1656. Design an Ordered Stream

Easy Design Array Hash Table Data Stream Leetcode Link

Problem Description

The problem deals with a stream of `n` unique `(idKey, value)` pairs where `idKey` is an integer ranging from `1` to `n`, and `value` is a string. The objective is to design a system that can accept these pairs in any arbitrary order and return chunks of values sorted by their `idKey`. Additionally, as we process and insert new `(idKey, value)` pairs, we should return the largest chunk of consecutive `idKey` values, in increasing order, that have been inserted up to that point.

The problem requires us to implement a class that can track the order of incoming data pairs and return the sorted output in parts (chunks), without waiting for all data to be inserted.

Intuition

To solve this, we need a way to keep track of the inserted values and know which value corresponds to which `idKey`. As these values can be inserted in any order, a simple list can store the values at their respective `(idKey - 1)` index (since array indices are 0-based, but our `idKeys` are 1-based).

The main challenge is figuring out whether we have a contiguous sequence of `idKey` values from the current pointer position. To handle this, we maintain a pointer that starts at 0 and only moves forward when we add a new value that fills the gap.

When we insert a value, we'll place it at the `idKey - 1` index of our data array. Then, we check from the current pointer's position forward to see if we have consecutive values without any gaps. We keep moving the pointer and collecting values until we hit an `idKey` that hasn't been filled yet. This collection of values is the chunk we want to return.

The idea is similar to having a lock with rotating disks, each disk representing an `idKey` with its respective value, and the pointer aligning the next open slot. When all 'disks' up to a particular point are aligned (values are filled), we can return the `idKeys` and their values in order up to that point.

Solution Approach

The solution uses a simple array-based approach to store the incoming values. This approach efficiently solves the problem by exploiting the `idKey` to index mapping and a pointer to keep track of the next `idKey` that should be output.

The `OrderedStream` class initializes an array (or list in Python) to hold `n` values, which are initially set to `None` to indicate they have not been filled yet. The `ptr` variable is used as a pointer to the current position we expect the next `idKey` to fill.

The core of the solution is in the `insert` method, which has the following steps:

1. **Insert Value**: Store the value at the `idKey - 1` index of the `data` array, since `idKey` is 1-based and the array index is 0-based.
2. **Get Chunk**: Once the value is inserted, we need to collect a chunk of consecutive values, starting from where the `ptr` points. So we initiate an empty list `ans` to store the chunk of values.
3. **Advance Pointer**: Starting from `ptr`, iterate through the `data` array until you find an `idKey` that has not been filled (contains `None`). During this iteration, add the non-`None` values to the `ans` list and increment `ptr` after each non-`None` value is found. This step is crucial since it moves the `ptr` past the values that have been used to form the current chunk.
4. **Return Chunk**: Once a `None` value is encountered, or the end of the list is reached, stop collecting values and return the `ans` list. This list represents the largest possible chunk of values that can be formed in consecutive `idKey` order at this point in the stream.

The algorithm's efficiency comes from its direct use of the `idKey` as an array index and its linear scan from the `ptr` position to identify the contiguous sequence. No sorting is necessary because the `idKey` already indicates where the value belongs, and the process only involves inserting and scanning forward.

The overall time complexity for each insert operation is O(n) in the worst case, where `n` is the number of values the stream is set to contain.

Here is the crucial part of the implementation:

```python
def insert(self, idKey: int, value: str) -> List[str]:
    self.data[idKey - 1] = value
    ans = []
    while self.ptr < len(self.data) and self.data[self.ptr]:
        ans.append(self.data[self.ptr])
        self.ptr += 1
    return ans
```

In this snippet, the `insert` method implements the solution approach, ensuring that a chunk of consecutive ordered values is returned each time a new `(idKey, value)` pair is inserted into the stream.

Example Walkthrough

Let's say we're given a stream with `n = 5` unique `(idKey, value)` pairs, and they are inserted in the following order: `(3, "C")`, `(1, "A")`, `(5, "E")`, `(4, "D")`, and `(2, "B")`. We will use the solution approach to handle the stream of data and illustrate how the chunks are returned after each insert.

When we first initialize our `OrderedStream` for `n = 5`, our data array and `ptr` look like this:

```
data: [None, None, None, None, None]
ptr: 0
```

First Insertion: A pair `(3, "C")` is inserted.

1. We put value `"C"` at index `3 - 1` in our data array.
2. The `ptr` is still at `0`, and since `data[0]` is `None`, we can't form a chunk.
3. No chunk is returned.
4. Data and `ptr` are now:

```
data: [None, None, "C", None, None]
ptr: 0
```

Second Insertion: A pair `(1, "A")` is inserted.

1. We put value `"A"` at index `1 - 1`.
2. Now, `ptr` points to data `[0]` which is no longer `None`, and it's the start of a new chunk.
3. We collect values until we hit a `None` value, resulting in a chunk `["A"]`.
4. The `ptr` is incremented by `1`.
5. Data and `ptr` are now:

```
data: ["A", None, "C", None, None]
ptr: 1
```

Third Insertion: A pair `(5, "E")` is inserted.

1. We put value `"E"` at index `5 - 1`.
2. The `ptr` points to `data[1]` which is still `None`, so no new chunk can be formed.
3. No chunk is returned.
4. Data and `ptr` are unchanged:

```
data: ["A", None, "C", None, "E"]
ptr: 1
```

Fourth Insertion: A pair `(4, "D")` is inserted.

1. We put value `"D"` at index `4 - 1`.
2. Since `ptr` is still at `1` and `data[1]` is `None`, no consecutive chunk is formed.
3. No chunk is returned.
4. Data and `ptr` are unchanged:

```
data: ["A", None, "C", "D", "E"]
ptr: 1
```

Fifth Insertion: A pair `(2, "B")` is inserted.

1. We put value `"B"` at index `2 - 1`.
2. Now `ptr` at index `1` finds a non-`None` value, and we can start forming a new chunk.
3. We collect values starting from `ptr` - `["B", "C", "D"]` - and keep incrementing `ptr` for each non-`None` value.
4. The chunk forming stops as `data[4]` (for `idKey` 5) is `None`.
5. We return the chunk `["B", "C", "D"]`.
6. The `ptr` has moved to index `4`.

```
data: ["A", "B", "C", "D", None]
ptr: 4
```

As you can see, each insertion leads to the result of the `insert` function, which is the largest chunk of consecutive values that can be formed at that time. After all pairs have been inserted, we've managed to return all the chunks using the solution approach, and the `data` array contains all values sorted by their `idKeys`.

Python Solution

```python
from typing import List

class OrderedStream:
    def __init__(self, size: int):
        # Initialize the OrderedStream with a specified size.
        # - self.data stores the stream values initialized to None.
        # - self.pointer points to the next item to release in the stream.
        self.data = [None] * size
        self.pointer = 0

    def insert(self, id_key: int, value: str) -> List[str]:
        # Insert the value at the position one less than id_key, then
        # return a list of all consecutively inserted values starting
        # from the current pointer up to the first None encountered.

        # The id_key is 1-indexed so we convert it to 0-indexed for the list.
        self.data[id_key - 1] = value

        # Initialize an empty list to hold the consecutively inserted values.
        answer = []

        # Start from the pointer and go until the end of the data list.
        while self.pointer < len(self.data) and self.data[self.pointer]:
            # If the current pointer is not None, append the value to the answer.
            answer.append(self.data[self.pointer])
            # Move the pointer forward.
            self.pointer += 1

        # Return the list of consecutively inserted values.
        return answer

# Example of usage:
# obj = OrderedStream(size)
# output_values = obj.insert(id_key, value)
```

Java Solution

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

/**
 * OrderedStream class represents a stream of data that is intended to be received in a specific order, but can be inserted out of or
 */
class OrderedStream {
    private String[] data; // Array to store data.
    private int ptr; // Pointer to next element to retrieve from stream.

    /**
     * OrderedStream constructor initializes a new OrderedStream of size n.
     * @param n the size of the stream.
     */
    public OrderedStream(int n) {
        data = new String[n]; // Create an array to hold the strings.
        ptr = 0; // Set the pointer to the beginning of the stream.
    }

    /**
     * Inserts a value into the stream at the given idKey and returns all the values in the correct order starting from the pointer,
     * up to the first null value encountered.
     *
     * @param idKey the 1-based index at which the value should be inserted.
     * @param value the value to be inserted into the stream.
     * @return a list containing the ordered values of the stream from the pointer up to the first null value.
     */
    public List<String> insert(int idKey, String value) {
        // Convert 1-based index idKey to 0-based for the array access
        data[idKey - 1] = value;

        // Prepare the answer list to collect elements in sequence.
        List<String> ans = new ArrayList<>();
        // Traverse the stream from the pointer to the next null value.
        while (ptr < data.length && data[ptr] != null) {
            ans.add(data[ptr++]); // Add the non-null values to the answer list and increment the pointer.
        }
        return ans; // Return the list of retrieved values.
    }
}

// Example of how to use OrderedStream:
// OrderedStream os = new OrderedStream(5);
// List<String> output = os.insert(3, "ccccc"); // Inserts and retrieves ("ccccc").
// More insertions and retrievals can follow as described in the comments.
```

C++ Solution

```cpp
#include <vector>
#include <string>

// A class that represents a stream of data that can be ordered based on keys.
class OrderedStream {
private:
    std::vector<std::string> data; // Vector to hold the data stream.
    int ptr; // Pointer to keep track of the next element to output.

public:
    // Constructor that initializes the data stream of a given size and sets the pointer to zero.
    OrderedStream(int n) : ptr(0) {
        data.resize(n, ""); // All elements initialized to empty strings to indicate unfilled.
    }

    // Inserts a value into the stream at the position just before the idKey.
    // After insertion, it outputs a vector of all consecutive, available data values starting from ptr.
    std::vector<std::string> insert(int idKey, std::string value) {
        data[idKey - 1] = value; // IdKey is 1-based, so we need to decrement by one for 0-based indexing.
        std::vector<std::string> ans; // Vector to store consecutive values from ptr.

        // Loop through the data from ptr and collect all consecutive non-empty strings.
        while (ptr < data.size() && data[ptr] != "") {
            ans.push_back(data[ptr]); // Add the current element to ans.
            ptr++; // Move the pointer forward.
        }

        return ans; // Return the consecutive data starting from ptr.
    }
};

// Example of usage:
// OrderedStream* obj = new OrderedStream(n);
// std::vector<std::string> output = obj->insert(idKey, value);
```

Typescript Solution

```typescript
// A global pointer for the current position in the stream.
let ptr: number = 0;
// A global array to store the values in the stream.
let vals: string[];

/**
 * Initializes the stream with a specified size.
 * @param {number} n - The size of the stream.
 */
function createOrderedStream(n: number): void {
    ptr = 0;
    vals = new Array(n);
}

/**
 * Inserts a value into the stream at a specified key.
 * @param {number} idKey - The 1-based index at which to insert the value.
 * @param {string} value - The value to insert at the index.
 * @returns {string[]} An array of strings representing the values from the current
 *                     pointer position up to the last contiguous filled position.
 */
function insert(idKey: number, value: string): string[] {
    // Adjust the idKey from a 1-based to a 0-based index.
    const index = idKey - 1;
    vals[index] = value;

    // Create an array to hold the results.
    const result: string[] = [];

    // Add all contiguous non-null values starting from the current pointer position.
    while (vals[ptr] != null) {
        result.push(vals[ptr]);
        ptr++;
    }

    // Return the contiguous values found.
    return result;
}

// Example Usage:
// createOrderedStream(5);
// const result = insert(3, 'cc'); // Should return an empty array as it's not contiguous starting from ptr.
```

Time and Space Complexity

Time Complexity

The time complexity of the `__init__` method is `O(n)` as it initializes a list of size `n` with `None`.

The time complexity of the `insert` method is `O(n)` in the worst case. This worst-case scenario occurs when all the previous elements (from the current `ptr` to the `idKey - 1`) are filled in, and the method appends all of them to the `ans` list in a single call to `insert`. However, on average, assuming the inserts are distributed evenly, the complexity for each call would be `O(1)` as each inserted value would only cause a single write and at most one read (when the ptr immediately moves forward). The complexity of moving the pointer forward is `O(1)` for each step since it only involves checking a value and incrementing an index.

Space Complexity

The space complexity of the OrderedStream object is `O(n)`. This space is required to store the stream of data of size `n`. No additional significant space is used during the insert operations; the `ans` list temporarily holds a number of elements equal to the number of elements from `ptr` to the current `idKey`, but since it does not grow larger than `n`, it does not affect the overall space complexity.