



Priority Queues (Heaps)

Dr. Abdallah Karakra

Computer Science Department

COMP242

Motivation

10	4	8	5	12	2	6	11	3	9	7	1
----	---	---	---	----	---	---	----	---	---	---	---

Queue ADT

Remove element that **first come to the queue.**



PRIORITY QUEUES

A priority queue is a queue where:

- Requests are inserted in the order of arrival
- The request with highest priority is processed first (deleted from the queue)

PRIORITY QUEUES

Removal based on the criteria (e.g. Hospital)

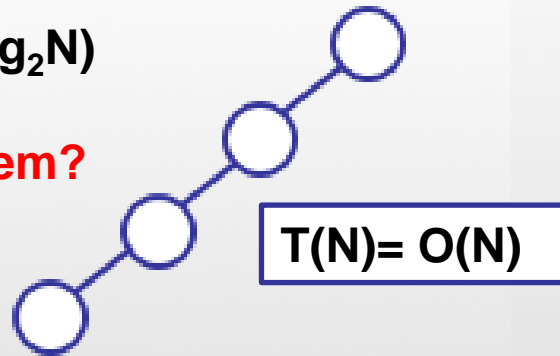
Representation (Implementation):

1. Sorted List: for example linked list, array $\rightarrow T(N) = O(N)$

What is the problem?

2. BST $\rightarrow T(N) = O(\log_2 N)$

What is the problem?



3. Heap

Heap ADT

Heap ADT:

Heap data structure is an array object that can be viewed as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

- **Complete Binary Tree**
- **If height h , all levels $< h$ are full, all nodes at h to the left**
- **Nodes are comparable**

Heap ADT

In a max - heap, every node i other than the root satisfies the following property :

$$A[\text{Parent}(i)] \geq A[i].$$

In a min - heap, every node i other than the root satisfies the following property :

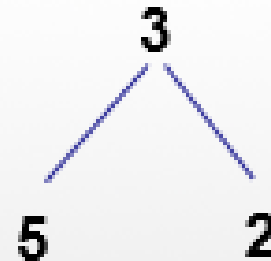
$$A[\text{Parent}(i)] \leq A[i].$$

Heap ADT

Is the below max or min heap?

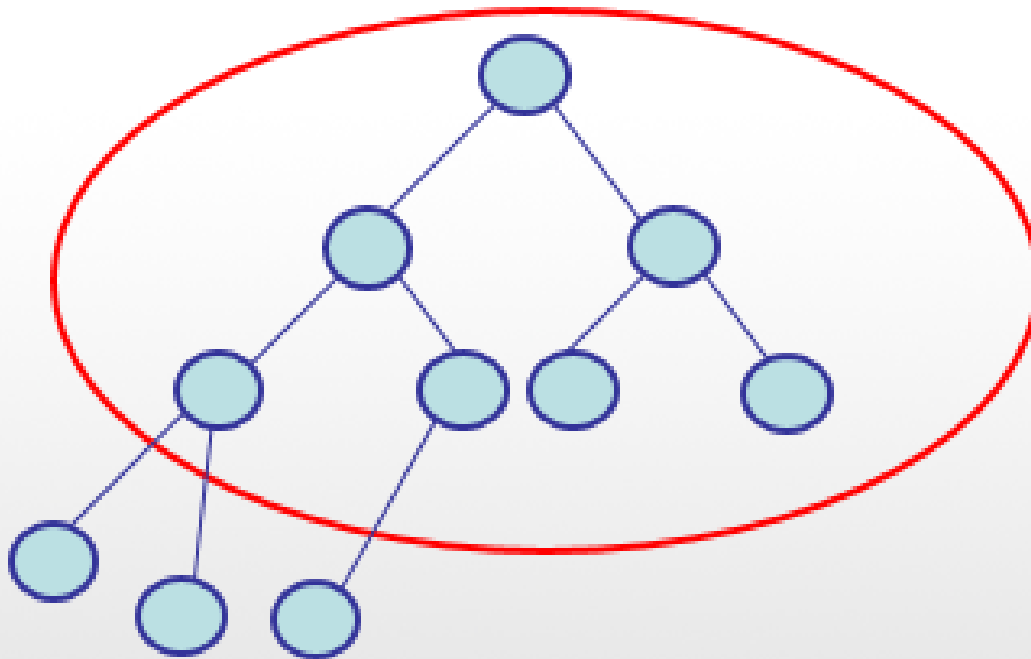
3	5	2
---	---	---

- Not a min heap
- Not a max heap



Heap

Is a complete Binary Tree



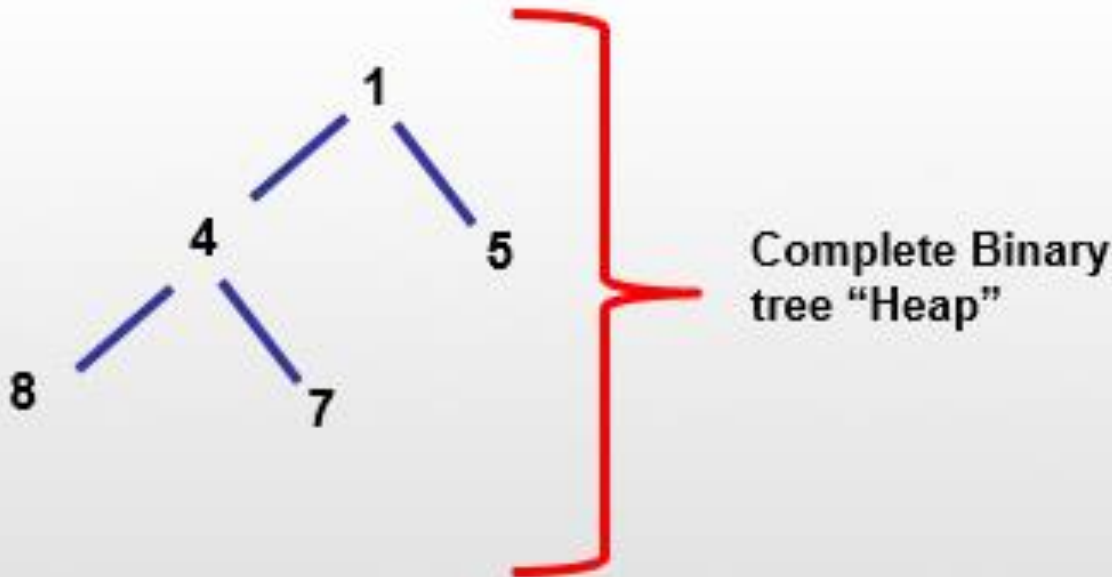
heap solve the long bath that may happen in BST

Heap : Implementation

Min Heap

Add

1. Add at level h (or $h+1$ if full)
2. Recursive swap with parent if less

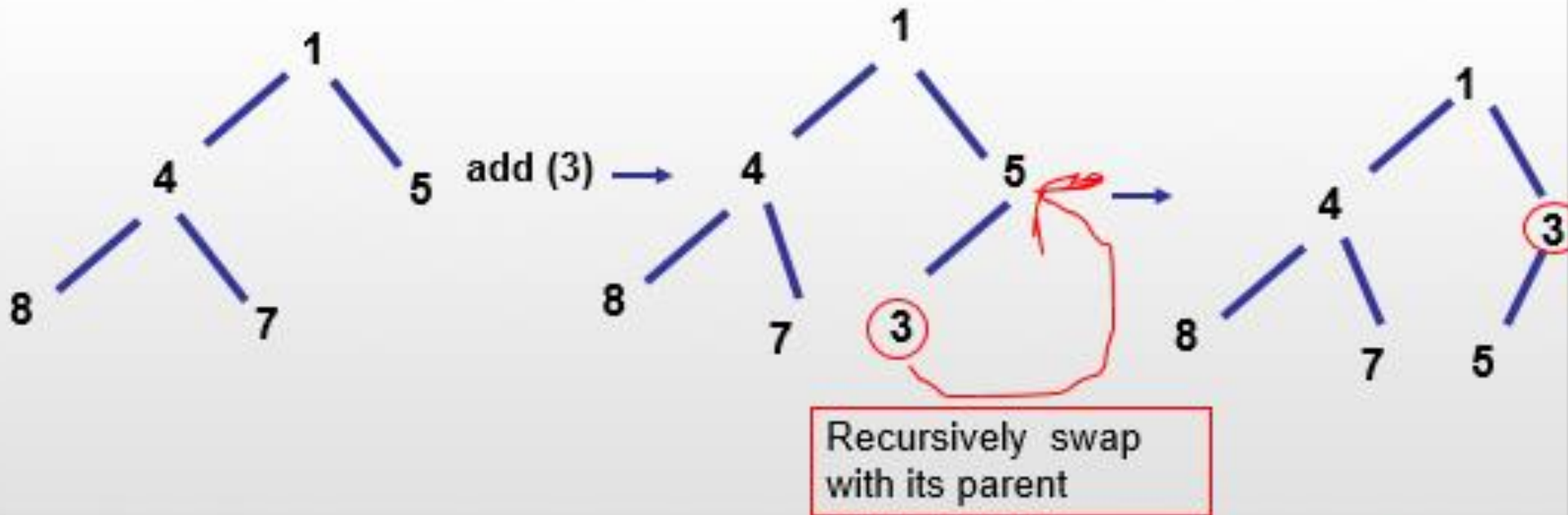


Heap : Implementation

Min Heap

Add

1. Add at level h (or h+1 if full)
2. Recursive swap with parent if less



Heap

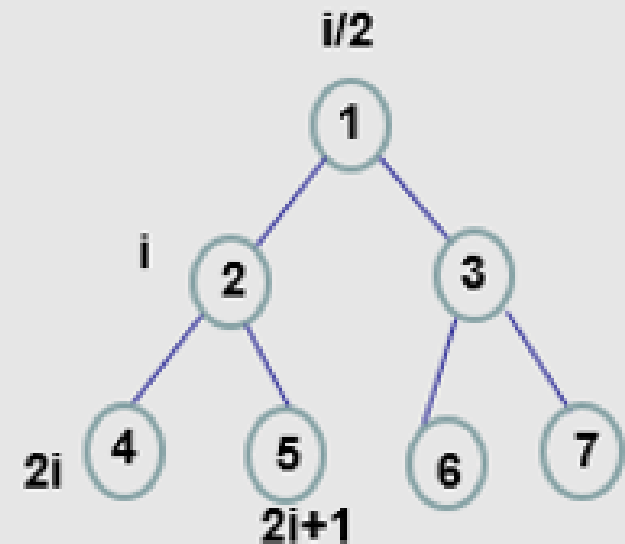
Note: If a node's index is i , its children's indices are $2i$ and $2i+1$, and its parent's index is $\text{floor}(i/2)$

Ex:

let $i = 2$ (Parent)

$2i \rightarrow 2 \times 2 = 4 \rightarrow$ first child

$2i+1 \rightarrow 2 \times 2 + 1 = 5 \rightarrow$ second child

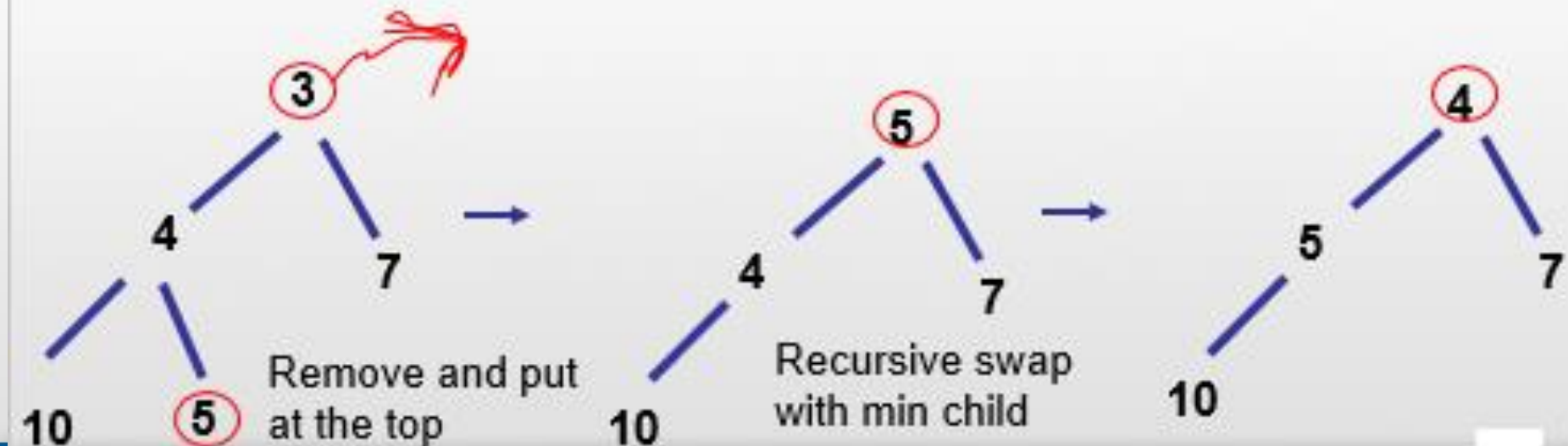
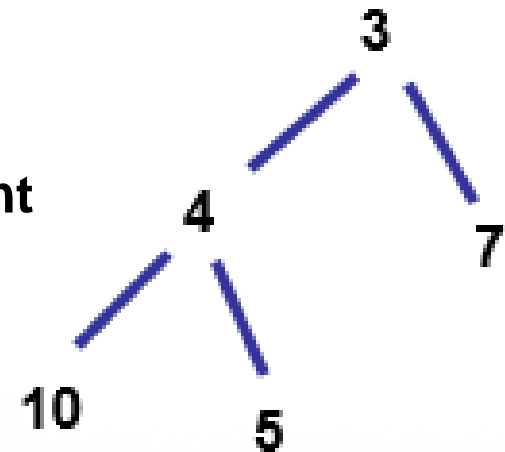


1	2	3	4	5	6	7
root	parent		child1	child2		

Heap : Implementation

Remove

1. Remove root and fill hole with last element
2. Recursive swap with min child if larger



Max Heap

In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

Implementation

```
MAX-HEAPIFY(A,i)
```

```
  l = LEFT(i)
```

```
  r = RIGHT(i)
```

```
  if l <= heapSize and A[l] > A[i]
```

```
    largest = l
```

```
  else
```

```
    largest = i
```

```
  if r <= heapSize and A[r] > A[largest]
```

```
    largest = r
```

```
  if largest != i
```

```
    exchange A[i] with A[largest]
```

```
    MAX-HEAPIFY(A,largest)
```

Complexity:
 $O(\log N)$

$A[i]$ – value at node (element) i

$LEFT(i)$ – index of left child node = $2i$

$RIGHT(i)$ – index of right child node = $2i + 1$

$PARENT(i)$ – index of parent node = $i / 2$

heapSize- actual number of elements in the heap

Building MAX HEAP

```
BUILD-MAX-HEAP(A)
for i = heapSize/2 downto 1
    MAX-HEAPIFY(A,i)
```

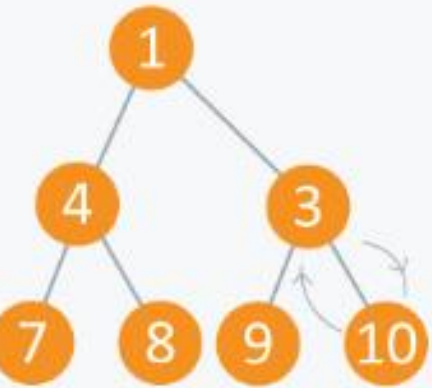

Example:

Suppose you have 7 elements stored in array Arr.

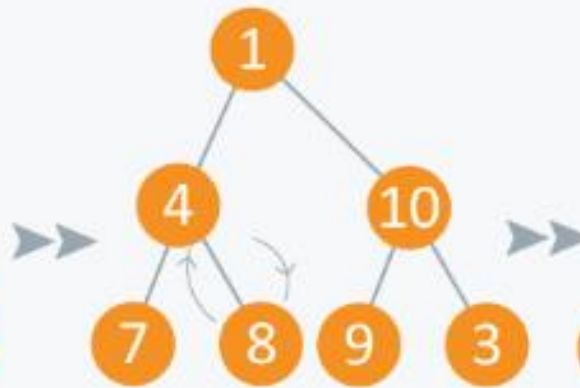
Arr		1	4	3	7	8	9	10
	0	1	2	3	4	5	6	7

Arr		1	4	3	7	8	9	10
	0	1	2	3	4	5	6	7

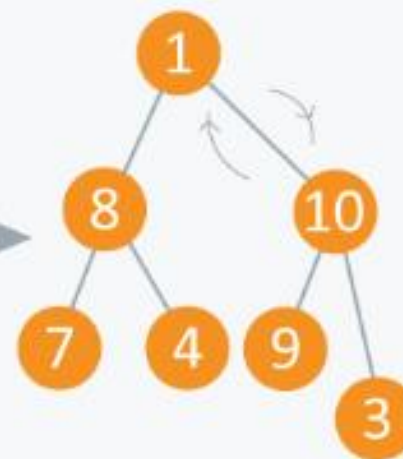
Step 1



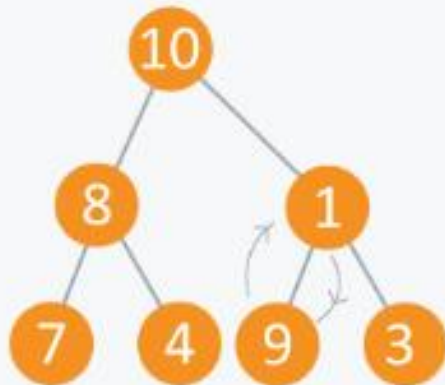
Step 2



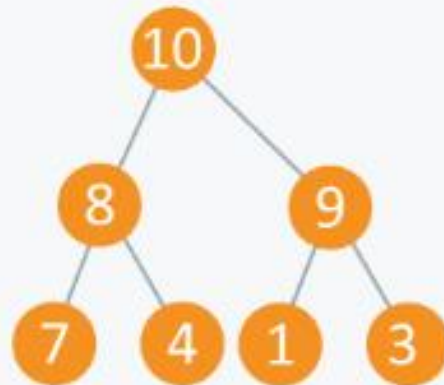
Step 3



Step 4



Step 5



```
BUILD-MAX-HEAP(A)
for i = heapSize/2 downto 1
    MAX-HEAPIFY(A,i)
```

```
MAX-HEAPIFY(A,i)
l = LEFT(i)
r = RIGHT(i)
if l <= heapSize and A[l] > A[i]
    largest = l
else
    largest = i
if r <= heapSize and A[r] > A[largest]
    largest = r
if largest != i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

Implementation

we finally get a max- heap and the elements in the array Arr will be :

Arr		10	8	9	7	4	1	3
	0	1	2	3	4	5	6	7

Implementation

Complexity:
 $O(\log N)$

```
MIN-HEAPIFY(A,i)
  l = LEFT(i)
  r = RIGHT(i)
  if l <= heapSize and A[l] < A[i]
    smallest = l
  else
    smallest = i
  if r <= heapSize and A[r] < A[smallest]
    smallest = r
  if smallest != i
    exchange A[i] with A[smallest]
  MIN-HEAPIFY (A,smallest)
```

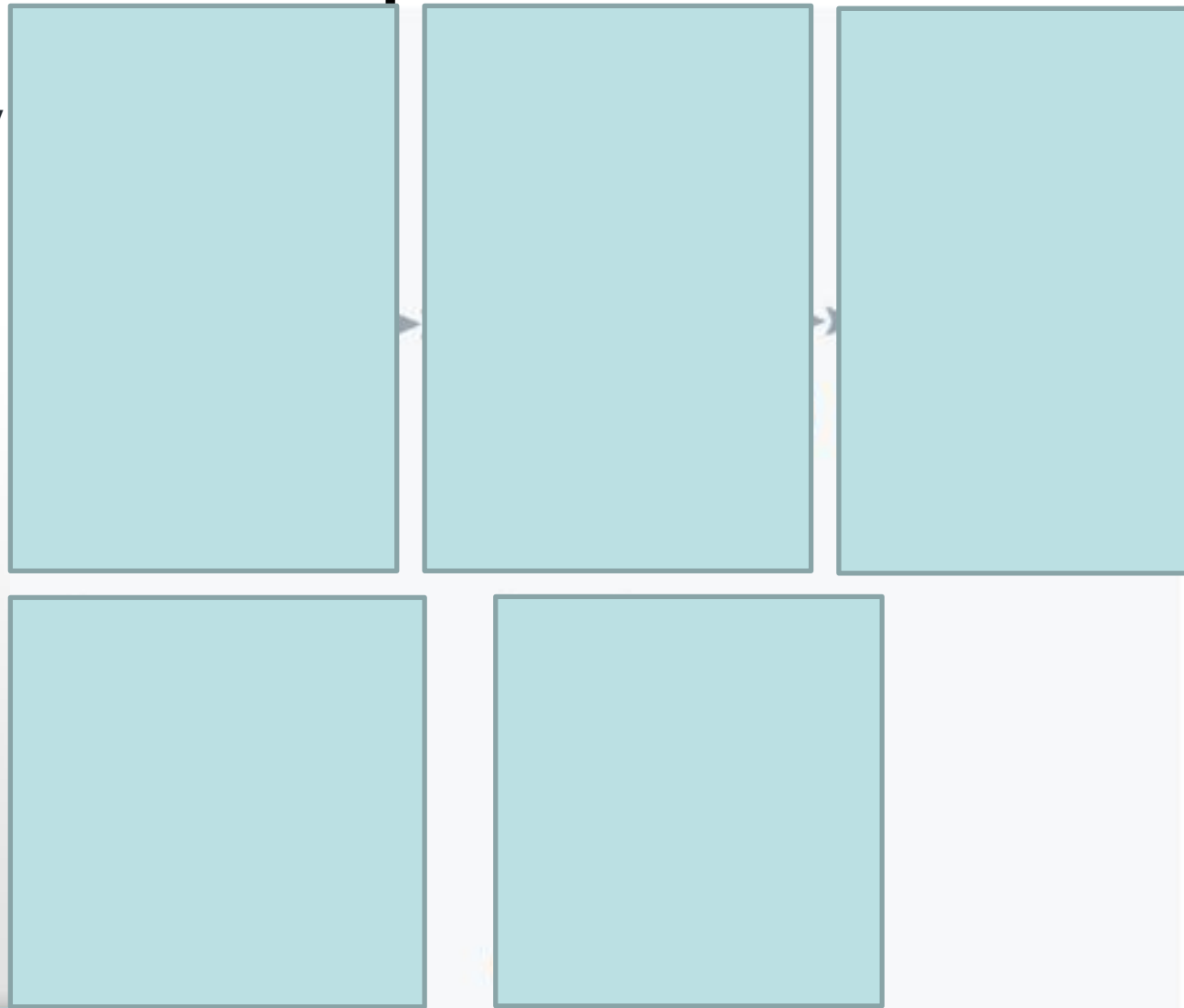
$A[i]$ – value at node (element) i
 $LEFT(i)$ – index of left child node = $2i$
 $RIGHT(i)$ – index of right child node = $2i + 1$
 $PARENT(i)$ – index of parent node = $i / 2$
heapSize- actual number of elements in the heap

Building MIN HEAP

```
BUILD-MIN-HEAP(A)
for i = heapSize/2 downto 1
    MIN-HEAPIFY(A,i)
```

Example

Consider elements in array
{10, 8, 9, 7, 6, 5, 4}. We will
run min_heapify on nodes
indexed from $N/2$ to 1.



Implementation:

length = number of elements in Arr.

Maximum :

```
// the max element is the root element in the max  
Heap-Max(A)
```

```
return A[1] heap
```

Complexity: $O(1)$

Implementation:

Extract Maximum: In this operation, the maximum element will be returned and the last element of heap will be placed at index 1 and max_heapify will be performed on node 1 as placing last element on index 1 will violate the property of max-heap.

```
Heap-Extract-Max(A)
  max=A[1]
  A[1]=A[heapSize]
  heapSize=heapSize-1
  Max-Heapify(A,1)
  return max
```

Complexity: $O(\log N)$.

Implementation:

Insert Value: In case increasing value of any node, may violate the property of max-heap, so we will swap the parent's value with the node's value until we get a larger value on parent node

```
Max-Heap-Insert(A, value)
  heapSize = heapSize+1
  i=heapSize
  A[i]=value
  while i>1 and A[i]>A[i/2]
    exchange A[i] with A[i/2]
    i=i/2
```

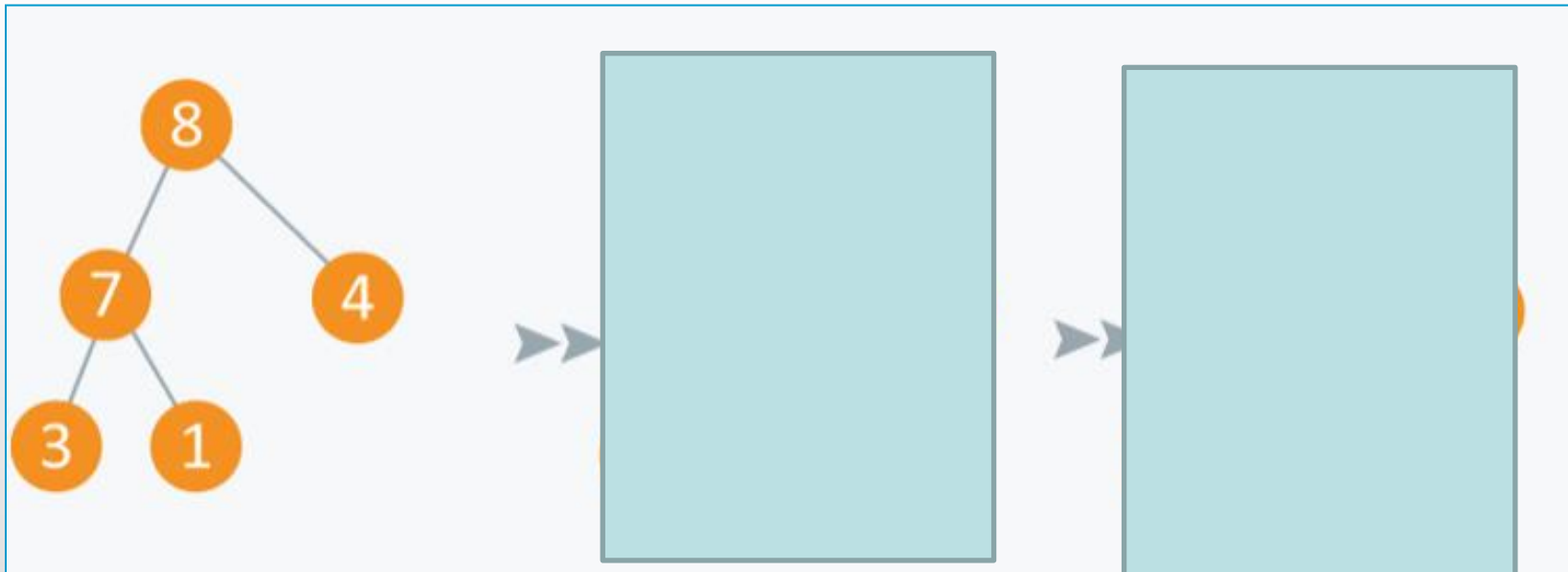
Complexity : $O(\log N)$.

Max Heap: insert value

Initially there are 5 elements in priority queue.

Operation: Insert Value(Arr, 6)

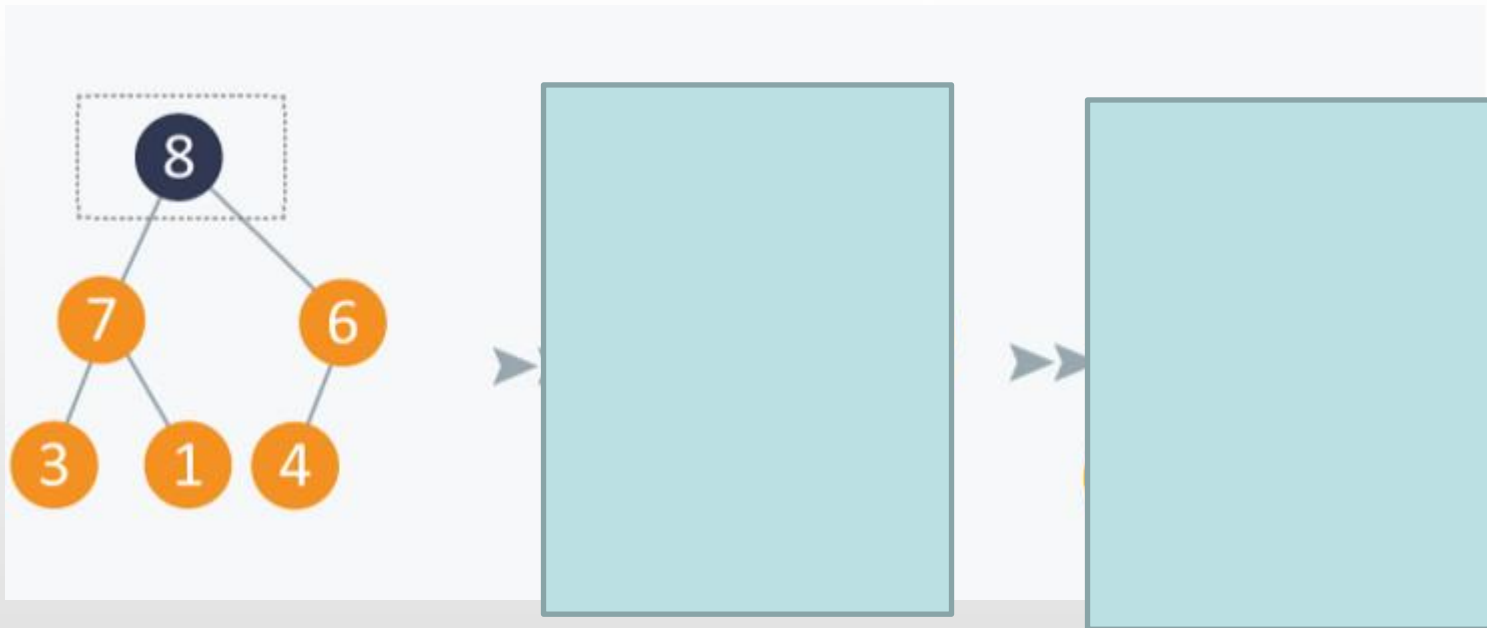
In the diagram below, inserting another element having value 6 is violating the property of max-priority queue, so it is swapped with its parent having value 4, thus maintaining the max priority queue.



Max Heap: delete value

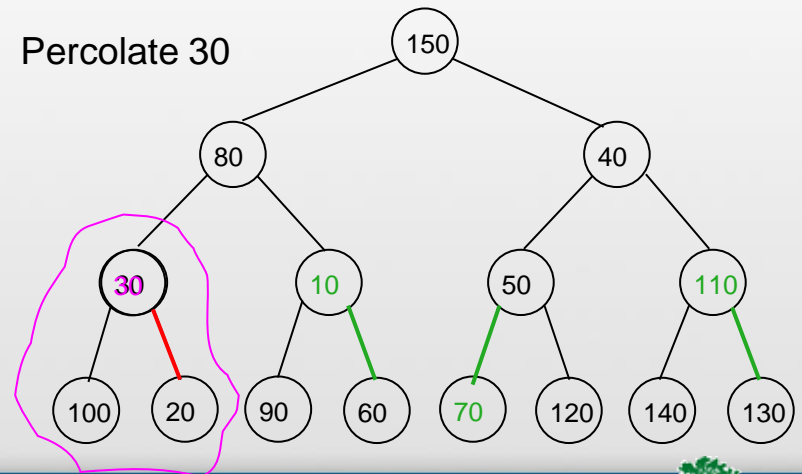
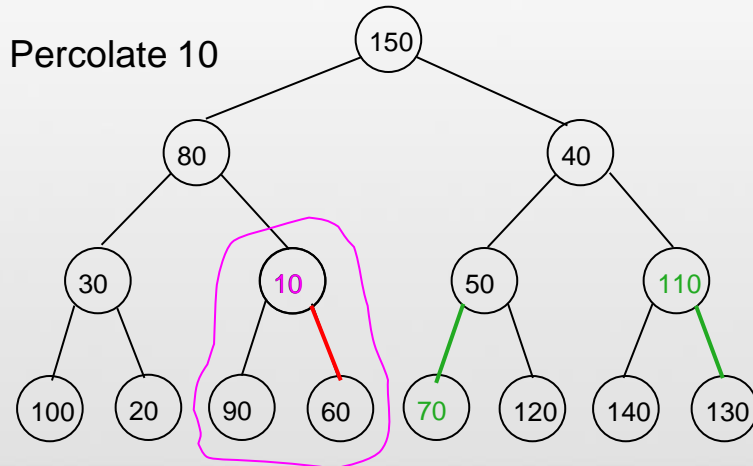
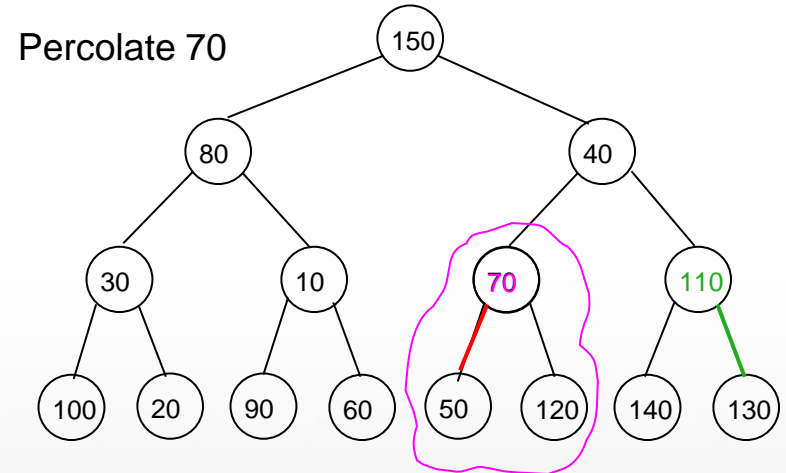
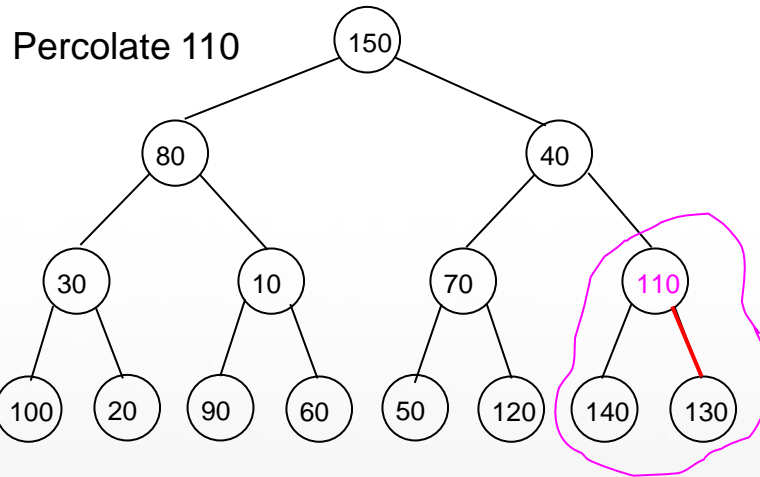
Operation: Extract Maximum:

In the diagram below, after removing 8 and placing 4 at node 1, violates the property of max-priority queue. So `max_heapify(Arr, 1)` will be performed which will maintain the property of max - priority queue



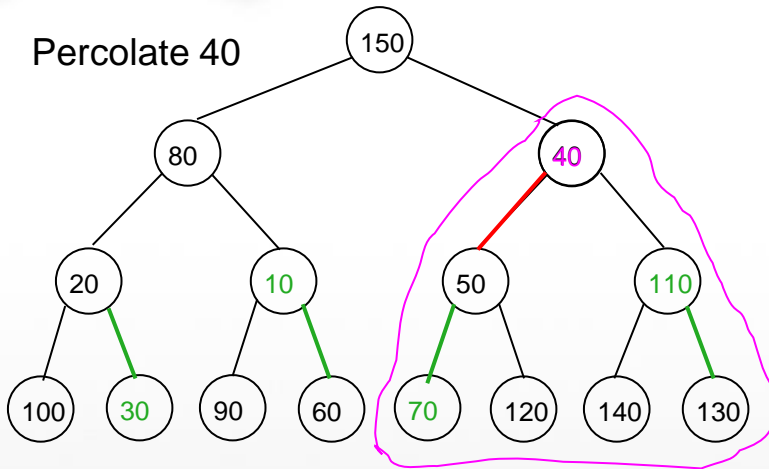
Build a min-heap

- Input: 150, 80, 40, 30, 10, 70, 110, 100, 20, 90, 60, 50, 120, 140, 130
 - apply percolate down for each node starting at the last non-leaf node

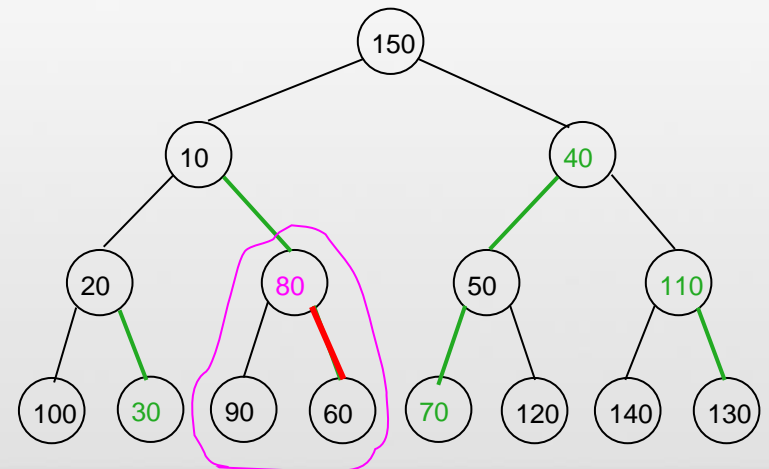
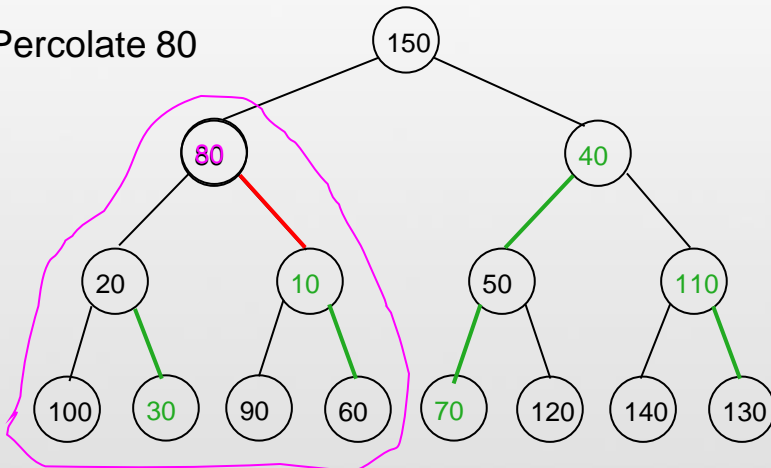


Build a min-heap

Percolate 40

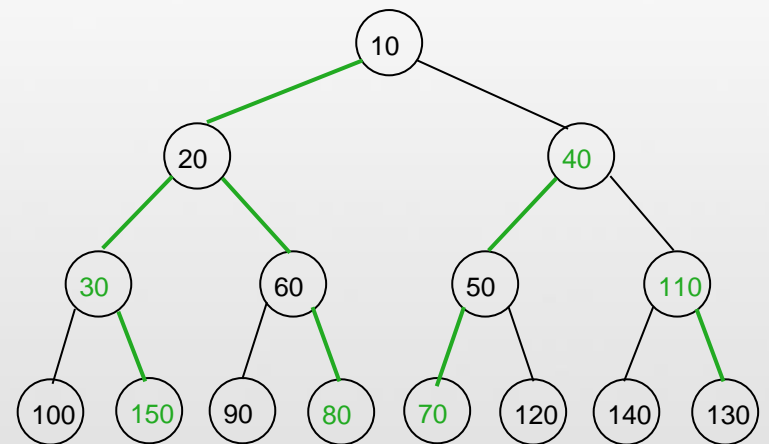
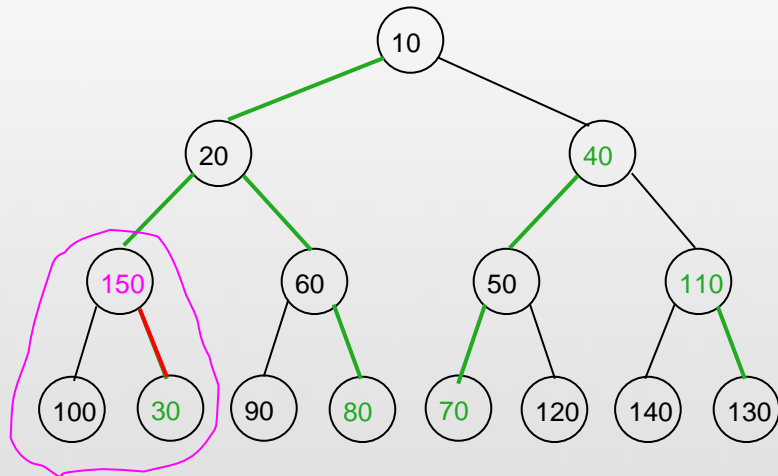
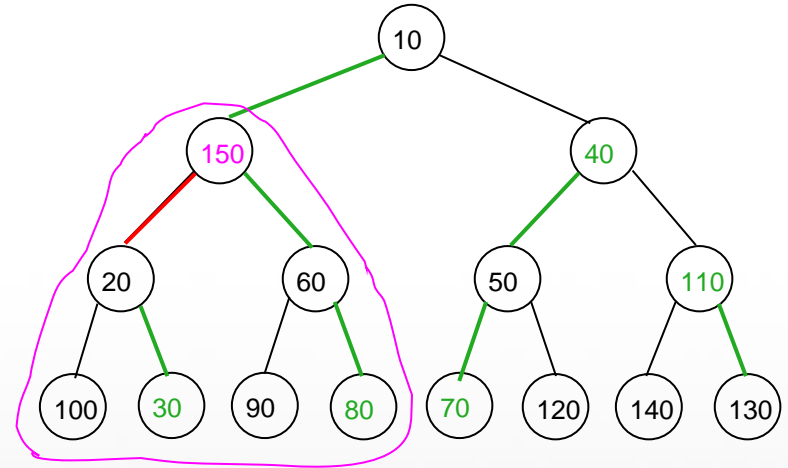
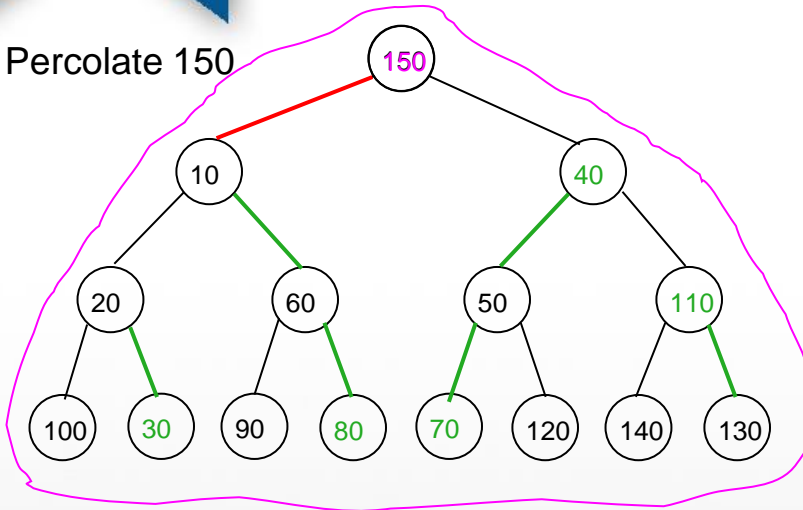


Percolate 80



Build a min-heap

Percolate 150



Binary Heap : H.W



Problem 1:

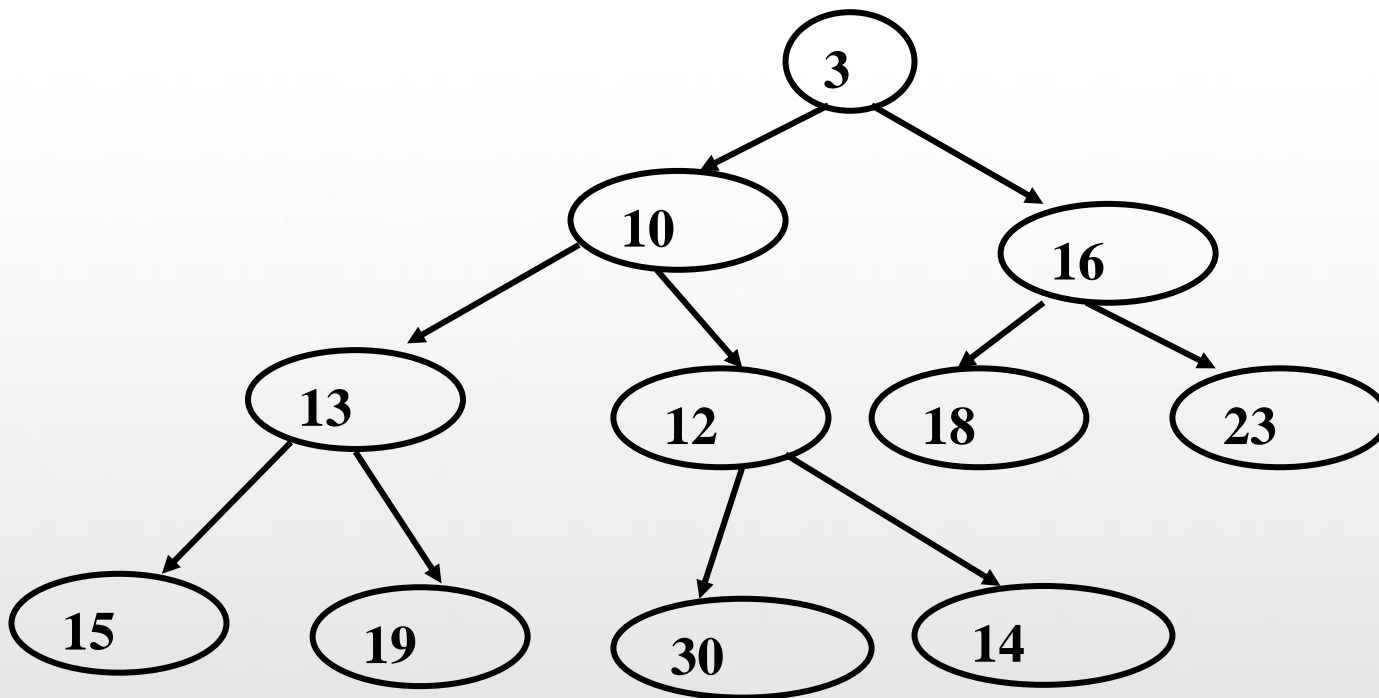
2	8	10	16	17	18	23	20	21	30
---	---	----	----	----	----	----	----	----	----

Reconstruct the binary heap (min_heap)

Binary Heap : H.W



Problem 2: Give the array representation for



Sorting Using heap tree

```
heap_sort(A)
  Build_heap;
  for(i=length(A); i > 1; i-- ) {
    exchange (A[i], A[1]);
    Decrement(heap_size);
    HEAPIFY (A, 1);
  }
```

HEAPIFY $\rightarrow O(\log n)$


for statement $\rightarrow O(n)$

heap_sort $\rightarrow O(n \log n)$

Question?



“Success is the sum of small efforts, repeated day in and day out.”
Robert Collier



Reference: <https://www.hackerearth.com/practice/notes/heaps-and-priority-queues/>

Mark Allen Weiss: Data Structures and Algorithm Analysis in Java

Lecture Notes, Lydia Sinapova, Simpson College