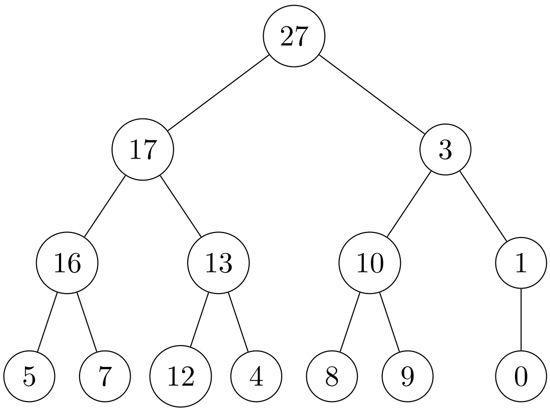
Problem Set 2

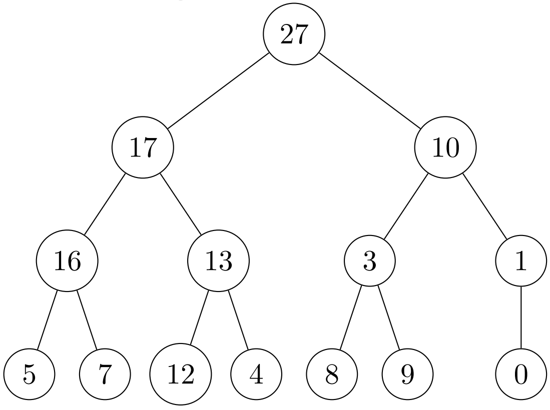
Justin Ely

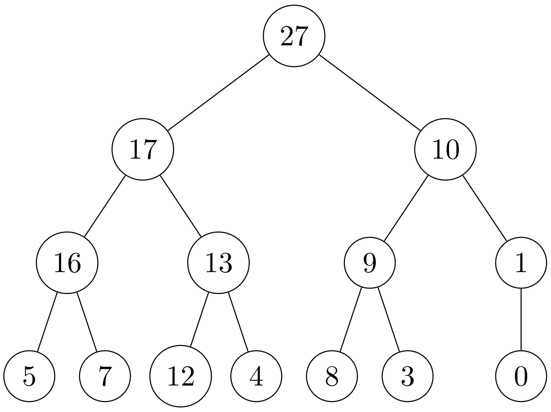
Foundations of Algorithms
June 20, 2016
612-240-0924

1a)

The illustration of MAX-HEAPIFY(A, 3) is shown in the diagrams below. The initial tree is given in the first diagram, with the two necessary permutations needed to transform the right subtree into a max-heap.

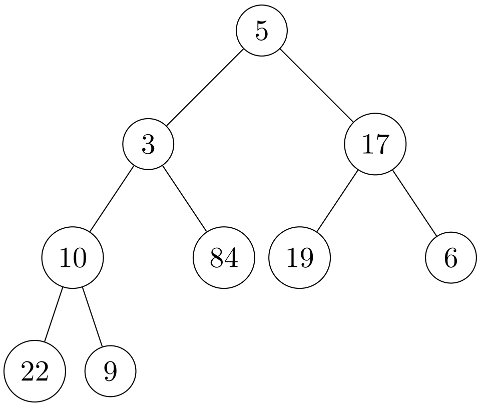


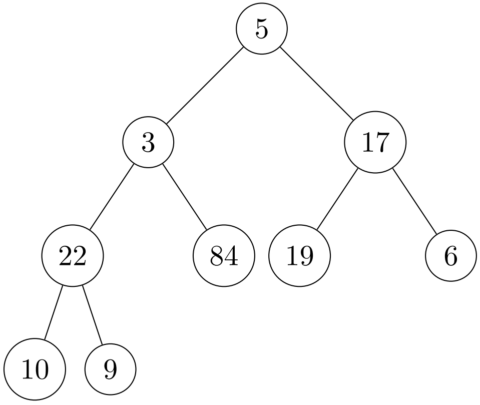


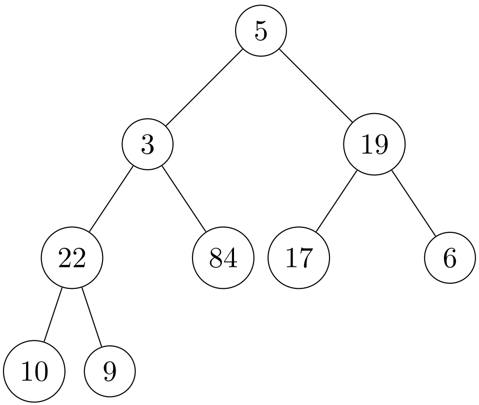


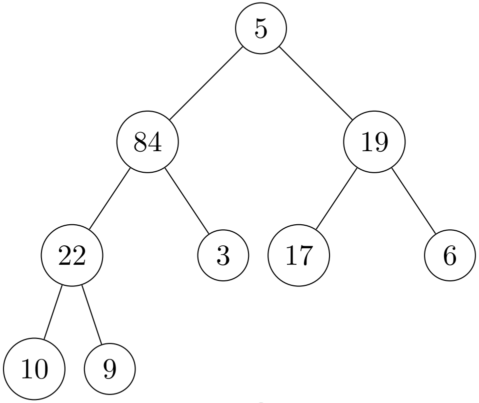
1b)

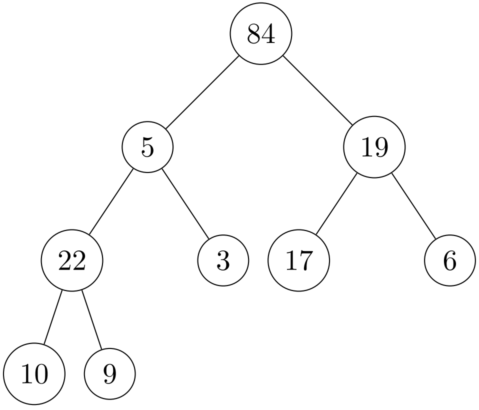
The operation of BUILD-MAX-HEAP is shown in the diagrams below. The initial tree is given in the first diagram, followed by the permutations necessary to build the maximum heap.

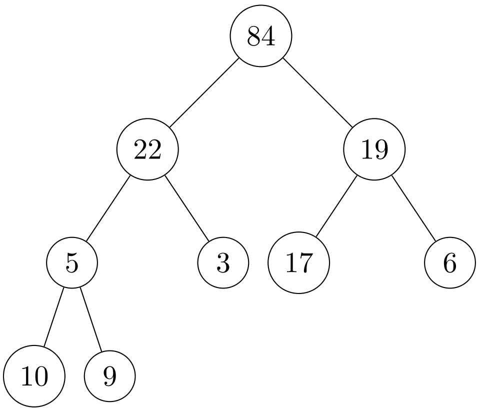


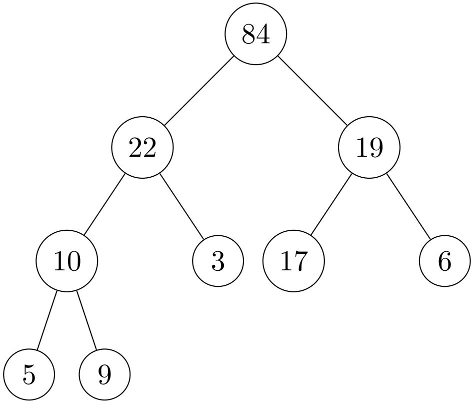



5
3
19
22
84
17
6
10
9


5
84
19
22
3
17
6
10
9

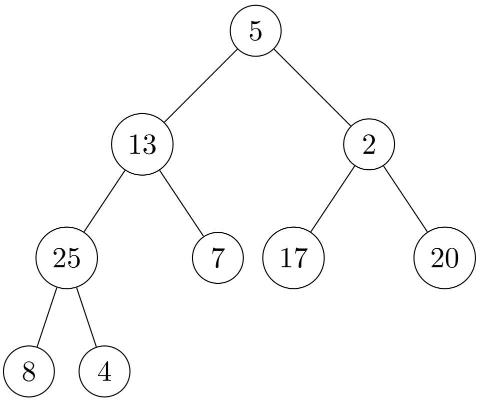

84
5
19
22
3
17
6
10
9

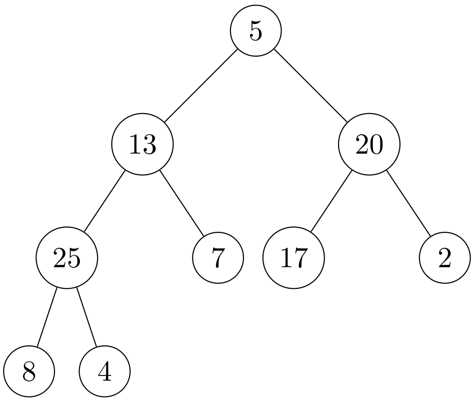


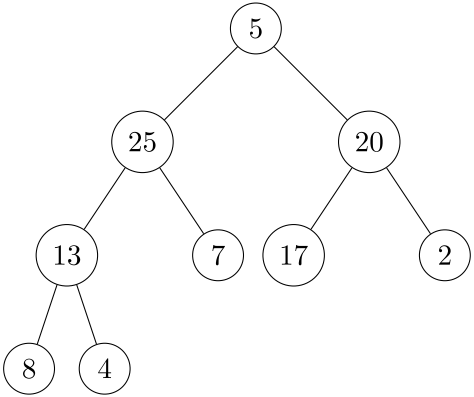


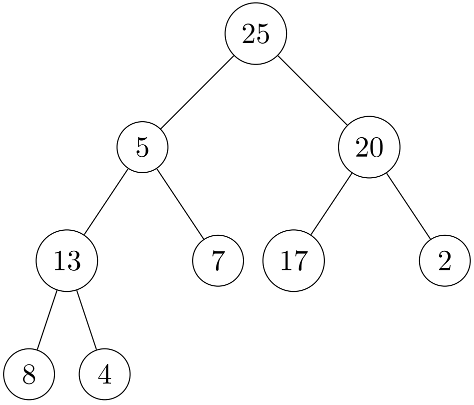
1c)

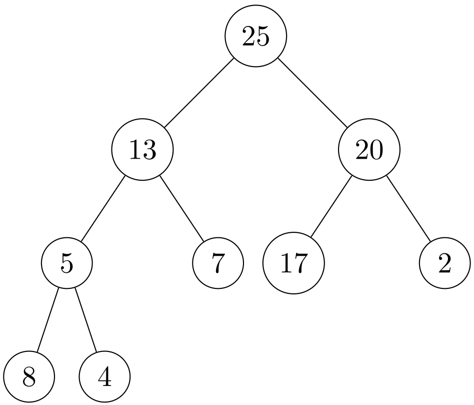
The HEAPSORT algorithm is shown in the diagrams below. The initial tree is displayed in the first diagram. The subsequent steps illustrate the building of the max heap, followed by the removal of the max values with subsequent re-heapification.

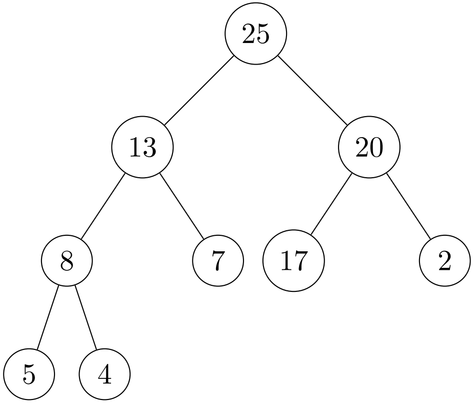



5
13
20
25
7
17
2
8
4

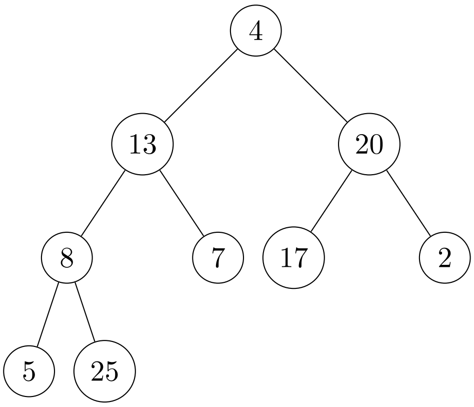

5
25
20
13
7
17
2
8
4

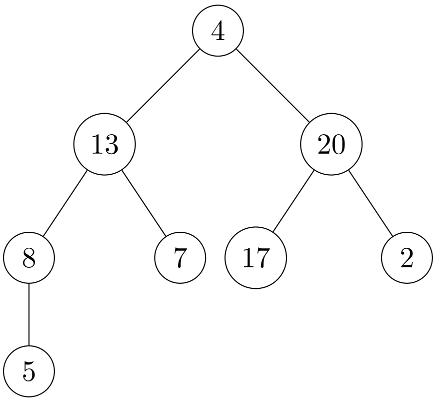

25
5
20
13
7
17
2
8
4



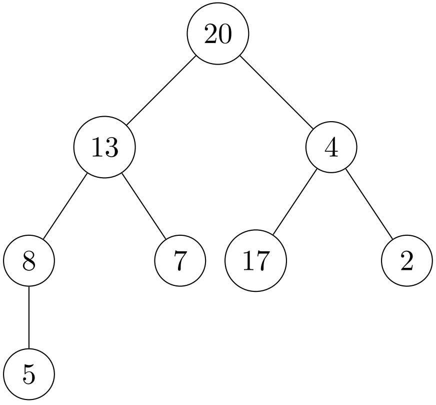


At this point, the max heap property has been established, and the sorting can begin.

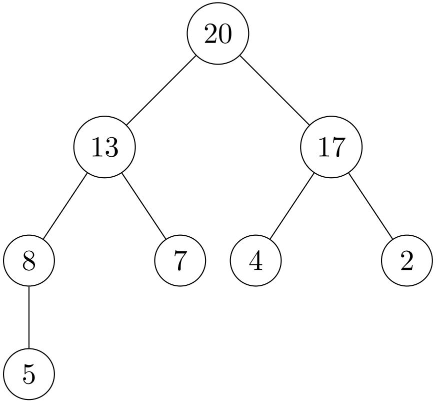




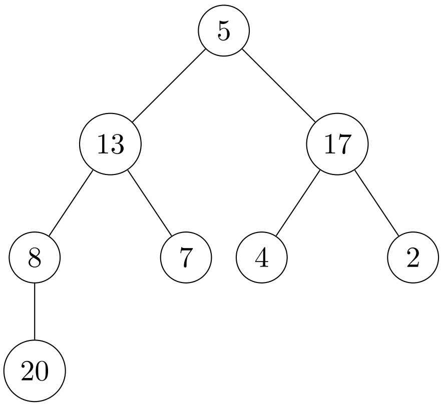
Sorted Array: 25,



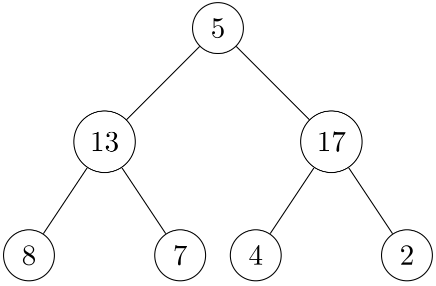
Sorted Array: 25,



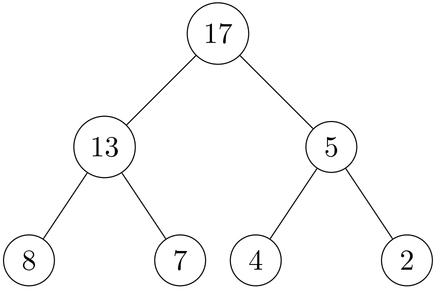
Sorted Array: 25,



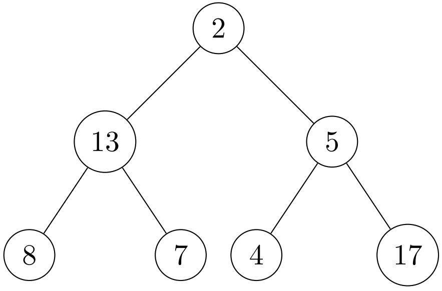
Sorted Array: 25,



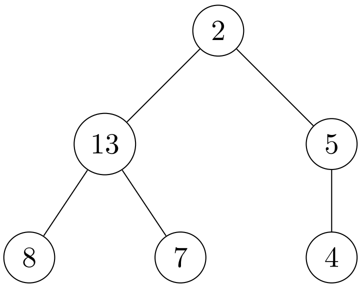
Sorted Array: 25, 20



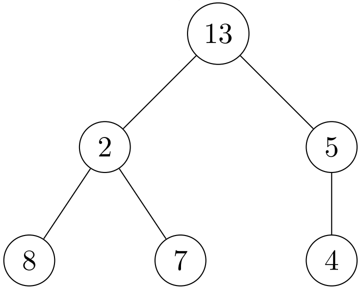
Sorted Array: 25, 20



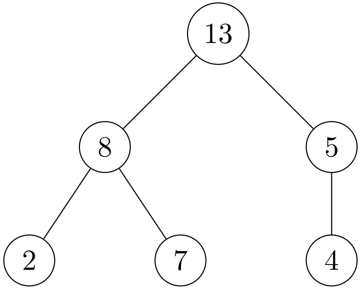
Sorted Array: 25, 20



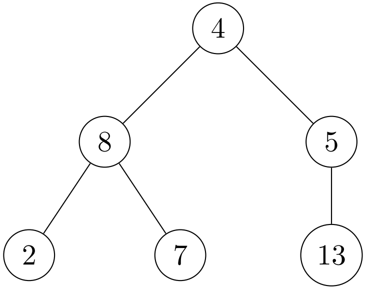
Sorted Array: 25, 20, 17



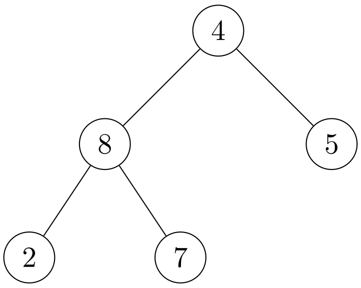
Sorted Array: 25, 20, 17



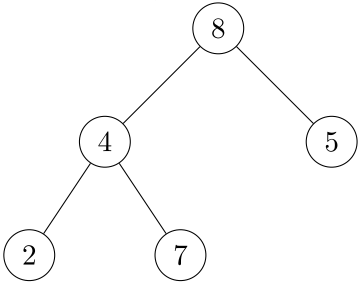
Sorted Array: 25, 20, 17



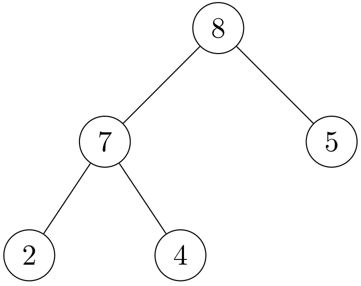
Sorted Array: 25, 20, 17



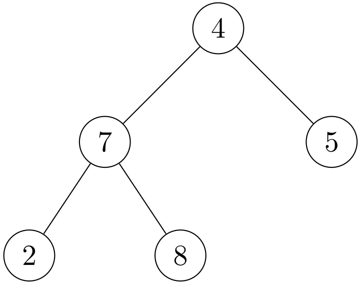
Sorted Array: 25, 20, 17, 13



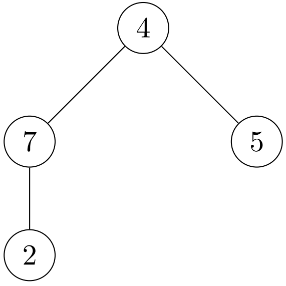
Sorted Array: 25, 20, 17, 13



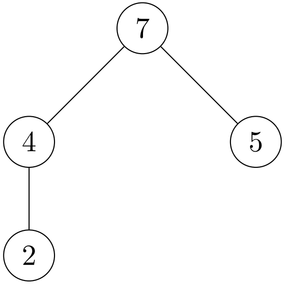
Sorted Array: 25, 20, 17, 13



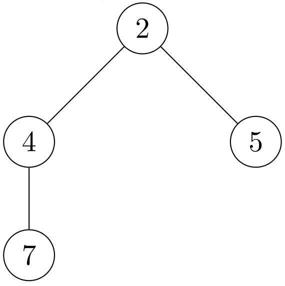
Sorted Array: 25, 20, 17, 13



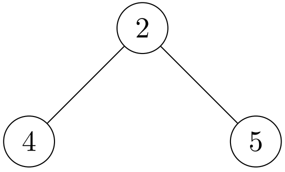
Sorted Array: 25, 20, 17, 13, 8



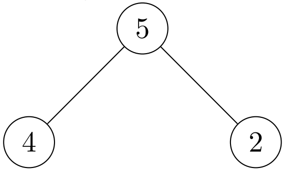
Sorted Array: 25, 20, 17, 13, 8



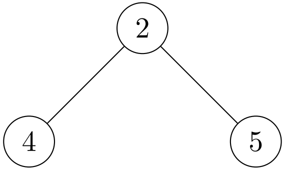
Sorted Array: 25, 20, 17, 13, 8



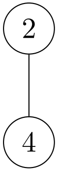
Sorted Array: 25, 20, 17, 13, 8, 7



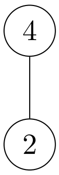
Sorted Array: 25, 20, 17, 13, 8, 7



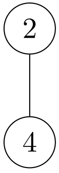
Sorted Array: 25, 20, 17, 13, 8, 7



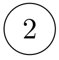
Sorted Array: 25, 20, 17, 13, 8, 7, 5



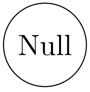
Sorted Array: 25, 20, 17, 13, 8, 7, 5



Sorted Array: 25, 20, 17, 13, 8, 7, 5



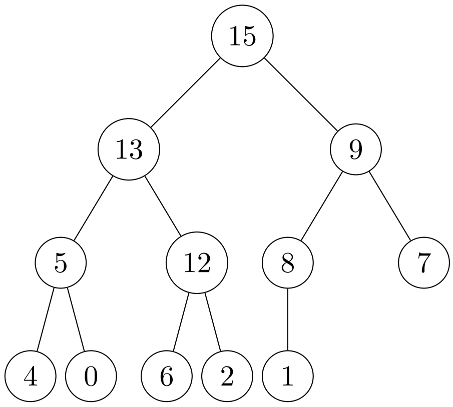
Sorted Array: 25, 20, 17, 13, 8, 7, 5, 4

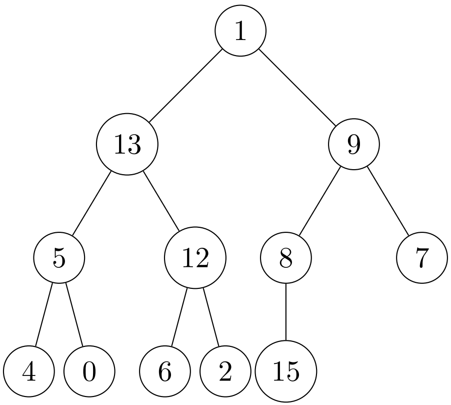


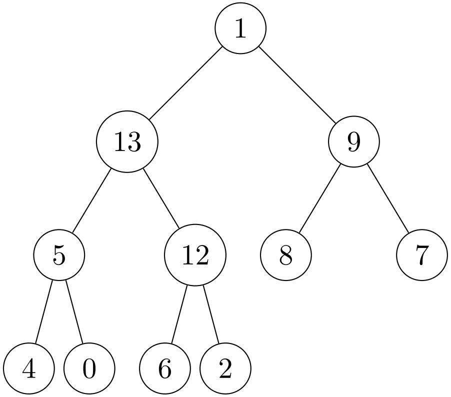
Sorted Array: 25, 20, 17, 13, 8, 7, 5, 4, 2

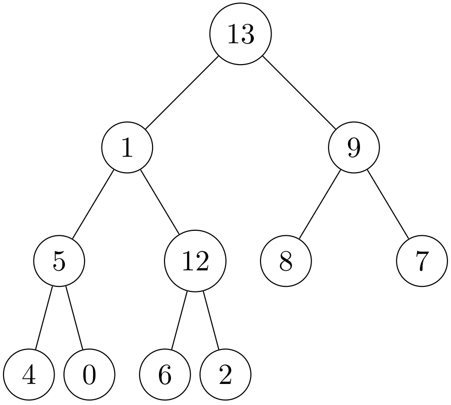
1d)

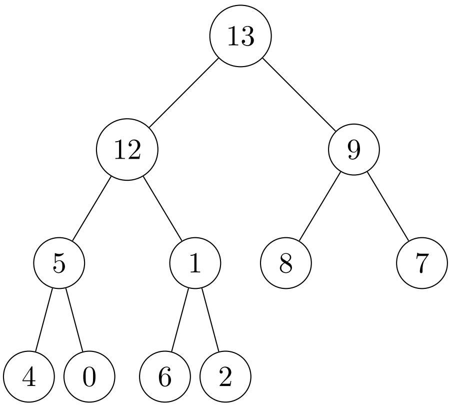
The procedure of HEAP-EXTRACT-MAX is illustrated in the diagrams below. The initial tree is displayed in the first diagram, and the subsequent steps displayed after. The steps are to exchange the maximum value with the last in the array, pull out the last value as the max, then re-heapify the tree to restore the max heap property.

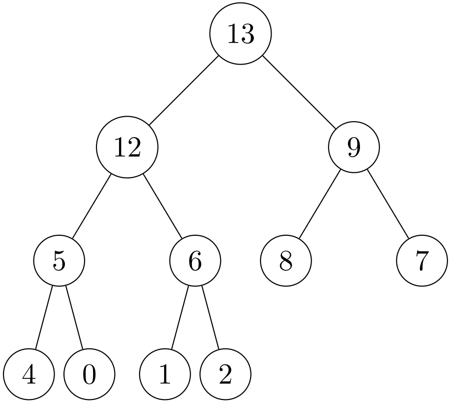



1
13
9
5
12
8
7
4
0
6
2
15


1
13
9
5
12
8
7
4
0
6
2

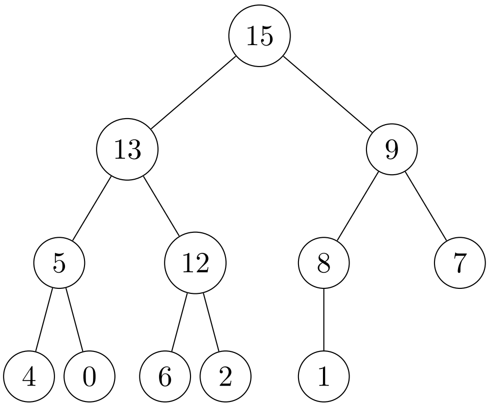

13
1
9
5
12
8
7
4
0
6
2

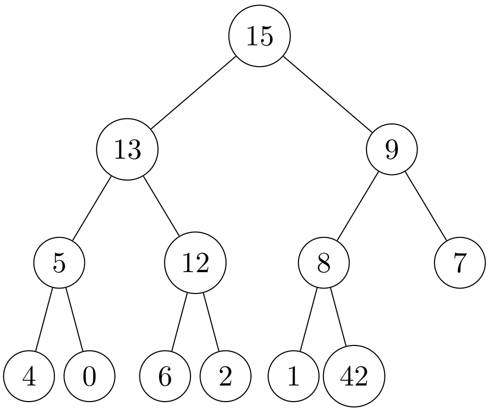


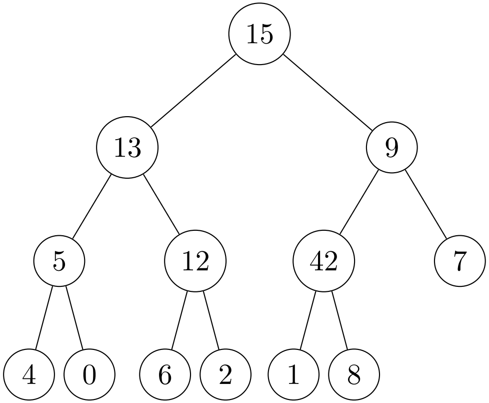


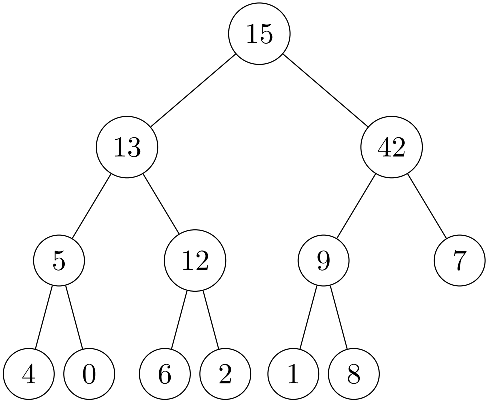
1e)

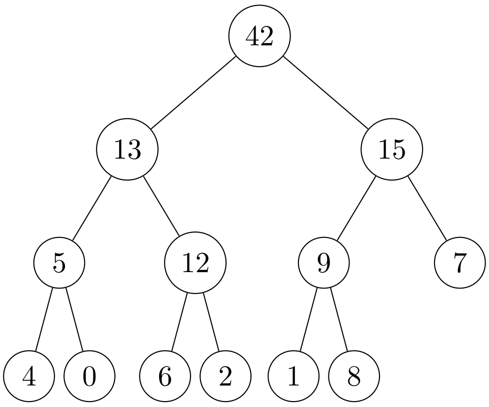
The operation of MAX-HEAP-INSERT is shown in the diagrams below. The first graph displays the initial configuration of the tree, the second shows the insertion of a new value (42) at the end of the array. The subsequent steps show the heapify operation necessary to restore the max heap property.




15
13
9
5
12
8
7
4
0
6
2
1
42


15
13
9
5
12
42
7
4
0
6
2
1
8


15
13
42
5
12
9
7
4
0
6
2
1
8



2a)

If all elements are equal, randomized quicksort's running time is $\Theta(n^2)$. This occurs as the partitioning is maximally unbalanced, and each recursive call partitions the array into a zero element subarray and an n-1 element subarray.

2b)

The modified partition algorithm is shown below. The algorithm is given in python.

```
#------------------------------------------------------

def partitionPrime(A, p, r):
    x = A[p]
    h = p
    i = p

    for j in range(p+1, r+1):
        if A[j] < x:

                #-- Swap values
            A[i], A[j], A[h+1] = A[j], A[h+1], A[i]
            i = i+1
            h = h+1

        elif A[j] == x:
                #-- Swap values
            A[h], A[j] = A[j], A[h]
            h = h+1
    return h, i

#————————————————————————————————————————————
```

2c)

The modified partition and quicksort algorithms are shown below. The algorithms are given in python.

```
#————————————————————————————————————————————

def randomizedQuicksortPrime(A, p, r):
    if p < r:
        q, t = randomizedPartition(A, p, r)

        #— ignore values equal to the pivot
        #— all values on the interval q..t
        quicksortPrime(A, p, q−1)
        quicksortPrime(A, t+1, r)

#————————————————————————————————————————————

def randomizedPartition(A, p, r):
    import random

    #— Select a random index between p and r
    i = random.randint(p, r)

    #— Swap p and i
    A[p], A[i] = A[i], A[p]

    #— Run standard partition function
    return partitionPrime(A, p, r)

#————————————————————————————————————————————
```

2d)

The analysis of textbook Section 7.4.2 remains largely the same. The initial analysis utilized the distinct values to assume that the partition sizes would be reasonably well split (not n-1 and 0), which gives rise to the $O(nlogn)$ performance.

The new algorithm can also use the same reasoning, as the modified algorithm prevents widely disparate splits in the partition size. Additionally, the cost at each partion remains the same $O(n)$ (with different constant terms due to additional checks). The major difference is that the modified algorithm can actually do significantly better than $O(nlogn)$ via the easy-out with few unique values.

3a)

The 4 steps of the radix-sort are shown below, ordered from top to bottom.

```
 1       2       3       4
---     ---     ---     ---
COW     SEA     TAB     BAR
DOG     TEA     BAR     BIG
SEA     MOB     EAR     BOX
RUG     TAB     TAR     COW
ROW     DOG     SEA     DIG
MOB     RUG     TEA     DOG
BOX     DIG     DIG     EAR
TAB     BIG     BIG     FOX
BAR     BAR     MOB     MOB
EAR     EAR     DOG     NOW
TAR     TAR     COW     ROW
DIG     COW     ROW     RUG
BIG     ROW     NOW     SEA
TEA     NOW     BOX     TAB
NOW     BOX     FOX     TAR
FOX     FOX     RUG     TEA
```

3b)

The steps of the bucket-sort algorithm are shown below, similarly to figure 8.4 in the textbook. The A column is the initial, scaled, array. The B init column shows the data initially placed in each bucket. The B sorted column shows each bucket after insertion sort. After this, a simple concatenation of the array will produce the final sorted array.

A	B init	B sorted
.78		
.17	.17 .12	.12 .17
.39	.26 .21 .23	.21 .23 .26
.26	.39	.39
.72		
.94		
.21	.68	.68
.12	.78	.72
.23		
.68	.94	.94

3c)

Bucket sort performs at it's worst case $O(n^2)$ when all elements are put into a single bucket. Since bucket sort utilizes a simpler sorting algorithm to sort each bucket, the algorithm then reduces to the complexity of the simpler algorithm. In the case of the bucket-sort implementation given in the textbook, this is insertion-sort which has a worst-case time complexity of $O(n^2)$.

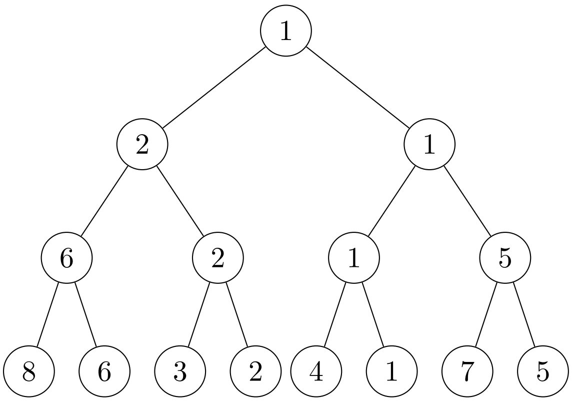
By swapping out the insertion-sort for a heap-sort would reduce the worst-case running time of a single bucket to the performance heap-sort: $O(nlogn)$.

4a)

In the tree below, each leaf represents the numbers from which we want to find the min. Each parent of every node is the smaller of the children. Each non-leaf node thus represents a comparison done during the determination of the minimum value. As can be seen from the binary tree below, this results in $n-1$ comparisons to be made to find the smallest value.

To find the second smallest, we need only compare those values that lost to the smallest value during the comparisons. Since there is just a single comparison to the smallest at each level in the tree, and we are comparing these values against each-other, this will result in $\lceil log(n) \rceil - 1$ additional comparisons.

This results in $(n-1) + \lceil log(n) \rceil - 1 = n + \lceil log(n) \rceil - 2$.



4b)

Given the x,y coordinates of the wells, the professor should take the arithmetic median of the y coordinates as the location of the pipeline. This will minimize the amount of pipe needed to connect each well to the pipeline, since the definition of the median is the value which minimizes the residual error (offset) in the datapoints.

To find this value in linear time, the OS-SELECT algorithm (which runs in $O(n)$ time should be used to find the (n/2)th smallest value in the y-coordinates of the wells.

5)

Given the functions outlined in the textbook, this can be done with a simple combination of OS-SELECT and OS-RANK.

```
x_rank = OS-RANK(T, x)
i_successor = OS-SELECT(T.root, x_rank+i)
```

OS-RANK on the input node x will return the rank of the x node. OS-SELECT called to find the (x_rank-i)'th smallest value in the subtree of x would return the ith successor in and in-order traversal. Since both functions operate in $O(lgn)$ time, and are each called once, this operation would be accomplished in $O(lgn)$ time.