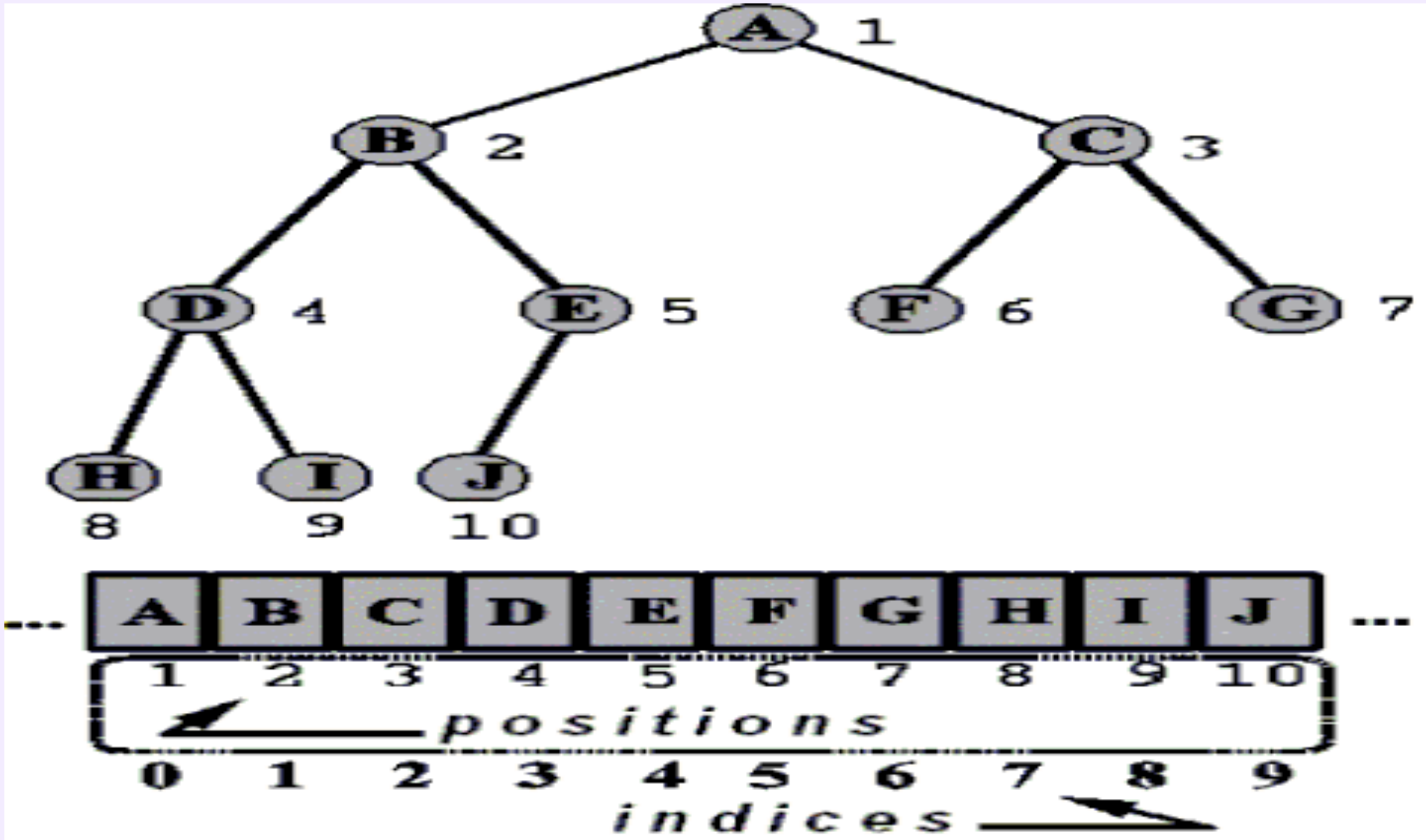
Algorithm HeapSort

- J. W. J. Williams (1964): a special binary tree called **heap** to obtain an $O(n \log n)$ worst-case sorting
- Basic steps:
  - Convert an array into a heap in linear time $O(n)$
  - Sort the heap in $O(n \log n)$ time by deleting n times the maximum item because each deletion takes the logarithmic time $O(\log n)$

Complete Binary Tree: linear array representation



Complete Binary Tree

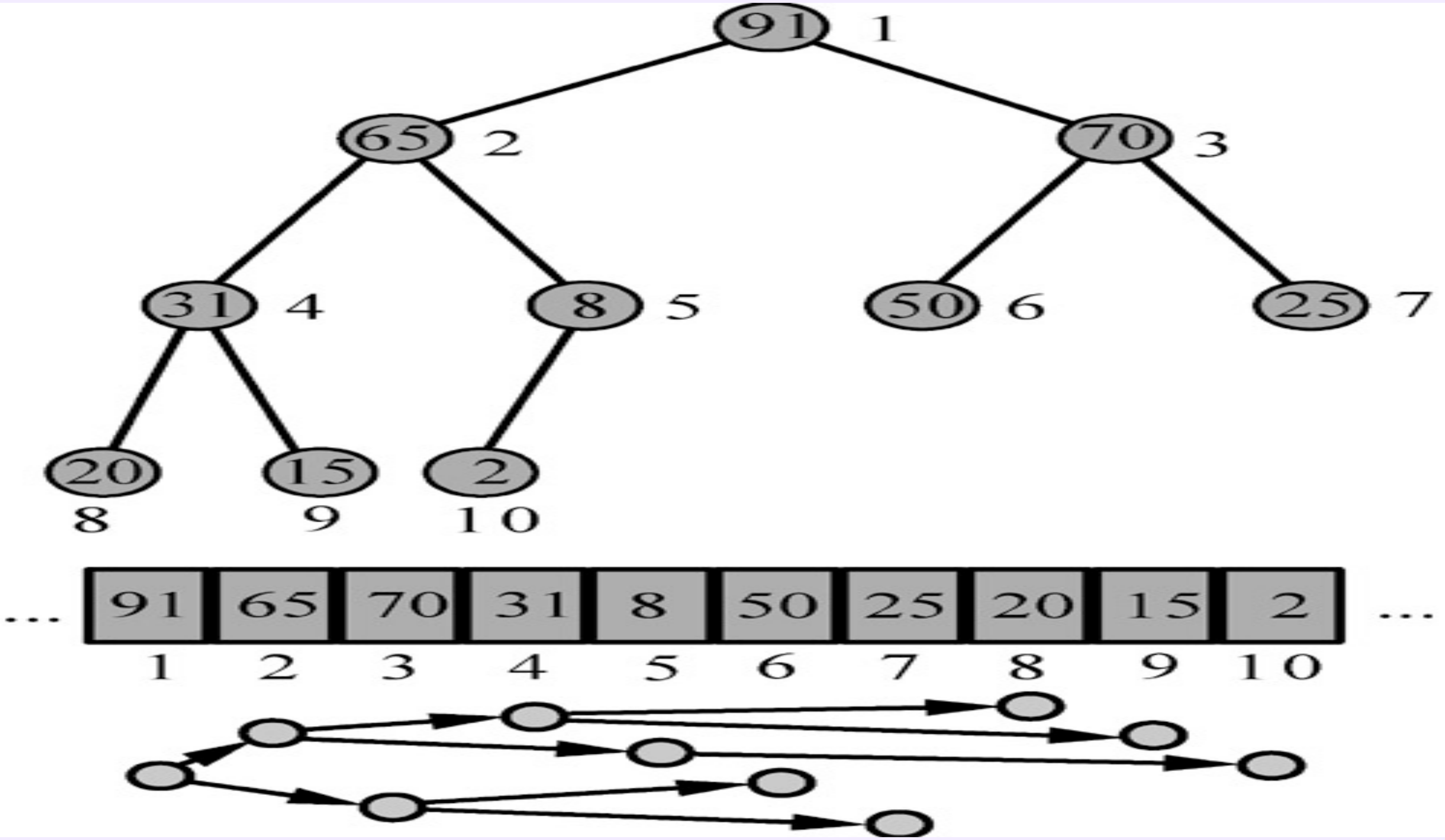
- A complete binary tree of the height h contains between 2^h and $2^{h+1}-1$ nodes
- A complete binary tree with the n nodes has the height $\lfloor \log_2 n \rfloor$
- Node positions are specified by the level-order traversal (the root position is 1)
- If the node is in the position p then:
  - the parent node is in the position $\lfloor p/2 \rfloor$
  - the left child is in the position $2p$
  - the right child is in the position $2p + 1$

Binary Heap

- A **heap** consists of a complete binary tree of height h with numerical keys in the nodes
- The defining feature of a heap:

  the key of each parent node is **greater than** or **equal to** the key of any child node
- The **root** of the heap has **the maximum key**

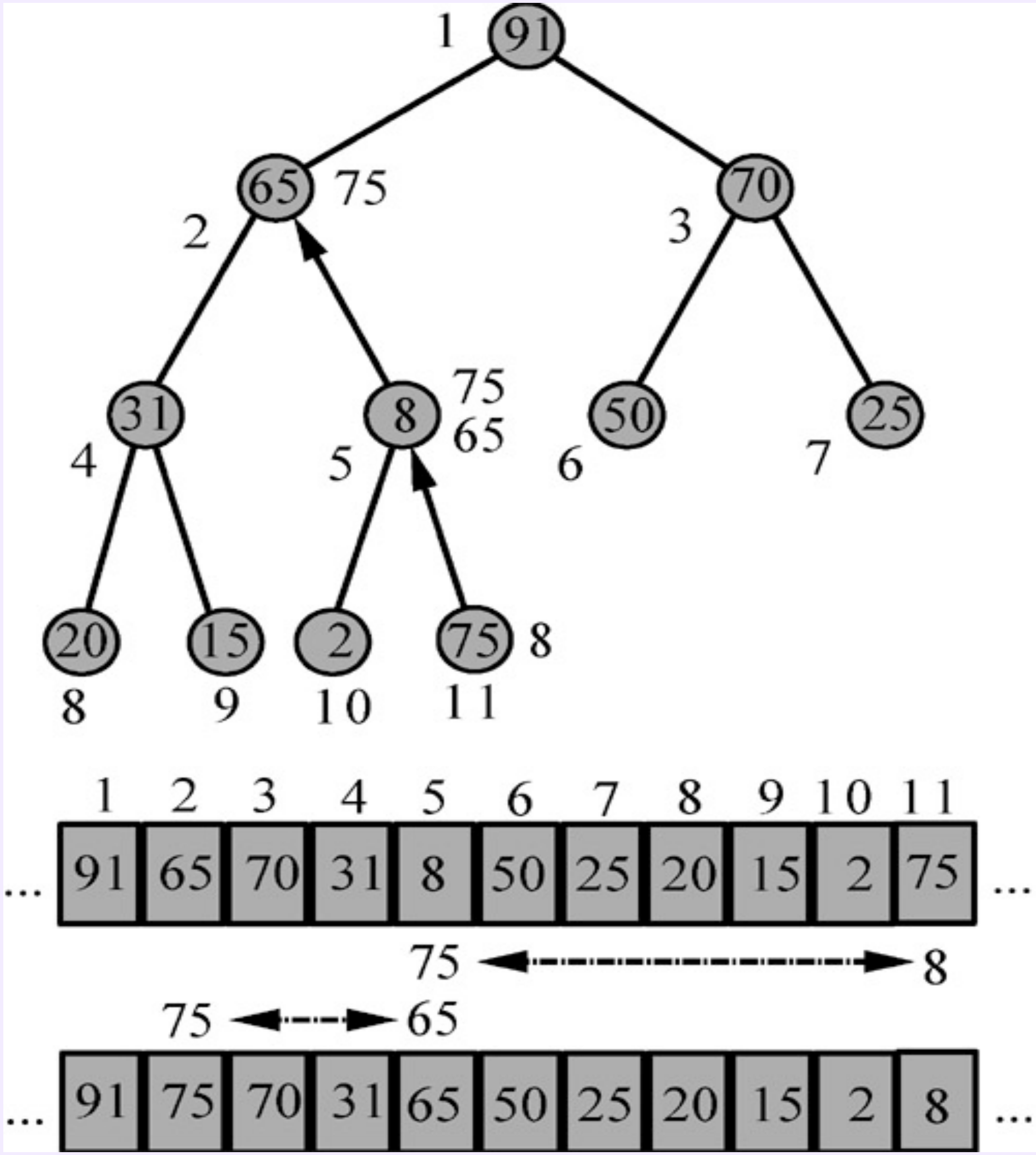
Binary Heap: linear array representation



Binary Heap: insert a new key

- **Heap** of k keys → into a heap of $k+1$ keys
- Logarithmic time $O(\log k)$ to insert a new key:
  - Create a new leaf position $k+1$ in the heap
  - **Bubble** (or **percolate**) the new key up by swapping it with the parent if the parent key is smaller than the new key

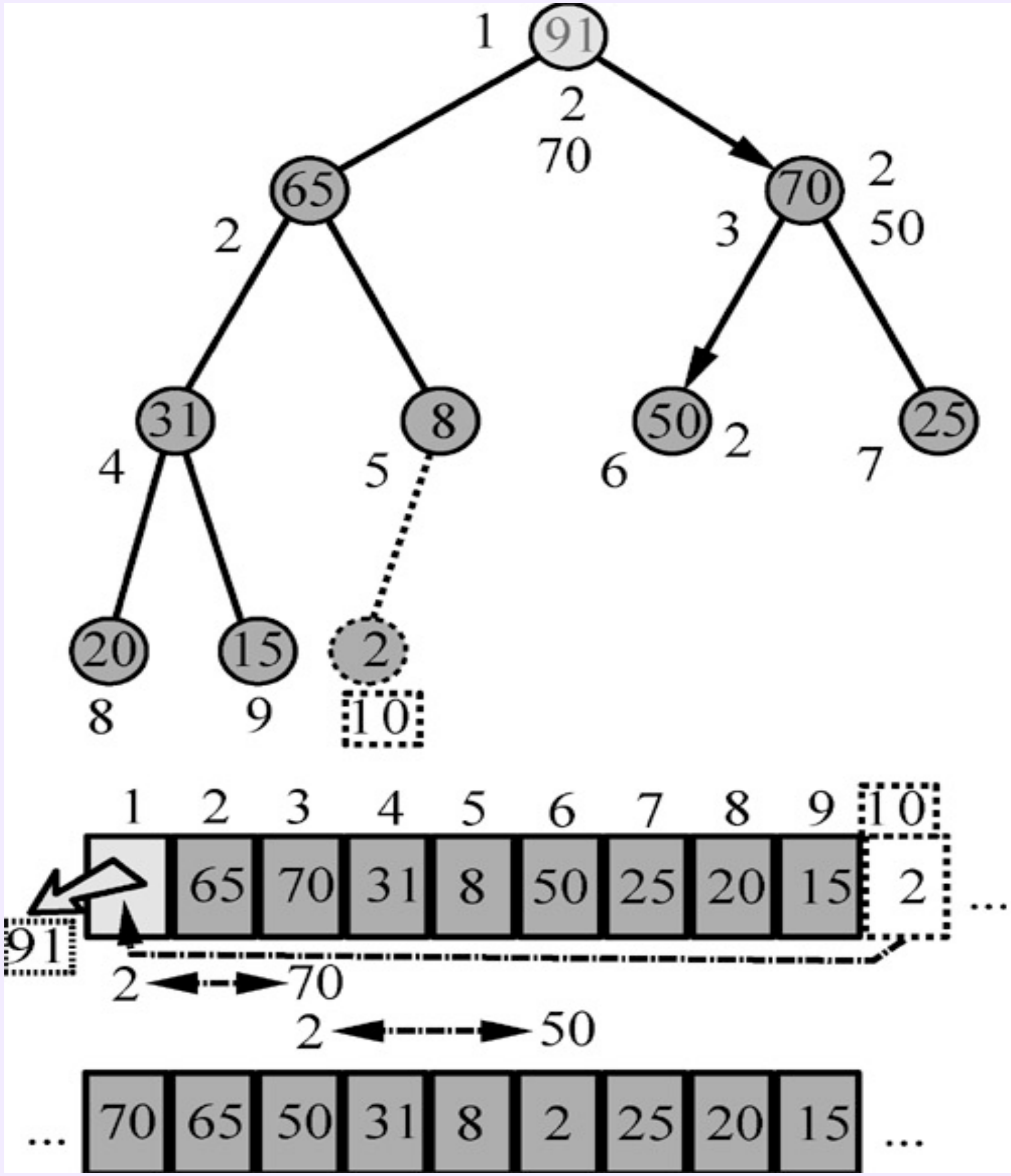
Binary Heap: an example of inserting a key



Binary Heap: delete the maximum key

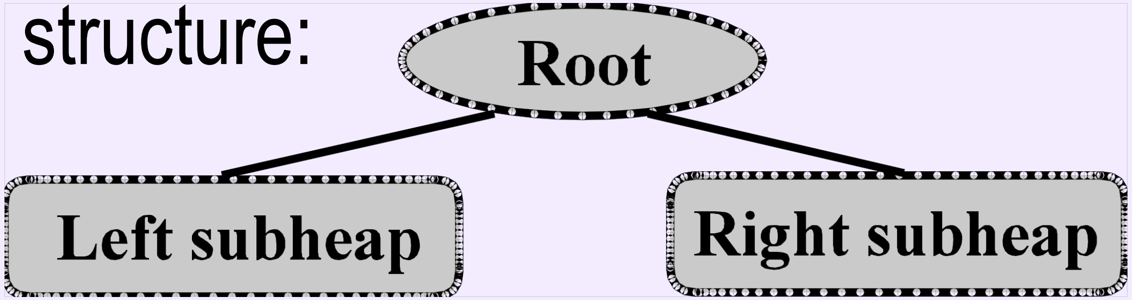
- Heap of k keys → into a heap of $k-1$ keys
- Logarithmic time $O(\log k)$ to delete the root (or maximum) key:
  - Remove the root key
  - Delete the leaf position k and move its key into the root
  - **Bubble** (**percolate**) the root key down by swapping it with the largest child if that child is greater

Binary Heap: an example of deleting the maximum key



Linear Time Heap Construction

- Do not use n insertions → $O(n \log n)$ time!
- Alternative $O(n)$ procedure uses a recursively defined heap structure:



  - form recursively the left and right subheaps
  - percolate the root down to establish the heap order everywhere

Non-recursive Heap Building

- Nodes percolate down in **reverse level order**
  - Each node p is processed after its descendants have been already processed
  - Leaves need not be percolated down
- Worst-case time $T(h)$ to build a heap of height h:

$$T(h) = 2T(h-1) + ch \rightarrow T(h) = O(2^h)$$

  - Form two subheaps of height at most $h - 1$
  - Percolate the root down a path of length at most h

Time to Build a Heap

$$T(h) = 2T(h-1) + ch$$

$$2T(h-1) = 2^2 T(h-2) + 2c(h-1)$$

$$\dots \quad \dots \quad \dots$$

$$2^{h-2}T(2) = 2^{h-1}T(1) + 2^{h-2}c \cdot 2$$

$$2^{h-1}T(1) = 2^h T(0) + 2^{h-1}c \cdot 1 = 2^{h-1}c \cdot 1$$

$$T(h) = c \cdot \left(1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + \dots + (h-2) \cdot 2^2 + (h-1) \cdot 2^1 + h \cdot 2^0\right)$$

$$= c \cdot \left(2^{h+1} - h - 1\right)$$

Worst-case Time Complexity

- A heap of n nodes is of height $h = \lfloor \log_2 n \rfloor$ so that $2^h \leq n \leq 2^{h+1}-1$
- Therefore, the time for converting an array into a heap is linear: $T(h) = O(2^h)$, or $T(n) = O(n)$
- To sort a heap, the maximum element is deleted n times, so that the worst-case time complexity of HeapSort is $O(\ n \log n\ )$
  - Each deletion takes logarithmic time $O(\ \log n)$

Steps of HeapSort

$p/_i$	$1/_0$	$2/_1$	$3/_2$	$4/_3$	$5/_4$	$6/_5$	$7/_6$	$8/_7$	$9/_8$	$10/_9$
a	70	65	50	20	2	91	25	31	15	8
HEAPIFY					8					2
				31				20		
			91			50				
	91		70							
h	**91**	**65**	**70**	**31**	**8**	**50**	**25**	**20**	**15**	**2**

Steps of HeapSort

a_1	2	65	70	31	8	50	25	20	15	**91**
Restore the heap (R.h.)	70		2							
			50			2				
H_9	70	65	50	31	8	2	25	20	15	
a_2	15	65	50	31	8	2	25	20	**70**	**91**
R.h.	65	15								
		31		15						
				20				15		
h_8	65	31	50	20	8	2	25	15		

Steps of HeapSort

a_3	15	31	50	20	8	2	25	**65**	**70**	**91**
R.h.	50		15							
			25				15			
h_7	50	31	25	20	8	2	15			
a_4	15	31	25	20	8	2	**50**	**65**	**70**	**91**
R.h.	31	15								
		20		15						
h_6	31	20	25	15	8	2				

Steps of HeapSort

a_5	2	20	25	15	8	**31**	**50**	**65**	**70**	**91**
R. h.	25		2							
h_5	25	20	2	15	8					
a_6	8	20	2	15	**25**	**31**	**50**	**65**	**70**	**91**
R. h.	20	8								
		15		8						
h_4	20	15	2	8						

Steps of HeapSort

a_7	8	15	2	**20**	**25**	**31**	**50**	**65**	**70**	**91**
R. h.	15	8								
h_3	15	8	2							
a_8	2	8	**15**	**20**	**25**	**31**	**50**	**65**	**70**	**91**
R. h.	8	2								
h_2	8	2								
a_9	2	**8**	**15**	**20**	**25**	**31**	**50**	**65**	**70**	**91**

sorted array