Reverse Prefix of Word

LeetCode
23340 Apr 15, 2024

Editorial

Solution

Overview

We are given a string `word` and need to reverse the prefix that starts at index `0` and ends at the first occurrence of `ch` .

If the `word` does not contain `ch` , we return `word` unmodified.

In C++, we can use built-in functions to accomplish this:

This is not the most universal solution, as many programming languages do not have built-in string reverse capabilities. Additionally, strings are immutable in many programming languages.

> Immutable means something cannot be changed once it has been created.

This problem is intended to provide practice with string manipulation, so we focus on approaches that use string manipulation techniques.

Approach 1: Stack

Intuition

Whenever a problem requires reversing a sequence, it is worth considering using a stack.

Stacks are a First-In-Last-Out (FILO) data structure, which means that the first items added to the stack are the last items removed from the stack. This means that if you push a sequence of items into a stack, and then remove all of the items, the sequence of items will be reversed. Learn more about stacks by reading our Stack Explore Card.

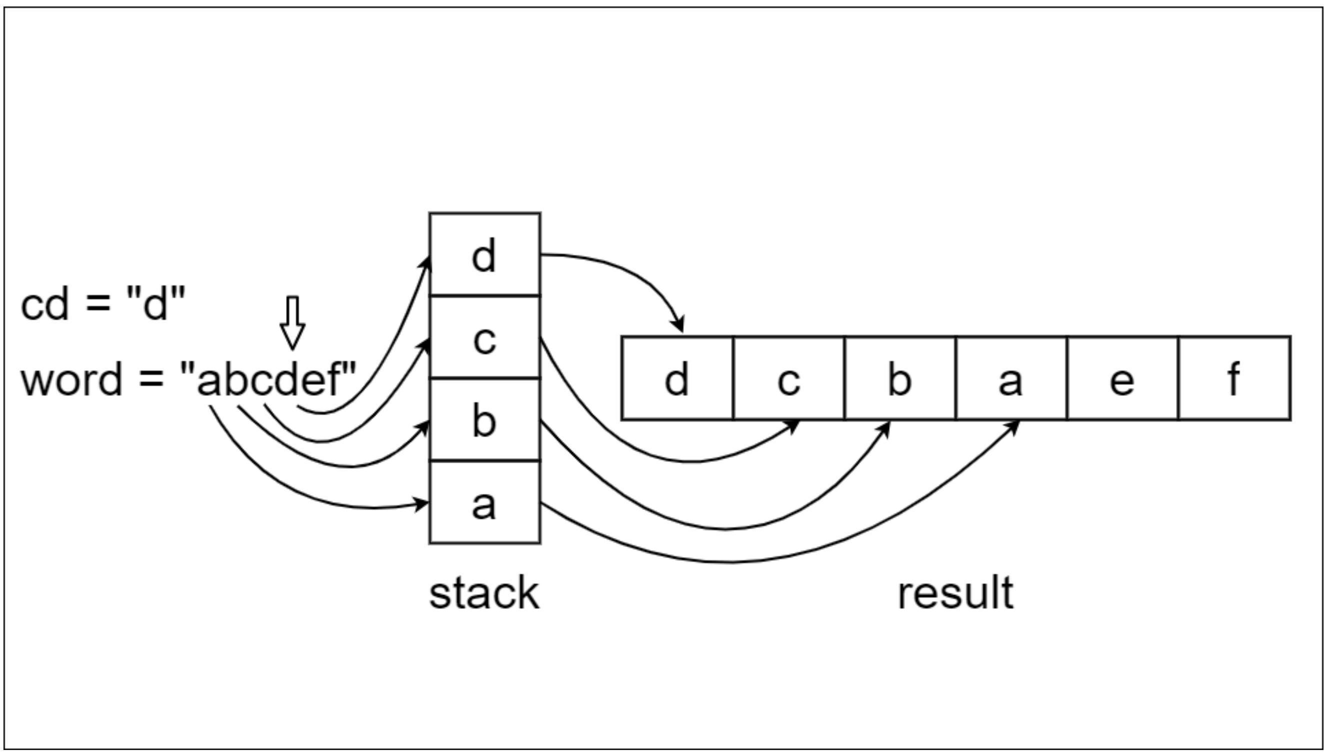
Since strings are immutable in many programming languages, we cannot directly modify the original string. Instead, we need to build a new string incrementally. In C++, we can use a string `result` to store the answer. In Python, we can use a list, and in Java, we can use a StringBuilder.

To reverse `word`, we loop through the characters of `word` and push each character onto the stack until we reach the first occurrence of `ch`.

Once we reach the character `ch`, we can start popping the characters off the stack and appending them to the `result` string. This will reverse the prefix of the `word`.

After we have emptied the stack, we can append the remaining characters of the `word` (i.e., the part of the word that comes after the first occurrence of `ch`) to the `result` string, in their original order.

Finally, we return `result`, converting it to a string if necessary. If `ch` was not found in `word`, we return the original `word` instead.



Algorithm

1. Initialize the following:
    - A `stack` to store characters that need to be reversed.
    - A string or list `result` for building the reversed string.
    - A variable `index` for iterating through the characters in `word`.
2. Loop through `word` until `index` reaches the end of `word`:

    - Push the character `word[index]` onto the `stack` .
    - If the current character equals `ch` :
        - Pop each of the characters from the stack and add them to the `result`
        - Increment `index` by `1` because we already added `ch` to the `result` .
        - Add the rest of the characters from `word` to `result` .
        - Return `result` and convert to a string if necessary.
    - Increment `index` by `1` ; we have not yet reached `ch` .
3. Return `word` , which does not contain `ch` .

Implementation

Complexity Analysis

Let n be the length of `word` .

- Time complexity: $O(n)$

  Finding `ch` in `word` and adding the characters to the stack takes up to $O(n)$ when `ch` is the last character in `word` .

  Adding the characters to `result` takes $O(n)$.

  Therefore, the time complexity is $O(2n)$, which we can simplify to $O(n)$.

- Space complexity: $O(n)$

  We use `stack` which can grow to contain up to n elements, so the space complexity is $O(n)$.

Approach 2: Find the Index and Fill Result

Intuition

We usually read text from left to right. Reading right to left, the text will appear reversed.

To reverse the prefix, we can "read" the prefix in reverse order (end to beginning and right to left).

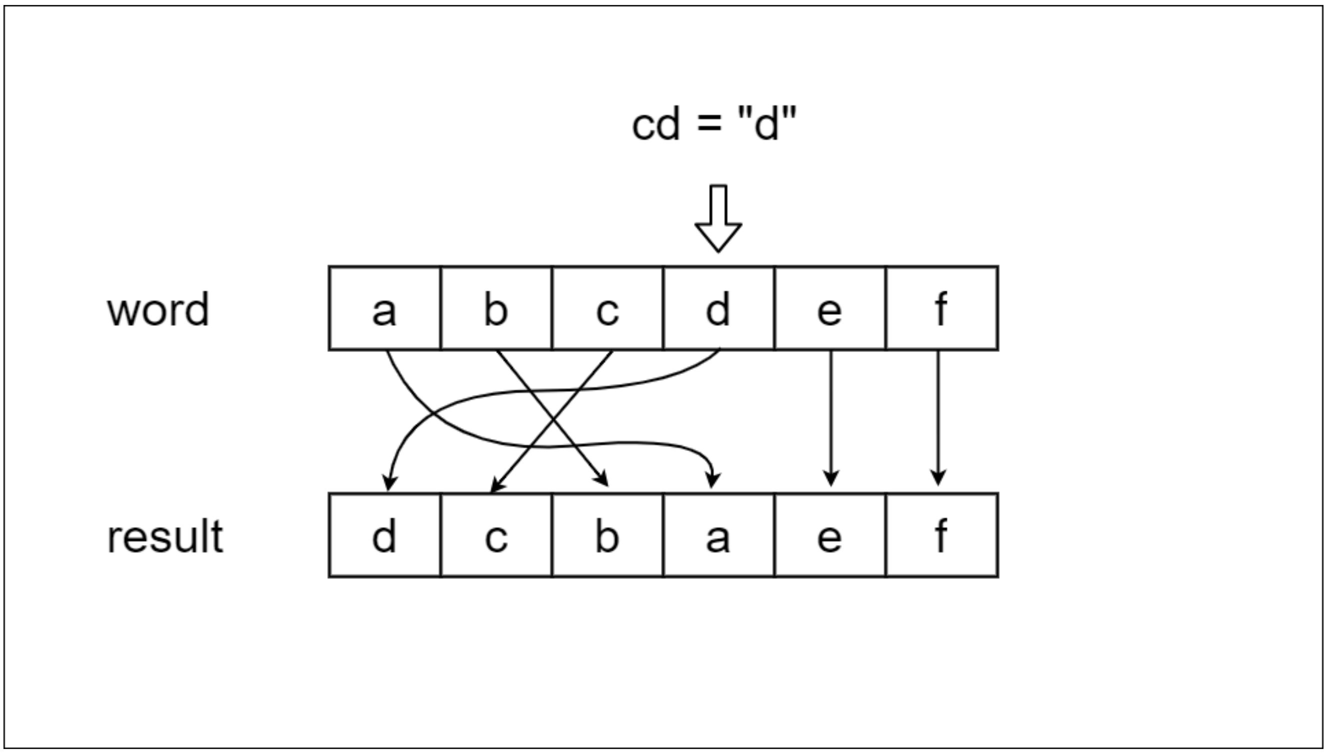
First, we need to find the end of the prefix. This will be the index in `word` of the first occurrence of `ch` . Most languages have a built-in function we can use to locate the index of a character in a string, which we will use to find `chIndex` .

If `ch` is not in `word` , we return `word` unaltered.

Similar to the previous solution, we will add characters to `result` one by one. To traverse `word` from the end of the prefix to the beginning while adding characters to `result` , we can use a standard `for` loop with `word[chIndex - i]` .

Once the prefix has been added to the `result` , we can then proceed with the `for` loop, appending `word[i]` to the result.

Finally, we return `result` , and if necessary, convert it to a string.



Algorithm

1. Find the index of `ch` in `word` and set the variable `chIndex` to this value.

2. If `chIndex` equals `-1` , `ch` is not in `word` , so return `word` .

3. Initialize a string or list `result` for building the string with the reversed prefix.

4. Loop through the characters of `word` using the iterator `i` :

    - If `i` is less than or equal to `chIndex` , the character at this index of `result` should be the corresponding character from `word` but in reverse. Append `word[chIndex - i]` to `result` .

    - Otherwise, the character at this index of `result` should contain a character in the original order. Append `word[i]` to `result`.
5. Return `result`, converting it to a string if necessary.

Implementation

Complexity Analysis

Let n be the length of `word`.

Certainly. I'll provide a more detailed explanation for each point:

- Time complexity: $O(n)$ or $O(n^2)$

  The time complexity varies across implementations:

  - In Java, using `StringBuilder` results in $O(n)$. The `append()` operation is $O(1)$, and we perform it n times.
  - In Python, string concatenation inside the loop leads to $O(n^2)$. Each '+=' operation creates a new string, taking $O(n)$ time, and we do this n times.
  - In C++, using `std::string` results in $O(n)$. The concetanation(+=) is $O(1)$ as C++ strings are mutable, and we perform it n times.

- Space complexity: $O(n)$

  We use `result`, which has a size of n, to build the answer.

  The space usage is consistent across implementations:

  - In Java, using `StringBuilder` results in $O(n)$. It preallocates space efficiently.
  - In Python, despite creating multiple strings, only one full-length string exists at any time, so $O(n)$. However, it may use more memory during execution due to the creation of temporary strings.
  - In C++, using `std::string` results in $O(n)$. C++ strings are mutable so the performance considerations of concatenation(+=) are less of a concern.

  Therefore, the space complexity is $O(n)$ for all implementations. While the actual memory usage may vary slightly, the asymptotic space complexity remains the same.

> Note: The main difference in efficiency comes from how each language handles string manipulations. Java's StringBuilder and C++'s string are optimized for repeated concatenations, while Python's strings, being immutable, require more operations for the same task.

Approach 3: Two-Pointer Swapping

Intuition

When we reverse a string, the characters at the ends are swapped. Likewise, the characters one spot away from the ends are swapped.

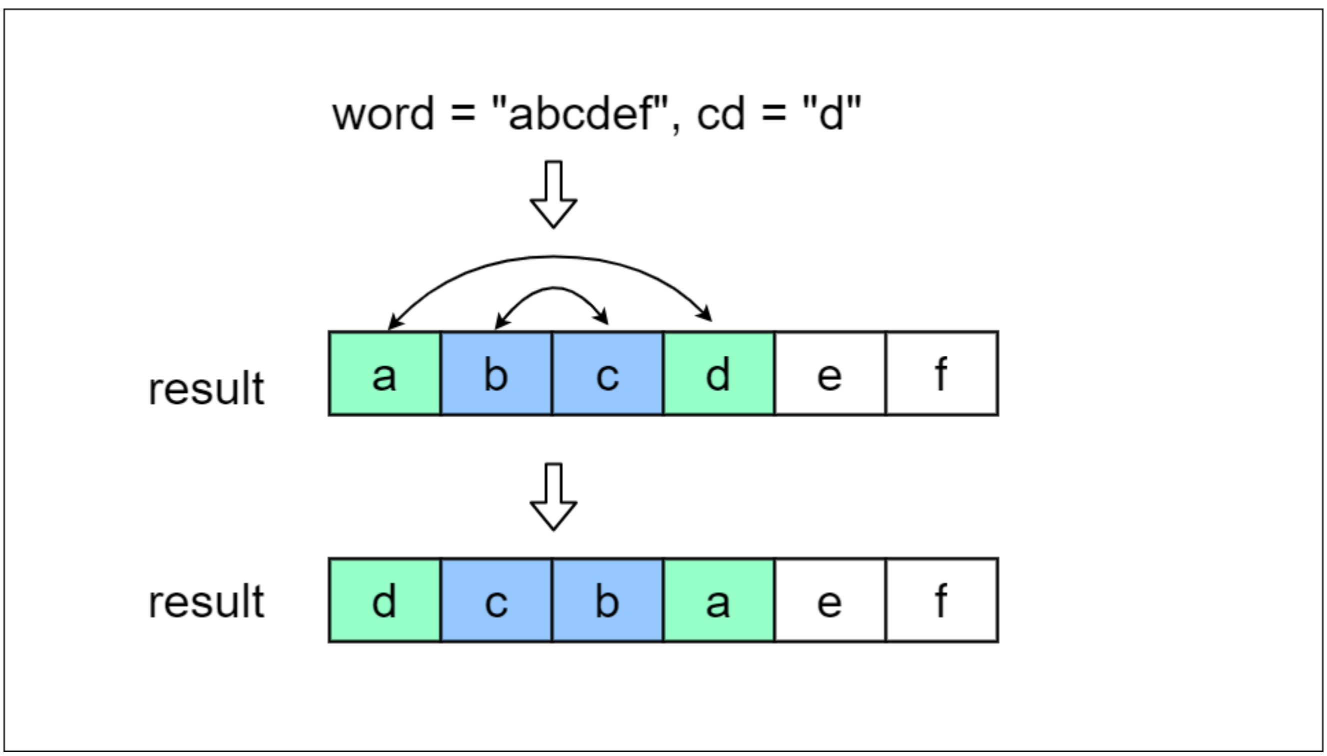
This reversal strategy can be performed in place, as demonstrated in this problem: [344 Reverse String](). However, this problem differs from the one at hand since the input is provided as a character array instead of a string.

We can utilize this strategy by initially adding the characters from `word` to `result` , where `result` is a list or array of characters.

We iterate through `result` using `right` until it reaches the first occurrence of `ch` . If `ch` is not in `word` , we return `word` .

Subsequently, we traverse through the prefix of `result` with two pointers, `left` pointing to the beginning of the prefix and `right` pointing to the end of the prefix, until they meet in the middle. During each iteration, we swap the values at the indices `left` and `right` , then progress each pointer one step towards each other.

Finally, we return `result` and convert it to a string if necessary.



Algorithm

1. Initialize a string or list `result` for building the string with the reversed prefix.

2. Initialize a pointer `left` to `0`.

3. Use a `for` loop to iterate through `result`, using the iterator `right`:

   - If `result[right]` is equal to `ch`:
     - While `left` is less than `right`, swap the characters of `result` at indices `left` and `right`, then increment `left` and decrement `right`.
   - After the loop, return `result` and convert it to a string if needed.

4. If the loop completes without finding `ch`, return the original `word`.

Implementation

Complexity Analysis

Let n be the length of `word`.

- Time complexity: $O(n)$

  Copying `word` to `result` takes $O(n)$.

  In the worst case scenario, when `ch` is located at the last index of `word`, we traverse `result` once to find `ch`, and then we swap $\frac{n}{2}$ elements.

  Therefore, the time complexity remains $O(n)$.

- Space complexity: $O(n)$ (Python and Java) or $O(1)$ (C++)

  We use the `result` array of size n to store and reverse the letters from `word`.

  > **Note:** The C++ version uses $O(1)$ space because the characters are reversed in place instead of using an auxiliary data structure. It is recommended to check with your interviewer before modifying the input, as it might lead to issues in certain scenarios.