Fall 2019

Priority Queues

CMPE 250 - Data Structures & Algorithms

Presenter: Meriç Turan

17 OCTOBER 2019

Why do we need priority queues?

- Although jobs sent to a printer are generally placed on a queue, this might not always be the best thing to do.
- One job might be particularly important, so it might be desirable to allow that job to be run as soon as the printer is available. Conversely, if, when the printer becomes available, there are several 1-page jobs and one 100-page job, it might be reasonable to make the long job go last, even if it is not the last job submitted.
- Similarly, in a multiuser environment, the **operating system scheduler** must decide which of several processes to run.

Model

- A priority queue is a data structure that allows at least the following two operations: **insert**, which does the obvious thing; and **deleteMin**, which finds, returns, and removes the minimum element in the priority queue.

- The insert operation is the equivalent of enqueue, and deleteMin is the priority queue equivalent of the queue's dequeue operation.

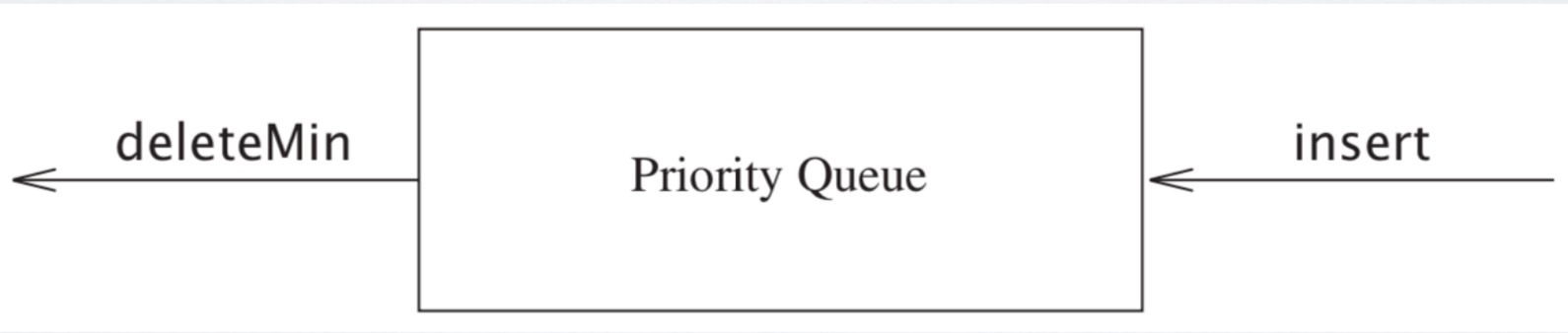

Figure: Basic model of priority queue.

Alternative Implementations

- There are several obvious ways to implement a priority queue.

Linked list implementation

- We could use a simple linked list, performing insertions at the front in O(1) and traversing the list, which requires O(N) time, to delete the minimum.
- Alternatively, we could insist that the list be kept always sorted; this makes insertions expensive (O(N)) and deleteMins cheap (O(1)). The former is probably the better idea of the two, based on the fact that there are never more deleteMins than insertions.

Binary search tree implementation

- Using a search tree could be overkill because it supports a host of operations that are not required.

Binary Heap

- The implementation we will use is known as a **binary heap**.
- Heaps have two properties
  - Structure property
  - Heap-order property

- A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a **complete binary tree**.
- It is easy to show that a complete binary tree of *height h* has between 2^h and $2^{h+1}-1$ nodes. This implies that the height of a complete binary tree is $\lfloor \log N \rfloor$, which is clearly *O(log N)*.

Structure Property

- A complete binary tree is so regular, it can be represented in an array and no links are necessary.

- For any element in array position i, the left child is in position 2i, the right child is in the cell after the left child (2i + 1), and the parent is in position $\lfloor i/2 \rfloor$.

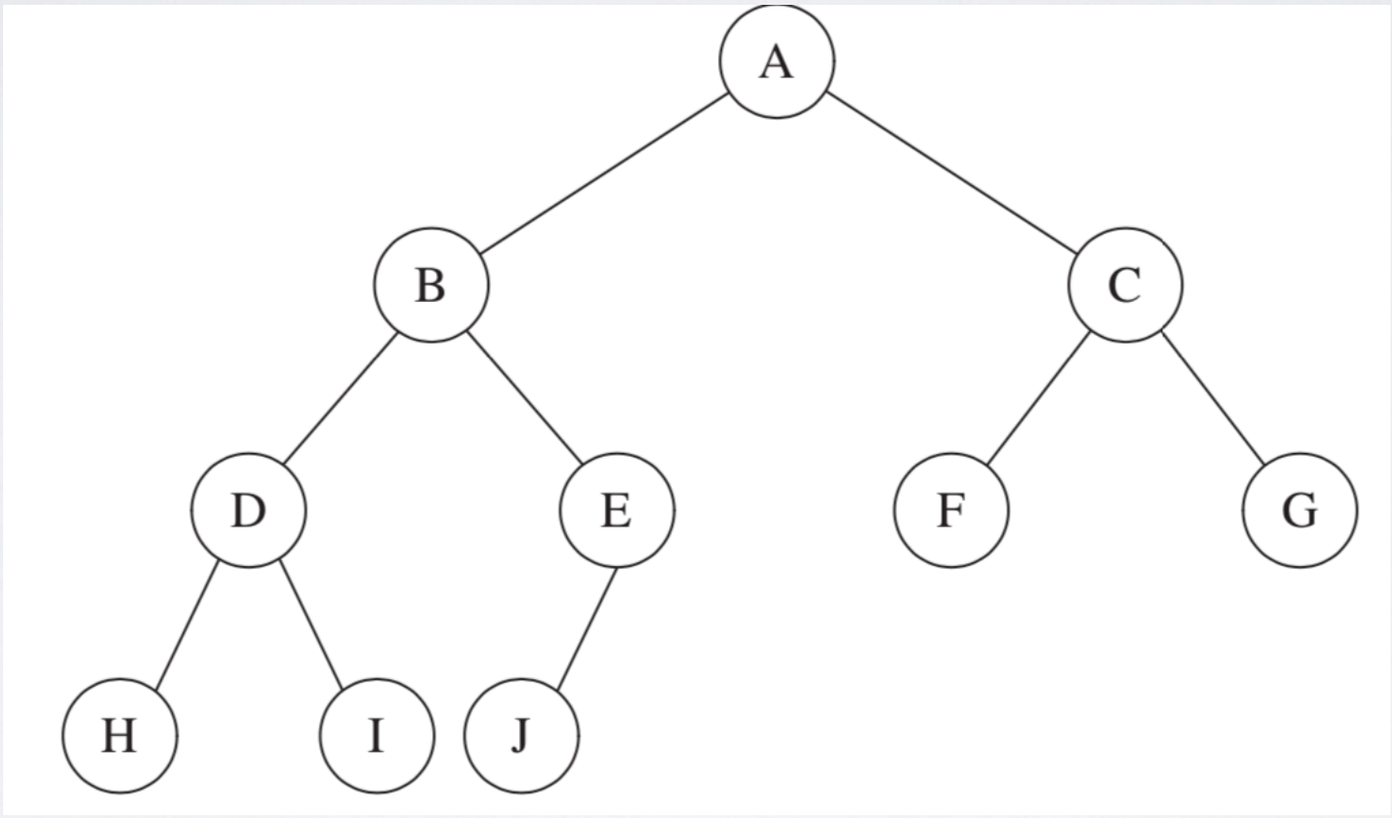

Figure: A complete binary tree.

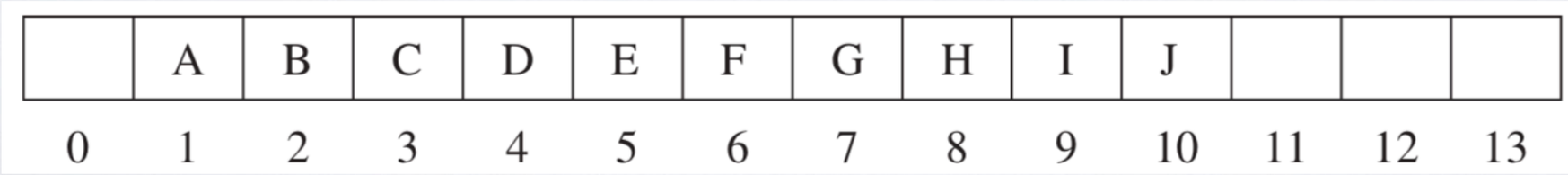

Figure: Array implementation of complete binary tree.

Heap-order Property

- The property that allows operations to be performed quickly is the heap-order property.
- In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root (which has no parent).
- By the heap-order property, the minimum element can always be found at the root. Thus, we get the extra operation, findMin, in constant time.

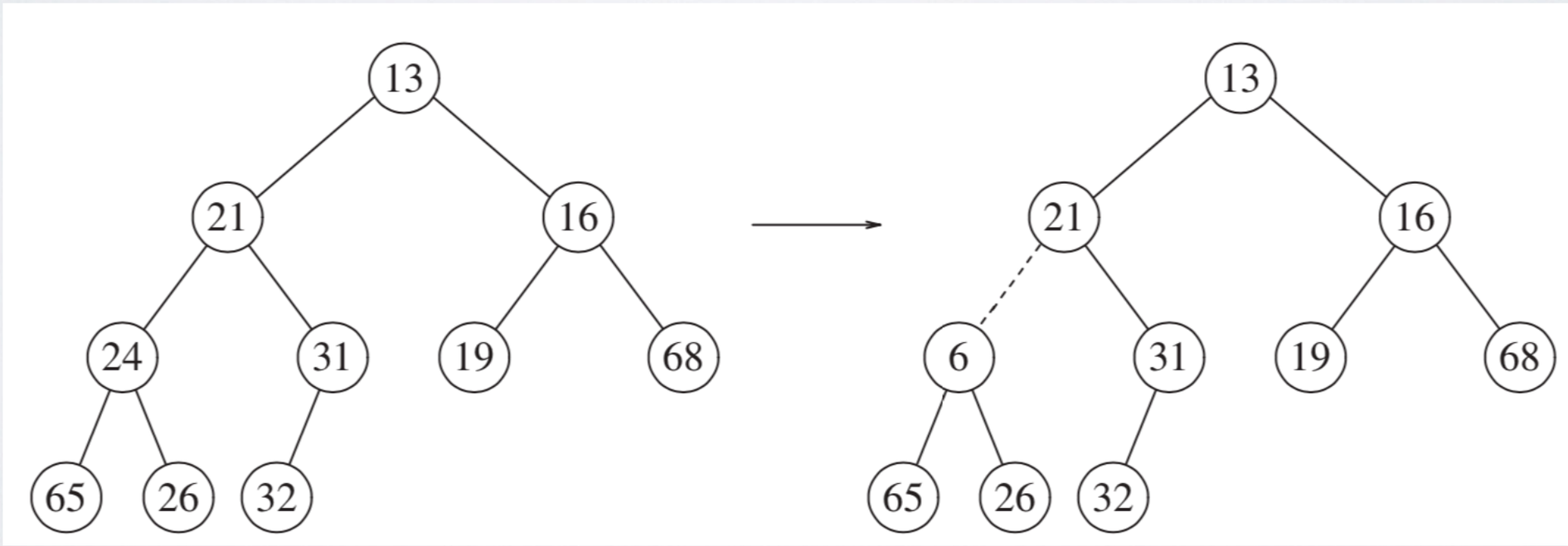

Figure: Two complete trees (only the left tree is a heap).

Insert Operation

- To insert an element X into the heap, we create a hole in the next available location, since otherwise, the tree will not be complete.
- If X can be placed in the hole without violating heap order, then we do so and are done.
- Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root.
- We continue this process until X can be placed in the hole.
- This general strategy is known as a **percolate up**; the new element is percolated up the heap until the correct location is found.

Insert Operation (Percolate up)

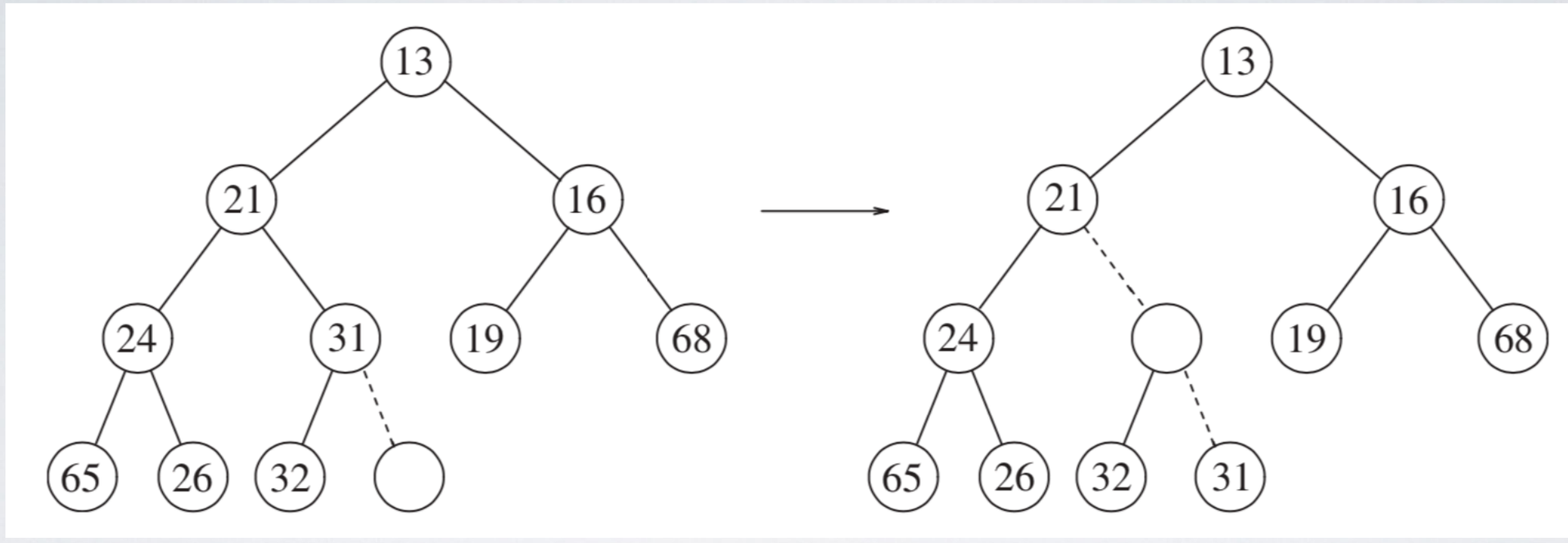

Figure: Attempt to insert 14: creating the hole, and bubbling the hole up.

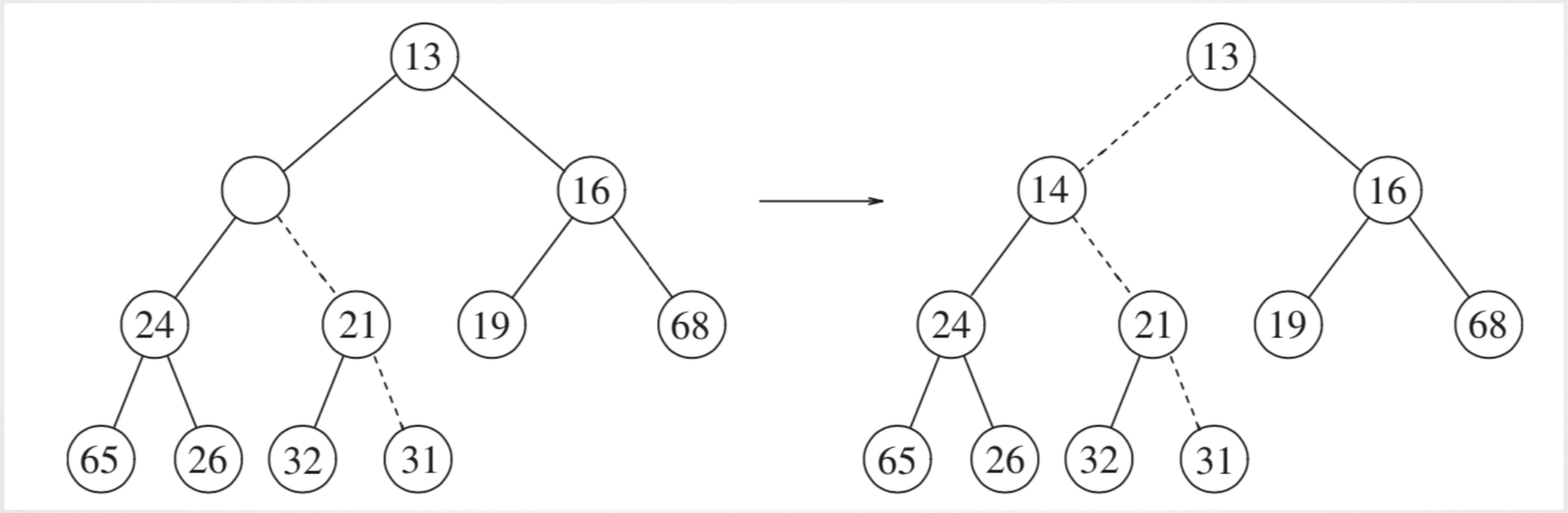

Figure: The remaining two steps to insert 14 in previous heap.

Insert Operation

- We could have implemented the percolation in the insert routine by performing repeated swaps until the correct order was established, but a swap requires three assignment statements.
- If an element is percolated up d levels, the number of assignments performed by the swaps would be 3d. Our method uses d + 1 assignments.
- The time to do the insertion could be as much as O(log N), if the element to be inserted is the new minimum and is percolated all the way to the root.

deleteMin Operation

- Finding the minimum is easy; the hard part is removing it.
- When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows that the last element X in the heap must move somewhere in the heap.
- If X can be placed in the hole, then we are done.
- This is unlikely, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until X can be placed in the hole.
- This general strategy is known as a **percolate down**.

deleteMin Operation (Percolate down)

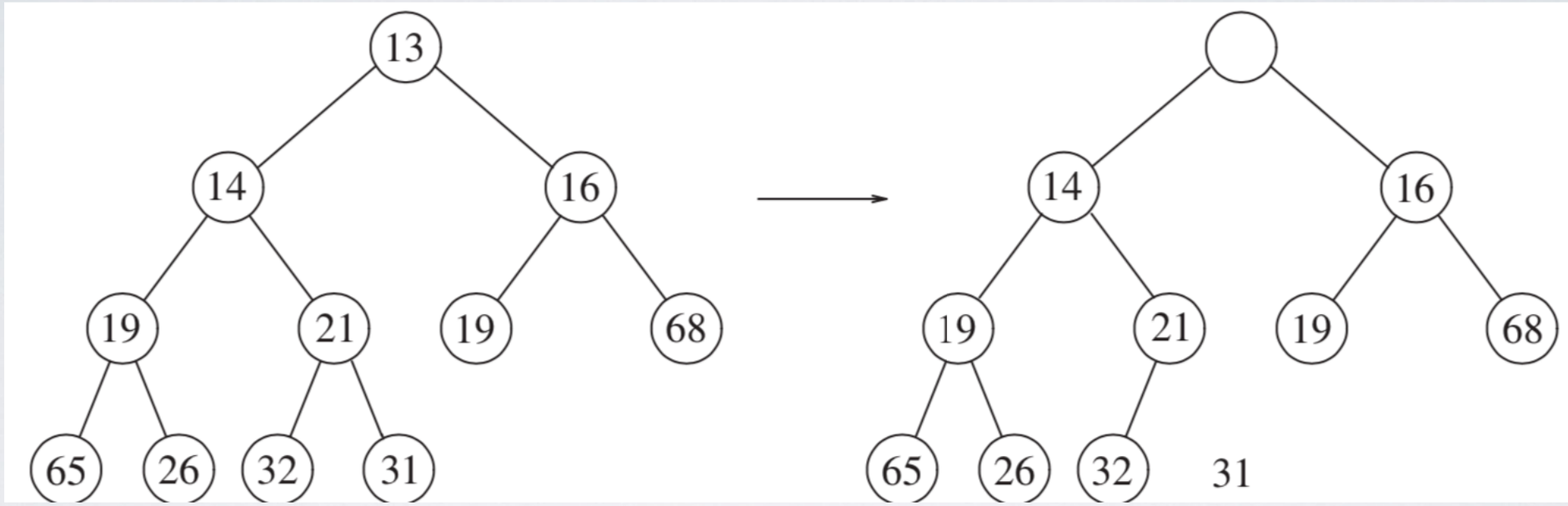

Figure: Creation of the hole at the root.

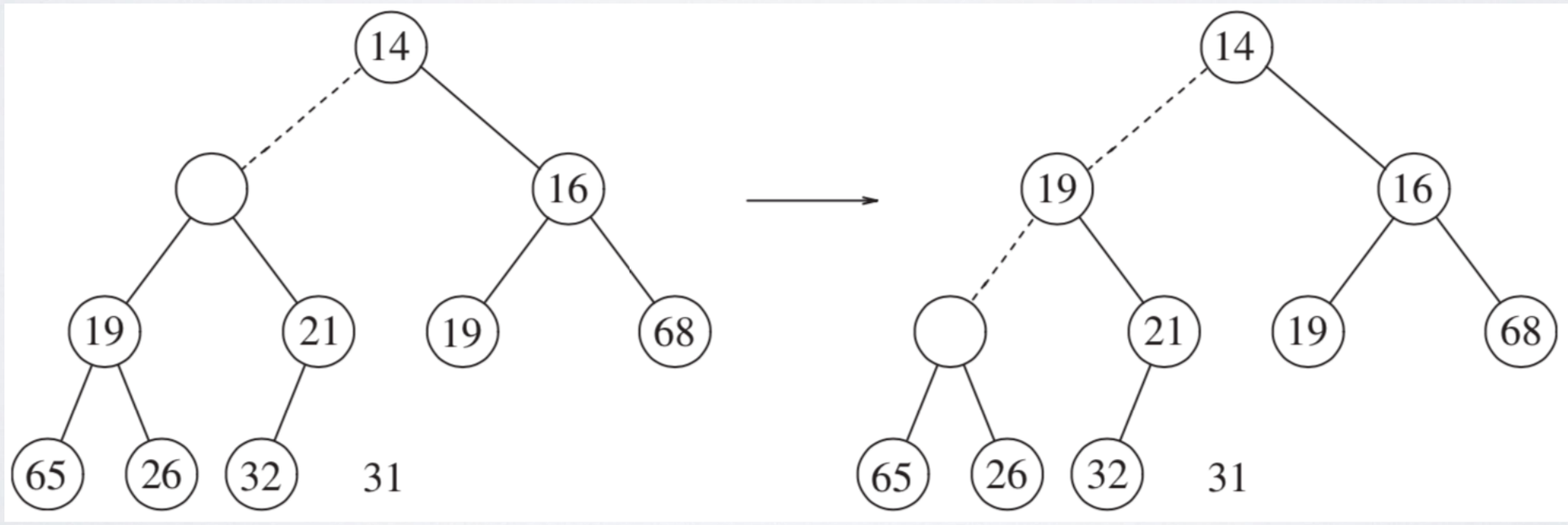

Figure: Next two steps in deleteMin.

deleteMin Operation (Percolate down)

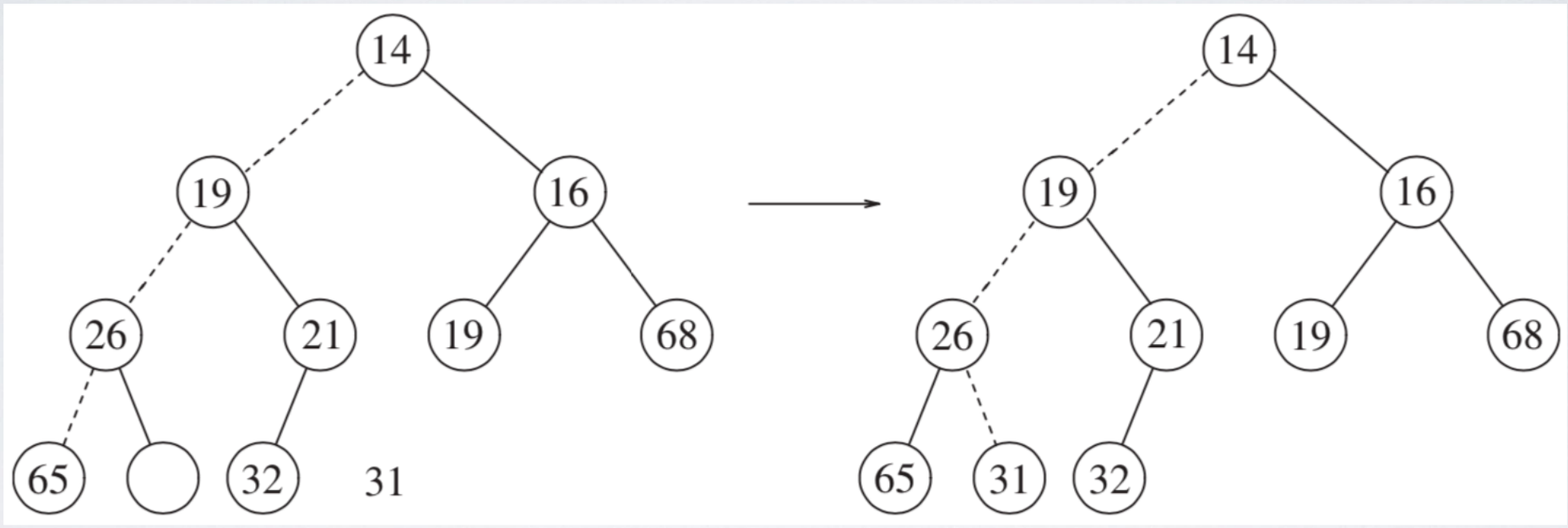

Figure: Last two steps in deleteMin.

buildHeap Operation

- The binary heap is sometimes constructed from an initial collection of items. This constructor takes as input N items and places them into a heap.
- Obviously, this can be done with N successive inserts.
- Since each insert will take O(1) average and O(log N) worst-case time, the total running time of this algorithm would be O(N) average but O(N log N) worst-case.
- The general algorithm is to place the N items into the tree in any order, maintaining the structure property.
- Then, percolateDown operation is followed.

buildHeap Operation

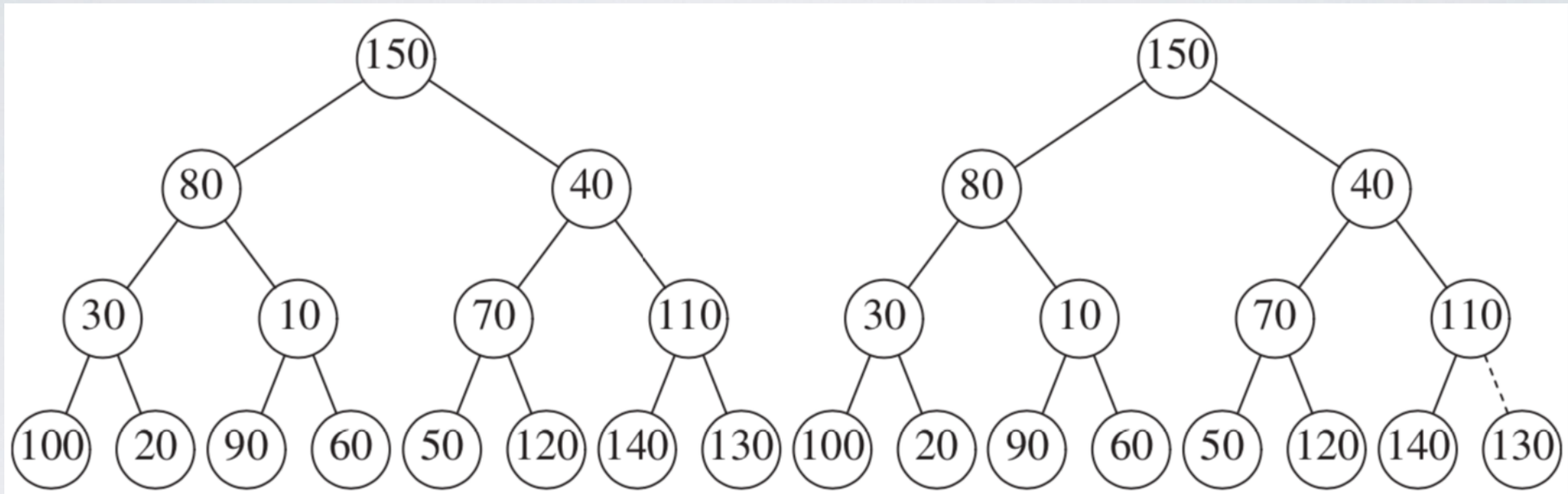

Figure: Left: initial heap; right: after percolateDown(7)

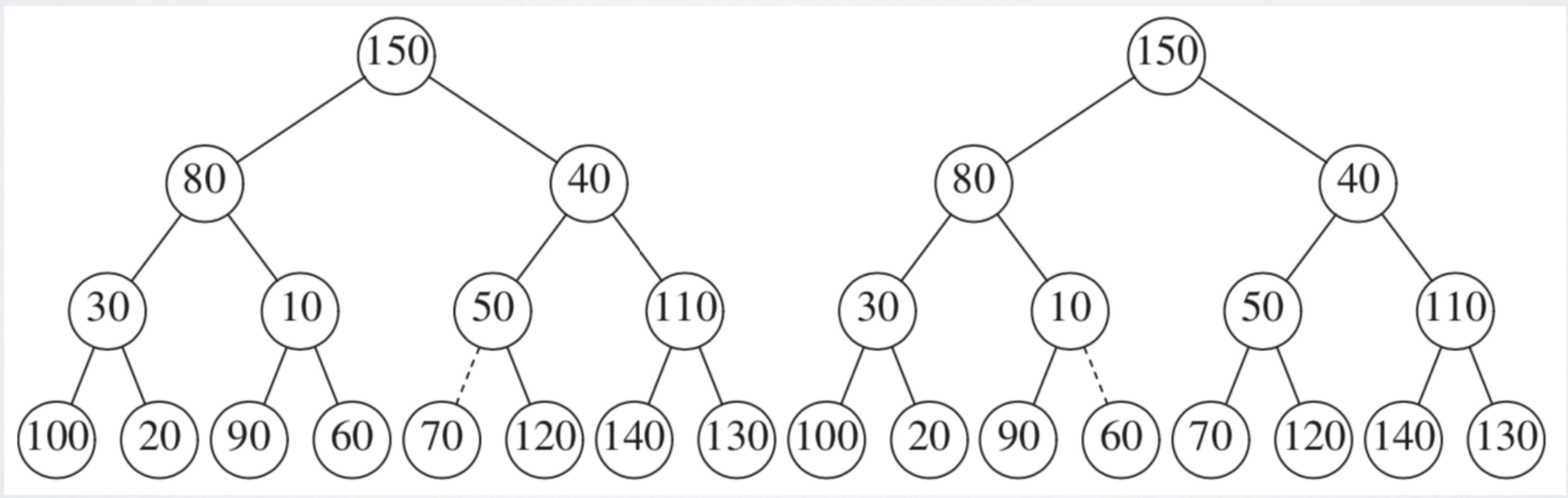

Figure: Left: after percolateDown(6); right: after percolateDown(5)

buildHeap Operation

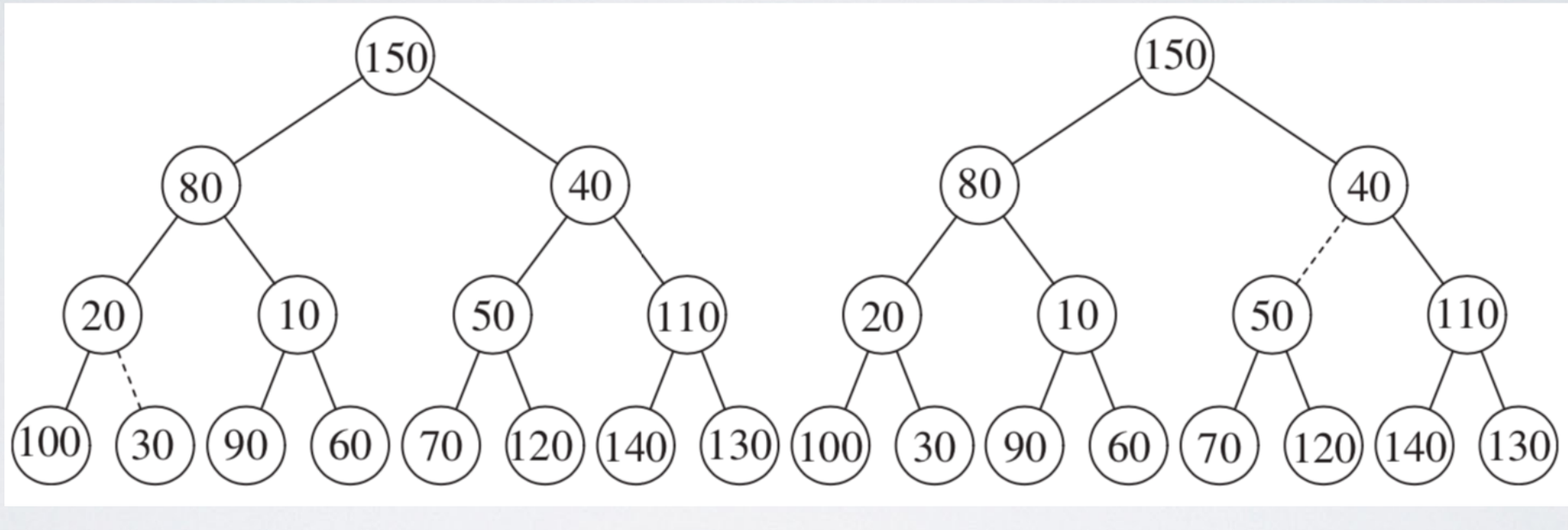

Figure: Left: after percolateDown(4); right: after percolateDown(3)

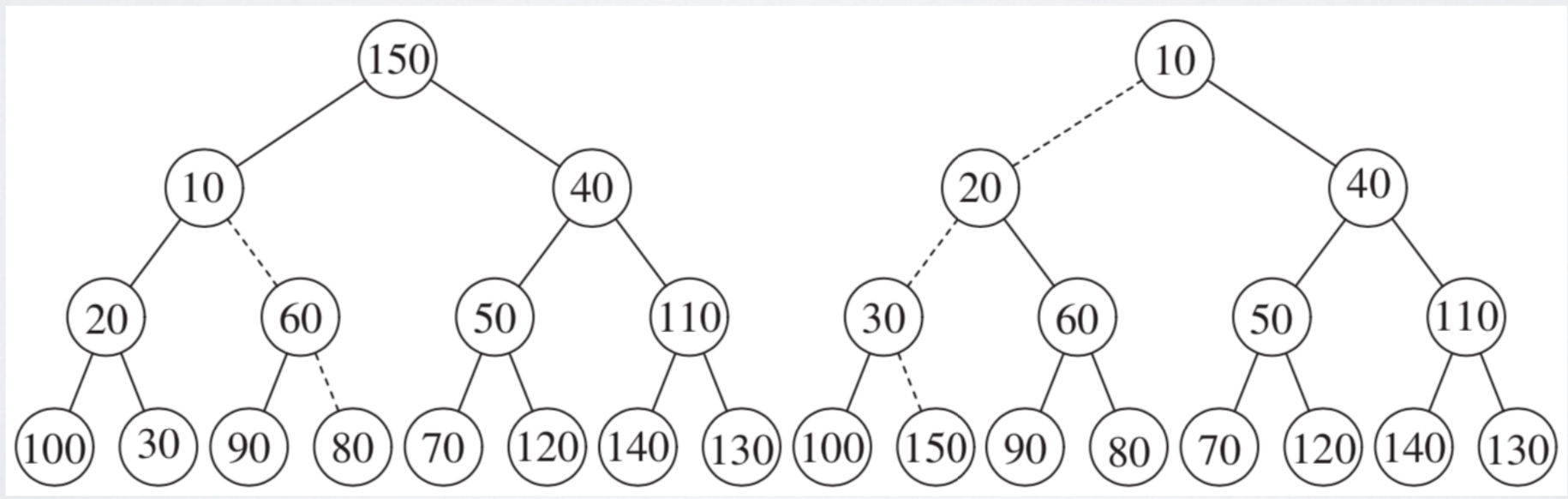

Figure: Left: after percolateDown(2); right: after percolateDown(1)

decreaseKey Operation

- The decreaseKey(p,ß) operation lowers the value of the item at position p by a positive amount ß.
- Since this might violate the heap order, it must be fixed by a percolate up.
- This operation could be useful to system administrators: They can make their programs run with highest priority.

increaseKey Operation

- The increaseKey(p,ß) operation increases the value of the item at position p by a positive amount ß.
- This is done with a percolate down.
- Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

Heap Visualization

https://www.cs.usfca.edu/~galles/visualization/Heap.html