

C++_STL 总结

郑华

2018 年 2 月 6 日

目录

第一章 容器-Container	7
1.1 容器底层数据结构实现	7
1.1.1 总结	7
1.1.2 设计的数据结构精要	8
1.2 Tree	9
1.2.1 二叉树前序、中序、后序遍历相互求法	9
1.2.2 平衡二叉树	11
1.2.3 Trie 字典树	20
1.3 priority_queue<int, vector<int>, greater<int> >	22
1.3.1 特点	22
1.3.2 用途	22
1.4 Vector	23
1.4.1 关于内存泄露的检验	23
1.4.2 细节	24
1.5 List	26
1.5.1 特点	26
1.5.2 用途	26
1.6 Set	27
1.6.1 排序	27
1.6.2 特点	27
1.6.3 用途	27
1.7 Map	28
1.7.1 特点	28
1.7.2 使用	28
1.8 Hash 序列-> unordered_map or set	30
1.8.1 特点	30
1.8.2 Hash tables	30
1.8.3 实现	31
1.8.4 用法	32

第二章 迭代器-Iterator	33
2.1 性质	33
2.2 使用方法	33
2.3 类别	34
2.4 删除	34
2.5 Iterator 失效概要	35
2.6 Insertion 失效总结	36
2.6.1 Sequence containers	36
2.6.2 Associative containers	36
2.6.3 Unsorted associative containers	37
2.6.4 Container adaptors	37
2.7 Eraser 失效总结	37
2.7.1 Sequence containers	37
2.7.2 Associative containers	37
2.7.3 Unsorted associative containers	37
2.7.4 Container adaptors	38
2.8 Resizing 失效总结	38
2.8.1 Sequence containers	38
2.8.2 Associative containers	38
2.8.3 Unsorted associative containers	38
2.8.4 Container adaptors	38
2.9 Note	39
2.10 插入迭代器	39
2.11 输入输出迭代器	41
2.12 参考	42
第三章 仿函数-Function	43
3.1 std::function	43
3.2 std::bind	44
第四章 算法-Algorithm	49
4.1 查找算法	50
4.2 排序和通用算法	51
4.3 删除和替换算法	52
4.4 排列组合算法	52
4.5 生成和异变算法	53
4.6 关系算法	53
4.7 集合算法	54
4.8 堆算法	54
4.9 算术算法	54

第一章 容器-Container

1.1 容器底层数据结构实现

1.1.1 总结

0.tuple tuple 元组是一个固定大小的不同类型值的集合，可以用来代替简单的结构体

```
#include <tuple>
#include <string>
#include <iostream>
using namespace std;
int main(int arc,char** argv)
{
    using MyTuple = tuple < int, string > ;

    tuple < int, string > tup1 = make_tuple(1, "jack"); // 创建一个MyTuple类型的元组
    MyTuple tup2 = make_tuple(2, "lily"); // 创建一个MyTuple类型的元组

    int var = 3;
    auto tup3 = tie(var,tup1,tup2);    // 创建一个tuple < int, MyTuple, MyTuple>类型的元组

    cout << get<0>(tup1) << endl;    // 输出 1
    cout << get<1>(tup2) << endl;    // 输出 lily
    cout << get<1>(get<1>(tup3)) << endl; // 输出 jack

    return 0;
}
```

1.vector 底层数据结构为数组，支持快速随机访问

array 专门的数组.. 固定大小

2.list 底层数据结构为双向链表，支持快速增删

forward_list 单向链表..

3.deque 底层数据结构为一个中央控制器和多个缓冲区，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问

deque 是一个双端队列 (double-ended queue)，也是在堆中保存内容的。它的保存形式如下：
[堆 1] -> [堆 2] -> [堆 3] -> ... 每个堆保存好几个元素，然后堆和堆之间有指针指向，看起来像是 list 和 vector 的结合品。

4.stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

5.queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

（stack 和 queue 其实是适配器，而不叫容器，因为是对容器的再封装）

6.priority_queue 的底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现

7.set 底层数据结构为红黑树，有序，不重复

8.multiset 底层数据结构为红黑树，有序，可重复

9.map 底层数据结构为红黑树，有序，不重复

10.multimap 底层数据结构为红黑树，有序，可重复

11.unordered_set 底层数据结构为 hash 表，无序，不重复

12.unordered_multiset 底层数据结构为 hash 表，无序，可重复

13.unordered_map 底层数据结构为 hash 表，无序，不重复

14.unordered_multimap 底层数据结构为 hash 表，无序，可重复

1.1.2 设计的数据结构精要

红黑树

参考文献 <http://blog.csdn.net/chenhua jie123/article/details/11951777>

1.2 Tree

1.2.1 二叉树前序、中序、后序遍历相互求法

首先，我们看看前序、中序、后序遍历的特性：

前序遍历：

1. 访问根节点
2. 前序遍历左子树
3. 前序遍历右子树

中序遍历：

1. 前序遍历左子树
2. 访问根节点
3. 前序遍历右子树

后序遍历：

1. 前序遍历左子树
2. 前序遍历右子树
3. 访问根节点

一、已知前序、中序遍历，求后序遍历：

前序遍历: GDAFEMHZ

中序遍历: ADEFGHMZ

1. 根据前序遍历的特点，我们知道根结点为 G 根据前序遍历的特点，我们知道根结点为 G
2. 观察中序遍历 ADEFGHMZ。其中 root 节点 G 左侧的 ADEF 必然是 root 的左子树，G 右侧的 HMZ 必然是 root 的右子树
3. 观察左子树 ADEF，左子树的中的根节点必然是大树的 root 的 leftchild。在前序遍历中，大树的 root 的 leftchild 位于 root 之后，所以左子树的根节点为 D
4. 同样的道理，root 的右子树节点 HMZ 中的根节点也可以通过前序遍历求得。在前序遍历中，一定是先把 root 和 root 的所有左子树节点遍历完之后才会遍历右子树，并且遍历的左子树的第一个节点就是左子树的根节点。同理，遍历的右子树的第一个节点就是右子树的根节点。
5. 观察发现，上面的过程是递归的

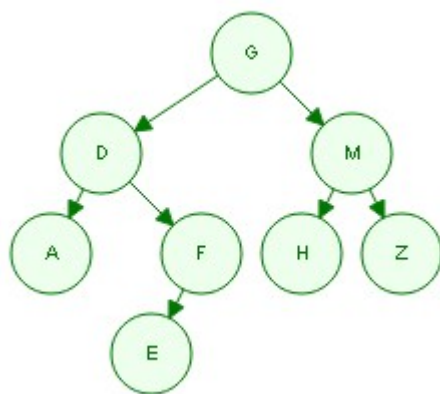


Fig 1.1: 根据先序和中序确定二叉树

二、已知中序和后序遍历，求前序遍历：

1. 根据后序遍历的特点，我们知道后序遍历最后一个结点即为根结点，即根结点为 G。
2. 观察中序遍历 ADEFGHMZ。其中 root 节点 G 左侧的 ADEF 必然是 root 的左子树，G 右侧的 HMZ 必然是 root 的右子树。
3. 观察左子树 ADEF，左子树中的根节点必然是大树的 root 的 leftchild。在前序遍历中，大树的 root 的 leftchild 位于 root 之后，所以左子树的根节点为 D。
4. 同样的道理，root 的右子树节点 HMZ 中的根节点也可以通过前序遍历求得。在前后序遍历中，一定是先把 root 和 root 的所有左子树节点遍历完之后才会遍历右子树，并且遍历的左子树的第一个节点就是左子树的根节点。同理，遍历的右子树的第一个节点就是右子树的根节点。
5. 观察发现，上面的过程是递归的

三、总结 根据上述思路，则大体思想如下：

1. 确定根, 确定左子树，确定右子树。
2. 在左子树中递归。
3. 在右子树中递归。
4. 打印当前根。

结果如图1.1所示：

四、代码实现思路

1.2.2 平衡二叉树

我们知道，对于一般的二叉搜索树（Binary Search Tree），其期望高度（即为一棵平衡树时）为 $\log_2 n$ ，其各操作的时间复杂度 $O(\log_2 n)$ 同时也由此而决定。但是，在某些极端的情况下（如在插入的序列是有序的时），二叉搜索树将退化成近似链或链，此时，其操作的时间复杂度将退化成线性的，即 $O(n)$ 。我们可以通过随机化建立二叉搜索树来尽量地避免这种情况，但是在进行了多次的操作之后，由于在删除时，我们总是选择将待删除节点的后继代替它本身，这样就会造成总是右边的节点数目减少，以至于树向左偏沉。这同时也会造成树的平衡性受到破坏，提高它的操作的时间复杂度。于是就有了我们下边介绍的平衡二叉树。

平衡二叉树定义：平衡二叉树（Balanced Binary Tree）又被称为 AVL 树（有别于 AVL 算法），且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。平衡二叉树的常用算法有红黑树、AVL 树等。在平衡二叉搜索树中，我们可以看到，其高度一般都良好地维持在 $O(\log_2 n)$ ，大大降低了操作的时间复杂度。

最小二叉平衡树的节点的公式如下：

$$F(n)=F(n-1)+F(n-2)+1$$

这个类似于一个递归的数列，可以参考 Fibonacci 数列，1 是根节点， $F(n-1)$ 是左子树的节点数量， $F(n-2)$ 是右子树的节点数量。

```
// 求树高度
int high(btnode *T)
{
    if(T==NULL)
        return 0;
    else
        return max(high(T->lchild),high(T->rchild))+1;
}

// 方法2
void high(btnode *T,int &h) 引用h为树的高度
{
    if(T==NULL)
        h=0;
    else
    {
        int h1,h2;
        high(T->lchild,h1);
        high(T->rchild,h2);
        h=max(h1,h2)+1;
    }
}
```

平衡查找树之 AVL 树

AVL 树定义：AVL 树是最先发明的自平衡二叉查找树。在 AVL 中任何节点的两个儿子子树的高度最大差别为 1，所以它也被称为高度平衡树， n 个结点的 AVL 树最大深度约 $1.44\log_2 n$ 。查找、插入和删除在平均和最坏情况下都是 $O(\log n)$ 。增加和删除可能通过一次或多次树旋转来重新平衡这个树。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在 $O(\log N)$ 。但是频繁旋转会使插入和删除牺牲掉 $O(\log N)$ 左右的时间，不过相对二叉查找树来说，时间上稳定了很多。

AVL 树的自平衡操作——旋转：

AVL 树最关键的也是最难的一步操作就是旋转。旋转主要是为了实现 AVL 树在实施了插入和删除操作以后，树重新回到平衡的方法。下面我们重点研究一下 AVL 树的旋转。

对于一个平衡的节点，由于任意节点最多有两个儿子，因此高度不平衡时，此节点的两颗子树的高度差 2。容易看出，这种不平衡出现在下面四种情况：

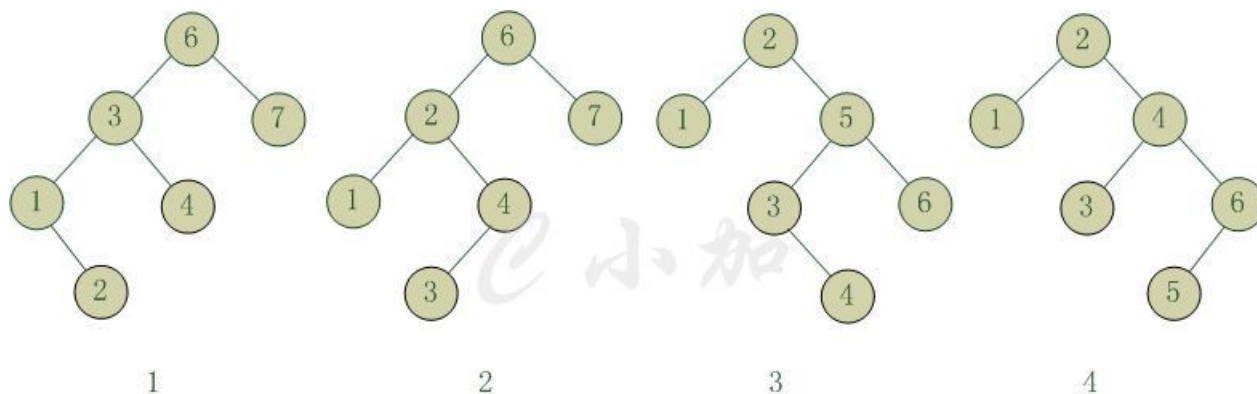


图2 四种不平衡的情况

Fig 1.2: AVL 4 种不平衡

- (1) 6 节点的左子树 3 节点高度比右子树 7 节点大 2，左子树 3 节点的左子树 1 节点高度大于右子树 4 节点，这种情况成为左左。
- (2) 6 节点的左子树 2 节点高度比右子树 7 节点大 2，左子树 2 节点的左子树 1 节点高度小于右子树 4 节点，这种情况成为左右。
- (3) 2 节点的左子树 1 节点高度比右子树 5 节点小 2，右子树 5 节点的左子树 3 节点高度大于右子树 6 节点，这种情况成为右左。
- (4) 2 节点的左子树 1 节点高度比右子树 4 节点小 2，右子树 4 节点的左子树 3 节点高度小于右子树 6 节点，这种情况成为右右。

从图1.2中可以看出，1 和 4 两种情况是对称的，这两种情况的旋转算法是一致的，只需要经过一次旋转就可以达到目标，我们称之为单旋转。2 和 3 两种情况也是对称的，这两种情况的旋转算法也是一致的，需要进行两次旋转，我们称之为双旋转。

单旋转：

单旋转是针对于左左和右右这两种情况的解决方案，这两种情况是对称的，只要解决了左左这种情况，右右就很好办了。图 3 是左左情况的解决方案，节点 k2 不满足平衡特性，因为它的左子树 k1 比右子树 Z 深 2 层，而且 k1 子树中，更深的一层的是 k1 的左子树 X 子树，所以属于左左情况。

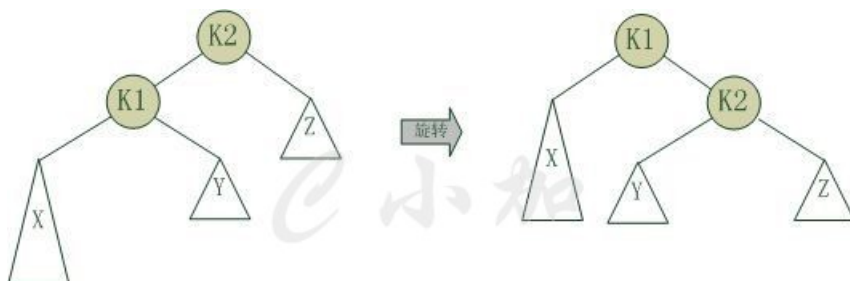


图3 左左情况下单旋转的过程

Fig 1.3: AVL 单旋转

为使树恢复平衡，我们把 k2 变成这棵树的根节点，因为 k2 大于 k1，把 k2 置于 k1 的右子树上，而原本在 k1 右子树的 Y 大于 k1，小于 k2，就把 Y 置于 k2 的左子树上，这样既满足了二叉查找树的性质，又满足了平衡二叉树的性质。

这样的操作只需要一部分指针改变，结果我们得到另外一颗二叉查找树，它是一颗 AVL 树，因为 X 向上一移动了一层，Y 还停留在原来的层面上，Z 向下移动了一层。整棵树的新高度和之前没有在左子树上插入的高度相同，插入操作使得 X 高度长高了。因此，由于这颗子树高度没有变化，所以通往根节点的路径就不需要继续旋转了。

双旋转：

对于左右和右左这两种情况，单旋转不能使它达到一个平衡状态，要经过两次旋转。双旋转是针对于这两种情况的解决方案，同样的，这样两种情况也是对称的，只要解决了左右这种情况，右左就很好办了。图 4 是左右情况的解决方案，节点 k3 不满足平衡特性，因为它的左子树 k1 比右子树 Z 深 2 层，而且 k1 子树中，更深的一层的是 k1 的右子树 k2 子树，所以属于左右情况。

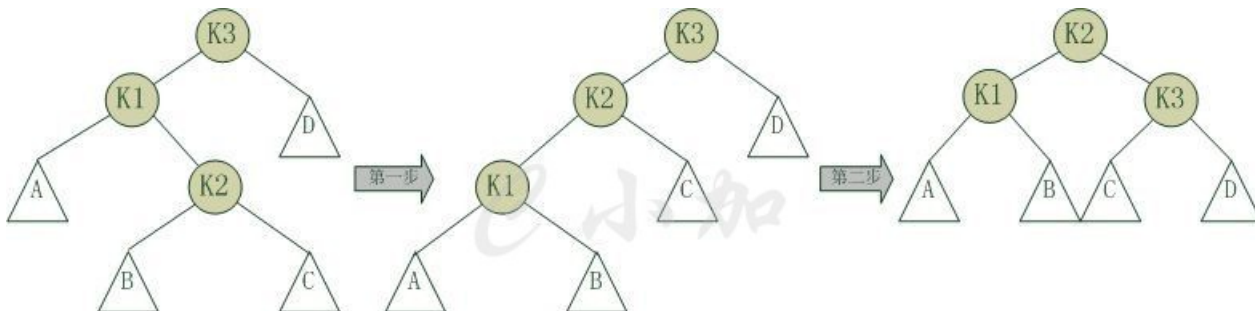


图4 左右情况下双旋转的过程

Fig 1.4: AVL 双旋转

为使树恢复平衡，我们需要进行两步，第一步，把 k1 作为根，进行一次右右旋转，旋转之后就变成了左左情况，所以第二步再进行一次左左旋转，最后得到了一棵以 k2 为根的平衡二叉树。

实现：

```
//AVL树节点信息
template<class T>
class TreeNode
{
public:
    TreeNode():lson(NULL),rson(NULL),freq(1),hgt(0){}
    T data;//值
    int hgt;//高度
    unsigned int freq;//频率
    TreeNode* lson;//指向左儿子的地址
    TreeNode* rson;//指向右儿子的地址
};

//AVL树类的属性和方法声明
template<class T>
class AVLTree
{
private:
    TreeNode<T>* root;//根节点
    void insertpri(TreeNode<T>* &node,T x);//插入
    TreeNode<T>* findpri(TreeNode<T>* node,T x);//查找
    void insubtree(TreeNode<T>* node);//中序遍历
    void Deletepri(TreeNode<T>* &node,T x);//删除
    int height(TreeNode<T>* node);//求树的高度
    void SingRotateLeft(TreeNode<T>* &k2);//左左情况下的旋转
    void SingRotateRight(TreeNode<T>* &k2);//右右情况下的旋转
    void DoubleRotateLR(TreeNode<T>* &k3);//左右情况下的旋转
    void DoubleRotateRL(TreeNode<T>* &k3);//右左情况下的旋转
    int Max(int cmpa,int cmpb);//求最大值

public:
    AVLTree():root(NULL){}
    void insert(T x);//插入接口
    TreeNode<T>* find(T x);//查找接口
    void Delete(T x);//删除接口
    void traversal();//遍历接口
};

//计算节点的高度
template<class T>
int AVLTree<T>::height(TreeNode<T>* node)
{
```

```

        if(node!=NULL)
            return node->hgt;
        return -1;
    }
    //求最大值
    template<class T>
    int AVLTree<T>::Max(int cmpa,int cmpb)
    {
        return cmpa>cmpb?cmpa:cmpb;
    }
    //左左情况下的旋转
    template<class T>
    void AVLTree<T>::SingRotateLeft(TreeNode<T>* &k2)
    {
        TreeNode<T>* k1;
        k1=k2->lson;
        k2->lson=k1->rson;
        k1->rson=k2;

        k2->hgt=Max(height(k2->lson),height(k2->rson))+1;
        k1->hgt=Max(height(k1->lson),k2->hgt)+1;
    }
    //右右情况下的旋转
    template<class T>
    void AVLTree<T>::SingRotateRight(TreeNode<T>* &k2)
    {
        TreeNode<T>* k1;
        k1=k2->rson;
        k2->rson=k1->lson;
        k1->lson=k2;

        k2->hgt=Max(height(k2->lson),height(k2->rson))+1;
        k1->hgt=Max(height(k1->rson),k2->hgt)+1;
    }
    //左右情况的旋转
    template<class T>
    void AVLTree<T>::DoubleRotateLR(TreeNode<T>* &k3)
    {
        SingRotateRight(k3->lson);
        SingRotateLeft(k3);
    }
    //右左情况的旋转
    template<class T>
    void AVLTree<T>::DoubleRotateRL(TreeNode<T>* &k3)
    {
        SingRotateLeft(k3->rson);
        SingRotateRight(k3);
    }

```

```

}
//插入
template<class T>
void AVLTree<T>::insertpri(TreeNode<T>* &node,T x)
{
    if(node==NULL)//如果节点为空,就在此节点处加入x信息
    {
        node=new TreeNode<T>();
        node->data=x;
        return;
    }
    if(node->data>x)//如果x小于节点的值,就继续在节点的左子树中插入x
    {
        insertpri(node->lson,x);
        if(2==height(node->lson)-height(node->rson))
            if(x<node->lson->data)
                SingRotateLeft(node);
            else
                DoubleRotateLR(node);
    }
    else if(node->data<x)//如果x大于节点的值,就继续在节点的右子树中插入x
    {
        insertpri(node->rson,x);
        if(2==height(node->rson)-height(node->lson))//如果高度之差为2的话就失去了平衡,需要旋
            转
            if(x>node->rson->data)
                SingRotateRight(node);
            else
                DoubleRotateRL(node);
    }
    else ++(node->freq);//如果相等,就把频率加1
    node->hgt=Max(height(node->lson),height(node->rson));
}
//插入接口
template<class T>
void AVLTree<T>::insert(T x)
{
    insertpri(root,x);
}
//查找
template<class T>
TreeNode<T>* AVLTree<T>::findpri(TreeNode<T>* node,T x)
{
    if(node==NULL)//如果节点为空说明没找到,返回NULL
    {
        return NULL;
    }
}

```



```

    if(node->data>x)//如果x小于节点的值,就继续在节点的左子树中查找x
    {
        return findpri(node->lson,x);
    }
    else if(node->data<x)//如果x大于节点的值,就继续在节点的右子树中查找x
    {
        return findpri(node->rson,x);
    }
    else return node;//如果相等,就找到了此节点
}
//查找接口
template<class T>
TreeNode<T>* AVLTree<T>::find(T x)
{
    return findpri(root,x);
}
//删除
template<class T>
void AVLTree<T>::Deletepri(TreeNode<T>* &node,T x)
{
    if(node==NULL) return ;//没有找到值是x的节点
    if(x < node->data)
    {
        Deletepri(node->lson,x);//如果x小于节点的值,就继续在节点的左子树中删除x
        if(2==height(node->rson)-height(node->lson))
            if(node->rson->lson!=NULL&&(height(node->rson->lson)>height(node->rson->rson))
                )
                DoubleRotateRL(node);
            else
                SingRotateRight(node);
    }

    else if(x > node->data)
    {
        Deletepri(node->rson,x);//如果x大于节点的值,就继续在节点的右子树中删除x
        if(2==height(node->lson)-height(node->rson))
            if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson)
                ))
                DoubleRotateLR(node);
            else
                SingRotateLeft(node);
    }

    else//如果相等,此节点就是要删除的节点
    {
        if(node->lson&&node->rson)//此节点有两个儿子
        {

```

```

        TreeNode<T>* temp=node->rson;//temp指向节点的右儿子
        while(temp->lson!=NULL) temp=temp->lson;//找到右子树中值最小的节点
        //把右子树中最小节点的值赋值给本节点
        node->data=temp->data;
        node->freq=temp->freq;
        Deletepri(node->rson,temp->data);//删除右子树中最小值的节点
        if(2==height(node->lson)-height(node->rson))
        {
            if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->
                lson) ))
                DoubleRotateLR(node);
            else
                SingRotateLeft(node);
        }
    }
    else//此节点有1个或0个儿子
    {
        TreeNode<T>* temp=node;
        if(node->lson==NULL)//有右儿子或者没有儿子
            node=node->rson;
        else if(node->rson==NULL)//有左儿子
            node=node->lson;
        delete(temp);
        temp=NULL;
    }
}

if(node==NULL) return;
node->hgt=Max(height(node->lson),height(node->rson))+1;
return;
}
//删除接口
template<class T>
void AVLTree<T>::Delete(T x)
{
    Deletepri(root,x);
}
//中序遍历函数
template<class T>
void AVLTree<T>::insubtree(TreeNode<T>* node)
{
    if(node==NULL) return;
    insubtree(node->lson);//先遍历左子树
    cout<<node->data<<" ";//输出根节点
    insubtree(node->rson);//再遍历右子树
}
//中序遍历接口

```

```
template<class T>
void AVLTree<T>::traversal()
{
    insubtree(root);
}
```

平衡查找树之红黑树

红黑树的定义：红黑树是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它是在 1972 年由鲁道夫·贝尔发明的，称之为“对称二叉 B 树”，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的一篇论文中获得的。它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的：它可以在 $O(\log n)$ 时间内做查找，插入和删除，这里的 n 是树中元素的数目。

红黑树和 AVL 树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。这不只是使它们在时间敏感的应用如实时应用（real time application）中有价值，而且使它们有在提供最坏情况担保的其他数据结构中作为建造板块的价值；例如，在计算几何中使用的很多数据结构都可以基于红黑树。此外，红黑树还是 2-3-4 树的一种等同，它们的思想是一样的，只不过红黑树是 2-3-4 树用二叉树的形式表示的。

红黑树的性质：

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制的一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

性质 1. 节点是红色或黑色。

性质 2. 根是黑色。

性质 3. 所有叶子都是黑色（叶子是 NIL 节点）。

性质 4. 每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）

性质 5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

B 树

B 树也是一种用于查找的平衡树，但是它不是二叉树。

B 树的定义：B 树（B-tree）是一种树状数据结构，能够用来存储排序后的数据。这种数据结构能够让查找数据、循序存取、插入数据及删除的动作，都在对数时间内完成。B 树，概括来说是一个一般化的二叉查找树，可以拥有多于 2 个子节点。与自平衡二叉查找树不同，B-树为系统最优化大块数据的读和写操作。B-tree 算法减少定位记录时所经历的中间过程，从而加快存取速度。这种数据结构常被应用在数据库和文件系统的实作上。

在 B 树中查找给定关键字的方法是，首先把根结点取来，在根结点所包含的关键字 K_1, \dots, K_n 查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查找的关键字在 K_i 与 K_{i+1} 之间， P_i 为指向子树根节点的指针，此时取指针 P_i 所指的结点继续查找，直至找到，或指针 P_i 为空时查找失败。

B 树作为一种多路搜索树（并不是二叉的）：

- 1) 定义任意非叶子结点最多只有 M 个儿子；且 $M > 2$ ；
- 2) 根结点的儿子数为 $[2, M]$ ；
- 3) 除根结点以外的非叶子结点的儿子数为 $[M/2, M]$ ；
- 4) 每个结点存放至少 $M/2 - 1$ （取上整）和至多 $M - 1$ 个关键字；（至少 2 个关键字）
- 5) 非叶子结点的关键字个数 = 指向儿子的指针个数 - 1；
- 6) 非叶子结点的关键字： $K[1], K[2], \dots, K[M - 1]$ ；且 $K[i] < K[i + 1]$ ；
- 7) 非叶子结点的指针： $P[1], P[2], \dots, P[M]$ ；其中 $P[1]$ 指向关键字小于 $K[1]$ 的子树， $P[M]$ 指向关键字大于 $K[M - 1]$ 的子树，其它 $P[i]$ 指向关键字属于 $(K[i - 1], K[i])$ 的子树；
- 8) 所有叶子结点位于同一层；

B+ 树

B+ 树是 B 树的变体，也是一种多路搜索树：

- 1) 其定义基本与 B-树相同，除了：
- 2) 非叶子结点的子树指针与关键字个数相同；
- 3) 非叶子结点的子树指针 $P[i]$ ，指向关键字值属于 $[K[i], K[i + 1])$ 的子树（B-树是开区间）；
- 4) 为所有叶子结点增加一个链指针；
- 5) 所有关键字都在叶子结点出现；

B* 树

B* 树是 B+ 树的变体，在 B+ 树的非根和非叶子结点再增加指向兄弟的指针，将结点的最低利用率从 $1/2$ 提高到 $2/3$ 。

1.2.3 Trie 字典树

Trie 树称为字典树，又称单词查找树，Trie 树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。

Trie 树的三个基本性质：

- 1) 根节点不包含字符，除根节点外每一个节点都只包含一个字符；
- 2) 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串；
- 3) 每个节点的所有子节点包含的字符都不相同。

Trie 树的应用：

- 1) 串的快速检索

给出 N 个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。

在这道题中，我们可以用数组枚举，用哈希，用字典树，先把熟词建一棵树，然后读入文章进行比较，这种方法效率是比较高的。

2) “串” 排序

给定 N 个互不相同的仅由一个单词构成的英文名，让你将他们按字典序从小到大输出。用字典树进行排序，采用数组的方式创建字典树，这棵树的每个结点的所有儿子很显然地按照其字母大小排序。对这棵树进行先序遍历即可。

3) 最长公共前缀

对所有串建立字典树，对于两个串的最长公共前缀的长度即他们所在的结点的公共祖先个数，于是，问题就转化为求公共祖先的问题。

1.3 `priority_queue<int, vector<int>, greater<int> >`

1.3.1 特点

- 允许重复元素
- 一种适配器 [包装了底层容器如 `vector` 的高层抽象]
- 可 `pop()`、`push()`
- `pop()` 的元素永远为当前序列里中最小或最大的。

1.3.2 用途

1.4 Vector

1.4.1 关于内存泄露的检验

随着用 string 越来越多，有的时候你会发现 string 的内存管理的问题，存在内存暂时泄露的问题。这个内存泄露与我们常规说的内存泄露问题不一样。它不是真的内存泄露，在程序结束的时候，内存还是会释放掉的，但是在程序的运行过程中，内存被 string or vector 对象占用着。比如你有个 string 对象保存了一个比较大的网页，用完了之后，你想通过 clear 来回收这个对象的内存，你这样是做不到的。因为 string 的 clear 并不会真正回收 string 对象分配内存。那要怎么做才能做到呢？

vector 与 string 类似，而 string 比较少。

```
vector<int> test;

cout << test.capacity() << endl;
cout << "after_pushOneData_test_capacity_is_" << test.capacity() << endl;

for (int i = 0; i < 100; ++i)
test.push_back(i);

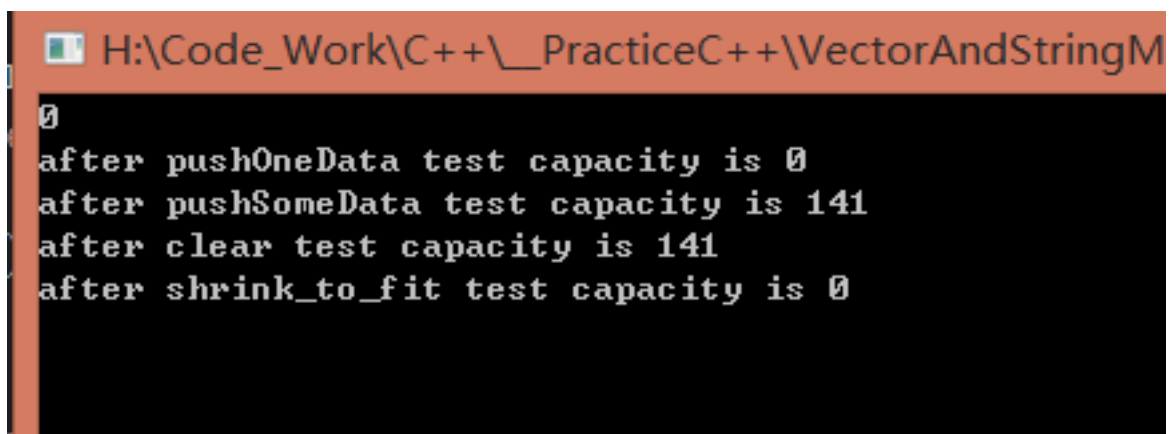
cout << "after_pushSomeData_test_capacity_is_" << test.capacity() << endl;

test.clear();

cout << "after_clear_test_capacity_is_" << test.capacity() << endl;

test.shrink_to_fit();
//test.swap(vector<int>()); 同样可以解决内存泄露问题.. 交换一个空的临时对象.. 这样就可以
    利用对象在离开作用域时析构的特性了

cout << "after_shrink_to_fit_test_capacity_is_" << test.capacity() << endl;
```

A screenshot of a C++ program's output in a terminal window. The window title is "H:\Code_Work\C++_PracticeC++\VectorAndStringM". The output shows four lines of text: "after pushOneData test capacity is 0", "after pushSomeData test capacity is 141", "after clear test capacity is 141", and "after shrink_to_fit test capacity is 0". The text is displayed in a monospaced font on a black background with a red border.

```
H:\Code_Work\C++\_PracticeC++\VectorAndStringM
0
after pushOneData test capacity is 0
after pushSomeData test capacity is 141
after clear test capacity is 141
after shrink_to_fit test capacity is 0
```

Fig 1.5: vector 内存泄露检测

Notice vector 依旧 string clear 后，需要调用.shrink_to_fit 才可将内存清除，防止内存泄露，或者与一个同类型空容器交换以释放内存。

1.4.2 细节

push_back 与 emplace_back

<http://blog.csdn.net/yockie/article/details/52674366>

```
#include <vector>
#include <string>
#include "time_interval.h"

class Foo {
public:
    Foo(std::string str) : name(str) {
        std::cout << "constructor" << std::endl;
    }
    Foo(const Foo& f) : name(f.name) {
        std::cout << "copy_constructor" << std::endl;
    }
    Foo(Foo&& f) : name(std::move(f.name)){
        std::cout << "move_constructor" << std::endl;
    }

private:
    std::string name;
};

int main()
{
    std::vector<Foo> v;
    int count = 10000000;
    v.reserve(count); //预分配十万大小，排除掉分配内存的时间

    {
        TIME_INTERVAL_SCOPE("push_back_T:");
        Foo temp("ceshi");
        v.push_back(temp); // push_back(const T&), 参数是左值引用
        //打印结果:
        //constructor
        //copy constructor
    }

    v.clear();
    {
        TIME_INTERVAL_SCOPE("push_back_move(T):");
        Foo temp("ceshi");
        v.push_back(std::move(temp)); // push_back(T &&), 参数是右值引用
    }
}
```



```

        //打印结果:
        //constructor
        //move constructor
    }

    v.clear();
    {
        TIME_INTERVAL_SCOPE("push_back(T&&):");
        v.push_back(Foo("ceshi")); // push_back(T &&), 参数是右值引用
        //打印结果:
        //constructor
        //move constructor
    }

    v.clear();
    {
        std::string temp = "ceshi";
        TIME_INTERVAL_SCOPE("push_back(string):");
        v.push_back(temp); // push_back(T &&), 参数是右值引用
        //打印结果:
        //constructor
        //move constructor
    }

    v.clear();
    {
        std::string temp = "ceshi";
        TIME_INTERVAL_SCOPE("emplace_back(string):");
        v.emplace_back(temp); // 只有一次构造函数, 不调用拷贝构造函数, 速度最快
        //打印结果:
        //constructor
    }
}

```

1.5 List

1.5.1 特点

- 允许重复元素
- 实现了基本的 `==` 和 `!=` 等 (如果两个链表里的元素都相等返回 `true`)
- 底层数据结构为双向链表
- 可前 (`push_front()`)、后 (`push_back()`)、任意位置插入 (`insert()`)
- 当然就有 `pop` 了

1.5.2 用途

1.6 Set

1.6.1 排序

<http://blog.csdn.net/wangran51/article/details/8836160>

1.6.2 特点

- 无重复元素
- 元素默认升序排序【1,2,3】
- 高效的查找：红黑二叉检索树
- 不可以修改元素
- 快捷的删除

1.6.3 用途

主要用于检索数据

1.7 Map

1.7.1 特点

- 无重复元素
- 高效的查找：红黑二叉检索树，查找和添加的复杂度都为 $O(\log(n))$, `unordered_map` 使用 hash 表作为基本的存储结构, $O(1)$ 。
- 可以修改元素
- 快捷的删除

1.7.2 使用

map 最基本的构造函数

- `map<string , int >mapstring;`
- `map< char ,string>mapchar;`
- `map<int ,char >mapint;`

map 添加数据

- `map<int ,string> maplive;`
1. `maplive.insert(pair<int,string>(102,"active"));`
 2. `maplive.insert(map<int,string>::value_type(321,"hai"));`
 3. `maplive[112]="April";` map 中最简单最常用的插入添加！

map 中元素的查找

- 定义迭代器准备保存查找返回位置:`map<int ,string >::iterator l_it;`
- Find 函数: `l_it=maplive.find(112);`
- 结果判断: `if(l_it==maplive.end()) not find`

map 中元素的删除

先查找，返回迭代器位置，`erase`。

map 的 sort 问题

Map 中的元素是自动按 key 升序排序，所以不能对 map 用 `sort` 函数

其他操作函数

- `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
- `rbegin()` 返回一个指向 `map` 尾部的逆向迭代器
- `size()` 返回 `map` 中元素的个数

1.8 Hash 序列→ unordered_map or set

1.8.1 特点

- 无重复元素
- 高效的查找：查找和添加复杂度均为 $O(1)$
- 可以修改元素
- 快捷的删除

1.8.2 Hash tables

Including hash tables (unordered associative containers) in the C++ standard library is one of the most recurring requests. It was not adopted in C++03 due to time constraints only. Although hash tables are less efficient than a balanced tree in the worst case (in the presence of many collisions), they perform better in many real applications.

Collisions are managed only via linear chaining because the committee didn't consider it to be opportune to standardize solutions of open addressing that introduce quite a lot of intrinsic problems (above all when erasure of elements is admitted). To avoid name clashes with non-standard libraries that developed their own hash table implementations, the prefix “**unordered**” was used *instead of* “**hash**” .

The new library has four types of hash tables, differentiated by whether or not they accept elements with the same key (**unique keys or equivalent keys**), and whether they map each key to an associated value. They correspond to the four existing binary-search-tree-based associative containers, with an `unordered_` prefix.

Type of hash table	Associated values	Equivalent keys
<code>std::unordered_set</code>	No	No
<code>std::unordered_multiset</code>	No	Yes
<code>std::unordered_map</code>	Yes	No
<code>std::unordered_multimap</code>	Yes	Yes

Fig 1.6: HashTable - 4types

```
#include <iostream>
#include <string>
#include <unordered_map>

int main()
```

```

{
    std::unordered_map<std::string, int> months;
    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    months["may"] = 31;
    months["june"] = 30;
    months["july"] = 31;
    months["august"] = 31;
    months["september"] = 30;
    months["october"] = 31;
    months["november"] = 30;
    months["december"] = 31;
    std::cout << "september_>" << months["september"] << std::endl;
    std::cout << "april_>" << months["april"] << std::endl;
    std::cout << "december_>" << months["december"] << std::endl;
    std::cout << "february_>" << months["february"] << std::endl;
    return 0;
}

```

1.8.3 实现

HashMap 基于 hash table（哈希表）。哈希表最大的优点，就是把数据的存储和查找消耗的时间大大降低，几乎可以看成是常数时间；而代价仅仅是消耗比较多的内存。然而在当前可利用内存越来越多的情况下，用空间换时间的做法是值得的

其基本原理是：使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数，也叫做散列函数），使得每个元素的关键字都与一个函数值（即数组下标，hash 值）相对应，于是用这个数组单元来存储这个元素；也可以简单的理解为，按照关键字为每一个元素“分类”，然后将这个元素存储在相应“类”所对应的地方，称为桶。

但是，不能够保证每个元素的关键字与函数值是一一对应的，因此极有可能出现对于不同的元素，却计算出了相同的函数值，这样就产生了“冲突”，换句话说，就是把不同的元素分在了相同的“类”之中。总的来说，“直接定址”与“解决冲突”是哈希表的两大特点

HashMap，首先分配一大片内存，形成许多桶。是利用 hash 函数，对 key 进行映射到不同区域（桶）进行保存。

插入过程是：

1. 得到 key
2. 通过 hash 函数得到 hash 值
3. 得到桶号（一般都为 hash 值对桶数求模）
4. 存放 key 和 value 在桶内

取值过程是：

1. 得到 key
2. 通过 hash 函数得到 hash 值
3. 得到桶号 (一般都为 hash 值对桶数求模)
4. 比较桶的内部元素是否与 key 相等，若都不相等，则没有找到。
5. 取出相等的记录的 value。

1.8.4 用法

与 map 大体一致。

第二章 迭代器-Iterator

2.1 性质

- 它是一个访问容器对象的技术手段，通过迭代器可以访问容器中的对象
- 功能类似于指针 或者 数组下标，通过指针加减与数组下标运算获得下一数据成员
- 在实现上，迭代器可以是指针，但并不必须是指针，也不必总是使用数据对象的地址。只要能够访问到数据对象就可以了。

2.2 使用方法

- 声明迭代器变量
- 使用引领操作符访问迭代器指向的当前目标对象
- 使用递增操作符获得下一对象的访问权
- 若迭代器新值超出容器的元素范围，类似指针值变成 NULL，目标对象不可引用

```
// 声明如下:
vector<T>::iterator it;
list<T>::iterator it;
deque<T>::iterator it;

#include <iostream>
#include <algorithm>
using namespace std;
const int size = 16;
int main()
{
    int a[size];
    for( int i = 0; i < size; ++i ) a[i] = i;
    int key = 7;
    int * ip = find( a, a + size, key );
    if( ip == a + size ) // 不要使用NULL做指针测试，直接使用过尾值 []
        cout << key << " not found." << endl;
    else
```

```

        cout << key << " found." << endl;
    return 0;
}

```

2.3 类别

表 2.1: 迭代器类别

类别	功能与使用
输入	从容器中读取元素
输出	向容器中写入元素
正向	组合输入迭代器和输出迭代器的功能，并保留在容器中的位置
双向	组合正向迭代器和逆向迭代器的功能，支持多遍算法
随机	组合双向迭代器的功能与直接访问容器中任何元素的功能，即可向前向后跳过任意个元素

表 2.2: 不同迭代器对应的可执行操作

类别	可使用的操作
所有	<code>++p, p++</code> [位置自增运算]
输入	<code>*p</code> [作为右值], <code>p = q</code> [将一个迭代器赋给另一个迭代器], <code>p == q</code> [相等比较运算，但不能比较大小]
输出	<code>*p</code> [作为左值], <code>p = q</code> [将一个迭代器赋给另一个迭代器]
正向	提供输入输出迭代器的所有功能
双向	<code>-p, p--</code> [位置自减运算]
随机	<code>p+=i</code> [将迭代器递增 <code>i</code> 位], <code>p+i</code> [在 <code>p</code> 位加 <code>i</code> 位后的迭代器], <code>p[i]</code> [返回 <code>p</code> 位元素偏离 <code>i</code> 位的元素引用], <code>p < p1</code> [如果迭代器 <code>p</code> 的位置在 <code>p1</code> 前, 返回 <code>true</code> , 否则返回 <code>false</code>]

2.4 删除

vector 用到 `erase` 的话, 那必须用迭代器了, 那么以下就要注意了

- 在执行 `erase` 后, 不可以进行 `++pos` 操作, 否则会报错
- 在不执行 `erase` 时, 要进行 `++pos` 进行前进

```

for(auto i = 1; i < nums.size(); )
{
    flag = true;
}

```

表 2.3: 各容器对应的迭代器类别

容器类别	可使用的迭代器
vector	随机
deque	随机
list	双向
set	双向 multi 同
map	双向 multi 同
stack	不支持
queue	不支持
priority_queue	不支持

```

    if(nums[i-1] == nums[i])
    {
        ++size;
        if(size > 2)
        {
            --lenth;
            nums.erase(pos); //这块执行pos 的话就不用再移位置了
            flag = false;
        }
    }
    else
    {
        size = 1;
    }
    if(flag) //否则，移位置前进
    {
        ++i;
        ++pos;
    }
}

```

2.5 Iterator 失效概要

Golden Rule 是：尽量不要使用容器的插入删除操作之前的迭代器。

对于 vector ,deque, list, 一种可行的方式是:

```

std::vector<int>::iterator it = my_container.begin();
for (it != my_container.end(); /**blank*/ ) {
    if (*it % 2 == 1) {
        it = my_container.erase(it);
    }
    else{

```

```

        it++;
    }
}

// Method Two --
std::vector<int>::iterator it = my_container.begin();
for (it != my_container.end(); /**blank*/ ) {
    if (*it % 2 == 1) {
        my_container.erase(it++);
    }
    else{
        it++;
    }
}
/*
    my_container.erase(it++) 巧妙得在执行erase()之前, it 先自增, 指向被删除元素后面的元素, 而给erase()传递的是未自增的it迭代器, 以定位要删除的元素。
    如果元素的值为奇数, 则删除此元素, it指向下一个元素, 如果元素的值为偶数, 则检查下一个元素的值。整个迭代过程中迭代器就不会失效了。
*/

```

2.6 Insertion 失效总结

2.6.1 Sequence containers

- **vector**: all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated)
- **deque**: all iterators and references are invalidated, unless the inserted member is at an end (front or back) of the deque (in which case all iterators are invalidated, but references to elements are unaffected)
- **list**: all iterators and references unaffected
- **forward_list**: all iterators and references unaffected (applies to `insert_after`)
- **array**: (n/a)

2.6.2 Associative containers

- **[multi]{set,map}**: all iterators and references unaffected

2.6.3 Unsorted associative containers

- `unordered_[multi]{set,map}`: all iterators invalidated when rehashing occurs, but references unaffected [23.2.5/8]. Rehashing does not occur if the insertion does not cause the container's size to exceed $z * B$ where z is the maximum load factor and B the current number of buckets.

2.6.4 Container adaptors

- `stack`: inherited from underlying container
- `queue`: inherited from underlying container
- `priority_queue`: inherited from underlying container

2.7 Eraser 失效总结

2.7.1 Sequence containers

- `vector`: every iterator and reference at or after the point of erase is invalidated
- `deque`: erasing the last element invalidates only iterators and references to the erased elements and the past-the-end iterator; erasing the first element invalidates only iterators and references to the erased elements; erasing any other elements invalidates all iterators and references (including the past-the-end iterator)
- `list`: only the iterators and references to the erased element is invalidated
- `forward_list`: only the iterators and references to the erased element is invalidated (applies to `erase_after`)
- `array`: (n/a)

2.7.2 Associative containers

- `[multi]{set,map}`: only iterators and references to the erased elements are invalidated

2.7.3 Unsorted associative containers

- `unordered_[multi]{set,map}`: all iterators invalidated when rehashing occurs, but references unaffected [23.2.5/8]. Rehashing does not occur if the insertion does not cause the container's size to exceed $z * B$ where z is the maximum load factor and B the current number of buckets.

2.7.4 Container adaptors

- `stack`: inherited from underlying container
- `queue`: inherited from underlying container
- `priority_queue`: inherited from underlying container

2.8 Resizing 失效总结

2.8.1 Sequence containers

- `vector`: as per insert/erase
- `deque`: as per insert/erase
- `list`: as per insert/erase
- `forward_list`: as per insert/erase
- `array`: (n/a)

2.8.2 Associative containers

- `[multi]{set,map}`: only iterators and references to the erased elements are invalidated

2.8.3 Unsorted associative containers

- `unordered_[multi]{set,map}`: all iterators invalidated when rehashing occurs, but references unaffected [23.2.5/8]. Rehashing does not occur if the insertion does not cause the container's size to exceed $z * B$ where z is the maximum load factor and B the current number of buckets.

2.8.4 Container adaptors

- `stack`: inherited from underlying container
- `queue`: inherited from underlying container
- `priority_queue`: inherited from underlying container

2.9 Note

- **Unless otherwise specified** (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to **a library function shall not invalidate iterators** to, or change the values of, objects within that container.
- **no swap()** function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [Note: **The end() iterator** does not refer to any element, so it **may be invalidated**. —end note]
- Other than the above caveat regarding swap(), it's not clear whether "end" iterators are subject to the above listed per-container rules; you should assume, anyway, that they are
- **vector** and all unordered associative containers support **reserve(n)** which guarantees that no automatic resizing will occur at least until the size of the container grows to **n**. Caution should be taken with unordered associative containers because a future proposal will allow the specification of a minimum load factor, which would allow rehashing to occur on **insert** after enough **erase** operations reduce the container size below the minimum; the guarantee should be considered potentially void after an **erase**.

2.10 插入迭代器

迭代器是一个纯粹抽象概念：任何东西，只要其行为类似迭代器，它就是一个迭代器。C++ 标准库提供了数个预先定义的特殊迭代器，即迭代器适配器（**iterator adapters**）。它们不仅起辅助作用，还能赋予整个迭代器抽象概念更强大的能力。

说明：适配器是使一事物的行为类似于另一事物的行为的一种机制。

插入器是一种迭代器适配器，带有一个容器参数，并**生成**一个迭代器，用于在指定容器中插入元素。通过插入迭代器赋值时，迭代器将会插入一个新元素。C++ 提供了三种插入器，其差别在于插入元素位置不同。

1. **back_inserter**: 创建使用**push_back** 实现插入的迭代器

back_inserter 内部调用**push_back**，在容器末尾插入元素。因此，只有在提供有**push_back** 成员函数的容器中才能使用。这样的容器有：**vector, deque, list**。元素排列次序和安插次序相同

2. **front_inserter**: 创建使用**push_front** 实现插入的迭代器

front_inserter 内部调用**push_front**，将元素安插于容器最前端。因此，只有在提供有**push_front** 成员函数的容器中才能使用。这样的容器有：**deque, list**。元素排列次序和安插次序相反

3. **insert**: 创建使用**insert** 实现插入的迭代器

Insertter 内部调用insert，在它的迭代器实参所标明的位置前面插入元素。所有STL 容器都提供insert 成员函数，因此这是唯一可用于关联容器上的插入迭代器。元素排列次序和安插次序相同

```
#include <iostream>
#include <vector>
#include <list>
#include <iterator>

using namespace std;

template<typename T>
void PrintElements(T c)
{
    T::const_iterator itr = c.begin();
    while(itr != c.end())
    {
        cout<<*itr++<<" ";
    }
}

int main()
{
    vector<int> vecSrc;
    list<int> vecDest;

    for(vector<int>::size_type i=0; i<3; ++i)
    {
        vecSrc.push_back(i);
    }

    /*
        list<int> coll1;
        vector<int> coll2;
        copy(coll1.begin(), coll1.end(), back_inserter(coll2));
    */
    copy(vecSrc.begin(),vecSrc.end(),back_insert_iterator<list<int> >(vecDest));
    PrintElements(vecDest);
    cout<<endl;

    copy(vecSrc.begin(),vecSrc.end(),front_insert_iterator<list<int> >(vecDest));
    PrintElements(vecDest);
    cout<<endl;

    copy(vecSrc.begin(),vecSrc.end(),insert_iterator<list<int> >(vecDest,++vecDest.
        begin())));
    PrintElements(vecDest);
```



```

        return 0;
    }

    // 2 0 1 - 2 1 0 - 0 1 2

```

2.11 输入输出迭代器

标准程序库定义有供输入及输出用的 `istream_iterator` 类，称为 `istream_iterator` 和 `ostream_iterator`，分别支持单一型别的元素读取和写入

1. `ostream_iterator`:

```

// ostream_iterator example
#include <iostream> // std::cout
#include <iterator> // std::ostream_iterator
#include <vector> // std::vector
#include <algorithm> // std::copy

int main () {
    std::vector<int> myvector;
    for (int i=1; i<10; ++i) myvector.push_back(i*10);

    std::ostream_iterator<int> out_it (std::cout, "\n");
    std::copy ( myvector.begin(), myvector.end(), out_it );
    std::unique_copy(myvector.begin(), myvector.end(), out_it);
    return 0;
}

// 10, 20, 30, 40, 50, 60, 70, 80, 90,

```

2. `istream_iterator`:

```

// istream_iterator example
#include <iostream> // std::cin, std::cout
#include <iterator> // std::istream_iterator

int main () {
    double value1, value2;
    std::cout << "Please, insert two values: ";

    std::istream_iterator<double> eos; // end-of-stream iterator
    std::istream_iterator<double> iit (std::cin); // stdin iterator

    if (iit!=eos) value1=*iit;

```

```
    ++iit;
    if (iit!=eos) value2=*iit;

    std::cout << value1 << "*" << value2 << "=" << (value1*value2) << '\n';

    return 0;
}

// Please, insert two values: 2 32 2*32=64
```

2.12 参考

<http://www.cnblogs.com/dongzhiquan/archive/2011/01/05/1994523.html>

<http://www.cnblogs.com/blueoverflow/p/4923523.html>

失效重要:<http://stackoverflow.com/questions/6438086/iterator-invalidation-rules>

<http://lib.csdn.net/article/cplusplus/28411>

C++ STL 插入器: <http://www.linuxidc.com/Linux/2015-02/113456.htm>

<http://blog.csdn.net/bonchoix/article/details/8062483>

第三章 仿函数-Function

3.1 std::function

在 C++ 中，可调用实体主要包括函数，函数指针，函数引用，可以隐式转换为函数指定的对象，或者实现了 `operator()` 的对象。C++0x 中，新增加了一个 `std::function` 对象，`std::function` 对象是对 C++ 中现有的可调用实体的一种类型安全的包裹（我们知道像函数指针这类可调用实体，是类型不安全的）。`std::function` object 最大的用处就是在实现函数回调，使用者需要注意，它不能被用来检查相等或者不相等

```
#include<iostream>// std::cout
#include<functional>// std::function

void func(void)
{
    std::cout << __FUNCTION__ << std::endl;
}

class Foo
{
public:
    static int foo_func(int a)
    {
        std::cout << __FUNCTION__ << "(" << a << ")_->:_";
        return a;
    }
};

class Bar
{
public:
    int operator() (int a)
    {
        std::cout << __FUNCTION__ << "(" << a << ")_->:_";
        return a;
    }
};

int main()
```

```

{
    // 绑定普通函数
    std::function<void(void)> fr1 = func;
    fr1();

    // 绑定类的静态成员函数
    std::function<int(int)> fr2 = Foo::foo_func;
    std::cout << fr2(100) << std::endl;

    // 绑定仿函数
    Bar bar;
    fr2 = bar;
    std::cout << fr2(200) << std::endl;

    return 0;
}

```

3.2 std::bind

bind 是这样一种机制，它可以预先把指定可调用实体的某些参数绑定到已有的变量，产生一个新的可调用实体，这种机制在回调函数的使用过程中也颇为有用。

C++11 中提供了 std::bind，可以说是一种飞跃的提升，bind 本身是一种延迟计算的思想，它本身可以绑定普通函数、全局函数、静态函数、类静态函数甚至是类成员函数。

std::bind 用来将可调用对象与其参数一起进行绑定。绑定后可以使用 **std::function** 进行保存，并延迟到我们需要的时候调用：

- bind 预先绑定的参数需要传具体的变量或值进去，对于预先绑定的参数，是 pass-by-value 的
- 对于不事先绑定的参数，需要传 std::placeholders 进去，从 _1 开始，依次递增。placeholder 是 pass-by-reference 的
- bind 的返回值是可调用实体，可以直接赋给 std::function 对象
- 对于绑定的指针、引用类型的参数，使用者需要保证在可调用实体调用之前，这些参数是可用的

在绑定部分参数的时候，通过使用 **std::placeholders** 来决定空位参数将会是调用发生时的第几个参数。

```

#include <iostream>
#include <functional>
using namespace std;

int TestFunc(int a, char c, float f)

```

```

{
    cout << a << endl;
    cout << c << endl;
    cout << f << endl;

    return a;
}

int main()
{
    auto bindFunc1 = bind(TestFunc, std::placeholders::_1, 'A', 100.1);
    bindFunc1(10);

    cout << "=====\n";

    auto bindFunc2 = bind(TestFunc, std::placeholders::_2, std::placeholders::_1,
        100.1);
    bindFunc2('B', 10);

    cout << "=====\n";

    auto bindFunc3 = bind(TestFunc, std::placeholders::_2, std::placeholders::_3, std
        ::placeholders::_1);
    bindFunc3(100.1, 30, 'C');

    return 0;
}

\\ Example 2:
int add1(int i, int j, int k) {
    return i + j + k;
}

class Utils {
public:
    Utils(const char* name) {
        strcpy(_name, name);
    }

    void sayHello(const char* name) const {
        std::cout << _name << "say:hello" << name << std::endl;
    }

    static int getId() {
        return 10001;
    }
}

```

```

int operator()(int i, int j, int k) const {
    return i + j + k;
}

private:
char _name[32];
};

int main(void) {

    // 绑定全局函数
    auto add2 = std::bind(add1, std::placeholders::_1, std::placeholders::_2, 10);
    // 函数add2 = 绑定add1函数, 参数1不变, 参数2不变, 参数3固定为10.
    std::cout << typeid(add2).name() << std::endl;
    std::cout << "add2(1,2)_=_ " << add2(1, 2) << std::endl;

    std::cout << "\n-----" << std::endl;

    // 绑定成员函数
    Utils utils("Vicky");
    auto sayHello = std::bind(&Utils::sayHello, utils/*调用者*/, std::placeholders::_1
        /*参数1*/);
    sayHello("Jack");

    auto sayHelloToLucy = std::bind(&Utils::sayHello, utils/*调用者*/, "Lucy"/*固定参数
        1*/);
    sayHelloToLucy();

    // 绑定静态成员函数
    auto getId = std::bind(&Utils::getId);
    std::cout << getId() << std::endl;

    std::cout << "\n-----" << std::endl;

    // 绑定operator函数
    auto add100 = std::bind(&Utils::operator (), utils, std::placeholders::_1, std::
        placeholders::_2, 100);
    std::cout << "add100(1,2)_=_ " << add100(1, 2) << std::endl;

    // 注意: 无法使用std::bind()绑定一个重载函数

    return 0;
}

```

从上面的代码可以看到, bind 能够在绑定时候就同时绑定一部分参数, 未提供的参数则使用占位符表示, 然后在运行时传入实际的参数值。

PS: 绑定的参数将会以值传递的方式传递给具体函数, 占位符将会以引用传递。

参考 : <http://blog.csdn.net/eclipser1987/article/details/24406203>

第四章 算法-Algorithm

算法 (Algorithm)，是用来操作容器中的数据的模板函数。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；

STL 算法部分主要由头文件 `<algorithm>`, `<numeric>`, `<functional>` 组成。要使用 STL 中的算法函数必须包含头文件 `<algorithm>`，对于数值算法须包含 `<numeric>`。

4.1 查找算法

表 4.1: 查找算法

函数名	功能与使用
adjacent_find	在 iterator 对标识元素范围内，查找一对相邻重复元素，找到则返回指向这对元素的第一个元素的 ForwardIterator。否则返回 last。
binary_search	在有序序列中查找 value，找到返回 true。重载的版本实用指定的比较函数对象或函数指针来判断相等
count	利用等于操作符，把标志范围内的元素与输入值比较，返回相等元素个数
count_if	利用输入的操作符，对标志范围内的元素进行操作，返回结果为 true 的个数
equal_range	功能类似 equal，返回一对 iterator，第一个表示 lower_bound，第二个表示 upper_bound
find	利用底层元素的等于操作符，对指定范围内的元素与输入值进行比较。当匹配时，结束搜索，返回该元素的一个 InputIterator
find_end	在指定范围内查找”由输入的另外一对 iterator 标志的第二个序列”的最后一次出现。找到则返回最后一对的第一个 ForwardIterator，否则返回输入的”另外一对”的第一个 ForwardIterator。重载版本使用用户输入的操作符代替等于操作
find_first_of	在指定范围内查找”由输入的另外一对 iterator 标志的第二个序列”中任意一个元素的第一次出现。重载版本中使用了用户自定义操作符
find_if	使用输入的函数代替等于操作符执行 find
lower_bound	返回一个 ForwardIterator，指向在有序序列范围内的可以插入指定值而不破坏容器顺序的第一个位置。重载函数使用自定义比较操作
upper_bound	返回一个 ForwardIterator，指向在有序序列范围内插入 value 而不破坏容器顺序的最后一个位置，该位置标志一个大于 value 的值。重载函数使用自定义比较操作
search	给出两个范围，返回一个 ForwardIterator，查找成功指向第一个范围内第一次出现子序列 (第二个范围) 的位置，查找失败指向 last1。重载版本使用自定义的比较操作
search_n	在指定范围内查找 val 出现 n 次的子序列。重载版本使用自定义的比较操作

4.2 排序和通用算法

表 4.2: 排序和通用算法

函数名	功能与使用
inplace_merge	合并两个有序序列，结果序列覆盖两端范围。重载版本使用输入的操作进行排序
merge	合并两个有序序列，存放到另一个序列。重载版本使用自定义的比较
nth_element	将范围内的序列重新排序，使所有小于第 <i>n</i> 个元素的元素都出现在它前面，而大于它的都出现在后面。重载版本使用自定义的比较操作
partial_sort	对序列做部分排序，被排序元素个数正好可以被放到范围内。重载版本使用自定义的比较操作
partial_sort_copy	与 partial_sort 类似，不过将经过排序的序列复制到另一个容器
partition	对指定范围内元素重新排序，使用输入的函数，把结果为 true 的元素放在结果为 false 的元素之前
random_shuffle	对指定范围内的元素随机调整次序。重载版本输入一个随机数产生操作
reverse	将指定范围内元素重新反序排序
reverse_copy	与 reverse 类似，不过将结果写入另一个容器
rotate	将指定范围内元素移到容器末尾，由 middle 指向的元素成为容器第一个元素
rotate_copy	与 rotate 类似，不过将结果写入另一个容器
sort	以升序重新排列指定范围内的元素。重载版本使用自定义的比较操作
stable_sort	与 sort 类似，不过保留相等元素之间的顺序关系
stable_partition	与 partition 类似，不过不保证保留容器中的相对顺序

4.3 删除和替换算法

表 4.3: 删除和替换算法

函数名	功能与使用
<code>copy</code>	复制序列
<code>copy_backward</code>	与 <code>copy</code> 相同，不过元素是以相反顺序被拷贝
<code>iter_swap</code>	交换两个 Forward Iterator 的值
<code>remove</code>	删除指定范围内所有等于指定元素的元素。注意，该函数不是真正删除函数。内置函数不适合使用 <code>remove</code> 和 <code>remove_if</code> 函数
<code>remove_copy</code>	将所有不匹配元素复制到一个制定容器，返回 OutputIterator 指向被拷贝的末元素的下一个位置
<code>remove_if</code>	删除指定范围内输入操作结果为 <code>true</code> 的所有元素
<code>remove_copy_if</code>	将所有不匹配元素拷贝到一个指定容器
<code>replace</code>	将指定范围内所有等于 <code>vold</code> 的元素都用 <code>vnew</code> 代替
<code>replace_copy</code>	与 <code>replace</code> 类似，不过将结果写入另一个容器
<code>replace_if</code>	将指定范围内所有操作结果为 <code>true</code> 的元素用新值代替
<code>replace_copy_if</code>	与 <code>replace_if</code> ，不过将结果写入另一个容器
<code>swap</code>	交换存储在两个对象中的值
<code>swap_range</code>	将指定范围内的元素与另一个序列元素值进行交换
<code>unique</code>	清除序列中重复元素，和 <code>remove</code> 类似，它也不能真正删除元素。重载版本使用自定义比较操作
<code>unique_copy</code>	与 <code>unique</code> 类似，不过把结果输出到另一个容器

实际上，`remove_if()` 等并没有删除容器中的任何元素，它没有改变 `container.end()`，调用 `remove_if()` 后容器元素个数不会改变!! 删除元素的工作交给 `erase()` 才能真正删除。

4.4 排列组合算法

表 4.4: 排列组合算法

函数名	功能与使用
<code>next_permutation</code>	复制序列
<code>prev_permutation</code>	取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回 <code>false</code> 。重载版本使用自定义的比较操作

4.5 生成和异变算法

表 4.5: 生成和异变算法

函数名	功能与使用
fill	将输入值赋给标志范围内的所有元素
fill_n	将输入值赋给 first 到 first+n 范围内的所有元素
for_each	用指定函数依次对指定范围内所有元素进行迭代访问，返回所指定的函数类型。该函数不得修改序列中的元素
generate	连续调用输入的函数来填充指定的范围
generate_n	与 generate 函数类似，填充从指定 iterator 开始的 n 个元素
transform	将输入的操作作用与指定范围内的每个元素，并产生一个新的序列。重载版本将操作作用在一对元素上，另外一个元素来自输入的另外一个序列。结果输出到指定容器

4.6 关系算法

表 4.6: 关系算法

函数名	功能与使用
equal	如果两个序列在标志范围内元素都相等，返回 true。重载版本使用输入的操作符代替默认的等于操作符
includes	判断第一个指定范围内的所有元素是否都被第二个范围包含，使用底层元素的 < 操作符，成功返回 true。重载版本使用用户输入的函数
lexicographical_compare	比较两个序列。重载版本使用用户自定义比较操作
max	返回两个元素中较大一个。重载版本使用自定义比较操作
max_element	返回一个 Forward Iterator，指出序列中最大的元素。重载版本使用自定义比较操作
min	返回两个元素中较小一个。重载版本使用自定义比较操作
min_element	返回一个 Forward Iterator，指出序列中最小的元素。重载版本使用自定义比较操作
mismatch	并行比较两个序列，指出第一个不匹配的位置，返回一对 iterator，标志第一个不匹配元素位置。如果都匹配，返回每个容器的 last。重载版本使用自定义的比较操作

4.7 集合算法

表 4.7: 集合算法

函数名	功能与使用
<code>set_union</code>	构造一个有序序列，包含两个序列中所有的不重复元素。重载版本使用自定义的比较操作
<code>set_intersection</code>	构造一个有序序列，其中元素在两个序列中都存在。重载版本使用自定义的比较操作
<code>set_difference</code>	构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存在的元素。重载版本使用自定义的比较操作
<code>set_symmetric_difference</code>	构造一个有序序列，该序列取两个序列的对称差集 (并集-交集)

4.8 堆算法

表 4.8: 堆算法

函数名	功能与使用
<code>make_heap</code>	把指定范围内的元素生成一个堆。重载版本使用自定义比较操作
<code>pop_heap</code>	并不真正把最大元素从堆中弹出，而是重新排序堆。它把 <code>first</code> 和 <code>last-1</code> 交换，然后重新生成一个堆。可使用容器的 <code>back</code> 来访问被”弹出”的元素或者使用 <code>pop_back</code> 进行真正的删除。重载版本使用自定义的比较操作
<code>push_heap</code>	假设 <code>first</code> 到 <code>last-1</code> 是一个有效堆，要被加入到堆的元素存放在位置 <code>last-1</code> ，重新生成堆。在指向该函数前，必须先把元素插入容器后。重载版本使用指定的比较操作
<code>sort_heap</code>	对指定范围内的序列重新排序，它假设该序列是个有序堆。重载版本使用自定义比较操作

4.9 算术算法

表 4.9: 算术算法

函数名	功能与使用
<code>accumulate</code>	<code>iterator</code> 对标识的序列段元素之和，加到一个由 <code>val</code> 指定的初始值上。重载版本不再做加法，而是传进来的二元操作符被应用到元素上
<code>partial_sum</code>	创建一个新序列，其中每个元素值代表指定范围内该位置前所有元素之和。重载版本使用自定义操作代替加法
<code>inner_product</code>	对两个序列做内积 (对应元素相乘，再求和) 并将内积加到一个输入的初始值上。重载版本使用用户定义的操作
<code>adjacent_difference</code>	创建一个新序列，新序列中每个新值代表当前元素与上一个元素的差。重载版本用指定二元操作计算相邻元素的差

第五章 参考

<http://www.cnblogs.com/biyemyhjob/archive/2012/07/22/2603525.html>