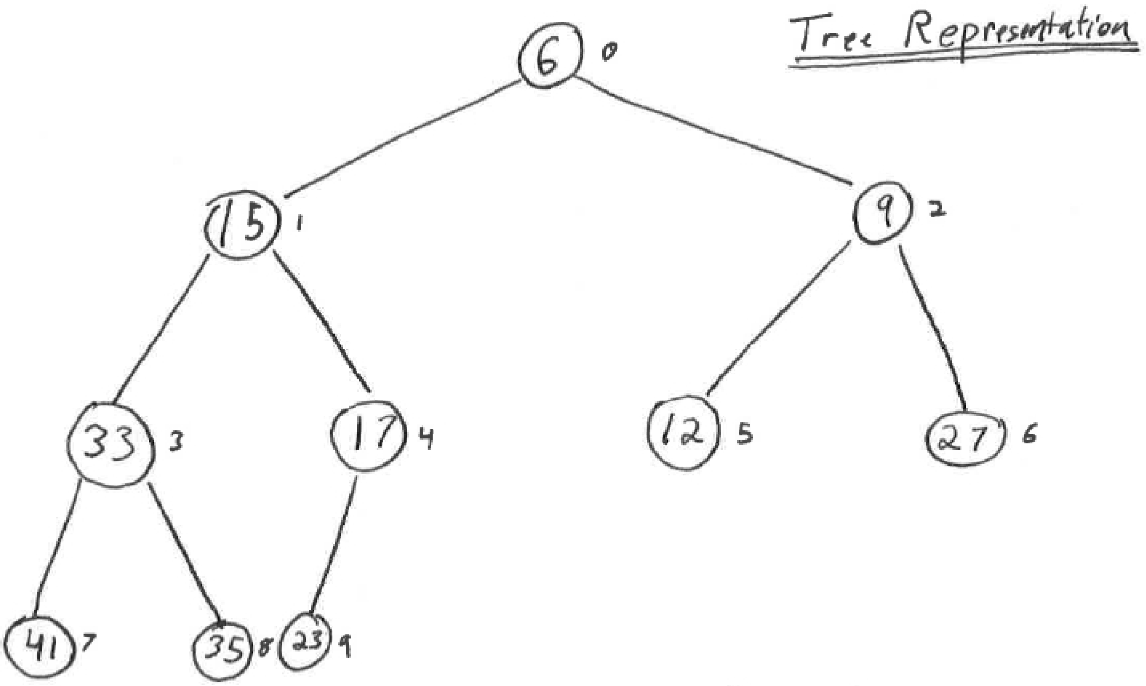
Binary Heaps

A **binary heap** is a datastructure for the implementation of **priority queues.** A **priority queue** allows for efficient implementation of **Find Min** (or **Find Max)** and **Delete Min** (or **Delete Max**). In other words, **priority queues** are good for operations where we want to quickly insert data, and find the smallest or largest element in the datastructure.

Binary Heaps are technically Binary Trees, but they are **NOT** Binary Search Trees. They have a somewhat different set of properties.

There are **two types of heaps**.
Min-Heaps (designed for Find Min)
Max-Heaps (designed for Find Max)

The following is an example of a **Binary Min-Heap** as implemented using an Array Representation.



Array Representation

A:	6	15	9	33	17	12	27	41	35	23
	0	1	2	3	4	5	6	7	8	9

<u>**Binary Min-Heap Properties**</u>

Binary Tree Property

Every Element in a binary-heap has up to two children

Min Heap Property

Every Element in a Min-Heap is smaller than **both** of it's children

Completeness Property

A binary heap is a complete binary tree, every level of the heap is full except possibly the last level. And the last level has children all the way to the left.

Binary Max-Heaps are similar except that they have a **Max-Heap Property**, that each element is larger than both it's children.

The **Completeness Property** allows binary heaps to be implemented as an array.

In order to implement the min-heap as an array we write down the elements from left to right, layer by layer (top to bottom) with the root (minimum) always at index 0 of the array.

We typically do not use pointers to implement a binary heap. Even though the heap is logically hierarchical, it is not constructed by means of pointers. Rather, we use an Array Representation for the physical implementation, even though logically it can be thought of as a Tree Representation.

For programming languages where the array indices start at 0 (rather than 1), any node at index 'i' has children:

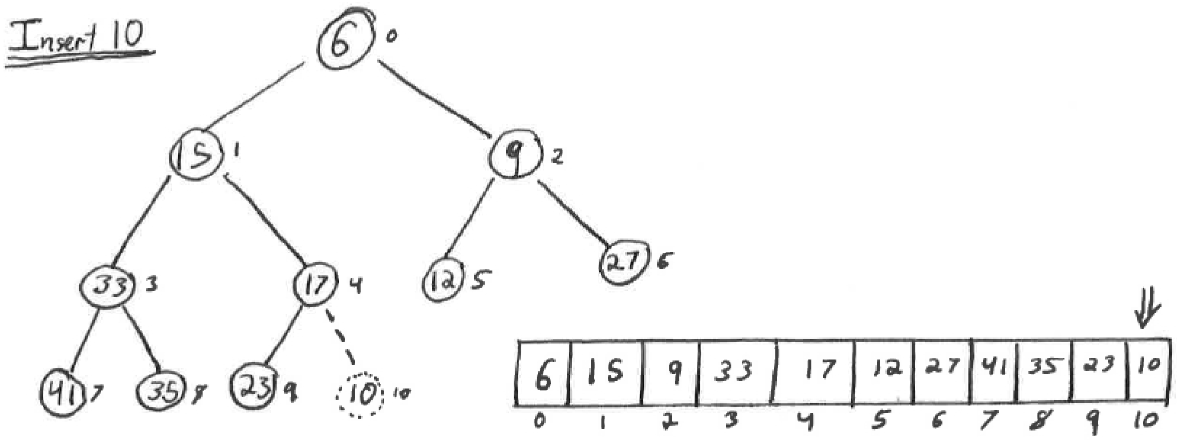
left child at index	2i + 1
right child at index	2i + 2
parent at index	(i-1) / 2

A Binary Min-Heap has the following asymptotic runtimes for operations

Insert	O lg N
RemoveMin	O lg N
Build Heap	O N

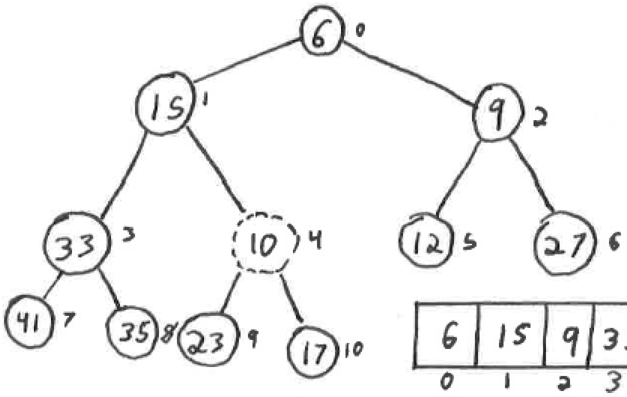
In order to insert, we place the new node at array index A[N], this places the node all the way to the right of the bottom layer, thereby maintaining the "Completeness" property. However, this may violate the Min-Heap property, so we need to "bubble up" the node by iteratively compare the newly inserted node with it's parent. So long as the node has a parent, and the parent is larger than the new node, we must iteratively swap the new node with it's parent until termination.

The following example shows an insertion.



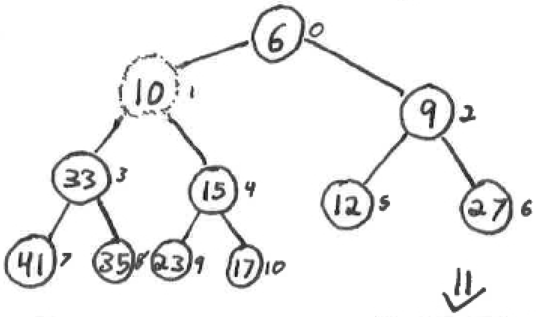
6	15	9	33	17	12	27	41	35	23	10
0	1	2	3	4	5	6	7	8	9	10

Step 1 10 added to lower right position.
Completeness is satisfied, but MinHeap property is not.



6	15	9	33	10	12	27	41	35	23	17
0	1	2	3	4	5	6	7	8	9	10

Step 2 10 swapped ~~#~~ with perent 17, because 10 < 17



6	10	9	33	15	12	27	41	35	23	17
0	1	2	3	4	5	6	7	8	9	10

Step 3 10 swapped with parent 15, because 10< 15.

The following is the pseudocode for insert.

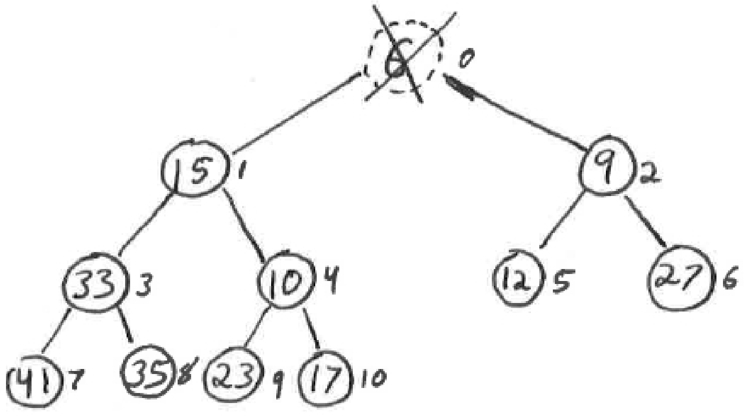
```
Insert (int *A, int N, int val)
{
    A[N] = val
    N++
    i = N-1
    p = (i-1)/2
    while (p>0 && A[i] < A[p])
    {
        temp = A[i]
        A[i] = A[p]
        A[p] = temp

        i=p
        p = (i-1)/2
    }
}
```

RemoveMin is a function works in the other direction. The min is always the root of the heap. But the only node that we can physically remove is A[N-1] which is the rightmost node in the last layer.

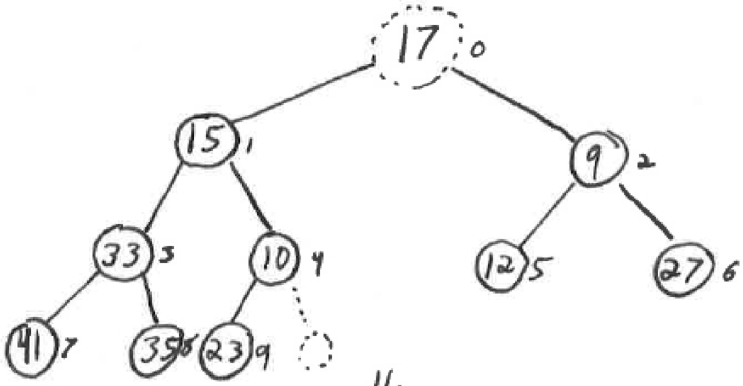
So we replace (or swap!) the root with the last node and then call a function MinHeapify in order to recursively propagate the node down a path of the heap until the Min Heap Property is satisfied.

The following is an example of RemoveMin



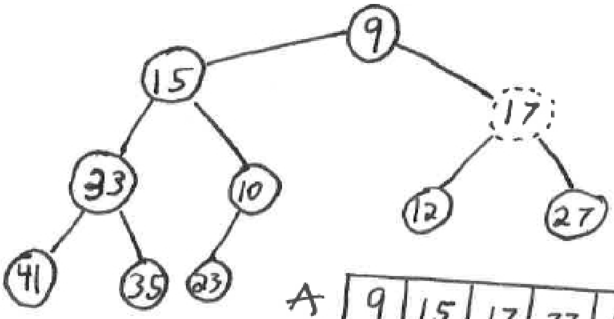
A	6	15	9	33	10	12	27	41	35	23	17
	0	1	2	3	4	5	6	7	8	9	10

Step 0 Minimum 6 to be removed.



A	17	15	9	33	10	12	27	41	35	23
	0	1	2	3	4	5	6	7	8	9

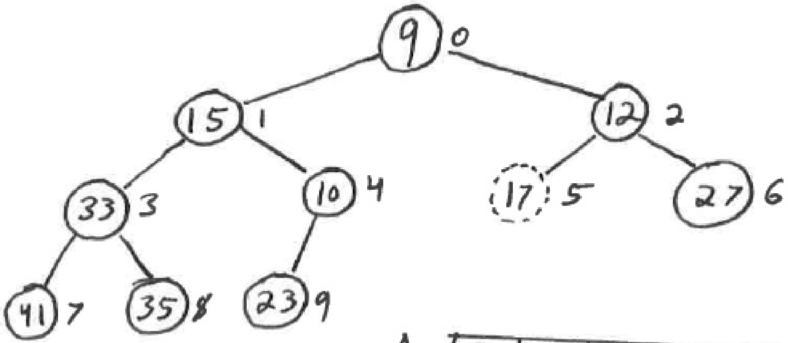
Step 1 6 replaced with 17



A

9	15	17	33	10	12	27	41	35	23
0	1	2	3	4	5	6	7	8	9

Step 3 Min Heapify 17 replaced by smallest child 9, because 9<17



A

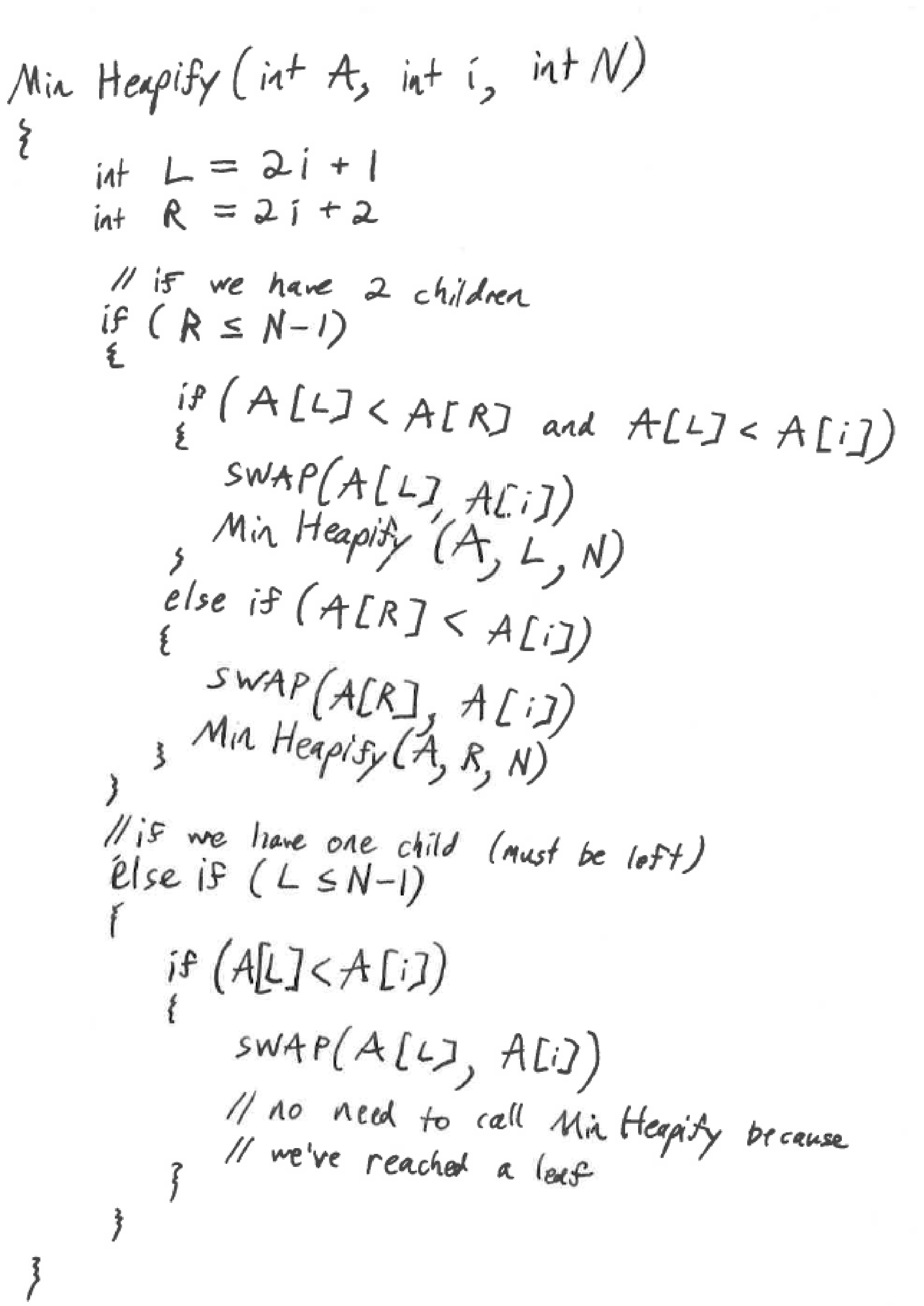
9	15	12	33	10	17	27	41	35	23	
0	1	2	3	4	5	6	7	8	9	

Step 4 Min Heapify 17 replaced by smallest child 12, because 12<17

The following is the pseudocode for RemoveMin

```
Remove Min()
{
    int min = A[0]
    A[0] = A[N-1]
    N--
    MinHeapify(A, 0, N)
    return min
}
```

The following is the pseudocode for MinHeapify



```
Min Heapify(int A, int i, int N)
{
    int L = 2i + 1
    int R = 2i + 2

    // if we have 2 children
    if (R ≤ N-1)
    {
        if (A[L] < A[R] and A[L] < A[i])
        {
            SWAP(A[L], A[i])
            Min Heapify (A, L, N)
        }
        else if (A[R] < A[i])
        {
            SWAP(A[R], A[i])
            Min Heapify(A, R, N)
        }
    }
    //if we have one child (must be left)
    else if (L ≤ N-1)
    {
        if (A[L] < A[i])
        {
            SWAP(A[L], A[i])
            // no need to call Min Heapify because
            // we've reached a leaf
        }
    }
}
```