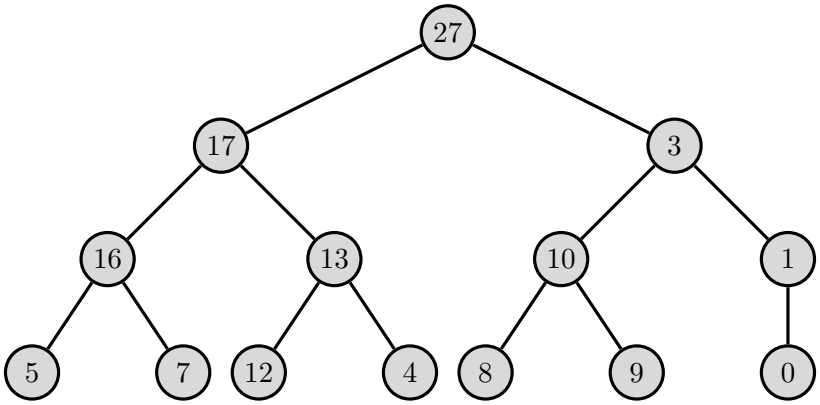
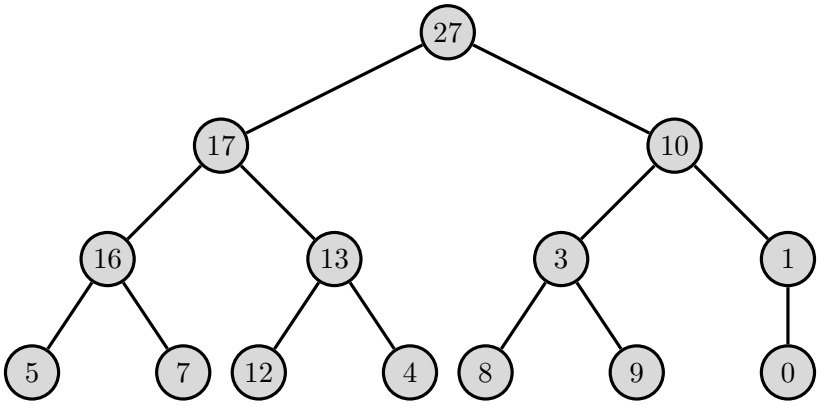
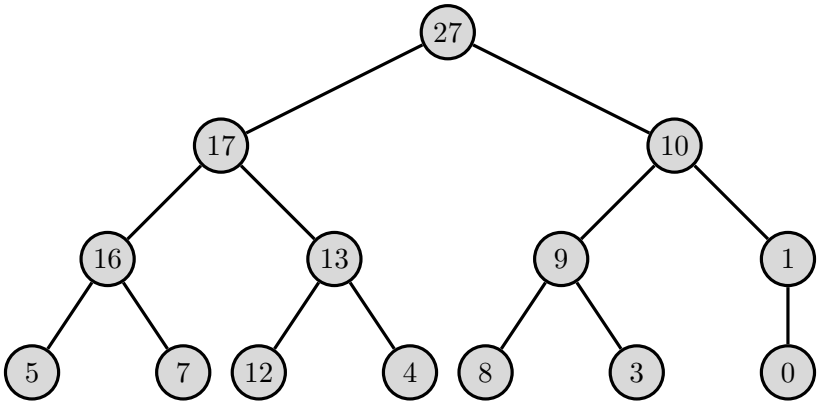
Q1 CLRS 6.2-1

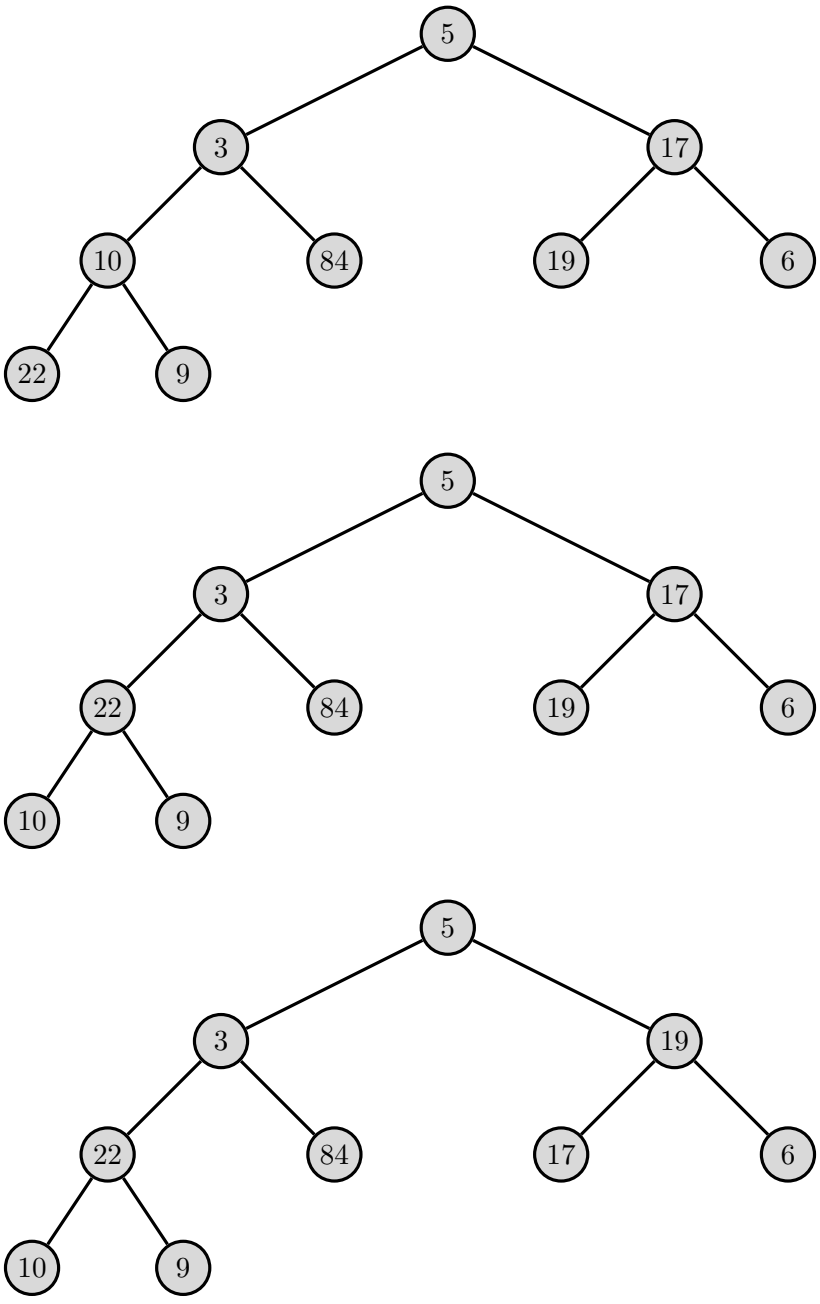
(a) Here follows an illustration of Max-Heapify(A,3) on
$A < 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 6, 9, 0 >$.

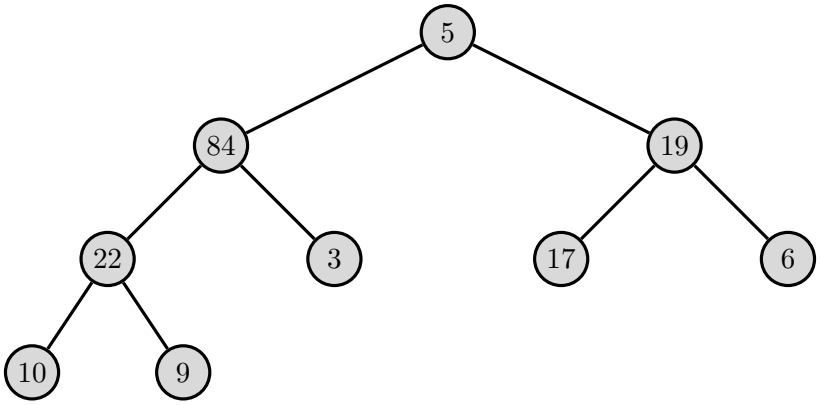


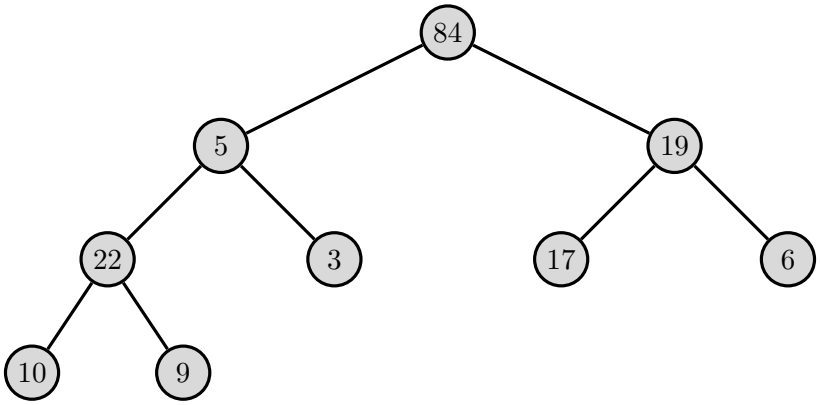


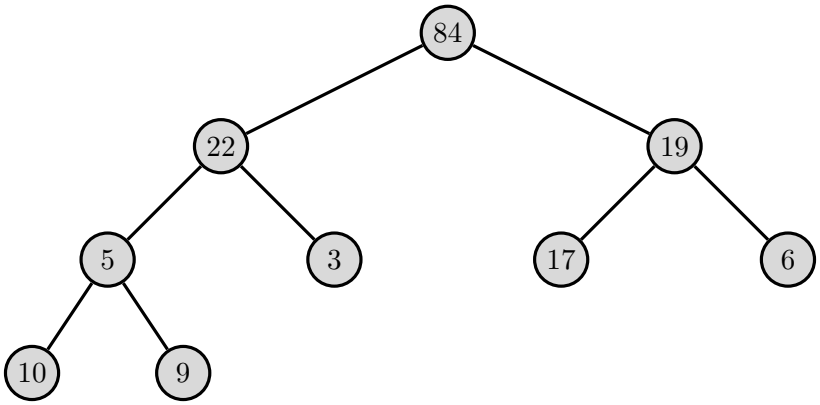


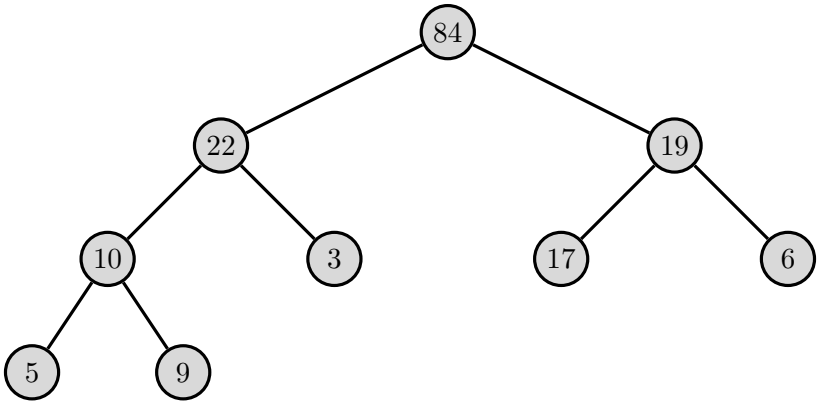
(b) Here follows an illustration of Build-Max-Heap(A) on $A < 5, 3, 17, 10, 84, 19, 6, 22, 4 >$.



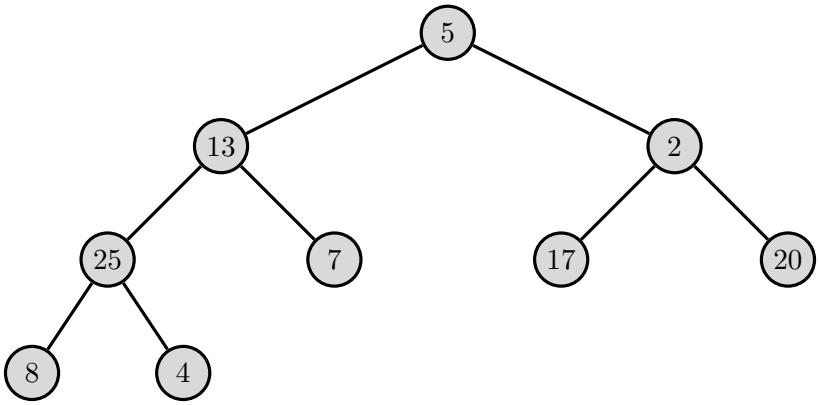

5
84
19
22
3
17
6
10
9

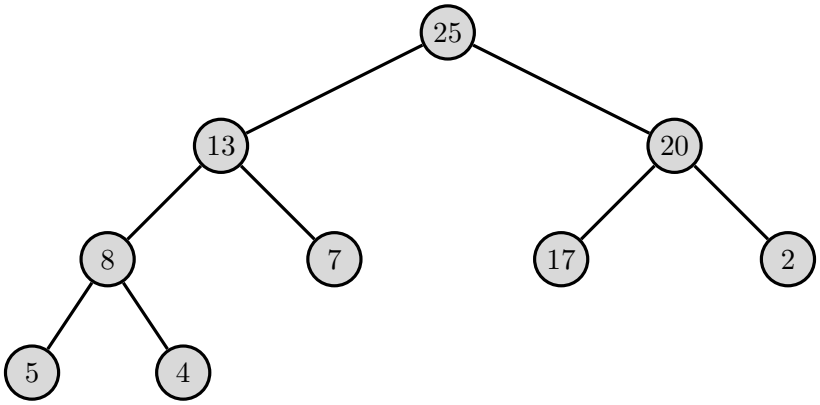

84
5
19
22
3
17
6
10
9


84
22
19
5
3
17
6
10
9

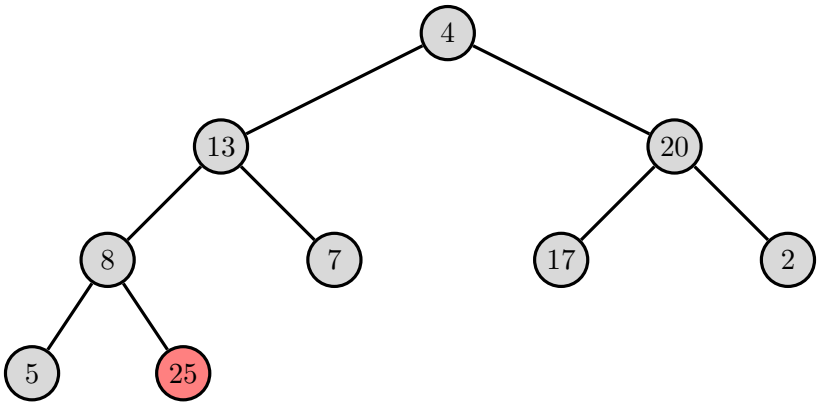

84
22
19
10
3
17
6
5
9

(c) Here follows an illustration of Heapsort(A) on $A < 5, 13, 2, 25, 7, 17, 20, 8, 4 >$.

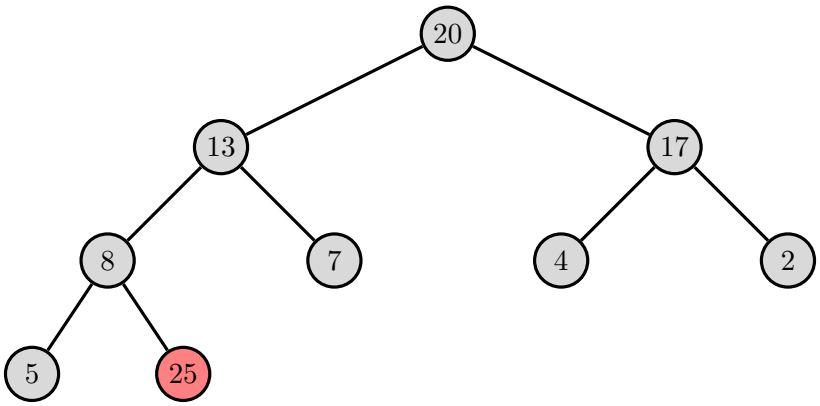




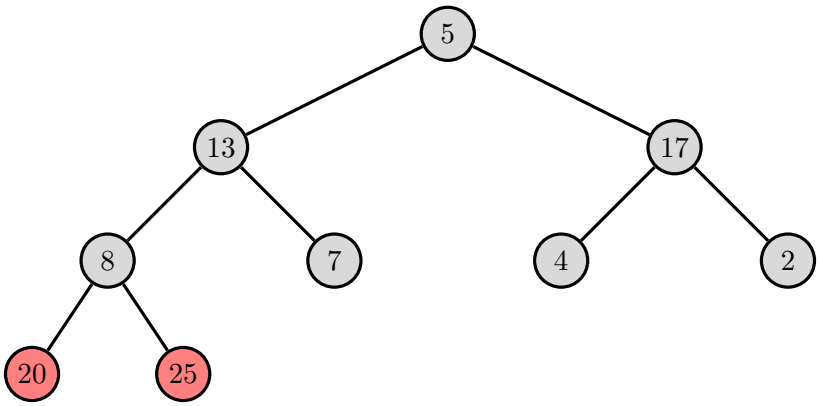
Build-Max-Heap on A.



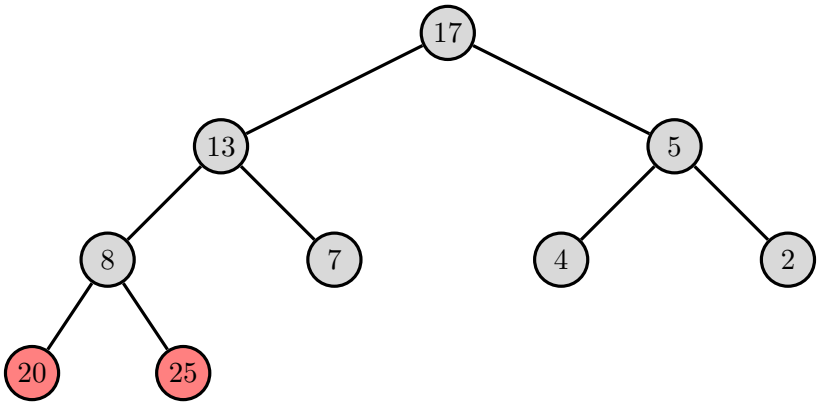
for i=A.length $\rightarrow$ 2 exchange A[1] with A[i], decrement heapsize, and Max-Heapify(A,1).



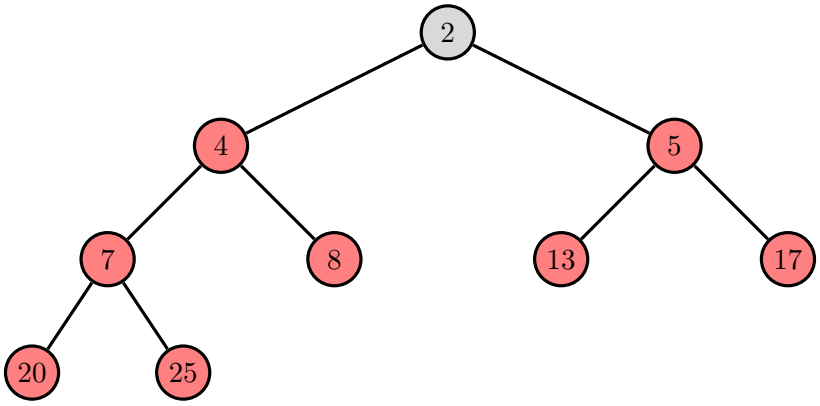
Reestablish heap.



Repeat...

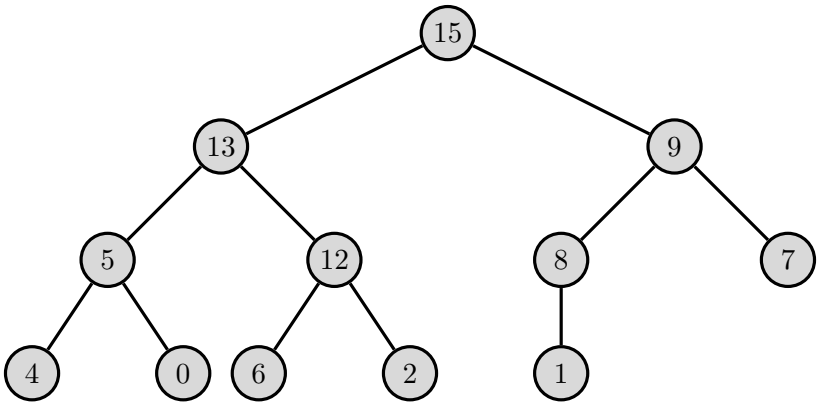


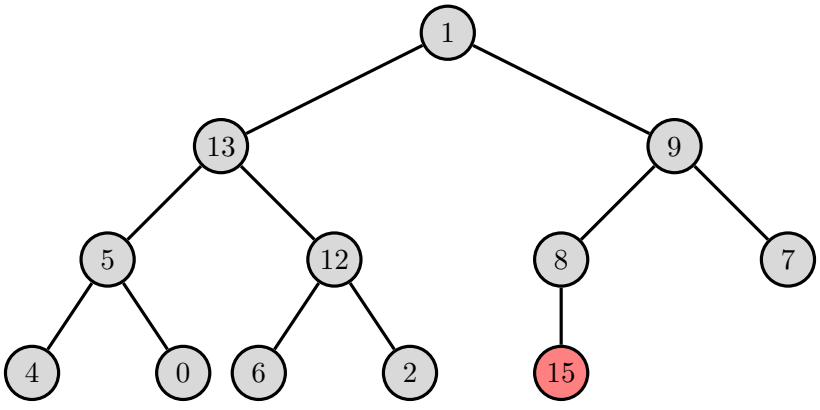
Reestablish heap.



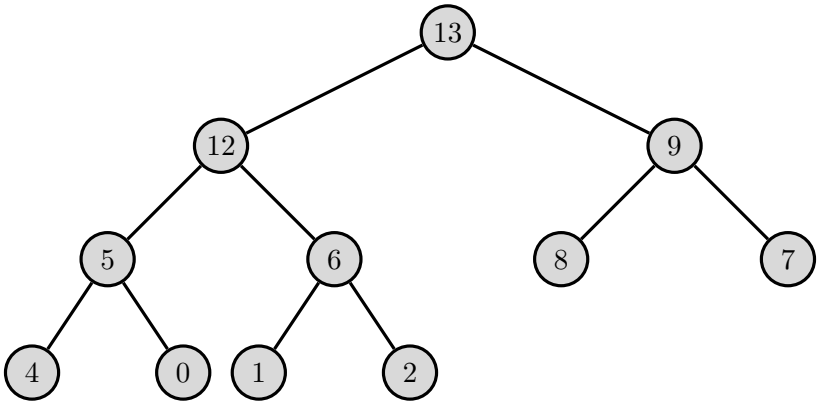
Repeat until sorted (completion of for-loop).

(d) Here follows an illustration of Heap-Extract-Max(A) on $A < 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 >$.





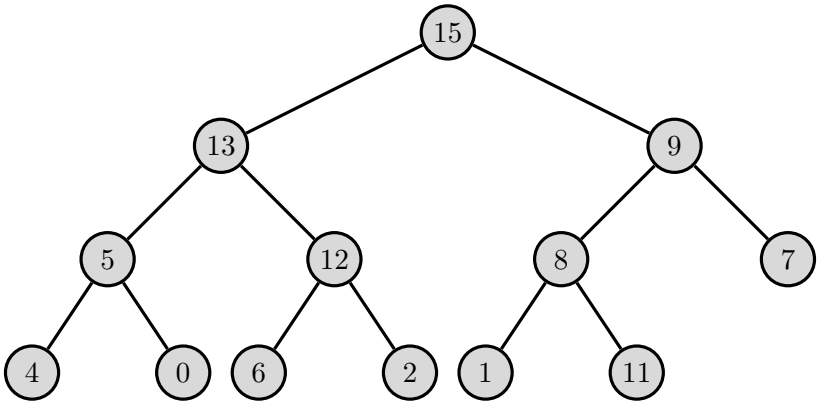
max = A[1] = 15
A[1] = A[A.heapsize] = A[12] = 1

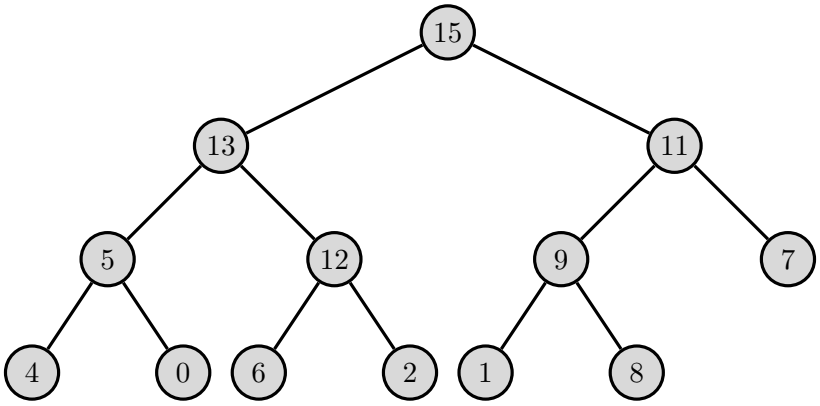


Result of Max-Heapify. Return max = 15.

(e) Here follows an illustration of Max-Heap-Insert(A,11) on $A < 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 >$.

A.heapsize = A.heapsize + 1 = 12 + 1 = 13
A[A.heapsize] = $-\infty \rightarrow$ A[13] = $-\infty$
Heap-Increase-Key(A,A.heapsize,11)
A[A.heapsize] = key $\rightarrow$ A[13] = 11





The result of the while-loop of Heap-Increase-Key(A,A.heapsize,11).

Q2 Dutch Flag Problem

This question can be solved using a partitioning method in manner similar to quicksort, utilizing three partitions and a swap() method. The partitions can be represented using three variables *i*, *j*, and *k*, setup initially as follows:

Here follows Java/pseudocode for the partitioning:

```
void DutchFlag(char[] A){
   int i = 0;
   int j = 0;
   int k = A.length-1;
   while (j <= k){
      // Case 1 - Put R character on left side at marker 'i'
      if (A[j] == R){
         swap(A[i],A[j]);
         i++;
         j++
      }
      // Case 2 - Put W character in middle at marker 'j'
      if (A[j] == W){
         j++
      }
      // Case 3 - Put B character on right side at marker 'k'
      if (A[j] == B){
         swap(A[j],A[k]);
         k--;
      }
   }
}
```

This method ensures that all 'R' characters are on the left side, all 'B' characters are on the right side, and all 'W' characters are in between by using *i*, *j*, and *k* as markers for where to insert/swap particular characters.

Q3 CLRS 7-2

(a) From CLRS page 175, "Worst case... partitioning routine produces one subproblem with n-1 elements and one with 0 elements."

All elements being the same will produce this worst case event, as partition(A,p,r) returns r, so q=r and thus quicksort recursively calls quicksort(A,p,q-1) n-1 times. The recurrence relation for this is $T(n) = T(n-1) + \Theta(n)$ which evaluates to $\Theta(n^2)$ for this worst case when all elements are the same.

(b) This can be done in a manner similar to Problem 2 (Dutch Flag Problem). The Java/pseudocode is as follows:

```
PartitionPrime(A,p,r){
   int x = A[r];
   int j = p-1;
   int k = p-1;
   int t = r+1;
   for (int i = p; i <= r-1; i++){
      if (A[i] < x){
         j++;
         k++;
         swap(A[i],A[j]);
      }
      if (A[i] == x){
         k++;
         swap(A[i],A[k]);
      }
      if (A[i] > x){
         t--;
      }
   }
   swap(A[t-1],A[r]);
   return j,t;
}
```

This method utilizes a left partition j, a right partition t, and a middle partition k. It iterates through the array and places all elements equal to the partition value (A[r]) in the middle partition. Finally, it swaps partition value A[r] with the element at A[t-1] to complete the process.

(c)

```
/*Java/Pseudocode for RandPartitionPrime*/
RandPartitionPrime(A,p,r){
   int i = Random(p,r);
   swap(A[r],A[i]);
   return PartitionPrime(A,p,r)
}

/*Java/Pseudocode for RandQuicksortPrime*/
RandQuicksortPrime(A,p,r){
```

```
   if (p < r) {
      q,t = RandPartitionPrime(A,p,r);
      RandQuicksortPrime(A,p,q);
      RandQuicksortPrime(A,t,r);
   }
}
```

(d) Examine two cases:

1. All elements distinct → O(n lg n).
2. All elements identical → O(n).

Number (2) is true because O(n) work is performed at each level but the recursion tree depth is simply 1, not lg n, because we do not recurse on elements equal to the initial pivot (which is all of them).

Thus, the time complexity of this modified Quicksort, QuicksortPrime, is O(n lg n) and the worst case occurs when there no duplicate elements.

Q4

(a) The following table demonstrates Radix Sort.

Start	First	Second	Final
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

(b) The following table demonstrates Bucket Sort.

A	B			
0.79	0	/		
0.13	1	0.13	0.16	/
0.16	2	0.2	/	
0.64	3	0.39	/	
0.39	4	0.42	/	
0.20	5	0.53	/	
0.89	6	0.64	/	
0.53	7	0.71	0.79	/
0.71	8	0.89	/	
0.42	9	/		

(c) Bucket Sort assumes a uniform distribution. The reason why Bucket Sort is $\Theta(n^2)$ is perhaps best illustrated with an example. Consider two cases.

1. Ten elements in ten buckets with insertion sort $O(n^2)$ yields $10(1^2) = 10$.
2. Ten elements in one bucket with insertion sort $O(n^2)$ yields $1(10^2) = 100$.

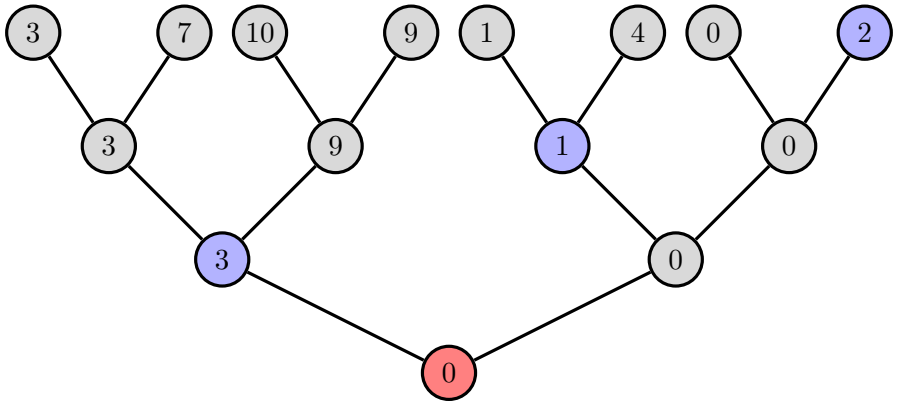
Clearly, the worst case is $O(n^2)$. If we simply use a sort more efficient than Insertion Sort (such as Merge Sort), then performance can be improved to O(n lg n).

Q5

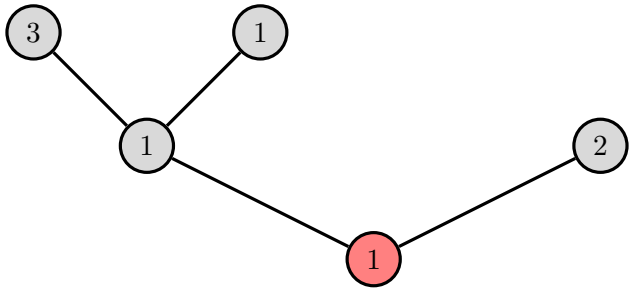
(a) First, compare pairs of elements to find the lowest. Compare "winners" in pairs to find the lowest. Repeat until only one remains (the minimum value).

To find the second lowest, compare the lg n "losers". This gives (n-1) comparisons to find the minimum and (lg n - 1) comparisons to find the second minimum.

See the following example.



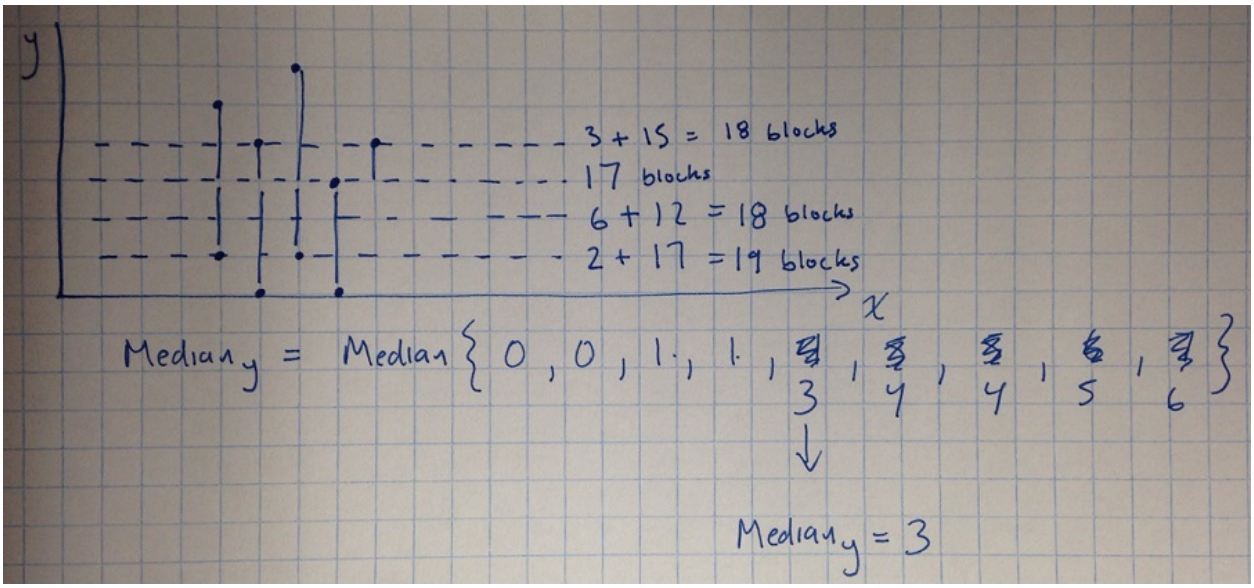
The main bracket to find the lowest. Note that n - 1 comparisons are made.



The "loser" bracket to find the second lowest. Note that lg n - 1 comparisons are made.

From this, we can see that the two smallest elements were found with only $n + \lceil lgn \rceil - 2$ comparisons.

(b) It is helpful to construct a diagram for this problem. In the figure below, we examine several possible locations for and east-west pipeline. However, only one choice provides the minimum distance for secondary pipelines. The ideal location then of the primary east-west pipline is located at the north-south axis (y-axis) median of the oil wells. To achieve this in linear time, we can simply use the Select() algorithm to determine the value of the median by finding the $(n/2)^{\text{th}}$ smallest element (when n is odd) and the average of the $\lceil (n+1)/2 \rceil^{\text{th}}$ and $\lfloor (n+1)/2 \rfloor^{\text{th}}$ smallest elements (when n is even).



Q6 CLRS 14.1-5

This can be done via the follow method:

```
/*Java/pseudocode for Successor method*/
Successor(T,x,i){
   rank = OS-RANK(T,x);
   pos = rank+i;
   successor = OS-SELECT(T.root,pos);
```

```
    return successor;
}
```