信息学竞赛

STL编程及应用

Standard Template Library

华南师大讲稿

常用STL类的使用

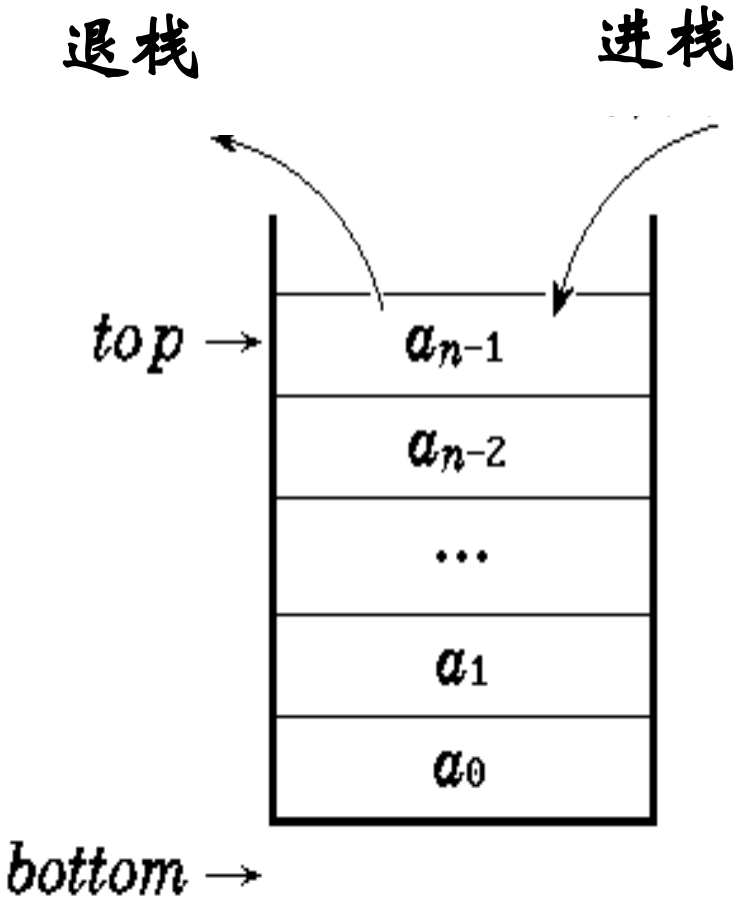
- 栈 stack
- 队列 queue
- 优先队列 priority_queue

栈(stack)

定义:只允许在一端插入和删除的线性表

栈顶(*top*):
允许插入和删除的一端

特点
后进先出 (*LIFO*)



使用方法

1.参考网站:【英文】
http://msdn.microsoft.com/en-us/library/h9sh48d5(VS.80).aspx

2.引用头文件

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

stack class

成员函数:

empty
Tests if the stack is empty.

pop
Removes the element from the top of the stack.

push
Adds an element to the top of the stack.

size
Returns the number of elements in the stack.

top
Returns a reference to an element at the top of the stack.

使用方法

1.参考网站:【中文】
msdn.microsoft.com/zh-cn/library/vstudio/56fa1zk5(v=vs.110).aspx

2.引用头文件

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

stack class

成员函数:

empty	测试stack是否为空。
pop	从stack的顶部移除元素。
push	将元素添加到stack顶部。
size	返回stack中的元素个数。
top	返回stack顶部元素。

使用例子

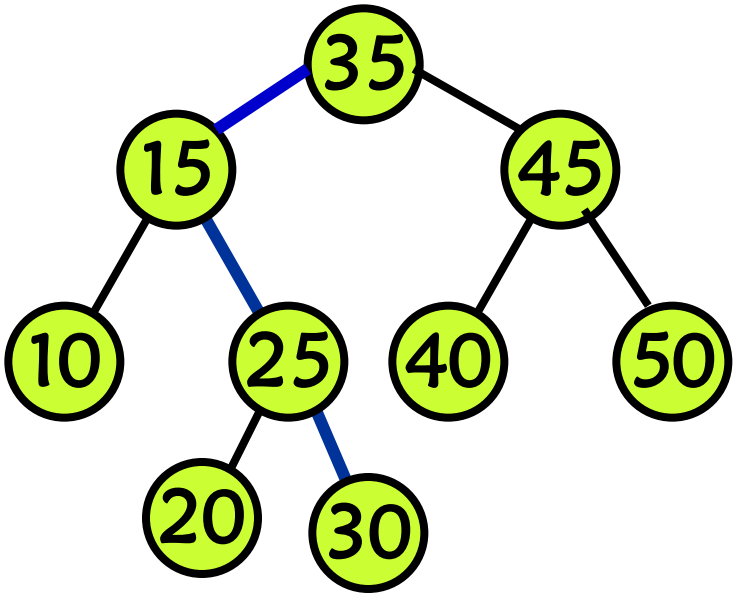
```
#include <stack>
#include <iostream>
using namespace std;

int main( )
{  stack <int> s1;

   s1.push( 10 );
   s1.push( 20 );
   s1.push( 30 );

   int  i;
   i = s1.top( );
   cout << "The element at the top of the stack is " << i << "." << endl;

   s1.pop( );
   i = s1.top( );
   cout << " The element at the top of the stack is "<< i << "." << endl;
}
```

运行结果：

The element at the top of the stack is 30.

The element at the top of the stack is 20.

应用例子：二叉树前序遍历

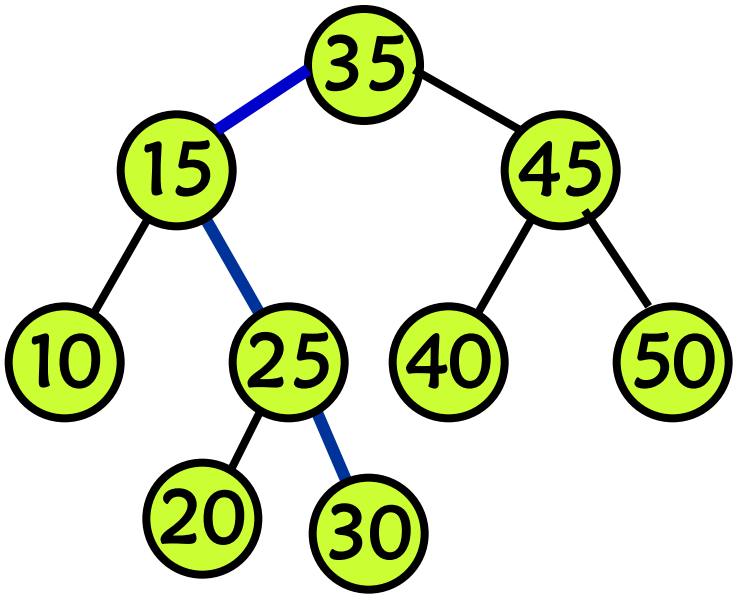


存储结构

```
struct node
{
    int data;
    struct node *child[2];
};
```

前序遍历（深度遍历）非递归算法

```
void DFS(struct node *root)
{
    stack<struct node *> st;

    while((!st.empty( ) || root!=0) )
    {
        while(root!=0)
        {
           cout<<root->data<<"  ";
           st.push(root);
           root=root->child[0];
         }
      if(!st.empty( )) {  root=st.top();  st.pop(); root=root->child[1]; }
    }
}
```



队列（Queue）

- 定义
  - 队列是只允许在一端删除，在另一端插入的线性表
  - 队头(*front*):允许删除的一端
  - 队尾(*rear*):允许插入的一端
- 特性
  - 先进先出(*FIFO, First In First Out*)

使用方法

- 1.参考网站:【英文】

http://msdn.microsoft.com/en-us/library/s1b6tszt(VS.80).aspx.

- 2.引用头文件

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

Queue class

- 成员函数:

back()

Returns a reference to the last and most recently added element at the back of the queue.

empty()

Tests if the queue is empty.

front()

Returns a reference to the first element at the front of the queue.

pop()

Removes an element from the front of the queue.

push()

Adds an element to the back of the queue.

size()

Returns the number of elements in the queue.

使用方法

- 1.参考网站:【中文】

http://msdn.microsoft.com/zh-cn/library/vstudio/s23s3de6%28v=vs.110%29.aspx

- 2.引用头文件

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

Queue class

- 成员函数:

back	返回队列中最后和最近添加的元素。
empty	测试queue是否为空。
front	返回队列中的第一个元素。
pop	从 queue的前面移除元素。
push	将元素添加到 queue中。
size	返回队列的元素个数。

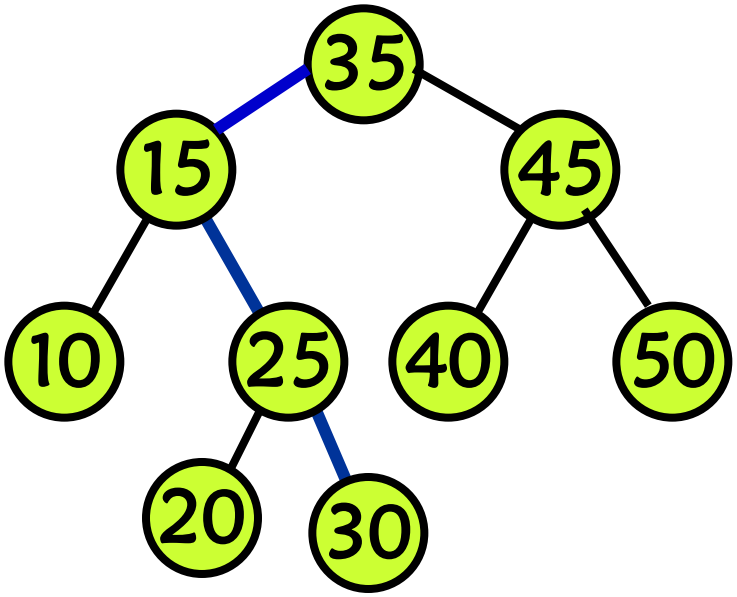
使用例子

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

int main( )
{  queue <int> q1;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   int i;

   i = q1.front( );
   cout << "The element at the front of the queue is "<< i << "." << endl;

   q1.pop( );

   i = q1. front ( );
   cout << "After a pop, the front of the queue is "   << i << "." << endl;
}
```

运行结果：

```
The element at the front of the queue is 10.
After a pop, the front of the queue is 20.
```

应用：二叉树的层次遍历

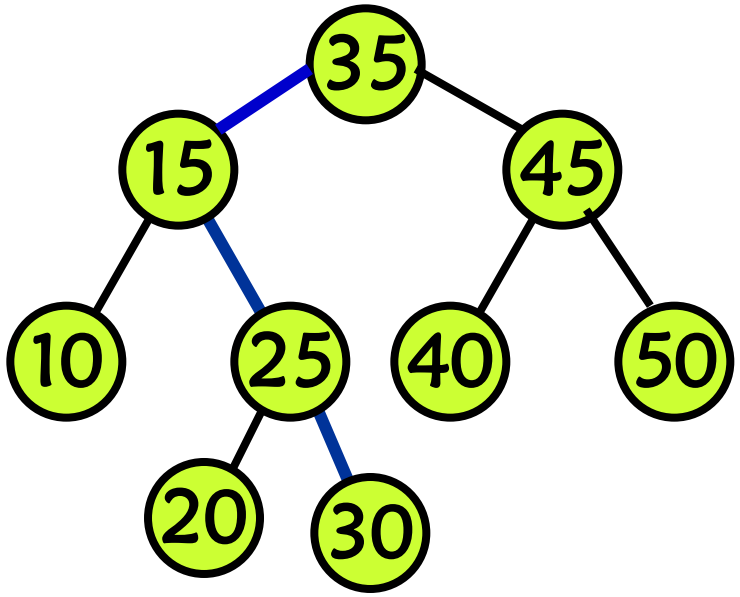


存储结构

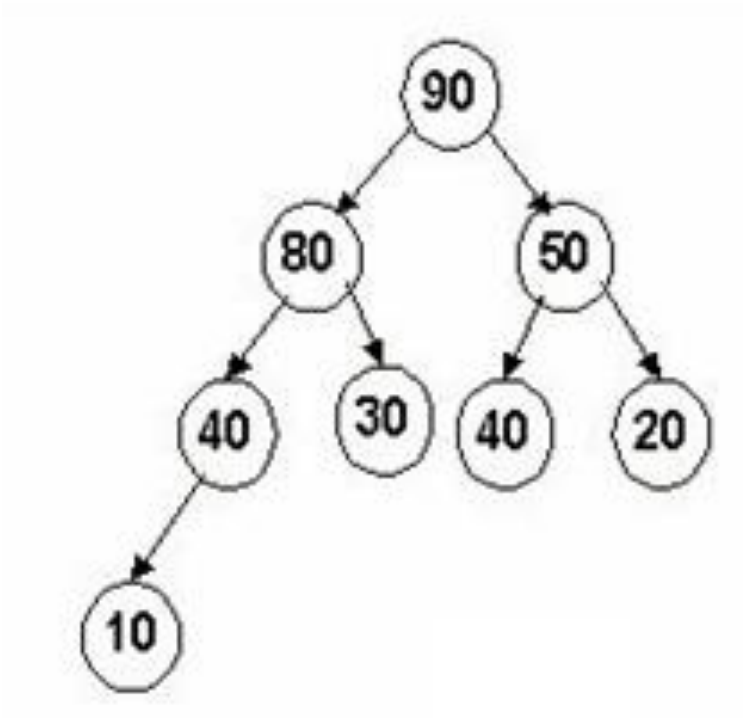
```
struct node
{
    int data;
     struct node *child[2];
};
```

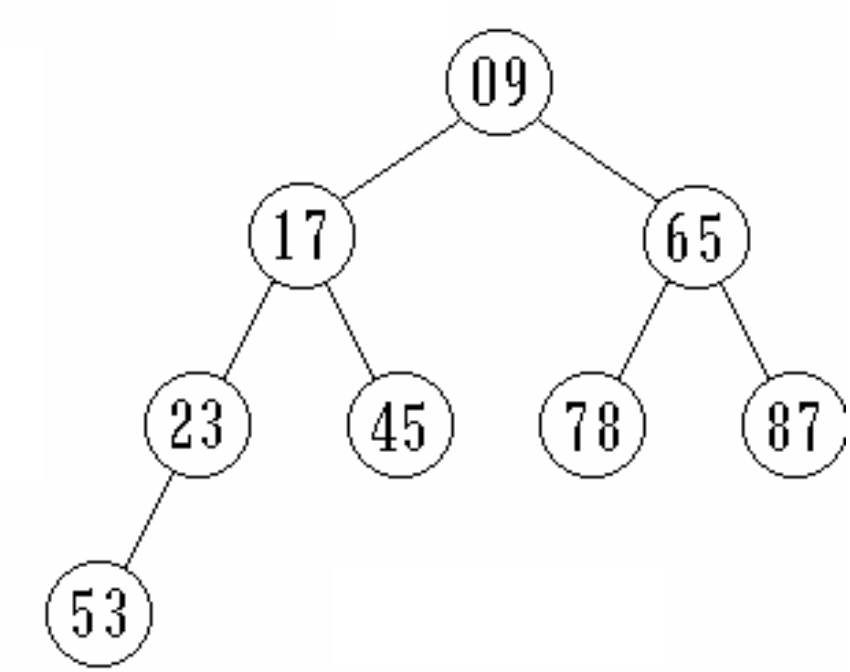
层次遍历（广度优先遍历）算法

```
void BFS(struct node *root)
{
    queue<struct node *> qu;
    struct node *temp;

    qu.push(root);
    while(!qu.empty())
    {
        temp=qu.front();
        qu.pop();
        cout<<temp->data<<"  ";

        if (temp->child[0]!=0)
            qu.push(temp->child[0]);

        if (temp->child[1]!=0)
            qu.push(temp->child[1]);
      // 可以改写为 for 循环
    }
}
```

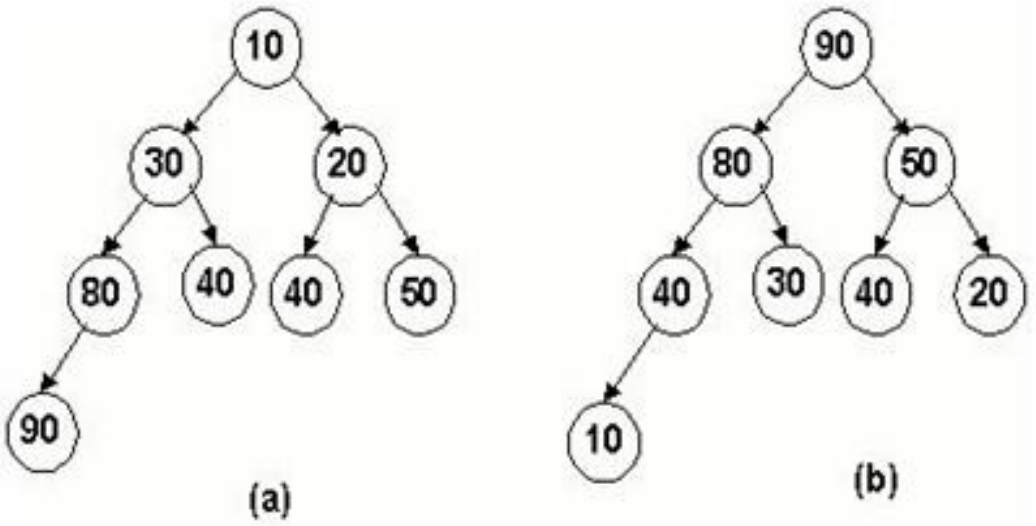


堆与优先队列





- **堆的定义**：设有n个数据元素组成的序列$\{a_1, a_2, \ldots, a_n\}$，若它满足下面的条件：
- （1）这些数据元素是一棵**完全二叉树**中的结点，且a_i(i=1, 2, ... n)是该完全二叉树中编号为i的结点；
- （2）若$2i \leq n$，有$a_{2i} \geq a_i$；
- （3）若$2i+1 \leq n$，有$a_{2i+1} \geq a_i$；
- 则称该序列为一个堆。



小根堆(最小堆)　　**大根堆（最大堆）**

优先队列：priority_queue

- priority_queue 用堆排序技术实现，保证最大的元素总是在最前面【最大堆】——也称最大优先队列。

使用方法

- 1.参考网站:【中文】

http://msdn.microsoft.com/zh-cn/library/4ef4dae9.aspx

- 2.引用头文件

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

priority_queue class

- 成员函数:
- empty 测试 priority_queue 是否为空。
- pop 从priority_queue顶部位置移除最大的元素。
- push 将元素添加到基于元素优先级的优先级队列中。
- size 返回priority_queue的元素个数。
- top 返回优先队列中的最大元素。

元素优先级的确定：默认的元素比较器是 less<T>

即元素值大则优先

测试程序1：

```
#include <queue>
#include <iostream>
using namespace std;
int main()  {
      priority_queue<double> p;
      p.push(3.2);
      p.push(9.8);
      p.push(5.4);
      while( !p.empty() )
      {    cout << p.top() << " ";    p.pop();   }
     return 0;
}
//输出结果：9.8 5.4 3.2
```

优先队列：priority_queue

- priority_queue 用堆排序技术实现，保证最大的元素总是在最前面【最大堆】——也称最大优先队列。
- 即执行pop操作时，删除的是最大的元素；
- 执行top操作时，返回的是最大元素的引用。

元素优先级的改变：改写元素比较器

最小优先队列的构建

例如：5，9，3，想以最小优先的关系存入优先队列中，则需要重新定义元素值之间的优先关系(改写<关系)。

```
struct node
{    int x;
     node(int i) { x=i; }
};

bool operator<(const node &a,const node &b)
{   if(a.x>b.x ) return true;  else   return false;      }

priority_queue<node> p;
```

测试程序2：

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

struct node
{     int x;
      node(int i) { x=i; }
};

bool operator<(const node &a,const node &b)
{   if(a.x>b.x ) return true;  else   return false;       }

int main()  {
    node x(0);
    priority_queue<node> p;
    p.push(node(3));       p.push(node(9));          p.push(node(5));
    while( !p.empty() ) {  x=p.top();   p.pop(); cout<<x.x;          }
   return 0;
}
```

输出结果为3 5 9

优先队列应用

- 将若干个学生信息(只含学号以及单科成绩信息)，以成绩大为优先的方式存入优先队列中，最后将优先队列中的学生信息按成绩从高到低的方式进行输出。

插入若干学生到优先队列

```
#include <queue>
#include <iostream>
using namespace std;
struct student
{    int number;
     int score;
};
bool operator < (const student &a,const student &b)
{  if (a.score<b.score) return true;      else return false;  }

ostream &operator<<(ostream &stream,const student &x)
{  stream<<x.number<<"   "<<x.score<<endl;   return stream;   }

istream &operator>>(istream &stream,student &x)
{    stream>>x.number>>x.score;    return stream;  }

int main()  {
       student x;        int i,n=10;        priority_queue<student> p;
       for(i=0;i<n;i++)
       {        cin>>x;          p.push(x);      }
      while( !p.empty() )       {  x=p.top();   p.pop(); cout<<x;           }
     return 0;
}
```

STL概述与使用

- STL有三大核心部分：容器、算法、迭代器。
- 容器：可容纳各种数据类型的数据结构。
- 迭代器：可依次存取容器中元素的指示器
- 算法：用来操作容器中的元素的函数模板。
  - 例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象。
  - 函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用.

- 比如，数组int array[100]就是个容器，而int*类型的指针变量就可以作为迭代器，可以为这个容器编写一个排序的算法

STL容器

- 容器分为三大种类，第一种与第二种合称为第一类容器：

1) 顺序容器

vector：后部插入/删除，直接访问
deque：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：LIFO
queue：FIFO
priority_queue：优先级高的元素先出

顺序容器简介

1) vector 头文件 <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。

2) deque 头文件 <deque>

也是个动态数组，随机存取任何元素都能在常数时间完成(但性能次于vector)。在两端增删元素具有较佳的性能。

3) list 头文件 <list>

双向链表，在任何位置增删元素都能在常数时间完成。不支持随机存取。

- 上述三种容器称为顺序容器，是因为元素的插入位置同元素的值无关。

关联容器简介

- 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。

1) set/multiset: 头文件 <set>

set 即集合。

set中不允许相同元素；

multiset中允许存在相同的元素。

2) map/multimap: 头文件 <map>

map与set的不同在于map中存放的是成对的key/value；

根据key对元素进行排序，可快速地根据key来检索元素；

map同multimap的不同在于是否允许多个元素有相同的key值。

容器适配器简介

1) stack :头文件 <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

2) queue :头文件 <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。

3）priority_queue :头文件 <queue>

优先级队列。最高优先级元素总是第一个出列

容器的共有成员函数

1) 所有标准库容器共有的成员函数：

- 相当于按词典顺序比较两个容器大小的运算符：
  =, <, <=, >, >=, ==, !=

- empty：判断容器中是否有元素

- max_size: 容器中最多能装多少元素

- size: 容器中元素个数

- swap: 交换两个容器的内容

比较两个容器的例子

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
        vector<int> v1;
        vector<int> v2;
        v1.push_back (5);
        v1.push_back (1);
        v2.push_back (1);
         v2.push_back (2);
        v2.push_back (3);
        cout << (v1 < v2);
        return 0;
}
```

- 若两容器长度相同、所有元素相等，则两个容器就相等，否则为不等。
- 若两容器长度不同，但较短容器中所有元素都等于较长容器中对应的元素，则较短容器小于另一个容器
- 若两个容器均不是对方的子序列，则取决于所比较的第一个不等的元素

运行结果：

0

容器的成员函数

2) 只在第一类容器中的函数：

begin 返回指向容器中第一个元素的迭代器

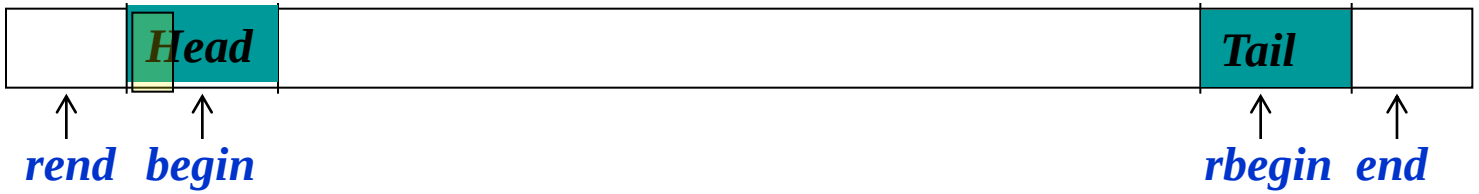
end 返回指向容器中最后一个元素后面的位置的迭代器

rbegin 返回指向容器中最后一个元素的迭代器

rend 返回指向容器中第一个元素前面的位置的迭代器

erase 从容器中删除一个或几个元素

clear 从容器中删除所有元素



迭代器

- 用于指向第一类容器中的元素。
- 迭代器上可以执行 ++ 操作，以指向容器中的下一个元素。
- 迭代器分const 和非 const两种。
- 通过迭代器可以读取它指向的元素，通过非const迭代器还能修改其指向的元素。迭代器用法和指针类似。
- 定义一个容器类的迭代器的方法可以是：

  容器类名::iterator 变量名;

或：

  容器类名::const_iterator 变量名;

- 访问一个迭代器指向的元素：

  * 迭代器变量名

例如：

```
#include <vector>
#include <iostream>
using namespace std;
int main()  {
    vector<int> v;//存放int元素的向量,一开始里面没有元素
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    vector<int>::const_iterator i;   //常量迭代器
    for( i = v.begin();i != v.end();i ++ )
        cout << * i << ",";
    cout << endl;
```

```
    vector<int>::reverse_iterator r;  //反向迭代器
    for( r = v.rbegin();r != v.rend();r++ )
        cout << * r << ",";
    cout << endl;
    vector<int>::iterator j;   //非常量迭代器
    for( j = v.begin();j != v.end();j ++ )
        * j =  100;
    for( i = v.begin();i != v.end();i++ )
        cout << * i << ",";
}
```

输出结果：

1,2,3,4,

4,3,2,1,

100,100,100,100,

迭代器

- 不同容器上支持的迭代器功能强弱有所不同。
- 容器的迭代器的功能强弱，决定了该容器是否支持STL中的某种算法。
  - 例1：只有第一类容器能用迭代器遍历。
  - 例2：排序算法需要通过随机迭代器来访问容器中的元素，那么有的容器就不支持排序算法。

STL 中的迭代器

- STL 中的迭代器按功能由弱到强分为5种：

  1. 输入：Input iterators 提供对数据的只读访问。

  1. 输出：Output iterators 提供对数据的只写访问

  2. 正向：Forward iterators 提供读写操作，并能一次一个地向前推进迭代器。

  3. 双向：Bidirectional iterators提供读写操作，并能一次一个地向前和向后移动。

  4. 随机访问：Random access iterators提供读写操作，并能在数据中随机移动。

- 编号大的迭代器拥有编号小的迭代器的所有功能，能当作编号小的迭代器使用。

容器所支持的迭代器类别

容器	迭代器类别
vector	随机
deque	随机
list	双向
set/multiset	双向
map/multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

不同迭代器所能进行的操作(功能)

- 所有迭代器： ++p, p ++
- 输入迭代器： * p, p = p1, p == p1 , p!= p1
- 输出迭代器： * p, p = p1
- 正向迭代器： 上面全部
- 双向迭代器： 上面全部，--p, p --,
- 随机访问迭代器： 上面全部，以及：
  - p+= i, p -= i,
  - p + i： 返回指向 p 后面的第i个元素的迭代器
  - p - i： 返回指向 p 前面的第i个元素的迭代器
  - p[i]: p 后面的第i个元素的引用
  - p < p1, p <= p1, p > p1, p>= p1

例如，vector的迭代器是随机迭代器，

所以遍历 vector 可以有以下几种做法：

```
vector<int> v(100);
vector<int>::value_type i; //等效于写 int i;
for(i = 0;i < v.size() ; i ++)
        cout << v[i];

vector<int>::const_iterator ii;
for( ii = v.begin(); ii != v.end ();ii ++ )
        cout << * ii;

//间隔一个输出：
ii = v.begin();
while( ii < v.end()) {
        cout << * ii;
        ii = ii + 2;
}
```

而 list 的迭代器是双向迭代器，所以以下代码可以：

```
list<int> v;
list<int>::const_iterator ii;
for( ii = v.begin(); ii ！ ＝ v.end ();ii ++ )
        cout << * ii;
```

以下代码则不行：

```
for( ii = v.begin(); ii < v.end ();ii ++ )
        cout << * ii;          //双向迭代器不支持 <

for(int i = 0;i < v.size() ; i ++)
        cout << v[i];          //双向迭代器不支持 []
for(int j = 0;j < v.size() ;  )
        cout << *(j+2);     //双向迭代器不支持  +2
```

顺序容器

- 除前述共同操作外，顺序容器还有以下共同操作：
  - front() :返回容器中第一个元素的引用
  - back() : 返回容器中最后一个元素的引用
  - push_back(): 在容器末尾增加新元素
  - pop_back(): 删除容器末尾的元素

向量（vector容器类）

- 参考网址：
- http://msdn.microsoft.com/zh-cn/library/vstudio/9xd04bzs%28v=vs.110%29.aspx

- 头文件： #include <vector>

- vector是一种动态数组，是基本数组的类模板。

- 支持随机访问迭代器，所有STL算法都能对vector操作。

- 随机访问时间为常数。

- 在尾部添加速度很快，在中间插入慢。

例1：

```
int main()
{       int i;
        int a[5] = {1,2,3,4,5 };    vector<int>  v(5);
        cout << v.end()  - v.begin() << endl;
        for( i = 0;i < v.size();i ++ )  v[i] = i;
        v.at(4) = 100;
        for( i = 0;i < v.size();i ++ )
                cout << v[i] << "," ;
        cout << endl;
        vector<int> v2(a,a+5);  //构造函数
        v2.insert( v2.begin() + 2, 13 ); //在begin()+2位置插入 13
        for( i = 0;i < v2.size();i ++ )
                cout << v2[i] << "," ;
        return 0;
}
```

运行结果：

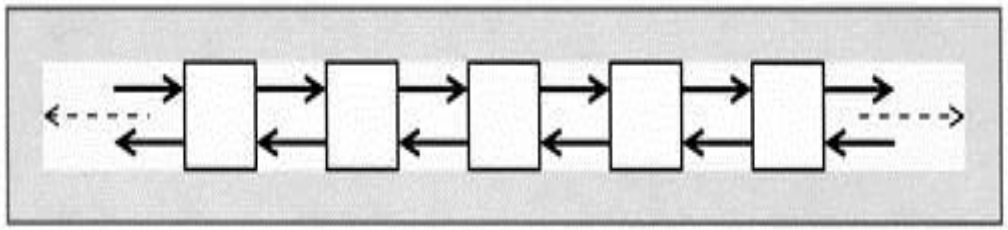
5

0,1,2,3,100,

1,2,13,3,4,5,

表（List容器类）

- List（#include<list>）又叫链表，是一种双线性列表，只能顺序访问（从前向后或者从后向前），list的数据组织形式如下图。
- 与vector容器类有一个明显的区别就是：它不支持随机访问。
- 要访问表中某个下标处的项需要从表头或表尾处（接近该下标的一端）开始循环。
- 不支持下标运算符：operator[]。



- 参考网址：
- http://msdn.microsoft.com/zh-cn/library/00k1x78a%28v=vs.90%29.aspx

表（List容器类）

- 在任何位置插入删除都是常数时间，不支持随机存取。除了具有所有顺序容器都有的成员函数以外，还支持8个成员函数：
  - push_front: 在前面插入
  - pop_front: 删除前面的元素
  - sort: 排序(list 不支持STL 的算法sort)
  - remove: 删除和指定值相等的所有元素
  - unique: 删除所有和前一个元素相同的元素
  - merge: 合并两个链表，并清空被合并的那个
  - reverse: 颠倒链表
  - splice: 在指定位置前面插入另一链表中的一个或多个元素，并在另一链表中删除被插入的元素

使用例子

```
#include <iostream>
#include <string>
#include <list>
using namespace std;
void PrintIt(list<int> n)
{   for(list<int>::iterator iter=n.begin(); iter!=n.end(); ++iter)
    cout<<*iter<<" ";//用迭代器进行输出循环
}
int main()
{
   list<int> listn1,listn2;   //给listn1,listn2初始化
   listn1.push_back(123);      listn1.push_back(0);      listn1.push_back(34);
   listn1.push_back(1123);   //now listn1:123,0,34,1123
   listn2.push_back(100);
   listn2.push_back(12);   //now listn2:12,100
   listn1.sort();
   listn2.sort();   //给listn1和listn2排序
   //now listn1:0,34,123,1123        listn2:12,100
   PrintIt(listn1);
   cout<<endl;
   PrintIt(listn2);
   listn1.merge(listn2);   //合并两个排序列表后,listn1:0，12，34，100，123，1123
   cout<<endl;
   PrintIt(listn1);
}
```

关联容器

- set, multiset, map, multimap
  - 内部元素有序排列，新元素插入的位置取决于它的值，查找速度快
  - map关联数组：元素通过键来存储和读取
  - set大小可变的集合，支持通过键实现的快速读取
  - multimap支持同一个键多次出现的map类型
  - multiset支持同一个键多次出现的set类型
- 与顺序容器的本质区别
  - 关联容器是通过键(key)存储和读取元素的
  - 而顺序容器则通过元素在容器中的位置顺序存储和访问元素。

关联容器

- 除了各容器都有的函数外，还支持以下成员函数：设m表容器，k表键值
  - m.find(k)：如果容器中存在键为k的元素，则返回指向该元素的迭代器。如果不存在，则返回end()值。
  - m.lower_bound(k)：返回一个迭代器，指向键不小于k的第一个元素
  - m.upper_bound(k)：返回一个迭代器，指向键大于k的第一个元素
  - m.count(k)：返回m中k的出现次数
  - 插入元素用 insert

集和多集（set 和multiset 容器类）

- set，顾名思义，就是集合的意思
- 包含头文件：#include<set>
- 一个集合就是一个容器，它其中所包含的元素的值是唯一的。这在收集一个数据的具体值的时候是有用的。
- 它支持插入，删除，查找，首元素，末元素等操作，就像一个集合一样。
- 而且所有的操作都是在严格logn时间之内完成，效率非常高,对于动态维护可以说是很理想的选择。
- 集合中的元素按一定的顺序排列，并被作为集合中的实例。缺省的比较器为less<T>。即缺省为升序。

集和多集（set 和multiset 容器类）

- set是一个有序的容器,里面的元素都是排好序的,缺省为升序。
- 如果先后往一个集中插入：12，2，3，123，5，65
- 则输出该集时为：2，3，5，12，65，123
- 集和多集的区别是：set支持唯一键值，set中的值都是特定的，而且只出现一次；
- 而multiset中可以出现副本键，同一值可以出现多次。
- 如果你需要一个键/值对（pair）来存储数据，map是一个更好的选择。

集和多集（set 和multiset 容器类）

- set的实现是通过二叉排序树来实现，就是将所有的元素用一个树来存储，根元素大于左子树的元素，小于右子树的元素，所有的操作都是基于这个树，通过判断元素的大小来选择对左子树操作还是右子树操作，这样操作数量和树的层数成正比。所有的操作都是在严格logn时间之内完成。
- 因此，在插入操作和删除操作上比向量（vector）快。

深入了解

- http://msdn.microsoft.com/zh-cn/library/y49kh4ha%28v=vs.71%29

自定义元素类型的比较器定义方法

- include<set>
  typedef struct
  {定义类型　}ss(类型名);

- bool operator<(const ss &a, const ss &b)
  {定义比较关系(必须满足上一页2个要求)}
  (运算符重载，重载<)

- set<ss> base; (创建一个元素类型是ss,名字是base的set)

set使用例子(升序)——自定义结构

```
#include<iostream>
#include<set>
using namespace std;
struct student
{   int age;
    student(int i) { age=i; }
};
bool operator<(const student &a,const student &b)
{   if(a.age<b.age ) return true;  else   return false;   }
void main()
{   set<student> base;
    base.insert(50);   base.insert(20);   base.insert(80);
    set<student>::iterator setit;
   for (setit=base.begin(); setit!=base.end(); setit++)
        cout<<(*setit).age<<endl;
}
```

```
20
50
80
Press any key to continue
```

set使用例子(降序)——自定义结构

```
#include<iostream>
#include<set>
using namespace std;
struct student
{   int age;
    student(int i) { age=i; }
};
bool operator<(const student &a,const student &b)
{   if(a.age>b.age ) return true;  else   return false;   }
void main()
{   set<student> base;
    base.insert(50);   base.insert(20);   base.insert(80);
    set<student>::iterator setit;
   for (setit=base.begin(); setit!=base.end(); setit++)
        cout<<(*setit).age<<endl;
}
```

```
80
50
20
Press any key to continue
```

对于内置类型自定义比较关系

- 对于内置的数据类型，如int，double等是不能重载<的，如果想自定义比较关系<可以用以下格式实现；
- 以int为例：
- struct cmp
- {

```
    bool operator() ( const int &a, const int &b ) const
    { 定义比较关系 < }
};
set<int,cmp> base;
```

- 这样就创建了一个元素类型是int，自定义比较关系的，名字是base的set.

set使用例子(降序)

```
#include<iostream>
#include<set>
using namespace std;
struct cmp
{   bool operator () ( const int &a, const int &b ) const
    { if (a>b) return true;   else return false; }
};
void main()
{  set<int,cmp> base;
   base.insert(10);    base.insert(5);   base.insert(100);
  set<int,cmp>::iterator setit;
  for (setit=base.begin(); setit!=base.end(); setit++)
          cout<<*setit<<endl;
}
```

```
100
10
5
Press any key to continue
```

Set的操作(成员函数)

- 以对象base为例说明Set的操作
- base.clear( )　　清空base
- base.insert(a) 插入元素a
- base.erase(a) 删除元素a,如果a是迭代器,就删除迭代器指向的元素
- base.lower_bound(a) 顺序查找>=a的第一个元素,返回迭代器
- base.upper_bound(a) 顺序查找>a的第一个元素,返回迭代器

Set操作(成员函数)

- base.begin( ) 求第 1 个元素，返回迭代器，由于是顺序的容器，所以缺省时第 1 个元素就是最小的元素

- base.end( ) 求容器的末尾，表示容器到了末尾，返回迭代器，这个迭代器里没有元素，专门表示末尾
- base.rbegin( ) 求倒数第 1 个元素，缺省时即最大元素，返回迭代器，（注意：这个迭代器是逆向迭代器和base.begin()的返回值类型不同）

Set操作(成员函数)

- base.size() 求容器里元素的数量，返回整数
- base.empty() 判断容器是否是空，空返回true,否则返回false;
- base.find(a) 查找元素a,如果查到了，返回指向a的迭代器，否则返回容器末尾迭代器，即base.end();
- base.count(a) 查找元素a的数量，返回整数

set和multiset

- Set要求容器里的元素在定义的比较关系下都是不相等的，如果已经有，那么insert是无效的，如果必须在容器中放入相同的元素就要使用multiset。
- multiset和set有几点不同：
- 创建 multiset<ss> base;
- 删除：如果删除元素a,那么在定义的比较关系下和a相等的所有元素都会被删除
- base.count(a)：set能返回 0 或者 1 ，multiset是有多少个返回多少个.
- Set和multiset都是引用<set>头文件,复杂度都是logn

set 程序例子

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

int main( )
{     set <int>::iterator s1_Iter;      set <int> s0; // Create an empty set s0 of key type integer

  // Create an empty set s1 with the key comparison , function of less than, then insert 4 elements
  set <int, less<int> > s1;      s1.insert( 10 );    s1.insert( 20 );      s1.insert( 30 );      s1.insert( 40 );

  // Create an empty set s2 with the key comparison , function of geater than, then insert 2 elements
  set <int, greater<int> > s2;       s2.insert(10);      s2.insert(20);

  cout << "s1 =";
  for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )         cout << " " << *s1_Iter;
  cout << endl;

  cout << "s2 = " << *s2.begin( ) << " " << *++s2.begin( ) << endl;
  return 0;
}

/*
s1 = 10 20 30 40
s2 = 20 10
*/
```

set使用例子(反向迭代器)

```
#include<iostream>
#include<set>
using namespace std;
struct student
{   int age;
    student(int i) { age=i; }
};
bool operator<(const student &a,const student &b)
{   if(a.age<b.age ) return true;  else   return false;   }
void main()
{   set<student> base;
    base.insert(50);   base.insert(20);   base.insert(80);
    set<student>::reverse_iterator setit; // 反向迭代器
   for (setit=base.rbegin(); setit!=base.rend(); setit++)
        cout<<(*setit).age<<endl;
}
```

```
80
50
20
Press any key to continue
```

set使用例子——删除元素

```
#include<iostream>
#include<set>
using namespace std;
struct student
{    int age;
     student(int i) { age=i; }
};
bool operator<(const student &a,const student &b)
{    if(a.age<b.age ) return true;  else   return false;    }
void main()
{    set<student> base;
     base.insert(50);   base.insert(20);   base.insert(80);
    base.erase(50);
     set<student>::iterator  setit;
   for (setit=base.begin(); setit!=base.end(); setit++)
          cout<<(*setit).age<<endl;
}
```

```
20
80
Press any key to continue
```

- set的应用例子

You have known that nuanran is a loyal fan of Kelly from the last contest. For this reason, nuanran is interested in collecting pictures of Kelly. Of course, he doesn't like each picture equally. So he gives each picture a score which is a positive integer. And the larger the score is, the more he likes that picture. Nuanran often goes to buy new pictures in some shops to increase his picture collection.

Sometimes nuanran's friends ask him for Kelly's picture and he will choose the picture having the smallest score. Because he has many pictures, this is a boring task. Can you help him again?

- Input

  For each test case, the first line gives an integer n (1 ⩽ n ⩽ 100000). Then n lines follow, each line has one of the following two formats.

  "B S". Denoting nuanran buys a new picture and S is the score of that picture.

  "G". Denoting nuanran gives a picture to his friends.

  The input is terminated when n=0.
- Output

  For each giving test case, you should output the score of the picture nuanran gives to his friends.
- Sample Input

  8 B 20 B 10 G B 9 G B 100 B 25 G 0
- Sample Output

  10 9 20

- 题目描述：
  nuanran有一些图片，他会收集到新的图片，但是也会有朋友找他要图片，所以nuanran想把当时价值最小的图片给他的朋友，请你帮助他.
- 输入：多组数据，每组数据第一行是整数n(1 ≤ n ≤ 100000)，表示下面有n行
  每行有2种形式：
  “B S”：表示得到价值为S的图片
  “G”：表示要给出当时价值最小的图片
  n = 0表示输入结束.

- 输出：按照“G”的顺序输出所有给出图片的价值
  开始时nuanran没有图片
- 样例输入：
  8
  B 20
  B 10
  G
  B 9
  G
  B 100
  B 25
  G
  0

样例输出：

10
9
20

解题思路

- 该题目的要求实质就是维护一个集合，支持2个操作，插入一个元素，删除最小元素，这个可以用堆来实现，不过如果用multiset来实现，代码会很轻松的实现！

补充资料

- http://hi.baidu.com/jjzhang166/blog/item/55038d4590971e378794734c.html
- http://msdn.microsoft.com/zh-cn/library/bb385157.aspx

查找问题

- 如何才能让查找效率高？

【例】11个元素的关键码分别为18，27，1，20，22，6，10，13，41，15，25。

问题一:如何存储才能使查找效率提高?

f（key）=key % 11

0	1	2	3	4	5	6	7	8	9	10
22	1	13	25	15	27	6	18	41	20	10

问题二:如何查找?

问题三:有何缺陷?

主要问题

1.映射函数如何构造?

2. 如何解决冲突?

2. 如何解决冲突?

(1) 线性探查方法

0	1	2	3	4	5	6	7	8	9	10
55	**01**	**23**	**14**	**68**		**82**	**36**	**19**		

例如：关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

H(key) = key % 11

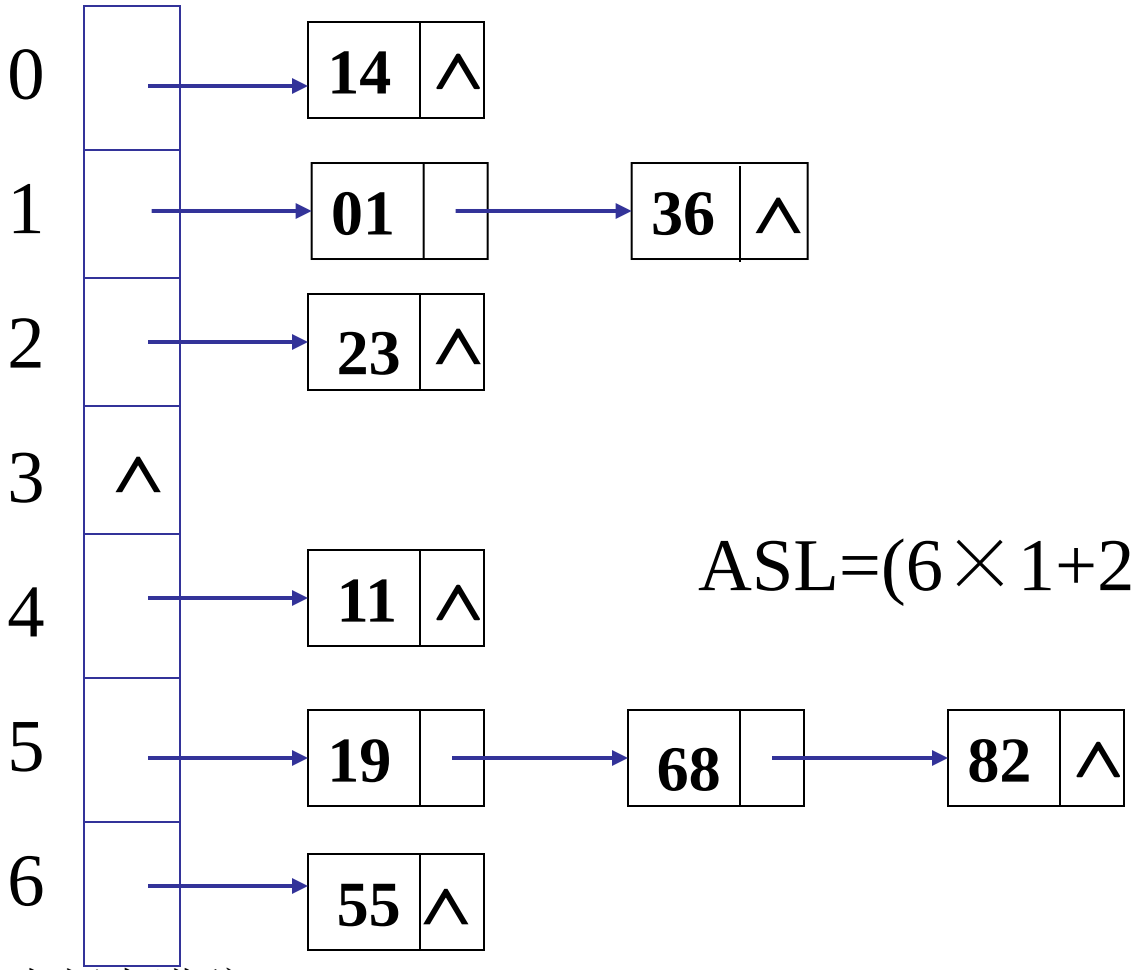
采用线性探查方法处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	**01**	**23**	**14**	**68**	**11**	**82**	**36**	**19**		

（2）链地址法

将所有哈希地址相同的记录都链接在同一链表中。

例 { 19, 01, 23, 14, 55, 68, 11, 82, 36 }　　H(key)=key % 7



ASL=(6×1+2×2+3)/9=13/9

pair模板

- pair模板类用来绑定两个对象为一个新的对象，该类型在<utility>头文件中定义。
- pair模板类支持如下操作：
  - 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
  - pair<T1, T2> p1(v1, v2)：创建一个pair对象，它的两个元素分别是T1和T2类型，其中first成员初始化为v1，second成员初始化为v2
  - 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
  - p1 < p2字典次序：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。
  - p.first：返回p中名为first的（公有）数据成员
  - p.second：返回p中名为second的（公有）数据成员

```
#include <set>
#include <iostream>
using namespace std;
int main( )
{
    typedef set<double,less<double> > double_set;
    const int SIZE = 5;
    double a[SIZE] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
    double_set doubleSet(a,a+SIZE);
    ostream_iterator<double> output(cout," ");
    cout << "1) ";
    copy(doubleSet.begin(),doubleSet.end(),output);
    cout << endl;
    pair<double_set::const_iterator, bool> p;
    p = doubleSet.insert(9.5);
    if( p.second )
        cout << "2) " << * (p.first)  << " inserted" << endl;
    else
        cout << "2) " << * (p.first)  << " not inserted" << endl;
    return 0;
}
```

insert函数返回值是一个pair对象, 其first是被插入元素的迭代器，second代表是否成功插入了

运行结果：
1) 2.1 3.7 4.2 9.5
2) 9.5 not inserted

映射和多重映射（map 和multimap）

- 头文件： #include<map>
- 映射和多重映射基于某一类型Key的键集的存在，提供对T类型的数据进行快速和高效的检索。
- 对map而言，键只是指存储在容器中的某一成员。
- map不支持副本键，multimap支持副本键。
- map和multimap对象包涵了键和各个键有关的值，键和值的数据类型是不相同的，这与set不同。
- set中的key和value是Key类型的，而map中的key和value是一个pair结构中的两个分量。
- map支持下表运算符operator[],用访问普通数组的方式访问map，不过下标为map的键。
- 在multimap中一个键可以对应多个不同的值。

multimap

```
template<class Key, class T, class Pred = less<Key>, class A = allocator<T> >
class multimap {
    ….
    typedef pair<const Key, T> value_type;
    …….
};  //Key 代表关键字
```

- multimap中的元素由 <关键字,值>组成，每个元素是一个pair对象。
- multimap 中允许多个元素的关键字相同。
- 元素按照关键字升序排列，缺省情况下用 less<Key> 定义关键字的“小于”关系

map

```
template<class Key, class T, class Pred = less<Key>, class A =
    allocator<T> >
class map {
    ….
    typedef pair<const Key, T> value_type;
    …….
};
```

- map 中的元素关键字各不相同。
- 元素按照关键字升序排列，缺省情况下用 less 定义“小于”

map

- 可以用pairs[key]访形式问map中的元素。
  - pairs为map容器名，key为关键字的值。
  - 该表达式返回的是对关键值为key的元素的值的引用。
  - 如果没有关键字为key的元素，则会往pairs里插入一个关键字为key的元素，并返回其值的引用
- 如：

  map<int,double> pairs;

  则 pairs[50] = 5.0; 会修改pairs中关键字为50的元素，使其值变成5.0

map

- find查找函数：
- 来定位数据出现位置，它返回的一个迭代器；
- 当数据出现时，它返回数据所在位置的迭代器；
- 如果map中没有要查找的数据，它返回的迭代器等于end函数返回的迭代器.

- 删除：int n = Pairs.erase(50);//用关键字删除,如果删除了会返回1，否则返回0

```
#include <iostream>
#include <map>
using namespace std;
ostream & operator <<( ostream & o,const pair<  int,double> & p)
{    o << "(" << p.first  << "," << p.second << ")";
     return o;
}

int main()
 {
    typedef map<int,double,less<int> > mmid;
    mmid pairs;
    cout << "1) " << pairs.count(15) << endl;
    pairs.insert(mmid::value_type(15,2.7));
    pairs.insert(make_pair(15,99.3));//make_pair生成pair对象
    cout << "2) " << pairs.count(15) << endl;
    pairs.insert(mmid::value_type(20,9.3));
```

输出：

1) 0

2) 1

```
    mmid::iterator i;
    cout << "3) ";
    for( i = pairs.begin(); i != pairs.end();i ++ )
        cout << * i  << ",";
cout << endl;
cout << "4) ";
int n =  pairs[40];//如果没有关键字为40的元素，则插入一个
for( i = pairs.begin(); i != pairs.end();i ++ )
    cout << * i  << ",";
cout << endl;
cout << "5) ";
pairs[15] = 6.28; //把关键字为15的元素值改成6.28
for( i = pairs.begin(); i != pairs.end();i ++ )
    cout << * i  << ",";
    return 0;
}
```

3) (15,2.7),(20,9.3),

4) (15,2.7),(20,9.3),(40,0),

5) (15,6.28),(20,9.3),(40,0),

删除元素以及反向输出的方法

```
pairs.erase(15);

cout << endl;
cout << “6) ";

mmid::reverse_iterator j;

for( j = pairs.rbegin(); j != pairs.rend();j ++ )
      cout << * j  << ",";
```

```
6)(40,0),(20,9.3),
```

map使用例子

- #include <map>
  #include <string>
  using namespace std;
- map<string, string> TNmap;
  map<char,int> MAP;
- //建立映射
  TNmap["kitty"]="中国人";
  TNmap["john"]="美国人";
  TNmap["jack"]="日本人";
- MAP[(char)(i+65)]=10;
- //查找语句
  if(TNmap.find("jack") != TNmap.end()){ … }

hash_map

- hash_map 查找速度会比map快，而且查找速度基本和数据数据量大小，属于常数级别；

- 而map的查找速度是log(n)级别

特别数据的hash函数

- 字符串的hash函数

- strhash
- elfhash
- hflp
- Hf

- 字典组织

思考题：map的应用例子1

- 如何用程序用来统计一篇英文文章中单词出现的频率（为简单起见，假定依次从键盘输入该文章）

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, int> wordCounter;
    map<string, int>::iterator it;
    string word;
    while (cin >> word)
        ++wordCounter[word];
    for ( it= wordCounter.begin(); it !=wordCounter.end(); ++it)
        cout<<"Word: "<<(*it).first<<" \tCount: "<<(*it).second<<endl;
    return 0;
}
```

map 的应用例子2

- acm.pku.edu.cn
- 2503 babelfish

Babelfish

- Description
- You have just moved from Waterloo to a big city. The people here speak an incomprehensible dialect of a foreign language. Fortunately, you have a dictionary to help you understand them.

- Input
- Input consists of up to 100,000 dictionary entries, followed by a blank line, followed by a message of up to 100,000 words. Each dictionary entry is a line containing an English word, followed by a space and a foreign language word. No foreign word appears more than once in the dictionary. The message is a sequence of words in the foreign language, one word on each line. Each word in the input is a sequence of at most 10 lowercase letters.
- Output
- Output is the message translated to English, one word per line. Foreign words not in the dictionary should be translated as "eh".

- Sample Input

dog ogday
cat atcay
pig igpay
froot ootfray
loops oopslay

atcay
ittenkay
oopslay

- Sample Output

cat
eh
loops

- Hint

Huge input and output,scanf and printf are recommended.

STL算法简介

- STL中提供能在各种容器中通用的算法，比如插入，删除，查找，排序等。
- 大约有70种标准算法。
  - 算法就是一个个函数模板。
  - 算法通过迭代器来操纵容器中的元素。许多算法需要两个参数，一个是起始元素的迭代器，一个是终止元素的后面一个元素的迭代器。比如，排序和查找
  - 有的算法返回一个迭代器。比如 find() 算法，在容器中查找一个元素，并返回一个指向该元素的迭代器。
  - 算法可以处理容器，也可以处理C语言的数组

STL算法分类

- 变化序列算法:可以改变容器内的数据；
  - 复制(copy)、交换(swap)、替换(replace)、删除(clear)，删除(remove)、逆置翻转(reverse)等算法。
- 非变化序列算法：处理容器内的数据而不改变他们；
  - adjacent-find, equal, mismatch,find ,count, search, count_if, for_each, search_n
- 排序值算法：包涵对容器中的值进行排序和合并的算法。
- 以上函数模板都在<algorithm> 中定义
- 此外还有其他算法，比如<numeric>中的算法

算法示例：find()

```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
```

- first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找区间起点和终点。
  - 这个区间是个左闭右开的区间，即区间的起点是位于查找范围之中的，而终点不是
- val参数是要查找的元素的值
- 函数返回值是一个迭代器。
- 如果找到，则该迭代器指向被找到的元素。
- 如果找不到，则该迭代器指向查找区间终点。

算法示例：find()

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
main()  {
        int array[10] = {10,20,30,40};
        vector<int> v;
        v.push_back(1);         v.push_back(2);
        v.push_back(3);         v.push_back(4);
        vector<int>::iterator p;
        p = find(v.begin(),v.end(),3);
        if( p != v.end())
                cout << * p << endl;
```

输出：
3

```
    p = find(v.begin(),v.end(),9);
    if( p == v.end())
        cout << "not found " << endl;
    p = find(v.begin()+1,v.end()-2,1);
    if( p != v.end())
        cout << * p << endl;
    int * pp = find( array,array+4,20);
    cout << * pp << endl;
}
```

输出:

```
3
not found
3
20
```

排序和查找算法

- Sort
  template<class RanIt>
  void sort(RanIt first, RanIt last);
  void sort(RanIt first, RanIt last, BinaryPredicate Comp);

- binary_search 折半查找，要求容器已经有序
  template<class FwdIt, class T>
  bool binary_search(FwdIt first, FwdIt last, const T& val);

Sort的使用说明

http://msdn.microsoft.com/en-us/library/ecdecxh1(VS.80).aspx

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.

```
template<class RandomAccessIterator>
void sort( RandomAccessIterator _First,RandomAccessIterator _Last);

template<class RandomAccessIterator, class Pr>
void sort( RandomAccessIterator _First,RandomAccessIterator _Last,
    BinaryPredicate _Comp
  );
```

Sort的使用说明

Parameters

_First

A random-access iterator addressing the position of the first element in the range to be sorted.

_Last

A random-access iterator addressing the position one past the final element in the range to be sorted.

_Comp

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns true when satisfied and false when not satisfied. This comparator function must impose a strict weak ordering on pairs of elements from the sequence. For more information, see Algorithms.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The sort algorithm is not stable and so does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm stable_sort does preserve this original ordering.

The average of a sort complexity is O(N log N), where N = _Last – _First.

Sort 使用例子

```
#include <vector>
#include <algorithm>
#include <functional>     // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 )
{  return elem1 > elem2;  }

int main( )
{
  using namespace std;    vector <int> v1;
   vector <int>::iterator Iter1;

  int i;
  for ( i = 0 ; i <= 5 ; i++ )
  {      v1.push_back( 2 * i );      }

  int ii;
  for ( ii = 0 ; ii <= 5 ; ii++ )
  {      v1.push_back( 2 * ii + 1 );    }

  cout << "Original vector v1 = ( " ;
  for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
  cout << ")" << endl;

  sort( v1.begin( ), v1.end( ) );
  cout << "Sorted vector v1 = ( " ;
  for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
  cout << ")" << endl;

// To sort in descending order. specify binary predicate
  sort( v1.begin( ), v1.end( ), greater<int>( ) );
  cout << "Resorted (greater) vector v1 = ( " ;
  for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
  cout << ")" << endl;

  // A user-defined (UD) binary predicate can also be used
  sort( v1.begin( ), v1.end( ), UDgreater );
  cout << "Resorted (UDgreater) vector v1 = ( " ;
  for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
  cout << ")" << endl;
}
```

输出：

```
Original vector v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Sorted vector v1 = ( 0 1 2 3 4 5 6 7 8 9 10 11 )
Resorted (greater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )
Resorted (UDgreater) vector v1= ( 11 10 9 8 7 6 5 4 3 2 1 0 )
```

Sort的使用例子2

```
#include<vector>
#include<iostream>
#include <algorithm>
#include<iterator>
using namespace std;
int main( )
{
      const int SIZE = 10;
      int a1[] = { 2,8,1,50,3,100,8,9,10,2 };
      vector<int> v(a1,a1+SIZE);
      ostream_iterator<int> output(cout," ");
      vector<int>::iterator location;
      location = find(v.begin(),v.end(),10);
      if( location != v.end()) {  cout << endl << "1) " << location - v.begin();   }
      //sort(v.begin(),v.end());
       if( binary_search(v.begin(),v.end(),9))
          cout << endl << “2) " << "9 found";
      else
          cout << endl << " 2) " << " 9 not found";
     return 0;
}
```

输出：(无sort语句)

1) 8

2) 9 not found

输出： (有sort语句)

1) 8

2) 9 found

Sort的使用例子2

```
#include<vector>
#include<iostream>
#include <algorithm>
#include<iterator>
using namespace std;
int main( )
{
      const int SIZE = 10;
      int a1[] = { 2,8,1,50,3,100,8,9,10,2 };
      vector<int> v(a1,a1+SIZE);
      ostream_iterator<int> output(cout," ");
      vector<int>::iterator location;
      location = find(v.begin(),v.end(),10);
      if( location != v.end()) {  cout << endl << "1) " << location - v.begin();   }
      //sort(v.begin(),v.end());
       if( binary_search(v.begin(),v.end(),100))
          cout << endl << “2) " << “100 found";
      else
          cout << endl << " 2) " << " 100 not found";
     return 0;
}
```

输出：(无sort语句)

1) 8

2) 100 found

输出： (有sort语句)

1) 8

2) 100 found

STL排序函数sort的例子

升序排序 sort.cpp

```
#include<iostream>
#include<algorithm>
using namespace std;
#define m 10
int a[m];
bool cmp(const int & a ,const int & b)
{            return a<b;     }

main()
{   int i;
   for(i=0;i<m;i++)
      cin>>a[i];

    sort(&a[0],&a[m],cmp);

    cout<<endl;
    for(i=0;i<m;i++)
       cout<<a[i]<<"   " ;
}
```

STL排序函数sort的例子

降序排序　sort0.cpp

```
#include<iostream>
#include<algorithm>
using namespace std;
#define m 10
int a[m];
bool cmp(const int & a ,const int & b)
{            return a>b;     }

main()
{   int i;
   for(i=0;i<m;i++)
      cin>>a[i];

    sort(&a[0],&a[m],cmp);

    cout<<endl;
    for(i=0;i<m;i++)
       cout<<a[i]<<"   " ;
}
```

sort

- sort 实际上是快速排序，时间复杂度 O(n*log(n));
  - 平均性能最优。但是最坏的情况下，性能可能非常差。

  如果要保证“最坏情况下”的性能，那么可以使用stable_sort

- stable_sort
  - stable_sort 实际上是归并排序（将两个已经排序的序列合并成一个序列），特点是能保持相等元素之间的先后次序
  - 在有足够存储空间的情况下，复杂度为 n * log(n)，否则复杂度为 n * log(n) * log(n)

- stable_sort 用法和 sort相同
  - 排序算法要求随机存取迭代器的支持，所以list 不能使用排序算法，要使用list::sort

quick sort　运用C++库函数

- 快速排序(Quick Sort)是一种有效的排序算法。
- 虽然算法在最坏的情况下运行时间为O(n^2)，但由于平均运行时间为O(nlogn)，并且在内存使用、程序实现复杂性上表现优秀，尤其是对快速排序算法进行随机化的可能，使得快速排序在一般情况下是最实用的排序方法之一。

- 快速排序被认为是当前最优秀的内部排序方法。

C++标准库的快速排序函数

- qsort(void* base, size_t num, size_t width, int(*)compare(const void* elem1, const void* elem2))

- 引用头文件： #include <stdlib.h>

- 参数表
- *base: 待排序的元素（数组，下标0起）。
- num: 元素的数量。
- width: 每个元素的内存空间大小(以字节为单位)。可用sizeof()测得。
- int(*)compare: 指向一个比较函数。*elem1 *elem2: 指向待比较的数据。
- 比较函数的返回值
  - 返回值是int类型，确定elem1与elem2的相对位置。
  - elem1在elem2右侧返回正数，elem1在elem2左侧返回负数。
  - 控制返回值可以确定升序/降序。

使用C++标准库的快速排序函数

一个升序排序的例子：

```
int Compare(const void *elem1, const void *elem2)
{    return *((int *)(elem1)) - *((int *)(elem2));  }

int main()
{     int a[100];
      ……
      qsort(a, 100, sizeof(int), Compare);
      return 0;
}
```

STL排序函数sort与标准库快速排序函数qsort

- 一、STL排序函数sort
- 1.http://msdn.microsoft.com/en-us/library/ecdecxh1(VS.80).aspx
- 2.引用头文件: <algorithm>
- 二、标准库快速排序函数qsort
- 1. http://msdn.microsoft.com/en-us/library/aa272872%28v=vs.60%29.aspx
- 2.引用头文件: <stdlib.h>

分别用sort与qsort来实现下图的效果

```
50 10 60 90 100 20 30 40 70 80

100   4
90   3
80   9
70   8
60   2
50   0
40   7
30   6
20   5
10   1
Press any key to continue
```

STL排序函数sort的例子

Sort1.cpp　降序排序

```
#include<iostream>
#include<algorithm>
using namespace std;
#define m 10
struct node {   int key;  int pos;   } a[m+1];

bool cmp(const node  & a ,const node & b)
{     return a.key>b.key;   }

main()
{
   int i;
   for(i=0;i<m;i++)
    {          cin>>a[i].key;          a[i].pos=i;    }

   sort(&a[0],&a[m],cmp);

   cout<<endl;
   for(i=0;i<m;i++)
         cout<<a[i].key<<"   " <<a[i].pos<<endl;
}
```

标准库快速排序函数qsort

降序排序　qsort.cpp

```
#include<iostream>
//#include<algorithm>
#include <stdlib.h>
using namespace std;
#define m 10
struct node {    int key;  int pos;   } a[m+1];

int cmp(const void  *a ,const void *b)
{    return (*(node *)b).key-(*(node *)a).key;     }

main()
{
  int i;
  for(i=0;i<m;i++)
   {        cin>>a[i].key;      a[i].pos=i;    }

   qsort(a, m, sizeof(node), cmp);

  cout<<endl;
  for(i=0;i<m;i++)
      cout<<a[i].key<<"   " <<a[i].pos<<endl;
}
```

练习题

- 1204
- 2734
- 2212
- 1431
- 2099
- 2727
- 1799
- 1431
- 1764
- 1614
- 2635
- 2397
- 1159
- 1400
- 1101
- 1076
- 2499

STL学习资料

- www.stlchina.org STL资料汇总网站