Data Structures & Algorithms

《数据结构与算法》课程组
重庆大学计算机学院

12

HEAP AND HEAP SORT

Outline

12.1 Heap

12.2 Heap Application

12.3 Heapsort

12.4 Comparison of Sorting Algorithms

12.1 Heap

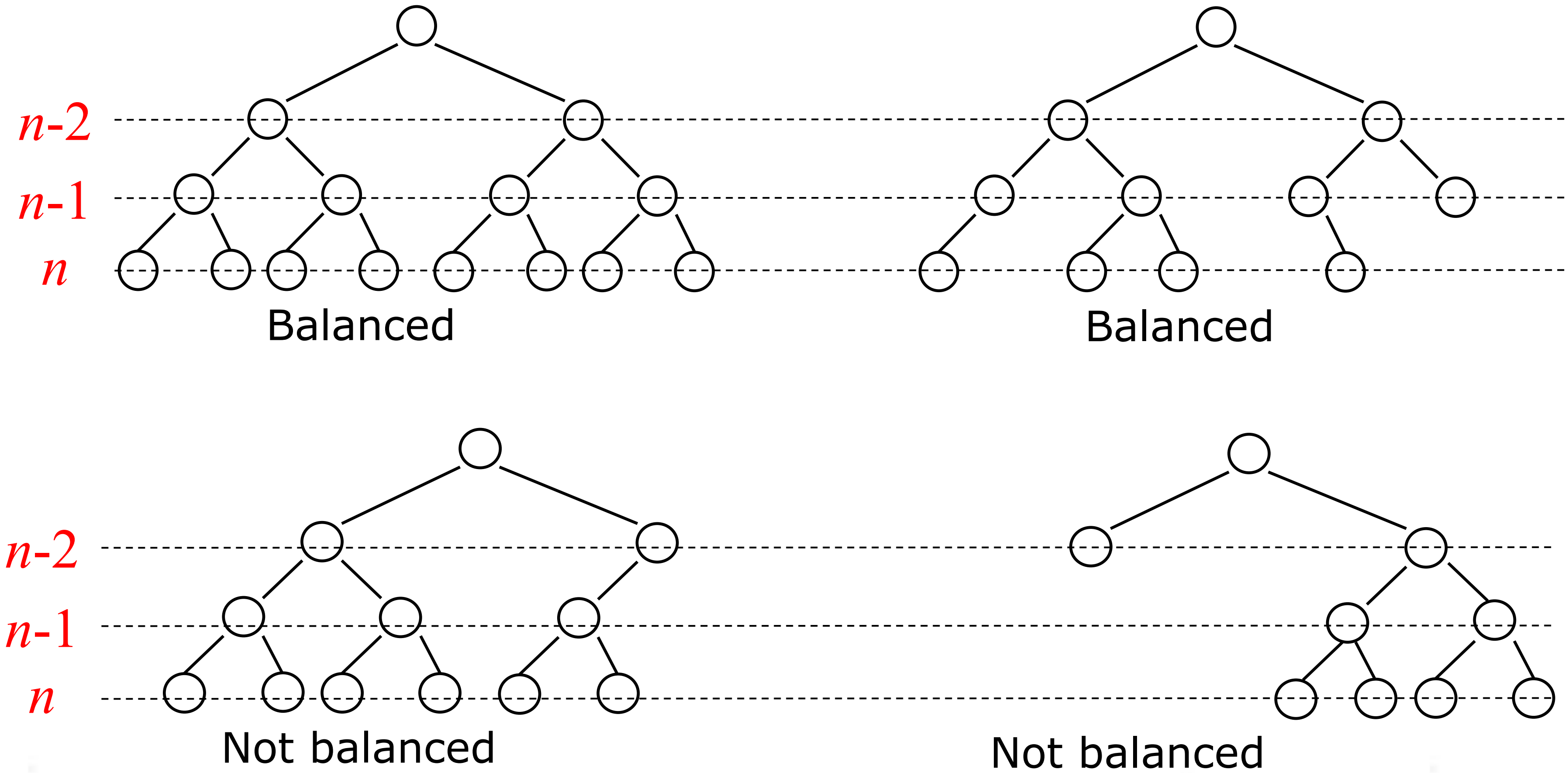
Heaps

- **Definitions of "Heap"**
  - **1. A large area of memory from which the programmer can allocate blocks as needed, and deallocate them when no longer needed**
  - **2. A balanced, left-justified binary tree (or complete tree) in which no node has a value greater (or smaller) than the value in its parent**
- **Heapsort uses the second definition**

Balanced Binary Trees

- Recall the binary trees
  - The **depth of a node** is its distance from the root
  - The **depth of a tree** is the depth of the deepest node
- A binary tree of depth n is **balanced** if all the nodes at depths 0 through n-2 have two children (full!)

Example of Balanced Binary Trees

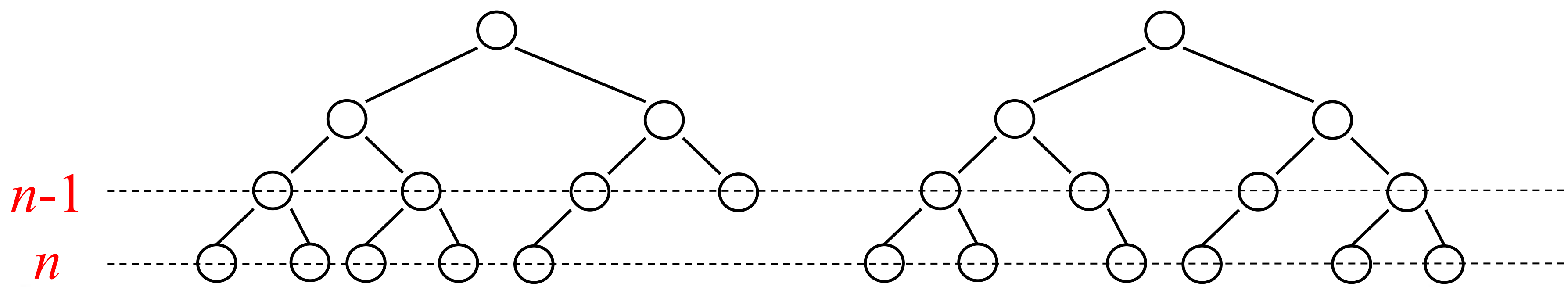


Left-justified Binary Trees (HEAP)

- A balanced binary tree is left-justified if:
  - it has 2^n nodes at depth n (the tree is "full")

or

  - all the leaves at depth n are to the left of all the nodes at depth n-1



Left-justified

Not left-justified

Heap

1. It is a left-justified (complete) binary tree
   – its height is guaranteed to be the minimum possible. In particular, a heap containing n nodes will have a height of $\lceil \log(n+1) \rceil$

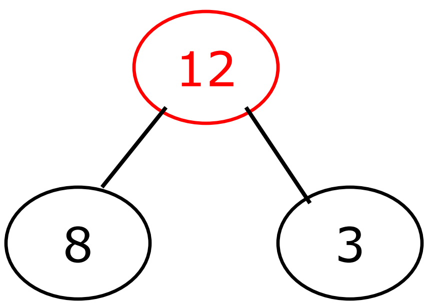
2. the values stored in a heap are **partially ordered**. This means that there is **a relationship between the value stored at any node and the values of its children**.

3. There are two variants of the heap, depending on the definition of this relationship:
   – MinHeap: key(parent) $\leq$ key(child)
   – MaxHeap: key(parent) $\geq$ key(child)]

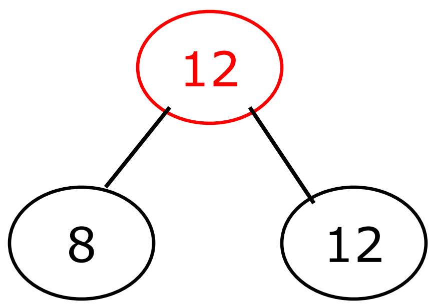
- Note : there is no necessary relationship between the value of a node and that of its sibling in either the min-heap or the max-heap.

The Max Heap

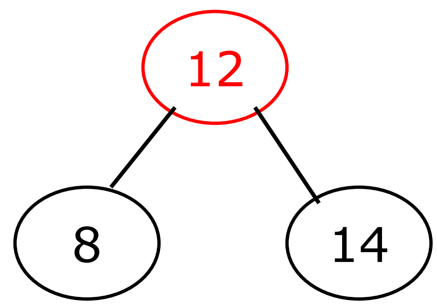
- **A heap where the maximum element is at the top of the heap and the next to be popped.**
  - **The max-heap property: the value in the node is as large as or larger than the values in its children.**



Red node has the max-heap property



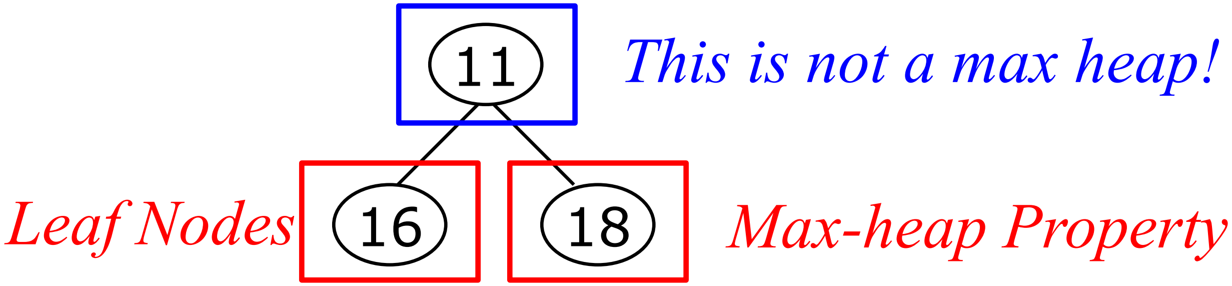
Red node has the max heap property



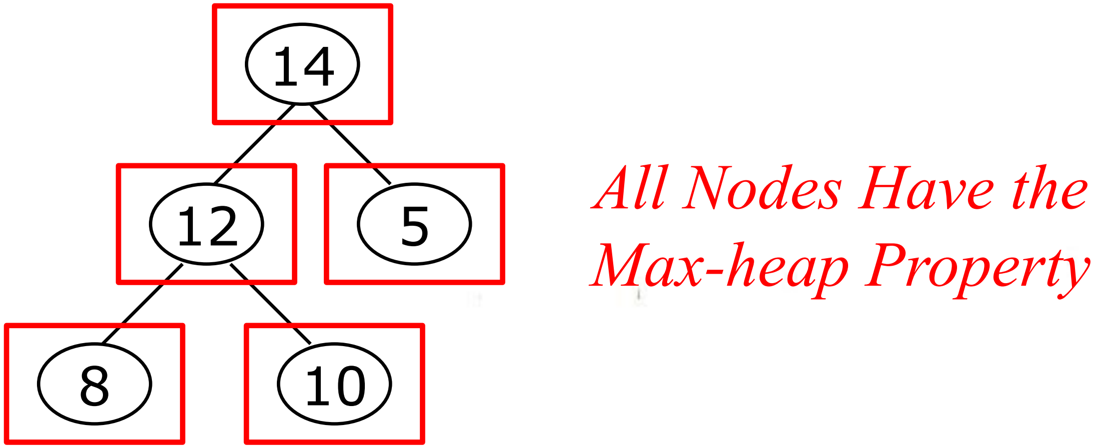
Red node does not have the max heap property

The Max-heap Property

- All leaf nodes automatically have the heap property



- A binary tree is a max-heap if all nodes in it have the max-heap property

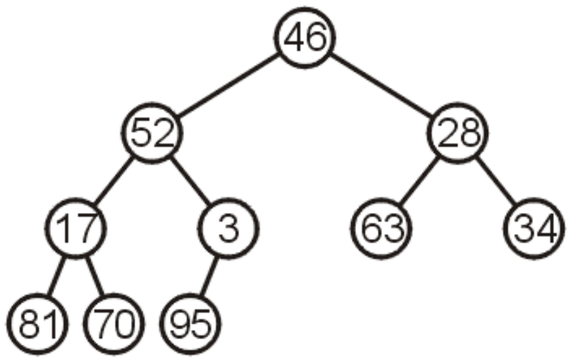


Constructing a Max-heap

- Consider this unsorted array with starting index at 1:

46	52	28	17	3	63	34	81	70	95

- We can transform this array into the following complete tree:



This is NOT a max-heap

- Where for each node:
  - The children of the k-th element are the $2k$-th and $2k+1$-th elements
  - The parent node of the k-th element is the $\lfloor k/2 \rfloor$
- **HOW TO TRANSFORM this array into a max-heap?**

Siftup operation

```
void siftup(int p) {                    //Heap[p]是需上升（前移）元素
    while(p > 1)                        // p不是根位置
    {
        if(Comp::prior(Heap[p], Heap[p/2])) {   //比父节点优先
            swap(Heap[p], Heap[p/2]);           //与父节点交换
            p = p / 2;                  //移到父节点位置，继续上升
        }else{
            break;                      //如果父节点优先，结束上升
        }
    }
}
```

Time complexity = $O(\log(n))$

Building a heap (Top-down)

- Insert the elements one at a time. (similar to insertion sort)

```
        //最大堆Heap[1..last], 插入新元素it
void insert(const E& it)  {
    Assert(last < maxsize, "Heap is full");
    Heap[++last] = it;    //新元素先放在堆最后
    siftup(last);         //向上移动
}
```

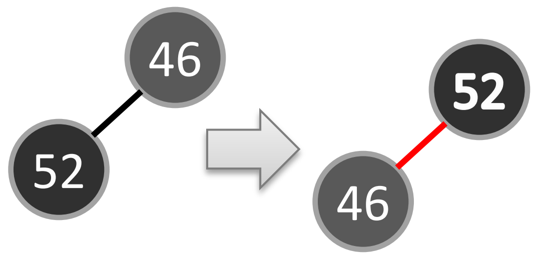
- Each call to insert takes $\Theta(\log(n))$ time in the worst case, because the value being inserted can move at most the distance from the bottom of the tree to the top of the tree.

- Thus, to insert n values into the heap, if we insert them one at a time, will take $\Theta(n \log(n))$ time in the worst case.

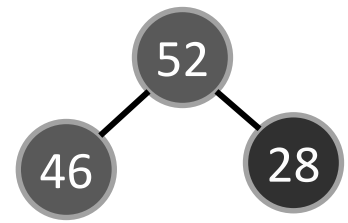
Building a heap (Top-down)

46	52	28	17	3	63	34	81	70	95

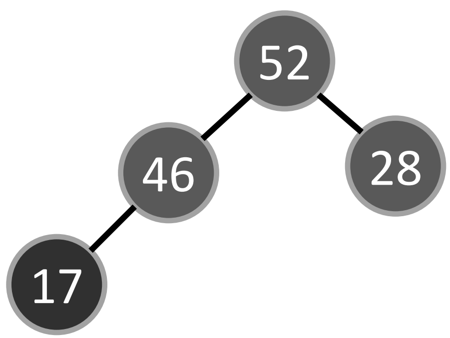
(1) insert 46



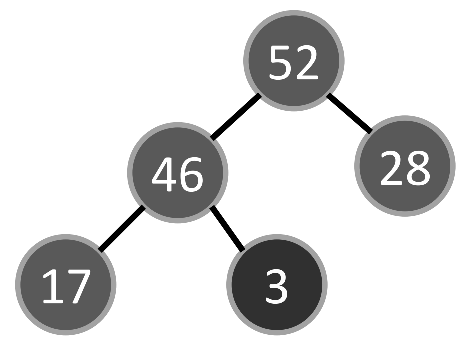
(2) insert 52



(3) insert 28



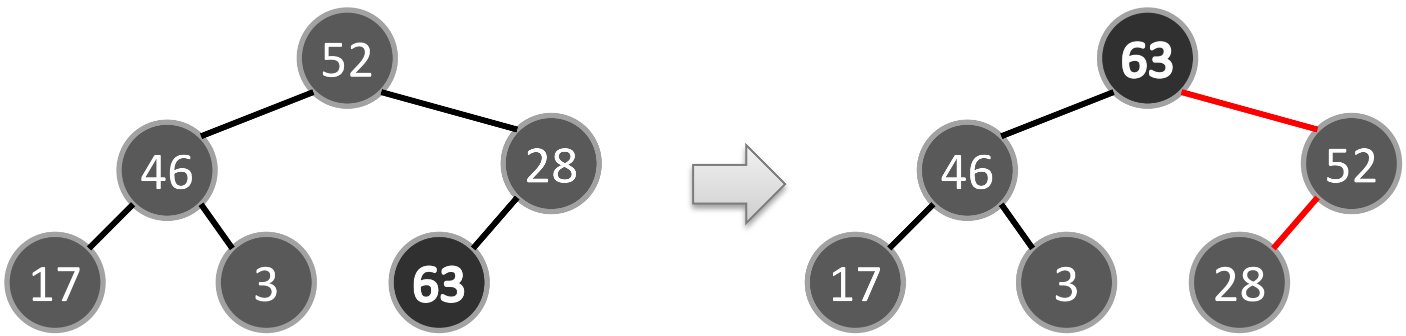
(4) insert 17



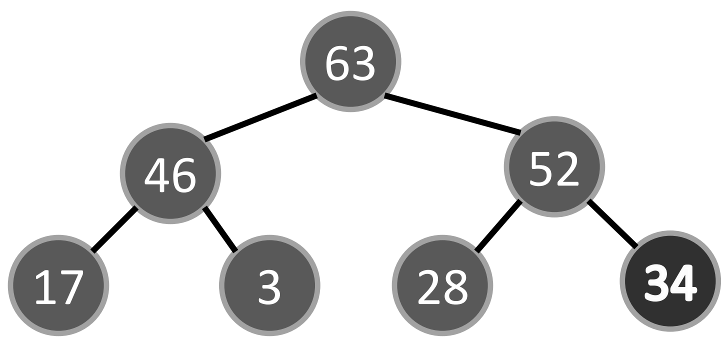
(5) insert 3

Building a heap (Top-down)

46	52	28	17	3	63	34	81	70	95



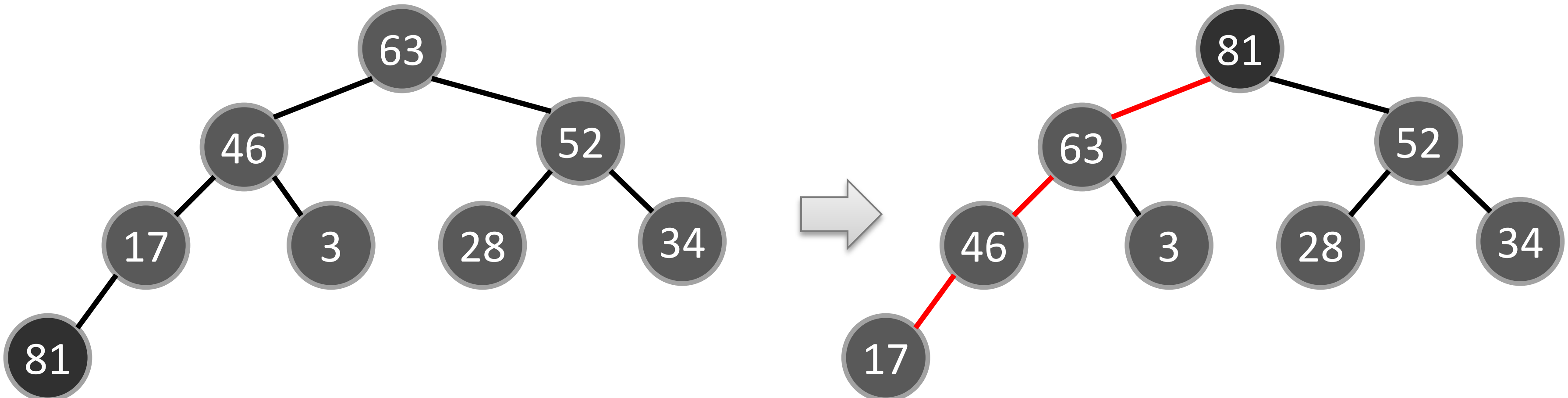
(6) insert 63



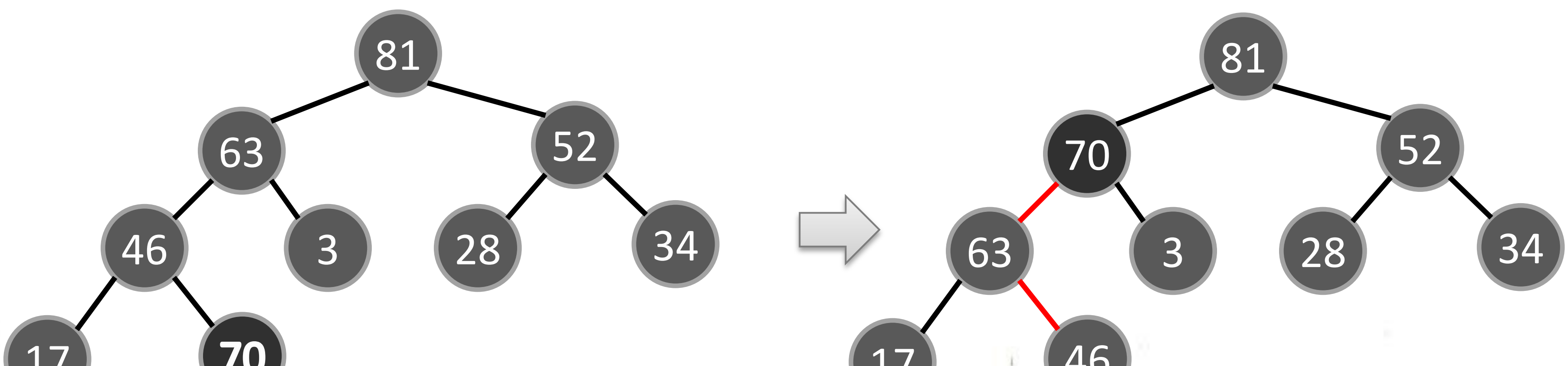
(7) insert 34

Building a heap (Top-down)

46	52	28	17	3	63	34	81	70	95

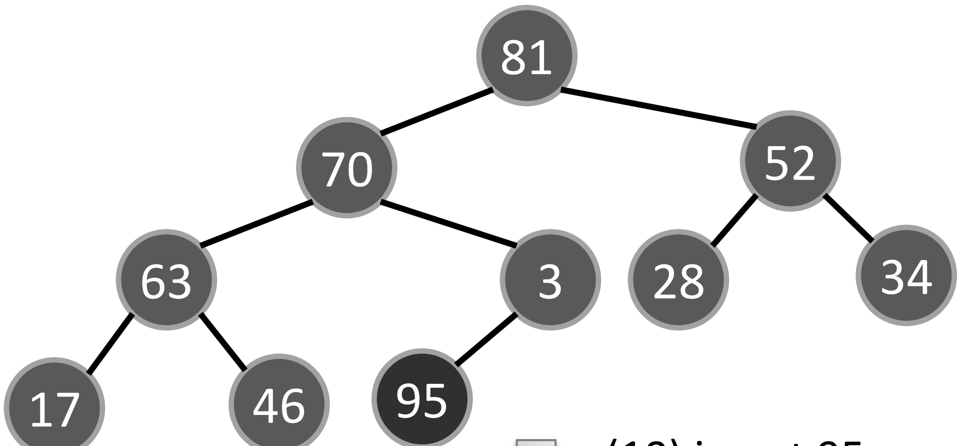


(8) insert 81

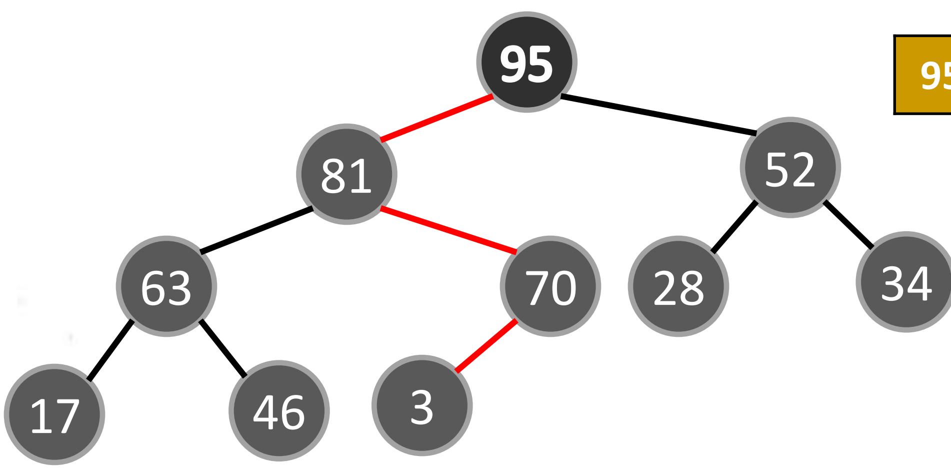


(9) insert 70

Building a heap (Top-down)



(10) insert 95



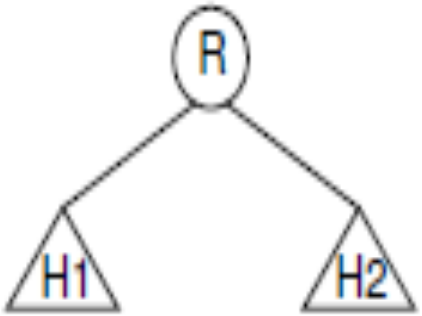
46	52	28	17	3	63	34	81	70	95

95	81	52	63	70	28	34	17	46	3

max heap

Building a heap (a faster way)

Suppose that the left and right subtrees of the root are already heaps, and R is the name of the element at the root. In this case there are two possibilities.



(1) Value(R) ≥Value(children) : construction is complete.

(2) Value(R) < one or both of Value(children): R should be exchanged with the child that has greater value.

- The result will be a heap, except that R might still be less than one or both of its (new) children.
- In this case, we simply continue the process of "percolating down" R until it reaches a level where it is greater than its children, or is a leaf node. This process is implemented by the method siftdown.

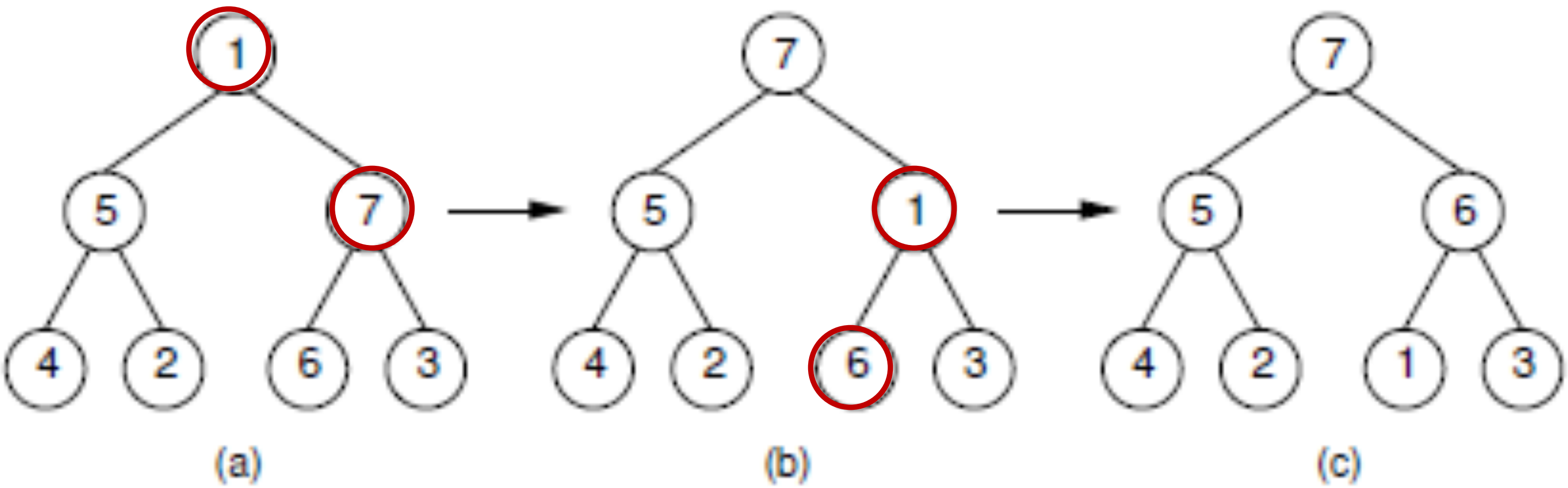
Siftdown operation

```
void siftdown(int p) {                    //Heap[p]是需下沉（后移）元素
    while(2*p <= last)                       // p不是叶子
    {
        int cld = 2*p;          //p的左边子节点
        if(cld <last && Comp::prior(Heap[cld+1], Heap[cld])
            cld = cld + 1;          //右子节点不为空，取较优值

        if(Comp::prior(Heap[cld], Heap[p])) {   //子节点更优
            swap(Heap[p], Heap[cld]);             //与子节点交换
            p = cld;                    //移到子节点位置，继续下沉
        }else{
            break;                        //如果p比子节点优先，结束下沉
        }
    }
}
```

Time complexity = $O(\log(n))$

Siftdown operation

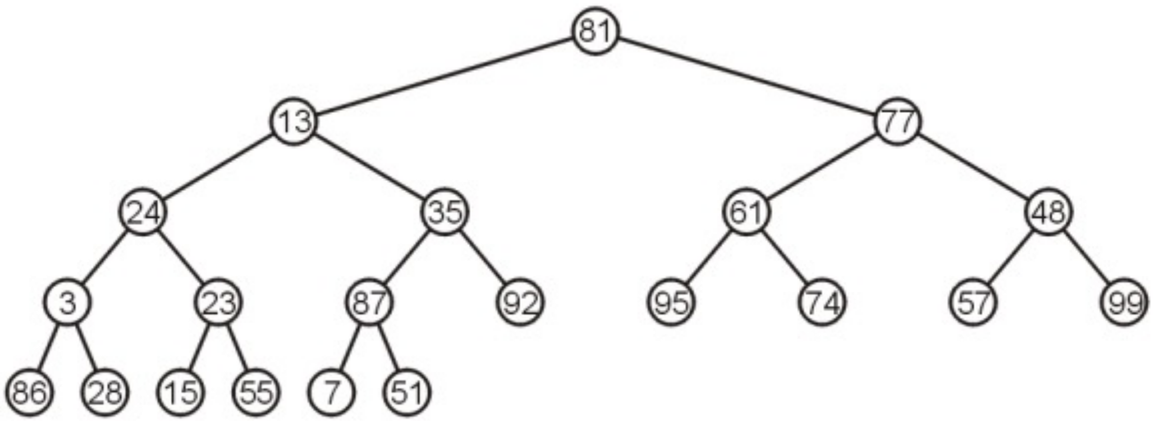


The subtrees of the root are assumed to be heaps.
(a) The partially completed heap.
(b) Values 1 and 7 are swapped.
(c) Values 1 and 6 are swapped to form the final heap.

Building a heap (Bottom-up)

- To see if this can be done, consider the following array:

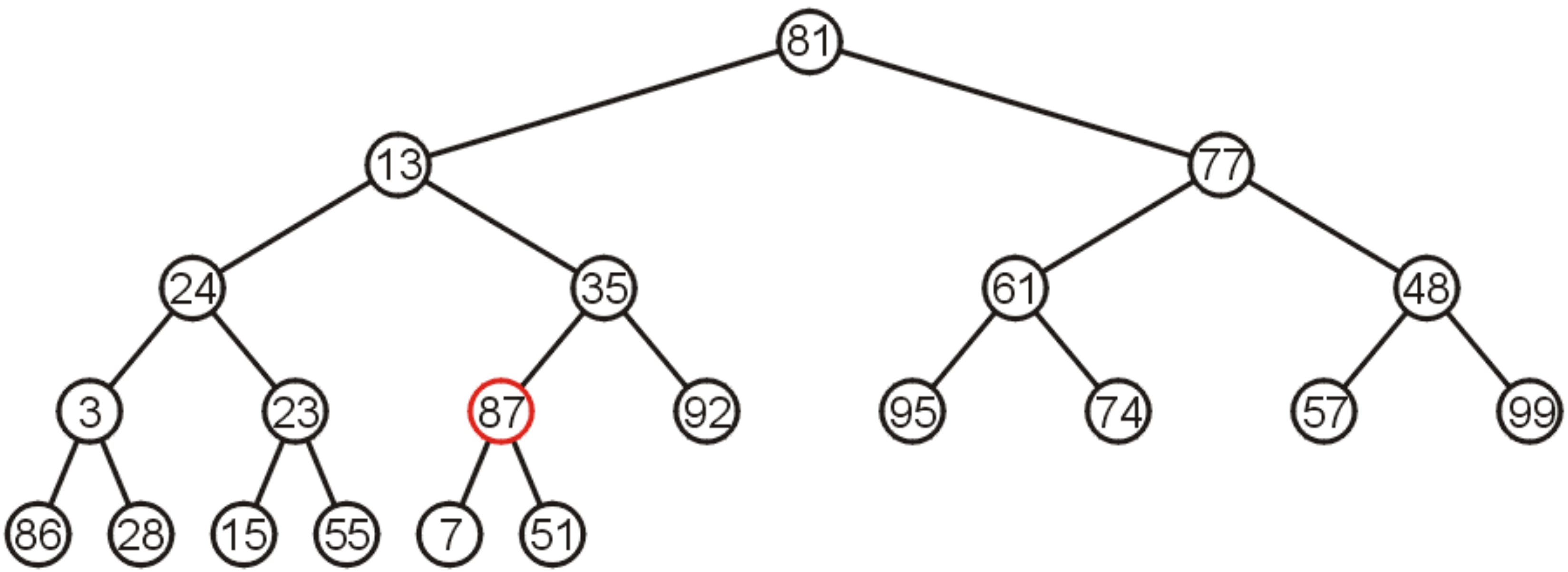
81	13	77	24	35	61	48	3	23	87	92	95	74	57	99	86	28	15	55	7	51



- It would be **exceptionally difficult** to start by determining what should be in the root.
  - We can work bottom-up instead: each **LEAF** node is a max-heap on its own.

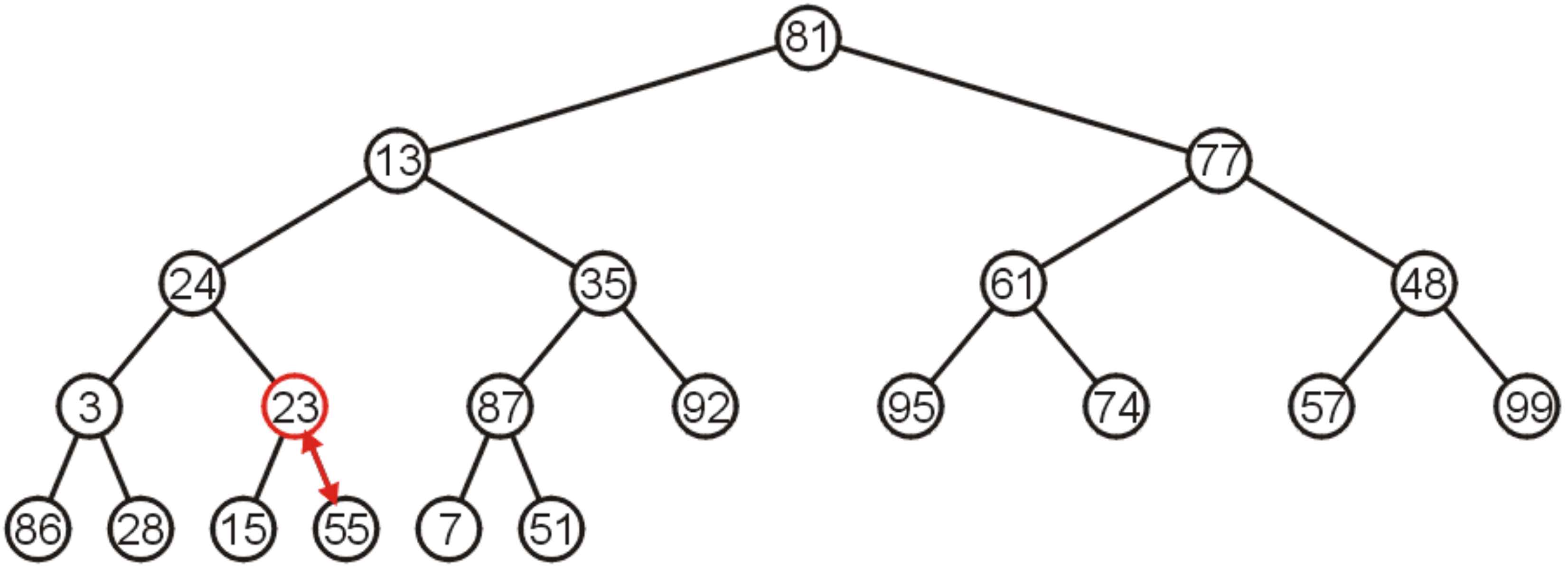
Building a heap (Bottom-up)

- **Starting at the back, we note that all leaf nodes are trivial heaps.**
- **Also, the sub-tree with the node 87 as the root is a max-heap.**



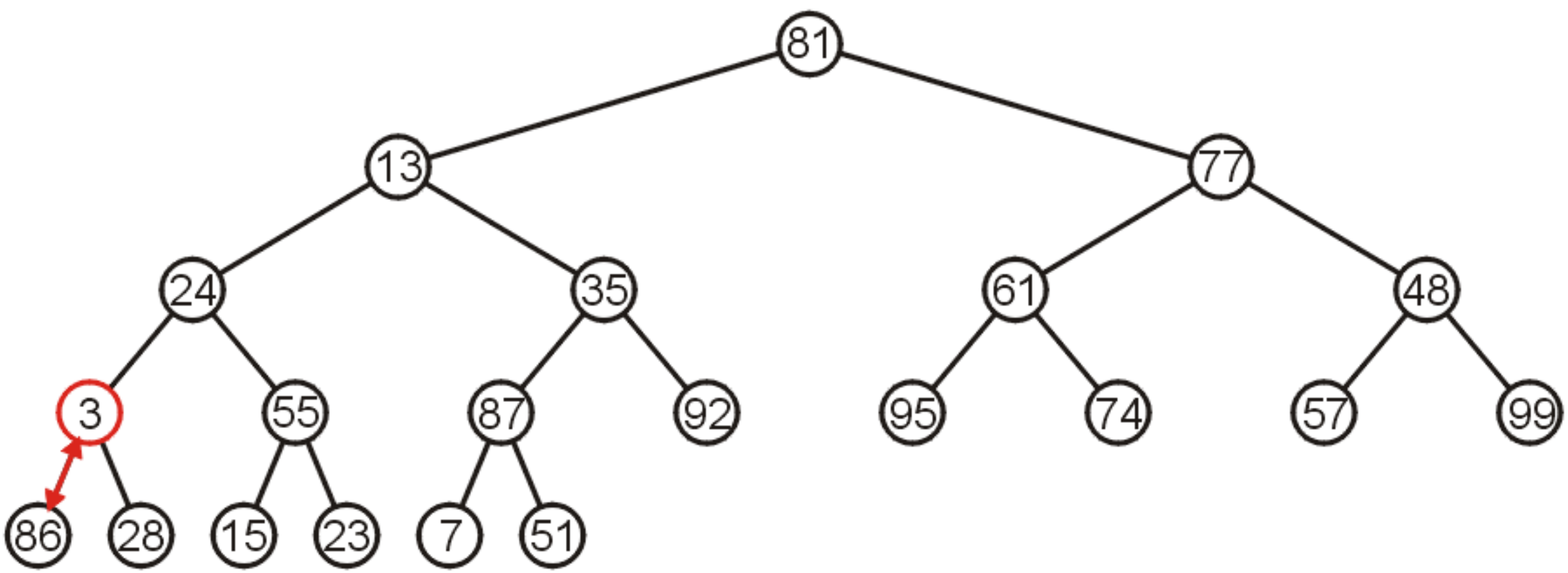
Building a heap (Bottom-up)

- The sub-tree with node 23 is not a max-heap, but swapping it with 55 creates a max-heap.
- This process is the aforementioned sift down.



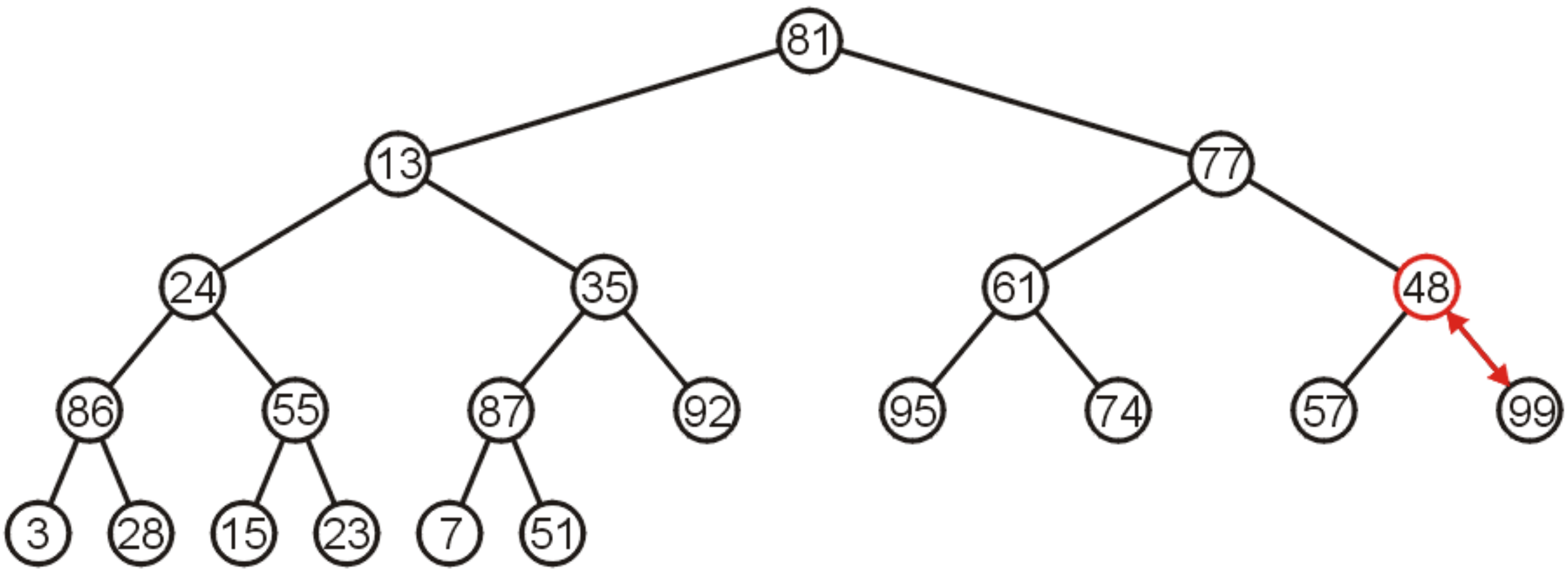
Building a heap (Bottom-up)

- The sub-tree with 3 as the root is not max-heap, but we can swap 3 and the maximum of its children: 86.



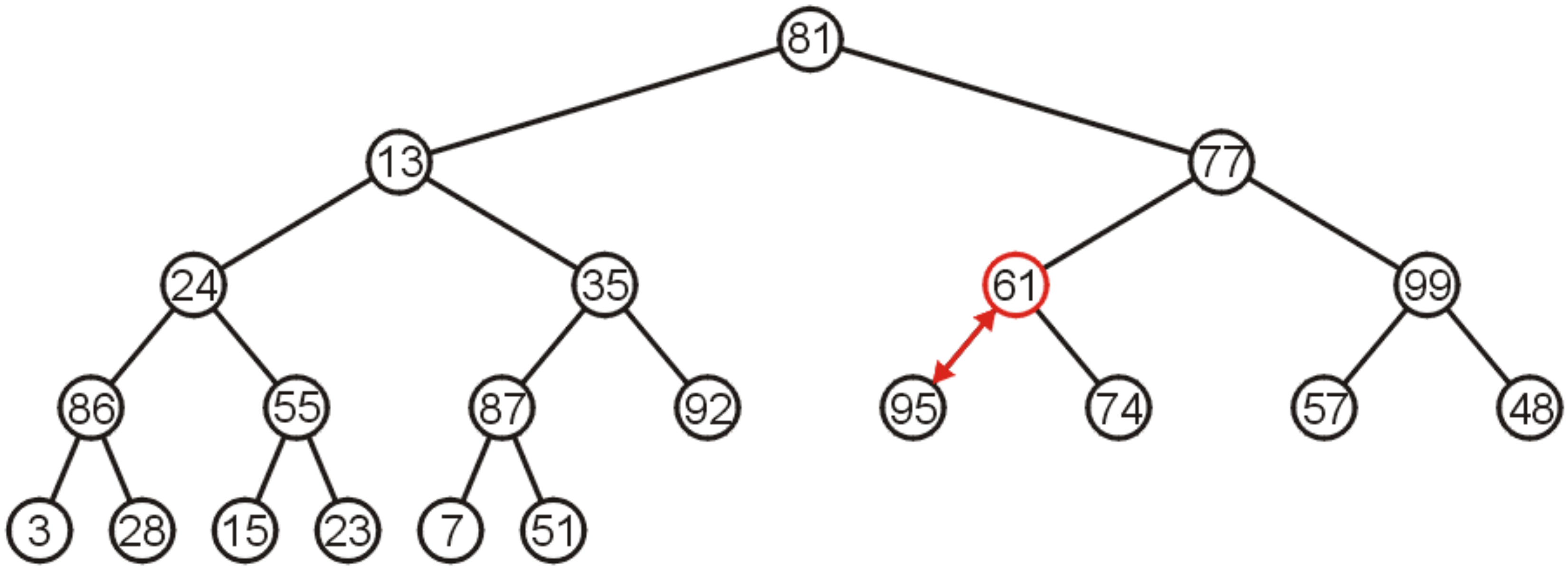
Building a heap (Bottom-up)

- Starting with the next higher level, the sub-tree with root 48 can be turned into a max-heap by swapping 48 and 99.



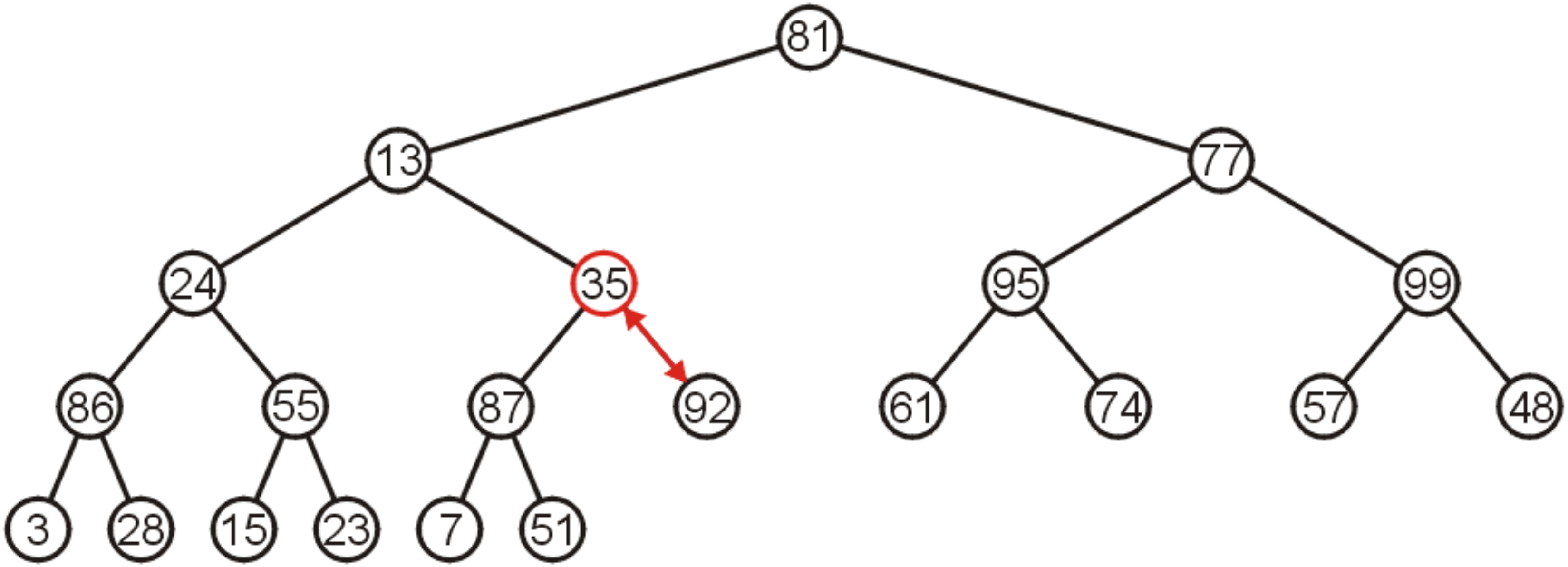
Building a heap (Bottom-up)

- **Similarly, swapping 61 and 95 creates a max-heap of the next sub-tree.**



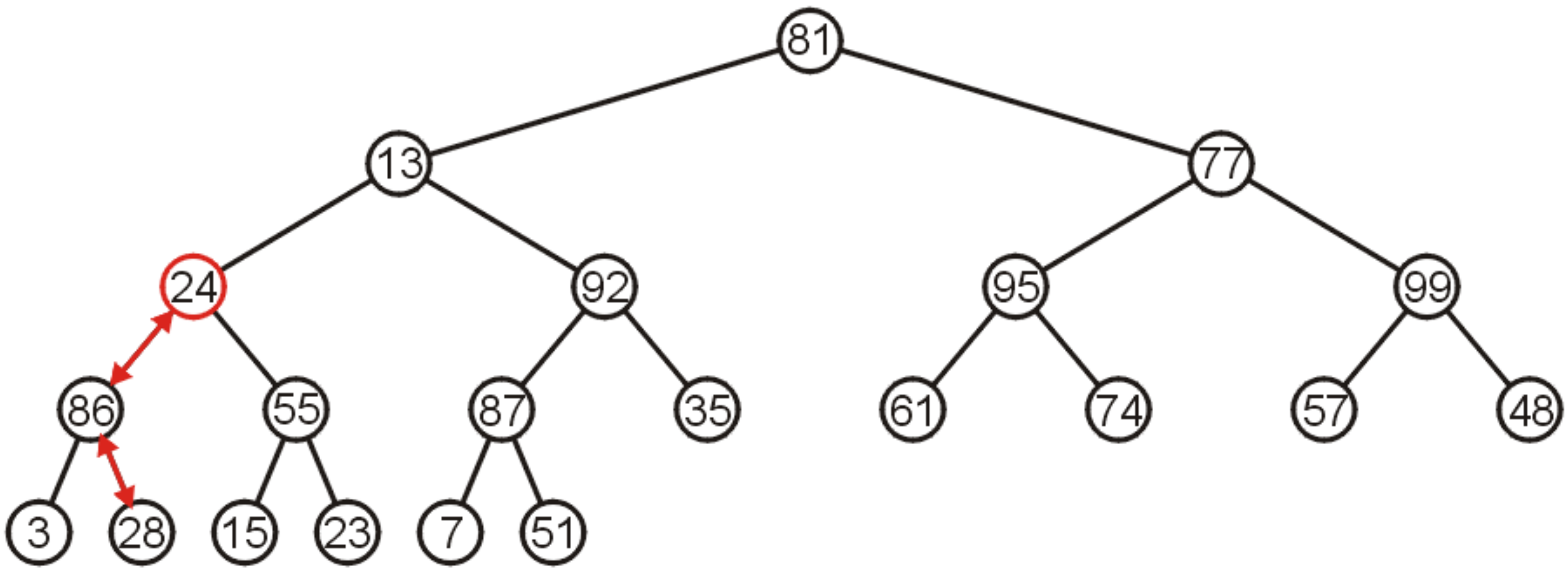
Building a heap (Bottom-up)

- As does swapping 35 and 92.



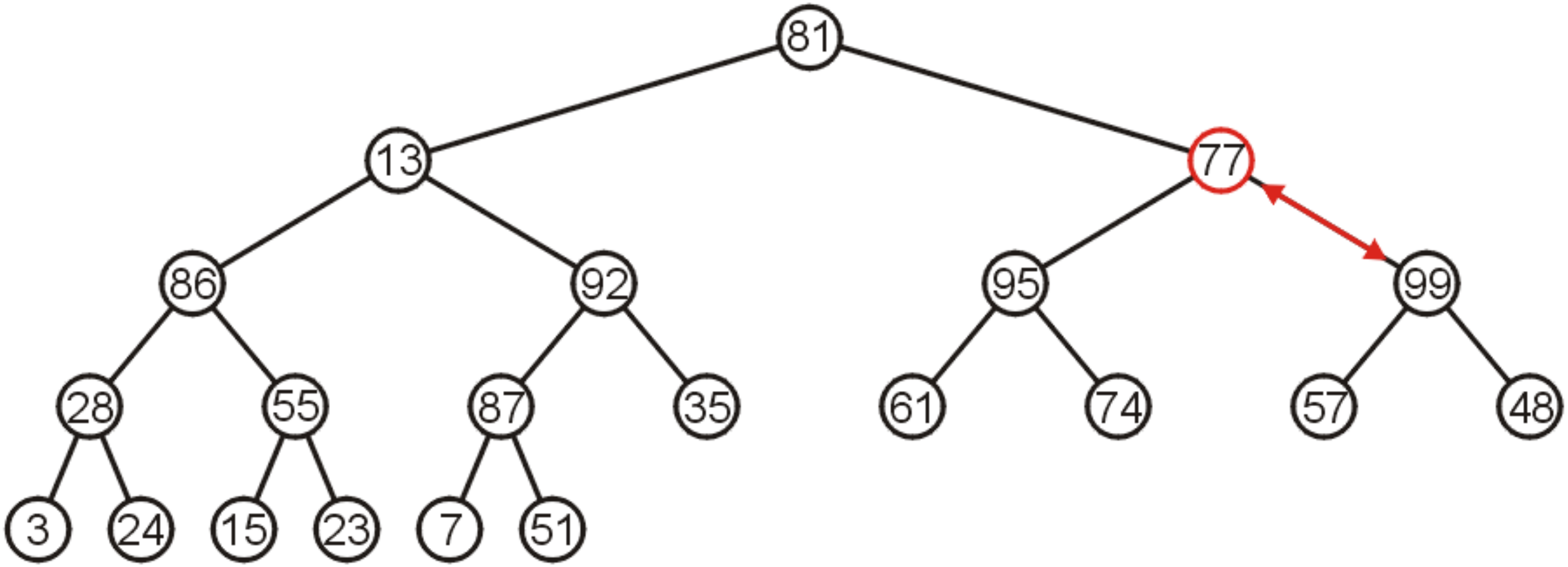
Building a heap (Bottom-up)

- The sub-tree with root 24 may be converted into a max-heap by first swapping 24 and 86 and then swapping 24 and 28.



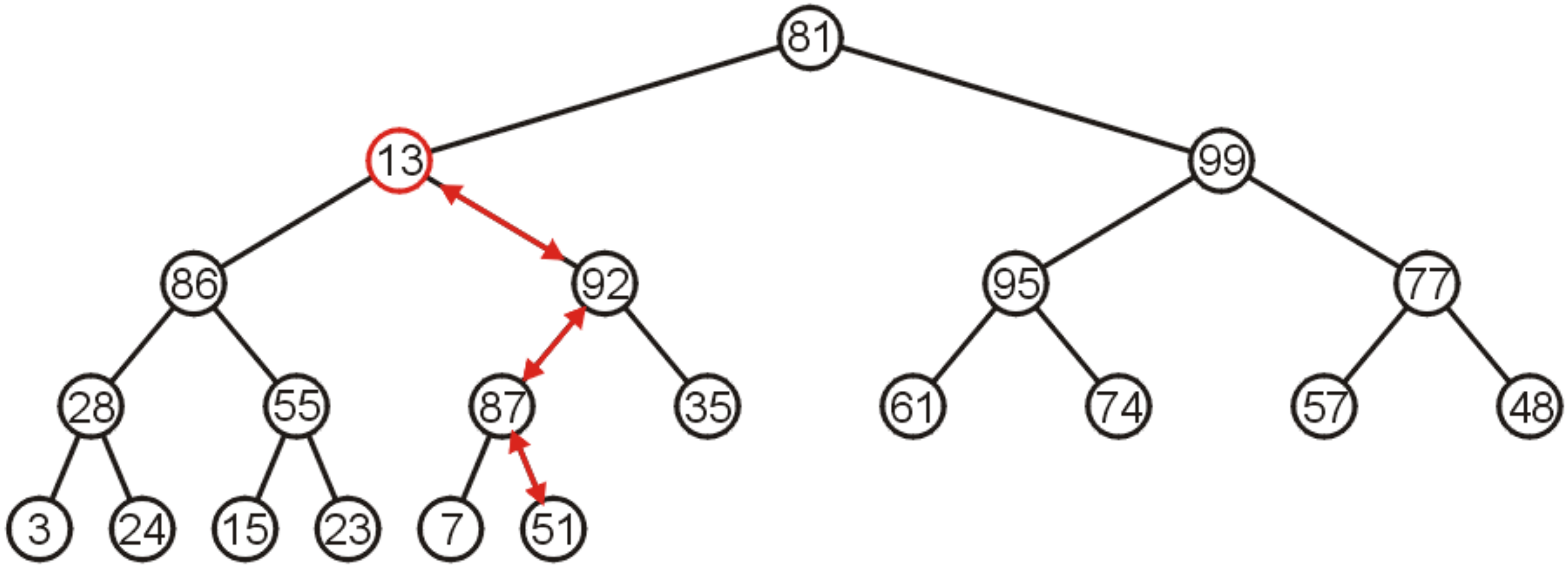
Building a heap (Bottom-up)

- The right-most sub-tree of the next higher level may be turned into a max-heap by swapping 77 and 99.



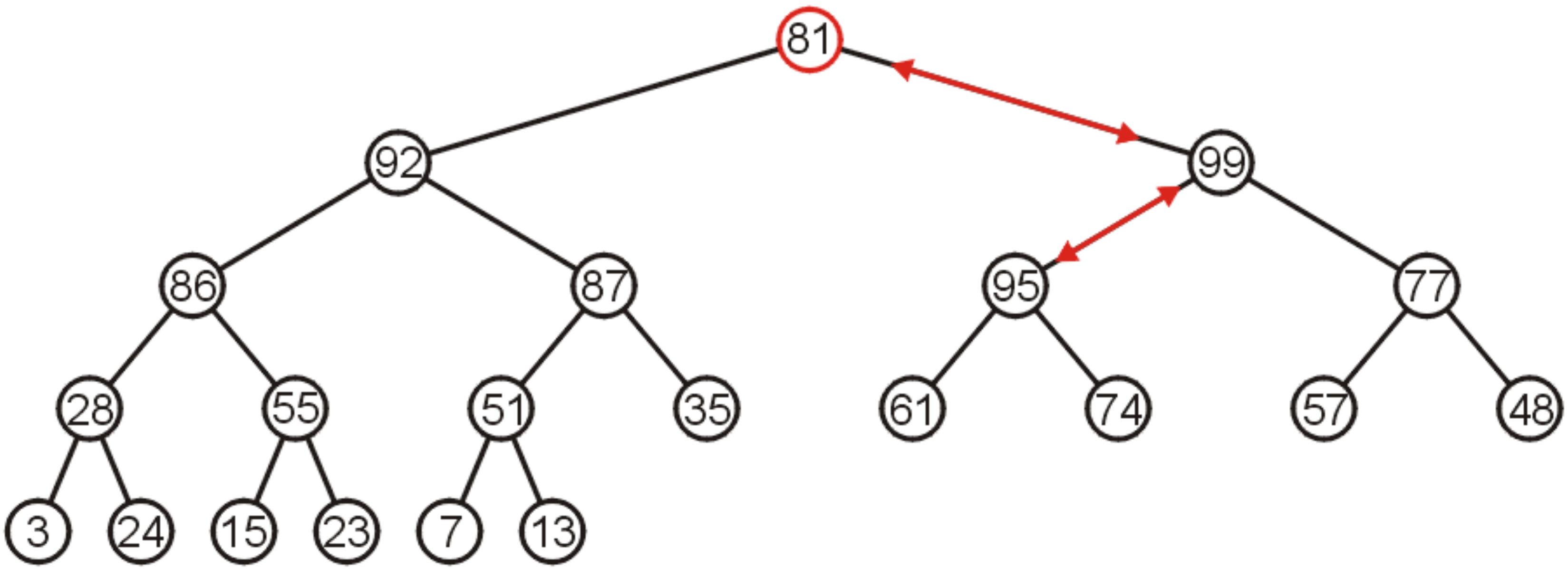
Building a heap (Bottom-up)

- However, to turn the next sub-tree into a max-heap requires that 13 be percolated down to a leaf node.



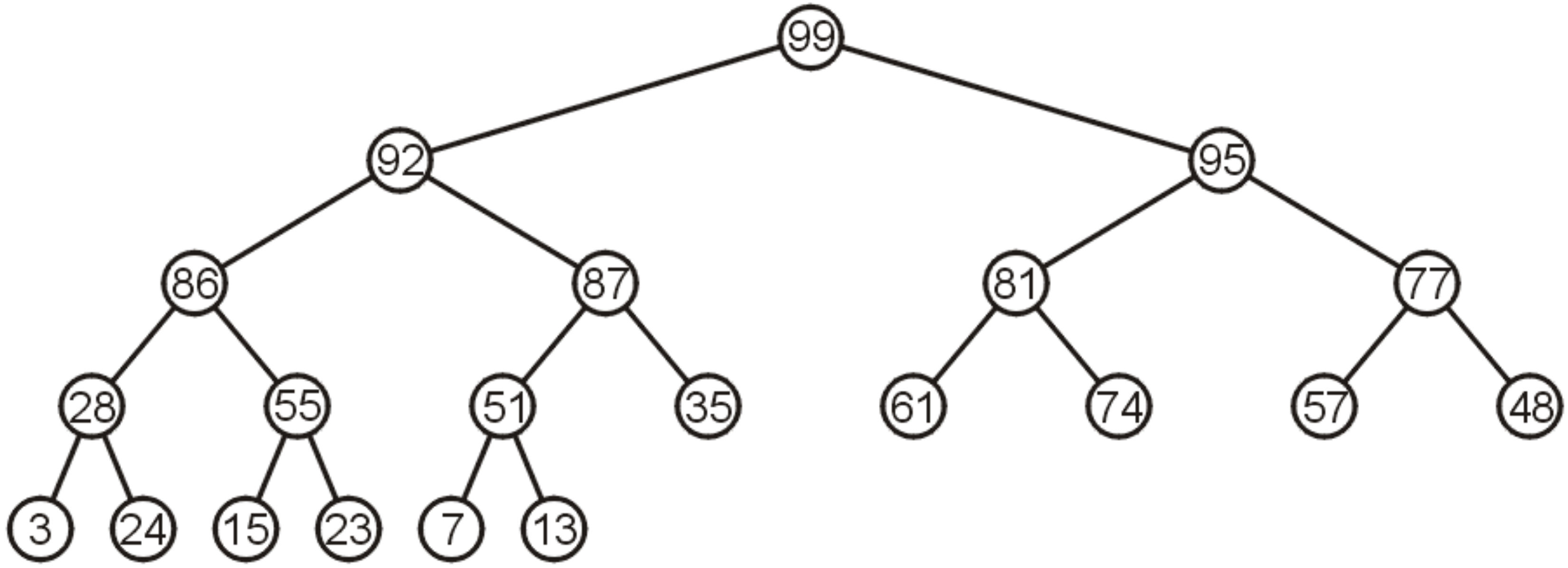
Building a heap (Bottom-up)

- **The root need only be percolated down by two levels.**



Building a heap (Bottom-up)

- **The final product is a max-heap.**



The General Idea of the Heap Construction

- A bottom-up approach
  - Starting from the last element of the given array

- To ensure that the checked nodes all posses the max-heap property
  - For each node, find the max node of the triple {current-node, left-child, right-child}
  - Adjust the structure of the triple by **siftdown**
  - Note that the percolating down could violate the max-heap-property of the **SUB-TREE** by the current node.
  - It is the most important to maintain the max-heap property of the sub-tree, which is done by **recursion**.
  - But note that to implement the percolating down, we only have to swap the current node with its **larger** child; thus, we only have to examine the max-heap-property of this sub-tree.

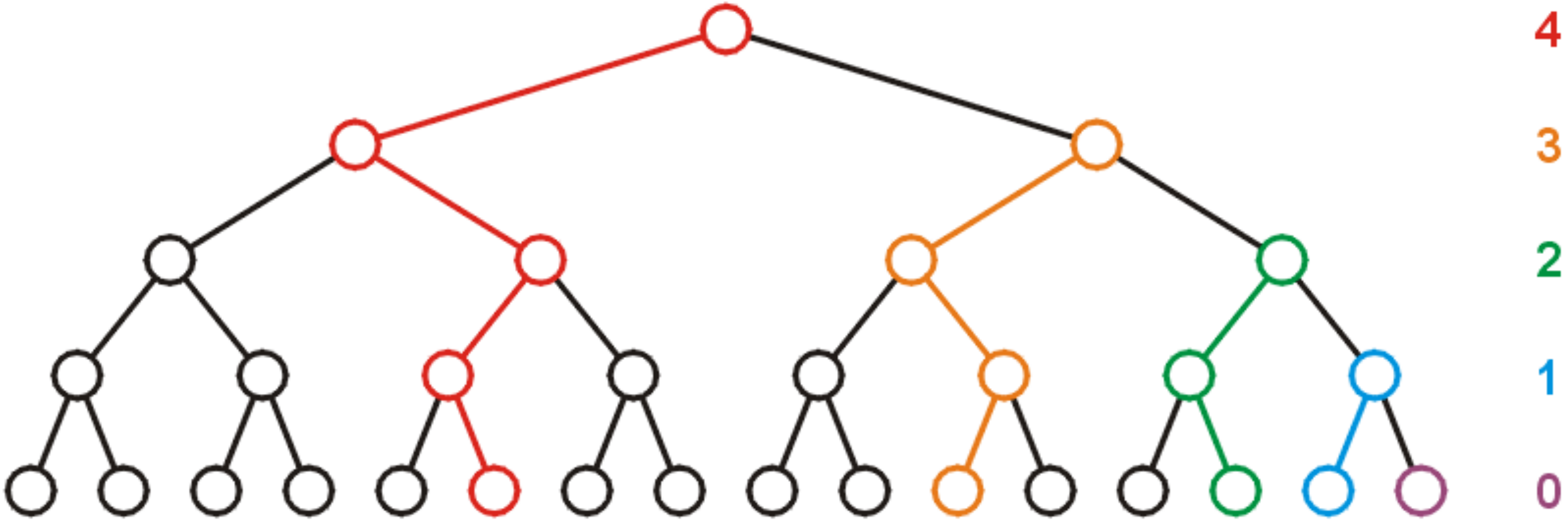
Building a heap (heapify)

```
void heapify( )  {
    for(int p=last/2; p>0; p--)  {  //从最右边的第一个中间节点开始
        siftdown(p);
    }
}
```

- Cost(heapify) = is the sum of all cost(siftdown)
- Each siftdown operation can cost at most the number of levels it takes for the node being sifted to reach the bottom of the tree.
- So, this algorithm takes $O(n)$ time in the worst case (why?)

Run-time Analysis of heapify

- **Considering a perfect tree of height *h* :**
  - **The maximum number of swaps which a second-lowest level would experience is 1; the next higher level, 2; and so on.**



Run-time Analysis of heapify

- At depth k, there are 2^k nodes and in the worst case, all of these nodes would have to sift down $h - k$ levels
  - In the worst case, this would requiring a total of $2^k(h - k)$ swaps
  - the mathematical expression of this sum comes to:

$$\sum_{k=0}^{h} 2^k(h-k) = \left(2^{h+1} - 1\right) - (h+1)$$

Run-time Analysis of heapify

- **A complete binary tree takes $n = 2^{h+1} - 1$ nodes**
- **$h + 1 = \log(n + 1)$**
- **therefore**

$$\sum_{k=0}^{h} 2^k (h - k) = n - \log(n+1)$$

- **Each swap requires two comparisons (which child is greatest), so there is a maximum of $2n$ (or $O(n)$) comparisons**

Heap removal

- Removing the maximum (root) value from a heap containing n elements requires
  - maintain the complete binary tree shape,
    - by moving the element in the last position in the heap (the current last element in the array) to the root position.
  - the remaining n-1 node values conform to the heap property.
    - If the new root value is not the maximum value in the new heap, use siftdown to reorder the heap.
- the cost of deleting the maximum element is $O(\log(n))$ in the average and worst cases, since the heap is log(n) levels deep.

Heap removal Implementation

- 取出堆顶元素（最大值或最小值）

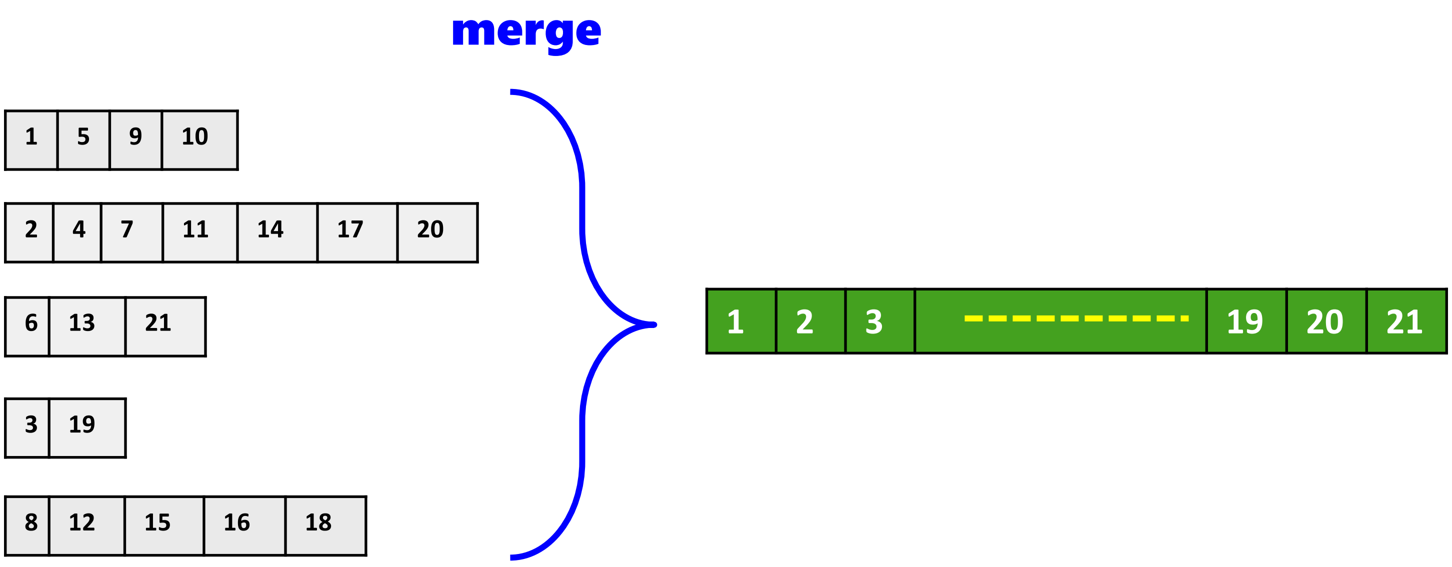
```
E removefirst()  {     // pop()
    Assert( last > 0, “Heap is empty”);

    E tmp = Heap[1];   //拷贝堆顶元素
    Heap[1] = Heap[last--];
                      //把最后一个元素移动堆顶
                      //堆长度减1
    siftdown(1);
                      //下沉堆顶，调整堆
    return tmp;
}
```

12.2 Heap Application

合并n个升序或降序序列

把n个升序（降序）序列：$A_1, A_2, \ldots, A_n$，合并成一个升序（降序）序列。

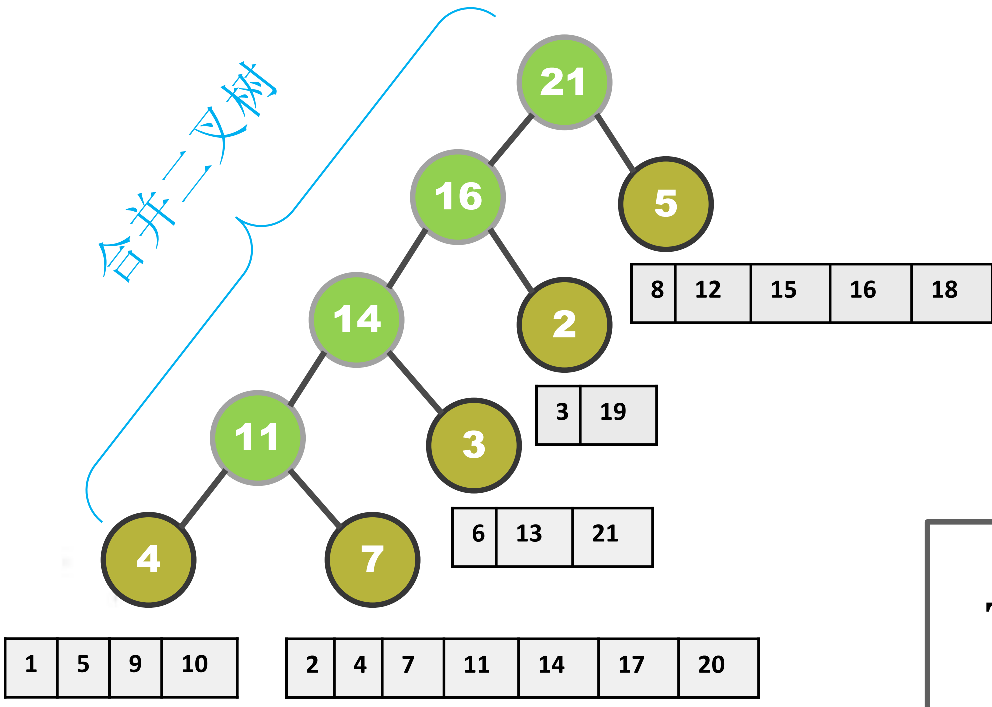


合并n个升序或降序序列

（方法1）顺序合并（两两合并）

- 可用N个叶节点的(Full)二叉树表达合并过程

- 比较总次数：62



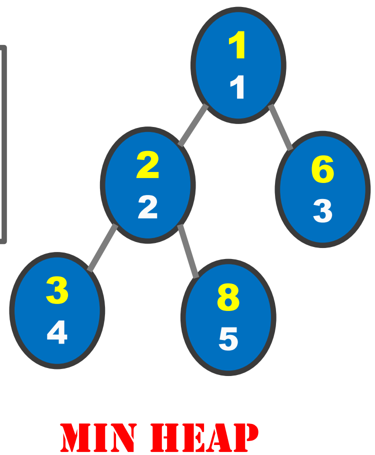
- 中间节点的权重表示两个序列合并后的长度及合并时间（比较次数）
- 合并总时间等于中间节点的权重和

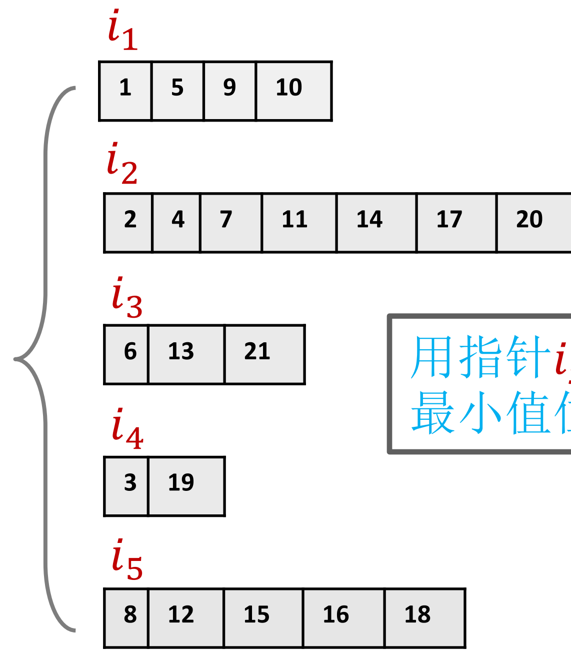
$$T\left(\sum_{1\le i\le n}|A_i|\right)=\sum_{i=1}^{n} Depth(A_i) * |A_i|$$

合并n个升序或降序序列

（方法2）同步合并 with heap!

设计存放所有$\langle A_x[i_x], x\rangle$的最小堆$(1 \le x \le n)$



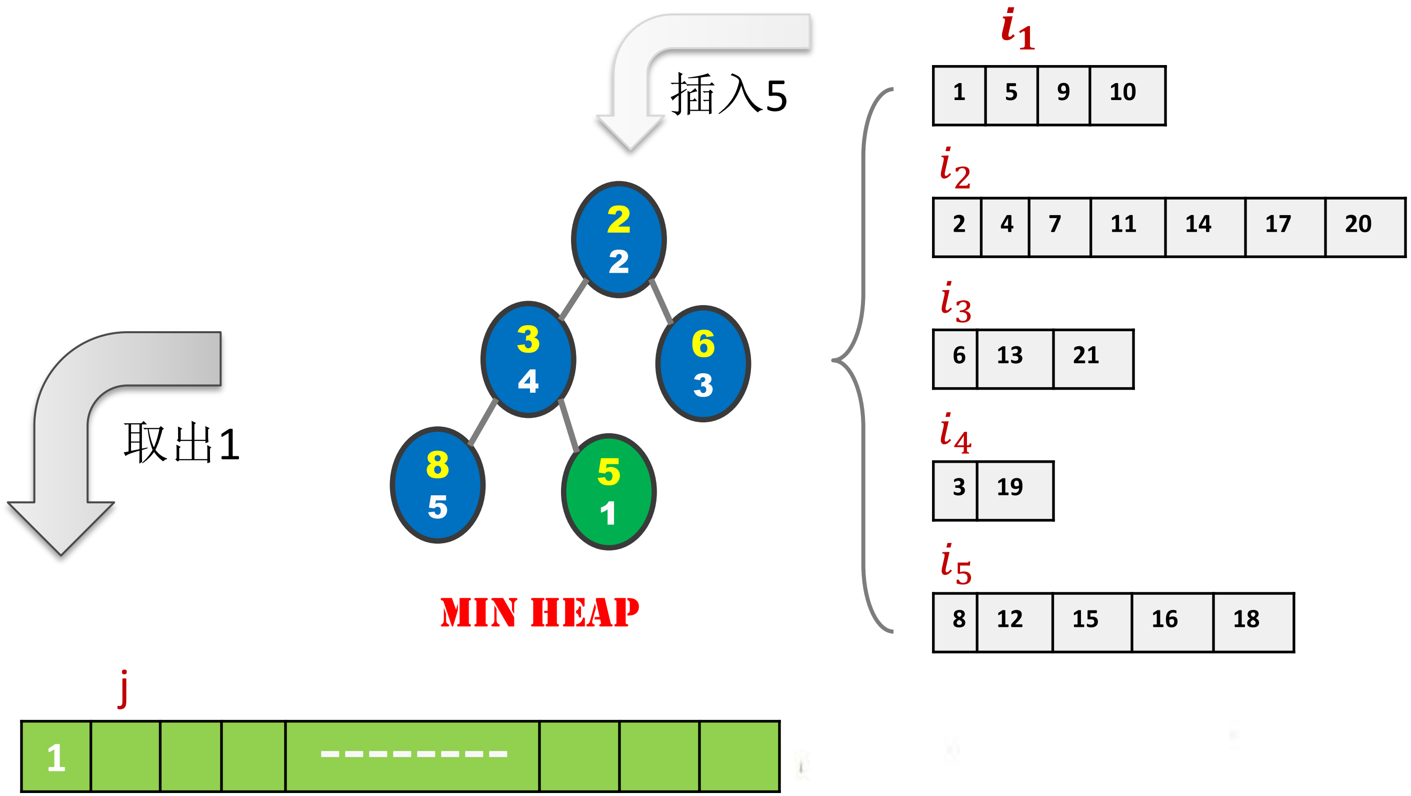


用指针i_x指向数组A_x 中最小值位置$(1 \le x \le n)$

① 取出堆顶元素$\langle A_y[i_y], y\rangle$并将$A_y[i_y]$放入合并后的数组
② 如果$i_y < |A_y|$，插入$\langle A_y[i_y+1], y\rangle$ 至堆，指针$i_y = i_y + 1$
③ 重复上述处理，直到合并完所有元素（堆变空！）

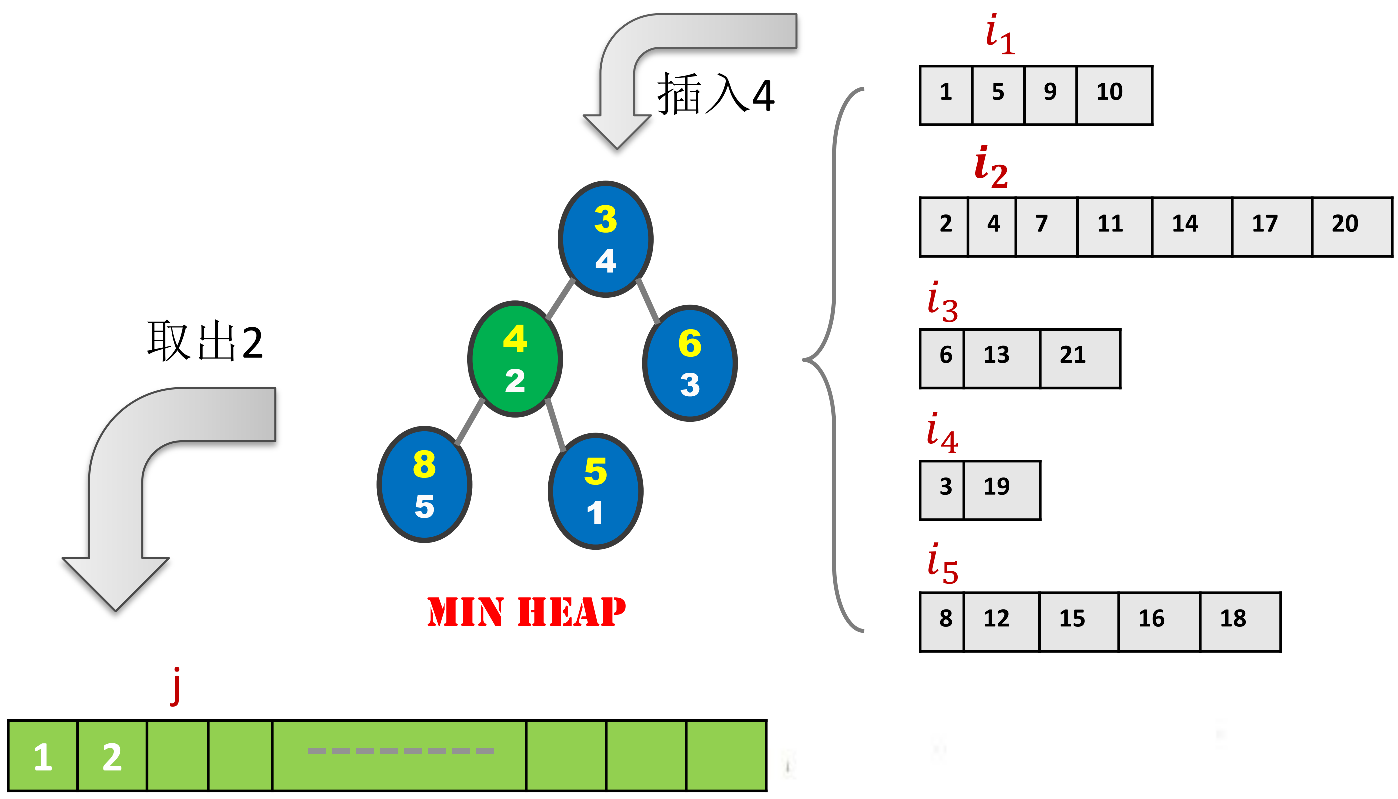
合并n个升序或降序序列

（方法2）同步合并 with heap!



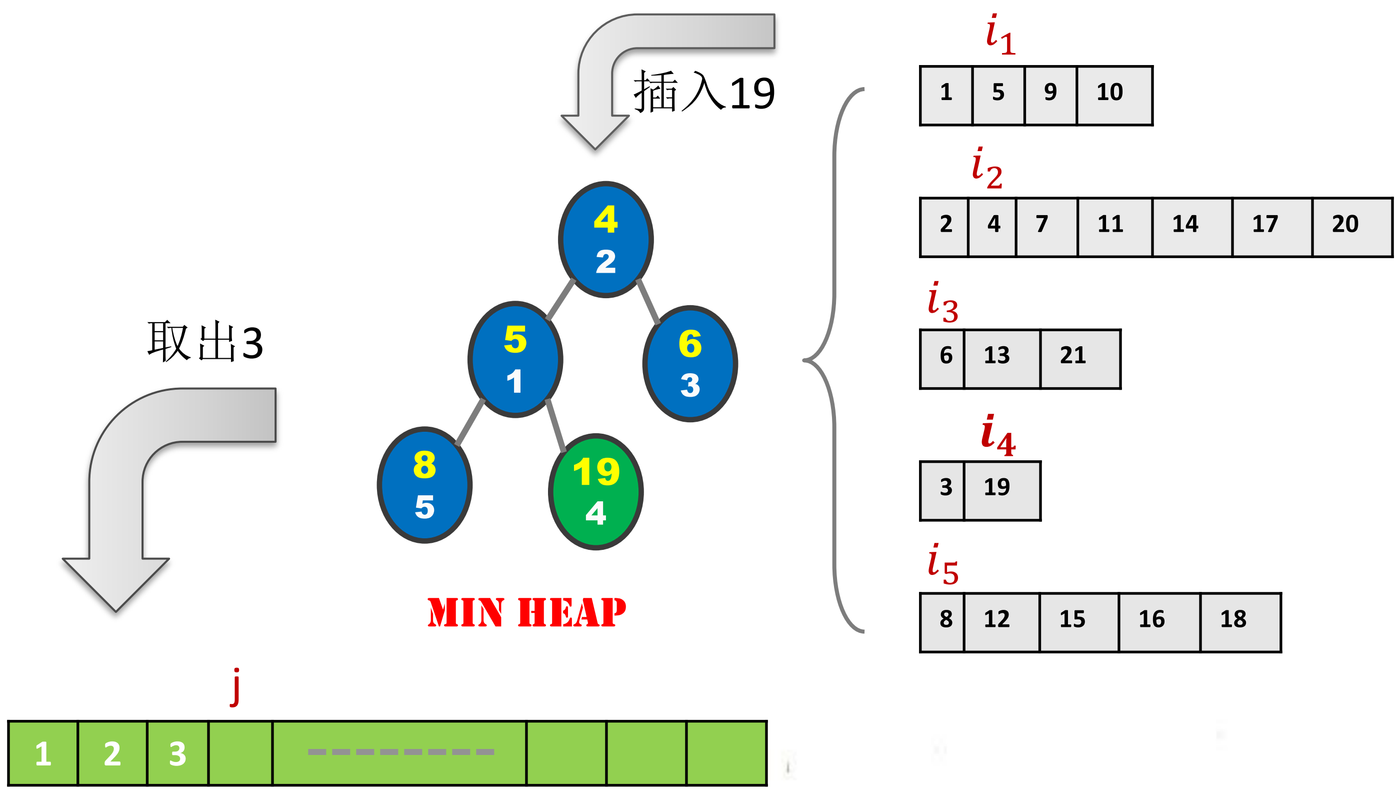
合并n个升序或降序序列

（方法2）同步合并 with heap!



合并n个升序或降序序列

（方法2）同步合并 with heap!



合并n个升序或降序序列

（方法2）同步合并 with heap!

- 用指针i_x指向数组A_x $(1 \le x \le n)$中最小值位置（$i_x = 1$）

- 设计存放所有$\langle A_x[i_x], x\rangle$的最小堆 $(1 \le x \le n)$

- 取出堆顶元素$\langle A_y[i_y], y\rangle$并放入合并后的数组；如果$i_y < |A_y|$，插入$\langle A_y[i_y+1], y\rangle$ 至堆，指针$i_y = i_y + 1$。重复该处理，直到合并完所有元素

$$\mathrm{T}\left(\sum_{1\le i\le n} |A_i|\right) = \left(\sum_{i=1}^{n} |A_i|\right) * \log(n)$$

12.3 Heap Sort

Heapsort

```
template <typename E, typename Comp>
void heapsort(E A[], int n) { // Heapsort
  E maxval;
  heap<E,Comp> H(A, n, n);     // Build the heap
  for (int i=0; i<n; i++)         // Now sort
    maxval = H.removefirst();     // Place maxval at end
}
```

Cost of heapsort: $\Theta(n \log n)$

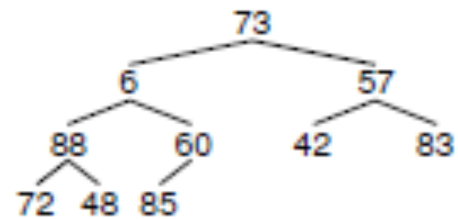
in the worst, average, and best cases.

1. **building the heap takes O(n) time**
2. **n deletions of the maximum element each take (log n) time**

Heapsort: example

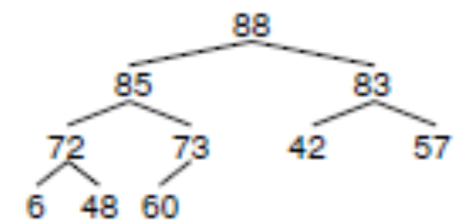
Original Numbers

73	6	57	88	60	42	83	72	48	85



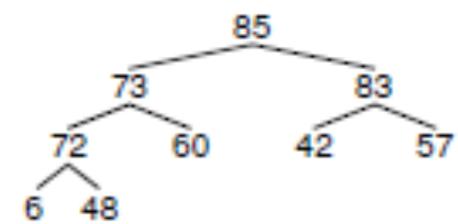
Build Heap

88	85	83	72	73	42	57	6	48	60



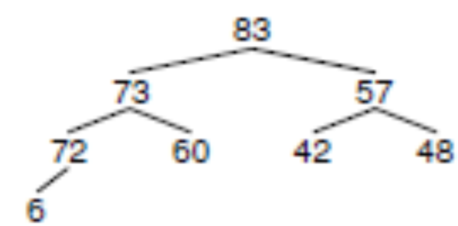
Remove 88

85	73	83	72	60	42	57	6	48	88



Remove 85

83	73	57	72	60	42	48	6	85	88



12.4 Comparison of Sorting Algorithms

Comparison of Running Time

Sorting Algorithms	Average	Best	Worst
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Shellsort	$O(n^{1.5})$	$\Theta(n \log n)$	$\Theta(n^2)$
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$ or $\Theta(n)$	$\Theta(n^2)$
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Radixsort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Comparison of Space

Sorting Algorithms	Auxiliary space
Insertionsort	$O(1)$
Shellsort	$O(1)$
Bubblesort	$O(1)$
Quicksort	$O(\log n)$ ~$O(n)$
Selectionsort	$O(1)$
Heapsort	$O(1)$
Mergesort	$O(n)$
Radixsort	O(n)

Comparison of Stability

（1）Stable Algorithms

- Insertion sort
- Bubble sort
- Selection sort
- Merge sort
- Radix sort

（2）Unstable Algorithms:

- Shell sort
- Quick sort
- Heap sort

数据结构与算法课程组
重庆大学计算机学院

End of Section.