## 1.六大组件说明

六大组件完成交互。容器（containers）通过空间配置器（allocator）取得数据存储空间，算法（algorithm）通过迭代器（iterator）存取容器（containers）内容。仿函数（functor）可以协助算法（algorithm）完成不同策略的变化，适配器（adapter）可以修饰或套接仿函数（functor）。

**容器**（Containers）：各种数据结构，如：vector、list、deque、set、map。用来存放数据。从实现的角度来看，STL容器是一种class template。

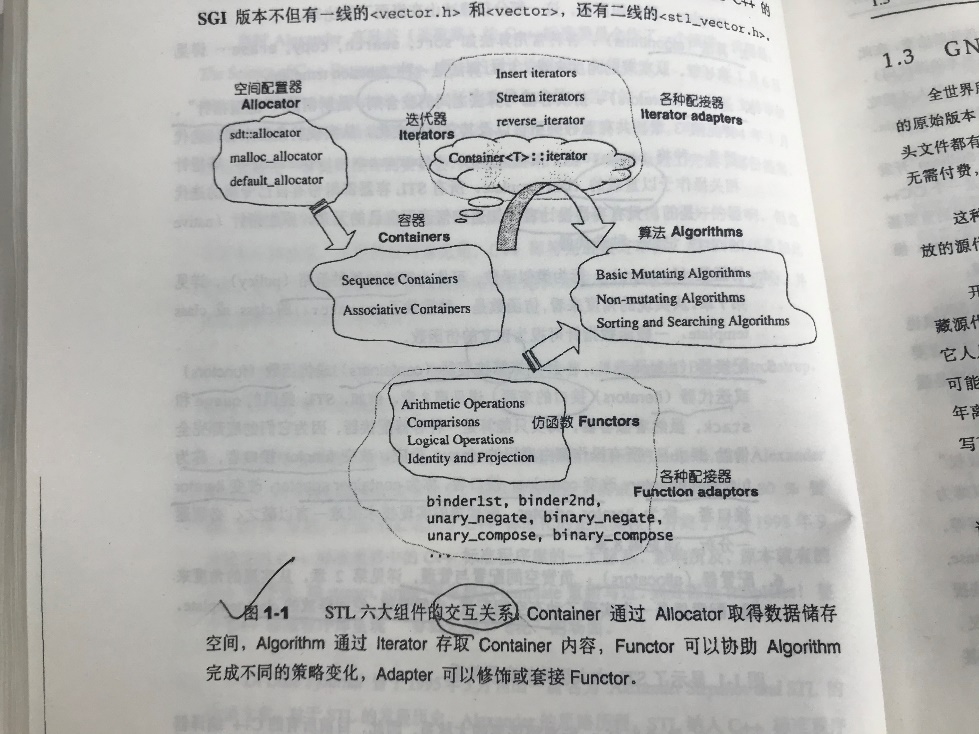
**算法**（algorithms）：各种常用算法，如：sort、search、copy、erase。从实现的角度来看，STL算法是一种 function template。

**迭代器**（iterators）：容器与算法之间的胶合剂，是所谓的“泛型指针”。共有五种类型，以及其他衍生变化。从实现的角度来看，迭代器是一种将 operator\*、operator->、operator++、operator- - 等指针相关操作进行重载的class template。所有STL容器都有自己专属的迭代器，只有容器本身才知道如何遍历自己的元素。原生指针(native pointer)也是一种迭代器。

**仿函数**（functors）：行为类似函数，可作为算法的某种策略（policy）。从实现的角度来看，仿函数是一种重载了operator()的class或class template。一般的函数指针也可视为狭义的仿函数。

**配接器**（adapters）：一种用来修饰容器、仿函数、迭代器接口的东西。例如：STL提供的queue 和 stack，虽然看似容器，但其实只能算是一种容器配接器，因为它们的底部完全借助deque，所有操作都由底层的deque供应。改变 functors接口者，称为function adapter；改变 container 接口者，称为container adapter；改变iterator接口者，称为iterator adapter。

**配置器**（allocators）：负责空间配置与管理。从实现的角度来看，配置器是一个实现了动态空间配置、空间管理、空间释放的class template。



## 2. type\_traits.h

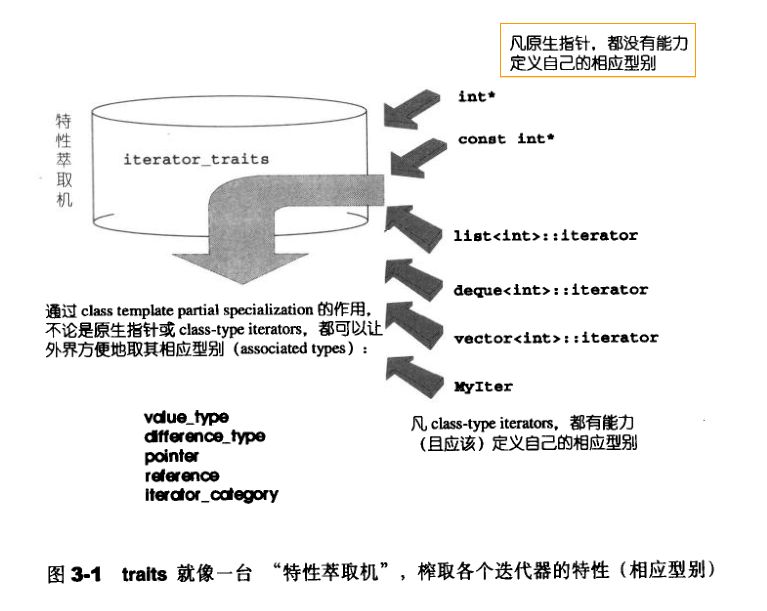
这个头文件用于提取类型信息. 关键词typename 的用意在于告诉编译器这是个型别

Template<class T>

Struct iterator\_traits{//traits意为特性

Typedef typename T::value\_type value\_type; };

Traits含义：如果T有自己的value type，那么通过萃取，就可以萃取出其型别。所有原生指针都没有能力定义自己的型别（如int），凡class-type iterators都有能力定义自己的相应型别。



PlainOld Data, 就表示POD是个普通的类型，朴素而又平凡，在C++常见的类型，标量类型和POD类类型都有这样的属性，而不像一些存在着复杂特性的类型，比如有这虚函数虚继承这样的类的类型这么特别。Old，则体现了其与C的兼容性。比如可以用memset函数初始化，memcpy函数进行复制。

具体的说，POD被分为两个基本概念的合集，即平凡的（trivial）和标准布局（standard layout）。

数据类型太复杂，不能兼容语义层面的检查，于是就需要POD类型。

其实在C++1`1中，很多内置类型默认都是POD类型的。那么用POD究竟有什么好处呢？

字节赋值。可以安全的使用 memset memcpy对POD类型初始化和拷贝操作。

提供对C内存布局兼容。C++程序可以与C函数进行互操作，POD类型的数据在C和C++间的操作总是安全的。

保证了静态初始化的安全有效。静态初始化的时在很多时候能够提高程序性能，而POD类型的对象初始化更简单，（比如放入目标文件的.bss段，在初始化直接赋0）

//萃取传入的T类型的类型特性

template<class T>

struct \_type\_traits {

typedef \_false\_type has\_trivial\_default\_constructor;

typedef \_false\_type has\_trivial\_copy\_constructor;

typedef \_false\_type has\_trivial\_assignment\_operator;

typedef \_false\_type has\_trivial\_destructor;

typedef \_false\_type is\_POD\_type;

};

## 3. iterator.h

**value type**：迭代器所指对象的型别。

**difference type** ：表示两个迭代器之间的距离，可用于表示容器的最大容量。

**pointer type：**代表迭代器所指对象的指针类型。简言之，operator->()的返回类型。

**reference type**：代表迭代器所指对象的引用类型。简言之，operator\*()的返回类型。分为两种，不允许改变“所指对象之内容”者称为constant iterators，例如 const int\* pic，允许改变“所指对象之内容”者称为 mutable iterators，例如 int\* pi。 \*pic/\*pi的型别constant T&/T&就是reference type。

**iterator category type**：提出5种迭代器的类型标识。

// 这个头文件用于迭代器设计，包含了一些模板结构体与全局函数，

**// 五种迭代器类型**

**Input Iterator:不允许外界改变。只读。**

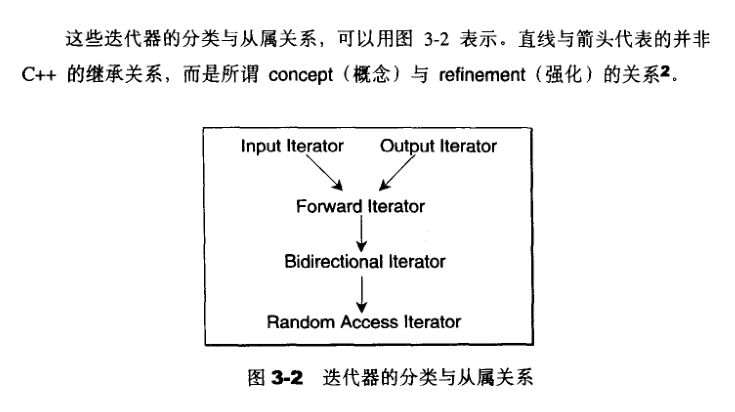
**Output Iterator:唯写（write only)**

**Forward Iterator: 允许写入型算法（例如replace（））在此种迭代器所形成的区间上进行读写操作。**

**Bidirectional Iterator：可双向移动。某些算法需要逆向走访某个迭代器区间（例如逆向拷贝某范围内的元素）**

**Random Access Iterator: 前四种迭代器都只供应一部分指针算术能力（前三种支持operator++，第四种再加上operator--），第五种则涵盖所有指针的算术能力，包括p+n,p-n,p1-p2,p1<p2.**

**注意：STL算法的一个命名规则：以算法所能接受的最低阶迭代器类型来为其迭代器型别参数命名**



struct input\_iterator\_tag {};

struct output\_iterator\_tag {};

struct forward\_iterator\_tag : public input\_iterator\_tag {};

struct bidirectional\_iterator\_tag : public forward\_iterator\_tag {};

struct random\_access\_iterator\_tag : public bidirectional\_iterator\_tag {};

// iterator 模板

template <class Category, class T, class Distance = ptrdiff\_t,

class Pointer = T \* , class Reference = T & >

struct iterator

{

typedef Category iterator\_category;

typedef T value\_type;

typedef Pointer pointer;

typedef Reference reference;

typedef Distance difference\_type;

};

## 4. construct.h

// 这个头文件包含两个函数 construct，destroy

// construct : 负责对象的构造

// destroy : 负责对象的析构

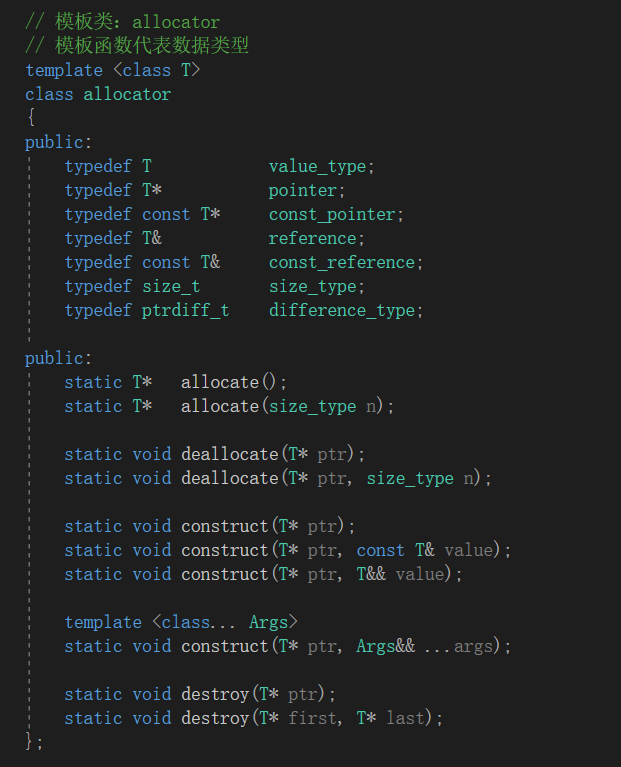
## 5. util.h

这个文件包含一些通用工具，包括 move, forward, swap,make\_pair 等函数，以及 pair 等

以及运算符==,<,>,!=,<=,>=的重载

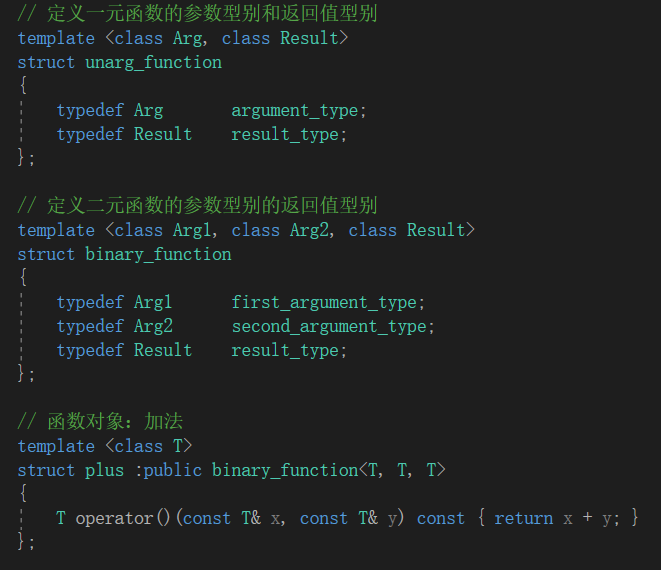
## 6. allocator.h

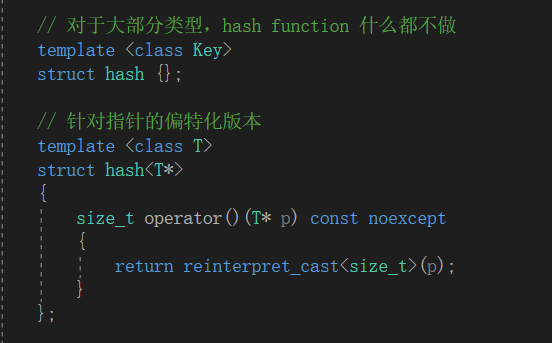
这个头文件包含一个模板类 allocator，用于管理内存的分配、释放，对象的构造、析构



## 7. functional.h

这个头文件包含了 myTinySTL 的函数对象与哈希函数





## 8. algobase.h

这个头文件包含了 myTinySTL 的基本算法,如max,min,

iter\_swap(将两个迭代器所指对象对调)

copy: 把 [first, last)区间内的元素拷贝到 [result, result + (last - first))内

copy\_backward:将 [first, last)区间内的元素拷贝到 [result - (last - first), result)内

// copy\_if

// 把[first, last)内满足一元操作 unary\_pred 的元素拷贝到以 result 为起始的位置上

// copy\_n

// 把 [first, first + n)区间上的元素拷贝到 [result, result + n)上

// 返回一个 pair 分别指向拷贝结束的尾部

// move

// 把 [first, last)区间内的元素移动到 [result, result + (last - first))内

// move\_backward

// 将 [first, last)区间内的元素移动到 [result - (last - first), result)内

// equal

// 比较第一序列在 [first, last)区间上的元素值是否和第二序列相等

// fill\_n

// 从 first 位置开始填充 n 个值

// fill

// 为 [first, last)区间内的所有元素填充新值

// lexicographical\_compare

// 以字典序排列对两个序列进行比较，

// mismatch

// 平行比较两个序列，找到第一处失配的元素，返回一对迭代器，分别指向两个序列中失配的元素

## 9. uninitialized.h

// 这个头文件用于对未初始化空间构造元素

// uninitialized\_copy

// 把 [first, last) 上的内容复制到以 result 为起始处的空间，返回复制结束的位置

// uninitialized\_copy\_n

// 把 [first, first + n) 上的内容复制到以 result 为起始处的空间，返回复制结束的位置

// uninitialized\_fill

// 在 [first, last) 区间内填充元素值

// uninitialized\_fill\_n

// 从 first 位置开始，填充 n 个元素值，返回填充结束的位置

// uninitialized\_move

// 把[first, last)上的内容移动到以 result 为起始处的空间，返回移动结束的位置

// uninitialized\_move\_n

// 把[first, first + n)上的内容移动到以 result 为起始处的空间，返回移动结束的位置

## 10. memory.h

// 这个头文件负责更高级的动态内存管理

// 包含一些基本函数、空间配置器、未初始化的储存空间管理，以及一个模板类 auto\_ptr

## 11. deque.h

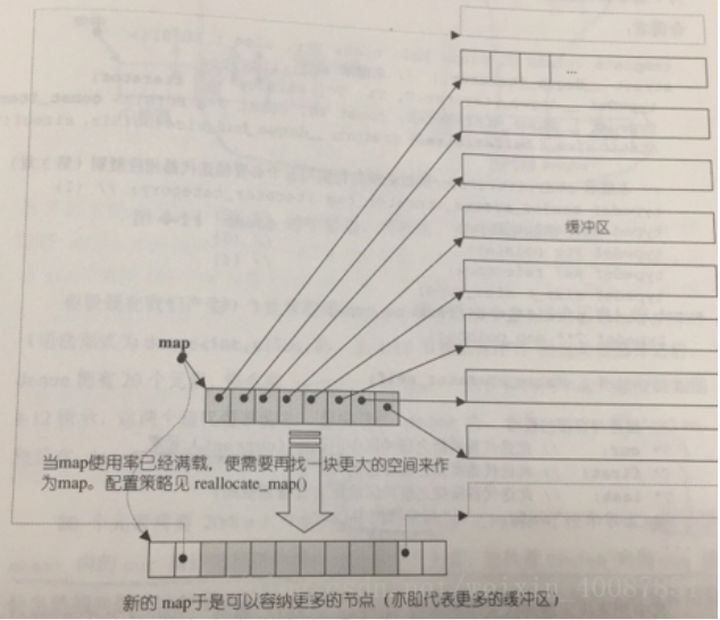
1）deque概述

* **vector和deque的区别**
  + vector是单向开口的连续线性空间，deque是双向开口的连续线性空间。



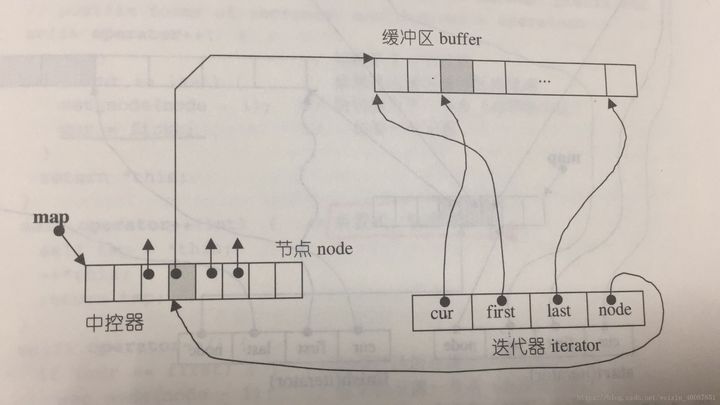
* + deque允许常数时间内对起头端进行元素的插入或移除操作。vector需要将起头端之后的元素进行整体的前移或后移操作，时间复杂度为O(N)，效率极差。
  + deque没有所谓的容量概念，它是**动态地以分段连续空间**组合而成，随时可以增加一段新的空间并链接起来（随后会介绍）。
* **deque的中控器**
  + deque采用一块所谓的map（不是map容器）作为主控。这里所谓的map是一小块连续空间，其中每个元素都是指针，指向另一段（较大的）连续线性空间，称为缓冲区。缓冲区才是deque的存储空间主体。
  + deque的连续空间其实是假象，是由一段一段的定量连续空间构成。

2）deque的中控器

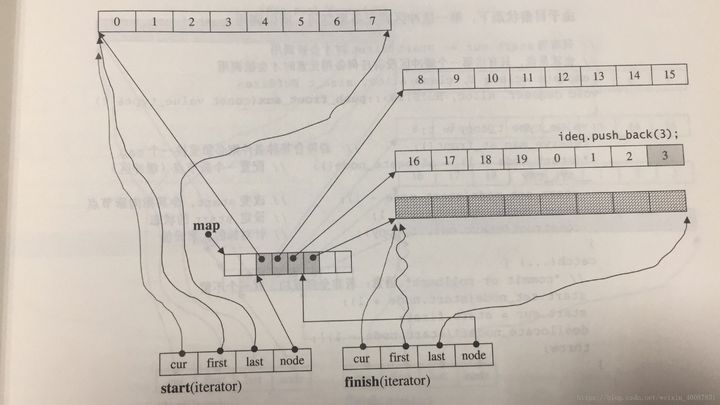


1. deque的迭代器

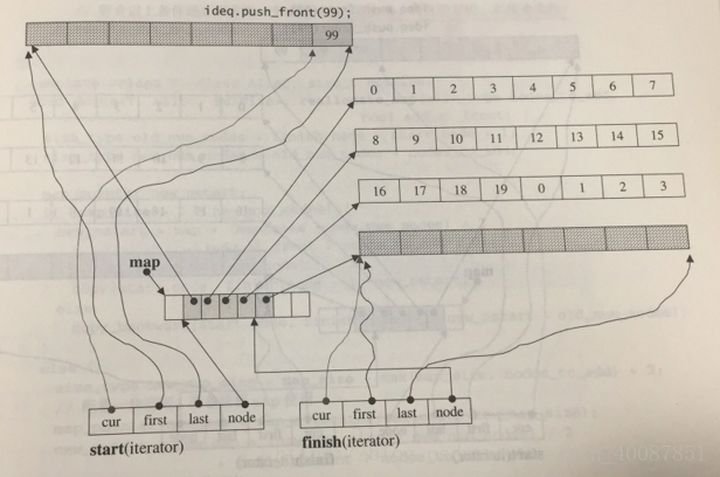
* deque的迭代器包含四个指针
  + cur：指向缓冲区中的现行元素
  + first：指向缓冲区的头
  + last：指向缓冲区的尾
  + node：指向管控中心
  + deque的中有一个start迭代器和一个finish迭代器，分别指向数据头和数据尾
* 当map所指的缓冲区容量满载时，需要对map进行扩充，**扩充过程**：配置更大的，拷贝原来的，释放原来的。因此deque的迭代器会失效。
* deque的中控器、缓冲区、迭代器的相互关系



* deque在尾端添加元素，引发新缓冲区的配置（条件：map中控器未满载，不需要考虑map）。

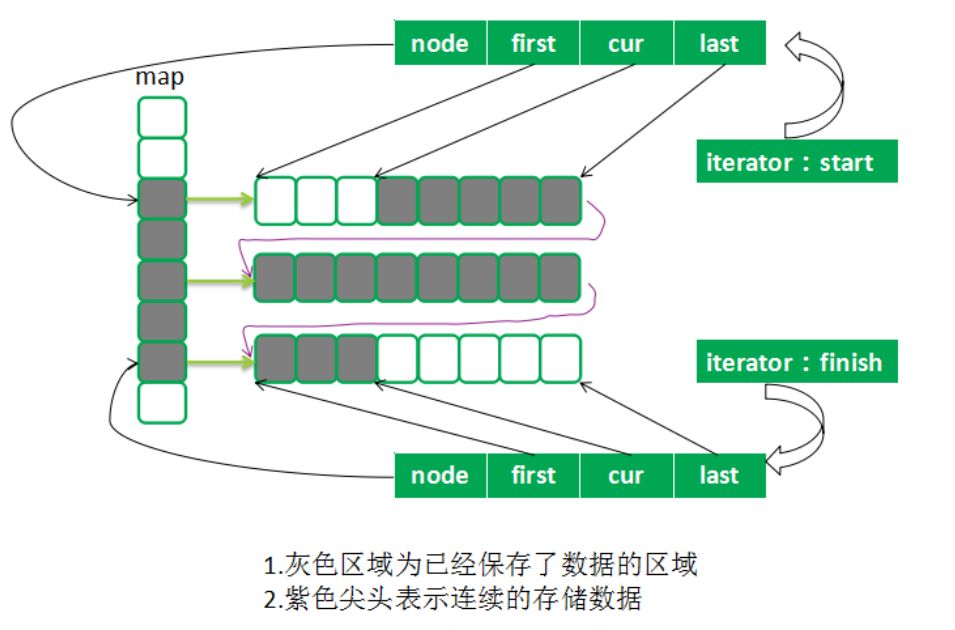


* deque在最前端添加元素，引发新缓冲区的配置（条件：map中控器未满载，不需要考虑map）。



* **map满载时，扩充map（配置更大的，拷贝原来的，释放原来的）。**
* **deque优缺点**
  + 可随机存取。时间复杂度为O(1)。可在头部和尾部进行插入操作，相对于vector的头插操作效率较高。
  + 插入和删除操作时间复杂度为O(N)（头插和尾插时间复杂度为O(1)，头删和尾删时间复杂度为O(1)）。若插入位置之前的元素个数较少，则将之前的元素前移，反之将之后的元素后移。删除操作类似插入操作。

Deque的元素操作



//在尾端插入、删除，在前端插入、删除

Void push\_back();

Void pop\_back();

Void push\_front();

Void push\_back();

//clear（）用来清空整个Deque，回到最初状态（无任何元素时），保留一个缓冲区。

Template<class T, class Alloc, size\_t BufSize>

void deque<T,Alloc,BufSize>:: clear() {…}

//清除pos所指的元素,pos为清除点

Iterator erase (iterator pos) {…}

//清除[first,last ) 区间内的所有元素

Template<class T, class Alloc, size\_t BufSize>

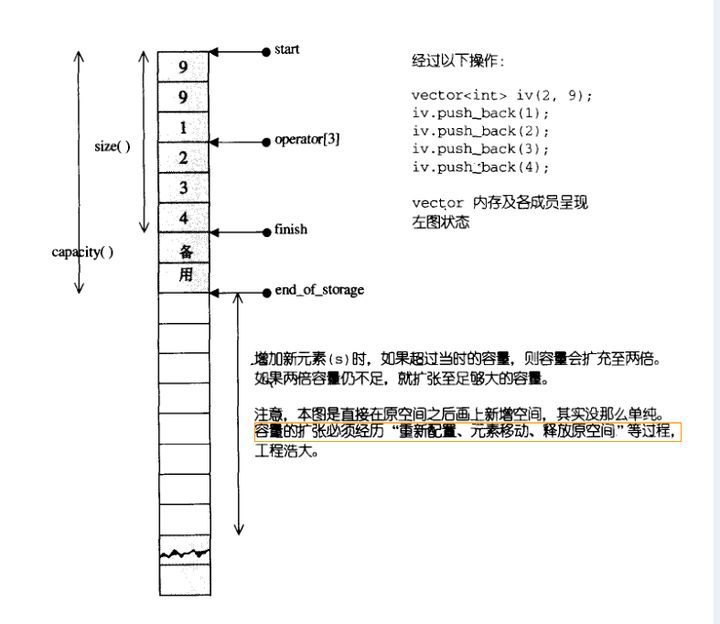
deque<T,Alloc,BufSize>::iterator deque<T,Alloc,BufSize>::erase(iterator first,iterator last) { …}

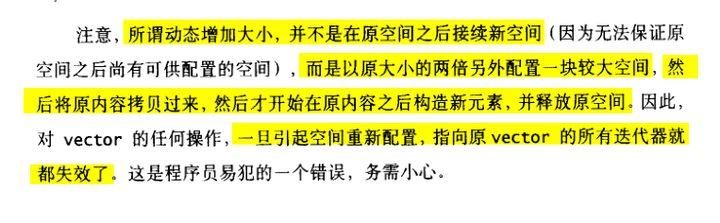
//在position处插入一个元素，其值为x

Iterator insert(iterator position, const value\_type& x) { …}

## 12. vector.h

* **vector的数据结构**
  + 数组。
* **vector和array的区别**
  + array是**静态空间**，一旦配置了就不能改变；要存入大于空间个数的元素，需要客户端自己实现。
  + vector是**动态空间**，随着元素的加入，它的内部机制会自行扩充以容纳新元素。是经过“**重新配置空间（原来长度的2倍），拷贝原空间内容，释放原空间**”完成的。指向原vector的所有迭代器都会失效。
  + 两者均为线性连续空间。





**Vector声明：**

template <class T, class Alloc = alloc>

class vector

{

public:

// 类型相关定义

typedef T value\_type;

typedef value\_type\* pointer;

typedef value\_type\* iterator;

typedef value\_type& reference;

typedef size\_t size\_type;

typedef ptrdiff\_t difference\_type;

protected:

//定义配置器

typedef simple\_alloc<value\_type, Alloc> data\_allocator;

iterator start; // 内存起始地址

iterator finish; // 当时使用内存的末尾地址，每次插入和删除都会修改

iterator end\_of\_storage; // 内存的结束地址

// 关键函数，在某个位置插入一个数据

void insert\_aux(iterator position, const T& x);

// 使用配置器释放内存

void deallocate()

{

if (start)

data\_allocator::deallocate(start, end\_of\_storage - start);

}

// 申请并初始化一块大小为n的内存，并初始化为value

void fill\_initialize(size\_type n, const T& value)

{

start = allocate\_and\_fill(n, value);

finish = start + n;

end\_of\_storage = finish;

}

public:

// 迭代器起始位置

iterator begin() { return start; }

// 迭代器结束位置

iterator end() { return finish; }

// 容器大小，即真实的数据个数

size\_type size() const { return size\_type(end() - begin()); }

// 容器容量，即申请的内存大小

size\_type capacity() const { return size\_type(end\_of\_storage - begin()); }

// 容器是否为空

bool empty() const { return begin() == end(); }

// 重载[]运算符，取出对应position的数据，下标从0开始

reference operator[](size\_type n) { return \*(begin() + n); }

// 构造函数

vector() : start(0), finish(0), end\_of\_storage(0) {}

vector(size\_type n, const T& value) { fill\_initialize(n, value); }

vector(int n, const T& value) { fill\_initialize(n, value); }

vector(long n, const T& value) { fill\_initialize(n, value); }

explicit vector(size\_type n) { fill\_initialize(n, T()); }

// 析构函数

~vector()

{

destroy(start, finish);

deallocate();

}

// 取出起始数据

reference front() { return \*begin(); }

// 取出末尾数据

reference back() { return \*(end() - 1); }

// 从尾部插入一个数据

void push\_back(const T& x)

{

if (finish != end\_of\_storage)

{

// 内存没有满，直接插入

construct(finish, x);

++finish;

}

else

{

// 内存满了，需要扩容内存，然后插入数据

insert\_aux(end(), x);

}

}

// 弹出最后一个数据

void pop\_back()

{

--finish;

destroy(finish);

}

// 删除时，将后面的数据覆盖前面的数据，然后释放最后一个数据；如果删除的数据是最后一个数据，那么直接

// 释放即可

iterator erase(iterator position)

{

if (position + 1 != end())

// 将position + 1到finish的数据，拷贝到position开始的地方

copy(position + 1, finish, position);

--finish;

destroy(finish);

return position;

}

// 修改vector的大小，新的size比老的size小，直接删除多余的数据；新的size比老的size大，直接插入

void resize(size\_type new\_size, const T& x)

{

if (new\_size < size())

erase(begin() + new\_size, end());

else

insert(end(), new\_size - size(), x);

}

// 外部统一调用接口，一层封装

void resize(size\_type new\_size) { resize(new\_size, T()); }

// 删除容器中所有数据，不会释放内存

void clear() { erase(begin(), end()); }

protected:

// 申请并初始化一块内存

iterator allocate\_and\_fill(size\_type n, const T& x)

{

iterator result = data\_allocator::allocate(n);

uninitialized\_fill\_n(result, n, x);

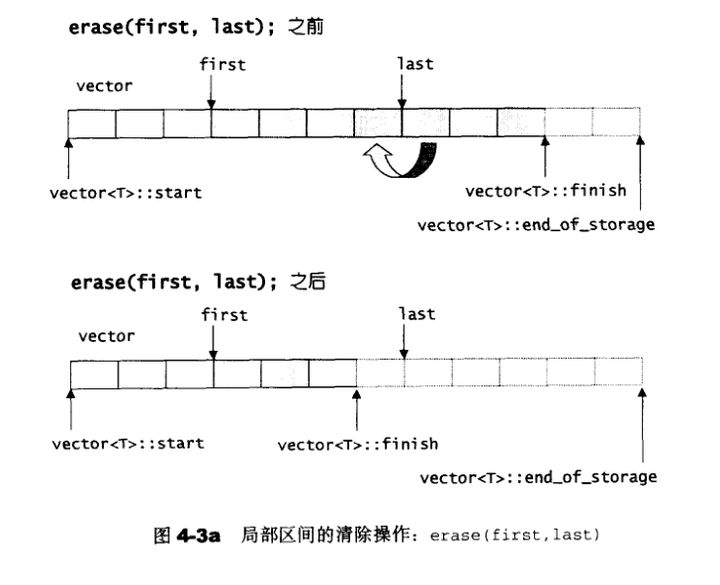
return result;

}

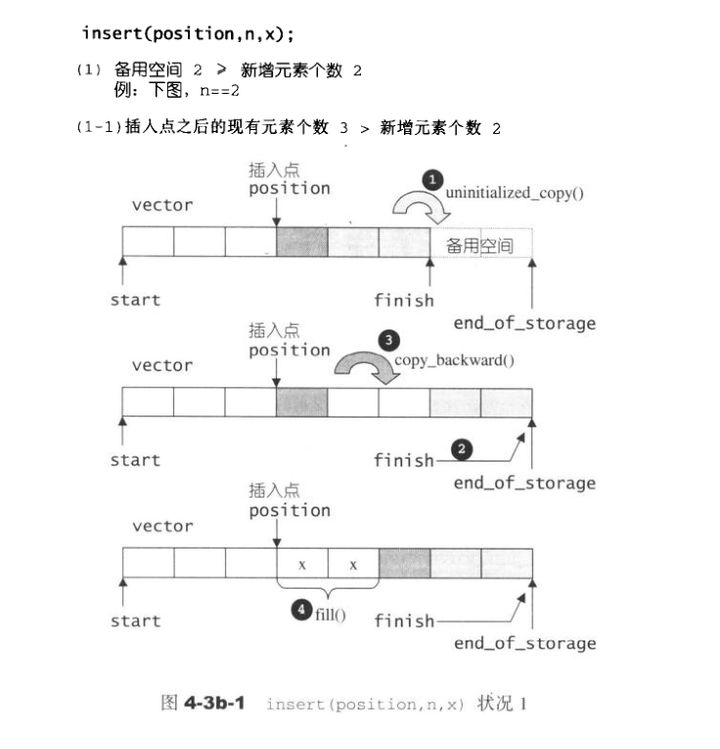
}

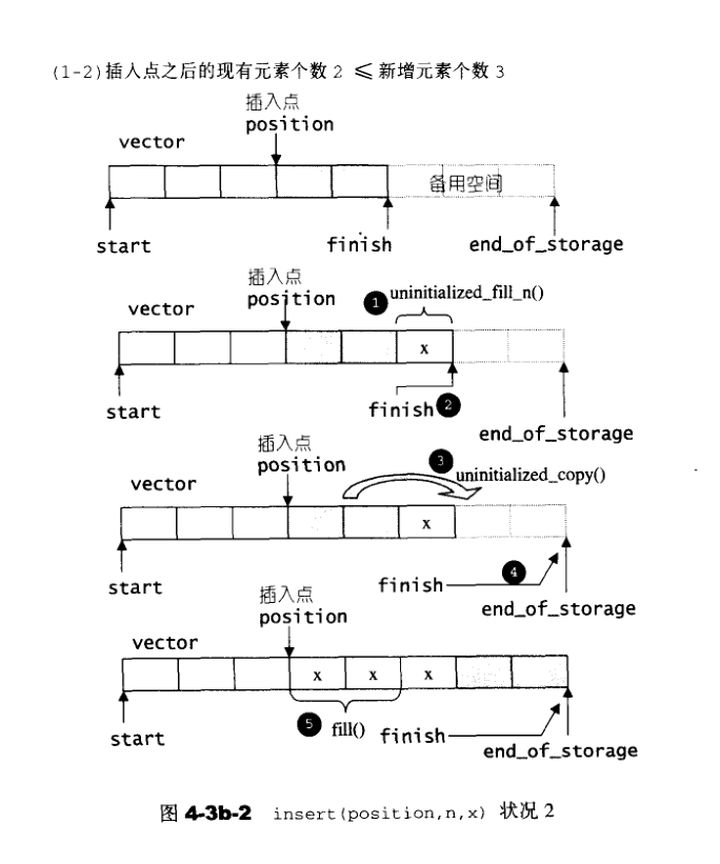
erase方法

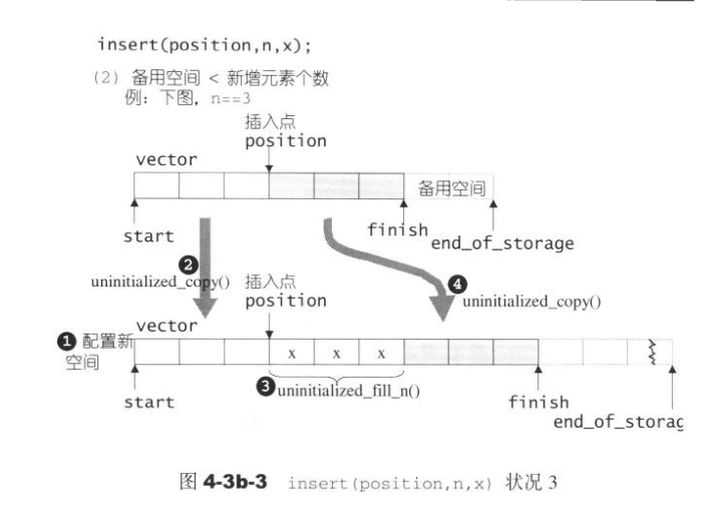




Insert方法：







常用方法：front/back/push\_back/pop\_back/erase/clear/size/insert(不建议使用，效率低)

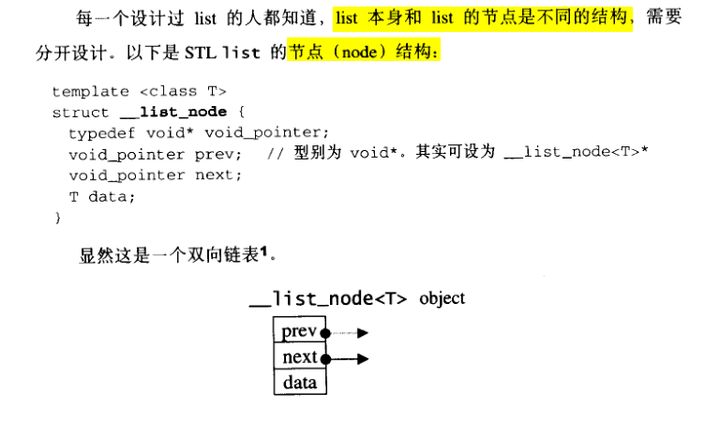
## 13. list.h 双向链表

1) List概述

**每次插入或删除一个元素，就配置或释放一个元素空间，而且永远是常数时间。**List不再能够想vector一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在。List的插入（insert）操作和接合（splice）操作都不会造成原有的list迭代器失效，这在vector是不成立的，因为vector的插入操作可能造成记忆体重新配置，导致原有的迭代器全部失效。

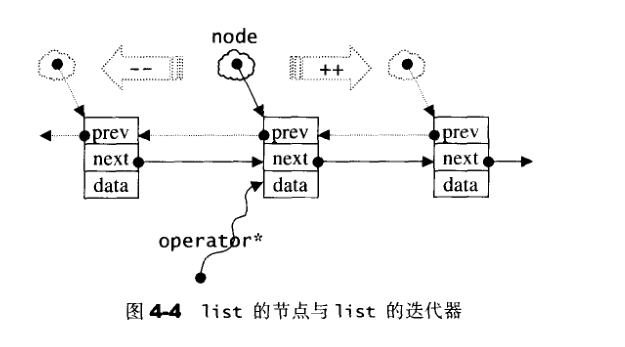
1. list节点

每次配置一个节点大小，当元素删除时，相应空间一并删除。

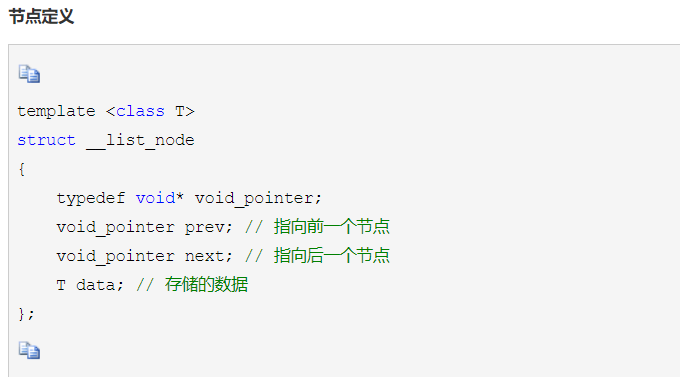


1. list迭代器

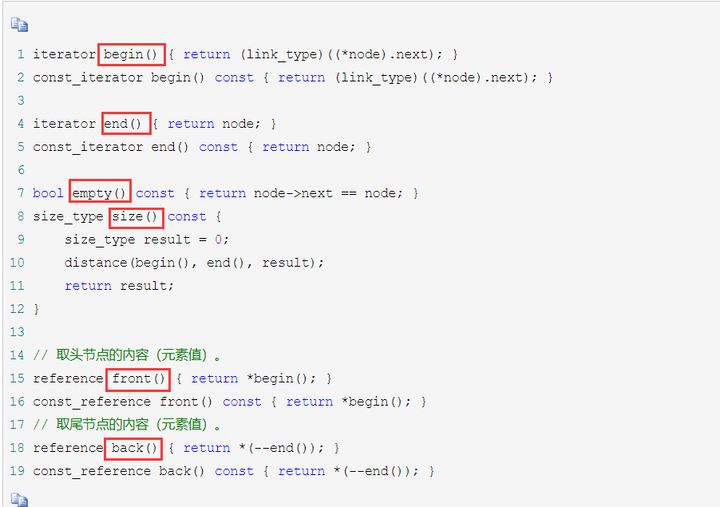
* list不能像vector一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在。
* list是一个环状双向链表，迭代器必须具备前移、后移的能力，所以list提供的是双向迭代器。
* 插入操作和接合操作都不会造成原有的list迭代器失效，只有在删除元素时，“指向被删除元素”的那个迭代器失效，其他迭代器不受影响。
* list的迭代器应该支持正确的递增、递减、取值、成员取用等操作。递增时指向下一个节点，递减时指向上一个节点，取值时，取得是节点的值，成员取用时取用的是节点的成员。
* 除此之外，list 还被设计成一个循环链表，**表尾加上一个空白的节点**，使其符合“前闭后开”的规则，这样就只需要一个标记，就可以完整的表示整个链表。

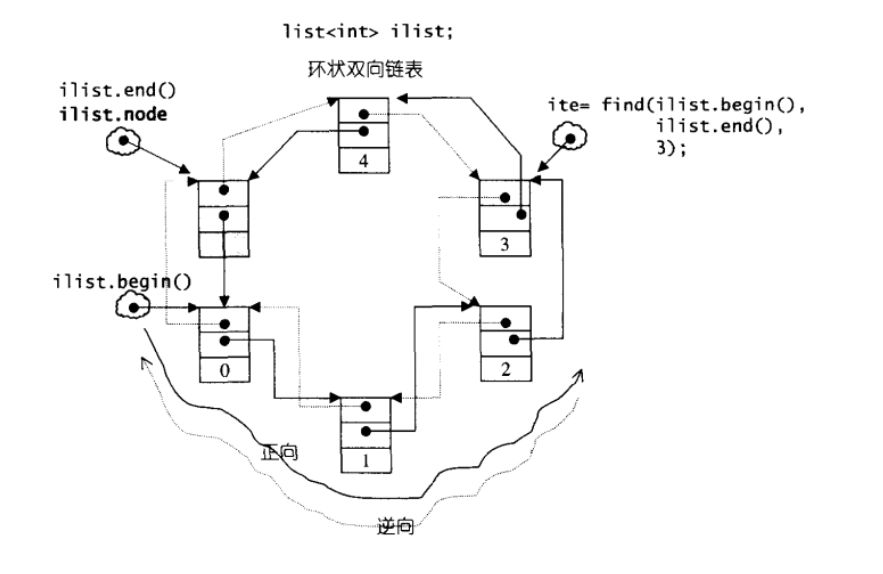


1. list的数据结构



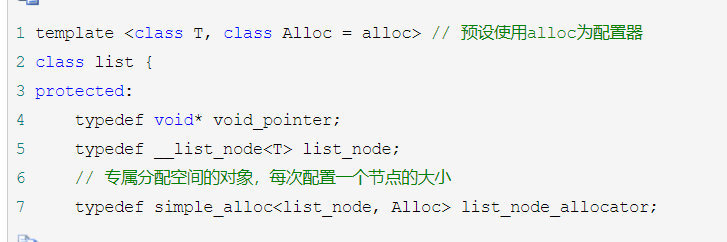




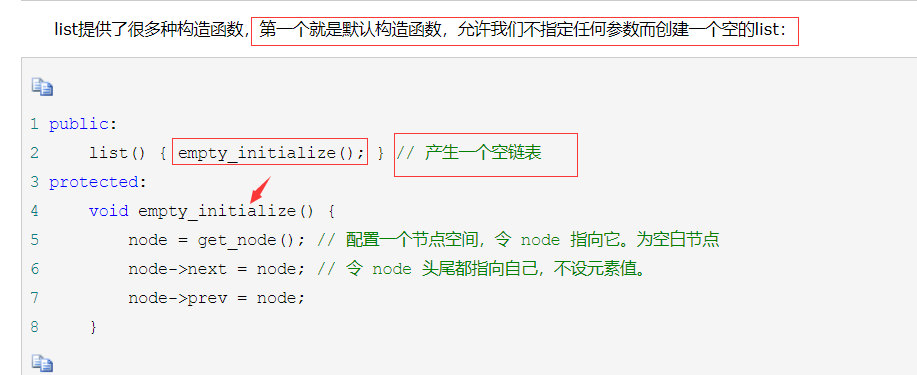


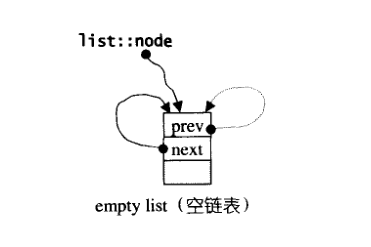
1. list的构造与内存管理

与Vector类似，list也定义了一个simple\_alloc<>对象来负责空间配置事宜，其**第一个类型参数为结点类\_\_list\_node<T>**，为的是每次都分配一个节点大小的空间，**第二个类型参数为alloc空间配置器**。

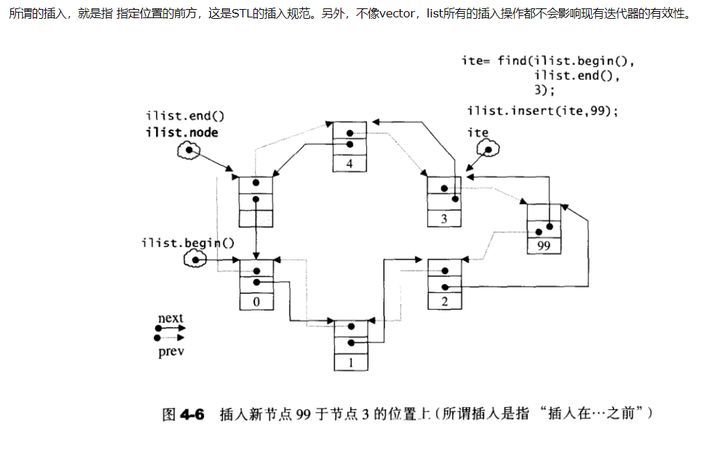












1. list的元素操作

//插入一个节点，作为头结点

Void push\_front(const T& x) { insert(begin(),x);}

//插入一个节点，作为一个尾节点

Void push\_back(const T& x) { insert(end(),x); }

//移除迭代器position所指节点

Iterator erase (iterator position){…}

//移除头结点

Void pop\_front() { erase(begin()); }

//移除尾结点

Void pop\_back() { iterator tmp = end(); erase(--tmp); }

//清除所有节点

Template <class T,class Alloc>

Void list<T,Alloc>:: clear(){…}

//将数值为value之所有元素移除

Template<class T, class Alloc>

Void list<T,Alloc>::remove(const T& value) {…}

//移除数值相同的连续元素。注意，只有“连续而相同的元素”才会被移除剩下一个

Template<class T, class Alloc>

Void list<T,Alloc> :: unique() {…}

//迁移操作：将某连续范围的元素迁移到某个特定位置之前。将[first,last ) 内的所有元素移动到position之前

Void transfer (iterator position, iterator first, iterator last) {…}

//将X接合与position所指位置之前。X必须不同于\*this

Void splice(iterator position,list& x) {…}

//将i所指元素接合于position所指位置之前。Position和i可指向同一个list

Void splice(iterator position, list&, iterator i) {…}

//将[first,last ) 内的所有元素接合于position所指位置之前

//position和[first,last )可指向同一个list

//但position不能位于[first,last ) 之内

Void splice(iterator position, list&, iterator first, iterator last) {…}

//merge()将X合并到\*this身上。两个list必须经过递增排序

Template<class T, class Alloc>

Void list<T,Alloc>:: merge(list<T,Alloc>& x){…}

//reverse（）将\*this的内容逆向重置

Template<class T, class Alloc>

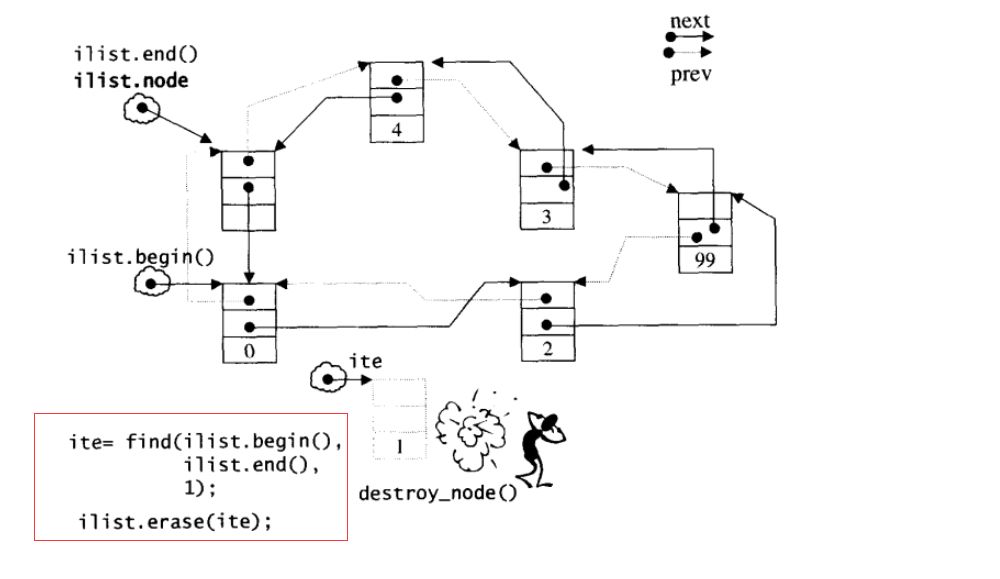
Void list<T,Alloc>:: reverse(){..}

//list不能使用STL算法的sort（），必须使用自己的sort() member function

//因为STL算法sort（）只接受RandomAccessIterator

Template<class T, class Alloc>

Void list<T,Alloc>:: sort(){ …}



## 14. queue.h

// 这个头文件包含了两个模板类 queue 和 priority\_queue

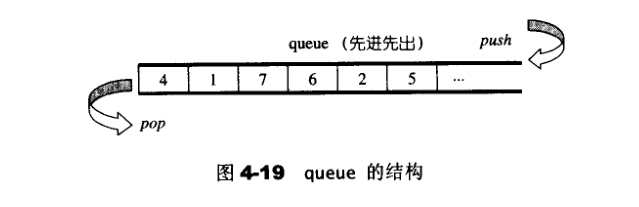
// queue : 队列

// priority\_queue : 优先队列

1) queue概述

Queue 是一种**先进先出**（FIFO）的数据结构，它只有一个出口。（以deque 或者 list作为底层容器，封闭其头端入口，尾端开口即可）。它允许新增元素，移除元素，从最底端加入元素，取得最顶端元素。**Queue同样没有迭代器，不允许遍历行为**。

将元素推入的操作称为push，将元素退出的操作称为pop。



2）元素操作

bool empty() const {…}

size\_type size() const {…}

reference front() {…}

reference back() {…}

void push (const value\_type& x){…}

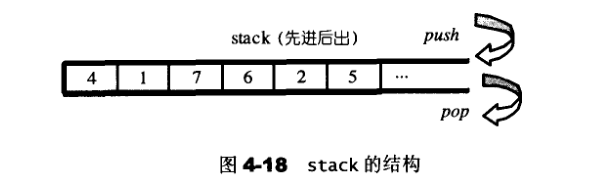
void pop {…}

## 15. stack.h

1） stack概述

Stack是一种**先进后出**(FILO)的数据结构，它只有一个出口。（以deque 或者 list作为底层容器，封闭其头端开口即可）。Stack允许在最顶端新增元素，移除元素，取得最顶端元素的值。**Stack没有迭代器，其不允许遍历行为**。

将元素推入的操作称为push，将元素退出的操作称为pop。



2）元素操作

bool empty() const {…}

size\_type size() const {…}

reference top() {…}

void push (const value\_type& x){…}

void pop {…}

## 16. heap概述

1）heap概述

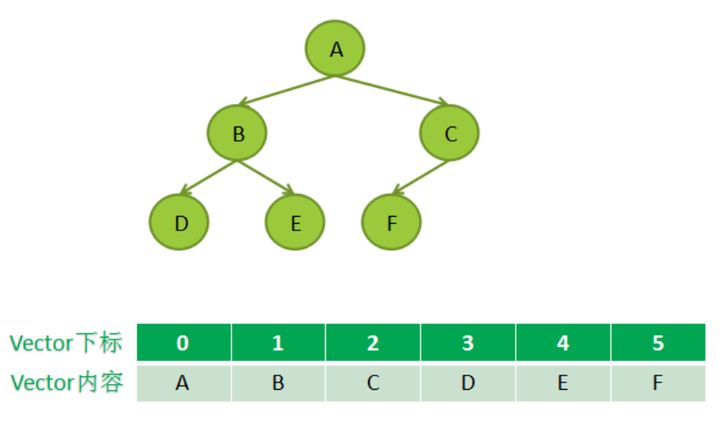
heap并不是stl中的一种容器，它实际是以**vector**作为完全二叉树的存储结构，再加上一些heap算法。它**是priority\_queue的底层实现（priority\_queue允许用户以任何次序将任何元素推入容器内，但取出时一定是从优先权最高也就是数值最高的元素开始取）**。**binary heap**：一种完全二叉树。可以用**数组（array）**表示。如果数组的**第一个元素保留**（设为无穷大或者无穷小），从第二个元素开始储存，则有：对于i处的节点，其子节点必定位于2i（左子节点）和2i+1（右子节点）处，其父节点位于i/2处。Heap操作包括建堆，push\_heap,pop\_heap，sort\_heap等。Heap的所有元素必须遵循特别的排序规则，所以不提供遍历功能。

Heap 分为 **max-heap**：每个节点的键值（key）都大于或等于其子节点的键值。因此max-heap的最大值在根节点，并总是位于底层vector或者array的起头处。

**min-heap**：每个节点的键值（key）都小于或等于其子节点的键值.。因此min-heap的最小值在根节点，并总是位于底层vector或者array的起头处。

介绍

* 二叉树
  + 二叉树是每个结点最多有两个子树的树结构
* 完全二叉树
  + 完全二叉树是一种特殊的二叉树
  + 定义：在一棵二叉树中，除了最后一层外，其余每一层的节点数都是满的
* 满二叉树
  + 满二叉树是一种特殊的完全二叉树
  + 定义：在一棵二叉树中，每一层的节点数都是满的
* 堆
  + 堆是一种特殊的完全二叉树
  + 父节点的值大于(小于)所有子节点的值
* 最大堆：
  + 最大堆是堆的一种
  + 父节点的值大于所有子节点的值
  + 根节点的值最大
* 最小堆
  + 最小堆是堆的一种
  + 父节点的值小于所有子节点的值
  + 根节点的值最小
* 堆的存储形式：
  + 节点存储，类似链表，每一个节点有两个指针，分别指向两个子节点
  + vector存储：类似数组，从上到下，从左到由，一次将堆中的数据保存vector中即可，具备以下性质
    - 前提：根节点下标是1
    - 节点i的左子节点是2i，右子节点是2i+1
    - 节点i的父节点是i/2
* 堆的存储如图所示：



2）heap算法

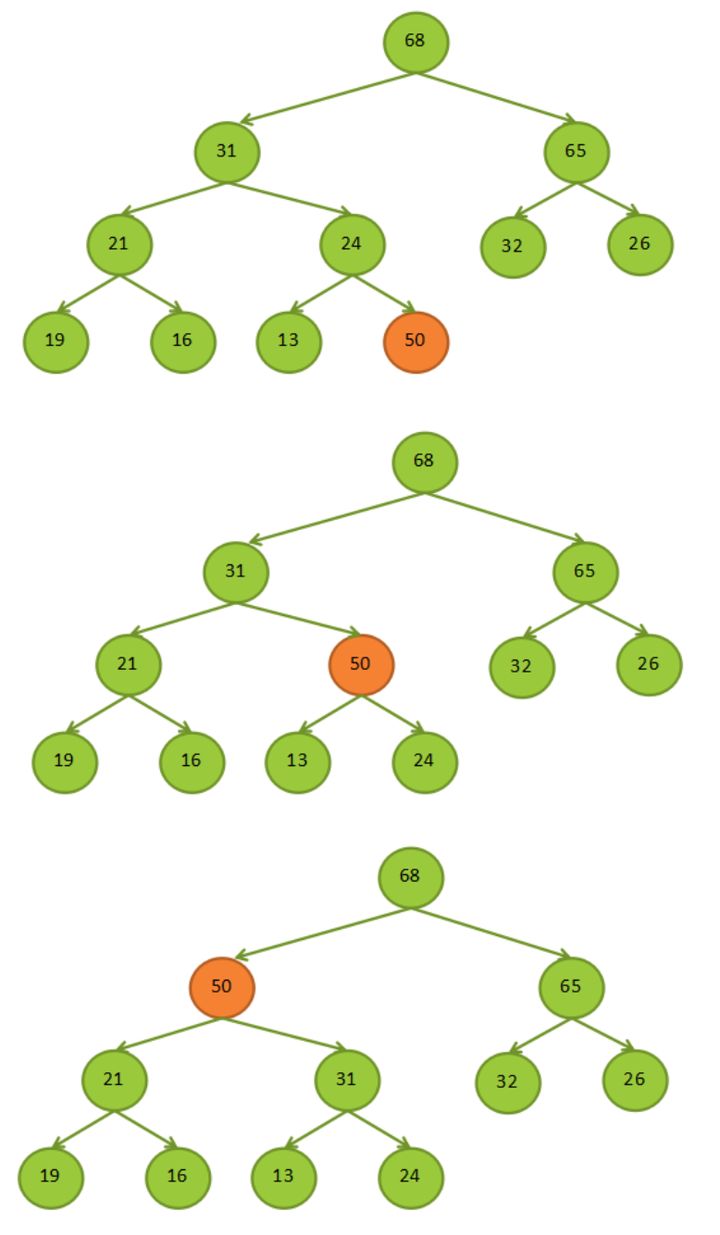
* push\_heap算法

template<class RandomAccessIterator>

inline push\_heap(RandomAccessIterator first, RandomAccessIterator last){…}

* 最大堆插入
  + 第一步：将需要插入的数据放在最下一层的最后一个位置
  + 第二步：将插入的节点和其父节点比较，**如果比父节点大**，就和父子节点**互换**，一直向上比较，直到根节点或是父节点比子节点大

下图是将50插入堆的流程：

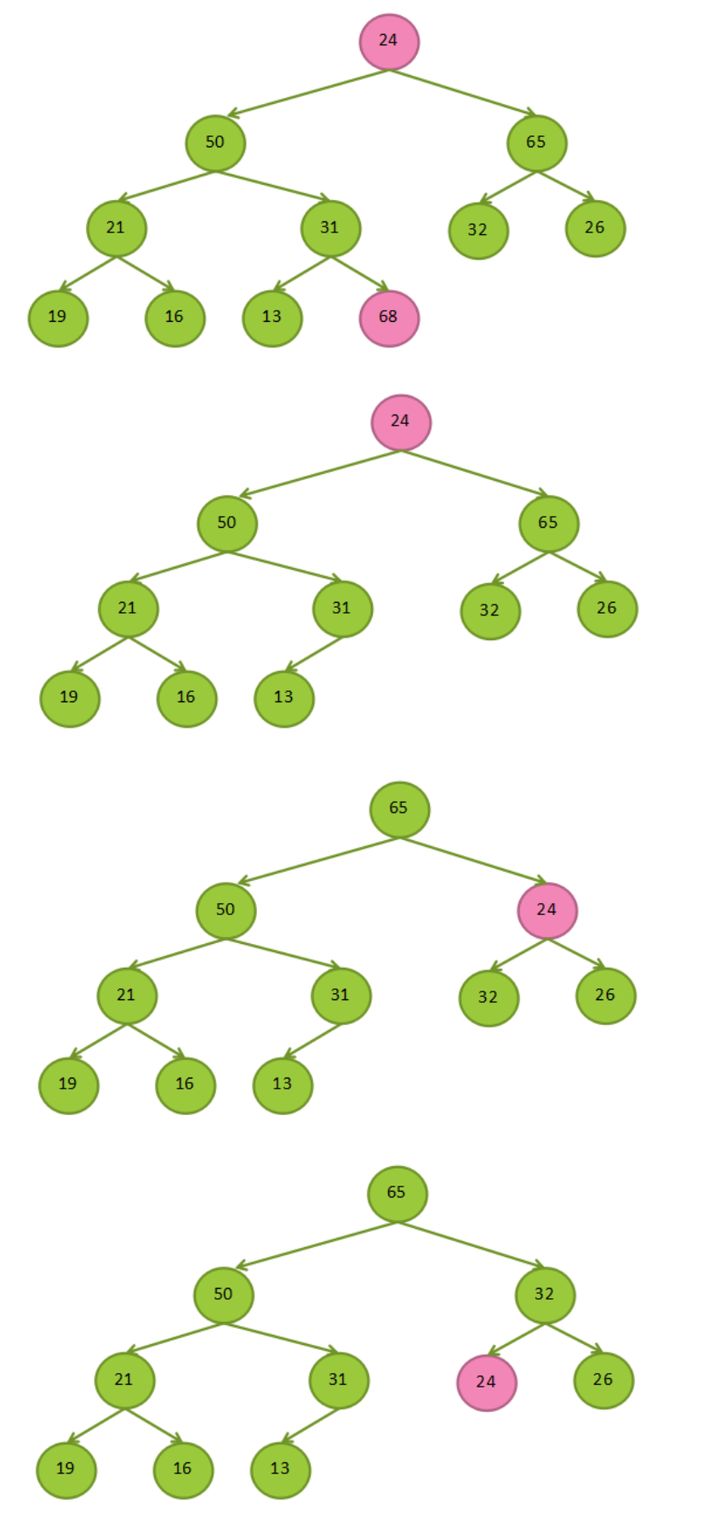


* pop-heap算法

template<class RandomAccessIterator>

inline pop\_heap(RandomAccessIterator first, RandomAccessIterator last){…}

* 最大堆删除
  + 堆的删除**只能是删除根节点**，不能删除非根节点
  + 第一步：互换根节点和最下面一层最右边的节点
  + 第二步：删除最下面一层最右边的节点(原来的根节点)
  + 第三步：将根节点与较大的子节点互换，依次向下处理，直到叶子节点，或比左右两个子节点都大
  + 下图是将根节点68删除的流程：



* sort\_heap算法
* 从堆中取出的值，每次都是最大值或是最小值，如果依次取出堆中的值，只到堆为空，将取出的值依次排列，这就是堆排序。**对一个heap进行sort\_heap操作，可以得到一个递增的序列**。

Template<class RandomAccessIterator>

Void sort\_heap(RandomAccessIterator first, RandomAccessIterator last) {

While(last -first > 1)

Pop\_heap(first,last--);

}

* make\_heap算法

作用：用于将容器中的数据，按照堆的规则进行调整，转化为一个heap

Template <class RandomAccessIterator>

Inline void make\_heap(RandomAccessIteator first, RandomAccessIterator last){….}

* 使用实例 #<include algorithm>

如：int ia[9]={0,1,2,4,5,6,7,8,12};

vector<int>ivec(ia,ia+9);

make\_heap(ivec.begin(), ivec.end() ); //得到一个max-heap

ivec.push\_back(3);

push\_heap(ivec.begin(), ivec.end() ); //加入元素

pop\_heap(ivec.begin(), ivec.end() );//取出最顶端元素

sort\_heap(ivec.begin(), ivec.end() );//排序，从大到小

for(int i=0;i<ivec.size();i++){

cout<<vec[i]<<” “;}

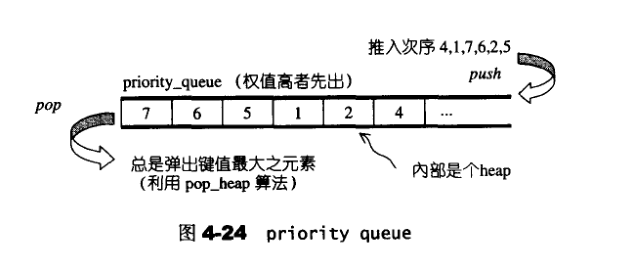
## 17. priority\_queue

1）概述

priority\_queue是一种拥有权值概念的queue，它允许加入新元素、移除新元素，审视元素值等功能。由于这是一个queue，所以**只允许在底端加入元素，在顶端取出元素**。因此取出元素的时候只能取出权值最大的元素。priority\_queue 带有权值观念，**其内的元素是自动按照元素的权值排列**（通常权值以实值表示），权值最高者排在最前面。

缺省情况下，priority\_queue 利用一个max-heap完成，max-heap是以vector表现的完全二叉树。priority\_queue也是container adapter，它以vector为存储结构，再以heap算法进行处理：priority\_queue的元素存放在一个vector当中，按元素权值排列成一个heap。priority\_queue也没有迭代器，不能遍历。

Adapter 是即配接器，是指“修改某物接口，形成另一种风貌”的接口。



2）元素操作

Template<class InputIterator>

Priority\_queue(InputIterator first, InputIterator last, const compare& x) {…}

Template<class InputIterator>

Priority\_queue(InputIterator first, InputIterator last) {…}

Bool empty() const {…}

Size\_type size() const {…}

Const reference top() const {…}

Void push (const value\_type& x) {…}

Void pop () {…}

3）使用实例

#include <queue>

int ia[9]={0,1,2,4,5,6,7,8,12};

priority\_queue<int>ipq(ia,ia+9);

cout<<”size= “<<ipq.size()<<endl;

cout<<”top = “<< ipq.top()<<endl;

ipq.push(3);

while(!ipq.empty()){

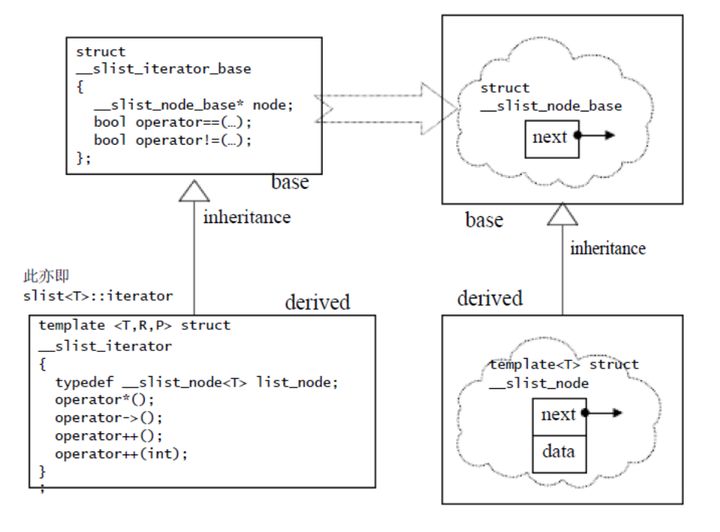
cout<<ipq.top()<<” “;

ipq.pop(); }

## 18. slist单向链表

1）概述

* 单向链表。它的插入，删除，接合等操作不会造成原有迭代器失效。
* 不在标准STL的范围内，可以学习
* 相比list，slist有如下特点：
  + slist的插入是在**插入点之后插入**，不是list的在插入点之前插入
  + slist消耗的空间更小，操作的速度更快
* slist的迭代器是forward iterator迭代器，而list是Bidirectional iterator
* 类似list中设计，slist也有一个门卫节点，门卫节点的下一个节点就是链表的头节点
* slist的结构设计如下所示：



2）元素操作

slist() {…};

~slist() {…}

Iterator begin() {…}

Iterator end() {…}

Size\_type size() {…}

Bool empty() {…}

Void swap (slist& sl) {…}

3）使用实例

#include<slist>

slist<int>islist;

islist.push\_front(9);

islist.push\_front(1);

islist.push\_front(2);

islist.push\_front(3);

islist.push\_front(4);

cout<<”size = “<< islist.size()<<endl;

slist<int>::iterator ite = islist.begin();

slist<int>::iterator ite2 = islist.end();

for(;ite!= ite2;++ite){

cout<<\*ite<<” “;} // 4 3 2 1 9

ite = find(islist.begin(),islist.end(),1);

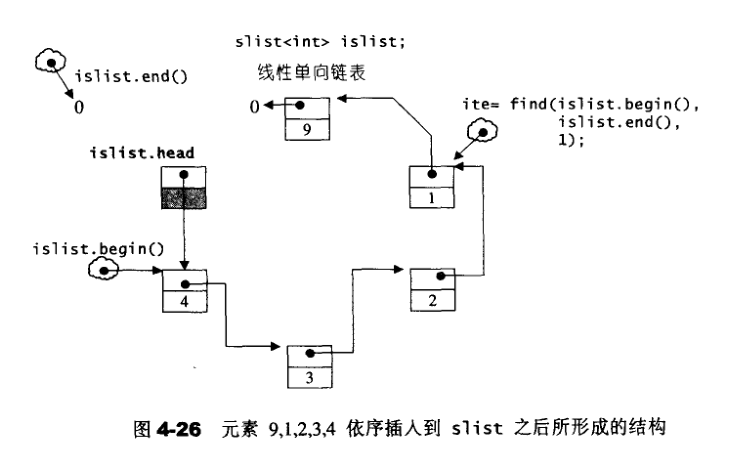
if(ite !=0){

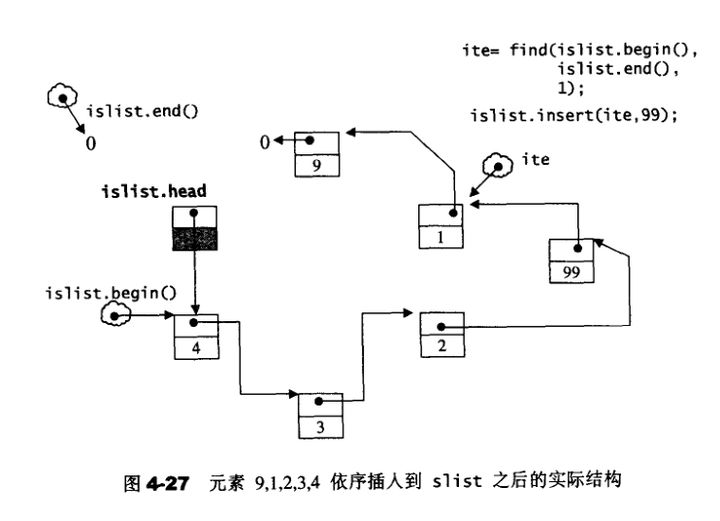
islist.insert(ite,99); } // 4 3 2 99 1 9

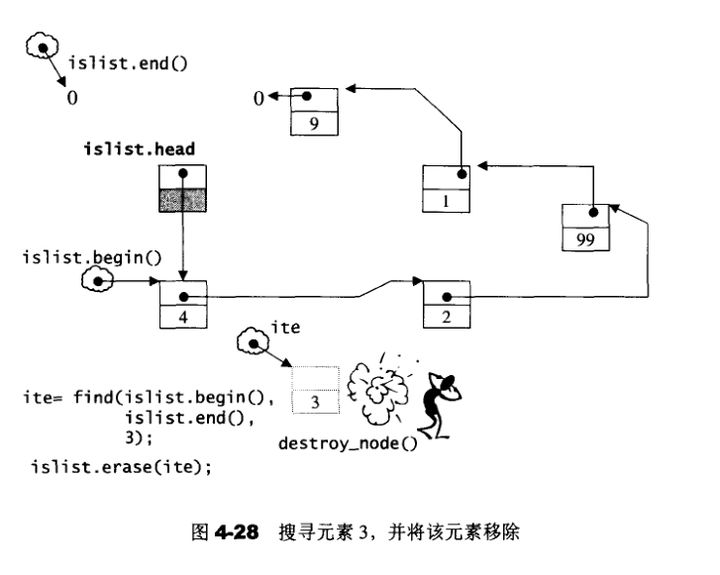
ite = find(islist.begin(),islist.end(),3);

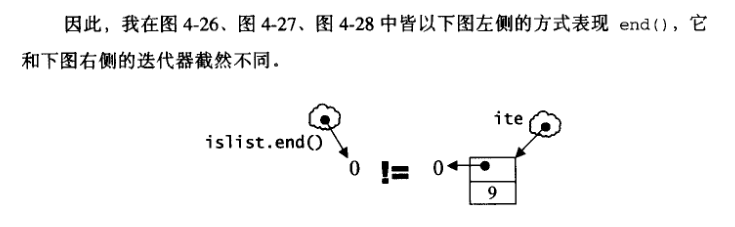
if(ite !=0){

cout<< \*(islise.erase(ite))<<endl; } // 2









## 19. 关联式容器概述

标准的STL关联式容器分为**set（集合）和map（映射表）两大类**，以及这两大类的衍生体multiset（多键集合） 和 multimap（多键映射表），这些**容器的底层机制均以RB-tree（红黑树）完成**，红黑树也是一个独立的容器，但并不开放给外界使用。

SGI STL还提供了一个不在标准规格之列的关联式容器： hash table（散列表），以及hash table为底层机制的hash\_set（散列集合），hash\_map（散列映射表），hash\_multiset（散列多键集合），hash\_multimap（散列多键映射表）

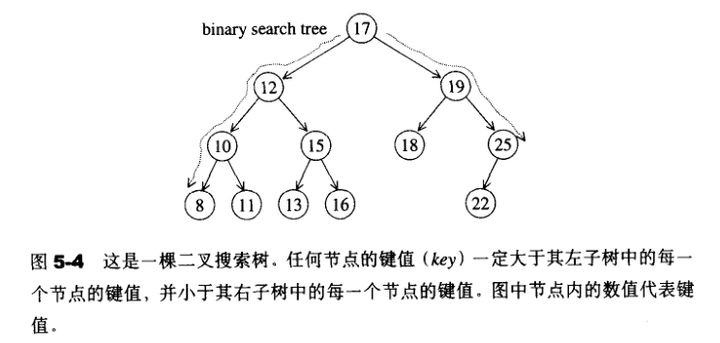
所谓关联式容器：**每笔数据（每个元素）都一个键值（key）和一个实值（value）**。当元素插入到关联式容器时，容器内部结构（可能是RB-tree ，也可能是hash table）便依照其键值大小，以某种特定规则将这个元素置于适当位置。关联式容器没有所谓的头尾。

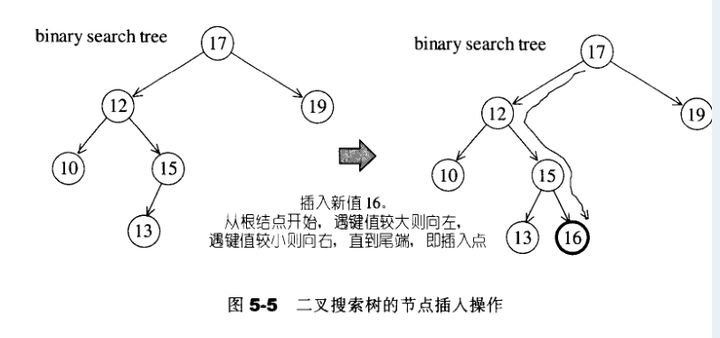
关联式容器的内部结构是一个平衡二叉树（balanced binary tree），以便获得良好的搜寻效率，平衡二叉树包括AVL-tree,RB-tree,AA-tree。

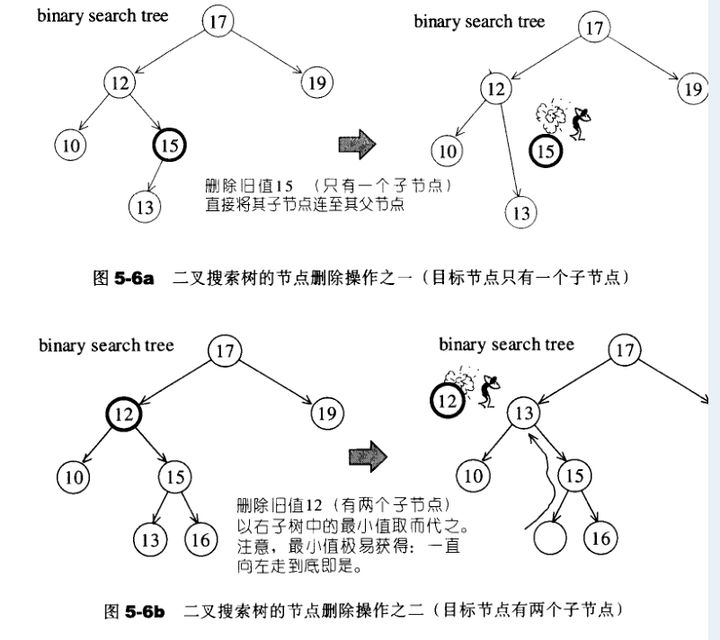
## 20. 树导览（BST,AVL）

**BST（二叉搜索树）**

二叉搜索树的**放置规则**是：任何节点的键值一定大于其左子树中的每一个节点的键值，并小于其右子树的每一个节点的键值。因此从根节点一直往左走，直至无左路可走，即得最小元素；从根节点一直往右走，直至无右可走，即得最大元素。







**2）平衡二叉搜索树（balanced binary search tree）**

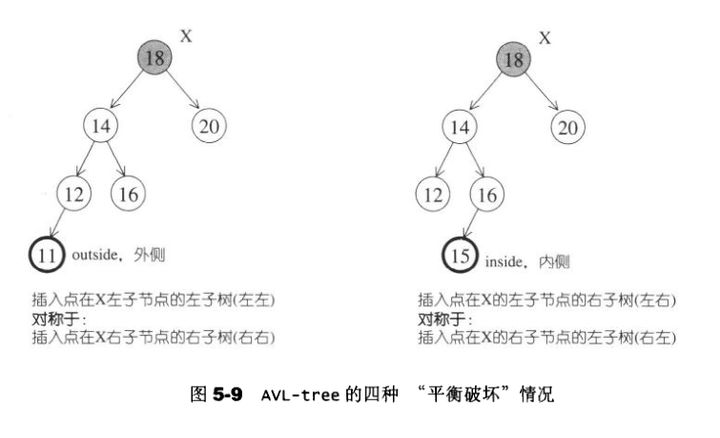
平衡：没有任何一个节点过深（深度过大）

**3） AVL-tree(Adelson-Velskii-Landis tree)**

AVL tree是一个“加上了额外平衡条件”的二叉搜索树。其平衡条件的建立是为了确保整棵树的深度为O(logN)，AVL tree要求任何节点的左右子树的高度相差做多为1，这是一个较弱的条件，但仍能保证“对数深度”平衡状态。

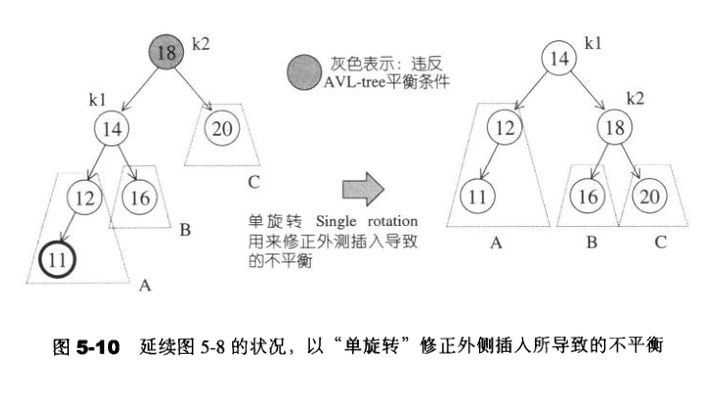
平衡被破坏的四种情况：

1. 插入点位于X的左子节点的左子树-左左
2. 插入点位于X的左子节点的右子树-左右
3. 插入点位于X的右子节点的左子树-右左
4. 插入点位于X的右子节点的右子树-右右

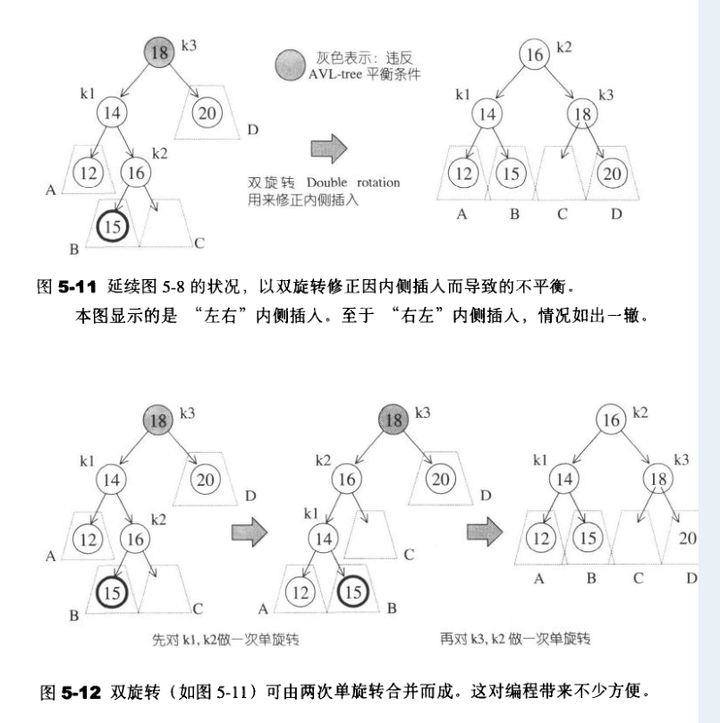


**单旋转：**

将违反AVL tree规则的子树的插入元素边的第一个节点K1提起来，变成跟节点，是K2自然下滑，并将B子树挂到K2的左侧，如下图。



**双旋转：**从下往上，单旋转两次



## 21. RB-tree红黑树

满足以下规则：

1、每个节点不是红色就是黑色

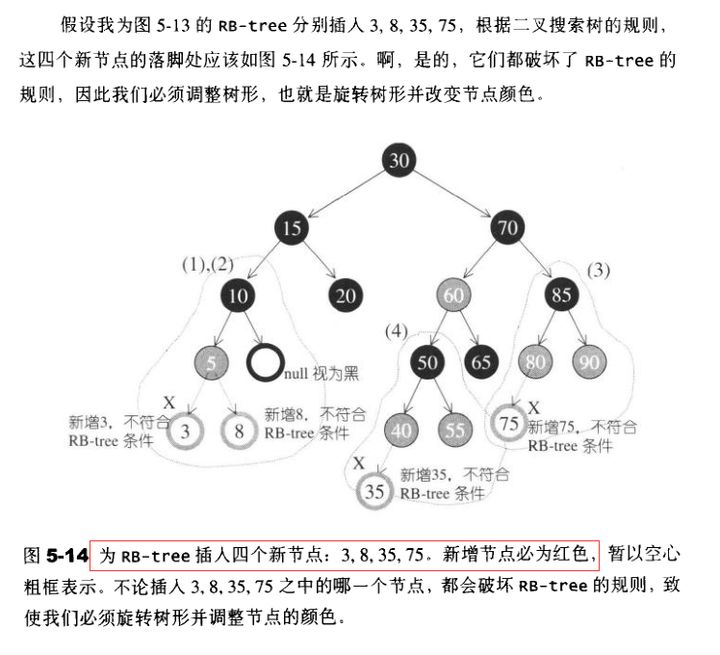
2、根节点为黑色

3、叶子节点是黑色。

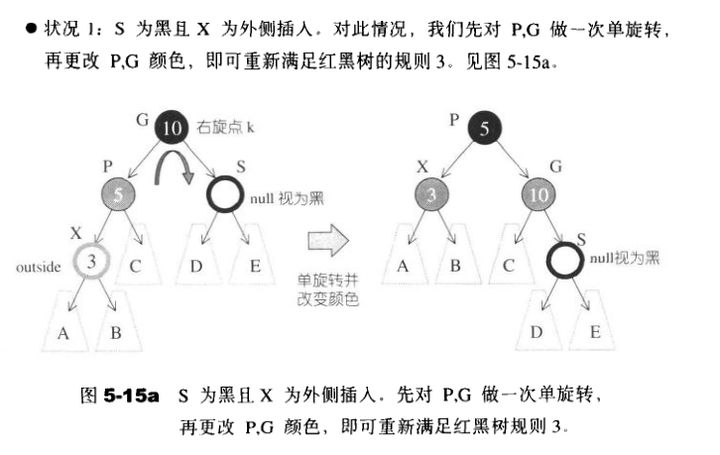
4、如果节点为红色，其子节点必须为黑色

5、任一节点至NULL（树尾端）的任何路径，所含黑节点数必须相同

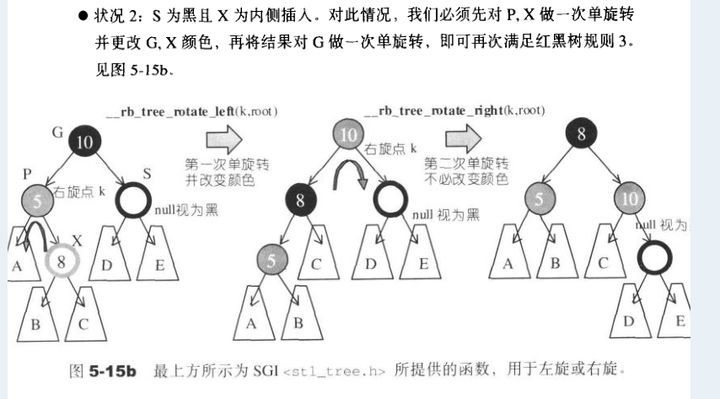
由规则5 => 新增节点必须为红色。规则4 => 新增节点父节点为黑色。如果插入不满足上述要求，就需要调整RB-tree。



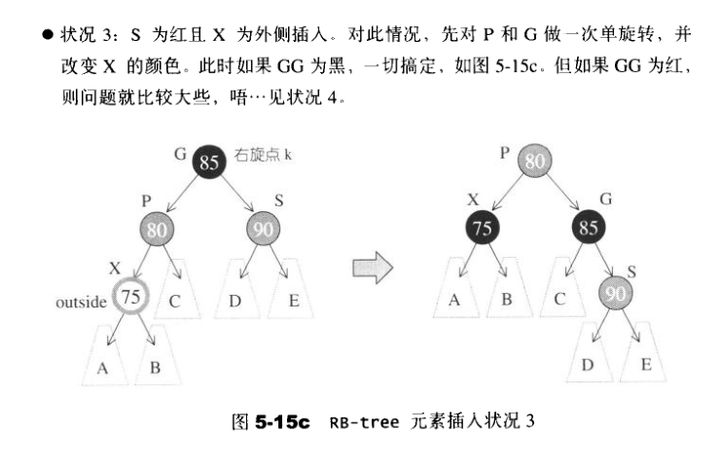
插入3



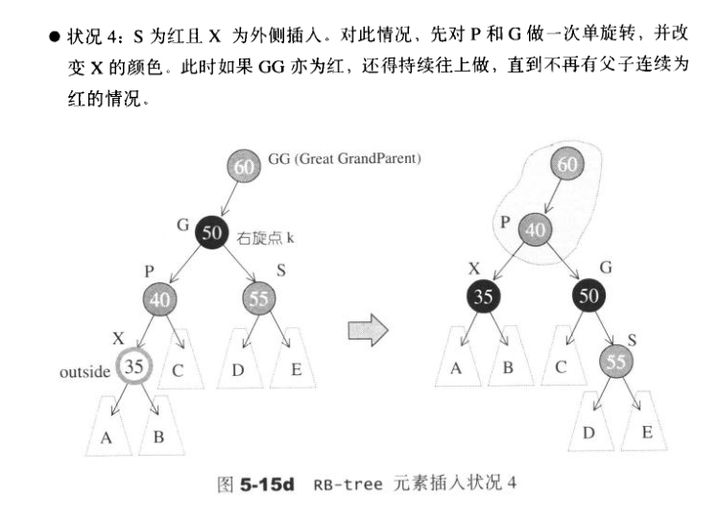
插入8



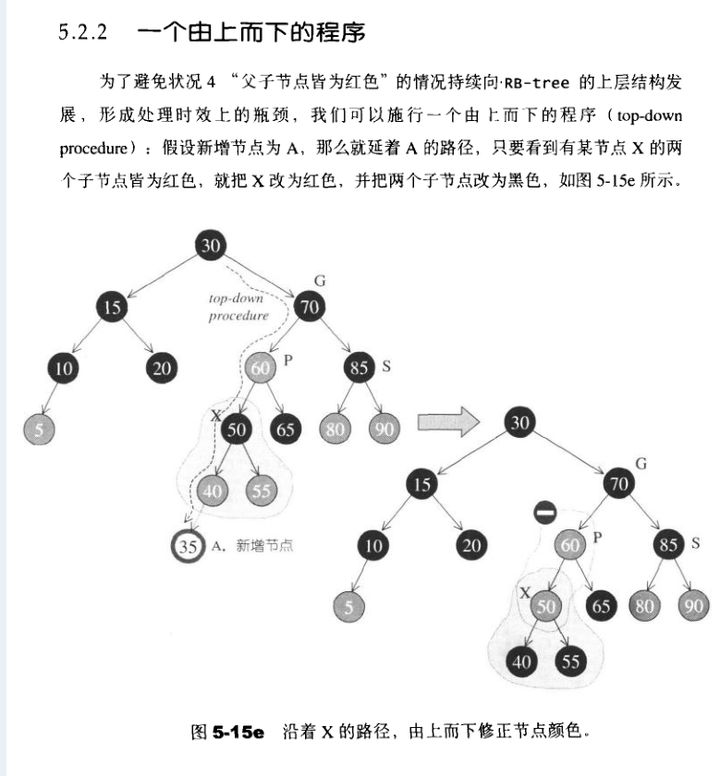
插入75



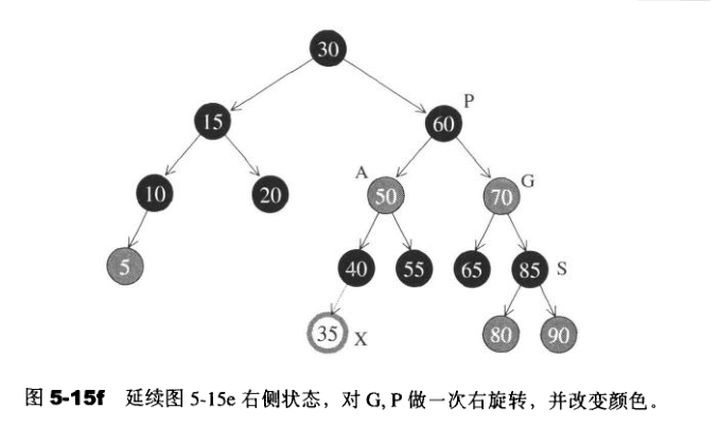
插入35



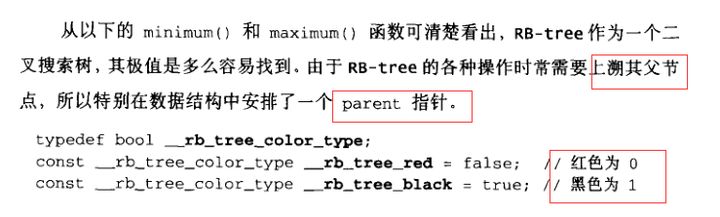
自上而下的程序：

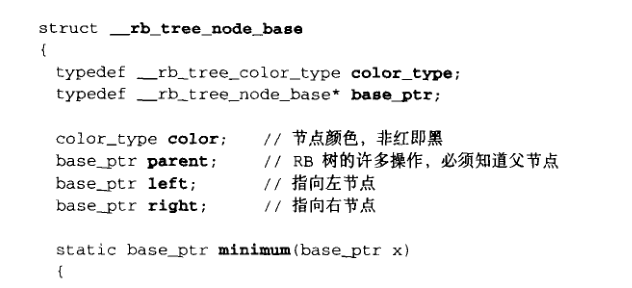


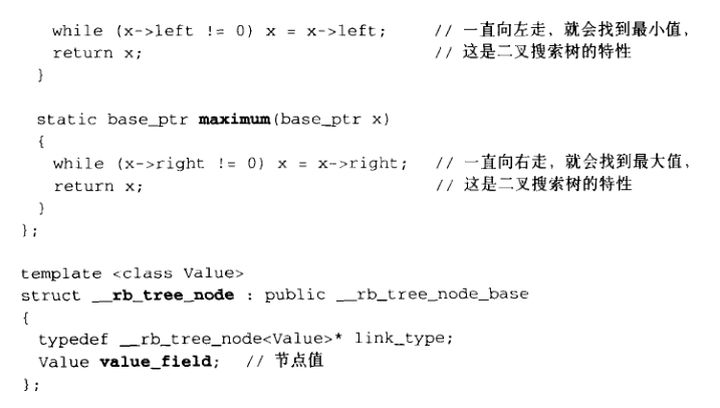
插入节点35

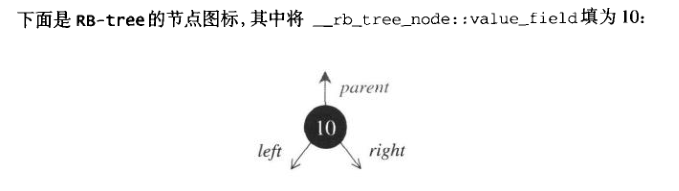


* RB-tree 的节点的设计

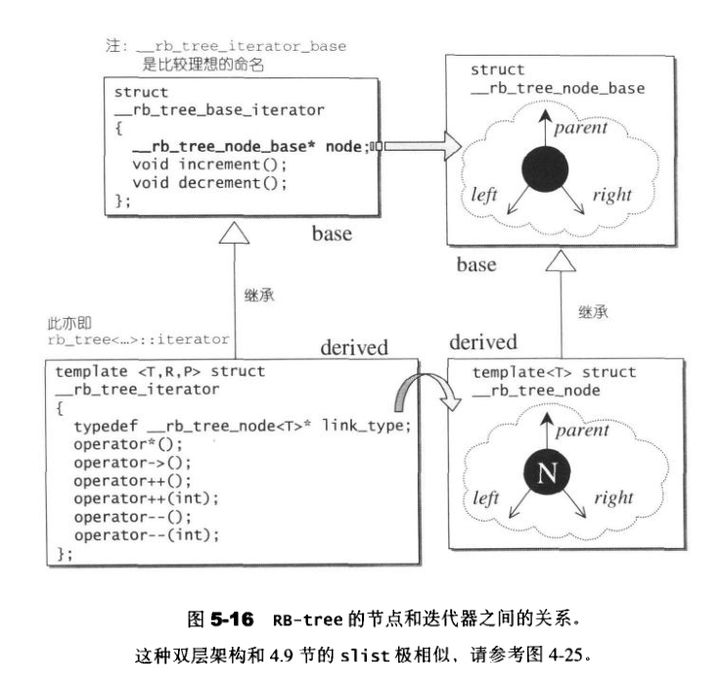






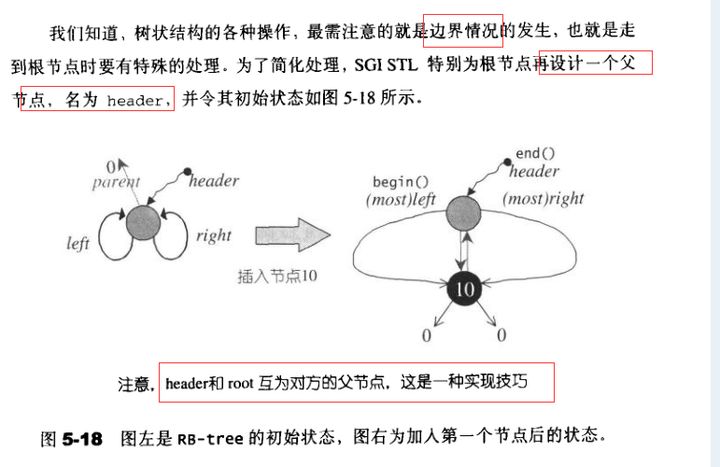


* RB-tree的迭代器



* RB-tree 的数据结构以及构造与内存管理

详细内容看书。



* RB-tree 的元素操作

元素插入操作:插入键值，节点键值允许重复

Template <class key, class value, class KeyOfValue, class compare, class Alloc>

Typename rb\_tree<key,value,KeyOfValue,compare,Alloc>::iterator

Rb\_tree< key,value,KeyOfValue,compare,Alloc>::**insert\_equal**(const value& x){…}

插入键值不允许重复，若重复则插入无效

template <class key, class value, class KeyOfValue, class compare, class Alloc>

pair<typename rb\_tree<key,value,KeyOfValue,compare,Alloc>::iterator,bool>

rb\_tree< key,value,KeyOfValue,compare,Alloc>::**insert\_equal**(const value& x){…}

元素的搜寻操作：

template <class key, class value, class KeyOfValue, class compare, class Alloc>

typename rb\_tree<key,value,KeyOfValue,compare,Alloc>::iterator

rb\_tree< key,value,KeyOfValue,compare,Alloc>::**find** (const value& x){…}

## 21. set

Set的特性是，所有元素都会根据元素的**键值自动排序**。Set的元素的键值就是实值，实值就是键值，set不允许两个元素有相同的键值。**Set以RB-tree为底层机制。**在客户端对set进行插入或删除操作后，之前的迭代器依然有效。

1）元素操作

begin() 返回指向map头部的迭代器

clear(） 删除所有元素

count() 返回指定元素出现的次数

empty() 如果map为空则返回true

end() 返回指向map末尾的迭代器

equal\_range() 返回集合中与给定值相等的上下限的两个迭代器

erase() 删除一个元素

find() 查找一个元素

get\_allocator() 返回map的配置器

insert() 插入元素

key\_comp() 返回比较元素key的函数

lower\_bound() 返回键值>=给定元素的第一个位置

max\_size() 返回可以容纳的最大元素个数

rbegin() 返回一个指向map尾部的逆向迭代器

rend() 返回一个指向map头部的逆向迭代器

size() 返回map中元素的个数

swap() 交换两个map

upper\_bound() 返回键值>给定元素的第一个位置

value\_comp() 返回比较元素value的函数

#include<set>

Int ia[5]={0, 1,2,3,4,5};

Set<int>iset(ia,ia+5);

Cout<<”size: “<<iset.size()<<endl;

Cout<<”3 count = “<<iset.count(3)<<endl;

iset.insert(6);

iset.erase(1);

set<int>::iterator ite1=iset.begin();

set<int>::iterator ite2=iset.end();

for(;ite1 != ite2;++ite1){

cout<<\*ite1; }

ite1 = find(iset.begin(),iset.end(),3 );

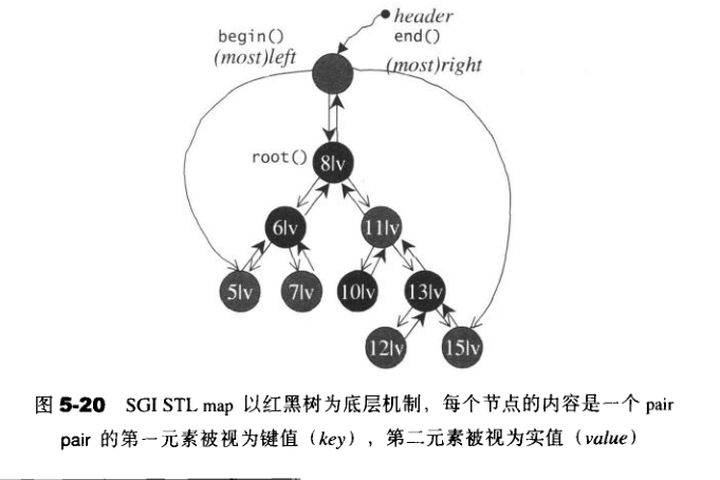
if(ite1 != iset.end()){

cout<<”3 found.”<<endl;}

else{ cout<<” 3 not found”<<endl; }

## 22. map

Map的特性是，所有元素都会根据元素的**键值自动排序**。Map的所有元素都是pair，同时拥有实值（value）和键值（key）,pair的第一元素视为键值，第二元素视为实值，map不允许两个元素拥有相同的键值。当用户对map进行了增加和删除操作后，所有的迭代器都依然有效。



## 23. hashtable

hash table(散列表)数据结构，在插入、删除、查找等操作具有 **“常数平均时间”** O(1) 的表现，这种表现是以统计为基础，不需要依赖输入元素的随机性。

1）概述

１．hashtable在插入，删除，搜寻操作上具有＂常数平均时间＂的表现，不依赖输入元素的随机性．

２．hashtable通过hash function将元素映射到不同的位置，**但当不同的元素通过hash function映射到相同位置时，便产生了＂碰撞＂问题**．解决碰撞问题的方法主要有线性探测，二次探测，开链法等．

３．线性探测

当hash function计算出某个元素的插入位置，而该位置的空间已不可用时，循序往下寻找下一个可用位置(到达尾端时绕到头部继续寻找)，会产生primary clustering（一次聚集）问题．

４．二次探测

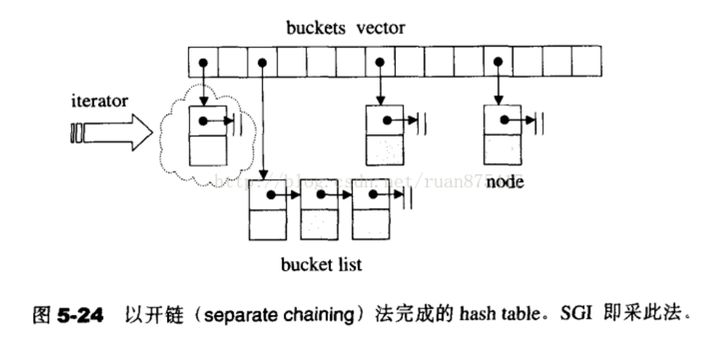
当hash function计算出某个元素的插入位置为H，而该位置的空间已经被占用，就尝试用H+１²、H+2²…，会产生secondary clustering(二次聚集)问题．

５．开链

在每一个表格元素中维护一个list：hash function为我们分配某个list，在那个list上进行元素的插入，删除，搜寻等操作．SGI STL解决碰撞问题的方法就是此方法．表格大小以质数来设计，SGI STL事先将28个质数存储，以备随时访问。

2）hashtable 的桶子（buckets）与节点（nodes）

遵循SGI的命名，称hash table表格内的元素为桶子（buckets），此名称的大约意义是，表格内的每个单元，涵盖的不只是个节点（元素），甚且可能是一“桶”节点。



下面是hash table节点的定义

template <class Value>

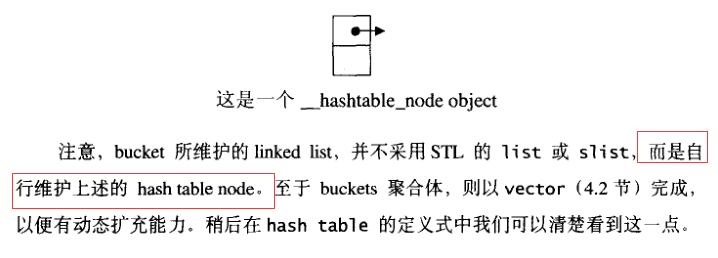
struct \_\_hashtable\_node

{

\_\_hashtable\_node\* next;

Value val;

};

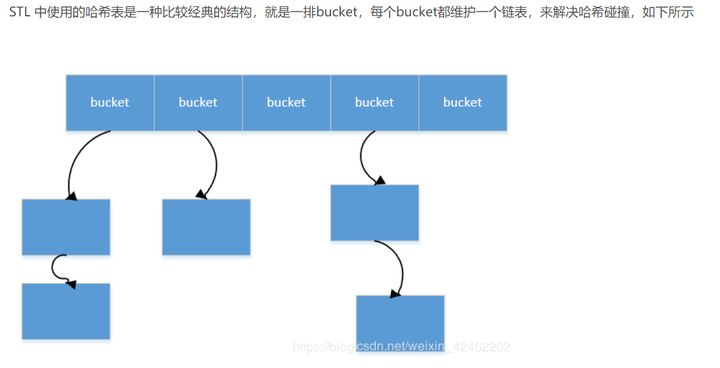


3）hashtable的迭代器

hashtable迭代器必须永远维系与整个”buckets vector”的关系，并记录目前所知节点．hashtable的迭代器没有后退操作，也没有逆向迭代器．

**前进操作**：首先尝试从目前所指的节点出发，前进一个位置（节点），由于节点被安置于list内，所以利用节点的next指针即可轻易达成前进操作，如果节点正巧是list的尾端，就跳至下一个bucket身上，那正是指向下一个list的头部节点。

4）hashtable的数据结构





value : 节点的实值类别   
key : 节点的键值类别   
HashFcn : hash function函数类别   
ExtractKey : 从节点中取出键值的方法   
EqualKey : 判断键值相同与否的方法   
Alloc : 空间配置器，默认使用std::alloc

首先是三个仿函数，这些仿函数都是从模板参数指定的，然后在构造函数中赋值

hash：用于获取 key 对应的哈希值，以确定要放到哪一个 bucket 中

equals：用于判断 key 是否相等

get\_key：用于从 value 中取得 key，前面说过 value = key + data

接下来是 buckets 和 num\_elements

buckets：维护哈希表的 bucket，是一个指针数组，每个元素都是 node\* 类型

num\_elements：元素的个数

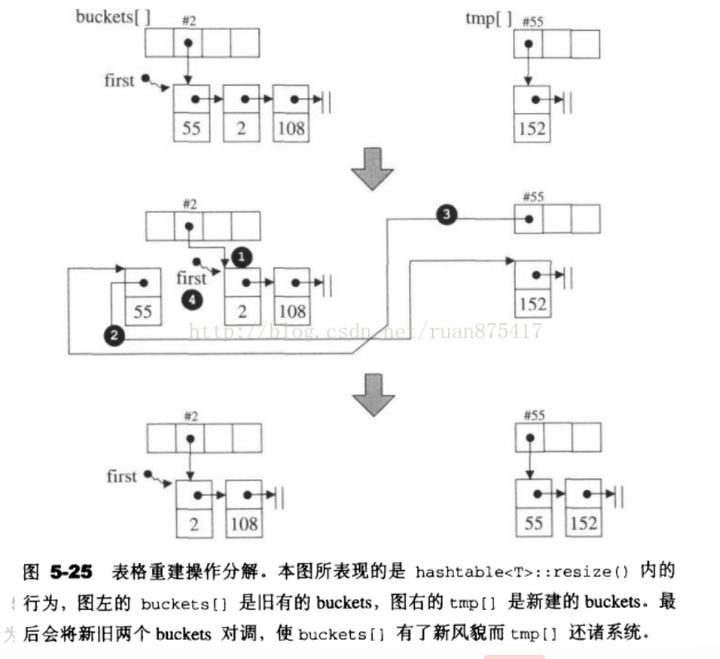


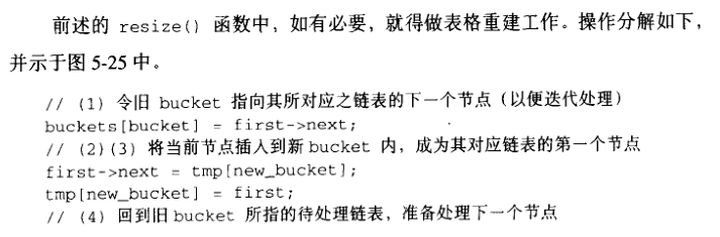
1. hashtable的构造与内存管理



**表格重建操作:**

重建表格是要将每一个元素从小进行hash，然后再delete oldhashtable的所有元素。表格是否需要重建**判断原则**：拿元素个数和bucket vector的大小来比，如果前者比后者大就重建表格．因此，每个bucket(list)的最大容量和bucket vector的大小相同．

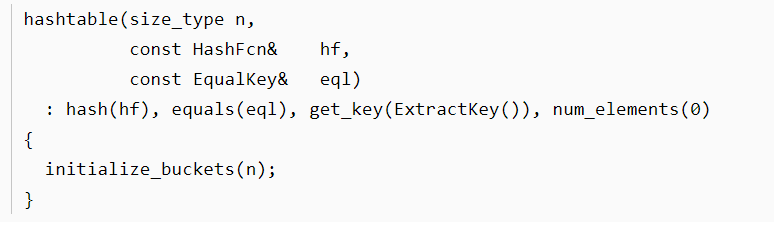




first = buckets[bucket];

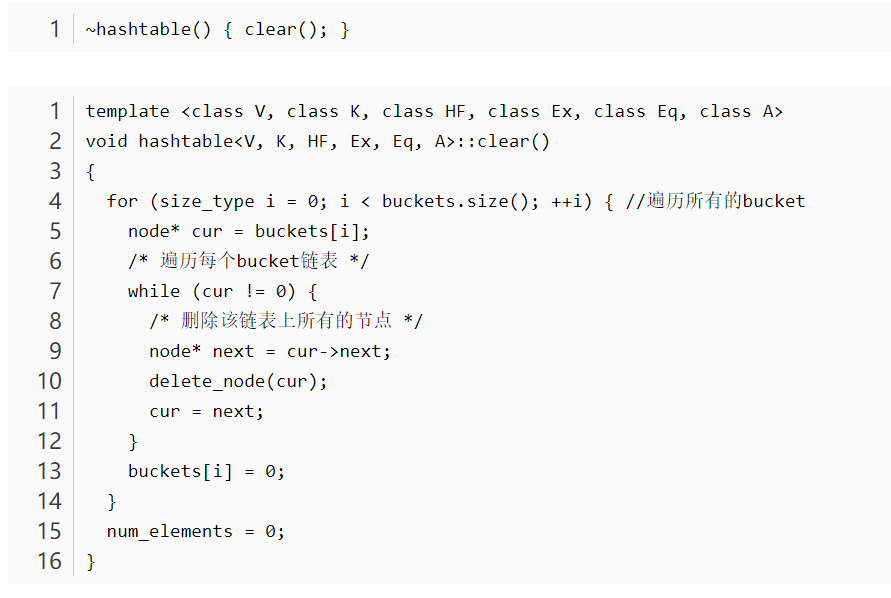
1. hashtable的操作

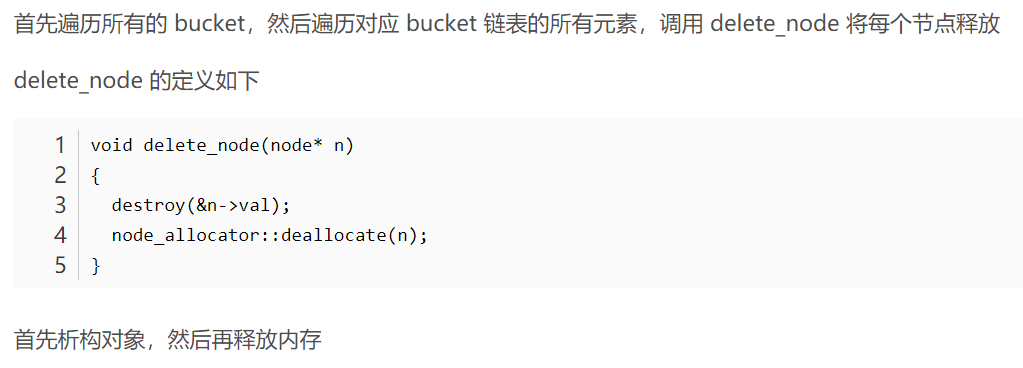
* 构造函数





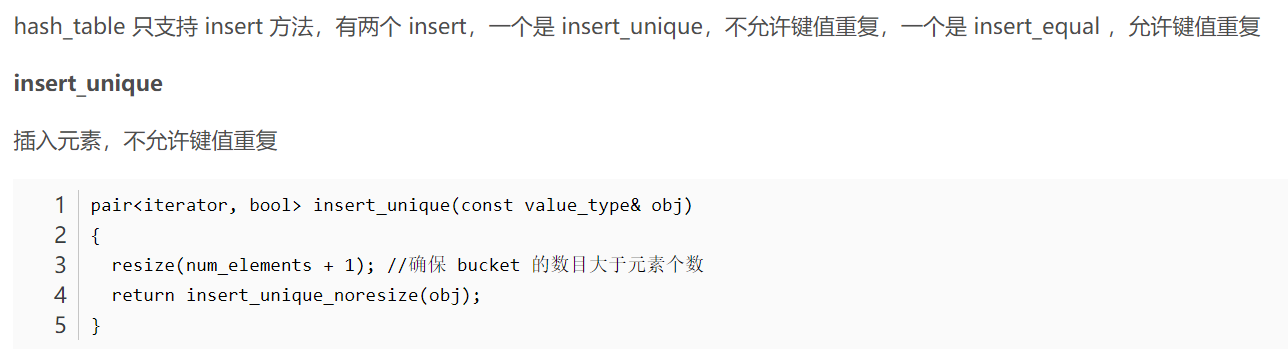
* 析构函数





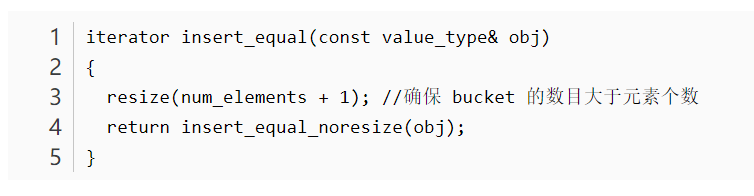
* 插入元素：不允许键值重复

回到 insert\_unique\_noresize，在计算完应该插入到哪个 bucket 之后，获取指定的 bucket，然后遍历该 bucket 链表，如果该链表上有一个节点的 key 和 插入元素的 key 相等，那么就返回插入失败。否则，生成一个新的节点，然后插入到指定的 bucket 链表中

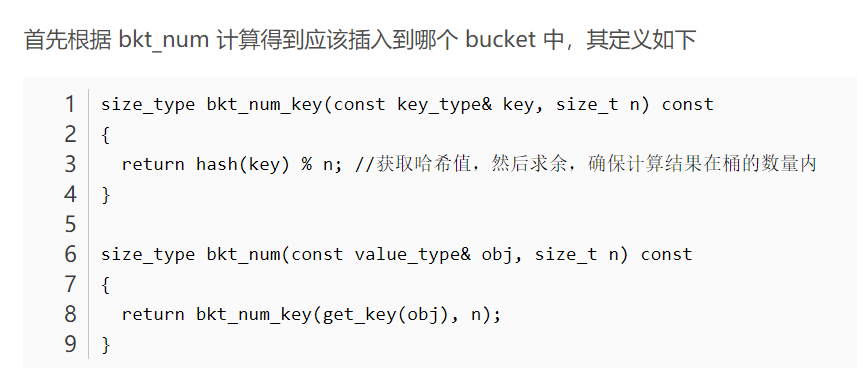


* 插入元素：允许键值重复

首先确定要在哪一个bucket插入，然后遍历bucket链表，如果找到相等的节点，那么就在该节点处之后插入。否则，在bucket链表头插入



* 判断元素的落脚处



* 复制

template <class V, class K, class HF, class Ex, class Eq, class A>

void hashtable<V, K, HF, Ex, Eq, A>::clear() {…}

* 整体删除

template <class V, class K, class HF, class Ex, class Eq, class A>

void hashtable<V, K, HF, Ex, Eq, A>::copy\_from(const hash\_table& ht) {…}

* begin()指向第一个元素的迭代器

Iterator begin(){…}

* end()指向结尾的迭代器

iterator end(){…}

* find()查找指定key的节点

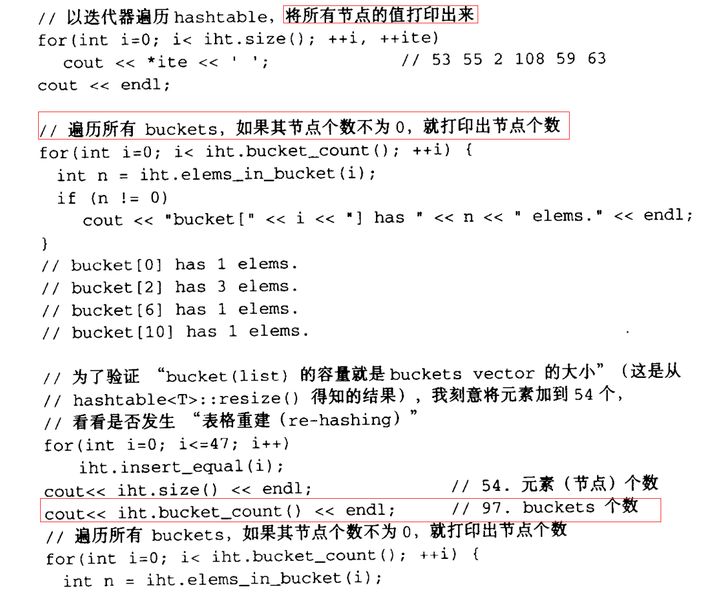
首先找到对应的bucket，然后遍历bucket链表查找等于指定 key 的节点

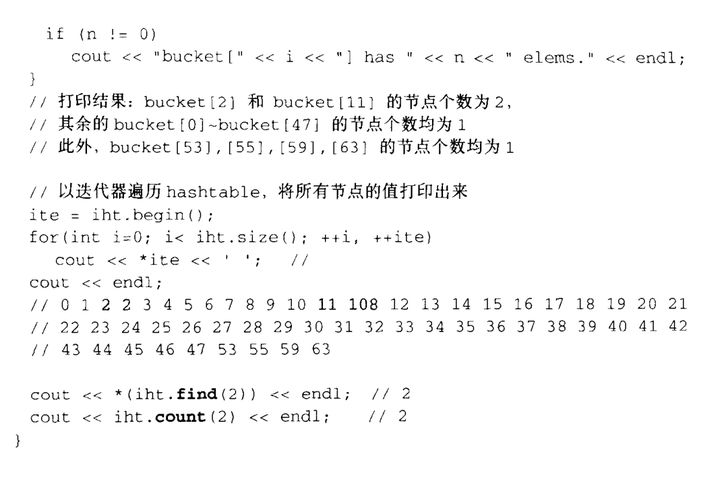
iterator find(const key\_type& key) {…}

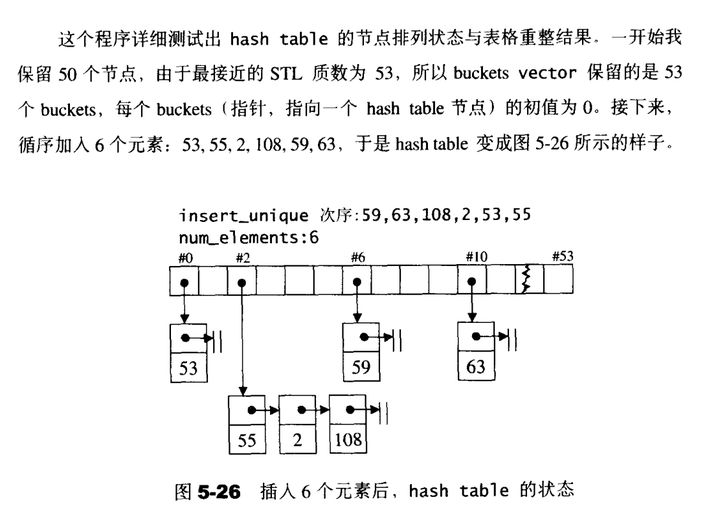
7) 使用实例

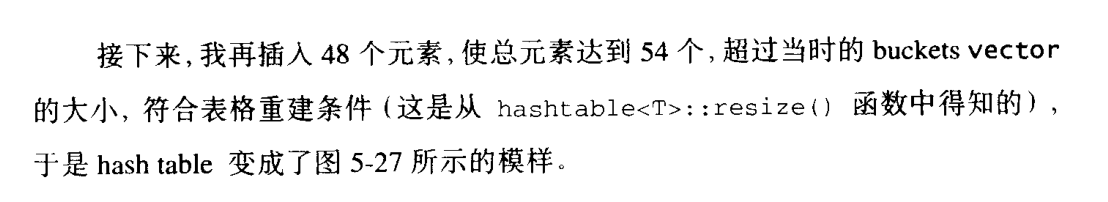


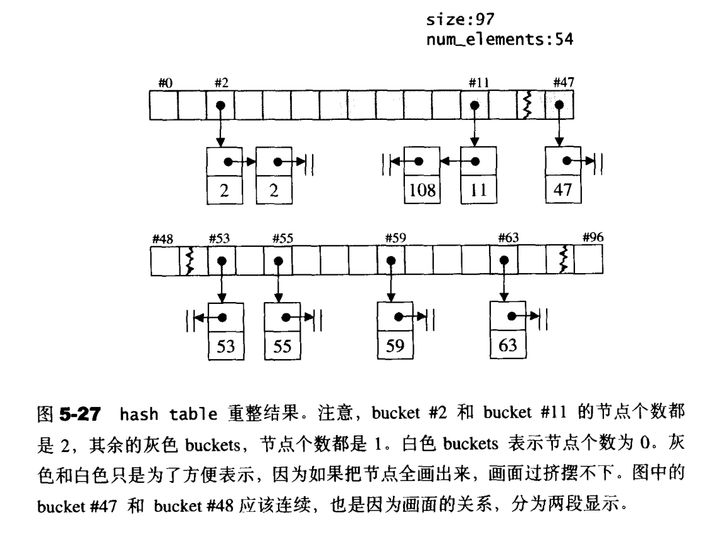












1. hash functions

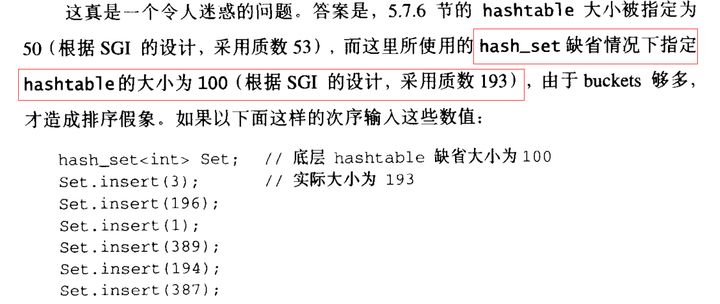


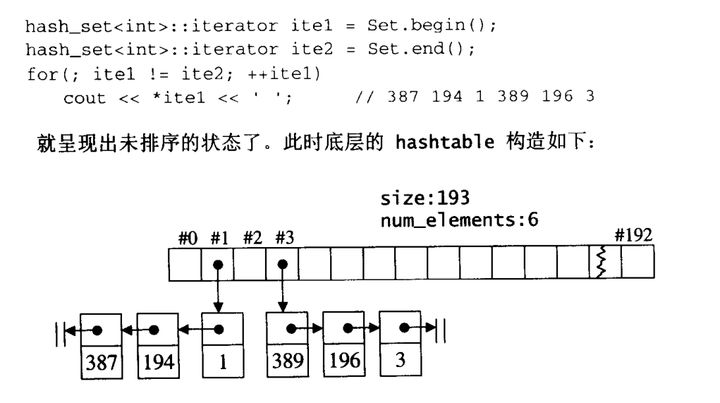
## 24. hash\_set

以hash\_table为底层机制，由于hash\_set所供应的操作接口，hashtable都提供了，所以几乎所有hash\_set的操作行为，都只是转调用hashtable的操作行为而已。Set的底层机制是RB-tree可以自动排序，hash\_set的底层机制是hashtable不能自动排序。hash\_set的使用方式与set完全相同。**运用set就是为了能够快速搜寻元素。**

1）使用实例







## 25. hash\_map

以hash table为底层机制，由于hash\_map所供应的操作接口，hashtable都提供了，所以几乎所有hash\_map的操作行为，都只是转调用hashtable的操作行为而已。map的底层机制是RB-tree可以自动排序，hash\_map的底层机制是hashtable不能自动排序。hash\_map的使用方式与map完全相同。**运用map就是为了能够根据键值快速搜寻元素。**

Map的特性是每一个元素都拥有一个键值（key）和一个实值（value）

1）hash\_table常用操作

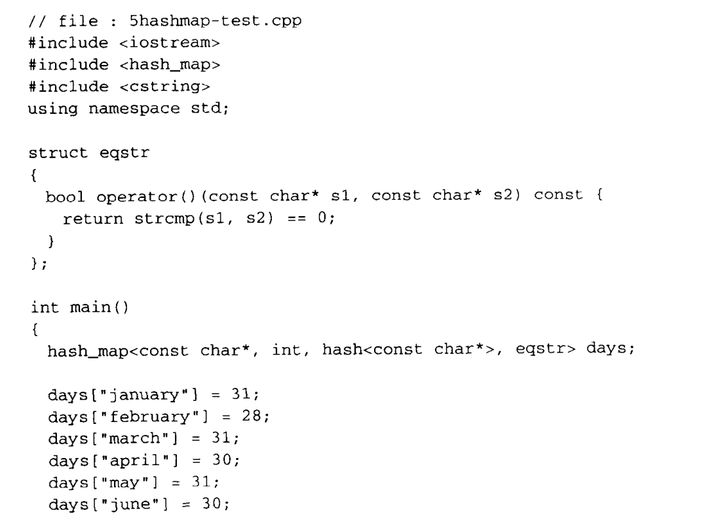
Void resize(size\_type hint) {…} //表格重建

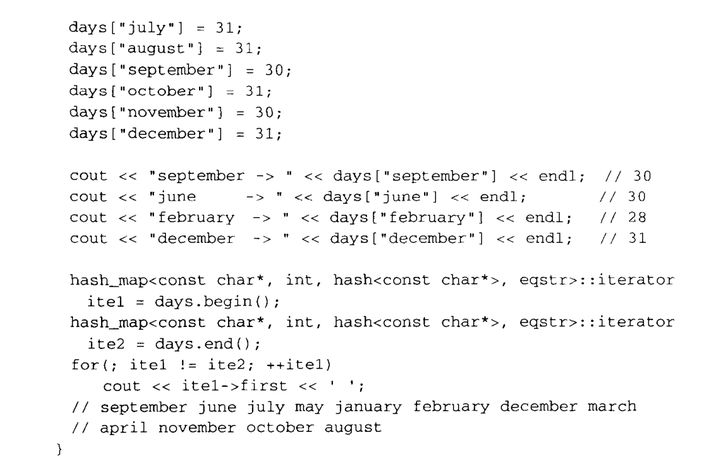
size\_type bucket\_count() const {…} //桶的大小

size\_type max\_bucket\_count() const{…} //桶的大小的最大值

size\_type elems\_in\_bucket(size\_type n) const {…} //n号桶里有几个元素

2） 使用实例





## 26. hash\_multiset

hash\_multiset与multiset的特性完全相同，唯一差别在于它的底层机制是hashtable，也因此hash\_multiset的元素并不会被自动排序。Multiset底层机制是RB-tree会自动排序。

hash\_multiset 与 hash\_set 实现上唯一差别在于，前者的元素插入操作采用底层机制hashtable的inset\_equal()，而后者采用的是insert\_unique()。

1）hash\_multiset常用操作

Void resize(size\_type hint) {…} //表格重建

size\_type bucket\_count() const {…} //桶的大小

size\_type max\_bucket\_count() const{…} //桶的大小的最大值

size\_type elems\_in\_bucket(size\_type n) const {…} //n号桶里有几个元素

## 27. hash\_mutimap

hash\_multimap与multimap的特性完全相同，唯一差别在于它的底层机制是hashtable，也因此hash\_multimap的元素并不会被自动排序。Multimap底层机制是RB-tree会自动排序。

hash\_multimap 与 hash\_map 实现上唯一差别在于，前者的元素插入操作采用底层机制hashtable的inset\_equal()，而后者采用的是insert\_unique()。

1）常用操作

Size\_type size() const {…}

Size\_type max\_size() const {…}

Bool empty() const {…}

Void swap(hash\_multimap& hm) {…}

Iterator begin() {…}

Iterator end() {…}

Const\_iterator begin() const {…}

Const iterator end() const {…}

Iterator insert(const value\_type& obj) { …}

Template<class InputIterator>

Void insert(InputIterator f, InputIterator l) {…}

Iterator find(const key\_type& key) {…}

Size\_type count(const key\_type& key) const {…}

Pair<iterator,iterator> equal\_range(const key\_type& key) {…}

Size\_type erase(const key\_type& key) { …}

Void erase(iterator it) {…}

Void erase(iterator f,iterator l) {….}

Void clear() {…}

1. 使用实例

