

# Data Structures

---

## **21. Heap (Priority Queues)**

# Motivation

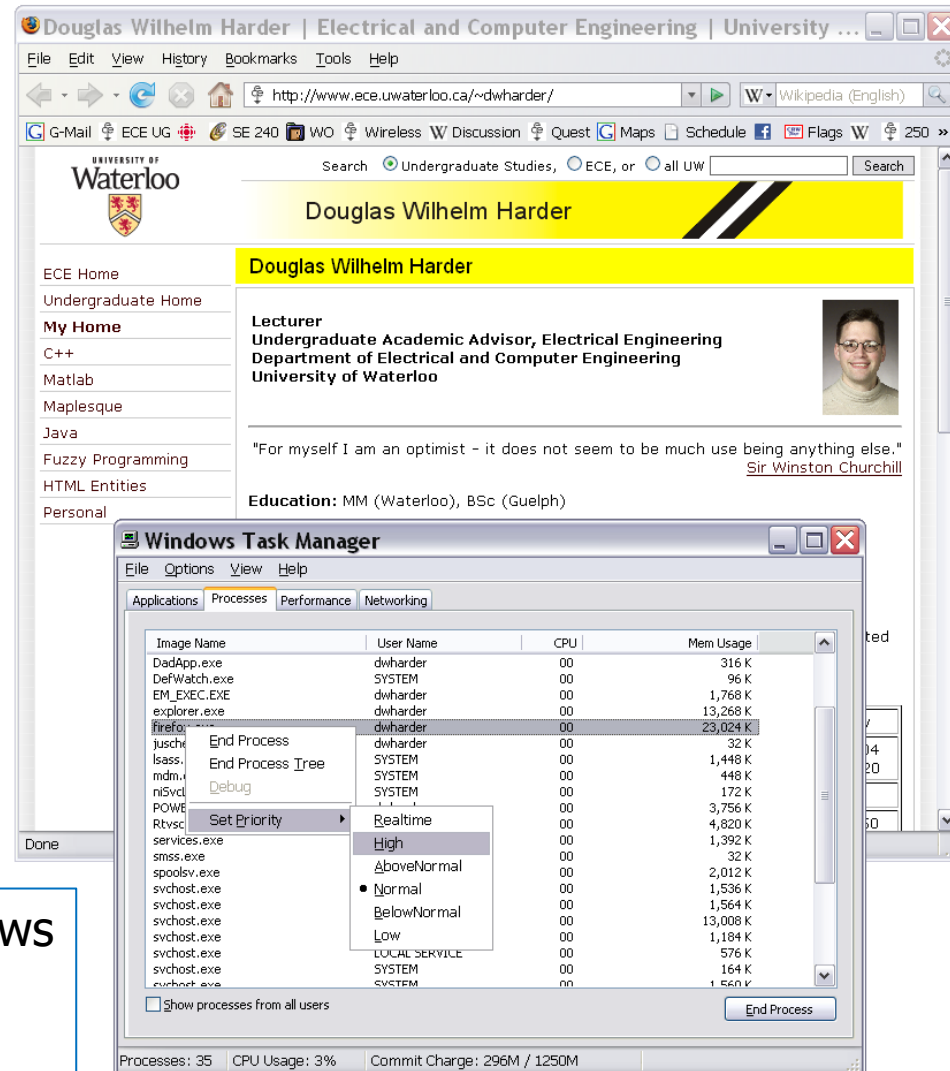
---

- With **queues** the order may be summarized by **first in, first out**
- Some tasks may be more important or timely than others
  - Higher priority
- **Priority queues**
  - Enqueue objects using a partial ordering based on priority
  - Dequeue that object which has highest priority

# Applications Of Priority Queue

- Hold jobs for a printer in order of length
- Store packets on network routers in order of urgency
- Ordering CPU jobs
- Emergency room admission processing

The priority of processes in Windows may be set in the Windows Task Manager



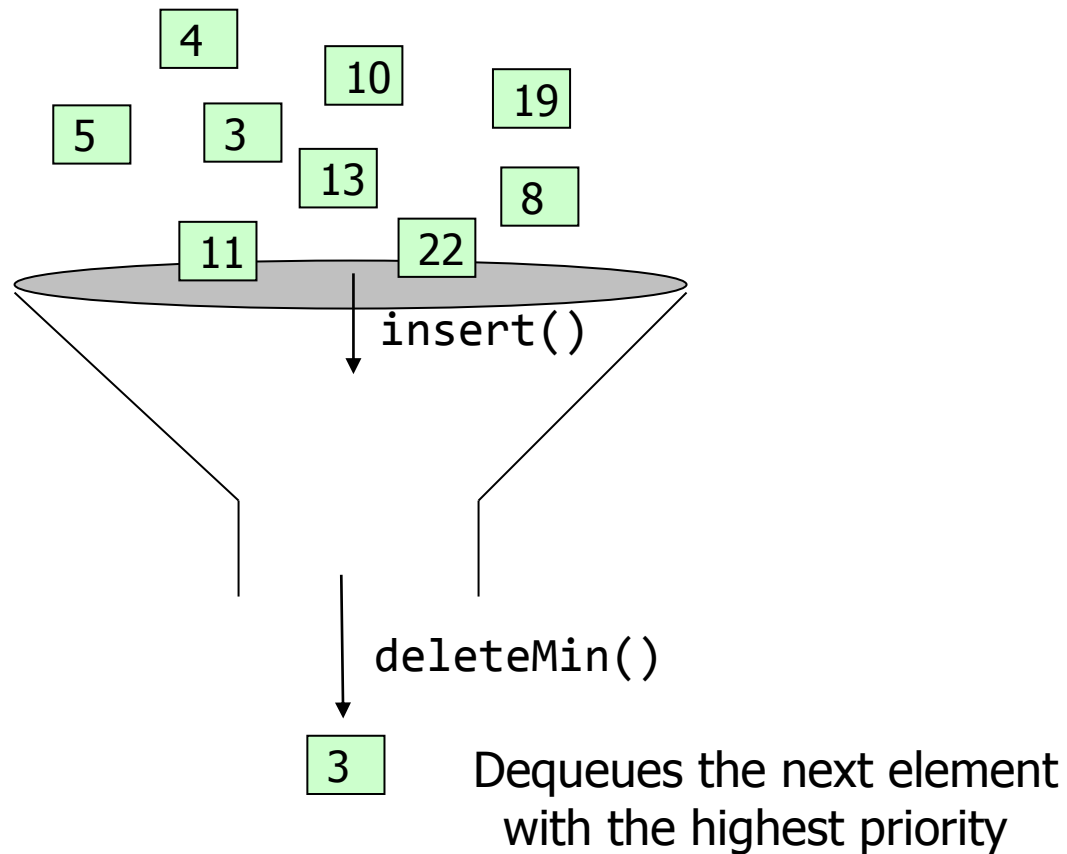
# Priority Queue – ADT

---

- **insert** (i.e., enqueue)
  - Dynamic insert
  - Specification of a priority level (0-high, 1,2.. Low)
- **deleteMin** (i.e., dequeue)
  - Returns the current “highest priority” element in the queue
    - Element with the minimum priority level
  - Deletes that element from the queue
- Performance goal is to make the run time of each operation as close to  $O(1)$  as possible

# Priority Queue – ADT

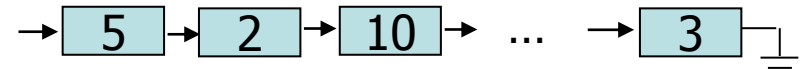
---



# Simple Implementations

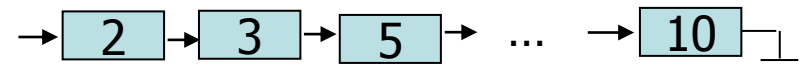
- **Unordered linked list**

- Insert –  $O(1)$  step
- deleteMin –  $O(n)$  steps



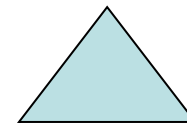
- **Ordered linked list**

- insert –  $O(n)$  steps
- deleteMin –  $O(1)$  step



- **Balanced binary tree**, e.g., AVL Tree

- insert –  $O(\log_2 n)$  steps
- deleteMin in how many steps?
  - Find min –  $O(\log_2 n)$  steps
  - Delete –  $O(\log_2 n)$  steps



Can we build a data structure better suited to store and retrieve priorities?

---

# Binary Heap

# Binary Heap

---

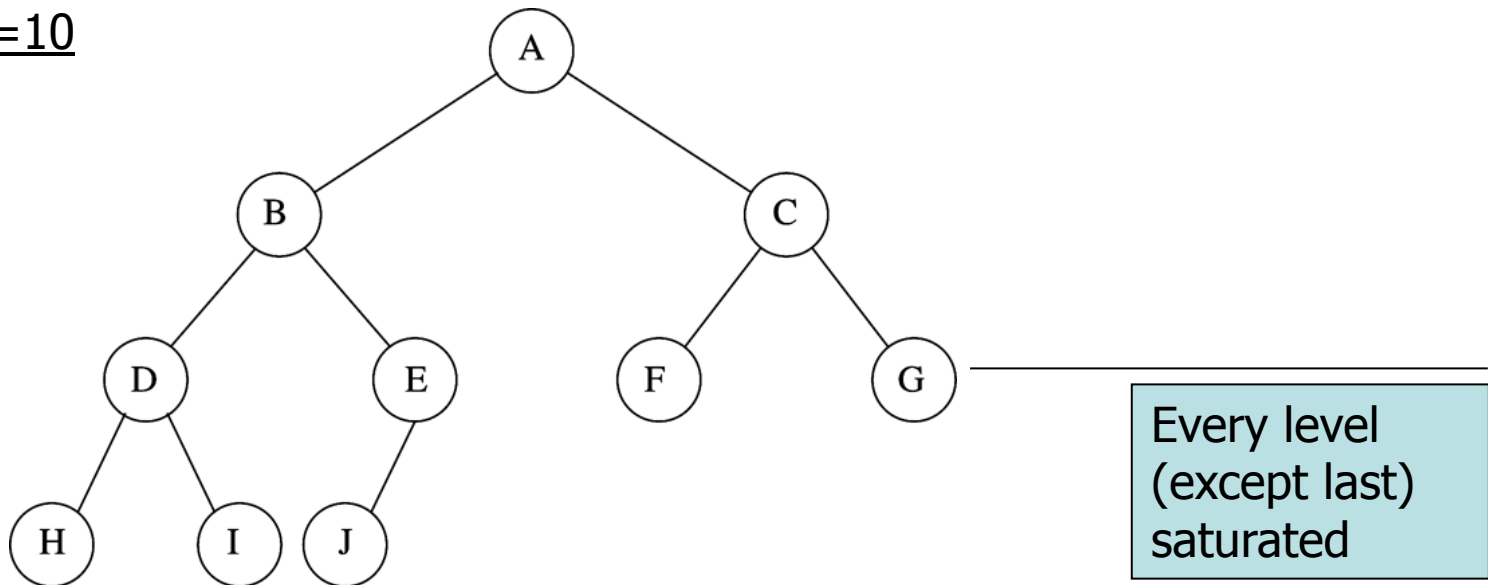
- A binary heap is a binary tree with two properties
  - Structure property
  - Heap-order property



# Binary Heap – Structure Property

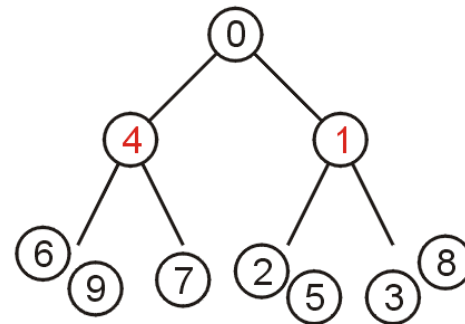
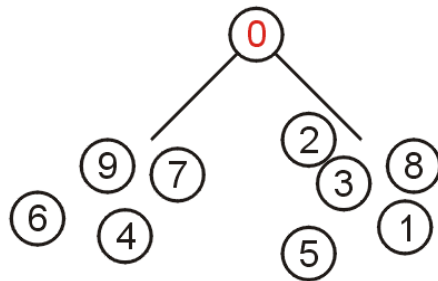
- A **binary heap** is **(almost) complete** binary tree
  - Each level (except possibly the bottom most level) is completely filled
  - The bottom most level may be partially filled (from left to right)

N=10



# Binary Heap – Heap-Order Property

- **Min-Heap** property
  - Key associated with the root is less than or equal to the keys associated with either of the sub-trees (if any)
  - Both of the sub-trees (if any) are also binary min-heaps



- **Properties** of min-heap
  - A single node is a min-heap
  - **Minimum** key always at **root**
  - For every node  $X$ ,  $\text{key}(\text{parent}(X)) \leq \text{key}(X)$
  - **No relationship** between nodes with **similar key**

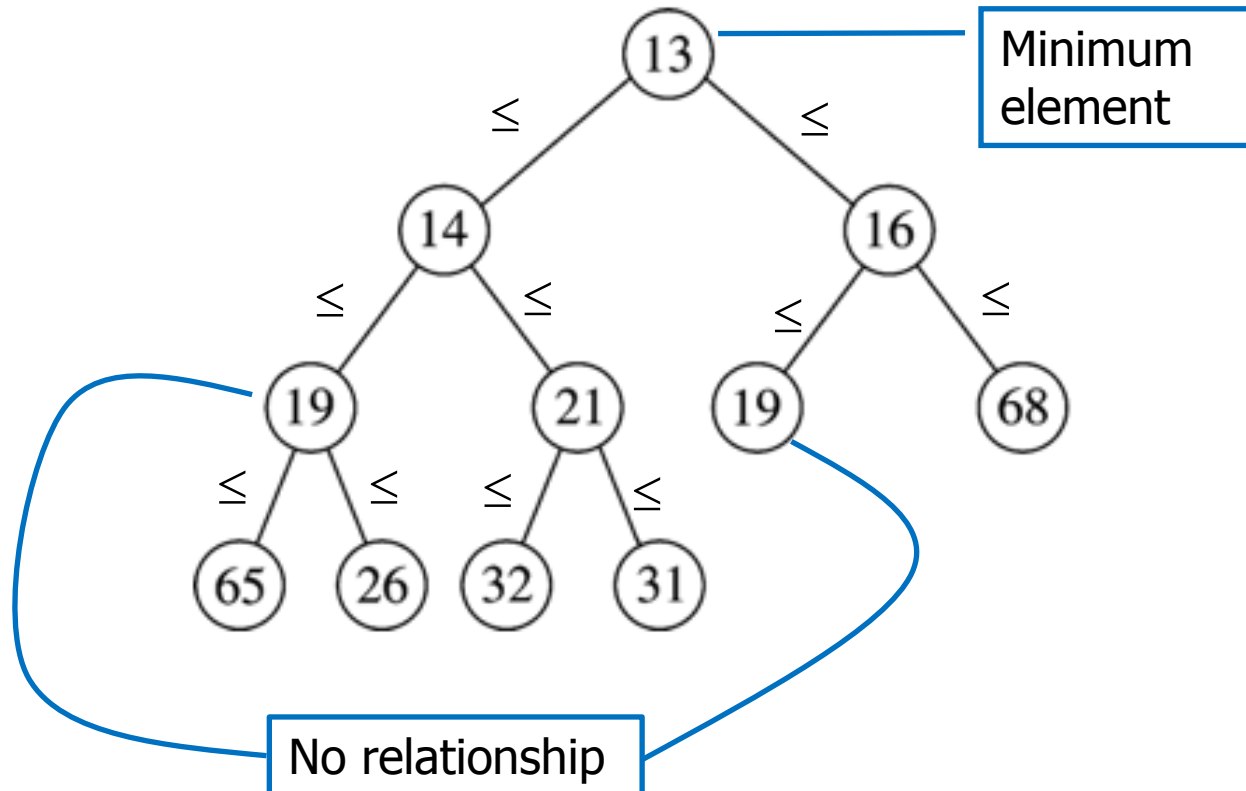
# Binary Heap – Heap-Order Property

---

- Max-Heap property
  - Maximum key at the root
  - For every node  $X$ ,  $\text{key}(\text{parent}(X)) \geq \text{key}(X)$
- Insert and deleteMin must maintain heap-order property

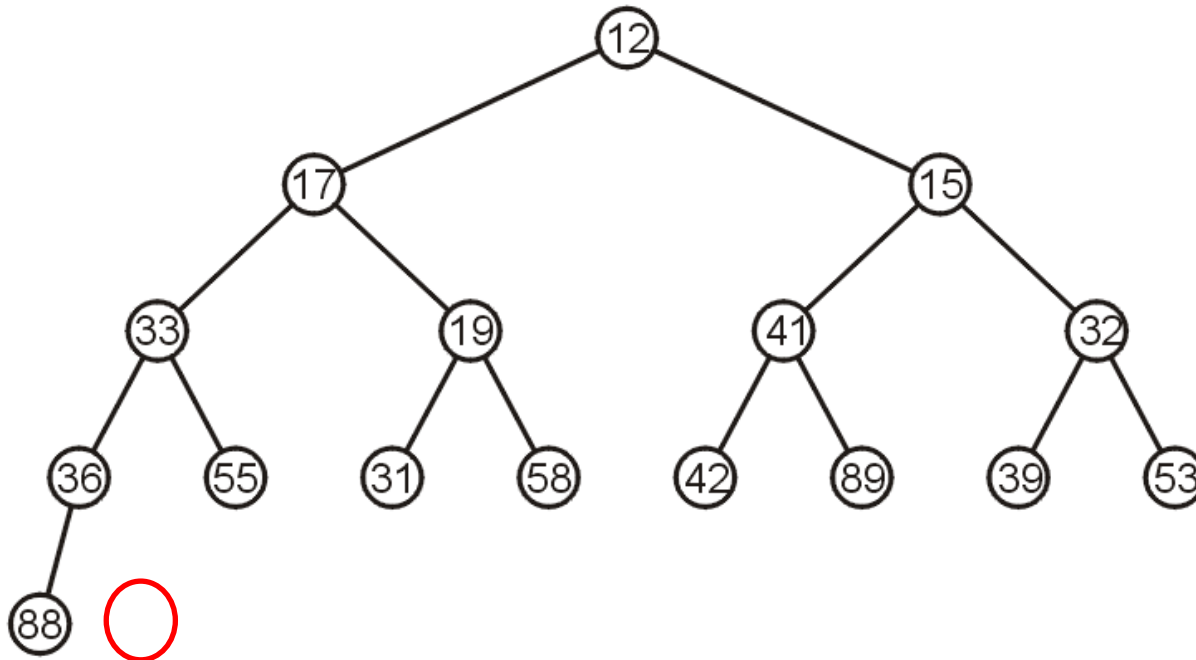
# Heap-Order Property – Example

- Min-Heap



# Heap Operations – insert

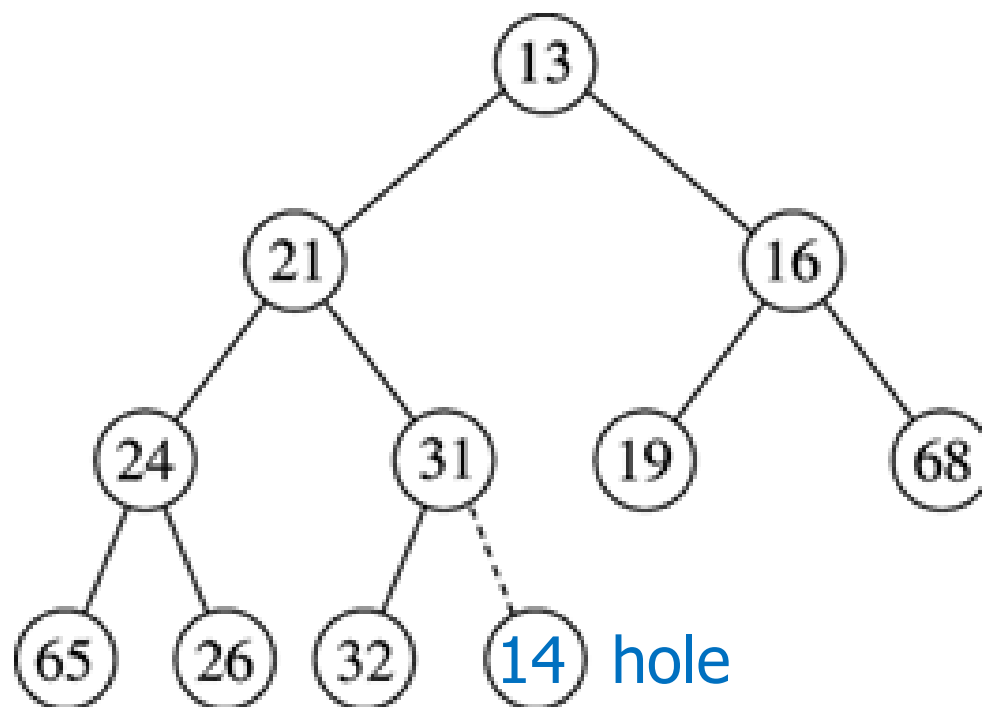
- Insert new element into the heap at the next available slot (“hole”)
  - Maintaining (almost) complete binary tree
- **Percolate** the element **up** the heap while heap-order property not satisfied



# Heap Insert – Example

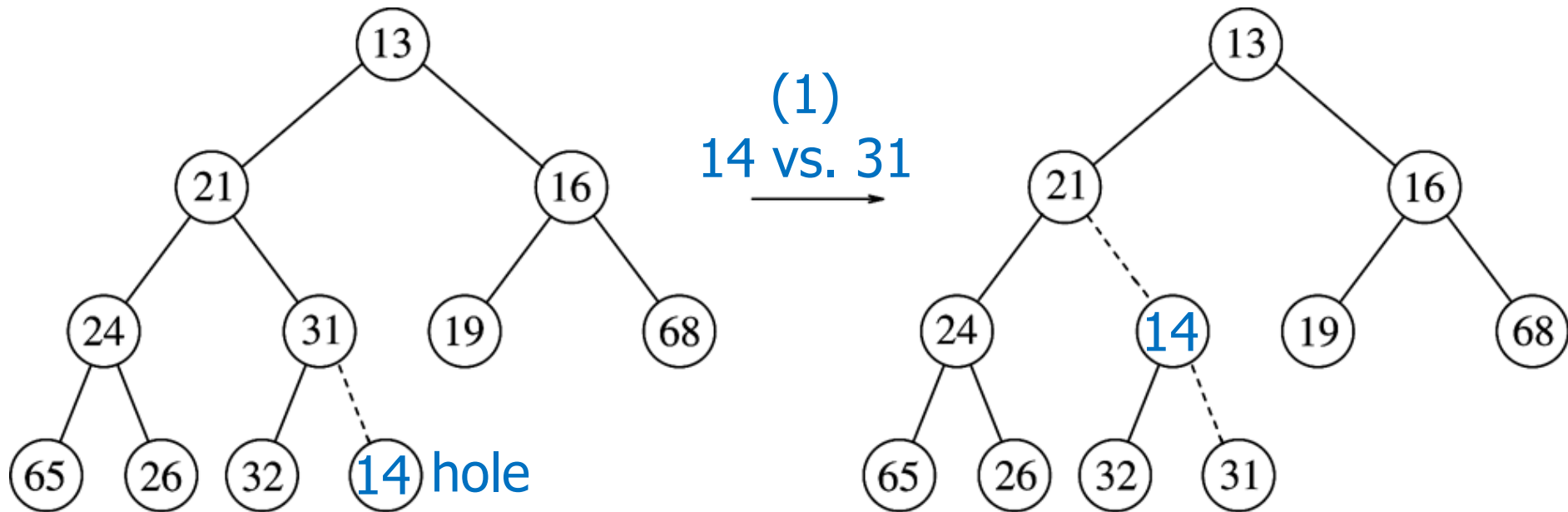
---

- Insert 14



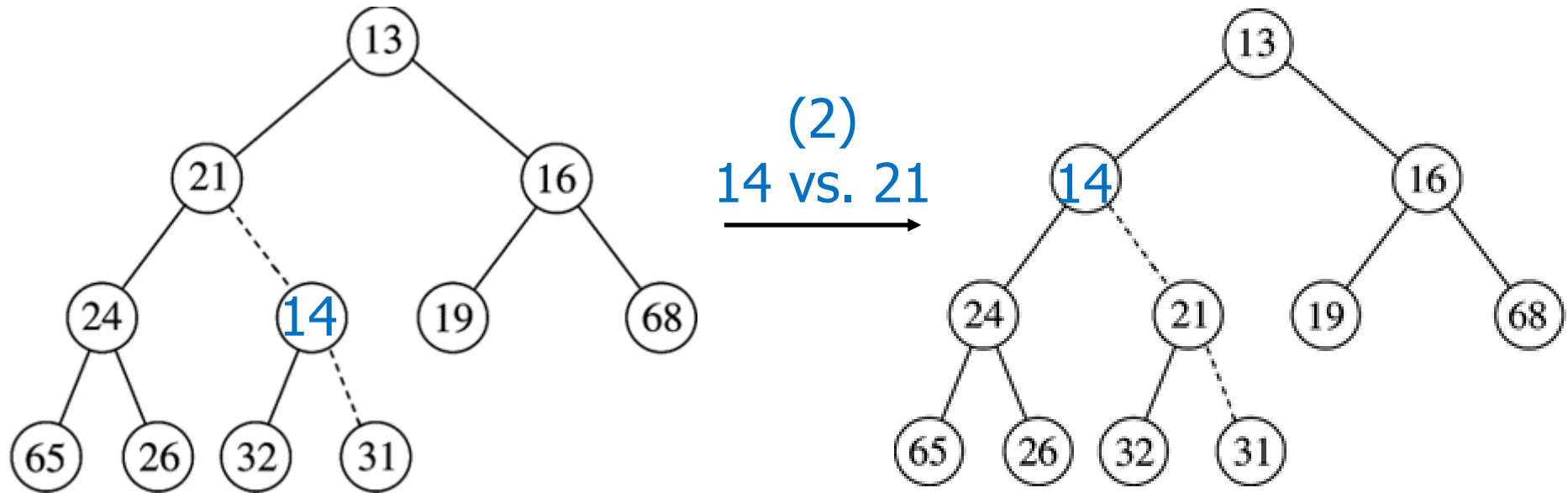
# Heap Insert – Example

- Insert 14



# Heap Insert – Example

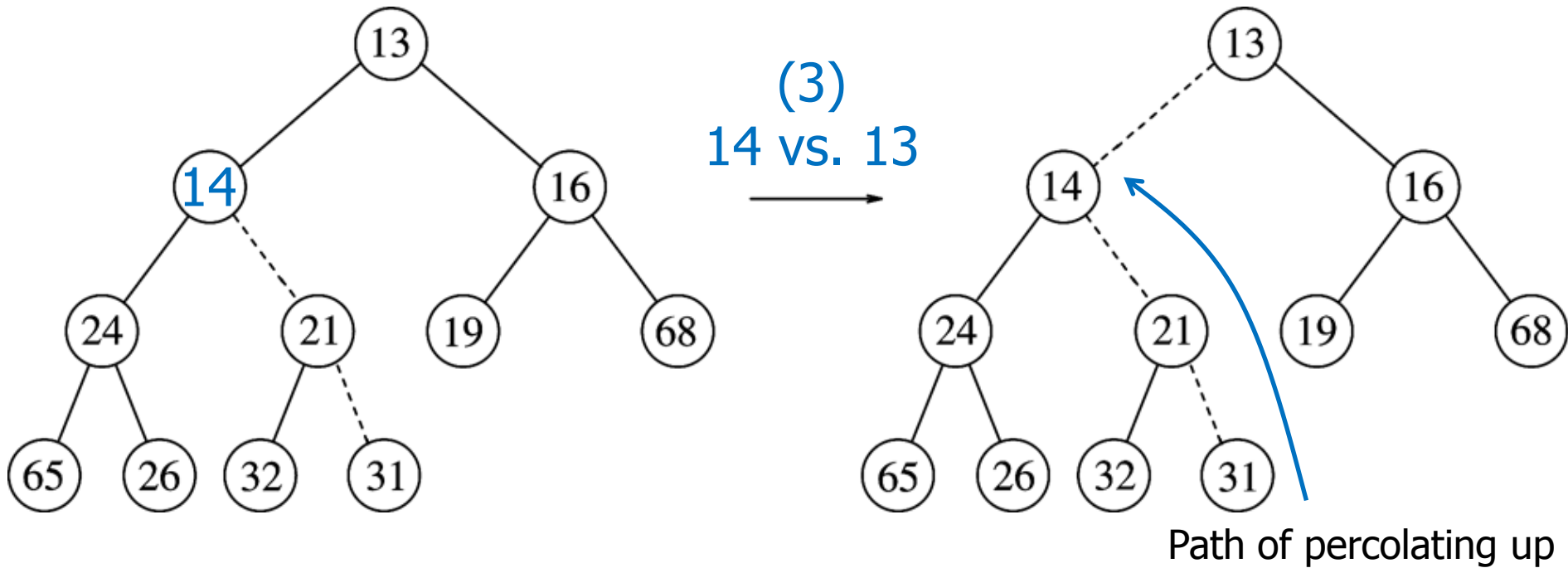
- Insert 14





# Heap Insert – Example

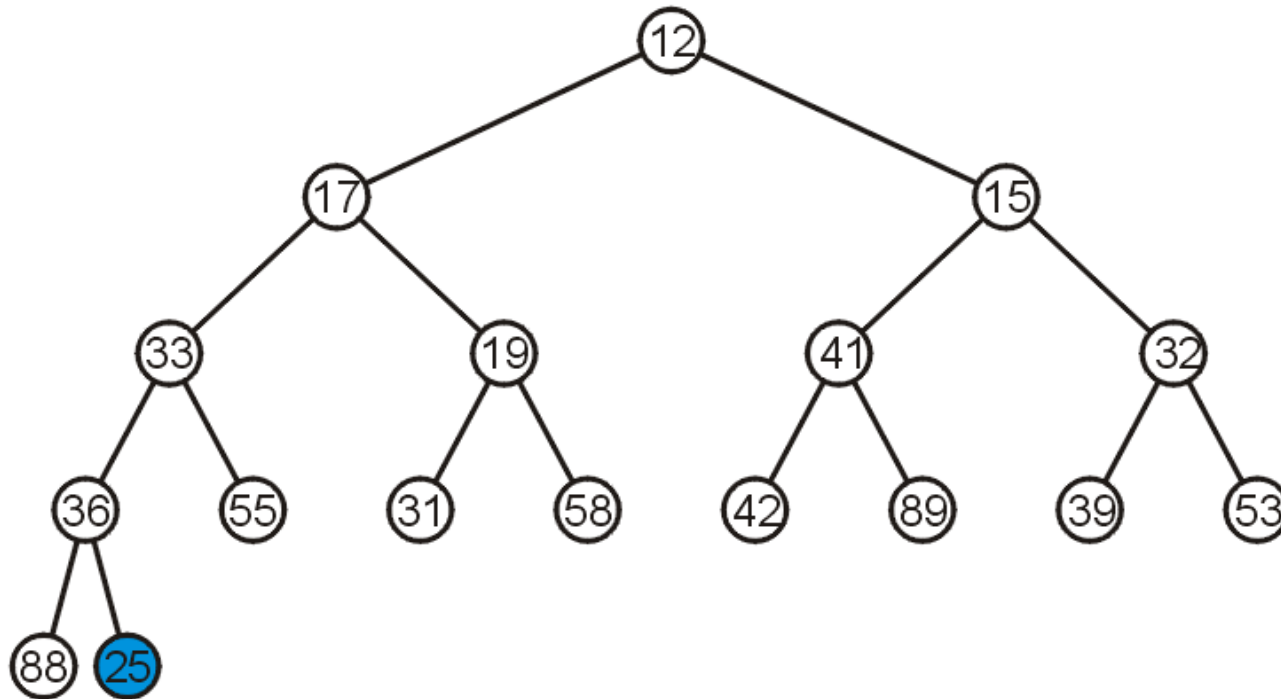
- Insert 14



- ✓ Heap order property
- ✓ Structure property

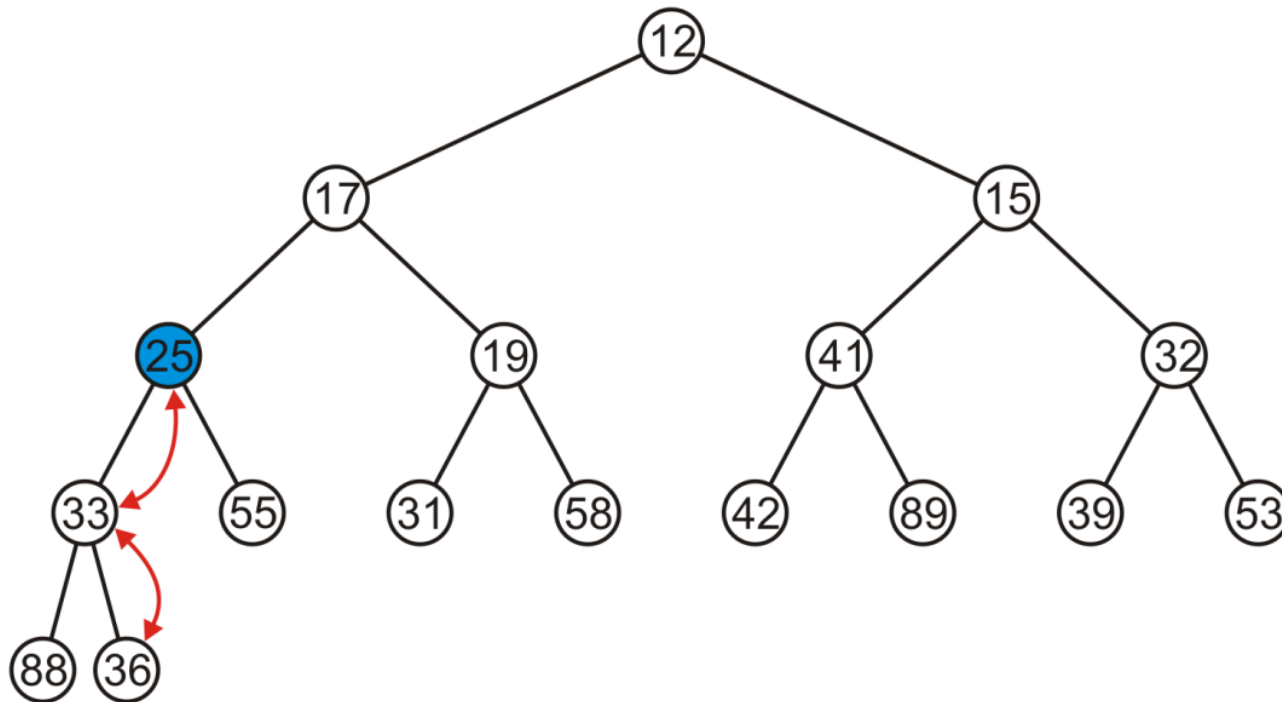
# Heap Insert – Example

- Insert 25



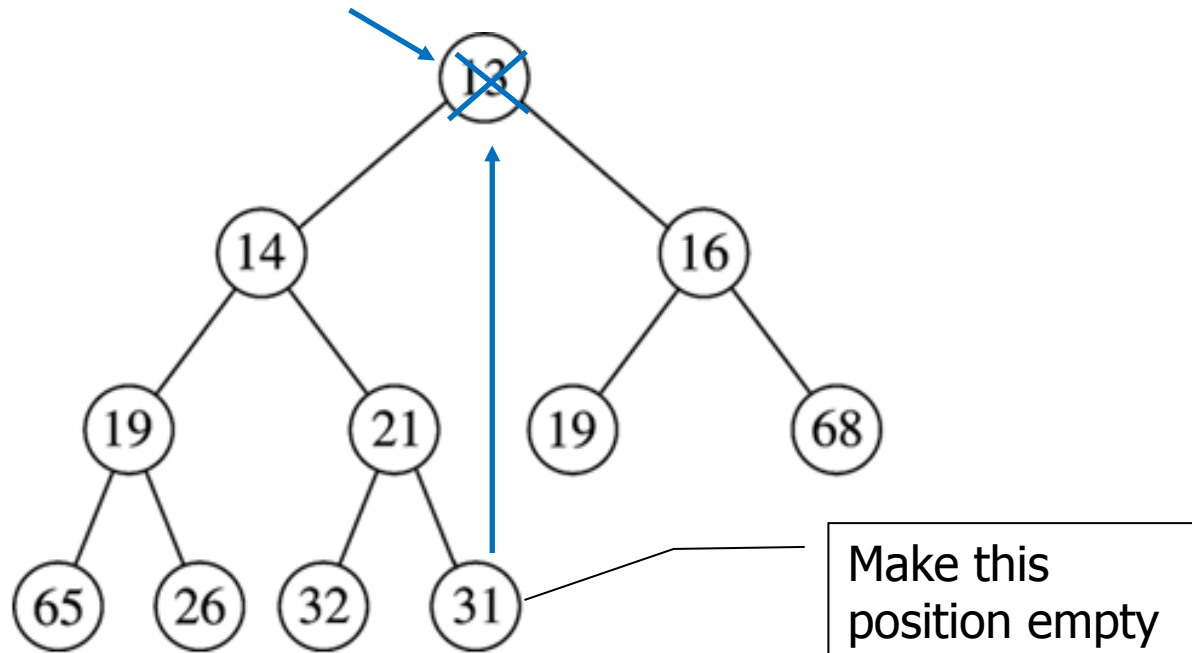
# Heap Insert – Example

- Percolate 25 up into its appropriate location
  - The resulting heap is still a complete tree

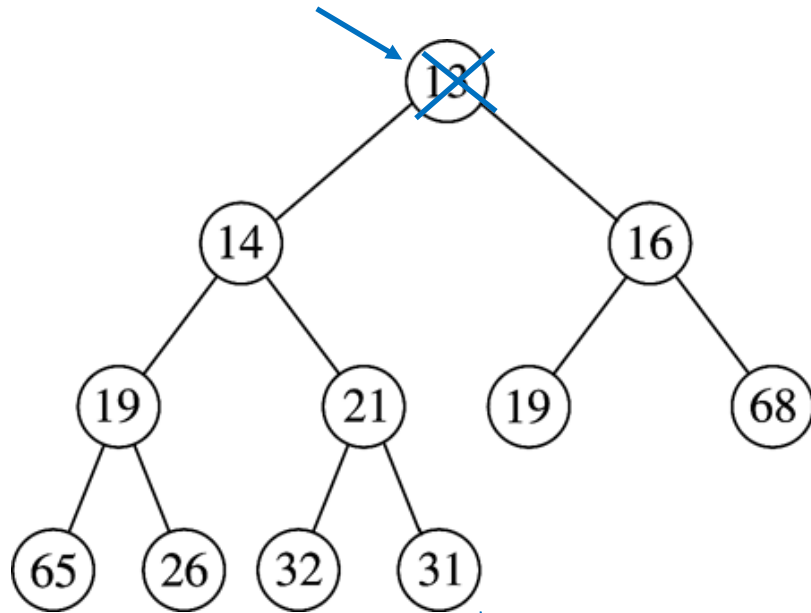


# Heap Operation – deleteMin

- Minimum element is always at the root
  - Return the element at the root and delete it
- Heap decreases by one in size
- Move last element of the tree into hole at root
- **Percolate down** while heap-order property not satisfied

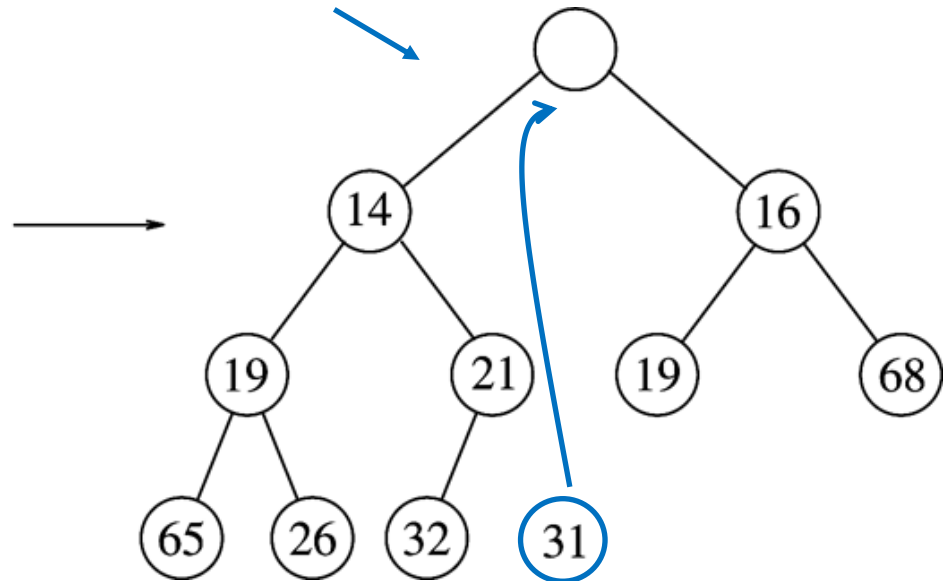


# deleteMin – Example



Make this  
position  
empty

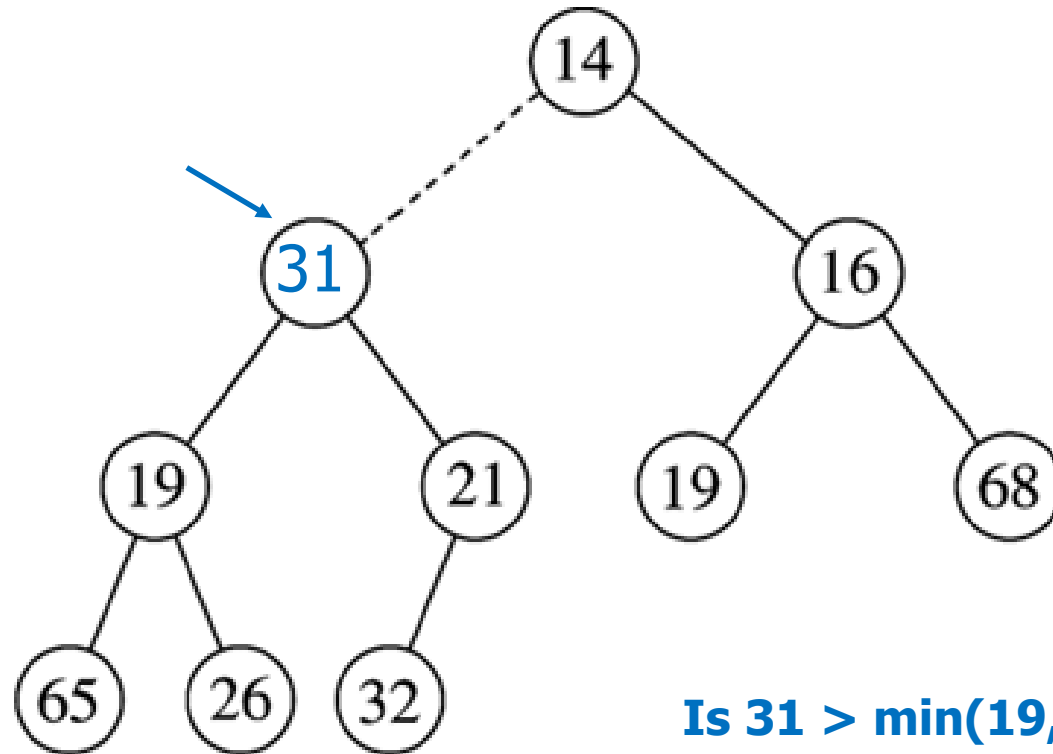
Copy 31 temporarily  
here and move it down



**Is 31 > min(14,16)?**  
- Yes - swap 31 with min(14,16)

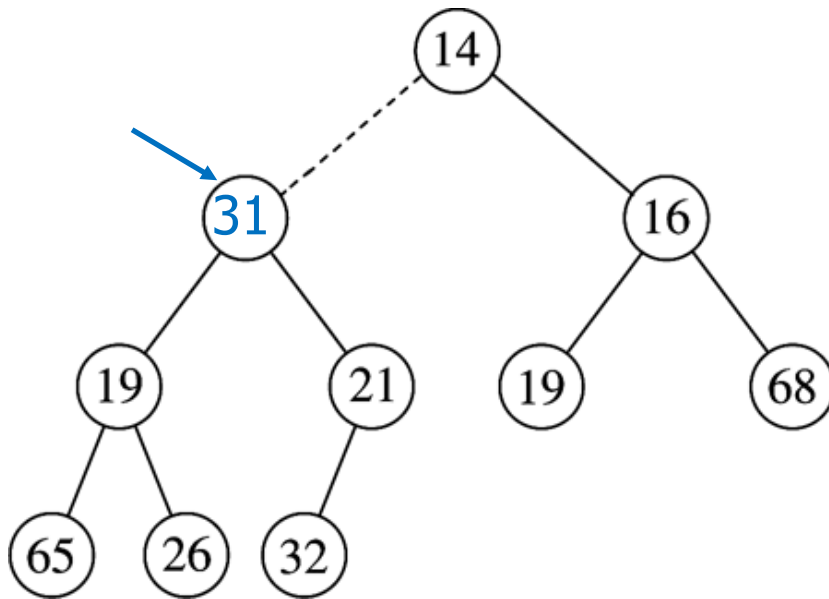
## deleteMin – Example

---



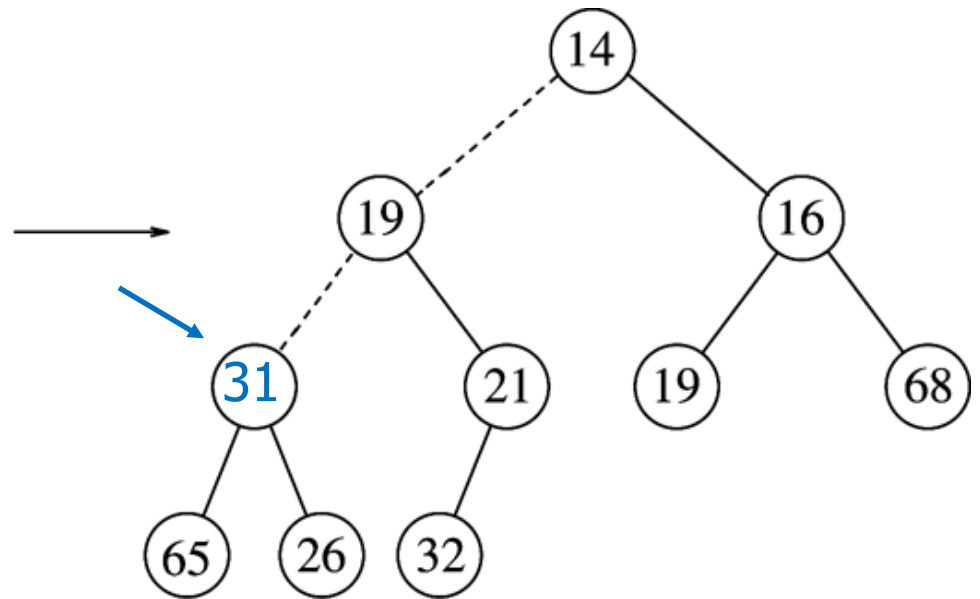
**Is 31 > min(19,21)?**  
- Yes - swap 31 with min(19,21)

# deleteMin – Example



**Is 31 > min(19,21)?**

- Yes - swap 31 with min(19,21)

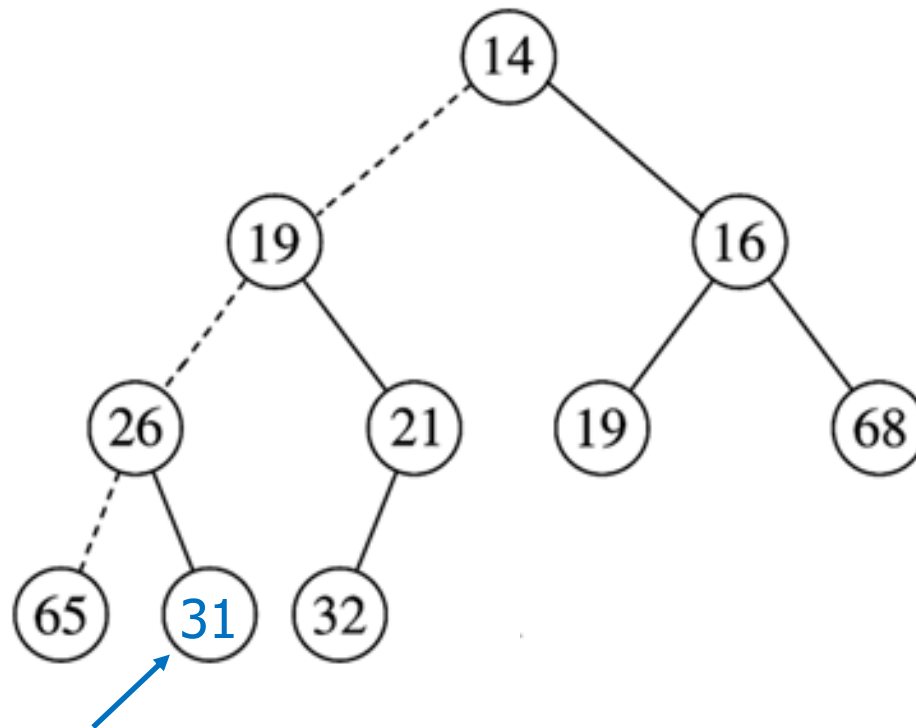


**Is 31 > min(65,26)?**

- Yes - swap 31 with min(65,26)

## deleteMin – Example

---

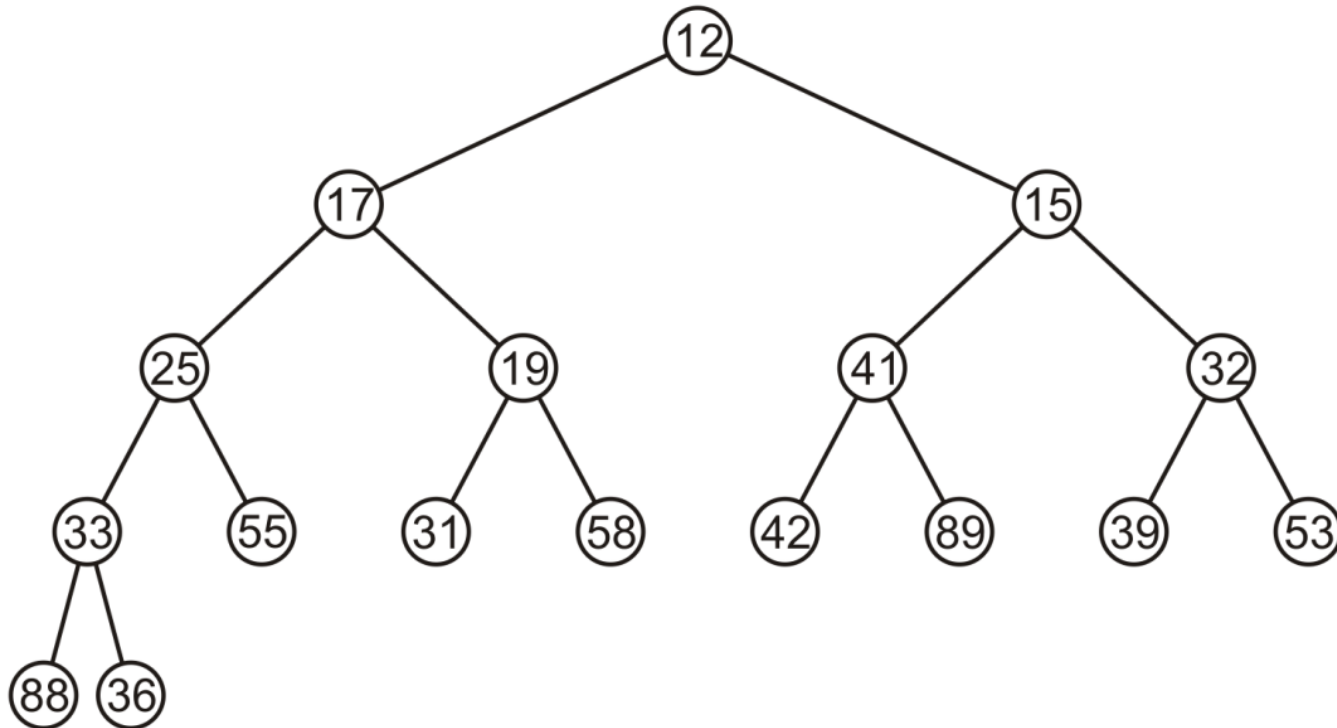




## deleteMin – Example

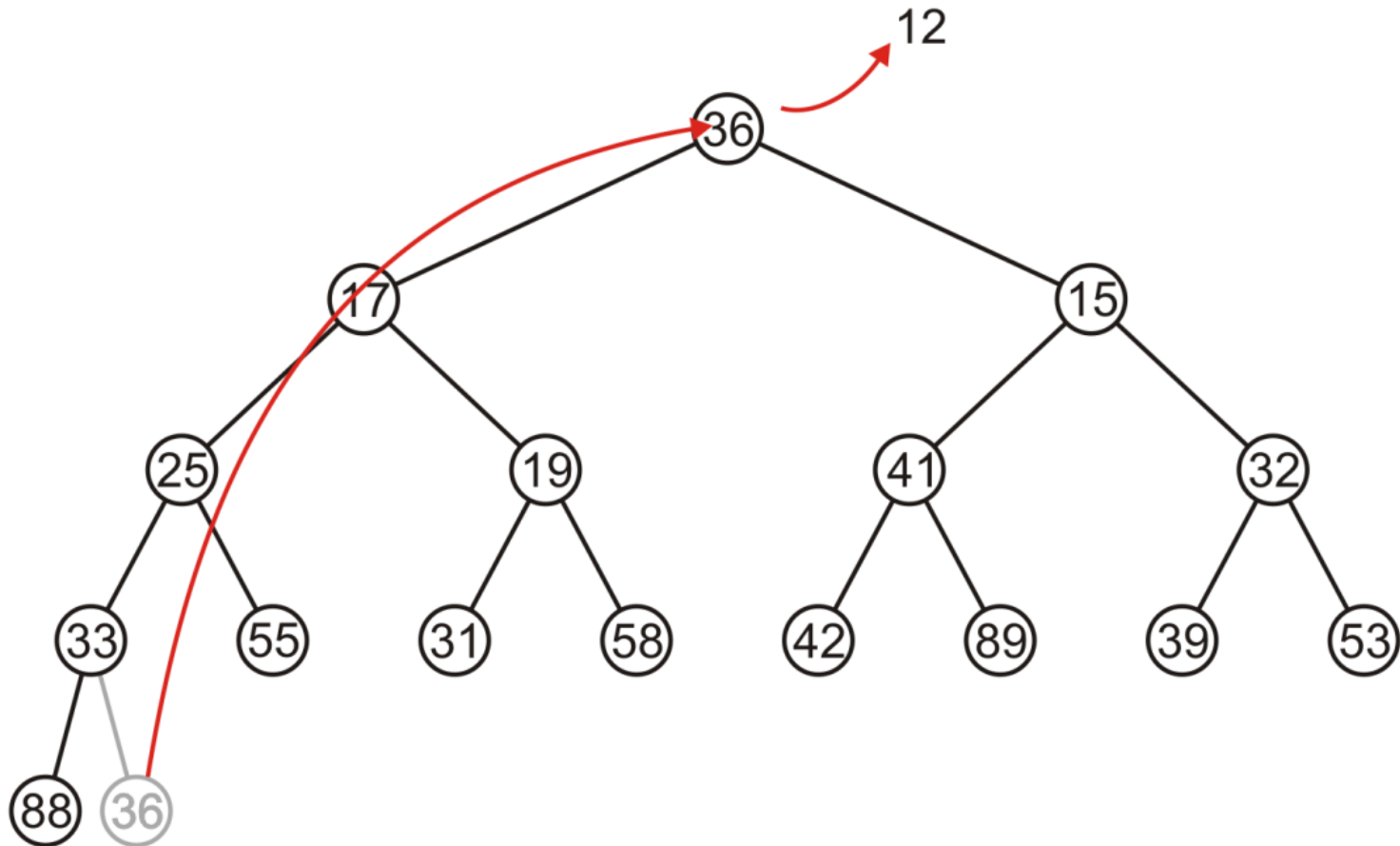
---

- deleteMin will dequeue element 12 from the top



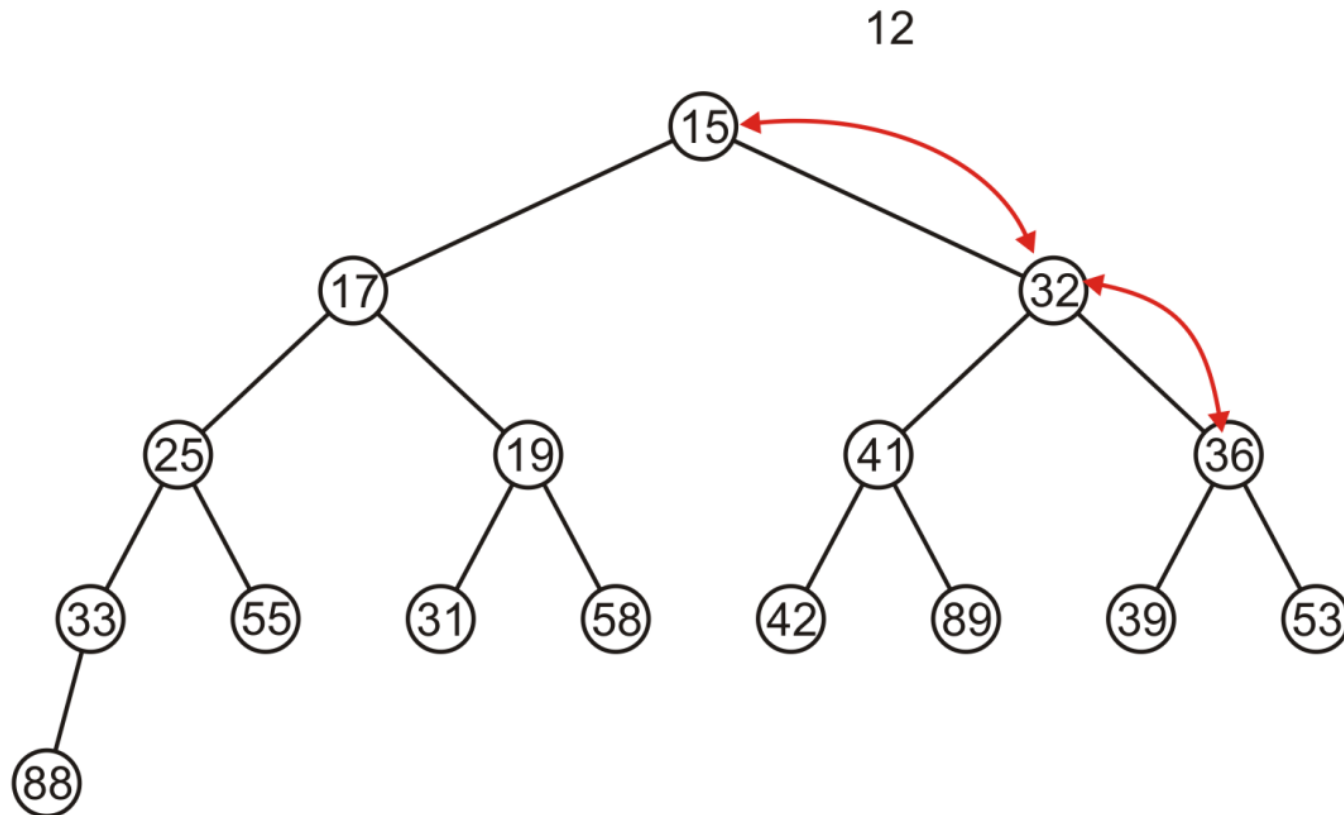
## deleteMin – Example

- Copy the last entry in the heap to the root

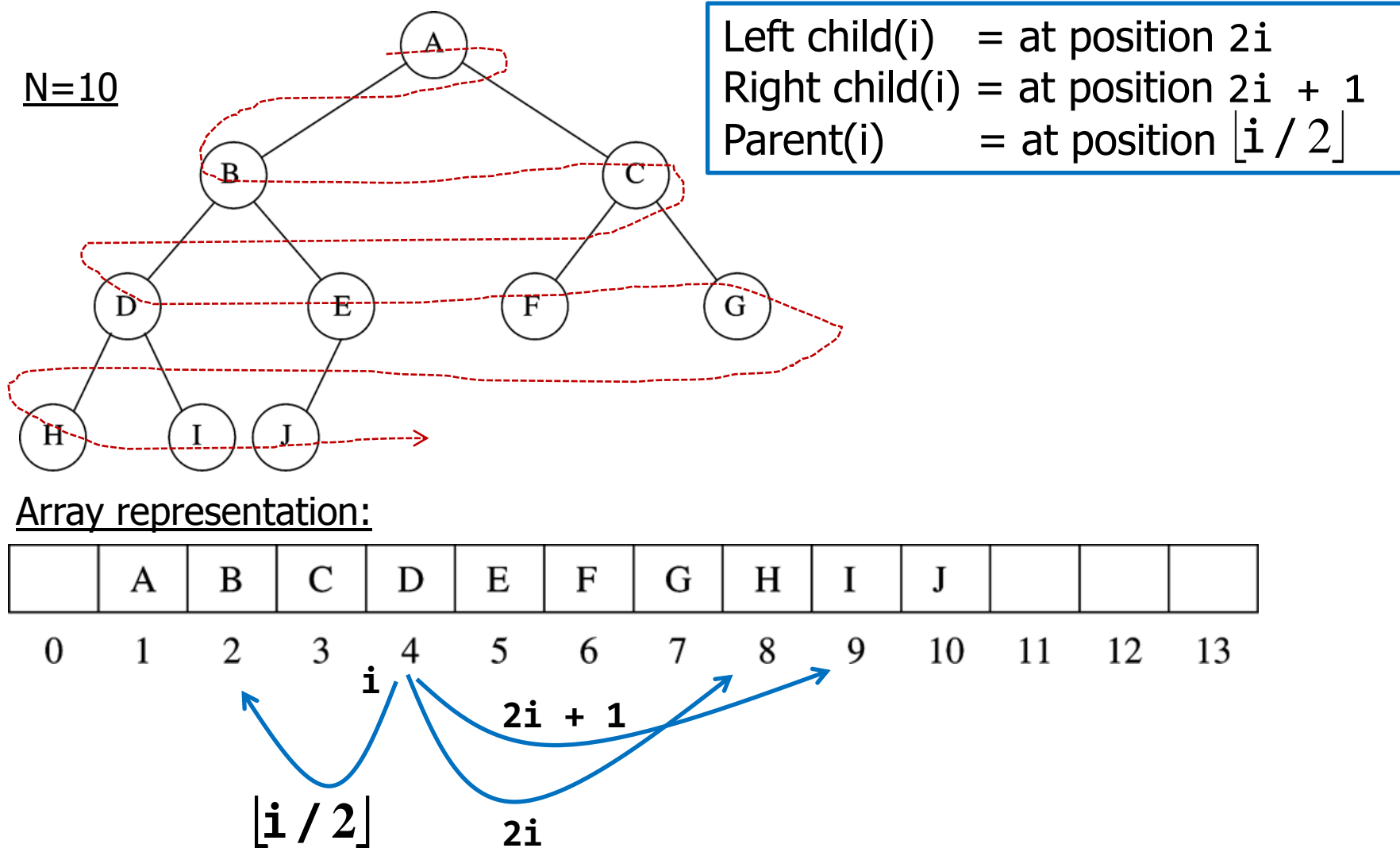


## deleteMin – Example

- Percolate 36 down swapping it with the smallest of its children
  - Halt when both children are larger

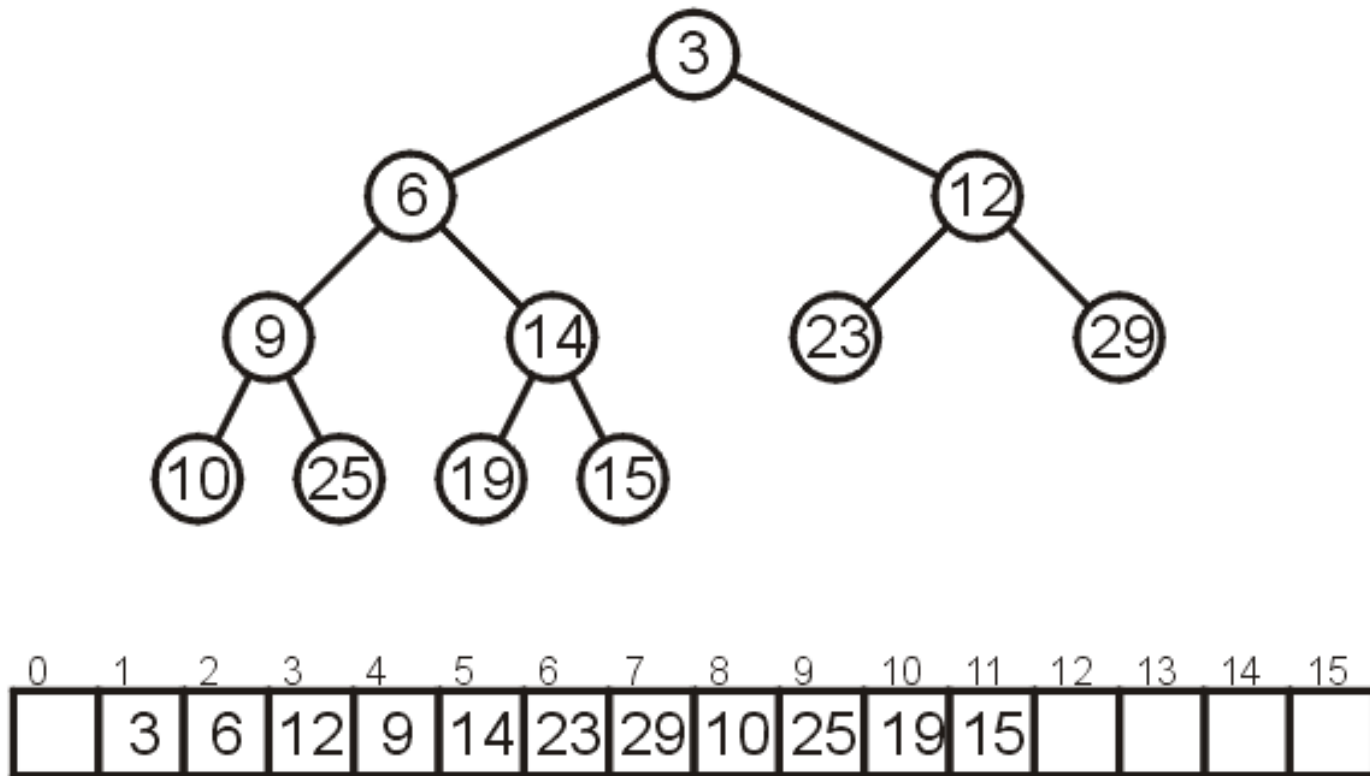


# Array-Based Implementation Of Binary Tree



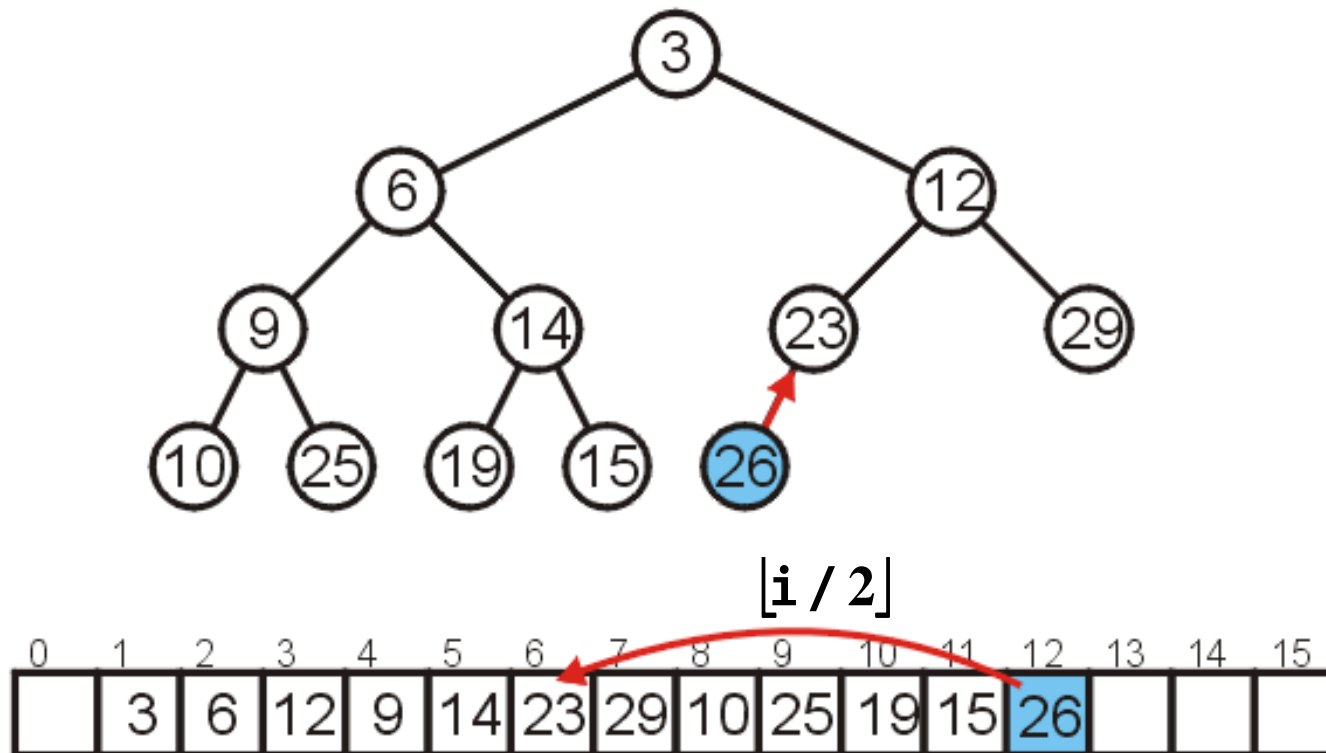
# Array-Based Implementation Of Binary Heap

- Consider the following heap, both as a tree and in its array representation



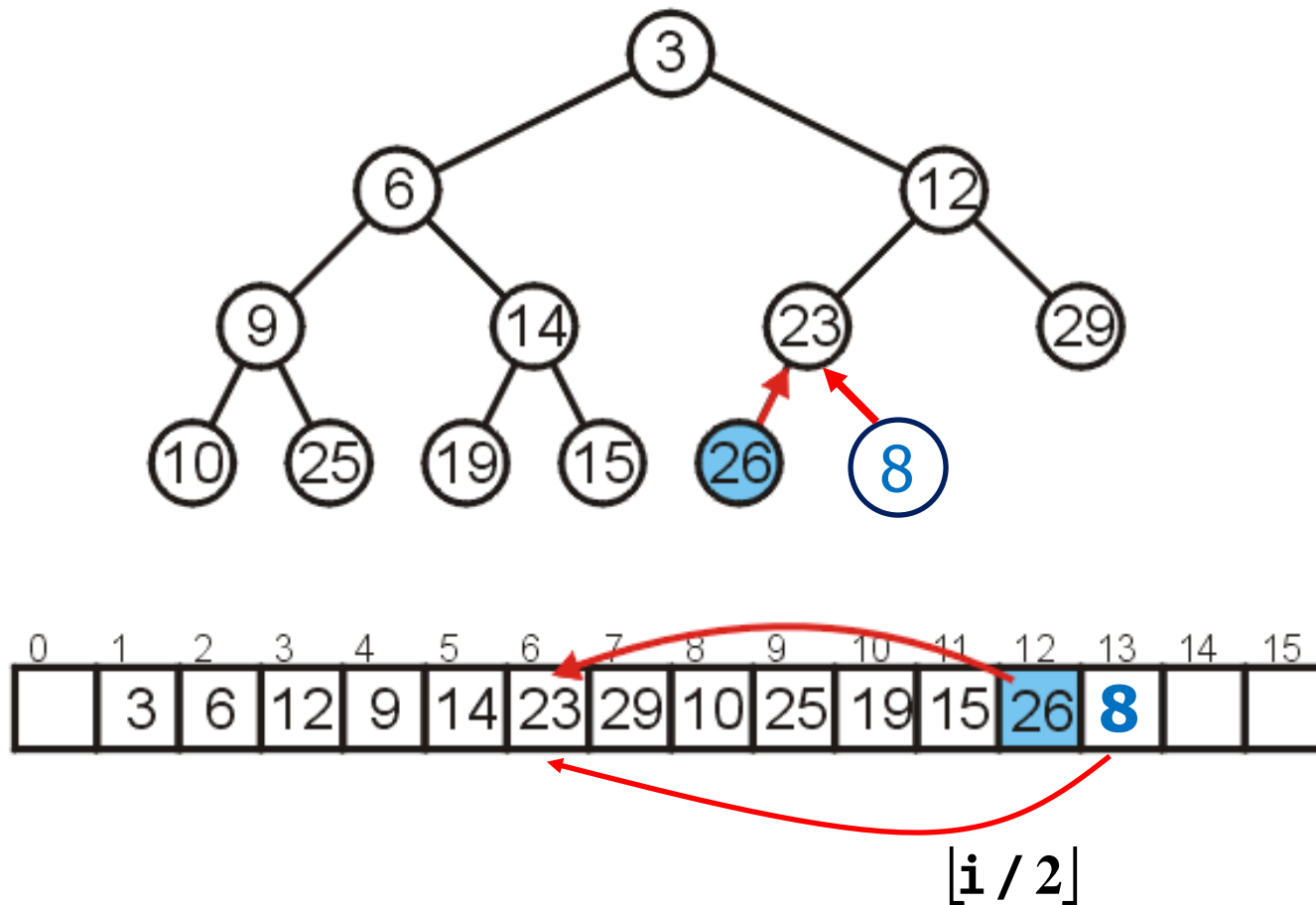
# Array-Based Implementation – insert

- Inserting 26 requires no changes



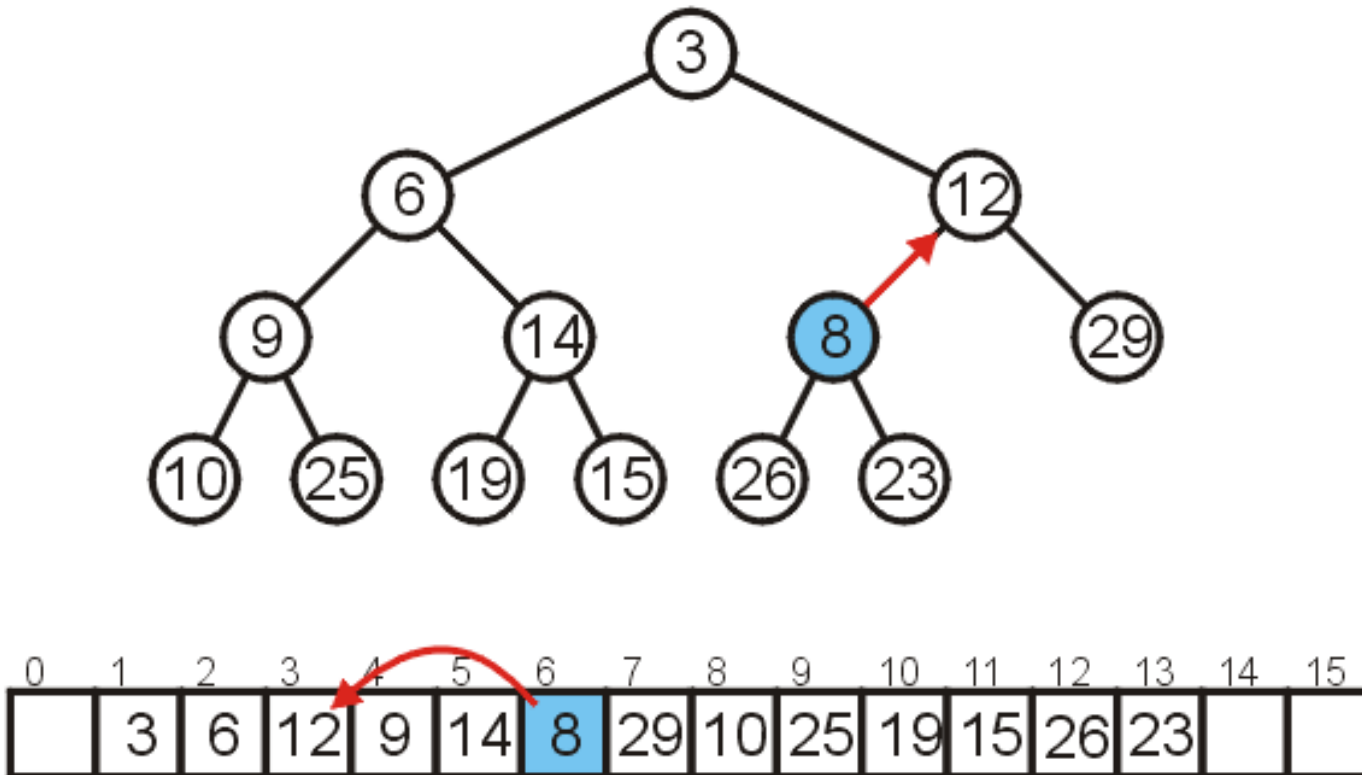
# Array-Based Implementation – insert

- Inserting 8 requires a few percolations
  - Swap 8 and 23



# Array-Based Implementation – insert

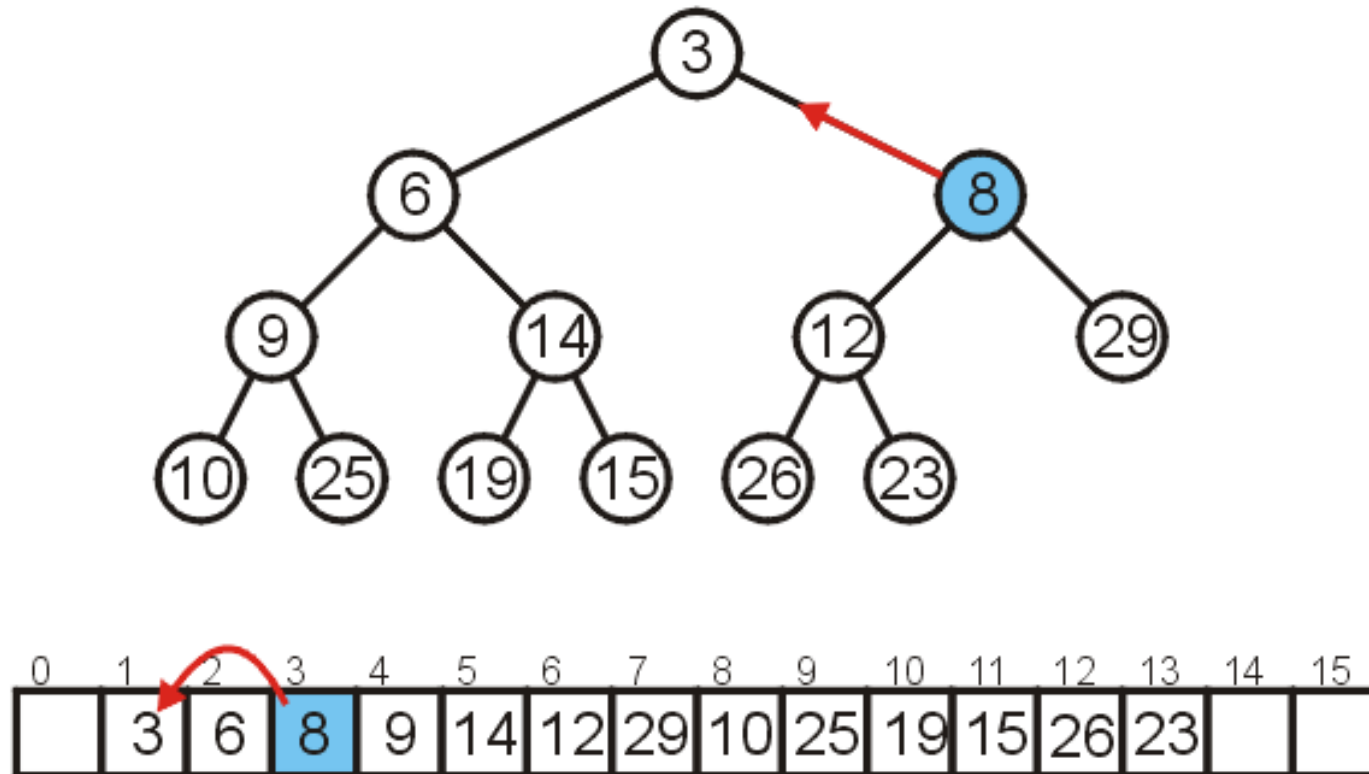
- Swap 8 and 12





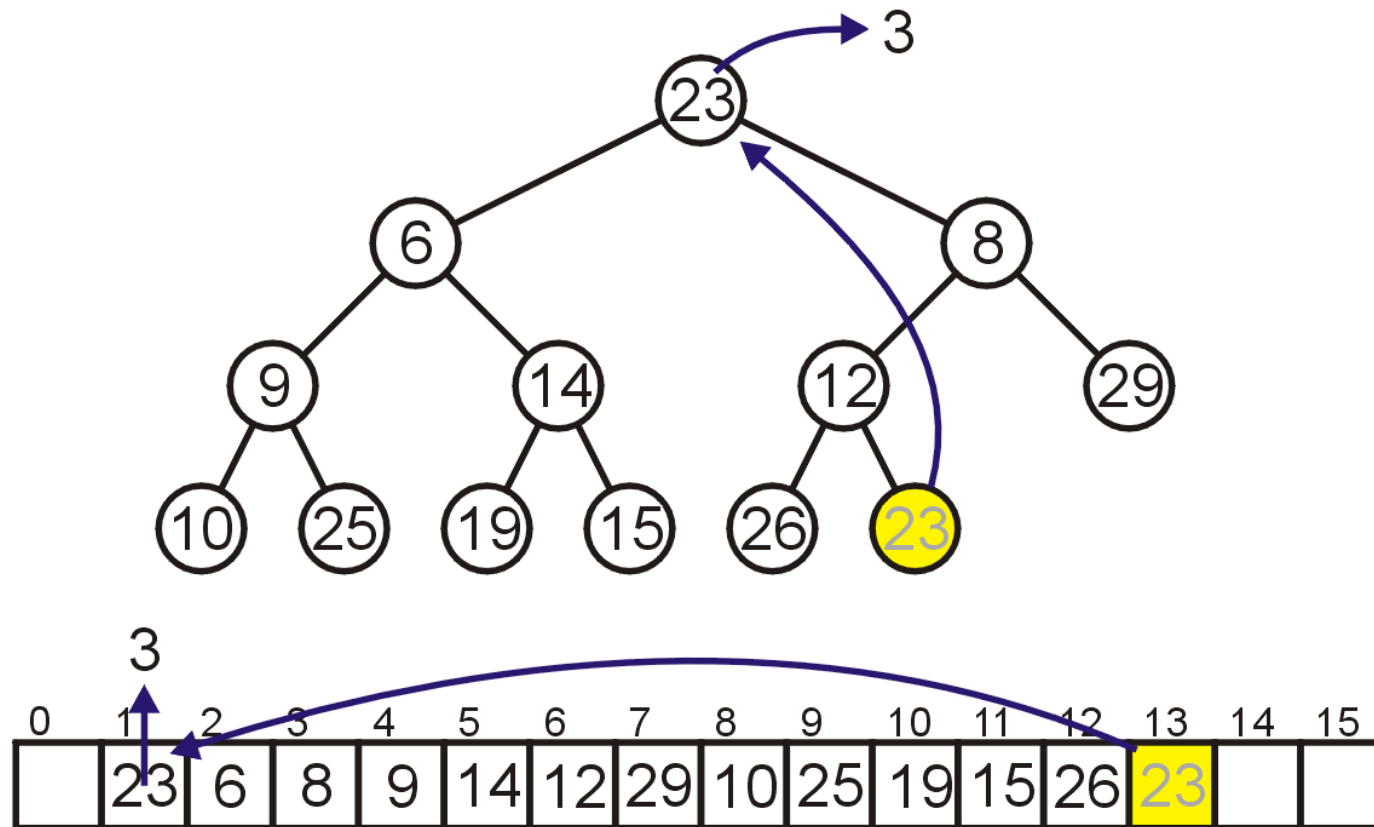
# Array-Based Implementation – insert

- At this point, 8 is greater than its parent, so we are finished



# Array-Based Implementation – deleteMin

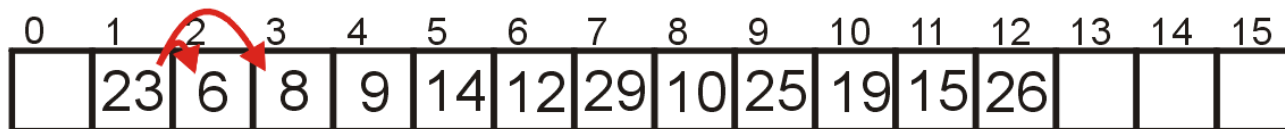
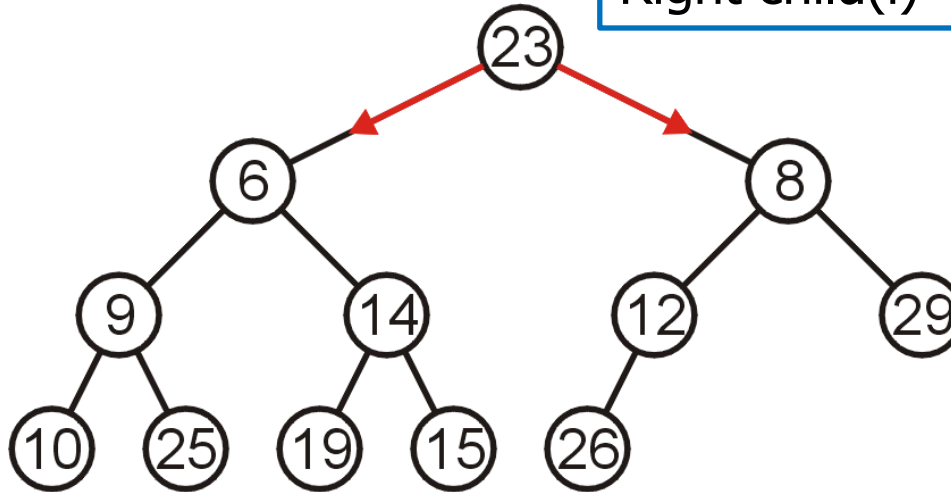
- Removing the top require copy of the last element to the top



# Array-Based Implementation – deleteMin

- Percolate down
  - Compare Node 1 with its children: Nodes 2 and 3
  - Swap 23 and 6

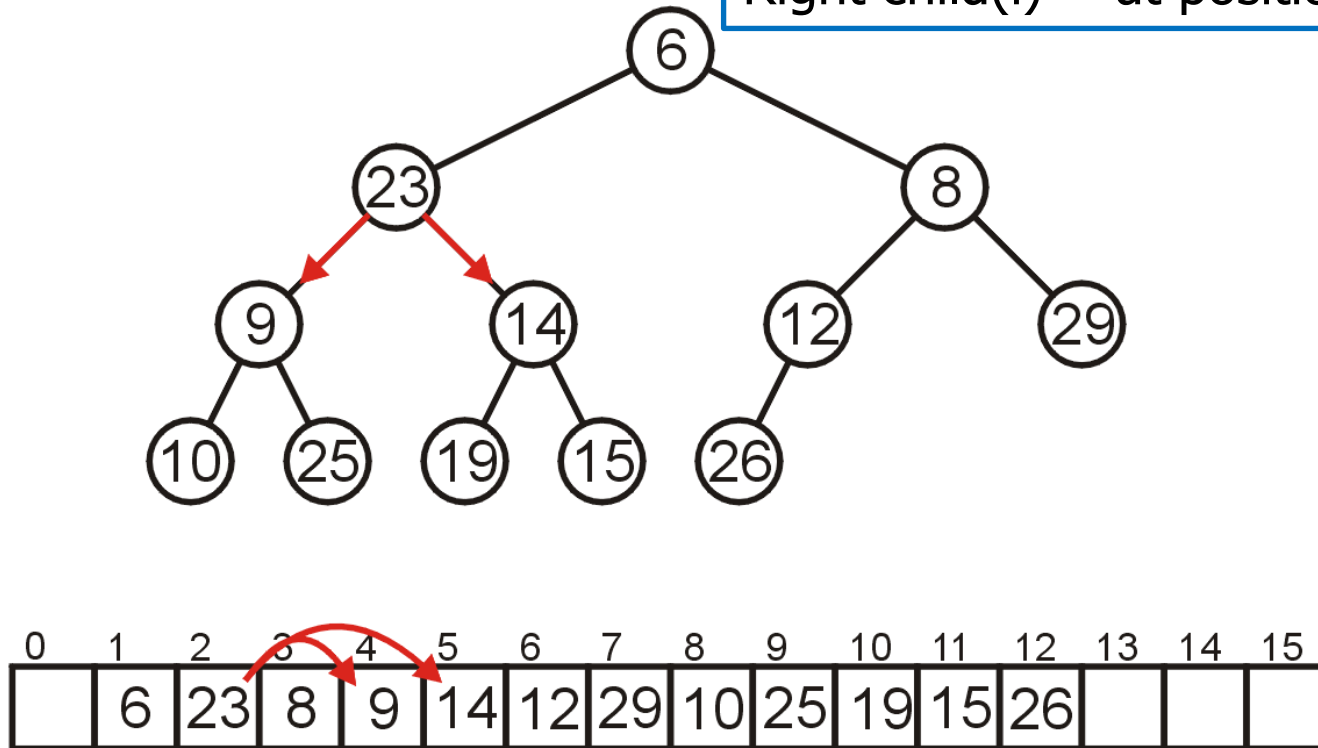
Left child(i) = at position  $2i$   
Right child(i) = at position  $2i + 1$



# Array-Based Implementation – deleteMin

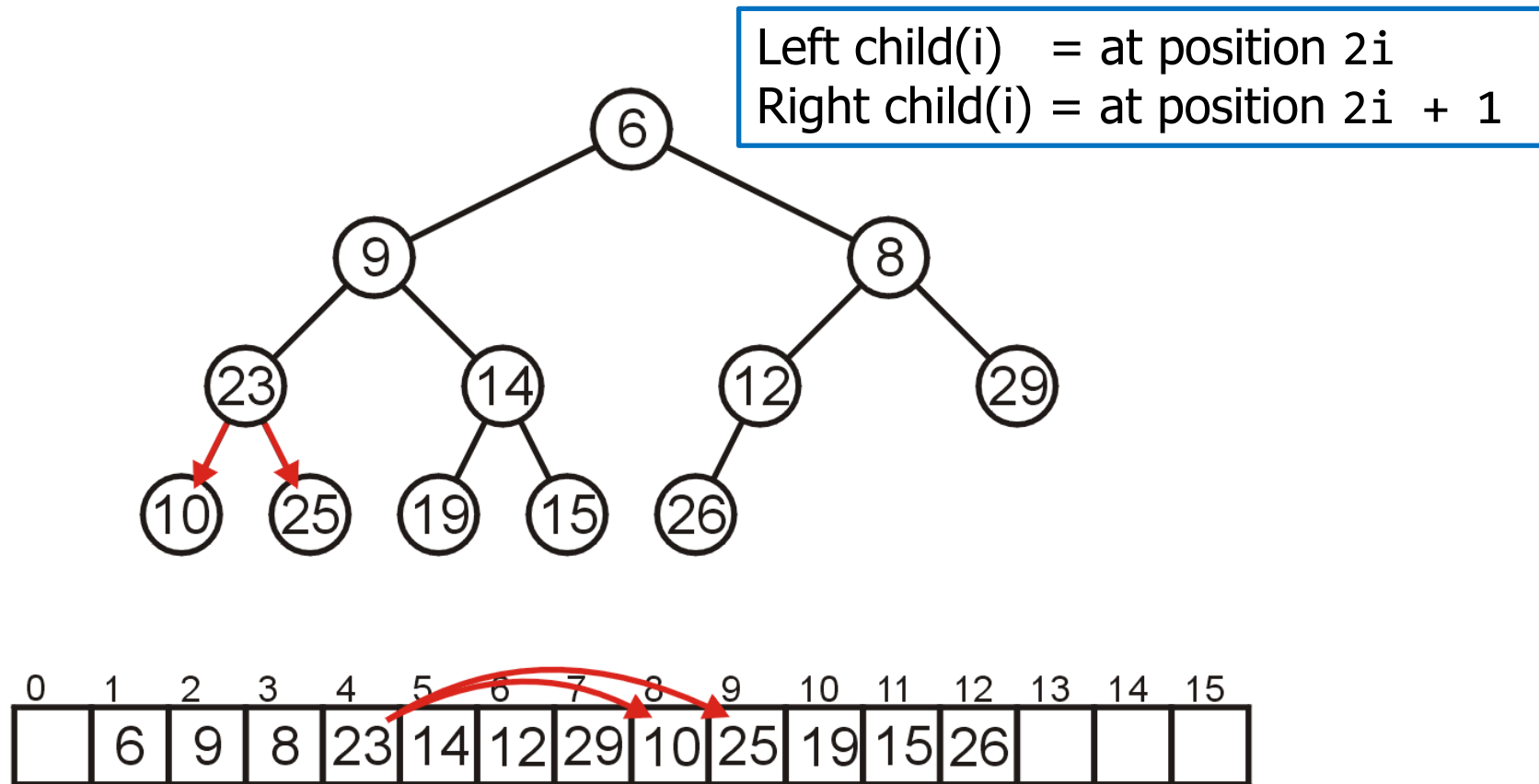
- Compare Node 2 with its children: Nodes 4 and 5
  - Swap 23 and 9

Left child(i) = at position  $2i$   
Right child(i) = at position  $2i + 1$



# Array-Based Implementation – deleteMin

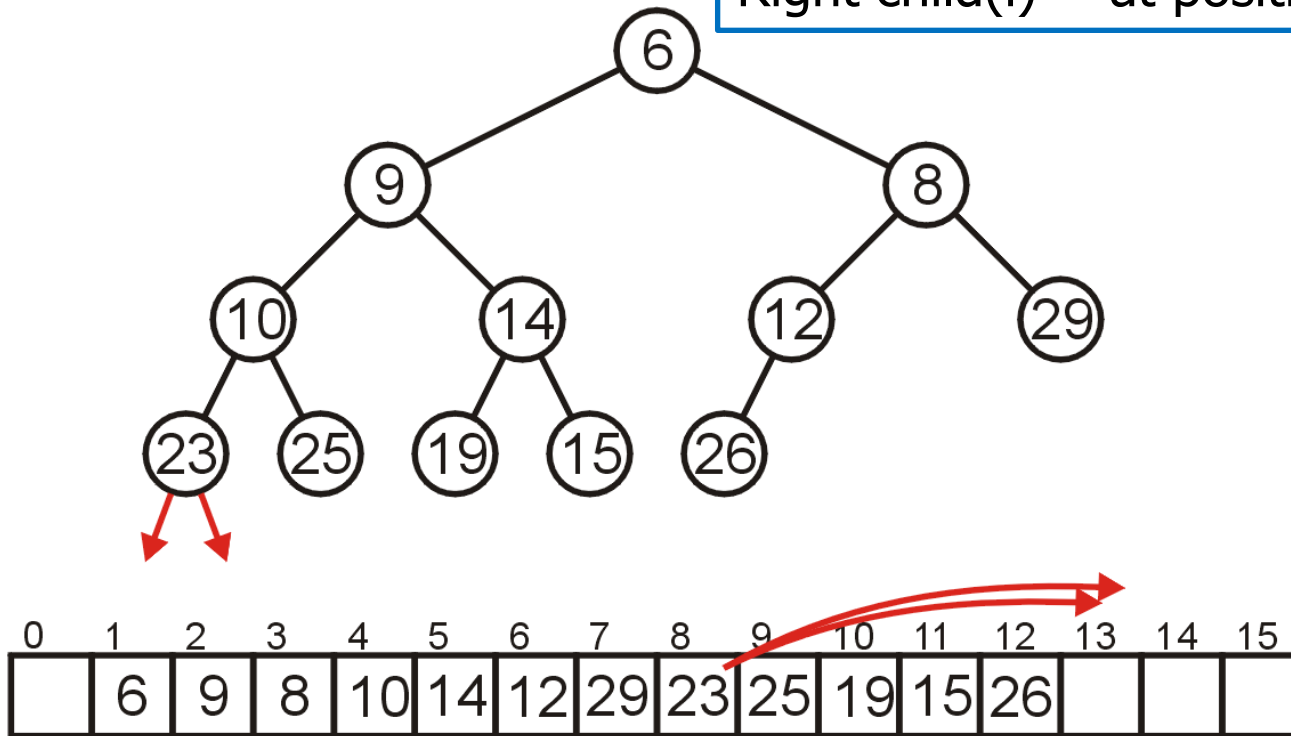
- Compare Node 4 with its children: Nodes 8 and 9
  - Swap 23 and 10



# Array-Based Implementation – deleteMin

- The children of Node 8 are beyond the end of the array:
  - Stop

Left child(i) = at position  $2i$   
Right child(i) = at position  $2i + 1$



# Runtime Analysis

---

- **insert** operation
  - Worst case: Inserting an element less than the root
    - $O(\log_2 n)$
  - Best case: Inserting an element greater than any other element
    - $O(1)$
  - Average case:  $O(1)$ 
    - Why ?
- **deleteMin** operation
  - Replacing the top element is  $O(1)$
  - Percolate down the top object is  $O(\log_2 n)$
  - We copy something that is already in the lowest depth
    - It will likely be moved back to the lowest depth

# Building a Heap

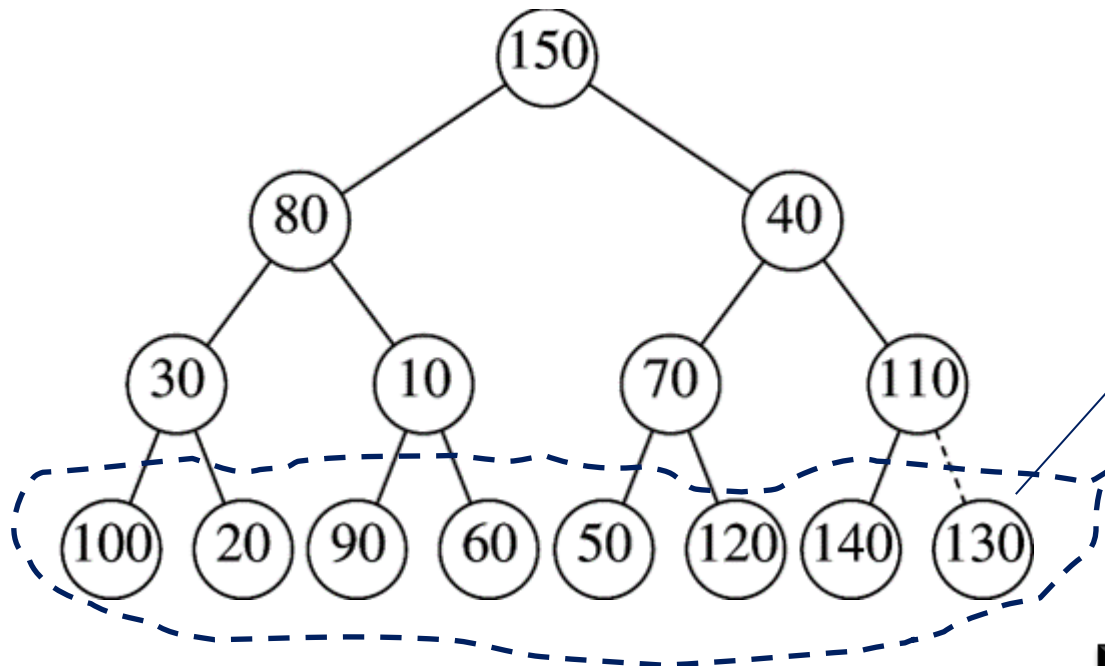
---

- What if all  $N$  elements are all available upfront?
  - Construct heap from initial set of  $N$  items
- Solution 1 (insert method)
  - Perform  $N$  inserts
- Solution 2 (BuildHeap method)
  - Randomly populate initial heap with structure property
  - Perform a percolate-down from each internal node
    - To take care of heap order property



# BuildHeap Example

- Input priority levels
  - { 150, 80, 40, 30, 10, 70, 110, 100, 20, 90, 60, 50, 120, 140, 130 }



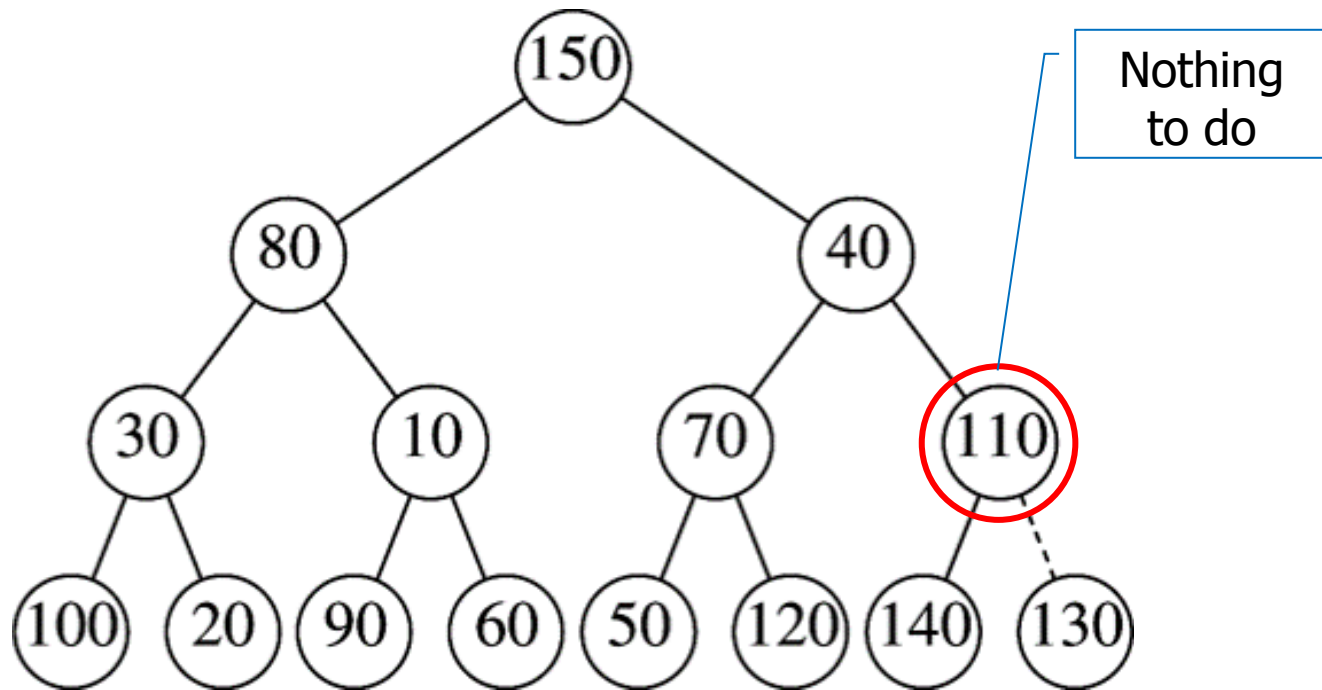
Leaves are all  
valid heaps  
(implicitly)

- Arbitrarily assign elements to heap nodes
- Structure property satisfied
- Heap order property violated
- Leaves are all valid heaps (implicit)

So, let us look at each  
internal node,  
from bottom to top,  
and fix if necessary

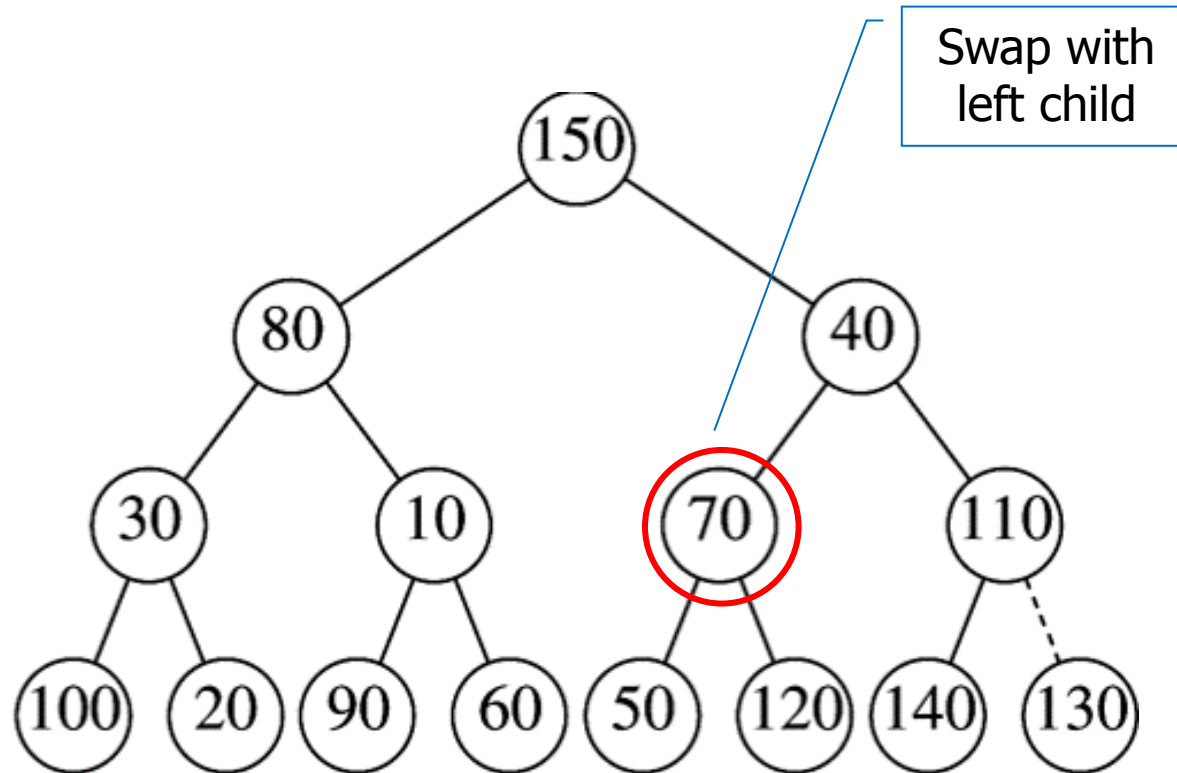
# BuildHeap Example

---



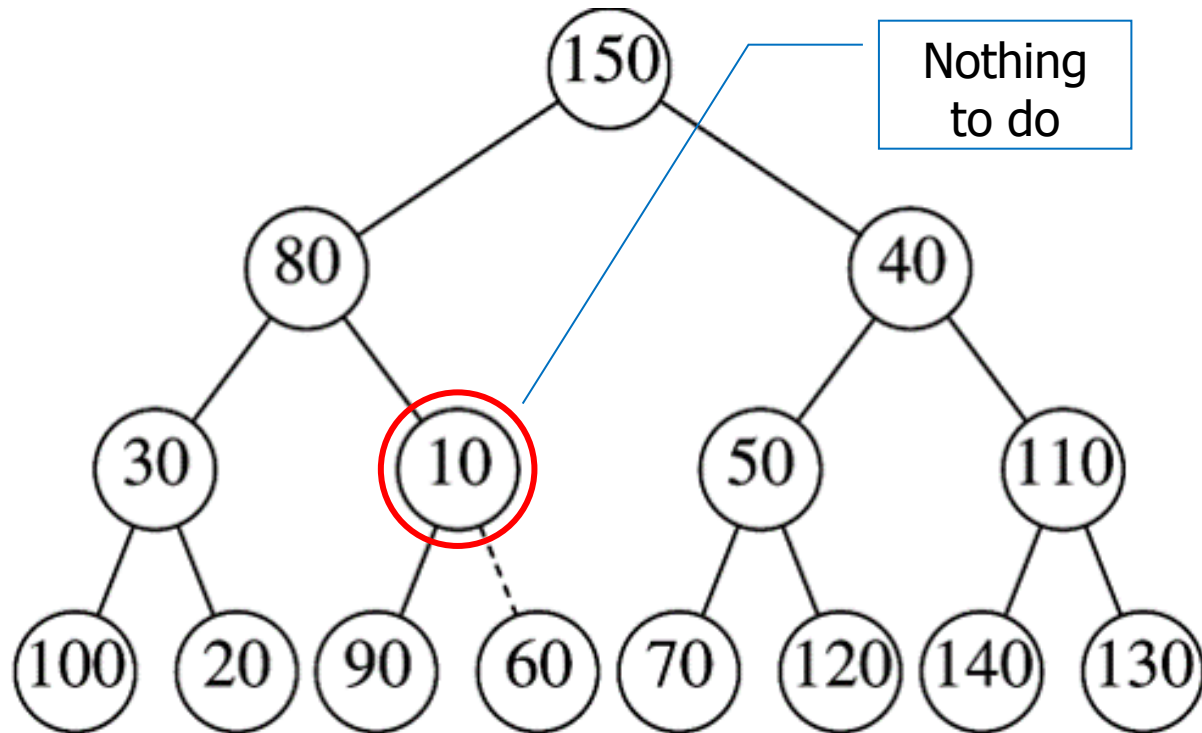
# BuildHeap Example

---



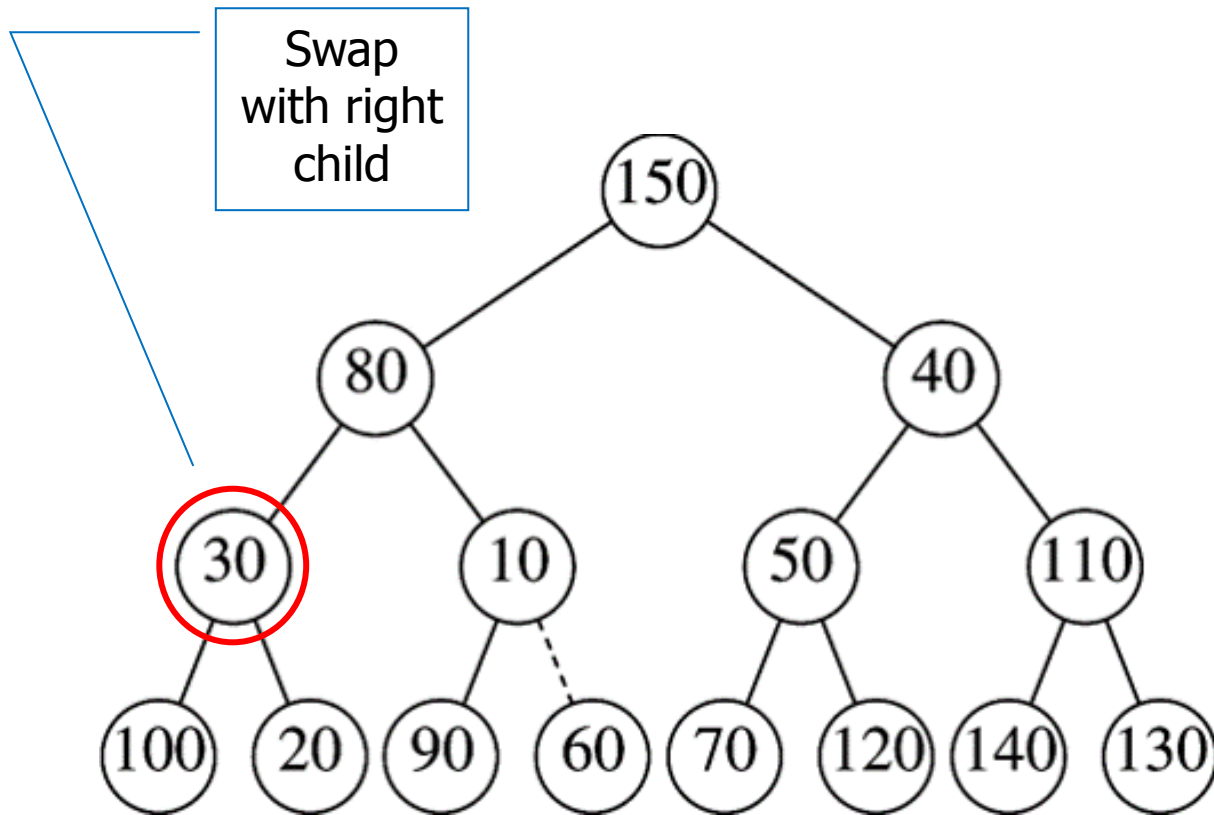
# BuildHeap Example

---



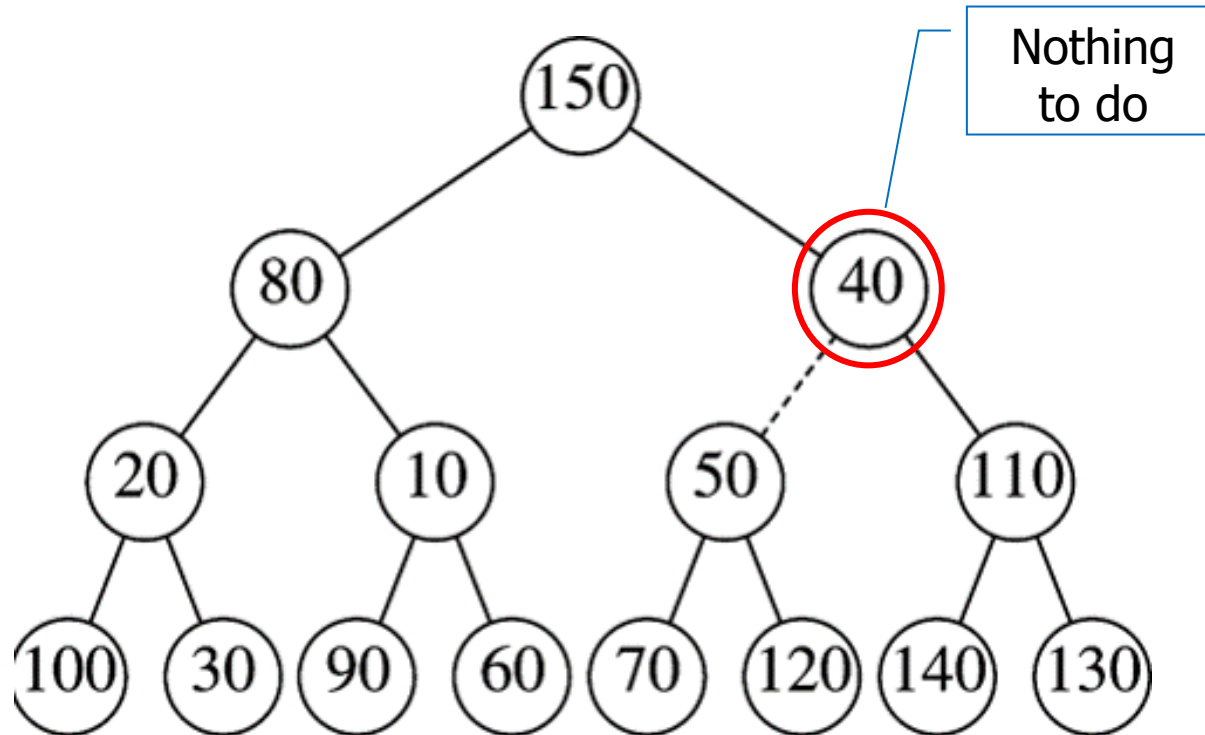
# BuildHeap Example

---

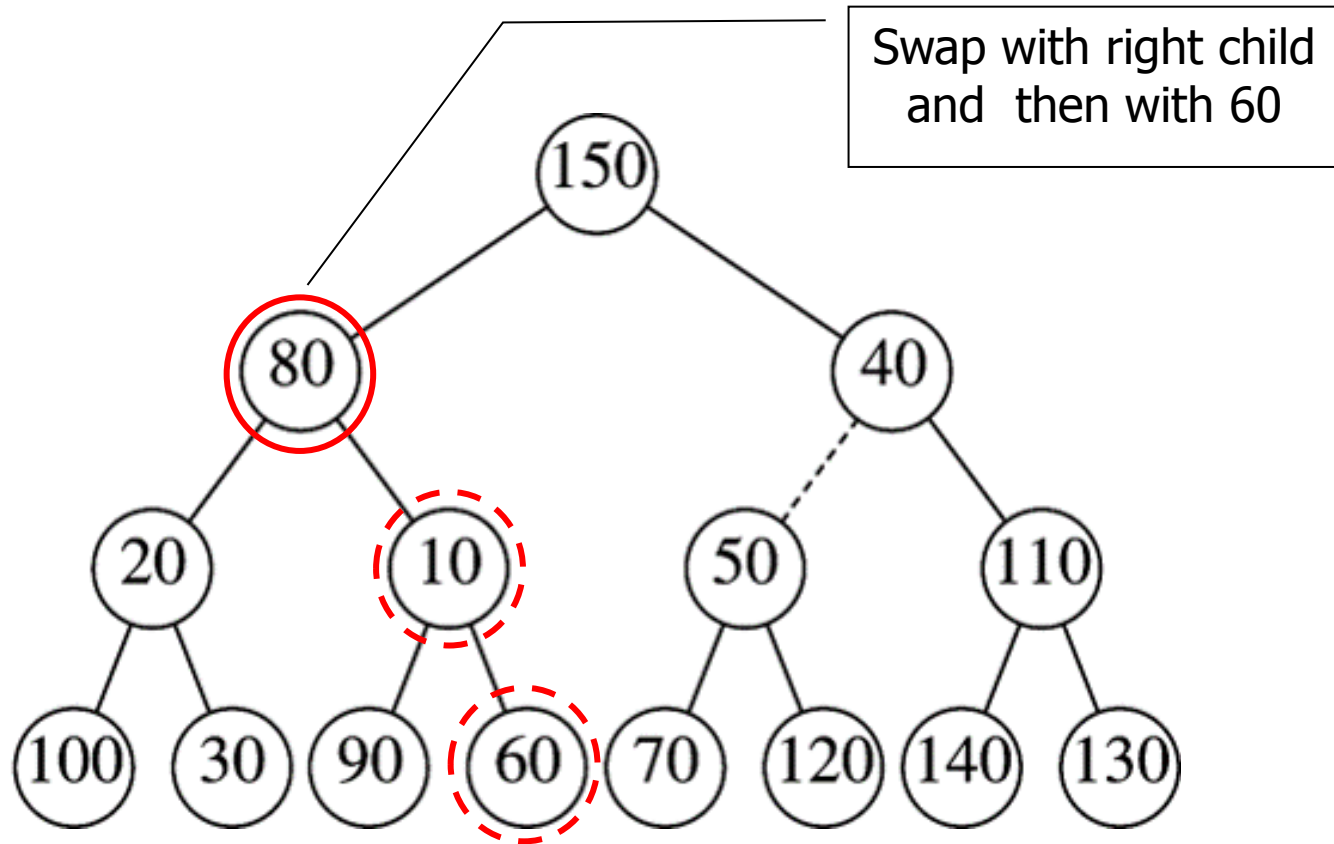


# BuildHeap Example

---

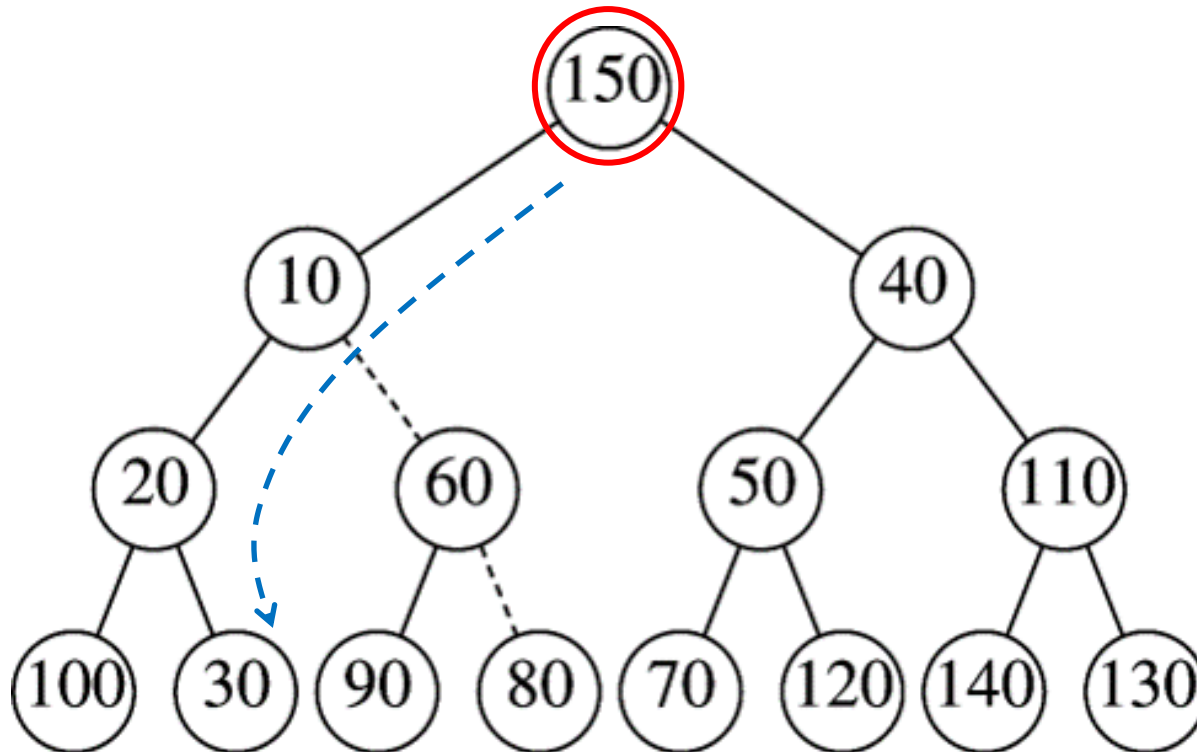


# BuildHeap Example



# BuildHeap Example

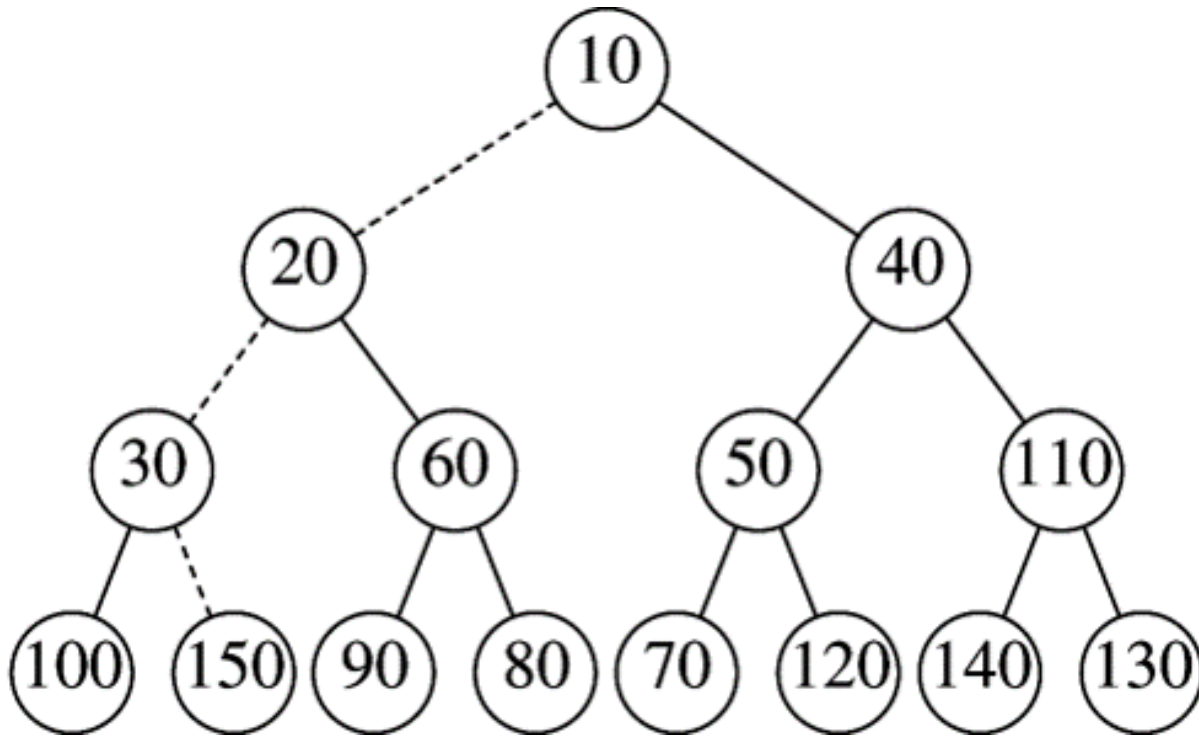
---





# BuildHeap Example

---



Final Heap

# Any Question So Far?

---

