数据结构

深圳技术大学
大数据与互联网学院

第十章 内部排序

10.3 快速排序

二. 快速排序

- 算法设计：
  - 任取待排序记录序列中的某个记录(例如取第一个记录)作为基准(枢)，按照该记录的关键字大小，将整个记录序列划分为左右两个子序列：
  - 左侧子序列中所有记录的关键字都小于或等于基准记录的关键字
  - 右侧子序列中所有记录的关键字都大于基准记录的关键字
  - 基准记录则排在这两个子序列中间(这也是该记录最终应安放的位置)
  - 然后分别对这两个子序列重复施行上述方法，直到所有的记录都排在相应位置上为止。

  - 基准记录也称为枢轴（或支点）记录。

10.3 快速排序

二. 快速排序

- 算法过程：
  - 取序列第一个记录为枢轴记录，其关键字为Pivotkey
  - 指针low指向序列第一个记录位置
  - 指针high指向序列最后一个记录位置
  - 一趟排序(某个子序列)过程

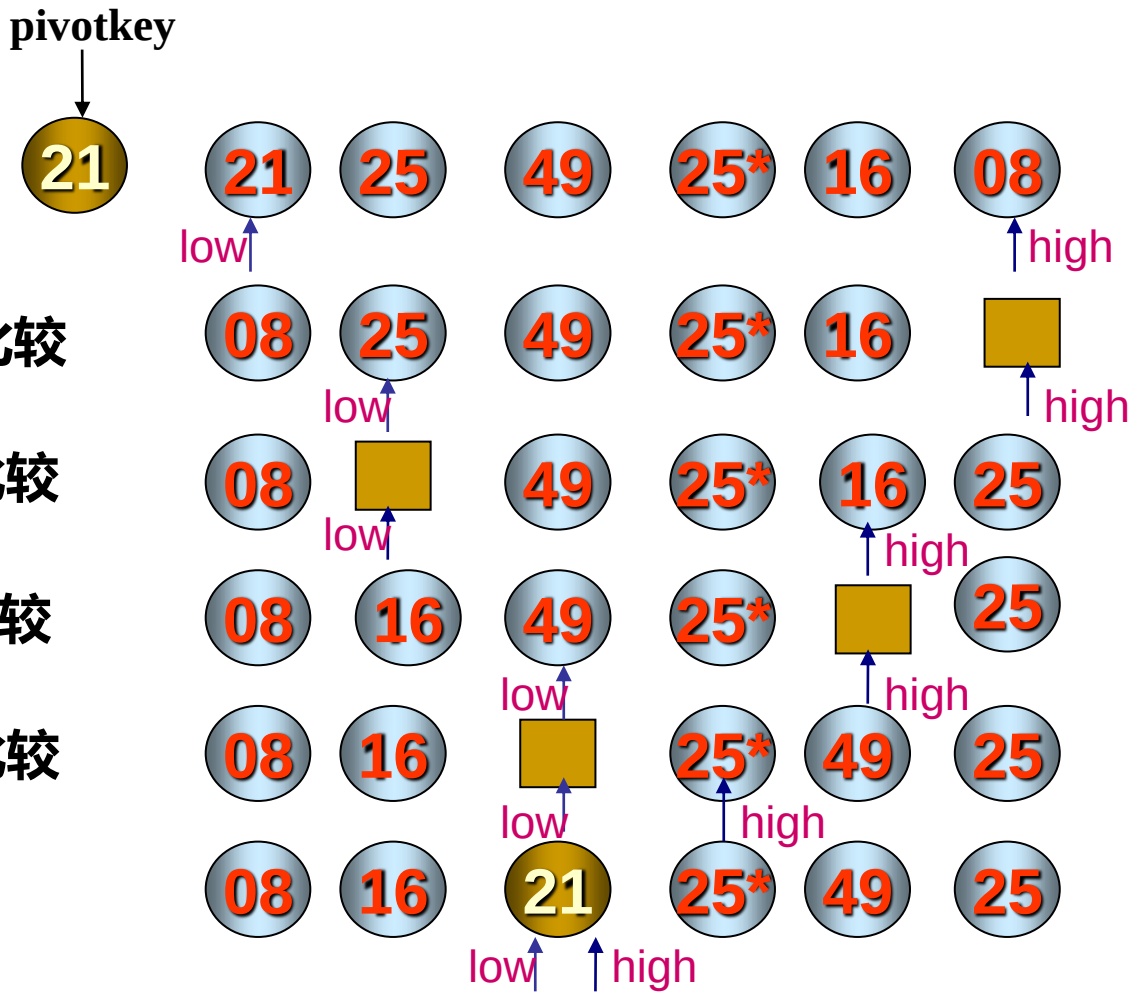
1. 从high指向的记录开始,向前找到第一个关键字的值小于Pivotkey的记录,将其放到low指向的位置,low+1
2. 从low指向的记录开始,向后找到第一个关键字的值大于Pivotkey的记录,将其放到high指向的位置,high-1
3. 重复1,2，直到low=high，将枢轴记录放在low(high)指向的位置

  - 对枢轴记录前后两个子序列执行相同的操作，直到每个子序列都只有一个记录为止

10.3 快速排序

二. 快速排序

- 举例



10.3 快速排序

二. 快速排序

- 举例

完成一趟排序

分别进行快速排序

有序序列

10.3 快速排序

二. 快速排序

- 算法实现

```
int Partition(SqList &L, int low, int high) {  // 算法10.6(a)
   // 交换顺序表L中子序列L.r[low..high]的记录，使枢轴记录到位，
   // 并返回其所在位置，此时，在它之前（后）的记录均不大（小）于它
   KeyType pivotkey;
   KeyType temp;
   pivotkey = L.r[low].key;        // 用子表的第一个记录作枢轴记录
   while (low<high) {              // 从表的两端交替地向中间扫描
      while (low<high && L.r[high].key>=pivotkey) --high;
      temp=L.r[low];
      L.r[low]=L.r[high];
      L.r[high]=temp;              // 将比枢轴记录小的记录交换到低端
      while (low<high && L.r[low].key<=pivotkey) ++low;
      temp=L.r[low];
      L.r[low]=L.r[high];
      L.r[high]=temp;              // 将比枢轴记录大的记录交换到高端
   }
   return low;                     // 返回枢轴所在位置
} // Partition
```

10.3 快速排序

二. 快速排序

- 算法实现

```
//算法a改进
int Partition(SqList &L, int low, int high) {  // 算法10.6(b)
   // 交换顺序表L中子序列L.r[low..high]的记录，使枢轴记录到位，
   // 并返回其所在位置，此时，在它之前（后）的记录均不大（小）于它
   KeyType pivotkey;
   L.r[0] = L.r[low];                  // 用子表的第一个记录作枢轴记录
   pivotkey = L.r[low].key;            // 枢轴记录关键字
   while (low<high) {                  // 从表的两端交替地向中间扫描
      while (low<high && L.r[high].key>=pivotkey) --high;
      L.r[low] = L.r[high];            // 将比枢轴记录小的记录移到低端
      while (low<high && L.r[low].key<=pivotkey) ++low;
      L.r[high] = L.r[low];            // 将比枢轴记录大的记录移到高端
   }
   L.r[low] = L.r[0];                  // 枢轴记录到位
   return low;                         // 返回枢轴位置
} // Partition
```

10.3 快速排序

二. 快速排序

- 程序实现

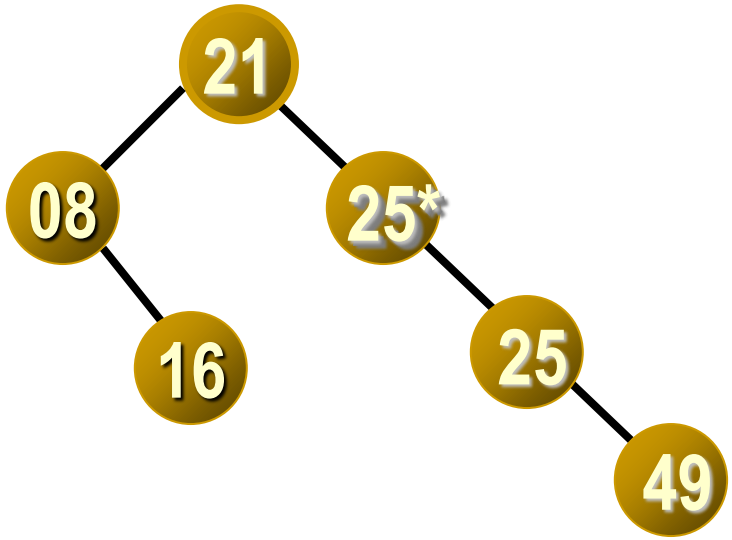
```
int Partition(SqList &L, int low, int high) {   // 算法10.6(b)
    // 交换顺序表L中子序列L.r[low..high]的记录，使枢轴记录到位，
    // 并返回其所在位置，此时，在它之前（后）的记录均不大（小）于它
    … …
}
void QSort(SqList &L, int low, int high) {   //算法10.7
  // 对顺序表L中的子序列L.r[low..high]进行快速排序
  int pivotloc;
  if (low < high) {                              // 长度大于1
    pivotloc = Partition(L, low, high); // 将L.r[low..high]一分为二
    QSort(L, low, pivotloc-1); // 对低子表递归排序，pivotloc是枢轴位置
    QSort(L, pivotloc+1, high);                  // 对高子表递归排序
  }
} // Qsort

void QuickSort(SqList &L) {   // 算法10.8
    // 对顺序表L进行快速排序
    QSort(L, 1, L.length);
} // QuickSort
```

10.3 快速排序

二. 快速排序

- 快速排序是一个递归过程，其递归树如图所示
- 利用序列第一个记录作为基准，将整个序列划分为左右两个子序列。只要是关键字小于基准记录关键字的记录都移到序列左侧



10.3 快速排序

二. 快速排序

- 在n个元素的序列中，对一个记录定位所需时间为O(n)。若设 t(n) 是对 n 个元素的序列进行排序所需的时间，而且每次对一个记录正确定位后，正好把序列划分为长度相等的两个子序列，此时，总的计算时间为：

$T(n) \le cn + 2T(n/2)$ // c 是一个常数

$\le cn + 2(cn/2 + 2T(n/4)) = 2cn + 4T(n/4)$

$\le 2cn + 4(cn/4 + 2T(n/8)) = 3cn + 8T(n/8)$

.....

$\le cn \log_2 n + nT(1) = O(n \log_2 n)$

10.3 快速排序

二. 快速排序

- 可以证明，快速排序的平均计算时间是$0(n\log_2 n)$
- 实验结果表明：就平均计算时间而言，快速排序是所有内排序方法中最好的一个
- 但快速排序是一种不稳定的排序方法

10.3 快速排序

二. 快速排序

- 在最坏情况下，即待排序记录序列已经按其关键字从小到大排好序，其递归树成为单支树，
- 每次划分只得到一个比上一次少一个记录的子序列，必须经过n-1 趟才能把所有记录定位，而且第 i 趟需要经过 n-i 次关键字比较才能找到第 i 个记录的安放位置，总的关键字比较次数将达到

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

- 改进：枢轴记录取low、high、(low+high)/2三者指向记录关键字居中的记录

10.4 选择排序

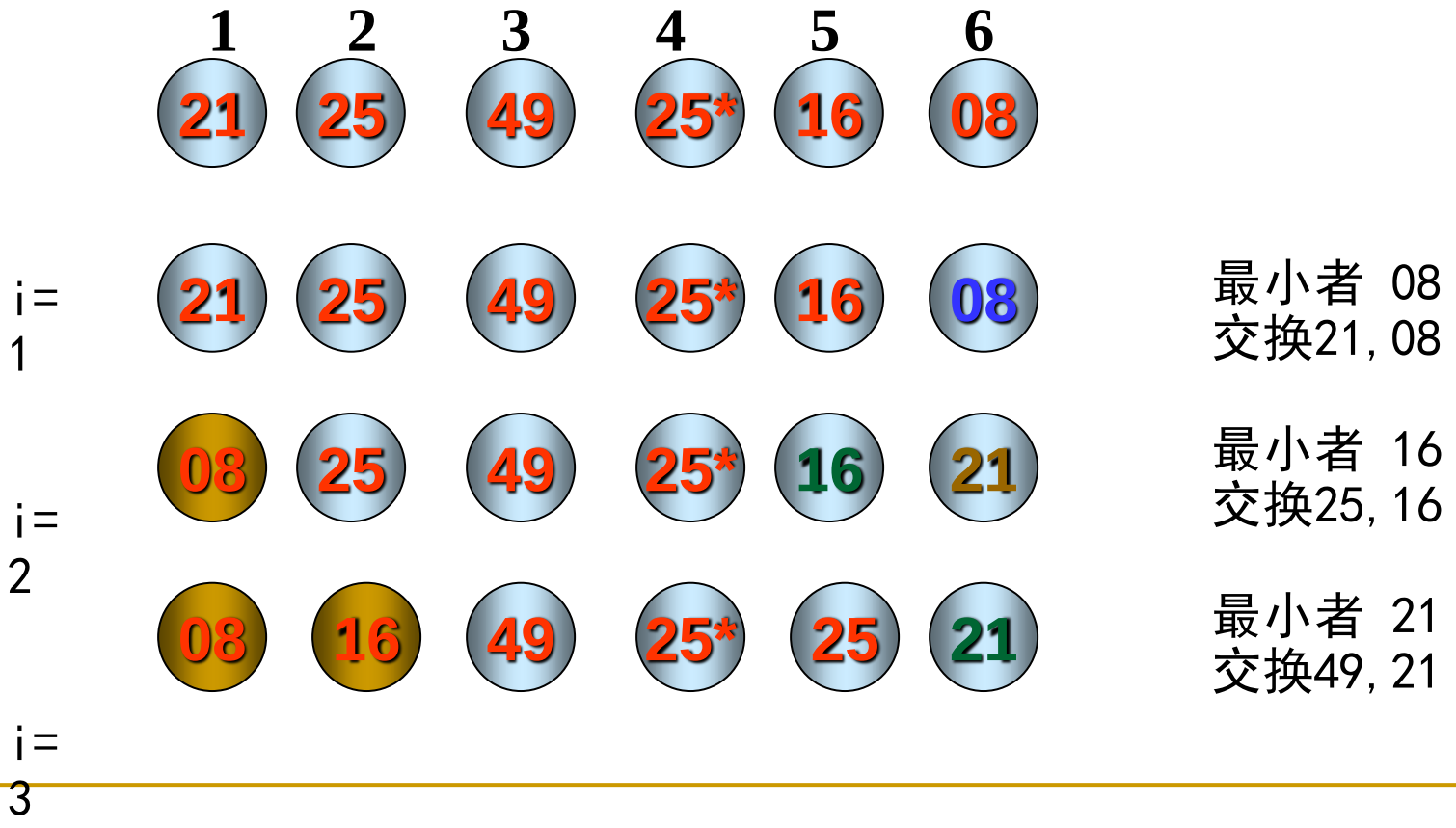
一. 简单选择排序

- 算法设计：
  - 每一趟(例如第i趟, i=1, 2, ..., n-1)在后面 n-i+1个待排序记录中通过n-i次比较，选出关键字最小的记录, 与第i个记录交换

10.4 选择排序

一. 简单选择排序

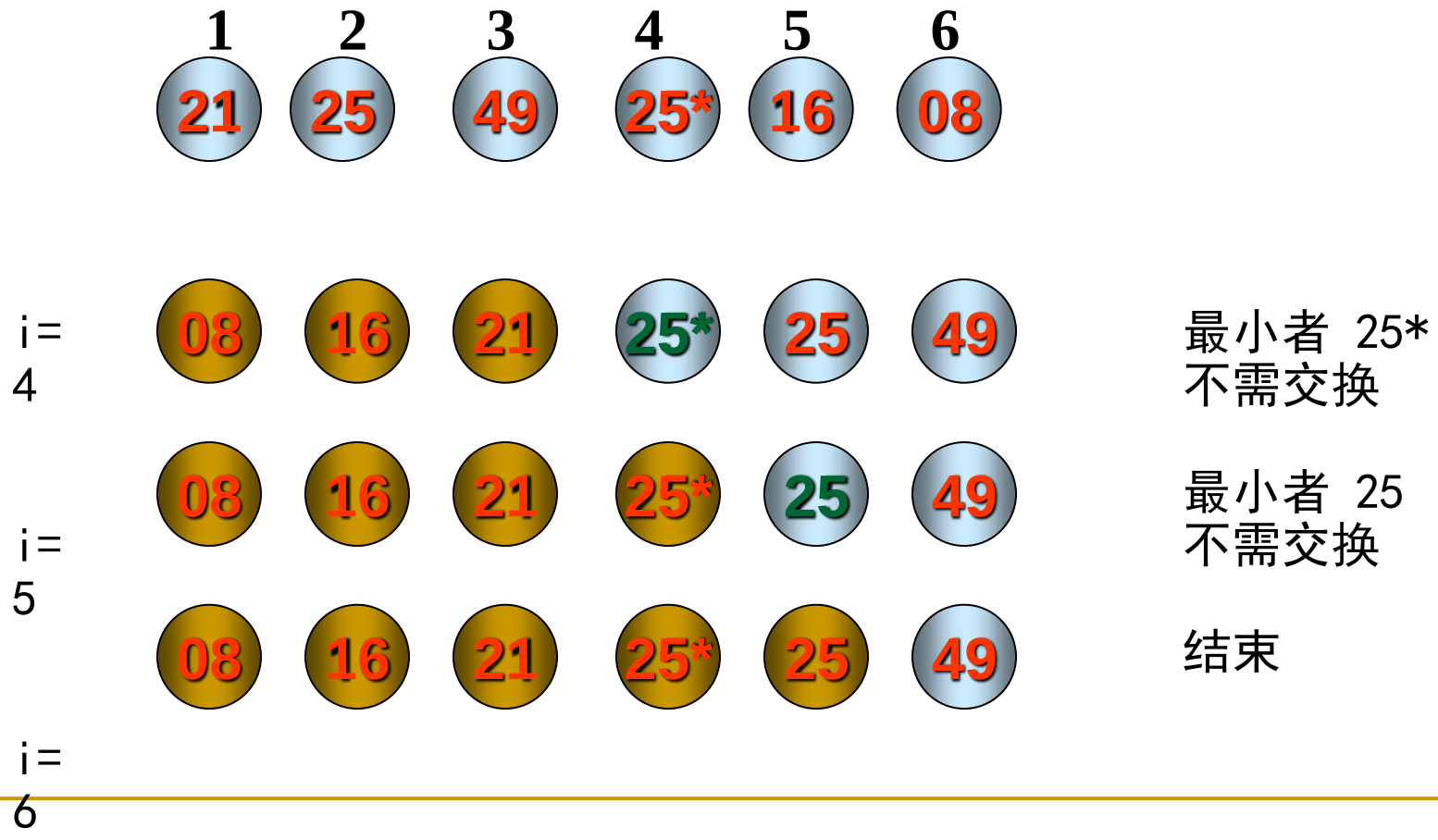
- 举例



10.4 选择排序

一. 简单选择排序

- 举例



10.4 选择排序

一. 简单选择排序

- 算法实现：

```
void SelectSort(SqList &L) {   // 算法10.9
  // 对顺序表L作简单选择排序。
  int i,j;
  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key最小的记录
    if (i!=j) {     // L.r[i]←→L.r[j];    与第i个记录交换
      RedType temp;
      temp=L.r[i];
      L.r[i]=L.r[j];
      L.r[j]=temp;
    }
  }
} // SelectSort
```

10.4 选择排序

一. 简单选择排序

- 算法性能：
  - 直接选择排序的关键字比较次数 KCN 与记录的初始排列无关。
  - 设整个待排序记录序列有n个记录，则第i趟选择具有最小关键字记录所需的比较次数总是 n-i次。总的关键字比较次数为

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

  - 记录的移动次数与记录序列的初始排列有关。当这组记录的初始状态是按其关键字从小到大有序的时候，记录的移动次数RMN=0，达到最少。
  - 最坏情况是每一趟都要进行交换，总的记录移动次数为

$$RMN = 3(n-1)$$

  - 直接选择排序是一种不稳定的排序方法。

10.4 选择排序

二. 堆排序

- 算法设计：
  - 设有一个关键字集合，按完全二叉树的顺序存储方式存放在一个一维数组中。对它们从根开始，自顶向下，同一层自左向右从 1 开始连续编号。若满足

$$K_i \geq K_{2i} \ \&\& \ K_i \geq K_{2i+1}$$

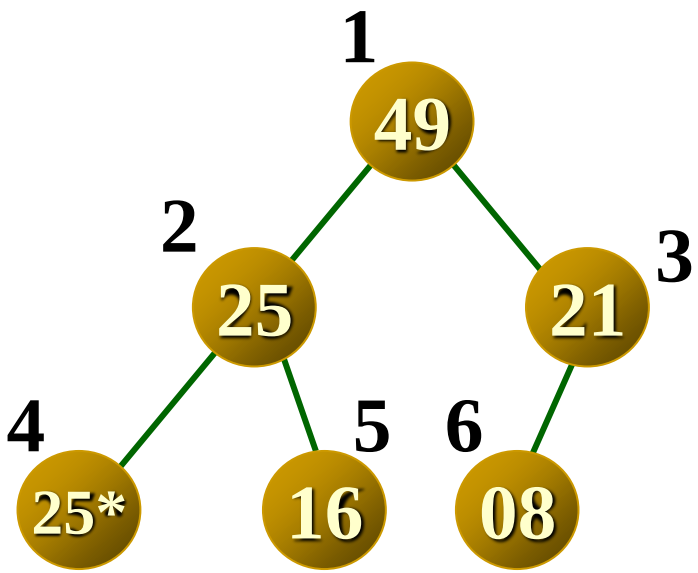
则称该关键字集合构成一个堆(最大堆)

  - 由此可见，堆的根结点必定是最大值或最小值

10.4 选择排序

二. 堆排序

- 举例



10.4 选择排序

二. 堆排序

- 算法实现的主要问题：
  - 如何根据给定的序列建初始堆
  - 如何在交换掉根结点后，将剩下的结点调整为新的堆(筛选)

10.4 选择排序

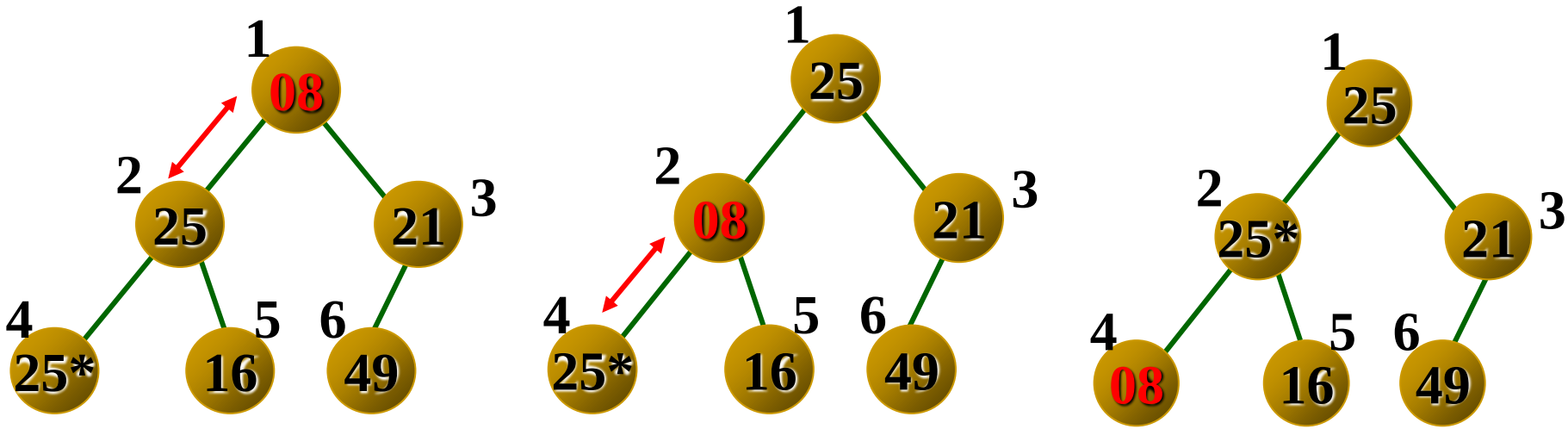
二. 堆排序

- 问题一：筛选方法
  - 输出根结点
  - 用最后结点代替根结点值
  - 比较根结点与两个子结点的值，如果小于其中一个子结点，则选择大的子结点与根结点交换
  - 继续将交换的结点与其子结点比较
  - 直到叶子结点或者根节点值大于两个子结点

10.4 选择排序

二. 堆排序

- 筛选举例



10.4 选择排序

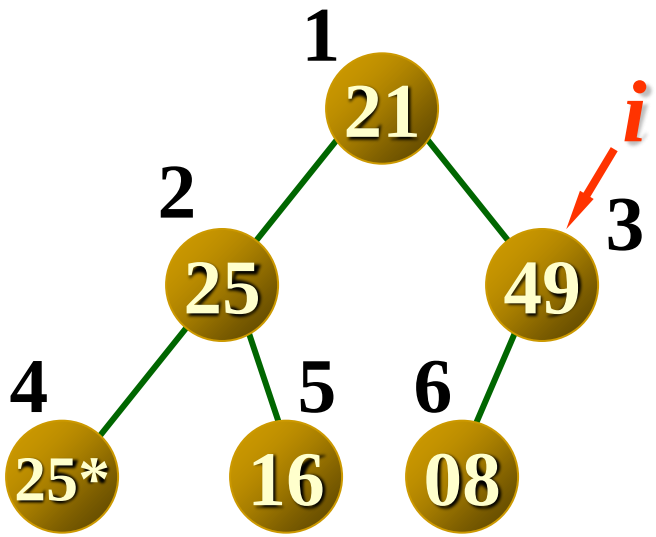
二. 堆排序

- 问题二：创建初始堆
  - 根据给定的序列，从1至n按顺序创建一个完全二叉树
  - 由最后一个非终端结点(第n/2个结点)开始至第1个结点，逐步做筛选

10.4 选择排序

二. 堆排序

- 创建初始堆举例
  - 已知待序的一组记录的初始排列为：21, 25, 49, 25*, 16, 08

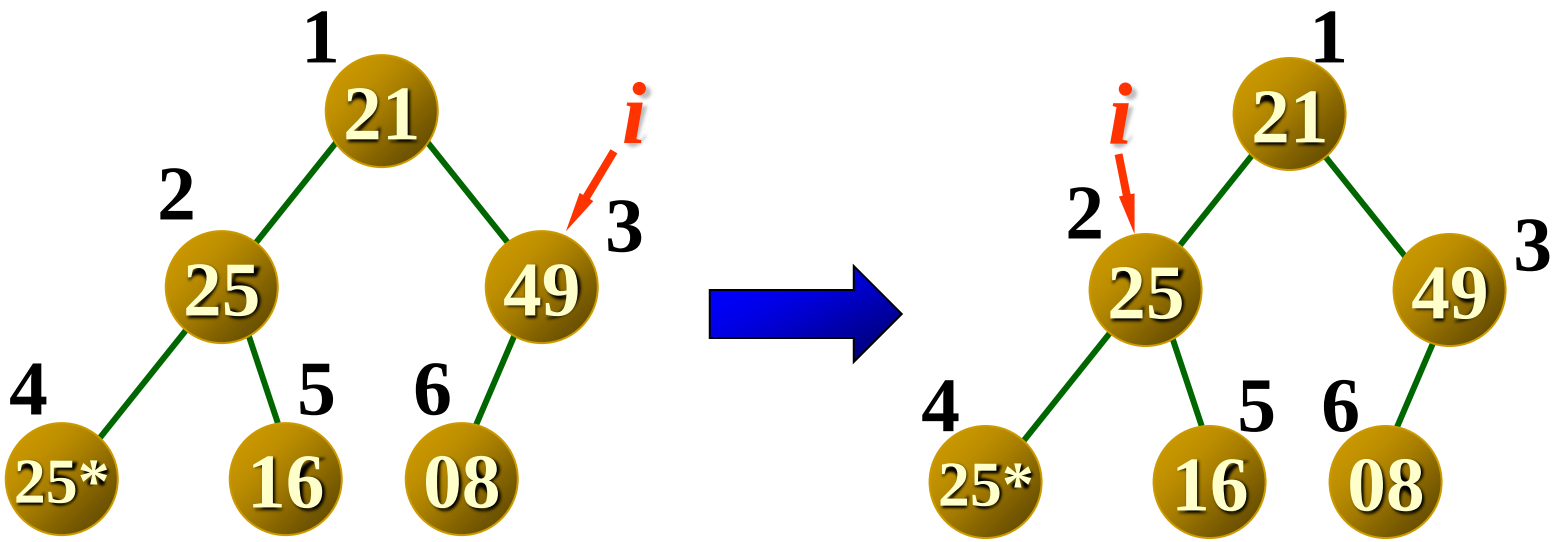


按1到n创建初始树

10.4 选择排序

二. 堆排序

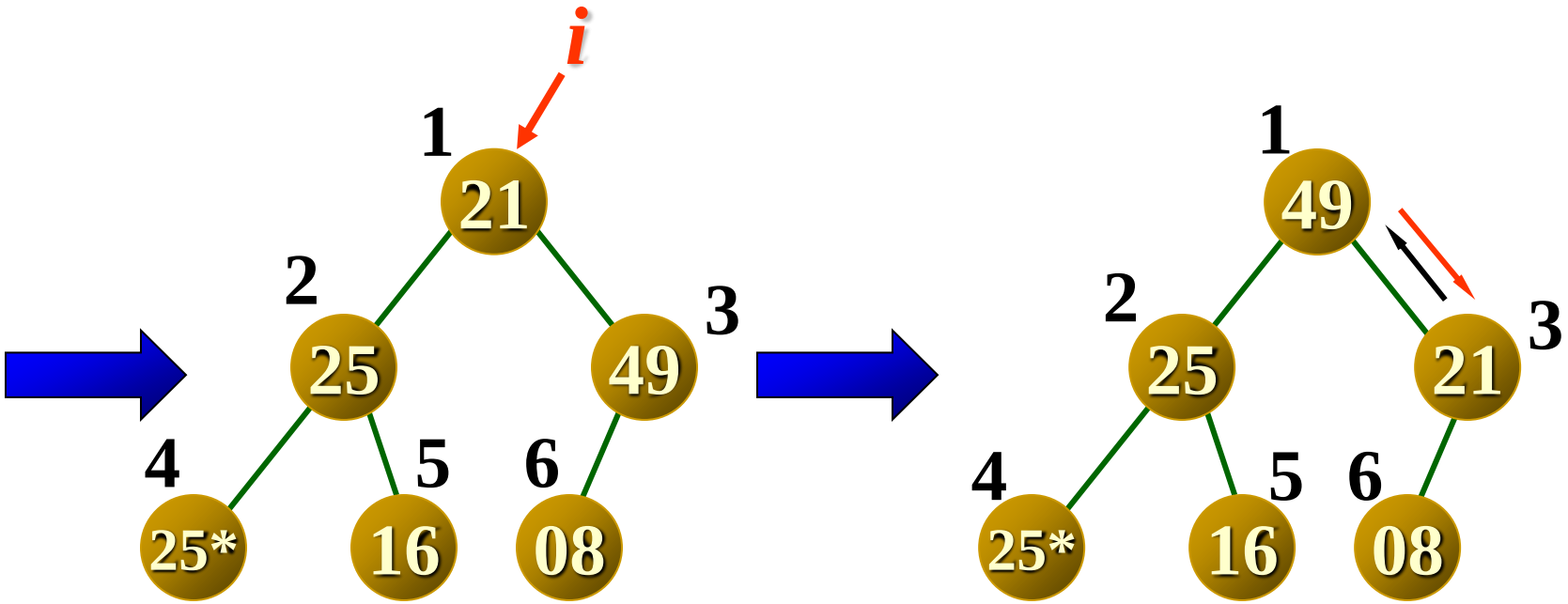
- 创建初始堆举例
  - 已知待序的一组记录的初始排列为：21, 25, 49， 25*, 16, 08
  - i=3时，结点49比它的叶子结点大，无须调整



10.4 选择排序

二. 堆排序

- 创建初始堆举例
  - 已知待序的一组记录的初始排列为：21, 25, 49， 25*, 16, 08
  - i=2时，结点比它的叶子结点大，无须调整
  - i=1时，发生局部调整



10.4 选择排序

二. 堆排序

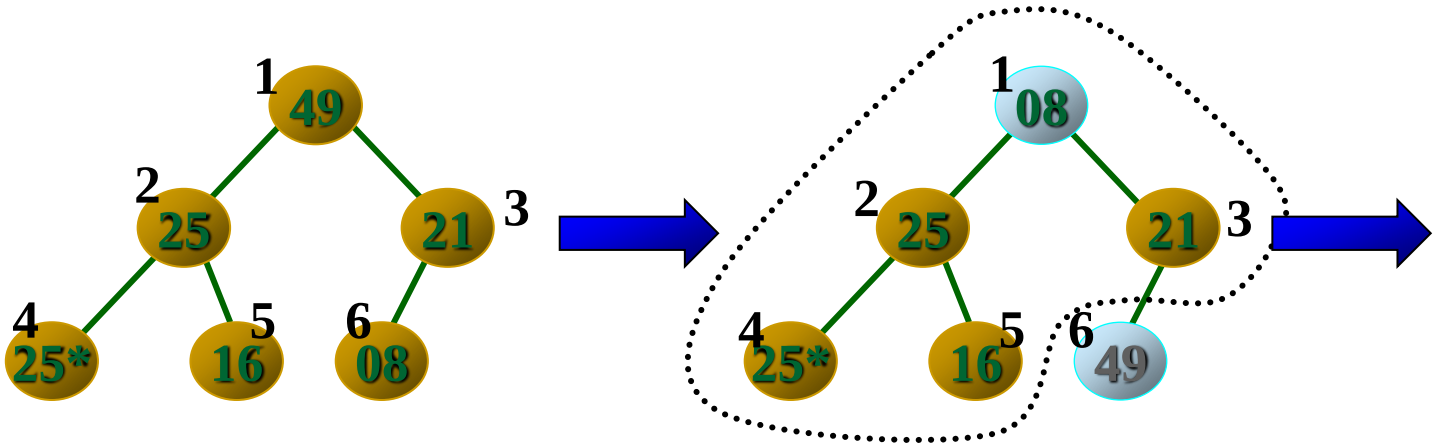
- 堆排序算法流程

1. 将初始序列从1至n按顺序创建一个完全二叉树
2. 将完全二叉树调整为堆
3. 用最后结点代替根结点值
4. 排除掉最后结点，重复步骤2，直到剩下一个结点

10.4 选择排序

二. 堆排序

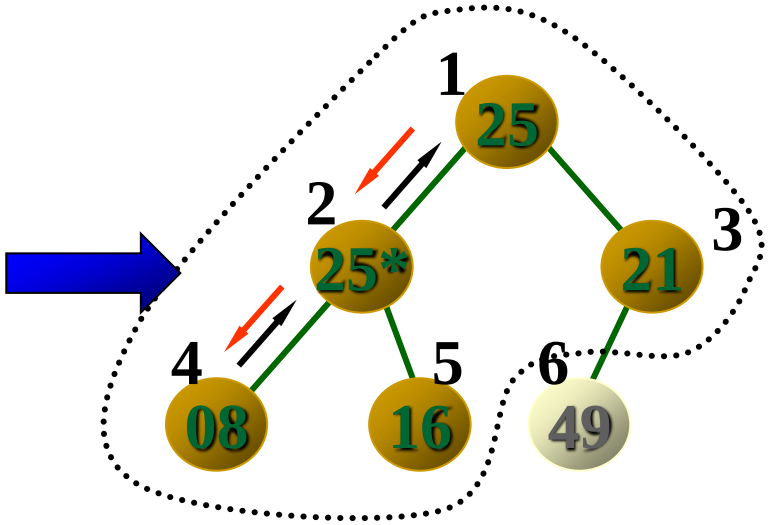
- 堆排序举例
  - 已知待序的一组记录的初始排列为：21, 25, 49, 25*, 16, 08
  - 经过初始最大堆后，根结点为最大值49，交换到末尾，然后排除末尾结点重建新的最大堆

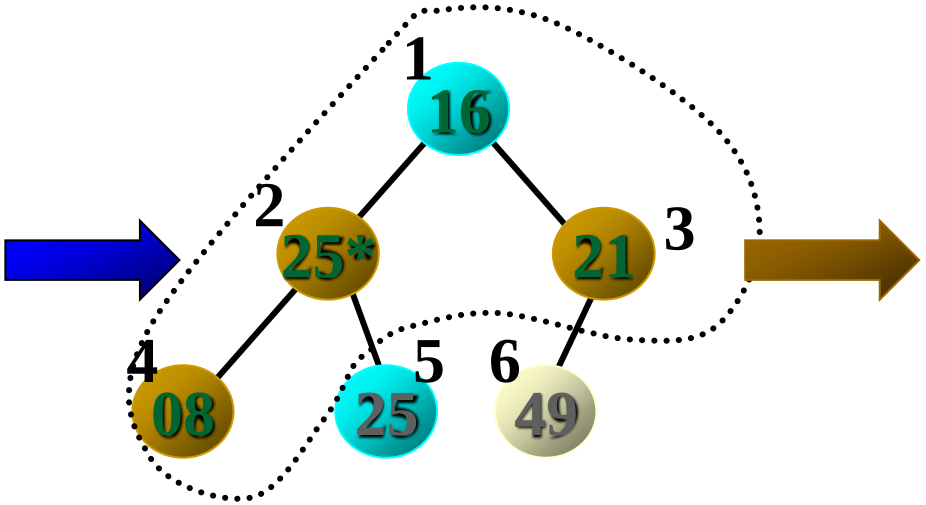


10.4 选择排序

二. 堆排序

- 堆排序举例





25	25*	21	08	16	49

16	25*	21	08	25	49

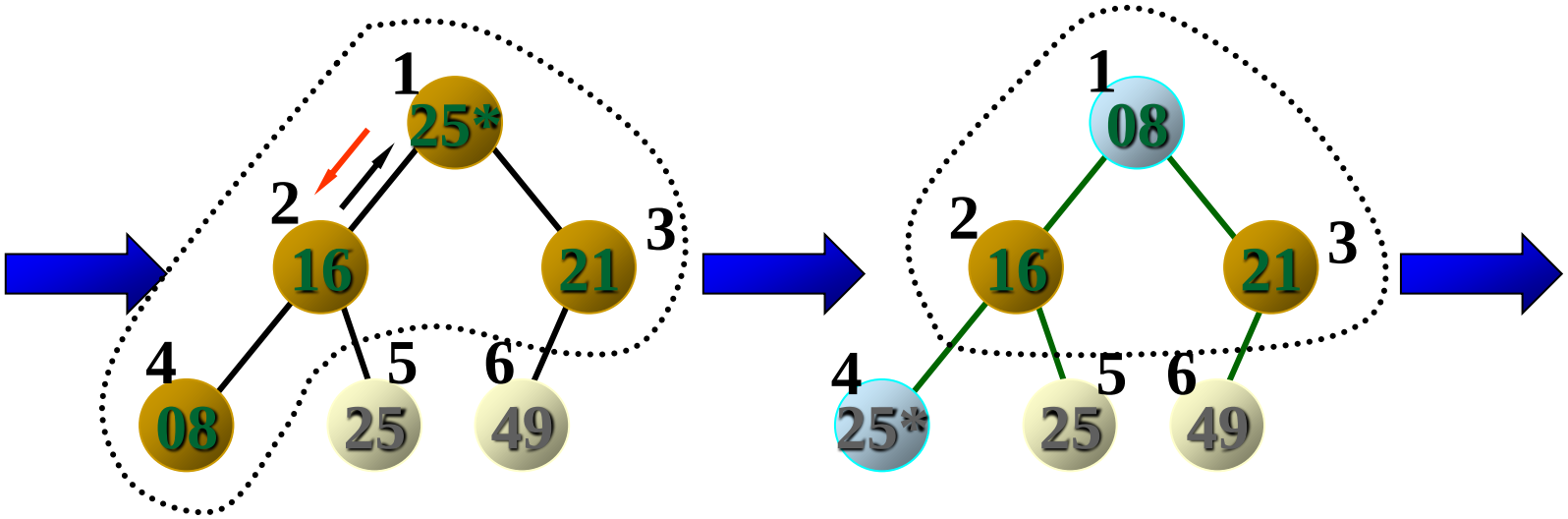
从 1 号到 5 号 重新调整为最大堆

交换 1 号与 5号记录，5 号记录就位

10.4 选择排序

二. 堆排序

- 堆排序举例



25*	16	21	08	25	49

08	16	21	25*	25	49

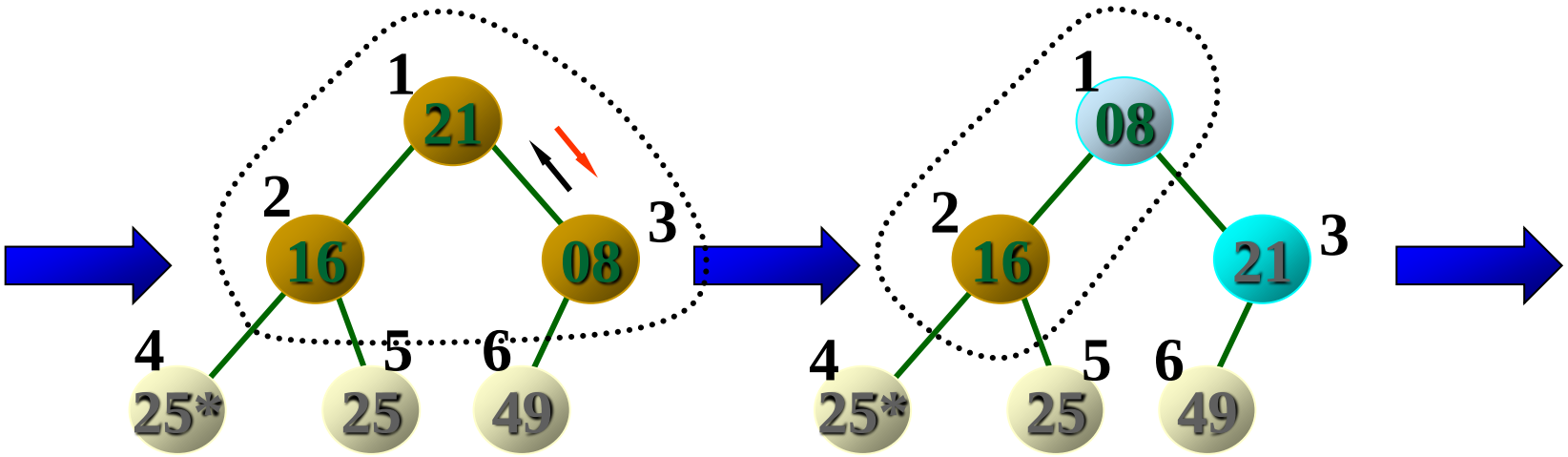
从 1 号到 4 号 重新调整为最大堆

交换 1 号与 4 号记录，4 号记录就位

10.4 选择排序

二. 堆排序

- 堆排序举例



21	16	08	25*	25	49

从 1 号到 3 号 重新调整为最大堆

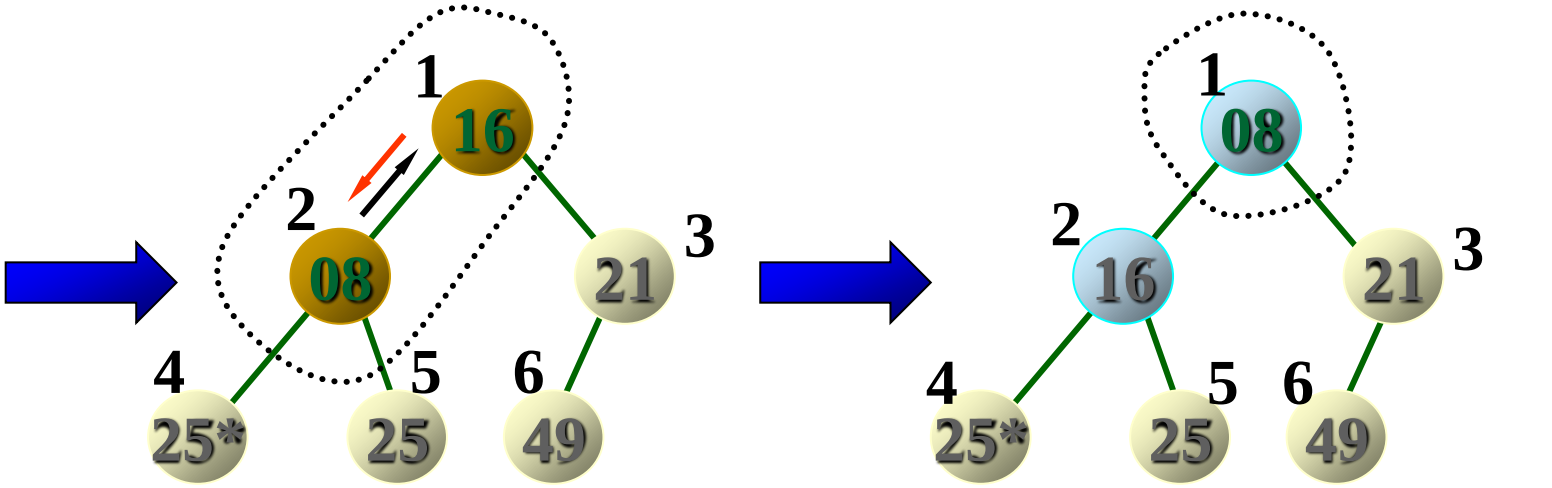
08	16	21	25*	25	49

交换 1 号与 3 号记录，3 号记录就位

10.4 选择排序

二. 堆排序

- 堆排序举例



16	08	21	25*	25	49

从 1 号到 2 号 重新调整为最大堆

08	16	21	25*	25	49

交换 1 号与 2 号记录，所有记录就位

10.4 选择排序

二. 堆排序

- 算法实现：

```
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].key之外均满足堆的定义，
  // 本函数调整H.r[s]的关键字，使H.r[s..m]成为一个大顶堆
  // （对其中记录的关键字而言）
  int j;
  RedType rc;
  rc = H.r[s];
  for (j=2*s; j<=m; j*=2) {    // 沿key较大的孩子结点向下筛选
    // j为key较大的记录的下标
    if (j<m && H.r[j].key<H.r[j+1].key) ++j;
    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;  // 插入
} // HeapAdjust
```

10.4 选择排序

二. 堆排序

- 算法实现：

```
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].key之外均满足堆的定义，
  // 本函数调整H.r[s]的关键字，使H.r[s..m]成为一个大顶堆
  // （对其中记录的关键字而言）
  ... ...
}
void HeapSort(HeapType &H) {
    // 对顺序表H进行堆排序。
    int i;
    RedType temp;
    for (i=H.length/2; i>0; --i)  // 把H.r[1..H.length]建成大顶堆
        HeapAdjust ( H, i, H.length );
    for (i=H.length; i>1; --i) {
        temp=H.r[i];
        H.r[i]=H.r[1];
        H.r[1]=temp;  // 将堆顶记录和当前未经排序子序列Hr[1..i]中
                      // 最后一个记录相互交换
        HeapAdjust(H, 1, i-1);  // 将H.r[1..i-1] 重新调整为大顶堆
    }
} // HeapSort
```

10.4 选择排序

二. 堆排序

- 算法性能：
  - 对于长度为n的序列，其对应的完全二叉树的深度为k（$2^{k-1} \leq n < 2^k$）
  - 对深度为k的堆，筛选算法中进行的关键比较次数至多为2(k-1)次
  - 堆排序时间主要耗费在建初始堆和调整建新堆(筛选)上
  - 建初始堆最多做n/2次筛选

10.4 选择排序

二. 堆排序

- 算法性能：
  - 对长度为n的序列，排序最多需要做n-1次调整建新堆(筛选)。建初始堆时，需要n/2次筛选，因此共需要O(nxk)量级的时间
  - k = $\log_2 n$，堆排序时间复杂度为O(n$\log_2 n$)
  - 堆排序是一个不稳定的排序方法