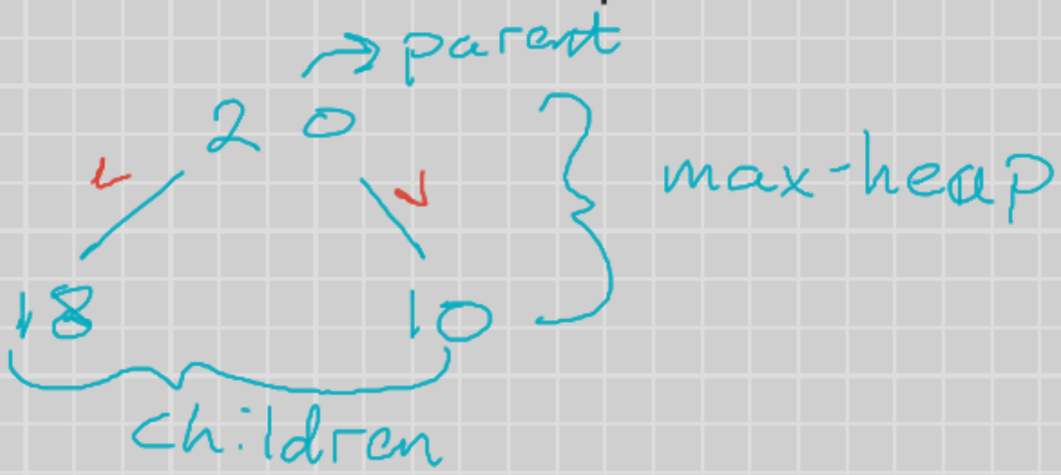
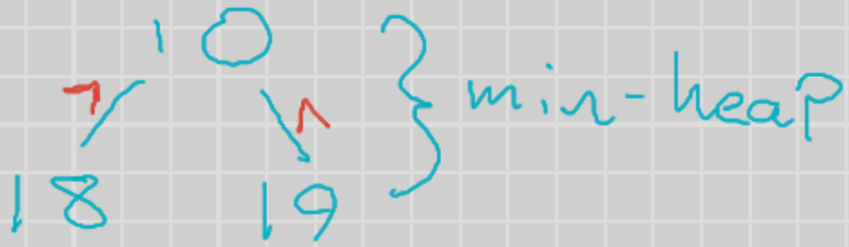
Heaps

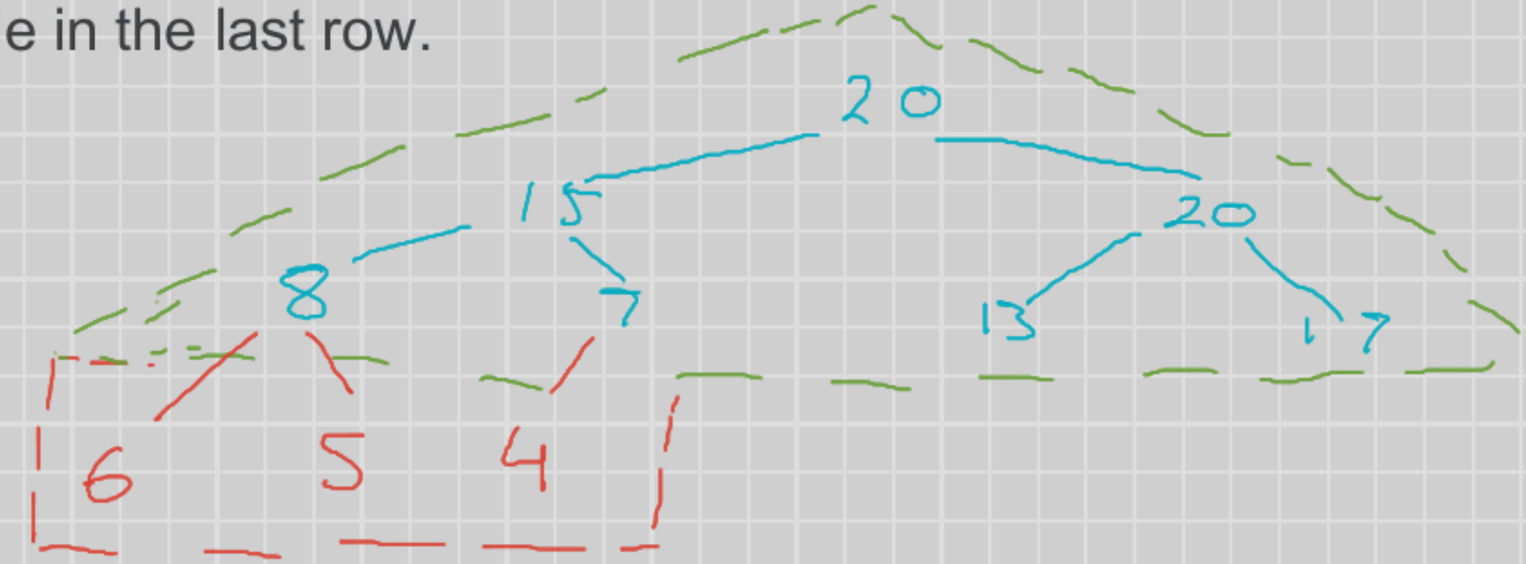
- A heap is a tree data structure
- For a tree to be a heap it must satisfy the following two properties:

1) Heap Property: The value of parent node must have some relationship to its children:
   a) If the heap is a MAX-HEAP then the value store in a parent node must be greater than its children.
   b) If the heap is a MIN-HEAP then the value store in a parent node must be less than its children.





2) Shape property: the tree must form a perfect triangle or a perfect triangle with a left-aligned rectangle in the last row.



* One application of the heap is a priority queue.

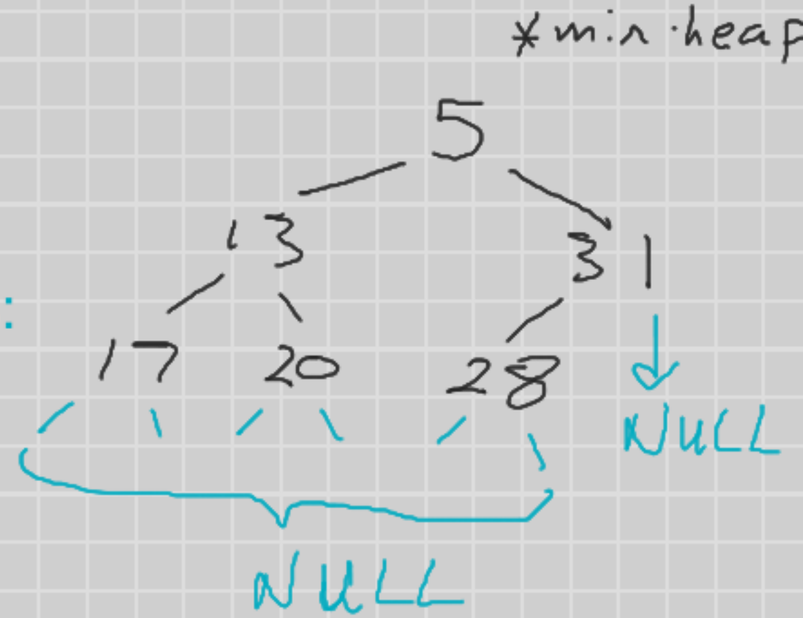
- A binary heap has nodes with a max of two children, it can be viewed as a nearly complete binary tree. It is a data structure usually stored as an array, each node corresponds to an element in the array.

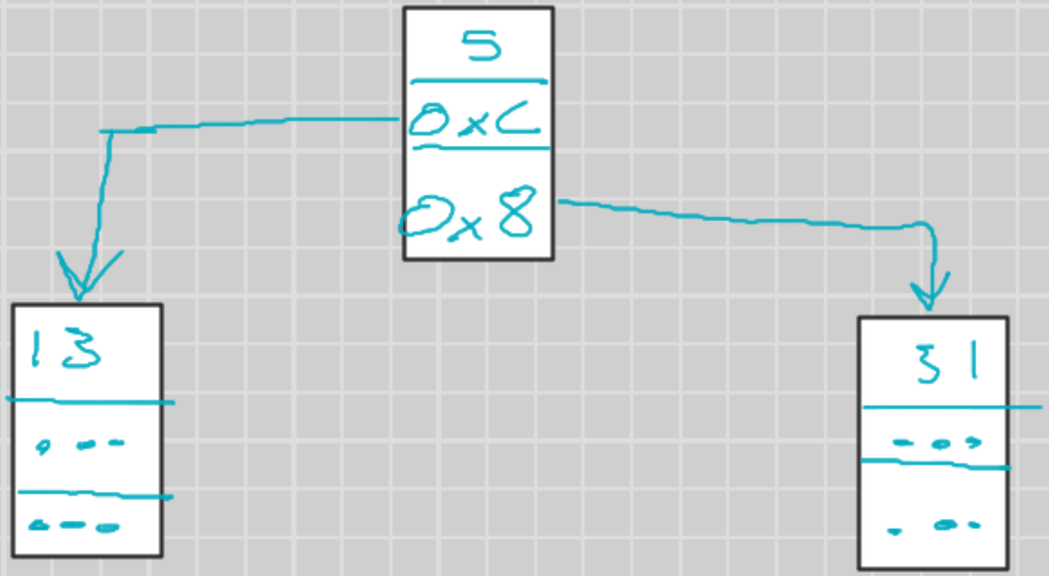
- We can have heaps with any number of n-branches, ternary, quarternary, n-ary, etc. They must a max of n branches.

Heap: Implementation

- We can implement heaps using memory pointers or arrays.

- If we use pointers, then every node in a heap will store three fields:
1) Data: the value of the node
2) Left pointer: the memory address of the left child
3) Right pointer: the memory address of the right child



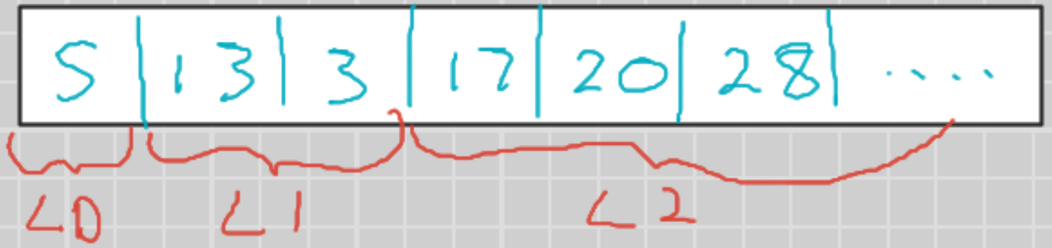


This type of representation is known as EXPLICIT REPRESENTATION, because the memory addresses are stored in the parent nodes.

- We can use an array to represent a heap.

- When we use an array to represent a heap (binary heap) then the root node is stored at index 0 in the array (level 0 of the heap). Level 1 nodes are located at indexes 1&2....and so on.

heap =



This is an IMPLICIT REPRESENTATION, becaus the memory addresses are implicitly given by their position in the array.

In general the nodes at level k will need (2^k) positions to store all values, and are placed starting at index [(2^k) - 1] to index [2^(k+1) - 2].

L0: $2^0 - 1 = 0$ → index

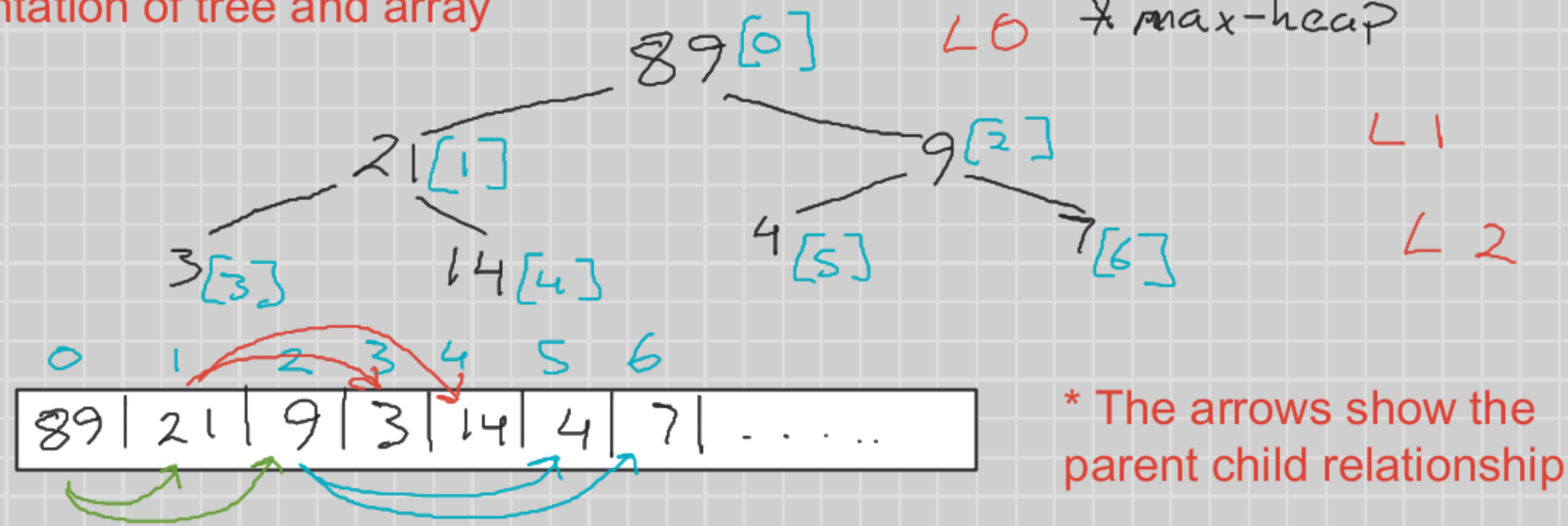
L1: 13, 3 to store

start at: $2^k - 1 \Rightarrow 2^1 - 1 = 1$, heap[1] ← 13

finish at: $2^{k+1} - 2 \Rightarrow 2^2 - 2 = 2$, heap[2] ← 3

NOTE that the most common way of string heaps is using an array.

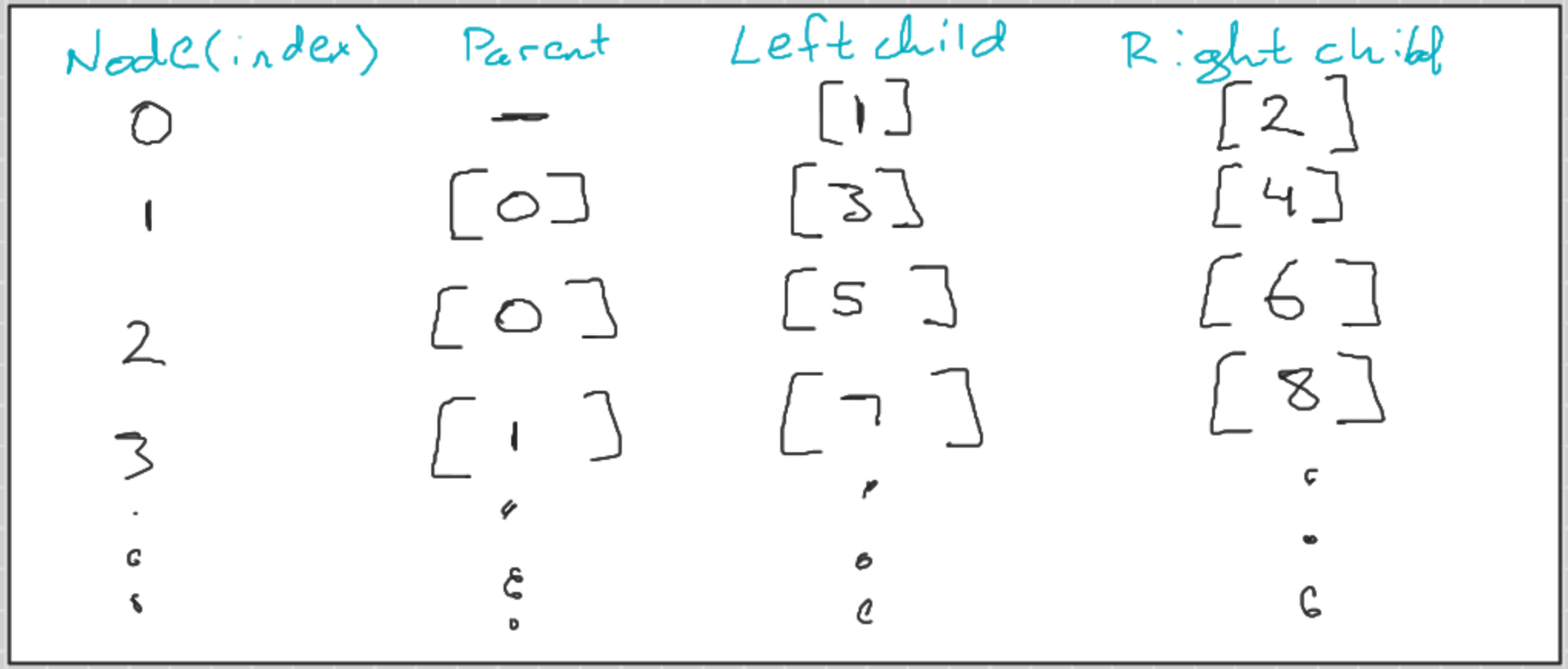
1:1 representation of tree and array



* The arrows show the parent child relationship

How to find Parent, Left, and Right children given any index (i): Algebraic Expressions:

1) Parent: floor[(i - 1)/2]
2) Left child: 2i+1
3) Right child: 2i+2



1) Parent: $\lfloor (i - 1)/2 \rfloor$
2) Left child: 2i+1
3) Right child: 2i+2

function Parent(i)
return floor[(i - 1)/2]

funciton Left(i)
return 2i+1

function Right(i)
return 2i+2

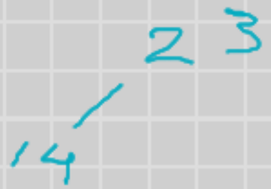
Heap: Insert (element by element)

- The insert operation allows us to insert elements into a heap to build it from scratch. It will insert elements such that the 1) Heap property and 2) Shapre property are preserved.

EXAMPLE: Let's create an empty max-heap and insert 23

- We initialize a variable called Size to track the number of elements in the heap, this helps us location the next available position. Size <- -1.

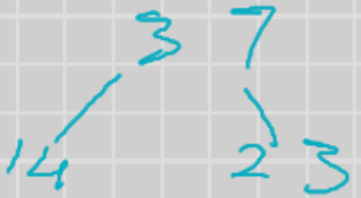
Let's add #14:
heap = [23, 14, , , ,] , Size = 2.



- After adding 14, we must check that the heap property is satisified. Check is 23 is greater than 14. Yes it is, so the heap property is satisfied. Because we are building a complete binary tree we know that the shape property is satisfied.

Next we'll #37:
heap = [23, 14, 37, , ,], Size = 3

- Check if the heap property (max-heap) is satisified. Is 23 greater than 37? NO. We must swap the root and the new node added.
heap = [37, 14, 23, , ,], Size = 3



Next we'll add #42

heap = [37, 14, 23, 42, ,], Size = 4

Check if the heap property (max-heap) is satisified. Is 42 greater than 14? NO. We must swap the root and the new node added.
heap = [37, 42, 23, 14, ,]

Check if the heap property (max-heap) is satisified. Is 42 greater than its parent, 37? NO. We must swap 42 and 37.

heap = [42, 37, 23, 14, ,], Size = 4

How do we know when to stop? We stop when we reach the root node, which has no parent.

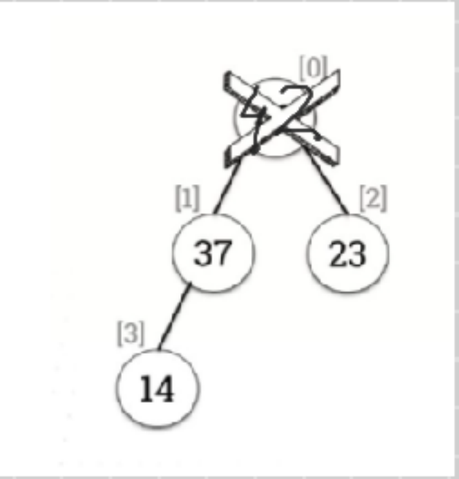
```
function INSERT(heapArray, val)
    pos = heapArray.length
    heapArray[pos]

    while (pos > 0 && heapArray[Parent(pos)] < heapArray[pos])
        SWAP(heapArray, heapArray[Parent(pos)] , heapArray[pos])
        pos = Parent(pos)
end function
```

Heap Insert: Deletion (extract Max)

- When nodes are extracted from a heap, they only get extracted from the root, unlike other trees.
- Since we are working with a max-heap, we will be removing that maximum value of the heap.
- The operation is called ExtractMax.

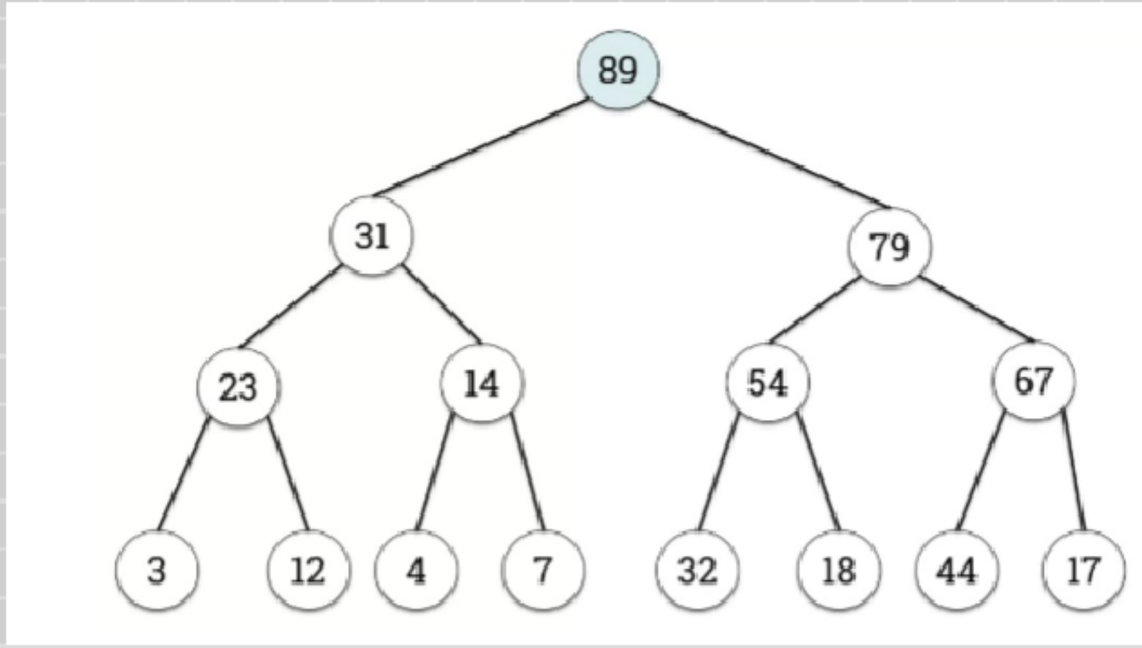
- It's possible to create an algorithm to remove any node from the heap, however, because the heap os not fully sorted, it would be computationally expensive.

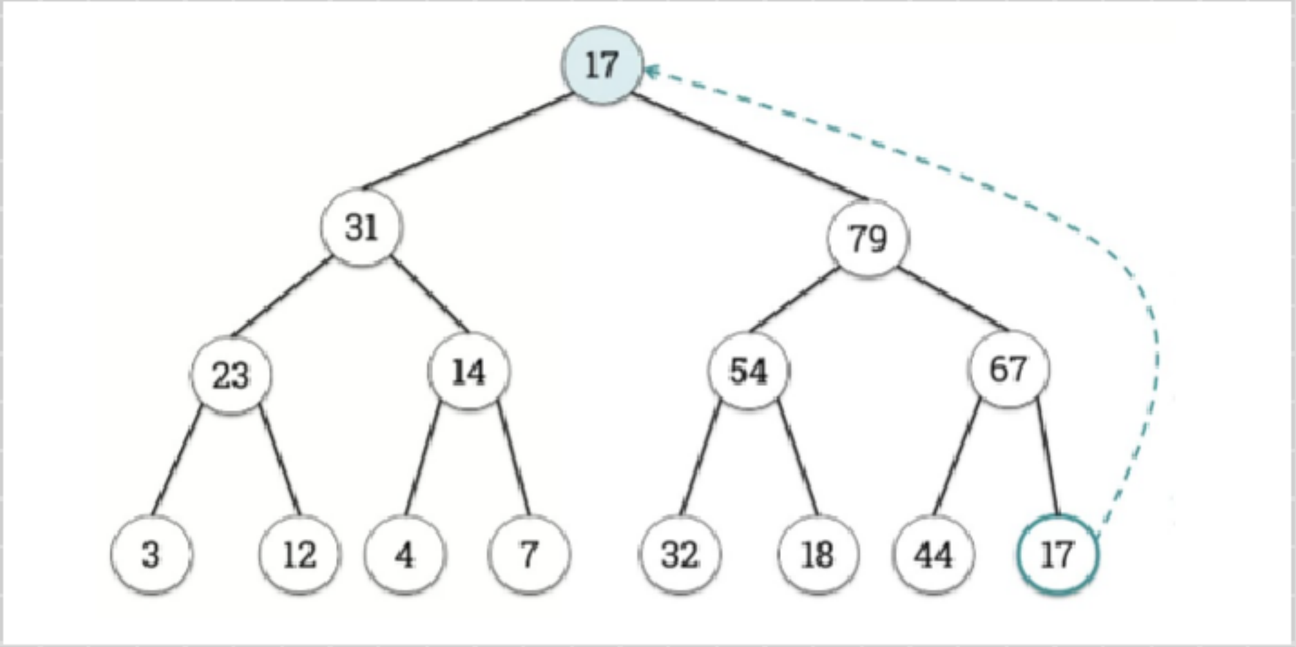
- We need to create an algorithm to extract the max value from the heap and preserve the heap (max-heap) property and the shape property.



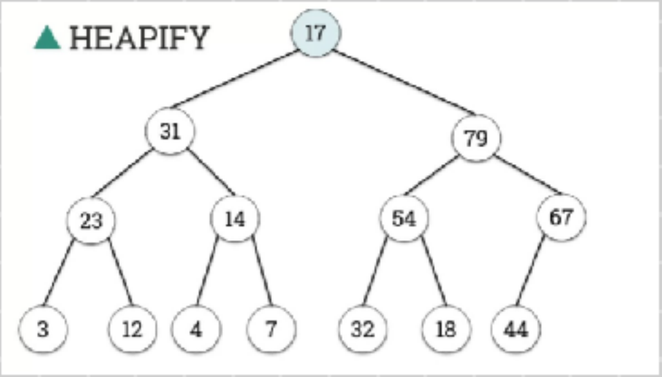
We want to remove 89, which is the largest value in the tree.

1) To delete this node (89), we will copy the value contained at the right-most, the last node inserted (17). Replace the root 89 with 17, and then delete 17 from the heap.

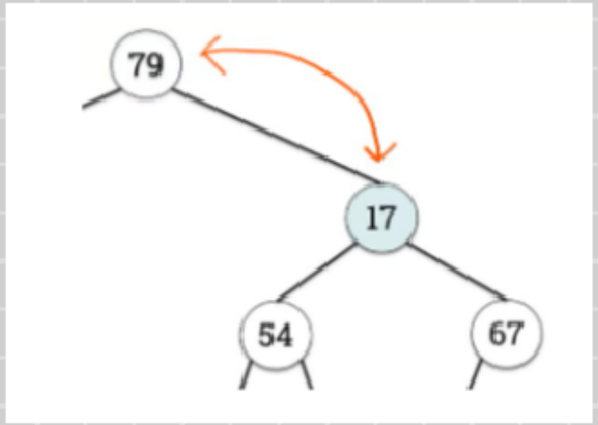


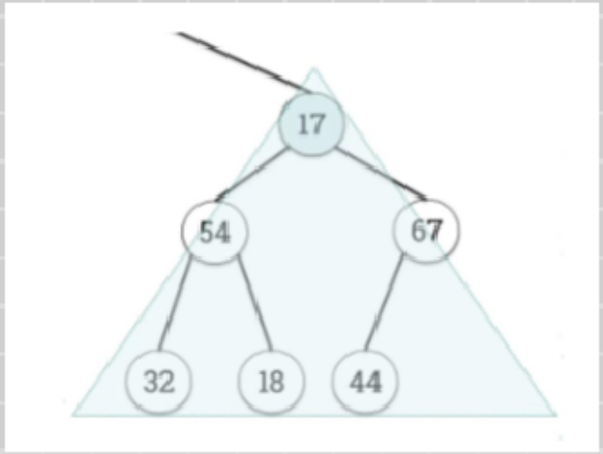


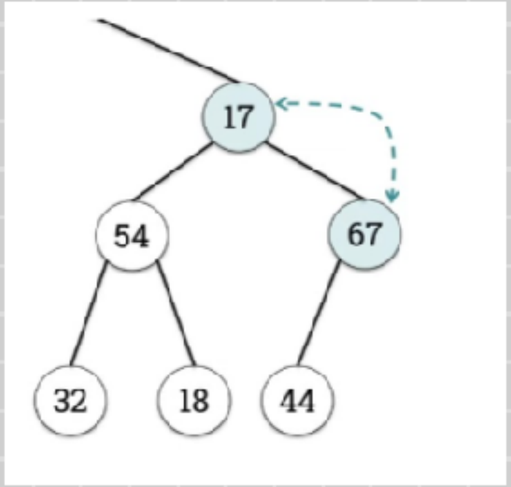
2) The heap doesn't comply with the max-heap property. We need to recover the heap property through a process known as HEAPIFY. We swap the root node with its largest child recursively.

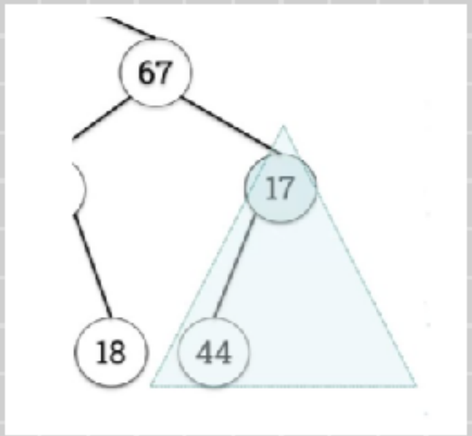


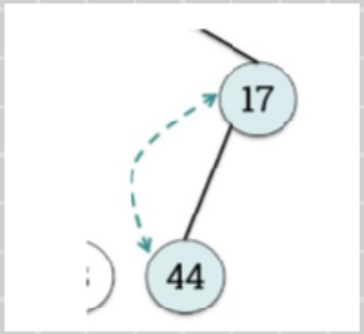
We will heapify, the right subtree recursively. 79 and 17 get swapped. Then 17 with 67. And finally 17 with 44.












```
function ExtractMax(heap)
    max = heap[0]
    heap[0] = heap[heap.length - 1]
    MaxHeapify(heap, 0)
end function
```

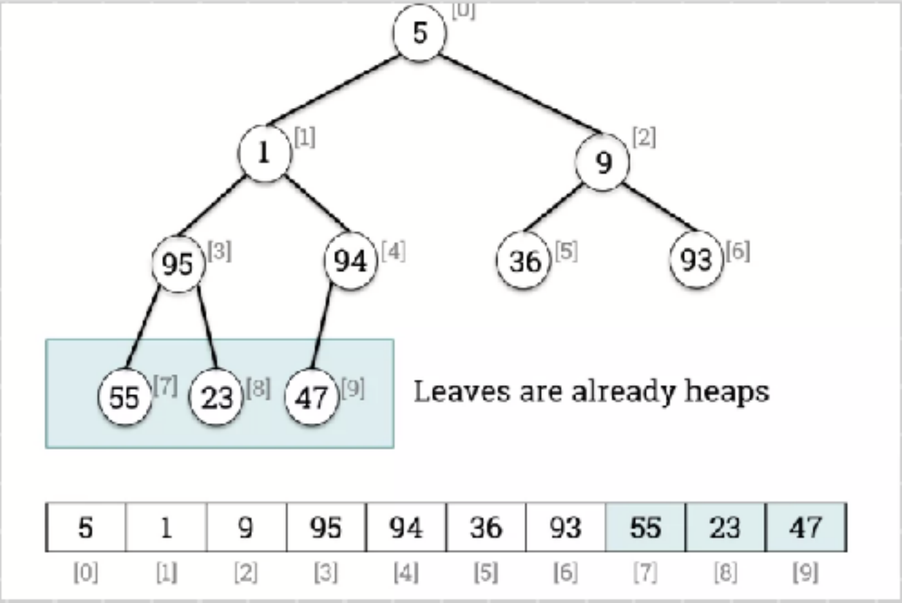
```
heap: heap array
root: value
function MaxHeapify(heap, root)
    largest = IndexOfLargest(heap, root) // finds the largest child of node passed in

    if (largest != root)
        SWAP(heap, heap[largest], heap[root]
        MaxHeapify(heap, largest)
end function

function IndexOfLargest(heap, root)
    left = LEFT(root)
    right = RIGHT(root)

    if (heap[root] > left && heap[root] > right)
        return root
    if (heap[left] > heap[right])
        return left
    return right
```

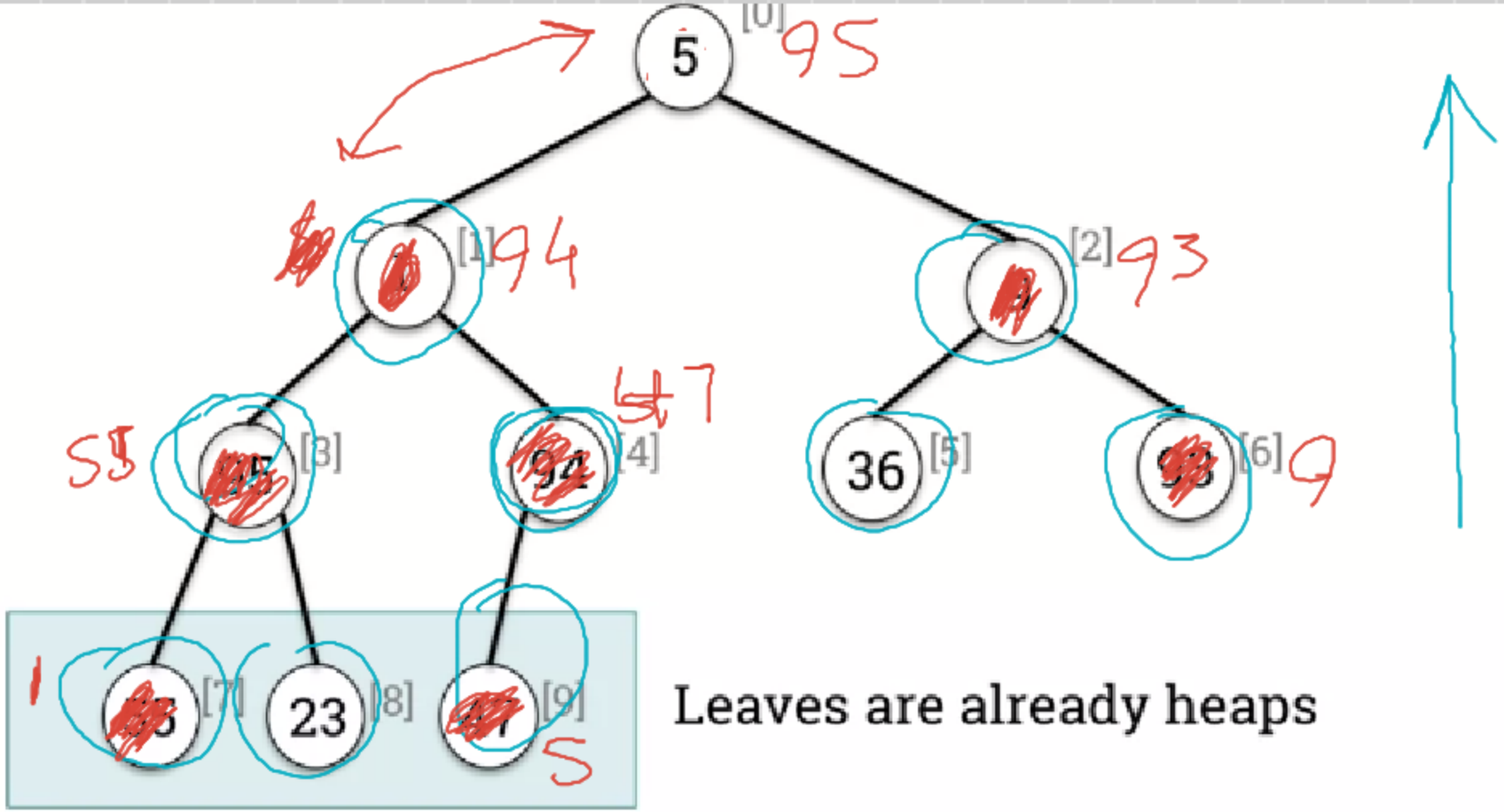
Heaps: Build In-Place

Note: There's an alternative to building a heap in-place, and that is Out-of-place using a second array. This algorithm would visit each element/node in an array, each time a node is visited, the MaxHeap operation is performed, each time pushing the number to a second array. We will focus on the In-place method of building a heap in place.

- With the In-place algorithm, we use a single array to create the maxHeap. We visit each element in the array, each time calling the MacHeapify function.
- The leaves of the binary tree represented by the input array, are already heaps, so we don't need to call MaxHeapify on them. The number of leaves in a binary tree are roughly half the number of nodes. So this means that we would only need to call MaxHeapify on the first FLOOR(heapSize/2) nodes.

We must heapify from the bottom up. In this example that means we would start at node 4/index 4.





5	1	9	95	94	36	93	55	23	47
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

95	94	93	55	47	36	9	1	23	5

Pseudocode for building maxHeap in-place

```
function BUILD-MAX-HEAP(A)
        heap_size=A.length
        for FLOOR(heap_size/2) < j ≤ 0
                MAX-HEAPIFY(A,j)
        end for
end function
```

1. A max-heap is built in-place from the array $[0, 22, 28, 69, 31, 29, 51, 24]$.

   What are the final contents of the newly-built heap? [69, 31, 51, 24, 0, 29, 28, 22]

2. A max-heap is built in-place from the array $[10, 63, 64, 28, 54, 22, 29, 38]$.

   What are the final contents of the newly-built heap?

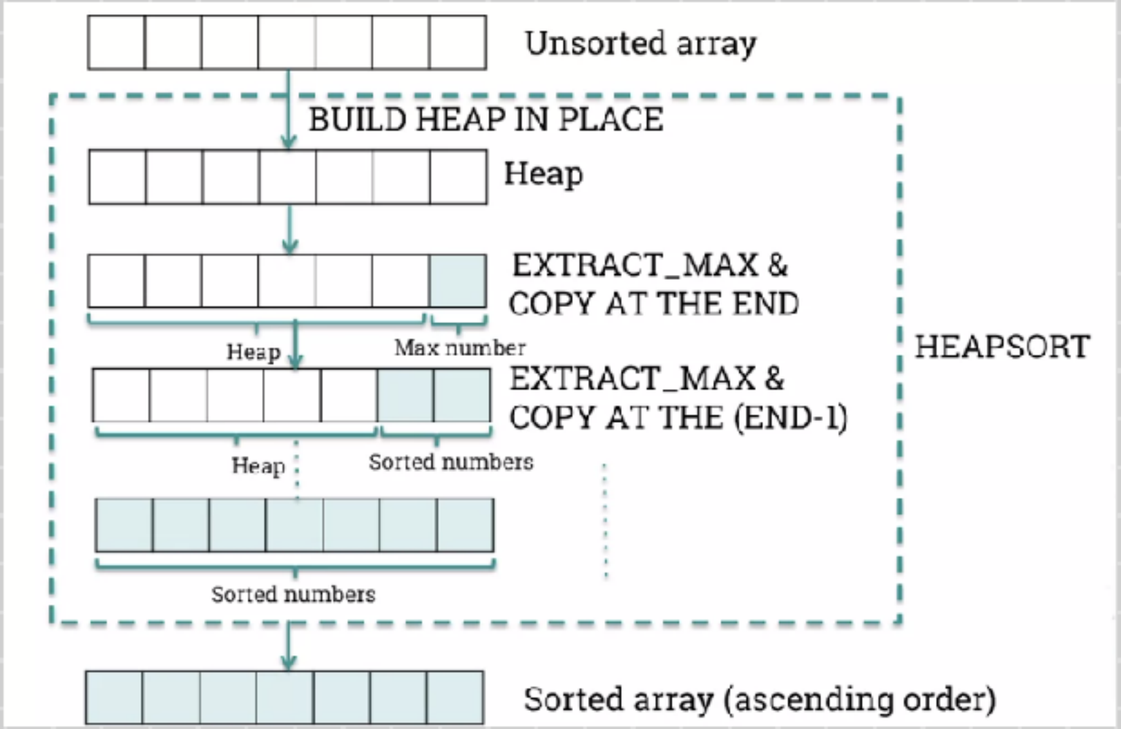
3. A max-heap is built in-place from the array $[14, 21, 48, 36, 30, 71, 52, 15]$.

   What are the final contents of the newly-built heap?

HeapSort

- is a sorting algorithm that takes as input an unsorted array and output a sorted array.
- the diagram below shows how the heapSort algo works:

1) From the unsorted array, a heap is built in-place
2) Extract-max is called and placed at the end of the the array, this happens repeatedly until all elements are sorted.
3) Note that the left side of the array is the heap, before extractMax is called, and the right side of the array contains the sorted numbers.
4) A sorted array is output.



Pseudocode for heapsort

```
function Heapsort(array)
        BUILD_MAX_HEAP(array)
        while heap_size > 0
                i=heap_size-1
                array[i]=EXTRACT_MAX(array)
        end while
        return array
end function
```

1. As heapsort is executed, one part of the array is used by the heap and the other part is used to store the sorted numbers.

   Part way through heapsort the underlying array has contents $[57, 54, 35, 44, 10, 99]$. What is the current heap size? (i.e. the number of elements in the heap part)

   heap → sorted

   5

2. As heapsort is executed, one part of the array (left) is used by the heap and the other part (right) is used to store the sorted numbers.

   Part way through heapsort the underlying array has contents $[23, 17, 4, 14, 15, 1, 29]$. What is the current heap size? (i.e. the number of elements in the heap part)

   6

3. As heapsort is executed, one part of the array (left) is used by the heap and the other part (right) is used to store the sorted numbers.

   Part way through heapsort the underlying array has contents $[4, 1, 14, 15, 17, 23, 29]$. What is the current heap size? (i.e. the number of elements in the heap part)

   2

HeapSort: Time Complexity

```
function Heapsort(array)
        BUILD_MAX_HEAP(array)
        while heap_size > 0
                i=heap_size-1
                array[i]=EXTRACT_MAX(array)
        end while
        return array
end function
```

2 var created

$N \log N$

$C_0 N + P$

$C_1 N$

$N \log N$

$\Theta(N \log N)$

Space Complexity $\Theta(1)$

$O(1), O(\log N), O(N), O(N \log N) \ldots$

```
function BUILD-MAX-HEAP(A)
        heap_size=A.length
        for FLOOR(heap_size/2) < j ≤ 0
                MAX-HEAPIFY(A,j)
        end for
end function
```

C_0

$N/2 \Rightarrow N$ (2 is a constant)

$N \log N$

$\Theta(N \log N)$

Lower bound time complexity of heapsort: $\Omega(N \log N)$

$\Omega(\log N), \Omega(N), \Omega(1) \ldots$

```
function MAX-HEAPIFY(heap,root)
        largest=INDEX_LARGEST_NODE(root)
        if(largest != root)
                SWAP(heap[largest],heap[root])
                MAX-HEAPIFY(heap,largest)
        end if
end function
```

c_0

c_1

c_2

$T(N) = c_1 + c_2 + c_3$

$\Theta(\log N)$

log N in worst case it would have to traverse one whole branch

```
function EXTRACT-MAX(heap)
        max=heap[0]
        heap[0]=heap[heap_size-1]
        heap_size=heap_size-1
        MAX-HEAPIFY(heap,0)
        return max
end function
```

constant

logN

constant

$\Theta(\log N)$