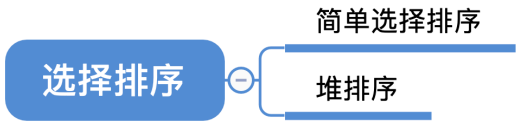
本节内容

堆排序

知识总览



选择排序：每一趟在待排序元素中选取关键字最小（或最大）的元素加入有序子序列

什么是“堆（Heap）”?

若n个关键字序列L[1…n]满足下面某一条性质，则称为堆（Heap）：

① 若满足：L(i)≥L(2i)且L(i)≥L(2i+1)（$1 \le i \le n/2$）—— 大根堆（大顶堆）

② 若满足：L(i)≤L(2i)且L(i)≤L(2i+1)（$1 \le i \le n/2$）—— 小根堆（小顶堆）

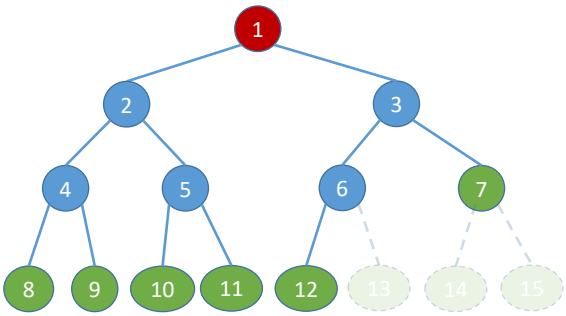
大根堆

	87	45	78	32	17	65	53	09
0	1	2	3	4	5	6	7	8

小根堆

	09	45	17	65	53	32	87	78
0	1	2	3	4	5	6	7	8

二叉树的顺序存储



几个重要常考的基本操作：

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——$\lfloor i/2 \rfloor$
- i 所在的层次 ——$\lceil \log_2(n+1) \rceil$或$\lfloor \log_2 n \rfloor + 1$

若完全二叉树中共有n个结点，则

- 判断 i 是否有左孩子？ ——2i ≤ n ?
- 判断 i 是否有右孩子？ ——2i+1 ≤ n ?
- 判断 i 是否是叶子/分支结点？ ——i > $\lfloor n/2 \rfloor$?

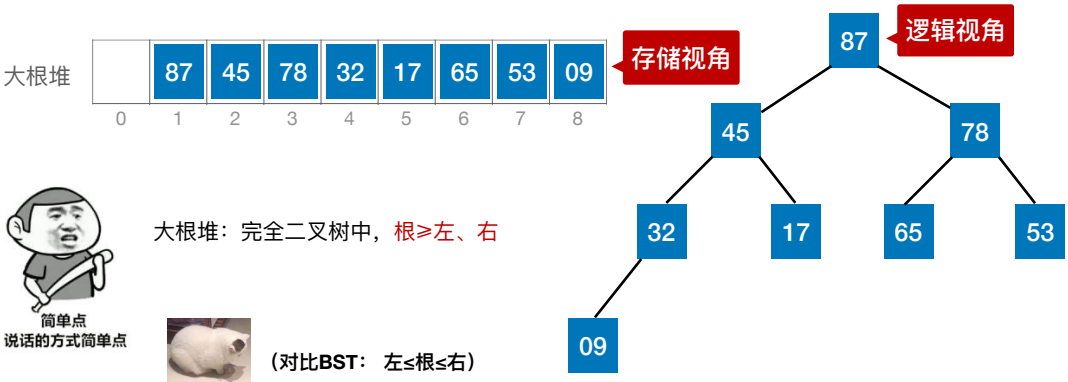
^	1	2	3	4	5	6	7	8	9	10	11	12	^	^	^	^
t[0]	t[1]	t[2]	.....													

什么是“堆（Heap）”?

若n个关键字序列L[1…n] 满足下面某一条性质，则称为堆（Heap）：

① 若满足：L(i)≥L(2i)且L(i)≥L(2i+1) （$1 \le i \le n/2$） —— 大根堆（大顶堆）

② 若满足：L(i)≤L(2i)且L(i)≤L(2i+1) （$1 \le i \le n/2$） —— 小根堆（小顶堆）



简单点
说话的方式简单点

大根堆：完全二叉树中，根≥左、右

（对比BST： 左≤根≤右）

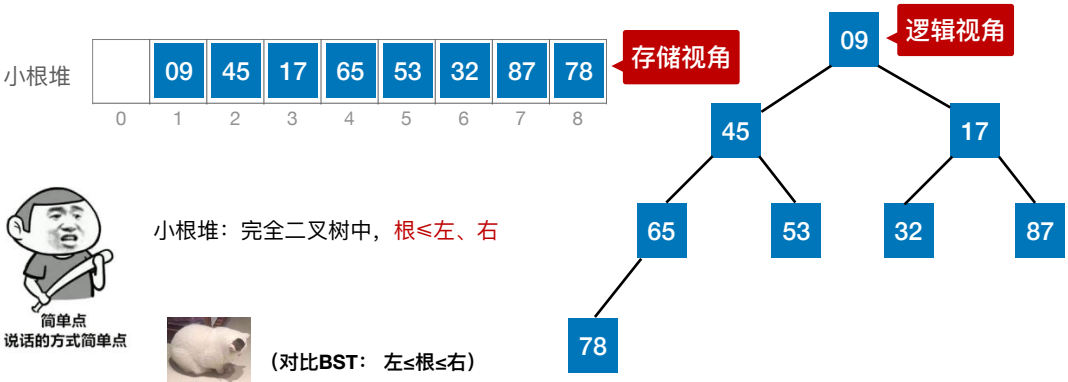
王道考研/CSKAOYAN.COM

什么是“堆（Heap）”?

若n个关键字序列L[1…n] 满足下面某一条性质，则称为堆（Heap）：

① 若满足：L(i)≥L(2i)且L(i)≥L(2i+1) （$1 \le i \le n/2$） —— 大根堆（大顶堆）

② 若满足：L(i)≤L(2i)且L(i)≤L(2i+1) （$1 \le i \le n/2$） —— 小根堆（小顶堆）



简单点
说话的方式简单点

小根堆：完全二叉树中，根≤左、右

（对比BST： 左≤根≤右）

王道考研/CSKAOYAN.COM

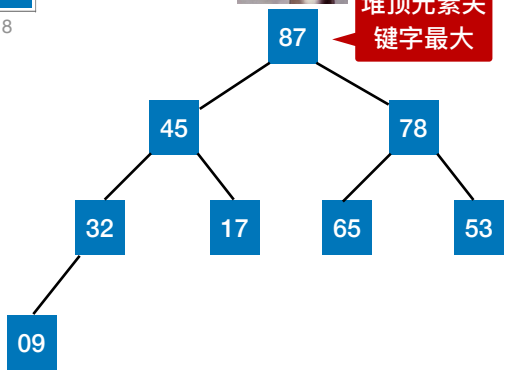
如何基于“堆”进行排序?

大根堆

	87	45	78	32	17	65	53	09
0	1	2	3	4	5	6	7	8

大根堆：根≥左、右

堆顶元素关键字最大



选择排序
- 简单选择排序
- 堆排序

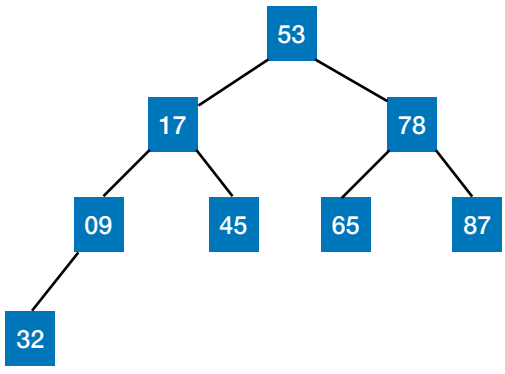
选择排序：每一趟在待排序元素中选取关键字最小（或最大）的元素加入有序子序列

建立大根堆

初始序列

	53	17	78	09	45	65	87	32
0	1	2	3	4	5	6	7	8

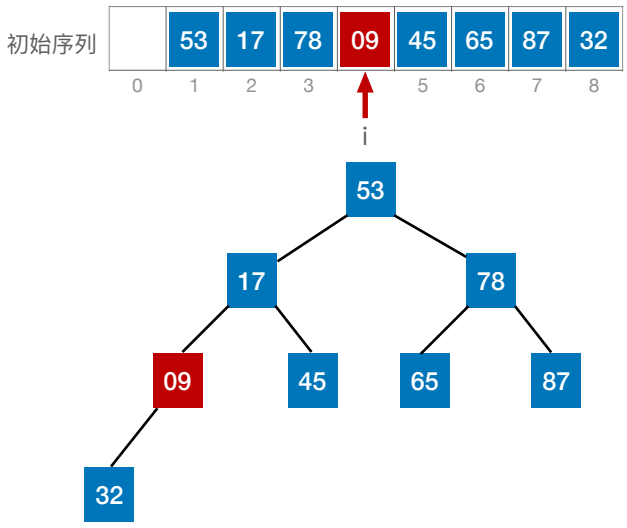
大根堆：根≥左、右



思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

在顺序存储的完全二叉树中，非终端结点编号 $i \le \lfloor n/2 \rfloor$

建立大根堆



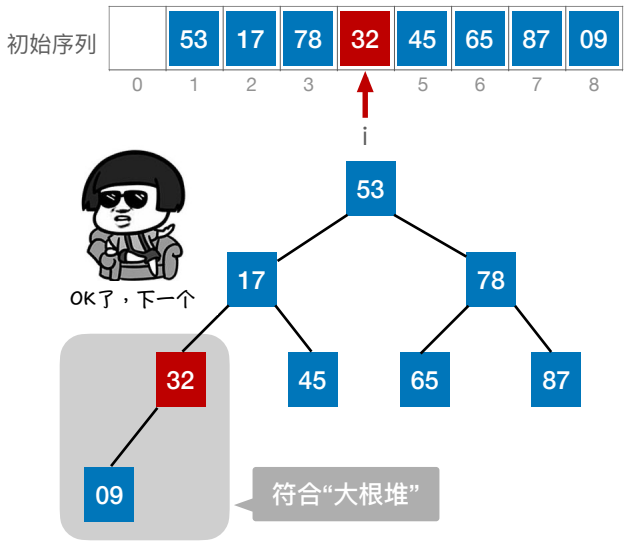
大根堆：根≥左、右

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆



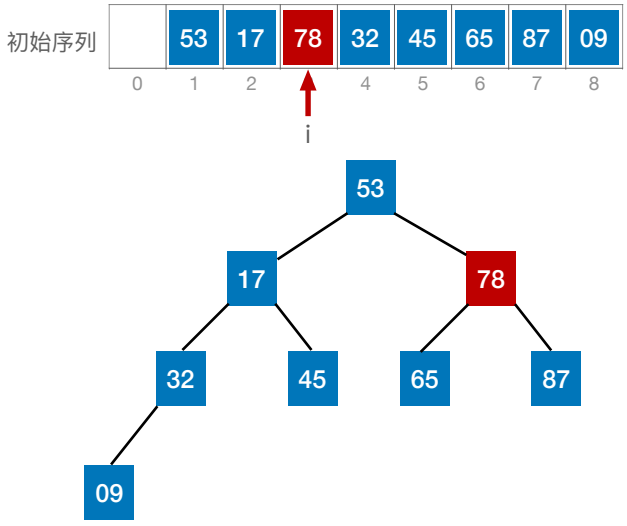
大根堆：根≥左、右

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆



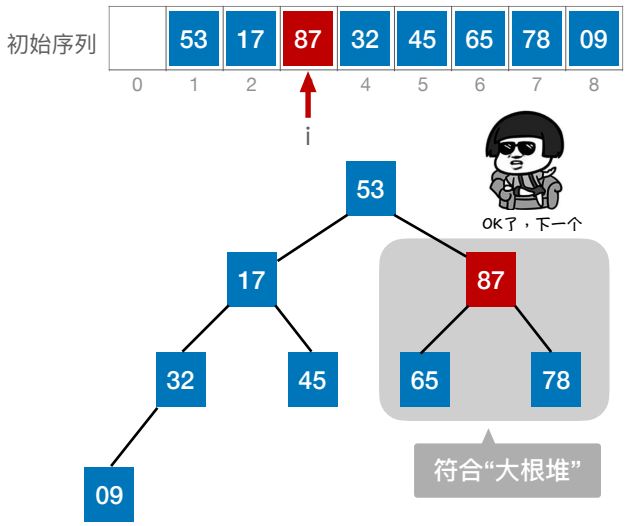
大根堆：根≥左、右

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆



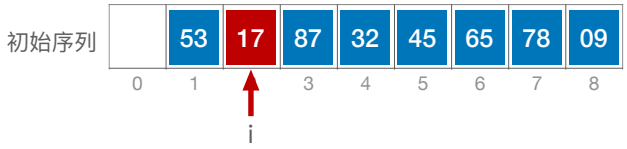
大根堆：根≥左、右

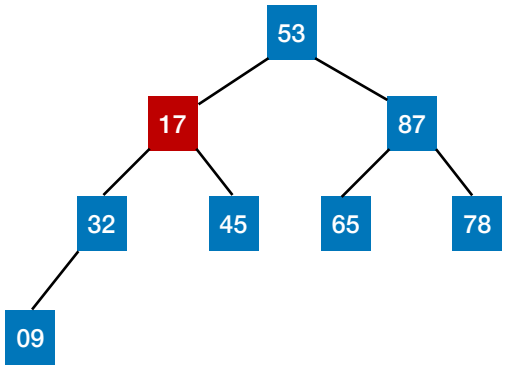
思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆





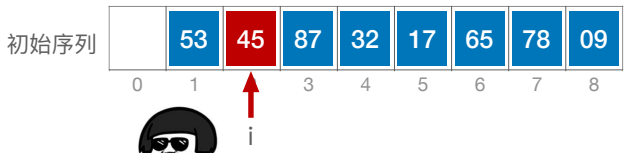
大根堆：根≥左、右

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆



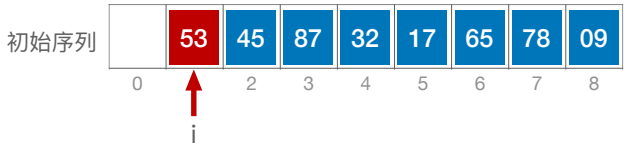
大根堆：根≥左、右

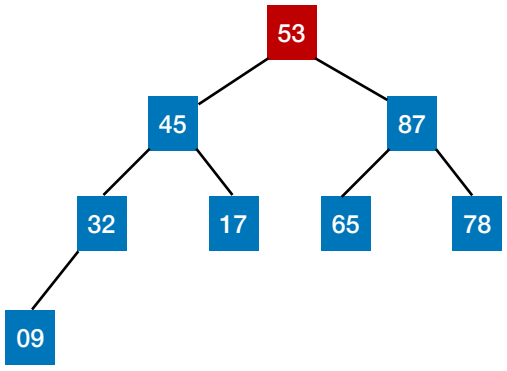
思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆





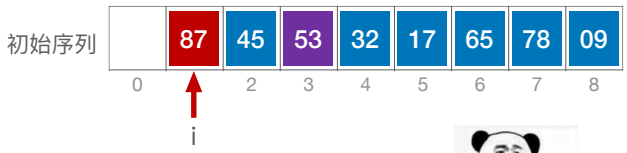
大根堆：根≥左、右

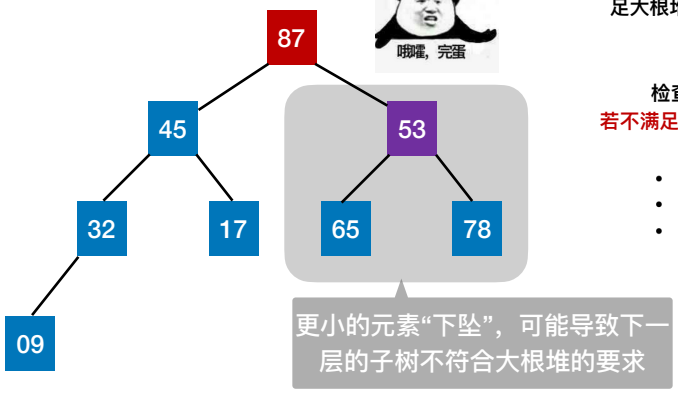
思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆





更小的元素“下坠”，可能导致下一层的子树不符合大根堆的要求

大根堆：根≥左、右

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

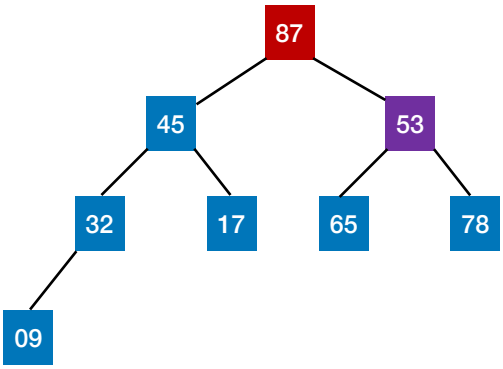
- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆

初始序列

	87	45	53	32	17	65	78	09
0		2	3	4	5	6	7	8

i



大根堆：根≥左、右

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

若元素互换破坏了下一级的堆，则采用相同的方法继续往下调整（小元素不断“下坠”）

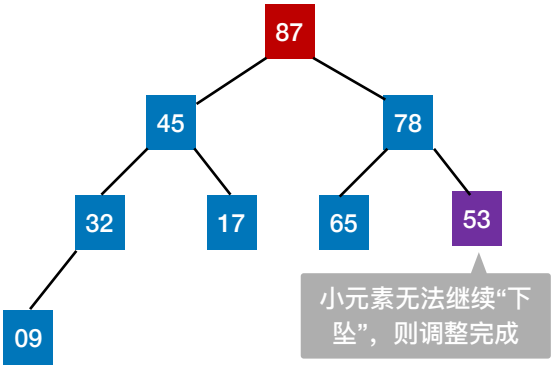
- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——$\lfloor i/2 \rfloor$

建立大根堆

初始序列

	87	45	78	32	17	65	53	09
0		2	3	4	5	6	7	8

i



大根堆：根≥左、右

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

若元素互换破坏了下一级的堆，则采用相同的方法继续往下调整（小元素不断“下坠”）

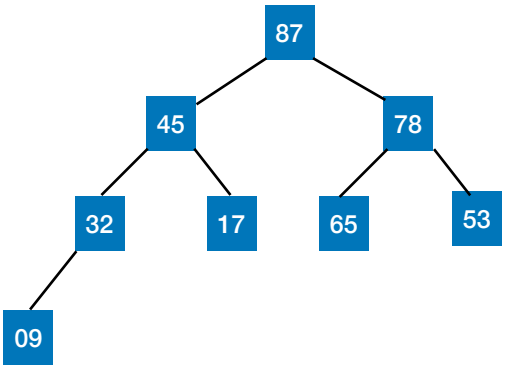
- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——$\lfloor i/2 \rfloor$

建立大根堆

大根堆

	87	45	78	32	17	65	53	09
0	1	2	3	4	5	6	7	8

大根堆：根≥左、右



思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整

检查当前结点是否满足 根≥左、右
若不满足，将当前结点与更大的一个孩子互换

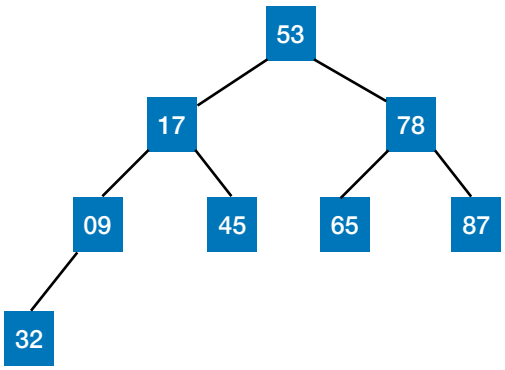
若元素互换破坏了下一级的堆，则采用相同的方法继续往下调整（小元素不断“下坠”）

- i 的左孩子 ——2i
- i 的右孩子 ——2i+1
- i 的父节点 ——⌊i/2⌋

建立大根堆（代码）

初始序列

	53	17	78	09	45	65	87	32
0	1	2	3	4	5	6	7	8

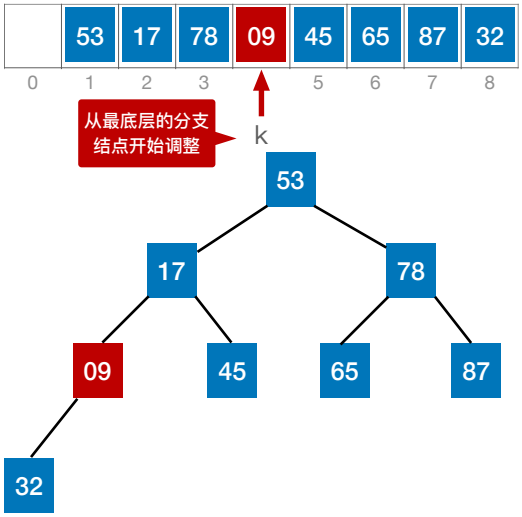



```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}

//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```

建立大根堆（代码）

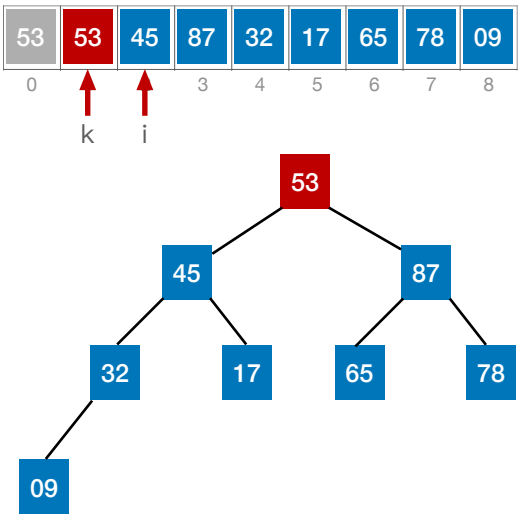
初始序列




```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}

//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```


建立大根堆（代码）

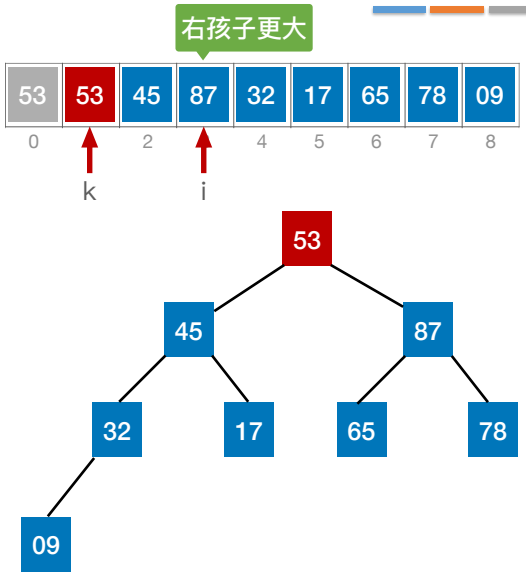



```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}

//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```

王道考研/CSKAOYAN.COM

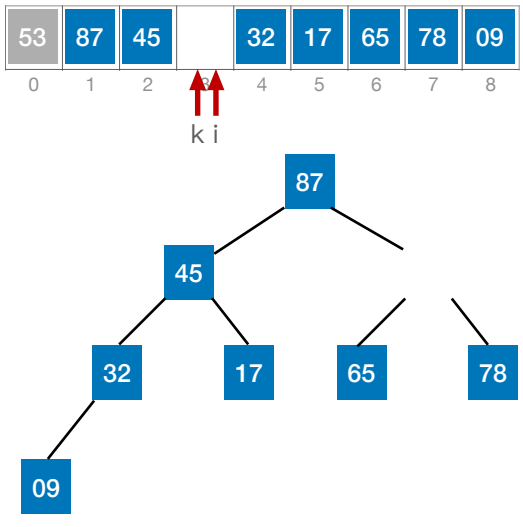
建立大根堆（代码）




```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}
```

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```

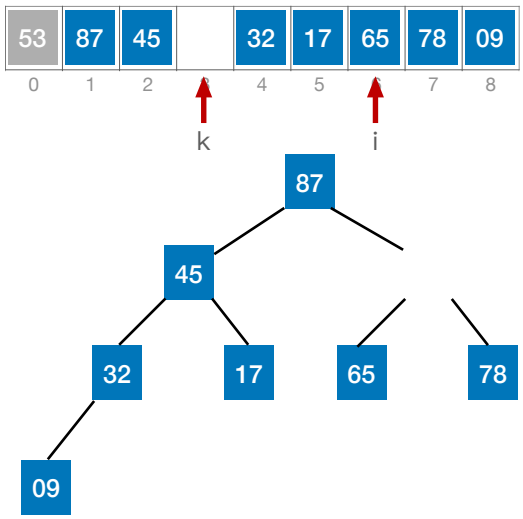

建立大根堆（代码）




```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}
```

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```

建立大根堆（代码）

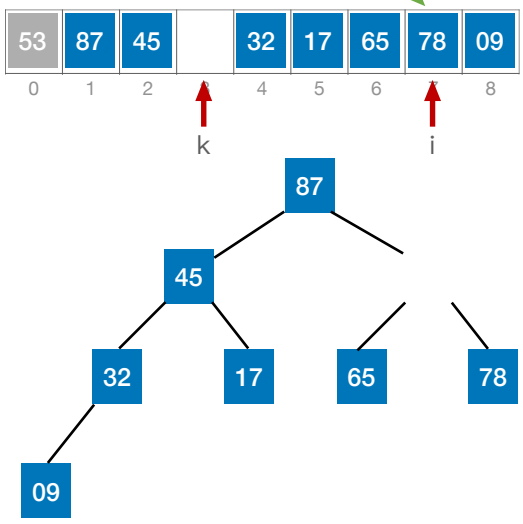



```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}
```

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```


建立大根堆（代码）

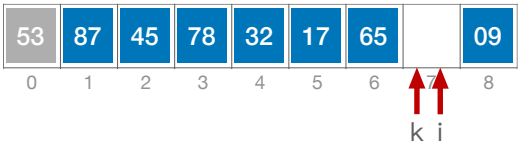
右孩子更大

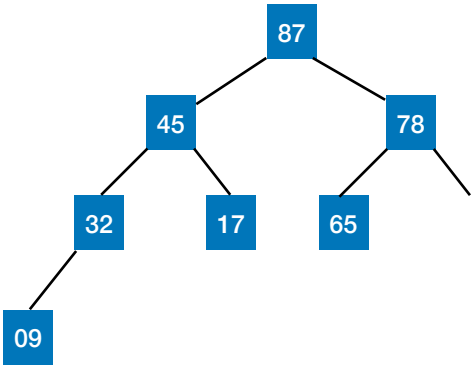



```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}
```

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```

建立大根堆（代码）

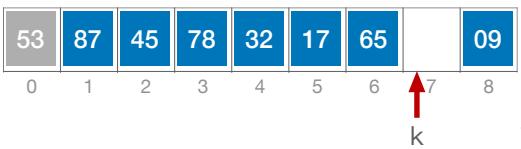


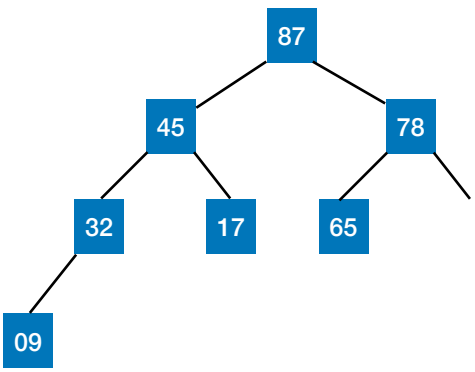



```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}
```

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```


建立大根堆（代码）



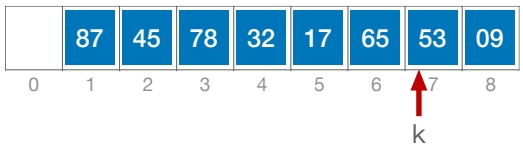


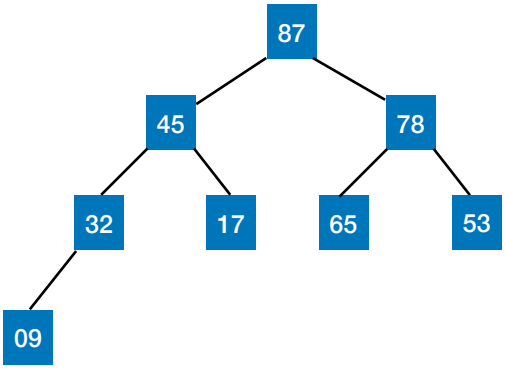

```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}
```

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```

王道考研/CSKAOYAN.COM

建立大根堆（代码）



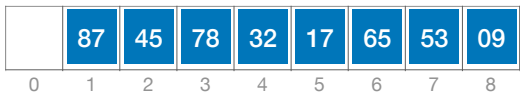



```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}
```

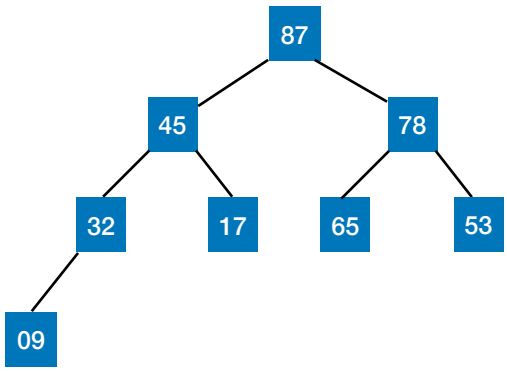
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int i=2*k;i<=len;i*=2){ //沿key较大的子结点向下筛选
        if(i<len&&A[i]<A[i+1])
            i++;                 //取key较大的子结点的下标
        if(A[0]>=A[i])  break;   //筛选结束
        else{
            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
}
```


基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列



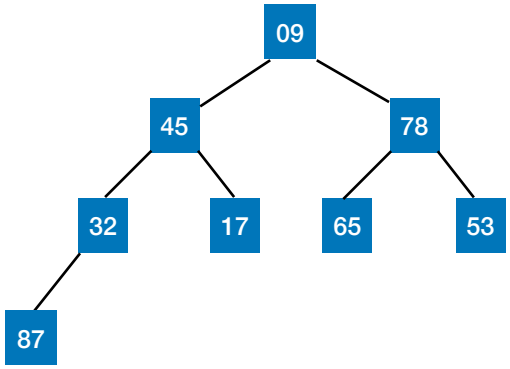
堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

基于大根堆进行排序

大根堆

	09	45	78	32	17	65	53	87
0	1	2	3	4	5	6	7	8

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列



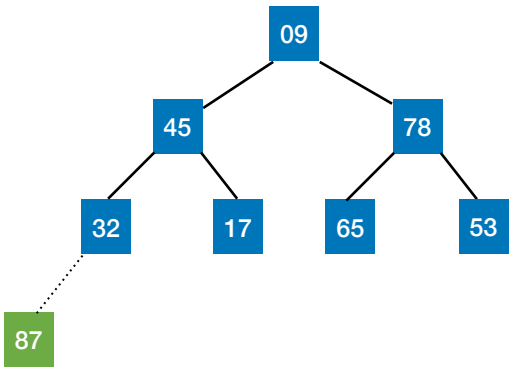
堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

基于大根堆进行排序

大根堆

	09	45	78	32	17	65	53	87
0	1	2	3	4	5	6	7	8

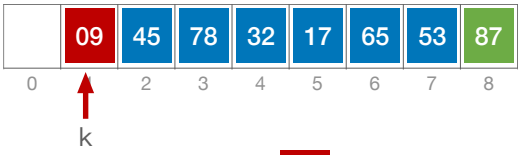
选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

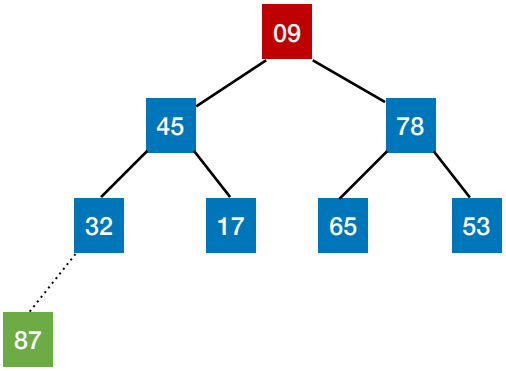


堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

基于大根堆进行排序

大根堆





选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

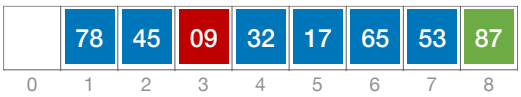
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

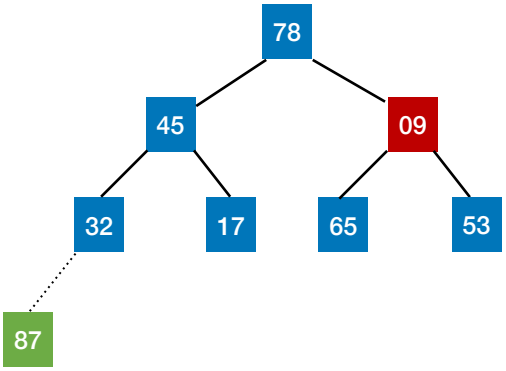
len=7

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```


基于大根堆进行排序

大根堆





选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

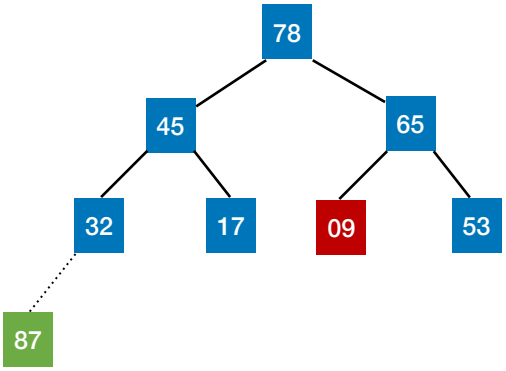
len=7

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

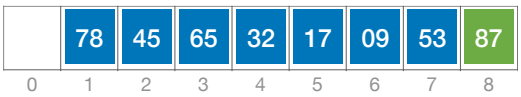
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=7

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

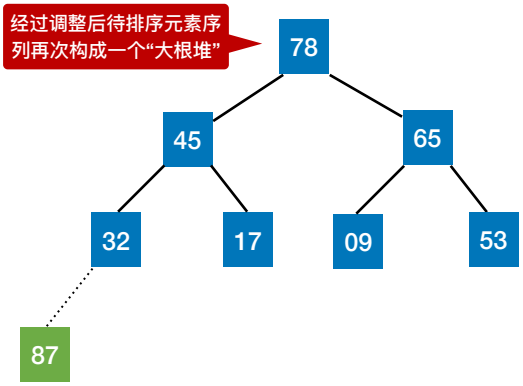

基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

经过调整后待排序元素序列再次构成一个“大根堆”



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=7

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

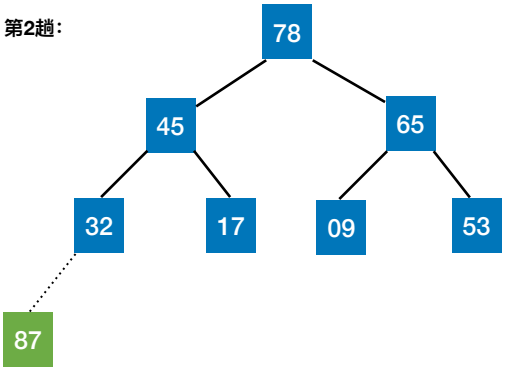
王道考研/CSKAOYAN.COM

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第2趟：

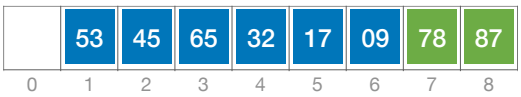


堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

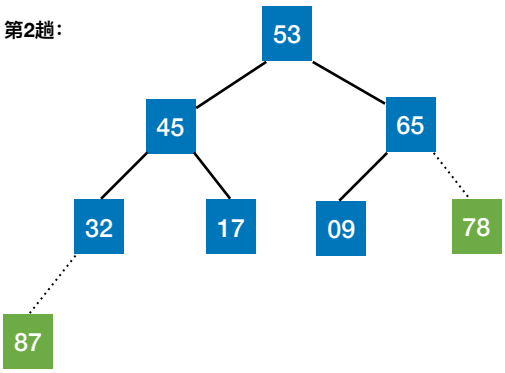
基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第2趟：

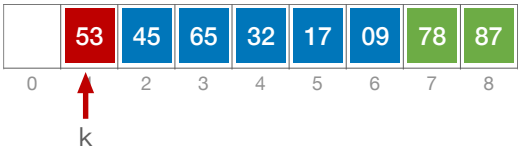


堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

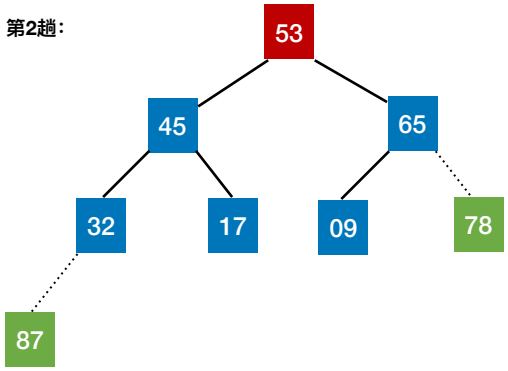
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

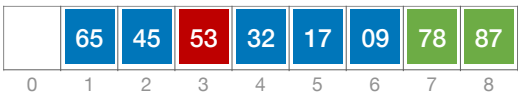
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=6

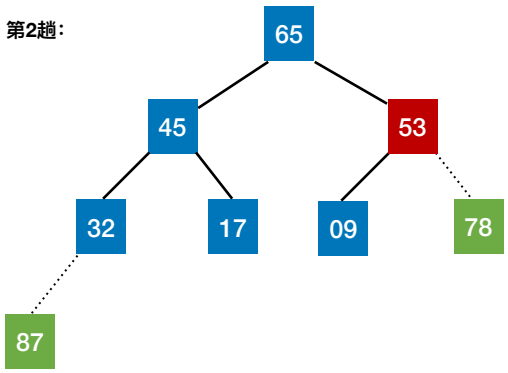
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```


基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

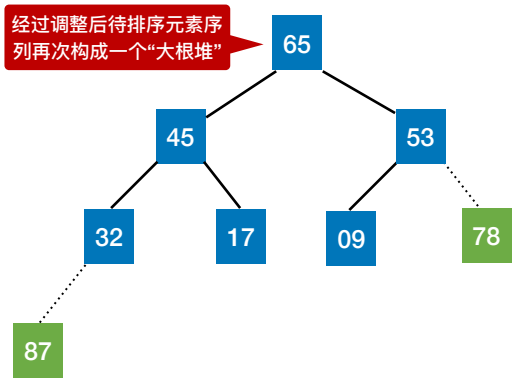
len=6

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

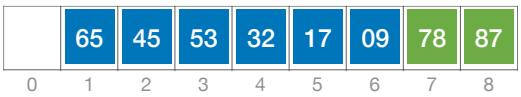
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=6

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

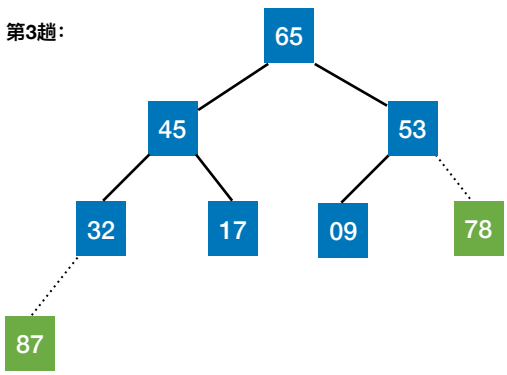

基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第3趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

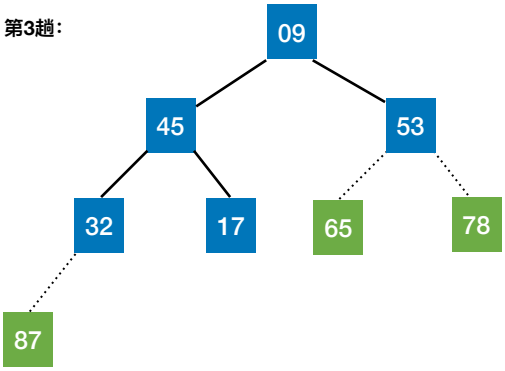
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第3趟：

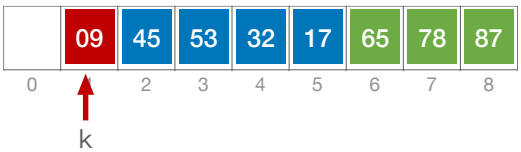


堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

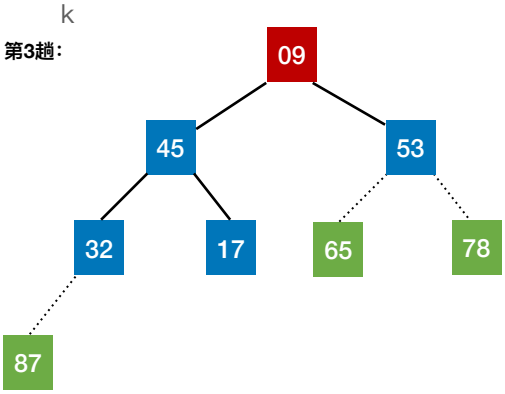
基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第3趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

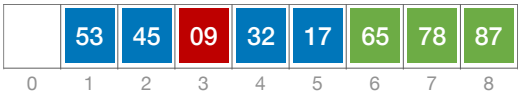
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=5

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

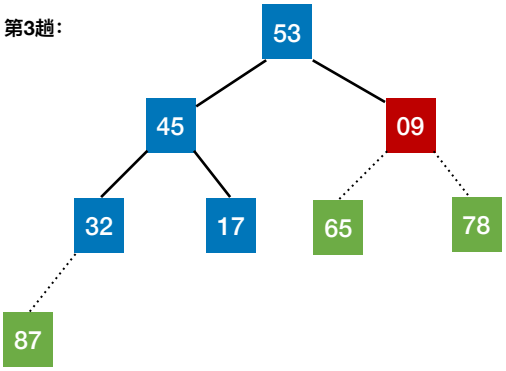
基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第3趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

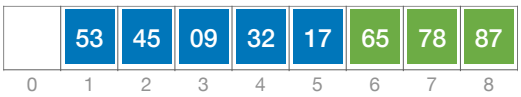
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=5

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

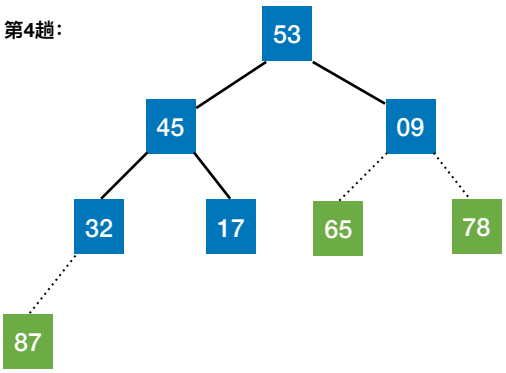

基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第4趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

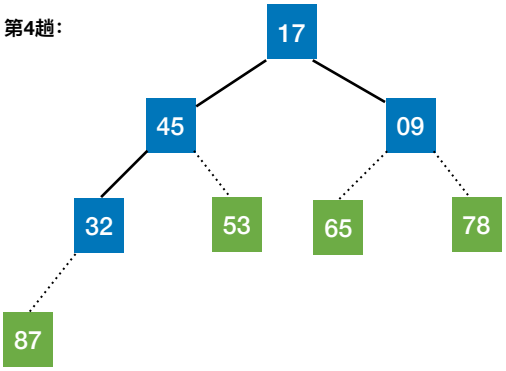
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第4趟：

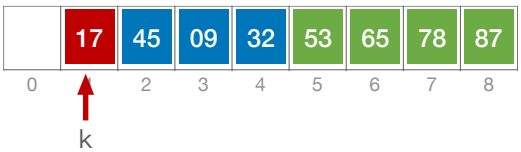


堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

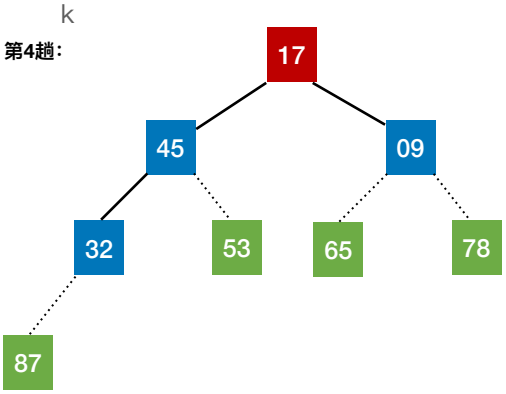
基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第4趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=4

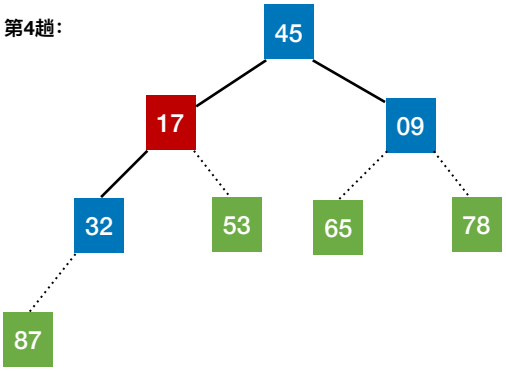
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第4趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=4

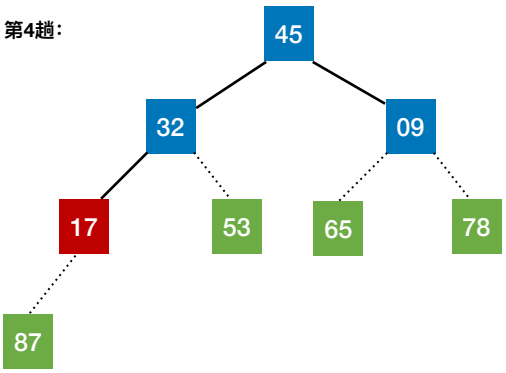
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```


基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第4趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=4

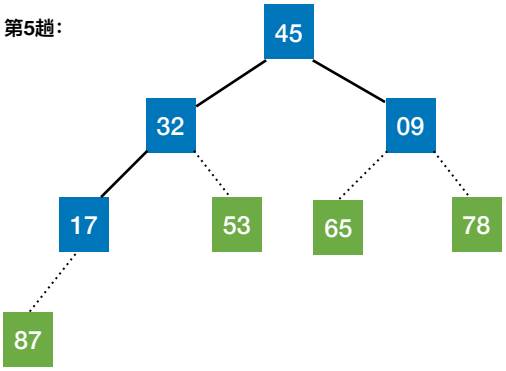
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第5趟：

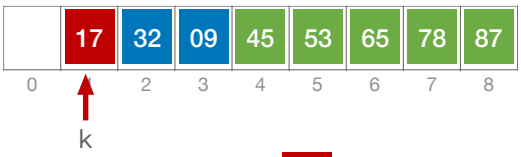


堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

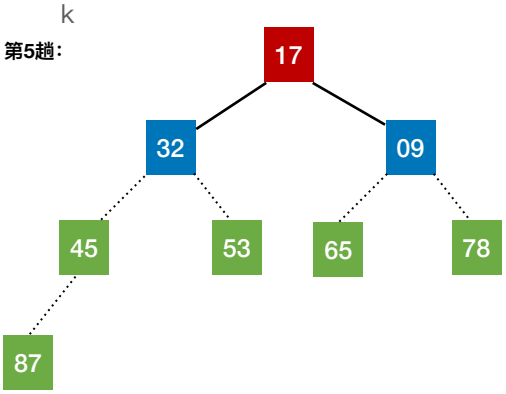
基于大根堆进行排序

大根堆



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第5趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=3

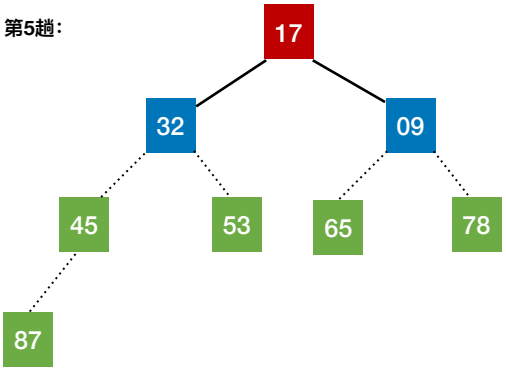
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第5趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=3

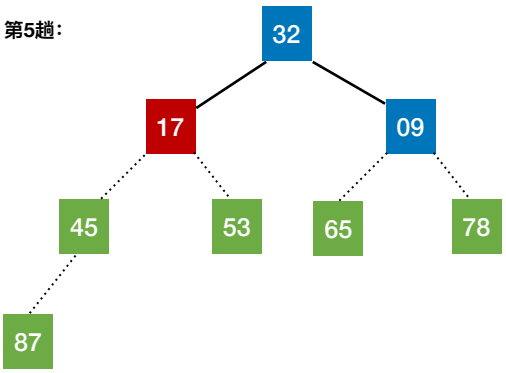
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```


基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第5趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=3

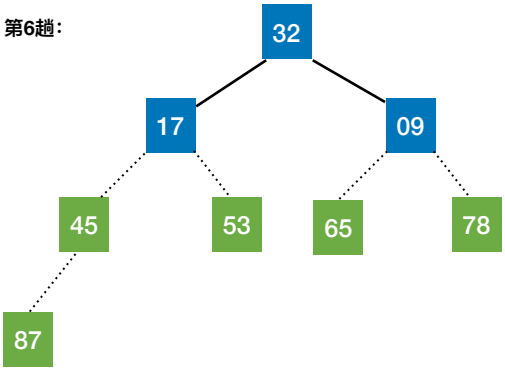
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第6趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

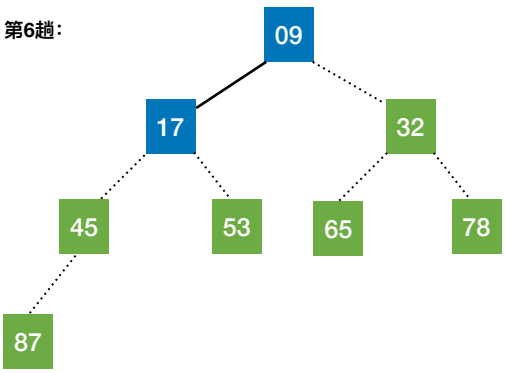
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第6趟：

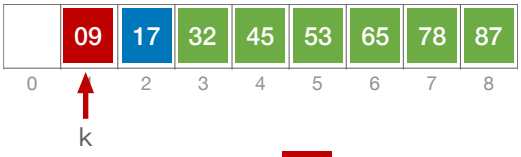


堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

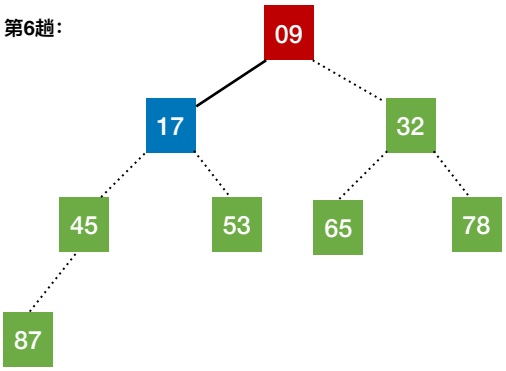
并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

基于大根堆进行排序

大根堆



第6趟：



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

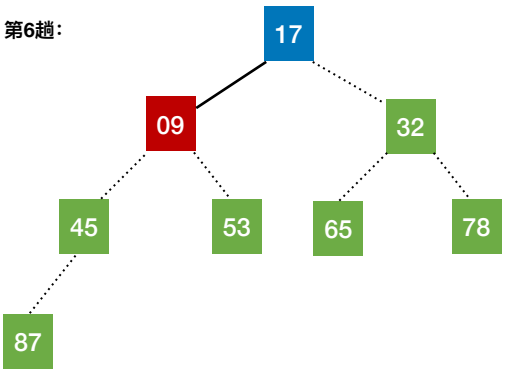
len=2

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```


基于大根堆进行排序

大根堆

第6趟：



选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

len=2

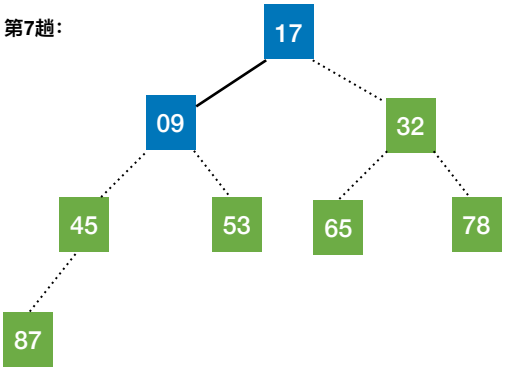
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)
```

基于大根堆进行排序

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第7趟：



堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

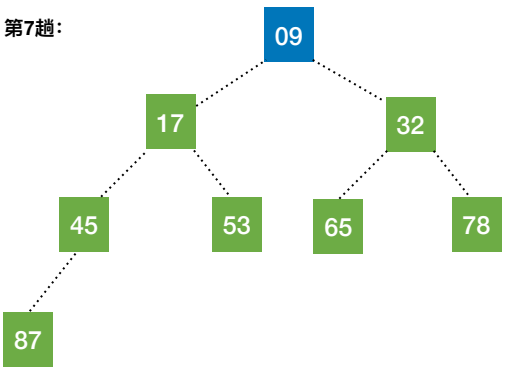
基于大根堆进行排序

只剩下最后一个待排序元素，不用再调整

大根堆

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列

第7趟：



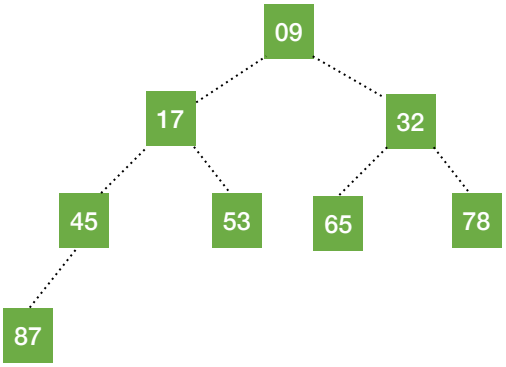
堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

基于大根堆进行排序

n−1趟 处理之后：

选择排序：每一趟在待排序元素中选取关键字最大的元素加入有序子序列



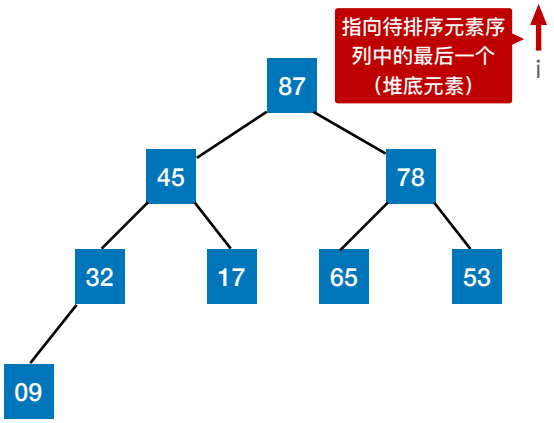
堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

注意：基于“大根堆”的堆排序得到“递增序列”

基于大根堆进行排序（代码）

大根堆




```
//建立大根堆
void BuildMaxHeap(int A[],int len)

//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len)

//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);        //初始建堆
    for(int i=len;i>1;i--){     //n-1趟的交换和建堆过程
        swap(A[i],A[1]);        //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);    //把剩余的待排序元素整理成堆
    }
}
```

堆排序：每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换）

并将待排序元素序列再次调整为大根堆（小元素不断“下坠”）

算法效率分析

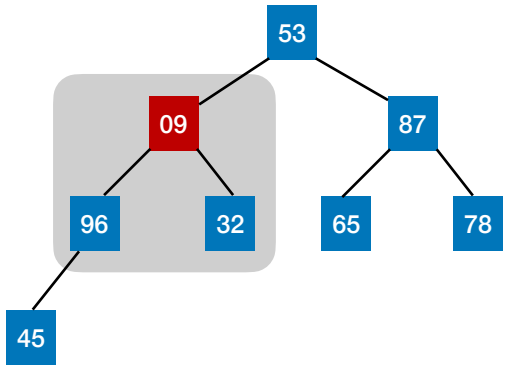
```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}


//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);         //初始建堆
    for(int i=len;i>1;i--){      //n-1趟的交换和建堆过程
        swap(A[i],A[1]);         //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);     //把剩余的待排序元素整理成堆
    }
}
```

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```


算法效率分析

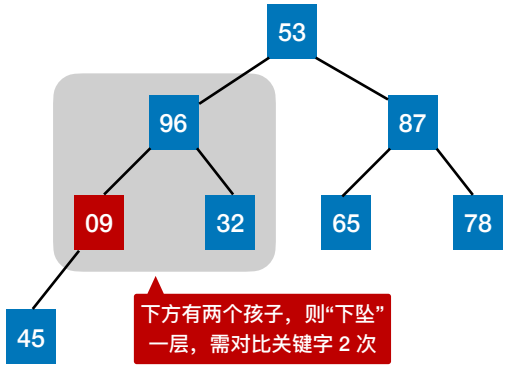

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int i=2*k;i<=len;i*=2){ //沿key较大的子结点向下筛选
        if(i<len&&A[i]<A[i+1])
            i++;                 //取key较大的子结点的下标
        if(A[0]>=A[i])  break;   //筛选结束
        else{
            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
}
```

王道考研/CSKAOYAN.COM

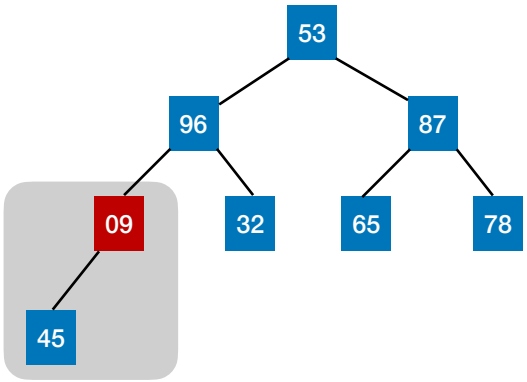
算法效率分析

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                    //A[0]暂存子树的根结点
    for(int 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
}
```



算法效率分析

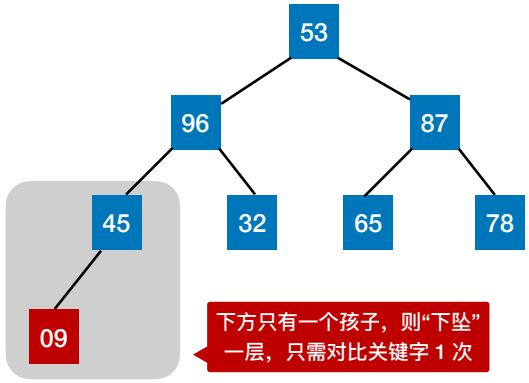
```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                    //A[0]暂存子树的根结点
    for(int i=2*k;i<=len;i*=2){ //沿key较大的子结点向下筛选
        if(i<len&&A[i]<A[i+1])
            i++;                  //取key较大的子结点的下标
        if(A[0]>=A[i])  break;   //筛选结束
        else{
            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
}
```



王道考研/CSKAOYAN.COM

算法效率分析

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                    //A[0]暂存子树的根结点
    for(int 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
}
```



结论：一个结点，每“下坠”一层，最多只需对比关键字2次

若树高为h，某结点在第 i 层，则将这个结点向下调整最多只需要“下坠” h-i 层，关键字对比次数不超过 2(h-i)

算法效率分析

结论：一个结点，每“下坠”一层，最多只需对比关键字2次

若树高为h，某结点在第 i 层，则将这个结点向下调整最多只需要“下坠” h-i 层，关键字对比次数不超过 2(h-i)

n个结点的完全二叉树树高 h=$\lfloor log_2 n \rfloor + 1$

第 i 层最多有 2^{i-1} **个结点，而只有第 1 ~ (h-1) 层的结点才有可能需要“下坠”调整**

差比数列求和（错位相减法）

将整棵树调整为大根堆，关键字对比次数不超过 $\sum_{i=h-1}^{1} 2^{i-1}2(h-i) = \sum_{i=h-1}^{1} 2^{i}(h-i) = \sum_{j=1}^{h-1} 2^{h-j}j \le 2n\sum_{j=1}^{h-1}\frac{j}{2^j} \le 4n$

求和结果小于2

建堆的过程，关键字对比次数不超过$4n$，建堆时间复杂度=$O(n)$

算法效率分析

```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}

//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);          //初始建堆
    for(int i=len;i>1;i--){       //n-1趟的交换和建堆过程
        swap(A[i],A[1]);          //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);      //把剩余的待排序元素整理成堆
    }
}
```

$O(n)$

总共需要n-1趟，每一趟交换后都需要将根节点“下坠”调整

```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                    //A[0]暂存子树的根结点
    for(int 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
}
```

根节点最多“下坠” h-1 层，每下坠一层

而每“下坠”一层，最多只需对比关键字2次，因此每一趟排序复杂度不超过 $O(h)=O(log_2n)$

共n-1 趟，总的时间复杂度 = $O(nlog_2n)$

算法效率分析


```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}

//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);          //初始建堆
    for(int i=len;i>1;i--){       //n-1趟的交换和建堆过程
        swap(A[i],A[1]);          //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);      //把剩余的待排序元素整理成堆
    }
}
```


$O(n)$

$O(nlog_2n)$


```
//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                    //A[0]暂存子树的根结点
    for(int 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
}
```

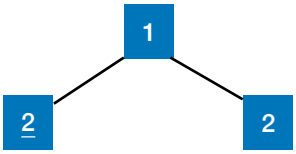

堆排序的时间复杂度 = $O(n)+O(nlog_2n)=O(nlog_2n)$

堆排序的空间复杂度 = $O(1)$

王道考研/CSKAOYAN.COM

稳定性

初始序列：



初始化大根堆

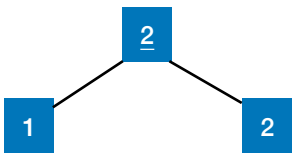
注意：若左右孩子一样大，则优先和左孩子交换

```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}

//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```


稳定性

大根堆：根≥左、右



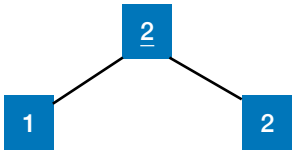
注意：若左右孩子一样大，则优先和左孩子交换

```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}

//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int 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
}
```

稳定性

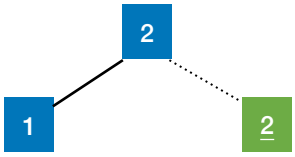
大根堆：根≥左、右




```
//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);          //初始建堆
    for(int i=len;i>1;i--){       //n-1趟的交换和建堆过程
        swap(A[i],A[1]);          //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);      //把剩余的待排序元素整理成堆
    }
}
```


稳定性

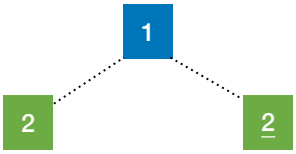
大根堆：根≥左、右




```
//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);          //初始建堆
    for(int i=len;i>1;i--){       //n-1趟的交换和建堆过程
        swap(A[i],A[1]);          //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);      //把剩余的待排序元素整理成堆
    }
}
```

稳定性

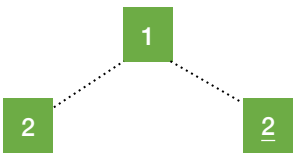
大根堆：根≥左、右




```
//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);          //初始建堆
    for(int i=len;i>1;i--){       //n-1趟的交换和建堆过程
        swap(A[i],A[1]);          //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);      //把剩余的待排序元素整理成堆
    }
}
```


稳定性

排序结果：



初始序列：


```
//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);          //初始建堆
    for(int i=len;i>1;i--){       //n-1趟的交换和建堆过程
        swap(A[i],A[1]);          //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);      //把剩余的待排序元素整理成堆
    }
}
```

结论：堆排序是不稳定的

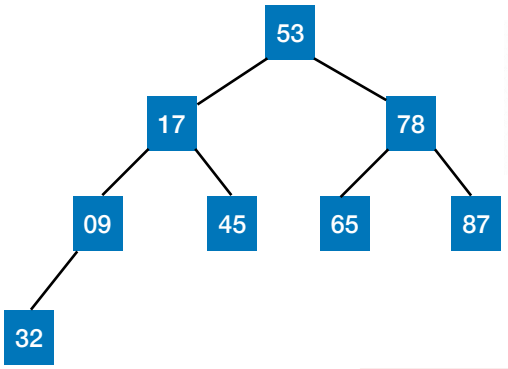
截屏练习：基于“小根堆”如何建堆、排序?

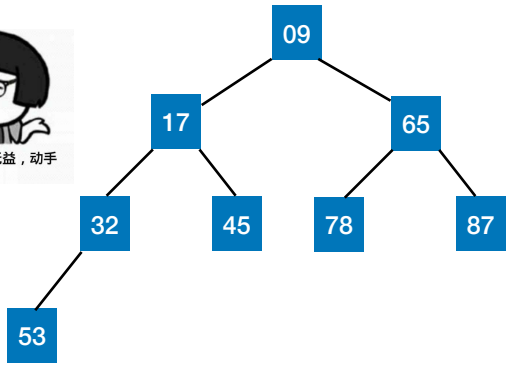
初始序列

0	1	2	3	4	5	6	7	8
	53	17	78	09	45	65	87	32

初始化小根堆

0	1	2	3	4	5	6	7	8
	09	17	65	32	45	78	87	53





注意：基于“小根堆”的堆排序得到“递减序列”

知识回顾与重要考点