package com.iimtiaz.day\_10;  
  
import java.util.Arrays;  
  
public class SubString {  
 public static void main(String[] args) {  
 String str = "ADOBECODEBANC", n = "ABC";  
 System.*out*.println(new Solution\_3().minWindow(str, n));  
 }  
}  
  
class Solution\_2 {  
 public String minWindow(String s, String t) {  
  
 int startans = 0, endans = 0;  
 int[] arr1 = new int[128];  
 int[] arr2 = new int[128];  
  
 int start = 0, end = 0, min = Integer.*MAX\_VALUE*;  
  
 Arrays.*fill*(arr1, 0);  
 Arrays.*fill*(arr2, 0);  
  
 for (char i : t.toCharArray()) arr2[i]++;  
  
 for (int i = 0; i < s.length(); i++) {  
 arr1[s.charAt(i)]++;  
 while (end <= i && isvalid(arr1, arr2)) {  
 if (min > i - end + 1) {  
 startans = i + 1;  
 endans = end;  
 min = i - end + 1;  
 }  
 arr1[s.charAt(end)]--;  
 end++;  
 }  
 }  
 return s.substring(endans, startans);  
 }  
  
 public boolean isvalid(int[] arr1, int[] arr2) {  
 for (int i = 'A'; i < 128; i++) if (arr1[i] < arr2[i]) return false;  
 return true;  
 }  
  
}

package com.iimtiaz.day\_10;  
  
public class SubString {  
 public static void main(String[] args) {  
 String str = "ADOBECODEBANC", n = "ABC";  
 System.*out*.println(new Solution\_3().minWindow(str, n));  
 }  
}  
  
*/\*\*  
 \* Time Complexity: O(n)  
 \*  
 \* O(m): Initializing the map array and processing characters of t.  
 \* O(n): Iterating through s using the two pointers start and end.  
 \* O(1): Operations within the loops (character comparisons, counter updates, array manipulations) are constant time.  
 \* Space Complexity: O(1)  
 \*  
 \* Uses a fixed-size array map (128 elements) and a few integer variables, irrespective of input string lengths.  
 \* No additional data structures with size dependent on input are used.  
 \*/*// \*\*Key Techniques:\*\*  
// - \*\*Two Pointers:\*\* The algorithm uses two pointers, `start` and `end`, to efficiently slide through `s` and  
// maintain a window that potentially contains all characters of `t`.  
// - \*\*Character Frequency Map:\*\* A frequency map `map` is used to track the remaining count of each character  
// from `t` needed within the window.  
//  
// \*\*Algorithm:\*\*  
// 1. \*\*Initialization:\*\*  
// - Create the frequency map `map` based on characters in `t`.  
// - Initialize `start`, `end`, `minStart`, `minLen`, and `counter` variables.  
//  
// 2. \*\*Expanding Window:\*\*  
// - Move `end` to expand the window.  
// - Decrement `map[c1]` (where `c1` is the character at `end`) and `counter` if `c1` is in `t`.  
//  
// 3. \*\*Shrinking Window:\*\*  
// - If `counter` is 0 (all characters of `t` are in the window):  
// - Check if the current window is smaller than the minimum found so far. If so, update `minStart` and `minLen`.  
// - Move `start` forward to shrink the window.  
// - Increment `map[c2]` (where `c2` is the character at `start`) and `counter` if `c2` is in `t`.  
//  
// 4. \*\*Return the Result:\*\*  
// - Return either an empty string (no window found) or the substring representing the minimum window.  
   
   
class Solution\_3 {  
 public String minWindow(String s, String t) {  
 // If s is shorter than t, there's no possible window, so return an empty string.  
 if (s.length() < t.length()) return "";  
  
 // Creates an array map to store character frequencies for t.  
 int[] map = new int[128];  
  
 // Iterates through t's characters and increments their counts in map.  
 for (char c : t.toCharArray()) map[c]++;  
  
 int start = 0, end = 0, minStart = 0, minLen = Integer.*MAX\_VALUE*, counter = t.length();  
  
 // Iterates until the end of s.  
 while (end < s.length()) {  
  
 // Gets the character at the current end index.  
 final char c1 = s.charAt(end);  
  
 // If c1 is in t, decrement counter (needed character found).  
 if (map[c1] > 0) counter--;  
  
 // Decrement the count of c1 in map.  
 map[c1]--;  
  
 // Move the end pointer to the next character.  
 end++;  
  
 // Runs if all characters from t are found in the current window.  
 while (counter == 0) {  
  
 // If the current window is shorter than the shortest found so far, update minStart and minLen.  
 if (minLen > end - start) {  
 minLen = end - start;  
 minStart = start;  
 }  
  
 // Gets the character at the current start index.  
 final char c2 = s.charAt(start);  
  
 // Increment the count of c2 in map (as we're potentially removing it from the window).  
 map[c2]++;  
  
 // If c2 was a character from t, increment counter (indicating a character from t is leaving the window).  
 if (map[c2] > 0) counter++;  
  
 // Move the start pointer to the next character, shrinking the window.  
 start++;  
 }  
 }  
 // This line checks whether the minLen variable still holds its initial value of Integer.MAX\_VALUE. This value  
 // was assigned at the beginning to represent the largest possible window size.  
 // This line acts as a conditional statement, returning either an empty string if no window was found or the  
 // actual minimum window substring if one was discovered within s.  
 return minLen == Integer.*MAX\_VALUE* ? "" : s.substring(minStart, minStart + minLen);  
 }  
}  
  
// \*\*Key Techniques:\*\*  
// - \*\*Two Pointers:\*\* The algorithm uses two pointers, `start` and `end`, to efficiently slide through `s` and  
// maintain a window that potentially contains all characters of `t`.  
// - \*\*Character Frequency Map:\*\* A frequency map `map` is used to track the remaining count of each character  
// from `t` needed within the window.  
//  
// \*\*Algorithm:\*\*  
// 1. \*\*Initialization:\*\*  
// - Create the frequency map `map` based on characters in `t`.  
// - Initialize `start`, `end`, `minStart`, `minLen`, and `counter` variables.  
//  
// 2. \*\*Expanding Window:\*\*  
// - Move `end` to expand the window.  
// - Decrement `map[c1]` (where `c1` is the character at `end`) and `counter` if `c1` is in `t`.  
//  
// 3. \*\*Shrinking Window:\*\*  
// - If `counter` is 0 (all characters of `t` are in the window):  
// - Check if the current window is smaller than the minimum found so far. If so, update `minStart` and `minLen`.  
// - Move `start` forward to shrink the window.  
// - Increment `map[c2]` (where `c2` is the character at `start`) and `counter` if `c2` is in `t`.  
//  
// 4. \*\*Return the Result:\*\*  
// - Return either an empty string (no window found) or the substring representing the minimum window.