data Structure](#)

- Use a TrieNode class with fields: TrieNode[] children and a boolean isEndOfWord.
- During search, if the current character is a wildcard (e.g., '!'), explore all possible child nodes. Otherwise, proceed with the corresponding child node if it exists.
- Continue until the end of the word is reached and ensure isEndOfWord is true

## 30. [Design Twitter](#)

- Map<Integer, Set<Integer>> for user follow relationships.
- Map<Integer, List<Tweet>> to store each user's tweets.
- A global timestamp to maintain tweet ordering.
- A max-heap (PriorityQueue) to fetch the most recent tweets from a user and their followees.
- Use Tweet class with fields time and id. And arrange Tweets via their time fields PriorityQueue<Tweet> maxHeap = new PriorityQueue<>((a, b) → b.time - a.time);

## 31. [Logger Rate Limiter](#)

- Use Map<String, Integer> limiter
- If time is greater than currentTimeStamp return false else update String timeStamp and return true

## 32. [Design Hit Counter](#)

Why is ArrayDeque better than LinkedList?

If you need add/remove of the both ends, ArrayDeque is significantly better than a linked list

- Use Deque of [timestamp, count] and getHits will be O(k)
- For more optimal approach use private final int[] times and private final int[] hits where complexity will be O(1)

## 33. [Finding MK Average](#)

### • Three Sorted Buckets:

- `smallestK` : holds the smallest  $k$  elements.
- `largestK` : holds the largest  $k$  elements.
- `middle` : holds the rest (used to compute the average).
- $MKAverage = \frac{\text{Sum of middle elements}}{m-2k}$

- **Queue ( stream ):**
  - Maintains the sliding window of size  $m$ .
- **TreeMaps:**
  - Used for efficient access to smallest/largest keys and handling duplicates via value counts.

#### 34. [Design Browser History](#)

- **backStack:** keeps track of previously visited pages.
- **forwardStack:** keeps track of pages you can go forward to.
- **currentPage:** holds the page you're currently on.

#### Operations:

- **visit(url):** push current page to backStack, set new URL, clear forwardStack.
- **back(steps):** move steps pages back by popping from backStack and pushing into forwardStack.
- **forward(steps):** move steps pages forward by doing the reverse.

A circular queue is excellent for:

- Fixed-size storage (like caching or rate-limiting).
- Overwriting oldest entries when full.

But for browser history, the behavior is:

- Unbounded growth (the user can keep visiting new pages).
- Dynamic resizing and clearing forward history on visit().

#### 35. [Design A Leaderboard](#)

- Use Map<Integer, Integer> to track the playerScores
- Use TreeMap<Integer, Integer> to store the scoreFrequencies

#### 36. [Design File System](#)

- Use Trie of int value and Map<String, Trie> children
- Place the children value and retrieve it

### 37. [Design a File Sharing System](#)

- `private int totalChunks;`

► Stores the total number of unique chunks available in the file sharing system (from 1 to totalChunks).
- `private int lastAssignedUserId;`

► Tracks the latest user ID assigned. It's incremented each time a new user joins (unless a reusable ID is available).
- `private TreeSet<Integer> availableUserIds;`
- ► A min-heap (via TreeSet) that keeps track of user IDs from users who have left. These IDs can be reassigned when a new user joins.
- `private TreeMap<Integer, Set<Integer>> userToChunksMap;`
- ► Maps each userId to a Set of chunk IDs that the user currently owns. It tracks chunk ownership per user.

### 38. [Kth Largest Element in a Stream](#)

- Use `PriorityQueue<Integer> minHeap`
- Add elements to `minHeap` at the constructor
- Keep only  $k$  largest elements in the heap
- Remove smallest if size exceeds  $k$  using `poll` operation
- The  $k$ th largest is always at the top

### 39. [Find Median from Data Stream](#)

- Max-heap (`lowerHalf`) for the smaller half of the numbers.
- Min-heap (`upperHalf`) for the larger half of the numbers.
- Ensure:
  - `lowerHalf.size() == upperHalf.size()` → median is average of two heap tops.
  - `lowerHalf.size() == upperHalf.size() + 1` → median is top of `lowerHalf`.
- `(lowerHalf.peek() + upperHalf.peek()) / 2.0` will be the answer

### 40. [Product of the Last K Numbers](#)

- Maintain a list of prefix products: `prefixProducts[i] = product of all numbers up to i`.
- Reset prefix when a zero is added.
- To get the product of last  $k$  numbers:
  - If there was a  $0$  in the last  $k$  numbers → return  $0$ .
  - Otherwise, compute as `prefix[n] / prefix[n - k]`.

#### 41. [Kth Ancestor of a Tree Node](#) (Binary Lift)

## Binary Lifting

- Precompute a table where `dp[node][i]` is the  $2^i$ th ancestor of `node`.
- To get the  $k$ -th ancestor:
  - Decompose  $k$  in binary.
  - Jump via powers of 2 using precomputed `dp`.

Or else obtain the ancestors of each node using level order traversal and find the  $k$ th ancestor

#### 42. [Flatten 2D Vector](#)

Move to the next row while current row is exhausted or empty

- Use Matrix traversal and
 

```
private int rowIndex;
private int colIndex;
private int[][] matrix;
```

#### 43. [Encode and Decode Strings](#)

- `encode (List<String> stringList)`
  - Converts each string into a format: [length as a char][original string]
  - Appends them one after another into a single string using `StringBuilder`
- `decode (String encodedString)`
  - Reads the first character (which represents the length of the upcoming word) using `List<String>`
  - Extracts that substring of that length.
  - Repeats until the full string is parsed.

#### 44. Peeking Iterator

- Use `Iterator<Integer> iterator` and `Integer nextElement;`
- If `iterator.hasNext()` set `nextElement` as `iterator.next()`
- For `hasNext`, `nextElement` shouldn't be `null`

#### 45. Design Excel Sum Formula

- `cells`: A 2D array storing the values of each cell.
- `cellSumMap`: A map that tracks which cells affect other cells' sum calculations.
- `sumCellMap`: A map that tracks which sum cells depend on others, aiding efficient recalculation when updating values.
- Use BFS

#### 46. Design Log Storage System

- `private List<LogEntry> logs = new ArrayList<>();`
- 👉 Stores all log entries (each with an id and timestamp).
- `private Map<String, Integer> granularityLengthMap = new HashMap<>();`
- 👉 Maps each time granularity (like "Year" or "Minute") to the length of the timestamp prefix used for comparison.
- Iterate and add ids to the result
- `granularityLengthMap.put("Year", 4);`
- `granularityLengthMap.put("Month", 7);`
- `granularityLengthMap.put("Day", 10);`
- `granularityLengthMap.put("Hour", 13);`
- `granularityLengthMap.put("Minute", 16);`
- `granularityLengthMap.put("Second", 19);`

#### 47. Design Search Autocomplete System

- Use `TrieNode` with fields
- `TrieNode[] children = new TrieNode[27]; // 26 letters + space`
- `int frequency;`

- String sentence = "";
- Update the frequency and sentence of the child node
- Use the minheap to collect top suggestions
- PriorityQueue<TrieNode> topSuggestions = new PriorityQueue<>((a, b) → a.frequency == b.frequency ? b(sentence).compareTo(a.sentence) : a.frequency - b.frequency);  
// Lexicographically higher comes first  
// Lower frequency comes first
- );

#### 48. [Implement Magic Dictionary](#)

- Use TrieNode with fields
  - TrieNode[] children = new TrieNode[26];
  - boolean isEnd = false;
- Iterate from i = 1 to 26
- If i == index and dfs(word, pos + 1, node.children[i], modified) return true
- Else if (!modified && dfs(word, pos + 1, node.children[i], true)) return true;
- Try changing this char (only if we haven't already)

#### 49. [Design Most Recently Used Queue](#)

- totalSize: Stores the size of the queue.
- queueArray: The underlying array simulating the queue.
- fetchedElement: The element at position k (1-indexed) to be moved to the end.
- Shift elements left from position k
- Move the fetched element to the end

#### 50. [Tweet Counts Per Frequency](#)

- private Map<String, TreeMap<Integer, Integer>> tweetTimeMap;
- tweetTimeMap is a mapping from tweet names to their timestamps.
- For each tweet name (like "elon" or "news"), we use a TreeMap<Integer, Integer> to store the number of times that tweet was recorded at specific times.

## 51. [Design an ATM Machine](#)

- Use private final int[] denominations = {20, 50, 100, 200, 500};
- private long[] noteCounts = new long[5];
- // stores count of each note type

## 52. [Encode and Decode TinyURL](#)

- shortToLongMap: Stores mapping from short key to original URL.
- longToShortMap: Avoids duplicating short links for the same original URL.
- counter: Generates unique short keys (can be made fancier with base62 for even shorter links).

## 53. [Design a Food Rating System](#)

- Map<String, String> foodToCuisine = new HashMap<>();
- Map<String, Integer> foodToRating = new HashMap<>();
- Map<String, TreeSet<Food>> cuisineToFoods = new HashMap<>();

## 54. [Design Task Manager](#)

- // taskId → [userId, priority]
- private final Map<Integer, int[]> taskDetailsMap = new HashMap<>();
- Quick access to task details by taskId.
- Stores a mapping from taskId to a pair of [userId, priority].

```
private final TreeSet<int[]> taskSet = new TreeSet<>((a, b) → {  
    if (a[0] == b[0]) {  
        return b[1] - a[1]; // higher taskId first if same priority  
    }  
    return b[0] - a[0]; // higher priority first  
});
```

- Maintains tasks sorted by priority (high → low).
- If priorities match, it breaks ties by taskId (high → low).
- Enables efficient retrieval of the highest priority task in O(log n) time.

## 55. [Design Spreadsheet](#)

- `private final Map<String, Integer> cellValueMap = new HashMap<>();`
- is used to store and track the current value of each non-empty cell in the spreadsheet.
- String → Represents the cell name, like "A1", "B2", "C5", etc.
- Integer → The value assigned to that cell by the user.

## 56. [Implement Router](#)

- `private final int memoryLimit;`
- // Stores the maximum number of packets the router can hold at any time.
- `private final Set<String> packetSet = new HashSet<>();`
- // Tracks unique packets using a combination of source, destination, and timestamp to prevent duplicates.
- `private final Deque<Packet> packetQueue = new ArrayDeque<>();`
- // Maintains the order of packets for FIFO (First-In-First-Out) processing and eviction when memory limit is reached.
- `private final Map<Integer, List<Integer>> destinationTimestamps = new HashMap<>();`
- // Maps each destination to a sorted list of timestamps for efficient range-based queries in getCount().

## 57. [Serialize and Deserialize Binary Tree](#)

- `serialize(TreeNode root);`
- Converts the binary tree to a comma-separated string using preorder traversal.
- "null" is used to indicate a missing child.
- `deserialize(String data);`
- Reconstructs the binary tree from the serialized string by consuming the nodes in order using a queue.
- `serialize:` Performs level-order traversal (BFS) and appends "null" for missing children.
- `deserialize:` Reconstructs the tree using a queue, assigning left and right children level by level.

## 58. [Range Sum Query - Immutable](#)

- Prefix Sum

## 59. [Range Sum Query 2D - Immutable](#)

- Prefix Sum
- ```
prefix[i + 1][j + 1] = matrix[i][j] + prefix[i][j + 1] + prefix[i + 1][j] - prefix[i][j];
```
- ```
public int sumRegion(int row1, int col1, int row2, int col2) { return prefix[row2 + 1][col2 + 1] - prefix[row1][col2 + 1] - prefix[row2 + 1][col1] + prefix[row1][col1]; }
```

### 30. [Range Sum Query - Mutable](#)

- Fenwick Tree (Binary Indexed Tree)
- Segment Tree

### 31. [Range Sum Query 2D - Mutable](#)

- Fenwick Tree (Binary Indexed Tree)
- Segment Tree

### 32. [Flatten Nested List Iterator](#)

- ```
Deque<NestedInteger> stack = new ArrayDeque<>();
```
- ```
next() → stack.pop().getInteger();
```
- ```
hasNext() → Unwrap the nested list inside while loop using stack.push(nestedList.get(i)) and return true if stack top is a integer
```

### 33. [Design Compressed String Iterator](#)

- ```
private final List<CharFrequency> charFrequencyList = new ArrayList<>(); stores pairs of characters and their respective frequencies as CharFrequency objects.
```
- ```
private int currentIndex = 0; keeps track of the current position in the charFrequencyList while traversing through the characters.
```

### 34. [Zigzag Iterator](#)

- ```
private List<Integer> indexes = new ArrayList<>();
```
- Stores the **current read index** for each input list (`v1`, `v2`, etc.).
- ```
private List<List<Integer>> vectors = new ArrayList<>();
```
- Stores **all input lists** (`v1`, `v2`, etc.) that need to be iterated in zigzag fashion.

### 35. [RLE Iterator](#)

- Use two pointers `i` and `j` and `int[] encoding` array
- ```
if (encoding[i] - j < n) → not enough left in current block
```

- Skip current block, update n, move to next pair.
- Else: Consume n from current block.
- Return the corresponding value encoding[i + 1].

### 36. [Insert Delete GetRandom O\(1\)](#)

- List<Integer> nums stores the elements of the set in a list to allow O(1) access for getRandom() and swapping during removal.
- Map<Integer, Integer> pos maps each number to its index in nums and used to locate and remove elements in O(1) time.
- Random rand generates random indices for the getRandom() method to return a random element from nums.

### 37. [Insert Delete GetRandom O\(1\) - Duplicates allowed](#)

- private List<Integer> nums;
- Stores all inserted elements (including duplicates).
- private Map<Integer, Set<Integer>> valToIndices;
- Maps each value to the set of indices where it appears in nums.
- private Random rand;
- Used for random index generation in getRandom.

### 38. [Online Stock Span](#)

- Utilize Stack<int[]> stack;
- Add stack.pop()[1]; to the span till the stack is not empty and stack.peek()[0] <= price

### 39. [Design SQL](#)

Use Map<String, List<List<String>>> tables = new HashMap<>(names.size()); to store the mapping of table names to table data rows. Directly simulate the operations in the problem.

### 70. [Design a Text Editor](#)

- private StringBuilder leftText Holds text left of the cursor (grows as you add text or move cursor right).
- private StringBuilder rightText; Holds text right of the cursor (grows as you move cursor left).

### 71. [Design Memory Allocator](#)

- `private final int[] memoryBlocks` : Simulates the memory blocks, where `0` means free and any `>0` value means allocated to that `memoryId` .
- `consecutiveFree` : Tracks how many contiguous free blocks are currently counted.
- `startIndex` : Marks the start position where memory will be allocated.
- `freedCount` : Counts how many blocks were released during `freeMemory` .

72. [Encode N-ary Tree to Binary Tree](#)

### `encode(Node root)` — N-ary → Binary

- Converts an N-ary tree into a binary tree.
- The first child of each N-ary node becomes the **left child** in binary.
- All **subsequent siblings** are linked using the **right pointer**.

### `decode(TreeNode data)` — Binary → N-ary

- Converts the binary tree back into its original N-ary form.
- Follows the left child for the first child, then repeatedly goes right for siblings.

73. [Moving Average from Data Stream](#)

- `private final int[] window`: Fixed-size array to store the last N values (the window).
- `private int windowSum`: Maintains the sum of current values in the window for average computation.
- `private int count`: Tracks how many total values have been seen.
- `indexToReplace = count % window.length`: Ensures circular replacement inside the window.

74. [Serialize and Deserialize BST](#)

- `Serialize` : Uses preorder traversal (root, left, right) and stores values in a string.
- `Deserialize` : Uses preorder values and BST property ( `min` and `max` bounds) to reconstruct the tree without needing to split the array or find midpoints.

75. [Unique Word Abbreviation](#)

`private Map<String, Set<String>> map = new HashMap<>();` stores a mapping from an **abbreviated word** to the **set of original words** in the dictionary that share this abbreviation.

76. [Data Stream as Disjoint Intervals](#)

We'll maintain a `TreeMap<Integer, int[]>` called `intervals`, where:

- **Key** = interval start
- **Value** = `[start, end]`

## 77. [Design Phone Directory](#)

`int maxNumbers;`

Stores the maximum number of phone numbers allowed in the directory.

`boolean[] available;`

Purpose: Tracks the availability status of each number.

`available[i] = true` means number  $i$  is free to assign.

`available[i] = false` means number  $i$  is already taken.

`Queue<Integer> unused;`

Purpose: Stores a queue of currently unused (free) numbers.

## 78. [Shortest Word Distance II](#)

- `wordIndicesMap`: Stores each word and the list of its positions in the `wordsDict`.
- `indices1, indices2`: Position lists of the two words we are comparing.
- `pointer1, pointer2`: Used to iterate through both lists using two-pointer technique.
- `minDistance`: Keeps track of the minimum distance found so far.

## 79. [Range Module](#)

[Use Segment Trees](#)

## 30. [My Calendar I](#)

- We store all booked intervals in a list.
- When a new booking is requested, iterate through existing bookings and check for conflicts.
- If no conflicts, add the interval to the list.
- Use a list of existing bookings and check each for overlap with the new one. For each interval  $[s, e]$ :
- Conflict exists if:
  - $\text{newStart} < \text{existingEnd} \&\& \text{existingStart} < \text{newEnd}$

## 31. [My Calendar II](#)

Track two separate things:

- booked: list of all single bookings.
- overlaps: list of intervals that are already double-booked (i.e., overlaps of existing bookings).

When trying to book a new interval:

- Check if it overlaps with any interval in overlaps → If yes, reject (triple booking).
- For every interval in booked, if there's an overlap with the new one, add the overlapping portion to overlaps.
- Add the new interval to booked.

## 32. [My Calendar III](#)

Segment Tree with lazy updates

## 33. [Prefix and Suffix Search](#)

- Use TrieNode with fields `TrieNode[] children = new TrieNode[27]; // 'a' to 'z' and '{'`
- and `index = -1`
- Insertion:
- For each word like "apple", insert all rotations of suffix + "{" + prefix:
- E.g. "e{apple", "le{apple", ..., "apple{apple"
- Trie traversal:
- Store characters 'a' to 'z' and '{' (which separates suffix and prefix).
- Always update node's index with the latest word index.

## 34. [Exam Room](#)

- `private int n;`
- `private Node head = new Node(-1);`
- `private Node tail = new Node(-1);`
- `private Map<Integer, Node> map = new HashMap<>();`

`// {p: student iterator}`

## 35. [Online Election](#)

- `private int[] times; Stores the time at which each vote occurred.`

- `private int[] leaders`: Stores the leader at each index  $i$  in time.

```
Map<Integer, Integer> voteCount = new HashMap<>();
```

- Purpose: Keeps track of how many votes each person has gotten so far.
- Use:
  - After every vote, update `voteCount[person]++`.
  - If that person's count is  $\geq$  current `maxVotes`, update the current leader.
- Why needed: Helps determine who the leader is after every vote.

### 36. Complete Binary Tree Inserter

- `private TreeNode root;`
- `private Queue<TreeNode> candidateQueue;`
- Purpose: Holds incomplete nodes (nodes with  $<2$  children) in level order.
- Why:
  - To know exactly where to insert the next node in  $O(1)$  time.
  - Maintain the complete binary tree structure (left to right filling).
- How it works:
- Initially filled using BFS.
- On `insert(val)`:
  - Always insert into the front of the queue.
  - If that node becomes full, remove it.
  - Add the new node to the back of the queue.

### 37. Number of Recent Calls

- `private Queue<Integer> recentPings;`
- `recentPings`: A queue to store timestamps of the pings.
- Maintains only valid pings within the time window  $[t - 3000, t]$ .
- Ensures efficient time-based sliding window using FIFO behavior of a queue.

### 38. Stream of Characters

- Use TrieNode with fields
- TrieNode[] children = new TrieNode[26];
- boolean isWord = false;

### 39. [Snapshot Array](#)

- Use a TreeMap<Integer, Integer> per index to store historical values:
- The key is snap\_id, and the value is the value set at that index during that snapshot.
- On get(), use .floorEntry(snap\_id) to find the closest snapshot  $\leq$  given snap\_id.

### 40. [Online Majority Element In Subarray](#)

- Preprocess positions of each number: Map<Integer, List<Integer>> positions.
- Use random sampling or segment tree with Boyer-Moore logic to find a candidate.
- Use binary search on the positions list to count how often a number appears in [left, right].

### 41. [Design SkipList](#)

- Use Deque<Node> nodes = new ArrayDeque<>();
- And
- class Node {
- public int val;
- public Node next;
- public Node down;
- public Node(int val, Node next, Node down) {
- this.val = val;
- this.next = next;
- this.down = down;
- }}

### 42. [Find Elements in a Contaminated Binary Tree](#)

- Use Set<Integer> values and preorder traversal to store recovered value
- And check if values contains the target

### 33. Iterator for Combination

Precompute all combinations using backtracking (dfs)

### 34. Apply Discount Every n Orders

- `Map<Integer, Integer> priceMap`: Stores product IDs mapped to their prices.
- `int customerCount`: Keeps track of how many customers have been billed so far.
- Every nth customer gets a discount.

### 35. Design Underground System

- `// Stores ongoing trips: id → (stationName, time)`
- `private Map<Integer, Pair<String, Integer>> checkIns;`
- `// Stores total travel time and trip counts for each route: "start#end" → (totalTime, tripCount)`
- `private Map<String, Pair<Integer, Integer>> travelTimes;`

### 36. First Unique Number

- `private Map<Integer, Integer> cnt = new HashMap<>();`
- To keep track of how many times each number appears in the stream.
- `private Set<Integer> unique = new LinkedHashSet<>();`
- To maintain the insertion order of numbers that are currently unique.

### 37. Subrectangle Queries

- Approach 1: Brute Force (Modify Matrix Directly)
- Approach 2: Optimized (Store Updates Separately)
- `private int[][] rectangle;`
- `private List<int[]> updates;`
- Instead of modifying the matrix, store a list of updates and apply them lazily during `getValue` using a for loop

### 38. Dot Product of Two Sparse Vectors

- `private Map<Integer, Integer> indexValueMap = new HashMap<>(128);`
- This map stores only the non-zero elements of the sparse vector.

- Key: the index where a non-zero value appears.
- Value: the actual non-zero value at that index.
- `Map<Integer, Integer> smallerMap = this.indexValueMap;`
- `Map<Integer, Integer> largerMap = otherVector.indexValueMap;`
- We always iterate on the smaller map (i.e., fewer non-zero elements) to reduce the number of iterations.

## 39. Binary Search Tree Iterator II

- Use
- `private List<Integer> nums = new ArrayList<>();`
- `private int i = -1; // prev`
- And inorder traversal

## 40. Throne Inheritance

- `private Set<String> dead = new HashSet<>();` is used to track the names of individuals who are deceased and, therefore, should not be included in the inheritance order.
- `private Map<String, List<String>> family = new HashMap<>();` holds the family tree:
  - Key: the name of a parent.
  - Value: a List of names of children (who are direct descendants of that parent).
- `private String kingName` is the name of the king at the top of the hierarchy.

## 41. Design Parking System

- We need three integer variables to track the available spots for each type of parking space:
- `big`: For big spots.
- `medium`: For medium spots.
- `small`: For small spots.

## 42. Design an Expression Tree With Evaluate Function

- `abstract class Node { public abstract int evaluate();`
- `protected String val;`
- `protected Node left;`
- `protected Node right;`

```
    };
```

- Use Stack

### Q3. Design an Ordered Stream

- `private String[] stream;`
- `private int ptr = 0;`
- `stream: an array to store the values at their id - 1 position.`
- `ptr: a pointer tracking the next expected index to output.`
- `Use List<String> result = new ArrayList<>();`

### Q4. Design Front Middle Back Queue

- Use two deques:
  - `Deque<Integer> left = new ArrayDeque<>();`
  - `Deque<Integer> right = new ArrayDeque<>();`
- And rebalance

### Q5. Design Authentication Manager

```
private int timeToLive;
```

- Stores the global lifetime duration of each token (in seconds).
- When a token is generated or renewed, this value is added to the currentTime to calculate the new expiration time.
- `tokenExpiryMap.put(tokenId, currentTime + timeToLive);`

```
private Map<String, Integer> tokenExpiryMap;
```

- Maps each tokenId to its expiration time.
- `tokenId` (a unique identifier for a token)
- The absolute expiration time (`currentTime + timeToLive`) at which the token becomes invalid.

### Q6. Implement Trie II (Prefix Tree)

```
private Trie[] children;  
// array representing the children nodes of the current node  
private int wordCount;  
// number of times a complete word terminates at this node
```

```
private int prefixCount;  
// number of words sharing the prefix that ends at this node
```

## 07. [Seat Reservation Manager](#)

- availableSeats: A min-heap of integers that stores all currently unreserved seat numbers.
- Constructor: Initializes all seats from 1 to n into the min-heap.
- reserve(): Retrieves and removes the smallest-numbered seat.
- unreserve(int seatNumber): Re-adds a previously reserved seat back into the min-heap.

## 08. [Finding Pairs With a Certain Sum](#)

- Map<Integer, Integer> freqMap2: Keeps track of frequencies of numbers in nums2, which allows for fast lookup in count().
- add(index, val): Updates the freqMap2 correctly when nums2[index] changes.
- count(tot): For each a in nums1, we look for tot - a in nums2 using the map.

## 09. [Design Movie Rental System](#)

- // {movie: (price, shop)} private Map<Integer, Set<Entry>> unrented = new HashMap<>();
- // {(shop, movie): price} private Map<Pair<Integer, Integer>, Integer> shopAndMovieToPrice = new HashMap<>();
- // (price, shop, movie) private Set<Entry> rented = new TreeSet<>(comparator);

## 10. [Operations on Tree](#)

- Map<Integer, Integer> lockedBy — stores which user locked which node.
- Map<Integer, List<Integer>> tree — maps each node to its children for traversal.
- upgrade() — verifies all 3 conditions:
  - Node is not locked.
  - No locked ancestor exists.
  - At least one locked descendant exists.

## 11. [Detect Squares](#)

- Map<Integer, Map<Integer, Integer>> pointsCount — counts occurrences of each point (x, y), stored by y row-wise.
- In count(point):

- For every other  $y$  ( $y_2 \neq y_1$ ), calculate potential square side length.
- Check if the required three other corners exist:  $(x_1, y_2)$ ,  $(x_2, y_1)$ ,  $(x_2, y_2)$ .
- Multiply their frequencies to count the number of such squares.

## 12. [Stock Price Fluctuation](#)

- `private Map<Integer, Integer> timestampPriceMap;`
- `private TreeMap<Integer, Integer> priceFrequencyMap;`
- `private int latestTimestamp;`
- `timestampPriceMap`: stores the most recent price for each timestamp.
- `priceFrequencyMap`: maintains frequencies of prices in sorted order using `TreeMap`, allowing efficient min/max queries.
- `latestTimestamp`: tracks the most recent timestamp for the current() operation.

## 13. [Simple Bank System](#)

- `long[] balance`: stores the balance of each account (1-indexed).
- `isValid(int account)`:
  - Checks if the account number is within valid range.
- `transfer(account1, account2, money)`:
  - Validates account2.
  - Withdraws money from account1, then deposits to account2.
  - Returns true if both succeed.
- `deposit(account, money)`:
  - Validates account.
  - Adds money to the account.
- `withdraw(account, money)`:
  - Validates account.
  - Checks if there is enough balance, then deducts.

## 14. [Range Frequency Queries](#)

- // Map from value to list of indices where it appears in the array
- private Map<Integer, List<Integer>> positionsMap;
- lowerBound & upperBound
- update is not required, hence segment trees are not optimal here due to no updates

## 15. Design Bitset

- private int size;
- private int countOnes;
- private boolean flipped;
- private int[] bits;
- Instead of physically flipping the entire bitset, we use a lazy flip technique using a flipped flag.
- flipped: A flag to simulate flips without actually changing every bit.
- countOnes: Maintains count of actual 1s to make all(), one() and count() operations efficient in O(1).
- Use XOR (^) to calculate the real bit depending on whether flipped or not.
- And StringBuilder for the toString operation

## 16. Encrypt and Decrypt Strings

```
Use class TrieNode {
    public TrieNode[] children = new TrieNode[26];
    public boolean isWord = false;
}
private Map<Character, String> keyToValue = new HashMap<>();
private Map<String, List<Character>> valueToKeys = new HashMap<>();
```

## 17. Count Integers in Interval s

- private TreeMap<Integer, Integer> intervals;
- To store non-overlapping merged intervals in sorted order by their start point.
- Key = start of interval, Value = end of interval.
- Why TreeMap:
- Allows efficient lookups of intervals using floorEntry(), ceilingEntry() — to find and merge overlapping intervals.

- Keeps keys sorted, which helps in ordered traversal and merging.

- intervals = {

```
1 → 5, // represents [1, 5]
```

```
10 → 12 // represents [10, 12]
```

```
}
```

## 18. [Smallest Number in Infinite Set](#)

- private int current;
- private PriorityQueue<Integer> minHeap;
- private Set<Integer> addedBack;
- Use a min-heap (PriorityQueue) to store numbers added back.
- Use a HashSet to track what's currently in the heap.
- Track a current counter to know what's the next number in the infinite set (if not already popped or added back)

## 19. [Design a Number Container System](#)

- // Stores the number at each index
- private Map<Integer, Integer> containerMap;
- // Maps each number to a min-heap of indices where the number appears
- private Map<Integer, PriorityQueue<Integer>> numberIndexMap;

## 20. [Longest Uploaded Prefix](#)

- boolean[] uploaded: To track which videos have been uploaded.
- int longestPrefix: To track the longest prefix ending at the highest continuous uploaded video.

## 26. [Find Consecutive Integers from a Data Stream](#)

```
private final int value;
```

Stores the specific integer value that we want to track in the stream.

```
private final int k;
```

Represents how many consecutive times the value must appear for consec() to return true.

```
private int consecutiveCount
```

Tracks how many times the target value has appeared consecutively in the stream so far.

## 28. Design a Todo List

- Use
- class Task {
- int taskId;
- String taskName;
- int dueDate;
- Set<String> tags; boolean finish;
- }
- private Map<Integer, TreeSet<Task>> tasks = new HashMap<>(); to sort tasks based on their dueDate
- new TreeSet<>(Comparator.comparingInt(a → a.dueDate)

## 29. Design Graph With Shortest Path Calculator

- Use Dijkstra's Algorithm
- Keep on marking the nodes as visited and return if they are already visited
- If destination node is reached return current distance (dist[u] + weight)

## 30. Frequency Tracker

- Map<Integer, Integer> numFreqMap → maps number to its frequency.
- Map<Integer, Integer> freqCountMap → maps frequency to how many numbers have that frequency.

## 31. Design Neighbor Sum Service

- private int[][] grid;
- Stores the original 2D matrix of integers.
- Used to access any cell's value directly during neighbor lookups in adjacentSum() or diagonalSum().
- private int rows, cols;
- These represent the number of rows and columns in the grid.
- Used for boundary checks to ensure neighbor indices don't go out of bounds during traversal.
- private Map<Integer, List<int[]>> valueToCoordinates;
- Maps each unique number in the grid to a list of coordinates (i, j) where that number appears.

- Used to quickly find all positions of a given value during calls to adjacentSum(value) or diagonalSum(value).

### 32. Design an Array Statistics Tracker

- `private final Deque<Integer> q = new ArrayDeque<>();`

Keeps track of the numbers in the order they were added.

- `private long s;`

Maintains the running sum of all numbers currently in the data stream.

- `private final Map<Integer, Integer> cnt = new HashMap<>();`

Keeps track of the frequency of each number.

- `private final MedianFinder medianFinder = new MedianFinder();`

Supports adding/removing numbers and finding the median efficiently.

- `private final TreeSet<int[]> ts = new TreeSet<>((a, b) → a[1] == b[1] ? a[0] - b[0] : b[1] - a[1]);`

Maintains (number, frequency) pairs in descending frequency order (and ascending number order in case of tie)

### 32. Design a 3D Binary Matrix with Efficient Layer Tracking

- `private final int[][][] grid;`

- `// 3D matrix to track cell states (0 or 1)`

- `private final int[] layerActiveCount;`

- `// Number of active cells in each x-layer`

- `// TreeSet to maintain [activeCellCount, layerIndex] pairs sorted by:`

- `// 1. descending activeCellCount`

- `// 2. descending layerIndex (if counts are equal)`

- `private final TreeSet<int[]> sortedLayers = new TreeSet<>((a, b) → a[0] == b[0] ? b[1] - a[1] : b[0] - a[0]);`

### 33. Booking Concert Tickets in Groups

Use Segment Trees

### 34. Sequentially Ordinal Rank Tracker

- `good Heap (min-heap of the top k players)`

```
private PriorityQueue<Map.Entry<Integer, String>> good = new PriorityQueue<>()
```

```
(a, b) → a.getKey().equals(b.getKey())
```

```

? b.getValue().compareTo(a.getValue())

// break ties: reverse lex order

: a.getKey() - b.getKey()));

// lower score first

• bad Heap (max-heap of better players than current top-k)

private PriorityQueue<Map.Entry<Integer, String>> bad = new PriorityQueue<>(
(a, b) → a.getKey().equals(b.getKey()))

? a.getValue().compareTo(b.getValue())

// break ties: normal lex order

: b.getKey() - a.getKey());

// higher score first

```

### 35. [Walking Robot Simulation II](#)

- Use DIRECTIONS array
- And turn 90 degrees counterclockwise) and 90 degrees clockwise inside a for loop

### 36. [Fancy Sequence](#)

- private List<Long> vals = new ArrayList<>();
- // Stores the sequence of values. Each value is transformed by a \* val + b.
- MOD = 1\_000\_000\_007; // Modulo for all calculations to handle large numbers and prevent overflow.
- Adjust the variables using +, -, \*, % operators and MOD

### 37. [Design TreeMap](#)

Designing a custom `TreeMap` involves understanding the functionality of a map that maintains its entries in a sorted order. A `TreeMap` in Java is typically backed by a self-balancing binary search tree (like a Red-Black Tree), but for educational purposes, we can implement one using a simpler structure.

## Core Design:

### 1. Key-Value Pairs:

- A `TreeMap` stores key-value pairs.
- The keys are sorted based on their natural order or by a comparator (if provided).

## 2. Tree Structure:

- The underlying data structure can be a **Binary Search Tree (BST)**, specifically a **Red-Black Tree** for balancing.

## 3. Operations:

- **Insertion:** Add key-value pairs while maintaining the order of keys.
- **Deletion:** Remove a key-value pair.
- **Search:** Retrieve the value associated with a key.
- **Traversal:** Iterate over the keys and values in sorted order.

For simplicity, let's design a basic version of a **TreeMap** backed by a **Binary Search Tree (BST)**.

## Simplified TreeMap Design using a Binary Search Tree (BST):

### 1. Node Structure:

- Each node contains:
  - A **key**.
  - A **value** associated with the key.
  - Left and Right children for the BST properties.

### 2. Operations:

- **put(key, value):** Insert a key-value pair.
- **get(key):** Retrieve the value for the given key.
- **remove(key):** Remove the key-value pair from the tree.
- **containsKey(key):** Check if the key exists in the map.
- **size():** Get the size of the map (number of key-value pairs).
- **inOrderTraversal():** Traverse the tree in sorted order.

---

### 1. Least Recently Used (LRU Cache)

move least recently used key to the head, remove from tail

Design a data structure that follows the constraints of a [Least Recently Used \(LRU\) cache](#).

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with positive size `capacity`.
- `int get(int key)` Return the value of the `key` if the `key` exists, otherwise return `-1`.
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, evict the least recently used key.

The functions `get` and `put` must each run in  $O(1)$  average time complexity.

Example 1:

**Input**  
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]  
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]  
**Output**  
[null, null, null, 1, null, -1, null, -1, 3, 4]  
**Explanation**  
LRUCache lRUCache = new LRUCache(2);  
lRUCache.put(1, 1); // cache is {1=1}  
lRUCache.put(2, 2); // cache is {1=1, 2=2}  
lRUCache.get(1); // return 1  
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}  
lRUCache.get(2); // returns -1 (not found)  
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}  
lRUCache.get(1); // return -1 (not found)  
lRUCache.get(3); // return 3  
lRUCache.get(4); // return 4

```
class LRUCache {  
    class Node {  
        int key;  
        int value;  
        Node prev;  
        Node next;  
  
        public Node(int key, int value) {  
            this.key = key;  
            this.value = value;  
        }  
    }  
}
```

```

private int capacity;
private Map<Integer, Node> map;
private Node head;
private Node tail;

public LRUCache(int capacity) {
    this.capacity = capacity;
    map = new HashMap<>();
    head = new Node(-1, -1);
    tail = new Node(-1, -1);
    head.next = tail;
    tail.prev = head;
}

public int get(int key) {
    if (!map.containsKey(key)) {
        return -1;
    }

    Node node = map.get(key);
    moveToHead(node);

    return node.value;
}

public void put(int key, int value) {
    if (map.containsKey(key)) {
        Node node = map.get(key);
        node.value = value;
        moveToHead(node);
    } else {
        Node newNode = new Node(key, value);
        map.put(key, newNode);
        addToHead(newNode);

        if (map.size() > capacity) {

```

```

        Node removed = removeTail();
        map.remove(removed.key);
    }
}

private void moveToHead(Node node) {
    removeNode(node);
    addToHead(node);
}

private void removeNode(Node node) {
    node.prev.next = node.next;
    node.next.prev = node.prev;
}

private void addToHead(Node node) {
    node.next = head.next;
    node.next.prev = node;
    head.next = node;
    node.prev = head;
}

private Node removeTail() {
    Node nodeToRemove = tail.prev;
    removeNode(nodeToRemove);
    return nodeToRemove;
}
}

public class Main {
    public static void main(String[] args) {
        LRUCache cache = new LRUCache(2); // Capacity is 2
        cache.put(1, 1);
        cache.put(2, 2);
        System.out.println(cache.get(1)); // Output: 1
    }
}

```

```

cache.put(3, 3);
System.out.println(cache.get(2)); // Output: -1 (not found)
cache.put(4, 4);
System.out.println(cache.get(1)); // Output: -1 (not found)
System.out.println(cache.get(3)); // Output: 3
System.out.println(cache.get(4)); // Output: 4
}
}

```

## ✖ Limitations of Using a Singly Linked List for LRUCache:

### ● 1. O(n) Removal of Arbitrary Node

- In LRUCache, we must move a node to the head when it's accessed or updated.
- In a singly linked list, removing a node requires traversal from the head to find the previous node, because you can't go backwards.

java

Copy    Edit

```
// Singly linked list: to delete a node, you need its previous node
// There's no direct way to get it in O(1)
```

➡ That makes removal O(n), violating the LRUCache's requirement of O(1) for all operations.

### ● 2. Cannot Remove Last Node Efficiently

- To evict the least recently used item, we remove the tail node.
- In a singly linked list, you can't go to the last node's previous without traversing from the head.

➡ Again, this is O(n).

### ● 3. No Easy Way to Reorder Nodes

- Moving nodes to the front (when accessed) is difficult in singly linked lists without rebuilding links manually, which again costs time.

---

## 2. Least Frequently Used (LFU Cache)

Move least frequently used key to the head

Design and implement a data structure for a [Least Frequently Used \(LFU\) cache](#).

Implement the `LFUCache` class:

- `LFUCache(int capacity)` Initializes the object with the `capacity` of the data structure.
- `int get(int key)` Gets the value of the `key` if the `key` exists in the cache. Otherwise, returns `-1`.
- `void put(int key, int value)` Update the value of the `key` if present, or inserts the `key` if not already present. When the cache reaches its `capacity`, it should invalidate and remove the least frequently used key before inserting a new item. For this problem, when there is a tie (i.e., two or more keys with the same frequency), the least recently used `key` would be invalidated.

To determine the least frequently used key, a use counter is maintained for each key in the cache. The key with the smallest use counter is the least frequently used key.

When a key is first inserted into the cache, its use counter is set to 1 (due to the `put` operation). The use counter for a key in the cache is incremented either a `get` or `put` operation is called on it.

The functions `get` and `put` must each run in  $O(1)$  average time complexity.

**Example 1:**

```
Input
["LFUCache", "put", "put", "get", "put", "get", "put", "get", "put", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]
Explanation
// cnt(x) = the use counter for key x
// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)
LFUCache lfu = new LFUCache(2);
lfu.put(1, 1); // cache=[1,_], cnt(1)=1
lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1
lfu.get(1); // return 1
// cache=[1,2], cnt(2)=1, cnt(1)=2
lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.
// cache=[3,1], cnt(3)=1, cnt(1)=2
lfu.get(2); // return -1 (not found)
lfu.get(3); // return 3
// cache=[3,1], cnt(3)=2, cnt(1)=2
lfu.put(4, 4); // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.
// cache=[4,3], cnt(4)=1, cnt(3)=2
lfu.get(1); // return -1 (not found)
lfu.get(3); // return 3
// cache=[3,4], cnt(4)=1, cnt(3)=3
lfu.get(4); // return 4
// cache=[4,3], cnt(4)=2, cnt(3)=3
```

```
import java.util.HashMap;
import java.util.Map;

class LFUCache {
```

```

class Node {
    int key;
    int value;
    int frequency;
    Node previous;
    Node next;

    Node(int key, int value) {
        this.key = key;
        this.value = value;
        this.frequency = 1;
    }
}

class DoublyLinkedList {
    Node head;
    Node tail;

    DoublyLinkedList() {
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head.next = tail;
        tail.previous = head;
    }
}

void addNodeToFront(Node node) {
    Node nextNode = head.next;
    head.next = node;
    node.previous = head;
    node.next = nextNode;
    nextNode.previous = node;
}

void removeNode(Node node) {
    Node previousNode = node.previous;

```

```

        Node nextNode = node.next;
        previousNode.next = nextNode;
        nextNode.previous = previousNode;
    }

    boolean isEmpty() {
        return head.next == tail;
    }

    Node removeLeastRecentlyUsed() {
        Node nodeToRemove = tail.previous;
        removeNode(nodeToRemove);
        return nodeToRemove;
    }
}

private final int cacheCapacity;
private int currentSize;
private int minimumFrequency;

private final Map<Integer, Node> keyToNodeMap;
private final Map<Integer, DoublyLinkedList> frequencyToListMap;

public LFUCache(int capacity) {
    this.cacheCapacity = capacity;
    this.currentSize = 0;
    this.minimumFrequency = 0;
    this.keyToNodeMap = new HashMap<>();
    this.frequencyToListMap = new HashMap<>();
}

public int get(int key) {
    if (!keyToNodeMap.containsKey(key)) {
        return -1;
    }
}

```

```

        Node accessedNode = keyToNodeMap.get(key);
        updateNodeFrequency(accessedNode);
        return accessedNode.value;
    }

    public void put(int key, int value) {
        if (cacheCapacity == 0) {
            return;
        }

        if (keyToNodeMap.containsKey(key)) {
            Node existingNode = keyToNodeMap.get(key);
            existingNode.value = value;
            updateNodeFrequency(existingNode);
        } else {
            if (currentSize == cacheCapacity) {
                DoublyLinkedList leastFrequentList = frequencyToListMap.get(minimumFrequency);
                Node nodeToRemove = leastFrequentList.removeLeastRecentlyUsed();
                keyToNodeMap.remove(nodeToRemove.key);
                currentSize--;
            }
        }

        Node newNode = new Node(key, value);
        keyToNodeMap.put(key, newNode);
        frequencyToListMap
            .computeIfAbsent(1, unused → new DoublyLinkedList())
            .addNodeToFront(newNode);
        minimumFrequency = 1;
        currentSize++;
    }
}

private void updateNodeFrequency(Node node) {
    int currentFrequency = node.frequency;
    DoublyLinkedList currentList = frequencyToListMap.get(currentFrequency);
    currentList.removeNode(node);
}

```

```
if (currentFrequency == minimumFrequency && currentList.isEmpty()) {  
    minimumFrequency++;  
}  
  
node.frequency++;  
frequencyToListMap  
    .computeIfAbsent(node.frequency, unused → new DoublyLinkedList())  
    .addNodeToFront(node);  
}  
}
```

### 3. Most Recently Used (MRU Cache)

move least recently used key to the head, remove from head

The Most Recently Used (MRU) Cache is a type of cache eviction policy where, when the cache is full, the item that was most recently used is removed first to make space for a new item.

This is the opposite of the Least Recently Used (LRU) cache, where the least recently accessed item is removed.

#### Use Case

The MRU cache is useful in scenarios where the least used items are more likely to be accessed again, and the most recently accessed items are less likely to be needed in the future.

#### Operations

Just like other caches, MRU should support:

- `get(key)`: Retrieve the value (will mark it as most recently used).
- `put(key, value)`: Insert or update a value (may evict the most recently used item if at capacity).

```

class MRUCache {
    class Node {
        int key;
        int value;
        Node prev;
        Node next;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }

    private int capacity;
    private Map<Integer, Node> map;
    private Node head;
    private Node tail;

    public MRUCache(int capacity) {
        this.capacity = capacity;
        map = new HashMap<>();
        head = new Node(-1, -1); // Dummy head (most recently used)
        tail = new Node(-1, -1); // Dummy tail (least recently used)
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        if (!map.containsKey(key)) {
    
```

```

        return -1;
    }

    Node node = map.get(key);
    moveToHead(node); // Mark as most recently used
    return node.value;
}

public void put(int key, int value) {
    if (map.containsKey(key)) {
        Node node = map.get(key);
        node.value = value;
        moveToHead(node);
    } else {
        Node newNode = new Node(key, value);
        map.put(key, newNode);
        addToHead(newNode);

        if (map.size() > capacity) {
            Node removed = removeHead(); // MRU eviction
            map.remove(removed.key);
        }
    }
}

private void moveToHead(Node node) {
    removeNode(node);
    addToHead(node);
}

private void removeNode(Node node) {
    node.prev.next = node.next;
    node.next.prev = node.prev;
}

private void addToHead(Node node) {

```

```

        node.next = head.next;
        node.next.prev = node;
        head.next = node;
        node.prev = head;
    }

private Node removeHead() {
    Node nodeToRemove = head.next;
    removeNode(nodeToRemove);
    return nodeToRemove;
}

public static void main(String[] args) {
    MRUCache cache = new MRUCache(2); // Capacity is 2
    cache.put(1, 1); // Cache: [1]
    cache.put(2, 2); // Cache: [2, 1]
    System.out.println(cache.get(1)); // Output: 1 → Cache: [1, 2]
    cache.put(3, 3); // Evicts 1 (MRU), Cache: [3, 2]
    System.out.println(cache.get(2)); // Output: 2 → Cache: [2, 3]
    cache.put(4, 4); // Evicts 2 (MRU), Cache: [4, 3]
    System.out.println(cache.get(3)); // Output: 3
    System.out.println(cache.get(4)); // Output: 4
}
}

```

---

#### 4. Min Stack

## 155. Min Stack

Medium

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with  $O(1)$  time complexity for each function.

**Example 1:**

**Input** ["MinStack","push","push","push","getMin","pop","top","getMin"]  
[[], [-2], [0], [-3], [], [], [], []]

**Output**

[null, null, null, null, -3, null, 0, -2]

**Explanation**

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top(); // return 0
minStack.getMin(); // return -2
```

**Constraints:**

- $-2^{31} \leq \text{val} \leq 2^{31} - 1$
- Methods `pop`, `top` and `getMin` operations will always be called on non-empty stacks.
- At most  $3 * 10^4$  calls will be made to `push`, `pop`, `top`, and `getMin`.

```
import java.util.Stack;

class MinStack {
```

```
private Stack<Integer> stack;
private Stack<Integer> minStack;

public MinStack() {
    stack = new Stack<>();
    minStack = new Stack<>();
}

public void push(int val) {
    stack.push(val);
    if (minStack.isEmpty() || val <= minStack.peek()) {
        minStack.push(val);
    }
}

public void pop() {
    if (stack.peek().equals(minStack.peek())) {
        minStack.pop();
    }
    stack.pop();
}

public int top() {
    return stack.peek();
}

public int getMin() {
    return minStack.peek();
}
```

---

5. Max Stack

Design a max stack that supports push, pop, top, peekMax and popMax.

- `push(x)` -- Push element `x` onto stack.
- `pop()` -- Remove the element on top of the stack and return it.
- `top()` -- Get the element on the top.
- `peekMax()` -- Retrieve the maximum element in the stack.
- `popMax()` -- Retrieve the maximum element in the stack, and remove it. If you find more than one maximum elements, only remove the top-most one.

**Example 1:**

```
MaxStack stack = new MaxStack();
stack.push(5);
stack.push(1);
stack.push(5);
stack.top(); -> 5
stack.popMax(); -> 5
stack.top(); -> 1
stack.peekMax(); -> 5
stack.pop(); -> 1
stack.top(); -> 5
```

**Note:**

- $-1e7 \leq x \leq 1e7$
- Number of operations won't exceed 10000.
- The last four operations won't be called when stack is empty.

```
import java.util.ArrayList;
import java.util.List;
import java.util.TreeMap;

class MaxStack {
    static class Node {
        public int val;
        public Node prev, next;

        public Node() {
        }

        public Node(int val) {
            this.val = val;
        }
    }
}
```

```
static class DoubleLinkedList {
    private final Node head = new Node();
    private final Node tail = new Node();

    public DoubleLinkedList() {
        head.next = tail;
        tail.prev = head;
    }

    public Node append(int val) {
        Node node = new Node(val);
        node.next = tail;
        node.prev = tail.prev;
        tail.prev = node;
        node.prev.next = node;
        return node;
    }

    public static Node remove(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
        return node;
    }

    public Node pop() {
        return remove(tail.prev);
    }

    public int peek() {
        return tail.prev.val;
    }
}

private DoubleLinkedList stack = new DoubleLinkedList();
private TreeMap<Integer, List<Node>> map = new TreeMap<>();
```

```
public MaxStack() {  
}  
  
public void push(int x) {  
    Node node = stack.append(x);  
    map.computeIfAbsent(x, k → new ArrayList<>()).add(node);  
}  
  
public int pop() {  
    Node node = stack.pop();  
    List<Node> nodes = map.get(node.val);  
    int x = nodes.remove(nodes.size() - 1).val;  
    if (nodes.isEmpty()) {  
        map.remove(node.val);  
    }  
    return x;  
}  
  
public int top() {  
    return stack.peek();  
}  
  
public int peekMax() {  
    return map.lastKey();  
}  
  
public int popMax() {  
    int x = peekMax();  
    List<Node> nodes = map.get(x);  
    Node node = nodes.remove(nodes.size() - 1);  
    if (nodes.isEmpty()) {  
        map.remove(x);  
    }  
    DoubleLinkedList.remove(node);  
    return x;  
}
```

```
}

public static void main(String[] args) {
    MaxStack stack = new MaxStack();

    stack.push(5);
    stack.push(1);
    stack.push(5);

    System.out.println("Top: " + stack.top());      // Output: 5
    System.out.println("PopMax: " + stack.popMax()); // Output: 5
    System.out.println("Top: " + stack.top());      // Output: 1
    System.out.println("PeekMax: " + stack.peekMax()); // Output: 5
    System.out.println("Pop: " + stack.pop());      // Output: 1
    System.out.println("Top: " + stack.top());      // Output: 5
}

/***
 * Your MaxStack object will be instantiated and called as such:
 * MaxStack obj = new MaxStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.peekMax();
 * int param_5 = obj.popMax();
 */

```

## Step-by-Step Execution:

### Initial State:

- Stack: Empty
  - TreeMap: Empty
- 

#### push(5)

- DoubleLinkedList → head  $\leftrightarrow$  5  $\leftrightarrow$  tail
  - TreeMap: { 5: [Node(5)] }
- 

#### push(1)

- DoubleLinkedList → head  $\leftrightarrow$  5  $\leftrightarrow$  1  $\leftrightarrow$  tail
  - TreeMap: { 1: [Node(1)], 5: [Node(5)] }
- 

#### push(5) (again)

- DoubleLinkedList → head  $\leftrightarrow$  5  $\leftrightarrow$  1  $\leftrightarrow$  5  $\leftrightarrow$  tail
- TreeMap: { 1: [Node(1)], 5: [Node(5a), Node(5b)] }

◆ `top()`

- Returns last pushed: 5

■ Output:

css

Copy Edit

```
Top: 5
```

◆ `popMax()`

- `peekMax() → 5` (highest key in TreeMap)
- Remove **last inserted Node(5)** from map and doubly linked list
- `DoubleLinkedList → head <-> 5 <-> 1 <-> tail`
- `TreeMap: { 1: [Node(1)], 5: [Node(5a)] }`

■ Output:

makefile

Copy Edit

```
PopMax: 5
```

◆ `top()`

- Returns last node: 1

■ Output:

css

Copy Edit

```
Top: 1
```

◆ peekMax()

- TreeMap lastKey → 5

■ Output:

makefile

Copy Edit

PeekMax: 5

◆ pop()

- Remove last inserted node (Node with value 1)
- DoubleLinkedList → head <=> 5 <=> tail
- TreeMap: { 5: [Node(5a)] }

■ Output:

makefile

Copy Edit

Pop: 1

◆ top()

- Last node is 5

■ Output:

css

Copy Edit

Top: 5

---

## 6. Maximum Frequency Stack

Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack.

Implement the `FreqStack` class:

- `FreqStack()` constructs an empty frequency stack.
  - `void push(int val)` pushes an integer `val` onto the top of the stack.
  - `int pop()` removes and returns the most frequent element in the stack.
- If there is a tie for the most frequent element, the element closest to the stack's top is removed and returned.

Example 1:

Input

```
["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop",
 "pop", "pop"]
[[], [5], [7], [5], [7], [4], [5], [], [], [], []]
```

Output

```
[null, null, null, null, null, null, null, 5, 7, 5, 4]
```

Explanation

```
FreqStack freqStack = new FreqStack();
freqStack.push(5); // The stack is [5]
freqStack.push(7); // The stack is [5,7]
freqStack.push(5); // The stack is [5,7,5]
freqStack.push(7); // The stack is [5,7,5,7]
freqStack.push(4); // The stack is [5,7,5,7,4]
freqStack.push(5); // The stack is [5,7,5,7,4,5]
freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes
[5,7,5,7,4].
freqStack.pop(); // return 7, as 5 and 7 is the most frequent, but 7 is
closest to the top. The stack becomes [5,7,5,4].
freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes
[5,7,4].
freqStack.pop(); // return 4, as 4, 5 and 7 is the most frequent, but 4 is
closest to the top. The stack becomes [5,7].
```

```
import java.util.*;

public class FreqStack {
    private int maxFreq = 0;
    private Map<Integer, Integer> count = new HashMap<>();
    private Map<Integer, Deque<Integer>> countToStack = new HashMap<>();

    public void push(int val) {
```

```

        count.merge(val, 1, Integer::sum);
    //  count.put(val, count.getOrDefault(val, 0) + 1);
    int freq = count.get(val);
    countToStack.putIfAbsent(freq, new ArrayDeque<>());
    countToStack.get(freq).push(val);
    maxFreq = Math.max(maxFreq, freq);
}

public int pop() {
    Deque<Integer> stack = countToStack.get(maxFreq);
    int val = stack.pop();
    count.merge(val, -1, Integer::sum);
    if (stack.isEmpty()) {
        countToStack.remove(maxFreq);
        maxFreq--;
    }
    return val;
}

public static void main(String[] args) {
    FreqStack freqStack = new FreqStack();
    freqStack.push(5);
    freqStack.push(7);
    freqStack.push(5);
    freqStack.push(7);
    freqStack.push(4);
    freqStack.push(5);
    System.out.println(freqStack.pop()); // Output: 5 (frequency 3)
    System.out.println(freqStack.pop()); // Output: 7 (frequency 2)
    System.out.println(freqStack.pop()); // Output: 5 (frequency 2)
    System.out.println(freqStack.pop()); // Output: 4 (frequency 1)
}

}

```

## 7. Implement Stack using Queues

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (`push`, `top`, `pop`, and `empty`).

Implement the `MyStack` class:

- `void push(int x)` Pushes element `x` to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

**Notes:**

- You must use only standard operations of a queue, which means that only `push to back`, `peek/pop from front`, `size` and `is empty` operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

**Example 1:**

```
Input
["MyStack", "push", "push", "top", "pop", "empty"]
[], [1], [2], [], [2], []
Output
[null, null, null, 2, 2, false]
Explanation
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // return 2
myStack.pop(); // return 2
myStack.empty(); // return False
```

```
import java.util.LinkedList;
import java.util.Queue;

class MyStack {

    private Queue<Integer> queue1;
    private Queue<Integer> queue2;

    public MyStack() {
        queue1 = new LinkedList<>();
        queue2 = new LinkedList<>();
```

```

}

public void push(int x) {
    queue2.offer(x);
    while (!queue1.isEmpty()) {
        queue2.offer(queue1.poll());
    }
    Queue<Integer> temp = queue1;
    queue1 = queue2;
    queue2 = temp;
}

public int pop() {
    if (queue1.isEmpty()) {
        throw new RuntimeException("Stack is empty");
    }
    return queue1.poll();
}

public int top() {
    if (queue1.isEmpty()) {
        throw new RuntimeException("Stack is empty");
    }
    return queue1.peek();
}

public boolean empty() {
    return queue1.isEmpty();
}

/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack obj = new MyStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 */

```

```
* int param_3 = obj.top();
* boolean param_4 = obj.empty();
*/
```

## 8. Implement Queue using Stacks

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `peek`, `pop`, and `empty`).

Implement the `MyQueue` class:

- `void push(int x)` Pushes element `x` to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.

Notes:

- You must use only standard operations of a stack, which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

```
Input["MyQueue", "push", "push", "peek", "pop", "empty"]
[], [1], [2], [], [], []
Output
[null, null, null, 1, 1, false]
ExplanationMyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

```
import java.util.Stack;

class MyQueue {

    private Stack<Integer> stack1;
    private Stack<Integer> stack2;
```

```

public MyQueue() {
    stack1 = new Stack<>();
    stack2 = new Stack<>();
}

public void push(int x) {
    stack1.push(x);
}

public int pop() {
    if (stack2.isEmpty()) {
        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }
    return stack2.pop();
}

public int peek() {
    if (stack2.isEmpty()) {
        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }
    return stack2.peek();
}

public boolean empty() {
    return stack1.isEmpty() && stack2.isEmpty();
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * MyQueue obj = new MyQueue();
 * obj.push(x);
 */

```

```
* int param_2 = obj.pop();
* int param_3 = obj.peek();
* boolean param_4 = obj.empty();
*/
```

---

## 9. Dinner Plate Stack

You have an infinite number of stacks arranged in a row and numbered (left to right) from 0, each of the stacks has the same maximum capacity.

Implement the `DinnerPlates` class:

- `DinnerPlates(int capacity)` Initializes the object with the maximum capacity of the stacks `capacity`.
- `void push(int val)` Pushes the given integer `val` into the leftmost stack with a size less than `capacity`.
- `int pop()` Returns the value at the top of the rightmost non-empty stack and removes it from that stack, and returns `-1` if all the stacks are empty.
- `int popAtStack(int index)` Returns the value at the top of the stack with the given index `index` and removes it from that stack or returns `-1` if the stack with that given index is empty.

**Example 1:****Input**

```
["DinnerPlates", "push", "push", "push", "push", "popAtStack", "push", "push",
 "popAtStack", "popAtStack", "pop", "pop", "pop", "pop", "pop"]
 [[2], [1], [2], [3], [4], [5], [0], [20], [21], [0], [2], [], [], [], []]
```

**Output**

```
[null, null, null, null, null, null, 2, null, null, 20, 21, 5, 4, 3, 1, -1]
```

**Explanation:**

```
DinnerPlates D = DinnerPlates(2); // Initialize with capacity = 2
```

```
D.push(1);
```

```
D.push(2);
```

```
D.push(3);
```

```
D.push(4);
```

```
D.push(5); // The stacks are now: 2 4
```

```
1 3 5  
└─ ─ ─
```

```
D.popAtStack(0); // Returns 2. The stacks are now: 4
```

```
1 3 5  
└─ ─ ─
```

```
D.push(20); // The stacks are now: 20 4
```

```
1 3 5  
└─ ─ ─
```

```
D.push(21); // The stacks are now: 20 4 21
```

```
1 3 5  
└─ ─ ─
```

```
D.popAtStack(0); // Returns 20. The stacks are now: 4 21
```

```
1 3 5  
└─ ─ ─
```

```
D.popAtStack(2); // Returns 21. The stacks are now: 4
```

```
1 3 5  
└─ ─ ─
```

```
D.pop(); // Returns 5. The stacks are now: 4
```

```
1 3  
└─ ─
```

```
D.pop(); // Returns 4. The stacks are now: 1 3
```

```
1  
└─ ─
```

```
D.pop(); // Returns 3. The stacks are now: 1
```

```
1  
└─ ─
```

```
D.pop(); // Returns 1. There are no stacks.
```

```
D.pop(); // Returns -1. There are still no stacks.
```

```
import java.util.*;
```

```
class DinnerPlates {  
    private final int capacity;  
    private final List<Stack<Integer>> stacks;
```

```

private final TreeSet<Integer> available;

public DinnerPlates(int capacity) {
    this.capacity = capacity;
    this.stacks = new ArrayList<>();
    this.available = new TreeSet<>();
}

public void push(int val) {
    if (!available.isEmpty()) {
        int index = available.first();
        stacks.get(index).push(val);
        if (stacks.get(index).size() == capacity) {
            available.remove(index);
        }
        return;
    }

    Stack<Integer> newStack = new Stack<>();
    newStack.push(val);
    stacks.add(newStack);
    if (capacity > 1) {
        available.add(stacks.size() - 1);
    }
}

public int pop() {
    int index = stacks.size() - 1;
    while (index >= 0 && stacks.get(index).isEmpty()) {
        stacks.remove(index);
        available.remove(index);
        index--;
    }

    if (index < 0)
        return -1;
}

```

```

Stack<Integer> stack = stacks.get(index);
int val = stack.pop();
if (stack.size() < capacity) {
    available.add(index);
}

return val;
}

public int popAtStack(int index) {
    if (index >= stacks.size() || stacks.get(index).isEmpty()) {
        return -1;
    }

    Stack<Integer> stack = stacks.get(index);
    int val = stack.pop();
    available.add(index);
    return val;
}

public static void main(String[] args) {
    DinnerPlates D = new DinnerPlates(2);

    D.push(1);
    D.push(2);
    D.push(3);
    D.push(4);
    D.push(5);

    System.out.println(D.popAtStack(0)); // 2
    D.push(20);
    D.push(21);
    System.out.println(D.popAtStack(0)); // 20
    System.out.println(D.popAtStack(2)); // 21
    System.out.println(D.pop()); // 5
}

```

```

        System.out.println(D.pop()); // 4
        System.out.println(D.pop()); // 3
        System.out.println(D.pop()); // 1
        System.out.println(D.pop()); // -1
    }
}
/**/
* Your DinnerPlates object will be instantiated and called as such:
* DinnerPlates obj = new DinnerPlates(capacity);
* obj.push(val);
* int param_2 = obj.pop();
* int param_3 = obj.popAtStack(index);
*/

```

10. How would you design a class that behaves similar to a Java Map <Date, Object> except that get() returns the mapped value for not only an exact match but also for the latest Date in the Map

```

import java.util.Date;
import java.util.TreeMap;

public class DateValueMap {
    private TreeMap<Date, Object> map;

    public DateValueMap() {
        this.map = new TreeMap<>();
    }

    public void put(Date date, Object value) {
        map.put(date, value);
    }

    public Object get(Date date) {
        Date floorKey = map.floorKey(date);
        if (floorKey == null) {
            return null;
        }
        Date ceilingKey = map.ceilingKey(date);
        if (ceilingKey != null &amp; date.equals(ceilingKey)) {
            return map.get(date);
        }
        Date previousKey = map.previousKey(date);
        if (previousKey != null &amp; date.equals(previousKey)) {
            return map.get(date);
        }
        return map.get(floorKey);
    }
}

```

```
    }
    return map.get(floorKey);
}

public void remove(Date date) {
    map.remove(date);
}

public void clear() {
    map.clear();
}
}
```

---

## 11. Design a Stack With Increment Operation

- Use two ArrayLists stack and inc
- Update inc[index] as inc[index] + val upon element addition
- And capacity variable
- Based on the index array update the stack elements

```

Input
["CustomStack","push","push","pop","push","push","increment","increment","pop","pop",
"pop","pop"]
[[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[],[],[]]
Output
[null,null,null,2,null,null,null,null,103,202,201,-1]
Explanation
CustomStack stk = new CustomStack(3); // Stack is Empty []
stk.push(1); // stack becomes [1]
stk.push(2); // stack becomes [1, 2]
stk.pop(); // return 2 --> Return top of the stack 2, stack
becomes [1]
stk.push(2); // stack becomes [1, 2]
stk.push(3); // stack becomes [1, 2, 3]
stk.push(4); // stack still [1, 2, 3], Do not add another elements
as size is 4
stk.increment(5, 100); // stack becomes [101, 102, 103]
stk.increment(2, 100); // stack becomes [201, 202, 103]
stk.pop(); // return 103 --> Return top of the stack 103, stack
becomes [201, 202]
stk.pop(); // return 202 --> Return top of the stack 202, stack
becomes [201]
stk.pop(); // return 201 --> Return top of the stack 201, stack
becomes []
stk.pop(); // return -1 --> Stack is empty return -1.

```

```

import java.util.*;

class CustomStack {
    private ArrayList<Integer> stack;
    private ArrayList<Integer> inc;
    private int maxSize;

    public CustomStack(int maxSize) {
        this.stack = new ArrayList<>();
        this.inc = new ArrayList<>();
        this.maxSize = maxSize;
    }

    public void push(int x) {
        if (stack.size() < maxSize) {
            stack.add(x);
            inc.add(0); // Initialize increment for the new element
        }
    }

    public int pop() {
        if (stack.isEmpty()) {
            return -1;
        }
        int top = stack.get(stack.size() - 1);
        stack.remove(stack.size() - 1);
        if (!inc.isEmpty() && inc.size() == 1) {
            inc.remove(0);
        } else {
            int incValue = inc.get(inc.size() - 1);
            inc.set(inc.size() - 1, incValue + 1);
        }
        return top;
    }

    public int increment(int k, int val) {
        if (k >= stack.size() || stack.isEmpty()) {
            return -1;
        }
        int start = stack.size() - k;
        for (int i = start; i < stack.size(); i++) {
            stack.set(i, stack.get(i) + val);
            if (!inc.isEmpty() && inc.size() == 1) {
                inc.remove(0);
            } else {
                int incValue = inc.get(inc.size() - 1);
                inc.set(inc.size() - 1, incValue + 1);
            }
        }
        return stack.get(start);
    }
}

```

```

        }

    }

    public int pop() {
        if (stack.isEmpty()) {
            return -1;
        }
        int idx = stack.size() - 1;
        int res = stack.get(idx) + inc.get(idx);
        if (idx > 0) {
            inc.set(idx - 1, inc.get(idx - 1) + inc.get(idx));
        }
        stack.remove(idx);
        inc.remove(idx);
        return res;
    }

    public void increment(int k, int val) {
        int idx = Math.min(k, stack.size()) - 1;
        if (idx >= 0) {
            inc.set(idx, inc.get(idx) + val);
        }
    }
}

/**
 * Your CustomStack object will be instantiated and called as such:
 * CustomStack obj = new CustomStack(maxSize);
 * obj.push(x);
 * int param_2 = obj.pop();
 * obj.increment(k,val);
 */

```

---

## 12. Design HashSet

- Use Node[] buckets for storing the elements
- Use SIZE = 1009 for prime distribution
- Using hash= key % SIZE, find the appropriate bucket and perform operations
- If key already exists in the add operation return

Design a HashSet without using any built-in hash table libraries.

Implement MyHashSet class:

- void add(key) Inserts the value key into the HashSet.
- bool contains(key) Returns whether the value key exists in the HashSet or not.
- void remove(key) Removes the value key in the HashSet. If key does not exist in the HashSet, do nothing.

**Example 1:**

**Input**  
 ["MyHashSet", "add", "add", "contains", "contains", "add", "contains", "remove", "contains"]  
 [[], [1], [2], [1], [3], [2], [2], [2], [2]]  
**Output**  
 [null, null, null, true, false, null, true, null, false]  
**Explanation**  

```
MyHashSet myHashSet = new MyHashSet();
    myHashSet.add(1);      // set = [1]
    myHashSet.add(2);      // set = [1, 2]
    myHashSet.contains(1); // return True
    myHashSet.contains(3); // return False, (not found)
    myHashSet.add(2);      // set = [1, 2]
    myHashSet.contains(2); // return True
    myHashSet.remove(2);   // set = [1]
    myHashSet.contains(2); // return False, (already removed)
```

**Constraints:**

- $0 \leq \text{key} \leq 10^6$
- At most  $10^4$  calls will be made to add, remove, and contains.

```
class MyHashSet {
    private final int SIZE = 1009; // Prime number for better distribution
    private Node[] buckets;

    public MyHashSet() {
```

```

        buckets = new Node[SIZE];
    }

private int hash(int key) {
    return key % SIZE;
}

public void add(int key) {
    int index = hash(key);
    if (buckets[index] == null) {
        buckets[index] = new Node(key);
    } else {
        Node curr = buckets[index];
        while (true) {
            if (curr.key == key) return; // Key already exists
            if (curr.next == null) break;
            curr = curr.next;
        }
        curr.next = new Node(key);
    }
}

public void remove(int key) {
    int index = hash(key);
    Node curr = buckets[index];
    if (curr == null) return;

    if (curr.key == key) {
        buckets[index] = curr.next;
        return;
    }

    while (curr.next != null) {
        if (curr.next.key == key) {
            curr.next = curr.next.next;
            return;
        }
    }
}

```

```

        }
        curr = curr.next;
    }
}

public boolean contains(int key) {
    int index = hash(key);
    Node curr = buckets[index];
    while (curr != null) {
        if (curr.key == key) return true;
        curr = curr.next;
    }
    return false;
}

private static class Node {
    int key;
    Node next;

    Node(int key) {
        this.key = key;
    }
}

```

---

### 13. [Design HashMap](#)

- Use Node[] buckets for storing the elements
- Use Node of key, value and next
- Use SIZE = 1009 for prime distribution
- Using hash= key % SIZE, find the appropriate bucket and perform operations

Design a `HashMap` without using any built-in hash table libraries.

Implement the `MyHashMap` class:

- `MyHashMap()` initializes the object with an empty map.
- `void put(int key, int value)` inserts a `(key, value)` pair into the `HashMap`. If the `key` already exists in the map, update the corresponding `value`.
- `int get(int key)` returns the `value` to which the specified `key` is mapped, or `-1` if this map contains no mapping for the `key`.
- `void remove(key)` removes the `key` and its corresponding `value` if the map contains the mapping for the `key`.

**Example 1:**

**Input**

```
["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]
[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]
```

**Output**

```
[null, null, null, 1, -1, null, 1, null, -1]
```

**Explanation**

```
MyHashMap myHashMap = new MyHashMap();
myHashMap.put(1, 1); // The map is now [[1,1]]
myHashMap.put(2, 2); // The map is now [[1,1], [2,2]]
myHashMap.get(1); // return 1, The map is now [[1,1], [2,2]]
myHashMap.get(3); // return -1 (i.e., not found), The map is now [[1,1], [2,2]]
myHashMap.put(2, 1); // The map is now [[1,1], [2,1]] (i.e., update the existing value)
myHashMap.get(2); // return 1, The map is now [[1,1], [2,1]]
myHashMap.remove(2); // remove the mapping for 2, The map is now [[1,1]]
myHashMap.get(2); // return -1 (i.e., not found), The map is now [[1,1]]
```

```
class MyHashMap {
    private static class Node {
        int key, value;
        Node next;
        Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }

    private final int SIZE = 10000;
    private Node[] buckets;
```

```

public MyHashMap() {
    buckets = new Node[SIZE];
}

private int hash(int key) {
    return key % SIZE;
}

public void put(int key, int value) {
    int index = hash(key);
    if (buckets[index] == null) {
        buckets[index] = new Node(-1, -1); // dummy head
    }
    Node prev = findNode(buckets[index], key);
    if (prev.next == null) {
        prev.next = new Node(key, value);
    } else {
        prev.next.value = value; // update existing
    }
}

public int get(int key) {
    int index = hash(key);
    if (buckets[index] == null) return -1;
    Node prev = findNode(buckets[index], key);
    return (prev.next == null) ? -1 : prev.next.value;
}

public void remove(int key) {
    int index = hash(key);
    if (buckets[index] == null) return;
    Node prev = findNode(buckets[index], key);
    if (prev.next != null) {
        prev.next = prev.next.next;
    }
}

```

```

    }

private Node findNode(Node head, int key) {
    Node prev = head;
    while (prev.next != null && prev.next.key != key) {
        prev = prev.next;
    }
    return prev;
}

/**
 * Your MyHashMap object will be instantiated and called as such:
 * MyHashMap obj = new MyHashMap();
 * obj.put(key,value);
 * int param_2 = obj.get(key);
 * obj.remove(key);
 */

```

---

#### 14. [Design Linked List](#)

```

Input
["MyLinkedList", "addAtHead", "addAtTail", "addAtIndex", "get", "deleteAtIndex", "get"]
[[], [1], [3], [1, 2], [1], [1], [1]]
Output
[null, null, null, null, 2, null, 3]

Explanation
MyLinkedList myLinkedList = new MyLinkedList();
myLinkedList.addAtHead(1);
myLinkedList.addAtTail(3);
myLinkedList.addAtIndex(1, 2);      // linked list becomes 1->2->3
myLinkedList.get(1);              // return 2
myLinkedList.deleteAtIndex(1);    // now the linked list is 1->3
myLinkedList.get(1);              // return 3

```

```

class MyLinkedList {

```

```

private class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
    }
}

private ListNode head;
private int size;

public MyLinkedList() {
    head = null;
    size = 0;
}

// Get the value of the index-th node in the linked list
public int get(int index) {
    if (index < 0 || index >= size) return -1;

    ListNode curr = head;
    for (int i = 0; i < index; i++) {
        curr = curr.next;
    }
    return curr.val;
}

// Add a node of value val before the first element of the linked list
public void addAtHead(int val) {
    ListNode newNode = new ListNode(val);
    newNode.next = head;
    head = newNode;
    size++;
}

```

```

// Append a node of value val as the last element of the linked list
public void addAtTail(int val) {
    ListNode newNode = new ListNode(val);
    if (head == null) {
        head = newNode;
    } else {
        ListNode curr = head;
        while (curr.next != null) {
            curr = curr.next;
        }
        curr.next = newNode;
    }
    size++;
}

// Add a node of value val before the index-th node in the linked list
public void addAtIndex(int index, int val) {
    if (index > size || index < 0) return;
    if (index == 0) {
        addAtHead(val);
        return;
    }

    ListNode newNode = new ListNode(val);
    ListNode curr = head;
    for (int i = 0; i < index - 1; i++) {
        curr = curr.next;
    }
    newNode.next = curr.next;
    curr.next = newNode;
    size++;
}

// Delete the index-th node in the linked list, if the index is valid
public void deleteAtIndex(int index) {
    if (index < 0 || index >= size) return;
}

```

```

if (index == 0) {
    head = head.next;
} else {
    ListNode curr = head;
    for (int i = 0; i < index - 1; i++) {
        curr = curr.next;
    }
    curr.next = curr.next.next;
}
size--;
}
}

```

- Use ListNode of val and next
- Using ListNode head and size (capacity) perform operations

## 15. Two Sum III - Data structure design

Design and implement a TwoSum class. It should support the following operations: `add` and `find`.

`add` - Add the number to an internal data structure.

`find` - Find if there exists any pair of numbers which sum is equal to the value.

**Example 1:**

```

add(1); add(3); add(5);
find(4) -> true
find(7) -> false

```

**Example 2:**

```

add(3); add(1); add(2);
find(3) -> true
find(6) -> false

```

```

class TwoSum {

    private Map<Integer, Integer> map;

    public TwoSum() {
        map = new HashMap<>();
    }

    // Adds the number to the internal data structure
    public void add(int number) {
        map.put(number, map.getOrDefault(number, 0) + 1);
    }

    // Finds if there exists any pair of numbers which sum is equal to the value
    public boolean find(int value) {
        for (int num : map.keySet()) {
            int complement = value - num;
            if (complement == num) {
                // Need at least two occurrences of the same number
                if (map.get(num) >= 2) return true;
            } else {
                if (map.containsKey(complement)) return true;
            }
        }
        return false;
    }
}

```

## 16. Map Sum Pairs

- Use TrieNode with fields `TrieNode[] children` and `sum`
- Update the sum of the child node using delta of current value and existing sum upon insertion

- Return the sum of the child node

Design a map that allows you to do the following:

- Maps a string key to a given value.
- Returns the sum of the values that have a key with a prefix equal to a given string.

Implement the `MapSum` class:

- `MapSum()` Initializes the `MapSum` object.
- `void insert(String key, int val)` Inserts the `key-val` pair into the map. If the `key` already existed, the original `key-value` pair will be overridden to the new one.
- `int sum(string prefix)` Returns the sum of all the pairs' value whose `key` starts with the `prefix`.

**Example 1:**

**Input**  
`["MapSum", "insert", "sum", "insert", "sum"]  
 [[], ["apple", 3], ["ap"], ["app", 2], ["ap"]]`  
**Output**  
`[null, null, 3, null, 5]`

**Explanation**  
`MapSum mapSum = new MapSum();  
 mapSum.insert("apple", 3);  
 mapSum.sum("ap"); // return 3 (apple = 3)  
 mapSum.insert("app", 2);  
 mapSum.sum("ap"); // return 5 (apple + app = 3 + 2 = 5)`

```
class MapSum {
    private final TrieNode root;
    private final Map<String, Integer> map;

    public MapSum() {
        root = new TrieNode();
        map = new HashMap<>();
    }

    public void insert(String key, int val) {
        int delta = val - map.getOrDefault(key, 0);
        map.put(key, val);
        updateTrie(root, key, delta);
    }

    public int sum(String prefix) {
        return searchTrie(root, prefix);
    }
}

class TrieNode {
    Map<Character, TrieNode> children;
    int value;
    boolean isLeaf;

    public TrieNode() {
        children = new HashMap<>();
        value = 0;
        isLeaf = false;
    }
}
```

```

map.put(key, val);
TrieNode node = root;

for (char c : key.toCharArray()) {
    int index = c - 'a';
    if (node.children[index] == null) {
        node.children[index] = new TrieNode();
    }
    node = node.children[index];
    node.sum += delta;
}

public int sum(String prefix) {
    TrieNode node = root;

    for (char c : prefix.toCharArray()) {
        int index = c - 'a';
        if (node.children[index] == null) return 0;
        node = node.children[index];
    }

    return node.sum;
}

// static inner class TrieNode
static class TrieNode {
    TrieNode[] children;
    int sum;

    TrieNode() {
        children = new TrieNode[26]; // for lowercase 'a' to 'z'
        sum = 0;
    }
}

```

```
    }  
}
```

---

## 17. All O`one Data Structure

- Use Doubly Linked List with Bucket node, prev, next, count and set of strings
- Map<String, Integer> keyCountMap to track count of each string
- Map<Integer, Bucket> to get all the nodes with a given count
- Min key will be at the head.next node
- Max key will be all the tail.prev node
- Using count keep on updating the nodes

Design a data structure to store the strings' count with the ability to return the strings with minimum and maximum counts.

Implement the `AllOne` class:

- `AllOne()` Initializes the object of the data structure.
- `inc(String key)` Increments the count of the string `key` by `1`. If `key` does not exist in the data structure, insert it with count `1`.
- `dec(String key)` Decrements the count of the string `key` by `1`. If the count of `key` is `0` after the decrement, remove it from the data structure. It is guaranteed that `key` exists in the data structure before the decrement.
- `getMaxKey()` Returns one of the keys with the maximal count. If no element exists, return an empty string `""`.
- `getMinKey()` Returns one of the keys with the minimum count. If no element exists, return an empty string `""`.

Note that each function must run in `O(1)` average time complexity.

**Example 1:**

**Input**

```
["AllOne", "inc", "inc", "getMaxKey", "getMinKey", "inc", "getMaxKey", "getMinKey"]
[], ["hello"], ["hello"], [], [], ["leet"], [], []]
```

**Output**

```
[null, null, null, "hello", "hello", null, "hello", "leet"]
```

**Explanation**

```
AllOne allOne = new AllOne();
allOne.inc("hello");
allOne.inc("hello");
allOne.getMaxKey(); // return "hello"
allOne.getMinKey(); // return "hello"
allOne.inc("leet");
allOne.getMaxKey(); // return "hello"
allOne.getMinKey(); // return "leet"
```

```
class AllOne {
    // Bucket class for doubly linked list nodes
    private static class Bucket {
        int count;
        Set<String> keys;
        Bucket prev, next;

        Bucket(int count) {
            this.count = count;
            this.keys = new HashSet<>();
```

```

        }

    }

private Map<String, Integer> keyCountMap;
private Map<Integer, Bucket> countBucketMap;
private Bucket head, tail;

public AllOne() {
    keyCountMap = new HashMap<>();
    countBucketMap = new HashMap<>();
    head = new Bucket(Integer.MIN_VALUE);
    tail = new Bucket(Integer.MAX_VALUE);
    head.next = tail;
    tail.prev = head;
}

public void inc(String key) {
    int count = keyCountMap.getOrDefault(key, 0);
    keyCountMap.put(key, count + 1);

    Bucket currBucket = countBucketMap.get(count);
    Bucket newBucket = countBucketMap.get(count + 1);
    if (newBucket == null) {
        newBucket = new Bucket(count + 1);
        countBucketMap.put(count + 1, newBucket);
        insertAfter(currBucket != null ? currBucket : head, newBucket);
    }

    newBucket.keys.add(key);
    if (currBucket != null) {
        currBucket.keys.remove(key);
        if (currBucket.keys.isEmpty()) {
            removeBucket(currBucket);
            countBucketMap.remove(count);
        }
    }
}

```

```

}

public void dec(String key) {
    if (!keyCountMap.containsKey(key)) return;

    int count = keyCountMap.get(key);
    Bucket currBucket = countBucketMap.get(count);
    if (count == 1) {
        keyCountMap.remove(key);
    } else {
        keyCountMap.put(key, count - 1);
        Bucket newBucket = countBucketMap.get(count - 1);
        if (newBucket == null) {
            newBucket = new Bucket(count - 1);
            countBucketMap.put(count - 1, newBucket);
            insertBefore(currBucket, newBucket);
        }
        newBucket.keys.add(key);
    }

    currBucket.keys.remove(key);
    if (currBucket.keys.isEmpty()) {
        removeBucket(currBucket);
        countBucketMap.remove(count);
    }
}

public String getMaxKey() {
    return tail.prev != head && !tail.prev.keys.isEmpty() ? tail.prev.keys.iterator().next();
}

public String getMinKey() {
    return head.next != tail && !head.next.keys.isEmpty() ? head.next.keys.iterator().next();
}

// Helper methods

```

```
private void insertAfter(Bucket prev, Bucket newBucket) {  
    Bucket next = prev.next;  
    prev.next = newBucket;  
    newBucket.prev = prev;  
    newBucket.next = next;  
    next.prev = newBucket;  
}  
  
private void insertBefore(Bucket next, Bucket newBucket) {  
    Bucket prev = next.prev;  
    prev.next = newBucket;  
    newBucket.prev = prev;  
    newBucket.next = next;  
    next.prev = newBucket;  
}  
  
private void removeBucket(Bucket bucket) {  
    bucket.prev.next = bucket.next;  
    bucket.next.prev = bucket.prev;  
}
```

---

17.

Time Based Key-Value Store

Use `private Map<String, TreeMap<Integer, String>> map;`

Design a time-based key-value data structure that can store multiple values for the same key at different time stamps and retrieve the key's value at a certain timestamp.

Implement the `TimeMap` class:

- `TimeMap()` initializes the object of the data structure.
- `void set(String key, String value, int timestamp)` Stores the key `key` with the value `value` at the given time `timestamp`.
- `String get(String key, int timestamp)` Returns a value such that `set` was called previously, with `timestamp_prev <= timestamp`. If there are multiple such values, it returns the value associated with the largest `timestamp_prev`. If there are no values, it returns `""`.

**Example 1:**

**Input**

```
["TimeMap", "set", "get", "get", "set", "get", "get"]
[[], ["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["foo", 5]]
```

**Output**

```
[null, null, "bar", "bar", null, "bar2", "bar2"]
```

**Explanation**

```
TimeMap timeMap = new TimeMap();
timeMap.set("foo", "bar", 1); // store the key "foo" and value "bar" along with timestamp = 1.
timeMap.get("foo", 1); // return "bar"
timeMap.get("foo", 3); // return "bar", since there is no value corresponding to foo at timestamp 3 and timestamp 2, then the only value is at timestamp 1 is "bar".
timeMap.set("foo", "bar2", 4); // store the key "foo" and value "bar2" along with timestamp = 4.
timeMap.get("foo", 4); // return "bar2"
timeMap.get("foo", 5); // return "bar2"
```

```
class TimeMap {

    private Map<String, TreeMap<Integer, String>> map;

    public TimeMap() {
        map = new HashMap<>();
    }

    public void set(String key, String value, int timestamp) {
        map.putIfAbsent(key, new TreeMap<>());
        map.get(key).put(timestamp, value);
    }
}
```

```

    }

    public String get(String key, int timestamp) {
        if (!map.containsKey(key)) return "";
        TreeMap<Integer, String> timeMap = map.get(key);
        Map.Entry<Integer, String> entry = timeMap.floorEntry(timestamp);
        return entry != null ? entry.getValue() : "";
    }
}

```

## 18. Design Circular Queue

- Use two pointers front and rear
- For enqueue operation increment rear as  $(\text{rear} + 1) \% \text{capacity}$ , place val at  $\text{data}[\text{rear}]$  and increment size. Check if the queue is fully occupied with max capacity
- For dequeue operation, increment front as  $(\text{front} + 1) \% \text{capacity}$  and decrement size and check if the queue is empty or not.

**Input**  
`["MyCircularQueue", "enQueue", "enQueue", "enQueue", "enQueue", "enQueue", "Rear", "isFull", "deQueue", "enQueue", "Rear"]  
[[3], [1], [2], [3], [4], [], [], [], [4], []]`

**Output**  
`[null, true, true, true, false, 3, true, true, true, 4]`

### Explanation

```

MyCircularQueue myCircularQueue = new MyCircularQueue(3);
myCircularQueue.enQueue(1); // return True
myCircularQueue.enQueue(2); // return True
myCircularQueue.enQueue(3); // return True
myCircularQueue.enQueue(4); // return False
myCircularQueue.Rear(); // return 3
myCircularQueue.isFull(); // return True
myCircularQueue.deQueue(); // return True
myCircularQueue.enQueue(4); // return True
myCircularQueue.Rear(); // return 4

```

```

class MyCircularQueue {
    private int[] data;

```

```
private int front;
private int rear;
private int size;
private int capacity;

public MyCircularQueue(int k) {
    data = new int[k];
    front = 0;
    rear = -1;
    size = 0;
    capacity = k;
}

public boolean enqueue(int value) {
    if (isFull()) return false;
    rear = (rear + 1) % capacity;
    data[rear] = value;
    size++;
    return true;
}

public boolean dequeue() {
    if (isEmpty()) return false;
    front = (front + 1) % capacity;
    size--;
    return true;
}

public int Front() {
    return isEmpty() ? -1 : data[front];
}

public int Rear() {
    return isEmpty() ? -1 : data[rear];
}
```

```

public boolean isEmpty() {
    return size == 0;
}

public boolean isFull() {
    return size == capacity;
}
}

```

### 19. Design Circular Dequeue

- Use two pointers front and rear
- For insertLast operation increment rear as  $(rear + 1) \% capacity$ , place val at data[rear] and increment size. Check if the dequeue is fully occupied with max capacity. And for insertFront increment front as  $(front - 1 + capacity) \% capacity$ .
- For deleteFront operation, increment front as  $(front + 1) \% capacity$  and decrement size and check if the queue is empty or not. And for deleteLast increment rear as  $(rear - 1 + capacity) \% capacity$ .
- queue[front] will be the front and queue[(rear - 1 + capacity) \% capacity] will be the rear

#### Input

```

["MyCircularDeque", "insertLast", "insertLast", "insertFront", "insertFront", "getRear",
"isFull", "deleteLast", "insertFront", "getFront"]
[[3], [1], [2], [3], [4], [], [], [], [4], []]

```

#### Output

```
[null, true, true, true, false, 2, true, true, true, 4]
```

#### Explanation

```

MyCircularDeque myCircularDeque = new MyCircularDeque(3);
myCircularDeque.insertLast(1); // return True
myCircularDeque.insertLast(2); // return True
myCircularDeque.insertFront(3); // return True
myCircularDeque.insertFront(4); // return False, the queue is full.
myCircularDeque.getRear(); // return 2
myCircularDeque.isFull(); // return True
myCircularDeque.deleteLast(); // return True
myCircularDeque.insertFront(4); // return True
myCircularDeque.getFront(); // return 4

```

```

class MyCircularDeque {
    private int[] deque;

```

```
private int front;
private int rear;
private int size;
private int capacity;

public MyCircularDeque(int k) {
    capacity = k;
    deque = new int[k];
    front = 0;
    rear = 0;
    size = 0;
}

public boolean insertFront(int value) {
    if (isFull()) return false;
    front = (front - 1 + capacity) % capacity;
    deque[front] = value;
    size++;
    return true;
}

public boolean insertLast(int value) {
    if (isFull()) return false;
    deque[rear] = value;
    rear = (rear + 1) % capacity;
    size++;
    return true;
}

public boolean deleteFront() {
    if (isEmpty()) return false;
    front = (front + 1) % capacity;
    size--;
    return true;
}
```

```

public boolean deleteLast() {
    if (isEmpty()) return false;
    rear = (rear - 1 + capacity) % capacity;
    size--;
    return true;
}

public int getFront() {
    if (isEmpty()) return -1;
    return deque[front];
}

public int getRear() {
    if (isEmpty()) return -1;
    return deque[(rear - 1 + capacity) % capacity];
}

public boolean isEmpty() {
    return size == 0;
}

public boolean isFull() {
    return size == capacity;
}
}

```

---

## 21. Design In-Memory File System

Use TrieNode with fields String fileName, boolean isFile, StringBuilder fileContent and Map<String, TrieNode> children

Implement the FileSystem class:

- `FileSystem()` Initializes the object of the system.
- `List<String> ls(String path)`
  - If `path` is a file path, returns a list that only contains this file's name.
  - If `path` is a directory path, returns the list of file and directory names **in this directory**.

The answer should be in **lexicographic order**.

- `void mkdir(String path)` Makes a new directory according to the given `path`. The given directory path does not exist. If the middle directories in the path do not exist, you should create them as well.
- `void addContentToFile(String filePath, String content)`
  - If `filePath` does not exist, creates that file containing given `content`.
  - If `filePath` already exists, appends the given `content` to original content.
- `String readContentFromFile(String filePath)` Returns the content in the file at `filePath`.

#### Input

```
["FileSystem", "ls", "mkdir", "addContentToFile", "ls", "readContentFromFile"]
[[], ["/"], ["/a/b/c"], ["/a/b/c/d", "hello"], ["/"], ["/a/b/c/d"]]
```

#### Output

```
[null, [], null, ["a"], "hello"]
```

#### Explanation

```
FileSystem fileSystem = new FileSystem();
fileSystem.ls("/"); // return []
fileSystem.mkdir("/a/b/c");
fileSystem.addContentToFile("/a/b/c/d", "hello");
fileSystem.ls("/"); // return ["a"]
fileSystem.readContentFromFile("/a/b/c/d"); // return "hello"
```

```
import java.util.*;

class TrieNode {
    String fileName;
    boolean isFile;
    StringBuilder fileContent = new StringBuilder();
```

```

Map<String, TrieNode> children = new HashMap<>();

TrieNode insertPath(String path, boolean isFile) {
    TrieNode current = this;
    String[] segments = path.split("/");
    for (int i = 1; i < segments.length; ++i) {
        String segment = segments[i];
        current.children.putIfAbsent(segment, new TrieNode());
        current = current.children.get(segment);
    }
    current.isFile = isFile;
    if (isFile) {
        current.fileName = segments[segments.length - 1];
    }
    return current;
}

TrieNode findPath(String path) {
    TrieNode current = this;
    String[] segments = path.split("/");
    for (int i = 1; i < segments.length; ++i) {
        current = current.children.get(segments[i]);
        if (current == null) return null;
    }
    return current;
}

class FileSystem {
    private final TrieNode root;

    public FileSystem() {
        this.root = new TrieNode();
    }

    public List<String> ls(String path) {

```

```
TrieNode node = root.findPath(path);
List<String> result = new ArrayList<>();
if (node == null) return result;

if (node.isFile) {
    result.add(node.fileName);
} else {
    result.addAll(node.children.keySet());
    Collections.sort(result);
}
return result;
}

public void mkdir(String path) {
    root.insertPath(path, false);
}

public void addContentToFile(String filePath, String content) {
    TrieNode fileNode = root.insertPath(filePath, true);
    fileNode.fileContent.append(content);
}

public String readContentFromFile(String filePath) {
    TrieNode fileNode = root.findPath(filePath);
    return fileNode != null ? fileNode.fileContent.toString() : "";
}
```

## 22. Design Video Sharing Platform

```
PriorityQueue<Integer> userIds  
Map<Integer, String>  
videoidToVideo  
  
Map<Integer, String> videoidToViews  
  
Map<Integer, String> videoidToLikes  
  
Map<Integer, String> videoidToDislikes
```

Implement the `VideoSharingPlatform` class:

- `VideoSharingPlatform()` Initializes the object.
- `int upload(String video)` The user uploads a `video`. Return the `videoId` associated with the video.
- `void remove(int videoId)` If there is a video associated with `videoId`, remove the video.
- `String watch(int videoId, int startMinute, int endMinute)` If there is a video associated with `videoId`, increase the number of views on the video by `1` and return the substring of the video string starting at `startMinute` and ending at `min(endMinute, video.length - 1 ) (inclusive)`. Otherwise, return `"-1"`.
- `void like(int videoId)` Increases the number of likes on the video associated with `videoId` by `1` if there is a video associated with `videoId`.
- `void dislike(int videoId)` Increases the number of dislikes on the video associated with `videoId` by `1` if there is a video associated with `videoId`.
- `int[] getLikesAndDislikes(int videoId)` Return a **0-indexed** integer array `values` of length `2` where `values[0]` is the number of likes and `values[1]` is the number of dislikes on the video associated with `videoId`. If there is no video associated with `videoId`, return `[-1]`.
- `int getViews(int videoId)` Return the number of views on the video associated with `videoId`, if there is no video associated with `videoId`, return `-1`.

### Input

```
["VideoSharingPlatform", "upload", "upload", "remove", "remove", "upload",  
"watch", "watch", "like", "dislike", "dislike", "getLikesAndDislikes",  
"getViews"]
```

```
[[], ["123"], ["456"], [4], [0], ["789"], [1, 0, 5], [1, 0, 1], [1], [1], [1], [1]]
```

### Output

```
[null, 0, 1, null, null, 0, "456", "45", null, null, null, [1, 2], 2]
```

### Explanation

```
VideoSharingPlatform videoSharingPlatform = new VideoSharingPlatform();
videoSharingPlatform.upload("123");
// The smallest available videold is 0, so return 0.
videoSharingPlatform.upload("456");
// The smallest available videold is 1, so return 1.
videoSharingPlatform.remove(4);
// There is no video associated with videold 4, so do nothing.
videoSharingPlatform.remove(0);
// Remove the video associated with videold 0.
videoSharingPlatform.upload("789");
// Since the video associated with videold 0 was deleted,
// 0 is the smallest available videold, so return 0.
videoSharingPlatform.watch(1, 0, 5);
// The video associated with videold 1 is "456".
// The video from minute 0 to min(5, 3 - 1) = 2 is "456", so return "453".
videoSharingPlatform.watch(1, 0, 1);
// The video associated with videold 1 is "456".
// The video from minute 0 to min(1, 3 - 1) = 1 is "45", so return "45".
videoSharingPlatform.like(1);
// Increase the number of likes on the video associated with videold 1.
videoSharingPlatform.dislike(1);
// Increase the number of dislikes on the video associated with videold 1.
videoSharingPlatform.dislike(1);
// Increase the number of dislikes on the video associated with videold 1.
videoSharingPlatform.getLikesAndDislikes(1);
// There is 1 like and 2 dislikes on the video associated with videold 1,
// so return [1, 2].
videoSharingPlatform.getViews(1);
// The video associated with videold 1 has 2 views, so return 2.
```

```
class VideoSharingPlatform {
    public int upload(String video) {
```

```

final int videold = getVideold();
videoldToVideo.put(videold, video);
return videold;
}

public void remove(int videold) {
    if (videoldToVideo.containsKey(videold)) {
        usedIds.offer(videold);
        videoldToVideo.remove(videold);
        videoldToViews.remove(videold);
        videoldToLikes.remove(videold);
        videoldToDislikes.remove(videold);
    }
}

public String watch(int videold, int startMinute, int endMinute) {
    if (!videoldToVideo.containsKey(videold))
        return "-1";
    videoldToViews.merge(videold, 1, Integer::sum);
    final String video = videoldToVideo.get(videold);
    return video.substring(startMinute, Math.min(endMinute + 1, video.length()));
}

public void like(int videold) {
    if (videoldToVideo.containsKey(videold)) {
        videoldToLikes.merge(videold, 1, Integer::sum);
    }
}

public void dislike(int videold) {
    if (videoldToVideo.containsKey(videold)) {
        videoldToDislikes.merge(videold, 1, Integer::sum);
    }
}

public int[] getLikesAndDislikes(int videold) {

```

```

        return videoldToVideo.containsKey(videold)
    ? new int[] {
        videoldToLikes.getOrDefault(videold, 0),
        videoldToDislikes.getOrDefault(videold, 0)
    }
    : new int[] {-1};
}

public int getViews(int videold) {
    return videoldToVideo.containsKey(videold)
    ? videoldToViews.getOrDefault(videold, 0)
    : -1;
}

private int currVideold = 0;
private PriorityQueue<Integer> usedIds = new PriorityQueue<>();
private Map<Integer, String> videoldToVideo = new HashMap<>();
private Map<Integer, Integer> videoldToLikes = new HashMap<>();
private Map<Integer, Integer> videoldToDislikes = new HashMap<>();
private Map<Integer, Integer> videoldToViews = new HashMap<>();

private int getVideold() {
    if (usedIds.isEmpty())
        return currVideold++;
    return usedIds.poll();
}

}

```

Designing a **Video Sharing Platform** (like YouTube, TikTok, or Vimeo) involves several key architectural decisions, system design trade-offs, and feature planning. Here's a breakdown of how to approach it:



## Requirements

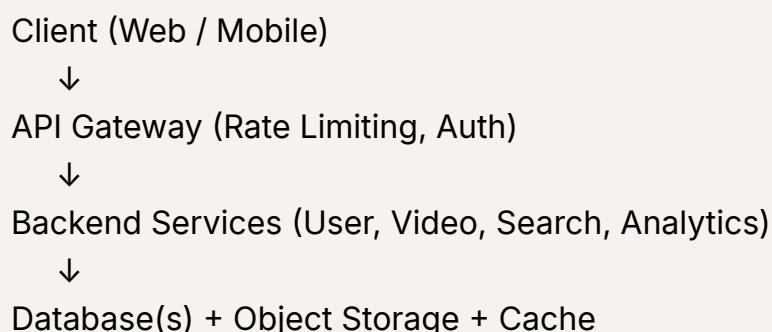
## 1. Functional Requirements

- User Registration/Login (OAuth or email/password)
- Upload videos (title, description, tags, thumbnail)
- Stream videos (adaptive bitrate)
- Like/Comment/Share/Subscribe
- Search videos (by title, tags, etc.)
- Recommendations (trending, personalized)
- View count tracking
- Video categorization/playlists
- Admin panel for moderation

## 2. Non-Functional Requirements

- Scalability (millions of users/videos)
- High availability
- Low latency for streaming
- Consistency in metadata
- Secure upload and access
- CDN integration for global delivery

## High-Level Architecture



↓  
CDN (for fast video delivery)

## Key Components

### 1. Frontend (React/Flutter)

- UI for upload, browse, profile
- Video player (HLS/DASH for streaming)
- Infinite scroll / short-form UI (if TikTok-style)

### 2. API Gateway

- Handles requests, routing, rate-limiting
- Auth validation (JWT/session-based)

### 3. User Service

- Register/Login
- User profile (avatar, bio, etc.)
- Subscriptions/following logic

### 4. Video Upload/Processing Service

- Accepts upload (possibly chunked)
- Pushes to a queue for transcoding (via FFmpeg)
- Stores in object storage (e.g., S3, GCS)
- Generates thumbnails & HLS chunks
- Writes metadata to database

### 5. Streaming Service

- Serves video via HLS/DASH
- Uses CDN for fast delivery

- Token-based URLs for secure access

## 6. Metadata/Video Service

- Stores video metadata (title, tags, uploader, timestamps)
- Controls visibility (public/private)
- Manages video retrieval for UI

## 7. Search Service

- Full-text search (Elasticsearch)
- Indexes title, description, tags
- Advanced filters (date, views, category)

## 8. Feed/Recommendation Service

- Trending (views/likes over time)
- Personalized (based on history + ML model)
- Followed user uploads

## 9. Comment & Like System

- Separate service for handling comments/likes
- Supports real-time updates via WebSockets (optional)

## 10. Analytics Service

- View count tracking
- Watch time, bounce rate
- Heatmaps (advanced)
- Event pipeline (Kafka → Clickhouse/BigQuery)

## 11. Moderation Tools

- Admin dashboard to report/delete
- Auto-flagging via AI (NSFW detection, copyright)

## Storage Choices

Data	Storage
Videos	Object storage (S3, GCS)
Metadata	Relational DB (PostgreSQL/MySQL)
Comments/Likes	NoSQL (MongoDB, DynamoDB)
Search	Elasticsearch
Analytics	Data Warehouse (BigQuery, Redshift)
Caching	Redis/Memcached

## Upload to Playback Flow

Client → Upload Endpoint → Video Service  
→ Store Original in S3  
→ Message Queue (e.g., Kafka)  
→ Transcoding Workers (FFMPEG)  
→ Upload HLS chunks to S3/CDN  
→ Update Metadata in DB  
→ Notify User (status)

## Advanced Features (Optional)

- **Live Streaming**
- **Monetization** (ads, subscriptions)
- **Short-form vs long-form video feed**
- **PWA support for mobile**
- **Edge caching via CDN + token-based HLS protection**
- **AI moderation (text/image/video classification)**

23.  Shuffle an Array

-  Use Random class

- Set original as nums.clone
- Iterate from i to n
- Set j as rand.nextInt(i + 1)
- Swap i and j
- Return shuffled array

```
Input
["Solution", "shuffle", "reset", "shuffle"]
[[[1, 2, 3]], [], [], []]
Output
[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]

Explanation
Solution solution = new Solution([1, 2, 3]);
solution.shuffle();      // Shuffle the array [1,2,3] and return its result.
                        // Any permutation of [1,2,3] must be equally likely to be returned.
                        // Example: return [3, 1, 2]
solution.reset();       // Resets the array back to its original configuration [1,2,3].
Return [1, 2, 3]
solution.shuffle();     // Returns the random shuffling of array [1,2,3]. Example: return [1,
3, 2]
```

```
import java.util.Random;

public class Solution {
    private int[] original;
    private Random rand;

    public Solution(int[] nums) {
        this.original = nums.clone();
        this.rand = new Random();
    }

    public int[] reset() {
        return original;
    }
```

```

public int[] shuffle() {
    int[] shuffled = original.clone();
    for (int i = 0; i < shuffled.length; i++) {
        int j = rand.nextInt(i + 1);
        // Swap elements i and j
        int temp = shuffled[i];
        shuffled[i] = shuffled[j];
        shuffled[j] = temp;
    }
    return shuffled;
}

/**
 * Your Solution object will be instantiated and called as such:
 * Solution obj = new Solution(nums);
 * int[] param_1 = obj.reset();
 * int[] param_2 = obj.shuffle();
 */

```

## 24. Snake And Ladders

- Flatten the board into a 1D array by traversing from the top row to the bottom, alternating the direction (left-to-right or right-to-left) after each row.
- Initialize BFS by marking the first cell as visited and adding it to the queue.
- While traversing, if the current index equals  $n * n - 1$ , return the current number of moves.
- For each position, iterate over possible dice rolls from 1 to 6. For each roll:
- Calculate  $next = current + dice$ .
- Determine the destination using:  $destination = flattenedBoard[next] == -1 ? next : flattenedBoard[next] - 1$ .
- If the destination hasn't been visited, mark it as visited, add it to the queue, and increment the move count.

**Example 1:**



**Input:** board = [[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1], [-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]

**Output:** 4

**Explanation:**

In the beginning, you start at square 1 (at row 5, column 0).

You decide to move to square 2 and must take the ladder to square 15.

You then decide to move to square 17 and must take the snake to square 13.

You then decide to move to square 14 and must take the ladder to square 35.

You then decide to move to square 36, ending the game.

This is the lowest possible number of moves to reach the last square, so return 4.

**Example 2:**

**Input:** board = [[-1,-1],[-1,3]]

**Output:** 1

```
import java.util.ArrayDeque;
import java.util.Queue;

public class Solution {
    public int snakesAndLadders(int[][] board) {
        int n = board.length;
        int[] flattenedBoard = new int[n * n];
```

```

int index = 0;
boolean leftToRight = true;
for (int i = n - 1; i >= 0; i--) {
    if (leftToRight) {
        for (int j = 0; j < n; j++) {
            flattenedBoard[index++] = board[i][j];
        }
    } else {
        for (int j = n - 1; j >= 0; j--) {
            flattenedBoard[index++] = board[i][j];
        }
    }
    leftToRight = !leftToRight;
}

Queue<Integer> queue = new ArrayDeque<>();
boolean[] visited = new boolean[n * n];
queue.offer(0);
visited[0] = true;
int moves = 0;

while (!queue.isEmpty()) {
    int size = queue.size();
    for (int i = 0; i < size; i++) {
        int current = queue.poll();

        if (current == n * n - 1) {
            return moves;
        }

        for (int dice = 1; dice <= 6; dice++) {
            int next = current + dice;
            if (next >= n * n) break;
            int destination = flattenedBoard[next] == -1 ? next : flattenedBoard[next];
            if (!visited[destination]) {

```

```

        visited[destination] = true;
        queue.offer(destination);
    }
}
moves++;
}

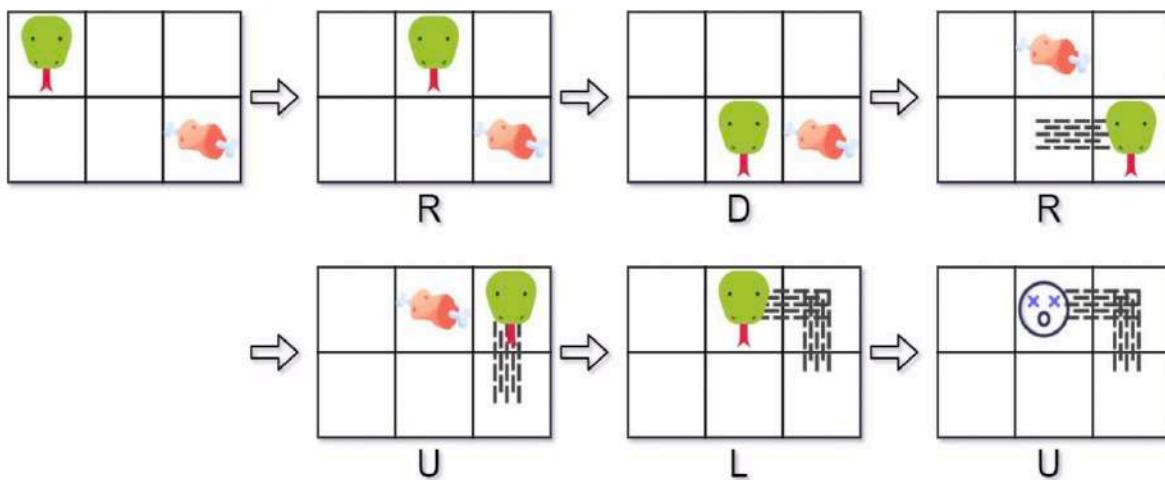
return -1;
}
}

```

---

## 25. Design Snake Game

- Use Queue<int[]> foodQueue to track the cells where food is placed
- Deque<int[]> snake to track the snake moments
- Set<String> bodySet to track the visited cells
- SnakeGame(int width, int height, int[][] food) Initializes the object with a screen of size `height x width` and the positions of the `food`.
- int move(String direction) Returns the score of the game after applying one `direction` move by the snake. If the game is over, return `-1`.



**Input**

```
["SnakeGame", "move", "move", "move", "move", "move", "move"]
[[3, 2, [[1, 2], [0, 1]]], ["R"], ["D"], ["R"], ["U"], ["L"], ["U"]]
```

**Output**

```
[null, 0, 0, 1, 1, 2, -1]
```

**Explanation**

```
SnakeGame snakeGame = new SnakeGame(3, 2, [[1, 2], [0, 1]]);
snakeGame.move("R"); // return 0
snakeGame.move("D"); // return 0
snakeGame.move("R"); // return 1, snake eats the first piece of food. The second piece
snakeGame.move("U"); // return 1
snakeGame.move("L"); // return 2, snake eats the second food. No more food appears.
snakeGame.move("U"); // return -1, game over because snake collides with border
```

```
class SnakeGame {
    private final int[][] DIRECTIONS = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // U, D, L, R
    private final int rows, cols;
    private final Queue<int[]> foodQueue;
    private final Deque<int[]> snake;
    private final Set<String> bodySet;

    public SnakeGame(int width, int height, int[][] food) {
        this.rows = height;
        this.cols = width;
        this.foodQueue = new LinkedList<>();
```

```

        for (int[] f : food) {
            foodQueue.offer(f);
        }
        this.snake = new LinkedList<>();
        this.snake.offerFirst(new int[]{0, 0}); // head starts at (0, 0)
        this.bodySet = new HashSet<>();
        this.bodySet.add("0,0");
    }

    public int move(String direction) {
        int[] currentHead = snake.peekFirst();
        int dir = getDirectionIndex(direction);
        int newRow = currentHead[0] + DIRECTIONS[dir][0];
        int newCol = currentHead[1] + DIRECTIONS[dir][1];

        if (newRow < 0 || newRow >= rows || newCol < 0 || newCol >= cols) {
            return -1;
        }

        int[] tail = snake.pollLast();
        bodySet.remove(tail[0] + "," + tail[1]);

        String newHeadPos = newRow + "," + newCol;
        if (bodySet.contains(newHeadPos)) {
            return -1;
        }

        snake.offerFirst(new int[]{newRow, newCol});
        bodySet.add(newHeadPos);

        if (!foodQueue.isEmpty() && foodQueue.peek()[0] == newRow && foodQueue.peek()[1] == newCol) {
            foodQueue.poll(); // eat
            snake.offerLast(tail); // keep the tail
            bodySet.add(tail[0] + "," + tail[1]); // restore tail
        }
    }
}

```

```

        return snake.size() - 1;
    }

private int getDirectionIndex(String direction) {
    switch (direction) {
        case "U": return 0;
        case "D": return 1;
        case "L": return 2;
        default: return 3; // "R"
    }
}

```

---

## 26. Design Tic-Tac-Toe

- Instead of storing the board, we maintain counters:
  - `rows[i]` = sum of moves in row `i`
  - `cols[j]` = sum of moves in column `j`
  - `diag` / `antiDiag` = sum on diagonals
- Player 1 adds `+1`, Player 2 adds `-1`
- If any counter hits `n` or `-n`, that player wins

Implement the `TicTacToe` class:

- `TicTacToe(int n)` Initializes the object the size of the board `n`.
- `int move(int row, int col, int player)` Indicates that the player with id `player` plays at the cell `(row, col)` of the board. The move is guaranteed to be a valid move, and the two players alternate in making moves. Return
  - `0` if there is **no winner** after the move,
  - `1` if **player 1** is the winner after the move, or
  - `2` if **player 2** is the winner after the move.

**Input**

```
["TicTacToe", "move", "move", "move", "move", "move", "move"]
[[3], [0, 0, 1], [0, 2, 2], [2, 2, 1], [1, 1, 2], [2, 0, 1], [1, 0, 2], [2, 1, 1]]
```

**Output**

```
[null, 0, 0, 0, 0, 0, 0, 1]
```

**Explanation**

```
TicTacToe ticTacToe = new TicTacToe(3);
```

Assume that player 1 is "X" and player 2 is "O" in the board.

```
ticTacToe.move(0, 0, 1); // return 0 (no one wins)
```

```
|X| | |
```

```
| | | | // Player 1 makes a move at (0, 0).
```

```
| | | |
```

```
ticTacToe.move(0, 2, 2); // return 0 (no one wins)
```

```
|X| |0|
```

```
| | | | // Player 2 makes a move at (0, 2).
```

```
| | | |
```

```
ticTacToe.move(2, 2, 1); // return 0 (no one wins)
```

```
|X| |0|
```

```
| | | | // Player 1 makes a move at (2, 2).
```

```
| | |X|
```

```
ticTacToe.move(1, 1, 2); // return 0 (no one wins)
```

```
|X| |0|
```

```
| |0| | // Player 2 makes a move at (1, 1).
```

```
| | |X|
```

```
ticTacToe.move(2, 0, 1); // return 0 (no one wins)
```

```
|X| |0|
```

```
| |0| | // Player 1 makes a move at (2, 0).
```

```
|X| |X|
```

```
ticTacToe.move(1, 0, 2); // return 0 (no one wins)
```

```
|X| |0|
```

```
|0|0| | // Player 2 makes a move at (1, 0).
```

```
|X| |X|
```

```
class TicTacToe {  
    private int[] rows;  
    private int[] cols;  
    private int diag;
```

```

private int antiDiag;
private int n;

public TicTacToe(int n) {
    this.n = n;
    rows = new int[n];
    cols = new int[n];
    diag = 0;
    antiDiag = 0;
}

public int move(int row, int col, int player) {
    int mark = (player == 1) ? 1 : -1;

    rows[row] += mark;
    cols[col] += mark;

    if (row == col) {
        diag += mark;
    }

    if (row + col == n - 1) {
        antiDiag += mark;
    }

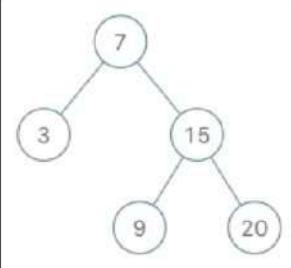
    if (Math.abs(rows[row]) == n ||
        Math.abs(cols[col]) == n ||
        Math.abs(diag) == n ||
        Math.abs(antiDiag) == n) {
        return player;
    }

    return 0; // No winner yet
}
}

```

---

27. Binary Search Tree Iterator



**Input**

```
[["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next"]
[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], []]]
```

**Output**

```
[null, 3, 7, true, 9, true, 15, true, 20, false]
```

**Explanation**

```
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);
bSTIterator.next();    // return 3
bSTIterator.next();    // return 7
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 9
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 15
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 20
bSTIterator.hasNext(); // return False
```

```
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int x) { val = x; }
}
```

```
class BSTIterator {
    private List<Integer> inorder;
    private int index;
```

```

public BSTIterator(TreeNode root) {
    inorder = new ArrayList<>();
    index = 0;
    inOrderTraversal(root);
}

private void inOrderTraversal(TreeNode node) {
    if (node == null) return;
    inOrderTraversal(node.left);
    inorder.add(node.val);
    inOrderTraversal(node.right);
}

public int next() {
    return inorder.get(index++);
}

public boolean hasNext() {
    return index < inorder.size();
}

```

---

## 28. [Implement Trie \(Prefix Tree\)](#)

- Use TrieNode of fields `TrieNode[] children` and `boolean isEndOfWord`;
- Traverse through the children and update the fields

```
Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]
```

#### **Explanation**

```
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");    // return True
trie.search("app");      // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");      // return True
```

```
class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;

    TrieNode() {
        children = new TrieNode[26]; // For lowercase 'a' to 'z'
        isEndOfWord = false;
    }
}

public class Trie {
    private final TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode curr = root;
        for (char c : word.toCharArray()) {
            int idx = c - 'a';
            if (curr.children[idx] == null) {
```

```

        curr.children[idx] = new TrieNode();
    }
    curr = curr.children[idx];
}
curr.isEndOfWord = true;
}

public boolean search(String word) {
    TrieNode node = searchPrefix(word);
    return node != null && node.isEndOfWord;
}

public boolean startsWith(String prefix) {
    return searchPrefix(prefix) != null;
}

private TrieNode searchPrefix(String prefix) {
    TrieNode curr = root;
    for (char c : prefix.toCharArray()) {
        int idx = c - 'a';
        if (curr.children[idx] == null) {
            return null;
        }
        curr = curr.children[idx];
    }
    return curr;
}

/**
 * Your Trie object will be instantiated and called as such:
 * Trie obj = new Trie();
 * obj.insert(word);
 * boolean param_2 = obj.search(word);

```

```
* boolean param_3 = obj.startsWith(prefix);
*/
```

## 29. Design Add and Search Words Data Structure

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the `WordDictionary` class:

- `WordDictionary()` Initializes the object.
- `void addWord(word)` Adds `word` to the data structure, it can be matched later.
- `bool search(word)` Returns `true` if there is any string in the data structure that matches `word` or `false` otherwise. `word` may contain dots `'.'` where dots can be matched with any letter.

**Example:**

**Input**

```
["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
```

**Output**

```
[null,null,null,null,false,true,true,true]
```

**Explanation**

```
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // return False
wordDictionary.search("bad"); // return True
wordDictionary.search(".ad"); // return True
wordDictionary.search("b.."); // return True
```

```
public class WordDictionary {
```

```
    private static class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isEndOfWord = false;
    }
```

```

private final TrieNode root;

public WordDictionary() {
    root = new TrieNode();
}

public void addWord(String word) {
    TrieNode currentNode = root;
    for (char character : word.toCharArray()) {
        int index = character - 'a';
        if (currentNode.children[index] == null) {
            currentNode.children[index] = new TrieNode();
        }
        currentNode = currentNode.children[index];
    }
    currentNode.isEndOfWord = true;
}

public boolean search(String word) {
    return searchFromNode(word, 0, root);
}

private boolean searchFromNode(String word, int wordIndex, TrieNode currentNode) {
    if (currentNode == null) return false;

    if (wordIndex == word.length()) {
        return currentNode.isEndOfWord;
    }

    char currentChar = word.charAt(wordIndex);

    if (currentChar == '!') {
        // Try all possible children for wildcard
        for (TrieNode childNode : currentNode.children) {
            if (childNode != null && searchFromNode(word, wordIndex + 1, childNode))
                return true;
        }
    } else {
        int index = currentChar - 'a';
        if (currentNode.children[index] == null)
            return false;
        return searchFromNode(word, wordIndex + 1, currentNode.children[index]);
    }
}

```

```

        }
    }
    return false;
} else {
    int index = currentChar - 'a';
    return searchFromNode(word, wordIndex + 1, currentNode.children[index]
}
}

// Driver code
public static void main(String[] args) {
    WordDictionary dictionary = new WordDictionary();

    dictionary.addWord("bad");
    dictionary.addWord("dad");
    dictionary.addWord("mad");

    System.out.println(dictionary.search("pad")); // false
    System.out.println(dictionary.search("bad")); // true
    System.out.println(dictionary.search(".ad")); // true
    System.out.println(dictionary.search("b..")); // true
    System.out.println(dictionary.search("b.d")); // true
    System.out.println(dictionary.search(..d)); // true
    System.out.println(dictionary.search(...)); // true
    System.out.println(dictionary.search("....")); // false
}
}

/**
 * Your WordDictionary object will be instantiated and called as such:
 * WordDictionary obj = new WordDictionary();
 * obj.addWord(word);
 * boolean param_2 = obj.search(word);
 */

```

---

### 30. Design Twitter

Implement the `Twitter` class:

- `Twitter()` Initializes your twitter object.
- `void postTweet(int userId, int tweetId)` Composes a new tweet with ID `tweetId` by the user `userId`. Each call to this function will be made with a unique `tweetId`.
- `List<Integer> getNewsFeed(int userId)` Retrieves the `10` most recent tweet IDs in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user themselves. Tweets must be ordered from most recent to least recent.
- `void follow(int followerId, int followeeId)` The user with ID `followerId` started following the user with ID `followeeId`.
- `void unfollow(int followerId, int followeeId)` The user with ID `followerId` started unfollowing the user with ID `followeeId`.

**Example 1:**

**Input**

```
["Twitter", "postTweet", "getNewsFeed", "follow", "postTweet", "getNewsFeed", "unfollow",
 "getNewsFeed"]
[], [1, 5], [1], [1, 2], [2, 6], [1], [1, 2], [1]
```

**Output**

```
[null, null, [5], null, null, [6, 5], null, [5]]
```

**Explanation**

```
Twitter twitter = new Twitter();
twitter.postTweet(1, 5); // User 1 posts a new tweet (id = 5).
twitter.getNewsFeed(1); // User 1's news feed should return a list with 1 tweet id -> [5].
return [5]
twitter.follow(1, 2); // User 1 follows user 2.
twitter.postTweet(2, 6); // User 2 posts a new tweet (id = 6).
twitter.getNewsFeed(1); // User 1's news feed should return a list with 2 tweet ids -> [6,
5]. Tweet id 6 should precede tweet id 5 because it is posted after tweet id 5.
twitter.unfollow(1, 2); // User 1 unfollows user 2.
twitter.getNewsFeed(1); // User 1's news feed should return a list with 1 tweet id -> [5],
since user 1 is no longer following user 2.
```

```
class Twitter {
    private static int timestamp = 0;

    private static class Tweet {
        int time;
```

```

int id;
public Tweet(int id, int time) {
    this.id = id;
    this.time = time;
}
}

private Map<Integer, Set<Integer>> followMap;
private Map<Integer, List<Tweet>> tweets;

public Twitter() {
    followMap = new HashMap<>();
    tweets = new HashMap<>();
}

public void postTweet(int userId, int tweetId) {
    followMap.putIfAbsent(userId, new HashSet<>());
    followMap.get(userId).add(userId); // follow self

    tweets.putIfAbsent(userId, new ArrayList<>());
    tweets.get(userId).add(new Tweet(tweetId, timestamp++));
}

public List<Integer> getNewsFeed(int userId) {
    List<Integer> result = new ArrayList<>();
    if (!followMap.containsKey(userId)) return result;

    PriorityQueue<Tweet> maxHeap = new PriorityQueue<>((a, b) → b.time - a.time);

    for (int followeeId : followMap.get(userId)) {
        List<Tweet> userTweets = tweets.getOrDefault(followeeId, new ArrayList<>());
        for (int i = userTweets.size() - 1; i >= Math.max(0, userTweets.size() - 10); i--) {
            maxHeap.offer(userTweets.get(i));
        }
    }
}

```

```

        while (!maxHeap.isEmpty() && result.size() < 10) {
            result.add(maxHeap.poll().id);
        }

        return result;
    }

    public void follow(int followerId, int followeeId) {
        followMap.putIfAbsent(followerId, new HashSet<>());
        followMap.get(followerId).add(followeeId);
    }

    public void unfollow(int followerId, int followeeId) {
        if (followerId == followeeId) return;
        followMap.getOrDefault(followerId, new HashSet<>()).remove(followeeId);
    }
}

/**
 * Your Twitter object will be instantiated and called as such:
 * Twitter obj = new Twitter();
 * obj.postTweet(userId,tweetId);
 * List<Integer> param_2 = obj.getNewsFeed(userId);
 * obj.follow(followerId,followeeId);
 * obj.unfollow(followerId,followeeId);
 */

```

---

31. [Logger Rate Limiter](#)

Design a logger system that receives a stream of messages along with their timestamps. Each **unique** message should only be printed **at most every 10 seconds** (i.e. a message printed at timestamp `t` will prevent other identical messages from being printed until timestamp `t + 10`).

All messages will come in chronological order. Several messages may arrive at the same timestamp.

Implement the `Logger` class:

- `Logger()` Initializes the `logger` object.
- `bool shouldPrintMessage(int timestamp, string message)` Returns `true` if the `message` should be printed in the given `timestamp`, otherwise returns `false`.

### Example 1:

```
Input
["Logger", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldP
[], [1, "foo"], [2, "bar"], [3, "foo"], [8, "bar"], [10, "foo"], [11, "foo"]]
Output
[null, true, true, false, false, false, true]

Explanation
Logger logger = new Logger();
logger.shouldPrintMessage(1, "foo"); // return true, next allowed timestamp for "foo"
logger.shouldPrintMessage(2, "bar"); // return true, next allowed timestamp for "bar"
logger.shouldPrintMessage(3, "foo"); // 3 < 11, return false
logger.shouldPrintMessage(8, "bar"); // 8 < 12, return false
logger.shouldPrintMessage(10, "foo"); // 10 < 11, return false
logger.shouldPrintMessage(11, "foo"); // 11 >= 11, return true, next allowed timestamp
```

```
class Logger {

    private Map<String, Integer> limiter;

    /** Initialize your data structure here. */
    public Logger() {
```

```
limiter = new HashMap<>();
}

/**
 Returns true if the message should be printed in the given timestamp, otherwise
 false. If this method returns false, the message will not be printed. The times
 seconds granularity.
*/
public boolean shouldPrintMessage(int timestamp, String message) {
    int t = limiter.getOrDefault(message, 0);
    if (t > timestamp) {
        return false;
    }
    limiter.put(message, timestamp + 10);
    return true;
}
}
```

---

32. [Design Hit Counter](#)

Design a hit counter which counts the number of hits received in the past 5 minutes.

Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.

It is possible that several hits arrive roughly at the same time.

**Example:**

```
HitCounter counter = new HitCounter();

// hit at timestamp 1.
counter.hit(1);

// hit at timestamp 2.
counter.hit(2);

// hit at timestamp 3.
counter.hit(3);

// get hits at timestamp 4, should return 3.
counter.getHits(4);

// hit at timestamp 300.
counter.hit(300);

// get hits at timestamp 300, should return 4.
counter.getHits(300);

// get hits at timestamp 301, should return 3.
counter.getHits(301);
```

## // Using Circular Arrays (Optimized for Constant Window Size)

```
class HitCounter {

    private final int[] times;
    private final int[] hits;

    public HitCounter() {
        times = new int[300];
        hits = new int[300];
    }

    public void hit(int timestamp) {
        int idx = timestamp % 300;
        if (times[idx] != timestamp) {
            times[idx] = timestamp;
            hits[idx] = 1;
        } else {
            hits[idx]++;
        }
    }

    public int getHits(int timestamp) {
        int start = timestamp % 300;
        int end = (timestamp + 5) % 300;
        int sum = 0;
        for (int i = start; i < end; i++) {
            sum += hits[i];
        }
        return sum;
    }
}
```

```

        }
    }

public int getHits(int timestamp) {
    int total = 0;
    for (int i = 0; i < 300; i++) {
        if (timestamp - times[i] < 300) {
            total += hits[i];
        }
    }
    return total;
}
}

```

```

class HitCounter {

    private Deque<int[]> queue; // each element: [timestamp, count]
    private int totalHits;

    public HitCounter() {
        queue = new LinkedList<>();
        totalHits = 0;
    }

    public void hit(int timestamp) {
        if (!queue.isEmpty() && queue.peekLast()[0] == timestamp) {
            queue.peekLast()[1]++;
        } else {
            queue.offerLast(new int[]{timestamp, 1});
        }
        totalHits++;
    }

    public int getHits(int timestamp) {
        while (!queue.isEmpty() && timestamp - queue.peekFirst()[0] >= 300) {

```

```

        totalHits -= queue.pollFirst()[1];
    }
    return totalHits;
}
}

```

### 33. Finding MK Average

Implement the `MKAverage` class:

- `MKAverage(int m, int k)` Initializes the `MKAverage` object with an empty stream and the two integers `m` and `k`.
- `void addElement(int num)` Inserts a new element `num` into the stream.
- `int calculateMKAverage()` Calculates and returns the `MKAverage` for the current stream rounded down to the nearest integer.

**Example 1:**

**Input**

```
["MKAverage", "addElement", "addElement", "calculateMKAverage", "addElement",
"calculateMKAverage", "addElement", "addElement", "addElement", "calculateMKAverage"]
[[3, 1], [3], [1], [], [10], [], [5], [5], [5], []]
```

**Output**

```
[null, null, null, -1, null, 3, null, null, null, 5]
```

**Explanation**

```
MKAverage obj = new MKAverage(3, 1);
obj.addElement(3);           // current elements are [3]
obj.addElement(1);           // current elements are [3,1]
obj.calculateMKAverage();    // return -1, because m = 3 and only 2 elements exist.
obj.addElement(10);          // current elements are [3,1,10]
obj.calculateMKAverage();    // The last 3 elements are [3,1,10].
                           // After removing smallest and largest 1 element the container will be [3].
                           // The average of [3] equals 3/1 = 3, return 3
obj.addElement(5);          // current elements are [3,1,10,5]
obj.addElement(5);          // current elements are [3,1,10,5,5]
obj.addElement(5);          // current elements are [3,1,10,5,5,5]
obj.calculateMKAverage();    // The last 3 elements are [5,5,5].
                           // After removing smallest and largest 1 element the container will be [5].
                           // The average of [5] equals 5/1 = 5, return 5
```

### • Three Sorted Buckets:

- `smallestK` : holds the smallest `k` elements.

- `largestK` : holds the largest  $k$  elements.
- `middle` : holds the rest (used to compute the average).
- **Queue ( `stream` ):**
  - Maintains the sliding window of size  $m$ .
- **TreeMaps:**
  - Used for efficient access to smallest/largest keys and handling duplicates via value counts.
- **Three Sorted Buckets:**
  - `smallestK` : holds the smallest  $k$  elements.
  - `largestK` : holds the largest  $k$  elements.
  - `middle` : holds the rest (used to compute the average).
- **Queue ( `stream` ):**
  - Maintains the sliding window of size  $m$ .
- **TreeMaps:**
  - Used for efficient access to smallest/largest keys and handling duplicates via value counts.

```

class MKAverage {
    private final int windowSize;
    private final int trimCount;
    private final Queue<Integer> stream = new ArrayDeque<>();
    private final TreeMap<Integer, Integer> smallestK = new TreeMap<>();
    private final TreeMap<Integer, Integer> middle = new TreeMap<>();
    private final TreeMap<Integer, Integer> largestK = new TreeMap<>();
    private int smallestKSize = 0;
    private int largestKSize = 0;
    private long middleSum = 0;

    public MKAverage(int m, int k) {
        this.windowSize = m;
    }
}

```

```

        this.trimCount = k;
    }

    public void addElement(int num) {
        stream.offer(num);
        addToMap(middle, num);
        middleSum += num;

        if (stream.size() > windowHeight) {
            int removed = stream.poll();
            if (largestK.containsKey(removed)) {
                removeFromMap(largestK, removed);
                largestKSize--;
            } else if (middle.containsKey(removed)) {
                removeFromMap(middle, removed);
                middleSum -= removed;
            } else {
                removeFromMap(smallestK, removed);
                smallestKSize--;
            }
        }
    }

    // Fill up largestK from middle if it has less than k elements
    while (!middle.isEmpty() && largestKSize < trimCount) {
        int maxMid = middle.lastKey();
        middleSum -= maxMid;
        addToMap(largestK, removeFromMap(middle, maxMid));
        largestKSize++;
    }

    // Rebalance between middle and largestK
    while (!middle.isEmpty() && !largestK.isEmpty() && middle.lastKey() > largestK.firstKey()) {
        int maxMid = middle.lastKey();
        int minTop = largestK.firstKey();
        middleSum -= maxMid;
        middleSum += minTop;
    }
}

```

```

        addToMap(largestK, removeFromMap(middle, maxMid));
        addToMap(middle, removeFromMap(largestK, minTop));
    }

    // Fill up smallestK from middle if it has less than k elements
    while (!middle.isEmpty() && smallestKSize < trimCount) {
        int minMid = middle.firstKey();
        middleSum -= minMid;
        addToMap(smallestK, removeFromMap(middle, minMid));
        smallestKSize++;
    }

    // Rebalance between middle and smallestK
    while (!middle.isEmpty() && !smallestK.isEmpty() && middle.firstKey() < sma
        int minMid = middle.firstKey();
        int maxBot = smallestK.lastKey();
        middleSum -= minMid;
        middleSum += maxBot;
        addToMap(smallestK, removeFromMap(middle, minMid));
        addToMap(middle, removeFromMap(smallestK, maxBot));
    }
}

public int calculateMKAverage() {
    if (stream.size() < windowSize)
        return -1;
    return (int) (middleSum / (windowSize - 2 * trimCount));
}

private void addToMap(TreeMap<Integer, Integer> map, int num) {
    map.merge(num, 1, Integer::sum);
}

private int removeFromMap(TreeMap<Integer, Integer> map, int num) {
    map.put(num, map.get(num) - 1);
    if (map.get(num) == 0) {

```

```

        map.remove(num);
    }
    return num;
}
}

```

---

### 34. Design Browser History

```

Input:
["BrowserHistory","visit","visit","visit","back","back","forward","visit","forward","back","back"]
[[["leetcode.com"], ["google.com"], ["facebook.com"], ["youtube.com"], [1], [1], [1],
["linkedin.com"], [2], [2], [7]]]
Output:
[null,null,null,null,"facebook.com","google.com","facebook.com",null,"linkedin.com","google.
com","leetcode.com"]

Explanation:
BrowserHistory browserHistory = new BrowserHistory("leetcode.com");
browserHistory.visit("google.com");           // You are in "leetcode.com". Visit "google.com"
browserHistory.visit("facebook.com");         // You are in "google.com". Visit "facebook.com"
browserHistory.visit("youtube.com");          // You are in "facebook.com". Visit "youtube.com"
browserHistory.back(1);                      // You are in "youtube.com", move back to
"facebook.com" return "facebook.com"
browserHistory.back(1);                      // You are in "facebook.com", move back to
"google.com" return "google.com"
browserHistory.forward(1);                   // You are in "google.com", move forward to
"facebook.com" return "facebook.com"
browserHistory.visit("linkedin.com");         // You are in "facebook.com". Visit "linkedin.com"
browserHistory.forward(2);                   // You are in "linkedin.com", you cannot move
forward any steps.
browserHistory.back(2);                     // You are in "linkedin.com", move back two steps
to "facebook.com" then to "google.com". return "google.com"
browserHistory.back(7);                     // You are in "google.com", you can move back only
one step to "leetcode.com". return "leetcode.com"

```

```

class BrowserHistory {
    private Deque<String> backStack;
    private Deque<String> forwardStack;
    private String currentPage;

    public BrowserHistory(String homepage) {

```

```

currentPage = homepage;
backStack = new ArrayDeque<>();
forwardStack = new ArrayDeque<>();
}

public void visit(String url) {
    backStack.push(currentPage);
    currentPage = url;
    forwardStack.clear(); // visiting a new page resets forward history
}

public String back(int steps) {
    while (steps-- > 0 && !backStack.isEmpty()) {
        forwardStack.push(currentPage);
        currentPage = backStack.pop();
    }
    return currentPage;
}

public String forward(int steps) {
    while (steps-- > 0 && !forwardStack.isEmpty()) {
        backStack.push(currentPage);
        currentPage = forwardStack.pop();
    }
    return currentPage;
}

/**
 * Your BrowserHistory object will be instantiated and called as such:
 * BrowserHistory obj = new BrowserHistory(homepage);
 * obj.visit(url);
 * String param_2 = obj.back(steps);
 * String param_3 = obj.forward(steps);
 */

```

---

35. Design A Leaderboard

```
class Leaderboard {  
    private Map<Integer, Integer> playerScores = new HashMap<>();  
    private TreeMap<Integer, Integer> scoreFrequencies = new TreeMap<>((a, b)  
        {  
            if (a < b) return -1;  
            else if (a == b) return 0;  
            else return 1;  
        });  
  
    public Leaderboard() {}  
  
    public void addScore(int playerId, int scoreToAdd) {  
        int newTotalScore = playerScores.getOrDefault(playerId, 0) + scoreToAdd;  
  
        // If player already has a score, update the old score's frequency  
        if (playerScores.containsKey(playerId)) {  
            int oldScore = playerScores.get(playerId);  
            scoreFrequencies.merge(oldScore, -1, Integer::sum);  
            if (scoreFrequencies.get(oldScore) == 0) {  
                scoreFrequencies.remove(oldScore);  
            }  
        }  
  
        // Update player's score  
        playerScores.put(playerId, newTotalScore);  
        scoreFrequencies.merge(newTotalScore, 1, Integer::sum);  
    }  
  
    public int top(int K) {  
        int total = 0;  
        for (Map.Entry<Integer, Integer> entry : scoreFrequencies.entrySet()) {  
            int score = entry.getKey();  
            int frequency = entry.getValue();  
  
            int count = Math.min(K, frequency);  
            total += score * count;  
            K -= count;  
        }  
        return total;  
    }  
}
```

```

        if (K == 0) break;
    }
    return total;
}

public void reset(int playerId) {
    int score = playerScores.remove(playerId);
    scoreFrequencies.merge(score, -1, Integer::sum);
    if (scoreFrequencies.get(score) == 0) {
        scoreFrequencies.remove(score);
    }
}
}

```

### 36. Design File System

**Input:**

```

["FileSystem","createPath","get"]
[[],["/a",1],["/a"]]

```

**Output:**

```
[null,true,1]
```

**Explanation:**

```
FileSystem fileSystem = new FileSystem();
```

```

fileSystem.createPath("/a", 1); // return true
fileSystem.get("/a"); // return 1

```

```

class Trie {
    Map<String, Trie> children = new HashMap<>();
    int value;

    Trie(int value) {

```

```

        this.value = value;
    }

boolean insert(String path, int value) {
    Trie currentNode = this;
    String[] directories = path.split("/");

    for (int i = 1; i < directories.length - 1; ++i) {
        String directory = directories[i];
        if (!currentNode.children.containsKey(directory)) {
            return false;
        }
        currentNode = currentNode.children.get(directory);
    }

    String lastDirectory = directories[directories.length - 1];
    if (currentNode.children.containsKey(lastDirectory)) {
        return false;
    }

    currentNode.children.put(lastDirectory, new Trie(value));
    return true;
}

int search(String path) {
    Trie currentNode = this;
    String[] directories = path.split("/");

    for (int i = 1; i < directories.length; ++i) {
        String directory = directories[i];
        if (!currentNode.children.containsKey(directory)) {
            return -1;
        }
        currentNode = currentNode.children.get(directory);
    }
}

```

```

        return currentNode.value;
    }
}

class FileSystem {
    private Trie rootTrie;

    public FileSystem() {
        this.rootTrie = new Trie(-1);
    }

    public boolean createPath(String path, int value) {
        return rootTrie.insert(path, value);
    }

    public int get(String path) {
        return rootTrie.search(path);
    }
}

```

### 37. Design a File Sharing System

Input:

```

["FileSharing","join","join","join","join","request","request","leave",
"request","leave","join"]
[[4],[[1,2]],[[2,3]],[[4]],[1,3],[2,2],[1],[2,1],[2],[]]

```

Output:

```
[null,1,2,3,[2],[1,2],null,[],null,1]
```

Explanation:

```

FileSharing fileSharing = new FileSharing(4);
// We use the system to share a file of 4 chunks.

```

```
fileSharing.join([1, 2]);
```

```
// A user who has chunks [1,2] joined the system,  
// assign id = 1 to them and return 1.  
  
fileSharing.join([2, 3]);  
// A user who has chunks [2,3] joined the system,  
// assign id = 2 to them and return 2.  
  
fileSharing.join([4]);  
// A user who has chunk [4] joined the system,  
// assign id = 3 to them and return 3.  
  
fileSharing.request(1, 3);  
// The user with id = 1 requested the third file chunk,  
// as only the user with id = 2 has the file, return [2] .  
// Notice that user 1 now has chunks [1,2,3].  
  
fileSharing.request(2, 2);  
// The user with id = 2 requested the second file chunk,  
users with ids [1,2] have this chunk, thus we return [1,2].  
  
fileSharing.leave(1);  
// The user with id = 1 left the system, all the  
file chunks with them are no longer available for other users.  
  
fileSharing.request(2, 1);  
// The user with id = 2 requested the first file chunk,  
no one in the system has this chunk, we return empty list [].  
  
fileSharing.leave(2);  
// The user with id = 2 left the system.  
  
fileSharing.join([]);  
// A user who doesn't have any chunks joined the system,  
// assign id = 1 to them and return 1.  
// Notice that ids 1 and 2 are free and we can reuse them.
```

```

class FileSharing {
    private int totalChunks;
    private int lastAssignedUserId;
    private TreeSet<Integer> availableUserIds; // reused user IDs
    private TreeMap<Integer, Set<Integer>> userToChunksMap;

    public FileSharing(int totalChunks) {
        this.totalChunks = totalChunks;
        this.lastAssignedUserId = 0;
        this.availableUserIds = new TreeSet<>();
        this.userToChunksMap = new TreeMap<>();
    }

    public int join(List<Integer> ownedChunks) {
        int userId;
        if (availableUserIds.isEmpty()) {
            lastAssignedUserId++;
            userId = lastAssignedUserId;
        } else {
            userId = availableUserIds.pollFirst();
        }
        userToChunksMap.put(userId, new HashSet<>(ownedChunks));
        return userId;
    }

    public void leave(int userId) {
        availableUserIds.add(userId);
        userToChunksMap.remove(userId);
    }

    public List<Integer> request(int requesterUserId, int chunkId) {
        if (chunkId < 1 || chunkId > totalChunks) {
            return Collections.emptyList();
        }
    }
}

```

```

List<Integer> ownersWithChunk = new ArrayList<>();
for (Map.Entry<Integer, Set<Integer>> entry : userToChunksMap.entrySet())
    if (entry.getValue().contains(chunkId)) {
        ownersWithChunk.add(entry.getKey());
    }
}

if (!ownersWithChunk.isEmpty()) {
    userToChunksMap
        .computeIfAbsent(requesterUserId, id → new HashSet<>())
        .add(chunkId);
}

return ownersWithChunk;
}
}

```

## 1. How would you scale the system to support millions of users and files?

- **Distributed Storage:** Implement distributed file storage using technologies like Distributed Hash Tables (DHT) for efficient file lookup and distribution.
- **Load Balancing:** Use load balancers to distribute requests evenly across servers, preventing any single server from becoming a bottleneck.
- **Caching:** Implement caching strategies for frequently accessed files to reduce latency and decrease load on the storage system.

## 2. How can the system handle concurrent file uploads and downloads?

- **Locking Mechanisms:** Use locks or synchronization primitives to manage concurrent accesses to the same file chunks, ensuring data consistency.
- **Version Control:** Implement a version control system for files to handle concurrent edits and updates safely.

### **3. How would you improve the system's fault tolerance and ensure data integrity?**

- **Replication:** Store multiple copies of file chunks across different servers or geographical locations to ensure availability in case of server failures.
- **Checksums and Hashing:** Use checksums or hash values for files and chunks to detect and correct data corruption.

### **4. Can the system support real-time file collaboration or editing?**

- **Operational Transformation (OT) or Conflict-Free Replicated Data Types (CRDTs):** For real-time collaboration, implement algorithms like OT or use CRDTs to handle concurrent operations on documents without conflicts.
- **Event Sourcing:** Store changes to files as a series of immutable events to enable collaborative editing and historical version tracking.

### **5. How would you implement user authentication and file access permissions?**

- **Authentication Service:** Integrate an authentication service (e.g., OAuth) to manage user identities and sessions securely.
- **Access Control Lists (ACLs):** Use ACLs or role-based access control (RBAC) to define and enforce file access permissions based on user roles or identities.

### **6. How can the system be extended to support additional media types or large files?**

- **Transcoding:** Implement transcoding services to convert files into different formats or resolutions based on user needs or device capabilities.
- **Chunking and Streaming:** For large files or media, use chunking to break files into smaller, manageable pieces and support streaming to allow users to start accessing content before the entire file is downloaded.

---

38. [Kth Largest Element in a Stream](#)

### Example 1:

#### Input:

```
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
```

#### Output: [null, 4, 5, 5, 8, 8]

#### Explanation:

```
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3); // return 4
kthLargest.add(5); // return 5
kthLargest.add(10); // return 5
kthLargest.add(9); // return 8
kthLargest.add(4); // return 8
```

```
class KthLargest {
    private PriorityQueue<Integer> minHeap;
    private int k;

    public KthLargest(int k, int[] nums) {
        this.k = k;
        minHeap = new PriorityQueue<>();

        // Keep only k largest elements in the heap
        for (int num : nums) {
            add(num);
        }
    }

    public int add(int val) {
        minHeap.offer(val);
    }
}
```

```

        // Remove smallest if size exceeds k
        if (minHeap.size() > k) {
            minHeap.poll();
        }

        // The kth largest is always at the top
        return minHeap.peek();
    }
}

/**
 * Your KthLargest object will be instantiated and called as such:
 * KthLargest obj = new KthLargest(k, nums);
 * int param_1 = obj.add(val);
 */

```

---

### 39. [Find Median from Data Stream](#)

```

Input
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
Output
[null, null, null, 1.5, null, 2.0]

```

#### **Explanation**

```

MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);      // arr = [1]
medianFinder.addNum(2);      // arr = [1, 2]
medianFinder.findMedian();  // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);      // arr[1, 2, 3]
medianFinder.findMedian();  // return 2.0

```

```

class MedianFinder {
    private PriorityQueue<Integer> lowerHalf; // Max-heap
    private PriorityQueue<Integer> upperHalf; // Min-heap

    public MedianFinder() {
        lowerHalf = new PriorityQueue<>(Collections.reverseOrder());
        upperHalf = new PriorityQueue<>();
    }

    public void addNum(int num) {
        lowerHalf.offer(num);
        upperHalf.offer(lowerHalf.poll());

        if (lowerHalf.size() < upperHalf.size()) {
            lowerHalf.offer(upperHalf.poll());
        }
    }

    public double findMedian() {
        if (lowerHalf.size() > upperHalf.size()) {
            return lowerHalf.peek();
        }
        return (lowerHalf.peek() + upperHalf.peek()) / 2.0;
    }

}

/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder obj = new MedianFinder();
 * obj.addNum(num);
 * double param_2 = obj.findMedian();
 */

```

## 40. Product of the Last K Numbers

**Input**  
["ProductOfNumbers","add","add","add","add","add","getProduct","getProduct","getProduct","add","getProduct"]  
[[], [3], [0], [2], [5], [4], [2], [3], [4], [8], [2]]

**Output**  
[null,null,null,null,null,null,20,40,0,null,32]

### Explanation

```
ProductOfNumbers productOfNumbers = new ProductOfNumbers();
productOfNumbers.add(3);           // [3]
productOfNumbers.add(0);          // [3,0]
productOfNumbers.add(2);          // [3,0,2]
productOfNumbers.add(5);          // [3,0,2,5]
productOfNumbers.add(4);          // [3,0,2,5,4]
productOfNumbers.getProduct(2);   // return 20. The product of the last 2 numbers is 5 * 4 =
                                // 20
productOfNumbers.getProduct(3);   // return 40. The product of the last 3 numbers is 2 * 5 * 4
                                // = 40
productOfNumbers.getProduct(4);   // return 0. The product of the last 4 numbers is 0 * 2 * 5
                                // * 4 = 0
productOfNumbers.add(8);          // [3,0,2,5,4,8]
productOfNumbers.getProduct(2);   // return 32. The product of the last 2 numbers is 4 * 8 =
                                // 32
```

```
class ProductOfNumbers {
    private List<Integer> prefixProducts;
    private int lastZeroIndex;

    public ProductOfNumbers() {
        prefixProducts = new ArrayList<>();
        prefixProducts.add(1); // base case for multiplication
        lastZeroIndex = -1;
    }

    public void add(int number) {
        if (number == 0) {
            prefixProducts.add(1);
        } else {
            prefixProducts.add(prefixProducts.get(prefixProducts.size() - 1) * number);
        }
    }

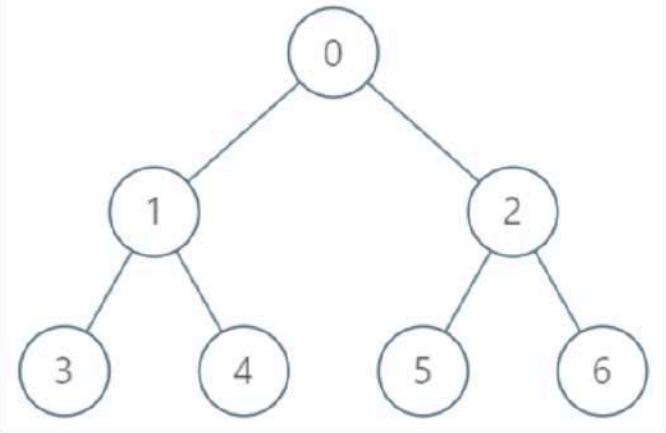
    public int getProduct(int k) {
        if (k < 0 || k >= prefixProducts.size()) {
            return 0;
        }
        if (lastZeroIndex != -1 && lastZeroIndex >= k) {
            return 0;
        }
        return prefixProducts.get(prefixProducts.size() - 1 - k);
    }
}
```

```
        lastZeroIndex = prefixProducts.size() - 1;
    } else {
        int lastProduct = prefixProducts.get(prefixProducts.size() - 1);
        prefixProducts.add(lastProduct * number);
    }
}

public int getProduct(int k) {
    int size = prefixProducts.size();
    if (size - k <= lastZeroIndex) {
        return 0;
    }
    return prefixProducts.get(size - 1) / prefixProducts.get(size - 1 - k);
}
}
```

---

41. [Kth Ancestor of a Tree Node](#) (Binary Lifting)



**Input**

```
["TreeAncestor", "getKthAncestor", "getKthAncestor", "getKthAncestor"]
[[7, [-1, 0, 0, 1, 1, 2, 2]], [3, 1], [5, 2], [6, 3]]
```

**Output**

```
[null, 1, 0, -1]
```

**Explanation**

```
TreeAncestor treeAncestor = new TreeAncestor(7, [-1, 0, 0, 1, 1, 2, 2]);
treeAncestor.getKthAncestor(3, 1); // returns 1 which is the parent of 3
treeAncestor.getKthAncestor(5, 2); // returns 0 which is the grandparent of 5
treeAncestor.getKthAncestor(6, 3); // returns -1 because there is no such ancestor
```

```

class TreeAncestor {
    private int[][] binaryLift;
    private int LOG;

    public TreeAncestor(int n, int[] parent) {
        LOG = 32 - Integer.numberOfLeadingZeros(n);
        binaryLift = new int[n][LOG];

        // Initialize the 2^0 (i.e., direct parent)
        for (int node = 0; node < n; ++node) {
            binaryLift[node][0] = parent[node];
        }

        // Build binary lifting table
        for (int j = 1; j < LOG; ++j) {

```

```

        for (int node = 0; node < n; ++node) {
            int intermediate = binaryLift[node][j - 1];
            binaryLift[node][j] = (intermediate == -1) ? -1 : binaryLift[intermediate][];
        }
    }
}

public int getKthAncestor(int node, int k) {
    for (int j = 0; j < LOG && node != -1; ++j) {
        if ((k & (1 << j)) != 0) {
            node = binaryLift[node][j];
        }
    }
    return node;
}

/***
 * Your TreeAncestor object will be instantiated and called as such:
 * TreeAncestor obj = new TreeAncestor(n, parent);
 * int param_1 = obj.getKthAncestor(node,k);
 */

```

```

/* Java program to calculate Kth ancestor of given node */
import java.util.*;
class GfG {
    // A Binary Tree Node
    static class Node
    {
        int data;
        Node left, right;
    }

    // function to generate array of ancestors

```

```

static void generateArray(Node root, int ancestors[])
{
    // There will be no ancestor of root node
    ancestors[root.data] = -1;

    // level order traversal to
    // generate 1st ancestor
    Queue<Node> q = new LinkedList<Node> ();
    q.add(root);

    while(!q.isEmpty())
    {
        Node temp = q.peek();
        q.remove();

        if (temp.left != null)
        {
            ancestors[temp.left.data] = temp.data;
            q.add(temp.left);
        }

        if (temp.right != null)
        {
            ancestors[temp.right.data] = temp.data;
            q.add(temp.right);
        }
    }

    // function to calculate Kth ancestor
    static int kthAncestor(Node root, int n, int k, int node)
    {
        // create array to store 1st ancestors
        int ancestors[] = new int[n + 1];

        // generate first ancestor array

```

```

generateArray(root,ancestors);

// variable to track record of number of
// ancestors visited
int count = 0;

while (node!= -1)
{
    node = ancestors[node];
    count++;

    if(count==k)
        break;
}

// print Kth ancestor
return node;
}

// Utility function to create a new tree node
static Node newNode(int data)
{
    Node temp = new Node();
    temp.data = data;
    temp.left = null;
    temp.right = null;
    return temp;
}

// Driver program to test above functions
public static void main(String[] args)
{
    // Let us create binary tree shown in above diagram
    Node root = newNode(1);
    root.left = newNode(2);
    root.right = newNode(3);
}

```

```

root.left.left = newNode(4);
root.left.right = newNode(5);

int k = 2;
int node = 5;

// print kth ancestor of given node
System.out.println(kthAncestor(root,5,k,node));
}
}

```

## 42. Flatten 2D Vector

**Input**

```
["Vector2D", "next", "next", "next", "hasNext", "hasNext", "next", "hasNext"]
[[[1, 2], [3], [4]]], [], [], [], [], [], []]
```

**Output**

```
[null, 1, 2, 3, true, true, 4, false]
```

**Explanation**

```

Vector2D vector2D = new Vector2D([[1, 2], [3], [4]]);
vector2D.next();    // return 1
vector2D.next();    // return 2
vector2D.next();    // return 3
vector2D.hasNext(); // return True
vector2D.hasNext(); // return True
vector2D.next();    // return 4
vector2D.hasNext(); // return False

```

```

class Vector2D {
    private int rowIndex;
    private int colIndex;
    private int[][] matrix;

    public Vector2D(int[][] vec) {
        this.matrix = vec;
        this.rowIndex = 0;
    }
}
```

```
    this.colIndex = 0;
}

public int next() {
    advanceToNext();
    return matrix[rowIndex][colIndex++];
}

public boolean hasNext() {
    advanceToNext();
    return rowIndex < matrix.length;
}

private void advanceToNext() {
    // Move to the next row while current row is exhausted or empty
    while (rowIndex < matrix.length && colIndex >= matrix[rowIndex].length) {
        rowIndex++;
        colIndex = 0;
    }
}
```

---

43. [Encode and Decode Strings](#)

## Example 1:

**Input:** dummy\_input = ["Hello", "World"]

**Output:** ["Hello", "World"]

**Explanation:**

Machine 1:

```
Codec encoder = new Codec();
```

```
String msg = encoder.encode(strs);
```

Machine 1 ---msg---> Machine 2

Machine 2:

```
Codec decoder = new Codec();
```

```
String[] strs = decoder.decode(msg);
```

## Example 2:

**Input:** dummy\_input = []

**Output:** []

```
public class Codec {
```

```

// Encodes a list of strings to a single string.
public String encode(List<String> stringList) {
    StringBuilder encodedString = new StringBuilder();
    for (String word : stringList) {
        encodedString.append((char) word.length()).append(word);
    }
    return encodedString.toString();
}

// Decodes a single string to a list of strings.
public List<String> decode(String encodedString) {
    List<String> decodedList = new ArrayList<>();
    int index = 0;
    int totalLength = encodedString.length();

    while (index < totalLength) {
        int wordLength = encodedString.charAt(index++) // retrieves the stored length
        String word = encodedString.substring(index, index + wordLength);
        decodedList.add(word);
        index += wordLength;
    }

    return decodedList;
}
}

```

---

#### 44. Peeking Iterator

```

Input
["PeekingIterator", "next", "peek", "next", "next", "hasNext"]
[[[1, 2, 3]], [], [], [], [], []]
Output
[null, 1, 2, 2, 3, false]

Explanation
PeekingIterator peekingIterator = new PeekingIterator([1, 2, 3]); // [1,2,3]
peekingIterator.next();    // return 1, the pointer moves to the next element [1,2,3].
peekingIterator.peek();   // return 2, the pointer does not move [1,2,3].
peekingIterator.next();    // return 2, the pointer moves to the next element [1,2,3]
peekingIterator.next();    // return 3, the pointer moves to the next element [1,2,3]
peekingIterator.hasNext(); // return False

```

```

import java.util.Iterator;

class PeekingIterator implements Iterator<Integer> {
    private Iterator<Integer> iterator;
    private Integer nextElement;

    public PeekingIterator(Iterator<Integer> iterator) {
        this.iterator = iterator;
        // Pre-fetch the first element
        if (iterator.hasNext()) {
            nextElement = iterator.next();
        }
    }

    // Returns the next element in the iteration without advancing the iterator.
    public Integer peek() {
        return nextElement;
    }

    @Override
    public Integer next() {
        Integer result = nextElement;
        // Advance and pre-fetch the next element
        if (iterator.hasNext()) {

```

```
    nextElement = iterator.next();
} else {
    nextElement = null;
}
return result;
}

@Override
public boolean hasNext() {
    return nextElement != null;
}
}
```

---

45. [Design Excel Sum Formula](#)

`Excel(int H, char W)` : This is the constructor. The inputs represents the height and width of the Excel form. **H** is a positive integer, range from 1 to 26. It represents the height. **W** is a character range from 'A' to 'Z'. It represents that the width is the number of characters from 'A' to **W**. The Excel form content is represented by a height \* width 2D integer array `C`, it should be initialized to zero. You should assume that the first row of `C` starts from 1, and the first column of `C` starts from 'A'.

`void Set(int row, char column, int val)` : Change the value at `C(row, column)` to be `val`.

`int Get(int row, char column)` : Return the value at `C(row, column)`.

`int Sum(int row, char column, List of Strings : numbers)` : This function calculate and set the value at `C(row, column)`, where the value should be the sum of cells represented by numbers. This function return the sum result at `C(row, column)`. This sum formula should exist until this cell is overlapped by another value or another sum formula.

`numbers` is a list of strings that each string represent a cell or a range of cells. If the string represent a single cell, then it has the following format: `ColRow`. For example, "F7" represents the cell at (7, F).

If the string represent a range of cells, then it has the following format:

`ColRow1:ColRow2`. The range will always be a rectangle, and ColRow1 represent the position of the top-left cell, and ColRow2 represents the position of the bottom-right cell.

```
class Excel {  
    int rows;  
    int columns;  
    int[][] cells;  
    Map<String, Map<String, Integer>> cellSumMap;  
    Map<String, Map<String, Integer>> sumCellMap;  
  
    public Excel(int H, char W) {  
        rows = H + 1;
```

```

columns = W - 'A' + 1;
cells = new int[rows][columns];
cellSumMap = new HashMap<String, Map<String, Integer>>();
sumCellMap = new HashMap<String, Map<String, Integer>>();
}

public void set(int r, char c, int v) {
    int prevValue = cells[r][c - 'A'];
    int difference = v - prevValue;
    String cellName = String.valueOf(c) + r;
    Map<String, Integer> cellMap = sumCellMap.getOrDefault(cellName, new H
    sumCellMap.remove(cellName);
    Set<String> cellSet = cellMap.keySet();
    for (String cell : cellSet) {
        Map<String, Integer> sumMap = cellSumMap.getOrDefault(cell, new Has
        sumMap.remove(cellName);
        cellSumMap.put(cell, sumMap);
    }
    updateRelevantCells(cellName, difference);
}

public int get(int r, char c) {
    return cells[r][c - 'A'];
}

public int sum(int r, char c, String[] strs) {
    int sum = 0;
    int prevValue = cells[r][c - 'A'];
    set(r, c, prevValue);
    String sumCellName = String.valueOf(c) + r;
    for (String str : strs) {
        if (str.indexOf(':') >= 0) {
            String[] array = str.split(":");
            String start = array[0], end = array[1];
            char startColumn = start.charAt(0), endColumn = end.charAt(0);
            int startRow = Integer.parseInt(start.substring(1)), endRow = Integer.parseInt(end.substring(1));
            for (int i = startRow; i <= endRow; i++) {
                for (int j = startColumn; j <= endColumn; j++) {
                    sum += cells[i][j];
                }
            }
        }
    }
    return sum;
}

```

```

        for (int i = startRow; i <= endRow; i++) {
            for (char j = startColumn; j <= endColumn; j++) {
                String cellName = String.valueOf(j) + i;
                Map<String, Integer> sumMap = cellSumMap.getOrDefault(cellName, new HashMap<String, Integer>());
                int sumCount = sumMap.getOrDefault(sumCellName, 0) + 1;
                sumMap.put(sumCellName, sumCount);
                cellSumMap.put(cellName, sumMap);

                Map<String, Integer> cellMap = sumCellMap.getOrDefault(sumCellName, new HashMap<String, Integer>());
                int cellCount = cellMap.getOrDefault(cellName, 0) + 1;
                cellMap.put(cellName, cellCount);
                sumCellMap.put(sumCellName, cellMap);

                sum += cells[i][j] - 'A';
            }
        }

    } else {
        Map<String, Integer> sumMap = cellSumMap.getOrDefault(str, new HashMap<String, Integer>());
        int sumCount = sumMap.getOrDefault(sumCellName, 0) + 1;
        sumMap.put(sumCellName, sumCount);
        cellSumMap.put(str, sumMap);

        Map<String, Integer> cellMap = sumCellMap.getOrDefault(sumCellName, new HashMap<String, Integer>());
        int cellCount = cellMap.getOrDefault(str, 0) + 1;
        cellMap.put(str, cellCount);
        sumCellMap.put(sumCellName, cellMap);

        char cellColumn = str.charAt(0);
        int cellRow = Integer.parseInt(str.substring(1));
        sum += cells[cellRow][cellColumn] - 'A';
    }

    int difference = sum - prevValue;
    updateRelevantCells(sumCellName, difference);
    return cells[r][c] - 'A';
}

private void updateRelevantCells(String cellName, int difference) {
    Queue<String> cellQueue = new LinkedList<String>();
}

```

```

Queue<Integer> differenceQueue = new LinkedList<Integer>();
cellQueue.offer(cellName);
differenceQueue.offer(difference);
while (!cellQueue.isEmpty()) {
    String cell = cellQueue.poll();
    int curDifference = differenceQueue.poll();
    char column = cell.charAt(0);
    int row = Integer.parseInt(cell.substring(1));
    cells[row][column - 'A'] += curDifference;
    Map<String, Integer> map = cellSumMap.getOrDefault(cell, new HashMap<String, Integer>());
    Set<String> set = map.keySet();
    for (String nextCell : set) {
        int count = map.get(nextCell);
        cellQueue.offer(nextCell);
        differenceQueue.offer(curDifference * count);
    }
}
}

/**
 * Your Excel object will be instantiated and called as such:
 * Excel obj = new Excel(H, W);
 * obj.set(r,c,v);
 * int param_2 = obj.get(r,c);
 * int param_3 = obj.sum(r,c,strs);
 */

```

---

#### 46. Design Log Storage System

Input

["LogSystem", "put", "put", "put", "retrieve", "retrieve"]

```
[[], [1, "2017:01:01:23:59:59"], [2, "2017:01:01:22:59:59"],  
[3, "2016:01:01:00:00:00"], ["2016:01:01:01:01", "2017:01:01:23:00:00",  
"Year"], ["2016:01:01:01:01", "2017:01:01:23:00:00", "Hour"]]
```

Output

```
[null, null, null, null, [3, 2, 1], [2, 1]]
```

Explanation

```
LogSystem logSystem = new LogSystem();
```

```
logSystem.put(1, "2017:01:01:23:59:59");
```

```
logSystem.put(2, "2017:01:01:22:59:59");
```

```
logSystem.put(3, "2016:01:01:00:00:00");
```

```
// return [3,2,1], because you need to return all logs between 2016 and 2017.
```

```
logSystem.retrieve("2016:01:01:01:01:01", "2017:01:01:23:00:00", "Year");
```

```
// return [2,1], because you need to return all logs between Jan. 1,
```

```
// 2016 01:XX:XX and Jan. 1, 2017 23:XX:XX.
```

```
// Log 3 is not returned because Jan. 1, 2016 00:00:00 comes
```

```
// before the start of the range.
```

```
logSystem.retrieve("2016:01:01:01:01:01", "2017:01:01:23:00:00", "Hour");
```

```
class LogSystem {  
    private List<LogEntry> logs = new ArrayList<>();  
    private Map<String, Integer> granularityLengthMap = new HashMap<>();
```

```
    public LogSystem() {
```

```
        granularityLengthMap.put("Year", 4);
```

```
        granularityLengthMap.put("Month", 7);
```

```
        granularityLengthMap.put("Day", 10);
```

```
        granularityLengthMap.put("Hour", 13);
```

```
        granularityLengthMap.put("Minute", 16);
```

```
        granularityLengthMap.put("Second", 19);
```

```
}
```

```
    public void put(int id, String timestamp) {
```

```

        logs.add(new LogEntry(id, timestamp));
    }

public List<Integer> retrieve(String start, String end, String granularity) {
    List<Integer> result = new ArrayList<>();
    int substringLength = granularityLengthMap.get(granularity);
    String startPrefix = start.substring(0, substringLength);
    String endPrefix = end.substring(0, substringLength);

    for (LogEntry log : logs) {
        String logPrefix = log.timestamp.substring(0, substringLength);
        if (startPrefix.compareTo(logPrefix) <= 0 && logPrefix.compareTo(endPrefix) <= 0)
            result.add(log.id);
    }
}

return result;
}
}

class LogEntry {
    int id;
    String timestamp;

    LogEntry(int id, String timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }
}

```

---

47. [Design Search Autocomplete System](#)

Input

```
["AutocompleteSystem", "input", "input", "input", "input"]
[[["i love you", "island", "ironman", "i love leetcode"],
[5, 3, 2, 2]], ["i"], [" "], ["a"], ["#"]]
```

Output

```
[null, ["i love you", "island", "i love leetcode"],
["i love you", "i love leetcode"], [], []]
```

Explanation

```
AutocompleteSystem obj = new
AutocompleteSystem(["i love you", "island",
"ironman", "i love leetcode"], [5, 3, 2, 2]);
obj.input("i"); // return ["i love you", "island",
// "i love leetcode"]. There are four sentences that have
// prefix "i". Among them, "ironman" and "i love leetcode"
// have same hot degree. Since ' ' has ASCII code 32 and 'r' has
// ASCII code 114, "i love leetcode" should be in front of "ironman".
// Also we only need to output top 3 hot sentences, so "ironman" will be ignored.
obj.input(" "); // return ["i love you", "i love leetcode"].
// There are only two sentences that have prefix "i ".
obj.input("a"); // return []. There are no sentences that have prefix "i a".
obj.input("#"); // return []. The user finished the input, the sentence
// "i a" should be saved as a historical sentence in system.
// And the following input will be counted as a new search.
```

```
class TrieNode {
    TrieNode[] children = new TrieNode[27]; // 26 letters + space
    int frequency;
    String sentence = "";

    // Inserts a sentence into the trie with its frequency
    void insert(String sentence, int count) {
        TrieNode currentNode = this;
        for (char character : sentence.toCharArray()) {
```

```

        int index = character == ' ' ? 26 : character - 'a';
        if (currentNode.children[index] == null) {
            currentNode.children[index] = new TrieNode();
        }
        currentNode = currentNode.children[index];
    }

    currentNode.frequency += count;
    currentNode.sentence = sentence;
}

// Searches and returns the node representing the prefix
TrieNode search(String prefix) {
    TrieNode currentNode = this;
    for (char character : prefix.toCharArray()) {
        int index = character == ' ' ? 26 : character - 'a';
        if (currentNode.children[index] == null) {
            return null;
        }
        currentNode = currentNode.children[index];
    }
    return currentNode;
}

class AutocompleteSystem {
    private TrieNode root = new TrieNode();
    private StringBuilder currentInput = new StringBuilder();

    public AutocompleteSystem(String[] sentences, int[] times) {
        for (int i = 0; i < sentences.length; i++) {
            root.insert(sentences[i], times[i]);
        }
    }

    public List<String> input(char c) {
        List<String> suggestions = new ArrayList<>();

```

```

if (c == '#') {
    root.insert(currentInput.toString(), 1);
    // Save sentence to Trie
    currentInput = new StringBuilder();
    // Reset for next input
    return suggestions;
}

currentInput.append(c);
TrieNode prefixNode = root.search(currentInput.toString());

if (prefixNode == null) return suggestions;

PriorityQueue<TrieNode> topSuggestions = new PriorityQueue<>(
    (a, b) → a.frequency == b.frequency
        ? b.sentence.compareTo(a.sentence)
        // Lexicographically smaller comes first
        : a.frequency - b.frequency
        // Lower frequency comes first
);

collectTopSuggestions(prefixNode, topSuggestions);

while (!topSuggestions.isEmpty()) {
    suggestions.add(0, topSuggestions.poll().sentence);
    // Reverse to highest frequency first
}

return suggestions;
}

private void collectTopSuggestions(TrieNode node, PriorityQueue<TrieNode>
if (node == null) return;

if (node.frequency > 0) {

```

```

        heap.offer(node);
        if (heap.size() > 3) {
            heap.poll(); // Keep only top 3 suggestions
        }
    }

    for (TrieNode child : node.children) {
        collectTopSuggestions(child, heap);
    }
}

```

#### 49. [Implement Magic Dictionary](#)

**Input**  
["MagicDictionary", "buildDict", "search", "search", "search", "search"]
[], [[{"hello", "leetcode"}], [{"hello"}, {"hhillo"}, {"hell"}, {"leetcoded"}]]  
**Output**  
[null, null, false, true, false, false]

**Explanation**  
MagicDictionary magicDictionary = new MagicDictionary();  
magicDictionary.buildDict(["hello", "leetcode"]);  
magicDictionary.search("hello"); // return False  
magicDictionary.search("hhillo"); // We can change the second 'h' to 'e' to match "hello" so we return True  
magicDictionary.search("hell"); // return False  
magicDictionary.search("leetcoded"); // return False

```

class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isEnd = false;
}

public class MagicDictionary {

```

```

private TrieNode root;

public MagicDictionary() {
    root = new TrieNode();
}

public void buildDict(String[] dictionary) {
    for (String word : dictionary) {
        insert(word);
    }
}

private void insert(String word) {
    TrieNode current = root;
    for (char ch : word.toCharArray()) {
        int idx = ch - 'a';
        if (current.children[idx] == null) {
            current.children[idx] = new TrieNode();
        }
        current = current.children[idx];
    }
    current.isEnd = true;
}

public boolean search(String searchWord) {
    return dfs(searchWord.toCharArray(), 0, root, false);
}

private boolean dfs(char[] word, int pos, TrieNode node, boolean modified)
{
    if (node == null) return false;
    if (pos == word.length) return modified && node.isEnd;

    int idx = word[pos] - 'a';

    for (int i = 0; i < 26; i++) {

```

```

        if (node.children[i] == null) continue;
        if (i == idx) {
            if (dfs(word, pos + 1, node.children[i], modified)) return true;
        } else {
            // Try changing this char (only if we haven't already)
            if (!modified && dfs(word, pos + 1, node.children[i], true))
                return true;
        }
    }
    return false;
}

/**
 * Your MagicDictionary object will be instantiated and called as such:
 * MagicDictionary obj = new MagicDictionary();
 * obj.buildDict(dictionary);
 * boolean param_2 = obj.search(searchWord);
 */

```

---

## 50. Design Most Recently Used Queue

Input:

```

["MRUQueue", "fetch", "fetch", "fetch", "fetch"]
[[8], [3], [5], [2], [8]]

```

Output:

```
[null, 3, 6, 2, 2]
```

Explanation:

```

MRUQueue mRUQueue = new MRUQueue(8);
// Initializes the queue to [1,2,3,4,5,6,7,8].

```

```

mRUQueue.fetch(3);
// Moves the 3rd element (3) to the end of the queue to become [1,2,4,5,6,7,8,3]
// and returns it.

mRUQueue.fetch(5);
// Moves the 5th element (6) to the end of the queue to become [1,2,4,5,7,8,3,6]
// and returns it.

mRUQueue.fetch(2);
// Moves the 2nd element (2) to the end of the queue to become [1,4,5,7,8,3,6,2]
// and returns it.

mRUQueue.fetch(8);
// The 8th element (2) is already at the end of the queue so just return it.

```

```

class MRUQueue {
    private int totalSize;
    private int[] queueArray;

    public MRUQueue(int n) {
        totalSize = n;
        queueArray = new int[n];
        for (int i = 0; i < n; i++) {
            queueArray[i] = i + 1;
        }
    }

    public int fetch(int k) {
        int fetchedElement = queueArray[k - 1];
        // Shift elements left from position k
        for (int i = k; i < totalSize; i++) {
            queueArray[i - 1] = queueArray[i];
        }
        // Move the fetched element to the end
        queueArray[totalSize - 1] = fetchedElement;
        return fetchedElement;
    }
}

```

```
    }  
}
```

---

## 50. [Tweet Counts Per Frequency](#)

```
Input  
["TweetCounts","recordTweet","recordTweet","recordTweet","getTweetCountsPerFrequency","getTw  
eetCountsPerFrequency", "recordTweet", "getTweetCountsPerFrequency"]  
[[],["tweet3",0],["tweet3",60],["tweet3",10],["minute","tweet3",0,59],  
["minute","tweet3",0,60],["tweet3",120],["hour","tweet3",0,210]]  
  
Output  
[null,null,null,null,[2],[2,1],null,[4]]  
  
Explanation  
TweetCounts tweetCounts = new TweetCounts();  
tweetCounts.recordTweet("tweet3", 0); // New tweet "tweet3" at time 0  
tweetCounts.recordTweet("tweet3", 60); // New tweet "tweet3" at time 60  
tweetCounts.recordTweet("tweet3", 10); // New tweet "tweet3" at time 10  
tweetCounts.getTweetCountsPerFrequency("minute", "tweet3", 0, 59); // return [2]; chunk [0,59] had 2 tweets  
tweetCounts.getTweetCountsPerFrequency("minute", "tweet3", 0, 60); // return [2,1]; chunk [0,59] had 2 tweets, chunk [60,60] had 1 tweet  
tweetCounts.recordTweet("tweet3", 120); // New tweet "tweet3" at time 120  
tweetCounts.getTweetCountsPerFrequency("hour", "tweet3", 0, 210); // return [4]; chunk [0,210] had 4 tweets
```

```
class TweetCounts {  
    private Map<String, List<Integer>> tweetTimeMap;  
  
    public TweetCounts() {  
        tweetTimeMap = new HashMap<>();  
    }  
  
    public void recordTweet(String tweetName, int time) {  
        tweetTimeMap
```

```

        .computeIfAbsent(tweetName, k → new ArrayList<>())
        .add(time);
    }

    public List<Integer> getTweetCountsPerFrequency(String frequency, String tw
        int intervalInSeconds = getIntervalInSeconds(frequency);
        int numberOfIntervals = (endTime - startTime) / intervalInSeconds + 1;

        List<Integer> tweetCounts = new ArrayList<>(Collections.nCopies(numberOfInt
        List<Integer> recordedTimes = tweetTimeMap.getOrDefault(tweetName, ne

        for (int tweetTime : recordedTimes) {
            if (tweetTime >= startTime && tweetTime <= endTime) {
                int intervalIndex = (tweetTime - startTime) / intervalInSeconds;
                tweetCounts.set(intervalIndex, tweetCounts.get(intervalIndex) + 1);
            }
        }

        return tweetCounts;
    }

    private int getIntervalInSeconds(String frequency) {
        switch (frequency) {
            case "minute":
                return 60;
            case "hour":
                return 3600;
            case "day":
                return 86400;
            default:
                throw new IllegalArgumentException("Invalid frequency: " + frequency)
        }
    }
}

```

```

/**
 * Your TweetCounts object will be instantiated and called as such:
 * TweetCounts obj = new TweetCounts();
 * obj.recordTweet(tweetName,time);
 * List<Integer> param_2 = obj.getTweetCountsPerFrequency(freq,tweetName,st
 */

```

## 51. Design an ATM Machine

- ATM has 5 types of banknotes: [20, 50, 100, 200, 500]
- Implement:
  - `deposit(banknotesCount)` → deposit banknotes
  - `withdraw(amount)` → withdraw banknotes (minimum number of notes, largest denominations first)

```

Input
["ATM", "deposit", "withdraw", "deposit", "withdraw", "withdraw"]
[[], [[0,0,1,2,1]], [600], [[0,1,0,1,1]], [600], [550]]
Output
[null, null, [0,0,1,0,1], null, [-1], [0,1,0,0,1]]

Explanation
ATM atm = new ATM();
atm.deposit([0,0,1,2,1]); // Deposits 1 $100 banknote, 2 $200 banknotes,
                           // and 1 $500 banknote.
atm.withdraw(600);        // Returns [0,0,1,0,1]. The machine uses 1 $100 banknote
                           // and 1 $500 banknote. The banknotes left over in the
                           // machine are [0,0,0,2,0].
atm.deposit([0,1,0,1,1]); // Deposits 1 $50, $200, and $500 banknote.
                           // The banknotes in the machine are now [0,1,0,3,1].
atm.withdraw(600);        // Returns [-1]. The machine will try to use a $500 banknote
                           // and then be unable to complete the remaining $100,
                           // so the withdraw request will be rejected.
                           // Since the request is rejected, the number of banknotes
                           // in the machine is not modified.
atm.withdraw(550);         // Returns [0,1,0,0,1]. The machine uses 1 $50 banknote
                           // and 1 $500 banknote.

```

```

class ATM {
    private final int[] denominations = {20, 50, 100, 200, 500};
    private long[] noteCounts = new long[5]; // stores count of each note type

    public ATM() {
        // Initializes noteCounts with 0 for all denominations
    }

    public void deposit(int[] depositCounts) {
        for (int i = 0; i < 5; i++) {
            noteCounts[i] += depositCounts[i];
        }
    }

    public int[] withdraw(int amount) {
        int[] notesToWithdraw = new int[5];
        int remainingAmount = amount;

        // Try to use largest denominations first (iterate backwards)
        for (int i = 4; i >= 0; i--) {
            int denomination = denominations[i];
            long availableNotes = noteCounts[i];
            int neededNotes = remainingAmount / denomination;
            int usedNotes = (int) Math.min(availableNotes, neededNotes);
            notesToWithdraw[i] = usedNotes;
            remainingAmount -= usedNotes * denomination;
        }

        if (remainingAmount == 0) {
            // Valid withdrawal, update noteCounts
            for (int i = 0; i < 5; i++) {
                noteCounts[i] -= notesToWithdraw[i];
            }
            return notesToWithdraw;
        } else {
    }
}

```

```

        // Cannot fulfill request, return -1s
        return new int[]{-1};
    }
}
}

/**
 * Your ATM object will be instantiated and called as such:
 * ATM obj = new ATM();
 * obj.deposit(banknotesCount);
 * int[] param_2 = obj.withdraw(amount);
 */

```

---

## 52. [Encode and Decode TinyURL](#)

**Input:** url = "https://leetcode.com/problems/design-tinyurl"  
**Output:** "https://leetcode.com/problems/design-tinyurl"

**Explanation:**

```

Solution obj = new Solution();
string tiny = obj.encode(url); // returns the encoded tiny url.
string ans = obj.decode(tiny); // returns the original url after decoding it.

```

```

public class Codec {
    private static final String BASE_HOST = "http://tinyurl.com/";
    private Map<String, String> shortToLongMap = new HashMap<>();
    private Map<String, String> longToShortMap = new HashMap<>();
    private int counter = 0;

    // Encodes a URL to a shortened URL.
    public String encode(String longUrl) {
        if (longToShortMap.containsKey(longUrl)) {
            return BASE_HOST + longToShortMap.get(longUrl);
        }
        String shortUrl = "http://tinyurl.com/" + String.valueOf(counter);
        longToShortMap.put(longUrl, shortUrl);
        shortToLongMap.put(shortUrl, longUrl);
        counter++;
        return shortUrl;
    }

    // Decodes a shortened URL to its original URL.
    public String decode(String shortUrl) {
        return shortToLongMap.get(shortUrl);
    }
}

```

```

    }

    String shortKey = Integer.toString(counter++);
    shortToLongMap.put(shortKey, longUrl);
    longToShortMap.put(longUrl, shortKey);
    return BASE_HOST + shortKey;
}

// Decodes a shortened URL to its original URL.
public String decode(String shortUrl) {
    String shortKey = shortUrl.replace(BASE_HOST, "");
    return shortToLongMap.get(shortKey);
}
}

```

### 53. Design a Food Rating System

**Input**

```
["FoodRatings", "highestRated", "highestRated", "changeRating", "highestRated",
"changeRating", "highestRated"]
[[["kimchi", "miso", "sushi", "moussaka", "ramen", "bulgogi"], ["korean", "japanese",
"japanese", "greek", "japanese", "korean"], [9, 12, 8, 15, 14, 7]], ["korean"],
["japanese"], ["sushi", 16], ["japanese"], ["ramen", 16], ["japanese"]]
```

**Output**

```
[null, "kimchi", "ramen", null, "sushi", null, "ramen"]
```

**Explanation**

```
FoodRatings foodRatings = new FoodRatings(["kimchi", "miso", "sushi", "moussaka", "ramen",
"bulgogi"], ["korean", "japanese", "japanese", "greek", "japanese", "korean"], [9, 12, 8,
15, 14, 7]);
foodRatings.highestRated("korean"); // return "kimchi"
                                    // "kimchi" is the highest rated korean food with a
rating of 9.
foodRatings.highestRated("japanese"); // return "ramen"
                                    // "ramen" is the highest rated japanese food with a
rating of 14.
foodRatings.changeRating("sushi", 16); // "sushi" now has a rating of 16.
foodRatings.highestRated("japanese"); // return "sushi"
                                    // "sushi" is the highest rated japanese food with a
rating of 16.
foodRatings.changeRating("ramen", 16); // "ramen" now has a rating of 16.
foodRatings.highestRated("japanese"); // return "ramen"
                                    // Both "sushi" and "ramen" have a rating of 16.
                                    // However, "ramen" is lexicographically smaller than
"sushi".
```

```

class FoodRatings {
    private Map<String, String> foodToCuisine = new HashMap<>();
    private Map<String, Integer> foodToRating = new HashMap<>();
    private Map<String, TreeSet<Food>> cuisineToFoods = new HashMap<>();

    public FoodRatings(String[] foods, String[] cuisines, int[] ratings) {
        for (int i = 0; i < foods.length; i++) {
            String food = foods[i];
            String cuisine = cuisines[i];
            int rating = ratings[i];

            foodToCuisine.put(food, cuisine);
            foodToRating.put(food, rating);

            cuisineToFoods
                .computeIfAbsent(cuisine, k → new TreeSet<>())
                .add(new Food(food, rating));
        }
    }

    public void changeRating(String food, int newRating) {
        String cuisine = foodToCuisine.get(food);
        int oldRating = foodToRating.get(food);

        TreeSet<Food> foodsSet = cuisineToFoods.get(cuisine);
        foodsSet.remove(new Food(food, oldRating));
        foodsSet.add(new Food(food, newRating));

        foodToRating.put(food, newRating);
    }

    public String highestRated(String cuisine) {
        return cuisineToFoods.get(cuisine).first().name;
    }
}

```

```

private static class Food implements Comparable<Food> {
    String name;
    int rating;

    Food(String name, int rating) {
        this.name = name;
        this.rating = rating;
    }

    public int compareTo(Food other) {
        if (this.rating != other.rating) {
            return Integer.compare(other.rating, this.rating); // Max-heap behavior
        }
        return this.name.compareTo(other.name); // Lexicographical order
    }

    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Food)) return false;
        Food food = (Food) o;
        return name.equals(food.name);
    }

    public int hashCode() {
        return name.hashCode();
    }
}

/**
 * Your FoodRatings object will be instantiated and called as such:
 * FoodRatings obj = new FoodRatings(foods, cuisines, ratings);
 * obj.changeRating(food,newRating);

```

```
* String param_2 = obj.highestRated(cuisine);
*/
```

#### 54. Design Task Manager

**Input:**  
["TaskManager", "add", "edit", "execTop", "rmv", "add", "execTop"]  
[[[[1, 101, 10], [2, 102, 20], [3, 103, 15]]], [4, 104, 5], [102, 8], [], [101], [5, 105, 15], []]

**Output:**  
[null, null, null, 3, null, null, 5]

**Explanation**

```
TaskManager taskManager = new TaskManager([[1, 101, 10], [2, 102, 20], [3, 103, 15]]); // Initializes with three tasks for Users 1, 2, and 3.  
taskManager.add(4, 104, 5); // Adds task 104 with priority 5 for User 4.  
taskManager.edit(102, 8); // Updates priority of task 102 to 8.  
taskManager.execTop(); // return 3. Executes task 103 for User 3.  
taskManager.rmv(101); // Removes task 101 from the system.  
taskManager.add(5, 105, 15); // Adds task 105 with priority 15 for User 5.  
taskManager.execTop(); // return 5. Executes task 105 for User 5.
```

```
class TaskManager {  
    // taskId → [userId, priority]  
    private final Map<Integer, int[]> taskDetailsMap = new HashMap<>();  
  
    // Sorted by priority descending, then taskId descending  
    private final TreeSet<int[]> taskSet = new TreeSet<>((a, b) → {  
        if (a[0] == b[0]) {  
            return b[1] - a[1]; // higher taskId first if same priority  
        }  
        return b[0] - a[0]; // higher priority first  
    });  
  
    public TaskManager(List<List<Integer>> tasks) {  
        for (List<Integer> task : tasks) {  
            int userId = task.get(0);
```

```

        int taskId = task.get(1);
        int priority = task.get(2);
        add(userId, taskId, priority);
    }
}

public void add(int userId, int taskId, int priority) {
    taskDetailsMap.put(taskId, new int[]{userId, priority});
    taskSet.add(new int[]{priority, taskId});
}

public void edit(int taskId, int newPriority) {
    int[] details = taskDetailsMap.get(taskId);
    int userId = details[0];
    int oldPriority = details[1];

    taskSet.remove(new int[]{oldPriority, taskId});
    taskSet.add(new int[]{newPriority, taskId});
    taskDetailsMap.put(taskId, new int[]{userId, newPriority});
}

public void rmv(int taskId) {
    int[] details = taskDetailsMap.remove(taskId);
    int priority = details[1];
    taskSet.remove(new int[]{priority, taskId});
}

public int execTop() {
    if (taskSet.isEmpty()) {
        return -1;
    }
    int[] top = taskSet.pollFirst();
    int taskId = top[1];
    int[] details = taskDetailsMap.remove(taskId);
    return details[0]; // return userId
}

```

```
    }  
}
```

---

## 55. Design Spreadsheet

**Input:**

```
["Spreadsheet", "getValue", "setCell", "getValue", "setCell", "getValue", "resetCell",  
"getValue"]  
[[3], ["=5+7"], ["A1", 10], [=A1+6"], ["B2", 15], [=A1+B2"], ["A1"], [=A1+B2"]]
```

**Output:**

```
[null, 12, null, 16, null, 25, null, 15]
```

**Explanation**

```
Spreadsheet spreadsheet = new Spreadsheet(3); // Initializes a spreadsheet with 3 rows and 26 columns  
spreadsheet.getValue("=5+7"); // returns 12 (5+7)  
spreadsheet.setCell("A1", 10); // sets A1 to 10  
spreadsheet.getValue("=A1+6"); // returns 16 (10+6)  
spreadsheet.setCell("B2", 15); // sets B2 to 15  
spreadsheet.getValue("=A1+B2"); // returns 25 (10+15)  
spreadsheet.resetCell("A1"); // resets A1 to 0  
spreadsheet.getValue("=A1+B2"); // returns 15 (0+15)
```

```
class Spreadsheet {  
    // Map to store each cell's value  
    private final Map<String, Integer> cellValueMap = new HashMap<>();  
  
    public Spreadsheet(int rows) {  
        // Row count is not used since we are only tracking cells explicitly set  
    }  
  
    // Set a cell to a specific value  
    public void setCell(String cellName, int value) {  
        cellValueMap.put(cellName, value);  
    }  
  
    // Reset a cell (remove its value)  
}
```

```
public void resetCell(String cellName) {  
    cellValueMap.remove(cellName);  
}  
  
// Evaluate a formula and return the computed value  
public int getValue(String formula) {  
    int result = 0;  
    // Remove '=' from the beginning, then split by '+'  
    String[] components = formula.substring(1).split("\\+");  
  
    for (String component : components) {  
        if (Character.isDigit(component.charAt(0))) {  
            result += Integer.parseInt(component);  
        } else {  
            result += cellValueMap.getOrDefault(component, 0);  
        }  
    }  
    return result;  
}  
}
```

---

56. [Implement Router](#)

**Input:**  
["Router", "addPacket", "addPacket", "addPacket", "addPacket", "addPacket", "forwardPacket", "addPacket", "getCount"]  
[[3], [1, 4, 90], [2, 5, 90], [1, 4, 90], [3, 5, 95], [4, 5, 105], [], [5, 2, 110], [5, 100, 110]]

**Output:**  
[null, true, true, false, true, true, [2, 5, 90], true, 1]

#### Explanation

```
Router router = new Router(3); // Initialize Router with memoryLimit of 3.  
router.addPacket(1, 4, 90); // Packet is added. Return True.  
router.addPacket(2, 5, 90); // Packet is added. Return True.  
router.addPacket(1, 4, 90); // This is a duplicate packet. Return False.  
router.addPacket(3, 5, 95); // Packet is added. Return True  
router.addPacket(4, 5, 105); // Packet is added, [1, 4, 90] is removed as number of packets exceeds memoryLimit.  
Return True.  
router.forwardPacket(); // Return [2, 5, 90] and remove it from router.  
router.addPacket(5, 2, 110); // Packet is added. Return True.  
router.getCount(5, 100, 110); // The only packet with destination 5 and timestamp in the inclusive range [100, 110] is  
[4, 5, 105]. Return 1.
```

```
import java.util.*;  
  
class Router {  
    private final int memoryLimit;  
    private final Set<String> packetSet = new HashSet<>();  
    private final Deque<Packet> packetQueue = new ArrayDeque<>();  
    private final Map<Integer, List<Integer>> destinationTimestamps = new HashMap<>();  
  
    public Router(int memoryLimit) {  
        this.memoryLimit = memoryLimit;  
    }  
  
    public boolean addPacket(int source, int destination, int timestamp) {  
        String packetKey = source + "_" + destination + "_" + timestamp;  
        if (packetSet.contains(packetKey)) {  
            return false;  
        }  
  
        if (packetQueue.size() == memoryLimit) {  
            List<Integer> timestamps = destinationTimestamps.get(destination);  
            if (timestamps != null) {  
                timestamps.remove(timestamp);  
            }  
            if (timestamps.isEmpty()) {  
                destinationTimestamps.remove(destination);  
            }  
        }  
        packetSet.add(packetKey);  
        packetQueue.add(new Packet(source, destination, timestamp));  
        List<Integer> timestamps = destinationTimestamps.getOrDefault(destination, new ArrayList<>());  
        timestamps.add(timestamp);  
        destinationTimestamps.put(destination, timestamps);  
        return true;  
    }  
  
    public List<Integer> forwardPacket() {  
        List<Integer> packet = packetQueue.poll();  
        packetSet.remove(packetKey);  
        return packet;  
    }  
  
    public int getCount(int destination, int start, int end) {  
        List<Integer> timestamps = destinationTimestamps.get(destination);  
        if (timestamps == null) {  
            return 0;  
        }  
        int count = 0;  
        for (int timestamp : timestamps) {  
            if (timestamp >= start && timestamp <= end) {  
                count++;  
            }  
        }  
        return count;  
    }  
}  
  
class Packet {  
    int source;  
    int destination;  
    int timestamp;  
}
```

```

        forwardPacket();
    }

    Packet newPacket = new Packet(source, destination, timestamp);
    packetQueue.offerLast(newPacket);
    packetSet.add(packetKey);

    destinationTimestamps
        .computeIfAbsent(destination, k → new ArrayList<>())
        .add(timestamp);

    return true;
}

public int[] forwardPacket() {
    if (packetQueue.isEmpty()) {
        return new int[0];
    }

    Packet oldestPacket = packetQueue.pollFirst();
    String packetKey = oldestPacket.source + "_" + oldestPacket.destination + '_';
    packetSet.remove(packetKey);

    List<Integer> timestamps = destinationTimestamps.get(oldestPacket.destination);
    if (timestamps != null) {
        int index = Collections.binarySearch(timestamps, oldestPacket.timestamp);
        if (index >= 0) {
            timestamps.remove(index);
        }
        if (timestamps.isEmpty()) {
            destinationTimestamps.remove(oldestPacket.destination);
        }
    }
}

return new int[]{oldestPacket.source, oldestPacket.destination, oldestPacket.timestamp};
}

```

```

public int getCount(int destination, int startTime, int endTime) {
    List<Integer> timestamps = destinationTimestamps.get(destination);
    if (timestamps == null || timestamps.isEmpty()) {
        return 0;
    }

    int startIndex = lowerBound(timestamps, startTime);
    int endIndex = upperBound(timestamps, endTime);
    return endIndex - startIndex;
}

private int lowerBound(List<Integer> list, int target) {
    int left = 0, right = list.size();
    while (left < right) {
        int mid = (left + right) / 2;
        if (list.get(mid) < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

private int upperBound(List<Integer> list, int target) {
    int left = 0, right = list.size();
    while (left < right) {
        int mid = (left + right) / 2;
        if (list.get(mid) <= target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

```

```
}

private static class Packet {
    int source;
    int destination;
    int timestamp;

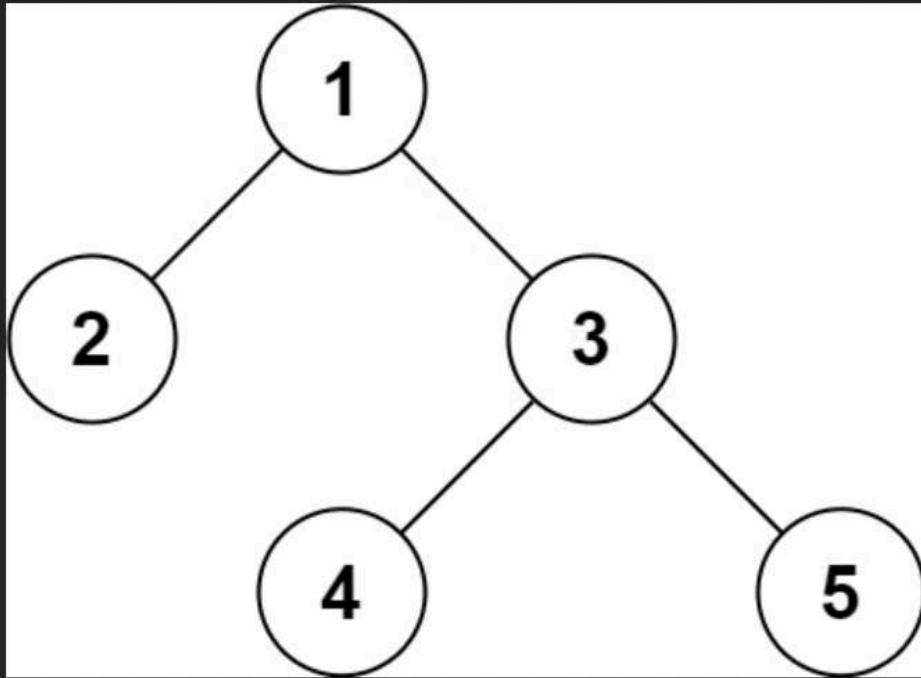
    Packet(int source, int destination, int timestamp) {
        this.source = source;
        this.destination = destination;
        this.timestamp = timestamp;
    }
}

/**
 * Your Router object will be instantiated and called as such:
 * Router obj = new Router(memoryLimit);
 * boolean param_1 = obj.addPacket(source,destination,timestamp);
 * int[] param_2 = obj.forwardPacket();
 * int param_3 = obj.getCount(destination,startTime,endTime);
 */
```

---

57. [Serialize and Deserialize Binary Tree](#)

**Example 1:**



**Input:** root = [1,2,3,null,null,4,5]  
**Output:** [1,2,3,null,null,4,5]

**Example 2:**

**Input:** root = []  
**Output:** []

```
public class Codec {

    // Encodes a tree to a single string using preorder traversal.
    public String serialize(TreeNode root) {
        StringBuilder serializedTree = new StringBuilder();
        serializeHelper(root, serializedTree);
        return serializedTree.toString();
    }

    private void serializeHelper(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("null");
        } else {
            sb.append(node.val);
            sb.append(",");
            serializeHelper(node.left, sb);
            serializeHelper(node.right, sb);
        }
    }
}
```

```

private void serializeHelper(TreeNode node, StringBuilder builder) {
    if (node == null) {
        builder.append("null,");
        return;
    }
    builder.append(node.val).append(",");
    serializeHelper(node.left, builder);
    serializeHelper(node.right, builder);
}

// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {
    Queue<String> nodesQueue = new LinkedList<>(Arrays.asList(data.split(",")));
    return deserializeHelper(nodesQueue);
}

private TreeNode deserializeHelper(Queue<String> nodes) {
    String current = nodes.poll();
    if (current.equals("null")) {
        return null;
    }
    TreeNode node = new TreeNode(Integer.parseInt(current));
    node.left = deserializeHelper(nodes);
    node.right = deserializeHelper(nodes);
    return node;
}
}

```

```

public class Codec {

    // Serializes a binary tree to a single string using BFS
    public String serialize(TreeNode root) {
        if (root == null) return "null";

        StringBuilder serializedData = new StringBuilder();

```

```

Queue<TreeNode> bfsQueue = new LinkedList<>();
bfsQueue.offer(root);

while (!bfsQueue.isEmpty()) {
    TreeNode currentNode = bfsQueue.poll();

    if (currentNode == null) {
        serializedData.append("null,");
        continue;
    }

    serializedData.append(currentNode.val).append(",");
    bfsQueue.offer(currentNode.left);
    bfsQueue.offer(currentNode.right);
}

return serializedData.toString();
}

// Deserializes the encoded string back into a binary tree using BFS
public TreeNode deserialize(String data) {
    if (data.equals("null")) return null;

    String[] values = data.split(",");
    TreeNode root = new TreeNode(Integer.parseInt(values[0]));
    Queue<TreeNode> bfsQueue = new LinkedList<>();
    bfsQueue.offer(root);

    int index = 1;
    while (!bfsQueue.isEmpty() && index < values.length) {
        TreeNode parent = bfsQueue.poll();

        // Process left child
        if (!values[index].equals("null")) {
            TreeNode leftChild = new TreeNode(Integer.parseInt(values[index]));
            parent.left = leftChild;
        }

        // Process right child
        if (!values[index+1].equals("null")) {
            TreeNode rightChild = new TreeNode(Integer.parseInt(values[index+1]));
            parent.right = rightChild;
        }
        bfsQueue.offer(leftChild);
        bfsQueue.offer(rightChild);
        index += 2;
    }
}

```

```

        bfsQueue.offer(leftChild);
    }
    index++;

    // Process right child
    if (index < values.length && !values[index].equals("null")) {
        TreeNode rightChild = new TreeNode(Integer.parseInt(values[index]));
        parent.right = rightChild;
        bfsQueue.offer(rightChild);
    }
    index++;
}

return root;
}
}

```

---

58. Range Sum Query - Immutable

**Input**

```
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
```

**Output**

```
[null, 1, -1, -3]
```

**Explanation**

```
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1
numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1
numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3
```

```

class NumArray {
    private int[] prefixSum;
```

```

public NumArray(int[] nums) {
    int n = nums.length;
    prefixSum = new int[n + 1]; // One extra space for convenience
    prefixSum[0] = 0; // Base case: sum of no elements is 0

    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }
}

// Query: Compute the sum of the range [left, right]
public int sumRange(int left, int right) {
    return prefixSum[right + 1] - prefixSum[left];
}

}

/***
 * Your NumArray object will be instantiated and called as such:
 * NumArray obj = new NumArray(nums);
 * int param_1 = obj.sumRange(left,right);
 */

```

---

59. [Range Sum Query 2D - Immutable](#)

3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5

**Input**

```
["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]
[[[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]],,
[2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]]
```

**Output**

```
[null, 8, 11, 12]
```

**Explanation**

```
NumMatrix numMatrix = new NumMatrix([[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4,
1, 0, 1, 7], [1, 0, 3, 0, 5]]);
numMatrix.sumRegion(2, 1, 4, 3); // return 8 (i.e sum of the red rectangle)
numMatrix.sumRegion(1, 1, 2, 2); // return 11 (i.e sum of the green rectangle)
numMatrix.sumRegion(1, 2, 2, 4); // return 12 (i.e sum of the blue rectangle)
```

```
class NumMatrix {
    public NumMatrix(int[][][] matrix) {
        if (matrix.length == 0)
            return;

        final int m = matrix.length;
        final int n = matrix[0].length;

        // prefix[i][j] := sum of matrix[0..i][0..j]
        prefix = new int[m + 1][n + 1];
    }
```

```

for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++j)
        prefix[i + 1][j + 1] = matrix[i][j] + prefix[i][j + 1] + prefix[i + 1][j] - prefix[i][j];
}

public int sumRegion(int row1, int col1, int row2, int col2) {
    return prefix[row2 + 1][col2 + 1] - prefix[row1][col2 + 1]
        - prefix[row2 + 1][col1] + prefix[row1][col1];
}

private int[][] prefix;
}

```

## 50. Range Sum Query - Mutable

**Input**

```

["NumArray", "sumRange", "update", "sumRange"]
[[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]

```

**Output**

```
[null, 9, null, 8]
```

**Explanation**

```

NumArray numArray = new NumArray([1, 3, 5]);
numArray.sumRange(0, 2); // return 1 + 3 + 5 = 9
numArray.update(1, 2); // nums = [1, 2, 5]
numArray.sumRange(0, 2); // return 1 + 2 + 5 = 8

```

```
class NumArray {
```

```

    private int[] tree;
    private int[] nums;
    private int size;
}
```

```

public NumArray(int[] nums) {
    this.size = nums.length;
    this.tree = new int[size + 1]; // 1-based indexing
    this.nums = new int[size];    // store original array

    for (int i = 0; i < size; i++) {
        this.nums[i] = nums[i];
        updateTree(i + 1, nums[i]); // build tree with initial values
    }
}

// Internal method to update BIT
private void updateTree(int i, int delta) {
    while (i <= size) {
        tree[i] += delta;
        i += i & -i;
    }
}

// Update the value at index `index` to `val`
public void update(int index, int val) {
    int delta = val - nums[index];
    nums[index] = val;
    updateTree(index + 1, delta);
}

// Query prefix sum from [1...i]
private int query(int i) {
    int sum = 0;
    while (i > 0) {
        sum += tree[i];
        i -= i & -i;
    }
    return sum;
}

```

```

// Query range sum [left, right]
public int sumRange(int left, int right) {
    return query(right + 1) - query(left);
}
}

```

### 31. Range Sum Query 2D - Mutable

3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5

The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

**Example:**

```

Given matrix = [
    [3, 0, 1, 4, 2],
    [5, 6, 3, 2, 1],
    [1, 2, 0, 1, 5],
    [4, 1, 0, 1, 7],
    [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
update(3, 2, 2)
sumRegion(2, 1, 4, 3) -> 10

```

```

class BinaryIndexedTree {
    private int n;
    private int[] c;

    public BinaryIndexedTree(int n) {
        this.n = n;
        c = new int[n + 1];
    }
}

```

```

public void update(int x, int delta) {
    while (x <= n) {
        c[x] += delta;
        x += lowbit(x);
    }
}

public int query(int x) {
    int s = 0;
    while (x > 0) {
        s += c[x];
        x -= lowbit(x);
    }
    return s;
}

public static int lowbit(int x) {
    return x & -x;
}

class NumMatrix {
    private BinaryIndexedTree[] trees;

    public NumMatrix(int[][] matrix) {
        int m = matrix.length;
        int n = matrix[0].length;
        trees = new BinaryIndexedTree[m];
        for (int i = 0; i < m; ++i) {
            BinaryIndexedTree tree = new BinaryIndexedTree(n);
            for (int j = 0; j < n; ++j) {
                tree.update(j + 1, matrix[i][j]);
            }
            trees[i] = tree;
        }
    }
}

```

```

}

public void update(int row, int col, int val) {
    BinaryIndexedTree tree = trees[row];
    int prev = tree.query(col + 1) - tree.query(col);
    tree.update(col + 1, val - prev);
}

public int sumRegion(int row1, int col1, int row2, int col2) {
    int s = 0;
    for (int i = row1; i <= row2; ++i) {
        BinaryIndexedTree tree = trees[i];
        s += tree.query(col2 + 1) - tree.query(col1);
    }
    return s;
}

}

/**
 * Your NumMatrix object will be instantiated and called as such:
 * NumMatrix obj = new NumMatrix(matrix);
 * obj.update(row,col,val);
 * int param_2 = obj.sumRegion(row1,col1,row2,col2);
 */

```

---

32. [Flatten Nested List Iterator](#)

**Example 1:**

**Input:** nestedList = [[1,1],2,[1,1]]  
**Output:** [1,1,2,1,1]  
**Explanation:** By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,1,2,1,1].

**Example 2:**

**Input:** nestedList = [1,[4,[6]]]  
**Output:** [1,4,6]  
**Explanation:** By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,4,6].

```
public class NestedIterator implements Iterator<Integer> {  
    private final Deque<NestedInteger> stack = new ArrayDeque<>();  
  
    public NestedIterator(List<NestedInteger> nestedList) {  
        // Push all elements to stack in reverse order  
        for (int i = nestedList.size() - 1; i >= 0; i--) {  
            stack.push(nestedList.get(i));  
        }  
    }  
  
    @Override  
    public Integer next() {  
        // Top element is guaranteed to be integer because hasNext checks it  
        return stack.pop().getInteger();  
    }  
  
    @Override  
    public boolean hasNext() {  
        while (!stack.isEmpty()) {  
            NestedInteger top = stack.peek();  
            if (top.isInteger()) {  
                return true;  
            }  
  
            // Unwrap the nested list  
        }  
    }  
}
```

```

        stack.pop();
        List<NestedInteger> nestedList = top.getList();
        for (int i = nestedList.size() - 1; i >= 0; i--) {
            stack.push(nestedList.get(i));
        }
    }
    return false;
}
}

```

33. [Design Compressed String Iterator](#)

```

StringIterator iterator = new StringIterator("L1e2t1C1o1d1e1");

iterator.next(); // return 'L'
iterator.next(); // return 'e'
iterator.next(); // return 'e'
iterator.next(); // return 't'
iterator.next(); // return 'C'
iterator.next(); // return 'o'
iterator.next(); // return 'd'
iterator.hasNext(); // return true
iterator.next(); // return 'e'
iterator.hasNext(); // return false
iterator.next(); // return ' '

```

```

class StringIterator {
    private final List<CharFrequency> charFrequencyList = new ArrayList<>();
    private int currentIndex = 0;

    public StringIterator(String compressedString) {
        int length = compressedString.length();
        int i = 0;
        while (i < length) {

```

```

        char currentChar = compressedString.charAt(i);
        int frequency = 0;
        while (++i < length && Character.isDigit(compressedString.charAt(i))) {
            frequency = frequency * 10 + (compressedString.charAt(i) - '0');
        }
        charFrequencyList.add(new CharFrequency(currentChar, frequency));
    }
}

public char next() {
    if (!hasNext()) {
        return ' ';
    }
    CharFrequency current = charFrequencyList.get(currentIndex);
    char result = current.character;
    if (--current.frequency == 0) {
        currentIndex++;
    }
    return result;
}

public boolean hasNext() {
    return currentIndex < charFrequencyList.size() && charFrequencyList.get(cu
}
}

class CharFrequency {
    char character;
    int frequency;

    CharFrequency(char character, int frequency) {
        this.character = character;
        this.frequency = frequency;
    }
}

```

---

## 34. Zigzag Iterator

### Example 1:

```
Input: v1 = [1,2], v2 = [3,4,5,6]
```

```
Output: [1,3,2,4,5,6]
```

```
Explanation: By calling next repeatedly until hasNext returns false, the order of ele
```

### Example 2:

```
Input: v1 = [1], v2 = []
```

```
Output: [1]
```

### Example 3:

```
Input: v1 = [], v2 = [1]
```

```
Output: [1]
```

```
public class ZigzagIterator {  
    private int cur;  
    private int size;  
    private List<Integer> indexes = new ArrayList<>();  
    private List<List<Integer>> vectors = new ArrayList<>();  
  
    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {  
        cur = 0;  
        size = 2;  
        indexes.add(0);  
        indexes.add(0);  
        vectors.add(v1);  
        vectors.add(v2);  
    }  
}
```

```

    }

public int next() {
    List<Integer> vector = vectors.get(cur);
    int index = indexes.get(cur);
    int res = vector.get(index);
    indexes.set(cur, index + 1);
    cur = (cur + 1) % size;
    return res;
}

public boolean hasNext() {
    int start = cur;
    while (indexes.get(cur) == vectors.get(cur).size()) {
        cur = (cur + 1) % size;
        if (start == cur) {
            return false;
        }
    }
    return true;
}

```

## 35. RLE Iterator

**Input:** ["RLEIterator","next","next","next","next"], [[[3,8,0,9,2,5]],[2],[1],[1],[2]]  
**Output:** [null,8,8,5,-1]  
**Explanation:**  
RLEIterator is initialized with RLEIterator([3,8,0,9,2,5]).  
This maps to the sequence [8,8,8,5,5].  
RLEIterator.next is then called 4 times:  
  
.next(2) exhausts 2 terms of the sequence, returning 8. The remaining sequence is now [8, 5, 5].  
  
.next(1) exhausts 1 term of the sequence, returning 8. The remaining sequence is now [5, 5].  
  
.next(1) exhausts 1 term of the sequence, returning 5. The remaining sequence is now [5].  
  
.next(2) exhausts 2 terms, returning -1. This is because the first term exhausted was 5, but the second term did not exist. Since the last term exhausted does not exist, we return -1.

```

class RLEIterator {
    private int[] encoding;
    private int i;
    private int j;

    public RLEIterator(int[] encoding) {
        this.encoding = encoding;
    }

    public int next(int n) {
        while (i < encoding.length) {
            if (encoding[i] - j < n) {
                n -= (encoding[i] - j);
                i += 2;
                j = 0;
            } else {
                j += n;
                return encoding[i + 1];
            }
        }
        return -1;
    }

}

/**
 * Your RLEIterator object will be instantiated and called as such:
 * RLEIterator obj = new RLEIterator(encoding);
 * int param_1 = obj.next(n);
 */

```

36. Insert Delete GetRandom O(1) .

```

Input
["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert",
"getRandom"]
[[], [1], [2], [2], [], [1], [2], []]
Output
[null, true, false, true, 2, true, false, 2]

Explanation
RandomizedSet randomizedSet = new RandomizedSet();
randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted
successfully.
randomizedSet.remove(2); // Returns false as 2 does not exist in the set.
randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].
randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.
randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].
randomizedSet.insert(2); // 2 was already in the set, so return false.
randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always
return 2.

```

```

import java.util.*;

class RandomizedSet {
    private List<Integer> nums;
    private Map<Integer, Integer> pos;
    private Random rand;

    public RandomizedSet() {
        nums = new ArrayList<>();
        pos = new HashMap<>();
        rand = new Random();
    }

    public boolean insert(int val) {
        if (pos.containsKey(val)) {
            return false;
        }
        pos.put(val, nums.size());
        nums.add(val);
        return true;
    }
}

```

```

public boolean remove(int val) {
    if (!pos.containsKey(val)) {
        return false;
    }
    int idx = pos.get(val);
    int lastElement = nums.get(nums.size() - 1);
    nums.set(idx, lastElement);
    pos.put(lastElement, idx);
    nums.remove(nums.size() - 1);
    pos.remove(val);
    return true;
}

public int getRandom() {
    return nums.get(rand.nextInt(nums.size()));
}
}

/**
 * Your RandomizedSet object will be instantiated and called as such:
 * RandomizedSet obj = new RandomizedSet();
 * boolean param_1 = obj.insert(val);
 * boolean param_2 = obj.remove(val);
 * int param_3 = obj.getRandom();
 */

```

---

37. Insert Delete GetRandom O(1) - Duplicates allowed

```

Input
["RandomizedCollection", "insert", "insert", "insert", "getRandom", "remove", "getRandom"]
[[], [1], [1], [2], [], [1], []]
Output
[null, true, false, true, 2, true, 1]

Explanation
RandomizedCollection randomizedCollection = new RandomizedCollection();
randomizedCollection.insert(1);      // return true since the collection does not contain 1.
                                         // Inserts 1 into the collection.
randomizedCollection.insert(1);      // return false since the collection contains 1.
                                         // Inserts another 1 into the collection. Collection now
contains [1,1].
randomizedCollection.insert(2);      // return true since the collection does not contain 2.
                                         // Inserts 2 into the collection. Collection now contains
[1,1,2].
randomizedCollection.getRandom();    // getRandom should:
                                         // - return 1 with probability 2/3, or
                                         // - return 2 with probability 1/3.
randomizedCollection.remove(1);      // return true since the collection contains 1.
                                         // Removes 1 from the collection. Collection now contains
[1,2].
randomizedCollection.getRandom();    // getRandom should return 1 or 2, both equally likely.

```

```

import java.util.*;

public class RandomizedCollection {
    private List<Integer> nums;
    private Map<Integer, Set<Integer>> valToIndices;
    private Random rand;

    public RandomizedCollection() {
        nums = new ArrayList<>();
        valToIndices = new HashMap<>();
        rand = new Random();
    }

    public boolean insert(int val) {
        boolean notPresent = !valToIndices.containsKey(val);
        valToIndices.putIfAbsent(val, new HashSet<>());
        valToIndices.get(val).add(nums.size());
        nums.add(val);
        return notPresent;
    }
}

```

```

}

public boolean remove(int val) {
    if (!valToIndices.containsKey(val)) {
        return false;
    }

    // Get arbitrary index of the value to remove
    int indexToRemove = valToIndices.get(val).iterator().next();
    valToIndices.get(val).remove(indexToRemove);

    // Move the last element to the position being removed
    int lastElement = nums.get(nums.size() - 1);
    nums.set(indexToRemove, lastElement);

    // Update the index set for the last element
    valToIndices.get(lastElement).add(indexToRemove);
    valToIndices.get(lastElement).remove(nums.size() - 1);

    nums.remove(nums.size() - 1);

    if (valToIndices.get(val).isEmpty()) {
        valToIndices.remove(val);
    }
}

return true;
}

public int getRandom() {
    return nums.get(rand.nextInt(nums.size()));
}
}

/** 
 * Your RandomizedCollection object will be instantiated and called as such:
 * RandomizedCollection obj = new RandomizedCollection();
 */

```

```
* boolean param_1 = obj.insert(val);
* boolean param_2 = obj.remove(val);
* int param_3 = obj.getRandom();
*/
```

---

38. Online Stock Span

```
Input
["StockSpanner", "next", "next", "next", "next", "next", "next", "next"]
[], [100], [80], [60], [70], [60], [75], [85]
Output
[null, 1, 1, 1, 2, 1, 4, 6]

Explanation
StockSpanner stockSpanner = new StockSpanner();
stockSpanner.next(100); // return 1
stockSpanner.next(80); // return 1
stockSpanner.next(60); // return 1
stockSpanner.next(70); // return 2
stockSpanner.next(60); // return 1
stockSpanner.next(75); // return 4, because the last 4 prices (including today's price of
75) were less than or equal to today's price.
stockSpanner.next(85); // return 6
```

```
import java.util.Stack;

class StockSpanner {
    Stack<int[]> stack;

    public StockSpanner() {
        stack = new Stack<>();
    }

    public int next(int price) {
        int span = 1;
```

```

        while (!stack.isEmpty() && stack.peek()[0] <= price) {
            span += stack.pop()[1];
        }
        stack.push(new int[]{price, span});
        return span;
    }

/*
 * Your StockSpanner object will be instantiated and called as such:
 * StockSpanner obj = new StockSpanner();
 * int param_1 = obj.next(price);
 */

```

---

### 39. Design SQL

#### Input

```

["SQL", "insertRow", "selectCell", "insertRow", "deleteRow", "selectCell"]
[[["one", "two", "three"], [2, 3, 1]], ["two", ["first", "second",
"third"]], ["two", 1, 3], ["two", ["fourth", "fifth", "sixth"]],
["two", 1], ["two", 2, 2]]

```

#### Output

```
[null, null, "third", null, null, "fifth"]
```

#### Explanation

```

SQL sql = new SQL(["one", "two", "three"], [2, 3, 1]);
// creates three tables.
sql.insertRow("two", ["first", "second", "third"]);
// adds a row to the table "two". Its id is 1.
sql.selectCell("two", 1, 3);
// return "third", finds the value of the third column in the
// row with id 1 of the table "two".
sql.insertRow("two", ["fourth", "fifth", "sixth"]);
// adds another row to the table "two". Its id is 2.

```

```
sql.deleteRow("two", 1);
// deletes the first row of the table "two". Note that the
// second row will still have the id 2.
sql.selectCell("two", 2, 2); // return "fifth",
// finds the value of the second column in the row with id 2 of the table "two".
```

```
class SQL {
    private Map<String, List<List<String>>> tables;

    public SQL(List<String> names, List<Integer> columns) {
        tables = new HashMap<>(names.size());
    }

    public void insertRow(String name, List<String> row) {
        tables.computeIfAbsent(name, k → new ArrayList<>()).add(row);
    }

    public void deleteRow(String name, int rowId) {
    }

    public String selectCell(String name, int rowId, int columnId) {
        return tables.get(name).get(rowId - 1).get(columnId - 1);
    }
}

/**
 * Your SQL object will be instantiated and called as such:
 * SQL obj = new SQL(names, columns);
 * obj.insertRow(name, row);
 * obj.deleteRow(name, rowId);
 * String param_3 = obj.selectCell(name, rowId, columnId);
 */
```

## 70. Design a Text Editor

### Input

```
["TextEditor", "addText", "deleteText", "addText", "cursorRight", "cursorLeft",
 "deleteText", "cursorLeft", "cursorRight"]
[[], ["leetcode"], [4], ["practice"], [3], [8], [10], [2], [6]]
```

### Output

```
[null, null, 4, null, "etpractice", "leet", 4, "", "practi"]
```

### Explanation

```
TextEditor textEditor = new TextEditor(); // The current text is "|". (The '|' character
represents the cursor)
textEditor.addText("leetcode"); // The current text is "leetcode|".
textEditor.deleteText(4); // return 4
    // The current text is "leet|".
    // 4 characters were deleted.
textEditor.addText("practice"); // The current text is "leetpractice|".
textEditor.cursorRight(3); // return "etpractice"
    // The current text is "leetpractice|".
    // The cursor cannot be moved beyond the actual text and thus did
not move.
    // "etpractice" is the last 10 characters to the left of the
cursor.
textEditor.cursorLeft(8); // return "leet"
    // The current text is "leet|practice".
    // "leet" is the last min(10, 4) = 4 characters to the left of the
cursor.
textEditor.deleteText(10); // return 4
    // The current text is "|practice".
    // Only 4 characters were deleted.
textEditor.cursorLeft(2); // return ""
    // The current text is "|practice".
    // The cursor cannot be moved beyond the actual text and thus did
not move.
    // "" is the last min(10, 0) = 0 characters to the left of the
cursor.
textEditor.cursorRight(6); // return "practi"
    // The current text is "practi|ce".
    // "practi" is the last min(10, 6) = 6 characters to the left of
the cursor.
```

```
class TextEditor {
    public void addText(String text) {
        sb.append(text);
    }
}
```

```

public int deleteText(int k) {
    final int numDeleted = Math.min(k, sb.length());
    for (int i = 0; i < numDeleted; ++i)
        sb.deleteCharAt(sb.length() - 1);
    return numDeleted;
}

public String cursorLeft(int k) {
    while (!sb.isEmpty() && k-- > 0) {
        stack.push(sb.charAt(sb.length() - 1));
        sb.deleteCharAt(sb.length() - 1);
    }
    return getString();
}

public String cursorRight(int k) {
    while (!stack.isEmpty() && k-- > 0)
        sb.append(stack.pop());
    return getString();
}

private String getString() {
    if (sb.length() < 10)
        return sb.toString();
    return sb.substring(sb.length() - 10).toString();
}

private StringBuilder sb = new StringBuilder();
private Deque<Character> stack = new ArrayDeque<>();
}

```

## 71. Design Memory Allocator

```
Input
["Allocator", "allocate", "allocate", "allocate", "freeMemory", "allocate", "allocate",
"allocate", "freeMemory", "allocate", "freeMemory"]
[[10], [1, 1], [1, 2], [1, 3], [2], [3, 4], [1, 1], [1, 1], [1], [10, 2], [7]]
Output
[null, 0, 1, 2, 1, 3, 1, 6, 3, -1, 0]

Explanation
Allocator loc = new Allocator(10); // Initialize a memory array of size 10. All memory units
are initially free.
loc.allocate(1, 1); // The leftmost block's first index is 0. The memory array becomes
[1,_,_,_,_,_,_,_,_]. We return 0.
loc.allocate(1, 2); // The leftmost block's first index is 1. The memory array becomes
[1,2,_,_,_,_,_,_,_]. We return 1.
loc.allocate(1, 3); // The leftmost block's first index is 2. The memory array becomes
[1,2,3,_,_,_,_,_,_]. We return 2.
loc.freeMemory(2); // Free all memory units with mID 2. The memory array becomes [1,_,
3,_,_,_,_,_,_]. We return 1 since there is only 1 unit with mID 2.
loc.allocate(3, 4); // The leftmost block's first index is 3. The memory array becomes
[1,_,3,4,4,4,_,_,_]. We return 3.
loc.allocate(1, 1); // The leftmost block's first index is 1. The memory array becomes
[1,1,3,4,4,4,1,_,_]. We return 1.
loc.allocate(1, 1); // The leftmost block's first index is 6. The memory array becomes
[1,1,3,4,4,4,1,1,1]. We return 6.
loc.freeMemory(1); // Free all memory units with mID 1. The memory array becomes
[_,_,3,4,4,4,1,1,1]. We return 3 since there are 3 units with mID 1.
loc.allocate(10, 2); // We can not find any free block with 10 consecutive free memory
units, so we return -1.
loc.freeMemory(7); // Free all memory units with mID 7. The memory array remains the same
since there is no memory unit with mID 7. We return 0.
```

```
class Allocator {
    private final int[] memorySlots;

    public Allocator(int n) {
        memorySlots = new int[n]; // 0 indicates a free slot
    }

    public int allocate(int size, int memoryID) {
        int freeCount = 0;

        for (int index = 0; index < memorySlots.length; ++index) {
```

```

        if (memorySlots[index] > 0) {
            freeCount = 0; // Block is taken, reset count
        } else if (++freeCount == size) {
            int startIndex = index - size + 1;
            Arrays.fill(memorySlots, startIndex, index + 1, memoryID);
            return startIndex;
        }
    }

    return -1; // No suitable free segment found
}

public int freeMemory(int memoryID) {
    int freedCount = 0;

    for (int index = 0; index < memorySlots.length; ++index) {
        if (memorySlots[index] == memoryID) {
            memorySlots[index] = 0; // Mark as free
            ++freedCount;
        }
    }

    return freedCount;
}

```

---

72. [Encode N-ary Tree to Binary Tree](#)

## N-ary Tree:



## Binary Tree:



```
class Codec {  
    // Encodes an n-ary tree to a binary tree.  
    public TreeNode encode(Node root) {
```

```

if (root == null) {
    return null;
}
TreeNode node = new TreeNode(root.val);
if (root.children == null || root.children.isEmpty()) {
    return node;
}
TreeNode left = encode(root.children.get(0));
node.left = left;
for (int i = 1; i < root.children.size(); i++) {
    left.right = encode(root.children.get(i));
    left = left.right;
}
return node;
}

// Decodes your binary tree to an n-ary tree.
public Node decode(TreeNode data) {
    if (data == null) {
        return null;
    }
    Node node = new Node(data.val, new ArrayList<>());
    if (data.left == null) {
        return node;
    }
    TreeNode left = data.left;
    while (left != null) {
        node.children.add(decode(left));
        left = left.right;
    }
    return node;
}

// Your Codec object will be instantiated and called as such:

```

```
// Codec codec = new Codec();
// codec.decode(codec.encode(root));
```

73. [Moving Average from Data Stream](#)

```
MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3
```

```
class MovingAverage {
    private final int[] window;
    private int windowSum = 0;
    private int count = 0;

    public MovingAverage(int size) {
        window = new int[size];
    }

    public double next(int value) {
        int indexToReplace = count % window.length;
        windowSum += value - window[indexToReplace];
        window[indexToReplace] = value;
        count++;
        return windowSum * 1.0 / Math.min(count, window.length);
    }
}
```

```
    }  
}
```

---

74. Serialize and Deserialize BST

**Example 1:**

| **Input:** root = [2,1,3]  
| **Output:** [2,1,3]

**Example 2:**

| **Input:** root = []  
| **Output:** []

```
public class Codec {  
    // Serialize the BST to a string using preorder traversal  
    public String serialize(TreeNode root) {  
        StringBuilder sb = new StringBuilder();  
        preorder(root, sb);  
        return sb.toString().trim();  
    }  
  
    private void preorder(TreeNode node, StringBuilder sb) {  
        if (node == null) return;  
        sb.append(node.val).append(" ");  
        preorder(node.left, sb);  
        preorder(node.right, sb);  
    }  
}
```

```

// Deserialize the string back into a BST
public TreeNode deserialize(String data) {
    if (data.isEmpty()) return null;
    String[] values = data.split(" ");
    int[] index = new int[1]; // mutable index for recursion
    return buildBST(values, index, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private TreeNode buildBST(String[] values, int[] index, int min, int max) {
    if (index[0] >= values.length) return null;

    int val = Integer.parseInt(values[index[0]]);
    if (val < min || val > max) return null;

    TreeNode node = new TreeNode(val);
    index[0]++;
    node.left = buildBST(values, index, min, val);
    node.right = buildBST(values, index, val, max);
    return node;
}
}

```

---

## 75. Unique Word Abbreviation

Input

```

["ValidWordAbbr", "isUnique", "isUnique", "isUnique", "isUnique", "isUnique", "isUnique"]
[[["deer", "door", "cake", "card"]], ["dear"], ["cart"], ["cane"],
 ["make"], ["cake"]]

```

Output

```
[null, false, true, false, true, true]
```

## Explanation

```
ValidWordAbbr validWordAbbr = new ValidWordAbbr(["deer", "door", "cake", "ca
validWordAbbr.isUnique("dear");
// return false, dictionary word "deer" and word "dear" have the same
// abbreviation "d2r" but are not the same.
validWordAbbr.isUnique("cart");
// return true, no words in the dictionary have the abbreviation "c2t".
validWordAbbr.isUnique("cane");
// return false, dictionary word "cake" and word "cane" have the same
// abbreviation "c2e" but are not the same.
validWordAbbr.isUnique("make");
// return true, no words in the dictionary have the abbreviation "m2e".
validWordAbbr.isUnique("cake");
// return true, because "cake" is already in the dictionary and
// no other word in the dictionary has "c2e" abbreviation.
```

```
class ValidWordAbbr {
    private Map<String, Set<String>> map = new HashMap<>();

    public ValidWordAbbr(String[] dictionary) {
        for (var s : dictionary) {
            map.computeIfAbsent(abbr(s), k → new HashSet<>()).add(s);
        }
    }

    public boolean isUnique(String word) {
        var ws = map.get(abbr(word));
        return ws == null || (ws.size() == 1 && ws.contains(word));
    }

    private String abbr(String s) {
        int n = s.length();
        return n < 3 ? s : s.substring(0, 1) + (n - 2) + s.substring(n - 1);
    }
}
```

```
/*
 * Your ValidWordAbbr object will be instantiated and called as such:
 * ValidWordAbbr obj = new ValidWordAbbr(dictionary);
 * boolean param_1 = obj.isUnique(word);
 */
```

## 76. Data Stream as Disjoint Intervals

**Input**  
 ["SummaryRanges", "addNum", "getIntervals", "addNum", "getIntervals", "addNum",
 "getIntervals", "addNum", "getIntervals", "addNum", "getIntervals"]
 [[], [1], [], [3], [], [7], [], [2], [], [6], []]

**Output**  
 [null, null, [[1, 1]], null, [[1, 1], [3, 3]], null, [[1, 1], [3, 3], [7, 7]], null, [[1,
 3], [7, 7]], null, [[1, 3], [6, 7]]]

**Explanation**  
 SummaryRanges summaryRanges = new SummaryRanges();  
 summaryRanges.addNum(1); // arr = [1]  
 summaryRanges.getIntervals(); // return [[1, 1]]  
 summaryRanges.addNum(3); // arr = [1, 3]  
 summaryRanges.getIntervals(); // return [[1, 1], [3, 3]]  
 summaryRanges.addNum(7); // arr = [1, 3, 7]  
 summaryRanges.getIntervals(); // return [[1, 1], [3, 3], [7, 7]]  
 summaryRanges.addNum(2); // arr = [1, 2, 3, 7]  
 summaryRanges.getIntervals(); // return [[1, 3], [7, 7]]  
 summaryRanges.addNum(6); // arr = [1, 2, 3, 6, 7]  
 summaryRanges.getIntervals(); // return [[1, 3], [6, 7]]

```
class SummaryRanges {
    private final TreeMap<Integer, int[]> intervals;

    public SummaryRanges() {
        intervals = new TreeMap<>();
    }

    public void addNum(int val) {
        if (intervals.containsKey(val)) return;
```

```

Integer lowerKey = intervals.floorKey(val);
Integer higherKey = intervals.ceilingKey(val);

boolean mergeLeft = lowerKey != null && intervals.get(lowerKey)[1] + 1 >= val;
boolean mergeRight = higherKey != null && higherKey == val + 1;

if (mergeLeft && mergeRight) {
    int[] leftInterval = intervals.get(lowerKey);
    int[] rightInterval = intervals.get(higherKey);
    leftInterval[1] = rightInterval[1];
    intervals.remove(higherKey);
} else if (mergeLeft) {
    intervals.get(lowerKey)[1] = Math.max(intervals.get(lowerKey)[1], val);
} else if (mergeRight) {
    int[] rightInterval = intervals.remove(higherKey);
    intervals.put(val, new int[] {val, rightInterval[1]});
} else {
    intervals.put(val, new int[] {val, val});
}
}

public int[][] getIntervals() {
    return intervals.values().toArray(new int[0][]);
}
}

/**
 * Your SummaryRanges object will be instantiated and called as such:
 * SummaryRanges obj = new SummaryRanges();
 * obj.addNum(value);
 * int[][] param_2 = obj.getIntervals();
 */

```

77. [Design Phone Directory](#)



### int maxNumbers;

- **Purpose:** Stores the **maximum number of phone numbers** allowed in the directory.
- **Use Case:** Ensures that all operations (like `check(int number)`) stay within valid bounds (`0 ≤ number < maxNumbers`).



### boolean[] available;

- **Purpose:** Tracks the **availability status** of each number.
  - `available[i] = true` means number `i` is free to assign.
  - `available[i] = false` means number `i` is already taken.
- **Use Case:**
  - Used in `check()` to verify availability.
  - Updated in `get()` and `release()` to toggle usage status.



### Queue<Integer> unused;

- **Purpose:** Stores a **queue of currently unused (free) numbers**.
- **Use Case:**
  - `get()` fetches the next available number from this queue.
  - `release()` re-adds the number to this queue when it's freed.
  - Ensures O(1) access to the next free number.

#### Input

```
["PhoneDirectory", "get", "get", "check", "get", "check", "release", "check"]
[[3], [], [], [2], [], [2], [2], [2]]
```

#### Output

```
[null, 0, 1, true, 2, false, null, true]
```

#### Explanation

```
PhoneDirectory phoneDirectory = new PhoneDirectory(3);
phoneDirectory.get();
```

```

// It can return any available phone number. Here we assume it returns 0.
phoneDirectory.get();
// Assume it returns 1.
phoneDirectory.check(2);
// The number 2 is available, so return true.
phoneDirectory.get();
// It returns 2, the only number that is left.
phoneDirectory.check(2);
// The number 2 is no longer available, so return false.
phoneDirectory.release(2);
// Release number 2 back to the pool.
phoneDirectory.check(2);
// Number 2 is available again, return true.

```

```

class PhoneDirectory {
    int maxNumbers;
    boolean[] available;
    Queue<Integer> unused;

    /** Initialize your data structure here
     * @param maxNumbers - The maximum numbers that can be stored in the ph
    public PhoneDirectory(int maxNumbers) {
        this.maxNumbers = maxNumbers;
        available = new boolean[maxNumbers];
        unused = new LinkedList<Integer>();
        for (int i = 0; i < maxNumbers; i++) {
            available[i] = true;
            unused.offer(i);
        }
    }

    /** Provide a number which is not assigned to anyone.
     * @return - Return an available number. Return -1 if none is available. */
    public int get() {
        if (unused.isEmpty())

```

```
        return -1;
    else {
        int next = unused.poll();
        available[next] = false;
        return next;
    }
}

/** Check if a number is available or not. */
public boolean check(int number) {
    if (number < 0 || number >= maxNumbers)
        return false;
    else
        return available[number];
}

/** Recycle or release a number. */
public void release(int number) {
    if (!available[number]) {
        unused.offer(number);
        available[number] = true;
    }
}
```

---

78. [Shortest Word Distance II](#)

**Example:**

Assume that words = ["practice", "makes", "perfect", "coding", "makes"] .

**Input:** word1 = "coding", word2 = "practice"  
**Output:** 3

**Input:** word1 = "makes", word2 = "coding"  
**Output:** 1

**Note:**

You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

```
class WordDistance {  
    private Map<String, List<Integer>> wordIndicesMap = new HashMap<>();  
  
    public WordDistance(String[] wordsDict) {  
        for (int index = 0; index < wordsDict.length; ++index) {  
            String word = wordsDict[index];  
            wordIndicesMap  
                .computeIfAbsent(word, k → new ArrayList<>())  
                .add(index);  
        }  
    }  
  
    public int shortest(String word1, String word2) {  
        List<Integer> indices1 = wordIndicesMap.get(word1);  
        List<Integer> indices2 = wordIndicesMap.get(word2);  
        int minDistance = Integer.MAX_VALUE;  
        int pointer1 = 0, pointer2 = 0;  
  
        while (pointer1 < indices1.size() && pointer2 < indices2.size()) {  
            int index1 = indices1.get(pointer1);  
            int index2 = indices2.get(pointer2);  
            minDistance = Math.min(minDistance, Math.abs(index1 - index2));  
            if (index1 < index2) {  
                pointer1++;  
            } else {  
                pointer2++;  
            }  
        }  
        return minDistance;  
    }  
}
```

```

        if (index1 <= index2) {
            pointer1++;
        } else {
            pointer2++;
        }
    }

    return minDistance;
}
}

```

## 79. Range Module

**Input**  
`["RangeModule", "addRange", "removeRange", "queryRange", "queryRange", "queryRange"]  
[[[], [10, 20], [14, 16], [10, 14], [13, 15], [16, 17]]]`

**Output**  
`[null, null, null, true, false, true]`

**Explanation**  
`RangeModule rangeModule = new RangeModule();  
rangeModule.addRange(10, 20);  
rangeModule.removeRange(14, 16);  
rangeModule.queryRange(10, 14); // return True,(Every number in [10, 14) is being tracked)  
rangeModule.queryRange(13, 15); // return False,(Numbers like 14, 14.03, 14.17 in [13, 15)  
are not being tracked)  
rangeModule.queryRange(16, 17); // return True, (The number 16 in [16, 17) is still being  
tracked, despite the remove operation)`

```

class RangeModule {
    class Node {
        int start, end;
        Node left, right;
        boolean tracked;

        Node(int start, int end) {

```

```

        this.start = start;
        this.end = end;
        this.tracked = false;
    }

private void pushDown() {
    if (left == null && end - start > 1) {
        int mid = start + (end - start) / 2;
        left = new Node(start, mid);
        right = new Node(mid, end);
        left.tracked = tracked;
        right.tracked = tracked;
    }
}

public void addRange(int l, int r) {
    if (r <= start || end <= l) return;
    if (l <= start && end <= r) {
        tracked = true;
        left = null;
        right = null;
        return;
    }
    pushDown();
    left.addRange(l, r);
    right.addRange(l, r);
    tracked = left.tracked && right.tracked;
}

public void removeRange(int l, int r) {
    if (r <= start || end <= l) return;
    if (l <= start && end <= r) {
        tracked = false;
        left = null;
        right = null;
        return;
    }
}

```

```

    }

    pushDown();
    left.removeRange(l, r);
    right.removeRange(l, r);
    tracked = left.tracked && right.tracked;
}

public boolean queryRange(int l, int r) {
    if (r <= start || end <= l) return true;
    if (l <= start && end <= r) return tracked;
    if (left == null) return tracked;
    return left.queryRange(l, r) && right.queryRange(l, r);
}
}

Node root;

public RangeModule() {
    root = new Node(0, 1_000_000_000); // Initialize with max constraint
}

public void addRange(int left, int right) {
    root.addRange(left, right);
}

public boolean queryRange(int left, int right) {
    return root.queryRange(left, right);
}

public void removeRange(int left, int right) {
    root.removeRange(left, right);
}
}

/**

```

```
* Your RangeModule object will be instantiated and called as such:  
* RangeModule obj = new RangeModule();  
* obj.addRange(left,right);  
* boolean param_2 = obj.queryRange(left,right);  
* obj.removeRange(left,right);  
*/
```

### 30. [My Calendar I](#)

**Input**  
["MyCalendar", "book", "book", "book"]  
[[], [10, 20], [15, 25], [20, 30]]  
**Output**  
[null, true, false, true]

**Explanation**  
MyCalendar myCalendar = new MyCalendar();  
myCalendar.book(10, 20); // return True  
myCalendar.book(15, 25); // return False, It can not be booked because time 15 is already booked by another event.  
myCalendar.book(20, 30); // return True, The event can be booked, as the first event takes every time less than 20, but not including 20.

```
class MyCalendar {  
    private List<int[]> bookings;  
  
    public MyCalendar() {  
        bookings = new ArrayList<>();  
    }  
  
    public boolean book(int start, int end) {  
        for (int[] event : bookings) {  
            int bookedStart = event[0], bookedEnd = event[1];  
            if (start < bookedEnd && bookedStart < end) {  
                return false; // Overlap found  
            }  
        }  
        bookings.add(new int[]{start, end});  
        return true;  
    }  
}
```

```

        }
        bookings.add(new int[]{start, end});
        return true;
    }
}

```

## 81. My Calendar II

### Input

```
["MyCalendarTwo", "book", "book", "book", "book", "book", "book"]
[], [10, 20], [50, 60], [10, 40], [5, 15], [5, 10], [25, 55]]
```

### Output

```
[null, true, true, true, false, true, true]
```

### Explanation

```
MyCalendarTwo myCalendarTwo = new MyCalendarTwo();
myCalendarTwo.book(10, 20); // return True, The event can be booked.
myCalendarTwo.book(50, 60); // return True, The event can be booked.
myCalendarTwo.book(10, 40); // return True, The event can be double booked.
myCalendarTwo.book(5, 15); // return False, The event cannot be booked, because it would
result in a triple booking.
myCalendarTwo.book(5, 10); // return True, The event can be booked, as it does not use time
10 which is already double booked.
myCalendarTwo.book(25, 55); // return True, The event can be booked, as the time in [25, 40)
will be double booked with the third event, the time [40, 50) will be single booked, and the
time [50, 55) will be double booked with the second event.
```

```

class MyCalendarTwo {
    List<int[]> booked;
    List<int[]> overlaps;

    public MyCalendarTwo() {
        booked = new ArrayList<>();
        overlaps = new ArrayList<>();
    }

    public boolean book(int start, int end) {
        // Check if it would cause a triple booking
        for (int[] interval : overlaps) {
            if (start < interval[1] && interval[0] < end) {

```

```

        return false;
    }
}

// Add overlapping portion to overlaps
for (int[] interval : booked) {
    if (start < interval[1] && interval[0] < end) {
        int overlapStart = Math.max(start, interval[0]);
        int overlapEnd = Math.min(end, interval[1]);
        overlaps.add(new int[]{overlapStart, overlapEnd});
    }
}

// Book the current event
booked.add(new int[]{start, end});
return true;
}
}

```

---

### 32. [My Calendar III](#)

```

class MyCalendarThree {

    class Node {
        int start, end, max, lazy;
        Node left, right;
        Node(int start, int end) {
            this.start = start;
            this.end = end;
            this.max = 0;
            this.lazy = 0;
        }
    }
}

```

```

private void push() {
    if (left == null) {
        int mid = start + (end - start) / 2;
        left = new Node(start, mid);
        right = new Node(mid + 1, end);
    }
    if (lazy != 0) {
        left.max += lazy;
        left.lazy += lazy;
        right.max += lazy;
        right.lazy += lazy;
        lazy = 0;
    }
}

public void update(int l, int r) {
    if (r < start || end < l) return;
    if (l <= start && end <= r) {
        max++;
        lazy++;
        return;
    }
    push();
    left.update(l, r);
    right.update(l, r);
    max = Math.max(left.max, right.max);
}
}

Node root;
int MAX_TIME = (int) 1e9;

public MyCalendarThree() {
    root = new Node(0, MAX_TIME);
}

```

```

public int book(int start, int end) {
    root.update(start, end - 1);
    return root.max;
}

/*
 * Your MyCalendarThree object will be instantiated and called as such:
 * MyCalendarThree obj = new MyCalendarThree();
 * int param_1 = obj.book(startTime,endTime);
 */

```

### 33. [Prefix and Suffix Search](#)

```

Input
["WordFilter", "f"]
[[["apple"]], ["a", "e"]]
Output
[null, 0]
Explanation
WordFilter wordFilter = new WordFilter(["apple"]);
wordFilter.f("a", "e"); // return 0, because the word at index 0 has prefix = "a" and suffix = "e".

```

```

class WordFilter {

    class TrieNode {
        TrieNode[] children = new TrieNode[27]; // 'a' to 'z' and '{'
        int index = -1;
    }

    private TrieNode root;
}

```

```

public WordFilter(String[] words) {
    root = new TrieNode();
    for (int idx = 0; idx < words.length; ++idx) {
        String word = words[idx] + "{"; // '{' is one after 'z' in ASCII
        int len = word.length();
        for (int i = 0; i < len; ++i) {
            String key = word.substring(i) + word;
            insert(key, idx);
        }
    }
}

private void insert(String key, int index) {
    TrieNode node = root;
    for (char ch : key.toCharArray()) {
        int charIndex = ch - 'a'; // '{' will be 26
        if (node.children[charIndex] == null) {
            node.children[charIndex] = new TrieNode();
        }
        node = node.children[charIndex];
        node.index = index; // Always update to latest index
    }
}

public int f(String prefix, String suffix) {
    TrieNode node = root;
    String key = suffix + "{" + prefix;
    for (char ch : key.toCharArray()) {
        int charIndex = ch - 'a';
        if (node.children[charIndex] == null) {
            return -1;
        }
        node = node.children[charIndex];
    }
    return node.index;
}

```

```

    }
}

/***
 * Your WordFilter object will be instantiated and called as such:
 * WordFilter obj = new WordFilter(words);
 * int param_1 = obj.f(pref,suff);
 */

```

### 34. Exam Room

```

Input
["ExamRoom", "seat", "seat", "seat", "seat", "leave", "seat"]
[[10], [], [], [], [], [4], []]
Output
[null, 0, 9, 4, 2, null, 5]

Explanation
ExamRoom examRoom = new ExamRoom(10);
examRoom.seat(); // return 0, no one is in the room, then the student sits at seat number 0.
examRoom.seat(); // return 9, the student sits at the last seat number 9.
examRoom.seat(); // return 4, the student sits at the last seat number 4.
examRoom.seat(); // return 2, the student sits at the last seat number 2.
examRoom.leave(4);
examRoom.seat(); // return 5, the student sits at the last seat number 5.

```

```

class Node {
    public Node prev;
    public Node next;
    public int value;

    public Node(int value) {
        this.value = value;
    }
}

class ExamRoom {

```

```

public ExamRoom(int n) {
    this.n = n;
    join(head, tail);
}

public int seat() {
    if (head.next == tail) {
        Node node = new Node(0);
        join(head, node);
        join(node, tail);
        map.put(0, node);
        return 0;
    }

    int prevStudent = -1;
    int maxDistToClosest = 0;
    int val = 0; // the inserted value
    Node pos = null; // the inserted position

    for (Node node = head; node != tail; node = node.next) {
        if (prevStudent == -1) { // We haven't insert anything before.
            maxDistToClosest = node.value; // the distance between it and the beg
            pos = node;
        } else if ((node.value - prevStudent) / 2 > maxDistToClosest) {
            maxDistToClosest = (node.value - prevStudent) / 2;
            val = (node.value + prevStudent) / 2;
            pos = node;
        }
        prevStudent = node.value;
    }

    if (n - 1 - tail.prev.value > maxDistToClosest) {
        pos = tail;
        val = n - 1;
    }
}

```

```

Node insertedNode = new Node(val);
join(pos.prev, insertedNode);
join(insertedNode, pos);

map.put(val, insertedNode);
return val;
}

public void leave(int p) {
    Node removedNode = map.get(p);
    join(removedNode.prev, removedNode.next);
}

private int n;
private Node head = new Node(-1);
private Node tail = new Node(-1);
private Map<Integer, Node> map = new HashMap<>(); // {p: student iterator}

private void join(Node node1, Node node2) {
    node1.next = node2;
    node2.prev = node1;
}

private void remove(Node node) {
    join(node.prev, node.next);
}

```

---

35. [Online Election](#)

```

Input
["TopVotedCandidate", "q", "q", "q", "q", "q", "q"]
[[[0, 1, 1, 0, 0, 1, 0], [0, 5, 10, 15, 20, 25, 30]], [3], [12], [25], [15], [24], [8]]
Output
[null, 0, 1, 1, 0, 0, 1]

```

**Explanation**

```

TopVotedCandidate topVotedCandidate = new TopVotedCandidate([0, 1, 1, 0, 0, 1, 0], [0, 5,
10, 15, 20, 25, 30]);
topVotedCandidate.q(3); // return 0, At time 3, the votes are [0], and 0 is leading.
topVotedCandidate.q(12); // return 1, At time 12, the votes are [0,1,1], and 1 is leading.
topVotedCandidate.q(25); // return 1, At time 25, the votes are [0,1,1,0,0,1], and 1 is
leading (as ties go to the most recent vote.)
topVotedCandidate.q(15); // return 0
topVotedCandidate.q(24); // return 0
topVotedCandidate.q(8); // return 1

```

```

class TopVotedCandidate {
    private int[] times;
    private int[] leaders;

    public TopVotedCandidate(int[] persons, int[] times) {
        this.times = times;
        leaders = new int[persons.length];
        Map<Integer, Integer> voteCount = new HashMap<>();
        int leader = -1;
        int maxVotes = 0;

        for (int i = 0; i < persons.length; i++) {
            int p = persons[i];
            voteCount.put(p, voteCount.getOrDefault(p, 0) + 1);
            if (voteCount.get(p) >= maxVotes) {
                leader = p;
                maxVotes = voteCount.get(p);
            }
            leaders[i] = leader;
        }
    }

    public int q(int t) {
        int left = 0, right = times.length - 1;

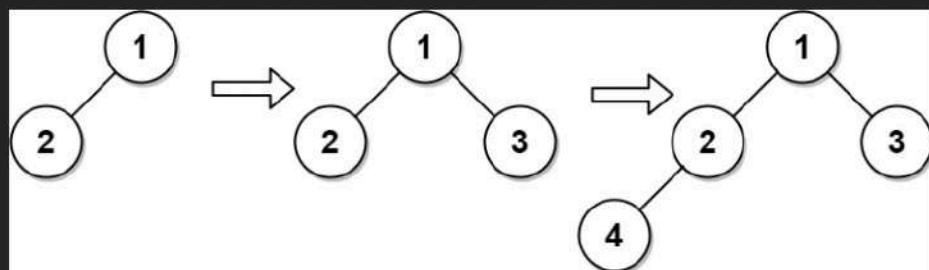
```

```
while (left < right) {  
    int mid = left + (right - left + 1) / 2;  
    if (times[mid] <= t) {  
        left = mid;  
    } else {  
        right = mid - 1;  
    }  
}  
return leaders[left];  
}  
  
/**  
 * Your TopVotedCandidate object will be instantiated and called as such:  
 * TopVotedCandidate obj = new TopVotedCandidate(persons, times);  
 * int param_1 = obj.q(t);  
 */
```

---

36. [Complete Binary Tree Inserter](#)

**Example 1:**



**Input**

```
["CBTInserter", "insert", "insert", "get_root"]
[[[1, 2]], [3], [4], []]
```

**Output**

```
[null, 1, 2, [1, 2, 3, 4]]
```

**Explanation**

```
CBTInserter cBTInserter = new CBTInserter([1, 2]);
cBTInserter.insert(3); // return 1
cBTInserter.insert(4); // return 2
cBTInserter.get_root(); // return [1, 2, 3, 4]
```

```
/**
 * Definition for a binary tree node.
 */
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
class CBTInserter {
    private TreeNode root;
```

```

private Queue<TreeNode> candidateQueue;

public CBTInserter(TreeNode root) {
    this.root = root;
    candidateQueue = new LinkedList<>();

    // Level-order traversal to populate candidateQueue
    Queue<TreeNode> bfsQueue = new LinkedList<>();
    bfsQueue.offer(root);

    while (!bfsQueue.isEmpty()) {
        TreeNode node = bfsQueue.poll();

        // If this node has missing children, it's a valid insertion candidate
        if (node.left == null || node.right == null) {
            candidateQueue.offer(node);
        }

        if (node.left != null) bfsQueue.offer(node.left);
        if (node.right != null) bfsQueue.offer(node.right);
    }
}

public int insert(int val) {
    TreeNode newNode = new TreeNode(val);
    TreeNode parent = candidateQueue.peek();

    if (parent.left == null) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
        candidateQueue.poll(); // Now it's full, remove from candidates
    }

    candidateQueue.offer(newNode); // New node might get children in future
    return parent.val;
}

```

```

    }

    public TreeNode get_root() {
        return root;
    }
}

/***
 * Your CBTInserter object will be instantiated and called as such:
 * CBTInserter obj = new CBTInserter(root);
 * int param_1 = obj.insert(val);
 * TreeNode param_2 = obj.get_root();
 */

```

### 37. Number of Recent Calls

<b>Input</b>	["RecentCounter", "ping", "ping", "ping", "ping"] [[], [1], [100], [3001], [3002]]
<b>Output</b>	[null, 1, 2, 3, 3]
<b>Explanation</b>	RecentCounter recentCounter = new RecentCounter(); recentCounter.ping(1); // requests = [1], range is [-2999,1], return 1 recentCounter.ping(100); // requests = [1, 100], range is [-2900,100], return 2 recentCounter.ping(3001); // requests = [1, 100, 3001], range is [1,3001], return 3 recentCounter.ping(3002); // requests = [1, 100, 3001, 3002], range is [2,3002], return 3

```

class RecentCounter {

    private Queue<Integer> recentPings;

    public RecentCounter() {

```

```
recentPings = new LinkedList<>();
}

public int ping(int t) {
    recentPings.offer(t);
    while (recentPings.peek() < t - 3000) {
        recentPings.poll(); // Remove outdated pings
    }
    return recentPings.size(); // Number of valid pings in the window
}

/**
 * Your RecentCounter object will be instantiated and called as such:
 * RecentCounter obj = new RecentCounter();
 * int param_1 = obj.ping(t);
 */
```

---

38. Stream of Characters

```

Input
["StreamChecker", "query", "query", "query", "query", "query", "query", "query",
"query", "query", "query"]
[[["cd", "f", "kl"]], ["a"], ["b"], ["c"], ["d"], ["e"], ["f"], ["g"], ["h"], ["i"], ["j"],
["k"], ["l"]]
Output
[null, false, false, false, true, false, true, false, false, false, false, true]

Explanation
StreamChecker streamChecker = new StreamChecker(["cd", "f", "kl"]);
streamChecker.query("a"); // return False
streamChecker.query("b"); // return False
streamChecker.query("c"); // return False
streamChecker.query("d"); // return True, because 'cd' is in the wordlist
streamChecker.query("e"); // return False
streamChecker.query("f"); // return True, because 'f' is in the wordlist
streamChecker.query("g"); // return False
streamChecker.query("h"); // return False
streamChecker.query("i"); // return False
streamChecker.query("j"); // return False
streamChecker.query("k"); // return False
streamChecker.query("l"); // return True, because 'kl' is in the wordlist

```

```

class StreamChecker {
    private static class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isWord = false;
    }

    private final TrieNode root = new TrieNode();
    private final StringBuilder stream = new StringBuilder();
    private final int maxWordLength;

    public StreamChecker(String[] words) {
        int maxLen = 0;
        for (String word : words) {
            maxLen = Math.max(maxLen, word.length());
            insert(word);
        }
        this.maxWordLength = maxLen;
    }
}

```

```

private void insert(String word) {
    TrieNode node = root;
    for (int i = word.length() - 1; i >= 0; i--) {
        char c = word.charAt(i);
        if (node.children[c - 'a'] == null)
            node.children[c - 'a'] = new TrieNode();
        node = node.children[c - 'a'];
    }
    node.isWord = true;
}

public boolean query(char letter) {
    stream.insert(0, letter); // prepend
    if (stream.length() > maxWordLength)
        stream.setLength(maxWordLength); // truncate to max word length

    TrieNode node = root;
    for (int i = 0; i < stream.length(); i++) {
        char c = stream.charAt(i);
        if (node.children[c - 'a'] == null) return false;
        node = node.children[c - 'a'];
        if (node.isWord) return true;
    }
    return false;
}

```

---

39. [Snapshot Array](#)

**Input:** ["SnapshotArray","set","snap","set","get"]  
[[3],[0,5],[],[0,6],[0,0]]

**Output:** [null,null,0,null,5]

**Explanation:**

```
SnapshotArray snapshotArr = new SnapshotArray(3); // set the length to be 3
snapshotArr.set(0,5); // Set array[0] = 5
snapshotArr.snap(); // Take a snapshot, return snap_id = 0
snapshotArr.set(0,6);
snapshotArr.get(0,0); // Get the value of array[0] with snap_id = 0, return 5
```

```
class SnapshotArray {
    private TreeMap<Integer, Integer>[] history;
    private int snapId = 0;

    public SnapshotArray(int length) {
        history = new TreeMap[length];
        for (int i = 0; i < length; i++) {
            history[i] = new TreeMap<>();
            history[i].put(0, 0); // default value at snap 0
        }
    }

    public void set(int index, int val) {
        history[index].put(snapId, val);
    }

    public int snap() {
        return snapId++;
    }

    public int get(int index, int snap_id) {
        return history[index].floorEntry(snap_id).getValue();
    }
}
```

## 90. Online Majority Element In Subarray

```

class MajorityChecker {
    private Map<Integer, List<Integer>> indicesMap = new HashMap<>();
    private int[] arr;
    private Random rand = new Random();

    public MajorityChecker(int[] arr) {
        this.arr = arr;
        for (int i = 0; i < arr.length; i++) {
            indicesMap.computeIfAbsent(arr[i], k → new ArrayList<>()).add(i);
        }
    }

    public int query(int left, int right, int threshold) {
        int len = right - left + 1;

        for (int i = 0; i < 20; i++) {
            int randomIndex = left + rand.nextInt(len);
            int candidate = arr[randomIndex];
            List<Integer> positions = indicesMap.get(candidate);

            int freq = countInRange(positions, left, right);
            if (freq >= threshold) return candidate;
        }

        return -1;
    }

    private int countInRange(List<Integer> positions, int left, int right) {
        int start = lowerBound(positions, left);
        int end = upperBound(positions, right);
        return end - start;
    }

    private int lowerBound(List<Integer> list, int target) {
        int low = 0, high = list.size();

```

```

        while (low < high) {
            int mid = (low + high) / 2;
            if (list.get(mid) < target) low = mid + 1;
            else high = mid;
        }
        return low;
    }

private int upperBound(List<Integer> list, int target) {
    int low = 0, high = list.size();
    while (low < high) {
        int mid = (low + high) / 2;
        if (list.get(mid) <= target) low = mid + 1;
        else high = mid;
    }
    return low;
}

/**
 * Your MajorityChecker object will be instantiated and called as such:
 * MajorityChecker obj = new MajorityChecker(arr);
 * int param_1 = obj.query(left,right,threshold);
 */

```

---

## 91. Design SkipList

```

Input
["Skiplist", "add", "add", "add", "search", "add", "search", "erase", "erase", "search"]
[[], [1], [2], [3], [0], [4], [1], [0], [1], [1]]
Output
[null, null, null, null, false, null, true, false, true, false]

Explanation
Skiplist skiplist = new Skiplist();
skiplist.add(1);
skiplist.add(2);
skiplist.add(3);
skiplist.search(0); // return False
skiplist.add(4);
skiplist.search(1); // return True
skiplist.erase(0); // return False, 0 is not in skiplist.
skiplist.erase(1); // return True
skiplist.search(1); // return False, 1 has already been erased.

```

```

class Node {
    public int val;
    public Node next;
    public Node down;

    public Node(int val, Node next, Node down) {
        this.val = val;
        this.next = next;
        this.down = down;
    }
}

class Skiplist {
    public boolean search(int target) {
        for (Node node = dummy; node != null; node = node.down) {
            node = advance(node, target);
            if (node.next != null && node.next.val == target)
                return true;
        }
        return false;
    }
}

```

```

}

public void add(int num) {
    // Collect nodes that are before the insertion point.
    Deque<Node> nodes = new ArrayDeque<>();
    for (Node node = dummy; node != null; node = node.down) {
        node = advance(node, num);
        nodes.push(node);
    }

    Node down = null;
    boolean shouldInsert = true;
    while (shouldInsert && !nodes.isEmpty()) {
        Node prev = nodes.poll();
        prev.next = new Node(num, prev.next, down);
        down = prev.next;
        shouldInsert = Math.random() < 0.5;
    }

    // Create a topmost new level dummy that points to the existing dummy.
    if (shouldInsert)
        dummy = new Node(-1, null, dummy);
}

public boolean erase(int num) {
    boolean found = false;
    for (Node node = dummy; node != null; node = node.down) {
        node = advance(node, num);
        if (node.next != null && node.next.val == num) {
            node.next = node.next.next;
            found = true;
        }
    }
    return found;
}

```

```
private Node dummy = new Node(-1, null, null);

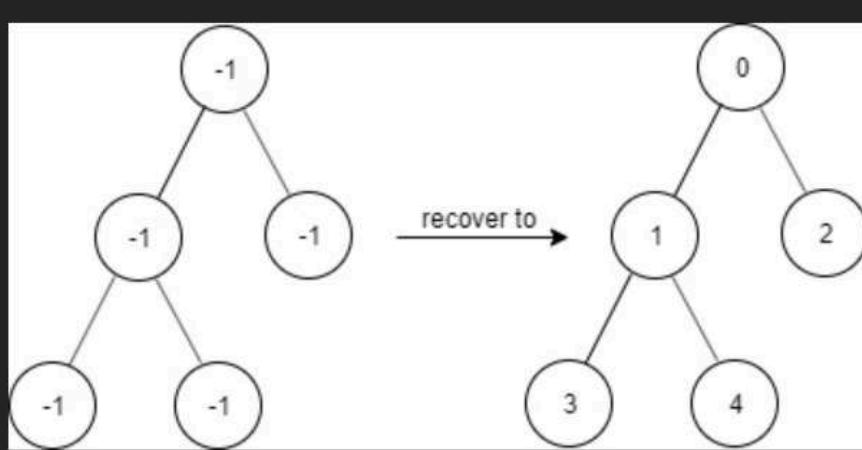
private Node advance(Node node, int target) {
    while (node.next != null && node.next.val < target)
        node = node.next;
    return node;
}

/***
 * Your Skiplist object will be instantiated and called as such:
 * Skiplist obj = new Skiplist();
 * boolean param_1 = obj.search(target);
 * obj.add(num);
 * boolean param_3 = obj.erase(num);
 */

```

---

32. [Find Elements in a Contaminated Binary Tree](#)



**Input**

```
["FindElements","find","find","find"]
[[[-1,-1,-1,-1,-1]],[1],[3],[5]]
```

**Output**

```
[null,true,true,false]
```

**Explanation**

```
FindElements findElements = new FindElements([-1,-1,-1,-1,-1]);
findElements.find(1); // return True
findElements.find(3); // return True
findElements.find(5); // return False
```

```
/*
 * Definition for a binary tree node.
 */
public class TreeNode {
    * int val;
    * TreeNode left;
    * TreeNode right;
    * TreeNode() {}
    * TreeNode(int val) { this.val = val; }
    * TreeNode(int val, TreeNode left, TreeNode right) {
        * this.val = val;
        * this.left = left;
        * this.right = right;
        * }
    * }
```

```

*/
class FindElements {
    private Set<Integer> values = new HashSet<>();

    public FindElements(TreeNode root) {
        recover(root, 0); // Start recovery with root value 0
    }

    private void recover(TreeNode node, int val) {
        if (node == null) return;

        node.val = val;
        values.add(val); // Store recovered value
        recover(node.left, 2 * val + 1);
        recover(node.right, 2 * val + 2);
    }

    public boolean find(int target) {
        return values.contains(target); // O(1) lookup
    }
}

/**
 * Your FindElements object will be instantiated and called as such:
 * FindElements obj = new FindElements(root);
 * boolean param_1 = obj.find(target);
 */

```

---

33. [Iterator for Combination](#)

```

Input
["CombinationIterator", "next", "hasNext", "next", "hasNext", "next", "hasNext"]
[["abc", 2], [], [], [], [], []]
Output
[null, "ab", true, "ac", true, "bc", false]

Explanation
CombinationIterator itr = new CombinationIterator("abc", 2);
itr.next();    // return "ab"
itr.hasNext(); // return True
itr.next();    // return "ac"
itr.hasNext(); // return True
itr.next();    // return "bc"
itr.hasNext(); // return False

```

```

class CombinationIterator {

    private List<String> combinations;
    private int index;

    public CombinationIterator(String characters, int combinationLength) {
        combinations = new ArrayList<>();
        generateCombinations(characters, combinationLength, 0, new StringBuilder());
        index = 0;
    }

    private void generateCombinations(String chars, int len, int start, StringBuilder path) {
        if (path.length() == len) {
            combinations.add(path.toString());
            return;
        }

        for (int i = start; i < chars.length(); i++) {
            path.append(chars.charAt(i));
            generateCombinations(chars, len, i + 1, path);
            path.deleteCharAt(path.length() - 1); // backtrack
        }
    }
}

```

```

public String next() {
    return combinations.get(index++);
}

public boolean hasNext() {
    return index < combinations.size();
}
}

```

#### 34. Apply Discount Every n Orders

**Input**

```
["Cashier","getBill","getBill","getBill","getBill","getBill","getBill"]
[[3,50,[1,2,3,4,5,6,7],[100,200,300,400,300,200,100]],[[1,2],[1,2]],[[3,7],[10,10]],
[[1,2,3,4,5,6,7],[1,1,1,1,1,1,1]],[[4],[10]],[[7,3],[10,10]],[[7,5,3,1,6,4,2],
[10,10,10,9,9,9,7]],[[2,3,5],[5,3,2]]]
```

**Output**

```
[null,500.0,4000.0,800.0,4000.0,4000.0,7350.0,2500.0]
```

**Explanation**

```
Cashier cashier = new Cashier(3,50,[1,2,3,4,5,6,7],[100,200,300,400,300,200,100]);
cashier.getBill([1,2],[1,2]);                                // return 500.0. 1st customer, no
discount.

cashier.getBill([3,7],[10,10]);                            // bill = 1 * 100 + 2 * 200 = 500.
discount.                                                 // return 4000.0. 2nd customer, no
discount.                                                 // bill = 10 * 300 + 10 * 100 = 4000.

cashier.getBill([1,2,3,4,5,6,7],[1,1,1,1,1,1,1]);      // return 800.0. 3rd customer, 50%
discount.                                                 // Original bill = 1600
                                                       // Actual bill = 1600 * ((100 - 50) /
100) = 800.

cashier.getBill([4],[10]);                                 // return 4000.0. 4th customer, no
discount.

cashier.getBill([7,3],[10,10]);                            // return 4000.0. 5th customer, no
discount.

cashier.getBill([7,5,3,1,6,4,2],[10,10,10,9,9,9,7]); // return 7350.0. 6th customer, 50%
discount.                                                 // Original bill = 14700, but with
                                                       // Actual bill = 14700 * ((100 - 50) /
100) = 7350.

cashier.getBill([2,3,5],[5,3,2]);                          // return 2500.0. 7th customer, no
discount.
```

```

class Cashier {
    private int n;
    private int discount;
    private Map<Integer, Integer> priceMap;
    private int customerCount;

    public Cashier(int n, int discount, int[] products, int[] prices) {
        this.n = n;
        this.discount = discount;
        this.priceMap = new HashMap<>();
        this.customerCount = 0;

        for (int i = 0; i < products.length; i++) {
            priceMap.put(products[i], prices[i]);
        }
    }

    public double getBill(int[] product, int[] amount) {
        customerCount++;
        double total = 0;

        for (int i = 0; i < product.length; i++) {
            total += priceMap.get(product[i]) * amount[i];
        }

        if (customerCount % n == 0) {
            total -= total * discount / 100.0;
        }

        return total;
    }
}

/**
 * Your Cashier object will be instantiated and called as such:

```

```

* Cashier obj = new Cashier(n, discount, products, prices);
* double param_1 = obj.getBill(product,amount);
*/

```

## 95. Design Underground System

### Input

```

["UndergroundSystem","checkIn","checkOut","getAverageTime","checkIn","checkOut","getAverageT
ime","checkIn","checkOut","getAverageTime"]
[],[10,"Leyton",3],[10,"Paradise",8],["Leyton","Paradise"],[5,"Leyton",10],
[5,"Paradise",16],["Leyton","Paradise"],[2,"Leyton",21],[2,"Paradise",30],
["Leyton","Paradise"]]

```

### Output

```
[null,null,null,5.00000,null,null,5.50000,null,null,6.66667]
```

### Explanation

```

UndergroundSystem undergroundSystem = new UndergroundSystem();
undergroundSystem.checkIn(10, "Leyton", 3);
undergroundSystem.checkOut(10, "Paradise", 8); // Customer 10 "Leyton" -> "Paradise" in 8-3
= 5
undergroundSystem.getAverageTime("Leyton", "Paradise"); // return 5.00000, (5) / 1 = 5
undergroundSystem.checkIn(5, "Leyton", 10);
undergroundSystem.checkOut(5, "Paradise", 16); // Customer 5 "Leyton" -> "Paradise" in 16-10
= 6
undergroundSystem.getAverageTime("Leyton", "Paradise"); // return 5.50000, (5 + 6) / 2 = 5.5
undergroundSystem.checkIn(2, "Leyton", 21);
undergroundSystem.checkOut(2, "Paradise", 30); // Customer 2 "Leyton" -> "Paradise" in 30-21
= 9
undergroundSystem.getAverageTime("Leyton", "Paradise"); // return 6.66667, (5 + 6 + 9) / 3 =
6.66667

```

```

class UndergroundSystem {
    private Map<Integer, Pair<String, Integer>> checkIns;
    private Map<String, Pair<Integer, Integer>> travelTimes;

    public UndergroundSystem() {
        checkIns = new HashMap<>();
        travelTimes = new HashMap<>();
    }

    public void checkIn(int id, String stationName, int t) {

```

```

        checkIns.put(id, new Pair<>(stationName, t));
    }

public void checkOut(int id, String stationName, int t) {
    Pair<String, Integer> checkInData = checkIns.remove(id);
    String routeKey = checkInData.getKey() + "#" + stationName;
    int duration = t - checkInData.getValue();

    travelTimes.putIfAbsent(routeKey, new Pair<>(0, 0));
    Pair<Integer, Integer> existing = travelTimes.get(routeKey);
    travelTimes.put(routeKey, new Pair<>(existing.getKey() + duration, existing.
}

public double getAverageTime(String startStation, String endStation) {
    String routeKey = startStation + "#" + endStation;
    Pair<Integer, Integer> data = travelTimes.get(routeKey);
    return (double) data.getKey() / data.getValue();
}

}

/***
 * Your UndergroundSystem object will be instantiated and called as such:
 * UndergroundSystem obj = new UndergroundSystem();
 * obj.checkIn(id,stationName,t);
 * obj.checkOut(id,stationName,t);
 * double param_3 = obj.getAverageTime(startStation,endStation);
 */

```

---

96. [First Unique Number](#)

```

Input:
["FirstUnique","showFirstUnique","add","showFirstUnique","add","showFirstUnique","add","showFirstUnique"]
[[[2,3,5]],[],[5],[],[2],[],[3],[]]
Output:
[null,2,null,2,null,3,null,-1]
Explanation:
FirstUnique firstUnique = new FirstUnique([2,3,5]);
firstUnique.showFirstUnique(); // return 2
firstUnique.add(5); // the queue is now [2,3,5,5]
firstUnique.showFirstUnique(); // return 2
firstUnique.add(2); // the queue is now [2,3,5,5,2]
firstUnique.showFirstUnique(); // return 3
firstUnique.add(3); // the queue is now [2,3,5,5,2,3]
firstUnique.showFirstUnique(); // return -1

```

```

class FirstUnique {
    private Map<Integer, Integer> cnt = new HashMap<>();
    private Set<Integer> unique = new LinkedHashSet<>();

    public FirstUnique(int[] nums) {
        for (int v : nums) {
            cnt.put(v, cnt.getOrDefault(v, 0) + 1);
        }
        for (int v : nums) {
            if (cnt.get(v) == 1) {
                unique.add(v);
            }
        }
    }

    public int showFirstUnique() {
        return unique.isEmpty() ? -1 : unique.iterator().next();
    }

    public void add(int value) {
        cnt.put(value, cnt.getOrDefault(value, 0) + 1);
        if (cnt.get(value) == 1) {
            unique.add(value);
        } else {
            unique.remove(value);
        }
    }
}

```

```
    }
}
```

---

## 97. Subrectangle Queries

```
Input
["SubrectangleQueries","getValue","updateSubrectangle","getValue","getValue","updateSubrectangle","getValue","getValue"]
[[[[1,2,1],[4,3,4],[3,2,1],[1,1,1]]],[0,2],[0,0,3,2,5],[0,2],[3,1],[3,0,3,2,10],[3,1],[0,2]]
Output
[null,1,null,5,5,null,10,5]
Explanation
SubrectangleQueries subrectangleQueries = new SubrectangleQueries([[1,2,1],[4,3,4],[3,2,1],[1,1,1]]);
// The initial rectangle (4x3) looks like:
// 1 2 1
// 4 3 4
// 3 2 1
// 1 1 1
subrectangleQueries.getValue(0, 2); // return 1
subrectangleQueries.updateSubrectangle(0, 0, 3, 2, 5);
// After this update the rectangle looks like:
// 5 5 5
// 5 5 5
// 5 5 5
// 5 5 5
subrectangleQueries.getValue(0, 2); // return 5
subrectangleQueries.getValue(3, 1); // return 5
subrectangleQueries.updateSubrectangle(3, 0, 3, 2, 10);
// After this update the rectangle looks like:
// 5 5 5
// 5 5 5
// 5 5 5
// 10 10 10
subrectangleQueries.getValue(3, 1); // return 10
subrectangleQueries.getValue(0, 2); // return 5
```

```
class SubrectangleQueries {
    private int[][] rectangle;

    public SubrectangleQueries(int[][][] rectangle) {
        this.rectangle = rectangle;
    }
```

```
public void updateSubrectangle(int row1, int col1, int row2, int col2, int newValue) {
    for (int i = row1; i <= row2; i++) {
        for (int j = col1; j <= col2; j++) {
            rectangle[i][j] = newValue;
        }
    }
}

public int getValue(int row, int col) {
    return rectangle[row][col];
}
```

---

38. [Dot Product of Two Sparse Vectors](#)

### Example 1:

```
Input: nums1 = [1,0,0,2,3], nums2 = [0,3,0,4,0]
Output: 8
Explanation: v1 = SparseVector(nums1) , v2 = SparseVector(nums2)
v1.dotProduct(v2) = 1*0 + 0*3 + 0*0 + 2*4 + 3*0 = 8
```

### Example 2:

```
Input: nums1 = [0,1,0,0,0], nums2 = [0,0,0,0,2]
Output: 0
Explanation: v1 = SparseVector(nums1) , v2 = SparseVector(nums2)
v1.dotProduct(v2) = 0*0 + 1*0 + 0*0 + 0*0 + 0*2 = 0
```

### Example 3:

```
Input: nums1 = [0,1,0,0,2,0,0], nums2 = [1,0,0,0,3,0,4]
Output: 6
```

```
class SparseVector {
    private Map<Integer, Integer> indexValueMap = new HashMap<>(128);

    public SparseVector(int[] nums) {
        for (int index = 0; index < nums.length; ++index) {
            if (nums[index] != 0) {
                indexValueMap.put(index, nums[index]);
            }
        }
    }

    // Return the dot product of two sparse vectors
    public int dotProduct(SparseVector otherVector) {
        Map<Integer, Integer> smallerMap = this.indexValueMap;
        Map<Integer, Integer> largerMap = otherVector.indexValueMap;
```

```
// Always iterate over the smaller map for efficiency
if (largerMap.size() < smallerMap.size()) {
    Map<Integer, Integer> temp = smallerMap;
    smallerMap = largerMap;
    largerMap = temp;
}

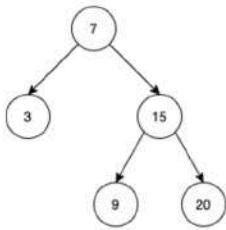
int dotProductSum = 0;
for (Map.Entry<Integer, Integer> entry : smallerMap.entrySet()) {
    int index = entry.getKey();
    int value = entry.getValue();
    dotProductSum += value * largerMap.getOrDefault(index, 0);
}

return dotProductSum;
}
}
```

---

99. [Binary Search Tree Iterator II](#)

**Example 1:**



**Input**

```
["BSTIterator", "next", "next", "prev", "next", "hasNext", "next", "next", "hasNext", "hasPrev", "prev", "prev"]
[[[7, 3, 15, null, null, 9, 20]], [null], [null], [null], [null], [null], [null], [null], [null], [null], [null]]
```

**Output**

```
[null, 3, 7, 3, 7, true, 9, 15, 20, false, true, 15, 9]
```

**Explanation**

```
// The underlined element is where the pointer currently is.
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]); // state is _ [3, 7, 9, 15, 20]
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 3
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 7
bSTIterator.prev(); // state becomes [3, 7, 9, 15, 20], return 3
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 7
bSTIterator.hasNext(); // return true
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 9
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 15
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 20
bSTIterator.hasNext(); // return false
bSTIterator.hasPrev(); // return true
bSTIterator.prev(); // state becomes [3, 7, 9, 15, 20], return 15
bSTIterator.prev(); // state becomes [3, 7, 9, 15, 20], return 9
```

```
class BSTIterator {
    private List<Integer> nums = new ArrayList<>();
    private int i = -1;

    public BSTIterator(TreeNode root) {
        dfs(root);
    }

    public boolean hasNext() {
        return i < nums.size() - 1;
    }

    public int next() {
        return nums.get(++i);
    }

    public boolean hasPrev() {
        return i > 0;
    }
}
```

```
}

public int prev() {
    return nums.get(--i);
}

private void dfs(TreeNode root) {
    if (root == null) {
        return;
    }
    dfs(root.left);
    nums.add(root.val);
    dfs(root.right);
}
}

/**
 * Your BSTIterator object will be instantiated and called as such:
 * BSTIterator obj = new BSTIterator(root);
 * boolean param_1 = obj.hasNext();
 * int param_2 = obj.next();
 * boolean param_3 = obj.hasPrev();
 * int param_4 = obj.prev();
 */

```

---

20. [Throne Inheritance](#)

```

Input
["ThroneInheritance", "birth", "birth", "birth", "birth", "birth", "birth",
"getInheritanceOrder", "death", "getInheritanceOrder"]
[[["king"], ["king", "andy"], ["king", "bob"], ["king", "catherine"], ["andy", "matthew"],
["bob", "alex"], ["bob", "asha"], [null], ["bob"], [null]]]
Output
[null, null, null, null, null, null, ["king", "andy", "matthew", "bob", "alex",
"asha", "catherine"], null, ["king", "andy", "matthew", "alex", "asha", "catherine"]]

```

#### **Explanation**

```

ThroneInheritance t= new ThroneInheritance("king"); // order: king
t.birth("king", "andy"); // order: king > andy
t.birth("king", "bob"); // order: king > andy > bob
t.birth("king", "catherine"); // order: king > andy > bob > catherine
t.birth("andy", "matthew"); // order: king > andy > matthew > bob > catherine
t.birth("bob", "alex"); // order: king > andy > matthew > bob > alex > catherine
t.birth("bob", "asha"); // order: king > andy > matthew > bob > alex > asha > catherine
t.getInheritanceOrder(); // return ["king", "andy", "matthew", "bob", "alex", "asha",
"catherine"]
t.death("bob"); // order: king > andy > matthew > bob > alex > asha > catherine
t.getInheritanceOrder(); // return ["king", "andy", "matthew", "alex", "asha", "catherine"]

```

```

class ThroneInheritance {
    public ThroneInheritance(String kingName) {
        this.kingName = kingName;
    }

    public void birth(String parentName, String childName) {
        family.putIfAbsent(parentName, new ArrayList<>());
        family.get(parentName).add(childName);
    }

    public void death(String name) {
        dead.add(name);
    }

    public List<String> getInheritanceOrder() {
        List<String> ans = new ArrayList<>();
        dfs(kingName, ans);
        return ans;
    }
}

```

```
private Set<String> dead = new HashSet<>();
private Map<String, List<String>> family = new HashMap<>();
private String kingName;

private void dfs(final String name, List<String> ans) {
    if (!dead.contains(name))
        ans.add(name);
    if (!family.containsKey(name))
        return;

    for (final String child : family.get(name))
        dfs(child, ans);
}

/**
 * Your ThroneInheritance object will be instantiated and called as such:
 * ThroneInheritance obj = new ThroneInheritance(kingName);
 * obj.birth(parentName,childName);
 * obj.death(name);
 * List<String> param_3 = obj.getInheritanceOrder();
 */
```

---

01. [Design Parking System](#)

```

Input
["ParkingSystem", "addCar", "addCar", "addCar", "addCar"]
[[1, 1, 0], [1], [2], [3], [1]]
Output
[null, true, true, false, false]

Explanation
ParkingSystem parkingSystem = new ParkingSystem(1, 1, 0);
parkingSystem.addCar(1); // return true because there is 1 available slot for a big car
parkingSystem.addCar(2); // return true because there is 1 available slot for a medium car
parkingSystem.addCar(3); // return false because there is no available slot for a small car
parkingSystem.addCar(1); // return false because there is no available slot for a big car.
It is already occupied.

```

```

class ParkingSystem {
    private int big;
    private int medium;
    private int small;

    // Constructor initializes the parking spots for each car type
    public ParkingSystem(int big, int medium, int small) {
        this.big = big;
        this.medium = medium;
        this.small = small;
    }

    // Method to park the car based on its type
    public boolean addCar(int carType) {
        // If the car type is 1 (Big), try to park it in a big spot
        if (carType == 1) {
            if (big > 0) {
                big--; // Decrement available big spots
                return true;
            }
        }
        // If the car type is 2 (Medium), try to park it in a medium spot
        if (carType == 2) {
            if (medium > 0) {
                medium--; // Decrement available medium spots
                return true;
            }
        }
        // If the car type is 3 (Small), try to park it in a small spot
        if (carType == 3) {
            if (small > 0) {
                small--; // Decrement available small spots
                return true;
            }
        }
    }
}

```

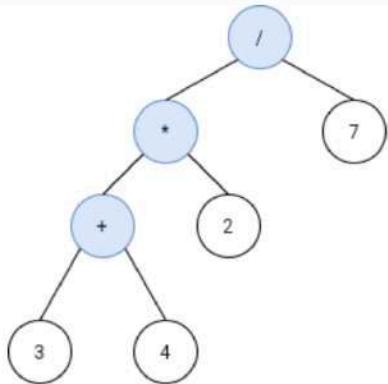
```
    }
}

// If the car type is 3 (Small), try to park it in a small spot
if (carType == 3) {
    if (small > 0) {
        small--; // Decrement available small spots
        return true;
    }
}

// If no spots are available for the car type, return false
return false;
}
}
```

---

## 02. Design an Expression Tree With Evaluate Function

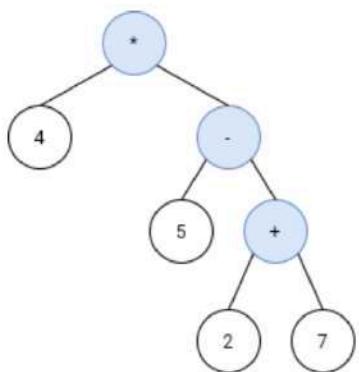


**Input:** s = ["3", "4", "+", "2", "\*", "7", "/"]

**Output:** 2

**Explanation:** this expression evaluates to the above binary tree with expression  $((3+4)*7)$

### Example 2:



**Input:** s = ["4", "5", "2", "7", "+", "-", "\*"]

**Output:** -16

**Explanation:** this expression evaluates to the above binary tree with expression  $4*(5-(2+7))$

```
abstract class Node {
    public abstract int evaluate();
    protected String val;
    protected Node left;
    protected Node right;
};
```

```
class MyNode extends Node {  
    public MyNode(String val) {  
        this.val = val;  
    }  
  
    public int evaluate() {  
        if (isNumeric()) {  
            return Integer.parseInt(val);  
        }  
        int leftVal = left.evaluate();  
        int rightVal = right.evaluate();  
        if (Objects.equals(val, "+")) {  
            return leftVal + rightVal;  
        }  
        if (Objects.equals(val, "-")) {  
            return leftVal - rightVal;  
        }  
        if (Objects.equals(val, "*")) {  
            return leftVal * rightVal;  
        }  
        if (Objects.equals(val, "/")) {  
            return leftVal / rightVal;  
        }  
        return 0;  
    }  
  
    public boolean isNumeric() {  
        for (char c : val.toCharArray()) {  
            if (!Character.isDigit(c)) {  
                return false;  
            }  
        }  
        return true;  
    }  
}
```

```

/**
 * This is the TreeBuilder class.
 * You can treat it as the driver code that takes the postinfix input
 * and returns the expression tree represnting it as a Node.
 */

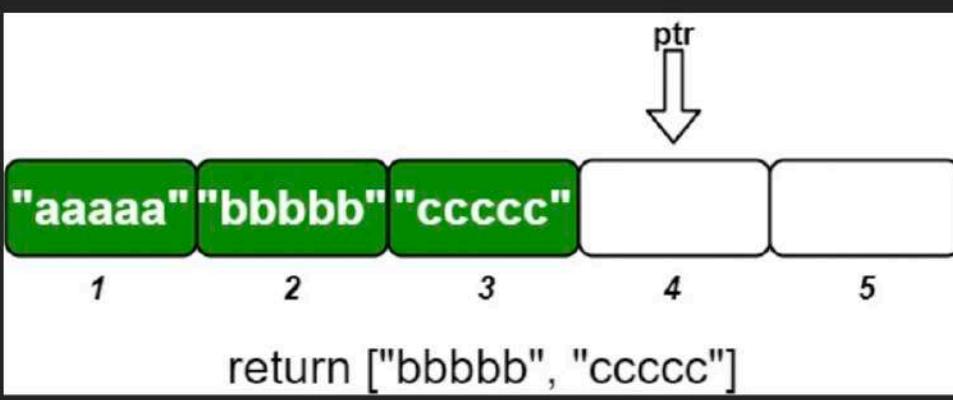
class TreeBuilder {
    Node buildTree(String[] postfix) {
        Deque<MyNode> stk = new ArrayDeque<>();
        for (String s : postfix) {
            MyNode node = new MyNode(s);
            if (!node.isNumeric()) {
                node.right = stk.pop();
                node.left = stk.pop();
            }
            stk.push(node);
        }
        return stk.peek();
    }
};

/** 
 * Your TreeBuilder object will be instantiated and called as such:
 * TreeBuilder obj = new TreeBuilder();
 * Node expTree = obj.buildTree(postfix);
 * int ans = expTree.evaluate();
 */

```

---

03. Design an Ordered Stream\$



#### Input

```
["OrderedStream", "insert", "insert", "insert", "insert", "insert"]
[[5], [3, "cccccc"], [1, "aaaaaa"], [2, "bbbbbb"], [5, "eeeeee"], [4, "dddddd"]]
```

#### Output

```
[null, [], ["aaaaaa"], ["bbbbbb", "cccccc"], [], ["dddddd", "eeeeee"]]
```

#### Explanation

```
// Note that the values ordered by ID is ["aaaaaa", "bbbbbb", "cccccc", "dddddd", "eeeeee"].
OrderedStream os = new OrderedStream(5);
os.insert(3, "cccccc"); // Inserts (3, "cccccc"), returns [].
os.insert(1, "aaaaaa"); // Inserts (1, "aaaaaa"), returns ["aaaaaa"].
os.insert(2, "bbbbbb"); // Inserts (2, "bbbbbb"), returns ["bbbbbb", "cccccc"].
os.insert(5, "eeeeee"); // Inserts (5, "eeeeee"), returns [].
os.insert(4, "dddddd"); // Inserts (4, "dddddd"), returns ["dddddd", "eeeeee"].
// Concatenating all the chunks returned:
// [] + ["aaaaaa"] + ["bbbbbb", "cccccc"] + [] + ["dddddd", "eeeeee"] = ["aaaaaa", "bbbbbb",
// "cccccc", "dddddd", "eeeeee"]
// The resulting order is the same as the order above.
```

```
class OrderedStream {
    private String[] stream;
    private int ptr;

    public OrderedStream(int n) {
        stream = new String[n];
        ptr = 0;
    }

    public List<String> insert(int idKey, String value) {
        stream[idKey - 1] = value; // place value at correct index
        List<String> result = new ArrayList<>();
        for (int i = ptr; i < stream.length; i++) {
            if (stream[i] != null) {
                result.add(stream[i]);
            } else {
                break;
            }
        }
        return result;
    }
}
```

```

        while (ptr < stream.length && stream[ptr] != null) {
            result.add(stream[ptr]);
            ptr++;
        }

        return result;
    }
}

/**
 * Your OrderedStream object will be instantiated and called as such:
 * OrderedStream obj = new OrderedStream(n);
 * List<String> param_1 = obj.insert(idKey,value);
 */

```

#### 04. Design Front Middle Back Queue

**Input:**  
["FrontMiddleBackQueue", "pushFront", "pushBack", "pushMiddle", "pushMiddle", "popFront",  
"popMiddle", "popMiddle", "popBack", "popFront"]  
[], [1], [2], [3], [4], [], [], [], []]  
**Output:**  
[null, null, null, null, null, 1, 3, 4, 2, -1]

**Explanation:**  
FrontMiddleBackQueue q = new FrontMiddleBackQueue();  
q.pushFront(1); // [1]  
q.pushBack(2); // [1, 2]  
q.pushMiddle(3); // [1, 3, 2]  
q.pushMiddle(4); // [1, 4, 3, 2]  
q.popFront(); // return 1 -> [4, 3, 2]  
q.popMiddle(); // return 3 -> [4, 2]  
q.popMiddle(); // return 4 -> [2]  
q.popBack(); // return 2 -> []  
q.popFront(); // return -1 -> [] (The queue is empty)

```

class FrontMiddleBackQueue {
    Deque<Integer> left = new ArrayDeque<>();
    Deque<Integer> right = new ArrayDeque<>();
}

```

```

private void rebalance() {
    while (left.size() > right.size() + 1) {
        right.addFirst(left.removeLast());
    }
    while (left.size() < right.size()) {
        left.addLast(right.removeFirst());
    }
}

public void pushFront(int val) {
    left.addFirst(val);
    rebalance();
}

public void pushMiddle(int val) {
    if (left.size() > right.size()) {
        right.addFirst(left.removeLast());
    }
    left.addLast(val);
}

public void pushBack(int val) {
    right.addLast(val);
    rebalance();
}

public int popFront() {
    if (isEmpty()) return -1;
    int val = left.isEmpty() ? right.removeFirst() : left.removeFirst();
    rebalance();
    return val;
}

public int popMiddle() {
    if (isEmpty()) return -1;
    int val = left.removeLast();
}

```

```

        rebalance();
        return val;
    }

    public int popBack() {
        if (isEmpty()) return -1;
        int val = right.isEmpty() ? left.removeLast() : right.removeLast();
        rebalance();
        return val;
    }

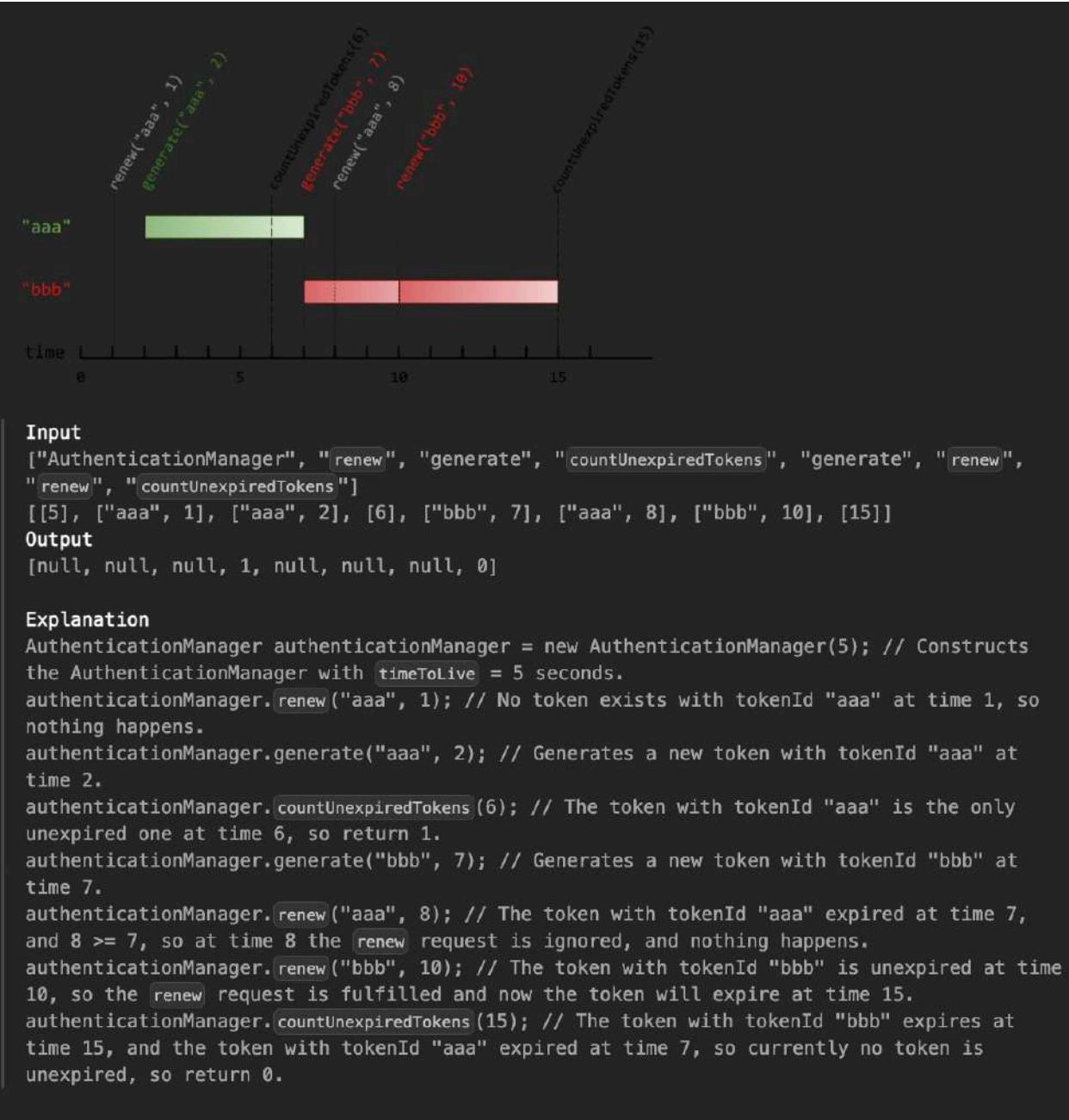
    private boolean isEmpty() {
        return left.isEmpty() && right.isEmpty();
    }
}

/**
 * Your FrontMiddleBackQueue object will be instantiated and called as such:
 * FrontMiddleBackQueue obj = new FrontMiddleBackQueue();
 * obj.pushFront(val);
 * obj.pushMiddle(val);
 * obj.pushBack(val);
 * int param_4 = obj.popFront();
 * int param_5 = obj.popMiddle();
 * int param_6 = obj.popBack();
 */

```

---

25. [Design Authentication Manager](#)



```
class AuthenticationManager {
    private int timeToLive;
    private Map<String, Integer> tokenExpiryMap;

    public AuthenticationManager(int timeToLive) {
        this.timeToLive = timeToLive;
        this.tokenExpiryMap = new HashMap<>();
```

```
}

public void generate(String tokenId, int currentTime) {
    tokenExpiryMap.put(tokenId, currentTime + timeToLive);
}

public void renew(String tokenId, int currentTime) {
    if (tokenExpiryMap.containsKey(tokenId) && tokenExpiryMap.get(tokenId) >
        tokenExpiryMap.put(tokenId, currentTime + timeToLive);
    }
}

public int countUnexpiredTokens(int currentTime) {
    int count = 0;
    for (int expiry : tokenExpiryMap.values()) {
        if (expiry > currentTime) {
            count++;
        }
    }
    return count;
}
}
```

---

26. [Implement Trie II \(Prefix Tree\)](#)

The problem involves implementing a class for a [trie](#), also known as a prefix tree. A trie is a specialized tree used to manage a set of strings in such a way that queries like insertion, search, prefix counting, and deletion can be performed efficiently. The specific operations to implement in this Trie class are:

- **Trie**: Initialize the trie.
- **insert**: Add a string `word` to the trie.
- **countWordsEqualTo**: Determine how many times a specific string `word` appears in the trie.
- **countWordsStartingWith**: Count the number of strings in the trie that start with a given `prefix`.
- **erase**: Remove a string `word` from the trie.

The goal is to handle these operations as efficiently as possible.

```
class Trie {  
    private Trie[] children; // array representing the children nodes of the current r  
    private int wordCount; // number of times a complete word terminates at this n  
    private int prefixCount; // number of words sharing the prefix that ends at this  
  
    public Trie() {  
        children = new Trie[26];  
        wordCount = 0;  
        prefixCount = 0;  
    }  
  
    public void insert(String word) {  
        Trie node = this;  
        for (char c : word.toCharArray()) {  
            int index = c - 'a';  
            if (node.children[index] == null) {  
                node.children[index] = new Trie();  
            }  
            node = node.children[index];  
            node.prefixCount++; // Increment the prefix count for each character  
        }  
        node.wordCount++; // Once the end of the word is reached, increment the v  
    }  
}
```

```

public int countWordsEqualTo(String word) {
    Trie node = search(word);
    return node == null ? 0 : node.wordCount;
}

public int countWordsStartingWith(String prefix) {
    Trie node = search(prefix);
    return node == null ? 0 : node.prefixCount;
}

public void erase(String word) {
    Trie node = this;
    for (char c : word.toCharArray()) {
        int index = c - 'a';
        node = node.children[index];
        node.prefixCount--;
    }
    node.wordCount--;
}

private Trie search(String word) {
    Trie node = this; // Start from the root node
    for (char c : word.toCharArray()) {
        int index = c - 'a'; // Convert char to index (0-25)
        if (node.children[index] == null) {
            return null; // If at any point the node doesn't have a required child, return
        }
        node = node.children[index]; // Otherwise, move to the child node
    }
    return node; // Return the node where the word or prefix ends
}

```

---

08. Seat Reservation Manager

```

Input
["SeatManager", "reserve", "reserve", "unreserve", "reserve", "reserve", "reserve",
"reserve", "unreserve"]
[[5], [], [], [2], [], [], [], [5]]
Output
[null, 1, 2, null, 2, 3, 4, 5, null]

Explanation
SeatManager seatManager = new SeatManager(5); // Initializes a SeatManager with 5 seats.
seatManager.reserve(); // All seats are available, so return the lowest numbered seat,
which is 1.
seatManager.reserve(); // The available seats are [2,3,4,5], so return the lowest of
them, which is 2.
seatManager.unreserve(2); // Unreserve seat 2, so now the available seats are [2,3,4,5].
seatManager.reserve(); // The available seats are [2,3,4,5], so return the lowest of
them, which is 2.
seatManager.reserve(); // The available seats are [3,4,5], so return the lowest of them,
which is 3.
seatManager.reserve(); // The available seats are [4,5], so return the lowest of them,
which is 4.
seatManager.reserve(); // The only available seat is seat 5, so return 5.
seatManager.unreserve(5); // Unreserve seat 5, so now the available seats are [5].

```

```

import java.util.PriorityQueue;

class SeatManager {
    private PriorityQueue<Integer> availableSeats;

    public SeatManager(int n) {
        availableSeats = new PriorityQueue<>();
        for (int seat = 1; seat <= n; seat++) {
            availableSeats.offer(seat);
        }
    }

    public int reserve() {
        return availableSeats.poll(); // Always gives the smallest available seat
    }

    public void unreserve(int seatNumber) {
        availableSeats.offer(seatNumber); // Add back the seat to available pool
    }
}

```

```

    }
}

/***
 * Your SeatManager object will be instantiated and called as such:
 * SeatManager obj = new SeatManager(n);
 * int param_1 = obj.reserve();
 * obj.unreserve(seatNumber);
 */

```

## 09. Finding Pairs With a Certain Sum

### Input

```
["FindSumPairs", "count", "add", "count", "count", "add", "add", "count"]
[[[1, 1, 2, 2, 2, 3], [1, 4, 5, 2, 5, 4]], [7], [3, 2], [8], [4], [0, 1], [1, 1], [7]]
```

### Output

```
[null, 8, null, 2, 1, null, null, 11]
```

### Explanation

```
FindSumPairs findSumPairs = new FindSumPairs([1, 1, 2, 2, 2, 3], [1, 4, 5, 2, 5, 4]);
findSumPairs.count(7); // return 8; pairs (2,2), (3,2), (4,2), (2,4), (3,4), (4,4) make 2 + 5 and pairs (5,1), (5,5) make 3 + 4
findSumPairs.add(3, 2); // now nums2 = [1,4,5,4,5,4]
findSumPairs.count(8); // return 2; pairs (5,2), (5,4) make 3 + 5
findSumPairs.count(4); // return 1; pair (5,0) makes 3 + 1
findSumPairs.add(0, 1); // now nums2 = [2,4,5,4,5,4]
findSumPairs.add(1, 1); // now nums2 = [2,5,5,4,5,4]
findSumPairs.count(7); // return 11; pairs (2,1), (2,2), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,4) make 2 + 5 and pairs (5,3), (5,5) make 3 + 4
```

```

class FindSumPairs {
    private int[] nums1;
    private int[] nums2;
    private Map<Integer, Integer> freqMap2;

    public FindSumPairs(int[] nums1, int[] nums2) {
        this.nums1 = nums1;
        this.nums2 = nums2;
        this.freqMap2 = new HashMap<>();
        for (int num : nums2) {

```

```

        freqMap2.put(num, freqMap2.getOrDefault(num, 0) + 1);
    }
}

public void add(int index, int val) {
    int oldVal = nums2[index];
    freqMap2.put(oldVal, freqMap2.getOrDefault(oldVal, 0) - 1);
    nums2[index] += val;
    freqMap2.put(nums2[index], freqMap2.getOrDefault(nums2[index], 0) + 1);
}

public int count(int tot) {
    int result = 0;
    for (int a : nums1) {
        int complement = tot - a;
        result += freqMap2.getOrDefault(complement, 0);
    }
    return result;
}

}

/***
 * Your FindSumPairs object will be instantiated and called as such:
 * FindSumPairs obj = new FindSumPairs(nums1, nums2);
 * obj.add(index,val);
 * int param_2 = obj.count(tot);
 */

```

---

10. [Design Movie Rental System](#)

```

Input
["MovieRentingSystem", "search", "rent", "rent", "report", "drop", "search"]
[[3, [[0, 1, 5], [0, 2, 6], [0, 3, 7], [1, 1, 4], [1, 2, 7], [2, 1, 5]]], [1], [0, 1], [1,
2], [], [1, 2], [2]]
Output
[null, [1, 0, 2], null, null, [[0, 1], [1, 2]], null, [0, 1]]

Explanation
MovieRentingSystem movieRentingSystem = new MovieRentingSystem(3, [[0, 1, 5], [0, 2, 6], [0,
3, 7], [1, 1, 4], [1, 2, 7], [2, 1, 5]]);
movieRentingSystem.search(1); // return [1, 0, 2], Movies of ID 1 are unrented at shops 1,
0, and 2. Shop 1 is cheapest; shop 0 and 2 are the same price, so order by shop number.
movieRentingSystem.rent(0, 1); // Rent movie 1 from shop 0. Unrented movies at shop 0 are
now [2,3].
movieRentingSystem.rent(1, 2); // Rent movie 2 from shop 1. Unrented movies at shop 1 are
now [1].
movieRentingSystem.report(); // return [[0, 1], [1, 2]]. Movie 1 from shop 0 is cheapest,
followed by movie 2 from shop 1.
movieRentingSystem.drop(1, 2); // Drop off movie 2 at shop 1. Unrented movies at shop 1 are
now [1,2].
movieRentingSystem.search(2); // return [0, 1]. Movies of ID 2 are unrented at shops 0 and
1. Shop 0 is cheapest, followed by shop 1.

```

```

class MovieRentingSystem {
    public MovieRentingSystem(int n, int[][][] entries) {
        for (int[] e : entries) {
            final int shop = e[0];
            final int movie = e[1];
            final int price = e[2];
            unrented.putIfAbsent(movie, new TreeSet<>(comparator));
            unrented.get(movie).add(new Entry(price, shop, movie));
            shopAndMovieToPrice.put(new Pair<>(shop, movie), price);
        }
    }

    public List<Integer> search(int movie) {
        return unrented.getOrDefault(movie, Collections.emptySet())
            .stream()
            .limit(5)
            .map(e → e.shop)
            .collect(Collectors.toList());
    }
}

```

```

public void rent(int shop, int movie) {
    final int price = shopAndMovieToPrice.get(new Pair<>(shop, movie));
    unrented.get(movie).remove(new Entry(price, shop, movie));
    rented.add(new Entry(price, shop, movie));
}

public void drop(int shop, int movie) {
    final int price = shopAndMovieToPrice.get(new Pair<>(shop, movie));
    unrented.get(movie).add(new Entry(price, shop, movie));
    rented.remove(new Entry(price, shop, movie));
}

public List<List<Integer>> report() {
    return rented.stream().limit(5).map(e → List.of(e.shop, e.movie)).collect(Collectors.toList());
}

private record Entry(int price, int shop, int movie) {}

private Comparator<Entry> comparator = Comparator.comparingInt(Entry::price)
    .thenComparingInt(Entry::shop)
    .thenComparingInt(Entry::movie);

// {movie: (price, shop)}
private Map<Integer, Set<Entry>> unrented = new HashMap<>();

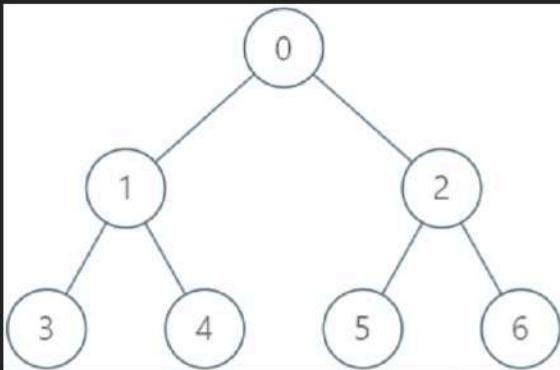
// {(shop, movie): price}
private Map<Pair<Integer, Integer>, Integer> shopAndMovieToPrice = new HashMap<>();

// (price, shop, movie)
private Set<Entry> rented = new TreeSet<>(comparator);
}

```

---

10. Operations on Tree



**Input**

```
["LockingTree", "lock", "unlock", "unlock", "lock", "upgrade", "lock"]
[[[-1, 0, 0, 1, 1, 2, 2]], [2, 2], [2, 3], [2, 2], [4, 5], [0, 1], [0, 1]]
```

**Output**

```
[null, true, false, true, true, true, false]
```

**Explanation**

```
LockingTree lockingTree = new LockingTree([-1, 0, 0, 1, 1, 2, 2]);
lockingTree.lock(2, 2);      // return true because node 2 is unlocked.
                           // Node 2 will now be locked by user 2.
lockingTree.unlock(2, 3);  // return false because user 3 cannot unlock a node locked by
                           // user 2.
lockingTree.unlock(2, 2);  // return true because node 2 was previously locked by user 2.
                           // Node 2 will now be unlocked.
lockingTree.lock(4, 5);   // return true because node 4 is unlocked.
                           // Node 4 will now be locked by user 5.
lockingTree.upgrade(0, 1); // return true because node 0 is unlocked and has at least one
                           // locked descendant (node 4).
                           // Node 0 will now be locked by user 1 and node 4 will now be
                           // unlocked.
lockingTree.lock(0, 1);   // return false because node 0 is already locked.
```

```
class LockingTree {

    private int[] parent;
    private Map<Integer, List<Integer>> tree = new HashMap<>();
    private Map<Integer, Integer> lockedBy = new HashMap<>();

    public LockingTree(int[] parent) {
        this.parent = parent;
        int n = parent.length;
        for (int i = 0; i < n; i++) {
            tree.computeIfAbsent(parent[i], k → new ArrayList<>()).add(i);
        }
    }

    public boolean lock(int node, int user) {
        if (lockedBy.containsKey(node)) {
            return false;
        }
        parent[node] = user;
        lockedBy.put(node, user);
        return true;
    }

    public boolean unlock(int node, int user) {
        if (!lockedBy.containsKey(node)) {
            return false;
        }
        if (parent[node] != user) {
            return false;
        }
        parent[node] = -1;
        lockedBy.remove(node);
        return true;
    }

    public boolean upgrade(int startNode, int endUser) {
        if (lockedBy.containsKey(startNode)) {
            return false;
        }
        if (parent[startNode] == -1) {
            return false;
        }
        if (parent[startNode] != endUser) {
            return false;
        }
        List<Integer> descendants = tree.get(startNode);
        for (int descendant : descendants) {
            if (parent[descendant] != -1) {
                return false;
            }
        }
        parent[startNode] = endUser;
        lockedBy.put(startNode, endUser);
        return true;
    }
}
```

```

}

public boolean lock(int num, int user) {
    if (lockedBy.containsKey(num)) return false;
    lockedBy.put(num, user);
    return true;
}

public boolean unlock(int num, int user) {
    if (!lockedBy.containsKey(num) || lockedBy.get(num) != user) return false;
    lockedBy.remove(num);
    return true;
}

public boolean upgrade(int num, int user) {
    if (lockedBy.containsKey(num)) return false;
    if (hasLockedAncestor(num)) return false;
    if (!hasLockedDescendant(num)) return false;

    // Unlock all locked descendants
    unlockAllDescendants(num);
    lockedBy.put(num, user);
    return true;
}

private boolean hasLockedAncestor(int num) {
    int p = parent[num];
    while (p != -1) {
        if (lockedBy.containsKey(p)) return true;
        p = parent[p];
    }
    return false;
}

private boolean hasLockedDescendant(int num) {
    if (lockedBy.containsKey(num)) return true;
}

```

```

if (!tree.containsKey(num)) return false;

for (int child : tree.get(num)) {
    if (hasLockedDescendant(child)) return true;
}
return false;
}

private void unlockAllDescendants(int num) {
    if (lockedBy.containsKey(num)) {
        lockedBy.remove(num);
    }
    if (!tree.containsKey(num)) return;

    for (int child : tree.get(num)) {
        unlockAllDescendants(child);
    }
}
}

/**
 * Your LockingTree object will be instantiated and called as such:
 * LockingTree obj = new LockingTree(parent);
 * boolean param_1 = obj.lock(num,user);
 * boolean param_2 = obj.unlock(num,user);
 * boolean param_3 = obj.upgrade(num,user);
 */

```

---

11. [Detect Squares](#)

The figure consists of six sub-diagrams arranged in a 2x3 grid, each showing a 2D coordinate system with x and y axes ranging from 0 to 15. Points are plotted as colored circles: blue, purple, teal, and red.

- Top Left:** Shows points at (3, 10), (11, 2), and (3, 2). A legend indicates: `detectSquares.add([3, 10]);` (blue dot), `detectSquares.add([11, 2]);` (purple dot), and `detectSquares.add([3, 2]);` (teal dot).
- Top Middle:** Shows points at (3, 10), (11, 2), and (11, 10). A legend indicates: `detectSquares.count([11, 10]);` (orange square).
- Top Right:** Shows points at (3, 10), (14, 8), and (11, 2). A legend indicates: `detectSquares.count([14, 8]);` (orange square).
- Bottom Left:** Shows points at (3, 10), (11, 2), and (11, 10). A legend indicates: `detectSquares.add([11, 2]);` (red dot).
- Bottom Middle:** Shows points at (3, 10), (11, 2), (11, 10), and (10, 2). A legend indicates: `detectSquares.count([11, 10]);` (orange square). Annotations explain: "choose the first, second, and third points".
- Bottom Right:** Shows points at (3, 10), (14, 8), (11, 2), and (11, 10). A legend indicates: `detectSquares.count([11, 10]);` (orange square). Annotations explain: "choose the first, third, and fourth points".

```

Input
["DetectSquares", "add", "add", "add", "count", "count", "add", "count"]
[[], [[3, 10]], [[11, 2]], [[3, 2]], [[11, 10]], [[14, 8]], [[11, 2]], [[11, 10]]]
Output
[null, null, null, null, 1, 0, null, 2]

Explanation
DetectSquares detectSquares = new DetectSquares();
detectSquares.add([3, 10]);
detectSquares.add([11, 2]);
detectSquares.add([3, 2]);
detectSquares.count([11, 10]); // return 1. You can choose:
                           // - The first, second, and third points
detectSquares.count([14, 8]); // return 0. The query point cannot form a square with any points in the
                           // data structure.
detectSquares.add([11, 2]);   // Adding duplicate points is allowed.
detectSquares.count([11, 10]); // return 2. You can choose:
                           // - The first, second, and third points
                           // - The first, third, and fourth points

```

```

class DetectSquares {
    private Map<Integer, Map<Integer, Integer>> pointsCount;

    public DetectSquares() {
        pointsCount = new HashMap<>();
    }
}

```

```

public void add(int[] point) {
    int x = point[0], y = point[1];
    pointsCount.putIfAbsent(y, new HashMap<>());
    pointsCount.get(y).put(x, pointsCount.get(y).getOrDefault(x, 0) + 1);
}

public int count(int[] point) {
    int x1 = point[0], y1 = point[1];
    int totalSquares = 0;

    if (!pointsCount.containsKey(y1)) return 0;

    // Iterate over all different y2 where there is at least one point (x, y2)
    for (int y2 : pointsCount.keySet()) {
        if (y2 == y1) continue;
        int dist = Math.abs(y2 - y1); // Side length of the square

        // Check for two possible x2: x1 - dist and x1 + dist
        for (int dx : new int[]{dist, -dist}) {
            int x2 = x1 + dx;

            int c1 = pointsCount.getOrDefault(y2, new HashMap<>()).getOrDefault(x2, 0);
            int c2 = pointsCount.getOrDefault(y1, new HashMap<>()).getOrDefault(x2, 0);
            int c3 = pointsCount.getOrDefault(y2, new HashMap<>()).getOrDefault(x1, 0);

            totalSquares += c1 * c2 * c3;
        }
    }

    return totalSquares;
}
}

```

---

13. [Stock Price Fluctuation](#)

```

Input
["StockPrice", "update", "update", "current", "maximum", "update", "maximum", "update", "minimum"]
[], [1, 10], [2, 5], [], [1, 3], [], [4, 2], []
Output
[null, null, null, 5, 10, null, 5, null, 2]

Explanation
StockPrice stockPrice = new StockPrice();
stockPrice.update(1, 10); // Timestamps are [1] with corresponding prices [10].
stockPrice.update(2, 5); // Timestamps are [1,2] with corresponding prices [10,5].
stockPrice.current(); // return 5, the latest timestamp is 2 with the price being 5.
stockPrice.maximum(); // return 10, the maximum price is 10 at timestamp 1.
stockPrice.update(1, 3); // The previous timestamp 1 had the wrong price, so it is updated to 3.
// Timestamps are [1,2] with corresponding prices [3,5].
stockPrice.maximum(); // return 5, the maximum price is 5 after the correction.
stockPrice.update(4, 2); // Timestamps are [1,2,4] with corresponding prices [3,5,2].
stockPrice.minimum(); // return 2, the minimum price is 2 at timestamp 4.

```

```

class StockPrice {

    private Map<Integer, Integer> timestampPriceMap;
    private TreeMap<Integer, Integer> priceFrequencyMap;
    private int latestTimestamp;

    public StockPrice() {
        timestampPriceMap = new HashMap<>();
        priceFrequencyMap = new TreeMap<>();
        latestTimestamp = 0;
    }

    public void update(int timestamp, int price) {
        if (timestampPriceMap.containsKey(timestamp)) {
            int oldPrice = timestampPriceMap.get(timestamp);
            int count = priceFrequencyMap.get(oldPrice);
            if (count == 1) {
                priceFrequencyMap.remove(oldPrice);
            } else {
                priceFrequencyMap.put(oldPrice, count - 1);
            }
        }
        timestampPriceMap.put(timestamp, price);
        priceFrequencyMap.put(price, priceFrequencyMap.getOrDefault(price, 0) +
    }
}

```

```

        latestTimestamp = Math.max(latestTimestamp, timestamp);
    }

    public int current() {
        return timestampPriceMap.get(latestTimestamp);
    }

    public int maximum() {
        return priceFrequencyMap.lastKey();
    }

    public int minimum() {
        return priceFrequencyMap.firstKey();
    }

}

/**
 * Your StockPrice object will be instantiated and called as such:
 * StockPrice obj = new StockPrice();
 * obj.update(timestamp,price);
 * int param_2 = obj.current();
 * int param_3 = obj.maximum();
 * int param_4 = obj.minimum();
 */

```

---

114. [Simple Bank System](#)

```

Input
["Bank", "withdraw", "transfer", "deposit", "transfer", "withdraw"]
[[[10, 100, 20, 50, 30]], [3, 10], [5, 1, 20], [5, 20], [3, 4, 15], [10, 50]]
Output
[null, true, true, true, false, false]

Explanation
Bank bank = new Bank([10, 100, 20, 50, 30]);
bank.withdraw(3, 10);      // return true, account 3 has a balance of $20, so it is valid to withdraw $10.
                           // Account 3 has $20 - $10 = $10.
bank.transfer(5, 1, 20); // return true, account 5 has a balance of $30, so it is valid to transfer $20.
                           // Account 5 has $30 - $20 = $10, and account 1 has $10 + $20 = $30.
bank.deposit(5, 20);     // return true, it is valid to deposit $20 to account 5.
                           // Account 5 has $10 + $20 = $30.
bank.transfer(3, 4, 15); // return false, the current balance of account 3 is $10,
                           // so it is invalid to transfer $15 from it.
bank.withdraw(10, 50);   // return false, it is invalid because account 10 does not exist.

```

```

public class Bank {
    public Bank(long[] balance) {
        this.balance = balance;
    }

    public boolean transfer(int account1, int account2, long money) {
        if (!isValid(account2))
            return false;
        return withdraw(account1, money) && deposit(account2, money);
    }

    public boolean deposit(int account, long money) {
        if (!isValid(account))
            return false;
        balance[account - 1] += money;
        return true;
    }

    public boolean withdraw(int account, long money) {
        if (!isValid(account))
            return false;
        if (balance[account - 1] < money)
            return false;
    }
}

```

```

        balance[account - 1] -= money;
        return true;
    }

    private long[] balance;

    private boolean isValid(int account) {
        return 1 <= account && account <= balance.length;
    }
}

/**
 * Your Bank object will be instantiated and called as such:
 * Bank obj = new Bank(balance);
 * boolean param_1 = obj.transfer(account1,account2,money);
 * boolean param_2 = obj.deposit(account,money);
 * boolean param_3 = obj.withdraw(account,money);
 */

```

## 15. Range Frequency Queries

```

Input
["RangeFreqQuery", "query", "query"]
[[[12, 33, 4, 56, 22, 2, 34, 33, 22, 12, 34, 56]], [1, 2, 4], [0, 11, 33]]
Output
[null, 1, 2]

Explanation
RangeFreqQuery rangeFreqQuery = new RangeFreqQuery([12, 33, 4, 56, 22, 2, 34, 33, 22, 12, 34, 56]);
rangeFreqQuery.query(1, 2, 4); // return 1. The value 4 occurs 1 time in the subarray [33, 4]
rangeFreqQuery.query(0, 11, 33); // return 2. The value 33 occurs 2 times in the whole array.

```

```

class RangeFreqQuery {
    // Map from value to list of indices where it appears in the array
    private Map<Integer, List<Integer>> positionsMap;

    public RangeFreqQuery(int[] arr) {

```

```

positionsMap = new HashMap<>();
for (int i = 0; i < arr.length; i++) {
    positionsMap.computeIfAbsent(arr[i], k → new ArrayList<>()).add(i);
}
}

public int query(int left, int right, int value) {
    List<Integer> indices = positionsMap.get(value);
    if (indices == null) return 0;

    int start = lowerBound(indices, left); // First index >= left
    int end = upperBound(indices, right); // First index > right

    return end - start;
}

// Binary search for first index >= target
private int lowerBound(List<Integer> list, int target) {
    int low = 0, high = list.size();
    while (low < high) {
        int mid = (low + high) / 2;
        if (list.get(mid) >= target) {
            high = mid;
        } else {
            low = mid + 1;
        }
    }
    return low;
}

// Binary search for first index > target
private int upperBound(List<Integer> list, int target) {
    int low = 0, high = list.size();
    while (low < high) {
        int mid = (low + high) / 2;
        if (list.get(mid) > target) {

```

```

        high = mid;
    } else {
        low = mid + 1;
    }
}
return low;
}

/**
 * Your RangeFreqQuery object will be instantiated and called as such:
 * RangeFreqQuery obj = new RangeFreqQuery(arr);
 * int param_1 = obj.query(left,right,value);
 */

```

## 16. Design Bitset

**Input**

```
["Bitset", "fix", "fix", "flip", "all", "unfix", "flip", "one", "unfix", "count", "toString"]
[[5], [3], [1], [], [], [0], [], [0], [0], [], []]
```

**Output**

```
[null, null, null, null, false, null, null, true, null, 2, "01010"]
```

**Explanation**

```
Bitset bs = new Bitset(5); // bitset = "00000".
bs.fix(3);      // the value at idx = 3 is updated to 1, so bitset = "00010".
bs.fix(1);      // the value at idx = 1 is updated to 1, so bitset = "01010".
bs.flip();       // the value of each bit is flipped, so bitset = "10101".
bs.all();        // return False, as not all values of the bitset are 1.
bs.unfix(0);    // the value at idx = 0 is updated to 0, so bitset = "00101".
bs.flip();       // the value of each bit is flipped, so bitset = "11010".
bs.one();        // return True, as there is at least 1 index with value 1.
bs.unfix(0);    // the value at idx = 0 is updated to 0, so bitset = "01010".
bs.count();      // return 2, as there are 2 bits with value 1.
bs.toString(); // return "01010", which is the composition of bitset.
```

```

class Bitset {
    private int size;
    private int countOnes;
}
```

```
private boolean flipped;
private int[] bits;

public Bitset(int size) {
    this.size = size;
    this.countOnes = 0;
    this.flipped = false;
    this.bits = new int[size];
}

public void fix(int idx) {
    if ((bits[idx] ^ (flipped ? 1 : 0)) == 0) {
        bits[idx] = flipped ? 0 : 1;
        countOnes++;
    }
}

public void unfix(int idx) {
    if ((bits[idx] ^ (flipped ? 1 : 0)) == 1) {
        bits[idx] = flipped ? 1 : 0;
        countOnes--;
    }
}

public void flip() {
    flipped = !flipped;
    countOnes = size - countOnes;
}

public boolean all() {
    return countOnes == size;
}

public boolean one() {
    return countOnes > 0;
}
```

```

public int count() {
    return countOnes;
}

public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int bit : bits) {
        sb.append((bit ^ (flipped ? 1 : 0)));
    }
    return sb.toString();
}
}

```

## 17. Encrypt and Decrypt Strings

**Input**

```
["Encrypter", "encrypt", "decrypt"]
[[["a", "b", "c", "d"], ["ei", "zf", "ei", "am"], ["abcd", "acbd", "adbc", "badc", "dacb", "cadb",
"cbda", "abad"]], ["abcd"], ["eizfeiam"]]
```

**Output**

```
[null, "eizfeiam", 2]
```

**Explanation**

```
Encrypter encrypter = new Encrypter([["a", "b", "c", "d"], ["ei", "zf", "ei", "am"], ["abcd", "acbd",
"adbc", "badc", "dacb", "cadb", "cbda", "abad"]]);
encrypter.encrypt("abcd"); // return "eizfeiam".
                                // 'a' maps to "ei", 'b' maps to "zf", 'c' maps to "ei", and 'd' maps to
                                // "am".
encrypter.decrypt("eizfeiam"); // return 2.
                                // "ei" can map to 'a' or 'c', "zf" maps to 'b', and "am" maps to 'd'.
                                // Thus, the possible strings after decryption are "abad", "cbad", "abcd",
                                // and "cbcd".
                                // 2 of those strings, "abad" and "abcd", appear in dictionary, so the
                                // answer is 2.
```

```

class Encrypter {

    private Map<Character, String> encMap;
    private Map<String, List<Character>> decMap;
    private Set<String> dict;
}

```

```

public Encrypter(char[] keys, String[] values, String[] dictionary) {
    encMap = new HashMap<>();
    decMap = new HashMap<>();
    for (int i = 0; i < keys.length; i++) {
        encMap.put(keys[i], values[i]);
        decMap.computeIfAbsent(values[i], k → new ArrayList<>()).add(keys[i]);
    }

    dict = new HashSet<>(Arrays.asList(dictionary));
}

public String encrypt(String word1) {
    StringBuilder sb = new StringBuilder();
    for (char c : word1.toCharArray()) {
        if (!encMap.containsKey(c)) return "";
        sb.append(encMap.get(c));
    }
    return sb.toString();
}

public int decrypt(String word2) {
    return dfs(word2, 0, new StringBuilder());
}

private int dfs(String word2, int idx, StringBuilder cur) {
    if (idx == word2.length()) {
        return dict.contains(cur.toString()) ? 1 : 0;
    }

    if (idx + 2 > word2.length()) return 0;

    String part = word2.substring(idx, idx + 2);
    if (!decMap.containsKey(part)) return 0;

    int count = 0;
    for (char c : decMap.get(part)) {

```

```

        cur.append(c);
        count += dfs(word2, idx + 2, cur);
        cur.deleteCharAt(cur.length() - 1);
    }
    return count;
}
}

```

---

## 18. Count Integers in Intervals

**Input**  
 ("CountIntervals", "add", "add", "count", "add", "count")  
 [[], [2, 3], [7, 10], [], [5, 8], []]  
**Output**  
 [null, null, null, 6, null, 8]

**Explanation**  
 CountIntervals countIntervals = new CountIntervals(); // initialize the object with an empty set of intervals.  
 countIntervals.add(2, 3); // add [2, 3] to the set of intervals.  
 countIntervals.add(7, 10); // add [7, 10] to the set of intervals.  
 countIntervals.count(); // return 6  
                           // the integers 2 and 3 are present in the interval [2, 3].  
                           // the integers 7, 8, 9, and 10 are present in the interval [7, 10].  
 countIntervals.add(5, 8); // add [5, 8] to the set of intervals.  
 countIntervals.count(); // return 8  
                           // the integers 2 and 3 are present in the interval [2, 3].  
                           // the integers 5 and 6 are present in the interval [5, 8].  
                           // the integers 7 and 8 are present in the intervals [5, 8] and [7, 10].  
                           // the integers 9 and 10 are present in the interval [7, 10].

```

class CountIntervals {

    private TreeMap<Integer, Integer> intervals;
    private int total;

    public CountIntervals() {
        intervals = new TreeMap<>();
        total = 0;
    }
}

```

```
public void add(int left, int right) {  
    int l = left, r = right;  
    Map.Entry<Integer, Integer> floor = intervals.floorEntry(right);  
  
    while (floor != null && floor.getValue() >= left) {  
        int start = floor.getKey();  
        int end = floor.getValue();  
        total -= end - start + 1;  
        intervals.remove(start);  
        l = Math.min(l, start);  
        r = Math.max(r, end);  
        floor = intervals.floorEntry(r);  
    }  
    intervals.put(l, r);  
    total += r - l + 1;  
}  
  
public int count() {  
    return total;  
}  
}
```

---

19. [Smallest Number in Infinite Set](#)

```

Input
["SmallestInfiniteSet", "addBack", "popSmallest", "popSmallest", "popSmallest", "addBack",
"popSmallest", "popSmallest", "popSmallest"]
[[], [2], [], [], [1], [], [], []]
Output
[null, null, 1, 2, 3, null, 1, 4, 5]

Explanation
SmallestInfiniteSet smallestInfiniteSet = new SmallestInfiniteSet();
smallestInfiniteSet.addBack(2); // 2 is already in the set, so no change is made.
smallestInfiniteSet.popSmallest(); // return 1, since 1 is the smallest number, and remove it from the
set.
smallestInfiniteSet.popSmallest(); // return 2, and remove it from the set.
smallestInfiniteSet.popSmallest(); // return 3, and remove it from the set.
smallestInfiniteSet.addBack(1); // 1 is added back to the set.
smallestInfiniteSet.popSmallest(); // return 1, since 1 was added back to the set and
// is the smallest number, and remove it from the set.
smallestInfiniteSet.popSmallest(); // return 4, and remove it from the set.
smallestInfiniteSet.popSmallest(); // return 5, and remove it from the set.

```

```

class SmallestInfiniteSet {

    private int current;
    private PriorityQueue<Integer> minHeap;
    private Set<Integer> addedBack;

    public SmallestInfiniteSet() {
        current = 1;
        minHeap = new PriorityQueue<>();
        addedBack = new HashSet<>();
    }

    public int popSmallest() {
        if (!minHeap.isEmpty()) {
            int smallest = minHeap.poll();
            addedBack.remove(smallest);
            return smallest;
        }
        return current++;
    }

    public void addBack(int num) {
        if (num < current && addedBack.add(num)) {

```

```

        minHeap.offer(num);
    }
}
}

```

## 20. Design a Number Container System

**Input**  
["NumberContainers", "find", "change", "change", "change", "change", "find", "change", "find"]  
[], [10], [2, 10], [1, 10], [3, 10], [5, 10], [10], [1, 20], [10]]  
**Output**  
[null, -1, null, null, null, null, 1, null, 2]

**Explanation**  
NumberContainers nc = new NumberContainers();  
nc.find(10); // There is no index that is filled with number 10. Therefore, we return -1.  
nc.change(2, 10); // Your container at index 2 will be filled with number 10.  
nc.change(1, 10); // Your container at index 1 will be filled with number 10.  
nc.change(3, 10); // Your container at index 3 will be filled with number 10.  
nc.change(5, 10); // Your container at index 5 will be filled with number 10.  
nc.find(10); // Number 10 is at the indices 1, 2, 3, and 5. Since the smallest index that is filled with 10 is 1, we return 1.  
nc.change(1, 20); // Your container at index 1 will be filled with number 20. Note that index 1 was filled with 10 and then replaced with 20.  
nc.find(10); // Number 10 is at the indices 2, 3, and 5. The smallest index that is filled with 10 is 2. Therefore, we return 2.

```

class NumberContainers {
    public void change(int index, int number) {
        if (indexToNumbers.containsKey(index)) {
            final int originalNumber = indexToNumbers.get(index);
            numberToIndices.get(originalNumber).remove(index);
            if (numberToIndices.get(originalNumber).isEmpty())
                numberToIndices.remove(originalNumber);
        }
        indexToNumbers.put(index, number);
        numberToIndices.putIfAbsent(number, new TreeSet<>());
        numberToIndices.get(number).add(index);
    }

    public int find(int number) {
        if (numberToIndices.containsKey(number))

```

```

        return numberToIndices.get(number).first();
    return -1;
}

private Map<Integer, TreeSet<Integer>> numberToIndices = new HashMap<>;
private Map<Integer, Integer> indexToNumbers = new HashMap<>();
}

```

## 21. Longest Uploaded Prefix

```

Input
["LUPrefix", "upload", "longest", "upload", "longest", "upload", "longest"]
[[4], [3], [], [1], [], [2], []]
Output
[null, null, 0, null, 1, null, 3]

Explanation
LUPrefix server = new LUPrefix(4);           // Initialize a stream of 4 videos.
server.upload(3);                          // Upload video 3.
server.longest();                         // Since video 1 has not been uploaded yet, there is no prefix.
                                         // So, we return 0.
server.upload(1);                          // Upload video 1.
server.longest();                         // The prefix [1] is the longest uploaded prefix, so we return 1.
server.upload(2);                          // Upload video 2.
server.longest();                         // The prefix [1,2,3] is the longest uploaded prefix, so we return
                                         3.

```

```

class LUPrefix {
    private boolean[] uploaded;
    private int longestPrefix;

    public LUPrefix(int n) {
        uploaded = new boolean[n + 2]; // 1-indexed, +1 for bounds safety
        longestPrefix = 0;
    }

    public void upload(int video) {
        uploaded[video] = true;
        // Move the prefix forward if the next in line is uploaded
    }
}

```

```

        while (uploaded[longestPrefix + 1]) {
            longestPrefix++;
        }
    }

    public int longest() {
        return longestPrefix;
    }
}

```

---

22. [Find Consecutive Integers from a Data Stream](#)

```

Input
["DataStream", "consec", "consec", "consec", "consec"]
[[4, 3], [4], [4], [4], [3]]
Output
[null, false, false, true, false]

Explanation
DataStream dataStream = new DataStream(4, 3); //value = 4, k = 3
dataStream.consec(4); // Only 1 integer is parsed, so returns False.
dataStream.consec(4); // Only 2 integers are parsed.
                     // Since 2 is less than k, returns False.
dataStream.consec(4); // The 3 integers parsed are all equal to value, so returns True.
dataStream.consec(3); // The last k integers parsed in the stream are [4,4,3].
                     // Since 3 is not equal to value, it returns False.

```

```

class DataStream {

    private final int value;
    private final int k;
    private int consecutiveCount;

    public DataStream(int value, int k) {
        this.value = value;
        this.k = k;
        this.consecutiveCount = 0;
    }
}

```

```

public boolean consec(int num) {
    if (num == value) {
        consecutiveCount++;
    } else {
        consecutiveCount = 0;
    }
    return consecutiveCount >= k;
}

```

### 23. [Design a Todo List](#)

#### Input

```

["TodoList", "addTask", "addTask", "getAllTasks", "getAllTasks",
 "addTask", "getTasksForTag", "completeTask", "completeTask",
 "getTasksForTag", "getAllTasks"]
[], [1, "Task1", 50, []], [1, "Task2", 100, ["P1"]],
[1], [5], [1, "Task3", 30, ["P1"]], [1, "P1"], [5, 1], [1, 2],
[1, "P1"], [1]]

```

#### Output

```

[null, 1, 2, ["Task1", "Task2"], [], 3, ["Task3", "Task2"],
 null, null, ["Task3"], ["Task3", "Task1"]]

```

#### Explanation

```

TodoList todoList = new TodoList();
todoList.addTask(1, "Task1", 50, []);
// return 1. This adds a new task for the user with id 1.
todoList.addTask(1, "Task2", 100, ["P1"]);
// return 2. This adds another task for the user with id 1.
todoList.getAllTasks(1);
// return ["Task1", "Task2"]. User 1 has two uncompleted tasks so far.
todoList.getAllTasks(5);
// return []. User 5 does not have any tasks so far.

```

```

todoList.addTask(1, "Task3", 30, ["P1"]);
// return 3. This adds another task for the user with id 1.
todoList.getTasksForTag(1, "P1");
// return ["Task3", "Task2"].
// This returns the uncompleted tasks that have the tag "P1"
// for the user with id 1.
todoList.completeTask(5, 1);
// This does nothing, since task 1 does not belong to user 5.
todoList.completeTask(1, 2);
// This marks task 2 as completed.
todoList.getTasksForTag(1, "P1");
// return ["Task3"]. This returns the uncompleted tasks
// that have the tag "P1" for the user with id 1.
// Notice that we did not include "Task2" because it is completed now.
todoList.getAllTasks(1);
// return ["Task3", "Task1"]. User 1 now has 2 uncompleted tasks.

```

```

class Task {
    int taskId;
    String taskName;
    int dueDate;
    Set<String> tags;
    boolean finish;

    public Task(int taskId, String taskName, int dueDate, Set<String> tags) {
        this.taskId = taskId;
        this.taskName = taskName;
        this.dueDate = dueDate;
        this.tags = tags;
    }
}

class TodoList {
    private int i = 1;
    private Map<Integer, TreeSet<Task>> tasks = new HashMap<>();

```

```

public TodoList() {

}

public int addTask(int userId, String taskDescription, int dueDate, List<String>
    Task task = new Task(i++, taskDescription, dueDate, new HashSet<>(tags))
    tasks.computeIfAbsent(userId, k → new TreeSet<>(Comparator.comparingLi
    return task.taskId;
}

public List<String> getAllTasks(int userId) {
    List<String> ans = new ArrayList<>();
    if (tasks.containsKey(userId)) {
        for (Task task : tasks.get(userId)) {
            if (!task.finish) {
                ans.add(task.taskName);
            }
        }
    }
    return ans;
}

public List<String> getTasksForTag(int userId, String tag) {
    List<String> ans = new ArrayList<>();
    if (tasks.containsKey(userId)) {
        for (Task task : tasks.get(userId)) {
            if (task.tags.contains(tag) && !task.finish) {
                ans.add(task.taskName);
            }
        }
    }
    return ans;
}

public void completeTask(int userId, int taskId) {
}

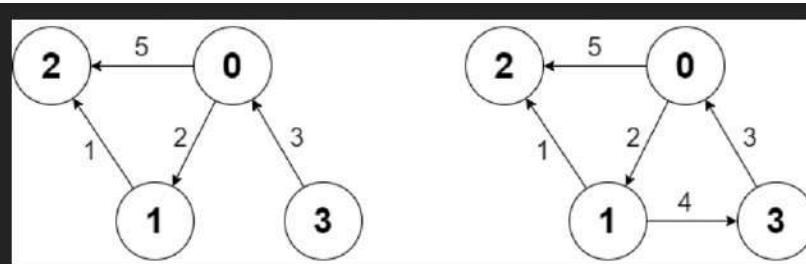
```

```

if (tasks.containsKey(userId)) {
    for (Task task : tasks.get(userId)) {
        if (task.taskId == taskId) {
            task.finish = true;
            break;
        }
    }
}
}

```

## 24. Design Graph With Shortest Path Calculator



**Input**  
 ["Graph", "shortestPath", "shortestPath", "addEdge", "shortestPath"]  
 [[4, [[0, 2, 5], [0, 1, 2], [1, 2, 1], [3, 0, 3]]], [3, 2], [0, 3], [[1, 3, 4]], [0, 3]]  
**Output**  
 [null, 6, -1, null, 6]

**Explanation**  
 Graph g = new Graph(4, [[0, 2, 5], [0, 1, 2], [1, 2, 1], [3, 0, 3]]);  
 g.shortestPath(3, 2); // return 6. The shortest path from 3 to 2 in the first diagram above is 3 → 0 → 1 → 2 with a total cost of 3 + 2 + 1 = 6.  
 g.shortestPath(0, 3); // return -1. There is no path from 0 to 3.  
 g.addEdge([1, 3, 4]); // We add an edge from node 1 to node 3, and we get the second diagram above.  
 g.shortestPath(0, 3); // return 6. The shortest path from 0 to 3 now is 0 → 1 → 3 with a total cost of 2 + 4 = 6.

```

class Graph {
    private final Map<Integer, List<int[]>> adjList;

    public Graph(int n, int[][] edges) {
        adjList = new HashMap<>();
        for (int i = 0; i < n; i++) {

```

```

        adjList.put(i, new ArrayList<>());
    }
    for (int[] edge : edges) {
        addEdge(edge);
    }
}

public void addEdge(int[] edge) {
    int from = edge[0], to = edge[1], weight = edge[2];
    adjList.get(from).add(new int[] { to, weight });
}

public int shortestPath(int node1, int node2) {
    PriorityQueue<int[]> minHeap = new PriorityQueue<>(Comparator.comparingInt((int[] e) -> e[1]));
    Map<Integer, Integer> dist = new HashMap<>();
    Set<Integer> visited = new HashSet<>();

    minHeap.offer(new int[] { node1, 0 });
    dist.put(node1, 0);

    while (!minHeap.isEmpty()) {
        int[] curr = minHeap.poll();
        int node = curr[0];
        int currentDist = curr[1];

        if (node == node2)
            return currentDist;
        if (visited.contains(node))
            continue;
        visited.add(node);

        for (int[] neighbor : adjList.getOrDefault(node, new ArrayList<>())) {
            int next = neighbor[0];
            int weight = neighbor[1];
            int newDist = currentDist + weight;

```

```

        if (!dist.containsKey(next) || newDist < dist.get(next)) {
            dist.put(next, newDist);
            minHeap.offer(new int[] { next, newDist });
        }
    }

    return -1; // unreachable
}

/**
 * Your Graph object will be instantiated and called as such:
 * Graph obj = new Graph(n, edges);
 * obj.addEdge(edge);
 * int param_2 = obj.shortestPath(node1,node2);
 */

```

---

25. Frequency Tracker

```

Input
["FrequencyTracker", "add", "add", "hasFrequency"]
[], [3], [3], [2]
Output
[null, null, null, true]

Explanation
FrequencyTracker frequencyTracker = new FrequencyTracker();
frequencyTracker.add(3); // The data structure now contains [3]
frequencyTracker.add(3); // The data structure now contains [3, 3]
frequencyTracker.hasFrequency(2); // Returns true, because 3 occurs twice

Example 2:

Input
["FrequencyTracker", "add", "deleteOne", "hasFrequency"]
[], [1], [1], [1]
Output
[null, null, null, false]

Explanation
FrequencyTracker frequencyTracker = new FrequencyTracker();
frequencyTracker.add(1); // The data structure now contains [1]
frequencyTracker.deleteOne(1); // The data structure becomes empty []
frequencyTracker.hasFrequency(1); // Returns false, because the data structure is empty

```

```

class FrequencyTracker {

    private Map<Integer, Integer> numFreqMap = new HashMap<>();
    private Map<Integer, Integer> freqCountMap = new HashMap<>();

    public FrequencyTracker() {}

    public void add(int number) {
        int freq = numFreqMap.getOrDefault(number, 0);
        numFreqMap.put(number, freq + 1);

        freqCountMap.put(freq, freqCountMap.getOrDefault(freq, 0) - 1);
        if (freqCountMap.get(freq) == 0) {
            freqCountMap.remove(freq);
        }
    }

    freqCountMap.put(freq + 1, freqCountMap.getOrDefault(freq + 1, 0) + 1);
}

```

```

}

public void deleteOne(int number) {
    if (!numFreqMap.containsKey(number)) return;

    int freq = numFreqMap.get(number);
    if (freq == 1) {
        numFreqMap.remove(number);
    } else {
        numFreqMap.put(number, freq - 1);
    }

    freqCountMap.put(freq, freqCountMap.getOrDefault(freq, 0) - 1);
    if (freqCountMap.get(freq) == 0) {
        freqCountMap.remove(freq);
    }

    if (freq > 1) {
        freqCountMap.put(freq - 1, freqCountMap.getOrDefault(freq - 1, 0) + 1);
    }
}

public boolean hasFrequency(int frequency) {
    return freqCountMap.getOrDefault(frequency, 0) > 0;
}

/** 
 * Your FrequencyTracker object will be instantiated and called as such:
 * FrequencyTracker obj = new FrequencyTracker();
 * obj.add(number);
 * obj.deleteOne(number);
 * boolean param_3 = obj.hasFrequency(frequency);
 */

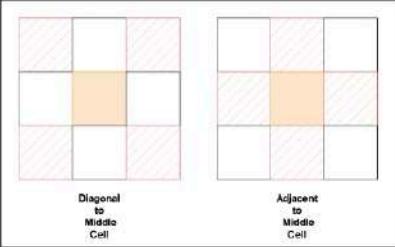
```

## 26. Design Neighbor Sum Service

You are given a  $n \times n$  2D array `grid` containing distinct elements in the range  $[0, n^2 - 1]$ .

Implement the `NeighborSum` class:

- `NeighborSum(int[][] grid)` initializes the object.
- `int adjacentSum(int value)` returns the sum of elements which are adjacent neighbors of `value`, that is either to the top, left, right, or bottom of `value` in `grid`.
- `int diagonalSum(int value)` returns the sum of elements which are diagonal neighbors of `value`, that is either to the top-left, top-right, bottom-left, or bottom-right of `value` in `grid`.

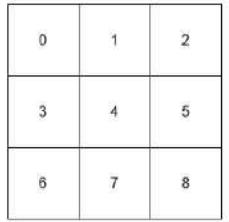


**Example 1:**

**Input:**  
["NeighborSum", "adjacentSum", "adjacentSum", "diagonalSum", "diagonalSum"]  
[[[[0, 1, 2], [3, 4, 5], [6, 7, 8]]], [1], [4], [4], [6]]

**Output:** [null, 6, 16, 16, 4]

**Explanation:**



- The adjacent neighbors of 1 are 0, 2, and 4.
- The adjacent neighbors of 4 are 1, 3, 5, and 7.
- The diagonal neighbors of 4 are 0, 2, 6, and 8.
- The diagonal neighbor of 8 is 4.

```
class NeighborSum {  
    private int[][] grid;  
    private int rows, cols;  
    private Map<Integer, List<int[]>> valueToCoordinates;  
  
    public NeighborSum(int[][] grid) {  
        this.grid = grid;  
        this.rows = grid.length;
```

```

this.cols = grid[0].length;
this.valueToCoordinates = new HashMap<>();

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        int val = grid[i][j];
        valueToCoordinates.putIfAbsent(val, new ArrayList<>());
        valueToCoordinates.get(val).add(new int[]{i, j});
    }
}

public int adjacentSum(int value) {
    int sum = 0;
    if (!valueToCoordinates.containsKey(value)) return 0;

    for (int[] pos : valueToCoordinates.get(value)) {
        int i = pos[0], j = pos[1];
        int[][] dirs = {{-1,0}, {1,0}, {0,-1}, {0,1}}; // up, down, left, right

        for (int[] d : dirs) {
            int ni = i + d[0], nj = j + d[1];
            if (ni >= 0 && ni < rows && nj >= 0 && nj < cols) {
                sum += grid[ni][nj];
            }
        }
    }
    return sum;
}

public int diagonalSum(int value) {
    int sum = 0;
    if (!valueToCoordinates.containsKey(value)) return 0;

    for (int[] pos : valueToCoordinates.get(value)) {
        int i = pos[0], j = pos[1];
    }
}

```

```

int[][] dirs = {{-1,-1}, {-1,1}, {1,-1}, {1,1}}; // diagonals

for (int[] d : dirs) {
    int ni = i + d[0], nj = j + d[1];
    if (ni >= 0 && ni < rows && nj >= 0 && nj < cols) {
        sum += grid[ni][nj];
    }
}
return sum;
}
}

```

127.

[Design an Array Statistics Tracker](#)

```

class MedianFinder {

    private final PriorityQueue<Integer> small = new PriorityQueue<>(Comparator.naturalOrder());
    private final PriorityQueue<Integer> large = new PriorityQueue<>();
    private final Map<Integer, Integer> delayed = new HashMap<>();
    private int smallSize;
    private int largeSize;

    public void addNum(int num) {
        if (small.isEmpty() || num <= small.peek()) {
            small.offer(num);
            ++smallSize;
        } else {
            large.offer(num);
            ++largeSize;
        }
    }
}

```

```

        rebalance();
    }

public Integer findMedian() {
    return smallSize == largeSize ? large.peek() : small.peek();
}

public void removeNum(int num) {
    delayed.merge(num, 1, Integer::sum);
    if (num <= small.peek()) {
        --smallSize;
        if (num == small.peek()) {
            prune(small);
        }
    } else {
        --largeSize;
        if (num == large.peek()) {
            prune(large);
        }
    }
    rebalance();
}

private void prune(PriorityQueue<Integer> pq) {
    while (!pq.isEmpty() && delayed.containsKey(pq.peek())) {
        if (delayed.merge(pq.peek(), -1, Integer::sum) == 0) {
            delayed.remove(pq.peek());
        }
        pq.poll();
    }
}

private void rebalance() {
    if (smallSize > largeSize + 1) {
        large.offer(small.poll());
        --smallSize;
    }
}

```

```

        ++largeSize;
        prune(small);
    } else if (smallSize < largeSize) {
        small.offer(large.poll());
        --largeSize;
        ++smallSize;
        prune(large);
    }
}
}

class StatisticsTracker {
    private final Deque<Integer> q = new ArrayDeque<>();
    private long s;
    private final Map<Integer, Integer> cnt = new HashMap<>();
    private final MedianFinder medianFinder = new MedianFinder();
    private final TreeSet<int[]> ts
        = new TreeSet<>((a, b) → a[1] == b[1] ? a[0] - b[0] : b[1] - a[1]);

    public StatisticsTracker() {
    }

    public void addNumber(int number) {
        q.offerLast(number);
        s += number;
        ts.remove(new int[] {number, cnt.getOrDefault(number, 0)});
        cnt.merge(number, 1, Integer::sum);
        medianFinder.addNum(number);
        ts.add(new int[] {number, cnt.get(number)});
    }

    public void removeFirstAddedNumber() {
        int number = q.pollFirst();
        s -= number;
        ts.remove(new int[] {number, cnt.get(number)});
        cnt.merge(number, -1, Integer::sum);
    }
}

```

```
        medianFinder.removeNum(number);
        ts.add(new int[] {number, cnt.get(number)}));
    }

    public int getMean() {
        return (int) (s / q.size());
    }

    public int getMedian() {
        return medianFinder.findMedian();
    }

    public int getMode() {
        return ts.first()[0];
    }
}
```

---

28. [Design a 3D Binary Matrix with Efficient Layer Tracking](#)

**Input:**

```
["Matrix3D", "setCell", "largestMatrix", "setCell", "largestMatrix", "setCell", "largestMatrix"]
[[3], [0, 0, 0], [], [1, 1, 2], [], [0, 0, 1], []]
```

**Output:**

```
[null, null, 0, null, 1, null, 0]
```

**Explanation**

```
Matrix3D matrix3D = new Matrix3D(3); // Initializes a 3 x 3 x 3 3D array matrix ,  
filled with all 0's.  
matrix3D.setCell(0, 0, 0); // Sets matrix[0][0][0] to 1.  
matrix3D.largestMatrix(); // Returns 0. matrix[0] has the most number of 1's.  
matrix3D.setCell(1, 1, 2); // Sets matrix[1][1][2] to 1.  
matrix3D.largestMatrix(); // Returns 1. matrix[0] and matrix[1] tie with the most  
number of 1's, but index 1 is bigger.  
matrix3D.setCell(0, 0, 1); // Sets matrix[0][0][1] to 1.  
matrix3D.largestMatrix(); // Returns 0. matrix[0] has the most number of 1's.
```

```
class Matrix3D {  
    private final int[][][] grid; // 3D matrix to track cell states (0 or 1)  
    private final int[] layerActiveCount; // Number of active cells in each x-layer  
  
    // TreeSet to maintain [activeCellCount, layerIndex] pairs sorted by:  
    // 1. descending activeCellCount  
    // 2. descending layerIndex (if counts are equal)  
    private final TreeSet<int[]> sortedLayers = new TreeSet<>(  
        (a, b) → a[0] == b[0] ? b[1] - a[1] : b[0] - a[0]  
    );  
  
    public Matrix3D(int n) {  
        grid = new int[n][n][n];  
        layerActiveCount = new int[n];  
    }  
  
    public void setCell(int x, int y, int z) {
```

```

if (grid[x][y][z] == 1) return;

grid[x][y][z] = 1;
sortedLayers.remove(new int[]{layerActiveCount[x], x});
layerActiveCount[x]++;
sortedLayers.add(new int[]{layerActiveCount[x], x});
}

public void unsetCell(int x, int y, int z) {
    if (grid[x][y][z] == 0) return;

    grid[x][y][z] = 0;
    sortedLayers.remove(new int[]{layerActiveCount[x], x});
    layerActiveCount[x]--;
    if (layerActiveCount[x] > 0) {
        sortedLayers.add(new int[]{layerActiveCount[x], x});
    }
}

public int largestMatrix() {
    return sortedLayers.isEmpty() ? grid.length - 1 : sortedLayers.first()[1];
}
}

```

29. [Booking Concert Tickets in Groups](#)

**Example 1:**

```
Input
["BookMyShow", "gather", "gather", "scatter", "scatter"]
[[2, 5], [4, 0], [2, 0], [5, 1], [5, 1]]
Output
[null, [0, 0], [], true, false]

Explanation
BookMyShow bms = new BookMyShow(2, 5); // There are 2 rows with 5 seats each
bms.gather(4, 0); // return [0, 0]
                    // The group books seats [0, 3] of row 0.
bms.gather(2, 0); // return []
                    // There is only 1 seat left in row 0,
                    // so it is not possible to book 2 consecutive seats.
bms.scatter(5, 1); // return True
                    // The group books seat 4 of row 0 and seats [0, 3] of row 1.
bms.scatter(5, 1); // return False
                    // There is only one seat left in the hall.
```

```
class Node {
    int l, r;
    long mx, s;
}

class SegmentTree {
    private Node[] tr;
    private int m;

    public SegmentTree(int n, int m) {
        this.m = m;
        tr = new Node[n << 2];
        for (int i = 0; i < tr.length; ++i) {
            tr[i] = new Node();
        }
        build(1, 1, n);
    }

    private void build(int u, int l, int r) {
        tr[u].l = l;
        tr[u].r = r;
        if (l == r) {
            tr[u].s = m;
```

```

        tr[u].mx = m;
        return;
    }
    int mid = (l + r) >> 1;
    build(u << 1, l, mid);
    build(u << 1 | 1, mid + 1, r);
    pushup(u);
}

public void modify(int u, int x, long v) {
    if (tr[u].l == x && tr[u].r == x) {
        tr[u].s = v;
        tr[u].mx = v;
        return;
    }
    int mid = (tr[u].l + tr[u].r) >> 1;
    if (x <= mid) {
        modify(u << 1, x, v);
    } else {
        modify(u << 1 | 1, x, v);
    }
    pushup(u);
}

public long querySum(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) {
        return tr[u].s;
    }
    int mid = (tr[u].l + tr[u].r) >> 1;
    long v = 0;
    if (l <= mid) {
        v += querySum(u << 1, l, r);
    }
    if (r > mid) {
        v += querySum(u << 1 | 1, l, r);
    }
}

```

```

        return v;
    }

public int queryIdx(int u, int l, int r, int k) {
    if (tr[u].mx < k) {
        return 0;
    }
    if (tr[u].l == tr[u].r) {
        return tr[u].l;
    }
    int mid = (tr[u].l + tr[u].r) >> 1;
    if (tr[u << 1].mx >= k) {
        return queryIdx(u << 1, l, r, k);
    }
    if (r > mid) {
        return queryIdx(u << 1 | 1, l, r, k);
    }
    return 0;
}

private void pushup(int u) {
    tr[u].s = tr[u << 1].s + tr[u << 1 | 1].s;
    tr[u].mx = Math.max(tr[u << 1].mx, tr[u << 1 | 1].mx);
}
}

class BookMyShow {
    private int n;
    private int m;
    private SegmentTree tree;

    public BookMyShow(int n, int m) {
        this.n = n;
        this.m = m;
        tree = new SegmentTree(n, m);
    }
}

```

```

public int[] gather(int k, int maxRow) {
    ++maxRow;
    int i = tree.queryIdx(1, 1, maxRow, k);
    if (i == 0) {
        return new int[] {};
    }
    long s = tree.querySum(1, i, i);
    tree.modify(1, i, s - k);
    return new int[] {i - 1, (int) (m - s)};
}

public boolean scatter(int k, int maxRow) {
    ++maxRow;
    if (tree.querySum(1, 1, maxRow) < k) {
        return false;
    }
    int i = tree.queryIdx(1, 1, maxRow, 1);
    for (int j = i; j <= n; ++j) {
        long s = tree.querySum(1, j, j);
        if (s >= k) {
            tree.modify(1, j, s - k);
            return true;
        }
        k -= s;
        tree.modify(1, j, 0);
    }
    return true;
}
}

/**
 * Your BookMyShow object will be instantiated and called as such:
 * BookMyShow obj = new BookMyShow(n, m);
 * int[] param_1 = obj.gather(k,maxRow);

```

```
* boolean param_2 = obj.scatter(k,maxRow);
*/
```

### 30. Sequentially Ordinal Rank Tracker

**Example 1:**

```
Input
["SORTracker", "add", "add", "get", "add", "get", "add", "get", "add", "get", "add", "get", "add", "get"]
[], ["bradford", 2], ["branford", 3], [], ["alps", 2], [], ["orland", 2], [], ["orlando", 3], [], ["alpine", 2], [], []
Output
[null, null, null, "branford", null, "alps", null, "bradford", null, "bradford", null, "bradford", "orland"]

Explanation
SORTracker tracker = new SORTracker(); // Initialize the tracker system.
tracker.add("bradford", 2); // Add location with name="bradford" and score=2 to the system.
tracker.add("branford", 3); // Add location with name="branford" and score=3 to the system.
tracker.get(); // The sorted locations, from best to worst, are: branford, bradford.
// Note that branford precedes bradford due to its higher score (3 > 2).
tracker.add("alps", 2); // This is the 1st time get() is called, so return the best location: "branford".
// Add location with name="alps" and score=2 to the system.
tracker.get(); // Sorted locations: branford, alps, bradford.
// Note that alps precedes bradford even though they have the same score (2).
// This is because "alps" is lexicographically smaller than "bradford".
// Return the 2nd best location "alps", as it is the 2nd time get() is called.
tracker.add("orland", 2); // Add location with name="orland" and score=2 to the system.
tracker.get(); // Sorted locations: branford, alps, bradford, orland.
// Return "bradford".
tracker.add("orlando", 3); // Add location with name="orlando" and score=3 to the system.
tracker.get(); // Sorted locations: branford, orlando, alps, bradford, orland.
// Return "bradford".
tracker.add("alpine", 2); // Add location with name="alpine" and score=2 to the system.
tracker.get(); // Sorted locations: branford, orlando, alpine, alps, bradford, orland.
// Return "bradford".
tracker.get(); // Sorted locations: branford, orlando, alpine, alps, bradford, orland.
// Return "orland".
```

```
class SORTracker {
    private PriorityQueue<Map.Entry<Integer, String>> good = new PriorityQueue<
        (a, b) → a.getKey().equals(b.getKey()) ? b.getValue().compareTo(a.getValue())
        : a.getKey() - b.getKey());
    private PriorityQueue<Map.Entry<Integer, String>> bad = new PriorityQueue<
        (a, b) → a.getKey().equals(b.getKey()) ? a.getValue().compareTo(b.getValue())
        : b.getKey() - a.getKey());

    public SORTracker() {
    }

    public void add(String name, int score) {
        good.offer(Map.entry(score, name));
        bad.offer(good.poll());
```

```
}

public String get() {
    good.offer(bad.poll());
    return good.peek().getValue();
}

}

/***
 * Your SORTracker object will be instantiated and called as such:
 * SORTracker obj = new SORTracker();
 * obj.add(name,score);
 * String param_2 = obj.get();
 */
```

---

31. [Walking Robot Simulation II](#)

**Example 1:**

**Input:** commands = [4,-1,3], obstacles = []

**Output:** 25

**Explanation:**

The robot starts at (0, 0):

1. Move north 4 units to (0, 4).
2. Turn right.
3. Move east 3 units to (3, 4).

The furthest point the robot ever gets from the origin is (3, 4), which squared is  $3^2 + 4^2 = 25$  units away.

**Example 2:**

**Input:** commands = [4,-1,4,-2,4], obstacles = [[2,4]]

**Output:** 65

**Explanation:**

The robot starts at (0, 0):

1. Move north 4 units to (0, 4).
2. Turn right.
3. Move east 1 unit and get blocked by the obstacle at (2, 4), robot is at (1, 4).
4. Turn left.
5. Move north 4 units to (1, 8).

The furthest point the robot ever gets from the origin is (1, 8), which squared is  $1^2 + 8^2 = 65$  units away.

```
public class Solution {  
    public int robotSim(int[] commands, int[][] obstacles) {  
        // Direction vectors for North, East, South, West  
        int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};  
        int x = 0, y = 0, direction = 0;  
  
        // Set of obstacles for quick lookup  
        Set<String> obstacleSet = new HashSet<>();  
        for (int[] obstacle : obstacles) {  
            obstacleSet.add(obstacle[0] + "," + obstacle[1]);  
        }  
  
        int maxDistSquared = 0;
```

```

for (int command : commands) {
    if (command == -2) { // Turn left (90 degrees counterclockwise)
        direction = (direction + 3) % 4;
    } else if (command == -1) { // Turn right (90 degrees clockwise)
        direction = (direction + 1) % 4;
    } else { // Move forward by `command` steps
        for (int i = 0; i < command; i++) {
            int newX = x + directions[direction][0];
            int newY = y + directions[direction][1];

            // Check if the new position is not blocked by an obstacle
            if (!obstacleSet.contains(newX + "," + newY)) {
                x = newX;
                y = newY;
            }
        }
    }

    // Calculate the maximum distance squared from origin to avoid floating point errors
    maxDistSquared = Math.max(maxDistSquared, x * x + y * y);
}

return maxDistSquared;
}
}

```

32. [Fancy Sequence](#)

Write an API that generates fancy sequences using the `append`, `addAll`, and `multAll` operations.

Implement the `Fancy` class:

- `Fancy()` Initializes the object with an empty sequence.
- `void append(val)` Appends an integer `val` to the end of the sequence.
- `void addAll(inc)` Increments all existing values in the sequence by an integer `inc`.
- `void multAll(m)` Multiplies all existing values in the sequence by an integer `m`.
- `int getIndex(idx)` Gets the current value at index `idx` (0-indexed) of the sequence modulo  $10^9 + 7$ . If the index is greater or equal than the length of the sequence, return -1.

**Example 1:**

**Input**  
["Fancy", "append", "addAll", "append", "multAll", "getIndex", "addAll", "append", "multAll", "getIndex", "getIndex", "getIndex"]  
[], [2], [3], [7], [2], [0], [3], [10], [2], [0], [1], [2]]  
**Output**  
[null, null, null, null, null, 10, null, null, 26, 34, 20]

**Explanation**  
Fancy fancy = new Fancy();  
fancy.append(2); // fancy sequence: [2]  
fancy.addAll(3); // fancy sequence: [2+3] -> [5]  
fancy.append(7); // fancy sequence: [5, 7]  
fancy.multAll(2); // fancy sequence: [5\*2, 7\*2] -> [10, 14]  
fancy.getIndex(0); // return 10  
fancy.addAll(3); // fancy sequence: [10+3, 14+3] -> [13, 17]  
fancy.append(10); // fancy sequence: [13, 17, 10]  
fancy.multAll(2); // fancy sequence: [13\*2, 17\*2, 10\*2] -> [26, 34, 20]  
fancy.getIndex(0); // return 26  
fancy.getIndex(1); // return 34  
fancy.getIndex(2); // return 20

```
class Fancy {  
    private static final int MOD = 1_000_000_007; // Modulo for all calculations to handle large numbers  
  
    private List<Long> vals = new ArrayList<>(); // Stores the sequence of values  
  
    private long a = 1; // Scaling factor for the sequence (default is 1). This controls the range of values.  
  
    private long b = 0; // Additive offset for the sequence (default is 0). This controls the starting point of the sequence.  
  
    // Method to calculate modular exponentiation: x^n % MOD  
    private int modPow(long x, long n) {  
        if (n == 0)  
            return 1;  
        if (n % 2 == 1)  
            return (int) (x * modPow(x % MOD, n - 1) % MOD);  
        return modPow(x * x % MOD, n / 2) % MOD;  
    }  
}
```

```

// Append a value to the sequence, adjusting it using the current multiplier `a` and offset `b`.
public void append(int val) {
    final long adjustedVal = (val - b + MOD) % MOD; // Adjust value by subtracting offset
    vals.add(adjustedVal * modPow(a, MOD - 2) % MOD); // Apply modular inverse of a^2
}

// Add a constant `inc` to all elements in the sequence by adjusting the offset `b`.
public void addAll(int inc) {
    b = (b + inc) % MOD; // Adjust the offset `b` by adding `inc`, ensuring positive
}

// Multiply all elements in the sequence by a constant `m`, adjusting the multiplier `a`.
public void multAll(int m) {
    a = (a * m) % MOD; // Adjust the multiplier `a` by multiplying with `m`.
    b = (b * m) % MOD; // Adjust the offset `b` by multiplying with `m`.
}

// Retrieve the value at index `idx`, applying the transformation a * val + b to it.
public int getIndex(int idx) {
    return idx >= vals.size() ? -1 : (int) ((a * vals.get(idx)) + b) % MOD; // Return transformed value
}

```

---

### 33. [Design TreeMap](#)

Designing a custom [TreeMap](#) involves understanding the functionality of a map that maintains its entries in a sorted order. A [TreeMap](#) in Java is typically backed by a self-balancing binary search tree (like a Red-Black Tree), but for educational purposes, we can implement one using a simpler structure.

#### Core Design:

## 1. Key-Value Pairs:

- A `TreeMap` stores key-value pairs.
- The keys are sorted based on their natural order or by a comparator (if provided).

## 2. Tree Structure:

- The underlying data structure can be a **Binary Search Tree (BST)**, specifically a **Red-Black Tree** for balancing.

## 3. Operations:

- **Insertion:** Add key-value pairs while maintaining the order of keys.
- **Deletion:** Remove a key-value pair.
- **Search:** Retrieve the value associated with a key.
- **Traversal:** Iterate over the keys and values in sorted order.

For simplicity, let's design a basic version of a `TreeMap` backed by a **Binary Search Tree (BST)**.

## Simplified TreeMap Design using a Binary Search Tree (BST):

### 1. Node Structure:

- Each node contains:
  - A **key**.
  - A **value** associated with the key.
  - Left and Right children for the BST properties.

### 2. Operations:

- **put(key, value):** Insert a key-value pair.
- **get(key):** Retrieve the value for the given key.
- **remove(key):** Remove the key-value pair from the tree.
- **containsKey(key):** Check if the key exists in the map.
- **size():** Get the size of the map (number of key-value pairs).

- **inOrderTraversal()**: Traverse the tree in sorted order.

```

class TreeMap {

    // Define the Node structure
    class Node {
        int key;
        int value;
        Node left, right;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
            left = right = null;
        }
    }

    private Node root;
    private int size;

    public TreeMap() {
        root = null;
        size = 0;
    }

    // Insertion of key-value pair
    public void put(int key, int value) {
        root = put(root, key, value);
    }

    private Node put(Node node, int key, int value) {
        if (node == null) {
            size++;
            return new Node(key, value);
        }
    }
}

```

```

        if (key < node.key) {
            node.left = put(node.left, key, value);
        } else if (key > node.key) {
            node.right = put(node.right, key, value);
        } else {
            node.value = value; // Update the value if the key exists
        }

        return node;
    }

    // Search for value by key
    public int get(int key) {
        Node node = get(root, key);
        if (node == null) {
            throw new IllegalArgumentException("Key not found");
        }
        return node.value;
    }

    private Node get(Node node, int key) {
        if (node == null) {
            return null;
        }

        if (key < node.key) {
            return get(node.left, key);
        } else if (key > node.key) {
            return get(node.right, key);
        } else {
            return node;
        }
    }

    // Remove a key-value pair

```

```

public void remove(int key) {
    root = remove(root, key);
}

private Node remove(Node node, int key) {
    if (node == null) {
        return null;
    }

    if (key < node.key) {
        node.left = remove(node.left, key);
    } else if (key > node.key) {
        node.right = remove(node.right, key);
    } else {
        // Key found, now delete it
        if (node.left == null) {
            return node.right;
        } else if (node.right == null) {
            return node.left;
        }
    }

    // Node with two children: Get the inorder successor (smallest in the right
    node.value = minValue(node.right);
    node.key = minKey(node.right);
    node.right = remove(node.right, node.key);
}

    return node;
}

// Get minimum value in a tree
private int minValue(Node node) {
    int minValue = node.value;
    while (node.left != null) {
        node = node.left;
        minValue = node.value;
    }
}

```

```

    }
    return minValue;
}

// Get minimum key in a tree
private int minKey(Node node) {
    int minKey = node.key;
    while (node.left != null) {
        node = node.left;
        minKey = node.key;
    }
    return minKey;
}

// Check if a key exists
public boolean containsKey(int key) {
    return get(root, key) != null;
}

// Get the size of the TreeMap
public int size() {
    return size;
}

// In-order traversal (sorted order)
public void inOrderTraversal() {
    inOrderTraversal(root);
}

private void inOrderTraversal(Node node) {
    if (node != null) {
        inOrderTraversal(node.left);
        System.out.println("Key: " + node.key + ", Value: " + node.value);
        inOrderTraversal(node.right);
    }
}

```

```
// Main method for testing
public static void main(String[] args) {
    TreeMap treeMap = new TreeMap();
    treeMap.put(10, 100);
    treeMap.put(20, 200);
    treeMap.put(5, 50);
    treeMap.put(15, 150);

    // In-order traversal (sorted by keys)
    treeMap.inOrderTraversal();

    System.out.println("Value for key 10: " + treeMap.get(10));
    treeMap.remove(10);
    treeMap.inOrderTraversal();
}
}
```

---

**Questions:**

LRU Cache

MRU Cache

LFU Cache

Min Stack

Max Stack

Maximum Frequency Stack

Implement Stack using Queues

Implement Queue using Stacks

## Dinner Plate Stacks

How would you design a class that behaves similar to a Java Map <Date, Object> except that get() returns the mapped value for not only an exact match but also for the latest Date in the Map

Design a Stack With Increment Operation

Design HashSet

Design HashMap

Design Linked List

Two Sum III - Data structure design

Design In-Memory File System

Design Video Sharing Platform

Shuffle an Array

Snake And Ladders

Design Snake Game

Design Tic-Tac-Toe

Binary Search Tree Iterator

Implement Trie (Prefix Tree)

Design Add and Search Words Data Structure

Design Twitter

Logger Rate Limiter

Design Hit Counter

Design Browser History

Design A Leaderboard

Design File System

Design a File Sharing System

Time Based Key-Value Store

Kth Largest Element in a Stream

Find Median from Data Stream  
Product of the Last K Numbers  
Kth Ancestor of a Tree Node  
Flatten 2D Vector  
Encode and Decode Strings  
Peeking Iterator  
Design Circular Queue  
Design Circular Deque  
Design Excel Sum Formula  
Design Log Storage System  
Design Search Autocomplete System  
Implement Magic Dictionary  
Map Sum Pairs  
Design Most Recently Used Queue  
Tweet Counts Per Frequency  
Design an ATM Machine  
Encode and Decode TinyURL  
Design a Food Rating System  
Design Task Manager  
Design Spreadsheet  
Implement Router  
Flatten Nested List Iterator  
Design Compressed String Iterator  
Zigzag Iterator  
RLE Iterator  
Serialize and Deserialize Binary Tree  
Range Sum Query - Immutable

Range Sum Query 2D - Immutable

Range Sum Query - Mutable

Range Sum Query 2D - Mutable

Insert Delete GetRandom O(1)

Insert Delete GetRandom O(1) - Duplicates allowed

Online Stock Span

All O`one Data Structure

Encode N-ary Tree to Binary Tree

Unique Word Abbreviation

Moving Average from Data Stream

Data Stream as Disjoint Intervals

Design Phone Directory

Serialize and Deserialize BST

Shortest Word Distance II

Range Module

My Calendar I

My Calendar II

My Calendar III

Prefix and Suffix Search

Exam Room

Online Election

Complete Binary Tree Inserter

Number of Recent Calls

Stream of Characters

Snapshot Array

Online Majority Element In Subarray

Design Skiplist

Find Elements in a Contaminated Binary Tree

Iterator for Combination

Apply Discount Every n Orders

Design Underground System

First Unique Number

Subrectangle Queries

Dot Product of Two Sparse Vectors

Binary Search Tree Iterator II

Throne Inheritance

Design Parking System

Design an Expression Tree With Evaluate Function

Design an Ordered Stream

Design Front Middle Back Queue

Design Authentication Manager

Implement Trie II (Prefix Tree)

Finding MK Average

Seat Reservation Manager

Finding Pairs With a Certain Sum

Design Movie Rental System

Operations on Tree

Detect Squares

Stock Price Fluctuation

Simple Bank System

Range Frequency Queries

Design Bitset

Encrypt and Decrypt Strings

Count Integers in Intervals

Design a Text Editor  
Smallest Number in Infinite Set  
Design a Number Container System  
Design SQL  
Longest Uploaded Prefix  
Design Memory Allocator  
Find Consecutive Integers from a Data Stream  
Design a Todo List  
Design Graph With Shortest Path Calculator  
Frequency Tracker  
Design Neighbor Sum Service  
Design an Array Statistics Tracker  
Design a 3D Binary Matrix with Efficient Layer Tracking  
Booking Concert Tickets in Groups  
Sequentially Ordinal Rank Tracker  
Walking Robot Simulation II  
Fancy Sequence