Data Structures

CSCI 2270-202: REC 13

Sanskar Katiyar

Logistics

Office Hours (Zoom ID on Course Calendar)

Wednesday: 3 pm – 5 pm

Thursday: 5 pm – 6 pm

Friday: 3 pm – 5 pm

Recitation Materials *(Notes, Slides, Code, etc.)*

sanskarkatiyar.github.io/CSCI2270

Announcement

Project out this week

Due on April 29 (midnight MDT)

Great job on the Midterm!

Posted grades for Part 2, should be visible soon

Recitation Outline

1. Priority Queue ADT
2. Binary Heap
3. Max-Heap: Operations
4. Exercise

Priority Queue ADT

Queue ordered over some criteria (*priority*: *comparable*)

- Each element has some priority
- Queue Front: access to element of highest priority

Elements of equal priority

- Order in which they were enqueued
- Or Undefined

Applications: Scheduling, Huffman coding, etc.

Recall Assignment 1:
Insert into Sorted Array

Priority Queue ADT

Insert(x)	Priority Queue (Max First)
$C^{(10)}$	$\mathbf{C^{(10)}}$
$Q^{(43)}$	$\mathbf{Q^{(43)}}$, $C^{(10)}$
$A^{(22)}$	$Q^{(43)}$, $\mathbf{A^{(22)}}$, $C^{(10)}$
$T^{(17)}$	$Q^{(43)}$, $A^{(22)}$, $\mathbf{T^{(17)}}$, $C^{(10)}$
$N^{(87)}$	$\mathbf{N^{(87)}}$, $Q^{(43)}$, $A^{(22)}$, $T^{(17)}$, $C^{(10)}$

Invariant: Some property that must be maintained at all times

Binary Heap ≈ Heap in this presentation

Binary Heap

Complete Binary Tree

[Height(left_subtree) – Height(right_subtree)] == 0 or 1

Node consists of a **comparable item** that defines an ordering over all nodes of the tree ***(key)***

Invariant

Each node's key >= its children's key **(Max-Heap)**

Each node's key <= its children's key **(Min-Heap)**

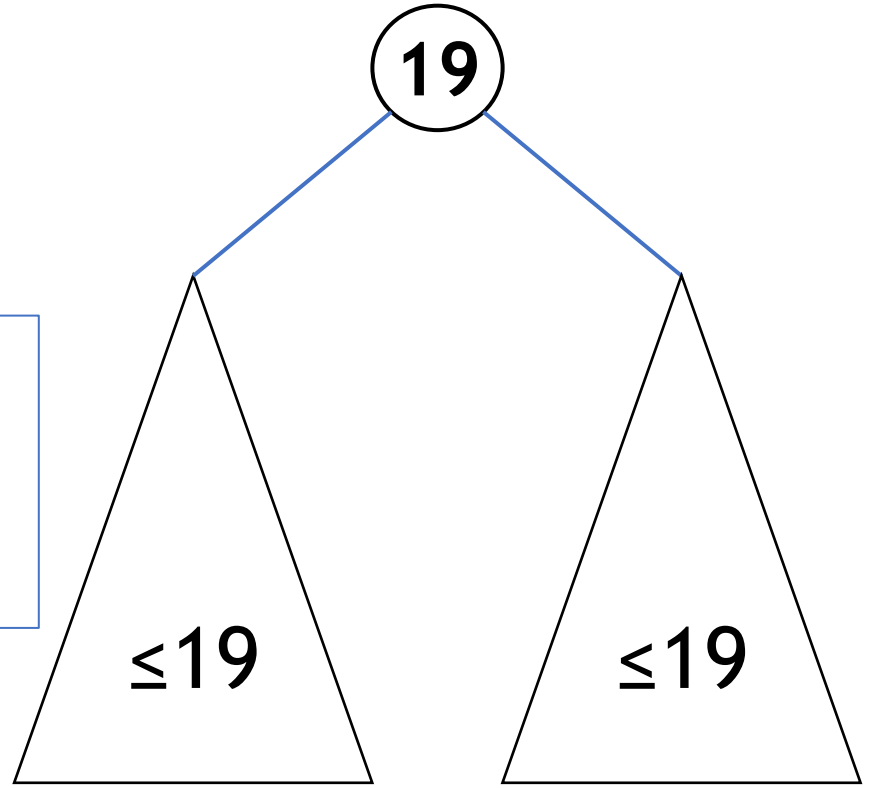
Heap

Heap = Complete BT + *Invariant*

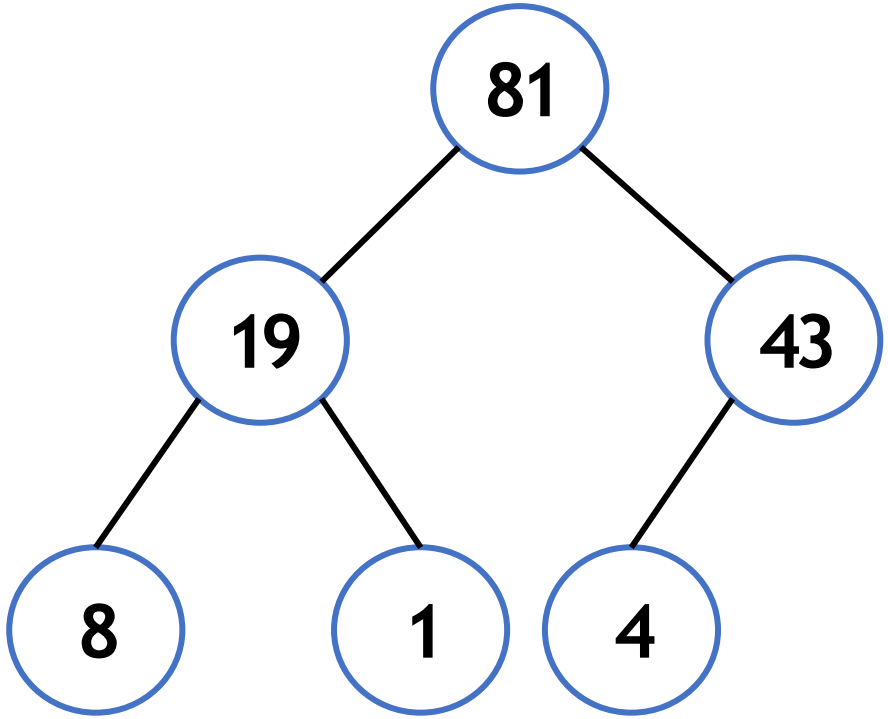
[MaxHeap] For any non-leaf node *N*:

$$\boldsymbol{X}.data \leq \boldsymbol{N}.data$$

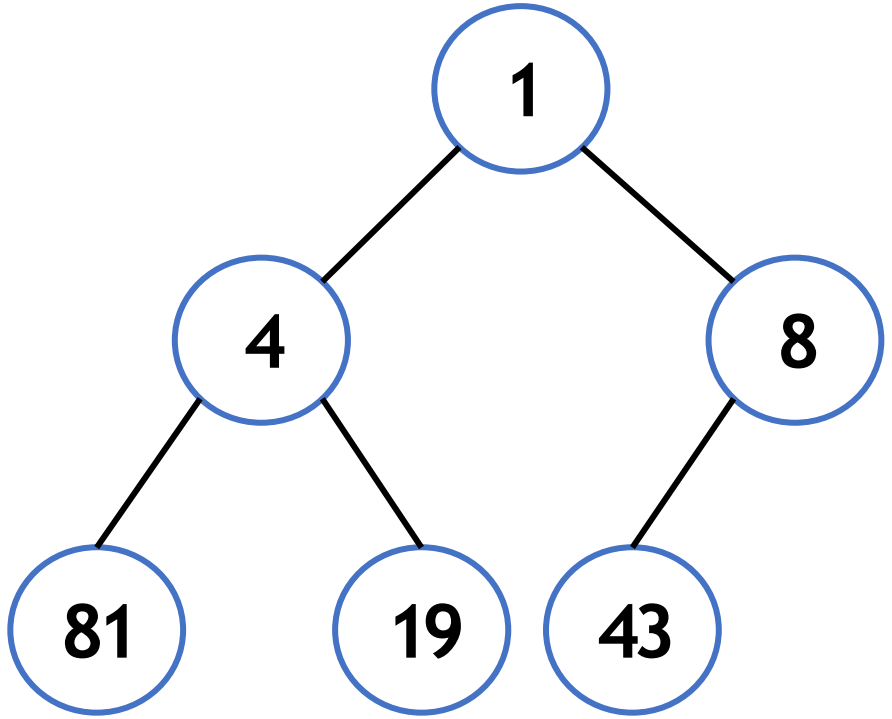
$$\forall\, \boldsymbol{X} \in \{\boldsymbol{N}.left\_subtree, \boldsymbol{N}.right\_subtree\}$$



Heap: Max-Heap, Min-Heap



Max-Heap



Min-Heap

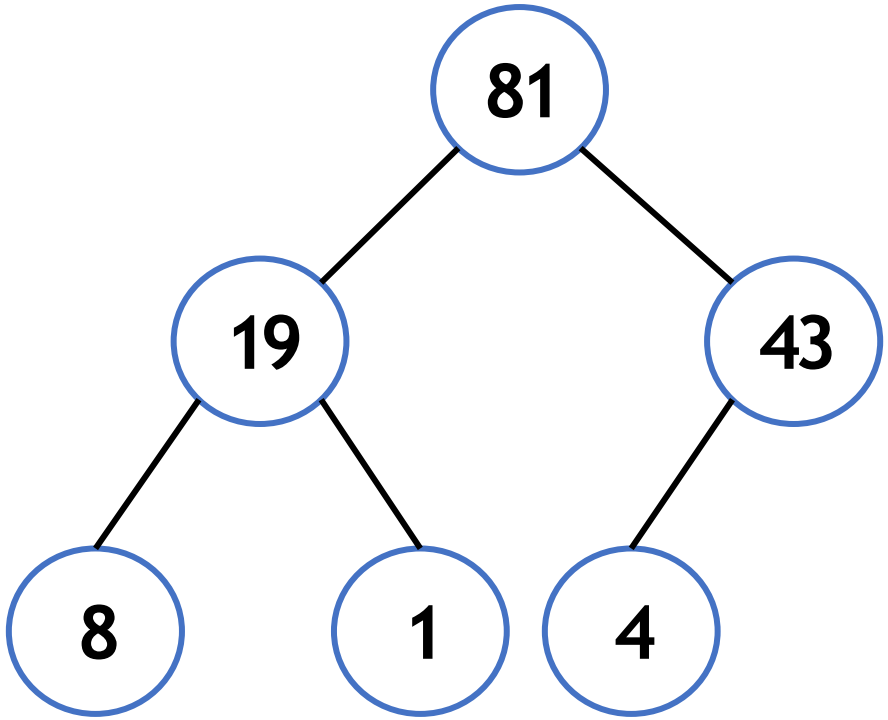
Heap: Max-Heap

Which node has maximum priority?

Root node

Where to find nodes with minimum priority?

Leaf nodes

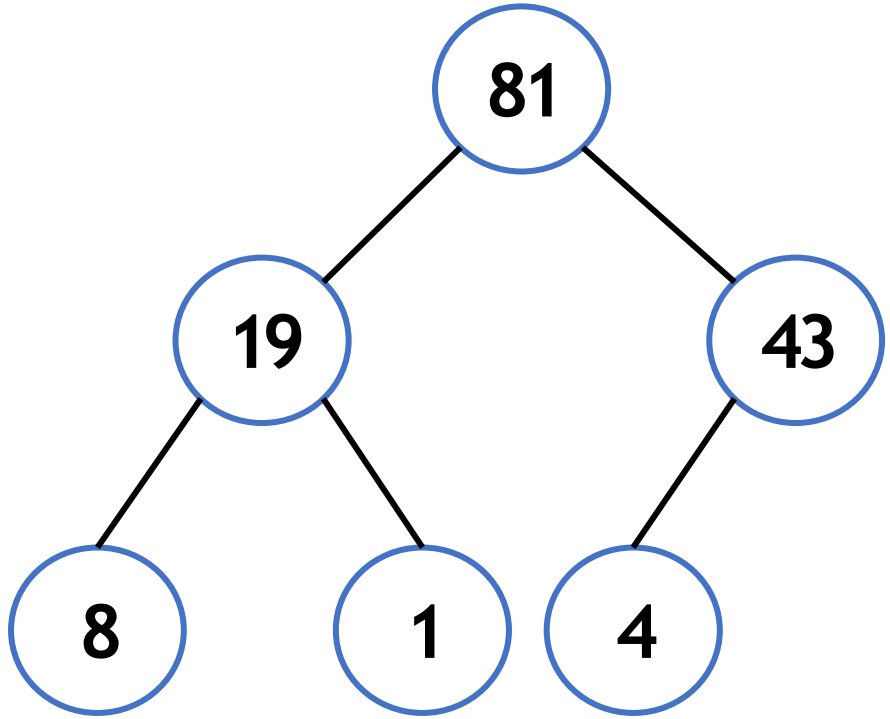


Max-Heap

Heap: Complete BT using an Array

Item Node	A[i]
Left Child	A[2*i + 1]
Right Child	A[2*i + 2]
Parent	A[floor((i-1) / 2)]

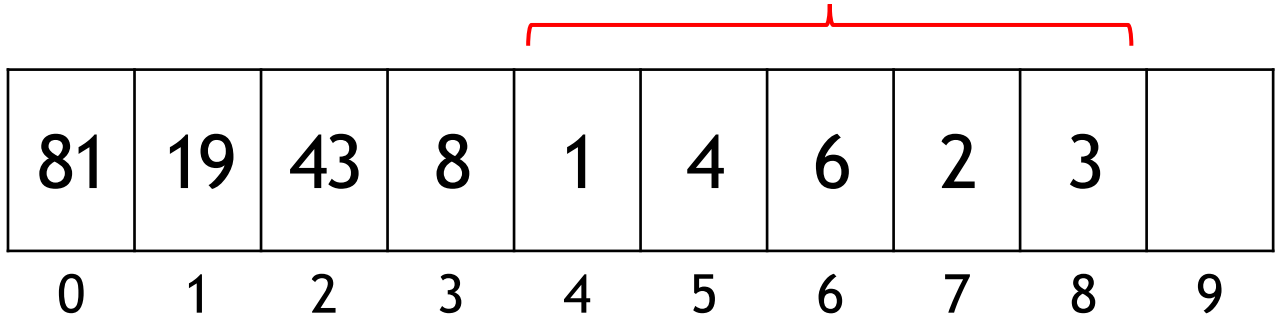
81	19	43	8	1	4	
0	1	2	3	4	5	6

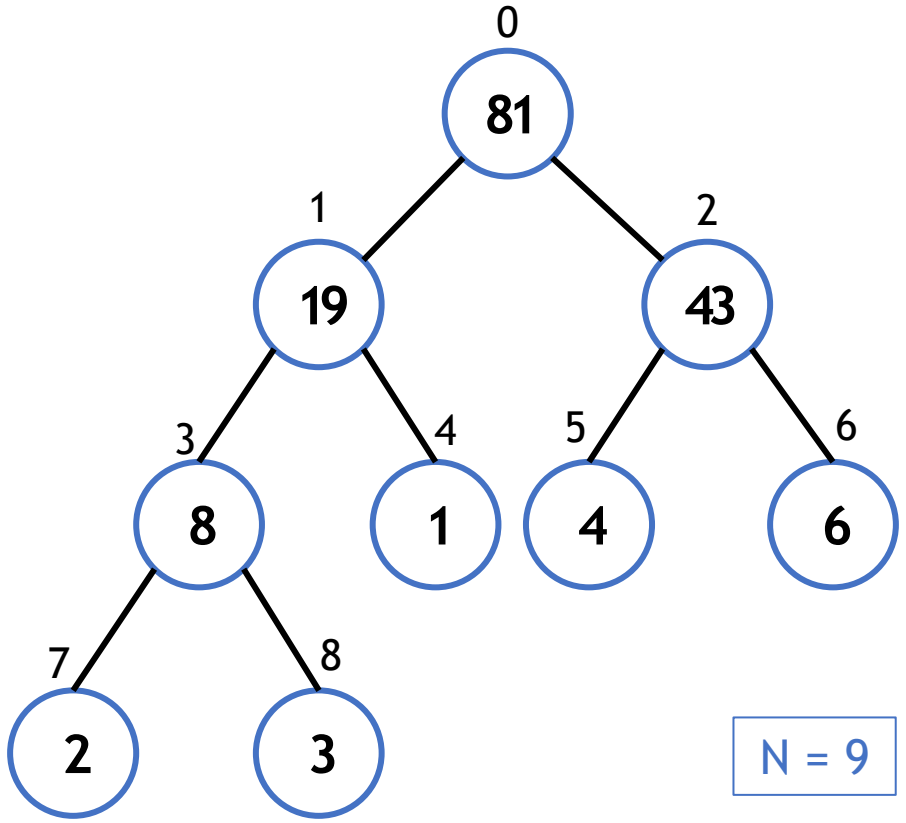


Heap: Complete BT using an Array

Range of Indices of Leaf Nodes

Starting at index = **`ceil((N - 1) / 2)`**
Ending at index = **`[N - 1]`**





Heap: Complete BT using an Array

Why Arrays for Heaps but not for BST?

Heap = Complete BT (unlike BST)

Pointer-based implementation takes more space, good for sparse trees

Arrays provide cache locality

Heap: Members & Operations

Data Members

```
heapArr[]

capacity

currentSize
```

Operations

```
insertElement(priority, val)

deleteElement(val)

extractMax/Min()

max/minHeapify(int i)

BuildMax/MinHeap()
```

Code on slide is truncated. For complete code check the Github repo.

Heap: MaxHeapify

Given an index, rearrange the subtree to maintain heap invariant

Key Point #1: Assumes that the subtrees of given index i, are heaps

Key Point #2: If a child doesn't exist:

(i) Check using an if-statement to by-pass `if(l < currentSize && r < currentSize)`

(ii) Assign some priority

Max-Heap: INT_MIN

Min-Heap: INT_MAX

Heap: MaxHeapify

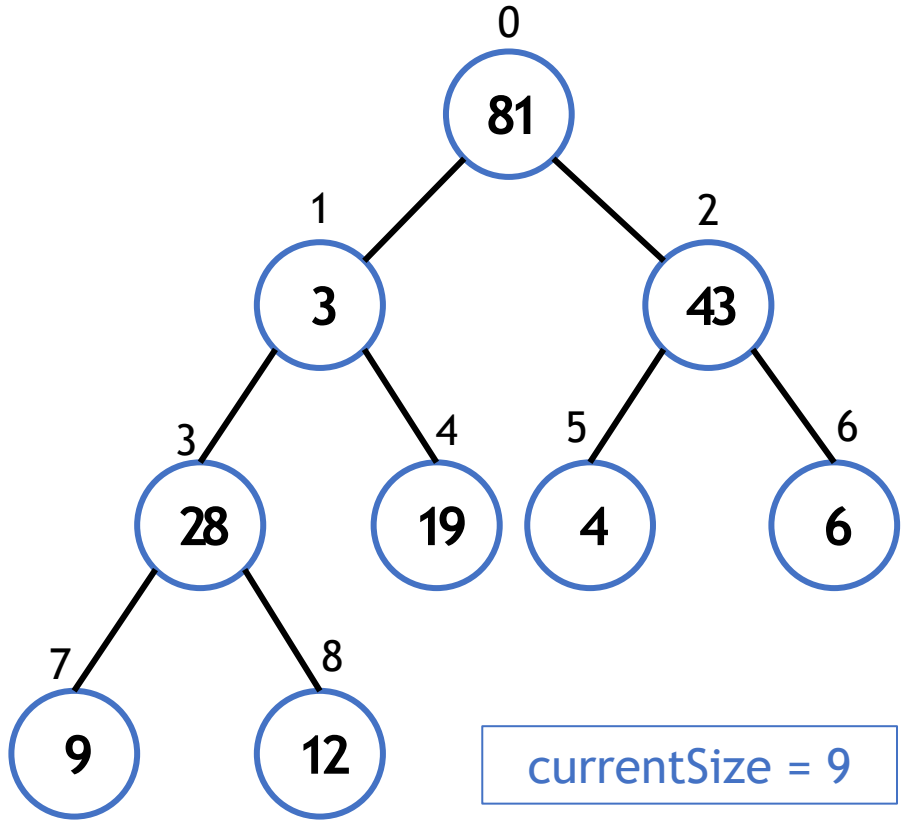
81	3	43	28	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(i)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

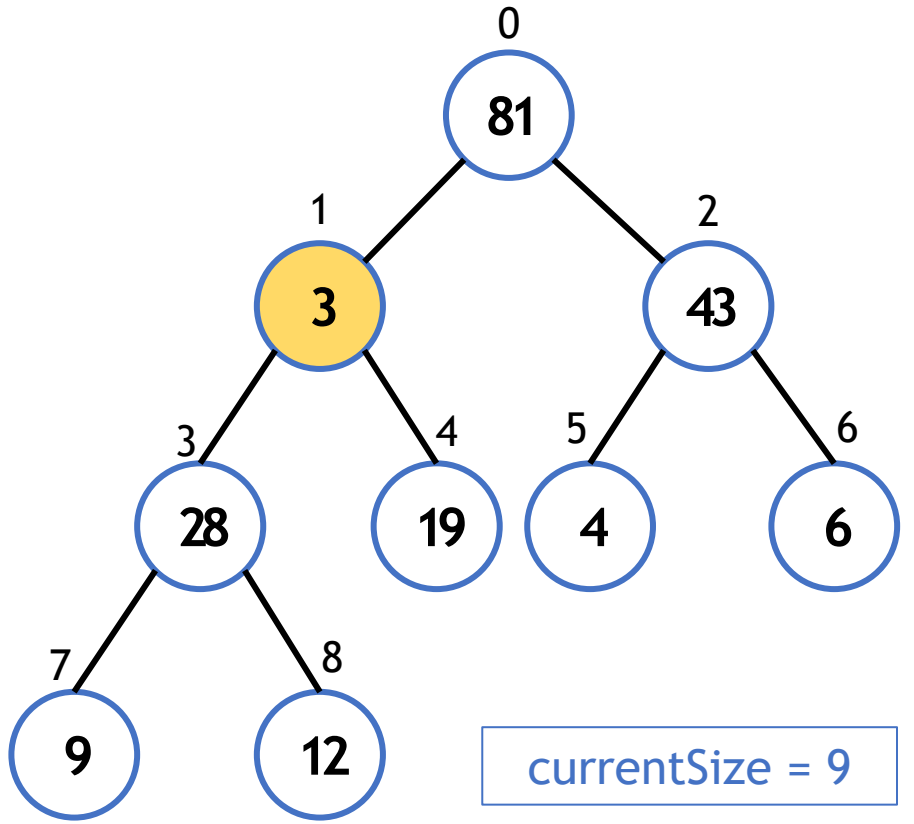
81	3	43	28	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

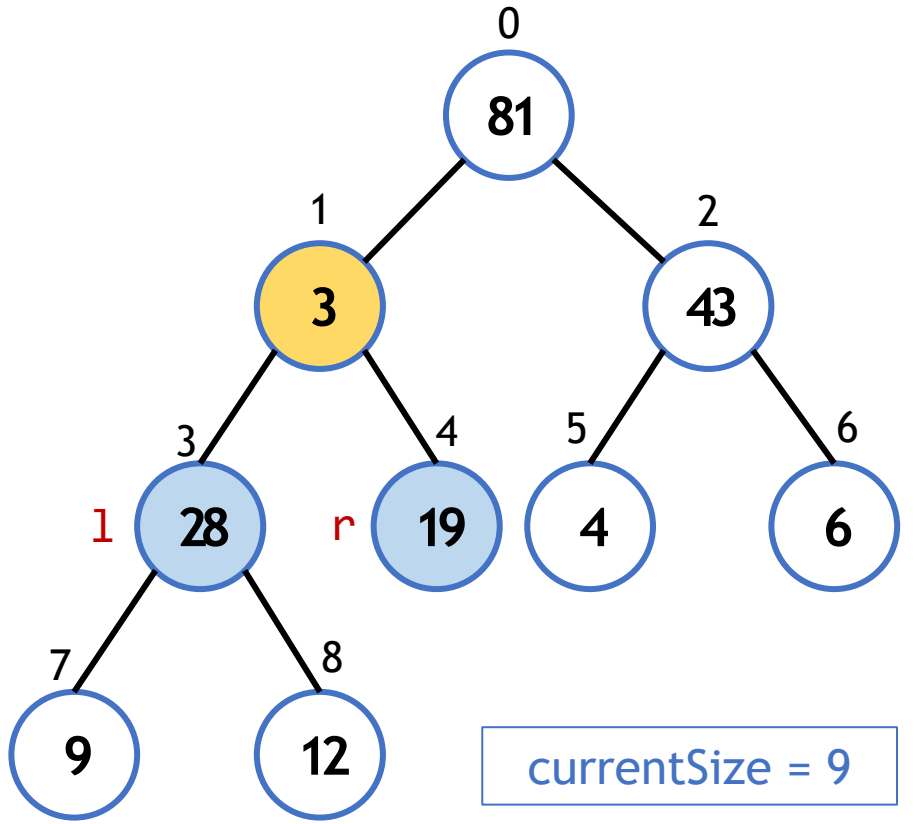
81	3	43	28	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

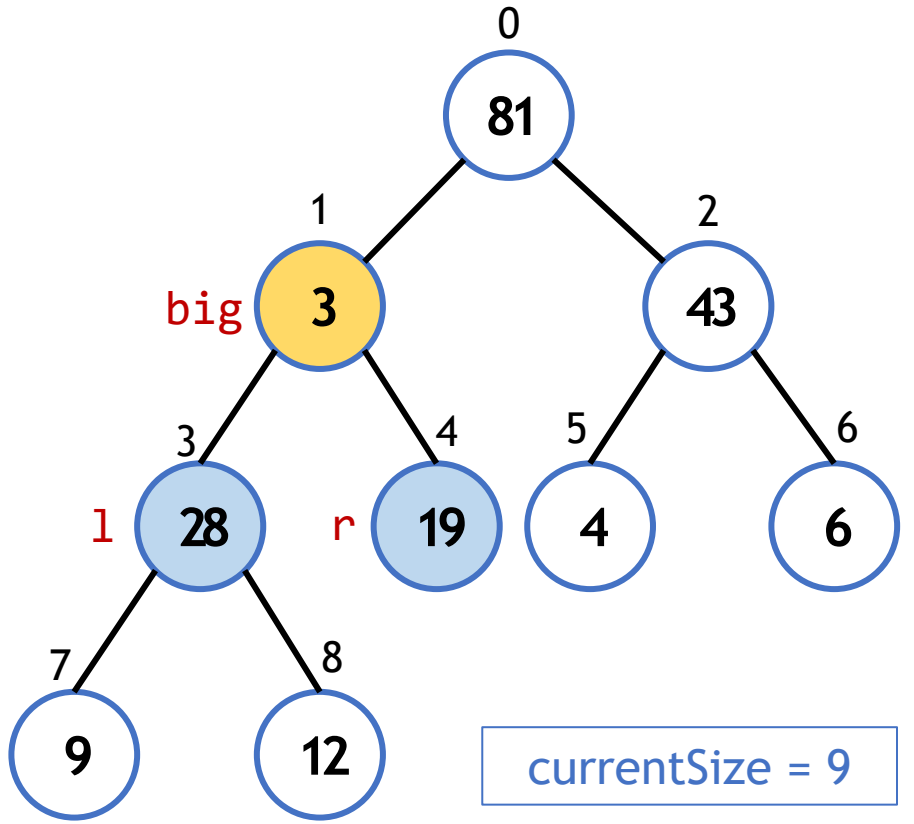
81	3	43	28	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

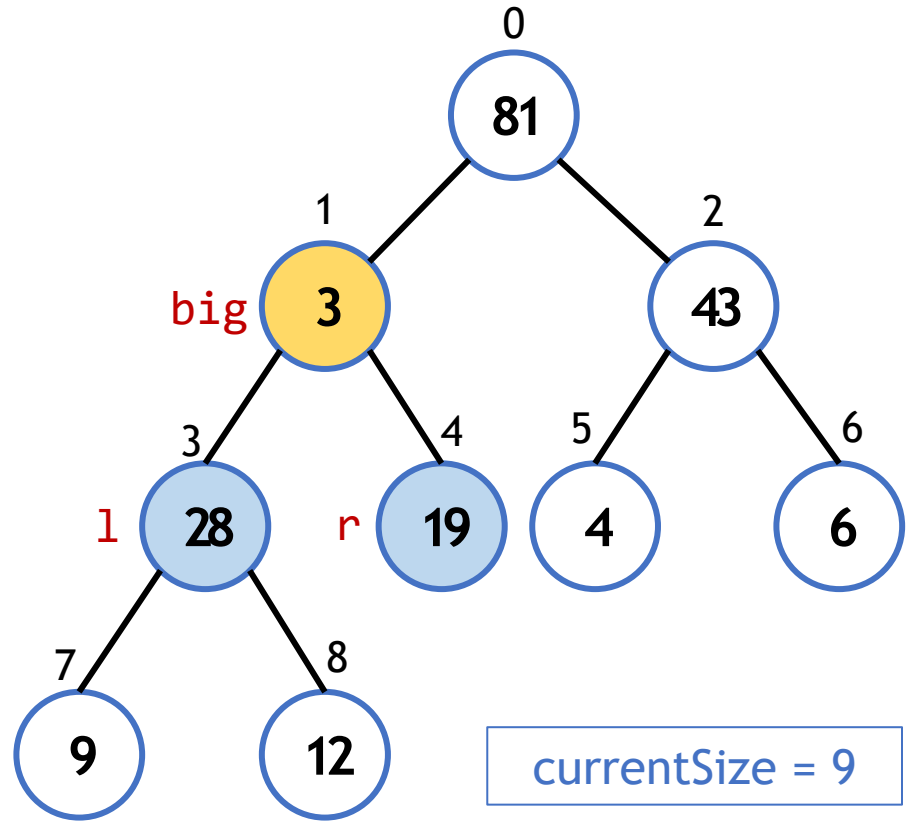
81	3	43	28	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

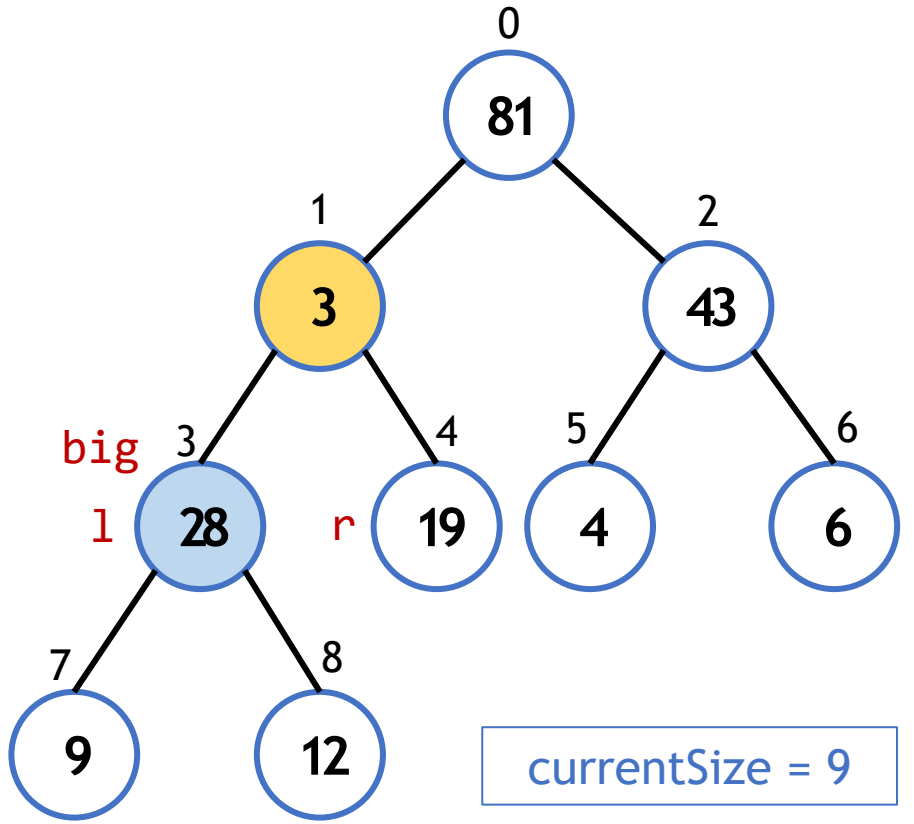
81	3	43	28	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

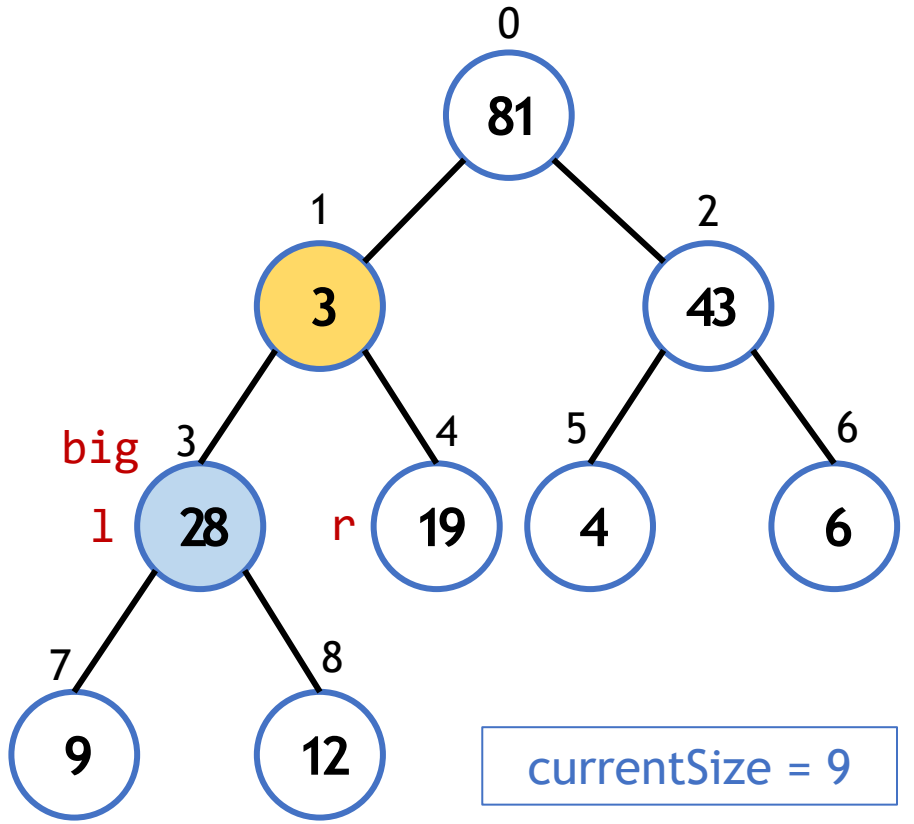
81	3	43	28	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

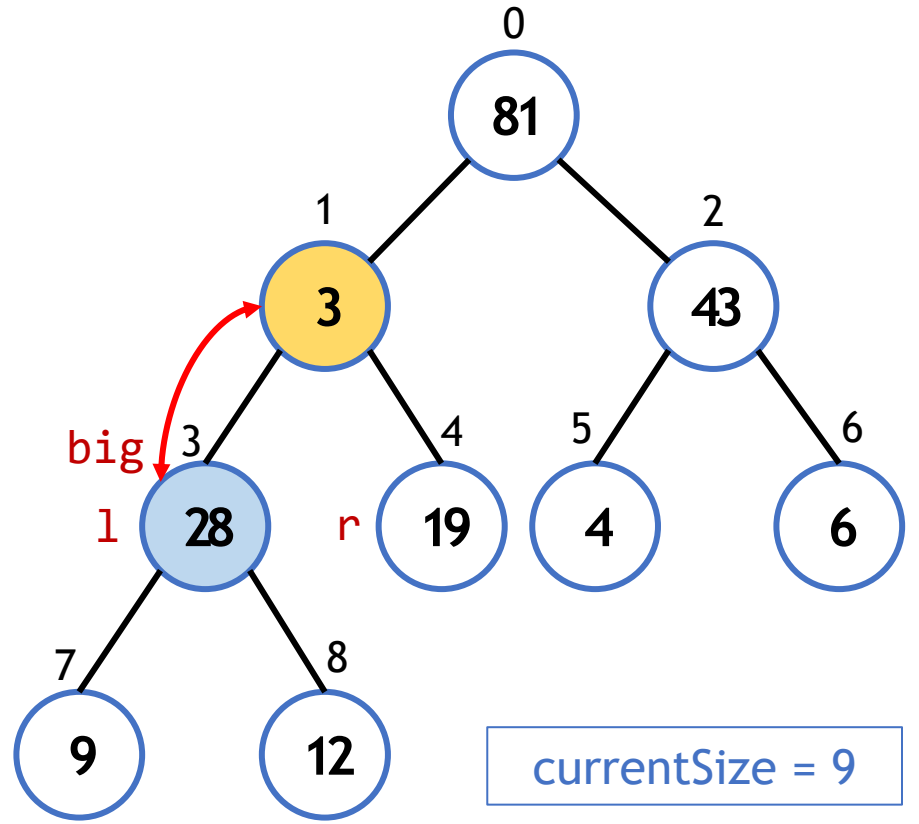
81	3	43	28	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

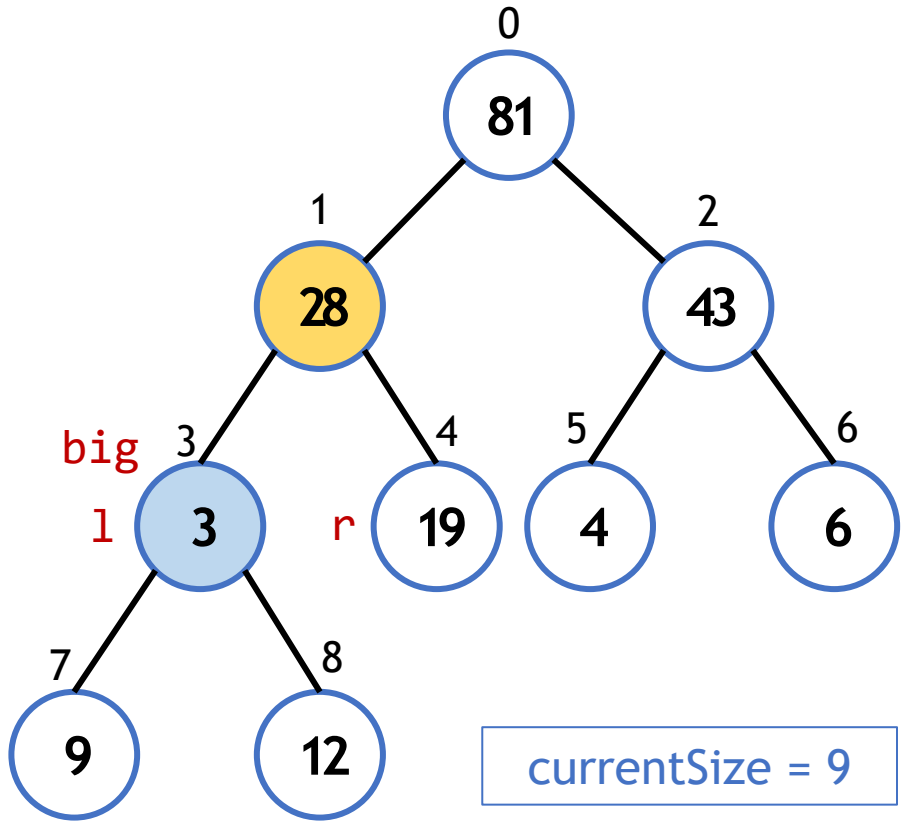
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

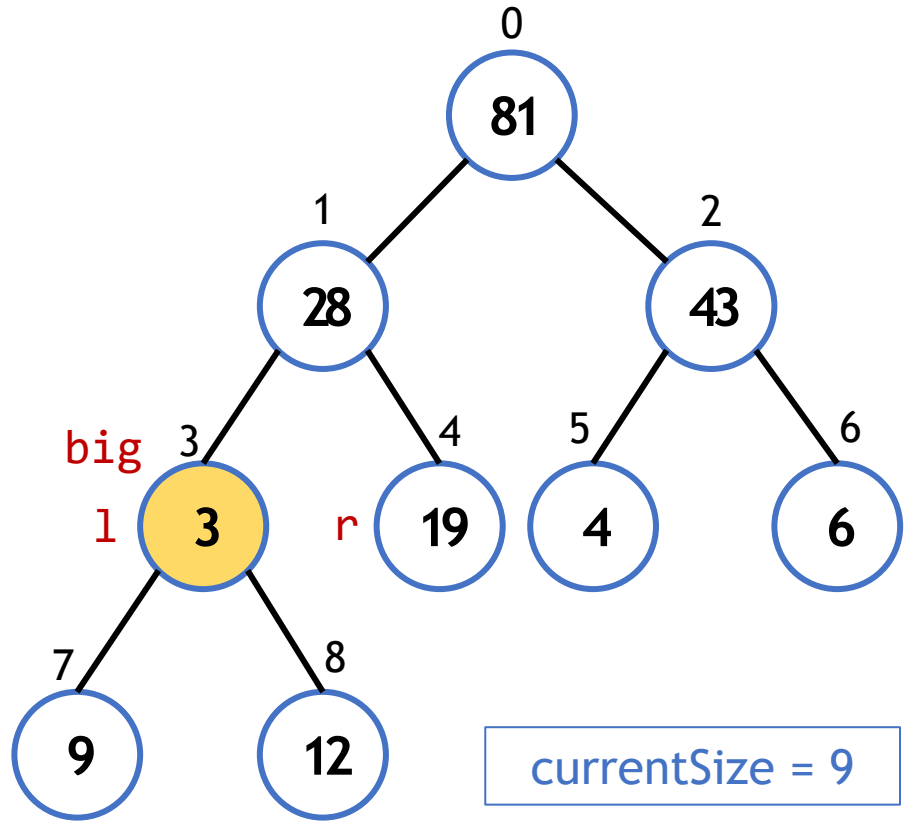
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(1)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

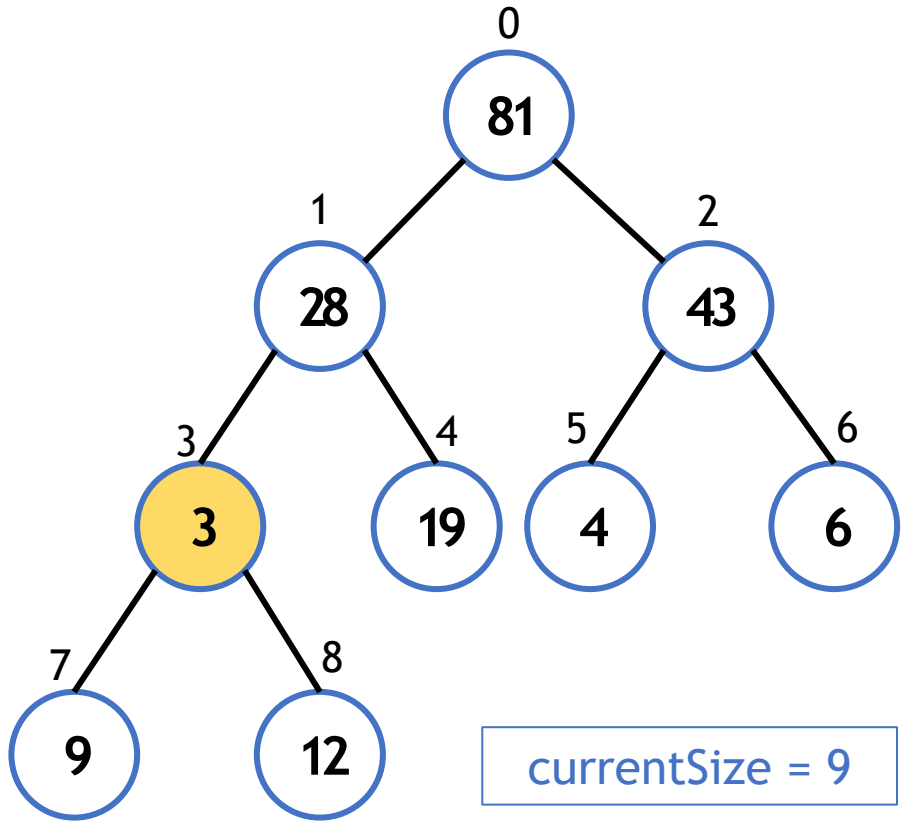
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

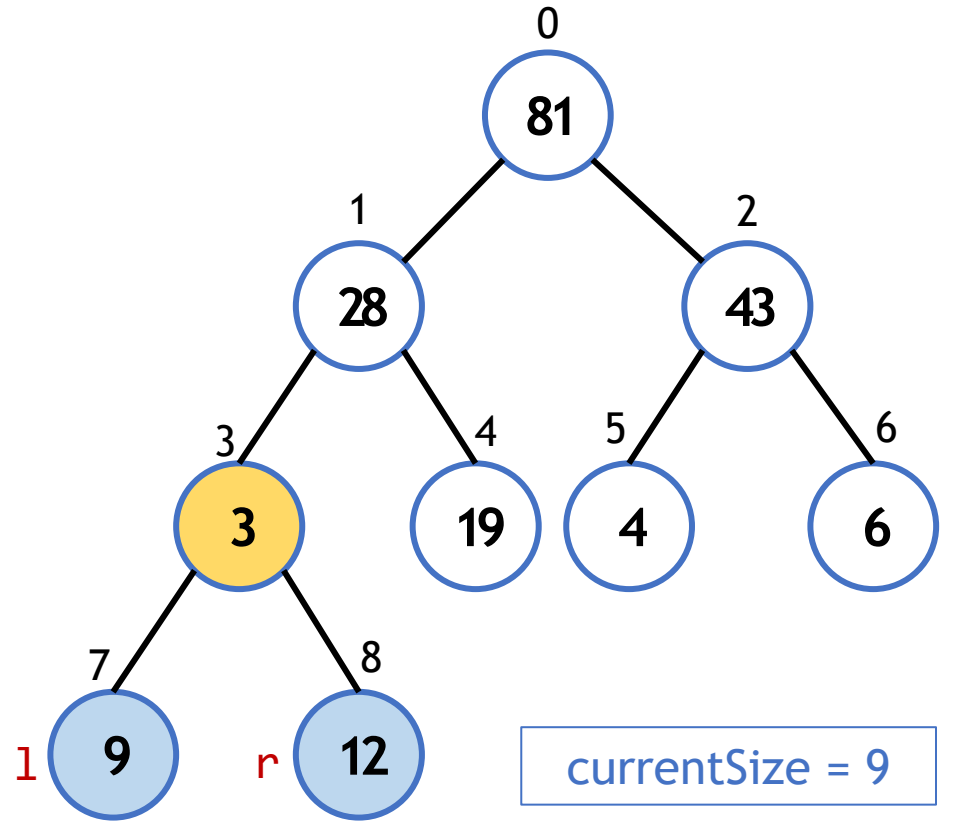
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

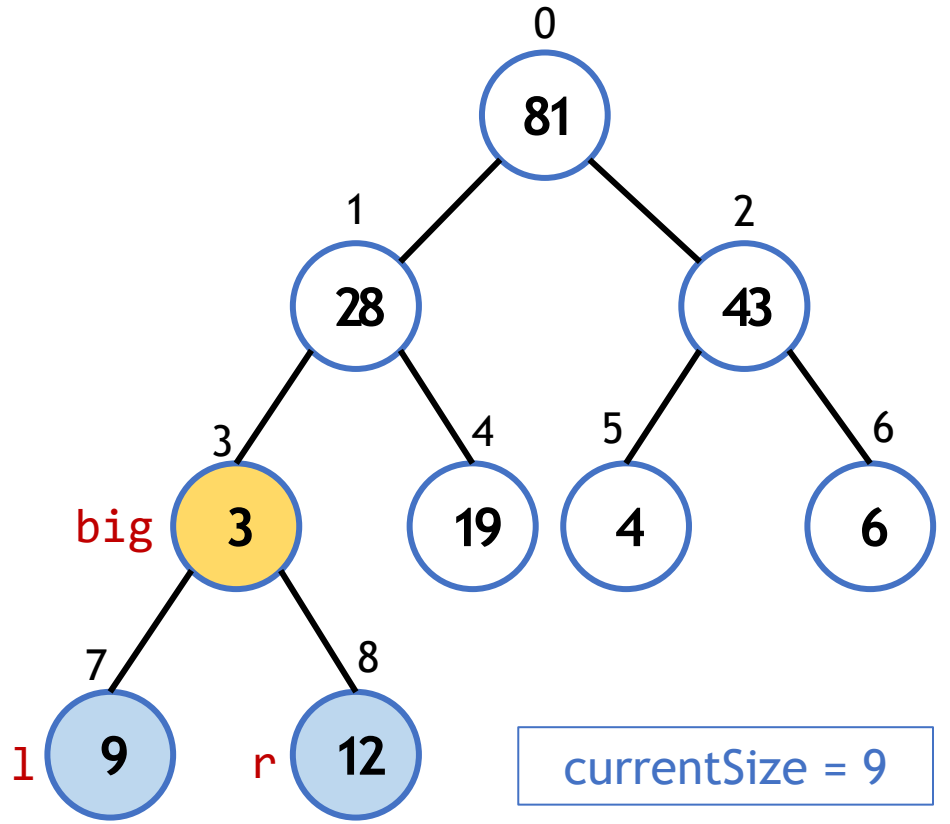
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

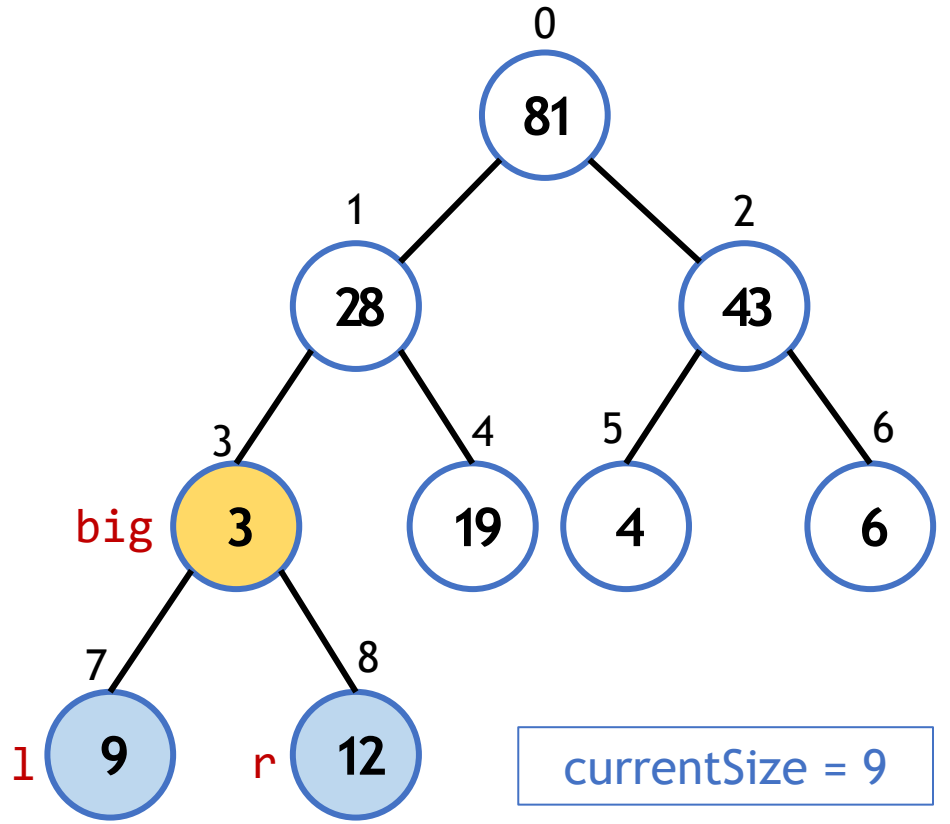
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

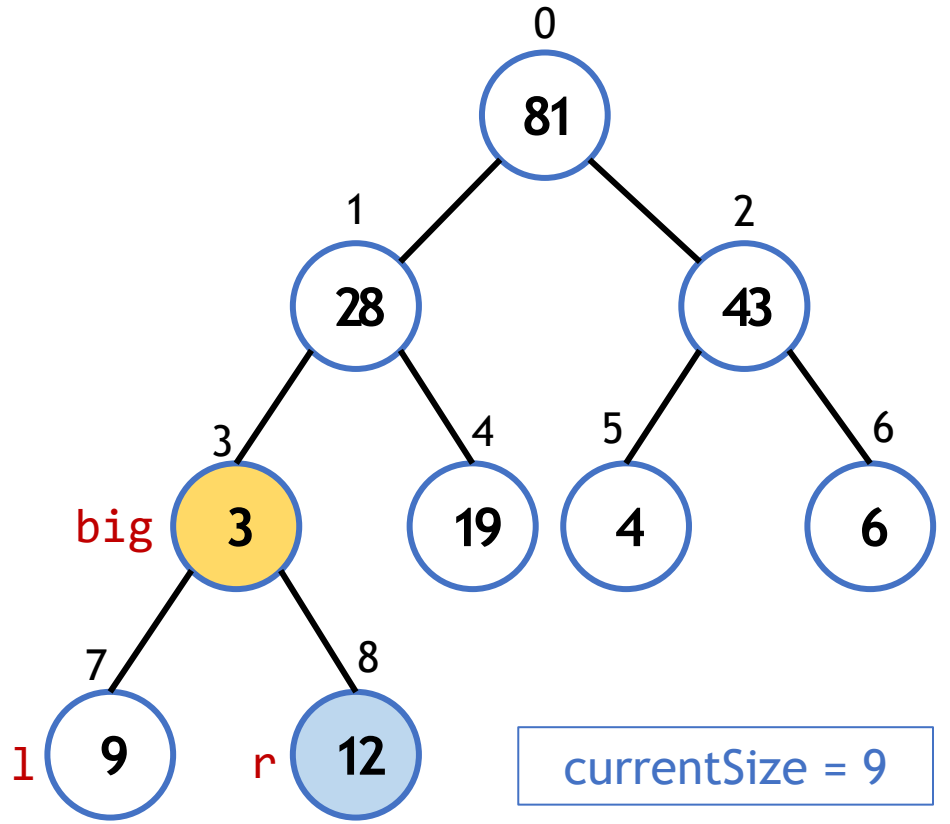
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

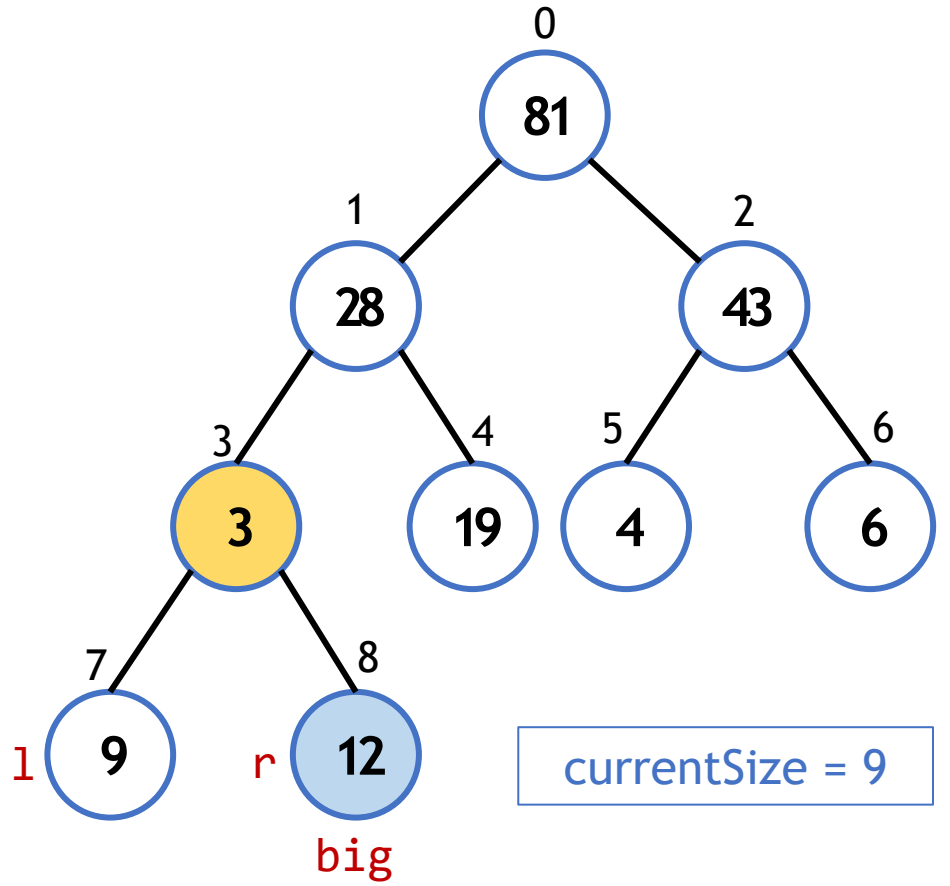
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

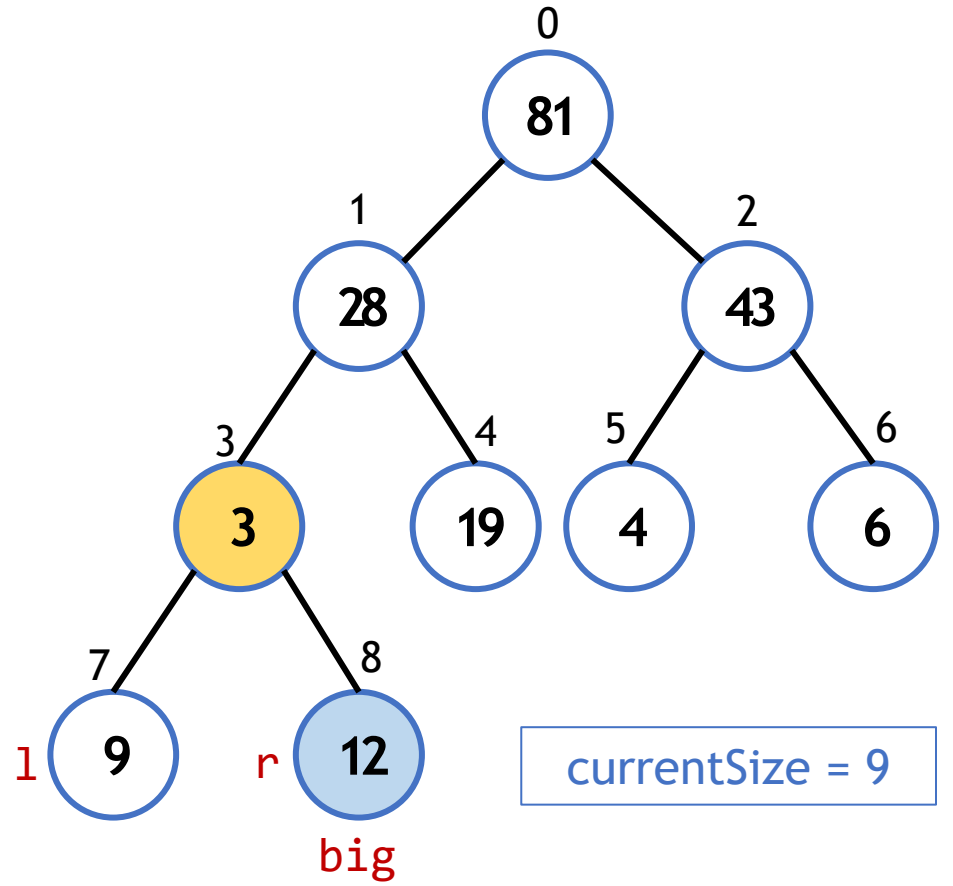
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

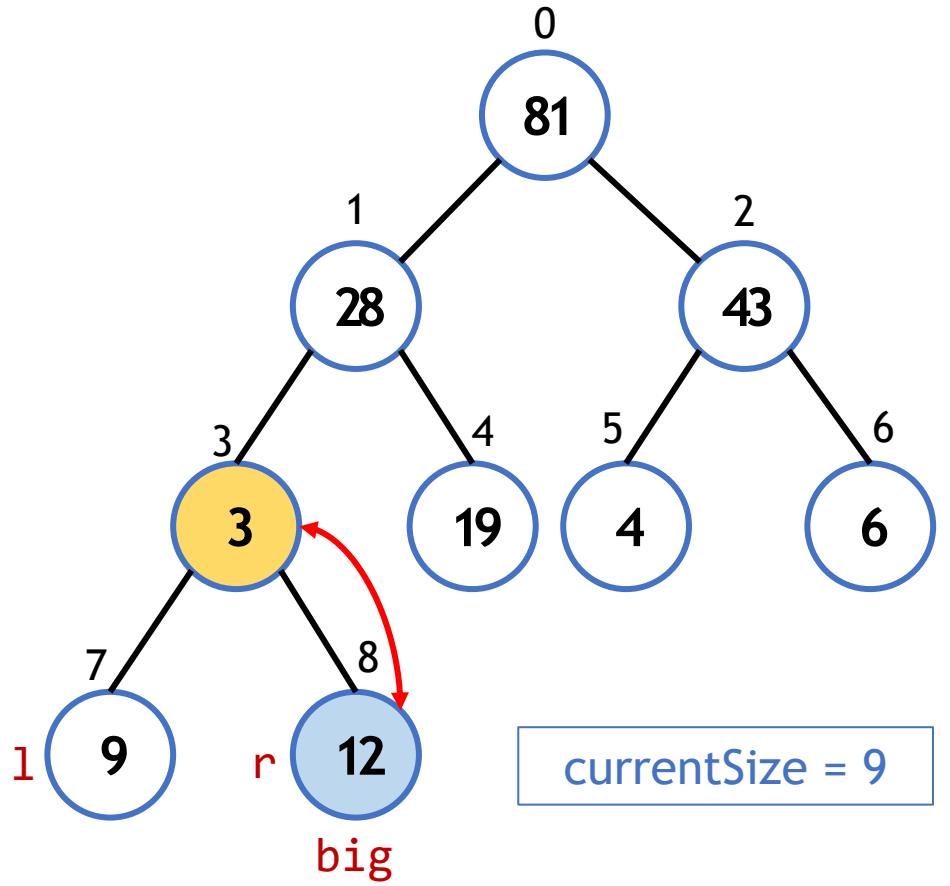
81	28	43	3	19	4	6	9	12
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

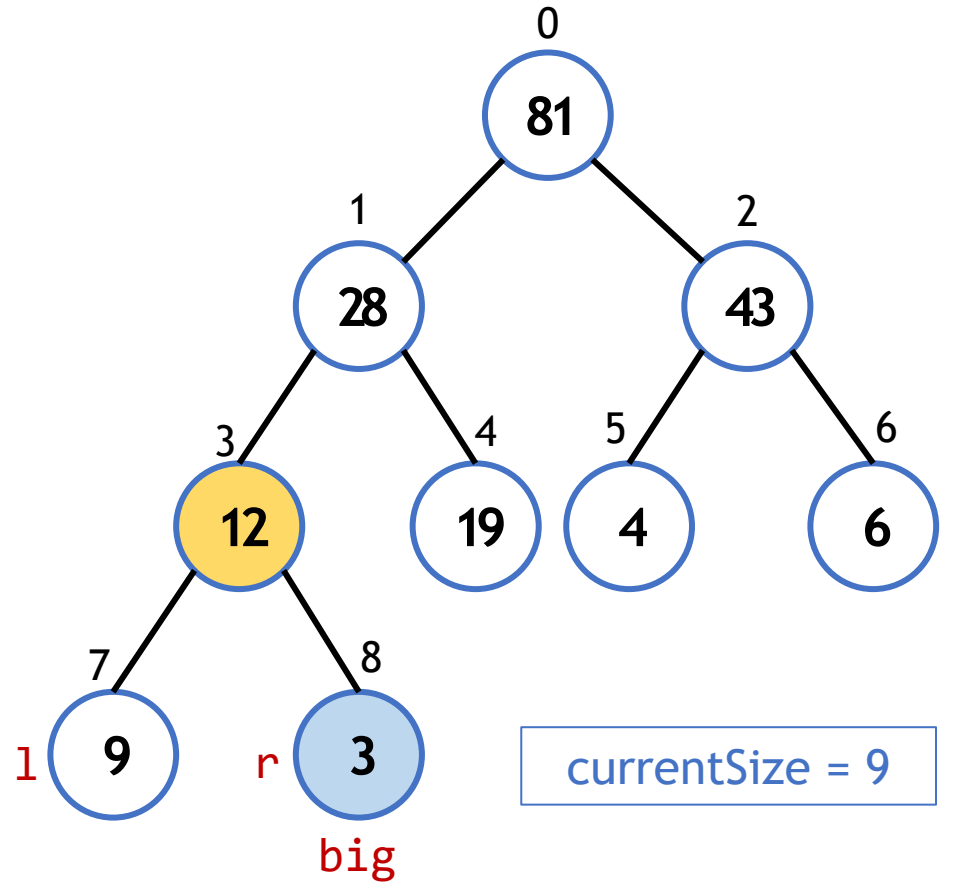
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

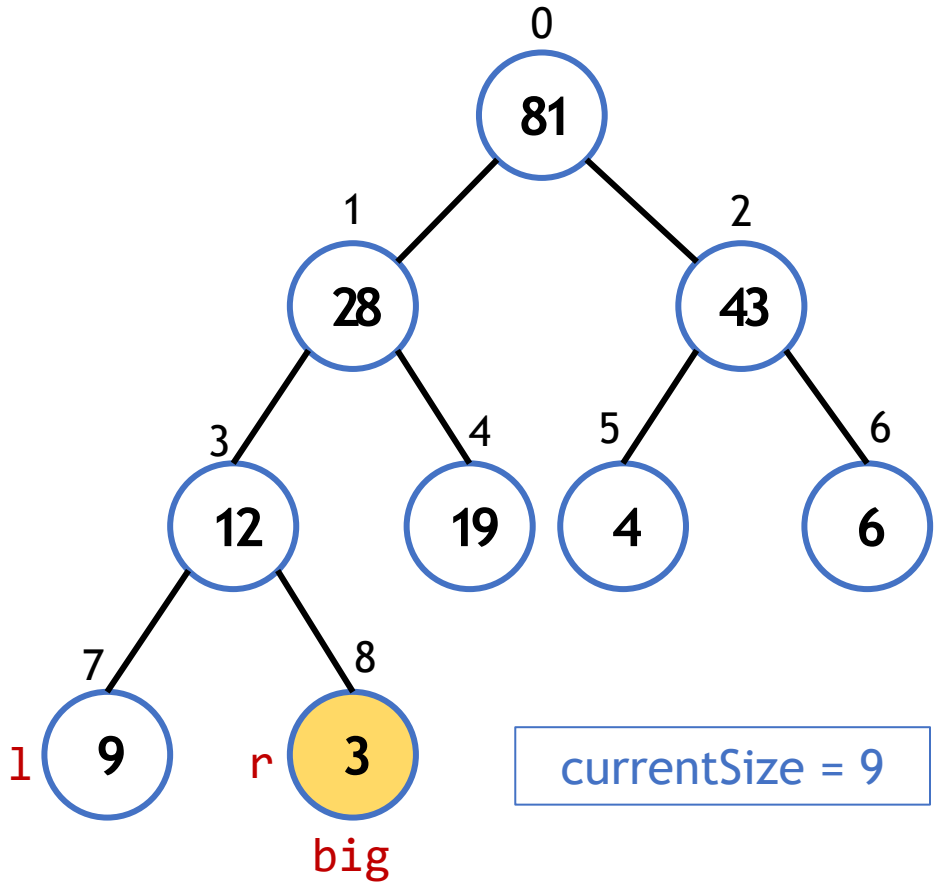
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(3)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

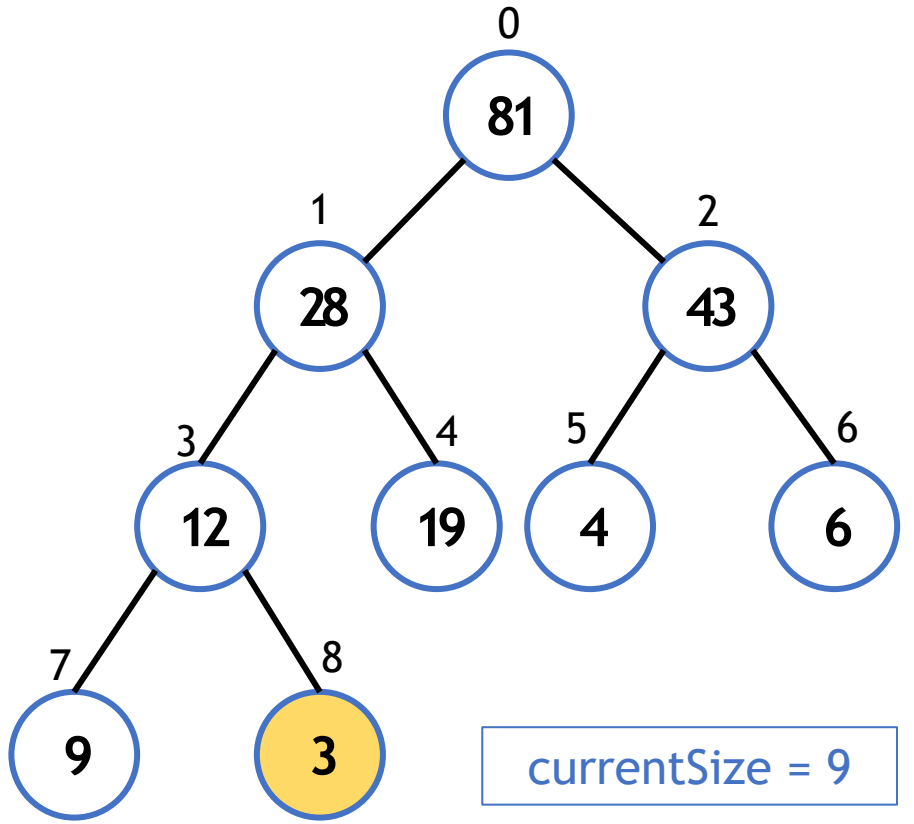
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(8)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

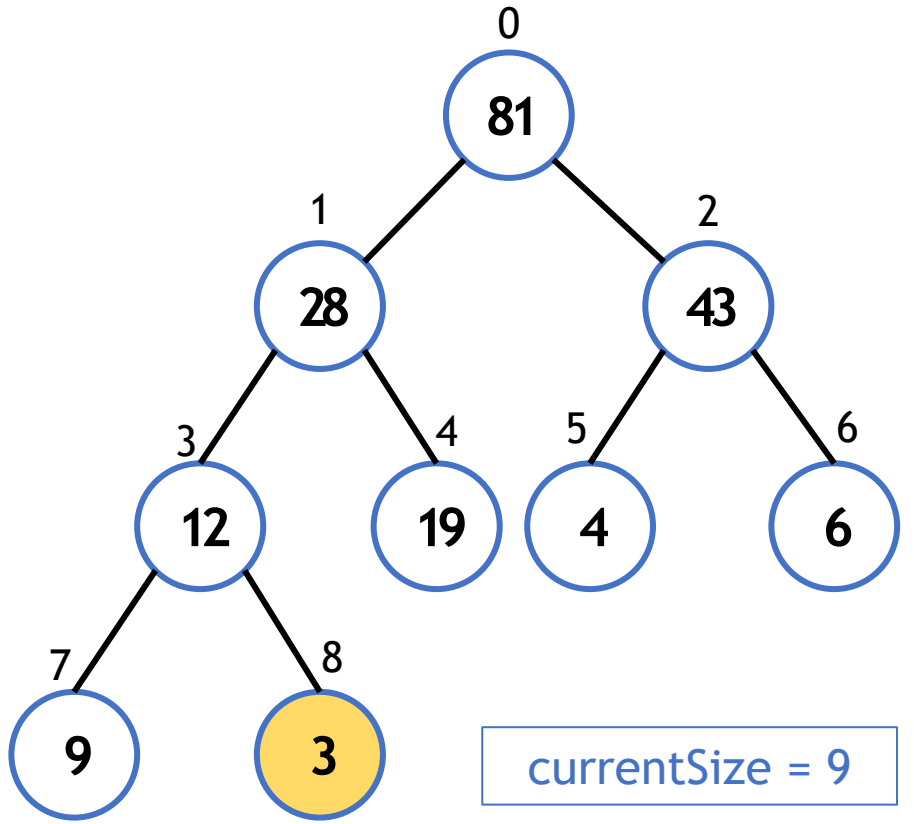
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(8)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

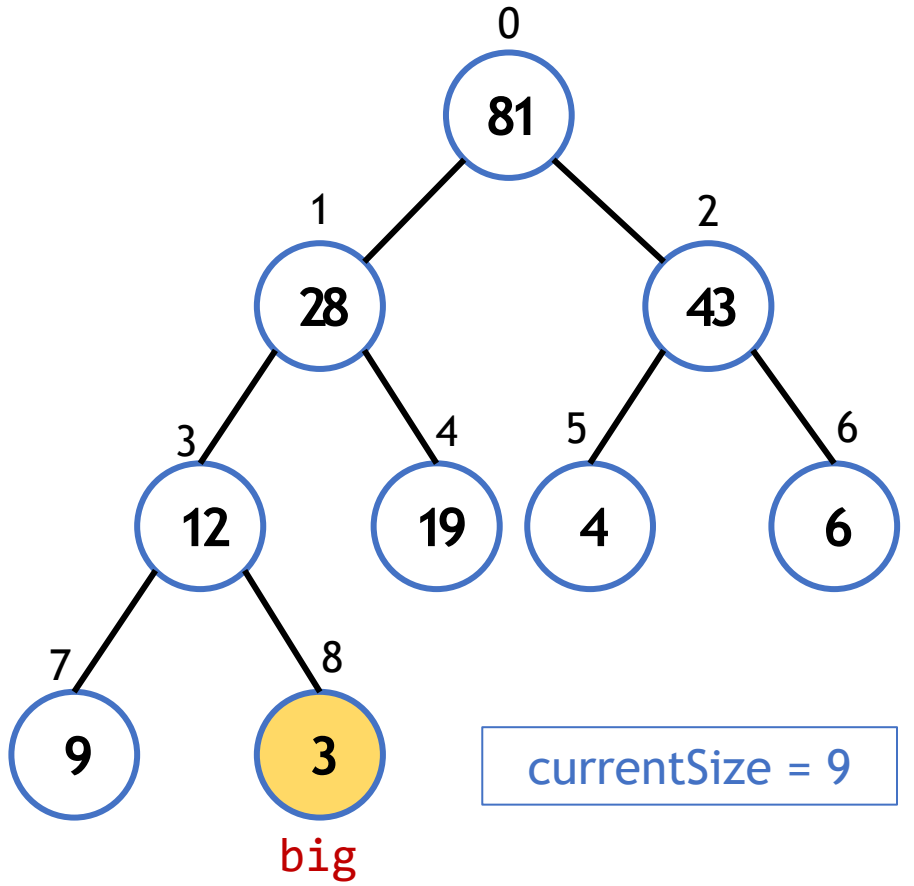
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(8)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

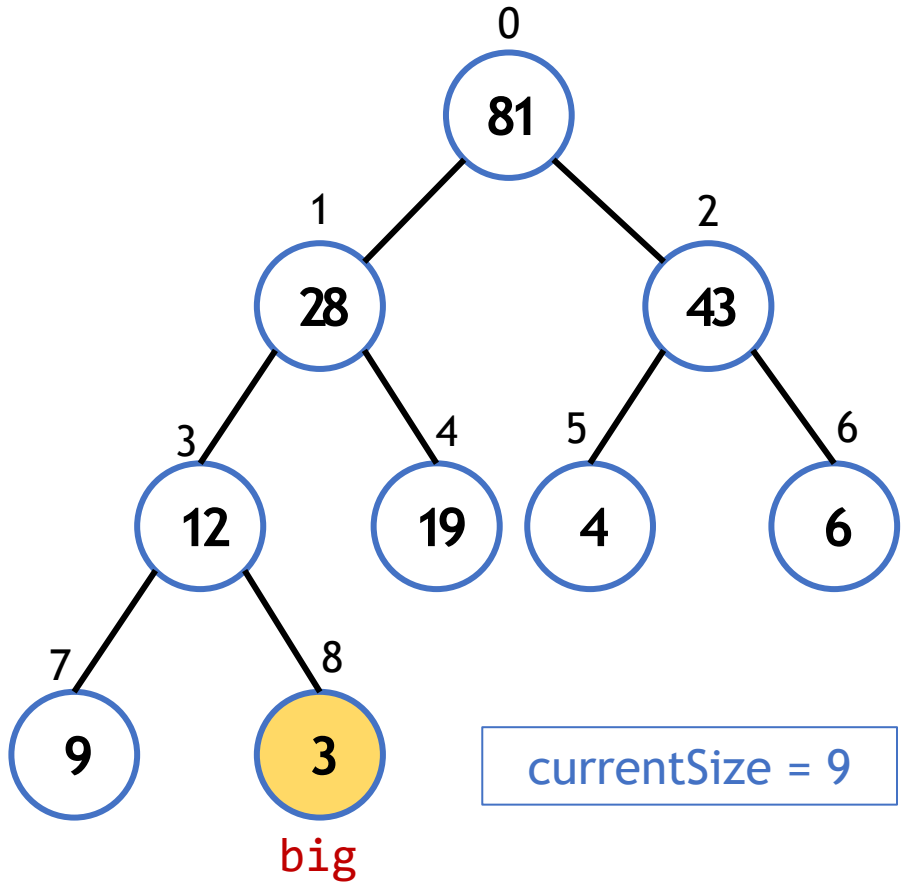
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(8)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

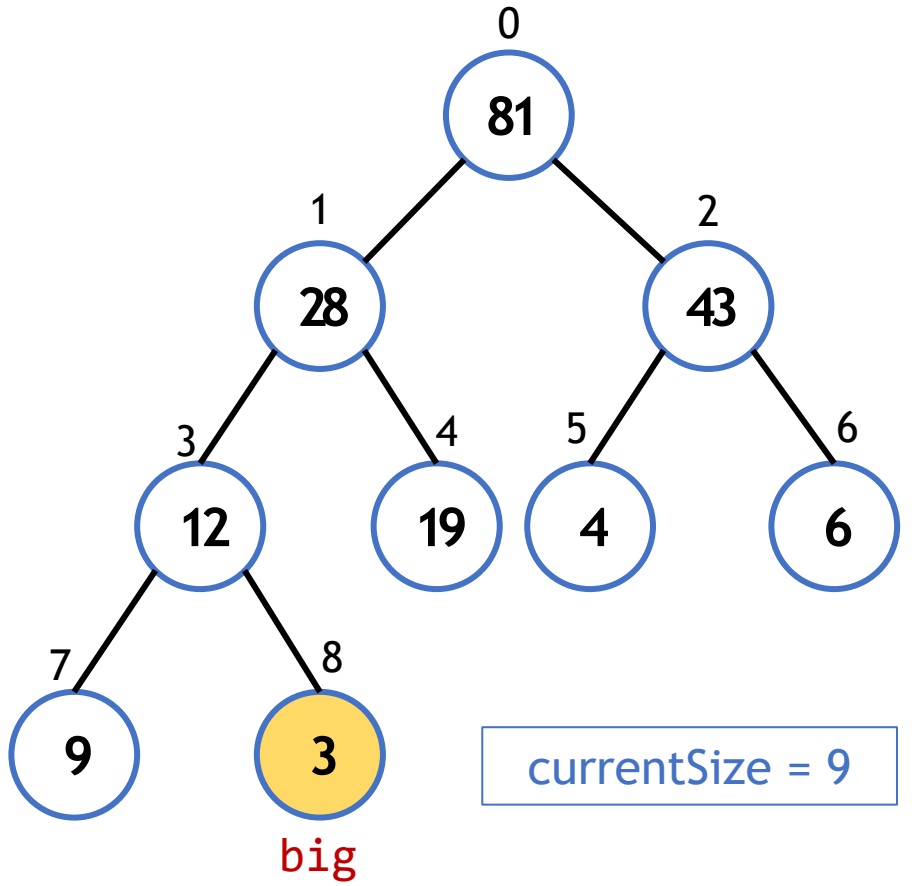
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(8)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

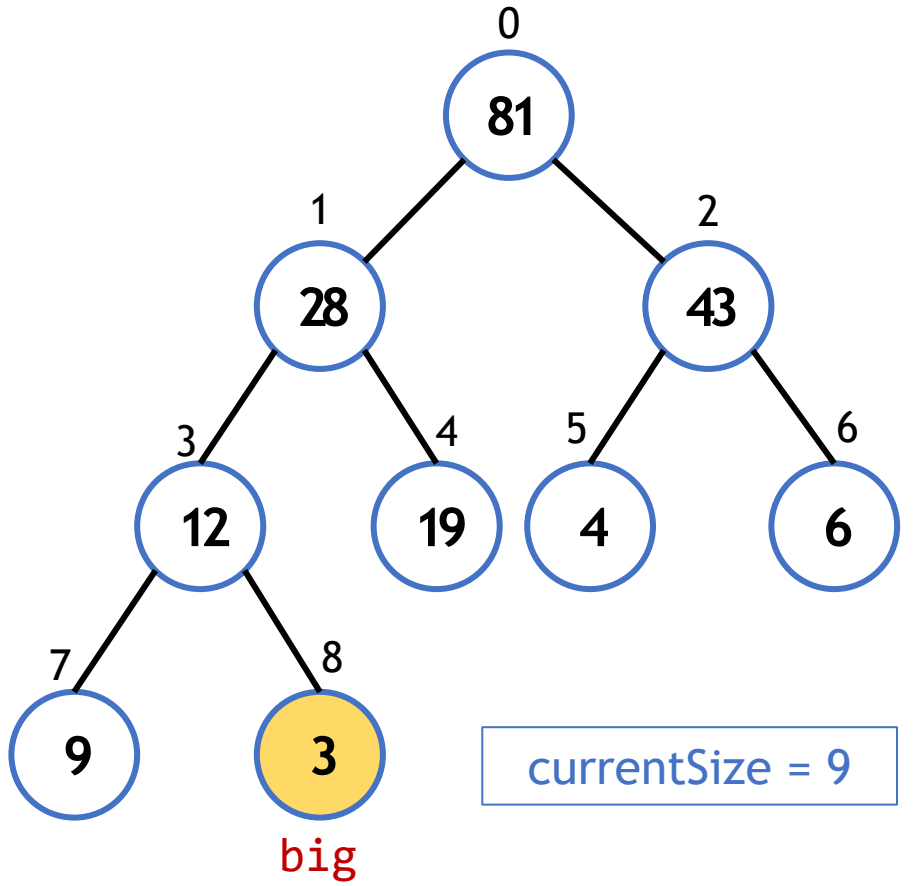
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(8)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

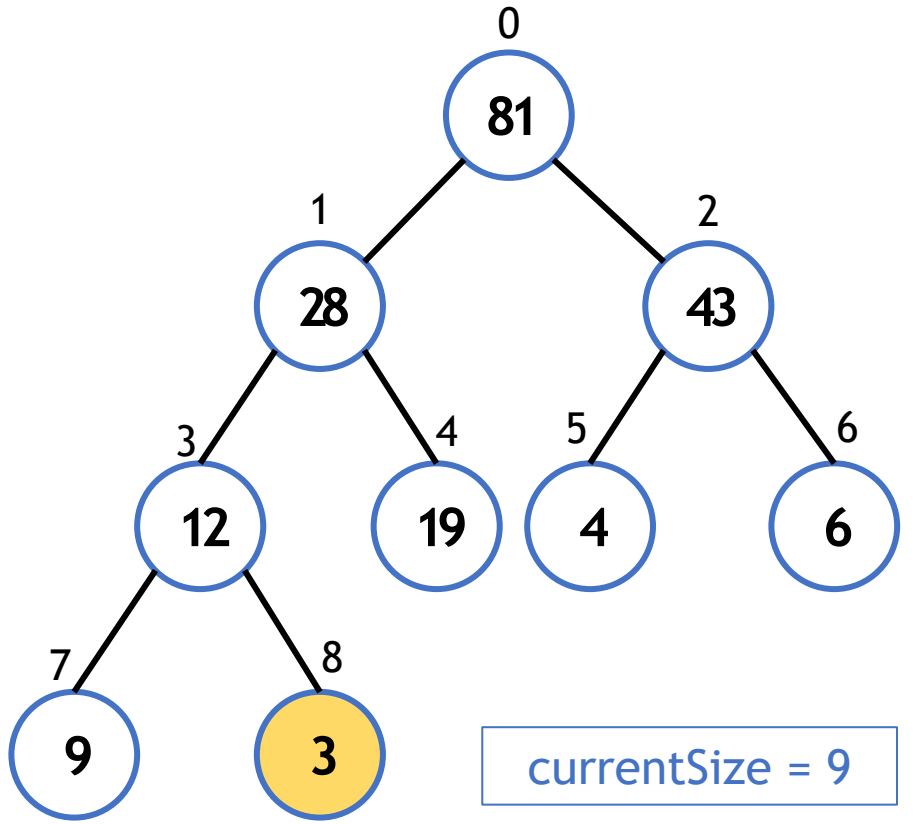
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(8)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: MaxHeapify

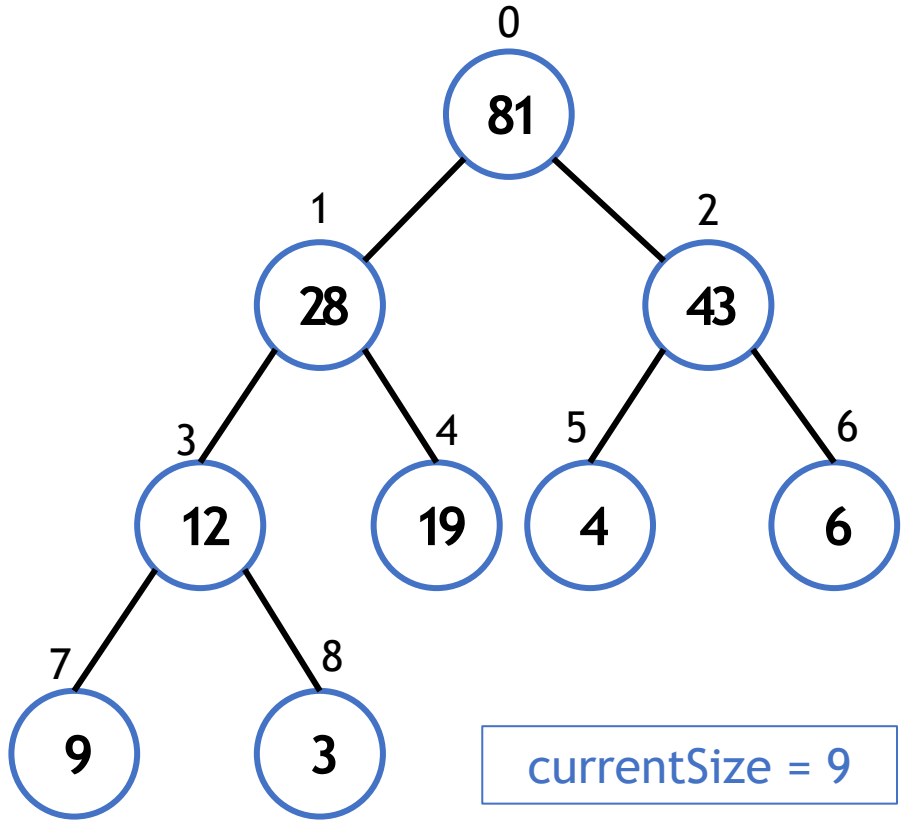
81	28	43	12	19	4	6	9	3
0	1	2	3	4	5	6	7	8

MaxHeapify(i)

```
l = leftChild(i); r = rightChild(i);
big = i;

if(A[l] > A[i] && A[l] > A[r]) big = l;
else if(A[r] > A[i]) big = r;

if(big != i) {
    swap(A[i], A[big]);
    MaxHeapify(big);
}
```



Heap: extractMax

Remove (and, in some cases, return) the Max element

```
if(current_size <= 0) return;
A[0] = A[currentSize - 1];      // copy min element
currentSize = currentSize - 1; // delete min element
MaxHeapify(0);                  // maintain heap
```

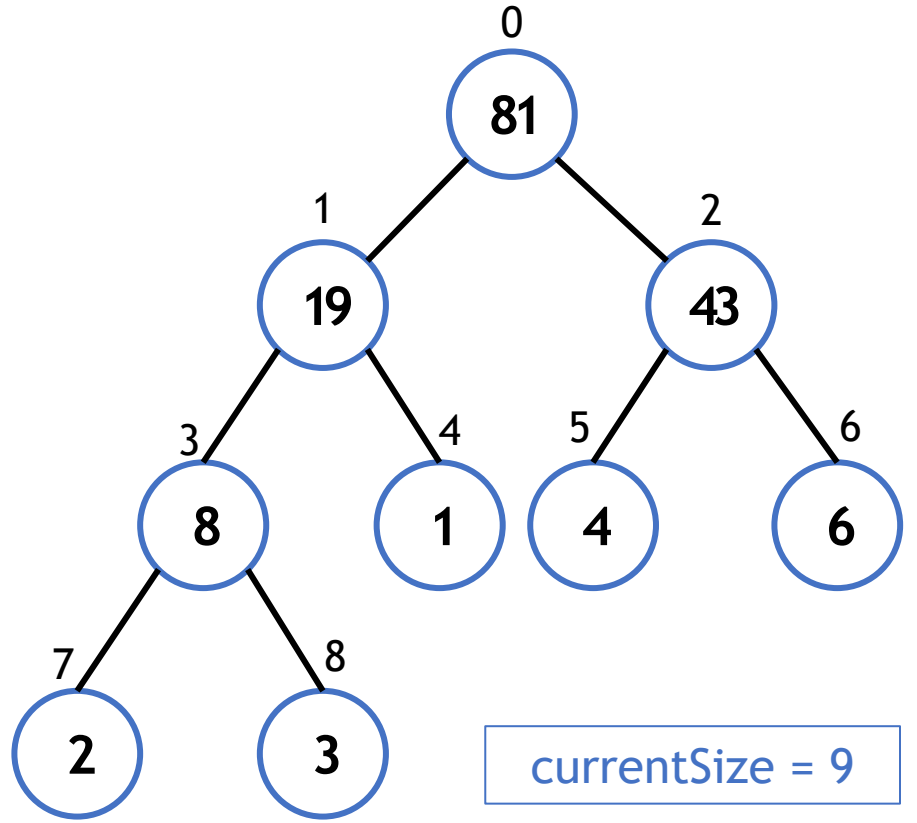
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

81	19	43	8	1	4	6	2	3
0	1	2	3	4	5	6	7	8



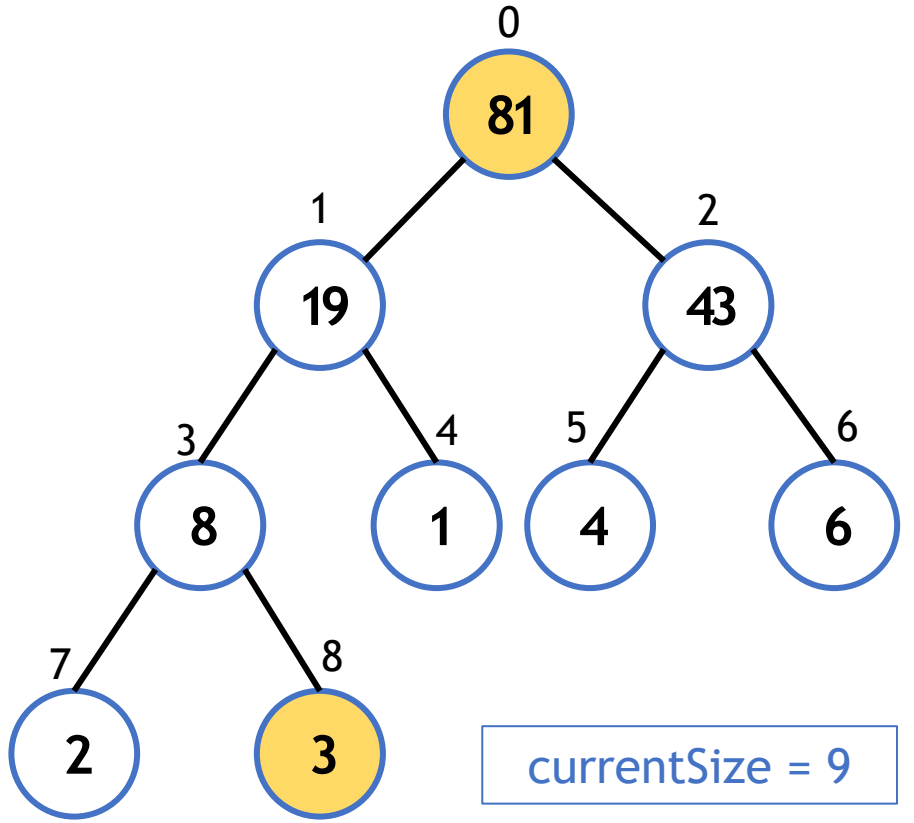
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

81	19	43	8	1	4	6	2	3
0	1	2	3	4	5	6	7	8



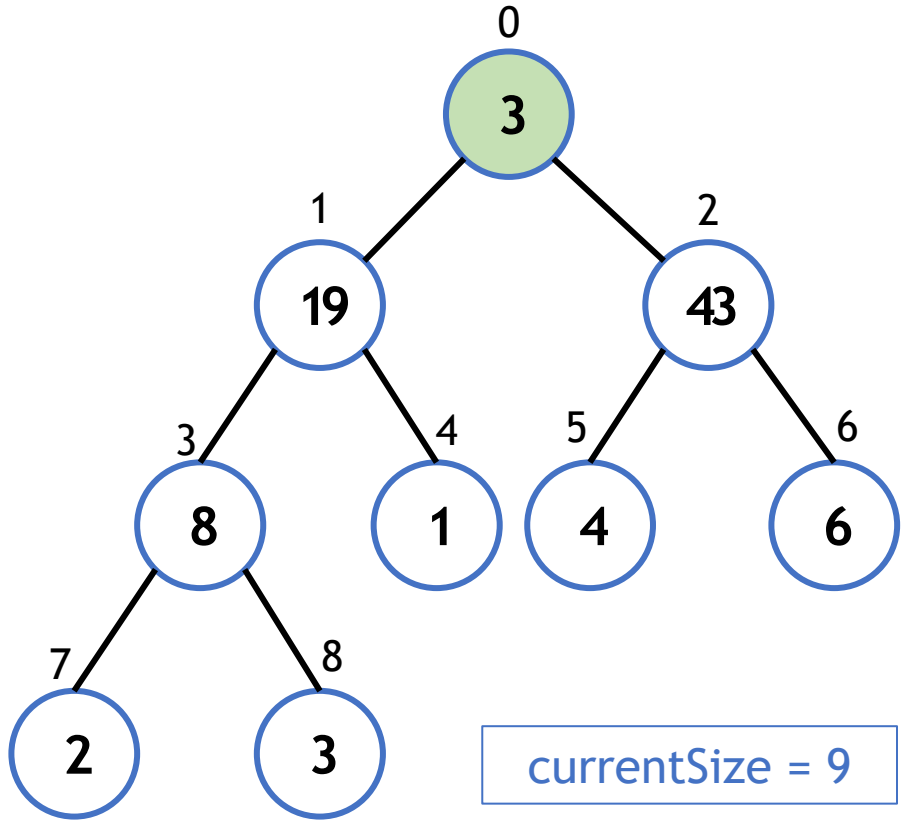
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

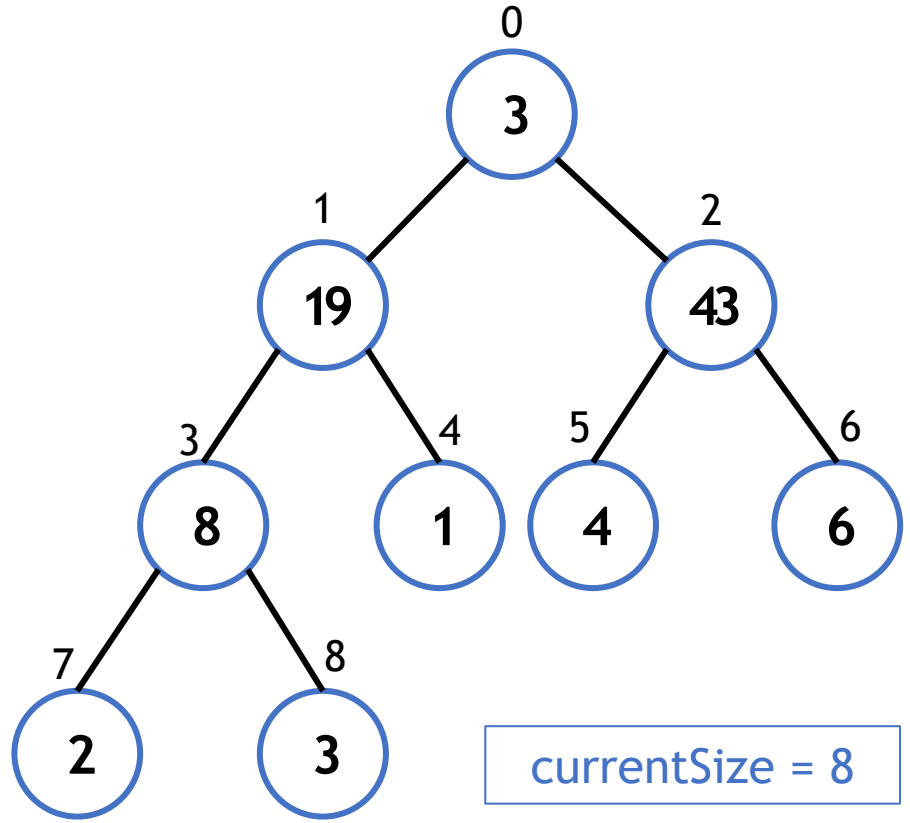
3	19	43	8	1	4	6	2	3
0	1	2	3	4	5	6	7	8



Heap: extractMax

```
A[0] = A[currentSize - 1];
currentSize = currentSize - 1;
MaxHeapify(0);
```

3	19	43	8	1	4	6	2	3
0	1	2	3	4	5	6	7	8



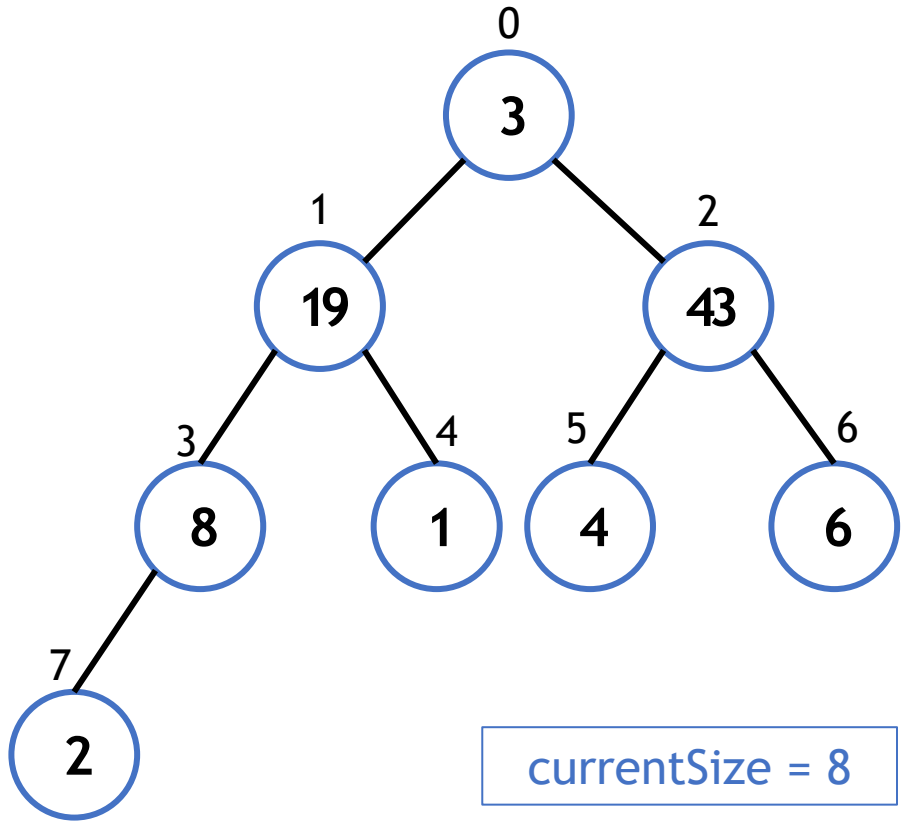
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

3	19	43	8	1	4	6	2	
0	1	2	3	4	5	6	7	8



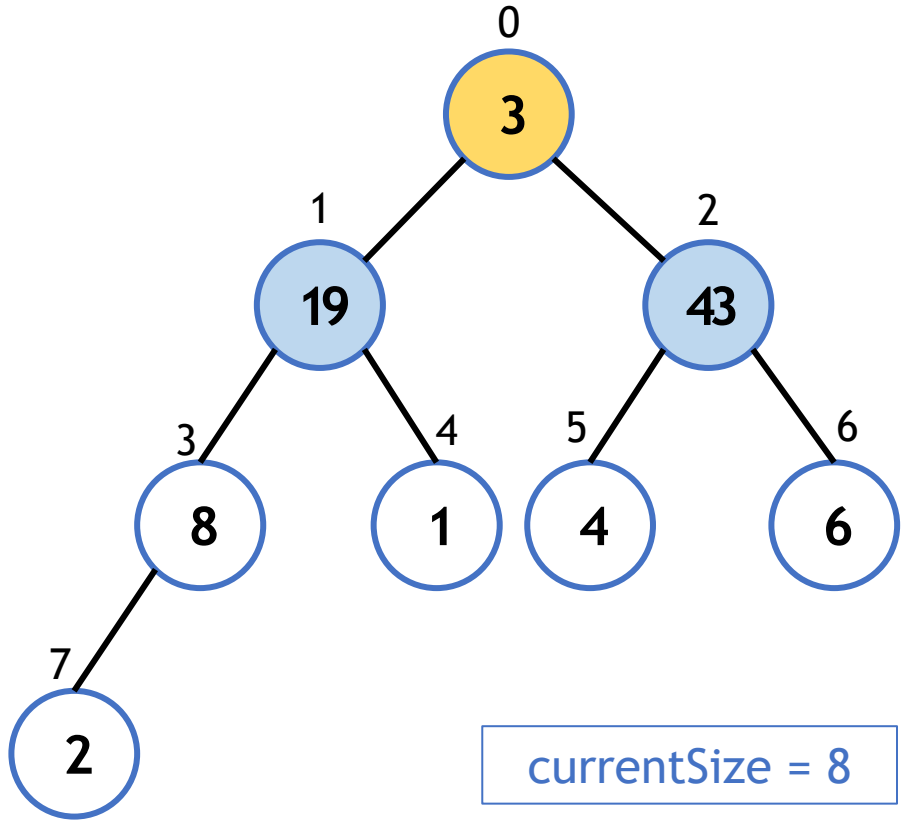
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

3	19	43	8	1	4	6	2	
0	1	2	3	4	5	6	7	8



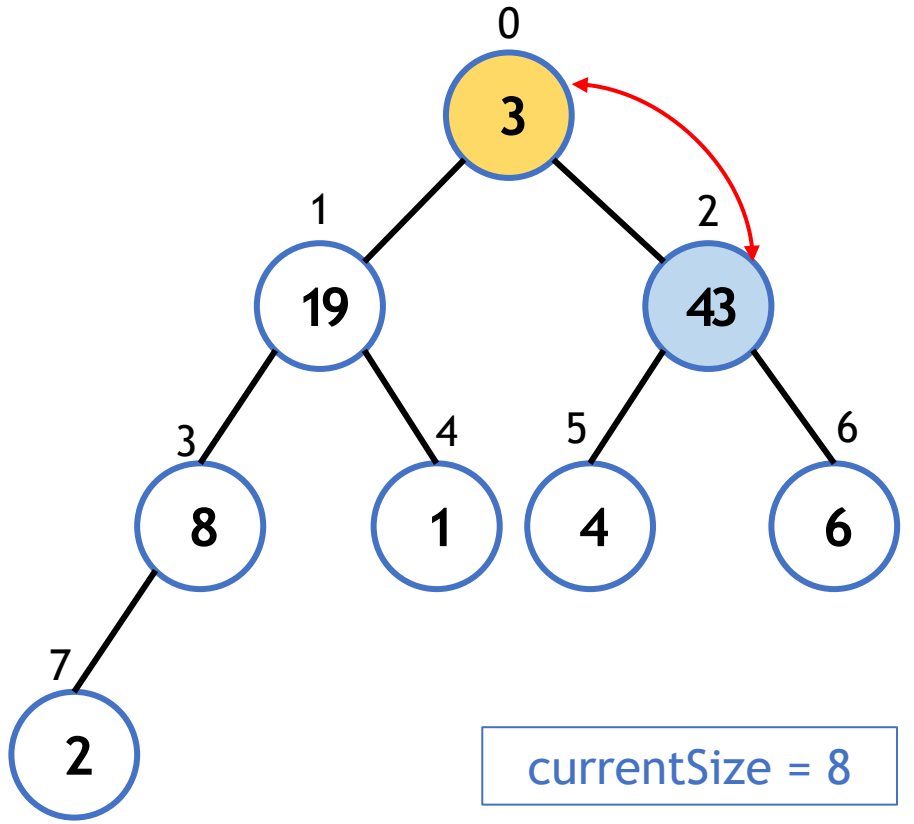
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

3	19	43	8	1	4	6	2	
0	1	2	3	4	5	6	7	8



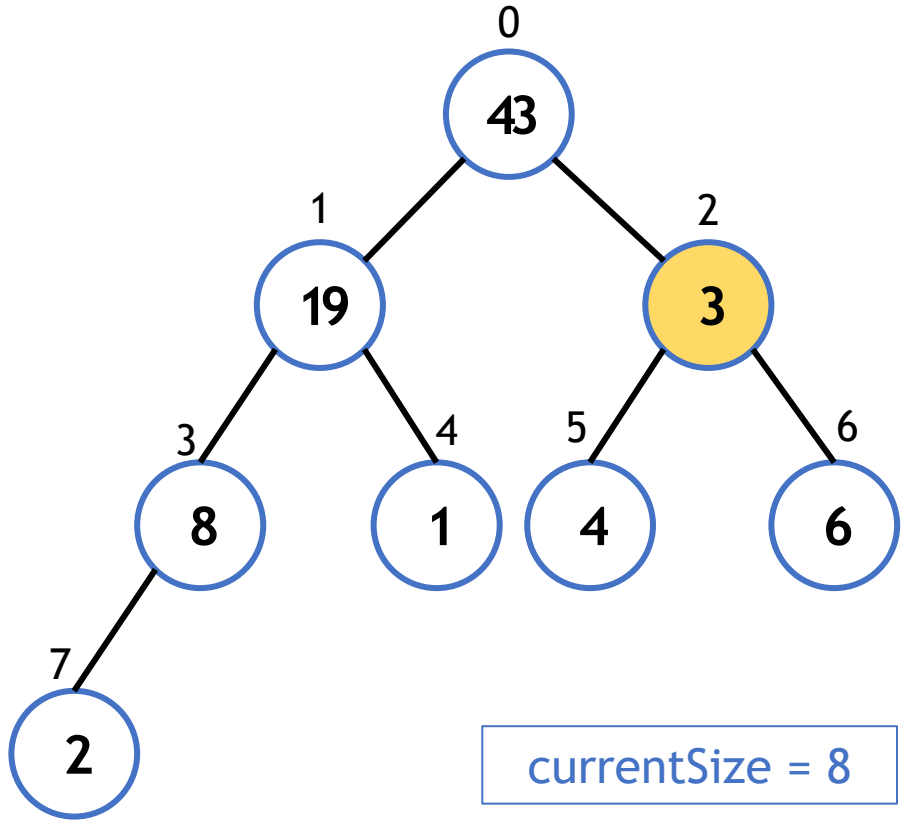
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

43	19	3	8	1	4	6	2	
0	1	2	3	4	5	6	7	8



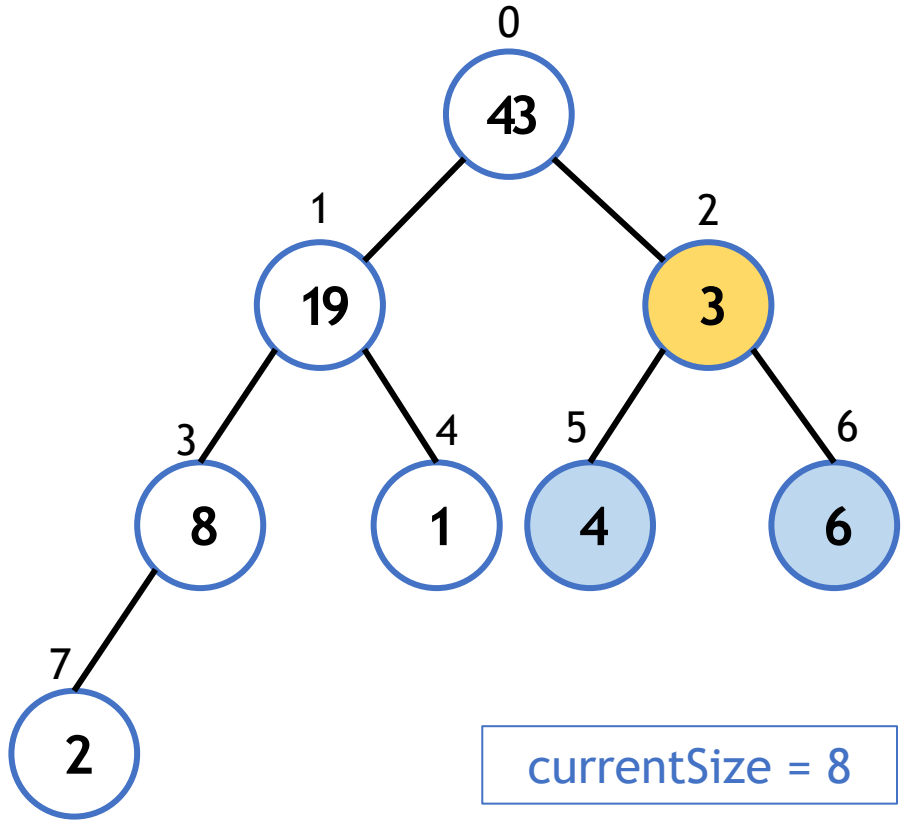
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

43	19	3	8	1	4	6	2	
0	1	2	3	4	5	6	7	8



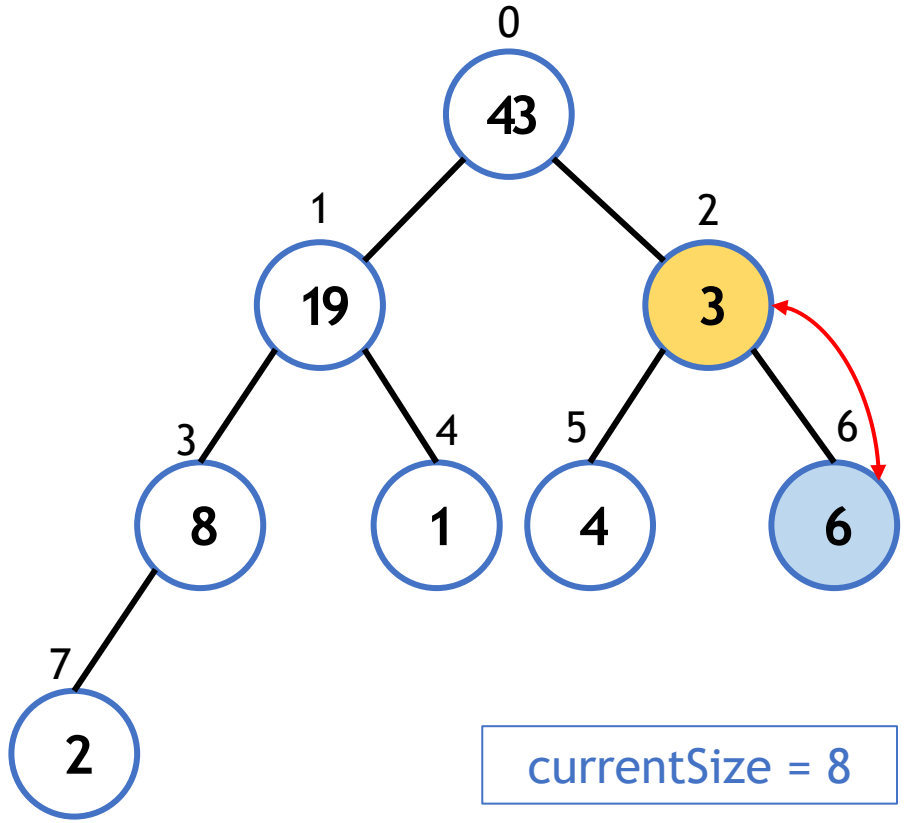
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

43	19	3	8	1	4	6	2	
0	1	2	3	4	5	6	7	8



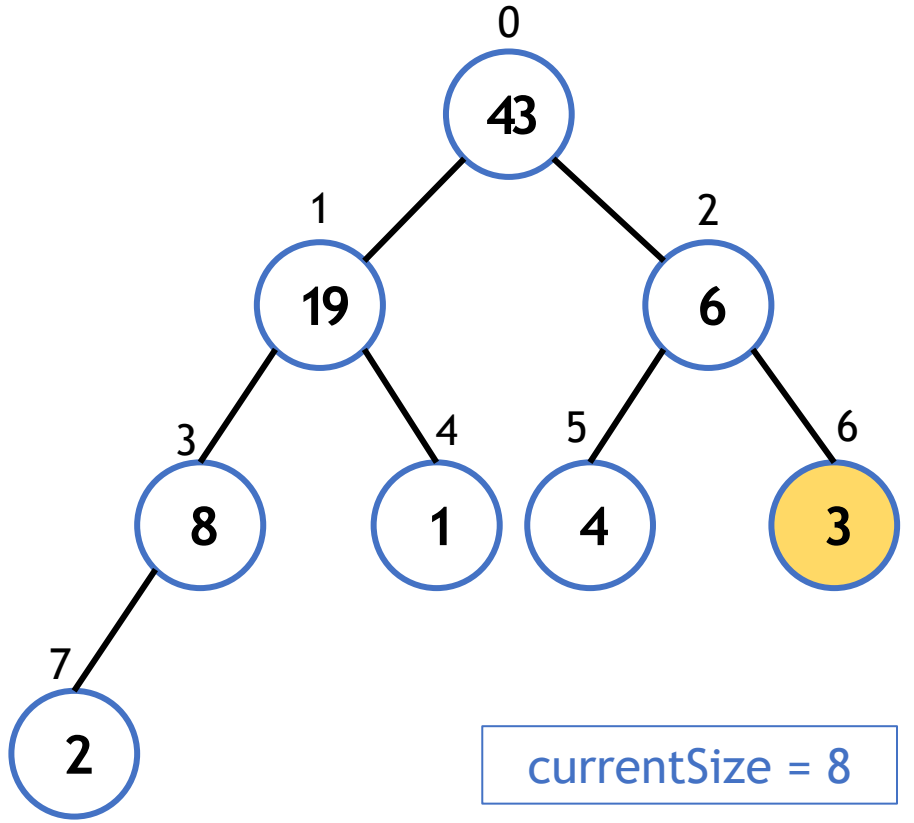
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

43	19	6	8	1	4	3	2	
0	1	2	3	4	5	6	7	8



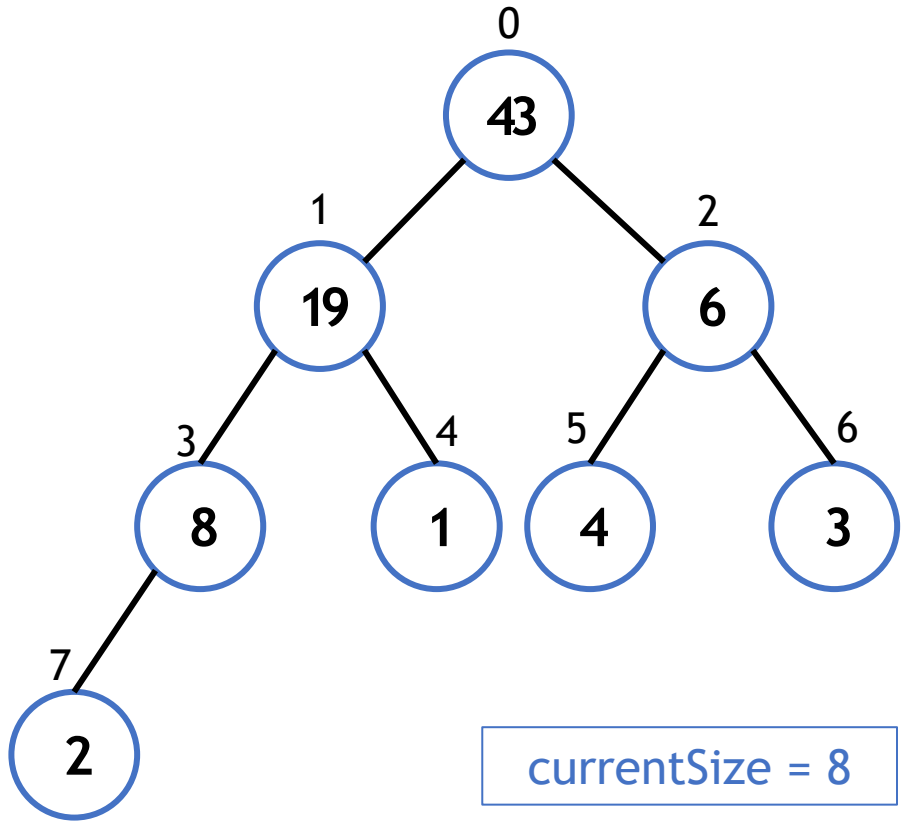
Heap: extractMax

```
A[0] = A[currentSize - 1];

currentSize = currentSize - 1;

MaxHeapify(0);
```

43	19	6	8	1	4	3	2	
0	1	2	3	4	5	6	7	8



Note 1: Ensure enough space in the array

Note 2: This is for a Max-Heap

Heap: insertElement

Insert a new element **(key, data)** ***such that heap invariant is maintained***

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].data = data;
A[i].key = key;

while (i != 0 && A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```

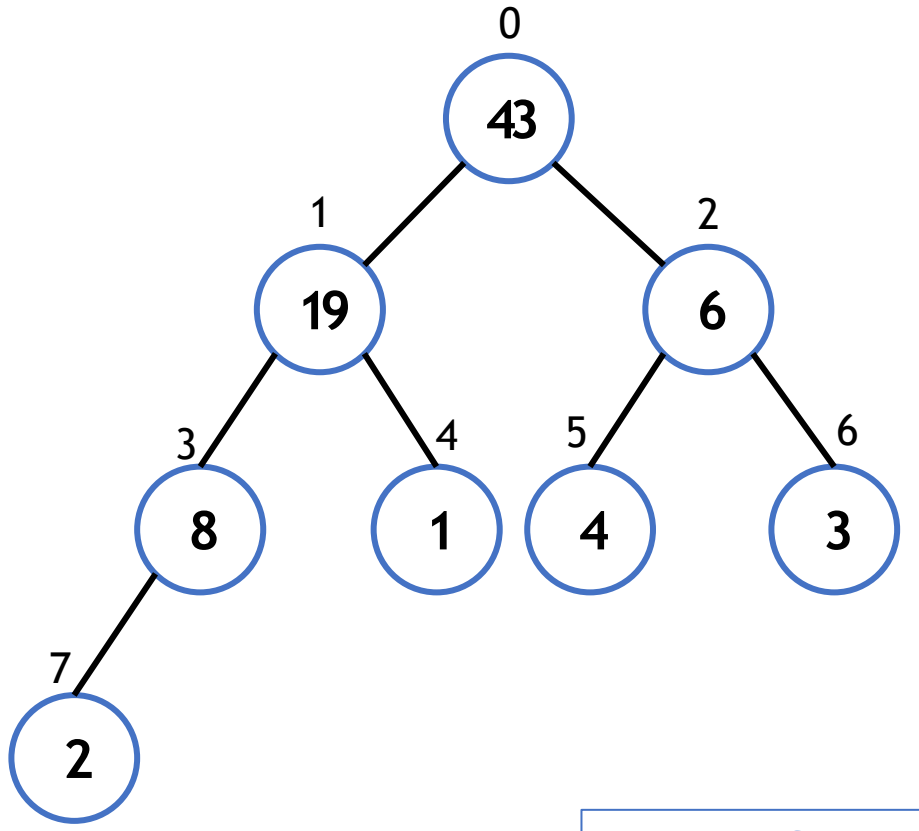
Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```

Insert item with priority: 81



currentSize = 8

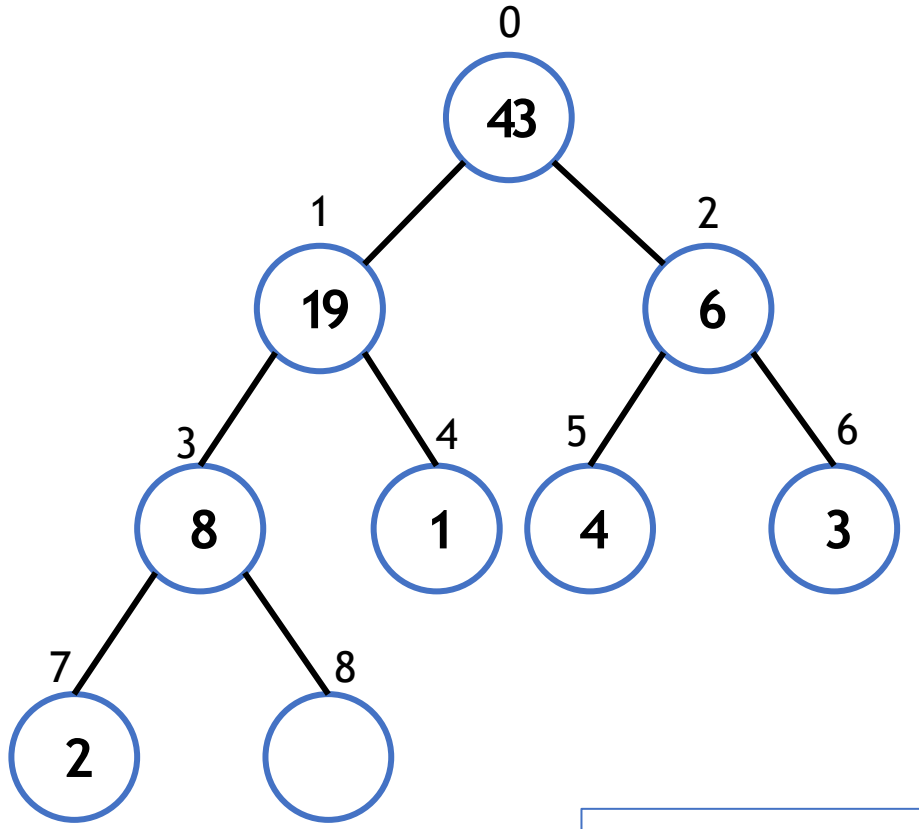
43	19	6	8	1	4	3	2	
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

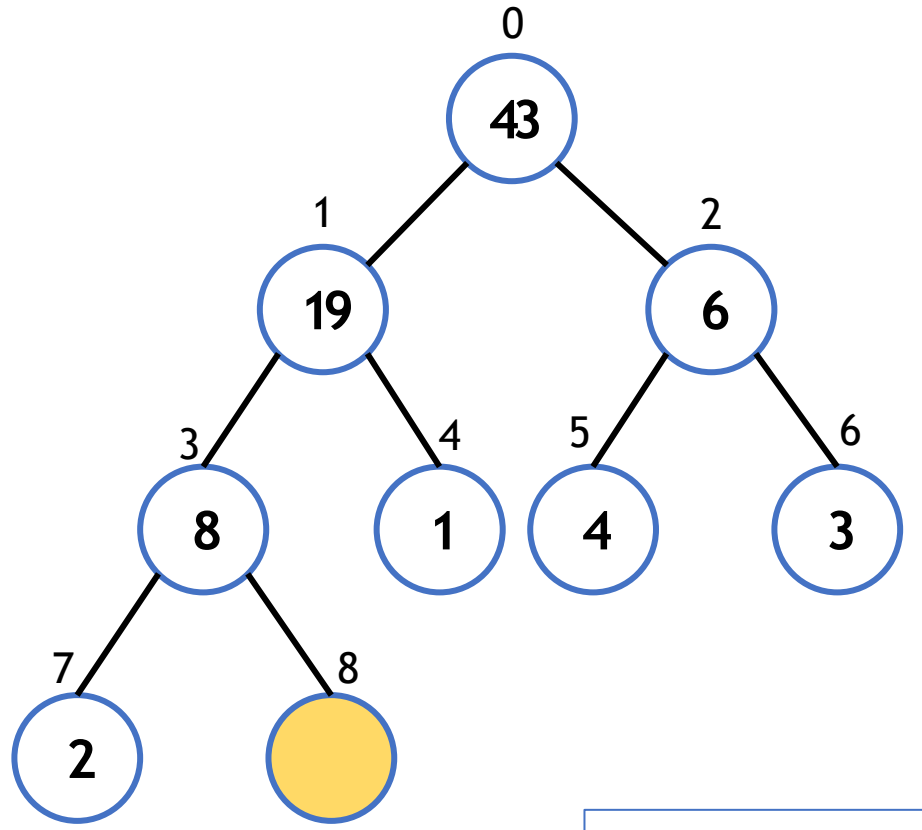
43	19	6	8	1	4	3	2	
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

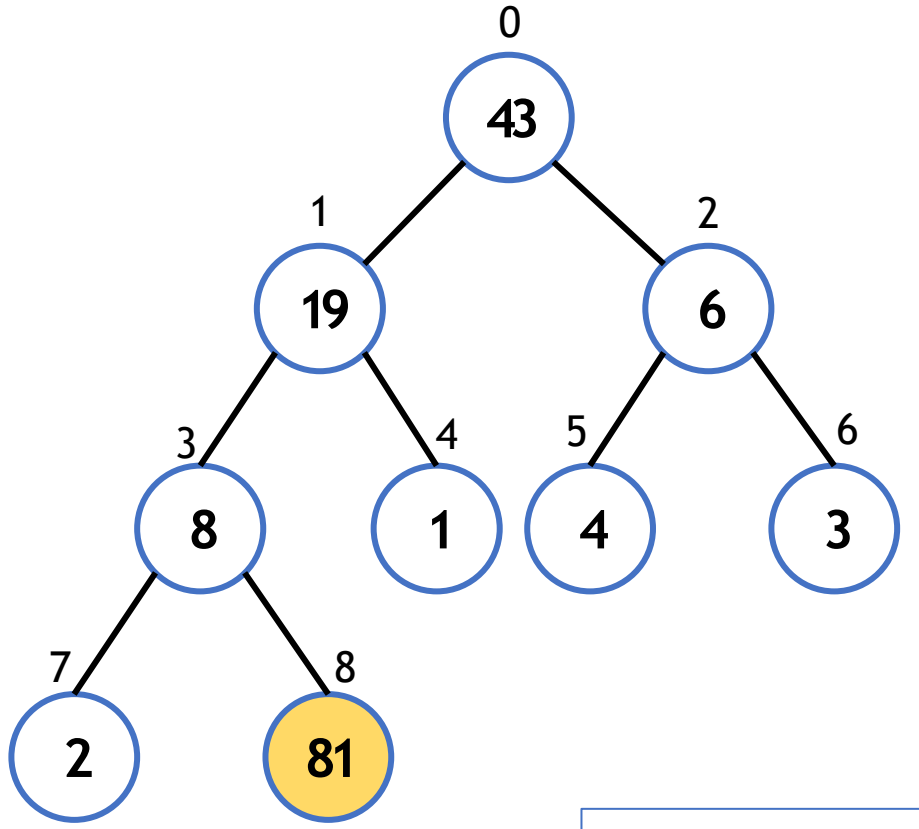
43	19	6	8	1	4	3	2	
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

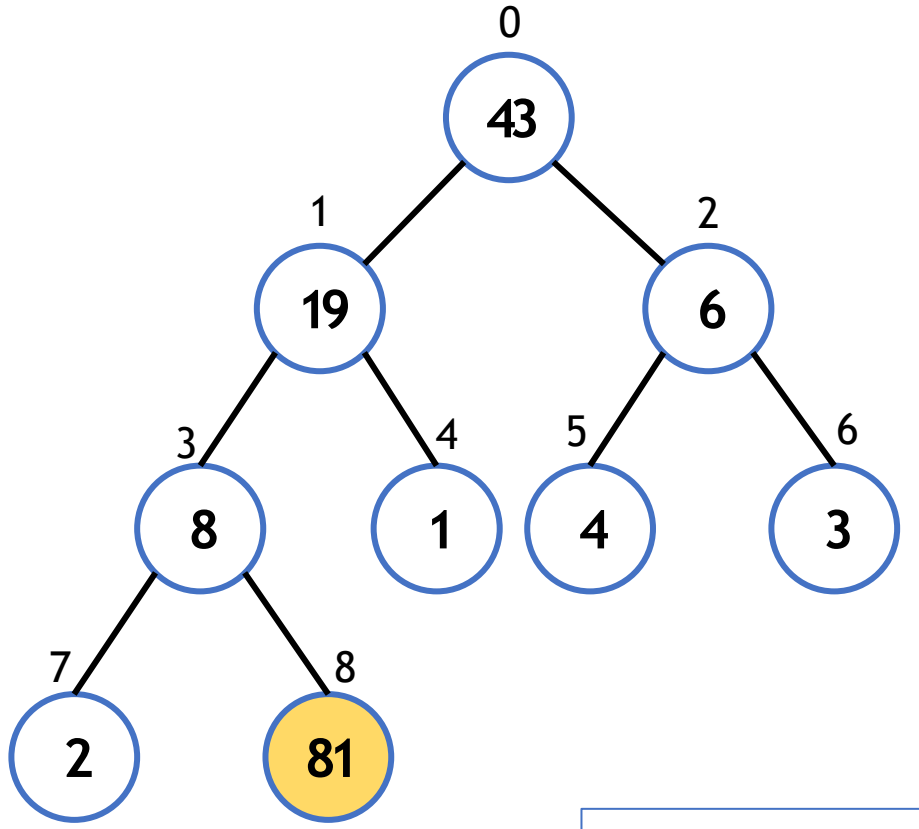
43	19	6	8	1	4	3	2	81
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

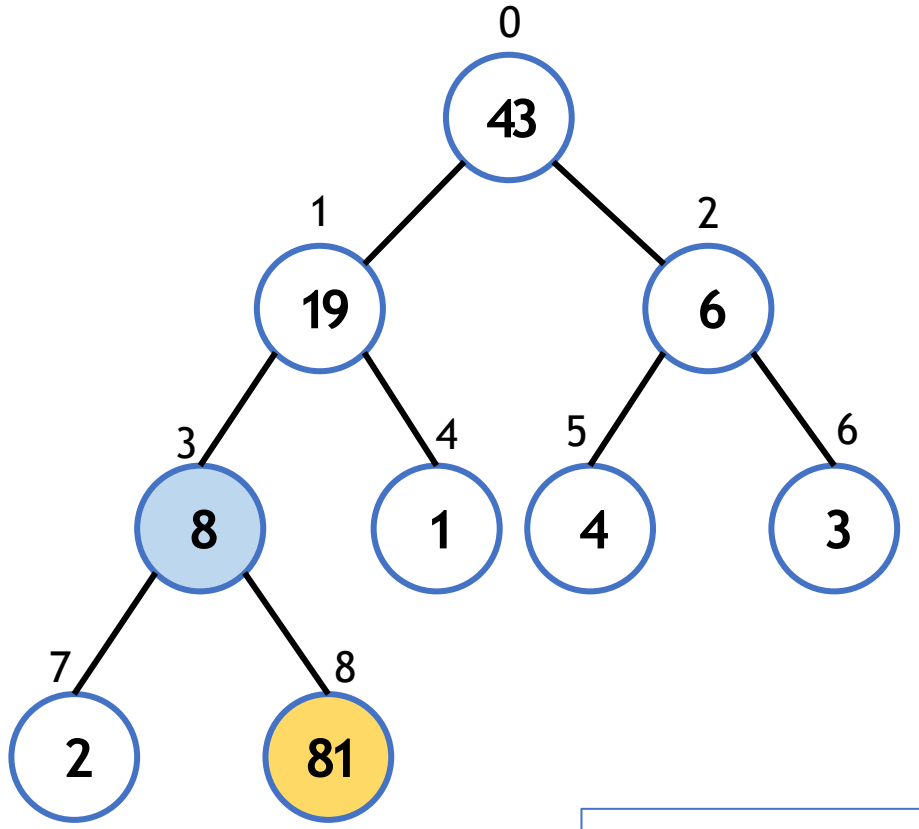
43	19	6	8	1	4	3	2	81
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

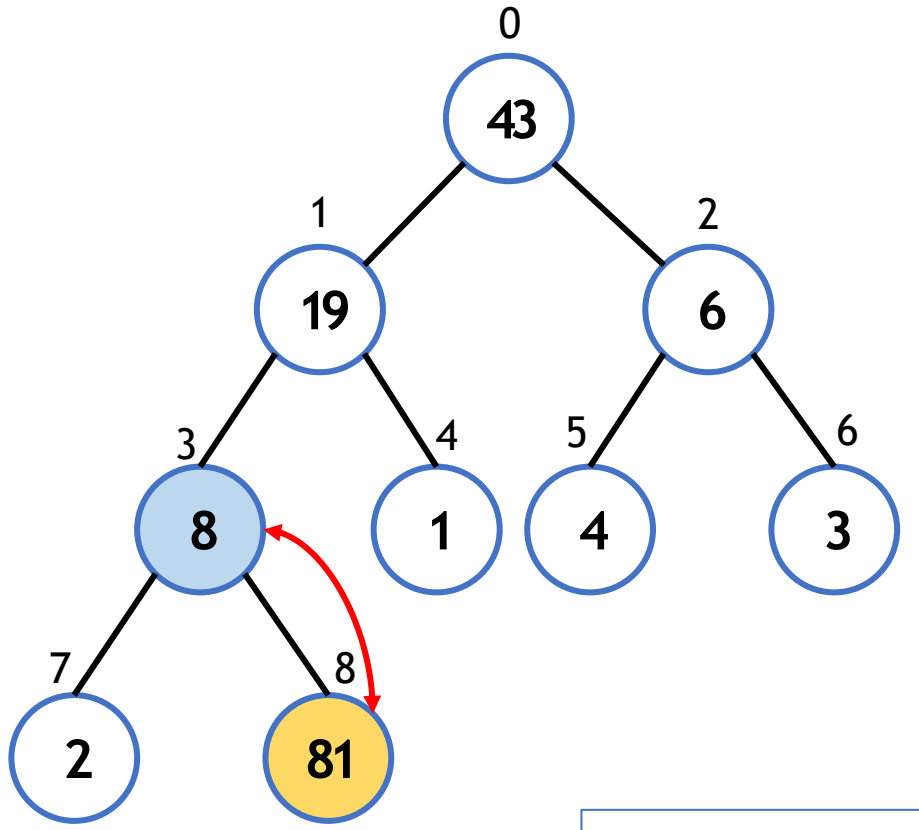
43	19	6	8	1	4	3	2	81
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

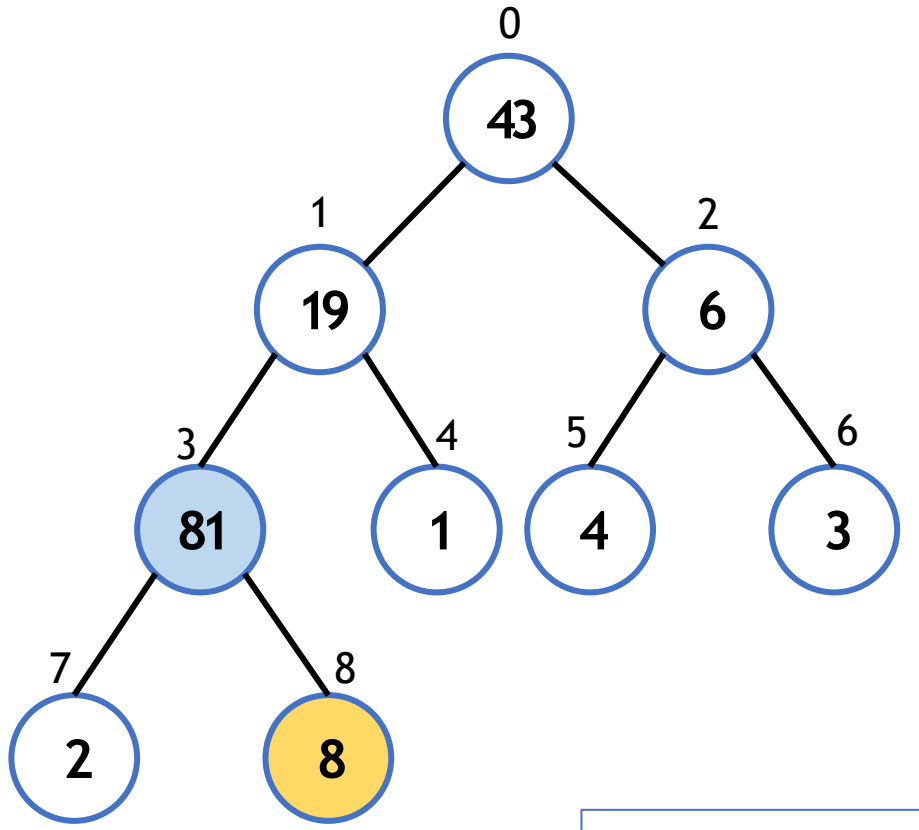
43	19	6	8	1	4	3	2	81
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

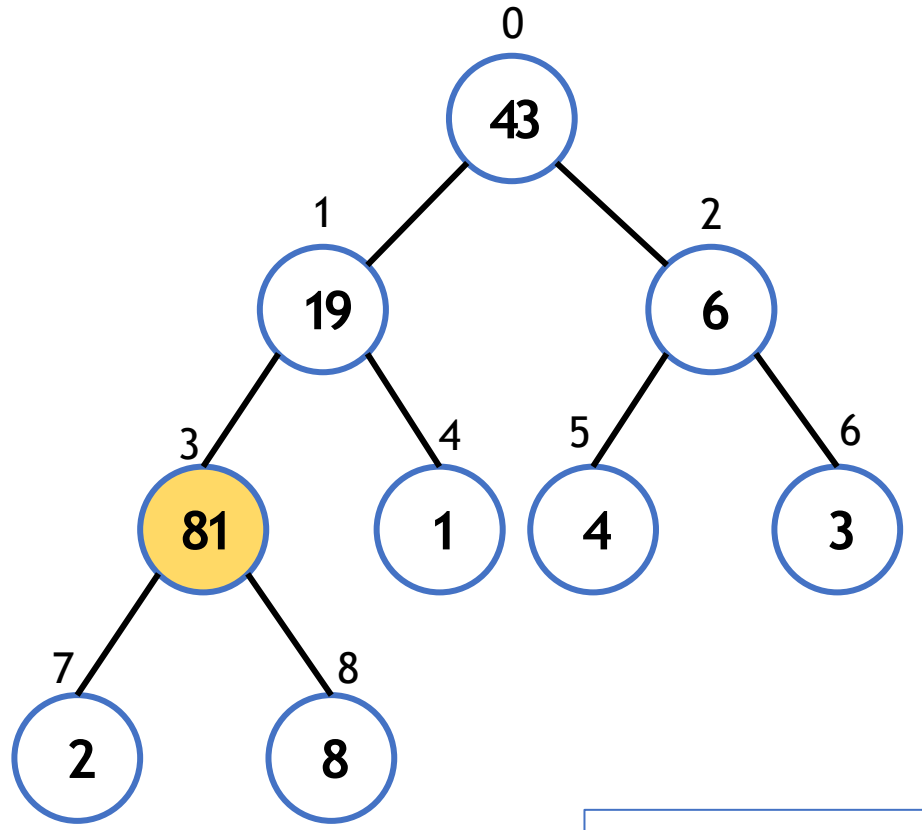
43	19	6	81	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

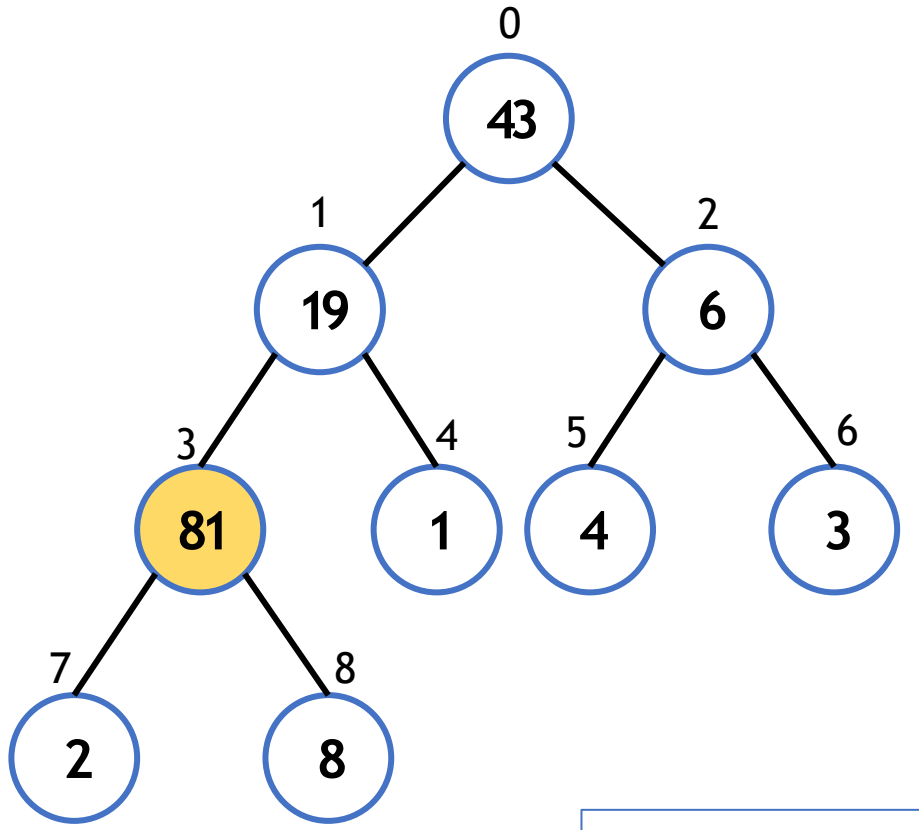
43	19	6	81	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

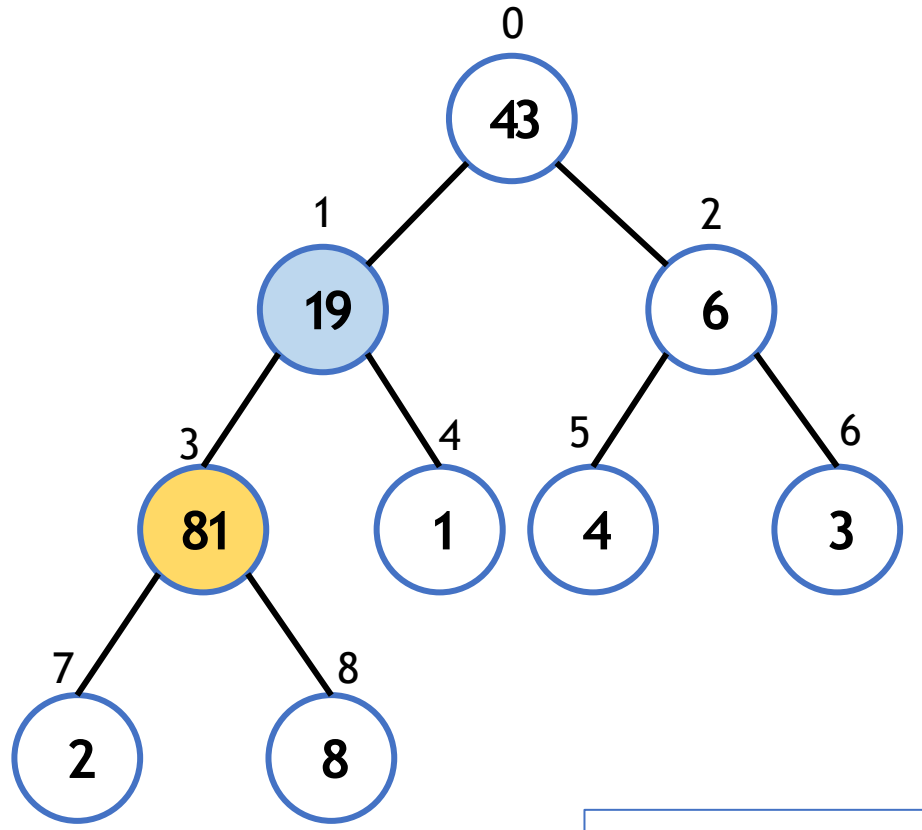
43	19	6	81	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

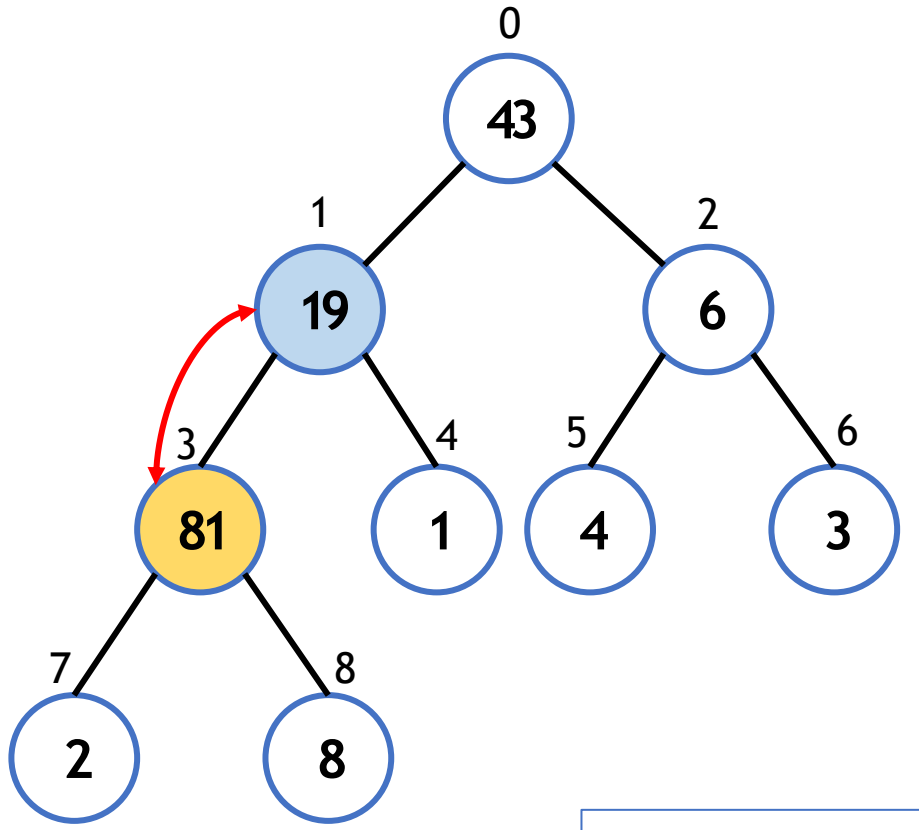
43	19	6	81	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

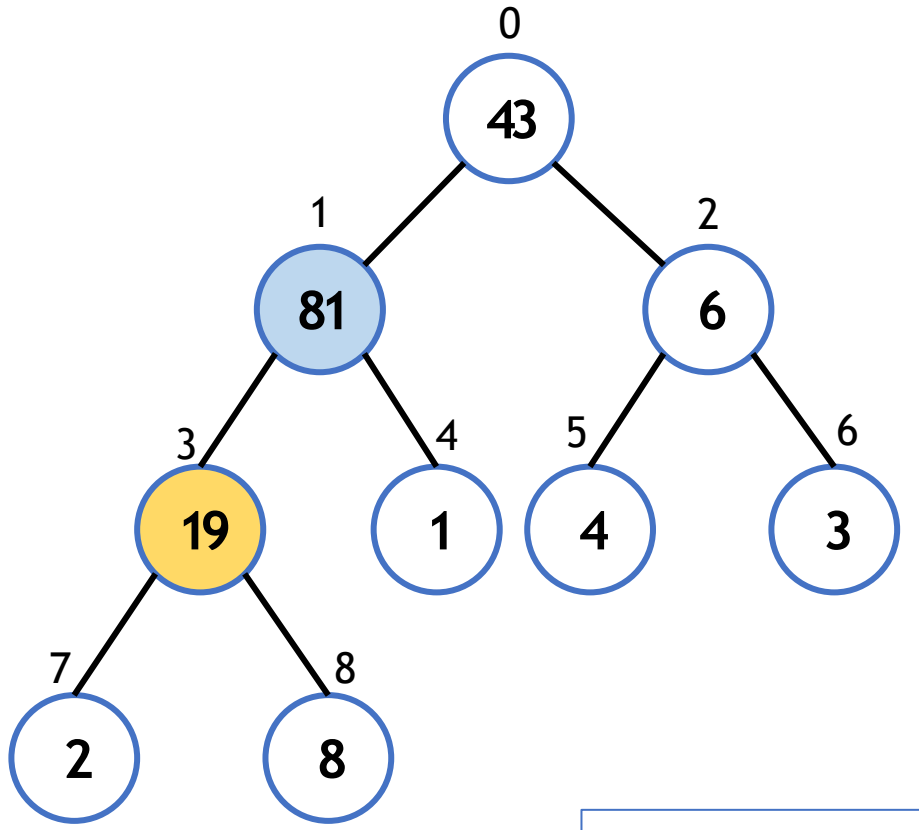
43	19	6	81	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

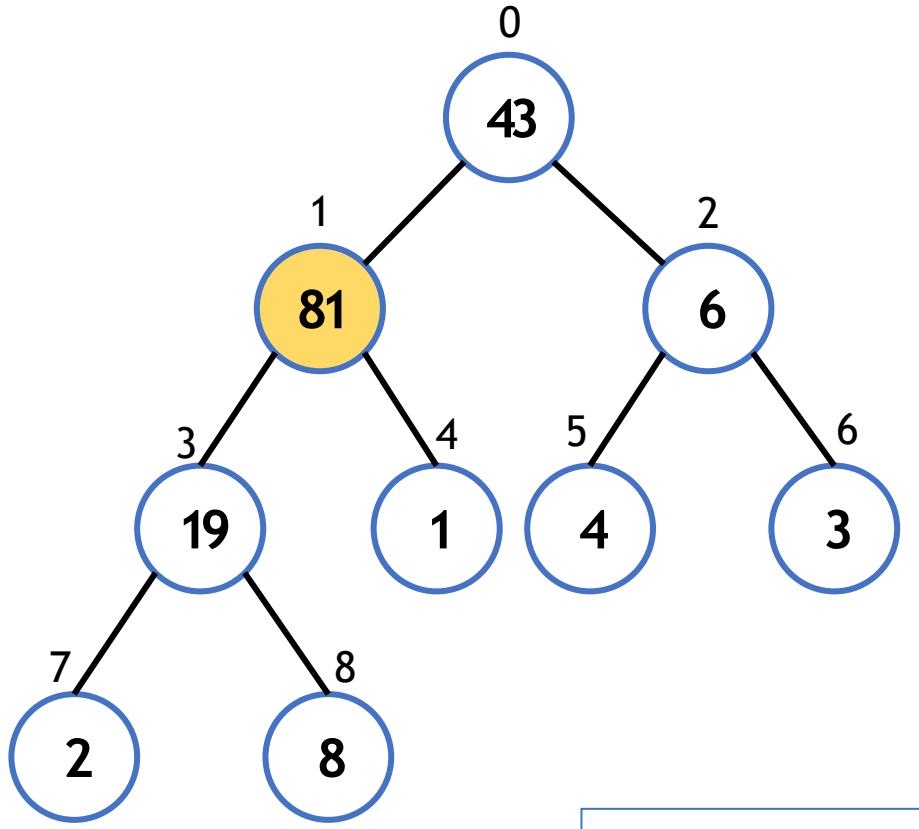
43	81	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

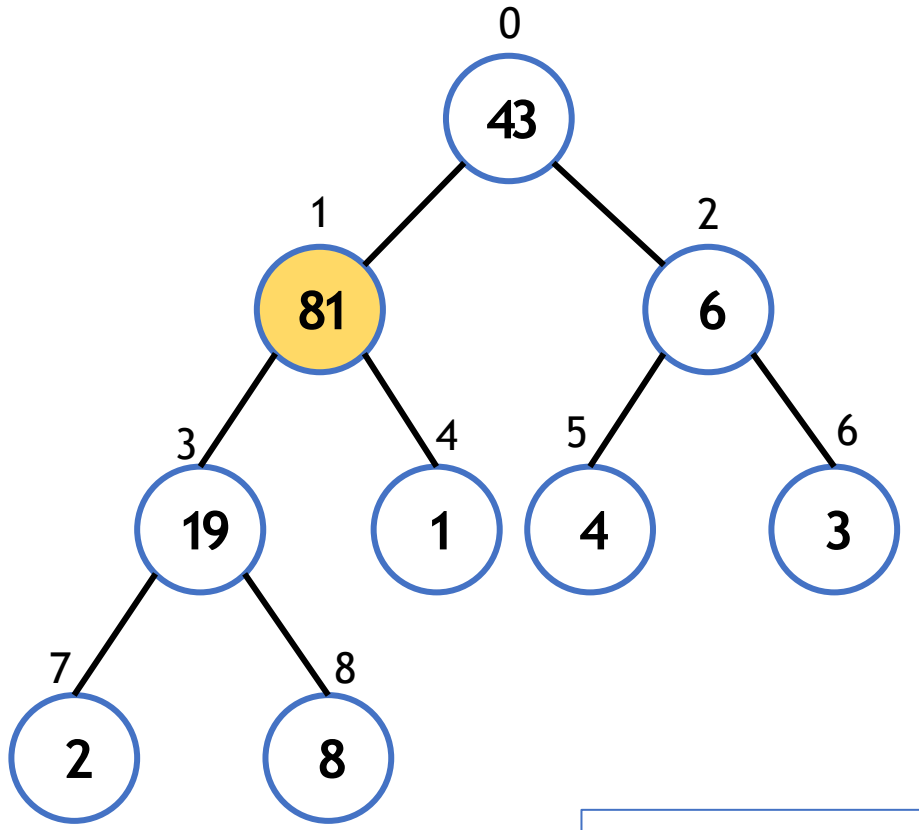
43	81	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

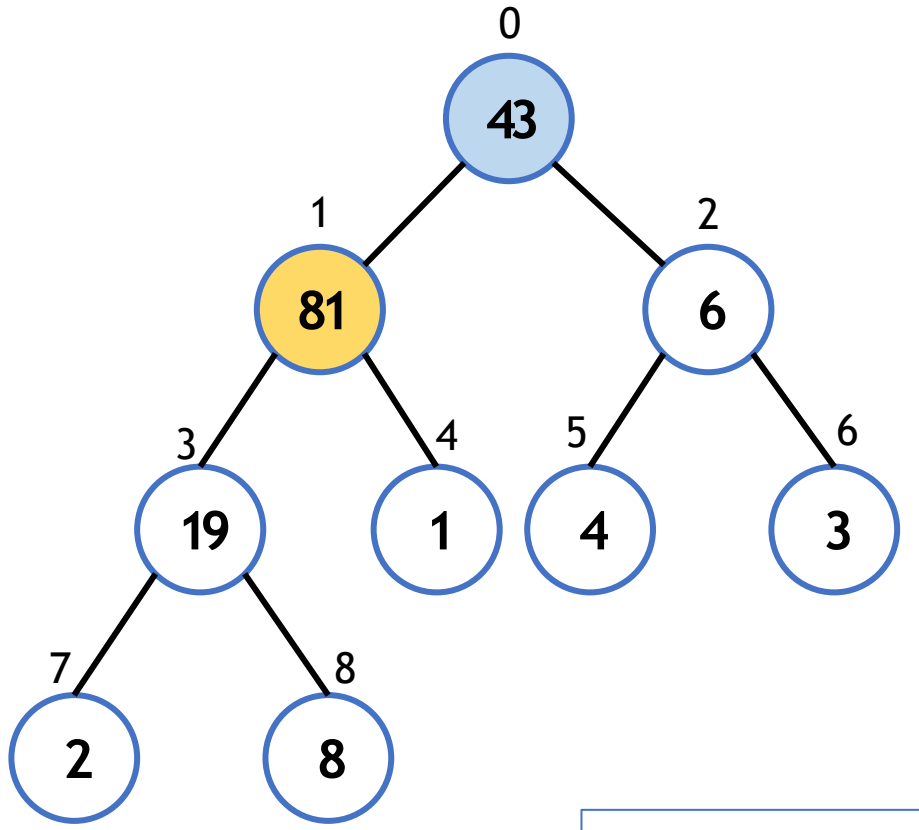
43	81	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

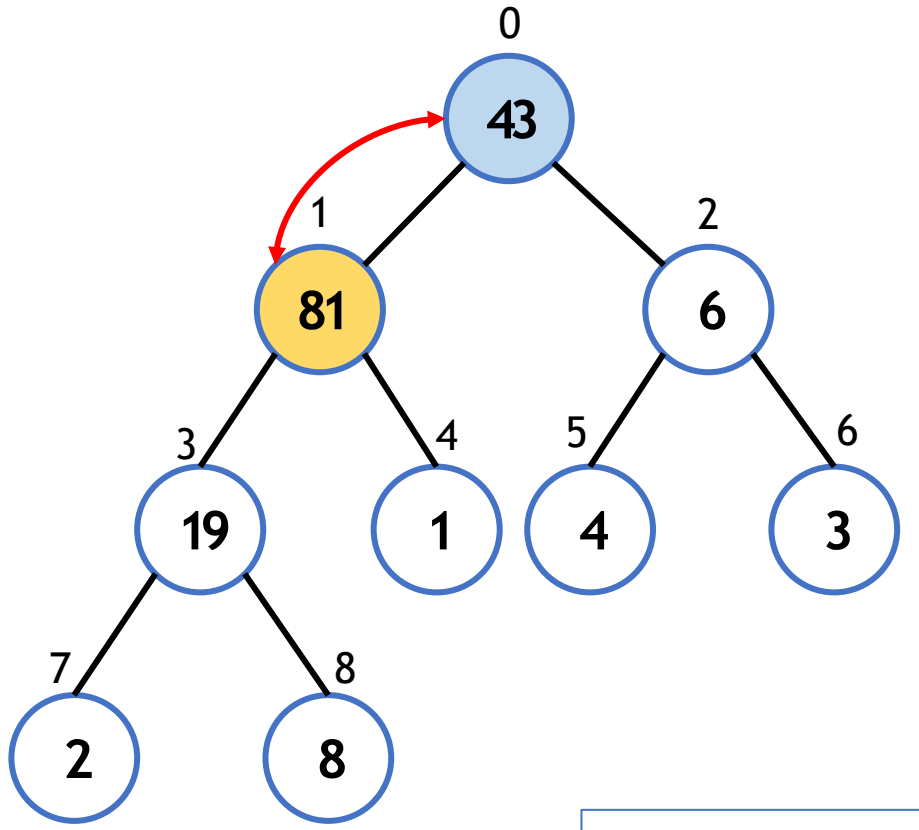
43	81	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

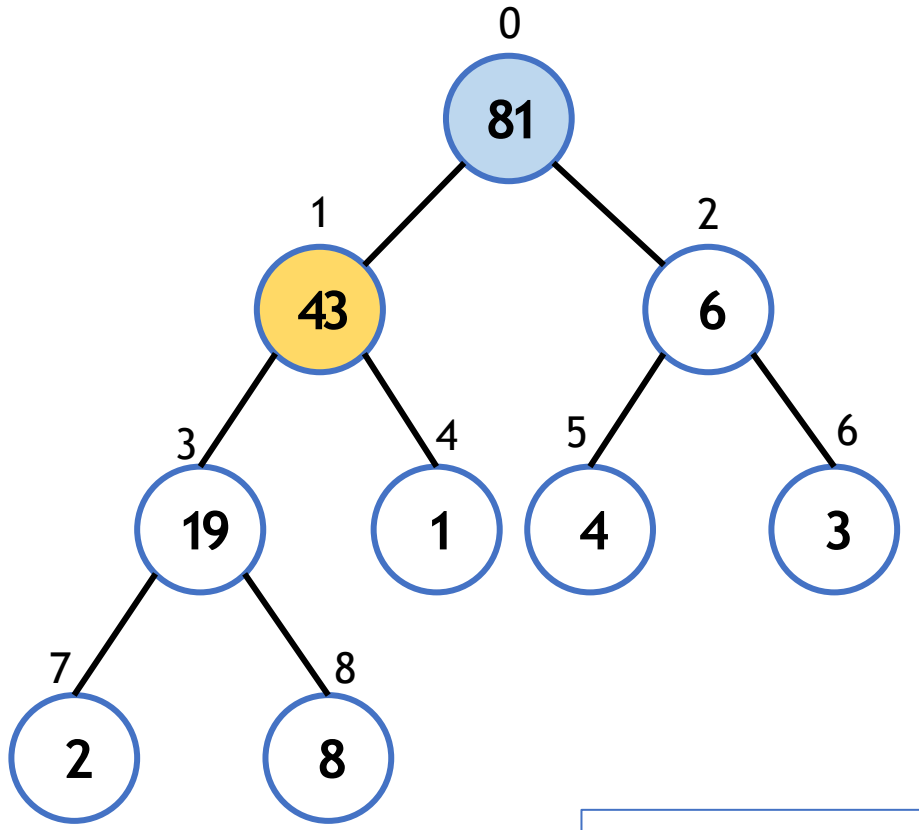
43	81	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

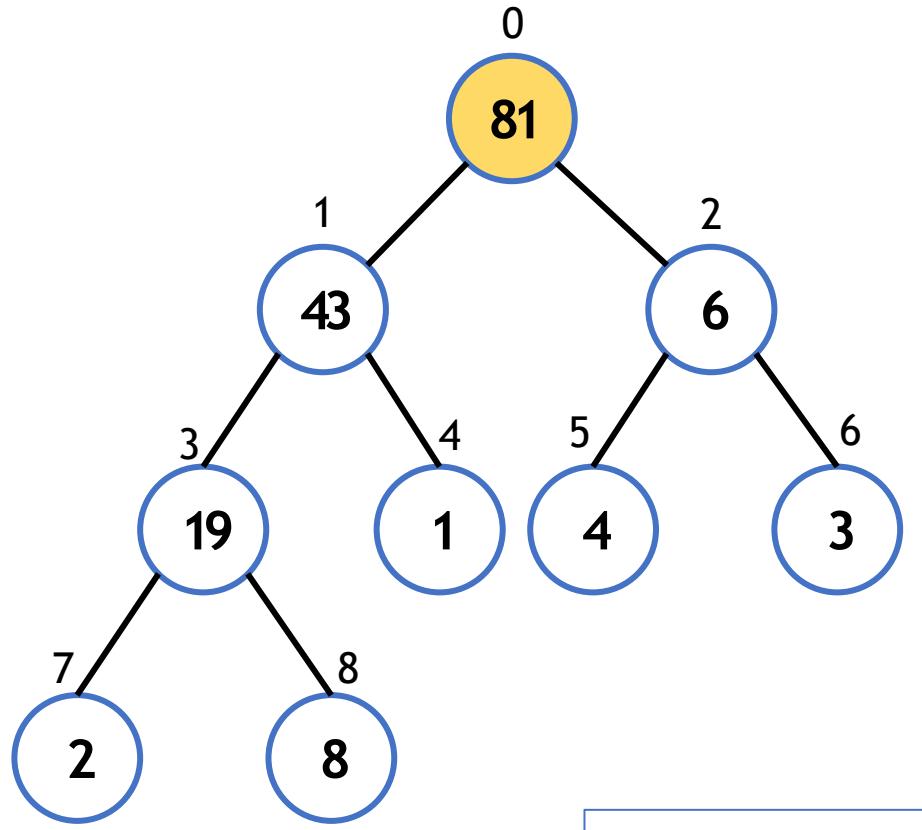
81	43	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

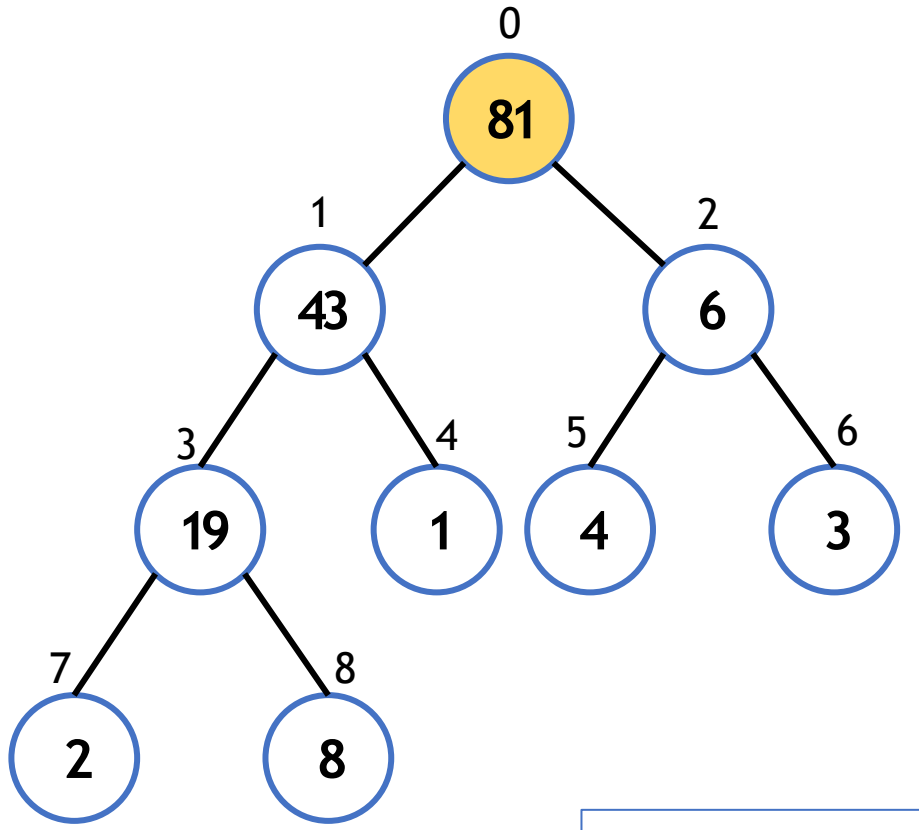
81	43	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

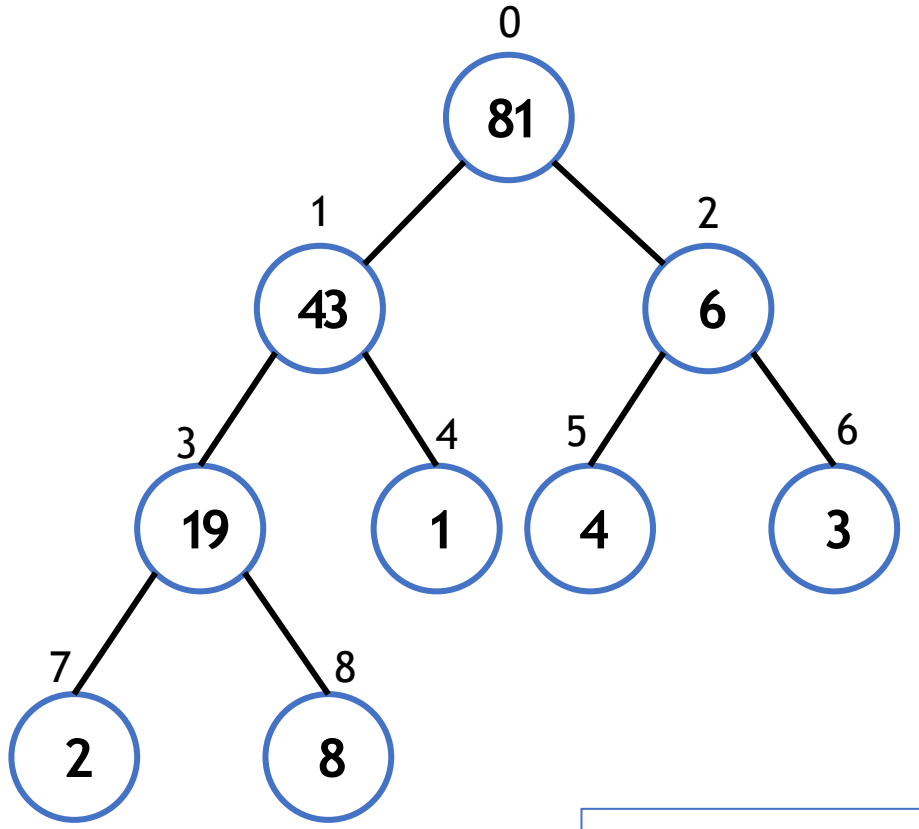
81	43	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

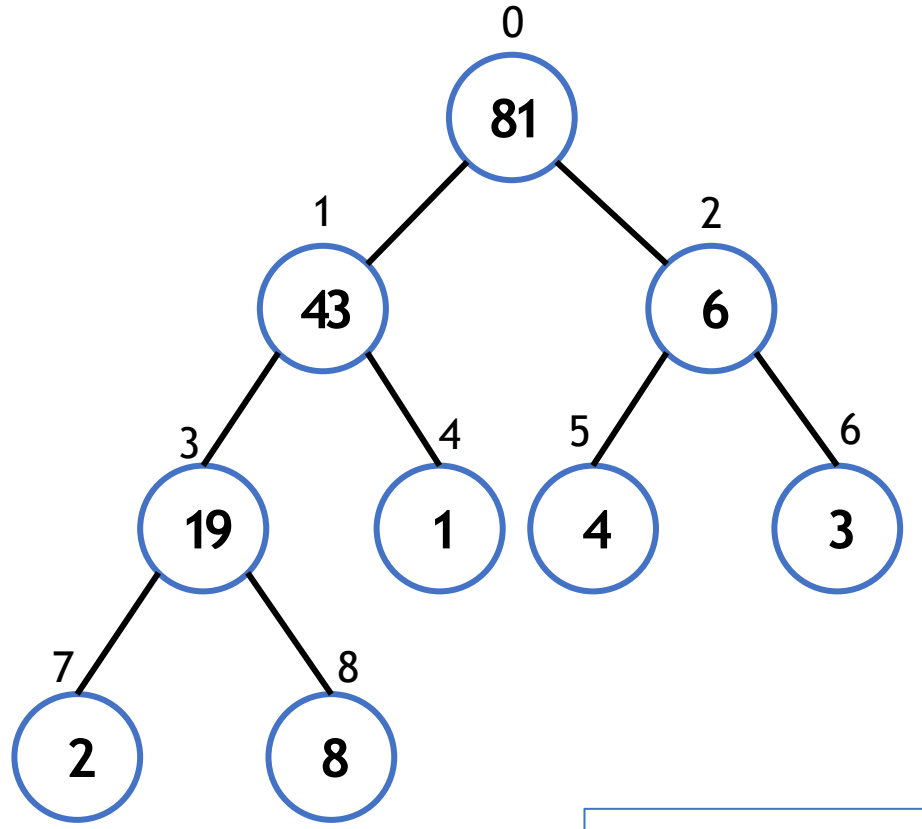
81	43	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: insertElement

```
currentSize = currentSize + 1;
i = currentSize - 1;

A[i].key = key; A[i].data = data;

while(i != 0 &&
          A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}
```



currentSize = 9

81	43	6	19	1	4	3	2	8
0	1	2	3	4	5	6	7	8

Heap: deleteElement

```
// Find index i for item to be deleted
A[i].key = INT_MAX;

while(i != 0 && A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    i = parent(i);
}

extractMax()
```

Heap: BuildMaxHeap

Given an unsorted tree, convert it into a heap

```
for(int i = currentSize/2 - 1; i >= 0; i--) {
    MaxHeapify(i);
}
```

Start Heapifying at Last non-leaf node

Keep heapifying until root is reached

Heap: Heapsort

Given an unsorted array, sort it in descending order

```
A_heap = BuildMaxHeap(A)
B = vector();
while(currentSize > 0)
    B.push_back(A_heap.extractMax())
```

Sort a Max-Heap in Ascending Order?
Use a Stack, or Print vector in reverse

Heap: Worst-case complexity

For N nodes a binary tree, ***recall***: **#levels ≈ $\log_2 N$**

`extractMax`	$O(\log_2 N)$
`insertElement`	$O(\log_2 N)$
`deleteElement`	$O(\log_2 N)$
`searchElement`	$O(N)$

Exercise

Heap

Exercise: Silver

extractMax()

```
A[0] = A[currentSize - 1];
currentSize = currentSize - 1;
MaxHeapify(0);
```

Implement:

`extractMin(), Heapify(int)`

Make sure l, r children are in range

```
if(l < currentSize && r < currentSize)
```

Notes

- Take a look at the Github page to see how MaxHeapify is done

  Tweak Max-Heapify to suit Min

- Remember it's a Min-Heap here

Exercise: Gold

Implement:

deleteKey(int)

```
// Find index i for item to be deleted
A[i].key = INT_MAX;

while(i != 0 && A[parent(i)].key < A[i].key)
{
    swap(A[i], A[parent(i)]);
    index = parent(i);
}
```

extractMax()