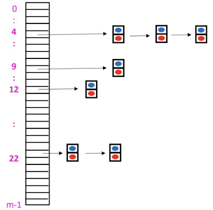
# Data Structures

Lecture 17 & 18: Heaps
- Binary heaps, ADT, operations, construction, Heap sort

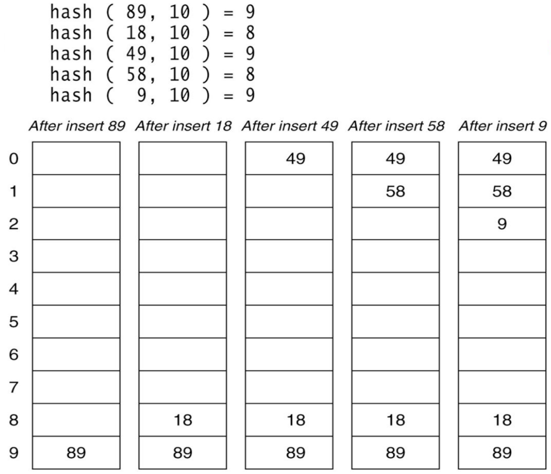
- Prof. Siddhartha Chandra
siddhartha_chandra@spit.ac.in

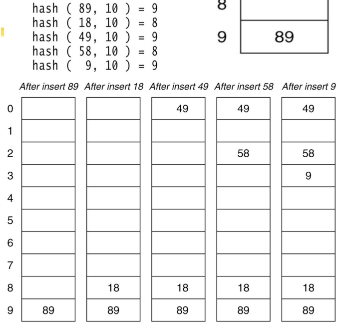
# Recap

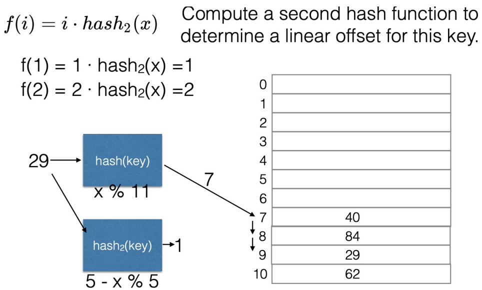



Collision Resolution Techniques:

- Separate chaining
- Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing

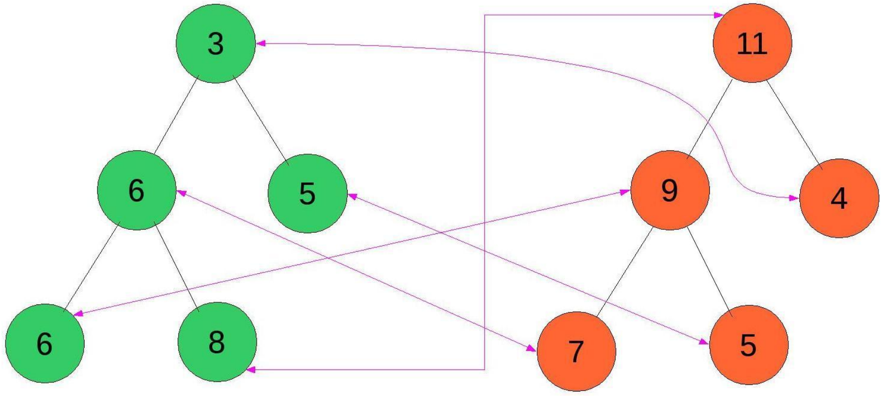


# Onto...

Heaps




Min Heap

Max Heap

# Recap | Priority Queues

- Min Priority Queue
- Max Priority Queue

# Recap | Operations in a Priority Queue

- **insert(DataType item, int pri)**
  Add an item to the queue which has priority pri

- **DataType peekMax()** or **DataType peekMin()**
  Peek at the item in the queue with the highest/lowest priority

- **DataType extractMax() or DataType extractMin()**
  Remove and return the item in the queue with the highest priority

# Heaps…

Is a binary tree satisfying 2 properties:

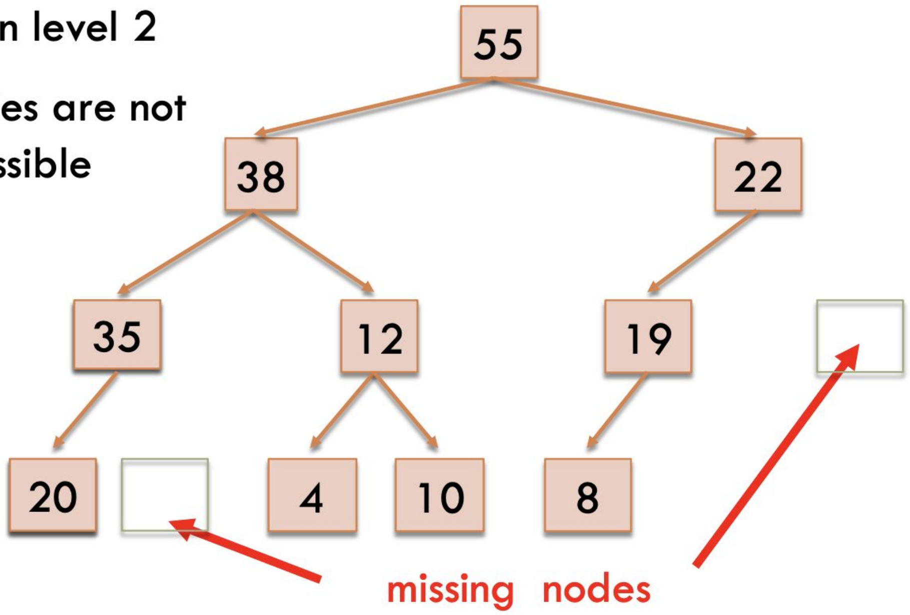
1) **Completeness.** Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.

Do not confuse with heap memory – different use of the word heap.

# Heaps : Identifying a heap

Not a heap because:

- missing a node on level 2
- bottom level nodes are not as far left as possible



# Heaps : Properties

Is a binary tree satisfying 2 properties:

1) **Completeness.** Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.

2) **Heap-order.**

“max on top”

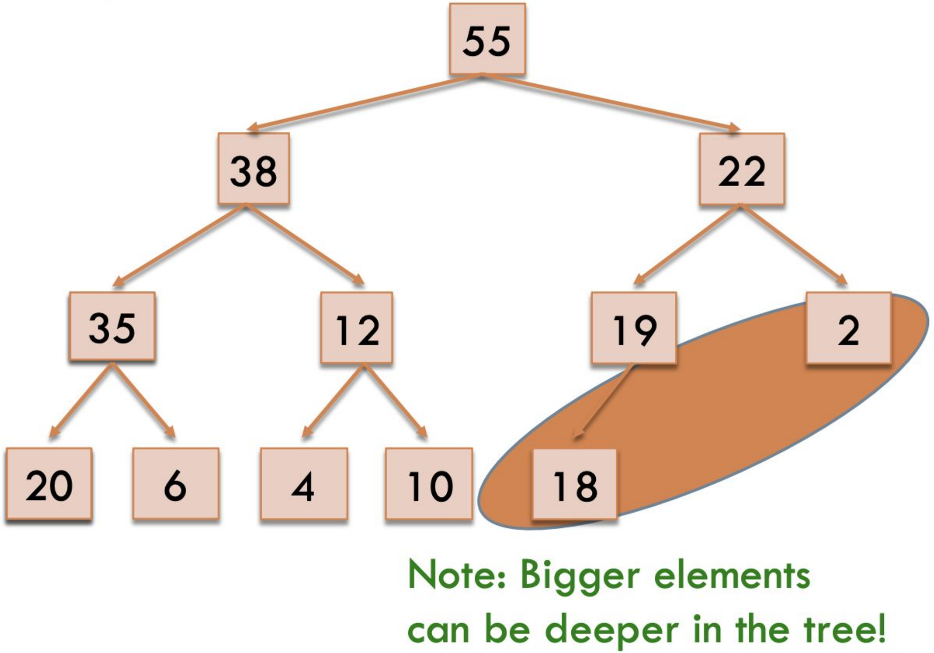
Max-Heap: every element in tree is <= its parent

Min-Heap: every element in tree is >= its parent

“min on top”

# Heaps: Properties

Every element is <= its parent



# Heaps: ADT implementation

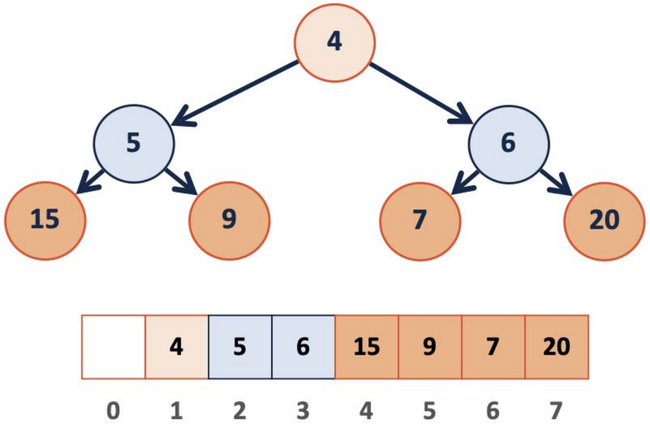
By storing as a complete tree, can avoid using pointers at all!

Can index from 0 or 1 (we will index from 1 in slides)

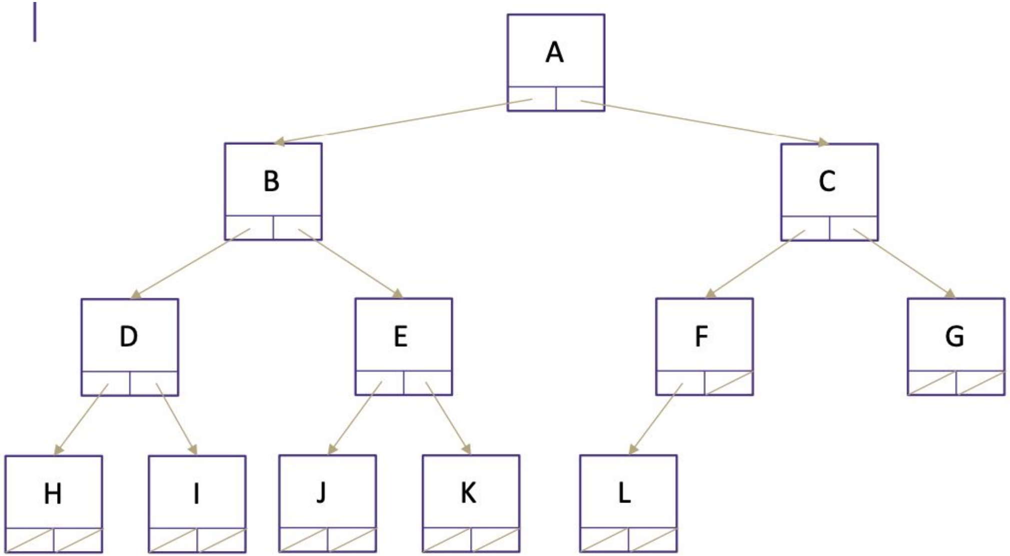
`leftChild(i): 2i`

`rightChild(i): 2i+1`

`parent(i): floor(i/2)`



# Heaps: Array implementation




Fill array in **level-order** from left to right

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

How do we find the minimum node?

$$peekMin() = arr[0]$$

How do we find the last node?

$$lastNode() = arr[size - 1]$$

How do we find the next open space?

$$openSpace() = arr[size]$$

How do we find a node's left child?

$$leftChild(i) = 2i + 1$$

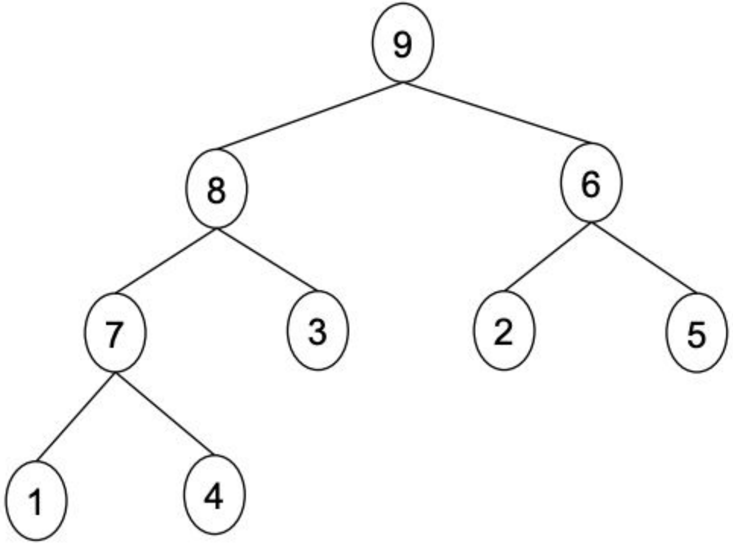
How do we find a node's right child?

$$rightChild(i) = 2i + 2$$

How do we find a node's parent?

$$parent(i) = \frac{(i-1)}{2}$$

# Max Heap




## Max Priority Queue ADT

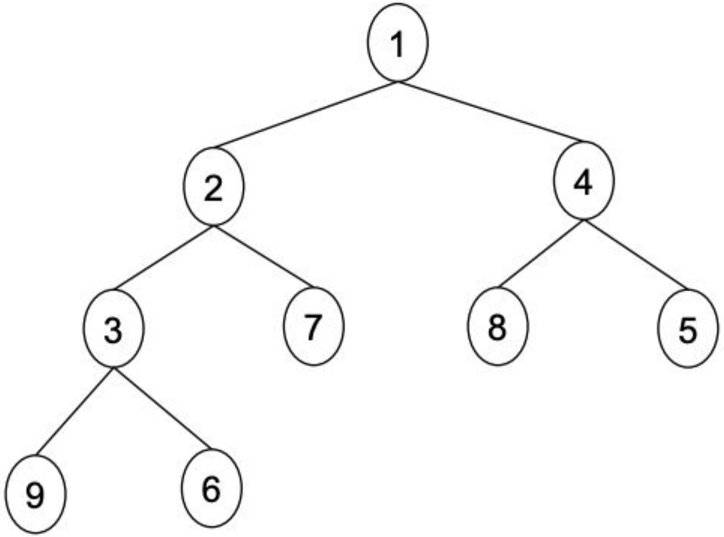
**State**

Set of comparable values ordered based on ‘priority’

**Behaviour**

- **extractMax() -** returns the element with the largest priority, removes it from the collection
- **peekMax() -** find, but do not remove the element with the largest priority
- **insert(value) -** add a new element to the collection

# Min heap



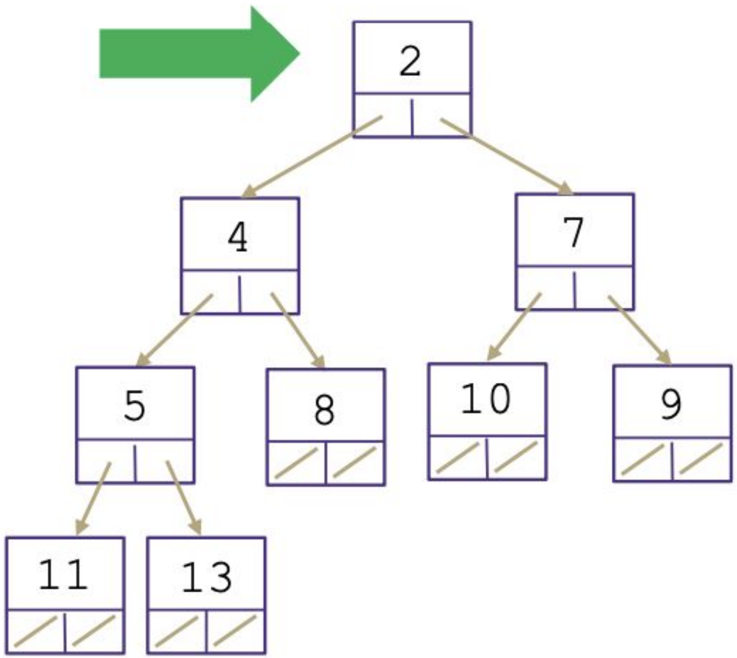

**Min Priority Queue ADT**

**State**
Set of comparable values ordered based on ‘priority’

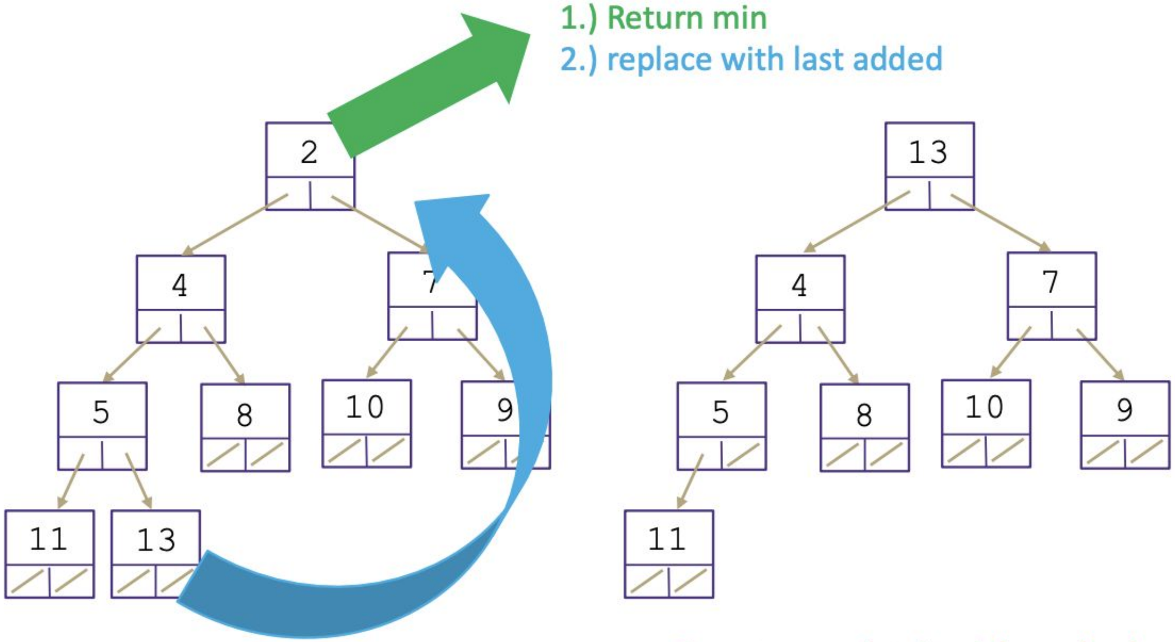
**Behaviour**

- **extractMin() -** returns the element with the smallest priority, removes it from the collection
- **peekMin() -** find, but do not remove the element with the smallest priority
- **insert(value) -** add a new element to the collection

# Heap Operation: PeekMin



# Heap Operation: ExtractMin

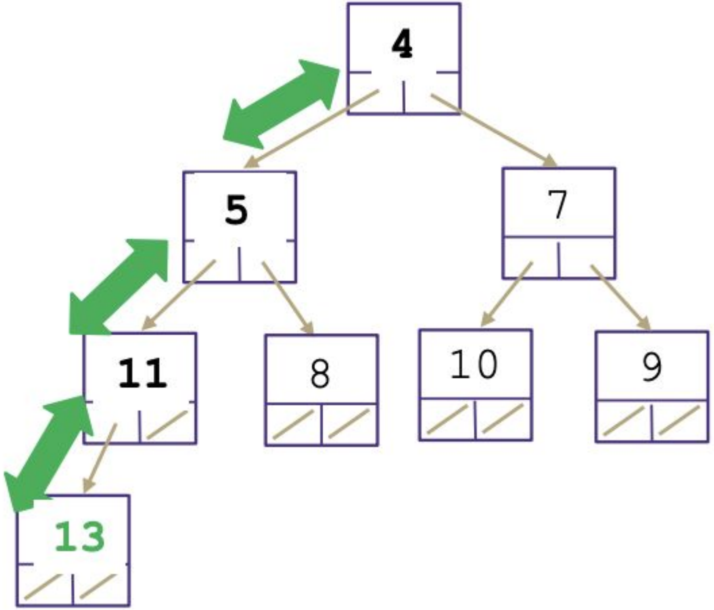



Structure maintained, heap broken

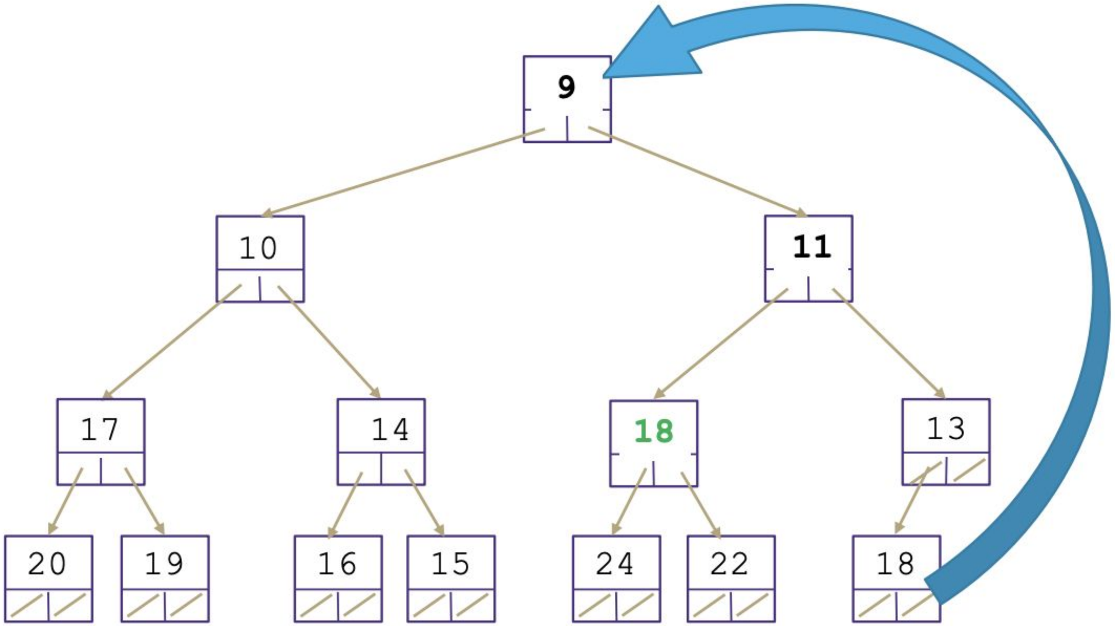
# Heap Operation: ExtractMin

3.) percolateDown()

Recursively swap parent with smallest child



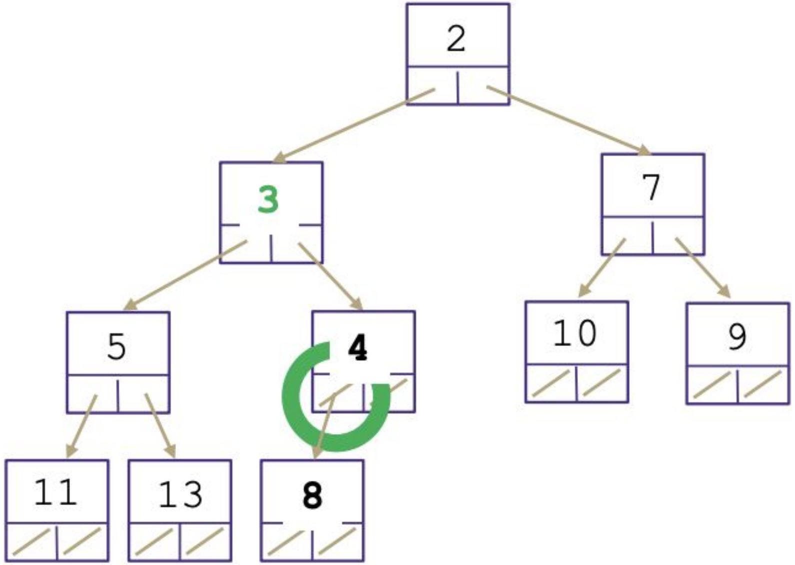
# Practice: ExtractMin



# Heap Operation: Insert

Algorithm:

- Insert a node to ensure no gaps
- Fix heap invariant
- percolate **UP**

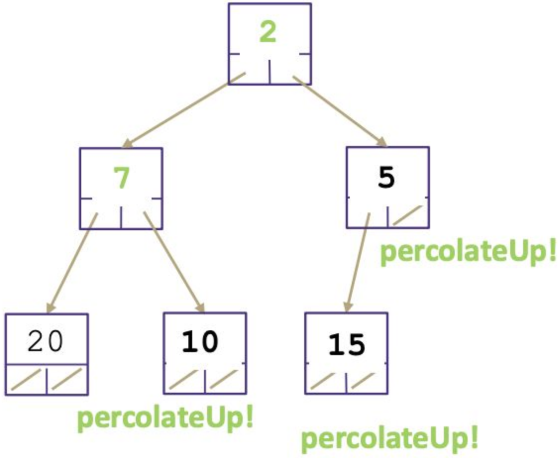


# Heap Operation: Practice

5, 10, 15, 20, 7, 2

**Min Binary Heap Invariants**

1. **Binary Tree** – each node has at most 2 children
2. **Min Heap** – each node's children are larger than itself
3. **Level Complete** - new nodes are added from left to right completely filling each level before creating a new one



# Heap construction: Process

1. Ensure that the heap retains the property of a max/min heap as the heap is built.
2. Build the heap and then transform the heap into a max/min heap ("heapify" the heap).

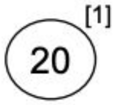
# Heap construction: Method 1

## Method 1: Add The First Element

- Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | | | | | |

- The corresponding tree



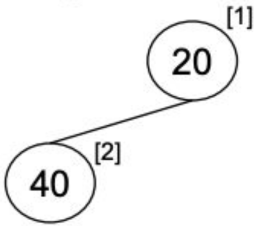
# Heap construction: Method 1

## Method 1: Add The Second Element

- Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | 40 | | | | |

- The corresponding tree



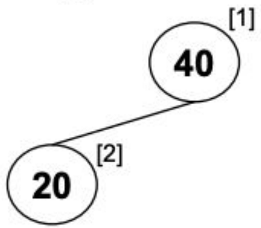
# Heap construction: Method 1

## Method 1: Swap The First And Second Elements

•Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | **40** | **20** | | | | |

•The corresponding tree



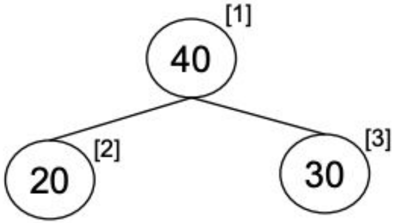
# Heap construction: Method 1

## Method 1: Add The Third Element

- Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 40 | 20 | 30 | | | |

- The corresponding tree



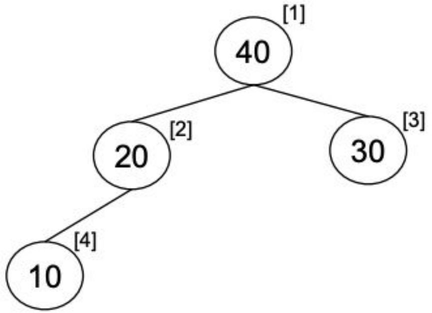
# Heap construction: Method 1

## Method 1: Add The Fourth Element

- Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 40 | 20 | 30 | 10 | | |

- The corresponding tree



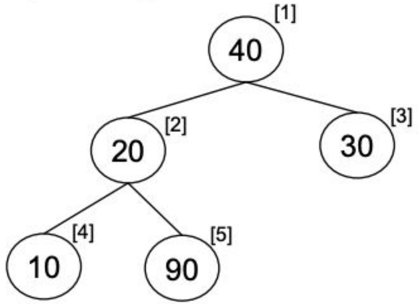
# Heap construction: Method 1

## Method 1: Add The Fifth Element

•Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 40 | 20 | 30 | 10 | 90 | |

•The corresponding tree



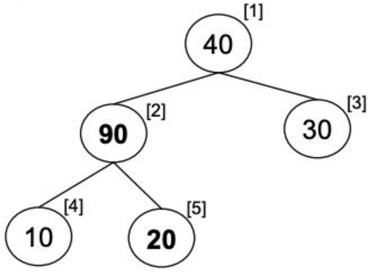
# Heap construction: Method 1

**Method 1: Swap the Second And Fifth Elements**

- Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 40 | **90** | 30 | 10 | **20** | |

- The corresponding tree



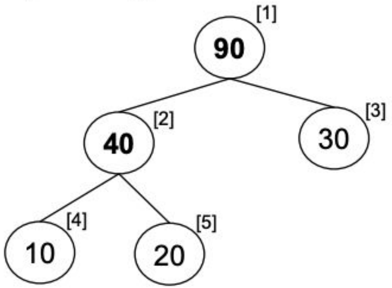
# Heap construction: Method 1

## <u>Method 1: Swap the First And Second Elements</u>

•Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | **90** | **40** | 30 | 10 | 20 | |

•The corresponding tree



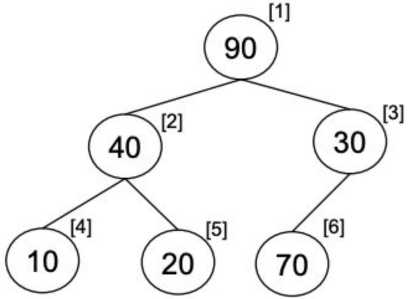
# Heap construction: Method 1

## Method 1: Add The Sixth Element

•Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 90 | 40 | 30 | 10 | 20 | 70 |

•The corresponding tree



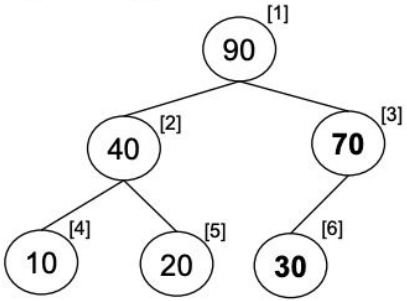
# Heap construction: Method 1

## Method 1: Swap The Third And Sixth Elements

•Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 90 | 40 | **70** | 10 | 20 | **30** |

•The corresponding tree



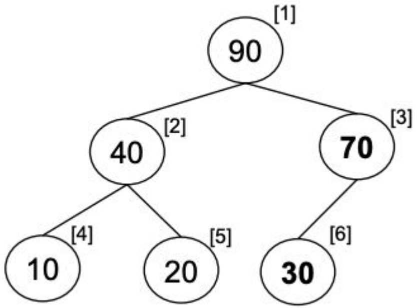
# Heap construction: Method 1

## Method 1: The Final State Of The Heap

•Array representation

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 90 | 40 | 70 | 10 | 20 | 30 |

•The corresponding tree



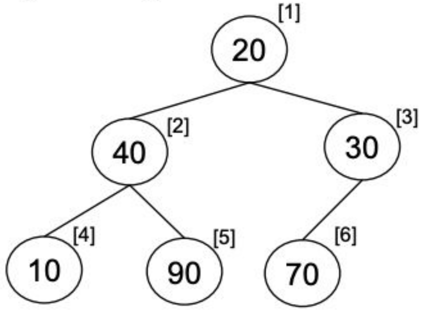
# Heap construction: Method 2

## Method 2 For Building A Heap

- The information is read into an array

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | 40 | 30 | 10 | 90 | 70 |

- The corresponding tree




Also called
**Floyd's method**

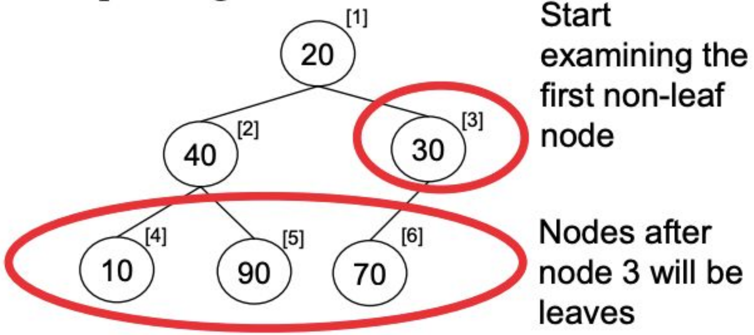
# Heap construction: Method 2

## <u>Method 2 For Building A Heap: Where To Start</u>

- Start with node [ $\lfloor noNodes/2 \rfloor$ ], examine if the heap is a maxheap

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | 40 | 30 | 10 | 90 | 70 |

- The corresponding tree



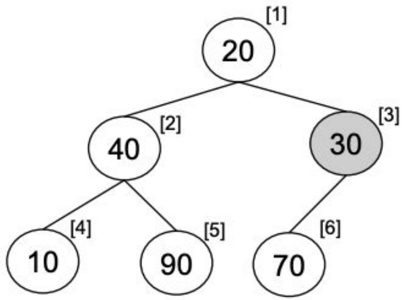
# Heap construction: Method 2

**Method 2 For Building A Heap: Examine Element [3]**

•The information is read into an array

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | 40 | 30 | 10 | 90 | 70 |

•The corresponding tree



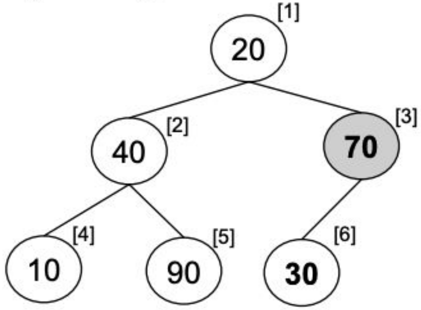
# Heap construction: Method 2

## Method 2 For Building A Heap: Reheap Related To Element [3]

•The information is read into an array

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | 40 | **70** | 10 | 90 | **30** |

•The corresponding tree



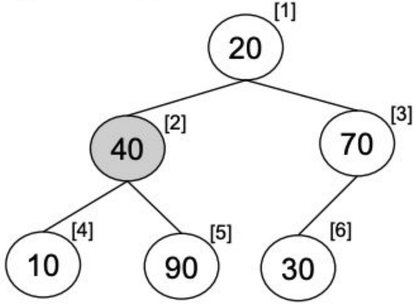
# Heap construction: Method 2

## Method 2 For Building A Heap: Examine Element [2]

- The information is read into an array

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | 40 | 70 | 10 | 90 | 30 |

- The corresponding tree



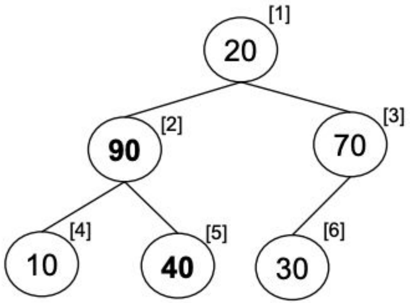
# Heap construction: Method 2

## Method 2 For Building A Heap: Reheap Related To Element [2]

- The information is read into an array

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | **90** | 70 | 10 | **40** | 30 |

- The corresponding tree



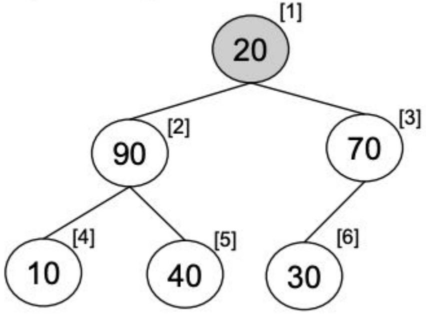
# Heap construction: Method 2

## Method 2 For Building A Heap: Examine Element [1]

- The information is read into an array

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 20 | 90 | 70 | 10 | 40 | 30 |

- The corresponding tree



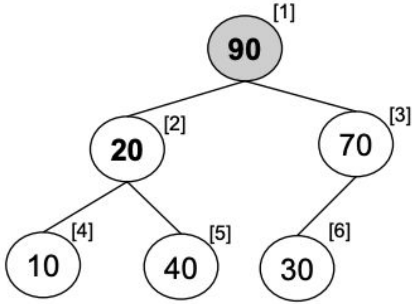
# Heap construction: Method 2

## Method 2 For Building A Heap: First Reheap Related To Element [1]

- The information is read into an array

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | **90** | **20** | 70 | 10 | 40 | 30 |

- The corresponding tree



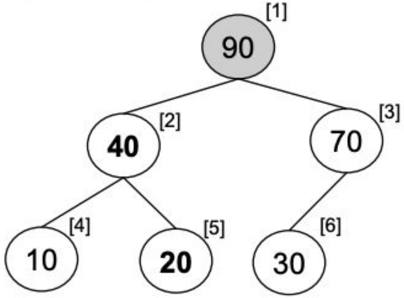
# Heap construction: Method 2

## Method 2 For Building A Heap: Second Reheap Related To Element [1]

•The information is read into an array

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| | 90 | 40 | 70 | 10 | 20 | 30 |

•The corresponding tree



# Heap construction: Method 2

**Min-heap example**

(UW slides: 18 - 23)

# Heap Sort

[MIT | 6-006 | Slide (21-27)]

# Recap

Heaps

- Properties
- Array implementation
- ADT
  - Max-heap
  - Min-heap
- Operations:
  - PeekMin()
  - ExtractMin()
  - Insert(val)
- Two methods for heap construction
- HeapSort