10.4 选择排序

选择排序(Selection Sort)的基本思想是：每一趟在n-i+1(i=1,2, …,n-1)个记录中选取关键字最小的记录作为有序序列中的第 i个记录。

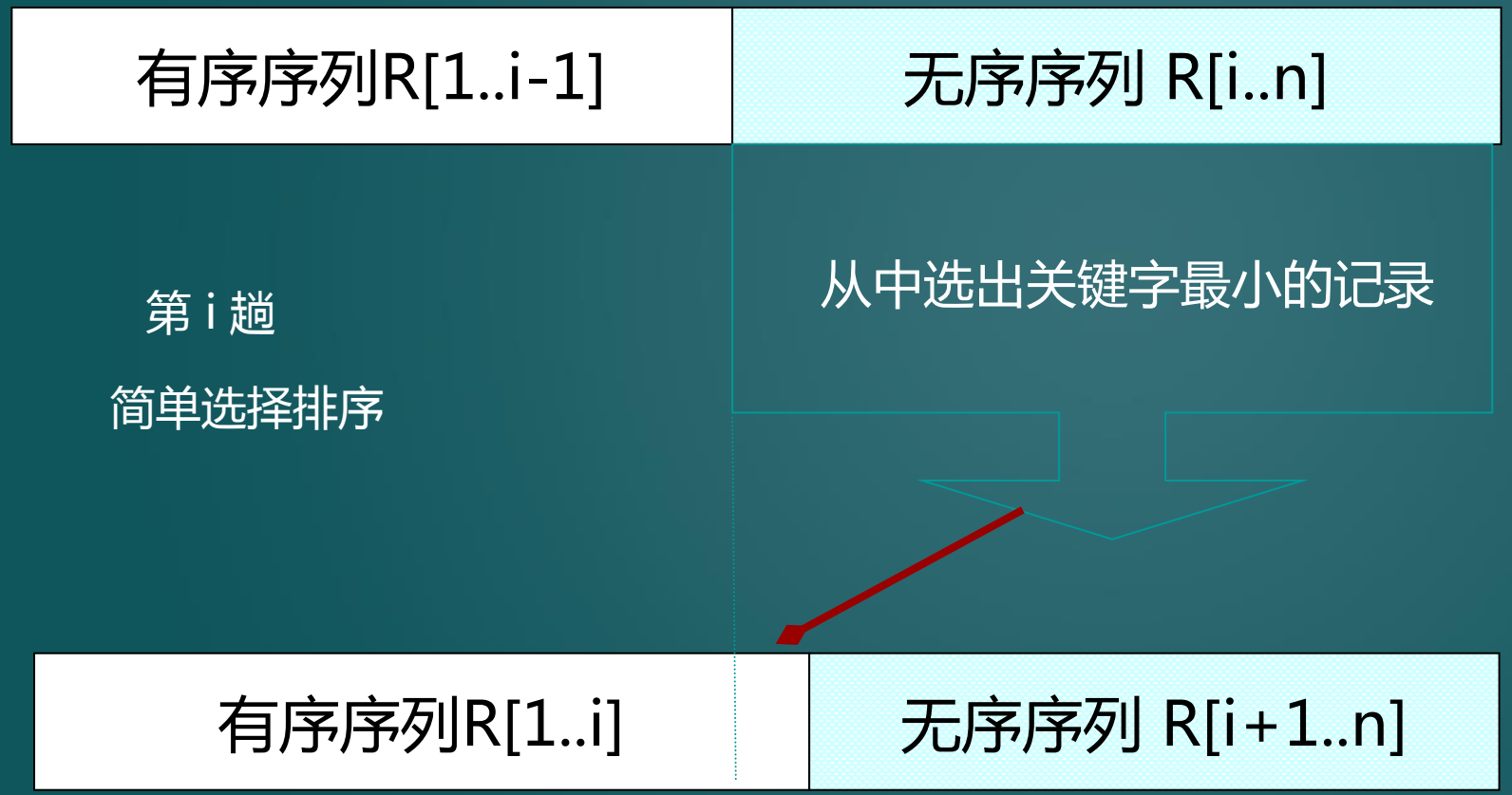
10.4.1 简单选择排序

10.4.2 树形选择排序

10.4.3 堆排序

10.4.1 简单选择排序

假设排序过程中，待排记录序列的状态为：



简单选择排序的算法描述如下：

```
void SelectSort (Sqlist  &L ) {
  // 对顺序表&L作简单选择排序。
  for (i=1; i<L.length; ++i) {
            // 选择第 i 小的记录，并交换到位
      j = SelectMinKey(L, i);
                // 在 L.r[i.. L.length] 中选择关键字最小的记录
      if (i!=j) L.r[i]←→L.r [j];   // 与第i个记录交换
  }
} // SelectSort
```

算法 10.9

时间性能分析

对 n 个记录进行简单选择排序，所需进行的**关键字间的比较次数**总计为：

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

移动记录的次数，最小值为 0, 最大值为3(n-1)。

10.4.3 堆排序（Heap Sort）

堆的定义:

堆是满足下列性质的数列$\{r_1, r_2, \ldots, r_n\}$：

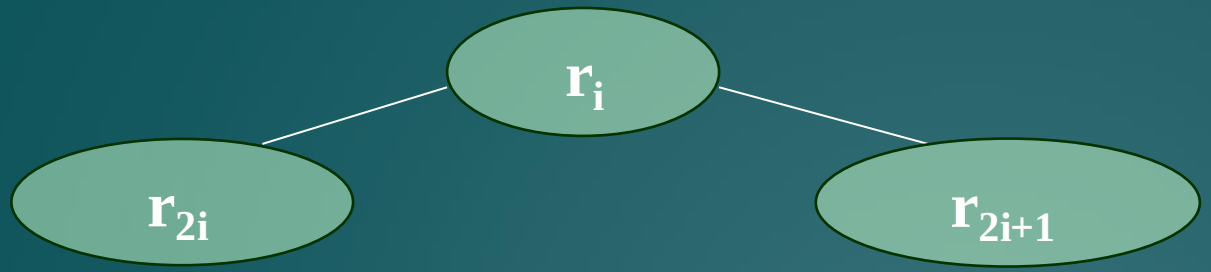
$$\begin{cases} r_i <= r_{2i} \\ r_i <= r_{2i+1} \end{cases} \text{(小顶堆)} \quad \textbf{或} \quad \begin{cases} r_i >= r_{2i} \\ r_i >= r_{2i+1} \end{cases} \text{(大顶堆)}$$

例如:

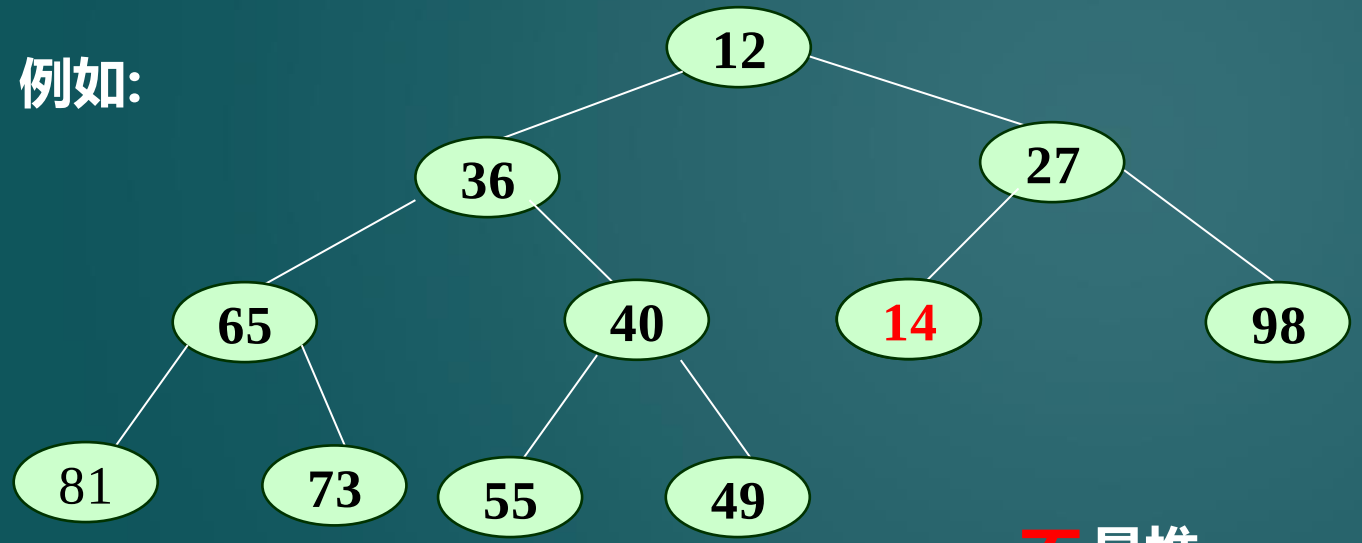
{12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49}　是小顶堆

{12, 36, 27, 65, 40, 14, 98, 81, 73, 55, 49}　不是堆

若将该数列视作完全二叉树，则 r_{2i} 是 r_i 的左孩子； r_{2i+1} 是 r_i 的右孩子。



例如:



不是堆

堆排序即是利用堆的特性对记录序列进行排序的一种排序方法。

以大堆顶为例，若在输出堆顶的最大值之后，使得剩余的n-1个元素的序列重又建成一个堆，则得到n个元素中的次大值，如此反复执行，便能得到一个有序序列，这个过程称之为堆排序。

例如：

{ 40, 55, 49, 73, 12, 27, 98, 81, 64, 36 }

建大顶堆

{ 98, 81, 49, 73, 36, 27, 40, 55, 64, 12 }

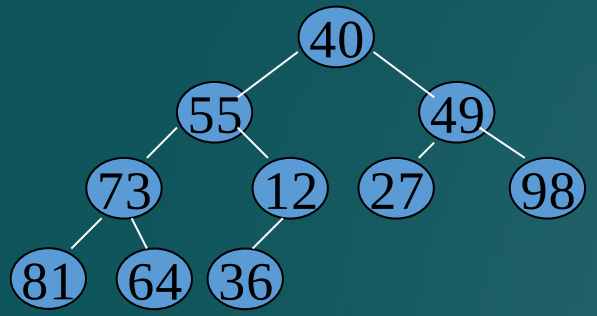
交换 98 和 12

{ 12, 81, 49, 73, 36, 27, 40, 55, 64, 98 }

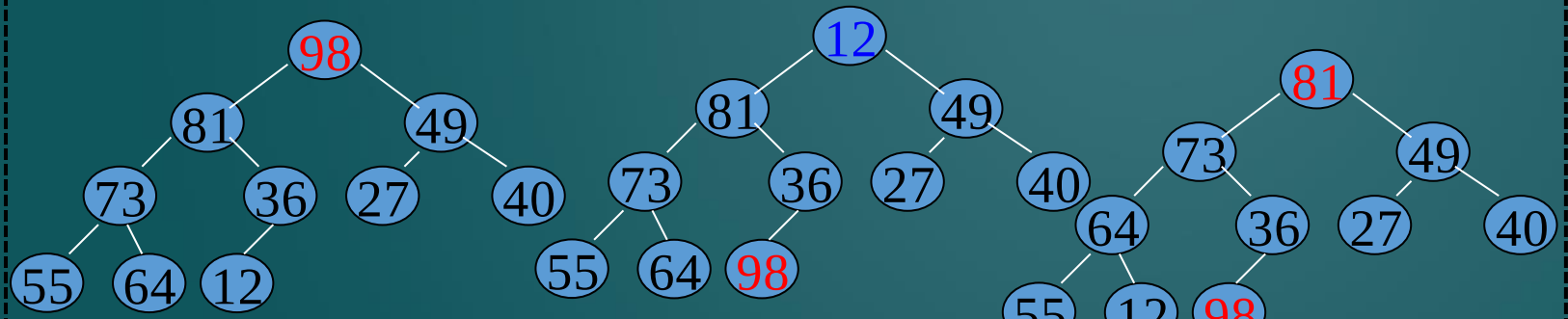
经过筛选

重新调整为大顶堆

{ 81, 73, 49, 64, 36, 27, 40, 55, 12, 98 }



初始序列



初始堆

交换98和12

重调整为堆

定义堆类型为:

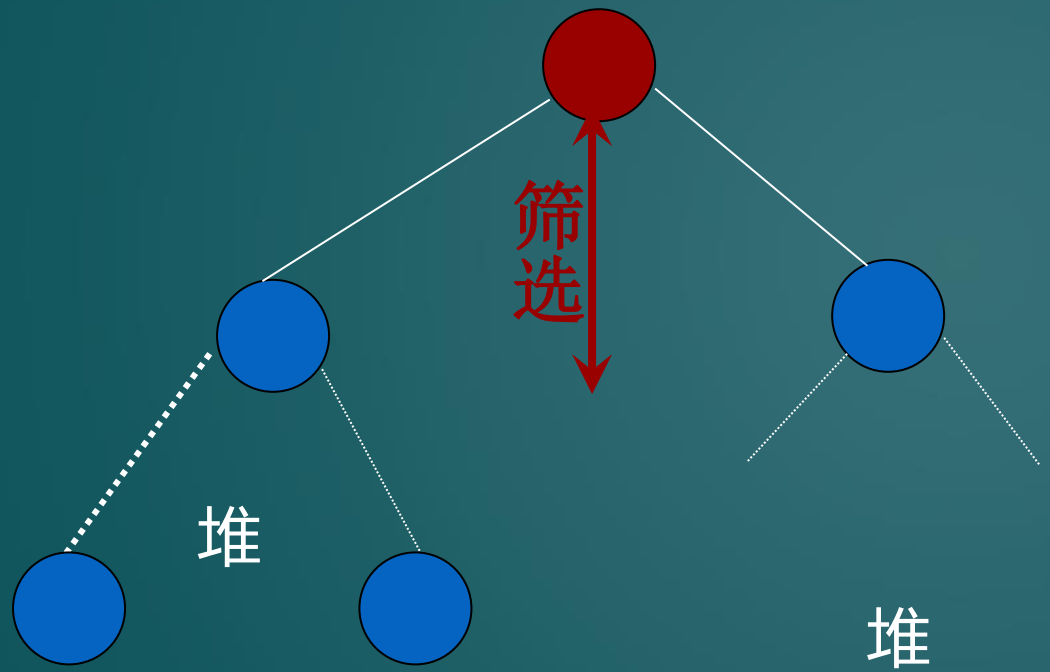
```
typedef  SqList HeapType;

                                    // 堆采用顺序表表示之
```

两个问题:

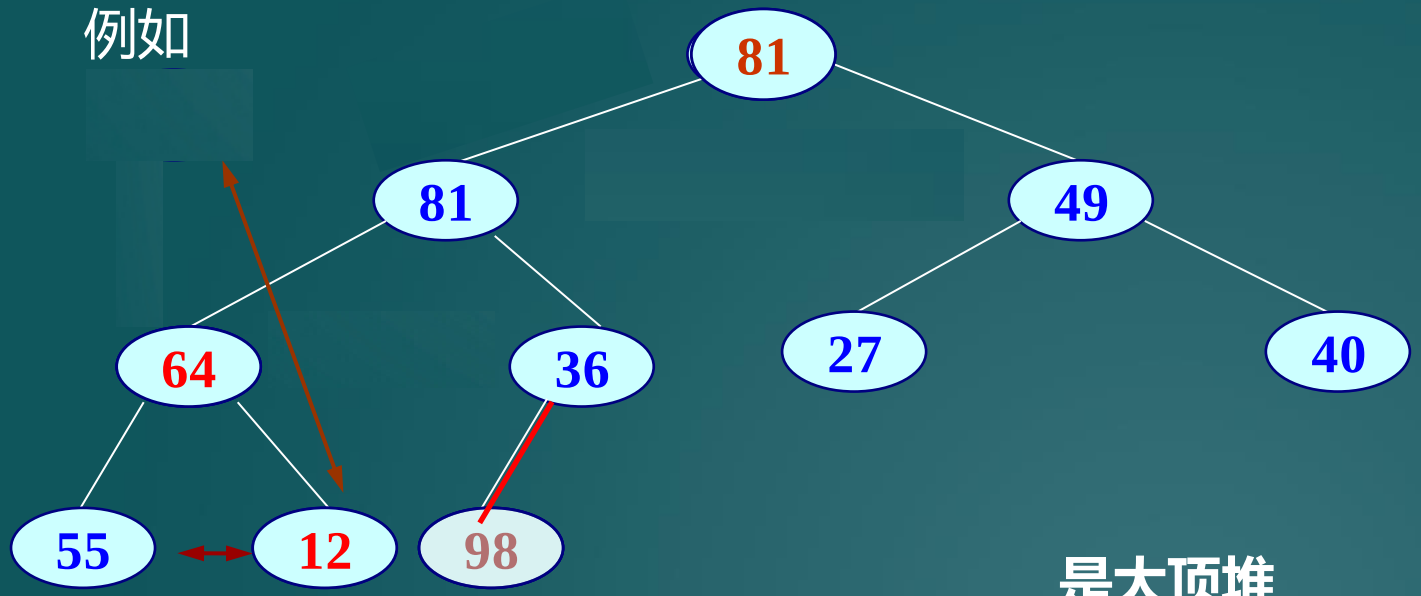
1、如何由一个无序序列“建堆”？

2、如何在输出堆顶元素之后，调整剩余元素成为一个新的堆，即“筛选”？

所谓“**筛选**”指的是，对一棵左/右子树均为堆的完全二叉树，“**调整**”**根结点**使整个二叉树也成为一个堆。



例如



是大顶堆

但在 98 和 12 进行互换之后，它就**不**是堆了，

因此，需要对它进行“筛选”。

```
void HeapAdjust (HeapType &H, int s, int m)
{   // 已知H.r[s..m]中记录的关键字除 H.r [s] 之外均满足堆的定
   //义，本函数自上而下调整 H.r[s] 的关键字，使 H.r[s..m] 也
   //成为一个大顶堆
   rc = H.r[s];    // 暂存 H.r[s]
   for ( j=2*s; j<=m; j*=2 ) { // 沿key较大的孩子向下筛选
      if ( j<m && H.r [j].key<H.r [j+1].key ) ++j; //j为key较大的
                                                   //记录的下标
      if ( rc.key >= H.r[j].key ) break;  //rc应插入在位置s上
      H.r [s] = H.r [j];    s = j;
       }
   H.r [s] = rc;  // 将调整前的堆顶记录插入到s位置
} // HeapAdjust
```

算法 10.10

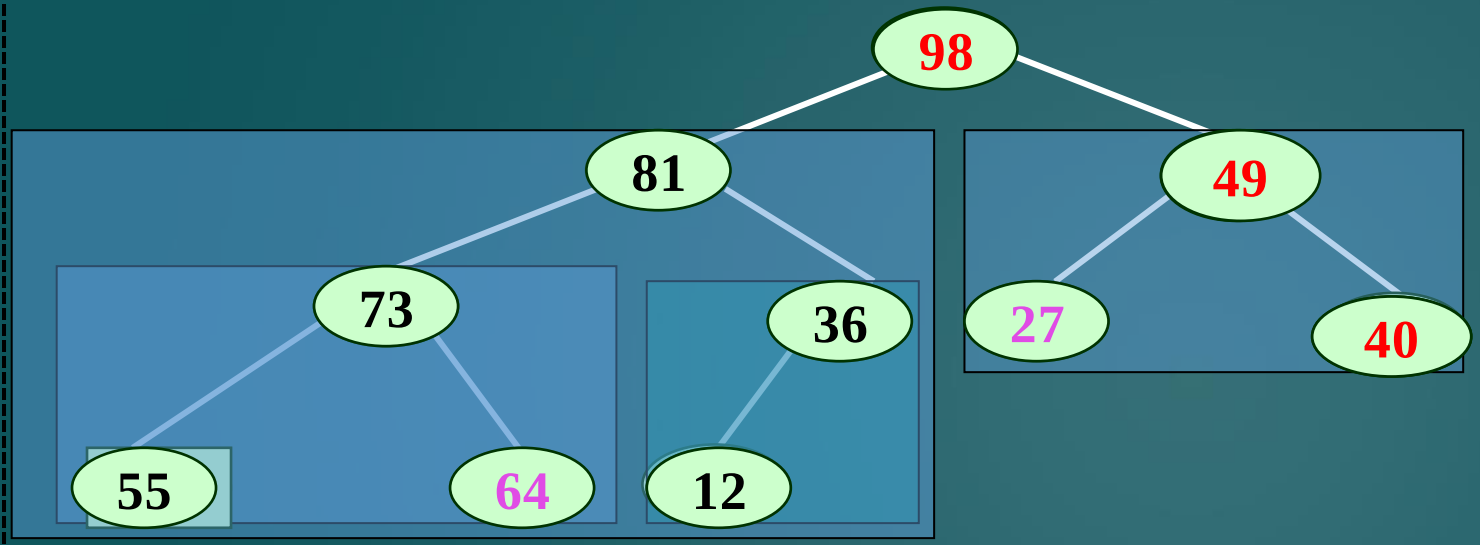
```
if ( j<m && R[j].key<R[j+1].key ) ++j;
        // 左/右“子树根”之间先进行相互比较
        // 令 j 指示关键字较大记录的位置

  if ( rc.key >= R[j].key ) break;
        // 再作“根”和“子树根”之间的比较，
        // 若“>=”成立，则说明已找到 rc 的插
        // 入位置 s ，不需要继续往下调整

    R[s] = R[j];   s = j;
        // 否则记录上移，尚需继续往下调整
```

```
void HeapSort ( HeapType &H ) {
  // 对顺序表 H 进行堆排序
    for ( i=H.length/2;   i>0;   --i )
HeapAdjust ( H, i, H.length );    // 建大顶堆
    for ( i=H.length; i>1; --i ) {
        H.r[1]←→H.r[i];
        // 将堆顶记录和当前未经排序子序列
         //  H.r[1..i]中最后一个记录相互交换
      HeapAdjust(H, 1, i-1);  // 对 H.r[1..i-1] 进行筛选
     }
} // HeapSort
```

算法 10.11

建堆是一个从下往上进行“筛选”的过程。

例如: 排序之前的关键字序列为



现在，左/右子树都已经调整为堆，最后只要调整根结点，使整个二叉树是个“堆”即可。

堆排序的时间复杂度分析：

1. 对深度为 k 的堆，“筛选”所需进行的关键字比较的次数至多为 2(k-1)；

2. 对 n 个关键字，建成深度为 $h=(\lfloor \log_2 n \rfloor+1)$ 的堆，所需进行的关键字比较的次数至多 $4n$；

3. 调整“堆顶” $n-1$ 次，总共进行的关键字比较的次数 不超过

$$2(\lfloor \log_2(n-1) \rfloor+\lfloor \log_2(n-2) \rfloor+\ldots+\log_2 2) < 2n(\lfloor \log_2 n \rfloor)$$

因此，堆排序的时间复杂度为 $O(n\log n)$。