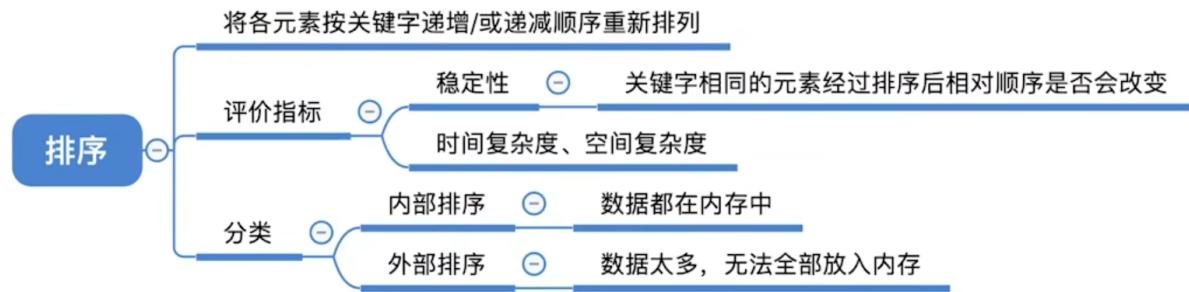


# 排序



排序 (Sort) , 就是重新排列表中的元素, 使表中的元素满足按关键字有序的过程。

输入 : $n$ 个记录 $R_1, R_2, \dots, R_n$ , 对应的关键字为 $k_1, k_2, \dots, k_n$

输出 :输入序列的一个重排 $R'_1, R'_2, \dots, R'_n$ , 使得有 $k'_1 \leq k'_2 \leq \dots \leq k'_n$ (也可递减)

## 排序算法的评价指标

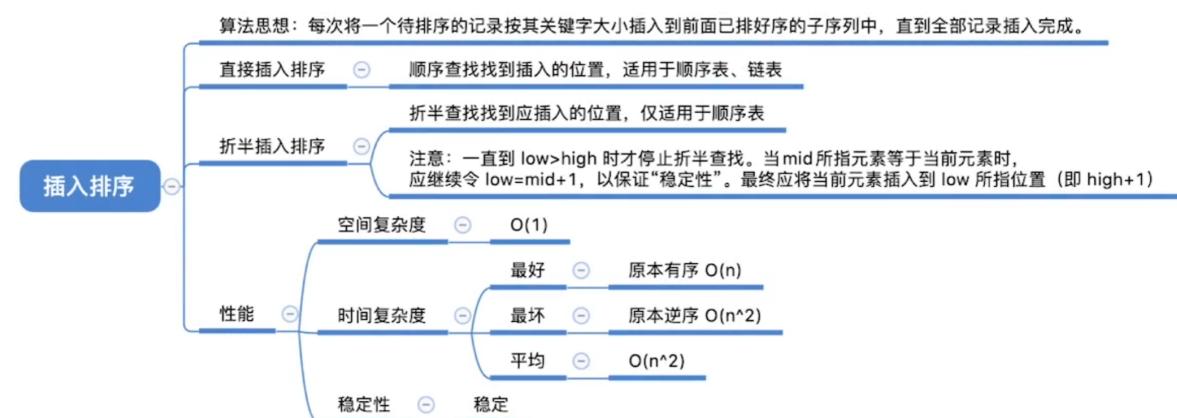
- 时间复杂度
- 空间复杂度
- 稳定性

若待排序表中有两个元素 $R_i$ 和 $R_j$ , 其对应的关键字相同即 $\text{key}_i = \text{key}_j$ , 且在排序前 $R_i$ 在 $R_j$ 的前面, 若使用某一排序算法排序后,  $R_i$ 仍然在 $R_j$ 的前面, 则称这个排序算法是稳定的, 否则称排序算法是不稳定的。

## 排序算法的分类

- 内部排序
  - 数据都在内存中。需关注如何使算法时、空复杂度更低。
- 外部排序
  - 数据太多，无法全部放入内存。还需关注如何使读/写磁盘次数更少。

# 插入排序



算法思想: 每次将一个待排序的记录按其关键字大小插入到已排好序的子序列中, 直到全部记录插入完成。

# 算法实现

```
//直接插入排序
void InsertSort(int A[], int n)
{
    int i,j,temp;
    for(i=1;i<n;i++)//将各元素插入已排好序的序列中
        if(A[i]<A[i-1])//若A[i]关键字小于前驱
    {
        temp=A[i];//用temp暂存A[i]
        for(j=i-1;j>=0 && A[j]>temp;--j)//检查所有前面已排好序的元素
            A[j+1]=A[j];//所有大于temp的元素都向后挪位
        A[j+1]=temp;//复制到插入位置
    }
}

//直接插入排序（带哨兵）
void InsertSort(int A[], int n)
{
    int i,j;
    for(i=2;i<=n;i++)//依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1])//若A[i]关键码小于其前驱，将A[i]插入有序表
    {
        A[0]=A[i];//复制为哨兵，A[0]不存放元素
        for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
            A[j+1]=A[j];//向后挪位
        A[j+1]=A[0];//复制到插入位置
    }
}
```

## 算法效率分析

空间复杂度：O(1)

时间复杂度：主要来自对比关键字、移动元素若有n各元素，则需要n-1趟处理

最好情况：原本就有序。共n-1趟处理，每一趟只需要对比关键字1次，不用移动元素

最好时间复杂度O(n)

最坏情况：原本就逆序。

第1趟：对比关键字2次，移动元素3次

第2趟：对比关键字3次，移动元素4次

第*i*趟：对比关键字*i* + 1次，移动元素*i* + 2次

...

第*n* - 1趟：对比关键字*n*次，移动元素*n* + 1次

空间复杂度：O(1)

最好时间复杂度（全部有序）：O(n)

最坏时间复杂度（全部逆序）：O( $n^2$ )

平均时间复杂度：O( $n^2$ )

算法稳定性：稳定

# 优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素

当 $low > high$ 时折半查找停止，应将 $[low, i-1]$ 内的元素全部右移，并将 $A[0]$ 复制到 $low$ 所指位置

当 $A[mid] == A[0]$ 时，为了保证算法的“稳定性”，应继续在 $mid$ 所指位置右边寻找插入位置

```
//折半插入排序
void InsertSort(int A[], int n)
{
    int i, j, low, high, mid;
    for(i=2; i<=n; i++) //依次将A[2]~A[n]插入前面的已排序序列
    {
        A[0]=A[i]; //将A[i]暂存到A[0]
        low=1; high=i-1; //设置折半查找的范围
        while(low<=high) //折半查找（默认递增有序）
        {
            mid=(low+high)/2; //取中间点
            if(A[mid]>A[0]) high=mid-1; //查找左半子表
            else low=mid+1; //查找右半子表
        }
        for(j=i-1; j>=high+1; --j)
            A[j+1]=A[j]; //统一后移元素，空出插入位置
        A[high+1]=A[0]; //插入操作
    }
}
```

比起“直接插入排序”，比较关键字的次数减少了，但是移动元素的次数没变，整体来看时间复杂度依然是 $O(n^2)$

## 对链表进行插入排序

移动元素的次数变少了，但是关键字对比的次数依然是 $O(n^2)$ 数量级，整体来看时间复杂度依然是 $O(n^2)$

## 希尔排序 (Shell Sort)

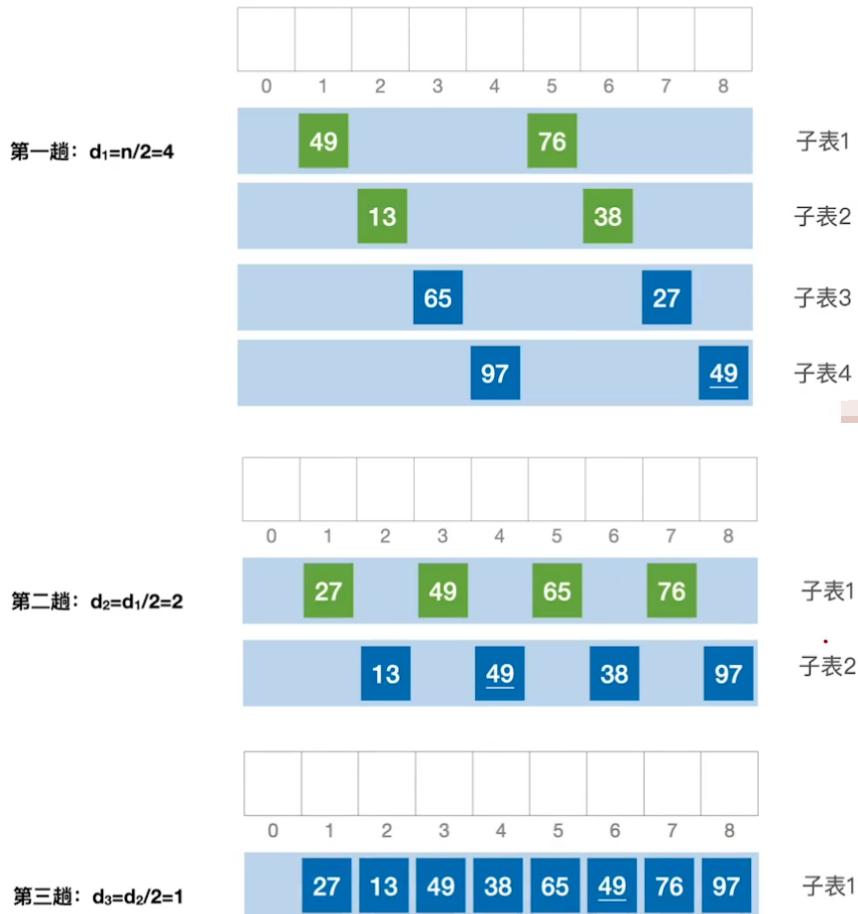
先将待排序表分割成若干形如 $L[i, i + d, i + 2d, \dots, i + kd]$ 的“特殊”子表，对各个子表分别进行直接插入排序。缩小增量 $d$ ，重复上述过程，直到 $d=1$ 为止。

希尔排序



希尔排序：先追求表中元素部分有序，再逐渐逼近全局有序

希尔排序：先将待排序表分割成若干形如 $L[i, i + d, i + 2d, \dots, i + kd]$ 的特殊子表  
对各个子表分别进行直接插入排序，  
缩小增量 $d$ ，重复上述过程，直到 $d = 1$ 为止



整个表已呈现出“基本有序”，对整体再进行一次“直接插入排序”

希尔本人建议：每次将增量缩小一半

## 算法实现

```
//希尔排序
void ShellSort(int A[], int n)
{
    int d, i, j;
    //A[0]只是暂存单元，不是哨兵，当j<=0时，插入位置已到
    for(d=n/2; d>=1; d=d/2)//步长变化
        for(i=d+1; i<=n; ++i)
            if(A[i]<A[i-d])//需将A[i]插入有序增量子表
            {
                A[0]=A[i];//暂存在A[0]
                for(j=i-d; j>0&&A[0]<A[j]; j-=d)
                    A[j+d]=A[j];//记录后移，查找插入的位置
                A[j+d]=A[0];//插入
            }
}
```

## 算法性能分析

空间复杂度: O(1)

时间复杂度: 和增量序列  $d_1, d_2, d_3 \dots$  的选择有关，目前无法用数学手段证明确切的时间复杂度

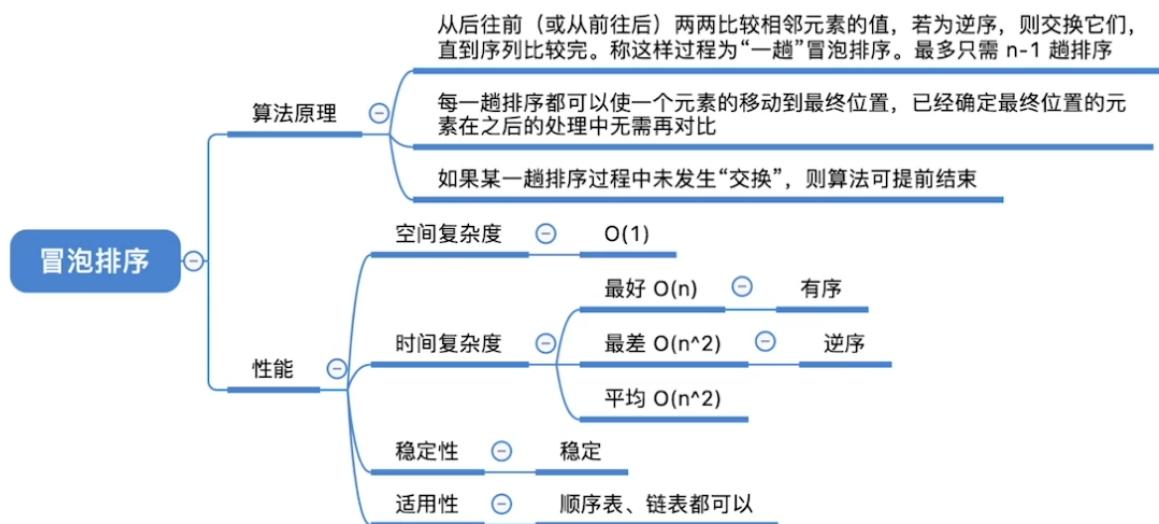
最坏时间复杂度为  $O(n^2)$ ，当  $n$  在某个范围内时，可达  $O(n^{1.3})$

稳定性：不稳定

适用性：仅适用于顺序表，不适用于链表

# 交换排序

## 冒泡排序



基于“交换”的排序：根据序列中两个元素关键字的比较结果来对换这两个记录在序列中的位置

从后往前（或从前往后）两两比较相邻元素的值，若为逆序（即  $A[i-1] > A[i]$ ），则交换它们，直到序列比较完。称这样过程为“一趟”冒泡排序。

第一趟排序使关键字值最小的一个元素“冒”到最前面

前边已经确定最终位置的元素不用再对比

第二趟结束后，最小的两个元素会“冒”到最前边

若某一趟排序没有发生“交换”，说明此时已经整体有序。

## 算法实现

```
//交换
void swap(int &a, int &b)
{
    int temp=a;
    a=b;
    b=temp;
}

//冒泡排序
void bubblesort(int A[], int n)
{
    for(int i=0;i<n-1;i++)
    {
        bool flag=false;//表示本趟冒泡是否发生交换的标志
        for(int j=n-1;j>1;j--)//一趟冒泡过程
            if(A[j-1]>A[j])//若为逆序
            {
                swap(A[j-1],A[j]);//交换
                flag=true;
            }
    }
}
```

```

    }
    if(flag==false)
        return;//本趟遍历后没有发生交换，说明表已经有序
}
}

```

*i*所指位置之前的元素都已“有序”

只有A[j-1]>A[j]时才交换，因此算法是稳定的

## 算法性能分析

空间复杂度：O(1)

最好情况（有序）：比较次数=n-1，交换次数=0，最好时间复杂度=O(n)

最坏情况（逆序）：

$$\text{比较次数} = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} = \text{交换次数}$$

$$\text{最坏时间复杂度} = O(n^2)$$

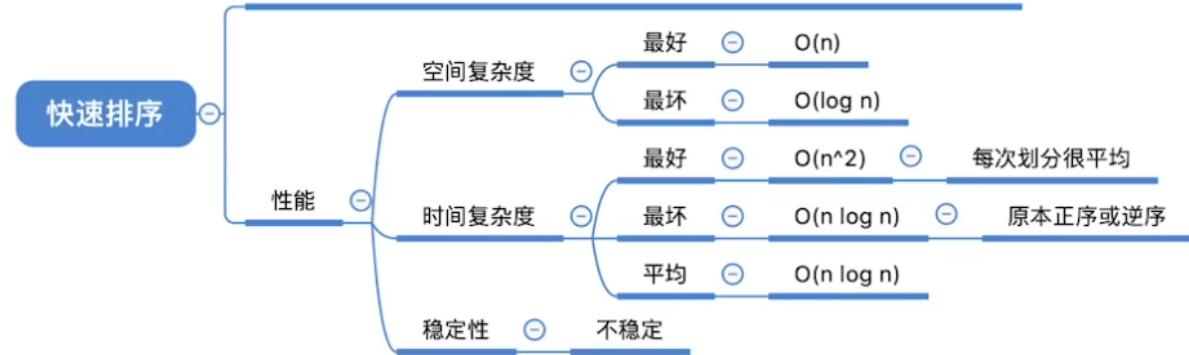
每次交换都需要移动元素3次

## 冒泡排序是否适用于链表？

可从前往后“冒泡”，每一趟将更大的元素“冒”到链尾

## 快速排序

算法表现主要取决于递归深度，若每次“划分”越均匀，则递归深度越低。  
“划分”越不均匀，递归深度越深



基于“交换”的排序：根据序列中两个元素关键字的比较结果来对换这两个记录在序列中的位置

算法思想：在待排序表L[1...n]中任取一个元素pivot作为枢轴（或基准，通常取首元素），通过一趟排序将待排序表划分为独立的两部分L[1...k-1]和L[k+1...n]，使得L[1...k-1]中的所有元素小于pivot，L[k+1...n]中的所有元素大于等于pivot，则pivot放在了其最终位置L(k)上，这个过程称为一次“划分”。然后分别递归地对两个子表重复上述过程，直至每部分内只有一个元素或空位置，即所有元素放在了其最终位置上。

更小的元素都交换到左边

更大的元素都交换到右边

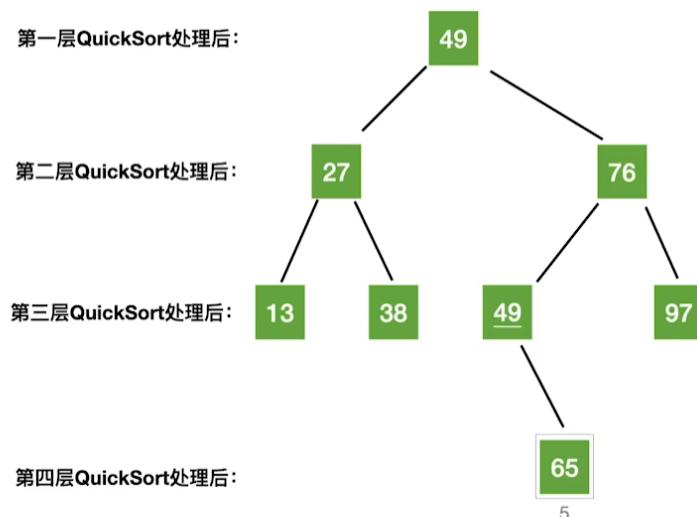


## 算法实现

```
//用第一个元素将待排序序列划分成左右两个部分
int Partition(int A[], int low, int high)
{
    int pivot=A[low];//第一个元素作为枢轴
    while(low<high)//用low,high搜索枢轴的最终位置
    {
        while(low<high&&A[high]>=pivot)--high;
        A[low]=A[high];//比枢轴小的元素移动到左端
        while(low<high&&A[low]<=pivot)++low;
        A[high]=A[low];//比枢轴大的元素移动到右端
    }
    A[low]=pivot;
    return low;
}
//快速排序
void QuickSort(int A[], int low, int high)
{
    if(low<high)//递归跳出的条件
    {
        int pivotpos=Partition(A,low,high);//划分
        QuickSort(A,low,pivotpos-1);//划分左子表
        QuickSort(A,pivotpos+1,high);//划分右子表
    }
}
```

## 算法效率分析

每一层的QuickSort只需要处理剩余的待排序元素，时间复杂度不超过O(n)



把n个元素组织成二叉树，二叉树的层数就是递归调用的层数

$n$ 个结点的二叉树

$$\text{最小高度} = \lfloor \log_2 n \rfloor + 1$$

$$\text{最大高度} = n$$

时间复杂度=O( $n$ \*递归层数)

$$\text{最好时间复杂度} = O(n \log_2 n)$$

$$\text{最坏时间复杂度} = O(n^2)$$

空间复杂度=O(递归层数)

$$\text{最好空间复杂度} = O(\log_2 n)$$

$$\text{最坏空间复杂度} = O(n)$$

$$\text{平均时间复杂度} = O(n \log_2 n)$$

快速排序是所有内部排序算法中平均性能最优的排序算法

## 比较好的情况

若每一次选中的“枢轴”将待排序序列划分为均匀的两个部分，则递归深度最小，算法效率最高

## 最坏的情况

若每一次选中的“枢轴”将待排序序列划分为很不均匀的两个部分，则会导致递归深度增加，算法效率变低

若初始序列有序或逆序，则快速排序的性能最差（因为每次选择的都是最靠边的元素）

---

快速排序算法优化思路：尽量选择可以把数据中分的枢轴元素。

1. 选头、中、尾三个位置的元素，取中间值作为枢轴元素

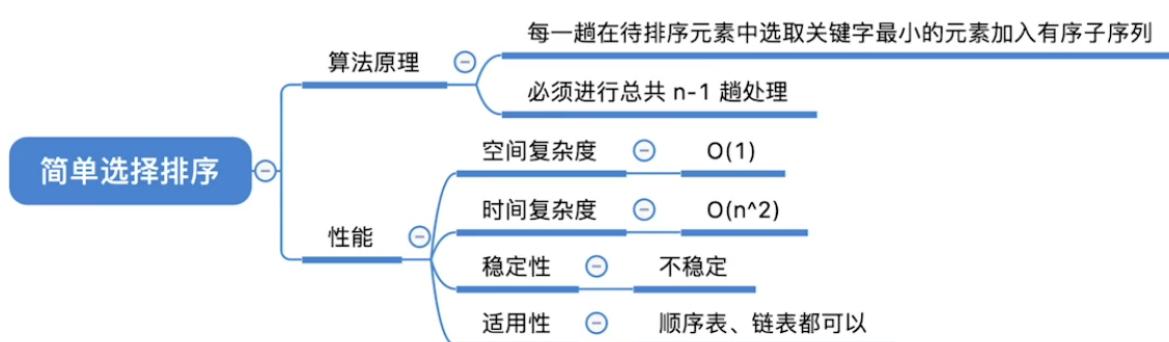
2. 随机选一个元素作为枢轴元素

## 稳定性

不稳定

# 选择排序

## 简单选择排序



选择排序：每一趟在待排序元素中选取关键字最小（或最大）的元素加入有序子序列

每一趟在待排序元素中选取关键字最小的元素加入有序子序列

$n$ 个元素的简单选择排序需要 $n-1$ 趟处理

# 算法实现

```
//简单选择排序
void selectSort(int A[], int n)
{
    for(int i=0;i<n-1;i++)//一共进行n-1趟
    {
        int min=i;//记录最小元素位置
        for(int j=i+1;j<n;j++)//在A[i...n-1]中选择最小的元素
            if(A[j]<A[min])min=j;//更新最小元素的位置
        if(min!=i)swap(A[i],A[min]);//封装的swap()函数共移动元素3次
    }
}
//交换
void swap(int &a, int &b)
{
    int temp=a;
    a=b;
    b=temp;
}
```



## 算法性能分析

空间复杂度O(1)

时间复杂度O(n^2)

无论有序、逆序、还是乱序，一定需要n-1趟处理

总共需要对比关键字

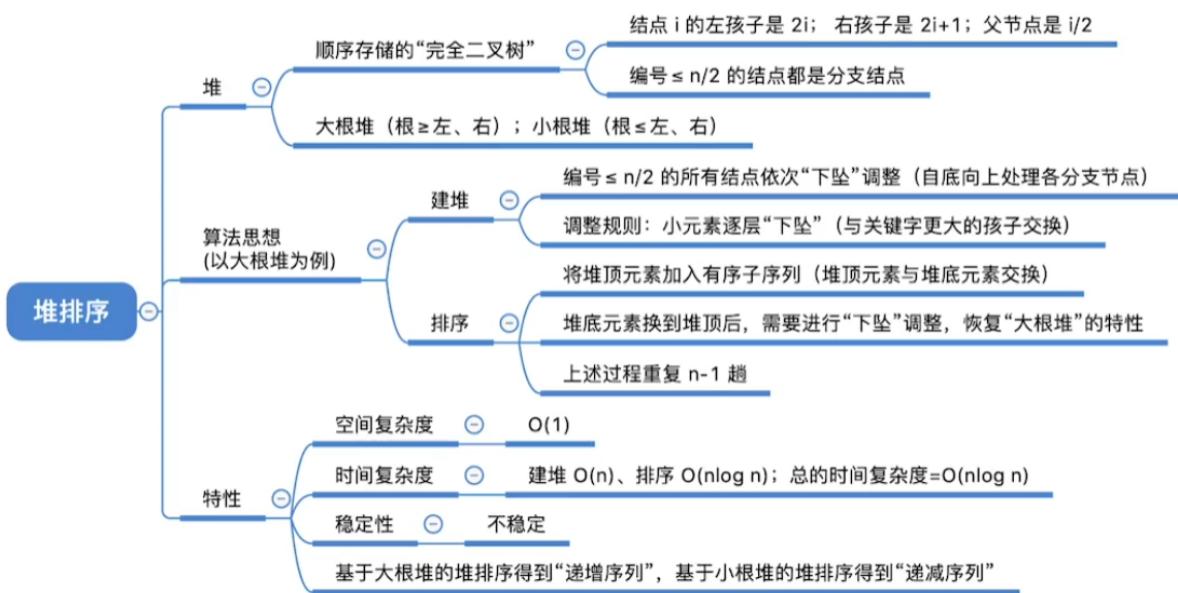
$$(n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2} \text{ 次}$$

元素交换次数<n-1

稳定性：不稳定

适用性：既可以用于顺序表，也可用于链表

# 堆排序



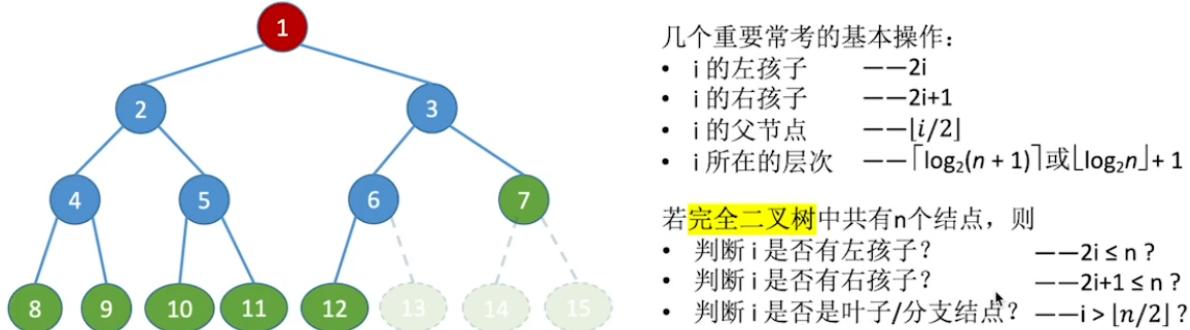
选择排序：每一趟在待排序元素中选取关键字最小（或最大）的元素加入有序子序列

## 什么是堆(Heap)?

若  $n$  个关键字序列  $L[1 \dots n]$  满足下面某一条性质，则称为堆(Heap)

- 若满足： $L(i) \geq L(2i)$  且  $L(i) \geq L(2i + 1)$  ( $1 \leq i \leq n/2$ )，大根堆(大顶堆)
- 若满足： $L(i) \leq L(2i)$  且  $L(i) \leq L(2i + 1)$  ( $1 \leq i \leq n/2$ )，小根堆(小顶堆)

## 二叉树的顺序存储



大根堆：完全二叉树中，根  $\geq$  左、右

小根堆：完全二叉树中，根  $\leq$  左、右

## 建立大根堆

大根堆：根  $\geq$  左、右

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

在顺序存储的完全二叉树中，非终端结点编号  $i \leq \lfloor n/2 \rfloor$

检查当前结点是否满足

根  $\geq$  左、右

若不满足，将当前结点与更大的一个孩子互换

若元素互换破坏了下一级的堆，则采用相同的方法继续往下调整（小元素不断“下坠”）

## 建立大根堆（代码）

```
//建立大根堆
void BuildMaxHeap(int A[], int len)
{
    for(int i=len/2; i>0; i--) //从后往前调整所有非终端结点
        HeadAdjust(A, i, len);
}
//将以k为根的子树调整为大根堆
void HeadAdjust(int A[], int k, int len)
{
    A[0]=A[k]; //A[0]暂存子树的根节点
    for(int i=2*k; i<=len; i*=2) //沿key较大的子结点向下筛选
    {
        if(i<len&&A[i]<A[i+1])
            i++; //取key较大的子结点的下标
        if(A[0]>=A[i]) break; //筛选结束
        else
        {
            A[k]=A[i]; //将A[i]调整到双亲结点上
            k=i; //修改k值，以便继续向下筛选
        }
    }
    A[k]=A[0]; //被筛选结点的值放入最终位置
}
```

从最底层的分支结点开始调整

## 基于大根堆进行排序

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

注意：基于“大根堆”的堆排序得到“递增序列”

## 基于大根堆进行排序（代码）

```
//建立大根堆
void BuildMaxHeap(int A[], int len);
//将以k为根的子树调整为大根堆
void HeadAdjust(int A[], int k, int len);
//堆排序的完整逻辑
void HeapSort(int A[], int len)
{
    BuildMaxHeap(A, len); //初始建堆
    for(int i=len; i>1; i--) //n-1趟的交换和建堆过程
    {
```

```

        swap(A[i], A[1]); //堆顶元素和堆底元素交换
        HeadAdjust(A, 1, i-1); //把剩余的待排序元素整理成堆
    }
}

```

i指向待排序元素序列中的最后一个（堆底元素）

## 算法效率分析

下方有两个孩子，则“下坠”一层，需要对比关键字2次

结论：一个结点，每“下坠”一层，最多只需对比关键字2次

若树高为h，某结点在第i层，则将这个结点向下调整最多只需要“下坠” $h-i$ 层，关键字对比次数不超过 $2(h-i)$

$n$ 个结点的完全二叉树树高 $h = \lfloor \log_2 n \rfloor + 1$

第i层最多有 $2^{i-1}$ 个结点，而只有第1 ( $h - 1$ )层的结点才有可能需要下坠调整

建堆的过程，关键字对比次数不超过 $4n$ ，建堆时间复杂度 =  $O(n)$

根结点最多“下坠” $h-1$ 层，每下坠一层

而每“下坠”一层，最多只需对比关键字2次，因此每一趟排序复杂度不超过

$$O(h) = O(\log_2 n)$$

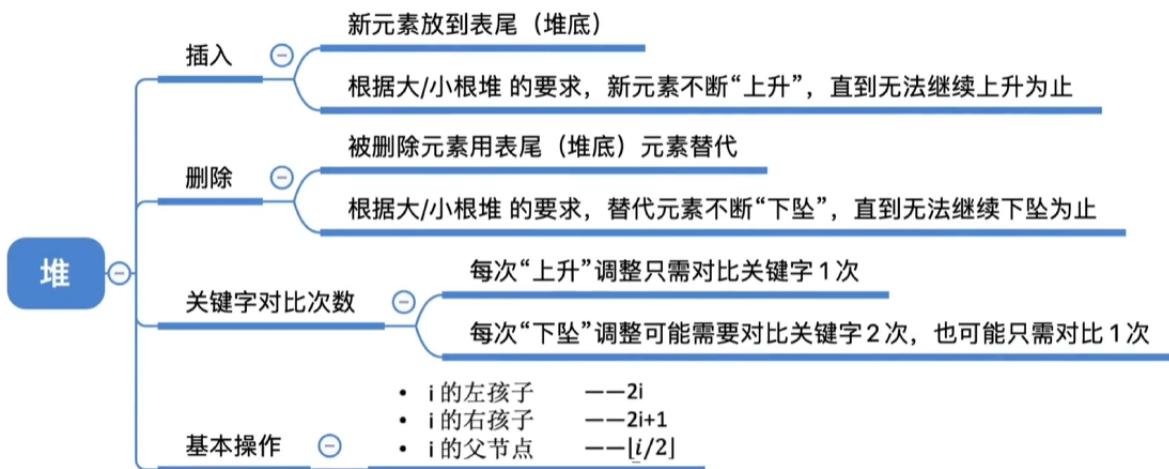
共 $n - 1$ 趟，总的时间复杂度 =  $O(n \log_2 n)$

堆排序的时间复杂度 =  $O(n) + O(n \log_2 n) = O(n \log_2 n)$

堆排序的空间复杂度 =  $O(1)$

结论：堆排序是不稳定的

## 堆的插入删除



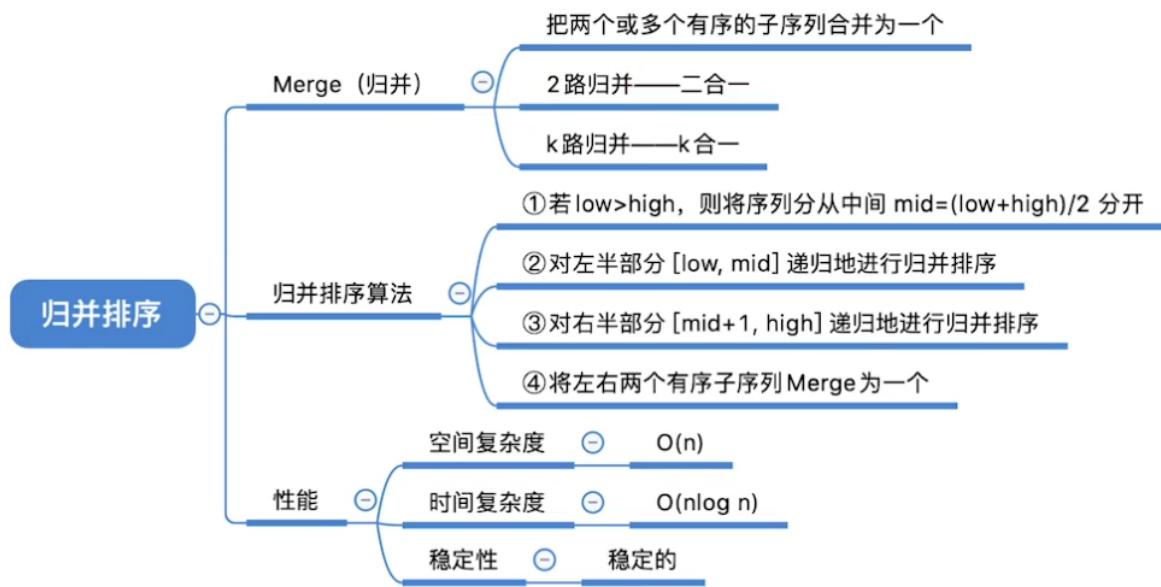
### 在堆中插入新元素

对于小根堆，新元素放到表尾，与父结点对比，若新元素比父结点更小，则将二者互换。新元素就这样一路“上升”，直到无法继续上升为止

## 在堆中删除元素

被删除的元素用堆底元素替代，然后让该元素不断“下坠”，直到无法下坠为止

## 归并排序



## 什么是Merge (归并/合并) ?

归并：把两个或多个已经有序的序列合并成一个

结论：m路归并，每选出一个元素需要对比关键字m-1次

核心操作：把数组内的两个有序序列归并为一个

## “2路”归并

“2路”归并——每选出一个小元素只需对比关键字1次

对比i,j所指元素，选择更小的一个放入k所指位置

## “4路”归并

“4路”归并——每选出一个元素只需对比关键字3次

对比p1,p2,p3,p4所指元素，选择更小的一个放入k所指位置

## 代码实现

```
int *B=(int *)malloc(n*sizeof(int)); //辅助数组B
//A[low...mid]和[mid+1...high]各自有序,将两个部分归并
void Merge(int A[], int low, int mid, int high)
{
    int i,j,k;
    for(k=low; k<=h; k++)
        B[k]=A[k]; //将A中所有元素复制到B中
    for(i=low, j=mid+1, k=i; i<=mid&&j<=high; k++)
    {
        if(B[i]<=B[j])
```

```

        A[k]=B[i++]; //将较小值复制到A中
    else
        A[k]=B[j++];
    }
    while(i<=mid)A[k++]=B[i++];
    while(j<=high)A[k++]=B[j++];
}

void MergeSort(int A[], int low, int high)
{
    if(low<high)
    {
        int mid=(low+high)/2; //从中间划分
        MergeSort(A, low, mid); //对左半部分归并排序
        MergeSort(A, mid+1, high); //对右半部分归并排序
        Merge(A, low, mid, high); //归并
    }
}

```

## 算法效率分析

2路归并的“归并树”——形态上就是一棵倒立的二叉树

二叉树的第 $h$ 层最多有 $2^{h-1}$ 个结点

若树高为 $h$ , 则应满足 $n \leq 2^{h-1}$

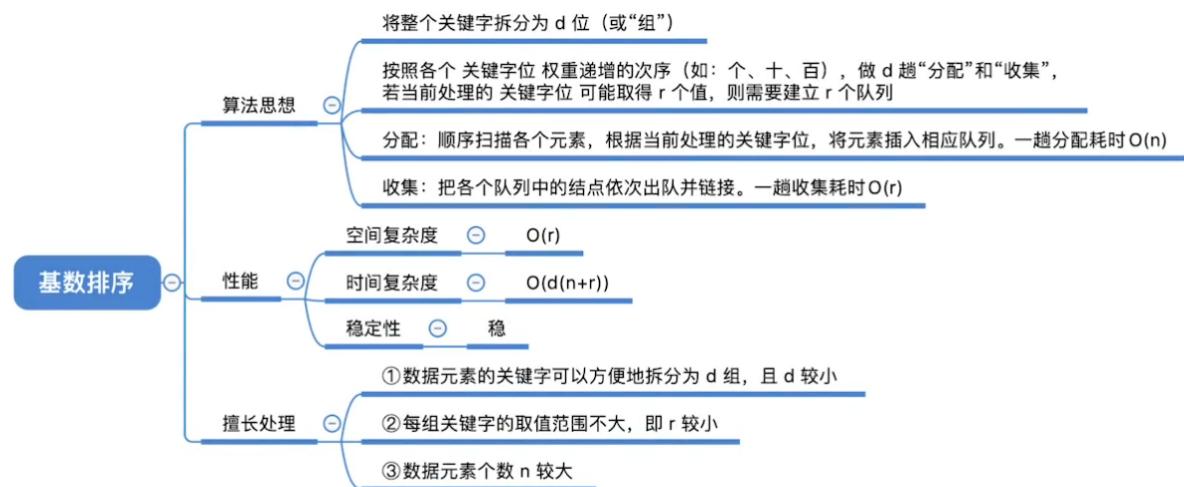
即 $h - 1 = \lceil \log_2 n \rceil$

结论:  $n$ 个元素进行2路归并排序, 归并趟数 =  $\lceil \log_2 n \rceil$

每趟归并时间复杂度 $O(n)$ , 则算法时间复杂度 $O(n \log_2 n)$

空间复杂度 =  $O(n)$ , 来自于辅助数组 $B$

## 基数排序 (Radix Sort)



第一趟: 以“个位”进行“分配”

第一趟“收集”结束: 得到按“个位”递减排序的序列

第二趟: 以“十位”进行“分配”, “个位”越大的越先入队

第二趟“收集”结束: 得到按“十位”递减排序的序列, “十位”相同的按“个位”递减排序

第三趟: 以“百位”进行“分配”, “十位”越大的越先入队

第三趟按“百位”分配、收集：得到一个按“百位”递减排列的序列，若“百位”相同则按“十位”递减排列，若“十位”还相同则按“个位”递减排列

假设长度为 $n$ 的线性表中每个结点 $a_j$ 的关键字由 $d$ 元组 $(k_j^{d-1}, k_j^{d-2}, k_j^{d-3}, \dots, k_j^1, k_j^0)$ 组成  
其中,  $0 \leq k_j^i \leq r - 1 (0 \leq j < n, 0 \leq i \leq d - 1)$ ,  $r$ 称为基数

基数排序得到递减序列的过程如下,

初始化：设置 $r$ 个空队列,  $Q_{r-1}, Q_{r-2}, \dots, Q_0$

按照各个关键位权重递增的次序(个, 十, 百), 对 $d$ 个关键位分别做分配和收集

分配：顺序扫描各个元素, 若当前处理的关键位 =  $x$ , 则将元素插入 $Q_x$ 队尾

收集：把 $Q_{r-1}, Q_{r-2}, \dots, Q_0$ 各个队列中的结点依次出队并链接

## 算法效率分析

基数排序通常基于链式存储实现

```
typedef struct LinkNode{
    ElemenType data;
    struct LinkNode *next;
}LinkNode, *LinkList;

typedef struct{//链式队列
    LinkNode *front,*rear;//队列的队头和队尾指针
}LinkQueue;
```

收集一个队列只需 $O(1)$ 的时间

```
p->next=Q[6].front;
Q[6].front=NULL;
Q[6].rear=NULL;
```

需要 $r$ 个辅助队列, 空间复杂度= $O(r)$

一趟分配 $O(n)$ , 一趟收集 $O(r)$ , 总共 $d$ 趟分配、收集, 总的时间复杂度= $O(d(n+r))$

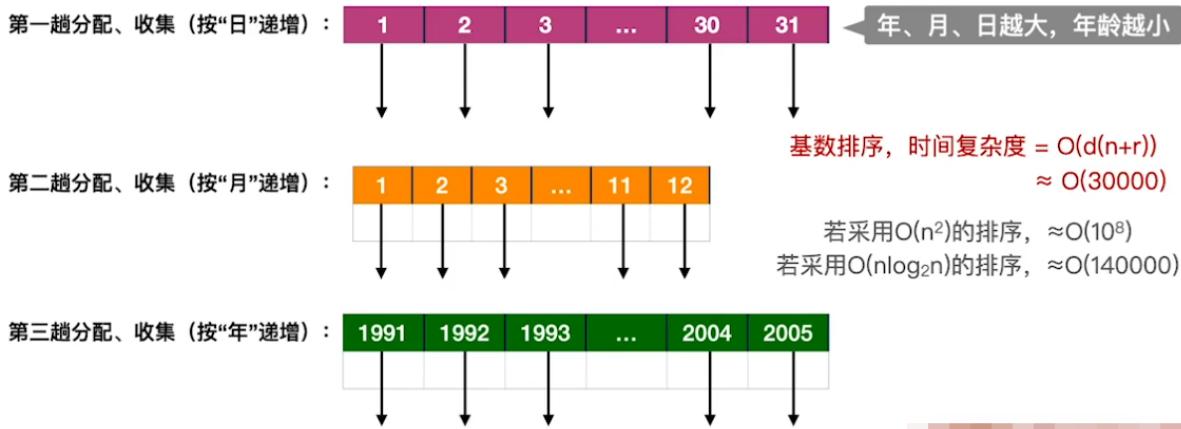
把关键字拆为 $d$ 个部分, 每个部分可能取得 $r$ 个值

基数排序是稳定的 基数太稳

# 基数排序的应用

某学校有 10000 学生，将学生信息按**年龄递减**排序

生日可拆分为三组关键字：年(1991~2005)、月(1~12)、日(1~31) 权重：年>月>日

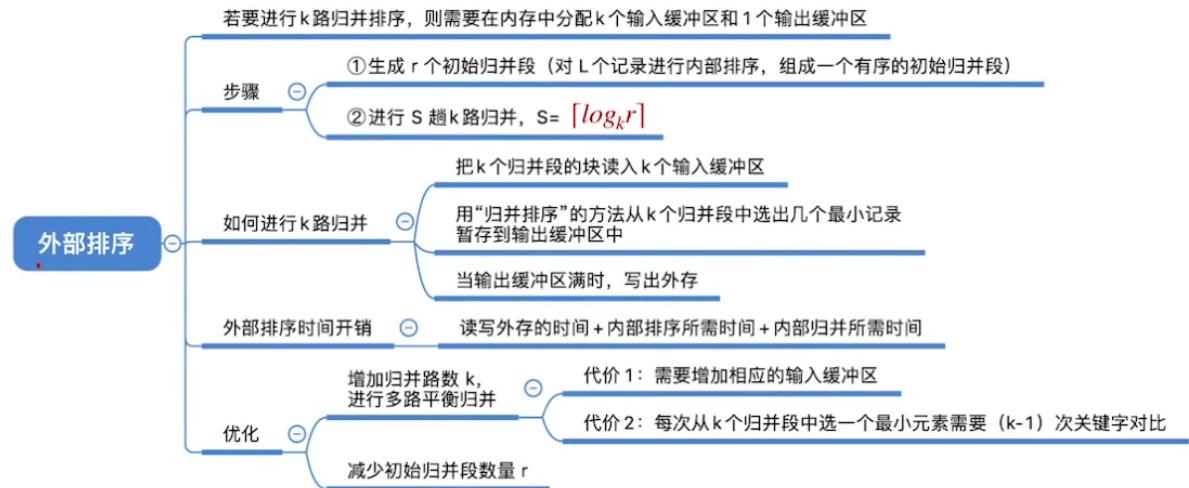


基数排序，时间复杂度=  $O(d(n+r))$

基数排序擅长解决的问题：

1. 数据元素的关键字可以方便地拆分为d组，且d较小
2. 每组关键字的取值范围不大，即r较小
3. 数据元素个数n较大

# 外部排序

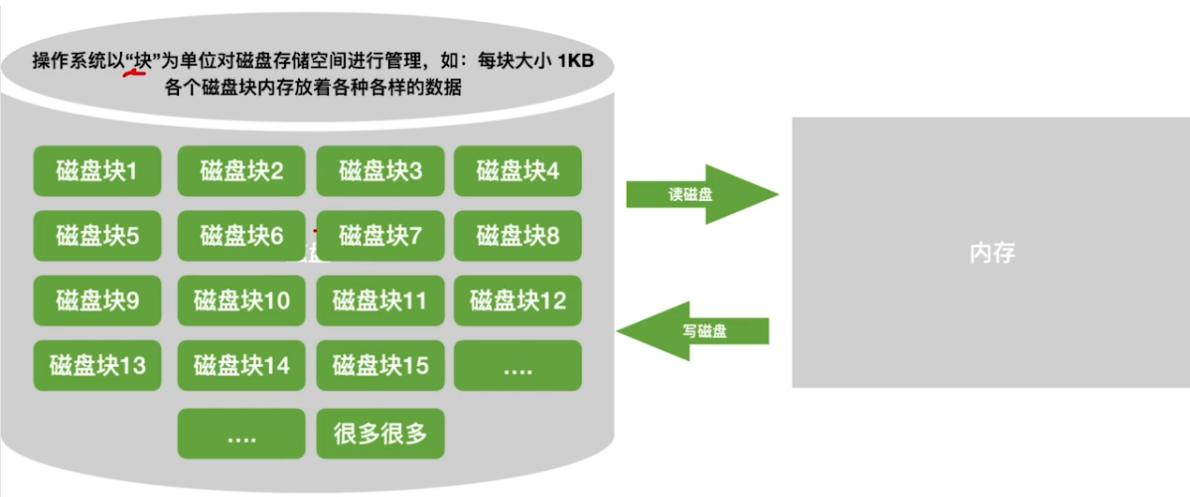


可用“败者树”减少关键字对比次数

注：生成初始归并段，若共N个记录，内存工作区可以容纳L个记录，则初始归并段数量  $r=N/L$

可用“置换-选择排序”进一步减少初始归并段数量

# 外存、内存之间的数据交换



操作系统以“块”为单位对磁盘存储空间进行管理，如：每块大小1KB各个磁盘块内存放着各种各样的数据

磁盘的读/写以“块”为单位 数据读入内存后才能被修改 修改完了还要写回磁盘

## 外部排序原理

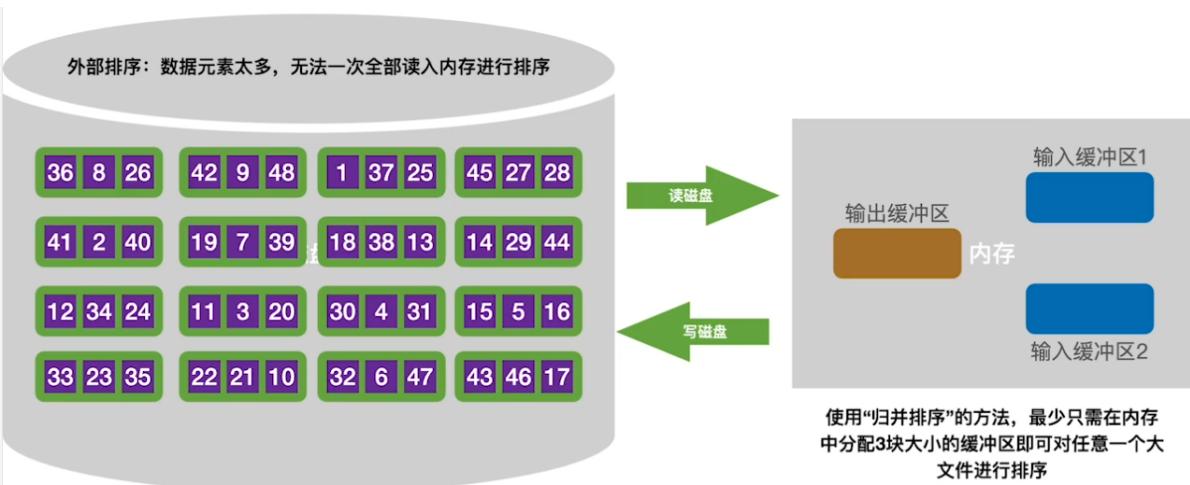
外部排序：数据元素太多，无法一次全部读入内存进行排序。

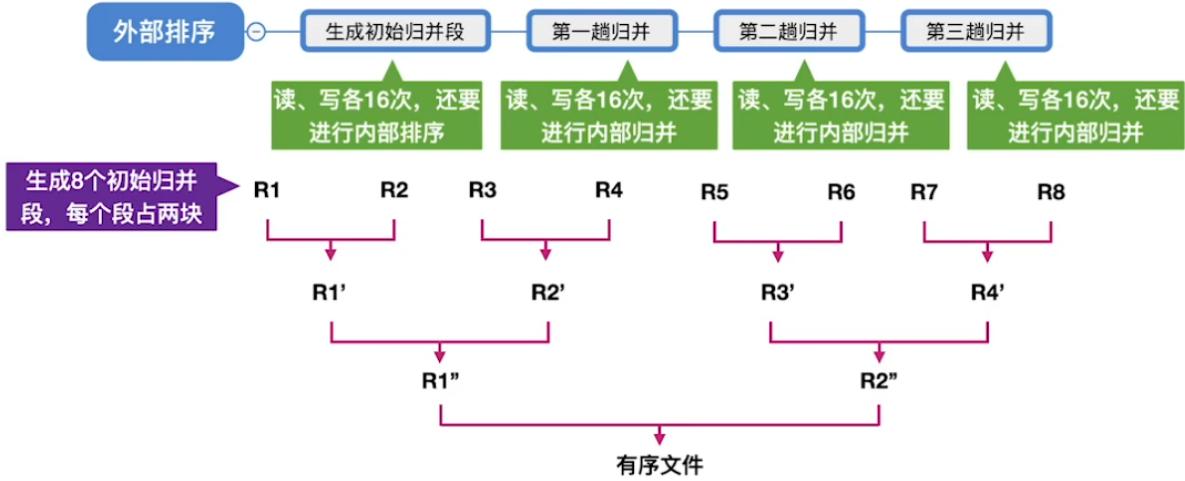
使用“归并排序”的方法，最少只需在内存中分配3块大小的缓冲区即可对任意一个大文件进行排序

## 构造初始“归并段”

“归并排序”要求各个子序列有序，每次读入两个块的内容，进行内部排序后写回磁盘

## 时间开销分析

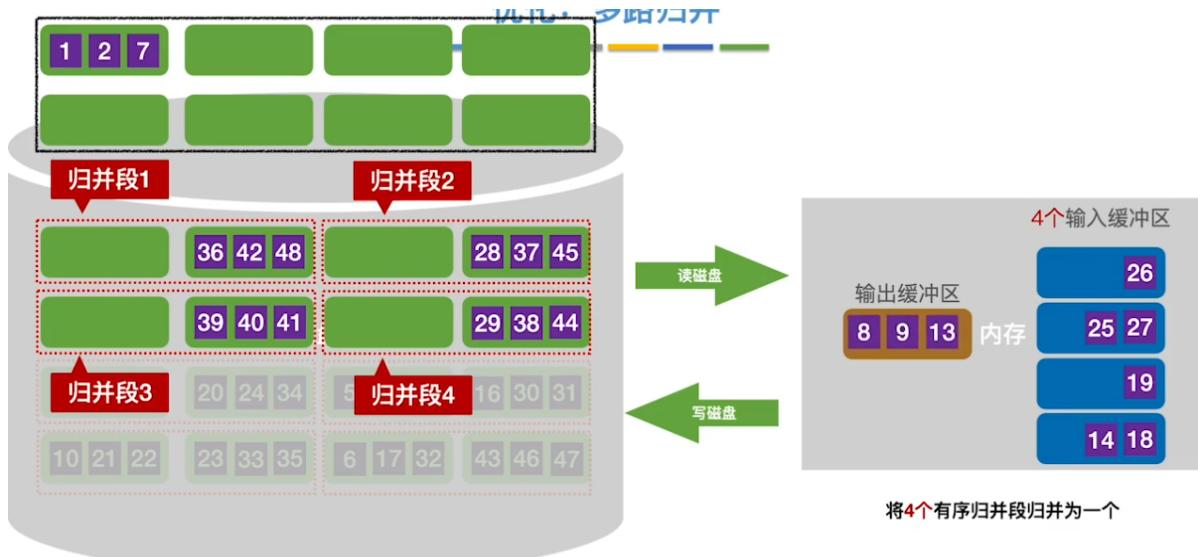




外部排序时间开销 = 读写外存的时间 + 内部排序所需时间 + 内部归并所需时间

## 如何优化

### 优化：多路归并



重要结论：采用多路归并可以减少归并趟数，从而减少磁盘I/O（读写）次数

对 $r$ 个初始归并段，做 $k$ 路归并，则归并树可用 $k$ 叉树表示

$$\text{若树高为 } h, \text{ 则归并趟数} = h - 1 = \lceil \log_k r \rceil$$

推导： $k$ 叉树第 $h$ 层最多有 $k^{h-1}$ 个结点

$$\text{则 } r \leq k^{h-1}, (h-1)_{\text{最小}} = \lceil \log_k r \rceil$$

$k$ 越大， $r$ 越小，归并趟数越少，读写磁盘次数越少

多路归并带来的负面映像：

1.  $k$ 路归并时，需要开辟 $k$ 个输入缓冲区，内存开销增加。
2. 每挑选一个关键字需要对比关键字 $(k-1)$ 次，内部归并所需时间增加。

## 优化：减少初始归并段数量

生成初始归并段的“内存工作区”越大，初始归并段越长

结论：若能增加初始归并段的长度，则可减少初始归并段数量r

## 什么是多路平衡归并？

1. 最多只能有k个段归并为一个
2. 每一趟归并中，若有m个归并段参与归并，则经过这一趟处理得到 $m/k$ 向上取整个新的归并段

## 败者树

### 多路平衡归并带来的问题

外部排序时间开销 = 读写外存的时间 + 内部排序所需时间 + 内部归并所需时间

归并趟数  $S = \lceil \log_k r \rceil$ ，归并路数  $k$  增加，归并趟数  $S$  减小，读写磁盘总次数减少

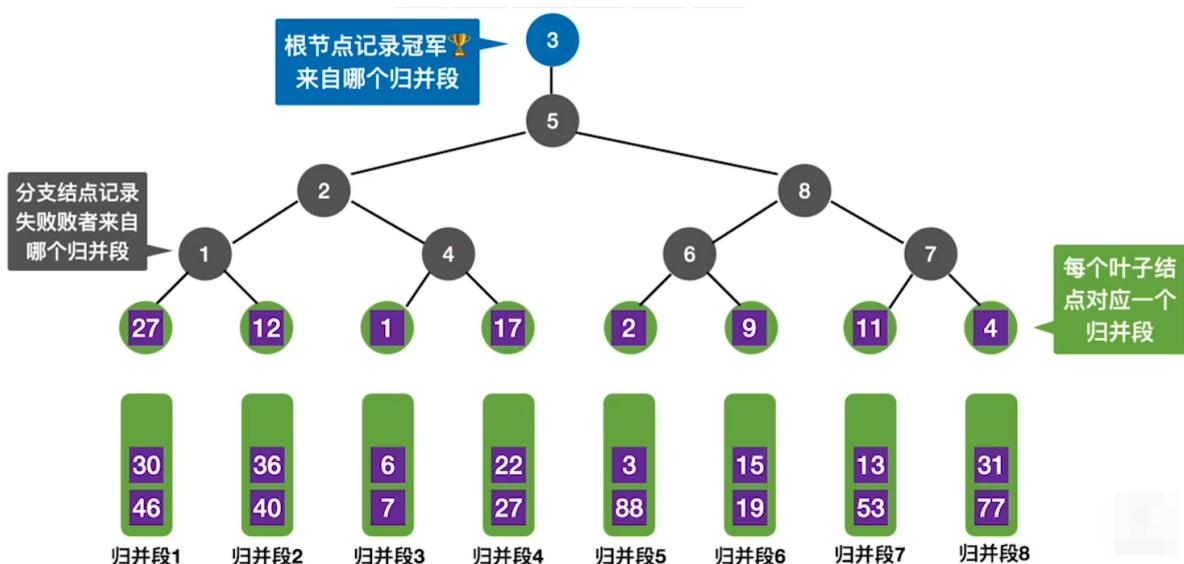
使用k路平衡归并策略，选出一个最小元素需要对比关键字( $k-1$ )次，导致内部归并所需时间增加

可用“败者树”进行优化

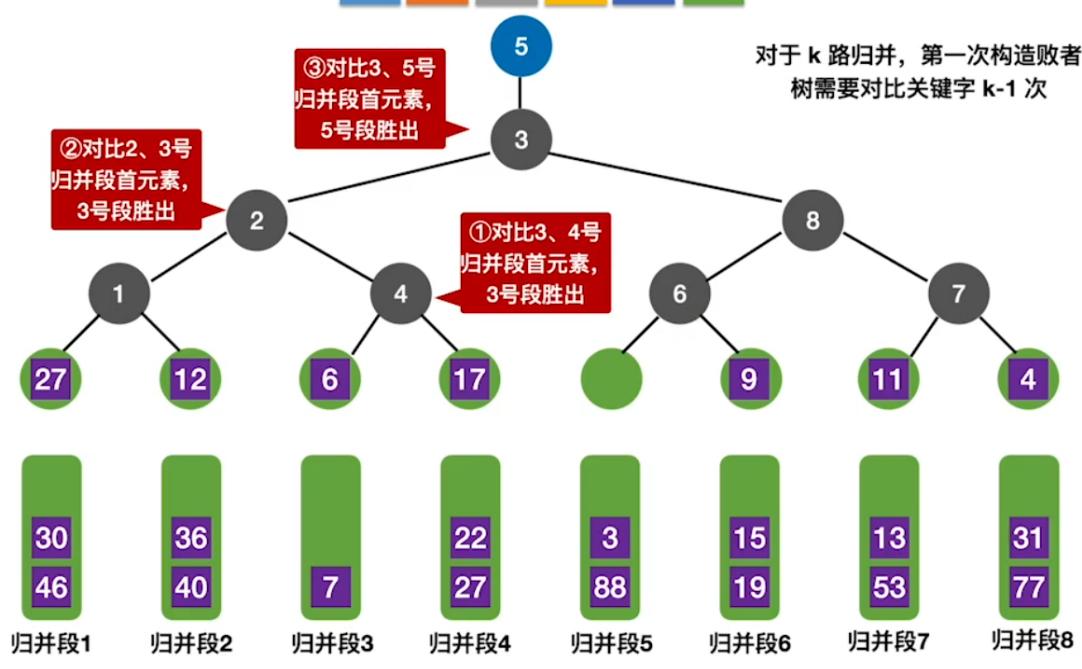
### 败者树的构造

败者树——可视为一棵完全二叉树（多了一个头头）。 $k$ 个叶结点分别是当前参加比较的元素，非叶子结点用来记忆左右子树中的“失败者”，而让胜者往上继续进行比较，一直到根结点。

### 败者树在多路平衡归并中的应用



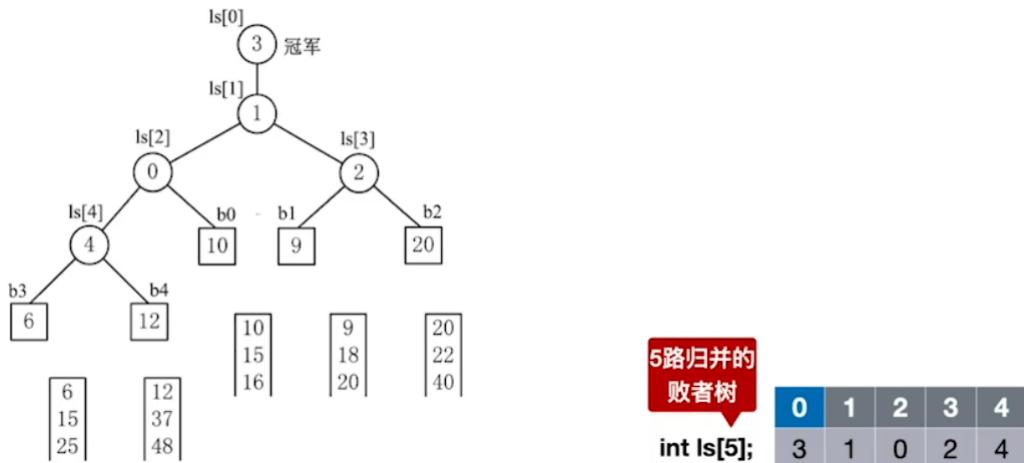
对于k路归并，第一次构造败者树需要对比关键字 $k-1$ 次



有了败者树，选出最小元素，只需对比关键字  $\lceil \log_2 k \rceil$  次

## 败者树的实现思路

k路归并的败者树只需要定义一个长度为k的数组即可



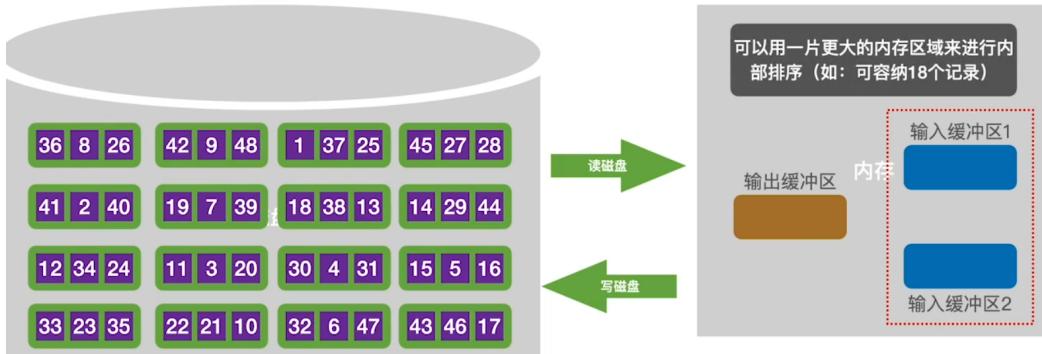
## 置换-选择排序

### 土办法构造初始归并段

可以用一片更大的内存区域来进行内部排序

用于内部排序的内存工作区WA可容纳l个记录，则每个初始归并段也只能包含l个记录，若文件共有n个记录，则初始归并段的数量

$$r = n/l$$



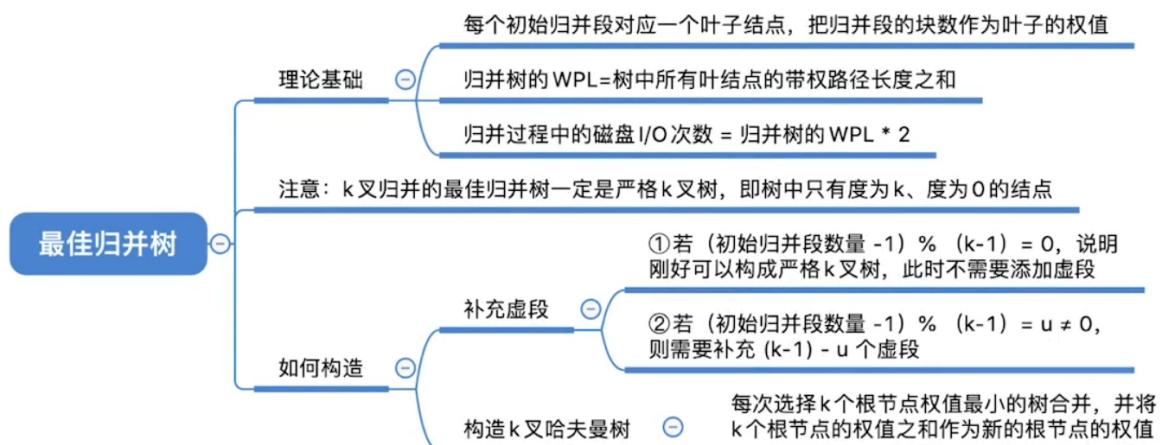
## 置换-选择排序

使用置换-选择排序，可以让每个初始归并段的长度超越内存工作区大小的限制。

设初始待排文件为FI，初始归并段输出文件为FO，内存工作区为WA，FO和WA的初始状态为空，WA可容纳w个记录。置换-选择算法的步骤如下：

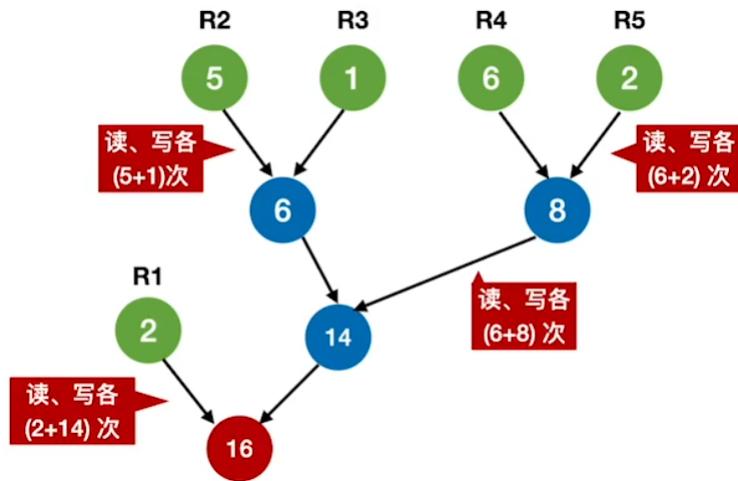
1. 从FI输入w个记录到工作区WA。
2. 从WA中选出其中关键字取最小值的记录，记为MINIMAX记录。
3. 将MINIMAX记录输出到FO中去。
4. 若FI不空，则从FI输入下一个记录到WA中。
5. 从WA中所有关键字比MINIMAX记录的关键字大的记录中选出最小关键字记录，作为新的MINIMAX记录。
6. 重复3到5，直至在WA中选不出新的MINIMAX记录为止，由此得到一个初始归并段，输出一个归并段的结束标志到FO中去。
7. 重复2到6，直至WA为空。由此得到全部初始归并段。

## 最佳归并树



## 归并树的神秘性质

5个初始归并段，所占块数各不相同，进行二路归并



每个初始归并段看作一个叶子结点，归并段的长度作为结点权值，则

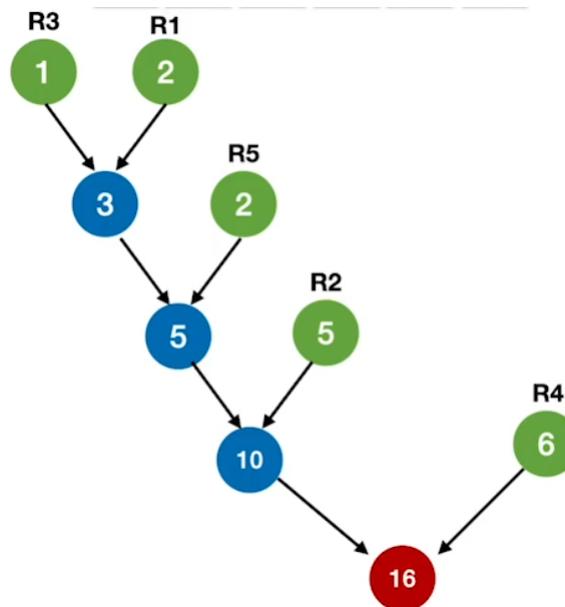
$$\text{上归并树的带权路径长度 } WPL = 2 * 1 + (5 + 1 + 6 + 2) * 3 = 44 = \text{读磁盘次数} = \text{写磁盘次数}$$

重要结论

$$\text{归并过程中的磁盘I/O次数} = \text{归并树的WPL} * 2$$

要让磁盘I/O次数最少，就要使归并树WPL最小——哈夫曼树

## 构造2路归并的最佳归并树

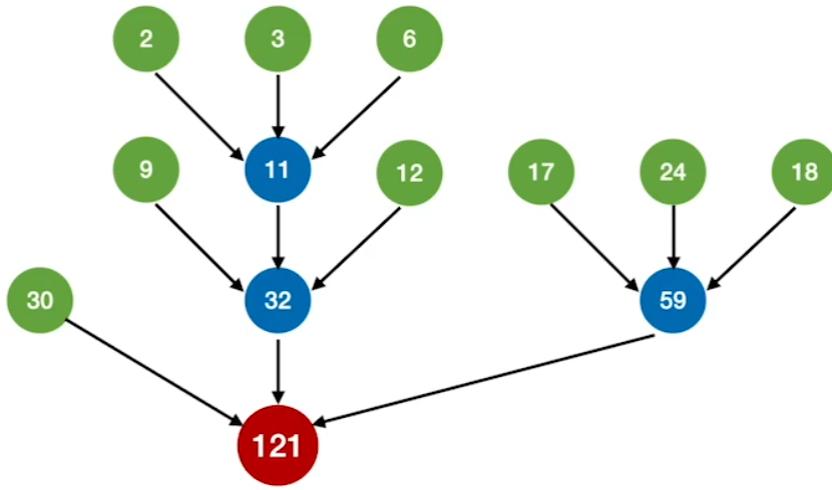


$$\text{最佳归并树 } WPL_{min} = (1 + 2) * 4 + 2 * 3 + 5 * 2 + 6 * 1 = 34$$

读磁盘次数 = 写磁盘次数 = 34次

$$\text{总的磁盘I/O次数} = 68$$

# 多路归并的最佳归并树

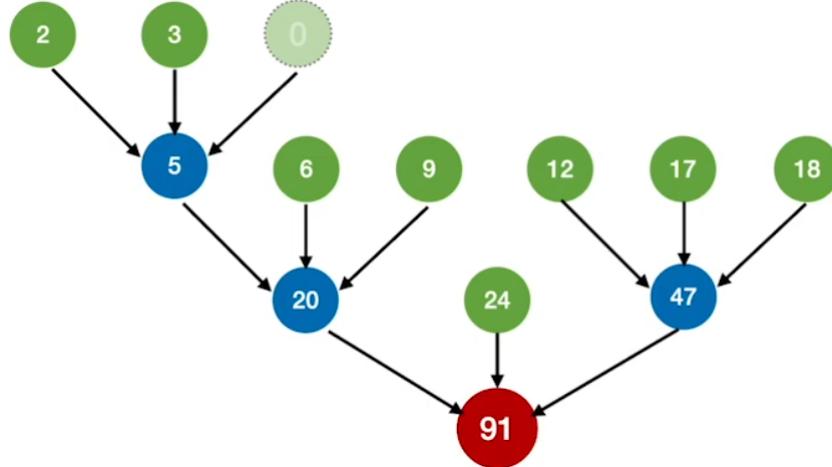


$$WPL_{min} = (2 + 3 + 6) * 3 + (9 + 12 + 17 + 24 + 18) * 2 + 30 * 1 = 223$$

归并过程中磁盘I/O总次数 = 446次

## 如何减少一个归并段

注意：对于k叉归并，若初始归并段的数量无法构成严格的k叉归并树，则需要补充几个长度为0的“虚段”，再进行k叉哈夫曼树的构造。



$$WPL_{min} = (2 + 3 + 0) * 3 + (6 + 9 + 12 + 17 + 18) * 2 + 24 * 1 = 163$$

归并过程中磁盘I/O总次数 = 326次

## 添加虚段的数量

k叉的最佳归并树一定是一棵严格的k叉树，即树中只包含度为k，度为0的结点。

设度为k的结点有 $n_k$ 个，度为0的结点有 $n_0$ 个，归并树总结点数 = n

$$\text{初始归并段数量} + \text{虚段数量} = n_0$$

$$n = n_0 + n_k$$

$$kn_k = n - 1$$

$$\Rightarrow n_0 = (k - 1)n_k + 1$$

$$\Rightarrow n_k = \frac{n_0 - 1}{k - 1}$$

1. 若(初始归并段数量 - 1)  $\mod (k - 1) = 0$ , 说明刚好可以构成严格k叉树, 此时不需要添加虚段
2. 若(初始归并段数量 - 1)  $\mod (k - 1) = u \neq 0$ , 则需要补充 $(k - 1) - u$ 个虚段

