# 一、常用STL

## list链表

### 1.初始化

list<string> lt(3,"a")

list<string>(3,"a") （匿名函数） ```

支持vector、数组初始化：

list<string> lt(a,a+4) a为数组

list<string> lt(b.begin(),b.end()) b为vector

有迭代器，不支持it+n

无法随机访问

### 2.方法

#### （1）后插：

lt.push\_back("a")

#### （2）前插：

lt.push\_front("a")

#### （3）后删：

lt.pop\_back()

#### （4）前删：

lt.pop\_front()

#### （5）指定位置插入

lt.insert(lt.begin(),"x") 在指定迭代器的位置插入

lt.insert(lt.begin(), n, "x") 插n个

lt.insert(lt.begin(), b.begin(), b.end()) 插入片段

lt.splice(lt.begin(), b) 将b插入到指定位置

lt.splice(lt.begin(), b, b.begin()) 将b中某位置插入指定位置

lt.splice(lt.begin(), b, b.begin(), b.end()) 片段差入

区别：insert是将副本插入，本体不变。splice是直接拼接，本体会清空。

#### （6）移除元素：

lt.erase(lt.begin())

lt.erase(lt.begin(),lt.end())

#### （7）遍历：

<1>迭代器遍历

for(list<int>::iterator it=a.begin();it!=a.end();it++){

cout<<\*it<<endl;

}

<2>auto遍历

#### （8）替代所有内容：

lt.assign(n,"a") 将lt变为n个"a"

lt.assgin(b.begin(),b.end()) 将lt变为b

#### （9）清空

lt.clear()

#### （10）删除某个值

lt.remove(2) 删除链表中所有为2的值

注意：其迭代器只支持自加++it，不支持it+n

#### （11）交换

lt.swap(b)

#### （12）排序

lt.sort(cmp)

#### （13）逆序

lt.reverse()

## stack栈

### 1.初始化：

stack<int> st; （也可以传第二个参数，定义其底层容器类型）

无迭代器

无法随机访问

### 2.方法：

#### （1）返回顶部元素：

st.top()

#### （2）入栈

st.push()

#### （3）出栈

st.pop()

#### （4）为空

st.empty()

## queue队列

### 1.初始化

queue<int> que

不支持利用数组、vector进行初始化

无迭代器

无法随机访问

### 2.方法

#### (1)进队

que.push(a)

#### (2)返回队头

que.front()

#### (3)返回队尾

que.back()

#### (4)出队

que.pop()

## priority\_queue优先队列

### 1.初始化

priority\_queue<int, vector<int>, less<int>> pq; 参数：类型、底层容器、顺序（less、greater），默认从大到小

支持数组、vector进行初始化

无迭代器  
 无法随机访问

### 2.方法

#### （1）进队

pq.push(a)

#### （2）返回队头

pq.top()

#### （3）出队

pq.pop()

## set

### 1.初始化

set内部通过红黑树实现，实现了一个自动排序（从小到大），元素值唯一的容器。查找的复杂度为（logn），set中的元素值不能直接被修改，在其中的查找属于二分查找。

set<int> se

有迭代器，不支持it+n

支持vector、list初始化

无法随机访问

### 2.方法

#### （1）插入

se.insert(a)

#### （2）删除

se.erase(iterator it)

#### （3）查找

se.find(2) //成功返回所在的迭代器，失败返回迭代器end()

se.count(2) //返回其数量

#### （4）寻找大于等于val的迭代器

se.lower\_bound(2)（底线）

#### （5）寻找大于val的迭代器

se.upper\_bound(2)

## vector

### 1.初始化

vector<int> vec;

vector<int> vec(num, val)

匿名 vector<int>(num,val)

return {1,2}

支持vector、数组初始化

拥有迭代器，可以it+n

### 2.方法

#### （1）尾插

vec.push\_back()

#### （2）删除

vec.erase(iterator) 删除iterator 所在位置的元素

vec.erase(it1,it2) 删除两者之间的元素，包括自身

vec.pop\_back()删除最后一个

#### （3）索引

vec[i] //由于vector重载了[]，所以可以利用[]直接访问已有元素

vec.at(1) //at()函数具有检测是否越界的功能，如果越界会抛出错误，所以安全性高于[]

#### （4）返回头部

vec.front()

#### （5）返回尾部

vec.back()

#### （6）sort、reverse、swap

#### （7）查找

vector<**int**>::iterator iter=find(vec.begin(),vec.end(),3);

（8）求最小值

vector<int>::iterator iter = min\_element(vec.begin(), vec.end())

## pair

### 1.定义

（1）pair<string, int> p1("qzy",2)

（2）pair<stirng, int> p1;

p1.first = "qzy";

p1.second = 3

（3）p1 = make\_pair("qzy",3)

## map

### 1.定义

map<string, int> m;

有迭代器，只支持自增it++

### 2.方法

#### （1）插入

m.insert(make\_pair("one", 1))

m.insert(pair<string, int>("one", 1))

m["one"]++

#### （2）删除

m.erase(iterator)

m.erase(it.begin(), it.end())

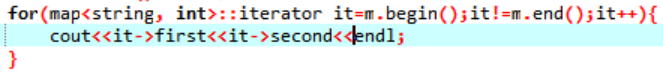
#### （3）查找

m.find(value) //找到就返回迭代器，否则返回end()

m.lower\_bound(1); //lower\_bound(查找第一个大于等于key的值)

m.upper.bound(1); //upper\_bound(查找第一个大于key的值)

（4）遍历



## deque双端队列

### 1.定义

deque<int> dq;

deque<int> dq(n,elem)

支持vector、数组初始化

### 2.方法

#### （1）尾插

push\_back(elem)

#### （2）尾删

pop\_back()

#### （3）头插

push\_front(elem)

#### （4）头删

pop\_front()

#### （5）取头

dq.front()

#### （6）取尾

dq.back()

#### （7）索引

dq[i]

dq.at(i)

#### （8）插入

dq.insert(pos, elem)

dq.insert(pos, n, elem)

dq.insert(pos, begin(),end())

#### （9）删除

dq.erase(iterator) 删除迭代器位置，并返回其下一个位置

dq.erase(it.begin(), it.end()) 删除区间，并返回下个位置

#### （10）覆盖

deque.assgin(a.begin(), a.end())

#### （11）重新定义长度

deque.resize(num，elem) //定义长度为num，超出的丢弃，多余的用elem填充

# 二、数据结构

## （一）二叉树

### 1.二叉搜索树

也叫：二叉排序树、二叉查找树、二叉搜索树

#### （1）定义：

根节点的值大于其左子树中任意一个节点的值，小于其右节点中任意一节点的值。（符合所有节点）

### 2.平衡二叉树

#### （1）定义

每个节点的左右两个子树的高度差的绝对值不超过1。

### 3.完全二叉树

#### （1）定义

每层都达到最大个数，最后一层叶子节点都连续靠左。

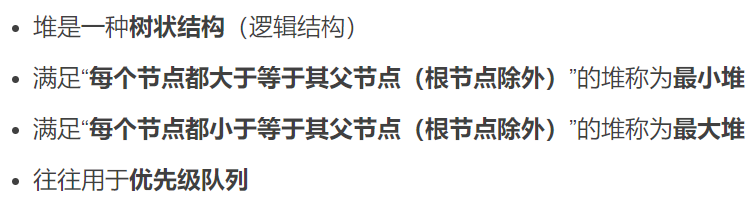
### 4.红黑树

#### （1）定义

1）每个节点或者是黑色，或者是红色。  
2）根节点是黑色。  
3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]  
4）如果一个节点是红色的，则它的子节点必须是黑色的。  
5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

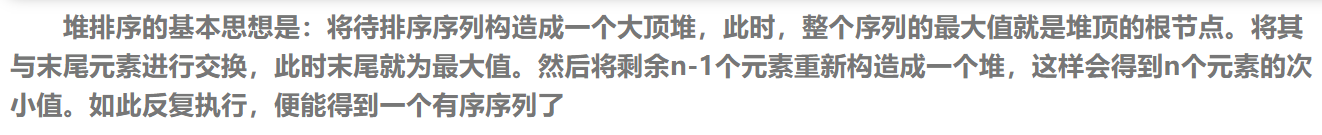
## （二）堆

### 1.定义



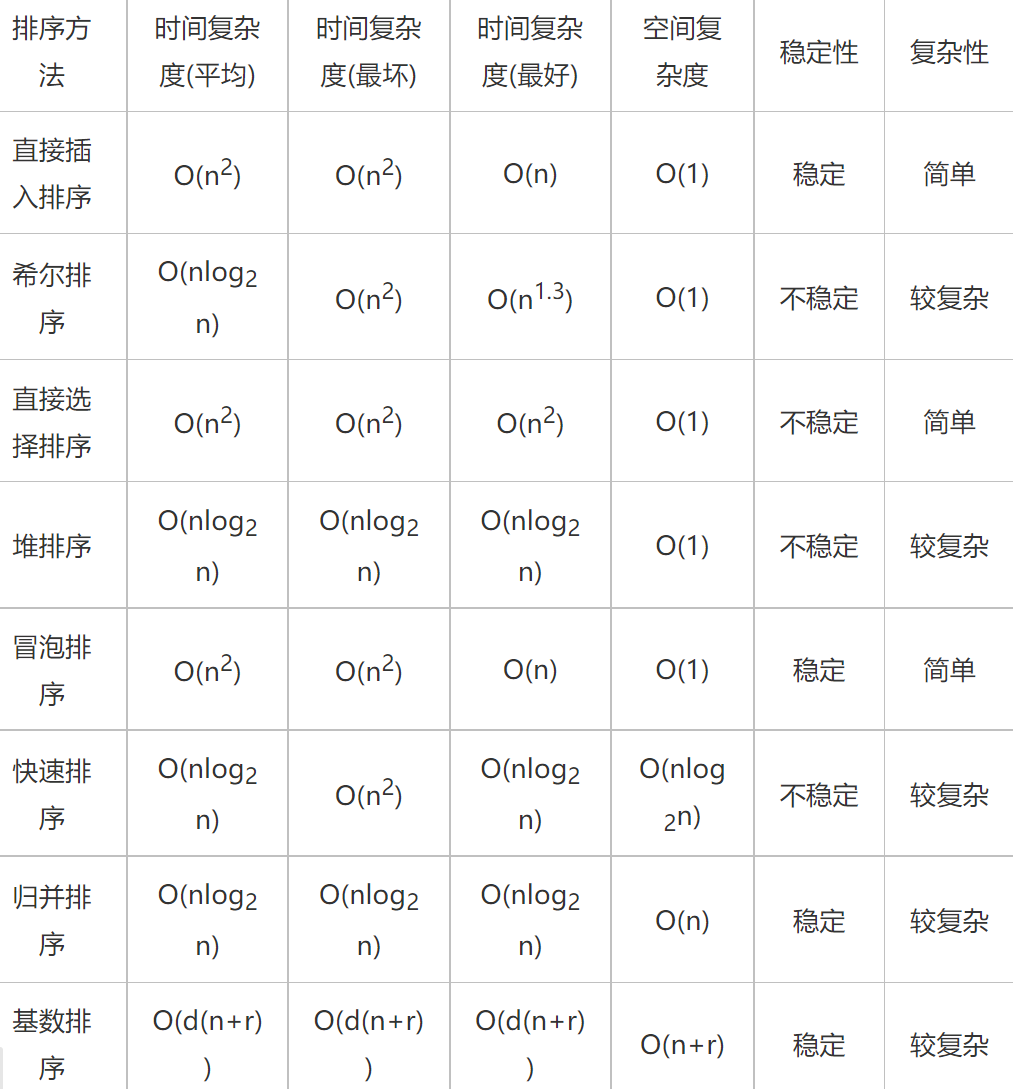
注：树状结构一般是完全二叉树

### 2.堆排序



# 三、算法

## （一）排序算法

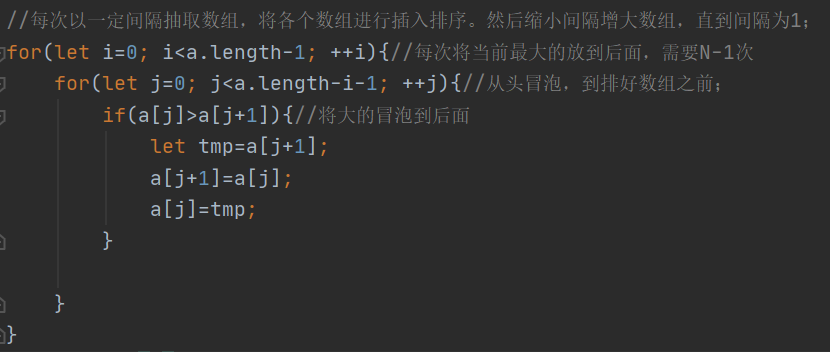


**比较排序：**

### 1.交换排序

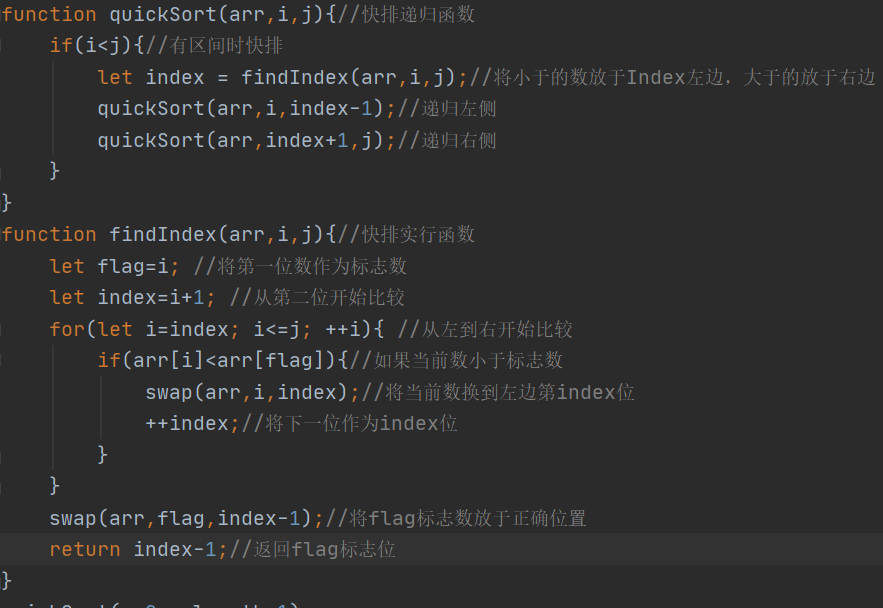
#### （1）冒泡排序

从头到尾，每次比较相邻的两个，将大的放在后面，这样每次都将把当前极大值放在末尾，重复n遍后得到结果。



#### （2）快速排序

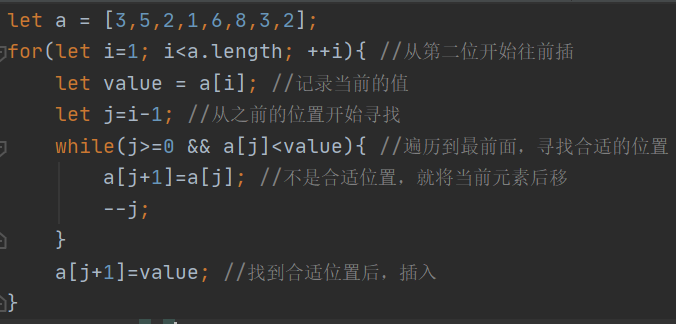
随机选一个数作为标准，遍历剩下的数，比它小的放在左边，比它大的放在右边。遍历结束后，标准数就位于正确位置了。然后对标准两边的数组进行递归，直到无法递归时排序完成。

****

### 2.插入排序

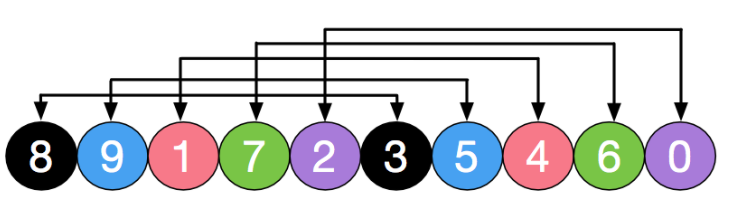
#### （1）简单插入排序

依次每次选一个数，插入到之前的有序数列中。（让其他的数后移，在需要插入的地方停止后移）

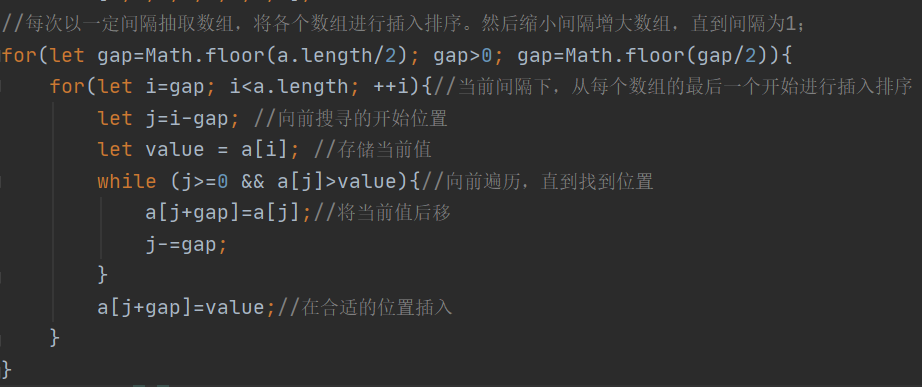


#### （2）希尔排序

是简单插入排序的改进版。它与插入排序的不同之处在于，它会优先比较距离较远的元素，组成多个分组分别进行插入排序，然后缩小分组数量。希尔排序又叫**缩小增量排序**。



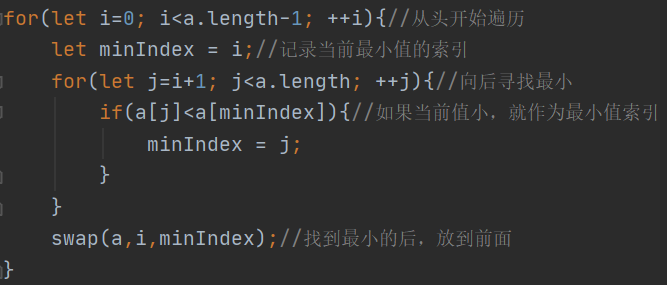
在实际实现的时候，是将每个元素作为待插入数，间隔gap内寻找插入位置。这种遍历的gap可以覆盖所有元素的插入情况（每次按间隔分组的每个数都会作为待插入元素）。



### 3.选择排序

#### （1）简单选择排序

从头到尾，寻找最小的那个，将其放在最前面。



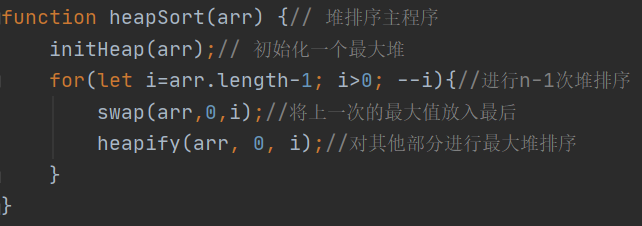
#### （2）堆排序

1）先将数组构建成最大堆（每个父节点都大于子节点）

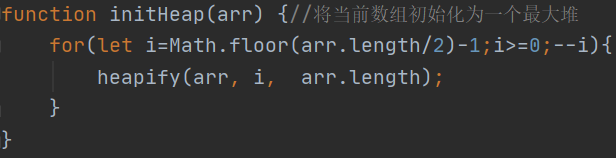
2）将极大值的根节点和最后的节点交换，放在最后面。

3）对除了极大值外的节点继续构造最大堆。不断迭代。

4）全部遍历后得到有序数组。



初始化：



将子树变成大顶堆

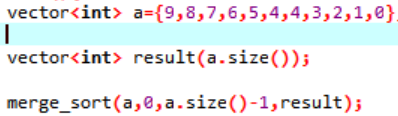


### 4.归并排序

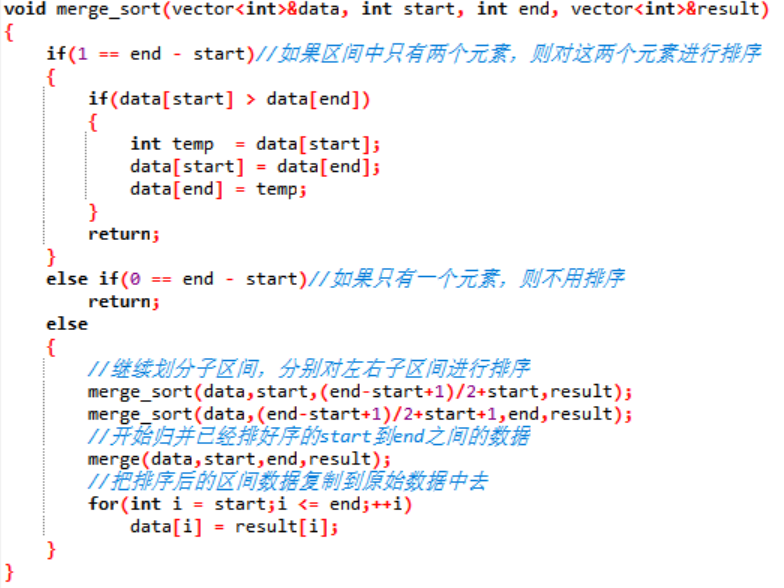
#### （1）二路归并排序

将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

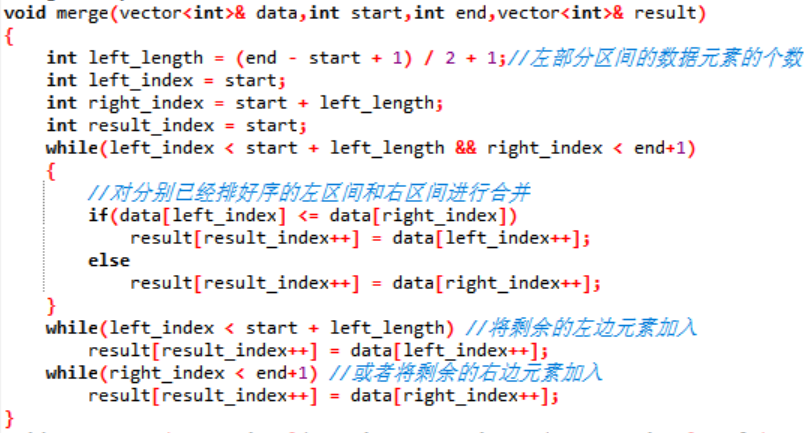
##### 主程序：



##### 递归：



##### 排序：



#### （2）多路归并排序

**非比较排序：**

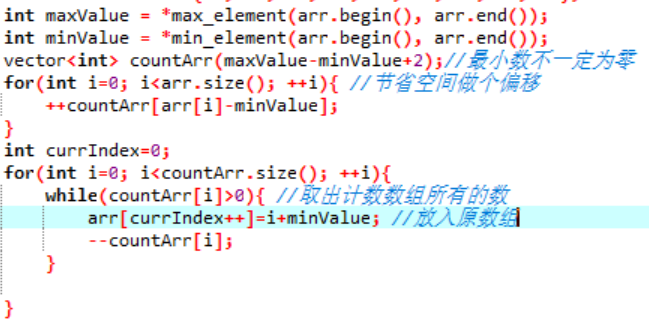
### 5.非比较排序

#### （1）计数排序

<1>根据原数组最大最小值建立计数数组，计数数组的索引为原数组的值

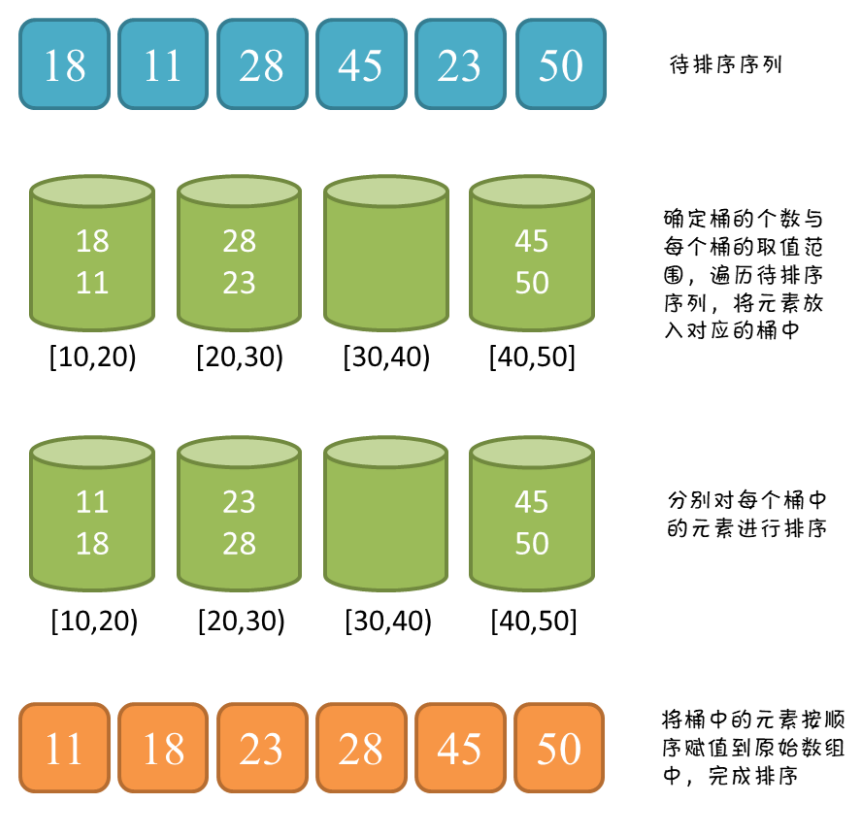
<2>将原数组每个数映射到计数数组

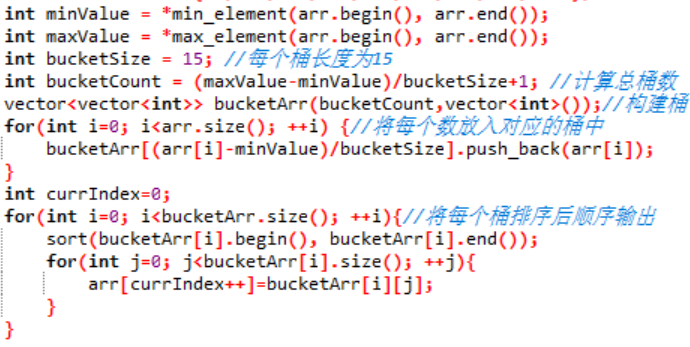
<3>遍历计数数组，取出后顺序放入原数组，完成排序

****

#### （2）桶排序

是计数排序的扩展，计数排序的每个索引只能存放同一个值，桶排序拓展了索引，令一个桶存放一定范围的数据，将每个桶内部排序，合并后实现排序。





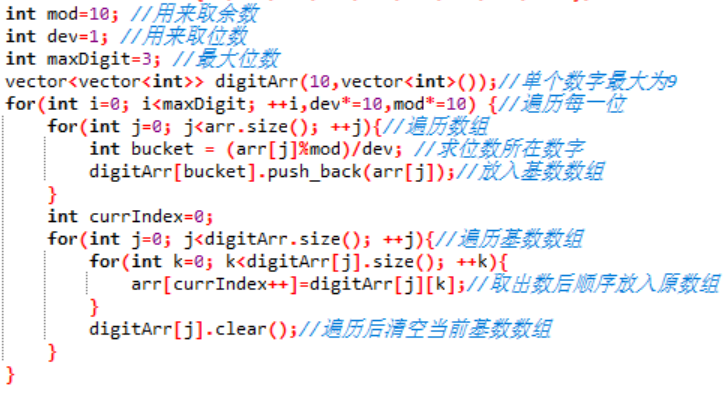
#### （3）基数排序

<1>根据位数建立索引表

<2>先按照低位大小，将原数组放入索引表

<3>从索引表顺序取出，再增加按照较高位大小，将原数组放入索引表

<4>直到所有位数都遍历后，完成排序。

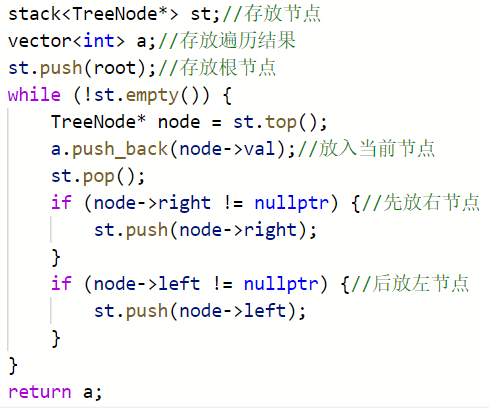


## （二）查找算法

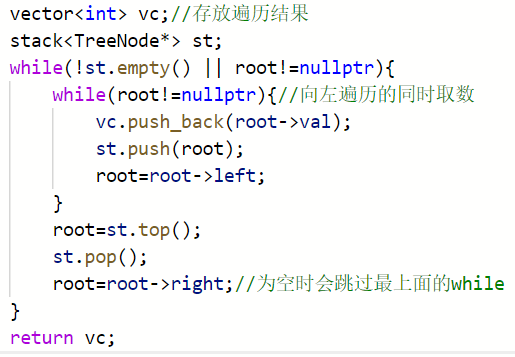
### 1.二叉树查找

#### （1）前序遍历

**迭代：**

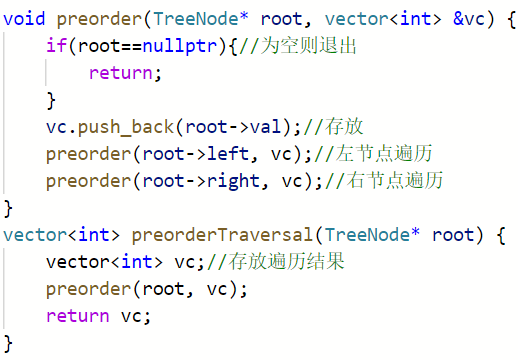


（也可以入栈后再取数）：



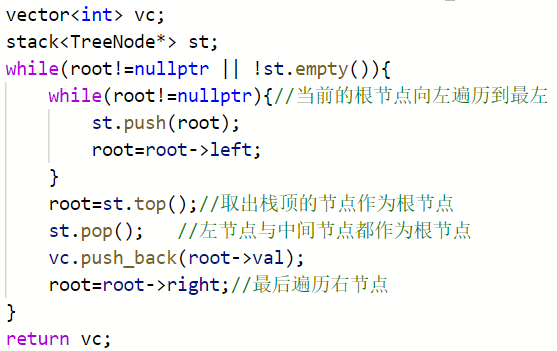
以根节点为基准入栈，分别将右左节点入栈

**递归：**

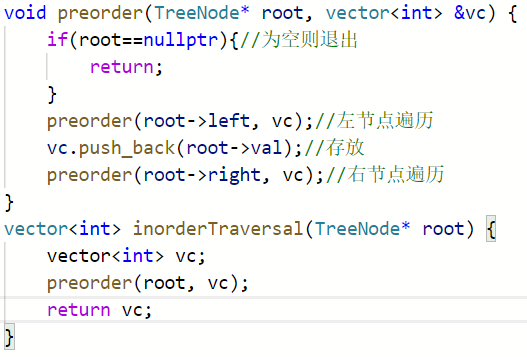
****

#### （2）中序遍历

**迭代：**

****

**递归：**



#### （3）后序遍历

**迭代：**

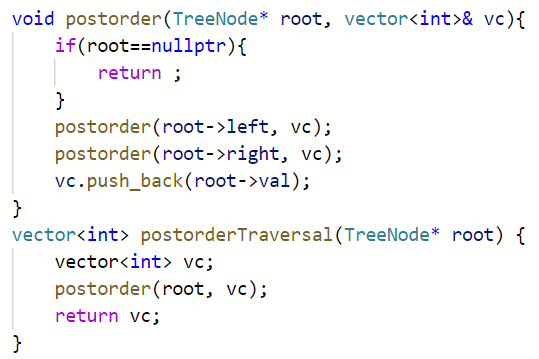
（反向遍历后，反转数组）

****

（高时间效率版，不推荐）

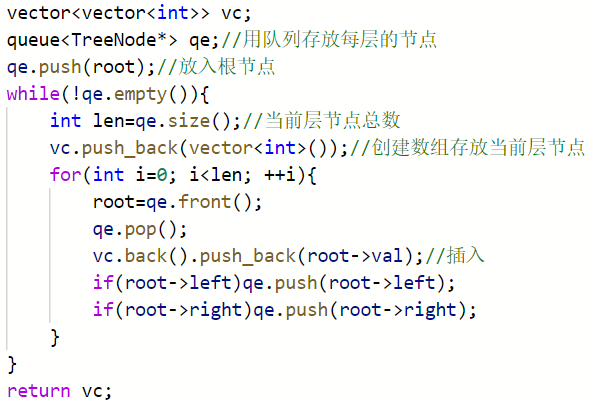
****

**递归：**

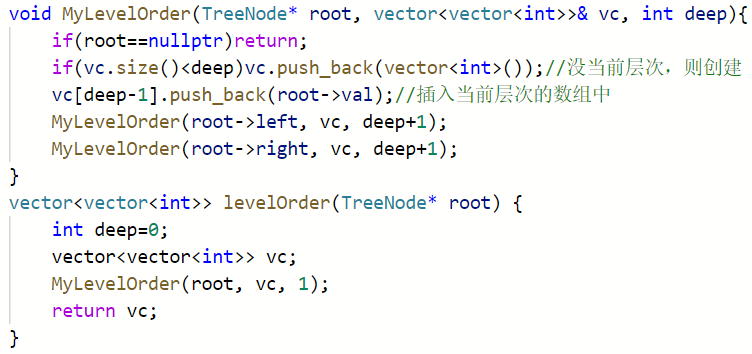


#### （4）层次遍历

**迭代：**

****

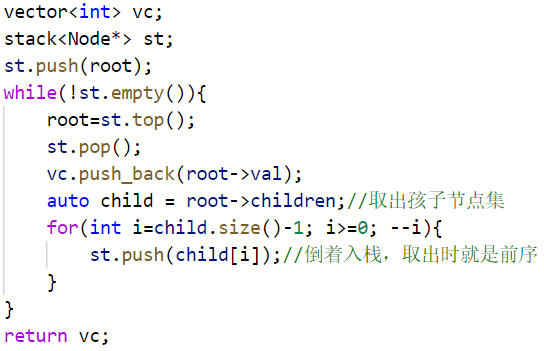
**递归：**



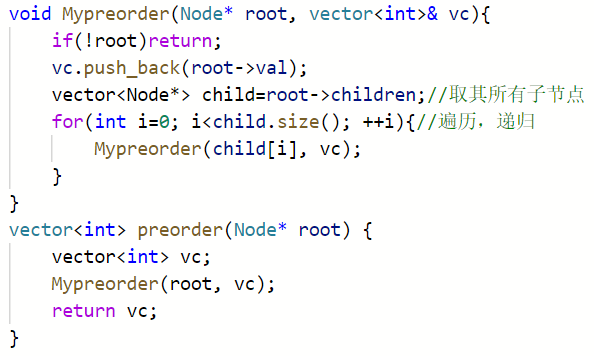
### 2.N叉树

#### （1）前序遍历

**迭代：**

****

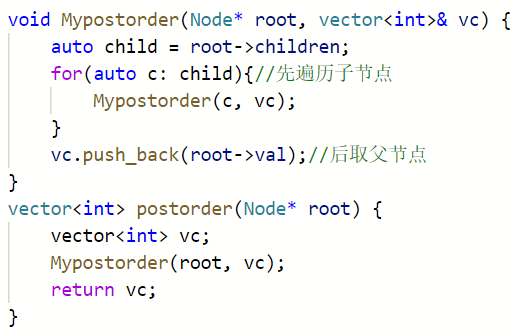
**递归：**

****

#### （2）后序遍历

**迭代：过于复杂，放弃**

**递归：**



### 3.堆查找

### 4.链查找

### 5.栈、队列

## （三）数据结构算法

1.树排序

2.堆排序

3.链排序

头插、穿针引线

## 矩阵旋转、对称

## 进制转换

构造数据结构stack/queue/....