CISC 235: Topic 7

Priority Queues and Binary Heaps

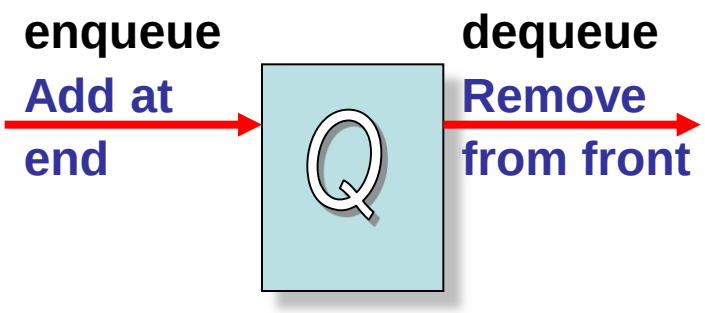
Outline

- Priority Queues
- Binary Heaps
  - Ordering Properties
  - Structural Property
- Array Representation
- Algorithms and Analysis of Complexity
    - insert
    - minimin
    - extractMin
    - buildHeap

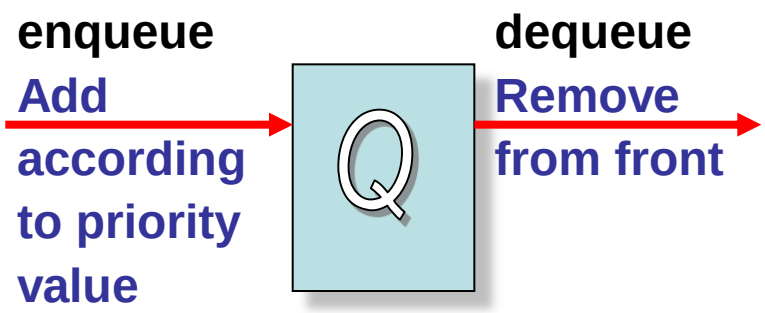
Priority Queues

A queue that is ordered according to some priority value

Standard Queue



Priority Queue



Example Applications

Line-up of Incoming Planes at Airport

Possible Criteria for Priority?

Operating Systems Priority Queues?

Several criteria could be mapped to a priority status

Min-Priority Queue Operations

insert(S, x) – Inserts element x into set S, according to its priority

minimum(S) – Returns, but does not remove, element of S with the smallest key

extractMin(S) – Removes and returns the element of S with the smallest key

Possible Implementations?

Binary Heaps

A binary tree with an ordering property and a structural property

Ordering Property: Min Heap

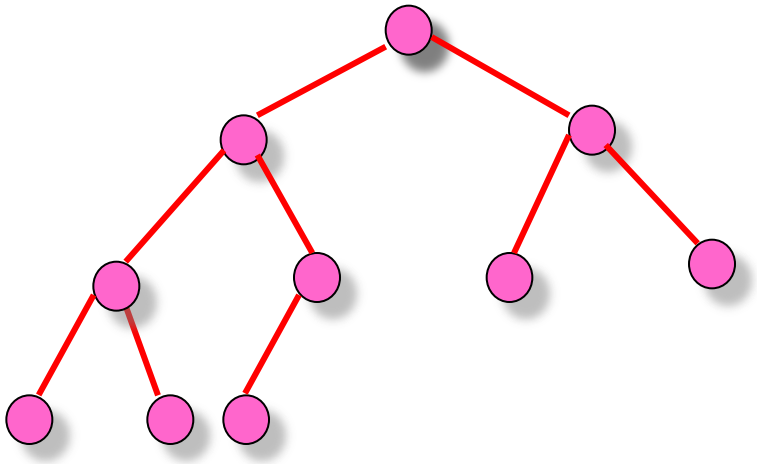
- The element in the root is less than or equal to all elements in both its sub-trees
- Both of its sub-trees are Min Heaps

Ordering Property: Max Heap

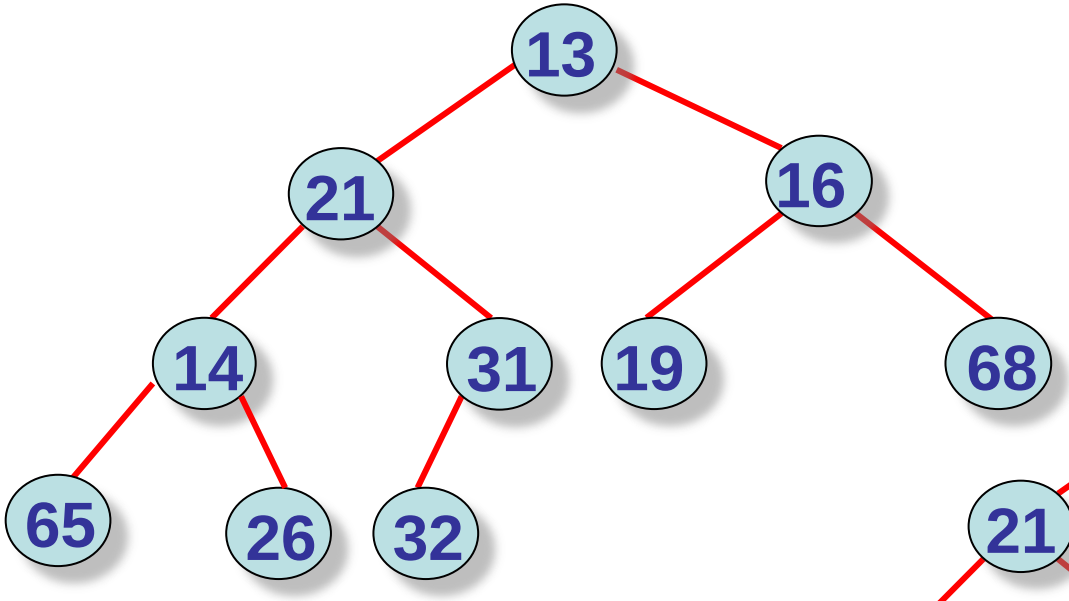
- The element in the root is greater than or equal to all elements in both its sub-trees
- Both of its sub-trees are Max Heaps

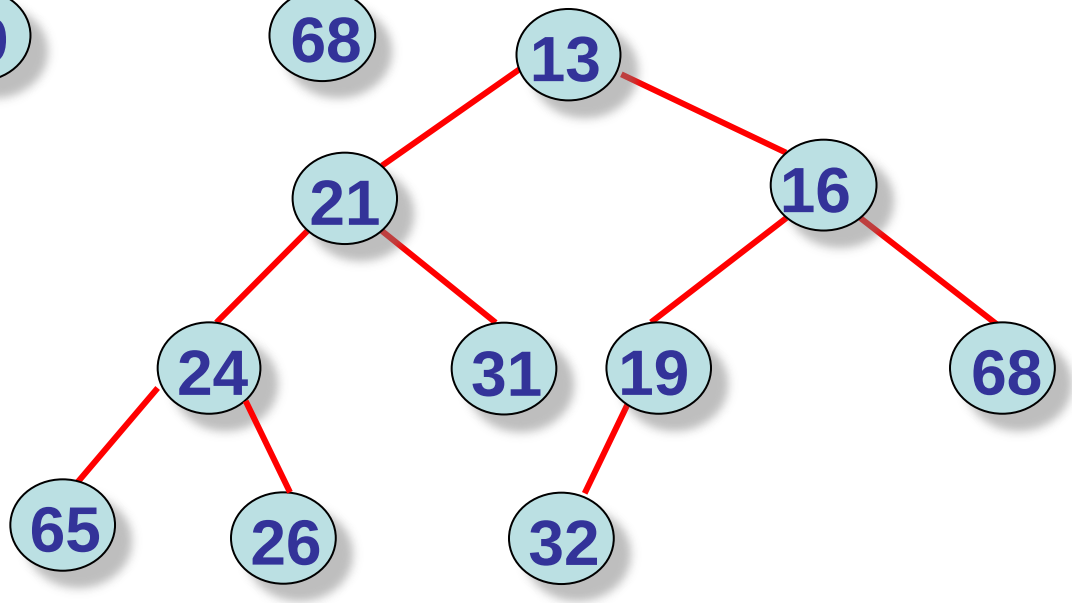
Binary Heap Structural Property

A binary heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right with no missing nodes.



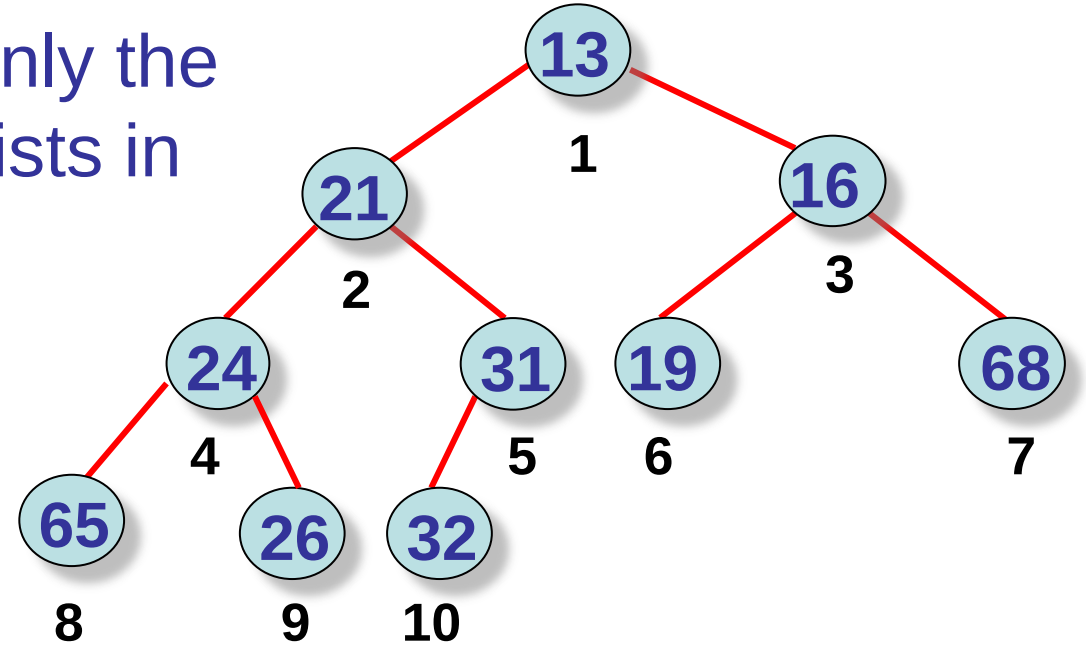
Which Tree is a Min Heap?





A Min Heap and its Array Representation

Note: Only the array exists in memory



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		13	21	16	24	31	19	68	65	26	32		

Locating Parents & Children

```
parent( i )
        return ⌊ i / 2 ⌋

left( i )
        return 2i

right( i )
        return 2i + 1
```

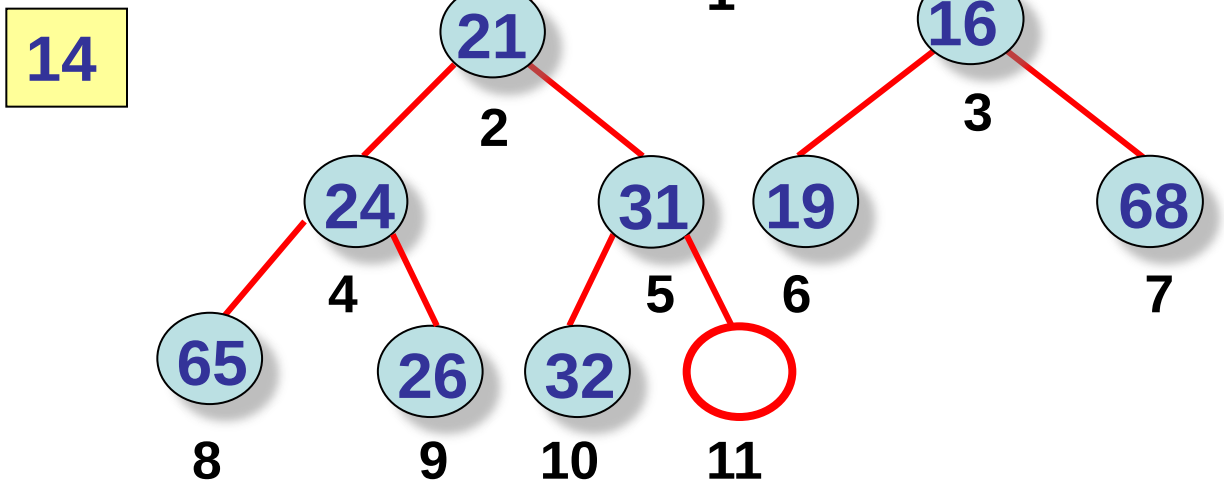
Min Heap Operations: Insert

1. Find the next position at which to insert – after the "last" element in the heap – and create a "hole" at that postion
2. "Bubble" the hole up the tree by moving its parent element into it until the hole is at a position where the new element ≤ its children and ≥ its parent:

   the percolateUp() operation
3. Insert the new element in the hole

Insert Step 1: Create hole at next insert position

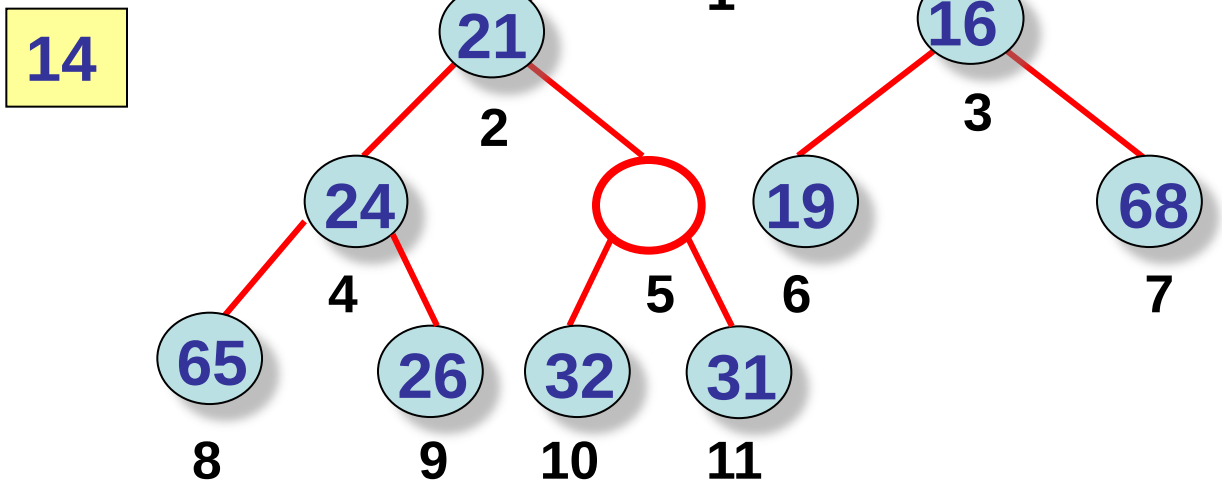
Insert 14



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		13	21	16	24	31	19	68	65	26	32		

Insert Step 2: Bubble hole up to where element ≤ its children and ≥ its parent

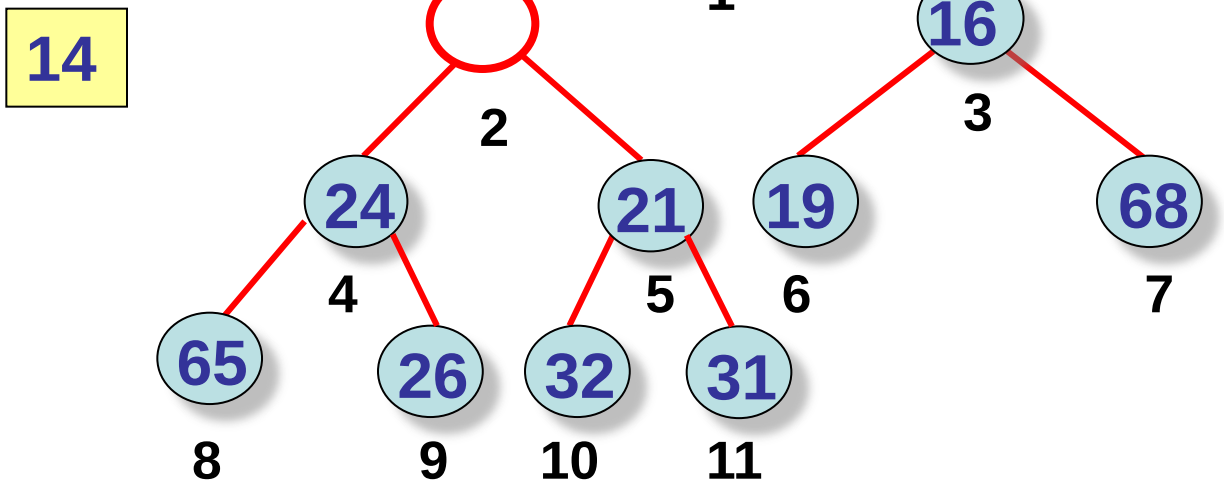
Insert 14



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		13	21	16	24		19	68	65	26	32	31	

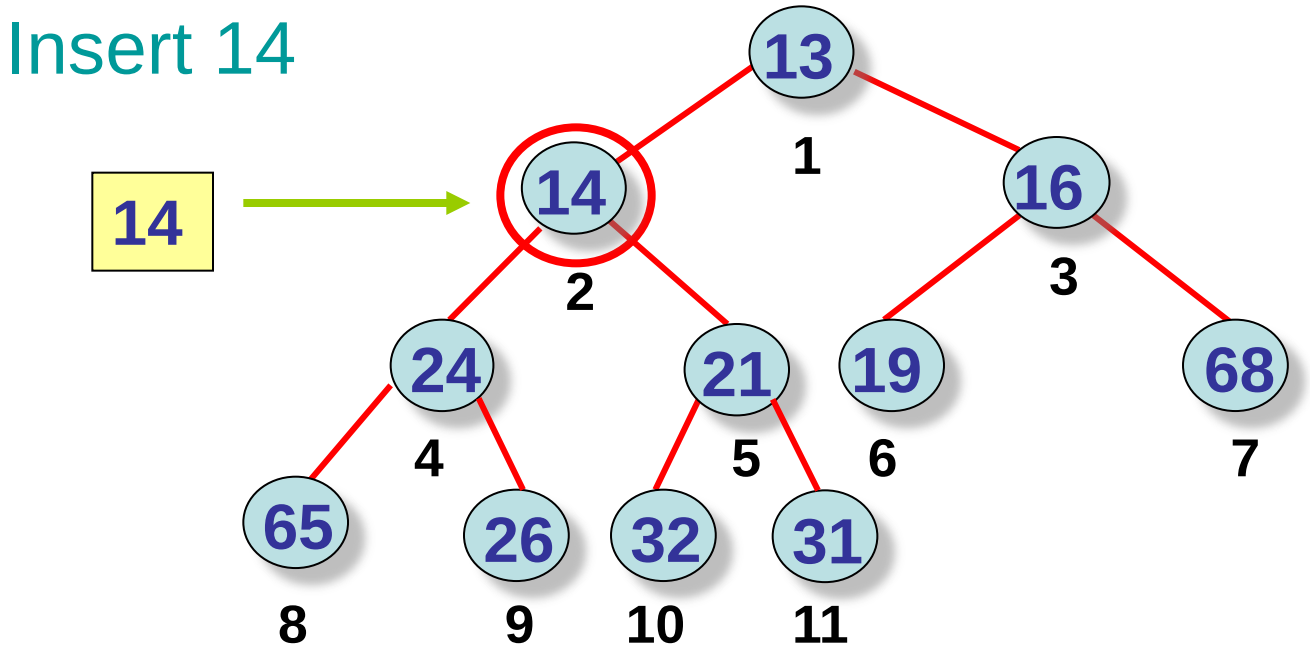
Insert Step 2: Bubble hole up to where element ≤ its children and ≥ its parent

Insert 14



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		13	○	16	24	21	19	68	65	26	32	31	

Insert Step 3:
Insert the new element in the hole



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		13	14	16	24	21	19	68	65	26	32	31	

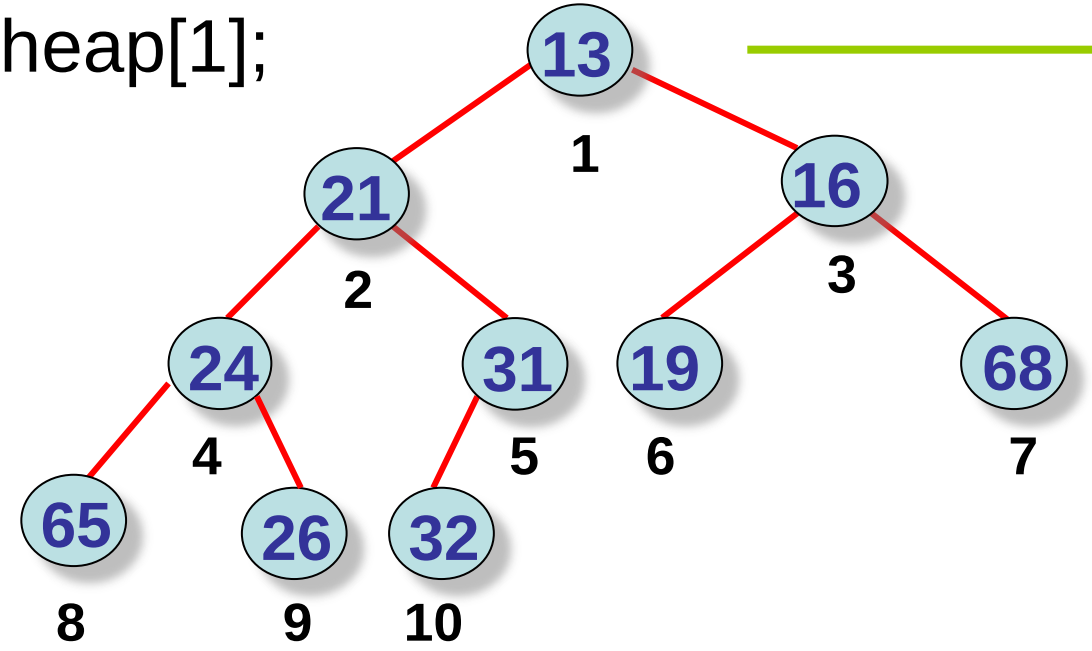
Analysis of Insert?

Worst-case complexity?

Min Heap Operations: Minimum

Return minimum at top of heap: O(1)

return heap[1];



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		13	21	16	24	31	19	68	65	26	32		

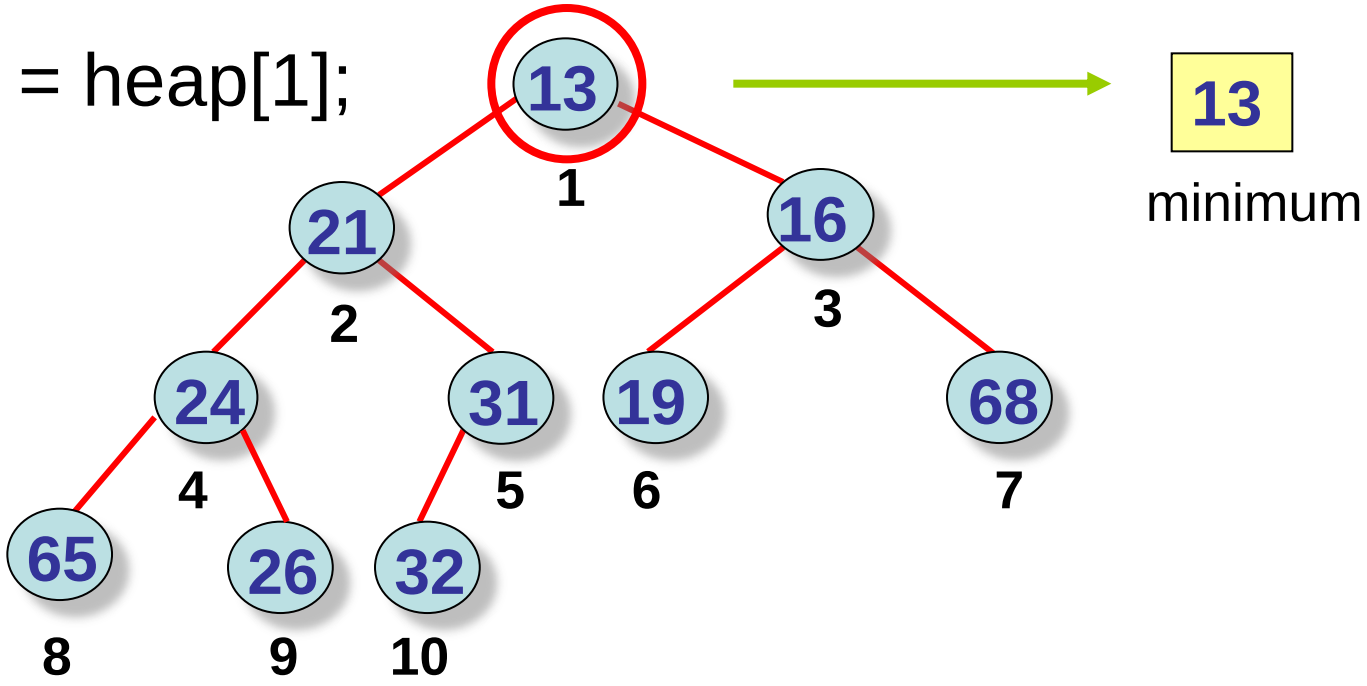
Min Heap Operations: ExtractMin

1. Copy the root (the minimum) to a temporary variable, thus creating a "hole" at the root.
2. Delete the last item in the heap by copying its element to a temporary location.
3. Find a place for that element by bubbling the hole down by moving its smallest child up until the element ≤ its children and ≥ its parent:

   the percolateDown() operation.
4. Place the former last element in that hole and return the former root.

ExtractMin Step 1:
Copy the root to a temporary variable

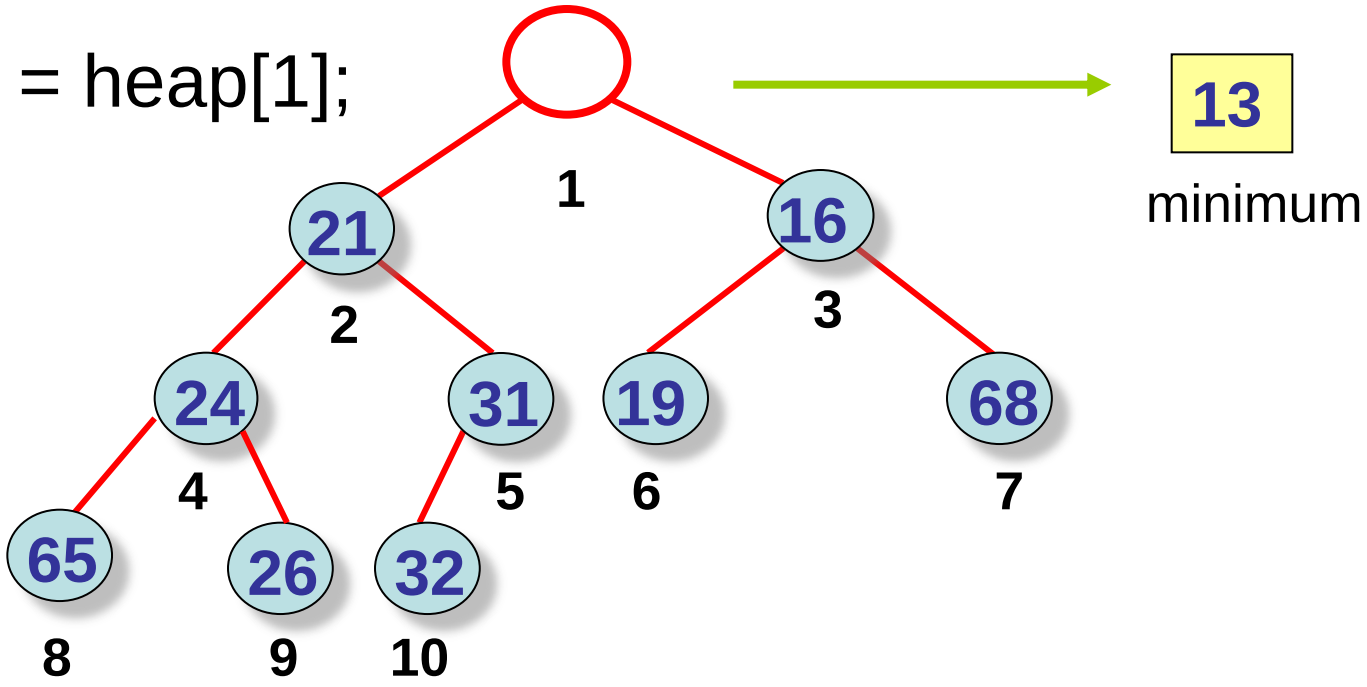
minimum = heap[1];



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		13	21	16	24	31	19	68	65	26	32		

ExtractMin Step 1: thus creating a “hole” at the root

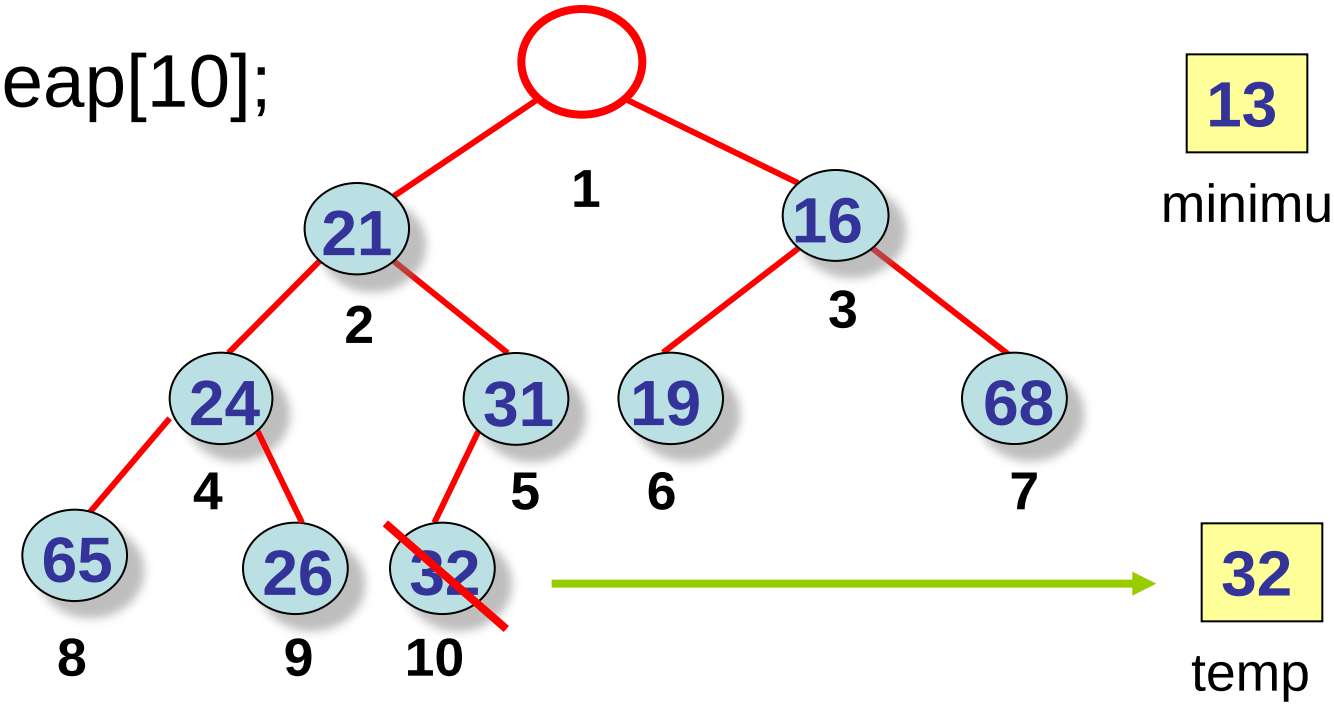
minimum = heap[1];



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap			21	16	24	31	19	68	65	26	32		

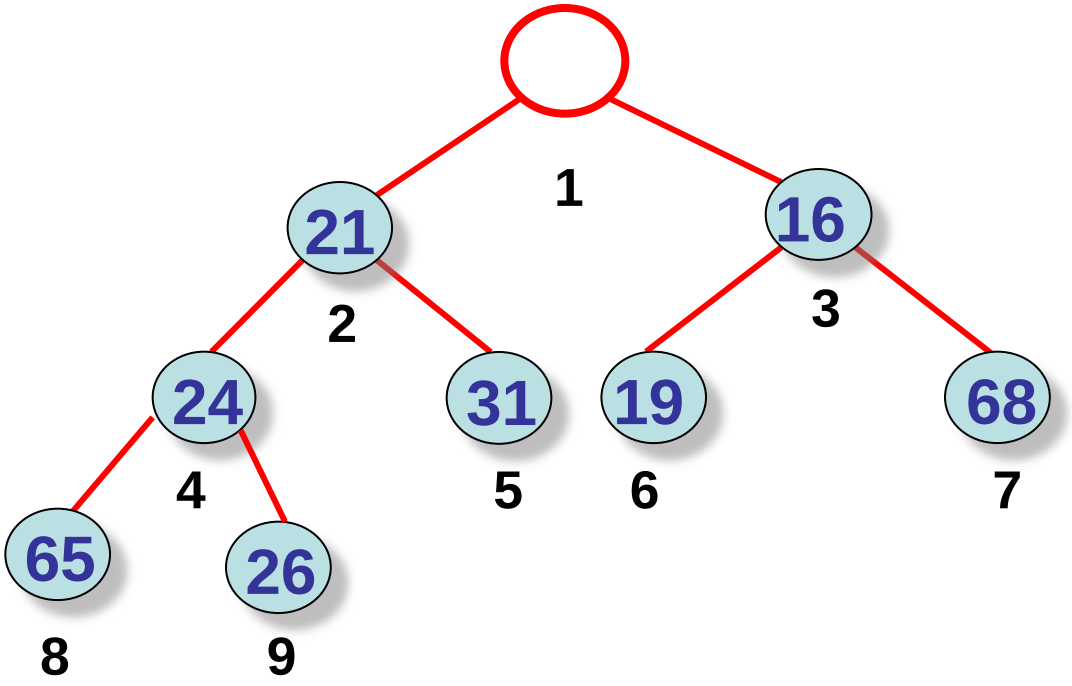
ExtractMin Step 2: Delete last item in heap by copying it to a temporary location

temp = heap[10];



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap			21	16	24	31	19	68	65	26	32		

ExtractMin Step 2: Delete last item in heap by copying it to a temporary location



13

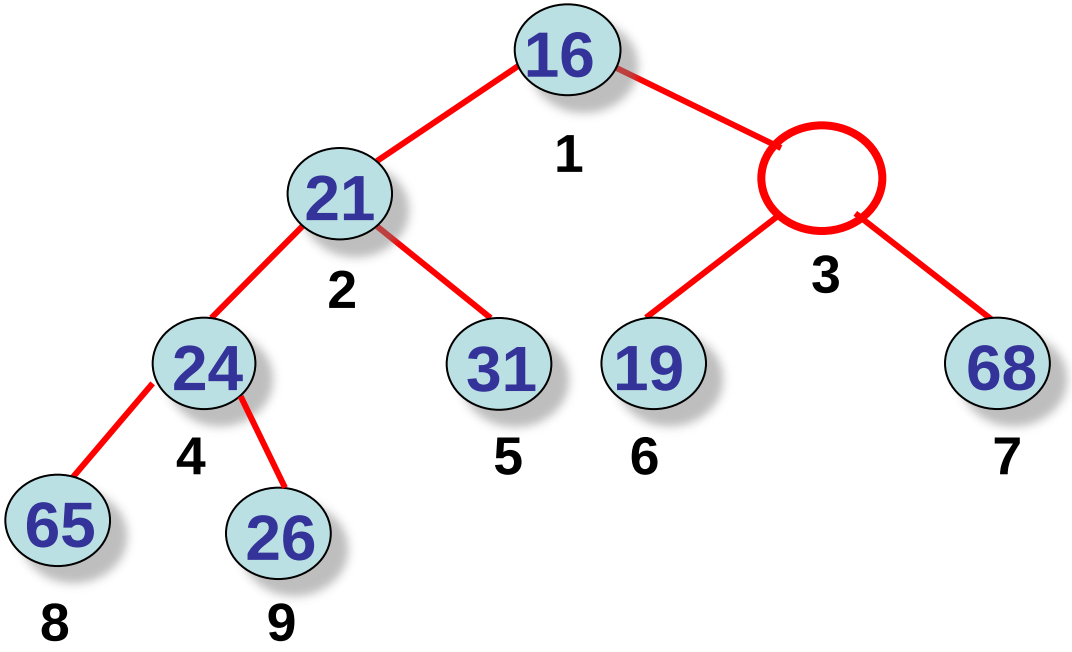
minimum

32

temp

	0	1	2	3	4	5	6	7	8	9	10	11	12
heap			21	16	24	31	19	68	65	26			

ExtractMin Step 3: Bubble hole down until temp ≤ its children and ≥ its parent



13

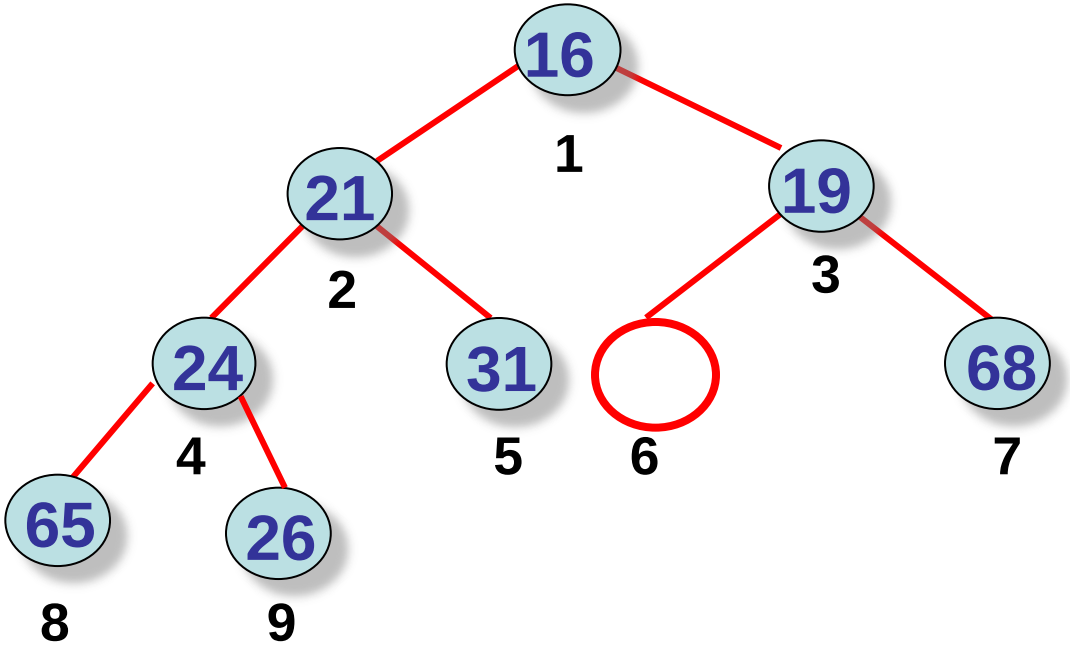
minimum

32

temp

	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		16	21		24	31	19	68	65	26			

ExtractMin Step 3: Bubble hole down until temp ≤ its children and ≥ its parent



13

minimum

32

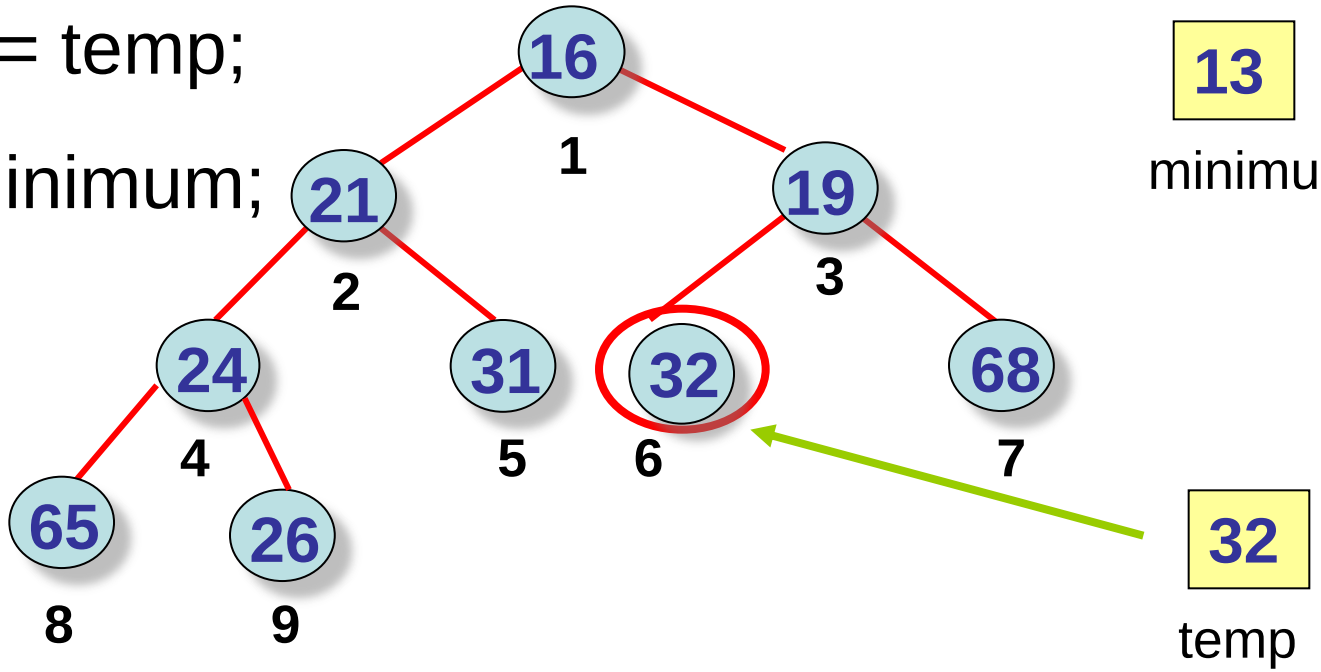
temp

	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		16	21	19	24	31		68	65	26			

ExtractMin Step 4: Copy temp to the hole and return the minimum

heap[6] = temp;

return minimum;



	0	1	2	3	4	5	6	7	8	9	10	11	12
heap		16	21	19	24	31	32	68	65	26			

Analysis of ExtractMin?

Worst-case complexity?

Insert n Elements into an Initially Empty Heap

Worst-case complexity?

Bottom-Up Heap Construction: BuildHeap

If all the keys are given in advance and stored in an array, we can build a heap in worst-case O(n) time.

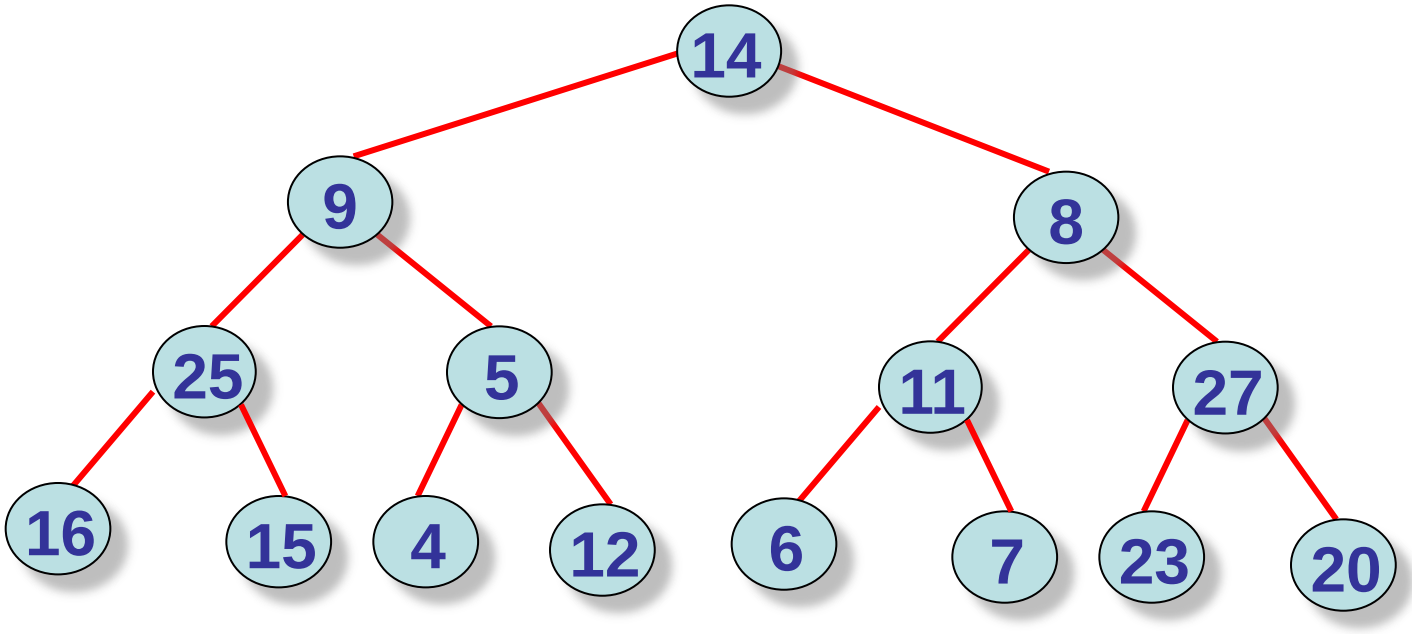
Note: For simplicity, we'll describe bottom-up heap construction for a full tree

Algorithm: Call percolateDown() on nodes in non-heap tree in reverse level-order traversal to convert tree to a heap.

BuildHeap Algorithm

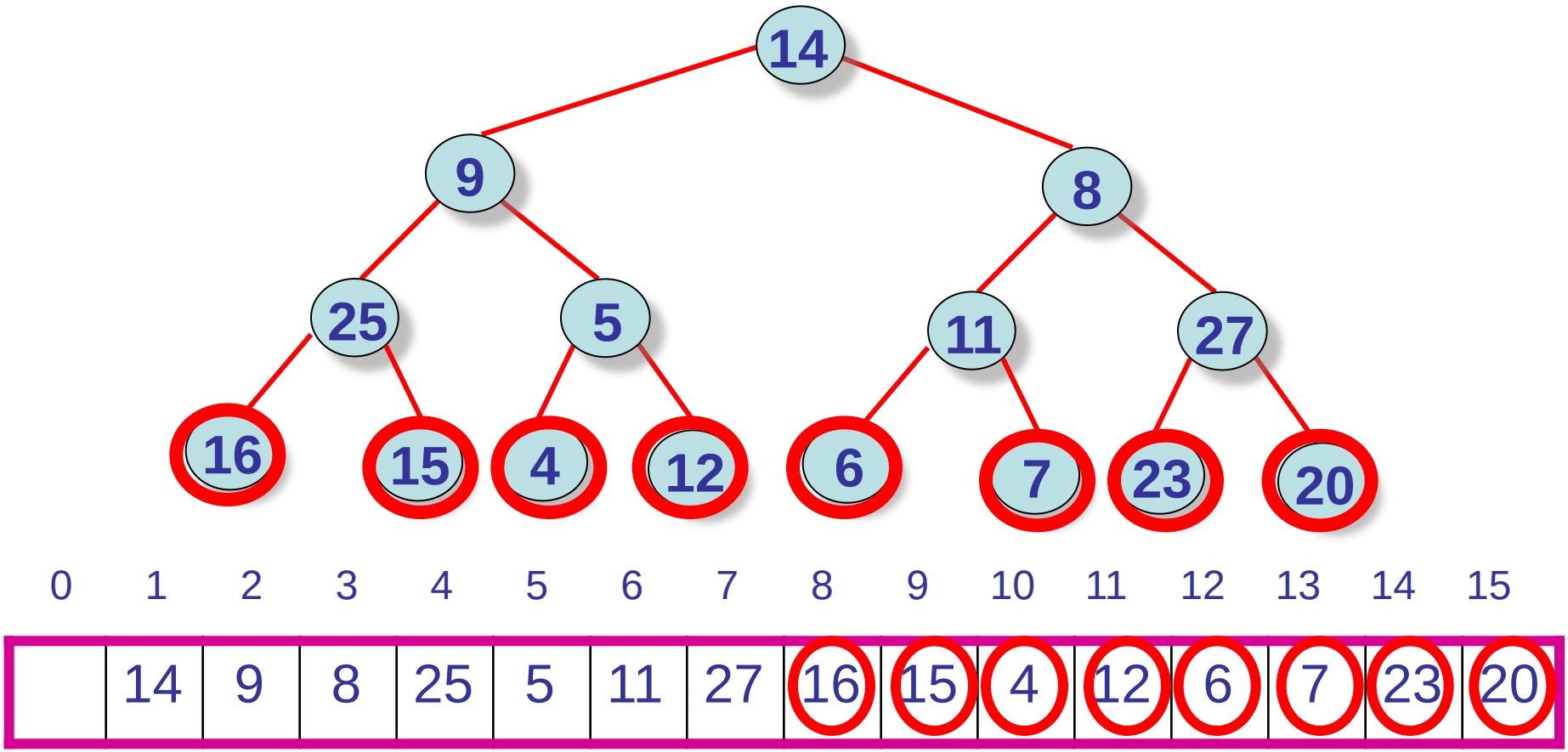
1. Build (n+1)/2 elementary heaps, with one key each, of the leaves of the tree (no work)
2. Build (n+1)/4 heaps, each with 3 keys, by joining pairs of elementary heaps with their parent as the root. Call percolateDown()
3. Build (n+1)/8 heaps, each with 7 keys, by joining pairs of 3-key heaps with their parent as the root. Call percolateDown()
4. Continue until reach root of entire tree.

BuildHeap at Start: Elements are Unordered: Not in Heap Order

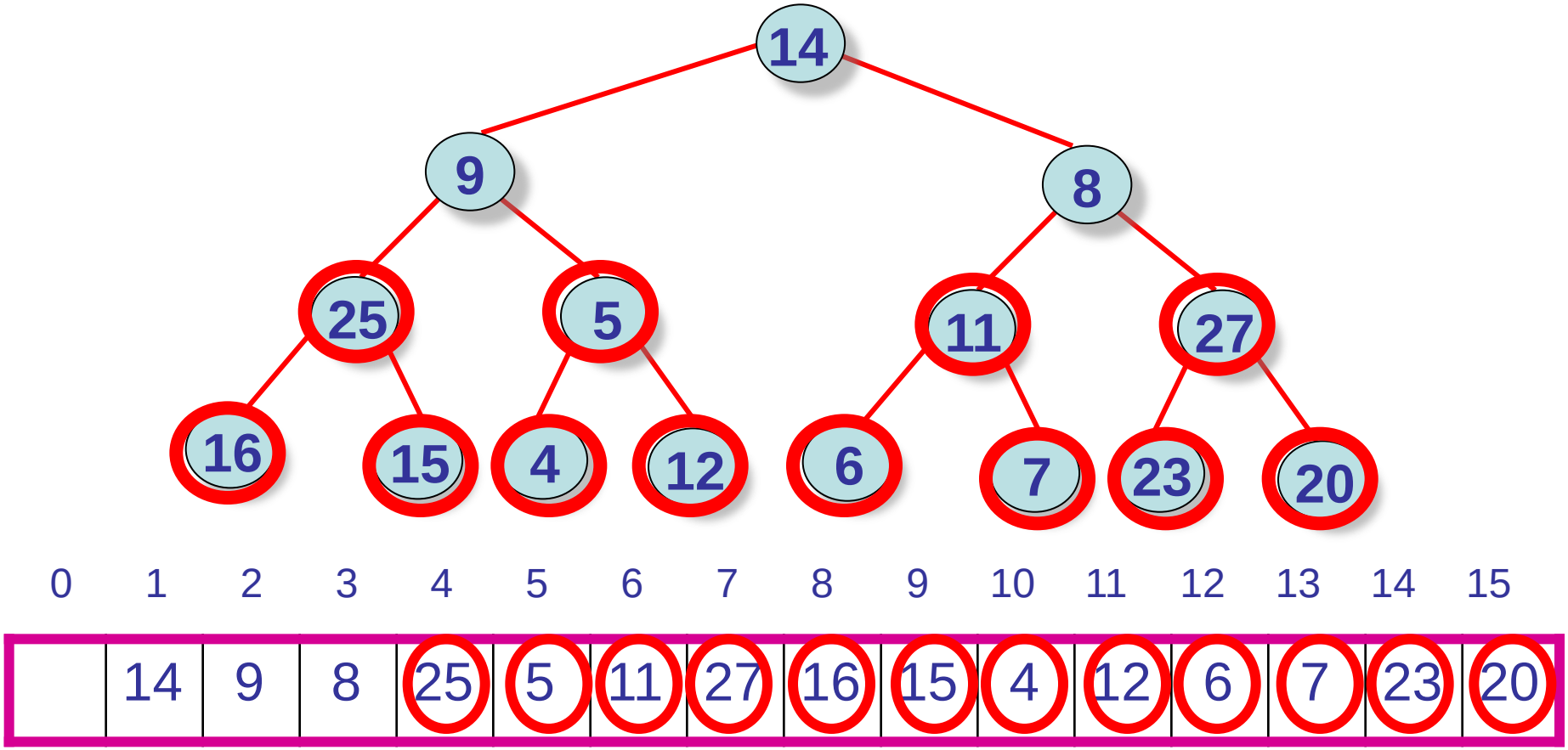


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	9	8	25	5	11	27	16	15	4	12	6	7	23	20

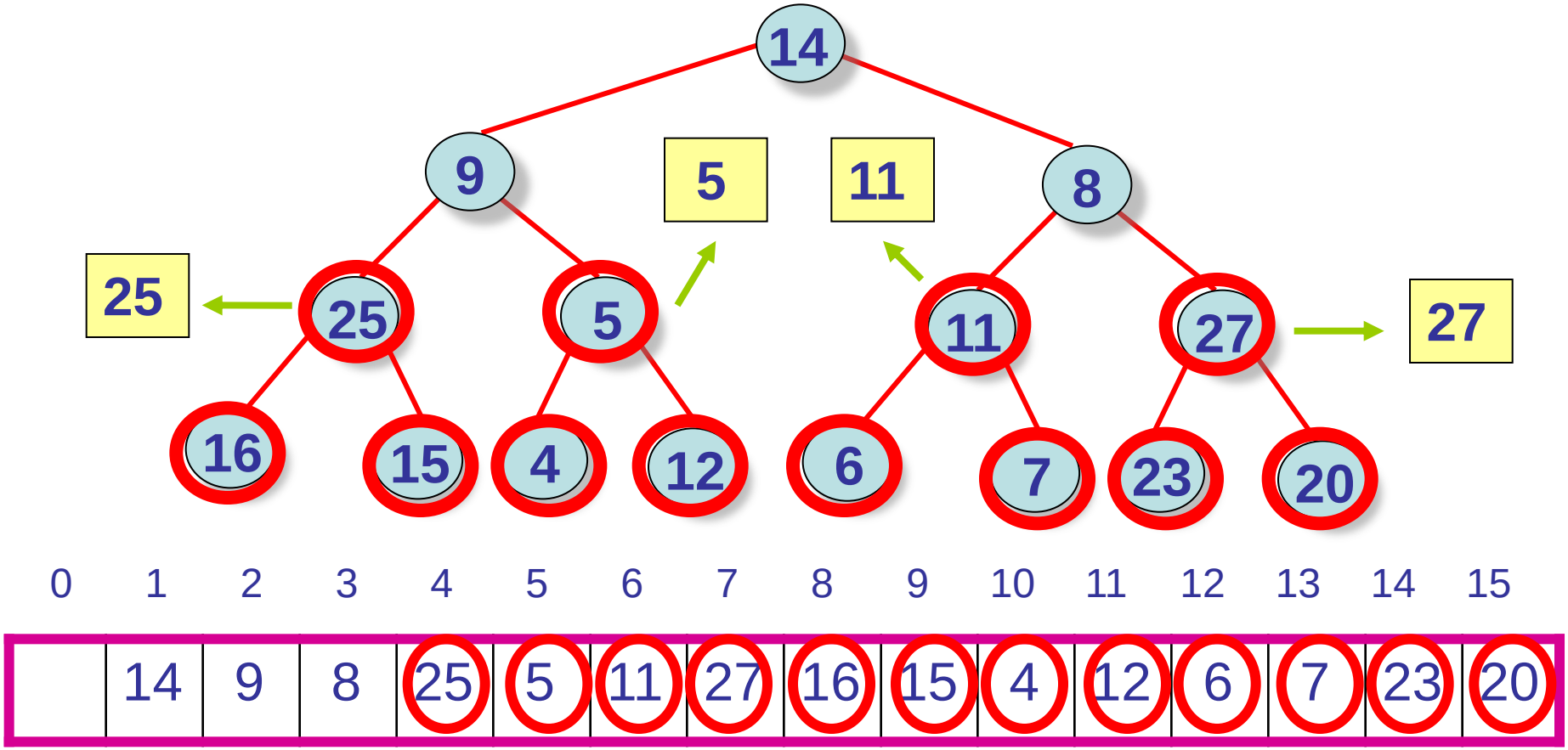
BuildHeap Step 1: Build (n+1)/2 elementary heaps, with one key each



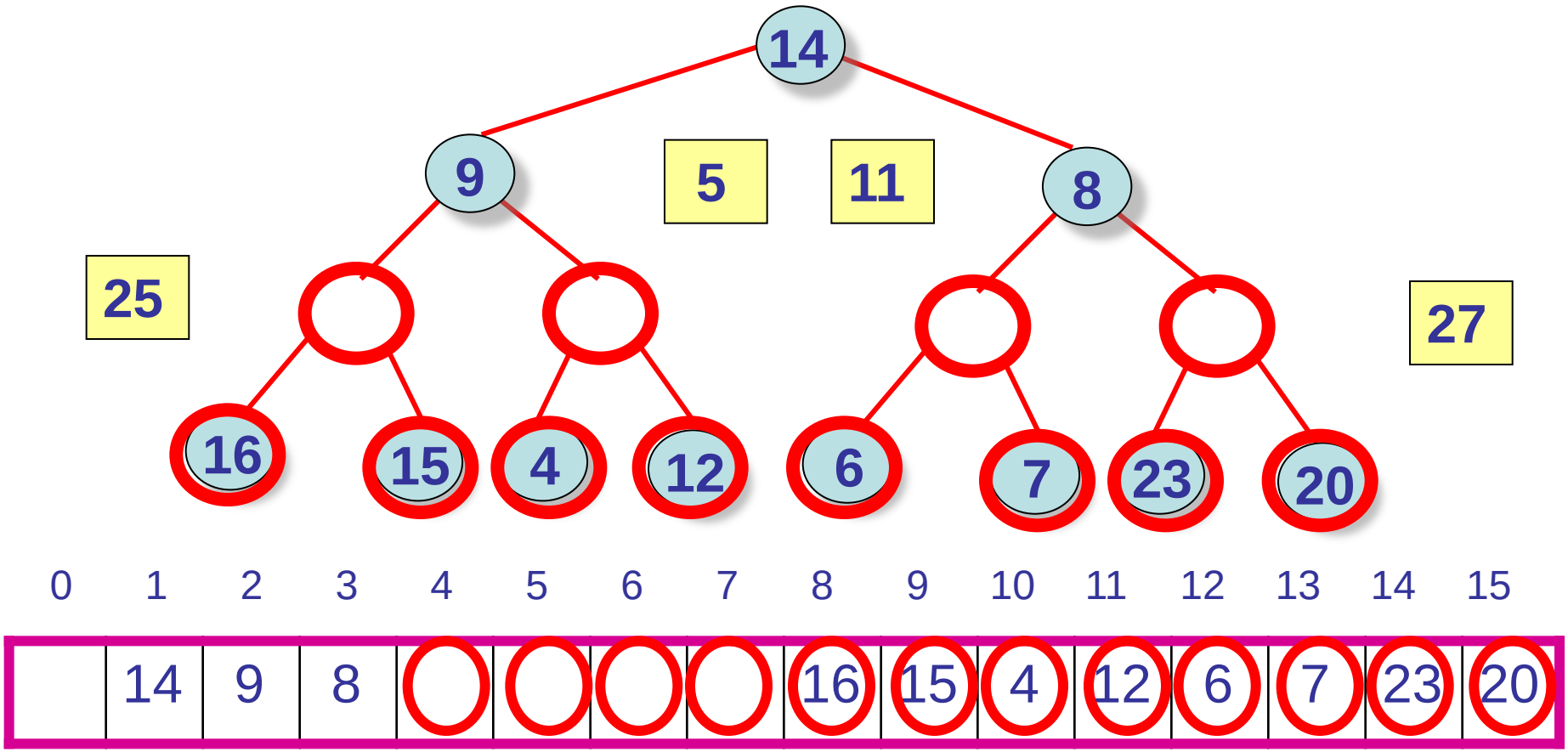
BuildHeap Step 2:
Build (n+1)/4 heaps, with 3 keys each



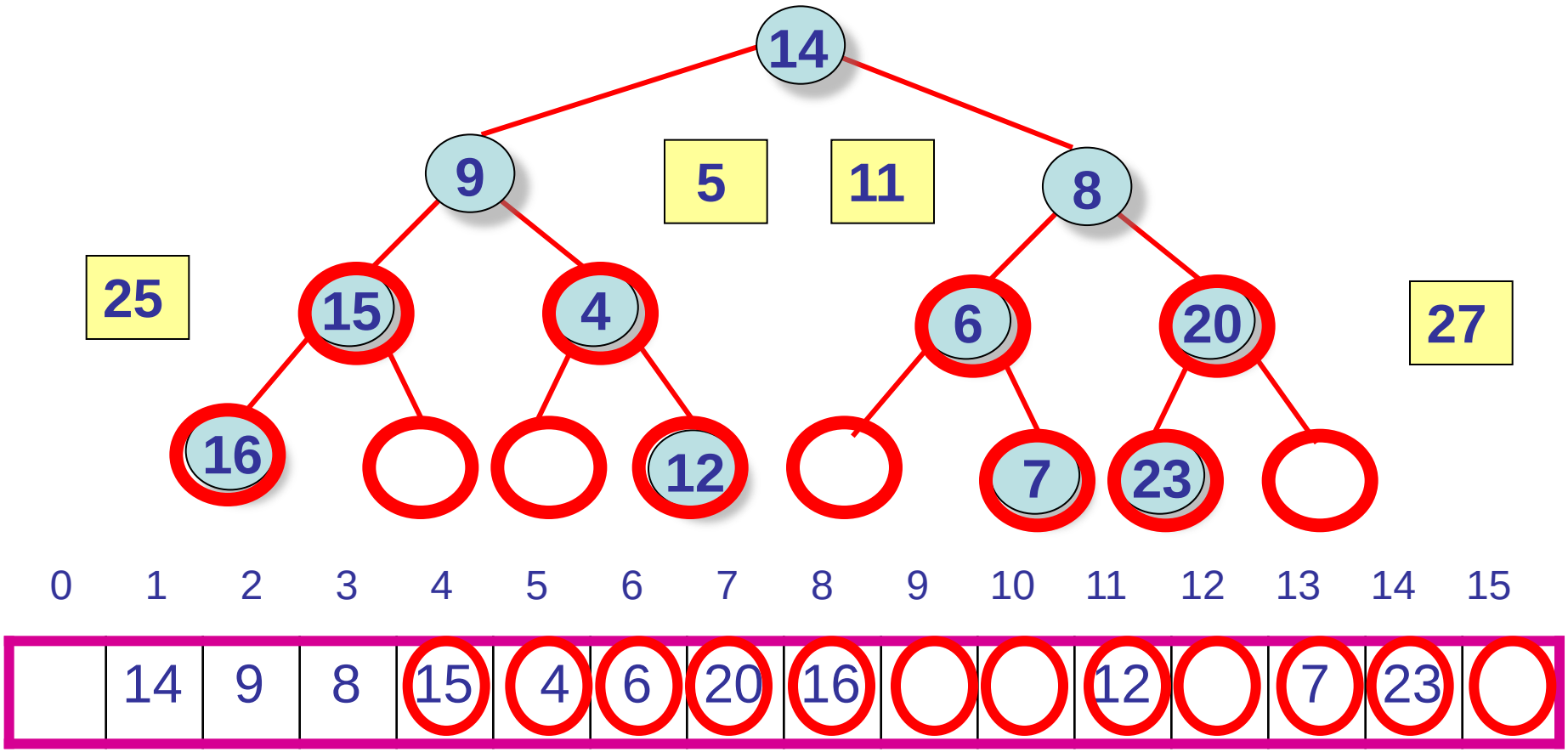
BuildHeap Step 2:
Call percolateDown( ) for each 3-key heap



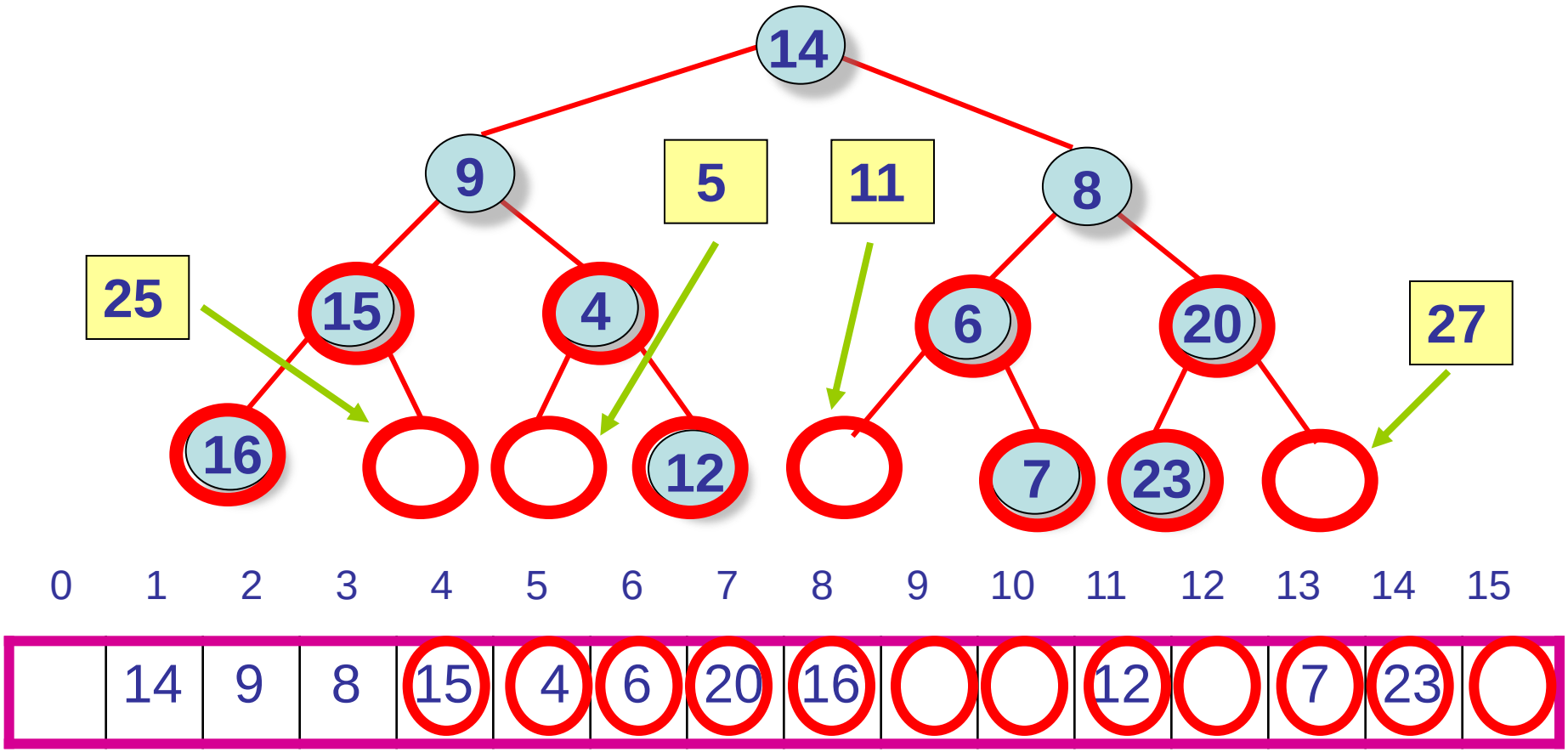
BuildHeap Step 2:
Call percolateDown( ) for each 3-key heap



BuildHeap Step 2:
Call percolateDown( ) for each 3-key heap

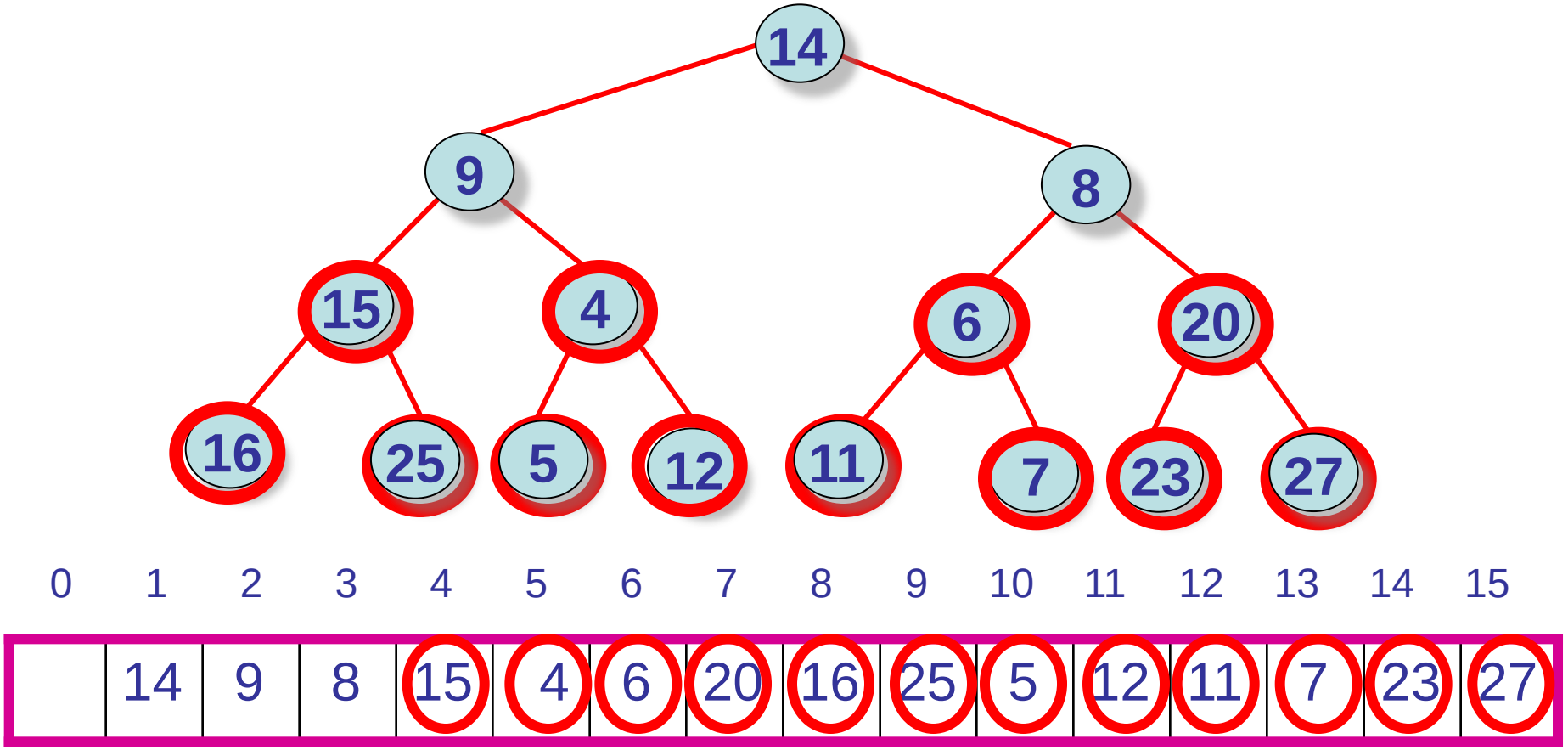


BuildHeap Step 2:
Call percolateDown( ) for each 3-key heap

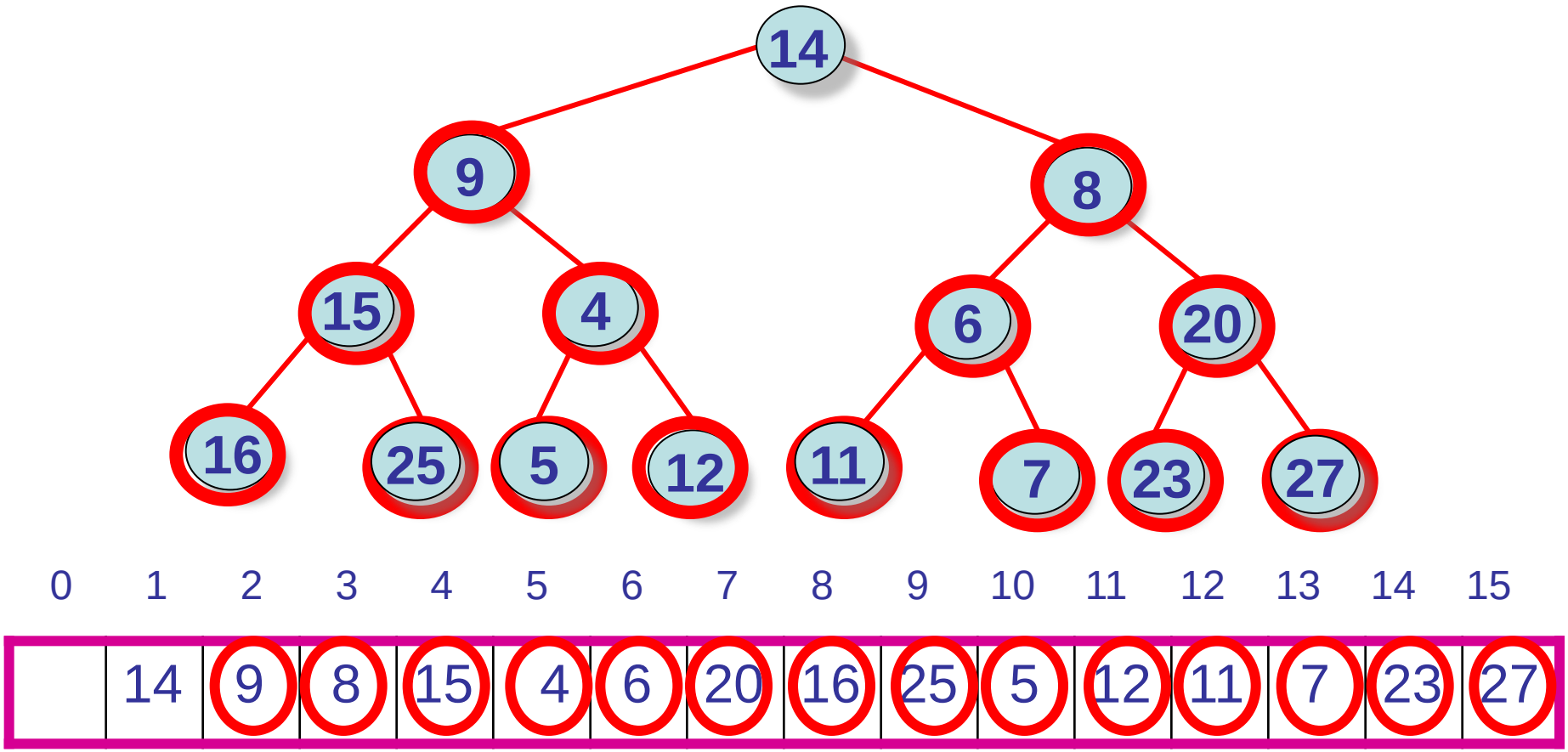


BuildHeap Step 2:

Now we have (n+1)/4 heaps, with 3 keys each

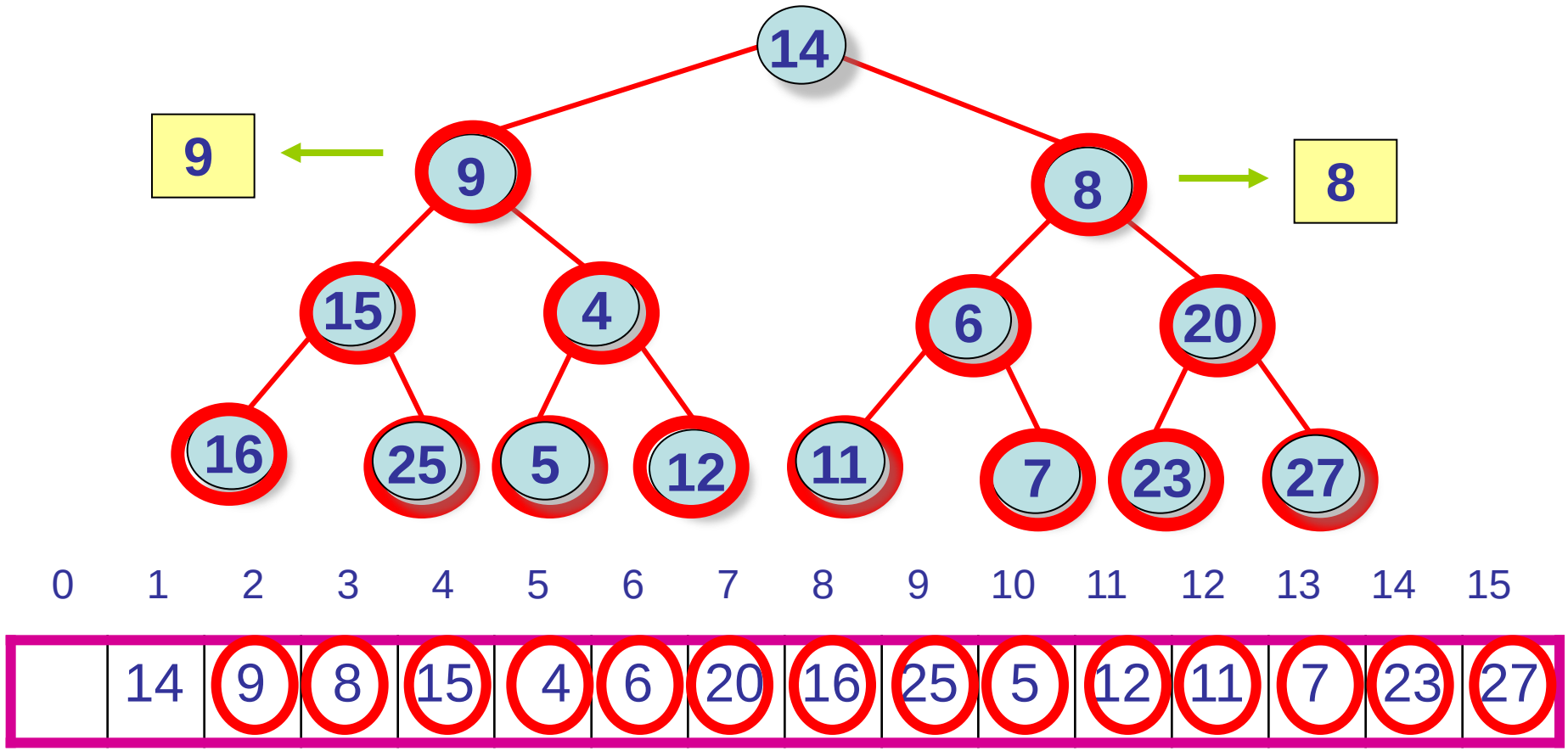


BuildHeap Step 3:
Build (n+1)/8 heaps, with 7 keys each

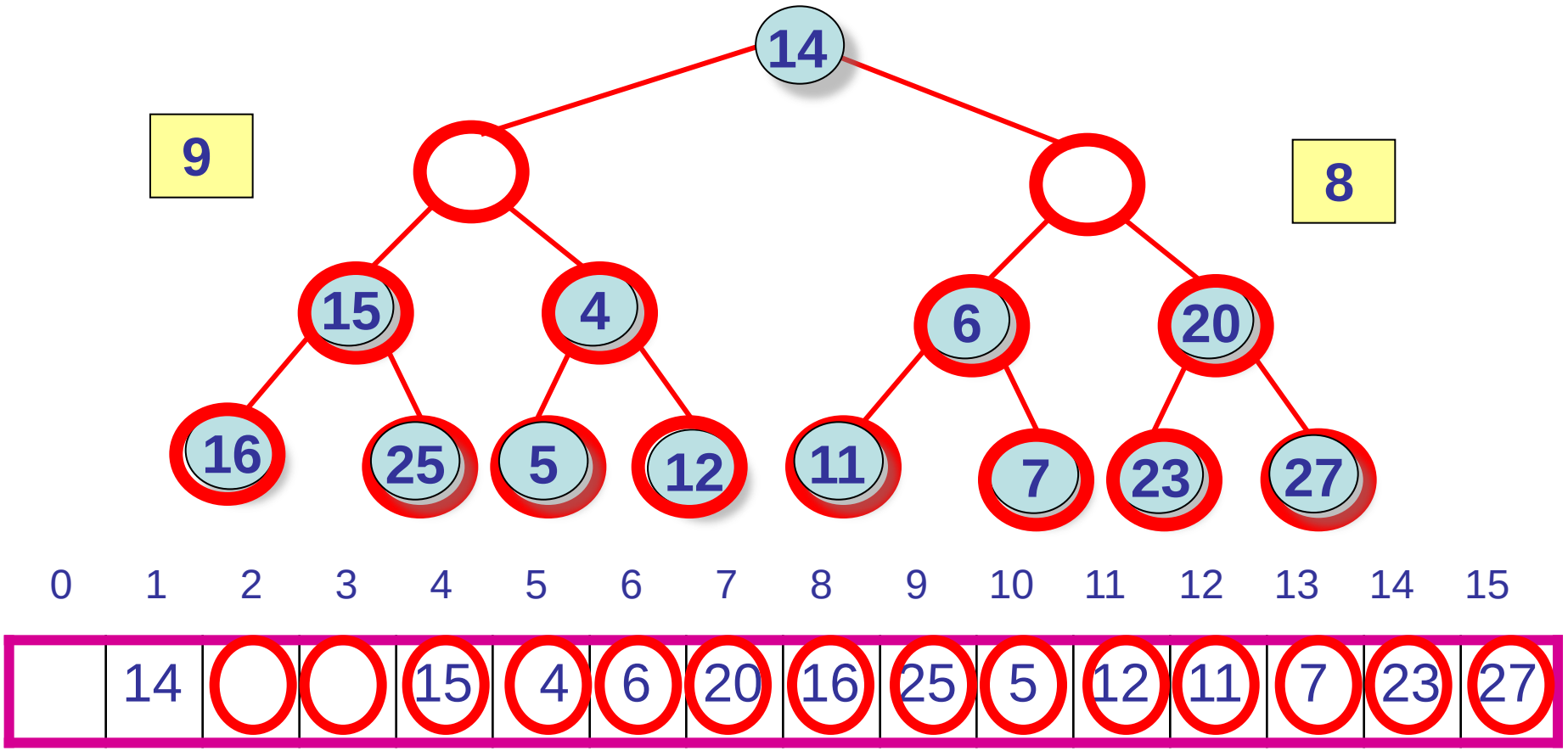


BuildHeap Step 3:

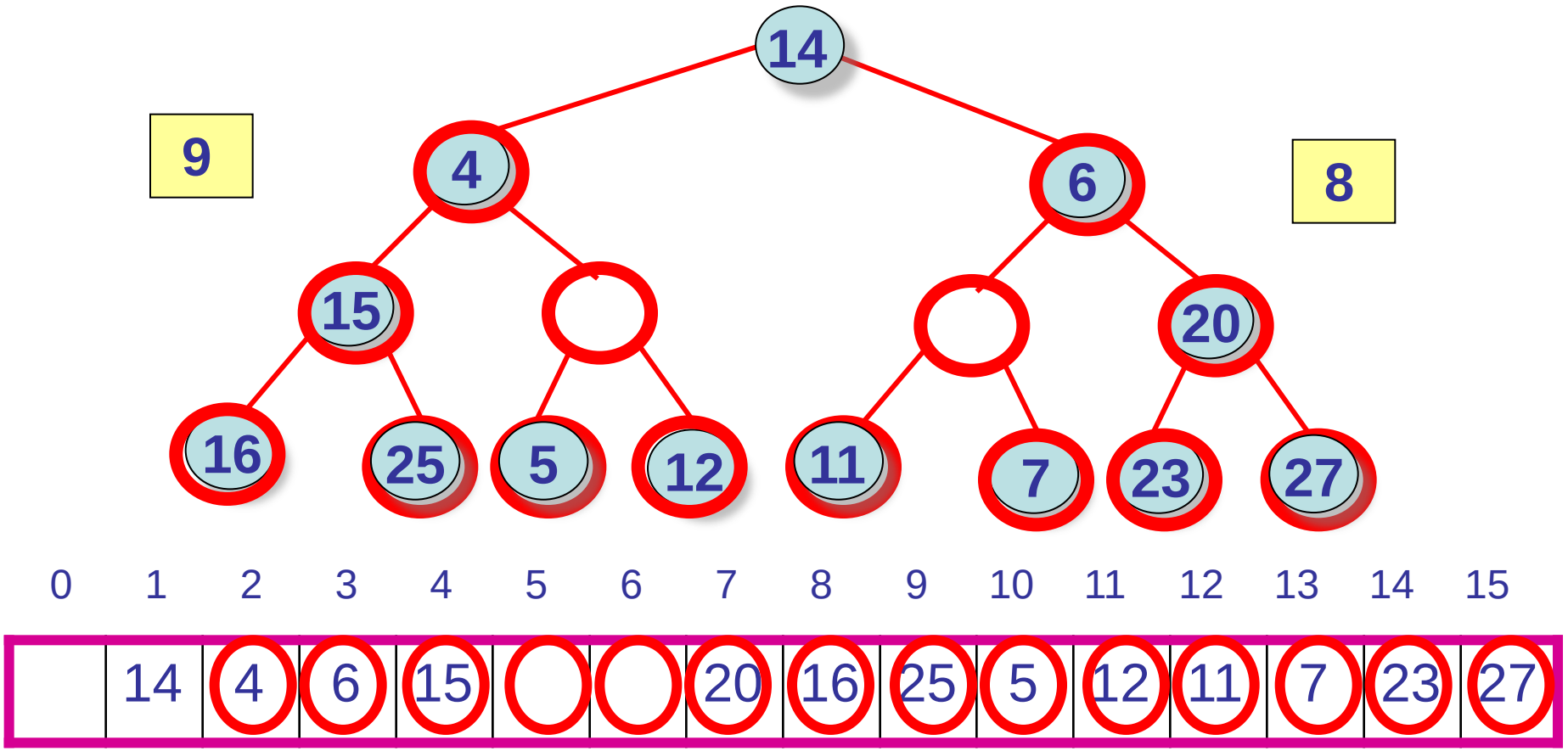
Call percolateDown( ) for each 7-key heap



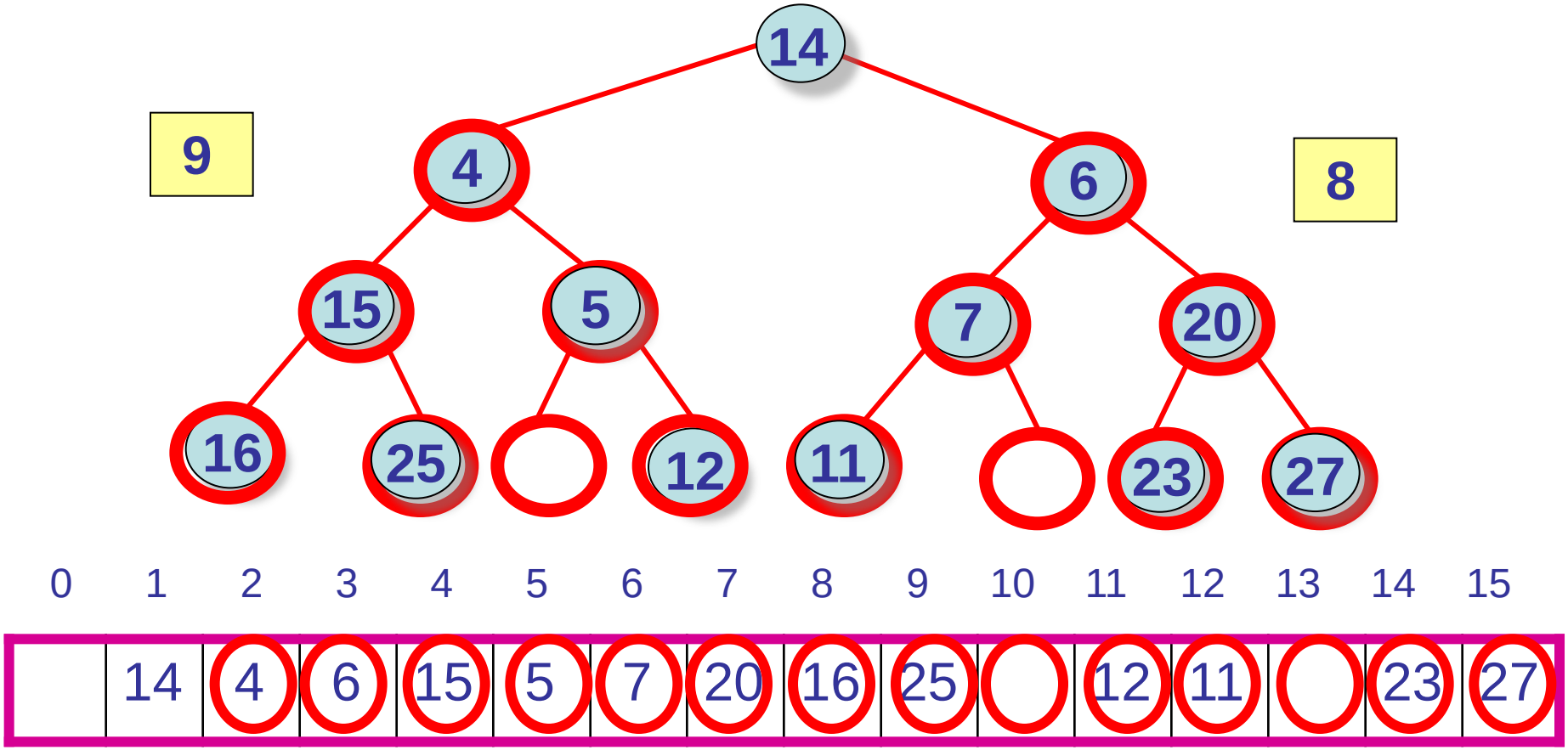
BuildHeap Step 3:
Call percolateDown( ) for each 7-key heap



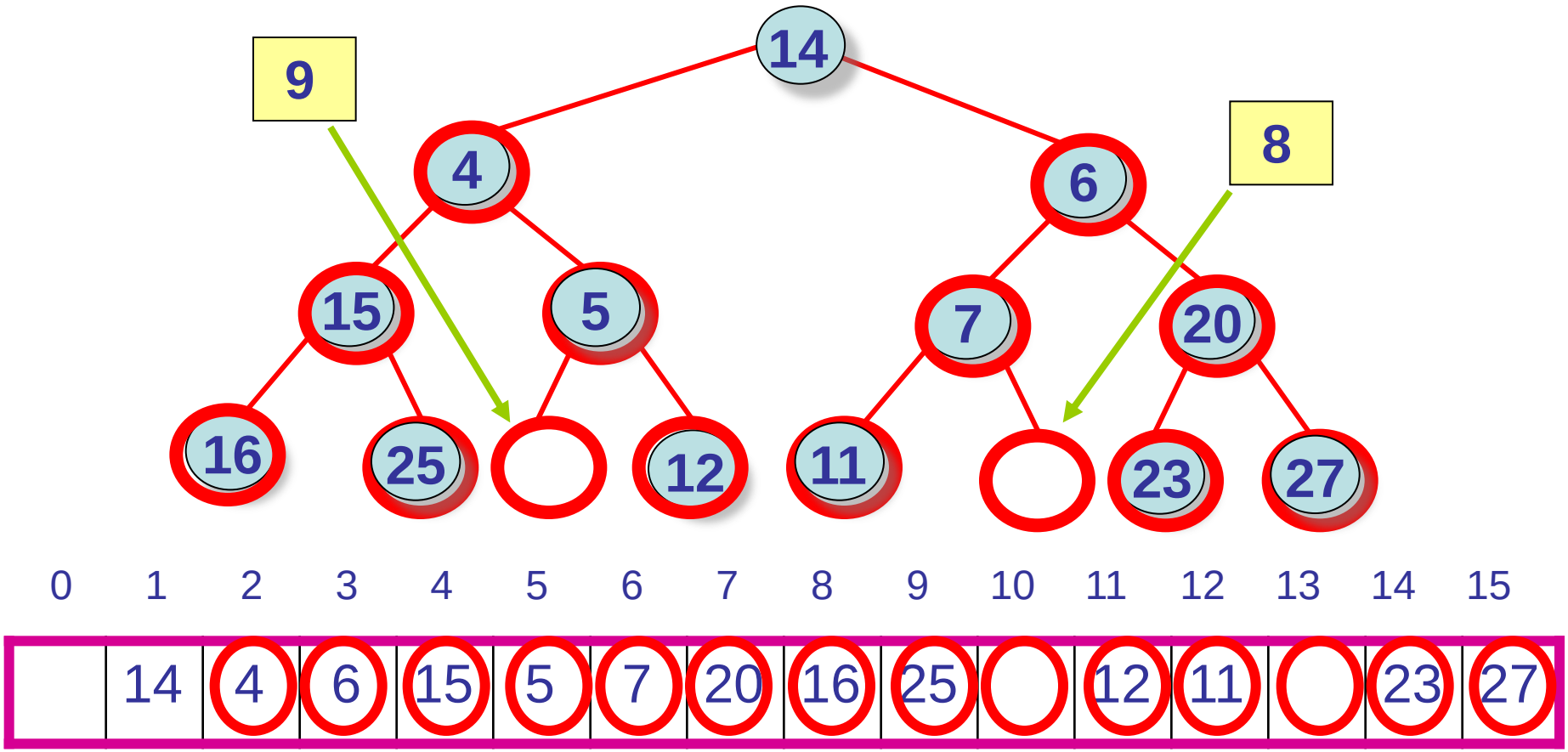
BuildHeap Step 3:
Call percolateDown( ) for each 7-key heap



BuildHeap Step 3:
Call percolateDown( ) for each 7-key heap

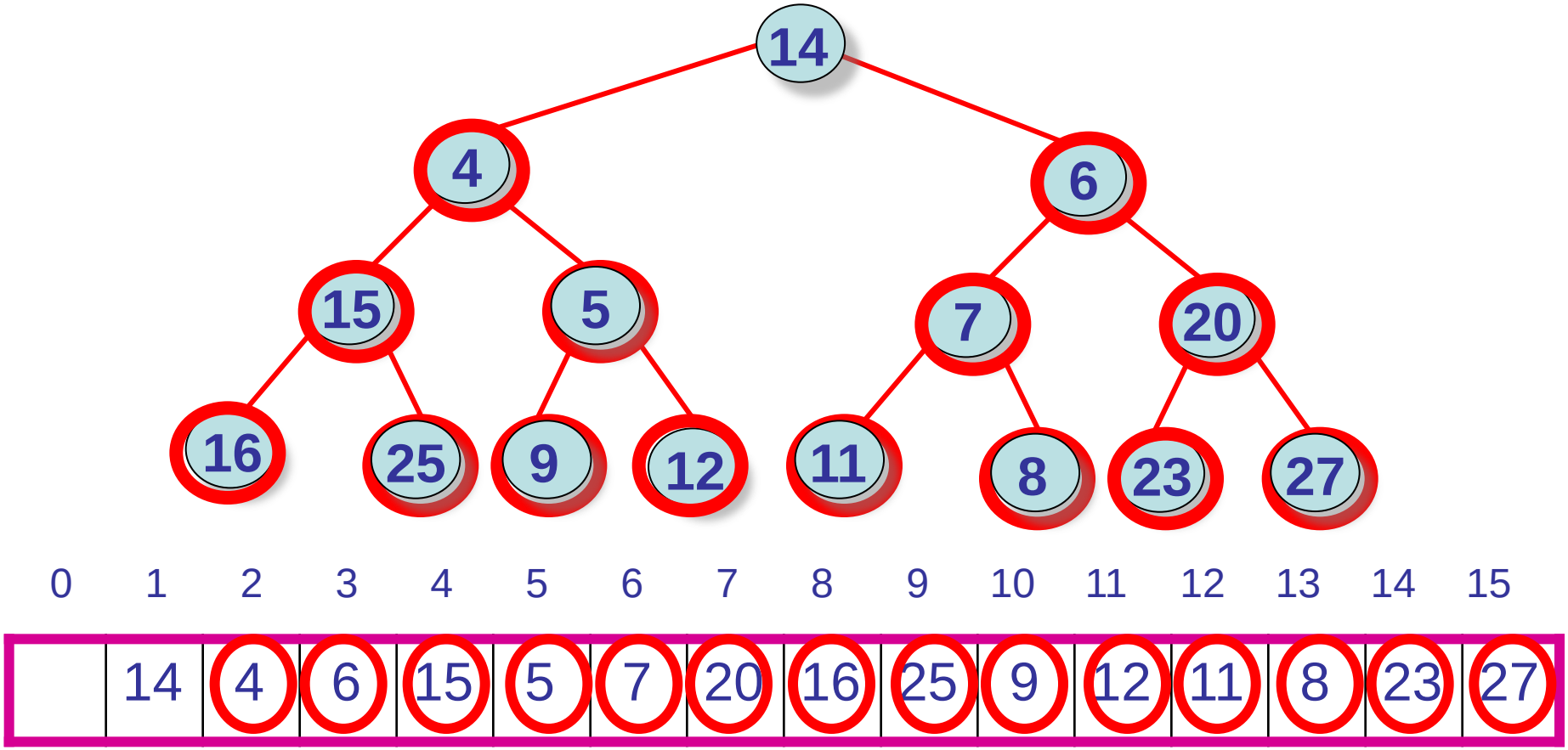


BuildHeap Step 3:
Call percolateDown( ) for each 7-key heap

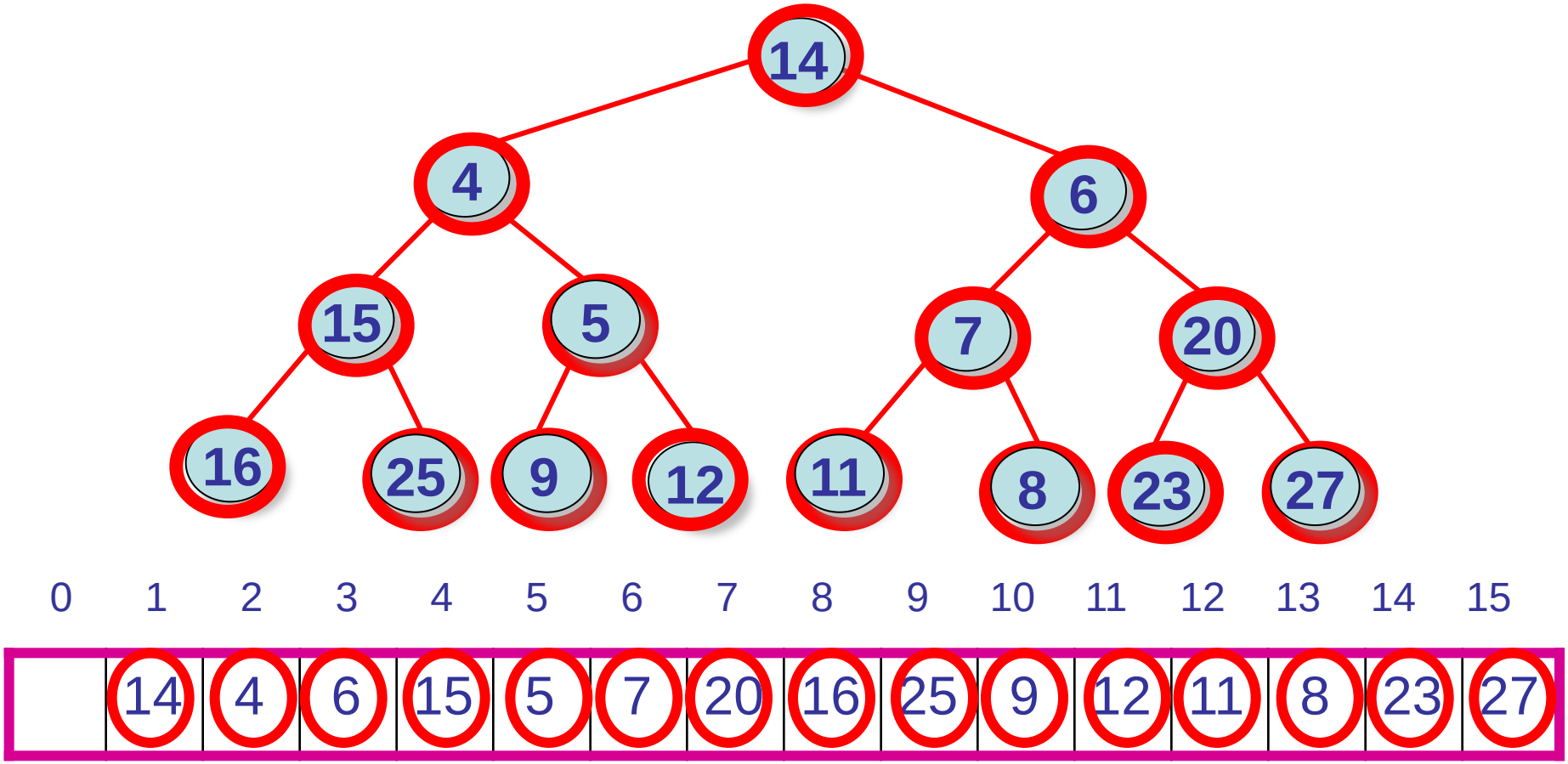


BuildHeap Step 3:

Now we have (n+1)/8 heaps, with 7 keys each

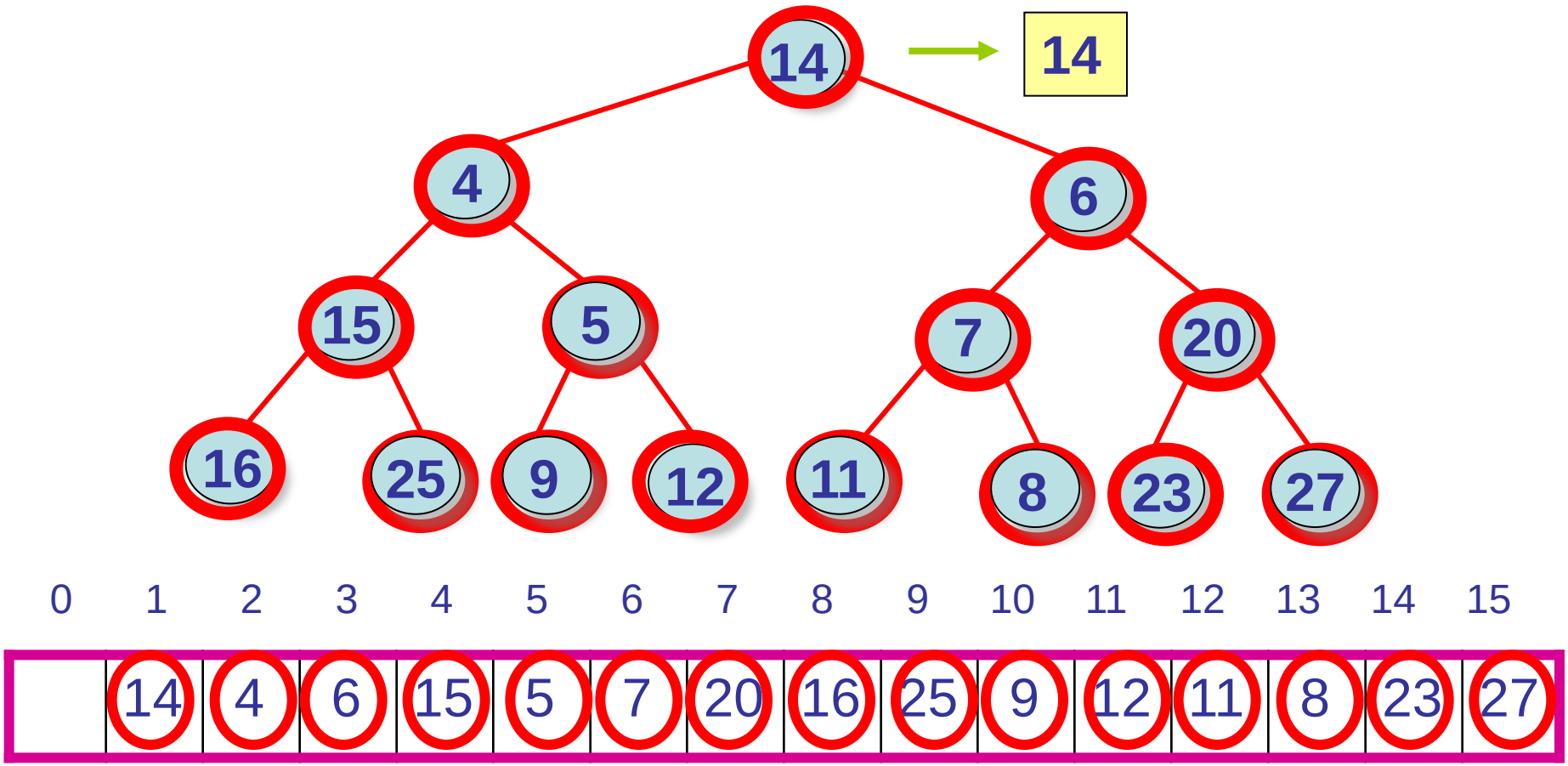


BuildHeap Step 4: Build (n+1)/16 heaps, with 15 keys each (only one here)



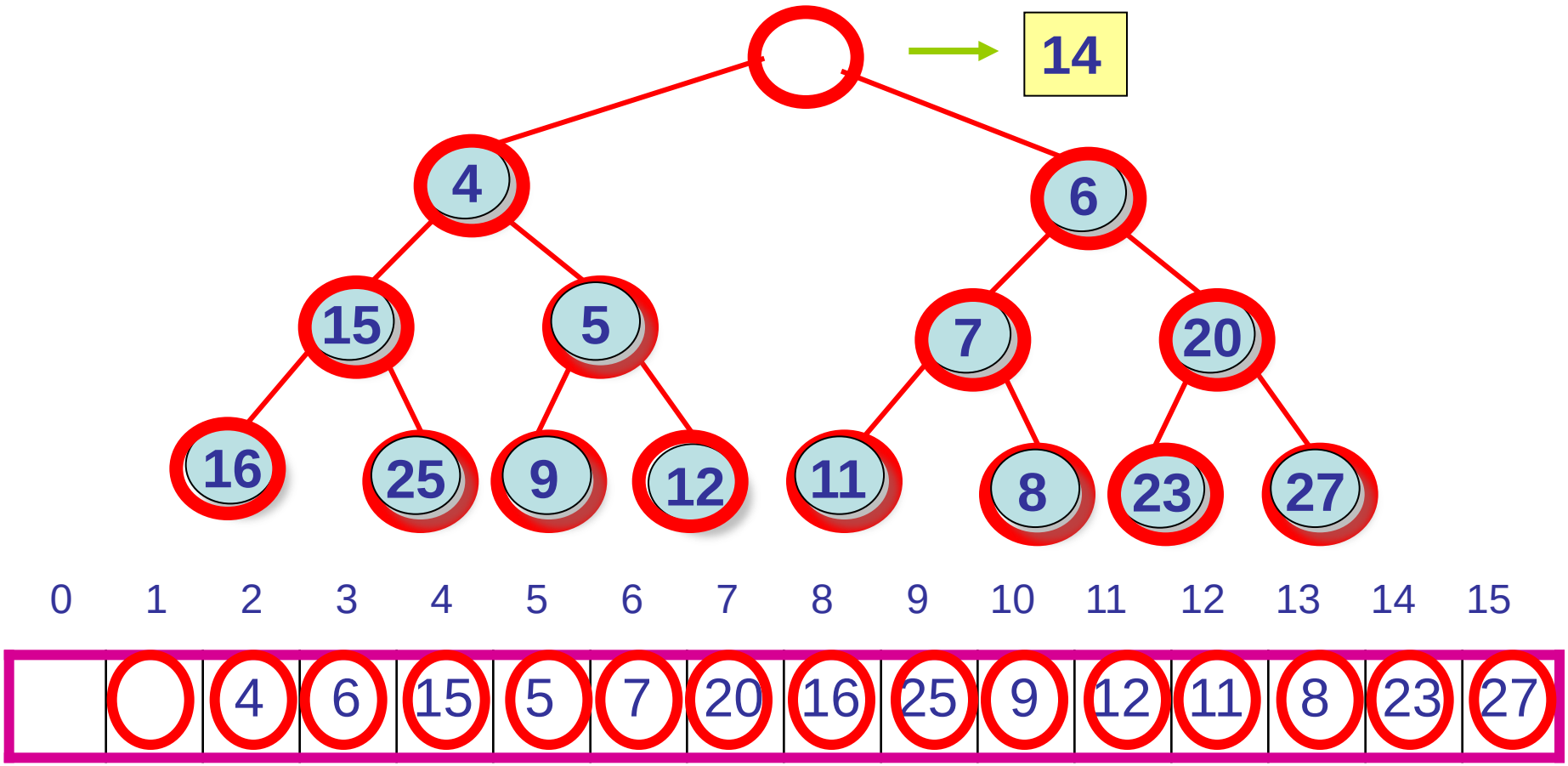
BuildHeap Step 4:

Call percolateDown( ) for each 15-key heap



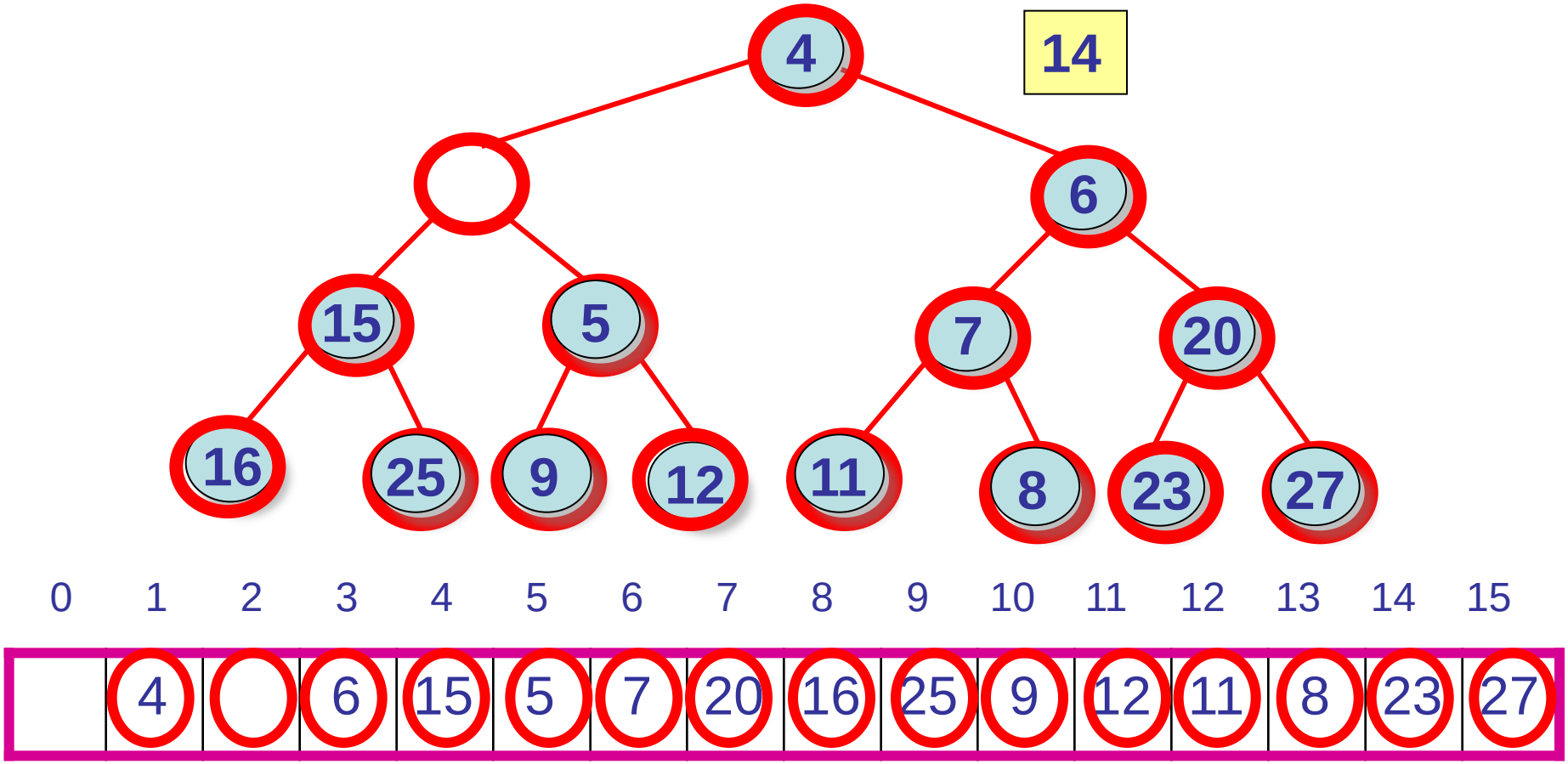
BuildHeap Step 4:

Call percolateDown( ) for each 15-key heap



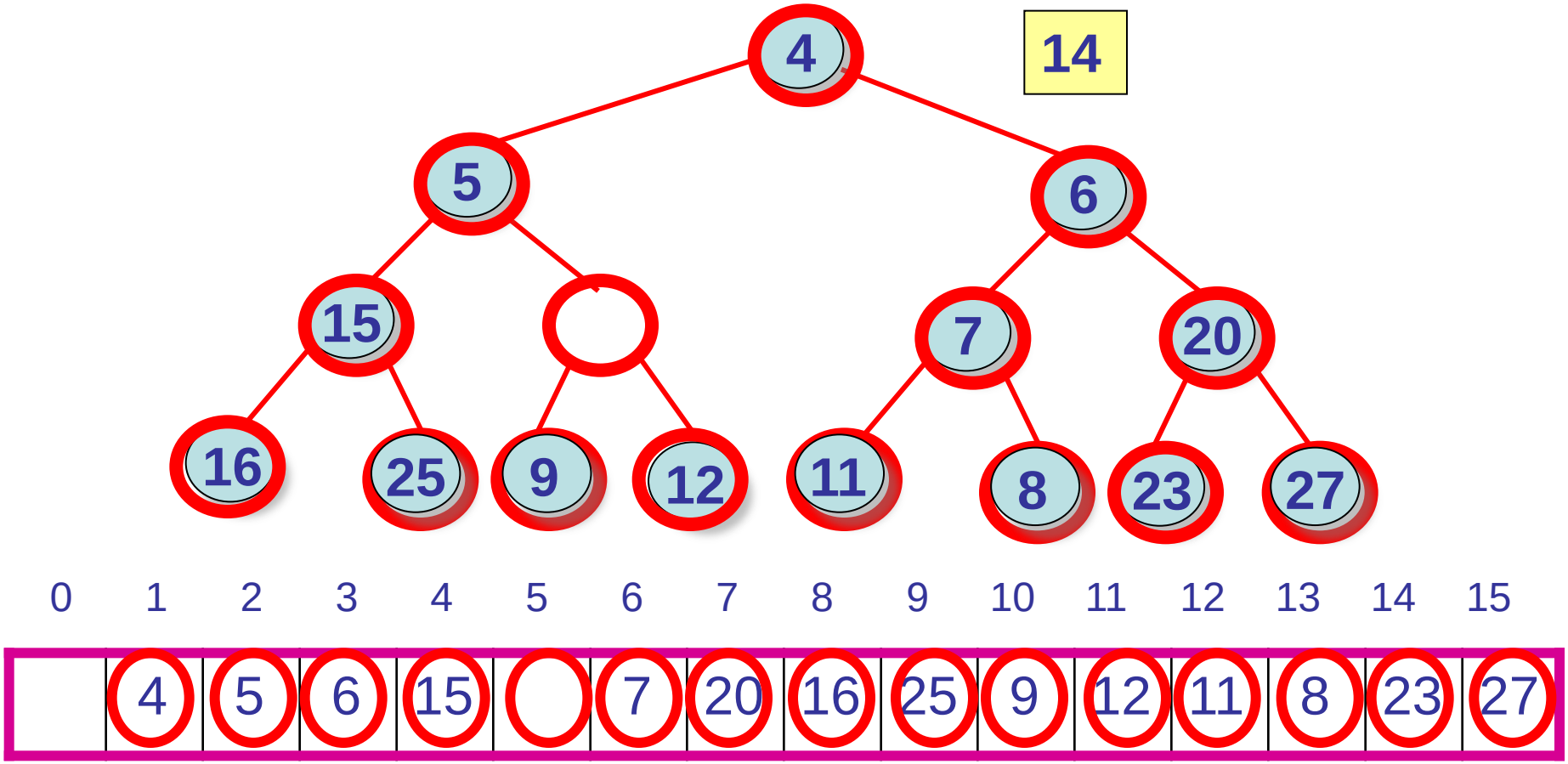
BuildHeap Step 4:

Call percolateDown( ) for each 15-key heap



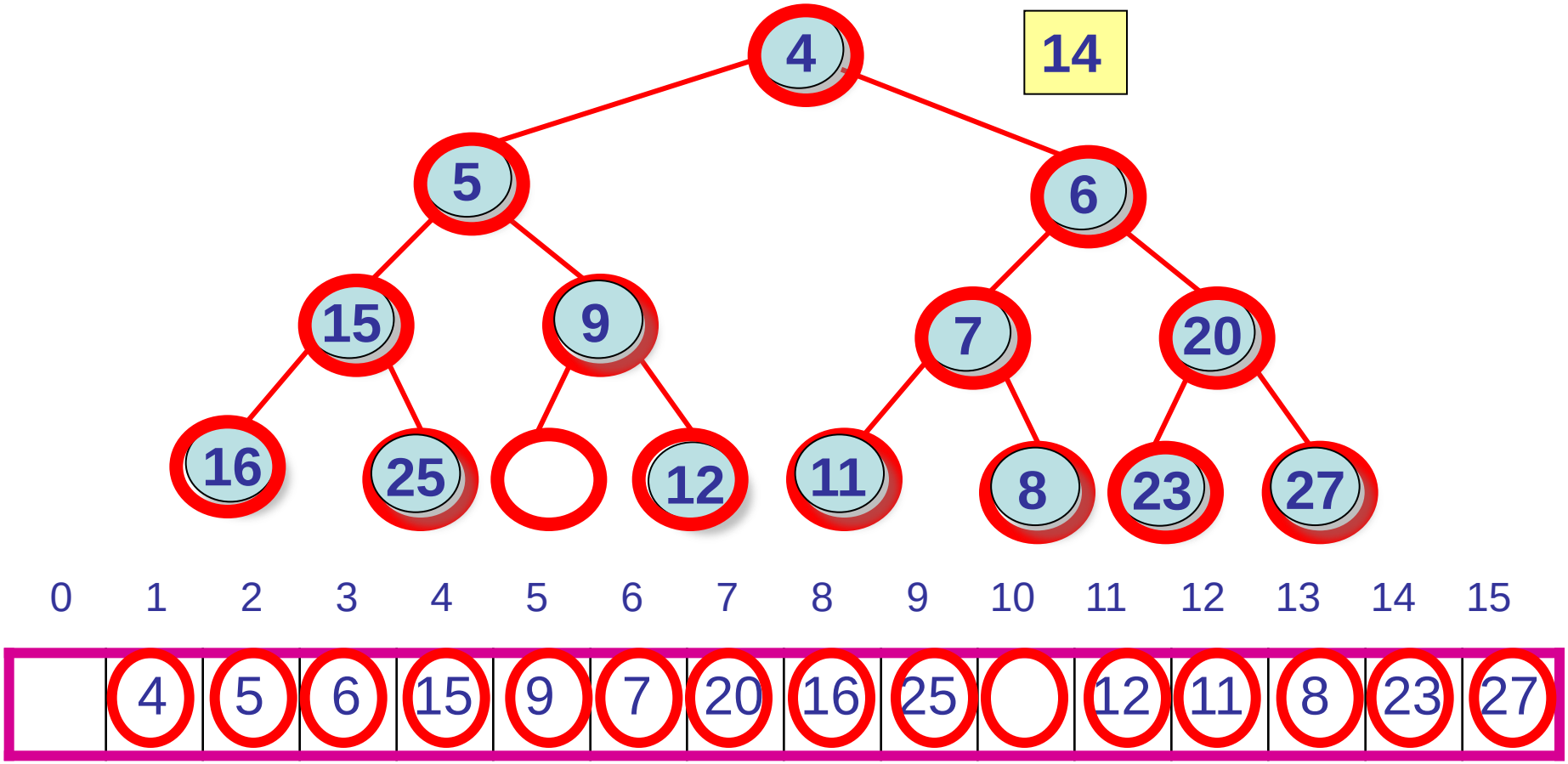
BuildHeap Step 4:

Call percolateDown( ) for each 15-key heap



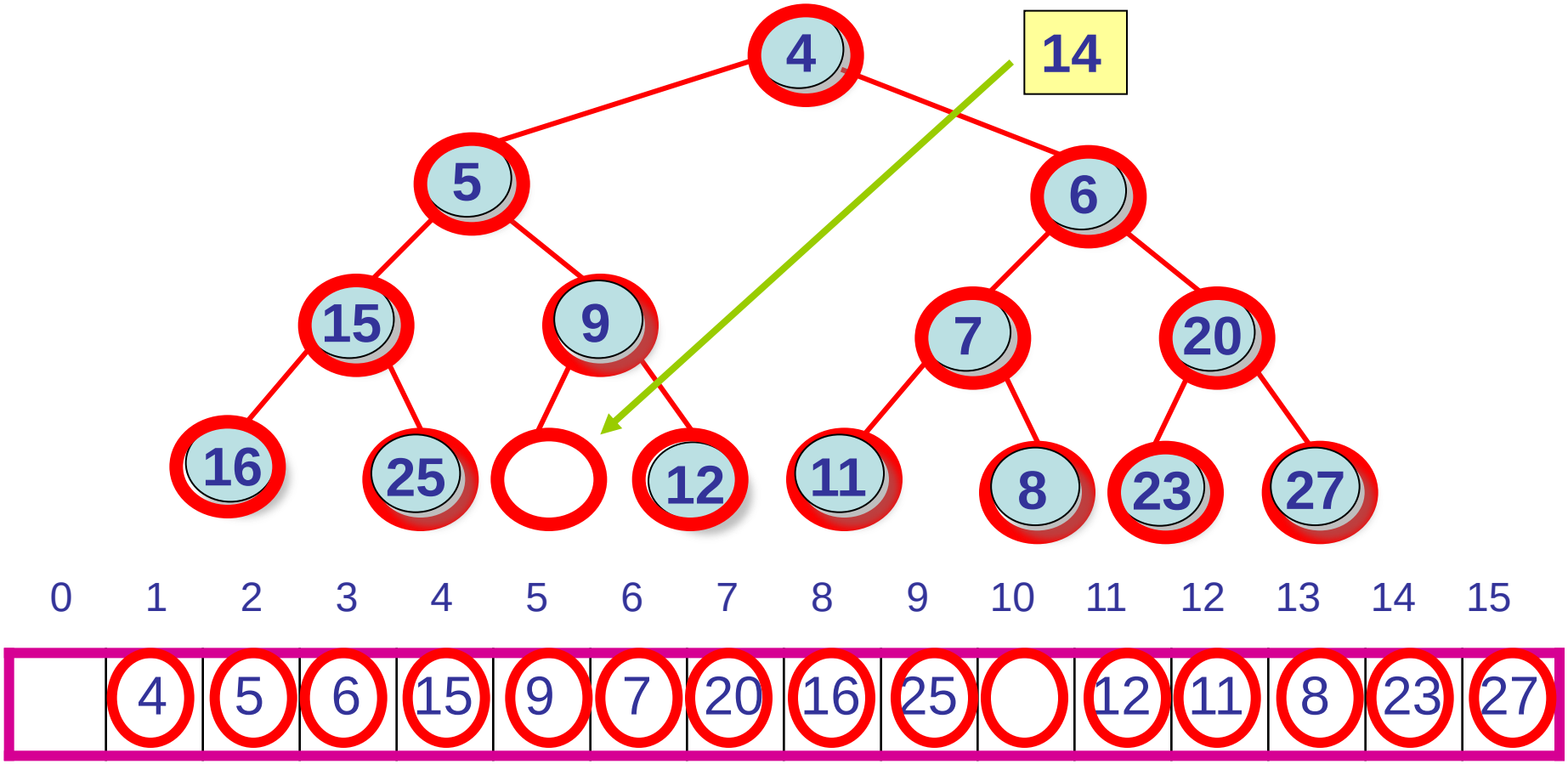
BuildHeap Step 4:

Call percolateDown( ) for each 15-key heap



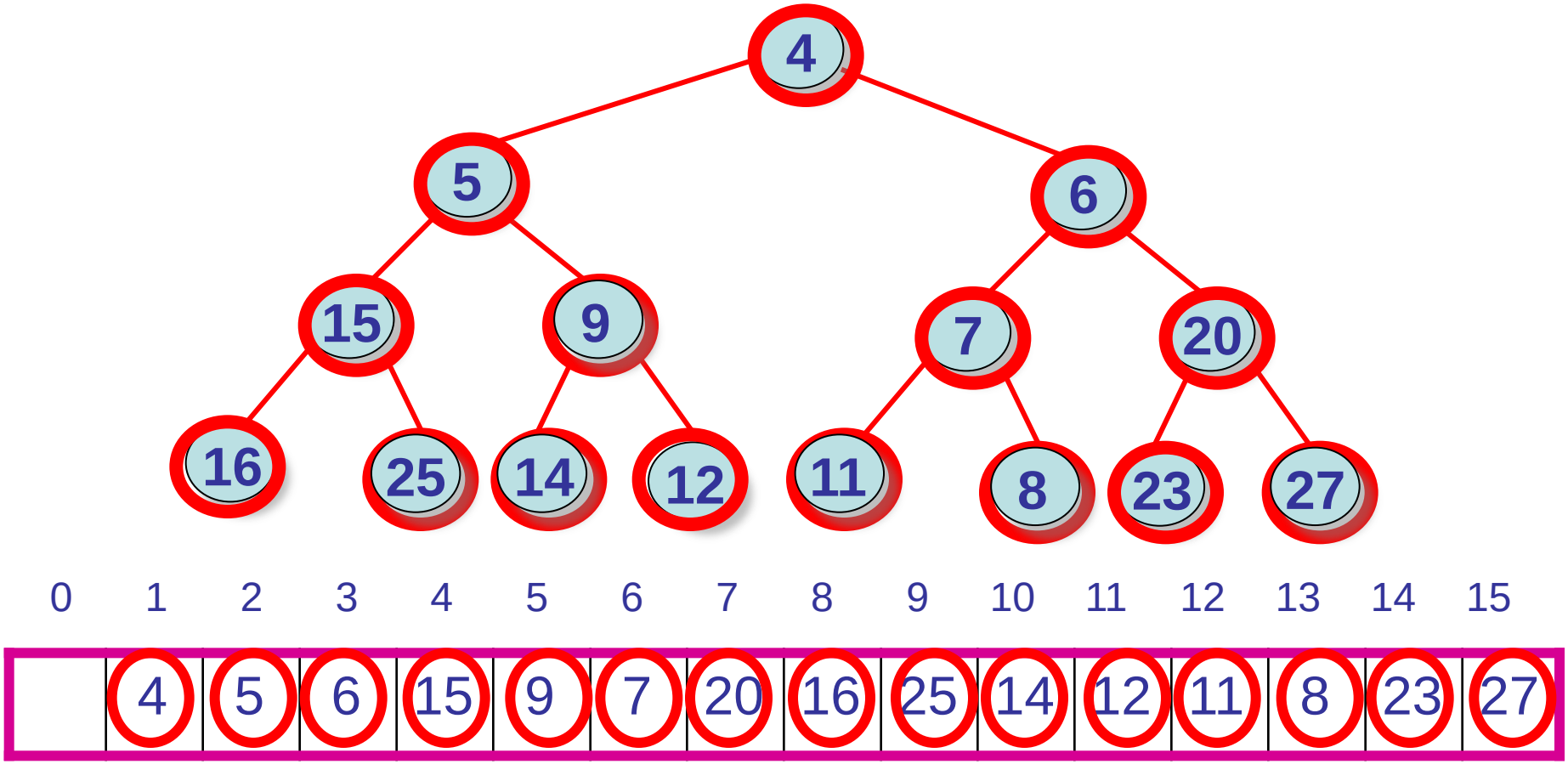
BuildHeap Step 4:

Call percolateDown( ) for each 15-key heap



BuildHeap Step 4:

The Entire Tree is Now a Heap!



Heapsort

How could we use a heap to sort an array?