Medium Array Design Queue Iterator **Leetcode Link**

Problem Description

281. Zigzag Iterator

Given two input vectors v1 and v2, the task is to create an iterator that returns their elements in a zigzag or alternating fashion. In other words, the iterator should return the first element of v1, then the first element of v2, followed by the second element of v1, and

2. The next index to be accessed in each vector.

To achieve this, the iterator should keep track of two pieces of information:

- The iterator should also be able to tell whether there are more elements to return, which involves checking both vectors for remaining elements.
- Intuition

1. Which vector's turn it is to provide an element.

so on, alternating between the two vectors.

The solution involves maintaining state for the iterator that includes which vector to access next and the current index within each

- vector. The core idea is to attempt to retrieve an element from one vector and then switch to the other, continuing this process alternately until all elements from both vectors are exhausted.
- In detail, the solution requires: 1. An initialization step that sets up the structure to hold the vectors and their corresponding indices.
- vector has an element that has not been returned yet.
- element and updating the stored state. 3. A method to check if there are any elements left (hasNext() function) by iterating over the vectors and ensuring that at least one

2. A way to move to the next element (next() function) that adheres to the zigzag order. This includes returning the current

The next() method retrieves the current element from the active vector based on the current index, then increments the index for that vector and updates which vector is active for the next call, ensuring the alternating pattern.

current vector is exhausted, it moves to the next vector and checks for the same. This process wraps around in a cycle until it

there are no more elements to return. Otherwise, it returns true, indicating that there are further elements to process.

• __init__: In this constructor method, we initialize critical pieces of information that the iterator needs:

Increment the index for the active vector, moving the iterator forward within that vector.

• hasNext: This method is responsible for determining if the iterator has additional elements to process:

element, we break the loop and return True, confirming that the iterator has more elements.

returns to the starting vector. If the starting vector is reached and it is also exhausted, the method returns false, indicating that

Solution Approach

The hasNext() method checks if the current vector is exhausted by comparing the current index with the length of the vector. If the

To implement the ZigzagIterator, we utilize a few concepts central to the iterator pattern and alternating patterns between two data structures.

be the length of an array of vectors.

each vector.

The data structures used include:

next: When this method is called, we:

by alternating between the vectors.

An array for self.vectors to hold the input vectors.

Create an instance of the ZigzagIterator, initializing the following:

o self.vectors = [[1, 3], [2, 4, 5, 6]]

4. Call next() again. This time, self.cur is 1, so we use v2.

 self.cur: A variable storing which vector is currently active (starting with the first vector, 0). o self.size: In this particular case, we have two vectors, so the size is static and set to 2. For a more generic solution, it could

o self.indexes: An array to track the current index within each vector, initialized to zeros since we start at the beginning of

Update self.cur to point to the next vector, wrapping around using the modulo operator %. This ensures the zigzag pattern

• We enter a loop where we check if the current vector (designated by self.cur) has been completely iterated over by

• If the current vector is exhausted, we increment self.cur, using % to wrap around, effectively moving to the next vector.

 Access the current vector using self.cur to determine whether v1 or v2 should be accessed. Retrieve the element at the current index of the active vector.

self.vectors: An array that stores the input vectors v1 and v2.

Algorithmically, the solution consists of steps performed in the following methods:

• We keep rotating through the vectors until we return to the starting vector (start), checking if any elements remain in any of them. • If we return to the starting vector and no elements are left, we return False. Otherwise, at first detection of an available

An array (or list in Python) for self.indexes to track the current index within each vector.

comparing the index in self.indexes for the current vector with its length.

• We start by setting a variable start to self.cur to remember the starting point.

A defining pattern of this implementation is the alternating access pattern to vectors, achieved by cycling through using % on self.cur, and a while loop in hasNext to traverse the vectors in round-robin fashion.

The overall time complexity for next is O(1), since it involves only simple operations without any loops. For hasNext, the time

complexity in the worst case can be O(m + n), where m and n are the lengths of v1 and v2, respectively, but this only happens when

we exhaust all the vectors (when the iterator ends). On average, assuming that the vectors are reasonably balanced in size, the

complexity for hasNext is amortized to O(1). **Example Walkthrough**

Let's illustrate the solution approach using two vectors, v1 and v2, where v1 = [1, 3] and v2 = [2, 4, 5, 6].

• self.indexes = [0, 0] (for tracking the current index of each vector) o self.cur = 0 (starting with the first vector) self.size = 2 (we are working with two vectors) Using next()

return False.

Python Solution

toggling self.cur between 1 and 0.

We return True for hasNext().

self.size = 2

def next(self) -> int:

return result

return True

40 # i, v = ZigzagIterator(v1, v2), []

current = 0;

public int next() {

return result;

return true;

* while (iterator.hasNext()) {

public boolean hasNext() {

indices.add(0);

vectors.add(v2);

41 # while i.hasNext(): v.append(i.next())

def hasNext(self) -> bool:

start = self.current

self.indices = [0] * self.size

Identify the current vector and its index

Increment the index for the current vector

self.current = (self.current + 1) % self.size

Move to the next vector for the following call

Store the starting point to avoid infinite loops

public ZigzagIterator(List<Integer> v1, List<Integer> v2) {

// Returns the next element in the zigzag iteration.

List<Integer> vector = vectors.get(current);

int index = indices.get(current);

if (start == current) {

return false;

* List<Integer> result = new ArrayList<>();

result.add(iterator.next());

indices.add(0); // Adding initial indices for both vectors.

vectors.add(v1); // Adding the provided lists to the vectors list.

// Get the current vector and the index for the element to return.

current = (current + 1) % listCount; // Move to the next vector.

while (indices.get(current) == vectors.get(current).size()) {

// Returns true if there is a next element in the iteration, false otherwise.

current = (current + 1) % listCount; // Move to next vector.

// then there are no elements left, return false.

// If we have found a vector that still has elements, return true.

// Current position to determine from which vector to take the next element

// Vector to keep track of the indexes of current read position for each vector

// Constructor that initializes ZigzagIterator with two vectors (v1 and v2)

listCount = 2; // Since we always have two vectors for this iterator

vectors.push_back(v1); // Adding the provided vectors to the vectors list

indices[current] = index + 1; // Update the index for the current vector

// Returns true if there is a next element in the iteration, false otherwise

current = (current + 1) % listCount; // Move to next vector

// If we have cycled through all vectors without finding an element,

2 let current: number = 0; // Holds the position to determine from which array to pick the next element.

const index = indices[current]; // Get the index for the element to be returned from the current array.

let listCount: number = 2; // In this setup, we always have two arrays defined.

let vectors: number[][] = []; // Stores the provided arrays for iteration.

let indices: number[] = [0, 0]; // Tracks the current read position for each array.

indices[current] = index + 1; // Update the read position for the current array.

current = (current + 1) % listCount; // Move to the next array for the following call.

// If we've checked all arrays and found no remaining elements, return false.

index, and then adjusting the current pointer cur. All these operations take constant time.

vectors = [v1, v2]; // Stores the two provided arrays (v1 and v2) for zigzag iteration.

indices.push_back(0); // Adding initial indices for both vectors

// Get the current vector and index for the element to return

current = (current + 1) % listCount; // Move to the next vector

ZigzagIterator(std::vector<int>& v1, std::vector<int>& v2) {

int result = vector.get(index); // Get the next element.

int start = current; // Remember the starting point.

listCount = 2; // Since we always have two lists (v1 and v2) for this iterator.

indices.set(current, index + 1); // Update the index for the current vector.

// Cycle through the vectors to find if any vector still has elements left.

// If we have cycled through all vectors without finding an element,

self.indices[self.current] = index + 1

Retrieve the next element from the current vector

vector = self.vectors[self.current]

index = self.indices[self.current]

Return the retrieved element

return False

Store the two input vectors

self.vectors = [v1, v2]

result = vector[index]

Initialization

6. On the next next() call, return v1[1] since self.cur is 0 and self.indexes[0] is 1. The value is 3. Increment self.indexes[0] to

5. We return v2[0], which is 2, and then self.indexes[1] becomes 1. We update self.cur to 0 (modulo 2).

1. Call next(), which checks self.cur to determine the current vector. Since self.cur is 0, we look at v1.

2. The current index for v1 is self.indexes[0], which is 0; hence we return v1[0], which is 1.

3. Increment self.indexes[0] to 1 and update self.cur to 1 (for the next vector, v2).

2 and update self.cur to 1.

10

11

12

13

14

15

16

17

19

20

22

23

24

25

26

27

28

29

35

36

37

38

42

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

59

61

62

63

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

33

34

35

36

37

38

39

40

41

47

48

49

50

51

52

53

55

63

64

66

67

11 }

12

15

16

17

19

20

28

29

31

32

33

34

35

37

44

36 }

38 // Usage

21 }

* }

*/

54 };

56 /**

Using hasNext()

The size variable indicates the number of vectors (always 2 in this case)

Check if the current vector is exhausted and move to the next one if necessary

while self.indices[self.current] == len(self.vectors[self.current]):

If we haven't returned yet, there's still at least one element left

Create a list to keep track of the current index in each vector

• Initially, self.cur is 0, and self.indexes[0] is 2, which is equal to the length of v1. So we move to v2. • self.indexes[1] is less than the length of v2, meaning there are elements left in v2.

7. Continue calling next() to return the next elements from v2: 4, 5, and 6, incrementing self.indexes[1] after each call and

class ZigzagIterator: def __init__(self, v1: List[int], v2: List[int]): # Initialize the current pointer to 0 to start with the first vector self.current = 0

Once all elements have been processed and self.indexes for both vectors are equal to their respective lengths, hasNext() will

30 # Move to the next vector self.current = (self.current + 1) % self.size 32 # If we loop back to the start, that means all vectors are exhausted 33 if self.current == start: 34

39 # Usage example:

```
Java Solution
   import java.util.ArrayList;
    import java.util.List;
     public class ZigzagIterator {
         // Current position to determine from which vector to take the next element.
         private int current;
  8
         // Total number of vectors
         private int listCount;
  9
 10
 11
         // List to keep track of the indexes of current read position for each vector.
 12
         private List<Integer> indices = new ArrayList<>();
 13
 14
         // List holding the two input vectors.
 15
         private List<List<Integer>> vectors = new ArrayList<>();
 16
 17
         // Constructor which initializes the ZigzagIterator with two lists (v1 and v2).
```

53 54 55 /** * Example usage: * ZigzagIterator iterator = new ZigzagIterator(v1, v2);

C++ Solution

private:

public:

1 #include <vector>

class ZigzagIterator {

int current;

int listCount;

// Total number of vectors

std::vector<int> indices;

current = 0;

return result;

bool hasNext() {

return true;

* std::vector<int> v1 = {1,2};

* while (iterator.hasNext()) {

* std::vector<int> result;

* std::vector<int> v2 = {3,4,5,6};

* Example usage:

Typescript Solution

int next() {

indices.push_back(0);

vectors.push_back(v2);

int index = indices[current];

if (start == current) {

* ZigzagIterator iterator = ZigzagIterator(v1, v2);

1 // Initialize variables to hold state for the iterator

// Initializes the iterator with two arrays (v1 and v2).

const vector = vectors[current]; // Get the current array.

current = (current + 1) % listCount; // Move to the next array.

const result = vector[index]; // Store the next element.

// Found an array that still has elements, return true.

function initialize(v1: number[], v2: number[]): void {

13 // Returns the next element in the zigzag iteration.

* // Now 'result' contains the zigzag iteration of v1 and v2

result.push_back(iterator.next());

return false;

// Vector holding the two input vectors

std::vector<std::vector<int>> vectors;

// Returns the next element in zigzag iteration

std::vector<int>& vector = vectors[current];

int result = vector[index]; // Get the next element

int start = current; // Remember the starting point

// then there are no elements left, return false

// If we found a vector that still has elements, return true

// Cycle through the vectors to find if any vector still has elements left 42 43 while (indices[current] == vectors[current].size()) { 44 45 46

22 23 // Checks if there are any more elements in any array. 24 function hasNext(): boolean { let start = current; // Save the starting position // Loop through the arrays to determine if any of them still have elements. 26 27 while (indices[current] === vectors[current].length) {

return true;

39 // initialize(v1, v2);

Time Complexity

// while (hasNext()) {

if (start === current) {

return false;

40 // let result: number[] = [];

result.push(next());

Time and Space Complexity

return result;

current = 0;

14 function next(): number {

```
hasNext() Method: The time complexity for hasNext() is 0(k) in the worst case, where k is the number of vectors (2 in this case).
This is because hasNext() might potentially go through both vectors to find out if there's a next element. In the current
```

averages out to 0(1) per call to next().

implementation, k is 2, so we can argue that the method runs in 0(1) as well, but if the iterator is extended to support more than two input vectors, the time complexity will depend on the number of vectors, hence O(k). Overall time complexity per element across the entire iteration process is still 0(1). Even though hasNext() may take 0(k) time in the

worst case, across n elements, it would only be checking each vector once per element, resulting in O(n) for n calls to next(), which

next() Method: The time complexity for next() is 0(1) because we are simply accessing an element from an array, updating an

Space Complexity The space complexity is 0(n1 + n2), where n1 and n2 are the lengths of v1 and v2, respectively. This is due to the storage requirement of the vectors list which contains references to the two input vectors. The additional space used by the iterator object

current context.

itself (for storing cur and indexes) is O(k) where k is the size of the indexes array or the number of input vectors, which is 2 in the Since we provided the vectors themselves as input to the constructor, we generally consider them part of the input size rather than attributing them to the auxiliary space complexity of the algorithm. Therefore, if we exclude the space taken by the input vectors, the auxiliary space complexity of the Zigzaglterator implementation is 0(1) as we only use a fixed number of additional variables that does not depend on the input size.