DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

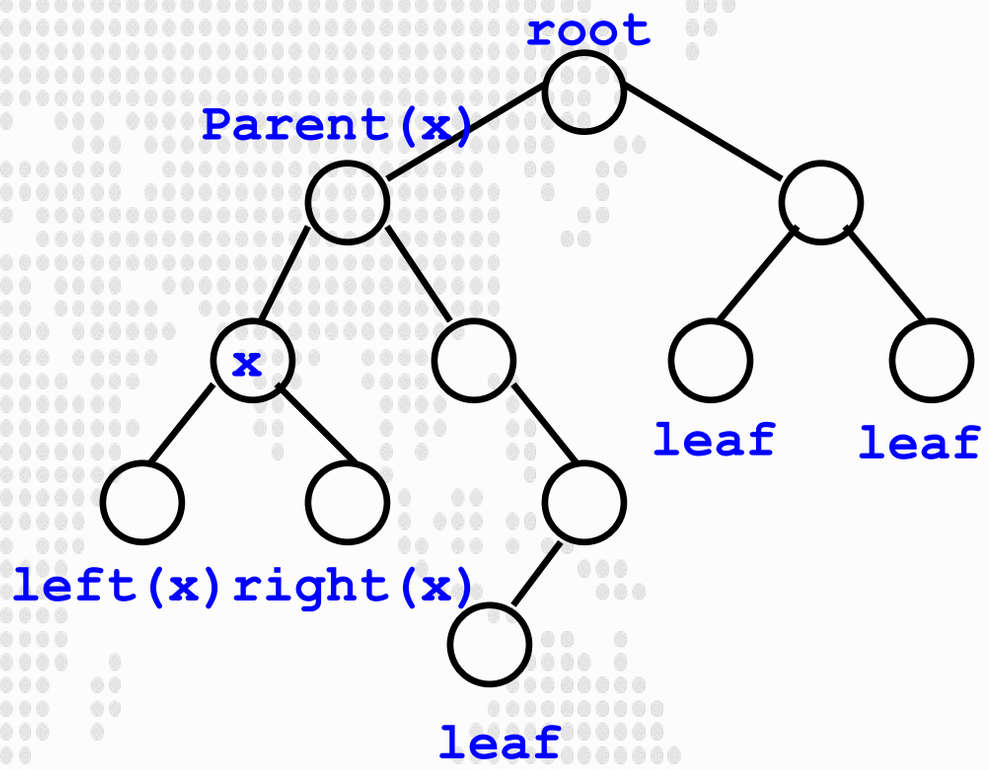
Lecture – 11
Heap, Heapsort, Priority Queue

Instructor: Engr. Nasr Kamal

Department of Computer Science and Information Technology

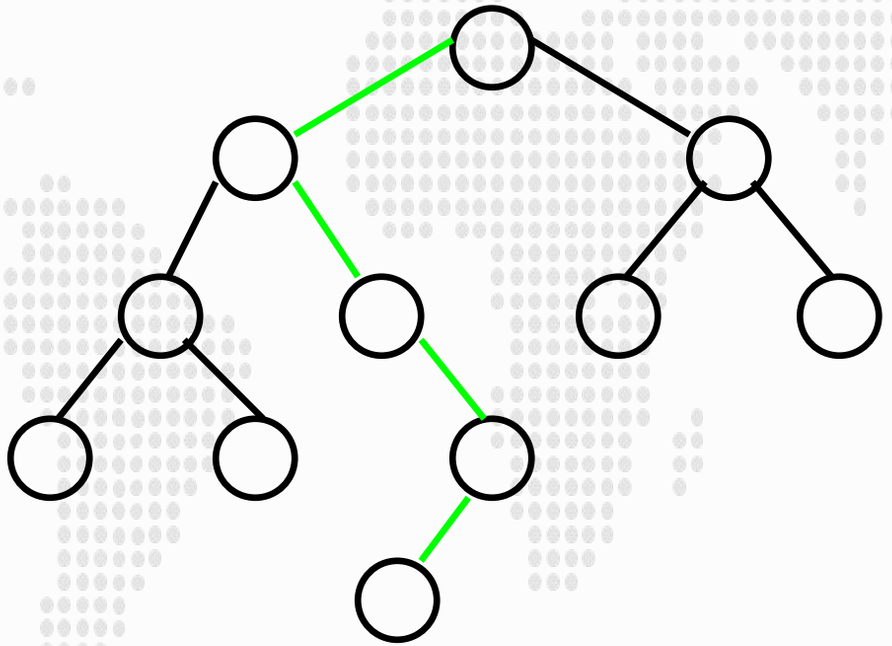
Background: Binary Trees

- Has a root at the topmost level
- Each node has zero, one or two children
- A node that has no child is called a leaf
- For a node x, we denote the left child, right child and the parent of x as left(x), right(x) and parent(x), respectively.



Height (Depth) of a Binary Tree

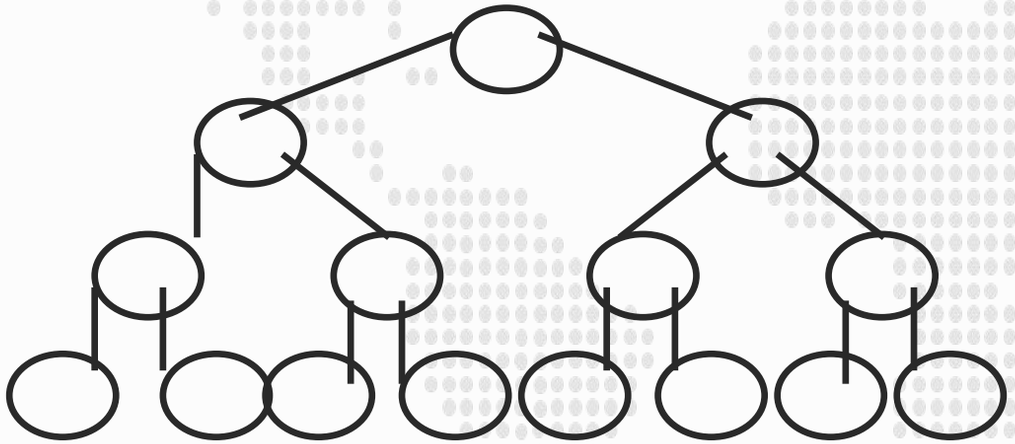
The number of edges on the longest path from the root to a leaf.



Height = 4

Background: Complete Binary Trees

- A complete binary tree is the tree
  - Where a node can have 0 (for the leaves) or 2 children and
  - All leaves are at the same depth



height	no. of nodes
0	1
1	2
2	4
3	8
d	2^d

- No. of nodes and height
  - A complete binary tree with N nodes has height O(logN)
  - A complete binary tree with height d has, in total, $2^{d+1}-1$ nodes

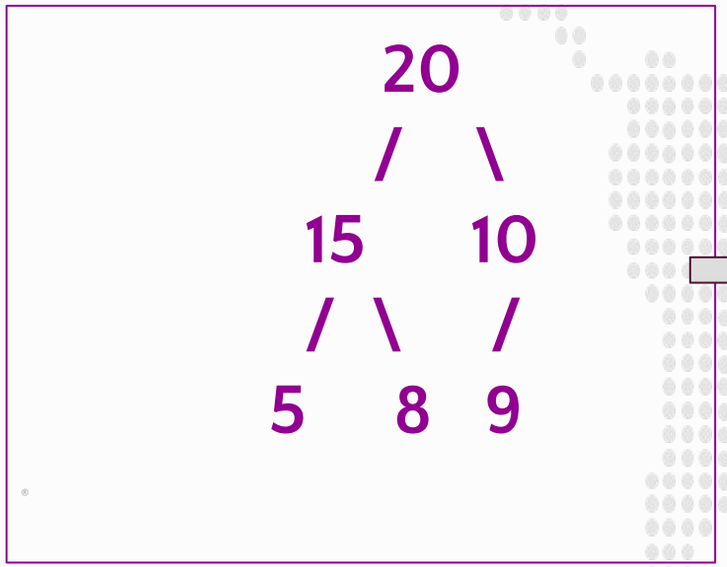
Proof: O(logN) Height

- Proof: a complete binary tree with **N** nodes has height of **O(logN)**
  1. Prove by induction that number of nodes at depth **d** is 2^d
  2. Total number of nodes of a complete binary tree of depth d is $\mathbf{1 + 2 + 4 + \ldots\ldots\ 2^d = 2^{d+1} - 1}$
  3. Thus $2^{d+1} - 1 = N$
  4. $\mathbf{d = \log(N+1) - 1 = O(\log N)}$

Binary Heap

A Heap is a specialized binary tree-based data structure that satisfies the heap property:

- **Max Heap**: Every parent node has a value greater than or equal to its children's values.
- **Min Heap**: Every parent node has a value less than or equal to its children's values.



Max Heap Verbal Representation of Tree

(Root Node is 20)
[(15 is left child node of 20)(5 is a left child node of 15)(8 is a right child node of 15)]
[(10 is right child node of 20)(9 is a left child node of 10)]

Properties and Applications

Properties:

- Complete binary tree: All levels are fully filled except possibly the last one, which is filled from left to right.
- Efficiently implemented using arrays.

Applications: Priority queues, scheduling, graph algorithms (like Dijkstra's shortest path), and heap sort.

Array Implementation of Binary Heap

Binary heap is a complete binary tree, nodes are filled level by level from left to right, which allows it to be efficiently represented in an array without needing pointers for left and right children. The indices in an array can represent the tree structure:

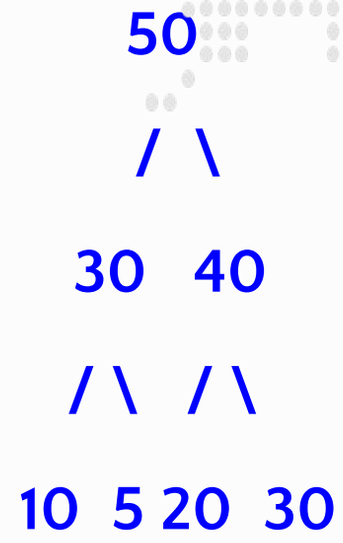
For a node at index iii:

- **Parent**: Located at index parent(i)=(i–1)/2
- **Left Child**: Located at index left(i)=2i+1
- **Right Child**: Located at index right(i)=2i+2\text{right}(i) = 2i + 2

These properties apply to both max-heaps and min-heaps

Array Implementation of Binary Heap

Example Binary Heap Tree (Max-Heap)



Max Heap Verbal Representation of Tree

(Root Node is 50)
[(30 is left child node of 50)(40 is a right child node of 15)]
[(10 is a left child node of 30)(5 is a right child node of 30)]
[(20 is left child node of 40)(30 is a left child node of 40)]

Conversion to Array

To convert this binary tree to an array, follow these steps:

1. **Start from the root** and assign it to the first index of the array.
2. Traverse the tree **level by level, from left to right**. This is also known as a **breadth-first traversal**.

Array Implementation of Binary Heap

Left Child: Located at index 2i+1

Right Child: Located at index 2i+2

Parent: Located at index (i–1)/2

Root (index 0):

- Value: 50
- Left Child: Index 1 (value 30)
- Right Child: Index 2 (value 40)

Node at index 1:

- Value: 30
- Left Child: Index 3 (value 10)
- Right Child: Index 4 (value 5)
- Parent: Index 0 (value 50)

Node at index 2:

- Value: 40
- Left Child: Index 5 (value 20)
- Right Child: Index 6 (value 30)
- Parent: Index 0 (value 50)

[50, 30, 40, 10, 5, 20, 30]

Heap Operations

Insert (Push):

- Insert the element at the end of the heap and then "heapify up" or "bubble up" to maintain the heap property.

Delete (Pop):

- Typically removes the root element (maximum in a max heap, minimum in a min heap).
- Swap the root with the last element, remove the last element, and "heapify down" to maintain the heap structure.

Heapify:

- Also known as "sift-down" or "bubble-down".
- Applied recursively to maintain the heap property when an element is added or removed

Pseudo-Code to Insert and Delete in Min Heap

Insert (value):

1. Add the new value at the end of the heap.
2. While the new value is lesser than its parent, swap them.

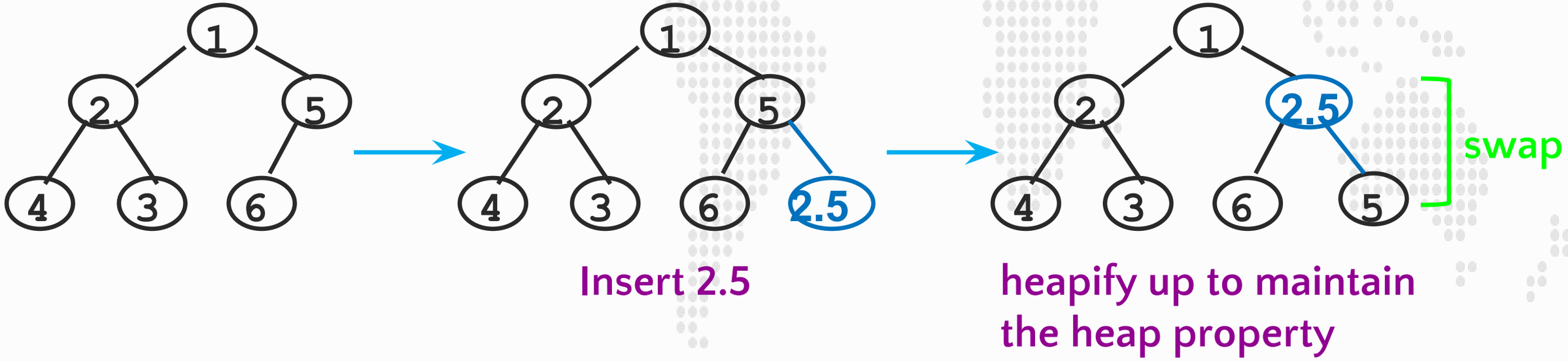
Delete (Min):

1. Swap the root with the last element.
2. Remove the last element.
3. While the new root is greater than either of its children, swap it with the smaller child.

Insertion

Algorithm

1. Add the new element to the next available position at the lowest level
2. Restore the min-heap property if violated
   - General strategy is "heapify up" (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.

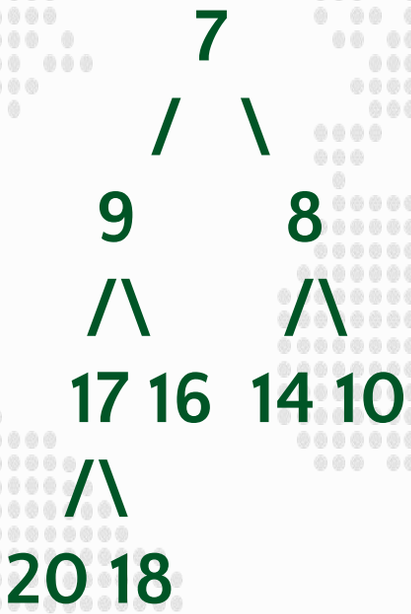


Insertion

```
// Insert a new element into the heap
 void insert(int value) {
heap.push_back(value); // Add at the
    end
heapifyUp(heap.size() - 1); // Restore
    heap property by moving up
}
```

```
void heapifyUp(int index)
{ while (index > 0)
 { int parent = (index - 1) / 2;
if (heap[index] < heap[parent])
 { // Min-heap property violated
swap(heap[index], heap[parent]);
index = parent;
}
else { break;
}}}
```

Insertion Complexity



A heap!

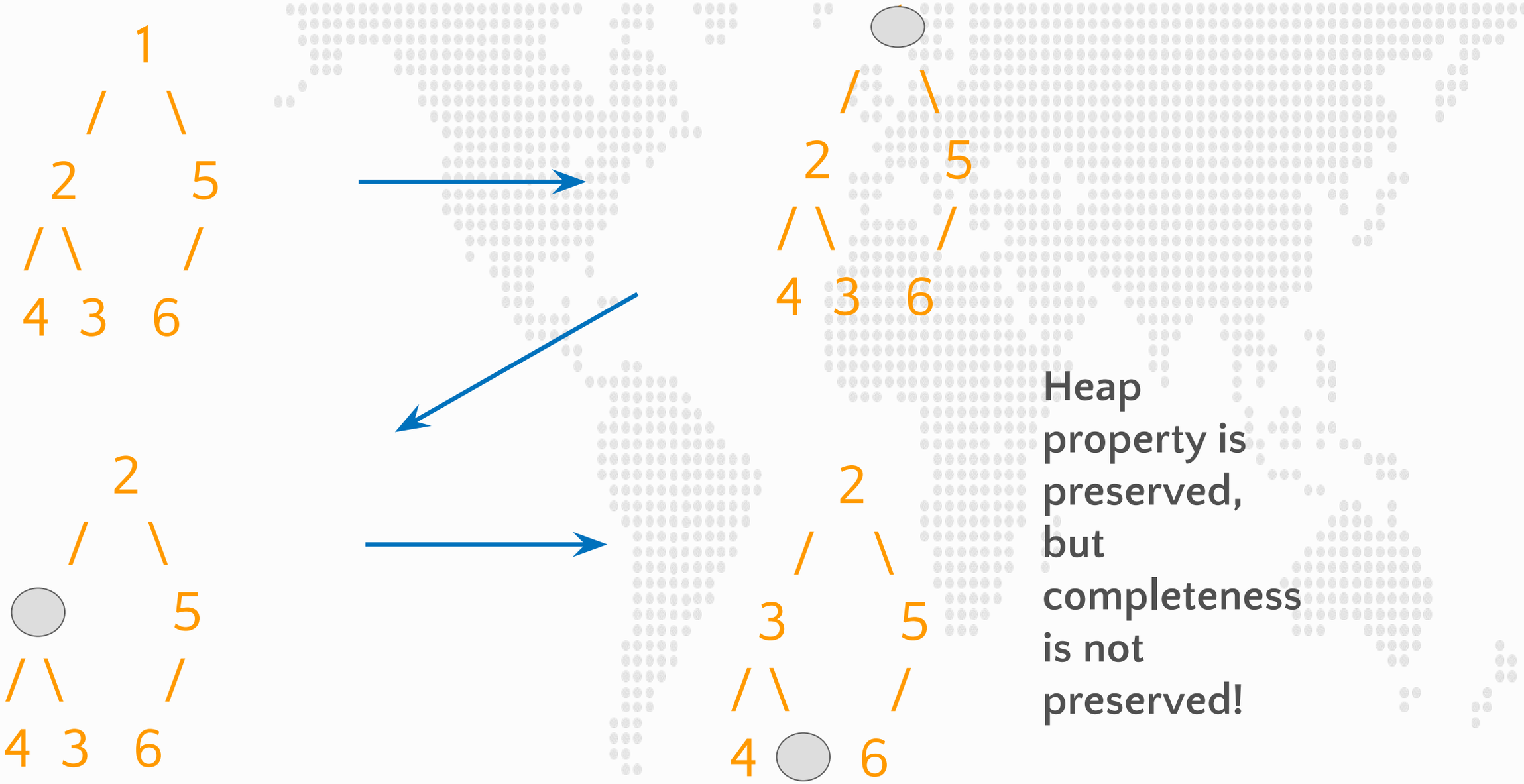
Time Complexity = O(height) = **O(logN)**

deleteMin: First Attempt

Algorithm

1. Delete the root.
2. Compare the two children of the root
3. Make the lesser of the two the root.
4. An empty spot is created.
5. Bring the lesser of the two children of the empty spot to the empty spot.
6. A new empty spot is created.
7. Continue

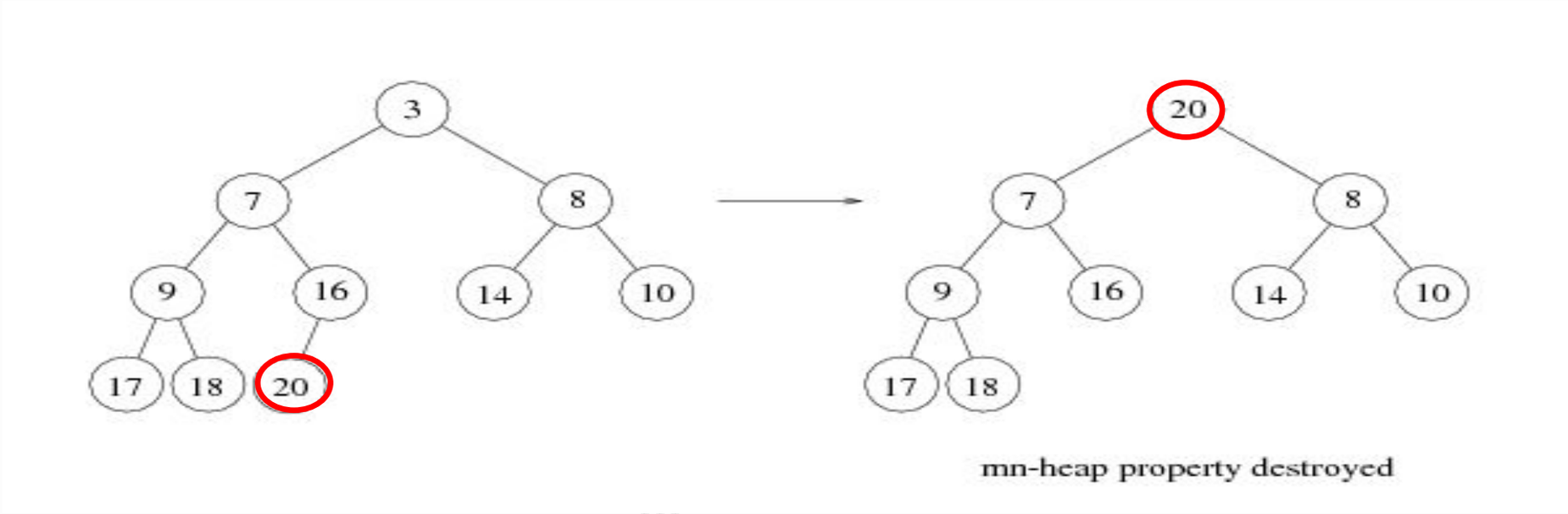
Example for First Attempt

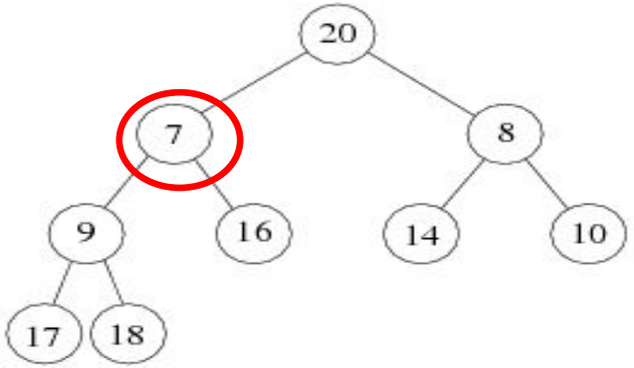


Heap property is preserved, but completeness is not preserved!

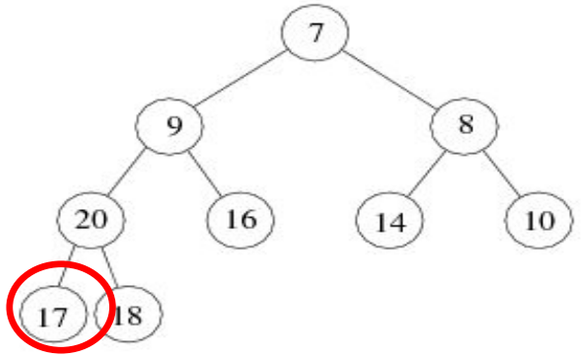
deleteMin

1. Copy the last number to the root (i.e. overwrite the minimum element stored there)
2. Restore the min-heap property by sift down (or bubble down)

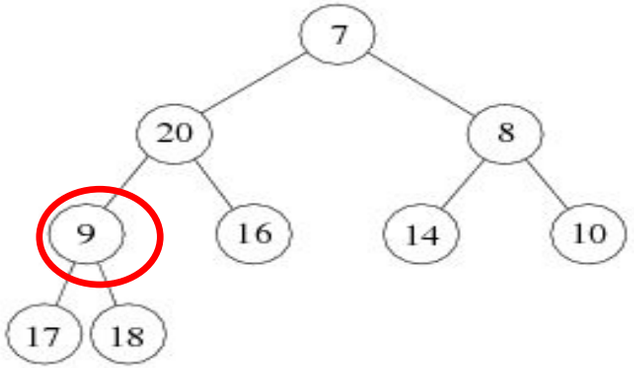




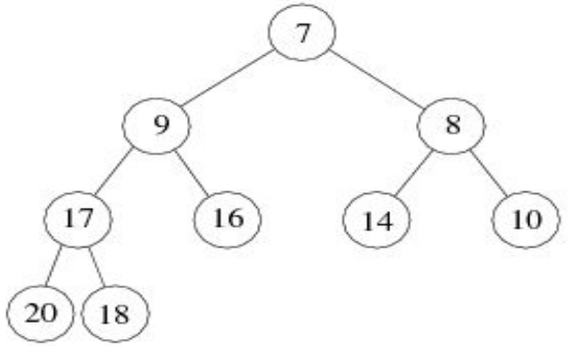
compare 20 with 7, the smallest of its two children



compare 20 with 17, the smallest of its two children



compare 20 with 9, the smallest of its two children



A heap!
Time complexity
= O(height) = O(log n)

deleteMin

```
// Remove and return the minimum element (root of the heap)
int extractMin() {
if (heap.empty())
{
cout << "Heap is empty!\n";
return -1;
// Or some error code
}
int minVal = heap[0]; // The root value (minimum)
heap[0] = heap.back(); // Move the last element to the root
heap.pop_back(); // Remove the last element
heapifyDown(0); // Restore heap property by moving down return minVal; }
```

deleteMin

```
void heapifyDown(int index) {
int size = heap.size();
while (index < size) {
 int left = 2 * index + 1;
int right = 2 * index + 2;
int smallest = index;
if (left < size && heap[left] < heap[smallest])
 smallest = left;
if (right < size && heap[right] < heap[smallest])
  smallest = right;
  if (smallest != index) { // Min-heap property violated
  swap(heap[index], heap[smallest]);
  index = smallest; }
  else {
  break;
  } } }
```

Heapsort

A comparison-based sorting technique based on a binary heap. It can be applied using either max-heaps (for ascending order) or min-heaps (for descending order)

(1) Build a binary heap of N elements

- the minimum element is at the top of the heap

(2) Perform N DeleteMin operations

- the elements are extracted in sorted order

(3) Record these elements in a second array and then copy the array back

Heapsort – Running Time Analysis

(1) Build a binary heap of N elements

- repeatedly insert N elements ⇒ O(N log N) time

(2) Perform N DeleteMin operations

- Each DeleteMin operation takes O(log N) ⇒ O(N log N)

(3) Record these elements in a second array and then copy the array back

- O(N)

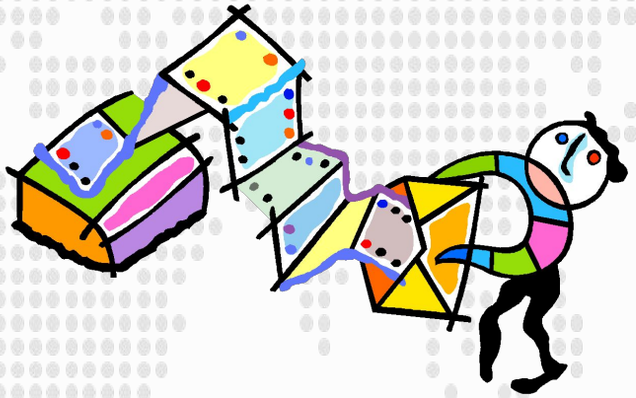
Total time complexity: O(N log N)

Memory requirement: uses an extra array, O(N)

Priority Queue: Motivating Example

3 jobs have been submitted to a printer in the order A, B, C.

Sizes: Job A – 100 pages
Job B – 10 pages
Job C -- 1 page



Average waiting time with FIFO service:

(100+110+111) / 3 = 107 time units

Average waiting time for shortest-job-first service:

(1+11+111) / 3 = 41 time units

A queue be capable to **insert** and **deletemin**?

Priority Queue

Priority Queue

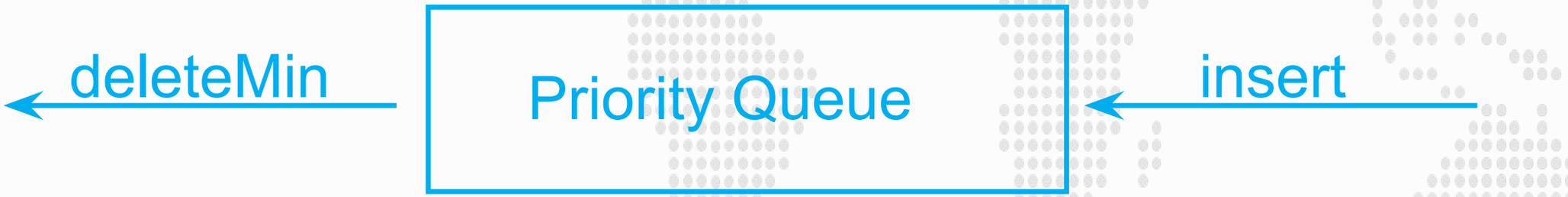
A priority queue is an abstract data structure that operates similarly to a queue but with an element priority. Elements with higher priority are served before elements with lower priority.

Implementation:

- Implemented using a heap (min-heap for minimum priority, max-heap for maximum priority).
- Common operations are insert, peek, and remove.

Priority Queue

- Priority queue is a data structure which allows at least two operations
  - **insert**
  - **deleteMin**: finds, returns and removes the minimum elements in the priority queue



Priority Queue Operations

Insert: Insert an element with a priority, which involves adding the element and rearranging it according to the priority.

Peek: Return the element with the highest priority (root in the case of a max-heap).

Remove: Remove the element with the highest priority and re-heapify to maintain order.

Priority Queue Implementation in C++

```
#include <queue>
int main() {
    // Max-Heap
    std::priority_queue<int> maxHeap;
    maxHeap.push(10);
    maxHeap.push(20);
    maxHeap.push(5);
```

```
    while (!maxHeap.empty()) {
        std::cout << maxHeap.top() << " ";
        maxHeap.pop();
    }

    return 0;
}
```


It’s a Heap!!!