STL之应用篇

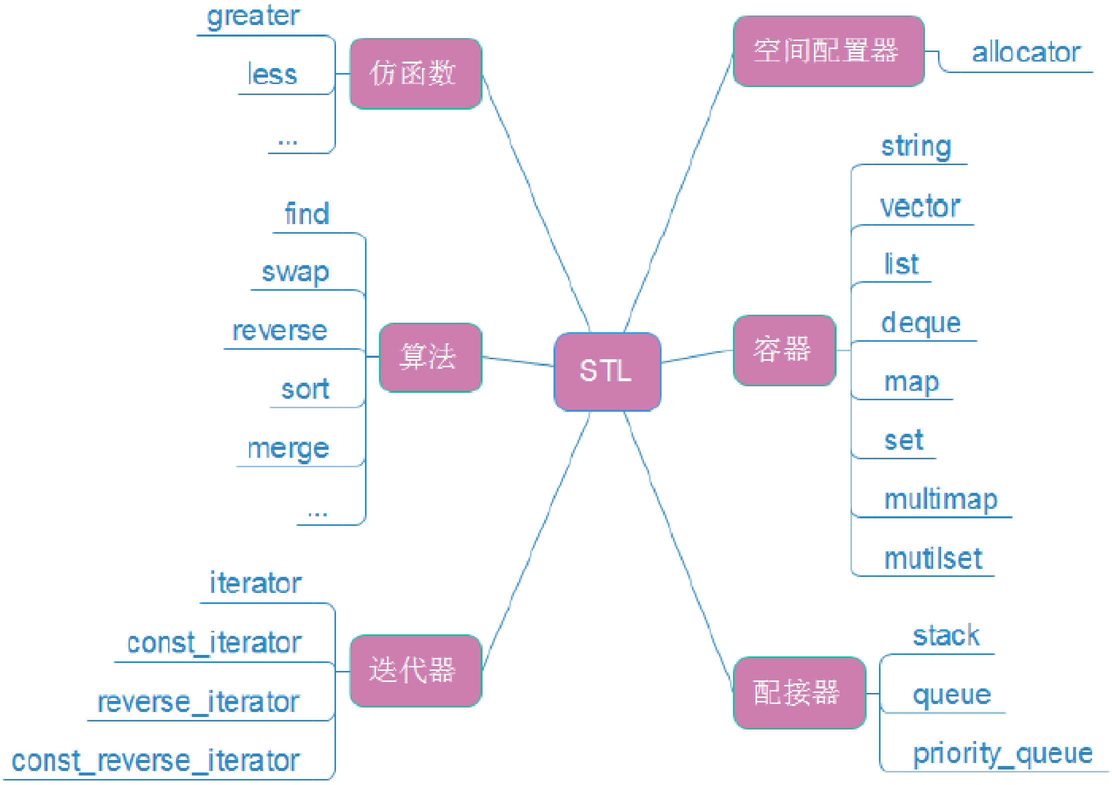
- 发生在自己身上的例子
- STL简介
- STL六大组件
- 容器
- 迭代器
- 算法
- 仿函数
- 适配器
- 空间配置器

> 不知道大家是否遇到过这样的经历：当你准备着手完成数据结构老师布置的作业时，或者平时在做一些练习题时，或者在为你所负责的项目添加某个功能时，发现自己需要用到链表或者哈希表之类的数据结构或者算法，而自己手头又没有现成的代码，于是只能自己实现一份。姑且认为大家都是高手中的高高手，那对于自己实现出来的代码：**是否考虑过能够应对所有的场景即通用性？运行起来效率怎么样？对于内存的使用情况？是否可扩展？以后相同情况出现了是否可重复使用？**以上所有问题，STL都替你考虑到了，本节课我们就站在巨人的肩膀上去攻城略地。

STL简介

> **STL(standard template libaray)：标准模板库，是C++程序设计语言的标准程序库，**是一个**包罗算法与数据结构的软件框架**，1998年正式纳入C++标准中，所有的C++编译器和操作系统都支持。
>
> STL的目的是标准化组件，所以在STL中使用了泛型编程的思想，对我们常用的数据结构：顺序表、链表、树、哈希以及常用的查找、排序等算法使用模板进行了封装，而且从运行效率以及内存使用上都基本达到了最优。引入STL后，再也不需要我们们重新造轮子，而且写出来的代码更加简洁，容易修改，可移植性高。万一STL所提供的容器或者算法不能满足我们的要求，我们也可以实现自己的容器或算法与STL中的其他组件进行交融。

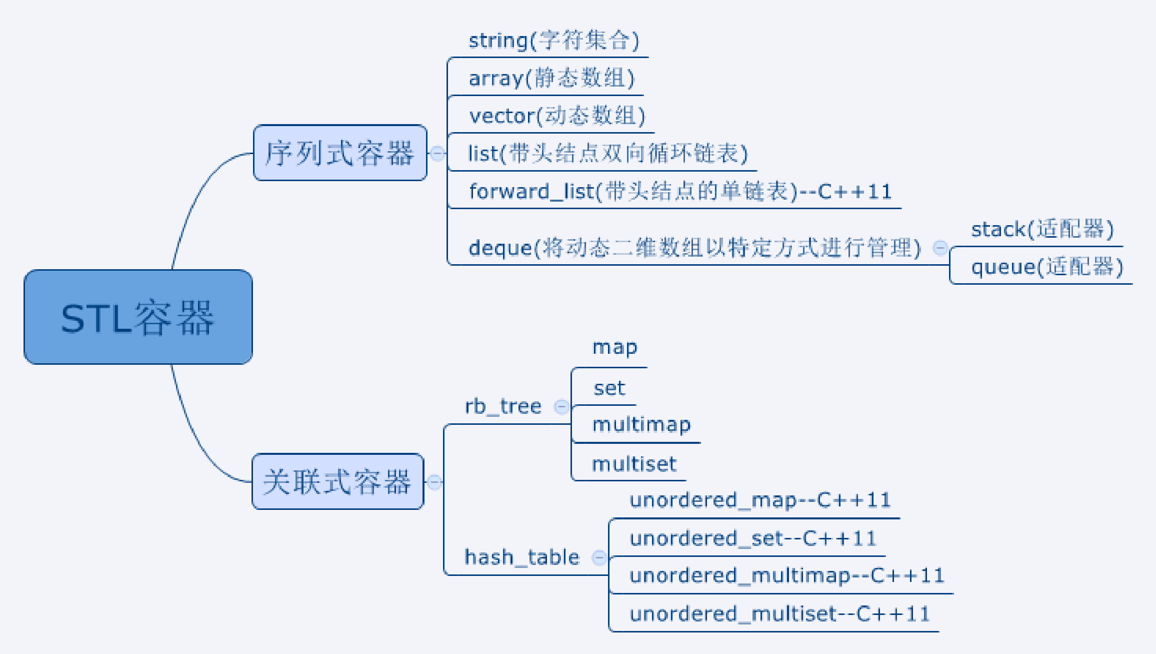
STL六大组件



容器

容器，是用来放东西的，而STL中的容器则是来放数据的，因此也称数据容器。由于对数据进行的操作不同，使用的场景各异，可能需要相应的数据结构来管理数据，常见的数据结构：array、list、tree、stack、queue、hash table、map、set等，因此STL中的容器便是对各种不同数据结构的封装。

根据数据在容器中的排序特性，容器分为序列式容器和关联式容器
序列式容器中的元素次序：即是按照其加入容器中的先后次序，不一定有序

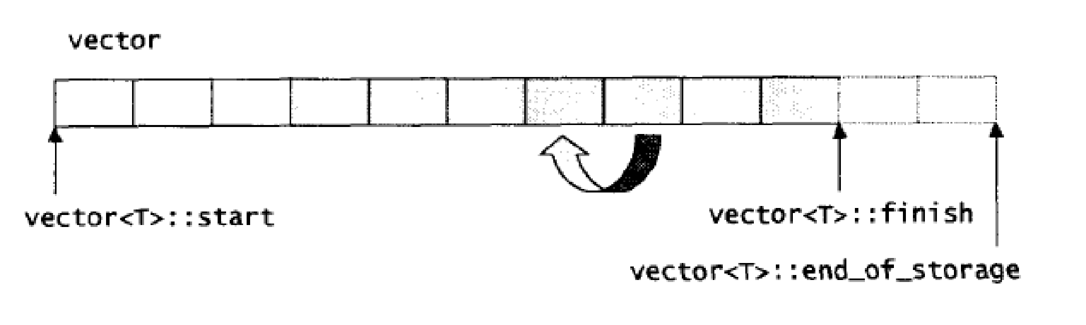


序列式容器

- vector

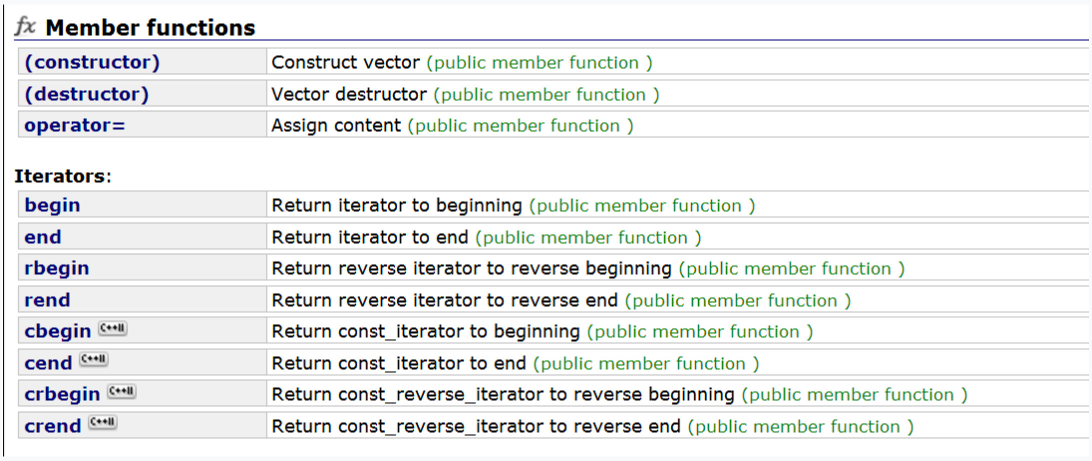
  vector底层动态维护了一段连续的空间，随着元素的加入，当容器中的元素存放满时，如果再要加入其它数据，vector的内部机制会自动的进行扩容以容纳新元素。在SGI版本的源码中，具体的操作是：当vector检测到空间不足时，动态开辟原空间大小两倍的空间，接着将旧空间中的元素高效的拷贝到新空间中，最后释放旧空间，所有的操作对用户来说都是透明的。

  1. vector的底层结构：

  2. vector的迭代器
  因为vector底层是一段连续空间，因此vector的迭代器可以是一个原生态指针，即：所存储元素类型的指针

  3. vector的操作
  使用vector时，必须包含vector头文件，同时vector处于std命名空间中，也要引入std命名空间

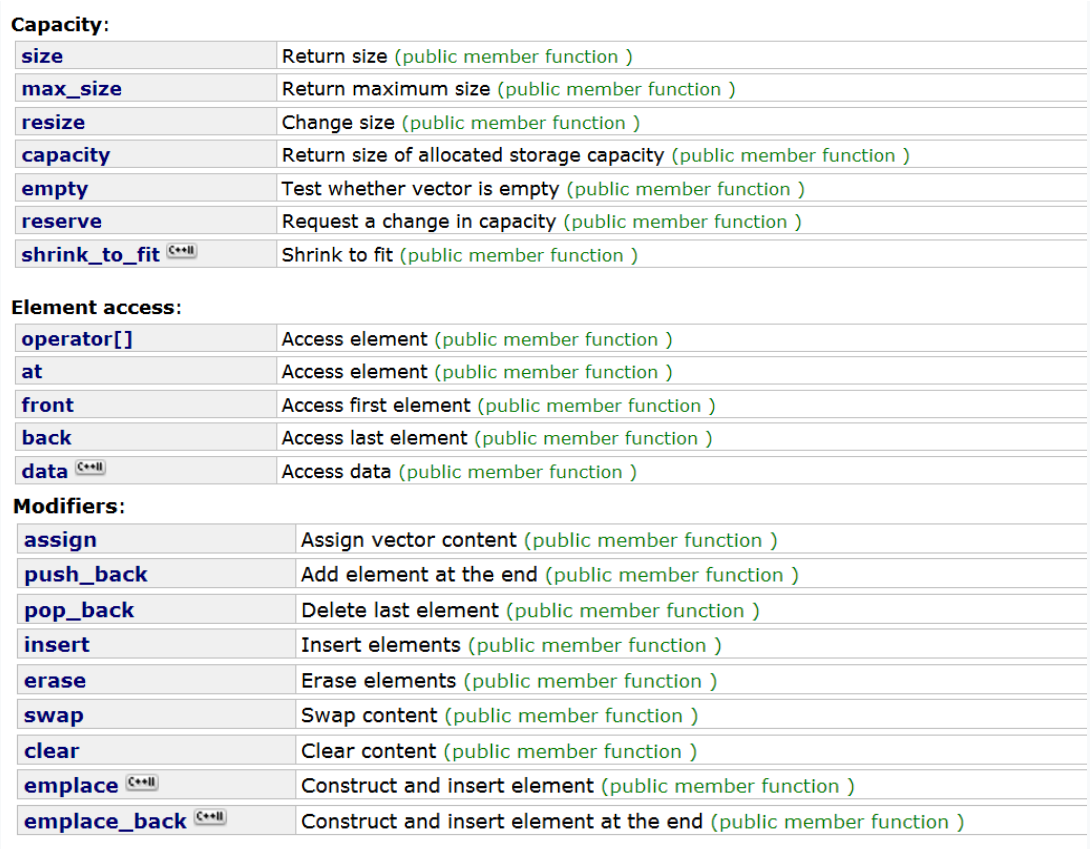
  

  fx **Member functions**

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

  Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)



Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

http://www.cplusplus.com/reference/vector/vector/?kw=vector

```

#include<vector>

// 验证vector的构造函数，此处仅验证了三种情况，其他请学生们自己验证
void TestVector1()
{
    // 构造一个空的vector
   vector<int> v1;

   // 构造一个空的vector,但是底层空间大小设置为20个元素
   vector<int> v2(20);

   int array[] = {1,2,3,4,5,6,7,8,9,0};
   // 构造一个vector对象，并用数组进行初始化
   vector<int> v3(array, array+sizeof(array)/sizeof(array[0]));

   // 查看v1中元素的个数
   cout<<"size = "<<v1.size()<<endl;
   // 查看v1容量的大小
   cout<<"capacity = "<<v1.capacity()<<endl;

   cout<<"size = "<<v2.size()<<endl;
   cout<<"capacity = "<<v2.capacity()<<endl;

   cout<<"size = "<<v3.size()<<endl;
   cout<<"capacity = "<<v3.capacity()<<endl;
```

```cpp

  // 将v3赋值给v1后再次查看v1的元素个数和容量大小
  v1 = v3;
  cout<<"size = "<<v1.size()<<endl;
  cout<<"capacity = "<<v1.capacity()<<endl;

  cout<<"v1 data:";
  for(size_t i = 0; i < v1.size(); ++i)
      cout<<v1[i]<<" ";
  cout<<endl;

  cout<<"v3 data:";
  for(size_t i = 0; i < v1.size(); ++i)
      cout<<v1[i]<<" ";
  cout<<endl;

  // 注意：出了该函数的作用域之后，vector会自动将其内部的空间释放掉
}

// push_back/pop_back/insert/erase
void TestVector2()
{
  // 构造一个空的vector，然后向里面尾插进1,2,3,4 4个元素
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);
  v.push_back(5);
  cout<<"size = "<<v.size()<<endl;
  cout<<"capacity = "<<v.capacity()<<endl;

  // 打印顺序表中的元素
  for(size_t i = 0; i < v.size(); ++i)
      cout<<v[i]<<" ";  // 顺序表底层为连续空间，支持随机访问，在vector中重载了[]
  cout<<endl;

  // 尾删
  v.pop_back();
  v.pop_back();
  cout<<"size = "<<v.size()<<endl;
  cout<<"capacity = "<<v.capacity()<<endl;
  for(size_t i = 0; i < v.size(); ++i)
      cout<<v.at(i)<<" ";
  cout<<endl;

  // 任意位置插入
  // 在vector起始位置插入数据0
  v.insert(v.begin(), 0);

  // 在vector末尾的位置插入3个4
  v.insert(v.end(), 3, 4);

  cout<<"size = "<<v.size()<<endl;
  cout<<"capacity = "<<v.capacity()<<endl;

  vector<int>::iterator it = v.begin();
  while(it != v.end())
      cout<<*it++<<" ";
  cout<<endl;

  // 删除任意位置元素
  // 删除起始位置
  v.erase(v.begin());

  // 删除指定区间内的元素，注意区间是【)
  v.erase(v.begin(), v.begin()+3);

  it = v.begin();
  while(it != v.end())
      cout<<*it++<<" ";
  cout<<endl;
}


void TestVector3()
{
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);
  for(size_t i = 0; i < v.size(); ++i)
      cout<<v[i]<<" ";
  cout<<endl;

  // assign()方法是给vector进行赋值
  // 在进行赋值之前，该方法会先将vector中原有的旧元素erase掉
  // 然后再将新元素插入进去
  // 给vector中5个值为10的元素
  v.assign(5,10);
  for(size_t i = 0; i < v.size(); ++i)
      cout<<v[i]<<" ";
  cout<<endl;
  cout<<"size = "<<v.size()<<endl;
  cout<<"capacity = "<<v.capacity()<<endl;

  // 将vector中的元素清空，注意底层容量的空间不变
  v.clear();
  for(size_t i = 0; i < v.size(); ++i)
      cout<<v[i]<<" ";
```

```cpp
    cout<<endl;
    cout<<"size = "<<v.size()<<endl;
    cout<<"capacity = "<<v.capacity()<<endl;

    // 将数组赋值给vector
    int array[] = {1,2,3,4,5,6,7,8,9,0};
    v.assign(array, array+sizeof(array)/sizeof(array[0]));
    for(size_t i = 0; i < v.size(); ++i)
        cout<<v[i]<<" ";
    cout<<endl;
    cout<<"size = "<<v.size()<<endl;
    cout<<"capacity = "<<v.capacity()<<endl;

    // resize(n, data):将vector中的元素改变到n个
    // 第二个参数可以不用传，默认情况下使用缺省值
    // 将vector中的元素缩小到5个，注意缩小时底层容量不变
    v.resize(5);
    for(size_t i = 0; i < v.size(); ++i)
        cout<<v[i]<<" ";
    cout<<endl;
    cout<<"size = "<<v.size()<<endl;
    cout<<"capacity = "<<v.capacity()<<endl;

    // 将vector中的元素增加到8个
    // 注意：如果增加到某个个数而没有超过vector的实际容量
    //       vector底层的容量不会改变
    v.resize(8);
    for(size_t i = 0; i < v.size(); ++i)
        cout<<v[i]<<" ";
    cout<<endl;
    cout<<"size = "<<v.size()<<endl;
    cout<<"capacity = "<<v.capacity()<<endl;

    // 将vector中的元素增加的20个
    // 注意：如果第二个参数没有传，多处的元素使用缺省值，否则多出的元素使用传递的值
    v.resize(20);
    for(size_t i = 0; i < v.size(); ++i)
        cout<<v[i]<<" ";
    cout<<endl;
    cout<<"size = "<<v.size()<<endl;
    cout<<"capacity = "<<v.capacity()<<endl;

    // 将vector底层容量空间增大到n个，注意此函数只是将容量增大，不改变元素的个数
    v.reserve(30);
    for(size_t i = 0; i < v.size(); ++i)
        cout<<v[i]<<" ";
    cout<<endl;
    cout<<"size = "<<v.size()<<endl;
    cout<<"capacity = "<<v.capacity()<<endl;
```

```
}
```

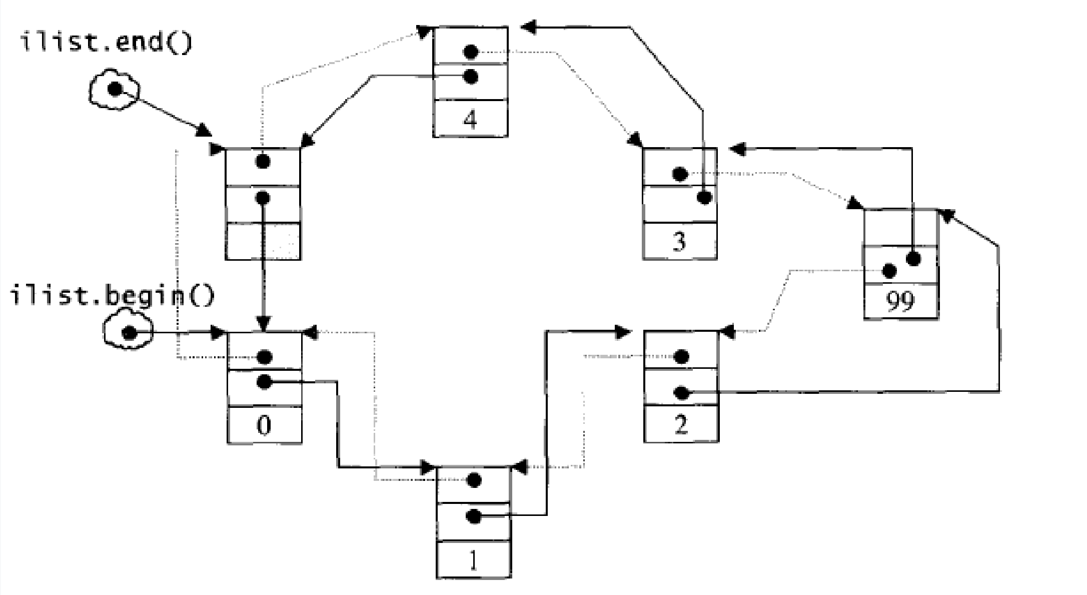
- list

  1. list简介

  由于vector底层搭载一段连续空间，在其任意位置进行数据的插入或删除时效率是非常低下的(O(N))，因此当集合中需要进行大量的插入和删除操作时候，可以考虑list，因为list底层结构为链表，在任意位置进行数据插入操作的时间复杂度均为O(1)

  2. list的底层结构

  list的底层是一个带头结点的双向循环链表，因此在其任意位置进行数据插入和删除操作是非常方便的

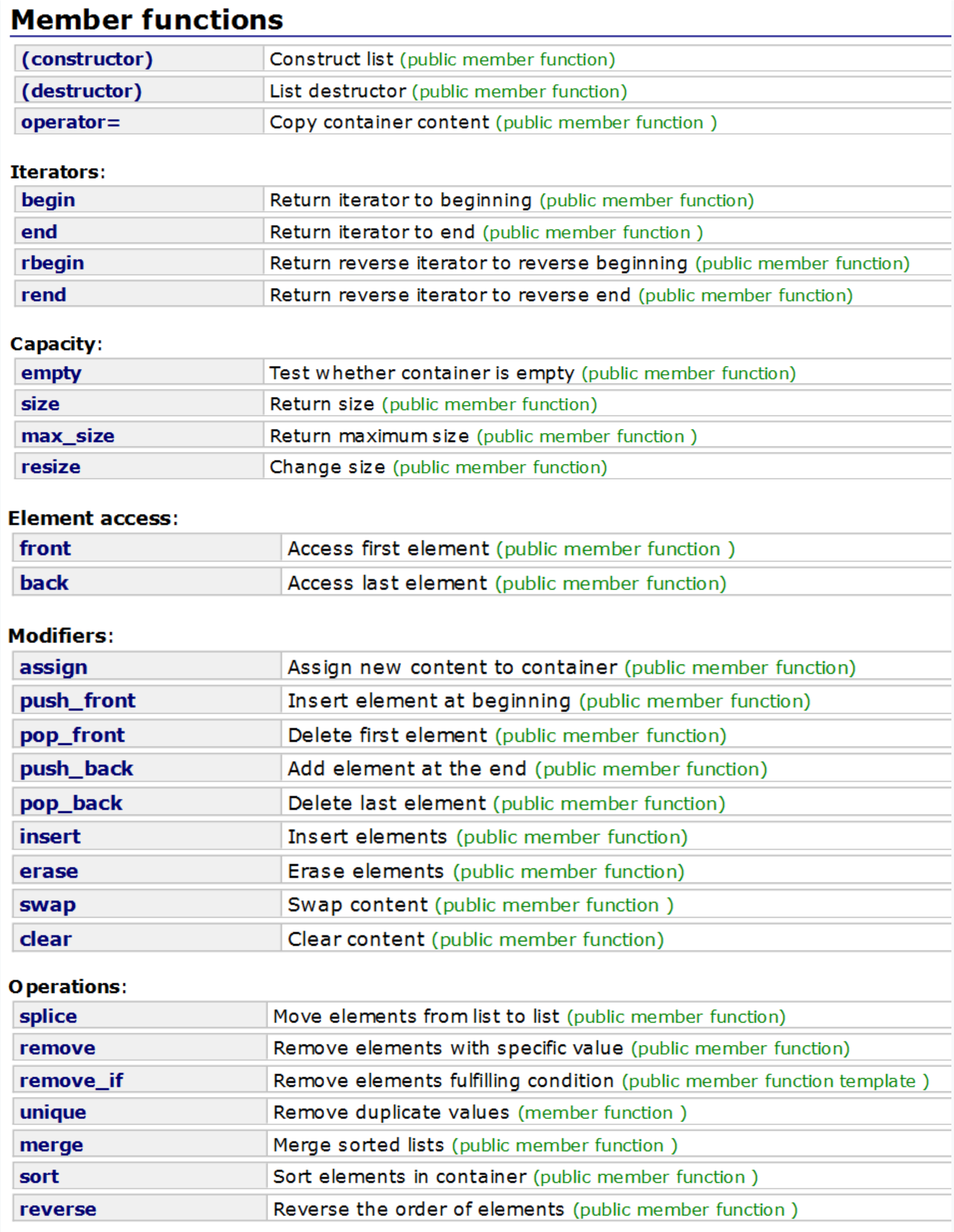
  

  3. list的迭代器

  因为list底层是带头结点的双向循环链表，因此list的迭代器需要list的提供者自己实现，否则当让迭代器++朝后走时，迭代器将不知道如何朝后移动，那迭代器该怎么实现呢？

  迭代器的本质是指针，是将指针封装出来的一种新的类型，因此指针有的操作，迭代器也要视情况支持这些操作。比如：指针可以++，–，*，->等操作，迭代器在其类中只需将这些操作重载出来即可。当利用迭代器来遍历list时，在迭代器上进行++操作，因为迭代器已经和链表绑定在一起，它内部根据自己底层的结构就知道如何朝后去移动。

  4. list的操作



Member functions

(constructor)	Construct list (public member function)
(destructor)	List destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)

Element access:

front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign new content to container (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

Operations:

splice	Move elements from list to list (public member function)
remove	Remove elements with specific value (public member function)
remove_if	Remove elements fulfilling condition (public member function template)
unique	Remove duplicate values (member function)
merge	Merge sorted lists (public member function)
sort	Sort elements in container (public member function)
reverse	Reverse the order of elements (public member function)

```
// 验证list的构造、遍历
void TestList1()
{
  // 构造空的list
  list<int> l1;

  // 构造10个值为1的list
  list<int> l2(10, 1);

  // 构造用一段区间中的元素构造list
  int array[] = {1,2,3,4,5,6,7,8,9};
  list<int> l3(array, array+sizeof(array)/sizeof(array[0]));

  // 用l3去构造l4
```

```
    list<int> l4(l3);

    cout<<"l1中元素个数:"<<l1.size()<<endl;
    cout<<"l2中元素个数:"<<l2.size()<<endl;
    cout<<"l3中元素个数:"<<l3.size()<<endl;
    cout<<"l4中元素个数:"<<l4.size()<<endl;

    // 遍历list1
    list<int>::iterator it1 = l1.begin();
    while(it1 != l1.end())
    {
        cout<<*it1<<" ";
        ++it1;
    }
    cout<<endl;

    // 遍历list2
    list<int>::iterator it2 = l2.begin();
    while(it2 != l2.end())
    {
        cout<<*it2<<" ";
        ++it2;
    }
    cout<<endl;

    // 遍历list3
    list<int>::iterator it3 = l3.begin();
    while(it3 != l3.end())
    {
        cout<<*it3<<" ";
        ++it3;
    }
    cout<<endl;

    // 遍历list4
    list<int>::iterator it4 = l4.begin();
    while(it4 != l4.end())
    {
        cout<<*it4<<" ";
        ++it4;
    }
    cout<<endl;

    // 逆向遍历list4
    list<int>::reverse_iterator it5 = l4.rbegin();
    while(it5 != l4.rend())
    {
        cout<<*it5<<" ";
        ++it5;
    }
    cout<<endl;
```

```
}

// 测试链表的插入和删除
void TestList2()
{
  // 构造空的list
  list<int> l;

  // 尾插
  l.push_back(1);
  l.push_back(2);
  l.push_back(3);
  l.push_back(4);
  l.push_back(5);
  l.push_back(6);

  list<int>::iterator it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;

  // 尾删
  l.pop_back();
  l.pop_back();
  it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;

  // 头插
  l.push_front(0);
  it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;

  // 头删
  l.pop_front();
  it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
```

```
        ++it;
    }
    cout<<endl;

    // 任意位置的插入insert，该函数返回插入的新节点
    list<int>::iterator pos = find(l.begin(), l.end(), 3);
    if(pos != l.end())
        pos = l.insert(pos, 0);

    it = l.begin();
    while(it != l.end())
    {
        cout<<*it<<" ";
        ++it;
    }
    cout<<endl;

    // 任意位置删除
    l.erase(pos);
    it = l.begin();
    while(it != l.end())
    {
        cout<<*it<<" ";
        ++it;
    }
    cout<<endl;

    // 给list重新赋值
    int array[] = {1,2,3,4,5,6,7,8,9};
    l.assign(array, array+sizeof(array)/sizeof(array[0]));
    it = l.begin();
    while(it != l.end())
    {
        cout<<*it<<" ";
        ++it;
    }
    cout<<endl;
    cout<<"获取链表第一个元素:"<<l.front()<<endl;
    cout<<"获取链表最后一个元素:"<<l.back()<<endl;
}

class Odd
{
public:
    bool operator()(int value)
    {
        return value&0x01;
    }
};

// 测试remove/remove_if
```

```
void TestList3()
{
  int array[] = {1,2,3,4,3,5,3,3,6,7,8,9};
  list<int> l(array, array+sizeof(array)/sizeof(array[0]));
  list<int>::iterator it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;

  l.remove(3);
  it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;

  l.remove_if(Odd());
  it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;
}

class Three
{
public:
  bool operator()(int left, int right)
  {
      return 0 == (left + right)%3;
  }
};
// unique：将list中连在一起的相同数据删除，只保留第一个
void TestList4()
{
  int array[] = {1,2,3,4,5, 1,2,3,4,5,6,7,8,9};
  list<int> l(array, array+sizeof(array)/sizeof(array[0]));
  l.unique();

  list<int>::iterator it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;

  l.sort();
  it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;

  l.unique();
  it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;

  // 如果list中连续两个数之和为3的倍数时，删除第二个数
  l.unique(Three());
  it = l.begin();
  while(it != l.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;
}

// merge:将连个已序链表合并成一个链表，合并好之后依然有序
// reverse:链表的逆置
void TestList5()
{
  list<int> l1;
  l1.push_back(1);
  l1.push_back(4);
  l1.push_back(6);

  list<int> l2;
  l2.push_back(3);
  l2.push_back(4);
  l2.push_back(7);

  // 将两个已序链表合并成一个链表，合并好之后依然有序
  l1.merge(l2);
  list<int>::iterator it = l1.begin();
```

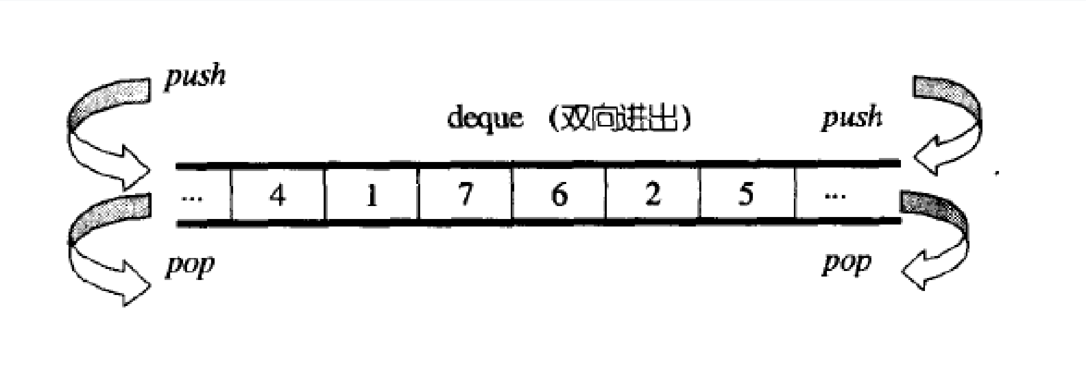
```
    while(it != l1.end())
    {
        cout<<*it<<" ";
        ++it;
    }
    cout<<endl;

    // 反转链表
    l1.reverse();
    it = l1.begin();
    while(it != l1.end())
    {
        cout<<*it<<" ";
        ++it;
    }
    cout<<endl;
}
```

面试题：vector和list有什么区别？分别在什么场景下应用？

- **deque**

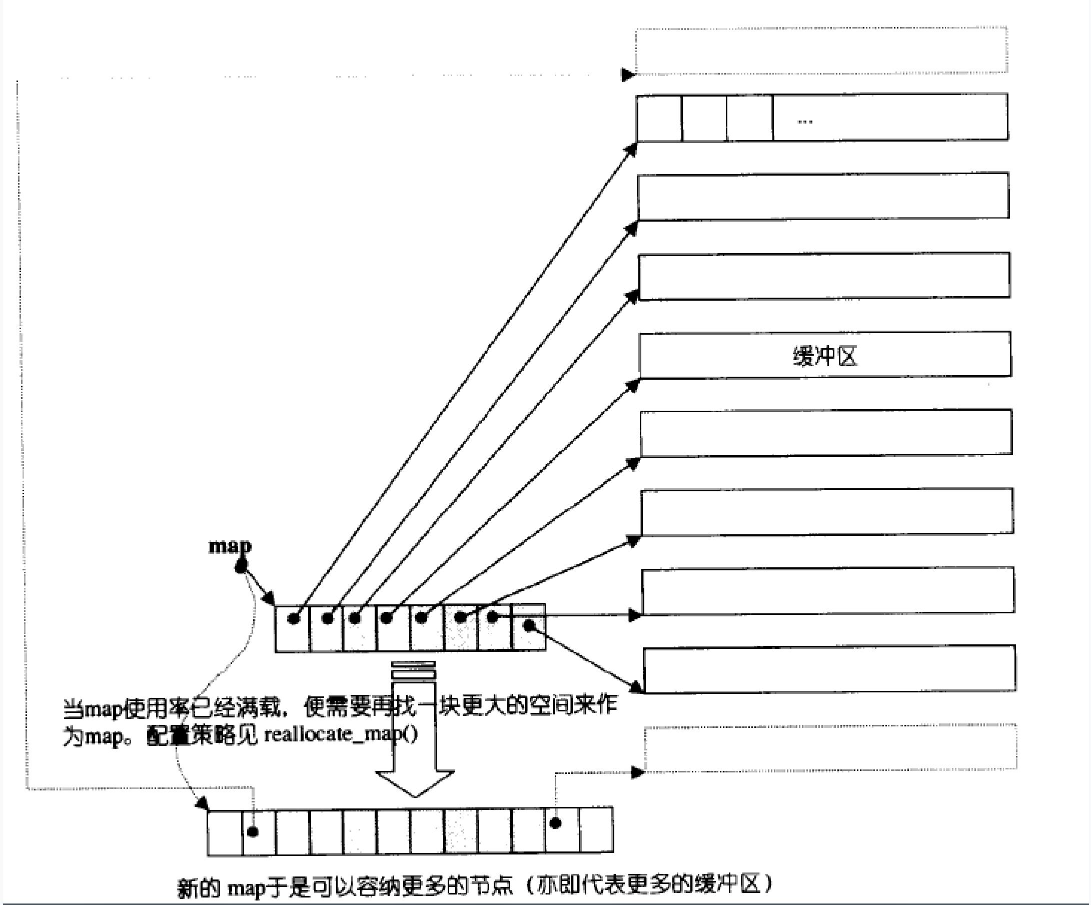
  1. deque简介

  deque： double ended queue，vector是单向开口的线性连续空间，deque是双向开口的线性连续空间。所谓双向开口：是指可以在头尾两端分别进行元素的插入和删除操作，因此deque允许常数时间内对头端进行数据的插入操作。

  2. deque底层结构

  deque是一段假想的连续空间，与vector不同。deque由一段一段定量的连续空间构成，一旦有必要再deque的前端或尾端增加新空间，便配置一段定量连续空间，串接在整个deque的头端和尾端，而deque的任意就是：在分段的定量连续空间上维护其连续的假象



缓冲区
map
当map使用率已经满载，便需要再找一块更大的空间来作为map。配置策略见 reallocate_map()
新的 map于是可以容纳更多的节点（亦即代表更多的缓冲区）

3. deque的迭代器

因为deque是分段连续空阿金，维护其“整体连续”假象的任务就落在其迭代器operator++和operator–上。因此deque的迭代器必须要知道分段连续空间在哪里，其次它必须能够识别出自己是否已经处于其所在某段连续空间的边缘，如果是，一旦前进或后退时就必须跳跃至下一个或上一个分段空间

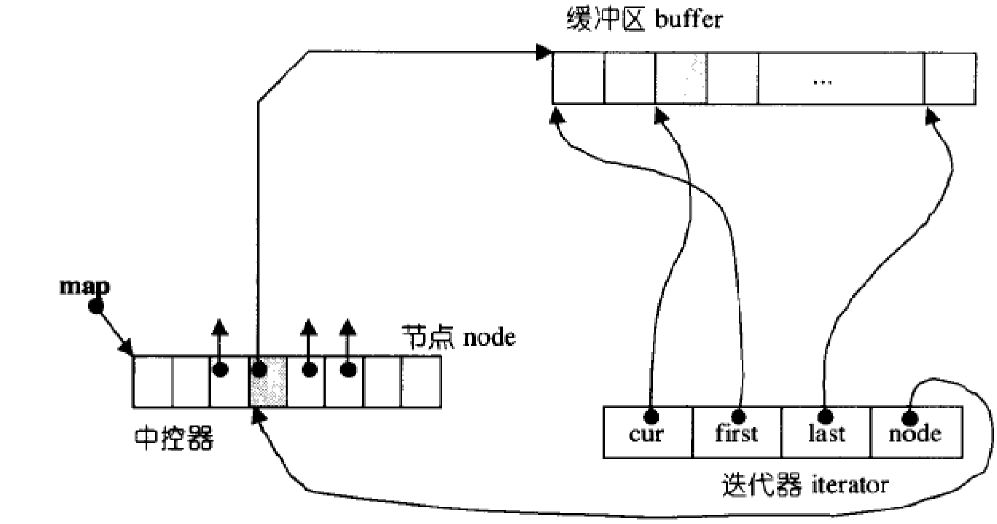
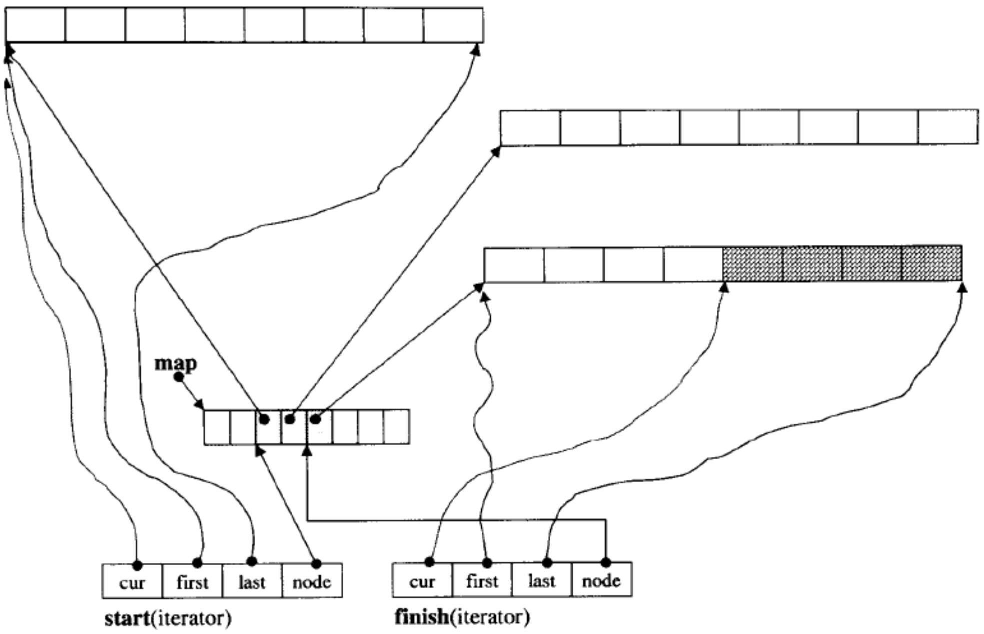
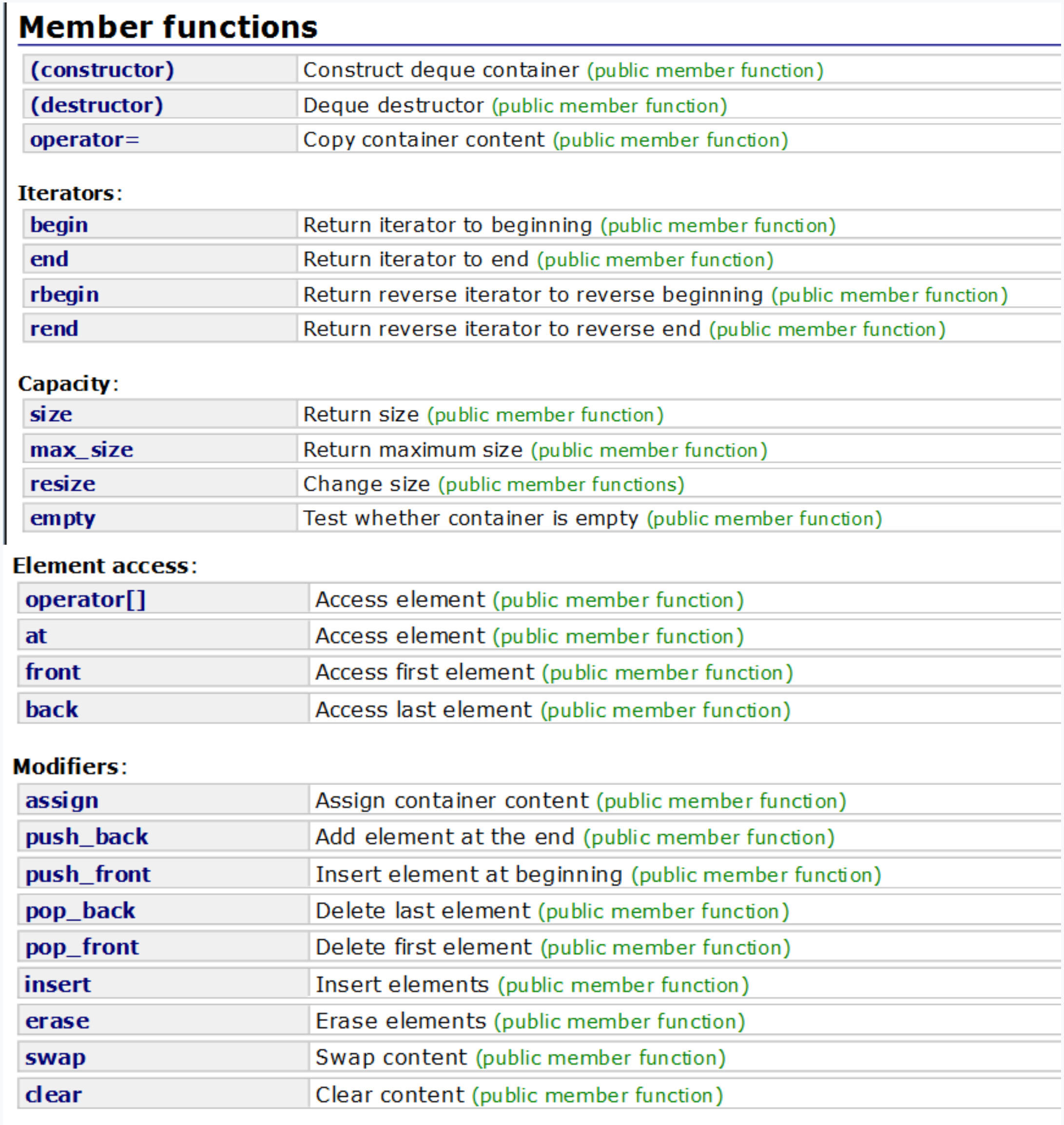


图 4-11　deque 的中控器、缓冲区、迭代器的相互关系



4. deque的操作

要使用deque时，必须包含其头文件和命名空间std



Member functions

(constructor)	Construct deque container (public member function)
(destructor)	Deque destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member functions)
empty	Test whether container is empty (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign container content (public member function)
push_back	Add element at the end (public member function)
push_front	Insert element at beginning (public member function)
pop_back	Delete last element (public member function)
pop_front	Delete first element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

面试题：

1. deque与vector的比较

2. 怎么对deque 中的元素排序

适配器

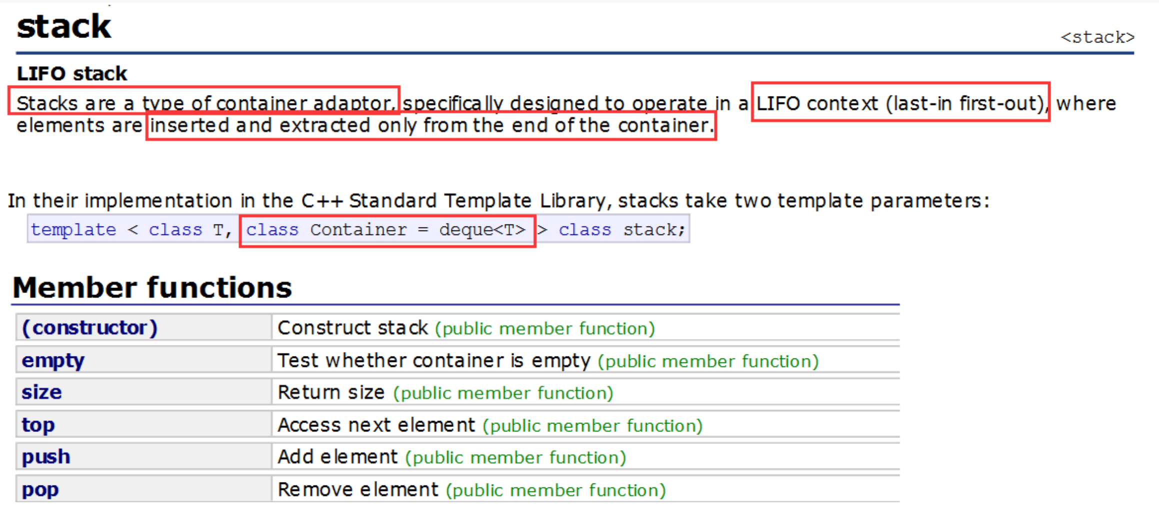
适配器也是一种设计模式，该中模式是将一个类的接口转换成用户希望的另外一个接口。简单的说：就是需要的东西就在眼前，但却不能用或者使用不是很方便，而段时间又无法改造它，那我们就通过已存在的东西去适配它。

STL中的适配器共有三种，应用于容器的即容器适配器，比如stack和queue就是对deque的接口进行了转调；应用于迭代器的即迭代器适配器，比如反向迭代器就是对迭代器的接口进行了转调；应用于仿函数的即函数适配器

此处只对容器适配器进行介绍，其他请参考《STL源码剖析》。

容器适配器
stack和queue都是一种特殊的线性数据结构，要求在其固定端进行数据的插入和删除操作；比如：
stack要求在其一端进行数据的插入和删除即入栈和出栈，该段称为stack的栈顶，另一端称为栈底；因此stack是一种后进先出的线性结构
queue要求在尾部进行数据的插入操作即入队列，在其头部进行数据的删除即出队列，因此queue是一种后进先出的线性结构
deque是双开口的结构，因此STL将其作为栈和队列的底层结构，将deque稍加改装就实现出stack和队列。像这种：将某个类的接口进行重新包装而实现出的新结构，称之为适配器

stack的操作



stack <stack>

LIFO stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from the end of the container.

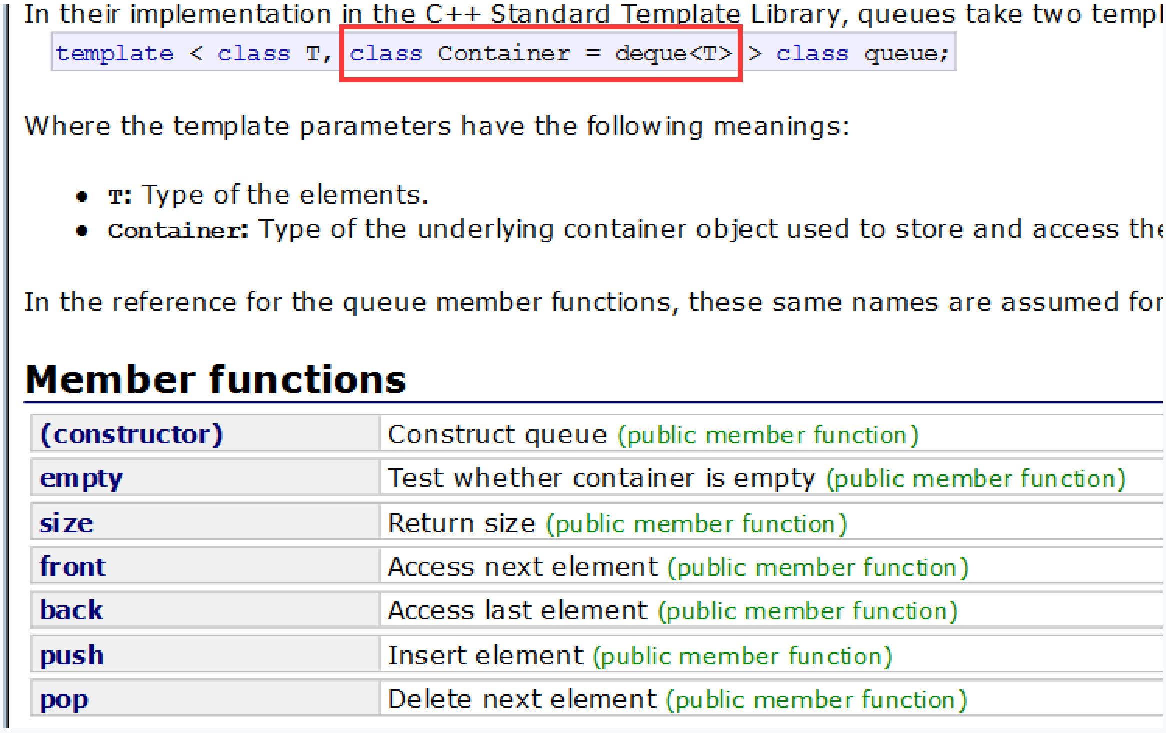
In their implementation in the C++ Standard Template Library, stacks take two template parameters:

```
template < class T, class Container = deque<T> > class stack;
```

Member functions

(constructor)	Construct stack (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access next element (public member function)
push	Add element (public member function)
pop	Remove element (public member function)

queue的操作



In their implementation in the C++ Standard Template Library, queues take two templ

```
template < class T, class Container = deque<T> > class queue;
```

Where the template parameters have the following meanings:

- **T:** Type of the elements.
- **Container:** Type of the underlying container object used to store and access the

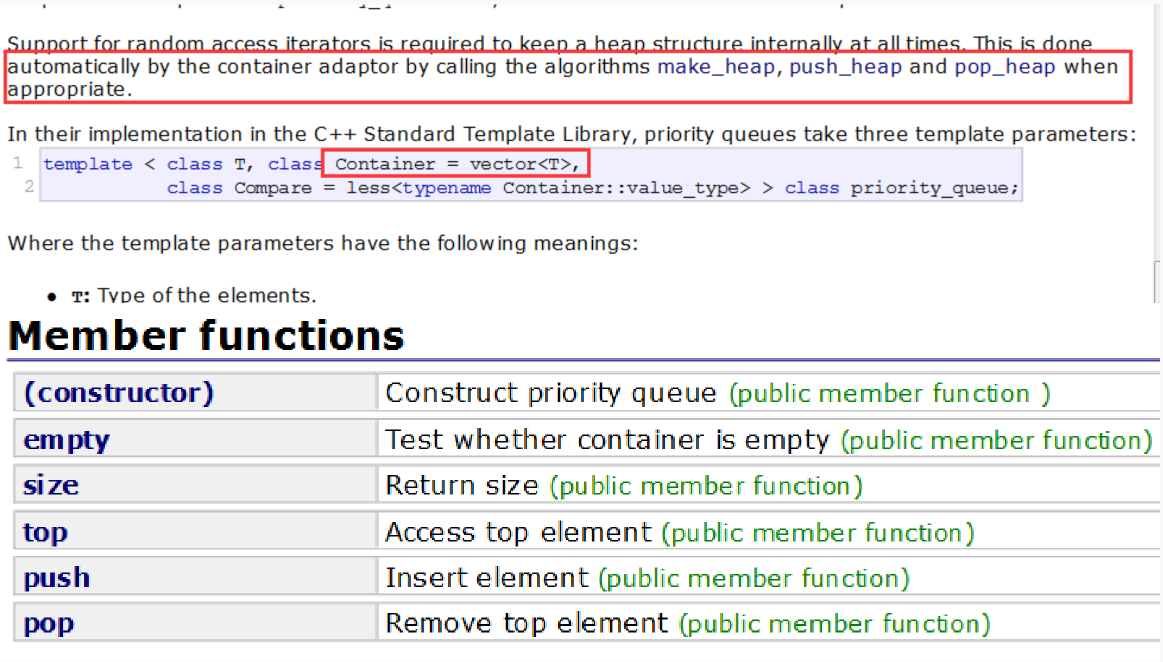
In the reference for the queue member functions, these same names are assumed for

Member functions

(constructor)	Construct queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
front	Access next element (public member function)
back	Access last element (public member function)
push	Insert element (public member function)
pop	Delete next element (public member function)

思考：stack和queue为什么没有迭代器？

priority_queue
priority_queue是一个拥有权值关键的队列，允许用户以任意次序将元素插入容器内，但取出时每次都是取优先级最高(低)的元素，这正是heap所具有该特性，因此priority_queue以vector作为底层存储元素空间，将heap算法进行包装，实现出了优先级队列



Support for random access iterators is required to keep a heap structure internally at all times. This is done automatically by the container adaptor by calling the algorithms make_heap, push_heap and pop_heap when appropriate.

In their implementation in the C++ Standard Template Library, priority queues take three template parameters:

```
template < class T, class Container = vector<T>,
           class Compare = less<typename Container::value_type> > class priority_queue;
```

Where the template parameters have the following meanings:

- **T:** Type of the elements.

Member functions

(constructor)	Construct priority queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access top element (public member function)
push	Insert element (public member function)
pop	Remove top element (public member function)

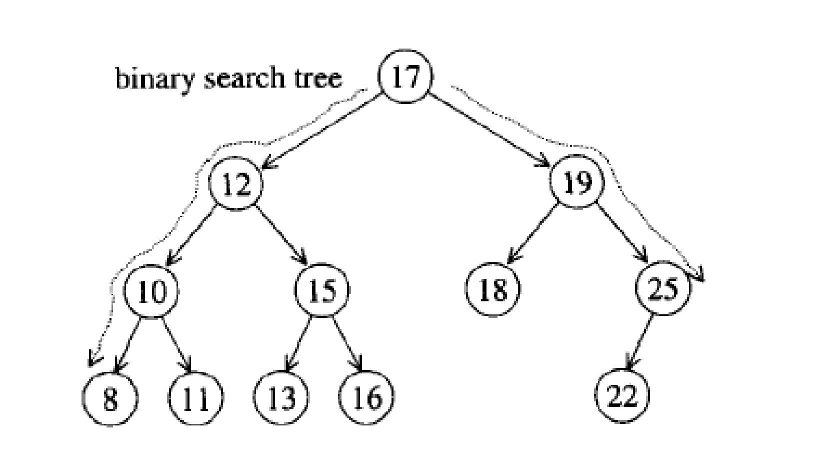
关联式容器

关联式容器中存放的是一个一个的键值对。
键值对其实是一个结构体，该结构体中有两个字段，一个为key，一个为value，key和

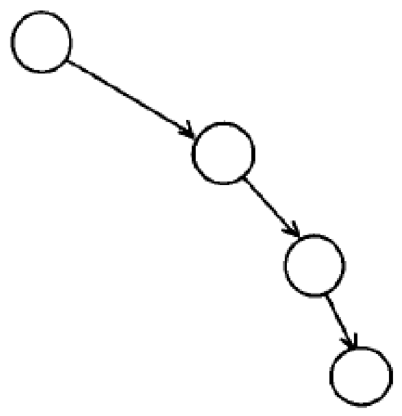
value具有一一对应的关系。比如：apple—苹果，banana—香蕉。
因此关联式容器中实际存放的是键值对的结构体。

通过前面序列式容器的学习，我们知道元素大小杂乱无章没有次序，因此在其中找某个元素只能使用顺序查找，其时间复杂度为O(N)，查找效率有点低下，因此关联式容器为了提高查询效率，将其底层结构设计为平衡搜索树结构，查找起来非常方便

- 二叉搜索树
  二叉搜索树又称二叉排序树，它或者是一棵空树，或者是具有以下性质的二叉树：
  1. 若它的左子树不为空，则左子树上所有节点的值都小于根节点的值
  2. 若它的右子树不为空，则右子树上所有节点的值都大于根节点的值
  3. 它的左右子树也分别为二叉搜索树



  但是，如果向容器中放置的元素恰巧是有序或者接近有序，上述二叉搜索树将会退化为单支树，查询效率又退化成线性结构



不平衡树

  因此，序列式容器底层并没有直接采用此二叉搜索树结构，而是对二叉搜索树进行优化，采用了平衡树结构

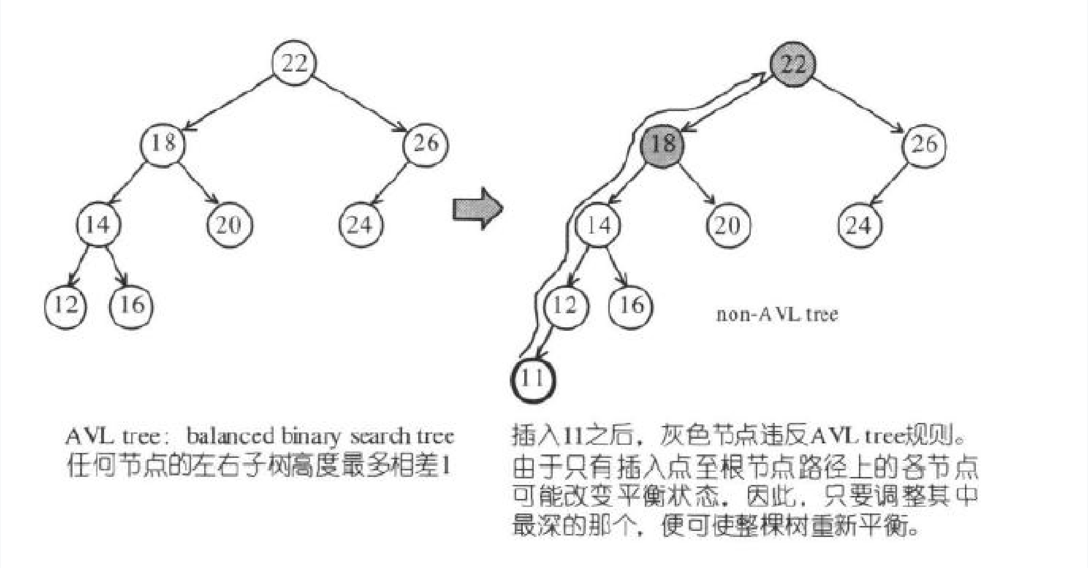
  二叉平衡树：即为左右子树高度差的绝对值不超过1(平衡因子)，则认为该树平衡，最典型的平衡树为AVL树。

- AVL树

一棵AVL树或者是空树，或者是具有以下性质的二叉搜索树：

1. 它的左右子树都是AVL树

2. 左子树和右子树高度之差(简称平衡因子)的绝对值不超过1(-1、0、1)


22
18
26
14
20
24
12
16
22
18
26
14
20
24
12
16
11
non-AVL tree
AVL tree: balanced binary search tree
任何节点的左右子树高度最多相差1
插入11之后，灰色节点违反AVL tree规则。
由于只有插入点至根节点路径上的各节点
可能改变平衡状态，因此，只要调整其中
最深的那个，便可使整棵树重新平衡。

当插入新节点后，AVL树的平衡性有可能会被破坏，一旦破坏，就必须对树的结构进行旋转处理，以保证AVL树的平衡性。根据插入元素的位置不同，AVL树的旋转分为四种情况：

1. 插入新节点位于较高左子树左侧—-左左—右单旋

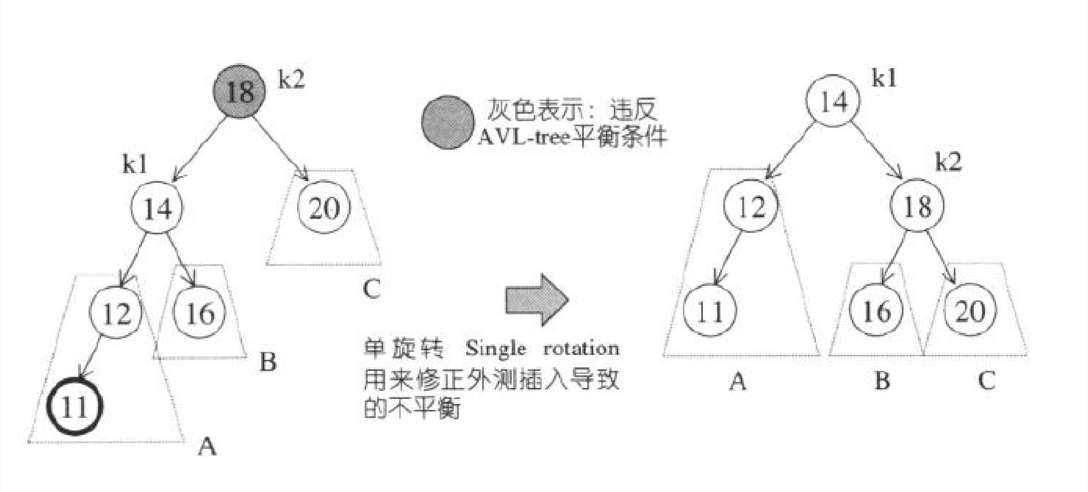
2. 插入新节点位于较高右子树右侧—-右右—左单旋

3. 插入新节点位于较高左子树右侧—-左右—左右双旋

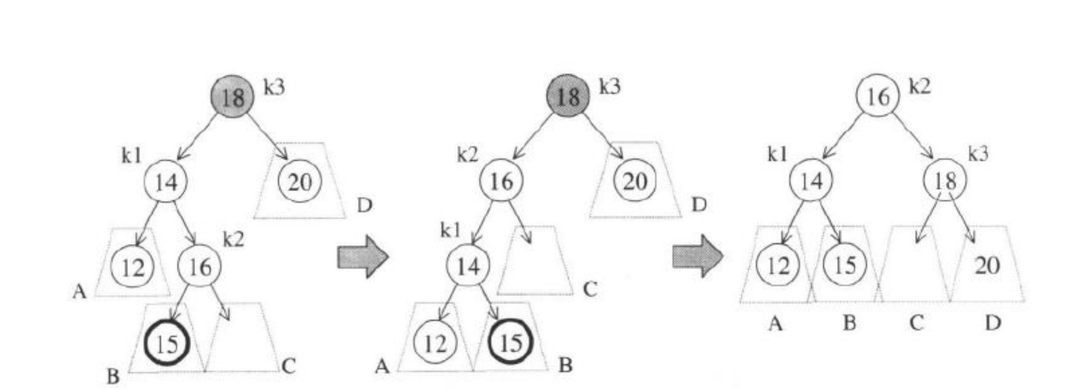
4. 插入新节点位于较高右子树左侧—-右左—右左双旋

以下只介绍两种旋转，详细请参考数据结构数据

左单旋



左右双旋

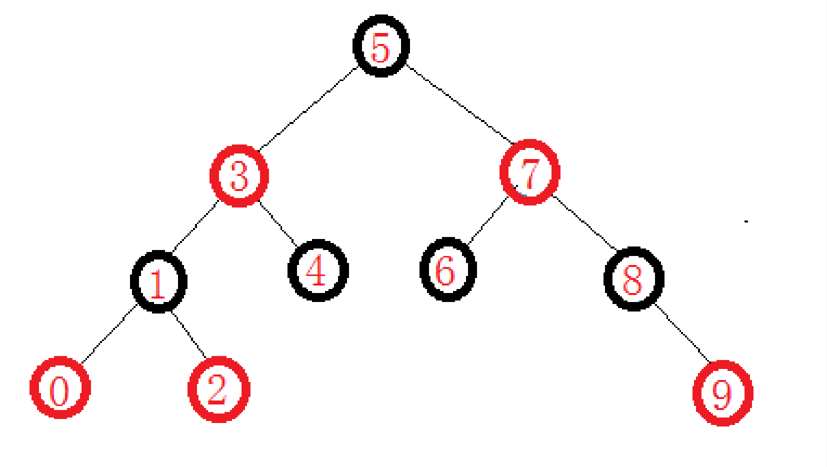


先对 k1, k2做一次单旋转 再对 k3, k2 做一次单旋转

AVL树中插入或者删除元素后，只要发现树的平衡性被破坏，就会自动进行旋转处理，而保证每个结点的左右子树高度差的绝对值不超过1，因此其查找效率可以达到$log2(N)$

因为AVL树中只要发现不平衡，就要进行旋转，即只要进行数据的插入或者删除可能要进行大量的旋转，因此AVL树在进行插入和删除时的性能不是很好，因此序列式容器底层没有使用AVL树，而采用了另一种效率较高的近似平衡的一棵二叉搜索树：红黑树

- 红黑树
  红黑树是一棵二叉搜索树，它在每个结点上增加了一个存储位来表示结点的颜色，可以是red或者black，通过对任何一条从根节点到叶子结点简单路径上的颜色来约束，红黑树保证最长路径不超过最短路径的两倍，因而近似平衡。



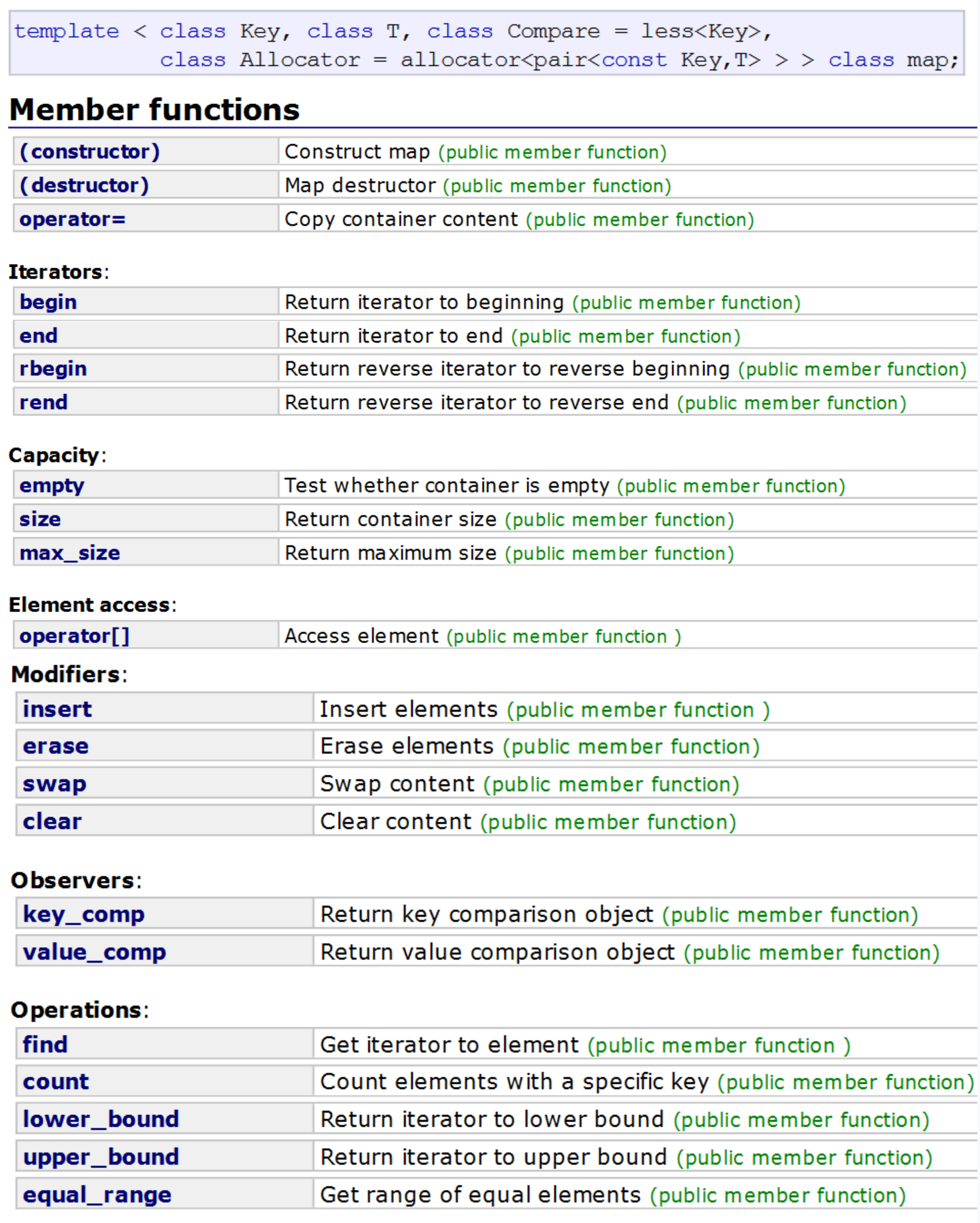
红黑树性质：
1. 每个结点不是红色就是黑色
2. 树的根节点是黑色的
3. 如果一个节点是红色的，则它的两个孩子结点是黑色的
4. 对于每个结点，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色结点
5. 每个叶子结点都是黑色的(此处的叶子结点指的是空结点)
此处不对红黑树做详细介绍，同学们可参考先关数据结构书籍

虽然红黑树是一棵近似平衡的搜索树，但是在实际应用中表现出的性能确实比AVL树更优，因此关联式容器选择将红黑树作为其底层结构。

- map

map的特性是：所有元素都会根据元素的键值key自动排序，其中的每个元素都是<key, value>的键值对，map中不允许有键值相同的元素，因此map中元素的键值key不能修改，但是可以通过key修改与其对应的value。如果一定要修改与value对应的键值key，可将已存在的key删除掉，然后重新插入。

map的基本操作



```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key,T> > > class map;
```

Member functions

(constructor)	Construct map (public member function)
(destructor)	Map destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return container size (public member function)
max_size	Return maximum size (public member function)

Element access:

operator[]	Access element (public member function)

Modifiers:

insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

Observers:

key_comp	Return key comparison object (public member function)
value_comp	Return value comparison object (public member function)

Operations:

find	Get iterator to element (public member function)
count	Count elements with a specific key (public member function)
lower_bound	Return iterator to lower bound (public member function)
upper_bound	Return iterator to upper bound (public member function)
equal_range	Get range of equal elements (public member function)

map的应用举例

使用map建立《水浒传》中人物人名绰号对应表

```
void TestMap1()
{
 map<string, string> m;
  m.insert(pair<string, string>("鲁智深", "花和尚"));
  m.insert(make_pair("史进", "九纹龙"));
  m.insert(make_pair("公孙胜", "入云龙"));
  m.insert(make_pair("李逵", "黑旋风"));
  m.insert(make_pair("石秀", "拼命三郎"));
  m.insert(make_pair("宋江", "及时雨"));
  cout<<"map中元素的个数为: "<<m.size()<<endl;

  // 尝试插入键值应经存在但是value不同的元素
  m.insert(make_pair("李逵", "铁牛"));
  cout<<"map中元素的个数为: "<<m.size()<<endl;

  // 将map中所有元素输出来
  map<string, string>::iterator it = m.begin();
  while (it != m.end())
  {
      cout<<it->first<<"--->"<<it->second<<endl;
      ++it;
  }
  cout<<endl;

  // 通过key查找key所对应的value
  cout<<"石秀"<<"--->"<<m["石秀"]<<endl;

  // 修改key所对应的value
  cout<<"李逵"<<"--->"<<m["李逵"]<<endl;
  m["李逵"] = "铁牛";
  cout<<"李逵"<<"--->"<<m["李逵"]<<endl;

  // 通过[]访问键值对中的value时，如果没有该key，
  // map会建立一个默认value的键值对插入进去，然后将
  // 该value返回
  cout<<"王伦"<<"--->"<<m["王伦"]<<endl;
  cout<<"map中元素的个数为: "<<m.size()<<endl;

  // 删除
  m.erase(m.find("王伦"));
  cout<<"map中元素的个数为: "<<m.size()<<endl;

// 将map中的元素清空
  m.clear();
  cout<<"map中元素的个数为: "<<m.size()<<endl;
}
```

如果一个好汉有多个绰号，需要将这些对应的绰号全部存起来，map还能处理吗？

- **multimap**
  multimap和map的唯一差别是map中key必须是唯一的，而multimap中的key是可以重复的

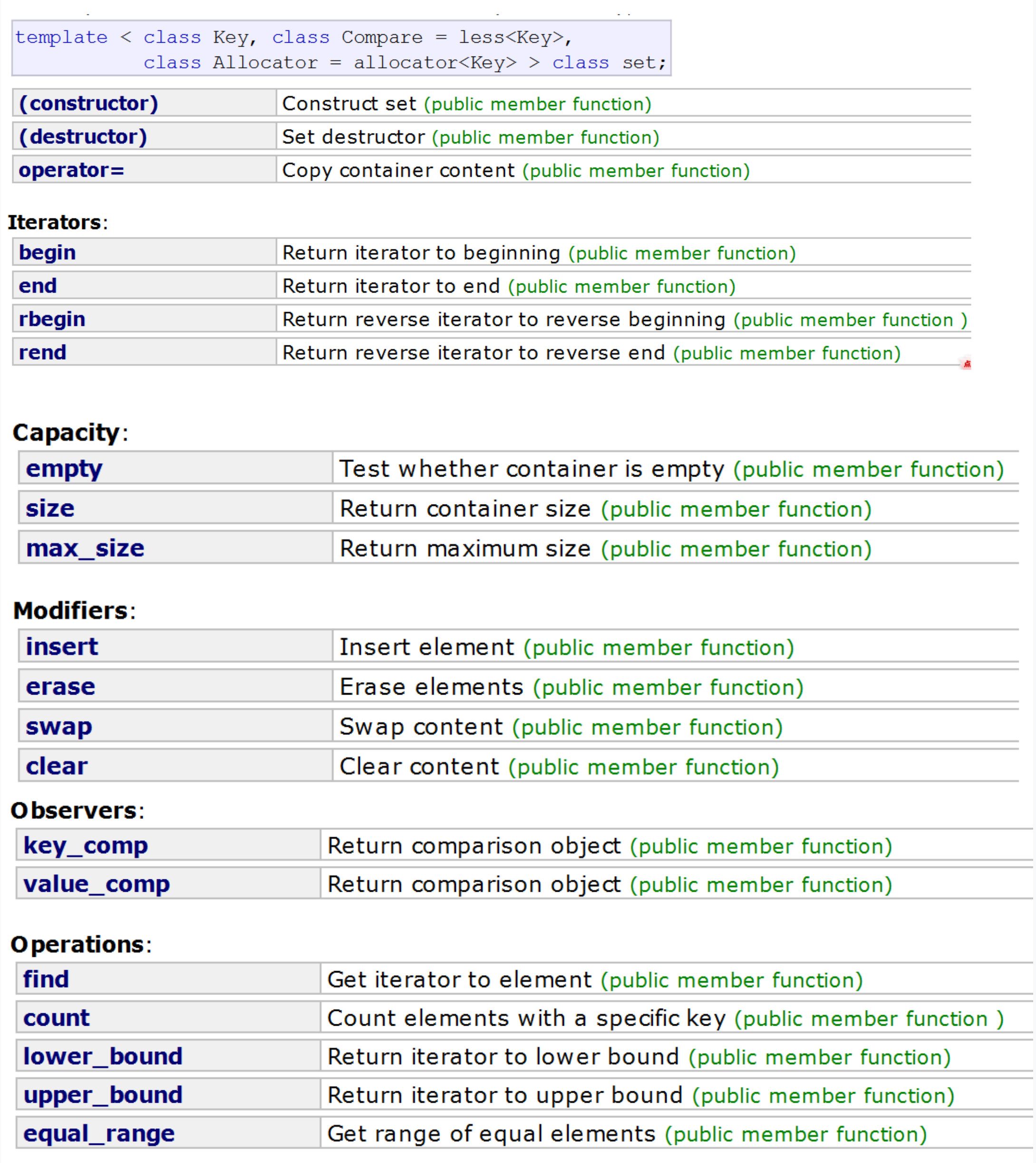
```
void TestMultimap()
{
 map<string, string> m;
  m.insert(pair<string, string>("鲁智深", "花和尚"));
  m.insert(make_pair("史进", "九纹龙"));
  m.insert(make_pair("公孙胜", "入云龙"));
  m.insert(make_pair("李逵", "黑旋风"));
  m.insert(make_pair("石秀", "拼命三郎"));
  m.insert(make_pair("宋江", "及时雨"));
  cout<<"map中元素的个数为: "<<m.size()<<endl;

  // 尝试插入键值应经存在但是value不同的元素
  m.insert(make_pair("李逵", "铁牛"));
  cout<<"map中元素的个数为: "<<m.size()<<endl;
}
<br>
void TestMultimap()
{
  multimap<int, int> m;
  for(int i = 0; i < 10; ++i)
      m.insert(make_pair(i, i));

  m.insert(make_pair(5, 11));
  m.insert(make_pair(5, 12));
  m.insert(make_pair(5, 13));
  m.insert(make_pair(5, 14));

  multimap<int, int>::iterator it = m.begin();
  while(it != m.end())
  {
      cout<<it->first<<"--->"<<it->second<<endl;
      ++it;
  }
  cout<<endl;
  cout<<"key为5的元素个数: "<<m.count(5)<<endl;

  it = m.lower_bound(5);
  cout<<it->first<<"--->"<<it->second<<endl;

  it = m.upper_bound(5);
  cout<<it->first<<"--->"<<it->second<<endl;

  typedef multimap<int, int>::iterator Iterator;

  pair<Iterator, Iterator> ret = m.equal_range(5);
  cout<<(ret.first)->first<<"--->"<<(ret.first)->second<<endl;
  cout<<(ret.second)->first<<"--->"<<(ret.second)->second<<endl;
}
```

- set

  set同map类似，所有元素都会根据元素的键值自动排序，与map不同的是：set中元素不像map那样可以同时拥有键值key和实值value，set中key就是value，value就是key。由于红黑树自动排序效果很不错，因此set也以红黑树作为其底层的数据结构，虽然set提供给用户的接口只需存放value，但实际其底层结构仍旧构建了一个键值对<key, key>

  问题：set中的元素能修改吗？怎么修改

  set的基本操作



```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> > class set;
```

(constructor)	Construct set (public member function)
(destructor)	Set destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return container size (public member function)
max_size	Return maximum size (public member function)

Modifiers:

insert	Insert element (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

Observers:

key_comp	Return comparison object (public member function)
value_comp	Return comparison object (public member function)

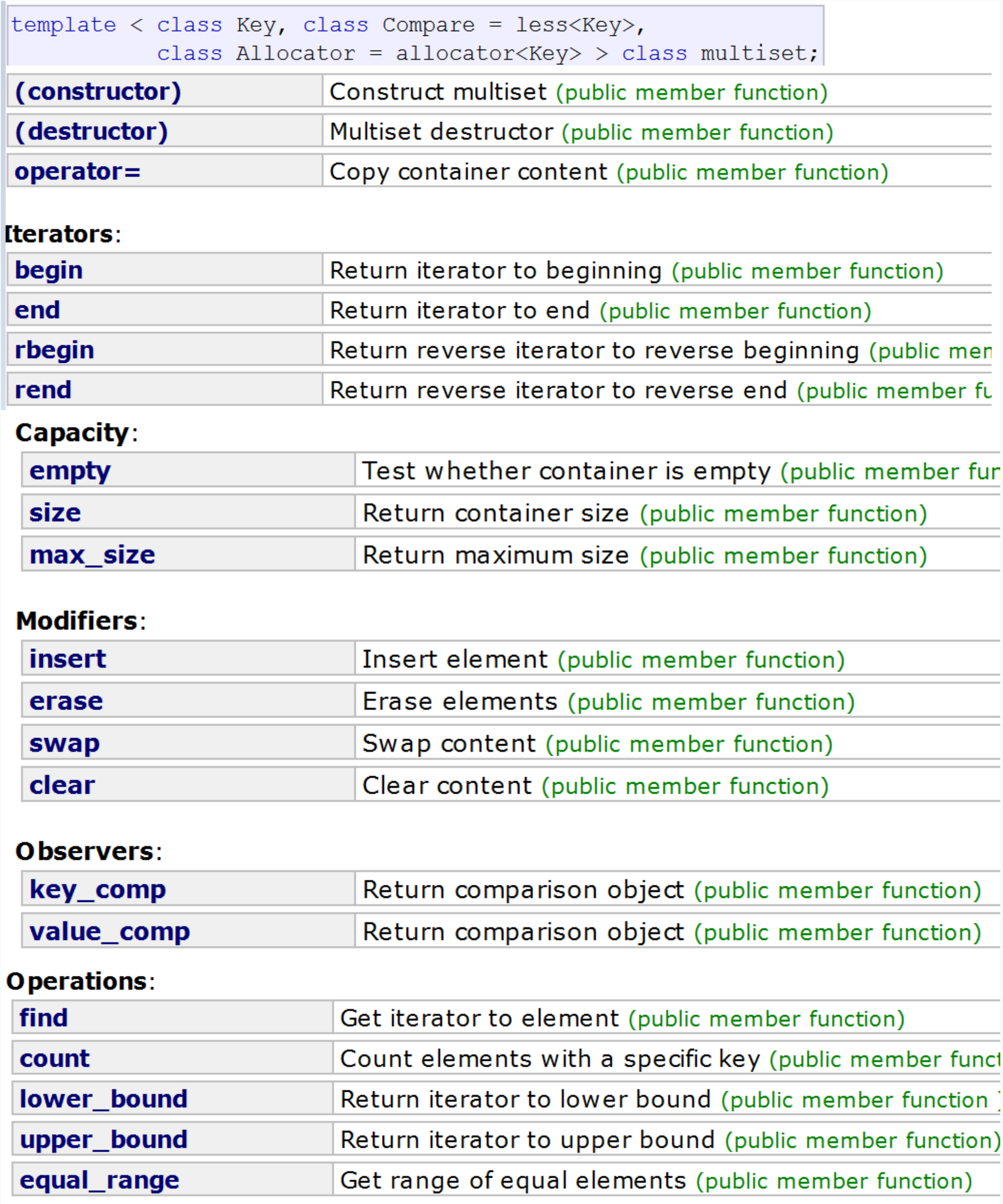
Operations:

find	Get iterator to element (public member function)
count	Count elements with a specific key (public member function)
lower_bound	Return iterator to lower bound (public member function)
upper_bound	Return iterator to upper bound (public member function)
equal_range	Get range of equal elements (public member function)

```
// 用set对集合中的元素去重
void TestSet()
{
  int array[] = {1,2,3,3,4,2,8,2,1,4,5,7,6,9};
  set<int> s;
  for(int i = 0; i < sizeof(array)/sizeof(array[0]); ++i)
      s.insert(array[i]);

  cout<<s.size()<<endl;

  set<int>::iterator it = s.begin();
  while(it != s.end())
  {
      cout<<*it<<" ";
      ++it;
  }
  cout<<endl;
}
<br>
void TestSet2()
{
  int array[] = {1,2,3,4,5};
  set<int> s(array, array+sizeof(array)/sizeof(array[0]));
  cout<<s.size()<<endl;
  cout<<s.count(3)<<endl;

  s.insert(3);
  cout<<s.size()<<endl;
  cout<<s.count(3)<<endl;

  s.insert(6);
  cout<<s.size()<<endl;

  s.erase(1);
  cout<<s.size()<<endl;

  // 使用STL算法find 元素1
  set<int>::iterator it = find(s.begin(), s.end(), 1);
  if(it != s.end())
      cout<< "1 is in set!!!"<<endl;
  else
      cout<<"1 is not in set!!!"<<endl;

  // 使用set中的find算法查找元素3
  it = s.find(3);
  if(it != s.end())
      cout<< "1 is in set!!!"<<endl;
  else
      cout<<"1 is not in set!!!"<<endl;

  // set中的元素不能被修改，会破坏树的结构，因此其迭代器实际为const类型迭代器
  it = s.begin();
  //*it = 10;
}
```

- **multiset**
  multiset与set类似，唯一的不同就是multiset中的元素可以重复，而set不能重复，接口也基本类似

multiset的操作



```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> > class multiset;
```

(constructor)	Construct multiset (public member function)
(destructor)	Multiset destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public men
rend	Return reverse iterator to reverse end (public member fu

Capacity:

empty	Test whether container is empty (public member fur
size	Return container size (public member function)
max_size	Return maximum size (public member function)

Modifiers:

insert	Insert element (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

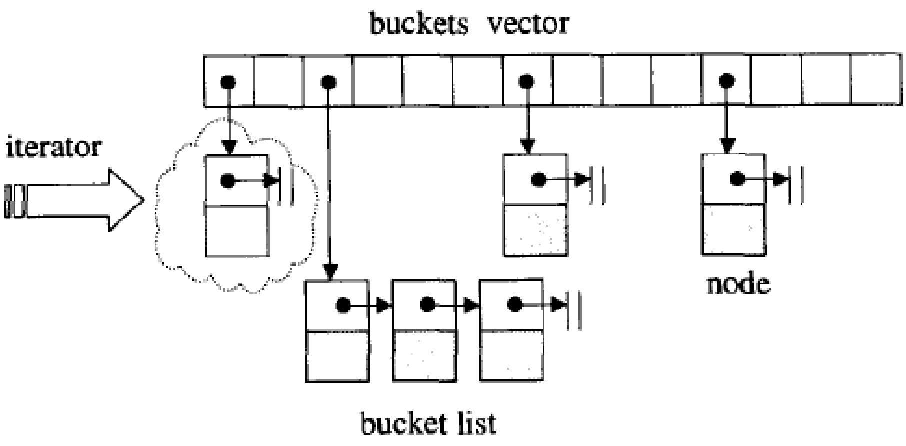
Observers:

key_comp	Return comparison object (public member function)
value_comp	Return comparison object (public member function)

Operations:

find	Get iterator to element (public member function)
count	Count elements with a specific key (public member funct
lower_bound	Return iterator to lower bound (public member function)
upper_bound	Return iterator to upper bound (public member function)
equal_range	Get range of equal elements (public member function)

上述介绍的关联式容器，其底层均为红黑树，只要给的随机足够随机，其插入的效果还是比较好的，要在其中查找某个数据，可提现出$log2(N)$的时间复杂度，但是哈希的查询效率可以体现在常数范围内，因此C++11中又给出了以哈希桶为底层数据结构的关联式容器。

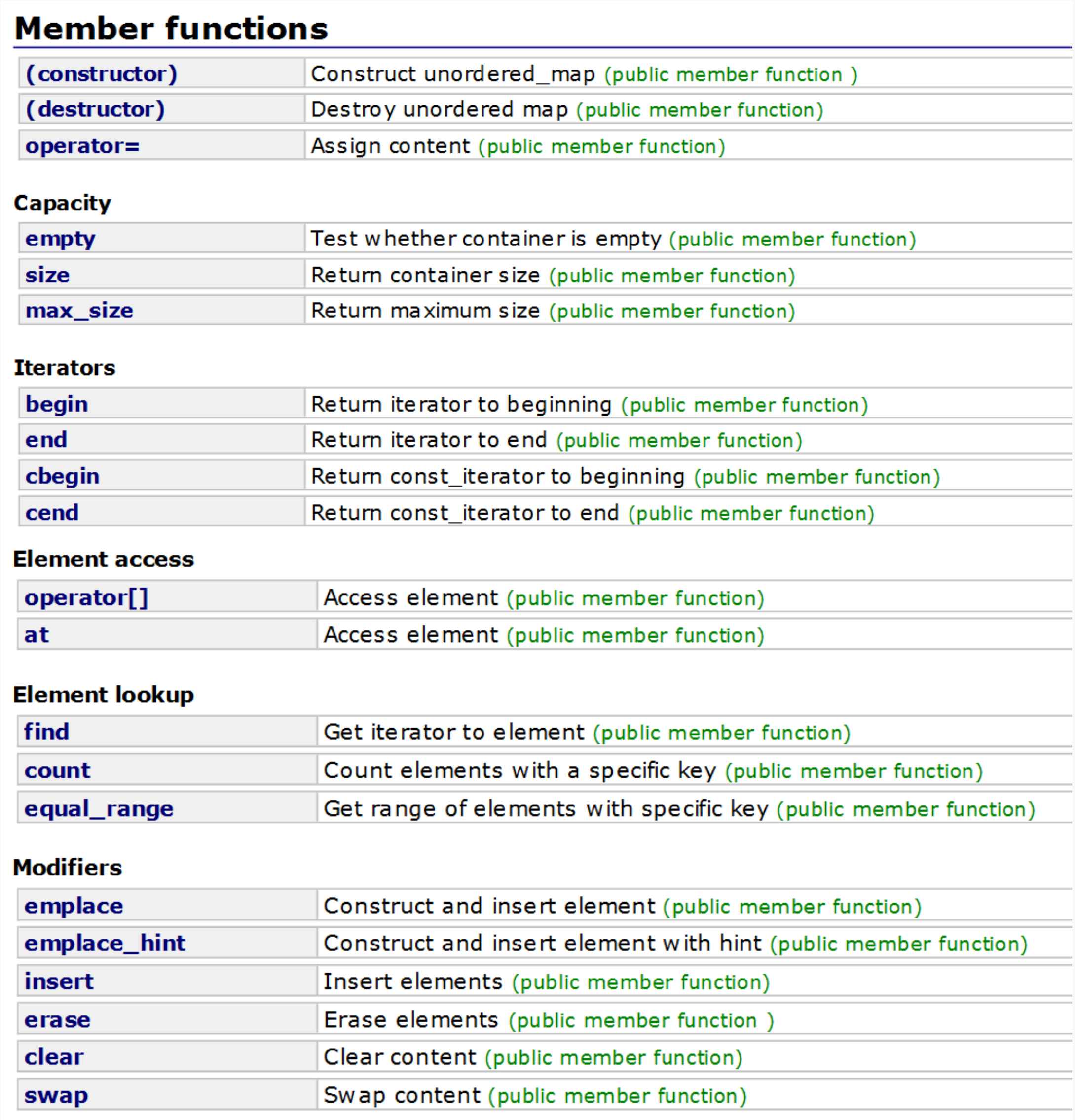


以开链（separate chaining）法完成的 hash table。

unordered_map/unordered_set/unordered_multimap/unordered_multiset这4个关联式容器与map/multimap/set/multiset功能基本类似，最主要就是底层结构不同，使用场景不容。如果需要得到一个有序序列，使用红黑树系列的关联式容器，如果需要更高的查询效率，使用以哈希表为底层的关联式容器。

此处只列举unordered_map，其他请同学们参考帮助文档。

unordered_map基本操作



Member functions

(constructor)	Construct unordered_map (public member function)
(destructor)	Destroy unordered map (public member function)
operator=	Assign content (public member function)

Capacity

empty	Test whether container is empty (public member function)
size	Return container size (public member function)
max_size	Return maximum size (public member function)

Iterators

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)

Element access

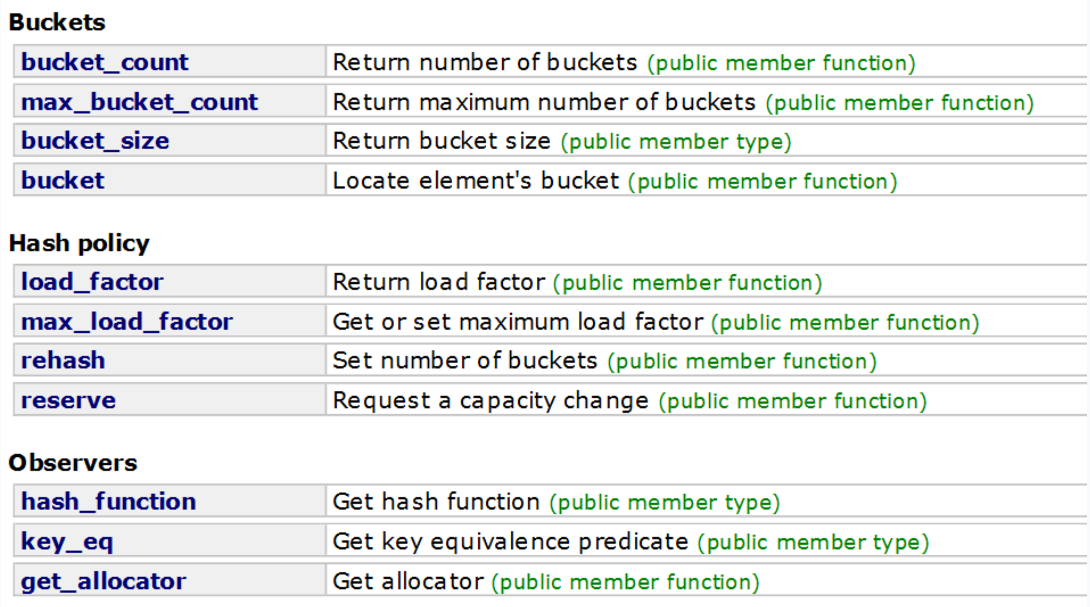
operator[]	Access element (public member function)
at	Access element (public member function)

Element lookup

find	Get iterator to element (public member function)
count	Count elements with a specific key (public member function)
equal_range	Get range of elements with specific key (public member function)

Modifiers

emplace	Construct and insert element (public member function)
emplace_hint	Construct and insert element with hint (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
clear	Clear content (public member function)
swap	Swap content (public member function)



Buckets

bucket_count	Return number of buckets (public member function)
max_bucket_count	Return maximum number of buckets (public member function)
bucket_size	Return bucket size (public member type)
bucket	Locate element's bucket (public member function)

Hash policy

load_factor	Return load factor (public member function)
max_load_factor	Get or set maximum load factor (public member function)
rehash	Set number of buckets (public member function)
reserve	Request a capacity change (public member function)

Observers

hash_function	Get hash function (public member type)
key_eq	Get key equivalence predicate (public member type)
get_allocator	Get allocator (public member function)

```
void Testunordered_map()
{
  unordered_map<int, int> m;

  //默认下桶的个数
  size_t bucket = m.bucket_count();
  cout<<bucket<<endl;

  // 最大桶的个数
  cout<<m.max_bucket_count()<<endl;

  // 负载因子：有效元素个数/桶的个数  默认情况下为1.0
  // 当哈希表中元素存储到一定个数时(负载因子)需要重新哈希
  // 否则冲突太高会影响哈希的效率
  cout<<m.load_factor()<<endl;

  for (size_t i = 0; i < bucket; ++i)
      m.insert(pair<int, int>(i, i));

  cout<<"桶的个数为: "<<m.bucket_count()<<endl;
  cout<<"map中元素的个数为:"<<m.size()<<endl;
  unordered_map<int,int>::iterator it = m.begin();
  while(it != m.end())
  {
      cout<<"<"<<it->first<<","<<it->second<<">"<<endl;
      ++it;
  }

  m.emplace(pair<int,int>(9,9));
  cout<<"桶的个数为: "<<m.bucket_count()<<endl;
  cout<<"map中元素的个数为:"<<m.size()<<endl;

  it = m.begin();
  while(it != m.end())
```

```
    {
        cout<<"<"<<it->first<<","<<it->second<<">"<<endl;
        ++it;
    }

    // 通过key访问与key对应的value
    cout<<m[8]<<endl;
    cout<<m.at(9)<<endl;

    // 指定key在桶中的个数
    cout<<m.count(8)<<endl;

    // 指定key所在的桶号
    cout<<m.bucket(5)<<endl;

    // 指定桶中元素的个数
    cout<<m.bucket_size(3)<<endl;

    // 按照设置的值进行重新哈希
    m.rehash(10);
    cout<<"桶的个数为: "<<m.bucket_count()<<endl;
    cout<<"map中元素的个数为:"<<m.size()<<endl;

    it = m.begin();
    while(it != m.end())
    {
        cout<<"<"<<it->first<<","<<it->second<<">"<<endl;
        ++it;
    }

    m.rehash(100);
    cout<<"桶的个数为: "<<m.bucket_count()<<endl;
    cout<<"map中元素的个数为:"<<m.size()<<endl;

    it = m.begin();
    while(it != m.end())
    {
        cout<<"<"<<it->first<<","<<it->second<<">"<<endl;
        ++it;
    }
}
```

迭代器

- **迭代器概念**
  迭代器(itreator)是一种抽象的设计概念，是设计模式的一种，其定义如下：提供

一种方法，使之能够依次寻访某个容器中所包含的所有元素，而又无需暴露该容器底层的结构

STL设计的中心思想在于：将数据容器和算法分离开，彼此独立设计，算法要操作容器中的元素时，通过迭代器去访问即可。因此，算法不需要去关心所操作数据底层的结构，只要能够按照迭代器去寻访到所需的数据即可，实现其通用性。比如：

```
template<class iterator, class T>
iterator find(iterator first, iterator last, const T& value)
{
    while(first != last && *first != value)
        ++first;

     return first;
}


void TestFind()
{
    int array[] = {1,2,3,4,5,6,7,8,9,0};
    vector<int> v(array, array+sizeof(array)/sizeof(array[0]));
    if(find(v.begin(), v.end(), 5) != v.end())
        cout<<"5 is in vector"<<endl;
    else
        cout<<"5 is not in vector"<<endl;

    list<int> l(array, array+sizeof(array)/sizeof(array[0]));
    if(find(l.begin(), l.end(), 5) != l.end())
        cout<<"5 is in list"<<endl;
    else
        cout<<"5 is not in list"<<endl;
}
```

- 迭代器本质
  迭代器实际是一种行为类似指针的对象，因此指针的所有操作迭代器都必须要支持，使用迭代器时可以像使用指针一样去使用。比如：指针的解引用、成员访问、前置/后置++，前置/后置–，==， ！=等 迭代器都要支持，而迭代器是一种行为类似指针的新类型，因此迭代器的实现只需将指针的上述操作在类中重载即可。

- 迭代器的实现
  迭代器是算法和容器的粘合剂，同一个算法可以操作不同类型的容器，而容器类型不容，意味着底层数据结构不容，数据结构不同，迭代器寻访的方式就不同，那迭代器是如何知道按照某种方式去寻访呢？

> **答案就是：容器的设计者负责该容器迭代器的实现，根据容器底层数据结构的特点，选择指针的相应操作去重载即可。比如vector和list的迭代器就不同，因为vector底层是一段连续的空间，其迭代器可以直接搭载原生态指针，而list底层是带头结点的双向循环链表，其迭代器实现需将原生态指针重新进行封装，最后再以统一的接口出现即可**
>
> **各个迭代器的实现，请参考《STL源码剖析》中各容器的介绍，此处可以暂时忽略，只需了解迭代器的原理以及能够正确使用即可**
>
> **面试题：什么是迭代器失效？**

仿函数

> **仿函数：又叫函数对象，一种行为类似函数的对象，调用者可以向函数一样使用该对象。其实现起来也比较简单：用户只需要实现一种新类型，在类中重载()即可，参数根据用户所要进行的操作选择匹配**
> **面试题：写一个冒泡排序**

```
template<class T>
class Less
{
public:
  bool operator()(const T& left, const T& right)
  {
      return left < right;
  }
};

template<class T>
class Greater
{
public:
  bool operator()(const T& left, const T& right)
  {
      return left > right;
  }
};

template<class T,class Com>
void BubbleSort(T* array, size_t size, Com Compare)
{
  bool isChange = false;
  for(size_t i = 0; i < size - 1; ++i)
  {
      isChange = false;
      for(size_t j = 0; j < size - i - 1; ++j)
      {
          if(Compare(array[j+1], array[j]))
          {
              swap(array[j], array[j+1]);
              isChange = true;
          }
      }

      if(!isChange)
          return;
  }
}

void TestFuncObj()
{
  int array[] = {2,1,5,4,9,8,6,0,7,3};
  BubbleSort(array, 10, Less<int>());
  BubbleSort(array, 10, Greater<int>());
}
```

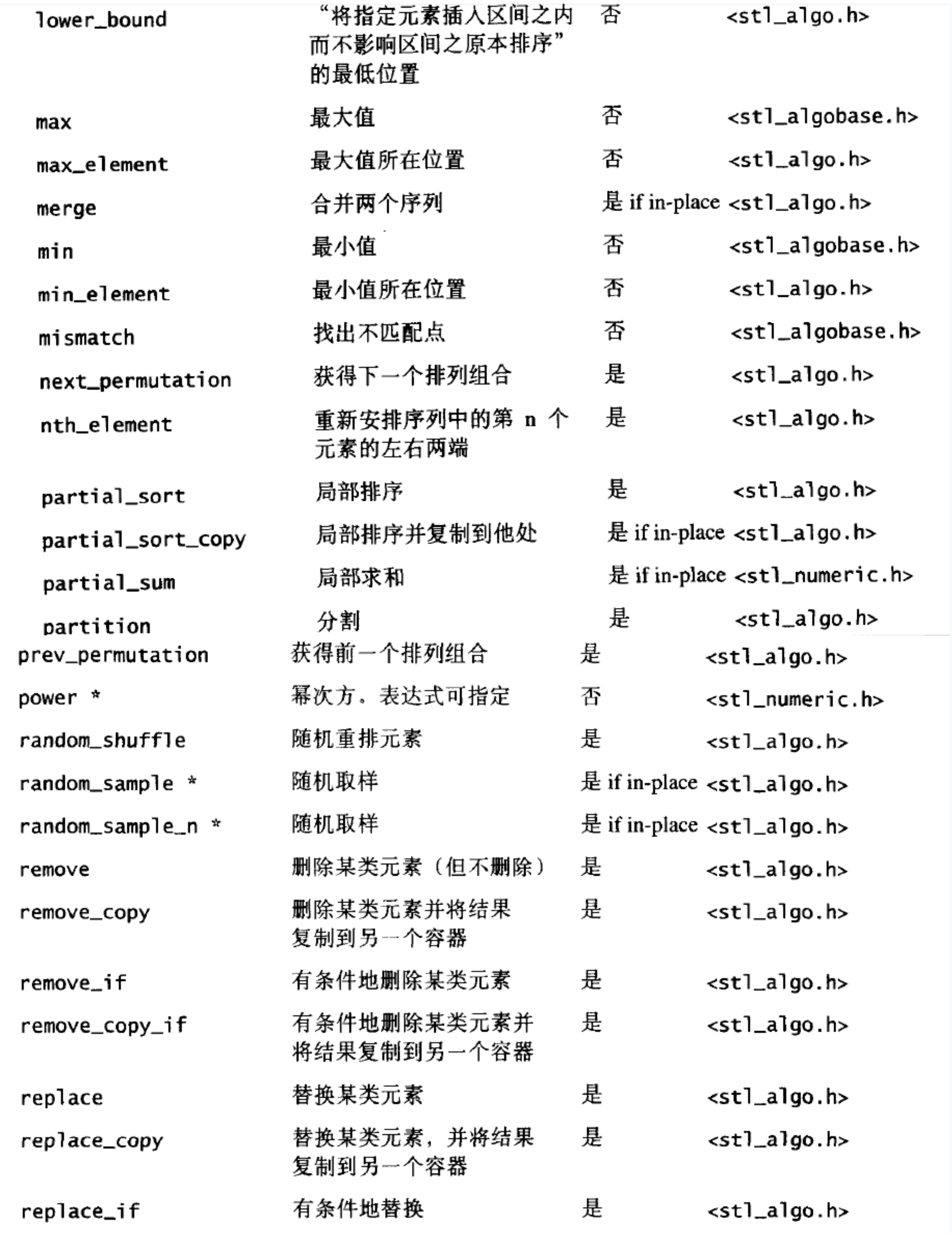
STL所提供的算法往往有两个算法往往有两个版本，其中一个版本表现出最常用的某种运算，第二个给出的是泛华版本，对用用户的数据范围，算法不支持特定的操作，此时就需要用户通过仿函数对象来定制其所需要的操作， 具体见下节算法。

算法

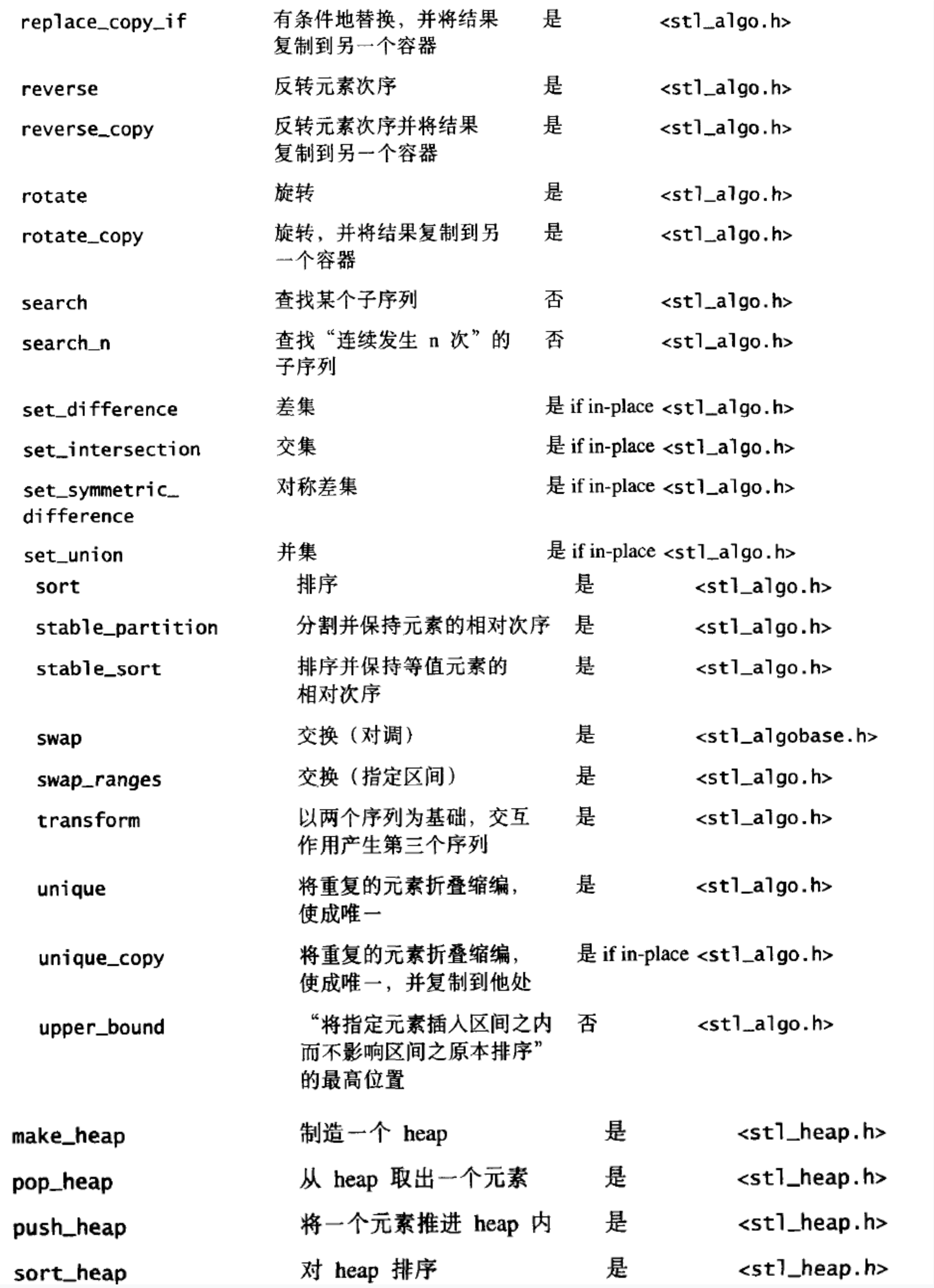
算法：问题之解法也，以有限的步骤，解决数学或逻辑中的问题。
STL中的算法是将常用的算法规范出来，算法只关心操作的步骤，与数据的结构没有任何的关系，而且STL在设计时就有一个目标，就是算法可复用，效率要尽可能的高。STL中收录了极具复用价值的70多个算法，包括：排序，查找，排列组合，数据移动，拷贝，删除，比较组合，运算等。

STL算法总览

算法名称	算法用途	质变?	所在文件
accumulate	元素累计	否	<stl_numeric.h>
adjacent_difference	相邻元素的差额	是 if in-place	<stl_numeric.h>
adjacent_find	查找相邻而重复（或符合某条件）的元素	否	<stl_algo.h>
binary_search	二分查找	否	<stl_algo.h>
Copy	复制	是 if in-place	<stl_algobase.h>
Copy_backward	逆向复制	是 if in-place	<stl_algobase.h>
Copy_n *	复制 n 个元素	是 if in-place	<stl_algobase.h>
count	计数	否	<stl_algo.h>
count_if	在特定条件下计数	否	<stl_algo.h>
equal	判断两个区间相等与否	否	<stl_algobase.h>
equal_range	试图在有序区间中寻找某值（返回一个上下限区间）	否	<stl_algo.h>
fill	改填元素值	是	<stl_algobase.h>
fill_n	改填元素值，n 次	是	<stl_algobase.h>
find	循序查找	否	<stl_algo.h>
find_if	循序查找符合特定条件者	否	<stl_algo.h>
find_end	查找某个子序列的最后一次出现点	否	<stl_algo.h>
find_first_of	查找某些元素的首次出现点	否	<stl_algo.h>
for_each	对区间内的每一个元素施行某操作	否	<stl_algo.h>
generate	以特定操作之运算结果填充特定区间内的元素	是	<stl_algo.h>
generate_n	以特定操作之运算结果填充 n 个元素内容	是	<stl_algo.h>
includes	是否涵盖于某序列之中	否	<stl_algo.h>
inner_product	内积	否	<stl_numeric.h>
inplace_merge	合并并就地替换（覆写上去）	是	<stl_algo.h>
Iota *	在某区间填入某指定值的递增序列	是	<stl_numeric.h>
is_heap *	判断某区间是否为一个 heap	否	<stl_algo.h>
is_sorted *	判断某区间是否已排序	否	<stl_algo.h>
iter_swap	元素互换	是	<stl_algobase.h>
lexicographical_compare	以字典顺序进行比较	否	<stl_numeric.h>



lower_bound	“将指定元素插入区间之内而不影响区间之原本排序”的最低位置	否	<stl_algo.h>
max	最大值	否	<stl_algobase.h>
max_element	最大值所在位置	否	<stl_algo.h>
merge	合并两个序列	是 if in-place	<stl_algo.h>
min	最小值	否	<stl_algobase.h>
min_element	最小值所在位置	否	<stl_algo.h>
mismatch	找出不匹配点	否	<stl_algobase.h>
next_permutation	获得下一个排列组合	是	<stl_algo.h>
nth_element	重新安排序列中的第 n 个元素的左右两端	是	<stl_algo.h>
partial_sort	局部排序	是	<stl_algo.h>
partial_sort_copy	局部排序并复制到他处	是 if in-place	<stl_algo.h>
partial_sum	局部求和	是 if in-place	<stl_numeric.h>
partition	分割	是	<stl_algo.h>
prev_permutation	获得前一个排列组合	是	<stl_algo.h>
power *	幂次方。表达式可指定	否	<stl_numeric.h>
random_shuffle	随机重排元素	是	<stl_algo.h>
random_sample *	随机取样	是 if in-place	<stl_algo.h>
random_sample_n *	随机取样	是 if in-place	<stl_algo.h>
remove	删除某类元素（但不删除）	是	<stl_algo.h>
remove_copy	删除某类元素并将结果复制到另一个容器	是	<stl_algo.h>
remove_if	有条件地删除某类元素	是	<stl_algo.h>
remove_copy_if	有条件地删除某类元素并将结果复制到另一个容器	是	<stl_algo.h>
replace	替换某类元素	是	<stl_algo.h>
replace_copy	替换某类元素，并将结果复制到另一个容器	是	<stl_algo.h>
replace_if	有条件地替换	是	<stl_algo.h>



replace_copy_if	有条件地替换，并将结果复制到另一个容器	是	<stl_algo.h>
reverse	反转元素次序	是	<stl_algo.h>
reverse_copy	反转元素次序并将结果复制到另一个容器	是	<stl_algo.h>
rotate	旋转	是	<stl_algo.h>
rotate_copy	旋转，并将结果复制到另一个容器	是	<stl_algo.h>
search	查找某个子序列	否	<stl_algo.h>
search_n	查找“连续发生 n 次”的子序列	否	<stl_algo.h>
set_difference	差集	是 if in-place	<stl_algo.h>
set_intersection	交集	是 if in-place	<stl_algo.h>
set_symmetric_difference	对称差集	是 if in-place	<stl_algo.h>
set_union	并集	是 if in-place	<stl_algo.h>
sort	排序	是	<stl_algo.h>
stable_partition	分割并保持元素的相对次序	是	<stl_algo.h>
stable_sort	排序并保持等值元素的相对次序	是	<stl_algo.h>
swap	交换（对调）	是	<stl_algobase.h>
swap_ranges	交换（指定区间）	是	<stl_algo.h>
transform	以两个序列为基础，交互作用产生第三个序列	是	<stl_algo.h>
unique	将重复的元素折叠缩编，使成唯一	是	<stl_algo.h>
unique_copy	将重复的元素折叠缩编，使成唯一，并复制到他处	是 if in-place	<stl_algo.h>
upper_bound	“将指定元素插入区间之内而不影响区间之原本排序”的最高位置	否	<stl_algo.h>
make_heap	制造一个 heap	是	<stl_heap.h>
pop_heap	从 heap 取出一个元素	是	<stl_heap.h>
push_heap	将一个元素推进 heap 内	是	<stl_heap.h>
sort_heap	对 heap 排序	是	<stl_heap.h>

STL中的算法都作用在由【fist, last) 指定的前闭后开的区间上

```
template<class InputIterator, class T>
 InputIterator find ( InputIterator first, InputIterator last, const T& value )
 {
   for ( ;first!=last; first++) if ( *first==value ) break;
   return first;
}
```

许多STL算法支持两个版本：一个处理缺省的运算行为；一个接收外部传入的仿函数进行特定的操作

```
void Testaccumulate()
{
  int array[] = {1,2,3,4,5};
  size_t size = sizeof(array)/sizeof(array[0]);

  //将数组中的所有值累加到第三个参数上
  cout<<accumulate(array, array+size, 10)<<endl;

  // 将数组中的值按照所提供的函数指针的方式累加的第三个参数上
  // 其中函数指针的第二个参数为数组中的元素
  cout<<accumulate(array, array+size, 0, myAdd)<<endl;

  // 将数组中的值按照所提供的仿函数对象的方式累加的第三个参数上
  cout<<accumulate(array, array+size, 0, MyAdd())<<endl;
}
```

STL常见算法举例：

```
// 查找类算法
// find/find_if/find_first_of/binary_search/count/count_if

// 排序和通用算法
// sort/stable_sort/partial_sort/partial_sum/partition/merge/reverse
// 删除和替换算法
// copy/copy_backward/remove/remove_if/swap/unique
// 排列组合算法
// prev_permutation/next_permutation
// 集合操作
//set_difference/set_union/set_intersection
```

chou/article/details/53204970” >https://blog.csdn.net/robinchou/article/details/53204970