Solution: Snapshot Array

Let's solve the Snapshot Array problem using the Custom Data Structures pattern.

We'll cover the following

Statement

In this challenge, you have to implement a **Snapshot Array** with the following properties:

- **Constructor (length)**: This is the constructor and it initializes the data structure to hold the specified number of indexes.
- **Set Value (idx, val)**: This property sets the value at a given index **idx** to value **val**.
- **Snapshot()**: This method takes no parameters and returns the **Snap ID**. **Snap ID** is the number of times that the snapshot function was called, less 1, as we start the count at 0. The first time this function is called, it saves a snapshot and returns 0. The n^{th} time it is called, after saving the snapshot, it returns $n - 1$.
- **Get Value (idx, Snap ID)** method returns the value at the index in the snapshot with the given **Snap ID**.

Suppose that we have three nodes whose values we wish to track in the snapshot array. Initially, the value of all the nodes will be 0. After calling the **Set Value (1, 4)** function, the value of node 1 will change to 4. If we take a snapshot at this point, the current values of all the nodes will be saved with **Snap ID** 0. Now, if we call **Set Value (1, 7)**, the current value for node 1 will change to 7. Now, if we call the **Get Value (1, 0)** function, we will get the value of node 1 from snapshot 0, that is, 4.

Constraints:

- $1 \leq$ `length` $\leq 5 \times 10^3$
- $0 \leq$ `idx` $<$ `length`
- $0 \leq$ `val` $\leq 10^9$
- $0 \leq$ `snapid` $<$ (the total number of times we call **Snapshot**)
- At most 5×10^3 calls will be made to **Set Value**, **Snapshot**, and **Get Value**.

Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

A naive approach to create an array-like data structure that supports snapping its values at different times would be to create a list to store the array values in each snapshot. We will increment the counter to keep track of the current snapshot ID. To take a snapshot, we will copy the current list and add it to the list of snapshots. To get the value on a given index and the snapshot ID, we access the given snapshot ID and the corresponding element in the snapshot list for the index.

For example, we want to create an array of length 5 and take three snapshots of its values at different times. We will create a list to store the values of the array at each snapshot, as well as a counter to keep track of the current snapshot ID:

```
arr = [0, 0, 0, 0, 0]
```

To set a value at a given index, we can update the corresponding element in the list:

```
arr[2] = 4
```

To take a snapshot, we will create a copy of the current list and add it to the list of snapshots. After taking three snapshots, the snapshots list will contain the following:

```
snapshots = [[0, 0, 0, 0, 0], [0, 0, 4, 0, 0], [0, 0, 4, 0, 0], [0, 0, 4, 0, 0]]
```

This naive approach will work for small arrays and a small number of snapshots. When the size of the array and the number of snapshots increases, this approach can become inefficient, since it requires copying the entire list for each snapshot. Let’s see if we can use the Custom Data Structures pattern to reduce the complexity of our solution.

Optimized approach using nested dictionaries

To solve this problem, we will start by creating a dictionary named **Node Value**. The node value will hold all the values, along with the nodes, at different times in further subdictionaries. The keys to the **Node Value** dictionary are the Snapshot IDs. For a given key, the values are also dictionaries. These inner dictionaries have node IDs as keys and the node’s value as values.

The **Node Value** dictionary only stores the values that have changed since the last snapshot, instead of storing the entire array for each snapshot. This allows for more efficient memory usage and faster performance, especially for large arrays and a large number of snapshots.

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

As an optimization, we will not initialize all the nodes. Instead, we will initialize a node when we set its value for the first time. We will use the **Set Value (idx, val)** function to set the node at the specified **idx** to **val**.

Java

```java
        List<List<List<Integer>>> nodesValue = Arrays.asList(
                Arrays.asList(Arrays.asList(0, 5), Arrays.asList(0, 1), Arrays.asList(2, 3), Arrays.asLis
                Arrays.asList(Arrays.asList(4, 1), Arrays.asList(2, 21)),
                Arrays.asList(Arrays.asList(4, 12), Arrays.asList(2, 61)),
                Arrays.asList(Arrays.asList(0, 15), Arrays.asList(0, 5), Arrays.asList(2, 13), Arrays.asL
                Arrays.asList(Arrays.asList(0, 32), Arrays.asList(0, 6), Arrays.asList(1, 2))
        );

        for (int i = 0; i < numNodes.size(); i++) {
            System.out.println(i + 1 + ".\tInitializing a data structure with " + numNodes.get(i) + " nod
            SnapshotArray snapshotArr = new SnapshotArray(numNodes.get(i));

            for (int j = 0; j < nodesValue.get(i).size(); j++) {
                for (int k = 0; k < nodesValue.get(i).get(j).size(); k++) {
                    System.out.println("\t\tSetting the value of node " + nodesValue.get(i).get(j).get(0)
                    snapshotArr.setValue(nodesValue.get(i).get(j).get(0), nodesValue.get(i).get(j).get(1)
                    System.out.println("\t\tContents of snapshot array:" + snapshotArr.nodeValue);
                    break;
                }
            }

            System.out.println(new String(new char[100]).replace('\0', '-'));
        }
    }
}


```

Snapshot Array

The dictionary will contain a copy of the current state of the nodes. All the nodes with their values will be added to this dictionary against the same **Snap ID**. We can easily verify that there is no error in the copy by printing and comparing it with the contents of the dictionary.

Java

```java
                Arrays.asList(Arrays.asList(4, 1), Arrays.asList(2, 21)),
                Arrays.asList(Arrays.asList(4, 12), Arrays.asList(2, 61)),
                Arrays.asList(Arrays.asList(0, 15), Arrays.asList(0, 5), Arrays.asList(2, 13), Arrays.asL
                Arrays.asList(Arrays.asList(0, 32), Arrays.asList(0, 6), Arrays.asList(1, 2))
        );

        for (int i = 0; i < numNodes.size(); i++) {
            System.out.println(i + 1 + ".\tInitializing a data structure with " + numNodes.get(i) + " nod
            SnapshotArray snapshotArr = new SnapshotArray(numNodes.get(i));

            for (int j = 0; j < nodesValue.get(i).size(); j++) {
                for (int k = 0; k < nodesValue.get(i).get(j).size(); k++) {
                    System.out.println("\t\tSnapshot before setting value: " + snapshotArr.nodeValue);
                    snapshotArr.snapshot();
                    System.out.println("\t\tSetting the value of node " + nodesValue.get(i).get(j).get(0)
                            " to " + nodesValue.get(i).get(j).get(1));
                    snapshotArr.setValue(nodesValue.get(i).get(j).get(0), nodesValue.get(i).get(j).get(1)
                    System.out.println("\t\tSnapshot after setting value: " + snapshotArr.nodeValue);
                    System.out.println();
                    break;
                }
            }

            System.out.println(new String(new char[100]).replace('\0', '-'));
    }
  }
}
```

Snapshot Array

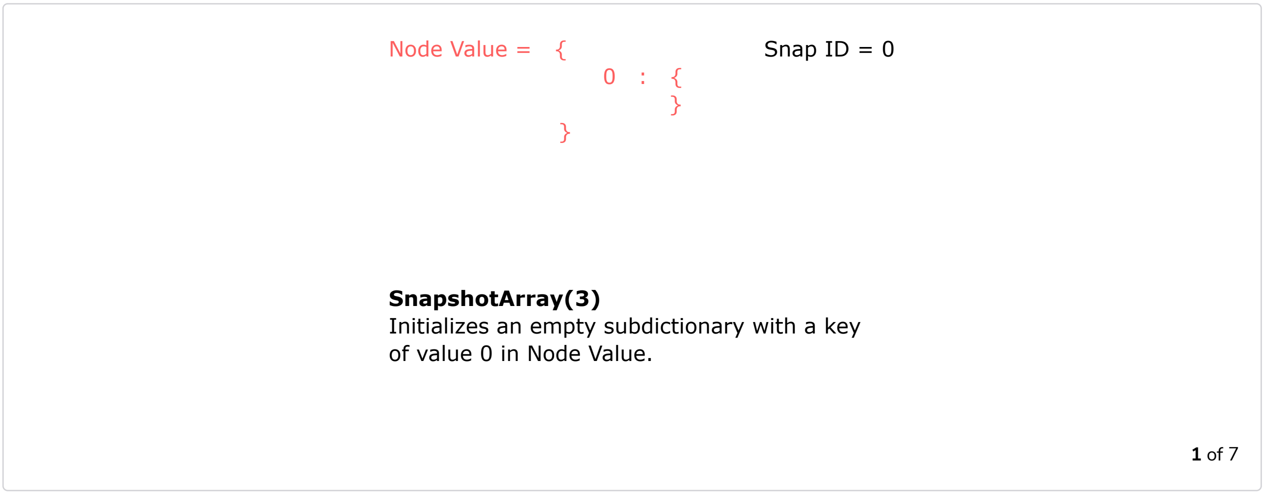
Now, even though we've made a copy of the current state of the dictionary, we won't save it anywhere. As a result, every time we set a value of a node, the previous state of the nodes is lost.

We can understand the limited functionality of the code above with the help of an example. Suppose we initialize a data structure with a length of 3. We first set the value of node 0 to 1 and then to 5. Even though we had made a copy of node 0 before setting it to 5, we lost that state of the node because we did not save the state anywhere.

So, where should we save the copy of the current state of the nodes?

Since we need to save the current state of the nodes at every snapshot, we will maintain a **Snap ID** to identify each snapshot. We will save every copy against a new **Snap ID**. So, after taking a snapshot, we will increment the **Snap ID**.

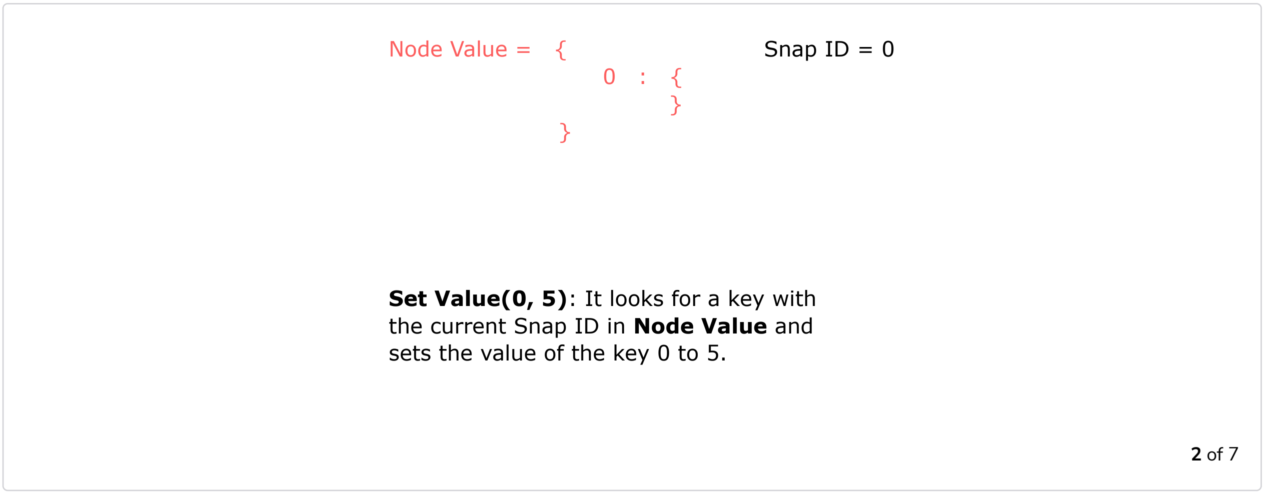
By using a **Snap ID** and a nested dictionary, we can save the current state of the nodes at every snapshot and retrieve the state of any node at any given **Snap ID**. The **Node Value** dictionary with the nested dictionary structure holds the state of all nodes at each snapshot. Each snapshot corresponds to a unique **Snap ID**, and the state of the nodes at each snapshot is saved as a copy in the nested dictionary with the corresponding Snap ID. This ensures that the previous states of the nodes are saved and can be retrieved when required.



```
Node Value =  {                    Snap ID = 0
                 0  :  {
                       }
              }
```

SnapshotArray(3)
Initializes an empty subdictionary with a key of value 0 in Node Value.

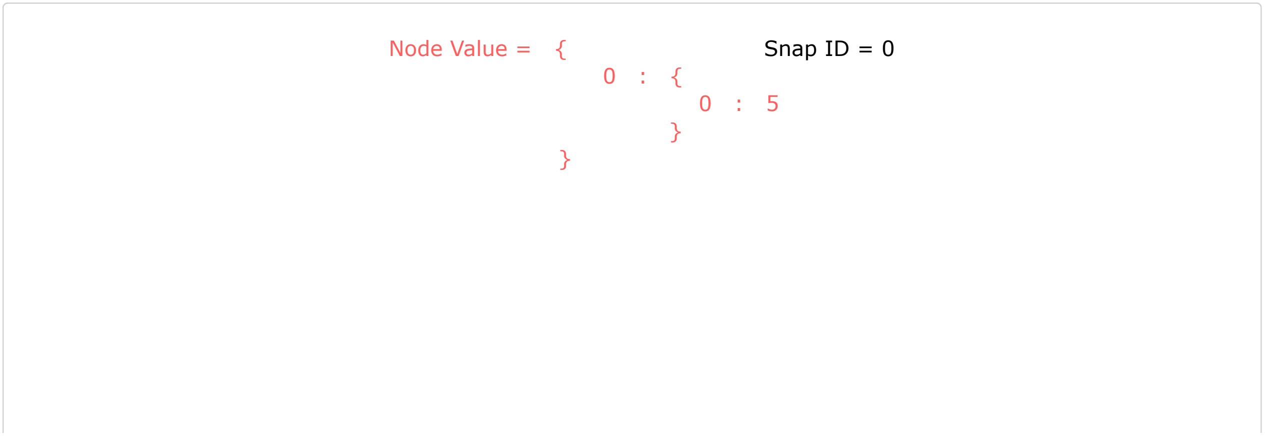
1 of 7



```
Node Value =  {                    Snap ID = 0
                 0  :  {
                       }
              }
```

Set Value(0, 5): It looks for a key with the current Snap ID in **Node Value** and sets the value of the key 0 to 5.

2 of 7



```
Node Value =  {                    Snap ID = 0
                 0  :  {
                          0  :  5
                       }
              }
```

```
Node Value =  {                   Snap ID = 0
                0  :  {
                         0  :  5
                      }
              }
```

Snapshot()
Increments the Snap ID by one and creates a new dictionary in **Node Value**, with the values of key with [Snap ID - 1].

```
Node Value =  {                   Snap ID = 1
                0  :  {
                         0  :  5
                      }
                1  :  {
                         0  :  5
                       }
              }
```

```
Node Value =  {                   Snap ID = 1
                0  :  {
                         0  :  5
                      }
                1  :  {
                         0  :  5
                       }
              }
```

Set Value(0, 6): It looks for a key with the current Snap ID in **Node Value** and sets the value of the key 0 to 6.

```
Node Value =  {                   Snap ID = 1
                0  :  {
                         0  :  5
                      }
                1  :  {
                         0  :  6
                       }
              }
```

Java

```java
import java.util.*;

class SnapshotArray {
    private int snapid;
    private Map<Integer, Map<Integer, Integer>> nodeValue;
    private int ncount;

    // Constructor
    public SnapshotArray(int length) {
        snapid = 0;
        nodeValue = new HashMap<Integer, Map<Integer, Integer>>();
        nodeValue.put(0, new HashMap<Integer, Integer>());
        ncount = length;
    }

    // Function setValue sets the value at a given index idx to val.
    public void setValue(int idx, int val) {
        if (idx < ncount) {
            nodeValue.get(snapid).put(idx, val);
        }
    }

    // This function takes no parameters and returns the snapid.
    // snapid is the number of times that the snapshot() function was called minus 1.
    public int snapshot() {
        nodeValue.put(snapid + 1, new HashMap<Integer, Integer>(nodeValue.get(snapid)));
        snapid++;
        return snapid - 1;
```

Snapshot Array

In the **Get Value** function, we first check if the input **Snap ID** is less than the total number of snapshots taken so far and if it is greater than or equal to zero. This is done to ensure that the input **Snap ID** is valid and corresponds to an actual snapshot.

If the input **Snap ID** is valid, we check if the input index **idx** is less than the total length of the array. If it is, then we check if the input index exists in the nested dictionary with the corresponding **Snap ID**. If it does, we return the value of the node at that index and **Snap ID**. If it does not, we return 0 as the node value.

If the input **Snap ID** is not valid, we return NULL.

```
Node Value =  {
                0  :  {
                         0  :  5
                      }
                1  :  {
                         0  :  6
                      }
              }
```

The snapshots above are taken at different times. To access any snapshot node value, we will use the **Get Value** function.

1 of 2

```
Node Value =  {
                 0  :  {
                          0  :  5
                       }
                 1  :  {
                          0  :  6
                        }
              }
```

Get Value(0, 0): It looks for a key with the given Snap ID in **Node Value** and gets the value of the given index. Therefore, the value at snapid 0 with node index 0 is 5.

2 of 2

Java

```java
            System.out.println(i + 1 + ".\tInitializing a data structure with " + numNodes.get(i) + " nod
            SnapshotArray snapshotArr = new SnapshotArray(numNodes.get(i));

            for (int j = 0; j < nodesValue.get(i).size(); j++) {
                for (int k = 0; k < nodesValue.get(i).get(j).size(); k++) {
                    System.out.println("\t\tSetting the value of node " + nodesValue.get(i).get(j).get(0)
                            " to " + nodesValue.get(i).get(j).get(1));
                    snapshotArr.setValue(nodesValue.get(i).get(j).get(0), nodesValue.get(i).get(j).get(1)
                    System.out.println("\t\tDictionary: " + snapshotArr.nodeValue);
                    System.out.println("\t\tSnap id: " + snapshotArr.snapshot() + "\n");
                    break;
                }
            }

            for (int x = 0; x < valueToGet.get(i).size(); x++) {
                for (int y = 0; y < valueToGet.get(i).get(x).size(); y++) {
                    System.out.println("\t\tNode value at index " + valueToGet.get(i).get(x).get(0) +
                            " with snapID: " + valueToGet.get(i).get(x).get(1) +
                            " is: " + snapshotArr.getValue(valueToGet.get(i).get(x).get(0), valueToGet.ge
                    break;
                }
            }
            System.out.println(new String(new char[100]).replace('\0', '-'));
        }
    }
}

```

Snapshot Array

Just the code

Here's the complete solution to this problem:

Java

```java
import java.util.*;

class SnapshotArray {
    private int snapid;
    private Map<Integer, Map<Integer, Integer>> nodeValue;
    private int ncount;

    // Constructor
    public SnapshotArray(int length) {
        snapid = 0;
        nodeValue = new HashMap<Integer, Map<Integer, Integer>>();
        nodeValue.put(0, new HashMap<Integer, Integer>());
        ncount = length;
    }

    // Function setValue sets the value at a given index idx to val.
    public void setValue(int idx, int val) {
        if (idx < ncount) {
            nodeValue.get(snapid).put(idx, val);
        }
    }

    // This function takes no parameters and returns the snapid.
    // snapid is the number of times that the snapshot() function was called minus 1.
    public int snapshot() {
        nodeValue.put(snapid + 1, new HashMap<Integer, Integer>(nodeValue.get(snapid)));
        snapid++;
        return snapid - 1;
```

1. Create a constructor that initializes the data structure to hold the specified number of indexes.
2. Create a function, **Set Value (idx, val)**, that sets the value at a given index to value.
3. Create a function, **Snapshot()**, that makes a copy of all the key-value pairs in the snapshot array and stores it as the latest snapshot, returning the count of snapshots taken so far.
4. Create a function, **Get Value (idx, Snap ID)**, that returns the value at the index in the snap with the given **Snap ID**.

Time complexity

Let n be the number of nodes and m be the number of snapshots.

- **Constructor:** The time complexity of the constructor is $O(1)$.
- **Get Value (idx, Snap ID):** The time complexity for this function is $O(1)$.
- **Set Value(idx, val):** The time complexity for this function is $O(1)$.
- **Snapshot():** The time complexity for this function is $O(n)$, as we are creating the deep copy of the dictionary.

Space complexity

To solve the problem given above, we will use $O(n * m)$ space, where n will be the number of nodes and m will be the number of snapshots taken.

← Back
Snapshot Array

Next →
Time-Based Key-Valu...

Mark as Completed