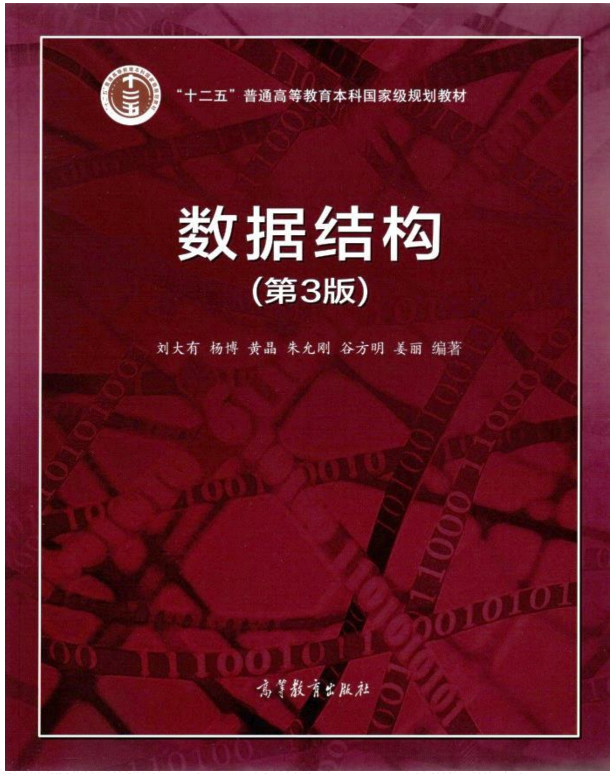
think.create.solve



堆排序

- 锦标赛排序
- 堆排序算法
- 堆与优先级队列
- 可合并堆
- Top K问题

数据之法
结构之美
算法之道

Last updated on 2022.11

zhuyungang@jlu.edu.cn

楼天城（ID: ACRush, 人称楼教主）
中国大学生编程第一人
清华大学博士
2次获ACM/ICPC全球总决赛金牌亚军
2次获Google全球编程挑战赛冠军
2次获Facebook骇客杯世界编程大赛季军
2次获百度之星程序设计大赛冠军
小马智行创始人兼CTO
身价75亿元人民币（胡润财富榜）

第一名可能会让你望尘莫及，直到最后也没办法追上。在此之前，要有一个正确的心态，不能自暴自弃，更不能妄自菲薄。然后要很实实在在的努力，每次必须做的事要做好，先把距离咬住再说。

人生发展是个积累的过程，基本上99%的成功都是通过日日夜夜的积累完成的。努力是你唯一能依赖的东西。

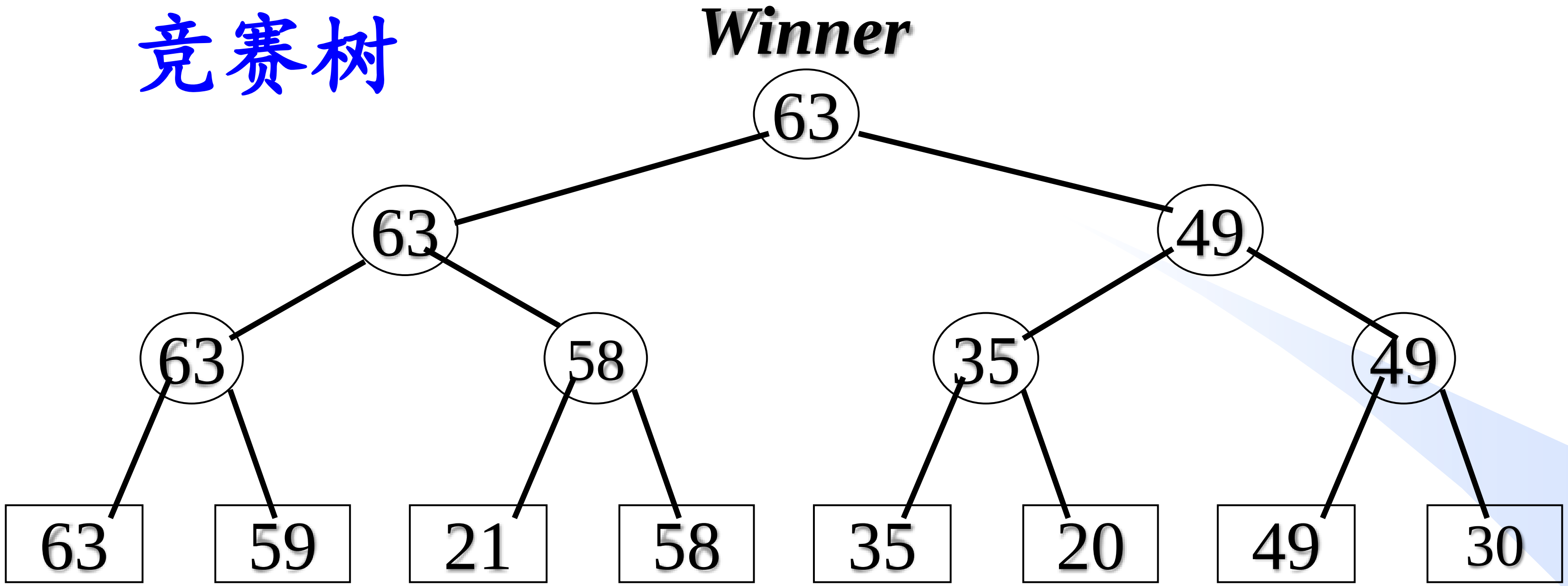
- 选择排序的关键

  找最大（小）记录的方法

- 锦标赛排序

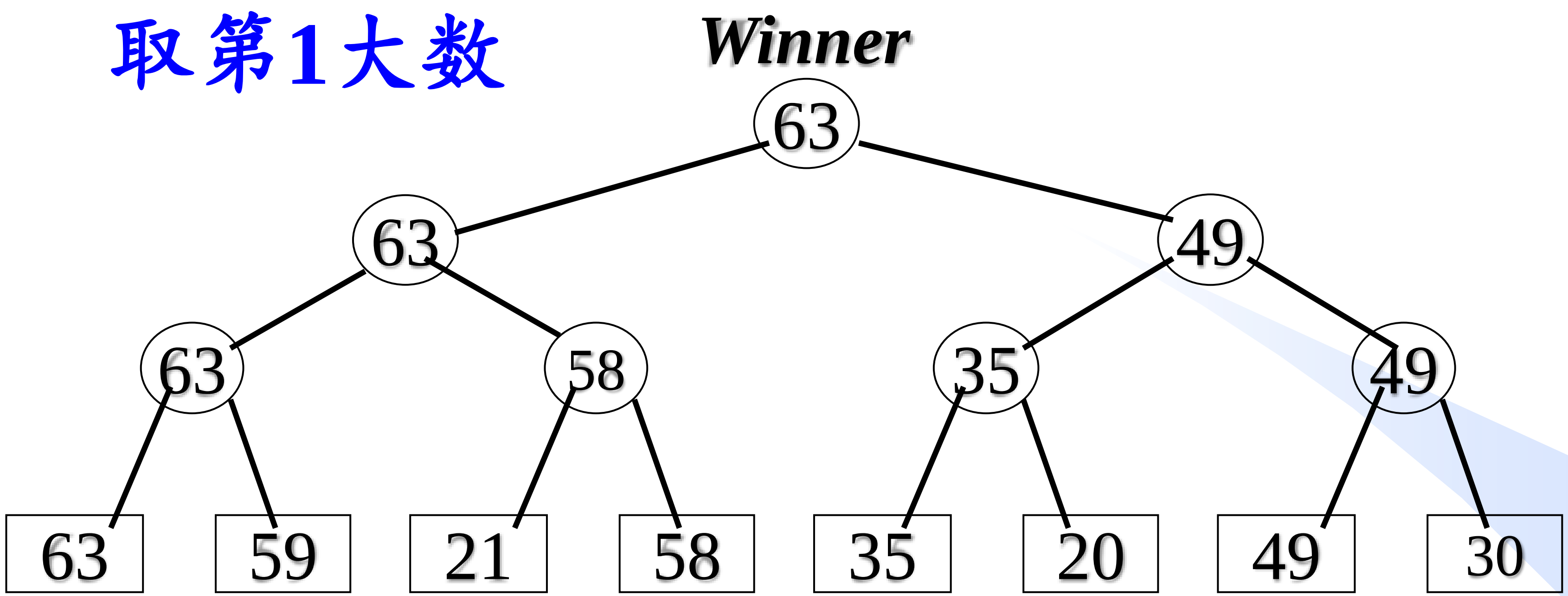
锦标赛排序是对简单选择排序方法的改进，简单选择排序算法的时间大部分都浪费在关键词的比较上，而锦标赛排序刚好用树形保存了前面比较的结果，下一次选择时直接利用前面比较的结果大大减少了比较次数。

竞赛树



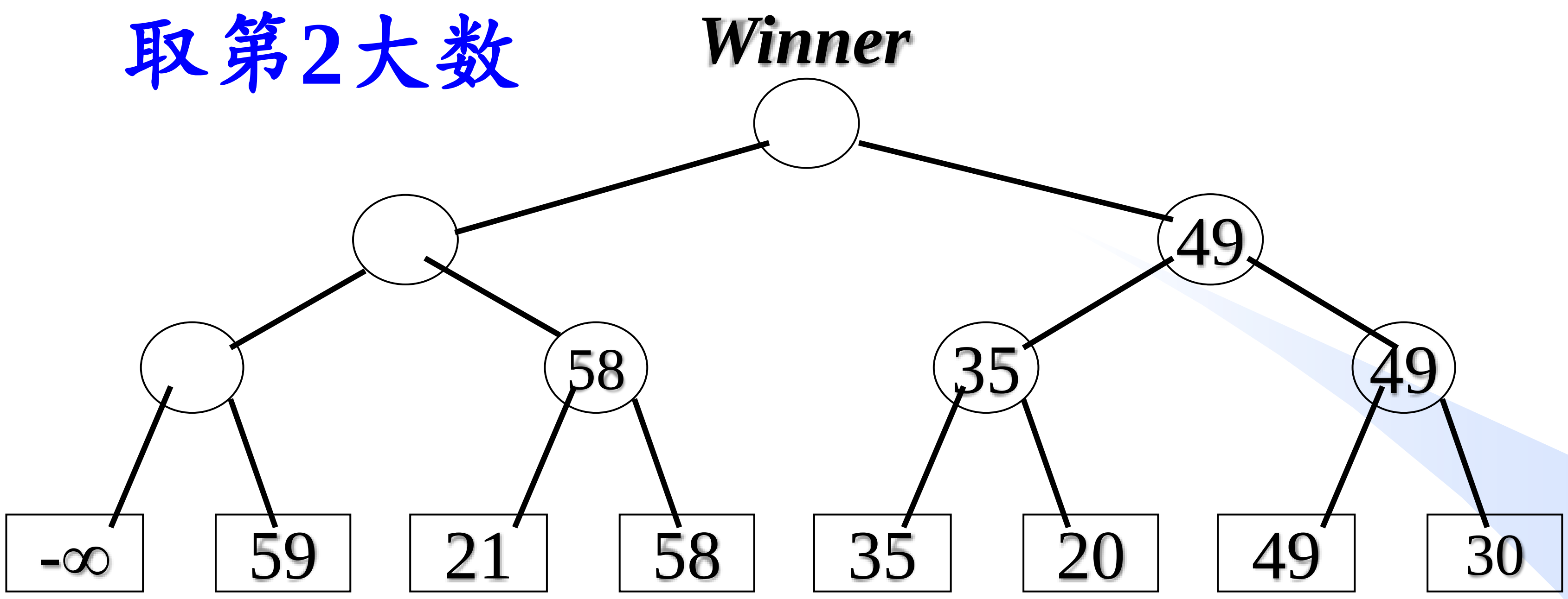
- 满二叉树；
- 叶结点：存放待排序元素的关键词；
- 非叶结点：存放关键词两两比较的结果；
- n 非 2 的 幂时，叶结点补足到满足2^k，$2^{k-1} < n \le 2^k$

取第1大数

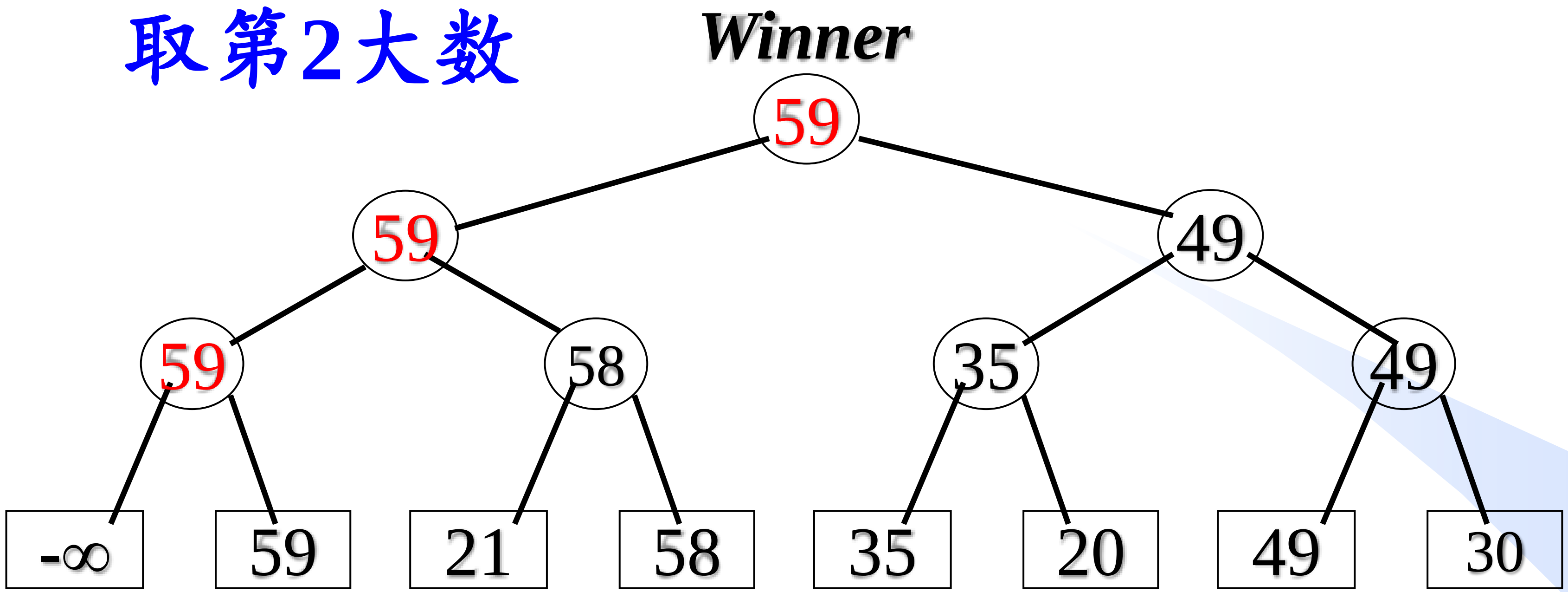


- 最大关键词上升到根
- 关键词比较次数O(n)

取第2大数

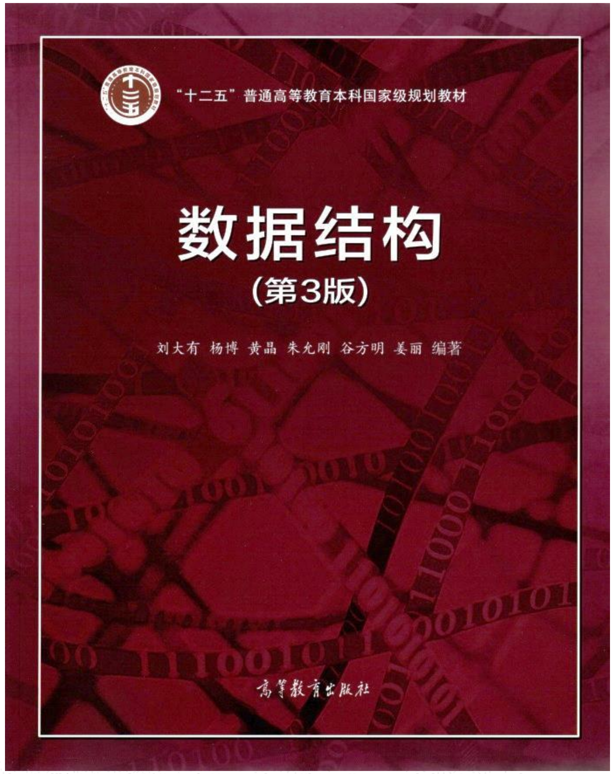


取第2大数



- 将剩余记录调整为新的竞赛树，得到第2大记录
- 关键词比较次数O(log*n*)
- 总时间复杂度$O(n) + O(\log n)+\ldots+ O(\log n)= O(n\log n)$
- 对于*n*个待排序记录，锦标赛算法至少需要2*n*-1个结点来存放竞赛树，故这是一个拿空间换时间的算法。

think.create.solve



堆排序

- 锦标赛排序
- **堆排序算法**
- 堆与优先级队列
- 可合并堆
- Top K问题

数据之法
结构之美
算法之道

Last updated on 2022.11

zhuyungang@jlu.edu.cn

堆排序

- **核心操作：** 在当前子数组中选最大的记录，并移到最后
- **方法：** 利用**堆**
- 由**J.W.J. Williams、Robert Floyd**提出。

ALGORITHM 232
HEAPSORT
J. W. J. WILLIAMS (Recd 1 Oct. 1963 and, revised, 15 Feb. 1964)
Elliott Bros. (London) Ltd., Borehamwood, Herts, England

comment The following procedures are related to *TREESORT* [R. W. Floyd, Alg. 113, *Comm. ACM 5* (Aug. 1962), 434, and A. F. Kaupe, Jr., Alg. 143 and 144, *Comm. ACM 5* (Dec. 1962), 604] but avoid the use of pointers and so preserve storage space. All the procedures operate on single word items, stored as elements 1 to n of the array A. The elements are normally so arranged that $A[i] \leqq A[j]$ for $2 \leqq j \leqq n$, $i = j \div 2$. Such an arrange-

ALGORITHM 245
TREESORT 3 [M1]
ROBERT W. FLOYD (Recd. 22 June 1964 and 17 Aug. 1964)
Computer Associates, Inc., Wakefield, Mass.

```
procedure TREESORT 3 (M, n);
  value n; array M; integer n;
```

comment *TREESORT* 3 is a major revision of *TREESORT* [R. W. Floyd, Alg. 113, *Comm. ACM 5* (Aug. 1962), 434] suggested by *HEAPSORT* [J. W. J. Williams, Alg. 232, *Comm. ACM 7* (June 1964), 347] from which it differs in being an in-place sort. It is shorter and probably faster, requiring fewer comparisons and only one division. It sorts the array $M[1:n]$, requiring no more than $2 \times (2 \uparrow p-2) \times (p-1)$, or approximately $2 \times n \times (\log_2(n)-1)$ comparisons and half as many exchanges in the worst case to sort $n = 2 \uparrow p - 1$ items. The algorithm is

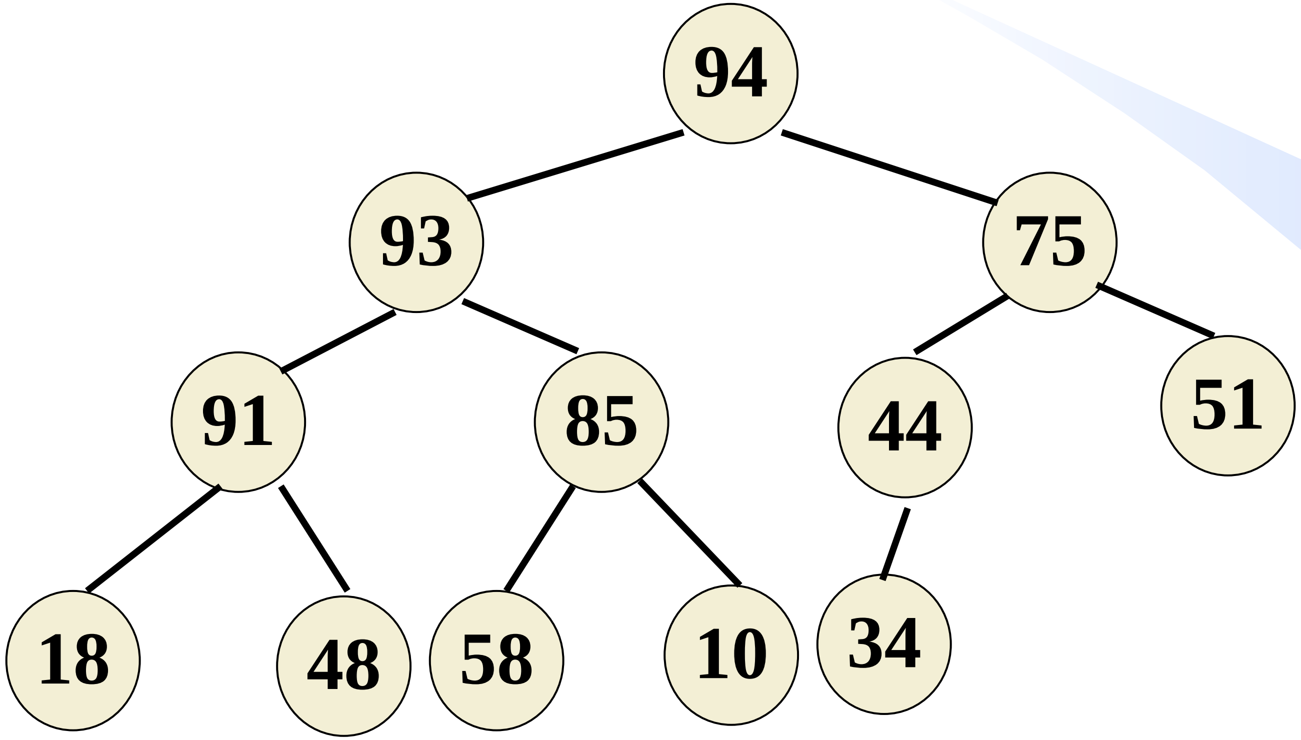
[1] **J. W. J. Williams**. *Heapsort*, Communications of the ACM, 7(6):378-348, 1964.
[2] **Robert W. Floyd**. *Treesort 3*, Communications of the ACM, 7(12):701, 1964.

Coca-Cola
Estrella Damm
movistar
LA VANGUARDIA
SEREM
3 3
BASF
REPSOL
WE WANT TO VOTE!
TARRAGONA

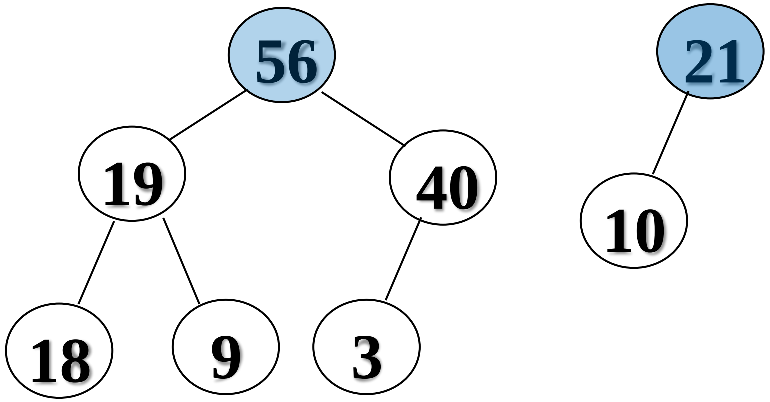
堆（Heap）

- 最大堆（大根堆）：一棵完全二叉树，其中任意结点的关键词大于等于它的两个孩子的关键词。
  - ✓ 结构性：完全二叉树。
  - ✓ 堆序性：任意结点的关键词大于等于其孩子的关键词。
- 最小堆（小根堆）：一棵完全二叉树，其中任意结点的关键词小于等于它的两个孩子的关键词。
- 最大堆中根结点的关键词最大，最小堆中根结点的关键词最小。

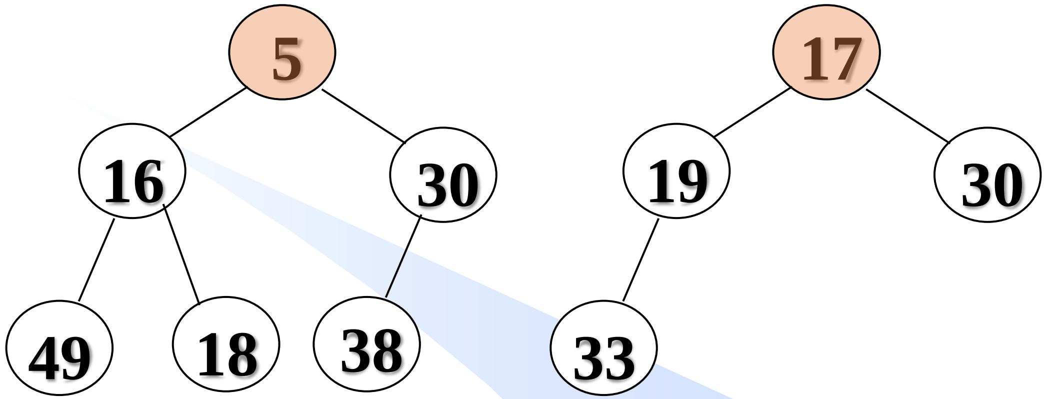
最大堆

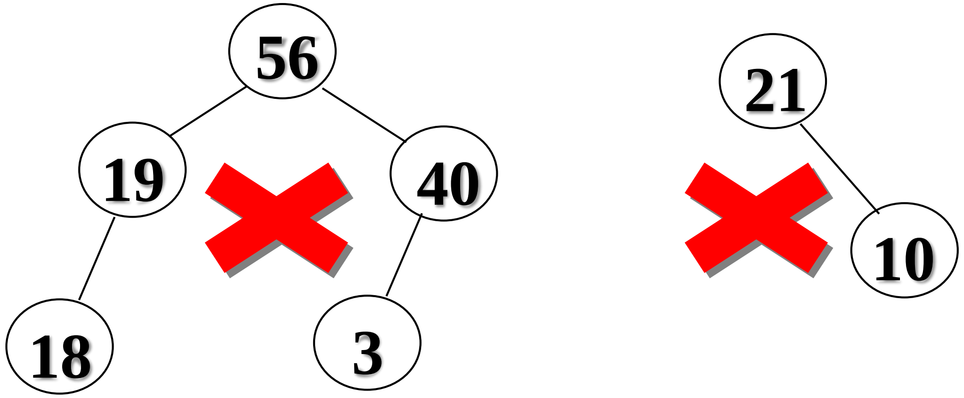


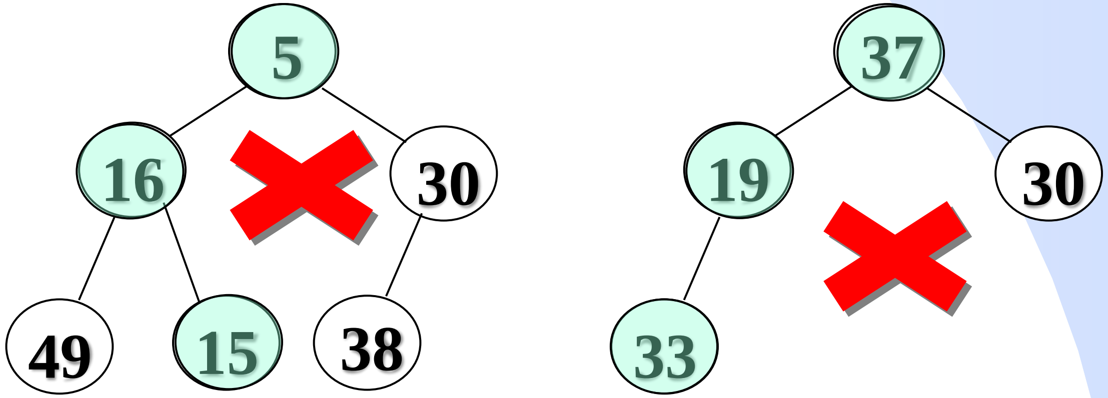
最大堆



最小堆







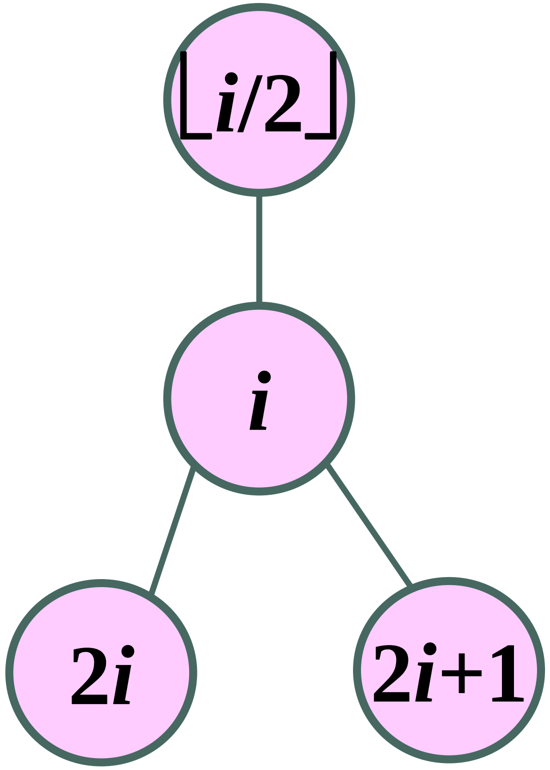
不满足结构性

不是堆

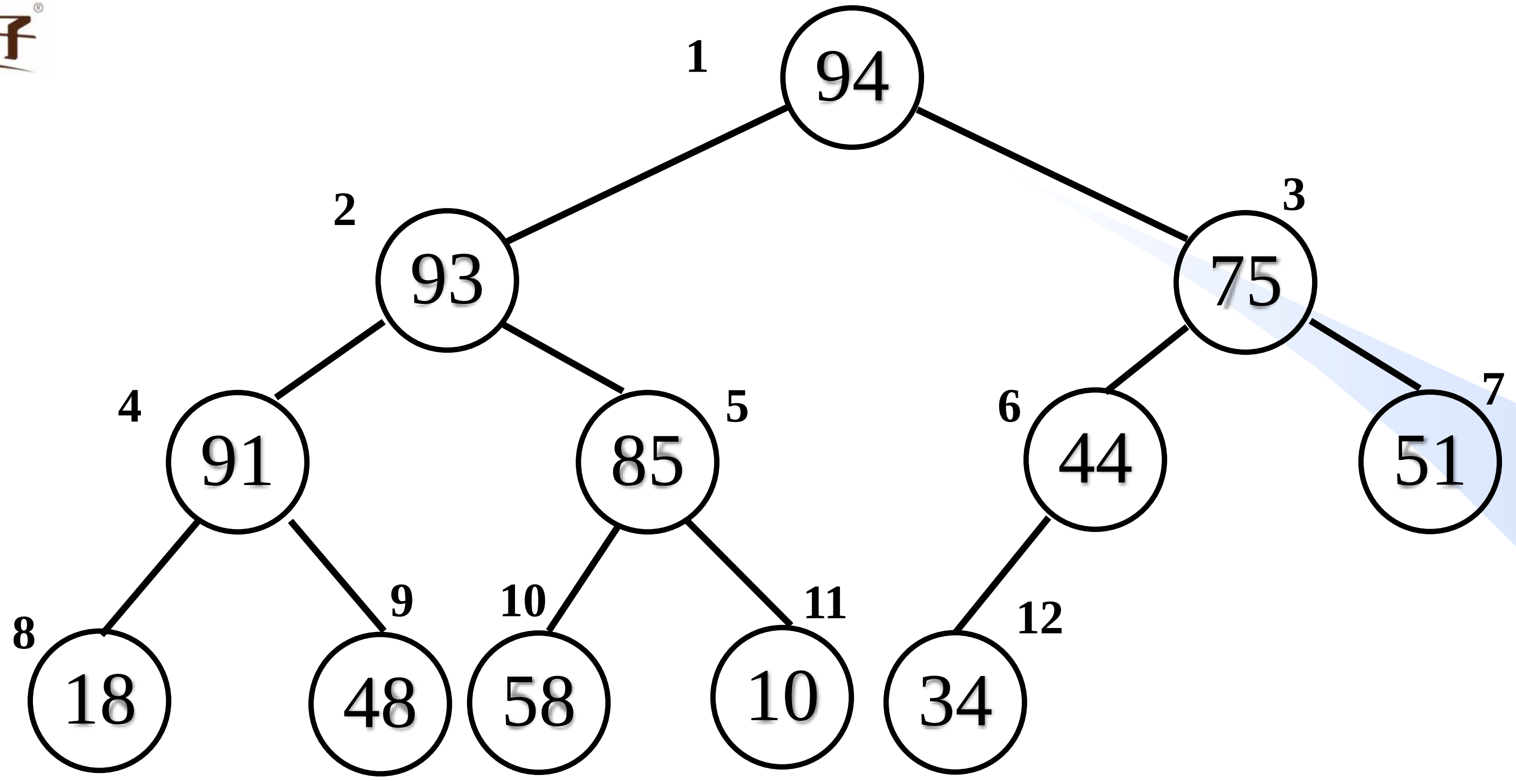
不满足堆序性

回顾： 完全二叉树的顺序存储

$R[1]$存储二叉树的根结点。结点$R[i]$的左孩子（若有的话）存放在$R[2i]$处，而$R[i]$的右孩子（若有的话）存放在$R[2i+1]$处。$R[i]$的父结点是$R[i/2]$。



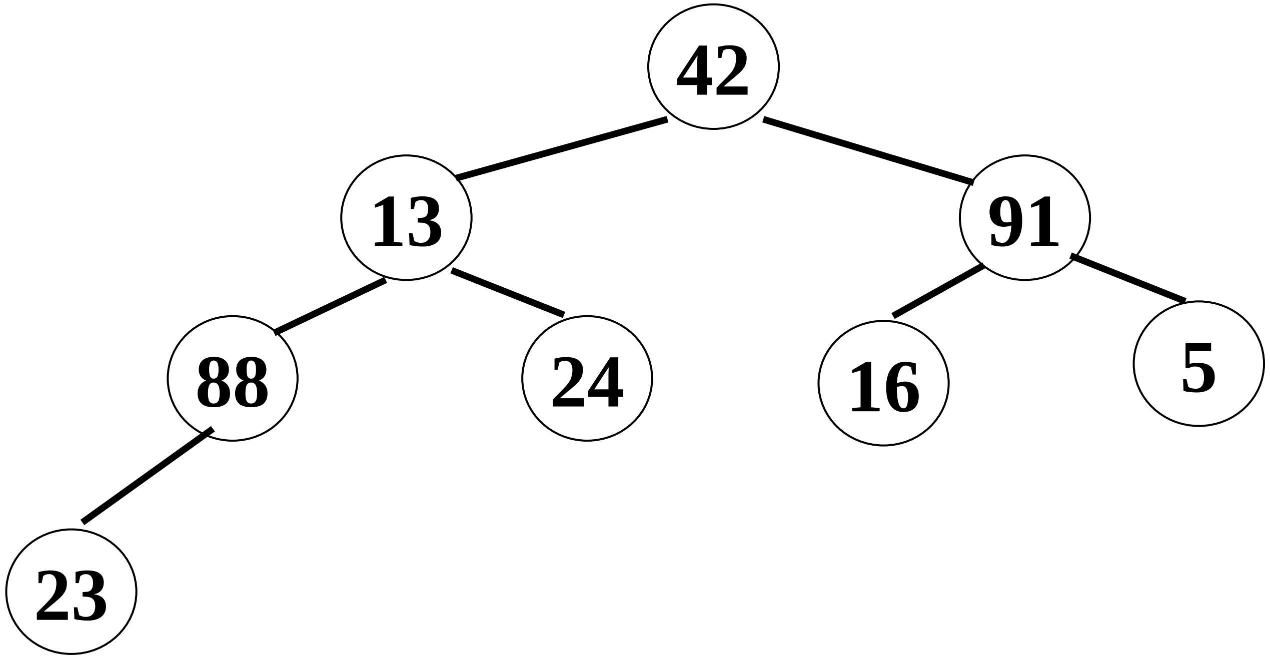
数组存储堆



i	1	2	3	4	5	6	7	8	9	10	11	12
R[*i*]	94	93	75	91	85	44	51	18	48	58	10	34

堆排序算法的思想

R	42	13	91	88	24	16	5	23

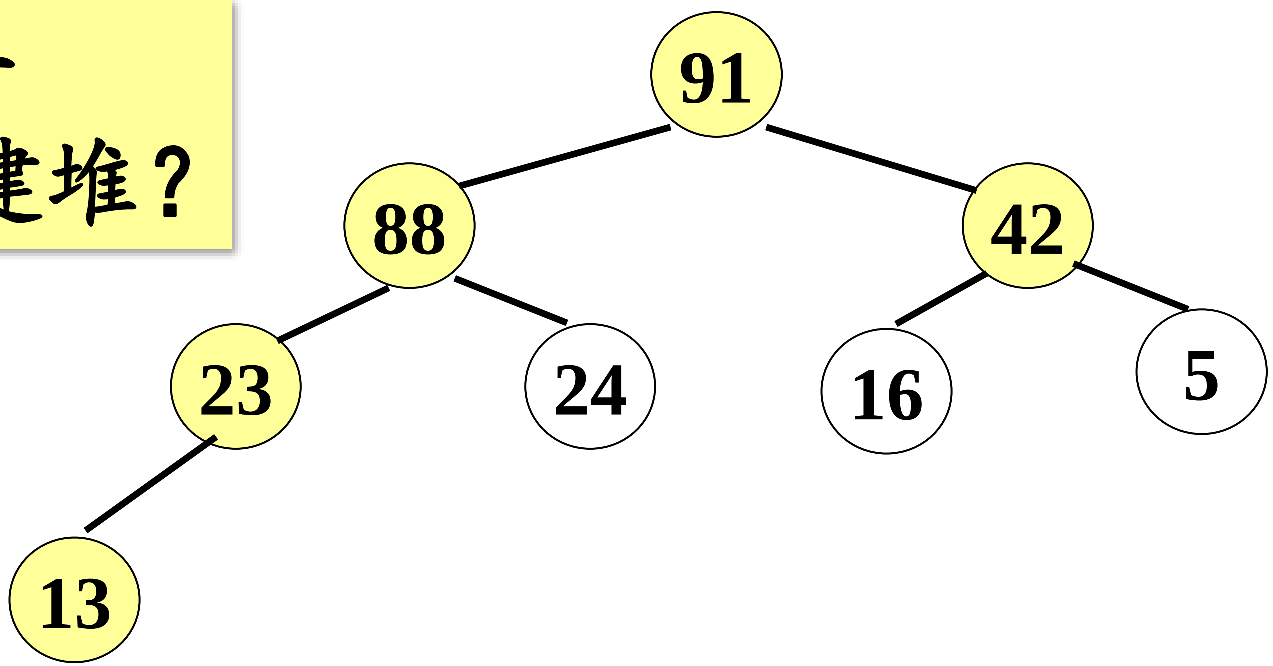


堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

R	91	88	42	23	24	16	5	13

问题一
如何初始建堆？

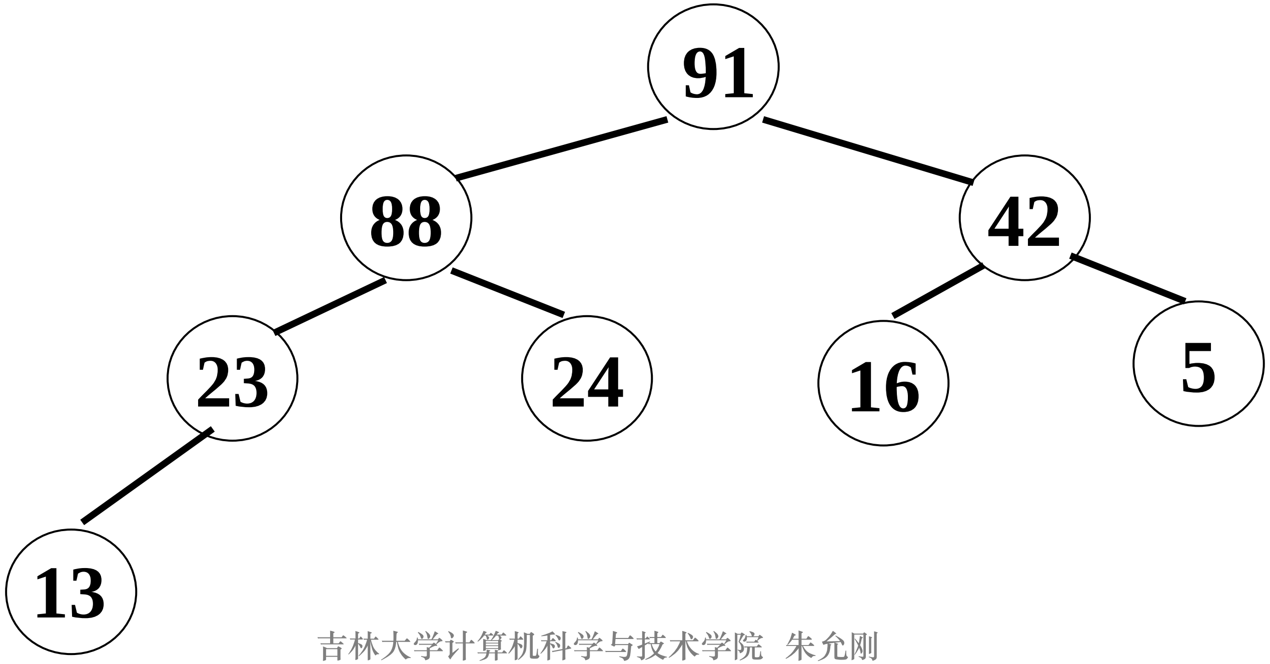


堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

（2）R[1]是关键词最大的记录，将R[1]和R[*n*]互换，使最大记录放在R[*n*]的位置。

R	91	88	42	23	24	16	5	13

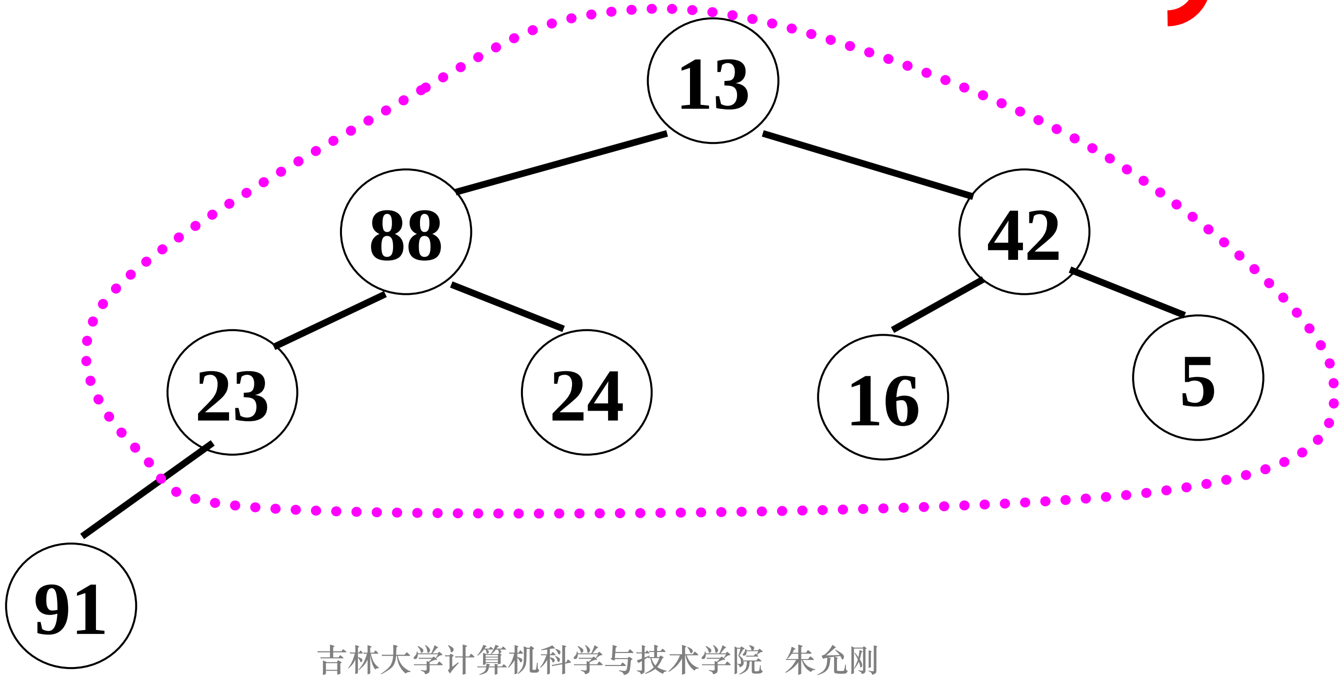


堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

（2）R[1]是关键词最大的记录，将R[1]和R[*n*]互换，使最大记录放在R[*n*]的位置。

R	13	88	42	23	24	16	5	91



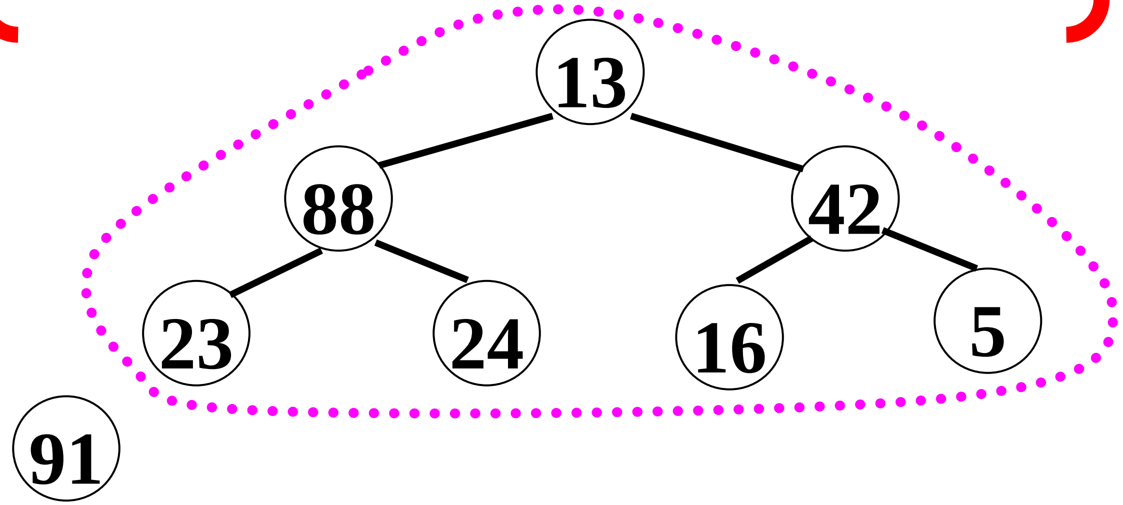
堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

（2）R[1]是关键词最大的记录，将R[1]和R[*n*]互换，使最大记录放在R[n]的位置。

（3）调整R[1], … , R[*n*-1]，将其重建为新堆，则R[1]是其中最大者，再交换R[1]与R[*n*-1]，使R[*n*-1]中放入第2大记录。

R	13	88	42	23	24	16	5	91



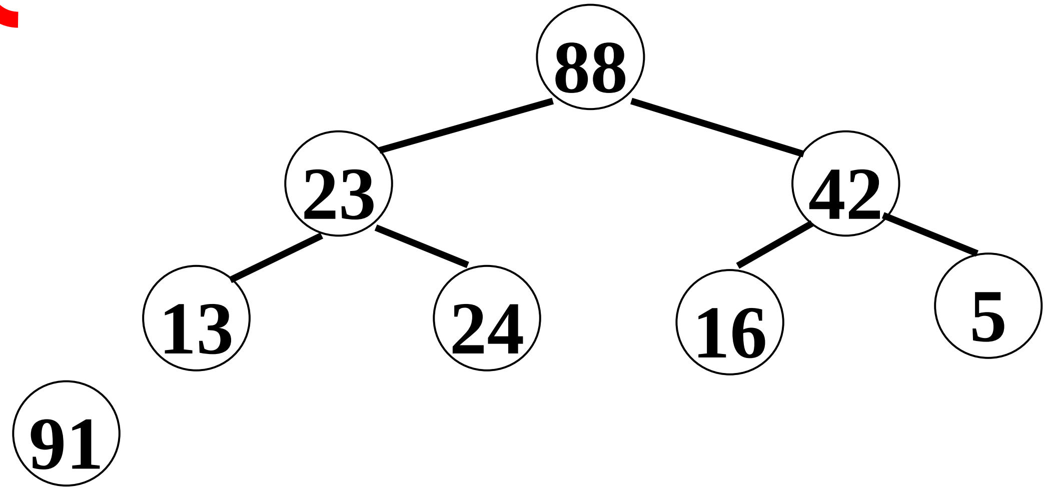
堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

（2）R[1]是关键词最大的记录，将R[1]和R[*n*]互换，使最大记录放在R[*n*]的位置。

（3）调整R[1], … , R[*n*-1]，将其重建为新堆，则R[1]是其中最大者，再交换R[1]与R[*n*-1]，使R[*n*-1]中放入第2大记录。

R								
	88	23	42	13	24	16	5	91



问题二
如何重建堆？

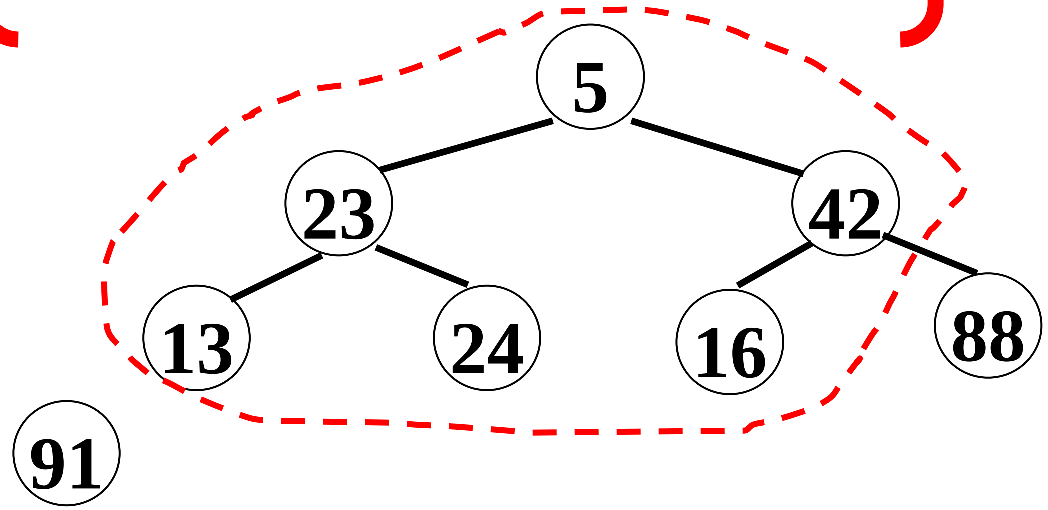
堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

（2）R[1]是关键词最大的记录，将R[1]和R[*n*]互换，使最大记录放在R[*n*]的位置。

（4）调整R[1], … , R[*n*-2]，将其重建为新堆，则R[1]是其中最大者，再交换R[1]与R[*n*-2]，使R[*n*-2]中放入第3大记录。

R								
	5	23	42	13	24	16	88	91



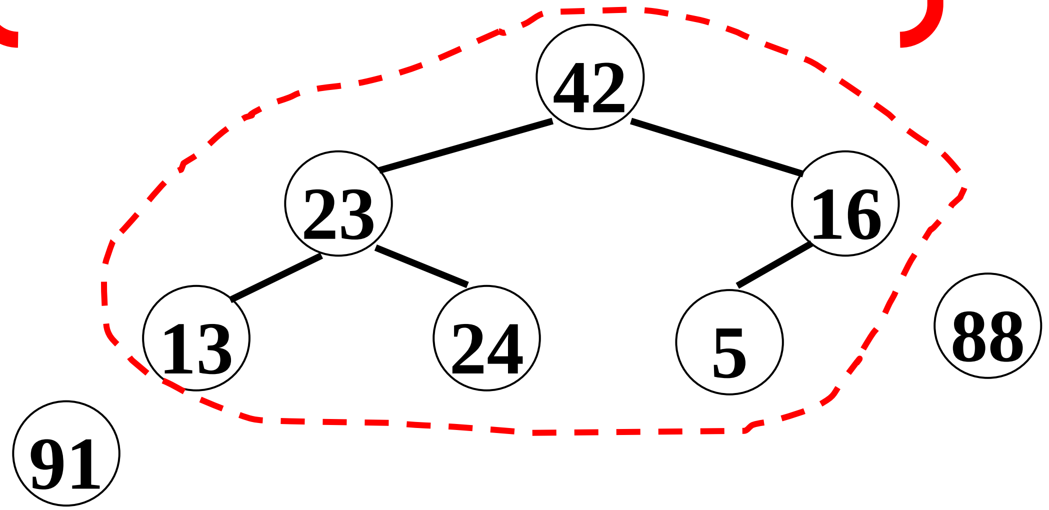
堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

（2）R[1]是关键词最大的记录，将R[1]和R[*n*]互换，使最大记录放在R[*n*]的位置。

（4）调整R[1], … , R[*n*-2]，将其重建为新堆，则R[1]是其中最大者，再交换R[1]与R[*n*-2]，使R[*n*-2]中放入第3大记录。

R								
	42	23	16	13	24	5	88	91



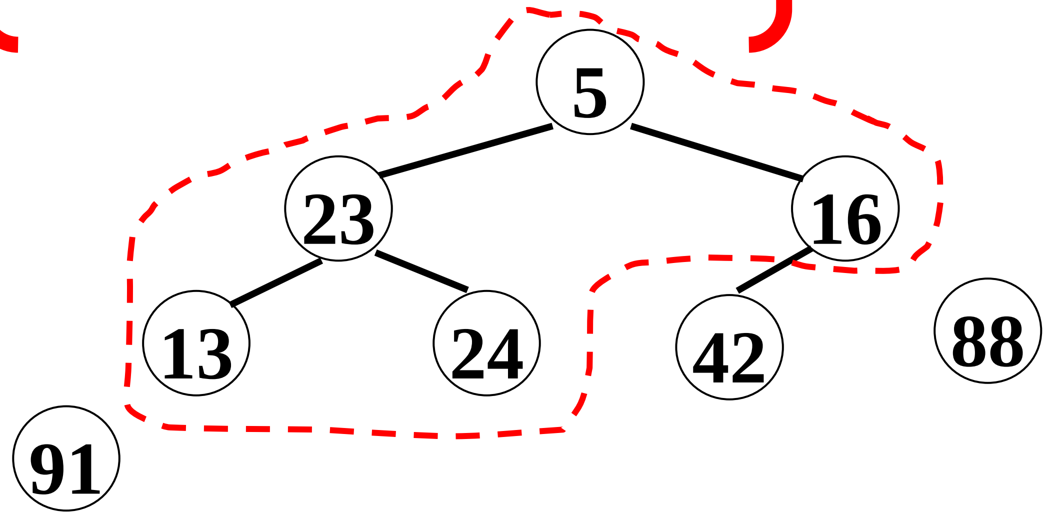
堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

（2）R[1]是关键词最大的记录，将R[1]和R[*n*]互换，使最大记录放在R[*n*]的位置。

（5）调整R[1], … , R[*n*-3]，将其重建为新堆，则R[1]是其中最大者，再交换R[1]与R[*n*-3]，使R[*n*-3]中放入第4大记录。

R	5	23	16	13	24	42	88	91



堆排序算法的思想

（1）将待排序数组*R*建成一个最大堆。

（2）R[1]是关键词最大的记录，将R[1]和R[*n*]互换，使最大记录放在R[*n*]的位置。

（3）调整R[1]，…，R[*n*-1]，将其重建为新堆，则R[1]是其中最大者，再交换R[1]与R[*n*-1]，使R[*n*-1]中放入第2大记录。

（4）调整R[1]，R[2]，…，R[*n*-2]，将其重建为新堆，再交换R[1]与R[*n*-2]。

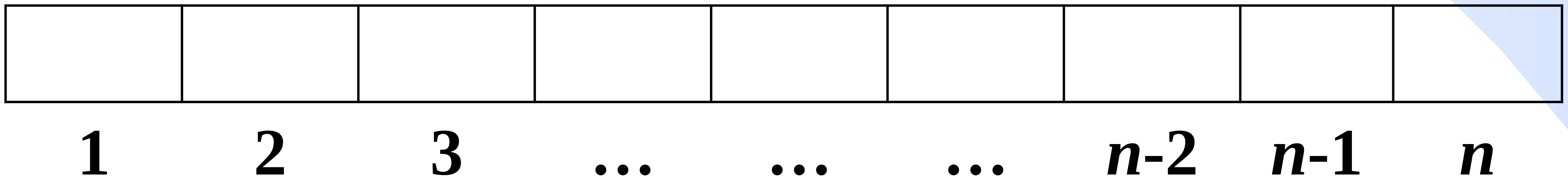
……

上述操作反复进行，直到调整范围只剩下一个记录R[1]为止。此时，R[1]是*n*个记录中最小的，且数组R中的记录已按从小到大排列。

堆排序算法的粗略描述如下：

(1) 建立包含$R[1], R[2], \ldots, R[n]$的堆；

(2)
```
for(int i=n; i>=2; i--){
        R[1]↔ R[i];   //i标识当前处理的堆的右边界
        重建包含R[1], R[2], …, R[i-1]的堆
}
```



堆排序需解决的两个问题：

① 如何初始建堆

② 输出堆顶元素后，如何重建新堆

虽然操作的是数组，但背后隐藏的灵魂是二叉树

重建堆

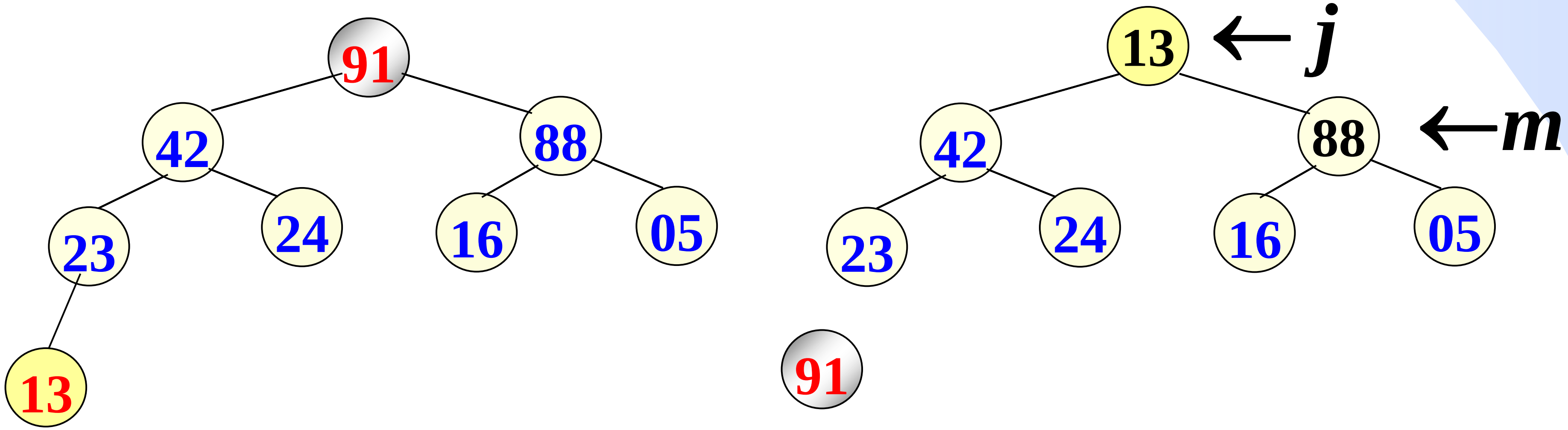
由于交换前为堆结构，则$R[1]$与$R[i]$交换后，只有$R[1]$与其左右孩子不满足堆的性质。

（1）令j指向堆根结点（$j \leftarrow$根结点在数组R中的下标）。

（2）将$R[j]$与其关键词最大的孩子$R[m]$比较；

（3）若$R[j] \geq R[m]$，则已满足堆序性，算法结束；

（4）若$R[m] > R[j]$，则交换$R[j]$与$R[m]$，$j \leftarrow m$，返回（2）



重建堆

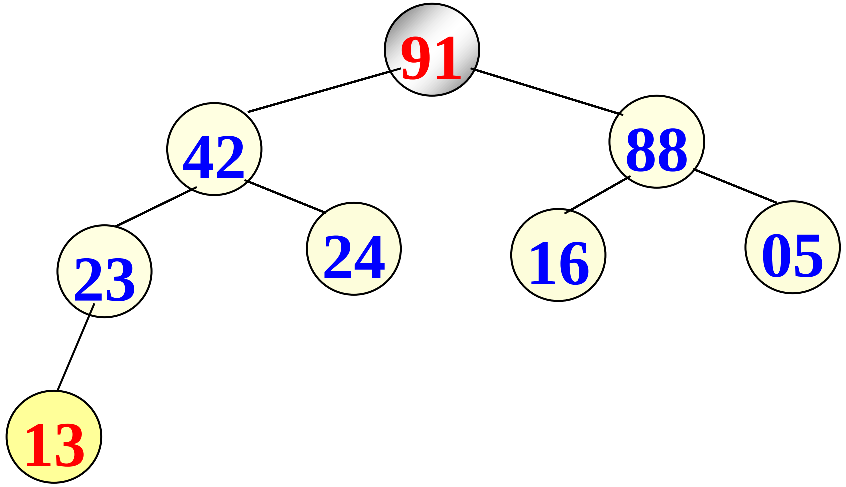
由于交换前为堆结构，则$R[1]$与$R[i]$交换后，只有$R[1]$与其左右孩子不满足堆的性质。

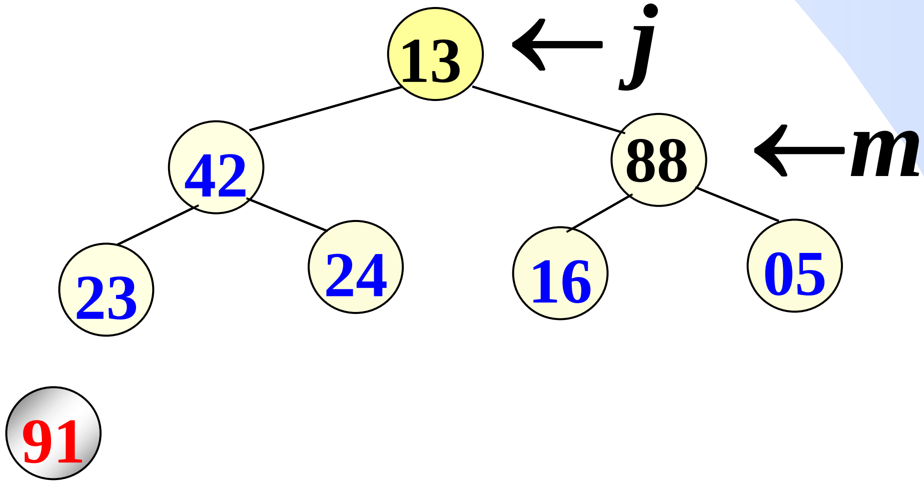
（1）令j指向堆根结点（$j \leftarrow$根结点在数组R中的下标）。

（2）将$R[j]$与其关键词最大的孩子$R[m]$比较；

（3）若$R[j] \geq R[m]$，则已满足堆序性，算法结束；

（4）若$R[m] > R[j]$，则交换$R[j]$与$R[m]$，$j \leftarrow m$，返回（2）





重建堆

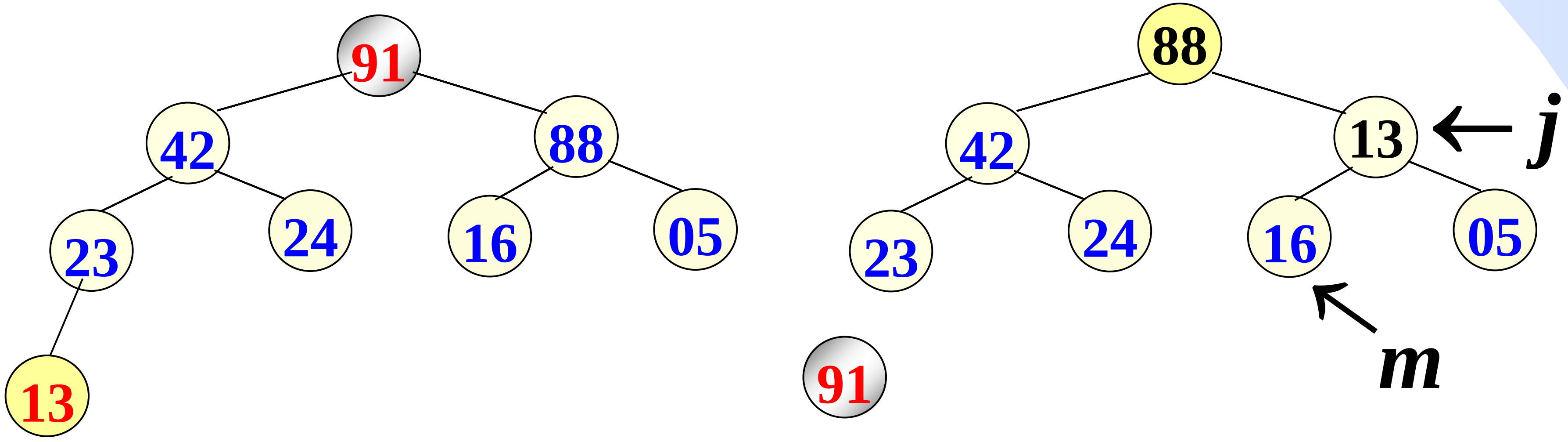
由于交换前为堆结构，则$R[1]$与$R[i]$交换后，只有$R[1]$与其左右孩子不满足堆的性质。

（1）令j指向堆根结点（$j\leftarrow$根结点在数组R中的下标）。

（2）将$R[j]$与其关键词最大的孩子$R[m]$比较；

（3）若$R[j] \geq R[m]$，则已满足堆序性，算法结束；

（4）若$R[m] > R[j]$，则交换$R[j]$与$R[m]$，$j\leftarrow m$，返回（2）



重建堆

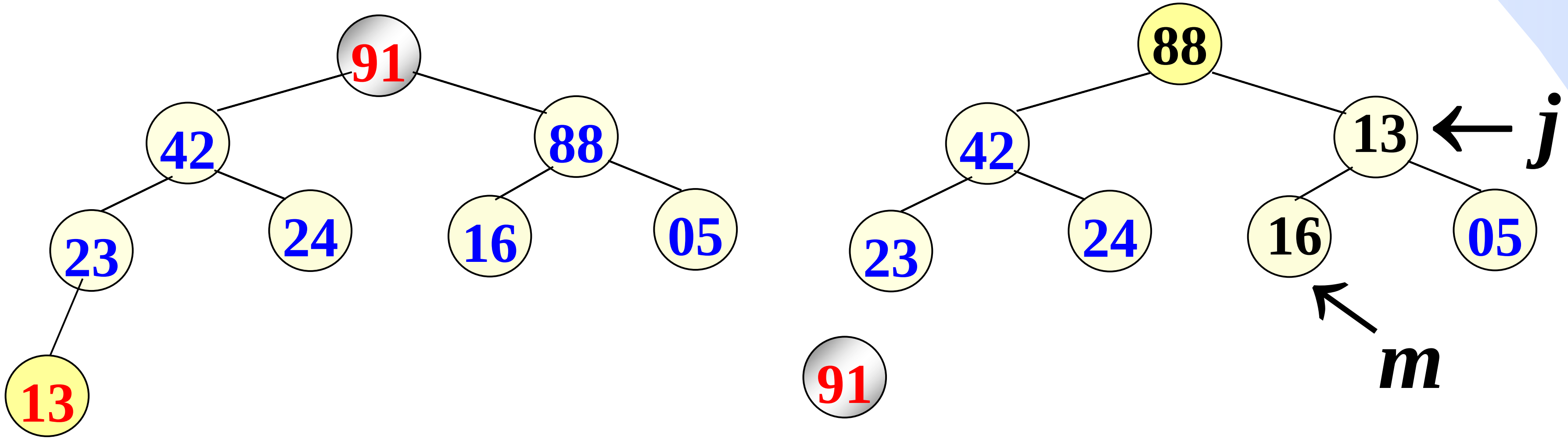
由于交换前为堆结构，则$R[1]$与$R[i]$交换后，只有$R[1]$与其左右孩子不满足堆的性质。

（1）令j指向堆根结点（$j \leftarrow$根结点在数组R中的下标）。

（2）将$R[j]$与其关键词最大的孩子$R[m]$比较；

（3）若$R[j] \geq R[m]$，则已满足堆序性，算法结束；

（4）若$R[m] > R[j]$，则交换$R[j]$与$R[m]$，$j \leftarrow m$，返回（2）



重建堆

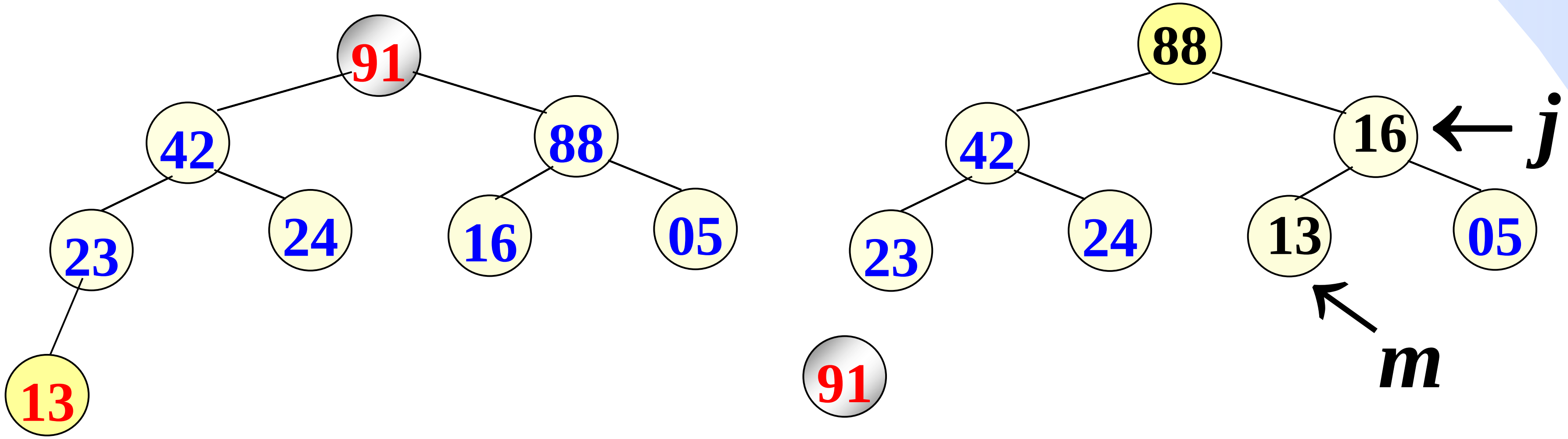
由于交换前为堆结构，则$R[1]$与$R[i]$交换后，只有$R[1]$与其左右孩子不满足堆的性质。

（1）令j指向堆根结点（j←根结点在数组R中的下标）。

（2）将$R[j]$与其关键词最大的孩子$R[m]$比较；

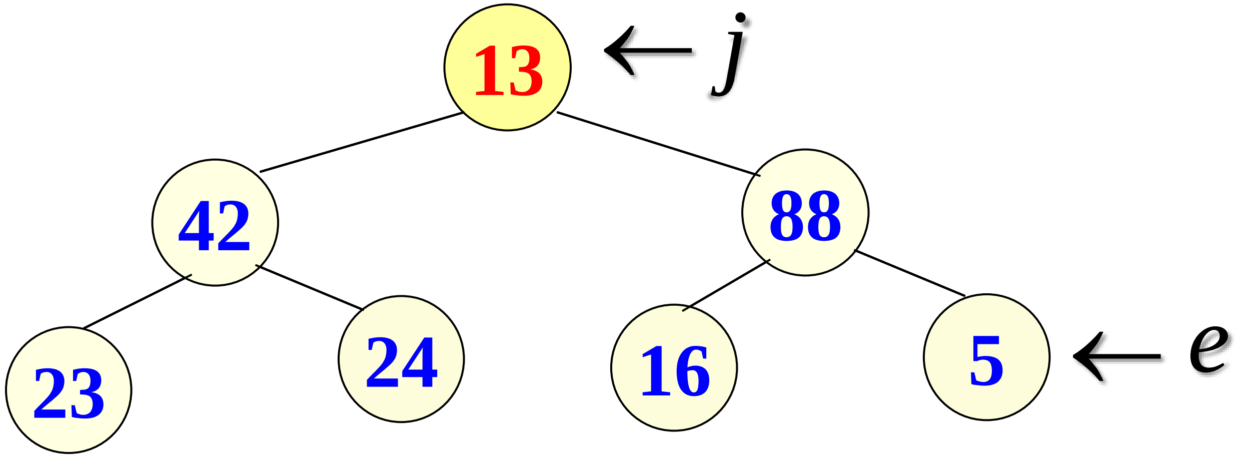
（3）若$R[j] \geq R[m]$，则已满足堆序性，算法结束；

（4）若$R[m] > R[j]$，则交换$R[j]$与$R[m]$，$j \leftarrow m$，返回（2）



重建堆

- j从根开始沿某一分支下行；
- j最多下行至最后一个非叶结点，其值最多为$e/2$，e为堆最后一个结点在数组R中的下标；

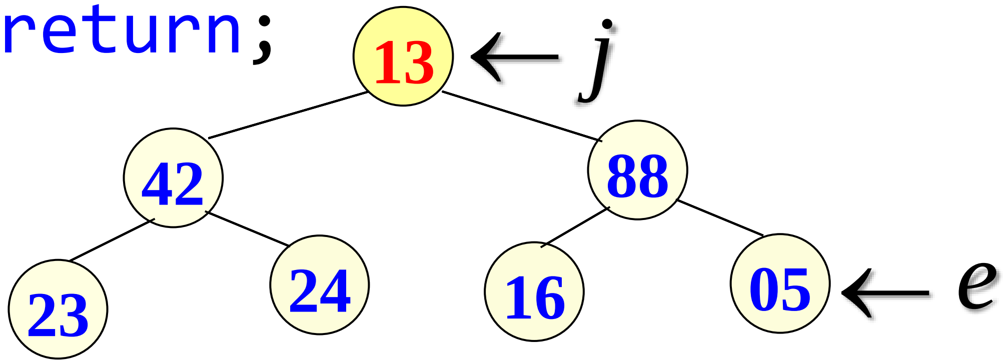


```
void Restore(int R[], int root, int e){
    //重建堆，root是堆的根结点在数组R中下标，e是堆最后一个元素在R中下标
    int m, j = root;   //初始化，j指向堆根
    while(j <= e/2){   //j最多下行至最后一个非叶结点
        if((2*j+1<=e) && (R[2*j]<R[2*j+1])) m=2*j+1;
        else m=2*j;    //R[m]为R[j]的最大孩子
        if(R[m]>R[j]){
            swap(R[m], R[j]); //交换R[m]和R[j]
            j = m;    //j继续下行
        }
        else return;
    }
}
```

若j有右孩子

j的右孩子比左孩子大

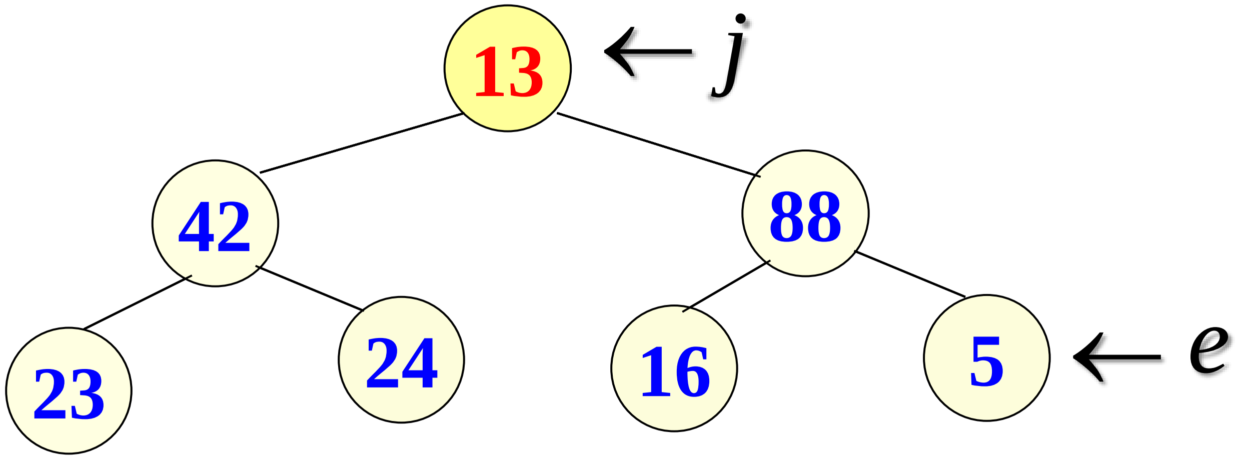
```
void swap(int &m,int &n){
    int temp = m;
    m = n;
    n = temp;
}
```



时间复杂度 O(logn)

重建堆

- j从根开始沿某一分支下行；
- 每个非叶结点对应2次关键词比较；
- 关键词比较次数取决于堆的高度，实际是高度×2；
- 时间复杂度O(log$n$)。



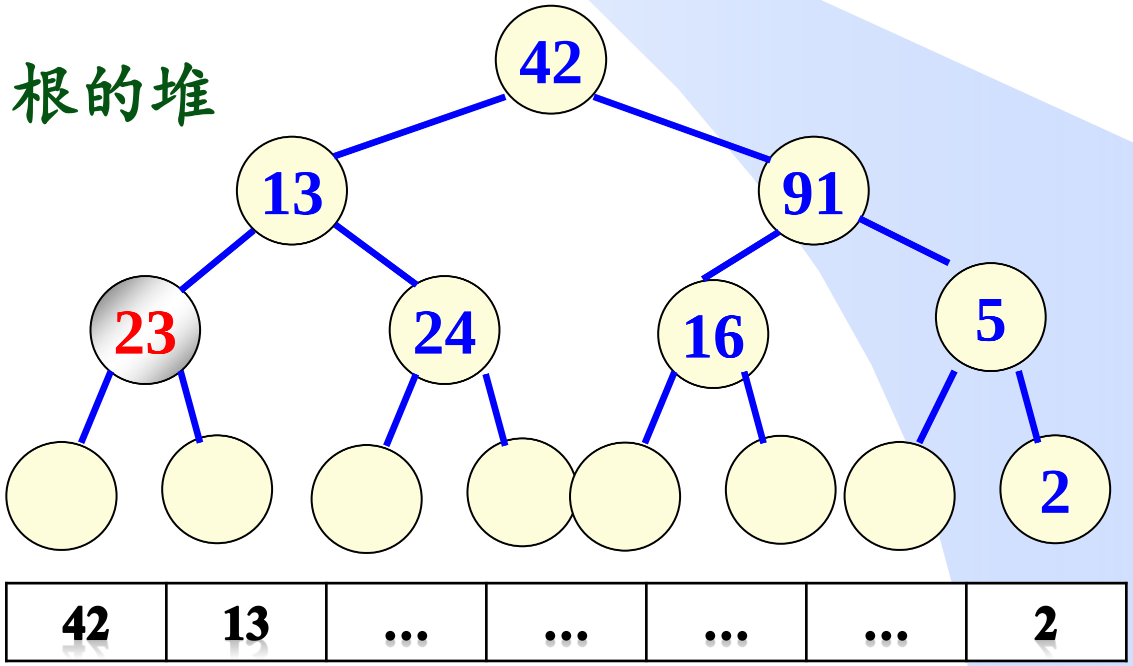
回顾：具有n个结点的完全二叉树的高度是$\lfloor \log_2 n \rfloor$.

初始建堆

- **能否调用重建堆算法？** 重建堆只是根结点变了，其余结点还是堆，还满足堆性质。但现在所有结点可能都不满足堆性质。
- **方案：** 依次把以$\lfloor n/2 \rfloor, \lfloor n/2 \rfloor-1, \ldots, 1$为根的子树重建为堆。

```
void BuildHeap(int R[],int n){
   for(int i=n/2; i>=1; i--)
        Restore(R, i, n);//重建以i为根的堆
}
```

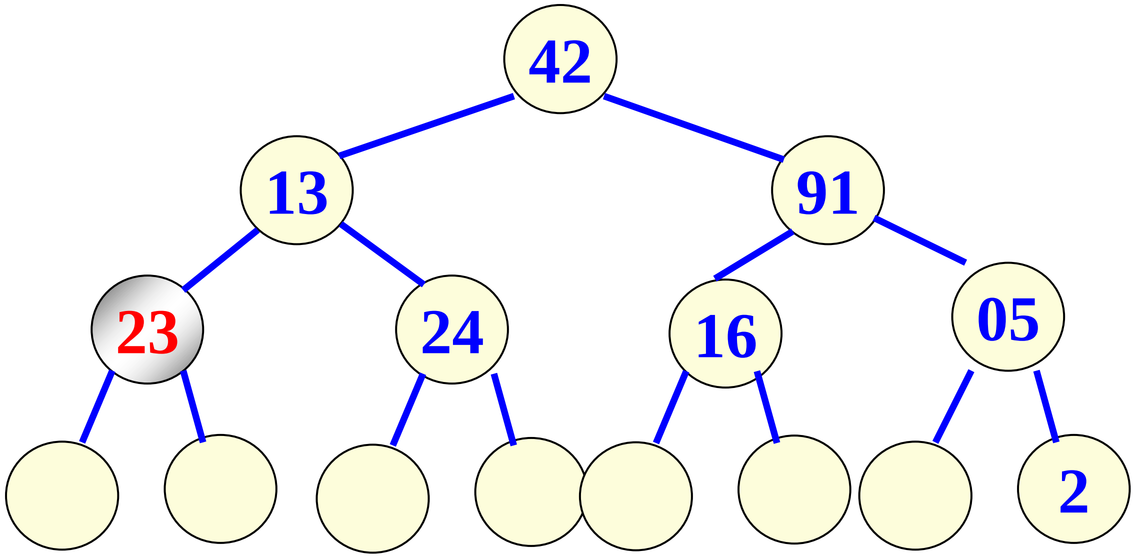
Floyd建堆算法
自底向上重建每棵子树



时间复杂度O($n$log$n$)?

初始建堆算法时间复杂度

- 重建以R[i]为根的堆，关键词比较次数取决于R[i]的高度（最多是以R[i]为根的子树的高度×2）。
- 初始建堆是重建了以每个非叶结点为根的堆，总时间取决于各结点的高度之和（实际是每个非叶结点）。



初始建堆算法时间复杂度

$$T = \frac{n}{4} + 2 \cdot \frac{n}{8} + 3 \cdot \frac{n}{16} + 4 \cdot \frac{n}{32} + \cdots$$

$$2T = \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + 4 \cdot \frac{n}{16} + \cdots$$

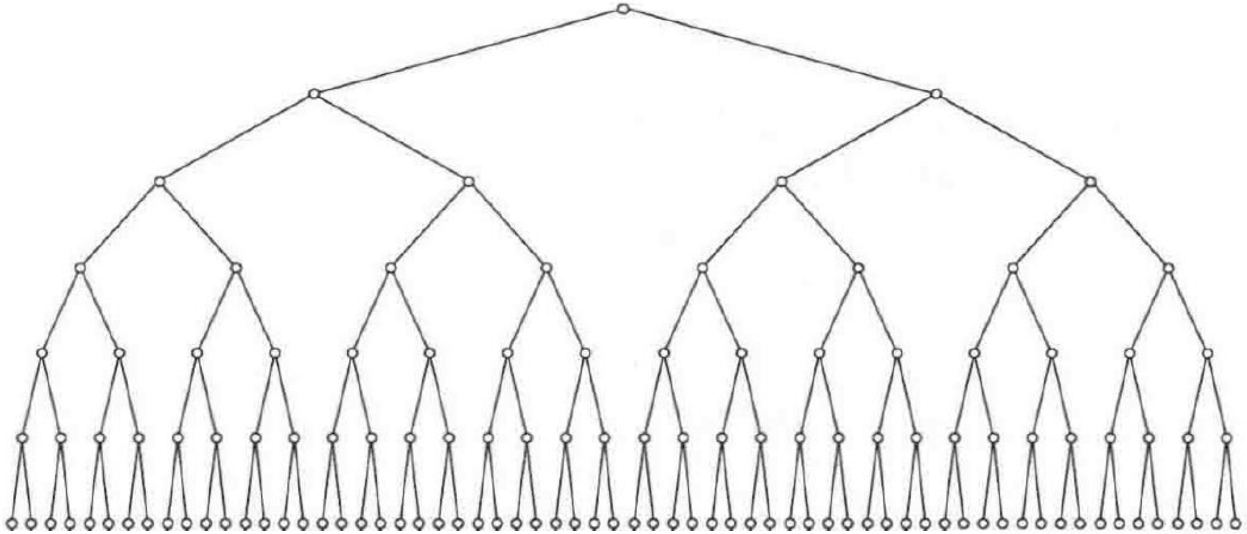
$$2T - T = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \frac{n}{32} + \cdots$$

$$T = n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots)$$

$$= n\left(\frac{\frac{1}{2}\left(1 - \left(\frac{1}{2}\right)^k\right)}{1 - \frac{1}{2}}\right) = n\left(1 - \left(\frac{1}{2}\right)^k\right) < n$$

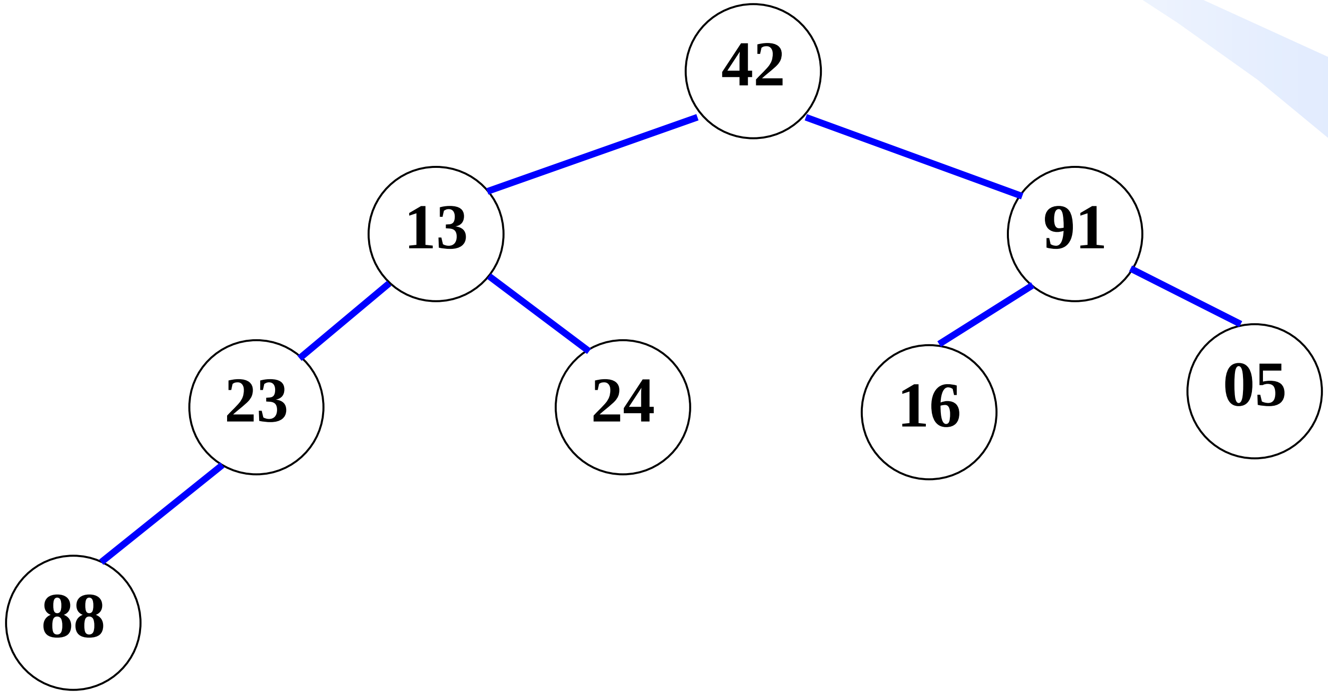
各结点的高度之和小于n

初始建堆算法的时间复杂度O(n)

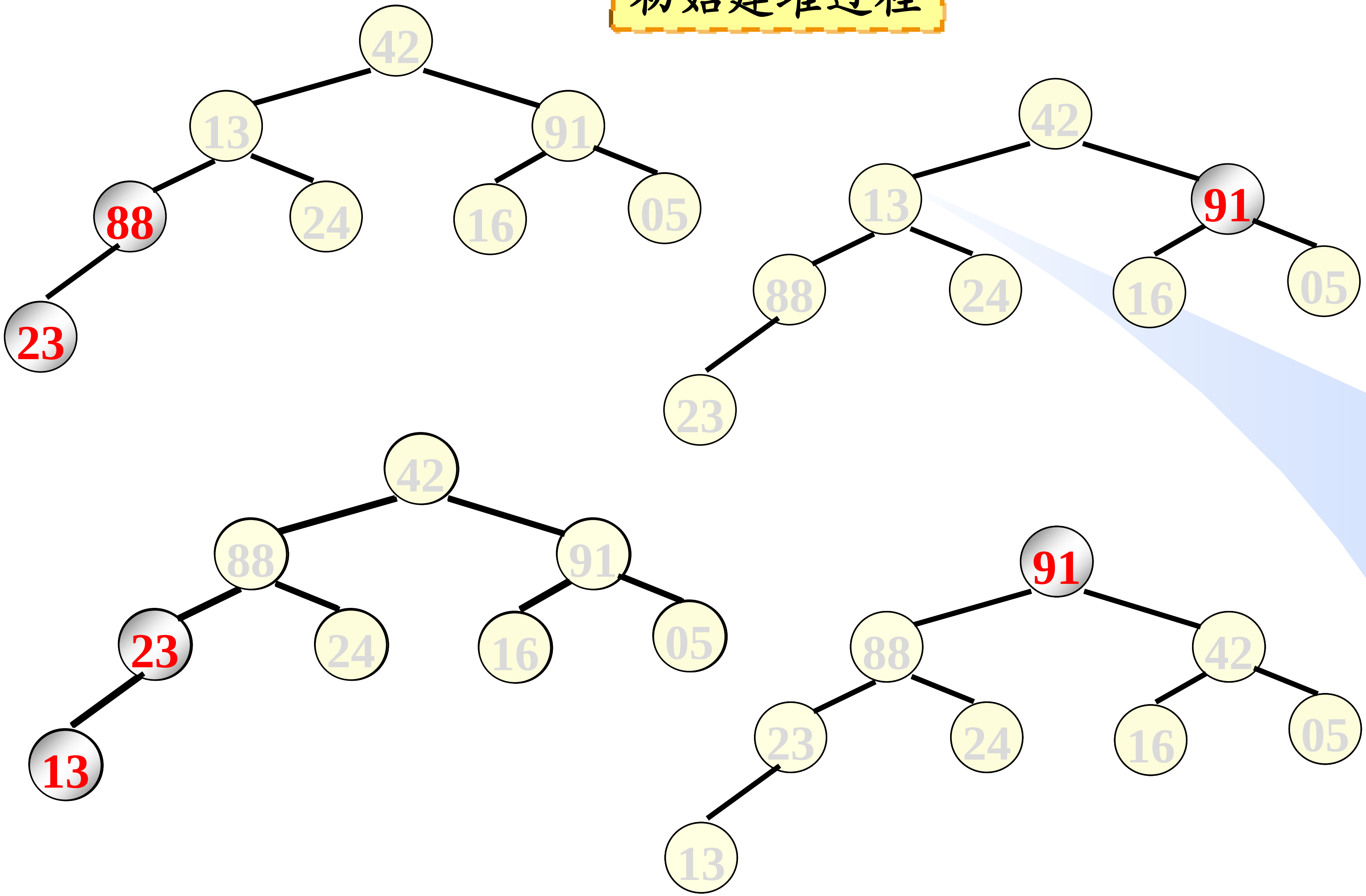


初始建堆例子

关键词序列(42, 13, 91, 23, 24, 16, 05, 88)。



初始建堆过程



(91, 88, 42, 23, 24, 16, 05, 13)

课下练习

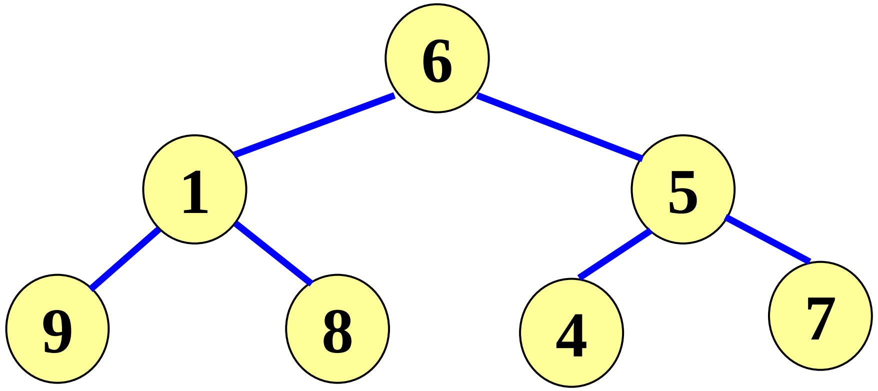
将数据序列 (6, 1, 5, 9, 8, 4, 7)建成大根堆，序列变化过程为______。【2018年考研题全国卷】

[A] 6 1 7 9 8 4 5 → 6 9 7 1 8 4 5 → 9 6 7 1 8 4 5 → 9 8 7 1 6 4 5

[B] 6 9 5 1 8 4 7 → 6 9 7 1 8 4 5 → 9 6 7 1 8 4 5 → 9 8 7 1 6 4 5

[C] 6 9 5 1 8 4 7 → 9 6 5 1 8 4 7 → 9 6 7 1 8 4 5 → 9 8 7 1 6 4 5

[D] 6 1 7 9 8 4 5 → 7 1 6 9 8 4 5 → 7 9 6 1 8 4 5 → 9 7 6 1 8 4 5

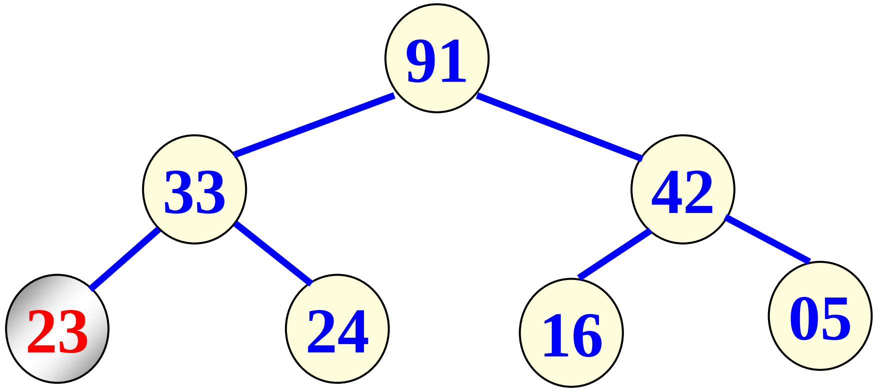


堆排序算法

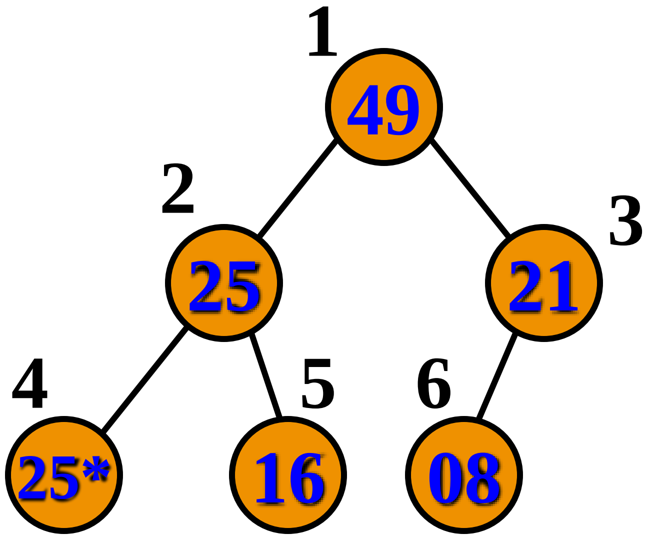
```
void HeapSort(int R[],int n){       //堆排序算法
    for(int i=n/2; i>=1; i--)       //初始建堆
        Restore(R, i, n);
    for(int i=n; i>=2; i--){
        swap(R[1], R[i]);           //R[1]和R[i]交换
        Restore(R, 1, i-1);         //重建R[1]…R[i-1]的堆
    }
}
```

时间复杂度

O(*nlogn*)

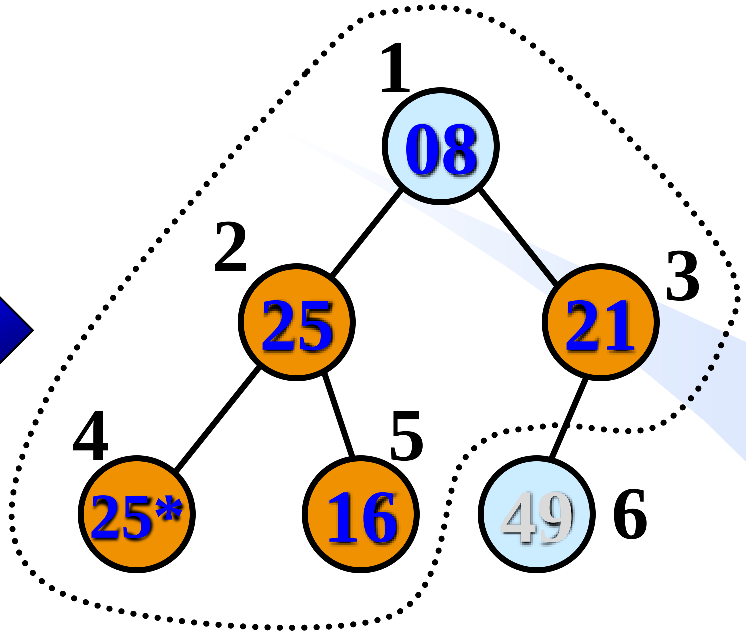


堆排序示例



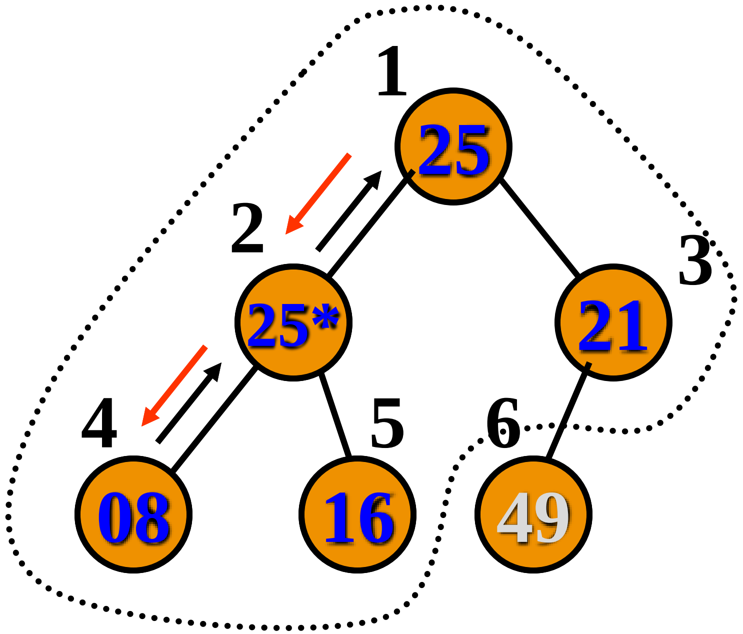
49	25	21	25*	16	08

初始堆



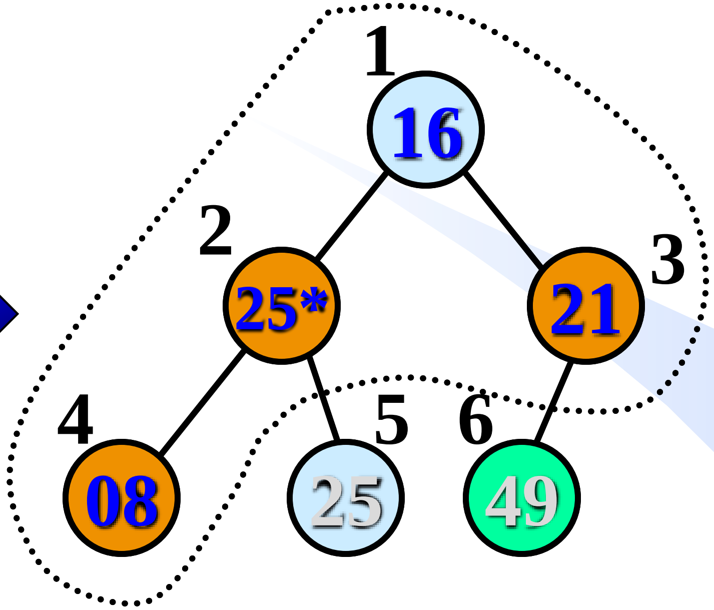
08	25	21	25*	16	49

交换 1 号与 6 号元素，
6 号元素就位



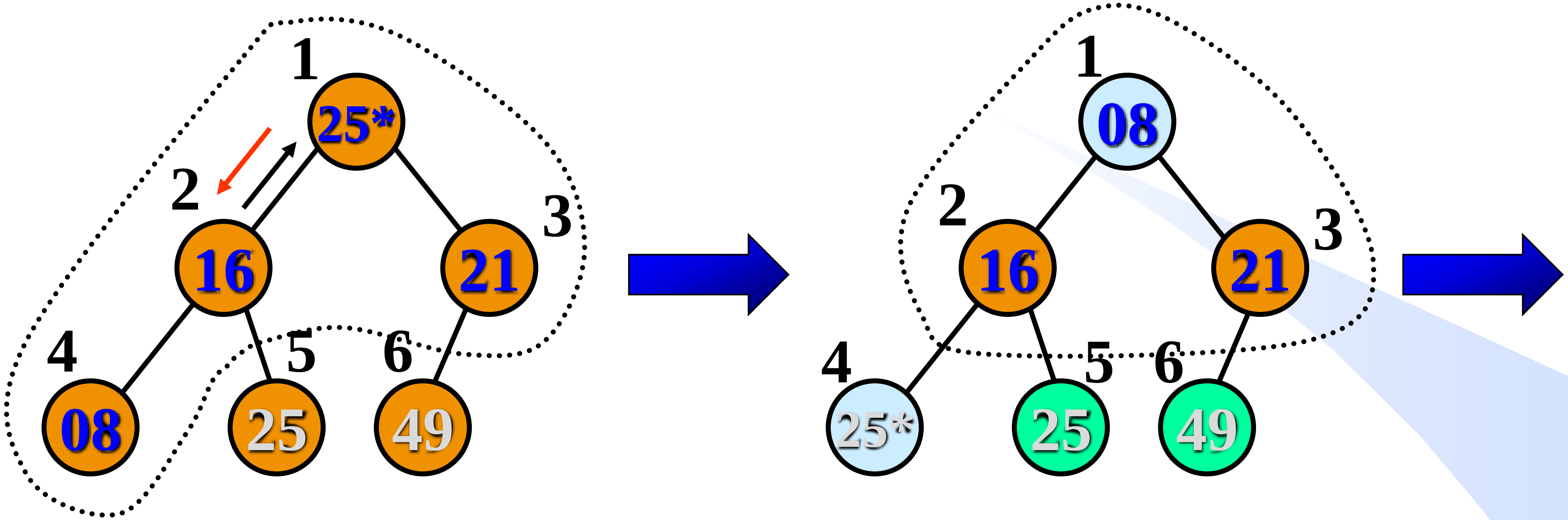
25	25*	21	08	16	49

从 1 号到 5 号 重新
调整为大根堆



16	25*	21	08	25	49

交换 1 号与 5 号元素，
5 号元素就位

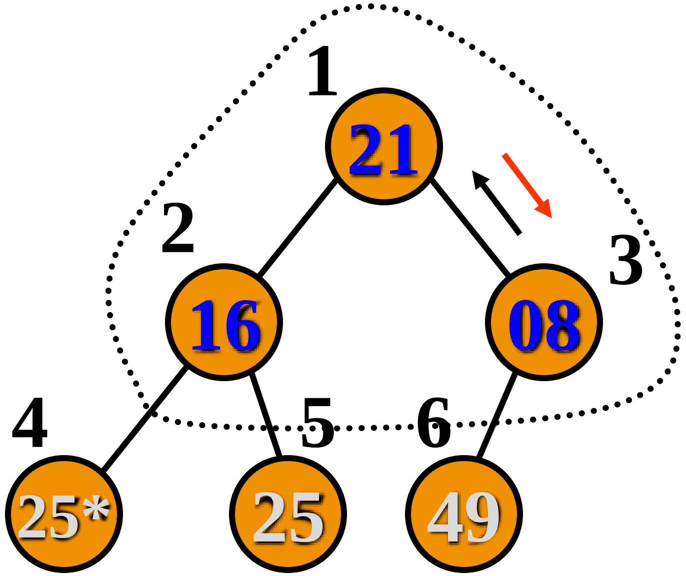


25*	16	21	08	25	49

从 1 号到 4号 重新调整为大根堆

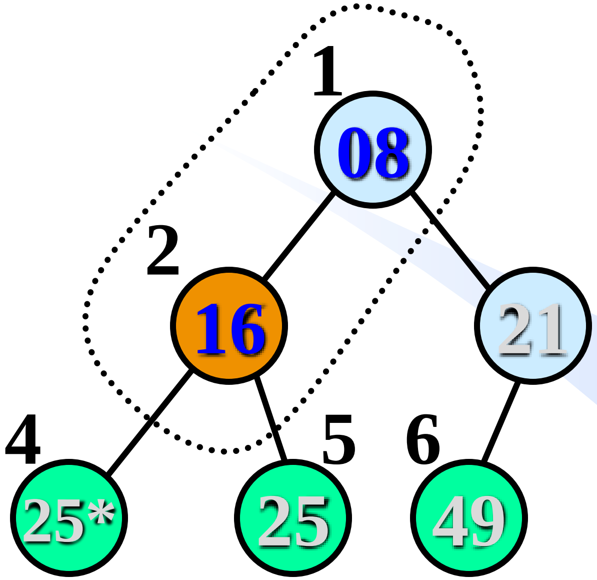
08	16	21	25*	25	49

交换 1 号与 4 号元素，4号元素就位



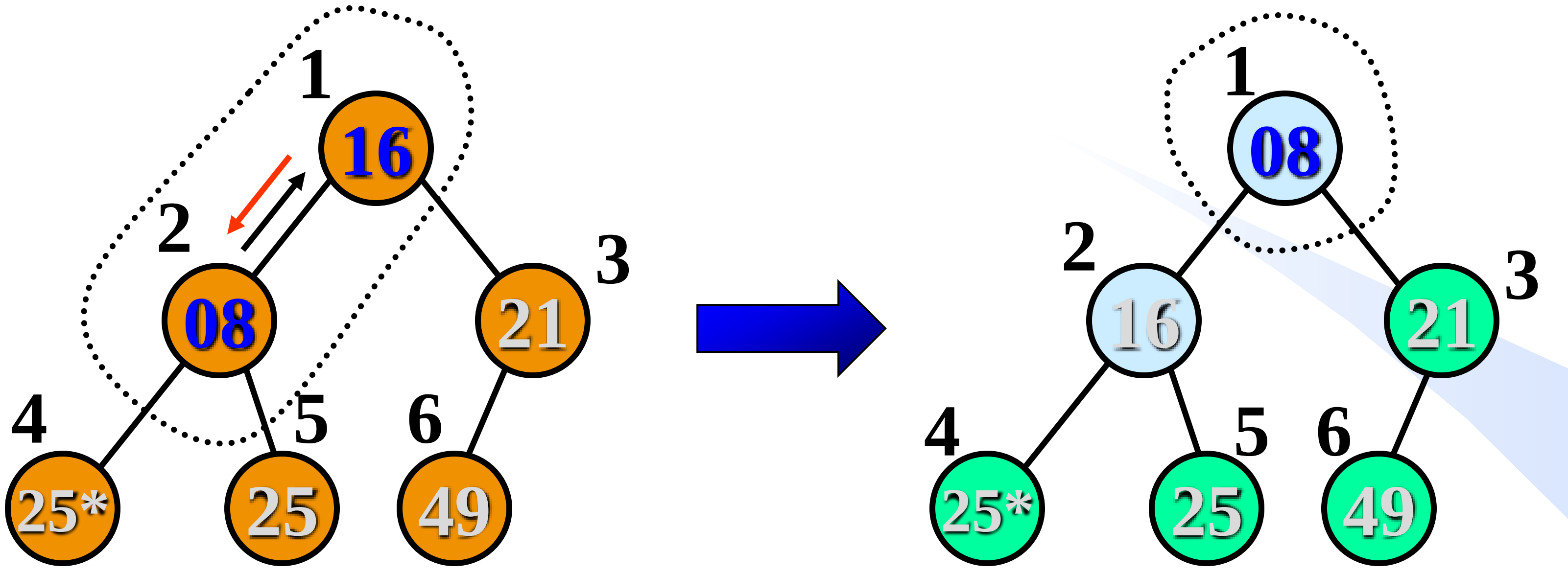
21	16	08	25*	25	49

从 1 号到 3 号 重新调整为大根堆



08	16	21	25*	25	49

交换 1 号与 3号元素，3号元素就位



16	08	21	25*	25	49

08	16	21	25*	25	49

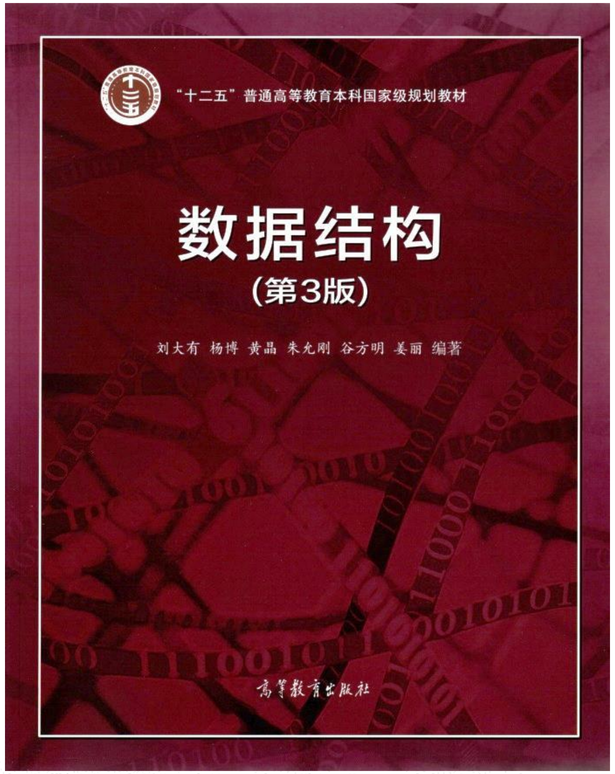
从 1 号到 2 号 重新
调整为大根堆

交换 1 号与 2 号元素，
2 号元素就位

堆排序算法

- 时间复杂度：O(*n*log*n*)（最好、最坏和平均）
- 空间复杂度： O(1)
- 稳定性：堆排序是不稳定的排序方法。

think.create.solve



堆排序

- 锦标赛排序
- 堆排序算法
- **堆与优先级队列**
- 可合并堆
- Top K问题

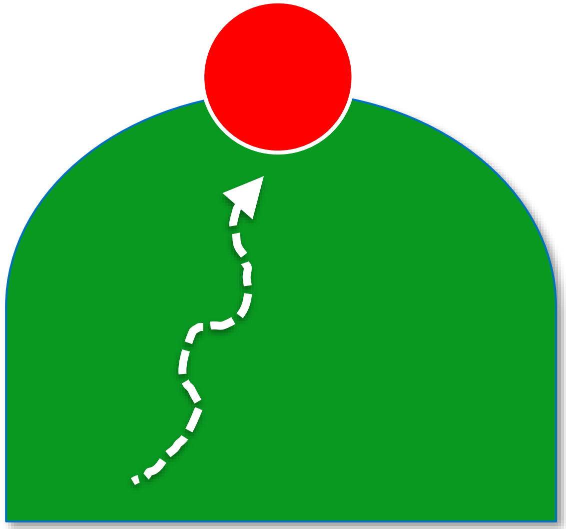
数据之法
结构之美
算法之道

Last updated on 2022.11

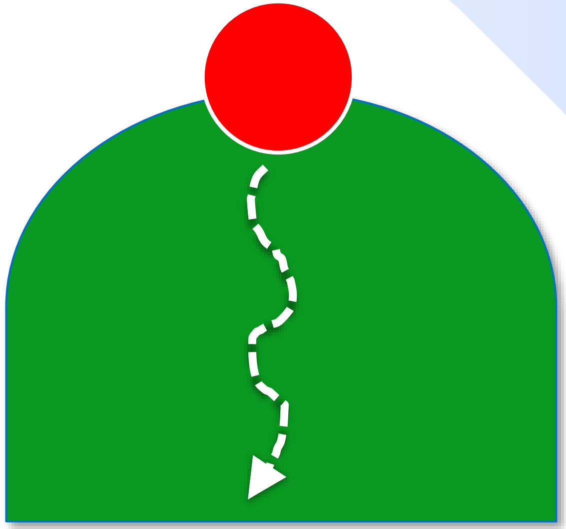
zhuyungang@jlu.edu.cn

堆的两个基本操作

上浮操作

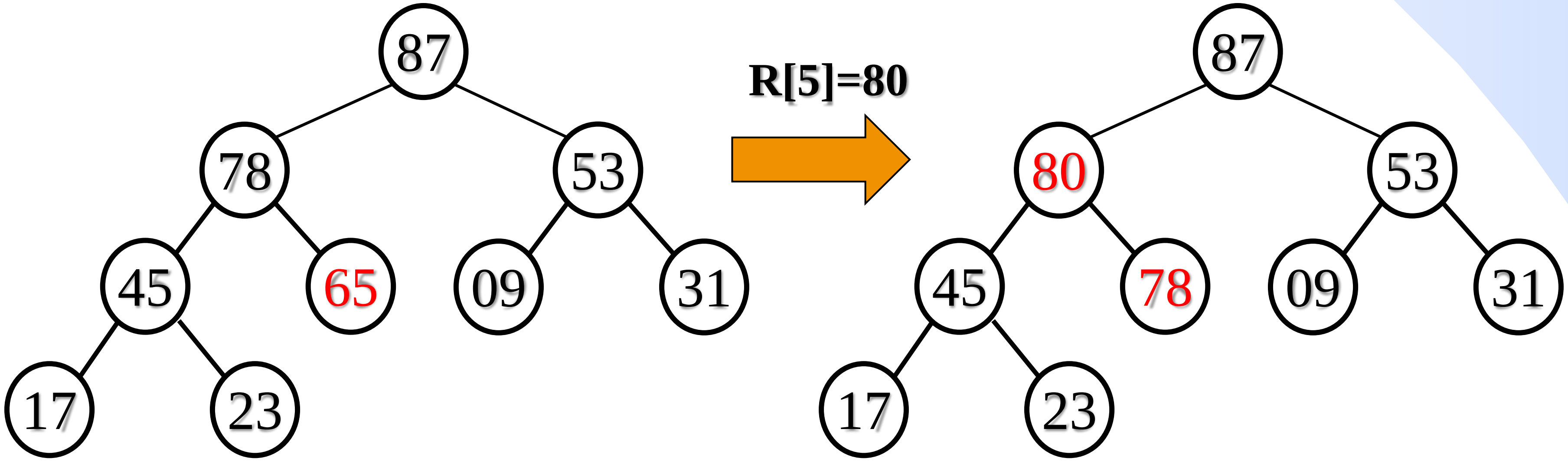


下沉操作



上浮操作

当大根堆的元素值R[*i*]变大时，该结点可能会上浮；



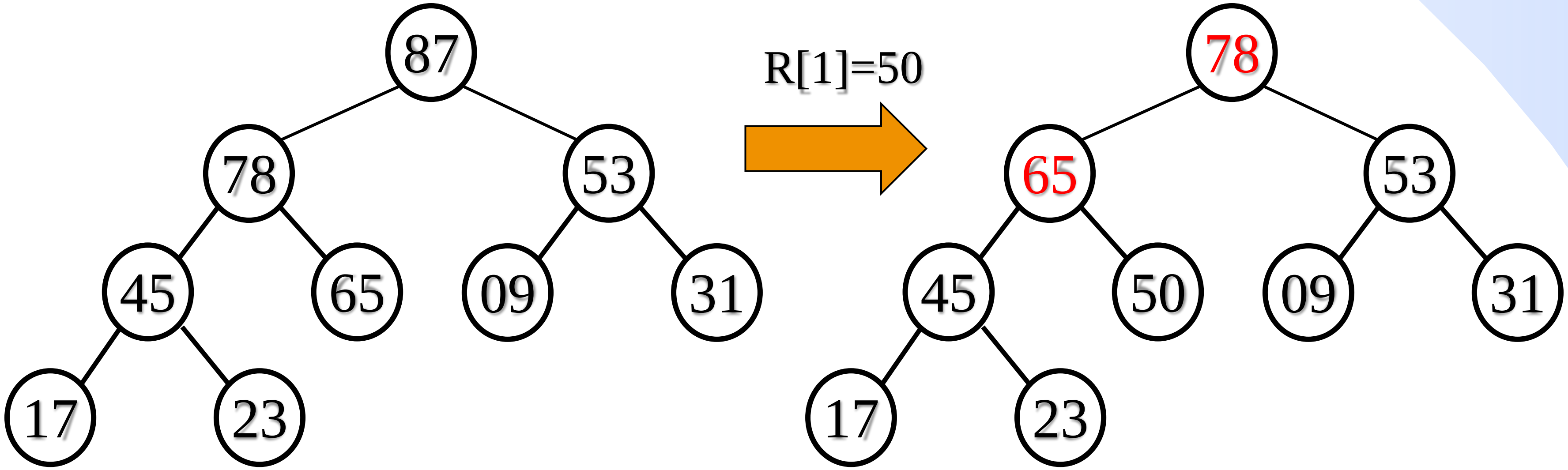
上浮操作—实现

```
void MoveUp(int R[], int n, int i){ //R[i]上浮
    while(i>1 && R[i]>R[i/2]){   //R[i]比父亲大
        swap(R[i], R[i/2]);      //交换R[i]和父亲
        i/=2;                    //i继续上行
    }
}
```

时间复杂度O(log*n*)

下沉操作

当大根堆的元素值R[*i*]变小时，该结点可能会下沉；



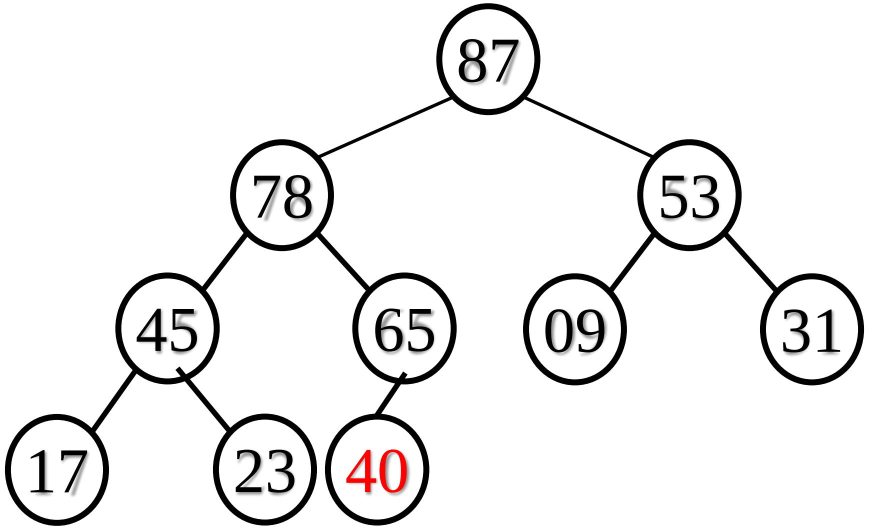
下沉操作—实现

```
void MoveDown(int R[], int n, int i) { //R[i]下沉
    Restore(R, i, n);   //重建以i为根的堆
}
```

时间复杂度O(log*n*)

堆的插入操作

- 插入一个元素，把该元素放在最后，再做MoveUp操作。
- $(R_1, R_2, \ldots, R_n)$是一个堆，把R_{n+1}添加到堆$(R_1, R_2, \ldots, R_n)$中，并使$(R_1, R_2, \ldots, R_n, R_{n+1})$成为一个堆。



插入操作—实现

```
void insert(int R[], int &n, int x){ //堆尾插入值x
    R[++n] = x;          //x放在R[n+1]处，堆元素个数加1
    MoveUp(R, n, n);  //元素R[n]上浮
}
```

时间复杂度O(log*n*)

另一种建堆算法

- 从空堆开始依次插入n个元素， 时间O($n\log n$)
- **Floyd建堆算法：** 时间O(n)

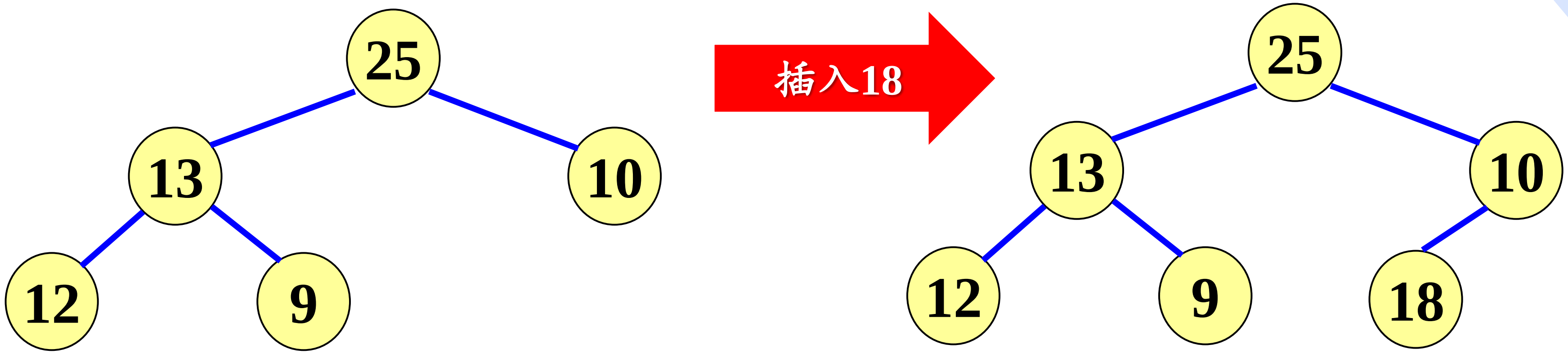
判断题：将n个元素建成一个堆，至少需要O($n\log n$)时间。

【清华大学考研题】

课下练习

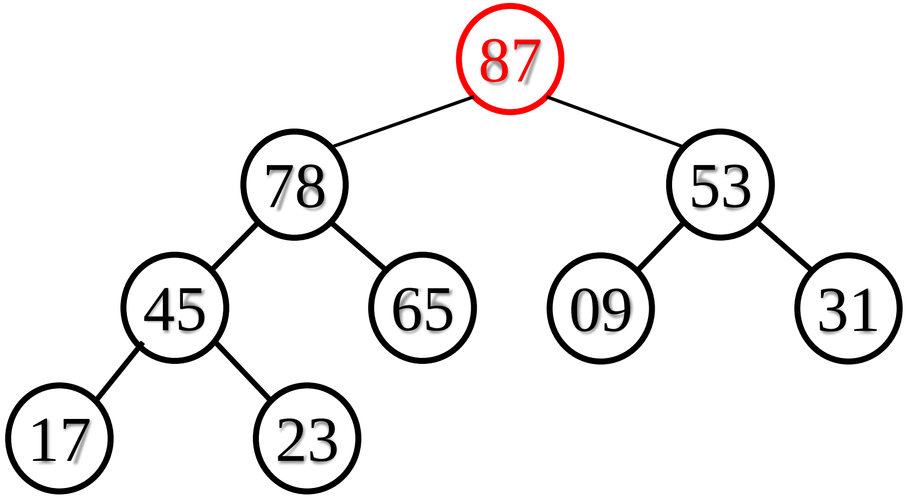
已知序列(25, 13, 10, 12, 9)是大根堆，在此序列尾部插入新元素18，将其再调整为大根堆，调整过程中元素比较次数为______。【考研题全国卷】

[A] 1　　[B] 2　　[C] 4　　[D] 5



堆的删除操作

- $(R_1, R_2, \ldots, R_n)$是一个堆，从堆中删除堆顶（根结点），使删除后的文件仍然是堆。
- 把R[*n*]移到根结点处，堆元素个数减一，再对根结点*R*[1]做下沉操作。



删除操作—实现

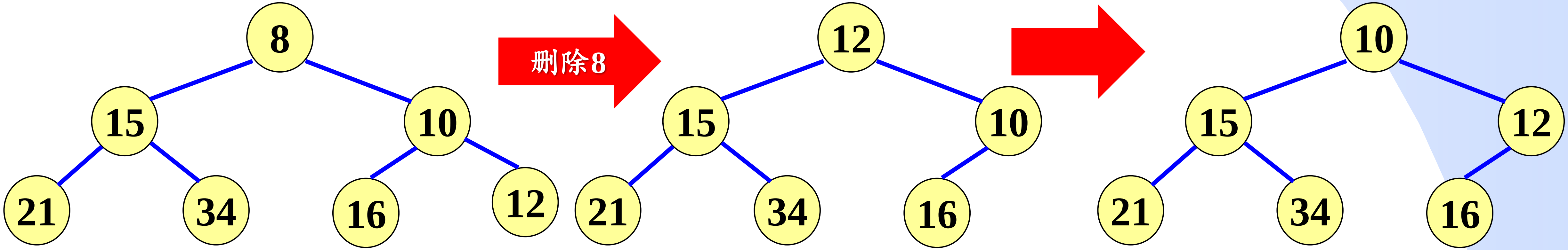
```
int DelMax(int R[], int &n){  //删除并返回堆顶
    int max=R[1];        //以最大堆为例，堆顶即最大元素
    R[1] = R[n];         //堆尾移至堆顶
    n--;                 //删除后堆元素个数减1
    MoveDown(R,n,1);     //新堆顶元素下沉
    return max;
}
```

时间复杂度O(log*n*)

课下练习

已知小根堆(8, 15, 10, 21, 34, 16, 12)，删除关键字8后需要重建堆，在此过程中，关键词比较次数为______。【考研题全国卷】

[A] 1　　[B] 2　　[C] 3　　[D] 4



优先级队列（Priority Queue）

- 优先级高的元素先出队，优先级低的元素后出队
- 用堆实现优先级队列，优先级↔关键词大小
- 入队：堆的插入（堆尾插入），时间O(log*n*)
- 出队：堆的删除（删除并返回堆顶），时间O(log*n*)

优先级队列（最小堆）优化哈夫曼算法

```
Priority_Queue MinPQ;       //基于最小堆创建优先级队列
... ...
for (int i=0; i<n-1; i++){
    Node* t = new Node;
    t->left = MinPQ.DeQueue(); //出队weight最小的结点
    t->right= MinPQ.DeQueue(); //出队weight最小的结点
    t->weight = t->left->weight + t->right->weight;
    MinPQ.EnQueue(t); //新创建的结点入队
}
```

队列中元素：哈夫曼树的结点指针
关键词：结点的weight域

用堆选取weight最小的结点

时间复杂度$O(n\log n)$

优先级队列（最小堆）优化Dijkstra算法

队列中元素：二元组(*v*, *dist*)
关键词：*dist*
用堆选取*dist*最小的顶点

```
void Dijkstra(Vertex *Head,int n,int s,int
  int S[N], i, j, min, v, w;
  for(i=1; i<=n; i++) {path[i]=-1; dist[i]=INF; S[i]=0;}//初始化
  dist[s]=0; Priority_Queue MinPQ;MinPQ.EnQueue(Node(s,dist[s]));
  while(!MinPQ.Empty()){
    Node q=MinPQ.DeQueue(); v=q.v;   //选D值最小的顶点v
    if(S[v]==1) continue;
    S[v]=1;                   //将顶点v放入S集合
    for(Edge* p=Head[v].adjacent;p;p=p->link) {
      w=p->VerAdj;            //更新v的邻接顶点的D值
      if(S[w]==0 && dist[v]+p->cost<dist[w]) {
        dist[w]=dist[v]+p->cost;  path[w]=v;
        MinPQ.EnQueue(Node(w,dist[w]));
      }
    }
  }
}
```

```
struct Node{
  int v, dist;
  Node(int a,int b){
    v=a; dist=b;
  }
};
```

时间复杂度O((*n*+*e*)log*n*)

Dijkstra
基于优先级队列的BFS

优先级队列（最小堆）优化的相关算法

	时间复杂度	稀疏图	稠密图
原始Dijkstra	$O(n^2+e)$	$\approx O(n^2)$	$\approx O(n^2)$
堆优化Dijkstra	$O((n+e)\log n)$	$\approx O(n\log n)$	$\approx O(n^2\log n)$

同理，若邻接表存图，**Prim+堆**时间复杂度为$O((n+e)\log n)$．

用堆选取最小跨边，堆中元素$(v, Lowcost[v])$

不加堆优化适合稠密图，堆优化适合稀疏图。

图的遍历

PFS：**Priority First Search (Dijkstra、Prim)**
（基于优先级的遍历）

DFS：**Depth First Search**

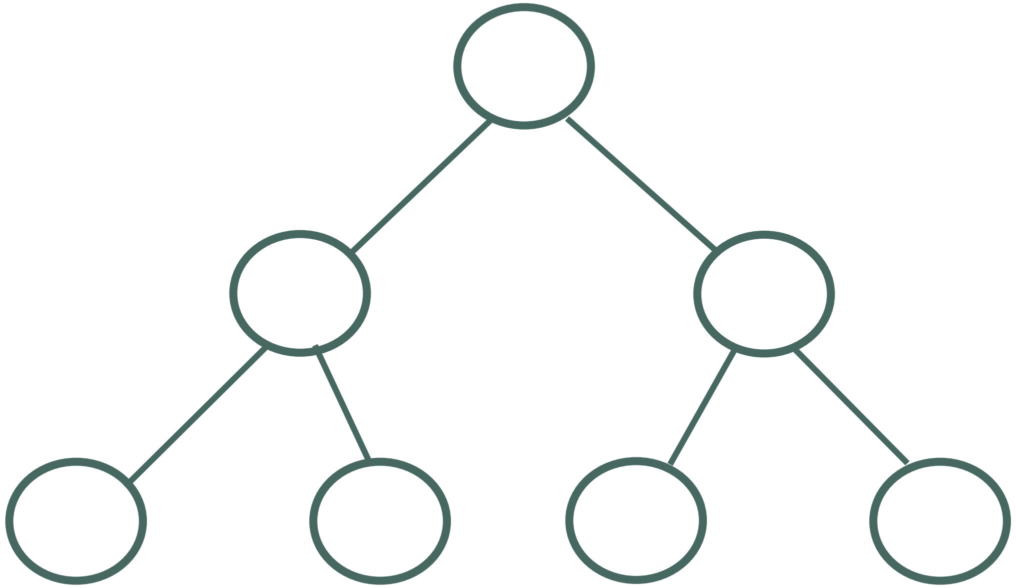
BFS：**Breath First Search**

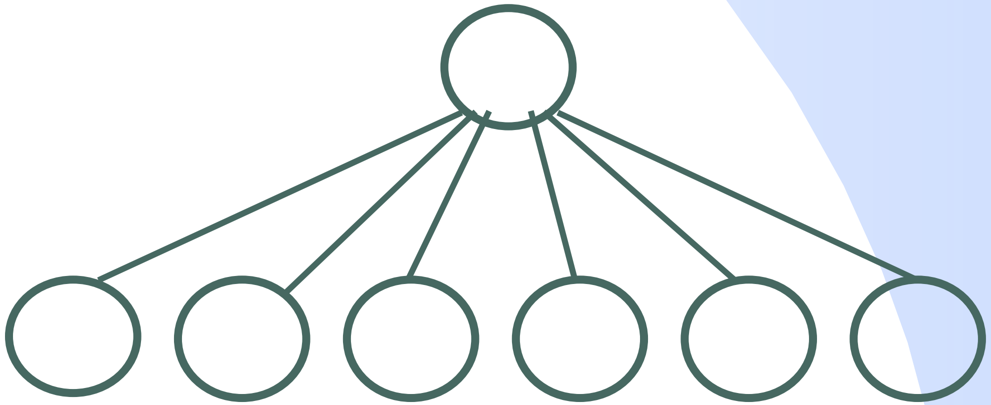
d堆（d-heap）

➢ 堆为什么用完全二叉树结构？

✓ 堆的各种操作时间复杂度都取决于树高，所以我们希望在同等结点数的情况下，树高尽可能矮

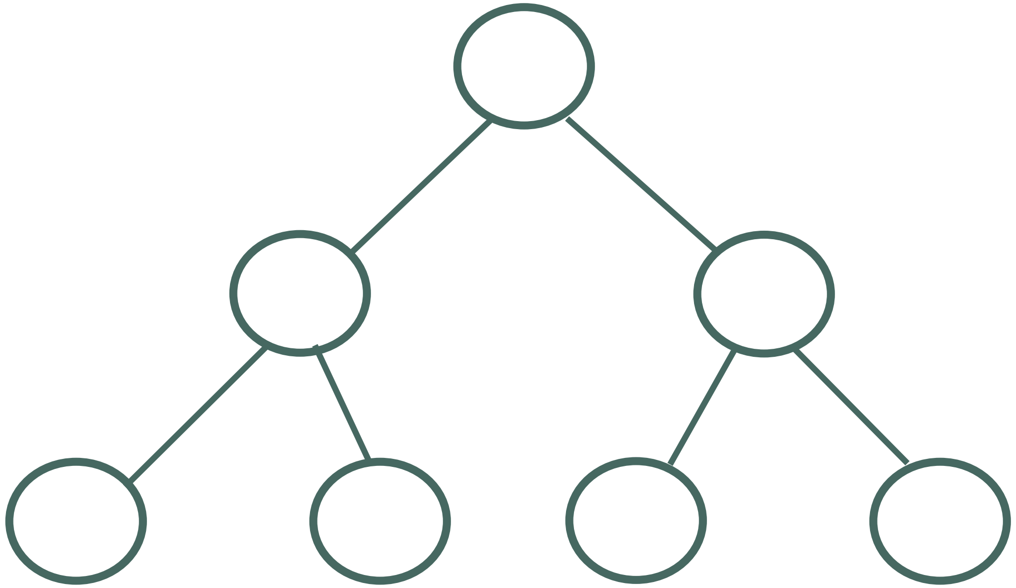
➢ 将二叉树改成d 叉树（$d>2$），树高降至$O(\log_d n)$

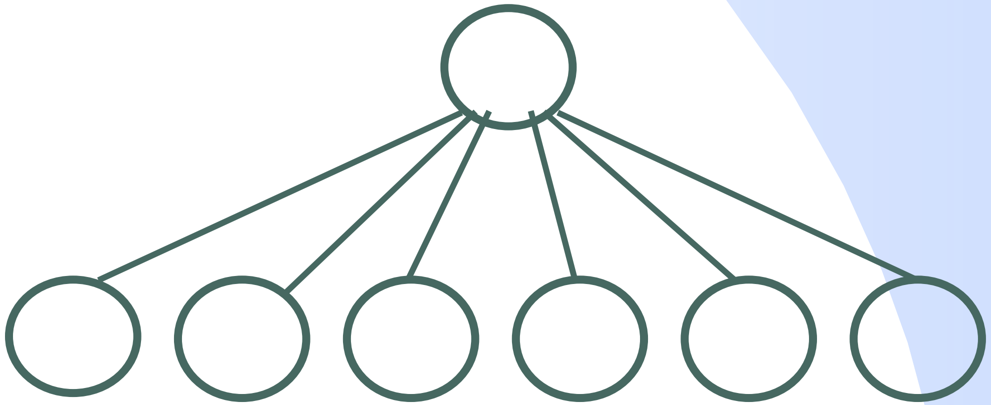




d堆（d-heap）

- 将二叉树改成d 叉树（d >2），树高降至$O(\log_d n)$
- 插入算法（依赖于上浮操作）时间降至$O(\log_d n)$
- 删除算法（依赖于下沉操作/重建堆）时间升至$O(d \cdot \log_d n)$





优先级队列（d堆）优化Dijkstra算法

```
void Dijkstra(Vertex *Head,int n,int s,int dist[],int path[]){
   int S[N], i, j, min, v, w;
   for(i=1; i<=n; i++) {path[i]=-1; dist[i]=INF; S[i]=0;}//初始化
   dist[s]=0; Priority_Queue MinPQ;MinPQ.EnQueue(Node(s,dist[s]));
   while(!MinPQ.Empty()){
     Node q=MinPQ.DeQueue(); v=q.v;    //出队-删除 O(d·log_d n)
     if(S[v]==1) continue;
     S[v]=1;                           //将顶点v放入S集合
     for(Edge* p=Head[v].adjacent;p;p=p->link) {
        w=p->VerAdj;                   //更新v的邻接顶点的D值
        if(S[w]==0 && dist[v]+p->cost<dist[w]) {
           dist[w]=dist[v]+p->cost;  path[w]=v;
           MinPQ.EnQueue(Node(w,dist[w])); //入队-插入O(log_d n)
        }
     }
   }
}
```

```
struct Node{
  int v, dist;
  Node(int a,int b){
    v=a; dist=b;
  }
};
```

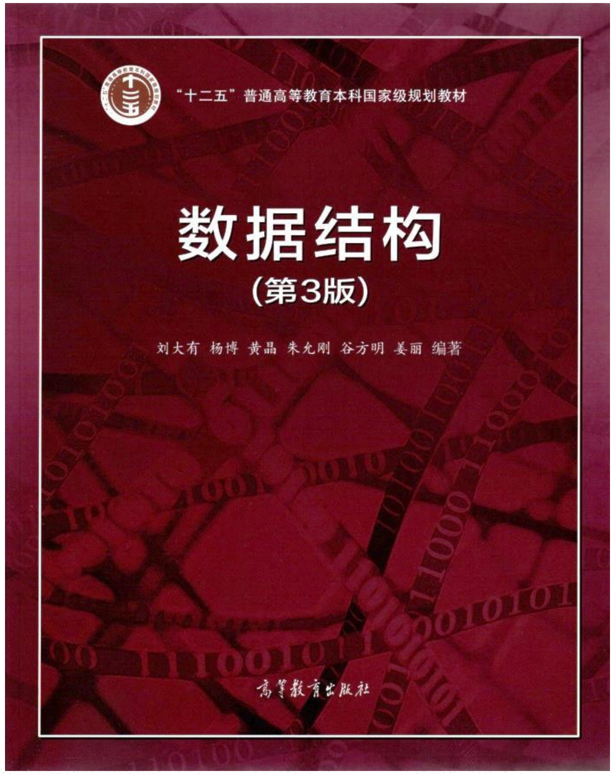
时间复杂度$O((n \cdot d+e)\log_d n)$

d堆（d-heap）

- 时间复杂度$O((n \cdot d+e)\log_d n)$
- 取$d \approx e/n+2$，则$O(e \cdot \log_{(e/n+2)} n)$
- 对稀疏图：$e \cdot \log_{(e/n+2)} n \approx n \cdot \log_{(1+2)} n \approx O(n\log n)$
- 对稠密图：$e \cdot \log_{(e/n+2)} n \approx n^2 \cdot \log_{(n+2)} n \approx O(n^2)$

	时间复杂度	稀疏图	稠密图
原始Dijkstra	$O(n^2+e)$	$\approx O(n^2)$	$\approx O(n^2)$
二叉堆	$O((n+e)\log n)$	$\approx O(n\log n)$	$\approx O(n^2\log n)$
d堆	$O((n \cdot d+e)\log_d n)$	$\approx O(n\log n)$	$\approx O(n^2)$

think.create.solve



堆排序

- 锦标赛排序
- 堆排序算法
- 堆与优先级队列
- **可合并堆**
- Top K问题

数据之法
结构之美
算法之道

Last updated on 2022.11

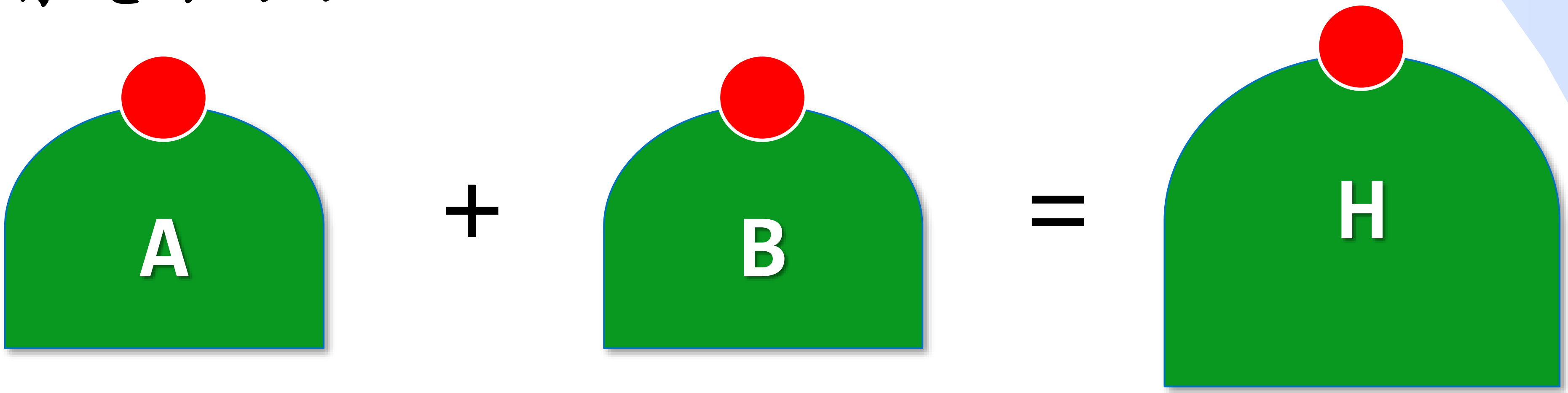
zhuyungang@jlu.edu.cn

可合并堆（*Mergeable Heap*）

- 讲“可合并堆”的高校（不完全统计）：清华大学、北京大学、上海交通大学、浙江大学、卡内基梅隆大学，斯坦福大学、普林斯顿大学、剑桥大学等

左式堆（*Leftist Heap*）

- H=merge(A,B)，将堆A和B合并成新堆
- ✓ 方案1：将A中元素逐个插入B，时间O(*n*log*n*)
- 方案2：将A和B合成一个大数组，然后对大数组进行初始建堆，时间O(*n*)
- 有没有更好的方法？



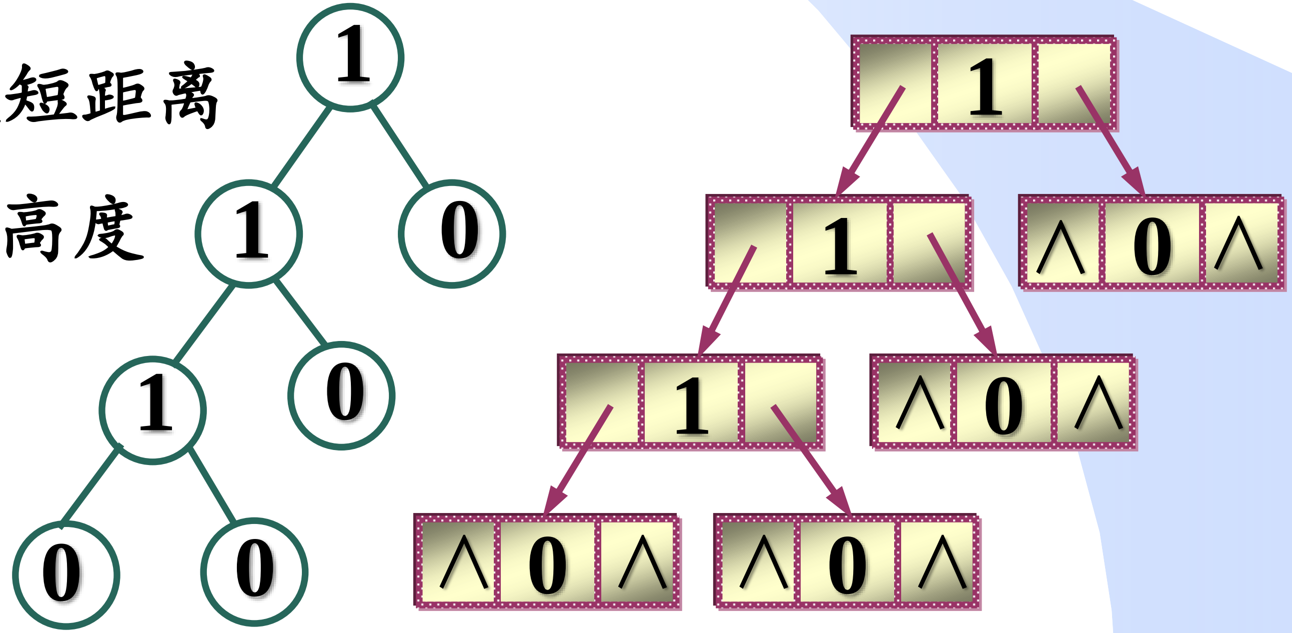
左式堆（*Leftist Heap*）

- ***npl*(*x*)**：结点*x*到***null***的最短距离，称为*x*的空路径长度（***Null Path Length***）.

$$npl(x)=\begin{cases}-1 & ,\ x = null \\ 1+\min\{npl(x\rightarrow left),\ npl(x\rightarrow right)\}, & \text{其 他}\end{cases}$$

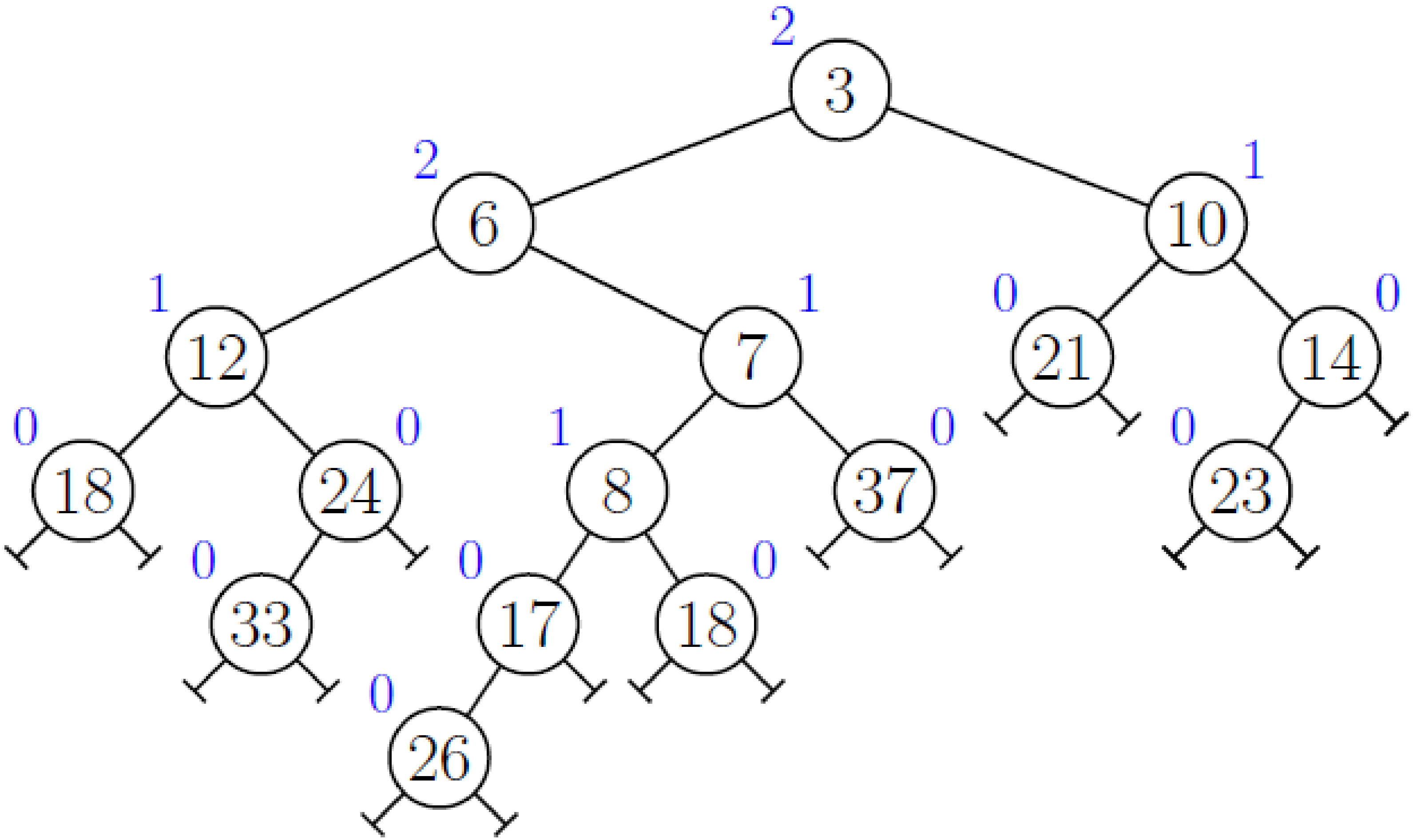
- ***npl*(*x*)**=*x*到空指针所在结点的最短距离

  =以*x*为根的最大满子树的高度



左式堆（*Leftist Heap*）

- ***npl*(*x*)**=以*x*为根的最大满子树的高度



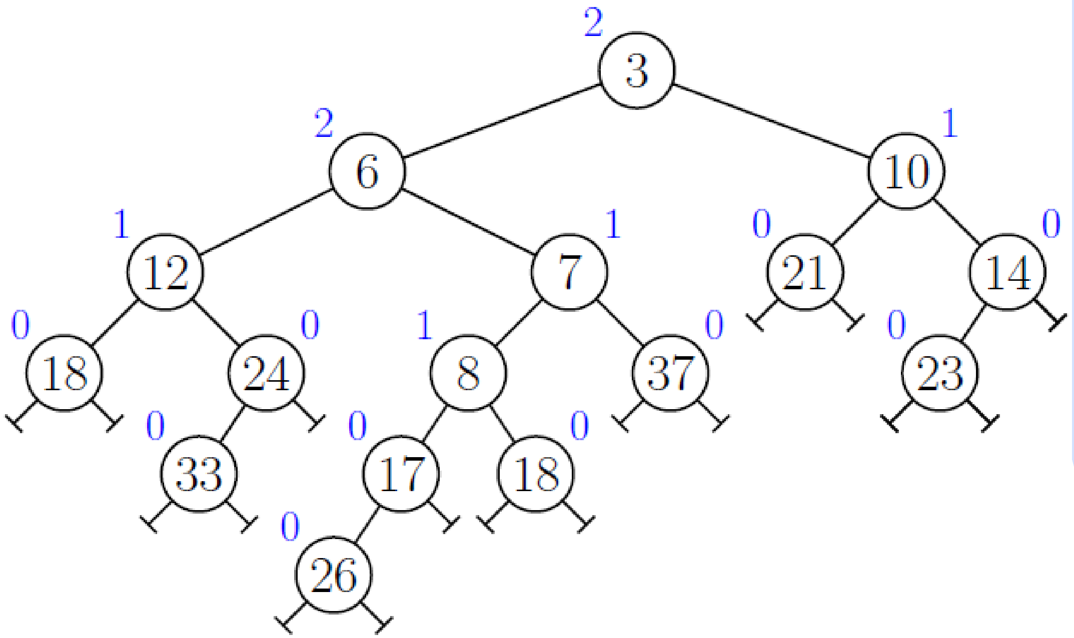
左式堆（*Leftist Heap*）

- 左式堆：对任意结点x，都有$npl(x \to left) \geq npl(x \to right)$.
- 推论：$npl(x) = 1 + \min\{npl(x \to left), npl(x \to right)\},$

$$= 1 + npl(x \to right),$$

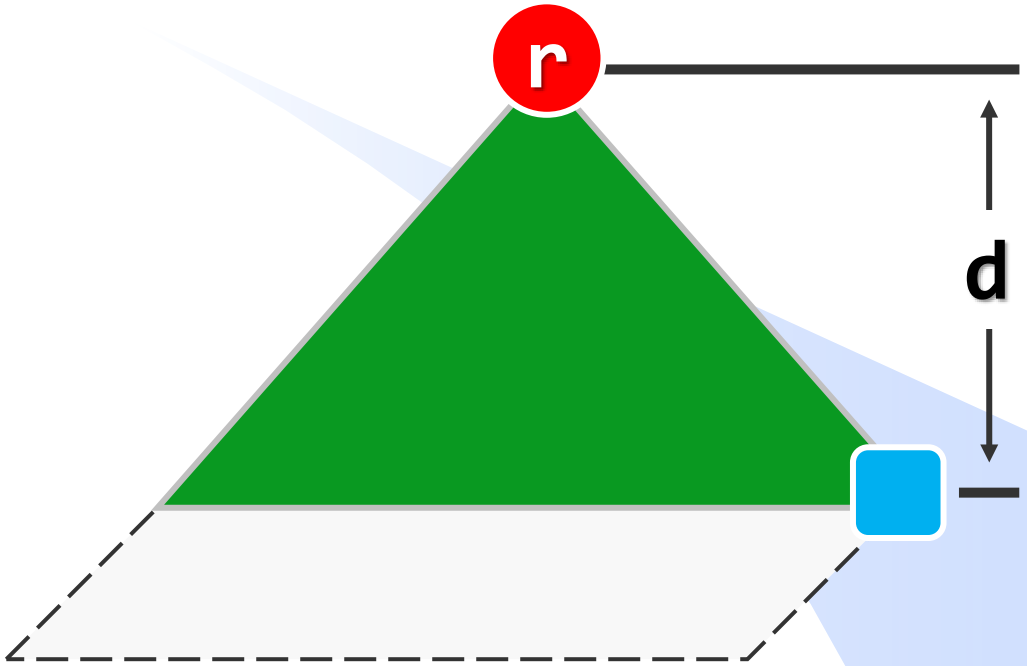
- 左式堆倾向于更**多**结点分布于**左**侧分支
- 左式堆不是完全二叉树，

  不使用数组存储



左式堆（Leftist Heap）

- 右侧链：从根结点r出发，一直沿右分支下行到没有右孩子的结点
- 右侧链的长度=$npl(r)=d$，则存在一棵以r为根、高度为d的满子树
- 在包含n个节点的左式堆中，右侧链的长度$d = O(\log n)$



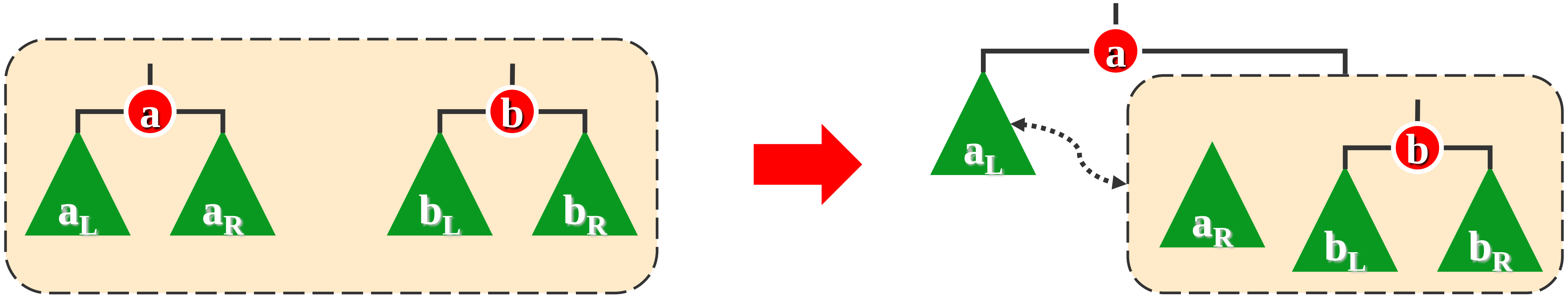
$$npl(r)= 1+ npl\ (r \rightarrow right)$$

左式堆——合并算法

```
struct node{
  int data;
  int npl;
  node* left;
  node* right;
};
```

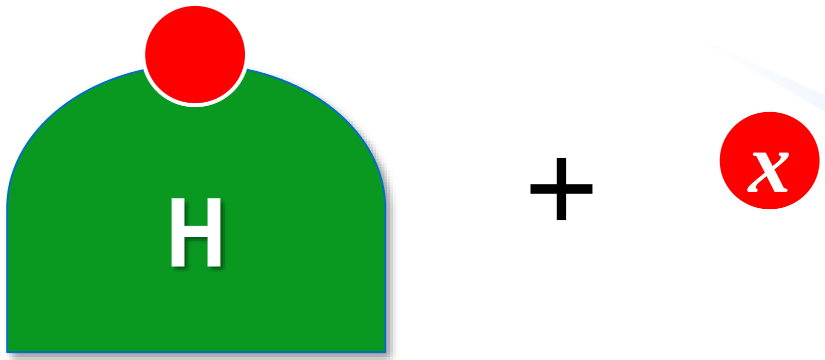
```
node* merge(node* a, node* b) {  //以最小堆为例
   if(a==NULL) return b; if(b==NULL) return a;
   if(a->data > b->data) swap(a,b); //确保a<=b
   a->right = merge(a->right,b);    //将a的右子树与b合并
   if(!a->left || a->left->npl < a->right->npl )
      swap(a->left, a->right); //确保a左孩子的npl>=右孩子
   if(a->right!=NULL) a->npl=1+a->right->npl; //更新a的npl
   else a->npl=0;
   return a;     //返回合并后的堆顶
}
```

时间复杂度O(log*n*)



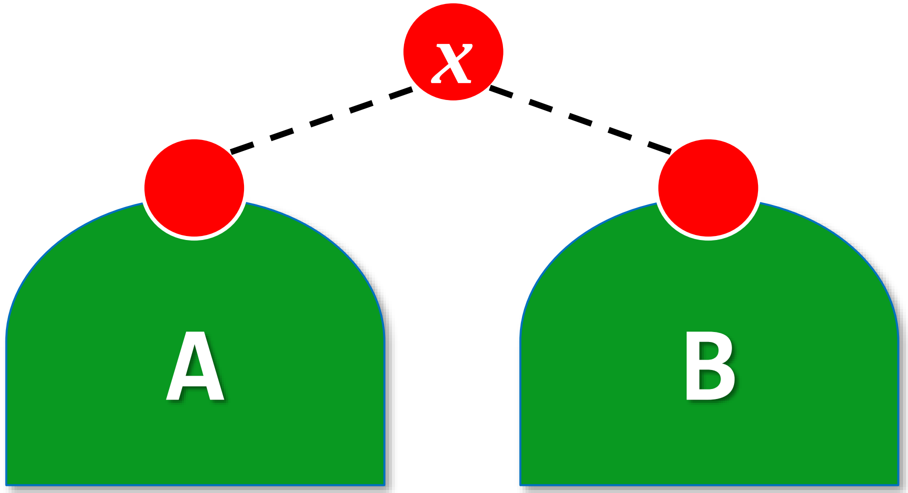
左式堆——插入和删除算法

- 插入节点x：将堆x与堆H合并



时间复杂度
$(\log n)$

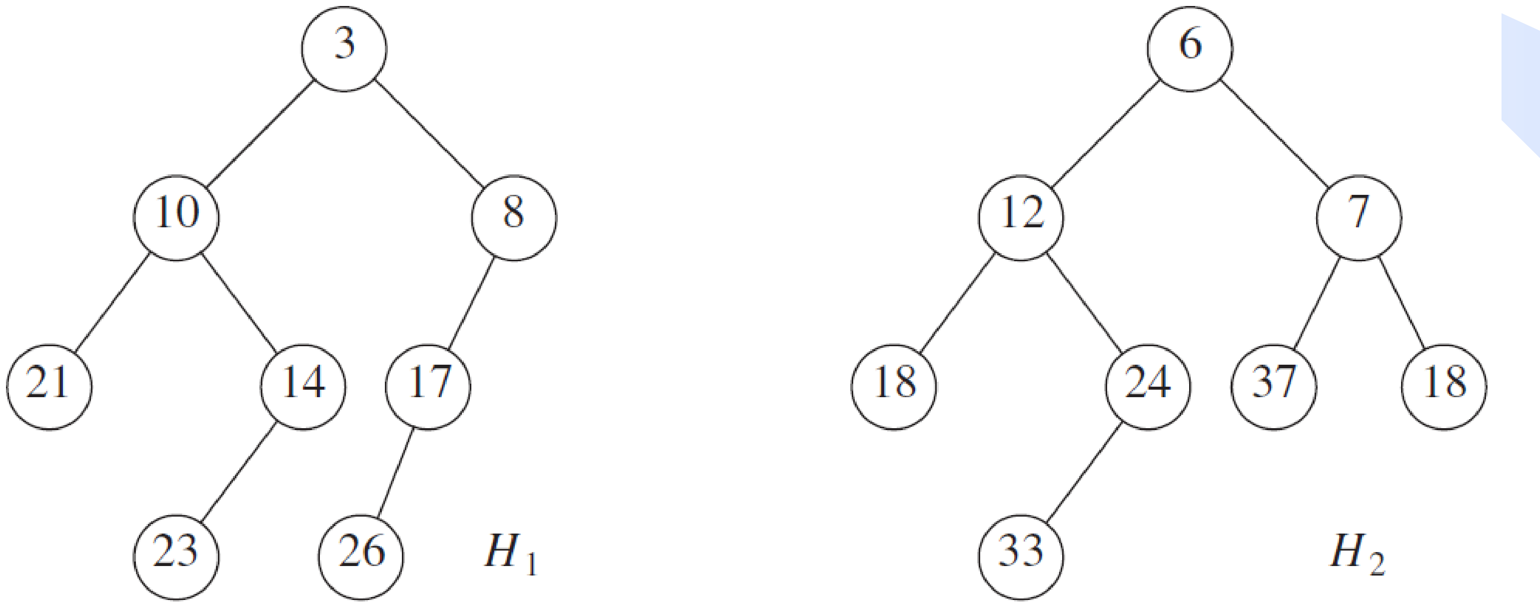
- 删除堆顶：将x取出，合并x的左右子树



时间复杂度
$(\log n)$

斜堆（*Skew Heap*）

- 满足堆序性的普通二叉树，不维护*npl* 值。



斜堆——合并算法

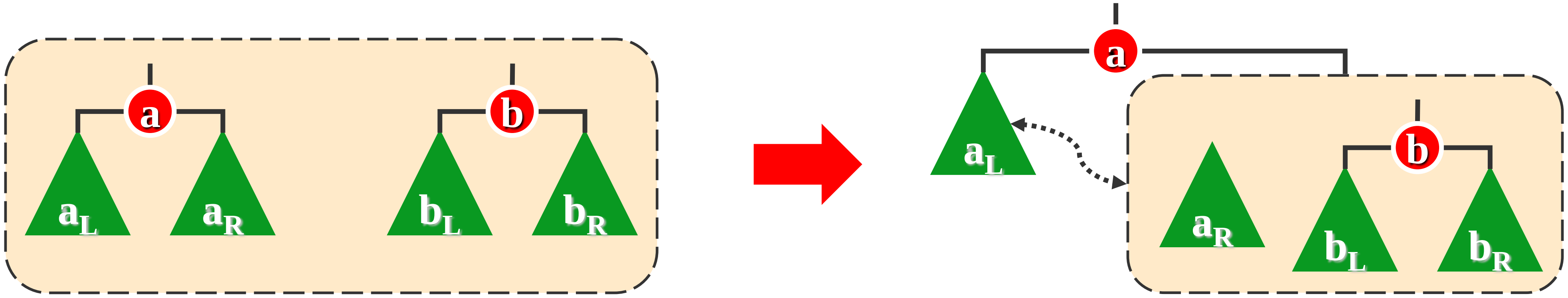
```
struct node{
   int data;
   node* left;
   node* right;
};
```

```
node* merge(node* a, node* b) { //以最小堆为例
   if(a==NULL) return b; if(b==NULL) return a;
   if(a->data > b->data) swap(a,b); //确保a<=b
   a->right = merge(a->right,b);    //将a的右子树与b合并

      swap(a->left, a->right);


   return a;    //返回合并后的堆顶
}
```

均摊时间复杂度
O(log$n$)

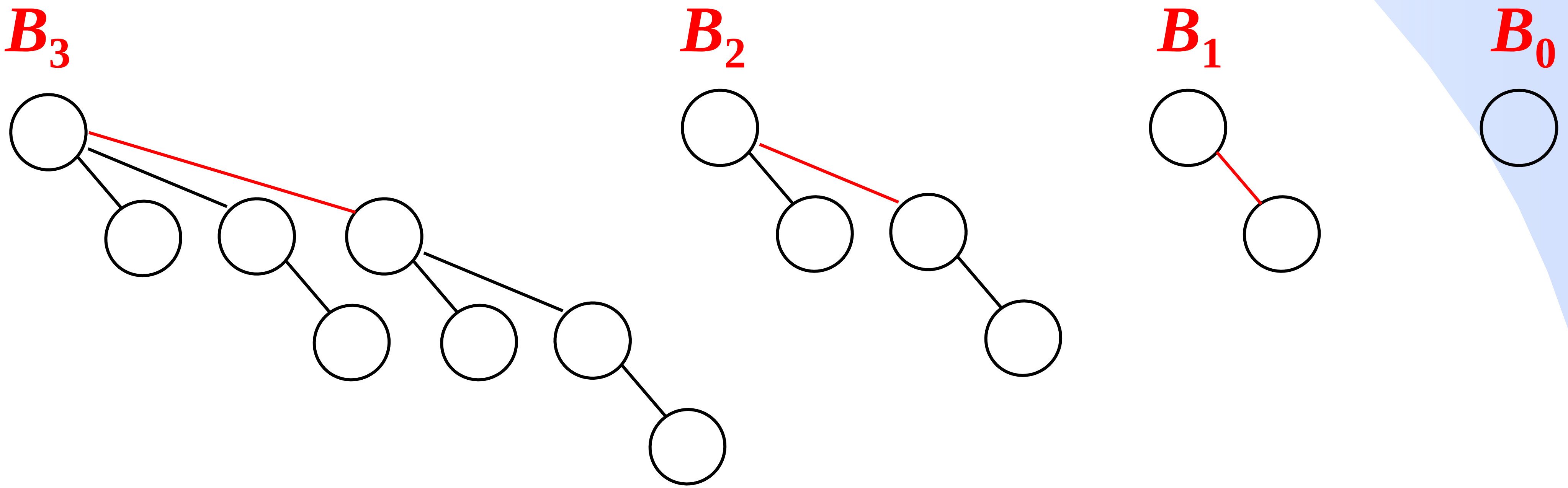


k阶二项树（Binomial Tree）

k阶二项树B_k递归定义：

✓ k=0：B_0只含根结点

✓ k>0：B_k由2棵B_{k-1}组成，其中一棵B_{k-1}的根作为另一棵B_{k-1}根的孩子。

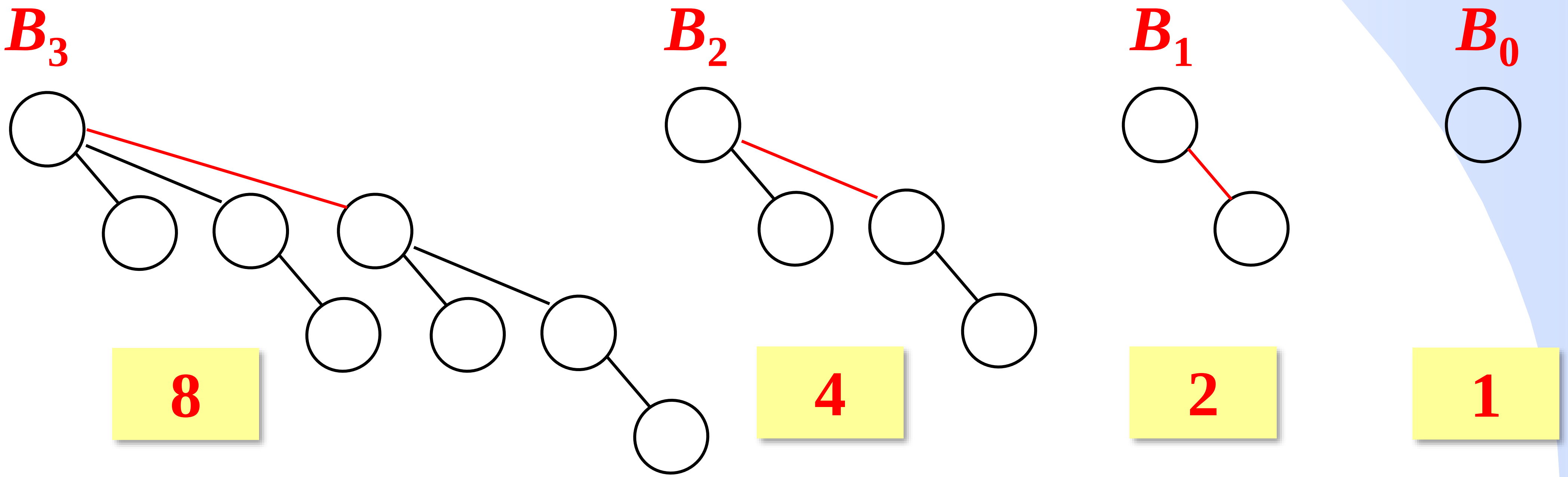


二项堆（*Binomial Heap*）

二项堆是二项树的森林，在该森林中：

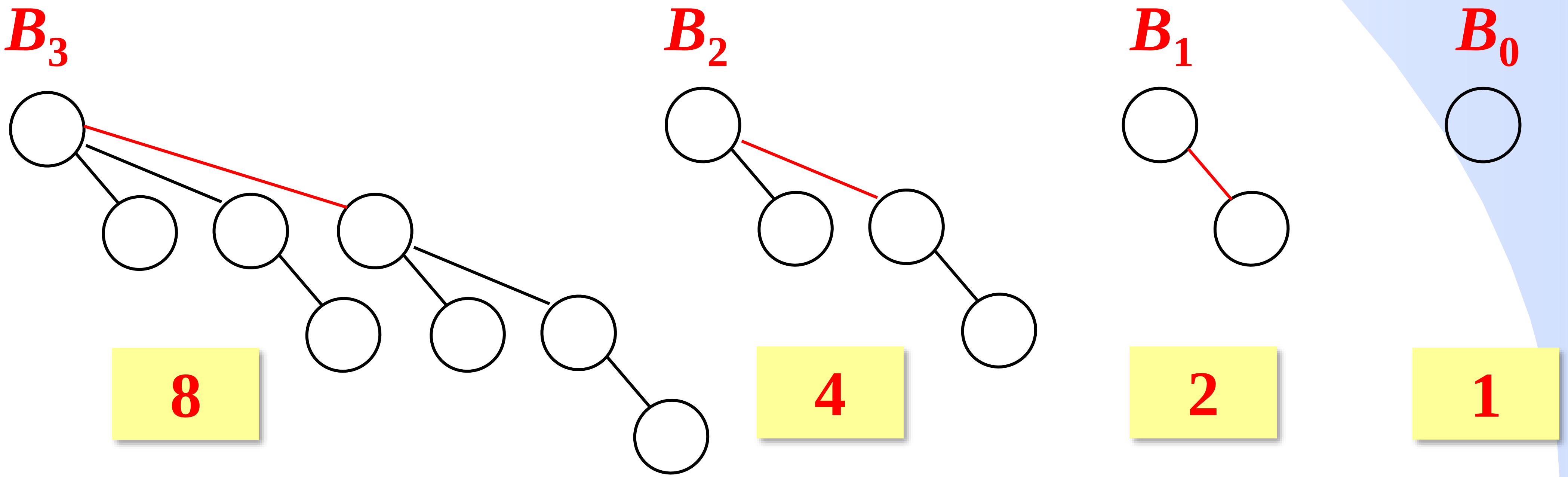
✓每棵二项树都满足堆序性；

✓同一阶的二项树最多出现1次。



二项堆的性质

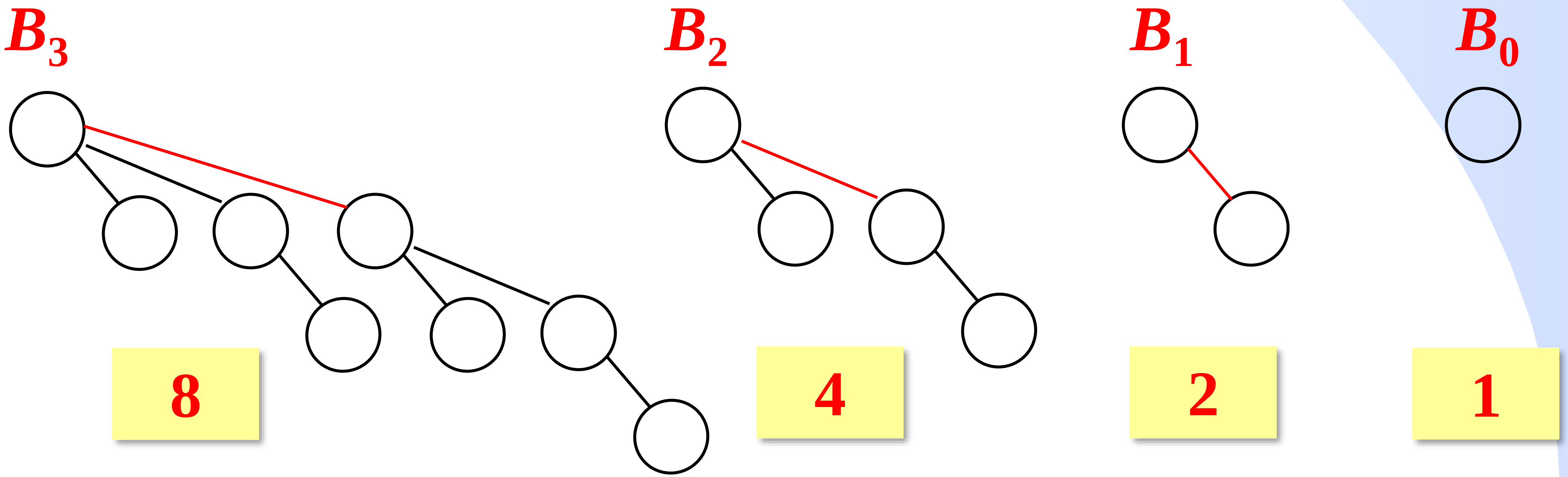
- B_k的高度为 <u>k</u> 。
- B_k的根有 <u>k</u> 棵子树，分别为 <u>$B_0, B_1, \ldots, B_{k-1}$</u> 。
- B_k包含 <u>2^k</u> 个结点。
- n个结点的二项堆最多包含 <u>$\lceil \log n \rceil$</u> 棵树。



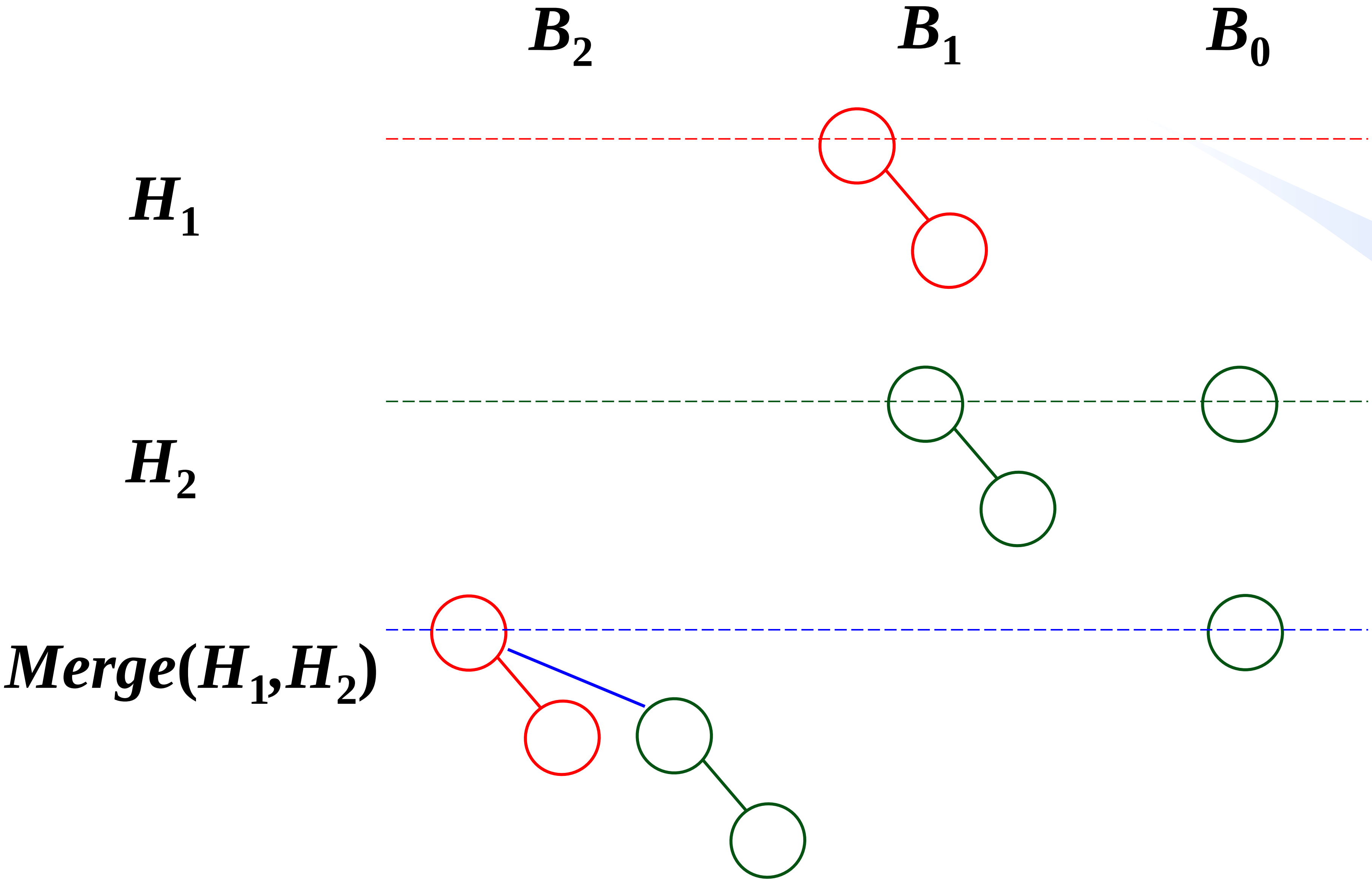
二项堆 ⇔ 二进制数

包含任意个结点的“二项堆”与“二进制数”一一对应。

- ✓包含5个结点二项堆 ⇔ 0101 ⇔ $\{B_2, B_0\}$
- ✓包含6个结点二项堆 ⇔ 0110 ⇔ $\{B_2, B_1\}$
- ✓包含11个结点二项堆 ⇔ 1011 ⇔ $\{B_3, B_1, B_0\}$



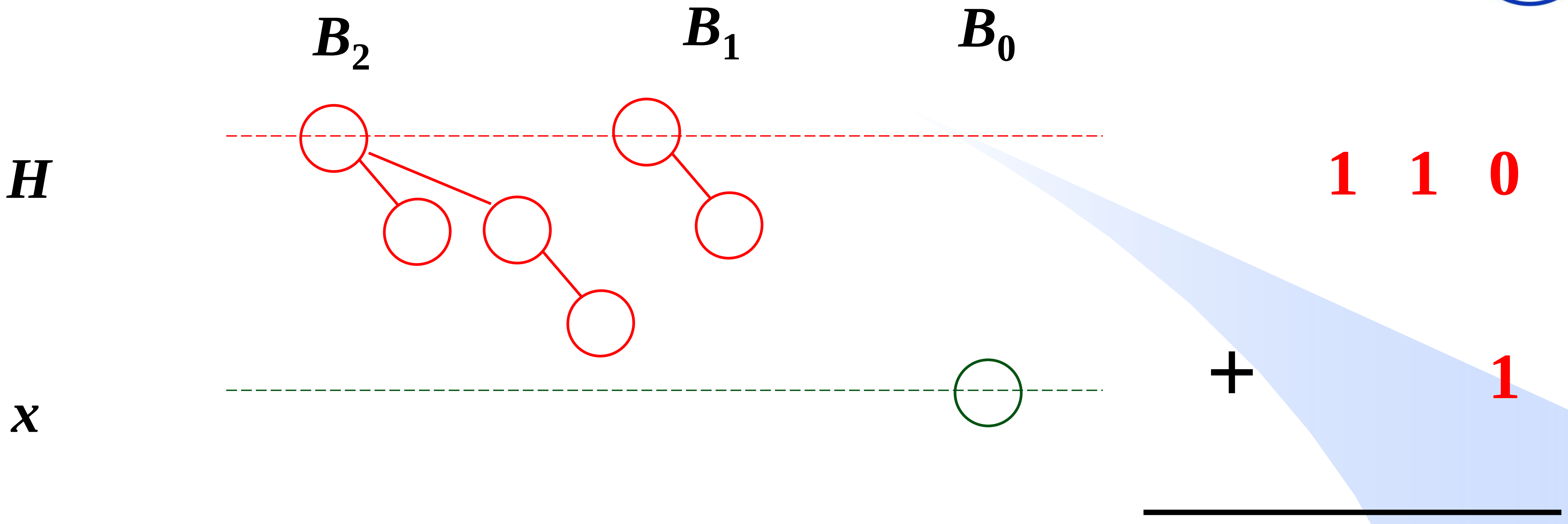
二项堆的合并——相当于二进制数加法



$$
\begin{array}{rccc}
 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 \\
\hline
 & 1 & 0 & 1
\end{array}
$$

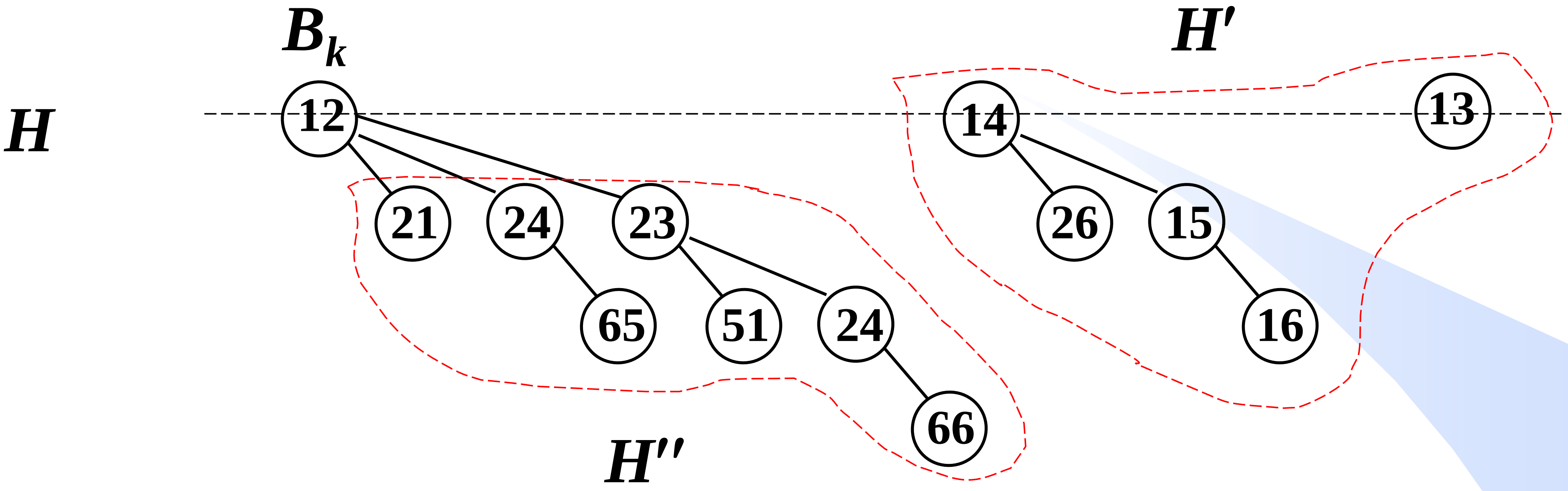
时间复杂度 O(log*n*)

二项堆的插入——合并的特例



- 插入结点*x*：将堆*x*与堆*H*合并
- 相当于*H*对应的二进制数加1
- 均摊时间复杂度O(1)

二项堆的删除



DelMin(H)

- 在H中找具有最小根的B_k，时间O(log$n$)，令$H'=H$ - B_k
- 将B_k的根删去，剩余H''，时间O(log$n$)
- Merge(H', H'')，时间O(log$n$)

时间复杂度
O(log$n$)

优先级队列（二项堆）优化Dijkstra算法

```
void Dijkstra(Vertex *Head,int n,int s,int dist[],int path[]){
  int S[N], i, j, min, v, w;
  for(i=1; i<=n; i++) {path[i]=-1; dist[i]=INF; S[i]=0;}//初始化
  dist[s]=0; Priority_Queue MinPQ;MinPQ.EnQueue(Node(s,dist[s]));
  while(!MinPQ.Empty()){
    Node q=MinPQ.DeQueue(); v=q.v;   //出队-删除 O(log n)
    if(S[v]==1) continue;
    S[v]=1;                  //将顶点v放入S集合
    for(Edge* p=Head[v].adjacent;p;p=p->link) {
       w=p->VerAdj;          //更新v的邻接顶点的D值
       if(S[w]==0 && dist[v]+p->cost<dist[w]) {
          dist[w]=dist[v]+p->cost;  path[w]=v;
          MinPQ.EnQueue(Node(w,dist[w])); //入队-插入O(1)
       }
    }
  }
}
```

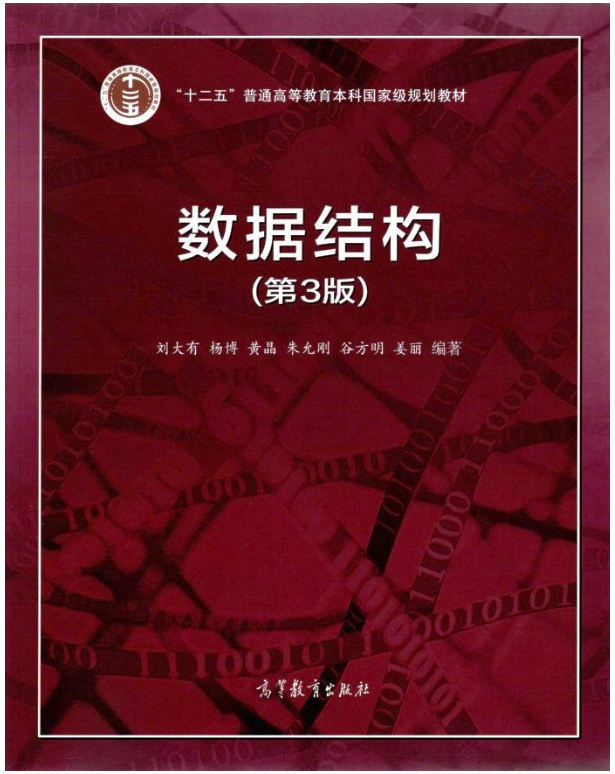
```
struct Node{
  int v, dist;
  Node(int a,int b){
    v=a; dist=b;
  }
};
```

时间复杂度O(nlogn+e)

堆优化的Dijkstra算法时间复杂度对比

	插入	删除	Dijkstra
不用堆			$O(n^2+e)$
二叉堆	$O(\log n)$	$O(\log n)$	$O((n+e)\log n)$
d堆	$O(\log_d n)$	$O(d\cdot\log_d n)$	$O((n\cdot d+e)\log_d n)$
二项堆	$O(1)$	$O(\log n)$	$O(n\log n+e)$

think.create.solve



堆排序

- 锦标赛排序
- 堆排序算法
- 堆与优先级队列
- 可合并堆
- **Top K问题**

数据之法
结构之美
算法之道

Last updated on 2022.11

zhuyungang@jlu.edu.cn

热搜榜

数组有n个元素，挑选其中最大的k ($k < n$)个元素。

数组有n个元素，挑选其中最大的k ($k < n$)个元素。【华为、腾讯、字节跳动、阿里、京东、美团、苹果、微软、谷歌面试题】

- **想法1：**递减排序后，选前k个数，O($n$log$n$)。
- **想法2：**借助直接选择排序思想，选k次最大元素，O(nk)。
- **想法3：**借助堆排序思想，选出k次堆顶元素，O(n+$k$log$n$)。

数组有n个元素，挑选其中最大的k ($k < n$)个元素。

解决方案：用包含k个元素的最小堆，存数组中最大的k个元素。

- 用前k个元素建堆，堆顶为这k个元素的最小值。
- 接着扫描数组剩余元素，每读入一个元素x，如果x大于堆顶元素，则x替换堆顶元素，调整堆。使堆始终存储当前扫描到的最大的k个元素。
- 时间复杂度$O(k+(n-k)\log k)=O(n\log k)$
- 适合增量环境。

课下思考

数组有n个元素，挑选其中最小的$k(k<n)$个元素。【当当、腾讯、百度、字节跳动、阿里、美团、微软面试题】