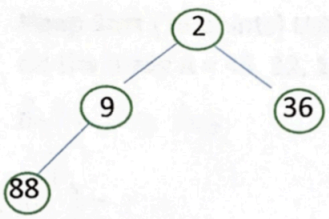
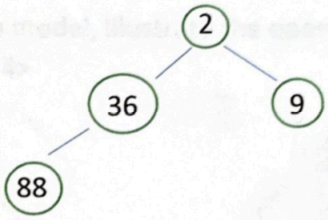
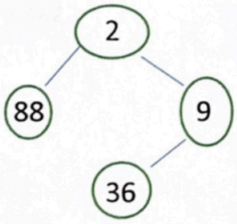
HW6

1. from Adam Keen







Answer: 2 must be the first node, because it is the smallest. No parent node can be larger than a child node, so 88 cannot have any child nodes. 88 can be a child of both 9 and 36, 36 can be a child of 2 or 9, and 9 can only be a child of 2. There are **three** possible MIN HEAPs.

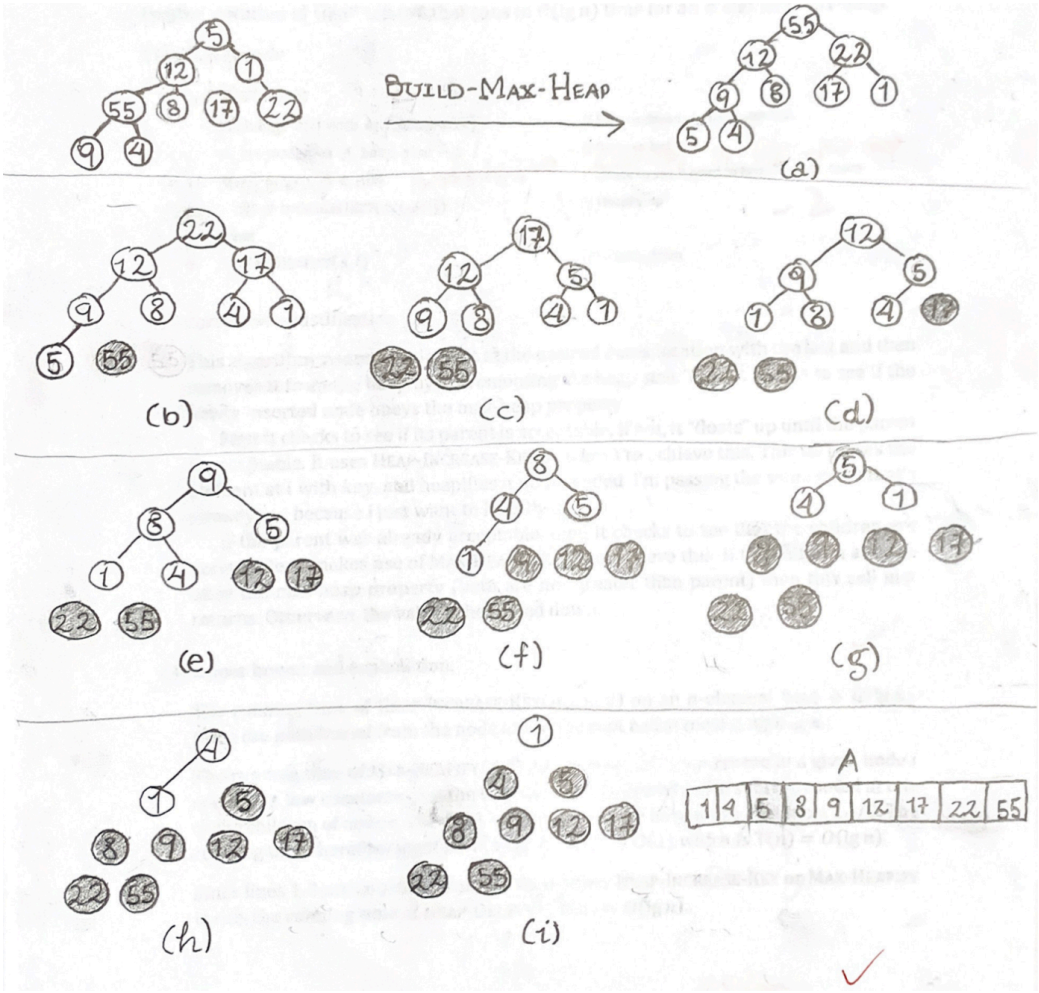
2.3 from Stavros Vallas

2. If you make a tree where the root is less than every other value in the tree then max-heapify will be called recursively at every node until it reaches a leaf node. The longest path to a leaf node would be if the recursion always went down the left child, choosing values to make this happen would cause the worst case running time to occur. The running time will be O(h) where h is the height of the tree, at eqach recursive call the running time is $\Theta(1)$ which proves that Max-heapify has a running time of $\Theta(\lg n)$, therefore its worst-case running time is $\Omega(\lg n)$.

3. First you search all the leaf nodes for the biggest element which takes O(n/2) time because there are n/2 leaf nodes roughly. So, O(n). next you replace the biggest element you found with the last element in the heap and decrease the heap size by 1 which takes O(1) time to return the biggest value. Lastly you apply min-heapify which takes O(lgn) time.

   $T(n) = O(n) + O(1) + O(\lg n) = \mathbf{O(n)}$

4. from Victor Espaillat



5 from Adam Keen and Stavros Vallas

1. HEAP-DELETE(A,i)

```
If (A[i] > A.heap-size)
        A[i] = A[heap-size]
        A.heap-size--; //this makes A[i] out of the heap
        MAX-HEAPIFY(A, i) //fix the rest of the heap
Else if (A[i] <= A.heap-size)
        HEAP-INCREASE-KEY(A, I, A.[A.heap-size])
        A.heap-size--; //need to make sure when we try to remove a node
```

it is not in the middle.

2. HEAP-DELETE will remove a node by moving the node i to the nth element of the heap (size A.heap-size) and then shrinking the bounds of the heap. If that element does not swap properly with the end element (in the event that it is still smaller than the end node after a swap) it will use HEAP-INCREASE-KEY to move its way to the end before the same removal process occurs.

3. This will run in O(lg n time) because we have already proved that MAX-HEAPIFY and HEAP-INCREASE-KEY = O(lg n) and this will take one of those two paths.

5.

```
HEAP-DELETE(A,i)
1       temp = A[i]                                          O(1)
2       exchange A[i] and A[A.heap-size]                     O(1)
3       A.heap-size = A.heap-size -1                         O(1)
4       if A[i] > A[A.heap-size]                             O(1)
5               MAX-HEAPIFY(A,i)                             O(lgn)
6       else
                HEAP-INCREASE-KEY(A,i, A[A.heap-size])       O(lgn)
```

*line 4: if A[i]> deletedValue