# Priority Queue

## A brief Overview

Md Awsaf Alam[1]
Ahmed Nafis Fuad[1]

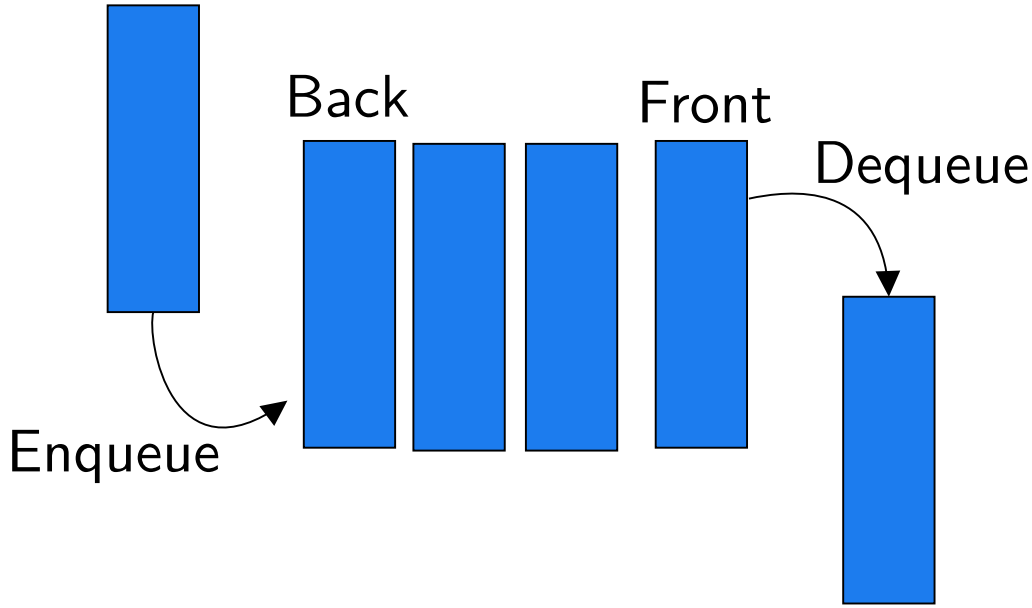
[1]Department of Computer Science and Engineering
BUET

July 23, 2018

# Table of Contents

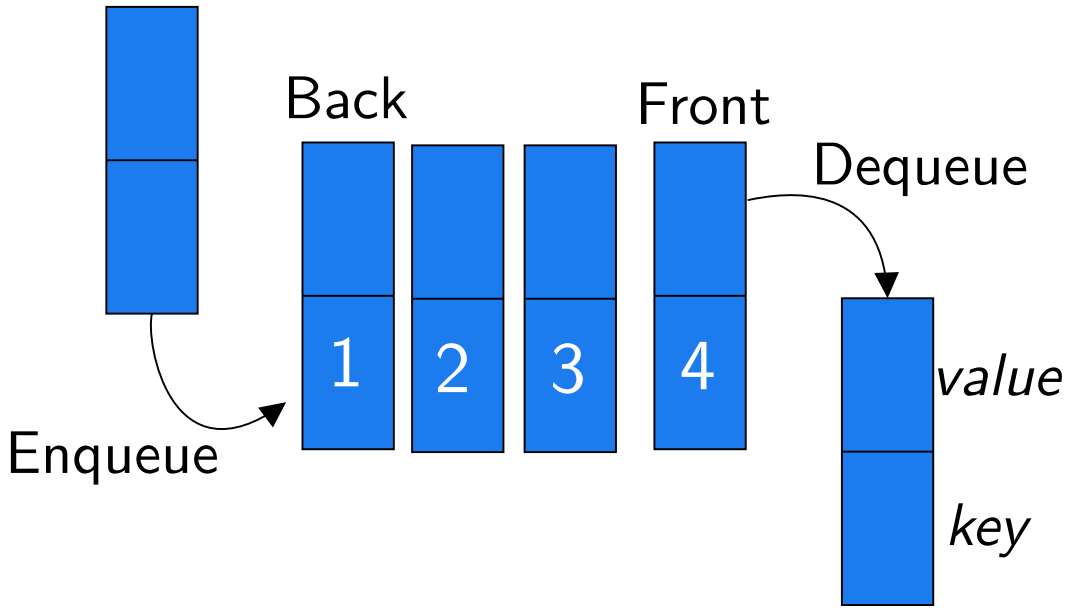
# What is a Queue?

A queue is an example of a linear data structure, which works on the basis of "**first-in-first-out**" (FIFO).



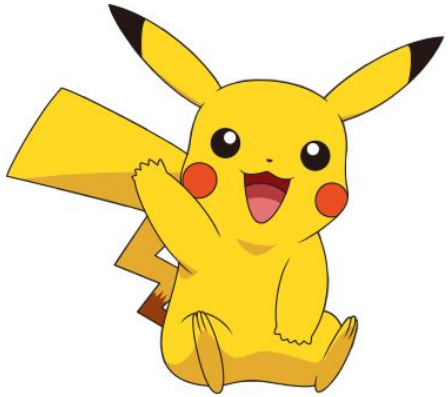
# What is a Priority Queue?

A priority queue is like a regular queue but where additionally each element has a "**priority**" associated with it. In a priority queue, an element with high priority is served before an element with low priority.



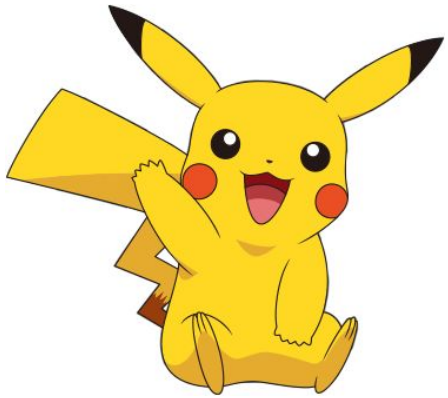
# Application of Queue

# Application of Queue

1

# Application of Queue

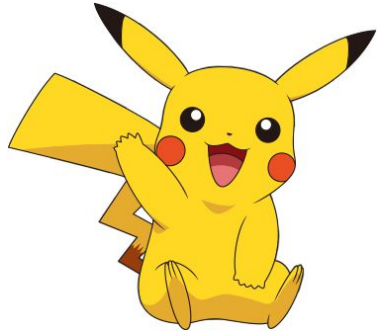



1

1

# Application of Queue

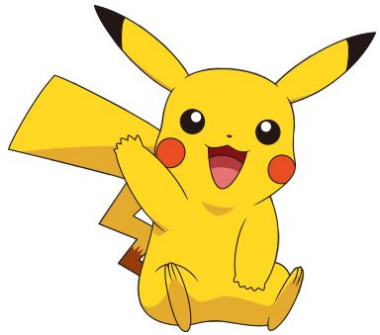

2



1

# Application of Queue


2



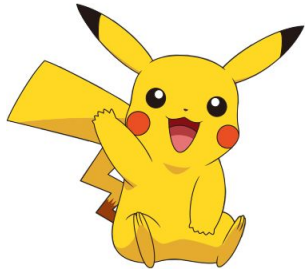
1


1

# Application of Queue


3

2

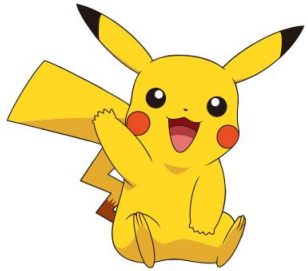


1

# Application of Queue


3

2



1

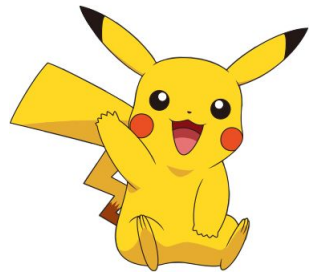
1

# Application of Queue


3

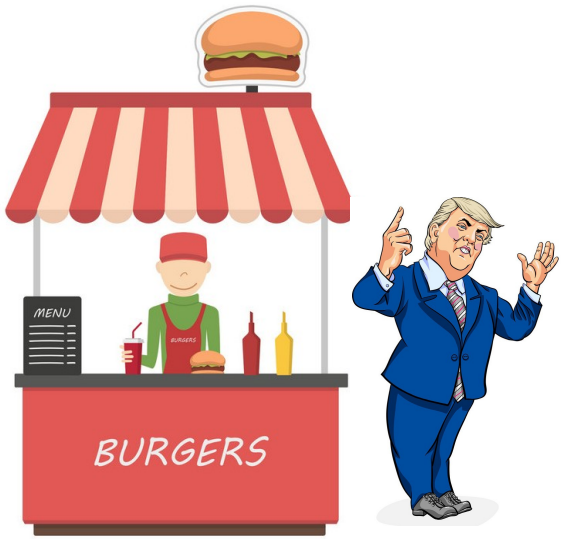
3

2



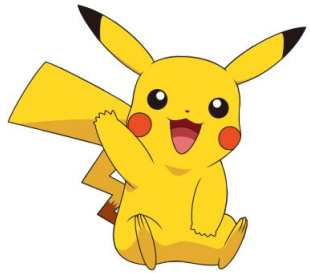
1

# Application of Queue




3

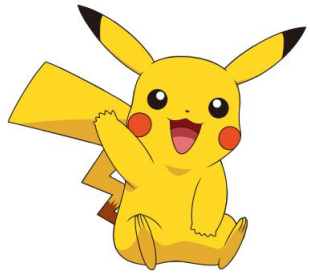
2



1

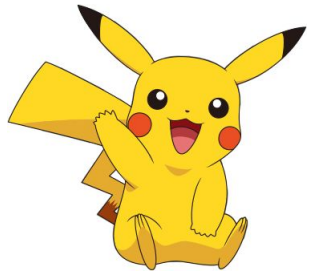
# Application of Queue


2



1

# Application of Queue

1

# Applications

- Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.

# Applications

- Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.
- Prim's algorithm: It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.

# Applications

- Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.
- Prim's algorithm: It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.
- Data compression : It is used in Huffman codes which is used to compresses data.

# Applications

- Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.
- Prim's algorithm: It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.
- Data compression : It is used in Huffman codes which is used to compresses data.
- CPU scheduling : Each queue may have its own scheduling algorithm, implemented using priority queue.

# Implementations of Priority Queue

Priority Queues can be implemented using :

- Fibonacci Heap
- Binary Heap

# Implementations of Priority Queue

Priority Queues can be implemented using :

- Fibonacci Heap
- Binary Heap

# Implementations of Priority Queue

Priority Queues can be implemented using :

- Fibonacci Heap
- Binary Heap

# Implementations of Priority Queue

Priority Queues can be implemented using :

- Fibonacci Heap
- Binary Heap

# Definition

A Binary (Max) Heap is a complete binary tree that maintains the Max Heap property. Binary Heap is one possible data structure to model an efficient Priority Queue.

# Definition

A Binary (Max) Heap is a complete binary tree that maintains the Max Heap property. Binary Heap is one possible data structure to model an efficient Priority Queue.

- The largest element in a max-heap is stored at the root
- Each node of the tree corresponds to an element of the array that stores the value in the node.

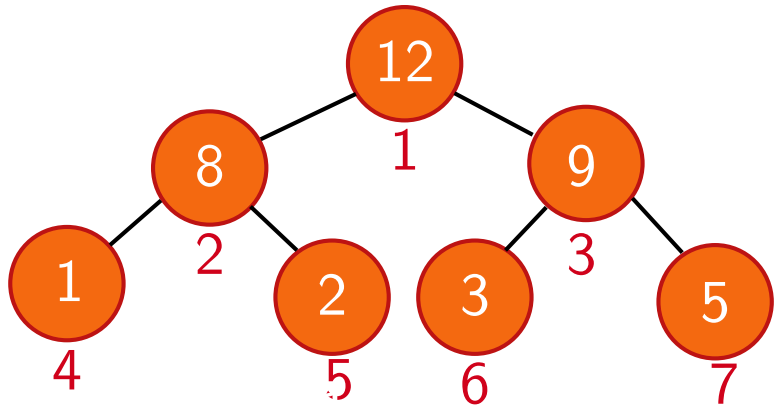
# Definition

A Binary (Max) Heap is a complete binary tree that maintains the Max Heap property. Binary Heap is one possible data structure to model an efficient Priority Queue.

- The largest element in a max-heap is stored at the root
- Each node of the tree corresponds to an element of the array that stores the value in the node.
- The tree is completely filled on all levels except possibly the lowest, where it is filled from the left up to a point.

# Representing Binary Heap as an array

1. The root node is A[1]
2. Node i is A[i]

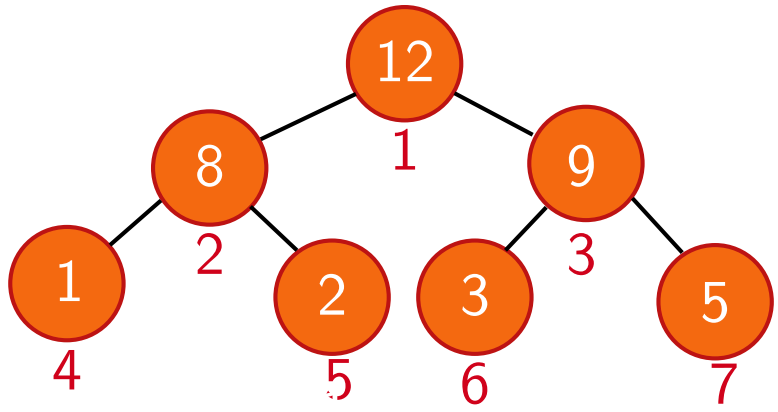



A =

| 12 | 8 | 9 | 1 | 2 | 3 | 5 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Representing Binary Heap as an array

1. The root node is A[1]
2. Node i is A[i]
3. The parent of node i is A[i/2]

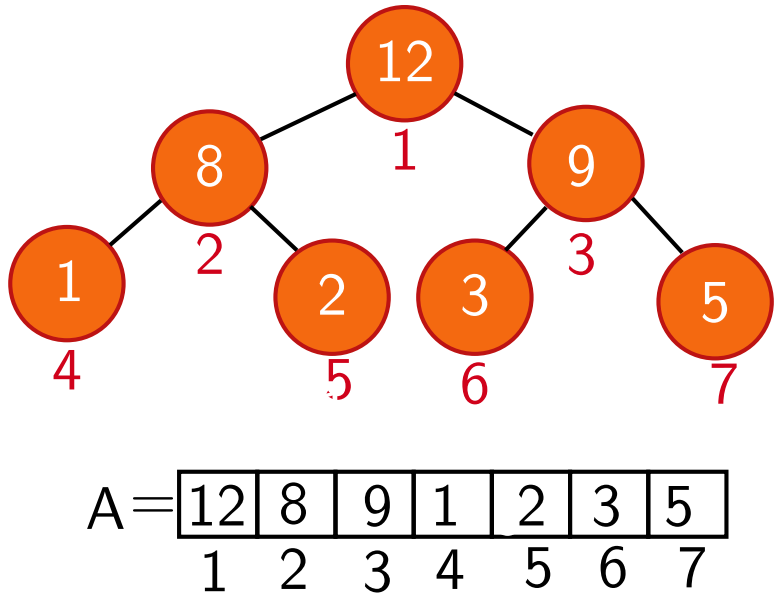



A =

| 12 | 8 | 9 | 1 | 2 | 3 | 5 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

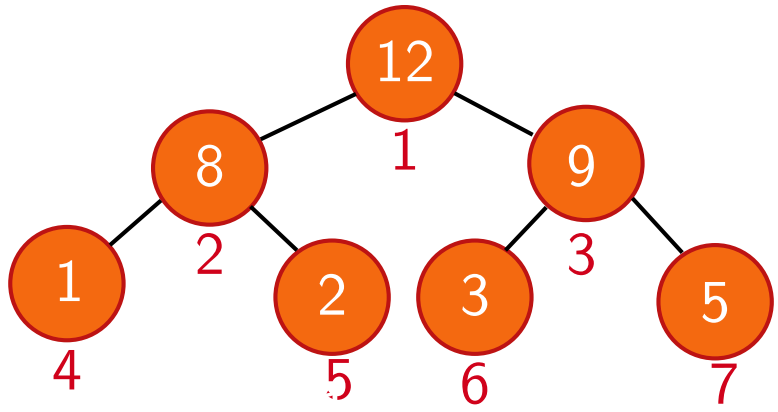
# Representing Binary Heap as an array

1. The root node is A[1]
2. Node i is A[i]
3. The parent of node i is A[i/2]
4. The left child of node i is A[2i]



# Representing Binary Heap as an array

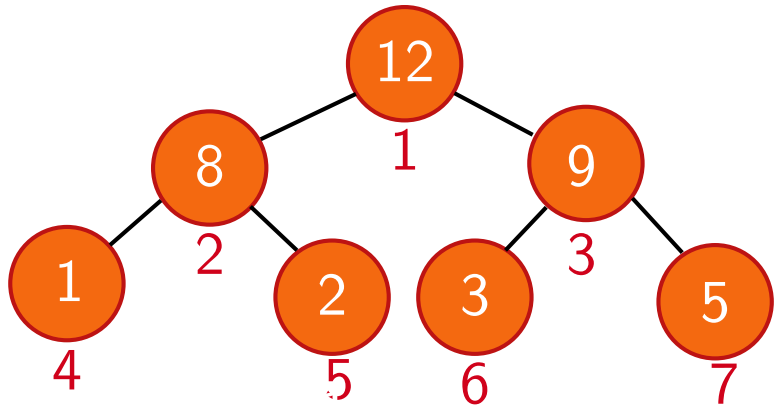
1. The root node is A[1]
2. Node i is A[i]
3. The parent of node i is A[i/2]
4. The left child of node i is A[2i]
5. The right child of node i is A[2i + 1]




| A = | 12 | 8 | 9 | 1 | 2 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Representing Binary Heap as an array

1. The root node is A[1]
2. Node i is A[i]
3. The parent of node i is A[i/2]
4. The left child of node i is A[2i]
5. The right child of node i is A[2i + 1]

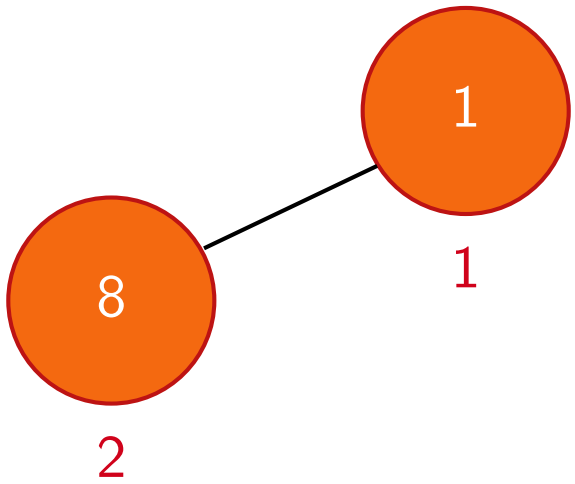



| A = | 12 | 8 | 9 | 1 | 2 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

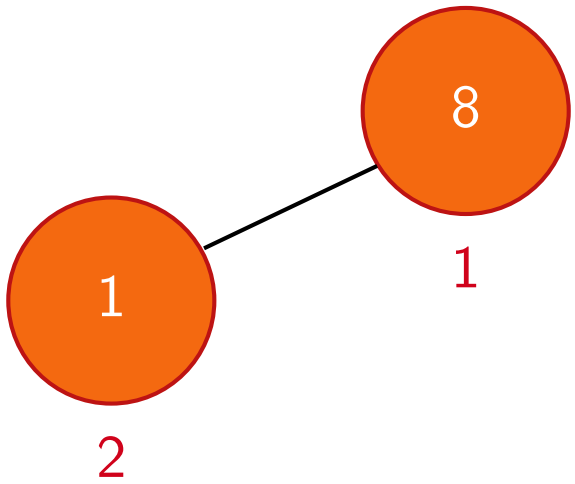
# Inserting 1


1

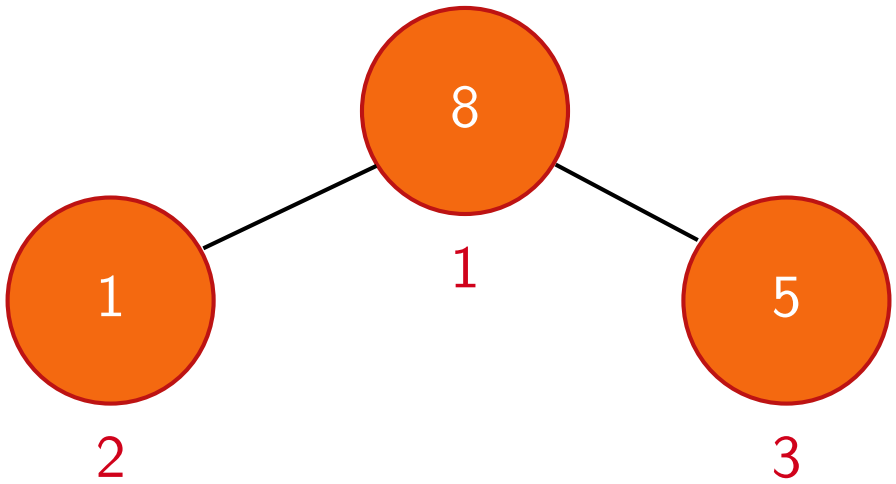
# Inserting 8



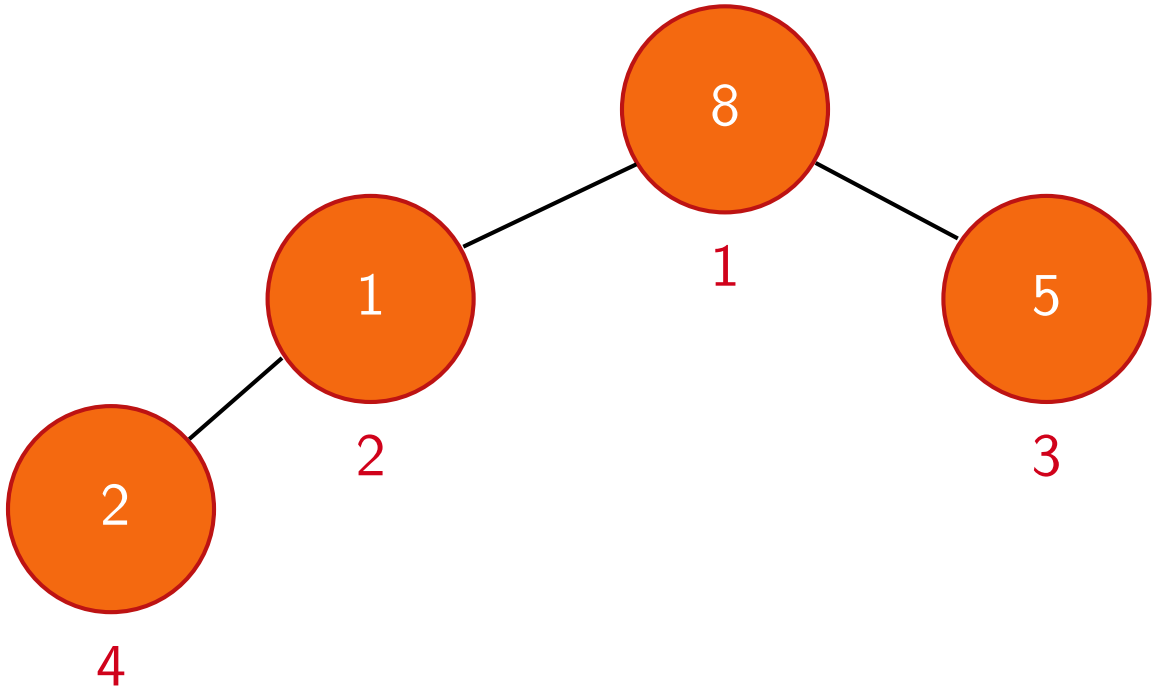
# Heapify()



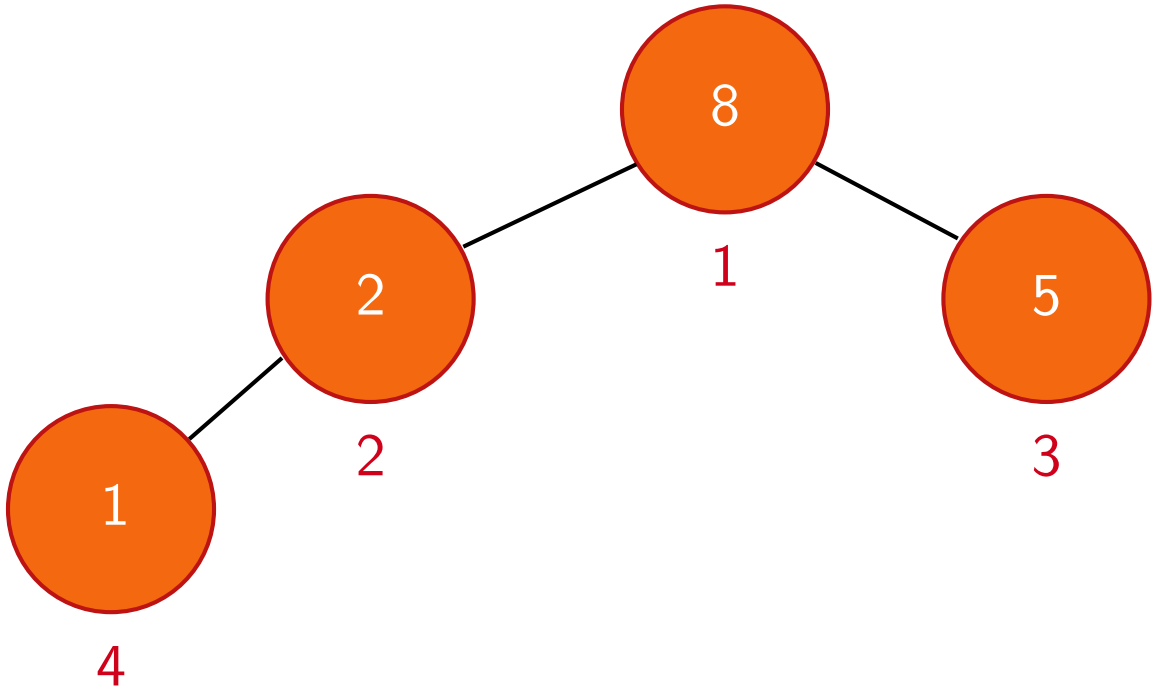
# Inserting 5



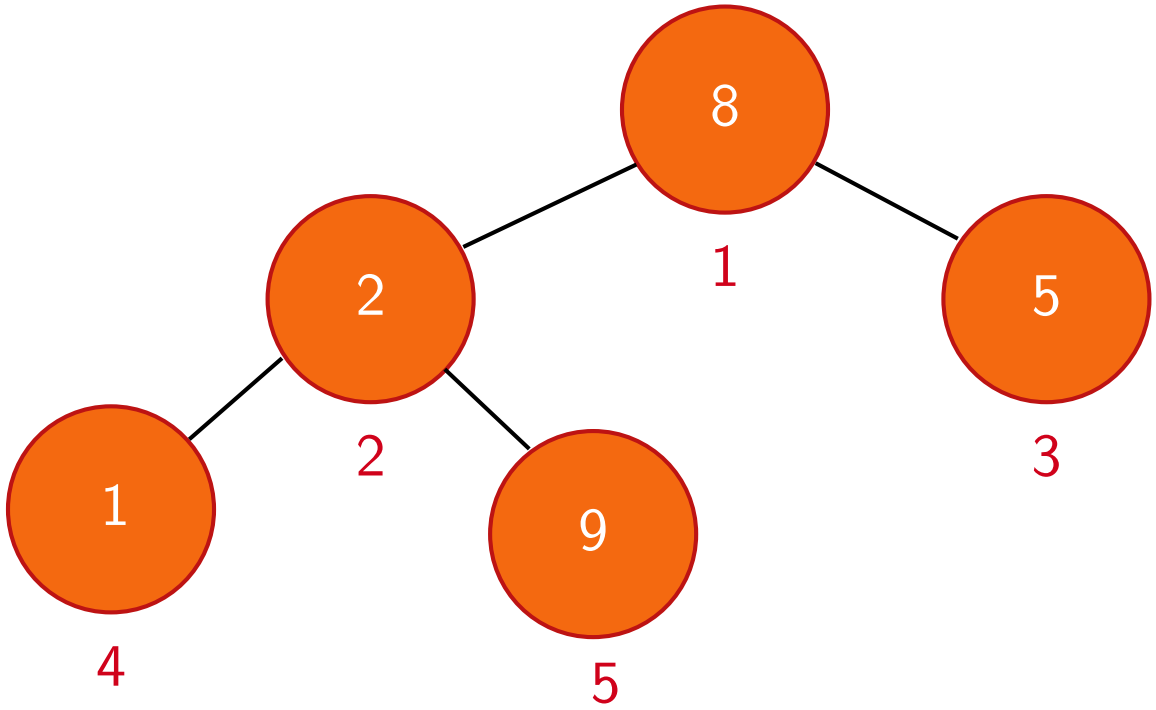
# Inserting 2



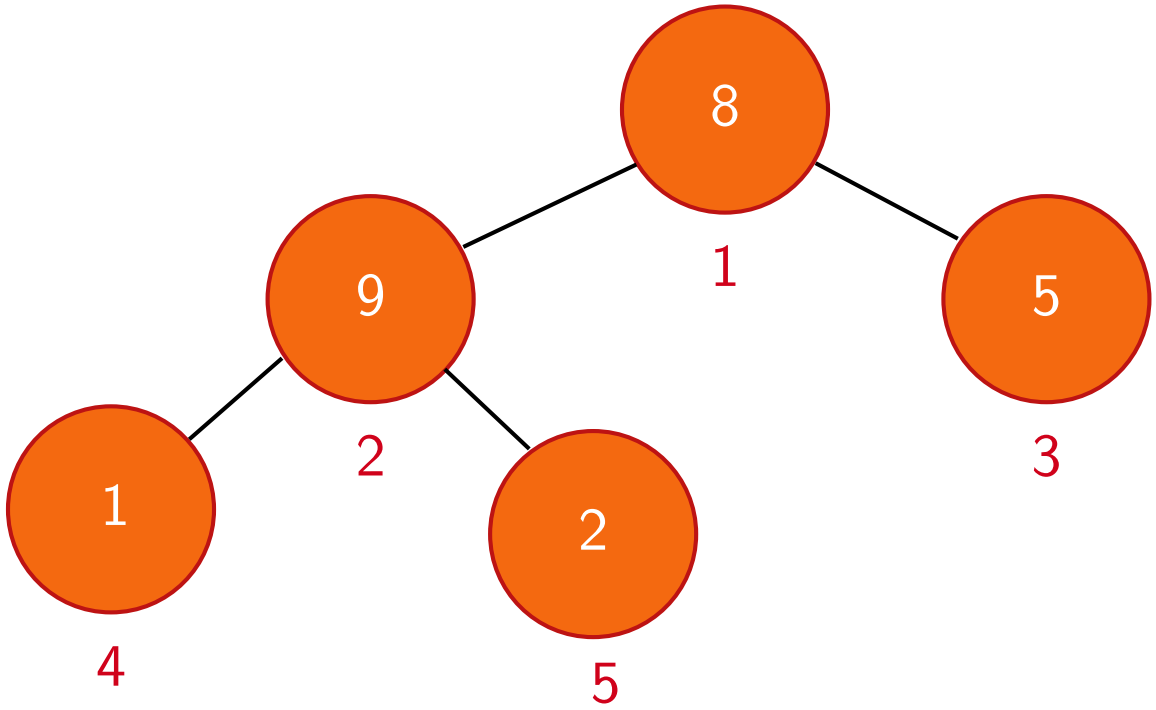
# Heapify()



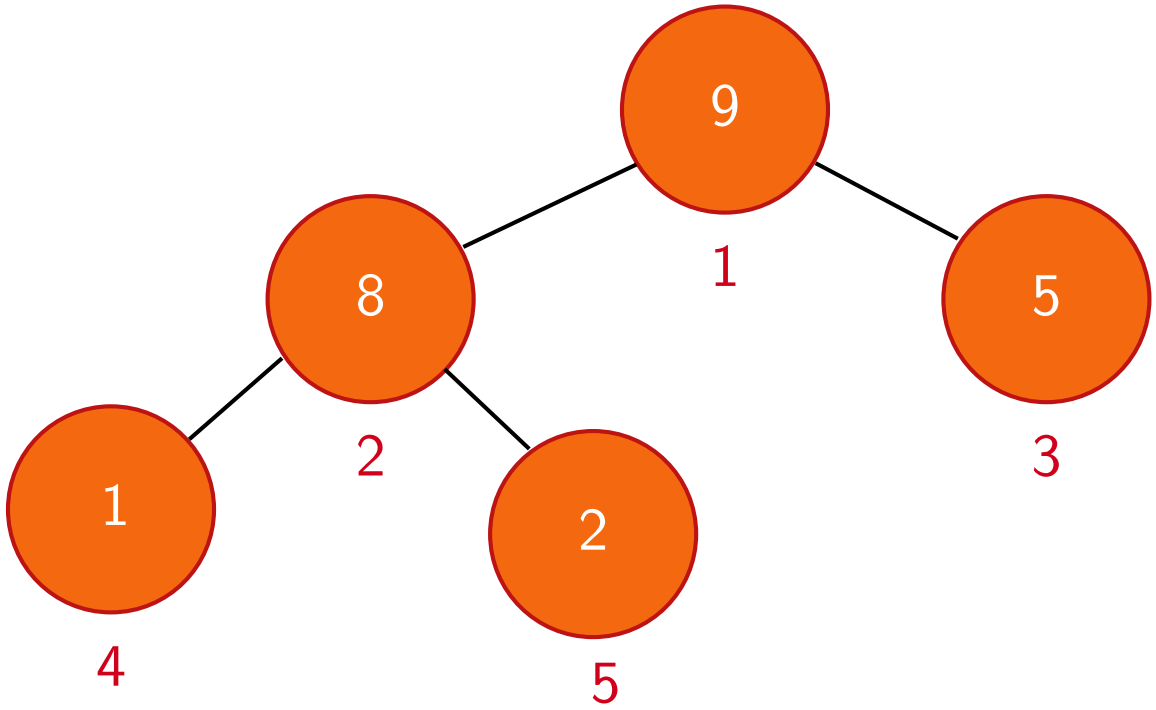
# Inserting 9



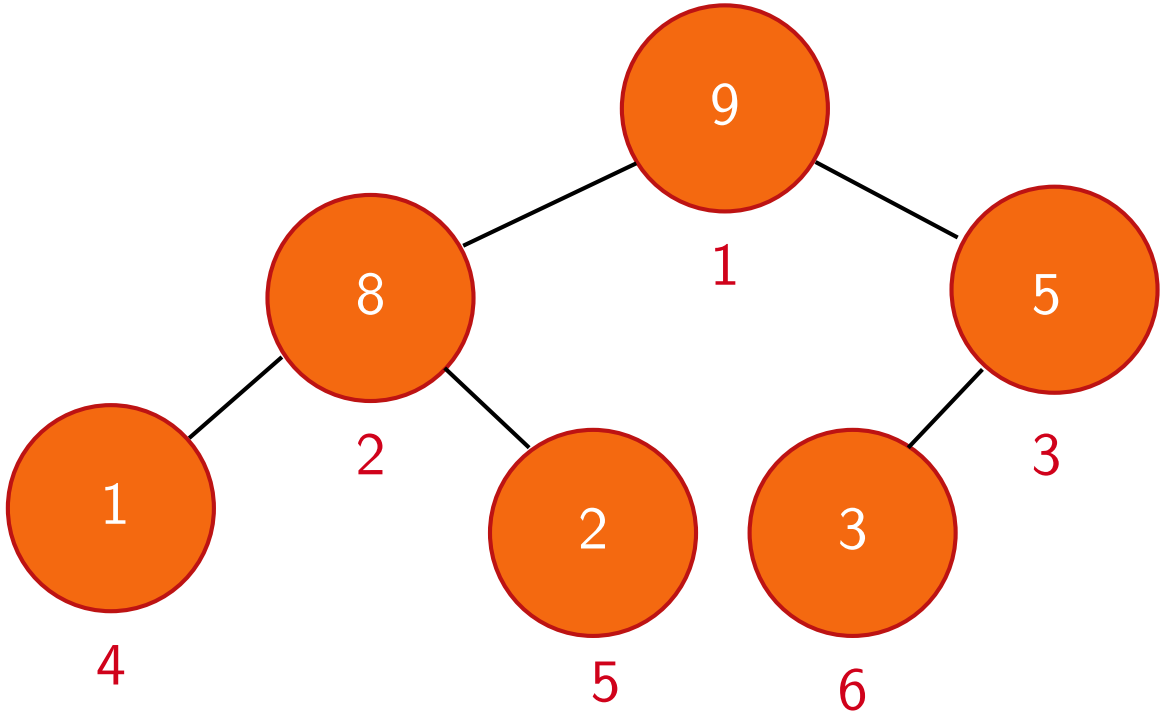
# Heapify()



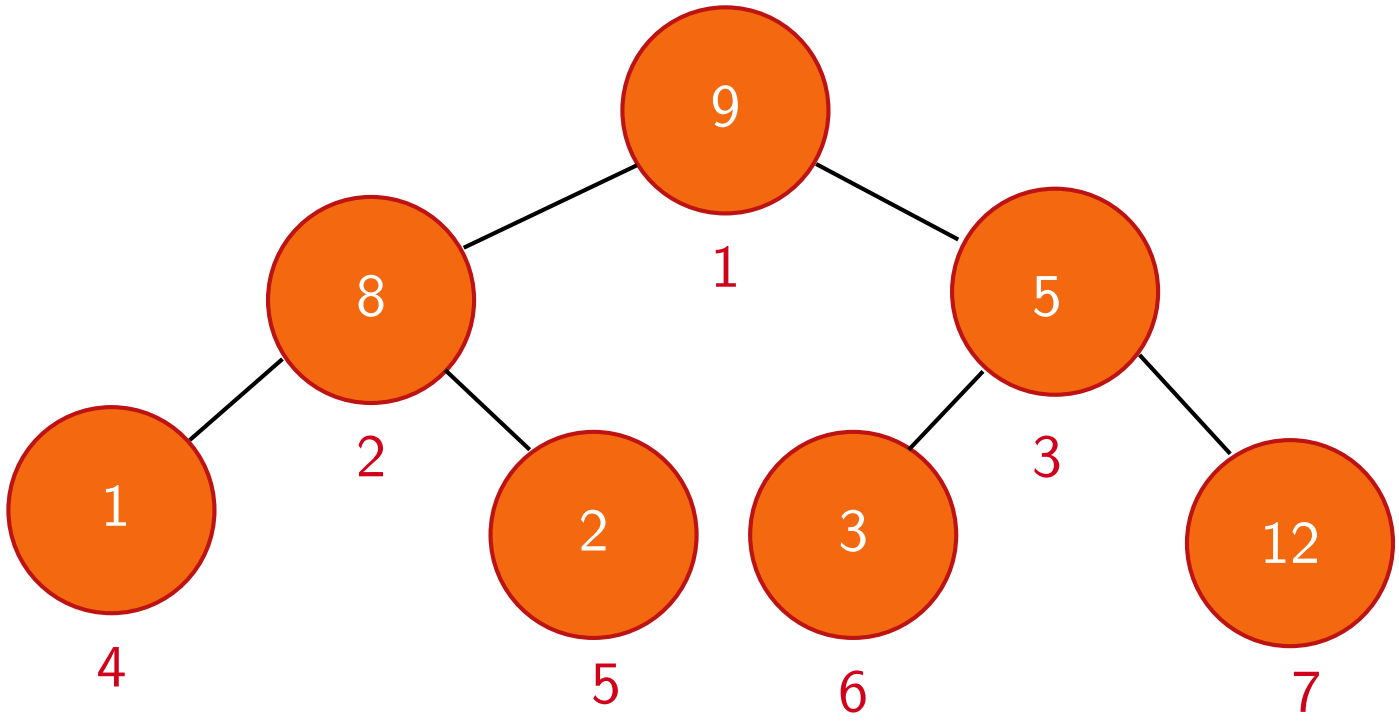
# Heapify()



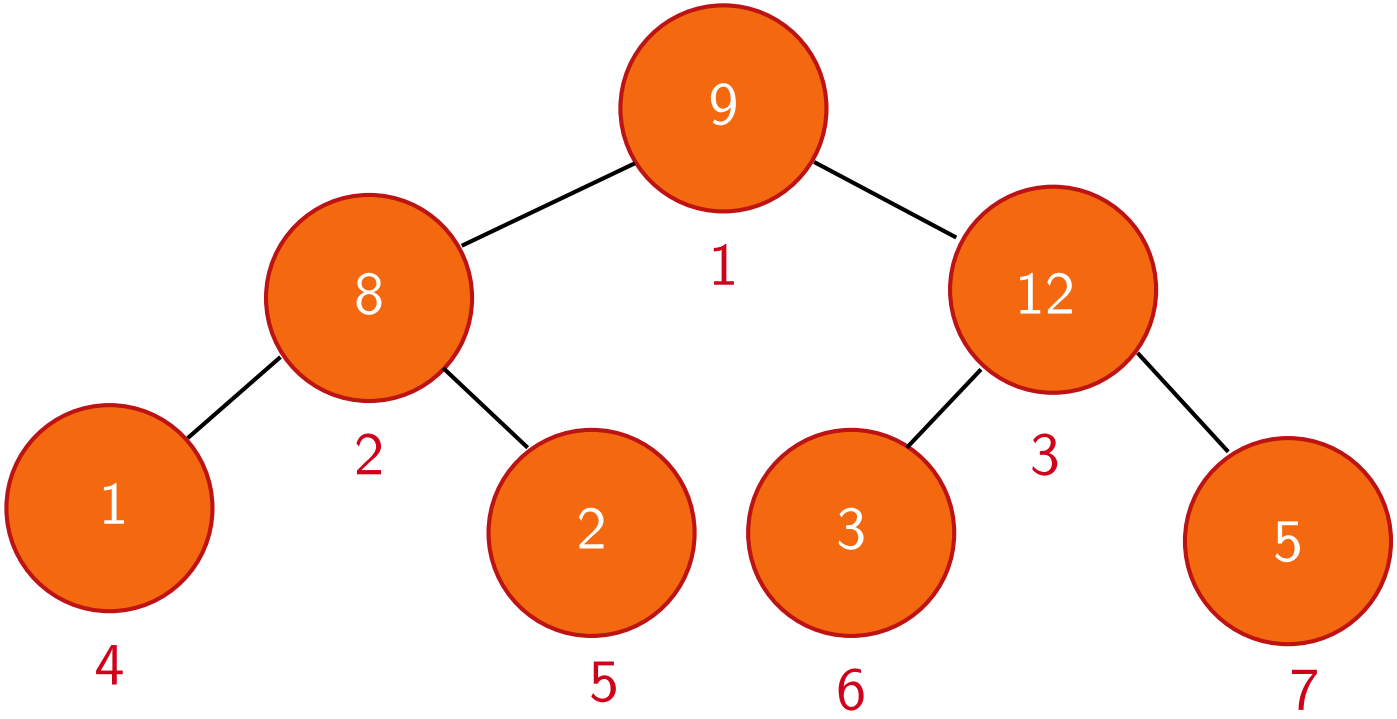
# Inserting 3



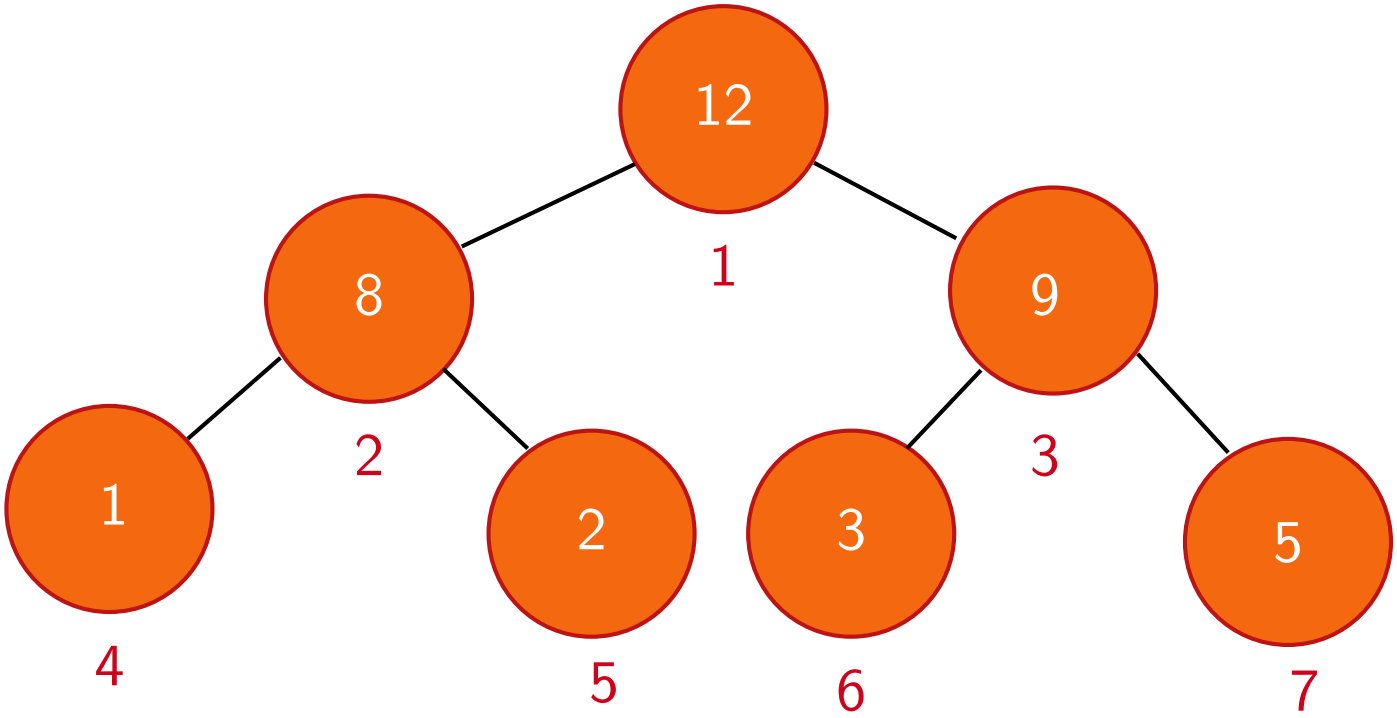
# Inserting 12



# Heapify()



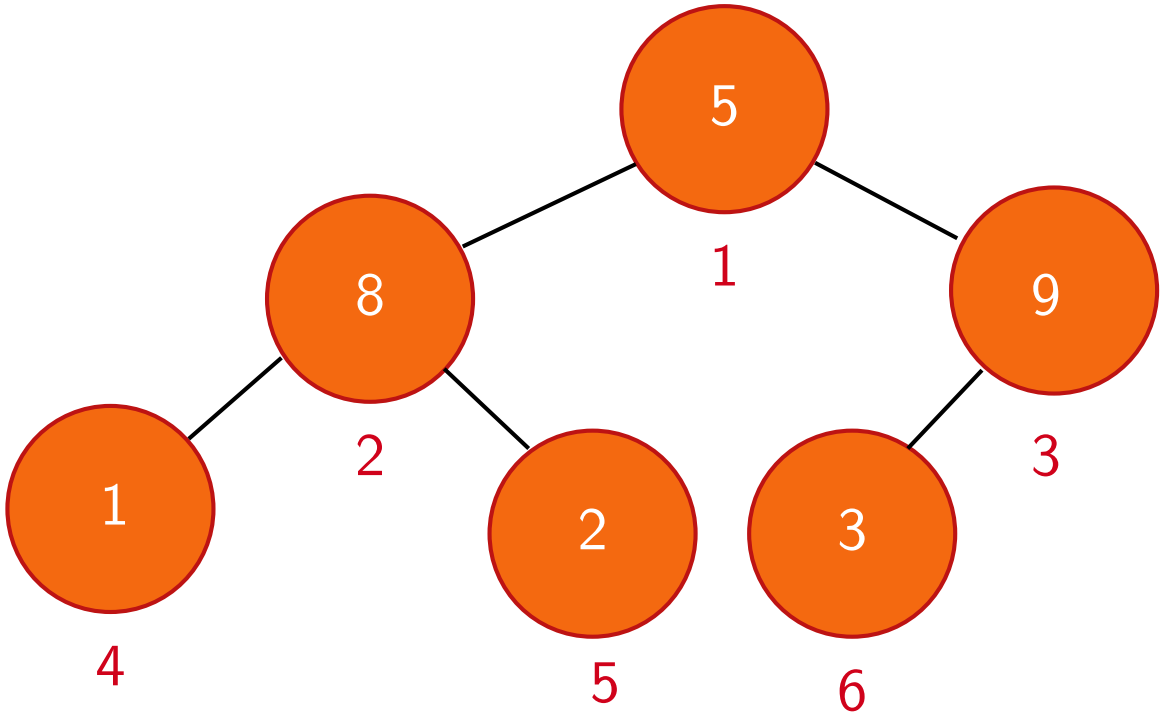
# Heapify()




A =

| 12 | 8 | 9 | 1 | 2 | 3 | 5 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

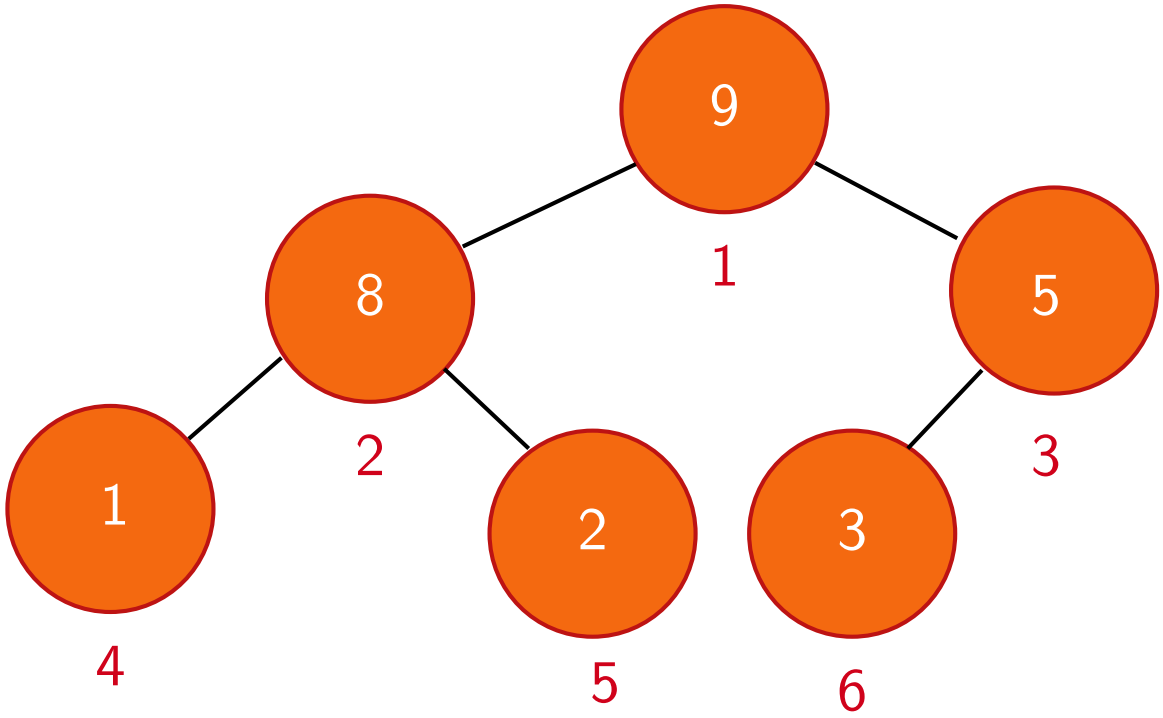
# Extract Max




A = | 5 | 8 | 9 | 1 | 2 | 3 |

1 2 3 4 5 6

# Heapify()




$$A = \begin{array}{|c|c|c|c|c|c|} \hline 9 & 8 & 5 & 1 & 2 & 3 \\ \hline \end{array}$$

1 2 3 4 5 6

# Max-Heapify Algorithm

MAX-HEAPIFY(A,$i$)

$l \leftarrow$ LEFT($i$)
$r \leftarrow$ RIGHT($i$)
**if** $l \leq heap\_size[A]$ *and* $A[l] > A[i]$ **then**
  $largest \leftarrow l$
**else**
  $largest \leftarrow i$
**end if**
**if** $r \leq heap\_size[A]$ *and* $A[r] > A[largest]$ **then**
  $largest \leftarrow r$
**end if**
**if** $largest \neq i$ **then**
  exchange $A[i] \leftrightarrow A[largest]$
**end if**
MAX-HEAPIFY(A,$largest$)

# Analysing Running Time

Extract Max : $O(1)$
Heapify: $T(n) \leq T(2n/3)O(1)$
Solving the recurrence, we have $T(n) = O(lgn)$
Thus, Heapify() takes $O(h)$ time for a node at height h.

# The End

Any Questions?