Solution: Reorganize String

Let's solve the Reorganize String problem using the Top K Elements pattern.

We'll cover the following

Statement

Given a string, `str`, rearrange it so that any two adjacent characters are not the same. If such a reorganization of the characters is possible, output any possible valid arrangement. Otherwise, return an empty string.

Constraints:

- $1 \leq$ `str.length` ≤ 500
- Input string consists of lowercase English letters.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

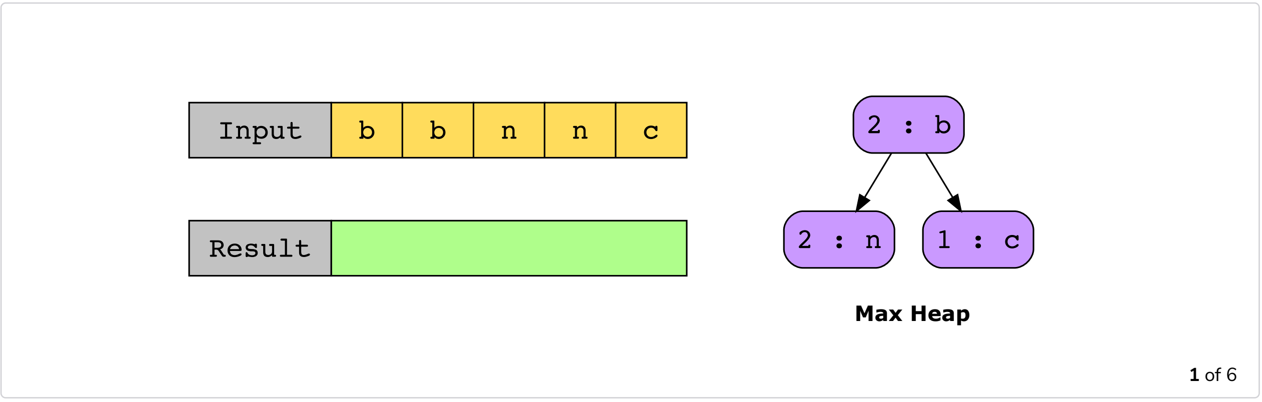
The naive approach is to generate all possible permutations of the given string and check if the generated string is a valid arrangement or not. If any permutation satisfies the condition of a valid reorganization of the string, return that permutation of the input string. If no permutation is a valid reorganization of the string, return an empty string.

Since $O(2^n)$ time is required to generate all possible permutations of the n characters in the input string, the time complexity of this naive approach is $O(2^n)$.

Optimized solution using top k elements

Let's use the top k elements technique to reorganize the input string. This technique will use a max-heap to store characters along with their frequencies so that the character with the highest frequency will always be at the root of the heap. If we build the required order from the most frequent element, followed by the second most frequent element, and keep following this trend, we'll likely find a valid reorganization of the string. If that fails, this means that it's impossible to rearrange the string.

The illustration below shows the whole process:



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

First, we calculate the frequency of the characters in the input string. We store each character as a key in a hash map and set its value to the frequency of its occurrence in the string.

Java

```java
import java.util.*;

class Reorganize {
    public static String reorganizeString(String str) {
        // initializing the hash map
        Map <Character, Integer> charCounter = new HashMap <> ();

        // Calculate the frequency of characters in string and store counts
        // of each character along with the character itself in hash map.
        for (char c: str.toCharArray()) {
            int freq = charCounter.getOrDefault(c, 0) + 1;
            charCounter.put(c, freq);
        }
        return charCounter.toString();
    }

    public static void main(String args[]) {
        String[] inputs = {
            "programming",
            "hello",
            "fofjjb",
            "abbacdde",
            "aba",
            "awesome"
        };
        for (int i = 0; i < inputs.length; i++) {
            System.out.print(i + 1);
```

Reorganize String

After getting all the characters and their respective frequencies, we initialize the heap to provide quick access to the most frequently occurring characters in the string. Since some languages, such as Python, don't have built-in max-heap functionality, we store the frequencies in a heap in such a way that will serve our purpose.

> **Note**: We'll implement the heap using the `PriorityQueue` class.

We first iterate through the hash map and store the negative count of each character and the character itself in a heap. The reason for storing the negative count of each character is that when we pop characters from the heap, the heap will return the character with the maximum frequency.

Java

```java
import java.util.*;

class Reorganize {
    public static String reorganizeString(String str) {
        // initializing the hash map
        Map <Character, Integer> charCounter = new HashMap <> ();

        // Calculate the frequency of characters in string and store counts
        // of each character along with the character itself in hash map.
        for (char c: str.toCharArray()) {
            int freq = charCounter.getOrDefault(c, 0) + 1;
            charCounter.put(c, freq);
        }
        // initializing max heap
        PriorityQueue <Map.Entry<Character, Integer>> maxFreqChars = new PriorityQueue <Map.Entry<Charact
            (item1, item2) -> item2.getValue() - item1.getValue());

        // store all characters with their frequencies to the max heap
        maxFreqChars.addAll(charCounter.entrySet());

        System.out.println(maxFreqChars);
        return maxFreqChars.toString();
    }

    public static void main(String args[]) {
        String[] inputs = {
            "programming",
            "hello"
```

Reorganize String

Now, we take two variables, `previous` and `result`. The `previous` variable stores the previous character that we used so that we don't use that character again. The `result` variable stores the final reorganized string.

The character with the highest frequency will always be at the root of our heap. We keep popping characters from the top of the heap to add them to the result string.

When we add a character to the result string, we won't push this character back onto the heap right away, even if its frequency of occurrence is greater than 0. Instead, we add it back to the heap in the next iteration. The reason is that we want to ensure that the same characters don't appear adjacent to each other in the result string. Therefore, we store the current character along with its frequency of occurrence in `previous` for use in the next iteration.

Java

```java
import java.util.*;

class Reorganize {
    public static String reorganizeString(String str) {
        // initializing the hash map
        Map <Character, Integer> charCounter = new HashMap <> ();

        // Calculate the frequency of characters in string and store counts
        // of each character along with the character itself in hash map.
        for (char c: str.toCharArray()) {
            int freq = charCounter.getOrDefault(c, 0) + 1;
            charCounter.put(c, freq);
        }

        // initializing the max heap
        PriorityQueue <Map.Entry <Character, Integer>> maxFreqChar = new PriorityQueue <Map.Entry <Charac
            (item1, item2) -> item2.getValue() - item1.getValue());

        // store all characters with their frequencies to the max heap
        maxFreqChar.addAll(charCounter.entrySet());

        // initializing variables
        Map.Entry <Character, Integer> previous = null;
        StringBuilder result = new StringBuilder(str.length());
        while (!maxFreqChar.isEmpty() || previous!=null) {

            Map.Entry < Character, Integer > currentEntry = maxFreqChar.poll();
```

Reorganize String

So far, our solution handles almost all the cases, but it's still incomplete. It does not handle the strings for which reorganization is not possible like `aaab`. This string cannot be reorganized.

To handle these types of cases, we add an extra condition to our loop. If our heap is empty and the `previous` variable is non-empty, that means we get to a point where we cannot generate any solution because no valid reordering of the string exists. Therefore, an empty string is returned.

Java

```java
import java.util.*;

class Reorganize {
    public static String reorganizeString(String str) {
        // initializing the hash map
        Map <Character, Integer> charCounter = new HashMap <> ();

        // Calculate the frequency of characters in string and store counts
        // of each character along with the character itself in hash map.
        for (char c: str.toCharArray()) {
            int freq = charCounter.getOrDefault(c, 0) + 1;
            charCounter.put(c, freq);
        }

        // initializing the max heap
        PriorityQueue <Map.Entry <Character, Integer>> maxFreqChar = new PriorityQueue <Map.Entry <Charac
            (item1, item2) -> item2.getValue() - item1.getValue());

        // store all characters with their frequencies to the max heap
        maxFreqChar.addAll(charCounter.entrySet());

        // initializing variables
        Map.Entry <Character, Integer> previous = null;
        StringBuilder result = new StringBuilder(str.length());
        while (!maxFreqChar.isEmpty() || previous!=null) {

            if (previous != null && maxFreqChar.isEmpty())
                return "";
```

Reorganize String

Just the code

Here's the complete solution to this problem:

Java

```java
import java.util.*;

class Reorganize {
    public static String reorganizeString(String str) {

        Map <Character, Integer> charCounter = new HashMap <> ();
        for (char c: str.toCharArray()) {
            int freq = charCounter.getOrDefault(c, 0) + 1;
            charCounter.put(c, freq);
        }
        PriorityQueue <Map.Entry <Character, Integer>> maxFreqChar = new PriorityQueue <Map.Entry <Charac
            (item1, item2) -> item2.getValue() - item1.getValue());

        maxFreqChar.addAll(charCounter.entrySet());

        Map.Entry <Character, Integer> previous = null;
        StringBuilder result = new StringBuilder(str.length());
        while (!maxFreqChar.isEmpty() || previous!=null) {

            if (previous != null && maxFreqChar.isEmpty())
                return "";

            Map.Entry <Character, Integer> currentEntry = maxFreqChar.poll();
            int count=currentEntry.getValue()-1;
            result.append(currentEntry.getKey());

            if(previous!=null){
                maxFreqChar.add(previous);
```

Reorganize String

Solution summary

1. Store each character and its frequency in a hash map.
2. Construct a max-heap using the character frequency data so that the most frequently occurring character is at the root of the heap.
3. Iterate over the heap until all the characters have been considered.
   - Pop the most frequently occurring character from the heap and append it to the result string.
   - Decrement the count of the popped character (since we have used one occurrence of it).
   - Push the popped character back onto the heap in the next iteration if the updated frequency is

5. If the heap becomes empty and there is still an element exist to push into the heap, it indicates that reorganization of the string is not possible, return an empty string.

Time complexity

As we iterate through the heap, every popped element may be pushed back onto the heap. This process is repeated until we have considered all the characters in the input string. Therefore, the iteration runs $O(n)$ times, where n is the number of characters in the string. The worst-case time complexity of the push operation is $O(\log(c))$, where c is the number of distinct characters in the string. Therefore, the time complexity is $O(n \log(c))$. Since the upper bound on c is the size of the alphabet, the $\log(c)$ term is effectively a constant, and we may say that the time complexity is $O(n)$.

Space complexity

In our solution, we employed two data structures: a hash map and a heap. The hash map is responsible for storing the frequencies of characters in the input string, while the heap is used in the solution to find the desired string. Both data structures store lowercase alphabets. The maximum capacity of each data structure is 26 — a fixed number. As a result, the space complexity of our solution is $O(1)$.

Back

Reorganize String

Next

K Closest Points to Ori...

✓ Mark as Completed