STL简介

fsdhmbb

STL 简介

fsdhmbb

- STL 容器（适配器）

可以高效存储一些数据，只介绍其中几种。

如 vector、deque、list、forward_list、set、multiset、map、unordered_map、stack、queue、priority_queue 等。

- STL 算法

STL 容器\STL 容器适配器

fsdhmbb

- vector 动态数组
- **可变长度**的数组，能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。
- 优势：
- vector 可以动态分配内存
- vector 重写了比较运算符及赋值运算符
- vector 便利的初始化

STL 容器\STL 容器适配器

vector

- 声明：vector <类型> 名字
- 如：

```
vector <int> vec;
```

- 在插入元素时可以使用：

```
vec.push_back(1);//在数组尾部插入元素 1
```

- 访问时可以直接使用其下标（类似静态数组），如：

```
vec[0];
```

- 或者：

```
vec.front();//访问数组首元素
vec.back();//访问数组末尾元素
vec.at(0);//访问数组下标为0的元素，会判断溢出
```

STL 容器\STL 容器适配器

vector

- 迭代器
- 可以自行了解迭代器的定义。

> - `begin()` 返回指向首元素的迭代器，其中 `*begin = front`。
> - `end()` 返回指向数组尾端占位符的迭代器，注意是没有元素的。
> - `rbegin() `返回指向逆向数组的首元素的逆向迭代器，可以理解为正向容器的末元素。
> - `rend()` 返回指向逆向数组末元素后一位置的迭代器，对应容器首的前一个位置，没有元素。

STL 容器\STL 容器适配器

vector

- 长度容量相关
- 判断容器是否为空，返回值为 `bool` 型：

```
vec.empty();//true为空，false为非空
```

- 返回容器长度（元素数量）：

```
vec.size();//如果已经存储了5个元素，则返回5
```

- 其他的如：

```
vec.resize();//改变 vector 的长度，多退少补。补充元素可以由参数指定。
vec.max_size();//返回容器的最大可能长度。
vec.reserve();//使得 vector 预留一定的内存空间，避免不必要的内存拷贝。
vec.capacity();//返回容器的容量，即不发生拷贝的情况下容器的长度上限。
vec.shrink_to_fit();//使得 vector 的容量与长度一致，多退但不会少。
```

STL 容器\STL 容器适配器

vector

- 元素增删及修改
- 清除所有元素：

```
vec.clear();//复杂度是常数的
```

- 在容器中插入与删除元素：

```
vec.insert();//注意复杂度是线性而非常数的
vec.erase();//注意复杂度是线性而非常数的
```

- 在容器尾插入与删除元素：

```
vec.push_back();//在末尾插入一个元素
vec.pop_back();//删除末尾元素
```

- 与另一个容器进行交换：

```
vec.swap();//与指定的另一容器进行交换
```

STL 容器\STL 容器适配器

vector

- 常用实例1：

```
vector <int> vec;
vector <int> demo;
vec.push_back(2);//在数组尾部插入元素 2
cout<<vec[0];//访问数组下标为0元素:2
vec[0]=1;//将数组下标为0的元素更改为 1
cout<<vec.front();//访问数组首元素:1
vec.push_back(2);//在数组尾部插入元素 2
cout<<vec.back();//访问数组末尾元素:2
cout<<vec.at(0);//访问数组下标为0的元素，会判断溢出
//cout<<vec.at(4);//抛出溢出异常
cout<<*vec.begin();//等价于vec.front()
vec.insert(vec.begin()+1,3);//在数组下标为1的位置插入元素 3
for(int i=0;i<vec.size();i++) cout<<vec[i];
vec.swap(demo);//交换vec与demo中的元素
for(auto i:demo) cout<<i;//顺序访问数组中的元素，auto会自动判断元素类型
```

- 输出：

21211132132

STL 容器\STL 容器适配器

vector

- 常用实例2：

```
struct test{
    string name;
    int ccf;
};
signed main()
{
    vector <test> kkk;
    kkk.push_back(test{"iloveccf",114514});
    kkk.push_back(test{"Nocommander",1919810});
    for(auto i:kkk) cout<<i.ccf<<" "<<i.name<<endl;
}
```

- 输出：

```
114514 iloveccf
1919810 Nocommander
```

STL 容器\STL 容器适配器

fsdhmbb

- deque 双端队列
- 双端队列，能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。
- 优势：
- deque 可以支持从**容器首**或容器末插入和删除元素，以及随机访问。
- 劣势：

> `deque` 的实现是很慢的，有着**很大的时空常数**。
> 此外，由于 `queue()` 和 `stack()` 也是依赖于 `deque` 的，因此在使用它们的时候也应当要注意。
> 如果**不要求支持随机访问队列下标**，那么 `deque` 可以用 `list` 快捷地代替。

题目：B3656

STL 容器\STL 容器适配器

deque

- 声明：deque <类型> 名字
- 如：

```
deque <int> deq;
```

- 在插入元素时可以使用：

```
deq.push_back(1);//在队尾部插入元素 1
deq.push_front(1);//在队首部插入元素 1
```

- 访问时可以直接使用其下标，如：

```
deq[0];
```

- 或者：

```
deq.front();//访问队首元素
deq.back();//访问队尾元素
deq.at(0);//访问容器中下标为0的元素，会判断溢出
```

STL 容器\STL 容器适配器

deque

- 迭代器
- 与 `vector` 的迭代器一致。
- 长度&容量
- 与 `vector` 一致，但是没有 `reserve()` 和 `capacity()` 函数。
- 元素增删及修改
- 与 `vector` 一致，并额外有向队列头部增加元素的函数。
- 在容器首插入与删除元素：

```
vec.push_front();//在容器首插入一个元素
vec.pop_front();//删除容器首元素
```

- 常用实例与 `vector` 基本一致。

STL 容器\STL 容器适配器

fsdhmbb

- list 双向链表\forward_list 单向链表
- 双（单）向链表，能够提供线性复杂度的随机访问，以及常数复杂度的插入和删除。
- 优势：
- 时空开销较小，单向链表的空间开销更小。
- 劣势：
- 不支持基于下标的随机访问，且随机访问的复杂度为线性。

题目：B3656

STL 容器\STL 容器适配器

list

- 声明：list <类型> 名字
- 如：

```
list <int> ls;
```

- 在插入元素时可以使用：

```
ls.push_back(1);//在队尾部插入元素 1
ls.push_front(1);//在队首部插入元素 1
```

- 访问时不可以直接使用其下标：

```
ls.front();//访问队首元素
ls.back();//访问队尾元素
```

- `list` 类型还提供了一些针对其特性实现的 STL 算法函数，这里不多介绍。
- `forward_list` 的使用方法与 `list` 几乎一致，但是迭代器只有单向的，因此其具体用法不作详细介绍。

STL 容器\STL 容器适配器

fsdhmbb

• set 集合\multiset 可重集合

• 集合，不会出现值相同的元素（可重集合就顾名思义了）。通过红黑树（一种平衡树）实现。非常适合处理需要同时兼顾**查找、插入与删除**的情况。

• 由于 `multiset` 与 `set` 用法基本相同，接下来只介绍 `set`。

STL 容器\STL 容器适配器

set

- 声明：set <类型> 名字
- 如：

```
set <int> se;
```

- 在插入元素时可以使用：

```
se.insert();//对数复杂度，当容器中没有等价元素的时候，将元素插入到 set 中。
```

- 在删除元素时可以使用：

```
se.erase(x);//对数复杂度，删除值为 x 的所有元素
se.erase(pos);//对数复杂度，删除迭代器为 pos 的元素
se.clear();//清空 set
```

- `set` 的迭代器仍有 `begin()`、`end()`、`rbegin()`、`rend()`。

STL 容器\STL 容器适配器

set

- 查找操作
- 如：

```
se.count(x);//返回 set 内元素为 x 的元素数量
se.find(x);//若有值为 x 的元素时会返回该元素的迭代器，否则返回 end()
se.lower_bound(x);//对数复杂度，返回指向首个不小于给定的元素 x 的迭代器。如果不存在这样的元素，返回 end()
se.upper_bound(x);//对数复杂度，返回指向首个大于给定的元素 x 的迭代器。如果不存在这样的元素，返回 end()
se.empty();//返回容器是否为空
se.size();//返回容器内元素个数
```

> `set` 自带的 `lower_bound` 和 `upper_bound` 的时间复杂度为 $O(log\ n)$。
> 但使用 `algorithm` 库中的 `lower_bound` 和 `upper_bou`函数对 `set` 中的元素进行查询，时间复杂度为 $O(n)$。
> `set` 没有提供自带的 `nth_element`。使用 `algorithm` 库中的 `nth_element` 查找第 k 大的元素时间复杂度为 $O(n)$。
> 如果需要实现平衡二叉树所具备的 $O(log\ n)$ 查找第 k 大元素的功能，需要自己手写平衡二叉树或权值线段树，或者选择使用 `pb_ds` 库中的平衡二叉树。

STL 容器\STL 容器适配器

set

• 在贪心中的使用

• 在贪心算法中经常会需要出现类似 **找出并删除 最小的 大于等于某个值的元素。**这种操作能轻松地通过 `set` 来完成。

```
// 现存可用的元素
set<int> se;
// 需要大于等于的值
int x;

// 查找最小的大于等于x的元素
auto it = se.lower_bound(x);
if (it == se.end()) {
  // 不存在这样的元素，则进行相应操作……
} else {
  // 找到了这样的元素，将其从现存可用元素中移除
  se.erase(it);
  // 进行相应操作……
}
```

STL 容器\STL 容器适配器

fsdhmbb

• map 映射\unordered_map 哈希

• `map` 是有序键值对容器，它的元素的键是唯一的。搜索、移除和插入操作拥有对数复杂度。`map` 通常实现为红黑树。

• `unordered_map` 是无序键值对容器，它的元素的键是唯一的。`unordered_map` 实现为哈希。

• 注意，`unordered_map` **在平均情况下** 大多数操作（包括查找，插入，删除）都能在常数时间复杂度内完成，但是由于存在哈希冲突，实际上如果出题人有意愿（事实上几乎为 100%），他们都可以通过制造大量哈希冲突使得一些操作的时间复杂度会与容器大小成线性关系。**所以请谨慎使用任何无序容器！**

• 由于 `unordered_map` 与 `map` 用法基本相同，接下来只介绍 `map`。

STL 容器\STL 容器适配器

map

- 声明：map <下标类型，值类型> 名字
- 如：

```
map <string,int> mp;//下标类型可以任意选取!
```

- 可以直接使用下标进行访问或插入操作：

```
mp["Alan"]=100;//可以直接通过下标进行赋值操作
```

- 也可以：

```
mp.insert(pair<string,int>("Alan",100));//pair 用于将两个变量关联在一起，组成一个对;将这样的映射关系使用 insert 插入 map
```

- 在删除元素时可以使用：

```
mp.erase(x);//删除下标为 x 的所有元素
mp.erase(pos);//删除迭代器为 pos 的元素
mp.clear();//清空 map
```

STL 容器\STL 容器适配器

map

- 查询操作
- 如：

```
mp.count(x);//返回容器内下标为 x 的元素数量
mp.find(x);//若容器内存在下标为 x 的元素，会返回该元素的迭代器；否则返回 end()
mp.lower_bound(x);//返回指向首个不小于给定键的 元素的迭代器
mp.upper_bound(x);//返回指向首个大于给定键的 元素的迭代器。若容器内所有元素均小于或等于给定键，返回 end()
mp.empty();//返回容器是否为空
mp.size();//返回容器内元素个数
```

- 在搜索中，我们有时需要存储一些较为复杂的状态（如坐标，无法离散化的数值，字符串等）以及与之有关的答案（如到达此状态的最小步数）。`map` 可以用来实现此功能。其中的键是状态，而值是与之相关的答案。

STL 容器\STL 容器适配器

fsdhmbb

- 容器适配器其实并不是容器。它们不具有容器的某些特点（如：有迭代器、有 `clear()` 函数……）。

> 「适配器是使一种事物的行为类似于另外一种事物行为的一种机制」，适配器对容器进行包装，使其表现出另外一种行为。

- **栈**(`stack`) 后进先出 (LIFO) 的容器，默认是对双端队列（`deque`）的包装。
- **队列**(`queue`) 先进先出 (FIFO) 的容器，默认是对双端队列（`deque`）的包装。
- **优先队列**(`priority_queue`) 元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列，默认是对向量（`vector`）的包装。

STL 容器\STL 容器适配器

fsdhmbb

- stack 栈

- 后进先出的容器适配器，仅支持查询或删除最后一个加入的元素（栈顶元素），不支持随机访问，且为了保证数据的严格有序性，不支持迭代器。

STL 容器\STL 容器适配器

stack

- 声明：stack <类型> 名字
- 如：

```
stack <int> sk;
```

- 只能在栈顶插入、删除、访问元素

```
sk.push(x);//在栈顶插入元素 x
sk.pop();//删除栈顶元素
sk.top();//访问栈顶元素（如果栈为空，此处会出错）
```

- 以及：

```
sk.size();//查询容器中的元素数量
sk.empty();//询问容器是否为空
```

STL 容器\STL 容器适配器

fsdhmbb

• queue 队列

• 先进先出的容器适配器，仅支持查询或删除第一个加入的元素（队首元素），不支持随机访问，且为了保证数据的严格有序性，不支持迭代器。

STL 容器\STL 容器适配器

queue

- 声明：queue <类型> 名字
- 如：

```
queue <int> qu;
```

- 只能在队尾插入，队首删除元素

```
qu.push(x);//在队尾插入元素 x
qu.pop();//删除队首元素
qu.front();//访问队首元素（如果队列为空，此处会出错）
qu.back();//访问队尾元素（如果队列为空，此处会出错）
```

- 以及：

```
qu.size();//查询容器中的元素数量
qu.empty();//询问容器是否为空
```

STL 容器\STL 容器适配器

fsdhmbb

• priority_queue 优先队列

• 使得队列中满足某种优先级条件的数据先出列，如：使得队列中最大的数先出列（大根堆）或使得队列中最小的数先出列（小根堆）。

STL 容器\STL 容器适配器

priority_queue

• 声明：

priority_queue <数据类型> 名字 默认为大根堆

priority_queue <数据类型,底层容器> 名字 默认为大根堆

priority_queue <数据类型,底层容器,比较类型> 名字 可以指定大\小根堆

• 如：

```
priority_queue <int> pq;//大根堆
priority_queue <int,vector<int> > pq;//大根堆，指定底层容器（默认为 vector）
priority_queue <int,vector<int>,greater<int> > pq;//greater 小根堆，less 大根堆
```

STL 容器\STL 容器适配器

priority_queue

- 插入、查询、删除元素：

```
pq.push(x);//插入元素，并对底层容器排序，对数复杂度
pq.pop();//删除堆顶元素（此时优先队列不能为空），对数复杂度
pq.top();//访问堆顶元素（此时优先队列不能为空）
```

- 以及：

```
pq.size();//查询容器中的元素数量
pq.empty();//询问容器是否为空
```

STL 算法

fsdhmbb

- 介绍：reverse、unique、shuffle、sort、stable_sort、nth_element、lower_bound、upper_bound

STL 算法

reverse

- 用于翻转数组、字符串
- 用法：

```
reverse(v.begin(),v.end());
reverse(v+begin,v+end);
```

- 实例：

```
string str="ABCDEFG";//vector、deque与string方法相似
int len=str.size();//获取字符串长度
reverse(str.begin(),str.end());//反转字符串
cout<<str<<endl;//GFEDCBA
reverse(str.begin(),str.begin()+len);//反转字符串
cout<<str<<endl;//ABCDEFG
int b[6]={1,2,3,4,5};
reverse(b,b+5);//反转数组
for(int i=0;i<5;i++)
{
    cout<<b[i];//54321
}
```

STL 算法

sort\stable_sort

- 用于排序，其中 stable_sort 为稳定排序，用法：

```
sort(v.begin(),v.end(),cmp);//cmp 为自定义的比较函数
sort(a+begin,a+end,cmp);//cmp 为自定义的比较函数
```

- 实例：

```
bool cmp(int a,int b){return a>b;/*从大到小排序*/}
signed main()
{
    int a[6]={2,3,4,5,1};
    sort(a,a+5);//默认从小到大 12345
    sort(a,a+5,cmp);//自定义cmp 54321
    vector <int> vec{2,1,3};
    sort(vec.begin(),vec.end());//默认从小到大 123
    sort(vec.begin(),vec.begin()+3);//默认从小到大 123
    sort(vec.begin(),vec.end(),cmp);//自定义cmp 321
    sort(vec.begin(),vec.begin()+3,greater<int>());//从大到小 321
}
```

STL 算法

unique

• 用于去重，去除容器中**相邻**的重复元素，返回值为指向 **去重后** 容器结尾的迭代器，原容器大小不变。与 sort 结合使用可以实现完整容器去重。用法：

```
unique(v.begin(),v.end());
unique(a+begin,a+end);
```

• 实例：

```
int a[6]={2,3,4,1,1};
sort(a,a+5);
int c=unique(a,a+5)-a;//从a[0]至a[c]即为去重后的数列
vector <int> vec{1,1,3};
sort(vec.begin(),vec.end());
vec.erase(unique(vec.begin(),vec.end()),vec.end());
```

STL 算法

nth_element

- 按指定范围进行分类，即找出序列中第 n 大的元素，使其左边均为小于它的数，右边均为大于它的数。用法：

```
nth_element(v.begin(),v.begin()+nth,v.end(),cmp);//cmp 为自定义的比较函数
nth_element(a+begin,a+nth,a+end,cmp);//cmp 为自定义的比较函数
```

- 实例：

```
bool cmp(int a,int b)
{
    return a>b;//从大到小排序
}
signed main()
{
    int a[10]={4,7,6,9,1,8,2,3,5};
    nth_element(a,a+2,a+9);//求出a[0]~a[8]共9个数中第3小的数
    cout<<a[2]<<endl;//会被放在第3（下标2）的位置
    nth_element(a,a+2,a+9,greater<int>());//求出a[0]~a[8]共9个数中第3大的数
    nth_element(a,a+2,a+9,cmp);//还可以自定义cmp
    vector <int> vec{4,7,6,9,1,8,2,3,5};
    nth_element(vec.begin(),vec.begin()+2,vec.end());//vector用法类似
}
```

STL 算法

shuffle

- 使用一个随机数产生器来打乱容器元素的顺序。用法：

```
shuffle(v.begin(),v.end(),rng);//rng为指定的随机数产生器
shuffle(a+begin,a+end,rng);//rng为指定的随机数产生器
```

- 实例：

```
mt19937 rng(time(0));//一种随机数产生器
int a[6]={1,2,3,4,5};
shuffle(a,a+5,rng);
vector <int> vec{1,2,3,4,5};
shuffle(vec.begin(),vec.end(),rng);
```

STL 算法

lower_bound\upper_bound

• 在一个**有序序列**中进行二分查找，返回指向第一个 **大于等于（lower_bound）\大于（upper_bound）** 的元素的位置的迭代器。如果不存在这样的元素，则返回 end() 。用法：

```
lower_bound(v.begin(),v.end(),x);//在范围内寻找x
lower_bound(a+begin,a+end,x);//在范围内寻找x
```

• 实例：

```
vector <int> vec{1,2,3,4,5};
sort(vec.begin(),vec.begin()+5);
int pos=lower_bound(vec.begin(),vec.end(),2)-vec.begin();
cout<<pos<<vec[pos];
pos=upper_bound(vec.begin(),vec.end(),2)-vec.begin();
cout<<pos<<vec[pos];
int a[6]={1,2,3,4,5};
sort(a,a+5);
pos=lower_bound(a,a+5,2)-a;
cout<<pos<<a[pos];
pos=upper_bound(a,a+5,2)-a;
cout<<pos<<a[pos];
```