C++ 标准模板库

C++ Standard Template Libarary

使用模板的程序设计法：

将一些常用的数据结构（比如链表，堆，二叉树）和算法（比如排序，二分查找）写成模板，以后则不论数据结构里放的是什么数据，算法针对什么样的对象，都不必重新编写数据结构和算法，直接调用现成的模板就行。

引言：STL

STL是C++标准模板库（Standard Template Library）的缩写。

就是一些常用数据结构和算法的模板的集合(主要由 Alex Stepanov 开发，于1998年被添加进C++标准)。

STL以模板类和模板函数的形式为程序员提供了各种数据结构和算法的精巧实现。如果能够充分地利用STL，可以在代码空间、执行时间和编码效率上获得极大的好处。有了STL，不必再从头写大多的标准数据结构和算法，并且可获得非常高的性能。

引言：STL

STL 大致可以分为三大类：

容器(Container)

迭代器(iterator)

算法(algorithm)

1.容器(Container)

容器：可容纳各种数据类型的数据结构。

容器分为三大类：

1) 顺序容器

vector：可变长度的数组。后部插入/删除，直接访问

deque：双端队列。前/后部插入/删除，直接访问

list：双向链表，任意位置插入/删除

2)关联容器

set：快速查找，无重复元素

multiset ：快速查找，可有重复元素

map：一对一映射，无重复元素，基于关键字查找

multimap ：一对一映射，可有重复元素，基于关键字查找

3)容器适配器

stack：栈。LIFO

queue：队列。FIFO

priority_queue：优先队列。优先级高的元素先出

stack

stack 栈

stack(栈)。是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项。即按照后进先出的原则。

#include <stack>
常用函数：
push(value) - 将元素压栈
top() - 返回栈顶元素的值，可读可写，但不移除
pop() - 从栈中移除栈顶元素，但不返回该元素的值
size()-计算栈中元素的个数
empty()-判断栈是否为空，为空结果为true,否则为false

stack 栈

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

int main ()
{
    stack<int> sk;                          //申明一个int类型的stack变量sk
    int sum =0;
    for (int i=1;i<=10;i++) sk.push(i);     //将数字1到10入栈
    cout<<sk.size();                        //输出栈中元素个数10
    while (!sk.empty())                     //只要栈不为空
    {
        sum += sk.top();                    //将栈顶元素的值累加
        sk.pop();                           //栈顶元素出栈
    }
  cout << sum << endl;
  return 0;
}
```

queue

queue 队列

queue（队列），插入只可以在尾部进行，删除、检索和修改只允许从头部进行。按照先进先出的原则。

#include <queue>
常用函数：
push(e) - 将元素加入队列
front() - 返回队首元素的值，可读可写，但不移除
back() - 返回队尾元素的值，可读可写，但不移除
pop() - 将队首元素移除，但不返回值
size()-队列中元素的个数
empty()-判断队列是否为空

queue 队列

```
#include<queue>
void spfa(int s)                          //s为起点
{
    queue<int>Q;    //申明一个int类型的队列Q
    int dis[maxn];     // dis记录节点到起点的距离
    bool f[maxn];     //用于标记节点是否在队列中
    int i,x;

    for(i=1;i<=n;i++)dis[i]=99999999;        //将距离初始化为一个很大的值
    Q.push(s);     f[s]=true;   dis[s]=0;        //初始化起点，将起点入队

    while(!Q.empty())                          //队列不为空，则继续讨论
    {
        x=Q.front();  f[x]=false;    Q.pop();   //取出队首元素的值，然后将其出队，标记为不在队中
        for(i=1;i<=n;i++)                       //讨论n个点中与x相连的点
          if (dis[x]+map[x][i]<dis[i])
          {
              dis[i]=dis[x]+map[x][i];
              if (f[i]==false)                  //如果i点松弛成功且不在队中，入队
              {
                  Q.push(i); f[i]=true;         //将i号点入队，并标记为已入队
              }
          }
     }
}
```

map

关联容器：map

关联式容器内的元素是**排序**的，插入任何元素，都按相应的排序准则来确定其位置。关联式容器的特点是在查找时具有非常好的性能。

map一种关联式的容器，它的基本信息如下：

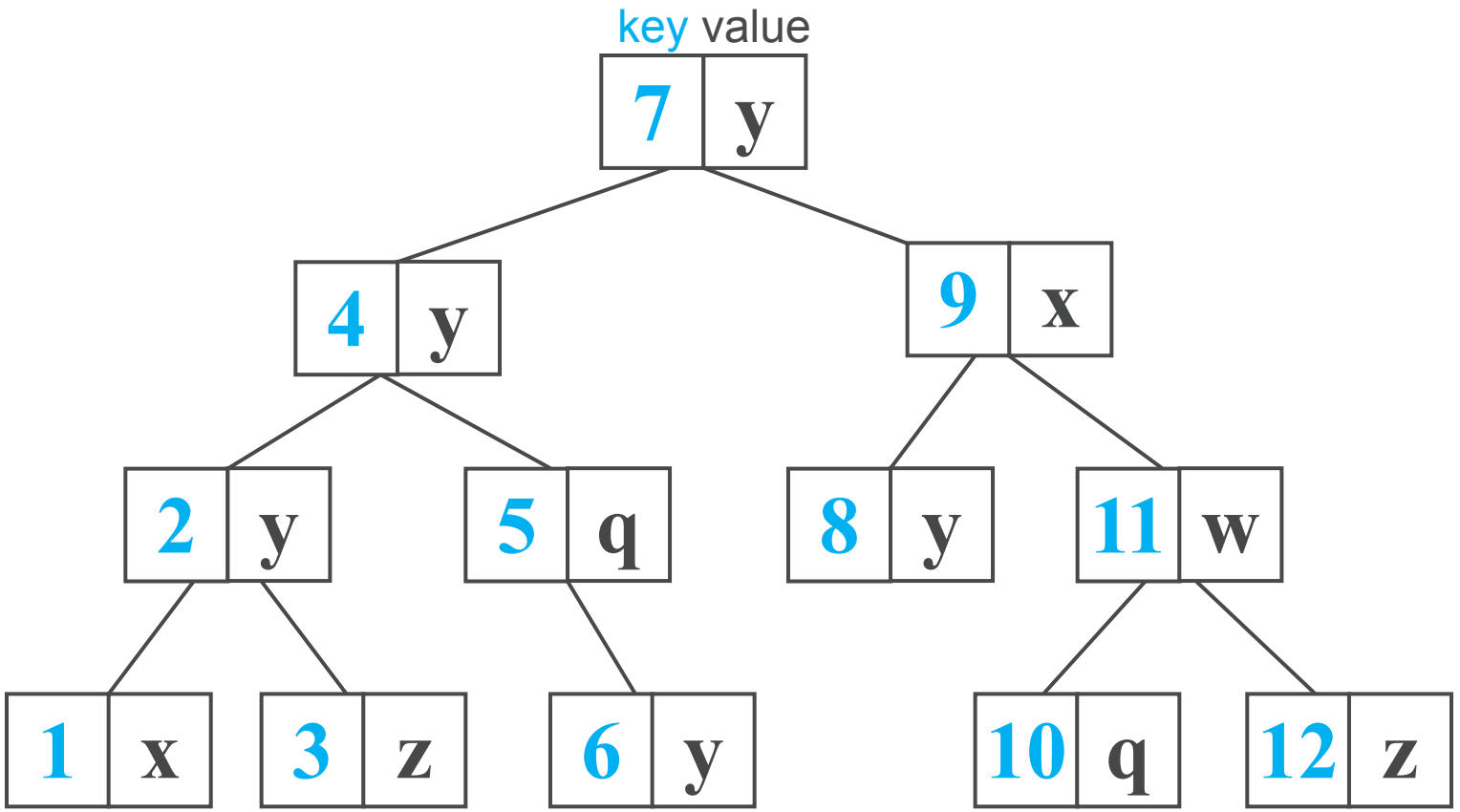
1. map/multimap　头文件　#include<map>
2. map中存放的是成对的数据，一个称为key("键")，一个称为value("值")。
3. map自动根据key值对元素进行排序，可快速地根据key来查找元素。
4. map**不允许**多个元素有相同的key值。

   multimap**允许**多个元素有相同的key值。
5. map的容积是动态的，可增可减，每个元素的key值不能更改。
6. map是以**平衡二叉树**方式实现，插入、查找和删除的时间都是 **O(logN)**

map的内部结构：平衡树

- map/multimap

  内部存储结构，使用平衡二叉树管理元素



map例题1：使用下标访问map元素

例1：某校开学，学生们注册报到。每个学生的名字都由字母构成(名字的长度<=1000)，开学当天，学校面临两种操作：

1号操作：一个名为S的新生加入了学校；

2号操作：查询学校是否有名为S的学生；

对于每次查询操作，输出查询到的名为S的学生的数目。

输入格式：

第一行，整数n,操作的总次数。

接下来n行，每行由一个数字和字符串构成，以空格间隔。数字表示操作，字符串表示学生的名字

输出格式：

若干行，每行一个整数，表示查询的答案。

样例输入：

```
7
1 Tom
1 Adan
1 Tom
1 Adan
2 Tom
1 Adan
2 Adan
```

样例输出：

```
2
3
```

map例1：使用下标访问map元素

#include<map> //使用map容器必须包含此头文件

map<string,int> Name;

//申明map类型的变量Name,它存储关联数据为string和int, string为key(键),int为value(值)

Name["Jim"] = 5;

这句代码的执行过程是：

首先自动在map容器Name中查找key为"Jim"的元素,

1.若未找到，将一个新的**键-值**数据对插入到Name中。它的key(键)存储string类型的数据"Jim"，而它的value(值)采用值初始化，在本例中值为5。

2.若找到，则将该元素的value(值)改为5。

注意：**键-值**数据对中的key即是map类型的下标，上例中Name["Jim"]=5,是通过下标"Jim"来访问的。对于map类型，用下标访问不存在的元素将导致在容器中添加一个新的元素，它的键即为该下标值。

map例1：使用下标访问map元素

```
#include<iostream>
#include<cstring>
#include<map>            //使用map容器必须包含此头文件
using namespace std;
map<string,int> Name; //申明map类型的变量Name,它存储关联数据为string和int,string为key
int main()
{
        int n,i,k;
        string str;
        cin>>n;
        for(i=1;i<=n;i++)
        {
                cin>>k>>str;
                if(k==1)Name[str]++; //若存在键为str的元素，个数增加1,否则增加一个键为str的元素
                else cout<<Name[str]<<endl;
        }
        return 0;
}
```

此处“Name”作用相当于字符串hash

map例题2：使用下标访问map元素

例2：某校开学，学生们注册报到。每个学生的名字都由字母构成，开学当天，学校面临三种操作：

1号操作：一个名为S的新生加入了学校；

2号操作：查询学校是否有名为S的学生；

对于每次查询操作，输出查询到的名为S的学生的数目。

3号操作：删除名为S的新生。

输入格式：

第一行，整数n,操作的总次数。

接下来n行，每行由一个数字和字符串构成，以空格间隔。数字表示操作，字符串表示学生的名字

输出格式：

若干行，每行一个整数，表示查询的答案。

样例输入：

```
7
1 Tom
1 Adan
1 Tom
1 Adan
2 Tom
3 Adan
2 Adan
```

样例输出：

```
2
0
```

map例2：使用下标访问map元素

```
#include<iostream>
#include<map>
#include<cstring>
using namespace std;
map<string,int> Name;
int main()
{
        int n,i,k;
        string str;
        cin>>n;
        for(i=1;i<=n;i++)
        {
                cin>>k>>str;
                if(k==1)Name[str]++;
                else if(k==2)cout<<Name[str]<<endl;
                        else Name[str]=0; //将键为str的元素对应的值标记为0
        }
        return 0;
}
```

map例2：使用下标访问map元素

```
#include<iostream>
#include<map>
#include<cstring>
using namespace std;
map<string,int> Name;
int main()
{
        int n,i,k;
        string str;
        cin>>n;
        for(i=1;i<=n;i++)
        {
                cin>>k>>str;
                if(k==1)Name[str]++;
                else if(k==2)cout<<Name[str]<<endl;
                        else Name.erase(str); //删除map容器中键为str的元素
        }
        return 0;
}
```

map的删除函数erase()
Name.erase("Tom");
删除map容器Name中键为"Tom"的元素
若是multimap，可写成
int cnt=Name.erase("Tom");
删除键为"Tom"的元素的同时，返回元素个数

map:查询

```
#include<map>
map<string,int> Name;
cout<<Name["Jim"] ;      //前面我们进行查询是用这样的下标的方式。
```

如果只想查询map中是否存在某个键，并不想插入一个新元素，则**最好不用**下标方式。

map容器提供了两个操作：count()和find()，用于检查某个键是否存在而不会插入该键。

Name.count("Jim");返回键为“Jim”的元素的个数。
因为map中不存在重复元素，故结果只有0和1。
multimap则是返回具体的个数。

Name.find("Jim");返回键为“Jim”的元素在map中出现的位置。
若没找到，则返回map的结束位置end()。

map:查询

```
#include<map>
map<string,int> Name;
cout<<Name["Jim"] ;      //前面我们进行查询是用这样的下标的方式。
```

Name.begin(); 指向map中的第一个元素，得到的结果是一个双向**迭代器**

Name.end(); 指向map中的最后一个元素，得到的结果是一个双向**迭代器**

```
if( Name.find("Jim")==Name.end() ) cout<<"没找到!"<<endl;
```

map 与 iterator

例题3：map的遍历

例1：某校开学，学生们注册报到。每个学生的名字都由字母构成，开学当天，学校面临三种操作：

1号操作：一个名为S的新生加入了学校；

2号操作：查询学校是否有名为S的学生,输出查询到的名为S的学生的数目。

3号操作：删除名为S的新生。

4号操作：按字典序打印出所有学生的名字和对应名字的学生的个数。

输入格式：

第一行，整数n,操作的总次数。

接下来n行，每行由一个数字和字符串构成，以空格间隔。数字表示操作，字符串表示学生的名字。4号操作只有一个数字“4”。

输出格式：

若干行，每行表示查询或打印的答案。

样例输入：

```
7
1 Tom
1 Adan
3 Tom
1 Adan
1 Tom
1 Green
4
```

样例输出：

```
2
Adan 2
Green 1
Tom 1
```

例题3：map的遍历

map默认按key值得由小到大来队数据排序

```
#include<iostream>
#include<map>
#include<cstring>
using namespace std;
map<string,int> Name;
int main()
{
        int n,i,k;
        string str;
        cin>>n;
        map<string,int>::iterator it;   //申明了一个名为it的迭代器,该迭代器操作的对象数map容器
        for(i=1;i<=n;i++)
        {
                cin>>k;
                if(k==1){ cin>>str;Name[str]++; }
                else if(k==2){ cin>>str; cout<<Name[str]<<endl; }
                else if(k==3){ cin>>str; Name.erase(str); }
                    else
                    {
                       for( it=Name.begin(); it!=Name.end(); it++)//从map的开始位置枚举到结束位置
                                    cout<<it->first<<" "<<it->second<<endl;
                    } //输出it所指的元素的key和value,“->first”表示key，“->second”表示value
        }             //“->”用于指向一个复式数据的成员数据，作用类似于结构体的“.”号
        return 0;
}
```

迭代器

迭代器用于指向容器中的元素，定义一个容器类的迭代器的方法是：

容器类名::iterator 变量名;

比如：map<int,double>::iterator haha;

```
map<string,int> Name;
map<string,int>::iterator it;
for( it=Name.begin(); it!=Name.end(); it++)//从map的开始位置枚举到结束位置
                    cout<<it->first<<" "<<it->second<<endl;
```

迭代器上可以执行 ++ 操作, 以指向容器中的下一个元素。如果迭代器到达了容器中的最后一个元素的后面，则迭代器变成past-the-end值。

"->"是指向数据结构成员的运算符，map是关联类型的数据，有key和value两个数据，关键字"first"代表第一个数据key,关键字"second"代表第二个数据value。

迭代器

- 迭代器（iterator）

  可遍历STL容器内全部或部分元素的对象

  指出容器中的一个特定位置

  迭代器的基本操作

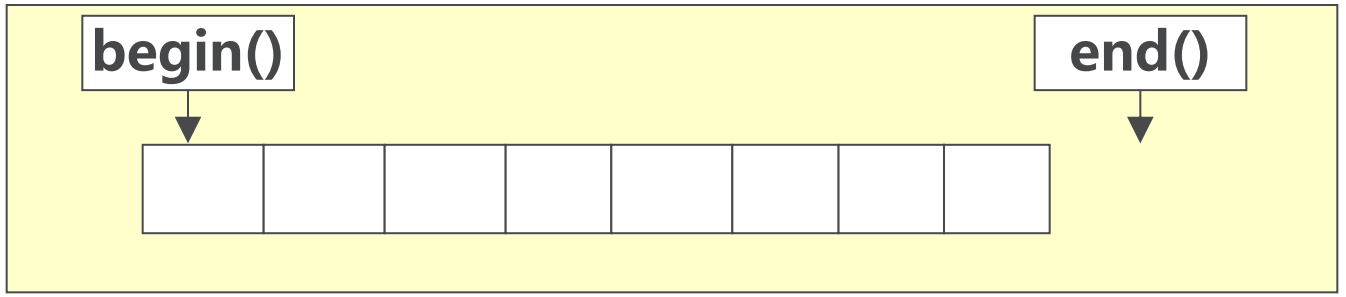
操作	效果
*	返回当前位置上的元素值。如果该元素有成员，可以通过迭代器以"->"取用
++	将迭代器前进至下一元素
==和!=	判断两个迭代器是否指向**同一位置**
=	为迭代器赋值（将所指元素的位置赋值过去）

迭代器

- 迭代器（iterator）

  所有容器都提供获得迭代器的函数

操作	效果
begin()	返回一个迭代器，指向第一个元素
end()	返回一个迭代器，指向最后一个元素之后



半开区间[begin(), end())的好处：

1.为遍历元素时循环的结束时机提供了简单的判断依据（只要未到达end()，循环就可以继续）

2.不必对空区间采取特殊处理（空区间的begin()就等于end()）

multimap与迭代器

下列代码的作用是？

```
multimap<string,string>Student;  //记录学生的姓名和家长的姓名
输入.....
int cnt = Student.count("Joe");
multimap<string, string>::iterator it = Student.find("Joe");
for(int i = 0; i != cnt; i++, it++)cout << it->second << endl;
```

输出所有名为"Joe"的学生的家长的名字。

map/multimap的操作

- map/multimap

  构造、拷贝和析构

操作	效果
map c	产生空的map容器
map c1(c2)	产生同类型的c1，并复制c2的所有元素
map c(op)	以op为排序准则产生一个空的map
map c(beg,end)	以区间[beg,end]内的元素产生一个map
map c(beg,end,op)	以op为排序准则，以区间[beg,end]内的元素产生一个map
clear()	销毁所有元素并释放内存。

其中map可以是下列形式

map<key,value> 一个以key的由小到大（<）为排序准则的map，

map<key,value,op> 一个以op为排序准则的map

map/multimap操作

- map/multimap

非变动性操作

操作	效果
c.size()	**返回元素个数**
c.empty()	**判断容器是否为空**
c.max_size()	返回元素最大可能数量
c1==c2	判断c1是否等于c2
c1!=c2	判断c1是否不等于c2
c1<c2	判断c1是否小于c2
c1>c2	判断c1是否大于c2
c1<=c2	判断c1是否大于等于c2
c1>=c2	判断c1是否小于等于c2

map/multimap操作

- map/multimap

  赋值

操作	效果
c1 = c2	将c2的全部元素赋值给c1
c1.swap(c2)	将c1和c2的元素互换
swap(c1,c2)	同上，全局函数

map/multimap操作

- map/multimap
  特殊搜寻操作

操作	效果
count(key)	返回”键值等于key”的元素个数
find(key)	返回”键值等于key”的第一个元素，找不到返回end
lower_bound(key)	返回”键值**小于等于**key”的第一个元素
upper_bound(key)	返回”键值**大于**key”的第一个元素
equal_range(key)	返回”键值**等于**key”的元素区间

map/multimap操作

- map/multimap

  迭代器相关函数

操作	效果
begin()	返回一个双向迭代器，指向第一个元素
end()	返回一个双向迭代器，指向最后一个元素之后
rbegin()	返回一个逆向迭代器，指向逆向遍历的第一个元素
rend()	返回一个逆向迭代器，指向逆向遍历的最后一个元素

map/multimap容器

- map/multimap

  安插（insert）元素

操作	效果
c.insert(pos,e)	在pos位置为起点插入e的副本，并返回新元素位置（插入速度取决于pos）
c.insert(e)	插入e的副本，并返回新元素位置
c.insert(beg,end)	将区间[beg,end]内所有元素的副本插入到c中

map/multimap容器

- map/multimap

移除（remove）元素

操作	效果
c.erase(pos)	删除迭代器pos所指位置的元素，无返回值
c.erase(val)	移除所有值为val的元素，返回移除元素个数
c.erase(beg,end)	删除区间[beg,end]内所有元素，无返回值
c.clear()	移除所有元素，清空容器

multimap的特殊处理

multimap的特殊性

multimap**不支持下标操作**
multimap允许一个键对应多个值
multimap用**insert()**添加新元素

例如，auther是一个书的作者与所写书的multimap容器，可以这样来添加2个元素：

```
multimap <string, string> author; //定义一个multimap
//插入第一个元素
author.insert(make_pair("Hemingway", "The Old Man and the Sea"));
//插入第二个元素
author.insert(make_pair("Hemingway", "The Sun Also Rises"));
```

先谈谈pair

pair的作用

pair用来把两个数据打包成一个数据

操作	说明
pair<T1,T2>p1;	创建一个空的pair对象，它的两个元素分别是T1和T2类型，采用值初始化。 就是申明一个pair类型的变量，例：pair<int,char>p1; cin>>p1.first>>p1.second;
pair<T1,T2>p1(v1, v2)	创建一个pair对象，其中first成员初始化为v1，second成员初始化为v2,例如：pair<int,char>p1(123,'A');
make_pair(v1, v2)	以v1和v2值创建一个新的pair对象，其元素类型分别是v1和v2的类型,例如：make_pair(123,'A');
p1 < p2	如果p1.first<p2.first或者!(p2.first < p1.first)&& p1.second<p2.second，则返回true
p1 == p2	如果两个pair对象的first和second成员依次相等，则这两个对象相等。
p1.first	返回p1中名为first的数据成员
p1.second	返回p1中名为second的数据成员

multimap的插入

multimap**不支持下标操作**
multimap允许一个键对应多个值
multimap用**insert()**添加新元素

例如，auther是一个书的作者与所写书的multimap容器，可以这样来添加2个元素：

```
multimap <string, string> author; //定义一个multimap
//插入第一个元素
author.insert(make_pair("Hemingway", "The Old Man and the Sea"));
//插入第二个元素
author.insert(make_pair("Hemingway", "The Sun Also Rises"));
//也可以这样
pair<string,string> p1;  //申明一个pair类型的变量p1
p1.first="Hemingway";  p1.second="To Be Or Not To Be"; //赋值
author.insert(p1);

pair<string,string>p2("Hemingway","Little Apple");//申明同时赋值
author.insert(p2);
```

multimap的删除

元素的删除erase():

1.以一个“键”作为参数的erase()将删除拥有该键的所有元素，并返回删除元素的个数。下面的语句删除作者是Hemingway的所有元素：

```
int i = author.erase("Hemingway");
```

2.带有一个或一对迭代器参数的erase()只删除指定位置的元素，返回void类型：

删除一个指定位置的元素

比如下列语句是删除找到的第一个作者为"Hemingway"的元素：

```
multimap<string,string>::iterator it=author.find("Hemingway");
author.erase(it);
```

删除一段元素

下列语句是删除作者"Hemingway"和作者"Moyan"间的所有元素

```
multimap<string,string>::iterator Beg,End;
Beg=author.find("Hemingway");
End=author.find("Moyan");
author.erase(Beg,End);//注意，区间是左闭右开[Beg,End)，作者为"Moyan"的书不会被删。
```

multimap的查找

元素的查找：

在multimap中，因为是自动按"key"排序，同一个键所关键的元素必然相邻存放，所以可用count统计出一个键的元素个数，然后用find逐个访问：

```
int cnt = author.count("Hemingway");
multimap <string, string>::iterator it = author.find("Hemingway");
for(int i = 0; i != cnt; i++, it++)
    cout << it->second << endl;  //输出Hemingway写的每一本书名
```

multimap的查找

在multimap中查找元素的更佳方案是用迭代器，需要用到两个函数：
lower_bound()，upper_bound()
当在容器中存在某个键时：
lower_bound()返回的迭代器指向该键的第一个元素，
upper_bound()返回的迭代器则指向大于该键的第一个元素，
这样形成一个左闭右开的区间（类似于begin,end）。

查找同一个"Hemingway"作者所有书的代码也可写成：

```
multimap <string, string>::iterator beg,end;
beg = author.lower_bound("Hemingway");
end = author.upper_bound("Hemingway");
while(beg != end) {
        cout << beg->second << endl;
        beg++;
}
```

当容器中不存在某个键时，则两者返回的迭代器相等，且**均指向依据元素的排序，该键应该插入的位置。**

vector

vector

vector，实际上就是个动态数组。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能,但在中间插入慢。它里面存储的元素可以是任意类型。

#include <vector>

常用函数：

push_back(e) - 在数组尾部添加一个元素e,数组长度自动+1

pop_back() - 删除数组最后一个元素,但无返回值,数组长度自动-1

front() - 得到数组第一个元素

back()-得到数组最后一个元素

size()-数组中元素的个数

empty()-判断数组是否为空

clear()-清空整个数组

vector

```
#include<vector>
using namespace std;
vector<int> vt;  //申明一个int类型的vector
int main()
{
        int n,i,x;
        cin>>n;
        for(i=1;i<=n;i++)
        {
                cin>>x;
                vt.push_back(x);        //将输入的数字x添加到数组尾部
        }

        vector<int>::iterator it;               //申明一个用于vector的迭代器
        for(it=vt.begin();it!=vt.end();it++)cout<<*it<<" ";    //从前往后依次输出数组元素
        cout<<endl;                                                     //输出1 2 5 7 9

        while(vt.size())  //从后往前依次输出数组元素，数组不为空也可写成while(!vt.empty())
        {
                cout<<vt.back()<<" ";     //输出数组尾部元素，输出9 7 5 3 1
                vt.pop_back();                  //删除输出尾部元素
        }
        return 0;
}
```

输入：

5

1 3 5 7 9

vector的下标操作

vector相当于是一个动态数组，它也可进行下标操作。

```
#include<vector>
using namespace std;
vector<int> vt;
int main()
{
        int n,x;
        cin>>n;
        for(int i=1;i<=n;i++)
        {
                cin>>x;
                vt.push_back(x);
        }
        for(int i=0;i<vt.size();i++)cout<<vt[i]<<" ";        //输出1 3 5 7 9
        cout<<endl;
        vt[3]=100;  //可用下标来修改已存在的元素
        vt[5]=99;   //错误，不能用下标来添加新元素，需要用push_back()
        for(i=vt.size()-1;i>=0;i--)cout<<vt[i]<<" ";          //输出?  9 100 5 3 1
        return 0;
}
```

输入：
5
1 3 5 7 9

vector用"[ ]"来随机访问已经存在的元素

vector存图1

vector可以方便表示一个邻接表。

```
#include<vector>
using namespace std;
vector<int> G[10];     //申明了一个vector数组，相当于有10个vector
int main()
{
    int i,n,m,x,y,k;
    cin>>n>>m;
    for(i=1;i<=m;i++)
    {
            cin>>x>>y;
            G[x].push_back(y);
    }
    cin>>k;
    for(i=0;i<G[k].size();i++)cout<<G[k][i]<<" ";
    return 0;    //输出跟k号点相连的点的编号
}
```

上面是用下标来访问的，也可迭代器的形式：

```
vector<int>::iterator it;
for(it=G[k].begin();it!=G[k].end();it++) cout<<*it<<" ";
```

输入格式：
第一行，两个整数n和m，分别代表有向图中点和边的数量
接下来m行，每行两个整数x，y 表示一条边从x出发指向y
接下来一行，一个整数k

样例输入：

```
4 6
1 3
2 3
1 4
2 1
3 1
2 4
2
```

样例输出？

3 1 4

vector存图2

```
vector<pair<int,int> > G[10]; //第一个int存长度，第二个int存编号
int main()                    //vector只能存单个数据，故用pair打包
{
    int i,n,m,x,y,z,k;
    cin>>n>>m;
    for(i=1;i<=m;i++)
    {
            cin>>x>>y>>z;
            G[x].push_back(make_pair(z,y));  //注意，打包后再加入
    }
    cin>>k;
    for(i=0;i<G[k].size();i++)
       cout<<G[k][i].second<<" "<<G[k][i].first<<endl;
    return 0;
}
```

输入格式：
第一行，两个整数n和m，分别代表有向图中点和边的数量
接下来m行，每行两个整数x, y, z表示一条边从x出发指向y, **长度为z**
接下来一行，一个整数k
样例输入：

```
4 6
1 3 5
2 3 7
1 4 1
2 1 3
3 1 9
2 4 11
2
```

样例输出？

```
3 7
1 3
4 11
```

上面是用下标来访问的，也可迭代器的形式：

```
vector<pair<int,int> >::iterator it;
for(it=G[k].begin();it!=G[k].end();it++)
    cout<<it->second<<" "<<it->first<<endl;
```

priority_queue

priority_queue 优先队列

priority_queue（优先队列）优先级最高的元素总是第一个出队。以某种排序准则（默认为值由大到小）自动管理队列中的元素。

#include <queue>
常用函数：
push(e) - 将元素加入队列
top() - 返回队首元素(默认队首元素是值队列中最大)的值
pop() - 将队首元素移除，但不返回值
size()-队列中元素的个数
empty()-判断队列是否为空

简单的说，在默认情况下，priority_queue就是**大根堆**，队首元素最大。

priority_queue 优先队列

```
#include <iostream>
#include <queue>
#include <ctime>
#include <cstdlib>
using namespace std;
int main(){
    priority_queue<int> PQ;                  //申明一个int类型的优先队列Q
  srand((unsigned)time(NULL));          //初始化随机种子
  for( int i= 1; i<=10; i++ )
              PQ.push( rand()%1000 );  //随机生成0到999的数字，入队
  while( !PQ.empty() )
      {                          //只要队列不为空， 依次取出队首元素
     cout << PQ.top() << endl;          //输出队首元素
     PQ.pop();                                  //删除队首元素
      }
      return 0;
}
```

priority_queue操作其他数据类型

使用格式为：

priority_queue<Type, Container, Functional>

Type 为堆中存储的数据类型
Container 为保存该型数据的容器
Functional 为堆中元素比较方式
Container 必须是用数组实现的容器，比如 vector, deque 但不能用 list.
STL里面默认用的是 vector，比较方式默认用 operator<，所以如果你把后面两个参数缺省的话，优先队列就是大根堆，堆顶元素最大。

```
//下面是普通用法，默认是大根堆
int main()
{
    priority_queue<int> q;
    for( int i= 1; i<=10; ++i ) q.push( rand()%100 );
    while( !q.empty() )
   {
        cout << q.top() << endl;
        q.pop();
    }
}
```

如果要用到小根堆，则要把三个参数都带进去。
STL里面定义了一个仿函数 **greater<>**，对于基本类型可以用这个仿函数声明小根堆

```
int main()
{
    priority_queue<int, vector<int>, greater<int> > q;
    for( int i=1; i<=10; ++i ) q.push( rand()%100 );
    while( !q.empty() )
    {
        cout << q.top() << endl;
        q.pop();
    }
}
```

例题：丑数

如果一个非负整数的因数是 2、3 或 5。我们称该数为丑数。
下面是最小的11个丑数(我们认为1是特殊的丑数)。1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ……
给你一个整数n，输出最小的n个丑数（n<=1000）。

```
#include <iostream>
#include <queue>
using namespace std;
typedef pair<long long, int> node;
int main()
{
    int n;
    cin>>n;
    priority_queue< node, vector<node>,greater<node> > Q;
    Q.push( make_pair(1, 2) );
    for (int i=1; i<=n; i++)
    {
        node tmp = Q.top();    Q.pop();
        switch(tmp.second)
        {
            case 2: Q.push( make_pair(tmp.first*2, 2) );
            case 3: Q.push( make_pair(tmp.first*3, 3) );
            case 5: Q.push( make_pair(tmp.first*5, 5) );
        }
        cout<<tmp.first<<" ";
    }
    return 0;
}
```

priority_queue操作其他数据类型

对于自定义数据类型，则必须自己**重载 operator<** 或者**自己写仿函数**

下面是重载operator<的方法：

```
struct Node
{
    int x, y;
    Node( int a, int b){ x=a; y=b; } //构造函数，用于给结构体类型赋初值
};

bool operator<( Node a, Node b )//重载Node类型的比较运算符,比较的第一关键字为Node中的x
{
    if( a.x== b.x ) return a.y> b.y;
    return a.x> b.x;
}

int main()
{
    priority_queue<Node> q;  //申明小根堆，重载比较运算符后，此处带一个参数就行
    for( int i= 1; i<=10; ++i )
    q.push( Node( rand()%100, rand()%100 ) );

    while( !q.empty() )
   {
        cout << q.top().x << ' ' << q.top().y << endl;
        q.pop();
    }
}
```

```
Node p= Node(3,5);
等价于
Node p;
p.x=3;
p.y=5;
```

```
等价于
Node p;
p.x=rand()%100;
p.y=rand()%100;
q.push(p);
```

priority_queue操作其他数据类型

对于自定义数据类型，则必须自己**重载 operator<** 或者**自己写仿函数**

下面是重载operator<的方法2：

```
struct Node
{
    int x, y;
    Node( int a, int b){ x=a; y=b; } //构造函数，用于给结构体类型赋初值

    bool operator<( const Node& b )const//重载Node类型的比较运算符,比较的第一关键字为Node中的x
    {
        if( x== b.x ) return y> b.y;
        return x> b.x;
    }

};


int main()
{
    priority_queue<Node> q;  //申明小根堆，重载比较运算符后，此处带一个参数就行
    for( int i= 1; i<=10; ++i )
    q.push( Node( rand()%100, rand()%100 ) );

    while( !q.empty() )
   {
        cout << q.top().x << ' ' << q.top().y << endl;
        q.pop();
    }
}
```

Node p= Node(3,5);
等价于
Node p;
p.x=3;
p.y=5;

等价于
Node p;
p.x=rand()%100;
p.y=rand()%100;
q.push(p);

priority_queue操作其他数据类型

对于自定义数据类型，则必须自己**重载 operator<** 或者**自己写仿函数**

下面是重载 自己写仿函数的模板：

```
struct Node
{
    int x, y;
    Node( int a, int b){ x=a; y=b; } //构造函数，用于给结构体类型赋初值
};
struct cmp
{
    bool operator() ( Node a, Node b )
    {
        if( a.x== b.x ) return a.y> b.y;
        return a.x> b.x;
    }
};
int main()
{
    priority_queue<Node,vector<Node>,cmp> q;  //申明小根堆
    for( int i= 1; i<=10; ++i )
    q.push( Node( rand()%100, rand()%100 ) );

    while( !q.empty() )
   {
        cout << q.top().x << ' ' << q.top().y << endl;
        q.pop();
    }
}
```

priority_queue 优先队列

```
//dijkstra+heap, 邻接表，存图，申明N个vector，每个点对应一个。
vector<pair<int,int> > G[N]; //第一个int存到起点的距离，第二个int存编号
int i,X,distX,dis[N];
priority_queue<pair<int,int> >heap;
vector<pair<int,int> >::iterator it;
for(i=0;i<N;i++)dis[i]=inf;
dis[s]=0;
heap.push(make_pair(-dis[s],s));
//默认的priority_queue是大根堆，所以存的-dis[]进去，当小根堆用
while(!heap.empty())
{
        X=heap.top().second;
        distX=-heap.top().first;
        heap.pop();
        if(distX!=dis[X])continue;
        for(it=G[X].begin();it!=G[X].end();it++)
        if(dis[it->second]>dis[X]+(it->first))
         {
                dis[it->second]=dis[X]+(it->first);
                heap.push(make_pair(-dis[it->second],it->second));
         }
}
```

完整的dijkstra+heap代码 nkoj1120

```
.....
vector<pair<int,int> >G[101];  //G[x]存x号点的邻接情况，第一个int存距离，第二个int存编号
priority_queue<pair<int,int> >Heap;  //堆，第一个int存距离，第二个int存编号
int n,dis[101];

int main()
{
        int m,i,x,y,z;
        scanf("%d%d",&n,&m);
        for(i=1;i<=m;i++)
        {
                scanf("%d%d%d",&x,&y,&z);
                G[x].push_back(make_pair(z,y));  //建图
        }
        scanf("%d%d",&x,&y);
        dijkstra_heap(x);
        printf("%d",dis[y]);
        return 0;
}
```

完整的dijkstra+heap代码 nkoj1120

```
void dijkstra_heap(int s)
{
        int i,x,y,len,MinDis;
        vector<pair<int,int> >::iterator it;

        for(i=1;i<=n;i++)dis[i]=inf;
        dis[s]=0;
        Heap.push(make_pair(-dis[s],s));

        while(!Heap.empty())
        {
                x=Heap.top().second;
                MinDis=-Heap.top().first;
                Heap.pop();
                if(MinDis!=dis[x])continue;
                for(it=G[x].begin();it!=G[x].end();it++)
                {
                        y=it->second;
                        len=it->first;
                        if(dis[y]>dis[x]+len)
                        {
                            dis[y]=dis[x]+len;
                            Heap.push(make_pair(-dis[y],y));
                        }
                }
        }
}
```

deque

deque 双端队列

deque（双端队列），队列，但在队首和队尾都可以进行插入和删除。一般用于模拟单调队列。所有适用于vector的操作都适用于deque

#include <deque>
常用函数：
front() - 返回队首元素的值，可读可写，但不移除
back() - 返回队尾元素的值，可读可写，但不移除
push_back(e) - 将元素e从队尾加入队列
push_front(e) - 将元素e从队首加入队列
pop()_front( ) - 将队首元素移除，但不返回值
pop()_back( ) - 将队尾元素移除，但不返回值
size()-队列中元素的个数
empty()-判断队列是否为空

queue 队列

```
//单调队列例题 nkoj2152 "滑动窗口"
#include<deque>
deque<pair<int,int>>Q;
//求长度为k的窗口中的最小值
for(i=1;i<=n;i++)
{
                    cin>>x;
                    while(Q.back()>x&&Q.size())Q.pop_back();
                    Q.puch_back(make_pair(x,i));
                    if(Q.front().second<i-k+1)Q.pop_front();
                    if(i>=k)cout<<Q.front()<<" ";

}
```

set

set容器

- set容器只是单纯键的集合，相当于数学意义上的“集合”
- set容器支持大部分的map操作，但有2个例外：

  set不支持下标操作

  set的元素类型不是pair，而是与键类型相同的类型

- set的定义，仍然是三种形式：

```
set<T> s;                  //定义类型是T的set，无元素
set<T> s2(s1);             //复制s1中的元素
set<T>s3(beg, end)   //用一对迭代器复制元素
```

set容器

- set在复制元素时会去掉重复的
- vector <int> v;

  for(int i=0; i<10; i++)v.push_back(i),v.push_back(i);　　//同一个数被添加2次

  set<int> s(v.begin(), v.end());　　//复制v中的元素

  cout << v.size();　　//20个

  cout << s.size();　　//10个

- 插入元素，两种方式：

  一次插入单个元素：

  set <string> s2;　　//空set

  s2.insert("the");　　//有1个元素

  s2.insert("best");　　//有2个元素

  用一对迭代器插入一段元素，也要去重：

  set<int>s;

  s.insert(v.begin(), v.end());

set容器

- 从set获取元素

set不提供下标操作，要访问元素，可用find。要判断某个元素是否存在，可用count运算。

multiset可用lower_bound和upper_bound运算。

- 对于前述有10个int型元素的set有：

```
s.find(1)                    //返回指向键值为1的迭代器
s.find(100)                  //返回指向end()的迭代器
s.count(1)                   //返回1
s.count(100)                 //返回0
```

- set中的元素只可读，不可写：

```
set<int>::iterator it = s.find(1);
*it = 100;                   //错误！
```

set/map容器自定义比较函数

- 在关联容器中，如果希望自定义键的大小关系，则可以定义一个仿函数，然后在定义容器时指定采用该仿函数比较键的大小。这种做法在STL中通用。

```
#include<string>
#include<iostream>
#include<set>
using namespace std;
struct cmp {                                    //自定义比较大小的仿函数
    bool operator () (string s1, string s2)  {
        return s1 > s2;
    }
};
```

set/map容器中自定义比较函数

```
int main() {
    set<string> s;
    s.insert(string("cpp"));        s.insert(string("apple"));        s.insert(string("english"));
    cout<<"s1:"<<endl;
    set<string,cmp>::iterator it;
    for( it = s.begin(); it != s.end(); it++)   cout<<*it<<" ";    //输出：apple cpp english
    cout<<endl<<"s2:"<<endl;
    set<string, cmp> s2(s.begin(), s.end());   //复制s中的元素，但重排键顺序
    for(it = s2.begin(); it != s2.end(); it++)  cout<<*it<<" ";  //输出: english cpp apple
    cout<<endl<<endl;
    return 0;
}
```

set容器专有的集合操作

- 在算法库中有专为set服务的集合操作函数

函数	说明
set_union()	求两个集合的并
set_intersection()	求两个集合的交
set_difference()	求第一个相对于第二个的差集
set_symmetric_difference()	第一个集相对于第二个的差集并上第二个相当于第一个的差集

•这几个函数的格式类似，以set_union为例说明：

```
s3.last set_union( s1.first, s1. last,
                   s2.first, s2.last,
                   s3.first );
```

函数接受5个参数，前4个分别是两个集合的起止点迭代器，s3.first是目标集合的起始迭代器

返回值是目标集合终止迭代器

set专用的集合运算

```
#include<iostream>#include<set>
#include<algorithm>
using namespace std;
int main() {
        int a1[] = {1, 3, 5, 7},  a2[] = {2, 3, 4};
        set<int> s1(a1, a1+4), s2(a2, a2+3) ,  s3;
        //inserter是一个STL函数，表示了要插入的容器及其插入点迭代器
        set_union(s1.begin(), s1.end(), s2.begin(), s2.end(), inserter(s3, s3.begin()));  //求并集
        set<int>::iterator it;
        for(it = s3.begin(); it != s3.end(); it++)              cout << *it << ' ';
        cout << endl;
        s3.clear();  //清空s3
        set_intersection(s1.begin(), s1.end(), s2.begin(), s2.end(), inserter(s3, s3.begin())); //求交集
        for(it = s3.begin(); it != s3.end(); it++)              cout << *it << ' ';
        cout << endl;
        return 0;
}
```

multiset

支持同一个键多次出现的set类型

STL习题

NKOJ1823