Sorting: Introduction and Notation

①排序: 将一组任意序列(list)的数据元素/记录(entry/record), 重新排列成一个按关键字值有序的序列. **A list of entries to be sorted into order by a key field within each list entry.**

约定: • list中的数据元素均以contiguous storage(其数据结构如下)
• 按非递减顺序排列 i.e. in natural order (若i<j, 则 key[i]≤key[j])

```
typedef struct list {
    ElemType R[Maxsize + 1];
    int length; } List;
```
⇦
```
typedef struct {
    KeyType key;
    … } ElemType;
```
⇦
```
typedef  … KeyType;
```

②分类:
- Internal Sorting: 待排记录均在memory中
- External Sorting: 排序过程中需访问外存(disks, tapes)并交换记录数据

③排序方法的稳定性(stability):

A sorting method is called stable if, whenever two entries have equal keys, then these two entries will be in the same order on completion of the sorting function that they were in before sorting.

P289 算法的稳定性由方法本身决定, 对不稳定的排序方法, 不管其描述形式如何, 总能举出一个不稳定的实例. 反之对稳定的排序方法, 总能找到一种引起不稳定的描述形式.

Sorting: Introduction and Notation

④排序算法分析:

时间复杂性

Comparisons of keys

The spent time of moving entries around within the list (事实上记录移动一次比比较一次的耗时更多)

Worst-case analysis

Average analysis(即n个记录初始状态应考虑有n!种排列的可能性)

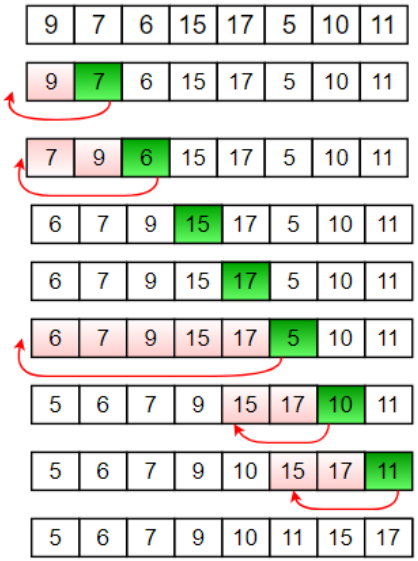
排序的两种基本操作: **(1)**比较⇨**LT, EQ, GT (2)**移动⇨**Assignment or Exchange**

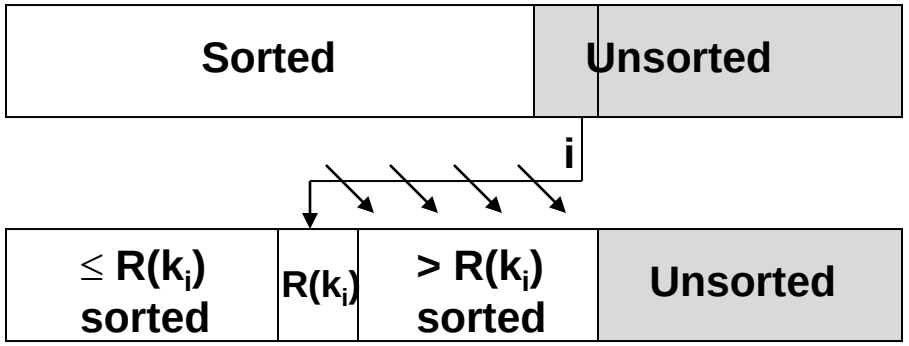
空间复杂性

若排序算法所需的辅助空间**(**额外空间**)**并不依赖于问题的规模**n(**数据输入**),** 即辅助空间是**O(1)(～constant extra space),** 则称之为就地排序**/ in-place sorting.**
对非就地排序算法则需考虑所需辅助空间的大小**,** 一般依赖于问题规模**n.**

Insert Sort/Sorting By Insertion 插入排序

思路: 设有n个初始记录序列{ $R(k_1), R(k_2), \ldots, R(k_n)$ }, 先将序列中的第一个记录看成是一个有序的子序列, 然后从第二个记录起逐个插入到该子序列中, 直至整个序列变成按关键字值非递减的有序序列.





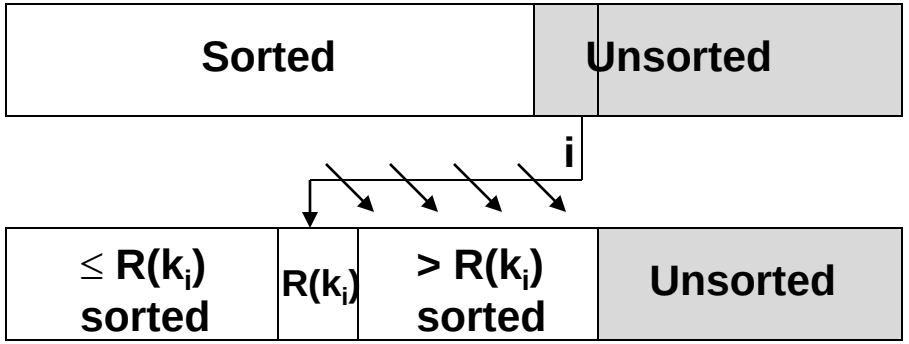

```
void InsertionSort(List L) { //算法描述: P265/A.10.1
//Pre: The contiguous list has been created. Each entry of list contains  a key.
//Post: The entries of list have been rearranged so that the keys in all the entries are
//sorted into nondecreasing order.
  for ( i=2; i<=L.length; i++ )
     if LT(L.r[i].key, L.r[i-1].key) {  //需将L.r[i]插入到有序子表中
        L.r[0] = L.r[i];  //L.r[0]用作sentinel
        for ( j=i-1; LT(L.r[0].key, L.r[j].key); j-- ) L.r[j+1] = L.r[j]; //记录后移
        L.r[j+1] = L.r[0]; //将L.r[i]插入到正确位置
     } }
```

Insert Sort/Sorting By Insertion 插入排序

```
{ for ( i=2; i<=L.length; i++ )
    if LT(L.r[i].key, L.r[i-1].key) {
      L.r[0] = L.r[i];
      for ( j=i-1; LT(L.r[0].key, L.r[j].key); j-- )  L.r[j+1] = L.r[j];
      L.r[j+1] = L.r[0]; } }
```

逆序时
⇦1次比较

⇦i次比较



算法分析:

- **in natural order**
  - Comparisons = $\sum_{i=2}^{n} 1 = n-1$
  - No moving of entries
- **in its reverse order**
  - Comparisons = $\sum_{i=2}^{n} (i+1) = \frac{1}{2}(n+4)(n-1)$
  - Moving number of entries: 同上“比较次数”
- **in random order (all possible orderings of the keys are equally likely)**

  第i(i≥2)个记录进行插入操作有i种可能出现的结果/插入位置:

  - ① No moving at all, 概率1/i(即一次比较, 0次移动)
  - ②插至前面的i-1中某一位置, 概率 $\frac{i-1}{i}$ (它对应着算法中第二个for语句)

Insert Sort/Sorting By Insertion 插入排序

上述②中i-1种可能出现的插入位置也就是算法中内循环语句执行的平均次数: $\frac{1+2+3+\ldots+(i-1)}{i-1} = \frac{i}{2}$ 即 $\frac{i}{2}$ 次比较和 $\frac{i}{2}$ 次移动.

因此每次的外循环(for each iteration) $\begin{cases} \frac{i}{2}+1 \text{ 次比较} \\ \frac{i}{2}+2 \text{ 次移动} \end{cases}$

综合①② $\begin{cases} \text{Comparisons} = \frac{1}{i}\times 1 + \frac{i-1}{i}\times\left(\frac{i}{2}+1\right) = \frac{i+1}{2} = \frac{i}{2} + O(1) \\ \text{Assignments} = \frac{1}{i}\times 0 + \frac{i-1}{i}\times\left(\frac{i}{2}+2\right) = \frac{i+3}{2} - \frac{2}{i} = \frac{i}{2} + O(1) \end{cases}$

共进行i=2 to n次iteration, 所以其平均比较、移动次数均为:

$$\sum_{i=2}^{n}\left(\frac{1}{2}i + O(1)\right) = \frac{1}{2}\sum_{i=2}^{n} i + O(n) = \frac{1}{4}n^2 + O(n)$$

- in partially-sorted order

算法稳定性: 该算法是稳定的排序算法.

插入排序算法的改进⇨Binary Insertion Sort

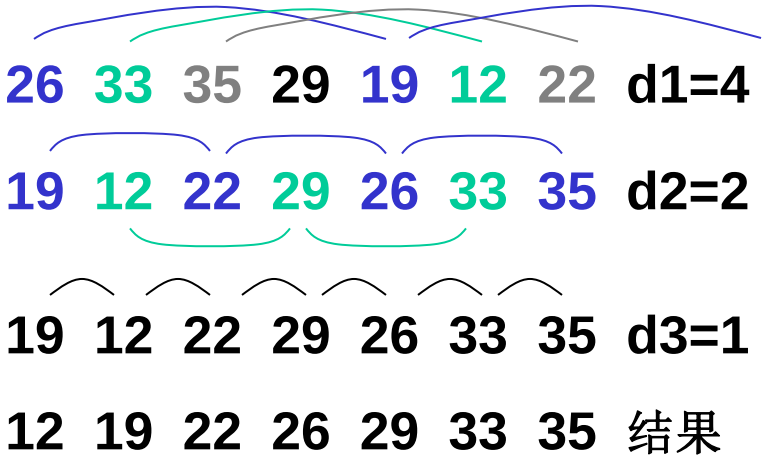
二分插入排序是指在已排好序的子序列(sorted sublist)中采用二分查找法查找下一个记录应该插入的位置. 因此该排序方法仅减少了关键字的比较次数, 而记录移动的次数未变. P267/A.10.2

Shell Sort希尔排序/ Diminishing-increment Sort缩小增量排序

背景: 插入排序方法每次总跟相邻的关键字作比较且作相应的记录移动, 而Shell排序的主要目的是减少关键字比较及记录移动次数.

思路: 先取一个整数d1(d1<n)(增量/increment), 把表中所有记录分成d1组(d1个子序列, 即把所有相差距离为d1倍数的记录看成一组), 然后在各组内进行插入排序(至此称之为一趟(one pass)); 再取d2<d1(缩小增量), 重复上述分组和插入法排序, 直至 d_i=1(i≥1), 即整个记录为一组(这时就相当于插入排序). 上述每一趟都向最终排序结果更进了一步.

例:设每趟increment = $\lceil \frac{n}{2} \rceil$




```
void ShellSort(List L) {
 increment = L.length;
 do { increment = increment/3 + 1;
        ShellInsertSort (L, increment); //为一趟
        //修改的插入排序, P272A.10.4
  } while (increment > 1);  }
void ShellInsertSort(List L, int dk)
//相比于InsertSort,前后记录位置的增量为dk,而非1
 { for ( i = dk + 1; i <= L.length; i++ )
     if LT(L.r[i].key, L.r[i - dk].key) {
        L.r[0] = L.r[i];  //r[0]是暂存单元
        for ( j = i - dk; j>0 &&
              LT(L.r[0].key, L.r[j].key); j -= dk )
           L.r[j + dk] = L.r[j];
        L.r[j+dk] = L.r[0]; } } //当dk=1时,则和插入
        //排序基本一致,只是通过j>0控制下标越界
```

Shell Sort希尔排序/ Diminishing-increment Sort缩小增量排序

说明: ① 有关increment值的选定方法到目前为止还是一个数学上的难题.

If the increments are chosen close together, it needs to make more passes, but there will likely be quicker.

If the increments are decrease rapidly, then fewer but longer passes will occur.

② “增量”序列的两个共同特征

- 增量序列中的最后一个增量值必为1
- 避免序列中增量值互为倍数, 如16, 8, 4, 2, 1(因为下一趟可能出现重复比较)

算法分析: Shell算法的时间复杂性是“增量”序列的函数 ($t(n)=f(d_k)$), 同样是一个数学上的难题. 大量实验证明当记录数目n在某个范围内, Shell算法所需比较和移动次数in range of $O(n^{1.25})$ to $O(1.6n^{1.25})$.

算法稳定性: 以{6 8 5^1 7 5^2 2}为例说明, 假设增量序列为3, 2, 1

Initial order of the list: 6 8 5^1 7 5^2 2

After the pass with increment=3: 6 5^2 2 7 8 5^1

After the pass with increment=2: 2 5^2 6 5^1 8 7

After the pass with increment=1: 2 5^2 5^1 6 7 8

该算法不稳定.

Bubble Sort / 冒泡排序

思路: A well-known algorithm called bubble sort processed by scanning the list from end to beginning, swapping entries whenever a pair of adjacent keys is found to be out of order. In this first pass, the smallest key in the list will have "bubbled" to the beginning, but the later keys may still be out of order. Thus the pass scanning for pairs out of order is put in a loop that first makes the scanning pass go all the way to list->length-1.

26	12	12	12	12	12
33	26	19	19	19	19
35	33	26	22	22	22
29	35	33	26	26	26
19	29	35	33	29	29
12	19	29	35	33	33
22	22	22	29	35	35
↑	↑	↑	↑	↑	↑
初	第	第	第	第	第
始	i=1	i=2	i=3	i=4	i=5
序	趟	趟	趟	趟	趟
列	后	后	后	后	后

```
void BubbleSort(List L) { //改进的版本
for(i=1,tag=1; tag && i<=L.length-1; i++) {
    tag=0; //tag=1 or 0分别表示该趟有或无记录交换
    for(j=L.length; j>i; j--)
        if (LT(L.r[j].key, L.r[j-1].key)) {
            Swap(L.r [j], L.r[j-1]); tag=1; } }
}
```

算法分析: 该算法是稳定的排序算法

Best-case(正序): 比较n-1次(O(n)), 无移动

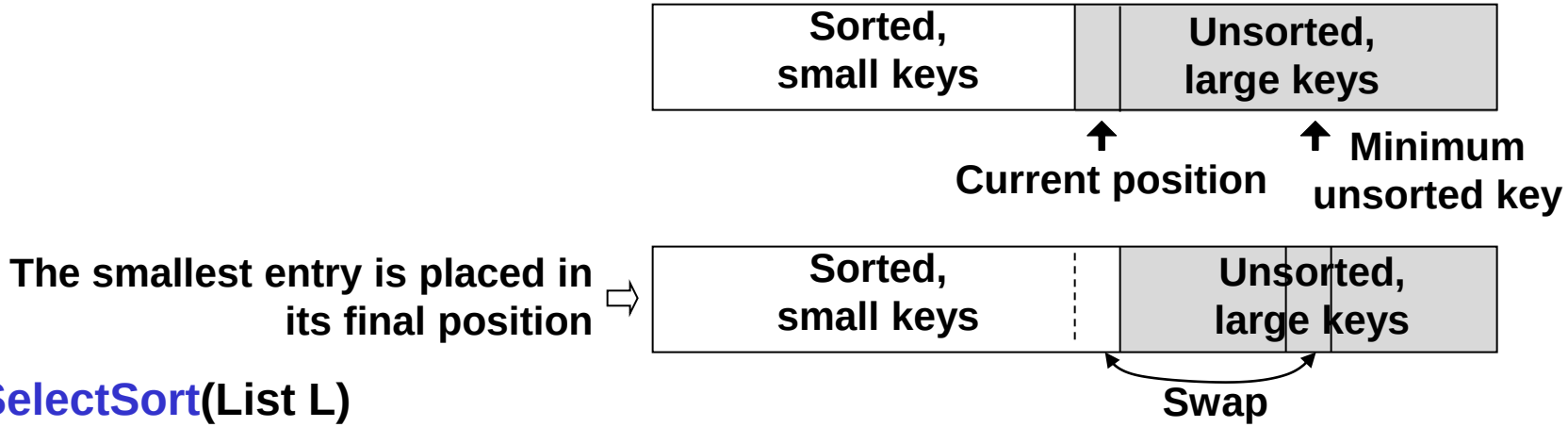
Worst-case(逆序): 比较、移动次数

$$= \sum_{i=n}^{2} (i-1) = n(n-1)/2 = O(n^2)$$

Selection Sort

背景: 插入排序每进行一次插入操作都会引起大量记录的移动, 特别是当采用顺序存储结构且记录本身又很大时, 这种移动会化去更多的时间开销. 因此选择排序的目的是如何减少记录移动的次数.

思路: 设有n个初始记录序列{ $R(k_1)$, $R(k_2)$, …, $R(k_n)$ }, 先选择其中key值最小的记录 $R(k_i)$, 并将$R(k_i)$与$R(k_1)$交换; 然后从剩下的n-1个记录中{ $R(k_2)$, $R(k_3)$, …, $R(k_n)$ }选择key值最小的记录$R(k_i)$并与$R(k_2)$交换; 如此循环至n-1趟, 从{ $R(k_{n-1})$, $R(k_n)$ }中选择较小的key值并放在$R(k_{n-1})$ 中并结束整个排序操作. P277-278/A.10.9




```
void SelectSort(List L)
{for (current = 1; current < L.length; current++)
    { min = MinKey(L, current); Swap(L.r[min], L.r[current]); } }

int MinKey(List L, int smallest)
{for (current = smallest+1; current <= L.length; current++)
    if (LT(L.r[current].key, L.r[smallest].key)) smallest = current;
 return smallest;  }
```

Selection Sort

算法分析:

- 选择排序与list中记录的初始顺序无关, 所以无需考虑其正序、逆序的情况
- 调用n-1次Swap, 因此assignments = 3(n-1) = 3n + O(1)
- 调用n-1次MinKey, 每次调用共需i-1次比较, i为剩下的entries, 因此:

$$\text{Comparisons} = \sum_{i=n}^{2} (i-1) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 + O(n)$$

算法稳定性: 以$\{ 2^1, 2^2, 1 \}$为例说明, 该算法不稳定.

插入排序与选择排序算法分析比较:

	Selection	Insertion(average)
Assignments of entries	3.0n + O(1)	0. 25n^2 + O(n)
Comparisons of entries	0.5n^2 + O(n)	0. 25n^2 + O(n)

Conclusions:

当n>>时, 0.25n^2>>3n, 这时从记录移动角度, 插入排序比选择排序费时(特别当entry is large时), 但选择算法的比较次数是插入算法的两倍.

⇨对选择算法 < move efficiently
comparisons redundant

Divide-and-Conquer Sorting ⇨ MergeSort & QuickSort

背景: 排序时表中的记录越少, 所花排序的工作量也就越少.

分治(divide-and conquer)的策略:

- 分解(divide): 将原问题分解为若干个子问题
- 求解(conquer): 递归地解各子问题,若子问题规模足够小,则直接求解
- 组合(combine)[可选]: 将各子问题的解组合成原问题的解

```
Sort(list) {
    if the list has length greater than 1 then
        { Partition the list into lowlist, highlist;
          Sort(lowlist); //recursive
          Sort(highlist);
          Combine(lowlist, highlist);
        }
}
```

MergeSort / 归并排序

含义:

所谓“归并/Merge”就是将两个(2-路归并) 长度分别为h的有序表(sorted list) 合并成一个新的长度为2h 的有序表.

利用“归并”技术进行排序的过程称归并排序.

归并排序有两种实现方法:

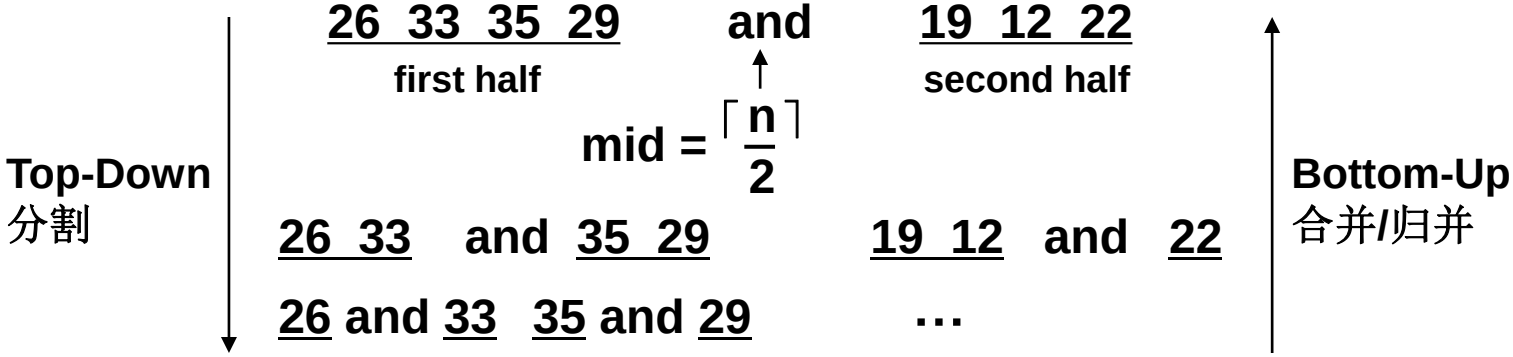
- 自顶向下 / Top-Down
- 自底向上 / Bottom-Up

MergeSort / 归并排序

归并排序自顶向下的基本思路: (递归形式)

- 分解: 将当前区间R[low..high]一分为二, 即计算分解点 mid = ⌈(low+high)/2⌉
- 求解: 递归地对两个子区间R[low..mid]和R[mid+1..high]进行归并排序
- 组合: 将已排序的两个子区间R[low..mid]和R[mid+1..high]归并为一个有序的区间R[low..high]

自顶向下归并排序的一个例子: 设有序列 { 26 33 35 29 19 12 22 }



归并排序自顶向下的算法描述:

```
void MergeSort (ElemType &R[], ElemType &R1[], int low, int high)
//对数组R归并排序, 数组R1辅存. 可以通过R1=R语句来一次性分配所需辅存空间
{ if (low < high) {  //区间长度大于1
    mid = low + (high - low)/2;  //分解, mid = ⌈(low+high)/2⌉
    MergeSort(R1, R, low, mid);  //递归地对R[low..mid]进行排序
    MergeSort(R1, R, mid + 1, high);  //递归地对R[mid+1..high]进行排序
    Merge(R, R1, low, mid, high);  //归并(同自底向上的方法)
} //可以通过上例的divide和merge的轨迹/递归树来进一步理解
```

MergeSort / 归并排序

归并排序自底向上的基本思路: (迭代形式)

- 先对子表表长h=1的表进行有序处理, 后不断使h=2×h再进行子表有序处理(称之为一趟), 直至h≥n为止(共$\lceil \log_2 n \rceil$趟) -- MergeSort算法
- 其中的一趟的操作是: 对某个确定长的(设长为h)子表, 先将n个记录分成$\lceil n/h \rceil$个长为h的子表, 后再两两归并(Merge算法/P284/A10.12), 共需循环执行Merge算法若干次($\lfloor \lceil n/h \rceil /2 \rfloor$次)后得到长度为n的表, 这一过程即为一趟的处理 -- MergePass算法

⇨归并排序: MergeSort{调用若干次MergePass{调用若干次Merge}}

自底向上归并排序的一个例子: 设有关键字序列: { 26 33 35 29 19 12 22 }

MergeSort 调用3趟 MergePass

初始关键字序列, h=1: 26 33 35 29 19 12 22 ⇦ 3次Merge

第一趟归并后, h=2: 26 33 29 35 12 19 22 ⇦ 2次Merge

第二趟归并后, h=4: 26 29 33 35 12 19 22 ⇦ 1次Merge

第三趟归并后, h=8: 12 19 22 26 29 33 35

MergeSort / 归并排序

归并排序自底向上的算法描述 -- Merge

```
void Merge (ElemType &sr[ ], ElemType &tr[ ], int i, int m, int n)
//P283~284/A10.12--2路归并: 将有序的sr[i..m]和sr[m+1..n]归并为有序的tr[i..n]
{ for (j = m+1, k = i; i <= m && j <= n; ++k) {  //将sr中的记录由小到大地并入tr
       if ( LQ(sr[i].key, sr[j].key) )  tr[k] = sr[i++]; //LQ ~ Less or Equal
      else tr[k] = sr[j++]; }
  if (i <= m)  tr[k..n] = sr[i..m]; //将剩下的sr[i..m]复制到tr
  if (j <= n)   tr[k..n] = sr[j..n];  //将剩下的sr[j..n]复制到tr
}
```

MergeSort / 归并排序

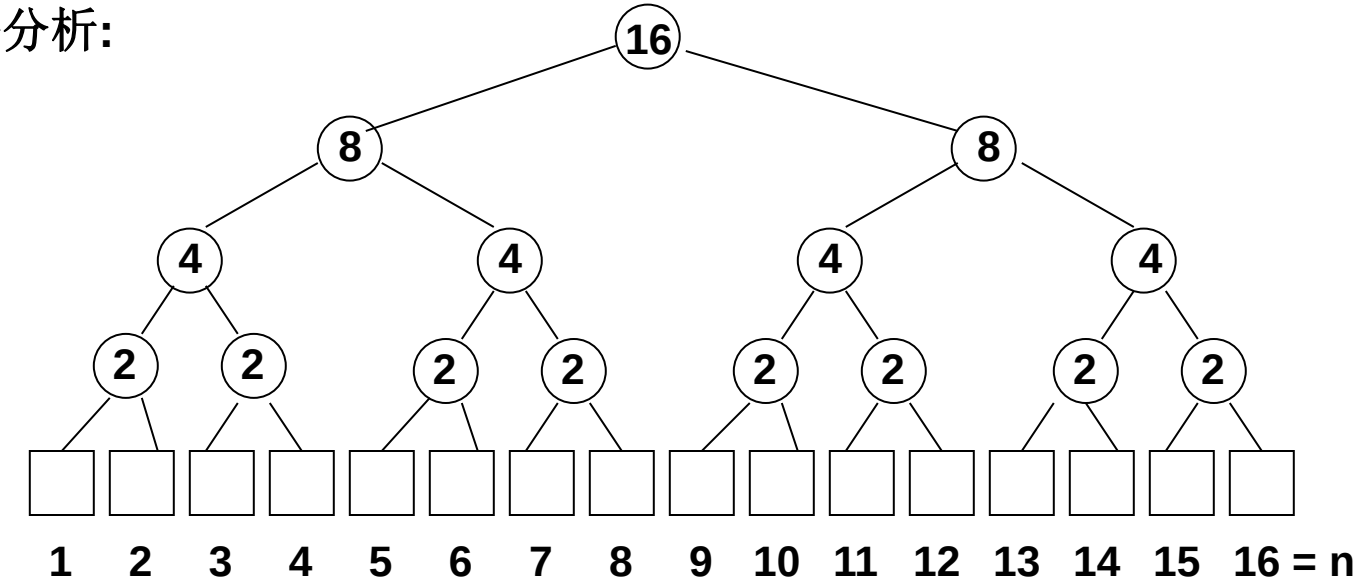
归并排序自底向上的算法描述 -- MergePass & MergeSort

```
void MergePass(ElemType &r, ElemType &r1, int h, int n)  //P284.A10.13
{ i = 1;   //从第一条记录开始
  while (n-i+1>=2*h) {  //剩下的记录数不小于两个子表长度时
      h1=i;  t1=h1+h-1;  t2=i+2*h-1;
      //h1为第一子表的首元素下标值，t1、t2分别为第一、第二子表尾元素下标值
      Merge(r, r1, h1, t1, t2);  //P283-284.A10.12
      i=i+2*h; }
  if ((n-i+1)<=h)   //剩下的记录数不大于一个子表的长度时
      for (j=i; j<=n; j++)   r1[j]=r[j];
  else {   //剩下的记录数大于一个而小于两个子表的长度时
      h1=i;  t1=h1+h-1;  t2=n;
      Merge(r, r1, h1, t1, t2);  } }

void MergeSort(List L)  //P284.A10.14
{ h = 1; n = L.length;
  while (h<n) {
      MergePass(L.r, L.r1, h, n);  h = h*2;  //这时r1用作辅助空间对r有序
      MergePass(L.r1, L.r, h, n);  h = h*2;  //这时r用作辅助空间对r1有序
   } }
```

MergeSort / 归并排序

算法分析:



- 树中的一层即为归并算法中的一趟
- 归并的趟数 ≤ $\lceil \log_2 n \rceil$ (n = L.length)
- 每一层(即一趟)最多有n次比较

∴ Comparisons ≤ $O(n\log_2 n) = O(n\log_2 n)$

同样, Assignments ≤ $O(n\log_2 n) = O(n\log_2 n)$

Contrast with InsertionSort:

- 当n > 16时, $n\log_2 n < n^2/4$ 即 $\log_2 n < n/4$.

  ∴当n >>时, comparisons: MergeSort << InsertionSort

对顺序存储结构, MergeSort需和源表一样大小的auxiliary space即O(n).

结论: 从空间角度, 适合外部排序.

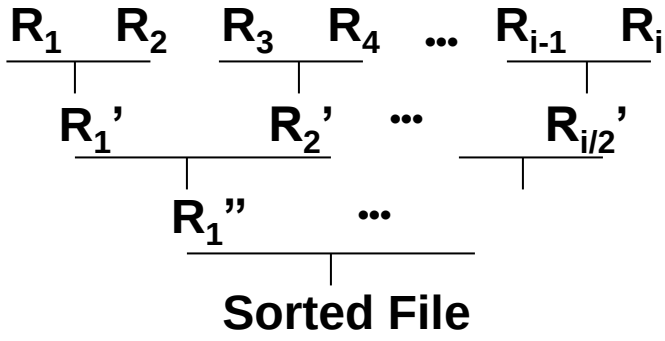
算法稳定性: 该算法是稳定的排序算法

MergeSort / 归并排序 -- External Sort / 外部排序 P295

基本思路:

1. 含n个记录的文件分成⌈n/h⌉(长度为h)个子文件/段segment, 依次读入RAM, 并用有效的内部排序方法对之排序, 排好序的段称为初始归并段.
2. 对外存中的若干初始归并段, 利用MergeSort方法逐项归并, 由小到大, 直至形成整个文件为止, 即:

其中R_x为归并段



归并的"示意性"工作原理: 以例子加以说明

设有4500个记录的文件, 若磁盘的读写单位(physical block)为250个记录的大小, RAM空间为750个记录的大小, 则可将RAM分成3等份:

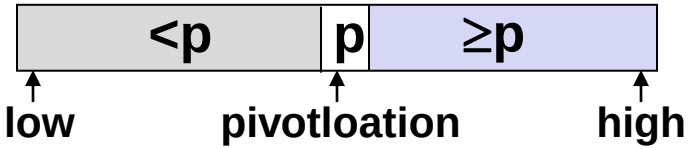
- 两部分用作存放两个初始归并段, 即各自读入前m个(250个)记录至输入Buffer, 若其中一路的m个记录已归并完成, 则读入这一路剩下的前m个记录.如此重复.
- 一部分用作归并排序的辅助空间, 即输出Buffer, 若满(>m个 或 250个记录)则转储/写入至外存上的数据文件/归并段.

Quick Sort / 快速排序

思路: 以当前表中某个记录的key值作为pivot(基准值), 通过一趟排序将表中的记录分割成两个独立部分, 其中一部分记录的关键字均比另一部分记录的关键字小. 再对这两部分继续按上述思路操作. 如此循环, 直至整个序列有序.

i.e. After each pass, we will get the target:

⇧ 分治法




```
void QuickSort(List L, int low, int high)
{ if (low<high ) { pivotpos=Partition(L,low,high);
                   QuickSort(L,low, pivotpos-1);
                   QuickSort(L,pivotpos+1,high); } }
```

关键问题:
- How to choose a pivot?
- How partition? / How to get the pivotposition(分区的索引/下标)?

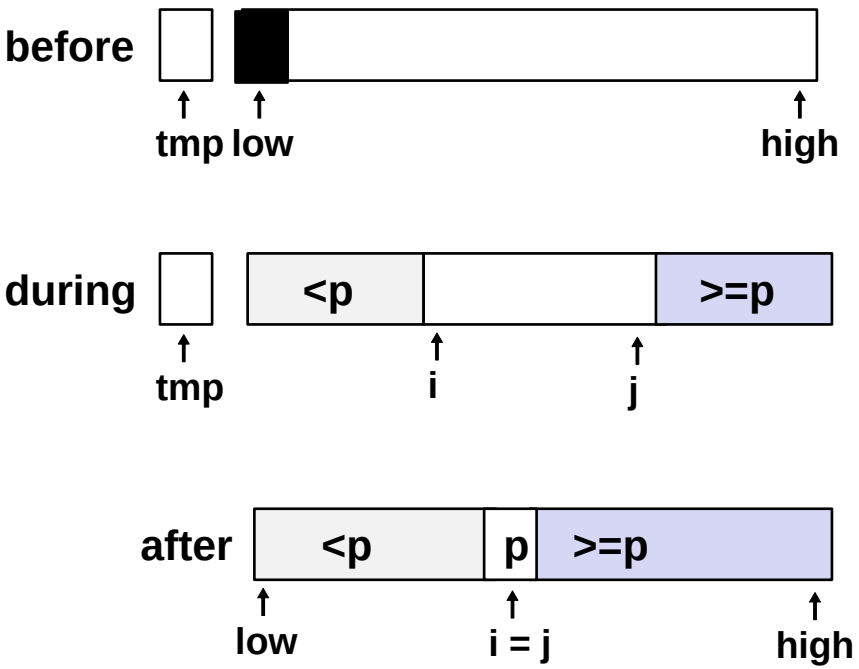
pivot的选择策略是能够使得当前表被划分成两个长度相当的子表.
理论上来说无序表中任何一个记录的关键字均可用作pivot.
简单起见通常使用当前表中的第一个或最后一个记录的key值作为pivot.

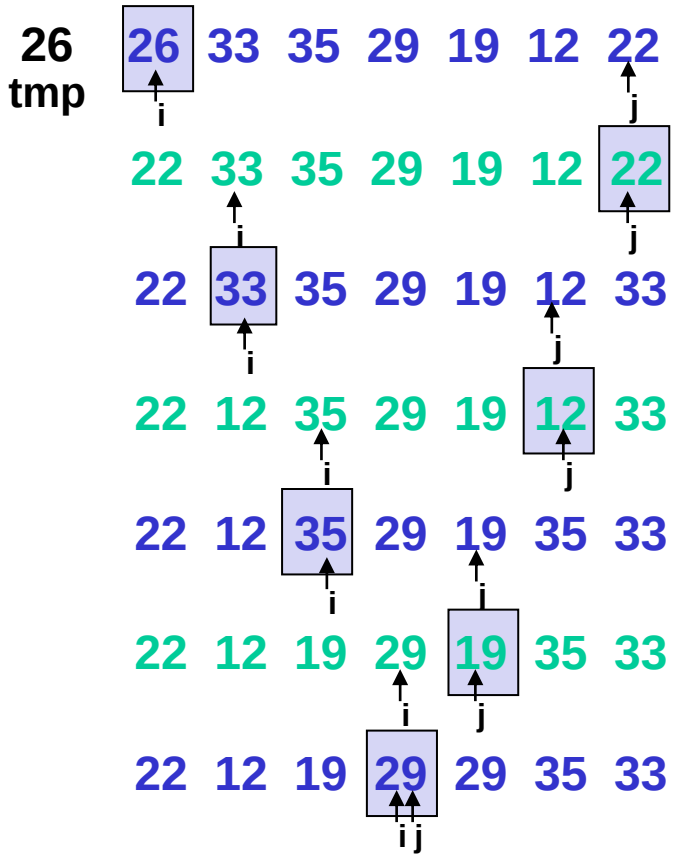
获得分区索引(下标)的值, 有不同方法.

Quick Sort / 快速排序

How to get the pivotposition(分区的索引/下标)? 假设用第1个记录的关键字值作为基准值.

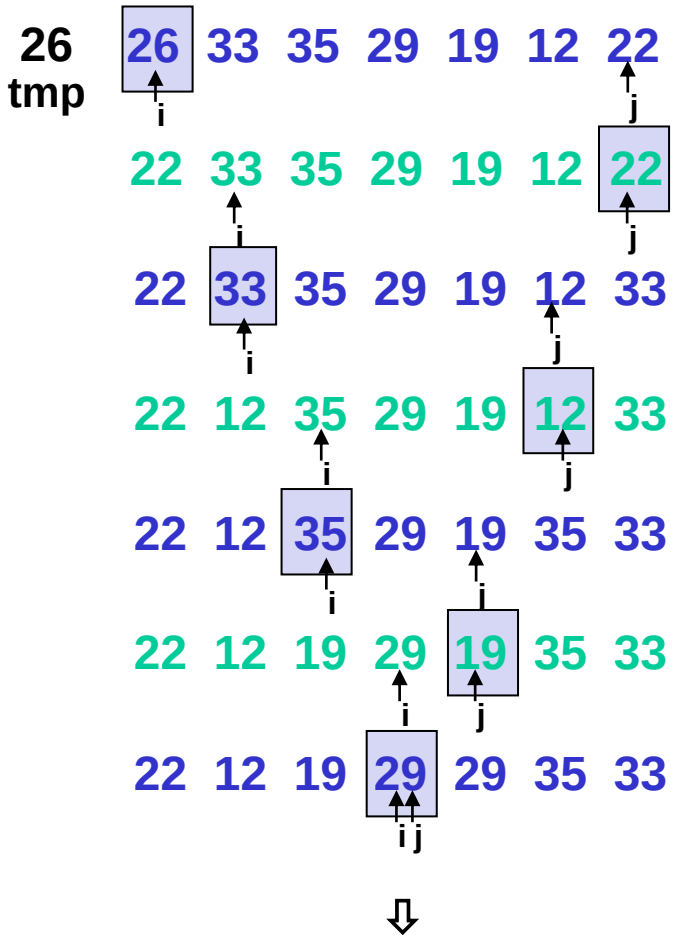
- 初始化: **pivot = L.r[1].key; tmp = L.r[1];** //第1个记录的位置相当于一个坑
  i = low; j = high; //分别用索引变量指针i,j控制从左向右和从右向左扫描
- 处理:
  - 先从右向左/j--, 若表中右侧记录L.[j]的关键字值<pivot, 则L.r[i++]=L.[j]
  - 再从左向右/i++, 若表中左侧记录L.[i]的关键字值>pivot, 则L.r[j--]=L.[i]
  - 若指针相遇/i=j, 则该位置i就是分区索引值, 即 L.r[i] = tmp; 且 返回索引指针i的值





Quick Sort / 快速排序

快速排序的算法描述: P275-276/A.10.8~7~6(b)



⇩

[22 12 19] 26 [29 35 33]

[19 12] 22 26 29 [35 33]

12 19 22 26 29 33 35

```
int Partition(List L, int low, int high) {
    int i = low; j = high;
    pivot = L.r[low].key; tmp = L.r[low];
    while (i < j) {
        while ((i < j) && L.r[j] >= pivot) j--; //从右向左
                                    //找第1个小于基准值的记录
        if (i < j) L.r[i++] = L.r[j]; //填坑&L.r[j]成为新坑
        while ((i < j) &&L.r[i] <= pivot) i++; //从左向右
                                    //找第1个大于基准值的记录
        if (i < j) L.r[j--] = L.r[i];
    }
    L.r[i] = tmp;
    return i;
}
```

```
void QuickSort(List L, int low, int high) {
    if (low >= high) return;
    pivotpos = Partition(L, low, high);
    QuickSort (L, low, pivotpos-1);
    QuickSort (L, pivotpos+1, high);
}
```

Quick Sort / 快速排序

Analysis of QuickSort:

记**C(n) =** 关键字比较个数**, n =** 表长
则**C(n) = n-1 + C(r) + C(n-r-1).** 其中**n-1**为**pivot**和表中其它关键字比较次数**,**
C(r)、**C(n-r-1)**为两个子表进行递归分区排序的比较次数**(r**、**n-r-1**分别是两**r**的值个子表的长度**)** ⇨计算**C(n)**就需知道 **r**的值

① C(n) of worst-case
对**QuickSort**来说**,** 最坏情况是指 ⇨ **one sublist has n-1 entries**
the other is empty
所以上述 **C(n) = n-1 + C(n-1)**
∵ C(1) = 0
C(2) = 1 + C(1) = 1
C(3) = 2 + C(2) = 2 + 1
C(4) = 3 + C(3) = 3 + 2 + 1 ⋯
∴ C(n) = n-1 + C(n-1) = (n-1) + (n-2) + ⋯ + 1 = n(n-1)/2 = $n^2/2$+O(n)

从关键字比较角度**,** **QuickSort**在最坏情况下的比较次数和**SelectionSort**相同**,**
而**SelectionSort** 是以冗余比较次数为代价换取记录移动次数的减少**.**

Quick Sort / 快速排序

Analysis of QuickSort:

② C(n) of best-case

对**QuickSort**来说, 最好情况是指 ⇨ 两个分区子表的长度相当.

所以上述 $C(n) = n-1 + 2C(n/2) <= n + 2C(n/2)$

$<= n + 2(2C(n/4)+n/2) = 2n + 4C(n/4)$

$<= 2n + 4(2C(n/8)+n/4)) = 3n + 8C(n/8)$

$<= \ldots$

$<= (\log_2 n)*n + n*C(1)$, 其中$C(1) = 0$

$= n\log_2 n$

③ C(n) of average-case

对**QuickSort**来说, 一般情况是指 ⇨ **all possible orderings of the list are equally likely.** 为方便说明, 用**p**表示**pivot**, 用**1~n**分别表示记录**key**值.

即 **(1, 2, … , p-1) p (p+1, p+2, … , n).**

⇨**Counting Comparisons**

$C(n) = n-1 + C(p-1) + C(n-p) = n + 2(C(0)+ C(1)+ \ldots +C(n-1)) /n$(∵ average the left expression for p=1 to n)

$\approx 1.3n\log_2 n + O(n)$

⇨ **Counting assignments**

同样思路和数学方法计算得到平均移动次数 $M(n) \approx 0.69n\log_2 n + O(n)$

Quick Sort / 快速排序

Analysis of QuickSort:

④该算法是非就地排序算法, 其空间复杂性(递归隐式栈的开销):

Worst-case: 即“递归树”为单枝树, 其高度为n, 故所需辅助的栈空间为O(n)

Best-case & Average-case: 即每次分区, 子表长度均大体相当, 这时“递归树”是平衡树, 其高度为$\lceil \log_2 n \rceil$, 因此所需辅助的栈空间为$O(\log_2 n)$

算法稳定性:以$\{ 2^1, 2^2, 1 \}$为例说明, 该算法不稳定.

Heaps and HeapSort / 堆及堆排序

Background: R.Floyd&J.Williams提出; 堆是对SelectionSort方法的改进(冗余比较).

Prerequisite:P124 ①Definition of complete binary tree; ②二叉树性质5.

Definition of a heap: P279

设有n个记录的序列$\{ R(k_1), R(k_2), \ldots, R(k_n) \}$, 当且仅当满足下列关系时称之为堆:

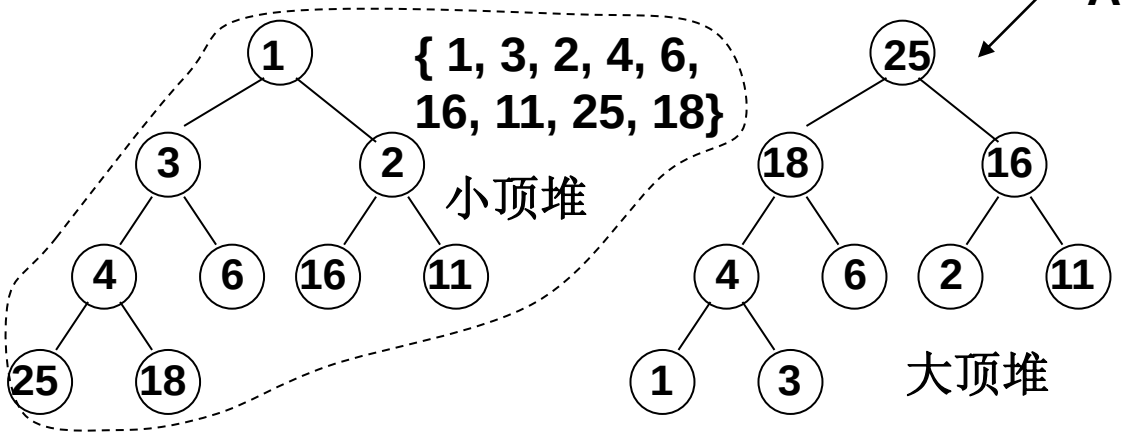
$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \text{(小顶堆/Min Heap)} \quad \text{或} \quad \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \text{(大顶堆/Max Heap)}$$

其中$i=1, 2, \ldots, \lfloor n/2 \rfloor$

A heap is a list in which each entry contains a key, and, for all positions *i* in the list, the key at position *i* is at least as small / large as the keys in positions *2i* and *2i+1*, provided these positions exist in list.

例如:设有{25, 18, 16, 4, 6, 2, 11, 1, 3}

An example of a heap:
A heap as a list and as a tree



注: 有些称之为二叉堆(Binary heap). 是完全二叉树. 用数组表示(简便&效率高).

Heaps and HeapSort / 堆及堆排序

HeapSort的思路:

将n个记录存于顺序结构的线性表中, 并把它“看成”是一个完全二叉树, 而此时二叉树中的关键字序列不一定满足堆的定义, 因此

- 如何将一个无序序列变成一个堆? ⇨建初堆(Building the initial heap)
- 如何在输出堆顶记录后, 调整剩余记录成为一个新的堆? ⇨堆排序

} 堆调整

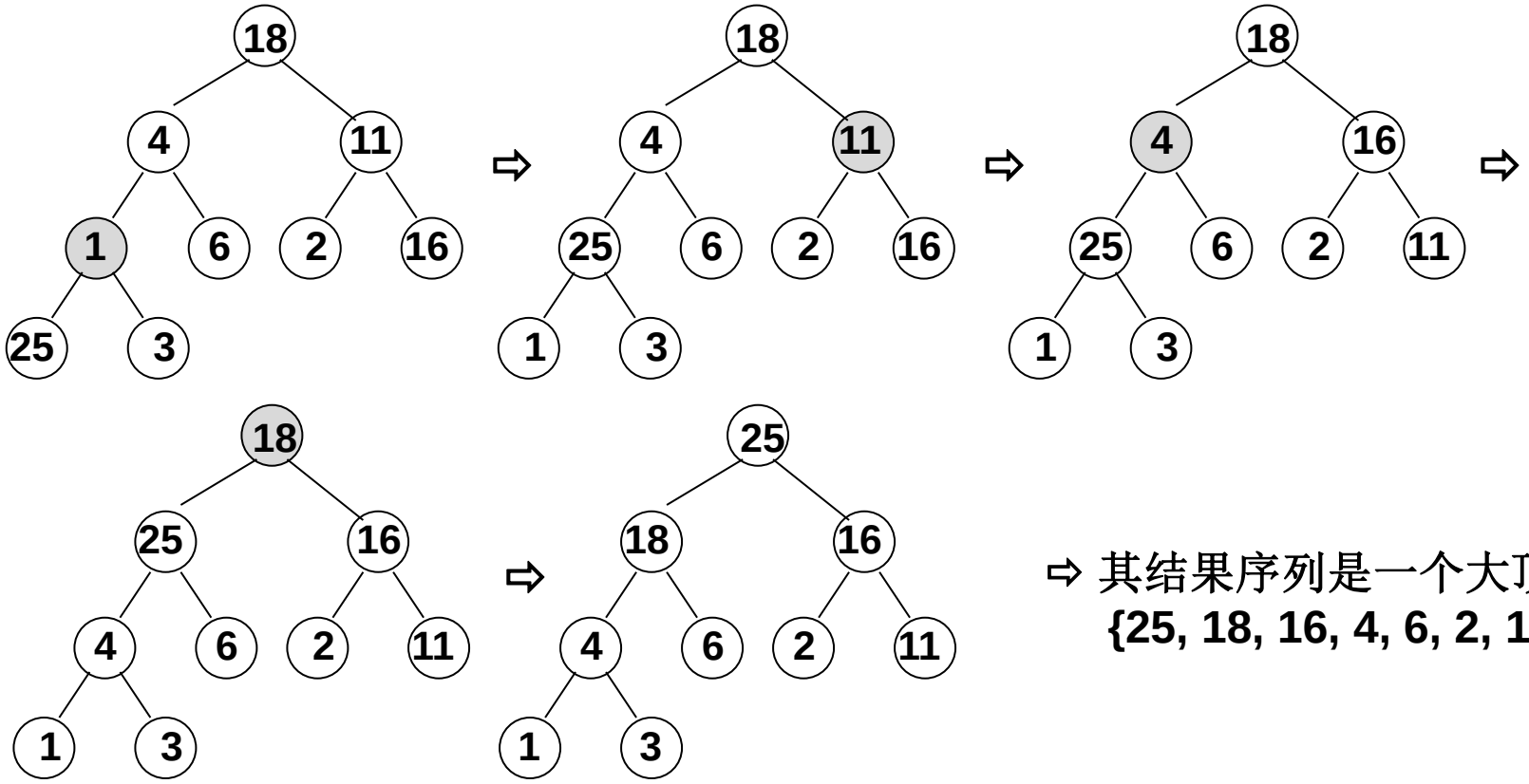
HeapAdjust / Heapify

从一个无序序列建初堆的过程就是反复“筛选”的过程. 若将此序列看成是一个完全二叉树, 则最后一个非叶结点是第$\lfloor n/2 \rfloor$个记录, 即从这个元素起向下进行“筛选”, 如此直至第1个记录.

堆排序过程中的“调整”, 是对剩余记录从当前“树根”结点起向下沿某个分支进行“筛选”, 直至某个叶结点为止. 若该过程中, 在“树”中某一层没有出现“筛选”时, 则该趟“调整”结束.

Heaps and HeapSort / 堆及堆排序

建初堆:设有初始序列{18, 4, 11, 1, 6, 2, 16, 25, 3}, 用堆调整方法建“大顶堆”

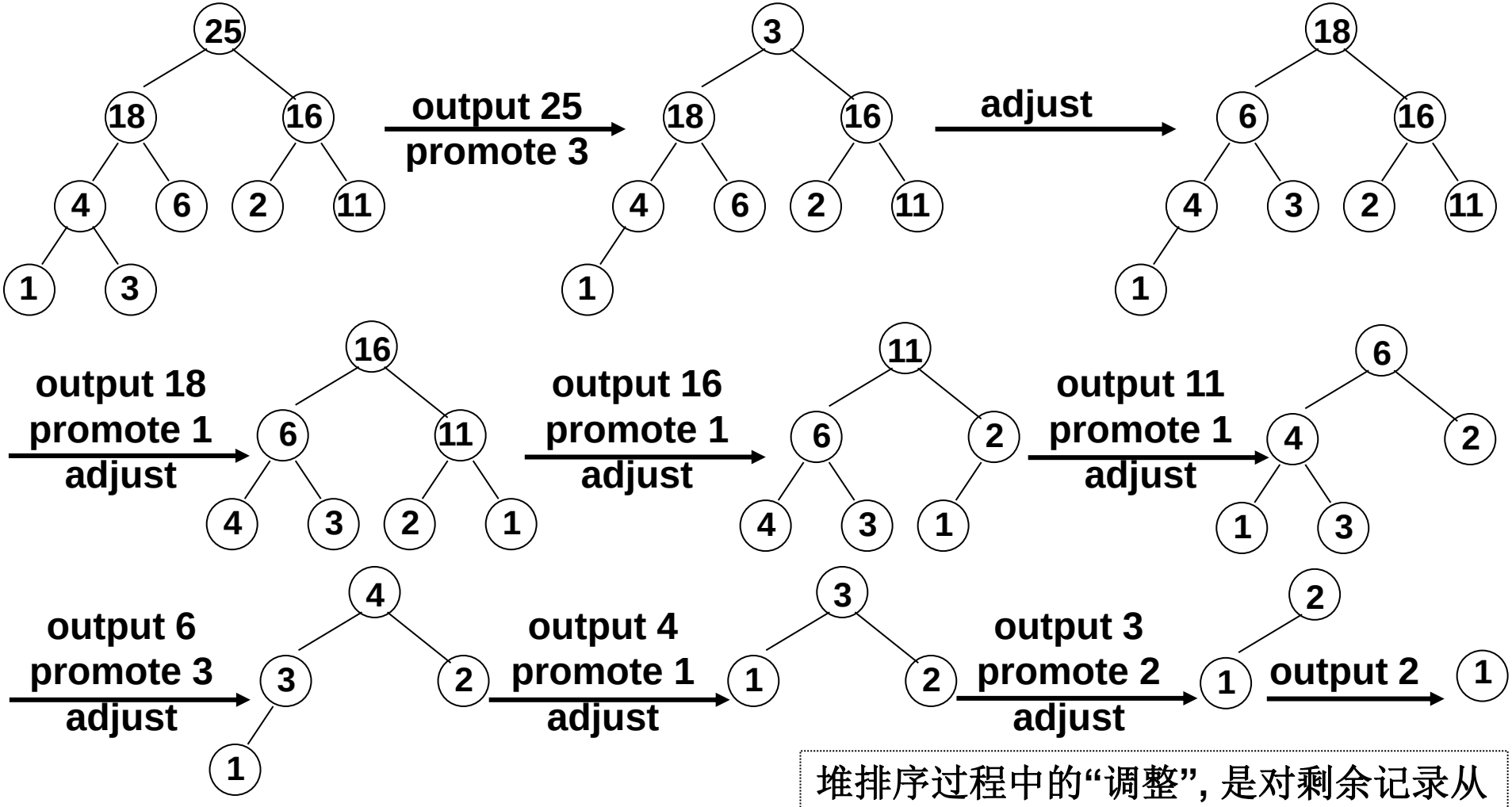


⇨ 其结果序列是一个大顶堆:
{25, 18, 16, 4, 6, 2, 11, 1, 3}

从一个无序序列建初堆的过程就是反复“筛选”的过程. 若将此序列看成是一个完全二叉树, 则最后一个非叶结点是第$\lfloor n/2 \rfloor$个记录, 即从这个元素起向下进行“筛选”, 如此直至第1个记录.

Heaps and HeapSort / 堆及堆排序

堆排序: 用堆调整方法对所建成的“大顶堆”进行排序



output 1 **1, 2, 3, 4, 6, 11, 16, 18, 25**

- 用“大顶堆”的排序结果为非降序序列
- 用“小顶堆”的排序结果为非升序序列

堆排序过程中的“调整”, 是对剩余记录从当前“树根”结点起向下沿某个分支进行“筛选”, 直至某个叶结点为止. 若该过程中, 在“树”中某一层没有出现“筛选”时, 则该趟“调整”结束.

Heaps and HeapSort / 堆及堆排序

建初堆的算法描述:

P282/A10.11中的第一个for语句;

其中堆调整的算法描述为P282/A10.10

堆排序的算法描述:

P282/A10.11中第二个for语句

```
//A.10.11:对顺序表H进行堆排序
void HeapSort(List H)
{ //首先建初堆
  for (i = H.length/2; i > 0; --i)
      HeapAdjust(H, i, H.length);
  //然后进行堆排序
  for (i = H.length; i > 1; --i) {
      Swap(H.r[1], H.r[i]);
      //将堆顶记录和当前未经排序的子序列
      //H.r[1..i]中的最后一个记录相互交换
      HeapAdjust(H, 1, i-1); }
 }
```

```
//A.10.10:已知H.r[s..m]中记录的关键字
//除H.r[s].key之外均满足堆的定义, 该函数调整
//H.r[s]的关键字, 使H.r[s..m]成为大顶堆
void HeapAdjust(List H, int s, int m)
{ rc = H.r[s];
  for ( j = 2*s; j <= m; j *= 2 ) {
   //沿key值较大的孩子结点向下筛选
    if ( j < m && H.r[j].key < H.r[j+1].key ) ++j;
    //j为key值较大的记录的下标
    if ( rc.key >= H.r[j].key ) break;
    //rc应插在位置s上
    H.r[s] = H.r[j]; s = j; }
  H.r[s] = rc;  } //插入
```

Heaps and HeapSort / 堆及堆排序

算法分析: 和SelectionSort一样, HeapSort的平均性能和worst-case性能没有太大区别.
因此这里仅分析其worst-case情形.

HeapSort = 建初堆 **+** 堆排序

建初堆: (1)考虑堆调整算法中的for循环, 每进行一次迭代时其j都翻一倍(j=j×2),
所以for循环语句的执行次数不会超过$\log_2(n/k)$ (其中n为表长, k=m downto 1
中的一个值, $m=\lfloor n/2 \rfloor$), 事实上, $\log_2(n/k)$值就是以L.r[k]为根结点的子树的高度.
(2)for语句每循环一次时, 有2次比较和1次移动.
故在HeapAdjust算法中, 至多有$2 \times \log_2(n/k)$次比较, $\log_2(n/k)$次移动.

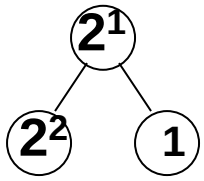
⇨建初堆总比较次数 $= 2\times\sum_{k=1}^{m} \log_2(n/k) = 2\times\sum_{k=1}^{m} (\log_2 n - \log_2 k)$

$= 2 \times (m\log_2 n - \log_2 m!) \approx 5m \approx 2.5n$ (因为$m = \lfloor n/2 \rfloor$, $\log_2 m = \log_2 n - 1$)
(Theorem: $\log_2 m! = m\log_2 m - 1.5m$)

堆排序:

⇨堆排序总比较次数 $= 2 \times \sum_{k=2}^{n} \log_2 k = 2\log_2 n! \approx 2n\log_2 n - 3n$

⇨堆排序总移动次数 $= \sum_{k=2}^{n} \log_2 k \approx n\log_2 n - 3n/2$



可见HeapSort主要以堆排序阶段所需时间为主($2n\log_2 n - 3n >> 2.5n$)

算法稳定性: 该算法不稳定: $\{2^1, 2^2, 1\}$, 如右边的初始大顶堆, 对其堆排序后得$\{1, 2^2, 2^1\}$.

Heaps and HeapSort / 堆及堆排序

HeapSort Compares with QuickSort:

	HeapSort	QuickSort(average)
Assignments of entries	$n\log_2 n + O(n)$	$0.69n\log_2 n + O(n)$
Comparisons of entries	$2n\log_2 n + O(n)$	$1.39n\log_2 n + O(n)$

Conclusions:

从平均角度, **HeapSort**要比**QuickSort**逊色, 但从**worst-case**角度, **HeapSort**要比**QuickSort**优. 因此**HeapSort is a more insurance policy.**
(因为即使在最坏情况下它不会导致算法灾难性的退化: $O(n\log_2 n) \Rightarrow O(n^2)$**)**

Sorting: Comparisons of Sorting Methods / 排序方法小结P289

Discuss: What will depend on to choose a sorting method?

The four important efficiency criteria:

- **The numbers, size, initial order, feature of keys(primary or secondary) and data structure of the unsorted entries**
- **Use of extra space**
- **Use of computing time**
- **Programming effort**

Suggestion: InsertionSort (or Sometimes ShellSort)

- ✓**easy to program and maintain (i.e. no recursion)**
- ✓**it runs efficiently for short lists**
- ✓**even for long lists, if the entries are nearly in the correct order,it'll be efficient**