(Ch 22) Stack & Queue Applications

- Stack Application
  - Parenthesis Matching
  - Tower of Hanoi
- Priority Queue
  - Priority Queue using Heap
  - Heap Sort

Stack Application: Parenthesis Matching

- Print out matching of the left and right parentheses in a character string
  - (a*(b+c)+d) : output ➔ (0,10), (3,7) match
  - (a+b))(
    - Output ➔ (0, 4) match
    - Output ➔ 5, 6 have no matching parentheses

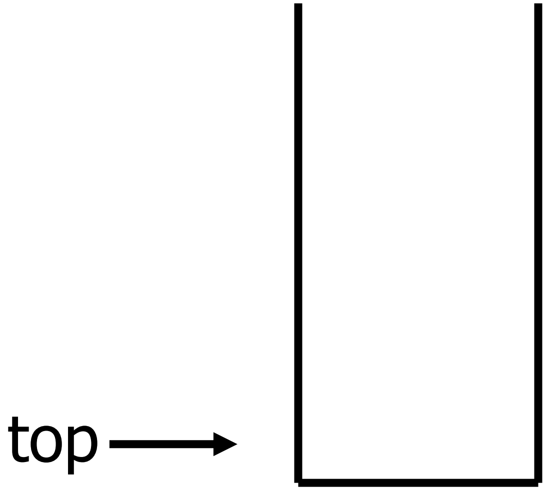
- Algorithm
  - Assume the stack class is ready (push(), pop() is available)
  - Scan the input expression from left to right
  - ‘ (‘ is encountered, add its position to the stack
  - ‘) ‘ is encountered, remove matching position from stack

- Complexity
  - Push / pop operations : O(n) time

Example: Parenthesis Matching [1/10]

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

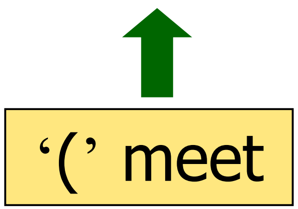


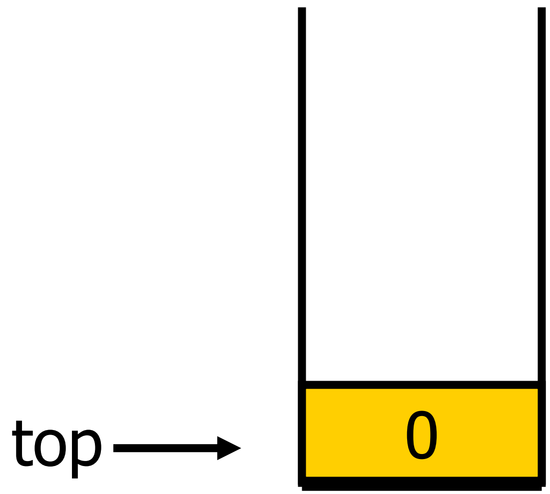
Stack for "("

Example: Parenthesis Matching [2/10]

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



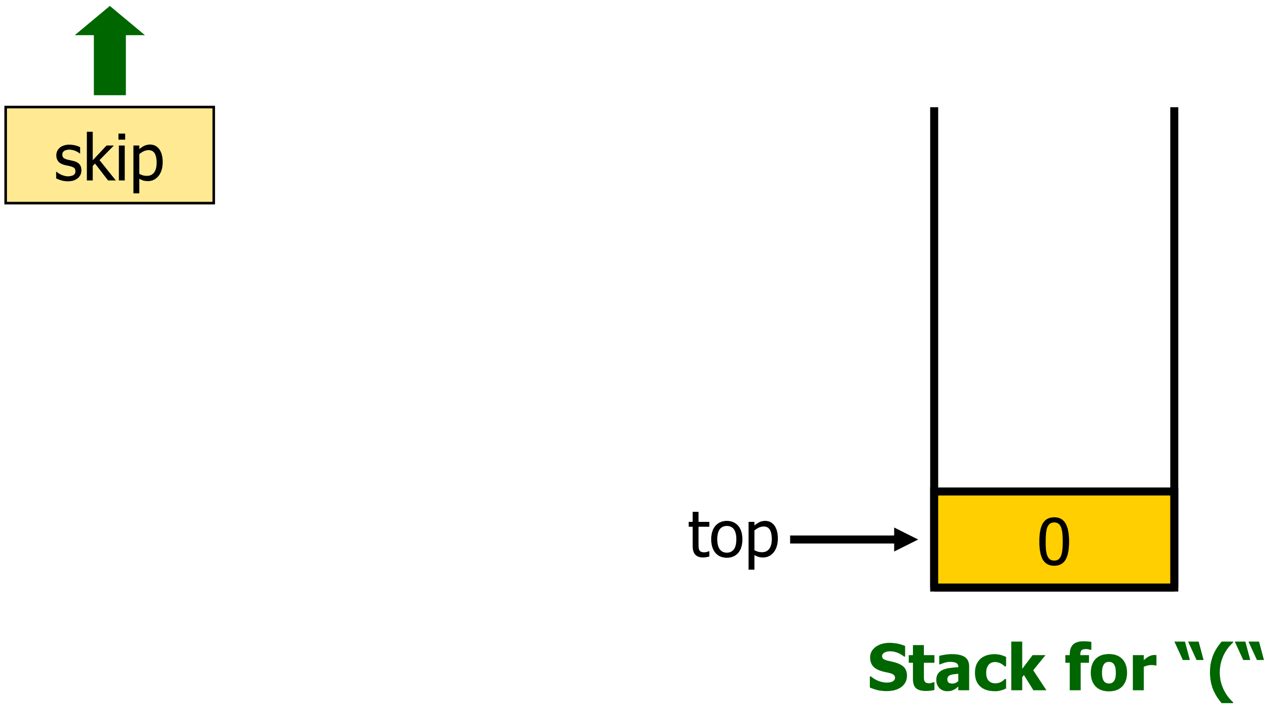


Stack for “(“

Example: Parenthesis Matching [3/10]

- (a*(b+c)+d)

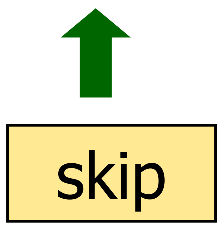
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

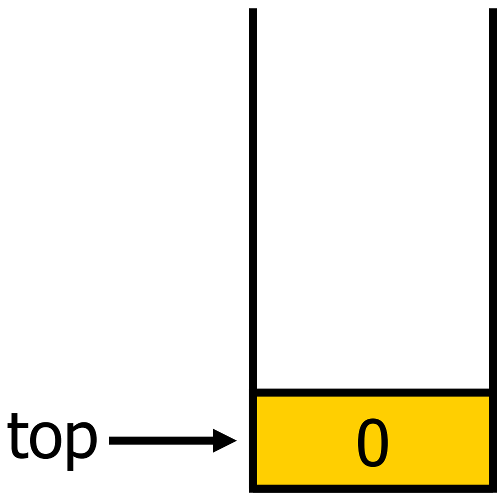


Example: Parenthesis Matching [4/10]

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)





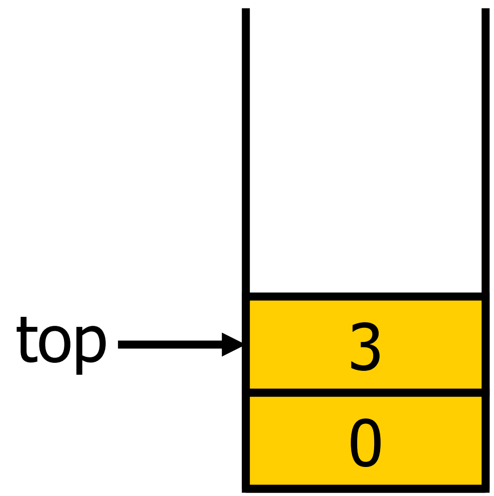
Stack for “(“

Example: Parenthesis Matching [5/10]

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

‘(‘ meet

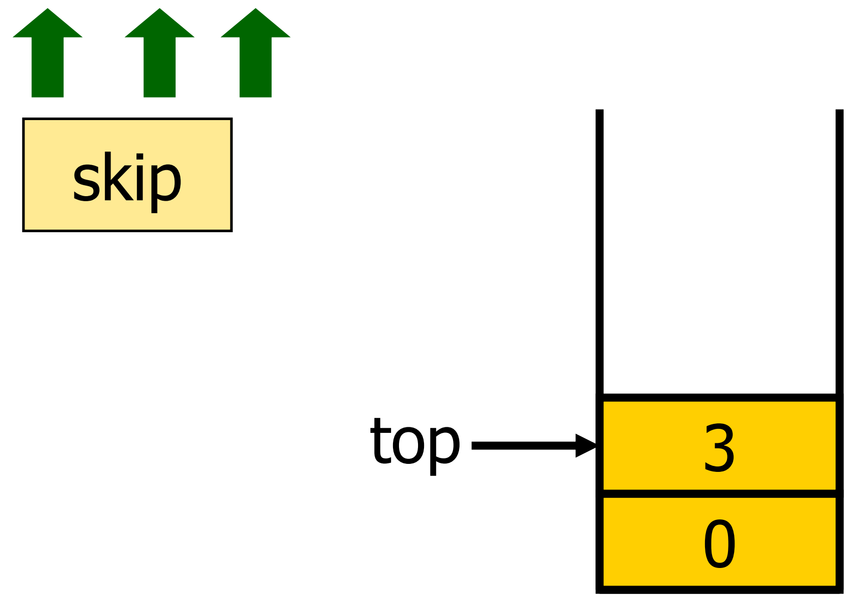


Stack for “(“

Example: Parenthesis Matching [6/10]

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

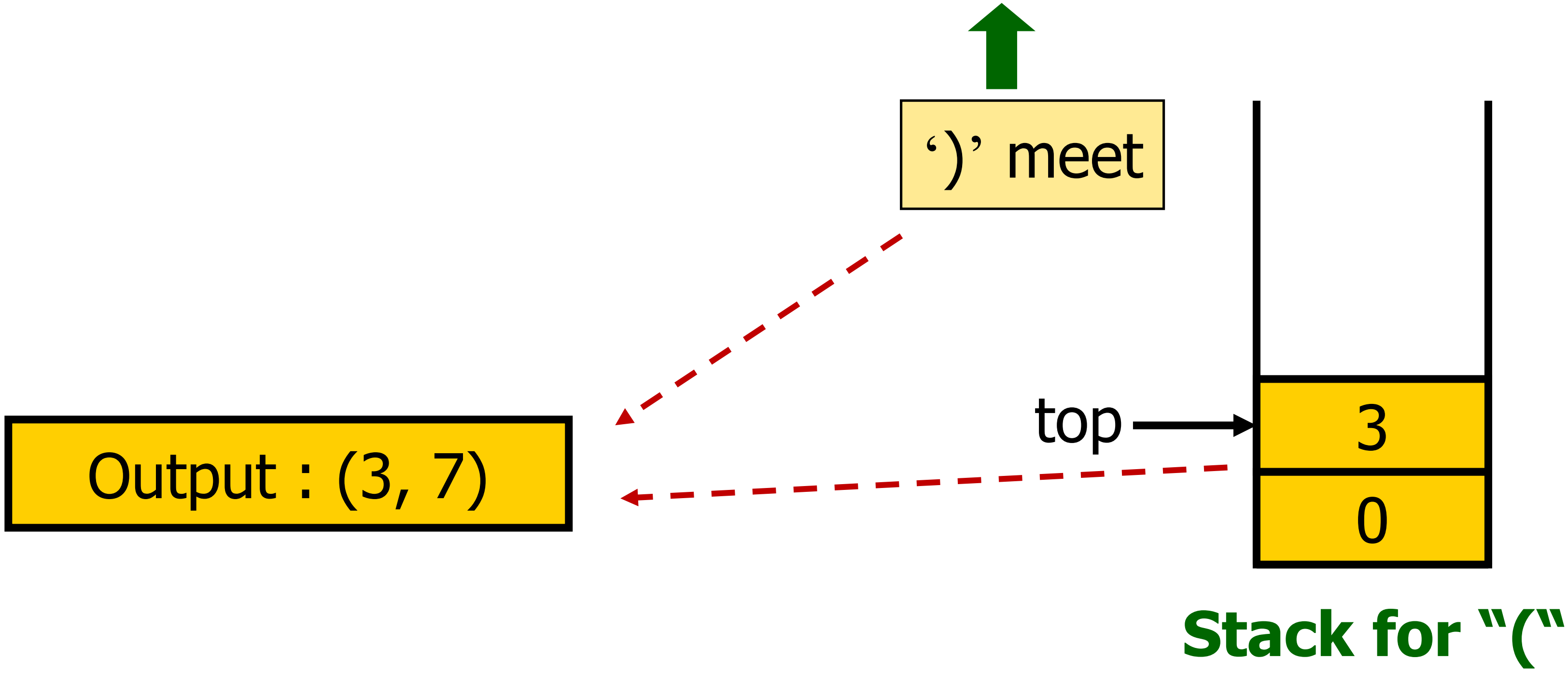


Stack for “(“

Example: Parenthesis Matching [7/10]

- (a*(b+c)+d)

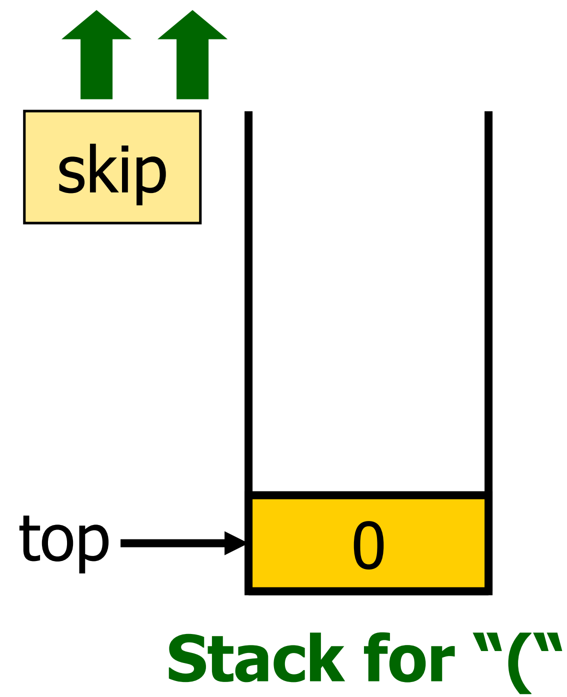
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



Example: Parenthesis Matching [8/10]

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

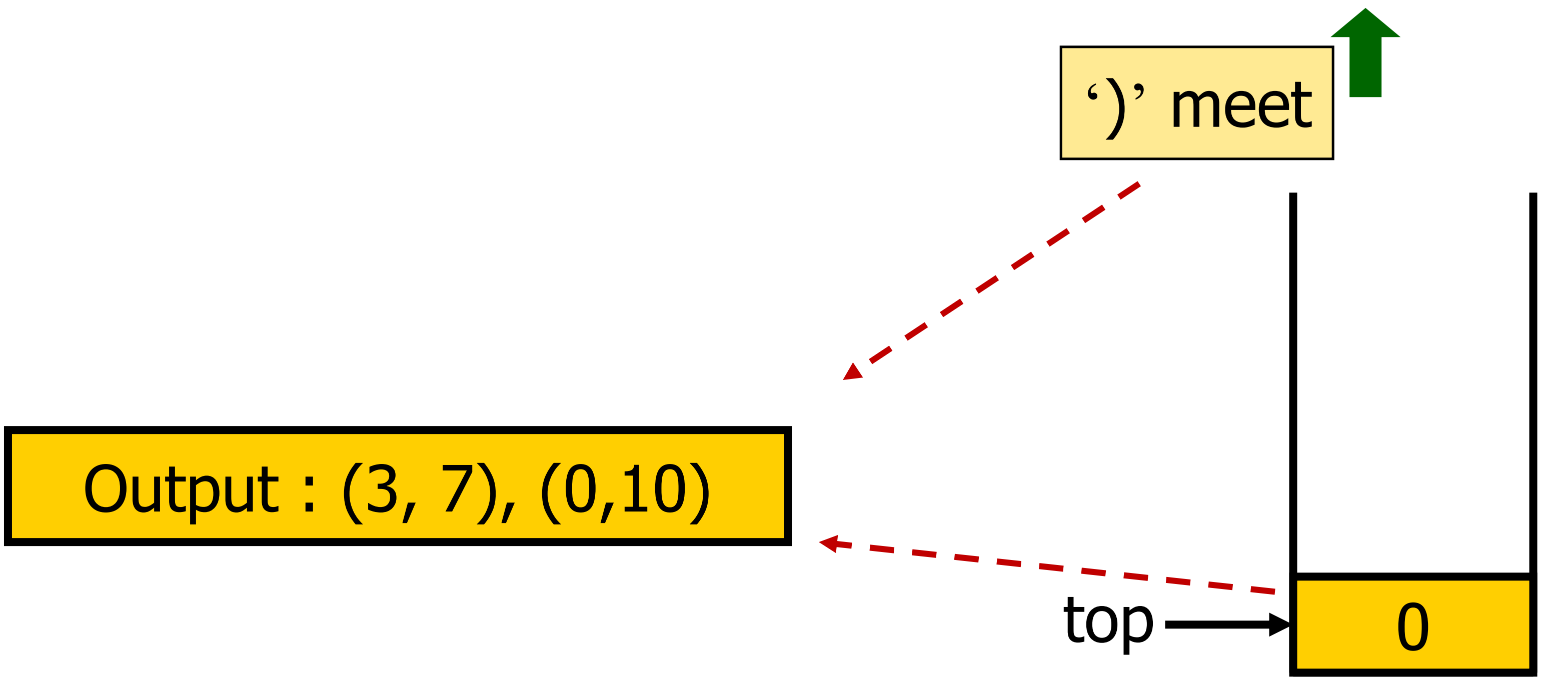


Output : (3, 7)

Example: Parenthesis Matching [9/10]

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



Stack for “(“

Example: Parenthesis Matching [10/10]

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

End!

Output : (3, 7), (0, 10)

Stack for "("

JAVA Code for Parenthesis Matching

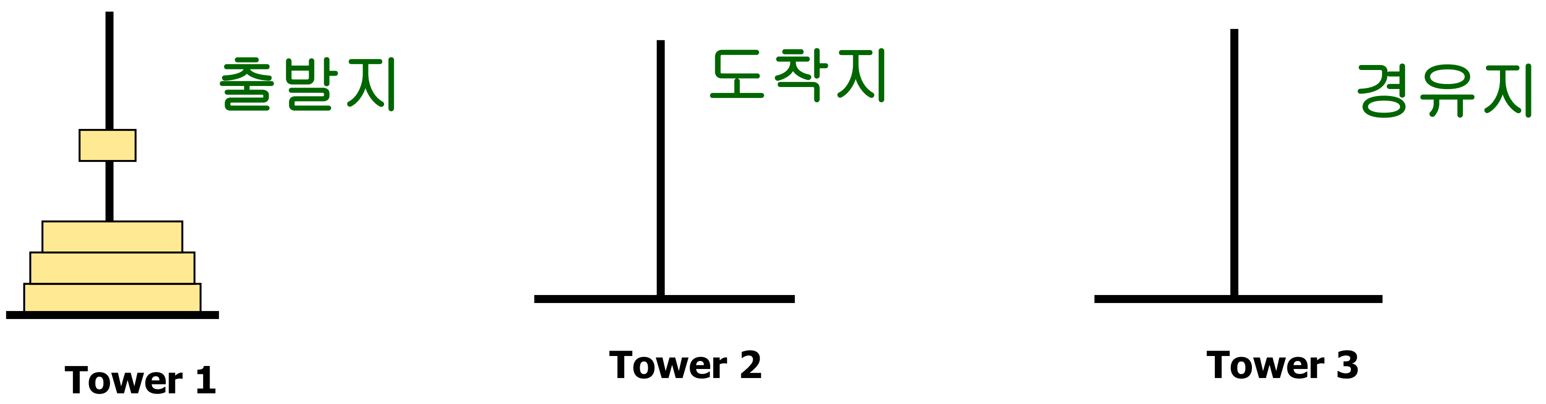
```
import dataStructures
public static void printMatchedPairs (String expr) {
    ArrayStack s = new ArrayStack()
    int length = expr.length()
    // scan expression expr for ( and )
     for (int i = 0; i < length; i++)
         if ( expr.charAt(i) == '(' )
               s.push(new Integer(i));
         else  if (expr.charAt(i) == ')')
                  try {  // remove location of matching '(' from stack
                         System.out.println( s.pop() + "  " + i );  }
                  catch (Exception e)   { // stack was empty, no match exists   }

     // remaining '(' in stack are unmatched
     while ( !s.empty() )
        System.out.println("No match for left parenthesis at " + s.pop() );
  }
```

(22) Stack & Queue Applications

- Stack Application
  - Parenthesis Matching
  - Tower of Hanoi
- Priority Queue
  - Priority Queue using Heap
  - Heap Sort

Stack Application: Towers of Hanoi



- Mission: Move the disks from tower1 to tower2

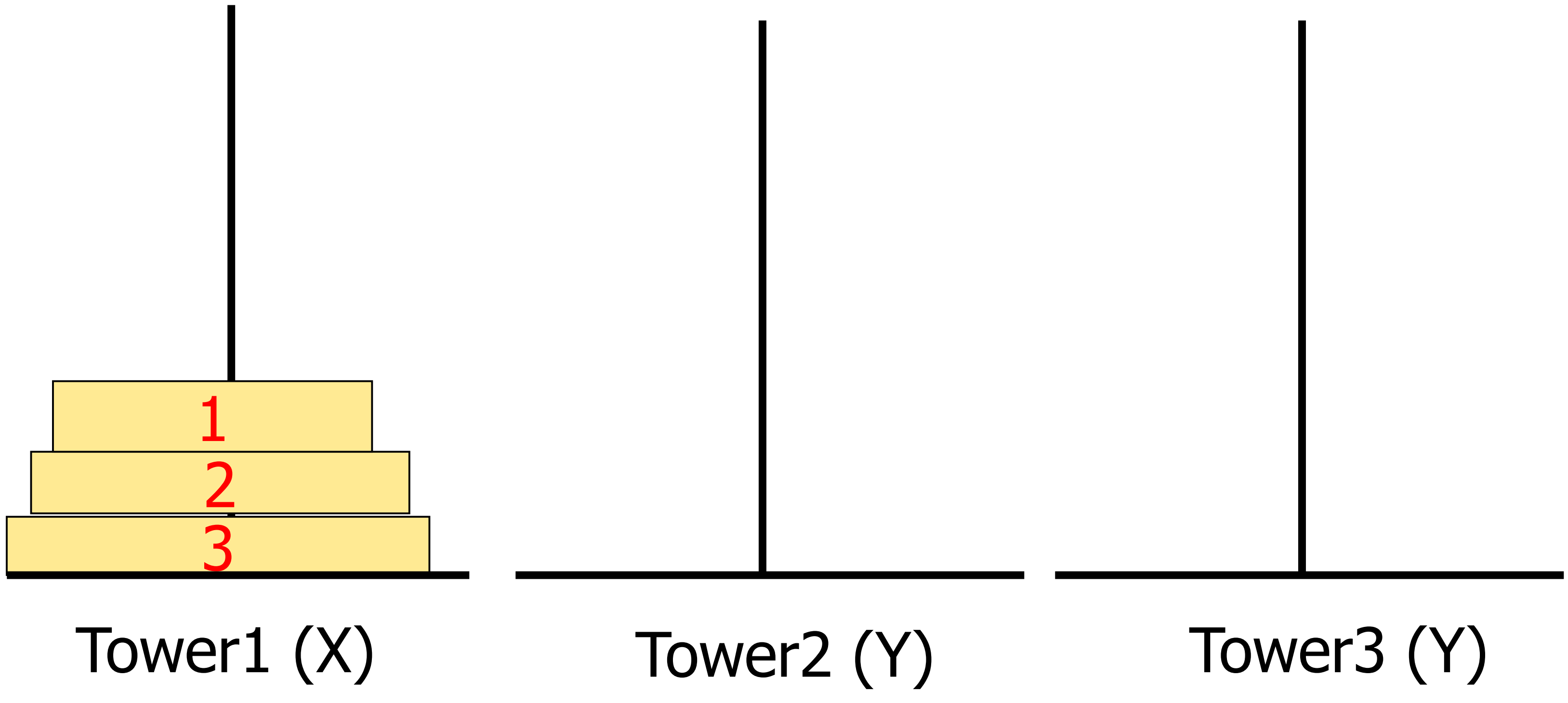
- Each tower operates as a stack
- Cannot place a big disk on top of a smaller one
  - Move n-1 disks to tower3 using tower2
  - Move the largest to tower2
  - Move the n-1 disks from tower3 to tower2 using tower1

- Move(N, Tower_A, Tower_B) : N개의 disk를 Tower_A 에서 Tower_B 이동

- Use of Recursion ➔ The runtime recursion stack is used!

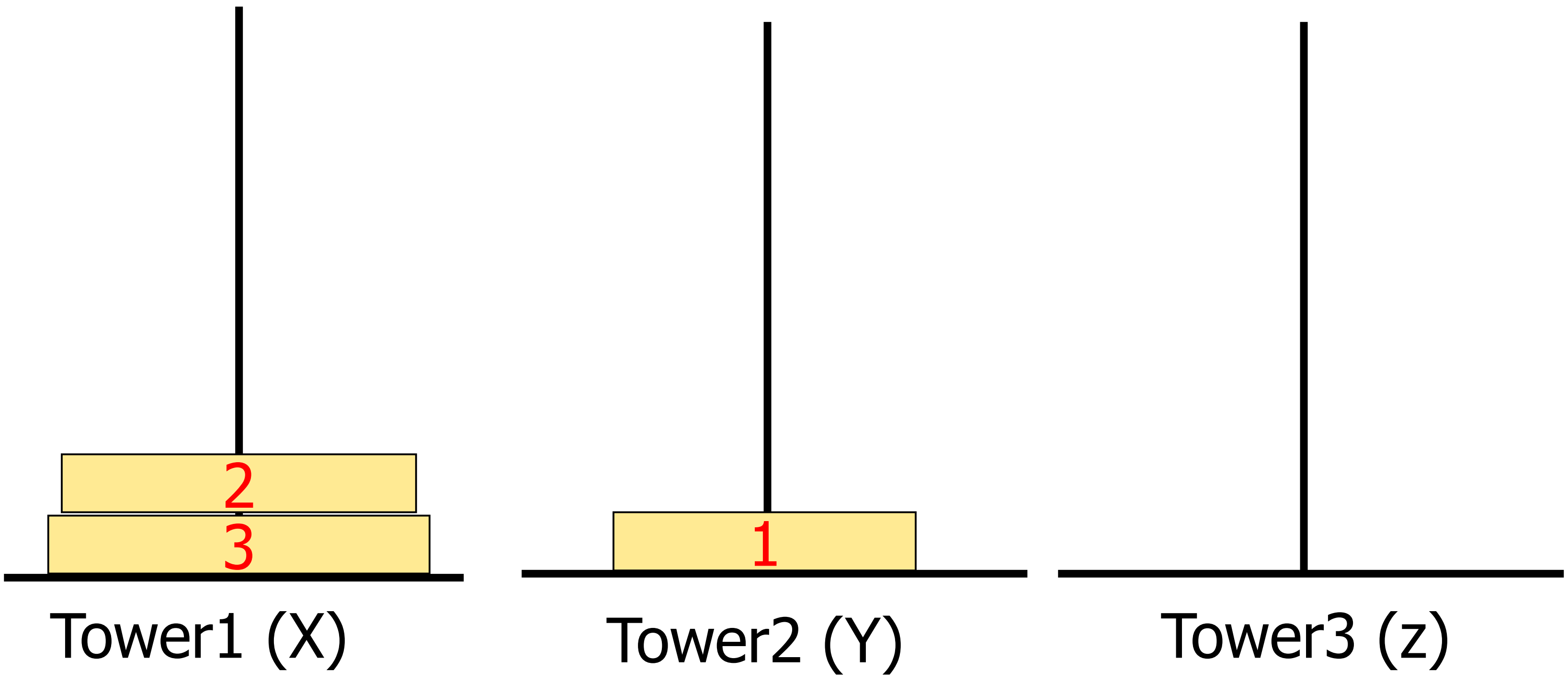
TOH Example [1/8]

- Mission: Move the disks from Tower1 to Tower2



TOH Example [2/8]

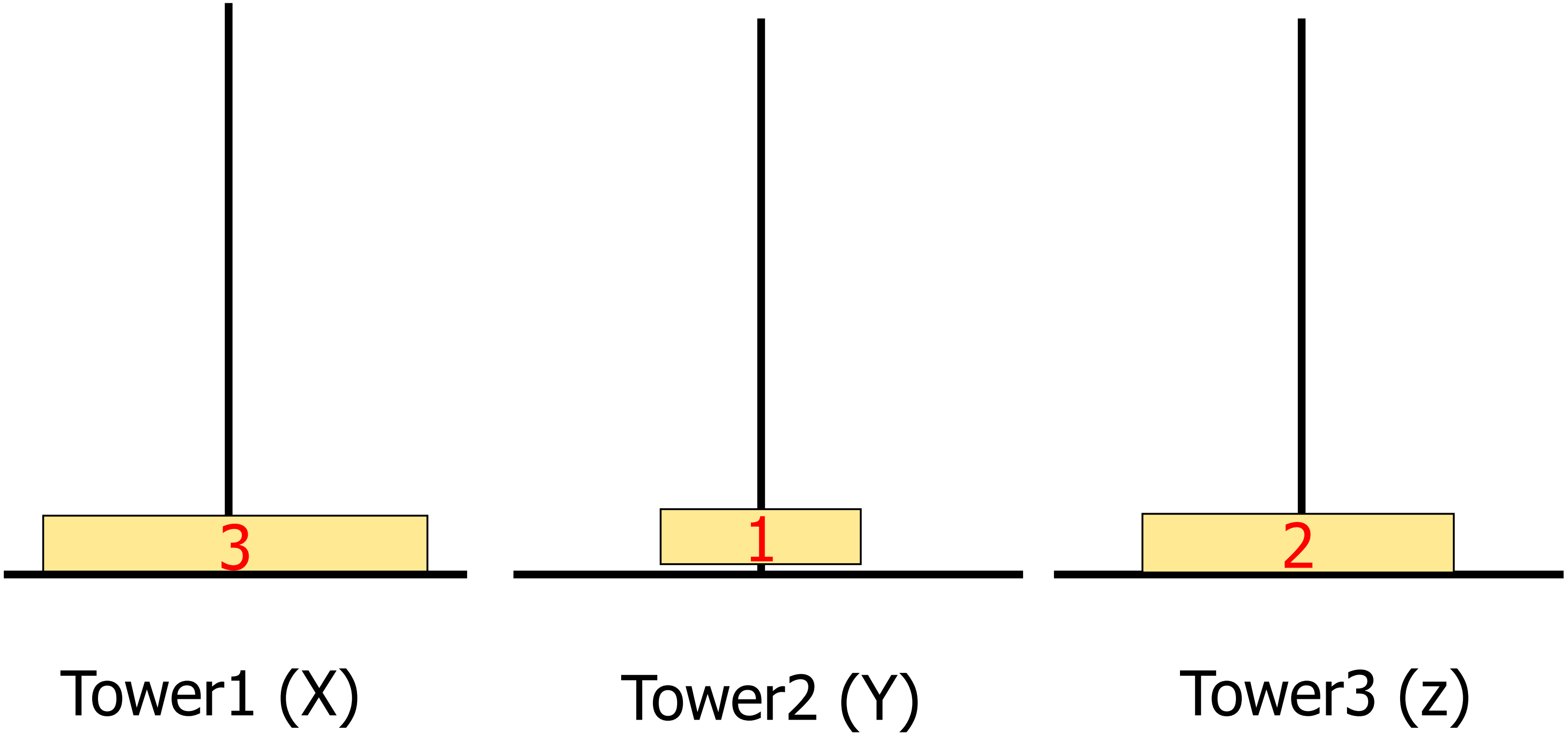
- Mission: Move the disks from Tower1 to Tower2



move 1 from x to y

TOH Example [3/8]

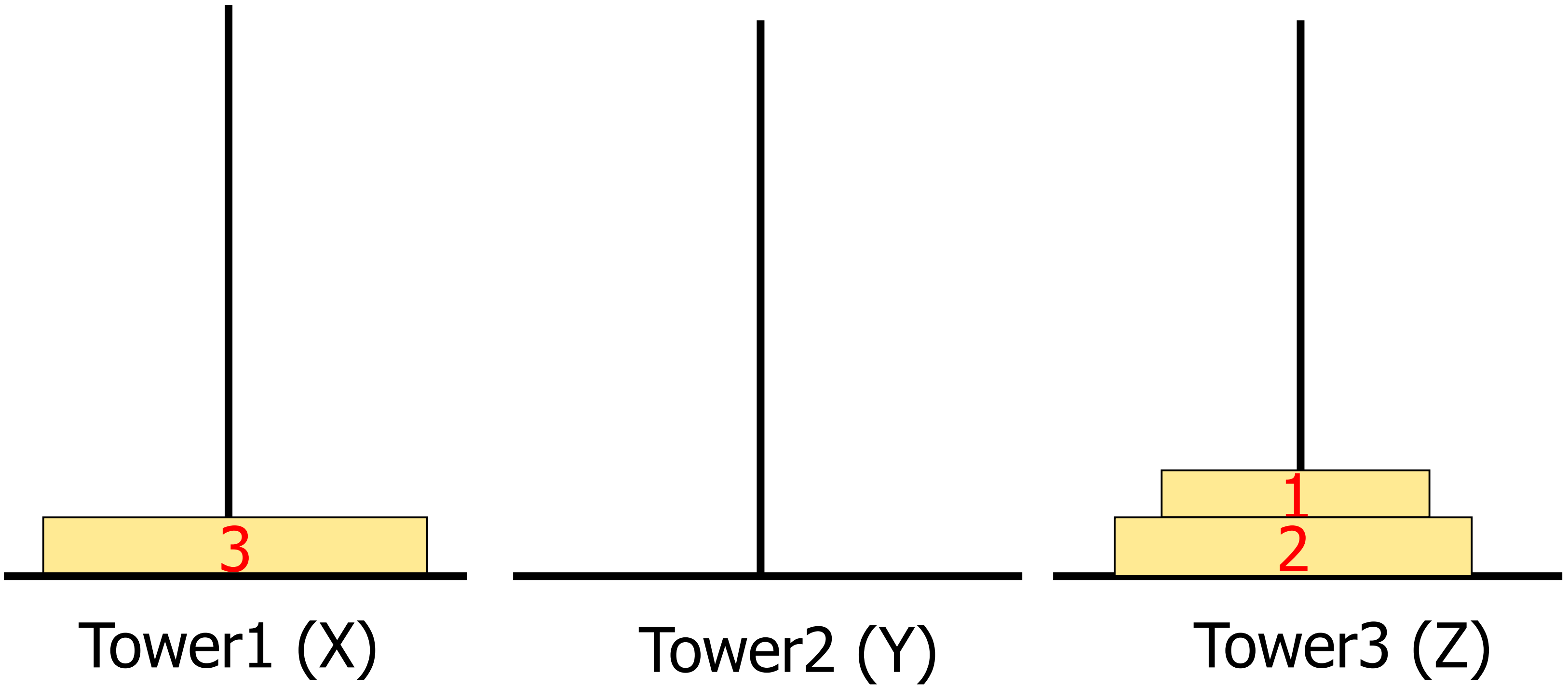
- Mission: Move the disks from Tower1 to Tower2



move 2 from x to z

TOH Example [4/8]

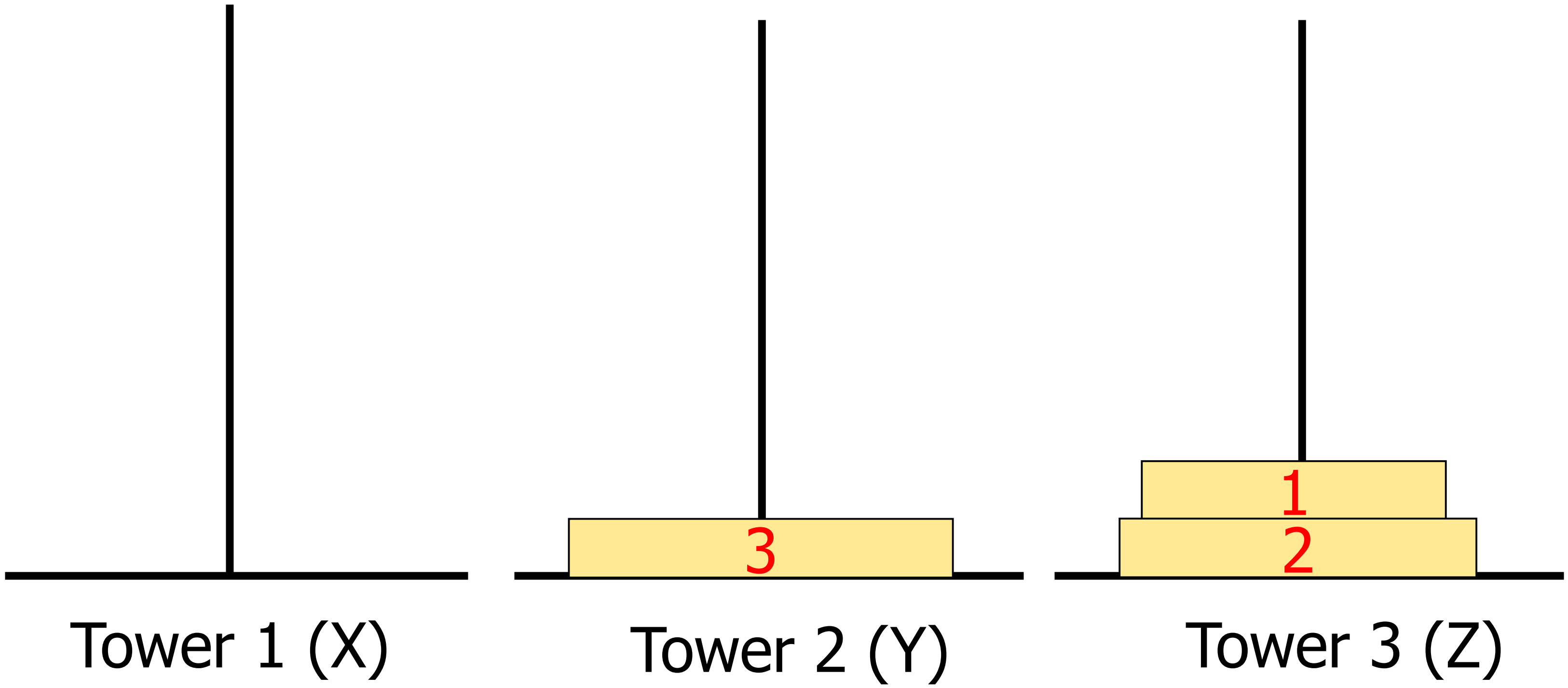
- Mission: Move the disks from Tower1 to Tower2



move 1 from y to z

TOH Example [5/8]

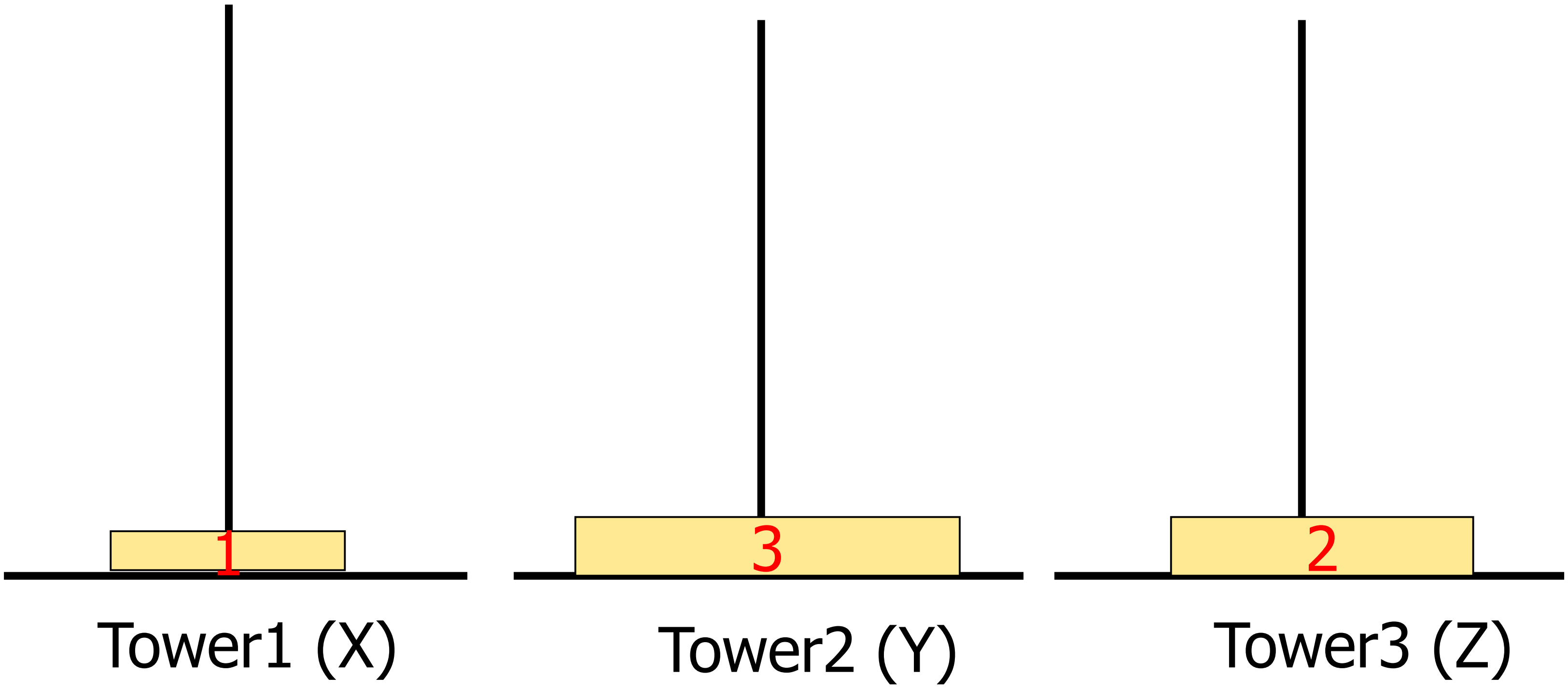
- Mission: Move the disks from Tower1 to Tower2



move 3 from x to y

TOH Example [6/8]

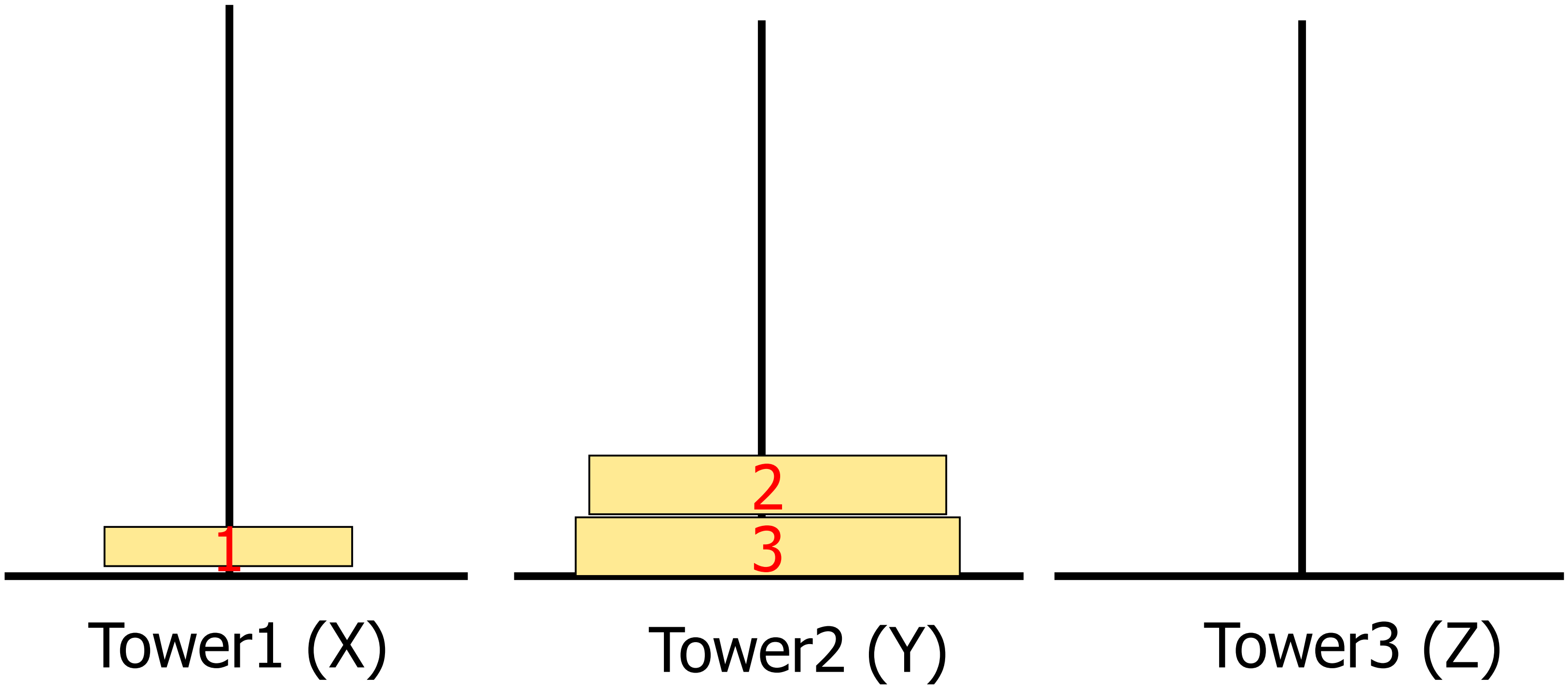
- Mission: Move the disks from Tower1 to Tower2



move 1 from z to x

TOH Example [7/8]

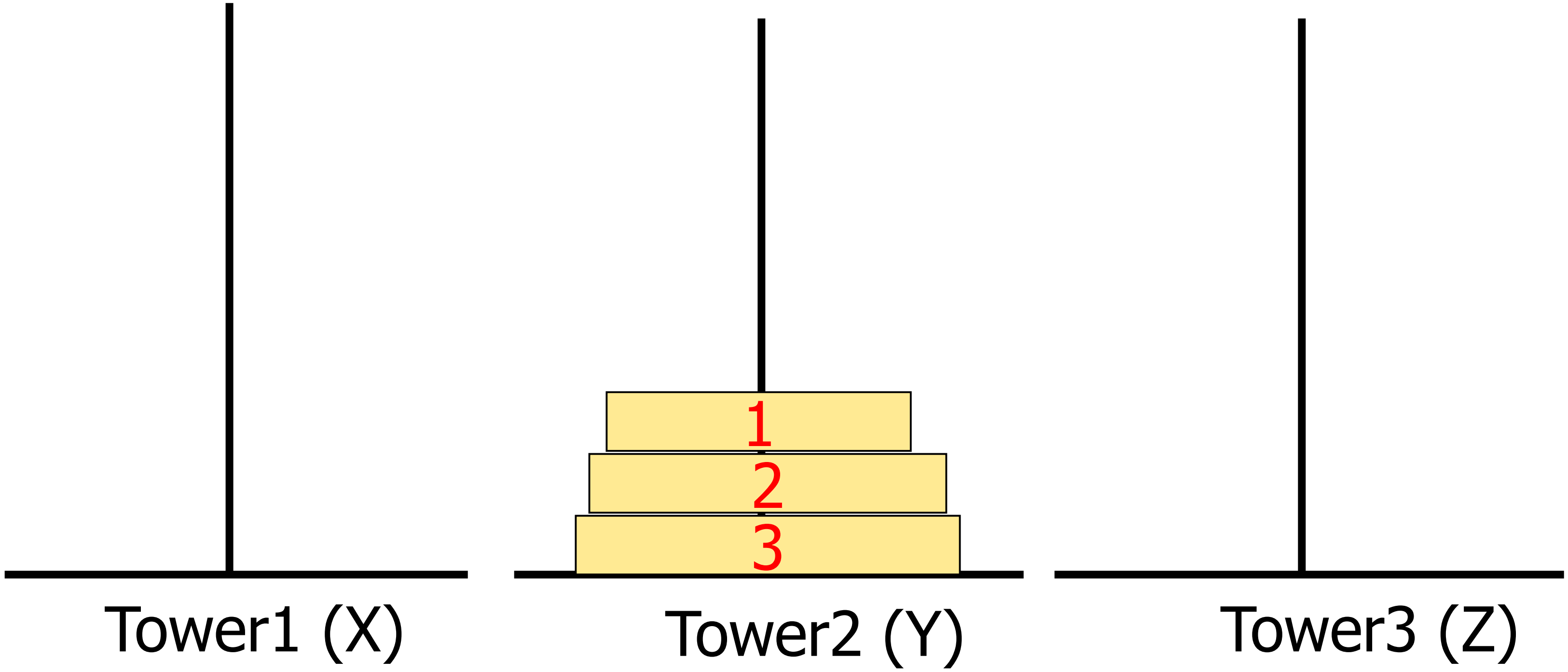
- Mission: Move the disks from Tower1 to Tower2



move 2 from z to y

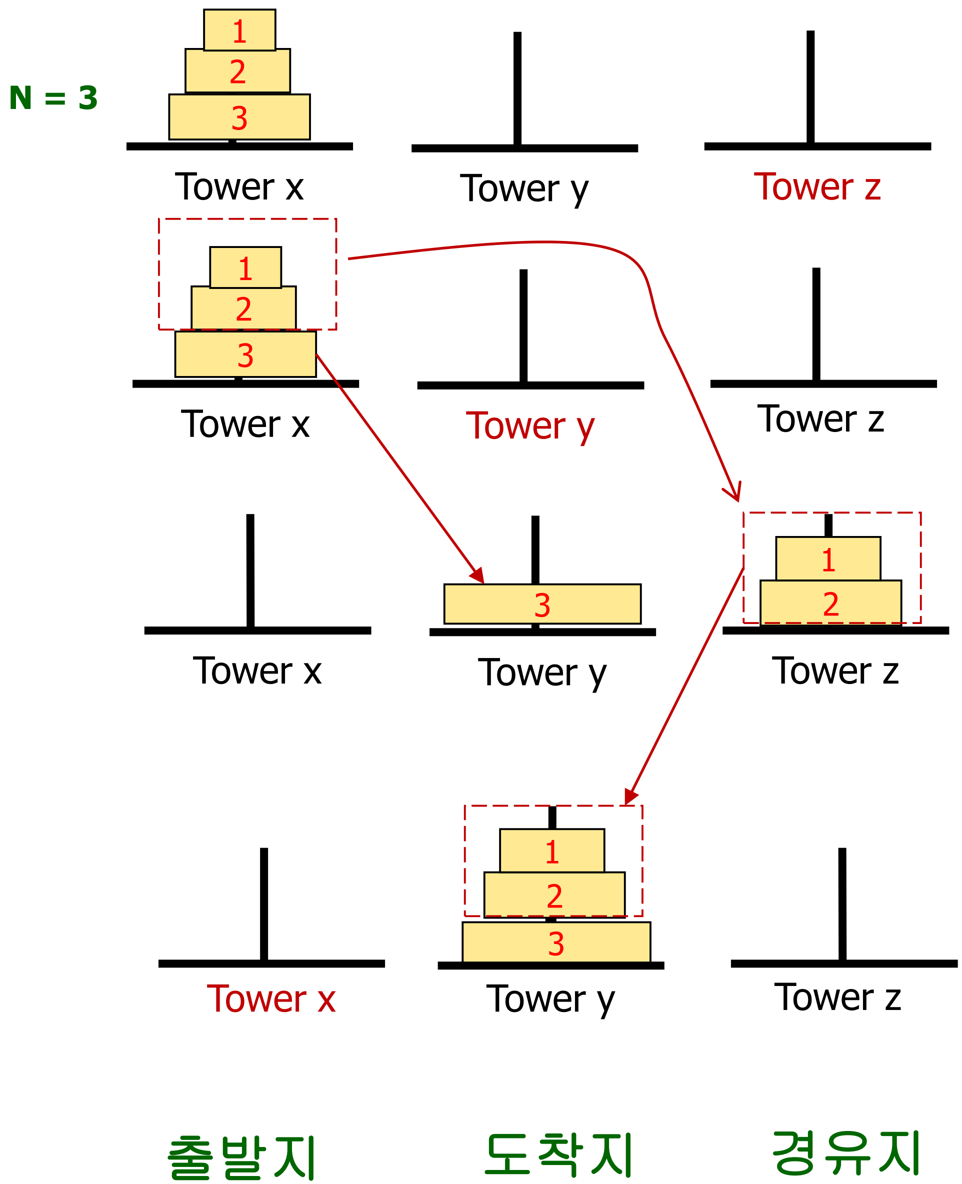
TOH Example [8/8]

- Mission: Move the disks from Tower1 to Tower2



move 1 from x to y

Recursion Mechanism in Tower of Hanoi



towersOfHanoi (3, x, y, z)
// move 3 disks from x to y, 경유지는 z

towersOfHanoi (2, x, z, y)
// move 2 disks from x to z, 경유지는 y

move topdisk from x to y

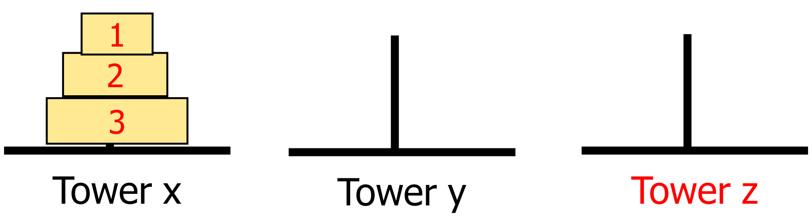
towersOfHanoi (2, z, y, x)
// move 2 disks from z to y, 경유지는 x

Code_1 : towersOfHanoi(n,1,2,3)

```
def  towersOfHanoi (n, x, y, z) {
    // Move the top n disks from  x to  y using z as intermediate storage
    if (n > 0):
      towersOfHanoi (n-1, x, z, y);
      print("Move the top disk from tower " + x +  " to top of tower " + y);
      towersOfHanoi (n-1, z, y, x);
    }
  }
```

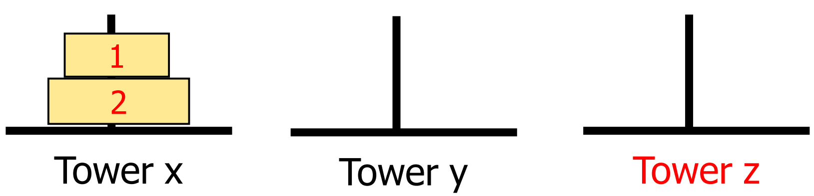
There is no actual stack for each tower, just recursion program is using the system's function recursion statck.

N = 3



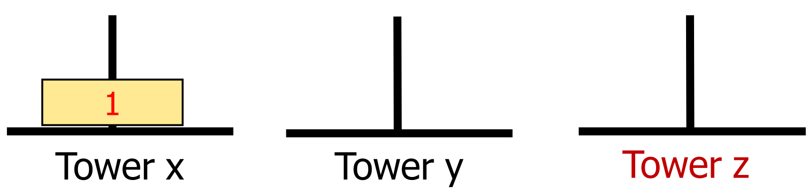
towersOfHanoi (3, x, y, z)
- → **towersOfHanoi (2, x, z, y) // move x to z**
- **move topdisk from x to y**
- **towersOfHanoi (2, z, y, x) // move x to z**

N = 2



towersOfHanoi (2, x, y, z)
- → **towersOfHanoi (1, x, z, y) // move x to z**
- **move topdisk from x to y**
- **towersOfHanoi (1, z, y, x) // move x to z**

N = 1



towersOfHanoi (1, x, y, z)
- → **move topdisk from x to y**

Actual execution for 3 disks

TOH(3, x, y, z)

TOH(2, x, z, y)

TOH(1, x, y, z): <u>move 1 from x to y</u>

: <u>move 2 from x to z</u>

TOH(1, y, z, x): <u>move 1 from y to z</u>

<u>move 3 from x to y</u>

TOH(2, z, y, x)

TOH(1, z, x, y): <u>move 1 from z to x</u>

: <u>move 2 from z to y</u>

TOH(1, x, y, z): <u>move 1 from x to y</u>

Complexity: towerOfHanoi( )

- The number of moves: moves(n)
  - n = 0 : moves(n) = 0
  - n > 0 : moves(n) = 2 * moves(n-1) + 1

moves(n) = 2 * (2 * moves(n-2)) + 1

moves(n) = 2 * (2 * (2 * moves(n-3))) + 1

…

moves(n)= 2 * (2 * (2 * (2 * …. moves(1)))))) + 1

moves(n) = $2^n - 1$

- Therefore moves(n) = $2^n - 1$
  - Time Complexity of Tower of Hanoi : **O(2^n)**

Code_2 : Tower of Hanoi using Actual Stacks

- The towerOfHanoi( ) gives only printing disk-move sequences
- Show the actual state of the 3 towers (the disk order bottom to top) ➔ use 3 stacks!

```
def TowersOfHanoiShowingStates( ):
  tower = [ ]
  def  towersOfHanoi (int n) {
    // create three stacks
    // add n disks to tower 1
    showTowerStates(n, 1, 2, 3); // move n disks from tower 1 to tower 2 using 3 as intermediate tower
   }

  def  showTowerStates (n, x, y, z) { // Move the top n disks from x to y
  if (n > 0):
       showTowerStates(n-1, x, z, y);
        // move d from top of tower x to top of tower y
       print("Move disk " + d + " from tower "+ x + " to top of tower " + y);
       showTowerStates(n-1, z, y, x);
   }
}
```

(22) Stack and Queue Applications

- Stack Application
  - Parenthesis Matching
  - Tower of Hanoi
- Priority Queue
  - Priority Queue using Heap
  - Heap Sort

Definition

- A priority queue is
  - Collection of zero or more elements with priority
- A min priority queue is
  - Find the element with minimum priority
  - Then, Remove the element
- A max priority queue is
  - Find the element with maximum priority
  - Then, Remove the element

- Priority queue is a conceptual queue where the output element has a certain property (i.e., priority)

15	35	65	20	17	80	12	45	2	4

The ADT MaxPriorityQueue

AbstractDataType MaxPriorityQueue {

instances

finite collection of elements, each has a priority

operations

heapify( ) : return the queue with a bunch of items

isEmpty() : return true if the queue is empty

size() : return number of elements in the queue

getMax() : return element with maximum priority

put(x) : insert the element x into the queue

removeMax() : remove the element with largest priority from the queue and return this element;

}

Priority Queue in Python

- "queue" module
  - Queue Class
  - PriorityQueue Class
  - LifoQueue Class

- PriorityQueue class
  - Subclass of Queue
  - Retrieves entries in the priority order
  - Entries are tuples of the form: (priority number, data)
  - get() : retrieve and return an item from the queue
  - put(item) : put an item into a queue

Max Tree & Max Heap

- A max tree is a tree in which the value in each node is greater than or equal to those in its children

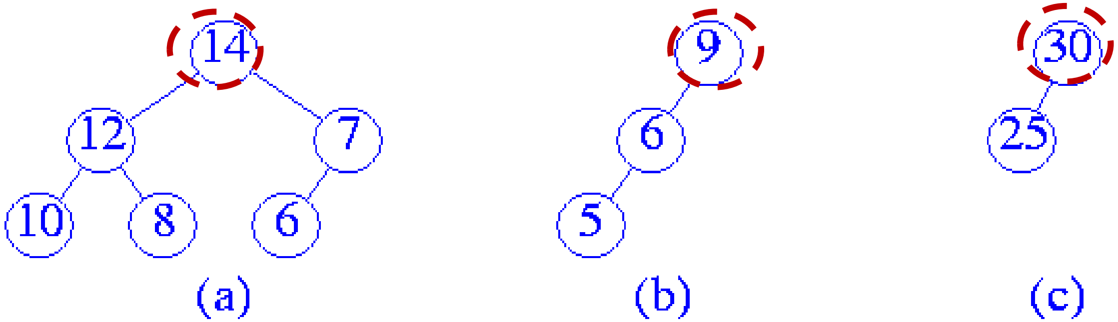


Figure 13.1 Max trees

모든 Node의 값은 Child Node의 값보다 같거나 커야 한다

- A max heap is
  - A max tree that is also a complete binary tree
  - Figure 13.1(b) : not CBT, so not max heap

아래상태의 queue를 유지하면서 priority가 높은 값을 retrieve하는것보다 MaxHeap Tree를 만들면 경제적!

15	35	65	20	17	80	12	45	2	4

Min Tree & Min Heap

- A min tree is a tree in which the value in each node is less than or equal to those in its children

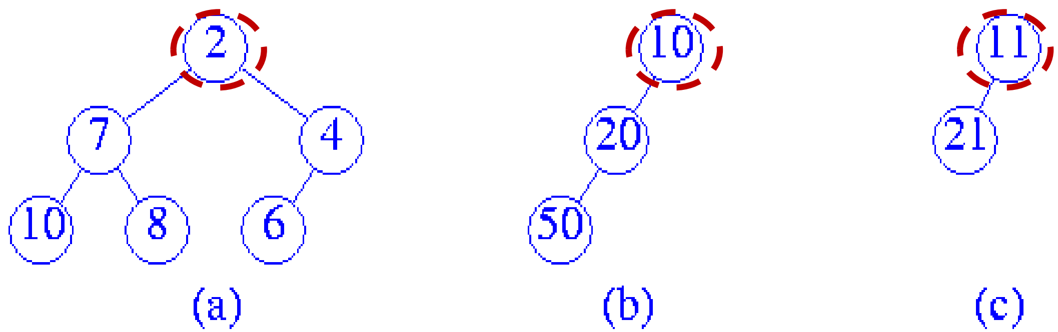


Figure 13.2 Min trees

모든 Node의 값은
Child Node의 값보다
같거나 작아야 한다

- A min heap is
  - A min tree that is also a complete binary tree
  - Figure 13.2(b) : not CBT, so not min heap

Height of Heap Tree

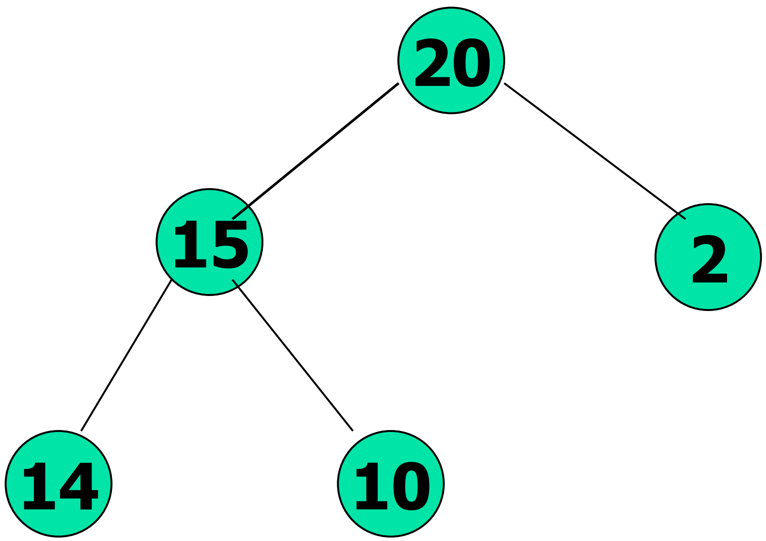
- Heap is a complete binary tree
  - A heap with n elements has height $\lceil \log_2(n+1) \rceil$

- put(): 0(height) ➔ 0(log n)
  - Increase array size if necessary
  - Find place for the new element
    - The new element is located as a leaf
    - Then moves up the tree for finding home

- removeMax(): 0(height) ➔ 0(log n)
  - Remove heap[1], so the root is empty
  - Move the last element in the heap to the root
  - Reheapify

Put an Element into Max Heap [1/4]

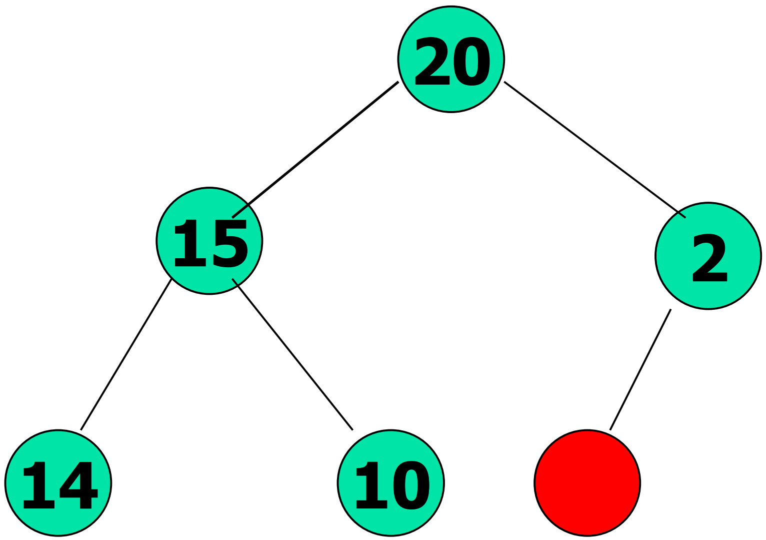
- Suppose Max Heap has five elements



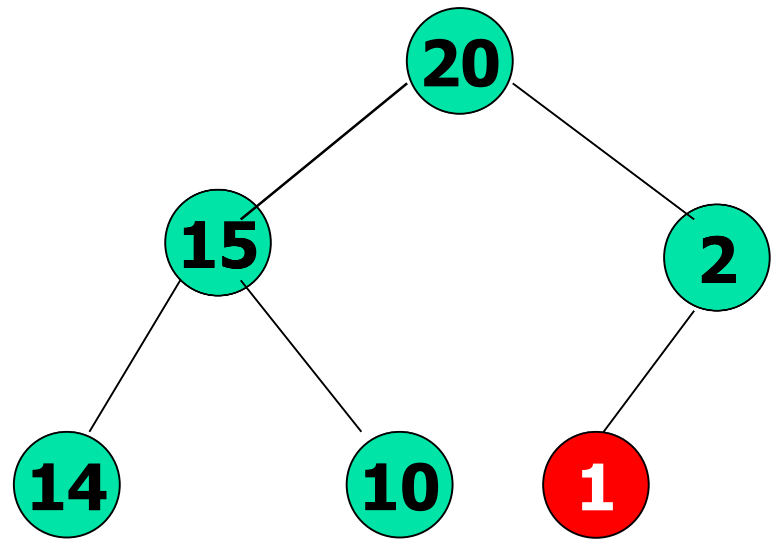
모든 **Node**의 값은 **Child Node**의 값보다 같거나 커야 한다

Put an Element into Max Heap [2/4]

- Max Heap is a complete binary tree
- When an element is added to this heap, the location for a new element is the red zone

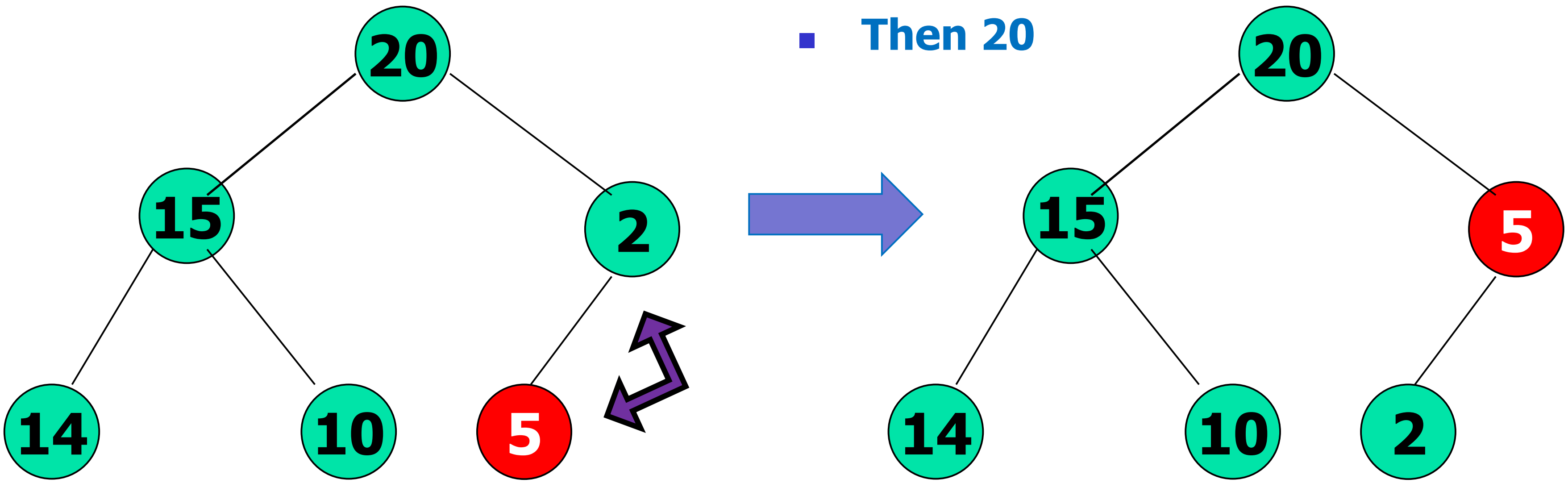


- Suppose the element to be inserted has value 1, the following placement is fine because 2 > 1



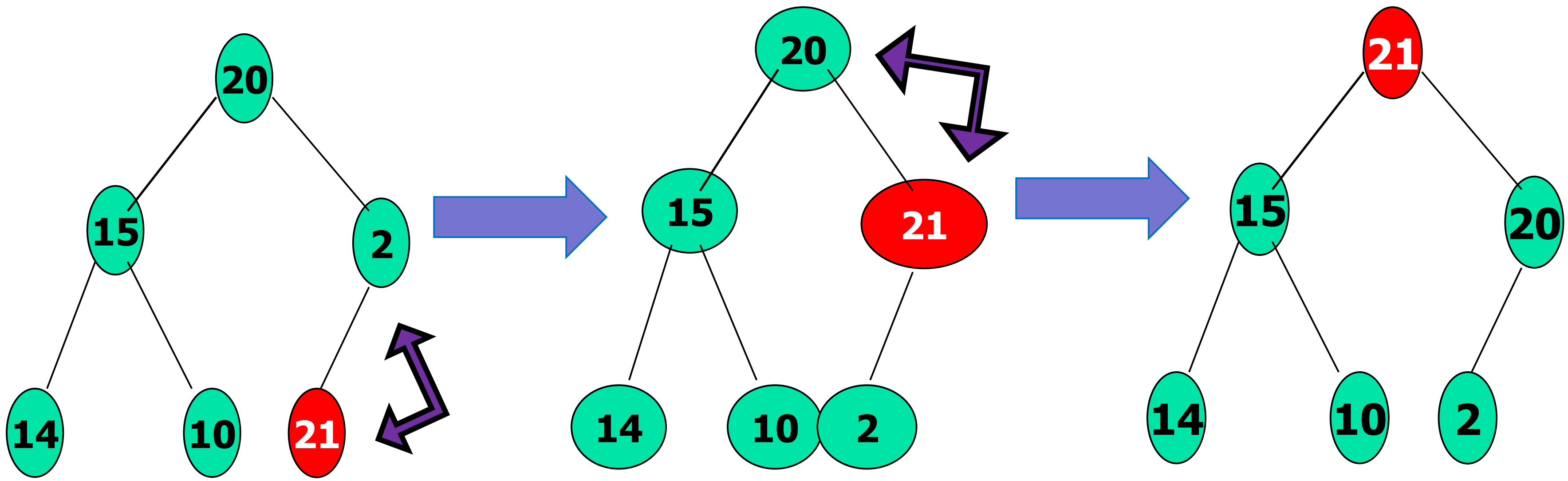
Put an Element into Max Heap class [3/4]

- Suppose the element to be inserted has value 5
- **The elements 2 and 5 must be swapped for maintaining the heap property because 5 > 2**
- **Then 20**



Put an Element into Max Heap [4/4]

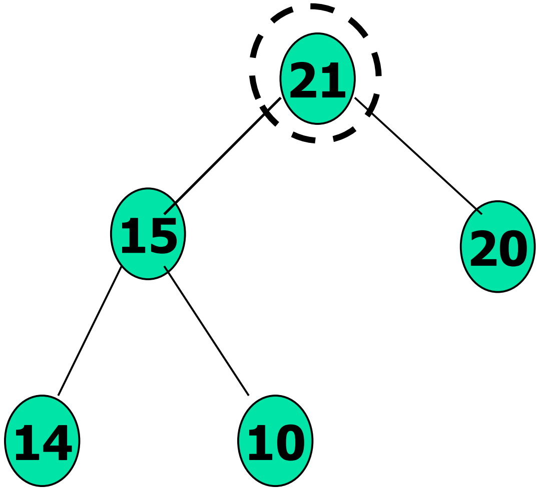
- Suppose the element to be inserted has value 21
- **The new element 21 will find its position by continuous swapping with the existing elements for maintaining the heap property**
- **Finally the new element 21 goes to the top**

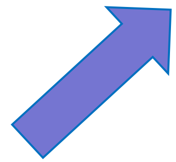


모든 **Node**의 값은 **Child Node**의 값보다 같거나 커야 한다

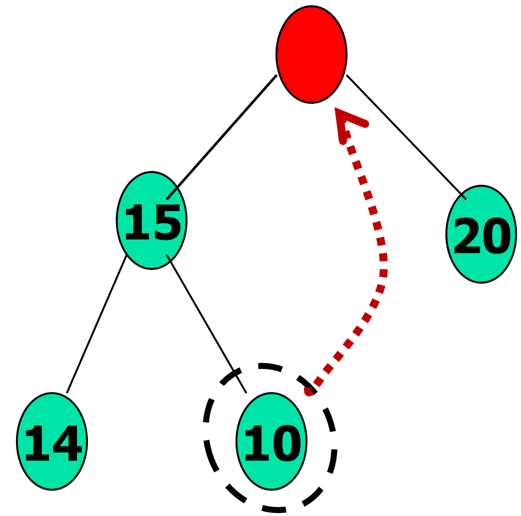
RemoveMax() from a MaxHeap class [1/2]

- The Max element “21” is in the root

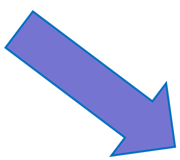


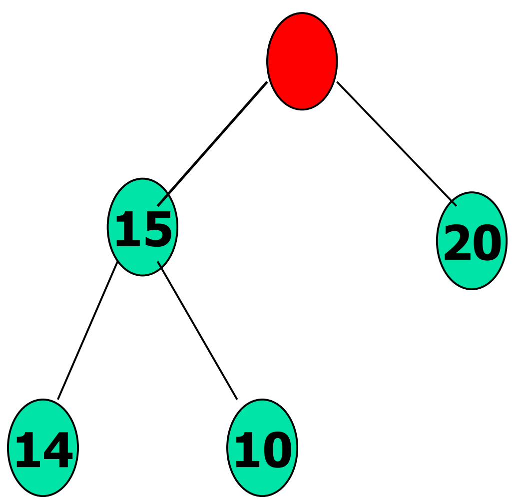


- Need to move the last leaf node to the top



- After the max element “21” is removed



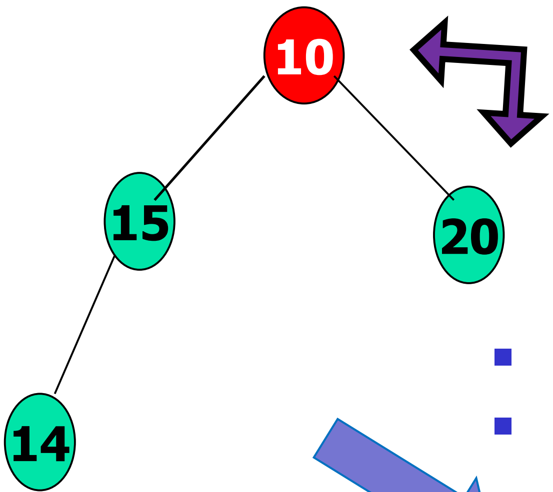


removeMax( ) = pop( )

RemoveMax() from a MaxHeap class [2/2]

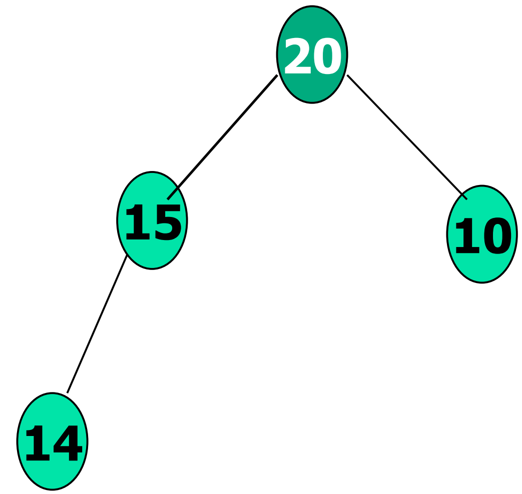
- Try to put **"10"** into the root,

then it is not a maxheap

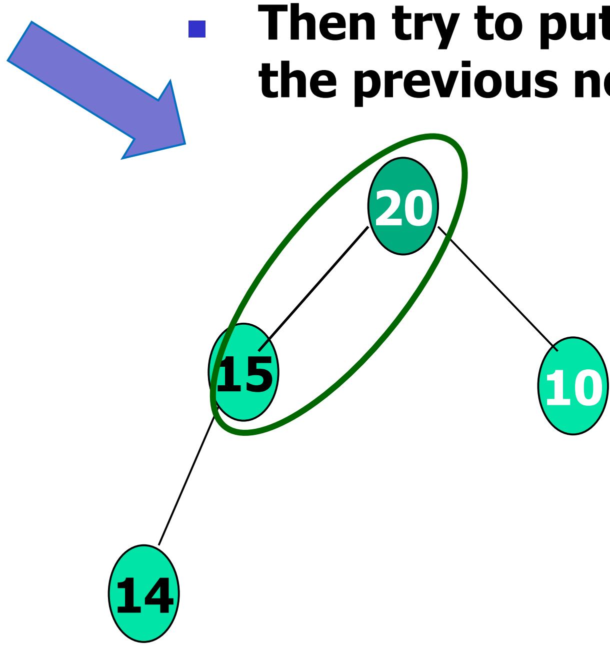


모든 Node의 값은
Child Node의 값보다
갈거나 커야 한다

- **This is a MaxHeap!**



- **Put 20 into the root**
- **Then try to put 10 into the previous node of 20**



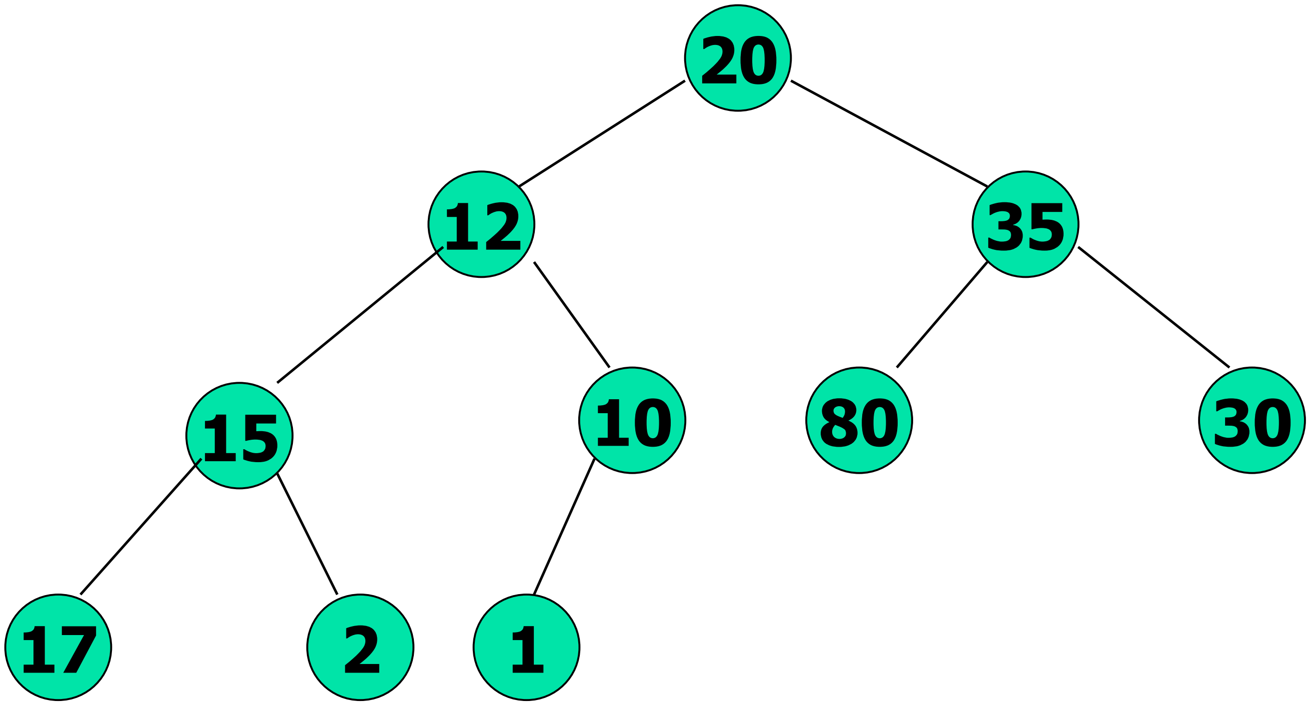
- **Compare 20 and 15**
- **This is OK**

Initialization (Heapify( )) in MaxHeap class

- Steps (from an array to a MaxHeap tree)
  - Allocate the elements in an array
  - Form a complete binary tree
  - In the array, start with the rightmost node having a child (= non-leaf node)
    - node number → n/2
  - Fix the heap in the node
  - Reverse back to the first (root) node in the binary tree

MaxHeap Initialization: heapify( ) [1/12]

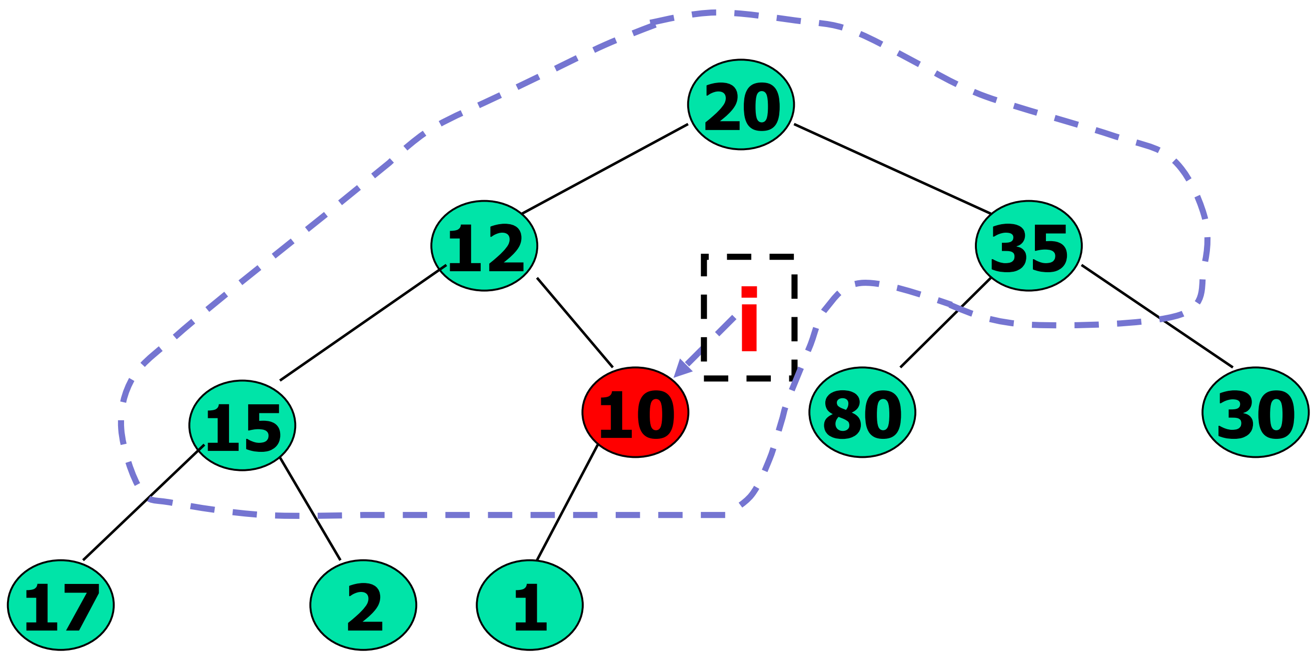
- Input array = [20, 12, 35, 15, 10, 80, 30, 17, 2, 1]
- Just make a complete binary tree
- We will check every non-leaf nodes
- Node swap이 발생하면 아래로 내려간 node의 new home을 찾아야함



모든 **Node**의 값은 **Child Node**의 값보다 같거나 커야 한다

MaxHeap Initialization : heapify( ) [2/12]

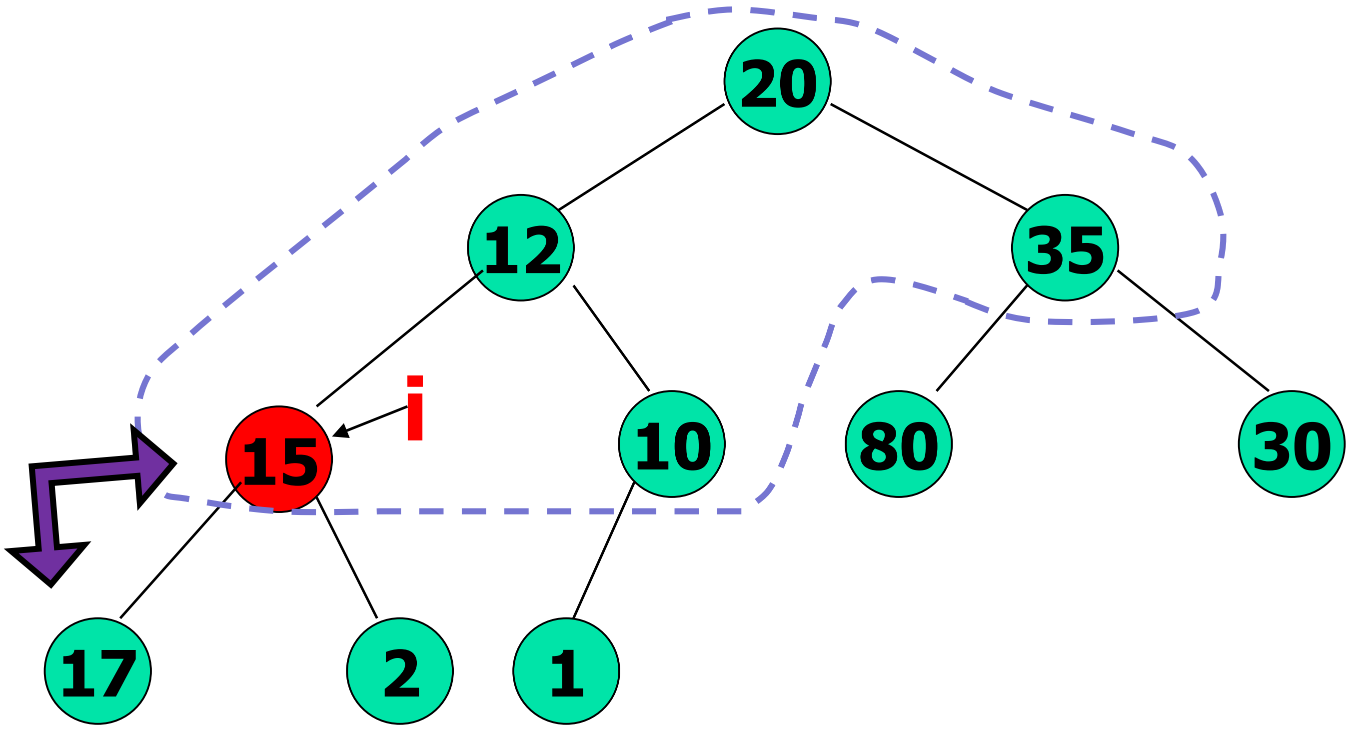
- Start at rightmost array position that has a child (non-leaf node)
  - Start from the rightmost non-leaf node whose index i is (n/2)th of the array
  - Check the node with its child nodes



This is OK!

MaxHeap Initialization : heapify( ) [3/12]

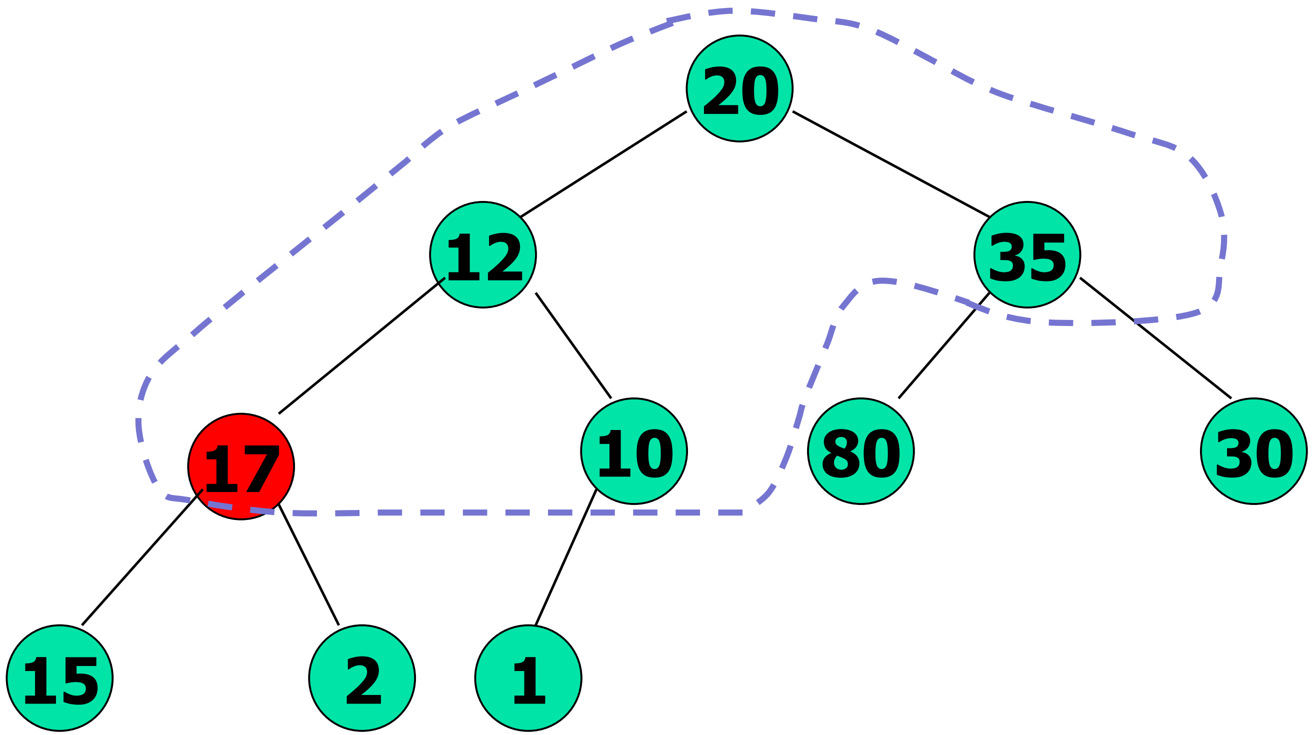
- Move to the next rightmost non-leaf node
- Check the node with its child nodes



This is not OK!
Need a swap!

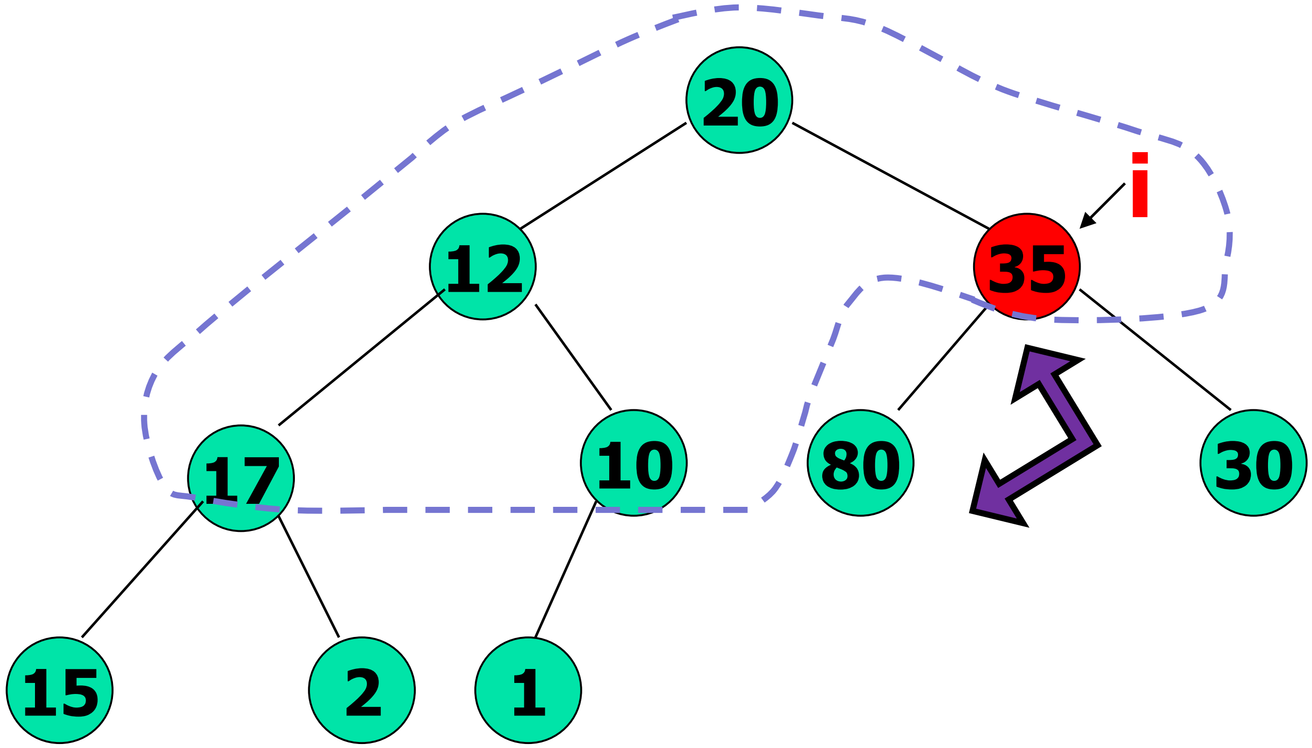
MaxHeap Initialization : heapify( ) [4/12]

- Replace a node with 17 and a node with with 15
- Now, the current position is OK



MaxHeap Initialization : heapify( ) [5/12]

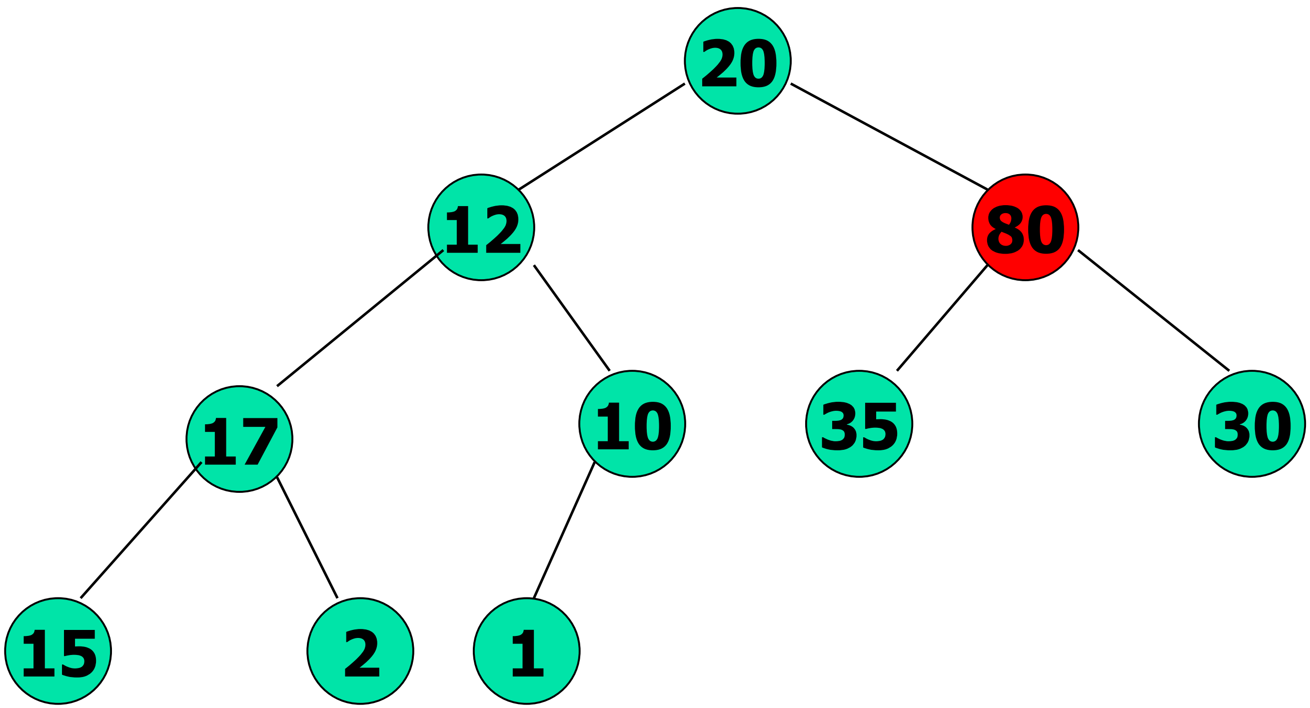
- Move to next lower array position
- Check the node with its child nodes



This is not OK!
Need a swap

MaxHeap Initializatio : heapify( ) [6/12]

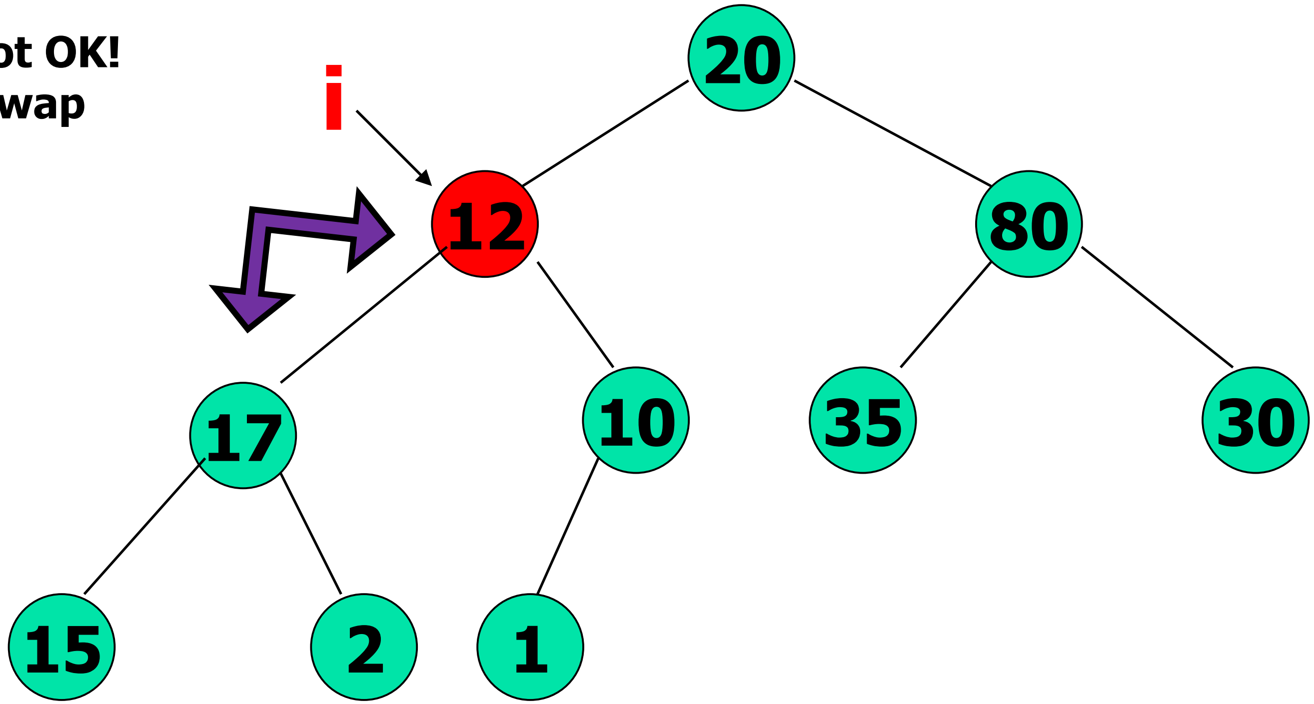
- Replace a node with 80 and a node with 35
- Need to find a new home for 35, but the current is OK
- The node with 80 will be checked later with the parent



MaxHeap Initialization : heapify( ) [7/12]

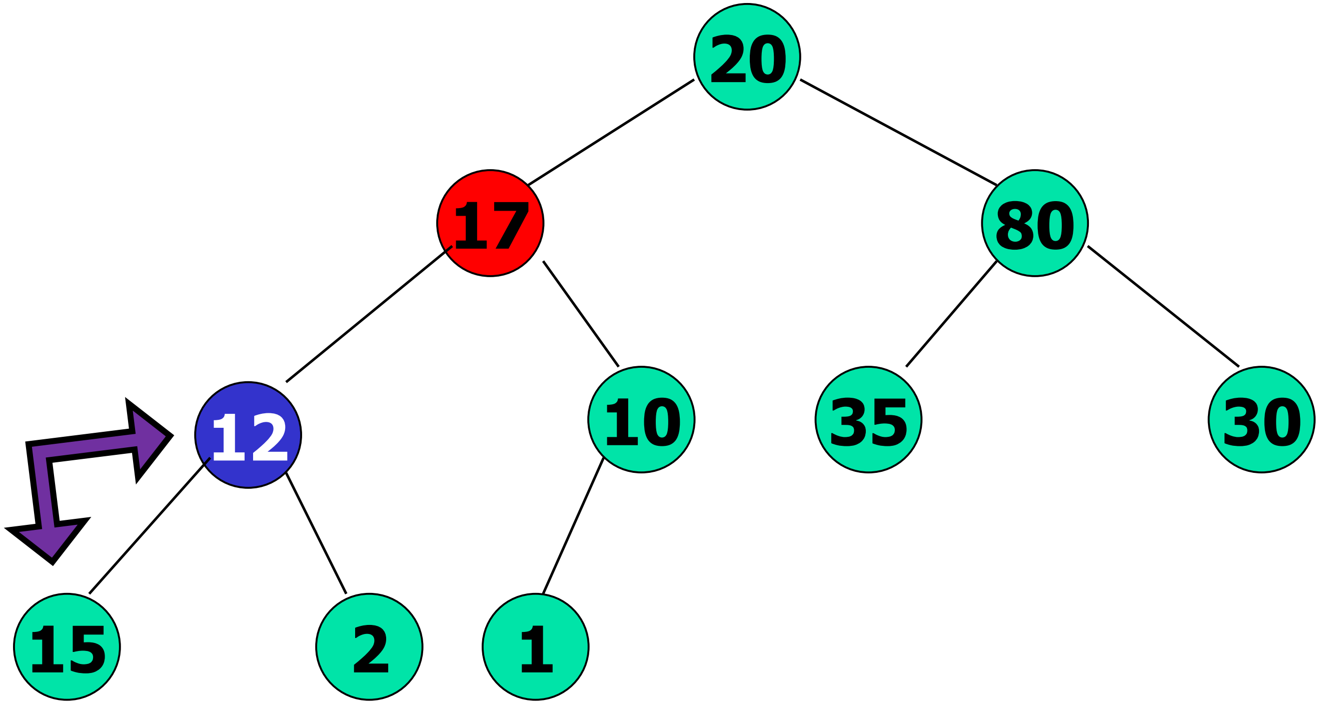
- Move to next lower array position
- Check the node with its child nodes

This is not OK!
Need a swap



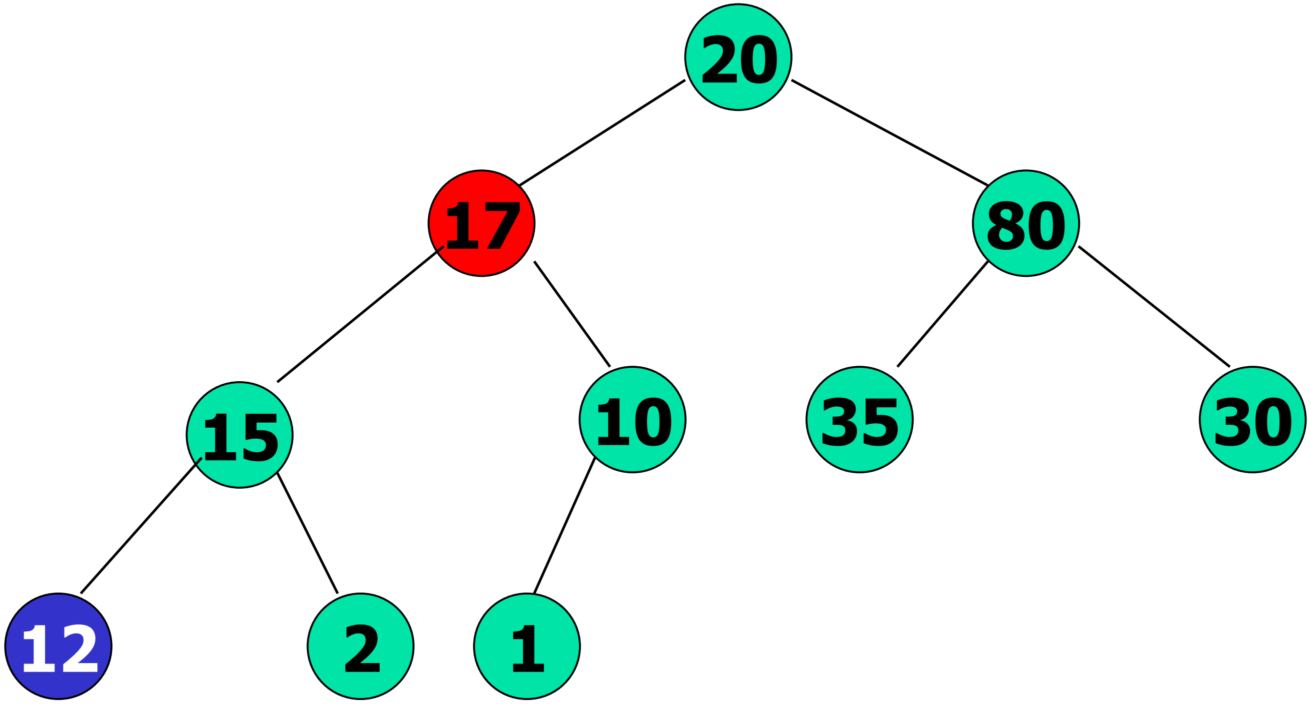
MaxHeap Initialization : heapify( ) [8/12]

- After replacing node with 17 and node with 12
- Now we need to find a new home for node with 12
  - Need to check the node of 12 with its child nodes



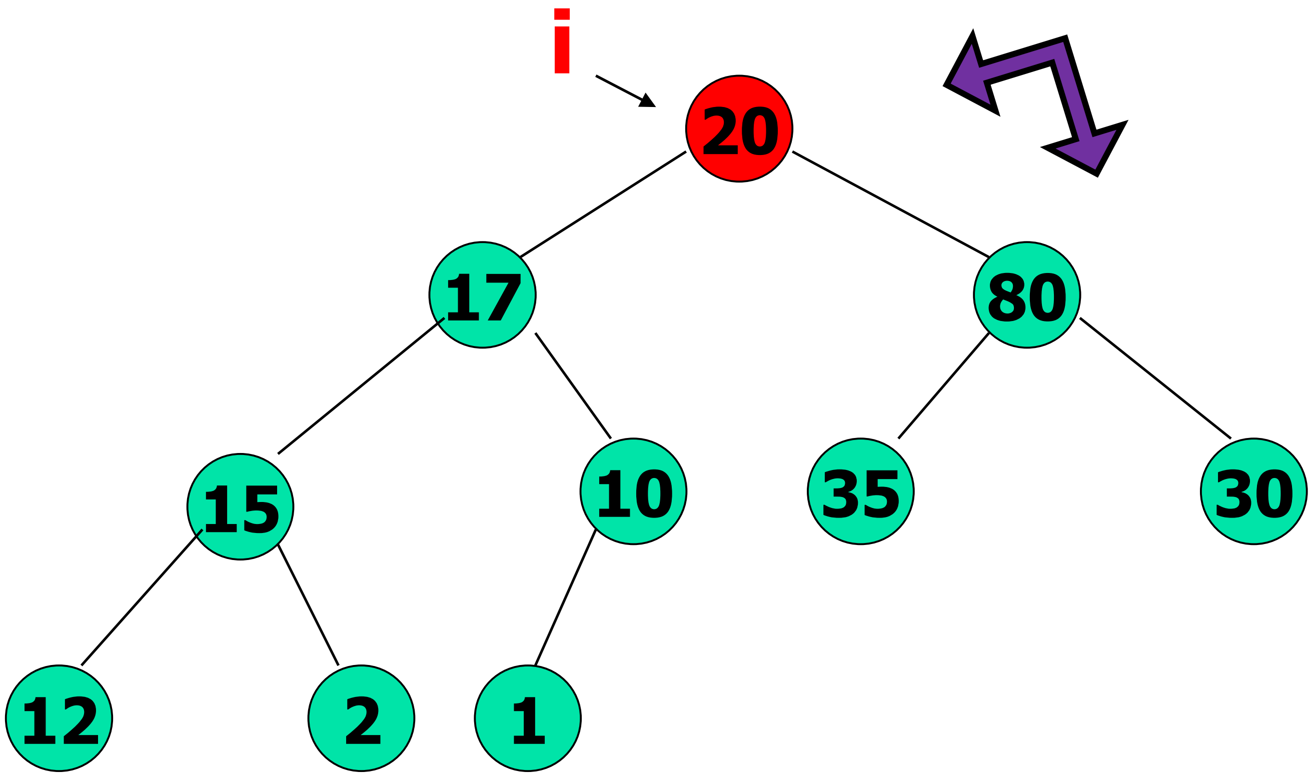
MaxHeap Initialization : heapify( ) [9/12]

- After replacing node with 15 with node with 12
  - The new position for the node 12 is OK



MaxHeap Initialization : heapify( ) [10/12]

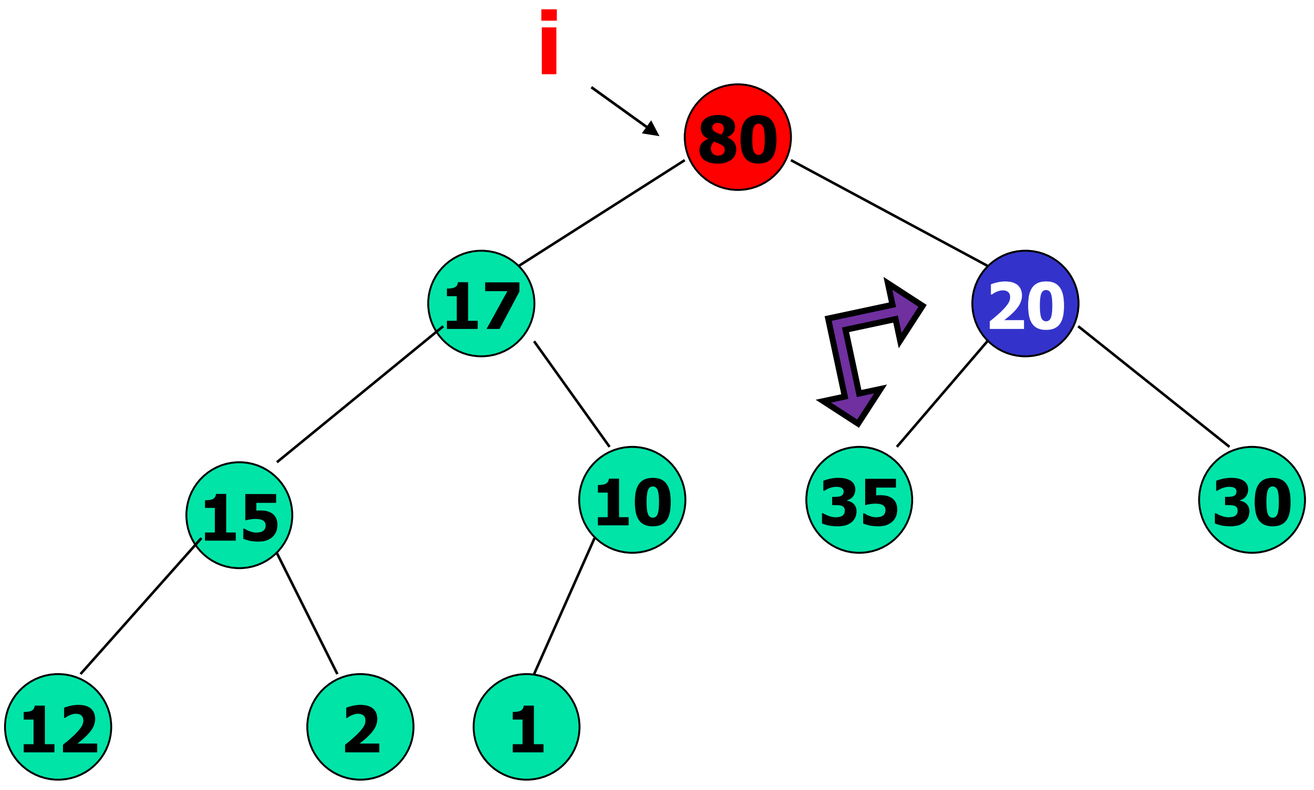
- Need to check if the node “20” is OK with the root position
  - Check the node with its child nodes



This is not OK!
Need a swap

MaxHeap Initialization : heapify( ) [11/12]

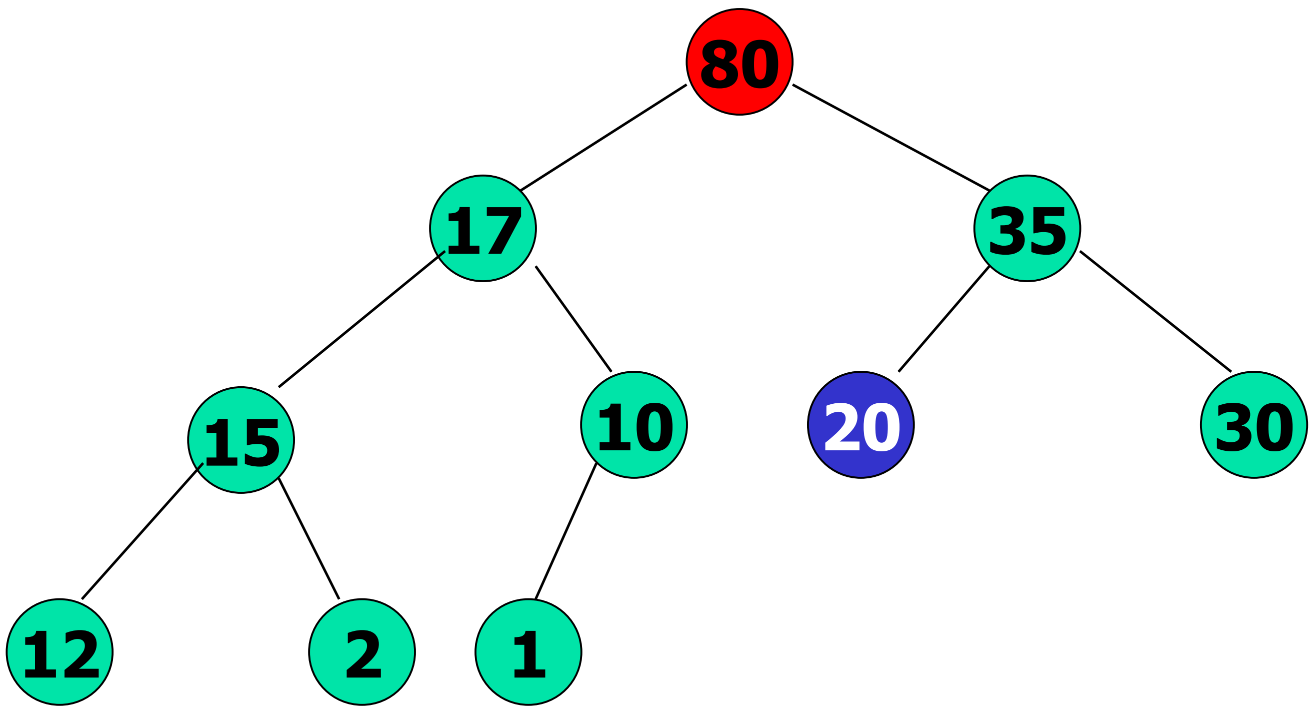
- We need to check whether the node of 20 is fine
  - Check the node with its child nodes



This is not OK!
Need a swap

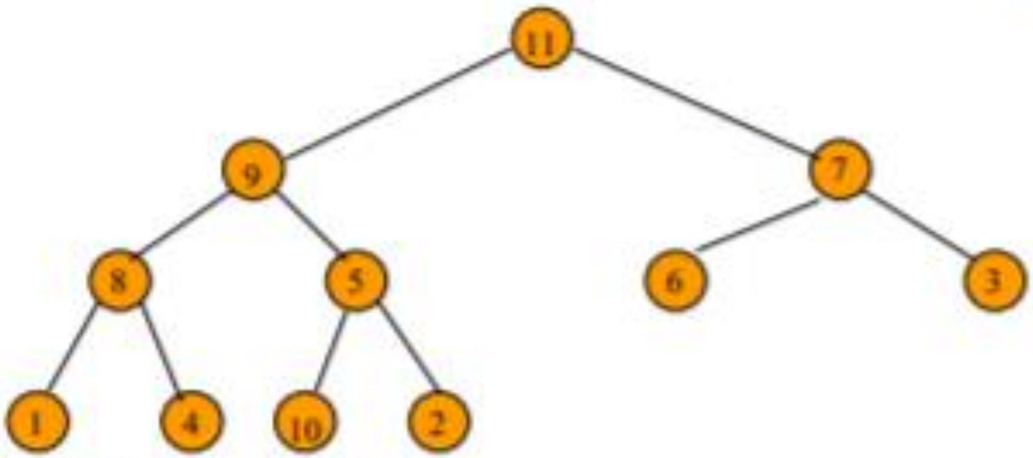
MaxHeap Initialization : heapify( ) [12/12]

- Now the tree is the max heap!
- We are done!



Complexity of Heap Initialization

Time Complexity



Height of heap = h.

Number of subtrees with root at level j is <= 2^{j-1}.

Time for each subtree is O(h-j+1).

Complexity

Time for level j subtrees is <= $2^{j-1}(h-j+1) = t(j)$.

Total time is $t(1) + t(2) + \ldots + t(h-1) = O(n)$.

Heap Initialization에 **Comparison**의 횟수가 **N**보다 작다**!!**

Binary Search Tree VS. MaxHeap Tree

	Binary Search Tree	MaxHeap Tree
Tree Initialization	O(n * log n) Sorting is required	O(log n) Sorting is not required
Insert an item	O(log n)	O(log n)
Delete an item	O(log n)	O(log n)
Sorting from Tree	Free	O(n * log n)

아래상태의 **queue**를 유지하면서 **max priority**값을
retrieve하는것보다 **MaxHeap Tree**를 만들면 경제적**!**

15	**35**	**65**	**20**	**17**	**80**	**12**	**45**	**2**	**4**

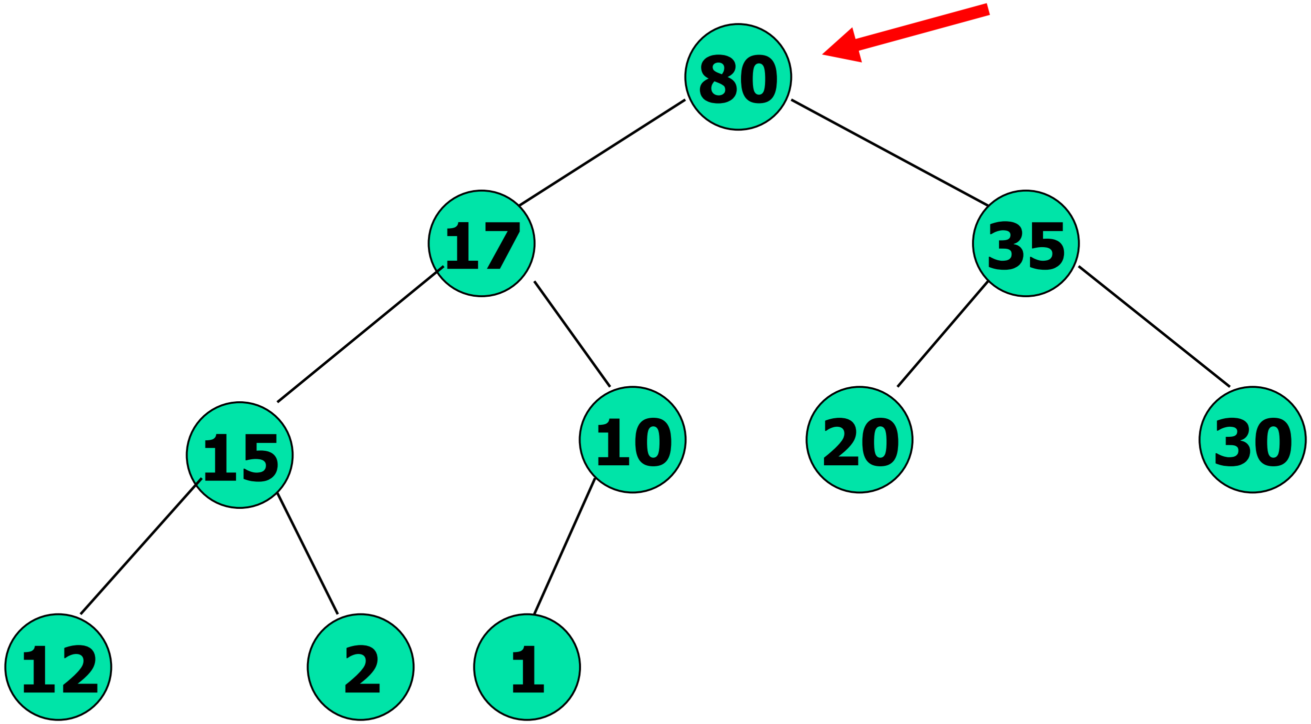
Sorting이 필요없고 **Max Priority** 값만 사용하는 상황이라면
Binary Search Tree보다 **MaxHeap Tree**를 만드는것이 경제적**!**

(22) Stack and Queue Applications

- Stack Application
  - Parenthesis Matching
  - Tower of Hanoi
- Priority Queue
  - Priority Queue using Heap
  - Heap Sort

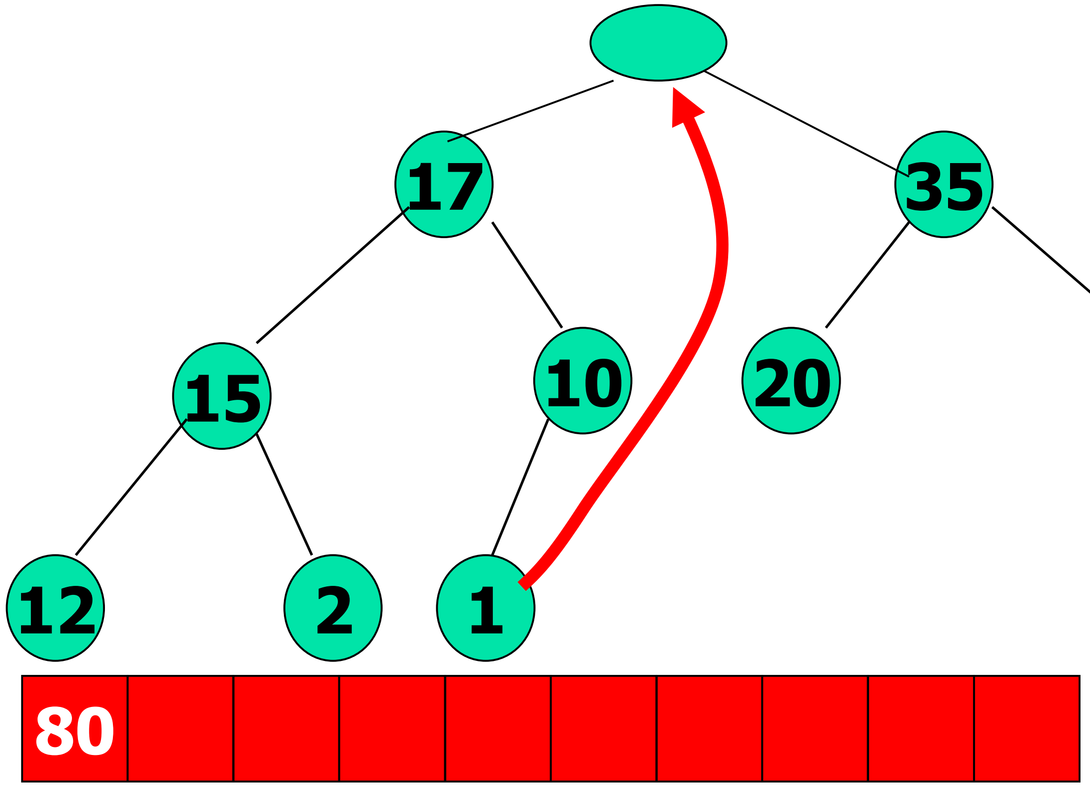
Heap Sort Example [1/20]

- Suppose we have a list of unsorted numbers
- Initialize (heapify( )) the max heap with input data
- Then, the sorting loop begins with the max heap



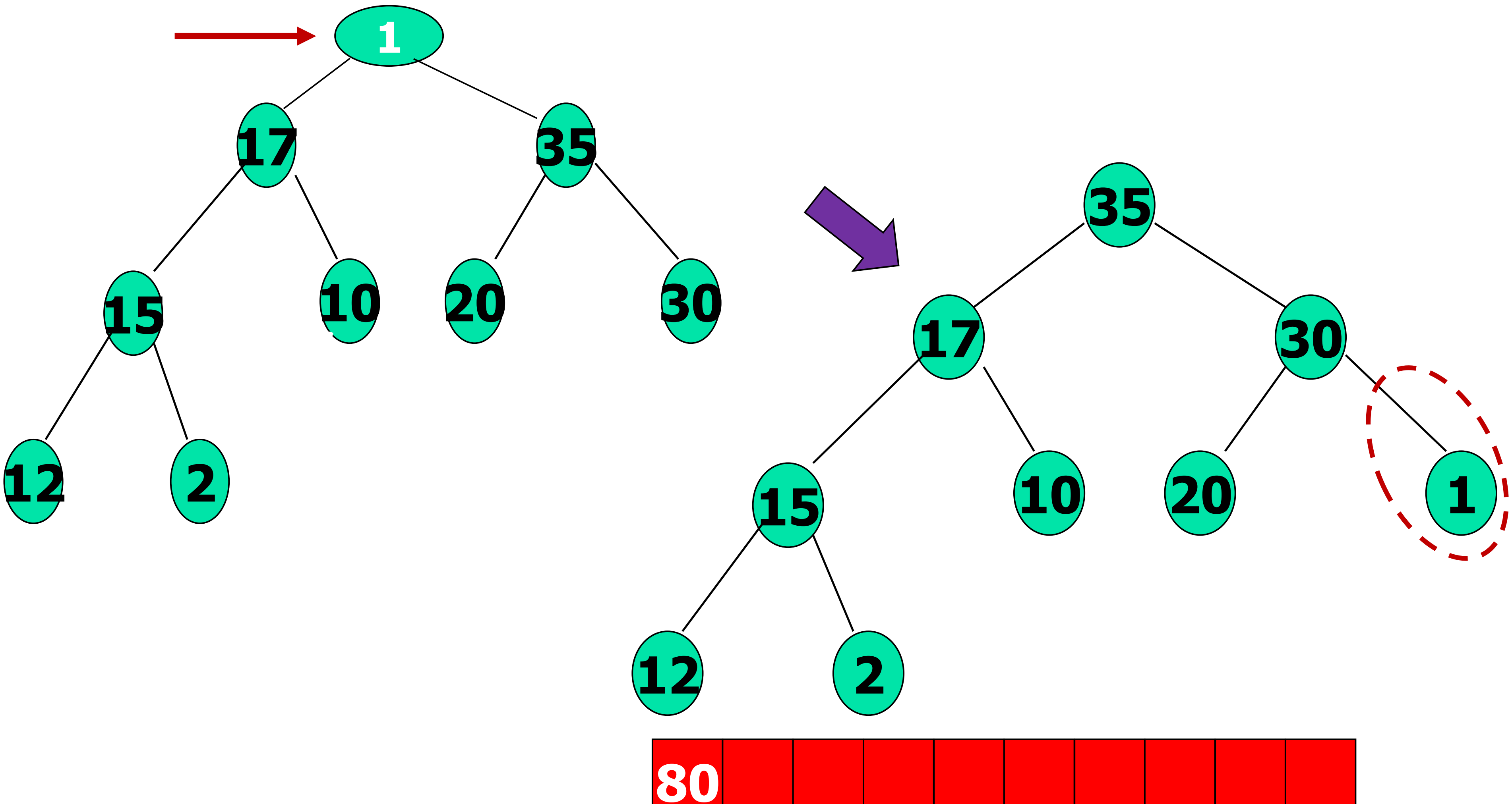
Heap Sort Example [2/20]

- RemoveMax( ) & Try to move the last element “1” to the root



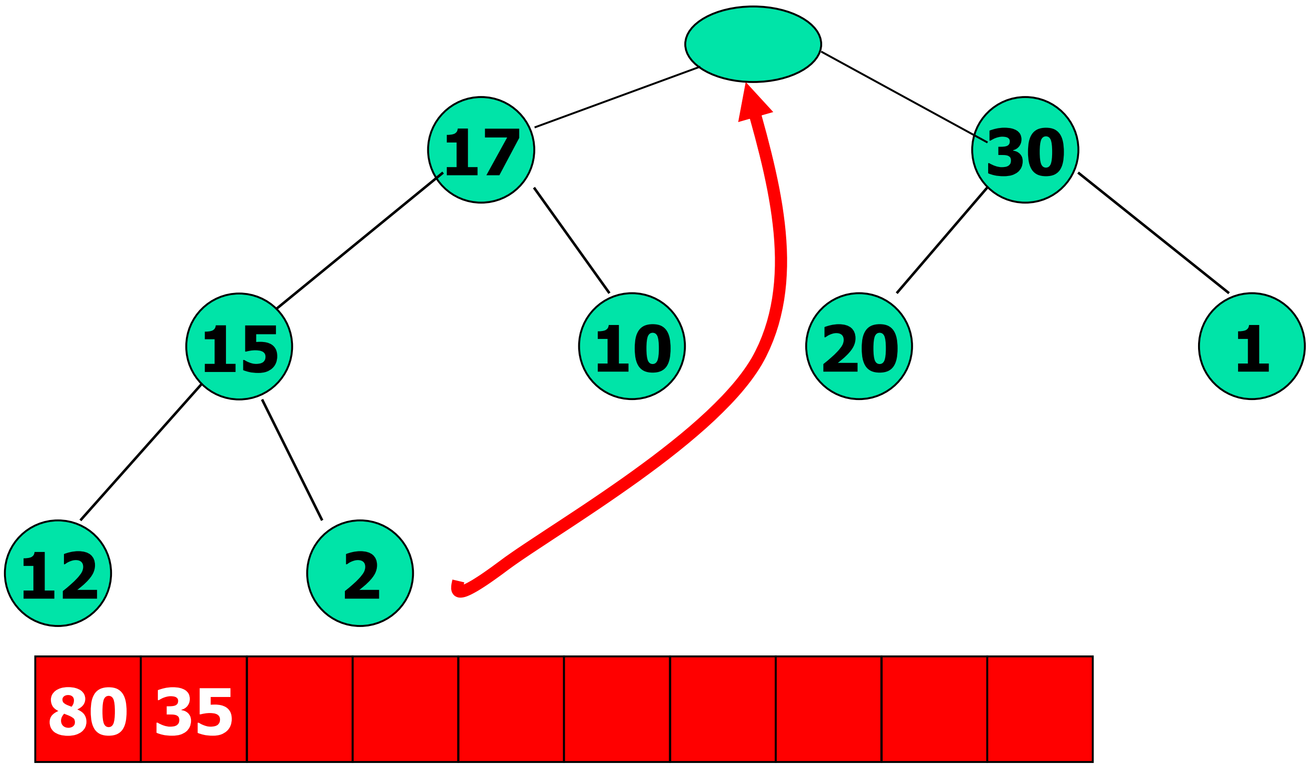
Heap Sort Example [3/20]

- Reheapify( ): Meld root.leftChild and root.rightChild
  - Find the correct home for the root item “1”



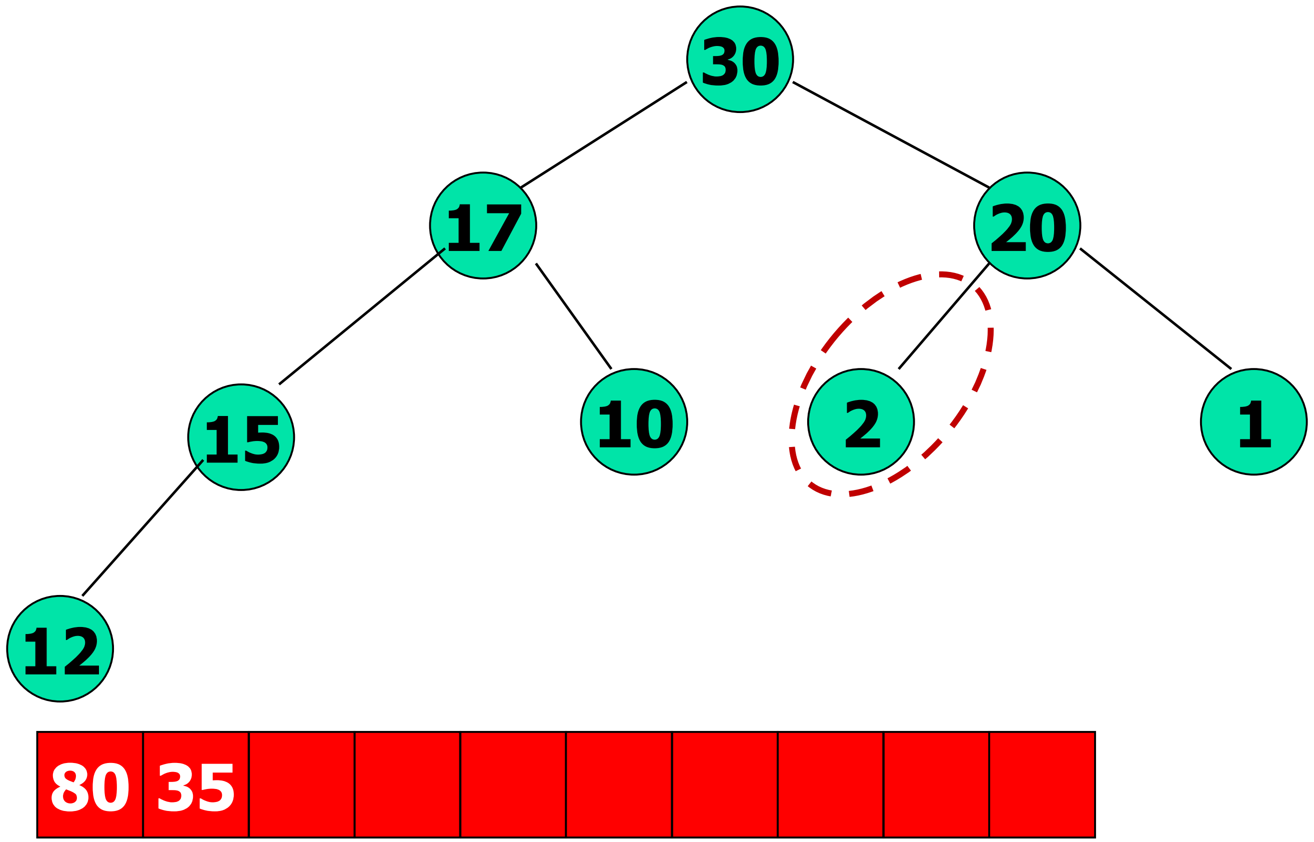
Heap Sort Example [4/20]

- RemoveMax( ) & Try to move the last element “2” to the root



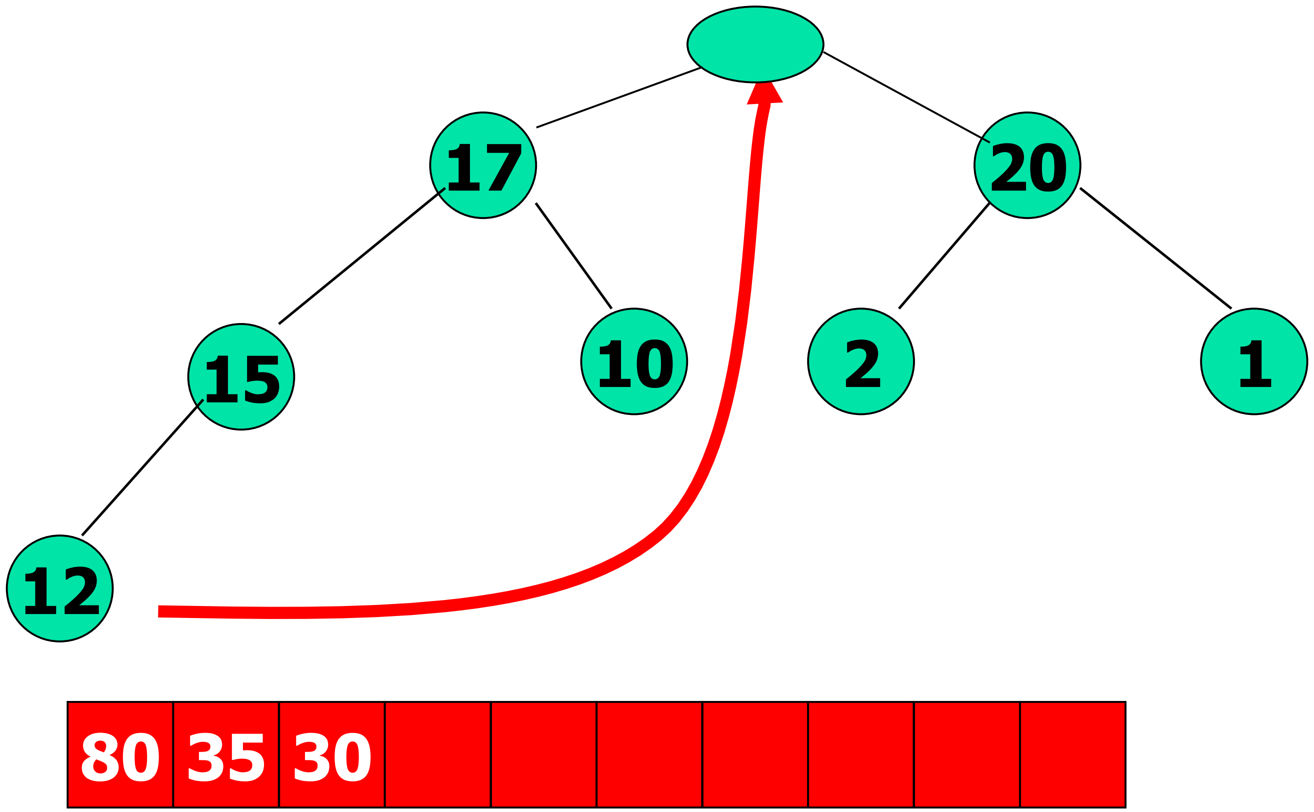
Heap Sort Example [5/20]

- Reheapify( ): Meld root.leftChild and root.rightChild
  - Find the correct home for the root item "2"



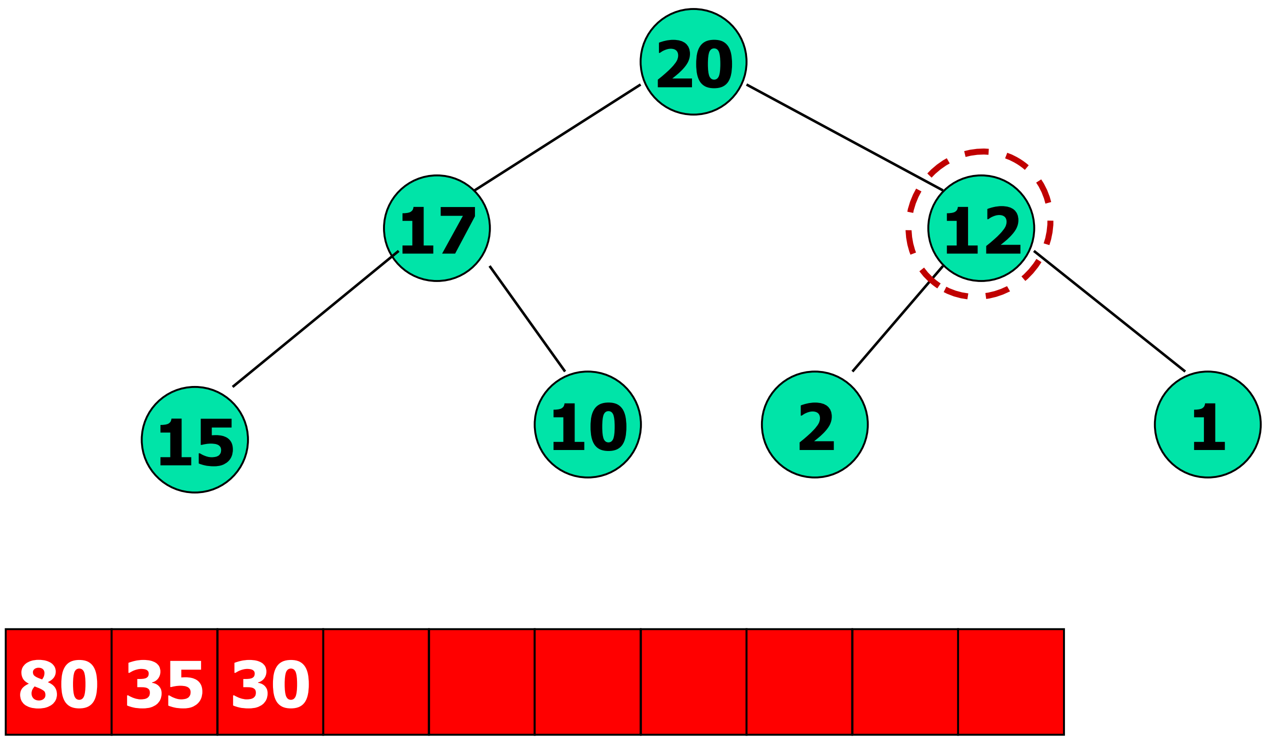
Heap Sort Example [6/20]

- RemoveMax( ) & Try to move the last element "12" to the root



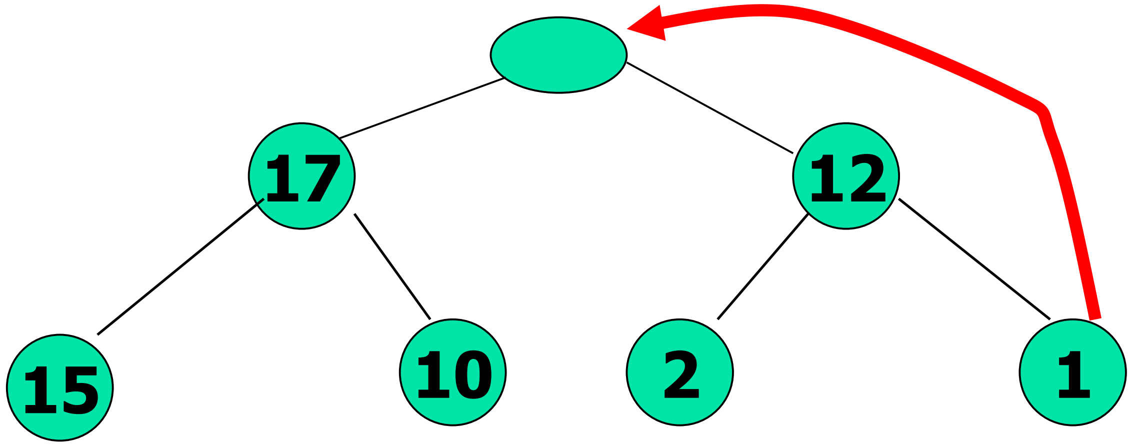
Heap Sort Example [7/20]

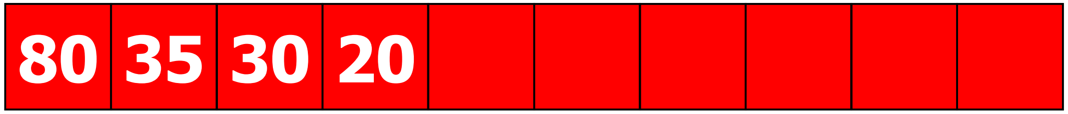
- Reheapify( ) : Meld root.leftChild and root.rightChild
  - Find the correct home for the root item “12”



Heap Sort Example [8/20]

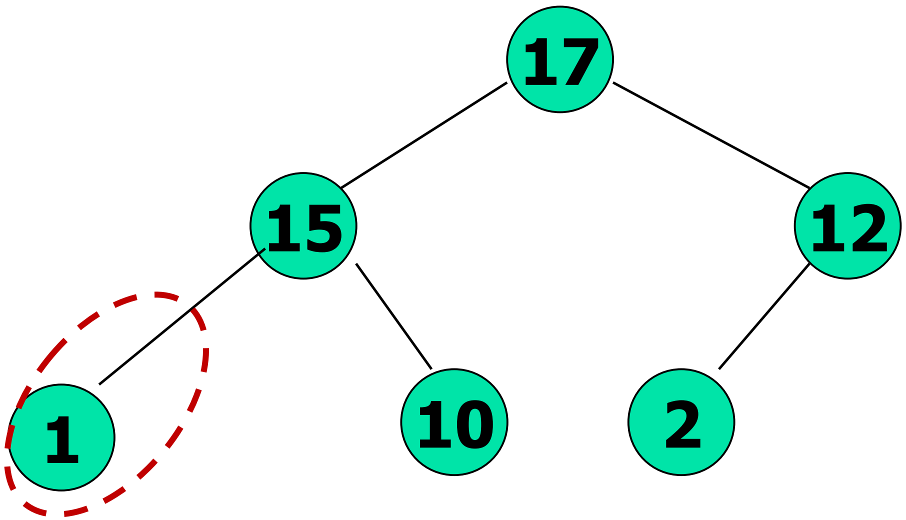
- RemoveMax( ) & Try to move the last element "1" to the root





Heap Sort Example [9/20]

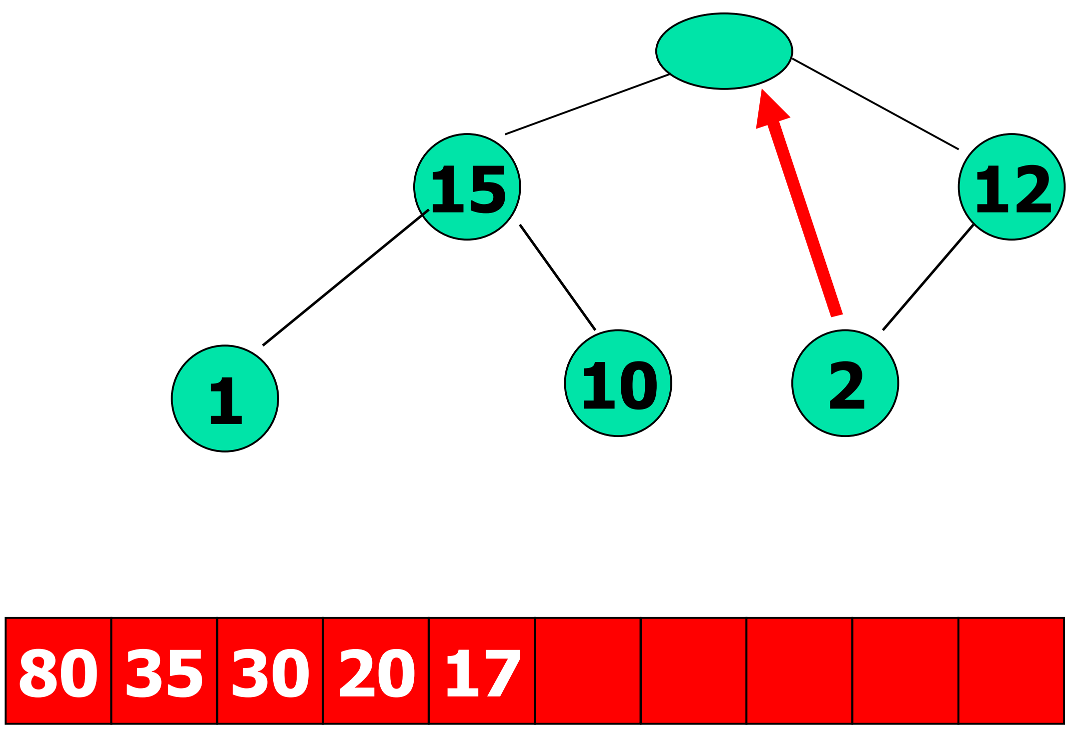
- Reheapify( ) : Meld root.leftChild and root.rightChild
  - Find the correct home for the root item “1”



80	35	30	20						

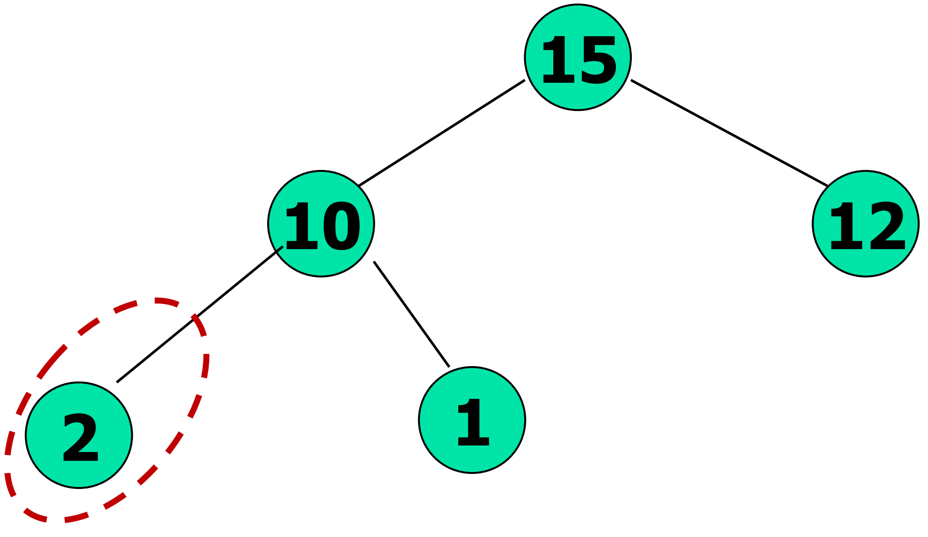
Heap Sort Example [10/20]

- RemoveMax( ) & Try to move the last element “2” to the root



Heap Sort Example [11/20]

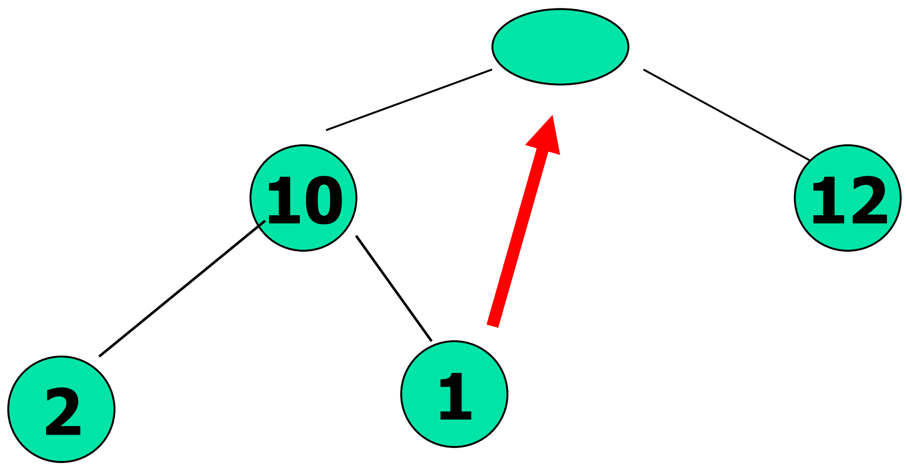
- Reheapify( ) : Meld root.leftChild and root.rightChild
  - Find the correct home for the root item “2”



80	35	30	20	17					

Heap Sort Example [12/20]

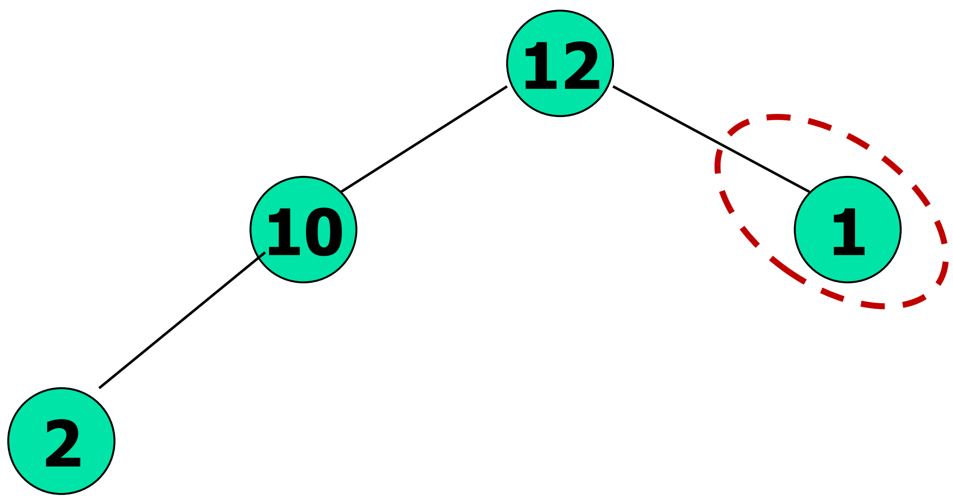
- RemoveMax( ) & Try to move the last element "1" to the root



80	35	30	20	17	15				

Heap Sort Example [13/20]

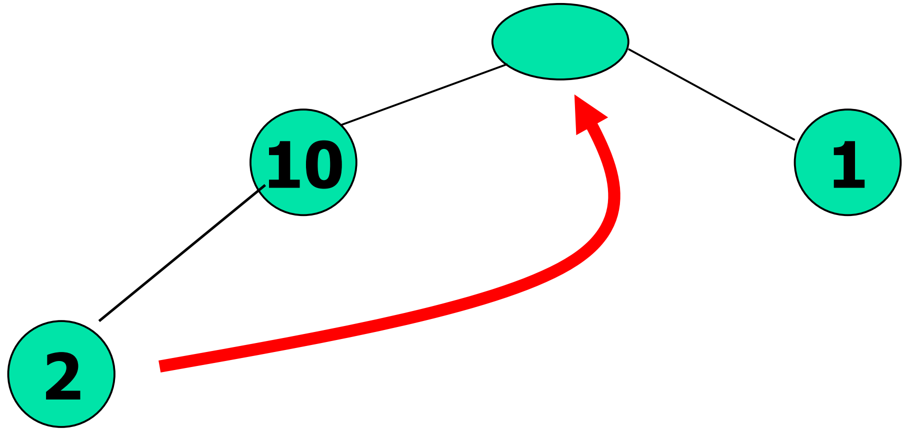
- Reheapify( ) : Meld root.leftChild and root.rightChild
  - Find the correct home for the root item “1”



80	35	30	20	17	15				

Heap Sort Example [14/20]

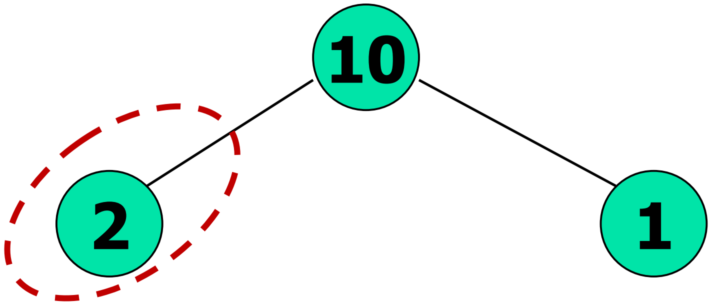
- RemoveMax( ) & Try to move the last element “2” to the root



80	35	30	20	17	15	12			

Heap Sort Example [15/20]

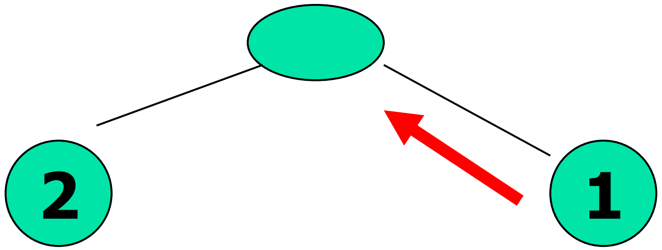
- Reheapify( ) : Meld root.leftChild and root.rightChild
  - Find the correct home for the root item “2”



80	35	30	20	17	15	12			

Heap Sort Example [16/20]

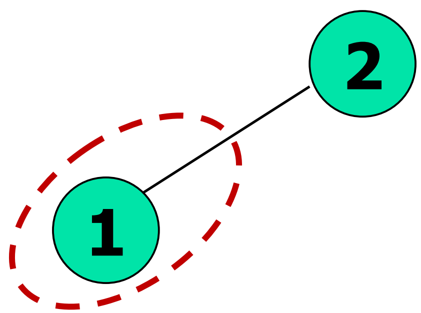
- RemoveMax( ) & Try to move the last element "1" to the root



80	35	30	20	17	15	12	10		

Heap Sort Example [17/20]

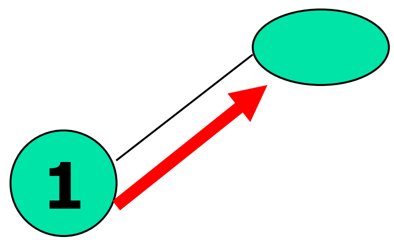
- Reheapify( ) : Meld root.leftChild and root.rightChild
  - Find the correct home for the root item “1”



80	35	30	20	17	15	12	10		

Heap Sort Example [18/20]

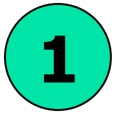
- RemoveMax( ) & move the last element "1" to the root



80	35	30	20	17	15	12	10	2	

Heap Sort Example [19/20]

- Reheapify( ) : Meld root.leftChild and root.rightChild
  - Find the correct home for the root item “1”



80	35	30	20	17	15	12	10	2	

Heap Sort Example [20/20]

- RemoveMax( ) & we are done!

80	**35**	**30**	**20**	**17**	**15**	**12**	**10**	**2**	**1**

- Complexity of Heap Sort : O(n * log n)
  - Heap Initialization : O(n)
  - Deletion : O(log n)
  - Sort ➔ deletion n times ➔ O(n * log n)

- Quick Sort or Merge Sort 와의 비교
  - Heap Initialization의 O(n) 계산량은 더 필요
  - 전체 soring 된 item이 필요없고, 몇 개의 가장 큰값만을 쓰는 경우에는 Heap Sort가 Quick Sort 나 Merge Sort 보다 유리!