④**稳定性**：在第 i 趟找到最小元素后，和第 i 个元素交换，可能导致第 i 个元素与其含有相同关键字元素的相对位置发生改变，因此不稳定。

⑤**适用性**：适用于链表；

2） 堆排序

◎**定义**：$L(i) >= L(2i)$且 $L(i) >= L(2i+1)$ $(1 \le i \le \lfloor n/2 \rfloor)$称为大根堆；$L(i) <= L(2i)$且 $L(i) <= L(2i+1)$ $(1 \le i \le \lfloor n/2 \rfloor)$称为小根堆；大根堆（大顶堆）最大元素放在根节点；

①**操作步骤**：首先将存放在 L[1…n]中的 n 个元素建成初始堆，堆顶元素就是最大值；输出堆顶元素后，将堆底元素送往堆顶，此时根节点不满足性质，将堆顶元素向下调整保持性质，再次输出堆顶元素；如此反复直到堆中仅剩一个元素为止；从后向前检查所有非终端结点$(1 \le i \le \lfloor n/2 \rfloor)$是否满足根>左、右，若不满足，将当前结点与更大的一个孩子结点更换；若元素互换破坏了下一级的堆，则采用相同方法继续向下调整；

```
void BuildMaxHeap(ElemType A[],int len){
    for(int i=len/2;i>0;i--)     //从 i=[n/2]~1，反复调整堆
        HeadAdjust(A,i,len);
}
void HeadAdjust(ElemType A[],int k,int len){
//函数 HeadAdjust 将元素 k 为根的子树进行调整
    A[0]=A[k];                     //A[0]暂存子树的根结点
    for(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
}
void HeapSort(ElemType A[],int len){
    BuildMaxHeap(A,len);           //初始建堆
    for(i=len;i>1;i--){            //n-1 趟的交换和建堆过程
        Swap(A[i],A[1]);           //输出堆顶元素(和堆底元素交换)
        HeadAdjust(A,1,i-1);       //调整，把剩余的 i-1 个元素整理成堆
    }
}
```

②**空间复杂度**：$O(1)$；

③**时间复杂度**：建堆时间为 $O(n)$；一个结点下坠一层对比 2 次，树高 h 结点在第 i 层，需下坠 h-i 次，即对比 2(h-i)次；有 n 个结点的完全二叉树，树高 $h = \lfloor n/2 \rfloor + 1$；关键字对比次数不超过$\sum_{i=1}^{n-1} 2 \cdot 2^{i-1}(h-i) \le 4n$，建堆时间复杂度为 $O(n)$；每趟最多下坠 h-1 层，因此每趟不超过 $O(h) = O(\log_2 n)$，共进行 h-1 趟，因此时间复杂度为 $O(n\log_2 n)$；

④**稳定性**：不稳定；

⑤在堆中插入删除元素：如果有两个孩子结点，则下坠 1 层需要对比 2 次，如果只有 1 个孩子，则需要对比 1 次；插入新元素放在堆底，将插入元素与父结点对比，(小根堆)如果小于父结点则交换，直到无法继续上升为止；删除元素后，用堆底元素代替被删除元素，然后让该元素与（小根堆）较小的孩子比较，不断下坠；

5. 归并排序和基数排序

1） 归并排序

①**操作步骤**：假定待排序表含有 n 个记录，可将其视为 n 个有序子表，每个字表长度为 1；两两归并，得到个长度为 2 或 1 的有序表；如此重复，直到合并为一个长度为 n 的有序表为止；m 路归并，每选出一个元素需要对比关键字 m-1 次；

```
ElemType *B=(ElemType *)malloc((n+1)*sizeof(ElemType)); //辅助数组 B
void Merge(ElemType A[],int low,int mid,int high){
//表 A 的两段 A[low…mid]和 A[mid+1…high]各自有序，将它们合并成一个有序表
    for(int k=low;k<=high;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])                      //比较 B 的左右两段中的元素
            A[k]=B[i++];                    //将较小值复制到 A 中
        else
            A[k]=B[j++];
    }//for
    while(i<=mid)   A[k++]=B[i++];          //若第一个表未检测完，复制
    while(j<=high)  A[k++]=B[j++];          //若第二个表未检测完，复制
}
void MergeSort(ElemType A[],int low,int high){
    if(low<high){
        int mid=(low+high)/2;               //从中间划分两个子序列
        MergeSort(A,low,mid);               //对左侧子序列进行递归排序
        MergeSort(A,mid+1,high);            //对右侧子序列进行递归排序
        Merge(A,low,mid,high);              //归并
    }//if
}
```

②**空间复杂度**：辅助空间为 n 个单元，空间复杂度为 $O(n)$；

③**时间复杂度**：每趟归并的时间复杂度为 $O(n)$；共需进行$\lceil \log_2 n \rceil$趟归并，时间复杂度为 $O(n\log_2 n)$；

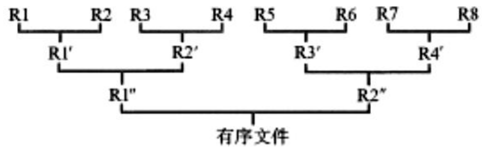
④**稳定性**：2 路归并是稳定的；

2） 基数排序

①**操作步骤**：假设长度为 n 的线性表中每个结点 aj 的关键字由 d 元组($k_j^{d-1}, k_j^{d-2}, ..., k_j^1, k_j^0$)组成。其中 $0 \le k_j^i \le r-1$ （$0 \le j < n, 0 \le i \le d-1$），r 称为基数，$k_j^{d-1}$为最主位关键字，$k_j^0$为最次位关键字；初始化设置 r 个空队列；按照各个关键字位权重递增次序（个十百），对 d 个关键字作分配和收集；分配：顺序扫描各个元素，若当前处理关键字位=x，就将元素插入 Qx 队尾；把各个队列中的结点依次出队并链接；第一趟收集结束，按个位递减排列；第二趟收集结束，按十位递减排列，十位相同按个位递减排列；

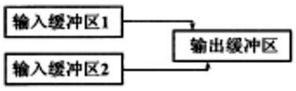
②**空间复杂度**：一趟排序需要的辅助存储空间位 r，空间复杂度为 $O(r)$；

③**时间复杂度**：基数排序需要进行 d 趟分配和收集，一趟分配需要 O(n)，一趟收集需要 O(r)，所以基数排序时间复杂度为 O(d(n + r))，它与序列的初始状态无关；



④**稳定性**：按位排序时必须稳定，所以基数排序稳定；

6. 外部排序



1） k 路归并

①**外部排序原理**：归并排序最少只需要 3 个内存块即可对任意大的文件排序；首先构造初始归并段，因为归并排序要求各个子序列有序，因此每次读两块内容，内部排序后写回磁盘；第一趟归并，将两个有序归并段归并为一个，缓冲区 2 空了就要用归并段 2 的下一块补上……；

②**时间复杂度**：外部排序时间开销=读写外存时间+内部排序时间+内部归并时间；R1…R8 为 8 次内部排序得到的 8 个初始归并段，初始归并和每一趟归并读写磁盘各 16 次，二路归并读写磁盘次数为 32+32*3=128 次；

③**多路归并优化**：采用多路归并可以减少归并趟数，从而减少 I/O 次数，对 r 个初始归并段，做 k 路归并，归并树可用 k 叉树表示，若树高为 h，则归并趟数=h-1=$\lceil \log_k r \rceil$；故 k 越大,r 越小归并趟数越少；缺点：k 路归并需要开辟 k 个缓冲区内存开销增大，每挑选一个关键字需要对比 k-1 次内部排序开销增大；　　　减少初始归并段的数量，如果能增加初始归并段的长度，就可以减少归并段的数量；

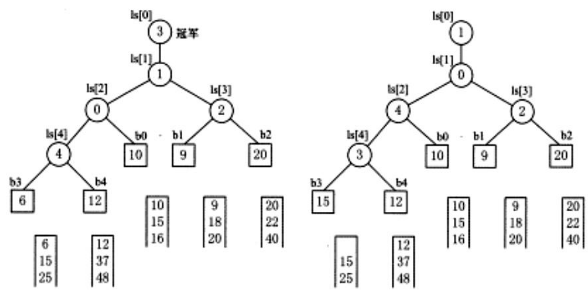
④**k 路平衡归并**：最多只能有 k 个段归并为 1 个；每一趟归并中，若有 m 个段参与归并，则一趟处理后得到$\lceil m/k \rceil$个归并段；

2） 多路平衡归并与败者树

①**败者树解决问题**：多路平衡归并排序中，从 k 个归并段选出一个最小/大元素需要对比关键字 k-1 次，构造败者树可以使关键字对比次数减少到$\lceil \log_2 k \rceil$；

②**败者树**：败者树是一颗完全二叉树多一个结点；k 个叶子结点对应 k 个归并段参加比较的元素，非页子结点用来记录左右子树中的失败者，胜者继续比较直到根节点之上的结点；对于 k 路归并，第一次构造败者树需要对比关键字 k-1 次；有败者树后，选出最小元素只需要对比关键字$\lceil \log_2 k \rceil$次；

③**败者树的构建**：如图



3） 置换-选择排序

◎**置换-选择排序解决问题**：让初始归并段长度增加，减少初始归并段数；

假设初始待排序文件为 FI，初始归并段文件为输出文件 FO，内存工作区为 WA，FO 与 WA 的初始状态为空，并假设内存工作去 WA 的容量可容纳 w 个记录，则置换-选择排序的操作的过程为：

①从 FI 输入 w 个记录到工作区 WA。

②从 WA 中选出其中关键字最小的记录，记为 MINIMAX 记录。

③将 MINIMAX 记录输出到 FO 中去。

④若 FI 不为空，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③~⑤，直至 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记到 FO 中去。

⑦重复②~⑥，直至 WA 为空。由此得到全部归并段。

4） 最佳归并树

①**最佳归并树解决问题**：经过置换选择排序后得到长度不等的初始归并段，最佳归并树使 I/O 次数最少；每个初始归并段视作叶子结点，归并段长度作为结点权值，则磁盘 I/O 次数=归并树的带权路径长度 WPL*2；可以运用哈夫曼树思想，让记录数最少的初始归并段最先归并，就可以建立总的 I/O 次数最少的最佳归并树；

②**非严格 k 叉树**：哈夫曼树中的最佳归并树应该是严格 k 叉树，即树中只有度为 3 或 0 的结点；若初始归并段不足以构成严格 k 叉树时，需添加长度为 0 的虚段；

③**如何判断添加虚段的数目**：设度为 0 的结点有n_0个，度为 k 的结点有n_k个，则对严格 k 叉树有$n_0 = (k-1)n_k + 1$，由此可得$n_k = \frac{n_0 - 1}{k-1}$；若$(n_0 - 1)\%(k-1) = 0$，则说明这$n_0$个叶结点正好构造 k 叉归并树；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，应当再加上 k-u-1 个空归并段；