National University of Computer & Emerging Sciences

Priority Queues AKA Heaps

Motivation

- Queues are a standard mechanism for ordering tasks on a first-come, first-served basis
- However, some tasks may be more important or timely than others (higher priority)
- Priority queues
  - Store tasks using a partial ordering based on priority
  - Ensure highest priority task at head of queue
- Heaps are the underlying data structure of priority queues

Applications of the 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

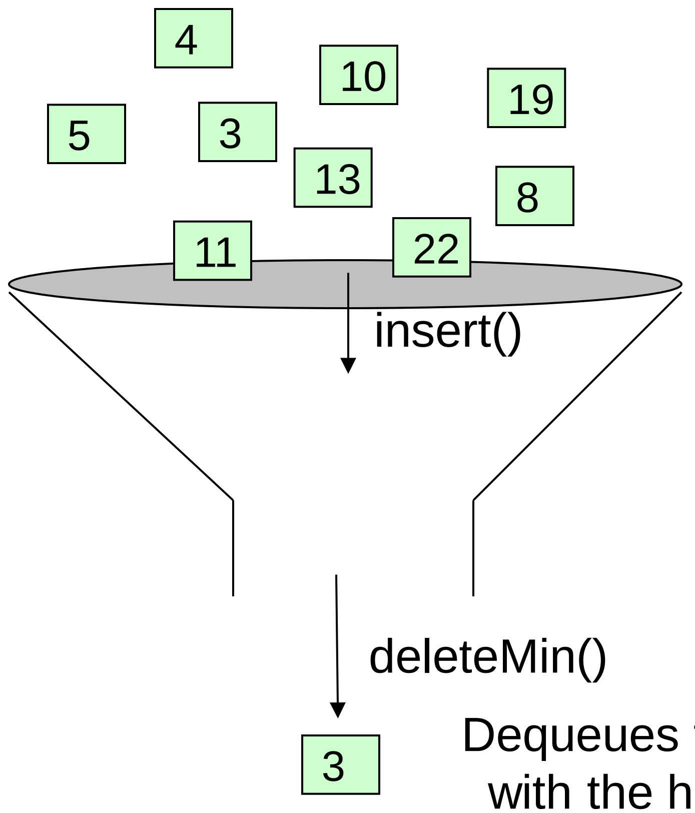
- Anything *greedy*

Priority Queues: Specification

- Main operations
  - **`insert`** (i.e., enqueue)
    - Dynamic insert
    - specification of a priority level (0-high, 1,2.. Low)
  - **`deleteMin`** (i.e., dequeue)
    - Finds the current minimum element (read: "highest priority") in the queue, deletes it from the queue, and returns it

- Performance goal is for operations to be "fast"

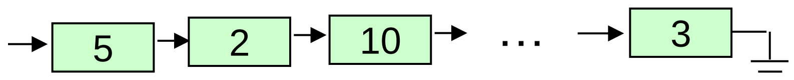
Using priority queues

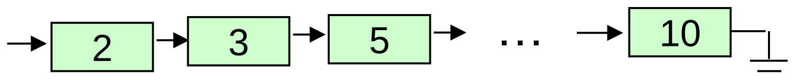


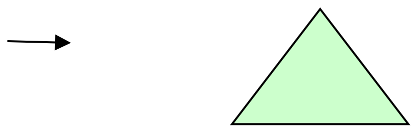
Dequeues the next element with the highest priority

Simple Implementations

- Unordered linked list
  - Insert in one step
  - deleteMin in n steps
- Ordered linked list
  - Insert in n steps
  - deleteMin in one step
- Balanced BT







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

Binary Heap

A priority queue data structure

Binary Heap

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

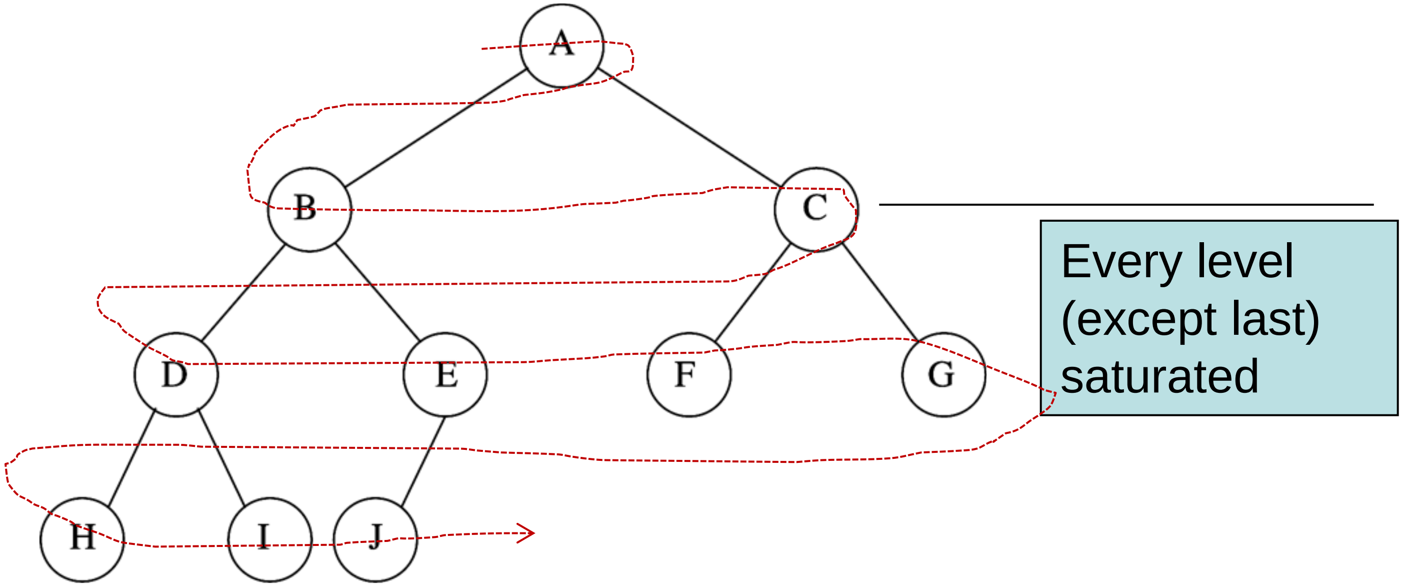
Structure Property

- A binary heap is a **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)

Binary Heap Example

Structure property

N=10



Every level (except last) saturated

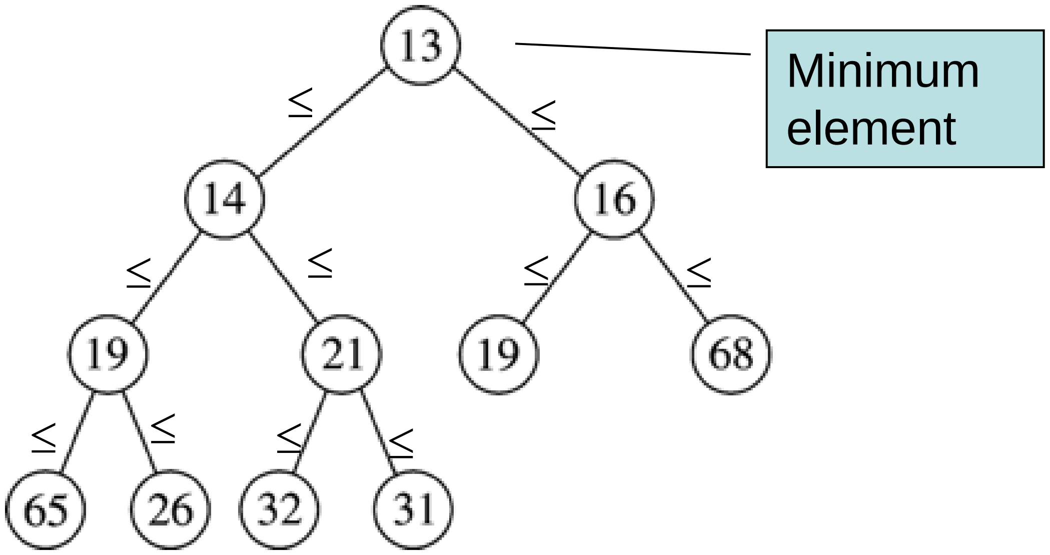
Array representation:

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap-order Property

- Heap-order property (for a "MinHeap")
  - For every node X, key(parent(X)) ≤ key(X)
  - Except root node, which has no parent
- Thus, minimum key always at root
  - Alternatively, for a "MaxHeap", always keep the maximum key at the root
- Insert and deleteMin must maintain heap-order property

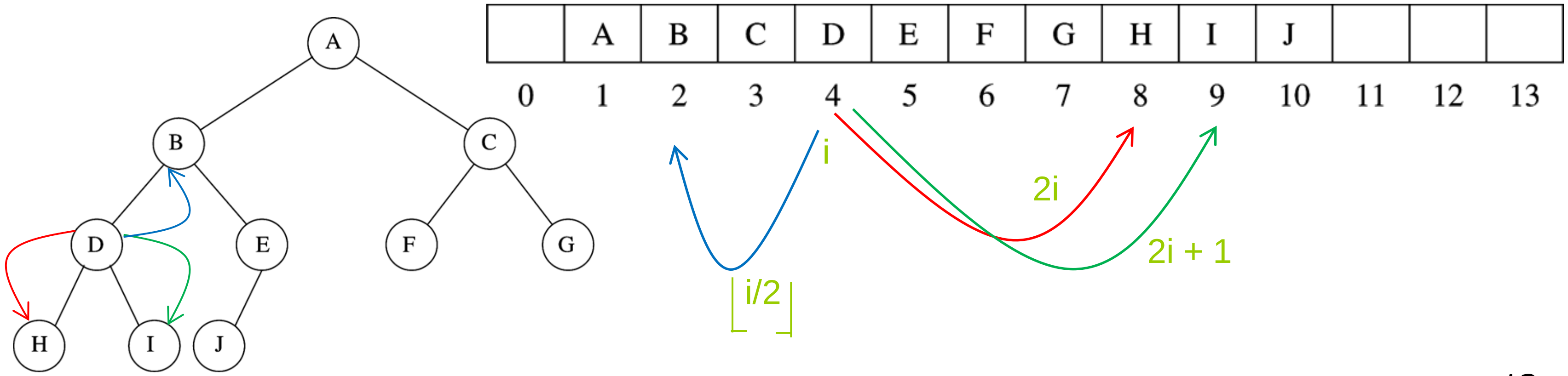
Heap Order Property



Implementing Complete Binary Trees as Arrays

- Given element at position i in the array
  - Left child(i) = at position 2i
  - Right child(i) = at position 2i + 1
  - Parent(i) = at position

$$\lfloor i/2 \rfloor$$



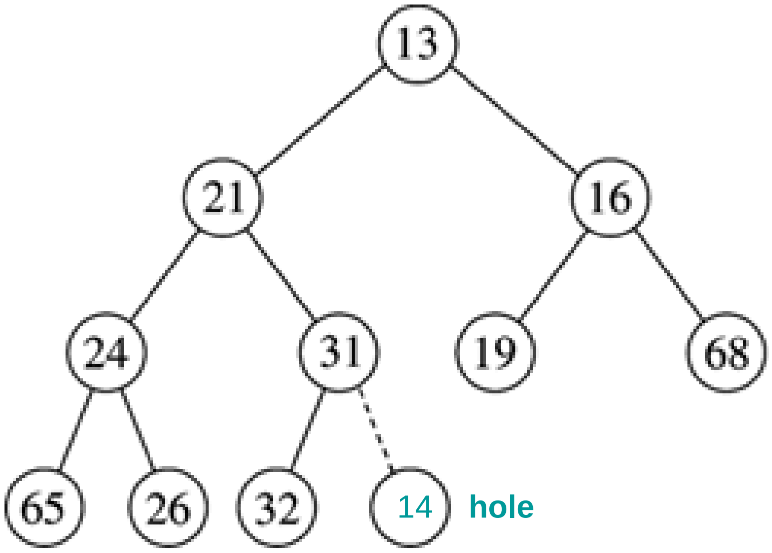
Heap Insert

- Insert new element into the heap at the next available slot ("hole")
  - According to maintaining a complete binary tree
- Then, "percolate" the element up the heap while heap-order property not satisfied

Heap Insert: Example

Percolating Up

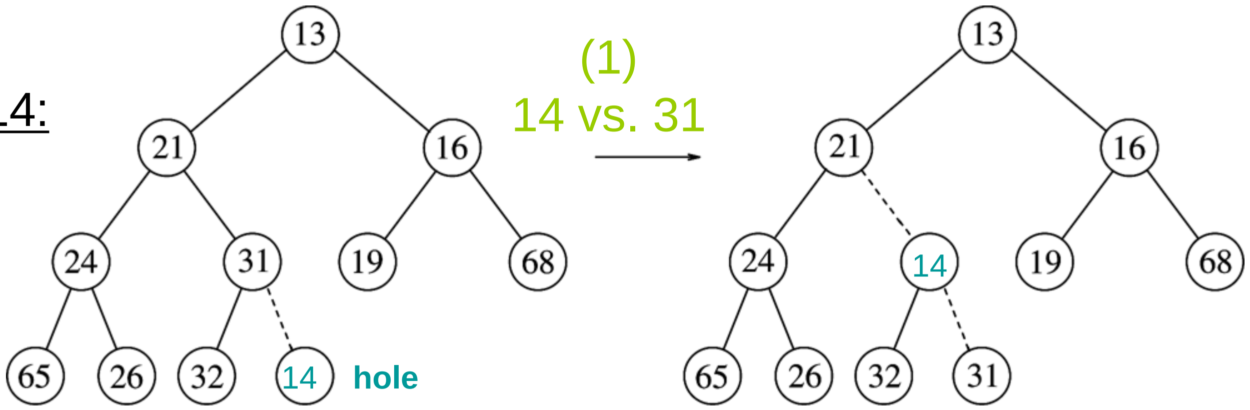
Insert 14:



Heap Insert: Example

Percolating Up

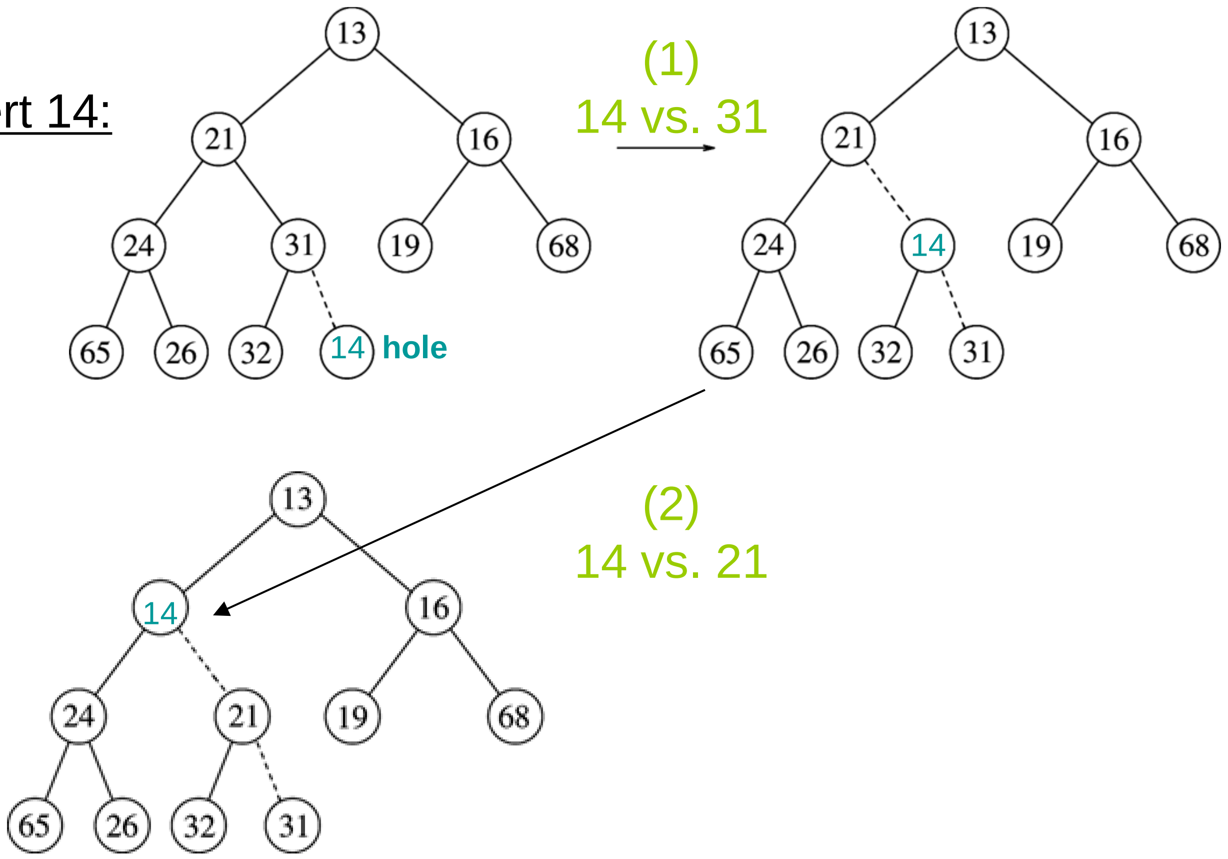
Insert 14:



Heap Insert: Example

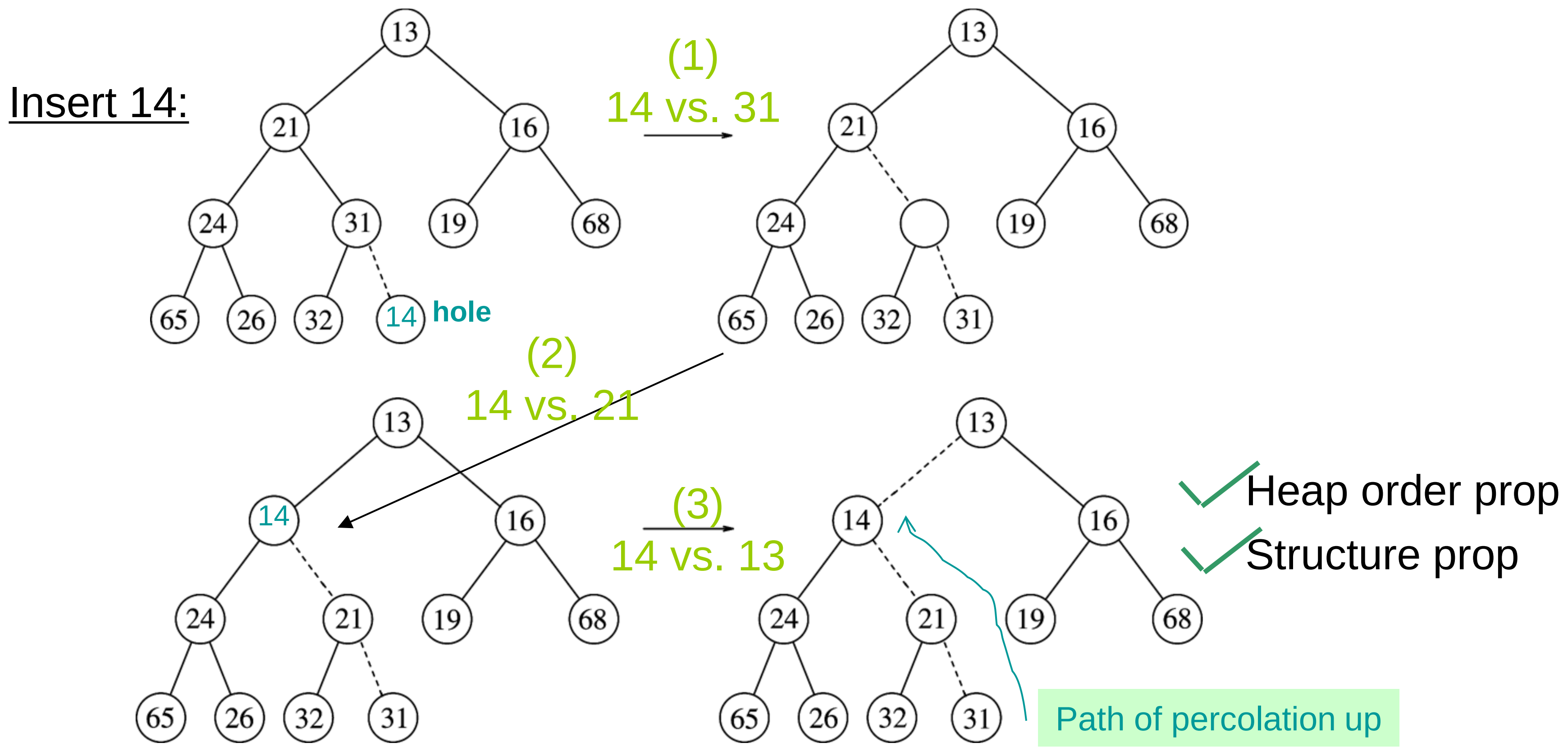
Percolating Up

Insert 14:



Heap Insert: Example

Percolating Up

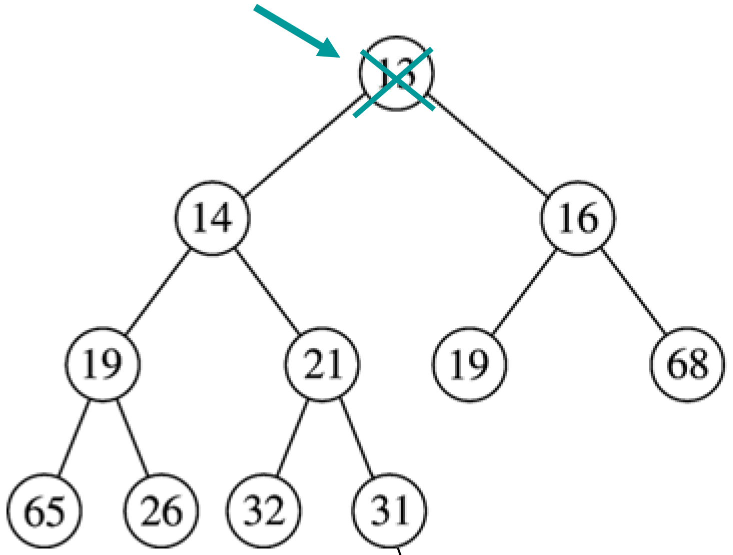


Heap DeleteMin

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

Heap DeleteMin: Example

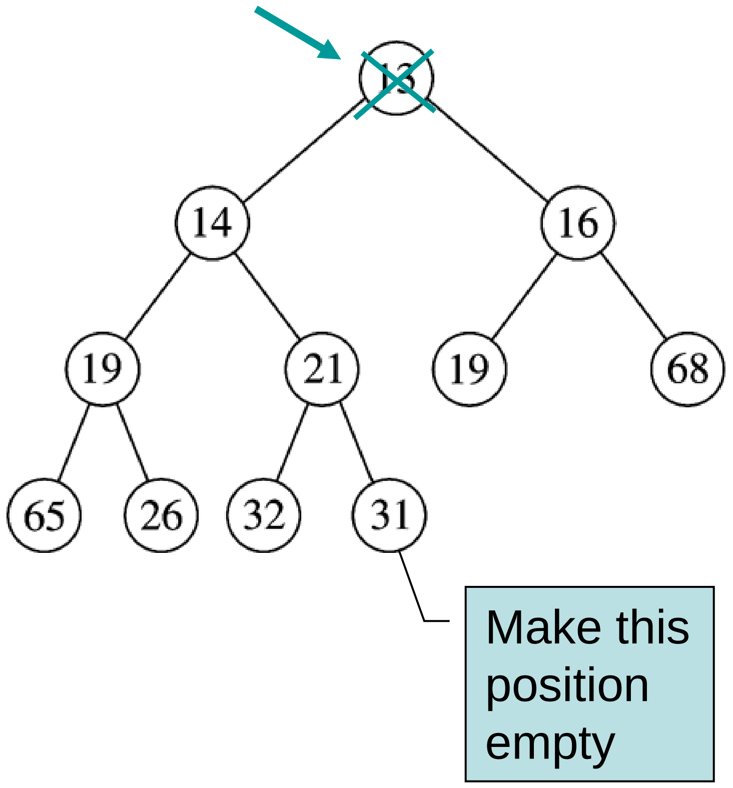
Percolating down…



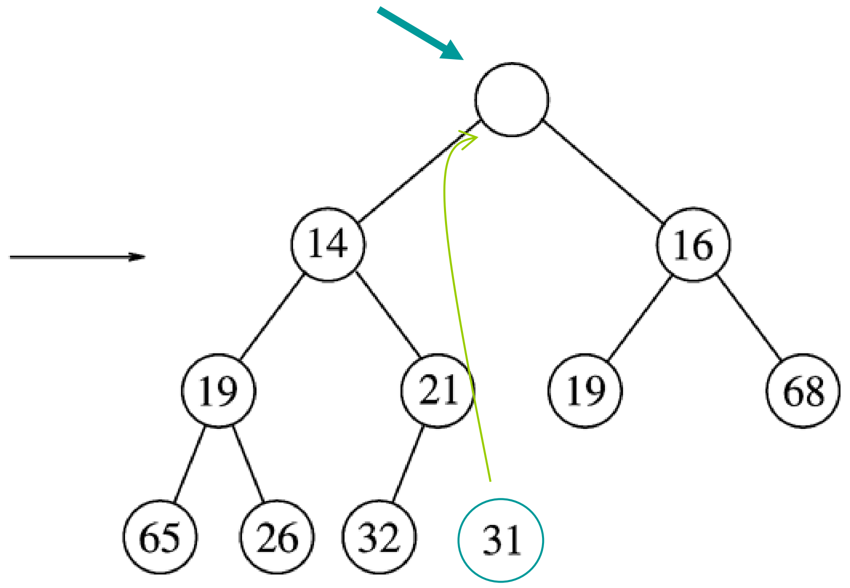
Make this position empty

Heap DeleteMin: Example

Percolating down…



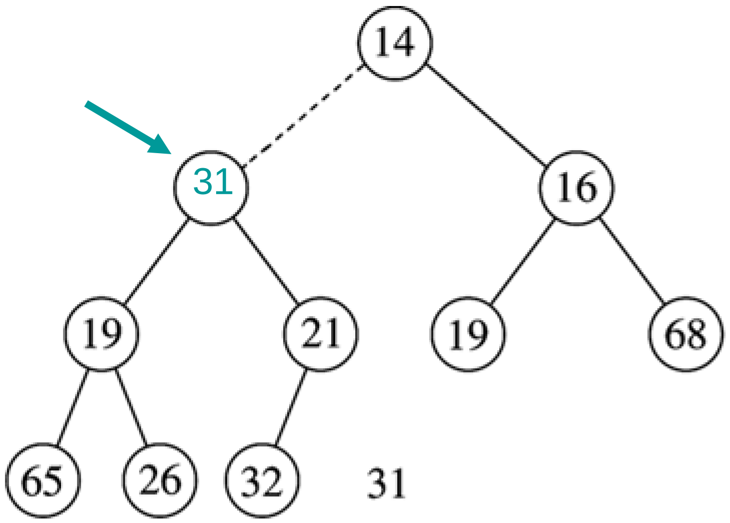
Copy 31 temporarily here and move it down



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

Heap DeleteMin: Example

Percolating down…

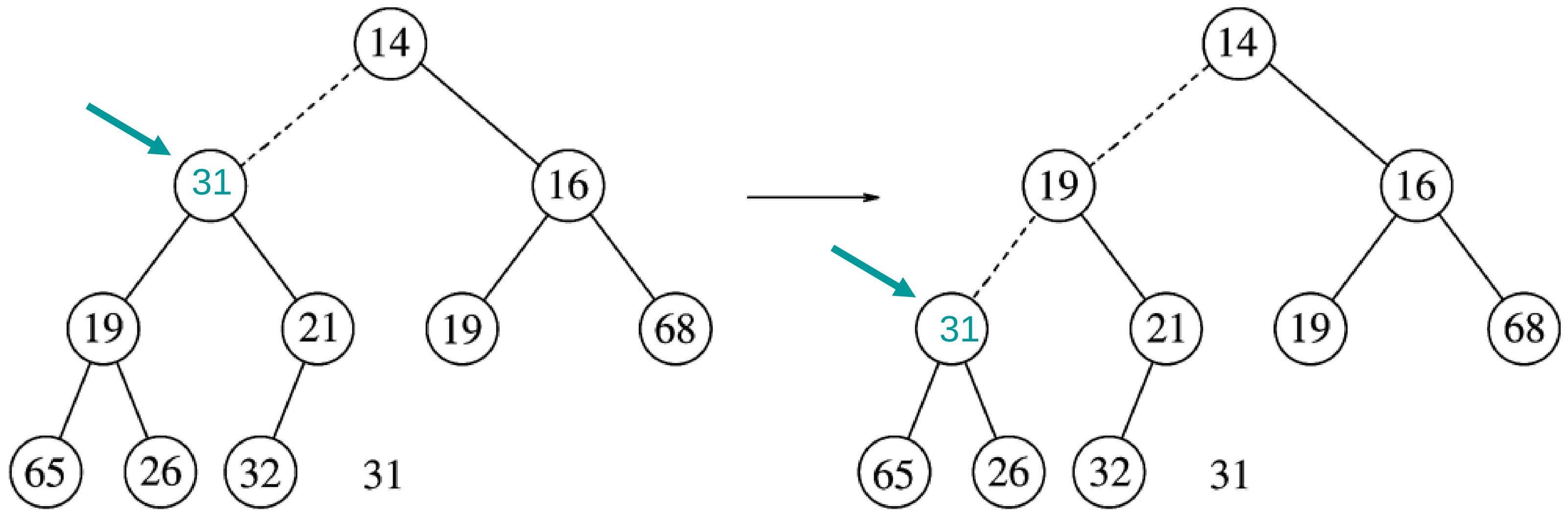


Is 31 > min(19,21)?

•Yes - swap 31 with min(19,21)

Heap DeleteMin: Example

Percolating down…



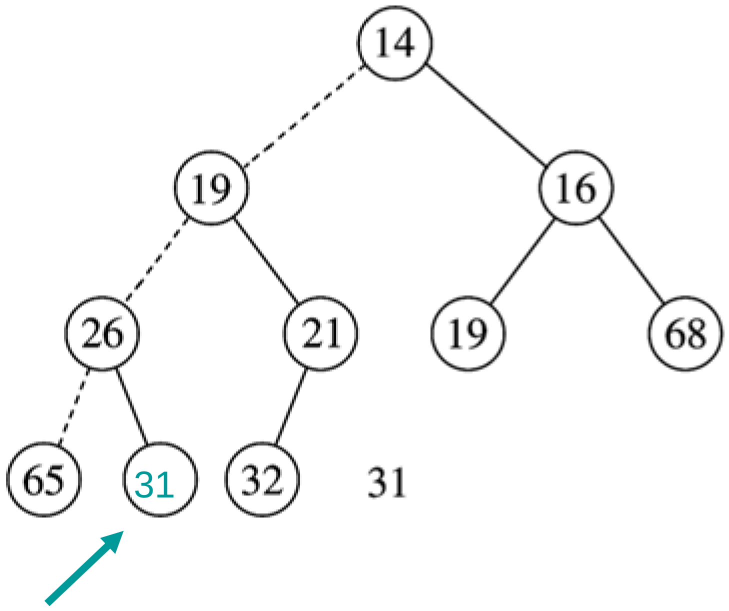
Is 31 > min(19,21)?
•Yes - swap 31 with min(19,21)

Is 31 > min(65,26)?
•Yes - swap 31 with min(65,26)

Percolating down…

Heap DeleteMin: Example

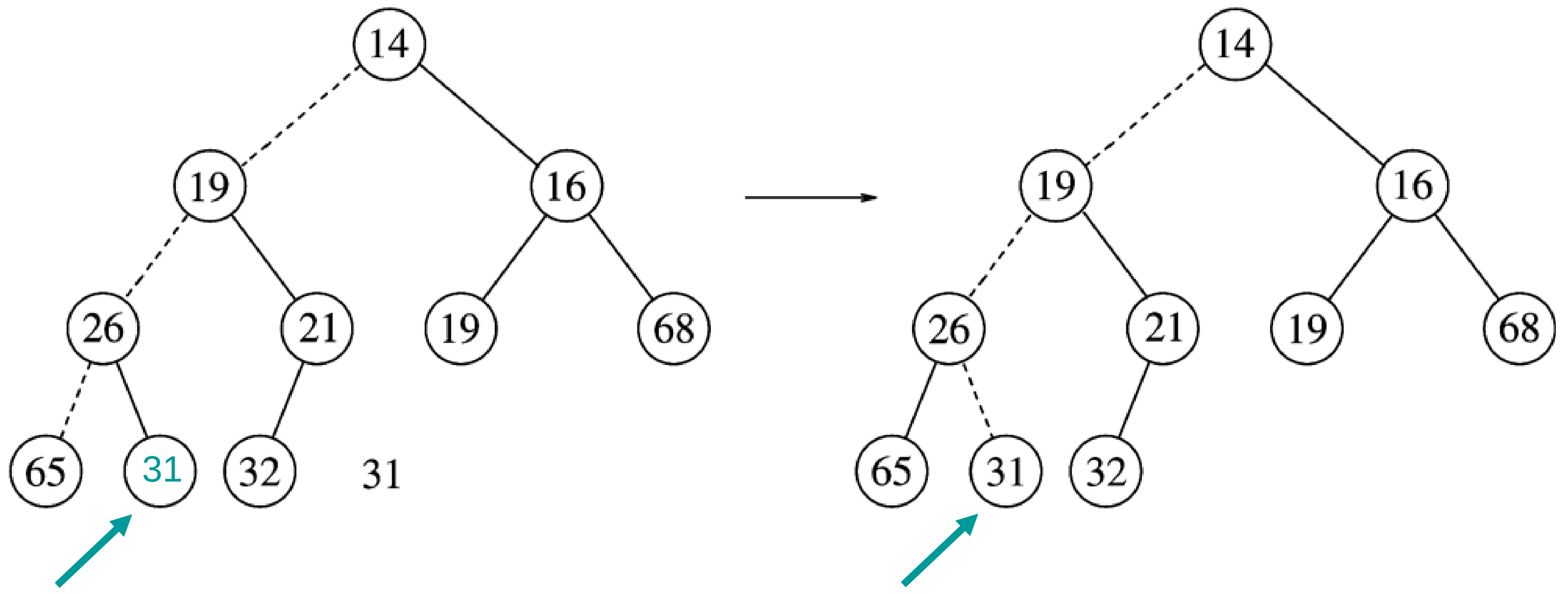
Percolating down…



Percolating down…

Heap DeleteMin: Example

Percolating down…



✓Heap order prop

✓Structure prop

Improving Heap Insert Time

- What if all N elements are all available upfront?

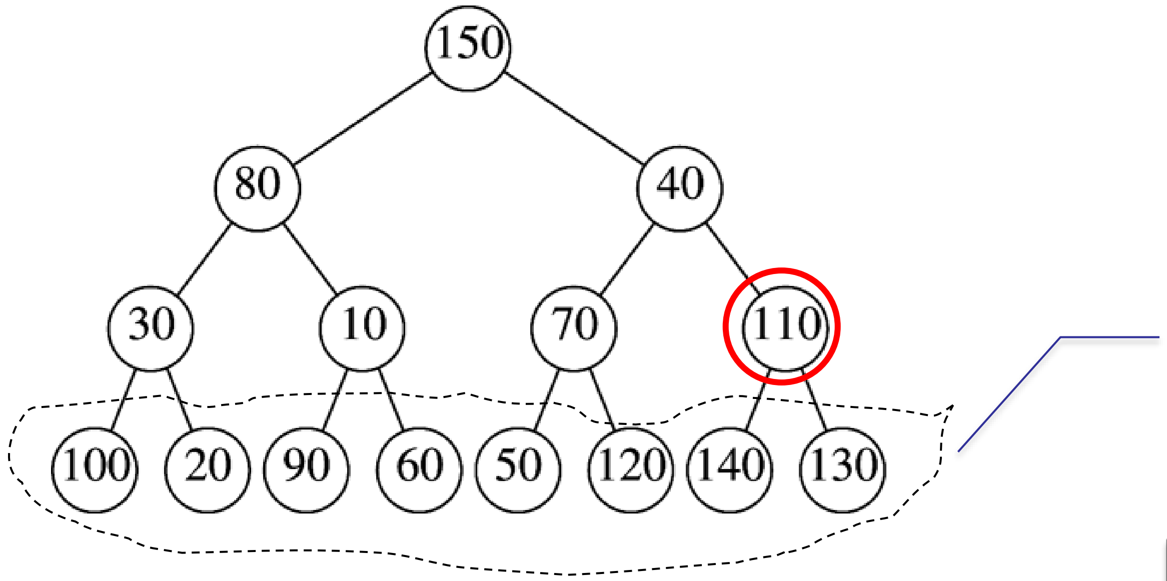
- To build a heap with N elements:
  - Default method takes more time
  - We will now see a new method

Building a Heap

- Construct heap from initial set of N items
- Solution 1
  - Perform N inserts
- Solution 2
  - 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: { 150, 80, 40, 30, 10, 70, 110, 100, 20, 90, 60, 50, 120, 140, 130 }

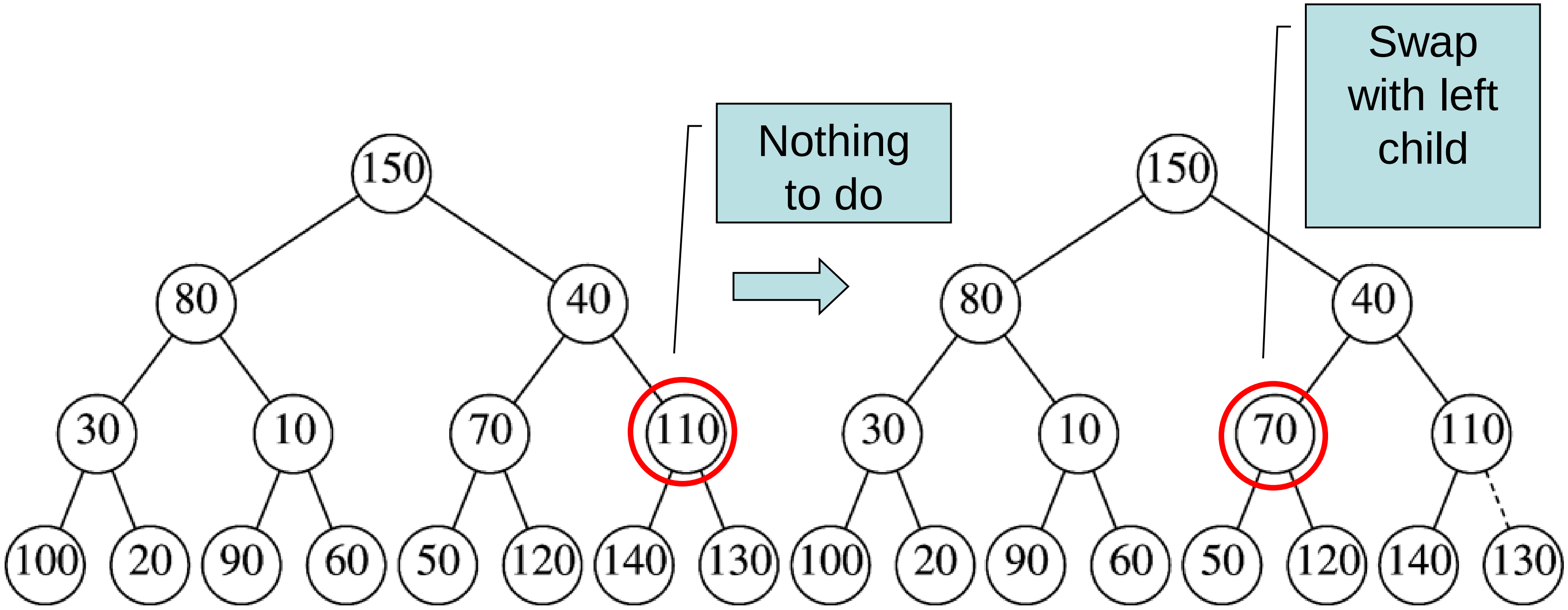


Leaves are all valid heaps (implicitly)

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

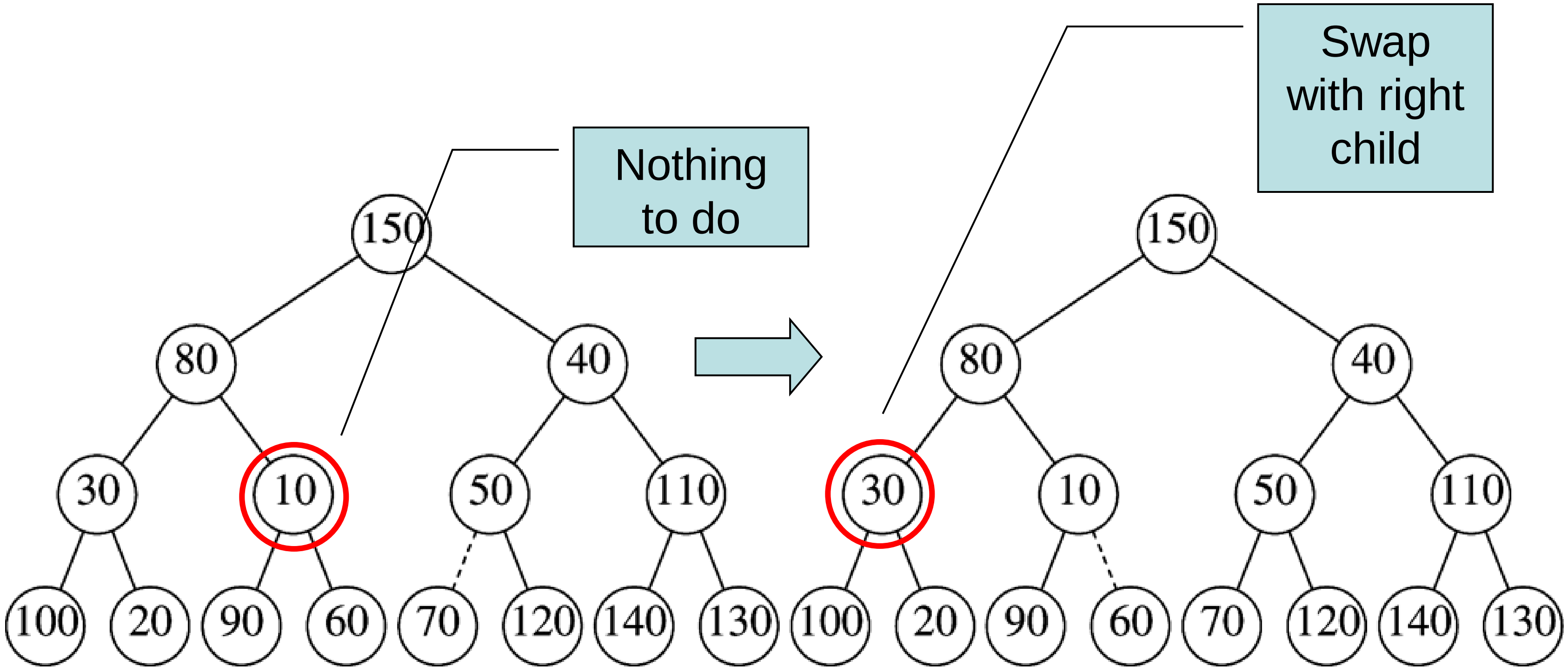
So, let us look at each internal node, from bottom to top, and fix if necessary

BuildHeap Example



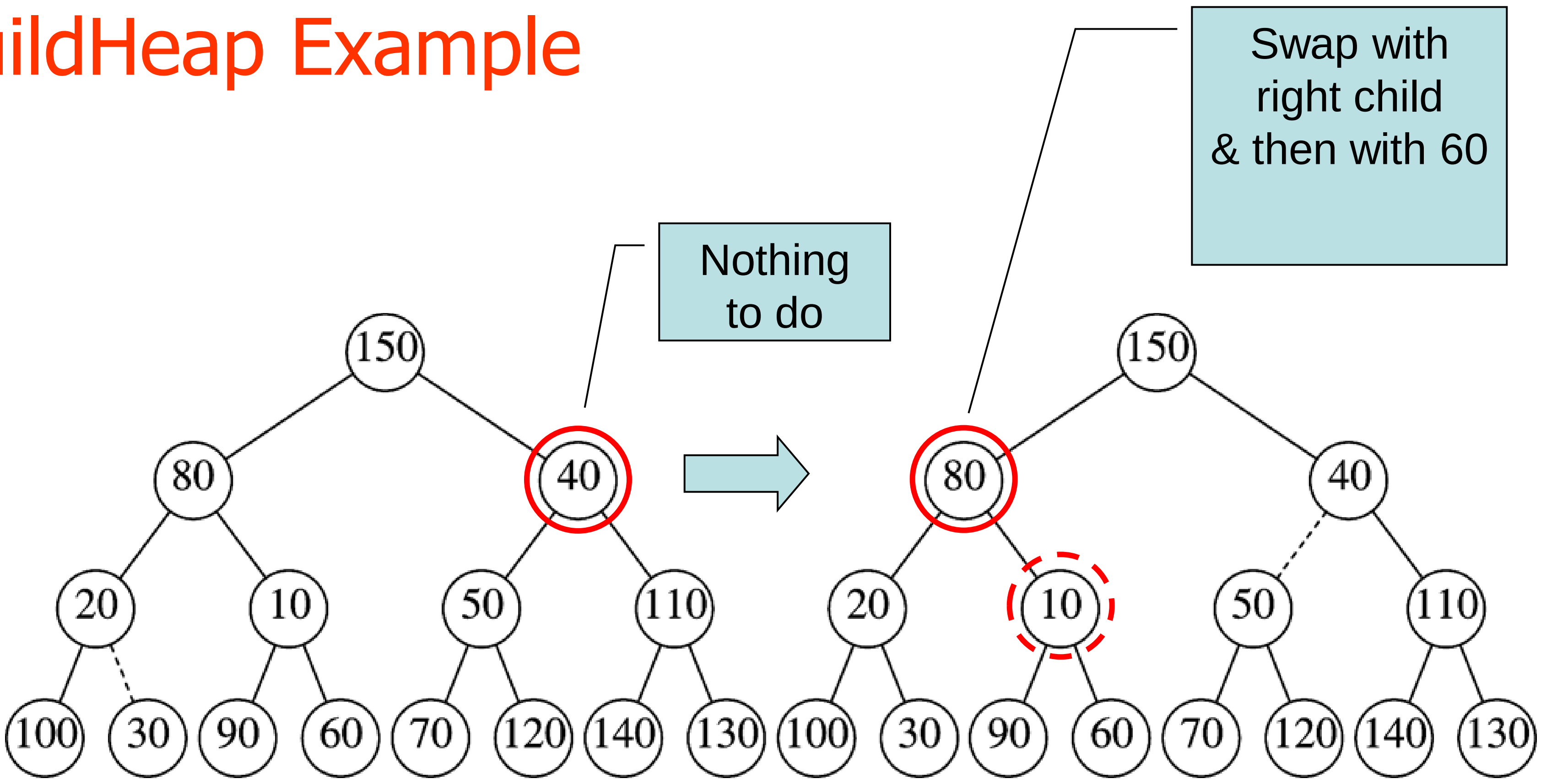
- Randomly initialized heap
- Structure property satisfied
- Heap order property violated
- Leaves are all valid heaps (implicit)

BuildHeap Example



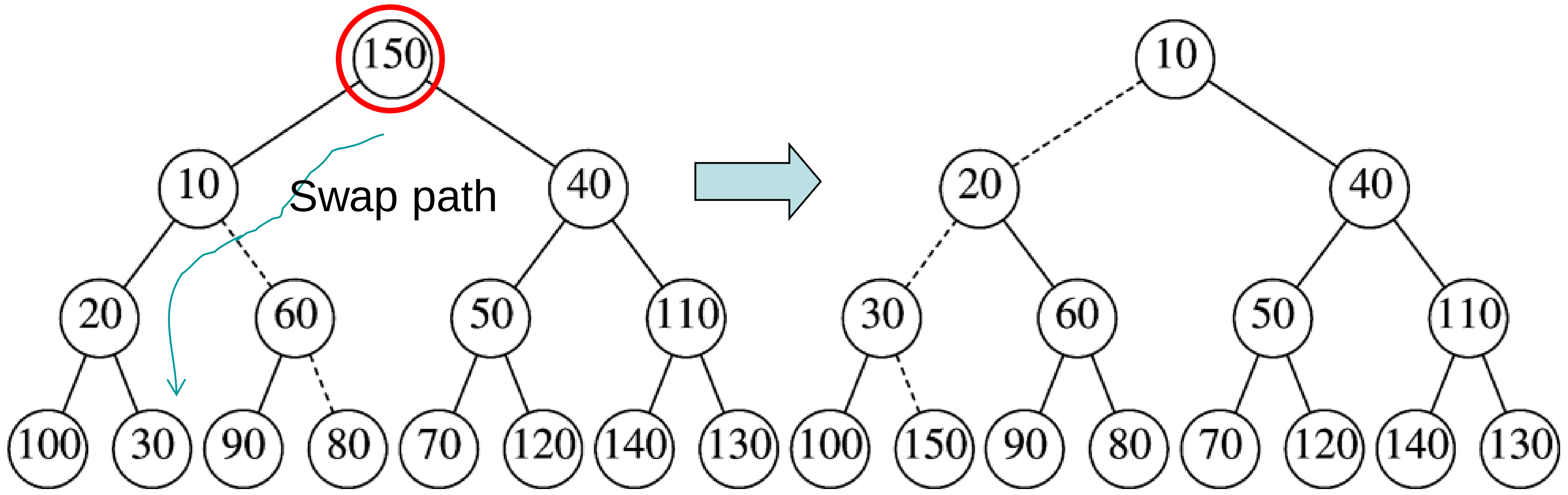
Dotted lines show path of percolating down

BuildHeap Example



Dotted lines show path of percolating down

BuildHeap Example



Final Heap

Dotted lines show path of percolating down

Questions?

"He who asks a question is a fool for five minutes; he who does not ask a question remains a fool forever"
Chinese Proverb

"The wise man doesn't give the right answers, he poses the right questions."
Claude Levi-Strauss

"A wise man can learn more from a foolish question than a fool can learn from a wise answer."
Bruce Lee