284. Peeking Iterator

Medium Design Array Iterator

Leetcode Link

Problem Description

The task is to design a `PeekingIterator` that extends the functionality of a standard Iterator. Unlike a regular Iterator that only supports `next` and `hasNext` operations, the `PeekingIterator` should also support a `peek` operation. The `peek` method allows the user to look at the next element the Iterator would return without actually moving the Iterator forward. This can be particularly useful when you need to make decisions based on the next item without actually removing it from the iteration sequence.

Intuition

The intuitive approach to solving this problem involves caching the next element of the Iterator after a `peek` call, because a regular Iterator would not allow us to see the next element without advancing to it. Thus, we need an additional state in `PeekingIterator` to remember whether we've peeked at the next element and what that element is.

Here is a step-by-step break-down of the intuition behind the solution:

1. **Initialize State**: When we create a `PeekingIterator`, we store the original `Iterator` and initialize a state indicating whether the Iterator has been peeked or not (`has_peeked`), and a variable to store the peeked value (`peeked_element`).
2. **Peek Operation**: For the `peek` function we check if we have already called a `peek` without a subsequent `next`. If we haven't peeked yet (`has_peeked` is `False`), we call `next` on the underlying Iterator and store the value. We then return this stored value without changing the position of the underlying Iterator.
3. **Next Operation**: If a `peek` has not occurred, the regular `next` operation of the underlying Iterator is used. If we've already peeked (`has_peeked` is `True`), we need to return the stored `peeked_element` and reset our state so the Iterator can advance when `next` is called again.
4. **HasNext Operation**: For the `hasNext` method, we simply return `True` if we are in the peeked state (as we already have a next element), or we defer to the underlying Iterator's `hasNext` method.

The critical part of this solution is to carefully manage the `has_peeked` and `peeked_element` variables so as to not lose track of the Iterator's state or prematurely advance the underlying Iterator.

Solution Approach

The solution involves implementing a wrapper class around the provided Iterator to support the additional `peek` operation. We make use of the object-oriented concept of class to create our own `PeekingIterator` that maintains some internal state.

Here's a step-by-step walk through the implementation:

1. **Initialization**: In the constructor (`__init__` method), we accept an `Iterator` as an argument. We then initialize two instance variables:
   - `self.has_peeked`: A boolean that indicates if we have already peeked at the next element.
   - `self.peeked_element`: An integer to hold the next element after a peek. These variables are critical to manage the state between successive `peek` and `next` calls.
2. **Peek**: The `peek` method checks if we have already peeked (`self.has_peeked`), and if not, it 'peeks' ahead by calling `next` on the stored Iterator, saving this value to `self.peeked_element` and setting `self.has_peeked` to `True`. The value of `self.peeked_element` is returned.
3. **Next**: In the `next` method, we first check if `self.has_peeked` is `True`. If it is, we already have a peeked element to return, which we do. We then reset `self.has_peeked` to `False` and `self.peeked_element` to `None` to reflect that we no longer have a peeked value. If `self.has_peeked` is `False`, we directly return the next element from the underlying Iterator.
4. **HasNext**: For `hasNext`, we return `True` if either `self.has_peeked` is `True` (since we have a peeked element ready) or if the underlying Iterator has more elements, as indicated by its `hasNext` method.

This implementation uses lazy evaluation for the `peek` operation. The element is not advanced until necessary (i.e., until the `next` method is called). This strategy minimizes the number of calls to the underlying Iterator's `next` method, as it only advances the Iterator when we're certain we need to consume an element. This approach effectively implements a look-ahead buffer with a capacity of one element.

Such an implementation could be useful in scenarios like comparison-based sorting or merging, where you might need to compare the next item without yet removing it from the Iterator.

Example Walkthrough

Let's walk through an example to illustrate the solution approach:

Suppose we have an Iterator over a collection with elements `[1, 2, 3]`. When we wrap this Iterator with the `PeekingIterator`, here's how we can interact with it and what happens internally:

1. We instantiate `PeekingIterator` with our Iterator. `self.has_peeked` is initialized to `False`, and `self.peeked_element` is initialized to `None`.
2. We call `peek()` method. Since `self.has_peeked` is `False`, the `next()` method is called on the underlying Iterator, giving us `1`. Now `self.peeked_element` becomes `1`, and `self.has_peeked` is now `True`. The `peek()` method returns `1`, but our Iterator's position hasn't advanced.
3. We call the `next()` method. As `self.has_peeked` is `True`, it will return `self.peeked_element`, which is `1`. After this, `self.has_peeked` is set to `False` and `self.peeked_element` to `None`. The underlying Iterator's next element is now `2`.
4. We call `peek()` again. The process from step 2 repeats; `self.peeked_element` is set to `2`, `self.has_peeked` becomes `True`, and the method returns `2`.
5. We call `next()`. Since `self.has_peeked` is `True`, `self.peeked_element` (which is `2`) is returned. Both `self.has_peeked` and `self.peeked_element` are reset as before.
6. We call `hasNext()`. Since `self.has_peeked` is `False`, we check with the underlying Iterator, which still has an element (which is `3`), so `hasNext()` returns `True`.
7. We call `next()` and receive `3`, which is the last element of the Iterator.
8. Finally, when we call `hasNext()` again, it returns `False` because both `self.has_peeked` is `False` and the underlying Iterator has no further elements.

This example did not include calls to `hasNext()` prior to `next()` or `peek()` calls, but in practice, the `hasNext()` method can be used anytime to check if there is a next element available before calling `next()` or `peek()` to avoid exceptions from reaching the end of the Iterator.

Python Solution

```python
class Iterator:
    # A class that represents an iterator.
    def __init__(self, nums):
        """
        Initialize the iterator object to the beginning of a list.

        :param nums: A list of integers.
        """
        # Your Iterator class here.

    def has_next(self):
        """
        Checks if the iteration has more elements.

        :return: True if there are more elements, False otherwise.
        """
        # Your has_next logic here.

    def next(self):
        """
        Returns the next element in the iteration.

        :return: The next integer in the iteration.
        """
        # Your next logic here.

class PeekingIterator:
    def __init__(self, iterator):
        """
        Initialize a PeekingIterator with an iterator.

        :param iterator: An instance of the Iterator class.
        """
        self.iterator = iterator
        self.peeked_element = None  # The element that has been peeked at.
        self.has_peeked = False     # Flag to check if we have a peeked element.

    def peek(self):
        """
        Returns the next element in the iteration without advancing the iterator.

        :return: The next integer in the iteration.
        """
        if not self.has_peeked:
            # If we haven't peeked yet, retrieve the next element
            self.peeked_element = self.iterator.next()
            self.has_peeked = True  # Set the flag to indicate that we have peeked
        return self.peeked_element

    def next(self):
        """
        Returns the next element and advances the iterator.

        :return: The next integer in the iteration.
        """
        if not self.has_peeked:
            # If we haven't peeked, we simply get the next element from the iterator
            return self.iterator.next()

        # If we have peeked, we return the peeked element and reset the peeked state
        result = self.peeked_element
        self.peeked_element = None
        self.has_peeked = False
        return result

    def has_next(self):
        """
        Checks if there are more elements in the iteration.

        :return: True if there are more elements, False otherwise.
        """
        # We have an element if we already peeked or if the iterator has more elements
        return self.has_peeked or self.iterator.has_next()

# Usage example
# iter = PeekingIterator(Iterator(nums))
# while iter.has_next():
#     val = iter.peek()  # View the next element but do not advance the iterator.
#     iter.next()        # Should return the same value as [val].
```

Java Solution

```java
import java.util.Iterator;

// The PeekingIterator class wraps a standard Iterator and allows for peeking ahead at the next element without advancing the iterato
public class PeekingIterator<Integer> implements Iterator<Integer> {
    private Iterator<Integer> iterator; // The underlying iterator for a collection of Integers
    private boolean hasPeeked; // A flag indicating if we've already peeked at the next element
    private Integer peekedElement; // The next element, if we've peeked

    // Constructor initializes the PeekingIterator using an existing Iterator.
    public PeekingIterator(Iterator<Integer> iterator) {
        this.iterator = iterator;
    }

    // The peek method returns the next element in the iteration without advancing the iterator.
    public Integer peek() {
        if (!hasPeeked) {
            if (iterator.hasNext()) {
                peekedElement = iterator.next(); // Store the next element so we can return it later
            } else {
                throw new NoSuchElementException(); // In case there are no more elements to peek
            }
            hasPeeked = true;
        }
        return peekedElement;
    }

    // The next method should behave the same as if using the Iterator directly.
    @Override
    public Integer next() {
        if (!hasPeeked) {
            return iterator.next(); // Directly return the next element from the iterator
        }
        Integer result = peekedElement; // Return the stored peeked element
        hasPeeked = false; // Reset the peeked flag
        peekedElement = null; // Clear the stored element
        return result;
    }

    // The hasNext method indicates whether the iteration has more elements.
    @Override
    public boolean hasNext() {
        return hasPeeked || iterator.hasNext(); // Return true if we've peeked or if the iterator has more elements
    }
}
```

C++ Solution

```cpp
class PeekingIterator : public Iterator {
private:
    bool hasPeeked;         // Flag to check if we already peeked at the next element
    int peekedValue;        // Stores the value of the peeked element if hasPeeked is true

public:
    PeekingIterator(const vector<int>& nums)
        : Iterator(nums), hasPeeked(false)
    {
        // Constructor inherits from the Iterator and initializes hasPeeked flag to false.
        // Note that initialization of hasPeeked is done above in the member initializer list.
    }

    // Peek method allows us to see what the next element in the iterator is without advancing the iterator.
    int peek() {
        if (!hasPeeked) {
            // If we haven't already peeked, fetch the next element and set the hasPeeked flag
            peekedValue = Iterator::next(); // Get the next element from the iterator
            hasPeeked = true;               // Set the peek flag
        }
        return peekedValue; // Return the peeked value
    }

    // Override the next() method to consider if we have already peeked at the next element.
    int next() {
        if (!hasPeeked) {
            // If we haven't peeked, return the next element from the iterator directly
            return Iterator::next();
        }
        // If there is a peeked value, reset the hasPeeked flag since next() consumes the peeked element
        hasPeeked = false;
        return peekedValue; // Return the peeked value which is the next element
    }

    // Override the hasNext() method to consider the peek scenario.
    bool hasNext() const {
        // Either we have a peeked element or if the underlying iterator has a next element,
        // we have a next element.
        return hasPeeked || Iterator::hasNext();
    }
};
```

Typescript Solution

```typescript
let hasPeeked: boolean = false;           // Flag to check if we already peeked at the next element
let peekedValue: number;                  // Stores the value of the peeked element if hasPeeked is true
let iterator: Iterator<number>;           // The iterator instance which is adopted by the PeekingIterator functionality

function PeekingIterator(nums: number[]) {
    // Initialize the iterator with nums.
    iterator = nums.values().iterator();
}

function peek(): number {
    if (!hasPeeked) {
        // If we haven't already peeked, fetch the next element and set the hasPeeked flag
        const result = iterator.next();
        if (!result.done) {
            peekedValue = result.value;
            hasPeeked = true;             // Set the peek flag
        }
    }
    return peekedValue; // Return the peeked value
}

function next(): number {
    if (!hasPeeked) {
        // If we haven't peeked, extract the next element from the iterator directly and return it
        const result = iterator.next();
        if (!result.done) {
            return result.value;
        }
    } else {
        // If there is a peeked value, reset the hasPeeked flag since next() consumes the peeked element
        hasPeeked = false;
        return peekedValue; // Return the peeked value which is the next element
    }
}

function hasNext(): boolean {
    // If there is a peeked element or if the underlying iterator has a next element, we have a next element
    return hasPeeked || !!iterator.next().done;
}
```

Time and Space Complexity

Time Complexity:

- `PeekingIterator.__init__(self, iterator)`:
  - This method only performs some basic variable assignments. The time complexity here is `O(1)`, since no loops or significant operations are involved.
- `PeekingIterator.peek(self)`:
  - If we have not peeked yet (`self.has_peeked` is `False`), we call `self.iterator.next()` once. Since the `next()` operation of the underlying Iterator is assumed to be `O(1)`, the overall time complexity for `peek()` is also `O(1)`. If we already have a peeked element, we just return it, which is again an `O(1)` operation.
- `PeekingIterator.next(self)`:
  - When we haven't peeked (`self.has_peeked` is `False`), the time complexity is `O(1)` as it directly returns the next element by calling the Iterator's `next()` method. Once we have peeked (`self.has_peeked` is `True`), it involves returning the peeked element and resetting `has_peeked` and `peeked_element` which is also `O(1)` time.
- `PeekingIterator.hasNext(self)`:
  - This method checks whether there's a next element by either checking the `self.has_peeked` attribute or calling the `hasNext()` method of the underlying Iterator, both of these operations have a time complexity of `O(1)`.

Space Complexity:

- Overall space complexity of the `PeekingIterator` class:
  - The additional space used by the `PeekingIterator` class consists of two variables: `self.has_peeked` and `self.peeked_element`, which take up `O(1)` space. The underlying Iterator (`self.iterator`) object is not counted towards the space complexity as it is an input to the class. Therefore, the space complexity of the `PeekingIterator` is `O(1)` as it uses a constant amount of extra space.

In conclusion, both the operations `peek()`, `next()`, and `hasNext()` in the `PeekingIterator` class have a time complexity of `O(1)` and the space complexity of the `PeekingIterator` class is `O(1)`.