Sorting III: Heapsort

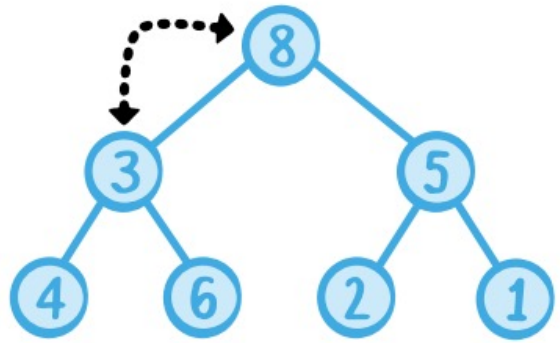
Instructor: Meng-Fen Chiang

COMPCSI220: WEEK 9

Slides adapted from Mark Wilson, Georgy Gimel'farb, Simone Linz and Tanya Gvozdeva

OUTLINE

- Heapsort
  - Algorithms
  - Illustrating Example
- Time Complexity Analysis



Heapsort

Given an array:

1. Build the heap
2. Delete the maximum repeatedly (arranging the elements in the output list in reverse order of deletion) until the heap is empty.

Step1: Building a Heap

- A straightforward method: sequentially insert nodes to the heap and maintain the heap property
- Inserting a node to a heap of height h, we need $h = \lfloor \log_2 n \rfloor$ comparisons at most. Then the worst-case running time complexity of building a heap of size n is:

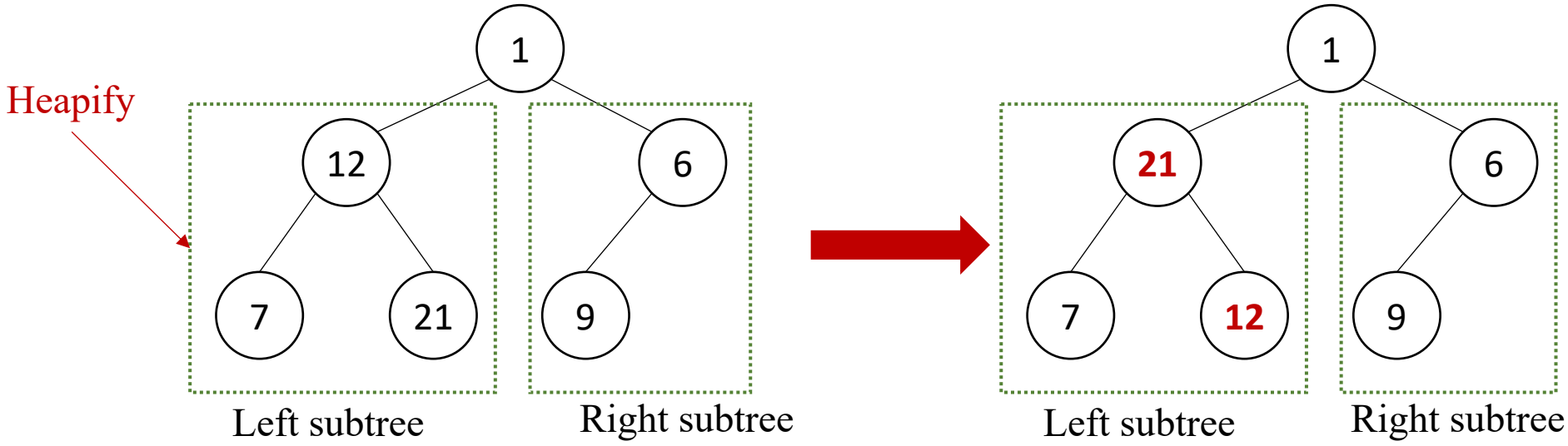
$$T(n) = \lfloor \log_2 1 \rfloor + \lfloor \log_2 2 \rfloor + \lfloor \log_2 3 \rfloor + \cdots + \lfloor \log_2 n \rfloor$$
$$\leq \log_2 n + \log_2 n + \log_2 n + \cdots + \log_2 n = n \log_2 n$$

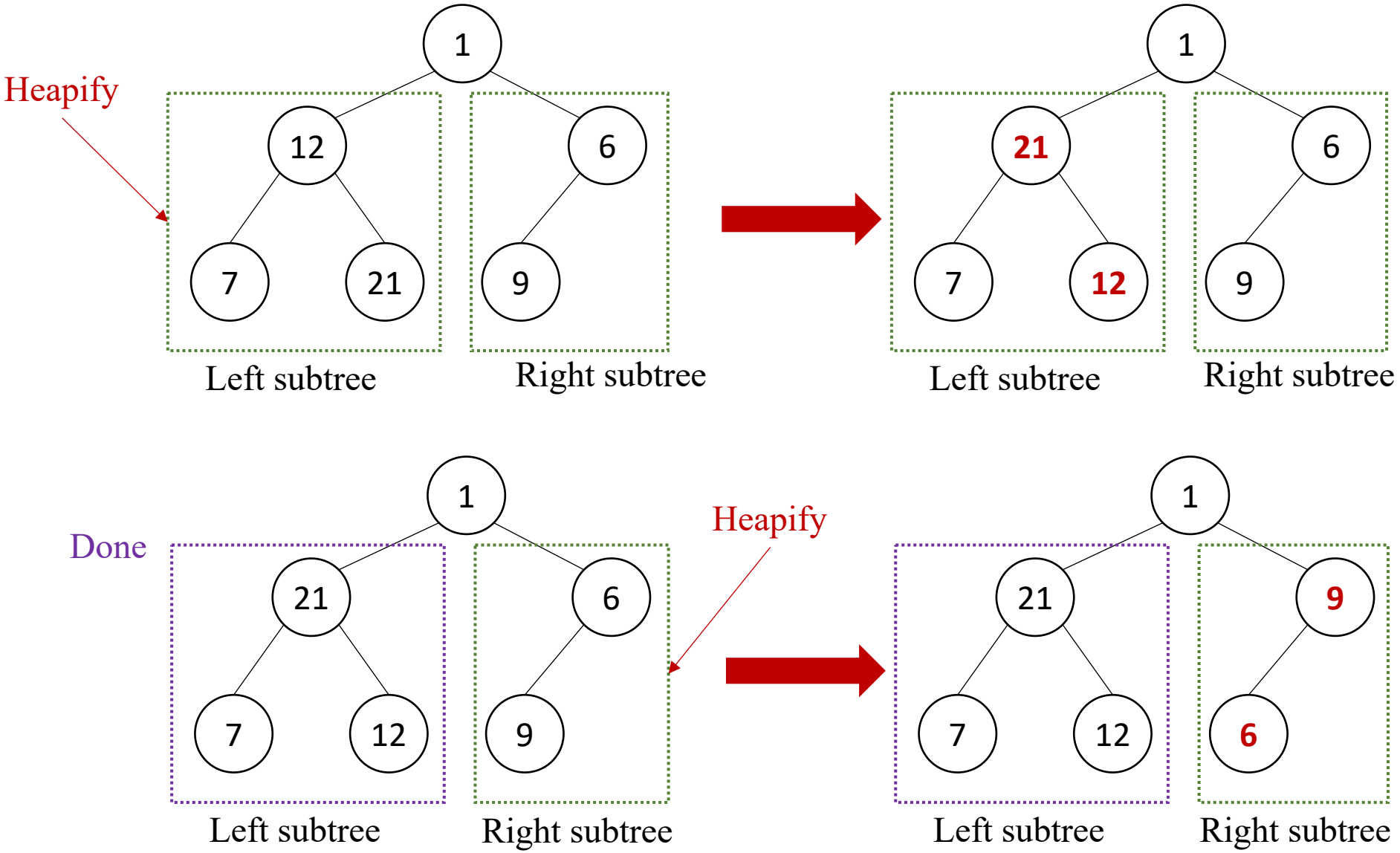
$$T(n) = \lfloor \log_2 1 \rfloor + \lfloor \log_2 2 \rfloor + \cdots + \left\lfloor \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rfloor + \left\lfloor \log_2 \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right) \right\rfloor + \cdots + \lfloor \log_2 n \rfloor$$
$$\geq \left\lfloor \log_2 \left(\left\lfloor \frac{n}{2} \right\rfloor \right) \right\rfloor + \left\lfloor \log_2 \left(\left\lfloor \frac{n}{2} \right\rfloor \right) \right\rfloor + \cdots + \left\lfloor \log_2 \left(\left\lfloor \frac{n}{2} \right\rfloor \right) \right\rfloor$$
$$= \left\lceil \frac{n}{2} \right\rceil \left\lfloor \log_2 \left(\left\lfloor \frac{n}{2} \right\rfloor \right) \right\rfloor = \left\lceil \frac{n}{2} \right\rceil \left(\left\lceil \log_2 \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right) \right\rceil - 1 \right) \geq \frac{n}{2} \log_2 \left(\left\lfloor \frac{n+2}{2} \right\rfloor \right) \geq \frac{n}{2} \log_2 \left(\frac{n}{2} \right)$$

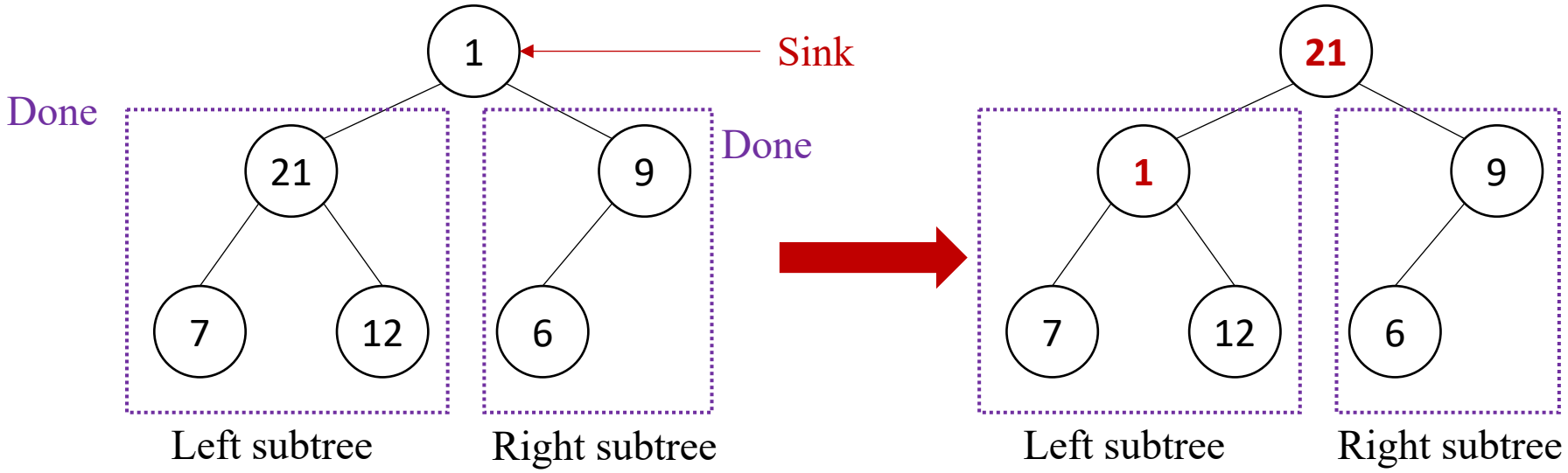
The above gives: $T(n) \in \Theta(n \log n)$

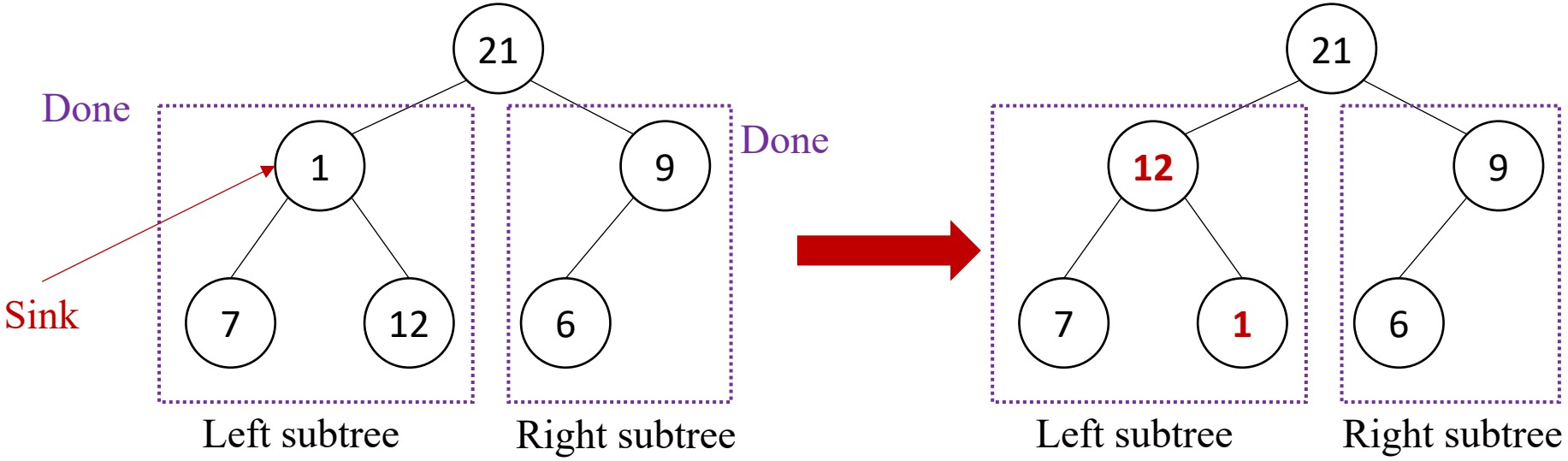
Can we do better?

- **Motivation of an alternative method.** We only need the heap property at the end, no need to keep it whenever an element is inserted.
  1. Build a complete binary tree without keeping the heap property – this is especially easy for array implementation which only needs to insert all elements to a list.
  2. After all elements are inserted, reorganize the positions of the elements to follow heap property, this operation is called heapify – recursively heapify the left and right subtrees, then put the root down to the right position.



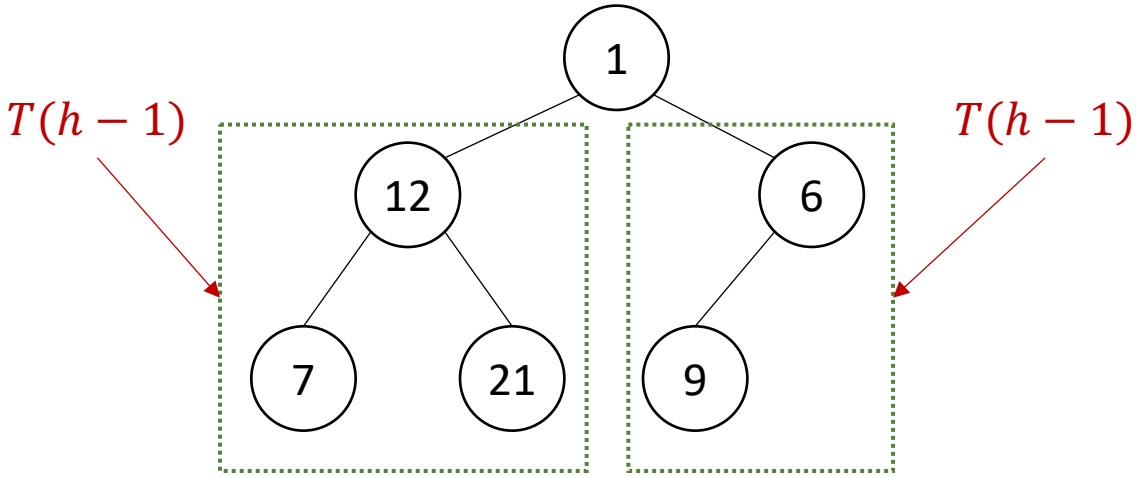

Heapify
1
12
6
7
21
9
Left subtree
Right subtree
1
21
6
7
12
9
Left subtree
Right subtree
Done
Heapify
1
21
6
7
12
9
Left subtree
Right subtree
1
21
9
7
12
6
Left subtree
Right subtree


Done
1
Sink
21
9
7
12
6
Done
Left subtree
Right subtree
21
1
9
7
12
6
Left subtree
Right subtree


Done
21
Sink
1
9
7
12
6
Done
Left subtree
Right subtree
21
12
9
7
1
6
Left subtree
Right subtree

Time Complexity Analysis: Heapifying

- Let $T(h)$ is the running time complexity for building a heap with height at most h. It comprises of
  - Running time of heapifying the two subtrees recursively $T(h-1)$
  - Running time of sinking the root to the right position needs at most $c(h-1)$ comparisons/swaps
  - Thus, $T(h) = 2T(h-1) + c(h-1),\ T(0) = 0$



We can show that $T(h)$ is $\Theta(2^h)$

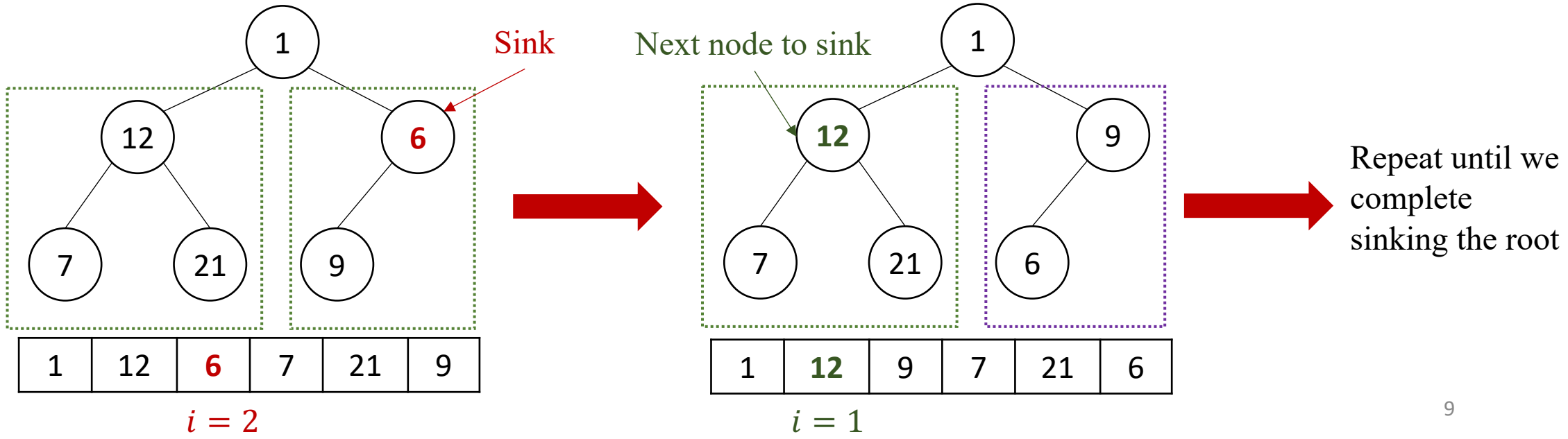
Because $h = \lfloor \log_2 n \rfloor$, then $T(n)$ is $\Theta(n)$

$$T(h) - 2T(h-1) = c(h-1)$$
$$2T(h-1) - 4T(h-2) = 2c(h-2)$$
$$4T(h-2) - 8T(h-3) = 4c(h-3)$$
$$\cdots\ \cdots$$
$$2^{h-1}T(1) - 2^h T(0) = 2^{h-1}c \cdot 0$$

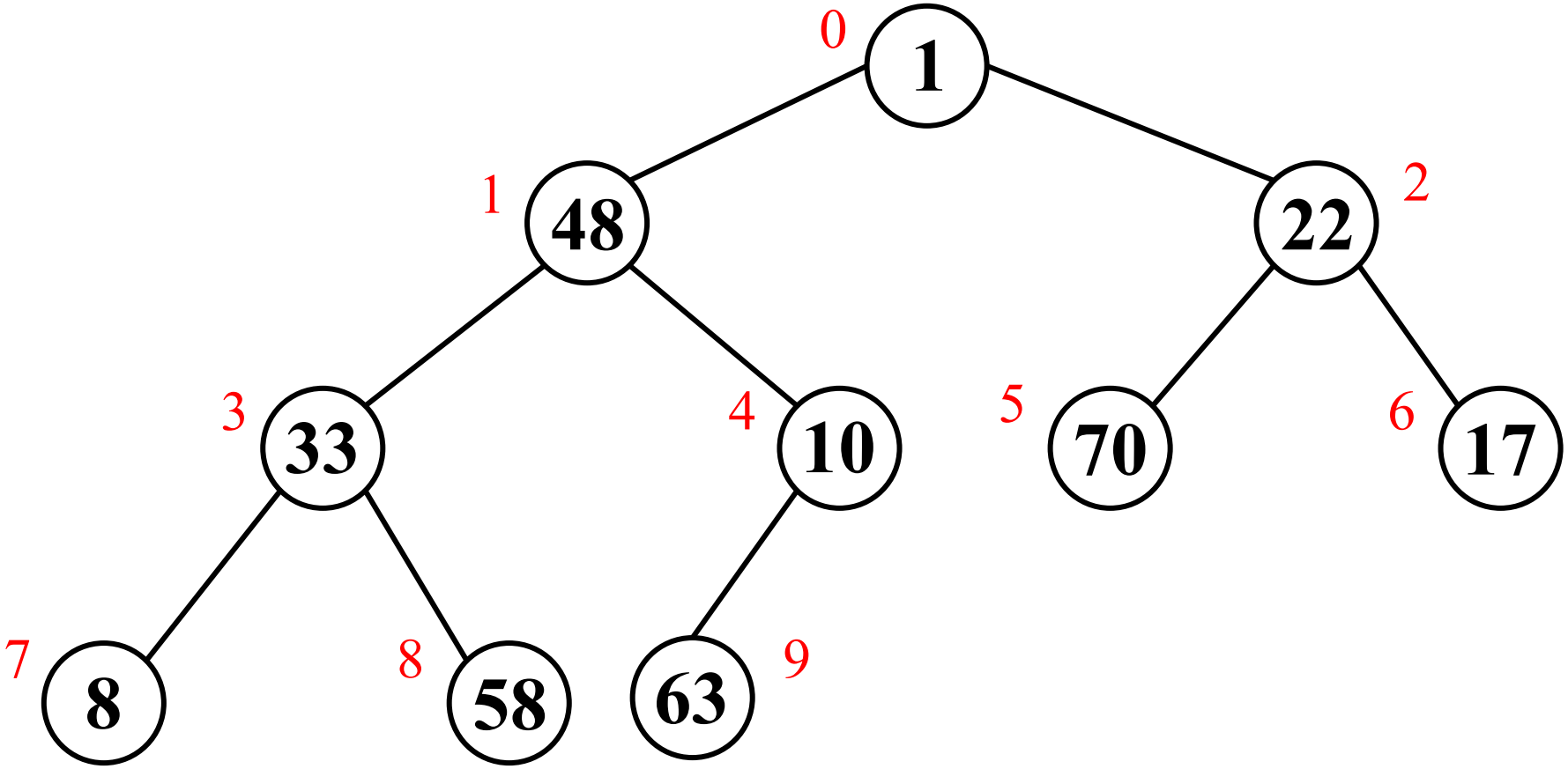
$$T(h) = c \cdot \left(0 + 2^{h-2} \cdot 1 + 2^{h-3} \cdot 2 + \cdots + 2^0 \cdot (h-1)\right)$$

Heapifying without Recursions

- Sink the nodes in a bottom-up manner. Consider the array implementation of heap.
  1. Find the lowest non-leaf node, which is at position $\lfloor(n-1)/2\rfloor$. Set the current node index $i=\lfloor(n-1)/2\rfloor$
  2. Sink the current node at position i to the correct position.
  3. Find the next non-leaf node, that is the node at position $i-1$, go to step 2.

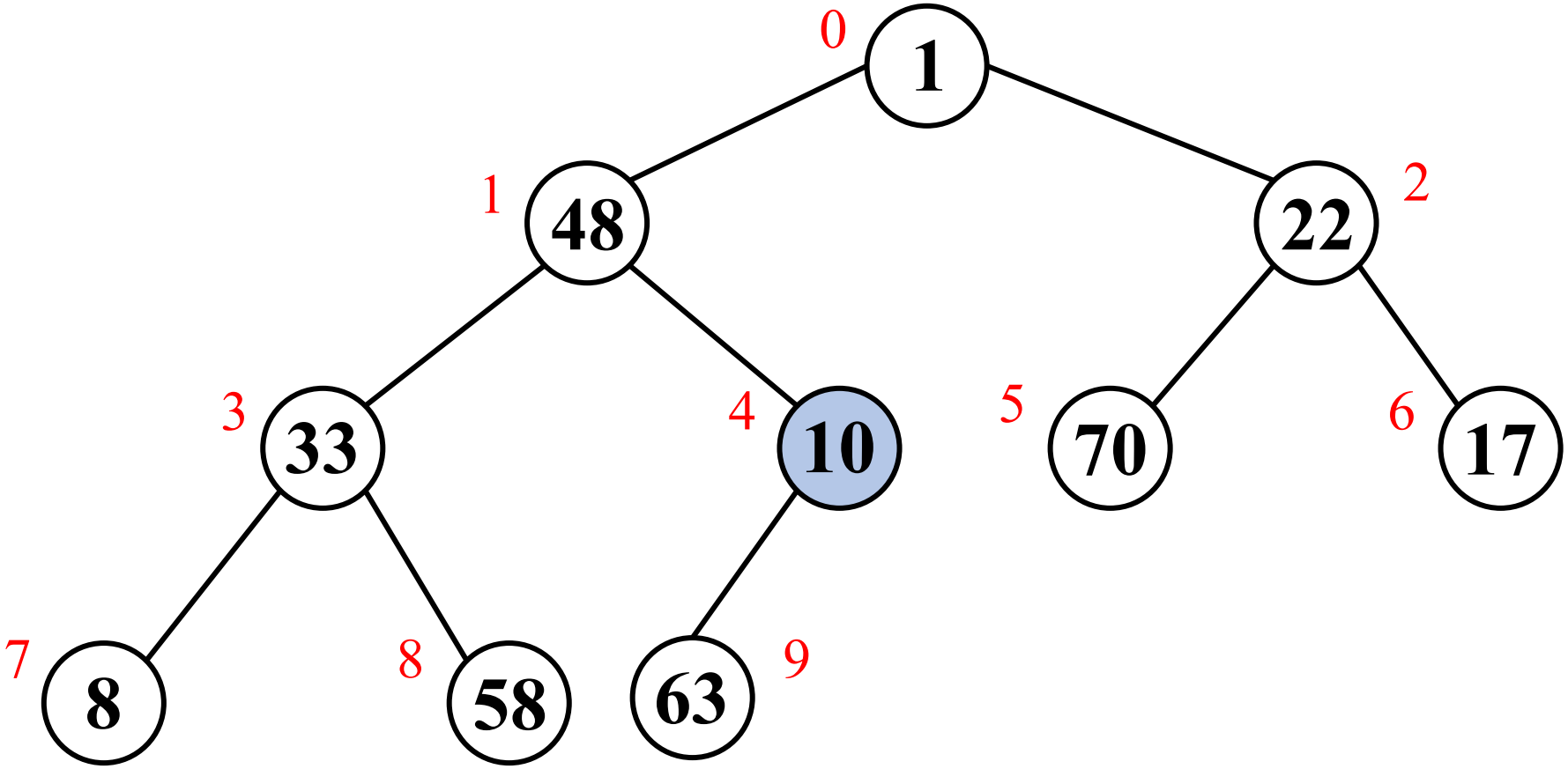


Heapsort algorithms: array implementation



1	48	22	33	10	70	17	8	58	63
0	1	2	3	4	5	6	7	8	9

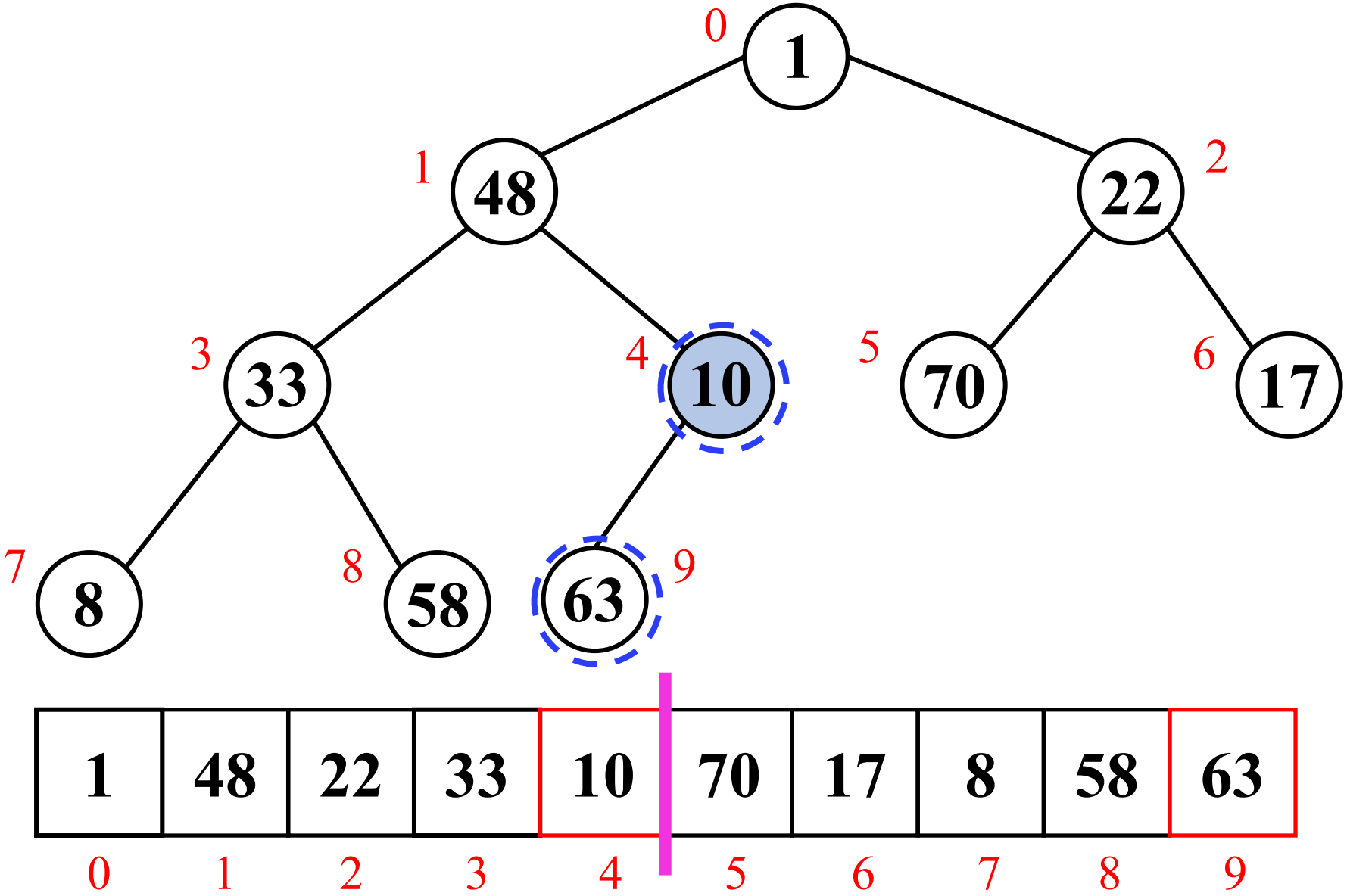
Heapsort algorithms: array implementation



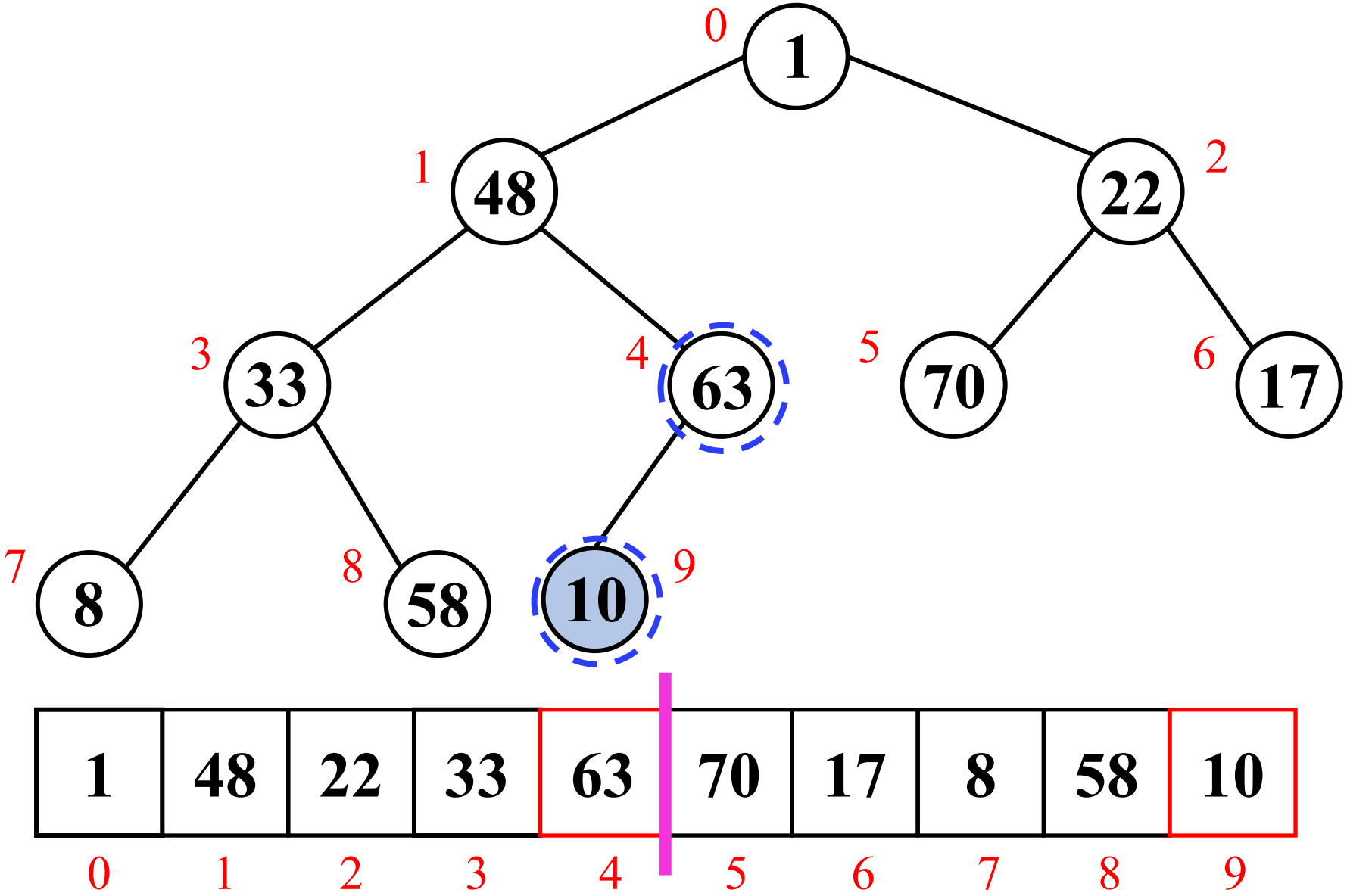
sink 10

1	48	22	33	10	70	17	8	58	63
0	1	2	3	4	5	6	7	8	9

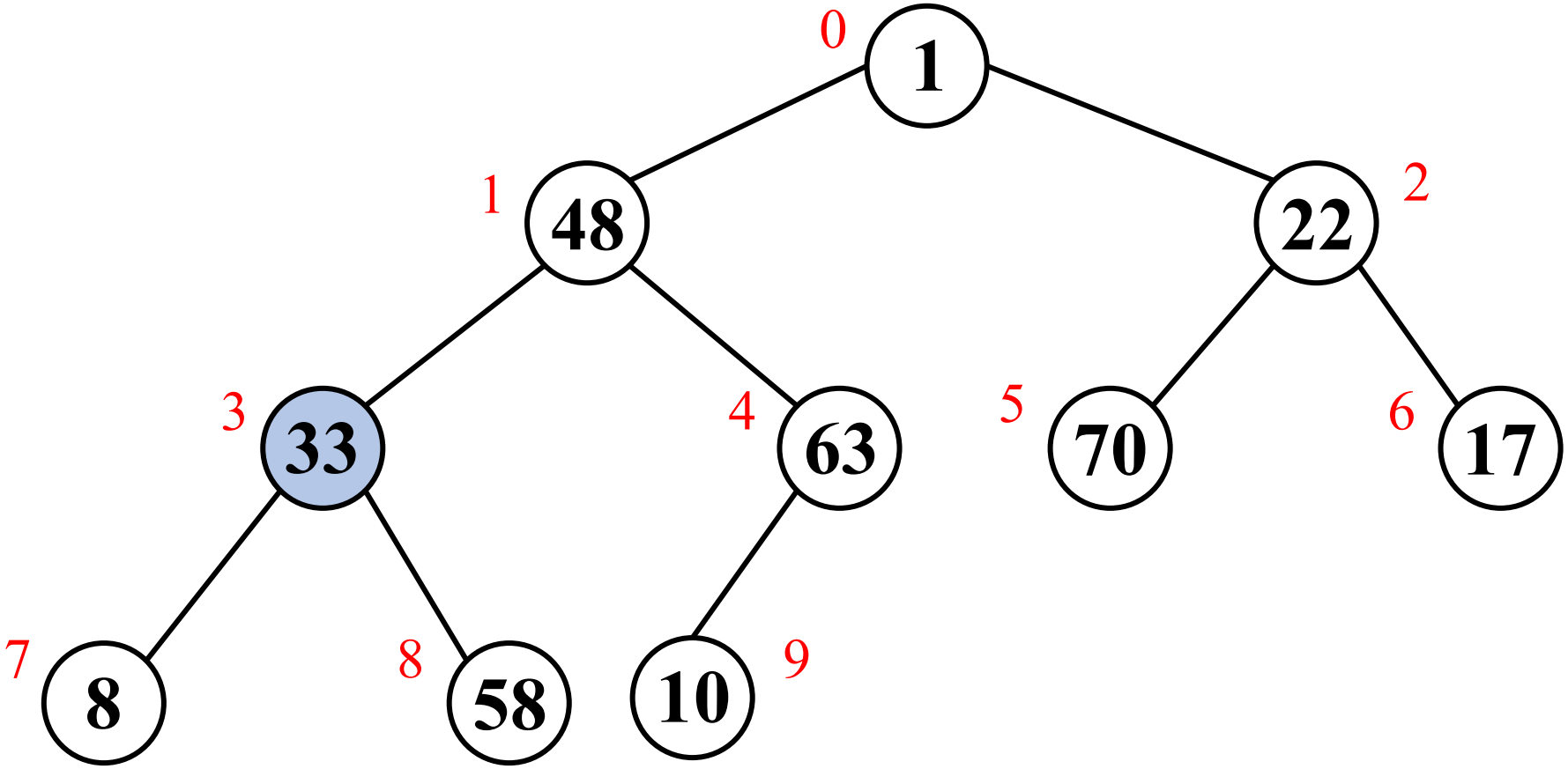
Heapsort algorithms: array implementation



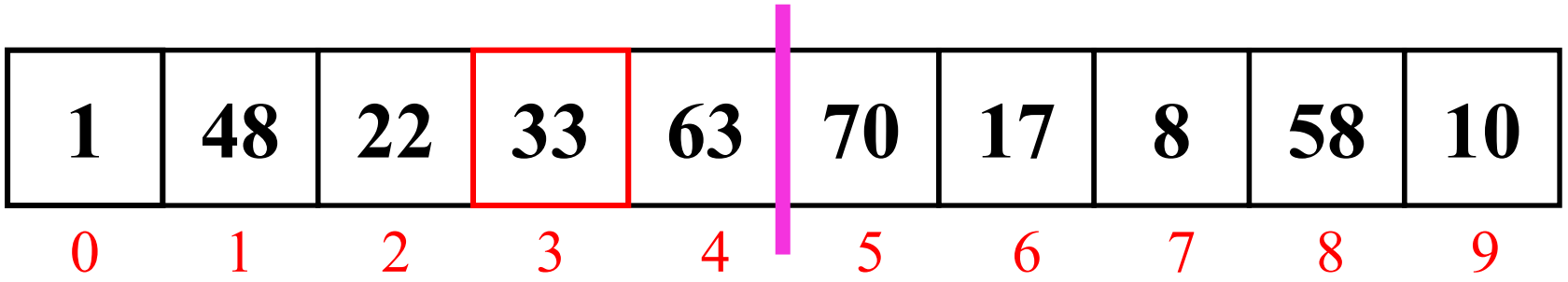
Heapsort algorithms: array implementation



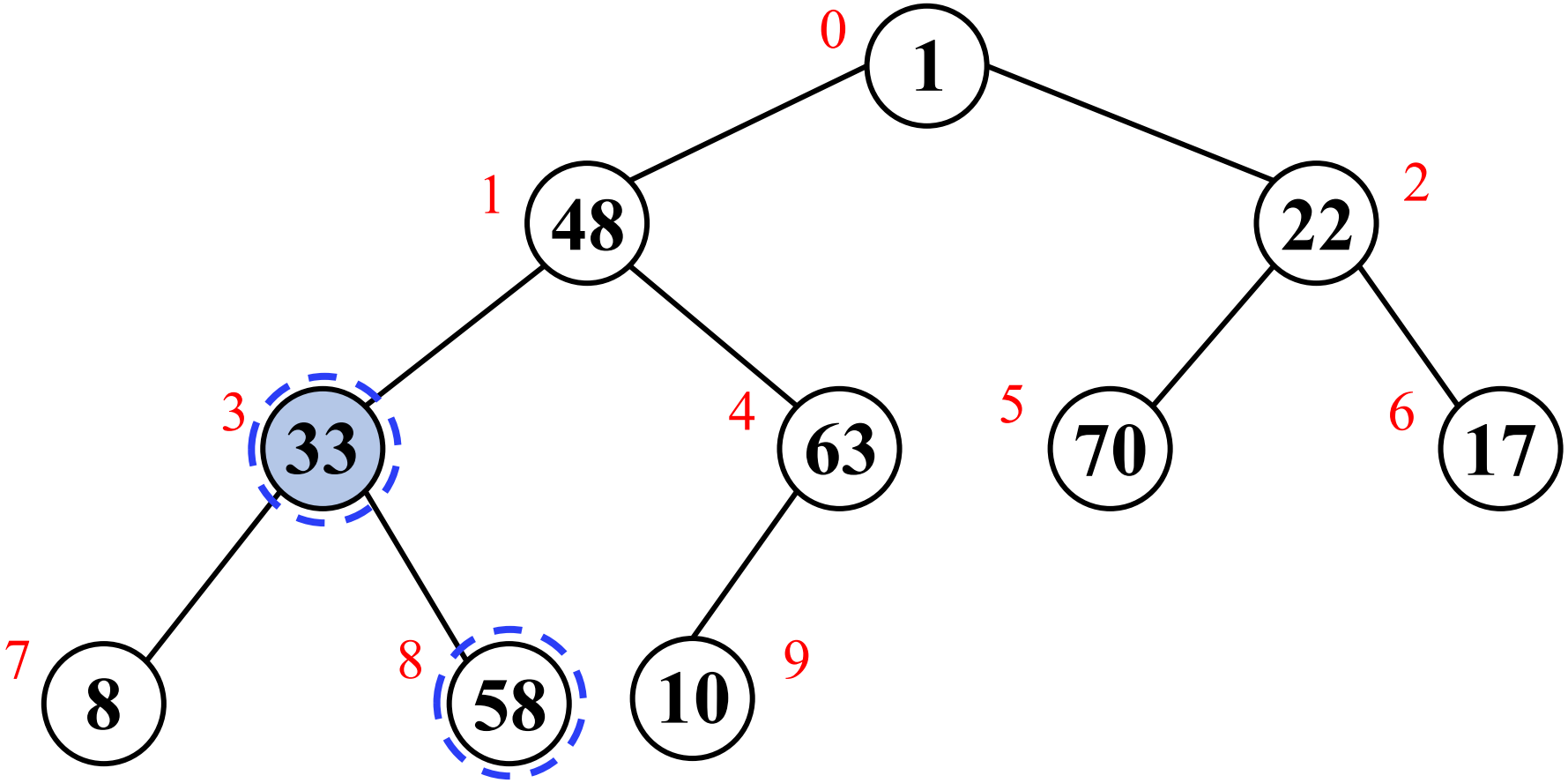
Heapsort algorithms: array implementation



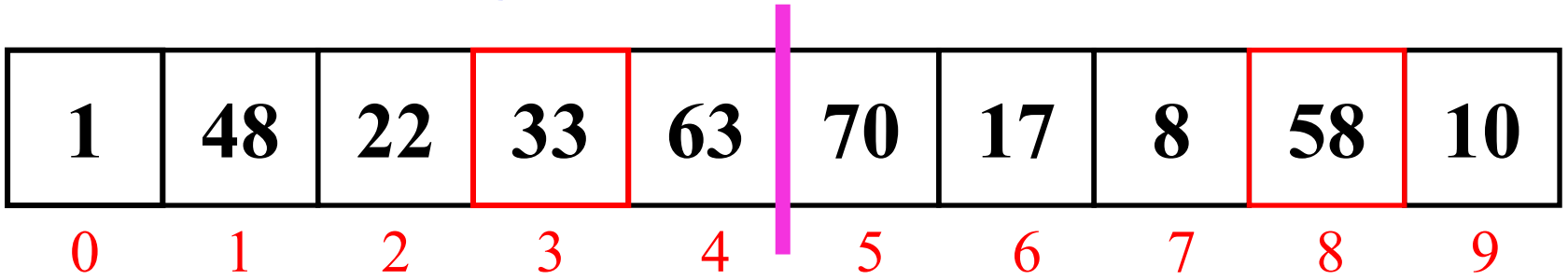
sink 33



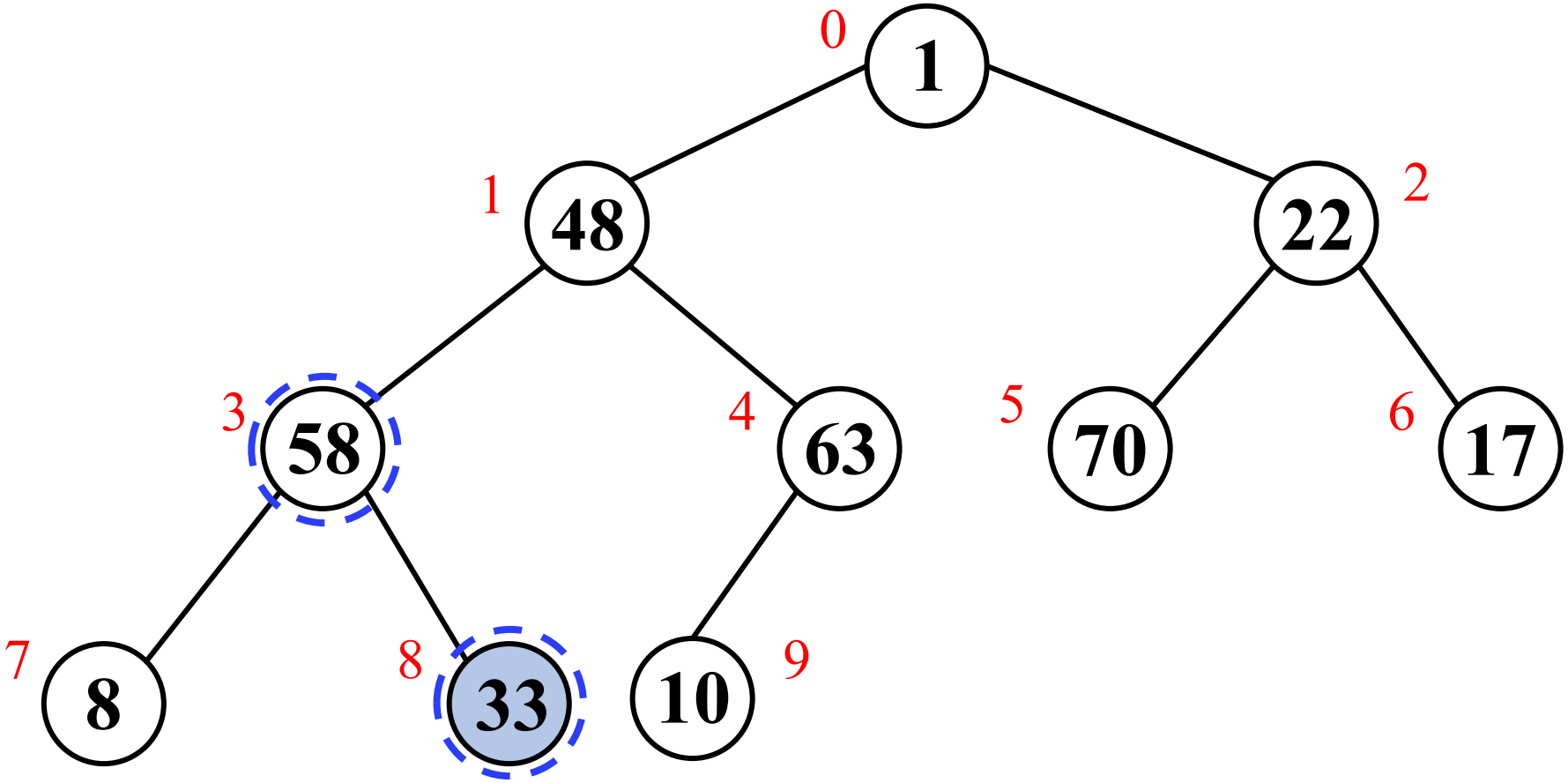
Heapsort algorithms: array implementation



sink 33



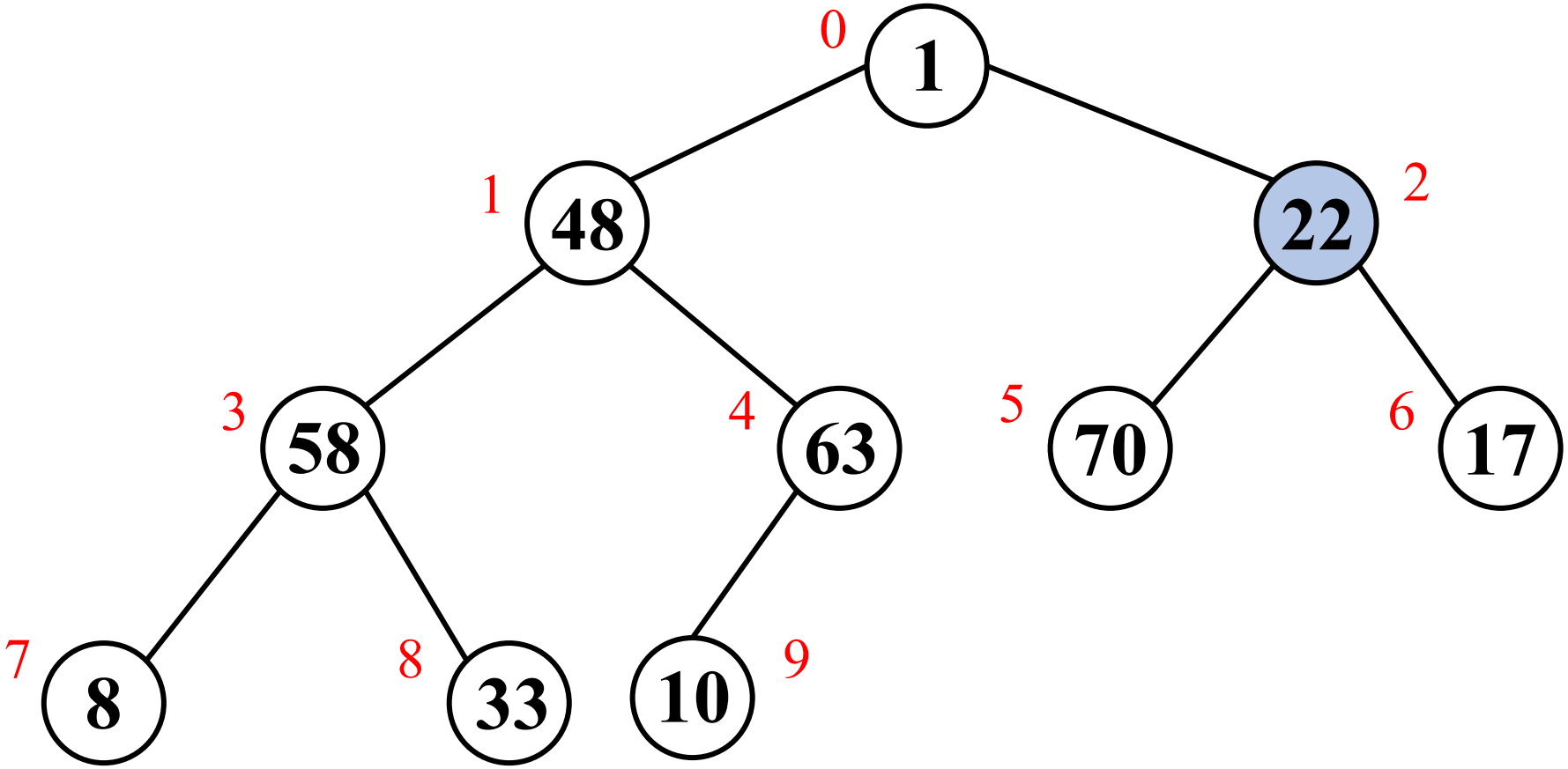
Heapsort algorithms: array implementation



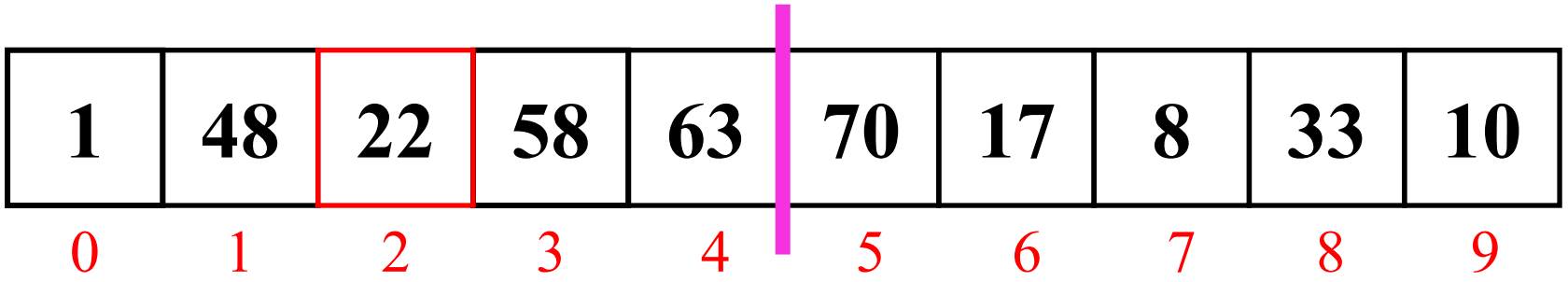
sink 33

1	48	22	58	63	70	17	8	33	10
0	1	2	3	4	5	6	7	8	9

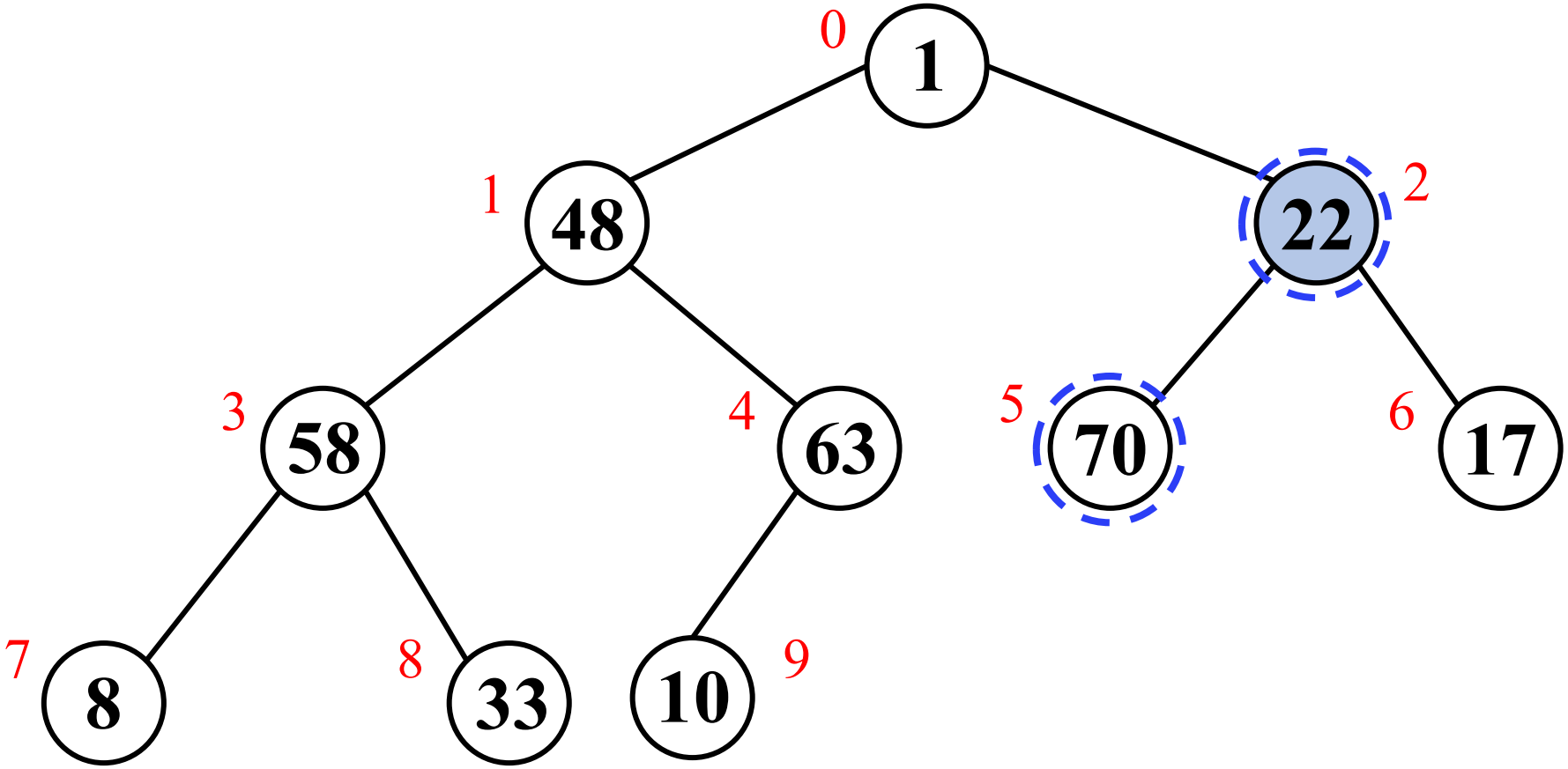
Heapsort algorithms: array implementation



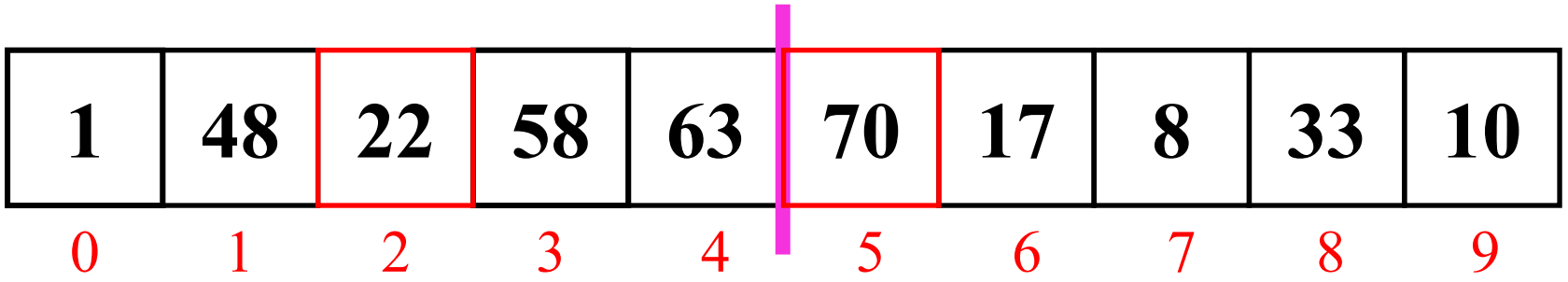
sink 22



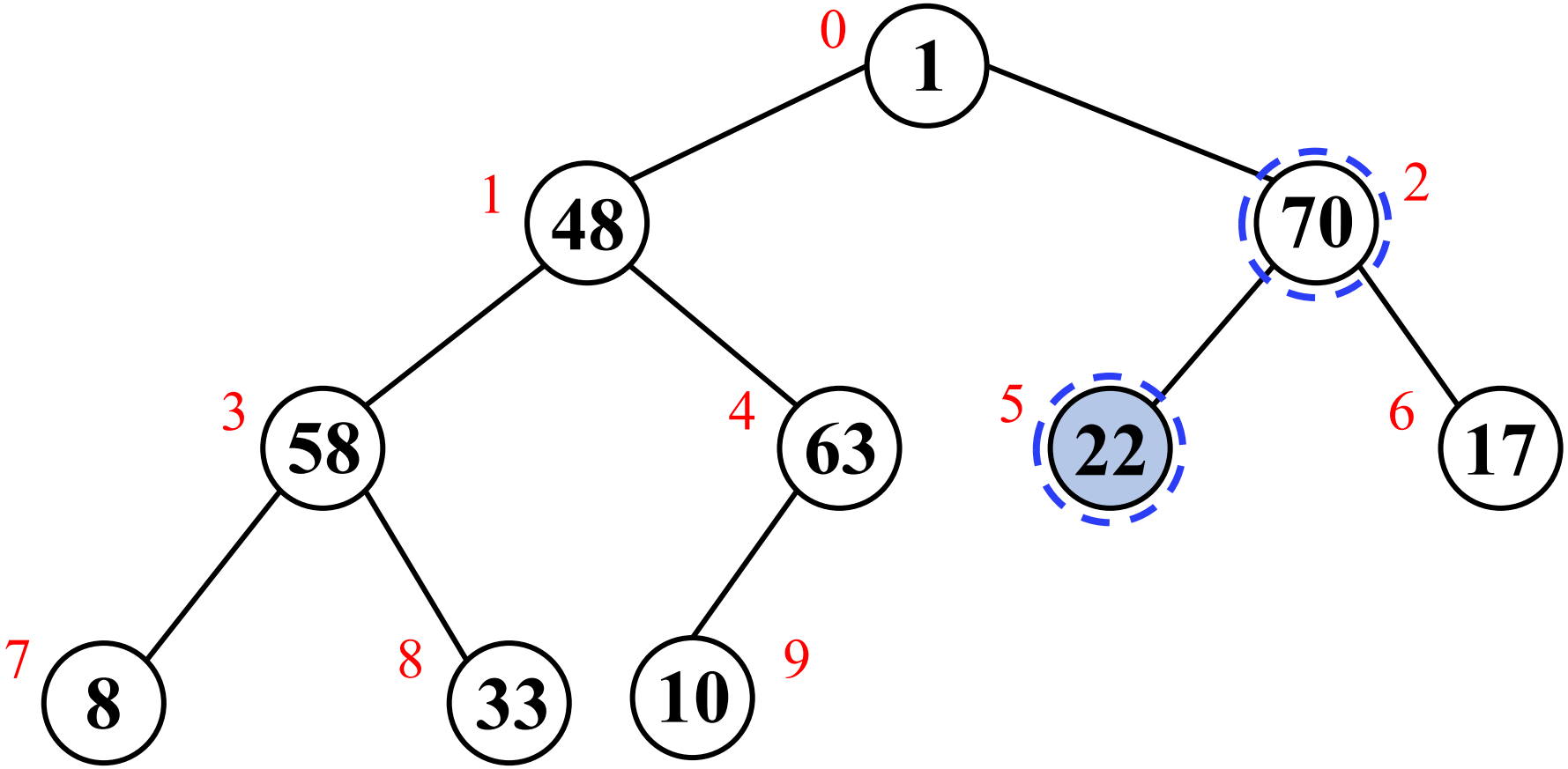
Heapsort algorithms: array implementation



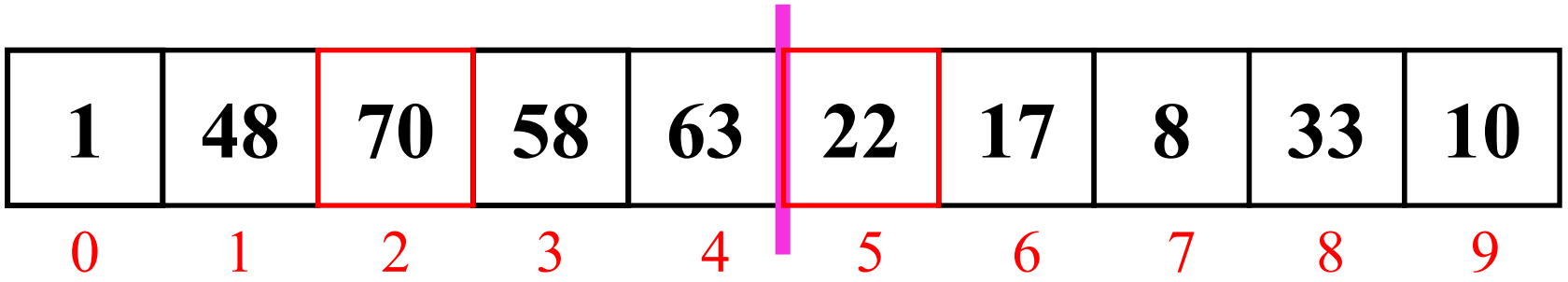
sink 22



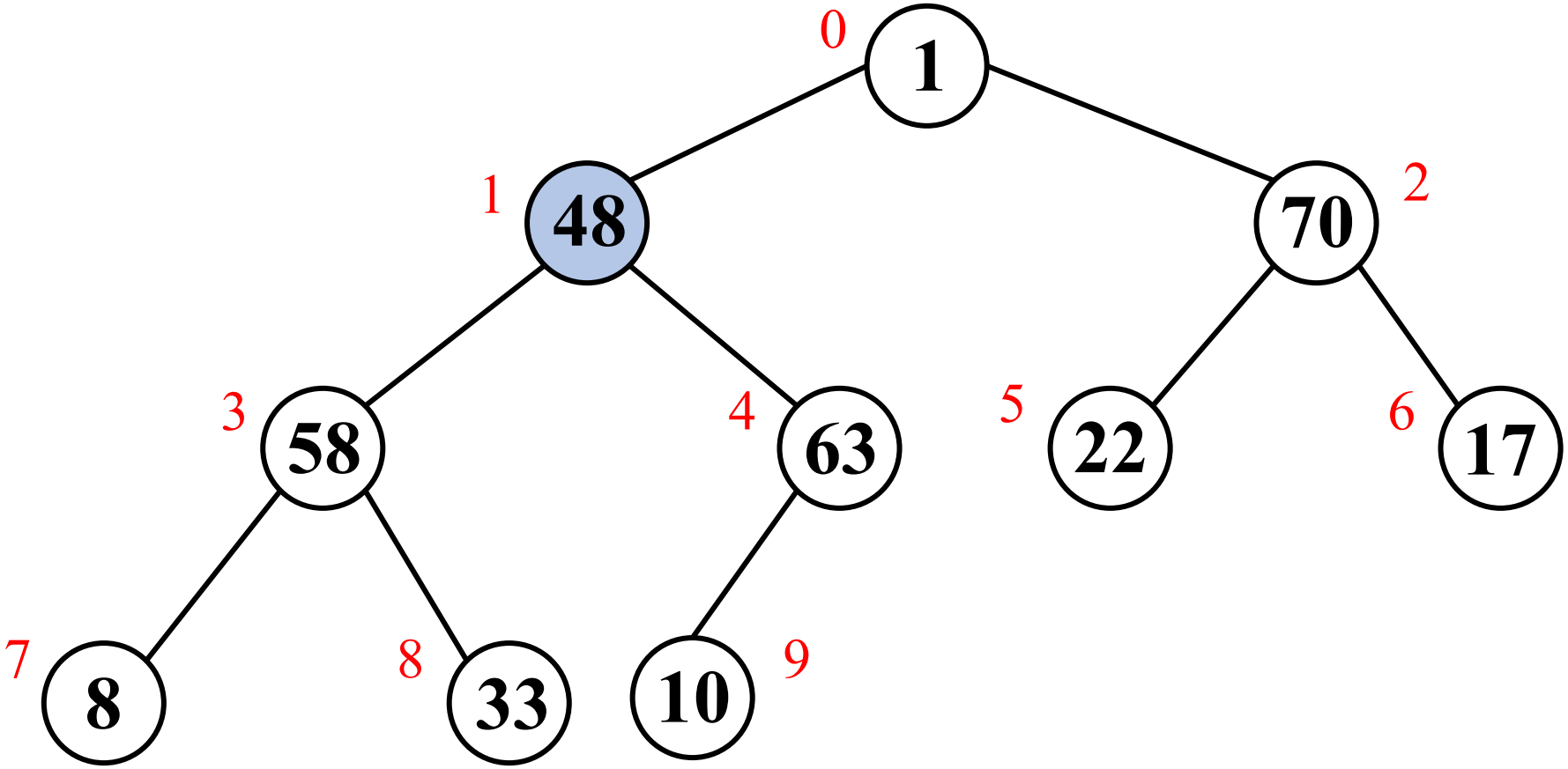
Heapsort algorithms: array implementation



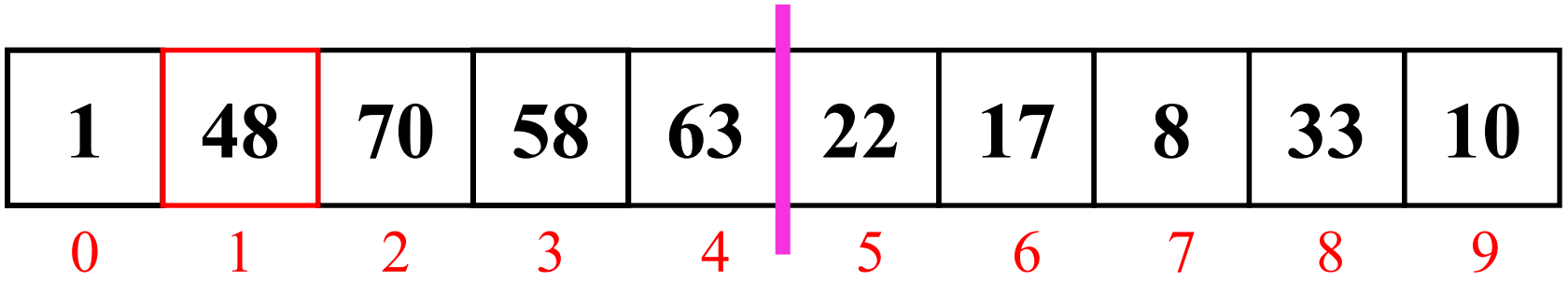
sink 22



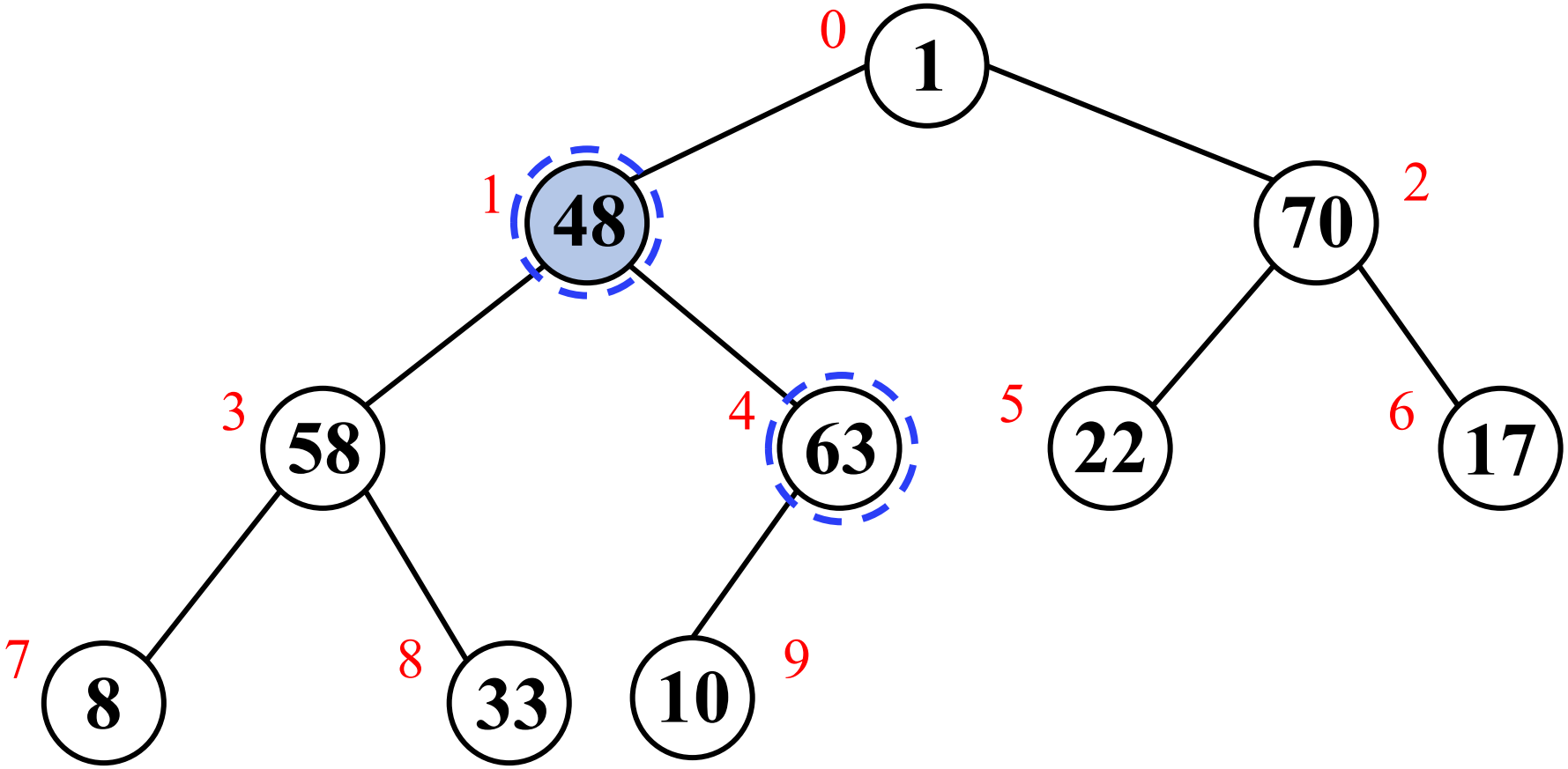
Heapsort algorithms: array implementation



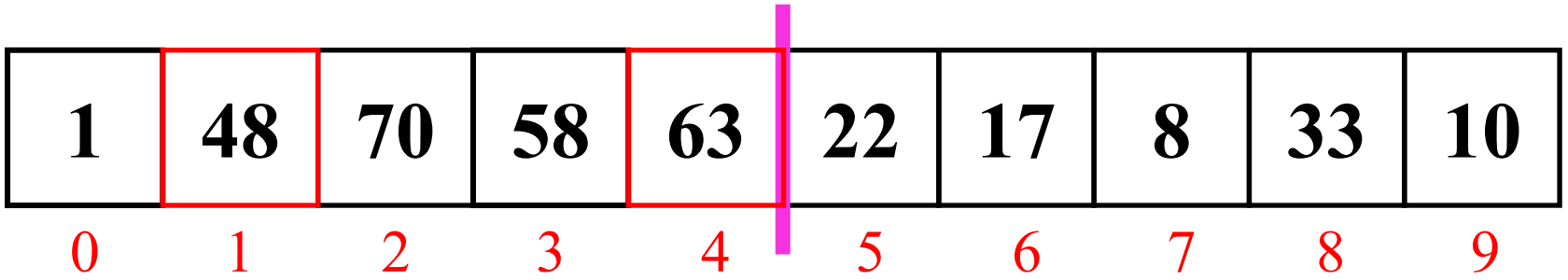
sink 48



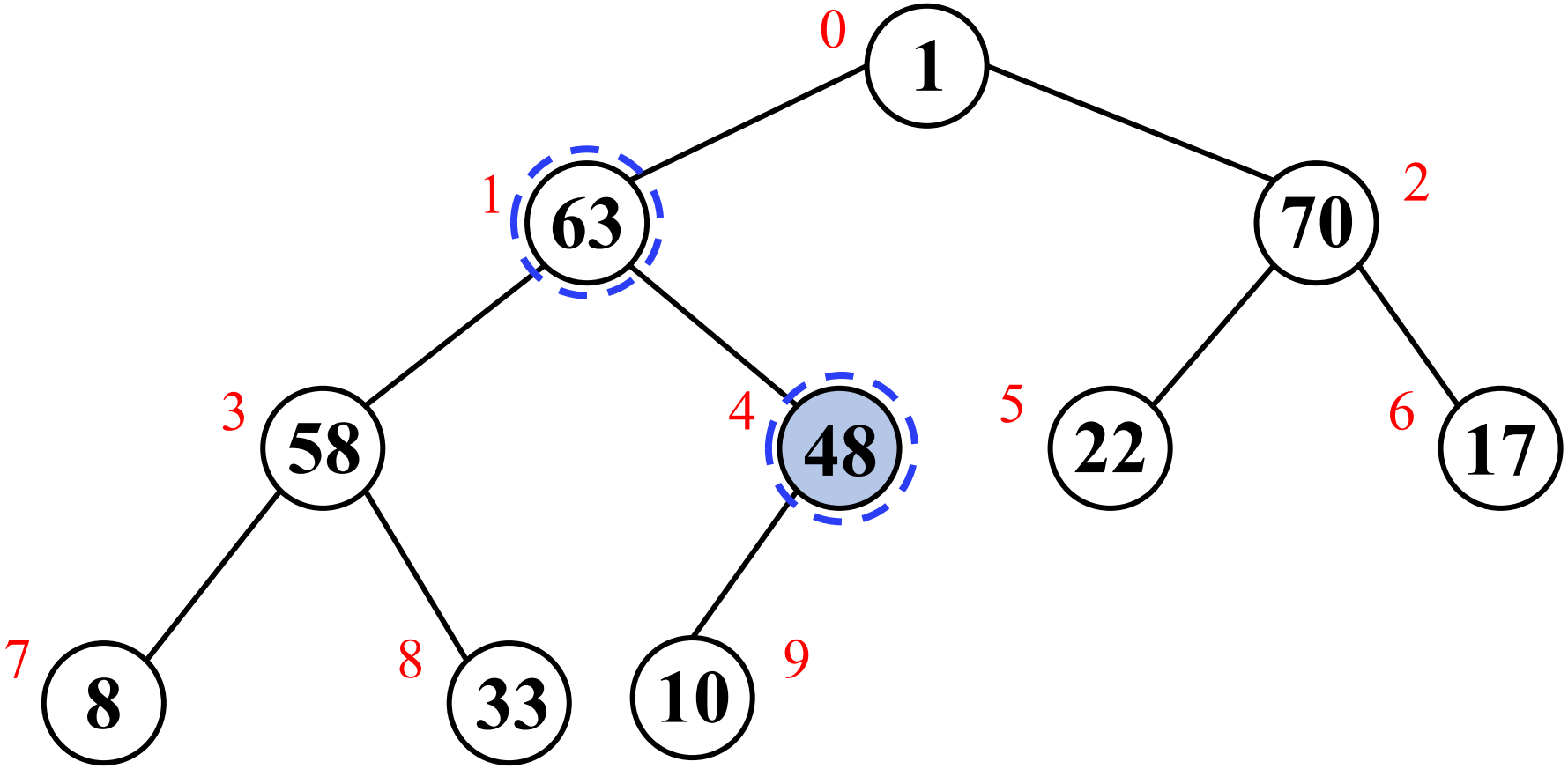
Heapsort algorithms: array implementation



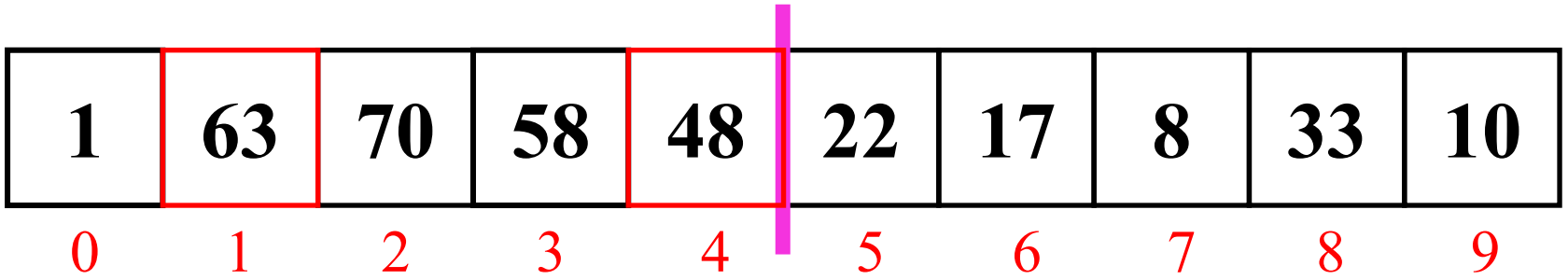
sink 48



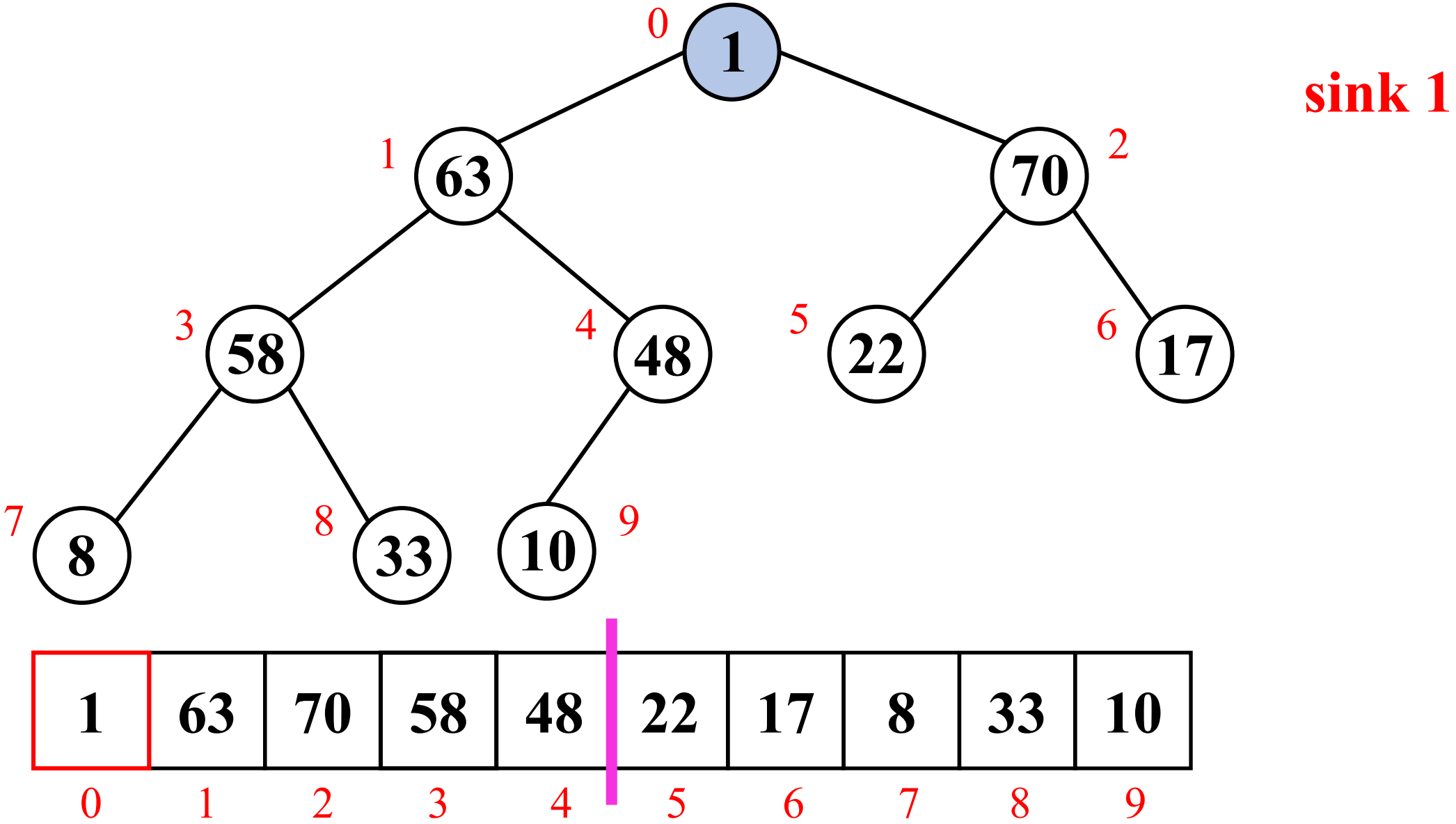
Heapsort algorithms: array implementation



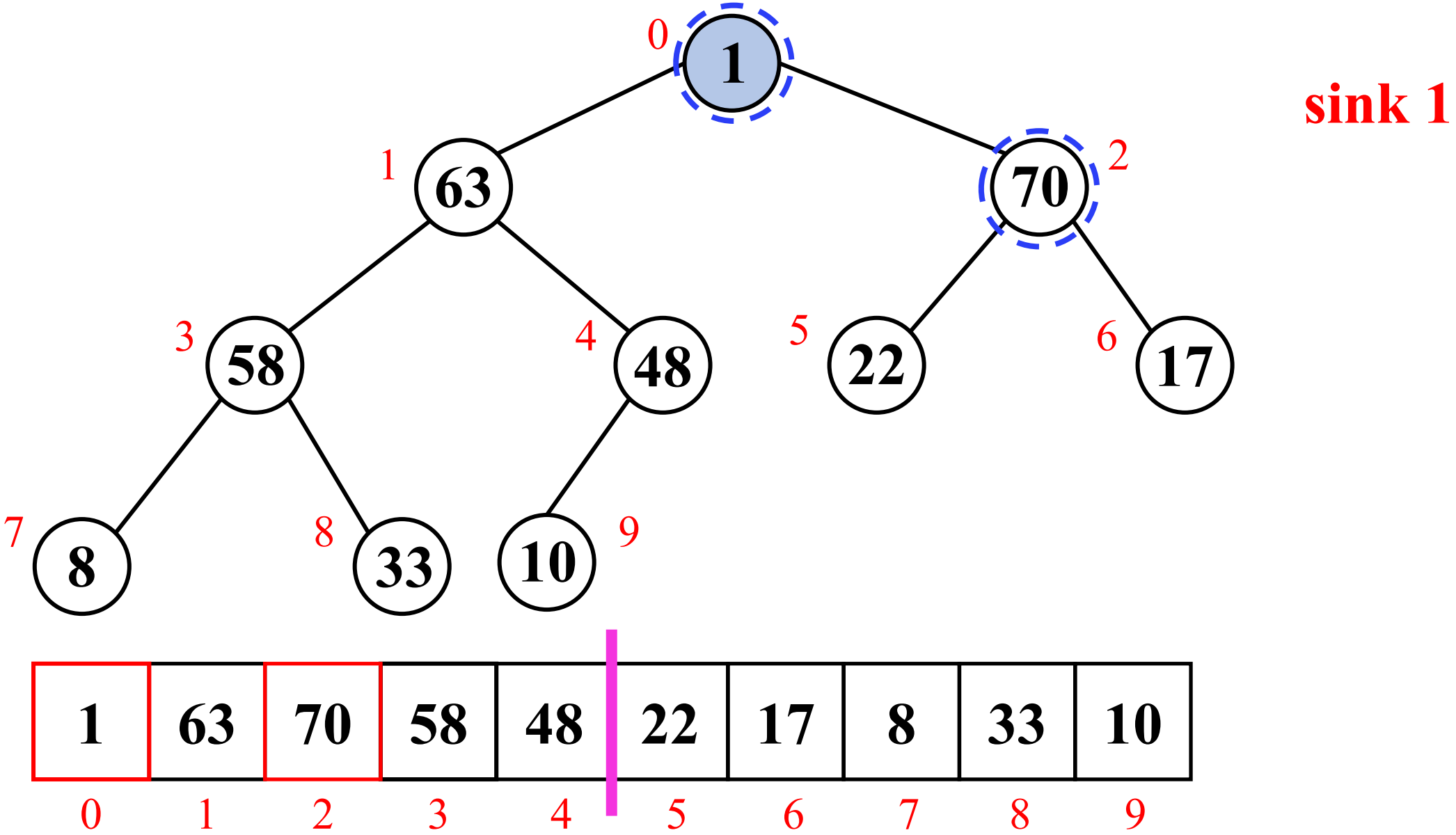
sink 48



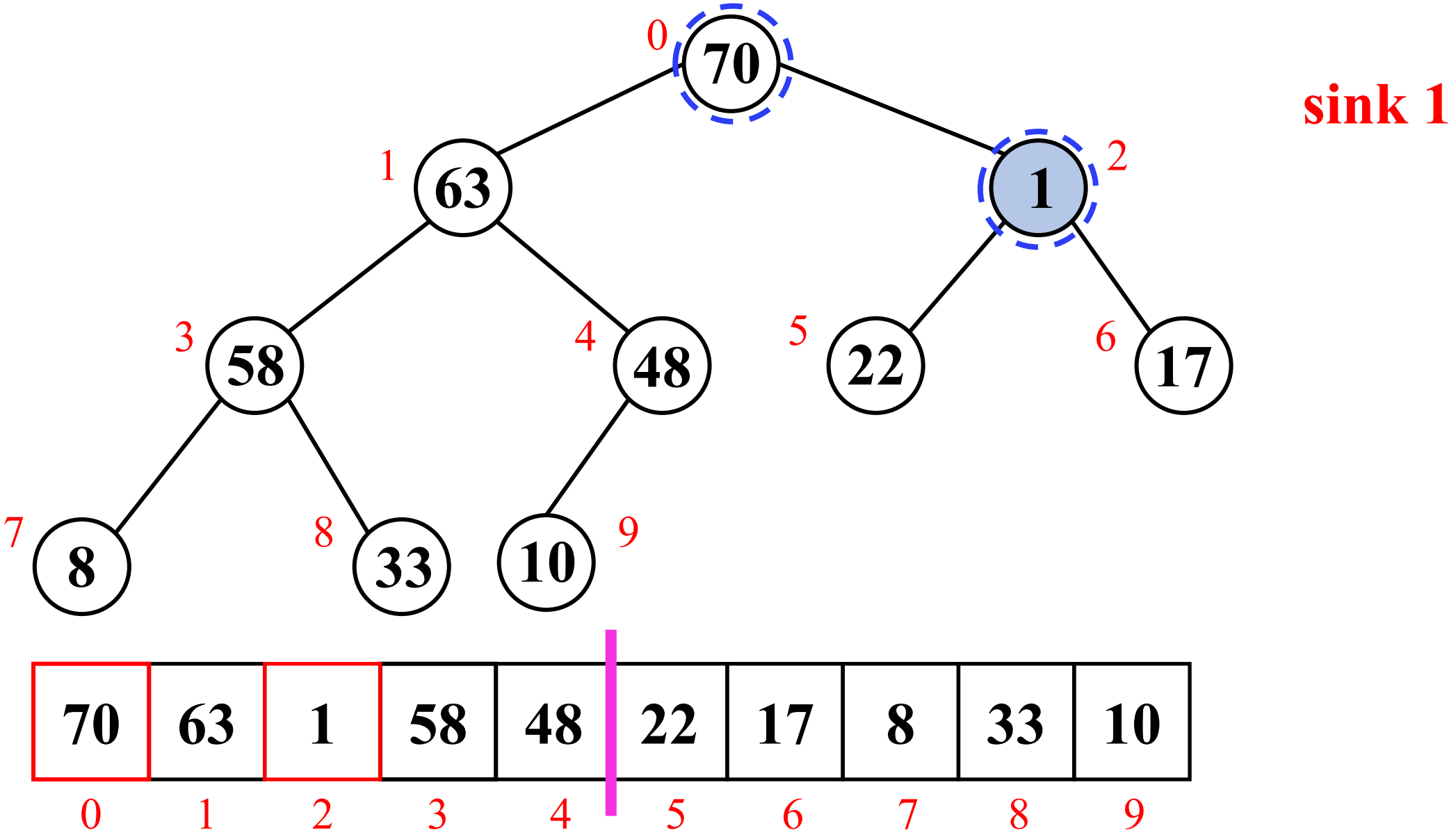
Heapsort algorithms: array implementation



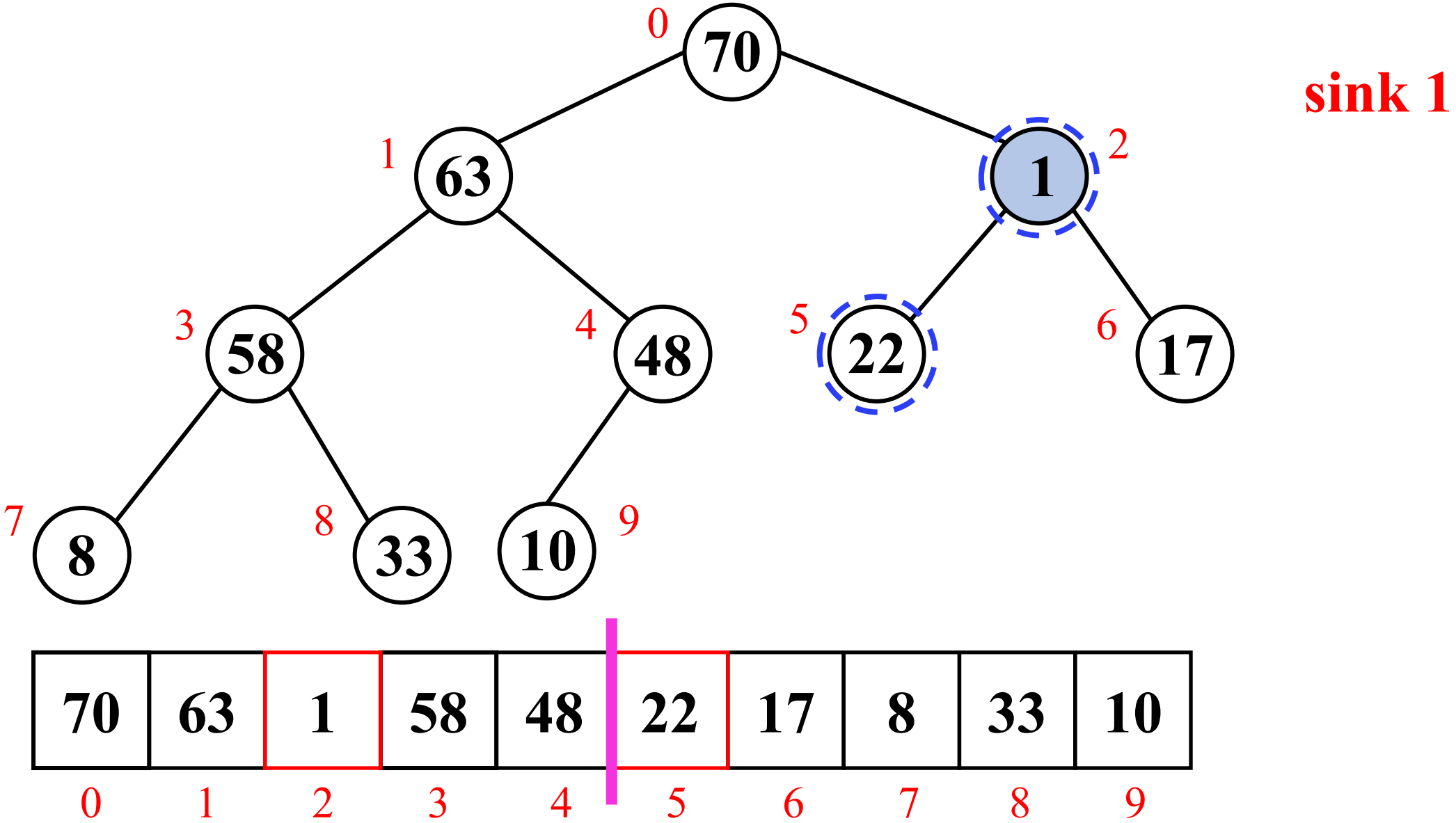
Heapsort algorithms: array implementation



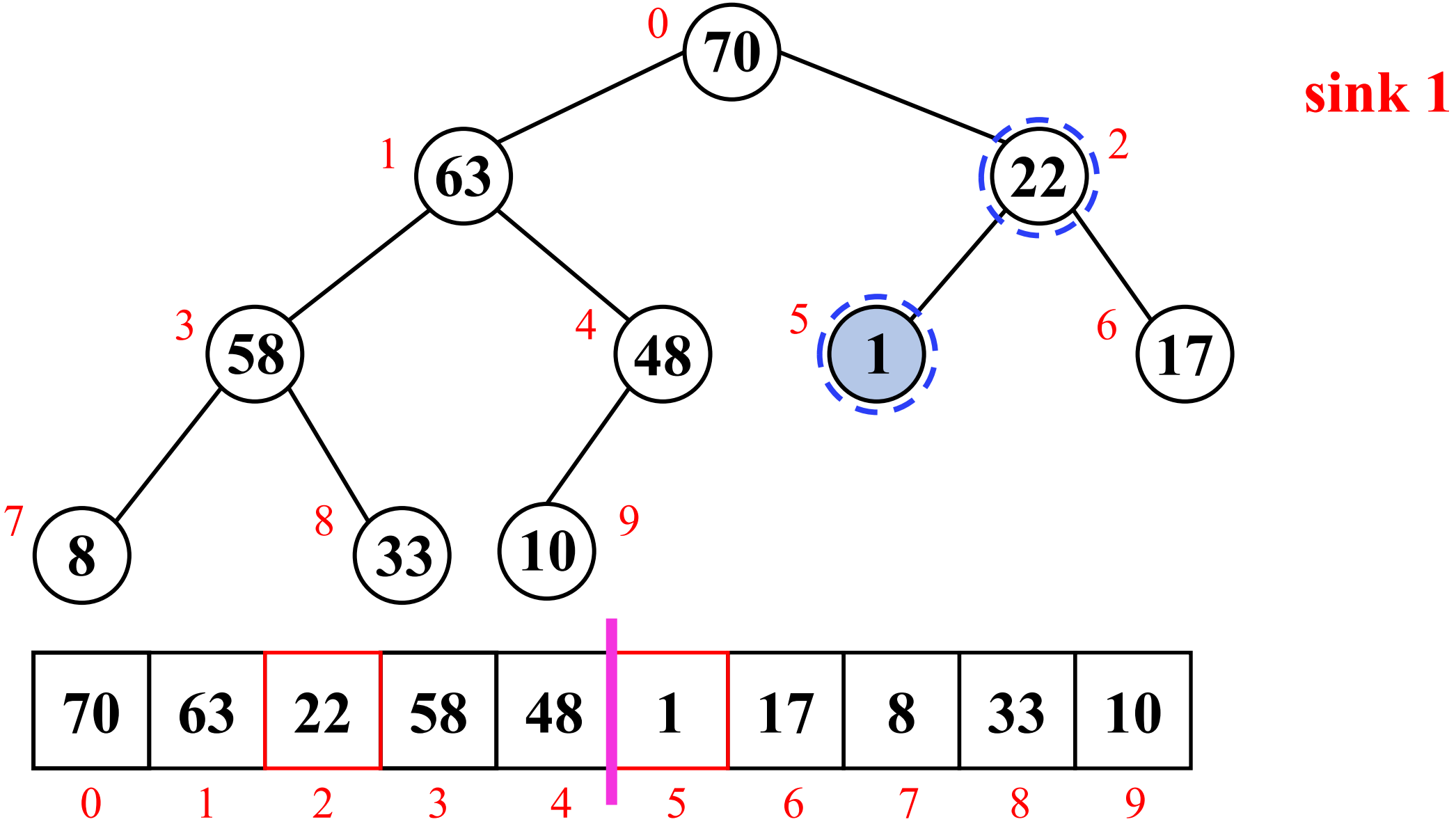
Heapsort algorithms: array implementation



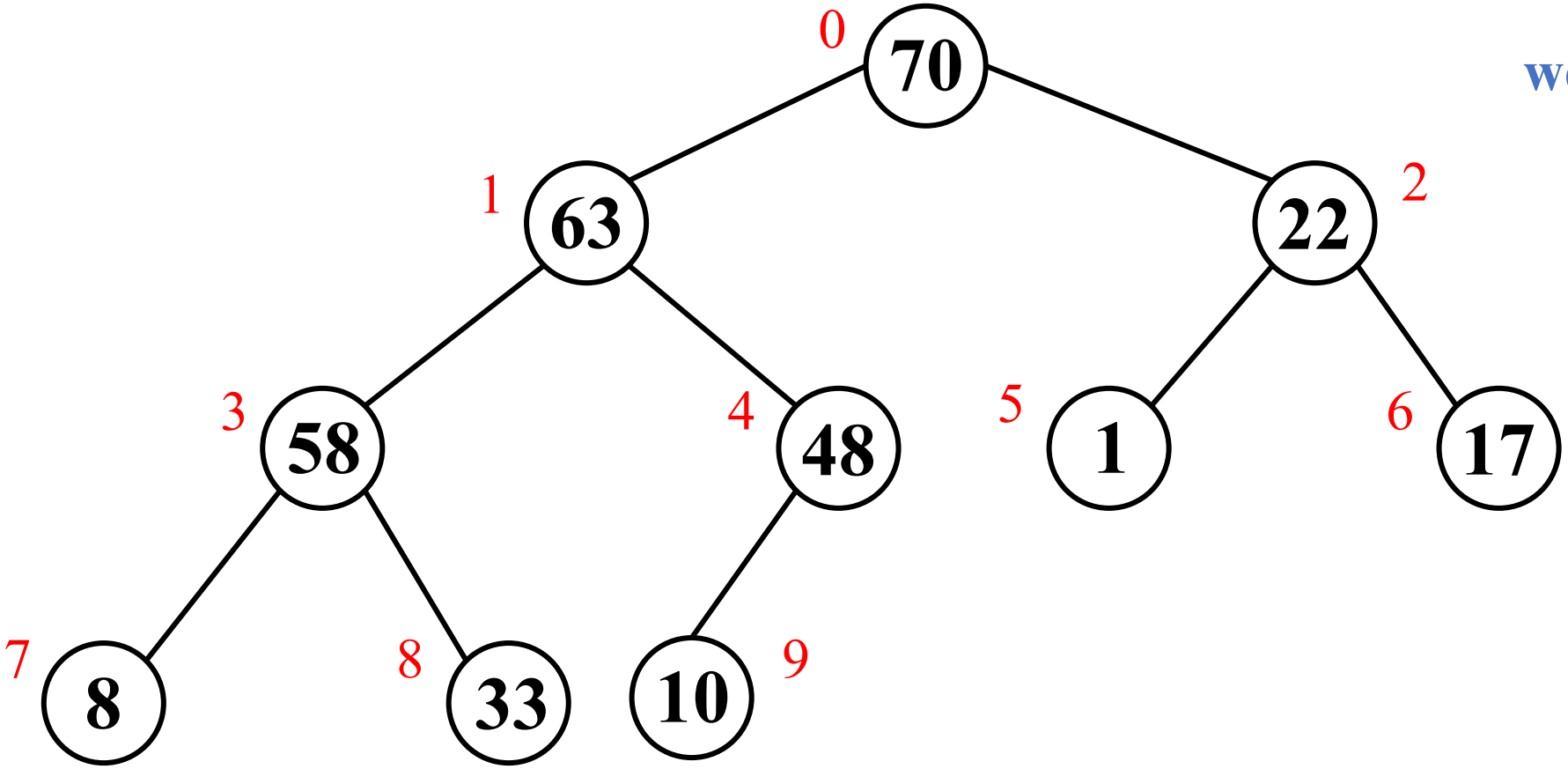
Heapsort algorithms: array implementation



Heapsort algorithms: array implementation



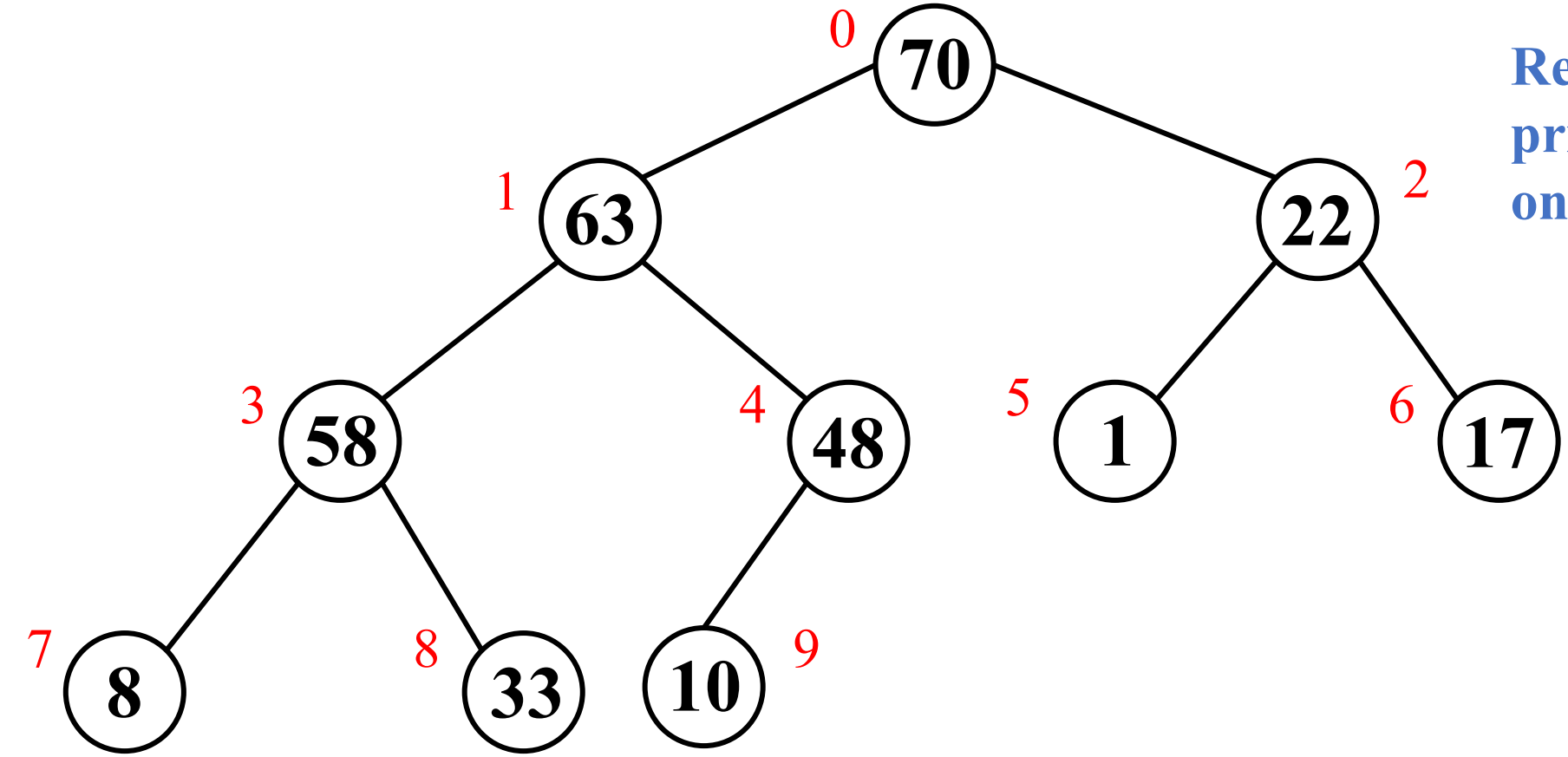
Heapsort algorithms: array implementation



we have built the heap

70	63	22	58	48	1	17	8	33	10
0	1	2	3	4	5	6	7	8	9

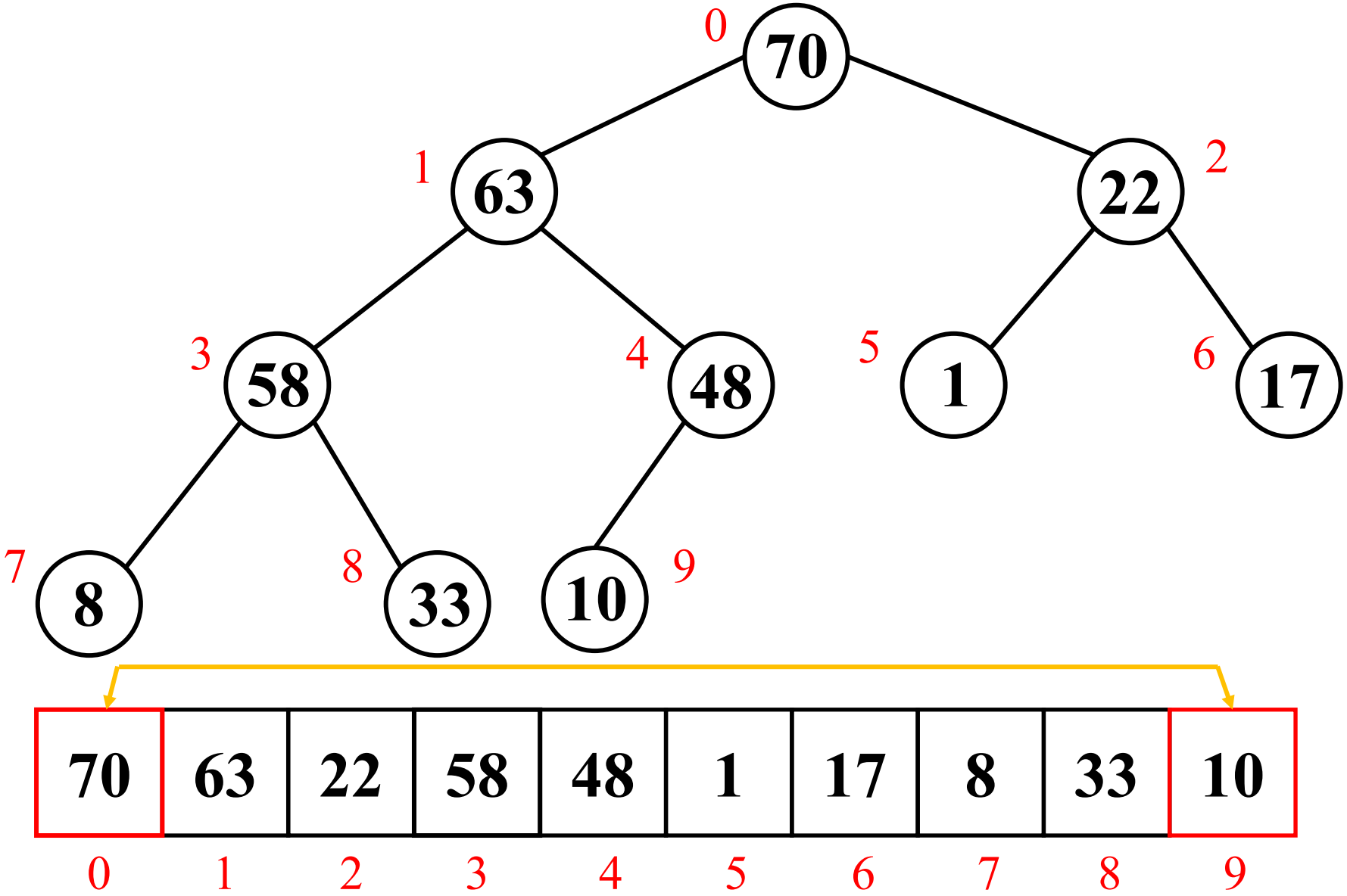
Heapsort algorithms: array implementation



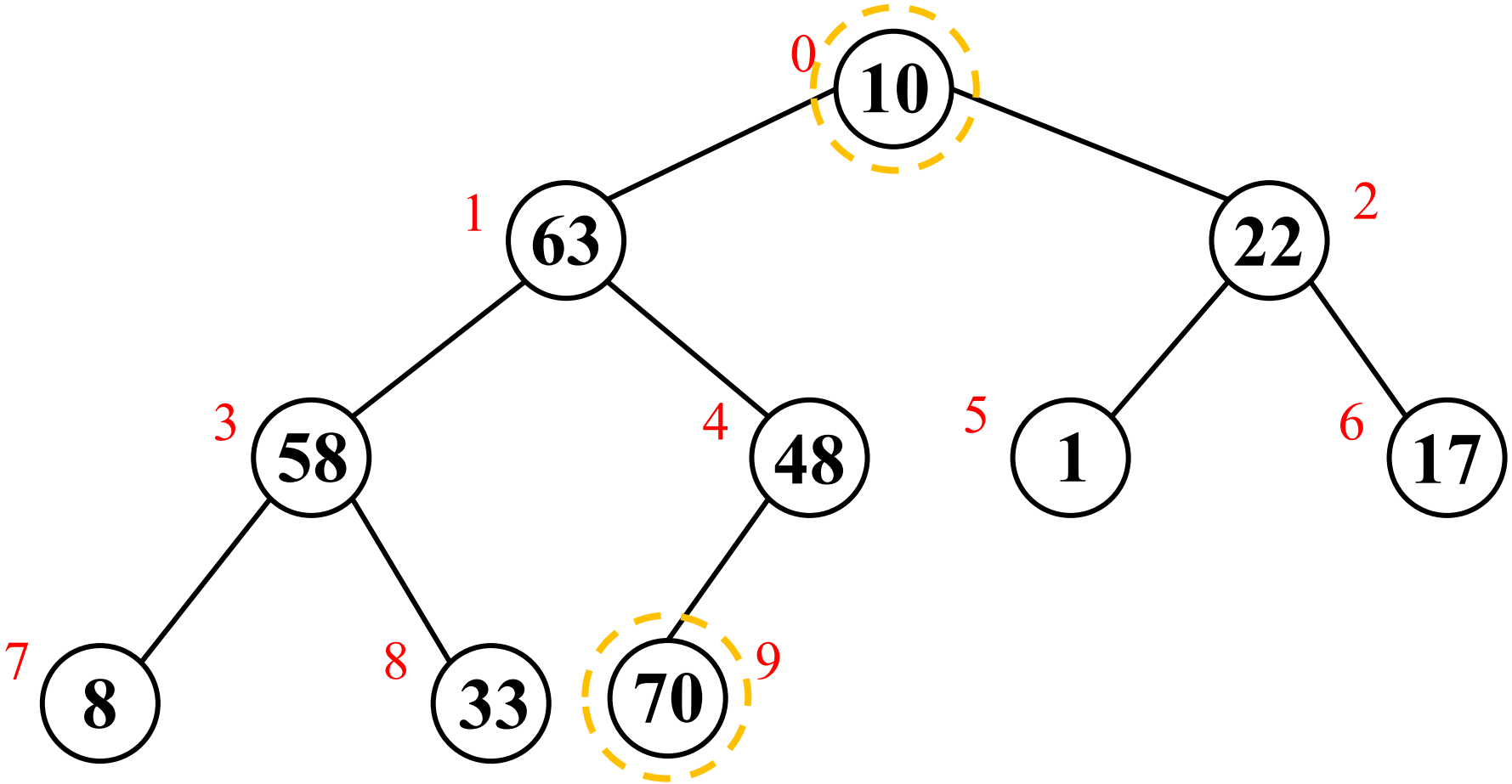
Remove the highest priority element one by one.

70	63	22	58	48	1	17	8	33	10
0	1	2	3	4	5	6	7	8	9

Heapsort algorithms: array implementation



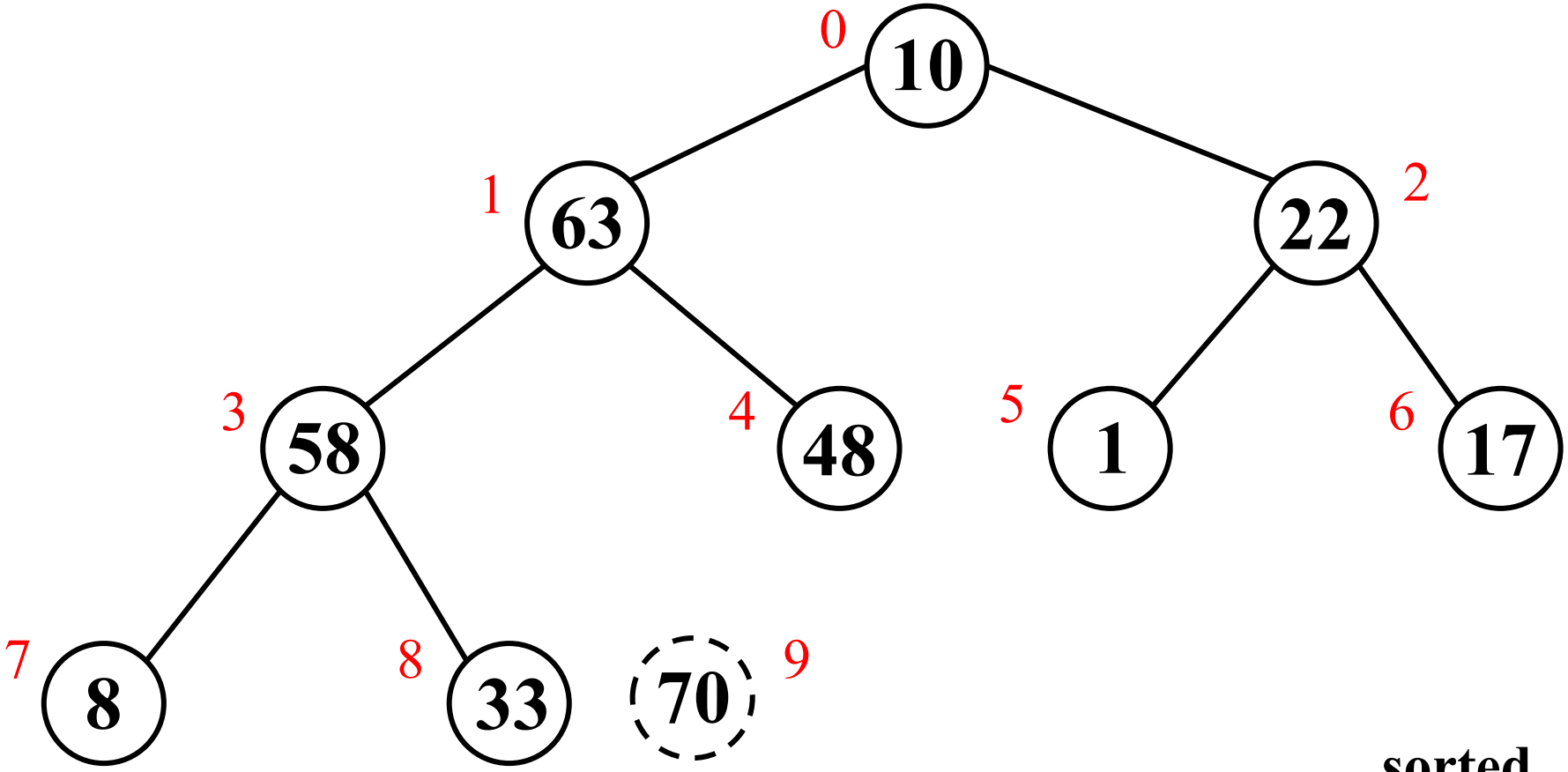
Heapsort algorithms: array implementation



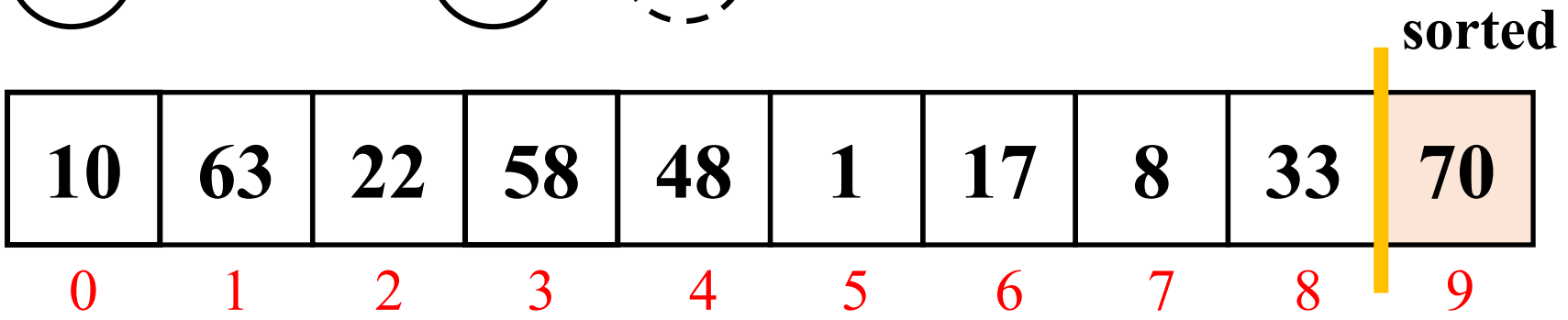
swap

10	63	22	58	48	1	17	8	33	70
0	1	2	3	4	5	6	7	8	9

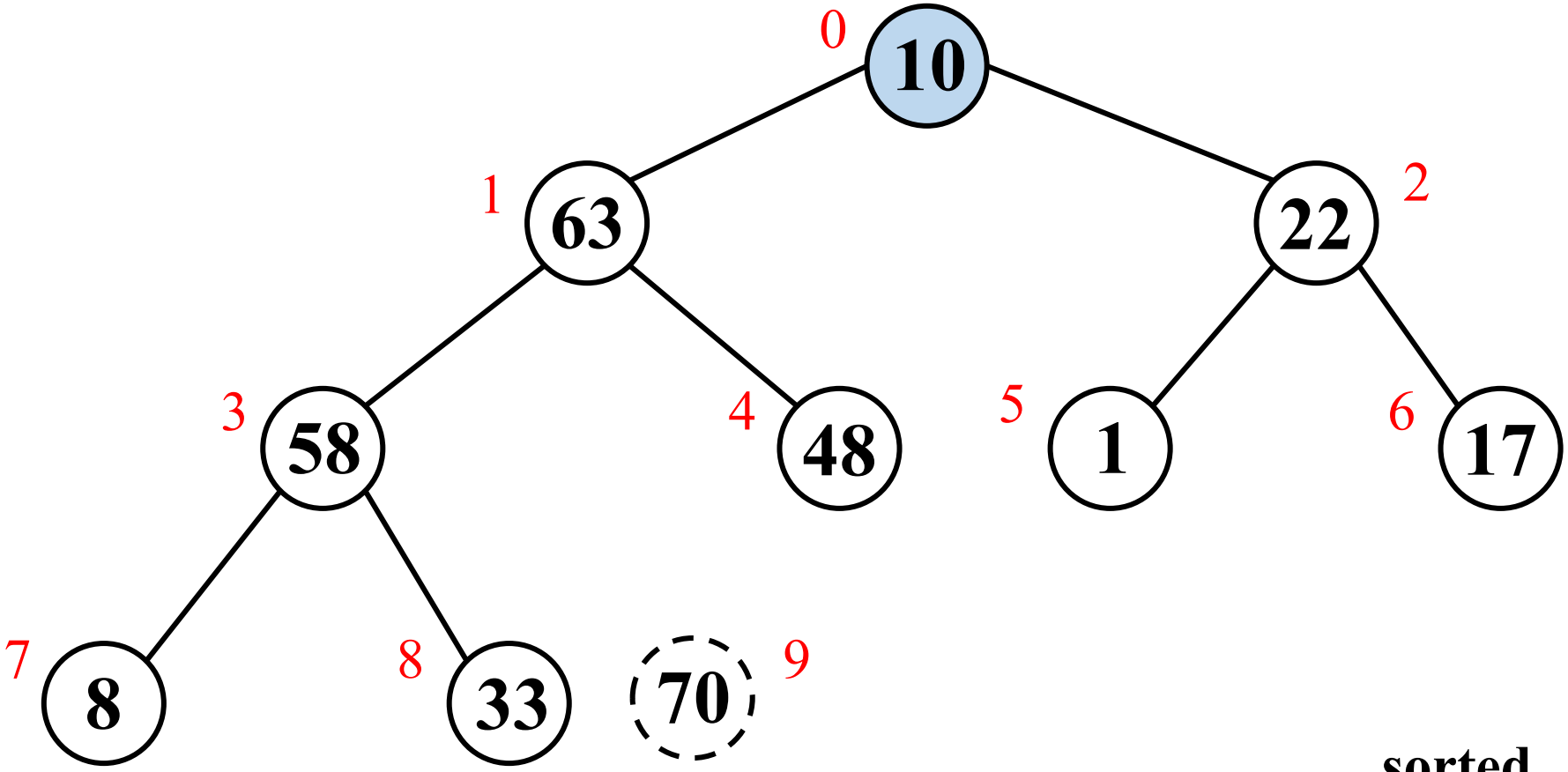
Heapsort algorithms: array implementation



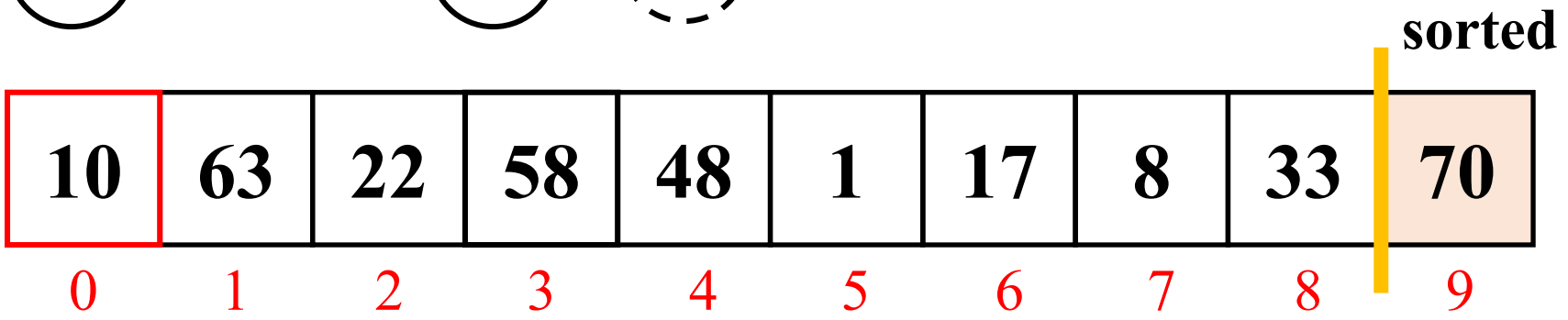
delete



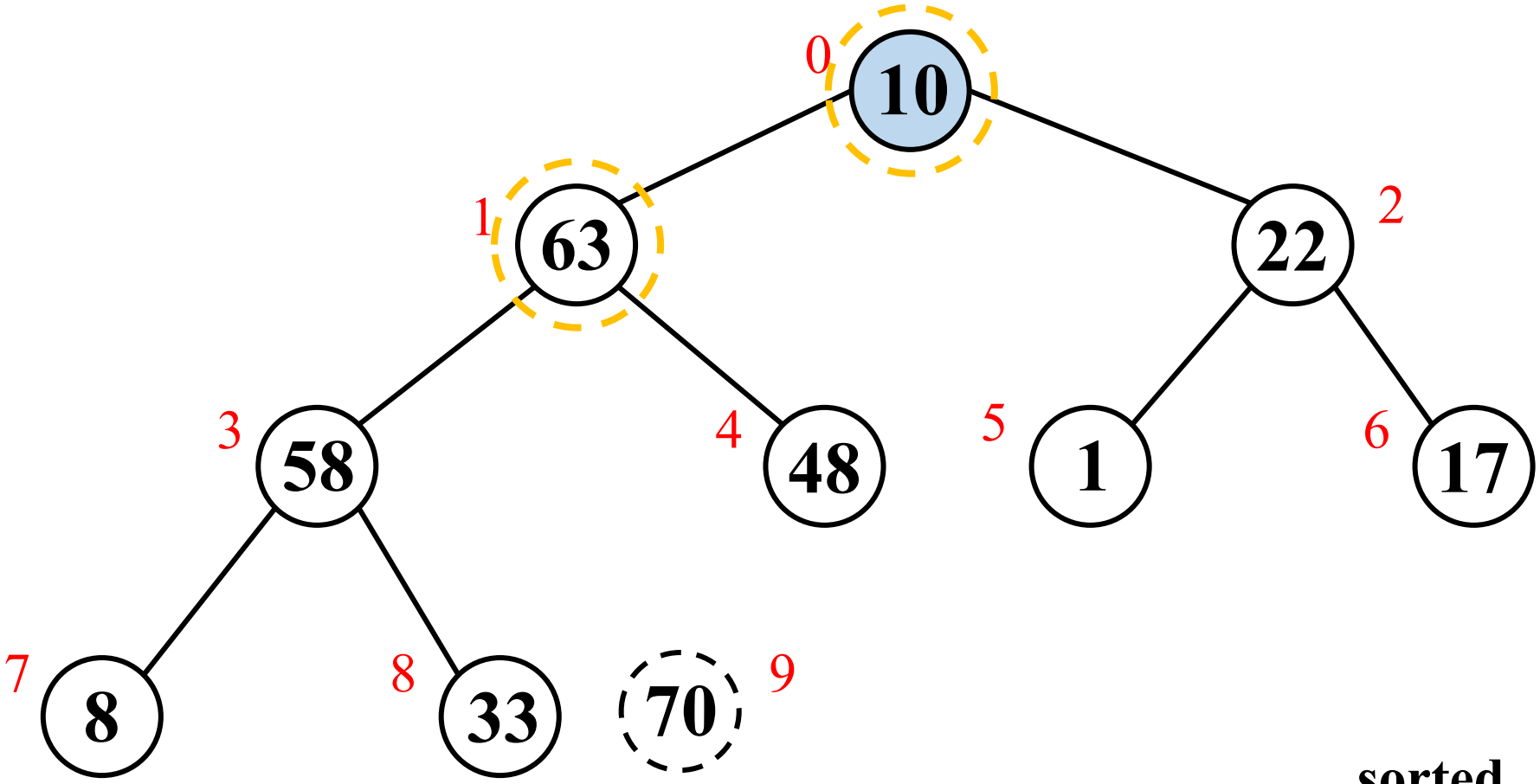
Heapsort algorithms: array implementation



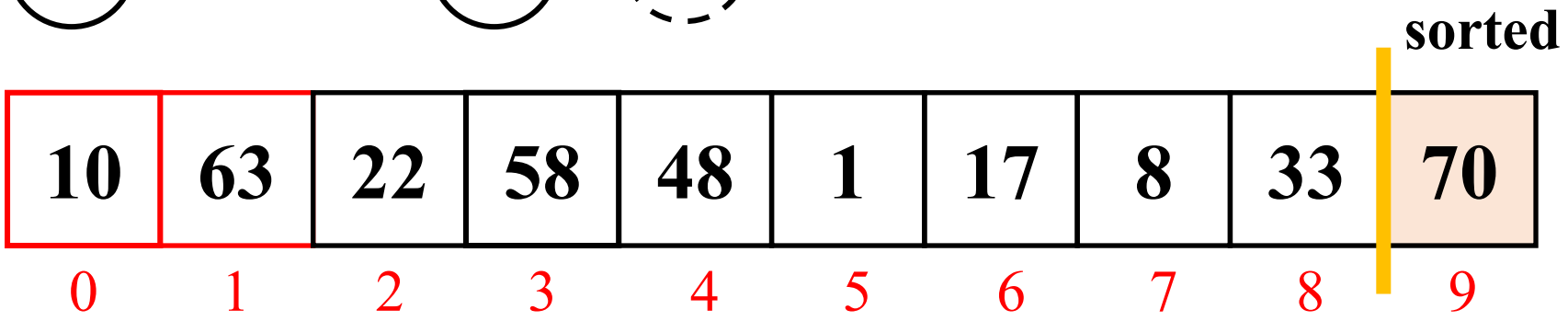
sink 10



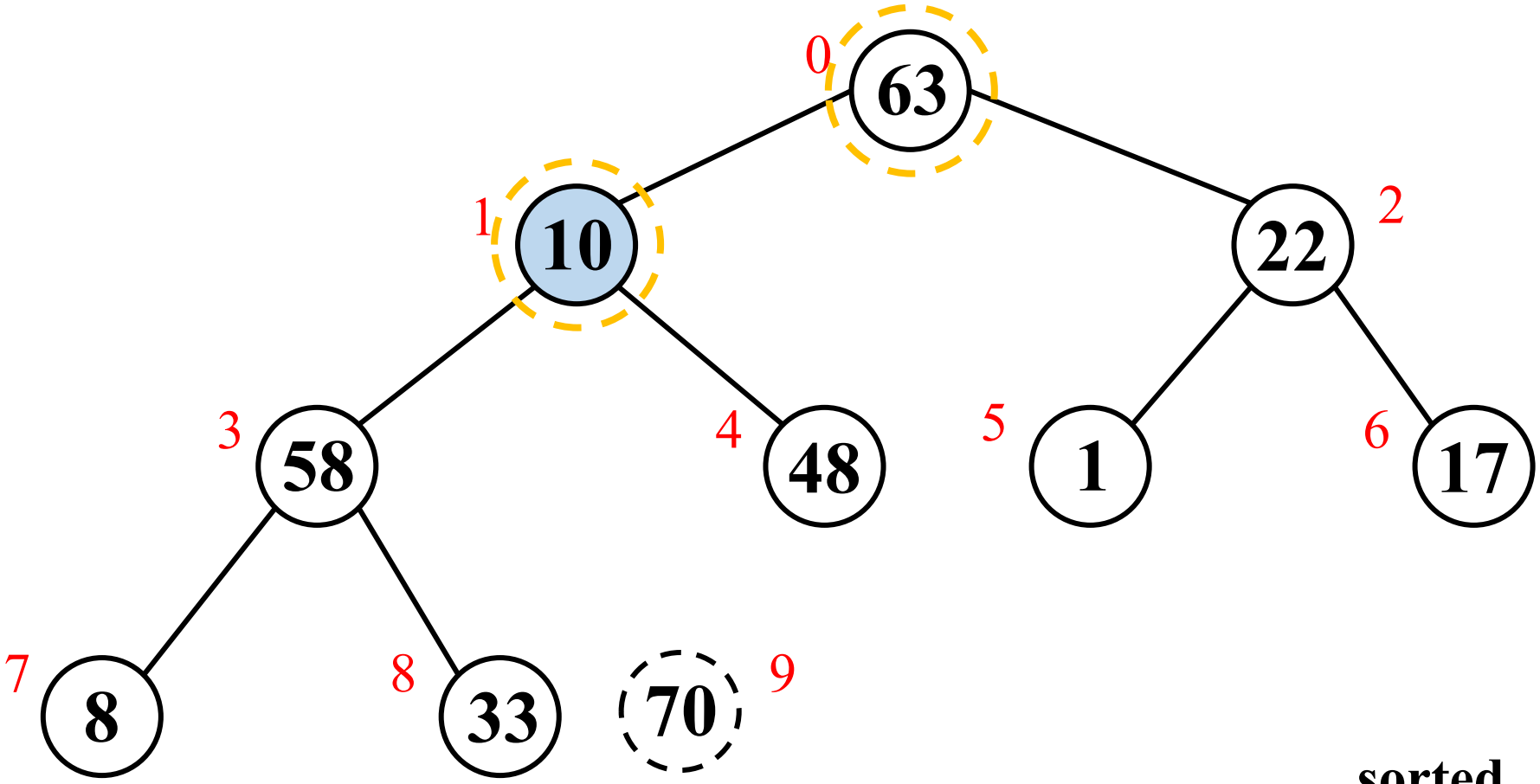
Heapsort algorithms: array implementation



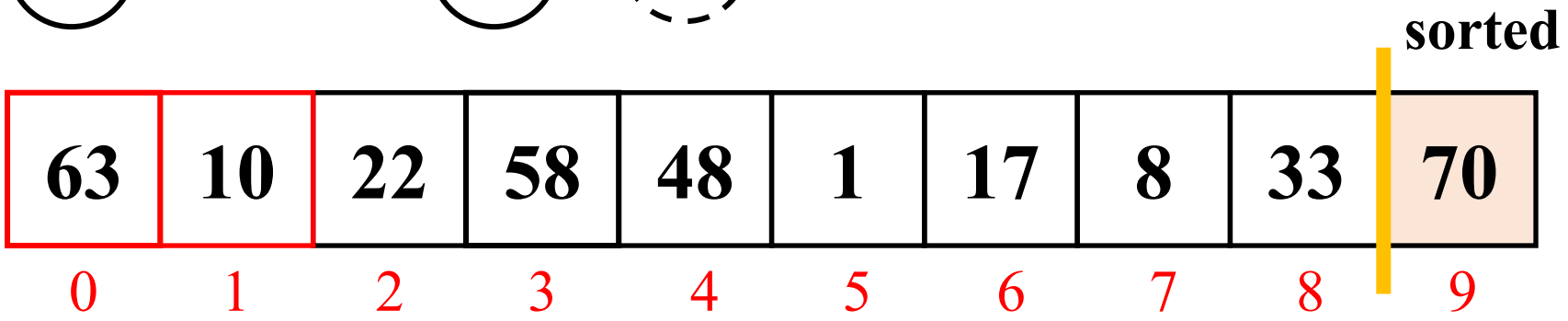
sink 10



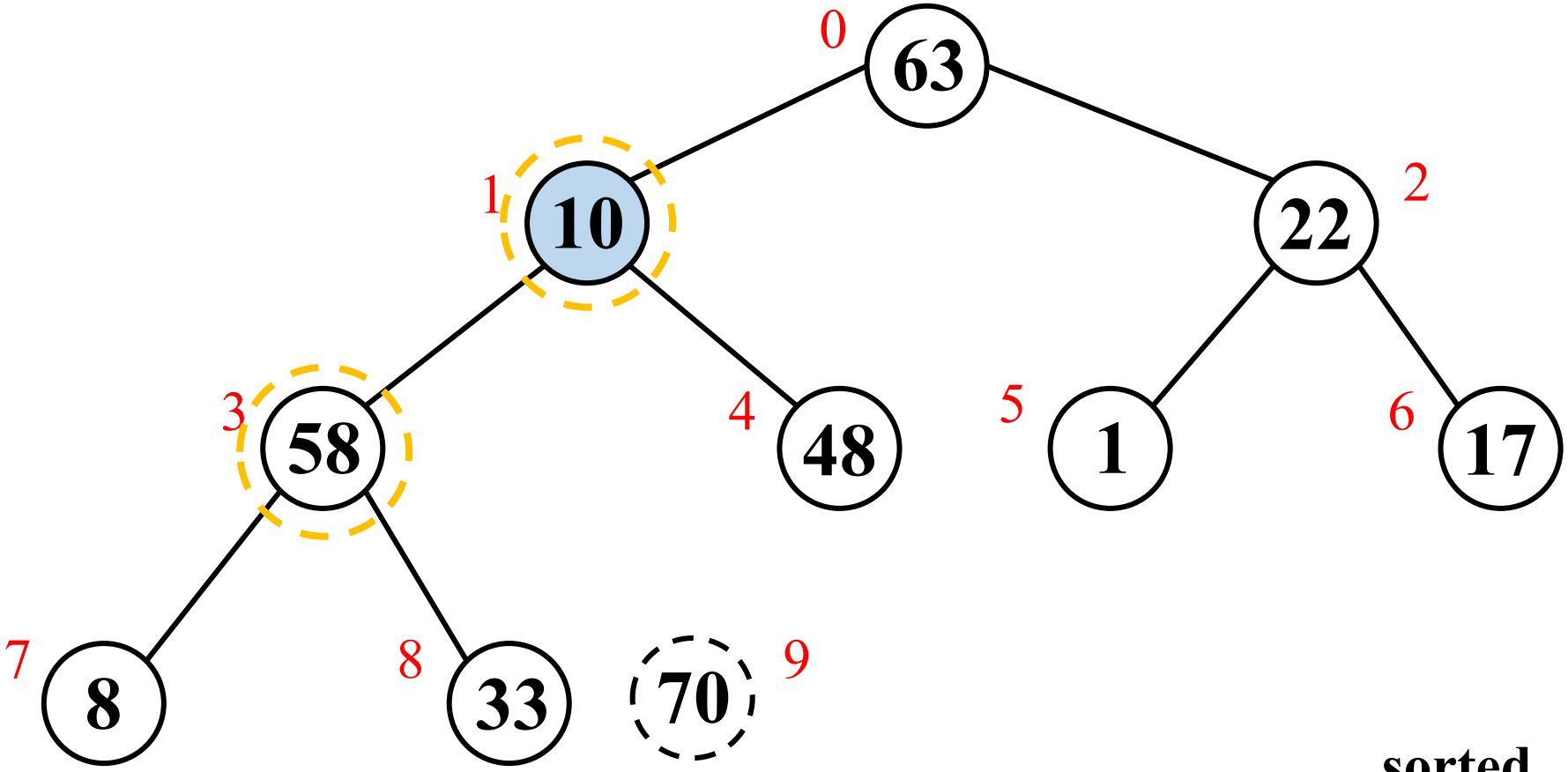
Heapsort algorithms: array implementation



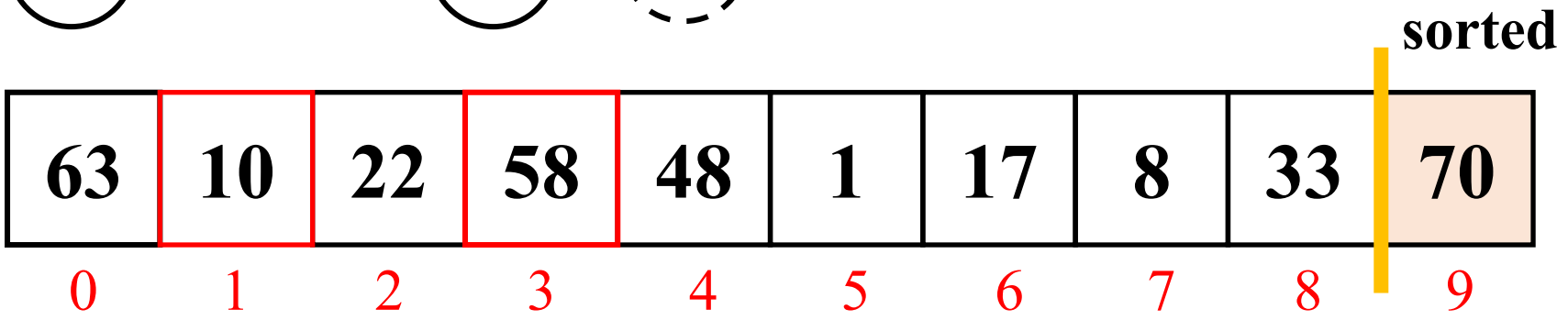
sink 10



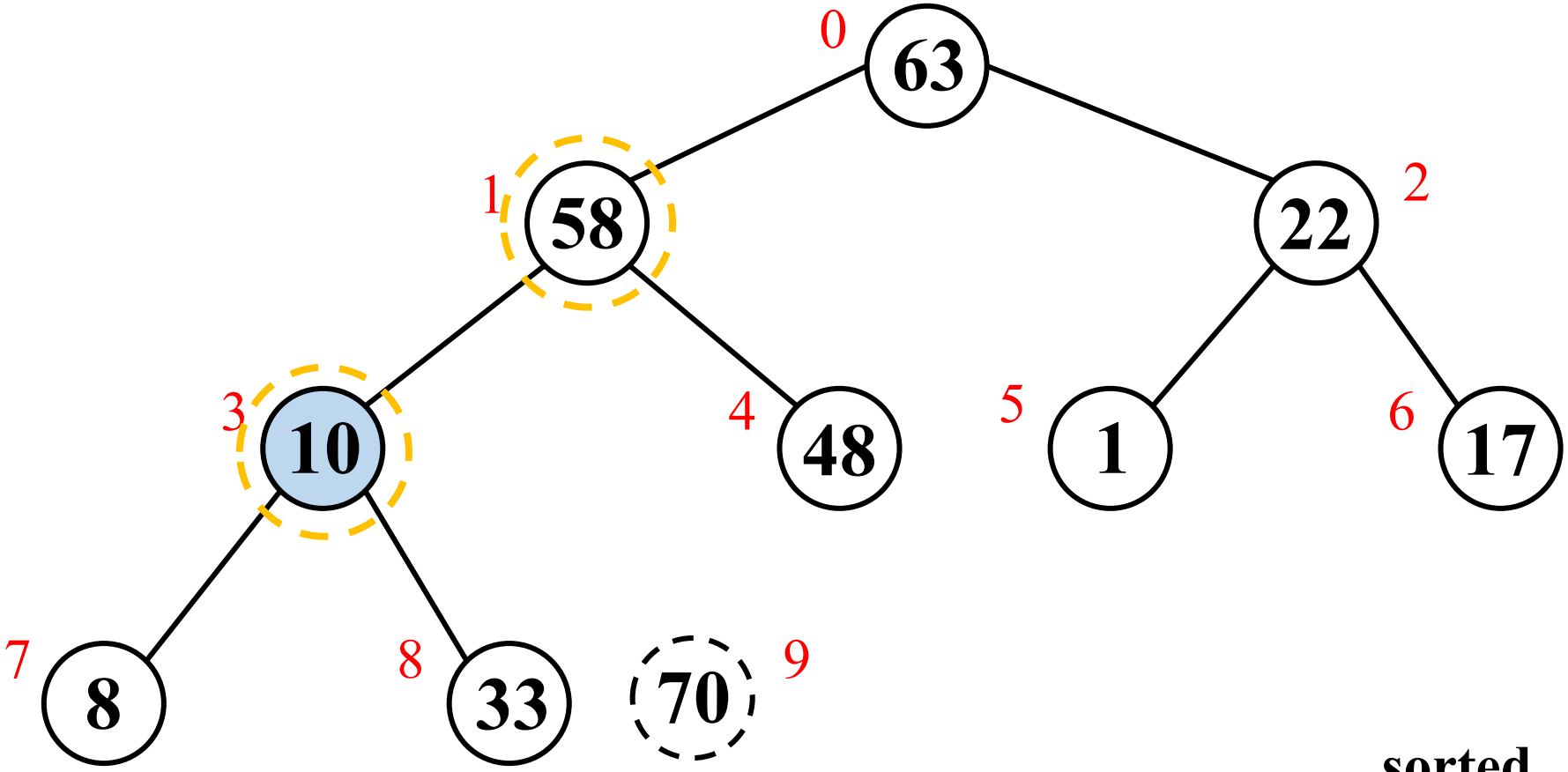
Heapsort algorithms: array implementation



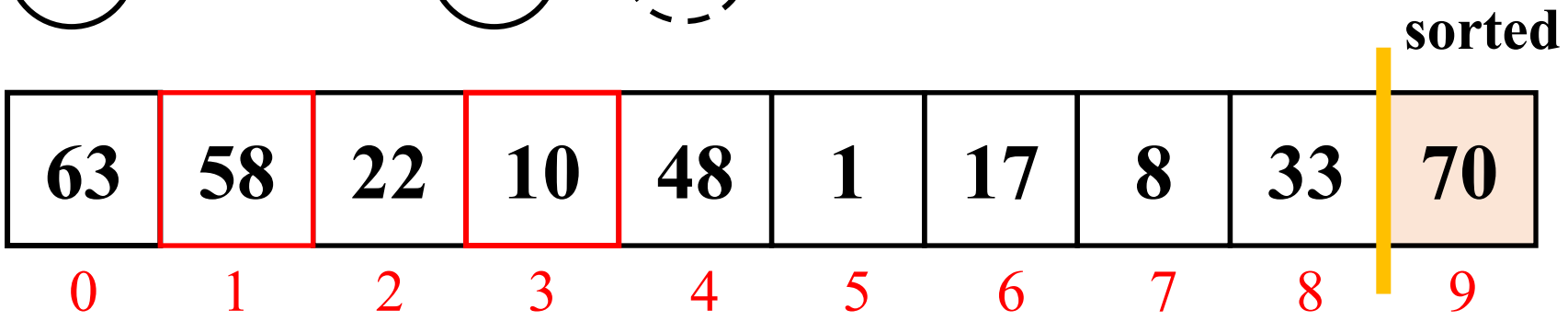
sink 10



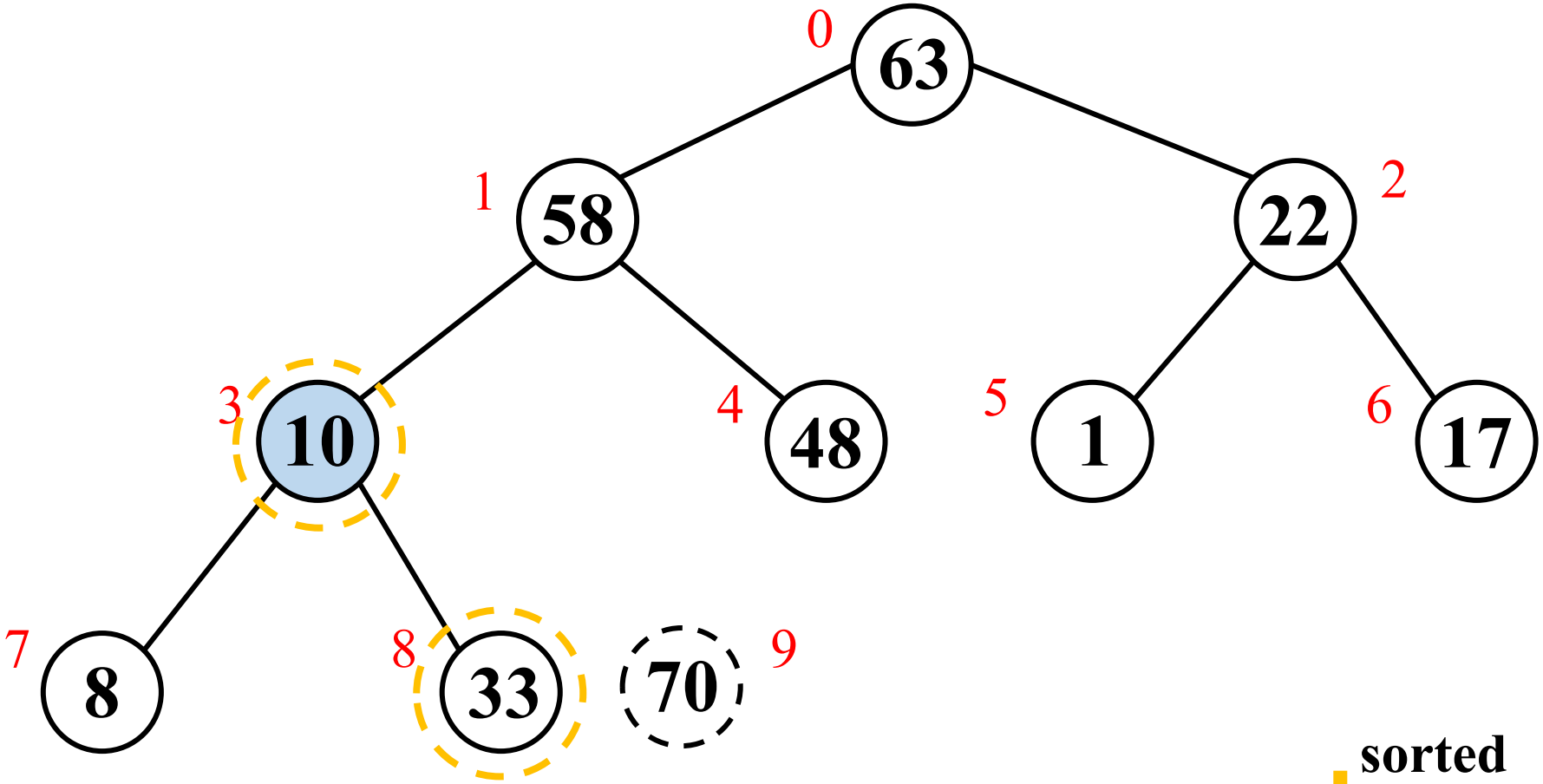
Heapsort algorithms: array implementation



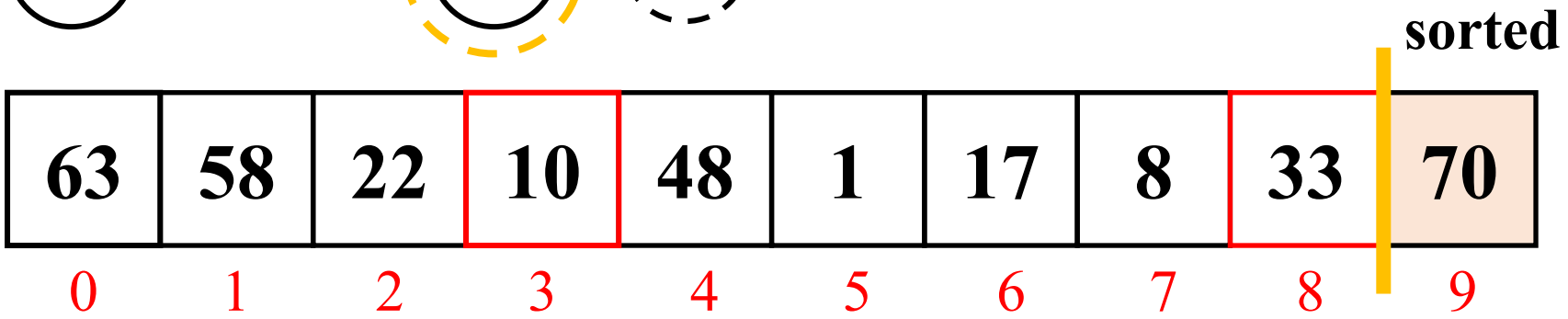
sink 10



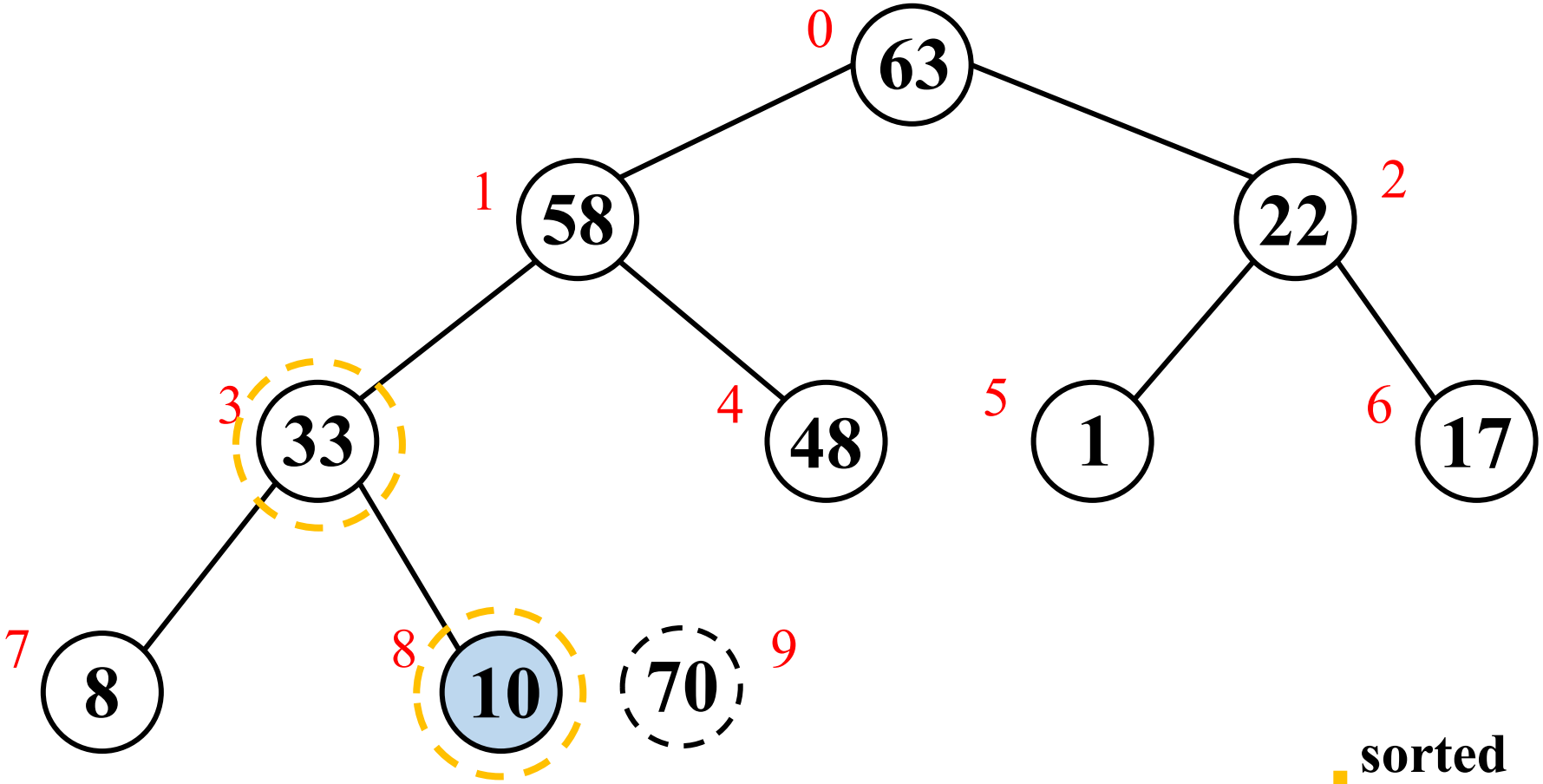
Heapsort algorithms: array implementation



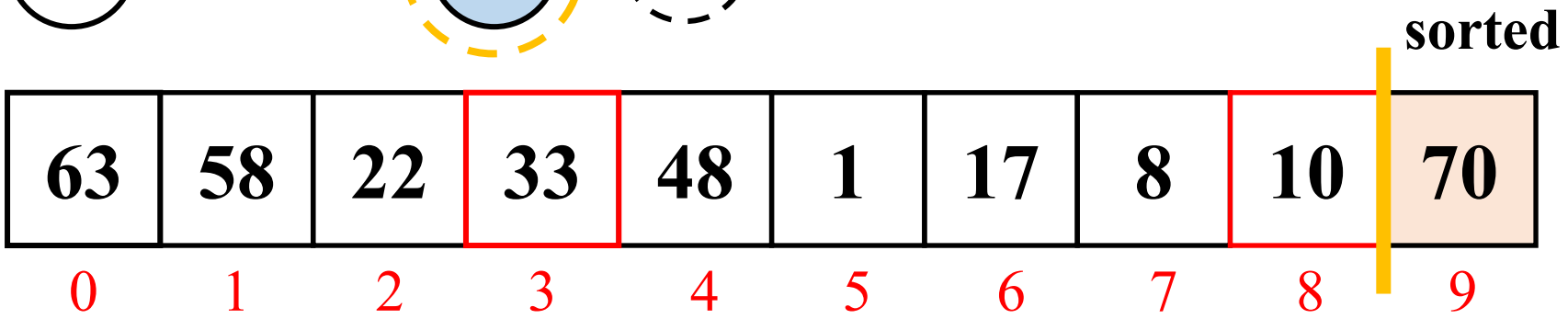
sink 10



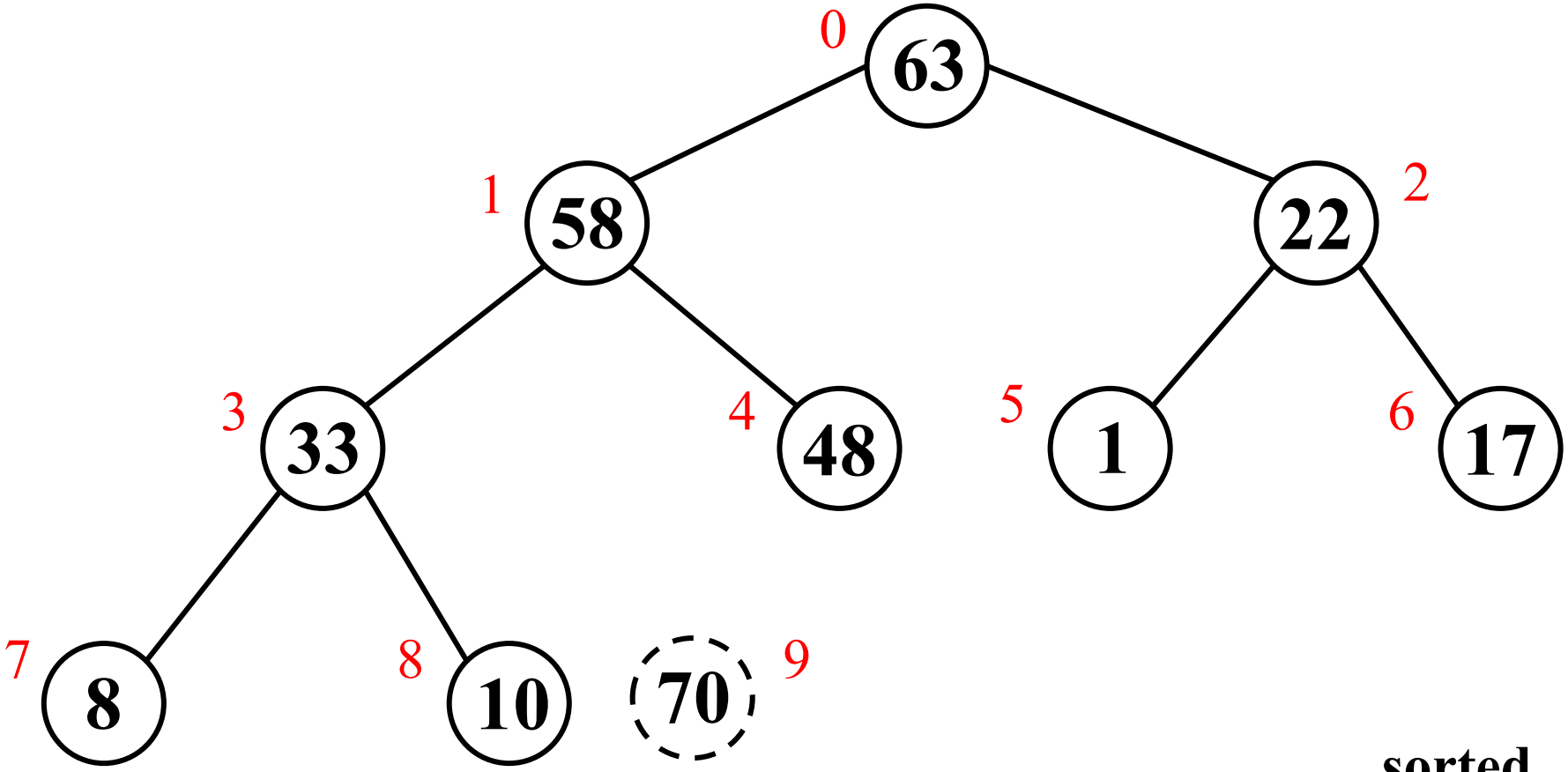
Heapsort algorithms: array implementation

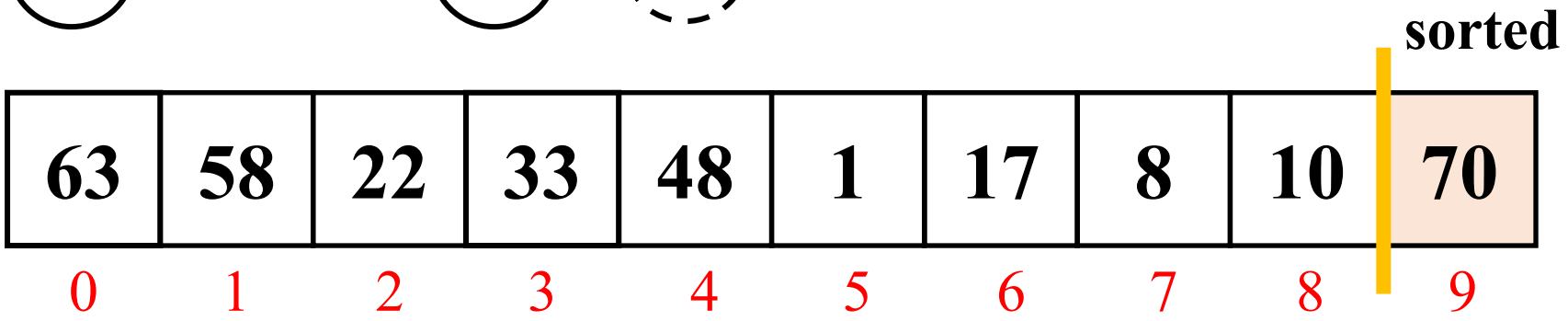


sink 10

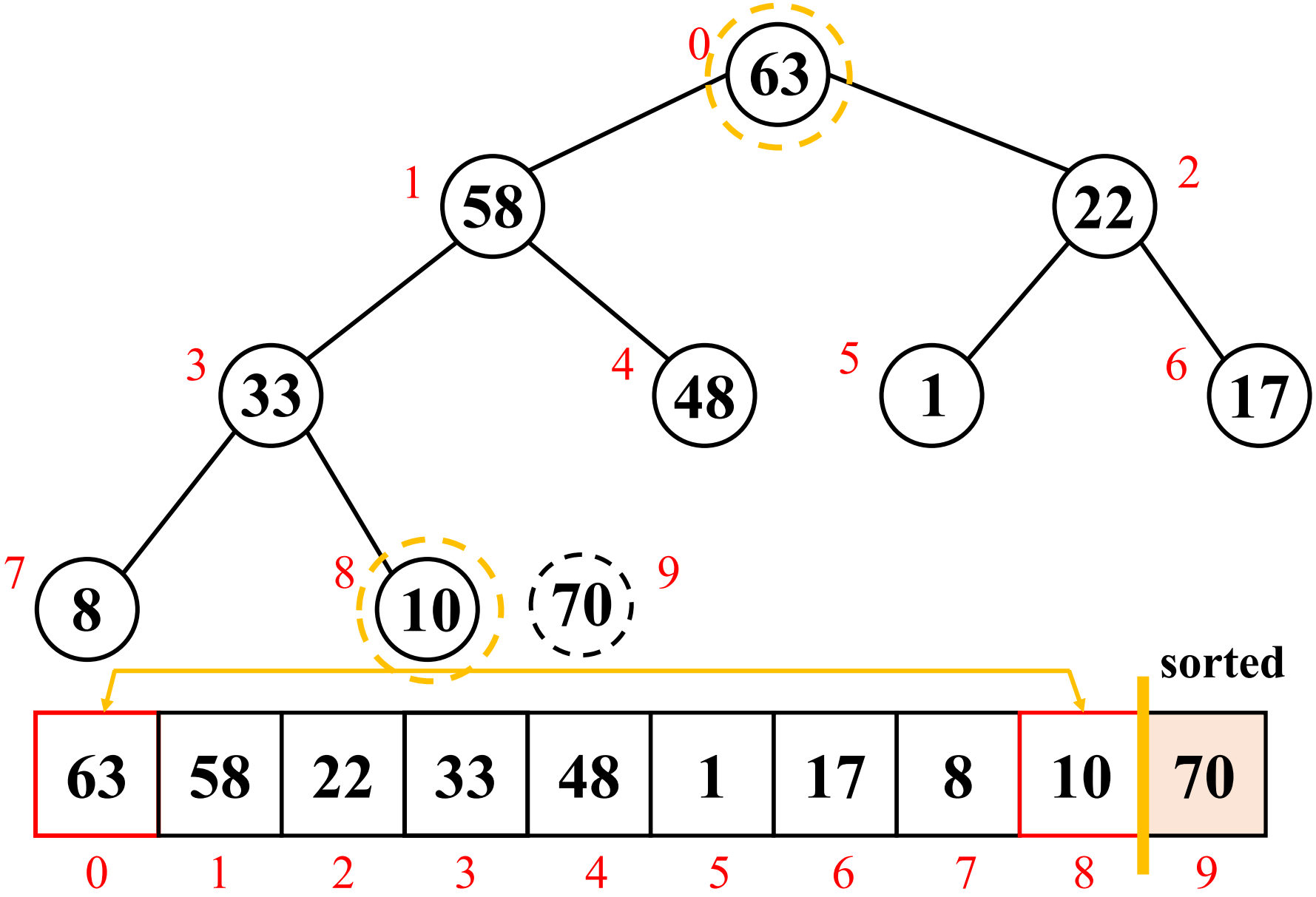


Heapsort algorithms: array implementation

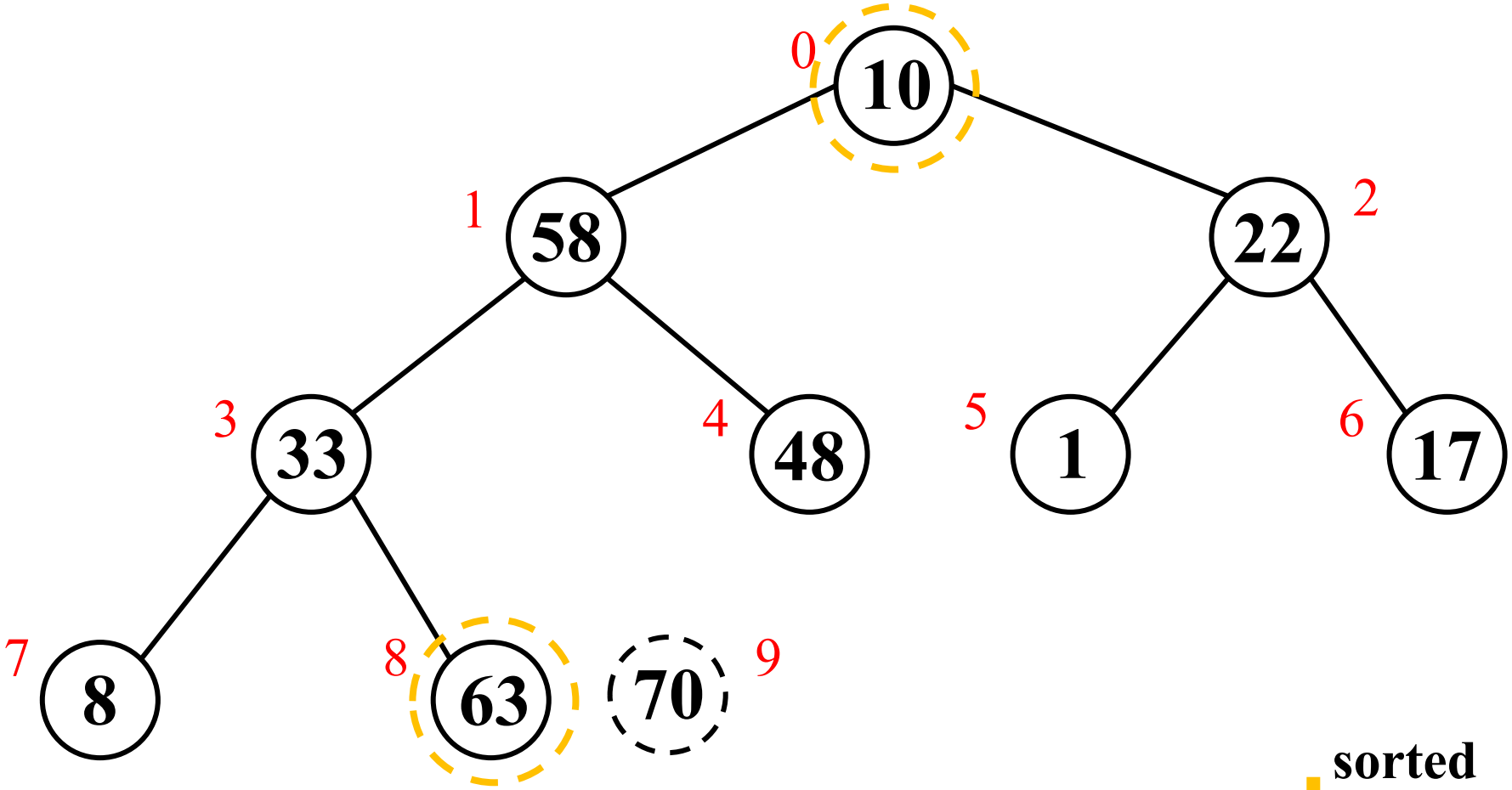




Heapsort algorithms: array implementation



Heapsort algorithms: array implementation

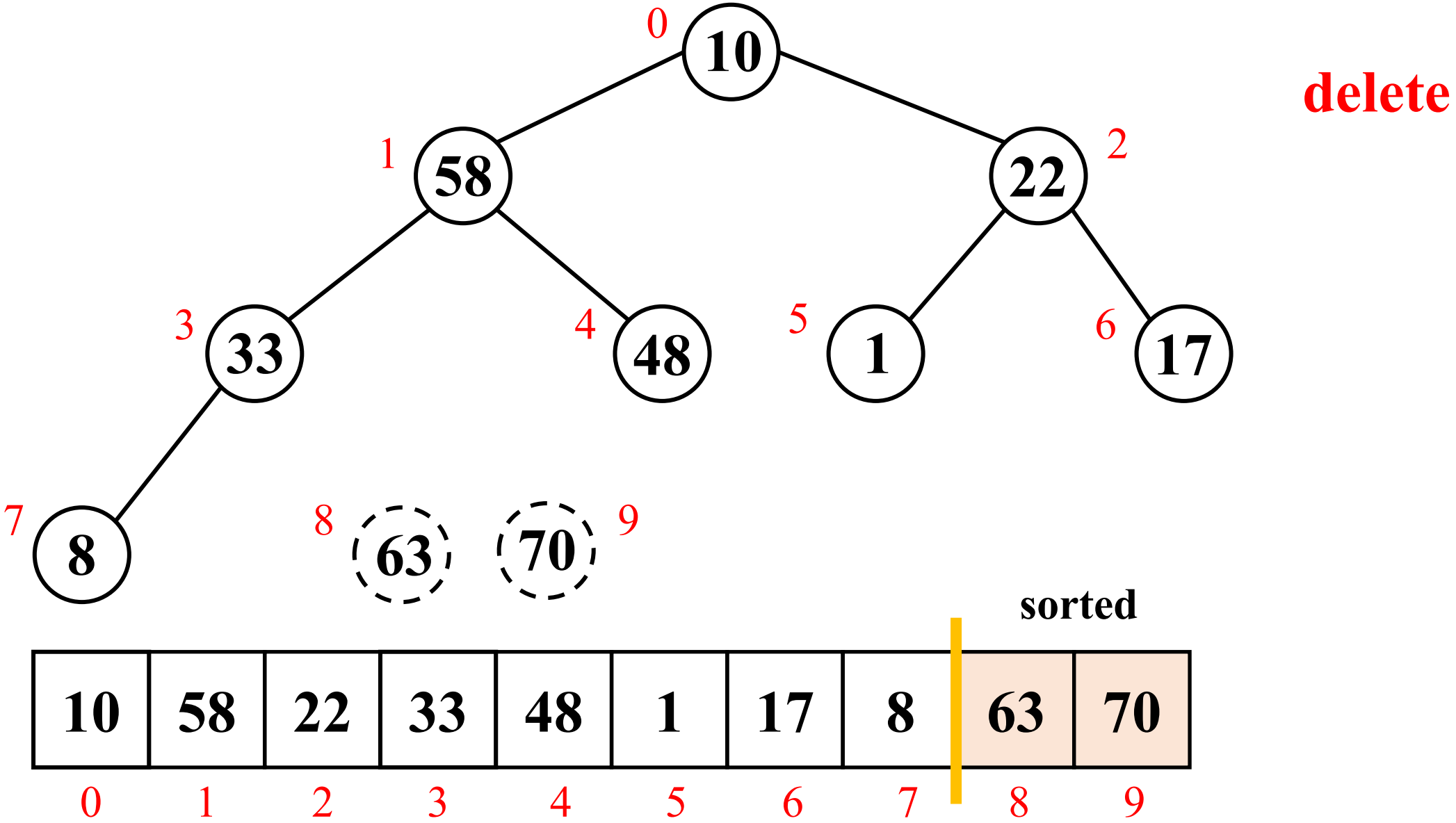


swap

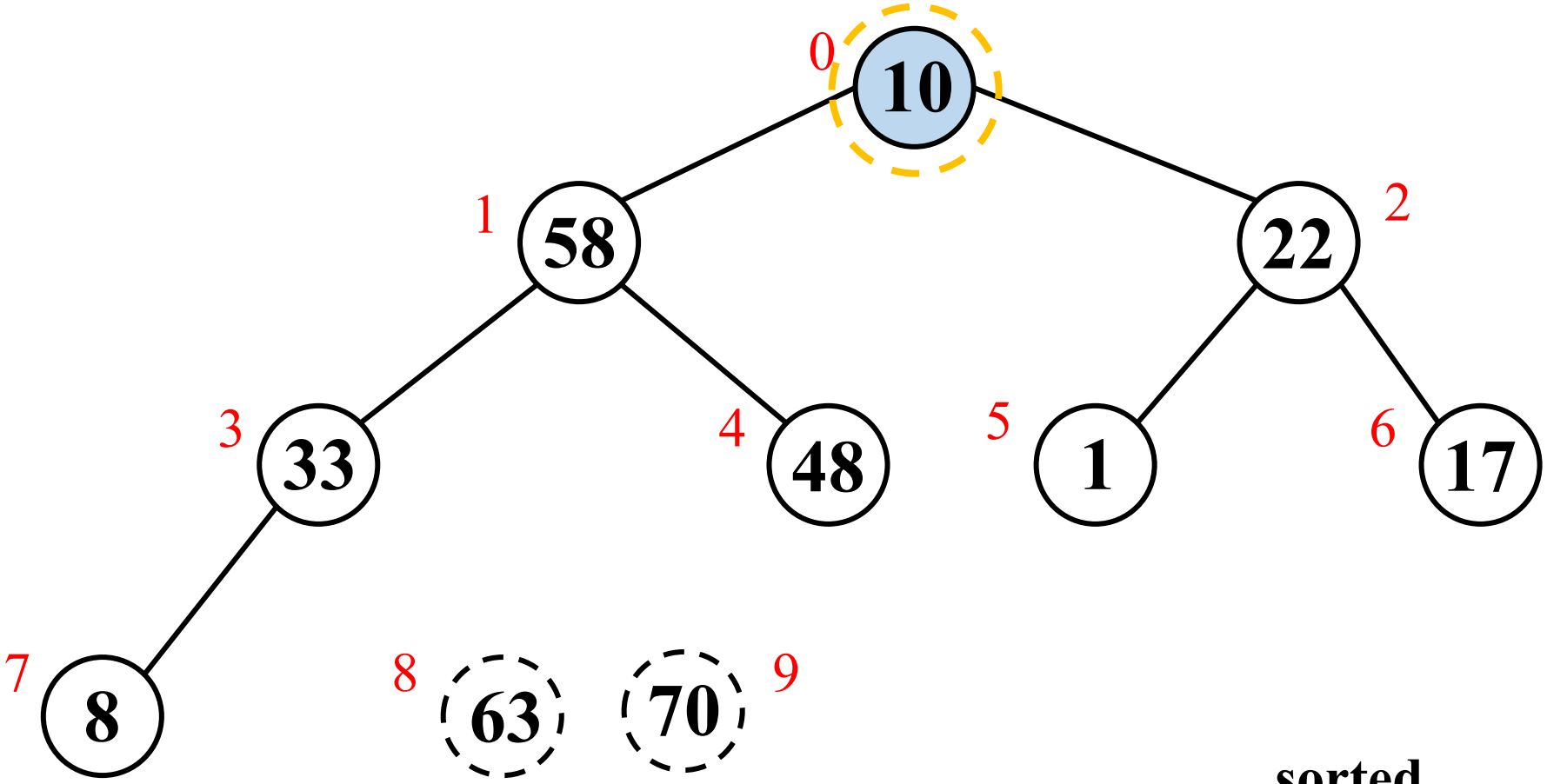
sorted

10	58	22	33	48	1	17	8	63	70
0	1	2	3	4	5	6	7	8	9

Heapsort algorithms: array implementation



Heapsort algorithms: array implementation

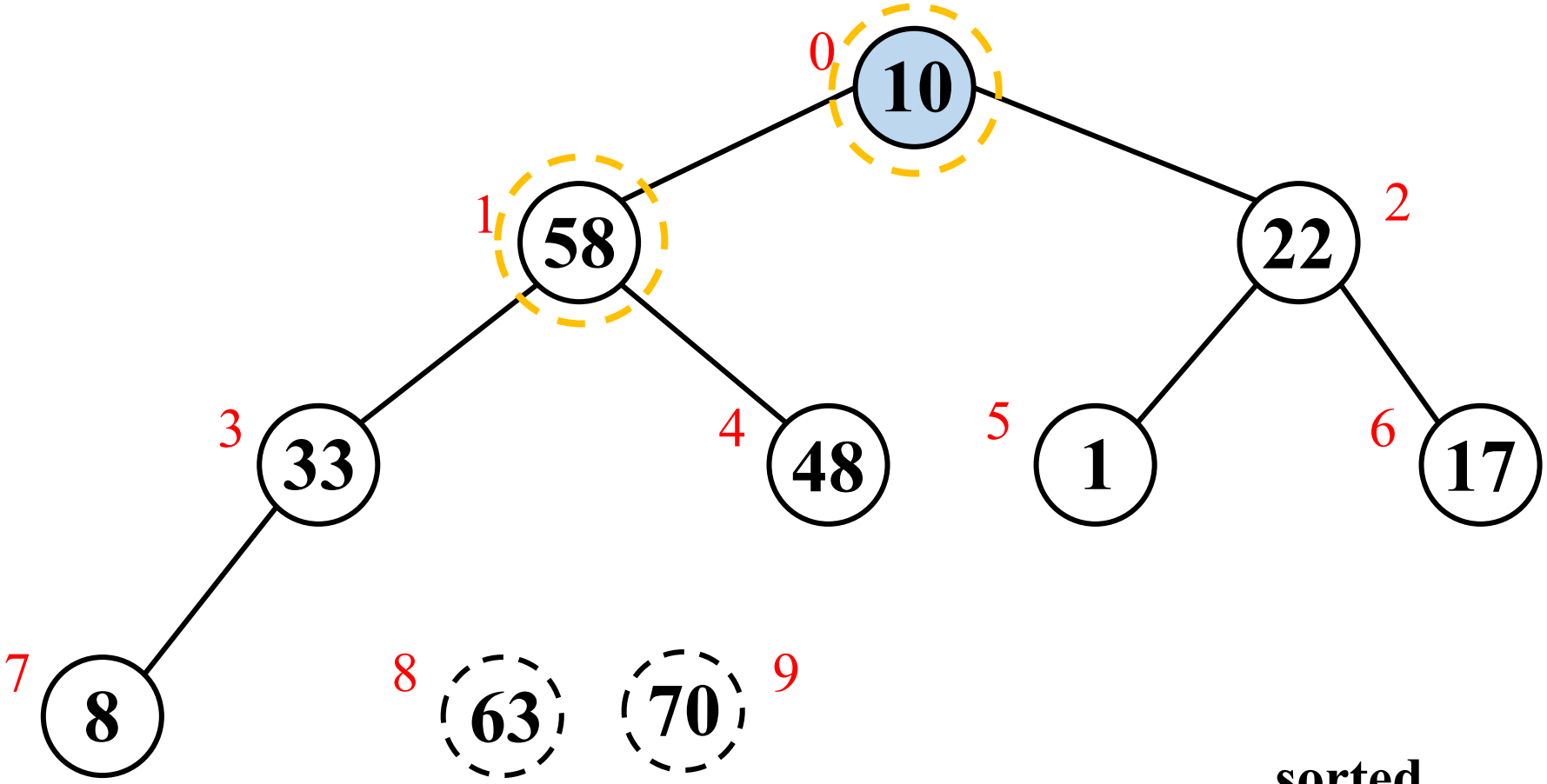


sink 10

sorted

10	58	22	33	48	1	17	8	63	70
0	1	2	3	4	5	6	7	8	9

Heapsort algorithms: array implementation

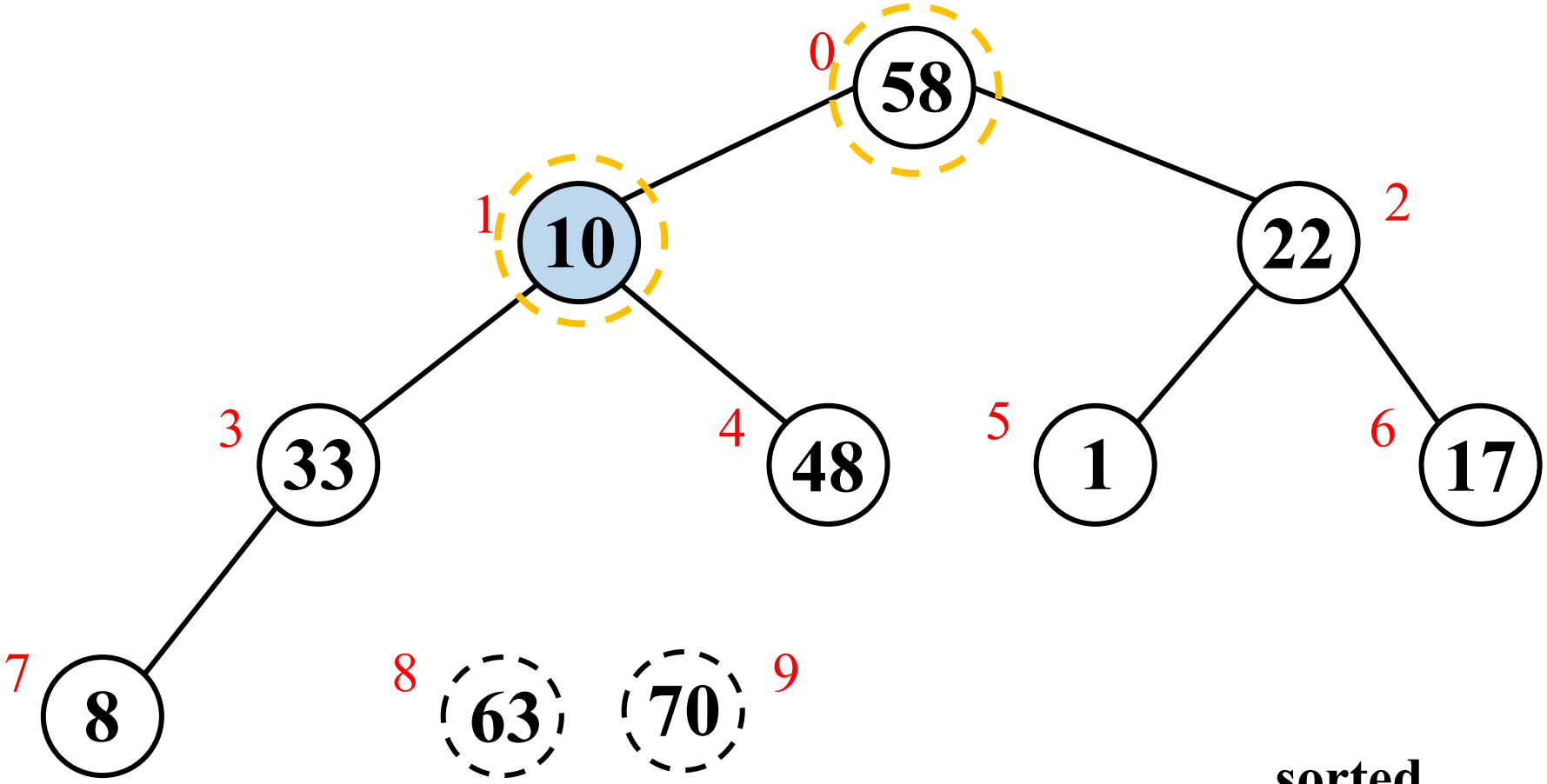


sink 10

sorted

10	58	22	33	48	1	17	8	63	70
0	1	2	3	4	5	6	7	8	9

Heapsort algorithms: array implementation

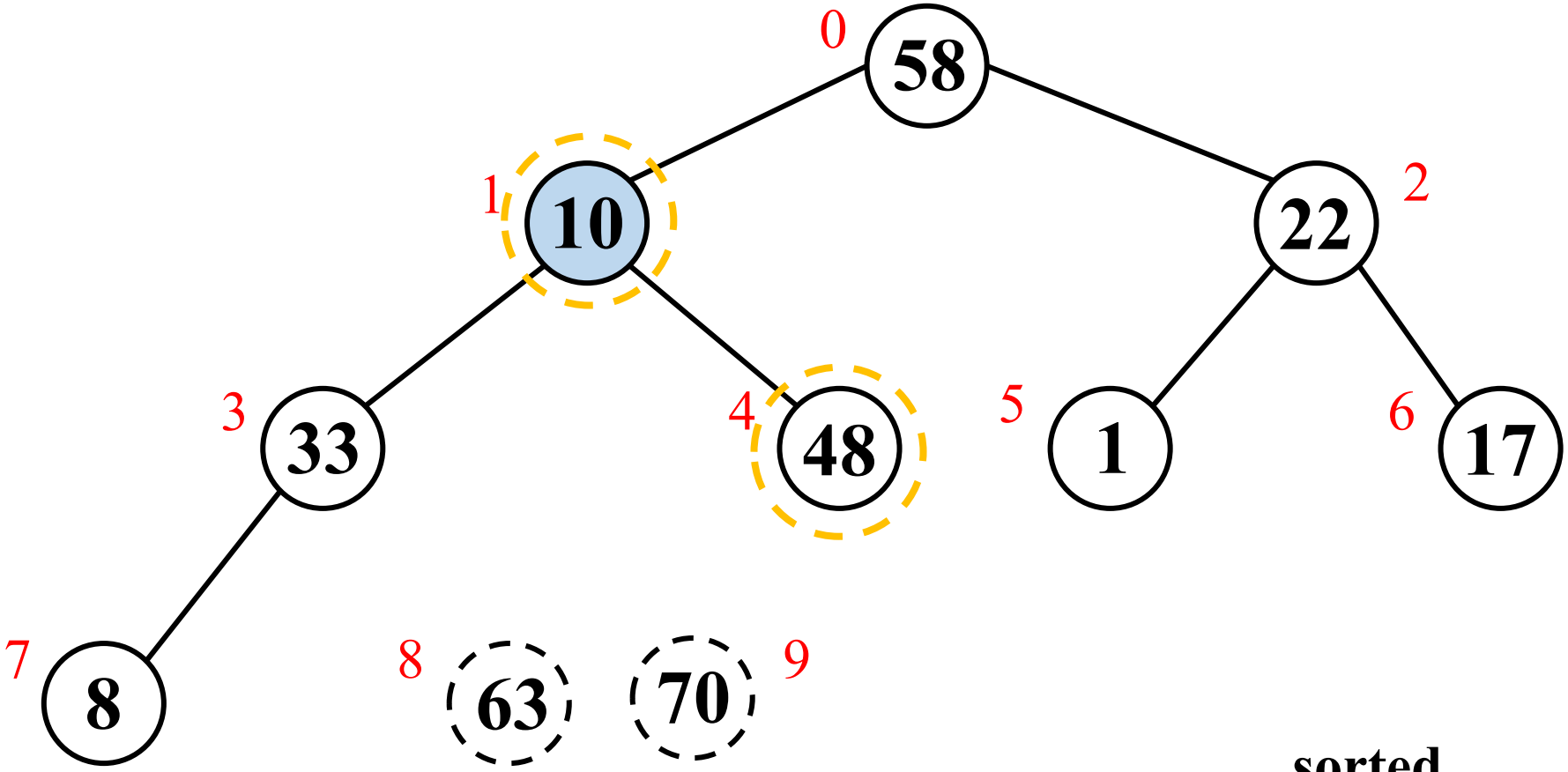


sink 10

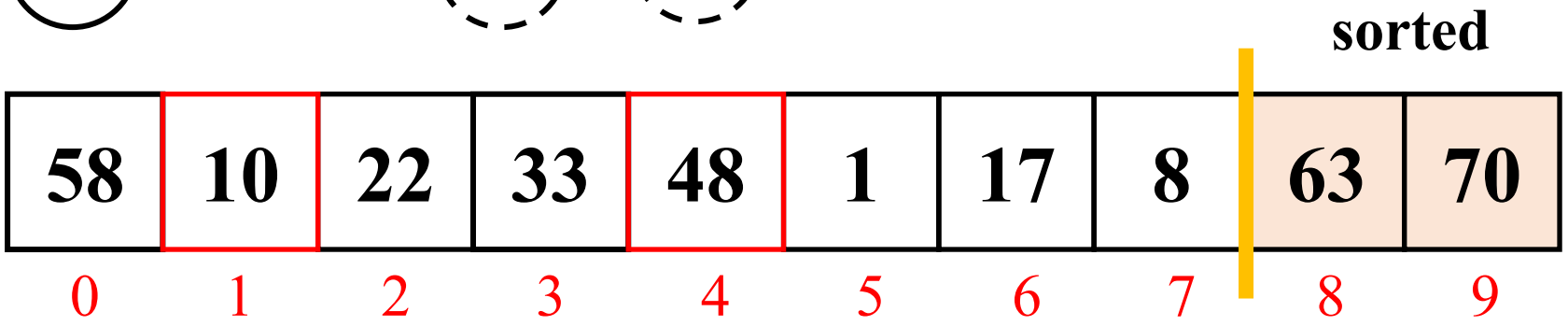
sorted

58	10	22	33	48	1	17	8	63	70
0	1	2	3	4	5	6	7	8	9

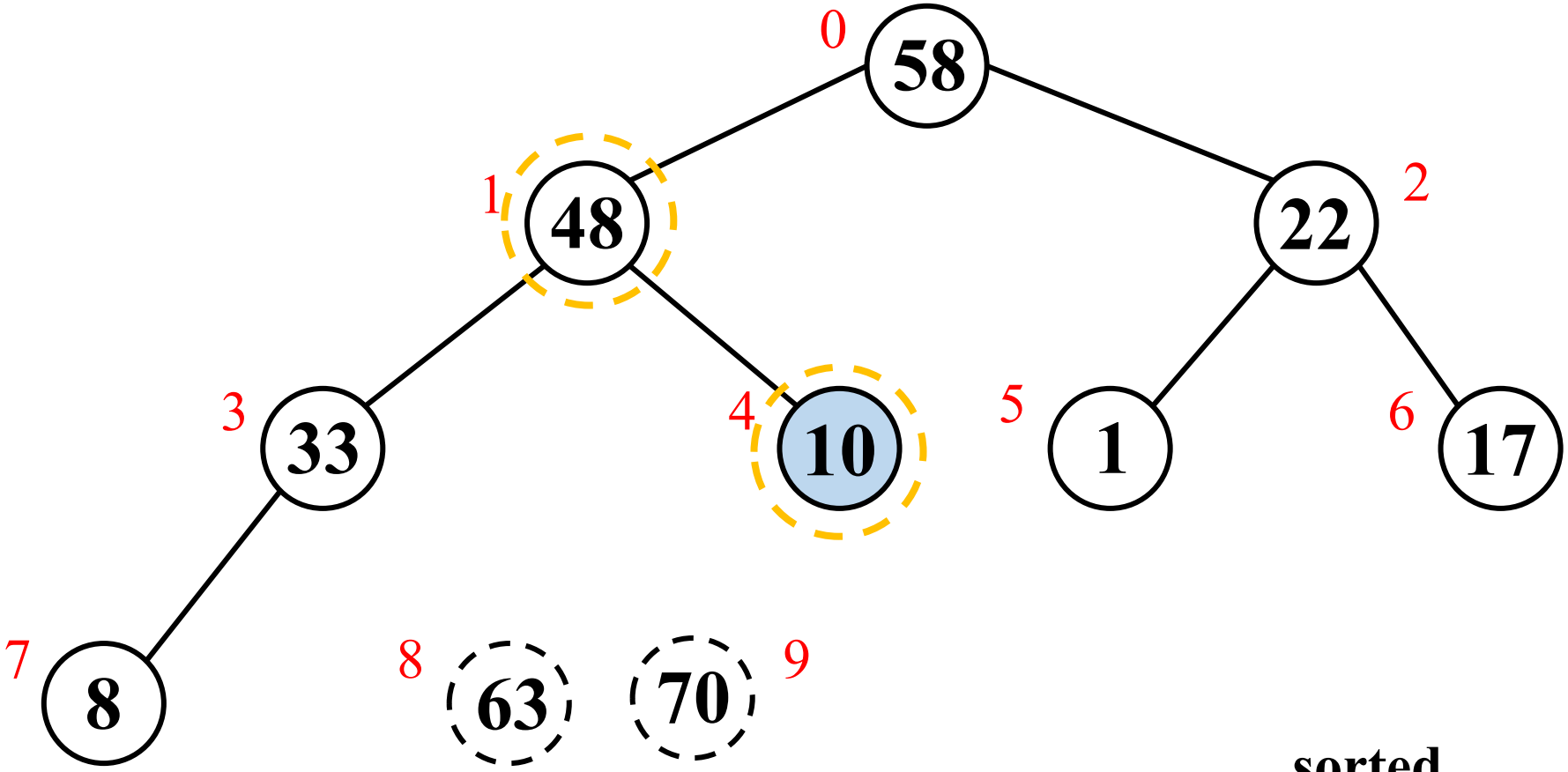
Heapsort algorithms: array implementation



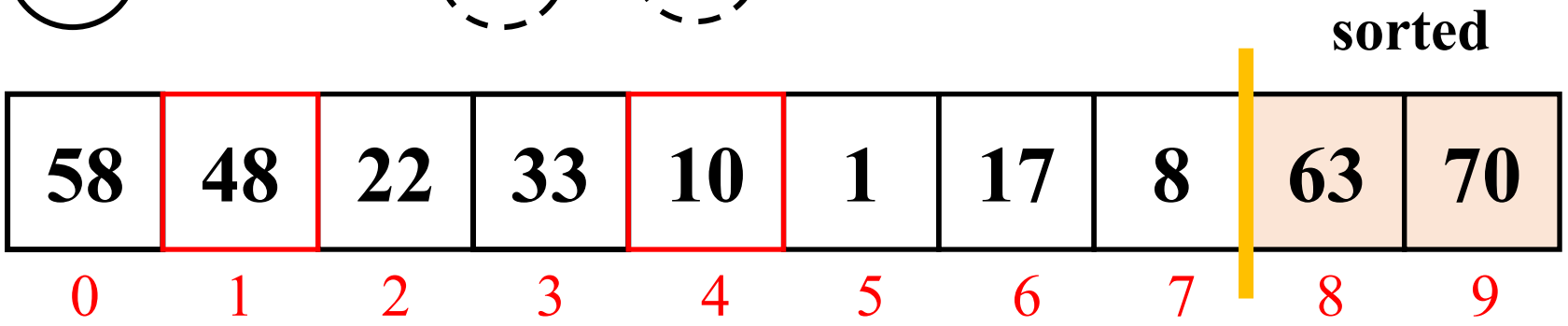
sink 10



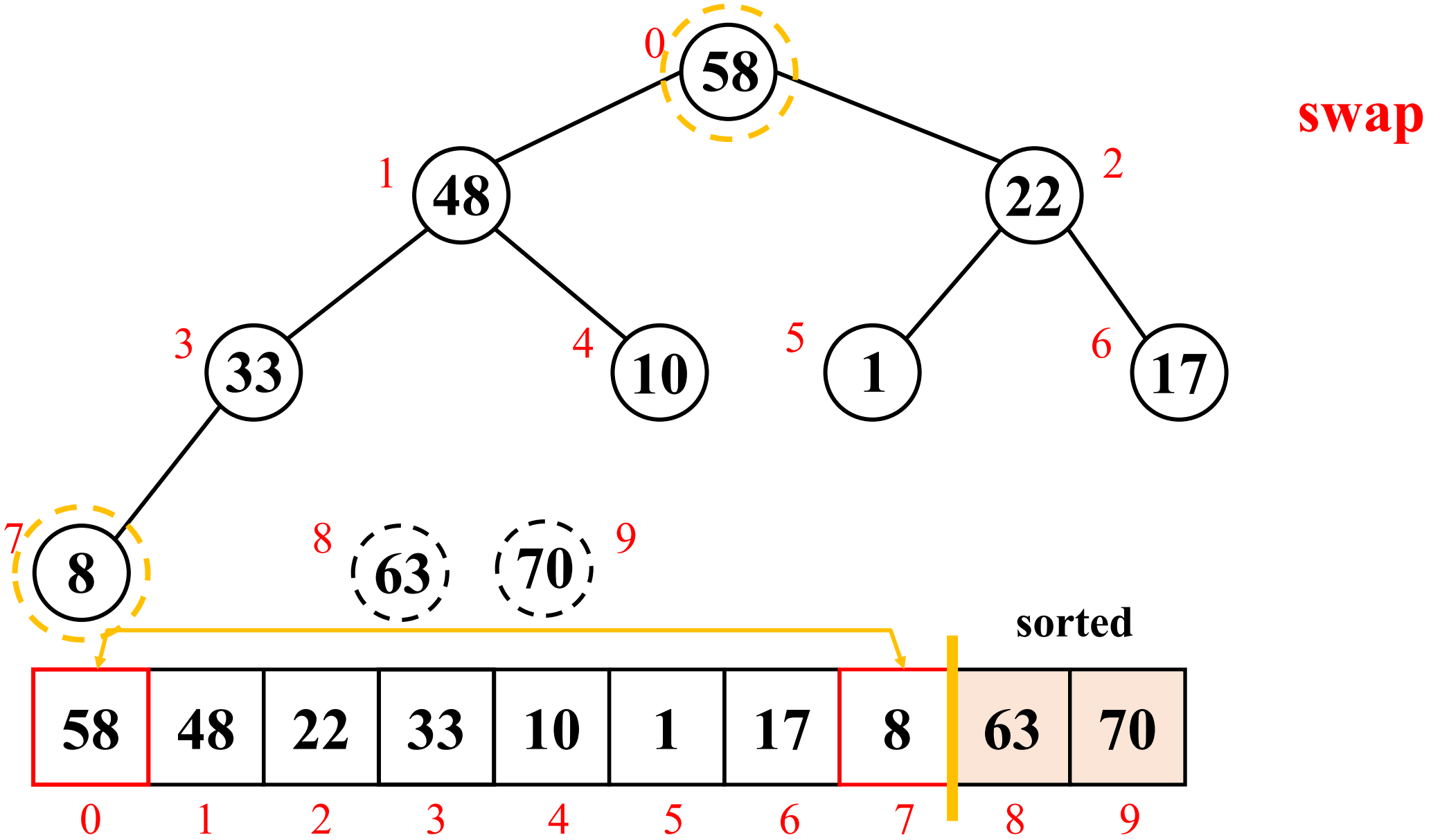
Heapsort algorithms: array implementation



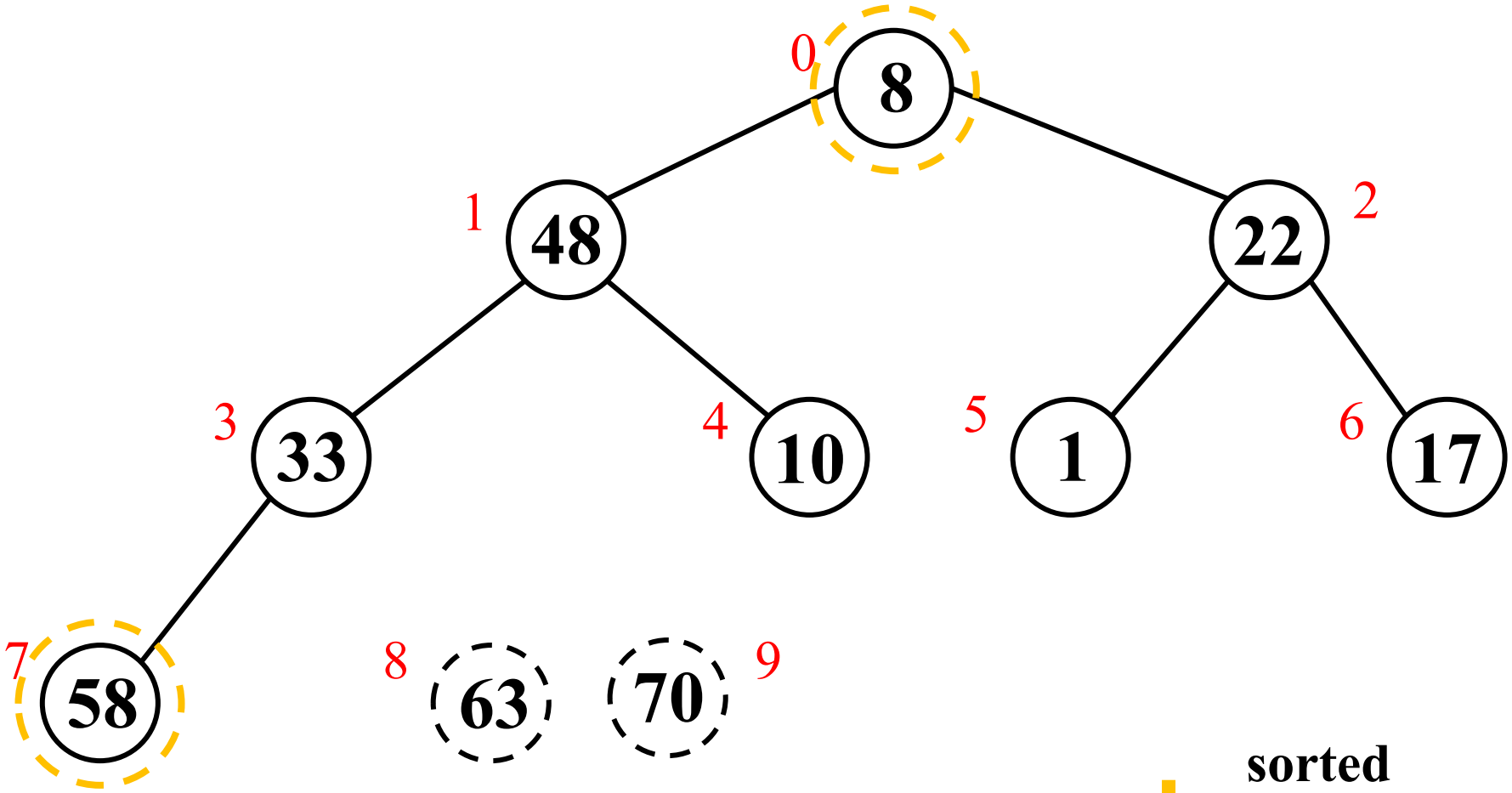
sink 10



Heapsort algorithms: array implementation



Heapsort algorithms: array implementation

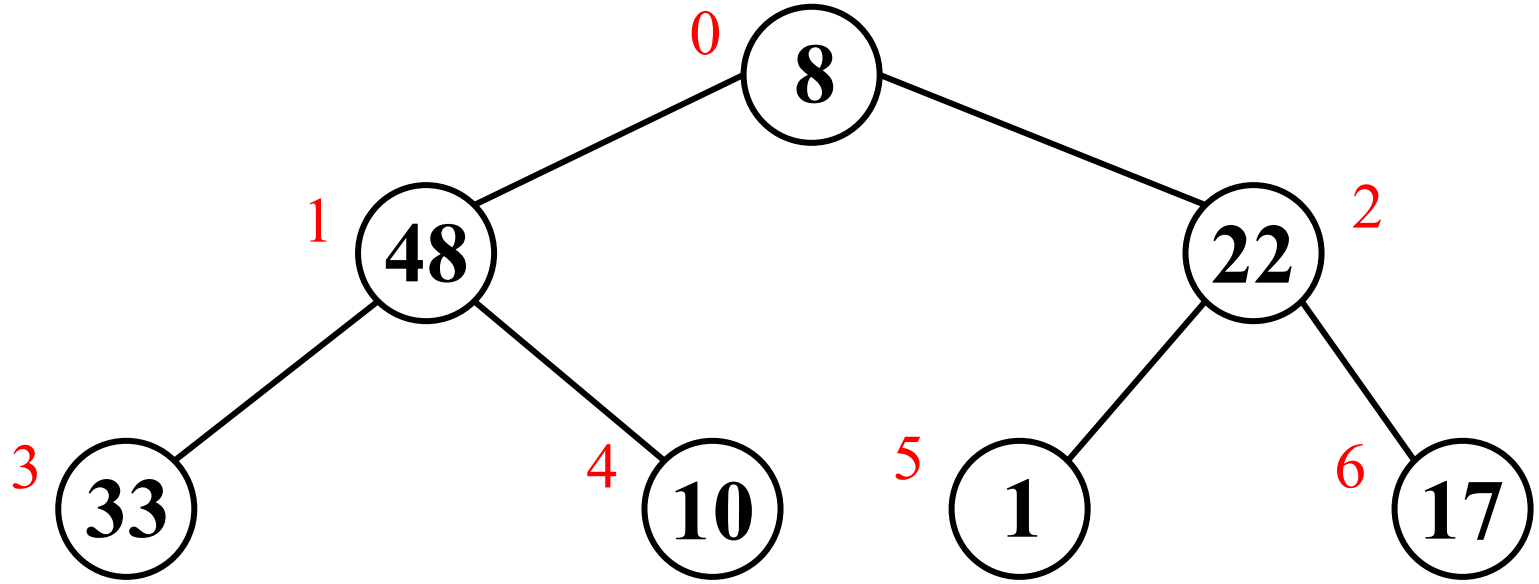


swap

sorted

8	48	22	33	10	1	17	58	63	70
0	1	2	3	4	5	6	7	8	9

Heapsort algorithms: array implementation

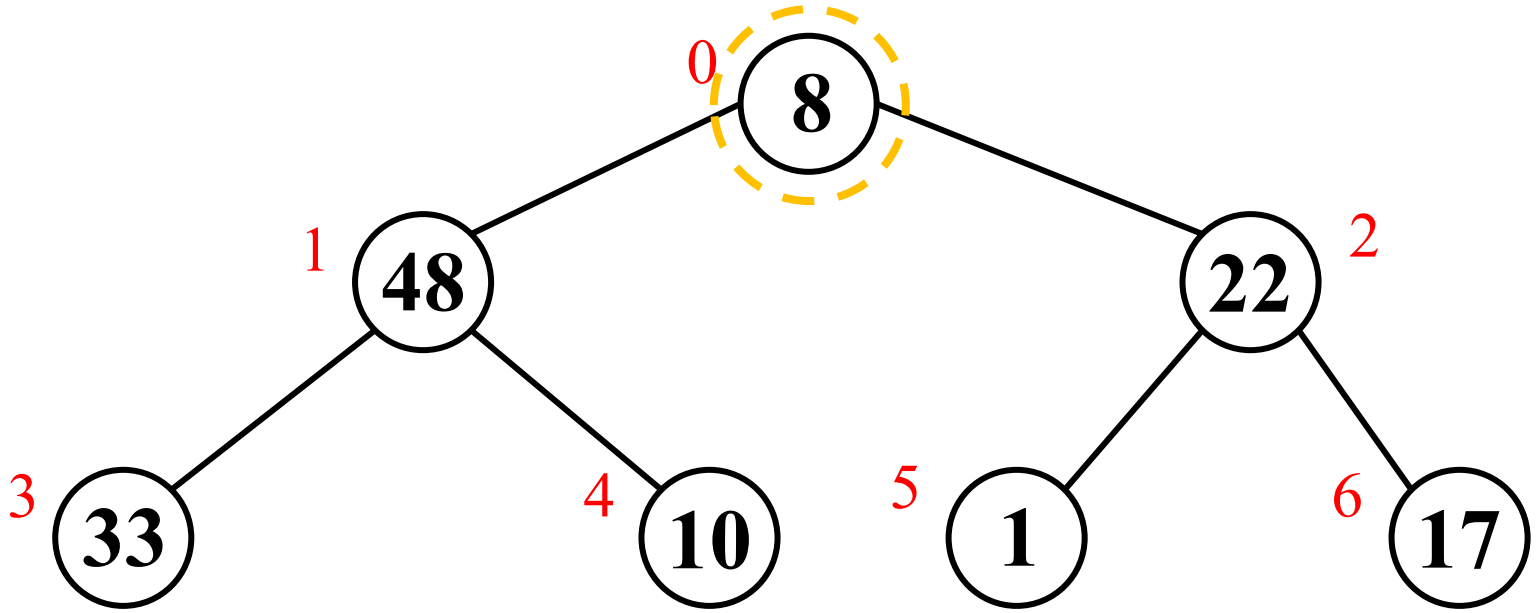


delete

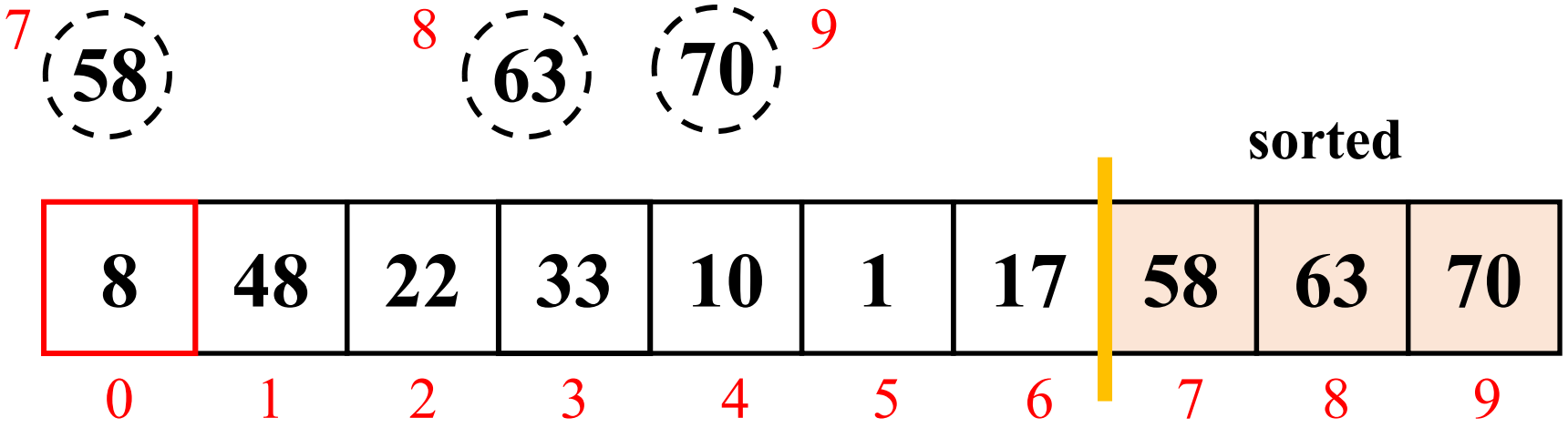
sorted

8	48	22	33	10	1	17	58	63	70
0	1	2	3	4	5	6	7	8	9

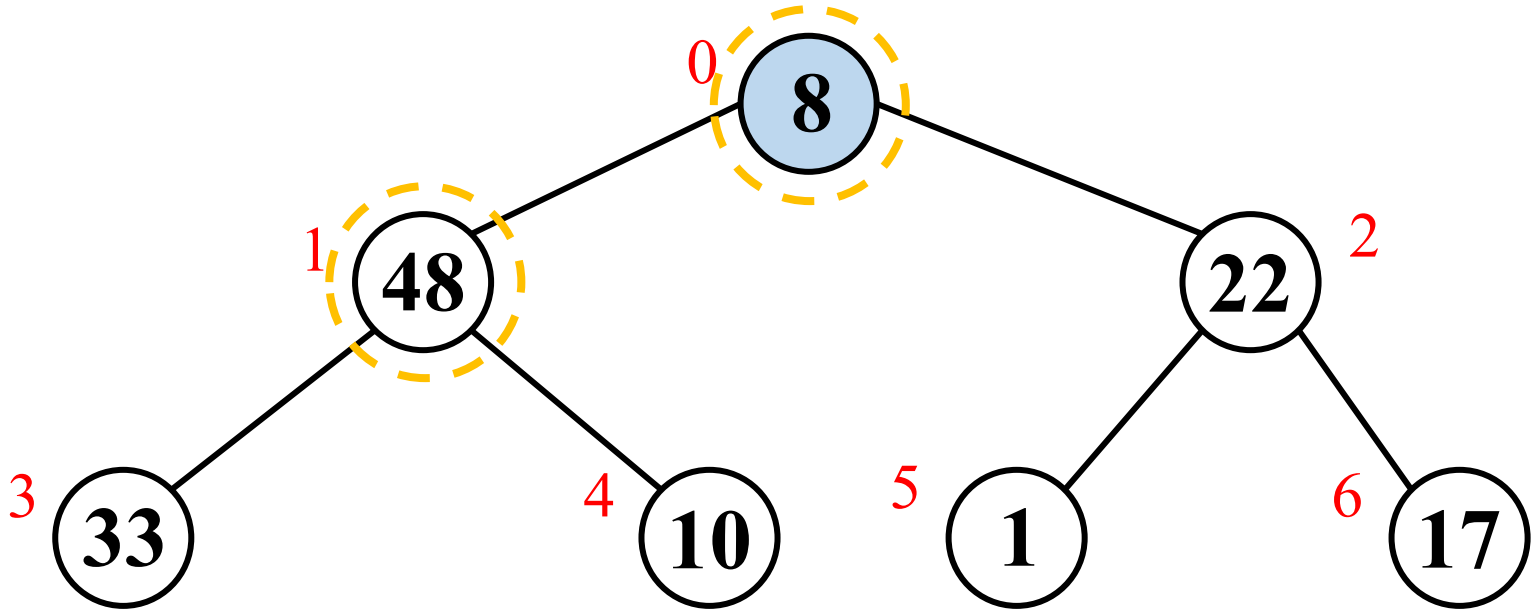
Heapsort algorithms: array implementation



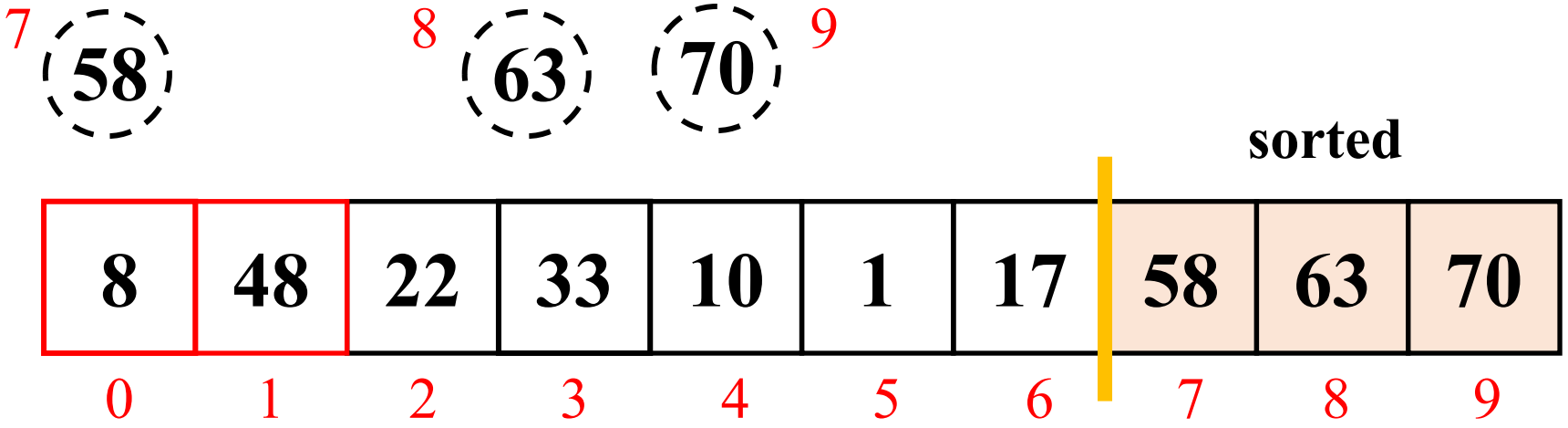
sink 8



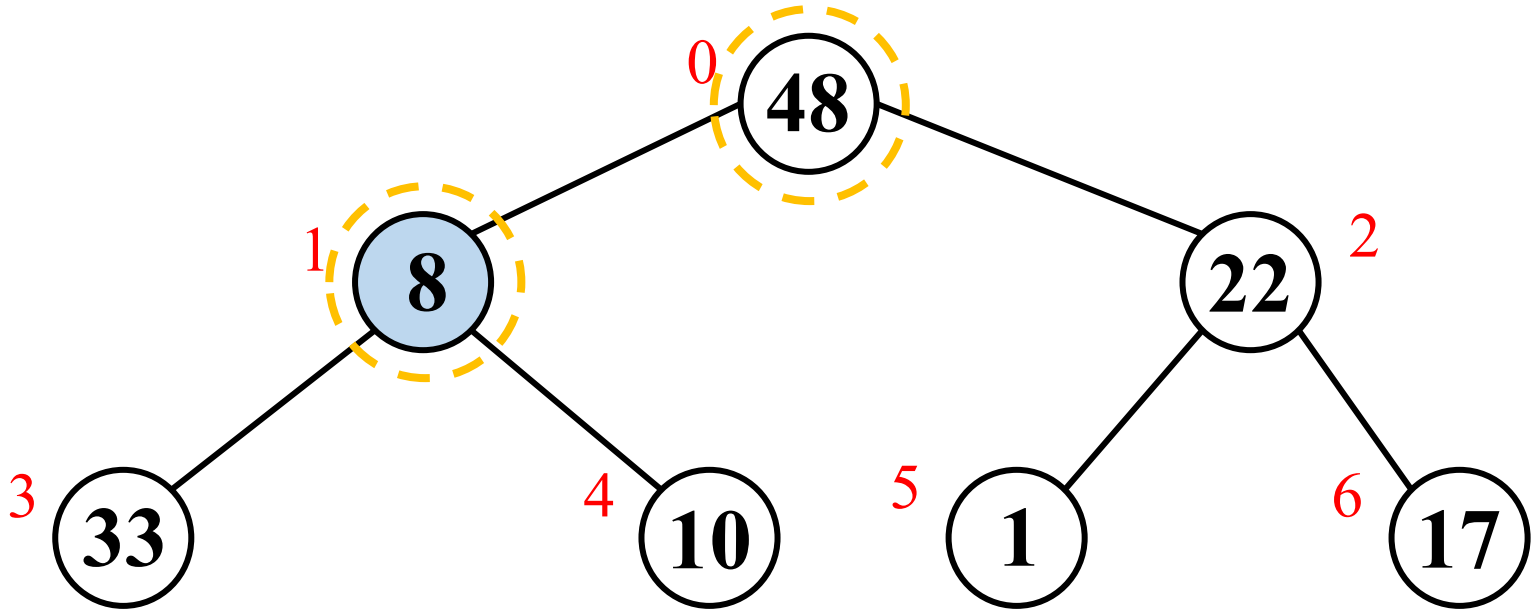
Heapsort algorithms: array implementation



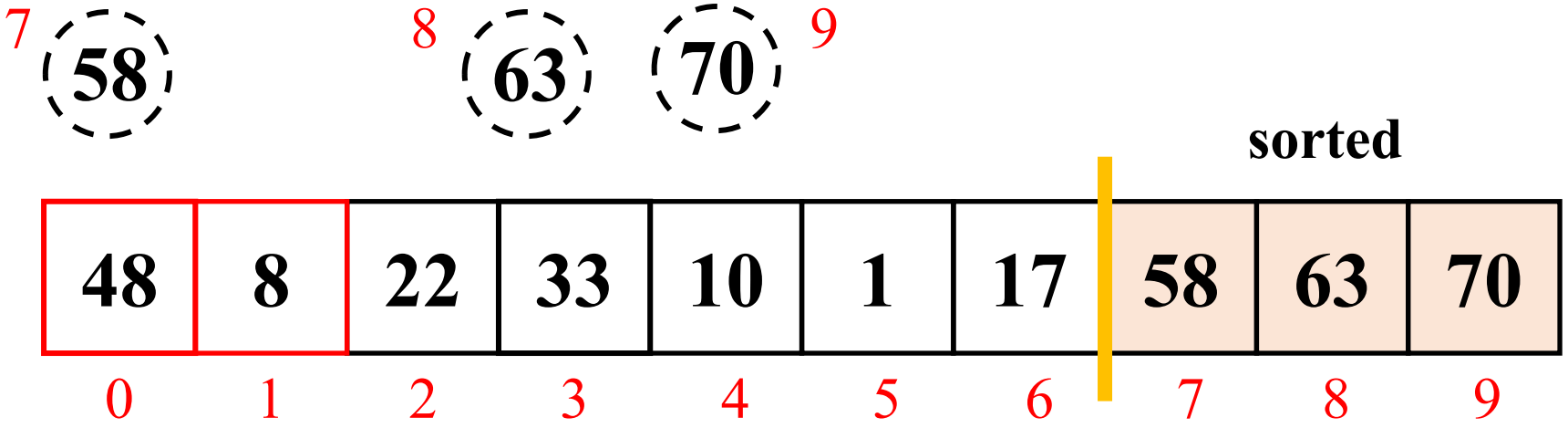
sink 8



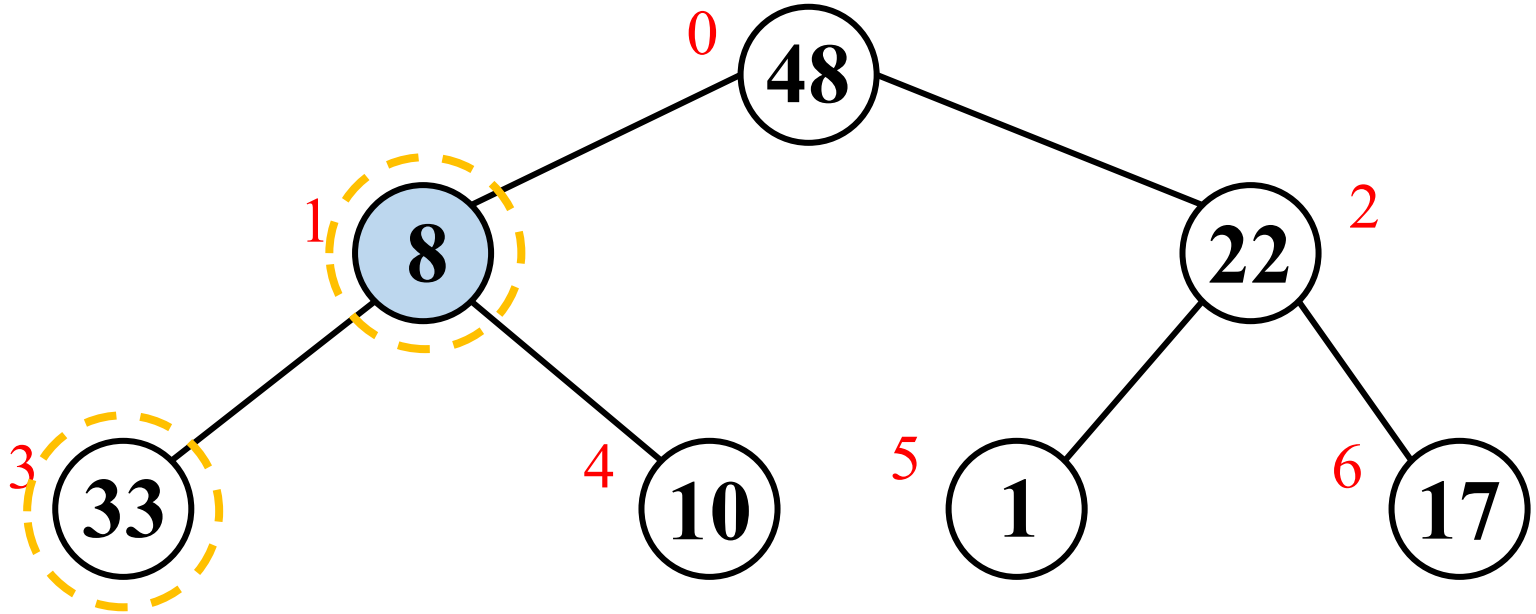
Heapsort algorithms: array implementation



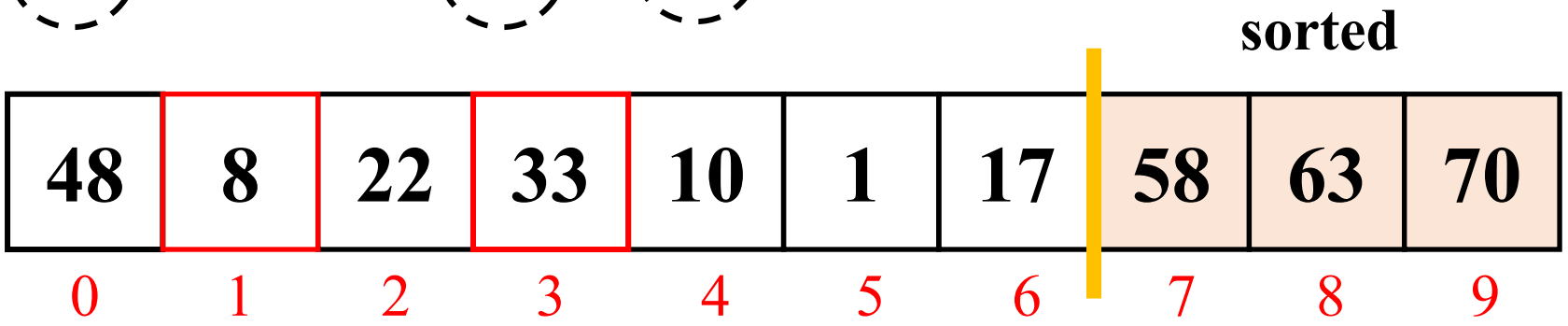
sink 8



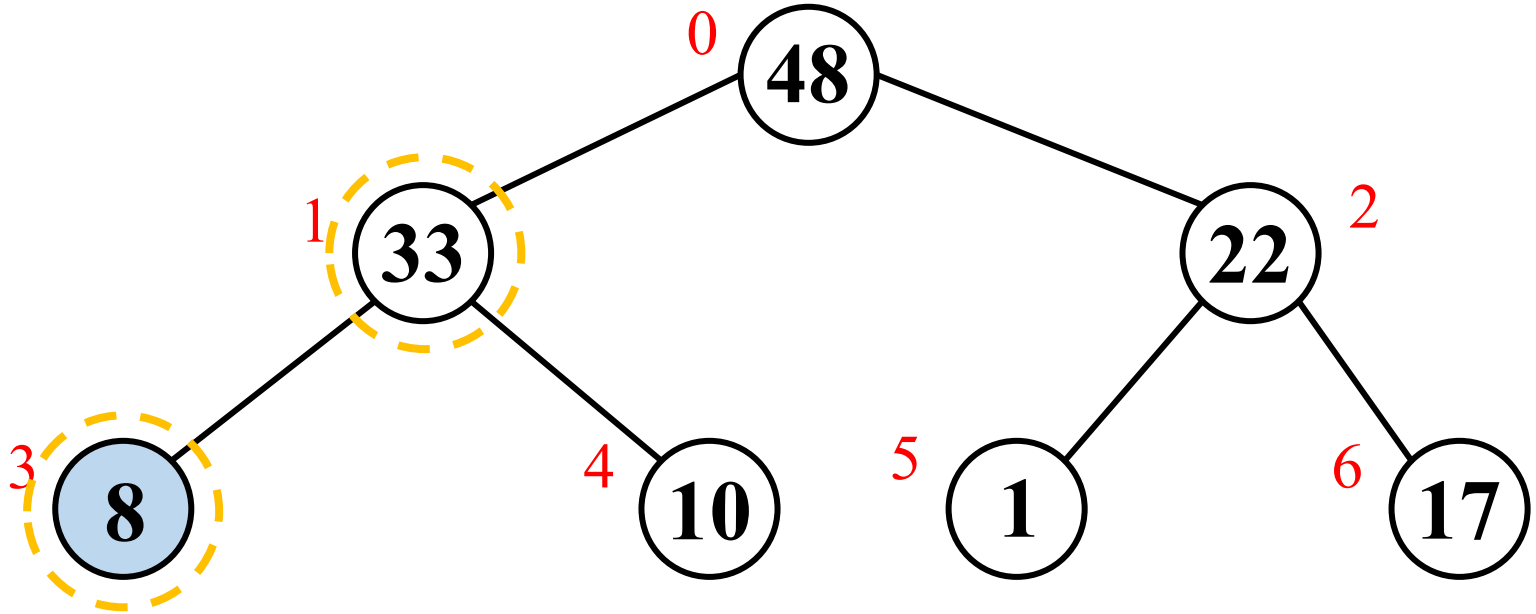
Heapsort algorithms: array implementation



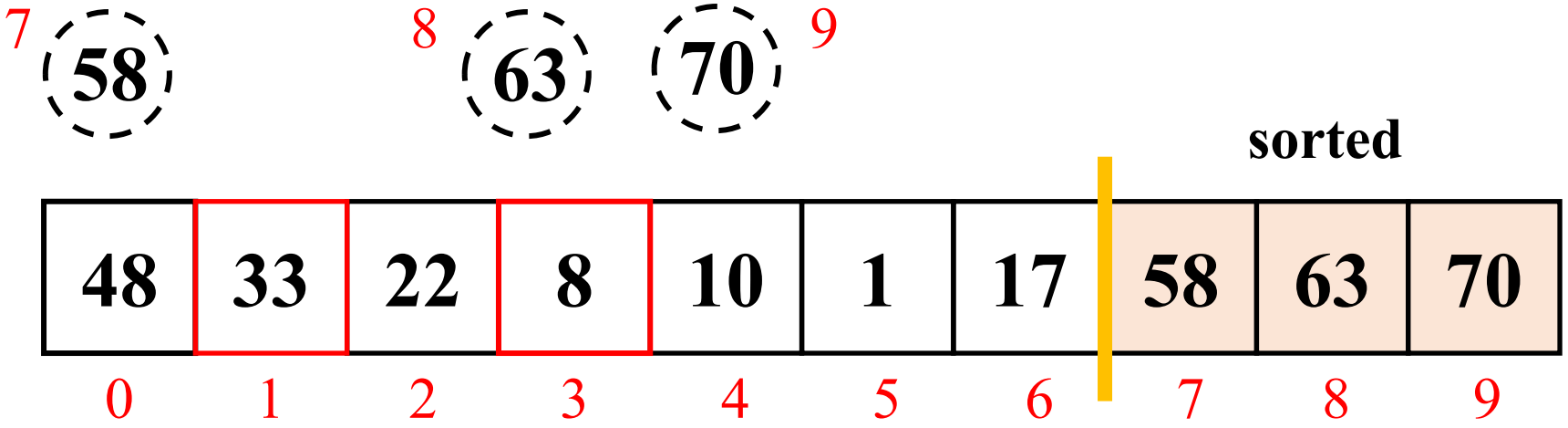
sink 8



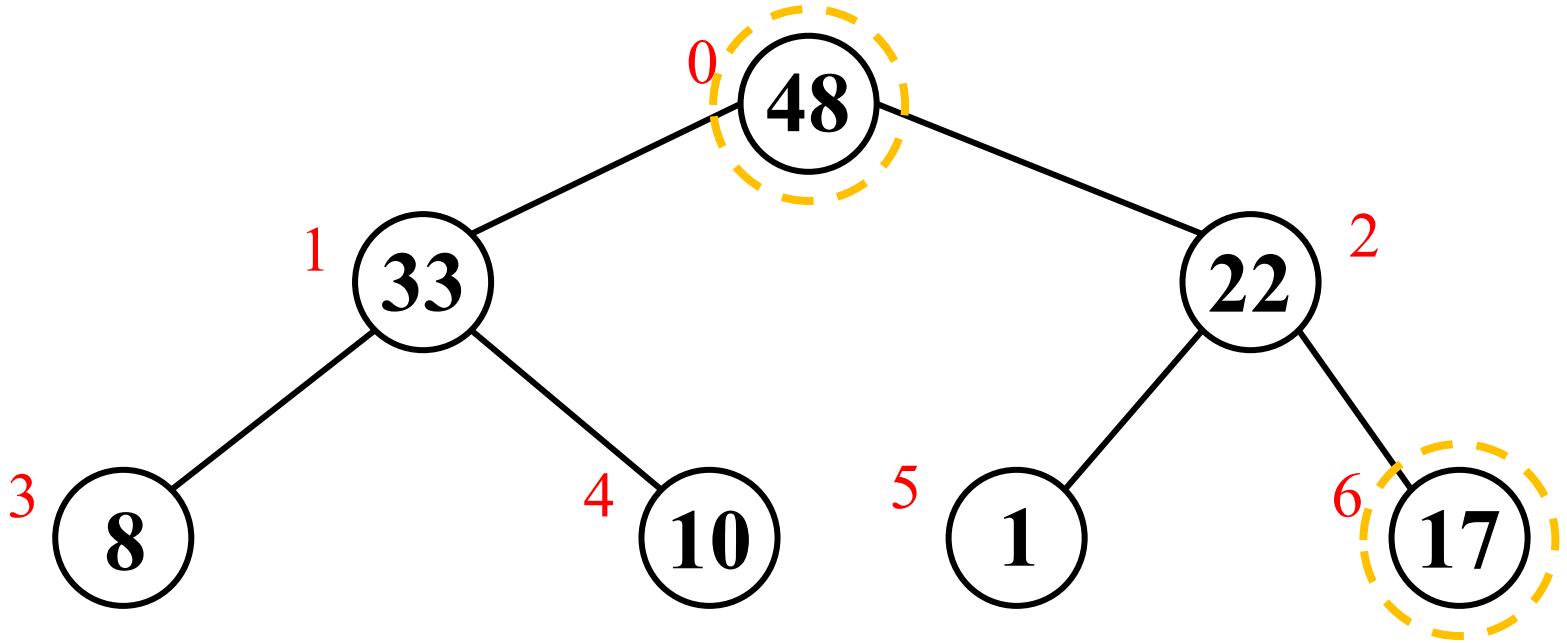
Heapsort algorithms: array implementation



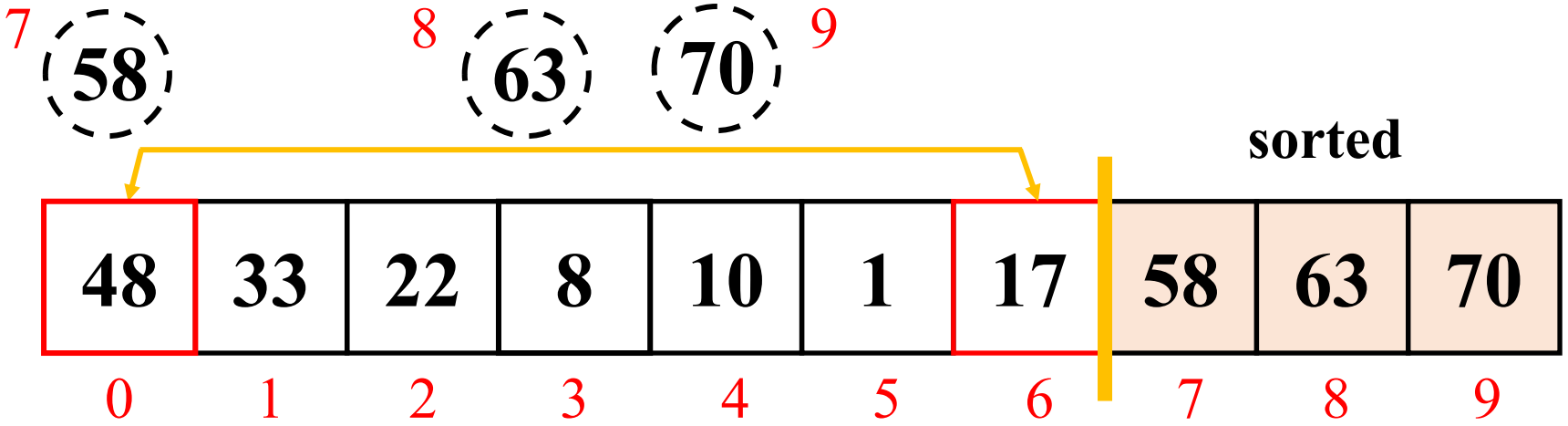
sink 8



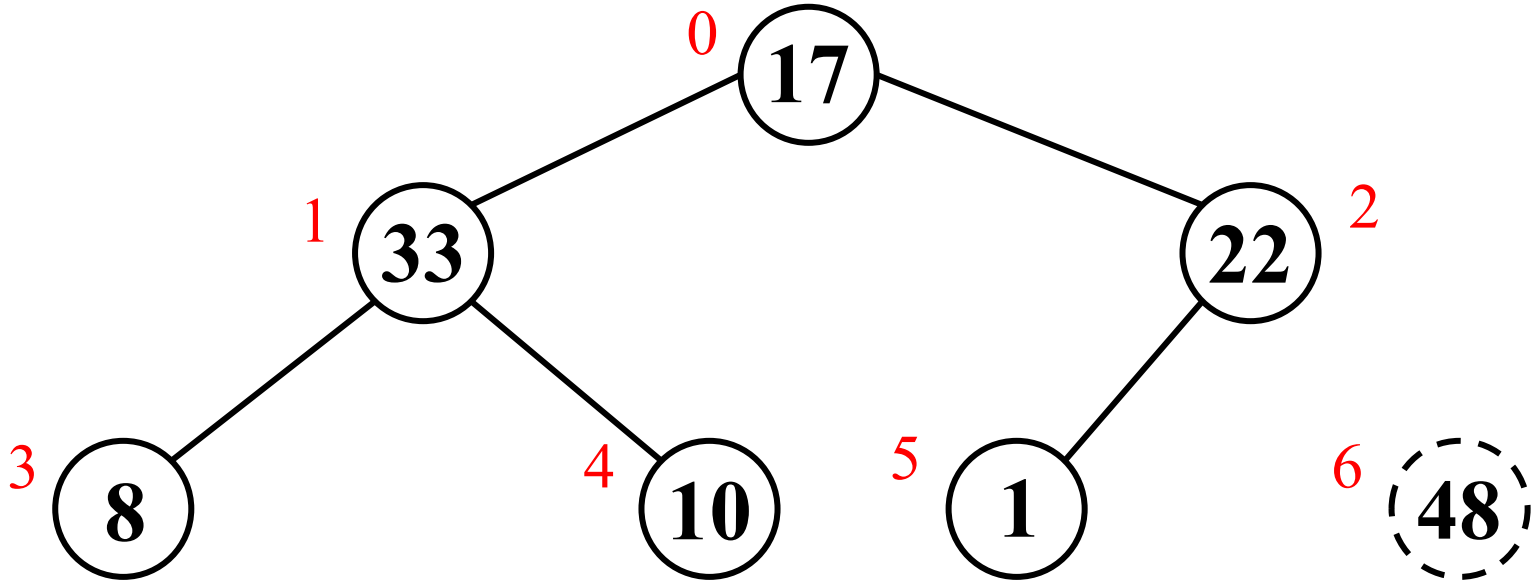
Heapsort algorithms: array implementation

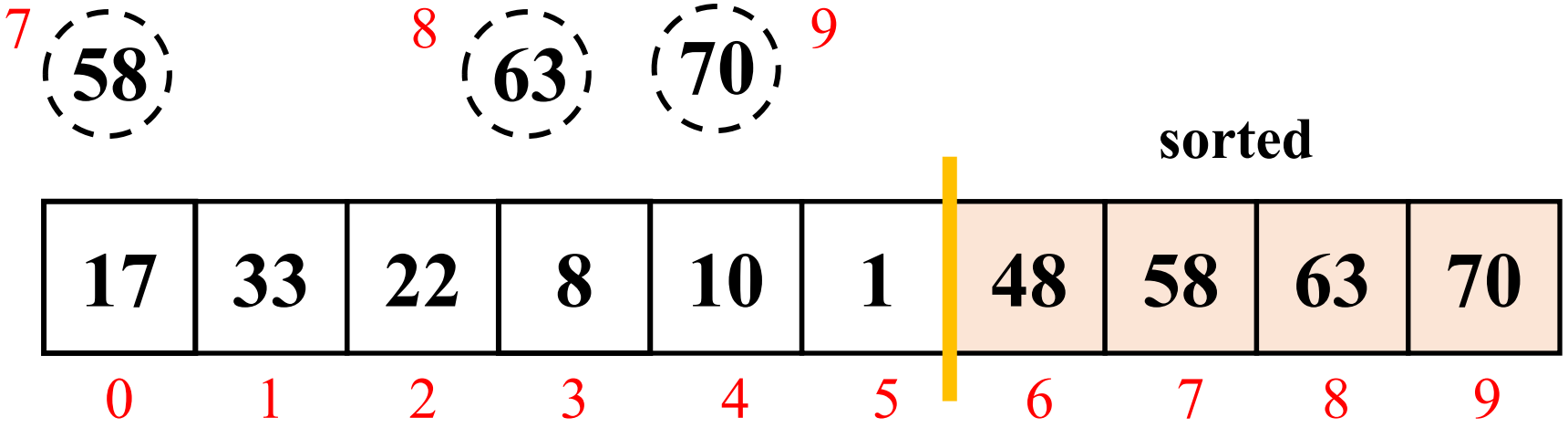


swap

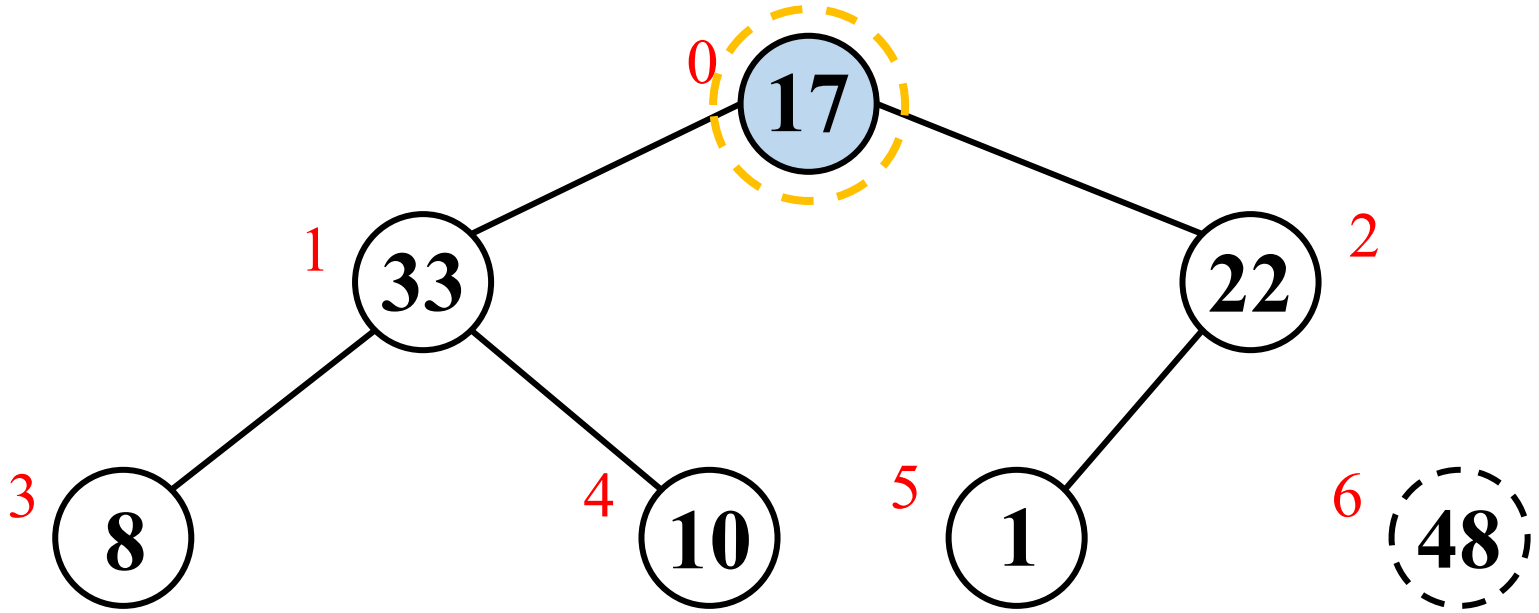


Heapsort algorithms: array implementation

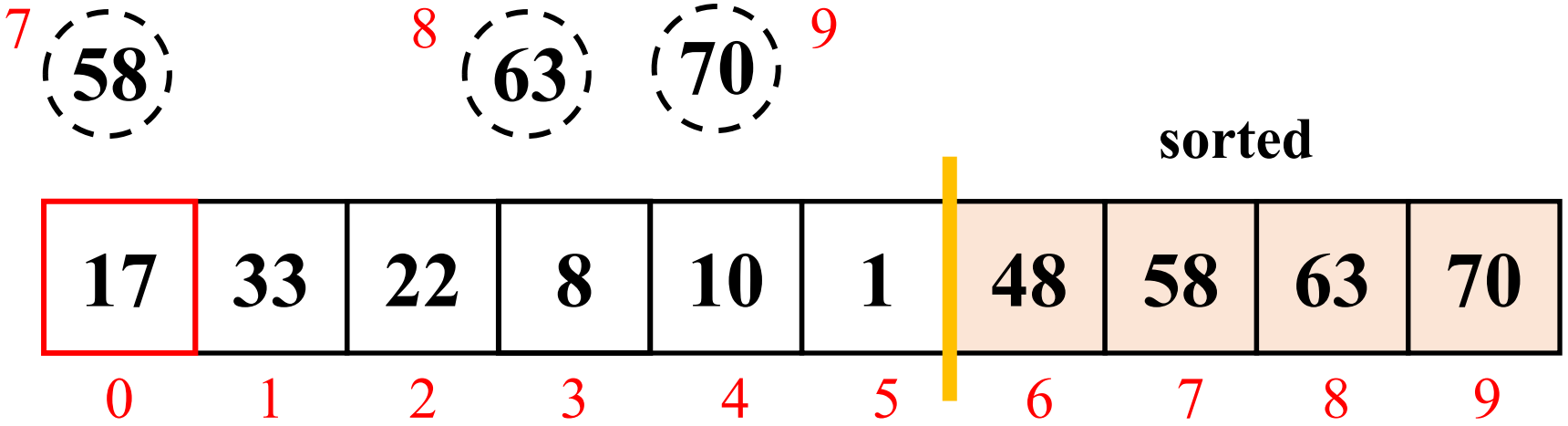




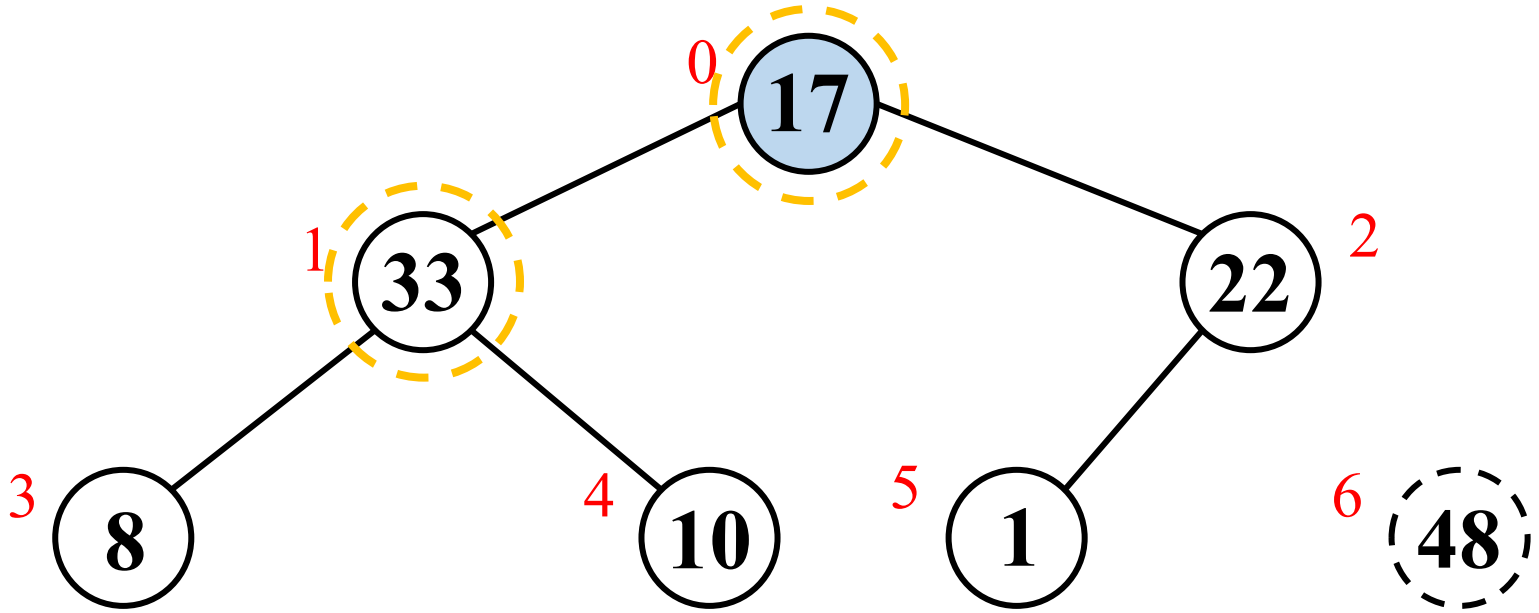
Heapsort algorithms: array implementation



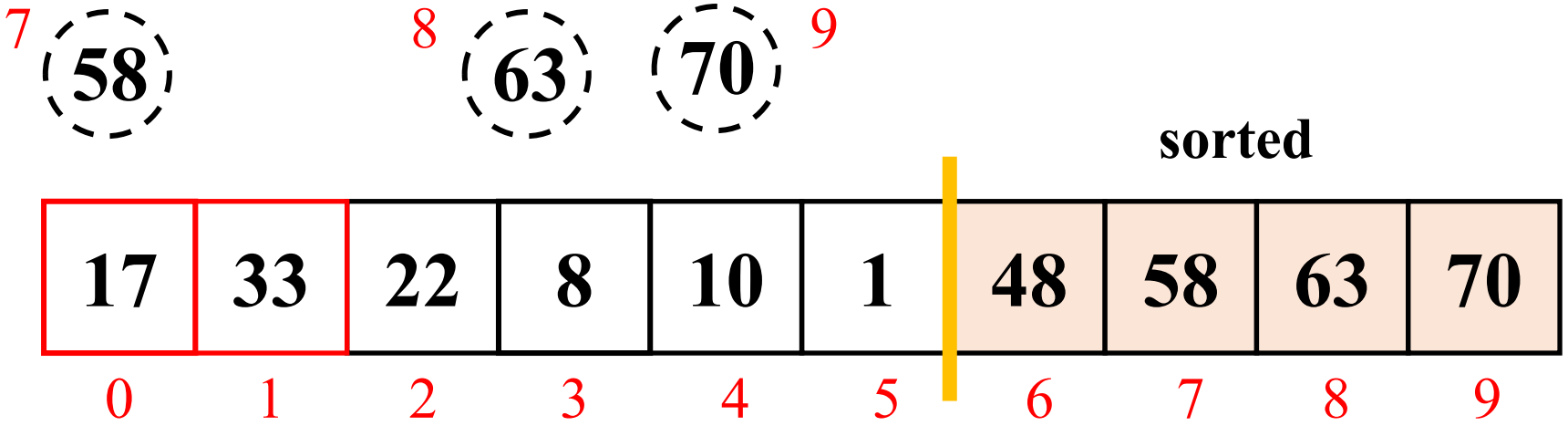
sink 17



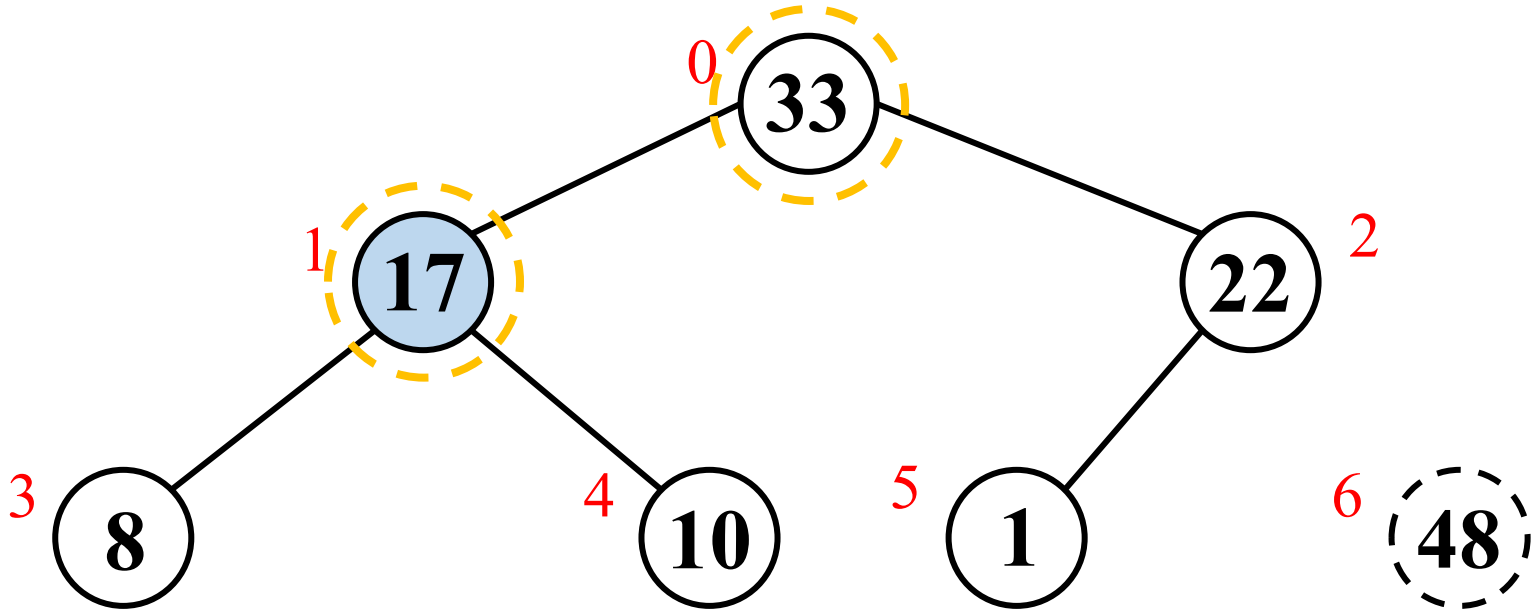
Heapsort algorithms: array implementation



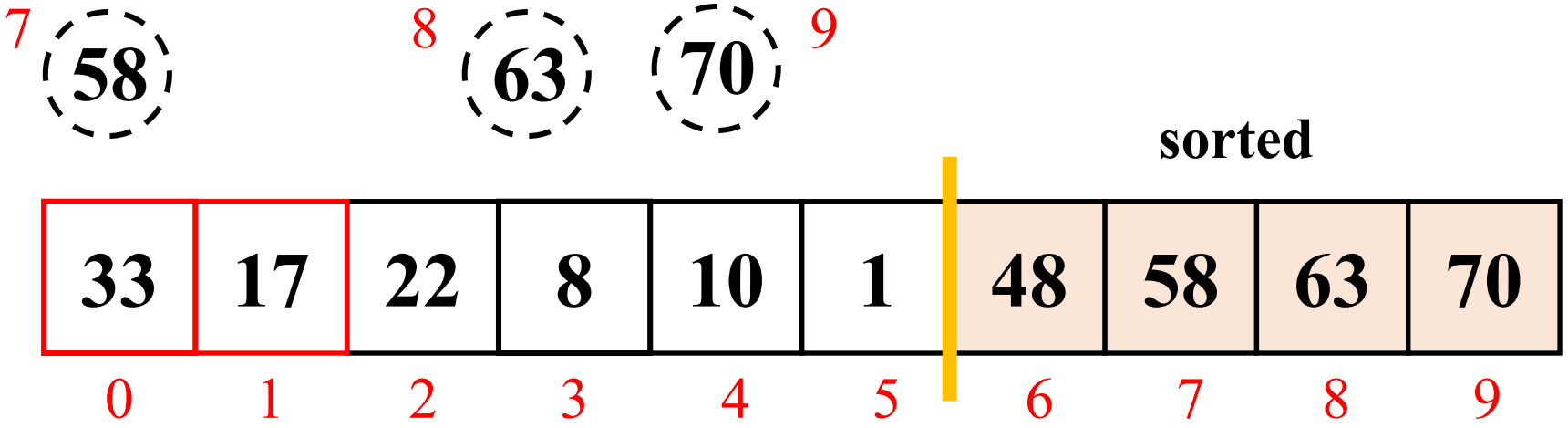
sink 17



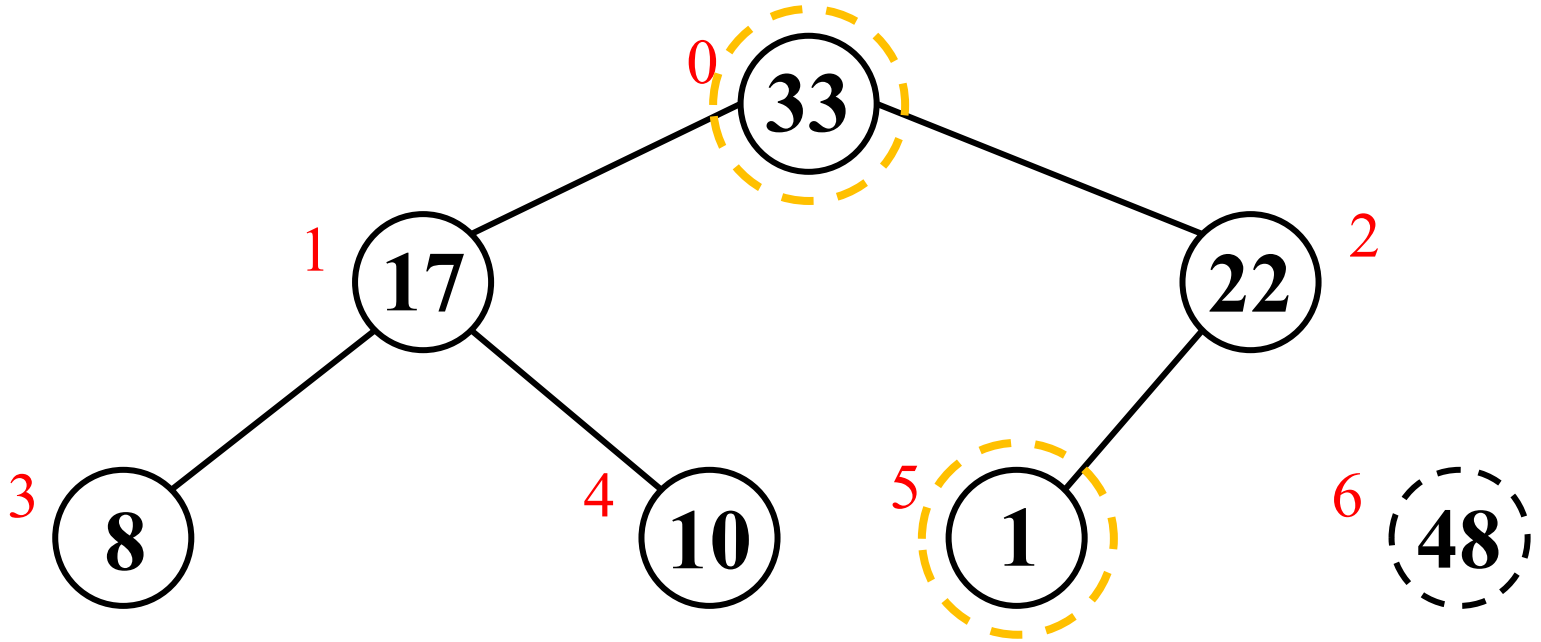
Heapsort algorithms: array implementation



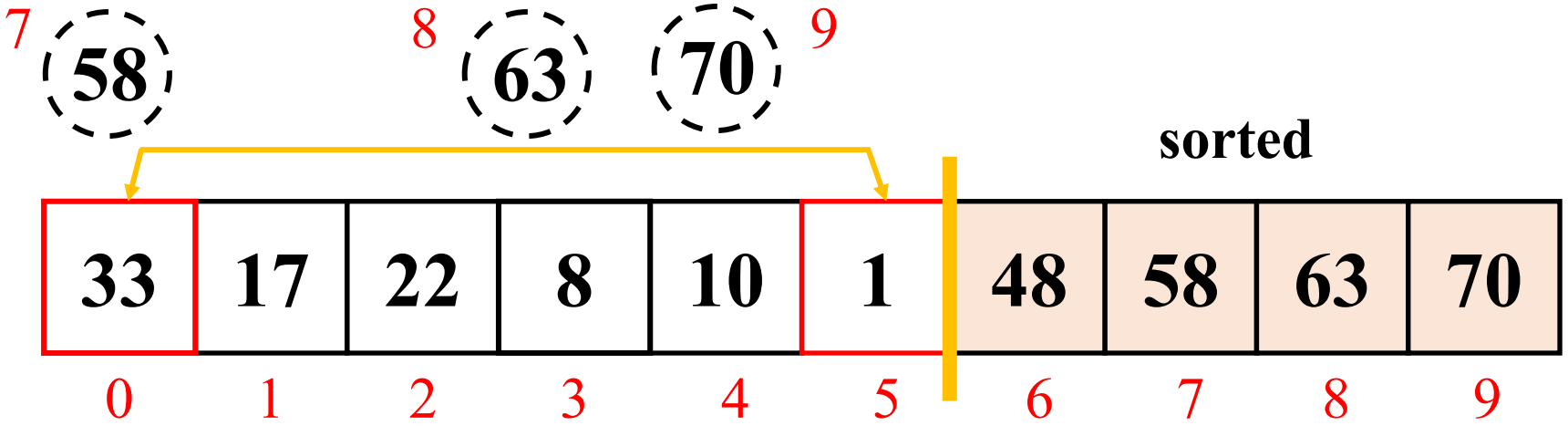
sink 17



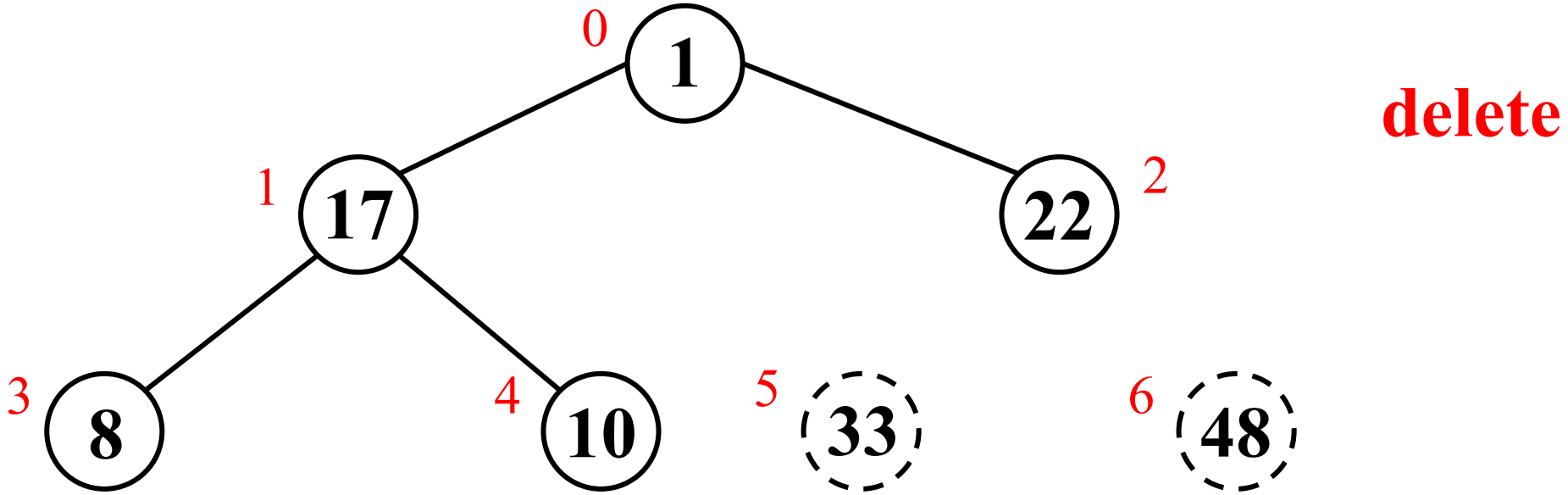
Heapsort algorithms: array implementation

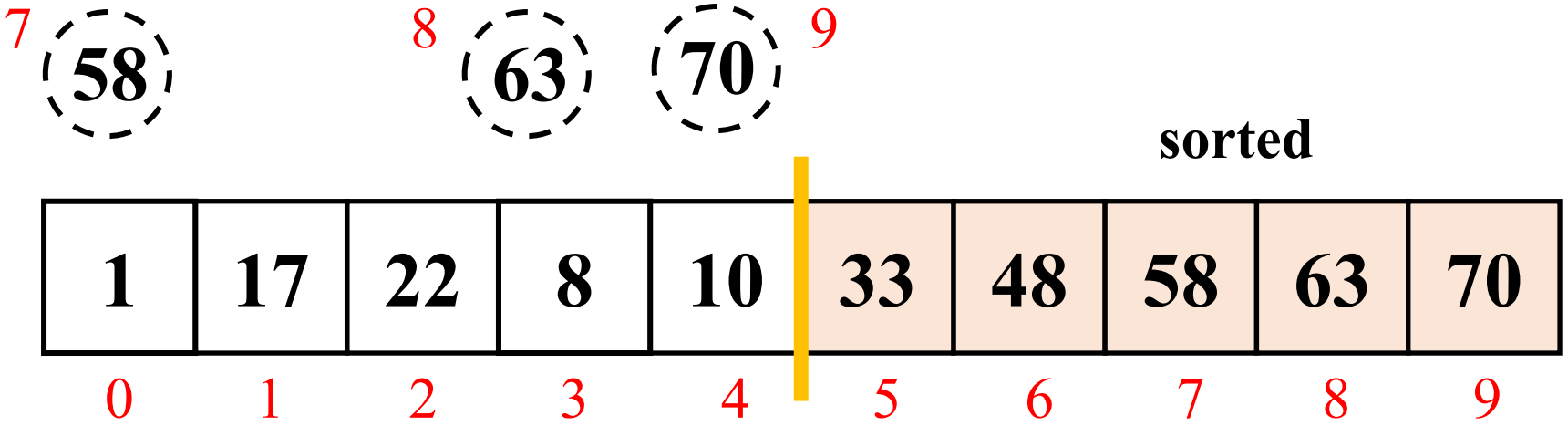


swap

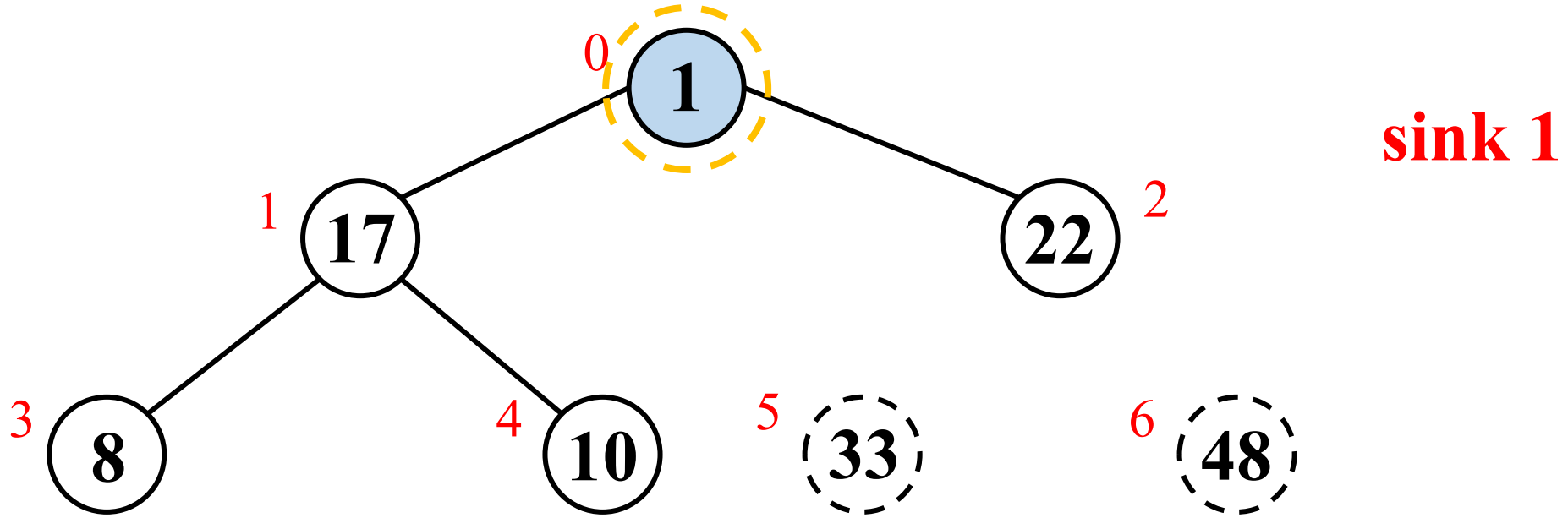


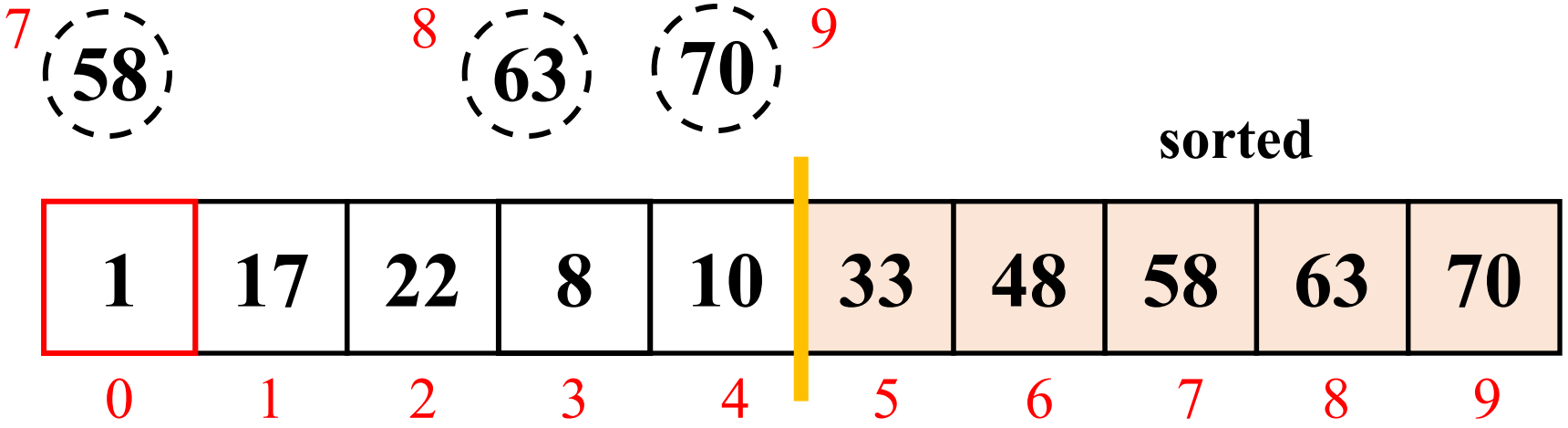
Heapsort algorithms: array implementation



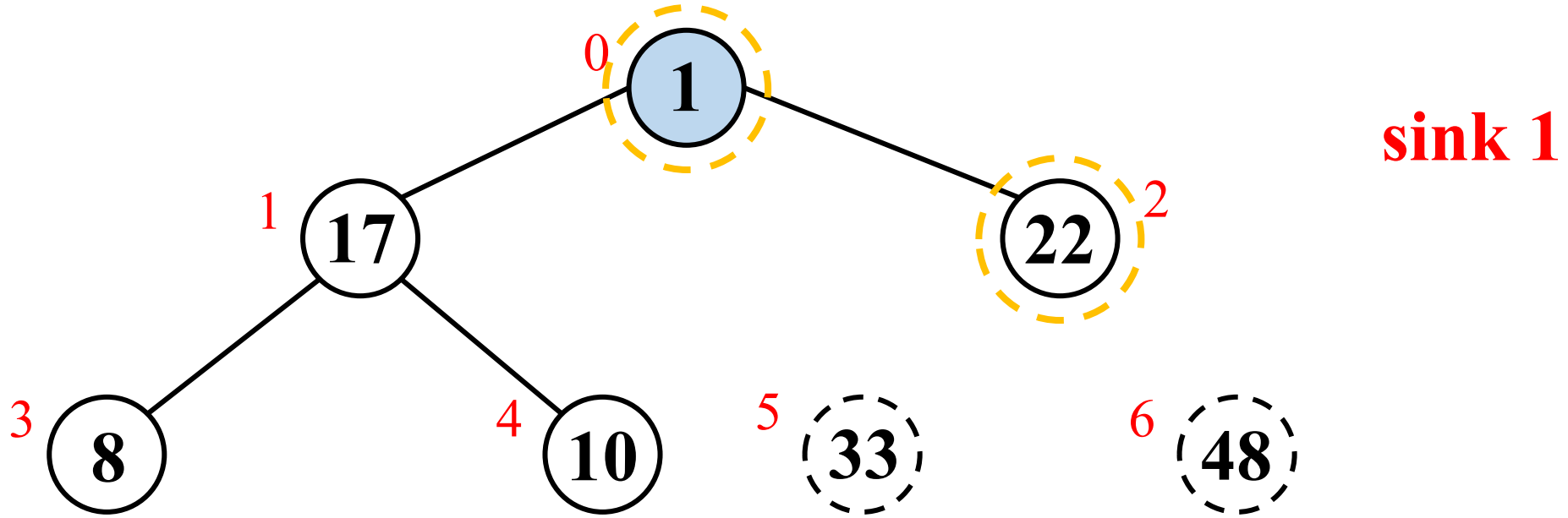


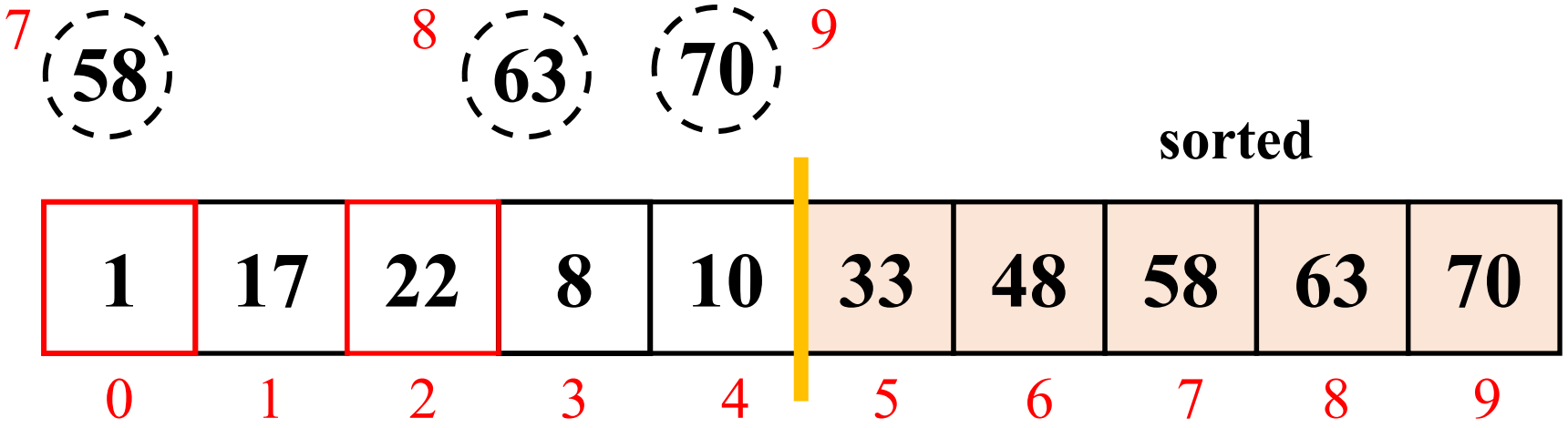
Heapsort algorithms: array implementation



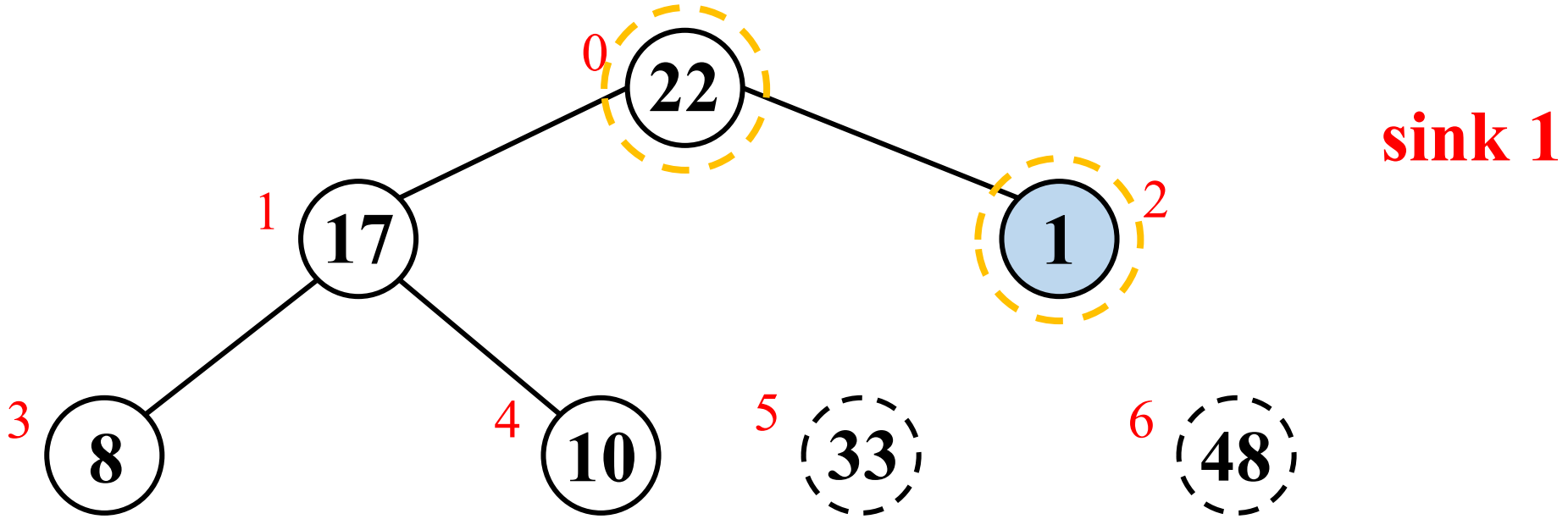


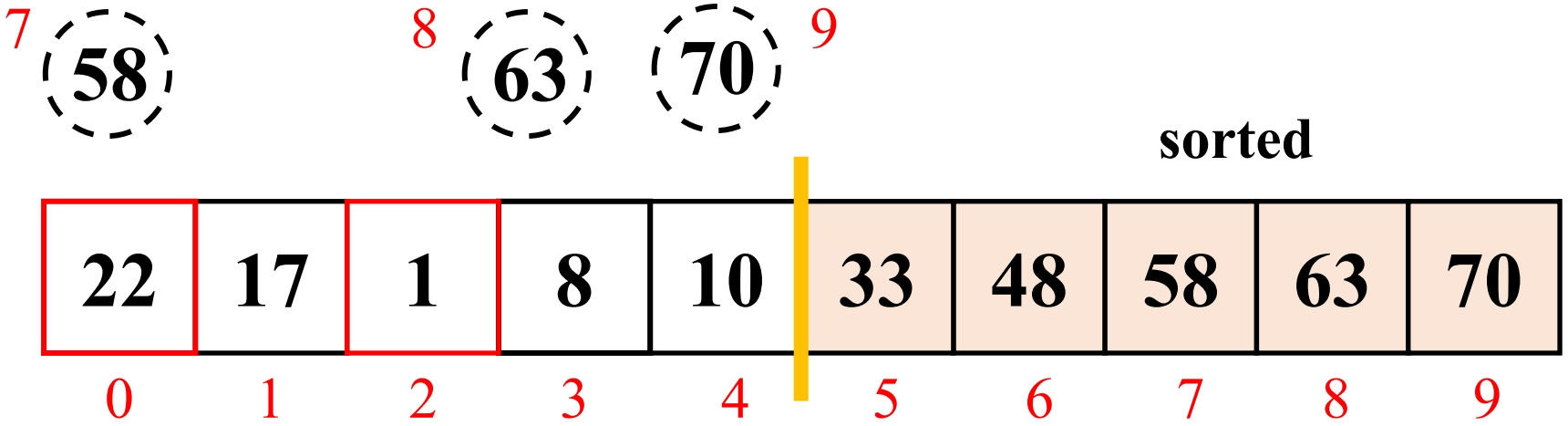
Heapsort algorithms: array implementation



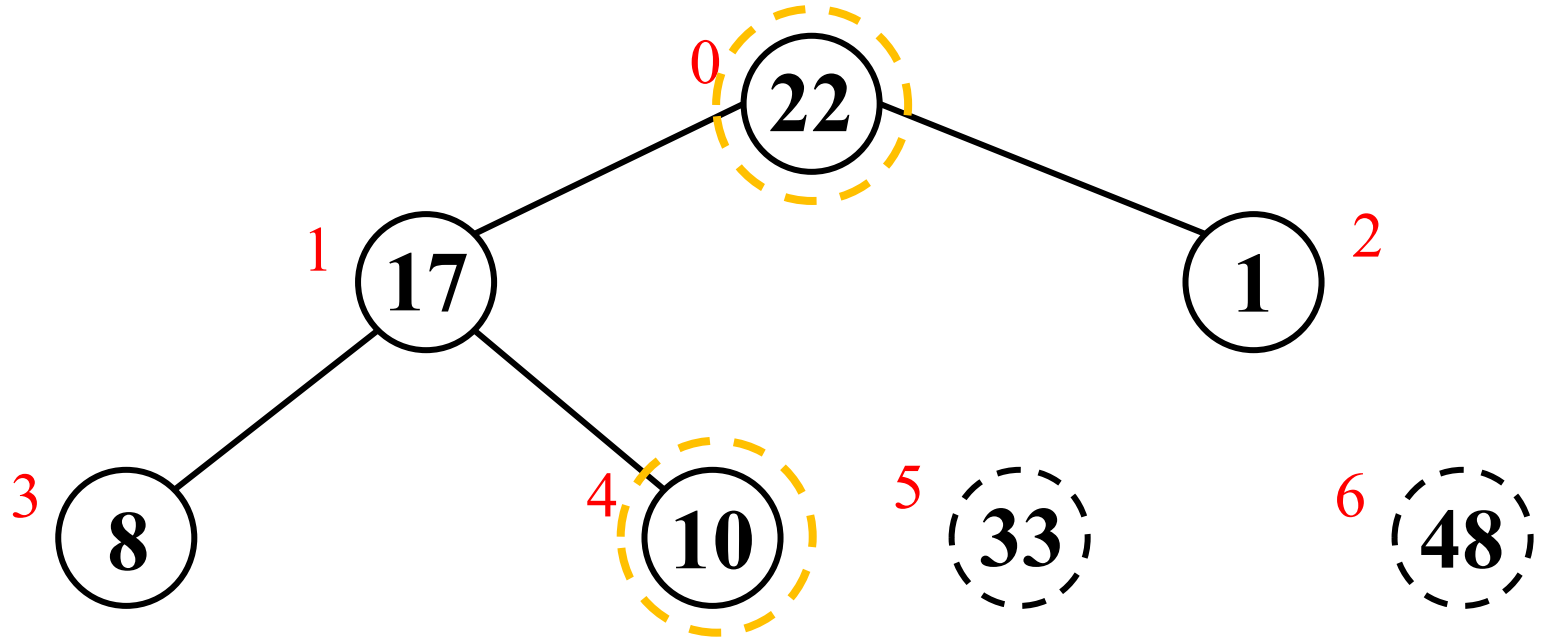


Heapsort algorithms: array implementation

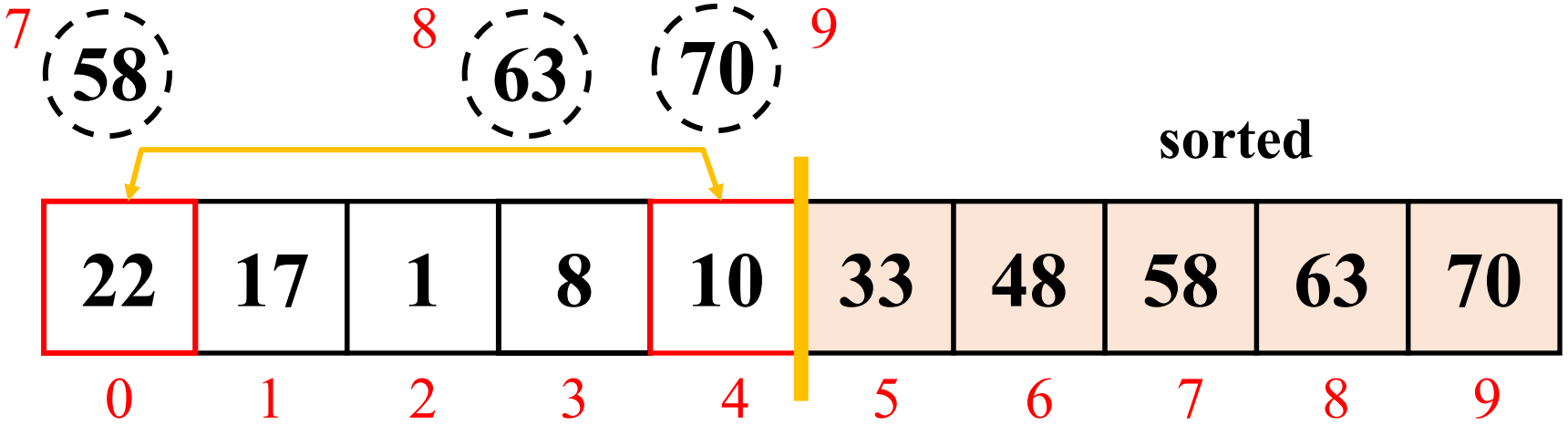




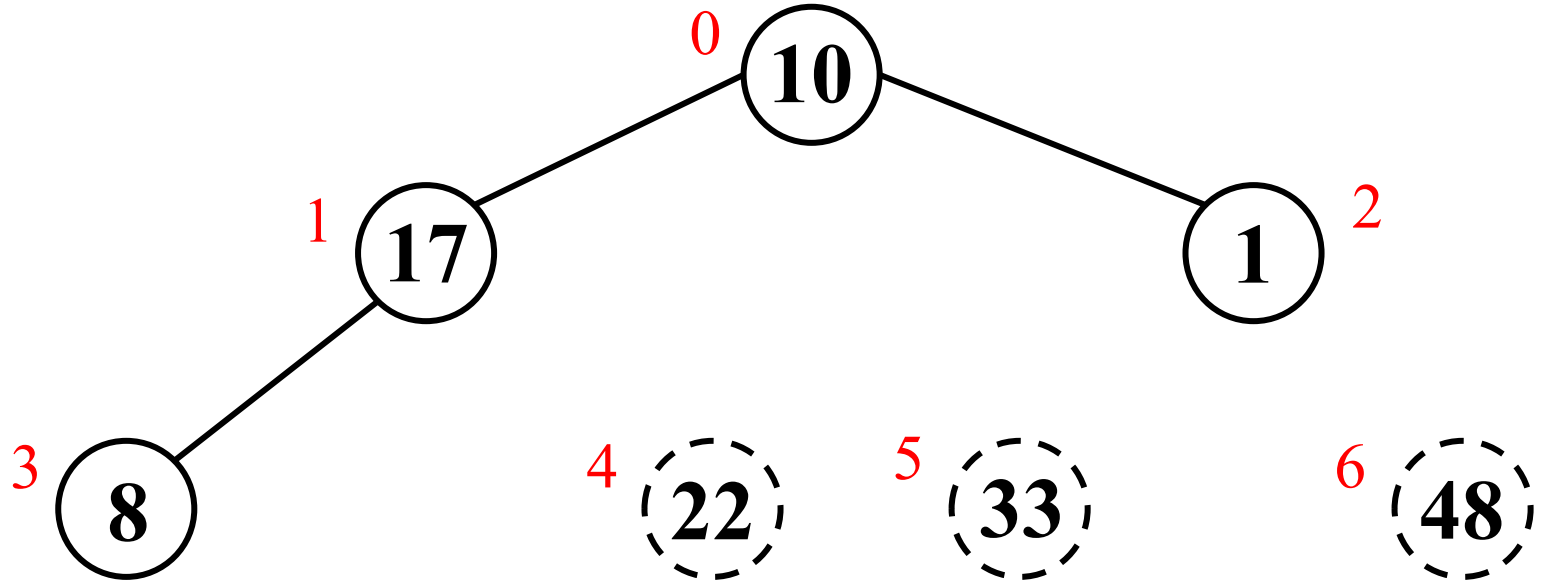
Heapsort algorithms: array implementation

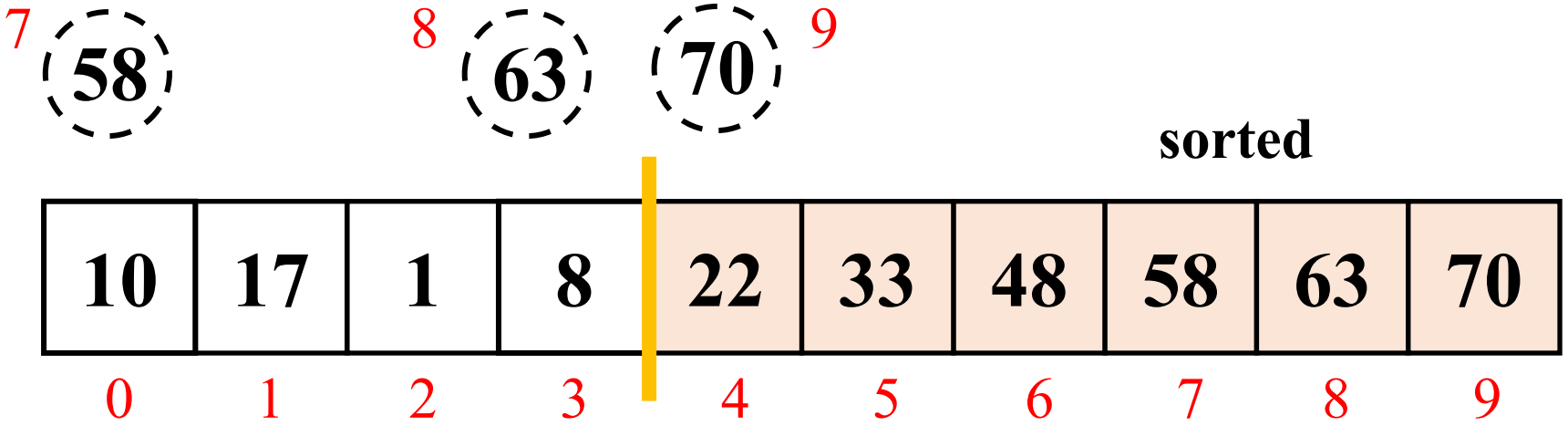


swap

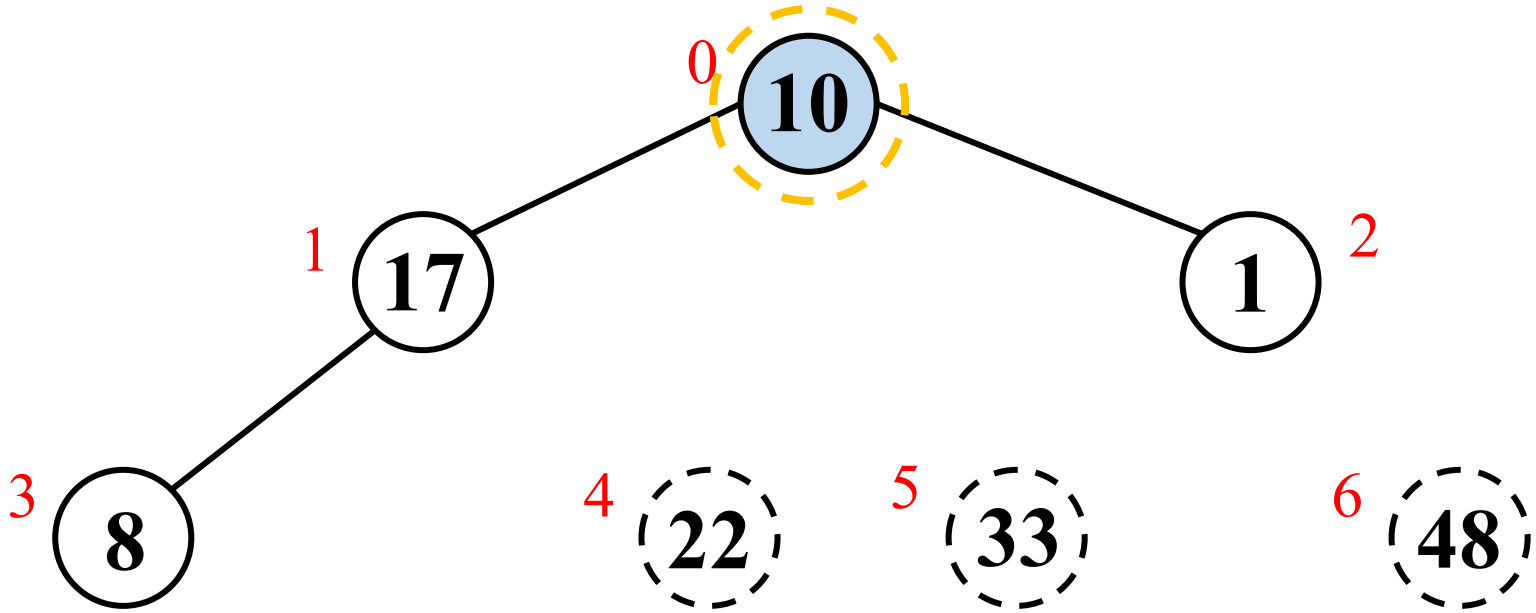


Heapsort algorithms: array implementation

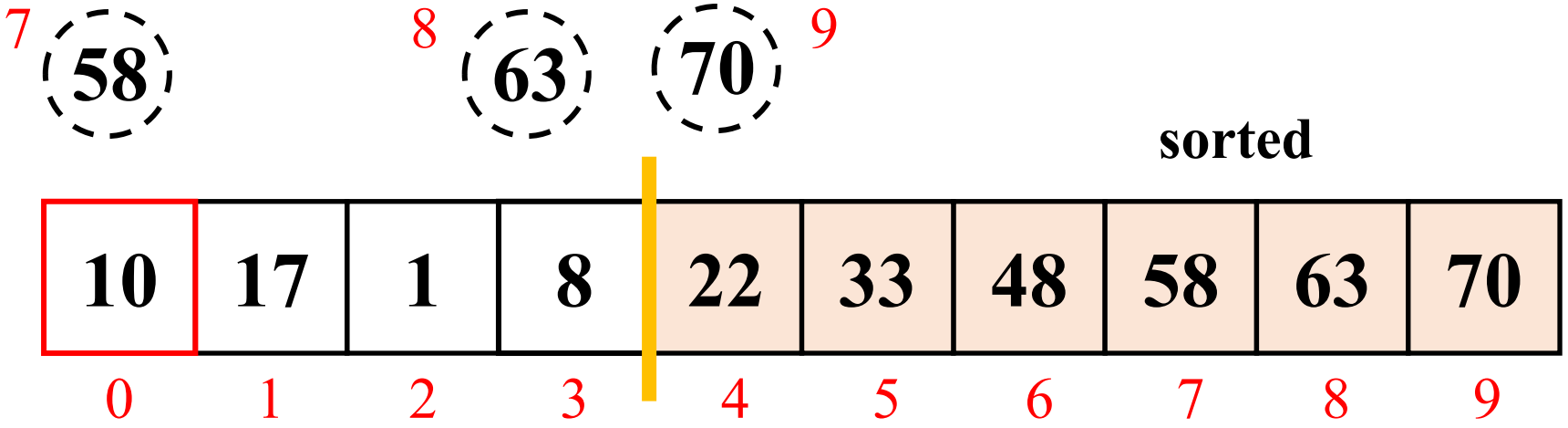




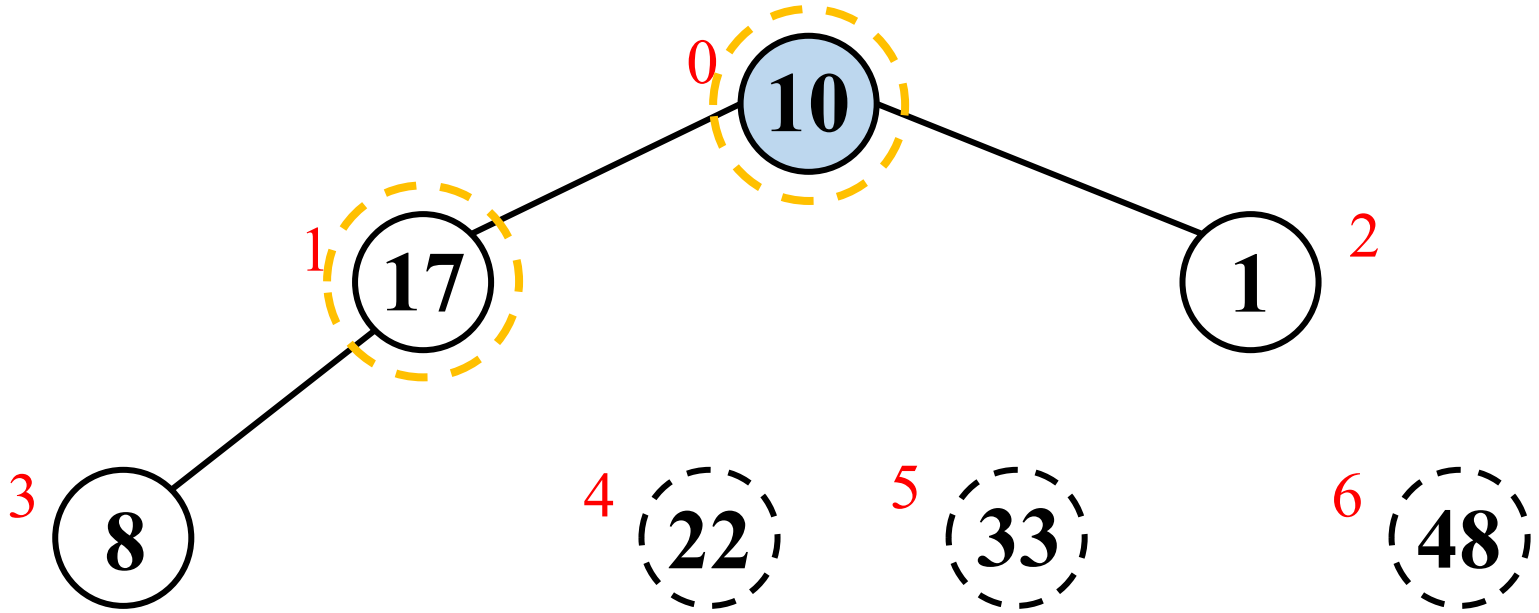
Heapsort algorithms: array implementation



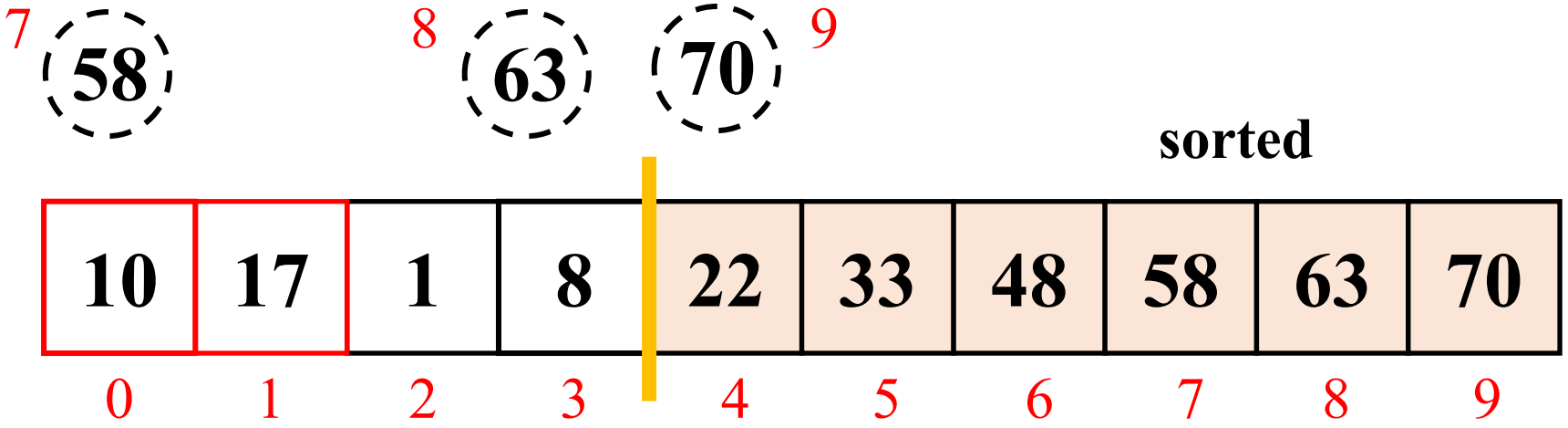
sink 10



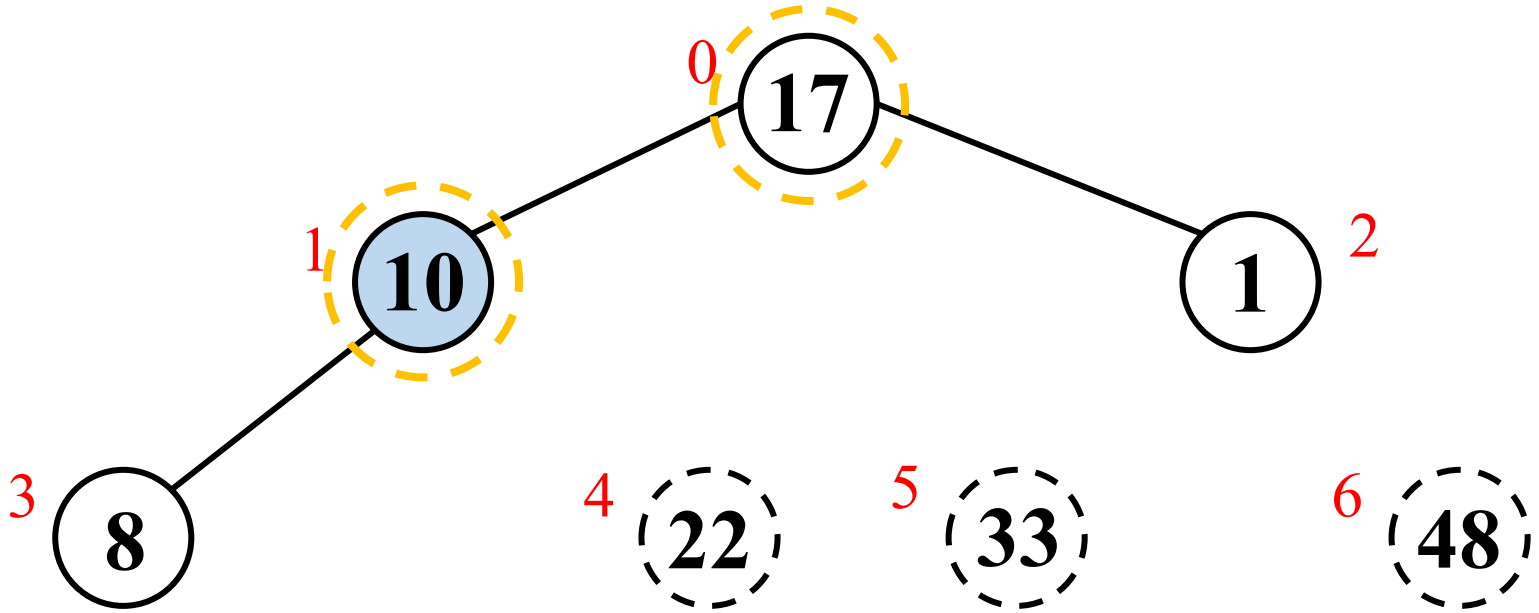
Heapsort algorithms: array implementation



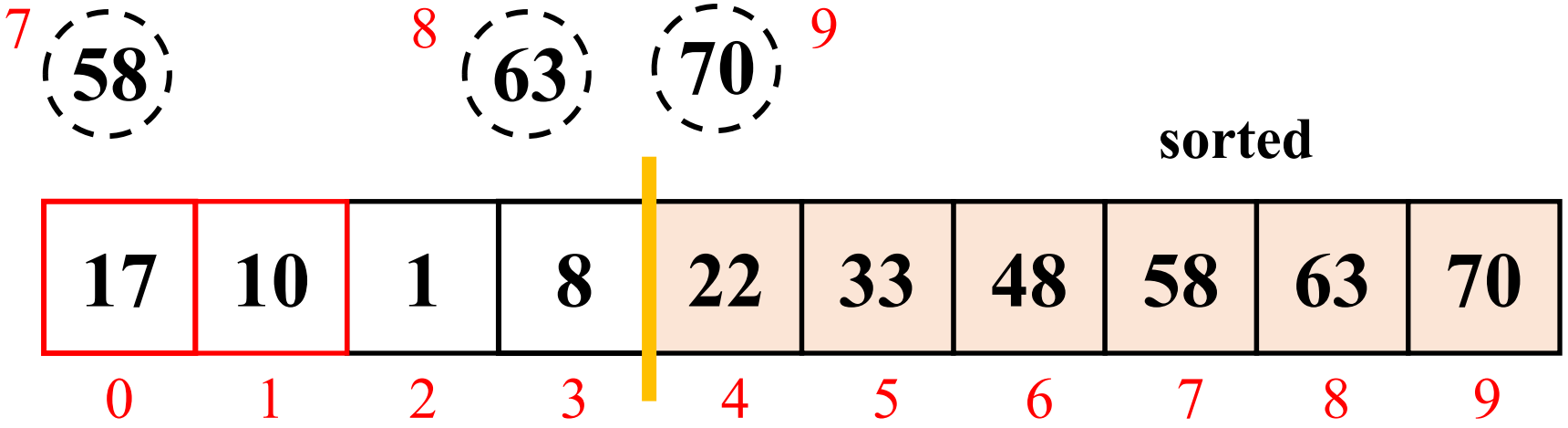
sink 10



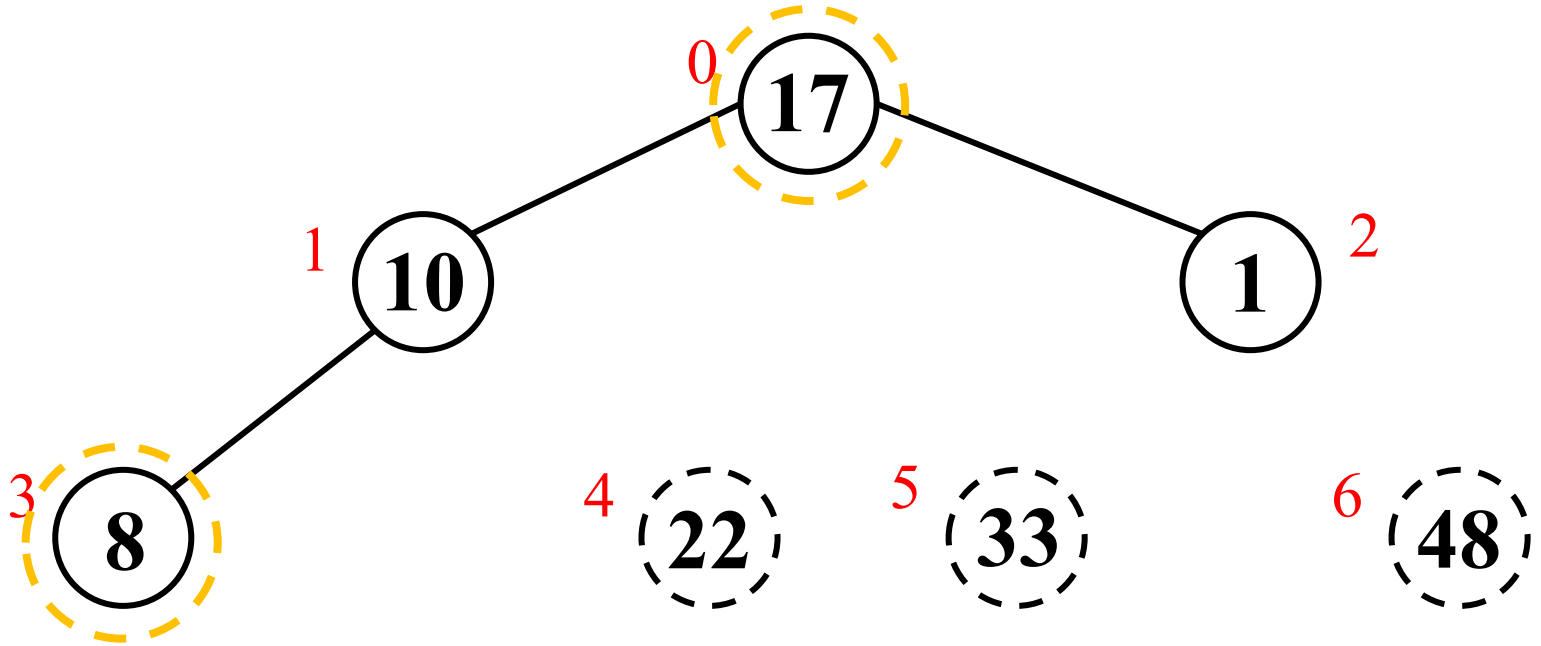
Heapsort algorithms: array implementation



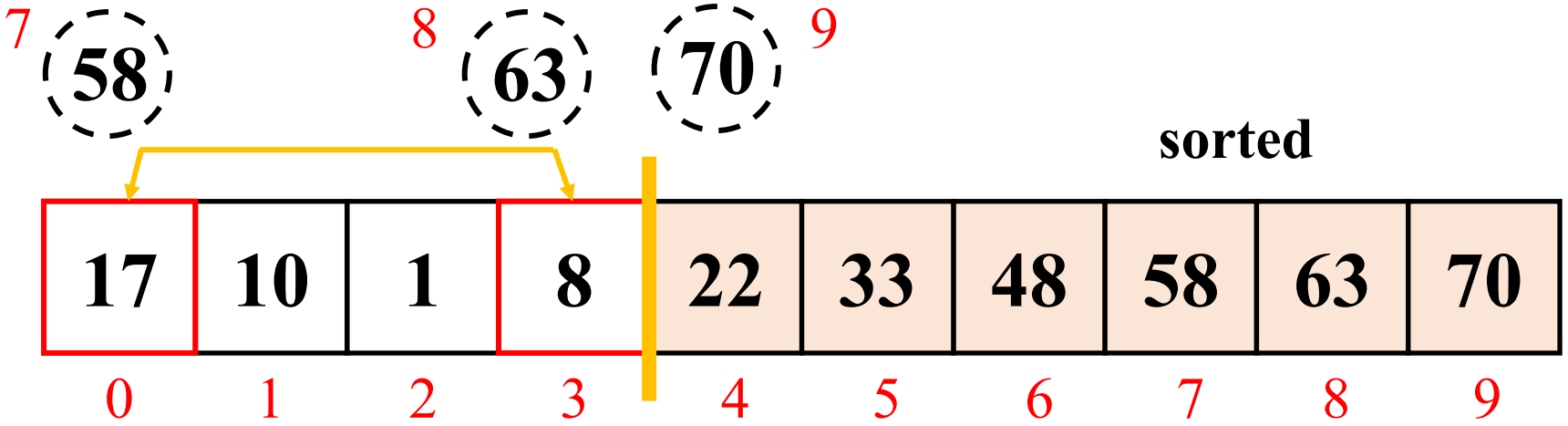
sink 10



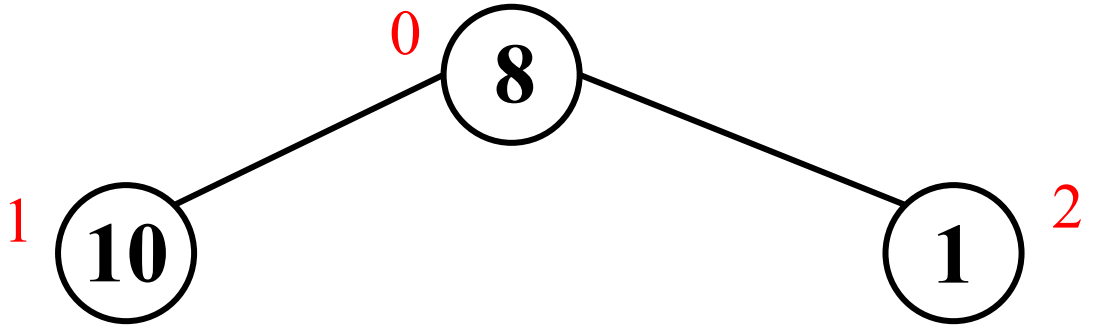
Heapsort algorithms: array implementation



swap

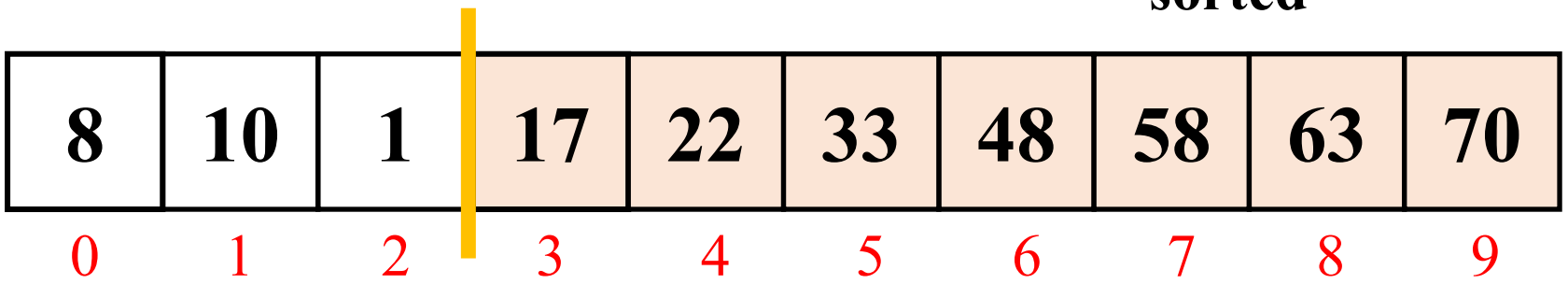


Heapsort algorithms: array implementation

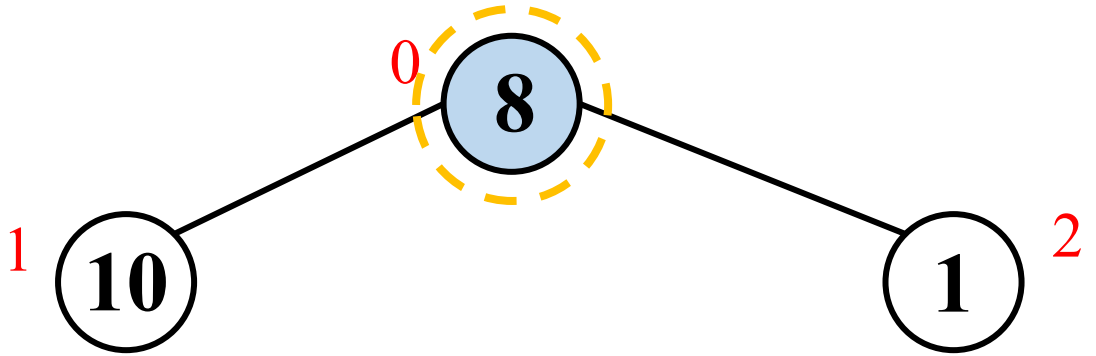


delete

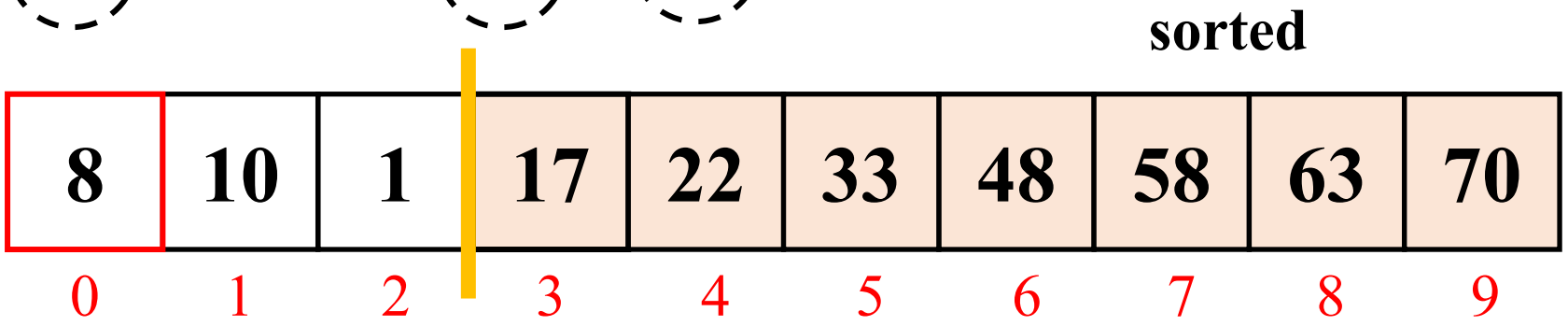
sorted



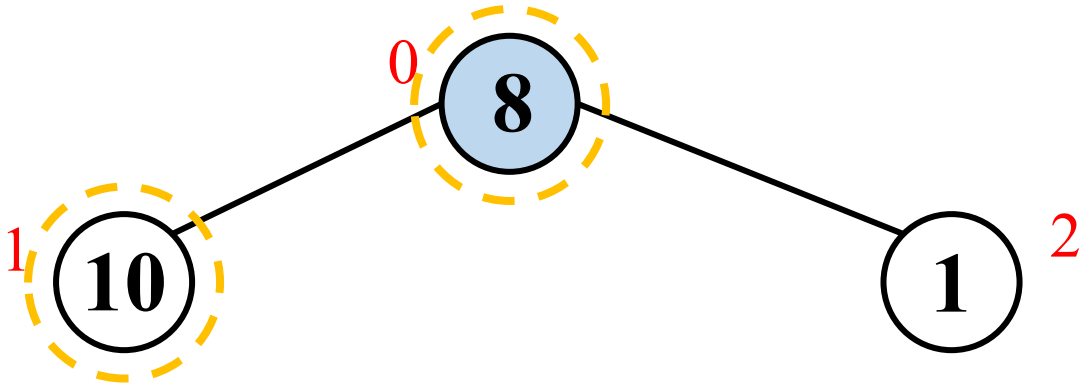
Heapsort algorithms: array implementation



sink 8



Heapsort algorithms: array implementation

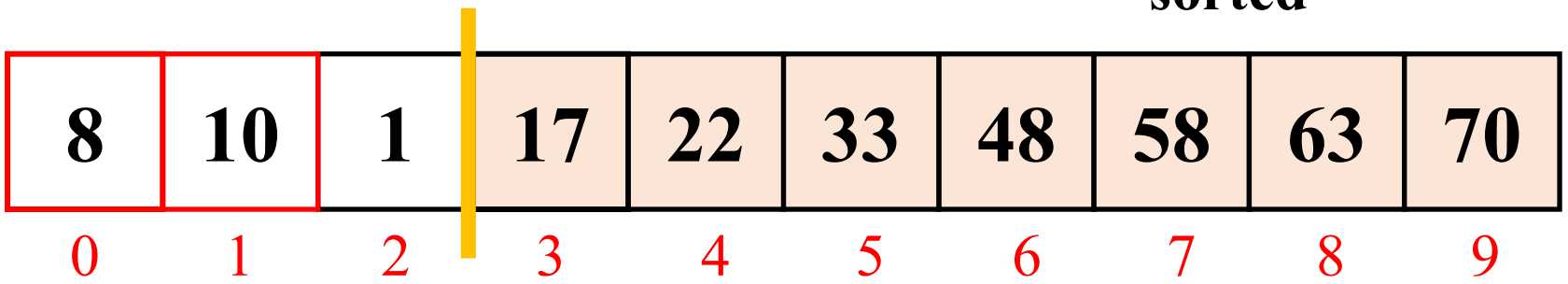


sink 8

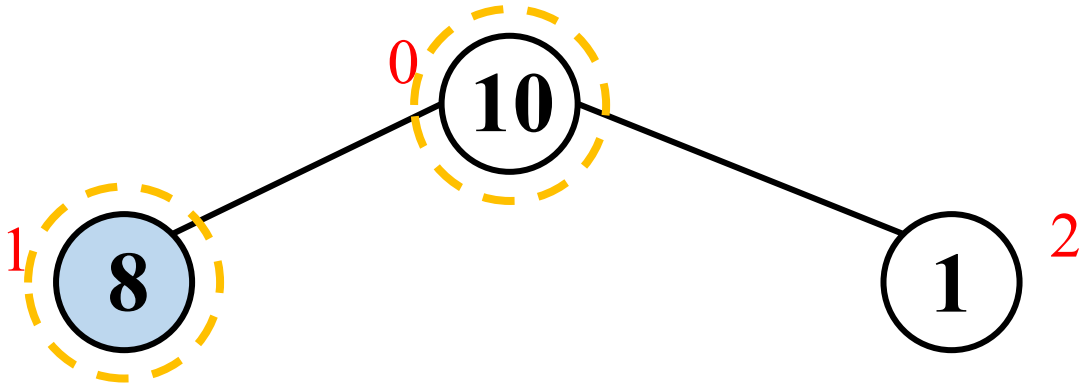
3 17 4 22 5 33 6 48

7 58 8 63 70 9

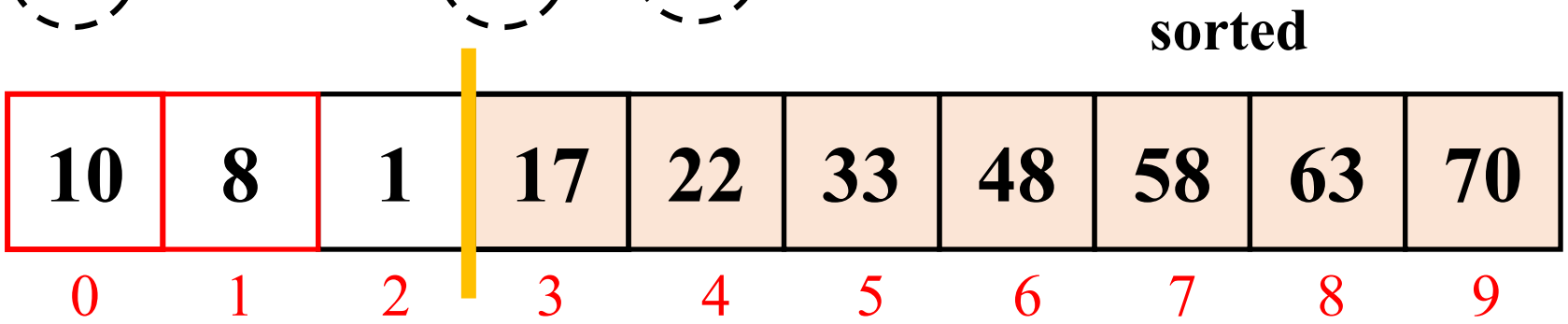
sorted



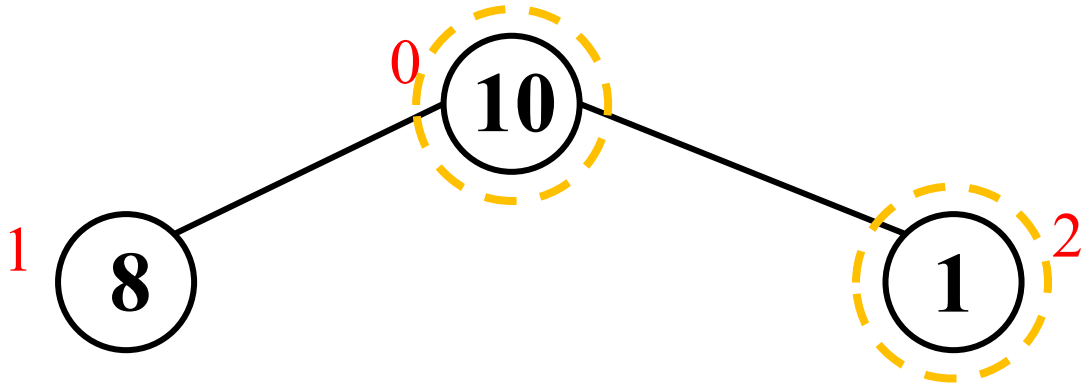
Heapsort algorithms: array implementation



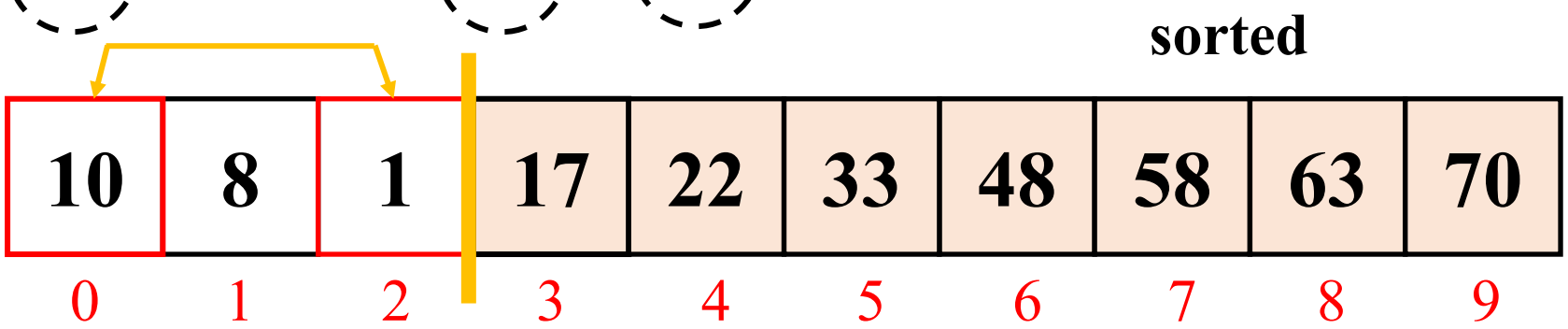
sink 8



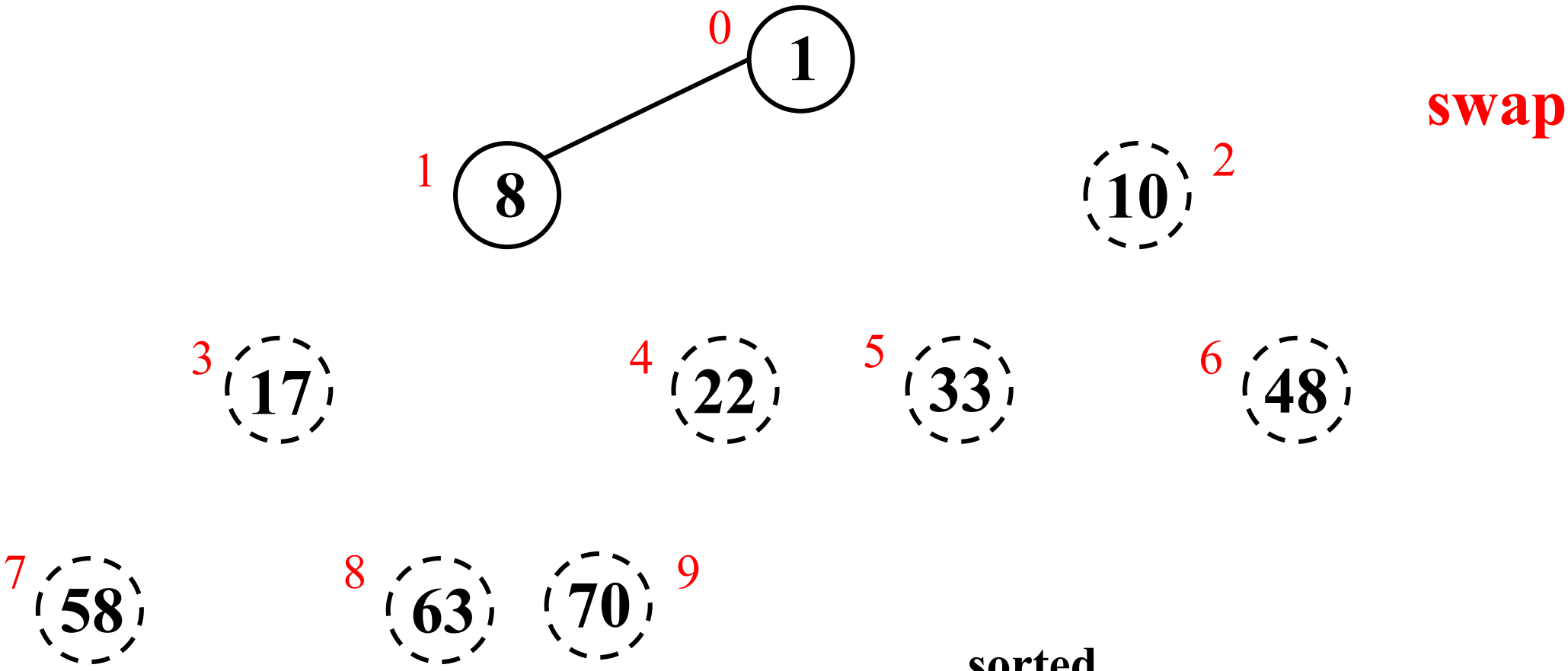
Heapsort algorithms: array implementation



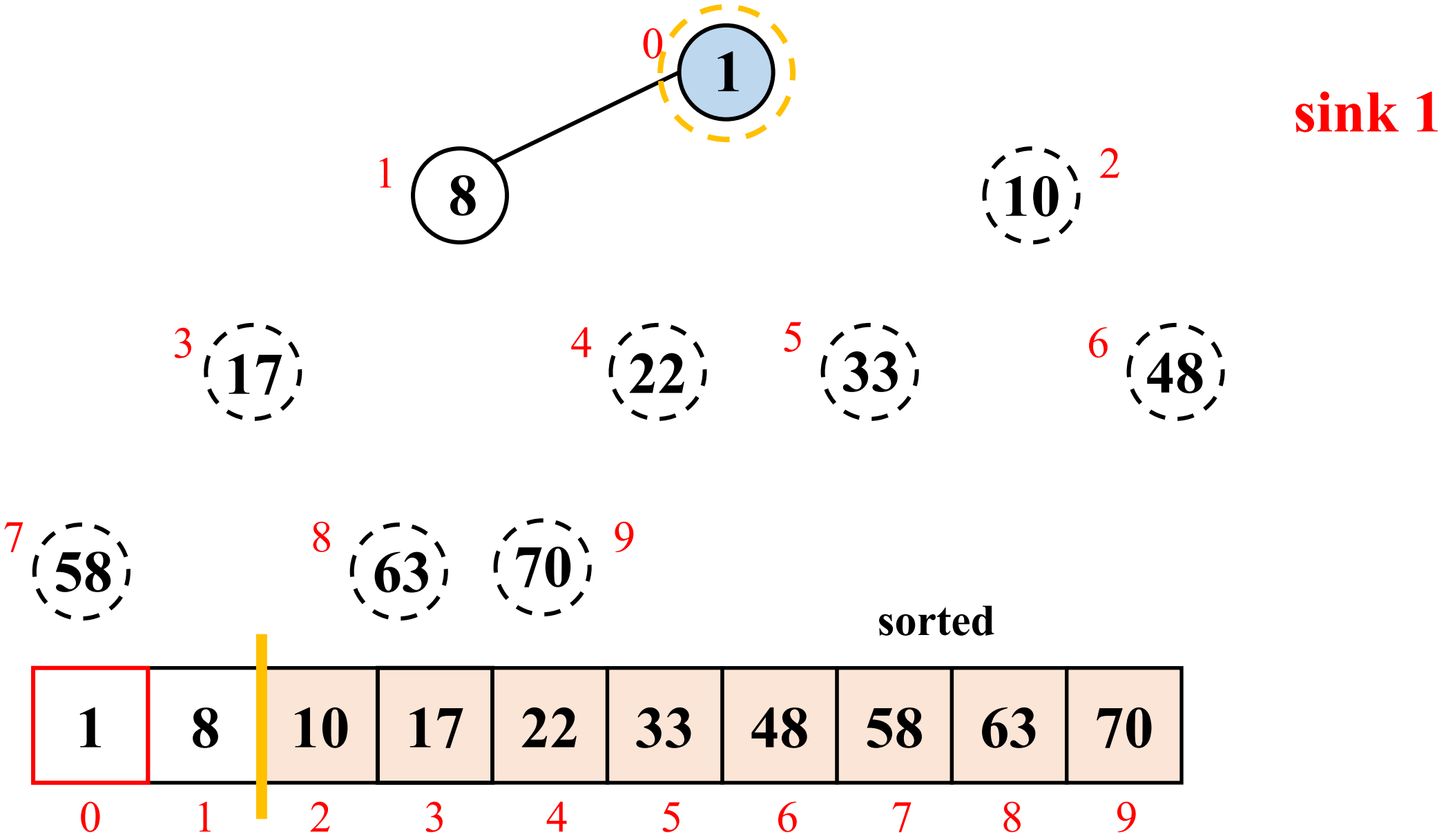
swap



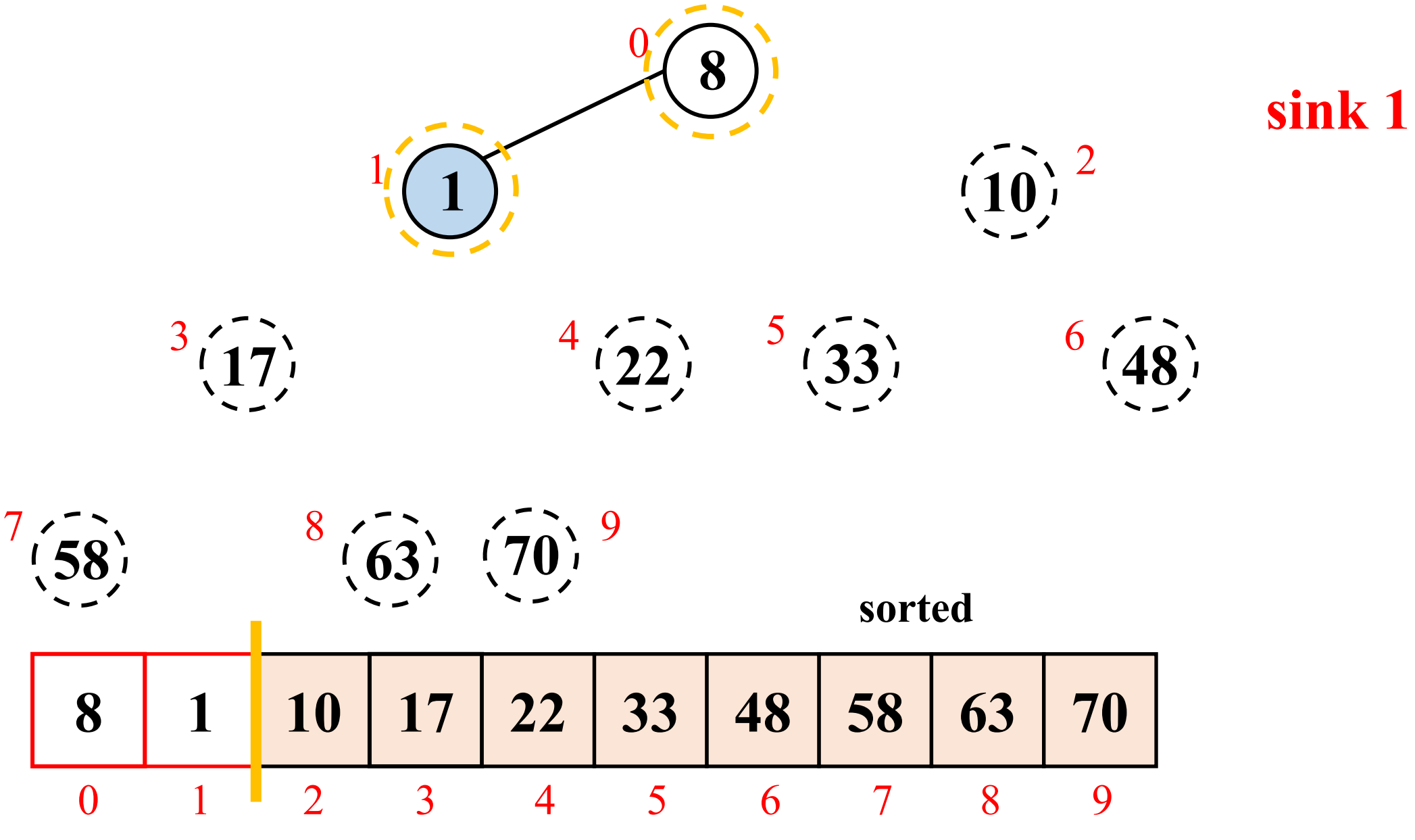
Heapsort algorithms: array implementation



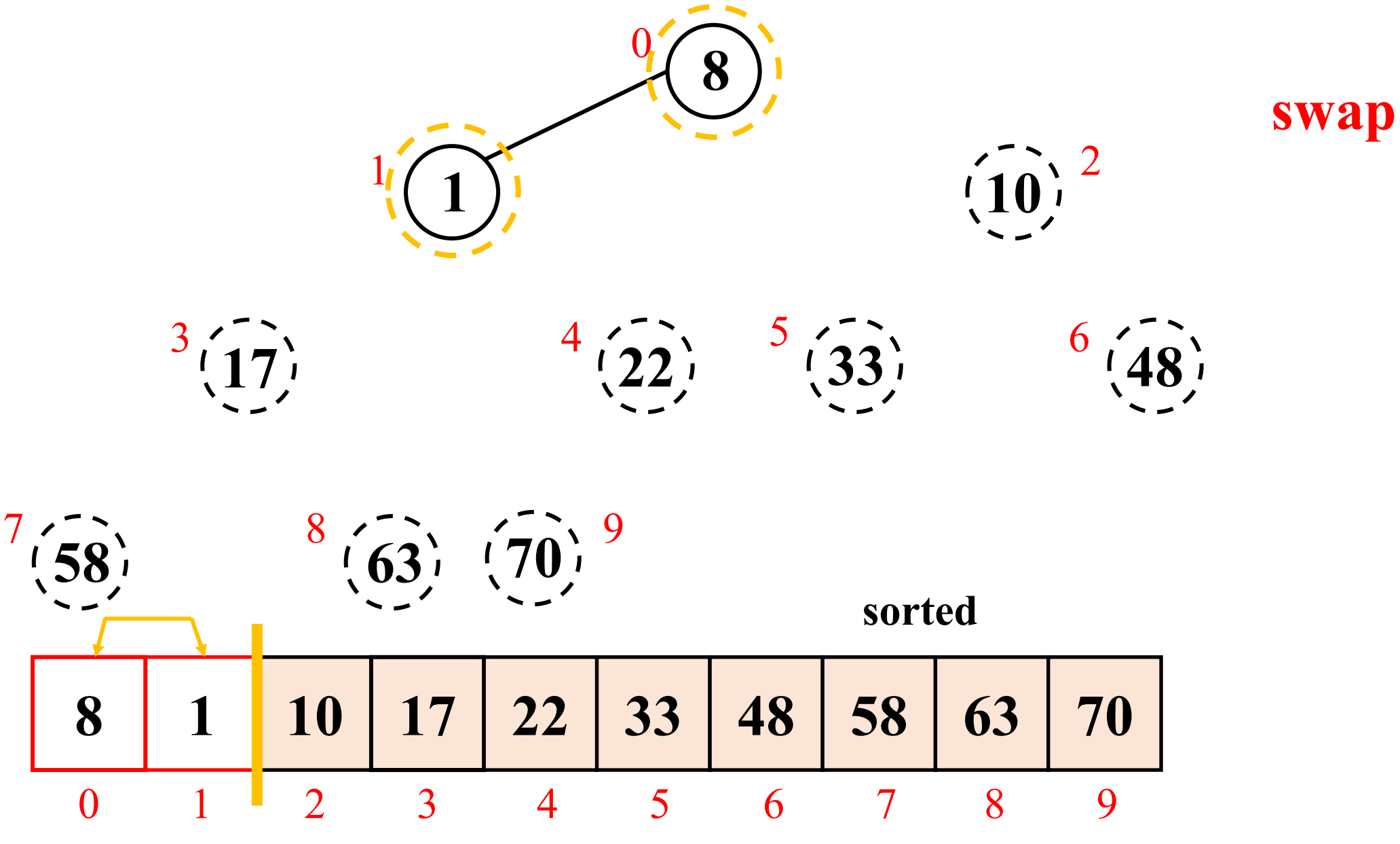
Heapsort algorithms: array implementation



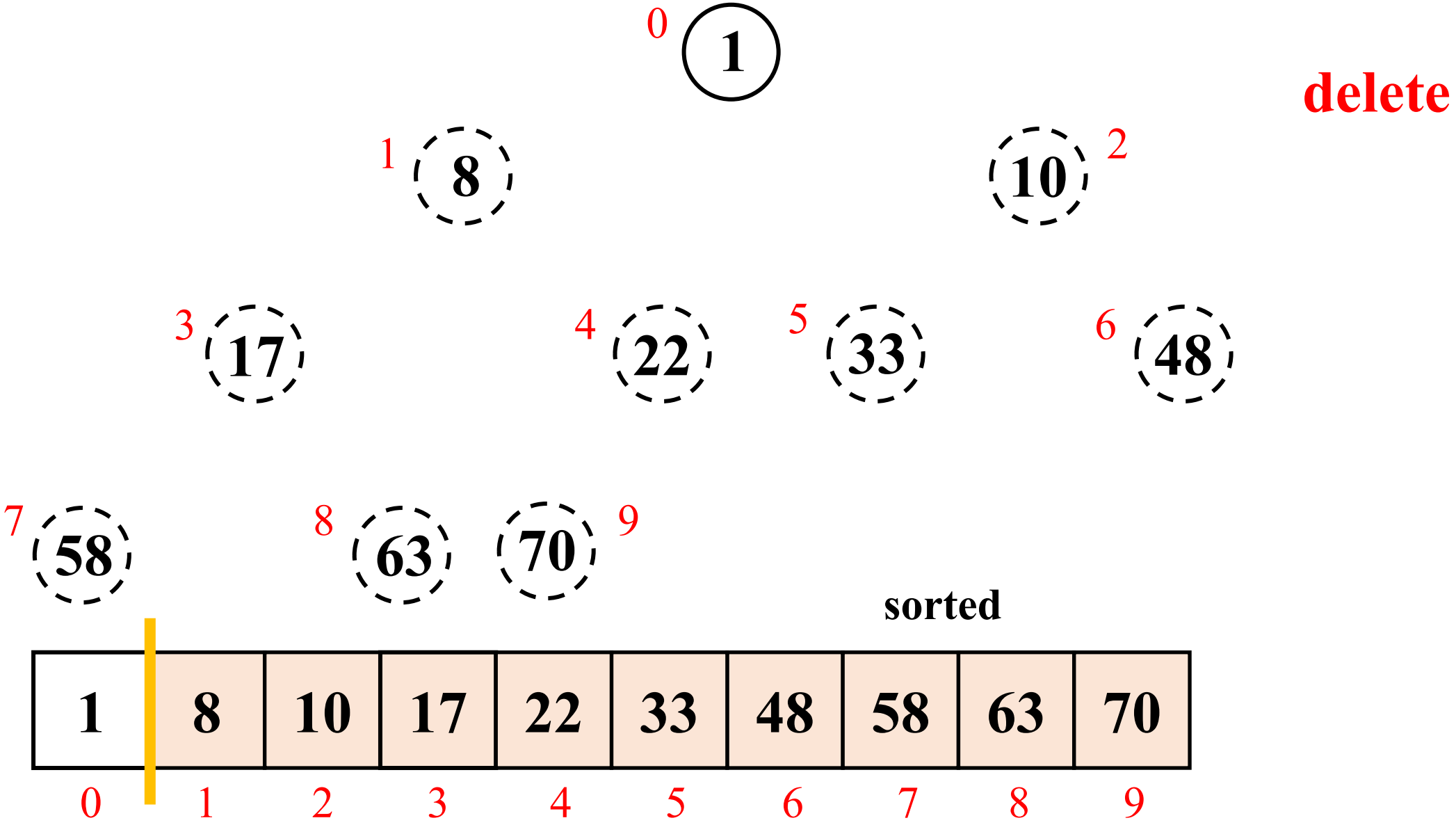
Heapsort algorithms: array implementation



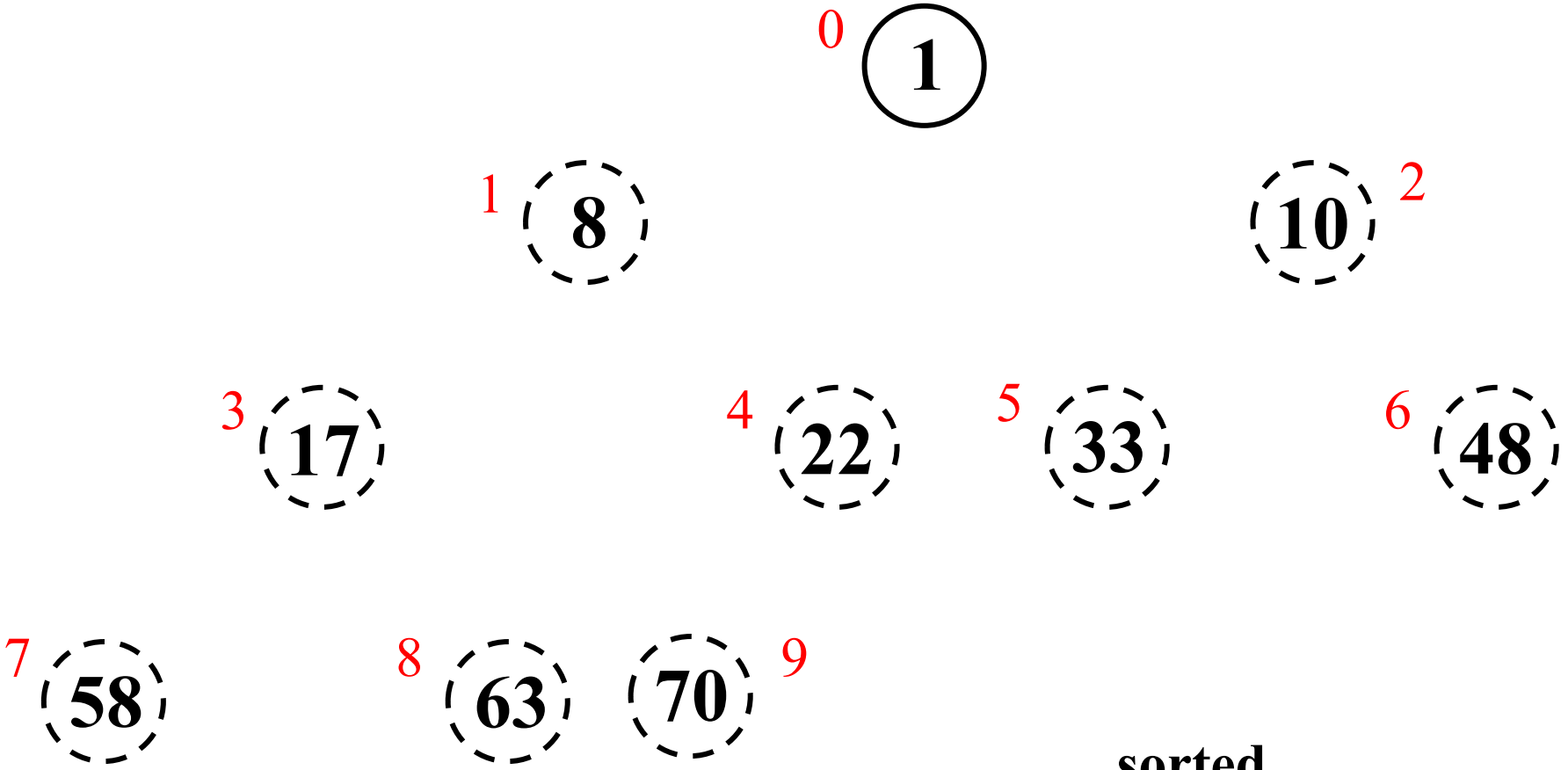
Heapsort algorithms: array implementation



Heapsort algorithms: array implementation



Heapsort algorithms: array implementation



sorted

1	8	10	17	22	33	48	58	63	70
0	1	2	3	4	5	6	7	8	9

Heapsort Algorithms: Array Implementation

Algorithm 1 Heapsort.

```
1: function HEAPSORT(array a[0..n − 1])
2:         k ← n − 1                                   ▷ index of last leaf in heap
3:         i ← ⌊(n−1)/2⌋
4:     while i ≥ 0 do
5:         sink(a, i, k)                               ▷ let element sink to its correct place
6:         i ← i − 1                                   ▷ heap now build
7:     while k > 0 do
8:         swap(a[k], a[0])
9:         k ← k − 1
10:        sink(a, 0, k)                               ▷ let element at root sink to its correct place
11:    return a                                        ▷ Array is now sorted
```

(Build the heap)

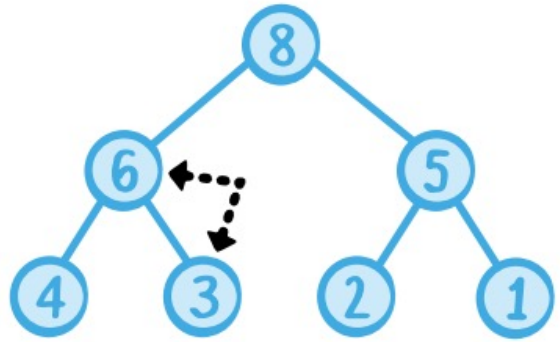
Heapsort Algorithms: Array Implementation (Contd.)

Algorithm 1 Heapsort.

1: **function** HEAPSORT(array $a[0..n-1]$)
2: $k \leftarrow n-1$ ▷ index of last leaf in heap
3: $i \leftarrow \left\lfloor \frac{n-1}{2} \right\rfloor$
4: **while** $i \geq 0$ **do**
5: $sink(a, i, k)$ ▷ let element sink to its correct place
6: $i \leftarrow i-1$ ▷ heap now build
7: **while** $k > 0$ **do**
8: $swap(a[k], a[0])$
9: $k \leftarrow k-1$ **(Delete the maximum repeatedly)**
10: $sink(a, 0, k)$ ▷ let element at root sink to its correct place
11: **return** a ▷ Array is now sorted

Time Complexity Analysis

- Heapsort runs in time in $\Theta(n \log n)$ in the worst and average case.
- **Proof**:
  1. Building the heap:
     - The heap can be constructed in time $\Theta(n \log n)$ with straightforward method. We can do better using the alternative bottom-up method with $\Theta(n)$ time.
  2. Removing the maximum key:
     - Then heapsort repeats n times the deletion of the maximum key and restoration of the heap property (each restoration is logarithmic in the worst and average case).
     - The running time is $\log(n) + \log(n-1) + \log(n-2) + \cdots + 1 = \log(n!) \in \Theta(n \log n)$.
  - Thus, the total running time is $\Theta(n \log n)$

SUMMARY

- Heapsort
  - Algorithms
  - Illustrating Example
- Time Complexity Analysis