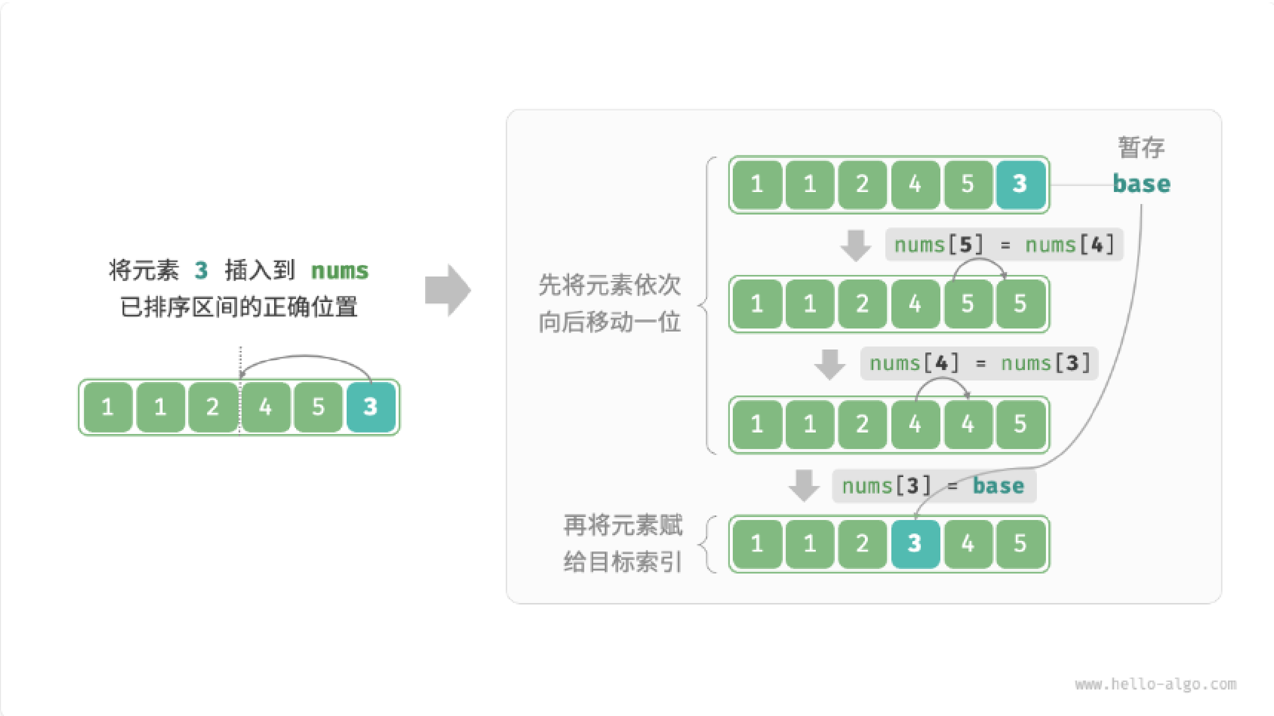
# 排序

- ==内部排序== 在排序期间元素全部存放在内存中的数据
- ==外部排序== 根据要求在内外存之间移动

# 插入排序

## 直接插入排序（稳定）




### 时间复杂度

最好 ： $O(n)$
最坏：$O(n^2)$
平均 ： $O(n^2)$

### 代码

```
void InserSort(ElemType A[ ],int n)//A[0]为哨兵
{
        int i,j;
        for(i=2;i<=n;i++){
                if(A[i]<A[i-1]){
                        A[0]=A[i];
                        for(j=i-1;A[0]<A[j];--j)
                        {
```

```
                    A[j+1]=A[j]
                }
                A[j+1]=A[0];
            }
        }
}
```

## 折半插入排序(稳定)

### 时间复杂度

$O(n^2)$

### 代码

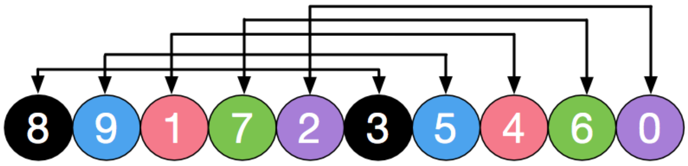
仅改变查找为折半查找

## 希尔排序（不稳定）

把待排序表分成步长相等的子表，先排子表，再排总表

原始数组 以下数据元素颜色相同为一组

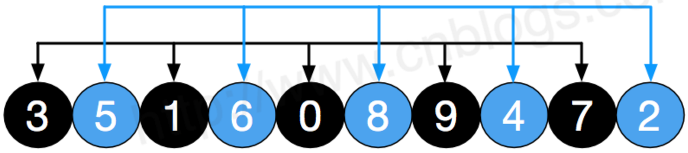

初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]




对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，

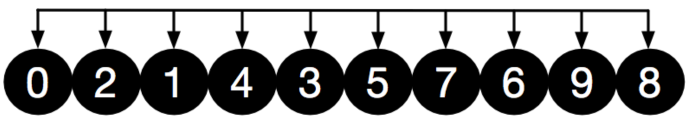
然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]






对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。

再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下




经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。


## 时间复杂度

$O(n^2)$

## 代码

```
void ShellSort(ElemType A[ ],int n)
{
        int dk,i,j;
        for(dk=n/2;dk>=1;dk=dk/2){
                for(i=dk+1;i<=n;++i){
                        if(A[i]<A[i-dk]){
                                A[0]=A[i];
```

```
                        for(j=i-dk;j>0&&A[0]<A[j];j-=dk)
                            A[j+dk]=A[j];
                        A[j+dk] = A[0];
                    }
                }
            }
}
```

## 交换排序 💱

### 冒泡排序 🫧 （稳定）


#### 时间复杂度

$O(n^2)$

#### 代码

```
void BUbbleSort(ElemType A[ ] int n)
{
    for(int i=0;i<n-1;i++)
    {
        bool flag=false;
        for(int j=n-1;j>i;j++){
            if(A[j-1]>A[j]){
                swap(A[j-1],A[j]);
                flag==true;
```

```
                }
                if(flag==false)
                    return ;
            }
        }
}
```

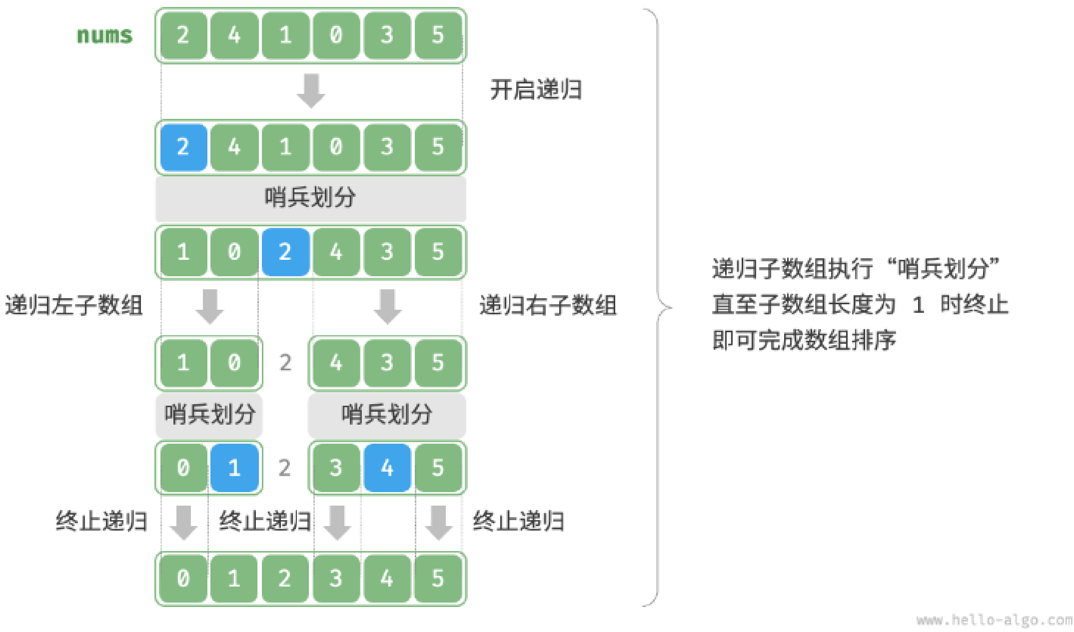
## 快速排序 （不稳定）

1. 选取数组最左端元素作为基准数，初始化两个指针 `i` 和 `j` 分别指向数组的两端。
2. 设置一个循环，在每轮中使用 `i` （`j`） 分别寻找第一个比基准数大（小）的元素，然后交换这两个元素。
3. 循环执行步骤 `2.` ，直到 `i` 和 `j` 相遇时停止，最后将基准数交换至两个子数组的分界线。

核心：$\text{左子数组} \le \text{基准数} \le \text{有子数组}$

### 算法流程

1. 首先，对原数组执行一次“哨兵划分”，得到未排序的左子数组和右子数组。
2. 然后，对左子数组和右子数组分别递归执行“哨兵划分”。
3. 持续递归，直至子数组长度为 1 时终止，从而完成整个数组的排序。




### 代码

```
void QuickSort(ElemType A[ ],int low,int high)
{
```

```
        if(low<high){
                int pivotpos = paration(A,low,high);
                QuickSort(A,low,pivotpos-1);
                QuickSort(A,pivotpos+1,high);
        }
}
int pivotpos(ElemType A[ ],int low,int high)
{
        Elemtype pivot = A[low];
        while(low<high)
        {
                while(low<high&&A[high]>=pivot) --high;
                        A[low]=A[high];
                while(low<high&&A[low]<=pivot) ++low;
                        A[high]=A[low];
        }
        A[low]=pivot;
        return low;
}
```

#### 时间复杂度

内部排序算法中平均性能最优的算法
时间复杂度 $O(nlog_2^n)$
空间复杂度 $O(n)$
越有序排的越慢
每个基准元素一趟后都在最终地位置

## 选择排序（不稳定）

### 简单选择排序

从后面依次选择

#### 时间复杂度

$O(n^2)$

#### 代码

```
void SelectSort(ElemType A[ ],int n)
{
        for(int i=0;i<n-1;i++)
```

```
    {
        int min=i;
        if(int j=i+1;j<n;j++)
            if(A[j]<A[min]) min=j;
        if(min !=i) swap(A[i],A[min]);
    }
}
```

## 堆排序(不稳定)

1. 输入数组并建立大顶堆。完成后，最大元素位于堆顶。
2. 将堆顶元素（第一个元素）与堆底元素（最后一个元素）交换。完成交换后，堆的长度减 1 ， 已排序元素数量加 1 。
3. 从堆顶元素开始，从顶到底执行堆化操作（sift down）。完成堆化后，堆的性质得到修复。
4. 循环执行第 `2.` 步和第 `3.` 步。循环 n-1 轮后，即可完成数组排序

### 建堆过程

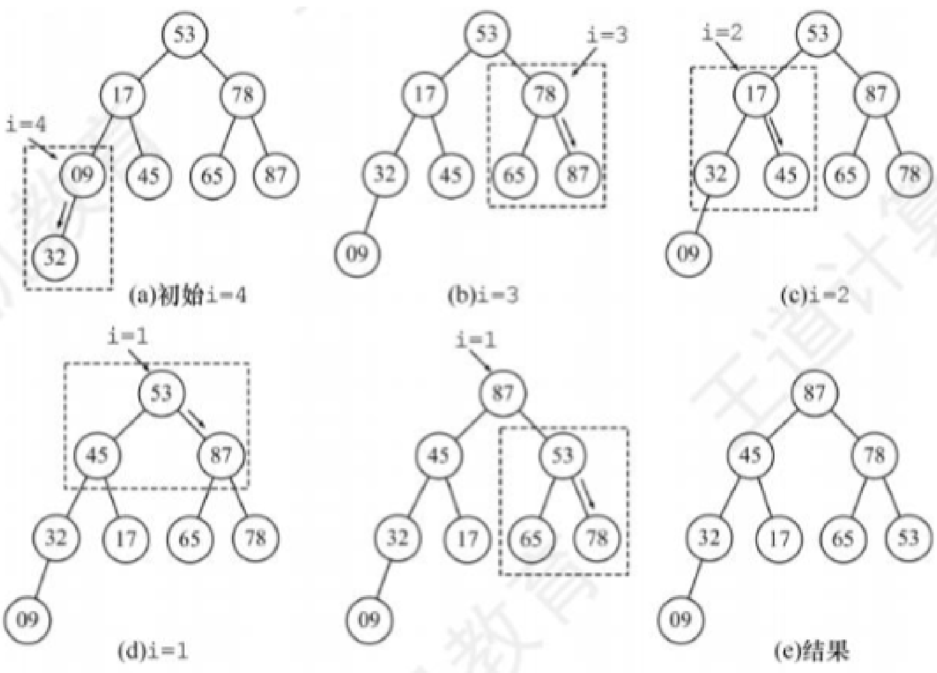



图 8.6　自下往上逐步调整为大根堆

### 堆的插入操作及比较次数分析

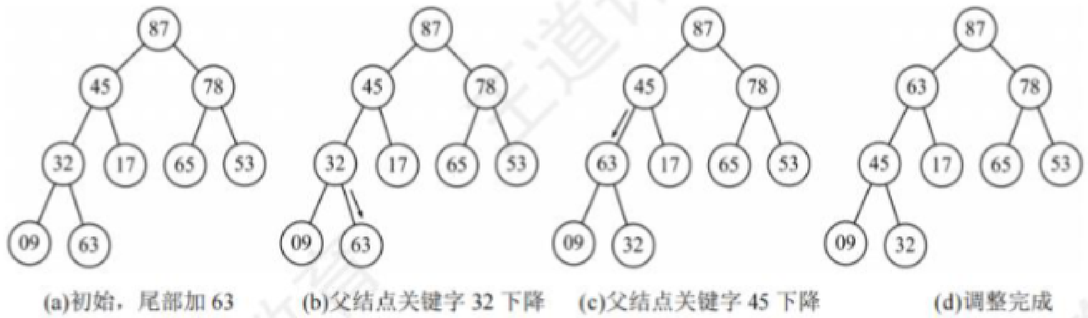



图 8.8 大根堆的插入操作示例

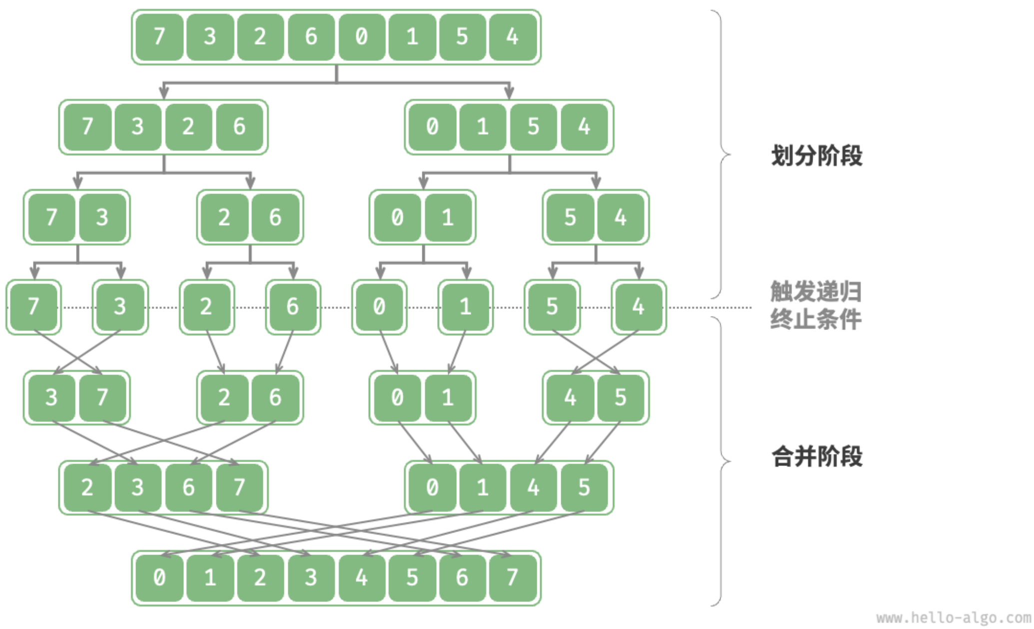
### 时间复杂度

$O(nlog_2^n)$

## 归并排序，基数排序，计数排序

### 归并排序(稳定)

1. **划分阶段**：通过递归不断地将数组从中点处分开，将长数组的排序问题转换为短数组的排序问题。
2. **合并阶段**：当子数组长度为 1 时终止划分，开始合并，持续地将左右两个较短的有序数组合并为一个较长的有序数组，直至结束。




### 时间复杂度

$O(nlog_2^n)$

### 算法

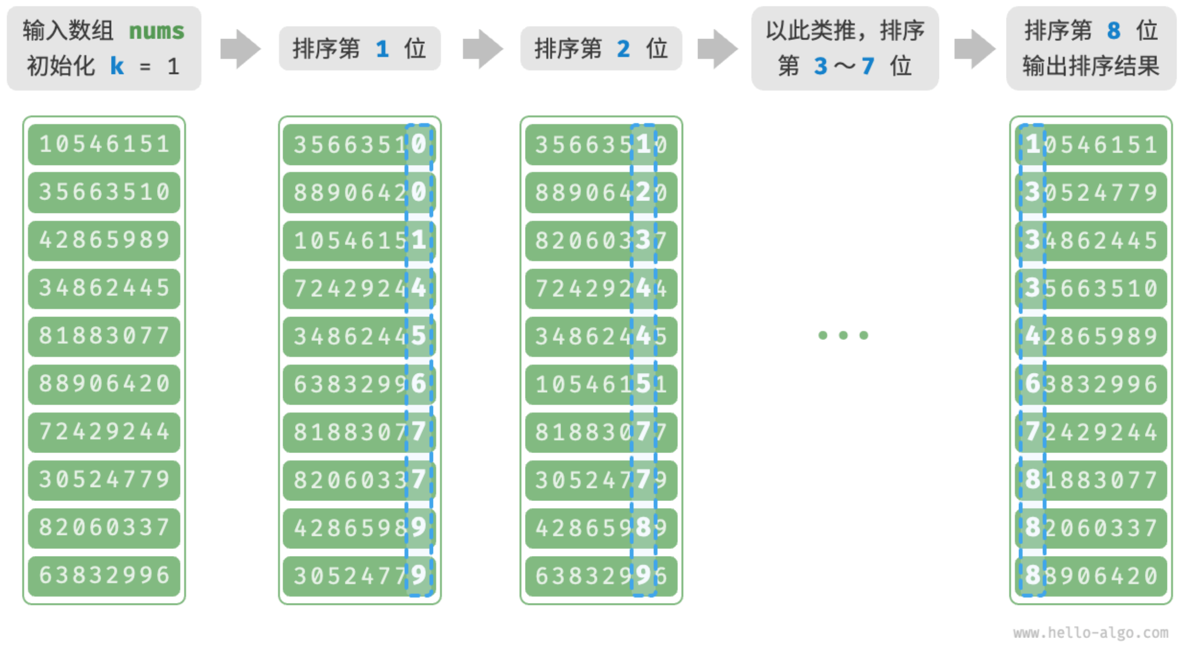
```
ElemType *B = (ElemType *)malloc((n+1)*sizeof(ElemType));
void Merge(ElemType A[ ],int low,int mid,int n)
{
        int i,j,k;
        for(k=low;k<=high;k++)
                B[k]=A[k];
        for(i=low,j=mid+1,k=i;i<=mid&&j<=high;k++)
        {       if(B[i]<=B[j])
                        A[k]=B[i++];
                else
                        A[k] = B[j++];
        }
        while(i<mid) A[k++]=B[i++];
        while(j<mid) A[k++]=B[j++];
}
void MergeSort(ElemType A[ ],int low,int high)
{
        if(low<high)
        {
                int mid=(low+high)/2;
                MergeSort(A,low,mid);
                MergeSort(A,mid+1,high);
                Merge(A,low,mid,high);
        }
}
```

### 基数排序（稳定）

将整数按位数切割成不同的数字，然后按每个位数分别比较

1. 初始化位数 k=1 。
2. 对学号的第 k 位执行“计数排序”。完成后，数据会根据第 k 位从小到大排序。

3. 将 k 增加 1 ，然后返回步骤 `2.` 继续迭代，直到所有位都排序完成后结束。

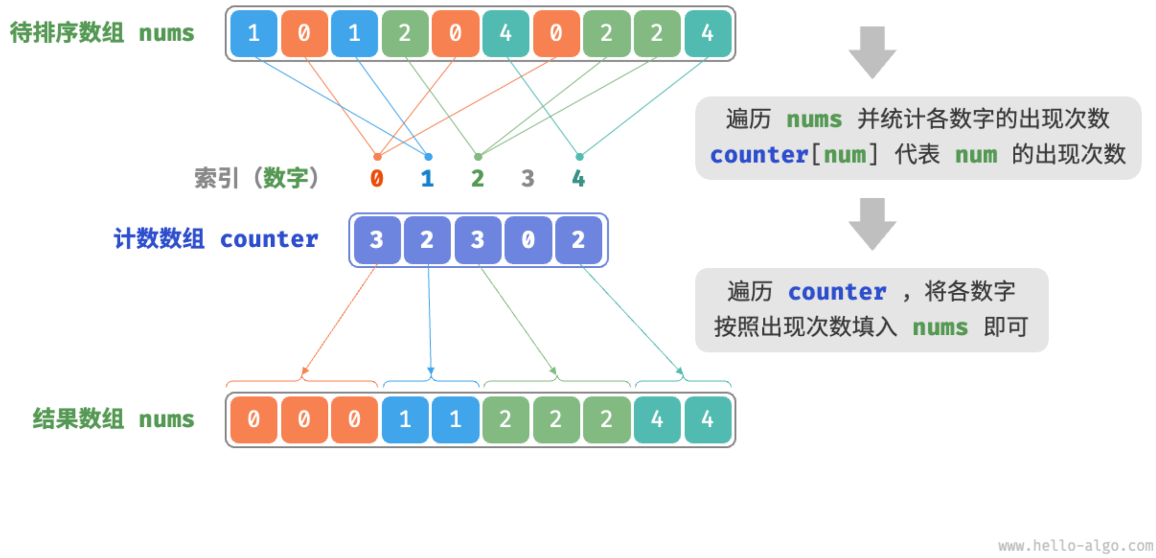



### 时间复杂度

$O(n * k)$
建立的队列数等于进制数

### 计数排序（了解思想即可）

1. 遍历数组，找出其中的最大数字，记为 m ，然后创建一个长度为 m+1 的辅助数组 `counter` 。
2. **借助 `counter` 统计 `nums` 中各数字的出现次数**，其中 `counter[num]` 对应数字 `num` 的出现次数。统计方法很简单，只需遍历 `nums`（设当前数字为 `num`），每轮将 `counter[num]` 增加 1 即可。
3. **由于 `counter` 的各个索引天然有序，因此相当于所有数字已经排序好了**。接下来，我们遍历 `counter` ，根据各数字出现次数从小到大的顺序填入 `nums` 即可。




## 各种内部排序算法的比较和应用

表 8.1 各种排序算法的性质

| 算法种类 | 时间复杂度 | | | 空间复杂度 | 是否稳定 |
|---|---|---|---|---|---|
| | 最好情况 | 平均情况 | 最坏情况 | | |
| 直接插入排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 是 |
| 冒泡排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 是 |
| 简单选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 否 |
| 希尔排序 | | | | $O(1)$ | 否 |
| 快速排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n^2)$ | $O(\log_2 n)$ | 否 |
| 堆排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(1)$ | 否 |
| 二路归并排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n)$ | 是 |
| 基数排序 | $O(d(n+r))$ | $O(d(n+r))$ | $O(d(n+r))$ | $O(r)$ | 是 |

## 外部排序

排序大文件，仅部分能进入内存

### 外部排序的方法

#### 归并排序算法

- 按照内存大小，将大文件分成若干长度为 l 的子文件（l 应小于内存的可使用容量）
- 然后将各个子文件依次读入内存，使用适当的内部排序算法对其进行排序(排好序的子文件统称为“归并段”或者“顺段”)

- 将排好序的归并段重新写入外存，为下一个子文件排序腾出内存空间
- 对得到的顺段进行合并，直至得到整个有序的文件为止

$$外部排序总时间 = 内部排序时间 + 外存信息读/写时间 + 内部归并时间$$

## 多路平衡归并与败者树

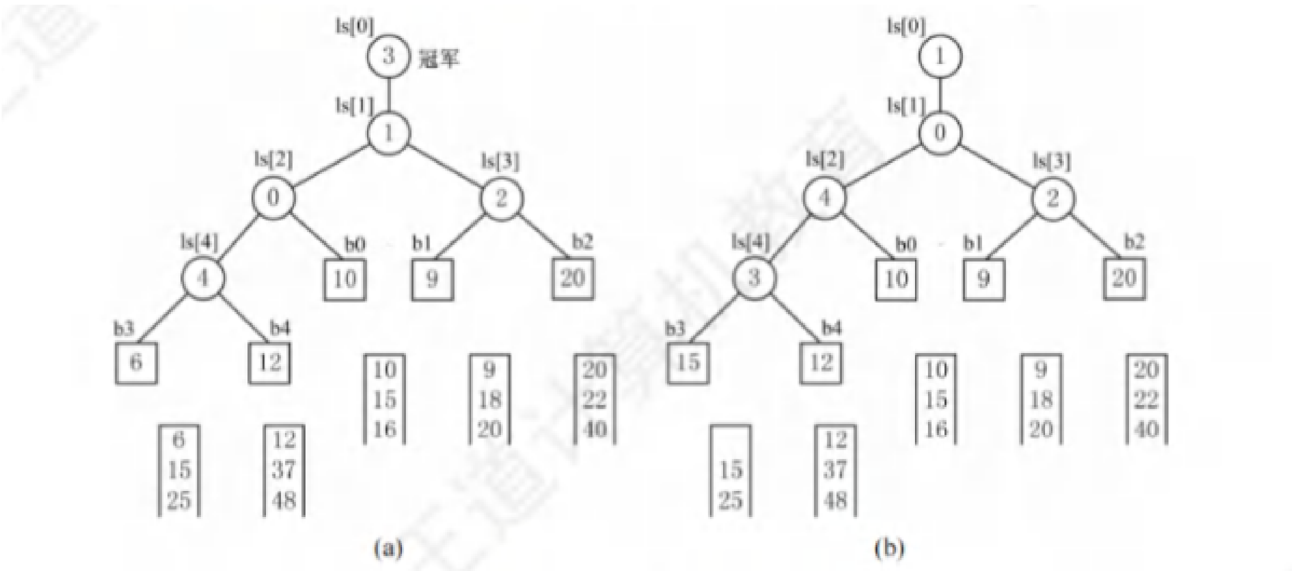



图 8.17　实现 5 路归并的败者树

## 置换-选择排序（生成初始归并段）

表 8.2　置换-选择排序过程示例

| 输出文件 FO | 工作区 WA | 输入文件 FI |
|---|---|---|
| — | — | 17, 21, 05, 44, 10, 12, 56, 32, 29 |
| — | 17 21 $\boxed{05}$ | 44, 10, 12, 56, 32, 29 |
| 05 | $\boxed{17}$ 21 44 | 10, 12, 56, 32, 29 |
| 05 17 | 10 $\boxed{21}$ 44 | 12, 56, 32, 29 |
| 05 17 21 | 10 12 $\boxed{44}$ | 56, 32, 29 |
| 05 17 21 44 | 10 12 $\boxed{56}$ | 32, 29 |
| 05 17 21 44 56 | 10 12 32 | 29 |
| 05 17 21 44 56 # | $\boxed{10}$ 12 32 | 29 |
| 10 | 29 $\boxed{12}$ 32 | — |
| 10 12 | $\boxed{29}$ 32 | — |
| 10 12 29 | $\boxed{32}$ | — |
| 10 12 29 32 | — | — |
| 10 12 29 32 # | — | — |

## 最佳归并树

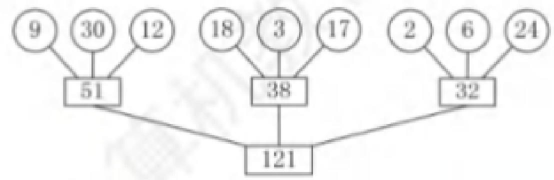



图 8.18　3 路平衡归并的归并树

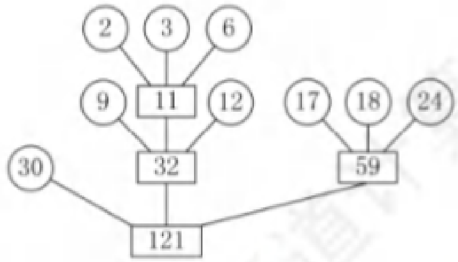



图 8.19　3 路平衡归并的最佳归并树

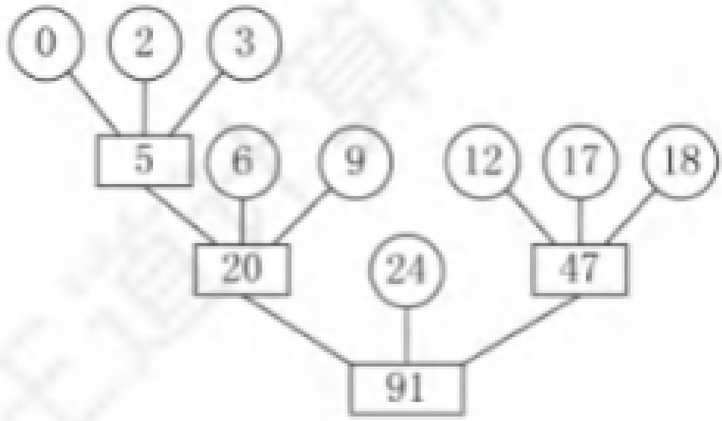



图 8.20　8 个归并段的最佳归并树

判定添加虚段的数目：

设度为 0 的结点有 $n_0$ 个，度为 $k$ 的结点有 $n_k$ 个，归并树的结点总数为 $n$，则有：

- $n = n_k + n_0$　（总结点数 = 度为 $k$ 的结点数 + 度为 0 的结点数）
- $n = kn_k + 1$　（总结点数 = 所有结点的度数之和 + 1）

因此，对严格 $k$ 叉树有 $n_0 = (k-1)n_k + 1$，由此可得 $n_k = (n_0 - 1)/(k-1)$。

- 若 $(n_0 - 1)\%(k-1) = 0$（%为取余运算），则说明这 $n_0$ 个叶结点（初始归并段）正好可以构造 $k$ 叉归并树。此时，内结点有 $n_k$ 个。
- 若 $(n_0 - 1)\%(k-1) = u \neq 0$，则说明对于这 $n_0$ 个叶结点，其中有 $u$ 个多余，不能包含在 $k$ 叉归并树中。为构造包含所有 $n_0$ 个初始归并段的 $k$ 叉归并树，应在原有 $n_k$ 个内结点的基础上再增加 1 个内结点。它在归并树中代替了一个叶结点的位置，被代替的叶结点加上刚才多出的 $u$ 个叶结点，即再加上 $k-u-1$ 个空归并段，就可以建立归并树。