1.(a)

91	90	15	77	60	11	-2	16	21	6

(b)

90	77	15	21	60	11	-2	16	6

(c)

91	90	15	21	77	11	-2	16	6	60

2.

Loop Invariant: At the start of each iteration of the while loop. Except node i's ancestor, all nodes are root of a min-heap.

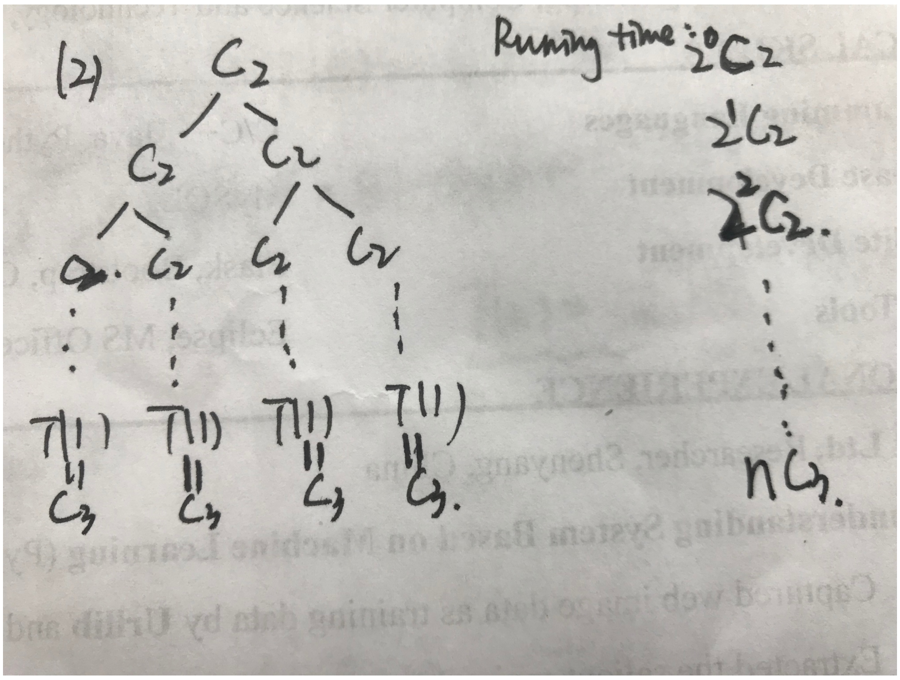
Initialization: Before decreasing node I's key, it its min-heap. After decreasing, node i is also a root of min-heap. The invariant is initially true.

Maintenance: By the loop invariant, both parent is node i and node i are min-heaps. Node i is already a root of min-heap. After exchanging node i with parent of node i when parent i's key is larger than node i's key, parent i is also a root of min-heap. Assigning parent i to i reestablishes the loop invariant at each iteration.

Termination: when the loop terminates, i<=1 or the key of parent of node I is smaller than node. By the loop invariant, each node notably node 1 is the root of a min-heap.

3.(a) $T(n) = \begin{cases} c_3 & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + c_2 & otherwise \end{cases}$

(b)



4.(a) Index starts from zero.

Parent(i)= $\begin{cases} \frac{i}{4} & otherwise \\ \frac{i}{4} - 1 & i\%4 = 0 \end{cases}$

From left to right,First_Child(i)=4*i+1;

Second_Child(i)=4*i+2;Third_Child(i)=4*i+3;Fourth_Child=4*i+4;

(b)$\log_4 n + 1$

(c)HEAP_EXTRACT_MAX(A)

```
    if A.heap_size<1
        error"heap underflow"
    max=A[0]
    A[0]=A[A.heap_size]
    A.heap_size=A.heap_size-1
    MAX_HEAPIFY(A,0)
    return min
```

Runing Time: O($\log_4 n$)

```
MAX_HEAPIFY(A,i)
    first=First_Child(i)
    second=Second_Child(i)
    third=Third_Child(i)
    fourth=Fourth_Child(i)
    if first<=A.heap_size and A[first]>A[i]
        largest=first
    else largest=i
    if second<=A.heap_size and A[second]>A[i]
        largest=second
    if third<=A.heap_size and A[third]>A[i]
        largest=third
    if fourth<=A.heap_size and A[fourth]>A[i]
        largest=fourth
    if largest≠i
        exchange A[i] with A[largest]
```

(d) MAX_HEAP_INSERT(A,key)

```
        A.heap_size=A.heap_size+1
```

Running Time: O($\log_4 n$)

```
            A[A.heap_size]=-∞
            HEAP_INCREASE_KEY(A, A.heap_size, key)
(e)HEAP_INCREASE_KEY(A, i, key)
   if key<A[i]
            error"new key is smaller"
     A[i]=key
     While i>1 and A[Parent(i)]<A[i]
            Exchange A[i] with A[Parent(i)]
            i = Parent(i)
```

Running Time: $O(\log_4 n)$

```
5. def function(names):
      BUILD_MIN_HEAP(names)
      sorted_names[names.length]
      sorted_name[1]=HEAP_EXTRACT_MIN(names)
      for i=2 to name.length
            if HEAP_EXTRACT_MIN(names) ≠ sorted_names[i-1]
                  sorted_names[i]= HEAP_EXTRACT_MIN(names)
      return sorted_names
```

Running Time: $O(nlogn)$,$\Omega(nlogn)$,$\Theta(nlogn)$

```
6.def function(A) // A is an empty array at first
      count=0 //currently  the number of the player in the team
      next= Getnextpplayer() // get the next player who wants to join in the team
      while(next≠null) // if there is no player who wants to join,the algorithm terminates.
            if count<=k      //when there are not enough k players, we join every palyer
                  MIN_HEAP_INSERT(A, next.level)                 in the team
                  count =count+1
            else   // when there is already k players in the team, we need to consider if the
                  if next.level>FIND_MIN(A)       next player has the higer level than the
                        HEAP_EXTRACT_MIN(A)              lowest level in the team
                        MIN_HEAP_INSERT(A)
      Return A      // A include the top k players
```

Space requirement O(k)

Running time:O(nlogk) n is the number of players who want to join in the team

7.def function(arriving_times)

```
    BUILD_MIN_HEAP(arriving_times)
    For i=1 to k
        first=HEAP_EXTRACT_HEAP(arriving_times)
        assign the person whose arriving time is first with number i
```

8. Key Observation: For any n>0, the number of leaves of nearly complete binary tree is $\lceil n/2 \rceil$. Proof by induction. Show that it's true for h=0. This is the direct result from above observation. Suppose it's true for h-1. Let N_h be the number of nodes at height h in the n-node tree T. Consider the tree T' formed by removing the leaves of T. It has n'=n-$\lceil n/2 \rceil$=$\lfloor n/2 \rfloor$ nodes. Note that the nodes at height h in T would be at height h-1 in tree T'. Let N'_{h-1} denote the number of nodes at height h-1 in T', we have $N_h = N'_{h-1}$. By induction, we have $N_h = N'_{h-1} = \lceil n'/2^h \rceil = \left\lceil \left\lfloor \frac{n}{2} \right\rfloor /2^h \right\rceil \le \left\lceil (\frac{n}{2})/2^h \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$

```
9.def function(A, key)
   medianLow=0
   medianLarge=0
   min_heap // to store all confidence that are smaller than median.
   max_heap // to store all confidence that are larger than median
   if A.length %2=1
        if key >medianLow
             MIN_HEAP_INSERT( min_heap,key)
             medianLarge=HEAP_EXTRACT_MIN(min_heap)
        else
             MAX_HEAP_INSERT(max_heap, key)
             midianLarge =medianLow
             midianLow = HEAP_EXTRACT_MAX(max_heap)
  return (medianLow+medianLarge)/2
else
        if key >mediaLarge
             MIN_HEAP_INSERT( min_heap, key)
             MAX_HEAP_INSERT(max_heap, medianLow)
             median_low =median_large
             median_large=HEAP_EXTRACT_MIN( min_heap)
```

```
        else if key<mediaLow
            MAX_HEAP_INSERT( max_heap, key)
            MIN_HEAP_INSERT(min_heap, medianLarge)
            median_large= median_low
            median_low=HEAP_EXTRAVT_MAX(max_heap)
        else
            MAX_HEAP_INSERT(max_heap, medianLow)
            medianLow=key
return medianLow
```