

白话经典算法系列（转载）

原文作者：MoreWindows

目录

白话经典算法系列（转载）	1
白话经典算法系列之一 冒泡排序的三种实现	2
白话经典算法系列之二 直接插入排序的三种实现	4
白话经典算法系列之三 希尔排序的实现	6
白话经典算法系列之四 直接选择排序及交换二个数据的正确实现	9
白话经典算法系列之五 归并排序的实现	11
白话经典算法系列之六 快速排序 快速搞定	15
白话经典算法系列之七 堆与堆排序	19
二叉堆的定义	19
堆的存储	19
堆的操作——插入删除	20
堆的插入	21
堆的删除	21
堆化数组	22
堆排序	24
转载 请 标 明 出 处 ， 原 文 地 址 ： http://www.cnblogs.com/morewindows/archive/2011/08/22/2149612.html	24

白话经典算法系列之一 冒泡排序的三种实现

冒泡排序是很容易理解和实现，以从小到大排序举例：

设数组长度为 N 。

1. 比较相邻的前后二个数据，如果前面数据大于后面的数据，就将二个数据交换。
2. 这样对数组的第 0 个数据到 $N-1$ 个数据进行一次遍历后，最大的一个数据就“沉”到数组第 $N-1$ 个位置。
3. $N=N-1$ ，如果 N 不为 0 就重复前面二步，否则排序完成。

按照定义很容易写出代码：

//冒泡排序 1

```
void BubbleSort1(int a[], int n)
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 1; j < n - i; j++)
            if (a[j - 1] > a[j])
                Swap(a[j - 1], a[j]);
}
```

下面对其进行优化，设置一个标志，如果这一趟发生了交换，则为 `true`，否则为 `false`。明显如果有一趟没有发生交换，说明排序已经完成。

//冒泡排序 2

```
void BubbleSort2(int a[], int n)
{
    int j, k;
    bool flag;

    k = n;
    flag = true;
    while (flag)
```

```

{
    flag = false;
    for (j = 1; j < k; j++)
        if (a[j - 1] > a[j])
        {
            Swap(a[j - 1], a[j]);
            flag = true;
        }
    k--;
}
}

```

再做进一步的优化。如果有 100 个数的数组，仅前面 10 个无序，后面 90 个都已排好序且都大于前面 10 个数字，那么在第一趟遍历后，最后发生交换的位置必定小于 10，且这个位置之后的数据必定已经有序了，记录下这位置，第二次只要从数组头部遍历到这个位置就可以了。

//冒泡排序 3

```

void BubbleSort3(int a[], int n)
{
    int j, k;
    int flag;

    flag = n;
    while (flag > 0)
    {
        k = flag;
        flag = 0;
        for (j = 1; j < k; j++)
            if (a[j - 1] > a[j])
            {
                Swap(a[j - 1], a[j]);

```

```

        flag = j;
    }
}
}

```

冒泡排序毕竟是一种效率低下的排序方法，在数据规模很小时，可以采用。数据规模比较大时，最好用其它排序方法。

白话经典算法系列之二 直接插入排序的三种实现

直接插入排序(Insertion Sort)的基本思想是：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中的适当位置，直到全部记录插入完成为止。

设数组为 $a[0...n-1]$ 。

1. 初始时， $a[0]$ 自成 1 个有序区，无序区为 $a[1..n-1]$ 。令 $i=1$
2. 将 $a[i]$ 并入当前的有序区 $a[0...i-1]$ 中形成 $a[0...i]$ 的有序区间。
3. $i++$ 并重复第二步直到 $i==n-1$ 。排序完成。

下面给出严格按照定义书写的代码（由小到大排序）：

```

void Insertsort1(int a[], int n)
{
    int i, j, k;
    for (i = 1; i < n; i++)
    {
        //为 a[i]在前面的 a[0...i-1]有序区间中找一个合适的位置
        for (j = i - 1; j >= 0; j--)
            if (a[j] < a[i])
                break;
        //如找到了一个合适的位置
        if (j != i - 1)
        {
            //将比 a[i]大的数据向后移

```

```

        int temp = a[i];
        for (k = i - 1; k > j; k--)
            a[k + 1] = a[k];
        //将 a[i]放到正确位置上
        a[k + 1] = temp;
    }
}

```

这样的代码太长了，不够清晰。现在进行一下改写，将搜索和数据后移这二个步骤合并。即每次 $a[i]$ 先和前面一个数据 $a[i-1]$ 比较，如果 $a[i] > a[i-1]$ 说明 $a[0...i]$ 也是有序的，无须调整。否则就令 $j=i-1, temp=a[i]$ 。然后一边将数据 $a[j]$ 向后移动一边向前搜索，当有数据 $a[j] < a[i]$ 时停止并将 $temp$ 放到 $a[j + 1]$ 处。

```

void Insertsort2(int a[], int n)
{
    int i, j;
    for (i = 1; i < n; i++)
        if (a[i] < a[i - 1])
        {
            int temp = a[i];
            for (j = i - 1; j >= 0 && a[j] > temp; j--)
                a[j + 1] = a[j];
            a[j + 1] = temp;
        }
}

```

再对将 $a[j]$ 插入到前面 $a[0...j-1]$ 的有序区间所用的方法进行改写，用数据交换代替数据后移。如果 $a[j]$ 前一个数据 $a[j-1] > a[j]$ ，就交换 $a[j]$ 和 $a[j-1]$ ，再 $j--$ 直到 $a[j-1] \leq a[j]$ 。这样也可以实现将一个新数据新并入到有序区间。

```

void Insertsort3(int a[], int n)
{
    int i, j;

```

```

    for (i = 1; i < n; i++)
        for (j = i - 1; j >= 0 && a[j] > a[j + 1]; j--)
            Swap(a[j], a[j + 1]);
}

```

白话经典算法系列之三 希尔排序的实现

希尔排序的实质就是分组插入排序，该方法又称缩小增量排序，因 DL. Shell 于 1959 年提出而得名。

该方法的基本思想是：先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前两种方法有较大提高。

以 $n=10$ 的一个数组 49, 38, 65, 97, 26, 13, 27, 49, 55, 4 为例

第一次 $gap = 10 / 2 = 5$

49 38 65 97 26 13 27 49 55 4

1A 1B

2A 2B

3A 3B

4A 4B

5A 5B

1A,1B, 2A,2B 等为分组标记，数字相同的表示在同一组，大写字母表示是该组的第几个元素，每次对同一组的数据进行直接插入排序。即分成了五组(49, 13) (38, 27) (65, 49) (97, 55) (26, 4)这样每组排序后就变成了(13, 49) (27, 38) (49, 65) (55, 97) (4, 26)，下同。

第二次 $gap = 5 / 2 = 2$

排序后

13 27 49 55 4 49 38 65 97 26

1A 1B 1C 1D 1E

2A 2B 2C 2D 2E

第三次 $\text{gap} = 2 / 2 = 1$

4 26 13 27 38 49 49 55 97 65

1A 1B 1C 1D 1E 1F 1G 1H 1I 1J

第四次 $\text{gap} = 1 / 2 = 0$ 排序完成得到数组:

4 13 26 27 38 49 49 55 65 97

下面给出严格按照定义来写的希尔排序

```
void shellsort1(int a[], int n)
{
    int i, j, gap;
    for (gap = n / 2; gap > 0; gap /= 2) //步长
        for (i = 0; i < gap; i++) //按组排序
        {
            for (j = i + gap; j < n; j += gap)
            {
                if (a[j] < a[j - gap])
                {
                    int temp = a[j];
                    int k = j - gap;
                    while (k >= 0 && a[k] > temp)
                    {
                        a[k + gap] = a[k];
                        k -= gap;
                    }
                    a[k + gap] = temp;
                }
            }
        }
}
```

```
}
```

很明显，上面的 `shellsort1` 代码虽然对直观的理解希尔排序有帮助，但代码量太大了，不够简洁清晰。因此进行下改进和优化，以第二次排序为例，原来是每次从 1A 到 1E，从 2A 到 2E，可以改成从 1B 开始，先和 1A 比较，然后取 2B 与 2A 比较，再取 1C 与前面自己组内的数据比较.....。这种每次从数组第 `gap` 个元素开始，每个元素与自己组内的数据进行直接插入排序显然也是正确的。

```
void shellsort2(int a[], int n)
{
    int j, gap;

    for (gap = n / 2; gap > 0; gap /= 2)
        for (j = gap; j < n; j++) //从数组第 gap 个元素开始
            if (a[j] < a[j - gap]) //每个元素与自己组内的数据进行直接插入排序
            {
                int temp = a[j];
                int k = j - gap;
                while (k >= 0 && a[k] > temp)
                {
                    a[k + gap] = a[k];
                    k -= gap;
                }
                a[k + gap] = temp;
            }
}
```

再将直接插入排序部分用 [白话经典算法系列之二 直接插入排序的三种实现](#) 中直接插入排序的第三种方法来改写下：

```
void shellsort3(int a[], int n)
{
    int i, j, gap;

    for (gap = n / 2; gap > 0; gap /= 2)
```



```

        for (i = gap; i < n; i++)
            for (j = i - gap; j >= 0 && a[j] > a[j + gap]; j -= gap)
                Swap(a[j], a[j + gap]);
    }

```

这样代码就变得非常简洁了。

附注：上面希尔排序的步长选择都是从 $n/2$ 开始，每次再减半，直到最后为 1。

其实也可以有另外的更高效的步长选择，如果读者有兴趣了解，请参阅维基百科上对希尔排序步长的说明：

<http://zh.wikipedia.org/wiki/%E5%B8%8C%E5%B0%94%E6%8E%92%E5%BA%8F>

白话经典算法系列之四 直接选择排序及交换二个数据的正 确实现

直接选择排序和[直接插入排序](#)类似，都将数据分为有序区和无序区，所不同的是直接播放排序是将无序区的第一个元素直接插入到有序区以形成一个更大的有序区，而直接选择排序是从无序区选一个最小的元素直接放到有序区的最后。

设数组为 $a[0\dots n-1]$ 。

1. 初始时，数组全为无序区为 $a[0\dots n-1]$ 。令 $i=0$
2. 在无序区 $a[i\dots n-1]$ 中选取一个最小的元素，将其与 $a[i]$ 交换。交换之后 $a[0\dots i]$ 就形成了一个有序区。
3. $i++$ 并重复第二步直到 $i==n-1$ 。排序完成。

直接选择排序无疑是最容易实现的，下面给出代码：

```

void Selectsort(int a[], int n)
{
    int i, j, nMinIndex;
    for (i = 0; i < n; i++)
    {
        nMinIndex = i; //找最小元素的位置
        for (j = i + 1; j < n; j++)

```

```

        if (a[j] < a[nMinIndex])
            nMinIndex = j;
        Swap(a[i], a[nMinIndex]); //将这个元素放到无序区的开头
    }
}

```

在这里，要特别提醒各位注意下 Swap()的实现，建议用：

```

inline void Swap(int &a, int &b)
{
    int c = a;
    a = b;
    b = c;
}

```

笔试面试时考不用中间数据交换二个数，很多人给出了

```

inline void Swap1(int &a, int &b)
{
    a ^= b;
    b ^= a;
    a ^= b;
}

```

在网上搜索下，也可以找到许多这样的写法。不过这样写存在一个隐患，如果 a, b 指向的是同一个数，那么调用 Swap1()函数会使这个数为 0。如：

```

int i = 6;
Swap1(i, i);
printf("%d\n", i);

```

当然谁都不会在程序中这样的写代码，但回到我们的 Selectsort()，如果 a[0]就是最小的数，那么在交换时，将会出现将 a[0]置 0 的情况，这种错误相信调试起来也很难发现吧，因此建议大家将交换二数的函数写成：

```

inline void Swap(int &a, int &b)
{
    int c = a;

```

```
    a = b;  
    b = c;  
}
```

或者在 Swap1()中加个判断，如果二个数据相等就不用交换了：

```
inline void Swap1(int &a, int &b)  
{  
    if (a != b)  
    {  
        a ^= b;  
        b ^= a;  
        a ^= b;  
    }  
}
```

白话经典算法系列之五 归并排序的实现

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

首先考虑下如何将二个有序数列合并。这个非常简单，只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。

//将有序数组 a[]和 b[]合并到 c[]中

```
void MemeryArray(int a[], int n, int b[], int m, int c[])  
{  
    int i, j, k;  
    i = j = k = 0;  
    while (i < n && j < m)
```

```

{
    if (a[i] < b[j])
        c[k++] = a[i++];
    else
        c[k++] = b[j++];
}
while (i < n)
    c[k++] = a[i++];
while (j < m)
    c[k++] = b[j++];
}

```

可以看出合并有序数列的效率是比较高的，可以达到 $O(n)$ 。

解决了上面的合并有序数列问题，再来看归并排序，其的基本思路就是将数组分成二组 A，B，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序。如何让这二组组内数据有序了？

可以将 A，B 组各自再分成二组。依次类推，当分出来的小组只有一个数据时，可以认为这个小组组内已经达到了有序，然后再合并相邻的二个小组就可以了。

这样通过先递归的分解数列，再合并数列就完成了归并排序。

下面给出了代码。

```

bool MergeSort(int a[], int n)
{
    int *pTempArray = new int[n];
    if (p == NULL)
        return false;
}

```

```

        mergesort(a, 0, n - 1, pTempArray);

        return true;
    }

void mergesort(int a[], int first, int last, int temp[])
{
    if (first < last)
    {
        int mid = (first + last) / 2;

        mergesort(a, first, mid, temp); //左边有序
        mergesort(a, mid + 1, last, temp); //右边有序
        mergearray(a, first, mid, last, temp); //再将二个有序数列合并
    }
}

//将有二个有序数列 a[first...mid]和 a[mid...last]合并。
void mergearray(int a[], int first, int mid, int last, int temp[])
{
    int i = first, j = mid + 1;
    int m = mid, n = last;
    int k = 0;

    while (i <= m && j <= n)
    {
        if (a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }

    while (i <= m)
        temp[k++] = a[i++];

```

```

while (j <= n)
    temp[k++] = a[j++];

for (i = 0; i < k; i++)
    a[first + i] = temp[i];
}

```

归并排序的效率是比较高的，设数列长为 N ，将数列分开成小数列一共要 $\log N$ 步，每步都是一个合并有序数列的过程，时间复杂度可以记为 $O(N)$ ，故一共为 $O(N \cdot \log N)$ 。

因为归并排序每次都是在相邻的数据中进行操作，所以归并排序在 $O(N \cdot \log N)$ 的几种排序方法（快速排序，归并排序，希尔排序，堆排序）也是效率比较高的。

在本人电脑上对冒泡排序，直接插入排序，归并排序及直接使用系统的 `qsort()` 进行比较（均在 Release 版本下）

对 20000 个随机数据进行测试：

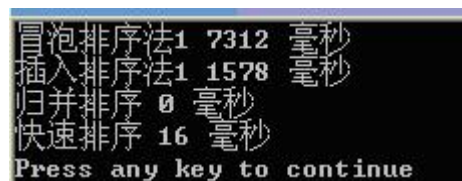


```

冒泡排序法 1 1235 毫秒
插入排序法 1 250 毫秒
归并排序 0 毫秒
快速排序 15 毫秒
Press any key to continue

```

对 50000 个随机数据进行测试：



```

冒泡排序法 1 7312 毫秒
插入排序法 1 1578 毫秒
归并排序 0 毫秒
快速排序 16 毫秒
Press any key to continue

```

再对 200000 个随机数据进行测试：

```
冒泡排序法1 122469 毫秒
插入排序法1 25891 毫秒
归并排序 62 毫秒
快速排序 78 毫秒
Press any key to continue
```

注：有的书上是在 `mergearray()` 合并有序数列时分配临时数组，但是过多的 `new` 操作会非常费时。因此作了下小小的变化。只在 `MergeSort()` 中 `new` 一个临时数组。后面的操作都共用这一个临时数组。

白话经典算法系列之六 快速排序 快速搞定

快速排序由于排序效率在同为 $O(N*\log N)$ 的几种排序方法中效率较高，因此经常被采用，再加上快速排序思想----分治法也确实实用，因此很多软件公司的笔试面试，包括像腾讯，微软等知名 IT 公司都喜欢考这个，还有大大小小的程序方面的考试如软考，考研中也常常出现快速排序的身影。

总的说来，要直接默写出快速排序还是有一定难度的，因为本人就自己的理解对快速排序作了下白话解释，希望对大家理解有帮助，达到快速排序，快速搞定。快速排序是 C.R.A.Hoare 于 1962 年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。

该方法的基本思想是：

1. 先从数列中取出一个数作为基准数。
2. 分区过程，将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。
3. 再对左右区间重复第二步，直到各区间只有一个数。

虽然快速排序称为分治法，但分治法这三个字显然无法很好的概括快速排序的全部步骤。因此我的对快速排序作了进一步的说明：**挖坑填数+分治法**：

先来看实例吧，定义下面再给出（最好能用自己的话来总结定义，这样对实现代码会有帮助）。

以一个数组作为示例，取区间第一个数为基准数。

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

72	6	57	88	60	42	83	73	48	85
----	---	----	----	----	----	----	----	----	----

初始时, $i = 0; j = 9; X = a[i] = 72$

由于已经将 $a[0]$ 中的数保存到 X 中, 可以理解成在数组 $a[0]$ 上挖了个坑, 可以将其它数据填充到这儿来。

从 j 开始向前找一个比 X 小或等于 X 的数。当 $j=8$, 符合条件, 将 $a[8]$ 挖出再填到上一个坑 $a[0]$ 中。 $a[0]=a[8]; i++$; 这样一个坑 $a[0]$ 就被搞定了, 但又形成了一个坑 $a[8]$, 这怎么办了? 简单, 再找数字来填 $a[8]$ 这个坑。这次从 i 开始向后找一个大于 X 的数, 当 $i=3$, 符合条件, 将 $a[3]$ 挖出再填到上一个坑中 $a[8]=a[3]; j--$;

数组变为:

0	1	2	3	4	5	6	7	8	9
48	6	57	88	60	42	83	73	88	85

$i = 3; j = 7; X = 72$

再重复上面的步骤, 先从后向前找, 再从前向后找。

从 j 开始向前找, 当 $j=5$, 符合条件, 将 $a[5]$ 挖出填到上一个坑中, $a[3] = a[5]; i++$;
从 i 开始向后找, 当 $i=5$ 时, 由于 $i=j$ 退出。

此时, $i = j = 5$, 而 $a[5]$ 刚好又是上次挖的坑, 因此将 X 填入 $a[5]$ 。

数组变为:

0	1	2	3	4	5	6	7	8	9
48	6	57	42	60	72	83	73	88	85

可以看出 $a[5]$ 前面的数字都小于它, $a[5]$ 后面的数字都大于它。因此再对 $a[0...4]$ 和 $a[6...9]$ 这二个区间重复上述步骤就可以了。

对挖坑填数进行总结

1. $i = L; j = R$; 将基准数挖出形成第一个坑 $a[i]$ 。
2. $j--$ 由后向前找比它小的数, 找到后挖出此数填前一个坑 $a[i]$ 中。
3. $i++$ 由前向后找比它大的数, 找到后也挖出此数填到前一个坑 $a[j]$ 中。
4. 再重复执行 2, 3 二步, 直到 $i=j$, 将基准数填入 $a[i]$ 中。

照着这个总结很容易实现挖坑填数的代码:

`int AdjustArray(int s[], int l, int r) //返回调整后基准数的位置`


```

{
    int i = l, j = r;
    int x = s[l]; //s[l]即 s[i]就是第一个坑
    while (i < j)
    {
        // 从右向左找小于 x 的数来填 s[i]
        while(i < j && s[j] >= x)
            j--;
        if(i < j)
        {
            s[i] = s[j]; //将 s[j]填到 s[i]中，s[j]就形成了一个新的坑
            i++;
        }
        // 从左向右找大于或等于 x 的数来填 s[j]
        while(i < j && s[i] < x)
            i++;
        if(i < j)
        {
            s[j] = s[i]; //将 s[i]填到 s[j]中，s[i]就形成了一个新的坑
            j--;
        }
    }
    //退出时，i 等于 j。将 x 填到这个坑中。
    s[i] = x;
    return i;
}

```

再写分治法的代码：

```

void quick_sort1(int s[], int l, int r)
{
    if (l < r)

```

```

{
    int i = AdjustArray(s, l, r); //先成挖坑填数法调整 s[]
    quick_sort1(s, l, i - 1); // 递归调用
    quick_sort1(s, i + 1, r);
}
}

```

这样的代码显然不够简洁，对其组合整理下：

//快速排序

```

void quick_sort(int s[], int l, int r)
{
    if (l < r)
    {
        //Swap(s[l], s[(l + r) / 2]); //将中间的这个数和第一个数交换 参见注 1
        int i = l, j = r, x = s[l];
        while (i < j)
        {
            while(i < j && s[j] >= x) // 从右向左找第一个小于 x 的数
                j--;
            if(i < j)
                s[i++] = s[j];

            while(i < j && s[i] < x) // 从左向右找第一个大于等于 x 的数
                i++;
            if(i < j)
                s[j--] = s[i];
        }
        s[i] = x;
        quick_sort(s, l, i - 1); // 递归调用
        quick_sort(s, i + 1, r);
    }
}

```

}

快速排序还有很多改进版本，如随机选择基准数，区间内数据较少时直接用另的方法排序以减小递归深度。有兴趣的筒子可以再深入的研究下。

注 1，有的书上是以中间的数作为基准数的，要实现这个方便非常方便，直接将中间的数和第一个数进行交换就可以了。

白话经典算法系列之七 堆与堆排序

堆排序与快速排序，归并排序一样都是时间复杂度为 $O(N \cdot \log N)$ 的几种常见排序方法。学习堆排序前，先讲解下什么是数据结构中的二叉堆。

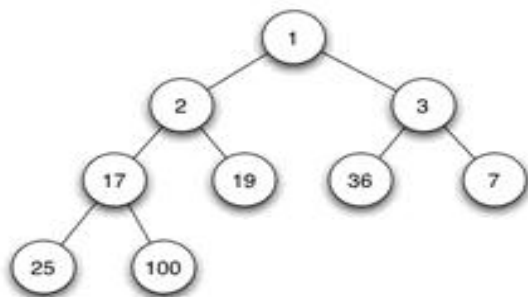
二叉堆的定义

二叉堆是完全二叉树或者是近似完全二叉树。

二叉堆满足二个特性：

1. 父结点的键值总是大于或等于（小于或等于）任何一个子节点的键值。
2. 每个结点的左子树和右子树都是一个二叉堆（都是最大堆或最小堆）。

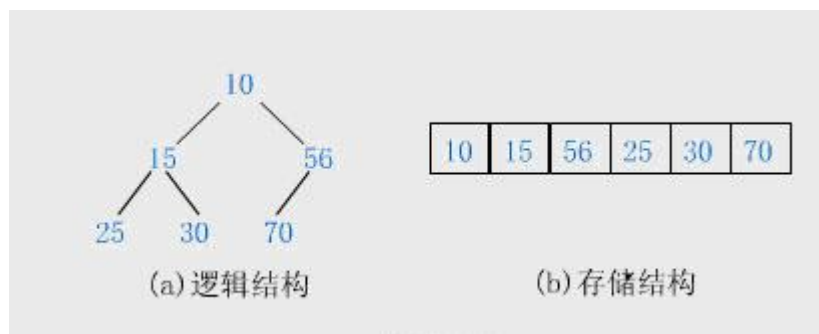
当父结点的键值总是大于或等于任何一个子节点的键值时为**最大堆**。当父结点的键值总是小于或等于任何一个子节点的键值时为**最小堆**。下图展示一个最小堆：



由于其它几种堆（二项式堆，斐波纳契堆等）用的较少，一般将二叉堆就简称为堆。

堆的存储

一般都用数组来表示堆， i 结点的父结点下标就为 $(i - 1) / 2$ 。它的左右子结点下标分别为 $2 * i + 1$ 和 $2 * i + 2$ 。如第 0 个结点左右子结点下标分别为 1 和 2。

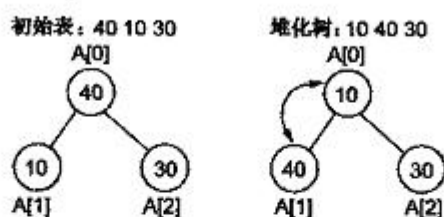


堆的操作——插入删除

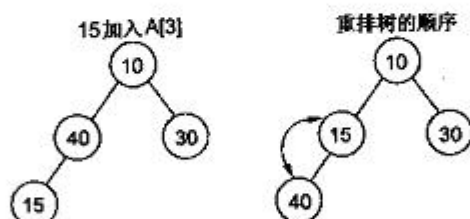
下面先给出《数据结构 C++语言描述》中最小堆的建立插入删除的图解，再给出本人的实现代码，最好是先看明白图后再去看代码。

1. 建立堆：数组具有对应的树表示形式。一般情况下，树并不满足堆的条件。通过重新排列元素，可以建立一棵“堆化”的树。

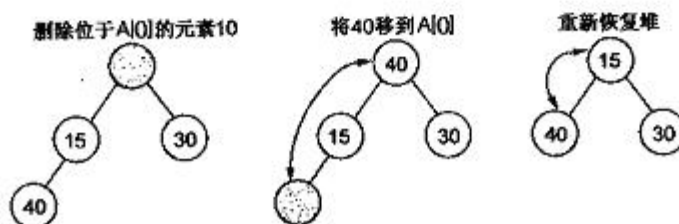
初始表: 40 10 30 堆化树: 10 40 30



2. 插入一个元素：新元素被加入到表层，随后树被更新以恢复堆次序。例如，下面的步骤将 15 加入到表中。



3. 删除一个元素：删除总是发生在根 A[0] 处。表中最后一个元素被用来填补空缺位置，结果树被更新以恢复堆条件。例如，以下步骤删除旧。



堆的插入

每次插入都是将新数据放在数组最后。可以发现从这个新数据的父结点到根结点必然为一个有序的数列，现在的任务是将这个新数据插入到这个有序数据中——这就类似于[直接插入排序](#)中将一个数据并入到有序区间中，对照《[白话经典算法系列之二 直接插入排序的三种实现](#)》不难写出插入一个新数据时堆的调整代码：

// 新加入 i 结点 其父结点为 $(i - 1) / 2$

```
void MinHeapFixup(int a[], int i)
{
    int j, temp;

    temp = a[i];
    j = (i - 1) / 2; //父结点
    while (j >= 0)
    {
        if (a[j] <= temp)
            break;

        a[i] = a[j]; //把较大的子结点往下移动,替换它的子结点
        i = j;
        j = (i - 1) / 2;
    }
    a[i] = temp;
}
```

更简短的表达为：

```
void MinHeapFixup(int a[], int i)
{
    for (int j = (i - 1) / 2; j >= 0 && a[i] > a[j]; i = j, j = (i - 1) / 2)
        Swap(a[i], a[j]);
}
```

插入时：

//在最小堆中加入新的数据 nNum

```
void MinHeapAddNumber(int a[], int n, int nNum)
{
    a[n] = nNum;
    MinHeapFixup(a, n);
}
```

堆的删除

按定义，堆中每次都只能删除第 0 个数据。为了便于重建堆，实际的操作是将最后一个数据的值赋给根结点，然后再从根结点开始进行一次从上向下的调整。调整时先在左右儿子结点中找最小的，如果父结点比这个最小的子结点还小说明不

需要调整了，反之将父结点和它交换后再考虑后面的结点。相当于从根结点将一个数据的“下沉”过程。下面给出代码：

// 从i节点开始调整,n为节点总数 从0开始计算 i节点的子节点为 $2*i+1, 2*i+2$

```
void MinHeapFixdown(int a[], int i, int n)
{
    int j, temp;
    temp = a[i];
    j = 2 * i + 1;
    while (j < n)
    {
        if (j + 1 < n && a[j + 1] < a[j]) //在左右孩子中找最小的
            j++;
        if (a[j] >= temp)
            break;
        a[i] = a[j]; //把较小的子结点往上移动,替换它的父结点
        i = j;
        j = 2 * i + 1;
    }
    a[i] = temp;
}
```

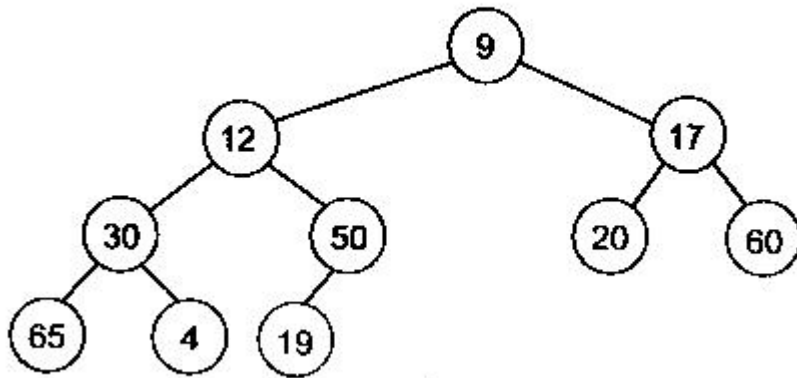
//在最小堆中删除数

```
void MinHeapDeleteNumber(int a[], int n)
{
    Swap(a[0], a[n - 1]);
    MinHeapFixdown(a, 0, n - 1);
}
```

堆化数组

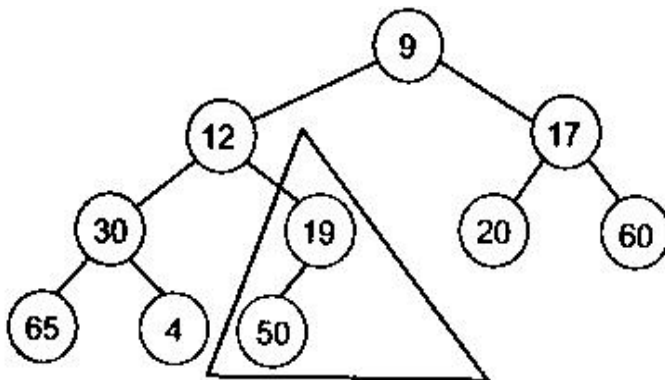
有了堆的插入和删除后，再考虑下如何对一个数据进行堆化操作。要一个一个的从数组中取出数据来建立堆吧，不用！先看一个数组，如下图：

```
int A[0] = {9, 12, 17, 30, 50, 20, 60, 65, 4, 49};
```

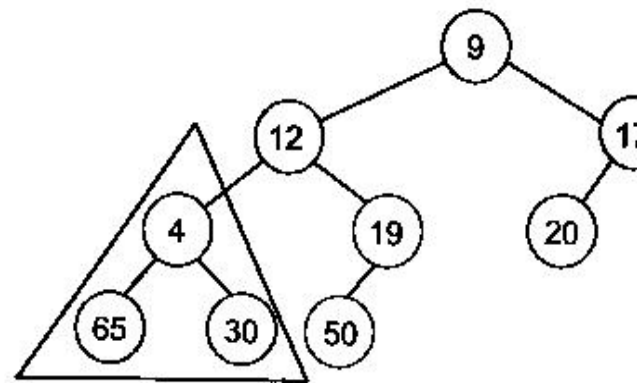


初始表

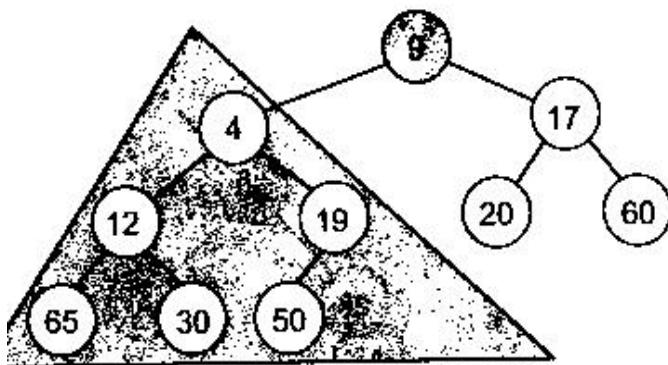
很明显，对叶子结点来说，可以认为它已经是一个合法的堆了即 20, 60, 65, 4, 49 都分别是一个合法的堆。只要从 $A[4]=50$ 开始向下调整就可以了。然后再取 $A[3]=30$, $A[2] = 17$, $A[1] = 12$, $A[0] = 9$ 分别作一次向下调整操作就可以了。下图展示了这些步骤：



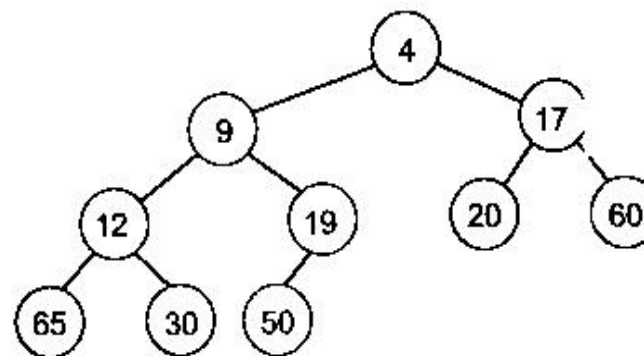
(A) FilterDown(4) 放置 50



(B) FilterDown(3) 放置 30



(C)



(D)

写出堆化数组的代码：

//建立最小堆

```
void MakeMinHeap(int a[], int n)
```

```
{
```

```
    for (int i = n / 2 - 1; i >= 0; i--)
```

```
        MinHeapFixdown(a, i, n);  
    }
```

至此，堆的操作就全部完成了(注 1)，再来看下如何用堆这种数据结构来进行排序。

堆排序

首先可以看到堆建好之后堆中第 0 个数据是堆中最小的数据。取出这个数据再执行下堆的删除操作。这样堆中第 0 个数据又是堆中最小的数据，重复上述步骤直至堆中只有一个数据时就直接取出这个数据。

由于堆也是用数组模拟的，故堆化数组后，第一次将 A[0]与 A[n - 1]交换，再对 A[0...n-2]重新恢复堆。第二次将 A[0]与 A[n - 2]交换，再对 A[0...n - 3]重新恢复堆，重复这样的操作直到 A[0]与 A[1]交换。由于每次都是将最小的数据并入到后面的有序区间，故操作完成后整个数组就有序了。有点类似于[直接选择排序](#)。

```
void MinheapsortTodescendarray(int a[], int n)  
{  
    for (int i = n - 1; i >= 1; i--)  
    {  
        Swap(a[i], a[0]);  
        MinHeapFixdown(a, 0, i);  
    }  
}
```

注意使用最小堆排序后是递减数组，要得到递增数组，可以使用最大堆。

由于每次重新恢复堆的时间复杂度为 $O(\log N)$ ，共 $N - 1$ 次重新恢复堆操作，再加上前面建立堆时 $N / 2$ 次向下调整，每次调整时间复杂度也为 $O(\log N)$ 。二次操作时间相加还是 $O(N * \log N)$ 。故堆排序的时间复杂度为 $O(N * \log N)$ 。

注 1 作为一个数据结构，最好用类将其数据和方法封装起来，这样即便于操作，也便于理解。此外，除了堆排序要使用堆，另外还有很多场合可以使用堆来方便和高效的处理数据，以后会一一介绍。

[转载 请 标 明 出 处 ， 原 文 地 址 ：](#)

<http://www.cnblogs.com/morewindows/archive/2011/08/22/2149612.html>