04.map和set的使用

1. 序列式容器和关联式容器

前面我们已经接触过STL中的部分容器如：string、vector、list、deque、array、forward_list等，这些容器统称为序列式容器，因为逻辑结构为线性序列的数据结构，两个位置存储的值之间一般没有紧密的关联关系，比如交换一下，他依旧是序列式容器。顺序容器中的元素是按他们在容器中的存储位置来顺序保存和访问的。

关联式容器也是用来存储数据的，与序列式容器不同的是，关联式容器逻辑结构通常是非线性结构，两个位置有紧密的关联关系，交换一下，他的存储结构就被破坏了。顺序容器中的元素是按关键字来保存和访问的。关联式容器有map/set系列和unordered_map/unordered_set系列。

本章节讲解的map和set底层是红黑树，红黑树是一颗平衡二叉搜索树。set是key搜索场景的结构，map是key/value搜索场景的结构。

2. set系列的使用

2.1 set和multiset参考文档

https://legacy.cplusplus.com/reference/set/

2.2 set类的介绍

- set的声明如下，T就是set底层关键字的类型
- set默认要求T支持小于比较，如果不支持或者想按自己的需求走可以自行实现仿函数传给第二个模版参数
- set底层存储数据的内存是从空间配置器申请的，如果需要可以自己实现内存池，传给第三个参数。
- 一般情况下，我们都不需要传后两个模版参数。
- set底层是用红黑树实现，增删查效率是 $O(logN)$ ，迭代器遍历是走的搜索树的中序，所以是有序的。
- 前面部分我们已经学习了vector/list等容器的使用，STL容器接口设计，高度相似，所以这里我们就不再一个接口一个接口的介绍，而是直接带着大家看文档，挑比较重要的接口进行介绍。

```
template < class T,                        // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value_compare
           class Alloc = allocator<T>      // set::allocator_type
           > class set;
```

2.3 set的构造和迭代器

set的构造我们关注以下几个接口即可。

set的支持正向和反向迭代遍历，遍历默认按升序顺序，因为底层是二叉搜索树，迭代器遍历走的中序；支持迭代器就意味着支持范围for，set的iterator和const_iterator都不支持迭代器修改数据，修改关键字数据，破坏了底层搜索树的结构。

```
// empty (1) 无参默认构造
explicit set (const key_compare& comp = key_compare(),
              const allocator_type& alloc = allocator_type());
// range (2) 迭代器区间构造
template <class InputIterator>
  set (InputIterator first, InputIterator last,
       const key_compare& comp = key_compare(),
       const allocator_type& = allocator_type());

// copy (3) 拷贝构造
set (const set& x);

// initializer list (5) initializer 列表构造
set (initializer_list<value_type> il,
     const key_compare& comp = key_compare(),
     const allocator_type& alloc = allocator_type());

// 迭代器是一个双向迭代器
iterator   -> a bidirectional iterator to const value_type

// 正向迭代器
iterator begin();
iterator end();
// 反向迭代器
reverse_iterator rbegin();
reverse_iterator rend();
```

2.4 set的增删查

set的增删查关注以下几个接口即可：

```
Member types
key_type   -> The first template parameter (T)
value_type -> The first template parameter (T)

```

```
// 单个数据插入，如果已经存在则插入失败
pair<iterator,bool> insert (const value_type& val);
// 列表插入，已经在容器中存在的值不会插入
void insert (initializer_list<value_type> il);
// 迭代器区间插入，已经在容器中存在的值不会插入
template <class InputIterator>
void insert (InputIterator first, InputIterator last);

// 查找val，返回val所在的迭代器，没有找到返回end()
iterator find (const value_type& val);
// 查找val，返回Val的个数
size_type count (const value_type& val) const;
// 删除一个迭代器位置的值
iterator  erase (const_iterator position);
// 删除val，val不存在返回0，存在返回1
size_type erase (const value_type& val);
// 删除一段迭代器区间的值
iterator  erase (const_iterator first, const_iterator last);

// 返回大于等val位置的迭代器
iterator lower_bound (const value_type& val) const;
// 返回大于val位置的迭代器
iterator upper_bound (const value_type& val) const;
```

2.5 insert和迭代器遍历使用样例：

```
#include<iostream>
#include<set>
using namespace std;

int main()
{
    // 去重+升序排序
    set<int> s;
    // 去重+降序排序（给一个大于的仿函数）
    //set<int, greater<int>> s;
    s.insert(5);
    s.insert(2);
    s.insert(7);
    s.insert(5);

    //set<int>::iterator it = s.begin();
    auto it = s.begin();
    while (it != s.end())
    {
```

```
            // error C3892: “it”: 不能给常量赋值
            // *it = 1;
            cout << *it << " ";
            ++it;
    }
    cout << endl;

    // 插入一段initializer_list列表值，已经存在的值插入失败
    s.insert({ 2,8,3,9 });
    for (auto e : s)
    {
            cout << e << " ";
    }
    cout << endl;

    set<string> strset = { "sort", "insert", "add" };
    // 遍历string比较ascll码大小顺序遍历的
    for (auto& e : strset)
    {
            cout << e << " ";
    }
    cout << endl;
}
```

2.6 find和erase使用样例：

```
#include<iostream>
#include<set>
using namespace std;

int main()
{
    set<int> s = { 4,2,7,2,8,5,9 };
    for (auto e : s)
    {
            cout << e << " ";
    }
    cout << endl;

    // 删除最小值
    s.erase(s.begin());
    for (auto e : s)
    {
            cout << e << " ";
    }
```

```
20      cout << endl;
21
22      // 直接删除x
23      int x;
24      cin >> x;
25      int num = s.erase(x);
26      if (num == 0)
27      {
28              cout << x << "不存在! " << endl;
29      }
30
31      for (auto e : s)
32      {
33              cout << e << " ";
34      }
35      cout << endl;
36
37      // 直接查找在利用迭代器删除x
38      cin >> x;
39      auto pos = s.find(x);
40      if (pos != s.end())
41      {
42              s.erase(pos);
43      }
44      else
45      {
46              cout << x << "不存在! " << endl;
47      }
48
49      for (auto e : s)
50      {
51              cout << e << " ";
52      }
53      cout << endl;
54
55      // 算法库的查找 O(N)
56      auto pos1 = find(s.begin(), s.end(), x);
57      // set自身实现的查找 O(logN)
58      auto pos2 = s.find(x);
59
60      // 利用count间接实现快速查找
61      cin >> x;
62      if (s.count(x))
63      {
64              cout << x << "在! " << endl;
65      }
66      else
```

```
67      {
68              cout << x << "不存在! " << endl;
69      }
70
71      return 0;
72  }
```

```
1  #include<iostream>
2  #include<set>
3  using namespace std;
4
5  int main()
6  {
7      std::set<int> myset;
8      for (int i = 1; i < 10; i++)
9              myset.insert(i * 10); // 10 20 30 40 50 60 70 80 90
10
11     for (auto e : myset)
12     {
13             cout << e << " ";
14     }
15     cout << endl;
16
17     // 实现查找到的[itlow,itup)包含[30, 60]区间
18
19     // 返回 >= 30
20     auto itlow = myset.lower_bound(30);
21     // 返回 > 60
22     auto itup = myset.upper_bound(60);
23
24     // 删除这段区间的值
25     myset.erase(itlow, itup);
26
27     for (auto e : myset)
28     {
29             cout << e << " ";
30     }
31     cout << endl;
32
33     return 0;
34 }
```


2.7 multiset和set的差异

multiset和set的使用基本完全类似，主要区别点在于multiset支持值冗余，那么insert/find/count/erase都围绕着支持值冗余有所差异，具体参看下面的样例代码理解。

```cpp
#include<iostream>
#include<set>
using namespace std;

int main()
{
    // 相比set不同的是，multiset是排序，但是不去重
    multiset<int> s = { 4,2,7,2,4,8,4,5,4,9 };
    auto it = s.begin();
    while (it != s.end())
    {
            cout << *it << " ";
            ++it;
    }
    cout << endl;

    // 相比set不同的是，x可能会存在多个，find查找中序的第一个
    int x;
    cin >> x;
    auto pos = s.find(x);
    while (pos != s.end() && *pos == x)
    {
            cout << *pos << " ";
            ++pos;
    }
    cout << endl;

    // 相比set不同的是，count会返回x的实际个数
    cout << s.count(x) << endl;

    // 相比set不同的是，erase给值时会删除所有的x
    s.erase(x);
    for (auto e : s)
    {
            cout << e << " ";
    }
    cout << endl;

    return 0;
}
```


2.8 349. 两个数组的交集 - 力扣（LeetCode）

```cpp
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        set<int> s1(nums1.begin(), nums1.end());
        set<int> s2(nums2.begin(), nums2.end());

        // 因为set遍历是有序的，有序值，依次比较
        // 小的++，相等的就是交集
        vector<int> ret;
        auto it1 = s1.begin();
        auto it2 = s2.begin();
        while(it1 != s1.end() && it2 != s2.end())
        {
            if(*it1 < *it2)
            {
                it1++;
            }
            else if(*it1 > *it2)
            {
                it2++;
            }
            else
            {
                ret.push_back(*it1);
                it1++;
                it2++;
            }
        }

        return ret;
    }
};
```


2.9 142. 环形链表 II - 力扣（LeetCode）

数据结构初阶阶段，我们通过证明一个指针从头开始走一个指针从相遇点开始走，会在入口点相遇，理解证明都会很麻烦。这里我们使用set查找记录解决非常简单方便，这里体现了set在解决一些问题时的价值，完全是降维打击。

```cpp
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        set<ListNode*> s;
        ListNode* cur = head;
```

```
        while(cur)
        {
            auto ret = s.insert(cur);
            if(ret.second == false)
                return cur;

            cur = cur->next;
        }

        return nullptr;

    }
};
```

3. map系列的使用

3.1 map和multimap参考文档

https://legacy.cplusplus.com/reference/map/

3.2 map类的介绍

map的声明如下，Key就是map底层关键字的类型，T是map底层value的类型，set默认要求Key支持小于比较，如果不支持或者需要的话可以自行实现仿函数传给第二个模版参数，map底层存储数据的内存是从空间配置器申请的。一般情况下，我们都不需要传后两个模版参数。map底层是用红黑树实现，增删查改效率是 $O(logN)$ ，迭代器遍历是走的中序，所以是按key有序顺序遍历的。

```
template < class Key,                                     // map::key_type
           class T,                                       // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

3.3 pair类型介绍

map底层的红黑树节点中的数据，使用pair<Key, T>存储键值对数据。

```
typedef pair<const Key, T> value_type;

template <class T1, class T2>
struct pair
{
```

```
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair(): first(T1()), second(T2())
    {}

    pair(const T1& a, const T2& b): first(a), second(b)
    {}

    template<class U, class V>
    pair (const pair<U,V>& pr): first(pr.first), second(pr.second)
    {}
};

template <class T1,class T2>
inline pair<T1,T2> make_pair (T1 x, T2 y)
{
    return ( pair<T1,T2>(x,y) );
}
```

3.4 map的构造

map的构造我们关注以下几个接口即可。

map的支持正向和反向迭代遍历，遍历默认按key的升序顺序，因为底层是二叉搜索树，迭代器遍历走的中序；支持迭代器就意味着支持范围for，map支持修改value数据，不支持修改key数据，修改关键字数据，破坏了底层搜索树的结构。

```
// empty (1) 无参默认构造
explicit map (const key_compare& comp = key_compare(),
              const allocator_type& alloc = allocator_type());

// range (2) 迭代器区间构造
template <class InputIterator>
  map (InputIterator first, InputIterator last,
       const key_compare& comp = key_compare(),
       const allocator_type& = allocator_type());
// copy (3) 拷贝构造
map (const map& x);

// initializer list (5) initializer 列表构造
map (initializer_list<value_type> il,
```

```
      const key_compare& comp = key_compare(),
      const allocator_type& alloc = allocator_type());

// 迭代器是一个双向迭代器
iterator   -> a bidirectional iterator to const value_type

// 正向迭代器
iterator begin();
iterator end();
// 反向迭代器
reverse_iterator rbegin();
reverse_iterator rend();
```

3.5 map的增删查

map的增删查关注以下几个接口即可：

map增接口，插入的pair键值对数据，跟set所有不同，但是查和删的接口只用关键字key跟set是完全类似的，不过find返回iterator，不仅仅可以确认key在不在，还找到key映射的value，同时通过迭代还可以修改value

```
Member types
key_type     -> The first template parameter (Key)
mapped_type  -> The second template parameter (T)
value_type   -> pair<const key_type,mapped_type>

// 单个数据插入，如果已经key存在则插入失败,key存在相等value不相等也会插入失败
pair<iterator,bool> insert (const value_type& val);
// 列表插入，已经在容器中存在的值不会插入
void insert (initializer_list<value_type> il);
// 迭代器区间插入，已经在容器中存在的值不会插入
template <class InputIterator>
void insert (InputIterator first, InputIterator last);

// 查找k，返回k所在的迭代器，没有找到返回end()
iterator find (const key_type& k);
// 查找k，返回k的个数
size_type count (const key_type& k) const;

// 删除一个迭代器位置的值
iterator  erase (const_iterator position);
// 删除k，k存在返回0，存在返回1
size_type erase (const key_type& k);
// 删除一段迭代器区间的值
iterator  erase (const_iterator first, const_iterator last);
```

```
25
26 // 返回大于等k位置的迭代器
27 iterator lower_bound (const key_type& k);
28 // 返回大于k位置的迭代器
29 const_iterator lower_bound (const key_type& k) const;
```

3.6 map的数据修改

前面我提到map支持修改mapped_type 数据，不支持修改key数据，修改关键字数据，破坏了底层搜索树的结构。

map第一个支持修改的方式时通过迭代器，迭代器遍历时或者find返回key所在的iterator修改，map还有一个非常重要的修改接口operator[]，但是operator[]不仅仅支持修改，还支持插入数据和查找数据，所以他是一个多功能复合接口

需要注意从内部实现角度，map这里把我们传统说的value值，给的是T类型，typedef为mapped_type。而value_type是红黑树结点中存储的pair键值对值。日常使用我们还是习惯将这里的T映射值叫做value。

```
1  Member types
2  key_type      -> The first template parameter (Key)
3  mapped_type   -> The second template parameter (T)
4  value_type    -> pair<const key_type,mapped_type>
5
6  // 查找k，返回k所在的迭代器，没有找到返回end()，如果找到了通过iterator可以修改key对应的
   mapped_type值
7  iterator find (const key_type& k);
8
9  // 文档中对insert返回值的说明
10 // The single element versions (1) return a pair, with its member pair::first
   set to an iterator pointing to either the newly inserted element or to the
   element with an equivalent key in the map. The pair::second element in the pair
    is set to true if a new element was inserted or false if an equivalent key
   already existed.
11 // insert插入一个pair<key, T>对象
12 // 1、如果key已经在map中，插入失败，则返回一个pair<iterator,bool>对象，返回pair对象
   first是key所在结点的迭代器，second是false
13 // 2、如果key不在在map中，插入成功，则返回一个pair<iterator,bool>对象，返回pair对象
   first是新插入key所在结点的迭代器，second是true
14 // 也就是说无论插入成功还是失败，返回pair<iterator,bool>对象的first都会指向key所在的迭
   代器
15 // 那么也就意味着insert插入失败时充当了查找的功能，正是因为这一点，insert可以用来实现
   operator[]
16 // 需要注意的是这里有两个pair，不要混淆了，一个是map底层红黑树节点中存的pair<key, T>，另
   一个是insert返回值pair<iterator,bool>
```

```
pair<iterator,bool> insert (const value_type& val);

mapped_type& operator[] (const key_type& k);

// operator的内部实现
mapped_type& operator[] (const key_type& k)
{
    // 1、如果k不在map中，insert会插入k和mapped_type默认值，同时[]返回结点中存储mapped_type值的引用，那么我们可以通过引用修改返映射值。所以[]具备了插入+修改功能
    // 2、如果k在map中，insert会插入失败，但是insert返回pair对象的first是指向key结点的迭代器，返回值同时[]返回结点中存储mapped_type值的引用，所以[]具备了查找+修改的功能
    pair<iterator, bool> ret = insert({ k, mapped_type() });
    iterator it = ret.first;
    return it->second;
}
```

3.7 构造遍历及增删查使用样例

```
#include<iostream>
#include<map>
using namespace std;

int main()
{
        // initializer_list构造及迭代遍历
        map<string, string> dict = { {"left", "左边"}, {"right", "右边"}, {"insert", "插入"},{ "string", "字符串" } };

        //map<string, string>::iterator it = dict.begin();
        auto it = dict.begin();
        while (it != dict.end())
        {
                //cout << (*it).first <<":"<<(*it).second << endl;

                // map的迭代基本都使用operator->,这里省略了一个->
                // 第一个->是迭代器运算符重载，返回pair*，第二个箭头是结构指针解引用取pair数据
                //cout << it.operator->()->first << ":" << it.operator->()->second << endl;
                cout << it->first << ":" << it->second << endl;

                ++it;
        }
        cout << endl;

```

```
        // insert插入pair对象的4种方式，对比之下，最后一种最方便
        pair<string, string> kv1("first", "第一个");
        dict.insert(kv1);
        dict.insert(pair<string, string>("second", "第二个"));
        dict.insert(make_pair("sort", "排序"));
        dict.insert({ "auto", "自动的" });

        // "left"已经存在，插入失败
        dict.insert({ "left", "左边，剩余" });

        // 范围for遍历
        for (const auto& e : dict)
        {
                cout << e.first << ":" << e.second << endl;
        }
        cout << endl;

        string str;
        while (cin >> str)
        {
                auto ret = dict.find(str);
                if (ret != dict.end())
                {
                        cout << "->" << ret->second << endl;
                }
                else
                {
                        cout << "无此单词，请重新输入" << endl;
                }
        }

        // erase等接口跟set完全类似，这里就不演示讲解了

        return 0;
}
```

3.8 map的迭代器和[]功能样例：

```
#include<iostream>
#include<map>
#include<string>
using namespace std;

int main()
{
```

```
        // 利用find和iterator修改功能，统计水果出现的次数
        string arr[] = { "苹果", "西瓜", "苹果", "西瓜", "苹果", "苹果", "西瓜", "苹果", "香蕉", "苹果", "香蕉" };
        map<string, int> countMap;
        for (const auto& str : arr)
        {
                // 先查找水果在不在map中
                // 1、不在，说明水果第一次出现，则插入{水果，1}
                // 2、在，则查找到的节点中水果对应的次数++
                auto ret = countMap.find(str);
                if (ret == countMap.end())
                {
                        countMap.insert({ str, 1 });
                }
                else
                {
                        ret->second++;
                }
        }

        for (const auto& e : countMap)
        {
                cout << e.first << ":" << e.second << endl;
        }
        cout << endl;

        return 0;
}

#include<iostream>
#include<map>
#include<string>
using namespace std;

int main()
{
        // 利用[]插入+修改功能，巧妙实现统计水果出现的次数
        string arr[] = { "苹果", "西瓜", "苹果", "西瓜", "苹果", "苹果", "西瓜", "苹果", "香蕉", "苹果", "香蕉" };
        map<string, int> countMap;
        for (const auto& str : arr)
        {
                // []先查找水果在不在map中
                // 1、不在，说明水果第一次出现，则插入{水果，0}，同时返回次数的引用，++一下就变成1次了
                // 2、在，则返回水果对应的次数++
                countMap[str]++;
```

```
        }

        for (const auto& e : countMap)
        {
                cout << e.first << ":" << e.second << endl;
        }
        cout << endl;

        return 0;
}

```

```
#include<iostream>
#include<map>
#include<string>
using namespace std;

int main()
{
        map<string, string> dict;
        dict.insert(make_pair("sort", "排序"));
        // key不存在->插入 {"insert", string()}
        dict["insert"];

        // 插入+修改
        dict["left"] = "左边";

        // 修改
        dict["left"] = "左边、剩余";

        // key存在->查找
        cout << dict["left"] << endl;

        return 0;
}
```


3.9 multimap和map的差异

multimap和map的使用基本完全类似，主要区别点在于multimap支持关键值key冗余，那么insert/find/count/erase都围绕着支持关键值key冗余有所差异，这里跟set和multiset完全一样，比如find时，有多个key，返回中序第一个。其次就是multimap不支持[]，因为支持key冗余，[]就只能支持插入了，不能支持修改。

3.10 138. 随机链表的复制 - 力扣（LeetCode）

数据结构初阶阶段，为了控制随机指针，我们将拷贝结点链接在原节点的后面解决，后面拷贝节点还得解下来链接，非常麻烦。这里我们直接让{原结点,拷贝结点}建立映射关系放到map中，控制随机指针会非常简单方便，这里体现了map在解决一些问题时的价值，完全是降维打击。

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        map<Node*, Node*> nodeMap;
        Node* copyhead = nullptr,*copytail = nullptr;
        Node* cur = head;
        while(cur)
        {
            if(copytail == nullptr)
            {
                copyhead = copytail = new Node(cur->val);
            }
            else
            {
                copytail->next = new Node(cur->val);
                copytail = copytail->next;
            }

            // 原节点和拷贝节点map kv存储
            nodeMap[cur] = copytail;

            cur = cur->next;
        }

        // 处理random
        cur = head;
        Node* copy = copyhead;
        while(cur)
        {
            if(cur->random == nullptr)
            {
                copy->random = nullptr;
            }
            else
            {
                copy->random = nodeMap[cur->random];
            }

            cur = cur->next;
```

```
            copy = copy->next;
        }

        return copyhead;
    }
};
```

3.11 692. 前K个高频单词 - 力扣（LeetCode）

本题目我们利用map统计出次数以后，返回的答案应该按单词出现频率由高到低排序，有一个特殊要求，如果不同的单词有相同出现频率，按字典顺序排序。

解决思路1：

用排序找前k个单词，因为map中已经对key单词排序过，也就意味着遍历map时，次数相同的单词，字典序小的在前面，字典序大的在后面。那么我们将数据放到vector中用一个稳定的排序就可以实现上面特殊要求，但是sort底层是快排，是不稳定的，所以我们要用stable_sort，他是稳定的。

```
class Solution {
public:
    struct Compare
    {
        bool operator()(const pair<string, int>& x, const pair<string, int>& y) const
        {
            return x.second > y.second;
        }
    };

    vector<string> topKFrequent(vector<string>& words, int k) {
        map<string, int> countMap;
        for(auto& e : words)
        {
            countMap[e]++;
        }

        vector<pair<string, int>> v(countMap.begin(), countMap.end());
        // 仿函数控制降序
        stable_sort(v.begin(), v.end(), Compare());
        //sort(v.begin(), v.end(), Compare());

        // 取前k个
        vector<string> strV;
        for(int i = 0; i < k; ++i)
        {
            strV.push_back(v[i].first);
```

```
        }

        return strV;
    }
};
```

解决思路2：

将map统计出的次数的数据放到vector中排序，或者放到priority_queue中来选出前k个。利用仿函数强行控制次数相等的，字典序小的在前面。

```
class Solution {
public:
    struct Compare
    {
        bool operator()(const pair<string, int>& x, const pair<string, int>& y) const
        {
            return x.second > y.second || (x.second == y.second && x.first < y.first);;
        }
    };

    vector<string> topKFrequent(vector<string>& words, int k) {
        map<string, int> countMap;
        for(auto& e : words)
        {
            countMap[e]++;
        }

        vector<pair<string, int>> v(countMap.begin(), countMap.end());
        // 仿函数控制降序,仿函数控制次数相等，字典序小的在前面
        sort(v.begin(), v.end(), Compare());

        // 取前k个
        vector<string> strV;
        for(int i = 0; i < k; ++i)
        {
            strV.push_back(v[i].first);
        }

        return strV;
    }
};
```

```cpp
class Solution {
public:
    struct Compare
    {
        bool operator()(const pair<string, int>& x, const pair<string, int>& y) const
        {
            // 要注意优先级队列底层是反的，大堆要实现小于比较，所以这里次数相等，想要字典序小的在前面要比较字典序大的为真
            return x.second < y.second || (x.second == y.second && x.first > y.first);
        }
    };

    vector<string> topKFrequent(vector<string>& words, int k) {
        map<string, int> countMap;
        for(auto& e : words)
        {
            countMap[e]++;
        }

        // 将map中的<单词，次数>放到priority_queue中，仿函数控制大堆，次数相同按照字典序规则排序
        priority_queue<pair<string, int>, vector<pair<string, int>>, Compare> p(countMap.begin(), countMap.end());

        vector<string> strV;
        for(int i = 0; i < k; ++i)
        {
            strV.push_back(p.top().first);
            p.pop();
        }

        return strV;
    }
};
```