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.

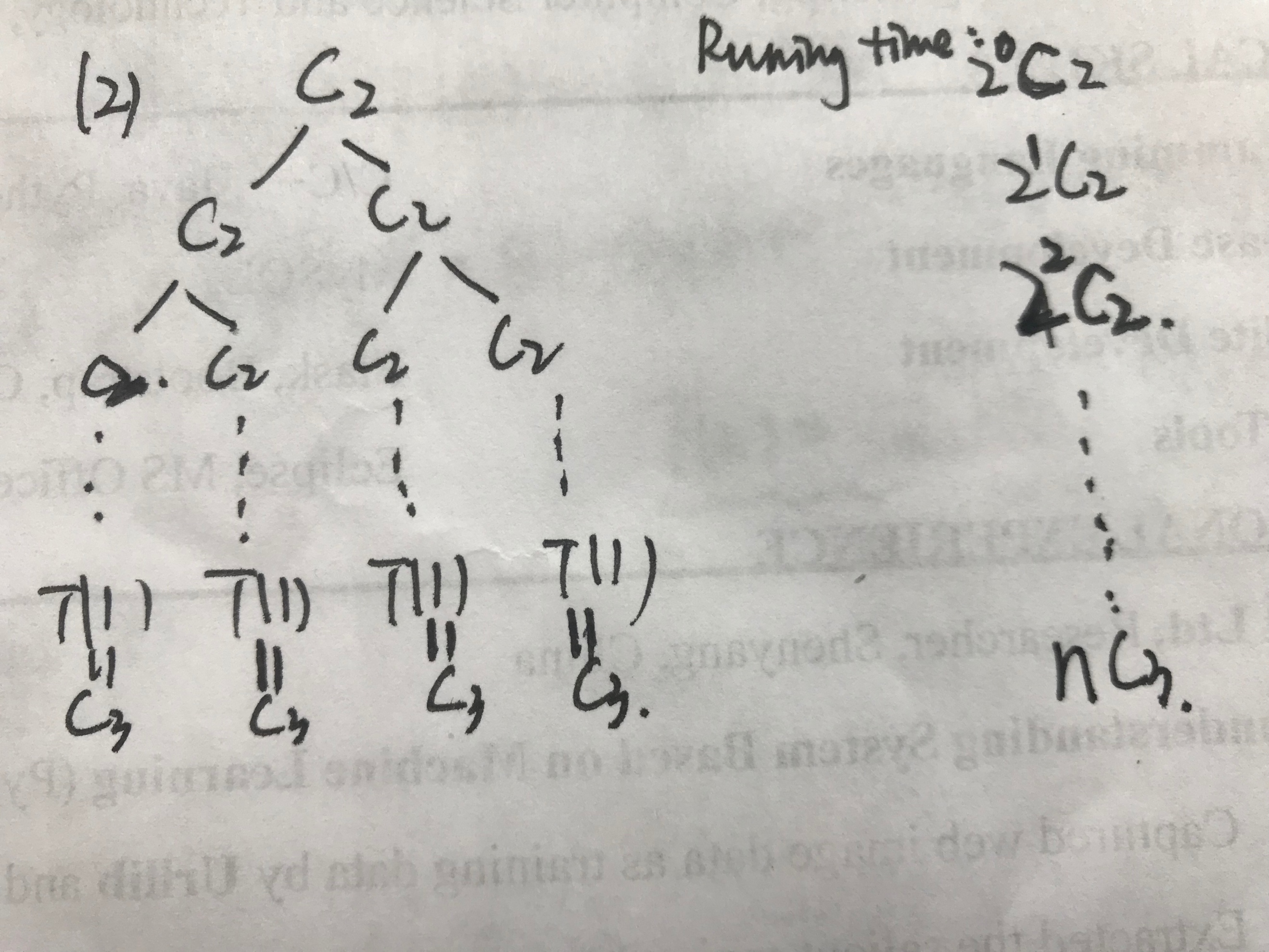
Maintenance: By the loop invariant, both parent is node i and node i are min-heaps. Node i is already a root of min-heap. After exchanging node i with parent of node i when parent i’s key is larger than node i’s key, parent i is also a root of min-heap.

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)

(b)



4.(a) Index starts from zero.

Parent(i)=

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)

(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 Runing Time: O()

MAX\_HEAPIFY(A,0)

return min

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 largesti

exchange A[i] with A[largest]

(d) MAX\_HEAP\_INSERT(A,key)

A.heap\_size=A.heap\_size+1 Running Time: O()

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] Running Time: O()

Exchange A[i] with A[Parent(i)]

i = Parent(i)

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,,

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(nextnull) // 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 . 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 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-= nodes. Note that the nodes at height h in T would be at height h-1 in tree T’. Let denote the number of nodes at height h-1 in T’, we have . By induction, we have =

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