1146. Snapshot Array Binary Search Hash Table Medium Design Leetcode Link Array

Problem Description The task is to create a data structure called SnapshotArray that represents an array-like entity where snapshots of the array's state

can be taken at different times. Initially, the array is of a given length and all elements are set to 0. The SnapshotArray should support the following operations: 1. Initialization: When an instance of SnapshotArray is created, it should be initialized with a certain length. All elements should be

- set to 0. 2. Set: There should be a function that updates the value of an element at a particular index. This means changing the element's
- value to a new one. 3. Snap: This function takes a snapshot of the current state of the array and returns an identifier for this snapshot, known as snap_id. The snap_id starts from 0 and increments by 1 each time a new snapshot is taken.
- 4. Get: Given an index and a snap_id, this function should return the value of the element at the given index as it was during the snapshot identified by snap_id.
- Intuition To solve this problem, we need a way to efficiently record the changes to each element of the array over time—this is done so that

Here's the intuition explained step by step:

2. Set: When setting a value at an index, we append a tuple to the list at that index containing the current snap_id and the new

index. Each entry in the list will be a tuple consisting of a snap_id and a value, representing the value of the array at that index

1. Initialization: We use a defaultdict of lists to represent the SnapshotArray. This allows us to keep a history of changes for each

value. This way, we are effectively recording a timeline of values for each index. 3. Snap: This operation increments the snap_id used to track the snapshots and returns the previous snap_id.

after the snapshot was taken.

we can later query the value of an element as it was at any snapshot point.

- 4. Get: To retrieve a value from a past snapshot, we use binary search (bisect_right) to find the first tuple where the snap_id is greater than the requested snap_id. We then take the previous tuple as it will have the value of the array at the time of the snapshot. If there is no such tuple (i.e., if the index was never set before the snap_id), we return 0.
- The solution uses binary search for efficient retrieval of the historical value and defaultdict to avoid having to initialize every index which makes it space-efficient if many indices remain at the initial value 0.
- Solution Approach

The implementation of the SnapshotArray uses a combination of data structures and algorithms to provide an efficient way to

Here's how each part of the solution approach works: A defaultdict is used to initialize the arr attribute. The defaultdict from Python's collections library creates a new list for each new key by default, which is ideal for our case where a key represents an index in the SnapshotArray, and the value is a list

• The set method appends a tuple (```self.idx````, val) to the list at the specified index. If the index is being set for the first

• The snap method increments self.idx, the internal snapshot index, and returns the previous snapshot index by doing self.idx

that stores the history of values set at that index along with their corresponding snap_id.

and space since unchanged elements are assumed to keep their previous values.

lists to maintain a complete history, and utilizing binary search for quick retrieval.

1. Initialization Create a SnapshotArray instance with a length of 3.

Let's go through a small example to illustrate the solution approach for implementing SnapshotArray.

time, the defaultdict will create an empty list and then append to it. It records the current snapshot index self.idx alongside the value val.

- 1. This operation assigns a new snapshot identifier and ensures that subsequent set operations will be considered part of a new snapshot.

• The get method retrieves the value for a given index at a specific snapshot. It uses binary search to find the correct value.

bisect_right(vals, (```snap_id````, inf)) finds the insertion point in vals, which is the list of all values set for index, to

snap_id, we take one step back with - 1. If the resultant index i is negative, it means no set operations were done before

• Lazy Update: Instead of updating all values at each snapshot, we only record changes. This saves on unnecessary operations

History of Operations: Keeping a list of all operations (snap_id, value) allows us to reconstruct the state at any snapshot time.

• Binary Search: To efficiently find the required state from the historical operations, we use binary search, which gives us a O(log

snap_id, so it returns 0. Otherwise, it returns the value, which is the second element of the tuple vals[i][1].

maintain the sorted order if snap_id were to be added to vals. Since we want the most recent set operation before or equal to

The essential patterns used in this implementation are:

execute the required operations.

- n) time complexity, where n is the number of operations recorded for an index. In conclusion, the SnapshotArray is efficiently implemented using defaultdict for lazy update initialization, appending operations to
- 1 snapArray = SnapshotArray(3) 2 // The internal structure now is: {0: [(0, 0)], 1: [(0, 0)], 2: [(0, 0)]},
- 1 snapArray.set(1, 5) 2 // The internal structure is now: {0: [(0, 0)], 1: [(0, 5)], 2: [(0, 0)]}. 3 // As the current snap_id is 0, we record that index 1 has value 5 after snapshot 0.

1 snapArray.set(1, 6) 2 // The internal structure is now: {0: [(0, 0)], 1: [(0, 5), (1, 6)], 2: [(0, 0)]}.

1 value = snapArray.get(1, 0)

1 snap_id_1 = snapArray.snap()

1 value = snapArray.get(1, 1)

Python Solution

from math import inf

class SnapshotArray:

10

11

12

13

20

21

22

23

24

25

26

27

28

29

30

31

32

38

58

C++ Solution

1 #include <vector>

2 using std::vector;

using std::pair;

6 public:

9

10

11

12

14

15

16

17

18

20

21

22

23

24

25

26

27

28

29

30

31

33

34

35

36

37

39

40

41

42

44

54

55

9 }

10

class SnapshotArray {

int snap() {

return idx++;

void set(int index, int val) {

int get(int index, int snap_id) {

while (left < right) {</pre>

} else {

* int snapIndex = obj->snap();

Typescript Solution

let snapshotIndex: number = 0;

snapshotIndex = 0;

* int val = obj->get(index, snap_id);

right = mid;

left = mid + 1;

int idx; // The current snapshot index.

// Initialize the SnapshotArray with a specific length.

// Sets the element at the given index to the given value.

// Increase and return the current snapshot index.

if (changes[mid].first > snap_id) {

return left == 0 ? 0 : changes[left - 1].second;

// Define the snapshot index and the array to hold snapshot data.

// Function to initialize the SnapshotArray with a specific length.

snapshotArray = new Array(length).fill(null).map(() => []);

11 // Function to set the element at the given index to the given value.

let snapshotArray: Array<Array<[number, number]>> = [];

function initializeSnapshotArray(length: number): void {

// Takes a snapshot of the array and returns the index of the snapshot.

int left = 0, right = changes.size(); // Binary search bounds.

// Retrieves the value at the given index according to the specified snapshot id.

SnapshotArray(int length) : idx(0), snaps(length) {

snaps[index].push_back({idx, val});

1 from collections import defaultdict

def __init__(self, length: int):

self.snapshot_index = 0

self.snapshot_index += 1

return self.snapshot_index - 1

def get(self, index: int, snap_id: int) -> int:

self.array_data = defaultdict(list)

def set(self, index: int, val: int) -> None:

from bisect import bisect_right

def snap(self) -> int:

Example Walkthrough

3 // with snap_id initialized at 0.

3. Snap Operation Take a snapshot.

1 snap_id_0 = snapArray.snap()

2. **Set Operation** Set the value at index 1 to 5.

4. Set Operation Set the value at index 1 to 6.

6. Snap Operation Take another snapshot.

3 // The internal structure is unchanged.

2 // This increments the internal snap_id to 1 and returns 0.

5. Get Operation Get the value at index 1 with snap id 0.

2 // This increments the internal snap_id to 2 and returns 1.

7. **Get Operation** Get the value at index 1 with snap_id 1.

8. Get Operation Get the value at index 2 with snap_id1.

during the get operation to efficiently retrieve the correct historical value.

3 // The internal structure is still: {0: [(0, 0)], 1: [(0, 5)], 2: [(0, 0)]}.

3 // Since the current snap_id is 1, we record that index 1 has value 6 after snapshot 1.

3 // that would come after (0, inf) if it were inserted, which is the tuple (1, 6). 4 // We take the previous tuple (0, 5) and return the value 5. 5 // The method would return 5, the value at index 1 during snapshot 0.

2 // Here we perform a binary search to find the first entry in the list at index 1

1 value = snapArray.get(2, 1) 2 // Since index 2 has not been set after snapshot 0, binary search finds 3 // the first tuple (0, 0) and returns 0 since there are no changes.

Initialize an index counter for snapshots and a default dictionary

to store values with their corresponding snapshot indices.

Take a snapshot of the array, incrementing the snapshot index.

Increment the snapshot index and return the ID of the snapshot taken.

Get the value at the specified index at the time of the given snapshot.

Retrieve the list of value-snapshot index pairs for the given index.

2 // The method looks for the entry after (1, inf) and finds nothing, so it returns the last value before,

3 // which is from the tuple (1, 6). Therefore, it returns 6, the value at index 1 during snapshot 1.

Set the value at a particular index to the specified value 14 within the latest snapshot. 16 17 # Append the current snapshot index and val to the list at the given index. self.array_data[index].append((self.snapshot_index, val)) 19

private final List<int[]>[] snapshotArray; // An array of lists to hold the elements at different snapshots

Through this example, we have shown how set, snap, and get operations interact with the SnapshotArray's underlying data structure.

We used defaultdict to initialize the snapshot array lazily, appended changes at each set operation, and applied binary search

```
33
           value_snapshots = self.array_data[index]
           # Find the rightmost value such that it's snapshot index is less than or equal to snap_id.
34
35
            i = bisect_right(value_snapshots, (snap_id, inf)) - 1
36
           # Return 0 if no such index is found, otherwise return the value at that index.
37
            return 0 if i < 0 else value_snapshots[i][1]
```

Java Solution

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

class SnapshotArray {

```
private int currentSnapId; // To keep track of the current snapshot id
 8
       // Constructor
 9
       public SnapshotArray(int length) {
10
            // Initialize the array of lists with the given length
11
12
            snapshotArray = new List[length];
           // Fill each list within the array
13
14
           Arrays.setAll(snapshotArray, k -> new ArrayList<>());
15
16
17
       // Set a value at a particular index
18
        public void set(int index, int val) {
19
            /* Add a pair (snapshot id, value) to the corresponding index.
20
             * If a snapshot is taken after this, it will have the new value. */
21
            snapshotArray[index].add(new int[] {currentSnapId, val});
22
23
24
       // Take a snapshot of the array and return the current snapshot id
25
        public int snap() {
26
           // Increase the snapshot id and return the id of the snapshot taken
27
            return currentSnapId++;
28
29
30
       // Get the value at a specific index for a given snapshot
31
       public int get(int index, int snapId) {
32
           // List for values at the index
33
            List<int[]> values = snapshotArray[index];
34
            // Binary search to find the floor entry of the provided snapId
35
            int left = 0, right = values.size();
            while (left < right) {
36
37
                int mid = (left + right) / 2; // mid-point for binary search
               // If the mid snapshot id is greater than snapId, search on the left
38
39
                if (values.get(mid)[0] > snapId) {
40
                    right = mid;
41
               } else {
42
                    // If not, search on the right
43
                    left = mid + 1;
44
45
46
           // If there's no element, return 0. Otherwise, return the value found at the snapId
47
            return left == 0 ? 0 : values.get(left - 1)[1];
48
49
50
51 /**
52
    * Usage:
    * SnapshotArray obj = new SnapshotArray(length);
    * obj.set(index, val);
54
    * int snapId = obj.snap();
   * int value = obj.get(index, snapId);
57
```

// Append a new pair of the current "snapshot index" and value to the changes of the specified index.

auto& changes = snaps[index]; // A reference to the list of value changes for the specified index.

// Perform a binary search to find the first change with snapshot index greater than snap_id.

// If left is 0, it means no changes were recorded for the index until snap_id, so return 0.

vector<vector<pair<int, int>>> snaps; // A 2D vector to store the value changes for each index.

// Otherwise, return the value before the first change that has snapshot index greater than snap_id.

int mid = (left + right) >> 1; // Calculate the middle point using bitwise shift.

46 }; 47 48 * Your SnapshotArray object will be instantiated and called as such: * SnapshotArray* obj = new SnapshotArray(length); * obj->set(index, val);

*/

private:

```
function setSnapshotArrayValue(index: number, value: number): void {
       const changes = snapshotArray[index];
13
       changes.push([snapshotIndex, value]);
14
15 }
16
  // Function to take a snapshot of the array and return the index of the snapshot.
   function snapSnapshotArray(): number {
       return snapshotIndex++;
19
20 }
21
   // Function to retrieve the value at the given index according to the specified snapshot ID.
   function getSnapshotArrayValue(index: number, snapshotId: number): number {
       const changes = snapshotArray[index];
24
25
       let left = 0;
26
       let right = changes.length;
27
28
       // Perform a binary search to find the element with snapshotId less than or equal to the requested snapshotId.
29
       while (left < right) {</pre>
30
           const mid: number = (left + right) >> 1;
31
           if (changes[mid][0] > snapshotId) {
32
               right = mid;
           } else {
33
34
               left = mid + 1;
35
36
37
38
       // If left is 0, it means no changes were recorded for the index until snapshotId, return 0.
       // Otherwise, return the last value before the snapshotId increased.
39
       return left === 0 ? 0 : changes[left - 1][1];
40
41 }
42
   // Example usage:
   // initializeSnapshotArray(3); // Initialize it with a length of 3.
45 // setSnapshotArrayValue(0, 5); // Set the first element to 5.
46 // const snapId = snapSnapshotArray(); // Take a snapshot and get the snapshot ID.
47 // const value = getSnapshotArrayValue(0, snapId); // Get the value at index 0 for this snapshot ID.
Time and Space Complexity
Time Complexity
```

__init__: O(1) - Initializing the array and idx takes constant time. • set: O(1) - Appending a value to the array list at a given index is an amortized constant time operation.

binary search has logarithmic time complexity relative to the number of elements it is searching through. Space Complexity

snap: O(1) - Incrementing the snapshot index is a constant time operation.

- Overall: O(N * S) Every set operation could be in a new index and could create up to S snapshots for every index in the worst case, where N is the number of indices in the SnapshotArray and S is the number of snapshots taken per index.
- Each entry in arr is a tuple storing two integers (snap_id, value), so every set operation adds another tuple to the space used by the
- data structure.

• get: O(log S) - Using binary search (bisect_right) where S is the number of snapshots at a particular index. This is because