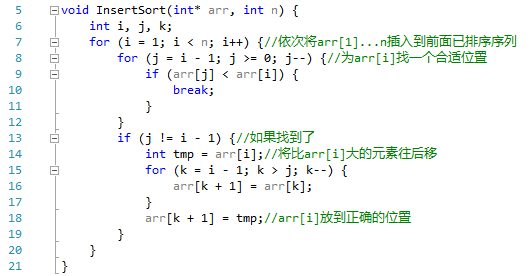
**5．常用数据结构与算法**

**5.1 直接插入排序**

直接插入排序(Straight Insertion Sort)的基本思想是：把n个待排序的元素看成为一个有序表和一个无序表。开始时有序表中只包含1个元素，无序表中包含有n-1个元素，排序过程中每次从无序表中取出第一个元素，将它插入到有序表中的适当位置，使之成为新的有序表，重复n-1次可完成排序过程。

O（n2） 稳定

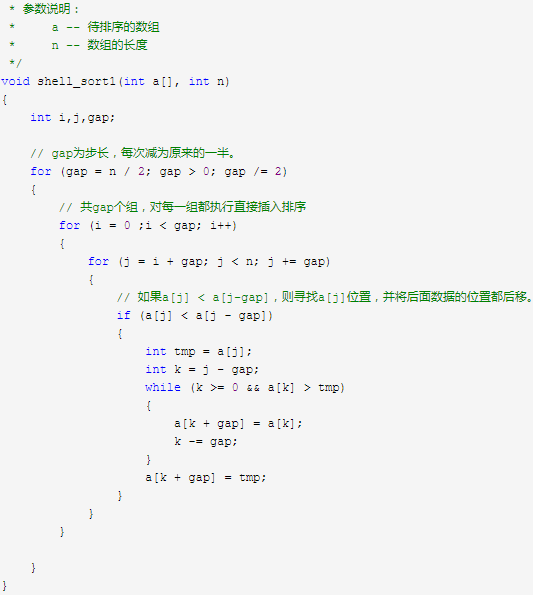


**5.2 希尔排序**

希尔排序实质上是一种分组插入方法。它的基本思想是：对于n个待排序的数列，取一个小于n的整数gap(gap被称为步长)将待排序元素分成若干个组子序列，所有距离为gap的倍数的记录放在同一个组中；然后，对各组内的元素进行直接插入排序。 这一趟排序完成之后，每一个组的元素都是有序的。然后减小gap的值，并重复执行上述的分组和排序。重复这样的操作，当gap=1时，整个数列就是有序的。

希尔排序的时间复杂度与增量(即，步长gap)的选取有关。例如，当增量为1时，希尔排序退化成了直接插入排序，此时的时间复杂度为O(N²)，而Hibbard增量的希尔排序的时间复杂度为O(N3/2)。

希尔排序是不稳定的算法，它满足稳定算法的定义。对于相同的两个数，可能由于分在不同的组中而导致它们的顺序发生变化。

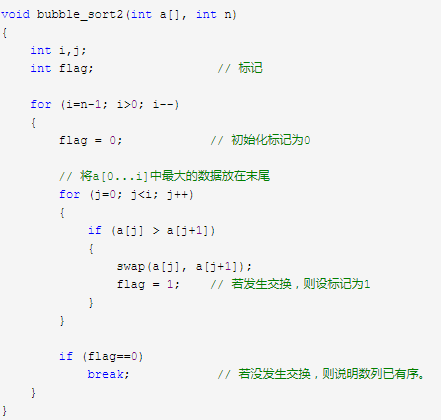


**5.3 冒泡排序**

冒泡排序(Bubble Sort)，又被称为气泡排序或泡沫排序。

它是一种较简单的排序算法。它会遍历若干次要排序的数列，每次遍历时，它都会从前往后依次的比较相邻两个数的大小；如果前者比后者大，则交换它们的位置。这样，一次遍历之后，最大的元素就在数列的末尾！ 采用相同的方法再次遍历时，第二大的元素就被排列在最大元素之前。重复此操作，直到整个数列都有序为止！

冒泡排序的时间复杂度是O(N2)，稳定



**5.4 快速排序**

快速排序(Quick Sort)使用分治法策略。

它的基本思想是：选择一个基准数，通过一趟排序将要排序的数据分割成独立的两部分；其中一部分的所有数据都比另外一部分的所有数据都要小。然后，再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

快速排序流程：

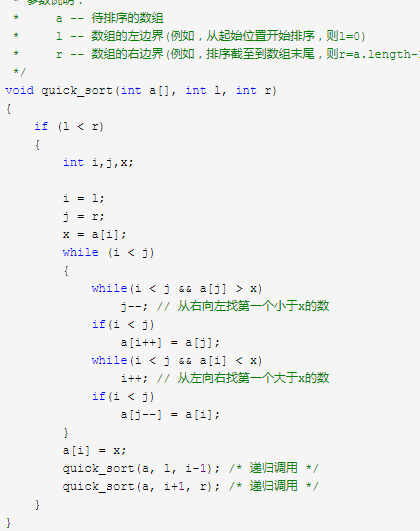
(1) 从数列中挑出一个基准值。

(2) 将所有比基准值小的摆放在基准前面，所有比基准值大的摆在基准的后面(相同的数可以到任一边)；在这个分区退出之后，该基准就处于数列的中间位置。

(3) 递归地把"基准值前面的子数列"和"基准值后面的子数列"进行排序。

快速排序的时间复杂度在最坏情况下是O(N2)，平均的时间复杂度是O(N\*lgN)

不稳定



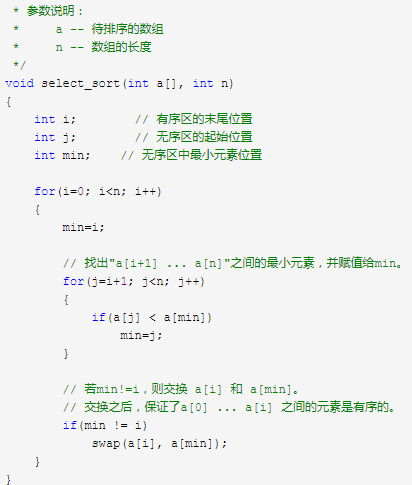
**5.5 选择排序**

选择排序(Selection sort)是一种简单直观的排序算法。

它的基本思想是：首先在未排序的数列中找到最小(or最大)元素，然后将其存放到数列的起始位置；接着，再从剩余未排序的元素中继续寻找最小(or最大)元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

O(N2)

稳定



**5.6 堆排序**

堆排序(Heap Sort)是指利用堆这种数据结构所设计的一种排序算法。

最大堆通常被用来进行"升序"排序，而最小堆通常被用来进行"降序"排序。

最大堆进行升序排序的基本思想：

① 初始化堆：将数列a[1...n]构造成最大堆。

② 交换数据：将a[1]和a[n]交换，使a[n]是a[1...n]中的最大值；然后将a[1...n-1]重新调整为最大堆。 接着，将a[1]和a[n-1]交换，使a[n-1]是a[1...n-1]中的最大值；然后将a[1...n-2]重新调整为最大值。 依次类推，直到整个数列都是有序的。

"数组实现的二叉堆的性质"：

在第一个元素的索引为 0 的情形中：

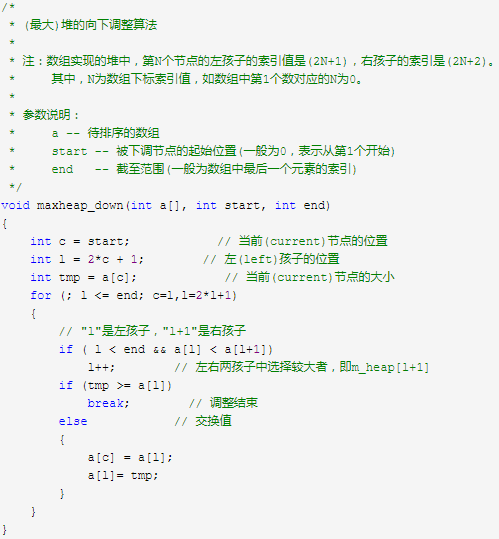
性质一：索引为i的左孩子的索引是 (2\*i+1);

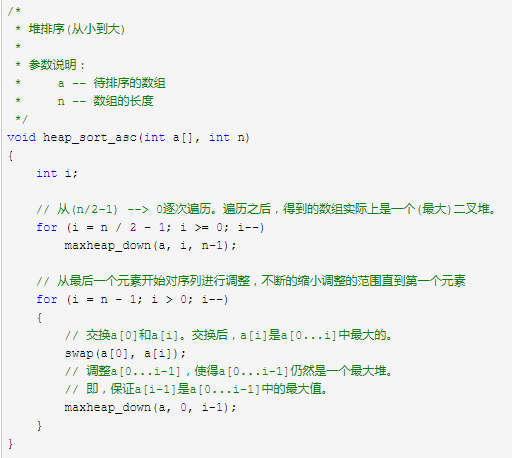
性质二：索引为i的左孩子的索引是 (2\*i+2);

性质三：索引为i的父结点的索引是 floor((i-1)/2);

堆排序的时间复杂度是O(N\*lgN)。

不稳定





**5.7 归并排序**

将两个的有序数列合并成一个有序数列，我们称之为"归并"。

归并排序(Merge Sort)就是利用归并思想对数列进行排序。根据具体的实现，归并排序包括"从上往下"和"从下往上"2种方式。

1. 从下往上的归并排序：将待排序的数列分成若干个长度为1的子数列，然后将这些数列两两合并；得到若干个长度为2的有序数列，再将这些数列两两合并；得到若干个长度为4的有序数列，再将它们两两合并；直接合并成一个数列为止。这样就得到了我们想要的排序结果。(参考下面的图片)

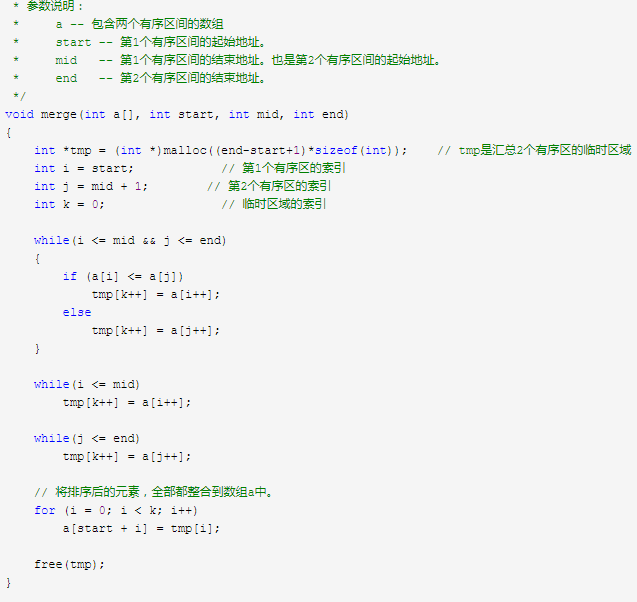
2. 从上往下的归并排序：它与"从下往上"在排序上是反方向的。它基本包括3步：

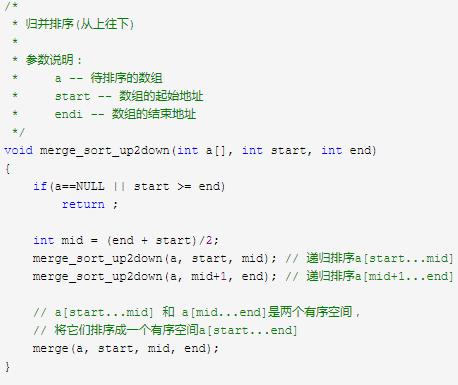
① 分解 -- 将当前区间一分为二，即求分裂点 mid = (low + high)/2;

② 求解 -- 递归地对两个子区间a[low...mid] 和 a[mid+1...high]进行归并排序。递归的终结条件是子区间长度为1。

③ 合并 -- 将已排序的两个子区间a[low...mid]和 a[mid+1...high]归并为一个有序的区间a[low...high]。







归并排序的时间复杂度是O(N\*lgN)。

稳定

**5.8 二分查找**

**5.9 LRU算法**

LRU (Least recently used) 最近最少使用，如果数据最近被访问过，那么将来被访问的几率也更高，LRU算法会将近期最不会访问的数据淘汰掉

实现思路：  map 存储数据 ，实现查找效率O(1)，双向链表实现算法逻辑

算法逻辑： 1.新数据会插入到链表头部

               2.当缓存数据被访问，将该缓存数据移到链表头部

               3.当新数据插入时达到缓存上限了，将尾部数据删除掉（也就是最近最少使用的），新数据放在头部。

#include<iostream>

#include<map>

using namespace std;

//\*\* 链表节点结构体

struct ListNode

{

int m\_key; //key,value 形式方便map存储。

int m\_value;

ListNode\* pPre;

ListNode\* pNext;

ListNode(int key ,int value)

{

m\_key = key;

m\_value=value;

pPre=NULL;

pNext=NULL;

}

};

//\* LRU缓存实现类 双向链表。

class LRUCache

{

public:

//\*\* 构造函数初始化缓存大小

LRUCache(int size)

{

m\_capacity = size;

pHead == NULL;

pTail == NULL;

}

~LRUCache()

{

//\*\* 一定要注意，map释放内存时，先释放内部new的内存，在释放map的内存

map<int , ListNode\*>::iterator it =mp.begin();

for(;it!=mp.end();)

{

delete it->second;

it->second = NULL;

mp.erase(it++); //\*\* 注意：一定要这样写，it++ 放在其他任何一个地方都会导致其迭代器失效。

}

delete pHead;

pHead == NULL;

delete pTail;

pTail ==NULL;

}

//\*\* 这里只是移除，并不删除节点

void Remove(ListNode\* pNode)

{

// 如果是头节点

if(pNode->pPre == NULL)

{

pHead = pNode->pNext;

pHead->pPre = NULL;

}

// 如果是尾节点

if(pNode->pNext == NULL)

{

pTail = pNode->pPre;

pTail->pNext = NULL;

}

else

{

pNode->pPre->pNext=pNode->pNext;

pNode->pNext->pPre = pNode->pPre;

}

}

// 将节点放到头部，最近用过的数据要放在队头。

void SetHead(ListNode\* pNode)

{

pNode->pNext = pHead;

pNode->pPre = NULL;

if(pHead == NULL)

{

pHead = pNode;

}

else

{

pHead->pPre=pNode;

pHead = pNode;

}

if(pTail == NULL)

{

pTail = pHead;

}

}

// \* 插入数据，如果存在就只更新数据

int Set(int key, int value)

{

map<int,ListNode\*>::iterator it=mp.find(key);

if(it != mp.end())

{

ListNode\* Node=it->second;

Node->m\_value=value;

Remove(Node);

SetHead(Node);

}

else

{

ListNode\* NewNode = new ListNode(key,value);

if(mp.size() >= m\_capacity)

{

map<int,ListNode\*>::iterator it=mp.find(pTail->m\_key);

//从链表移除

Remove(pTail);

//删除指针指向的内存

delete it->second;

//删除map元素

mp.erase(it);

}

//放到头部

SetHead(NewNode);

mp[key] = NewNode;

}

}

//获取缓存里的数据

int Get(int key)

{

map<int , ListNode\*>:: iterator it = mp.find(key);

if(it != mp.end())

{

ListNode\* Node=it->second;

Remove(Node);

SetHead(Node);

return Node->m\_value;

}

else

{

return -1; //这里不太好，有可能取得值也为-1

}

}

int GetSize()

{

return mp.size();

}

private:

int m\_capacity; //缓存容量

ListNode\* pHead; //头节点

ListNode\* pTail; //尾节点

map<int, ListNode\*> mp; //mp用来存数据，达到find为o(1)级别。

};