CSC 225

Algorithms and Data Structures I

Rich Little
rlittle@uvic.ca
ECS 516

Priority Queues

- A *priority queue (PQ)* is a container of elements, each having an associated *key*
- The keys determine the *priority* used to remove elements (i.e., removeMin or removeMax)
- The keys of the elements in a PQ must satisfy the *total order properties*
- The operations of a PQ are
  - **insert(*k*, *e*):** insert an element *e* with key *k* into PQ
  - **removeMin():** return and remove from PQ an element with the smallest key

Sequence-based Priority Queue

- Implementation with an unsorted sequence
  - Store the items of the priority queue in a list-based sequence, in arbitrary order

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin takes $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted sequence
  - Store the items of the priority queue in a sequence, sorted by key

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin takes $O(1)$ time since the smallest key is at the beginning of the sequence

Sorting with a Priority Queue

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert(e, e) operations
  2. Remove the elements in sorted order with a series of removeMin() operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm ***PQ-Sort*****(*S*, *P*)**
Input**:** An n-element array, ***S***, and a priority queue ***P***
Output**:** Array ***S*** sorted in increasing order

for $i \leftarrow 1$ **to** n **do**
  $e \leftarrow S[i]$
  $P.insert(e, e)$
for $i \leftarrow 1$ **to** n **do**
  $e \leftarrow P.removeMin()$
  $S[i] \leftarrow e$

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

- Running time of Selection-sort:
  - Inserting the elements into the priority queue with n insert(e,e) operations takes $O(n)$ time
  - Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence

- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with n insert(e,e) operations takes time proportional to

$$1 + 2 + \ldots + n$$

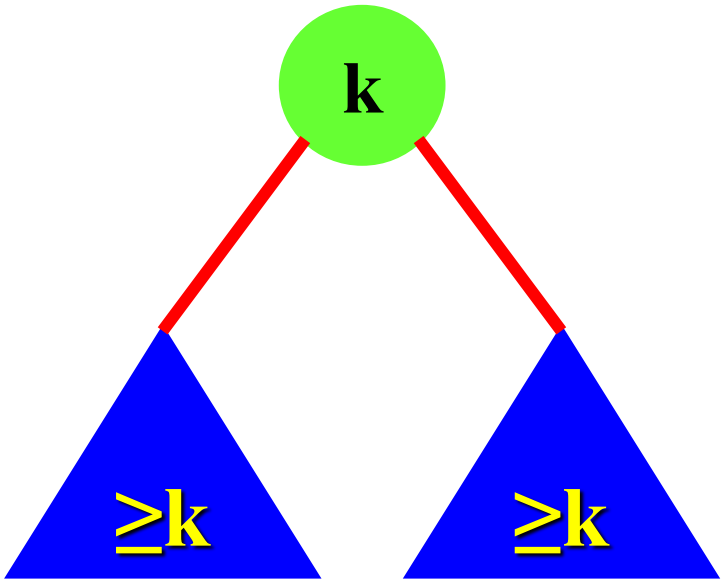
  - Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

The Heap Data Structure

- A *heap* is a realization of a Priority Queue that is efficient for both insertions and removals of minima (meaning one of possibly many minimum values)

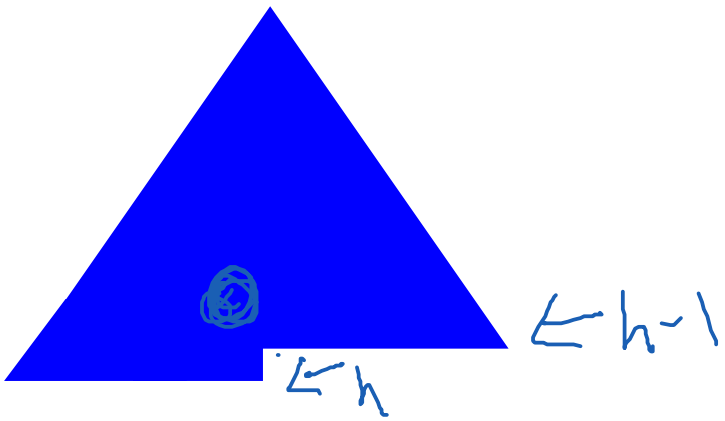
Heap Definition

- A *heap* is a complete binary tree that stores a collection of keys at its internal nodes
- A heap satisfies the following properties:
  - *Heap-Order Property*: for every node *v* other than the root, the key stored at *v* is greater than or equal to the key stored at *v*'s parent.



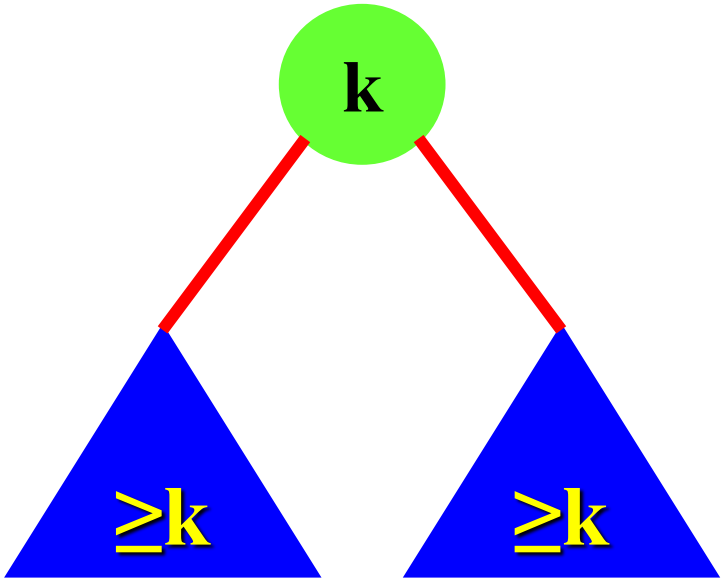
Complete Binary Tree

- *Complete Binary Tree*: A binary tree with height *h* is complete if
  - the depths 0, 1, 2,…, *h*–2 have the maximum number of nodes possible and
  - at depth *h-1,* the internal nodes are all to the left of the leaves
- Heap Shape property
  - A heap is a complete binary tree

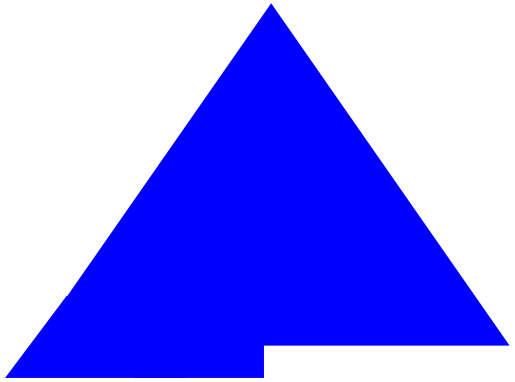


Heap Properties

- Order property



- Shape property



Height of a Heap

- **Theorem:** A heap storing n keys has height $O(\log n)$

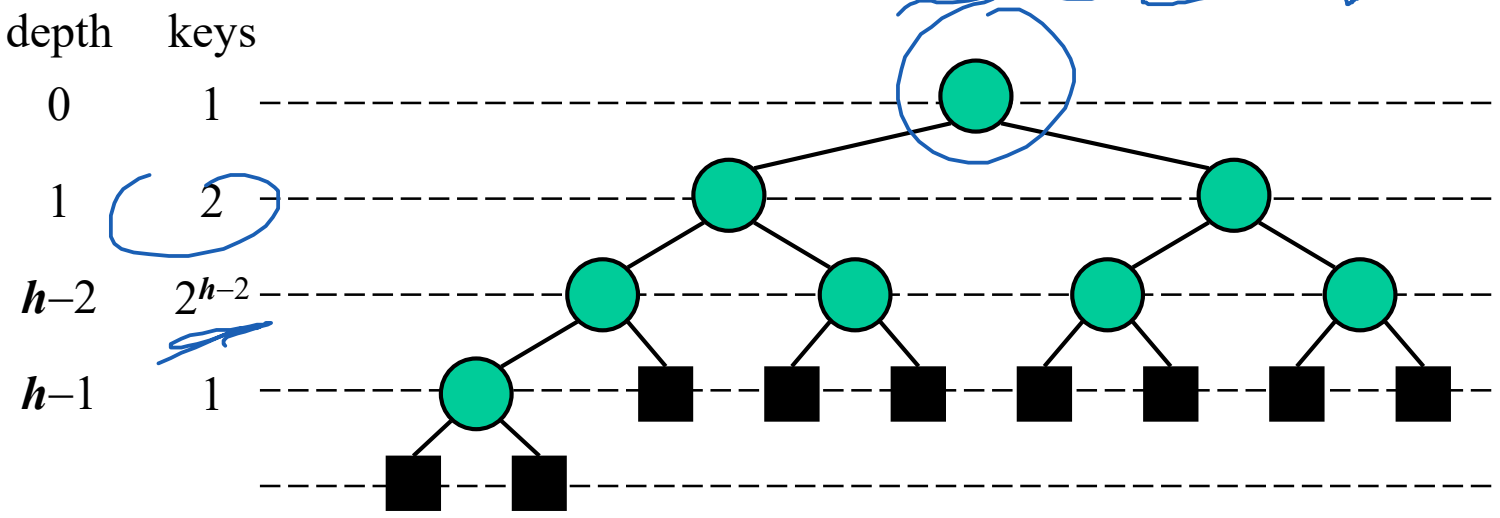
  Proof: (we apply the complete binary tree property)

$$n \geq 2^0 + 2^1 + \cdots + 2^{h-2} + 1 = \frac{1 - 2^{(h-1)}}{1-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$$

$$\log_2 n \geq h-1 \Rightarrow h \leq \log n + 1$$

$$h \leq \log n + 1 \in O(\log n)$$

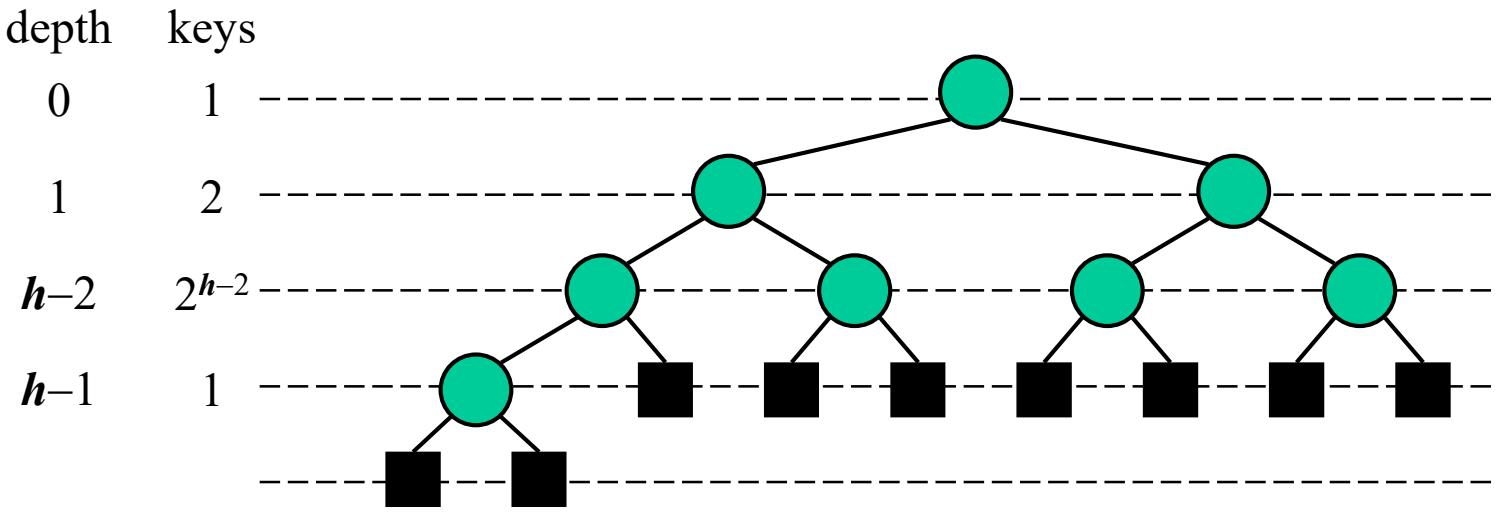
min heap



Height of a Heap

- Theorem: A heap storing n keys has height $O(\log n)$

  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$

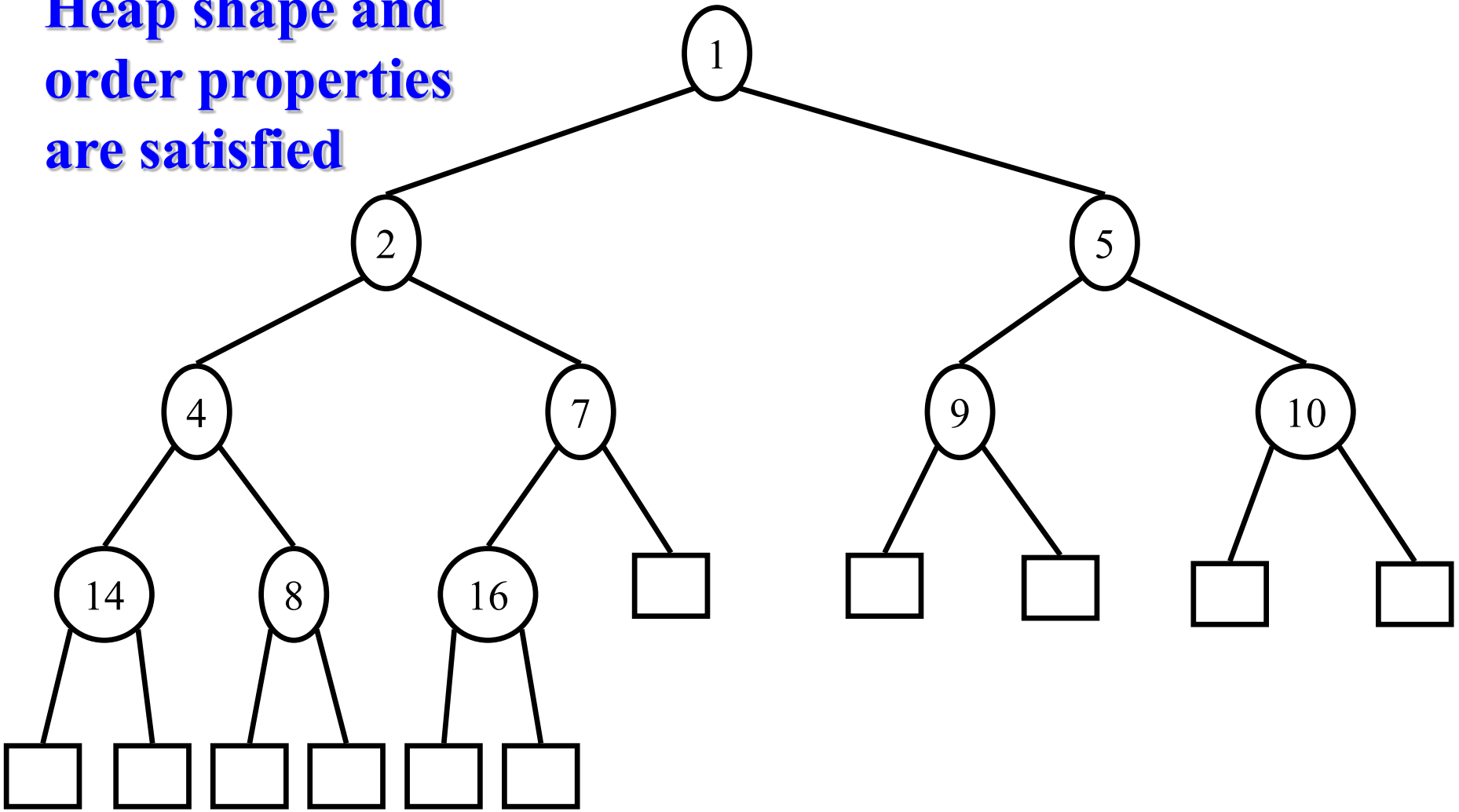


Inserting an Element into a Heap

- Satisfy shape and order properties

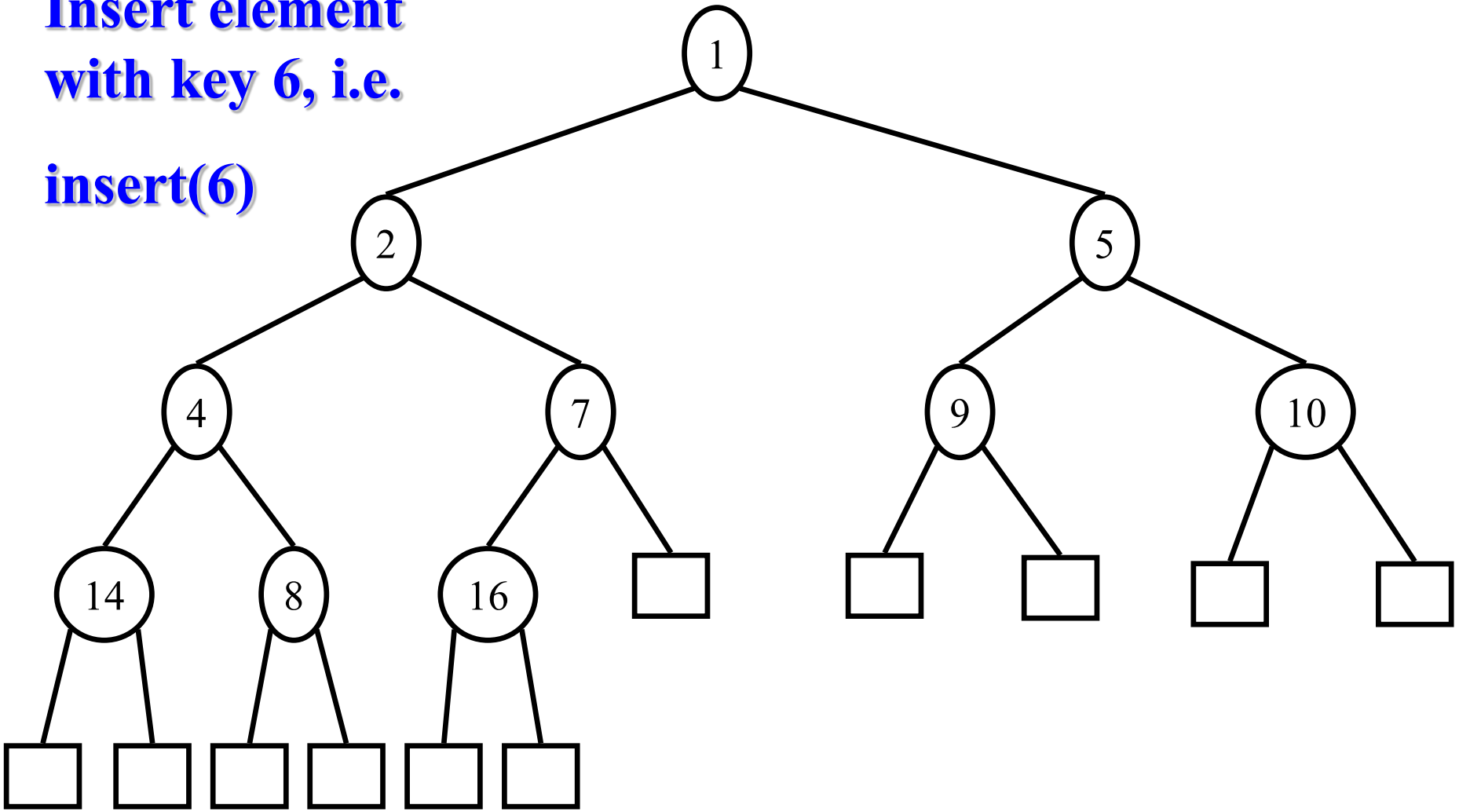
- Satisfy shape property
  - Insert element at first available spot
- Satisfy order property
  - Bubble the element up the tree until the order property is restored

Heap shape and order properties are satisfied

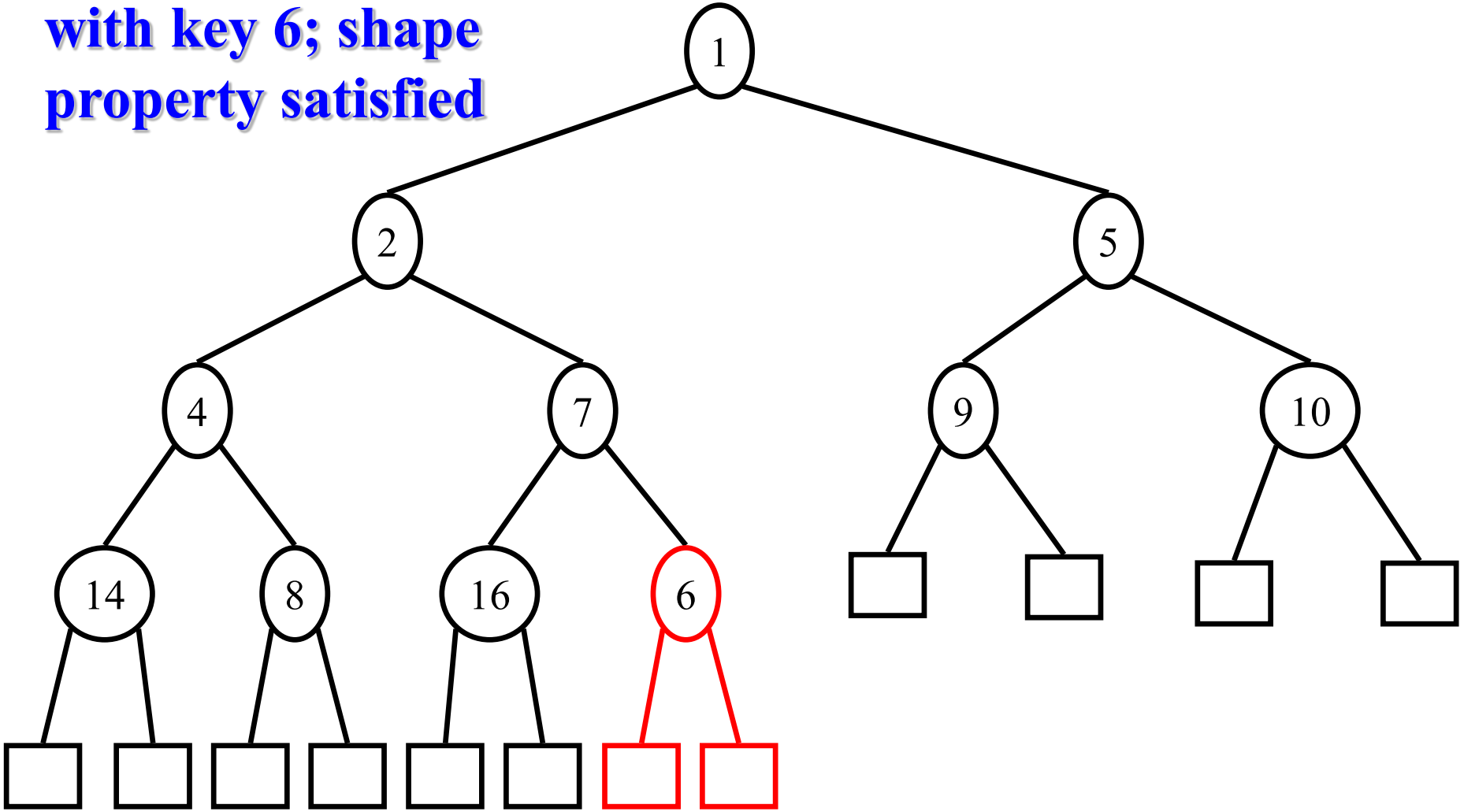


Insert element with key 6, i.e.

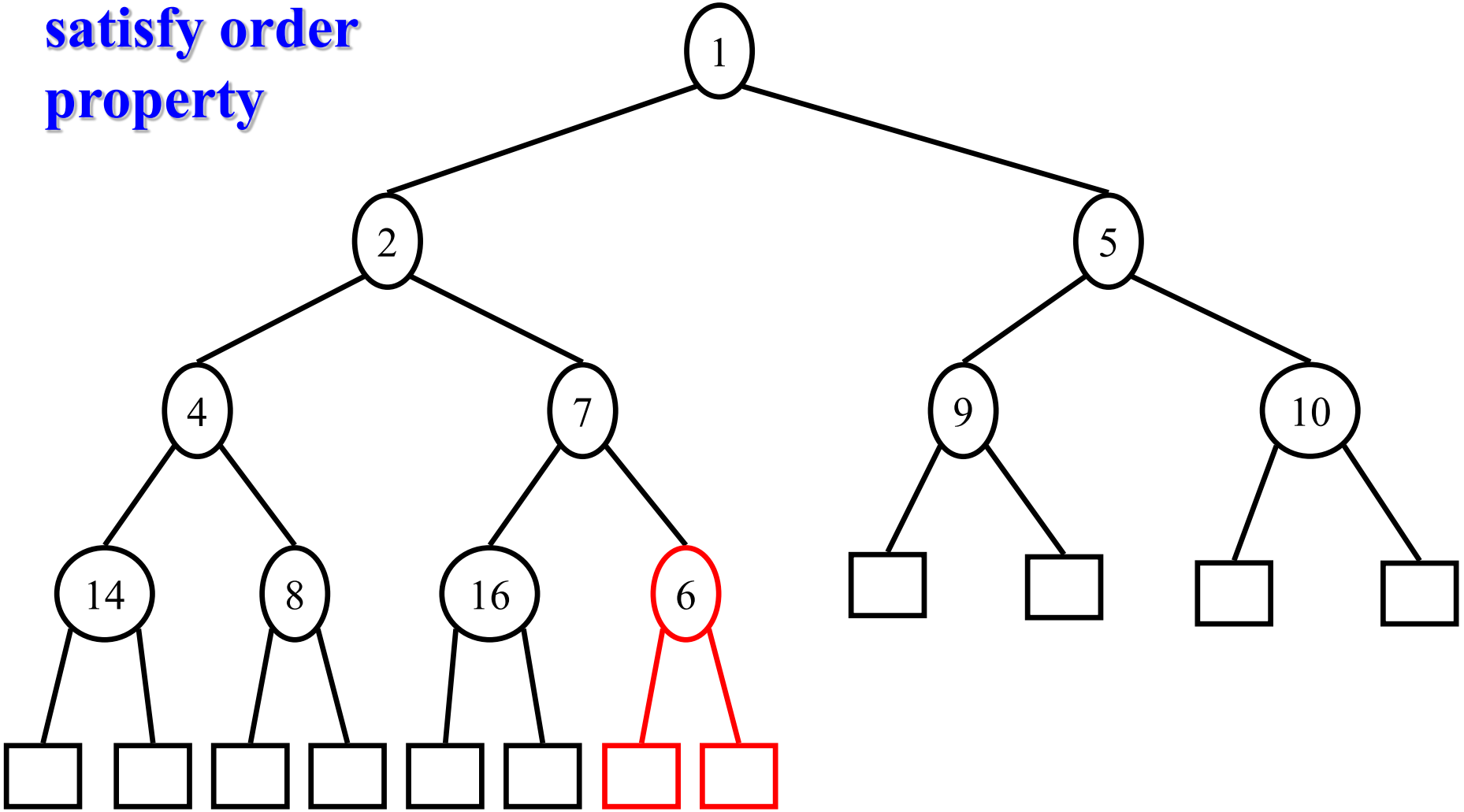
insert(6)



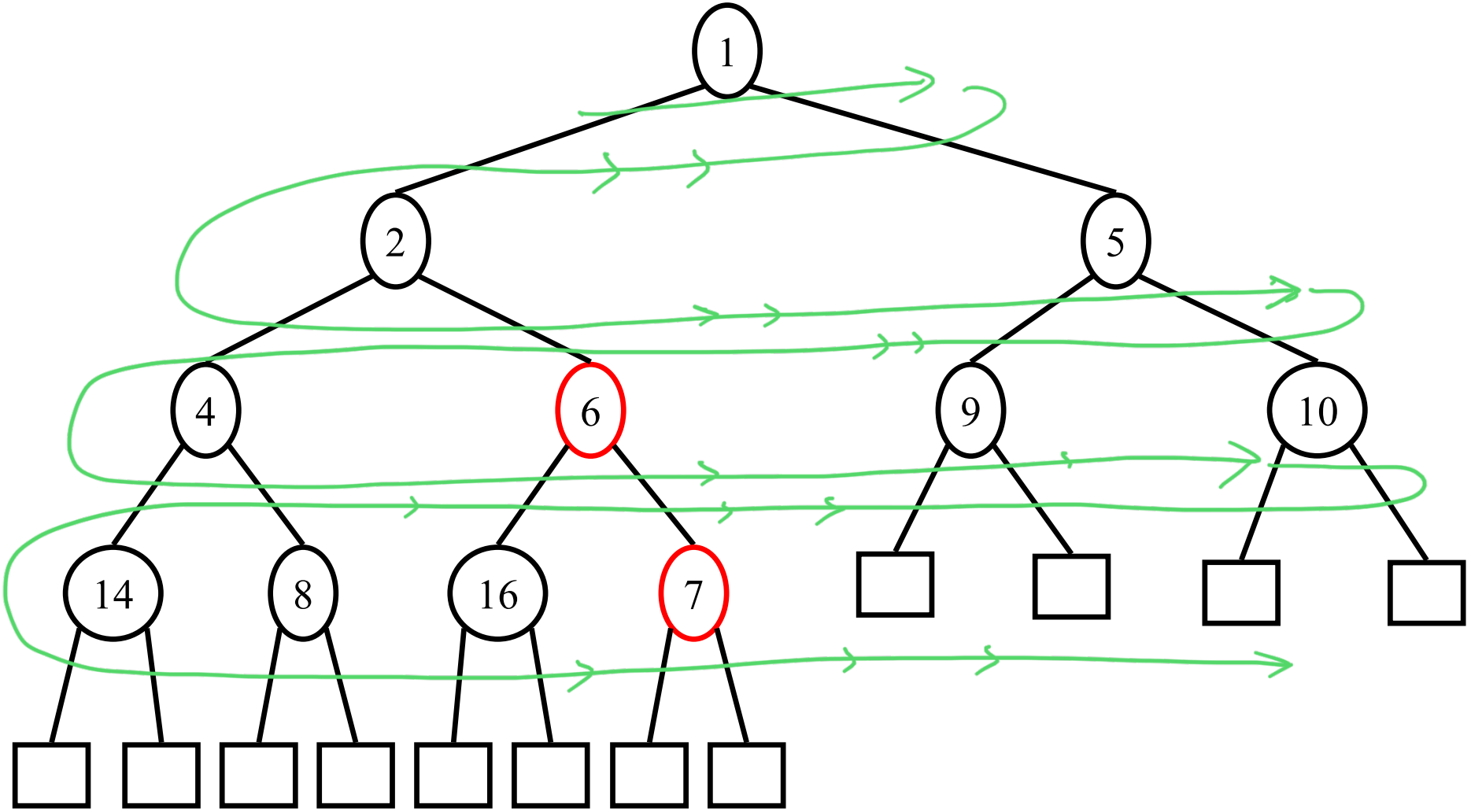
Insert element with key 6; shape property satisfied



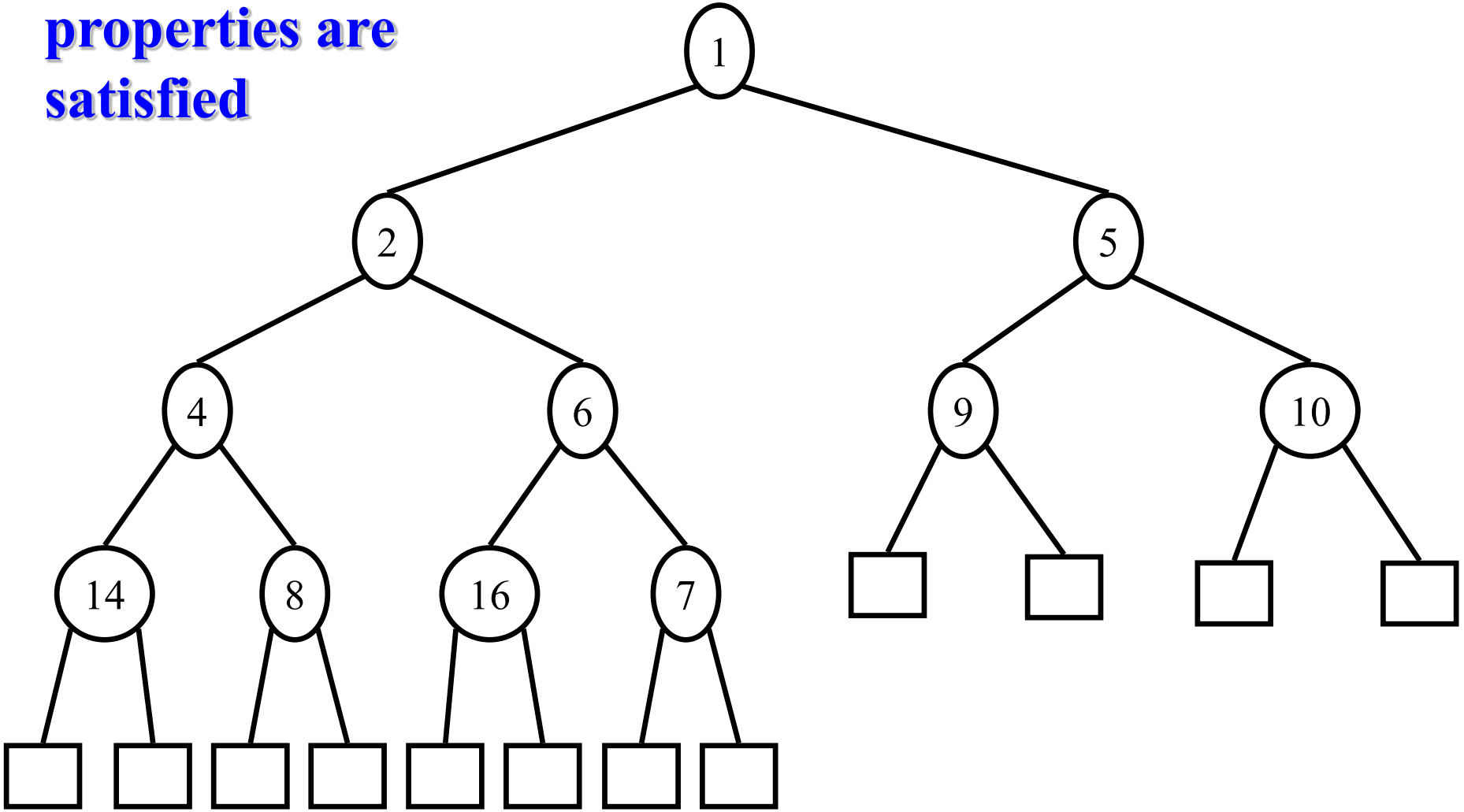
Bubble up 6 to satisfy order property



Order property satisfied



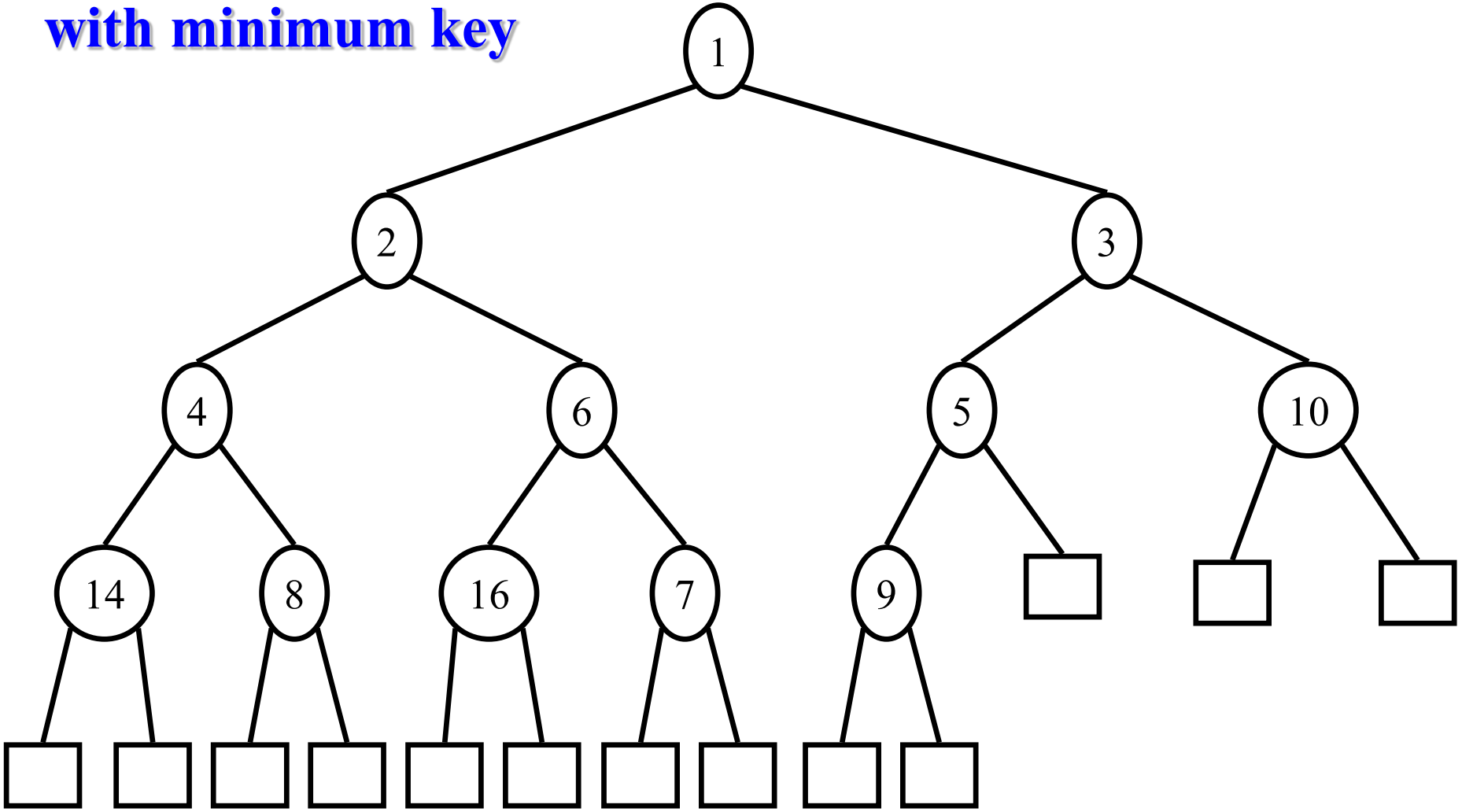
Shape and order properties are satisfied



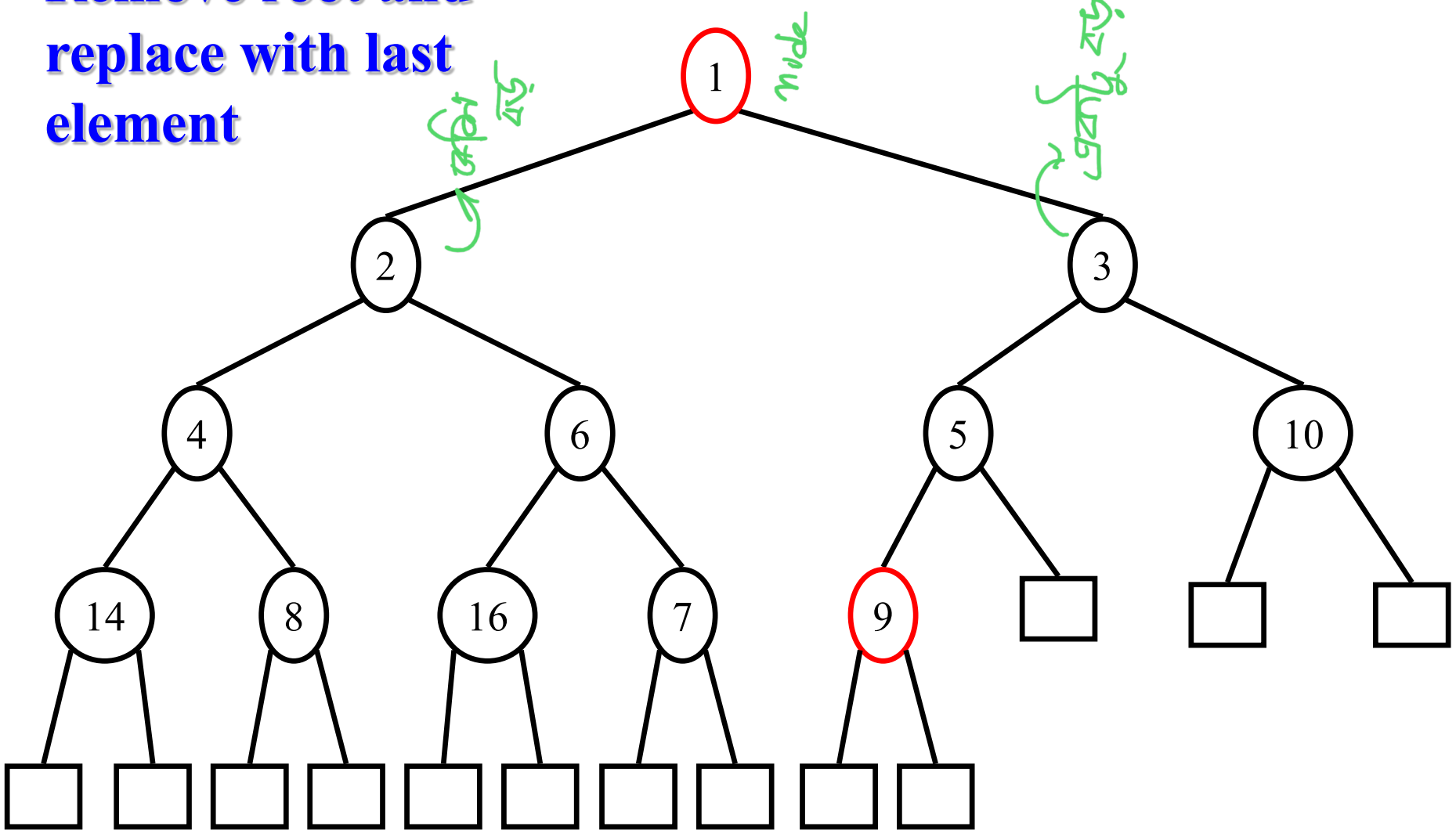
Removing Min Element from a Heap

- Remove root
- Satisfy shape property
  - Move last element to root spot
- Satisfy order property
  - Bubble root element down the tree until order property is restored

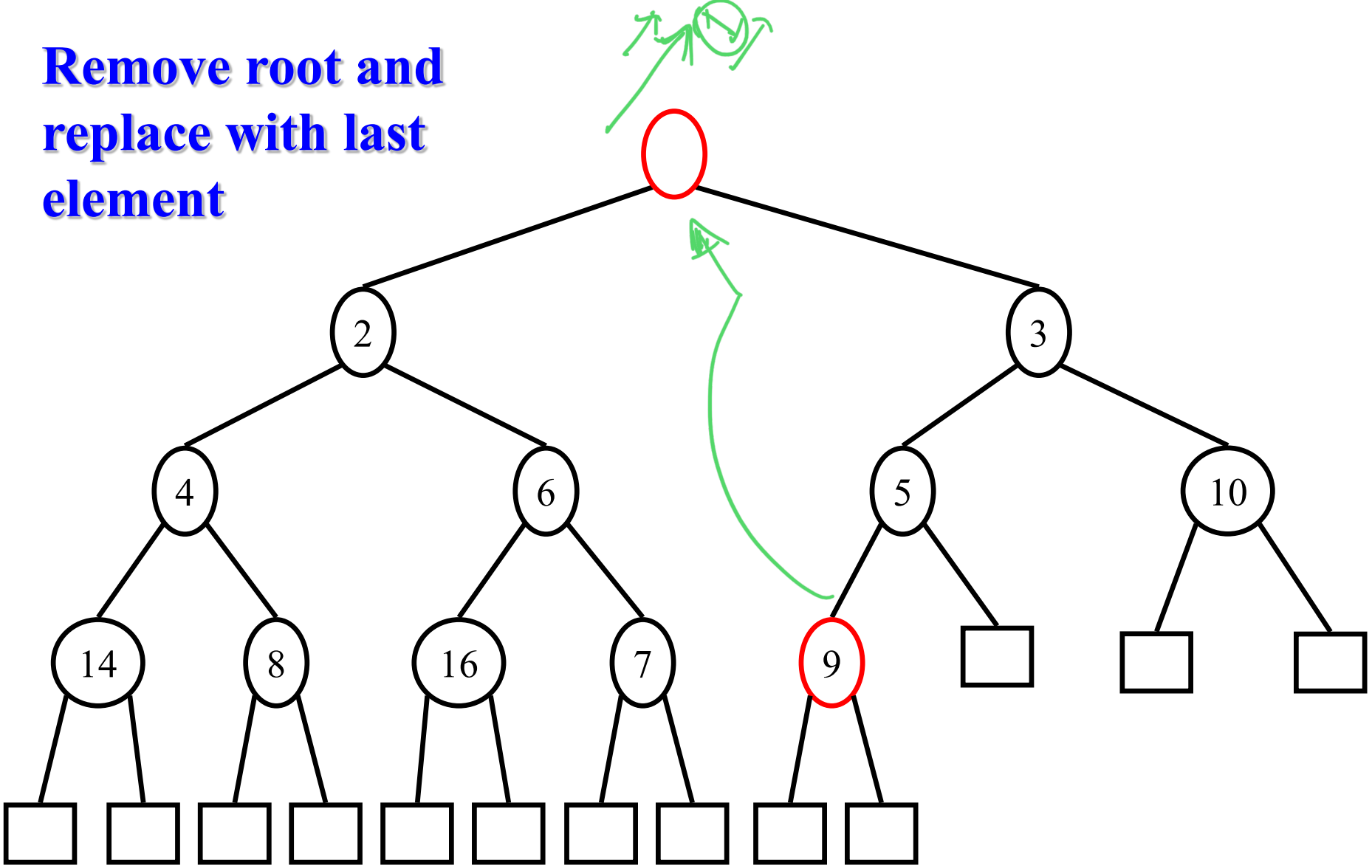
Remove element with minimum key



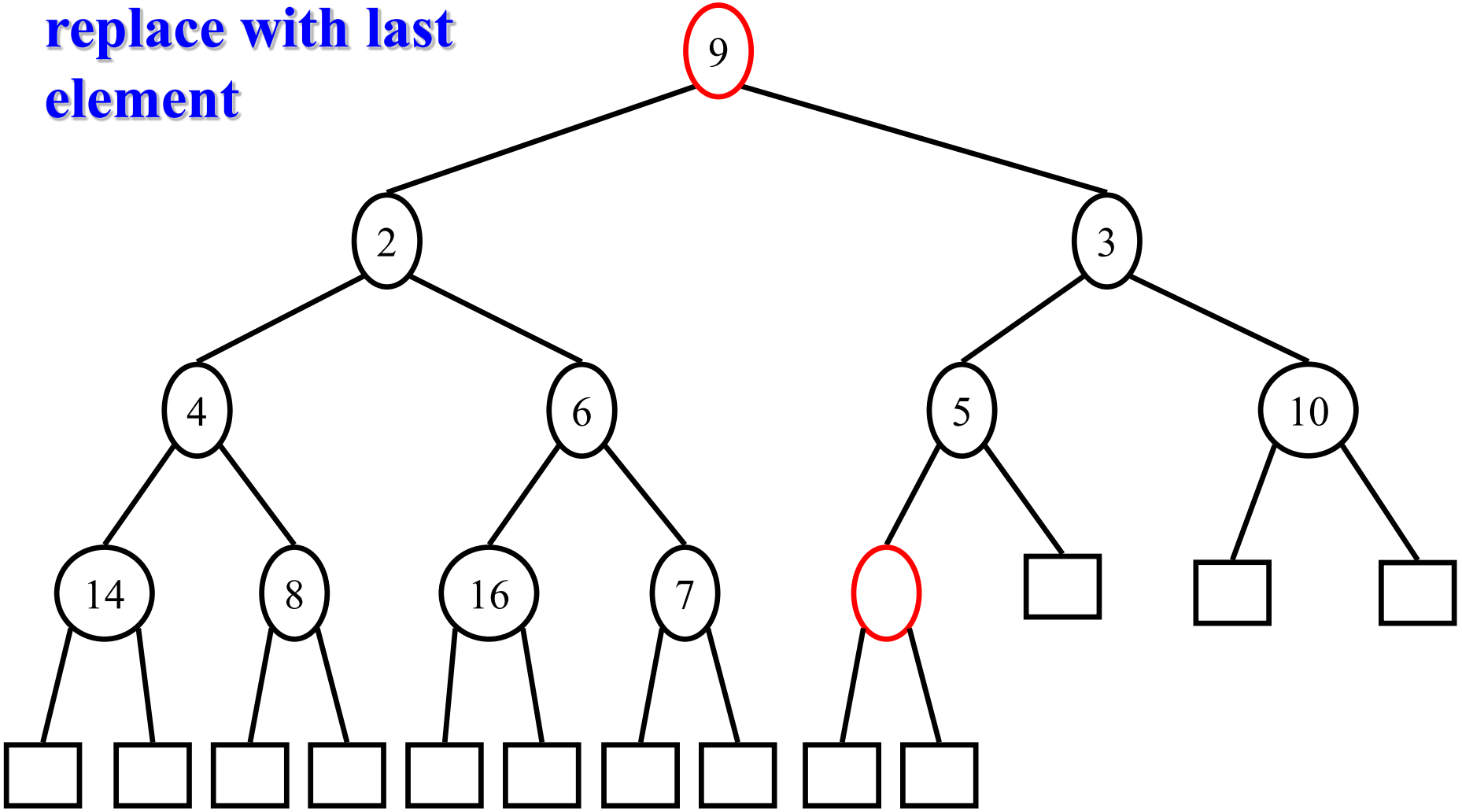
Remove root and replace with last element



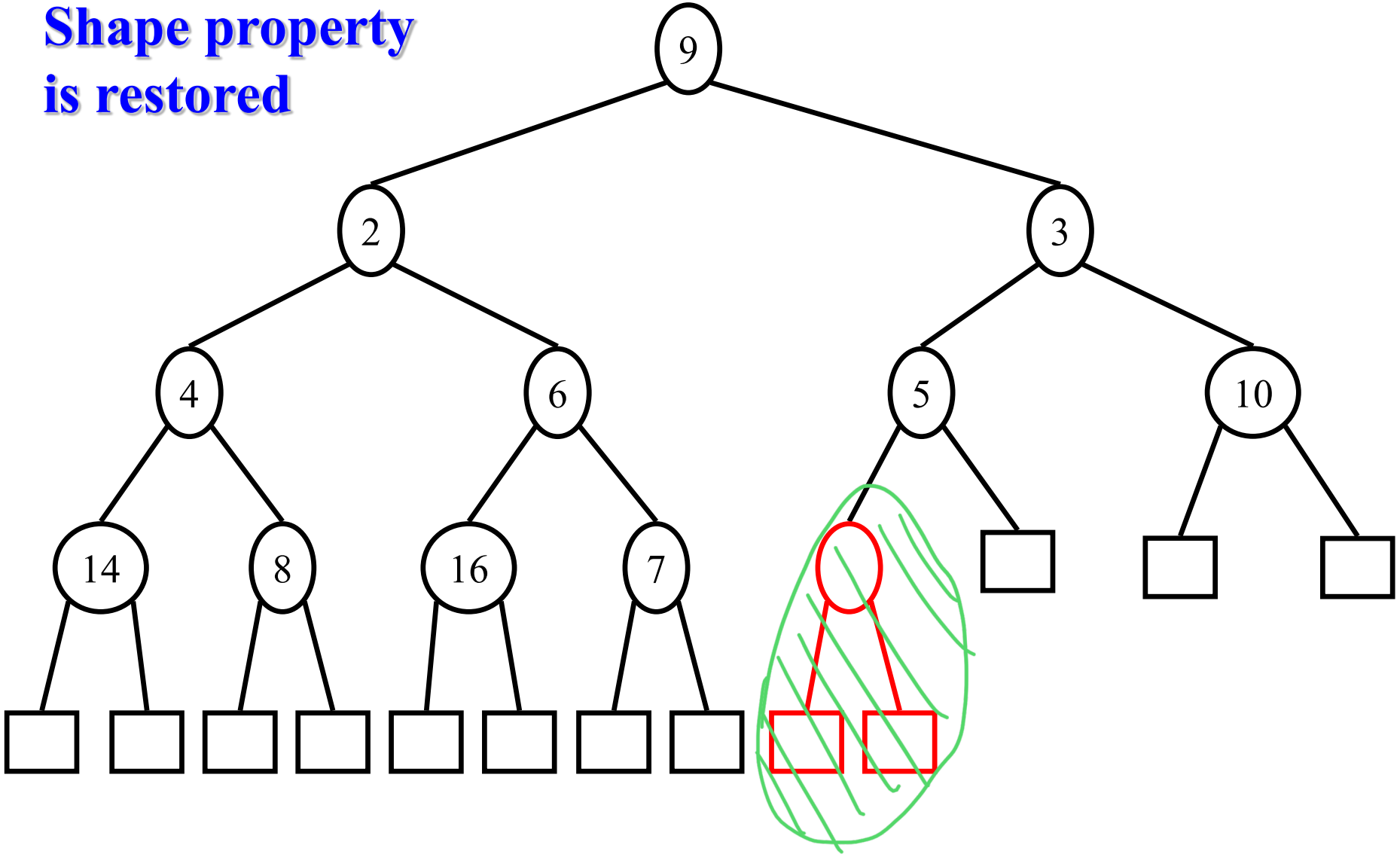
Remove root and replace with last element



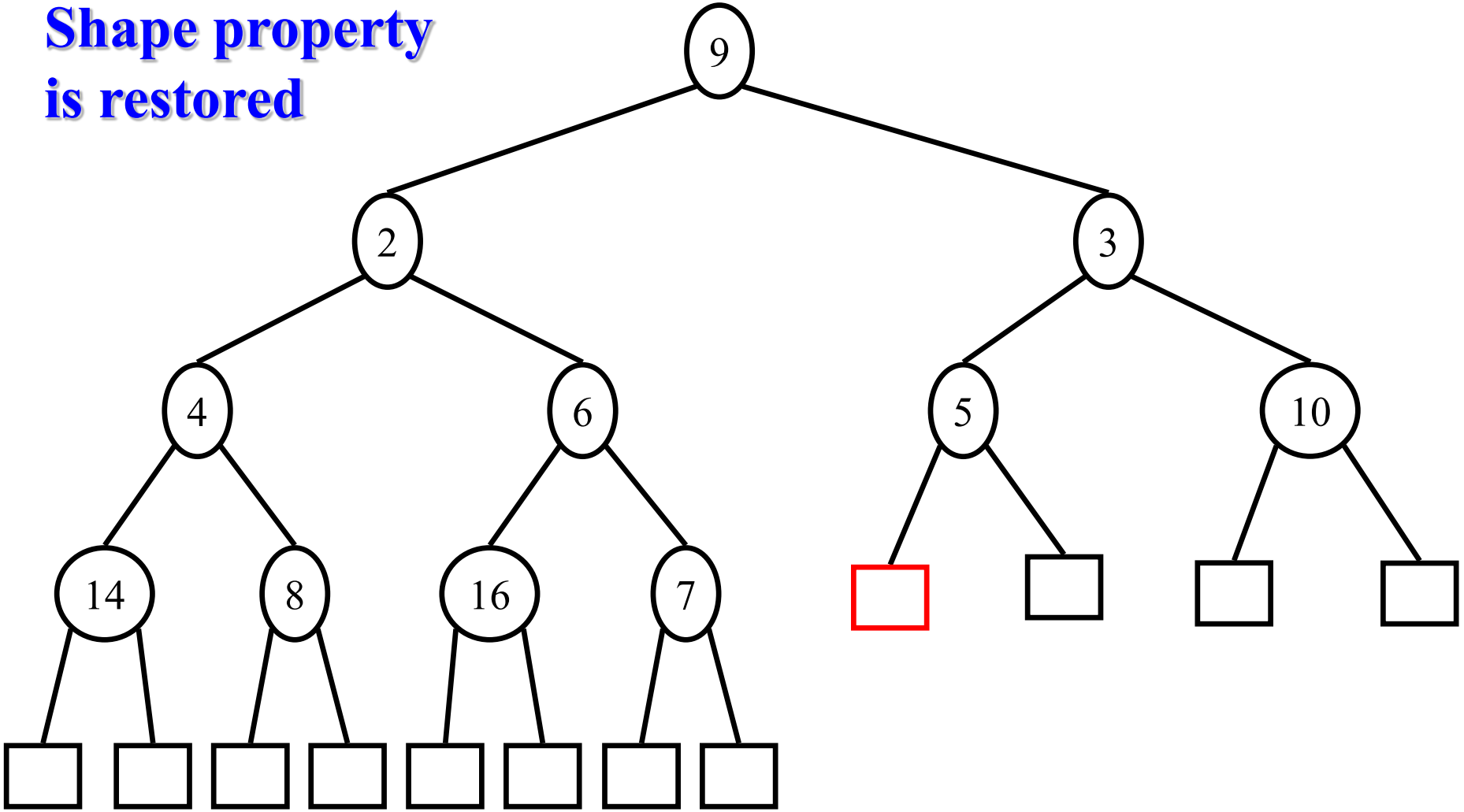
Remove root and replace with last element



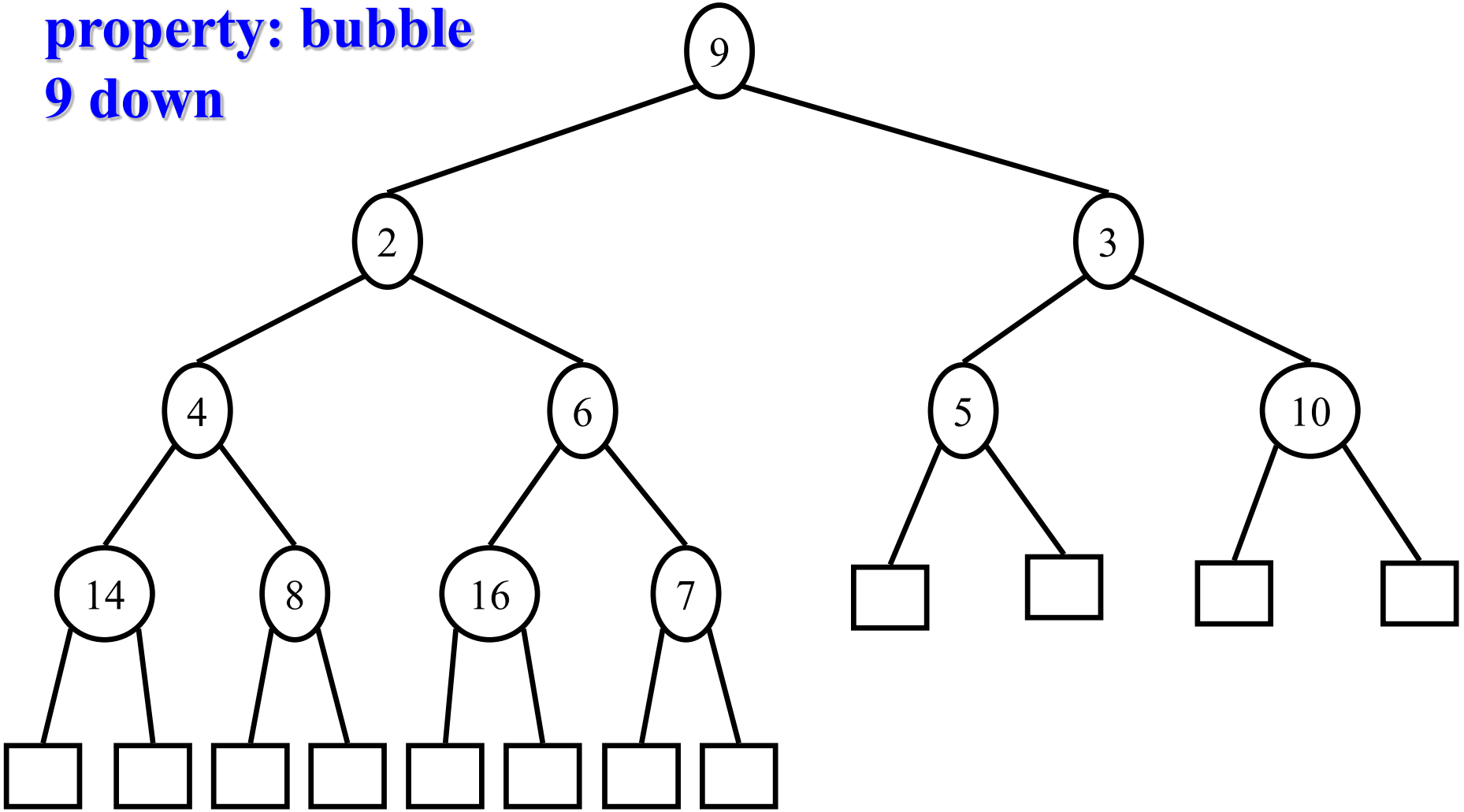
Delete last element
Shape property
is restored



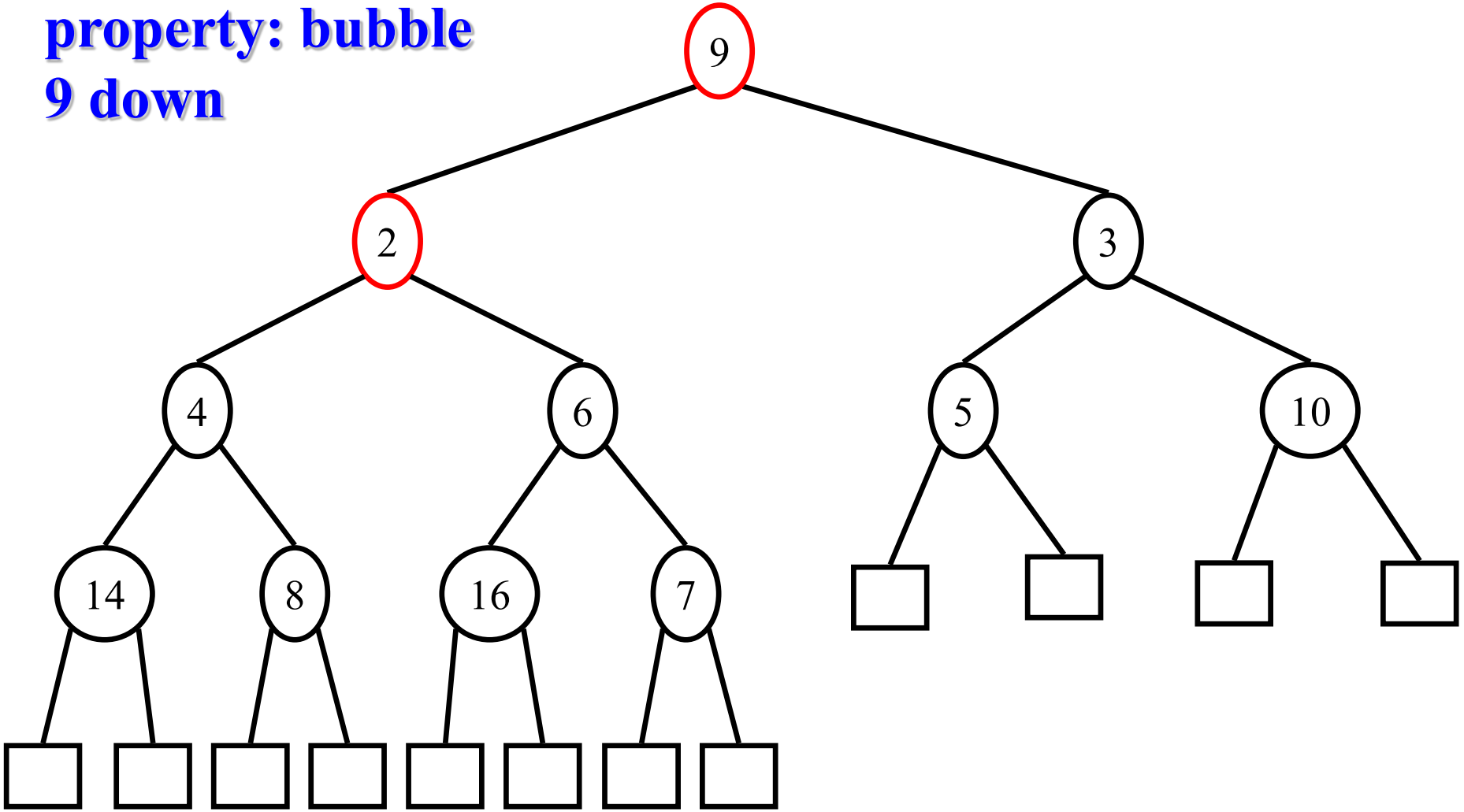
Delete last element
Shape property
is restored



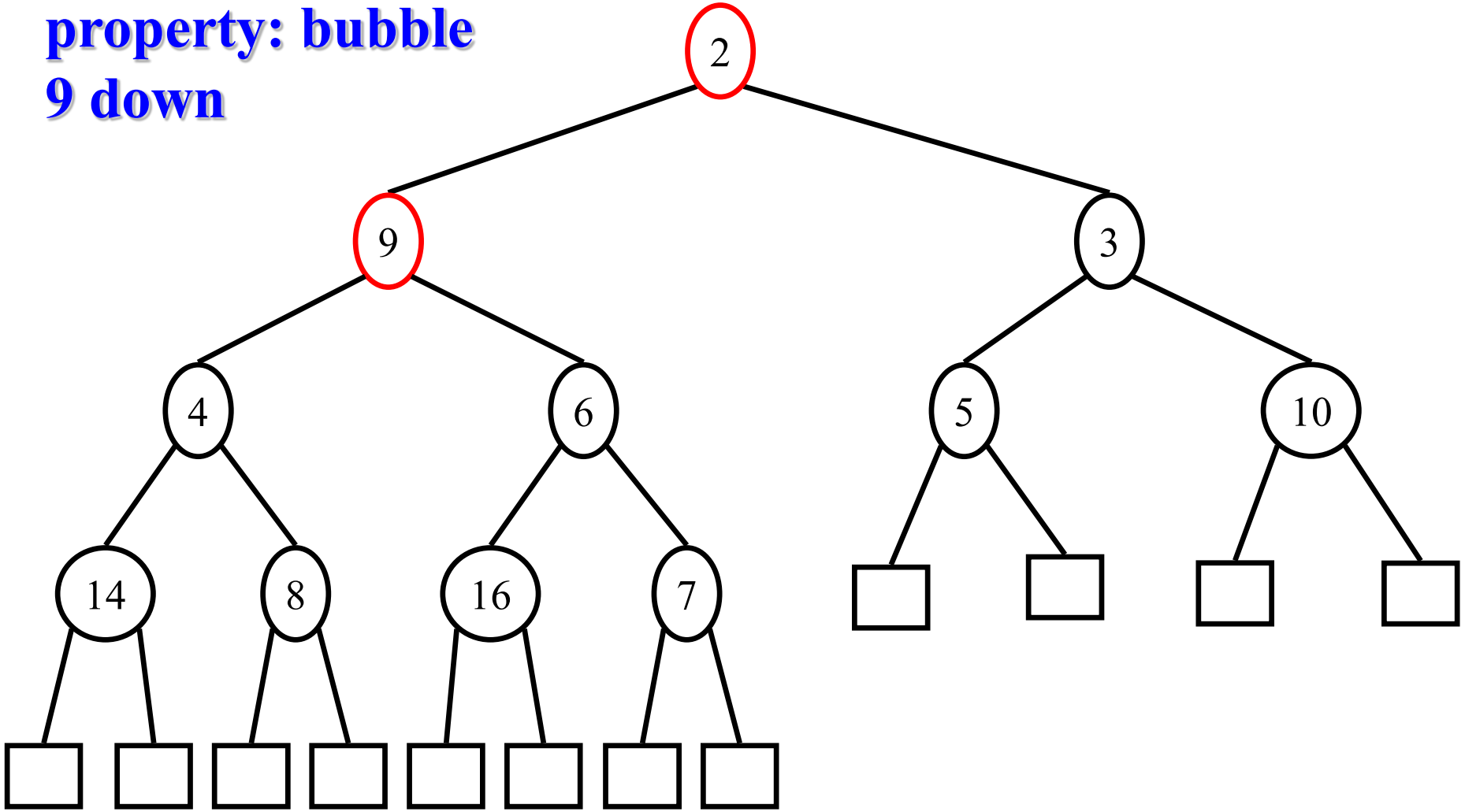
Reestablish order property: bubble 9 down



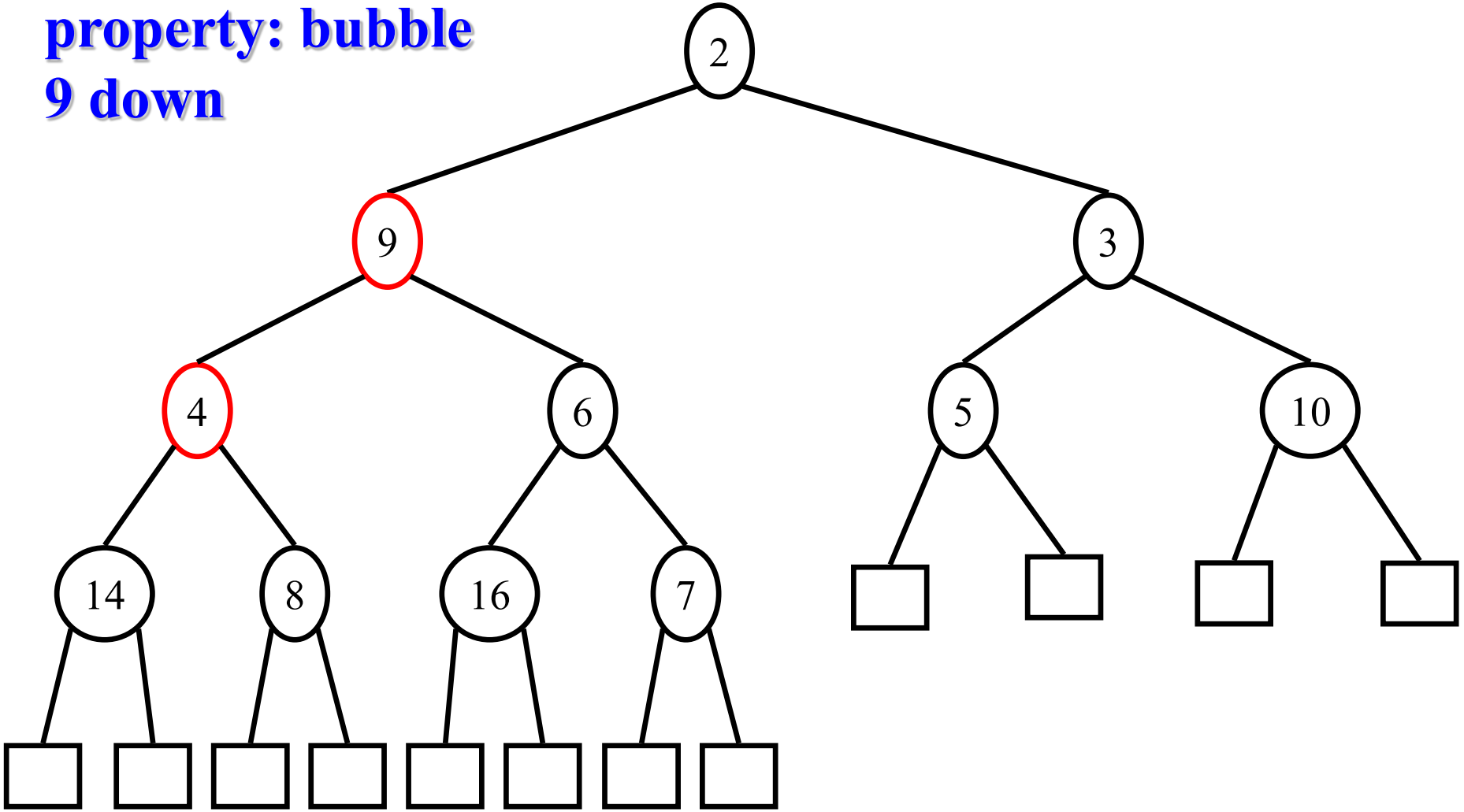
Reestablish order property: bubble 9 down



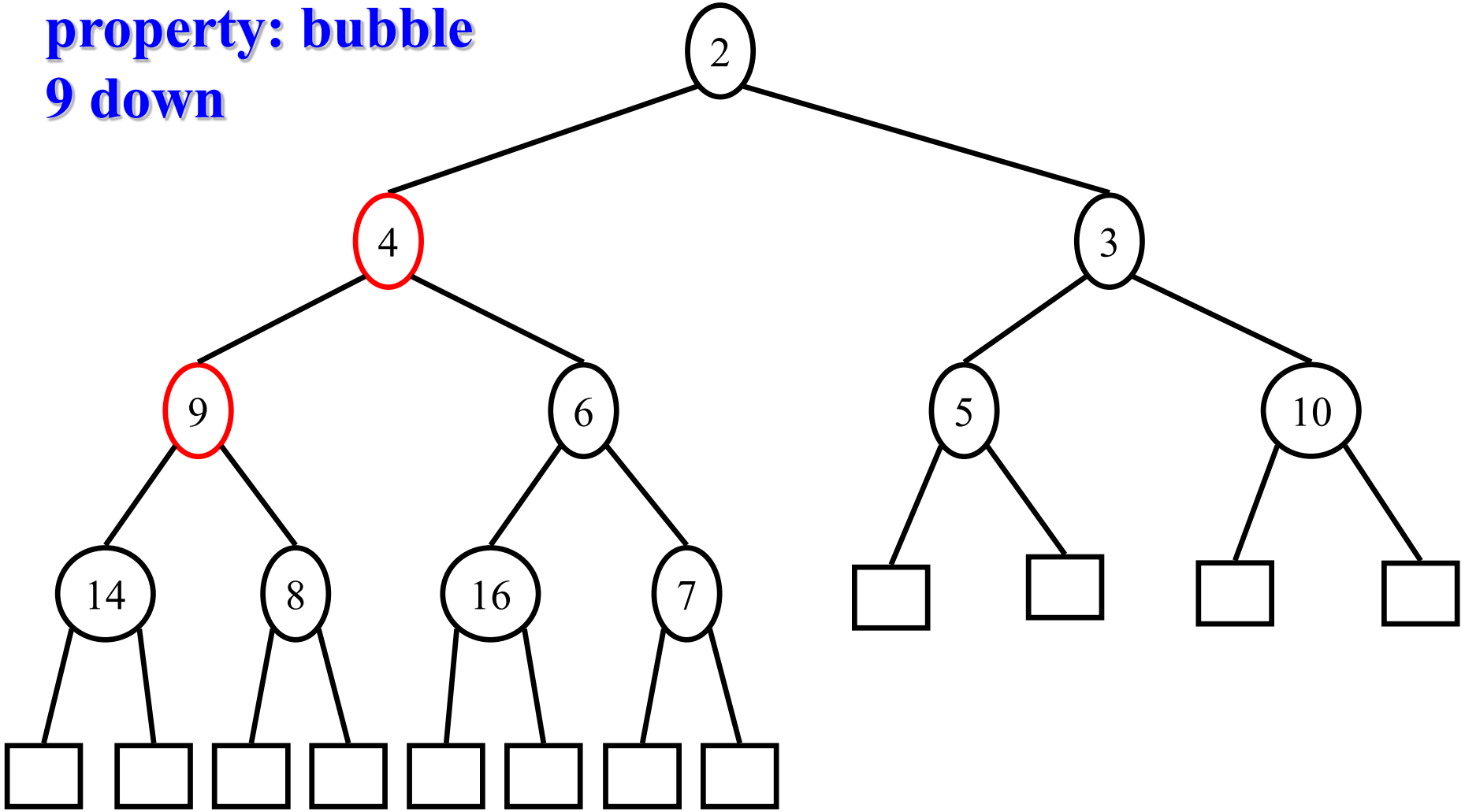
Reestablish order property: bubble 9 down



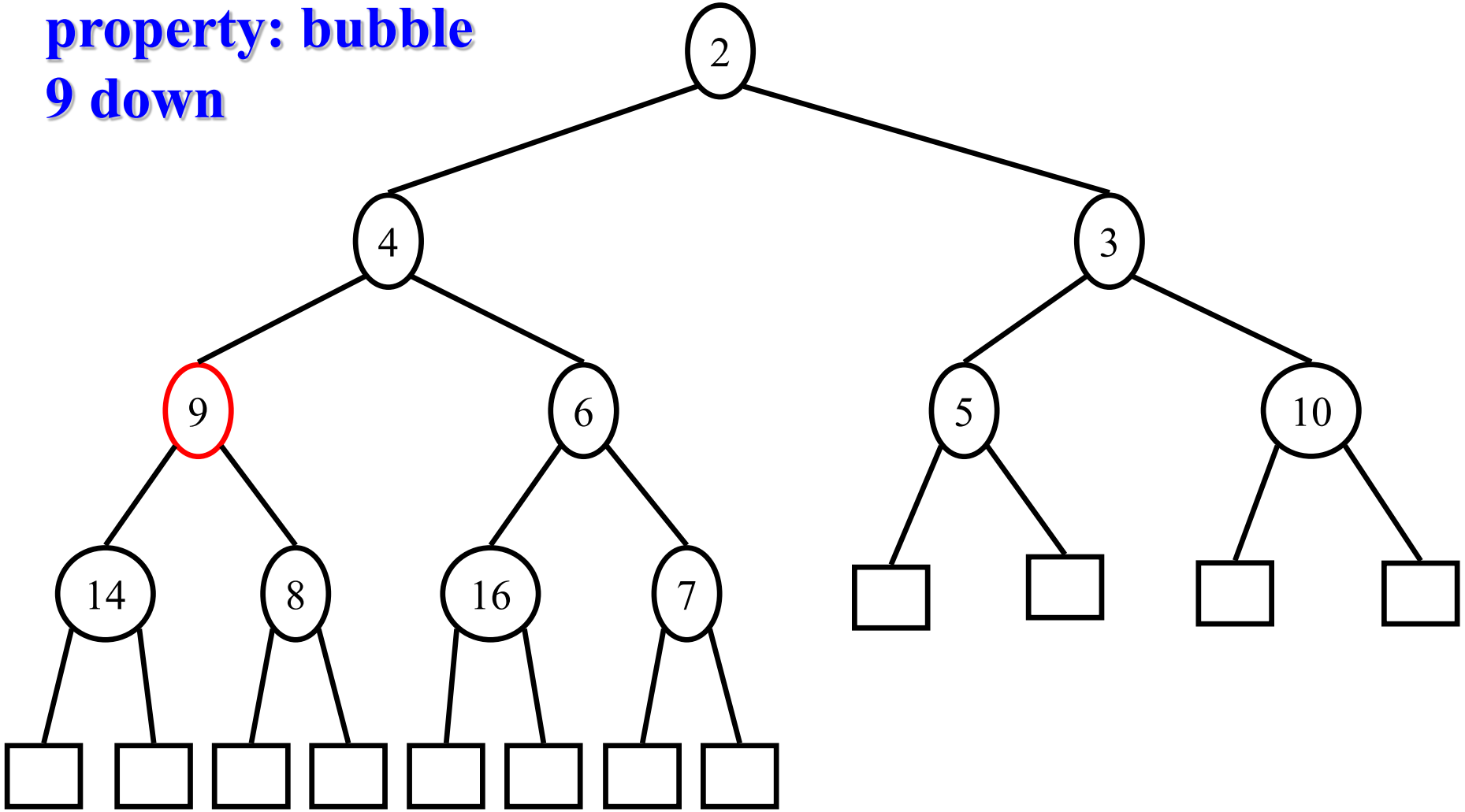
Reestablish order property: bubble 9 down



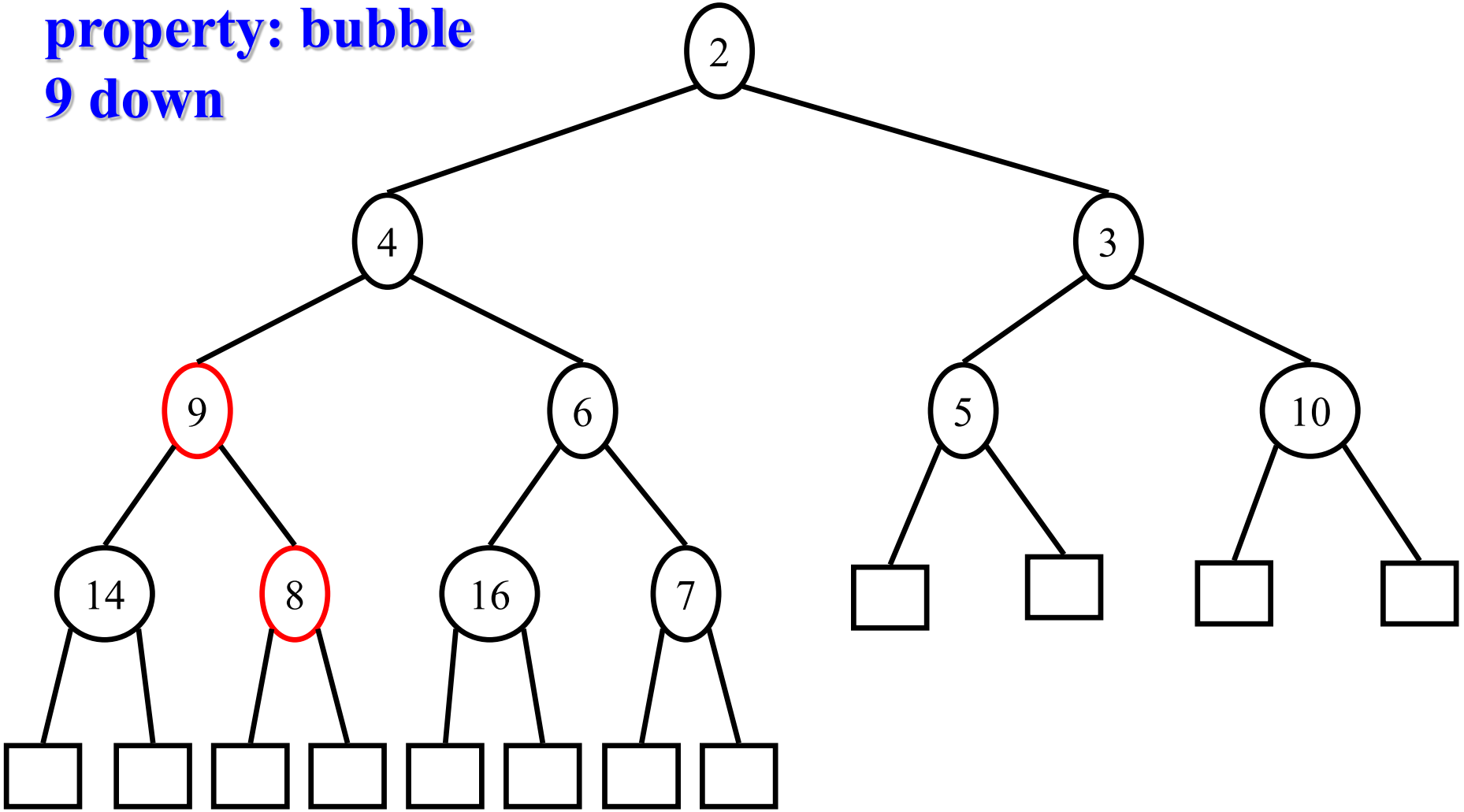
Reestablish order property: bubble 9 down



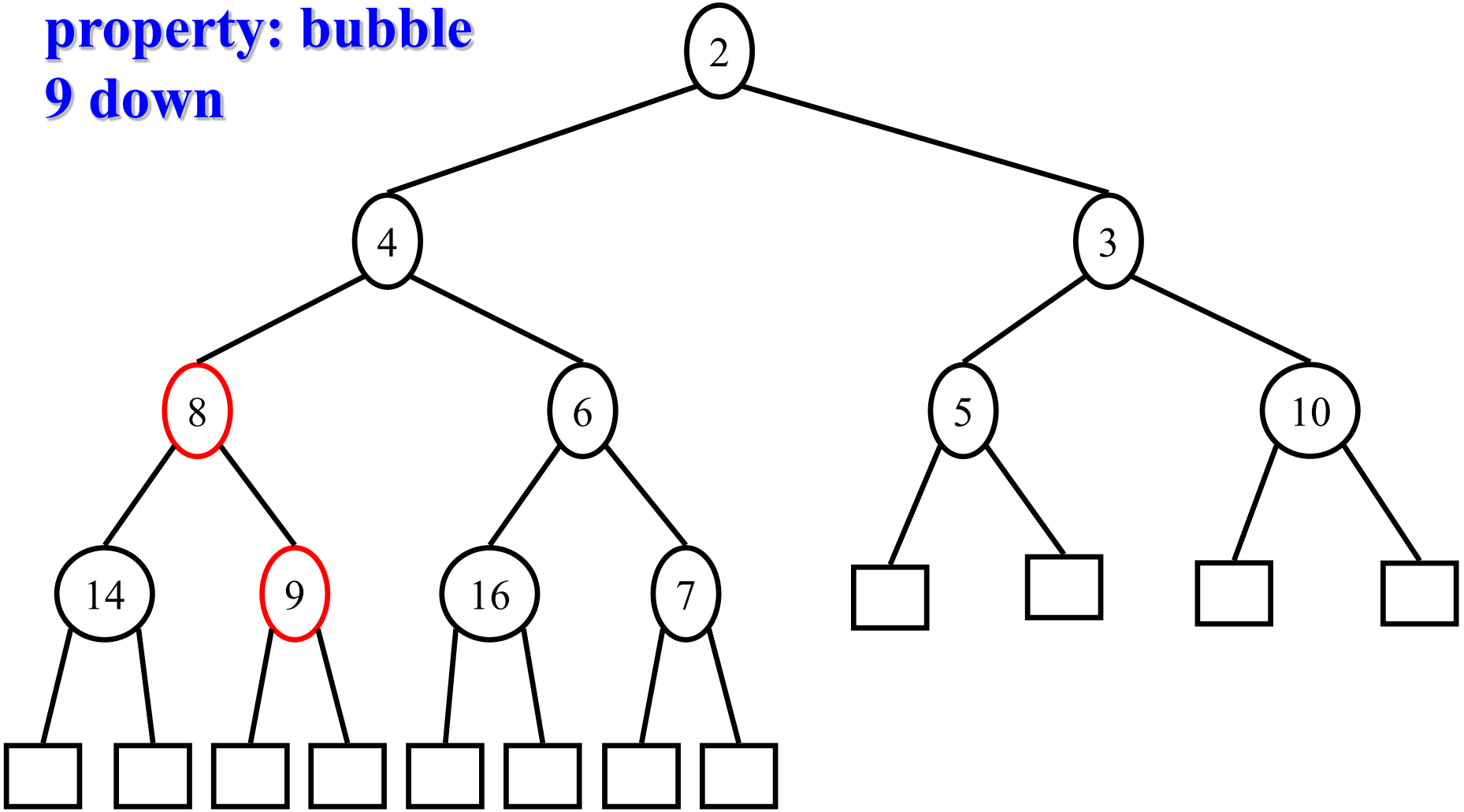
Reestablish order property: bubble 9 down



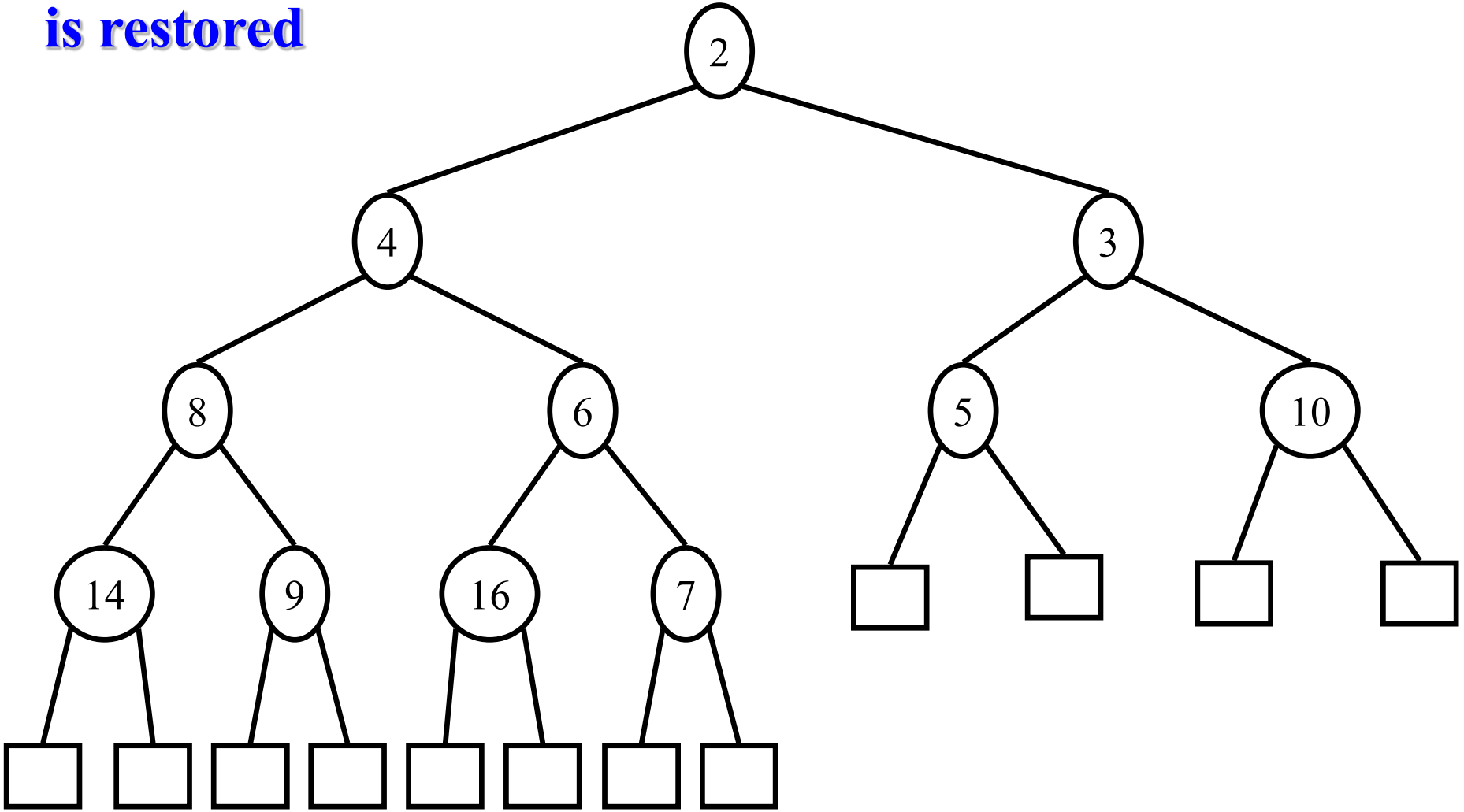
Reestablish order property: bubble 9 down



Reestablish order property: bubble 9 down



Order property is restored



Time Complexity of Heap Operations

- **Theorem**
  - The height of a balanced binary tree is O(log n) (i.e., $\log_2 n$)
- **Theorem**
  - The running times of the **Bubble up** and **Bubble down** operations in a Heap data structure are $O(\log n)$
- **Theorem**
  - The running times of the **insert** and **removeMin** operations in a Heap data structure are $O(\log n)$

Heapsort

1. Insert all elements from a given sequence *S* into a Priority Queue implemented as a Heap
2. Remove minimum element and insert element into *S*
3. Repeat Step 2 until every element is inserted back in *S*

Building up a heap

- Both loops are executed n times
- ***S*** indexing takes $O(1)$ time
- ***P*.removeMin()** and ***P*.insert()** take $O(\log n)$
- Thus, each loop takes $O(n \log n)$ time
- **Theorem**
  - The worst-case time complexity of Heapsort is $O(n \log n)$

Algorithm *Heapsort*(*S*, *P*)
Input*:** An n-element array, ***S, and a heap ***P***
Output*:** Array ***S sorted in increasing order

for $i \leftarrow 1$ **to** n **do**
    $e \leftarrow S[i]$
    P.insert(e, e)
for $i \leftarrow 1$ **to** n **do**
    $e \leftarrow$ *P.removeMin*()
    $S[i] \leftarrow e$

Heap Encoding

Parent of A[k] is at A[k/2]

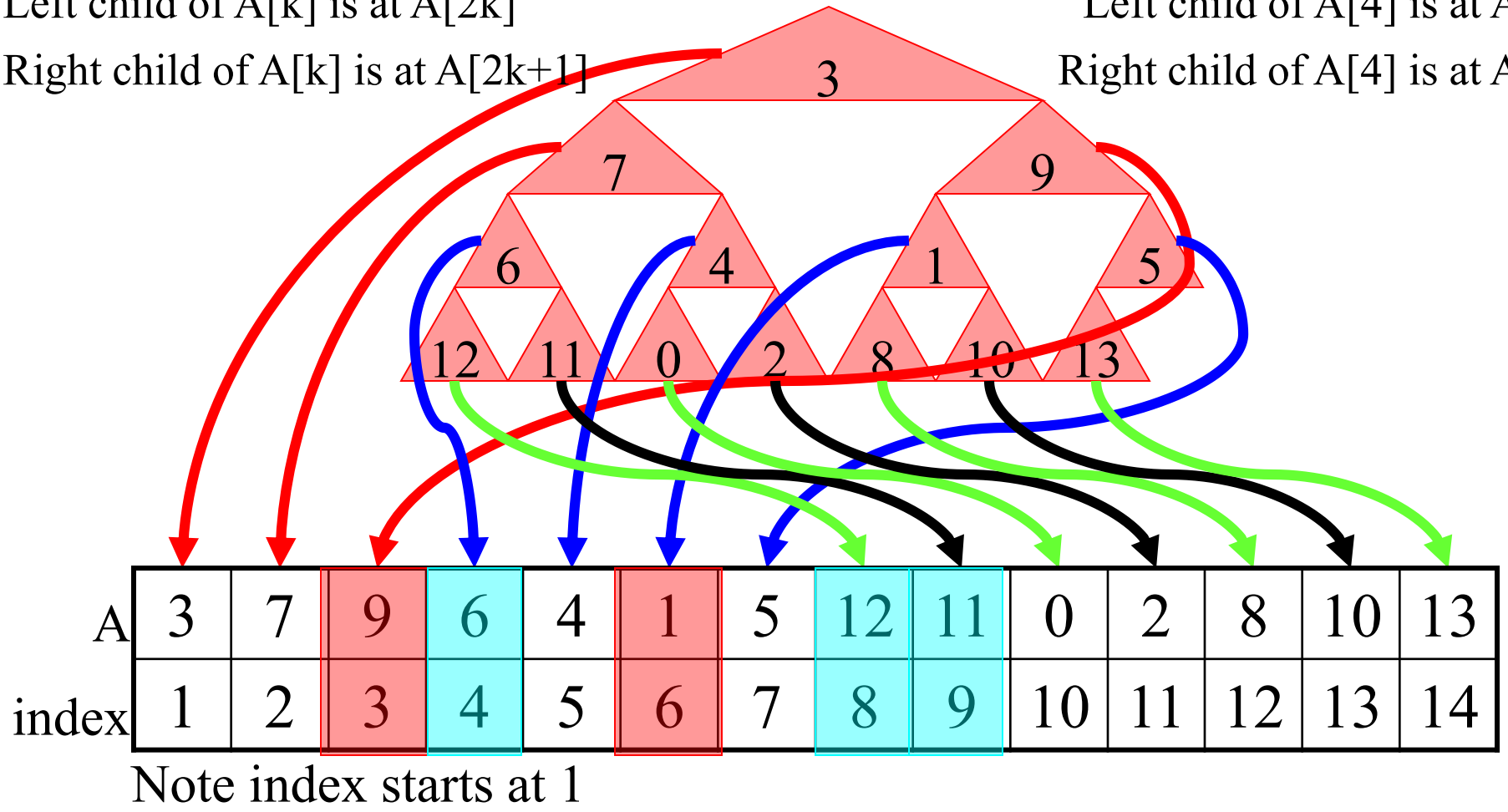
Left child of A[k] is at A[2k]

Right child of A[k] is at A[2k+1]

Parent of A[6] is at A[3]

Left child of A[4] is at A[8]

Right child of A[4] is at A[9]



A	3	7	9	6	4	1	5	12	11	0	2	8	10	13
index	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Note index starts at 1