284. Peeking Iterator

Feb. 22, 2020 | 6.5K views

Average Rating: 4.84 (19 votes)

Given an Iterator class interface with methods: `next()` and `hasNext()`, design and implement a PeekingIterator that support the `peek()` operation -- it essentially peek() at the element that will be returned by the next call to next().

Example:

```
Assume that the iterator is initialized to the beginning of the list: [1,2,3].

Call next() gets you 1, the first element in the list.
Now you call peek() and it returns 2, the next element. Calling next()
 after that still return 2.
You call next() the final time and it returns 3, the last element.
Calling hasNext() after that should return false.
```

Follow up: How would you extend your design to be generic and work with all types, not just integer?

Solution

Overview - What is an Iterator?

Feel free to skip this section if you're already familiar with this material. We have included it, as this is a beginners question on iterators.

If you've heard of `Iterator`s, you might assume they're simply a way of iterating over indexed or finite data structures. You've probably used them in loops, e.g. `for item in items:` in Python or `for (int num : nums)` in Java.

This may make it seem strange that we would need a `.peek()` on an `Iterator`. After all, couldn't we just convert our data structure into an array and use indexing to peek?

But actually `Iterator`s have some interesting properties that make them widely useful for not only indexed collections (e.g. Array) and other finite data structures (e.g. `LinkedList` or `HashMap` keys), but also for (possibly-infinite) generated data. We'll look at an example of that soon.

The first property of an `Iterator` that we'll looked at is that it only needs to know how to get the next item. It doesn't need to store the entire data in memory if we don't need the entire data structure. For massive data structures, this is invaluable!

For example consider a linked list `Iterator`. We'll use Python as it's nice and compact. The same ideas apply to Java and C++:

```
class LinkedListIterator:
    def __init__(self, head):
        self._node = head

    def hasNext(self):
        return self._node is not None

    def next(self):
        result = self._node.value
        self._node = self._node.next
        return result
```

Notice how we store the head at the start, but as items are consumed, we discard the current one and replace it with the item in the node after?

This means that if we're simply iterating a Linked List, and don't ever need to go back to the head, then we only need to keep one value around at a time.

Another really interesting property of `Iterator`s is that they can represent sequences without even using a data structure!

For example consider a range `Iterator`:

```
class RangeIterator:
    def __init__(self, min, max):
        self._max = max
        self._current = min

    def hasNext(self):
        return self._current < self._max

    def next(self):
        self._current += 1
        return self._current - 1
```

If we simply converted this to an Array, we'd need to allocate a large chunk of memory if `min` and `max` are far apart. For the most part, this would probably be wasted space.

However, by using an `Iterator`, we can use features like `for i in range(40, 20000000)` while still retaining the $O(1)$ space of classic `for (int i = min; i < max; i++)` style loops seen in other languages.

Our final property is one that we couldn't even do by copying values into an Array—handling an infinite sequence. For example consider an `Iterator` of squares:

```
class SquaresIterator:
    def __init__(self):
        self._n = 0

    def hasNext(self):
        # It continues forever,
        # so definitely has a next!
        return True

    def next(self):
        result = self._n
        self._current += 1
        return result ** 2
```

Notice that because `.hasNext()` always returns `True`, this `Iterator` will never run out of items. And this is to be expected, there's always another square after the previous, so our `Iterator` can give as many as we ask from it.

Now that we've looked at why `Iterator`s are awesome, let's consider what they are at a base level.

An `Iterator` only provides two methods:

- `.next()` This returns the next item in the sequence. You can't assume that this item actually "exists" yet, it might be created when you call `.next()`, or it might already exist in a data structure that you have an `Iterator` over.

  Once `.next()` is called, it will update the state of the `Iterator`. This means once you've got a value from `.next()` you won't be able to get the same value again. Therefore, if you don't store or process the value you got from the `Iterator` then it's possibly gone forever!

- `.hasNext()` This returns whether or not another item is available. For example, an array `Iterator` should return `False` if we're at the end of the array. But for an `Iterator` that can produce items indefinitely, such as our square generator above, it might never return `False`.

A further property of `Iterator`s is that they provide a clean interface for the code using them. Without `Iterator`s, Linked List's, for example, tend to be particularly messy, as the code for traversing them gets muddled within the application code. With an `Iterator`, the external code doesn't even know how the underlying data structure works. For all it knows, the data could be coming from a Linked List, an Array, a Tree, a State Machine, a clever number generator, a file reader, a robot sensor, etc.

Not having to deal with nodes, state, indexes, etc leads to clean code. We call this the *Iterator Pattern*, and it is one of the most important design patterns for a software engineer to know.

As shown above, with only two methods, we get a lot of benefit (e.g. infinite sequences) and increased performance (e.g. not expanding sequences like range into arrays). We also get a nice way of separating the underlying data structure from the code that uses it.

Approach 1: Saving Peeked Value

Intuition

Each time we call `.next(...)`, a value is returned from the `Iterator`. If we call `.next(...)` again, a new value will be returned. This means that if we want to use a particular value multiple times, we had better save it.

Our `.peek(...)` method needs to call `.next(...)` on the `Iterator`. But because `.next(...)` will return a different value next time, we need to store the value we peeked so when `.next(...)` is called we return the correct value.

Algorithm

Java | Python | Copy

```
class PeekingIterator:
    def __init__(self, iterator):
        self._iterator = iterator
        self._peeked_value = None

    def peek(self):
        # If there's not already a peeked value, get one out and store
        # it in the _peeked_value variable. We aren't told what to do if
        # the iterator is actually empty -- here I have thrown an exception
        # but in an interview you should definitely ask! This is the kind of
        # thing they expect you to ask about.
        if self._peeked_value is None:
            if not self._iterator.hasNext():
                raise StopIteration()
            self._peeked_value = self._iterator.next()

        return self._peeked_value

    def next(self):
        # Firstly, we need to check if we have a value already
        # stored in the _peeked_value variable. If we do, we need to
        # return it and also set _peeked_value to null so that the value
        # isn't returned again.
        if self._peeked_value is not None:
            to_return = self._peeked_value
            self._peeked_value = None
            return to_return
```

Complexity Analysis

- Time Complexity : All methods: $O(1)$.

  The operation performed to update our peeked value are all $O(1)$.

  The actual operations from `.next()` are impossible for us to analyse, as they depend on the given `Iterator`. By design, they are none of our concern. Our addition to the time is only $O(1)$ though.

- Space Complexity : All methods: $O(1)$.

  Like with time complexity, the `Iterator` itself is probably using memory in its own implementation. But again, this is not our concern. Our implementation only uses a few variables, so is $O(1)$.

Approach 2: Saving the Next Value

Intuition

Instead of only storing the next value after we've peeked at it, we can store it immediately in the constructor and then again in the `next(...)` method. This greatly simplifies the code, because we no longer need conditionals to check whether or not we are currently storing a peeked at value.

Algorithm

Note that in the Java code, we need to be careful not to cause an exception to be thrown from the *constructor*, in the case that the `Iterator` was empty at the start. We can do this by checking it has a next, and if it doesn't, then we set the next variable to `null`.

Java | Python | Copy

```
class PeekingIterator:
    def __init__(self, iterator):
        self._next = iterator.next()
        self._iterator = iterator

    def peek(self):
        return self._next

    def next(self):
        if self._next is None:
            raise StopIteration()
        to_return = self._next
        self._next = None
        if self._iterator.hasNext():
            self._next = self._iterator.next()
        return to_return

    def hasNext(self):
        return self._next is not None
```

Complexity Analysis

- Time Complexity : All methods: $O(1)$.

  Same as Approach 1.

- Space Complexity : All methods: $O(1)$.

  Same as Approach 1.

The Follow Up

For the most part, our code would work fine if we replaced integers with another data type (e.g. strings).

There is one case where this does not work, and that's if the underlying `Iterator` might return `null` / `None` from `.next(...)` as an actual value. If our code is using `null` to represent an exhausted `Iterator`, or to represent that we don't currently have a peeked value stored away (as in Approach 1), then the conditionals in `PeekingIterator` will not behave as expected on these values coming out of the underlying `Iterator`.

We can solve it by using separate `boolean` variables to state whether or not there's currently a peeked value or the `Iterator` is exhausted, instead of trying to infer this information based on `null` status of value variables.

In Java, you can also use *generics* on your `Iterator`.