Previous Next

604. Design Compressed String Iterator

June 10, 2017 | 12.1K views

Average Rating: 4.64 (11 votes)

Design and implement a data structure for a compressed string iterator. It should support the following operations: `next` and `hasNext`.

The given compressed string will be in the form of each letter followed by a positive integer representing the number of this letter existing in the original uncompressed string.

`next()` - if the original string still has uncompressed characters, return the next letter; Otherwise return a white space.
`hasNext()` - Judge whether there is any letter needs to be uncompressed.

Note:
Please remember to **RESET** your class variables declared in StringIterator, as static/class variables are **persisted across multiple test cases**. Please see here for more details.

Example:

```
StringIterator iterator = new StringIterator("L1e2t1C1o1d1e1");

iterator.next(); // return 'L'
iterator.next(); // return 'e'
iterator.next(); // return 'e'
iterator.next(); // return 't'
iterator.next(); // return 'C'
iterator.next(); // return 'o'
iterator.next(); // return 'd'
iterator.hasNext(); // return true
iterator.next(); // return 'e'
iterator.hasNext(); // return false
iterator.next(); // return ' '
```

Solution

Approach #1 Uncompressing the String [Time Limit Exceeded]

Algorithm

In this approach, we make use of precomputation. We already form the uncompressed string and append the uncompressed letters for each compressed letter in the $compressedString$ to the res stringbuilder. To find the uncompressed strings to be stored in res, we traverse over the given $compressedString$. Whenver we find an alphabet, we find the number following it by making use of decimal mathematics. Thus, we get the two elements(alphabet and the count) required for forming the current constituent of the uncompressed string.

Now, we'll look at how the `next()` and `hasNext()` operations are performed:

1. `next()`: We start off by checking if the compressed string has more uncompressed letters pending. If not, `hasNext()` returns a False value and `next()` returns a ' '. Otherwise, we return the letter pointed by ptr, which indicates the next letter to be returned. Before returning the letter, we also update the ptr to point to the next letter in res.

2. `hasNext()`: If the pointer ptr reaches beyond the end of res array, it indicates that no more uncompressed letters are left beyond the current index pointed by ptr. Thus, we return a False in this case. Otherwise, we return a True value.

Java Copy

```java
public class StringIterator {
    StringBuilder res=new StringBuilder();
    int ptr=0;
    public StringIterator(String s) {
        int i = 0;
        while (i < s.length()) {
                char ch = s.charAt(i++);
                int num = 0;
                while (i < s.length() && Character.isDigit(s.charAt(i))) {
                    num = num * 10 + s.charAt(i) - '0';
                    i++;
                }
                for (int j = 0; j < num; j++)
                    res.append(ch);
        }
    }
    public char next() {
        if (!hasNext())
            return ' ';
        return res.charAt(ptr++);
    }
    public boolean hasNext() {
        return ptr!=res.length();
    }
}
```

Performance Analysis

- We precompute the elements of the uncompressed string. Thus, the space required in this case is $O(m)$, where m refers to the length of the uncompressed string.
- The time required for precomputation is $O(m)$ since we need to generate the uncompressed string of length m.
- Once the precomputation has been done, the time required for performing `next()` and `hasNext()` is $O(1)$ for both.
- This approach can be easily extended to include `previous()`, `last()` and `find()` operations. All these operations require the use an index only and thus, take $O(1)$ time. Operations like `hasPrevious()` can also be easily included.
- Since, once the precomputation has been done, `next()` requires $O(1)$ time, this approach is useful if `next()` operation needs to be performed a large number of times. However, if `hasNext()` is performed most of the times, this approach isn't much advantageous since precomputation needs to be done anyhow.
- A potential problem with this approach could arise if the length of the uncompressed string is very large. In such a case, the size of the complete uncompressed string could become so large that it can't fit in the memory limits, leading to memory overflow.

Approach #2 Pre-Computation [Accepted]

Algorithm

In this approach, firstly, we split the given $compressedString$ based on numbers(0-9) and store the values(alphabets) obtained in $chars$ array. We also split the $compressedString$ based on the alphabets(a-z, A-Z) and store the numbers(in the form of a string) in a $nums$ array(after converting the strings obtained into integers). We do the splitting by making use of regular expression matching.

A regular expression is a special sequence of letters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

This splitting using regex is done as a precomputation step. Now we'll look at how the `next()` and `hasNext()` operations are implemented.

1. `next()`: Every time the `next()` operation is performed, firstly we check if there are any more letters to be uncompressed. We check it by making use of `hasNext()` function. If there aren't any more letters left, we return a ' '. We make use of a pointer ptr to keep a track of the letter in the $compressedString$ that needs to be returned next. If there are more letters left in the uncompressed string, we return the current letter pointed to by ptr. But, before returning this letter, we also decrement the $nums[ptr]$ entry to indicate that the current letter is pending in the uncompressed string by one lesser count. On decrementing this entry, if it becomes zero, it indicates that no more instances of the current letter exist in the uncompressed string. Thus, we update the pointer ptr to point to the next letter.

2. `hasNext()`: For performing `hasNext()` operation, we simply need to check if the ptr has already reached beyong the end of $chars$ array. If so, it indicates that no more compressed letters exist in the $compressedString$. Hence, we return a False value in this case. Otherwise, more compressed letters exist. Hence, we return a True value in this case.

Java Copy

```java

import java.util.regex.Pattern;
public class StringIterator {
    int ptr = 0;
    String[] chars;int[] nums;
    public StringIterator(String compressedString) {
        nums = Arrays.stream(compressedString.substring(1).split("[a-zA-
Z]+")).mapToInt(Integer::parseInt).toArray();;
        chars = compressedString.split("[0-9]+");
    }
    public char next() {
        if (!hasNext())
            return ' ';
        nums[ptr]--;
        char res=chars[ptr].charAt(0);
        if(nums[ptr]==0)
            ptr++;
        return res;
    }
    public boolean hasNext() {
        return ptr != chars.length;
    }
}
```

Performance Analysis

- The space required for storing the results of the precomputation is $O(n)$, where n refers to the length of the compressed string. The $nums$ and $chars$ array contain a total of n elements.
- The precomputation step requires $O(n)$ time. Thus, if `hasNext()` operation is performed most of the times, this precomputation turns out to be non-advantageous.
- Once the precomputation has been done, `hasNext()` and `next()` requires $O(1)$ time.
- This approach can be extended to include the `previous()` and `hasPrevious()` operations, but that would require making some simple modifications to the current implementation.

Approach #3 Demand-Computation [Accepted]

Algorithm

In this approach, we don't make use of regex for finding the individual components of the given $compressedString$. We do not perform any form of precomputation. Whenever an operation needs to be performed, the required results are generated from the scratch. Thus, the operations are performed only on demand.

Let's look at the implementation of the required operations:

1. `next()`: We make use of a global pointer ptr to keep a track of which compressed letter in the $compressedString$ needs to be processed next. We also make use of a global variable num to keep a track of the number of instances of the current letter which are still pending. Whenever `next()` operation needs to be performed, firstly, we check if there are more uncompressed letters left in the $compressedString$. If not, we return a ' '. Otherwise, we check if there are more instances of the current letter still pending. If so, we directly decrement the count of instances indicated by $nums$ and return the current letter. But, if there aren't more instances pending for the current letter, we update the ptr to point to the next letter in the $compressedString$. We also update the num by obtaining the count for the next letter from the $compressedString$. This number is obtained by making use of decimal arithmetic.

2. `hasNext()`: If the pointer ptr has reached beyond the last index of the $compressedString$ and num becomes, it indicates that no more uncompressed letters exist in the compressed string. Hence, we return a False in this case. Otherwise, a True value is returned indicating that more compressed letters exist in the $compressedString$.

Java Copy

```java

public class StringIterator {
    String res;
    int ptr = 0, num = 0;
    char ch = ' ';
    public StringIterator(String s) {
        res = s;
    }
    public char next() {
        if (!hasNext())
            return ' ';
        if (num == 0) {
            ch = res.charAt(ptr++);
            while (ptr < res.length() && Character.isDigit(res.charAt(ptr))) {
                num = num * 10 + res.charAt(ptr++) - '0';
            }
        }
        num--;
        return ch;
    }
    public boolean hasNext() {
        return ptr != res.length() || num != 0;
    }
}

```

Performance Analysis

- Since no precomputation is done, constant space is required in this case.
- The time required to perform `next()` operation is $O(1)$.
- The time required for `hasNext()` operation is $O(1)$.
- Since no precomputations are done, and `hasNext()` requires only $O(1)$ time, this solution is advantageous if `hasNext()` operation is performed most of the times.
- This approach can be extended to include `previous()` and `hasPrevious()` operationsm, but this will require the use of some additional variables.

Analysis written by: @vinod23

Rate this article:

Previous Next

Abyss2018 ★ 21 June 11, 2017 11:03 PM
Hope in the next contests, this kind of questions should state that "positive integer representing the number of this letter could be very large!" (I waste a lot of time on brute force!)
2 Share Reply
SHOW 1 REPLY

aman25 ★ 30 August 10, 2019 10:39 PM
@vinod23 The last solution runs in an endless recursion for malicious test case like "L1e0t1C1o1d1e1", where one of the char counts is 0.
A better implementation is to keep the moveNext logic independent of the next function call and moveNext till the char count > 0.
Read More
1 Share Reply

Subhadeep2704 ★ 219 June 27, 2018 9:52 AM
Great solution - last one.
0 Share Reply

pbu ★ 258 March 9, 2020 1:52 AM
on demand java easy to read one:

```
class StringIterator {
```

Read More
0 Share Reply

lc19890306 ★ 36 February 21, 2020 8:21 AM
spacewise approach #2 and #3 has no difference
0 Share Reply

sunnyleevip ★ 52 December 17, 2019 6:49 AM
I have some doubt with **Performance Analysis** from the 3rd approach. As we need to save the compressedString, space complexity should be O(n). And to look into the function next(), combine all next() of one test case, it just traverse the whole string, which is the same thing we doing in the 2nd approach. So total time complexity would be the same as the 2nd approach. And each call of function next should be longer then the 2nd approach. The only advantage of the 3rd approach is we save the
Read More
0 Share Reply
SHOW 1 REPLY

nwadhwa12345 ★ 30 August 12, 2019 4:26 AM
``
Simple OnDemand Iterator storage--

class StringIterator {
String str;
Read More
0 Share Reply