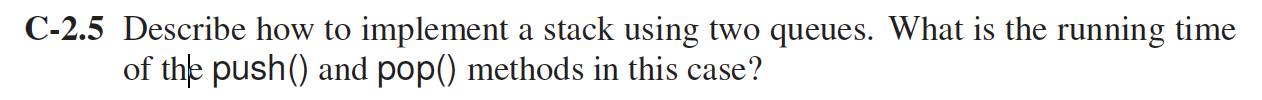
C 2.5



The push method is used to push x to stack s:

push(s,x):

1. Enqueue x to q1
2. One by one dequeue everything from q2 and enqueue it to q1
3. Swap the names of q1 and q2

The running time of push(s,x) method is O(n).

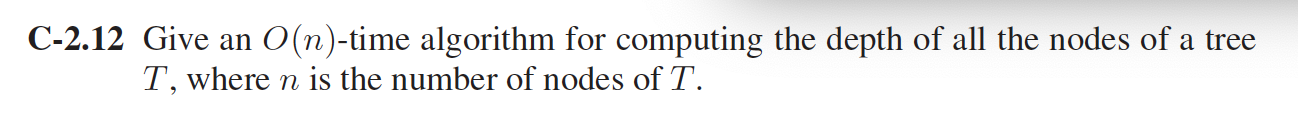
The pop method is used to pop an element from stack:

pop(s):

1. Dequeue an element from q2 and return it.

The running time of pop method is O(1).

C-2.12 (with pseudo-code),



Algorithm ComputeTreeDepth (T, v):

Input: tree T; v is a node of T

Output: depth of each node in the subtree of T rooted at v

if (T.isRoot(v)) then

setDepth(v, 0);

else{

setDepth(v, 1 + getDepth(T.parent(v)));

}

children = T.children(v);

while (children.hasNext()) do

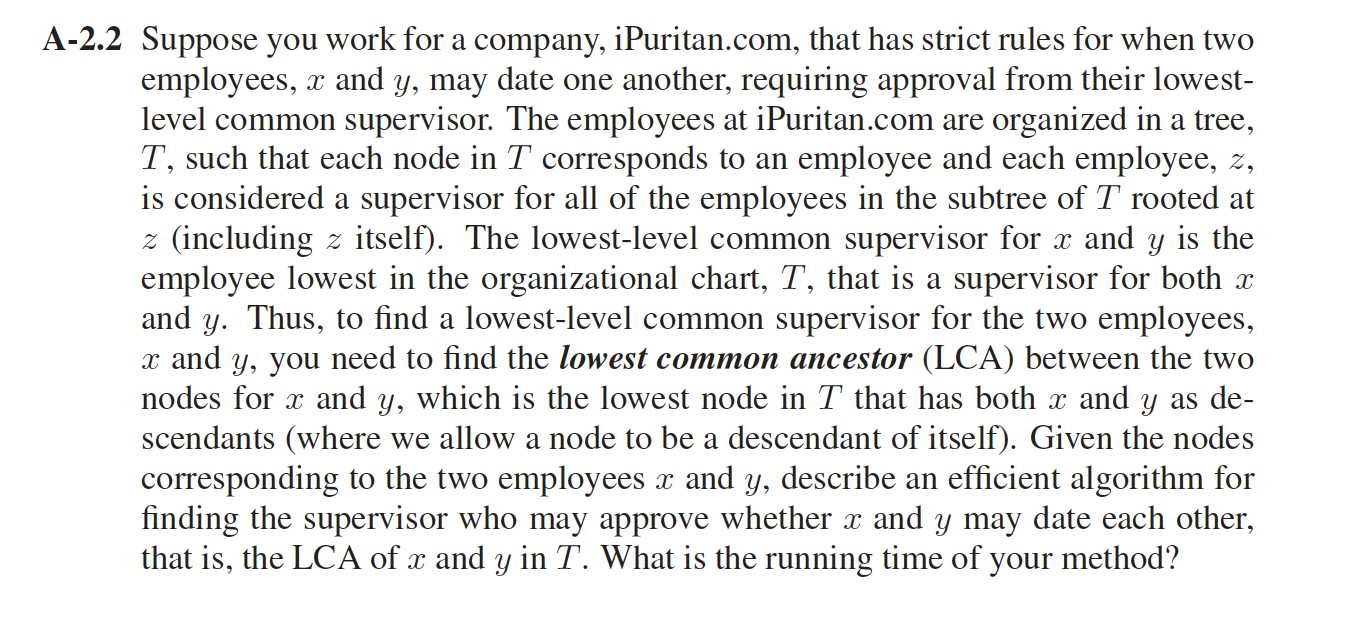
{

child = children.next()

ComputeTreeDepth(T, child)

}

A-2.2



Answer:

Algorithm FindLCA (z,x,y):

Input: tree T, z is initiated as a the root of all employees; x and y are employees.

Output: z is the LCA node for A and B.

FindLCA( z,x,y)

{

If ( z==0) then

return null;

If(z==x || z == y) then

return z;

left = FindLCA(z.getLeft(),x,y)

right = FindLCA(z.getRight(),x,y)

If (left!= null && right !=null)then

return z;

If (left !=null){

return left;

}

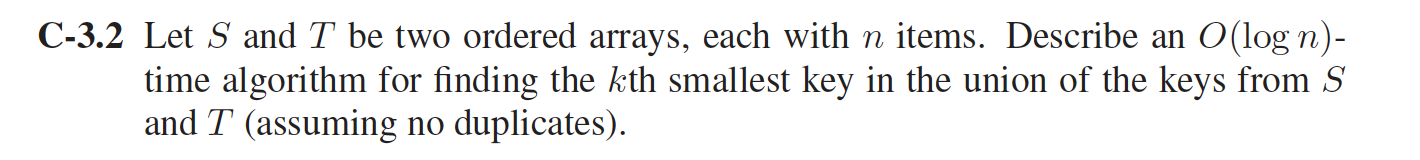
Else

return right;

}

We used BFS, The Running time of the algorithm is O(n).

C-3.2



Use the divide-and-conquer technique.

FindkthSmall(k,T,S)

{

Let median\_s= len(S)/2 and the median of S is S[median\_s], let median\_t=len(T)/2 and the median of T is T[median\_t];

If (median\_s==0 || median\_t==0) then{

Return kth value of S or T.

}

If (median\_s+ median\_t)>k;then{

If (S[median\_s]> T[median\_t]);then{

Discard bigger half of S;

FindkthSmall(K,T,S1/2);

}else{

Discard bigger half of T;

Findkthsmall(k,T1/2,S);

}

}else{

If (S[median\_s]> T[median\_t]);then{

Discard smaller half of T;

FindkthSmall(K-median\_t,T1/2,S);

}else{

Discard smaller half of S;

FindkthSmall(k-media\_s,S1/2,T);

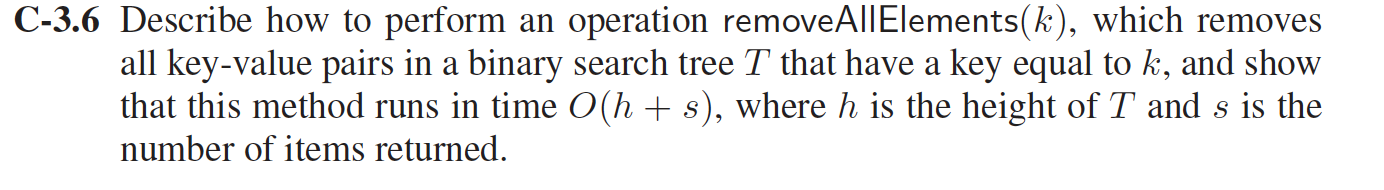
}

}

}

At each iteration, we discard a half of some array. So the time complexity is O(log(len(S))+log(len(T))) , which is O(logn) for input arrays S and T of size n.

C-3.6 (with pseudo-code)



The process can be divided into 2 steps:

1. Do a binary search to find an element equal to k.

Algorithm

BinarySearch(Tree T,key k,node v)

If v is an external node then

Return v

If k<key(v) then

Return BinarySearch(T,k,T.left(v));

Else if k>key(v) then

Return BinarySearch(T,k,T.right(v));

Else

Return FindALlElements(k,v,c) // goto process 2 to see the details

Return v

1. Note that after k is found , if it is occurs again, it will be placed in the left most internal node of the right sub tree.

Algorithm FindALlElements(k,v,c)

Input: the search key k, a node of tree T is v and a collection c

Output : an iterable collection containing the foud entries

If v is an external node then

Return c;

If k=key(v) then

c.addlist(v);

Return FindALlElements(k,T.right(v),c);

Else if k< key(v) then

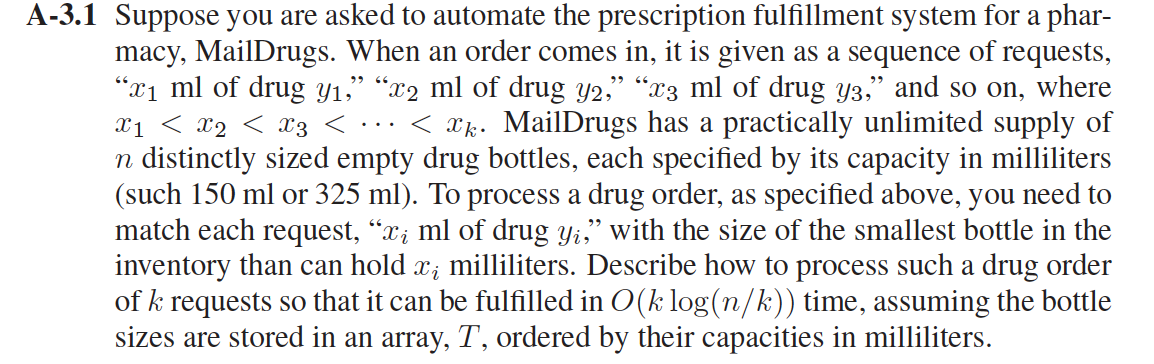
Return FindALlElements(k,T.left(v),c)

Else

Return FindALlElements(k,T.right(v),c)

In summary, It takes O(logn) time for the binary search  (where n is the total number of elements), and s time return all of the elements k. Therefore we have a solution running in at most O(logn+s) time. We know that height of the tree can be given as h = logn. Therefore, time complexity can be written as O(logn+s), which is equivalent to O(h+s).

A-3.1



Answer:

1. Devide the n bottles into k blocks. In each block , it contains n/k bottles.
2. Build k’s red-black tree( T1 to Tk ) for every block of the empty drug bottles. Since the bottle is originally ordered by their capacities , each node has three values: index of bottle, capacity of bottle；and best (largest) in all the bottles represented by the subtree rooted at the node. The ordering of the tree is by the bottle index.
3. Sort the drug order of k requests by their capacity xi. the result is stored in SortedK[i]. using bucket sort ,which time complexity is O(n).
4. Algorithm is below, if the SortedK[i] is bigger that the biggest capacity of the subtree Tcount,then we will go to next tree:

Count =0;

For i from 0 to k-1:

if (SortedK[i]>Tcount.root.bestCapacityinSubtree)then

++Count;

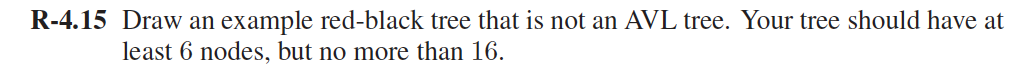
endif

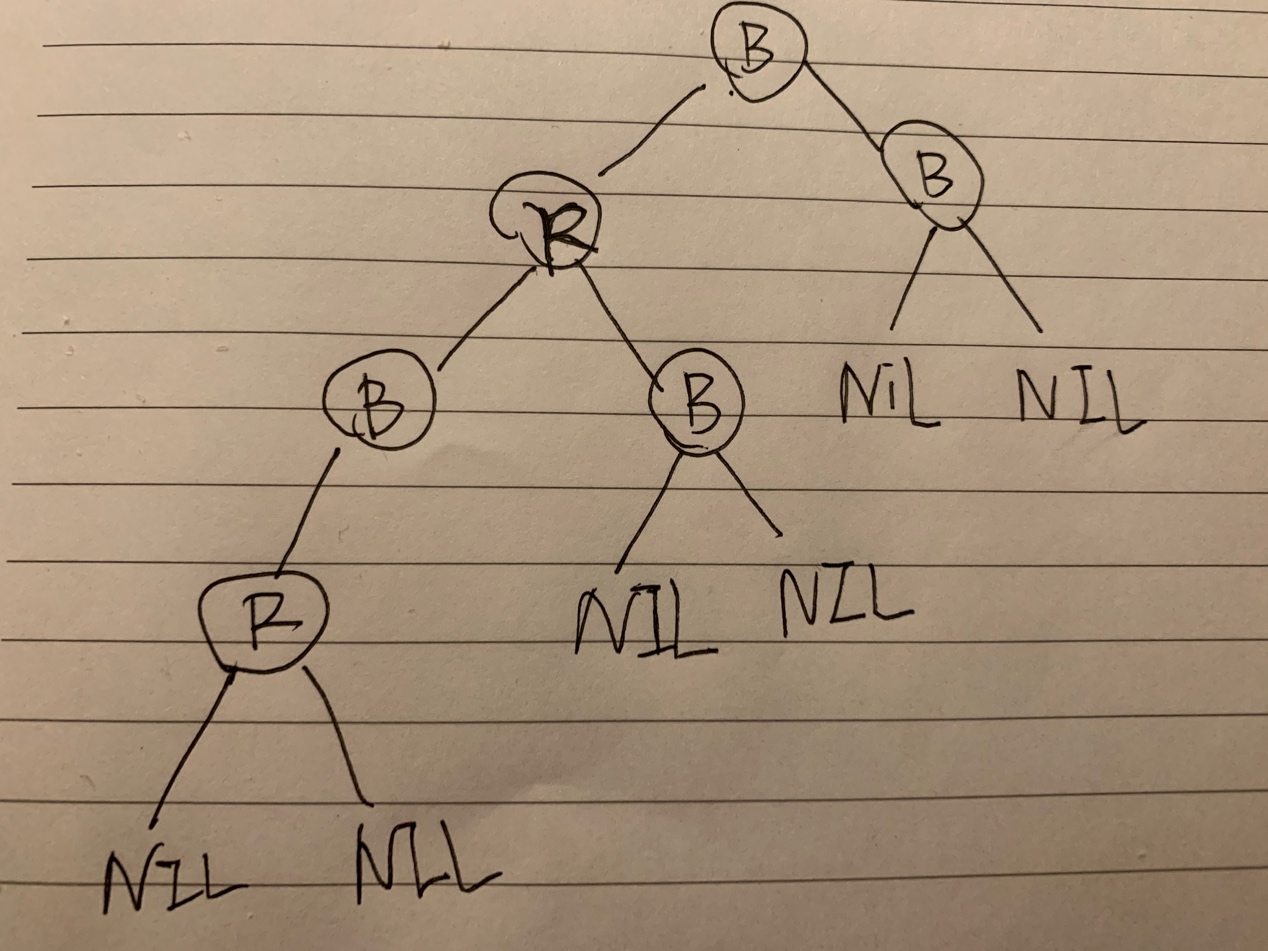
Do BinarySearch SortedK[i] in Tcount;

Endfor

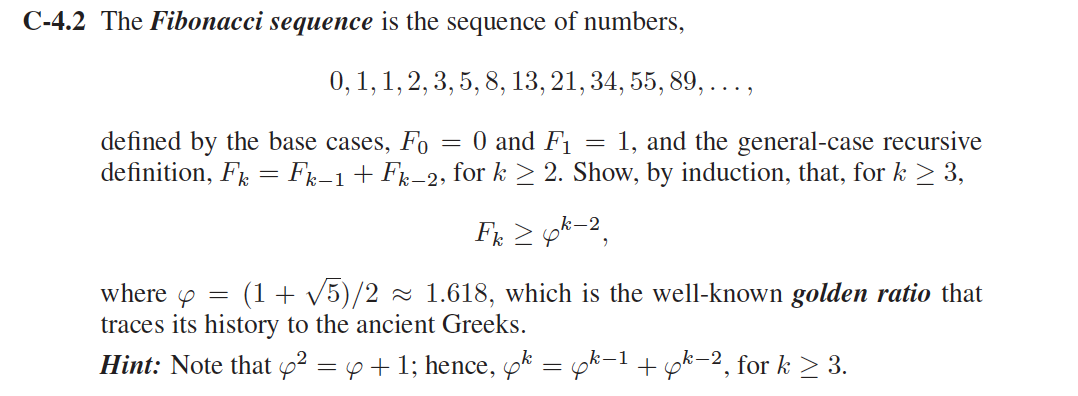
Hence, in every block T,the nodes of the tree is no more than n/k, for each tree, the time complexity of inner binary search is O(log(n/k)). And from the outside, we will go through k times loop. So the total time complexity is O(klog(n/k));

R-4.15





C-4.2



Answer:

The base case for k=3

F3= F2+F1 = 2 ;F3>ϕ=1.618

Hence true;

Induction Hypothesis: Let us assume that it is true for k;

To prove it is true for k+1:

Because Fk+1=Fk+Fk-1

According to hypothesis:

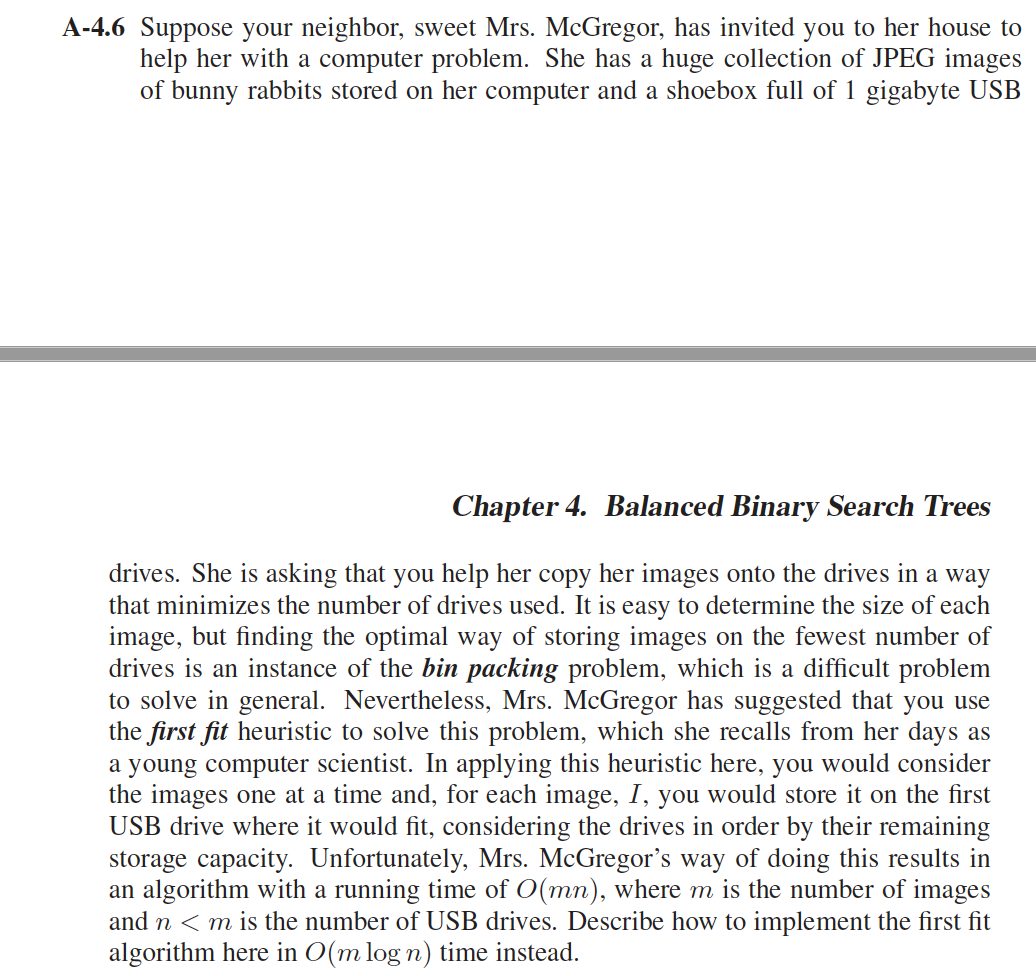
Fk+1=Fk+Fk-1 ≥ ϕk-2+ ϕk-3

And because ϕk = ϕk-1 + ϕk-2, hence ϕk-2+ ϕk-3= ϕk-1

Hence Fk+1≥ϕk-1

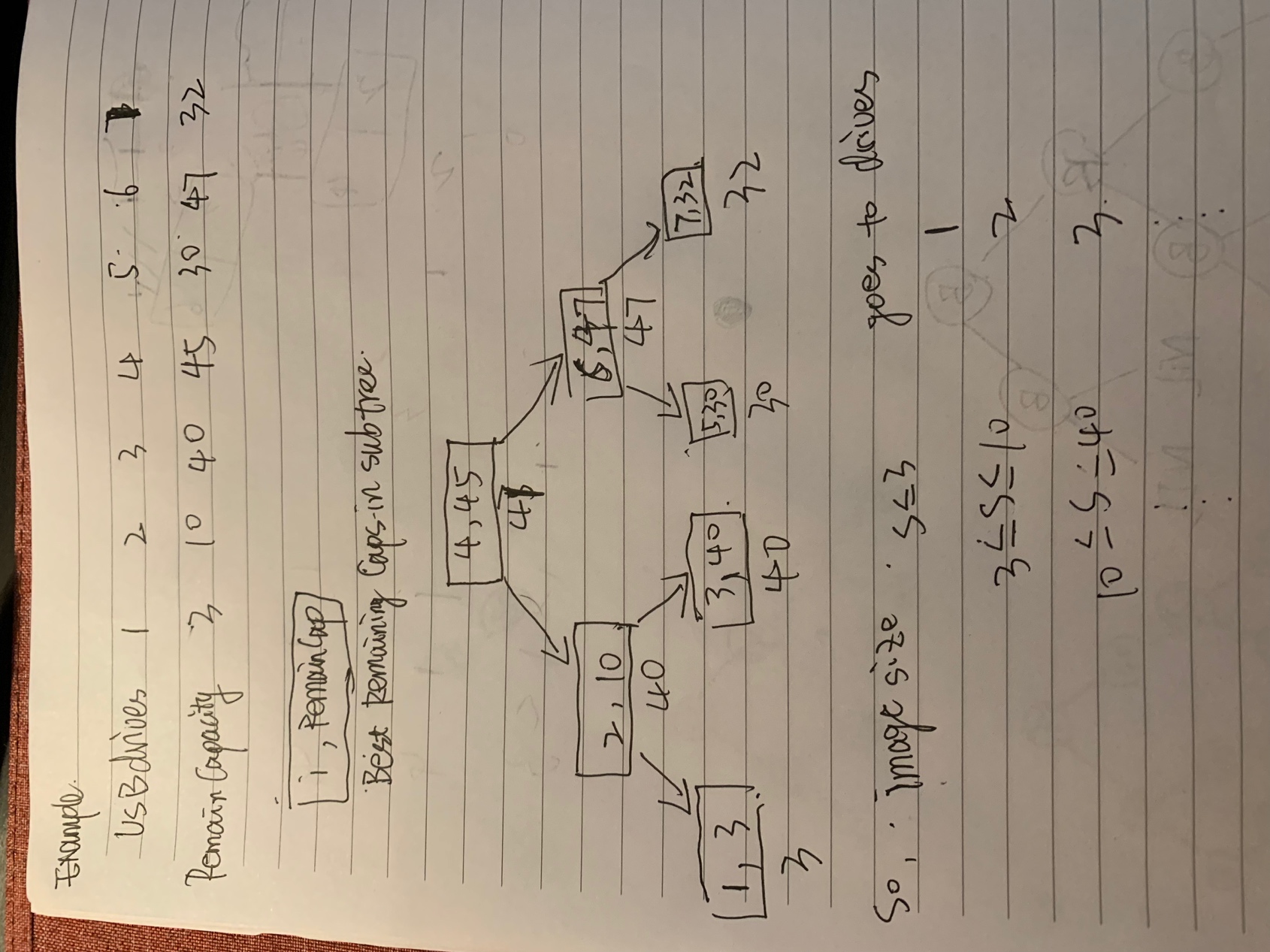
Hence proved.

A-4.6



Answer:

1. Build a red-black tree with height O(log n) for all the items of USB drives，so the drives is like the bin of the bin-packing problem；Each node has three values: index of bin, remaining capacity of bin, and best (largest) in all the bins represented by the subtree rooted at the node；The ordering of the tree is by the bin index.



1. And the algorithm is :

For i from 0 to m ;then

FindThePerfectPositionInRedBlackTree(ImageSize[i]);

End

For the search tree the time complexity is O(logn), and the total time complexity is O(mlogn).