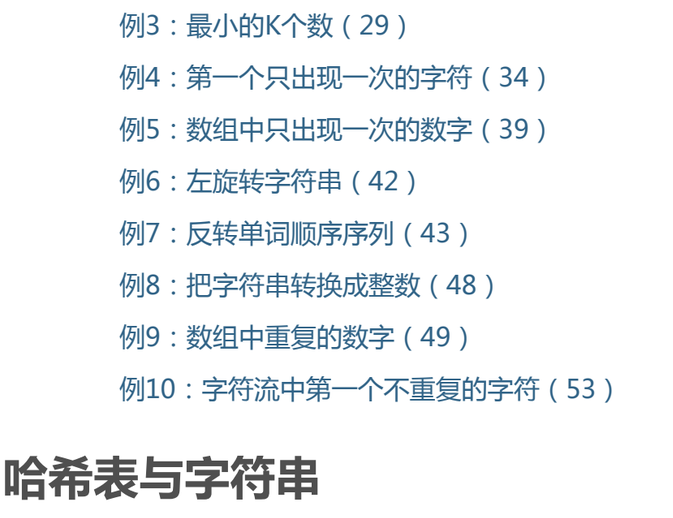
第七章 哈希表与字符串

文章目录

哈希表与字符串

哈希表基础知识

哈希表（Hash table，也叫散列表），是根据关键字（key）直接进行访问的数据结构，它通过把关键值映射到表中一个位置（数组下标）来直接访问，以加快查找关键值的速度。这个映射函数叫做哈希（散列）函数，存放记录的数组叫做哈希（散列）表。
给定表（M），存在函数f（key），对任意的关键值key，代入函数后若能得到包含该关键字的表中地址，称表M为哈希表，函数f（key）为哈希函数。



- 最简单的哈希——字符哈希

```
int main(){
    // ASCII码，从0~127，故使用数组下标做映射，最大范围至128
    int char_map[128] = {0};
    string str = "abcdefgaaxxy";
    // 统计字符串中，各个字符的数量。若char_map['a']++;即char_map[97]++;
    for(int i=0;i<str.length();i++)char_map[str[i]]++;
    for(int i=0;i<128;i++)
        if(char_map[i]>0)
            printf("[%c][%d] : %d\n", i, i, char_map[i]);
    return 0;
}
```

- Hash表的实现

问题引入1：任意元素的映射
解决：利用哈希函数
将关键值（大整数、字符串、浮点数等）转化为整数再对表长取余，从而关键字值被转换为哈希表的表长范围内的整数。

问题引入2：不同整数或字符串，由于哈希函数的选择，映射到同一个下标，发生冲突
解决：拉链法解决冲突
将所有哈希函数结果相同的结点连接在同一个单链表中。
若选定的哈希表长度为m，则可将哈希表定义为一个长度为m的指针数组t[0...m-1]，指针数组中的每个指针指向哈希函数结果相同的单链表。
插入value:
将元素value插入哈希表，若元素value的哈希函数值为hash_key，将value对应的结点以头插法的方式插入到以t[hash_key]为头指针的单链表中。
查找value:
若元素value的哈希函数值为hash_key，遍历以t[hash_key]为头指针的单链表，查找链表各个结点的值域是否为value。

```
    public class HashNode{
        // Hash表数据结构为单链表构成的数组
        int val;
        HashNode next;
        HashNode(int x){
```

```
            val = x;
            next = null;
        }
    }

    int hash_func(int key,int table_len) {
        // 整数哈希函数：直接对表长取余。获得表长范围内的正整数。
        return key % table_len;
    }

    void insert(HashNode[] hash_table, HashNode node, int table_len) {
        // 向哈希表中插入元素，采用头插法
        int hash_key = hash_func(node.val, table_len);
        node.next = hash_table[hash_key].next;
        hash_table[hash_key].next = node;
    }

    boolean search(HashNode[] hash_table, int value, int table_len) {
        // 从哈希表中查找元素，遍历哈希值对应的单链表
        int hash_key = hash_func(value, table_len);
        HashNode head = hash_table[hash_key].next;
        while(head != null) {
            if(head.val == value)return true;
            head = head.next;
        }
        return false;
    }
```

HashMap 基本使用

1. 基本使用

（1） 插入键值对数据

public V put(K key, V value)

（2）根据键值获取键值对值数据

public V get(Object key)

（3）获取Map中键值对的个数

public int size()

（4）判断Map集合中是否包含键为key的键值对

public boolean containsKey(Object key)

（5）判断Map集合中是否包含值为value的键值对

boolean containsValue(Object value)

（6）判断Map集合中是否没有任何键值对

public boolean isEmpty()

（7）清空Map集合中所有的键值对

public void clear()

（8）根据键值删除Map中键值对

public V remove(Object key)

2. 遍历

（1）将Map中所有的键装到Set集合中返回

```
//public Set keySet();
Set set=map. keySet()
```

（2）返回集合中所有的value的值的集合

```
// public Collection values();
Collection c=map.values()
```

（3）将每个键值对封装到一个个Entry对象中,再把所有Entry的对象封装到Set集合中返回

```
// public Set<Map.Entry<K,V>> entrtSet();
Set<Map.Entry<K,V>> entrys=map.entrySet()
```

```
        HashMap<String,Integer> map =new HashMap<>();
        map.put("a", 1);
        map.put("b", 2);
        map.put("c", 3);
        map.put("d", 4);

        // 遍历方法1—将Map中所有的键装到Set集合中返回
         Iterator<String> iterator = map.keySet().iterator();
         while (iterator.hasNext()) {
          Object key = iterator.next();
          System.out.println("map.get(key) is :"+map.get(key));
         }

         // 遍历方法2—将每个键值对封装到一个个Entry对象中，再把所有Entry对象封装到Set集合中返回
         Set<Map.Entry<String, Integer>> set=map.entrySet();
         Iterator<Map.Entry<String, Integer>> it=set.iterator();
         while(it.hasNext()){
          //System.out.println(list.get(0) );
             Map.Entry<String, Integer> e=it.next();
          System.out.println(e.getKey()+":"+e.getValue());
         }
```

leetcoe

例1：最长回文串（409）

题目描述

给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。
在构造过程中，请注意区分大小写。比如 “Aa” 不能当做一个回文字符串。
注意:假设字符串的长度不会超过 1010。
示例 1:

```
输入:
"abccccdd"

输出:
7

解释:
我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。
```

解题思路

使用字符串s 中的字符，任意组合，生成新的字符串，若生成的字符串为回文字符串，需要除了中心字符，其余字符只要头部出现，尾部就要对应出现

1. 利用字符哈希方法，统计字符串中所有的字符数量；
2. 设置最长回文串偶数字符长度为max_length = 0;
3. 设置是否有中心标记flag = 0;
4. 遍历每一个字符，字符数为count，若count为偶数，max_length += count;若count为奇数，max_length+=count-1,flag=1;
5. 最终最长回文子串长度：max_length+flag

程序代码

```
    // 解法1：（采用HashMap）
    public int longestPalindrome(String s) {
        // 定义一个哈希表，哈希表的键值为字符串中的字母，值为对应字母出现的次数。
        int length = 0;
        boolean isAddOne = false;
        // 定义一个HashMap。键为字符串的字母，值为字母出现的次数
        HashMap<String,Integer> m = new HashMap<String,Integer>();
        for(int i=0;i<s.length();i++) {
                String c = s.charAt(i)+"";
                if(m.get(c) == null)m.put(c, 0);
                m.put(c, m.get(c)+1);
        }
        // 遍历HashMap，最长回文串中对于每个字母对应的会问串长度分两种情况讨论：
        // (1) 若该字母出现次数为偶数，则可作为一个回文字符串
        // (2) 若该字母出现次数为奇数，则如果位于中间则可作为一个回文字符串，如果位于两边则-1作为一个回文字符串
        Iterator<String> iterator = m.keySet().iterator();
        while (iterator.hasNext()) {
          String key = iterator.next();
          Integer value = m.get(key);
          if(value%2==0)length += value;
          else {length += value-1;isAddOne = true;}
        }
        if(isAddOne)length++;
        return length;
    }

    // 解法2：采用自定义HashMap
    public int longestPalindrome2(String s) {
        // 定义一个长度为128哈希表（包含大小写字符），哈希表的键值为字符串中的字母，值为对应字母出现的次数。
        int max_length = 0;
        boolean flag = false; // 若存在单数次数字母，则可作为回文串中心字符
        int[] hash_table = new int[128];
        for(int i=0;i<128;i++)hash_table[i] = 0; // 初始化，每个ASCII码字符出现个数为0
        for(int i=0;i<s.length();i++)hash_table[s.charAt(i)]++; // 记录字符串中每个字符出现次数
        for(int i=0;i<128;i++) {
                int value = hash_table[i];
                if(value>0) {
                    // 遍历每个字符的出现次数，如果出现次数为偶数，则作为最长回文串的一部分直接加入;
                    // 若出现次数为奇数，偶数部分作为两端，长度为value-1；奇数部分作为中心字符，flag = true;
                    if(value%2 == 0)max_length += value;
                    if(value%2 == 1) {max_length += value -1;flag = true;}
                }
        }
        if(flag)max_length++;
        return max_length;
    }
```

例2：词语模式（290）

题目描述

给定一种规律 pattern 和一个字符串 str ，判断 str 是否遵循相同的规律。
这里的 遵循 指完全匹配，例如， pattern 里的每个字母和字符串 str 中的每个非空单词之间存在着双向连接的对应规律。

```
示例1:

输入: pattern = "abba", str = "dog cat cat dog"
输出: true
示例 2:

输入:pattern = "abba", str = "dog cat cat fish"
输出: false
```

解题思路

匹配：字符串str中的单词与pattern中地字符一一对应（一一遍历，一一比较）

结论：
（1）当拆解出一个单词时，若该单词已出现，则当前单词对应的pattern字符必为该单词曾经对应地pattern字符。
（2）当拆解出一个单词时，若该单词未曾出现，则当前单词对应的pattern字符也必须未曾出现。
（3）单词的个数与pattern字符串中字符数量相同
算法思路：

1. 设置单词（字符串）到pattern字符的映射（哈希）；使用数组used[128]记录pattern字符是否使用。
2. 遍历str，按照空格拆分单词，同时对应的向前移动指向pattern字符的指针，每拆分出一个单词，判断
   如果该单词从未出现在哈希表中：
   如果当前的pattern字符已被使用，返回false
   将单词与当前指向pattern字符做映射；标记当前指向pattern字符已使用
   否则：
   如果当前单词在哈希表中的映射字符不是当前指向的pattern字符，则返回false
3. 若单词个数与pattern字符个数不符，则返回false

程序代码

```
public boolean wordPattern(String pattern, String str) {
        // 匹配：字符串str中的单词与pattern中地字符一一对应（一一遍历，一一比较）
            // 结论：
            //（1）当拆解出一个单词时，若该单词已出现，则当前单词对应的pattern字符必为该单词曾经对应地pattern字符。
            //（2）当拆解出一个单词时，若该单词未曾出现，则当前单词对应的pattern字符也必须未曾出现。
            //（3）单词的个数与pattern字符串中字符数量相同

            // 算法思路：
            // 1.设置单词（字符串）到pattern字符的映射（哈希）；使用数组used[128]记录pattern字符是否使用。
            // 2.遍历str，按照空格拆分单词，同时对应的向前移动指向pattern字符的指针，每拆分出一个单词，判断
            // 如果该单词从未出现在哈希表中：
            //      如果当前的pattern字符已被使用，返回false
            //      将单词与当前指向pattern字符做映射；标记当前指向pattern字符已使用
            // 否则：
            //      如果当前单词在哈希表中的映射字符不是当前指向的pattern字符，则返回false
            // 3.若单词个数与pattern字符个数不符，则返回false

            HashMap<String, Integer> hash_table = new HashMap<String, Integer>(); // 定义一个哈希表，存储单词对应的模式字符
            boolean[] used = new boolean[128]; // 定义used数组，表示各模式字符是否被使用过,初始化均未使用
            for(int i=0; i<128; i++)used[i] = false;

            String[] str_list = str.split(" "); // 将字符串拆分为单词数组
            if(str_list.length != pattern.length())return false;    // 情况1：如果模式串长度与单词数组长度不同，则不匹配
            for(int i=0;i<str_list.length;i++) {
                String cur_str = str_list[i];
                int cur_pattern = pattern.charAt(i);
                if(hash_table.containsKey(cur_str)) {
                    // 若单词出现在哈希表中，则该单词对应的模式字符一定为之前存储在哈希表中的模式字符
                    // 情况2：模式字符串中对应字符 与 哈希表存储模式字符 不同，则不匹配
                    if(hash_table.get(cur_str) != cur_pattern)return false;
                }else {
                    // 若该单词从未出现在哈希表中，说明是一个新单词。该单词对应的模式为一个新字符
                    // 情况3：若未使用过的字符 出现在 used表中(向前遍历pattern已使用)，则不匹配
                    if(used[cur_pattern])return false;
                    // 添加该字符的映射到哈希表，并标记使用
                    hash_table.put(cur_str, cur_pattern);
                    used[cur_pattern] = true;
                }
            }
            return true;
    }
```

例3：同字符词语分组（49）

题目描述
给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

```
输入: ["eat", "tea", "tan", "ate", "nat", "bat"],
输出:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

说明：
所有输入均为小写字母。
不考虑答案输出的顺序。
解题思路
问题：如何建立哈希表，怎样设计哈希表的key与value，可将各个字符数相同的字符串映射到一起？
设计：哈希表以内部进行排序的各个单词为key，以字符串向量（列表）为value，存储各个字符asc码相同的字符串。
算法思路：
设置字符串到字符串向量的哈希表anagram，遍历字符串向量strs中的单词strs[i]:

1. 设置临时变量str = strs[i]，对str进行排序。
2. 若str未出现在anagram中，设置str到一个空字符串向量的映射。
3. 将strs[i]添加至字符串向量anagram[str]中。
4. 遍历哈希表anagram，将全部key对应的value push到最终结果中。

程序代码

```
    public List<List<String>> groupAnagrams(String[] strs) {
        // 定义一个哈希表，存储(按各个字符ASCII排序后的单词,该单词在原字符串数组中下标数组）键值对
        // 遍历该哈希表，将相同字符存储的位置对应的单词进行分组
            List<List<String>> result = new ArrayList<>();
```

```
        HashMap<String,ArrayList> hash_table = new HashMap<String,ArrayList>();
        for(int i=0;i<strs.length;i++) {
            // 对单词按ASCII码排序，可以将字母异位词分为一组
            String words = sortByASC(strs[i]);
            // 对于每个字母异位词，记录他们在原字符串数组中的位置
            // 对于每个字母异位词，存储在哈希表中，键为按ASCII码排序的单词，值为原单词所构成的列表（字符串向量）
            ArrayList position_list = new ArrayList<>();
            if(hash_table.containsKey(words)) {
                position_list = hash_table.get(words);
                position_list.add(strs[i]);
                hash_table.replace(words, position_list);
            }else {
                position_list.add(strs[i]);
                hash_table.put(words, position_list);
            }
        }
        // 遍历哈希表，根据每个字母异位词的位置数组找到它们对应的单词并进行分组
        // 将分组后的结果存储
        Iterator<String> iterator = hash_table.keySet().iterator();
        while (iterator.hasNext()) {
                ArrayList result_item = new ArrayList<>();
                String key = iterator.next();
                ArrayList<Integer> value = hash_table.get(key);
                for(int i=0;i<value.size();i++) {
                    result_item.add(value.get(i));
                }
                result.add(result_item);
        }
        // 输出结果
            return result;
    }

    public String sortByASC(String s) {
        // 将String 字符串按ASCII码顺序排序
        // 这里利用哈希表进行排序
            int[] hash_table = new int[128];
            String result = "";
            for(int i=0;i<128;i++)
                hash_table[i] = 0;
            for(int i=0;i<s.length();i++) {
                hash_table[s.charAt(i)]++;
            }
            for(int i=0;i<128 ;i++){
                if(hash_table[i]>0)
                for(int j=0;j<hash_table[i];j++){
                    char char_element = (char)i;
                    result += char_element;
                }
            }
            return result;
    }
```

例4：无重复字符的最长子串（3）

题目描述

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

```
示例 1:

输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

解题思路

（优化的）滑动窗口Sliding Window——字符串匹配问题

在暴力法中，我们会反复检查一个子字符串是否含有有重复的字符，但这是没有必要的。如果从索引 i到 j - 1之间的子字符串 s{ij}已经被检查为没有重复字符。我们只需要检查 s[j]对应的字符是否已经存在于子字符串 s{ij}中。

要检查一个字符是否已经在子字符串中，我们可以检查整个子字符串，这将产生一个复杂度为 O(n^2)的算法，但我们可以做得更好。

通过使用 HashSet 作为滑动窗口，我们可以用 O(1)的时间来完成对字符是否在当前的子字符串中的检查。

滑动窗口是数组/字符串问题中常用的抽象概念。 窗口通常是在数组/字符串中由开始和结束索引定义的一系列元素的集合，即 [i, j)（左闭，右开）。而滑动窗口是可以将两个边界向某一方向“滑动”的窗口。例如，我们将 [i, j) 向右滑动 11 个元素，则它将变为 [i+1, j+1)（左闭，右开）。

回到我们的问题，我们使用 HashSet 将字符存储在当前窗口 [i, j)（最初 j = i）中。 然后我们向右侧滑动索引 j，如果它不在 HashSet 中，我们会继续滑动 j。直到 s[j] 已经存在于 HashSet 中。此时，我们找到的没有重复字符的最长子字符串将会以索引 i开头。

我们可以做以下优化：我们可以定义字符到索引的映射，而不是使用集合来判断一个字符是否存在。 当我们找到重复的字符时，我们可以立即跳过该窗口。

也就是说，如果 s[j]s[j] 在 [i, j)范围内有与 j’ 重复的字符，我们不需要逐渐增加 i 。 我们可以直接跳过 [i，j’]范围内的所有元素，并将 i 变为j ’ +1。

算法思路

1. 定义result为当前最长子串的长度
2. 定义begin指向滑动窗口的起始索引，end指向滑动窗口的结束索引
3. 定义哈希表记录当前各字符的位置
4. 指针向后逐个扫描字符串中的字符，并用哈希表记录当前字符的位置。
5. 如果该字符在哈希表中，说明该字符重复遍历，此时起始索引为 当前起始索引 与 上一次遍历该字符的索引 中较大值。
6. 如果该字符不在哈希表中，则在哈希表中记录该字符索引。
7. 整个过程中，begin 与 end 维护一个窗口，窗口中的子串为无重复字符的子串。若该子串长度大于当前最长子串长度result,则更新result

程序代码

```
    public int lengthOfLongestSubstring(String s) {
        // 滑动窗口
        int result = 0; // result为当前最长子串的长度
```

```
        int begin = 0;int end = 0; // begin指向滑动窗口的起始索引，end指向滑动窗口的结束索引
        HashMap<Character, Integer> hash_map = new HashMap<Character, Integer>(); // 利用哈希表记录当前各字符的位置
        // 指针向后逐个扫描字符串中的字符，并用哈希表记录当前字符的位置。
        // 如果该字符在哈希表中，说明该字符重复遍历，此时起始索引为 当前起始索引 与 上一次遍历该字符的索引 中较大值。
        // 如果该字符不在哈希表中，则在哈希表中记录该字符索引。
        // 整个过程中，begin 与 end 维护一个窗口，窗口中的子串为无重复字符的子串。若该子串长度大于当前最长子串长度result,则更新result
        for(end = 0;end < s.length();end++) {
                Character cur_char = s.charAt(end);
                if(hash_map.containsKey(cur_char)) {
                    // 滑动窗口的优化，避免无用的遍历（包含该重复字符），将起始索引指向该字符的索引+1 与 当前索引 中较大值
                    begin = Math.max(hash_map.get(cur_char)+1, begin);
                }
                    hash_map.put(cur_char, end); // hash表更新当前字符的位置
                    result = Math.max(result, end-begin+1); // 比较当前窗口的长度与当前最长子串长度result大小，若大于，则更新result
        }
        return result;
    }
```

例5：重复的DNA序列（187）

题目描述

所有 DNA 由一系列缩写为 A，C，G 和 T 的核苷酸组成，例如：“ACGAATTCCG”。在研究 DNA 时，识别 DNA 中的重复序列有时会对研究非常有帮助。

编写一个函数来查找 DNA 分子中所有出现超多一次的10个字母长的序列（子串）。

示例:

```
输入: s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
输出: ["AAAAACCCCC", "CCCCCAAAAA"]
```

解题思路

枚举DNA字符串中所有长度为10的子串。将其插入到哈希Map中，并记录子串数量；遍历HashMap,将所有出现超过1次的子串存储到结果中。算法复杂度O（n）

滑动窗口+双Set

1. 定义一个set（不包含重复元素的容器）作为结果容器result，存放所有出现超过一次且长度为10的序列
2. 定义一个set作为哈希数组hash_map，存放所遍历过的长度为10的字母串
3. 滑动窗口为长度为10的字母子串。
4. 每次向后遍历一个位置，若此时滑动窗口中的字母串已存在hash_map，则说明出现超过1次，则放入结果数组
5. 将该滑动窗口字母串加入hash_map中存储

程序代码

```
    public List<String> findRepeatedDnaSequences(String s) {
        // 滑动窗口+双Set
        // 定义一个set（不包含重复元素的容器）作为结果容器result，存放所有出现超过一次且长度为10的序列
        // 定义一个set作为哈希数组hash_map，存放所遍历过的长度为10的字母串
        // 滑动窗口为长度为10的字母子串。
        // 每次向后遍历一个位置，若此时滑动窗口中的字母串已存在hash_map，则说明出现超过1次，则放入结果数组
        // 将该滑动窗口字母串加入hash_map中存储
        Set<String> result = new HashSet<String>(); // 利用Set存放结果序列，防止重复
        Set<String> hash_map = new HashSet<String>(); // 利用Set记录已遍历的长度为10的字母序列
        int begin = 0;int end = 10; // begin指向滑动窗口的起始索引，end指向滑动窗口的结束索引
        for(end = 10;end <= s.length();begin++,end++) {
                String cur_s = s.substring(begin, end); // 当前滑动窗口的字母串
                if(hash_map.contains(cur_s)) {  // 出现超过1次字母序列，加入结果序列
                    result.add(cur_s);
                }
                hash_map.add(cur_s); // 将当前字母串加入hash_map，表示遍历过的字母序列
        }
        return new ArrayList<String>(result); // 转换类型
    }
```

例6：最小窗口子串（76）

题目描述

给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。

示例：

```
输入: S = "ADOBECODEBANC", T = "ABC"
输出: "BANC"
```

说明：

如果 S 中不存这样的子串，则返回空字符串 “”。

如果 S 中存在这样的子串，我们保证它是唯一的答案。

解题思路

滑动窗口算法的思路是这样：

1. 我们在字符串 S 中使用双指针中的左右指针技巧，初始化 left = right = 0，把索引闭区间 [left, right] 称为一个「窗口」。
2. 我们先不断地增加 right 指针扩大窗口 [left, right]，直到窗口中的字符串符合要求（包含了 T 中的所有字符）。
3. 此时，我们停止增加 right，转而不断增加 left 指针缩小窗口 [left, right]，直到窗口中的字符串不再符合要求（不包含 T 中的所有字符了）。同时，每次增加 left，我们都要更新一轮结果。

上述过程可以简单地写出如下伪码框架：

```
string s, t;
// 在 s 中寻找 t 的「最小覆盖子串」
int left = 0, right = 0;
string res = s;

while(right < s.size()) {
    window.add(s[right]);
    right++;
    // 如果符合要求，移动 left 缩小窗口
    while (window 符合要求) {
        // 如果这个窗口的子串更短，则更新 res
```

```
        res = minLen(res, window);
        window.remove(s[left]);
        left++;
    }
}
return res;
```

现在就剩下一个比较棘手的问题：如何判断 window 即子串 s[left...right] 是否符合要求，是否包含 t 的所有字符呢？
可以用两个哈希表当作计数器解决。用一个哈希表 needs 记录字符串 t 中包含的字符及出现次数，用另一个哈希表 window 记录当前「窗口」中包含的字符及出现的次数，如果 window 包含所有 needs 中的键，且这些键对应的值都大于等于 needs 中的值，那么就可以知道当前「窗口」符合要求了，可以开始移动 left 指针了。

程序代码

```
    public String minWindow(String s, String t) {
            String result = ""; // 最小子串
            Integer length = Integer.MAX_VALUE; // 最小子串长度
            Integer left = 0;Integer right = 0; // left指向滑动窗口的起始索引，right指向滑动窗口的结束索引
            HashMap<Character,Integer> need = new HashMap<Character,Integer>(); // 定义一个哈希表need，存储窗口中所需要各个字符的数目
            HashMap<Character,Integer> window = new HashMap<Character,Integer>(); // 定义一个哈希表window，存储窗口中已遍历各个字符的数目

            if(s.length() == 0 || t.length() == 0)return result;

            // 初始化need哈希表，存储满足题意的字符及字符个数
            for(int i=0;i<t.length();i++)
                need.put(t.charAt(i), need.containsKey(t.charAt(i))?need.get(t.charAt(i))+1:1);

            while(right < s.length()){
                // 向后遍历字符串S，逐渐增加right指针扩大窗口，直到窗口中的字符串符合要求（包含T中所有字符）
                Character cur_right_char = s.charAt(right);
                right++;
                if(t.contains(cur_right_char+"")) { // 更新已遍历所需要的字母出现次数
                    window.put(cur_right_char, window.containsKey(cur_right_char)?window.get(cur_right_char)+1:1);
                }
                // 当window满足题设条件（包含T中所有字符）
                // 此时停止增加right，转而不断增加left指针缩小窗口，知道窗口中的字符串不再符合要求（不包含T中所有字符）
                if(isSatisfyNeed(need,window)) {
                while(isSatisfyNeed(need,window)) {
                    Character cur_left_char = s.charAt(left);
                    left++;
                    if(t.contains(cur_left_char+"")) {// 更新已遍历所需要的字母出现次数
                        if(window.containsKey(cur_left_char) && window.get(cur_left_char)==1)window.remove(cur_left_char);
                        else if(window.containsKey(cur_left_char))window.put(cur_left_char, window.get(cur_left_char)-1);
                    }
                }
                // 不符合要求时，则left-1是当前符合要求的最短字符串，将该字符串与当前记录结果比较
                // 若比上一次结果长度短，则更新结果
                if(left>0)left--;
                Integer cur_length = right-left+1;
                if(cur_length < length) {
                    length = cur_length;
                    result = s.substring(left,right);
                }
                left++;
            }}
            return result;
    }

    public boolean isSatisfyNeed(HashMap<Character,Integer> need,HashMap<Character,Integer> window) {
        // 若滑动窗口的大小与所需字符大小相同，且所含字符数>=所需字符数，则包含所需字符。满足条件
            if(need.size()!=window.size())return false;
            Iterator<Character> iterator = need.keySet().iterator();
            while (iterator.hasNext()) {
              Character key = iterator.next();
              Integer value = need.get(key);
              if(value > window.get(key))return false;
            }
            return true;
    }
```

剑指offer

例1：替换空格（2）

题目描述
请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

算法思路
遍历字符串，如果遇到空格，则替换为%20

程序代码

```
public String replaceSpace(StringBuffer str) {
                        if(str == null)return null;
                        // 遍历字符串，如果遇到空格，则替换为%20
                        for(int i=0;i<str.length();i++) {
                            if(str.charAt(i) == ' ')
                                str.replace(i, i+1, "%20");
                        }
                        return str.toString();
                }
                // 补充:StringBuffer/StringBuilder 扩展方法
                // append(String s): 将指定字符串追加到此字符序列
                // reverse(): 将此字符换列用反转形式取代
                // delete(int start,int end): 删除此序列的子字符串中的字符
                // insert(int offset, int i): 将 int 参数的字符串表示形式插入此序列中。
                // replace(int start, int end, String str): 使用给定 String 中的字符替换此序列的子字符串中的字符。
                // String长度不可变，StringBuffer/StringBuilder 是长度可变的，其中StringBuffer线程安全，StringBuilder线程不安全
```

例2：数组中出现次数超过一半的数字（28）

题目描述

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

程序代码

```
// 28. 数组中出现次数超过一半的数字
// 数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。
// 例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。
public int MoreThanHalfNum_Solution(int [] array) {
    // 定义HashMap 存储每个数字在数组中出现的次数
        // 每次遍历一个数字都能获得他此时出现的次数，若次数超过数组长度的一半，则返回结果
        if(array == null || array.length == 0)return 0;  // 数据为空，不存在
        if(array.length == 1)return array[0];           // 只有一个数据时返回该数据
        Map<Integer,Integer> countHash = new HashMap<Integer,Integer>();

        for(int i=0;i<array.length;i++) {
            int key = array[i];
            if(countHash.containsKey(key)) {
                int value = countHash.get(key) + 1;
                if(value > array.length/2)return key;
                countHash.replace(key, value);
            }
            else countHash.put(key, 1);
        }
        return 0;
}
```

例3：最小的K个数（29）

题目描述

输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4,。

程序代码

```
// 29.最小的K个数
// 输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4,。
public ArrayList<Integer> GetLeastNumbers_Solution(int [] input, int k) {
        // Arrays.sort(int[]) 使用双轴快速排序算法,时间复杂度为0(logn)
        // Collections.sort(List) 是一种优化过的合并排序算法,时间复杂度是O(n)
        ArrayList<Integer> result = new ArrayList<Integer>();
        if(k==0 || k > input.length)return result;
        Arrays.sort(input);
        for(int i=0;i<k;i++)result.add(input[i]);

        return result;
}
```

例4：第一个只出现一次的字符（34）

题目描述

在一个字符串(0<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置, 如果没有则返回 -1（需要区分大小写）.

程序代码

```
    // 34.第一个只出现一次的字符
// 在一个字符串(0<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置,
// 如果没有则返回 -1（需要区分大小写）.
public int FirstNotRepeatingChar(String str) {
    // 定义一个HashMap 存储每个字符出现的次数
    // 第一次遍历记录字符的出现次数
    // 第二次遍历，若字符出现次数为1则返回该字符的位置
        Map<Character,Integer> charCount = new HashMap<Character,Integer>();    // 存储每个字符出现次数
        for(int i=0;i<str.length();i++) {
            Character key = str.charAt(i);
            if(charCount.containsKey(key))
                charCount.replace(key, charCount.get(key)+1);
            else charCount.put(key, 1);
        }
        for(int i=0;i<str.length();i++) {
            Character key = str.charAt(i);
            if(charCount.containsKey(key) && charCount.get(key) == 1)
                return i;
        }
        return -1;
}
```

例5：数组中只出现一次的数字（39）

题目描述

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

程序代码

```
// 39. 数组中出现一次的数字
// 一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。
// num1,num2分别为长度为1的数组。传出参数
// 将num1[0],num2[0]设置为返回结果
public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {
    // 1. 定义一个HashSet，存储各个数字
        // 2. 若重复访问，则删除该数字
        // 3. HashSet中存储的即为只出现一次的数字
        HashSet<Integer> countSet = new HashSet<Integer>();
        for(int i=0;i<array.length;i++) {
            int key = array[i];
            if(countSet.contains(key)) countSet.remove(key);
            else countSet.add(key);
        }
        Iterator<Integer> iterator = countSet.iterator();
        num1[0] = iterator.next();
        num2[0] = iterator.next();
}
```

例6：左旋转字符串（42）

题目描述

汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。对于一个给定的字符序列S，请你把其循环左移K位后的序列输出。例如，字符序列S=”abcXYZdef”,要求输出循环左移3位后的结果，即“XYZdefabc”。是不是很简单？OK，搞定它！

程序代码

```
        public String LeftRotateString(String str,int n) {
                if(n>str.length())return "";
            String leftStr = str.substring(0,n);
            String rightStr = str.substring(n,str.length());
            return rightStr + leftStr;
        }
```

例7：反转单词顺序序列（43）

题目描述

牛客最近来了一个新员工Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事Cat对Fish写的内容颇感兴趣，有一天他向Fish借来翻看，但却读不懂它的意思。例如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了，正确的句子应该是“I am a student.”。Cat对一一的翻转这些单词顺序可不在行，你能帮助他么？

程序代码

```
public String ReverseSentence(String str) {
                    // 1. 获取字符串中各个单词
                    // 2. 将各个单词序列倒转
                    // 3. 将顺序倒转后的单词进行拼接
                    String result = "";
                String[] strList = str.split(" ");  // 获取字符串中各个单词
                if(strList == null || strList.length == 0)return str;
                for(int i=0;i<strList.length/2;i++) {
                        String temp = strList[i];
                        strList[i] = strList[strList.length-1-i];
                        strList[strList.length-1-i] = temp;
                }
                for(int i=0;i<strList.length;i++) {
                        if(i == strList.length-1)result += strList[i];
                        else result += strList[i]+" ";
                }
                return result;
            }
```

例8：把字符串转换成整数（48）

题目描述

将一个字符串转换成一个整数(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库函数。 数值为0或者字符串不是一个合法的数值则返回0。

输入描述:

输入一个字符串,包括数字字母符号,可以为空

输出描述:

如果是合法的数值表达则返回该数字，否则返回0

程序代码

```
        // 48. 把字符串转换成整数
        // 将一个字符串转换成一个整数(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库函数。 数值为0或者字符串不是一个合法的数值则返回0。
        // 输入描述:
        //  输入一个字符串,包括数字字母符号,可以为空
            // 输出描述:
            // 如果是合法的数值表达则返回该数字，否则返回0
        public int StrToInt(String str) {
            // 字符串逐个遍历
                // 若该字符位于'0'~'9'之间，则转换为数字,否则返回0
                // 第一个字符单独处理
                int result = 0;
                boolean isPostive = true;
                if(str!=null && str.length()>0) {
                    Character firstChar = str.charAt(0);
                    if(convertCharToInt(firstChar)==-1) {
                        if(firstChar == '+' || firstChar == '-') {
                            if(firstChar == '-')isPostive = false;
                        }
                        else return 0;
                    }else result = convertCharToInt(firstChar);

                    for(int i=1;i<str.length();i++) {
                        if(convertCharToInt(str.charAt(i))==-1) return 0;
                        else result = result*10 + convertCharToInt(str.charAt(i));
                    }
                }
                return isPostive?result:0-result;
        }

        public int convertCharToInt(Character c) {
                if(c>='0' && c<='9')return c-'0';
                else return -1;
        }
```

例9：数组中重复的数字（49）

题目描述

在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是第一个重复的数字2。

程序代码

```
// 49.数组中重复的数字
        // 在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。
        // 请找出数组中任意一个重复的数字。 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是第一个重复的数字2。
        // Parameters:
        //    numbers:     an array of integers
```

```
//    length:      the length of array numbers
//    duplication: (Output) the duplicated number in the array number,length of duplication array is 1,so using duplication[0] = ? in implementation;
//                 Here duplication like pointor in C/C++, duplication[0] equal *duplication in C/C++
//    这里要特别注意~返回任意重复的一个，赋值duplication[0]
// Return value:       true if the input is valid, and there are some duplications in the array number
//                     otherwise false
public boolean duplicate(int numbers[],int length,int [] duplication) {
        // 利用HashSet作为字符字典存储字符
        // 每遍历一个字符查看其是否位于字典中，若位于则放入duplication数组
        // 若第一次访问，则放入字典中
        boolean isDuplication = false;
        HashSet<Integer> dictionary = new HashSet<Integer>();
        int j = 0;  // 重复
        for(int i=0;i<length;i++) {
            if(dictionary.contains(numbers[i])) {
                duplication[j] = numbers[i];
                isDuplication = true;
            }
            else dictionary.add(numbers[i]);
        }
        return isDuplication;
}
```

例10：字符流中第一个不重复的字符（53）

题目描述

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是"g"。当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"。

输出描述:

如果当前字符流没有存在出现一次的字符，返回#字符。

程序代码

```
// 53.字符流中第一个不重复的字符
// 请实现一个函数用来找出字符流中第一个只出现一次的字符。
// 例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是"g"。
// 当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"。
// 输出描述:
    // 如果当前字符流没有存在出现一次的字符，返回#字符。
Character[] visited = new Character[128];           // 存储访问过的元素,数组下标为对应字符ASCII
Character[] dup = new Character[128];               // 存储重复的元素
List<Character> datas = new ArrayList<Character>();     // 存储数据
//Insert one char from stringstream
public void Insert(char ch)
{
    if(visited[ch] != null)dup[ch] = ch;
    else visited[ch] = ch;
    datas.add(ch);
}
//return the first appearence once char in current stringstream
  public char FirstAppearingOnce()
  {
          for(int i=0;i<datas.size();i++)
              if(dup[datas.get(i)] == null)return datas.get(i);
          return '#';
  }
```