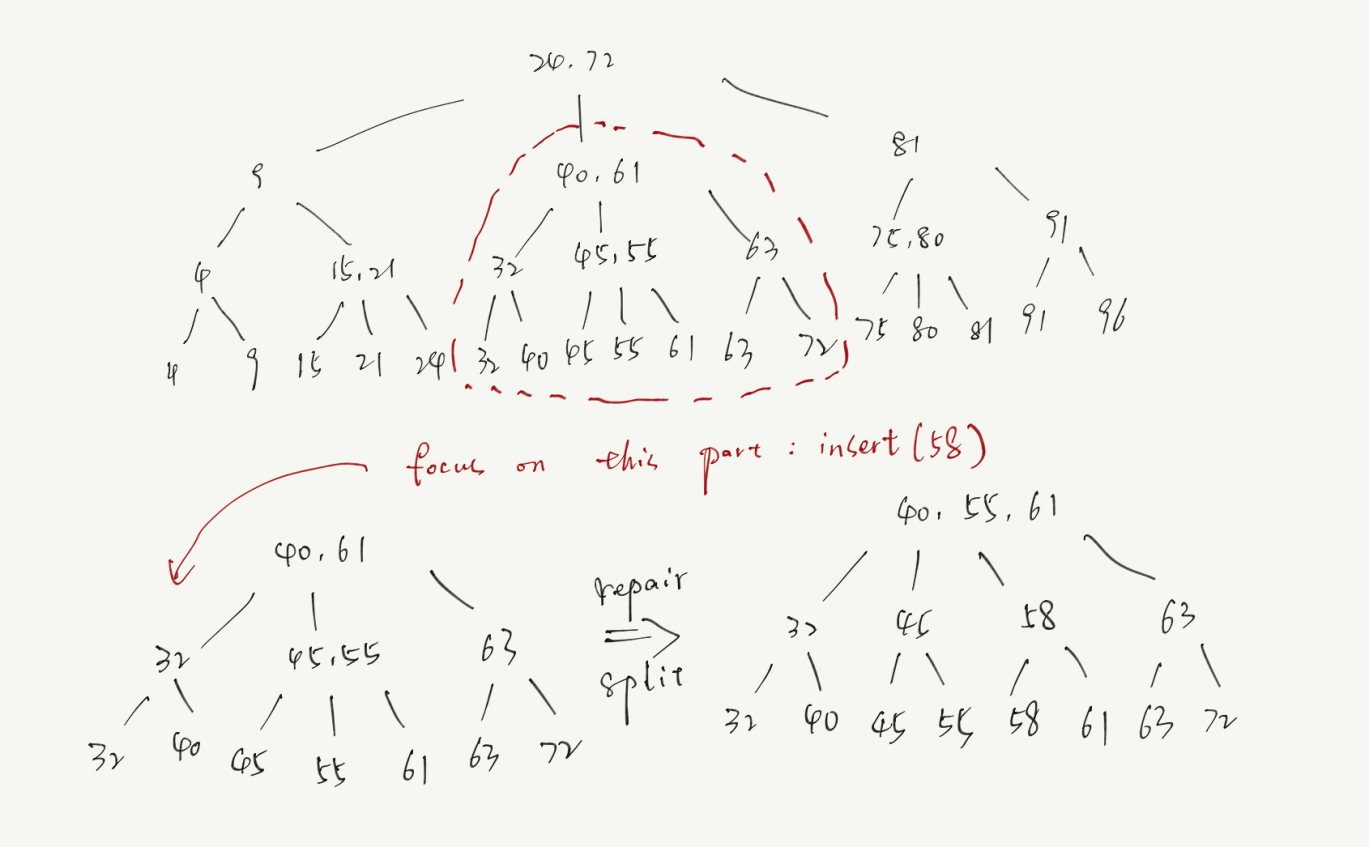
HW5

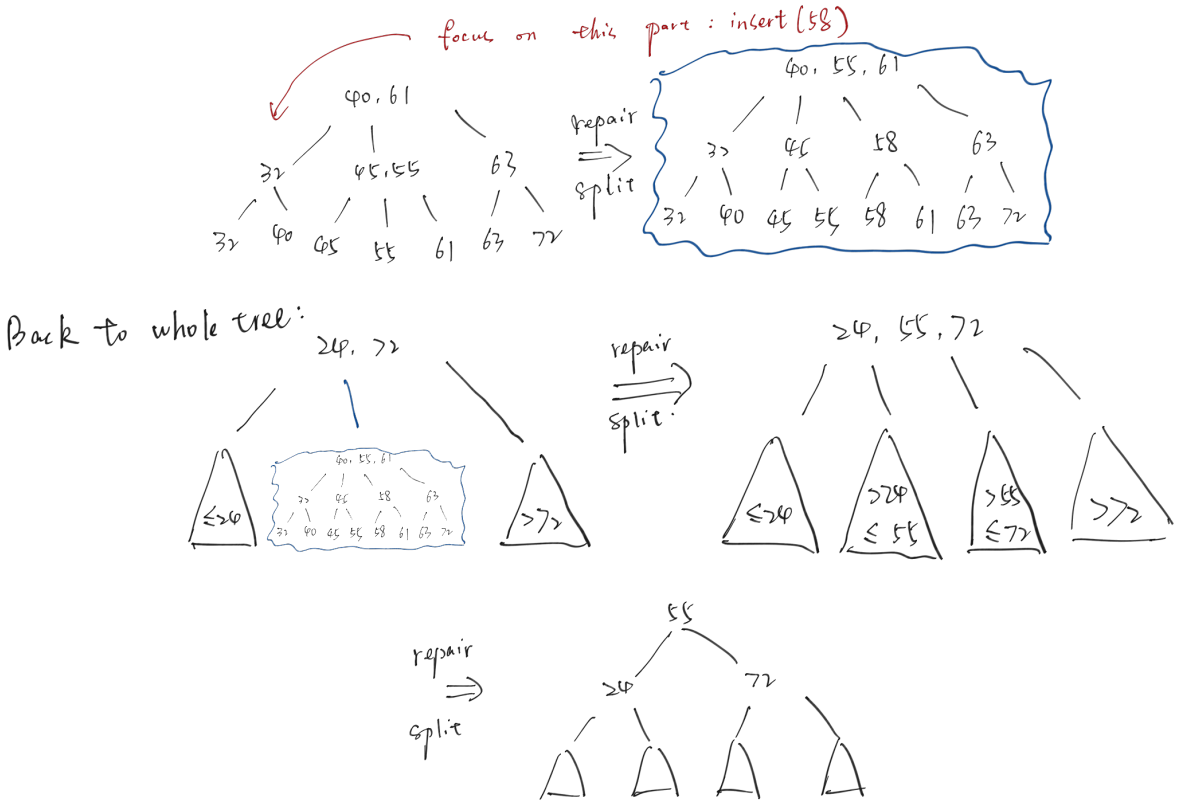
1. *a. Perform the operation Insert(58) on the following 2–3 tree. Show each step of the update.*

   Ans:

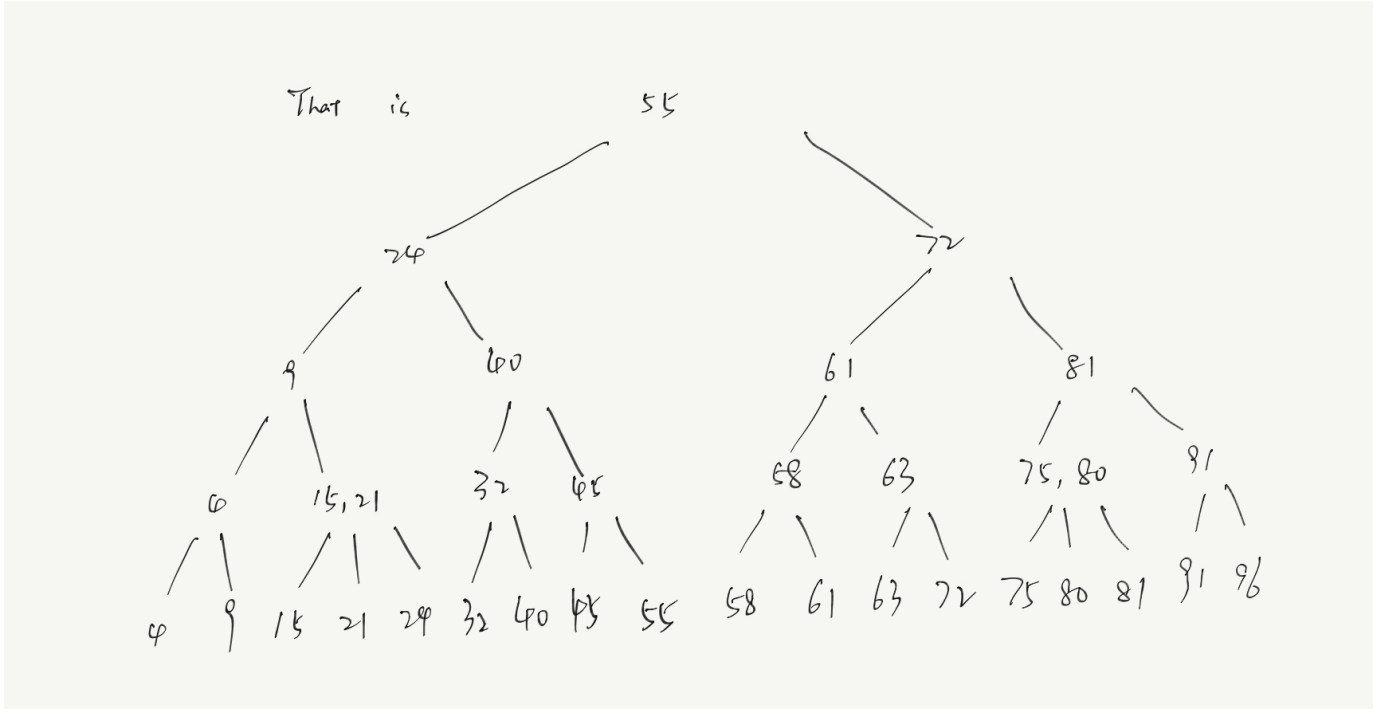
   First, we consider the part that will be changed when inserting



   As its node is more than 4, then continue splitting twice get the final result:



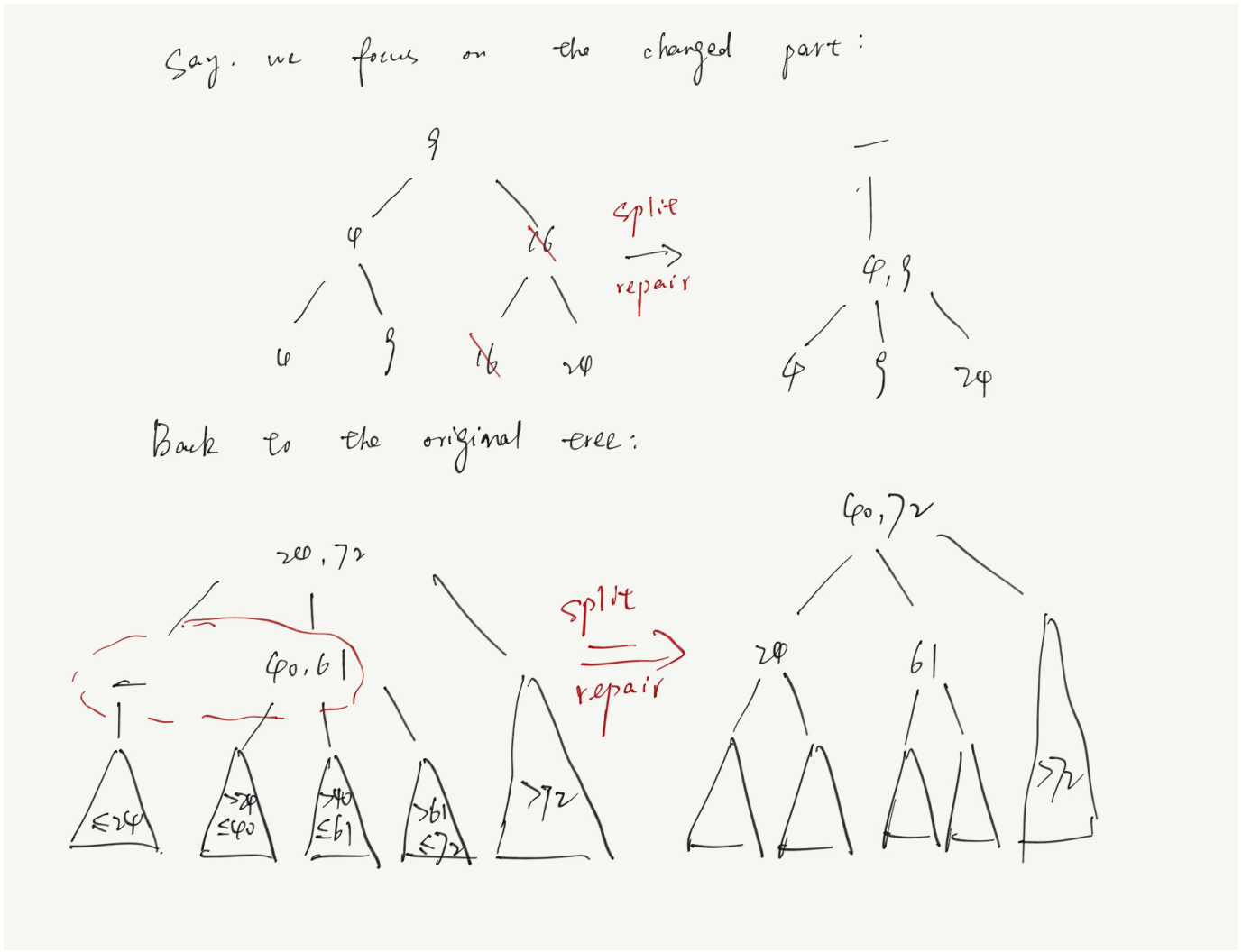
Here is the final result with labels on:



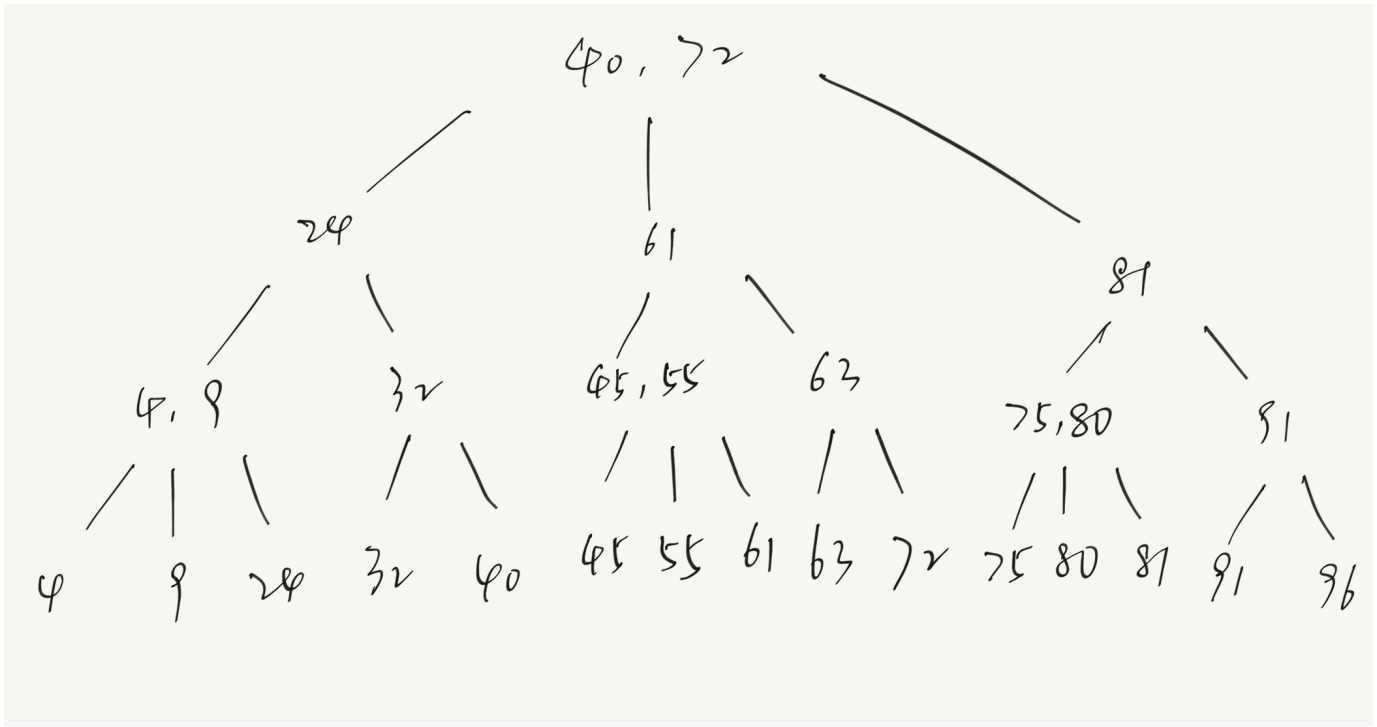
b. Perform the operation Delete(16) on the following 2–3 tree. Show each step of the update.

Ans:

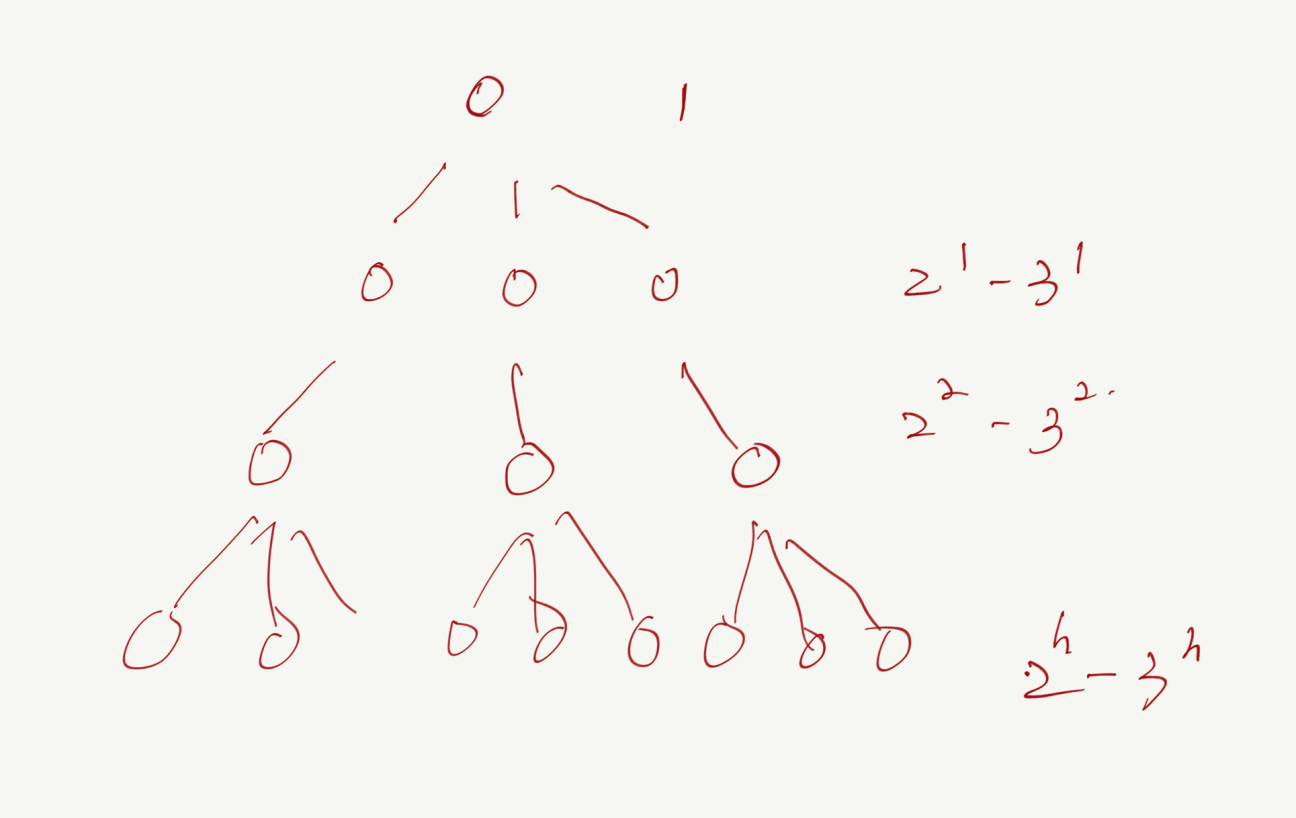
Here is the steps:



The final result with all labels on is:



2. *Give an O(n) time algorithm to output the items in a 2–3 tree in sorted order.*

> T is the 2-3 tree.
>
> The idea is to print the leaves one by one as they are stored in sorted order:
>
> 1. If T is a leaf, then output its item
> 2. Otherwise iterate all its child to recursively output the items in the child.
>
>   As the function is called from the left of the tree to the right, the leaves will be output from left to right.
>
> The first step costs $O(1)$ for each, and the second step excutes $2^{h+1}-1$ times:
>
> > In level 0, OutputItem will excute once.
> >
> > In level 1, OutputItem will excute 2^1 to 3^1.
> >
> > In level 2, OutputItem will excute 2^2 to 3^2.
> >
> > .....
> >
> > In level h, OutputItem will excute 2^h to 3^h.
> >
> > The total cost will be $1+2^1+2^2+\cdots+2^h=2^{h+1}-1$ (Or donated by 3)
> >
> > As $h=\log n$, the cost will be $cn+c$
>
> 
>

```
OutputItem(T)
  if T is a leaf then output T.item
  else do
    for each child w in T do
      OutputItem(T.w)
    end for
  end if
```

Hence, the time complexity of recusive function is $O(n)$

3. *a. Clearly characterize those 2-3 trees which will increase their height when an item with value v is inserted; i.e. provide a specification such that those trees satisfying the condition increase their height when v is inserted, but every tree not meeting the condition keeps the same height.*

   Ans:

   Assume h is the height of 2-3 trees and the leaves of 2-3 trees are between 2^h to 3^h, $i.e.$, number of the leaves n is $2^h \leq n \leq 3^h$.

   1. When $n = 2^h$:

      In this case, the tree has the least number of leaves in the height of h, which means each node has only 2 leaves. There is no possible to increase the height because an inserted item can only make one node have three leaves. No split or reparation needs to operate.

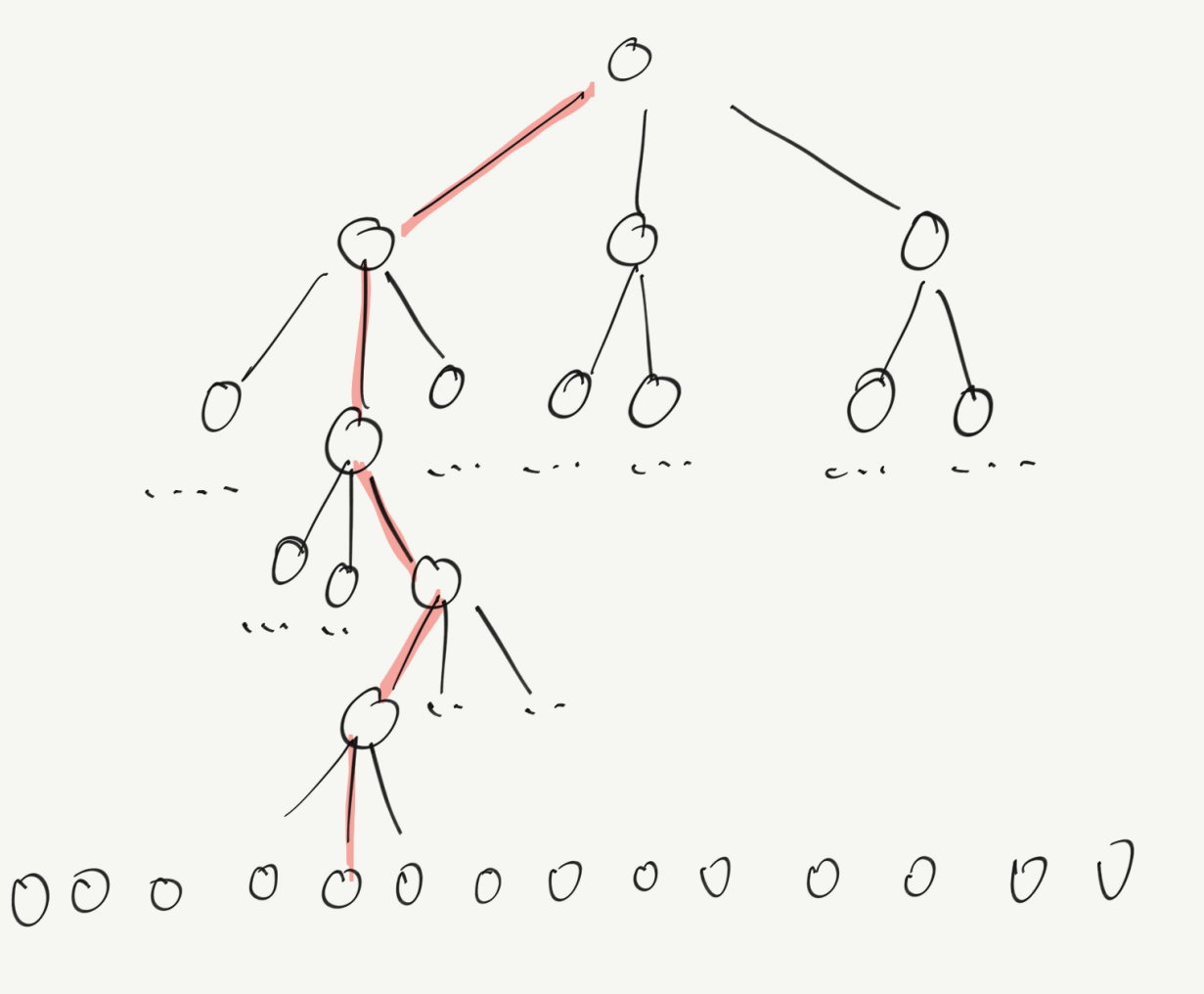
   2. When $n = 3^h$:

      In this case, the tree has the largest number of leaves in the height of h, which means each node has 3 leaves. The height will definately increase if an item is inserted because no more nodes could could an extra item in this height.

   3. When $2^h < n < 3^h$:

      The nodes in this situation at least has one three-leaf node. We can denote it as $n = 2^h + t < 3^h, t$ is the number of 3-leaf nodes. In this case, the necessary condition of increasing height is to fullfil the least requirement, $i.e.$, $n = 2^{h+1}$. If $n = 2^h + t >= 2^{h+1} - 1$, when inserting an item, n would be larger or equal to 2^{h+1}, which meets the least requirement for the next level.

      To increase the height, we have to split and repair the tree whose root has 3 nodes. If the root has only 2 nodes, after splitting, the root has at most 3 nodes. Its height cannot increase. Then, one of the subtrees under the root splits, the height will increase. The root of the subtree also has to have 3 nodes and one of the subtrees has to have 3 nodes. Continue this constraint until the deepest level of the tree. For example:



In short, this tree has to have a path where the each node in it has three decending nodes.

The sum of the leaves are $2(1+2^1+2^2+\cdots+2^{h-1}+1)=2^{h+1}-1$.

there may be several situations:

(a) Insert on a 2-leaf node, height will not increase.

(b) Insert on the particular 3-leaf nodes, the tree will split step by step to the root and increase height

(c) Insert on the other 3-leaf nodes, height will not increase.

Hence, if 2-3 trees (1) have 3^h nodes, the height will increase, or (2) they have to have at least $2^{h-1}+1$ leaves and have at least one particular path (mentioned above) from root to the node in the deepest level and insert an item on this node.

b. Clearly characterize those 2-3 trees which will decrease their height when an item with value v is deleted; i.e. provide a specification such that those trees satisfying the condition decrease their height when v is deleted, but every tree not meeting the condition keeps the same height.

Ans:

Assume h is the height of 2-3 trees and the leaves of 2-3 trees are between 2^h to 3^h, $i.e.$, number of the leaves n is $2^h \leq n \leq 3^h$.

1. When $n = 2^h$:

   In this case, the tree has the least number of leaves in the height of h, which means each node has only 2 leaves. The height will definately decrease if an item is inserted because no more nodes could could an extra item in this height.

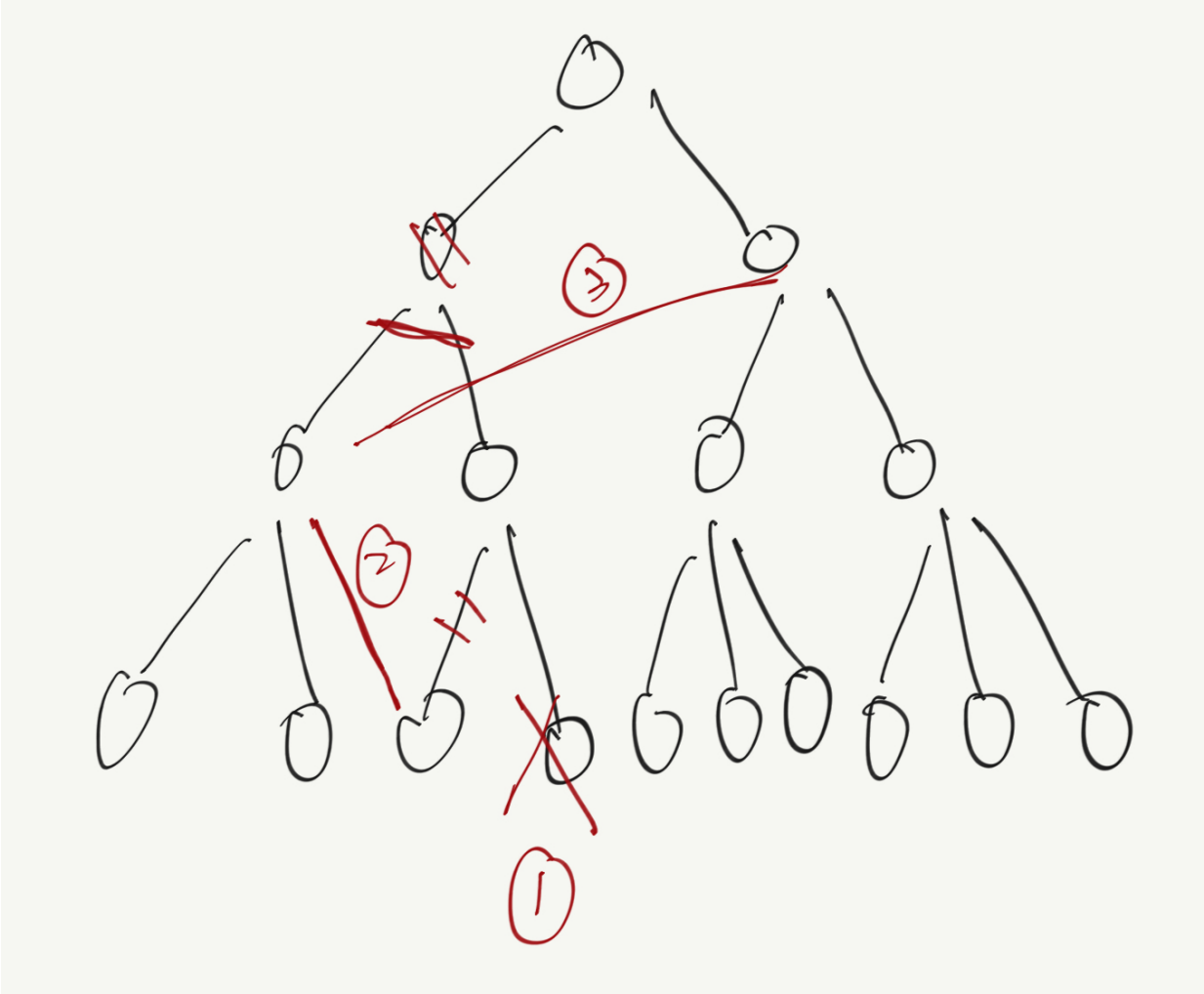
2. When $n = 3^h$:

   In this case, the tree has the largest number of leaves in the height of h, which means each node has 3 leaves. There is no possible to decrease the height because an inserted item can only make one node have two leaves. No split or reparation needs to operate.

3. When $2^h < n < 3^h$:

   The nodes in this situation at least has one two-leaf node. We can denote it as $n = 2^h + t > 2^t, t$ is the number of 3-leaf nodes. In this case, the necessary condition of increasing height is to fullfil the least requirement, $i.e.$, $n = 3^{h-1}$. If $n = 2^h + t <= 3^{h-1}$+1, when deleting an item, n would be less or equal to 3^{h-1}, which meets the least requirement for a smaller level.

   To decrease the height, we have to split and repair the nodes from the bottom to the root. The node deleted must be on a 2-leaf node because otherwise no split will happen. When deleting the leave, the silbings should not have 3 nodes, otherwise they will combine together and the height will not decrease. Continue splitting and repair until to the root, the height will decrease. For example:



The silbings and its parent of deleted node should have exactly 2 nodes as well as the nodes in level 0 and 1, thus the split and repair can continue and the subtree can be combined into the right node in level 1. The nodes in the right subtrees after level 1 have no constraints.

In short, the root has to have exactly two childs. The chidren of one of them has two 2-3 subtrees and the other one perform this recursively.

4. *Consider maintaining a database for a biologist storing results concerning pea plant experiments. Each experiment records a distinct (integer) id and a plant height. We would like to support the following query:*

   How many plants have heights in the range [h1,h2]?

   Ans:

   We create two 2-3 trees to support the query:

   (1) We use id as the key for each experiment record to store it in the 2-3 tree. Each experiement record has two fields: id and height.

   (2) And then we also use height as the key for each experiment record to store it in the 2-3 tree. Each experiement record has two fields: id and height. For each subtree, we have an additional field, the number of the records in the subtree, and store this number at the each node like what we learned in class.

In these cases, the insert and delete queries will be in time $O(\log n)$ in this two trees.

We have known augmenting 2-3 trees can be used to seach kth smallest item in rank order, so here we use the same tree to know the rank of the searched item, i.e., the k. Take the tree in (2) for example, assume the root has three subtrees holding $n1, n2$ and $(n - n1 - n2)$ items.

Idea: compare h_1 with guides to decide which subtree to go, and plus the number of prior subtree to the rank k. For example, h_1 is in the range of the second subtree, then k + n_1. Continue recursively, we will find item which is the smallest item that larger than h_1 and its rank. Say it's k_1th smallest item in the database. The operation is similar to the search in original 2-3 tree, so the cost is $O(\log n)$. Samilar way to get the largest item that smaller than h_2. Say it's k_2 smallest item in the database.

For $Insert(id, ht)$: perform two operations in tree (1) and (2)

1. Insert(id, ht) in the first tree. The cost is $O(\log n)$.
2. Insert(ht, id) in the second tree. The cost is $O(\log n)$.

The total cost will be $O(\log n)$.

For $Delete(id)$: perform two operations in tree (1) and (2)

1. Delete(id) in the first tree and get its ht. The cost is $O(\log n + c) = O(\log n)$.
2. Delete(ht) in the second tree. The cost is $O(\log n)$.

The total cost will be $O(\log n)$.

For query: perform operations in tree (2)

1. Find the smallest item which is bigger than h_1, whose rank is k_1. The cost is $O(\log n)$.
2. Find the largest item which is smaller than h_2, whose rank is k_2. The cost is $O(\log n)$.
3. The number of plants have heights in the range $[h_1, h_2]$ is $k_2 - k_1 + 1$