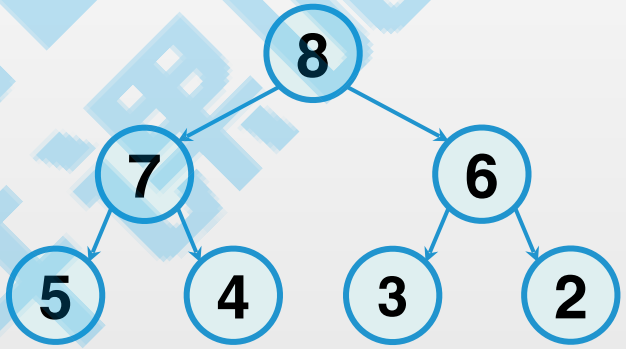
堆

什么是堆，如何构建堆，如何使用堆排序

什么是堆

- 假设二叉树 T 是一棵完全二叉树，如果树 T 中的任一结点的值**不小于**它的任一子结点的值（如果存在子结点的话），那么称树 T 是一个（最大）堆（heap）

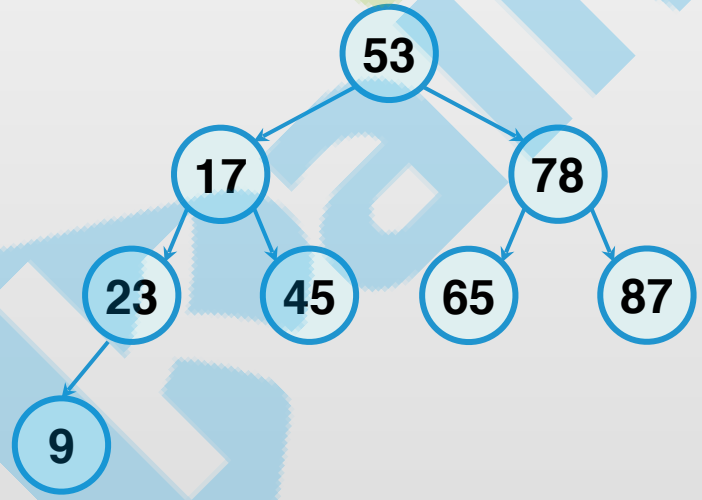


最大堆

堆的构建

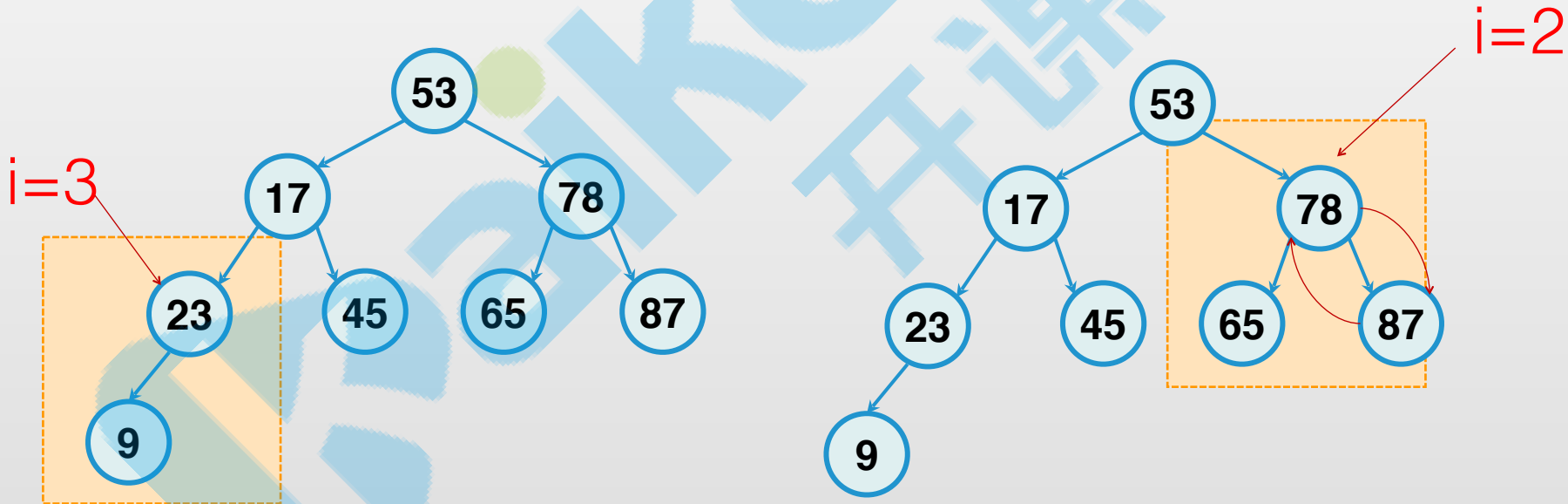
对于具有 n 个结点的完全二叉树 T，可以按**层次序**把树中的所有结点存放在数组 a[n] 中。这样，就可以用数组 a[n] 表示完全二叉树 T。

例: a = {53, 17, 78, 23, 45, 65, 87, 9}



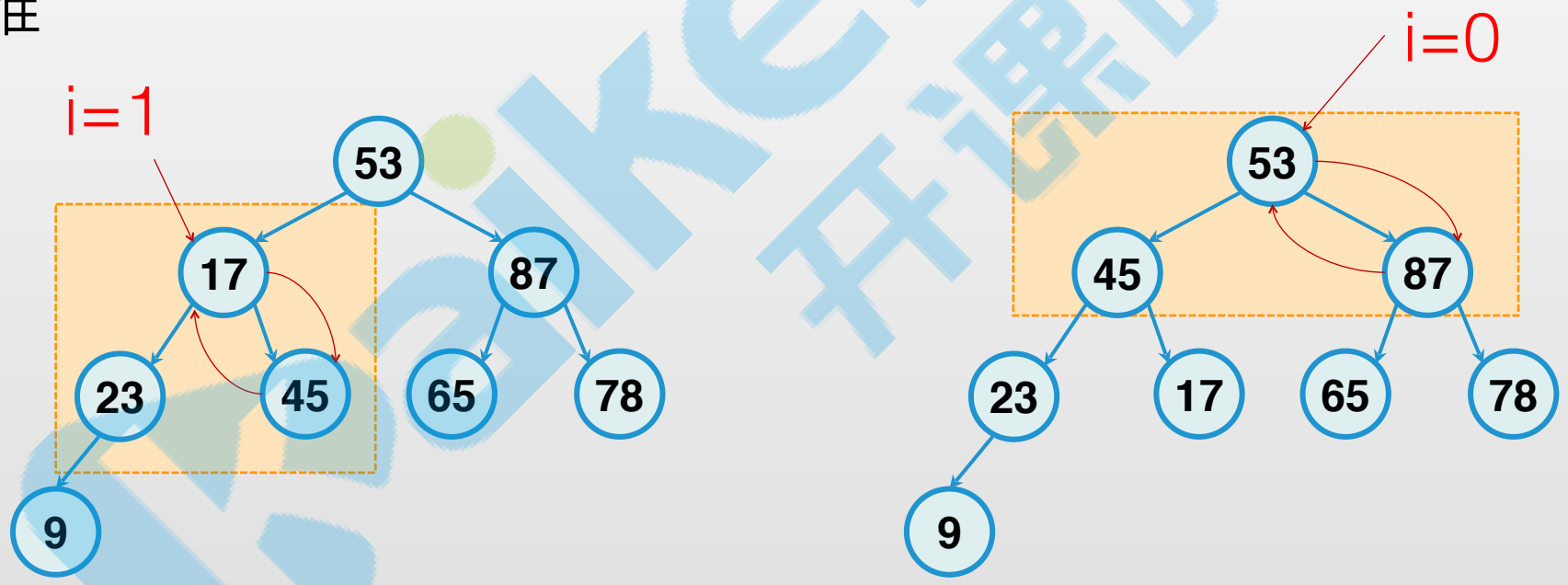
堆的构建

对于有 n 个结点的完全二叉树，我们从 n/2 个结点处开始**自下向上**逐步调整为最大堆



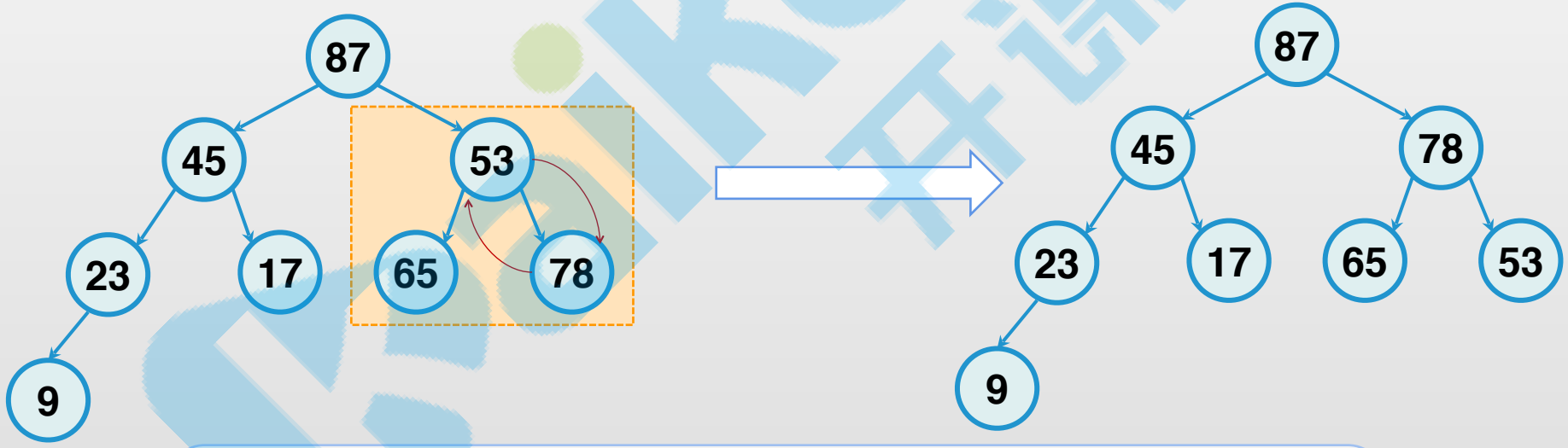
堆的构建

对于有 n 个结点的完全二叉树，我们从 n/2 个结点处开始**自下向上**逐步调整为最大堆



堆的构建

对于有 n 个结点的完全二叉树，我们从 n/2 个结点处开始**自下向上**逐步调整为最大堆



调整完成后的数组 a 为

```
{87, 45, 78, 23, 17, 65, 53, 9}
```

堆的性质

如果具有 n 个结点的完全二叉数 T 是一个最大堆，那么按层次序存放 T 的数组 a[n] 中的元素具有如下的性质：

（1）若 2i + 1 < n，则 a[i] ≥ a[2i + 1]

（2）若 2i + 2 < n，则 a[i] ≥ a[2i + 2]

```
void heap (int[] a, int n) {
   for (int i = (n - 2) / 2; i >= 0; i--)
       // 调整为堆
       shiftdown(a, i, n);
}
```

```
void siftdown(int[] a, int i, int n) {
    int j;
    int t = a[i];
    while ((j = 2 * i + 1 ) < n) {
        // 子结点中选大者
        if (j < n - 1 && a[j] < a[j + 1])
            j++;
        if (t < a[j]) {
            a[i] = a[j];
            i = j;
        } else
            break;
    }
    a[i] = t;
}
```

堆的用途

- 堆排序
  - 把最大堆的根节点和末尾节点互换，则值最大的节点位置确定在最后；循环往复，实现从小到大的排序
- 优先队列
  - 入队=插入到末尾
  - 出队=删除根节点
- Top-k
  - 逐个取出堆顶，取末尾填充；重复k次

堆排序

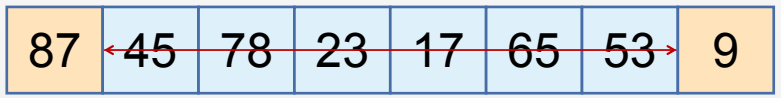
堆排序

- 堆排序是借助堆结构的排序，这样的处理过程称为堆排序。

排序思路

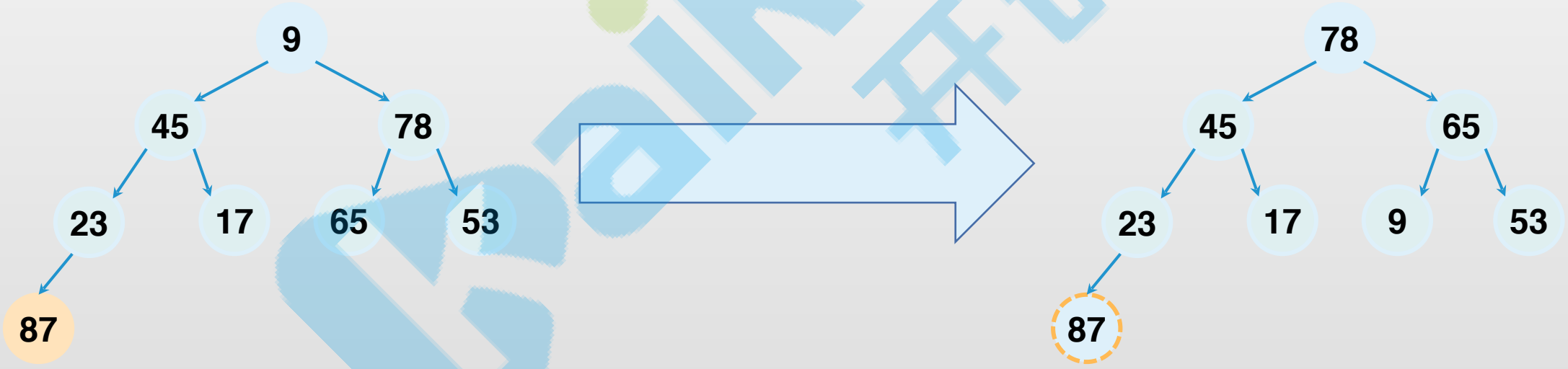
1. 如果a[0]，a[1]，…，a[n-1]表示一个最大堆，那么a[0]是最大的。将a[0]与a[n-1]对调，把最大值调到a[n-1]。
2. 然后，把a[0]，...，a[n-2]再次调整为最大堆，继续将a[0]与a[n-2]对调
3. 重复进行调整，直到调整范围只有一个结点a[0]为止。
4. 经过处理后，a[0]，a[1]，…，a[n-1]已按从小到大的次序排了序

堆排序



交换a[0]与a[7]

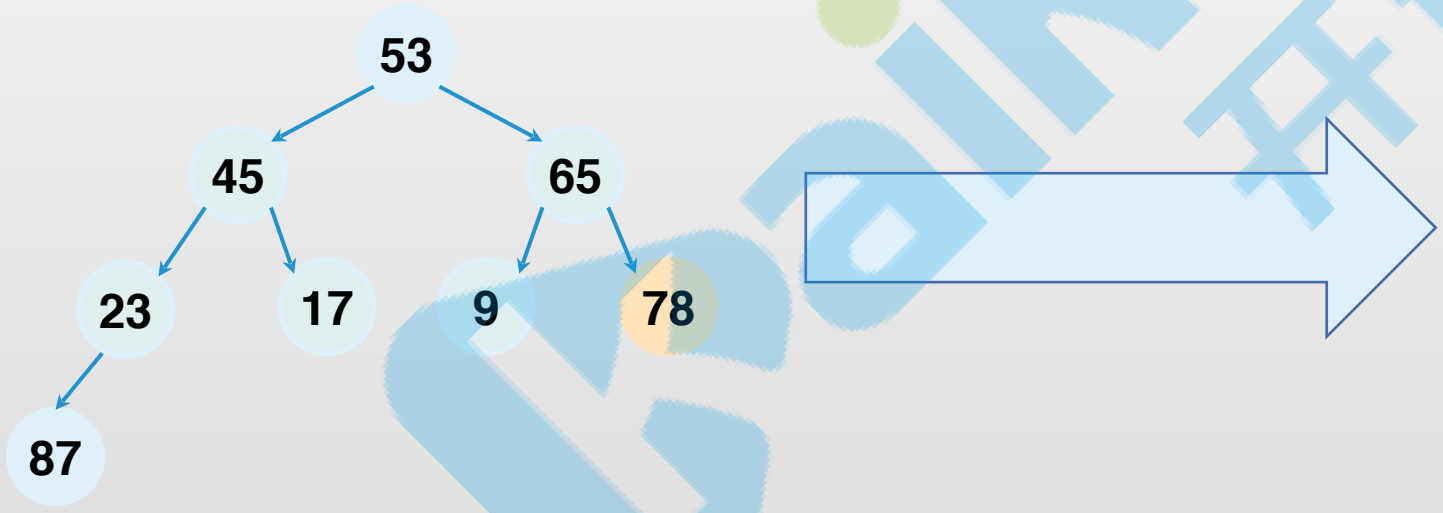
构建a[0],..,a[6]最大堆



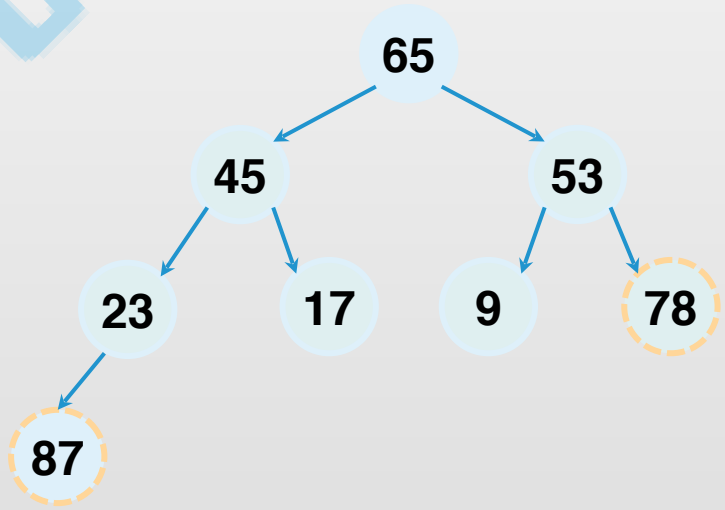
堆排序

78	45	65	23	17	9	53	87

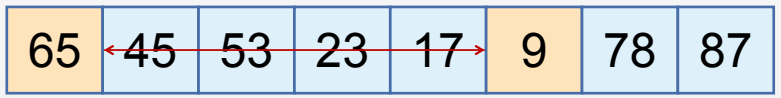
交换a[0]与a[6]



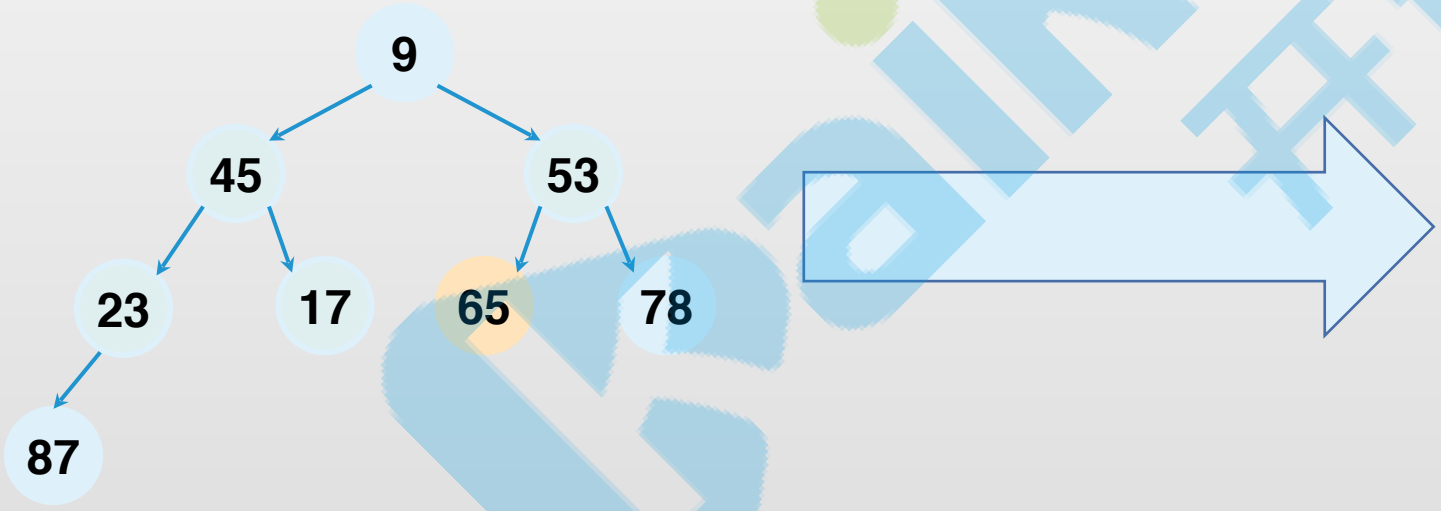
构建a[0],..,a[5]最大堆



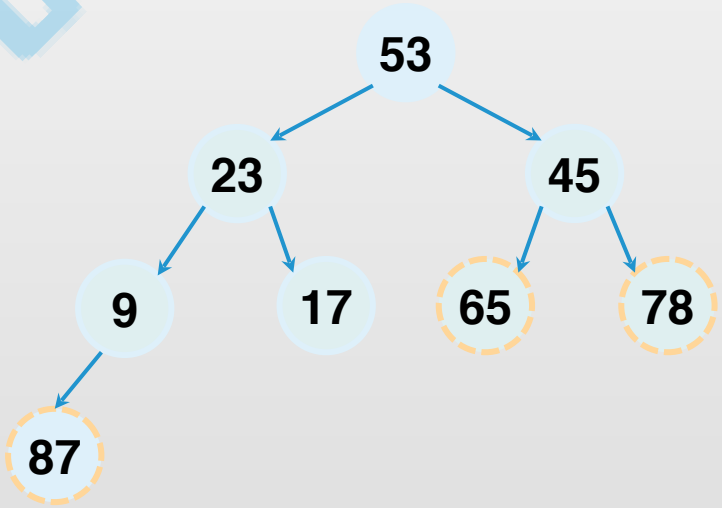
堆排序



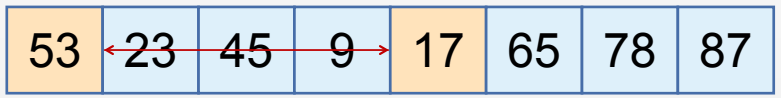
交换a[0]与a[5]



构建a[0],..,a[4]最大堆

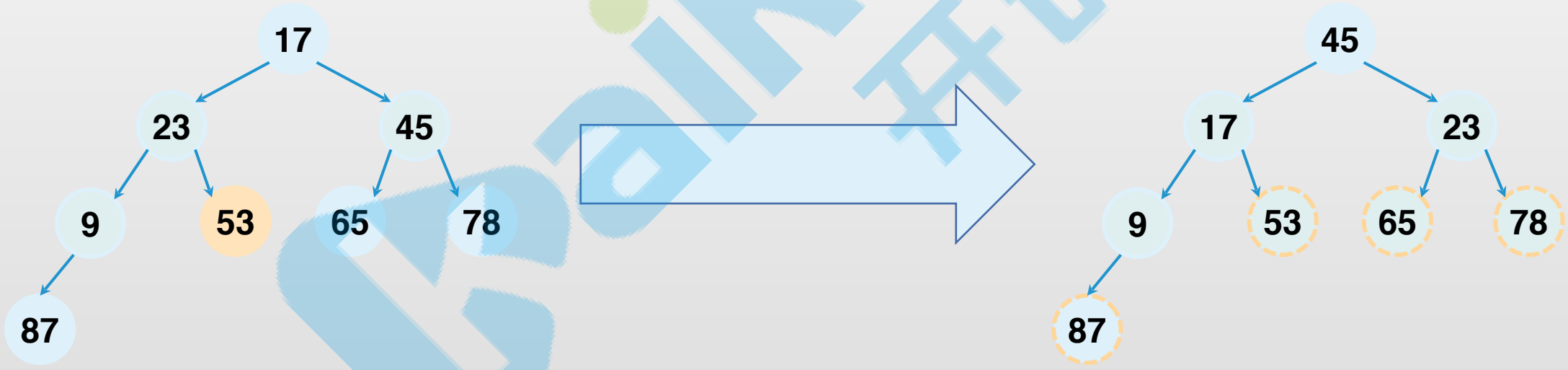


堆排序



交换a[0]与a[4]

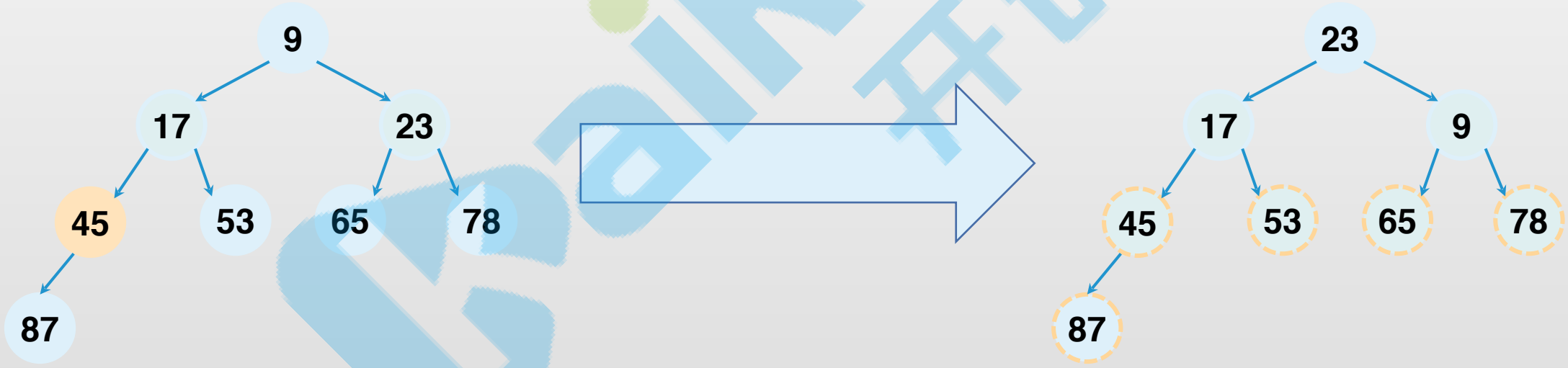
构建a[0],..,a[3]最大堆



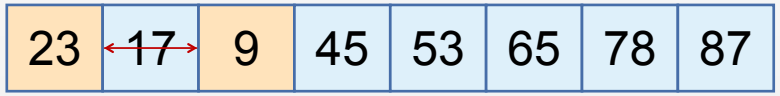
堆排序

交换a[0]与a[3]

构建a[0],..,a[2]最大堆

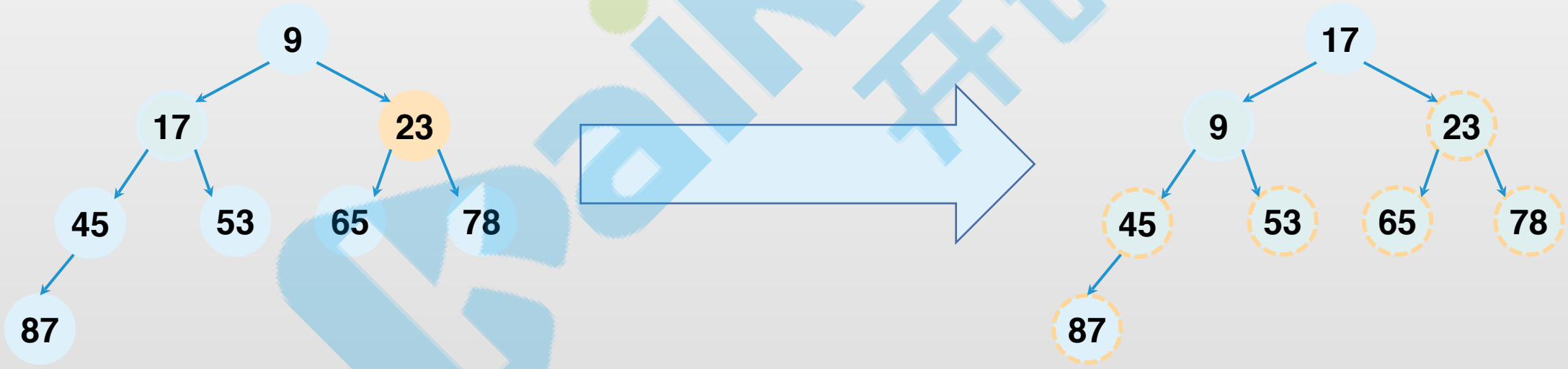


堆排序



交换a[0]与a[2]

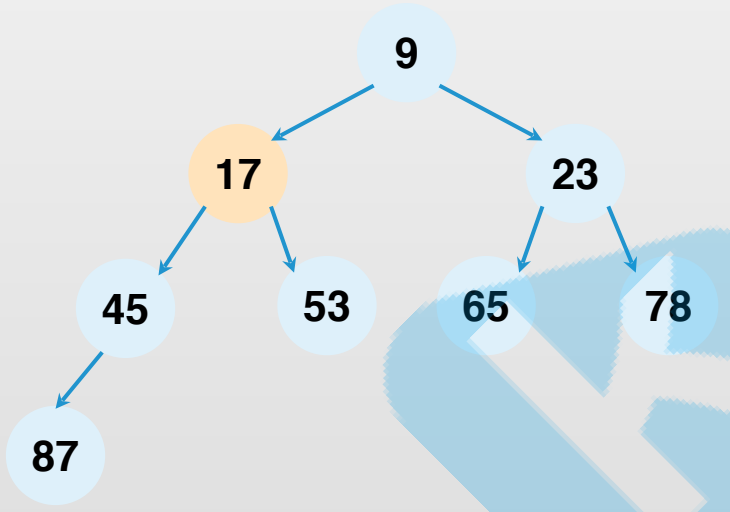
构建a[0],a[1]最大堆



堆排序

17	9	23	45	53	65	78	87

交换a[0]与a[1]



至此，数组已完成
从小到大排序

9	17	23	45	53	65	78	87

堆排序

```
void heap_sort(int[] a, int n) {
    int i; int t;
    for (i = (n - 2) / 2; i >= 0; i--) {
        siftdown(a, i, n); //调整为堆
    }
    for (i = n - 1; i > 0; i--) {
        //进行堆排序
        t = a[0];
        a[0] = a[i];
        a[i] = t;
        siftdown(a, 0, i);
    }
}
```

```
void siftdown(int[] a, int i, int n) {
    int j;
    int t = a[i];
    while((j = 2*i + 1 ) < n) {
        // 两子女中选大者
        if(j < n - 1 && a[j] < a[j + 1])
            j++;
        if(t < a[j]) {
            a[i] = a[j];
            i = j;
        } else
            break;
    }
    a[i] = t;
}
```

siftdown(a, i, n)

循环外的执行时间： O(1)

循环体的执行时间： O(1)

循环执行次数 ≦ $\log_2 n + 1$

$O(\log_2 n)$

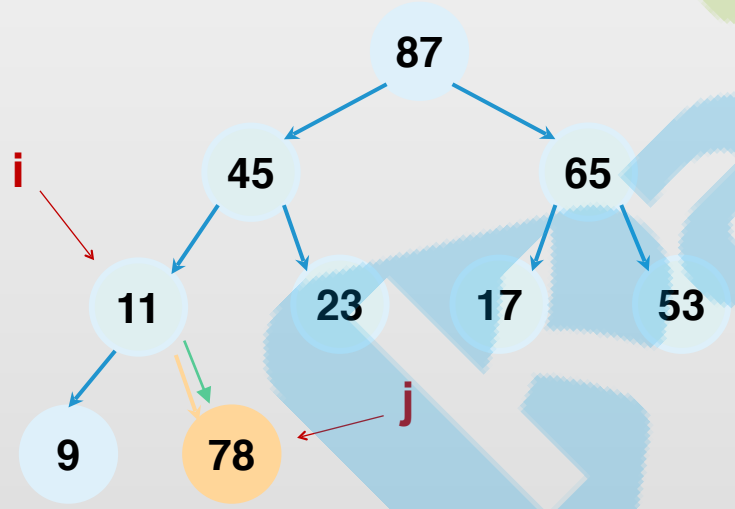
heap_sort(a, n) **$O(n\log_2 n)$**

第一个循环执行时间：$O(n\log_2 n)$

第二个循环执行时间：$O(n\log_2 n)$

堆的插入

堆结构的数据插入如何操作?
我们假设新的元素先被放在堆末尾位置 k 的结点中



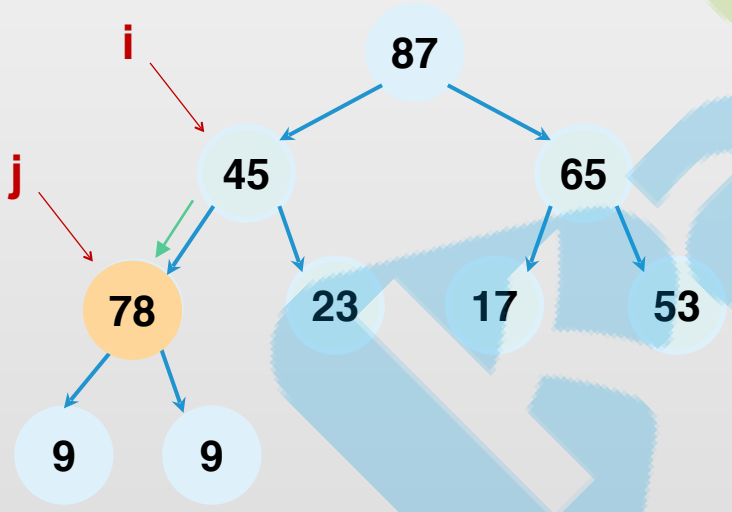

```
void siftup (int[] a, int k) {
    int j = k, i = (j - 1) / 2;
    int t = a[j];
    while (j > 0) {
        if (a[i] >= t)
            break;
        else {
            a[j] = a[i];
            j = i;
            i = (i - 1) / 2;
        }
    }
    a[j] = t;
}
```

我们使用siftup向上调整，进行堆插入元素的调整

堆的插入

堆结构的数据插入如何操作?
我们假设新的元素先被放在堆末尾位置k的单元中

向上调整

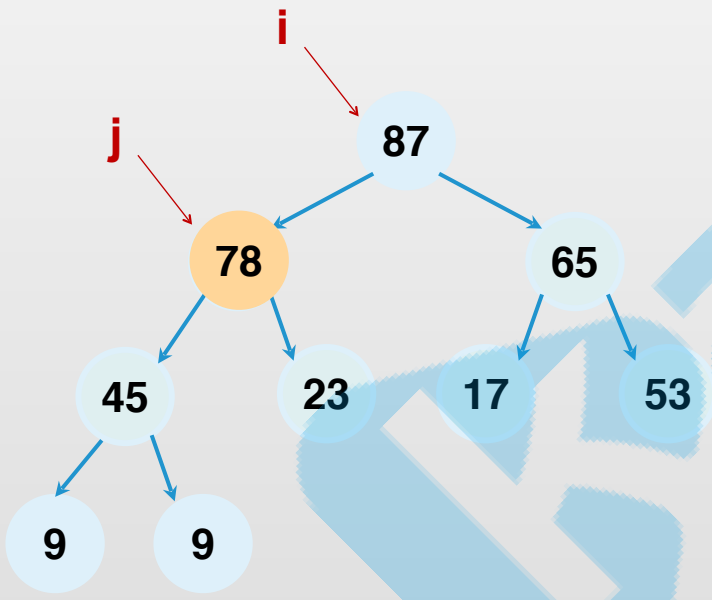



```
void siftup (int[] a, int k) {
    int j = k, i = (j - 1) / 2;
    int t = a[j];
    while (j > 0) {
        if (a[i] >= t)
            break;
        else {
            a[j] = a[i];
            j = i;
            i = (i - 1) / 2;
        }
    }
    a[j] = t;
}
```

我们使用siftup向上调整，进行堆插入元素的调整

堆的插入

堆结构的数据插入如何操作?
我们假设新的元素先被放在堆末尾位置k的单元中

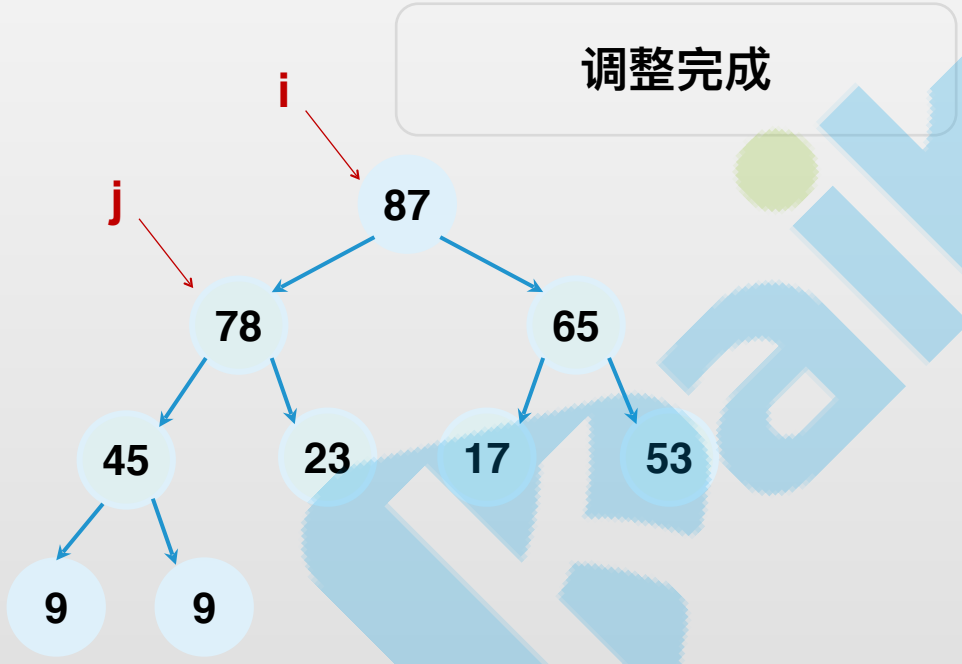



```
void siftup (int[] a, int k) {
    int j = k, i = (j - 1) / 2;
    int t = a[j];
    while (j > 0) {
        if (a[i] >= t)
            break;
        else {
            a[j] = a[i];
            j = i;
            i = (i - 1) / 2;
        }
    }
    a[j] = t;
}
```

我们使用siftup向上调整，进行堆插入元素的调整

堆的插入

堆结构的数据插入如何操作?
我们假设新的元素先被放在堆末尾位置k的单元中




```
void siftup (int[] a, int k) {
    int j = k, i = (j - 1) / 2;
    int t = a[j];
    while (j > 0) {
        if (a[i] >= t)
            break;
        else {
            a[j] = a[i];
            j = i;
            i = (i - 1) / 2;
        }
    }
    a[j] = t;
}
```

我们使用siftup向上调整，进行堆插入元素的调整