Max-Heaps

Abstract Data Type:

- Describes an object and its operations

Abstract Data Type:

- Describes an object and its operations

Data Structure:

- Some specific implementation of an Abstract Data Type

Abstract Data Type:

- Describes an object and its operations

Data Structure:

- Some specific implementation of an Abstract Data Type

Example of an Abstract Data Type: Priority Queues

Priority Queues

- Object : Set S of elements with “keys” (“priority”) that can be compared

Priority Queues

- **Object** : Set S of elements with “keys” (“priority”) that can be compared

- Operations:

Priority Queues

- Object : Set S of elements with “keys” (“priority”) that can be compared

- Operations:
  - **Insert(**S, x**):** Insert element x in S

Priority Queues

- **Object** : Set S of elements with "keys" ("priority") that can be compared

- **Operations**:
    - **Insert(**S, x**):** Insert element x in S
    - **Max(**S**):** Returns an element of highest priority in S

Priority Queues

- **Object** : Set S of elements with “keys” (“priority”) that can be compared

- **Operations**:
  - **Insert(**S, x**):** Insert element x in S
  - **Max(**S**):** Returns an element of highest priority in S
  - **Extract_Max(**S**):** returns Max(S) <u>and</u> removes it from S.

An Application of Priority Queue

The OS can use a Priority Queue to maintain a set S of jobs and schedule them according to their priorities:

- When a new job x arrives, the OS does **Insert**(S,x).
- When a processor becomes available to execute a job, the OS does

  Extract_Max(S**)**

Simple Data Structures for implementing a Priority Queue

Worst Case Time For	Insert	Extract_Max
Unordered Linked List		

Simple Data Structures for implementing a Priority Queue

Worst Case Time For	Insert	Extract_Max
Unordered Linked List	$\Theta(1)$	

Simple Data Structures for implementing a Priority Queue

Worst Case Time For	Insert	Extract_Max
Unordered Linked List	**$\Theta(1)$**	**$\Theta(n)$**

Simple Data Structures for implementing a Priority Queue

Worst Case Time For	Insert	Extract_Max
Unordered Linked List	$\Theta(1)$	$\Theta(n)$
Ordered Linked List		

Simple Data Structures for implementing a Priority Queue

Worst Case Time For	Insert	Extract_Max
Unordered Linked List	$\Theta(1)$	$\Theta(n)$
Ordered Linked List		$\Theta(1)$

Simple Data Structures for implementing a Priority Queue

Worst Case Time For	Insert	Extract_Max
Unordered Linked List	$\Theta(1)$	$\Theta(n)$
Ordered Linked List	$\Theta(n)$	$\Theta(1)$

Simple Data Structures for implementing a Priority Queue

Worst Case Time For	Insert	Extract_Max
Unordered Linked List	**$\Theta(1)$**	**$\Theta(n)$**
Ordered Linked List	**$\Theta(n)$**	**$\Theta(1)$**

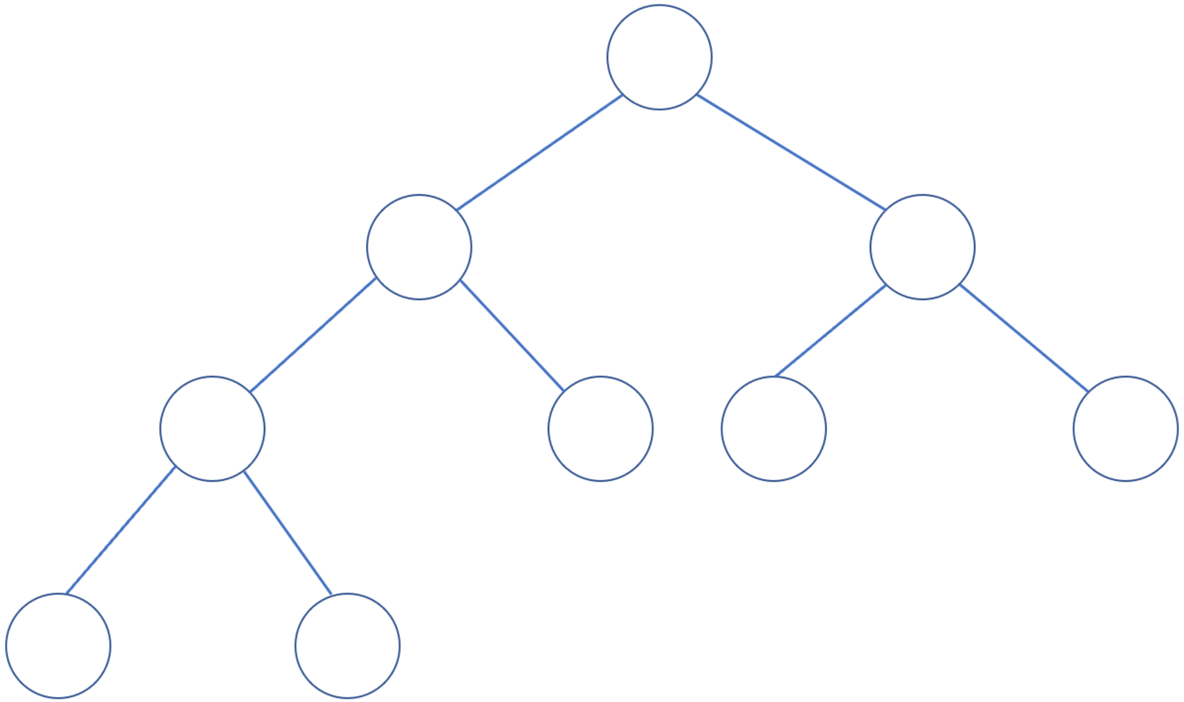
Goal: Data Structure that does each operation in $\Theta(\log n)$ time

Max-Heaps

Visualizing Max-Heaps

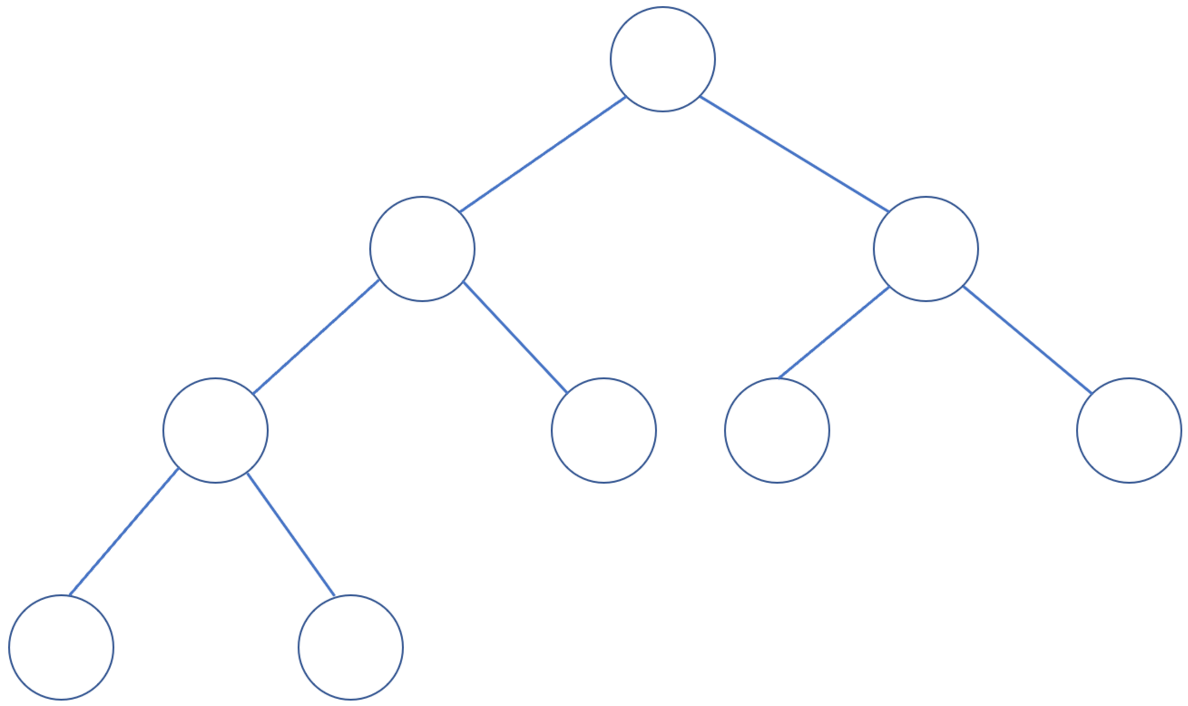
Elements of Max-Heap are stored in a Complete Binary Tree.

Complete Binary Tree (CBT)



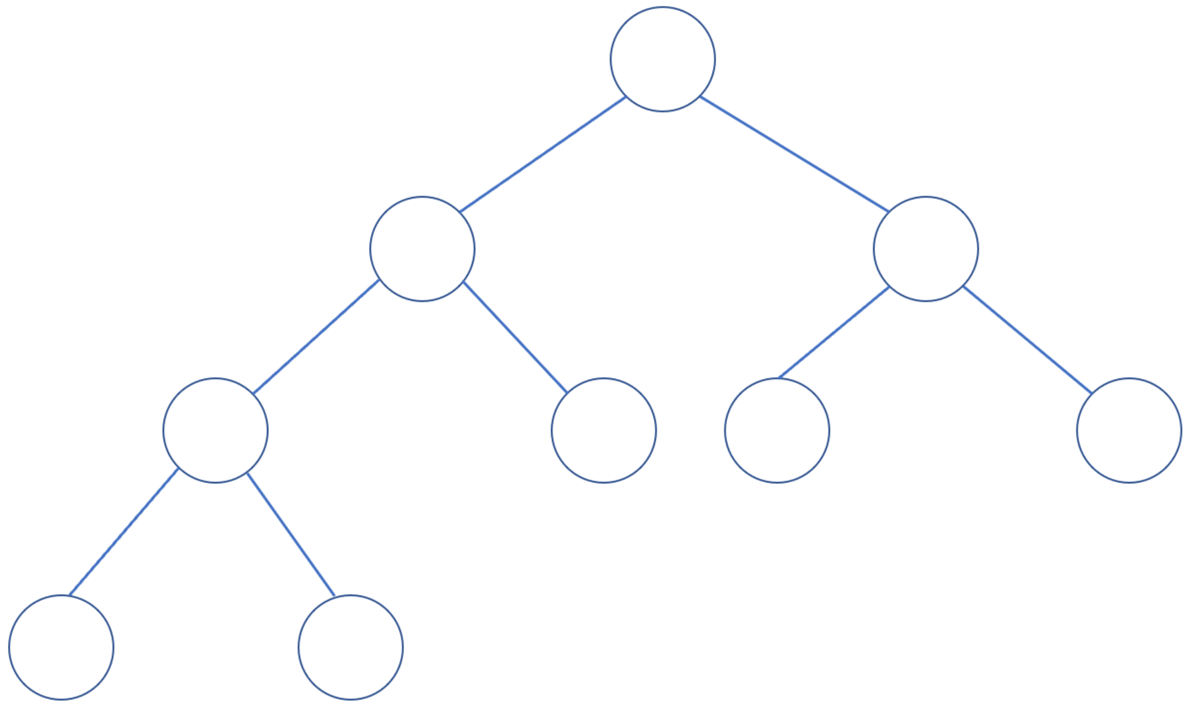
Complete Binary Tree (CBT)

- A Binary Tree



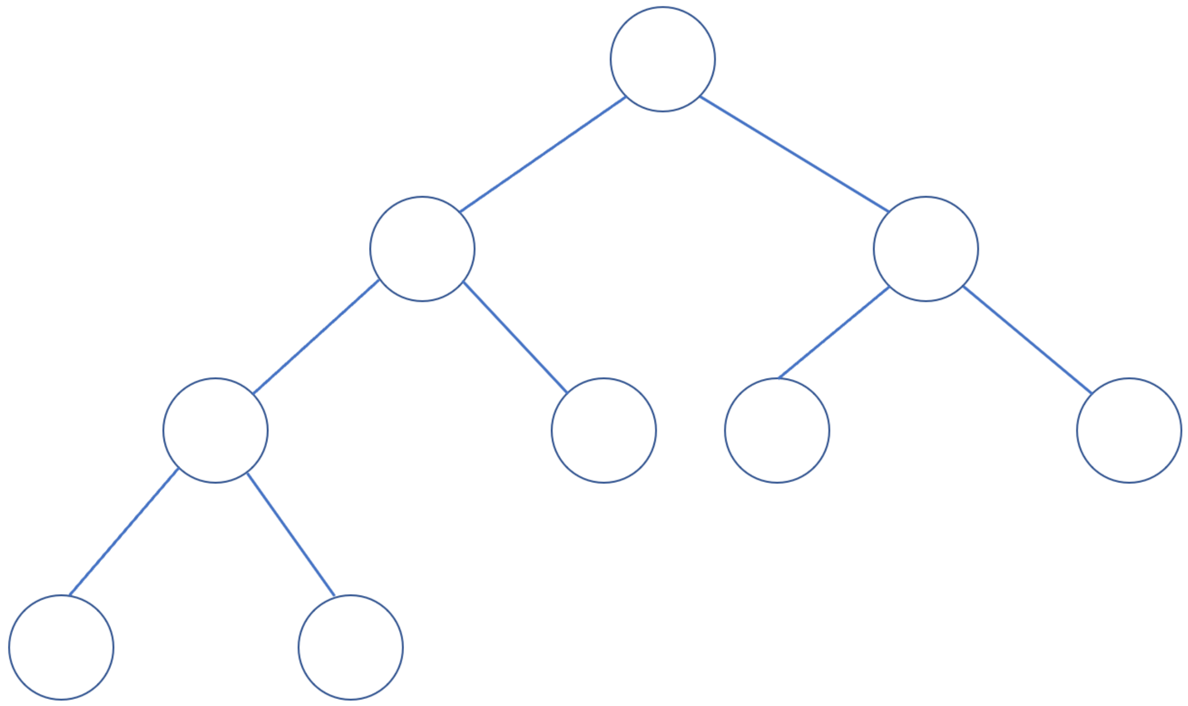
Complete Binary Tree (CBT)

- A Binary Tree
- Every level L, except maybe the bottom level, has 2^L nodes.

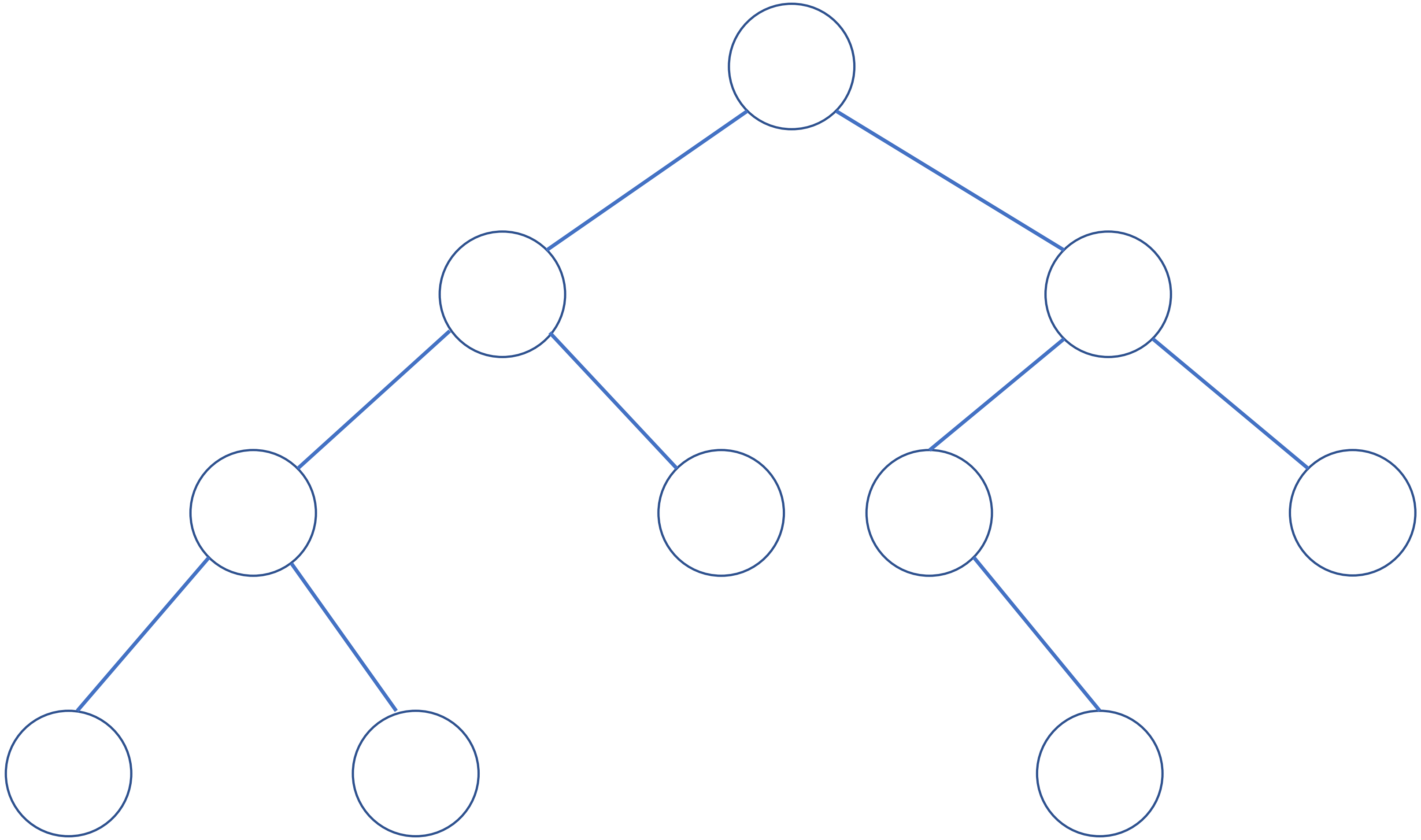


Complete Binary Tree (CBT)

- A Binary Tree
- Every level L, except maybe the bottom level, has 2^L nodes.
- All the nodes in the bottom level are as far to the left as possible.

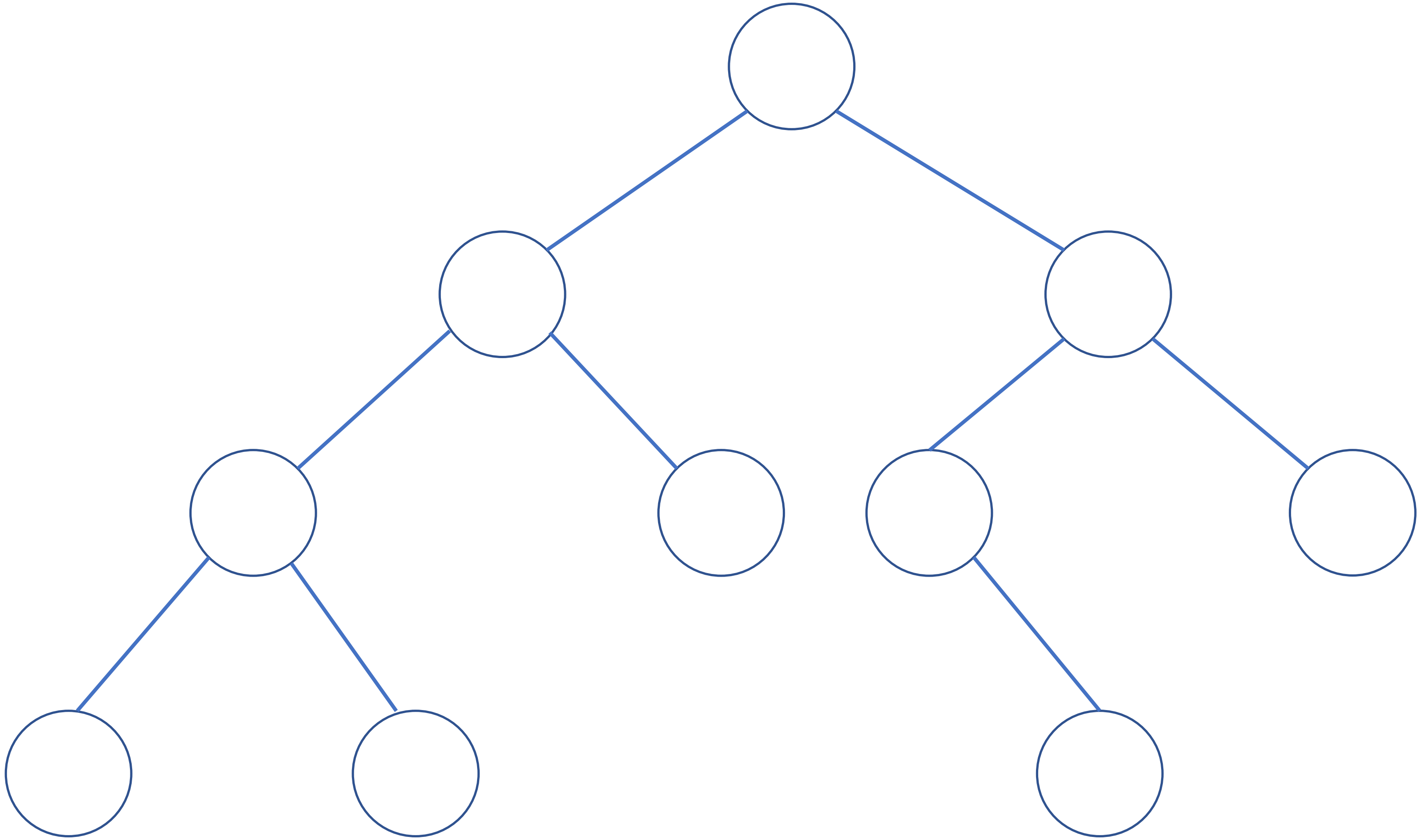


Is this a Complete Binary Tree?

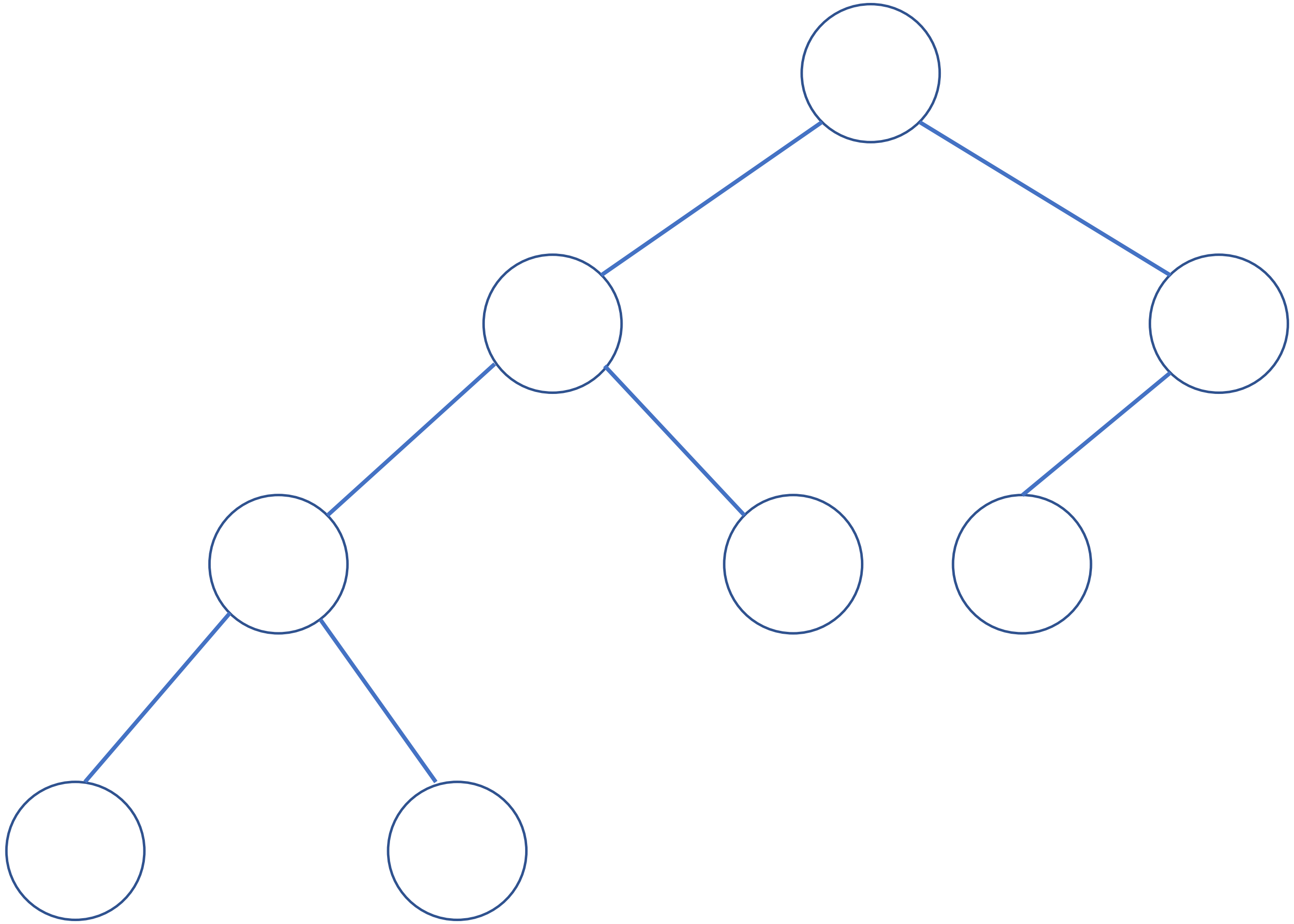


Is this a Complete Binary Tree?

No!

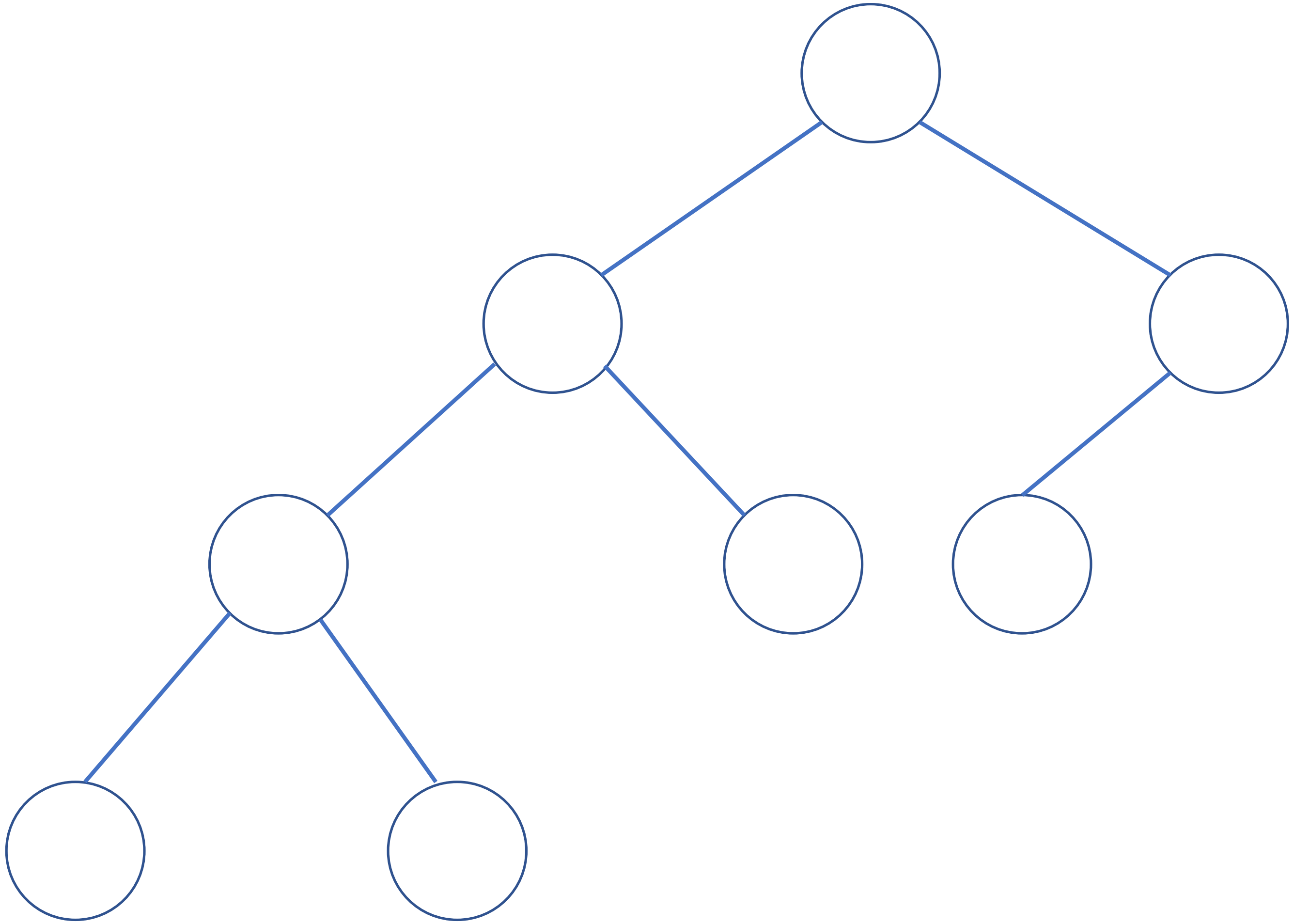


Is this a Complete Binary Tree?

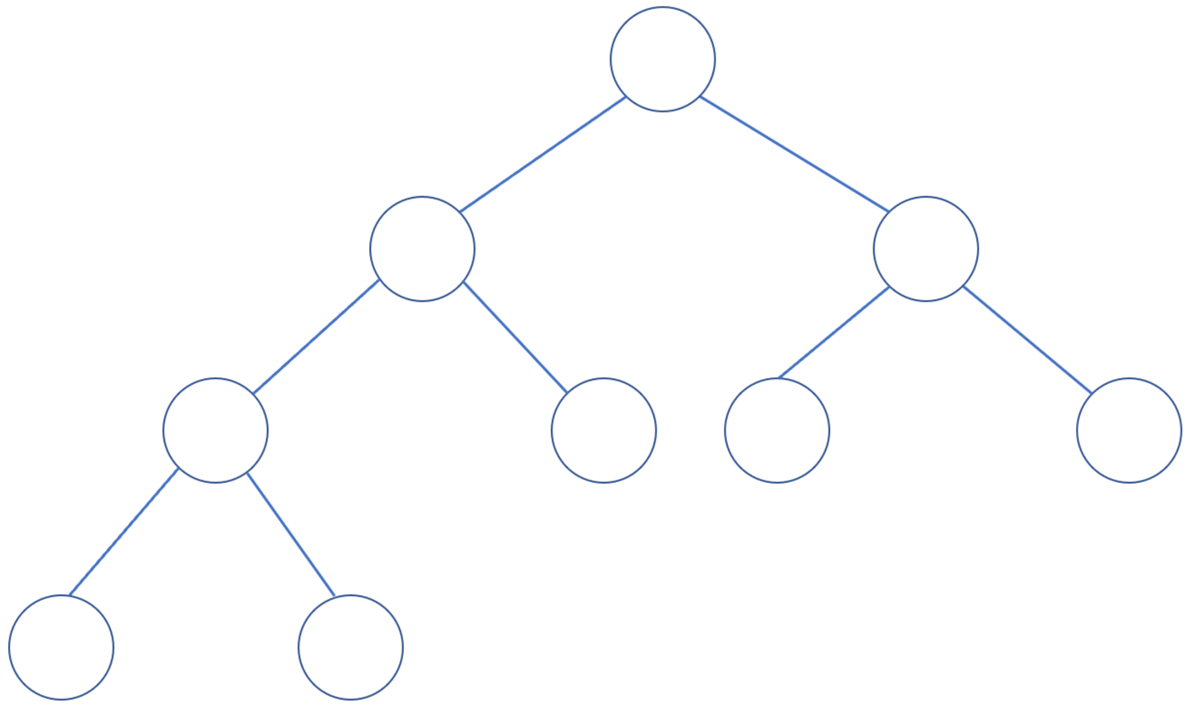


Is this a Complete Binary Tree?

No!

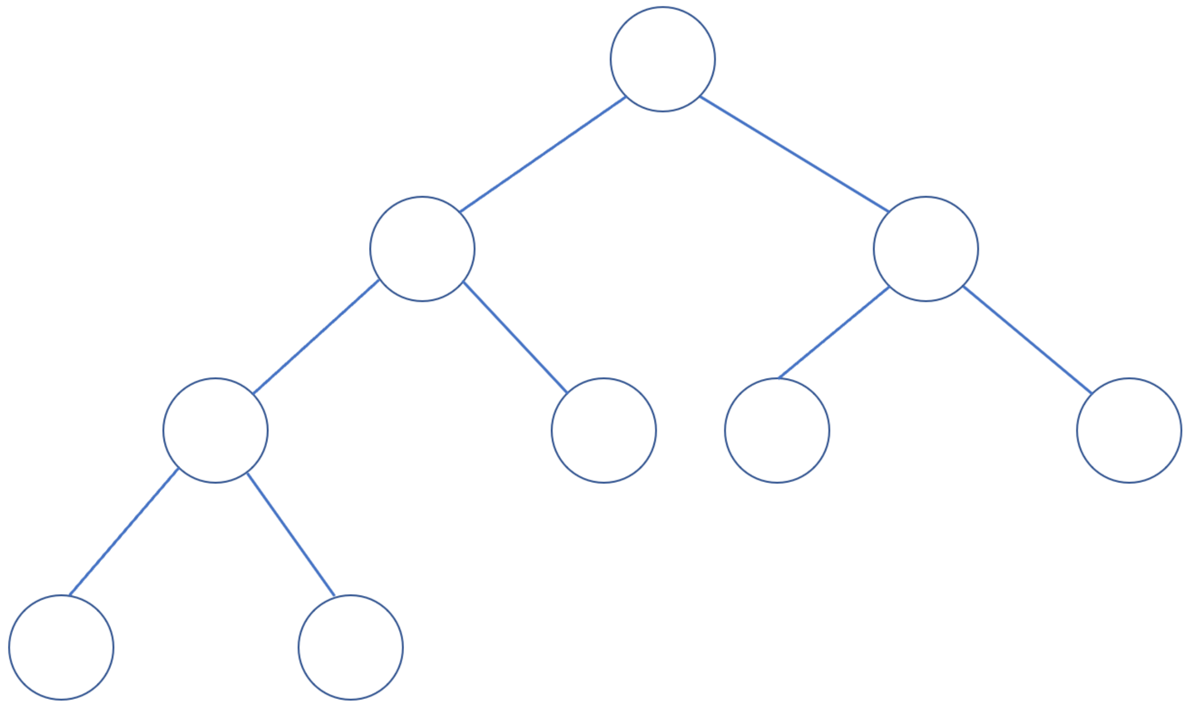


Height of a Tree



Height of a Tree

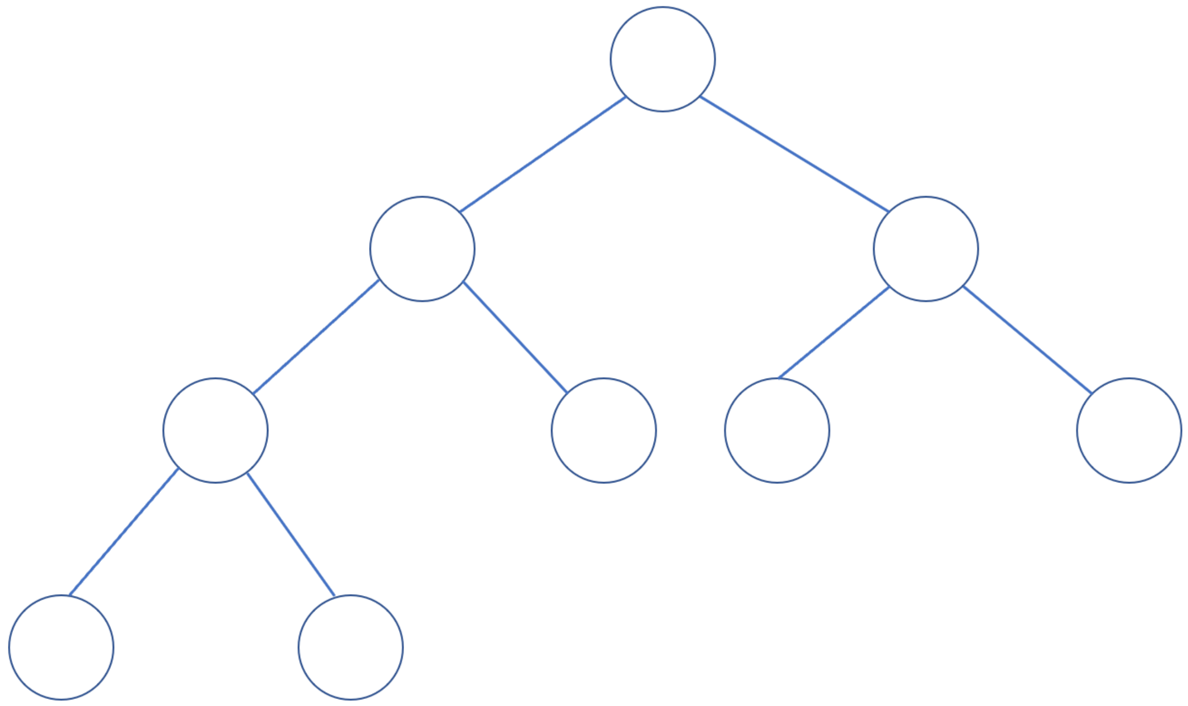
Number of edges in the longest path from root to any leaf.



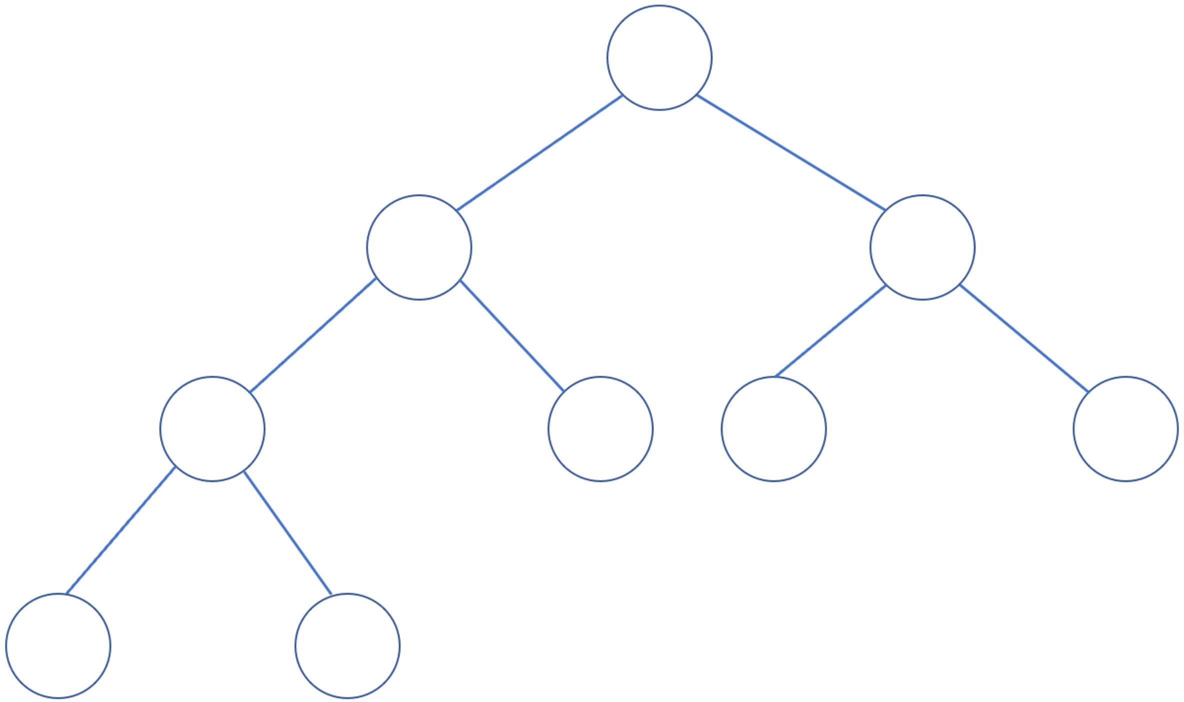
Height of a Tree

Number of edges in the longest path from root to any leaf.

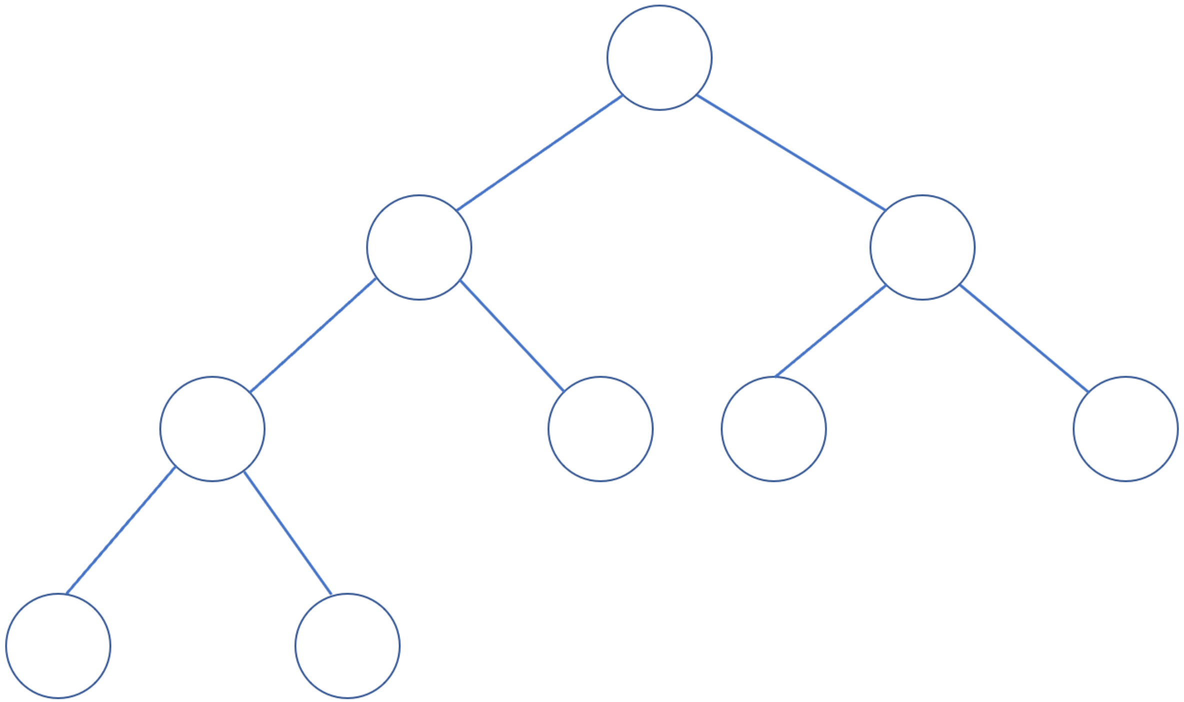
Height: 3



FACT: Height of a CBT with n nodes is $\lfloor \log_2 n \rfloor$



FACT: Height of a CBT with n nodes is $\lfloor \log_2 n \rfloor$



Here n = 9, and height = $\lfloor \log_2 9 \rfloor = 3$

Max-Heap: The n elements of a Max-Heap are stored in a CBT with n nodes such that the following property holds.

Max-Heap Property: Priority of each node >= Priority of its children

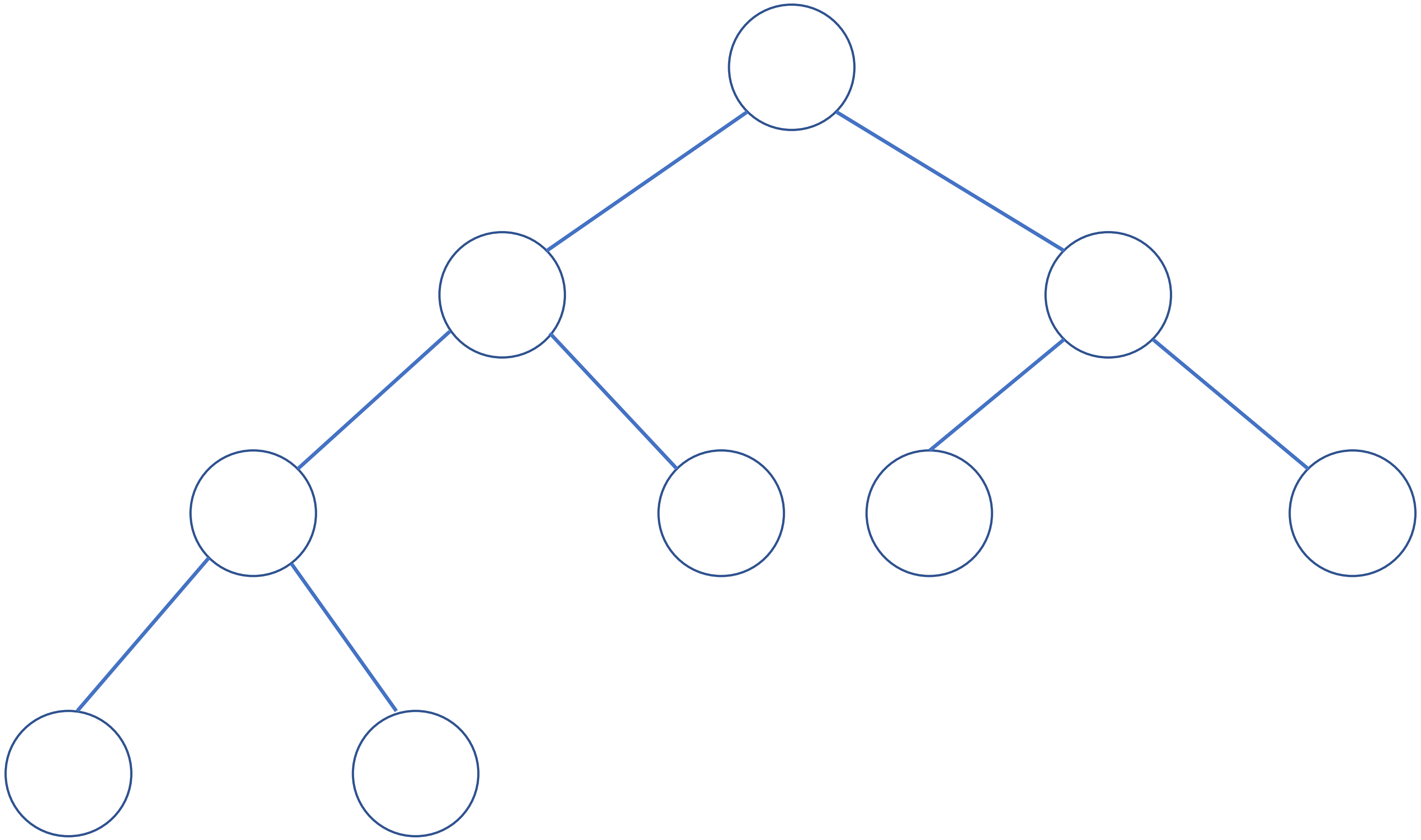
Example of a Max-Heap

S = {3, 4, 5, 7, 7, 9, 9, 12, 17}

(for simplicity, we identify each element with its priority)

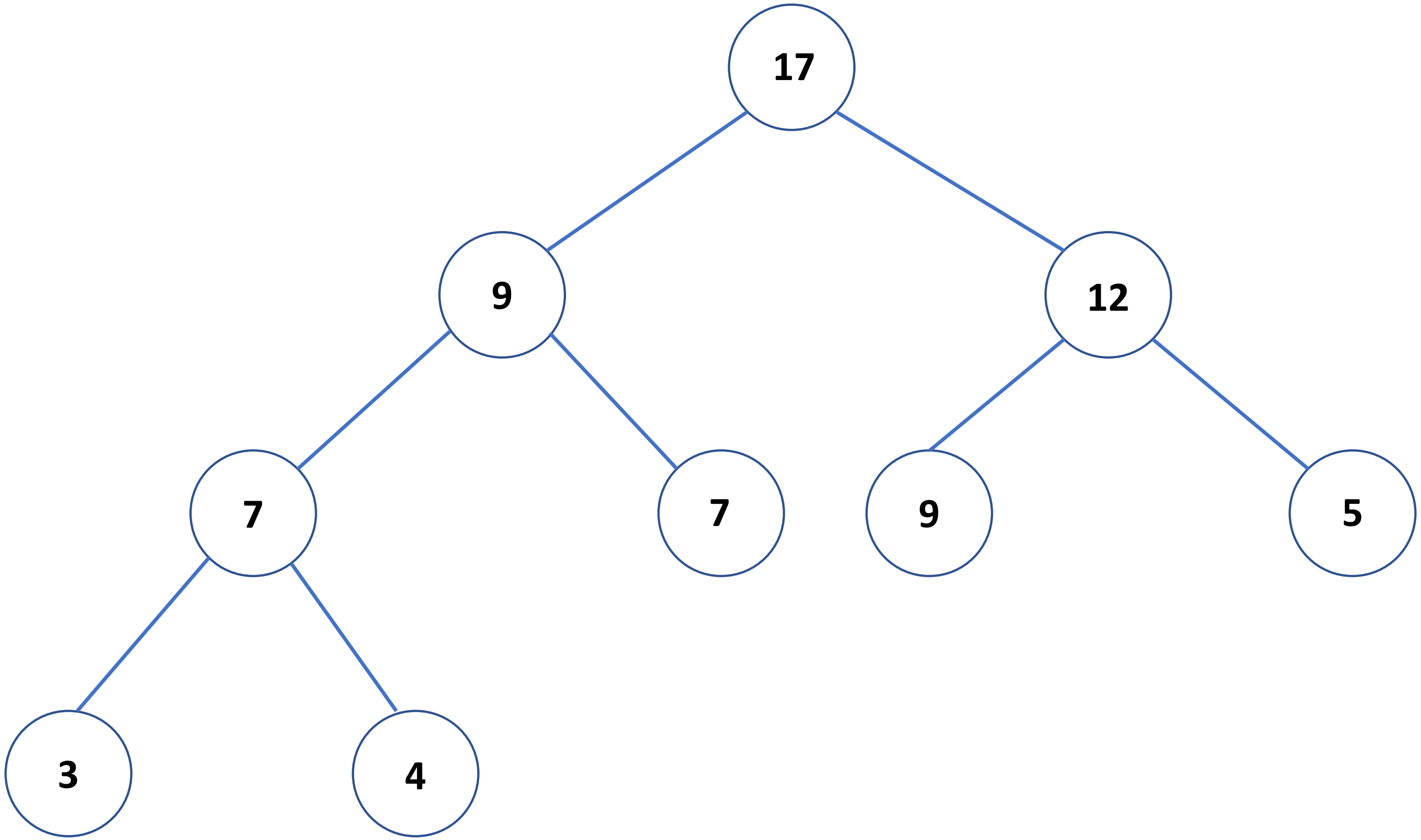
Example of a Max-Heap

$S = \{3, 4, 5, 7, 7, 9, 9, 12, 17\}$



Example of a Max-Heap

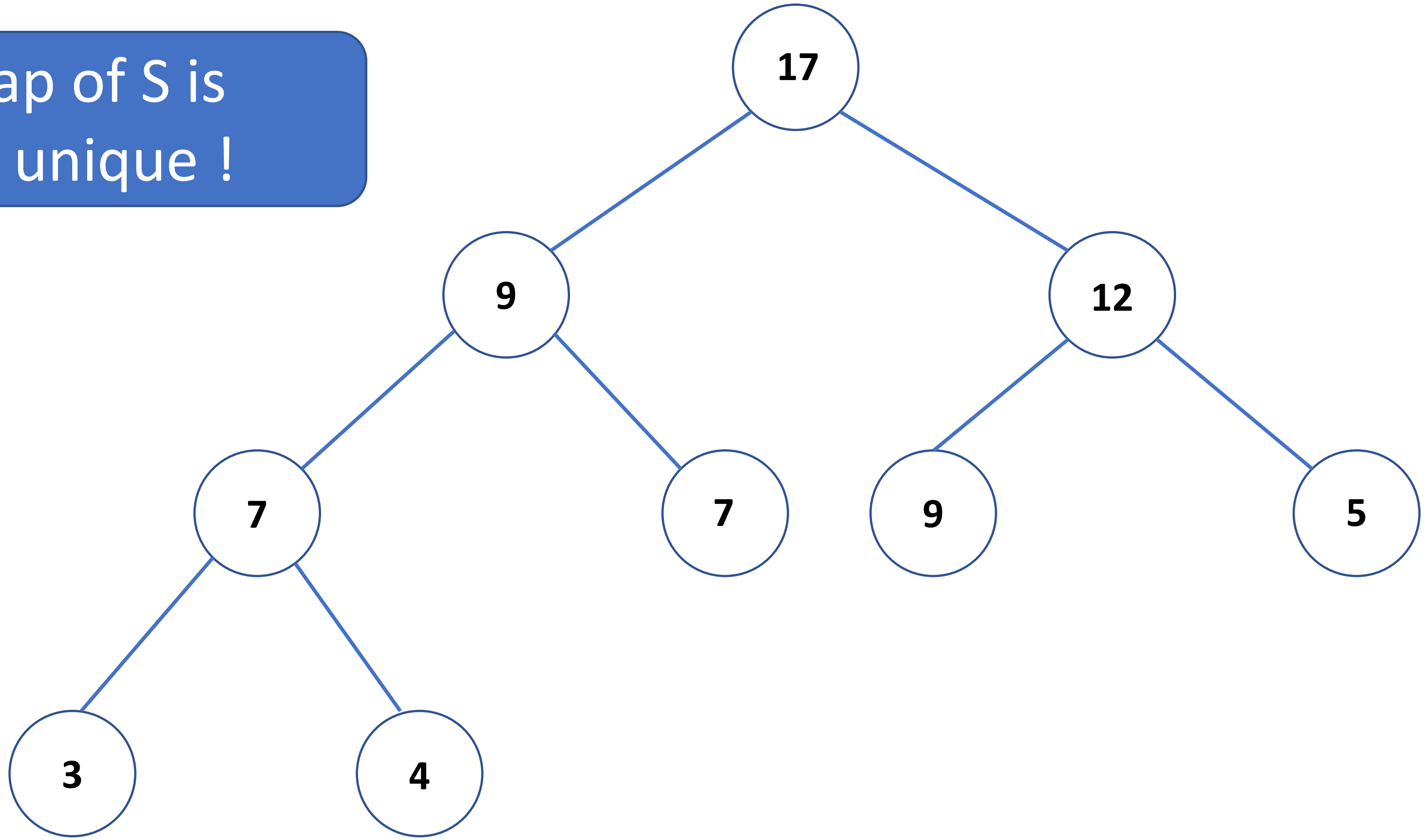
S = {3, 4, 5, 7, 7, 9, 9, 12, 17}



Example of a Max-Heap

$S = \{3, 4, 5, 7, 7, 9, 9, 12, 17\}$

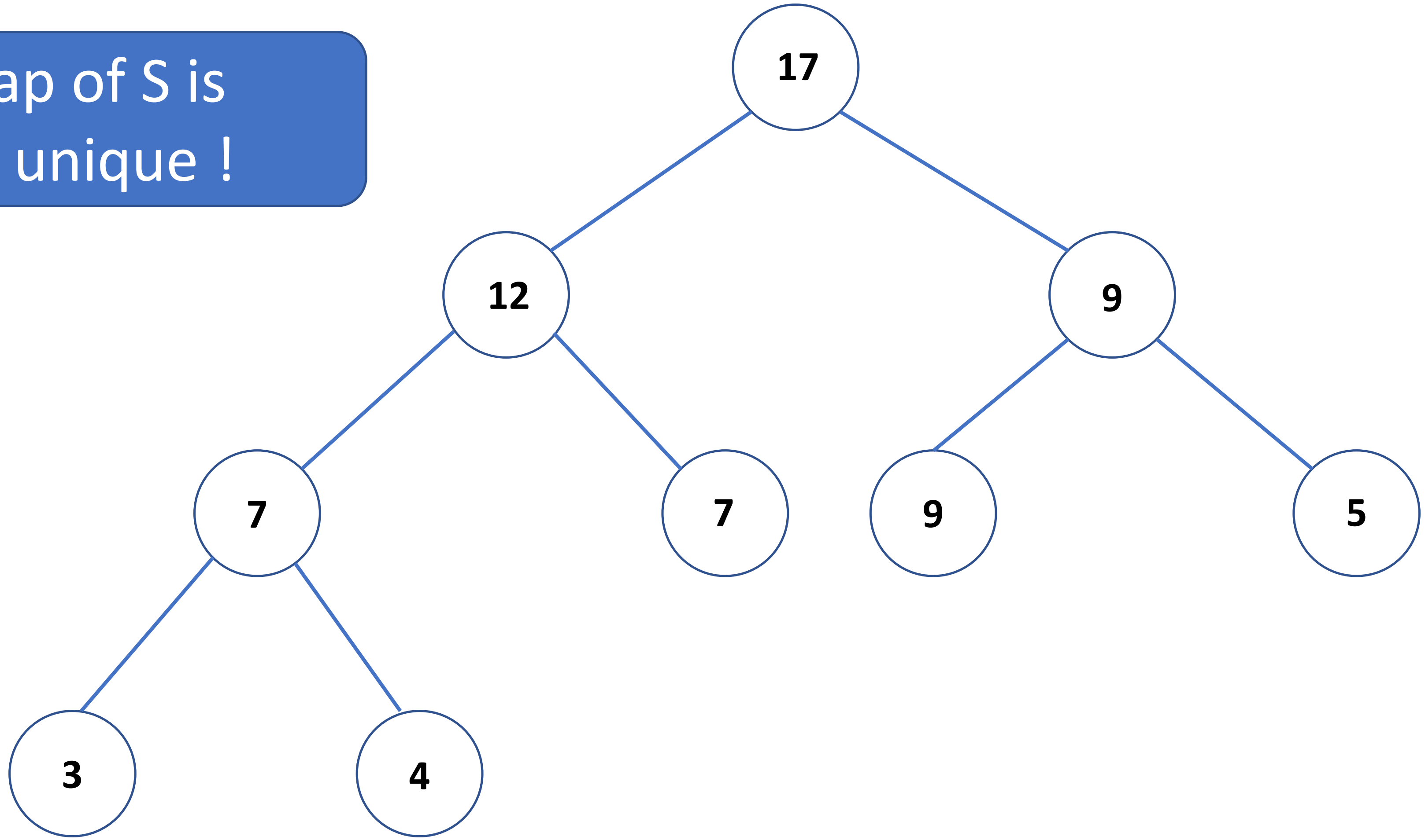
Heap of S is not unique !



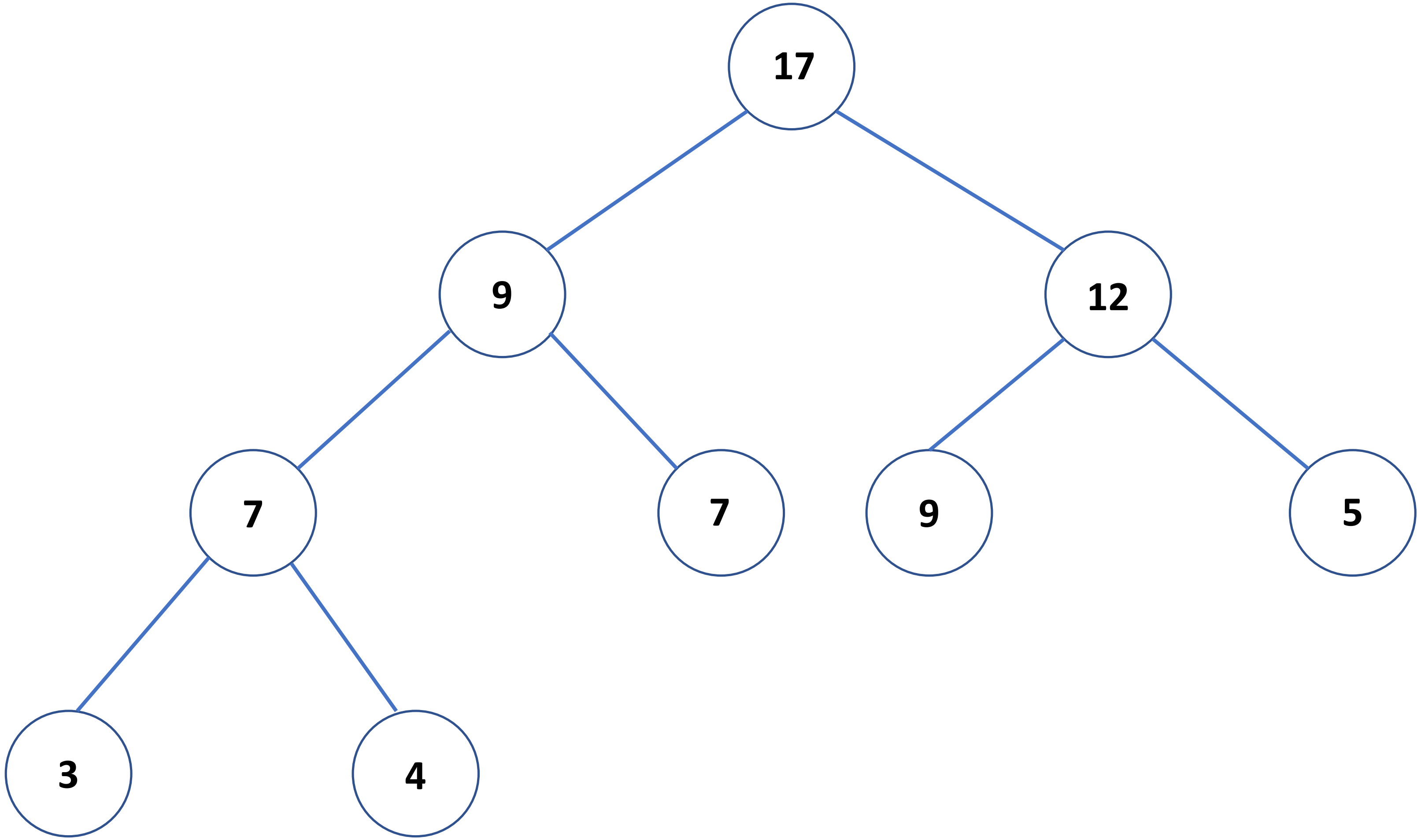
Example of a Max-Heap

S = {3, 4, 5, 7, 7, 9, 9, 12, 17}

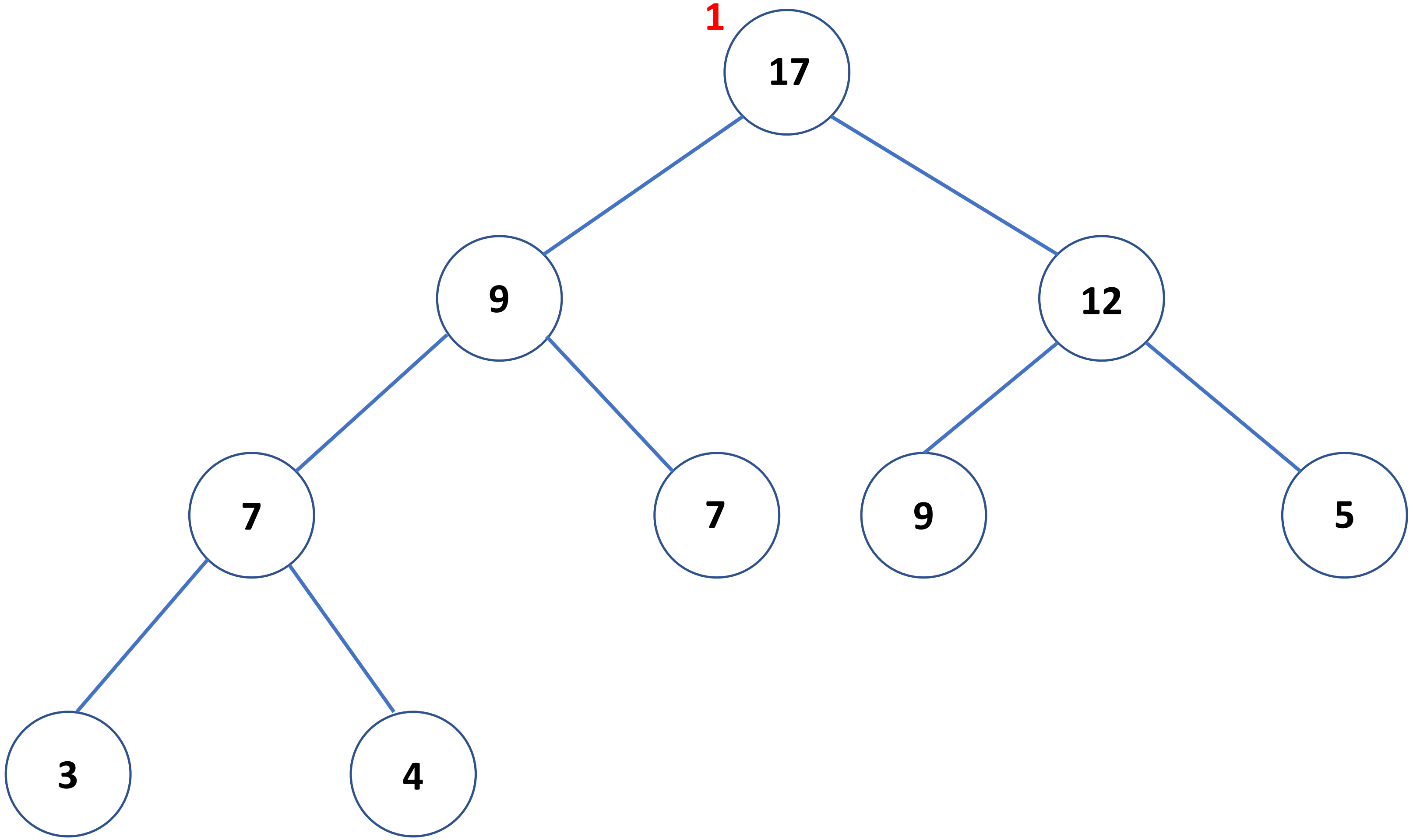
Heap of S is not unique !



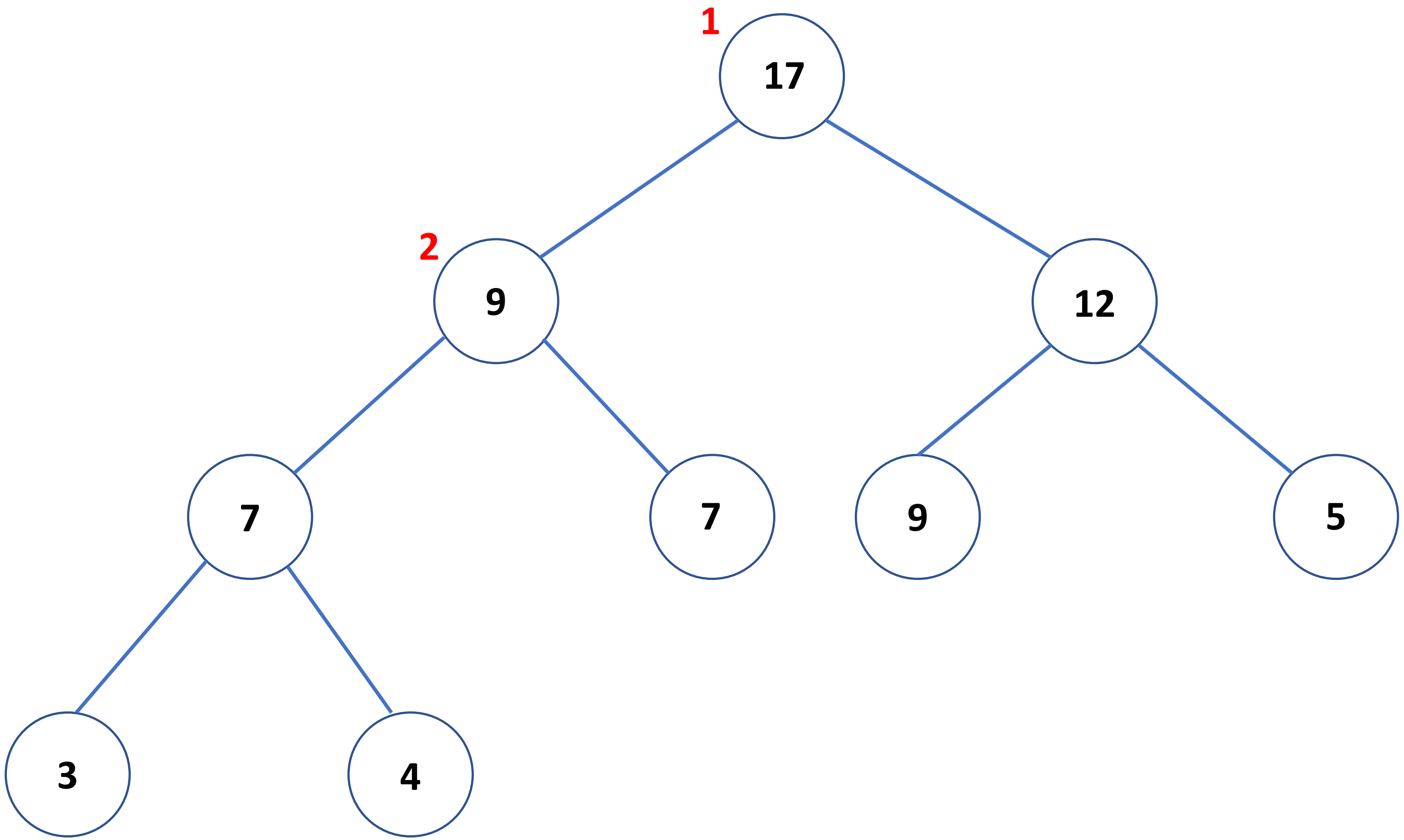
Array Representation of a Heap



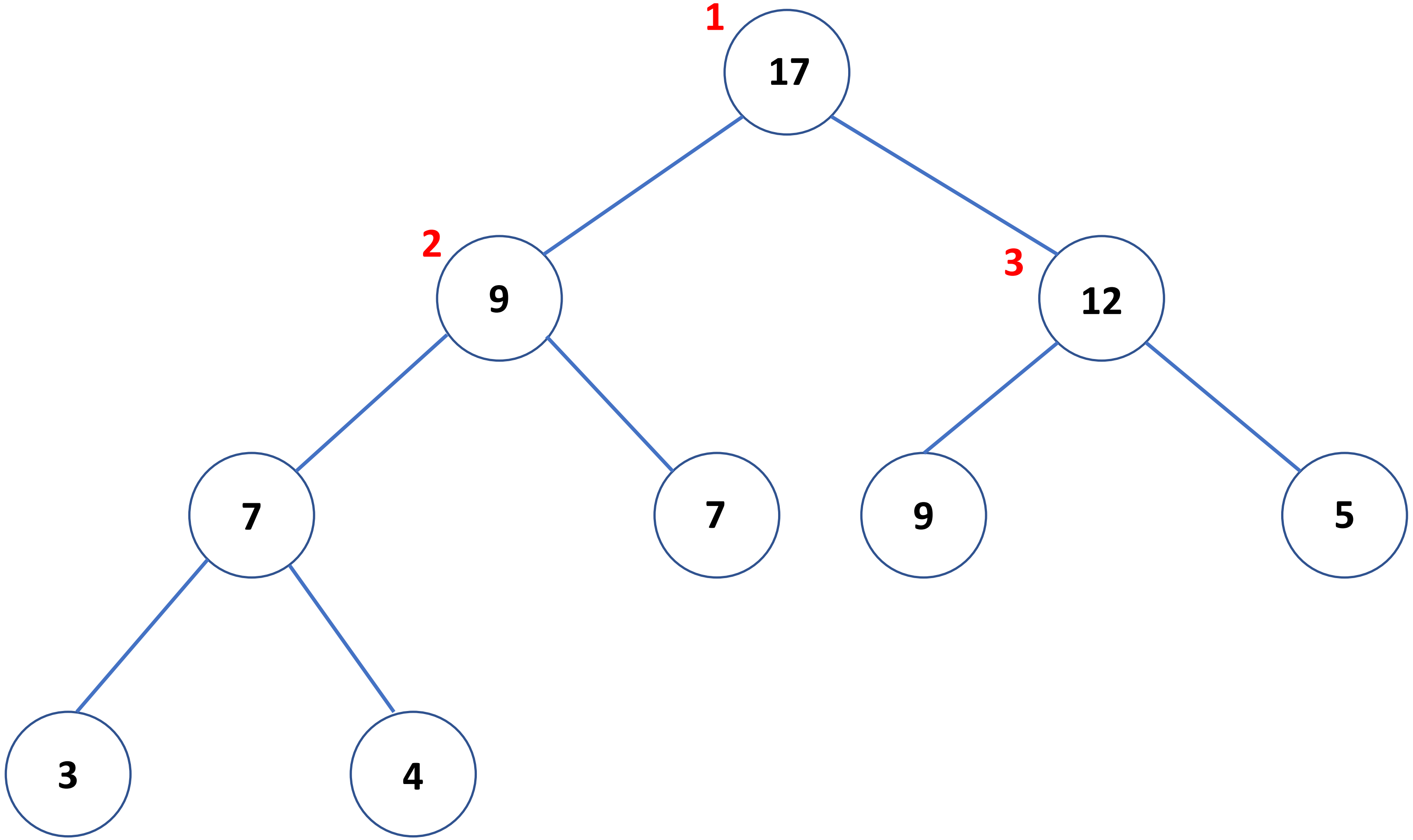
A:									
Index	1	2	3	4	5	6	7	8	9



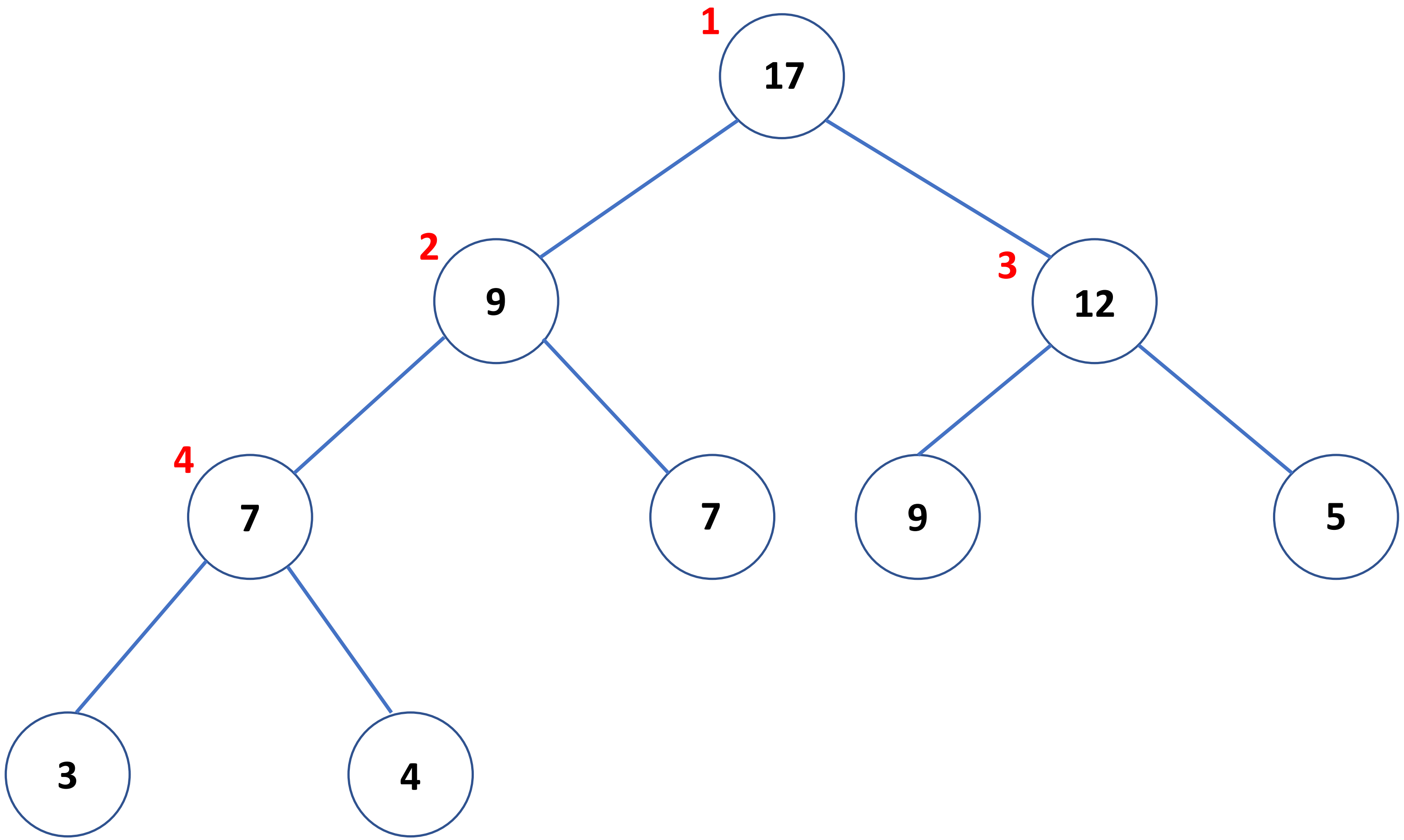
A:	17								
Index	1	2	3	4	5	6	7	8	9



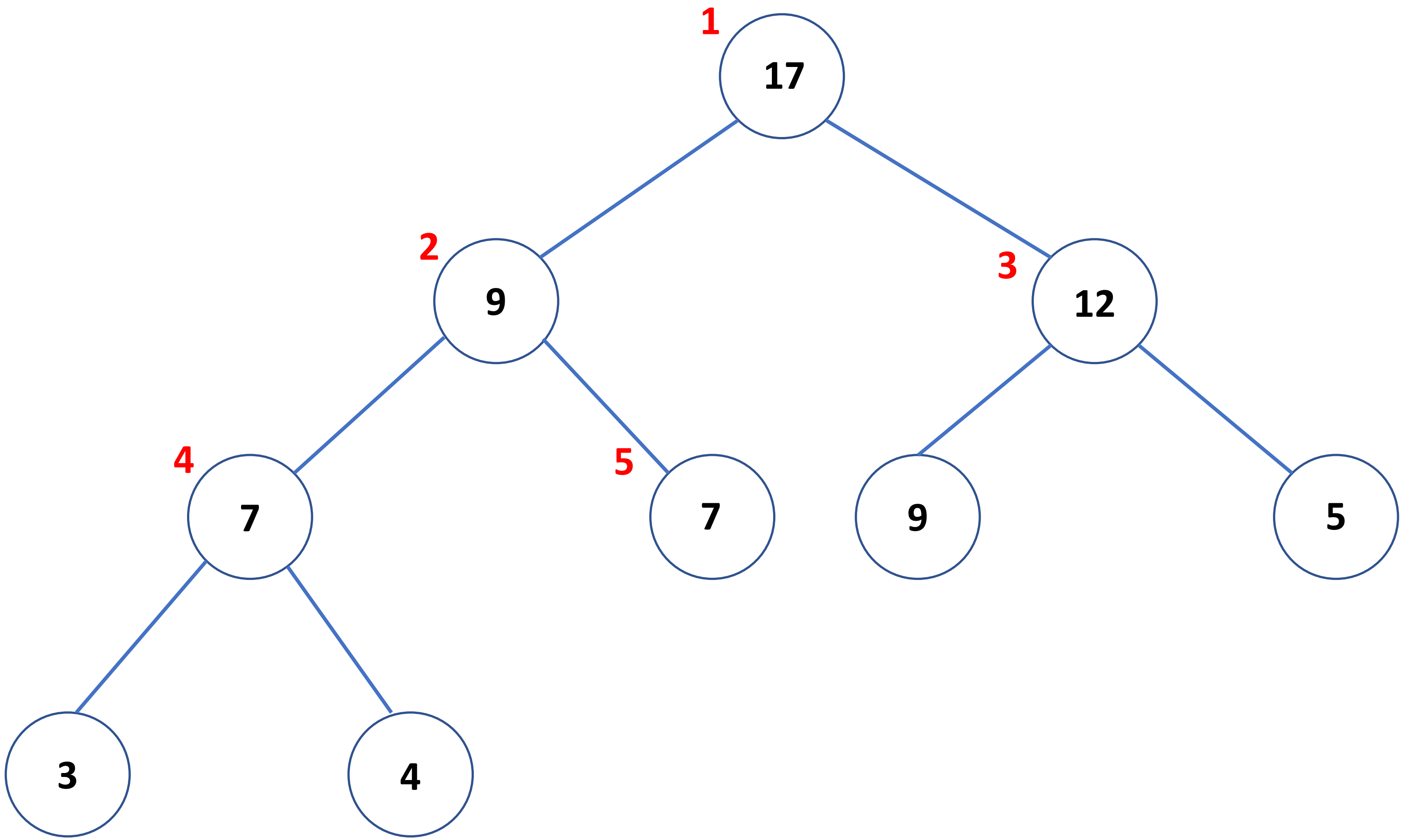
A:	17	9							
Index	1	2	3	4	5	6	7	8	9



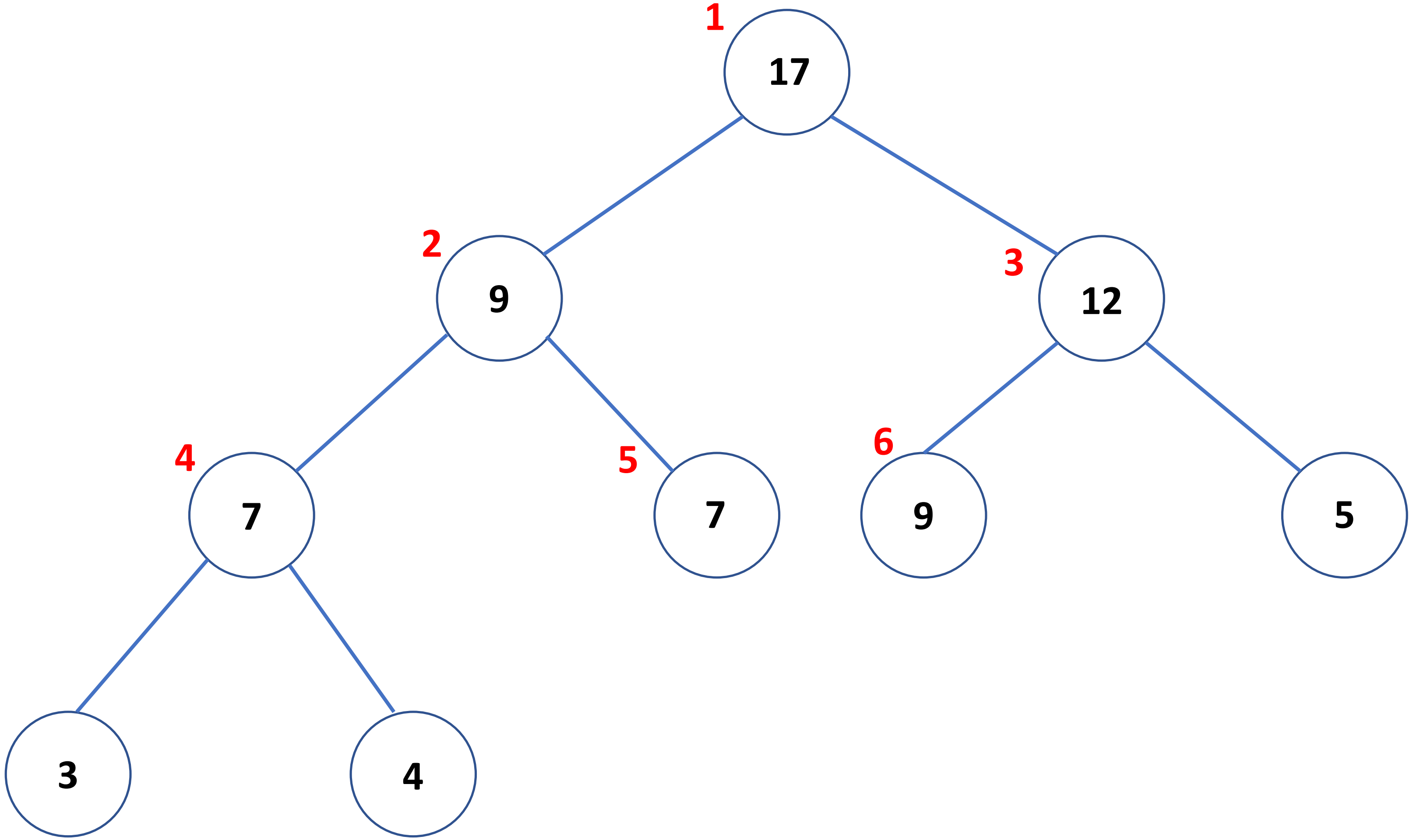
A:	17	9	12						
Index	1	2	3	4	5	6	7	8	9



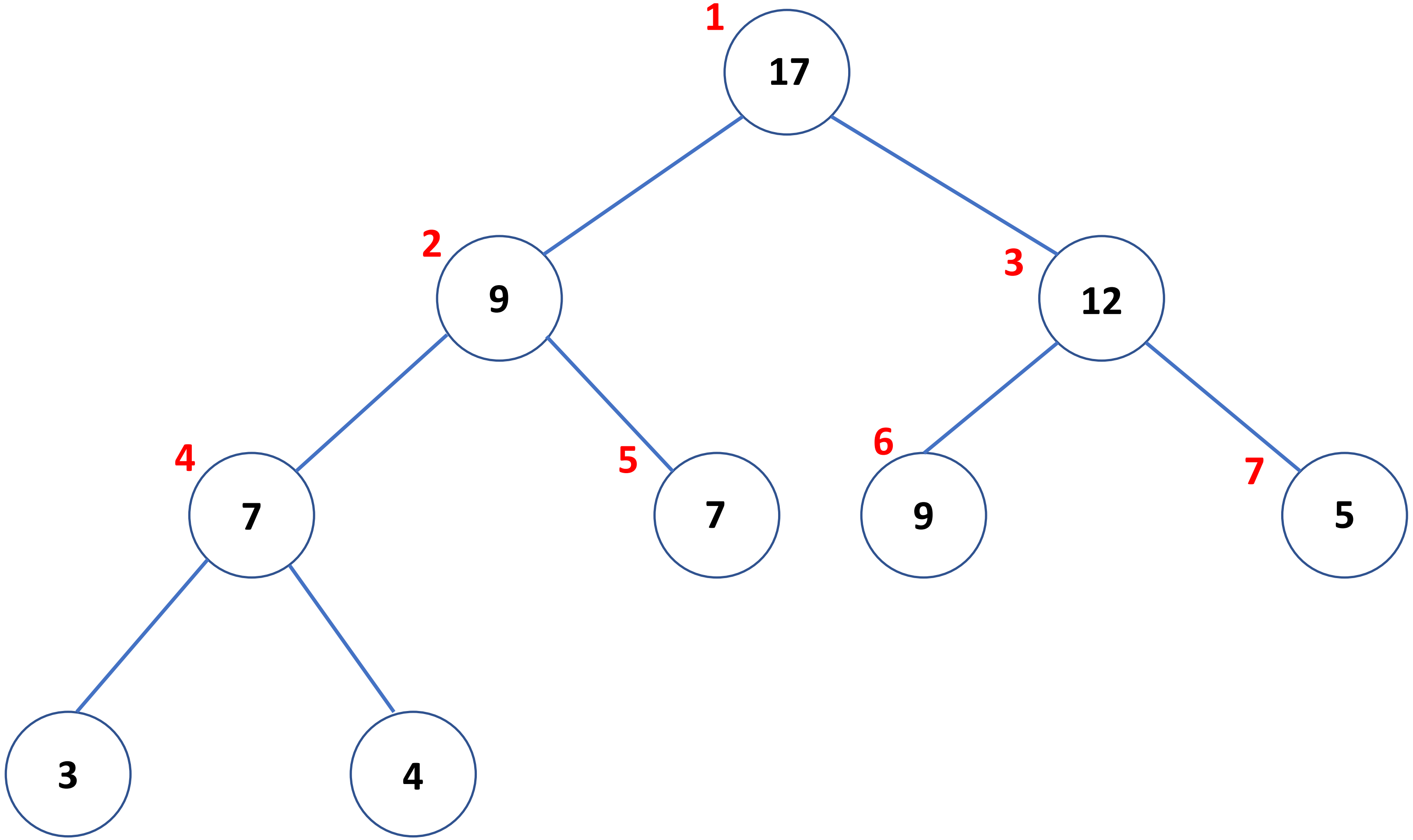
A:	17	9	12	7					
Index	1	2	3	4	5	6	7	8	9



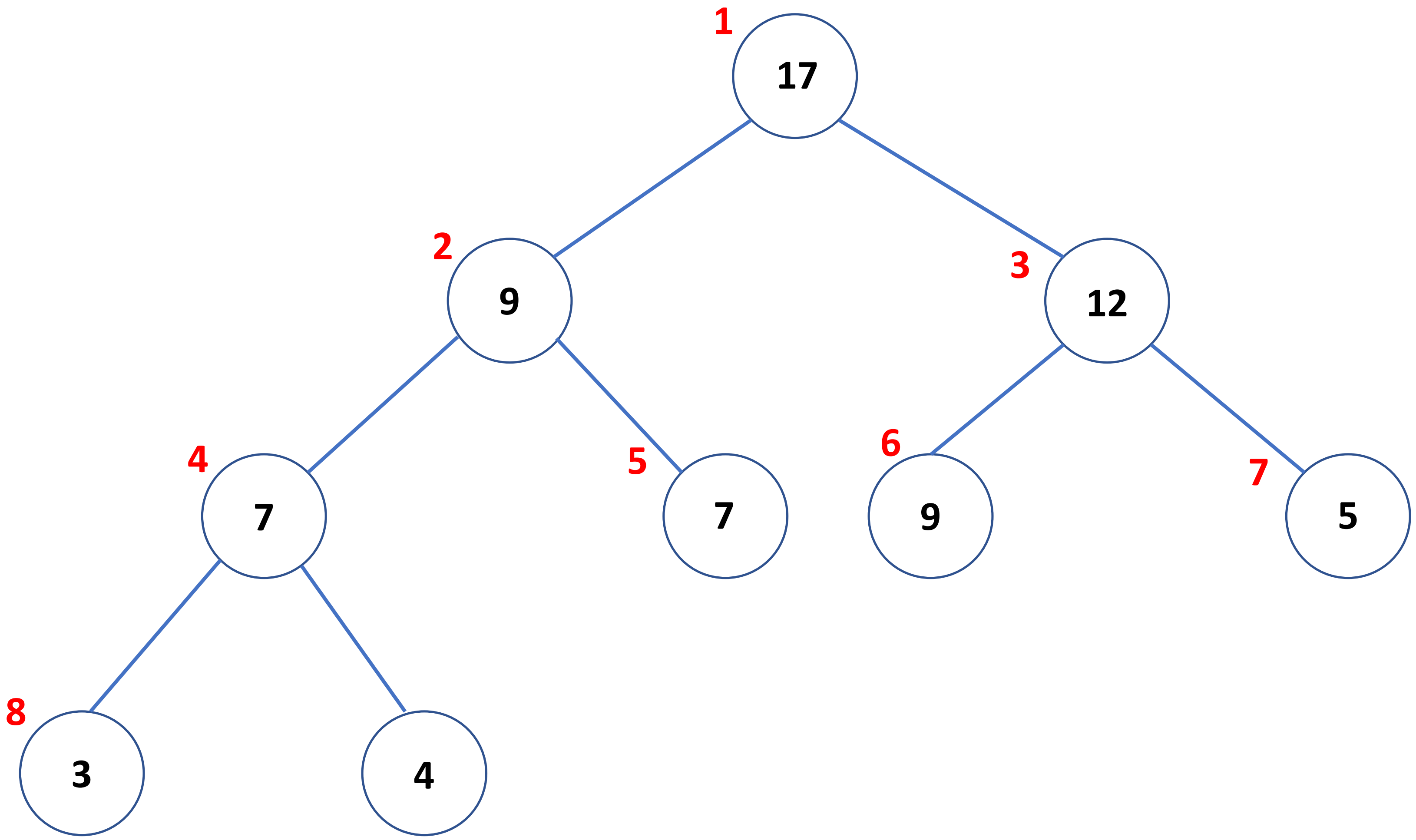
A:	17	9	12	7	7				
Index	1	2	3	4	5	6	7	8	9



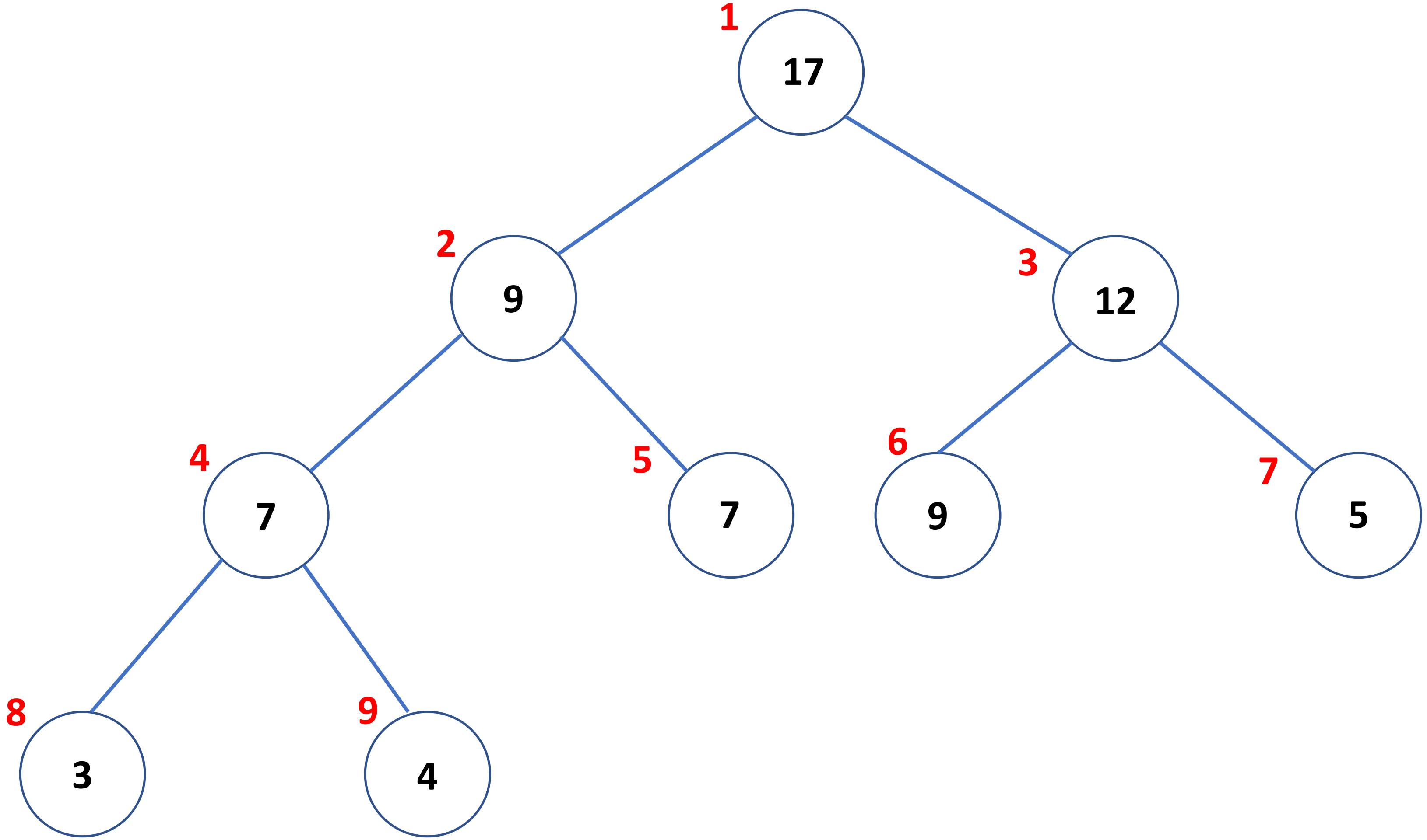
A:	17	9	12	7	7	9			
Index	1	2	3	4	5	6	7	8	9



A:	17	9	12	7	7	9	5		
Index	1	2	3	4	5	6	7	8	9

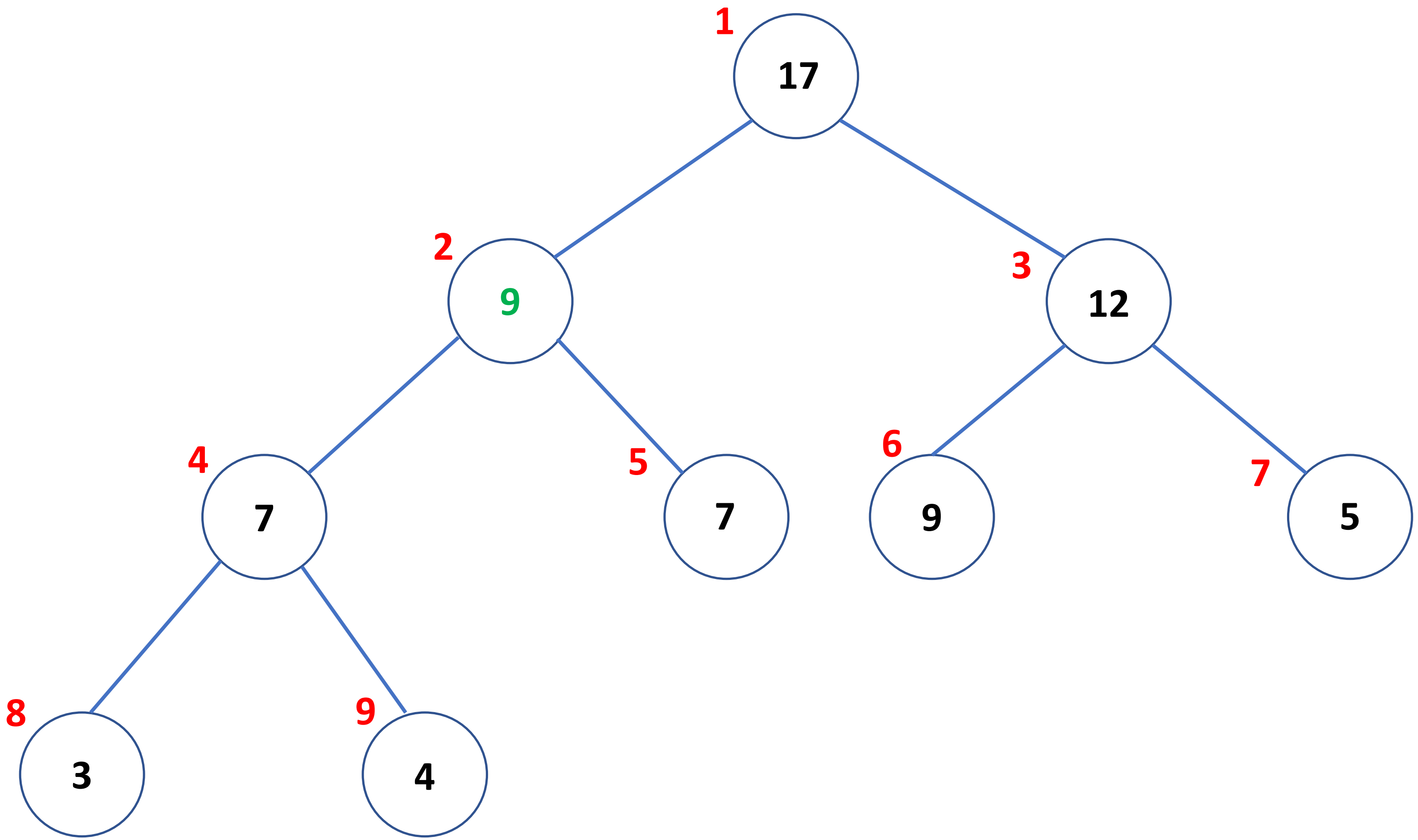


A:	17	9	12	7	7	9	5	3	
Index	1	2	3	4	5	6	7	8	9

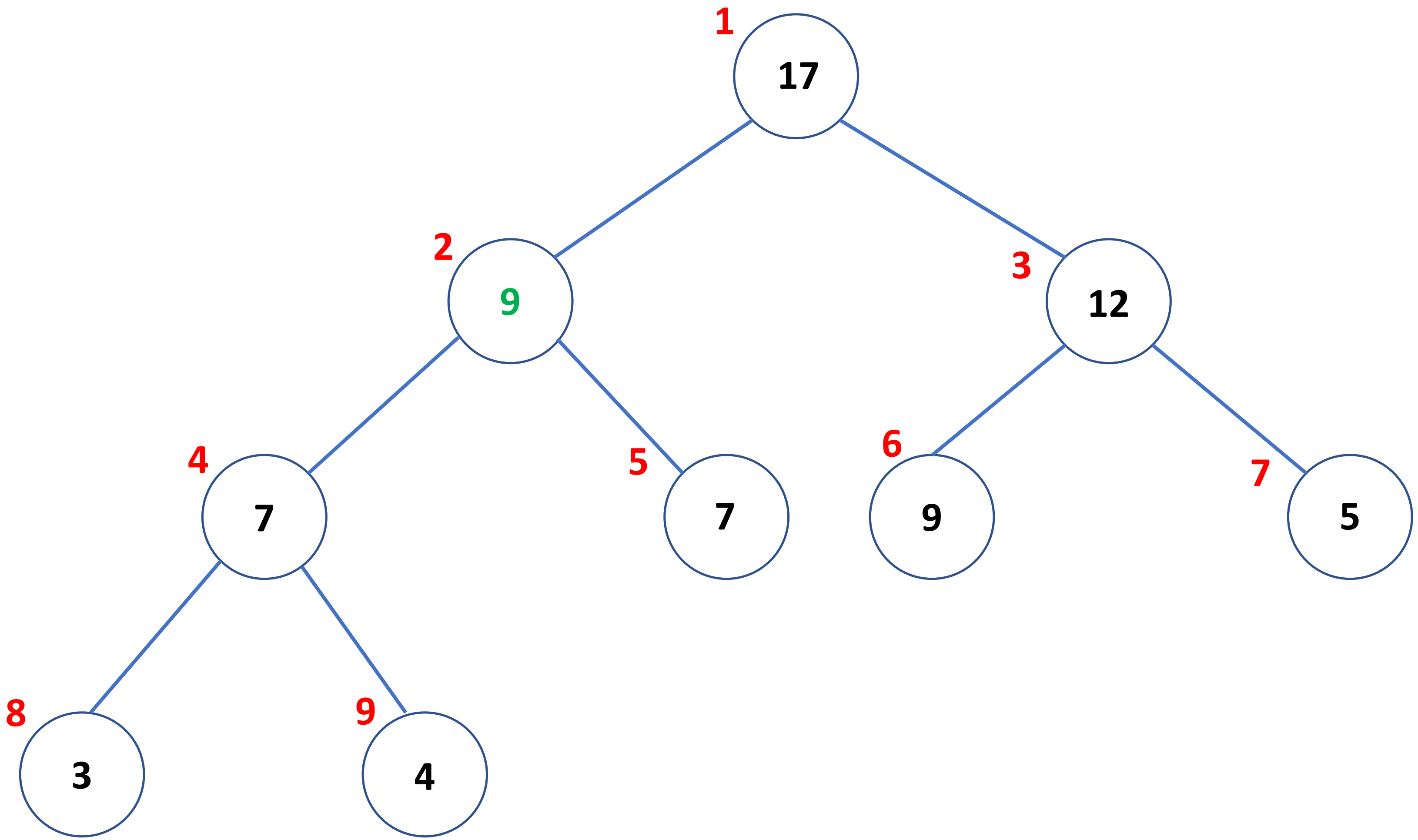


A:	17	9	12	7	7	9	5	3	4
Index	1	2	3	4	5	6	7	8	9

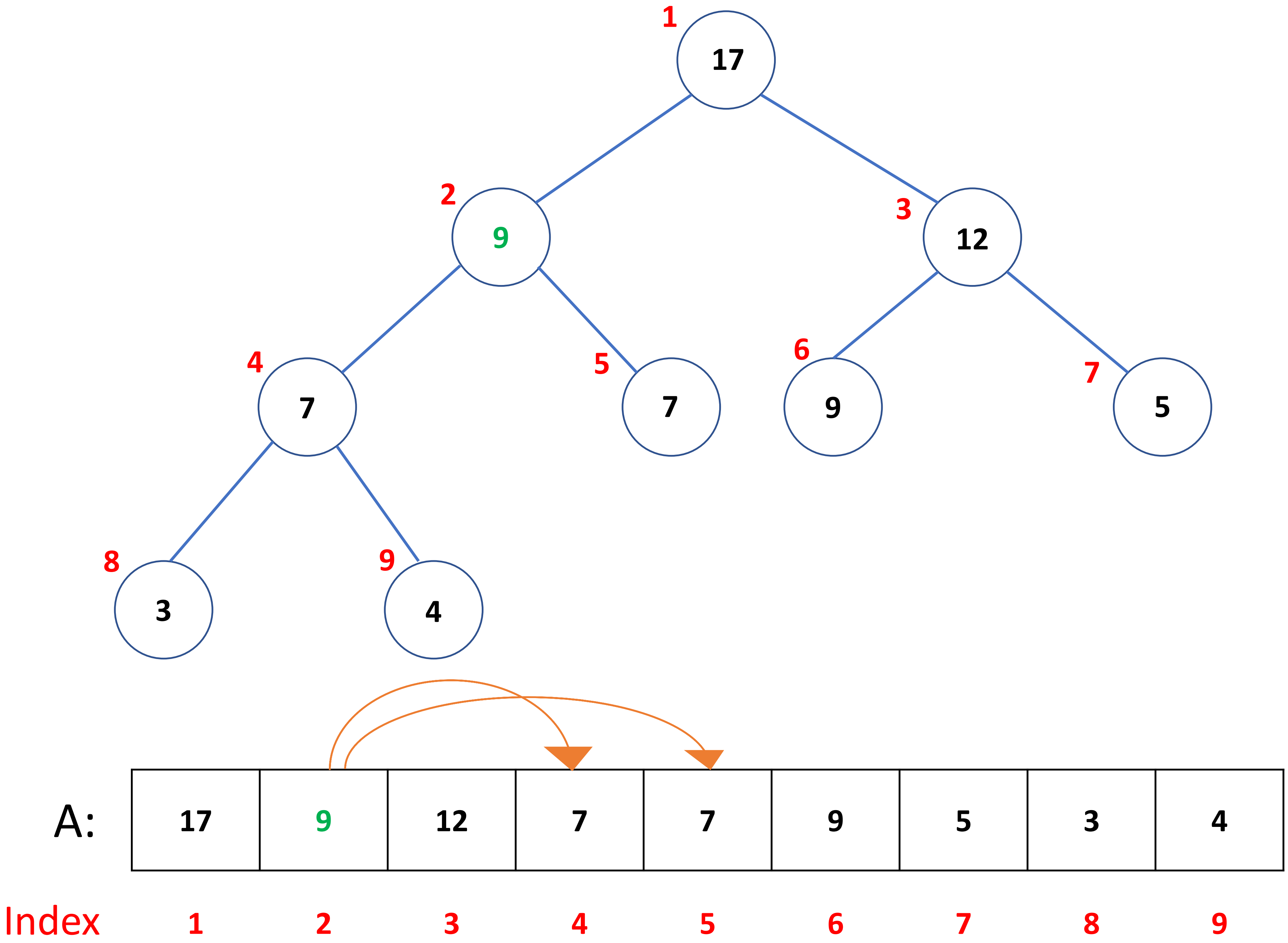
A.Heapsize = 9

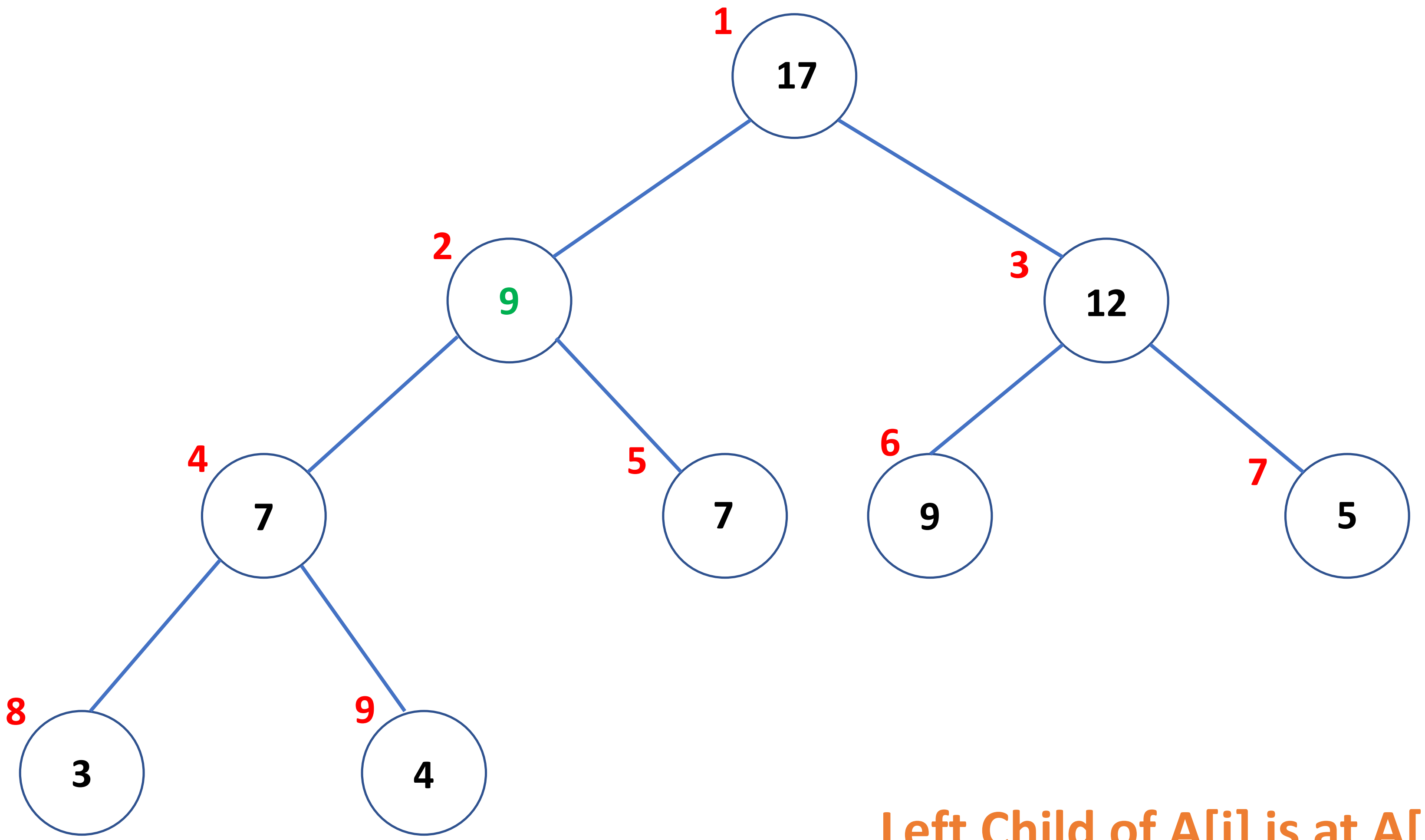


A:	17	9	12	7	7	9	5	3	4	A.Heapsize = 9
Index	1	2	3	4	5	6	7	8	9	



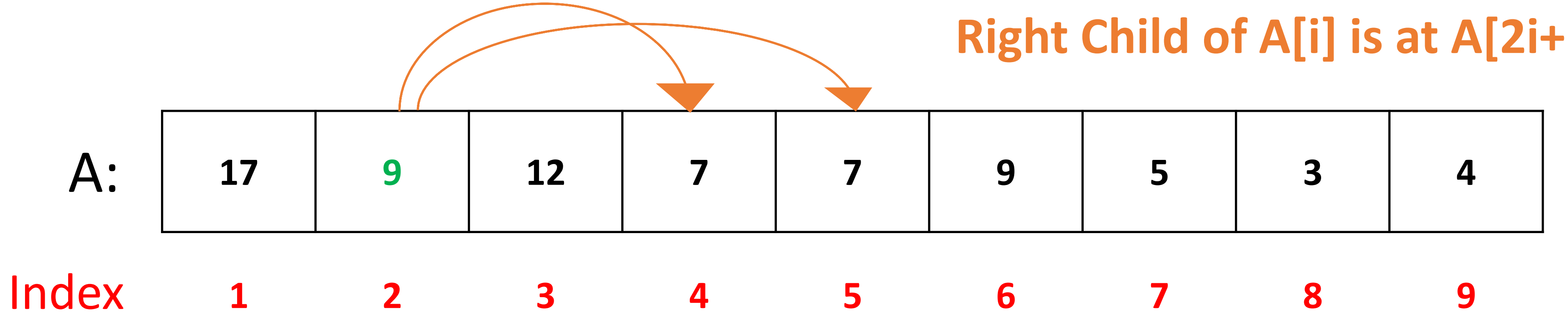
A:	17	9	12	7	7	9	5	3	4
Index	1	2	3	4	5	6	7	8	9

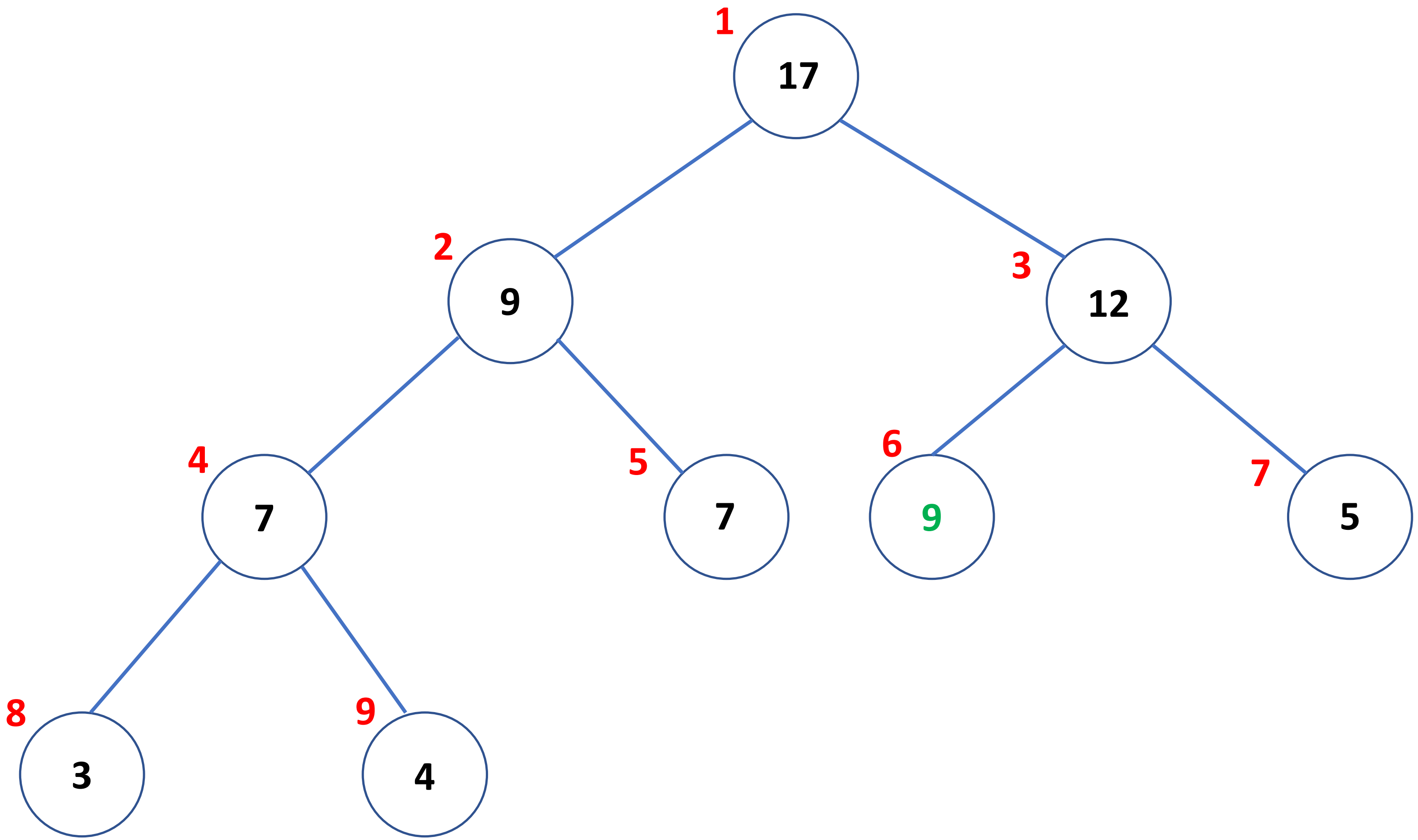

1
17
2
9
3
12
4
7
5
7
6
9
7
5
8
3
9
4
A:
17
9
12
7
7
9
5
3
4
Index
1
2
3
4
5
6
7
8
9



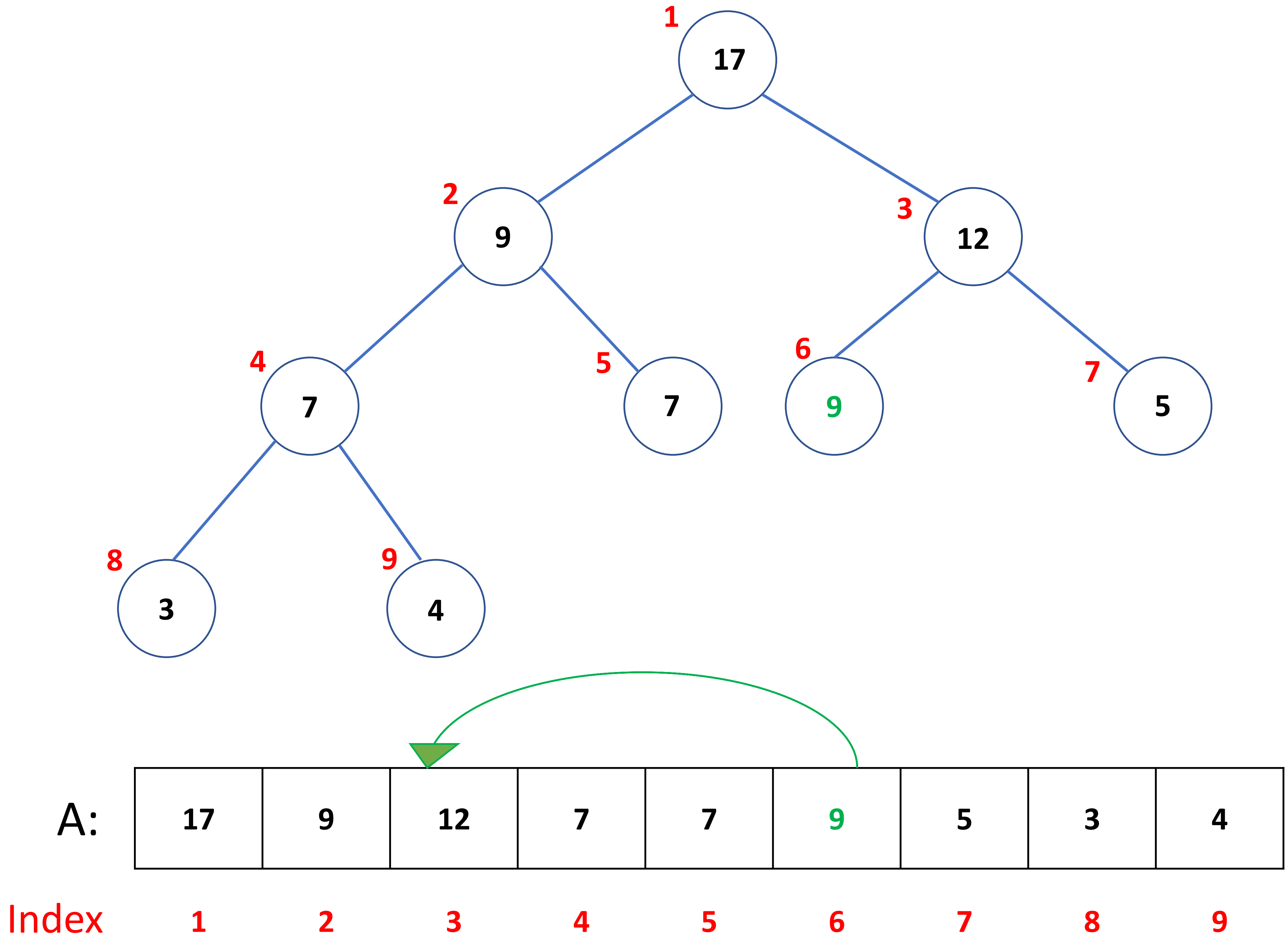
Left Child of A[i] is at A[2i]

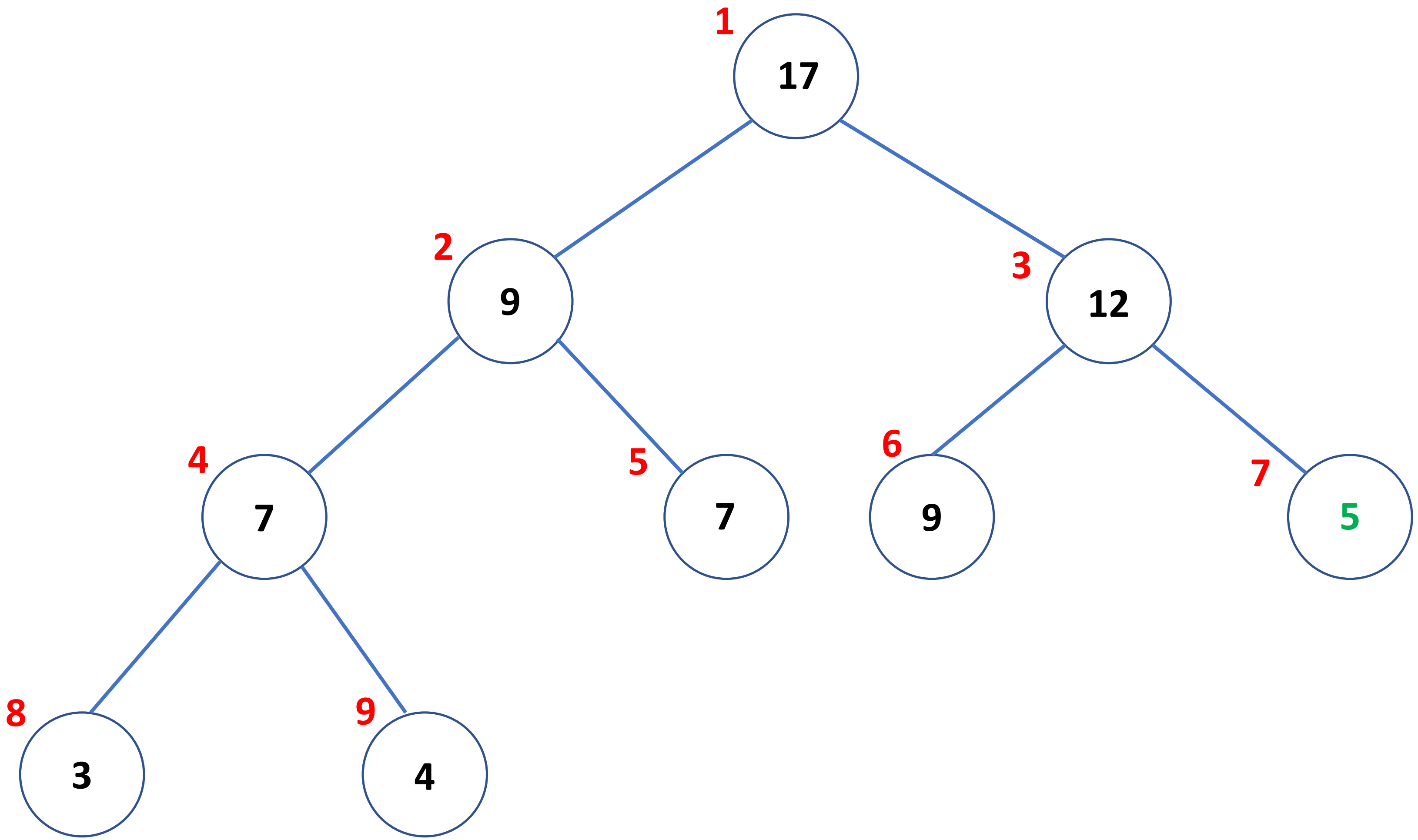
Right Child of A[i] is at A[2i+1]



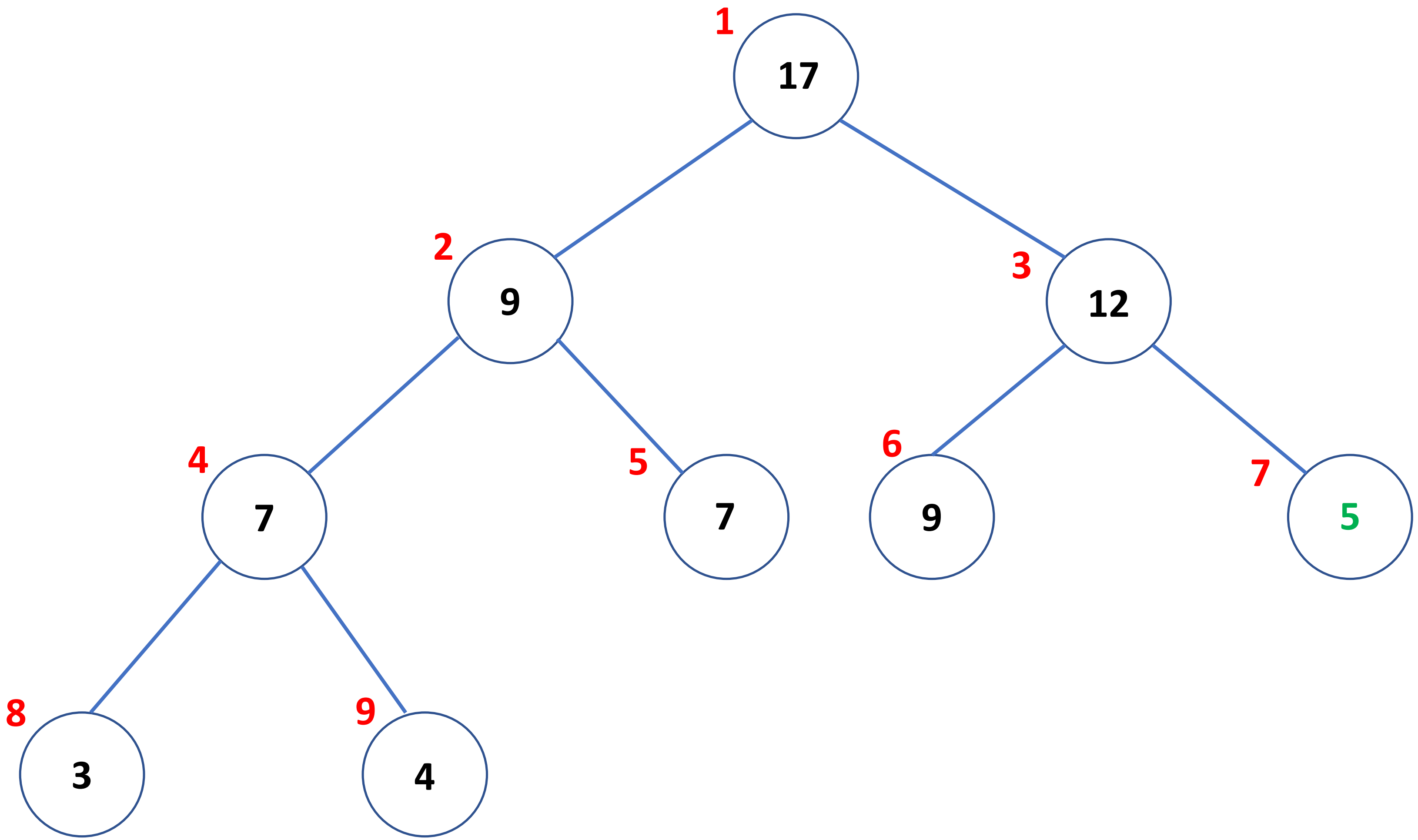


A:	17	9	12	7	7	9	5	3	4
Index	1	2	3	4	5	6	7	8	9

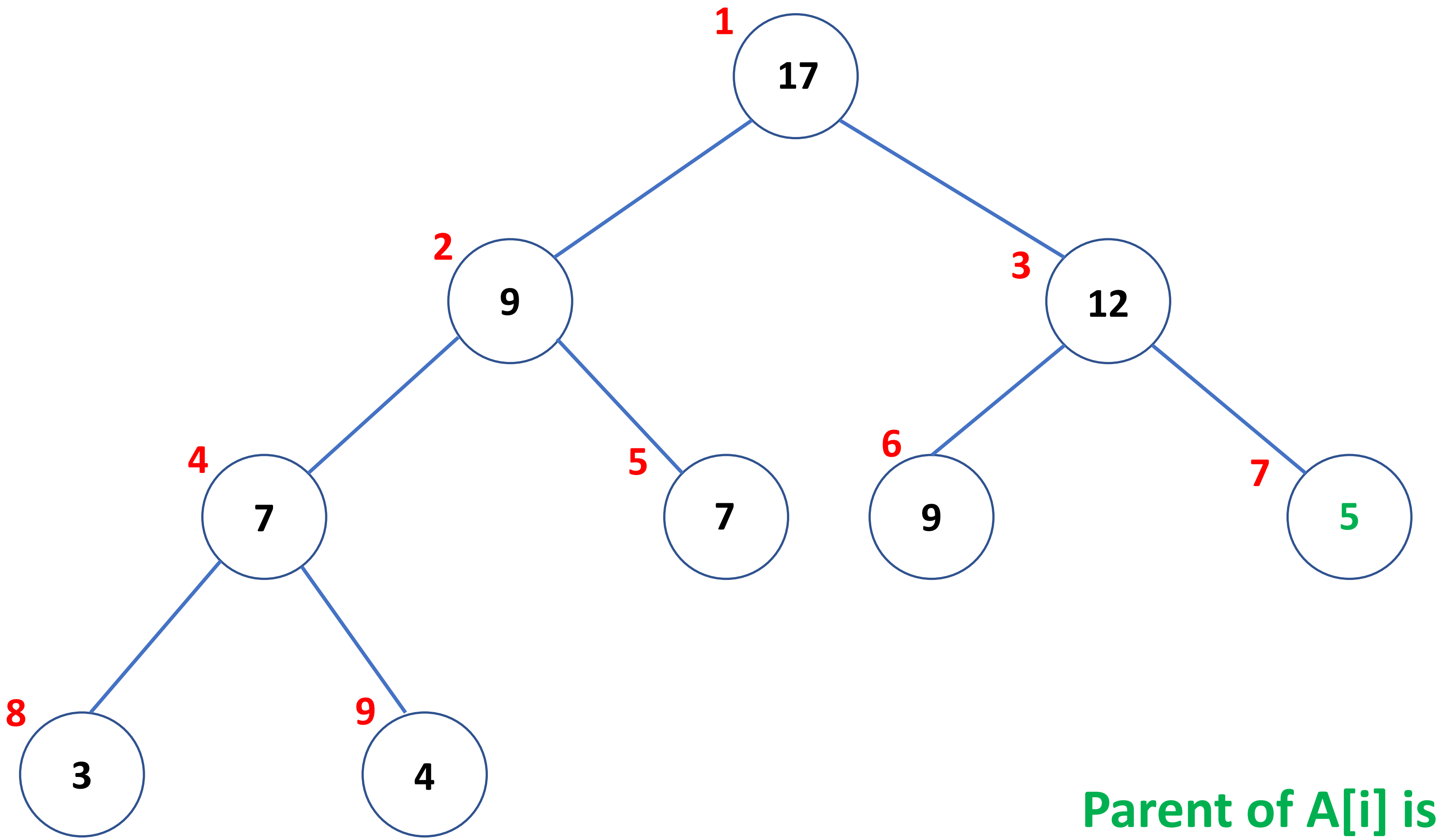

1
17
2
9
3
12
4
7
5
7
6
9
7
5
8
3
9
4
A:
17
9
12
7
7
9
5
3
4
Index
1
2
3
4
5
6
7
8
9



A:	17	9	12	7	7	9	5	3	4
Index	1	2	3	4	5	6	7	8	9



A:	17	9	12	7	7	9	5	3	4
Index	1	2	3	4	5	6	7	8	9



Parent of A[i] is A[$\lfloor i/2 \rfloor$]

A:	17	9	12	7	7	9	5	3	4
Index	1	2	3	4	5	6	7	8	9

Efficient navigation in the tree

- Left Child of A[i] is at A[2i]
- Right Child of A[i] is at A[2i+1]
- Parent of A[i] is A[$\lfloor i/2 \rfloor$]

Heap Operations

We will implement the following operations:

- **Insert(A, x)**
- **Max(A)**
- **Extract_Max(A)**

We will implement the following operations:

- **Insert(A, x)**
- **Max(A)**
- **Extract_Max(A)**

<u>High-Level idea for operations:</u>

We will implement the following operations:

- **Insert(A, x)**
- **Max(A)**
- **Extract_Max(A)**

<u>High-Level idea for operations:</u>

1) Maintain the CBT shape

We will implement the following operations:

- **Insert(A, x)**
- **Max(A)**
- **Extract_Max(A)**

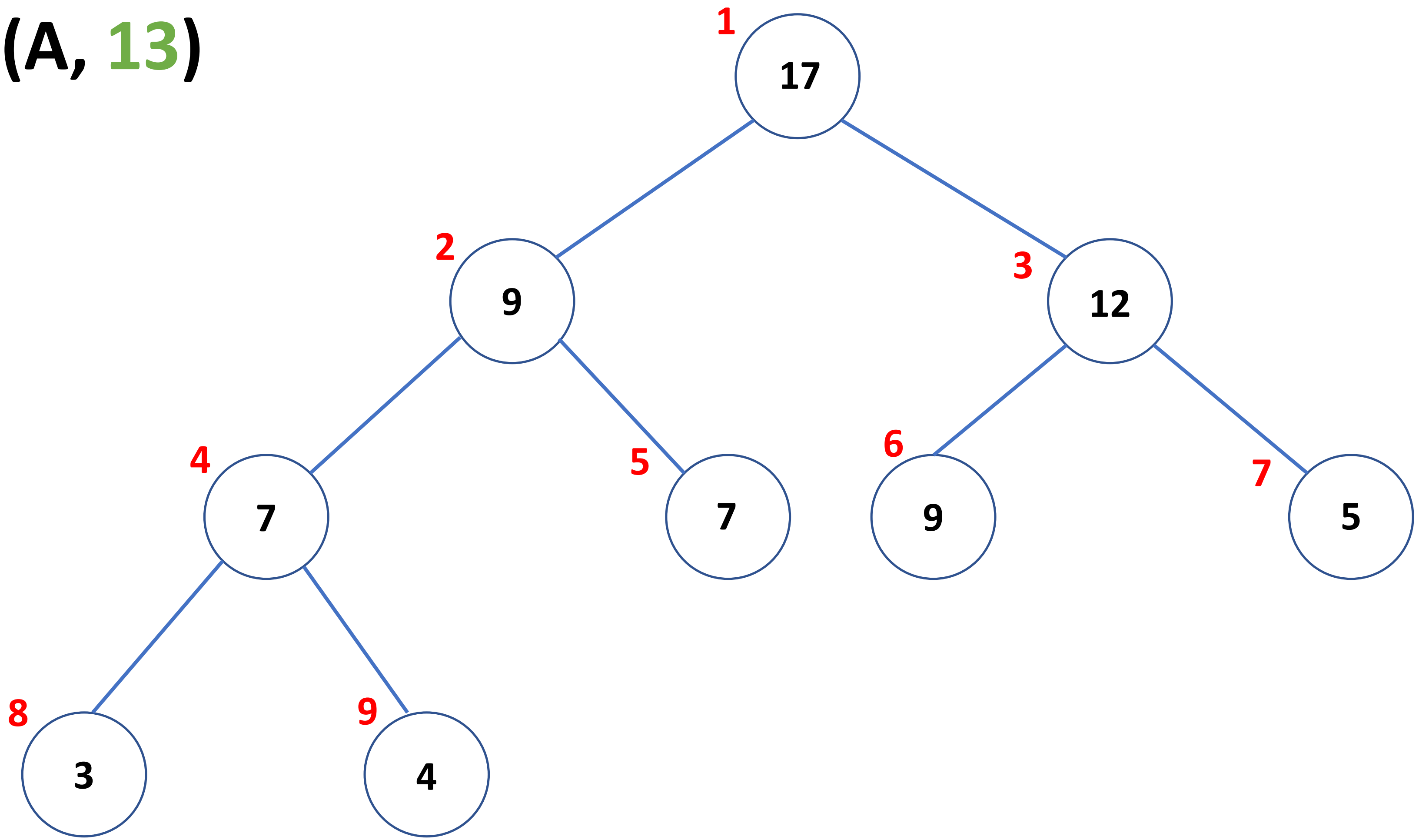
<u>High-Level idea for operations:</u>

1) Maintain the CBT shape

2) Maintain the Max-Heap property

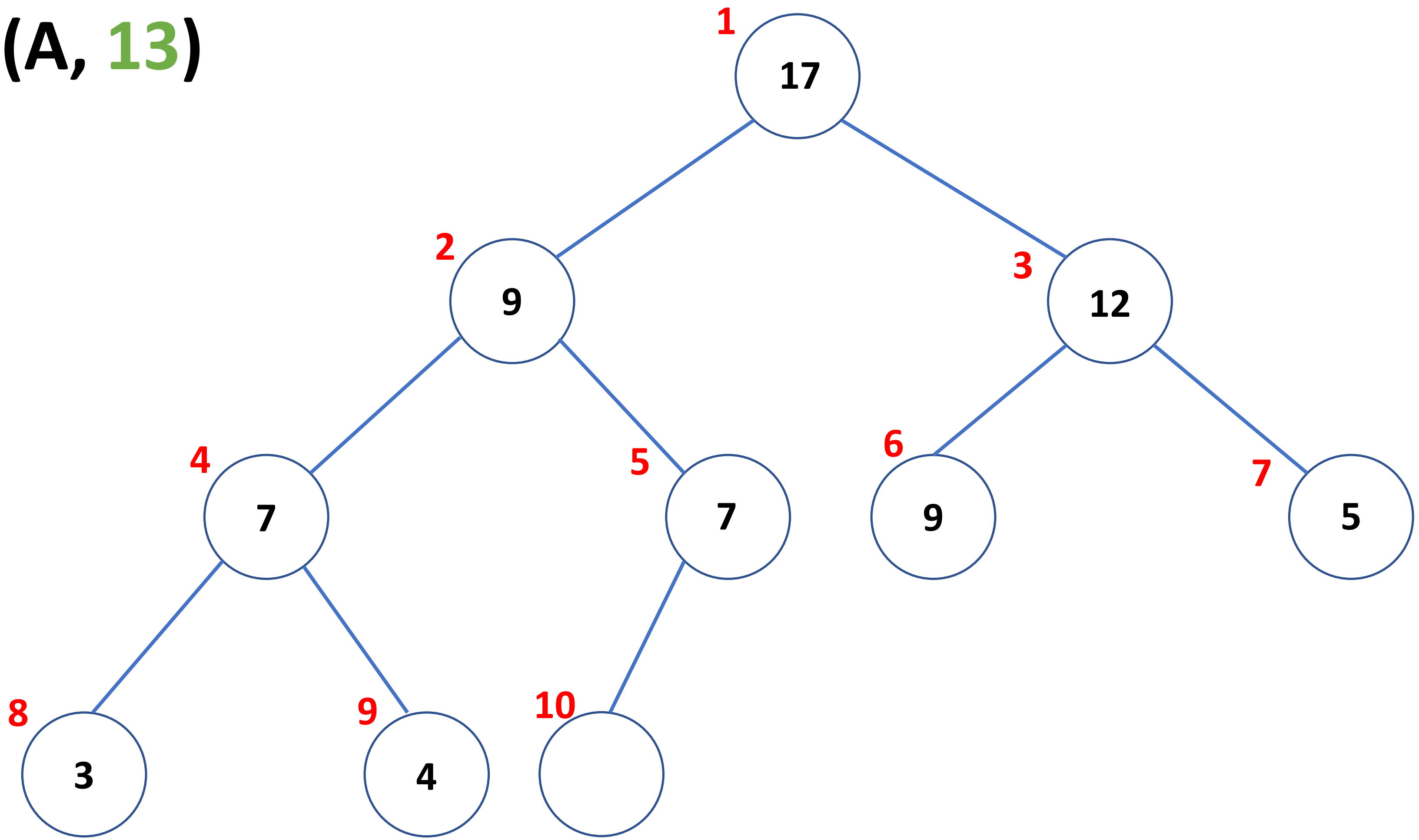
Insert(A, x)

Insert(A, 13)



A:	17	9	12	7	7	9	5	3	4	A.Heapsize = 9
Index	1	2	3	4	5	6	7	8	9	

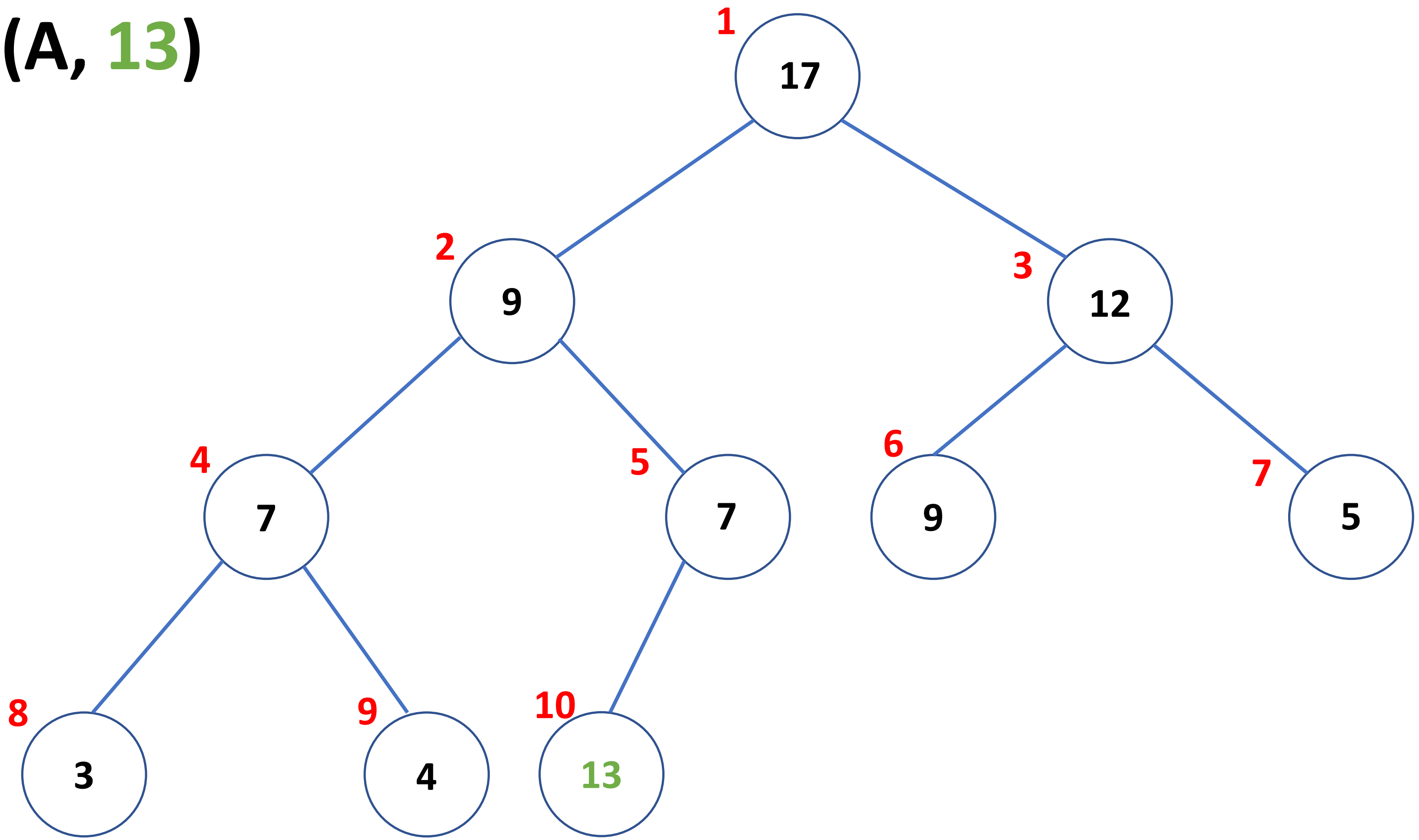
Insert(A, 13)



A:	17	9	12	7	7	9	5	3	4	
Index	1	2	3	4	5	6	7	8	9	10

A.Heapsize = 10

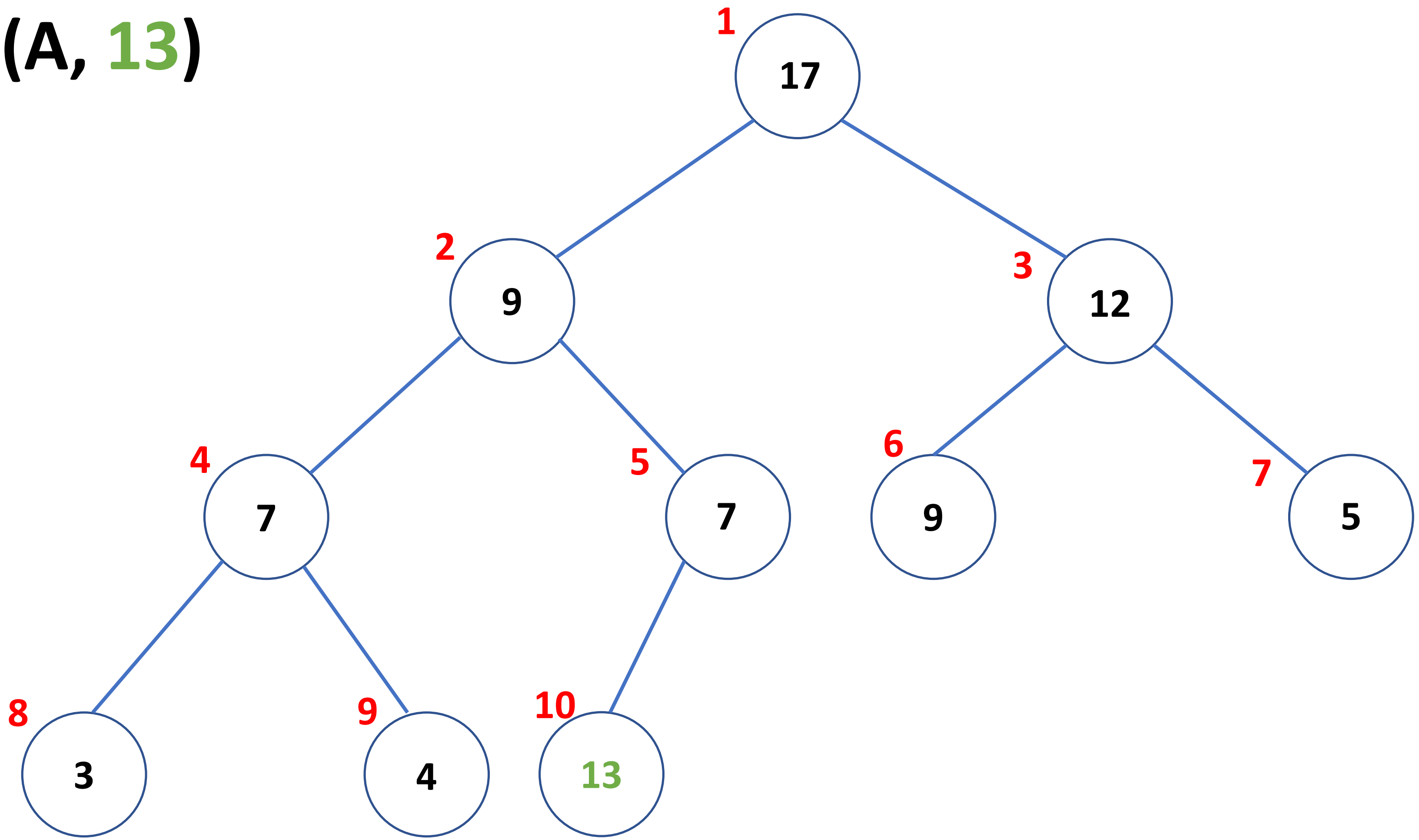
Insert(A, 13)



A:	17	9	12	7	7	9	5	3	4	13
Index	1	2	3	4	5	6	7	8	9	10

A.Heapsize = 10

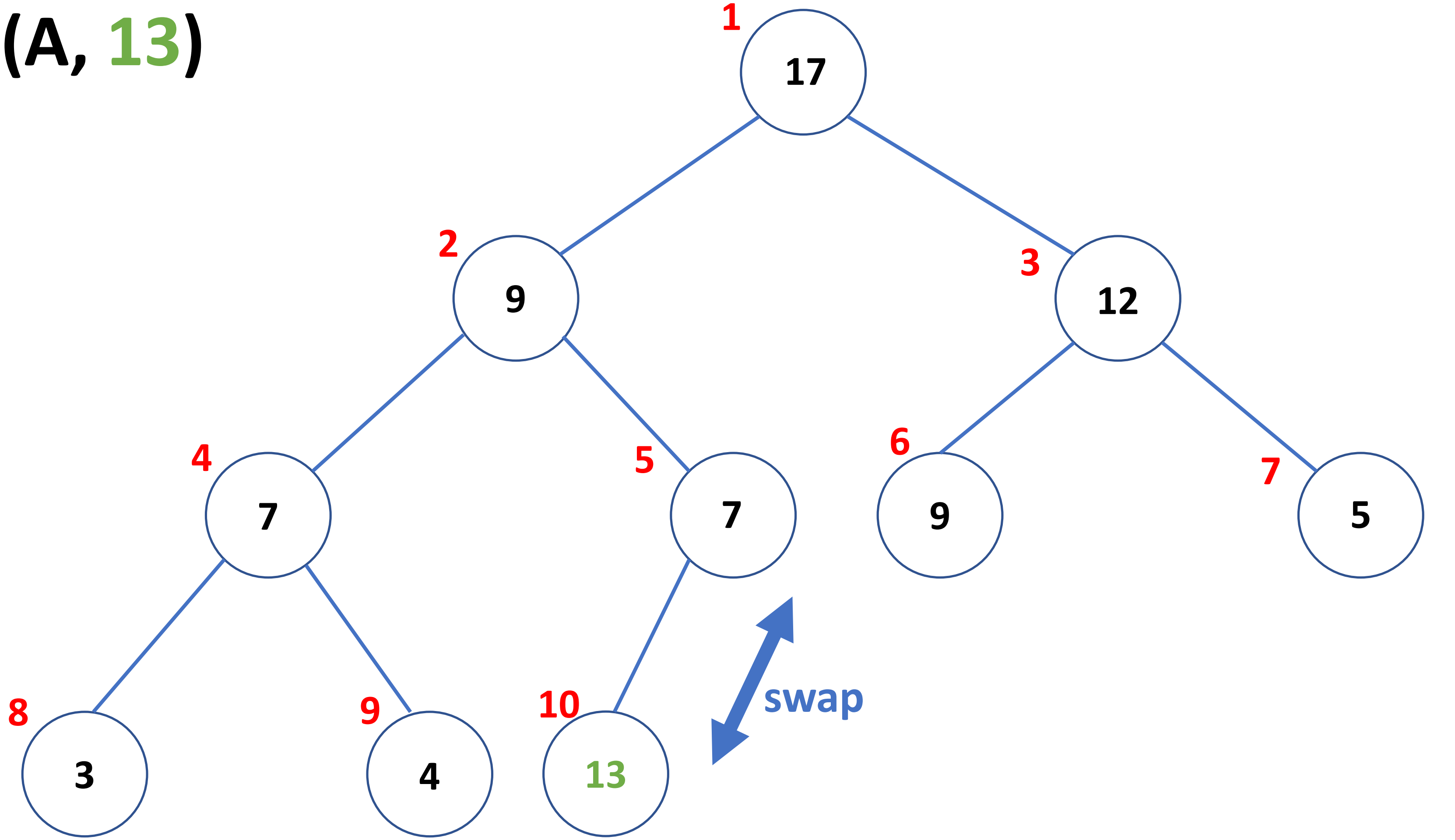
Insert(A, 13)



This is a CBT, but Max-Heap Property has been violated.

A:	17	9	12	7	7	9	5	3	4	13	A.Heapsize = 10
Index	1	2	3	4	5	6	7	8	9	10	

Insert(A, 13)

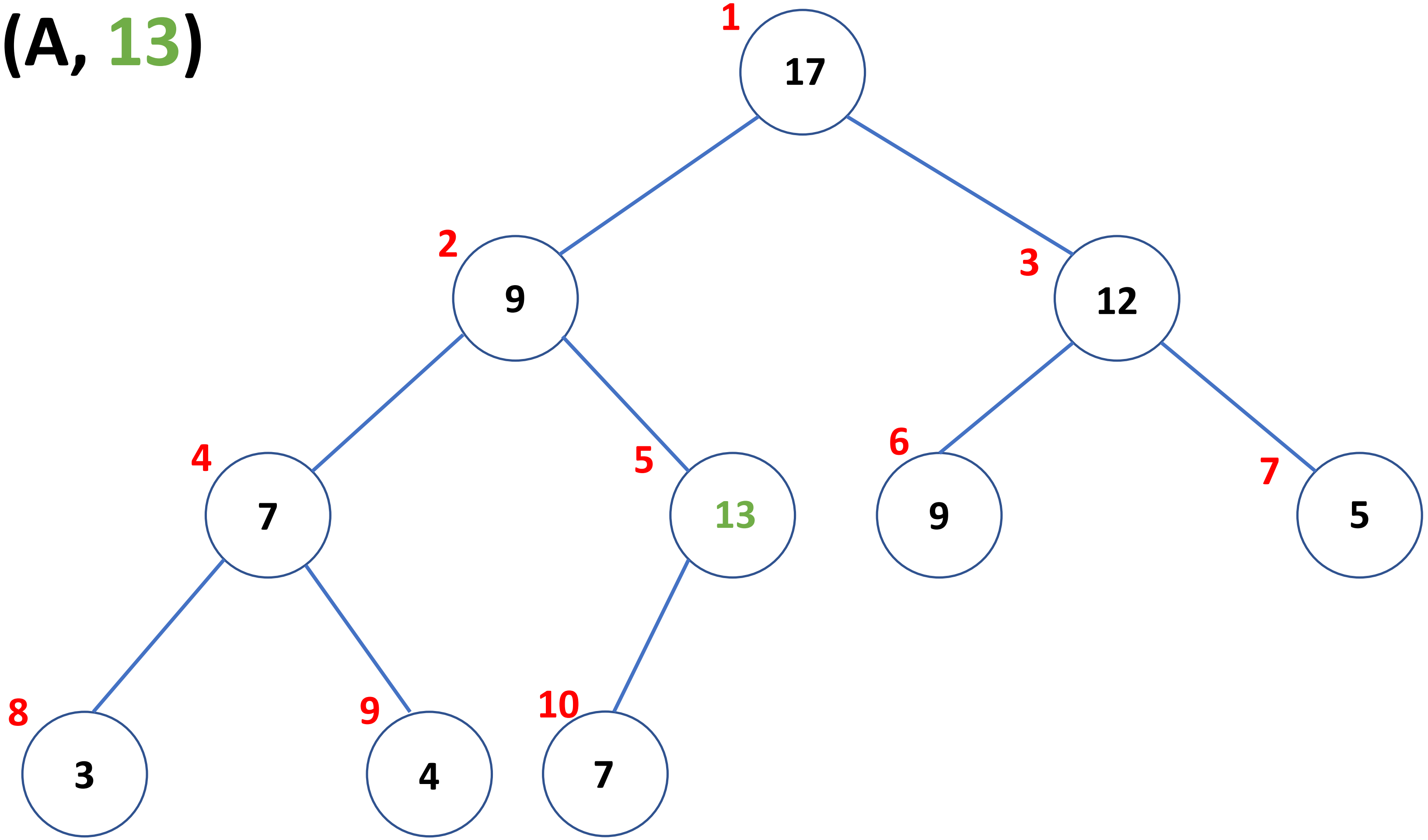


swap

A:	17	9	12	7	7	9	5	3	4	13
Index	1	2	3	4	5	6	7	8	9	10

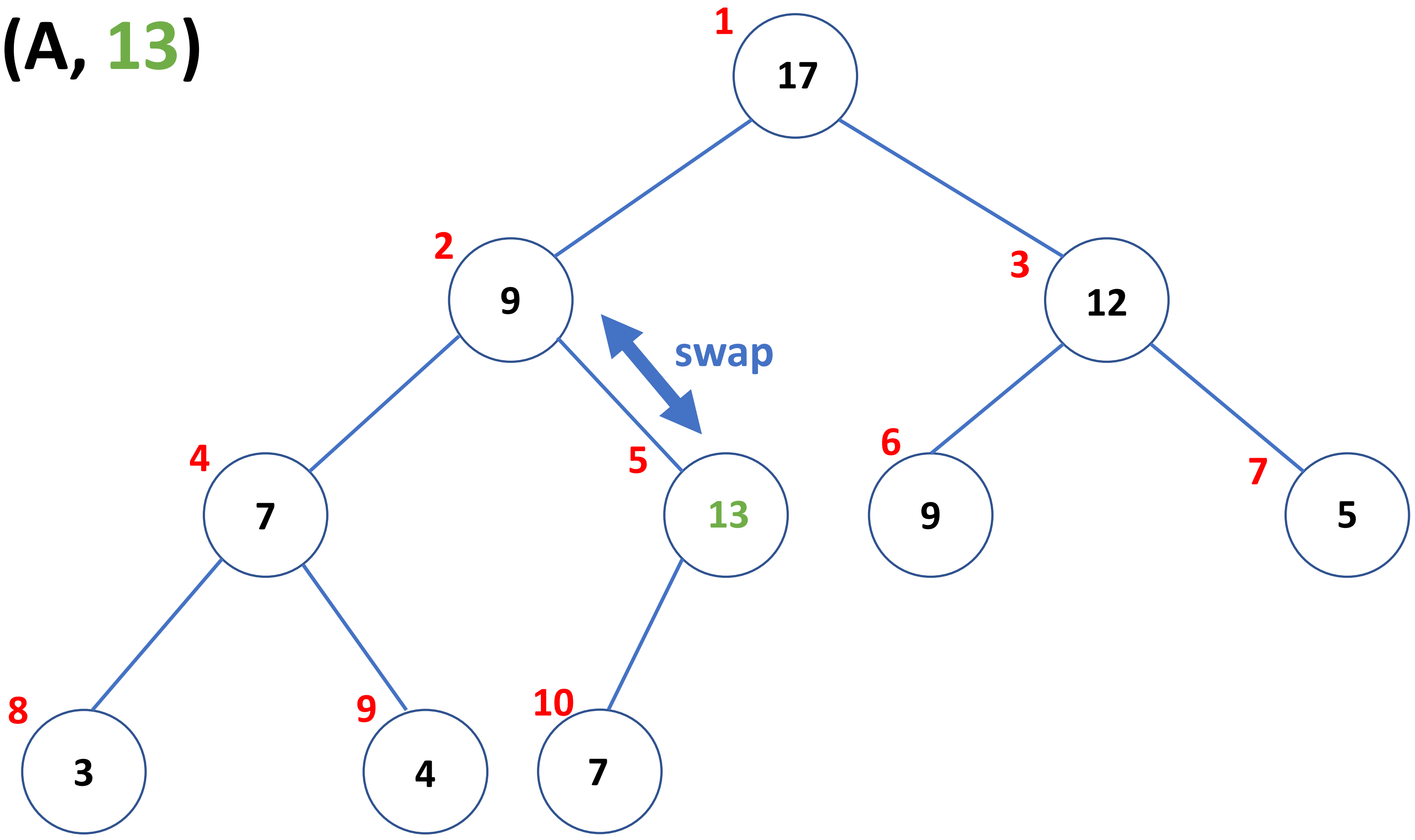
A.Heapsize = 10

Insert(A, 13)



A:	17	9	12	7	13	9	5	3	4	7	A.Heapsize = 10
Index	1	2	3	4	5	6	7	8	9	10	

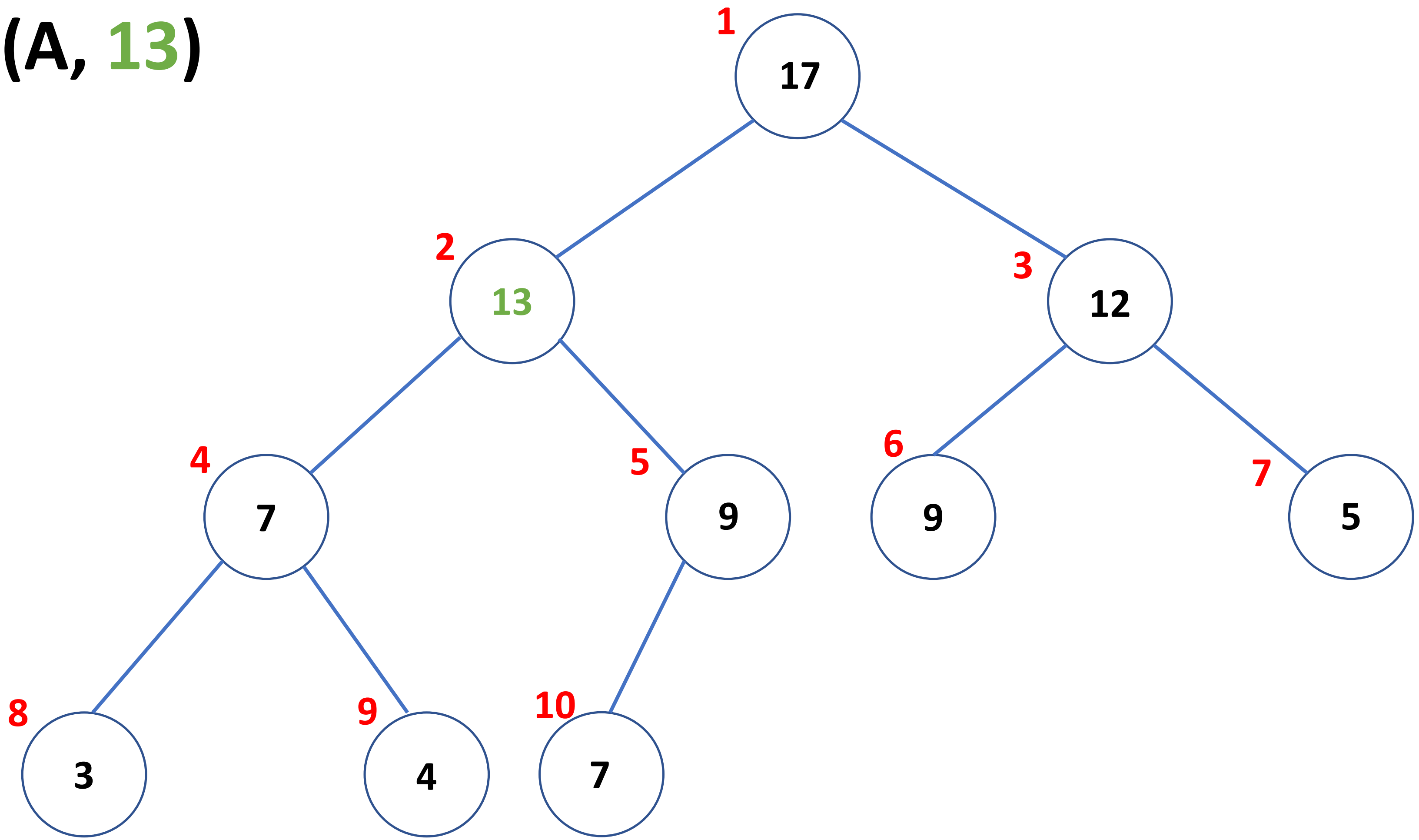
Insert(A, 13)



swap

A:	17	9	12	7	13	9	5	3	4	7	A.Heapsize = 10
Index	1	2	3	4	5	6	7	8	9	10	

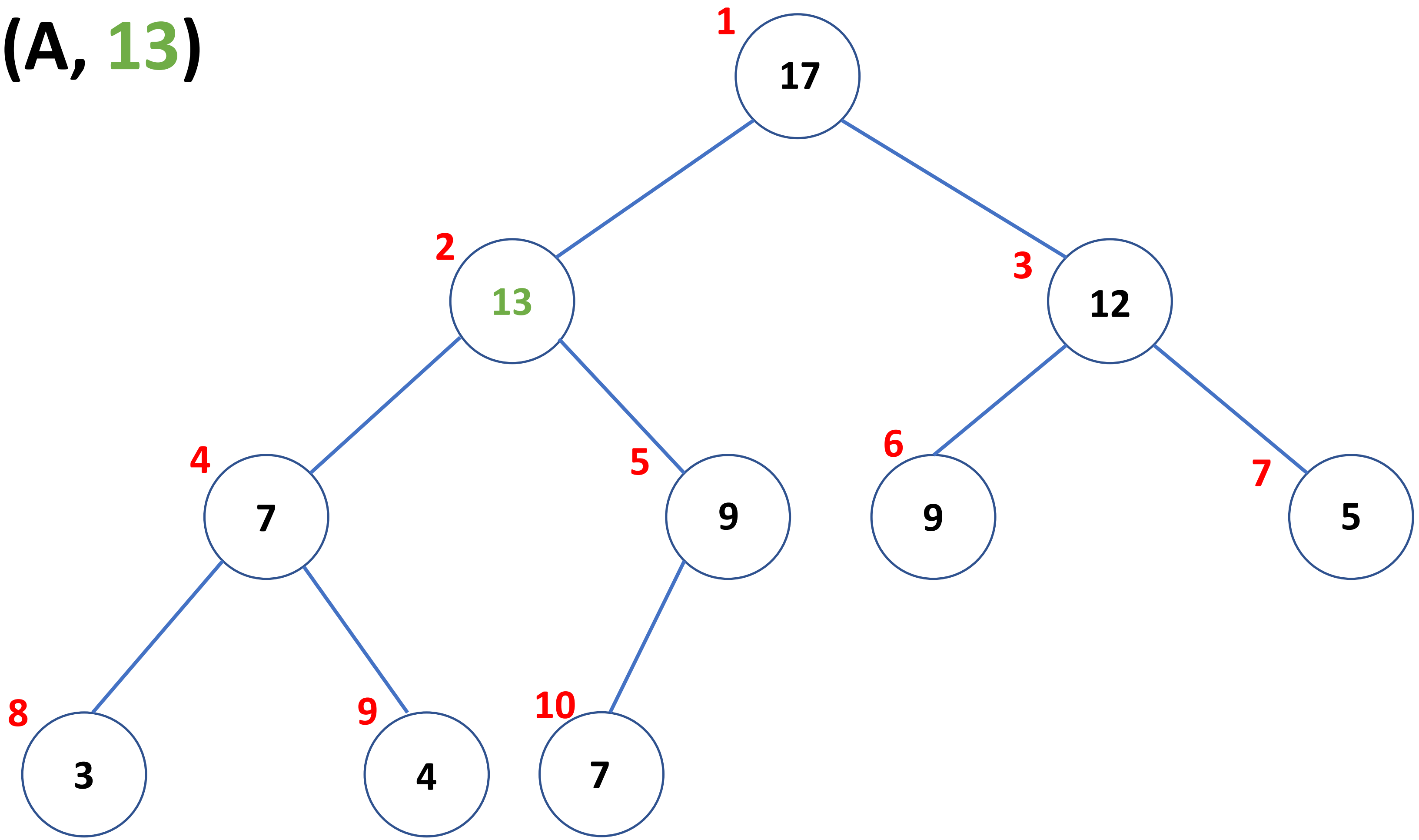
Insert(A, 13)



A:	17	13	12	7	9	9	5	3	4	7
Index	1	2	3	4	5	6	7	8	9	10

A.Heapsize = 10

Insert(A, 13)



Max-Heap Property has been restored!

A:										
	17	13	12	7	9	9	5	3	4	7
Index	1	2	3	4	5	6	7	8	9	10

A.Heapsize = 10

Insert(A, x)

1. Put x at the bottom left of the tree:

   Increment A.heapsize and set A[A.heapsize] = x

2. Percolate x up the tree:

   While priority of x > priority of its parent

   Swap x with parent

Insert(A, x)

1. Put x at the bottom left of the tree:

    Increment A.heapsize and set A[A.heapsize] = x

2. Percolate x up the tree:

    While x is not root AND priority of x > priority of its parent

        Swap x with parent

Insert(A, x)

1. Put x at the bottom left of the tree:

   Increment A.heapsize and set A[A.heapsize] = x

2. Percolate x up the tree:

   While x is not root AND priority of x > priority of its parent

   Swap x with parent

Maintains CBT shape

Maintains Max-Heap property

What is the Worst-Case Complexity of **Insert(A,x)** ?

What is the Worst-Case Complexity of **Insert(A,x)** ?

O(log n)

For **every** input A,x of size n,
the algorithm takes
at most c_1. log n steps.

What is the Worst-Case Complexity of **Insert(A,x)** ?

$O(\log n)$

For **every** input A,x of size n,
the algorithm takes
at most $c_1 . \log n$ steps.

$\Omega(\log n)$

For **some** input A,x of size n,
the algorithm takes
at least $c_2 . \log n$ steps.

What is the Worst-Case Complexity of **Insert(A,x)** ?

$O(\log n)$

For **every** input A,x of size n,
the algorithm takes
at most c_1. log n steps.

$\Omega(\log n)$

For **some** input A,x of size n,
the algorithm takes
at least c_2. log n steps.

Priority of x is > Priority of root

What is the Worst-Case Complexity of **Insert(A,x)** ?

$\Theta(\log n)$ $\xrightarrow{\text{BOTH}}$ $O(\log n)$ and $\Omega(\log n)$

BOTH

$O(\log n)$

For **every** input A,x of size n,
the algorithm takes
at most $c_1 .\log n$ steps.

$\Omega(\log n)$

For **some** input A,x of size n,
the algorithm takes
at least $c_2 .\log n$ steps.

Max(A)

Max(A):

Return A[1]

Max(A):

Return A[1]

Root of the CBT

Max(A):

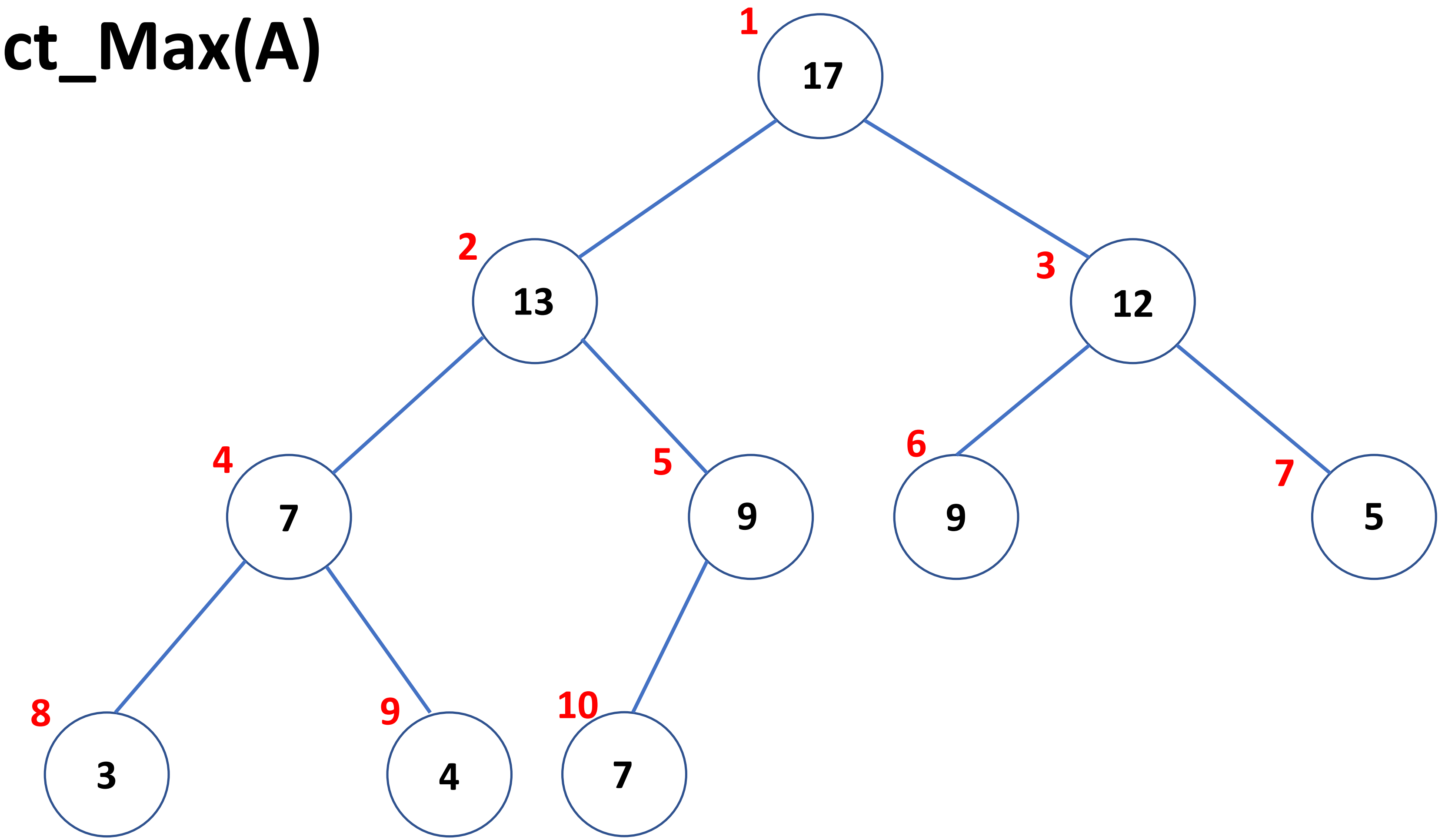
Return A[1]

Root of the CBT

Worst-Case Time Complexity is $\Theta(1)$

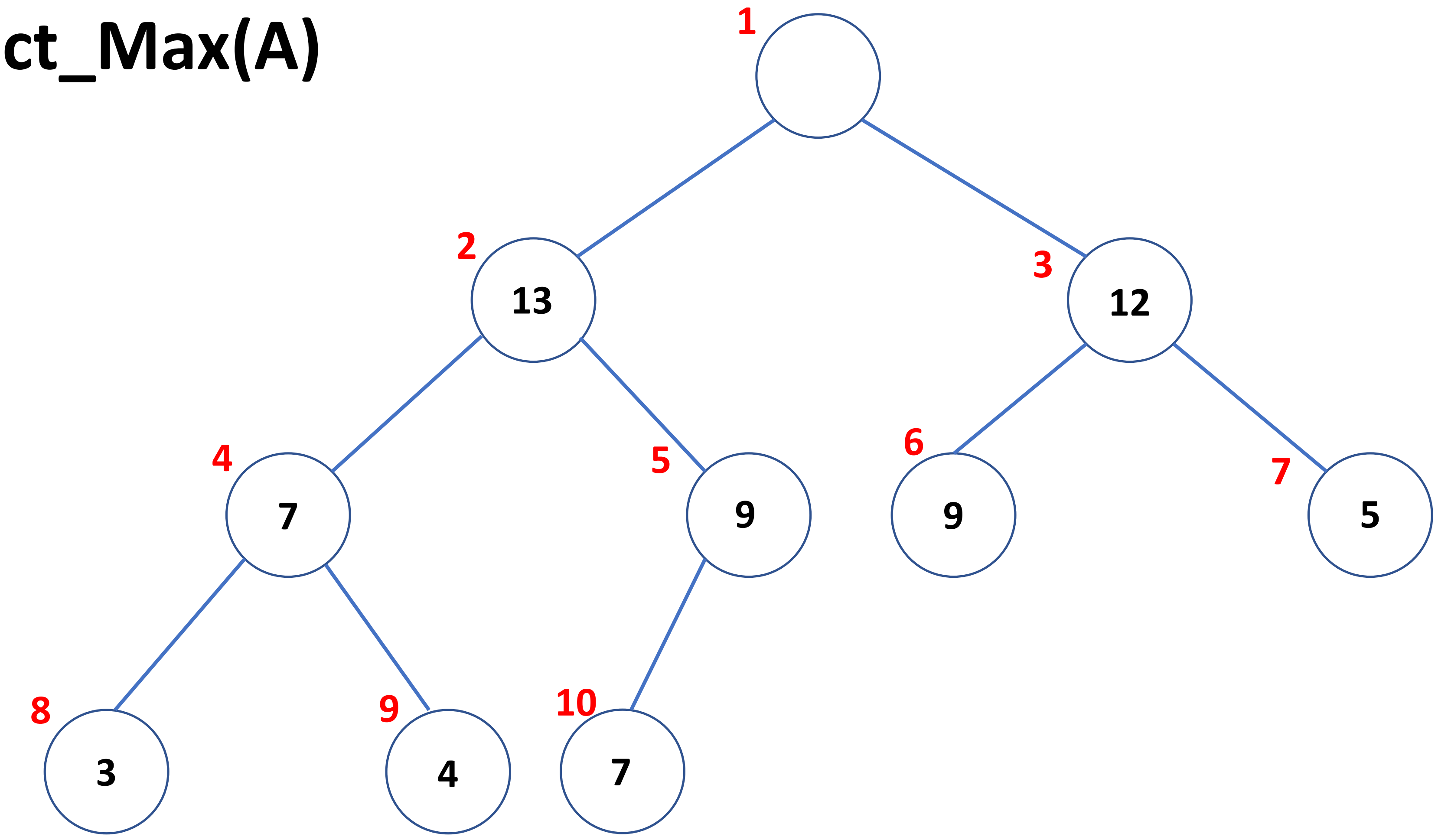
Extract_Max(A)

Extract_Max(A)



A:	17	13	12	7	9	9	5	3	4	7	A.Heapsize = 10
Index	1	2	3	4	5	6	7	8	9	10	

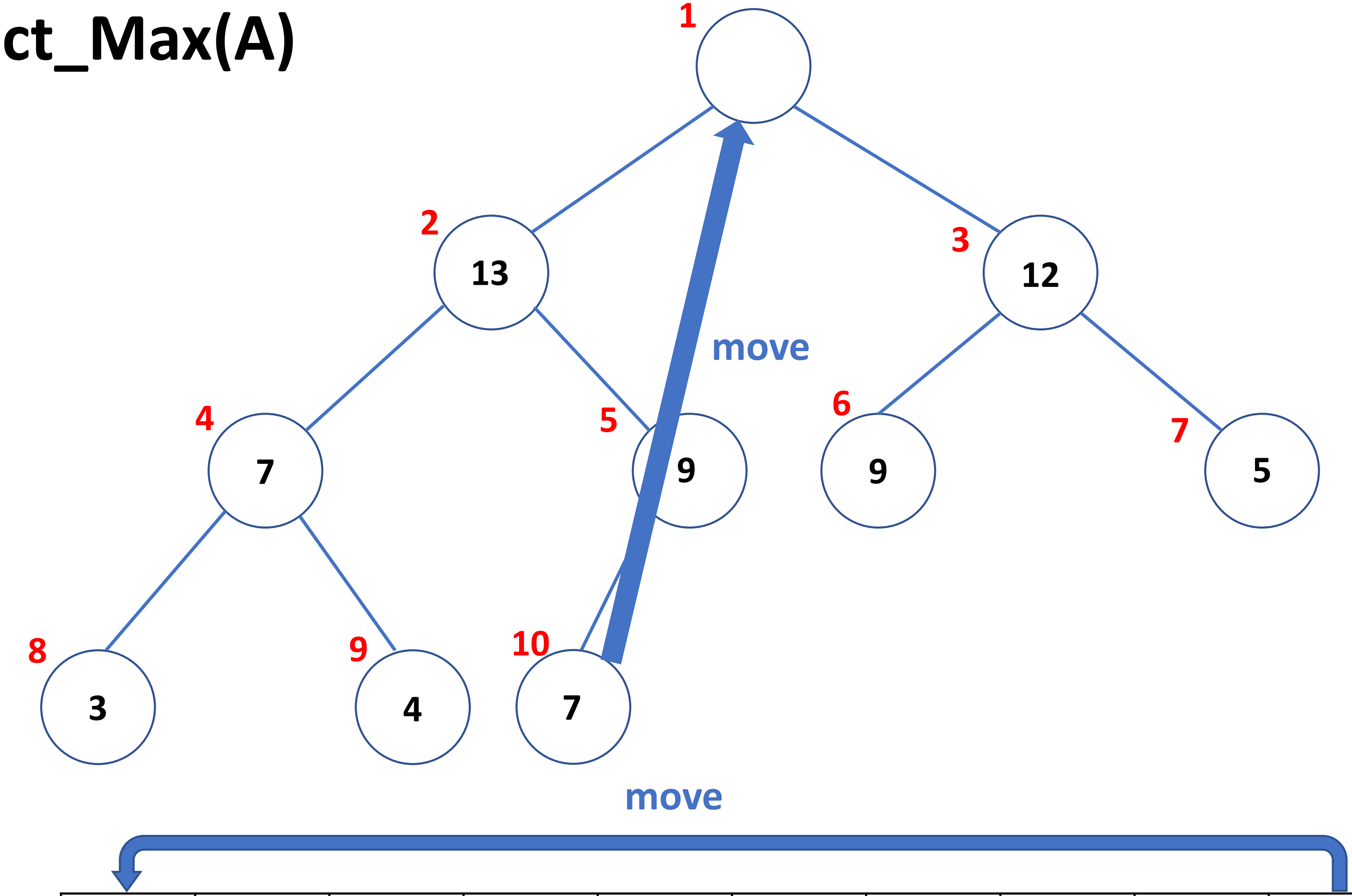
Extract_Max(A)



A:		13	12	7	9	9	5	3	4	7
Index	1	2	3	4	5	6	7	8	9	10

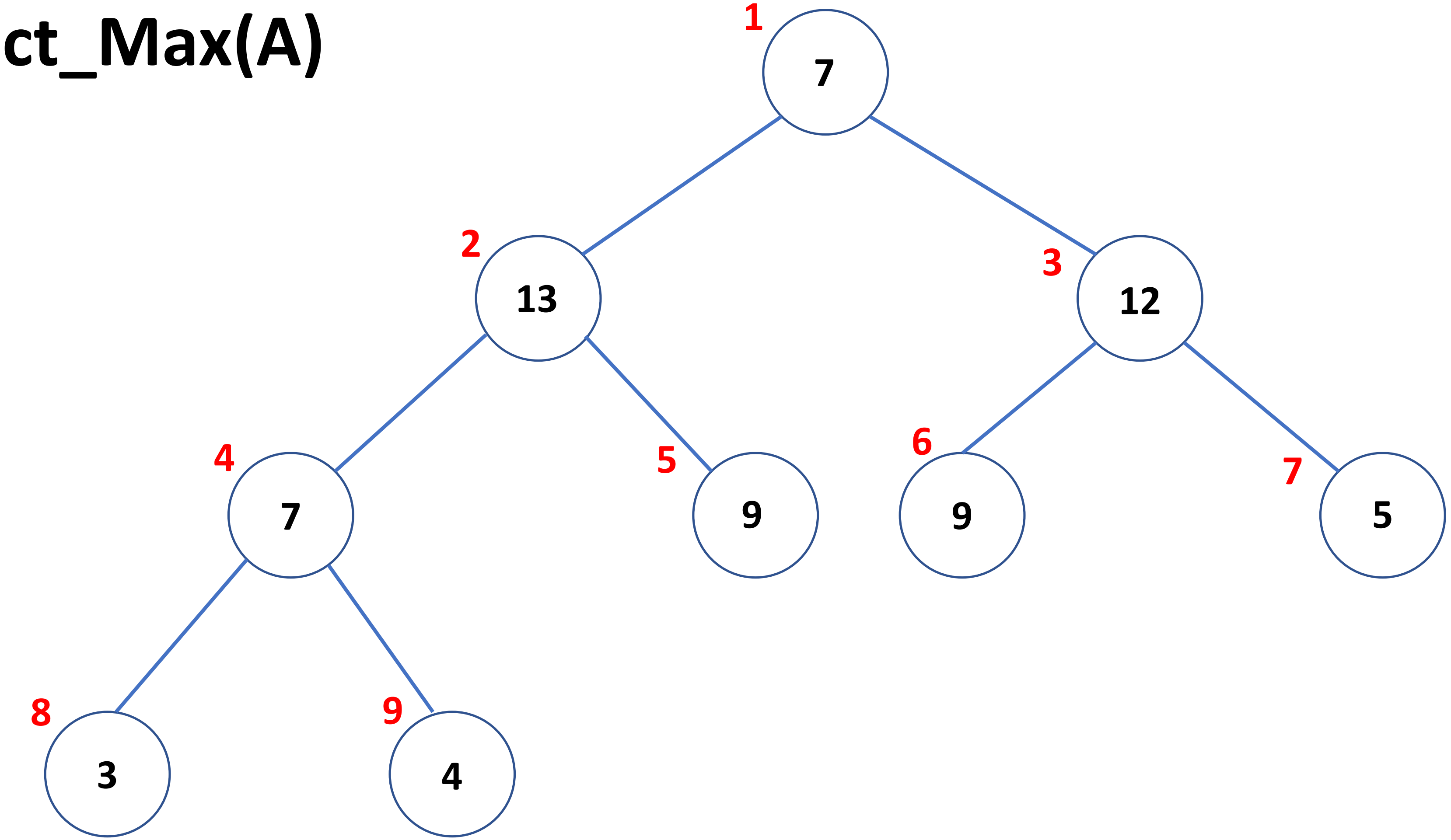
A.Heapsize = 10

Extract_Max(A)



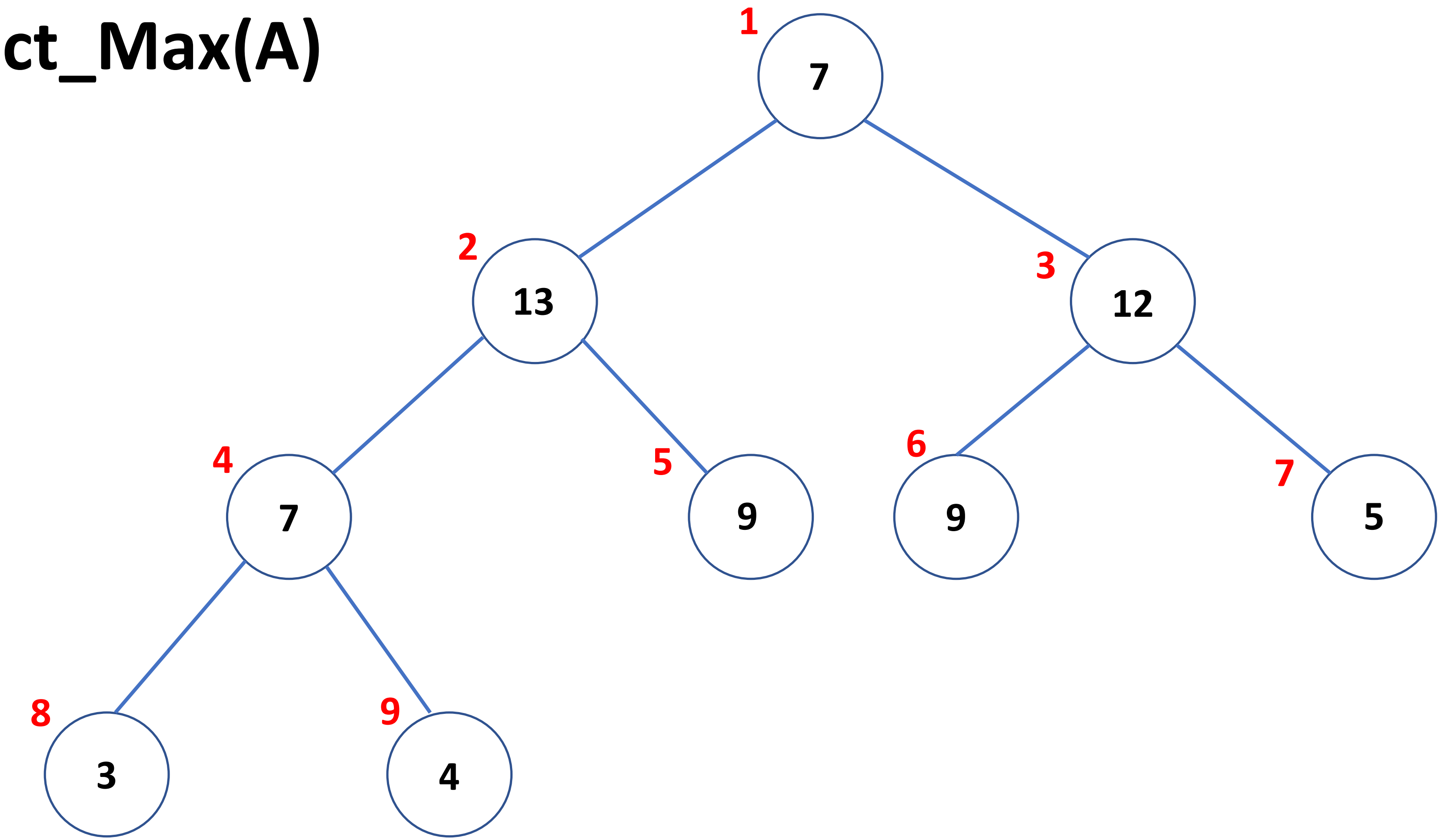
A:											
		13	12	7	9	9	5	3	4	7	A.Heapsize = 10
Index	1	2	3	4	5	6	7	8	9	10	

Extract_Max(A)



A:	7	13	12	7	9	9	5	3	4	A.Heapsize = 9
Index	1	2	3	4	5	6	7	8	9	

Extract_Max(A)

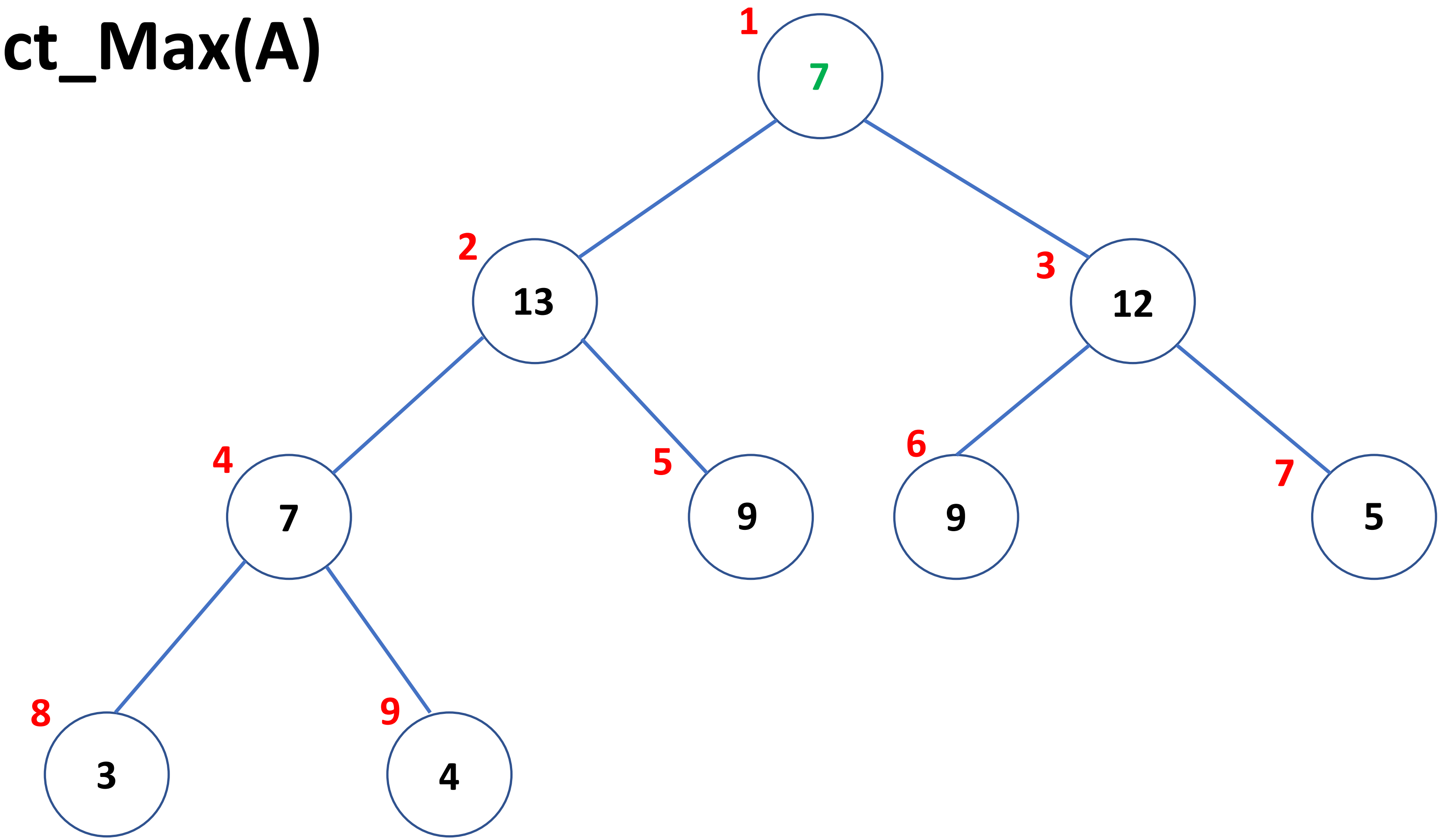


This is a CBT, but Max-Heap Property has been violated.

A:									
	7	13	12	7	9	9	5	3	4
Index	1	2	3	4	5	6	7	8	9

A.Heapsize = 9

Extract_Max(A)

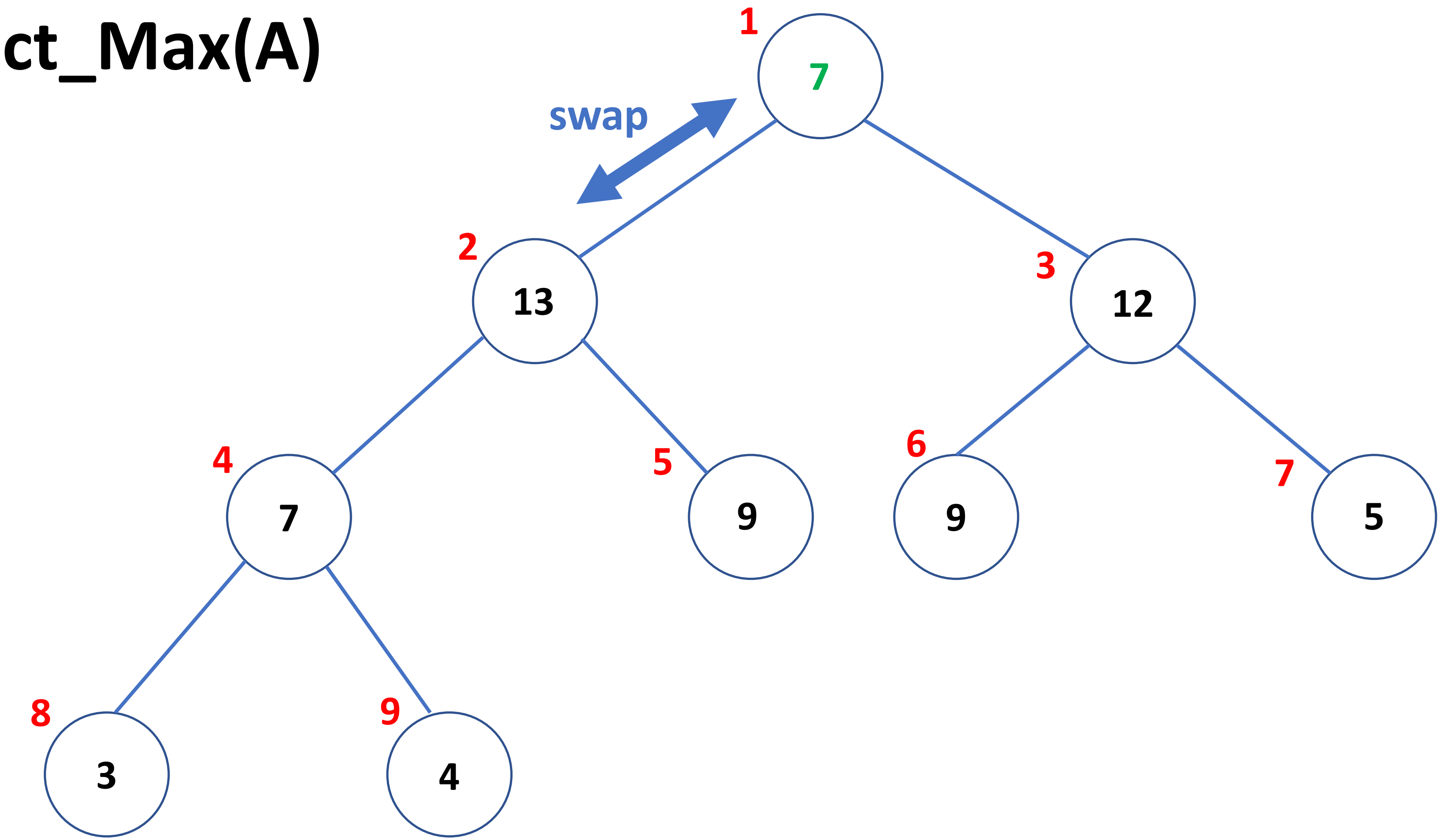


This is a CBT, but Max-Heap Property has been violated.

A:	7	13	12	7	9	9	5	3	4
Index	1	2	3	4	5	6	7	8	9

A.Heapsize = 9

Extract_Max(A)

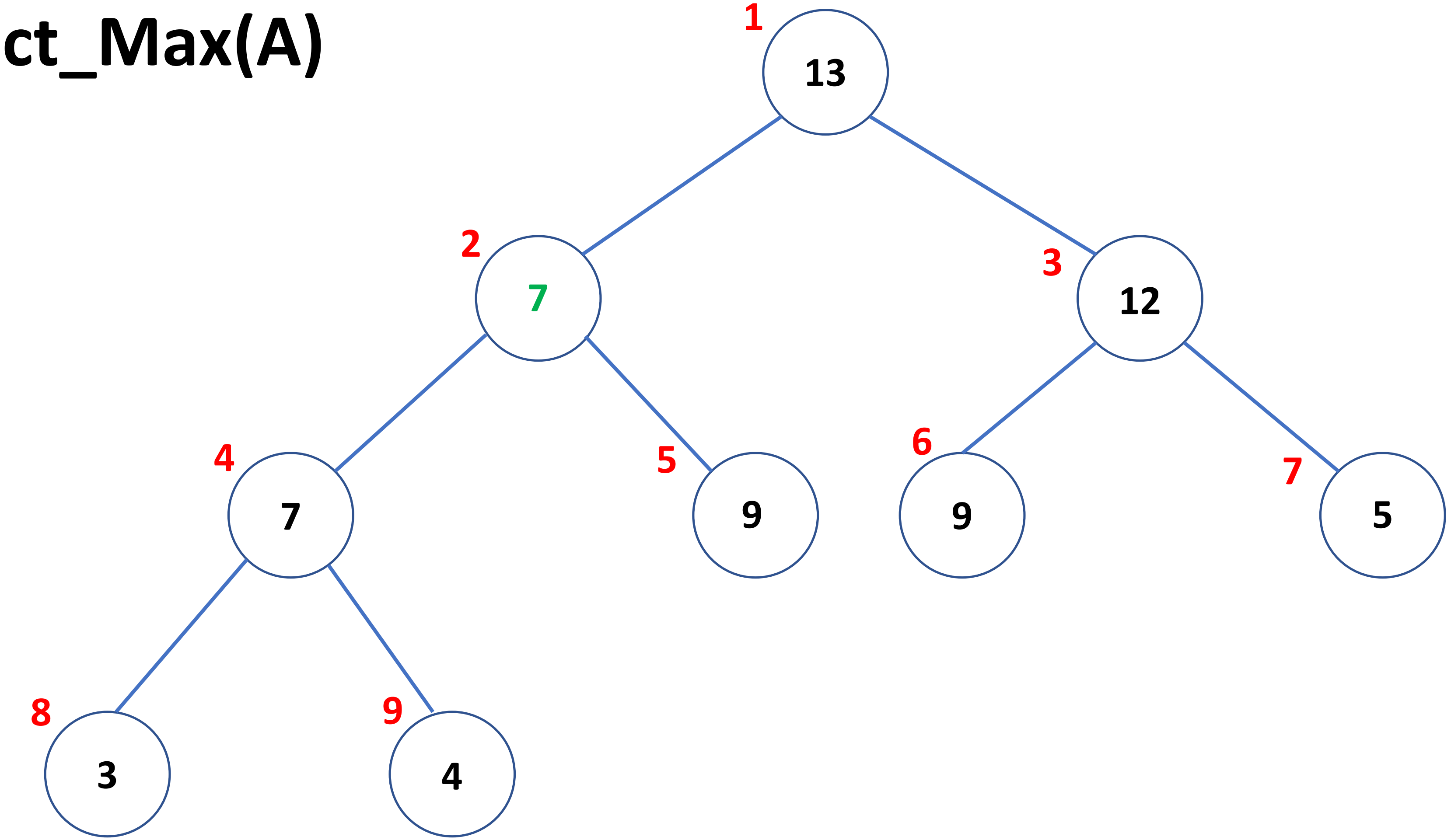


swap

A:	7	13	12	7	9	9	5	3	4
Index	1	2	3	4	5	6	7	8	9

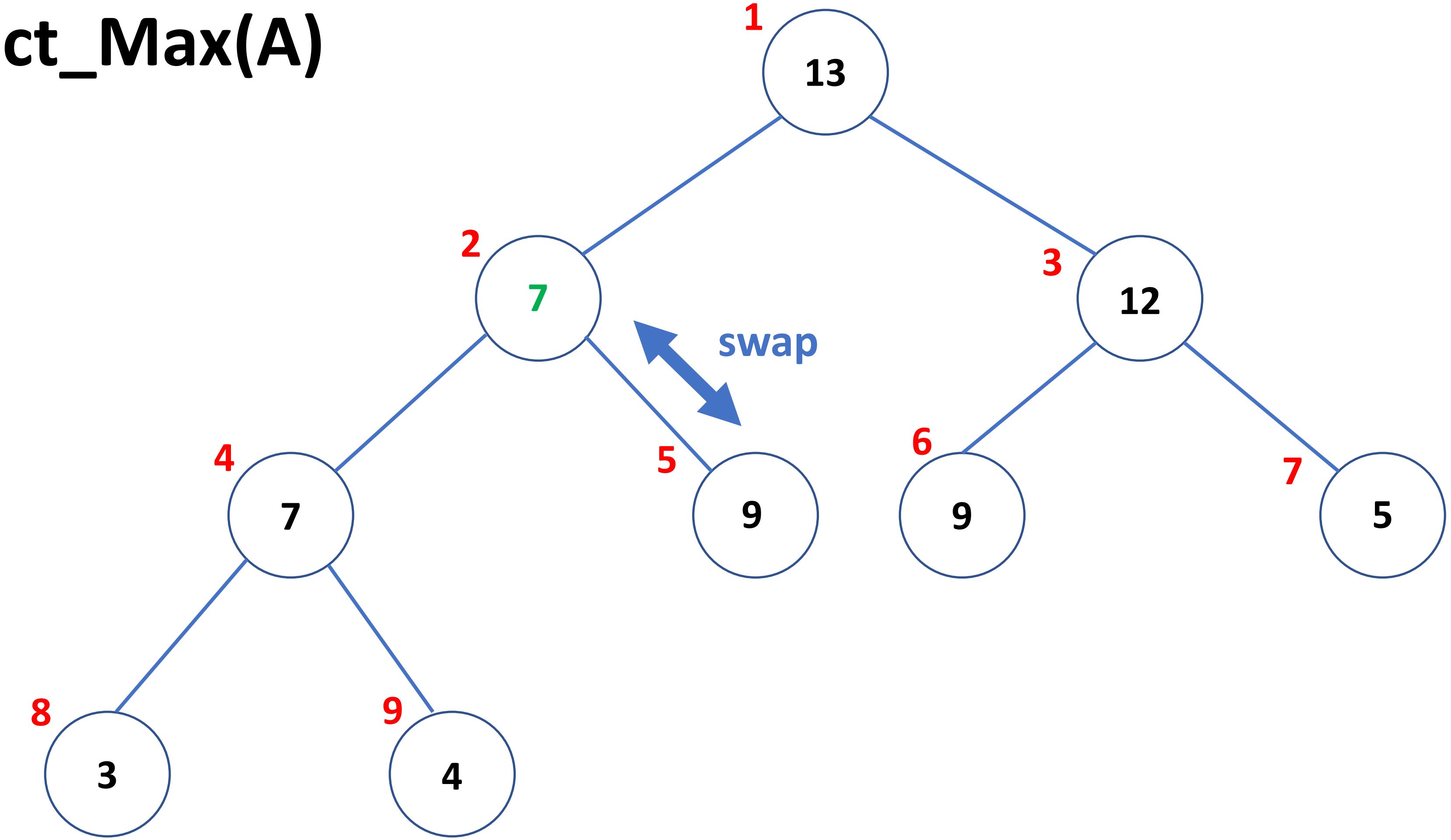
A.Heapsize = 9

Extract_Max(A)



A:										
A:	13	7	12	7	9	9	5	3	4	A.Heapsize = 9
Index	1	2	3	4	5	6	7	8	9	

Extract_Max(A)

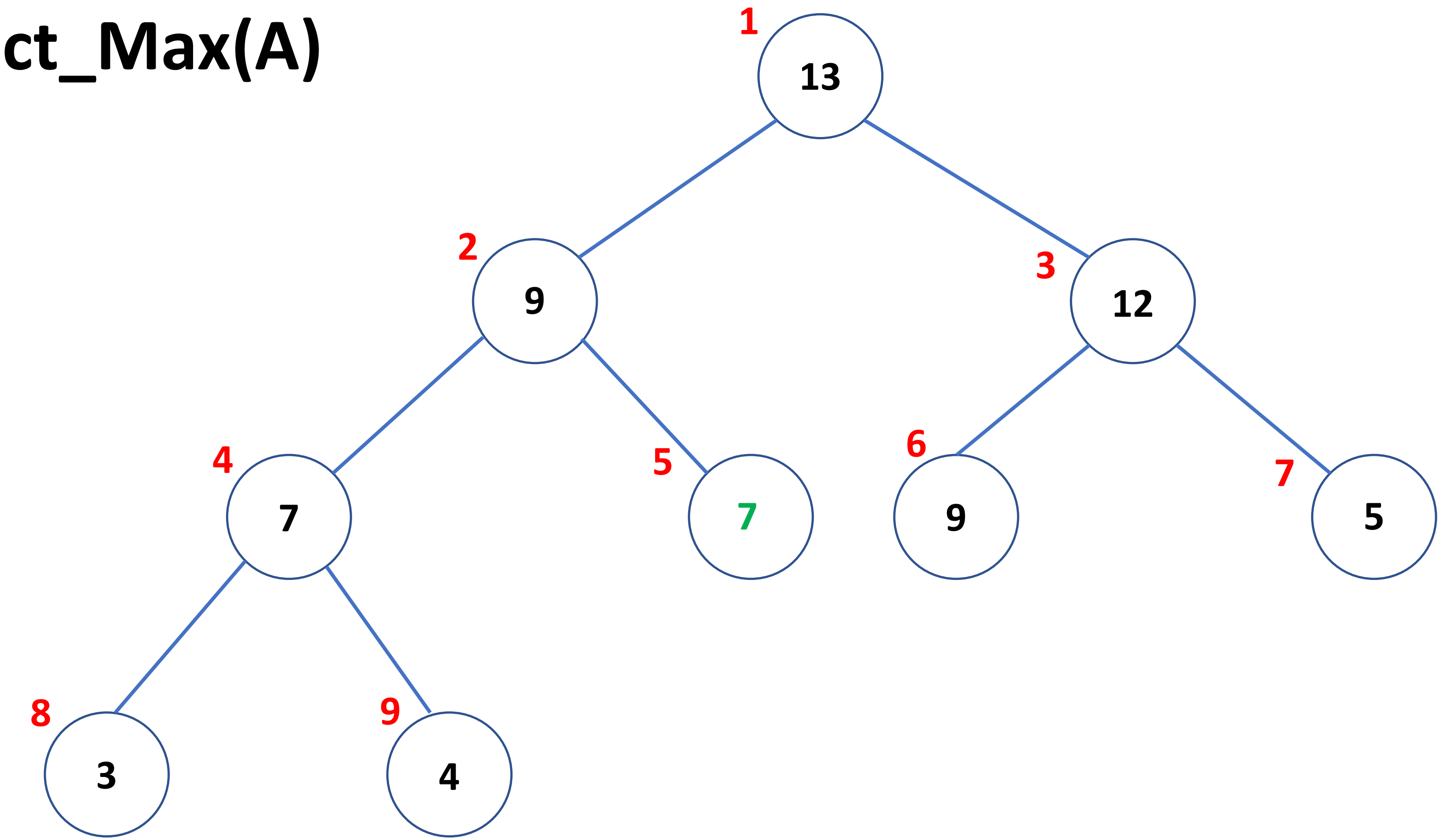


swap

A:	13	7	12	7	9	9	5	3	4
Index	1	2	3	4	5	6	7	8	9

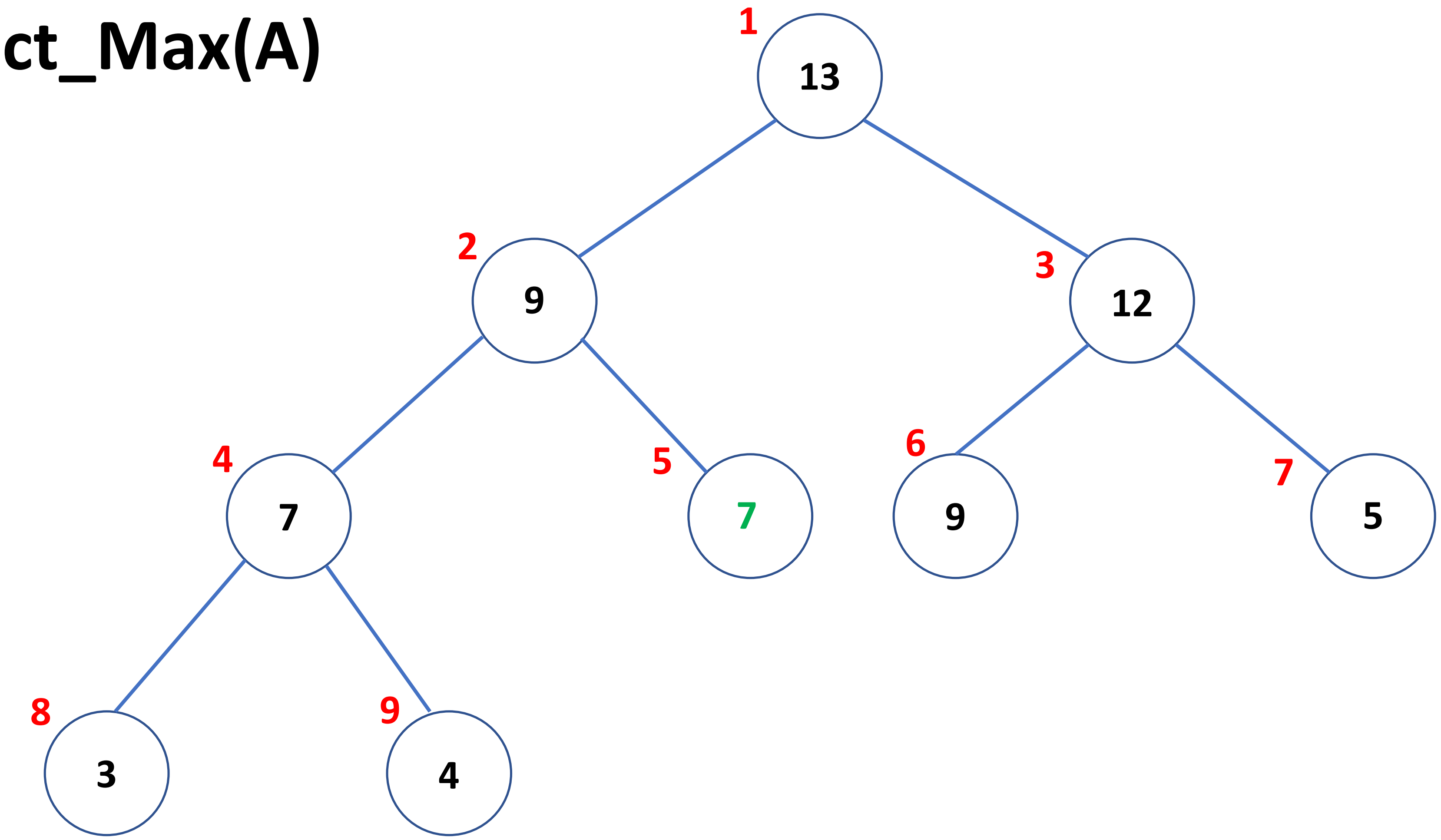
A.Heapsize = 9

Extract_Max(A)



A:	13	9	12	7	7	9	5	3	4	A.Heapsize = 9
Index	1	2	3	4	5	6	7	8	9	

Extract_Max(A)



Max-Heap Property has been restored!

A:

13	9	12	7	7	9	5	3	4
1	2	3	4	5	6	7	8	9

A.Heapsize = 9

Index: 1 2 3 4 5 6 7 8 9

Extract_Max(A)

1. Return the root A[1].

2. Remove the returned element from the heap:

    Set A[1] = A[A.heapsize] and decrement A.heapsize

3. Drip the element in A[1] down the tree:

    Let x be the element in A[1]

    While priority of some child of x > priority of x

        Swap x with the highest-priority child of x

Extract_Max(A)

1. Return the root A[1].

2. Remove the returned element from the heap:

   Set A[1] = A[A.heapsize] and decrement A.heapsize

3. Drip the element in A[1] down the tree:

   Let x be the element in A[1]

   While x is not a leaf AND priority of some child of x > priority of x

   Swap x with the highest-priority child of x

Extract_Max(A)

1. Return the root A[1].

2. Remove the returned element from the heap:

   Set A[1] = A[A.heapsize] and decrement A.heapsize

3. Drip the element in A[1] down the tree:

   Let x be the element in A[1]

   While x is not a leaf AND priority of some child of x > priority of x

   Swap x with the highest-priority child of x

Index of x <= $\lfloor n/2 \rfloor$

Extract_Max(A)

1. Return the root A[1].

2. Remove the returned element from the heap:

   Set A[1] = A[A.heapsize] and decrement A.heapsize

Maintains CBT shape

3. Drip the element in A[1] down the tree:

   Let x be the element in A[1]

   While <u>x is not a leaf</u> AND priority of some child of x > priority of x

   Swap x with the highest-priority child of x

Maintains Max-Heap property

Index of x <= $\lfloor n/2 \rfloor$

What is the Worst-Case Complexity of **Extract_Max(A)** ?

What is the Worst-Case Complexity of **Extract_Max(A)** ?

$\Theta(\log n)$

BOTH

$O(\log n)$

For **every** input A of size n, the algorithm takes **at most** c_1. log n steps.

$\Omega(\log n)$

For **some** input A of size n, the algorithm takes **at least** c_2. log n steps.

What is the Worst-Case Complexity of **Extract_Max(A)** ?

$\Theta(\log n)$

BOTH

$O(\log n)$

For **every** input A of size n,
the algorithm takes
at most $c_1 . \log n$ steps.

$\Omega(\log n)$

For **some** input A of size n,
the algorithm takes
at least $c_2 . \log n$ steps.

A[A.heapsize] has the smallest priority

Application: HeapSort

To sort an Array A of n elements:

Application: HeapSort

To sort an Array A of n elements:

- Make a heap out of the elements of A

Application: HeapSort

To sort an Array A of n elements:

- Make a heap out of the elements of A

How do you do this?

Application: HeapSort

To sort an Array A of n elements:

- Make a heap out of the elements of A

This can be done in $\Theta(n)$ time !

Application: HeapSort

To sort an Array A of n elements:

- Make a heap out of the elements of A

  This can be done in $\Theta(n)$ time !

- **Extract_Max(A)** n times [Each one takes $\Theta(\log n)$ time]

Application: HeapSort

To sort an Array A of n elements:

- Make a heap out of the elements of A

This can be done in $\Theta(n)$ time !

- **Extract_Max(A)** n times [Each one takes $\Theta(\log n)$ time]

Worst-Case time complexity is $\Theta(n \log n)$

Application: HeapSort

To sort an Array A of n elements:

- Make a heap out of the elements of A This can be done in $\Theta(n)$ time !
- **Extract_Max(A)** n times [Each one takes $\Theta(\log n)$ time]

Worst-Case time complexity is $\Theta(n \log n)$

This sorting can be done "in-place" in A (Refer CLRS Section 6.4)