# Question

Given an input string s, reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

Return *a string of the words in reverse order concatenated by a single space.*

**Note** that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

**Example 1:**

**Input:** s = "the sky is blue"

**Output:** "blue is sky the"

**Example 2:**

**Input:** s = " hello world "

**Output:** "world hello"

**Explanation:** Your reversed string should not contain leading or trailing spaces.

**Example 3:**

**Input:** s = "a good example"

**Output:** "example good a"

**Explanation:** You need to reduce multiple spaces between two words to a single space in the reversed string.

**Example 4:**

**Input:** s = " Bob Loves Alice "

**Output:** "Alice Loves Bob"

**Example 5:**

**Input:** s = "Alice does not even like bob"

**Output:** "bob like even not does Alice"

**Constraints:**

* 1 <= s.length <= 104
* s contains English letters (upper-case and lower-case), digits, and spaces ' '.
* There is **at least one** word in s.

**Follow up:**

* Could you solve it **in-place** with O(1) extra space?

# Solution

#### **Overview**

Different interviewers would probably expect different approaches for this problem. The holy war question is to use or not use built-in methods. As you may notice, most of design problems on Leetcode are voted down because of two main reasons:

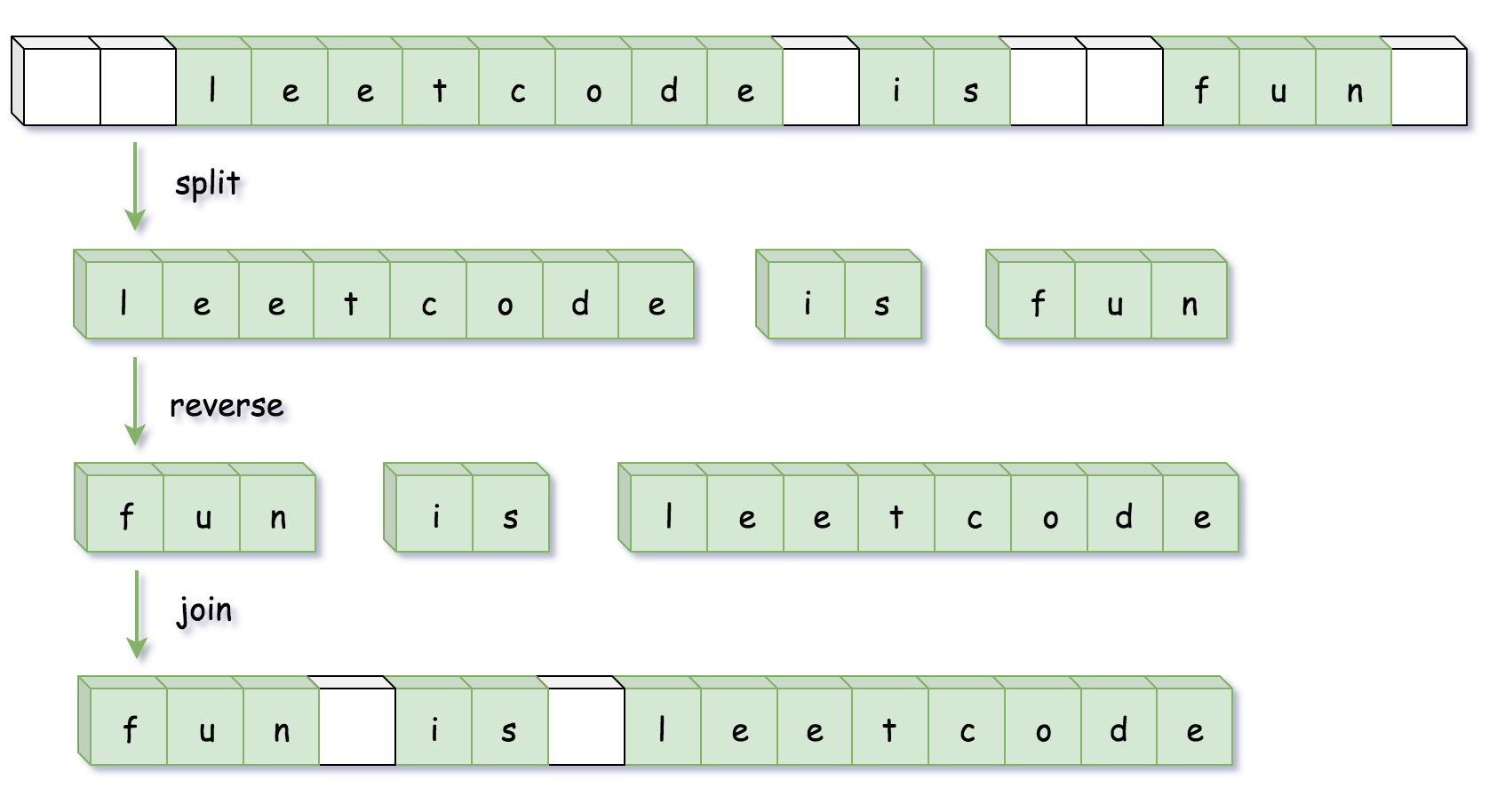
1. There was no approach with built-in methods / data structures in the article.
2. One of the approaches in the article did contain built-in methods / data structures.

Seems like the community has no common opinion yet, and in practice that means an unpredictable interview experience for some sort of problems.

Here we consider three different solutions of linear time and space complexity:

1. Use built-in split and reverse. Benefits: in-place in Python (in-place, but linear space complexity!) and the simplest one to write.
2. The most straightforward one. Trim the whitespaces, reverse the whole string and then reverse each word.  
   Benefits: could be done in-place for the languages with mutable strings.
3. Two passes approach with a deque. Move along the string, word by word, and push each new word in front of the deque. Convert the deque back into string. Benefits: two passes.

#### **Approach 1: Built-in Split + Reverse**



**Implementation**

|  |
| --- |
| class Solution {  public String reverseWords(String s) {  // remove leading spaces  s = s.trim();  // split by multiple spaces  List<String> wordList = Arrays.asList(s.split("\\s+"));  Collections.reverse(wordList);  return String.join(" ", wordList);  }  } |

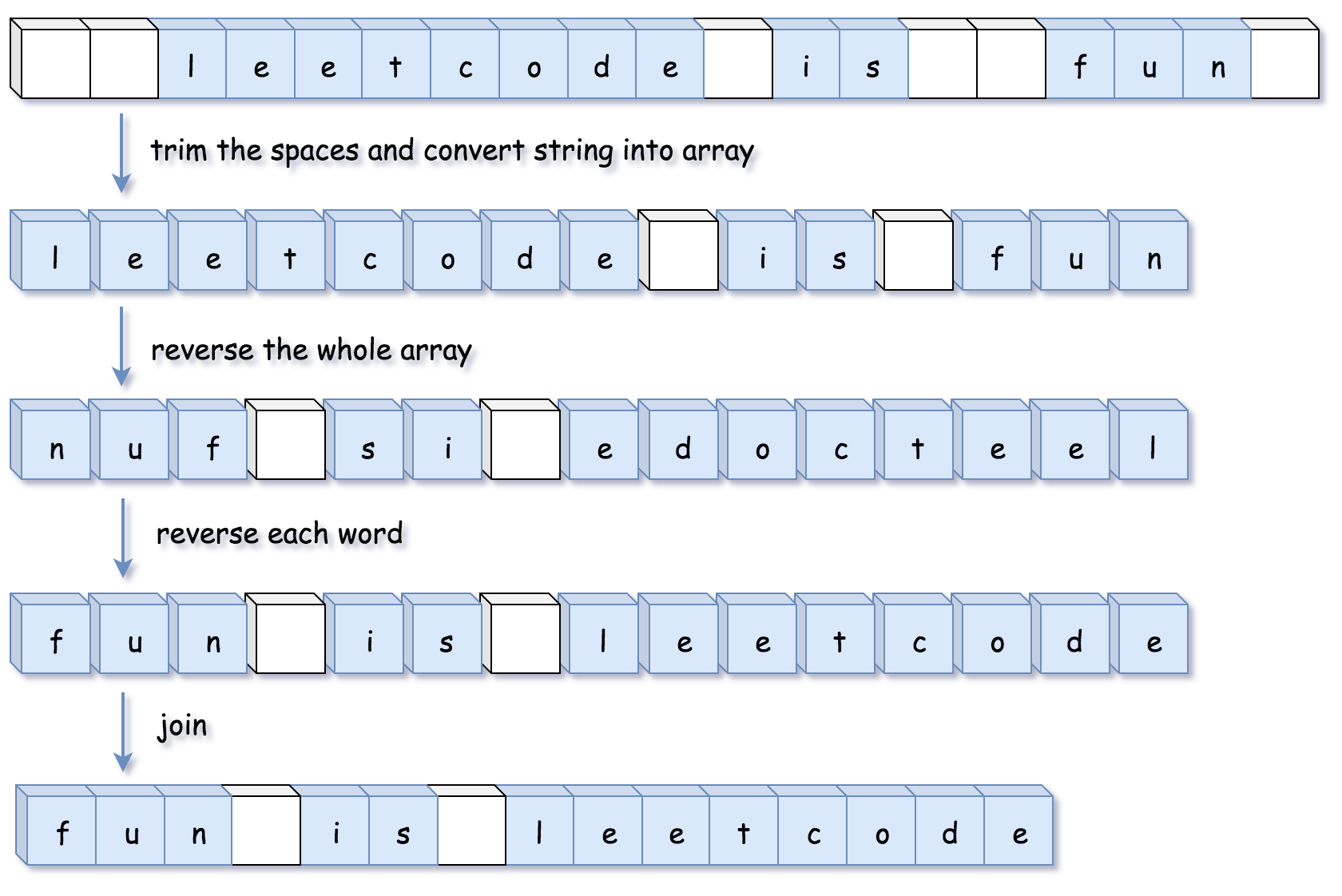
**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N is a number of characters in the input string.
* Space complexity: \mathcal{O}(N)O(*N*), to store the result of split by spaces.

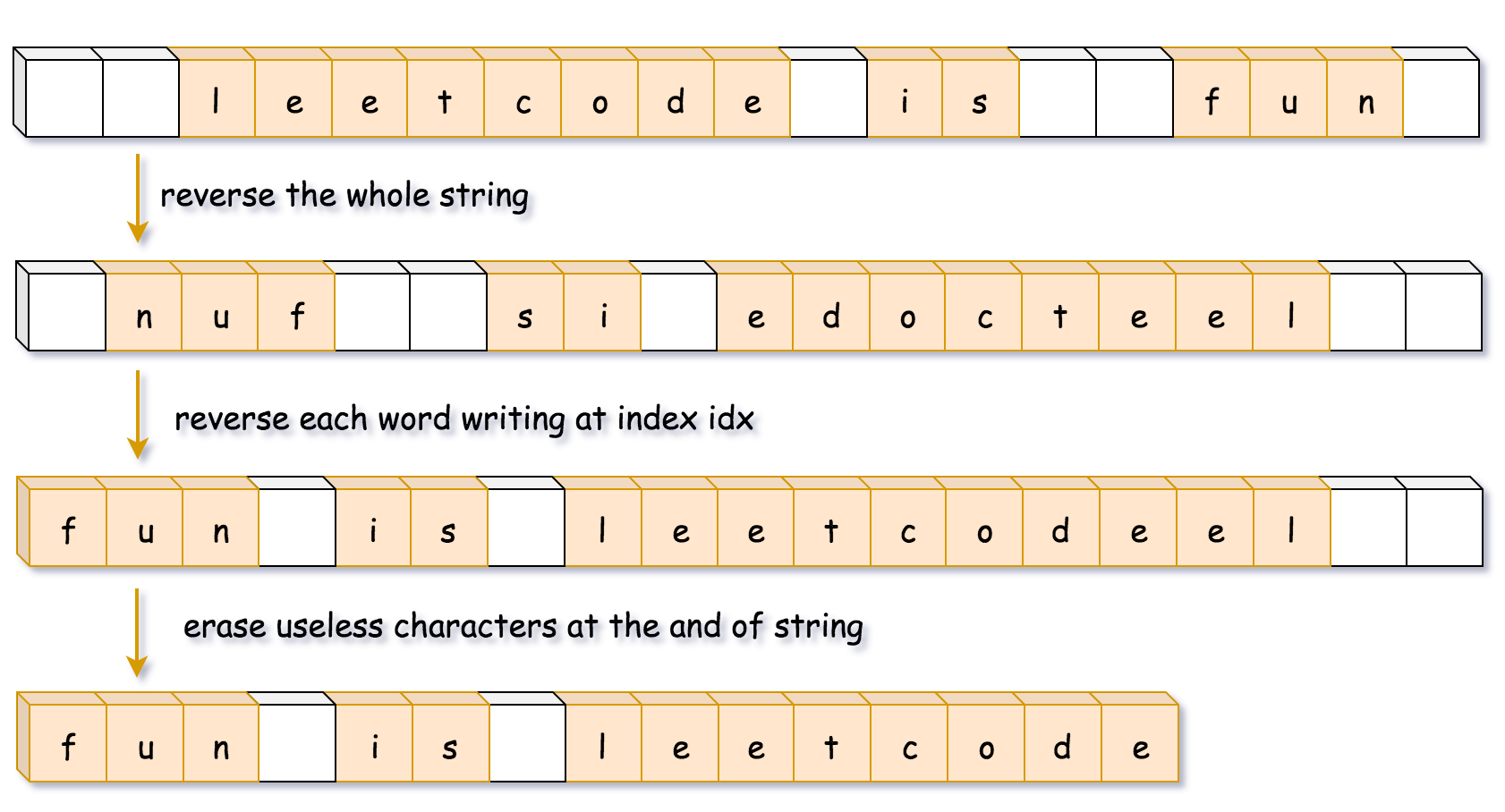
#### **Approach 2: Reverse the Whole String and Then Reverse Each Word**

The implementation of this approach will be different for Java/Python (= immutable strings) and C++ (= mutable strings).

In the case of immutable strings one has first to convert string into mutable data structure, and hence it makes sense to trim all spaces during that conversion.



In the case of mutable strings, there is no need to allocate an additional data structure, one could make all job done in-place. In such a case it makes sense to reverse words and trim spaces at the same time.



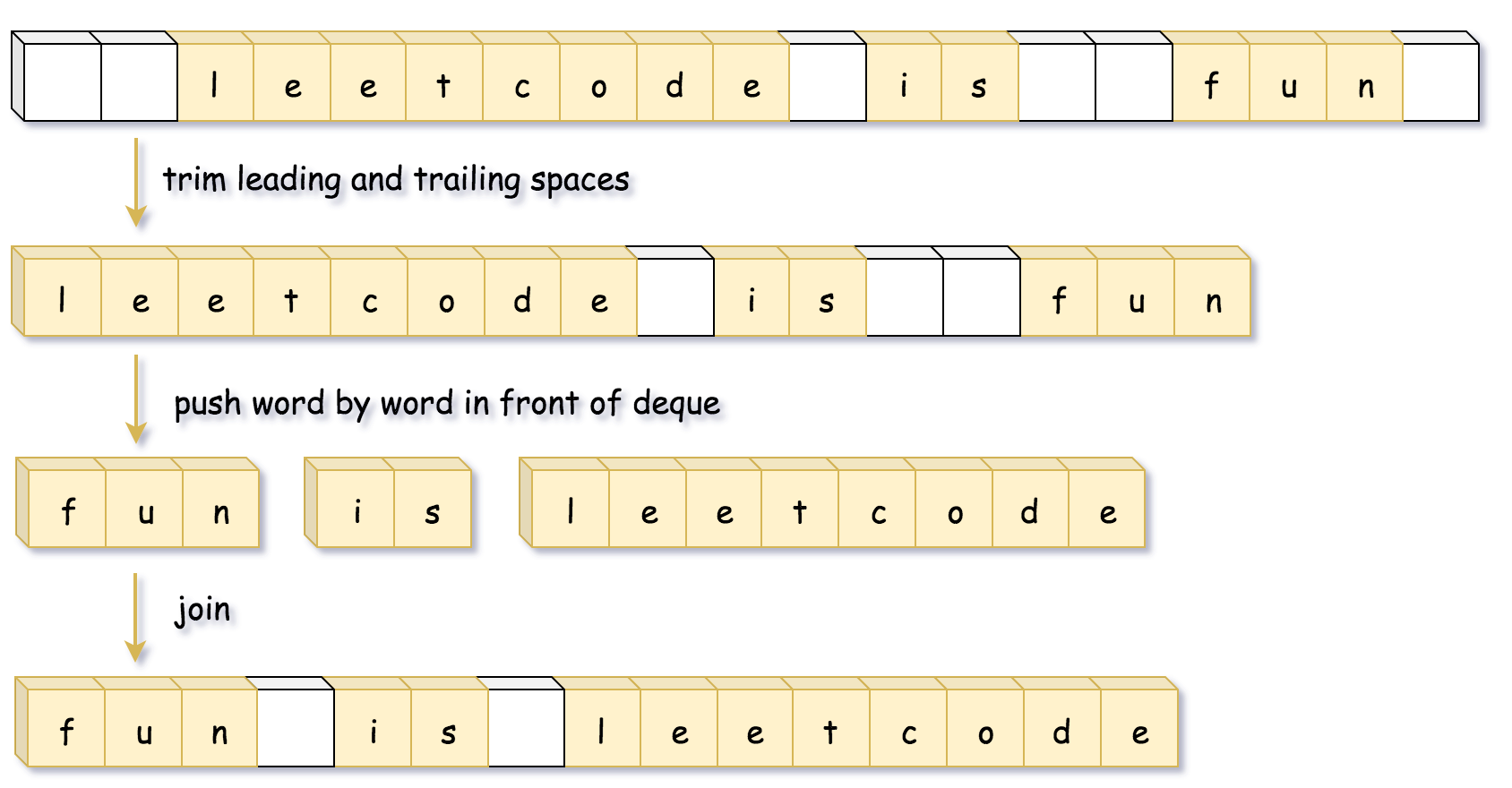
**Implementation**

|  |
| --- |
| class Solution {  public StringBuilder trimSpaces(String s) {  int left = 0, right = s.length() - 1;  // remove leading spaces  while (left <= right && s.charAt(left) == ' ') ++left;  // remove trailing spaces  while (left <= right && s.charAt(right) == ' ') --right;  // reduce multiple spaces to single one  StringBuilder sb = new StringBuilder();  while (left <= right) {  char c = s.charAt(left);  if (c != ' ') sb.append(c);  else if (sb.charAt(sb.length() - 1) != ' ') sb.append(c);  ++left;  }  return sb;  }  public void reverse(StringBuilder sb, int left, int right) {  while (left < right) {  char tmp = sb.charAt(left);  sb.setCharAt(left++, sb.charAt(right));  sb.setCharAt(right--, tmp);  }  }  public void reverseEachWord(StringBuilder sb) {  int n = sb.length();  int start = 0, end = 0;  while (start < n) {  // go to the end of the word  while (end < n && sb.charAt(end) != ' ') ++end;  // reverse the word  reverse(sb, start, end - 1);  // move to the next word  start = end + 1;  ++end;  }  }  public String reverseWords(String s) {  // converst string to string builder  // and trim spaces at the same time  StringBuilder sb = trimSpaces(s);  // reverse the whole string  reverse(sb, 0, sb.length() - 1);  // reverse each word  reverseEachWord(sb);  return sb.toString();  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(N)O(*N*).

#### **Approach 3: Deque of Words**



**Implementation**

|  |
| --- |
| class Solution {  public String reverseWords(String s) {  int left = 0, right = s.length() - 1;  // remove leading spaces  while (left <= right && s.charAt(left) == ' ') ++left;  // remove trailing spaces  while (left <= right && s.charAt(right) == ' ') --right;  Deque<String> d = new ArrayDeque();  StringBuilder word = new StringBuilder();  // push word by word in front of deque  while (left <= right) {  char c = s.charAt(left);  if ((word.length() != 0) && (c == ' ')) {  d.offerFirst(word.toString());  word.setLength(0);  } else if (c != ' ') {  word.append(c);  }  ++left;  }  d.offerFirst(word.toString());  return String.join(" ", d);  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(N)O(*N*).