Data Structures

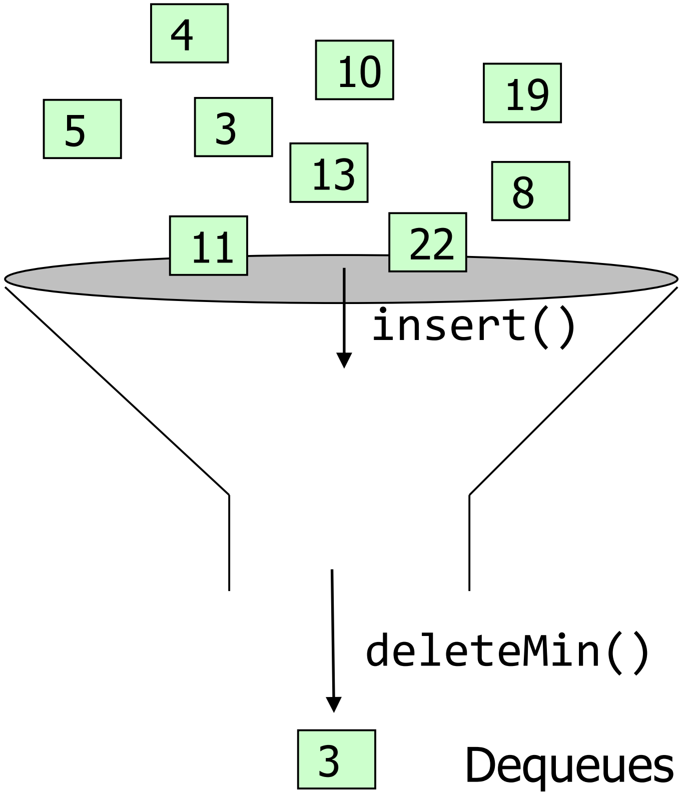
Fall 2023

---

18. Heap Sort

Binary Heap

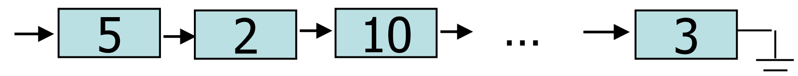
Recall: Priority Queue



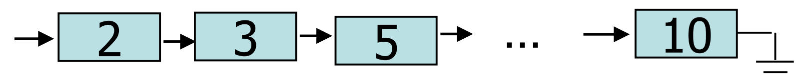
Dequeues the next element with the highest priority

Recall: Priority Queue

- Unordered linked list
  - `Insert` – `O(1)` step
  - `deleteMin` – `O(n)` steps



- Ordered linked list
  - `insert` – `O(n)` steps
  - `deleteMin` – `O(1)` step



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

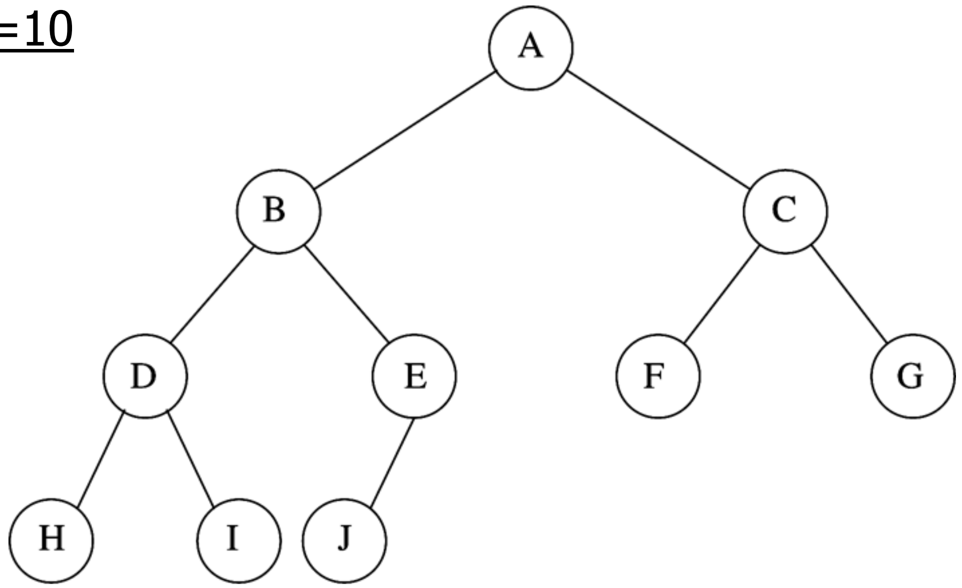
Binary Heap

- 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)

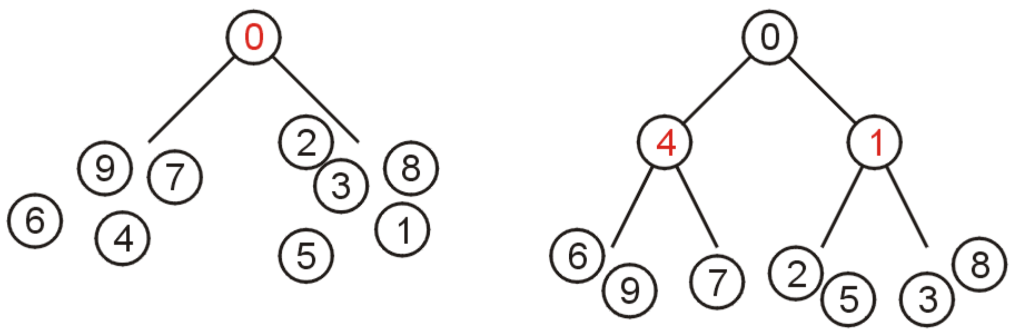
N=10



Every level (except last) saturated

Binary Heap – Heap-Order Property

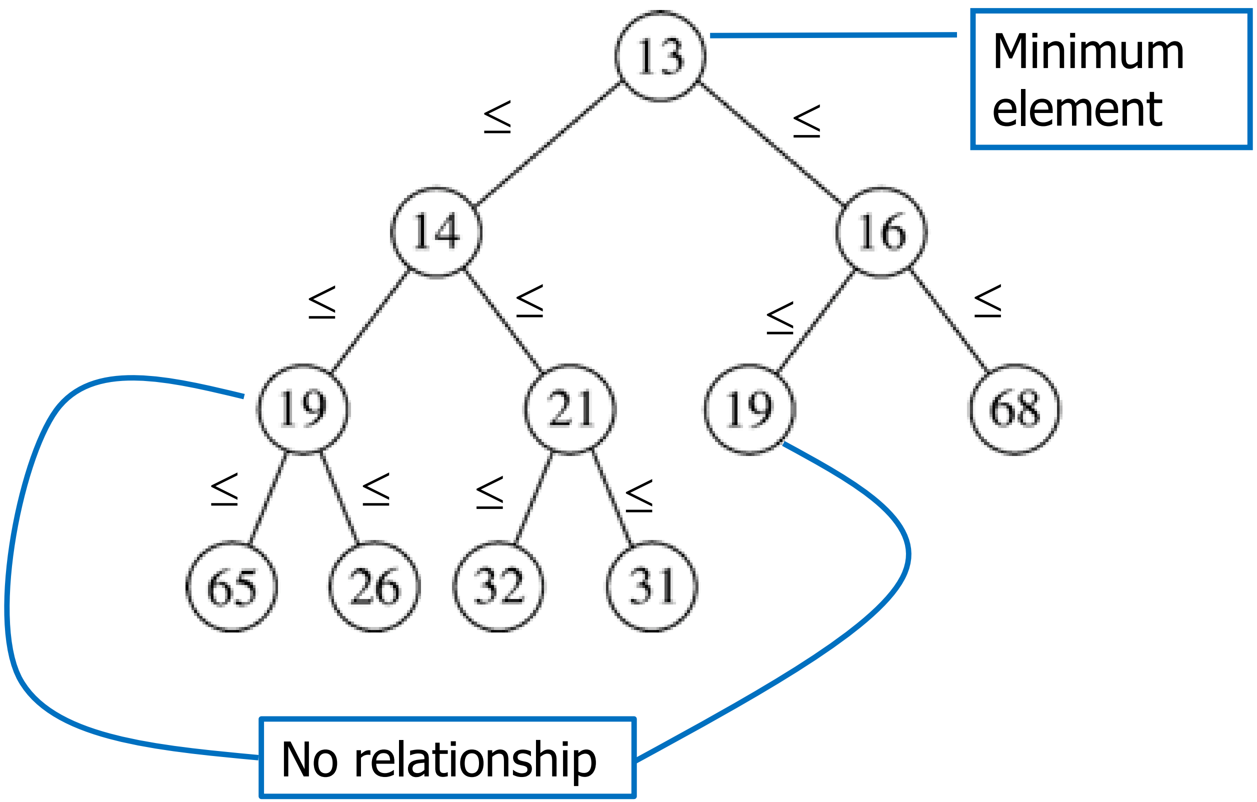
- Min-Heap property
  - Key associated with the root is less than or equal to the keys associated with either of the sub-trees (if any)
  - Both of the sub-trees (if any) are also binary min-heaps



- Properties of min-heap
  - A single node is a min-heap
  - Minimum key always at root
  - For every node `X`, `key(parent(X)) ≤ key(X)`
  - No relationship between nodes with similar key

Heap-Order Property – Example

- Min-Heap



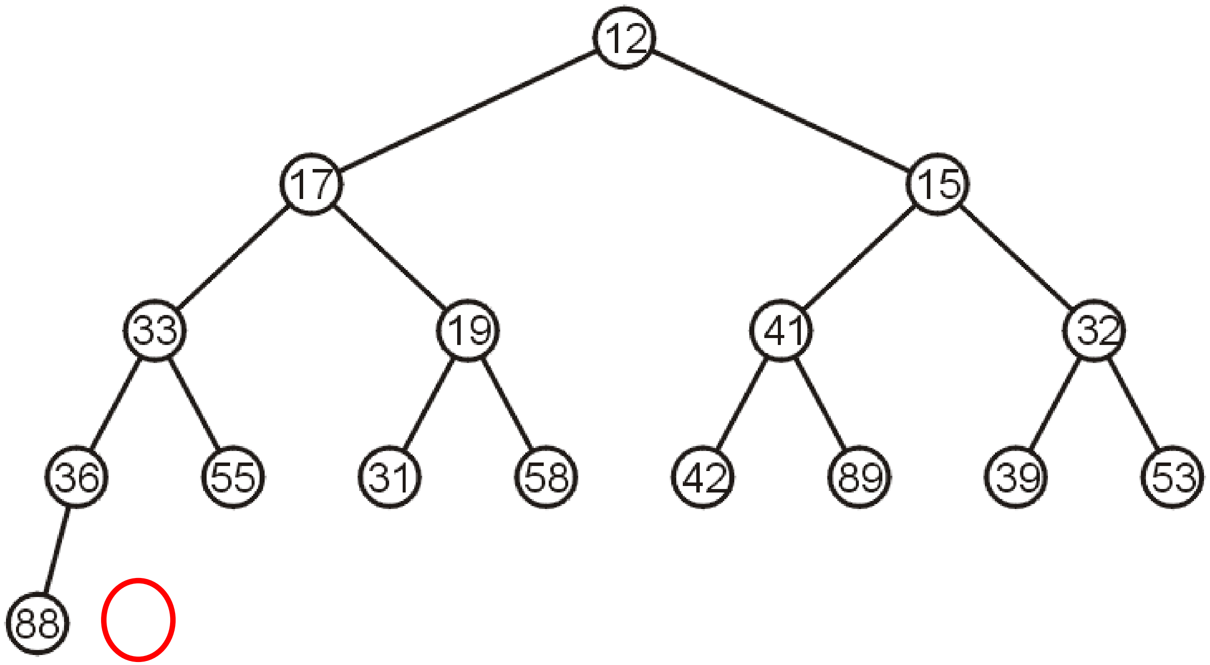
Binary Heap – Heap-Order Property

- Max-Heap property
  - Maximum key at the root
  - For every node `X`, `key(parent(X)) ≥ key(X)`

- `Insert` and `deleteMin` must maintain heap-order property

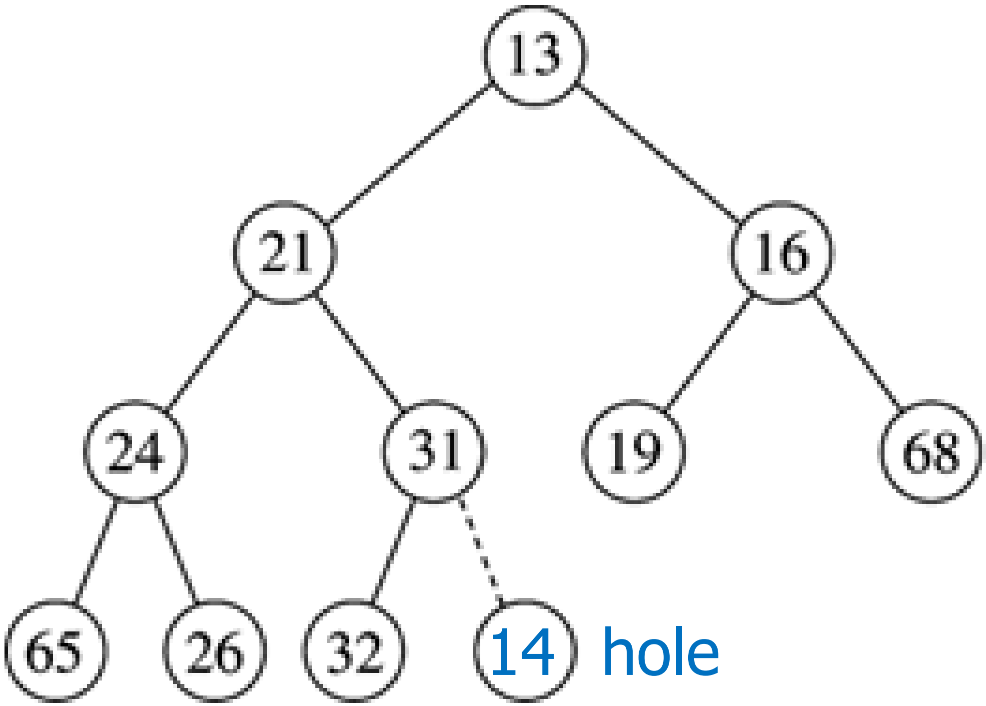
Heap Operations – `insert`

- 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



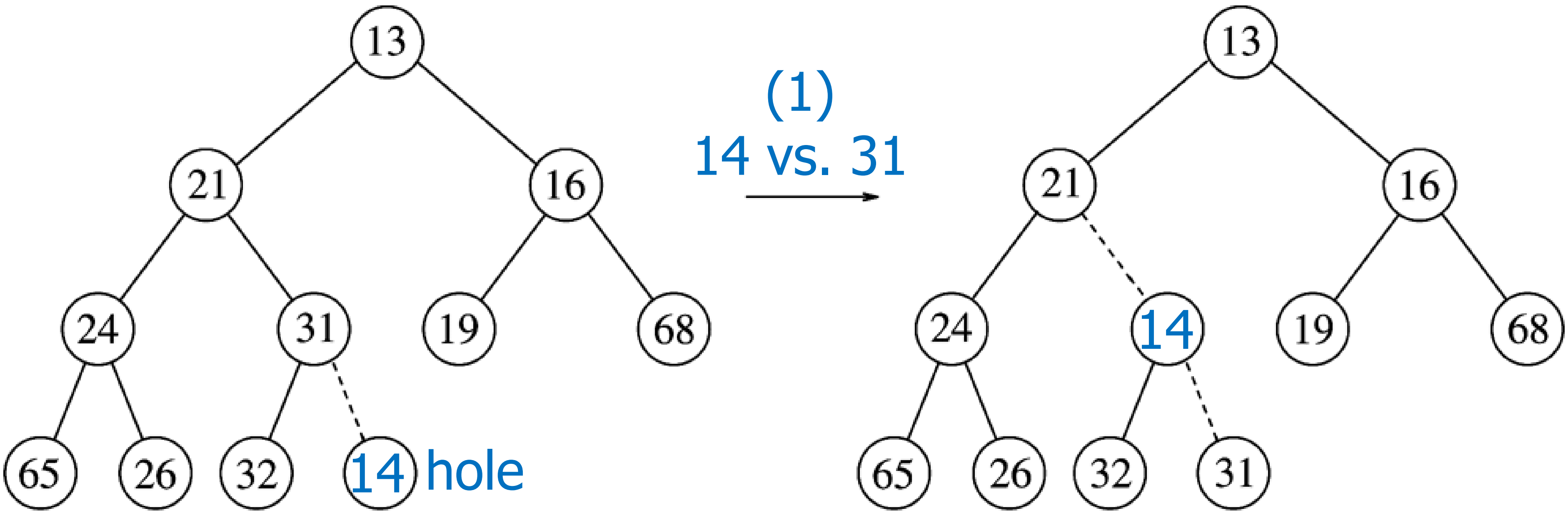
Heap Insert – Example

- Insert 14



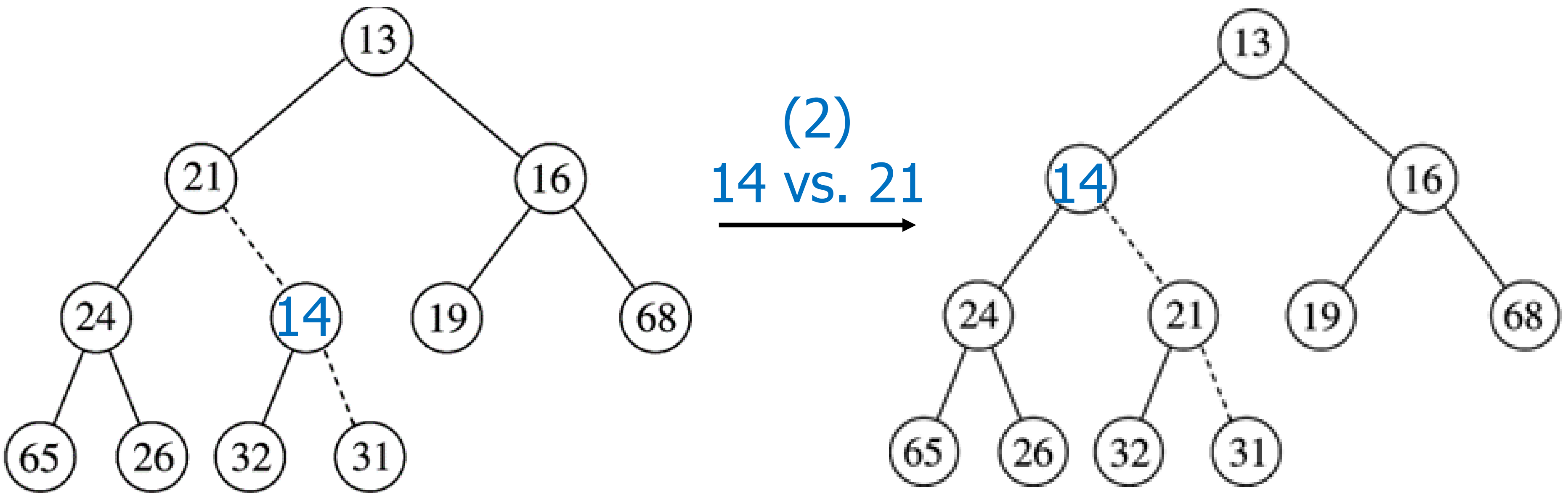
Heap Insert – Example

- Insert 14



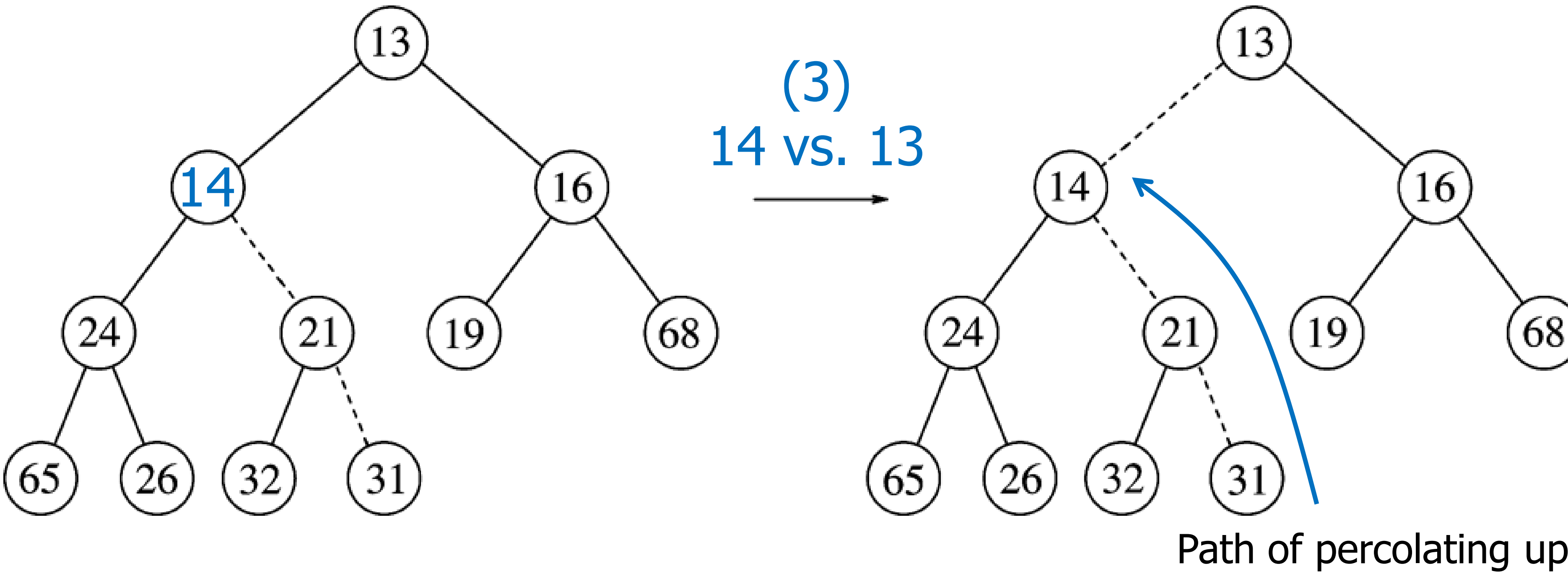
Heap Insert – Example

- Insert 14



Heap Insert – Example

- Insert 14

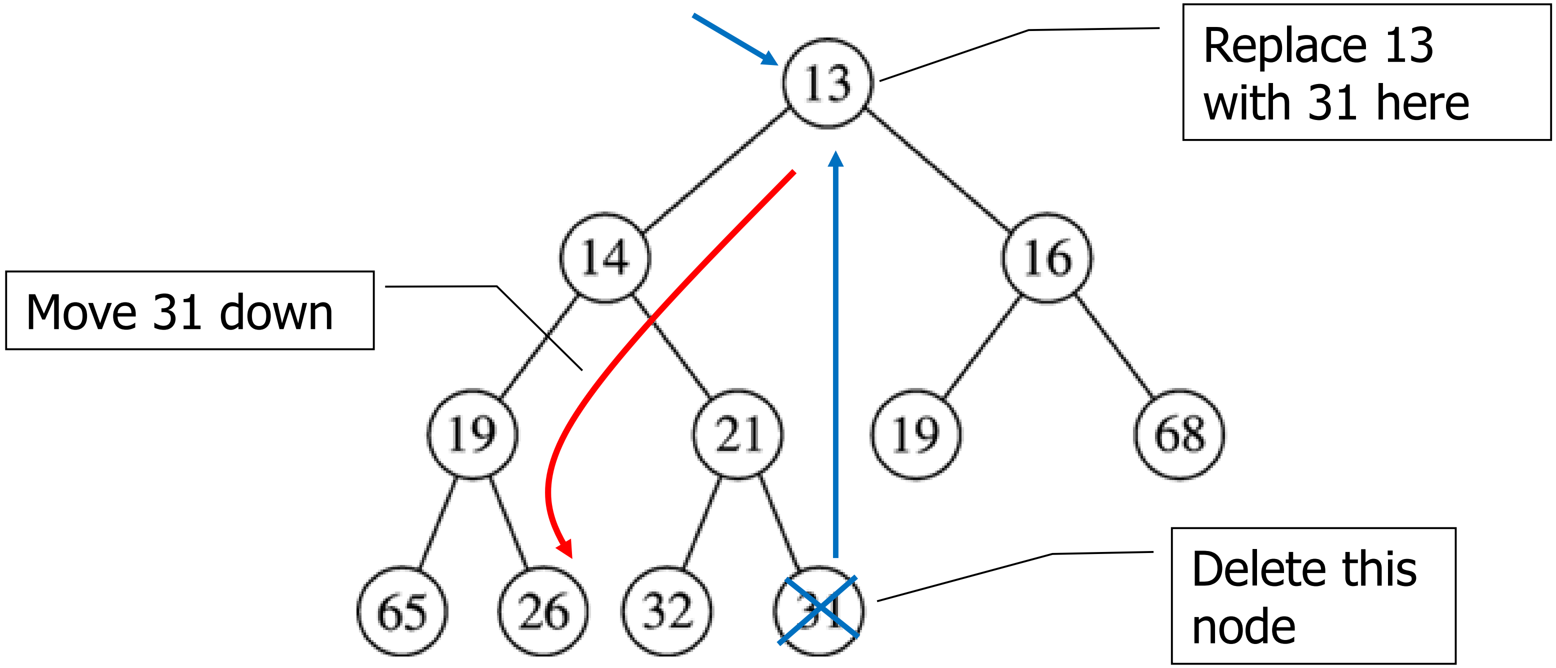


Heap order property

Structure property

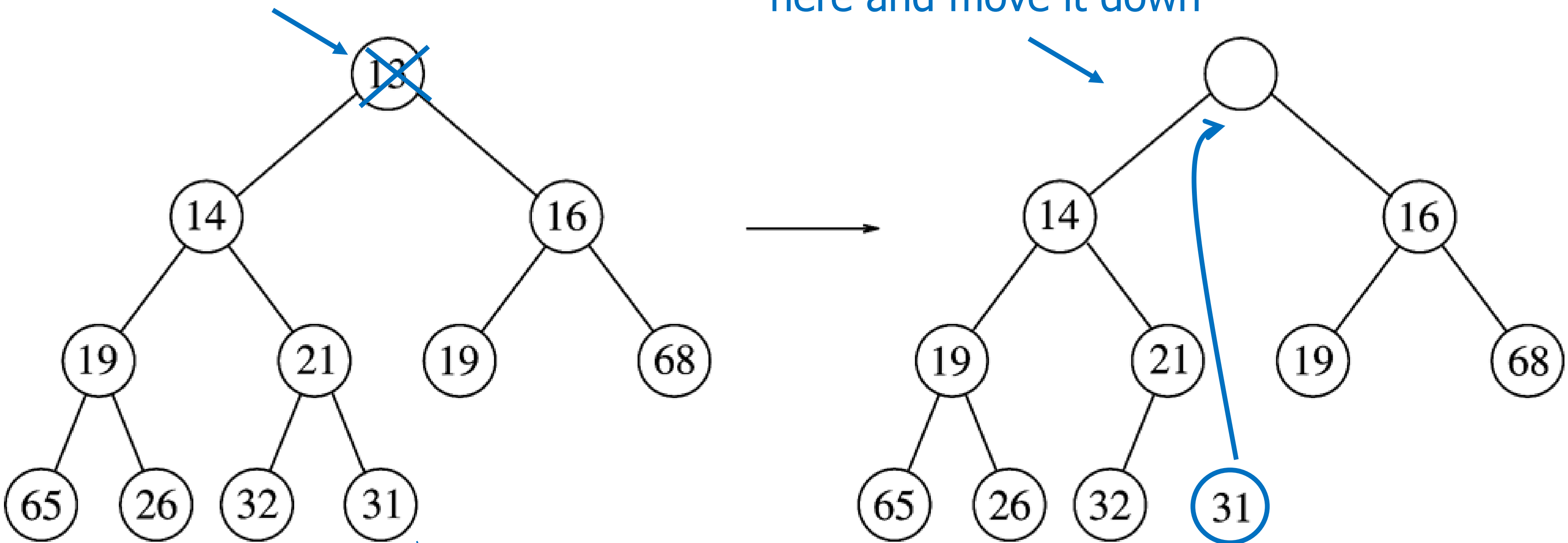
Heap Operation – `deleteMin`

- Minimum element is always at the root
  - Return the element at the root
- Copy value of last element of the tree into hole at root and delete last node
- **Heapify:** Percolate down until heap-order property not satisfied



`deleteMin` – Example

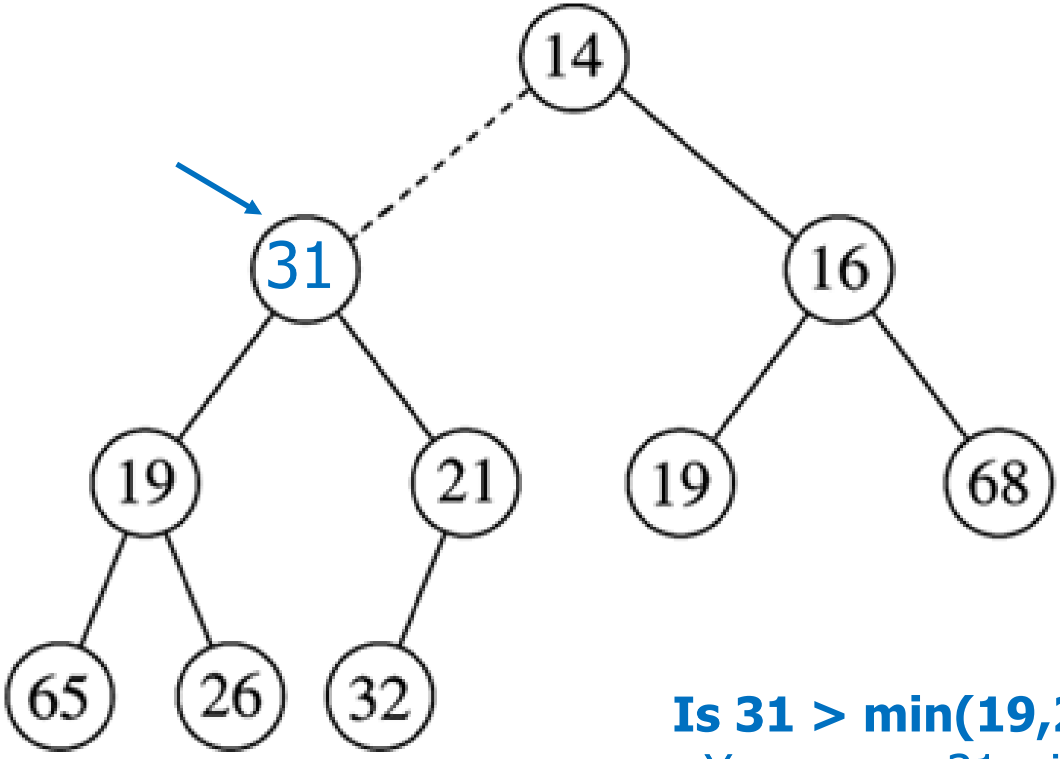
Copy 31 temporarily
here and move it down



Make this
position
empty

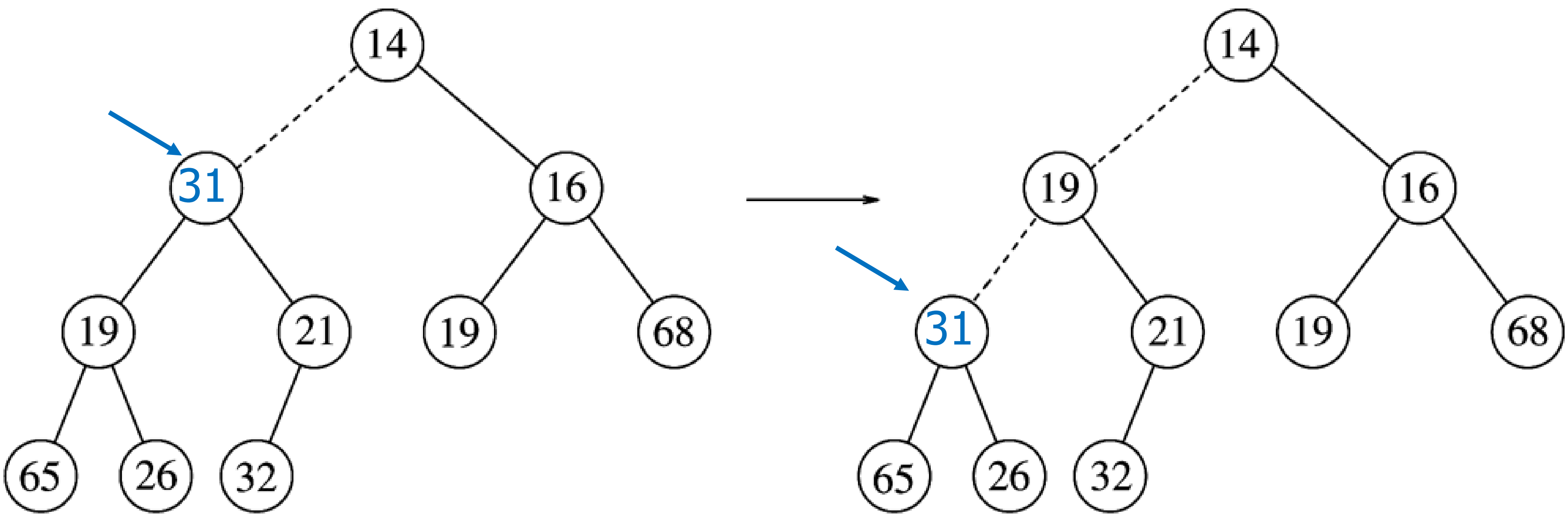
Is 31 > min(14,16)?
- Yes - swap 31 with min(14,16)

`deleteMin` – Example



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

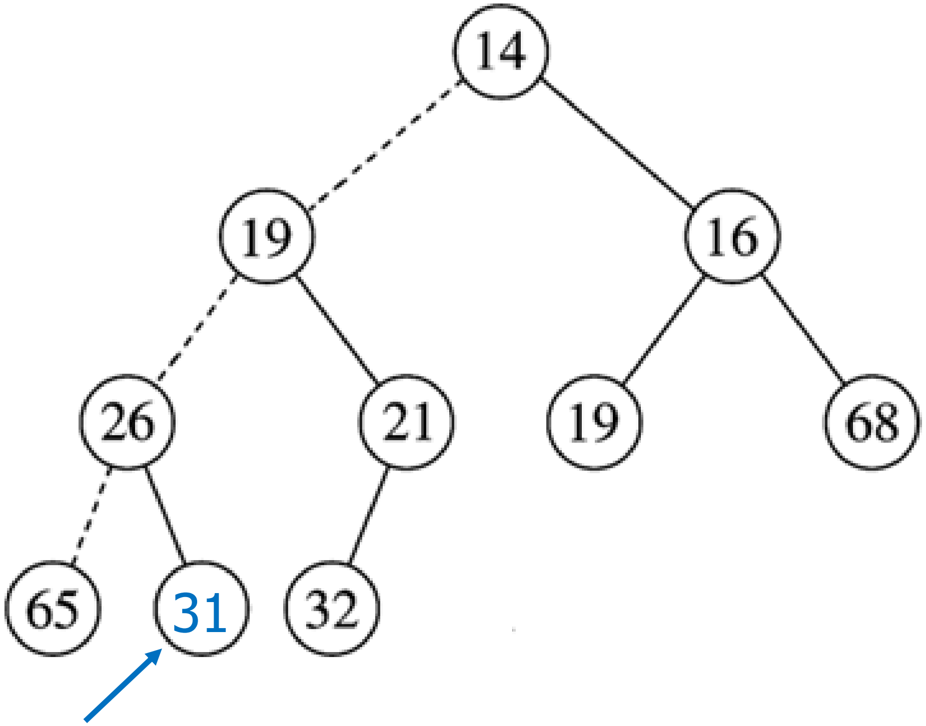
`deleteMin` – Example



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



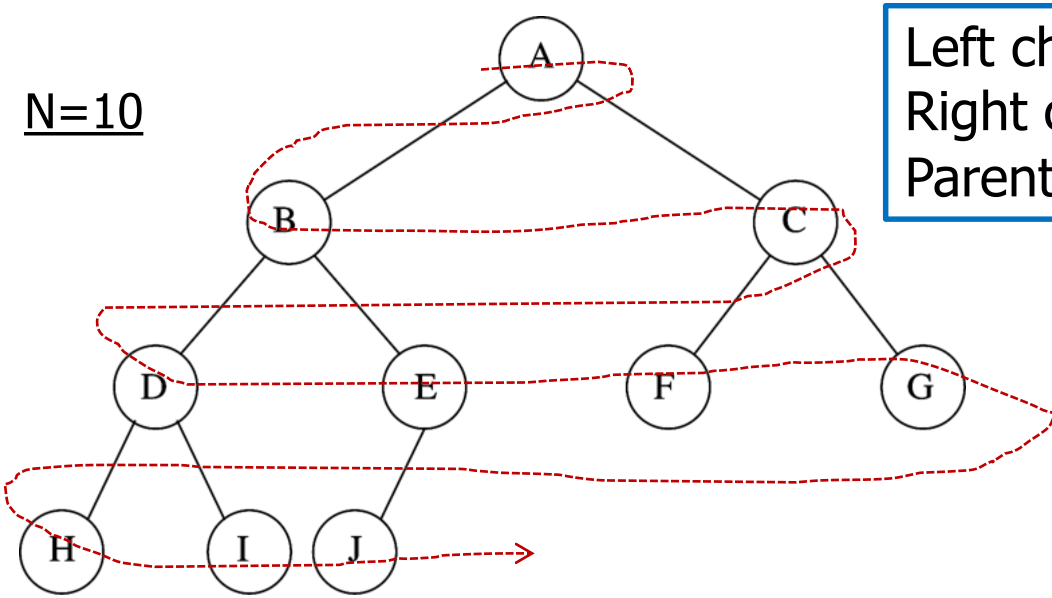
Runtime Analysis

- `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

Array-Based Implementation Of Binary Tree

N=10



Left child(i) = at position 2i
Right child(i) = at position 2i + 1
Parent(i) = at position $\lfloor i / 2 \rfloor$

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

i

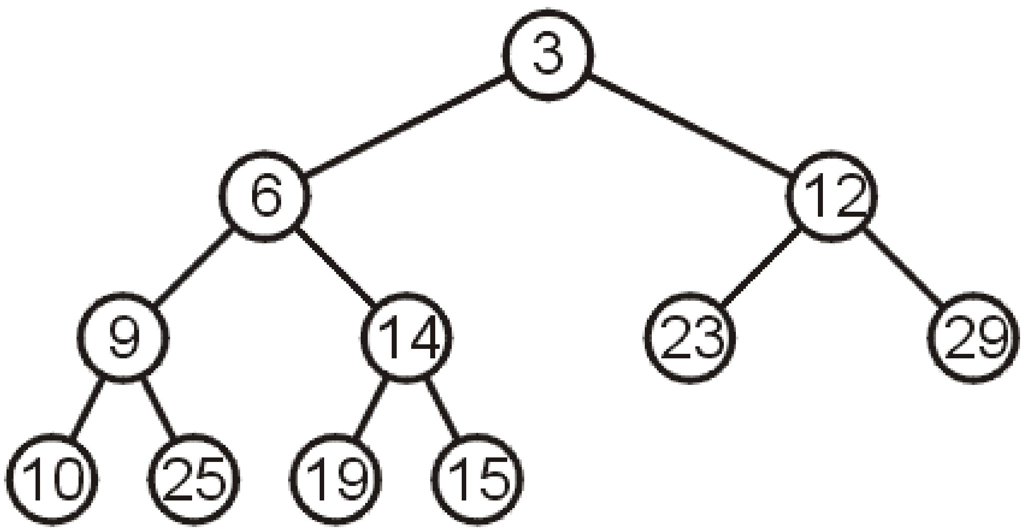
2i + 1

$\lfloor i / 2 \rfloor$

2i

Array-Based Implementation Of Binary Heap

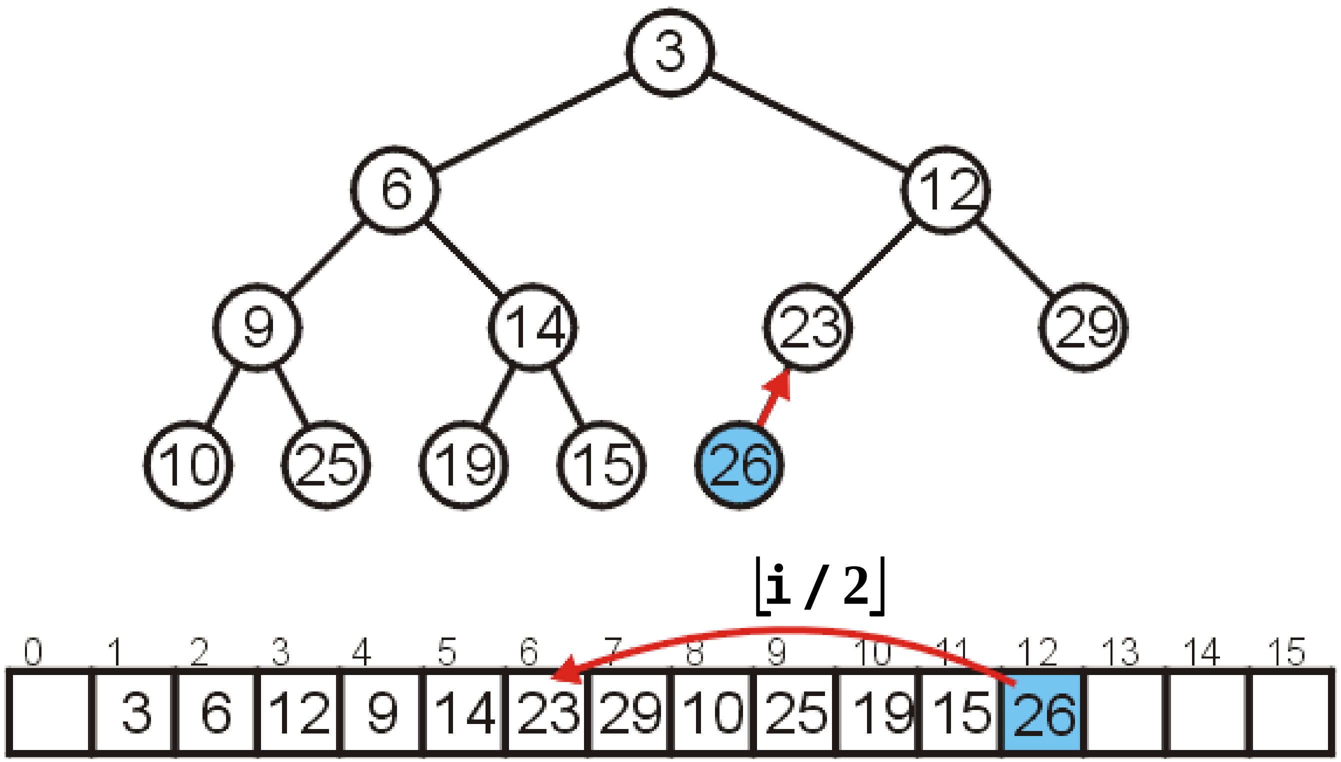
- Consider the following heap, both as a tree and in its array representation



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	12	9	14	23	29	10	25	19	15				

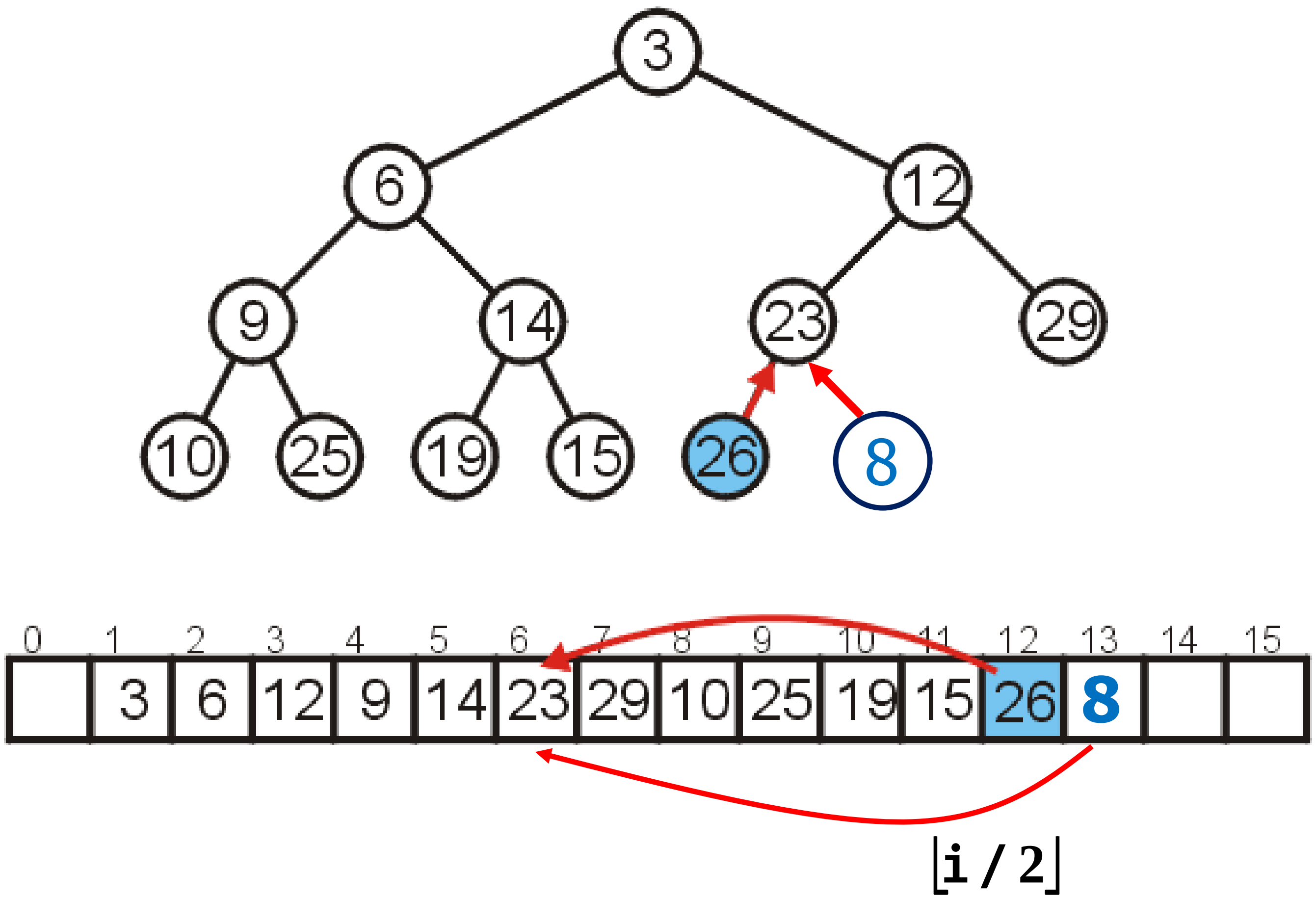
Array-Based Implementation – `insert`

- Inserting 26 requires no changes



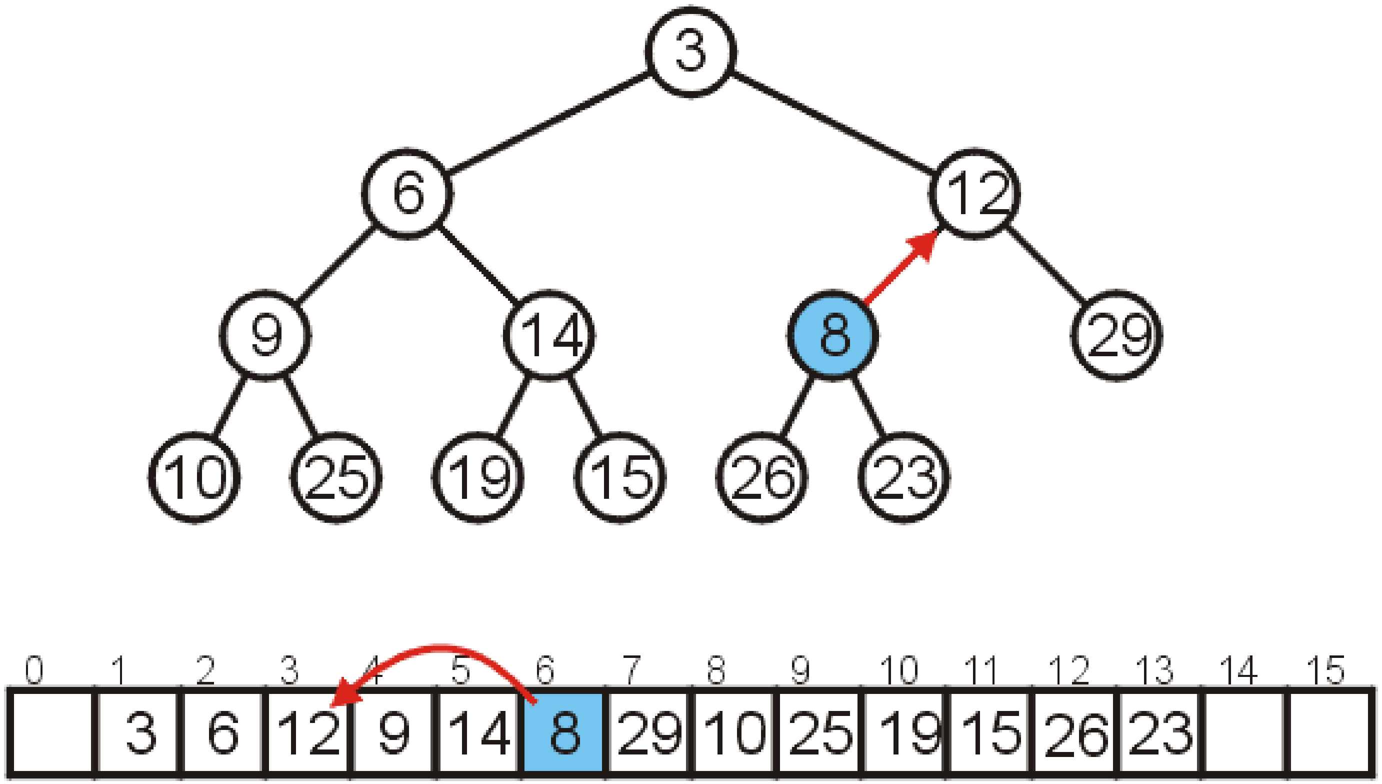
Array-Based Implementation – `insert`

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



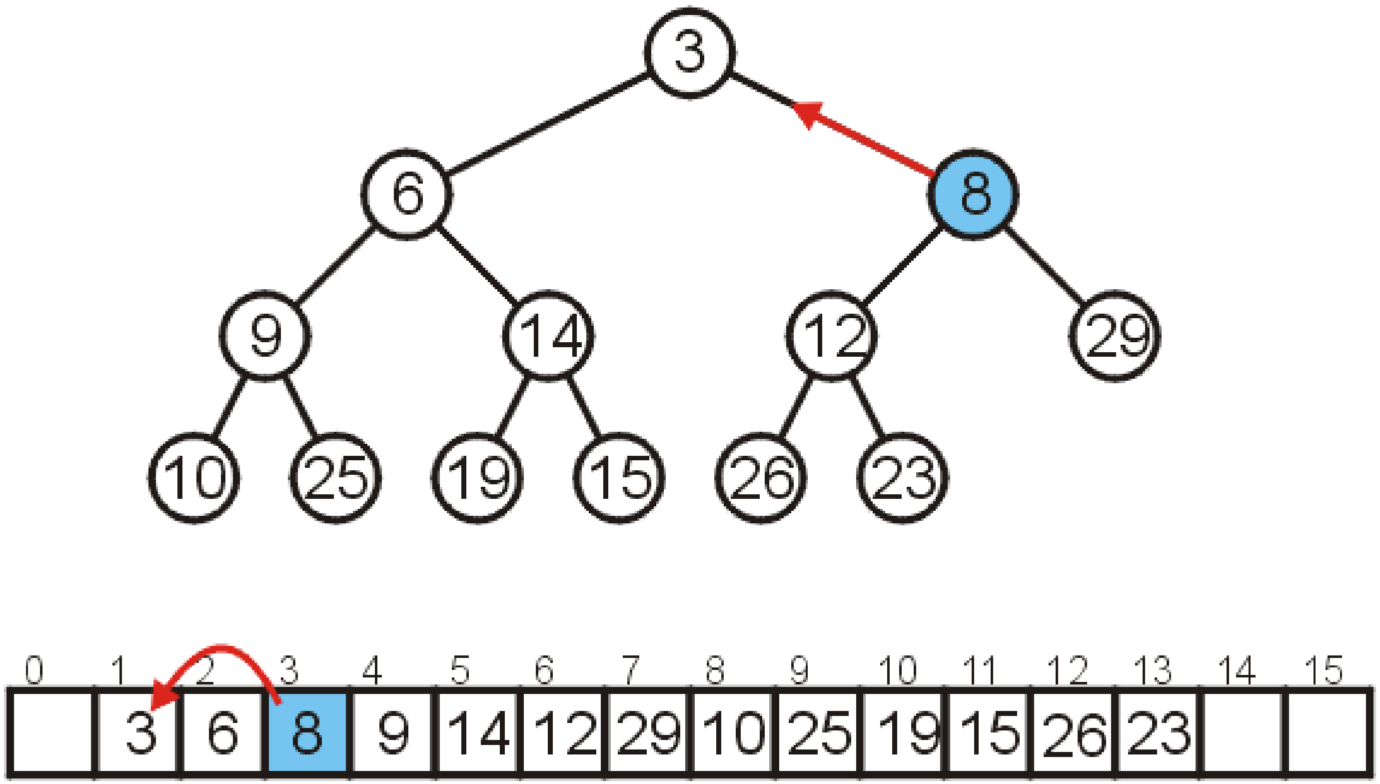
Array-Based Implementation – `insert`

- Swap 8 and 12



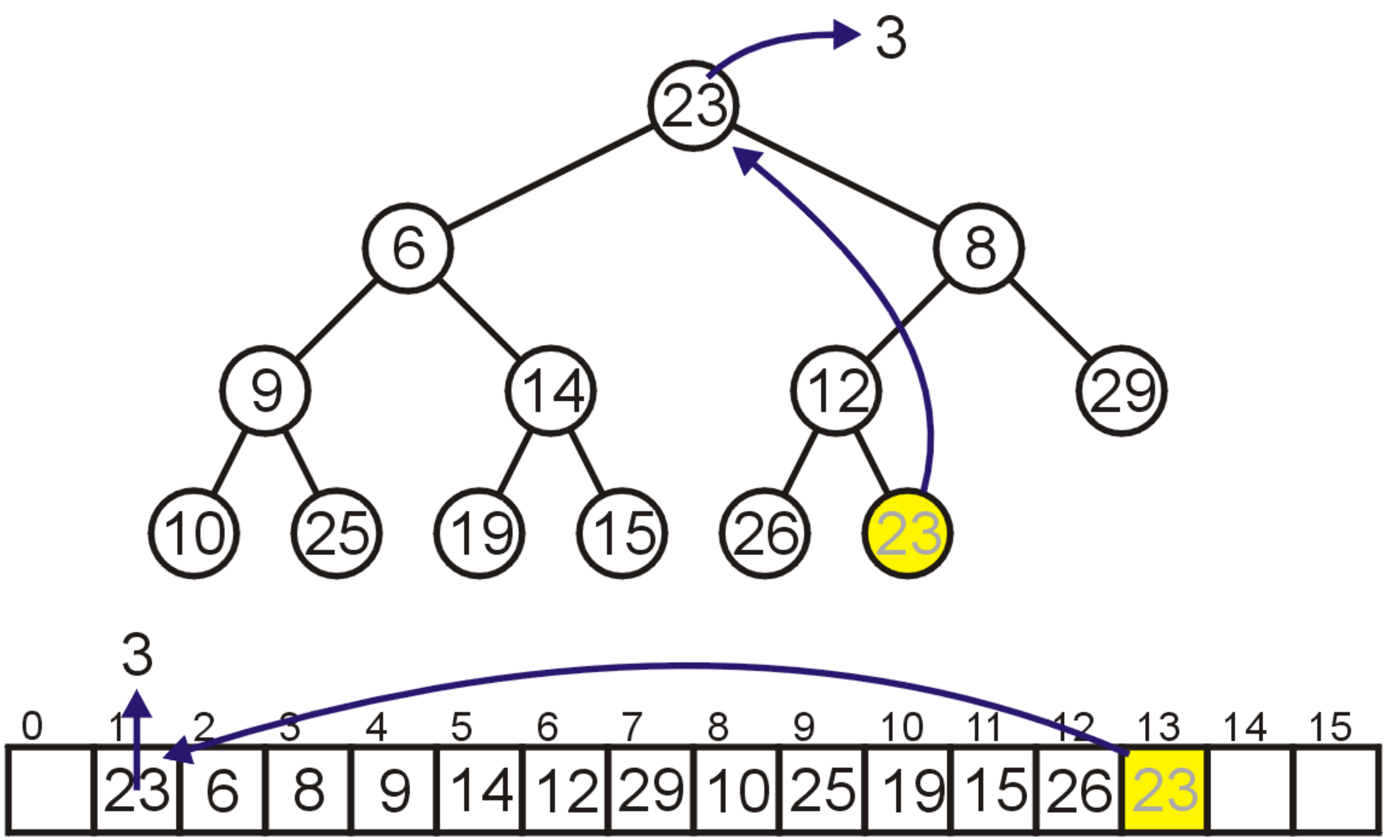
Array-Based Implementation – `insert`

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



Array-Based Implementation – `deleteMin`

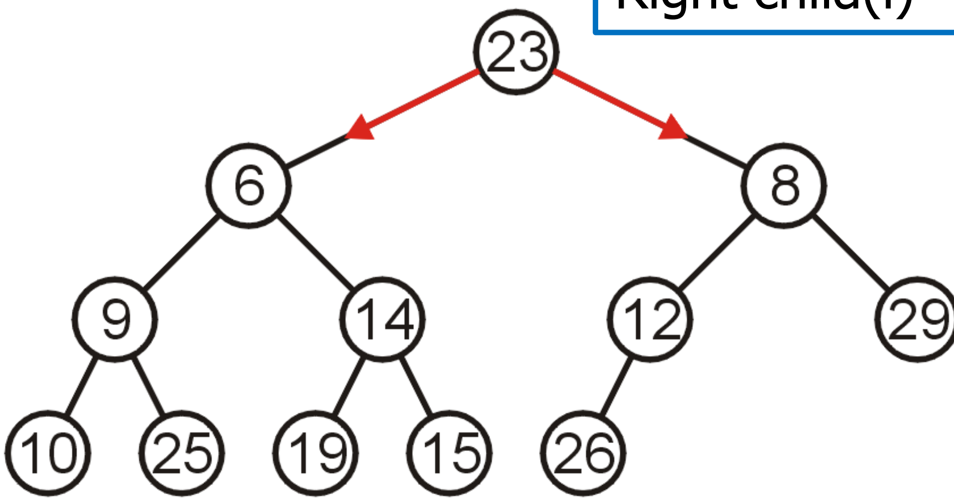
- Removing the top require copy of the last element to the top

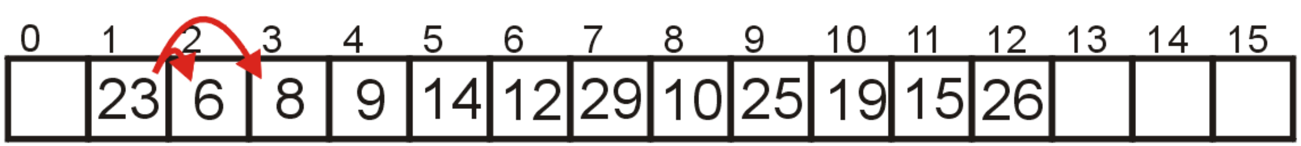


Array-Based Implementation – `deleteMin`

- **Heapify:** 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`

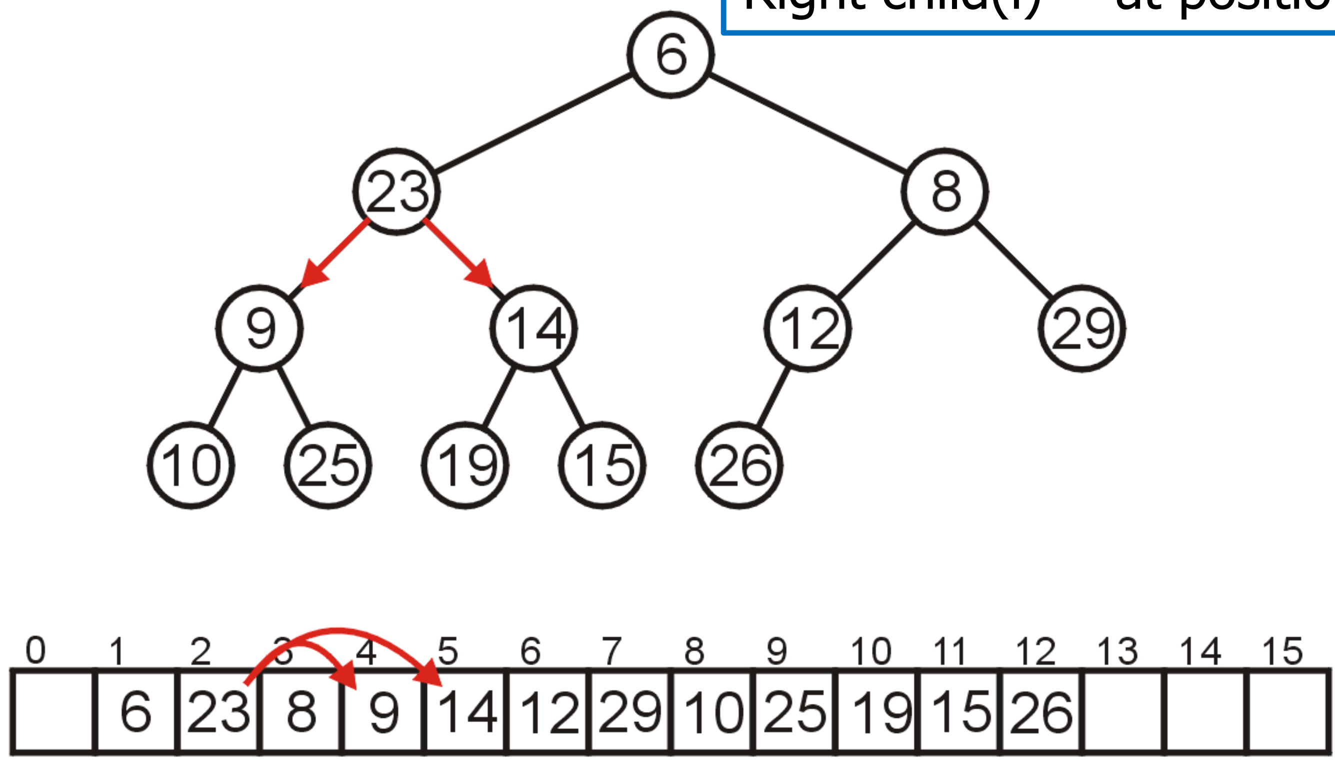




Array-Based Implementation – `deleteMin`

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

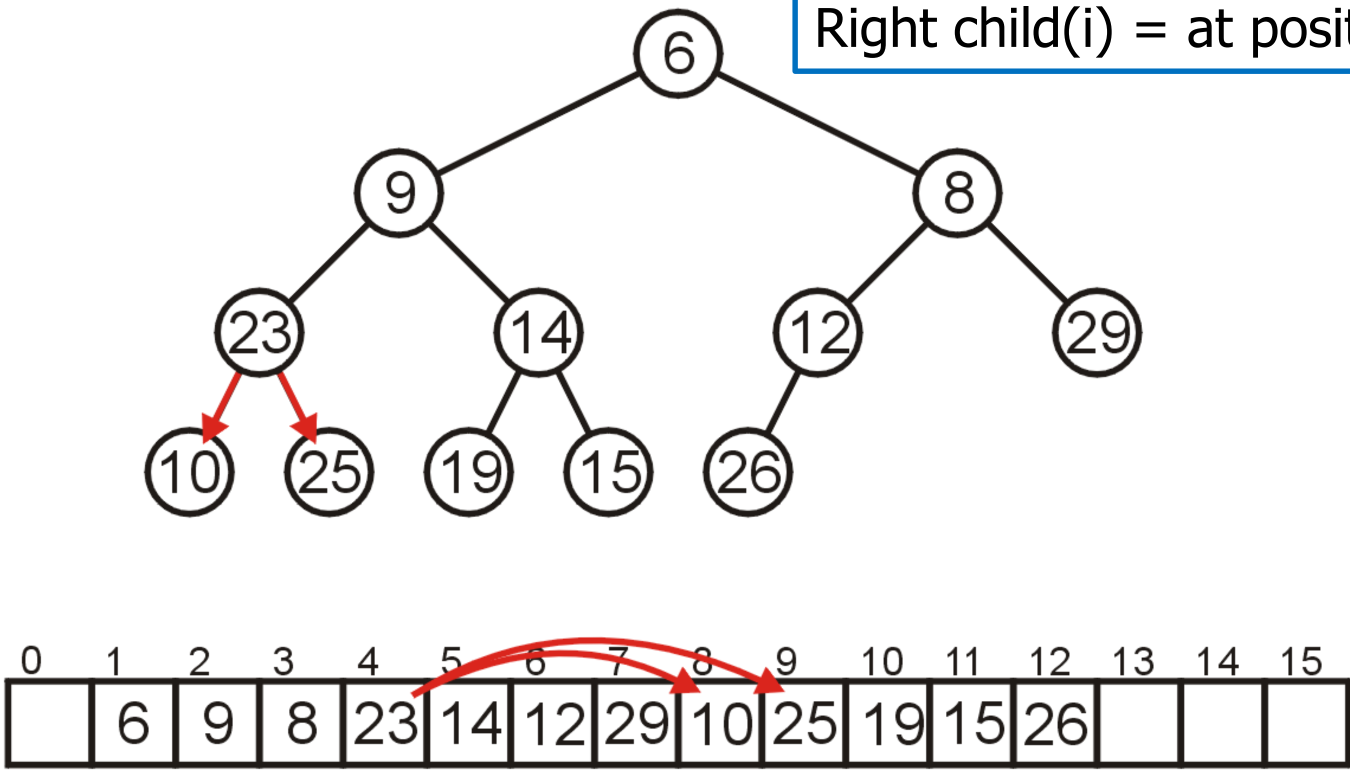
Left child(i) = at position `2i`
Right child(i) = at position `2i + 1`



Array-Based Implementation – `deleteMin`

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

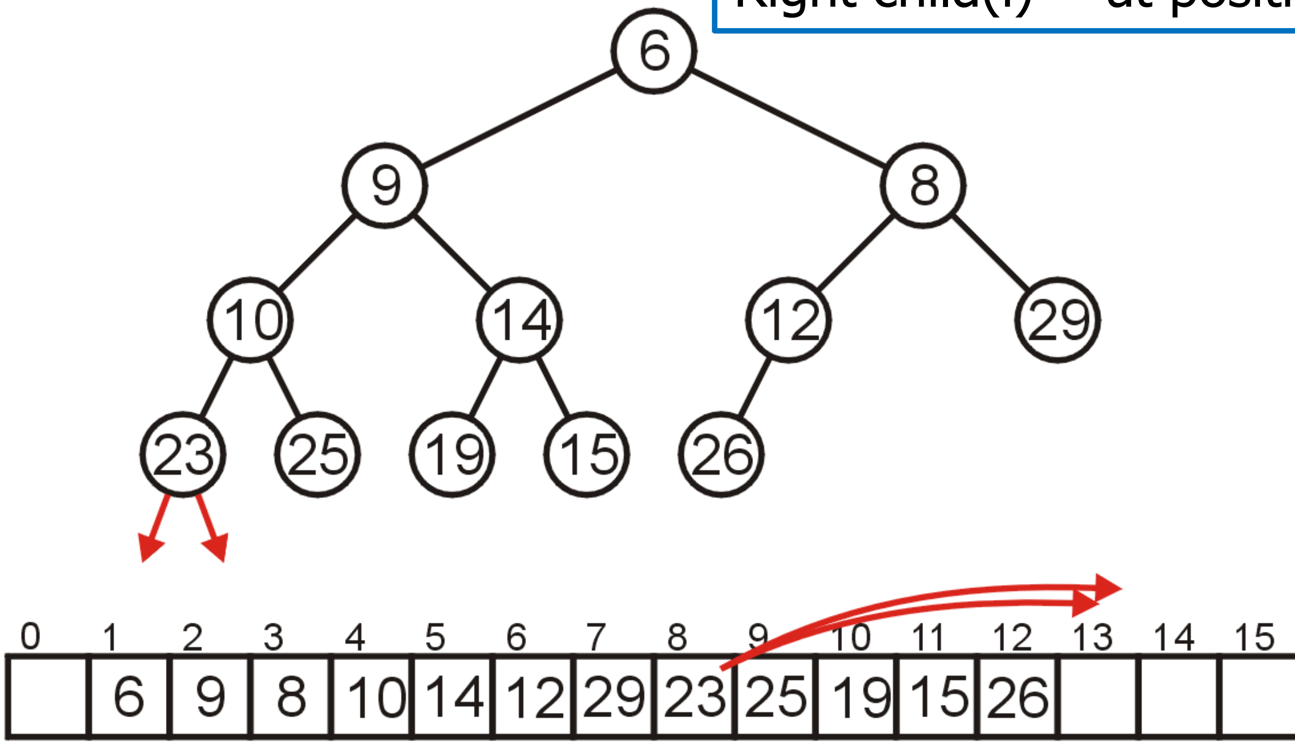
Left child(i) = at position `2i`
Right child(i) = at position `2i + 1`



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`



Building a Heap

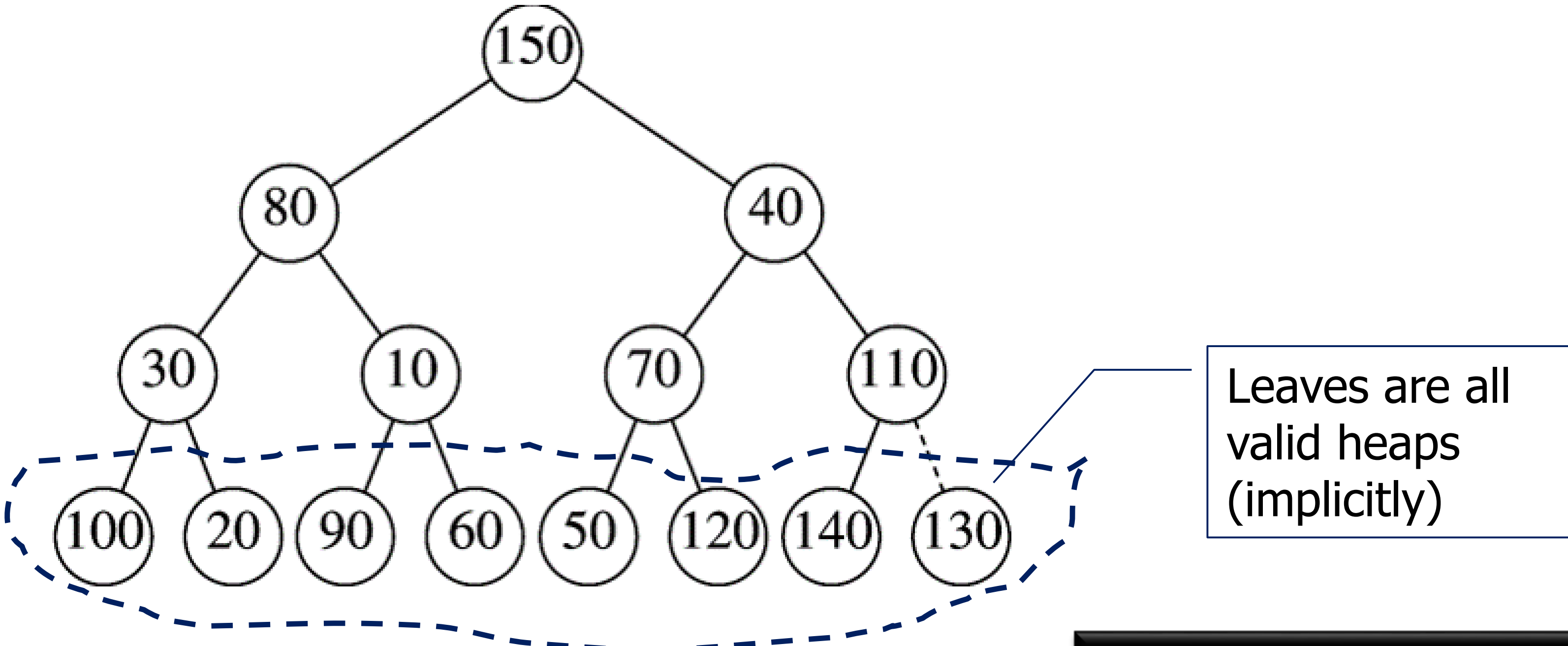
- What if all N elements are all available upfront?
  - Construct heap from initial set of N items

- Solution 1 (insert method)
  - Perform N inserts

- Solution 2 (BuildHeap method)
  - Randomly populate initial heap with structure property
  - Perform a heapify/percolate-down operation from each internal node
    - To take care of heap order property

BuildHeap Example

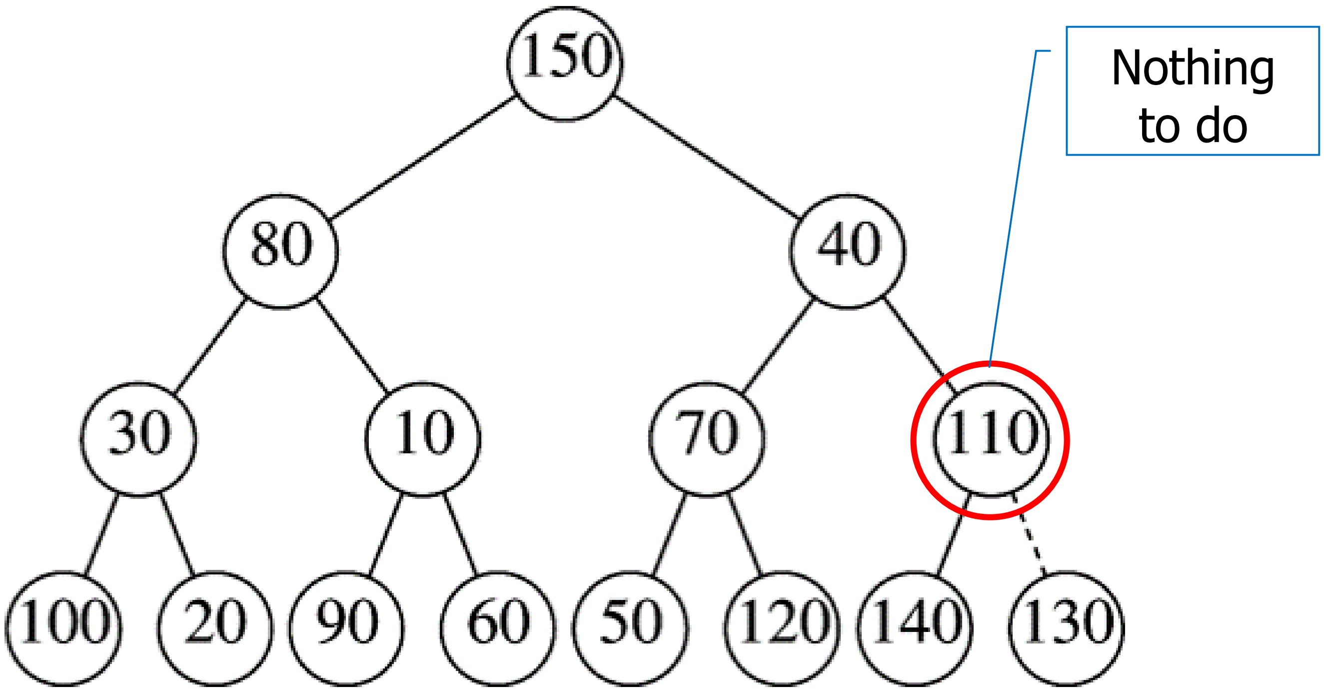
- Input priority levels
  - { 150, 80, 40, 30, 10, 70, 110, 100, 20, 90, 60, 50, 120, 140, 130 }



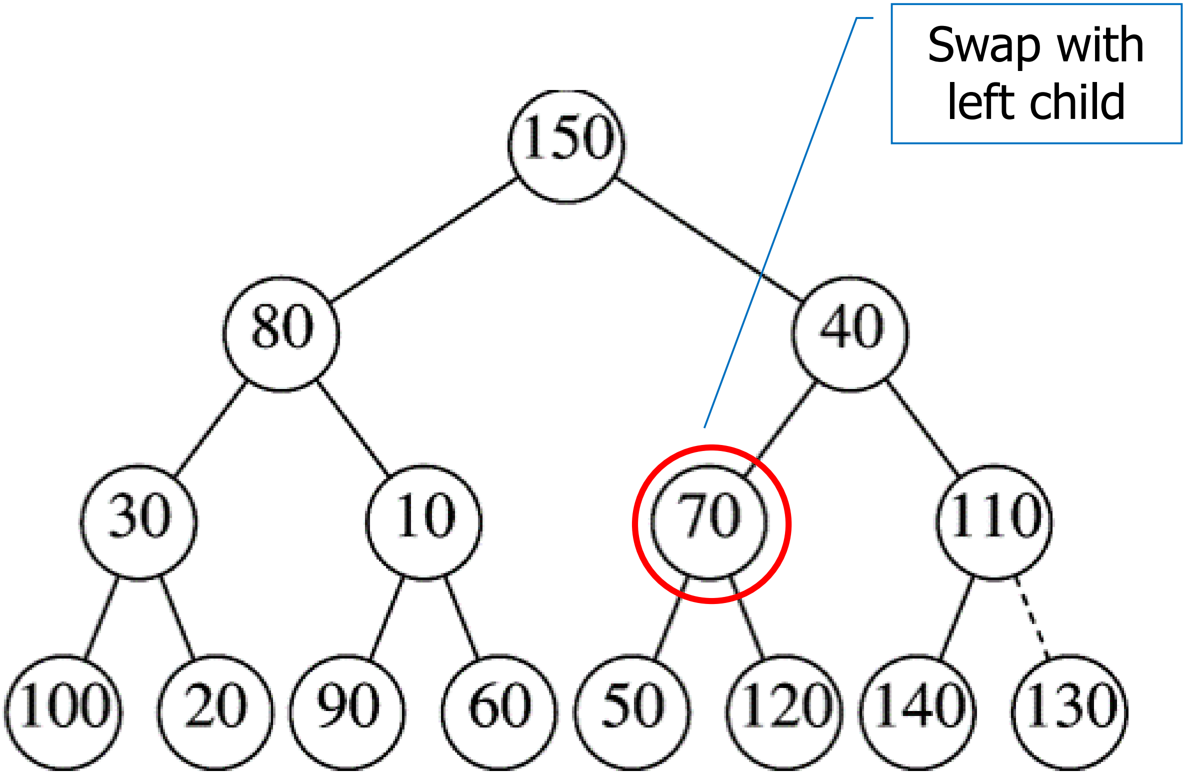
- Arbitrarily assign elements to heap nodes
- Structure property satisfied
- Heap order property violated
- Leaves are all valid heaps (implicit)

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

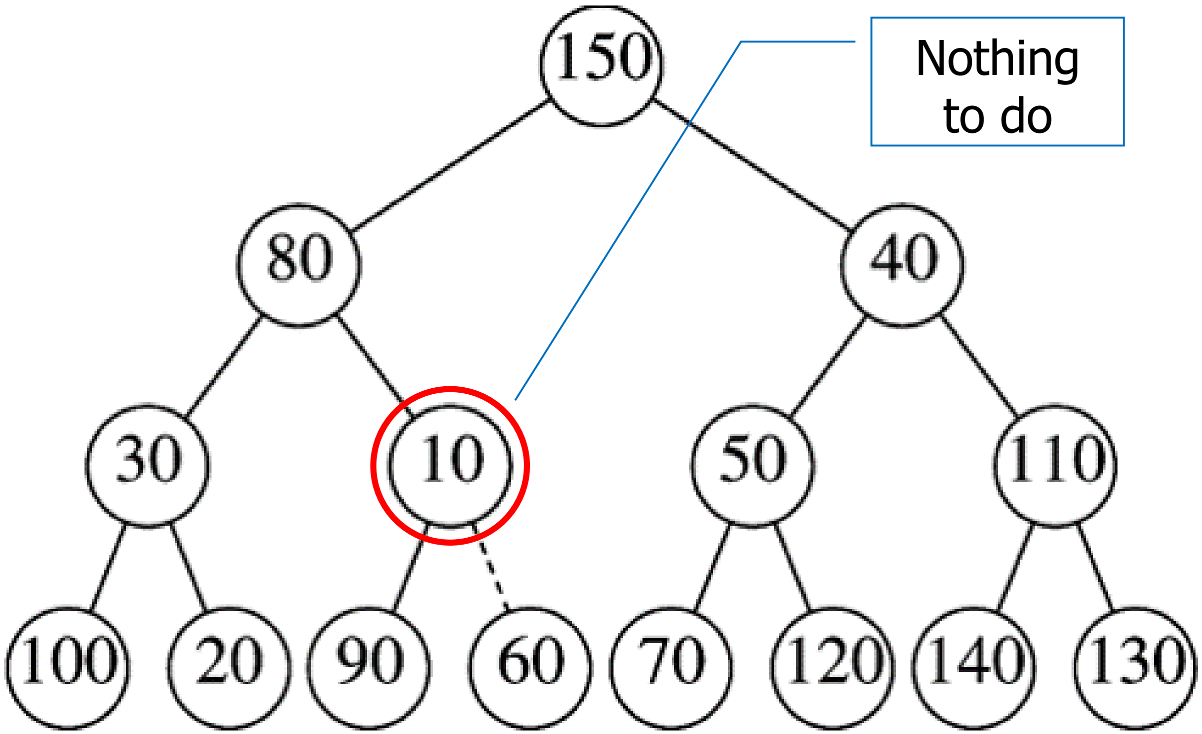
BuildHeap Example



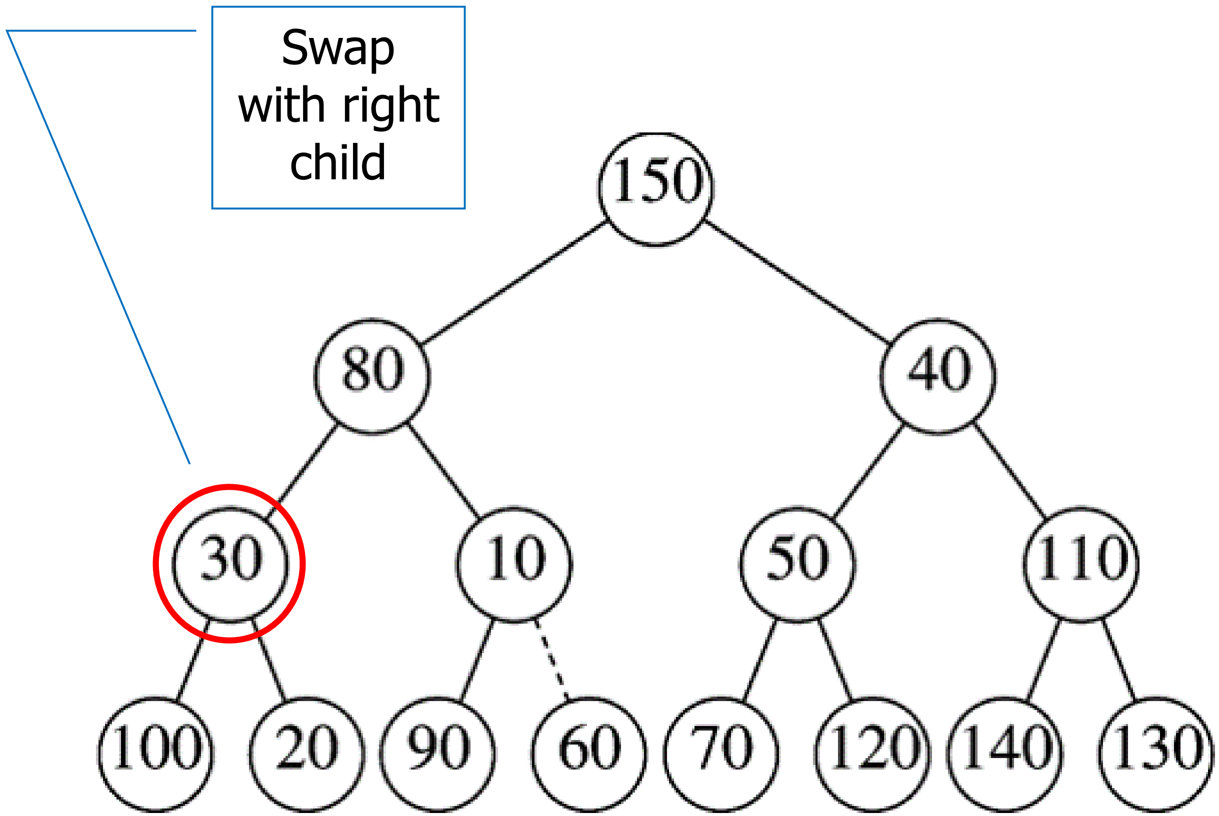
BuildHeap Example



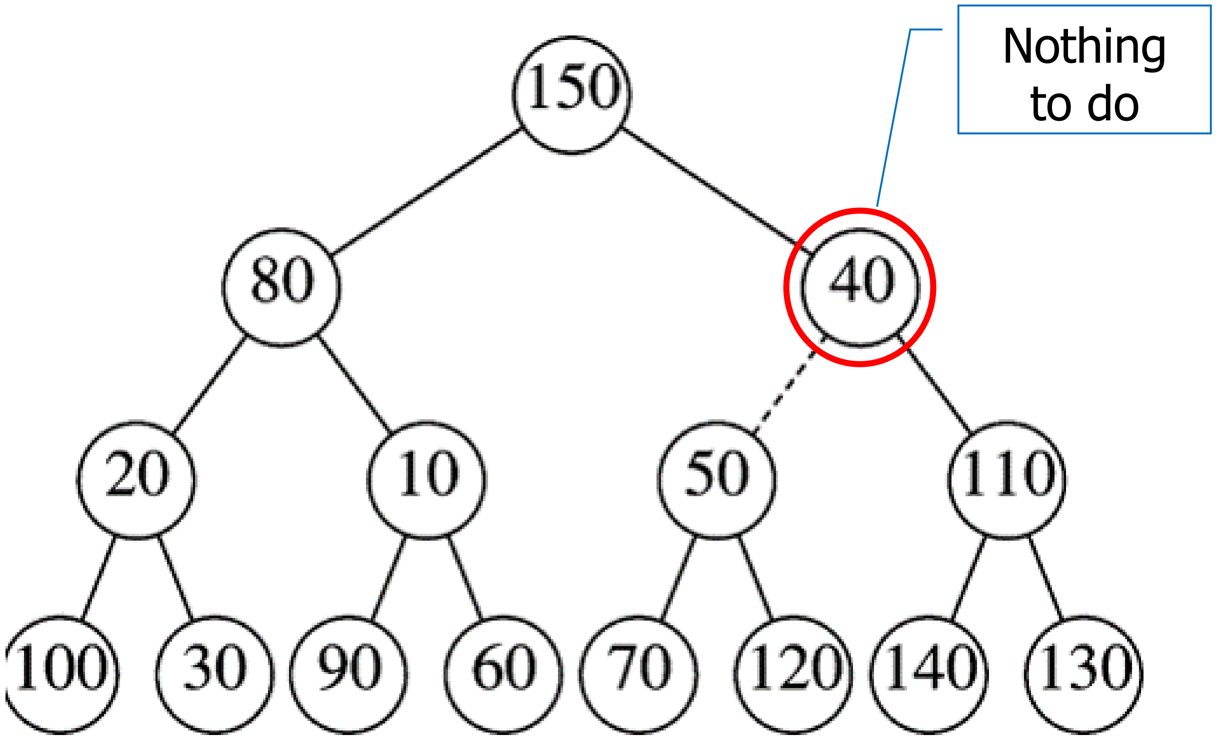
BuildHeap Example



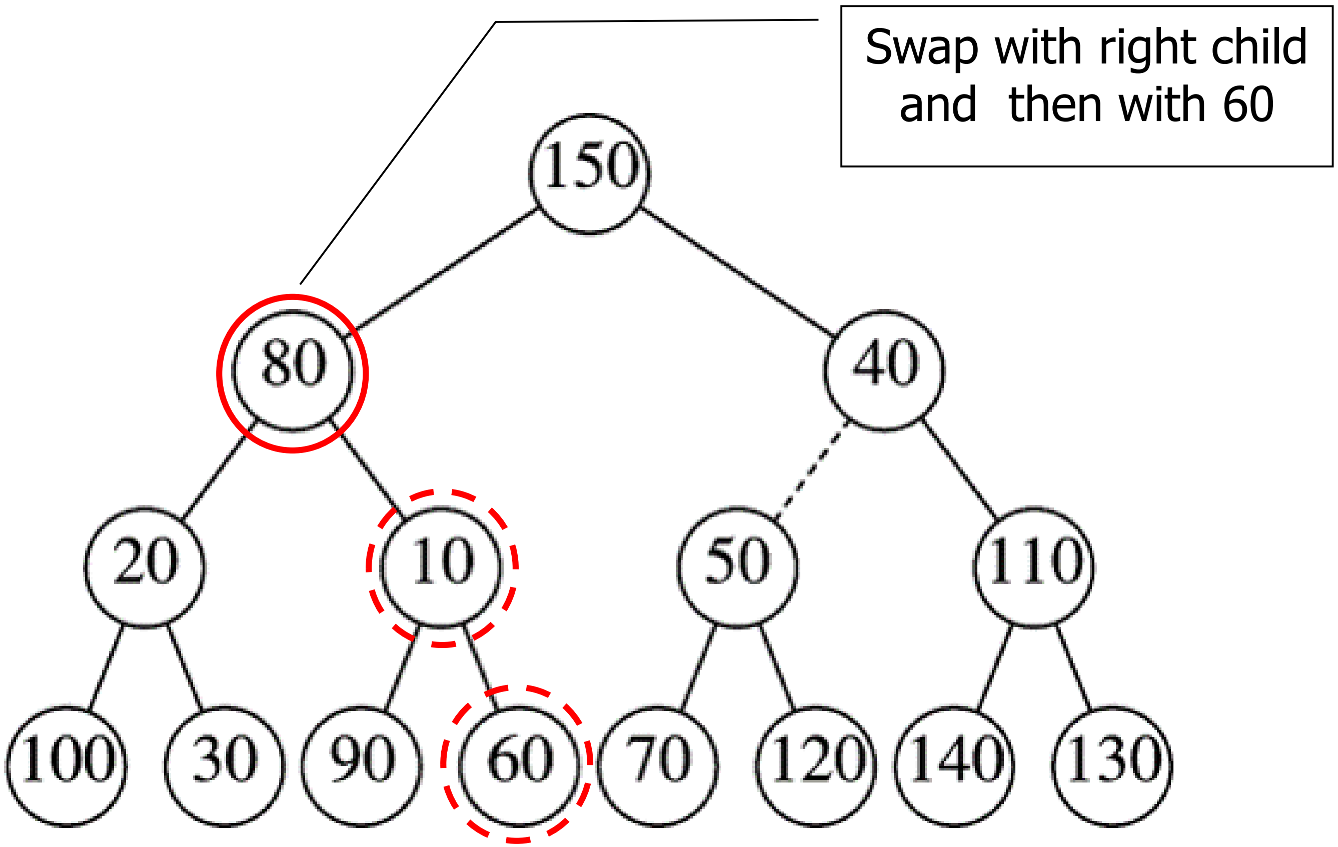
BuildHeap Example



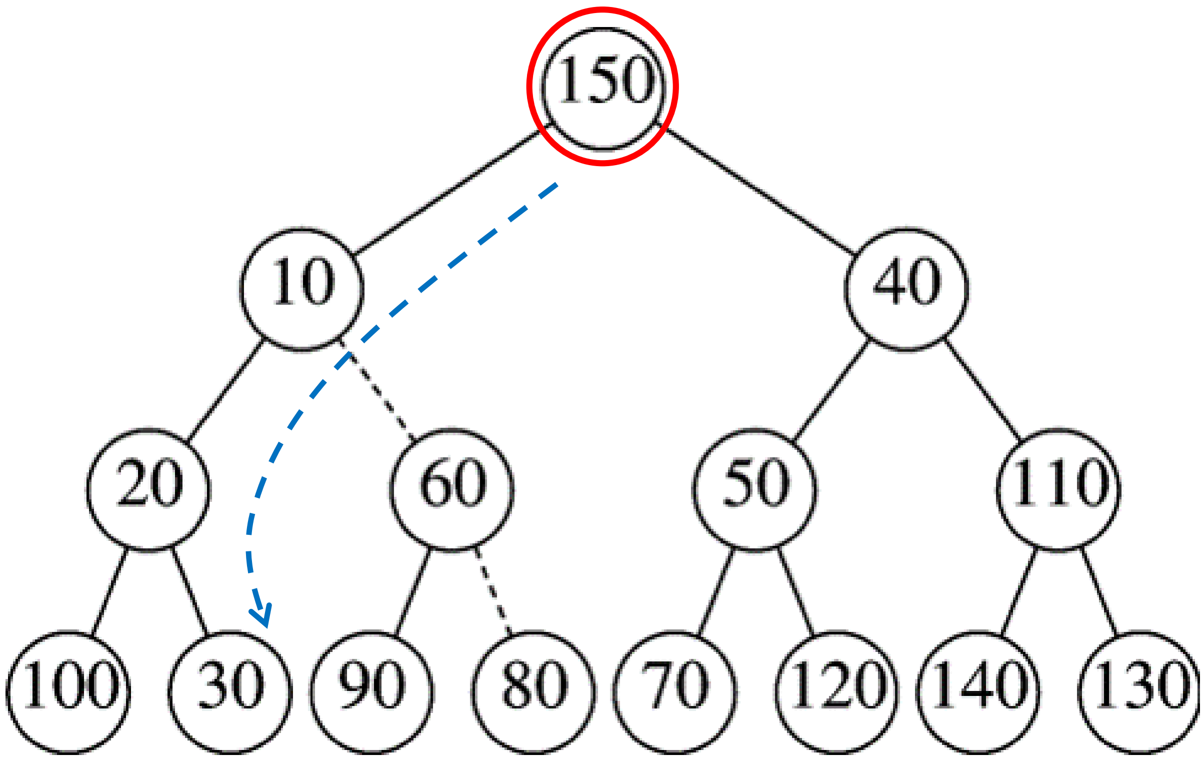
BuildHeap Example



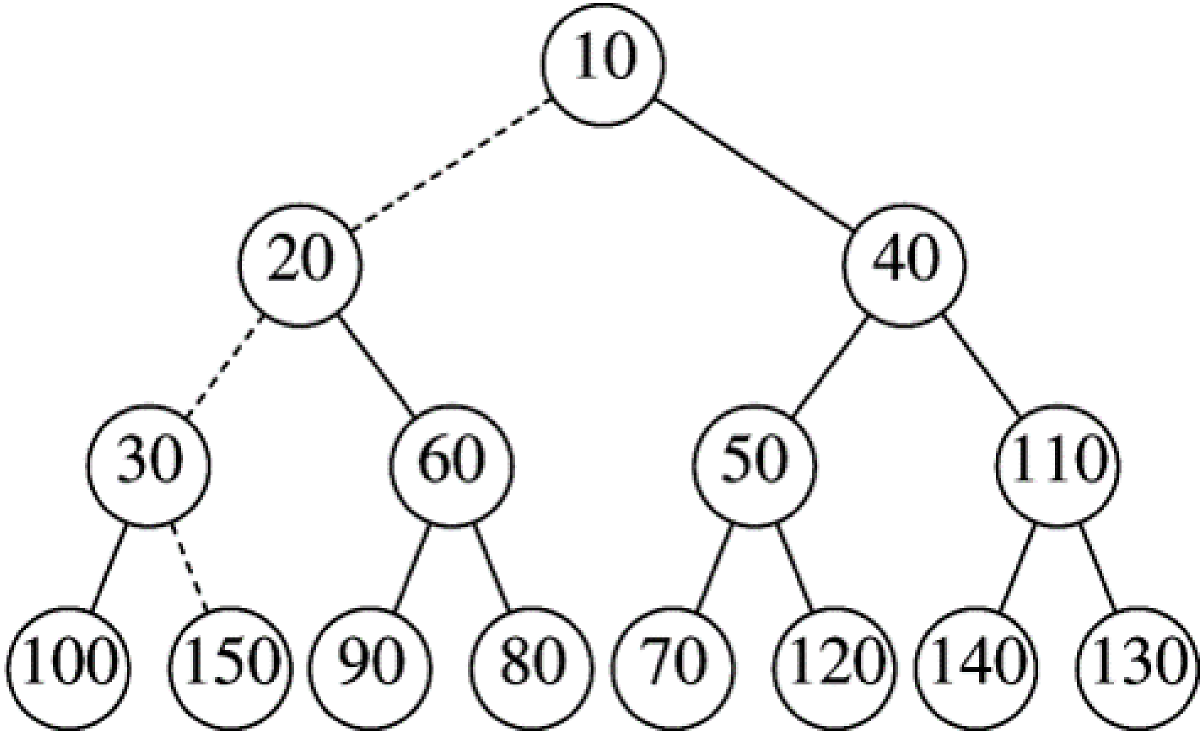
BuildHeap Example



BuildHeap Example



BuildHeap Example



Final Heap

Heap Sort

- Consists of two steps:
  - Build Heap
  - Delete elements one by one

- Algorithm:
  - Build a heap from the given input array
    - Heapify all non-leaf nodes
  - Repeat the following steps until the heap contains only one element
    - Swap the root element of the heap with the last element of the heap
    - Remove the last element of the heap
    - Heapify the remaining elements of the heap to maintain heap order

- Use max-heap for ascending sort and min-heap for descending sort

Heap Sort

```
void heapify(int arr[], int N, int i) {

    int largest = i;         // Initialize largest as root
    int l = 2 * i + 1;       // left = 2*i + 1
    int r = 2 * i + 2;       // right = 2*i + 2
    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;
    // If right child is larger than largest so far
    if (r < N && arr[r] > arr[largest])
        largest = r;
    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        // Recursively heapify the affected sub-tree
        heapify(arr, N, largest);
    }
}
```

Heap Sort

```
void heapSort(int arr[], int N) {
    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);
    // One by one extract an element from heap
    for (int i = N - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);
        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Any Question So Far?