Notes
on
Design and Analysis
of
Algorithm

By

Dr. Satyasundara Mahapatra

(Module –II)

Advanced Data Structures: Red-Black Trees, B – Trees, Binomial Heaps, Fibonacci Heaps, Tries, Skip List

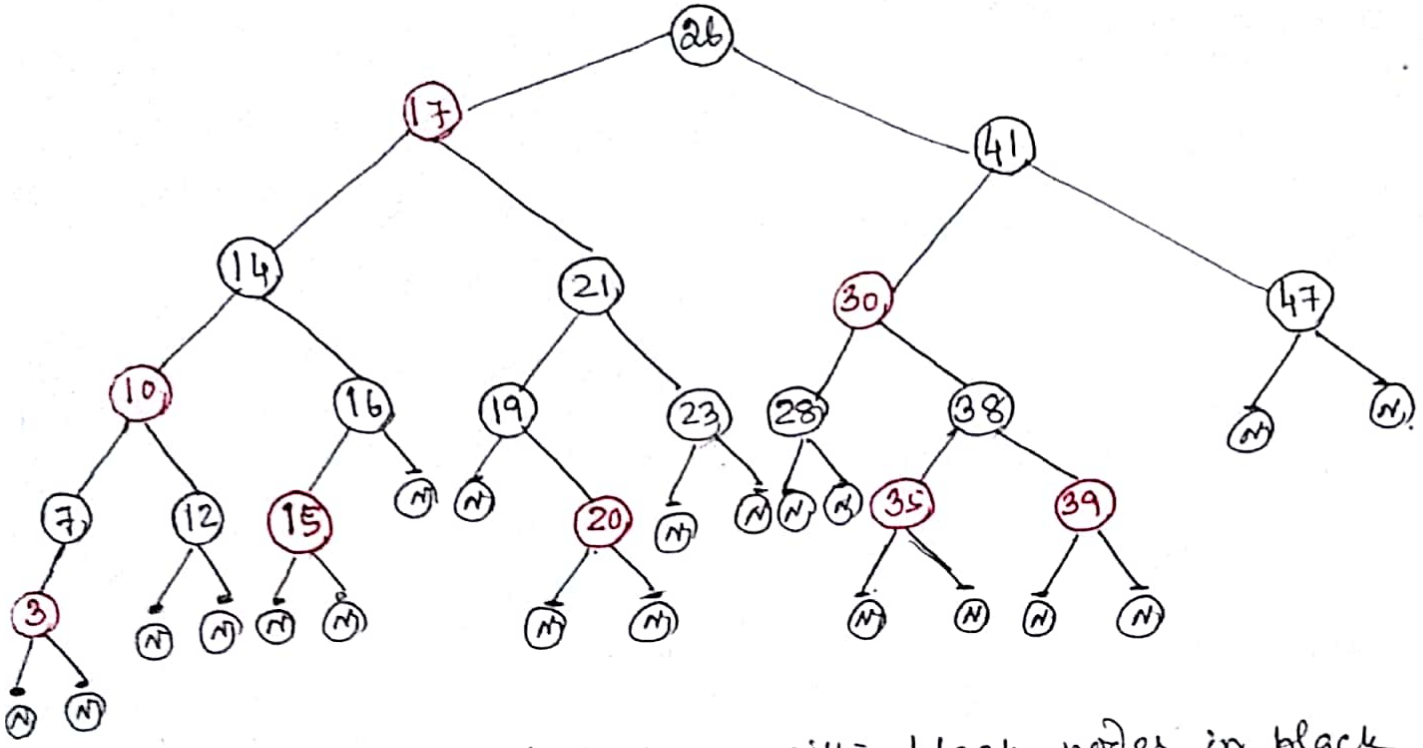
Red-Black Trees

A red black tree is a binary search tree (BST) with one extra bit of storage per node: i.e its color, which can be either Red or Black. Hence each node of the tree contains the following attributes. i.e color, key or item or data, left, right and p. (note: if a child or parent of a node does not exist, the corresponding pointer attributes of the node contain the value NULL.

A red-black tree is a binary search tree that satisfies the following properties:-

1. Every node is either red or black.
2. The root is always black.
3. The every leaf node (i.e the node who have no child) is always black.
4. If a node is red, then both its children is always black.
5. For each node, all simple paths from the node to descendent leaves contain the same number of black nodes. (i.e otherwise known as black-height (bh) of a node denoted by bh(node).

An example of red-black tree is given on the next page.



(Figure: A red-black tree with black nodes in black color and rednodes in red color.
(λ) ← Red nodes (λ) ← Black nodes.

Rotations

. The search operations on a red-black tree with n keys or items take O(lg n) time. These operation are basically two types. They are Tree-Insert and Tree-delete. These operations modify the tree and the results may violate the properties of Red-black tree. So to restore these properties, there is a need to change the color of some nodes in the tree and also change the pointer structure of the tree.

The change of pointer structure was done through rotation. which is a common operation in RB-Tree for preserving the property of BST as well as RB-Tree. These

Rotation are basically two types. They are left rotation and right rotation.

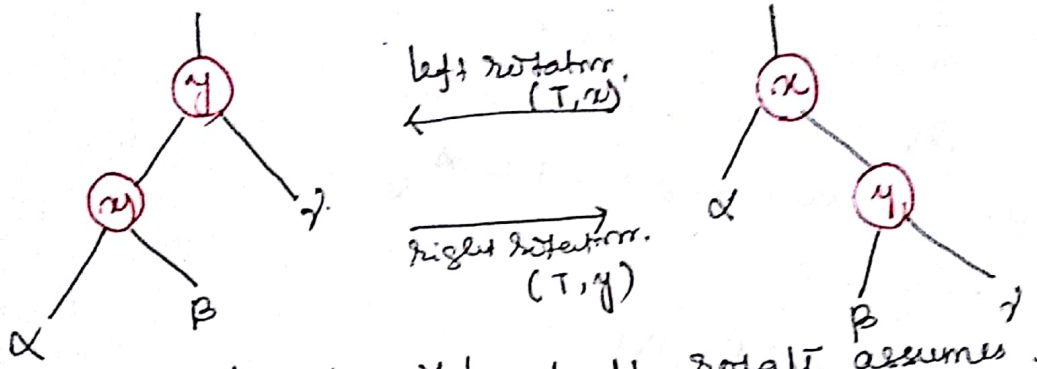
Left rotation

When a left rotation on a node is was performed,

→ it will be assumed that the right child of node x is not null.

(i.e x may be any node in the tree T whose right child is not T→null)

→ the left rotation is from x to y, as shown in below figure.



The pseudocode for Left-rotate assumes that $x \rightarrow right != T \rightarrow NULL$ and the parent of root is $T \rightarrow NULL$.

```
Left Rotate (T, x)
1. y = x → right          // set y
2. x → right = y → left.  // transfer y → left subtrees to the right of x.
3. if y → left != T → NULL
4.     y → left → P = x
5. y → P = x → P.
6.
```

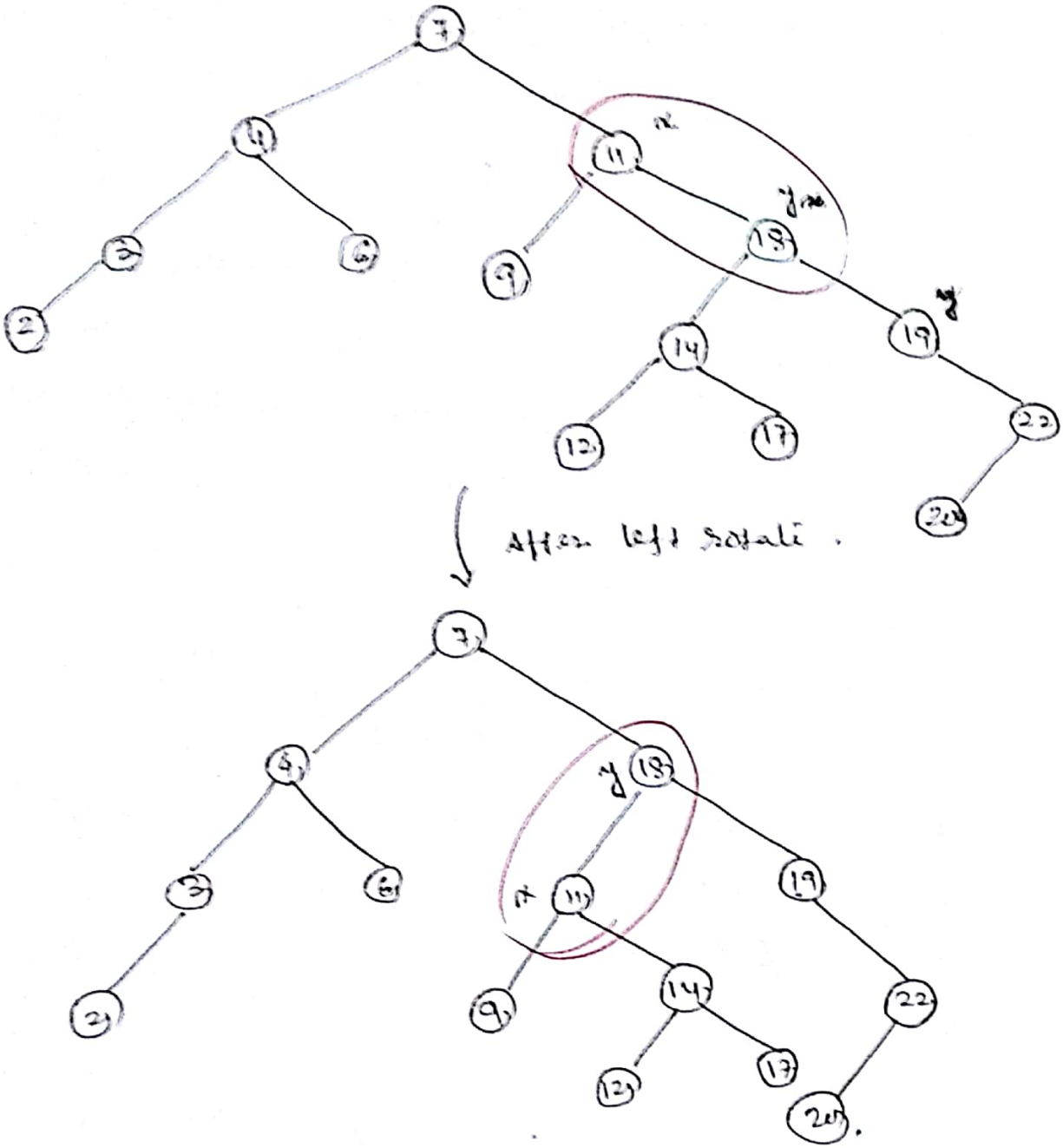
```
6. if x->p = T.NULL
7.       T->root = y
8. else if x == x->p->left
9.       x->p->left = y
10. else x->p->right = y
11. y->left = x
12. x->p = y
```

The right-rotate algorithm is symmetric.

```
Right Rotate (T, y)
1. y = x->left.
2. x->left = y->right
3. If y->right = T->NULL
4.       x->y->right->p = x
5. x->p = y->p.
6. If y->p = T->NULL
7.       T->root = y
8. else if y == y->p->right
9.       y->p->right = x
10. else  y->p->left = x
11. x->right = y
12. y->p = x
```

Note: Both Left-Rotate and Right-Rotate run in O(1) time.

An example of Left Rotation is given below.



Insertion:

→ Insert a node into an n-node red-black tree takes O(lg n) time.

→ A slightly modified version of Tree-Insert procedure for BST is used.

→ where node z is inserted into the BST Tree T and then the color of z changed to red.

→ The an auxiliary procedure named as RB-Insert-fixup() is called to Preserved the Red-black properties by Recoloring the nodes and performing rotations.

→ An auxilary procedure RB-Insert fixup() is used to recolor nodes and perform rotations for preserving the properties of red-black tree.

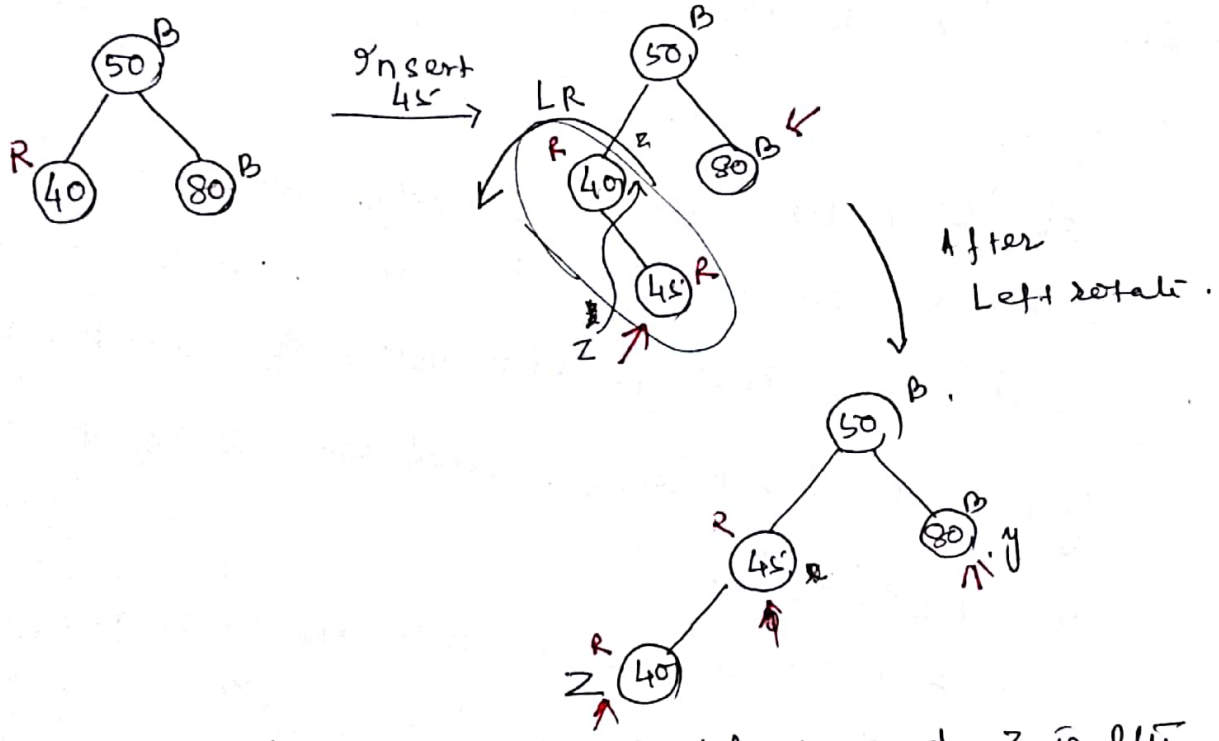
(Note: Assumed that, the key value of insert node z is already fillup.)

```
RB-Insert(T,z)
1.  y = NULL
2.  x = T → root.
3.  while (x != NULL)
4.        y = x
5.        if z → key < x → key
6.              x = x → left.
7.        else  x = x → right.
8.  z → p = y
9.  if y = NULL
10.       root = z
11. else if z → key < y → key
12.       y → left = z
13. else y → right = z.
```

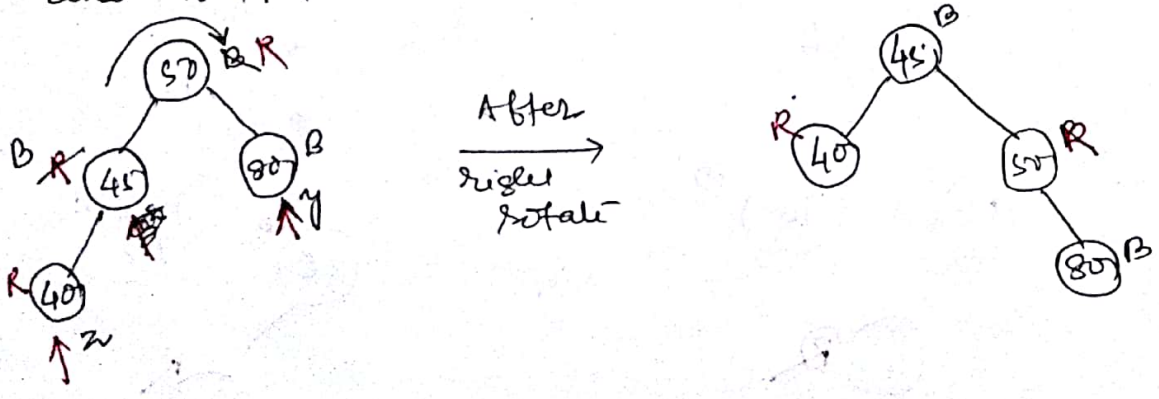
Case-2

If y (uncle of x) is black and z is the right child of its parent then new value of z is the parent of z i.e z=z→p or z=p[z] and Left Rotate (T, z).

Let explain with an example.



Case-3 If y (uncle of x) is black and z is the left child of its parent then change the color z→p or p[z] and z→p→p or p[p[z]] and Right_Rotate (T, z→p→p)



14. z → left = NULL

15. z → right = NULL

16. z → color = red

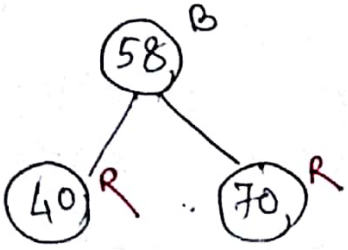
17. RB-Insert-fixup(T, z)

The rebalancing in RB-Tree at the time of insertion follow three cases. They are.

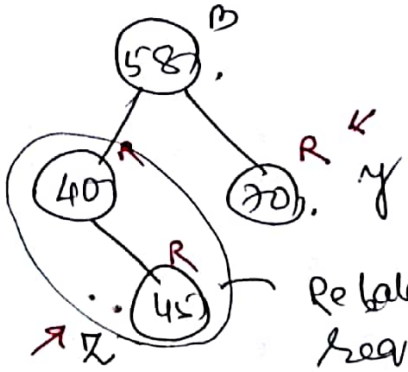
Case 1:

If 'z' is the inserting node whose color is always red and 'y' (the uncle of z) is also red then change the color of z → p → p (i.e p[p[z]]), y, z → p (i.e p[z]). and then z = z → p → p i.e, z = p[p[z]].

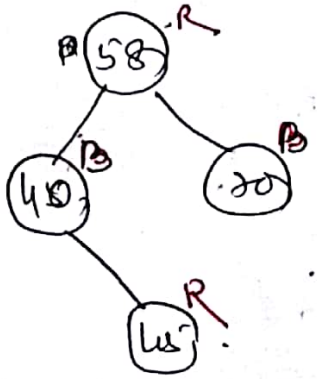
Let us explain with an example.



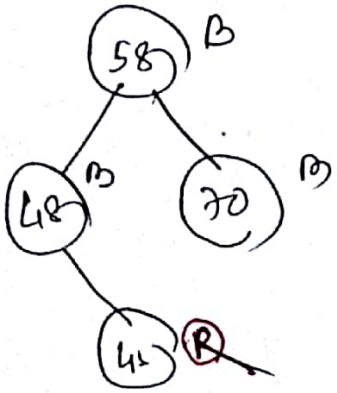
Insert 45



↓ apply case 1.



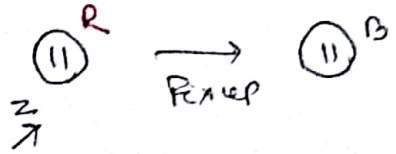
←



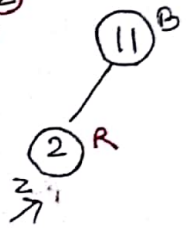
Q. Insert the following elements into an empty RB-Tree

<11, 2, 14, 1, 7, 15, 5, 8, 4>

Insert-(11)

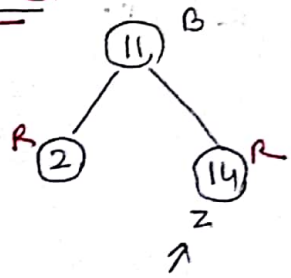


Insert-(2)



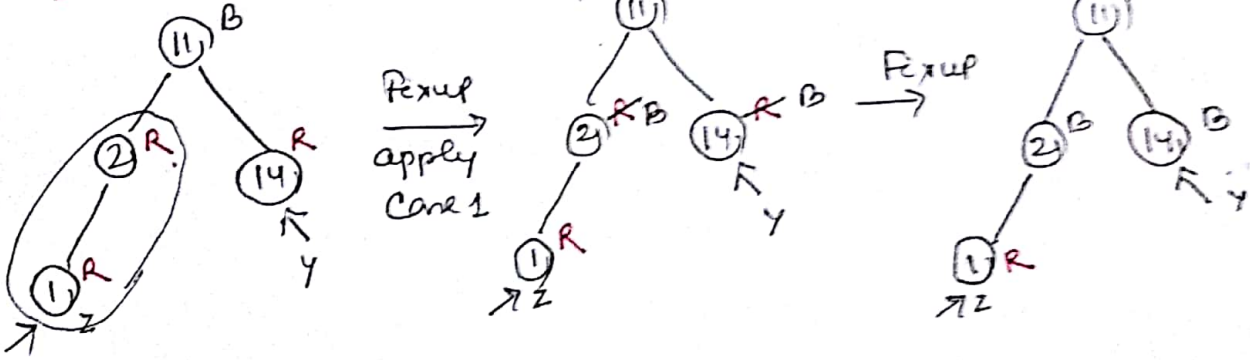
→ no fixup Required.

Insert-(14)

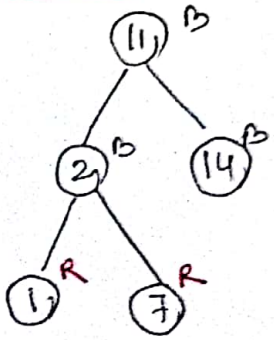


→ no fixup Required.

Insert-(1)

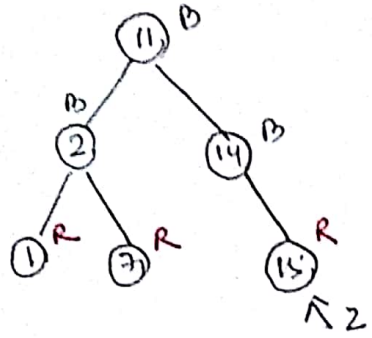


Insert-(7)



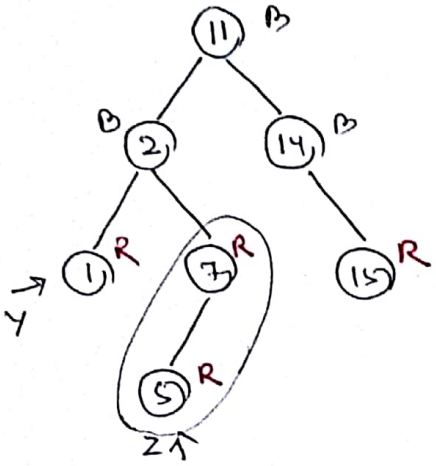
→ no fixup required

Insert - (15)

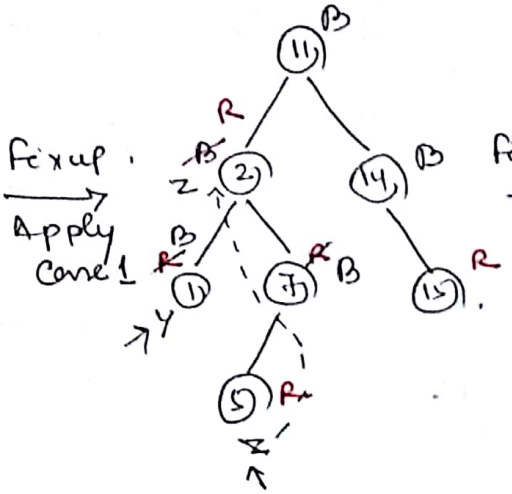


→ (no fixup required)

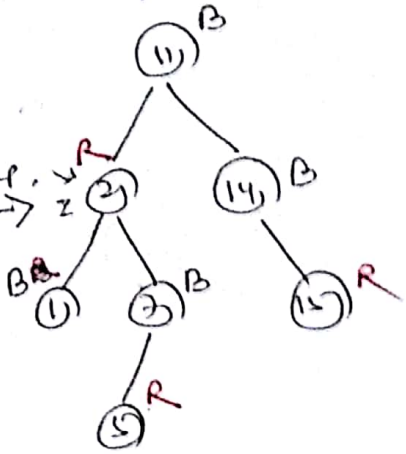
Insert - (5)



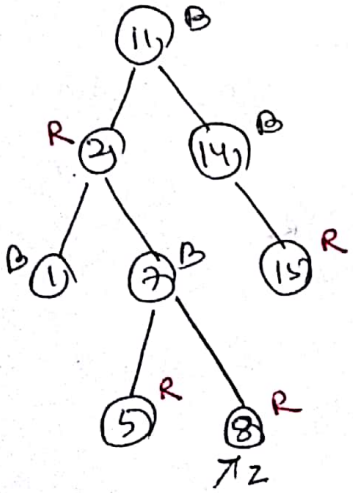
fixup → Apply Case 1



fixup →

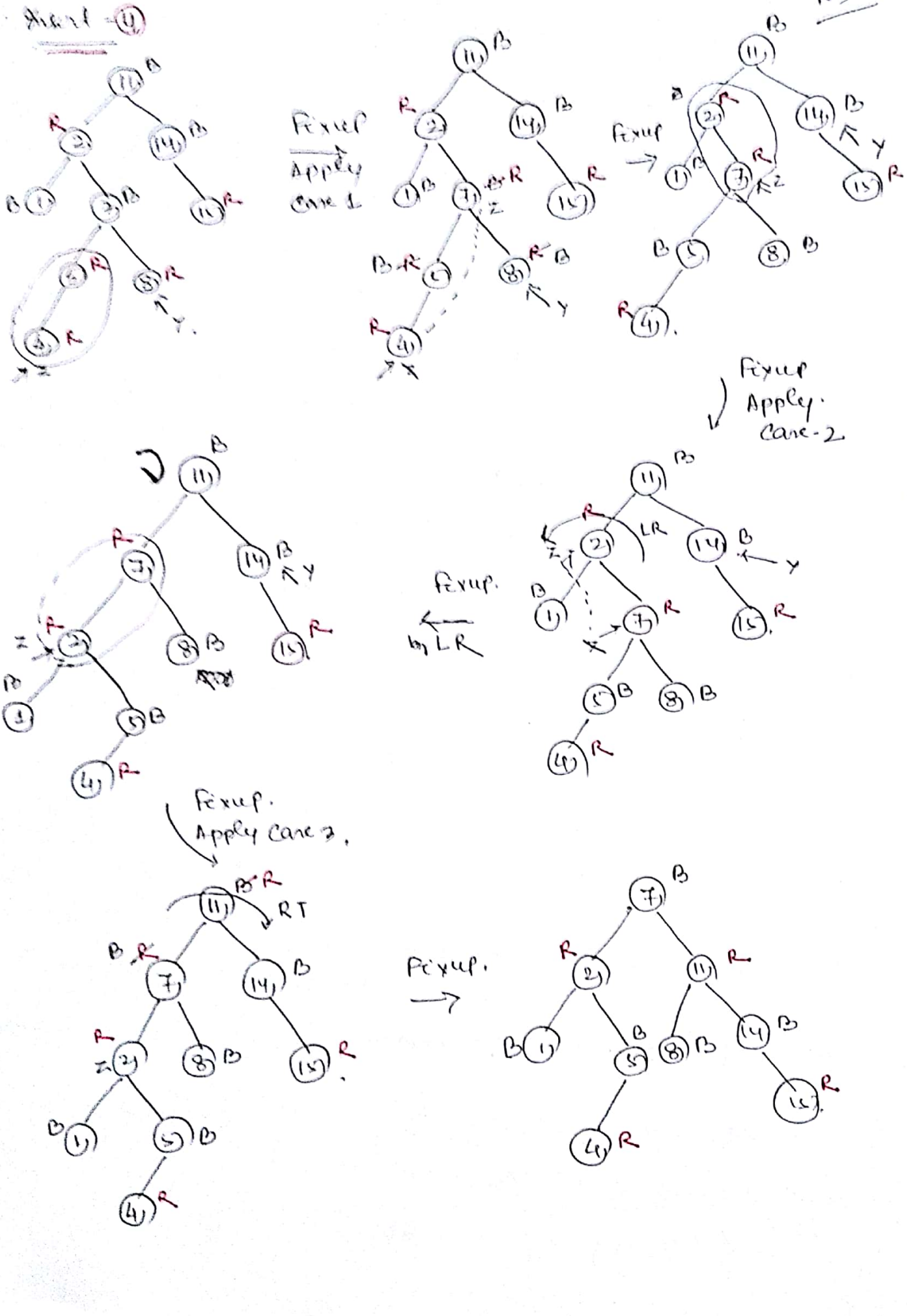


Insert - (8)



→ (no fix up required)

RB-6



RB-Insert-fixup (T, z)

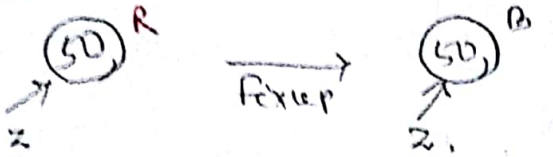
1. while $z \to p \to color = Red$
2. if $z \to p = z \to p \to p \to left$
3. $y = z \to p \to p \to right$.
4. if $y \to color = red$
5. $z \to p \to color = Black$
6. $y \to color = Black$
7. $z \to p \to p \to color = Red$
8. $z = z \to p \to p$.
9. else if $z = z \to p \to right$.
10. $z = z \to p$.
11. Left-Rotate (T, z)
12. $z \to p \to color = Black$
13. $z \to p \to p \to color = Red$
14. Right-Rotate (T, $z \to p \to p$)
15. else (same as then clause with right and left exchanged) (*) Available on next page problem.
16. $T \to root \to Color = Black$

Q. Insert the following keys in a initially empty red-black tree.

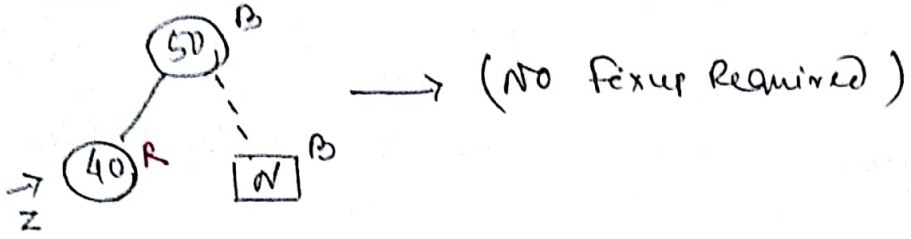
⟨ 50, 40, 30, 45, 20, 5 ⟩

The keys are < 50, 40, 30, 45, 20, 5 >

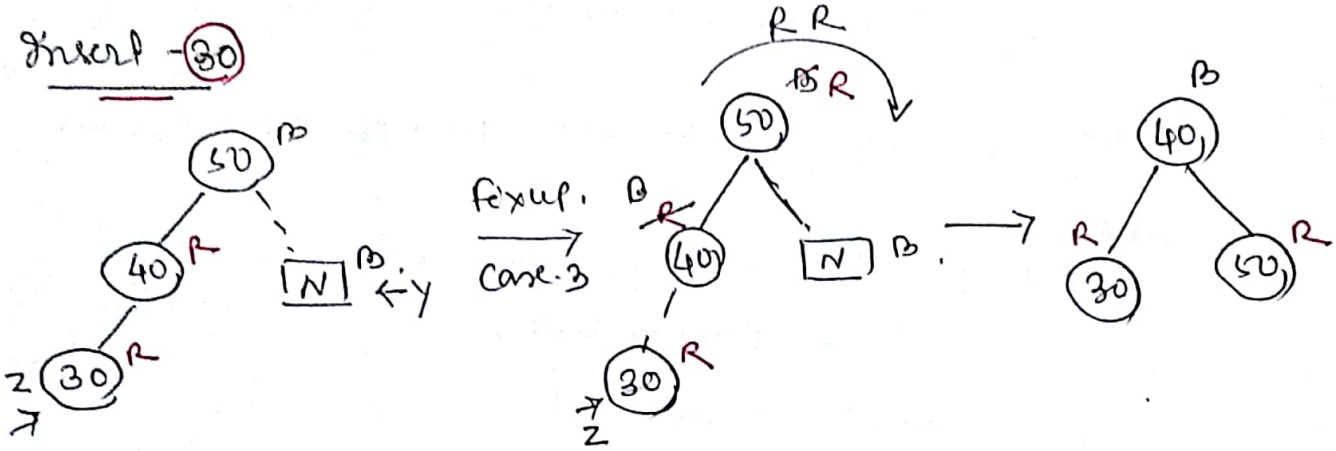
Insert - 50



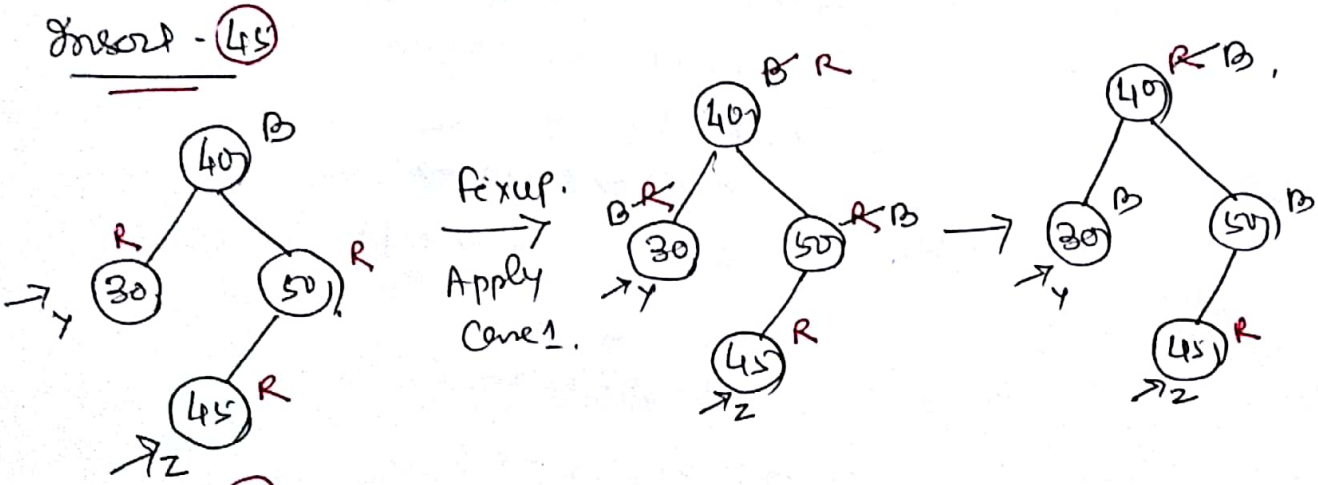
Insert - 40



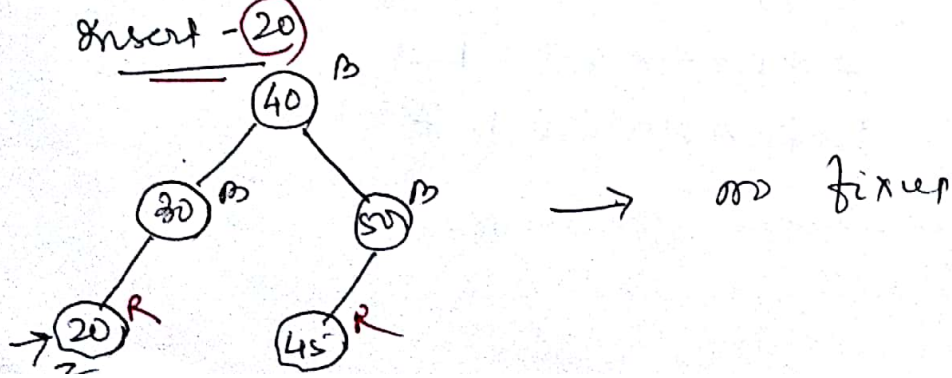
Insert - 30



Insert - 45

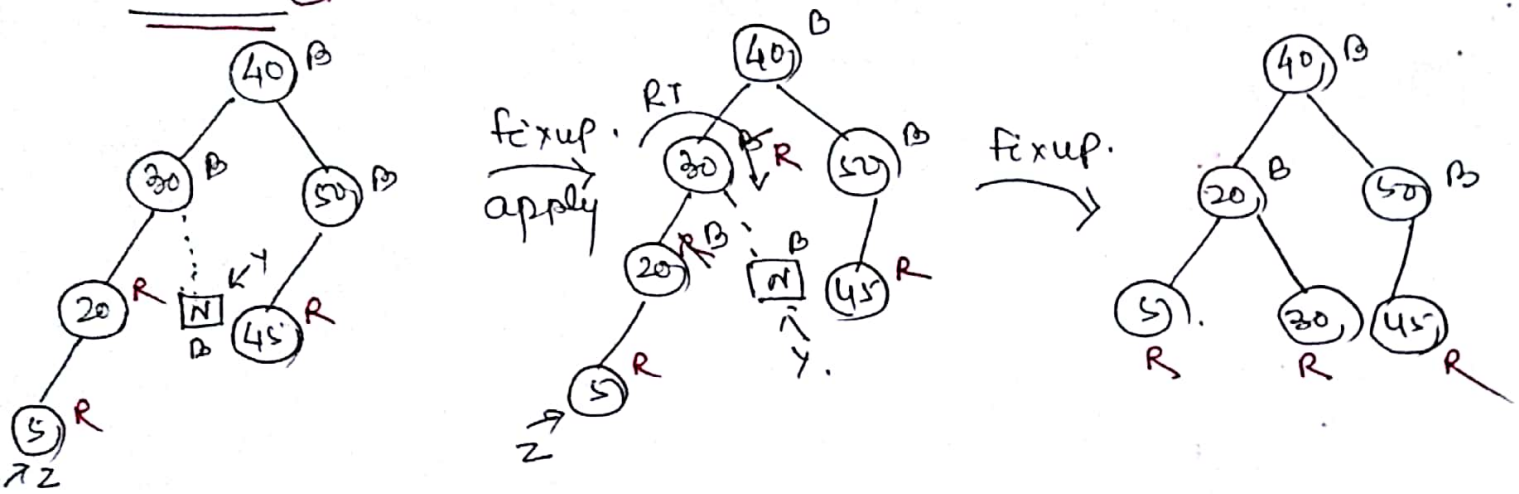


Insert - 20



→ no fixup required.

Insert - (5)



Q. Insert the following elements in an empty (Initially) RB Tree.

$\langle 4, 7, 12, 15, 3, 5, 14, 18 \rangle$

(*) Algo of the line is (i.e the else part of RB-Insert-Fixup(T,z)

```
else if z -> p = z -> p -> p -> right.
        y = z -> p -> p -> left.
        If y -> color = Red
            z -> p -> color = Black
            z -> p -> p -> color = Red
            y -> color = Black
            z = z -> p -> p
        else if z = z -> p -> left
                z = z -> p.
                Right rotate (T, z)

            z -> p -> color = Black
            z -> p -> p -> color = Red
            Left rotate (T, z -> p -> p)
```

now execute the above problem.

Deletion in RB-Tree.

Initially, we delete a node for RB-tree as we delete a node in normal BST. as the algorithm given below.

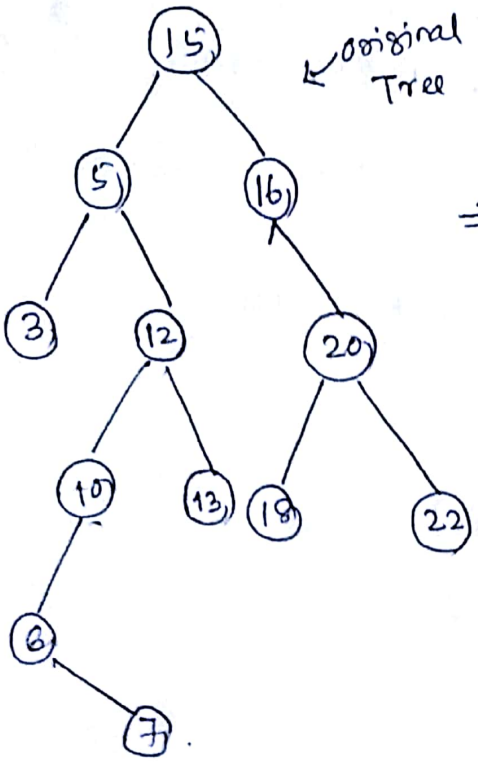
RB-Delete (T, z)

```
if z→left = NULL or z→right = NULL
    then y ← z
else
    y = Tree-Successor(z)
if y→left ≠ NULL
    then x = y→left
else
    x = y→right.

x→p = y→p.
if y→p = NULL
    then T→root = x
else if y = y→p→left.
    then y→p→left = x
else
    y→p→right = x.

if y ≠ z
    then z→key = y→key
        (i.e copy data of y to data of z)
if y→color = Black
    then call RB-Delete-fixup (T, x)
return y.
```

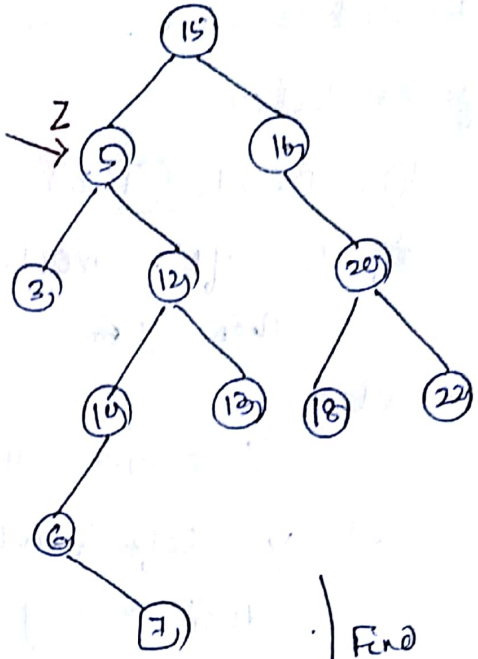
Let us execute the RB-Delete(T,z) with the help of an example.



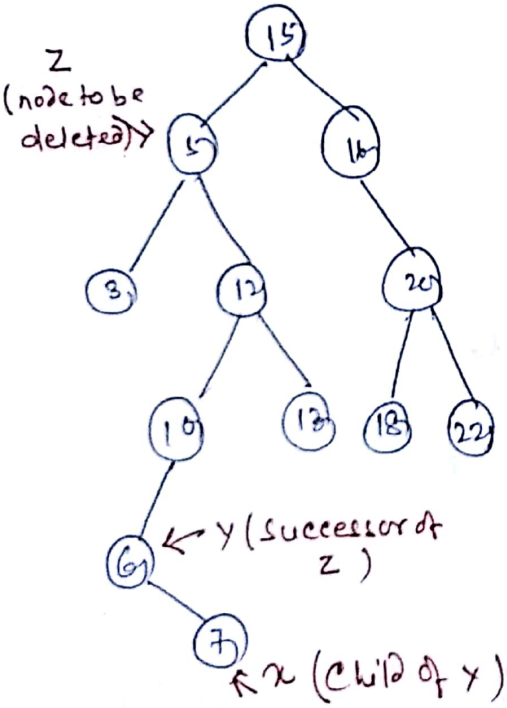
original Tree

⇒

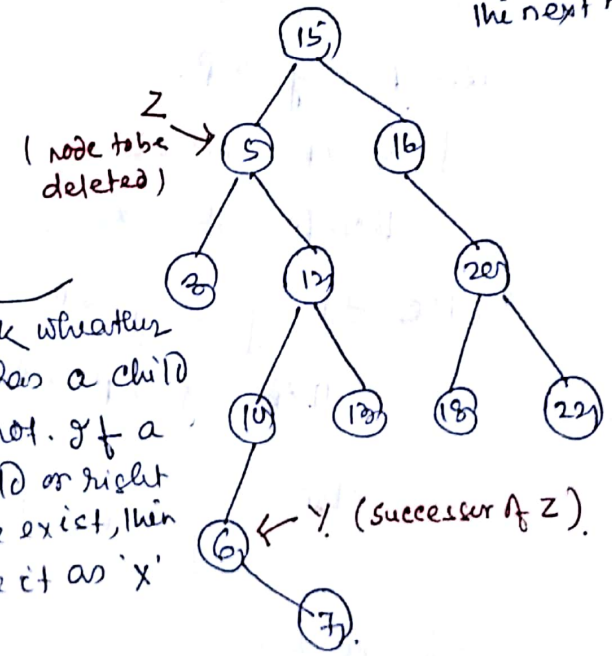
node z want to delete



Find Successor, i.e y' the next no. of z



Check whether y has a child or not. If a child or right side exict, then name it as 'x'



Basic Idea for deletion: (Move the extra black up to the tree until)

→ x point to a red & black node ⇒ turn it into black node

→ x points to the root ⇒ just remove the extra black

→ we can do certain rotations and recoloring the nodes and finish.

~~delete from~~

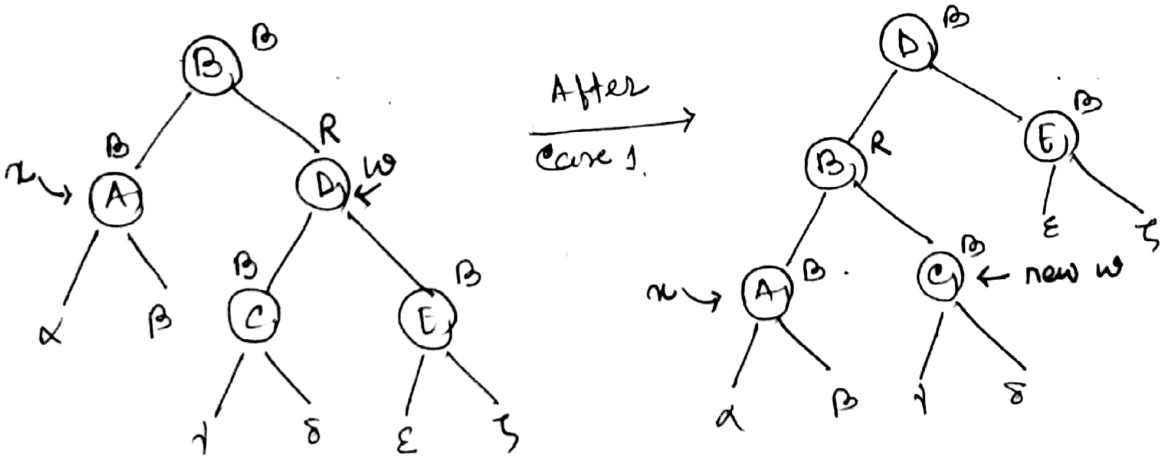
We execute the above idea with the help of RB-Delete-fixup function, and the function is executing until $x \neq root$ and $x \to color = Black$

For that we first find the sibling of 'x' as 'w' and 'w' can not be NULL.

There are 8 cases, 4 of which are symmetric to the other 4. (like RB-Tree-Insertion fixup).

We will look at cases in which x is the left child of it's parent.

Case 1. when 'w' (sibling of 'x') is red, and w is x→p→right



~~Rules for Case 1~~

~~→ w must be a black have black children~~

~~→ Make w → color = Black and x → p → Color = Red.~~

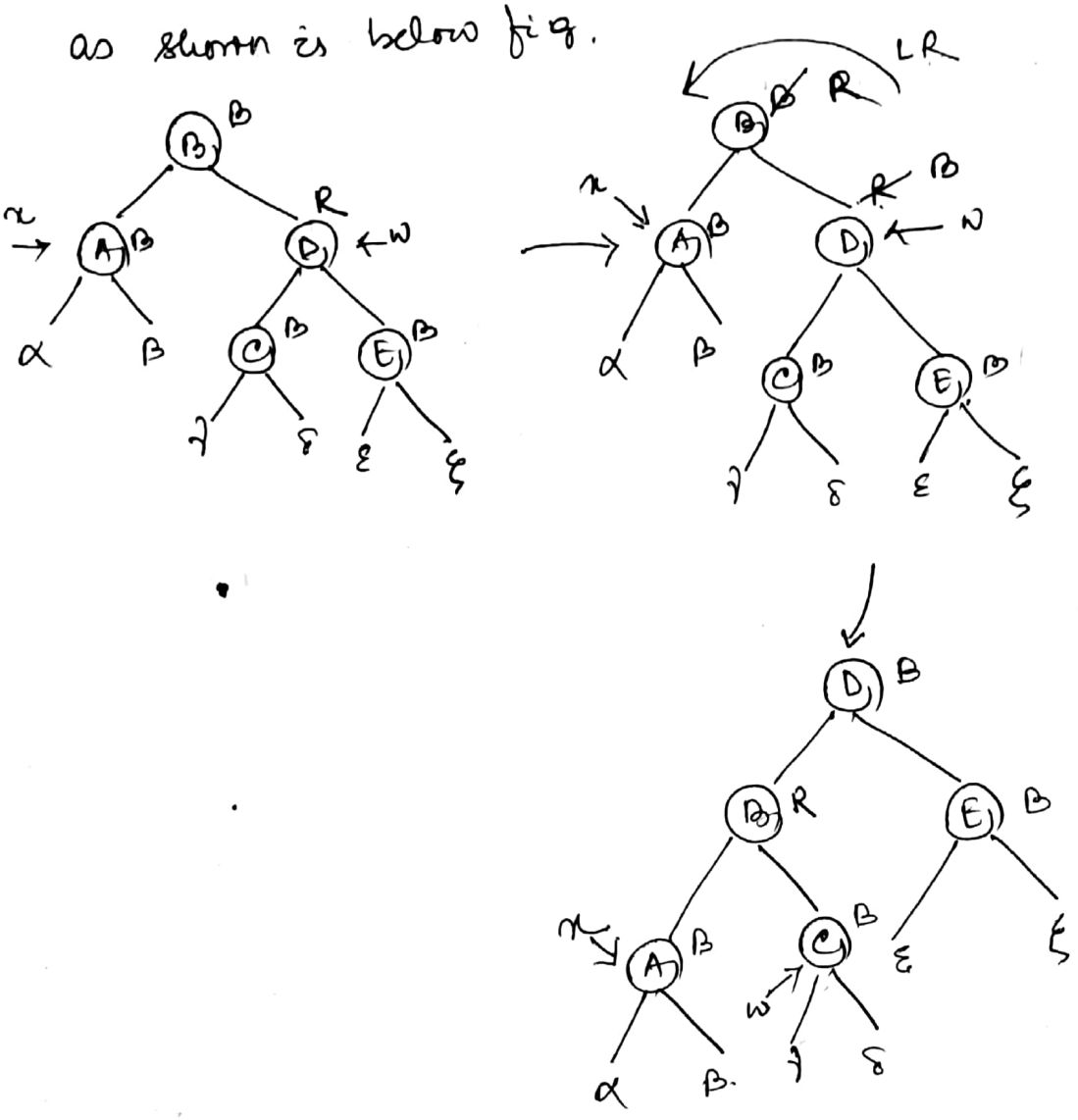
~~→ Then left rotate on x → p.~~

~~→ New sibling~~

Case 1 Rules.

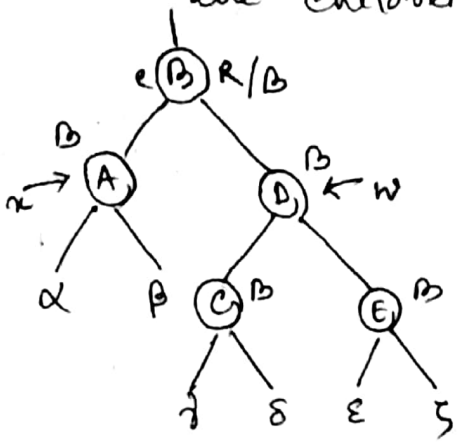
1. change the $w \rightarrow$ color = Black
2. change the $x \rightarrow p \rightarrow$ color = Red
3. Left - rotate $(T, x \rightarrow p)$
4. Move $w = x \rightarrow p \rightarrow$ right.

as shown in below fig.



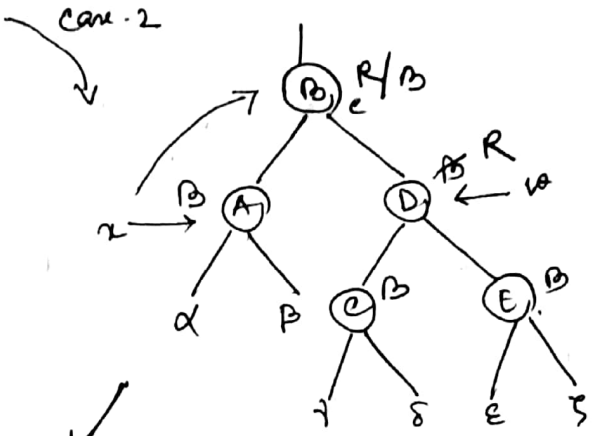
Case 2: When 'w' (Sibling of 'x') is black and both the children of 'w' is also black.

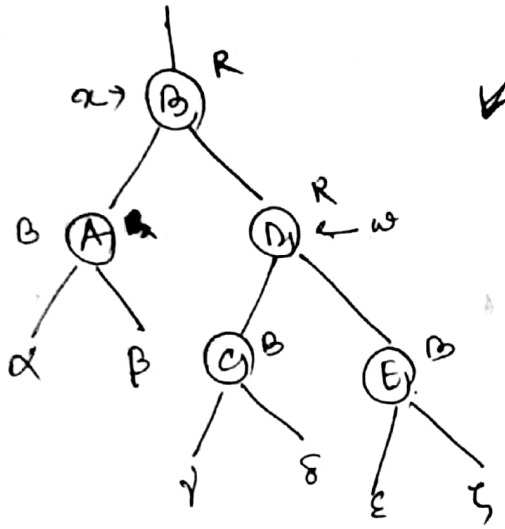
(Note: node with R/B represent unknown color represented by e)



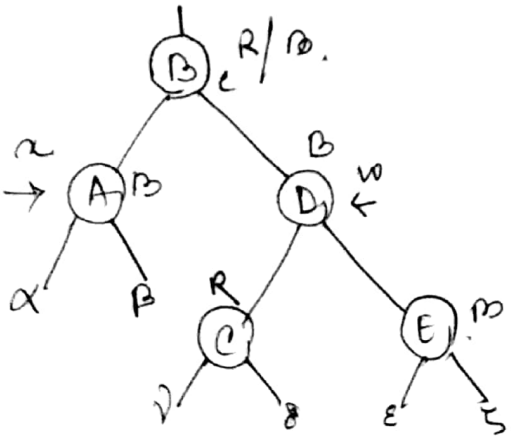
Rule of Case 2

1. Make $w \rightarrow color = Red$
2. Move $x \leftarrow x \rightarrow p$.



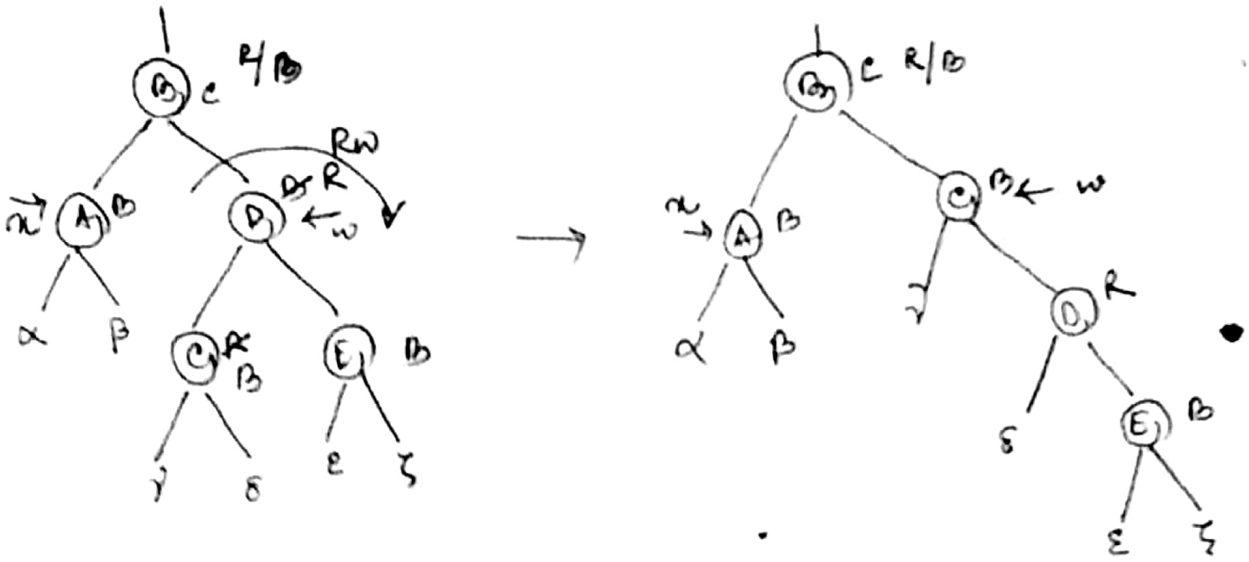


Case-3: When 'w' (Sibling of 'x') is black, w's left child is red, and w's right child is black

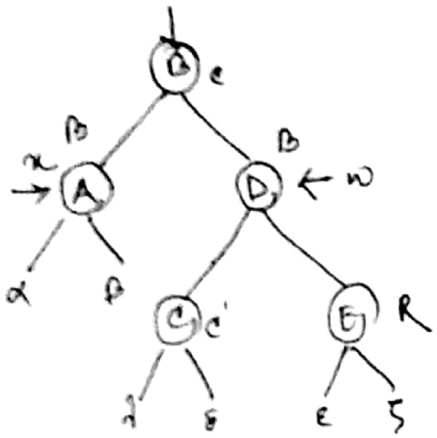


Rules for Case-3

1. $w \rightarrow$ ~~right~~ left $\rightarrow color = Black$
2. $w \rightarrow color = Red$
3. Right rotate (T, w)
4. Now Move $w \leftarrow x \rightarrow p \rightarrow$ Right



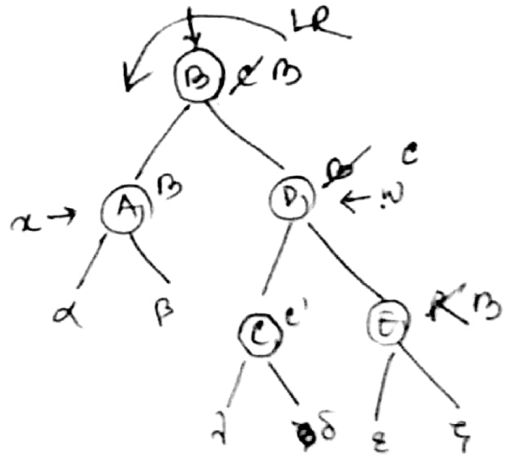
Case 4 when 'w' (sibling of 'x') is black,
'w' left child is black or Red (i.e unknown color c')
and 'w' right child is red



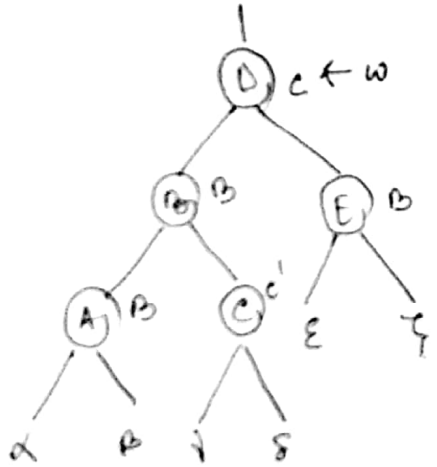
Rules for Case 4

1. $w \rightarrow color = x \rightarrow p \rightarrow color$
2. $x \rightarrow p \rightarrow color = Black$
3. $w \rightarrow right \rightarrow color = Black$
4. Left rotate $(T, x \rightarrow p)$
5. $x \leftarrow T \rightarrow root$.

After Case 4 St-1



After Case 4 St-2



here $x = T \rightarrow root$

The algorithm for removing violations by calling RB-Delete-fixup() is given below,

```
1.  while x ≠ T→root and x→color = Black
2.      do if x = x→p→left.
3.          then w = x→p→right.   // Sibling of x
4.              if w→color = Red          // case 1
5.                  then w→color = Black
6.                      x→p→color = Red
7.                      Left-Rotate (T, x→p)
8.                      w = x→p→right.
9.              if w→left→color = Black and w→right→color = Black
                                                          //case 2
10.                 then w→color = Red
11.                     x = x→p
12.             else if w→right→color = Black  //case-3
13.                 then w→left→color = Black
14.                     w→color = Red
15.                     Right-Rotate (T, w)
16.                     w = x→p→right
17.             w→color = x→p→color
18.             x→p→color = Black
19.             x→right→color = Black
20.             Left-Rotate (T, x→p)
21.             x = T→root.
22      else (same as then clause with "right" and "left" exchanged)
Analysis  x→color = Black
```

Action on case 1

Action on case 2

Action on case-3

Action on case-4

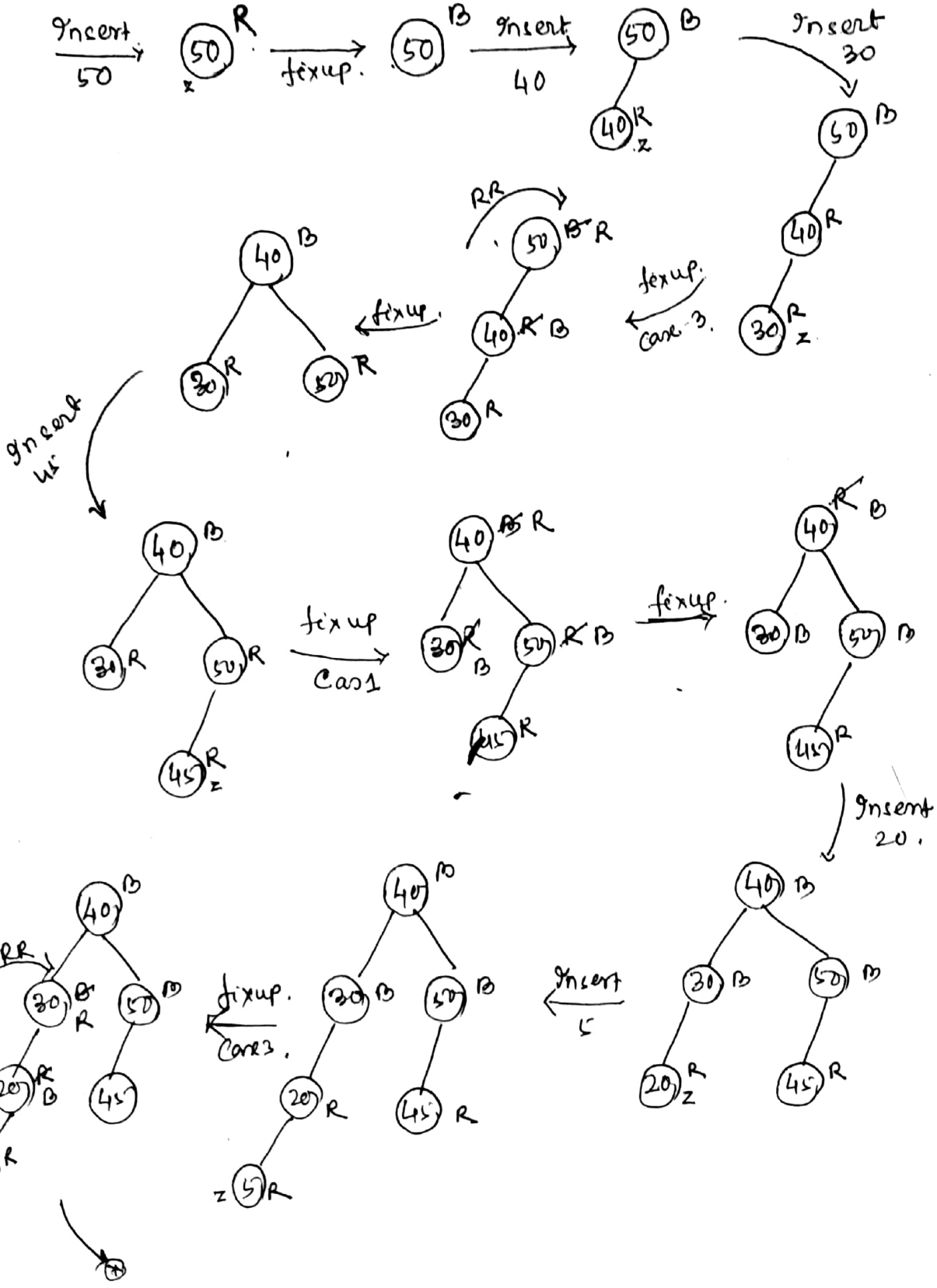
Since the height of a red-black tree of n-nodes is $O(\lg n)$, the total cost of the of RB-Delete without RB-Delete-fixup takes $O(\lg n)$ time. where are all the cases of RB-Delete fixup takes $O(\lg n)$ time. So the overall time for RB-Delete is therefore also $O(\lg n)$.

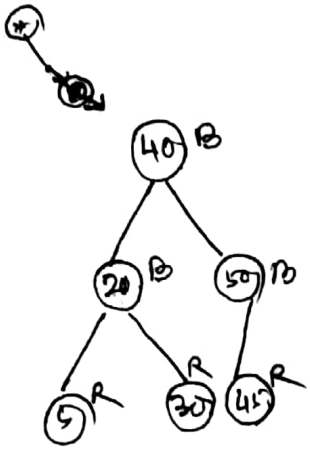
Questions on Red-Black Tree.

Q1. Show the following key first insert in an initially empty Red-Black tree.

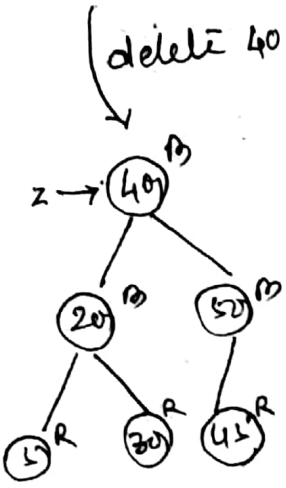
⟨ 50, 40, 30, 45, 20, 5 ⟩

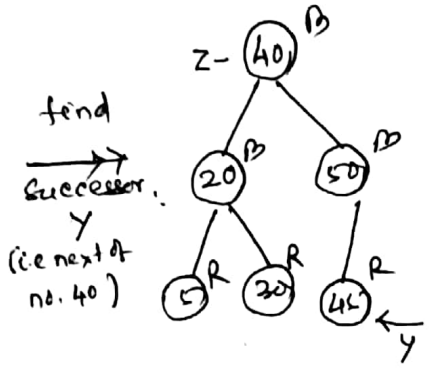
and then perform delete 40, 20, 30, 45 & 5

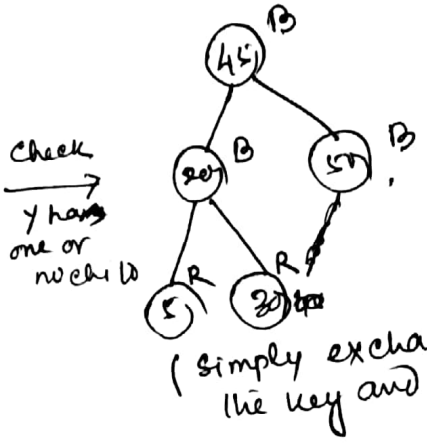


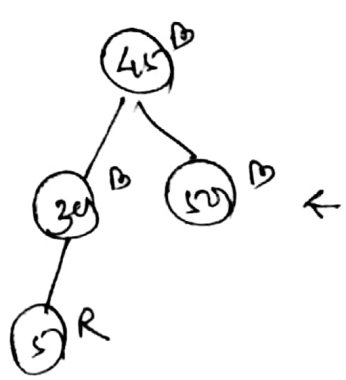


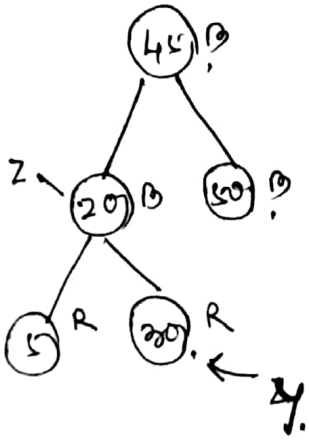
now start deletion operation sequentially on following data:
40, 20, 30, 45 & 5



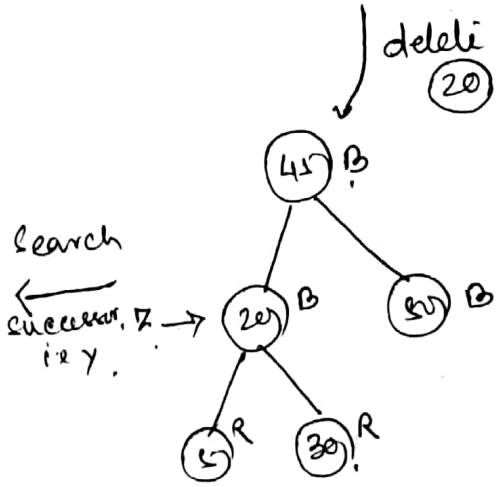


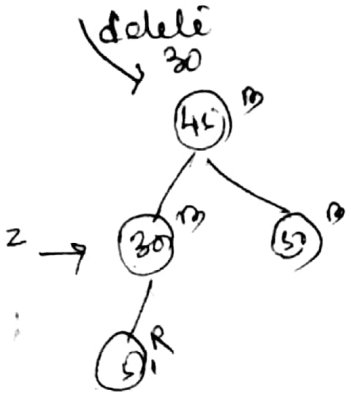


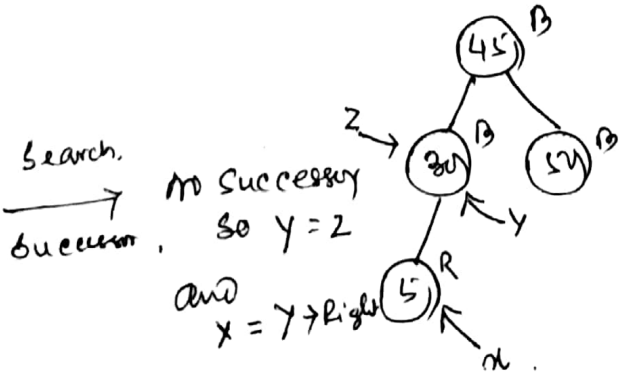


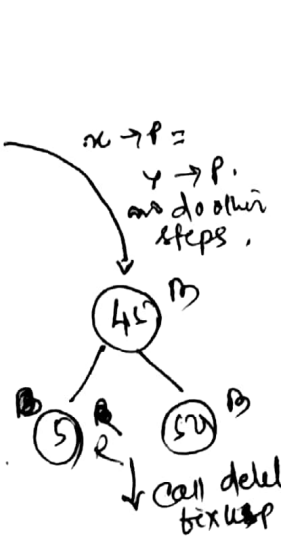


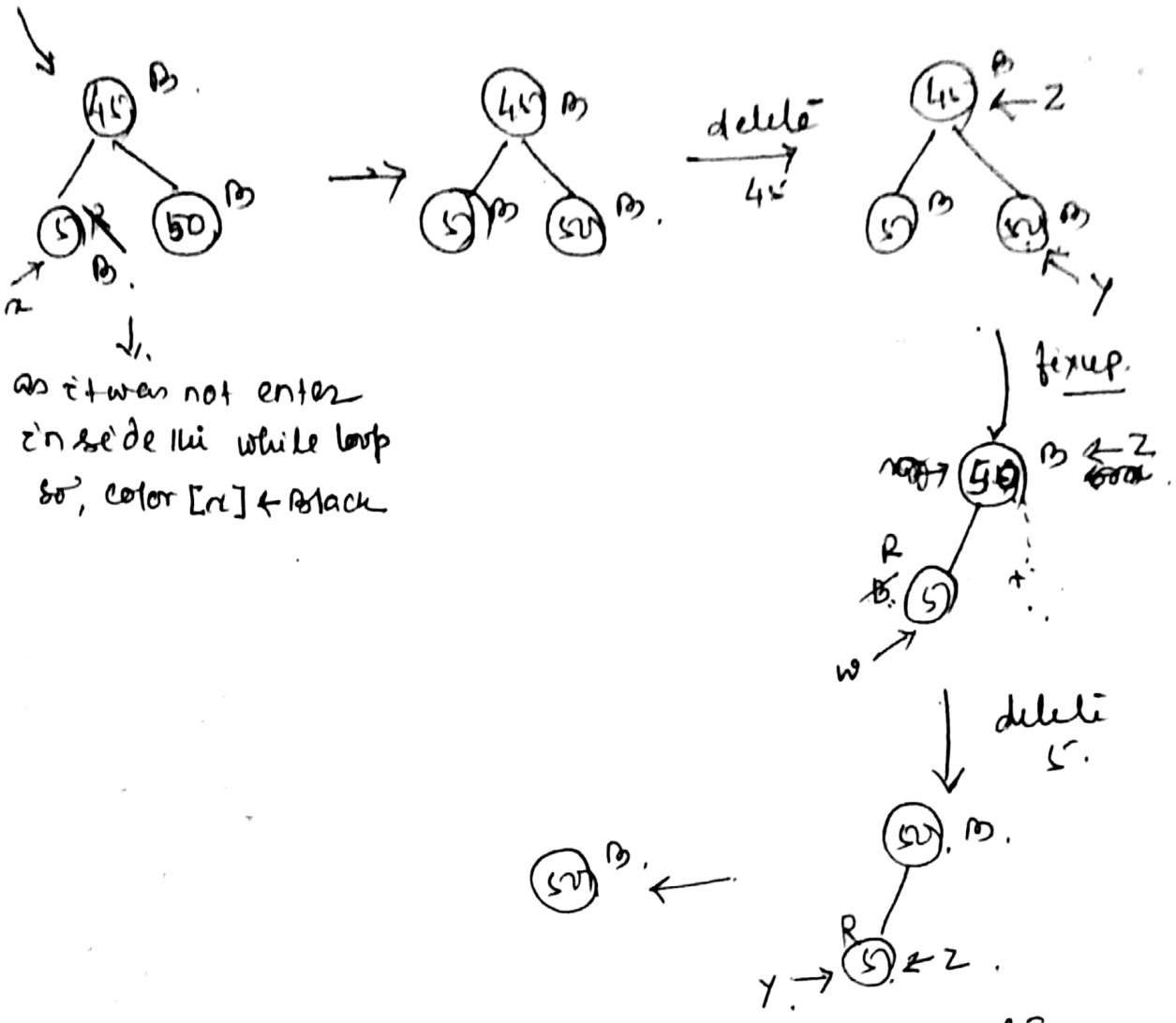
(simply exchange the key and delete.











as it was not enter inside the while loop so, color[x] ← Black

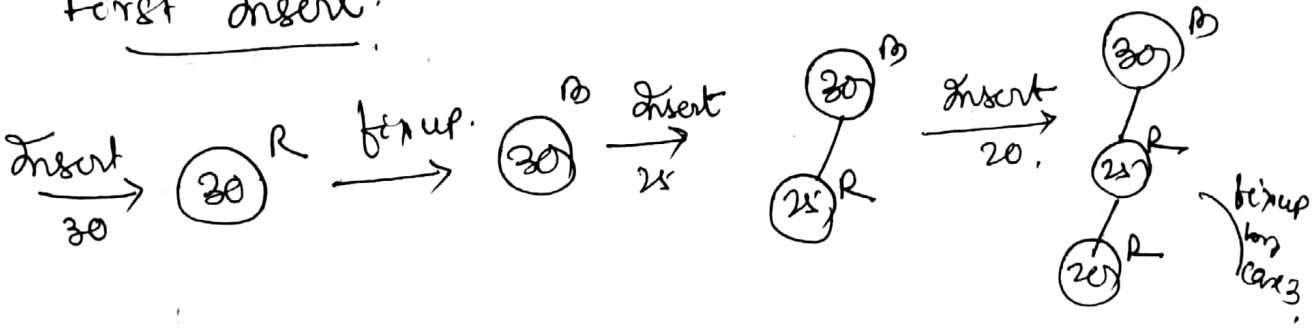
Q2. Insert the following element in ~~a~~ red black tree ~~and then~~

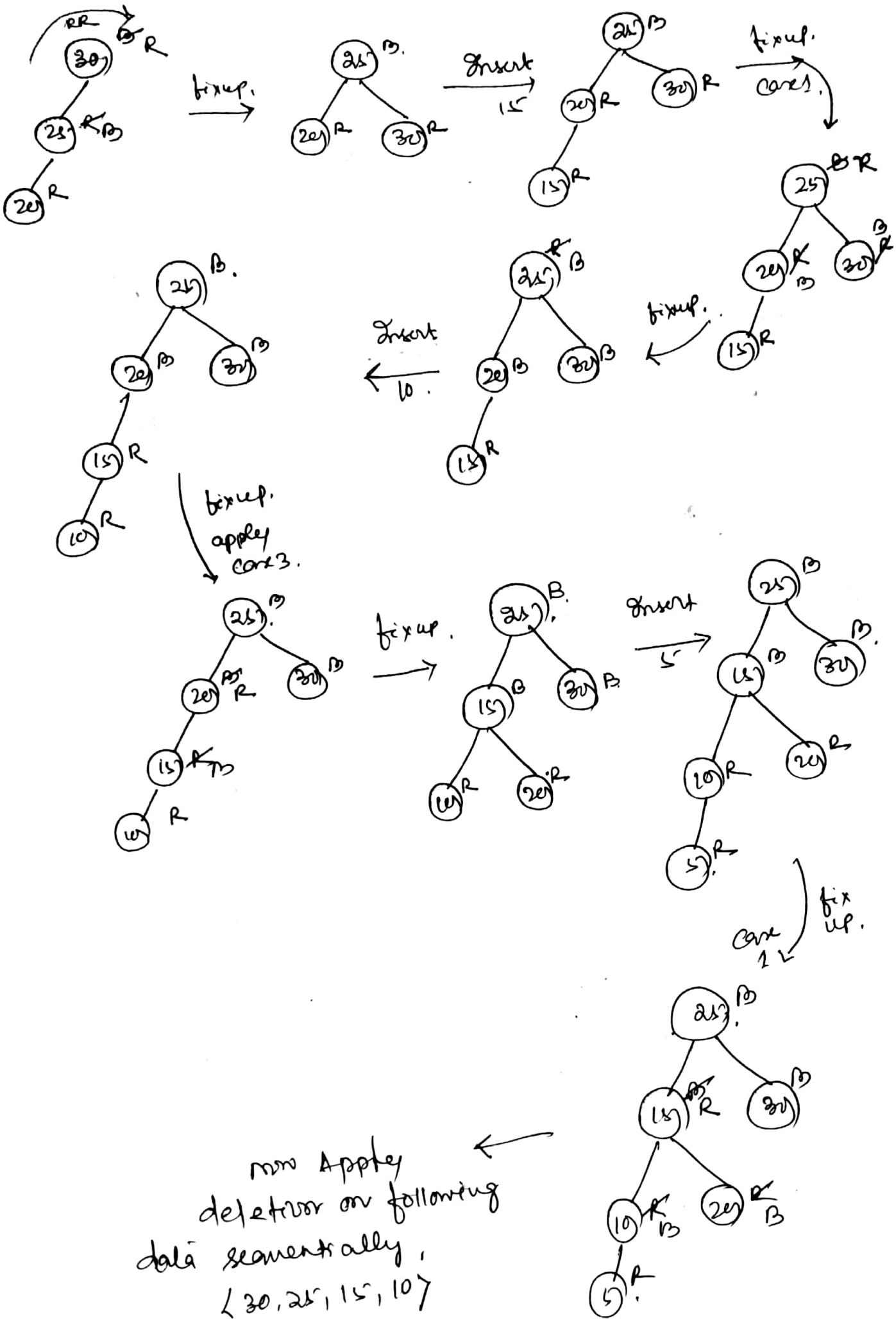
⟨30, 25, 20, 15, 10, 5⟩

and then delete the following element sequentially.

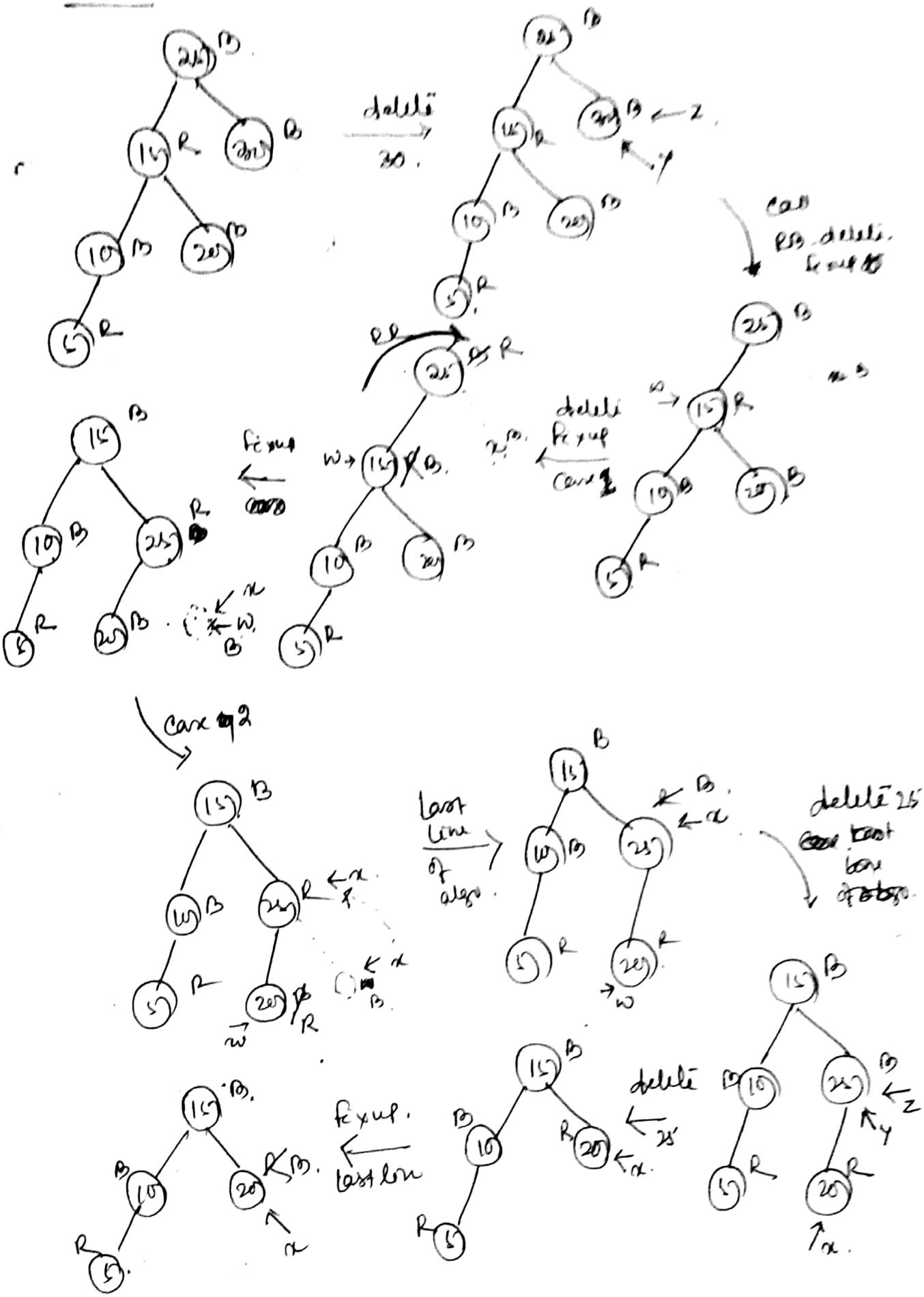
⟨30, 25, 15, 10⟩.

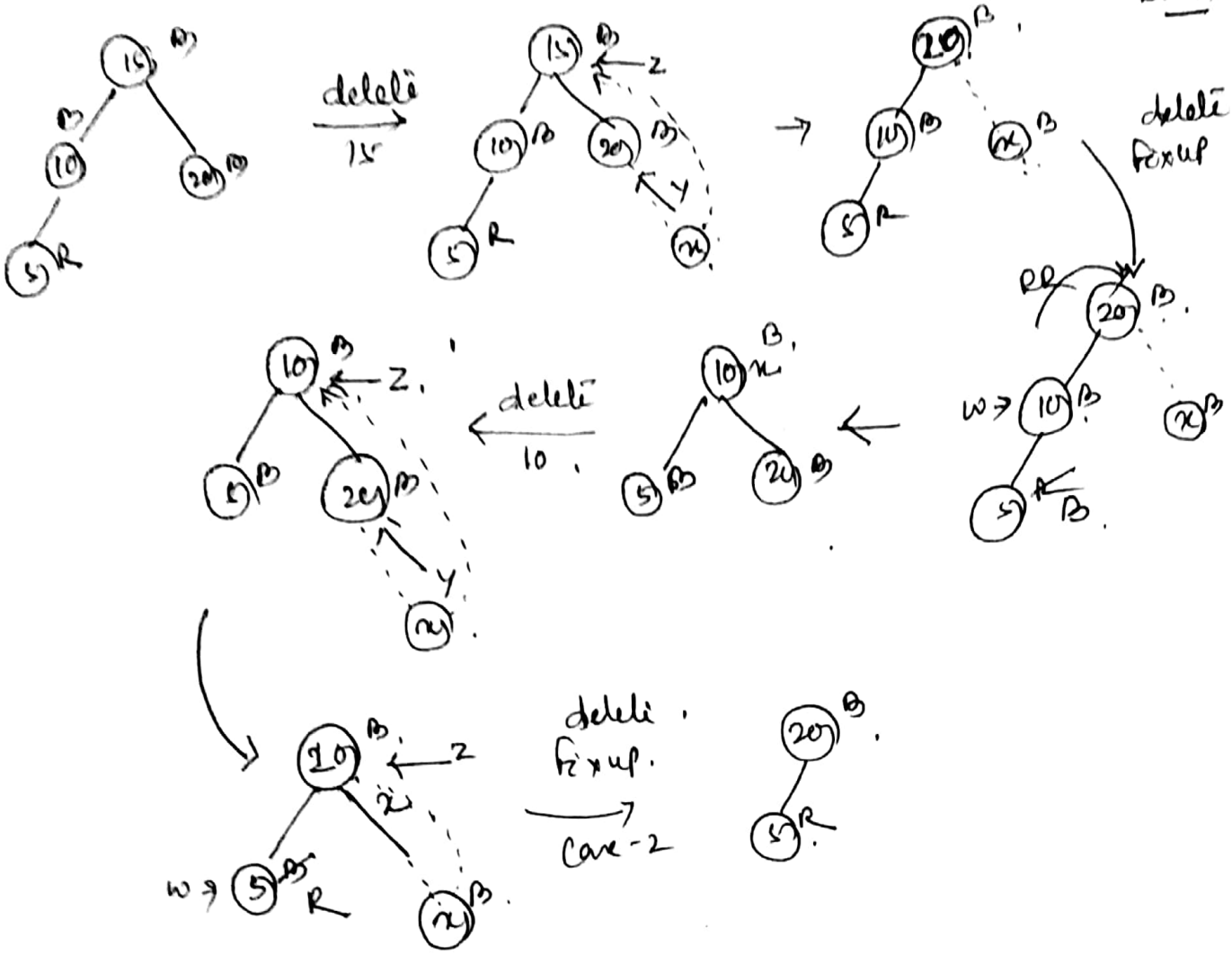
First Insert.





deletion





Lemma

A red-black tree with n internal nodes has height at most $2 \lg(n+1)$

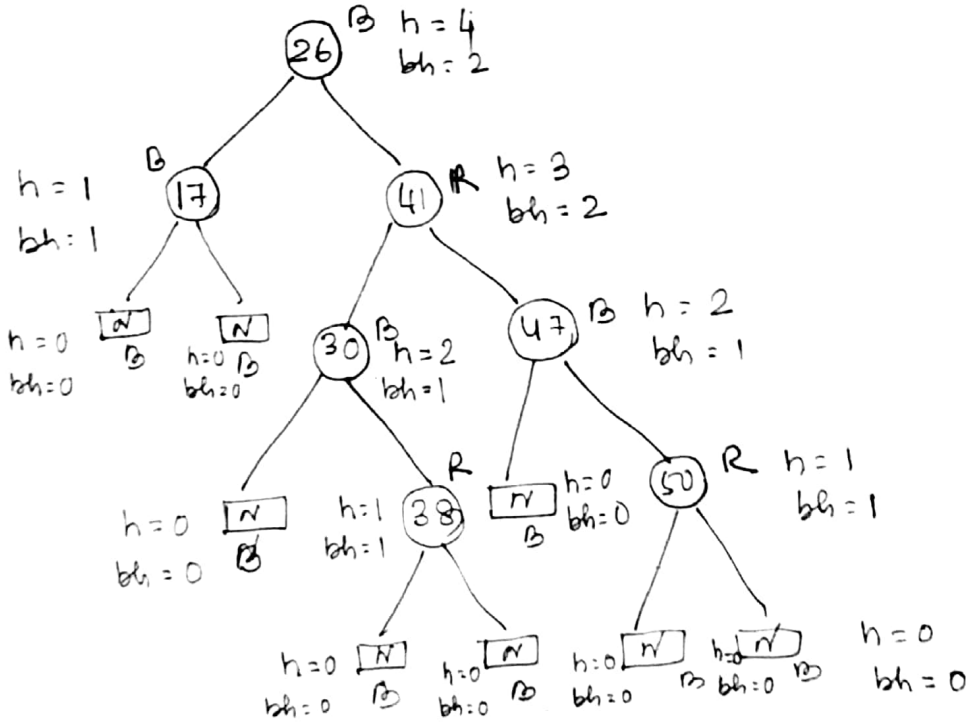
Proof:

Consider any node x and the subtree rooted at x. We will first show that "the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes."

We prove this by induction on the height of x.

→ If the height of x is 0, then x must be a leaf node.

Let us take an example:



(Note: For convenience, we add NIL nodes and refer to them as the leaves of the tree)

Claim-1

Now we first prove that "the subtree rooted at any node x contains at least $2^{bh(x)}-1$ internal nodes (i.e the nodes who bearing keys only)

And we prove it by induction hypothesis.

Basis If $h(x) = 0$.

Hence x is a leaf node.

So $bh(x) = 0$. and

Internal node $(x) = 2^{bh(x)} - 1$

$= 2^0 - 1 = 1 - 1 = 0$.

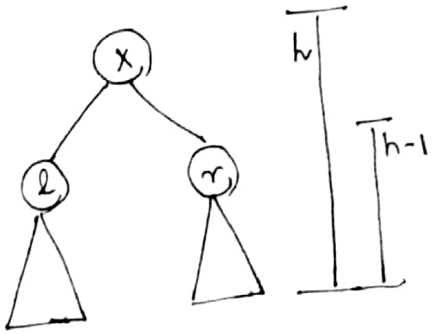
Inductive hypothesis

Assume that it is true for

$h(x) = h-1$

Inductive step:

$h[x] = h$.



Internal node (x) =

Internal node (l) + Internal node (r) +1

by using Inductive hypothesis.

Internal node $(x) \geq \left(2^{bh(l)} - 1\right) + \left(2^{bh(r)} - 1\right) + 1$

For the inductive step; consider a node 'x' that has positive height and is an internal node with two children of left and right. Each children has a black height of either $bh(x)$ or $bh(x)-1$, depending on wheather it's color is red or black respectively.

i.e

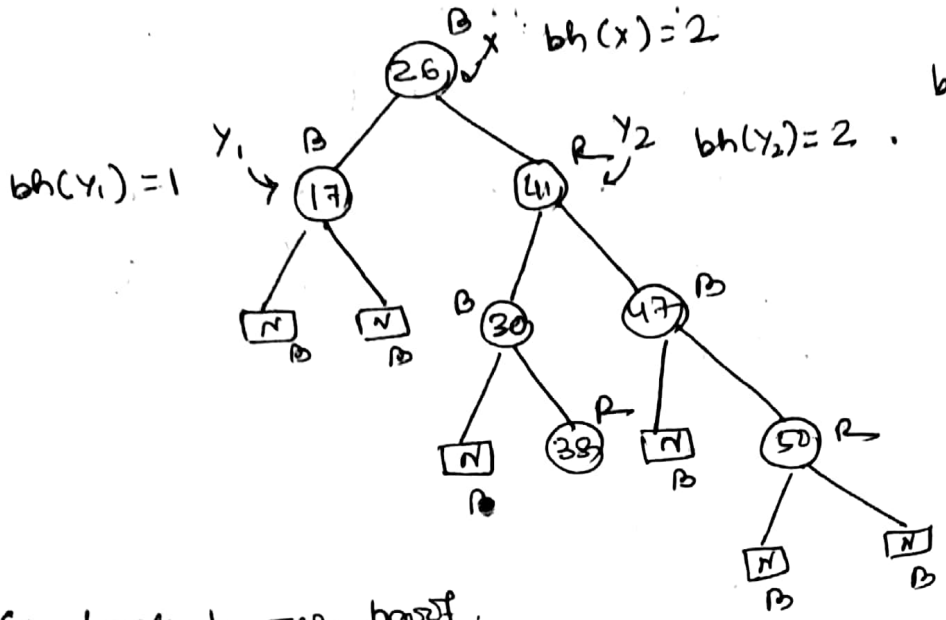
Let $bh(x) = b$, then any child y of x has

* $bh(y) = b$ (if the child is red)
* $bh(y) = b-1$ (if the child is black).

which means

$bh(y) \geq bh(x) - 1$

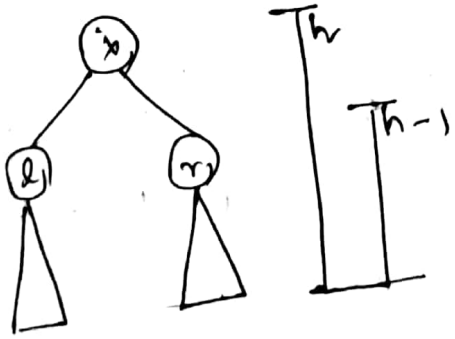
Let's take an example.



So back to our proof.

Internal node $(x) \geq \left(2^{bh(l)} - 1\right) + \left(2^{bh(r)} - 1\right) + 1$

Assume that both l child trees and r-child tree available at $h-1$ height. as shown in below fig.



So Internal node $(x) \geq \left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1$

$= 2\left(2^{bh(x)-1} - 1\right) + 1$

$$= 2 \cdot 2^{bh(x)} \cdot 2^{-1} + 1 - 2$$

$$= x \cdot 2^{bh(x)} \cdot \frac{1}{x} + 1 - 2$$

$$= 2^{bh(x)} - 1$$

Claim-2

To complete the proof of the lemma, prove that: Any node x with height $h(x)$ has $bh(x) \geq h(x)/2$

Let h be the height of the tree,

According to the property 4 (i.e if a node is red, then both its children is black), at least half of the nodes on any simple path from the root to leaf node, not including root, must be black. i.e the black height of the root must be $h/2$. Hence $bh(x) = h/2$

So,

Claim = 1	Claim-2
no. of internal node = n	$h(x) = h =$ where x = root.
$n \geq 2^{bh(x)} - 1$	$\geq 2^{h(x)/2} - 1$ or $2^{h/2} - 1$

So $n \geq 2^{h/2} - 1$

Solve for h.

$$n + 1 \geq 2^{h/2}$$

Take log both side.

$$\log(n+1) \geq \log 2^{h/2}$$

$$\Rightarrow \log(n+1) \geq h/2 \log 2$$

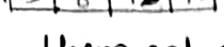
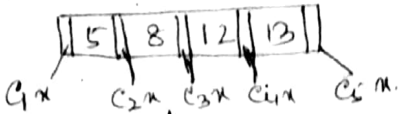
$$\Rightarrow \log(n+1) \geq h/2$$

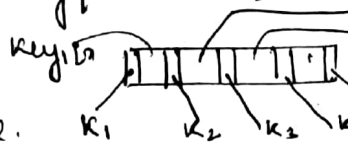
$$\Rightarrow h \leq 2 \log(n+1)$$

Proved

B-Tree

A B-tree T is a rooted tree having the following properties:-

1. Every node x has the following fields:-
   a) $n[x]$, the number of keys currently stored in node x. e.g.  $n[x] = 4$
   b) The $n[x]$ keys themselves, stored in nondecreasing order, so that $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$
   c) $leaf[x]$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined. (or empty or NULL)
   

3. The $key_i[x]$ separate the ranges of keys stored in each subtree: if K_i is any key stored in the subtree with root $c_i[x]$, then
   $$K_1 \leq key_1[x] \leq K_2 \leq key_2[x] \leq \cdots \leq key_{n[x]}[x] \leq K_{n[x]+1}$$
   

4. All leaf nodes have the same depth, which is the tree's height h.
5.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:

a) Every node other than the root must have at least $t-1$ keys.

Every internal node other than the root thus has at least t children.

If the tree is nonempty, the root must have at least one key.

b) Every node can contain at most $2t-1$ keys. Therefore, an internal node can have at most $2t$ children.

(we can say that a node is full if it contains exactly $2t-1$ keys.

The simplest B-tree occurs when $t=2$. Here Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree.

In practice, however much larger values of t are typically used.

For

Q. Draw a B-tree of minimum degree $t = 3$ of given sequence. and assume that B-tree was initially empty.

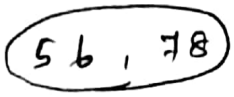
78, 56, 52, 95, 88, 105, 15, 35, 22, 47, 43, 50, 19, 31, 40, 41, 59.

So Minimum key $= t - 1 = 3 - 1 = 2$

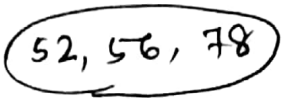
Maximum key $= 2t - 1 = 6 - 1 = 5$.

Insert 78

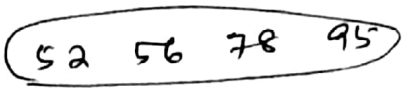
Insert 56



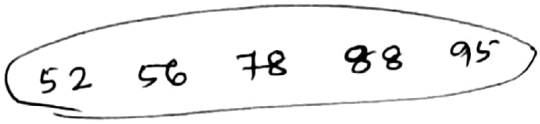
Insert 52



Insert 95



Insert 88



now we cannot insert a new key into an existing leaf node as the maximum key size limit is achivid. So we introduce a split function, which split a full node y around its median key & Y into two nodes having only t-1 keys each.

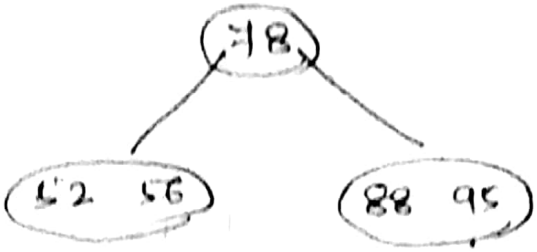
[note the median formula is $\{(n+1) \div 2\}^{th}$, where n is the number of items.]

The median key moves up into y's parent to identify the dividing point between these two new trees. But if y's parent is also full, we must split it before we can insert the new key,

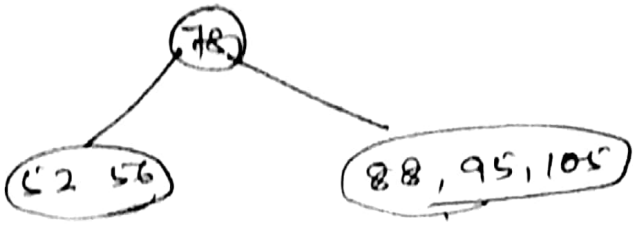
and thus we could end up splitting from nodes all the way up the tree.

now Insert 101

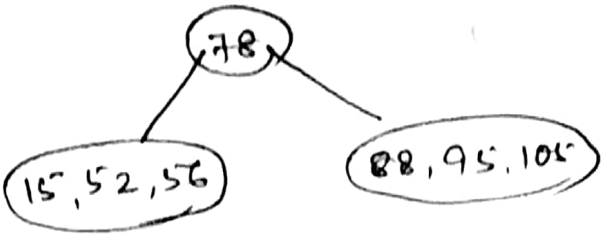
First split.



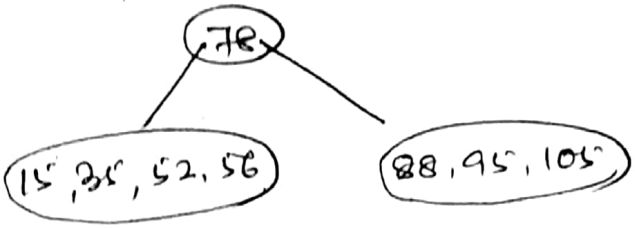
⇓ then insert 101



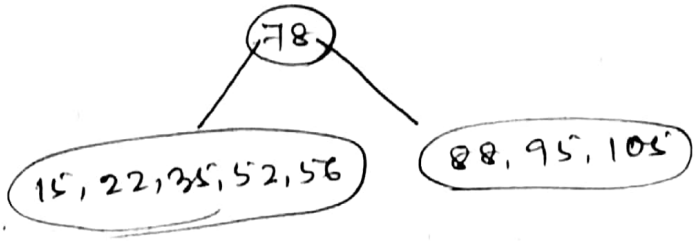
Insert 15



Insert 35

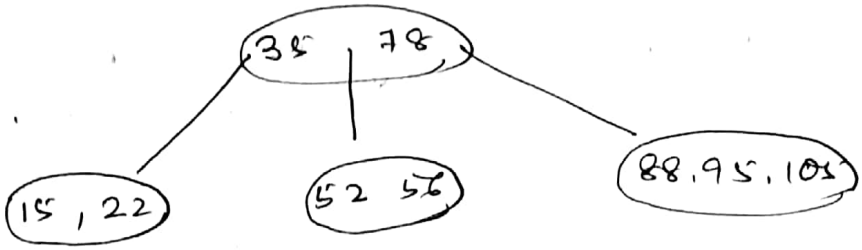


Insert 22

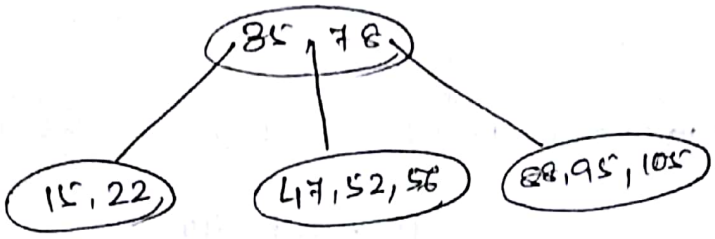


Insert 47

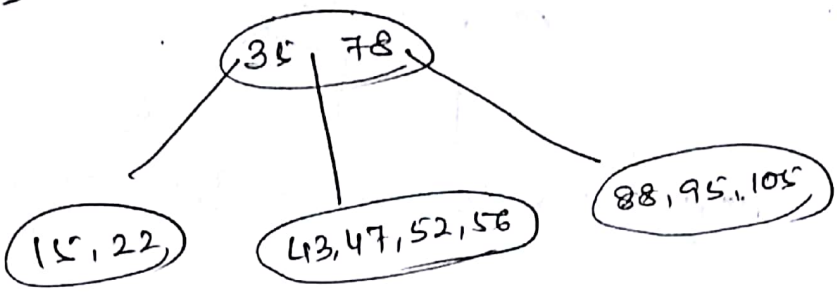
first split.



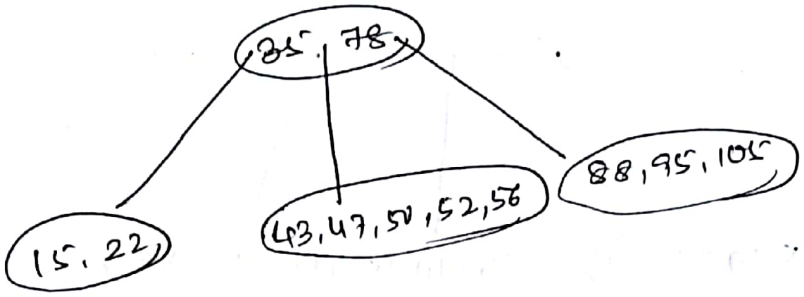
// Insert 47 .



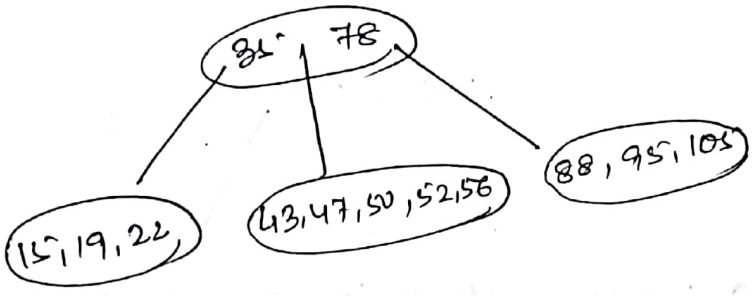
Insert 43



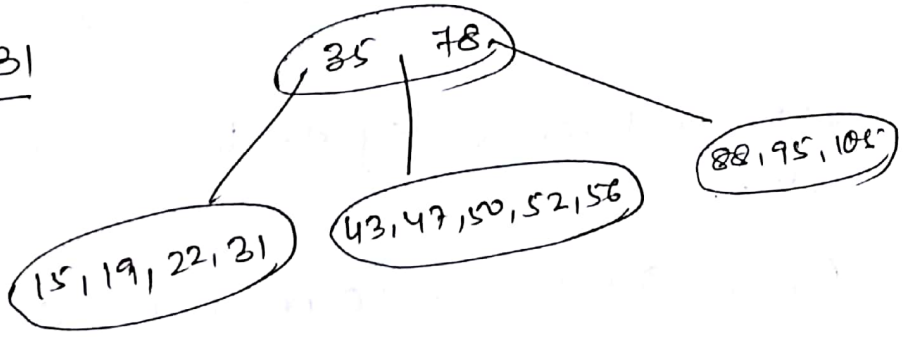
Insert 50



Insert 19

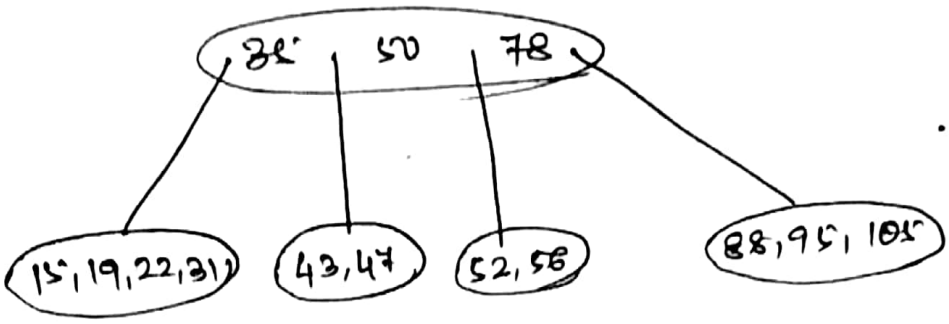


Insert 31

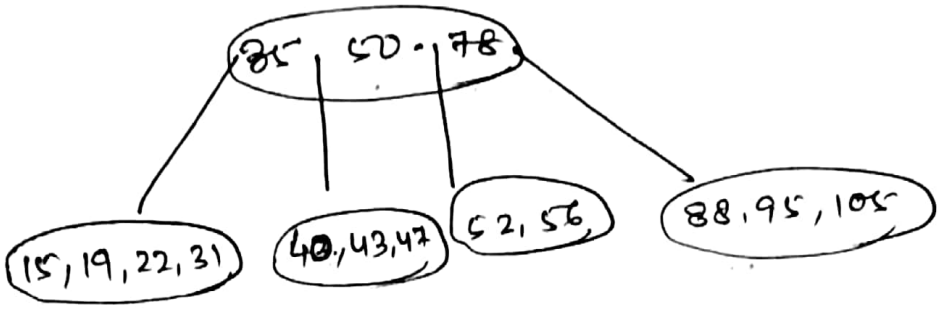


Insert 40

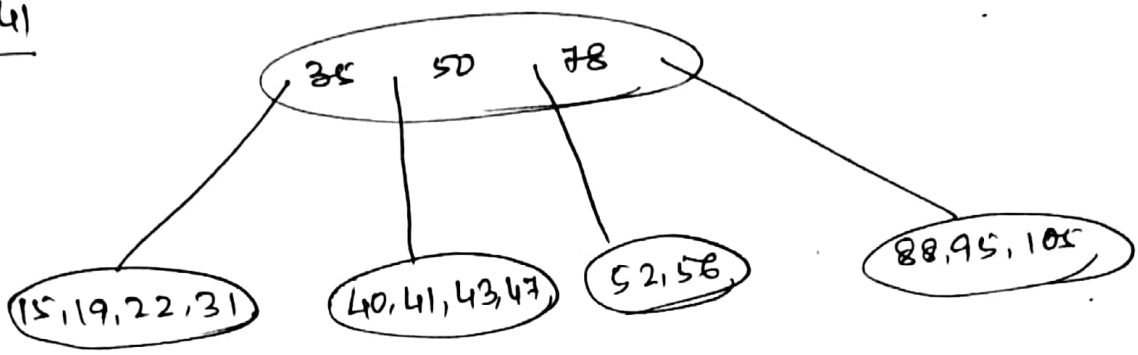
First split and then insert .



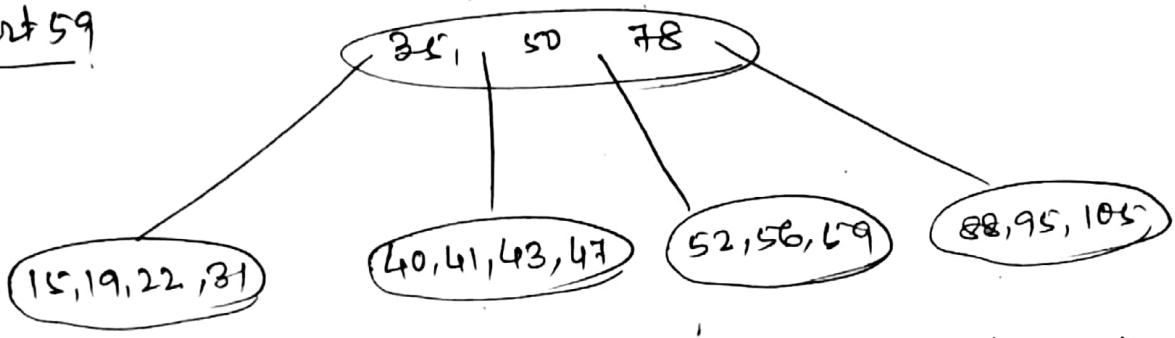
↓ Insert - 40.



Insert 41



Insert 59



Q. Construct B-Tree of degree $t = 3$ on following data set.

E, A, S, Y, Q, U, E, S, T, I, O, N, I, N, G

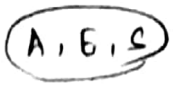
Assume that the B-Tree is initially empty.

order : 3 Max key : $2t-1 = 5$ Min key : $t-1 = 2$

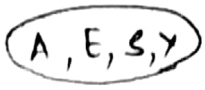
Insert E

Insert A

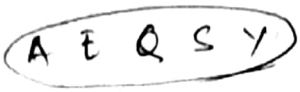
Insert S



Insert Y

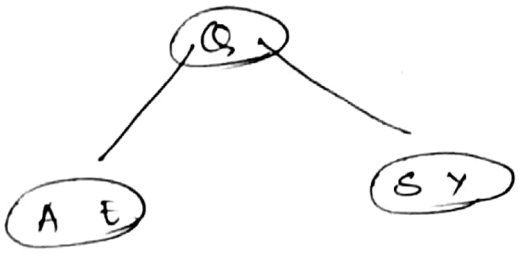


Insert Q

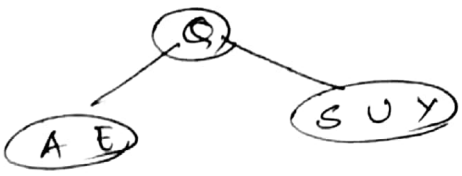


Insert U

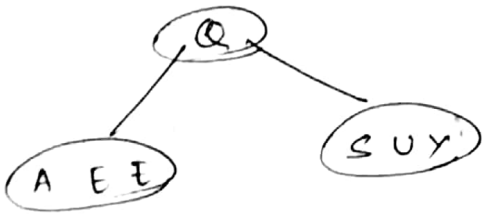
First split and then insert U



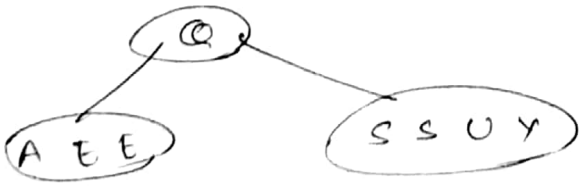
↓ Insert U



Insert E

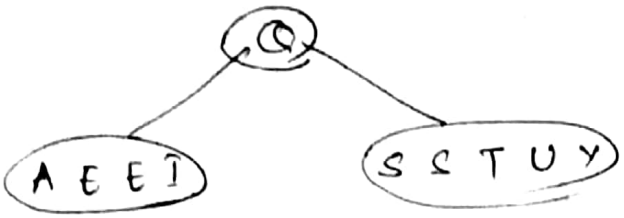


Insert S

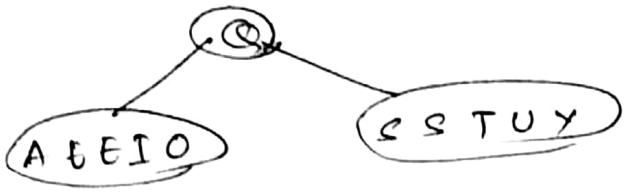


Insert T

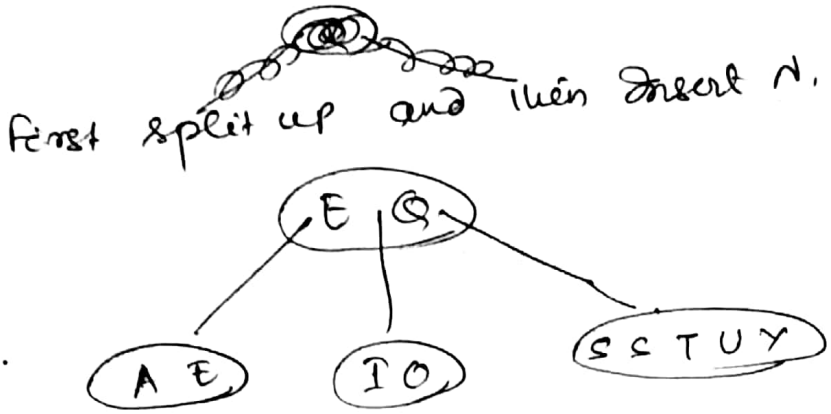
Insert I



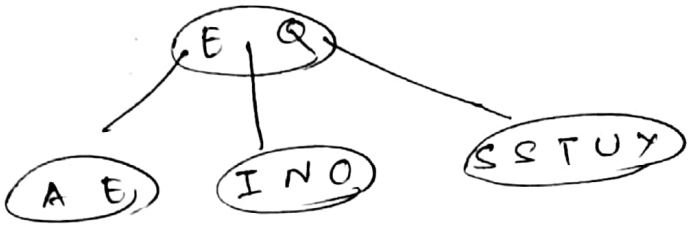
Insert O



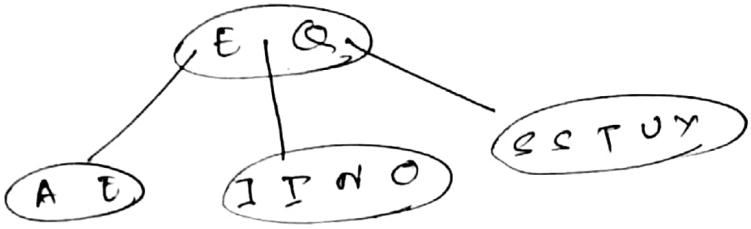
Insert N



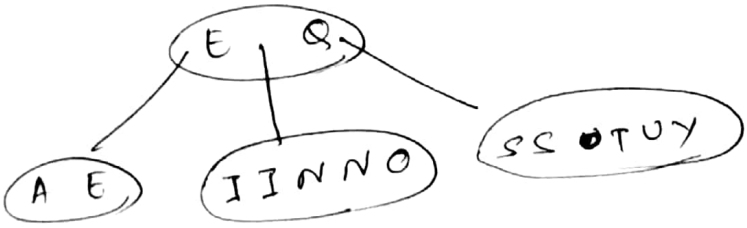
↓ insert N.



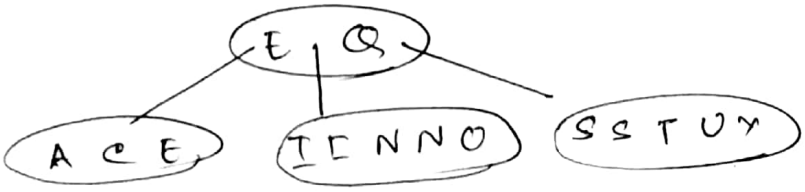
Insert I



Insert N



Insert C



A B-tree can be constructed by order as well as degree.

order (m)	degree (t)
Max key = m - 1	Max key = 2t - 1
Min key = $\lceil \frac{m}{2} \rceil - 1$	Min key = t - 1

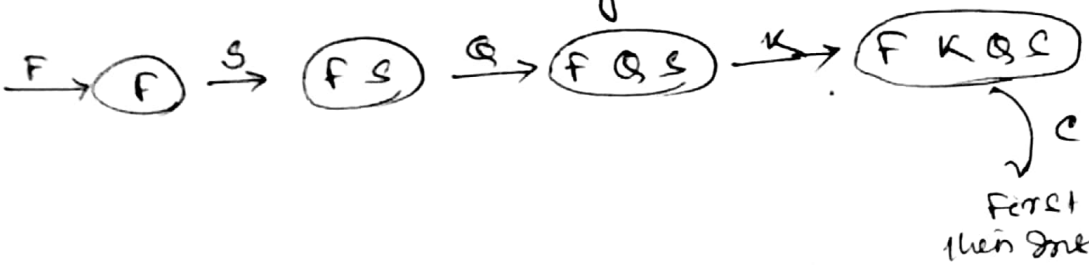
Q Construct a B-Tree of order 5 or following sequence of data. Assume that tree is empty initially.

< F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, Z, D, E. >

Ans:- Order = 5 = M

Hence the Max key = M - 1 = 4

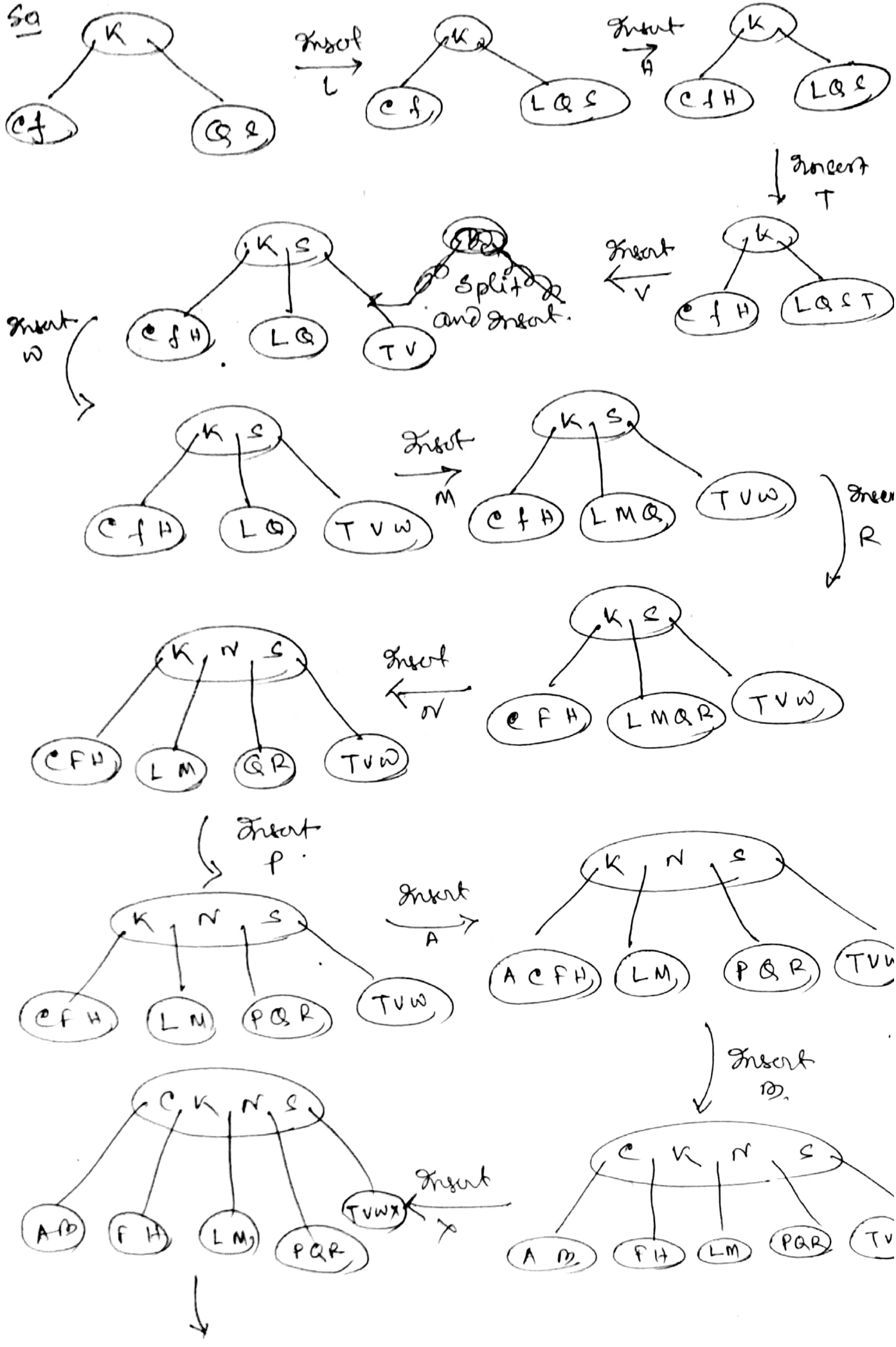
Min key = $\lceil \frac{M}{2} \rceil - 1 = 3 - 1 = 2$

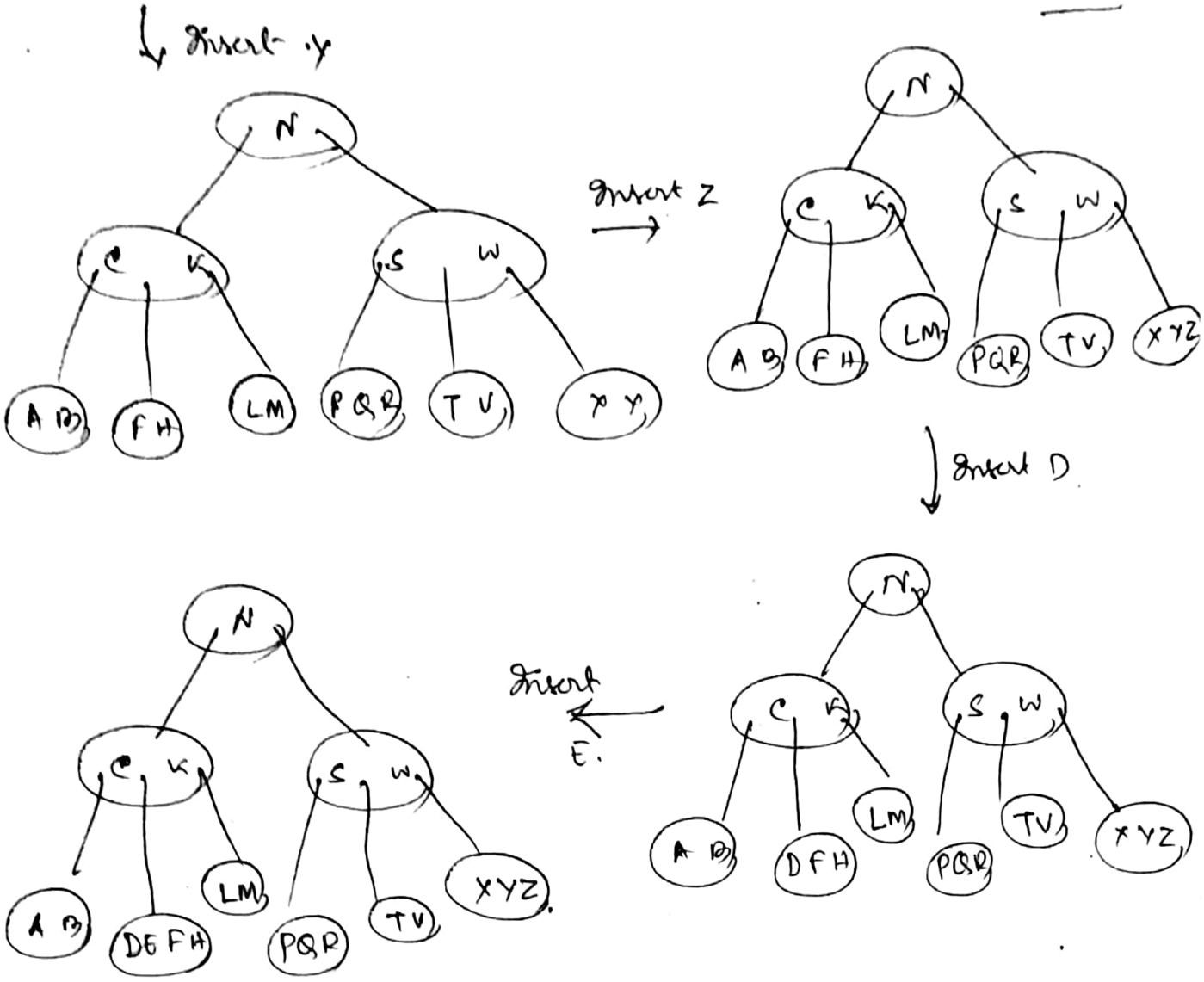


Note:

When the key elements are even and there is a need of split at that time we get two nodes i.e. m and m+1. In this case the split is done for following ways.

1. If the inserted item is < m then split from m.
2. If the inserted item is > m+1 then split from m+1
3. If the inserted item is > m and < m+1 then split on inserted item.





Q. Why we don't allow min degree (t) = 1

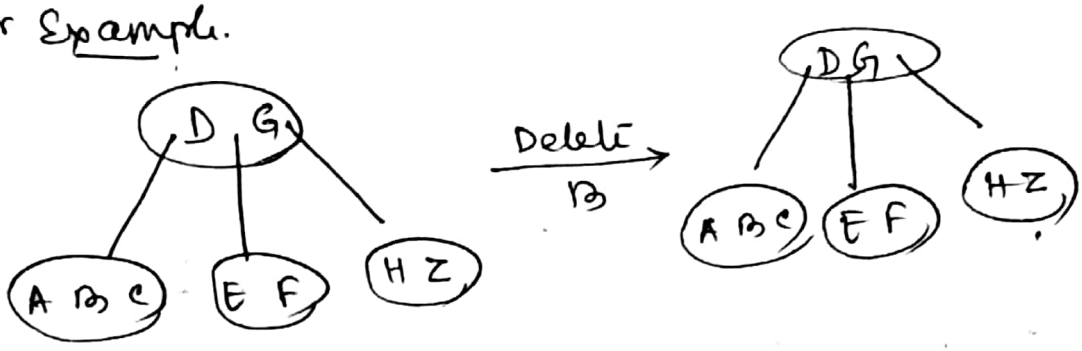
Because if $t = 1$ then min key in any node is 0 (zero) which is not true! So degree must be $t \geq 2$.

Deletion of key from B-Tree.

Case 1 If x is a leaf node & the leaf node have more than (t-1) keys then the key can just be removed without disturbing the tree.

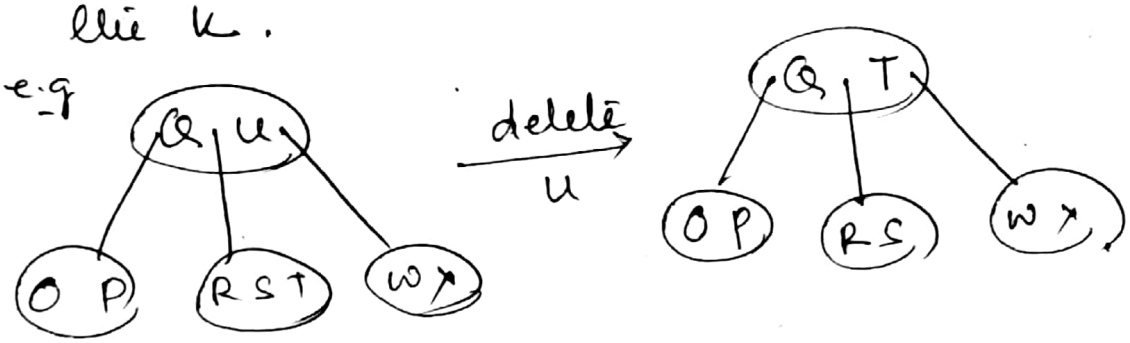
Let the degree (t) = 3.

For Example.



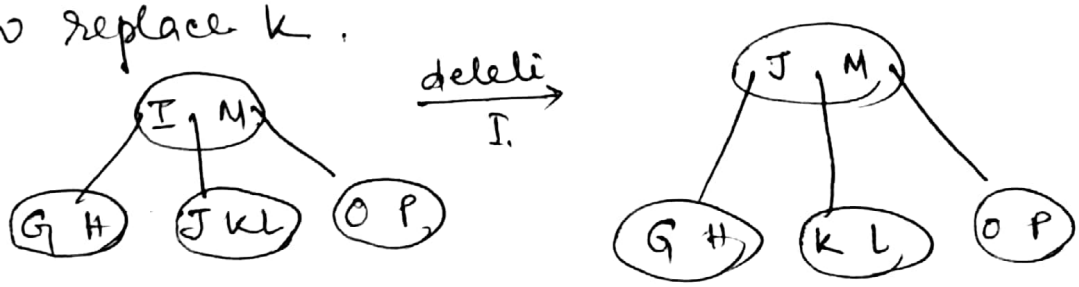
Case 2 a) If x is an internal node. and the key left children have atleast t key, then the largest value can be moved up to replace the k.

e.g



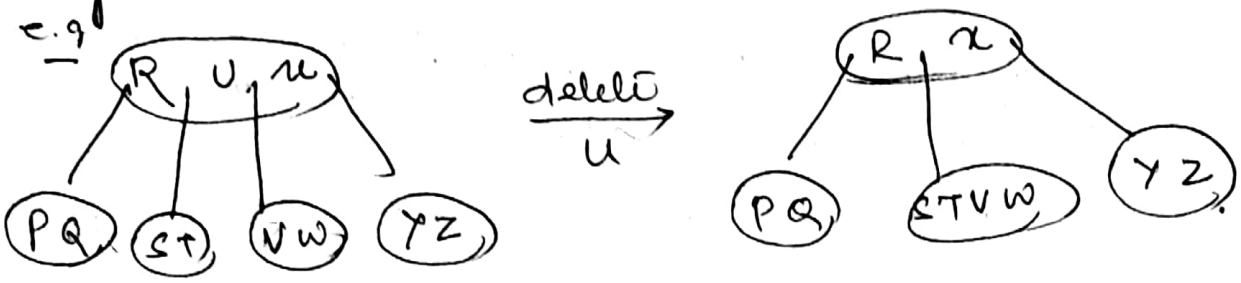
b) If the key right child has atleast t keys then its smallest value can be move up to replace k.

e.g.



c) If neither child has atleast t keys then the two childs of key must be merge into one and key must be removed.

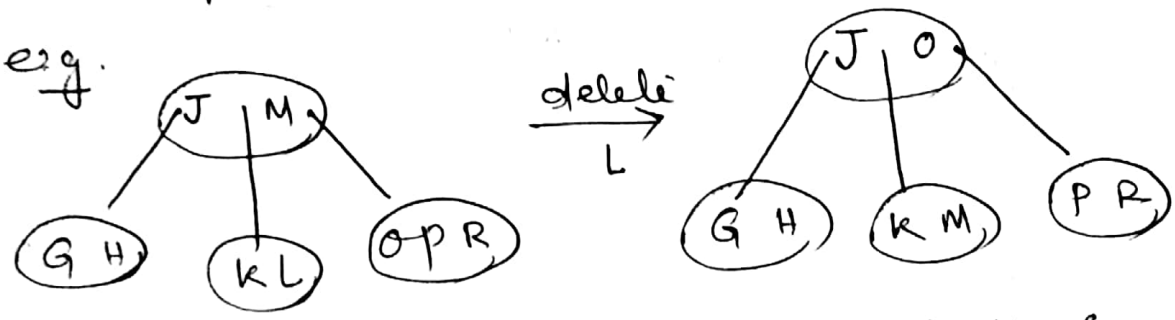
e.g.



Case-3. If x has (t-1) keys

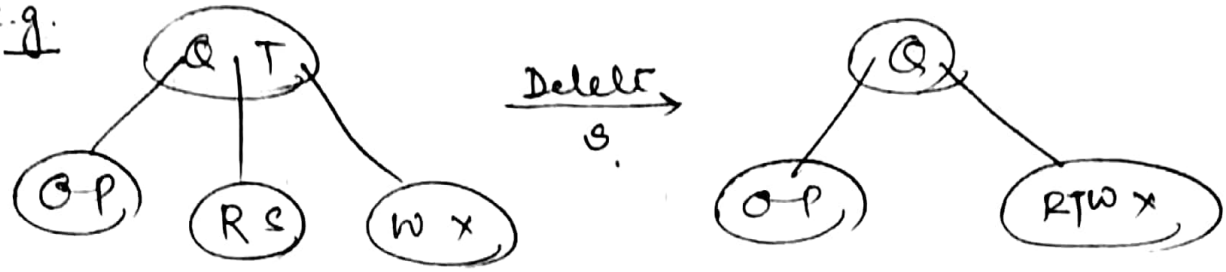
a) If x has a sibling (always right) with atleast t keys move x parent into x and move the appropriate element from the x sibling into open slot into the parent x sibling (for leaf node, then delete the desired value. (for leaf node).

e.g.



b) If x sibling also have (t-1) keys merge x with one of its sibling by bringing down the parent to be the Median value then delete desired value.

e.g:

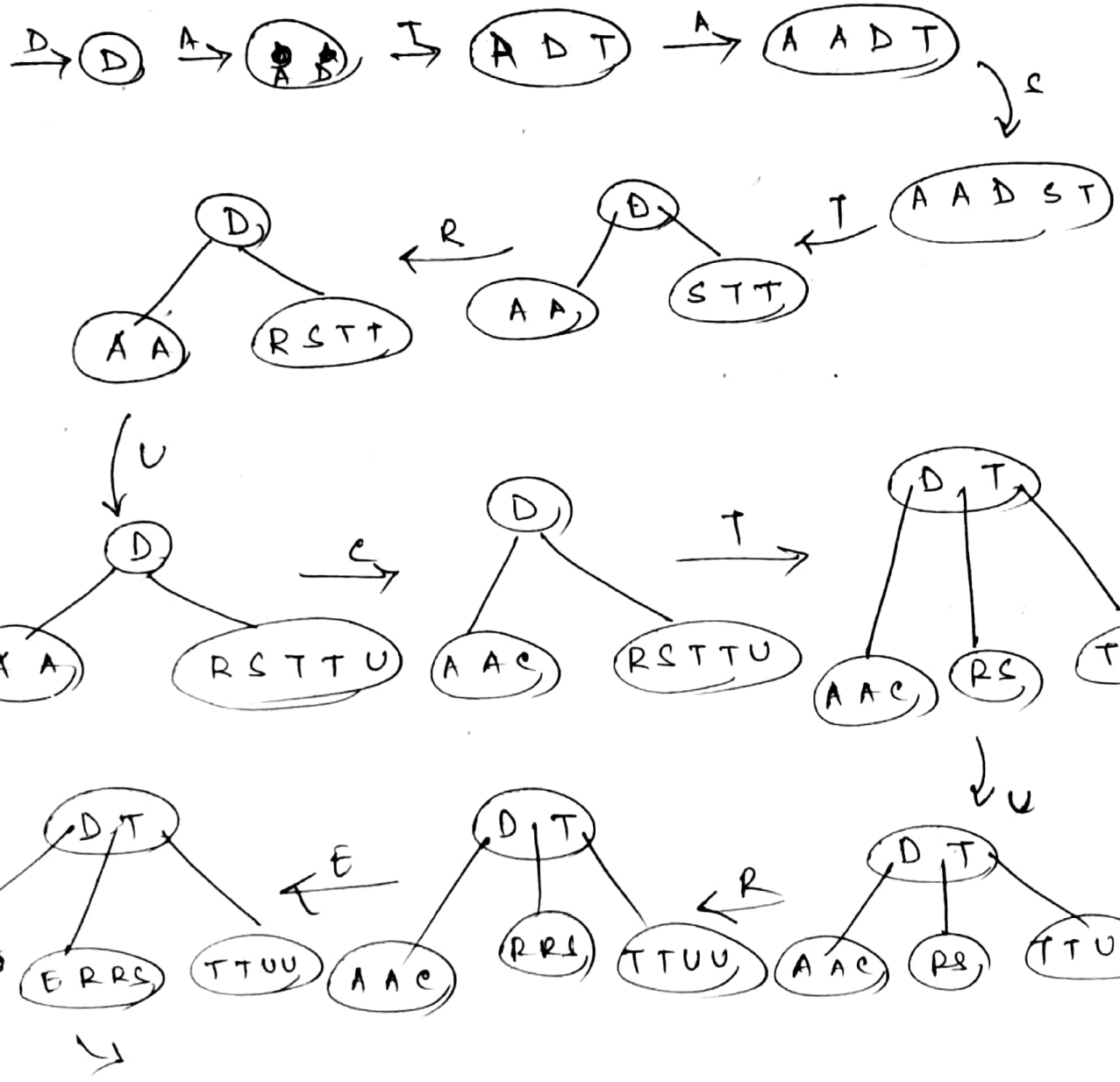


Q. Construct a B-Tree of following data set of degree 3

DATA STRUCTURE USING C

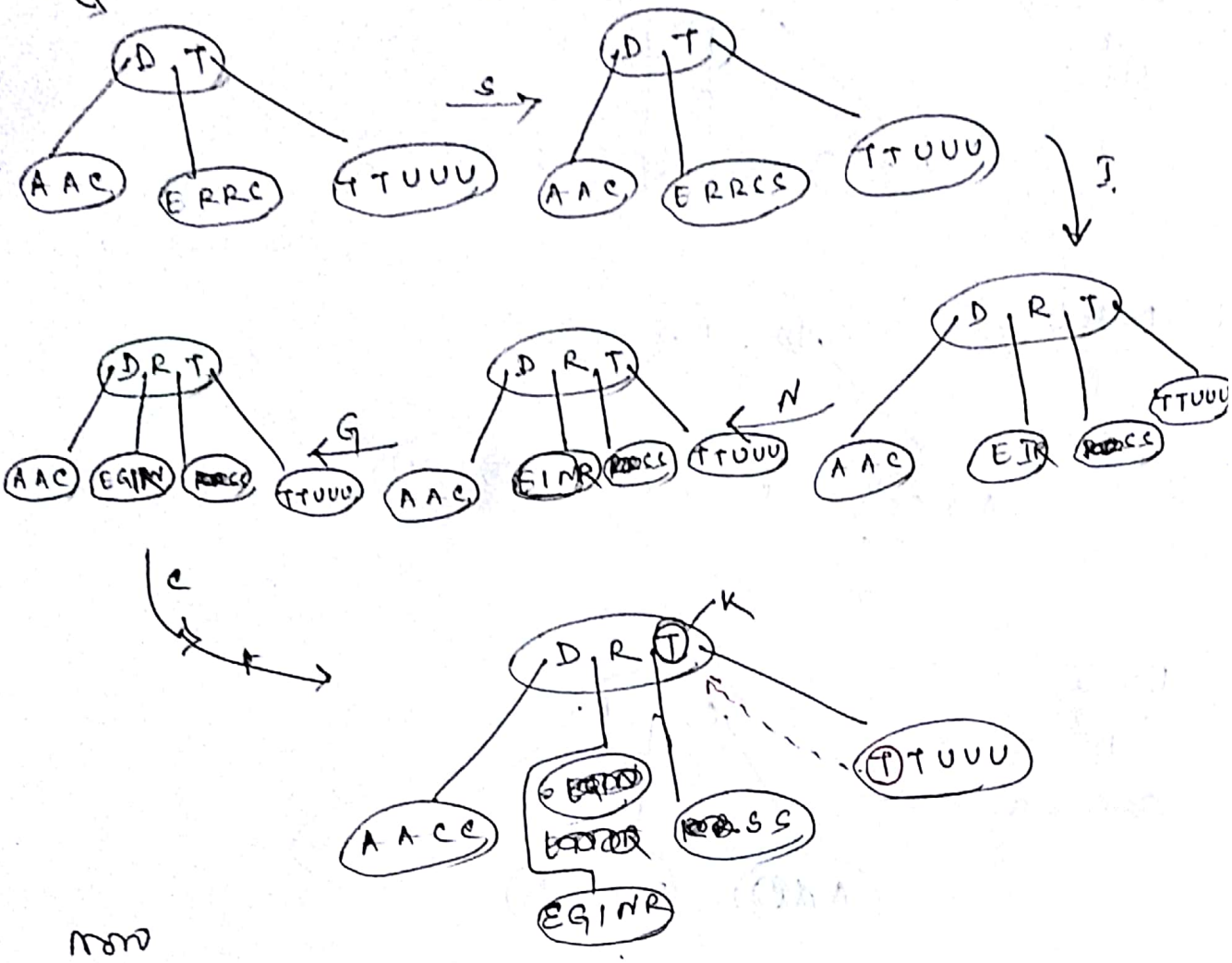
and perform the delete operation with following data sequentially.

First insert tree. Max key = 5, Min key = 2



B-Tree 8.

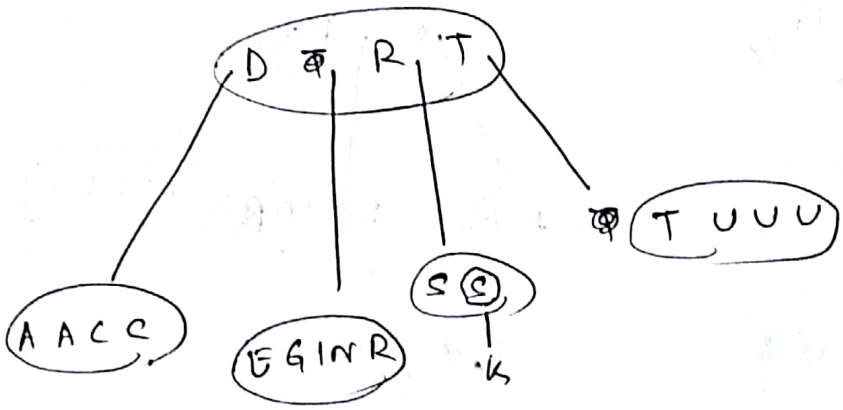
Insert U



Now

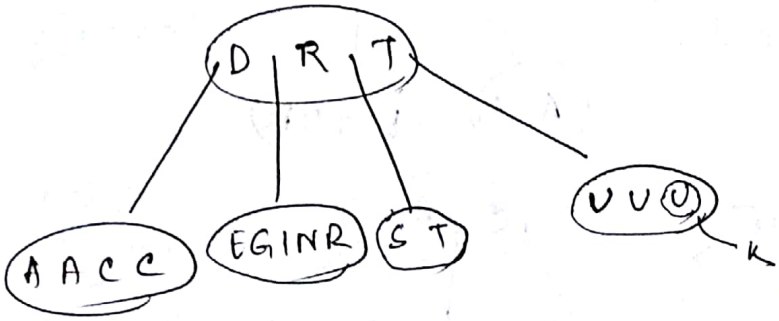
Delet T

Case 2(b)

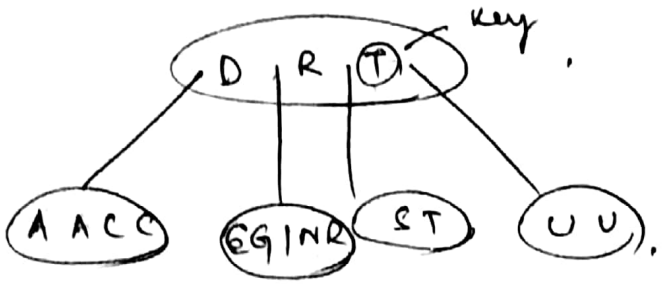


Delete S

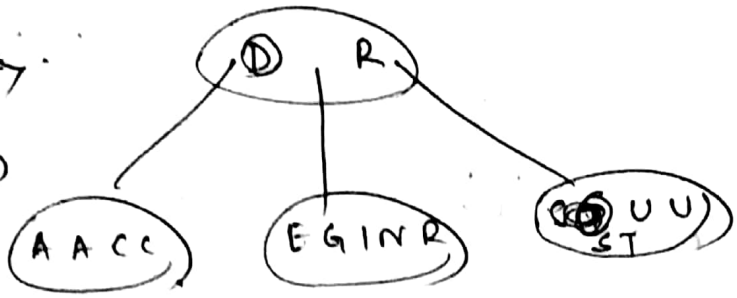
Case 3 @



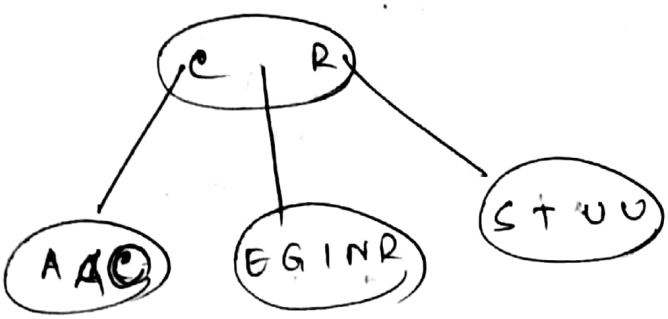
Delete → U
Case 1.



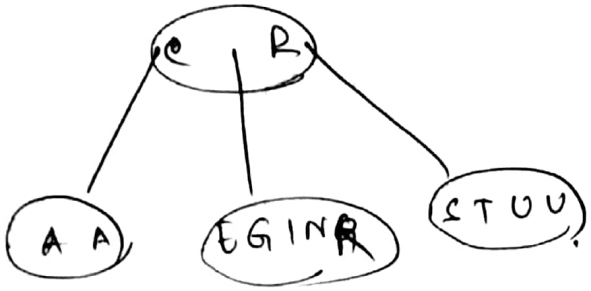
Delete → T
Case 2(c)



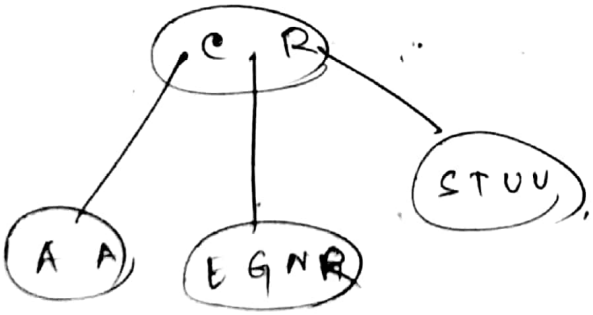
Delete → D
Case 2 a



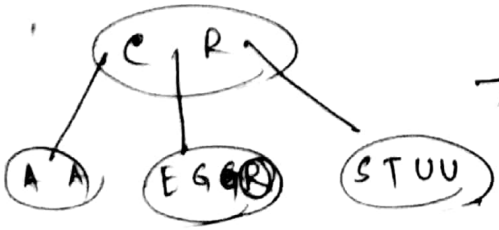
Delete → C
Case 1



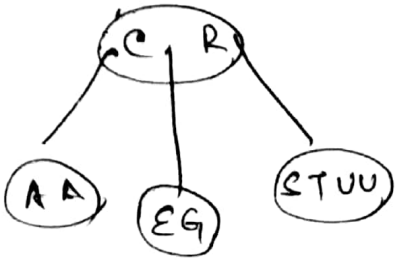
Delete → I
Case 1.



Delete → N
Case - 1.

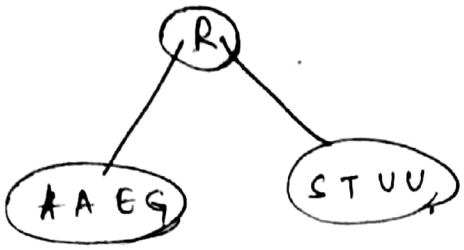


Delete → R
Case 1



• Delete C

Case 2(c)

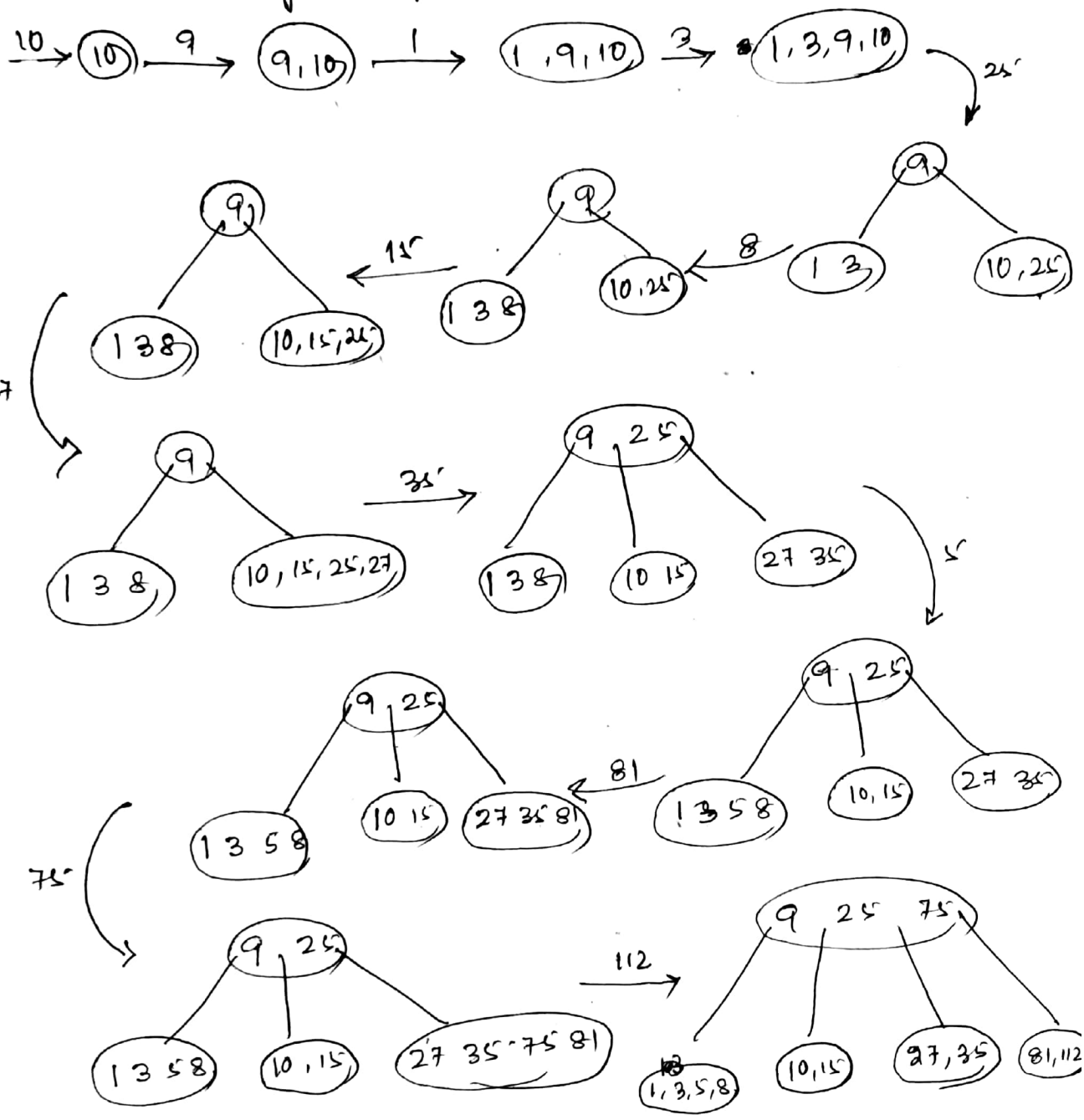


Q. Construct a B-Tree of order 5 & the keys are.

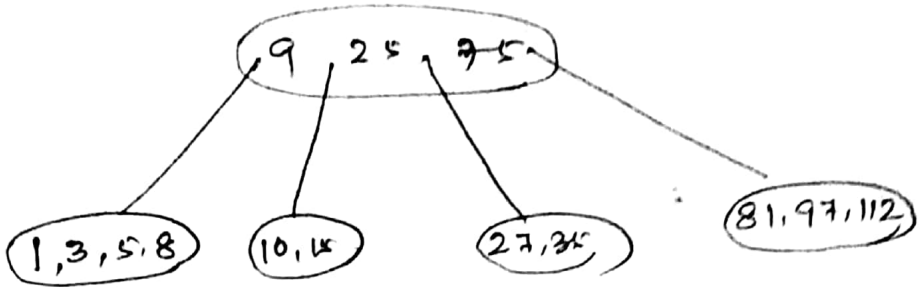
10, 9, 1, 3, 25, 8, 15, 27, 35, 5, 81, 75, 112, 97.

Max key : $M - 1 = 5 - 1 = 4$

Min key $= \lceil M/2 \rceil - 1 = \lceil 5/2 \rceil - 1 = 3 - 1 = 2$.



97 →

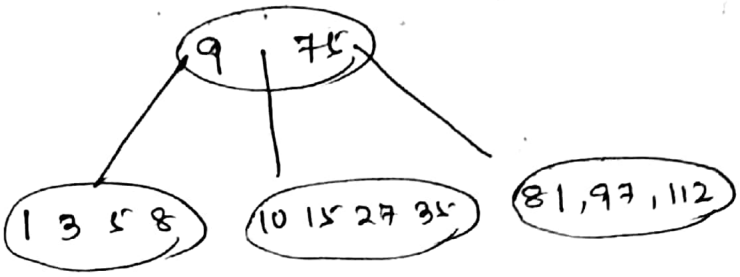


Next perform delete operation on following data set sequentially.

25, 75, 27, 5, 15, 81, 97, 112, 10

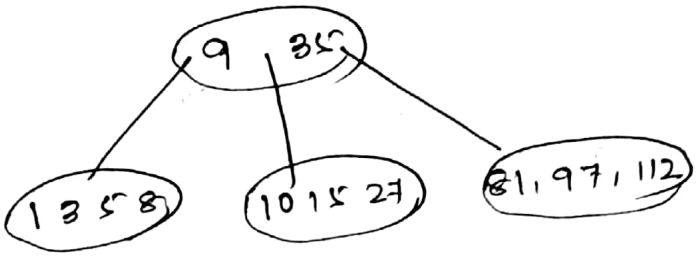
delete 25 →

Case 2(c)



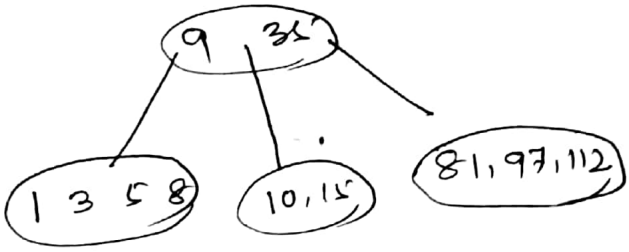
delete 75 →

Case 2(a)



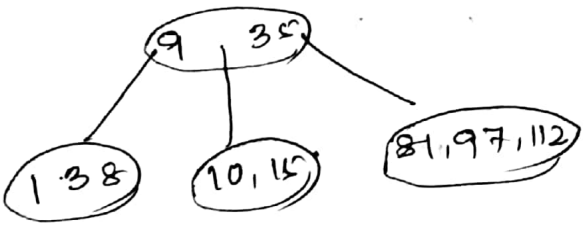
delete 27 →

Case 1



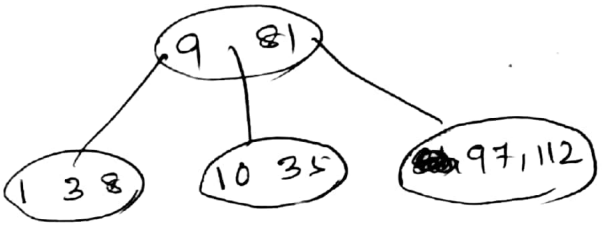
delete 5 →

Case 1



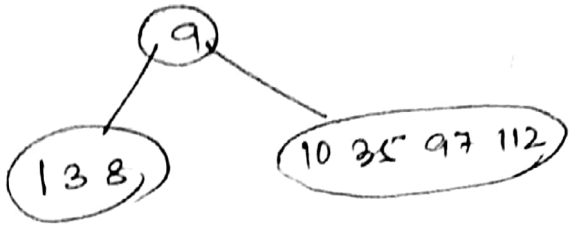
delete 15 →

Case 3a



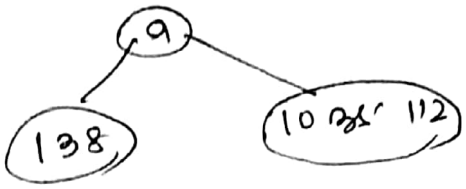
delete 81

Case 2(b).



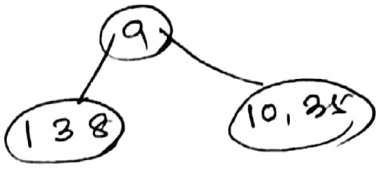
delete 97

Case 1



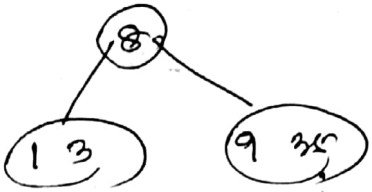
delete 112

Case 1



delete 10

Case 2(a).



Let us analyze the height of B-tree in worst case.

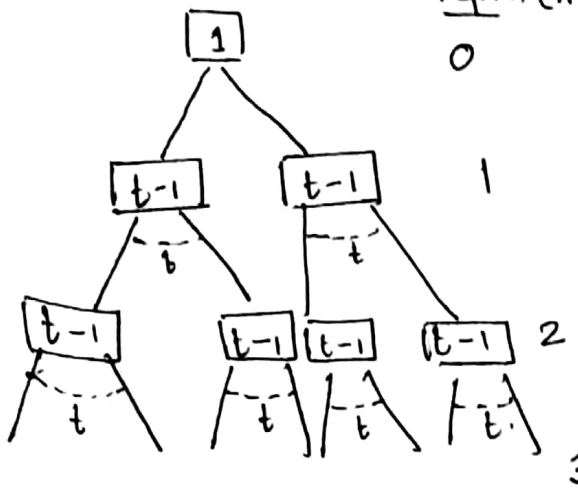
Theorem.

If $n \geq 1$, then for any n-key B-Tree T of height h and minimum degree $t \geq 2$, $h \leq \log_t \frac{n+1}{2}$

Proof.

* The root of B-Tree contains at least one key and all other nodes contain at least t-1 keys Thus, T, whose height is h, has at least 2 nodes at depth 1, at least 2t nodes at depth 2, at least $2t^2$ node at depth 3, and so on, until it has at least $2t^{h-1}$ nodes.

On the next page the figure illustrates for $h=3$. Thus, the number of keys satisfies the inequality.



depth (h)	no. of node	no. of elements
0	1	1
1	$2 = 2t^0$	$2(t-1)$
2	$4 = 2t^1$	$2t(t-1)$
3	$8 = 2t^2$	$2t^2(t-1)$

$$n \geq 1 + 2(t-1) + 2t(t-1) + 2t^2(t-1) + \cdots + 2t^{h-1}(t-1)$$

$$n \geq 1 + 2(t-1)\left[\underbrace{1 + t + t^2 + t^3 + \cdots + t^{h-1}}_{\text{G.P series.}}\right]$$

$$n \geq 1 + 2(t-1)\,\frac{t^{h-1+1} - 1}{(t-1)}$$

$$n \geq 1 + 2(t^h - 1)$$

$$n \geq 1 + 2t^h - 2$$

$$n \geq 2t^h - 1$$

$$n+1 \geq 2t^h.$$

$$\Rightarrow t^h = \frac{n+1}{2}$$

Apply log both side.

$$\log t^h = \log \frac{n+1}{2}$$

$$\Rightarrow h = \log_t \left(\frac{n+1}{2}\right)$$

Proved

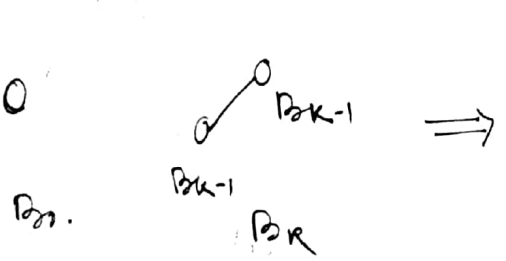
Binomial Heap.

A Binomial heap is a collection of Binomial Trees.

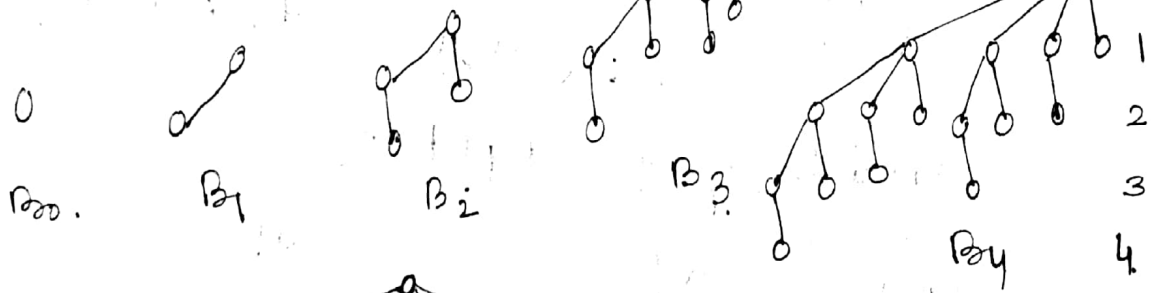
What is a Binomial Tree?

The Binomial tree (B_k) is an ordered tree defined recursively. The binomial tree B_0 consists of a single node. The Binomial tree B_k consists of two binomial trees B_{k-1} that are linked together so that the root of one is the left most child of the root of the other. An example of Binomial trees B_0 through B_4 given below.

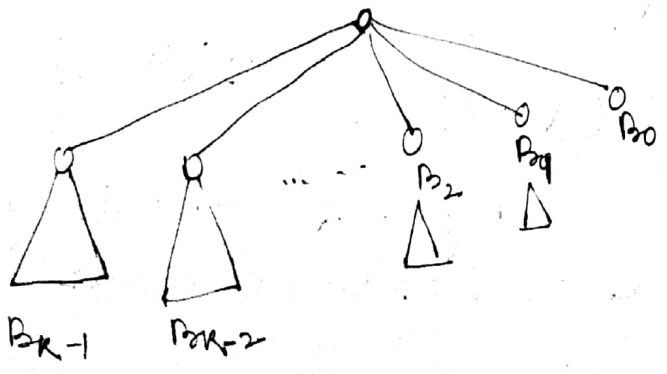
a)



b)



c)



Properties of Binomial trees:

1. there are 2^k nodes. if the degree is k. i.e $n = 2^k$
2. the height of the tree is k.
3. there are exactly $\binom{k}{i}$ nodes at depth i for $i \leftarrow 0, 1, 2, \ldots k$ and.

For Example:

Assume we check for B_4 where $k = 4$ and depth $= 2$ so $i = 2$

$$\text{so } \binom{k}{i} = \frac{k!}{i!(k-i)!} = \frac{4!}{2! \times (4-2)!}$$

$$= \frac{4 \times 3 \times 2 \times 1}{2 \times 1 \times 2 \times 1} = 6 \text{ nodes at depth - 2.}$$

Similarly. for depth 4

$$\binom{k}{i} = \frac{k!}{i!(k-i)!} = \frac{4!}{4! \times (4-4)!} = 1$$

Similarly for depht 3.

$$\binom{k}{i} = \frac{k!}{i!(k-i)!} = \frac{4!}{3!(4-3)!} = \frac{4 \times 3!}{3! \times 1!} = 4$$

4. The root has degree k, which is greater than that of any other node, moreover, if the

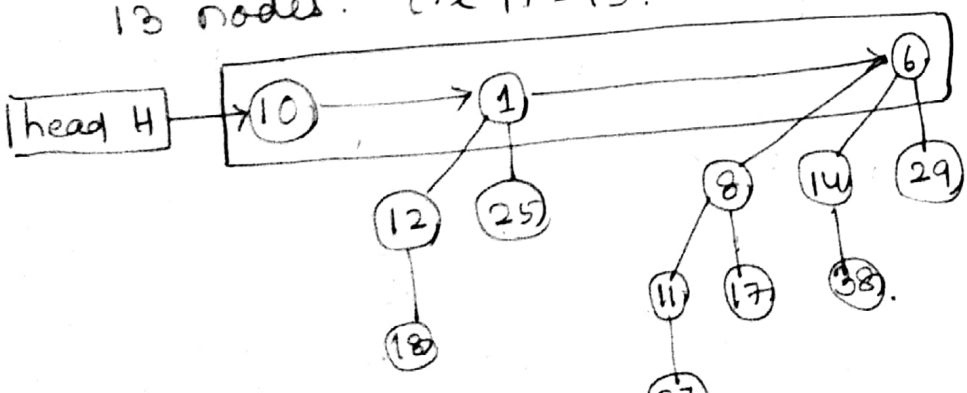
Children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, then child i is the root of a subtree B_i. (Refer to figure c)

Binomial Heap.

A Binomial Heap H is a set of binomial trees that satisfies the following properties:

1. Each binomial tree in H obeys the min-heap property.
   i.e the key of a node is greater than or equal to the key of its parent.
2. For any non-negative integer k, there is at most one binomial tree in H whose root has degree k.
   e.g → it implies that any or node Binomial heap H consists of at most $\lfloor \lg n \rfloor + 1$ binomial Tree.

An Example of Binomial heap H with 13 nodes. i.e $n = 13$.



$n = 13$.

$\lfloor \lg 13 \rfloor + 1$

$\Rightarrow 3 + 1$

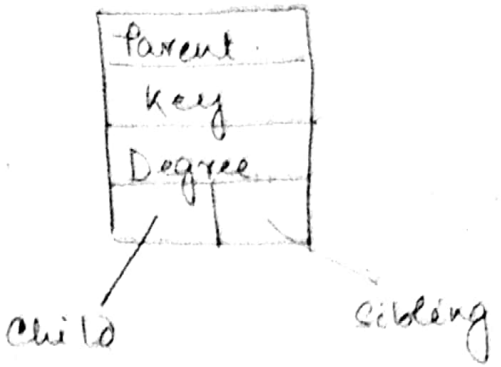
$\Rightarrow 4$.

So at most 4

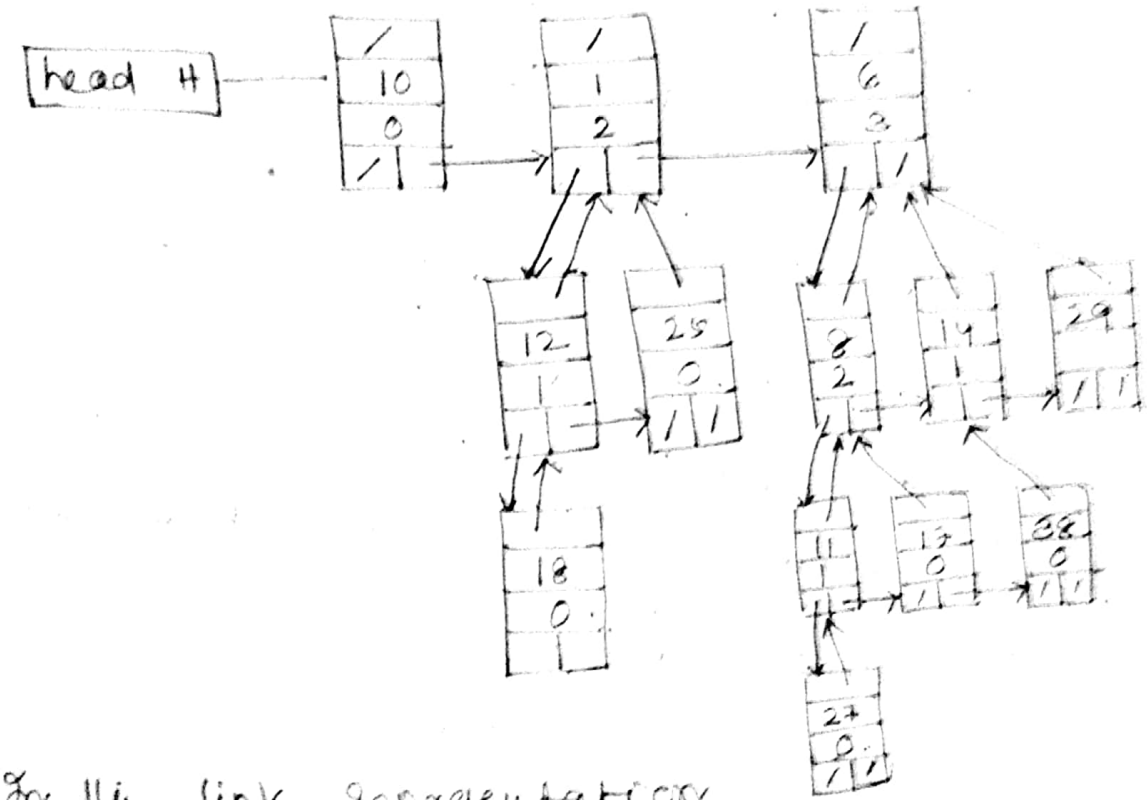
Binomial heap consists of Binomial Trees B_0, B_2, and B_3.

Link representation of Binomial heap.

- Each node of binomial tree contains information about. →



An example of Link representation of Binomial heap.



In the link representation

each node 'n' contains pointers

n → parent.

n → key

n → child.

n → sibling

n → degree.

Functions ~~Operations~~ on Binomial Heap.

1. Makeheap () → Create and return a new heap containing no elements.

2. Insert (H, x) → Insert node x whose key field has already been filled in. to the heap H.

3. Minimum (H) → Return a pointer to the node in heap H where key is minimum.

4. Extractmin (H) → Delete the node from heap H, whose key is minimum returning a pointer to the node.

5. Union (H_1, H_2) → Create and return a new heap that contains all the node of H_1 & H_2. H_1 & H_2 are destroyed by this operation.

6. Decrease key (H, x, K) → Assign a node x within heap H, the new key value K which is assumed no greater than its current value.

7. ~~Decrease key~~ Delete (H, x) → delete node x from the heap H.

Operations on Binomial Heap:

1. Create a new Binomial Heap

To make an empty Binomial heap the Make_heap() procedure simply allocates the memory for the heap.

The running time is $O(1)$

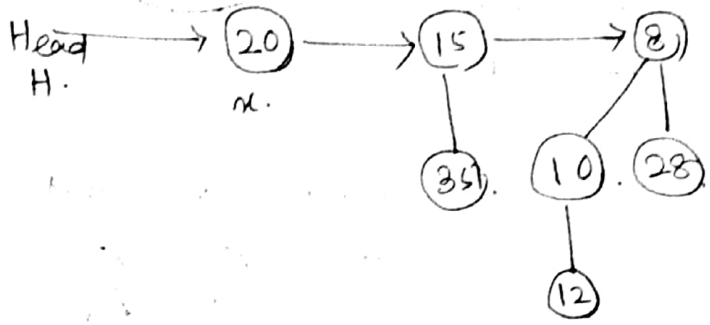
2. Find the minimum key in Binomial Heap.

The procedure Binomial-heap-Minimum(H) is used to find minimum element in a binomial heap.

Binomial-heap-Minimum(H).

```
1. y ← NULL
2. x ← H → head
3. minimum = ∞.
4. while (x != NULL)
5.     if (x → key < minimum)
6.         then minimum ← x → key.
7.              y ← x.
8.              x = x → sibling
9. return y.
```

Example.



x = 20.
x → key < min.
min = 20.
y = 20.
x = 15
x → key < min.
min = 15
y = 15
x = 8.
x → key < min.
min = 8
y = 8
x = NULL.

returns. y = 8.

(iii) Union of two binomial heap :-

The union operation of two binomial heap is used as a sub routine binomial-heap-Union (H_1, H_2). This operation is used mostly on the remaining on all the operations. The binomial-heap-union procedure recursively link binomial trees whose root have same degree. The binomial heap-union function required binomial-link procedure for linking two binomial heap tree. There are four cases for the operation of binomial heap union. These cases are discussed in the next page.

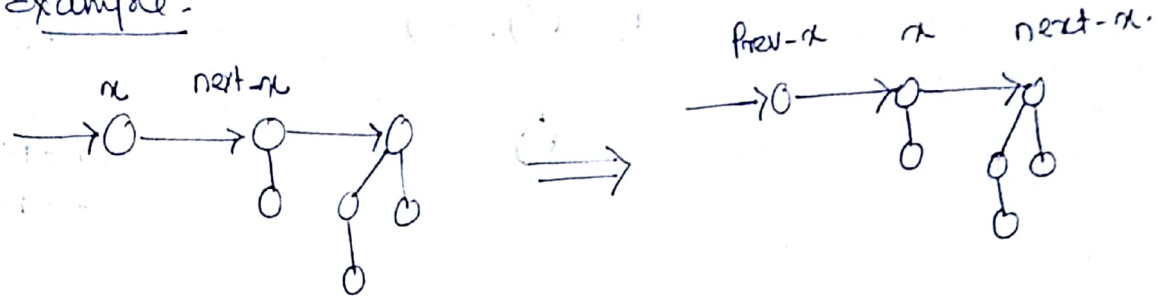
Case-1 if degree [x] ≠ degree [next-x]

then

Prev-x ← x

x ← next-x.

Example-



Case-2 if degree [x] = degree [next-x] =

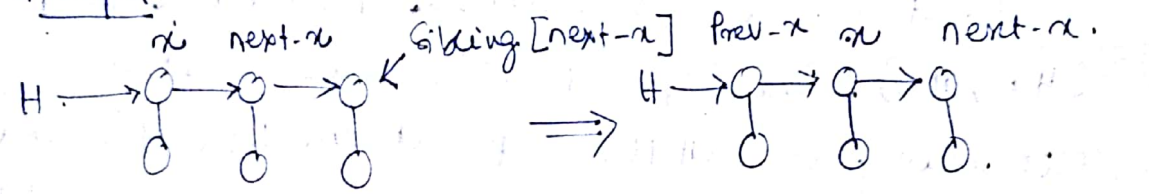
degree [sibling [next-x]]

then

Prev-x ← x

x ← next-x.

Example.



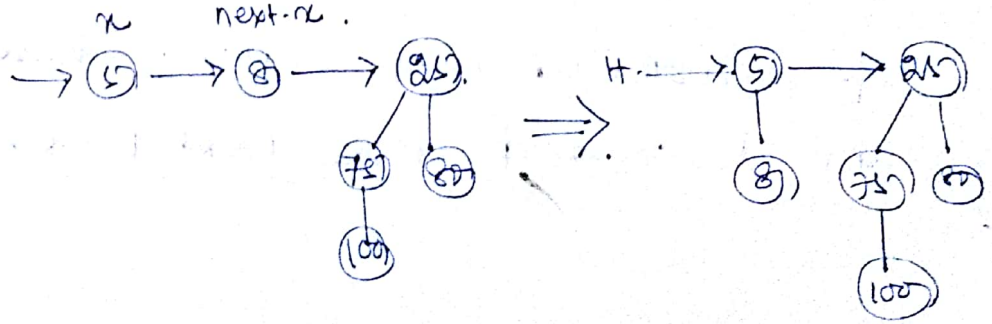
Case-3 if degree [x] = degree [next-x] and

key [x] ≤ key [next-x].

then sibling [x] ← sibling [next-x].

Binomial link (next-x, x).

Example:

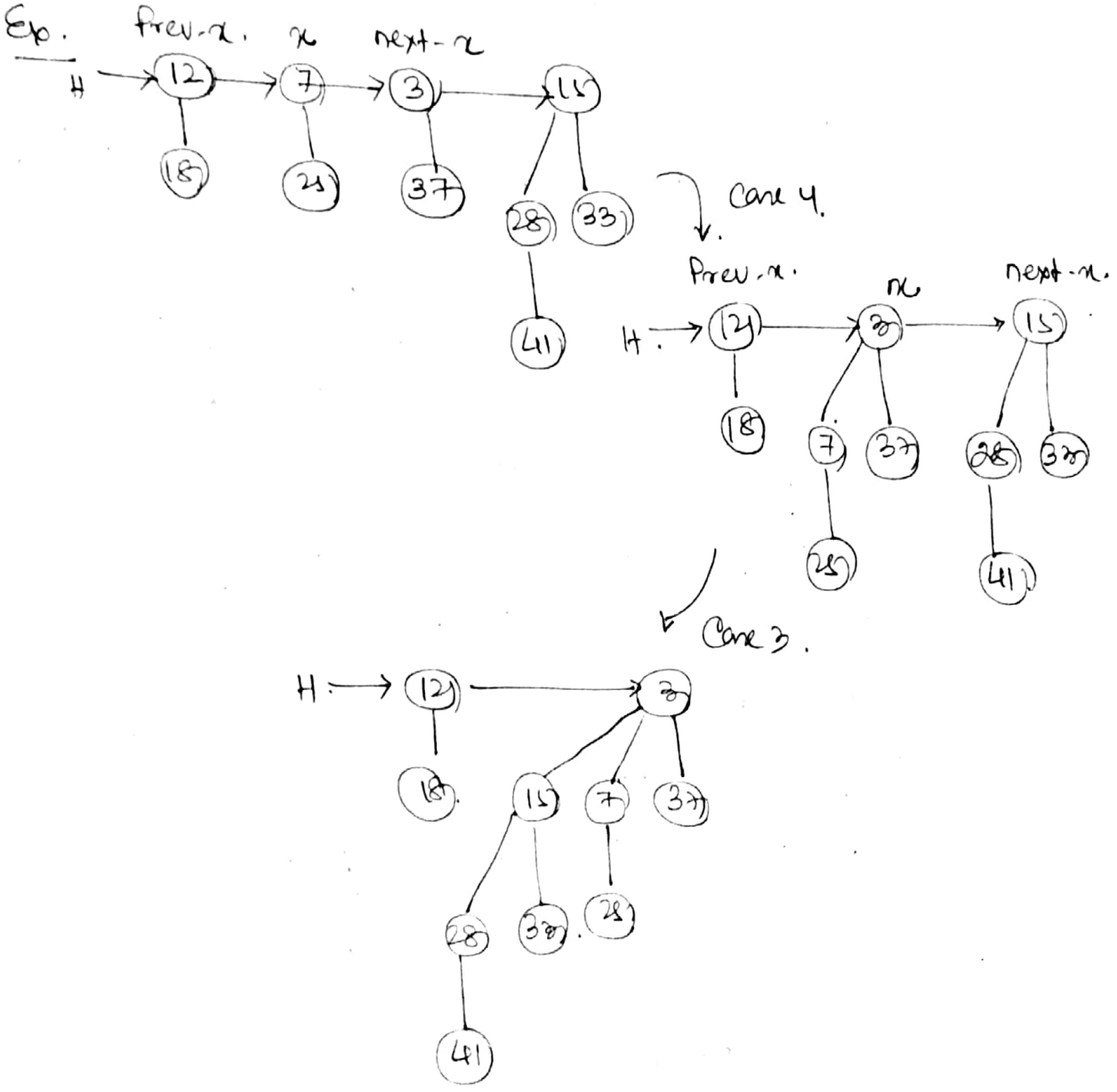


Case 4 — If degree[x] = degree[next-x] and key[x] ≤ key[next-x]

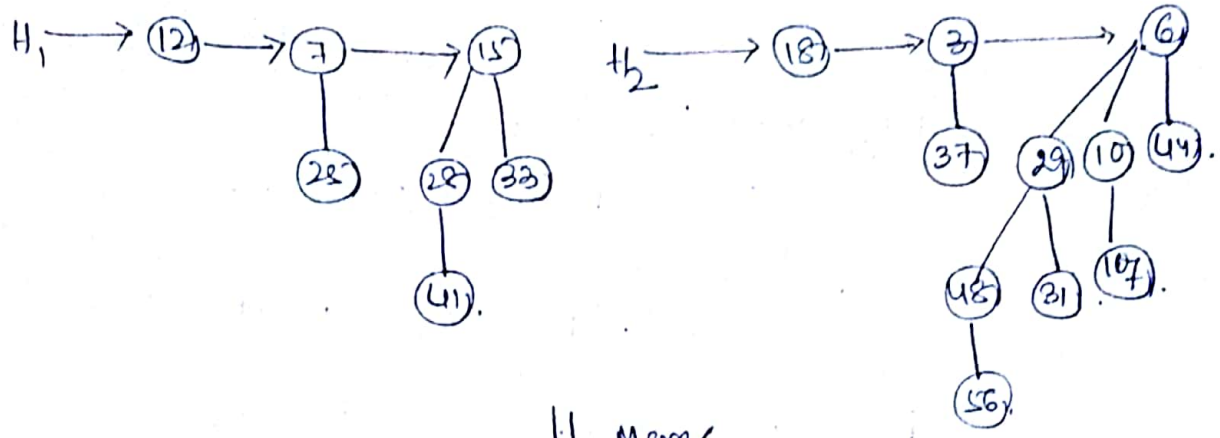
BH = 5

```
if (prev-x == NULL)
    then head[H] = next-x.
else
    sibling[prev-x] = next-x
Binomial-link(x, next-x).
```

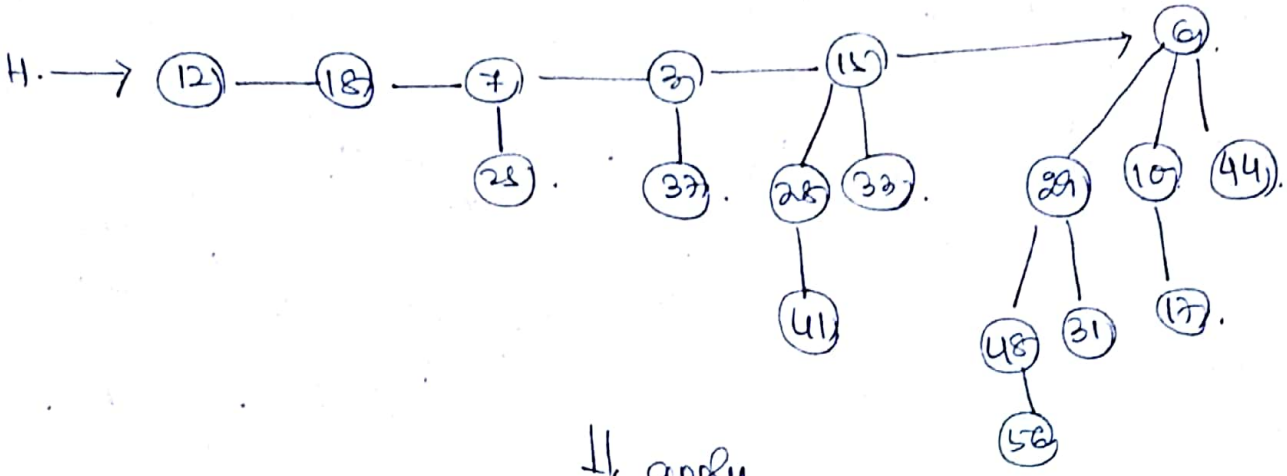
Ex.



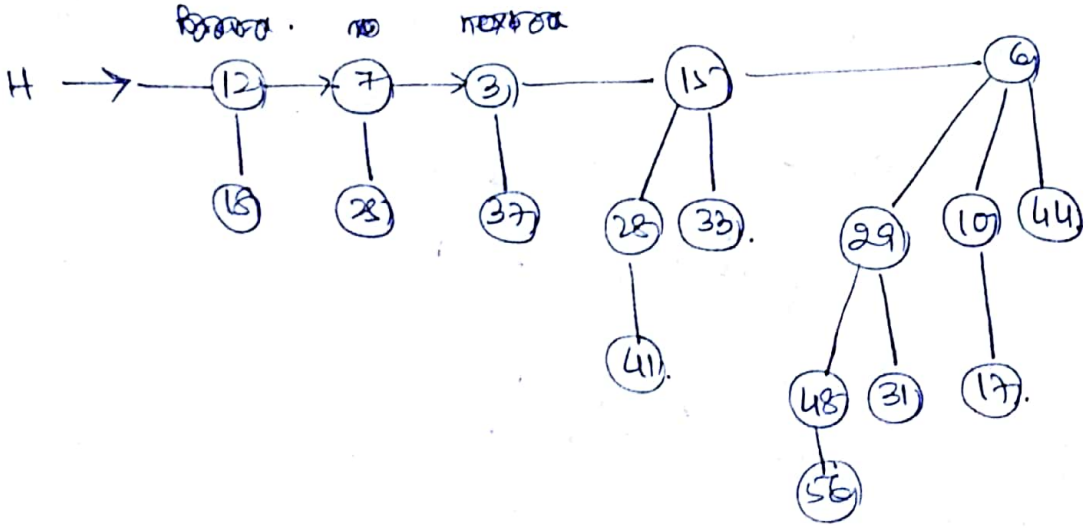
Question. Merge the following two Binomial heap.



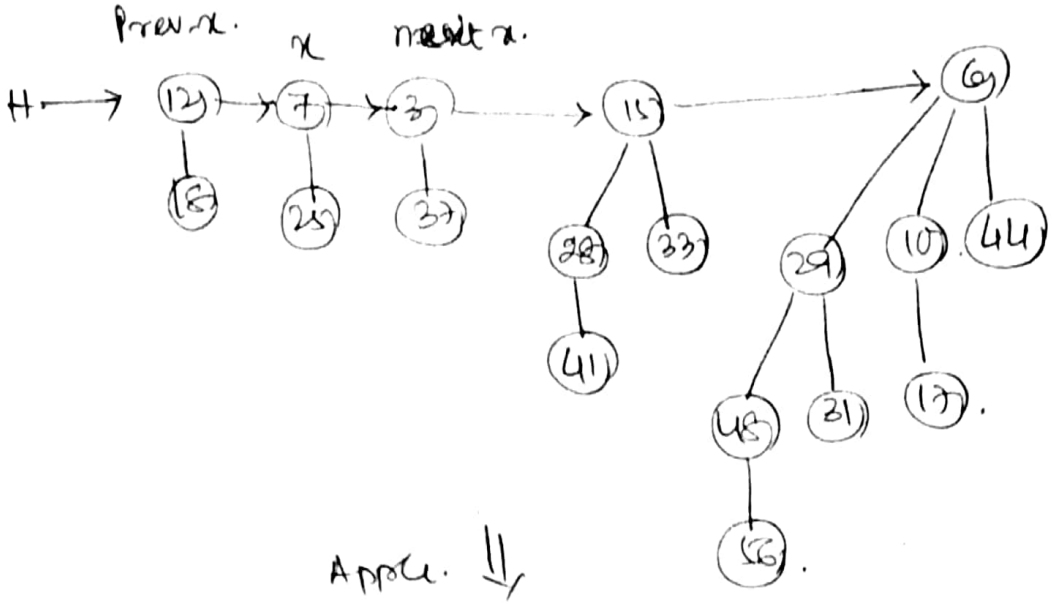
↓↓ Merge.



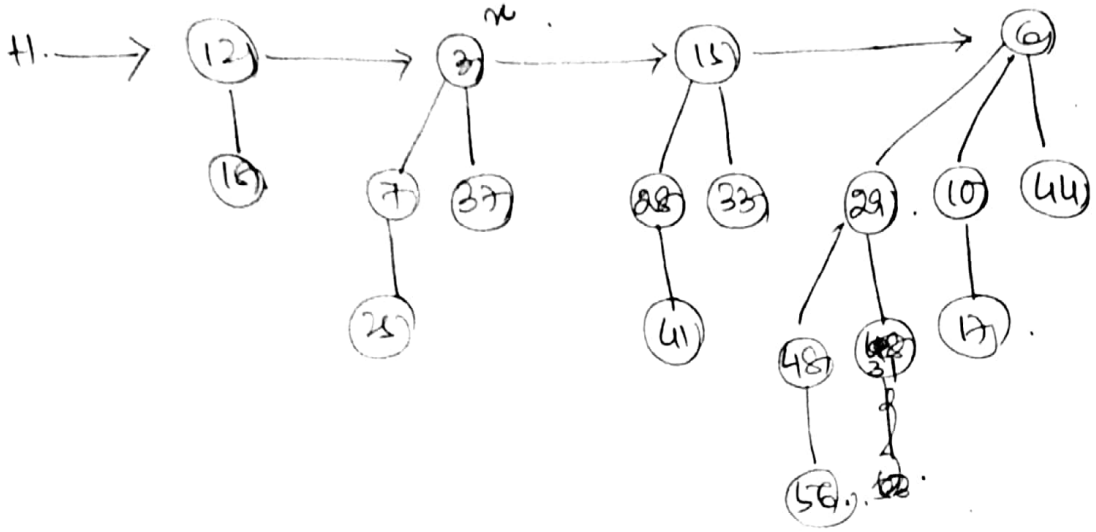
↓↓ apply case-3.



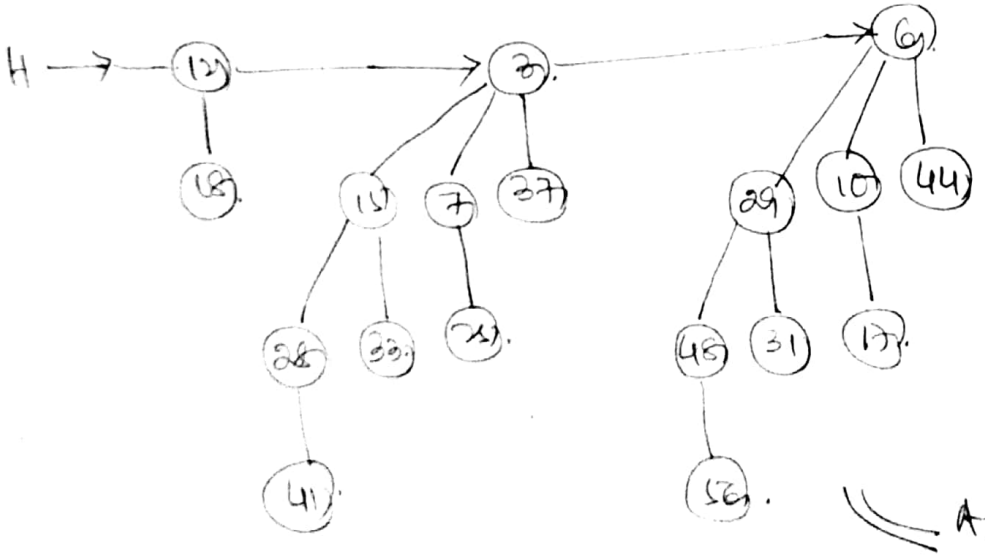
Apply. ⇒ case-2



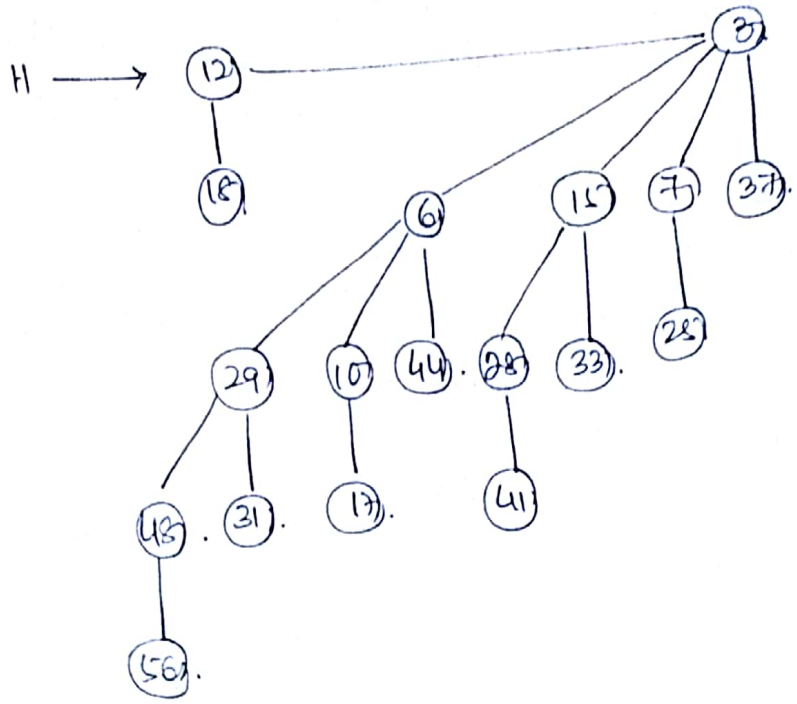
Apply ⇓ Case-4



Apply Case-3 ⇓



Apply Case 3.



Binomial-Link (y, z)

1. p[y] ← z
2. sibling[y] ← child[z]
3. child[z] ← y
4. degree[z] ← degree[z] + 1

Binomial-Heap-Union (H_1, H_2).

1. H ← Make-Binomial-Heap()
2. head[H] ← Binomial-heap-Merge(H_1, H_2)
3. free the objects H_1 and H_2 but not the list they point to.
4. if head[H] = NULL
5. then return H.
6. prev-x ← NULL
7. x ← head[H]
8. next-x ← sibling[x]
9. while next-x ≠ NULL
10. do if (degree[x] ≠ degree[next-x]) or
(sibling[next-x] ≠ NULL and
degree[sibling[next-x]]
= degree[x])

```
Case 1 & Case 2   11.        then prev-x ← x
                  12.             x ← next-x
                  13.        else if key[x] ≤ key[next-x]
                  14.             then sibling[x] ← sibling[next-x]
Case 3            15.                  Binomial-Link(next-x, x)
                  16.             else if prev-x = NULL
                  17.                  then head[H] ← next-x
Case 4.           18.                  else sibling[prev-x] ← next-x
                  19.                  Binomial-Link(x, next-x)
                  20.             x ← next-x
                  21.        next-x ← sibling[x]
                  22. return H.
```

Running time of Binomial-heap-Union (H_1, H_2) is $O(\lg n)$.

Running time of Binomial-Link(y, z) is $O(1)$

Finding the minimum key

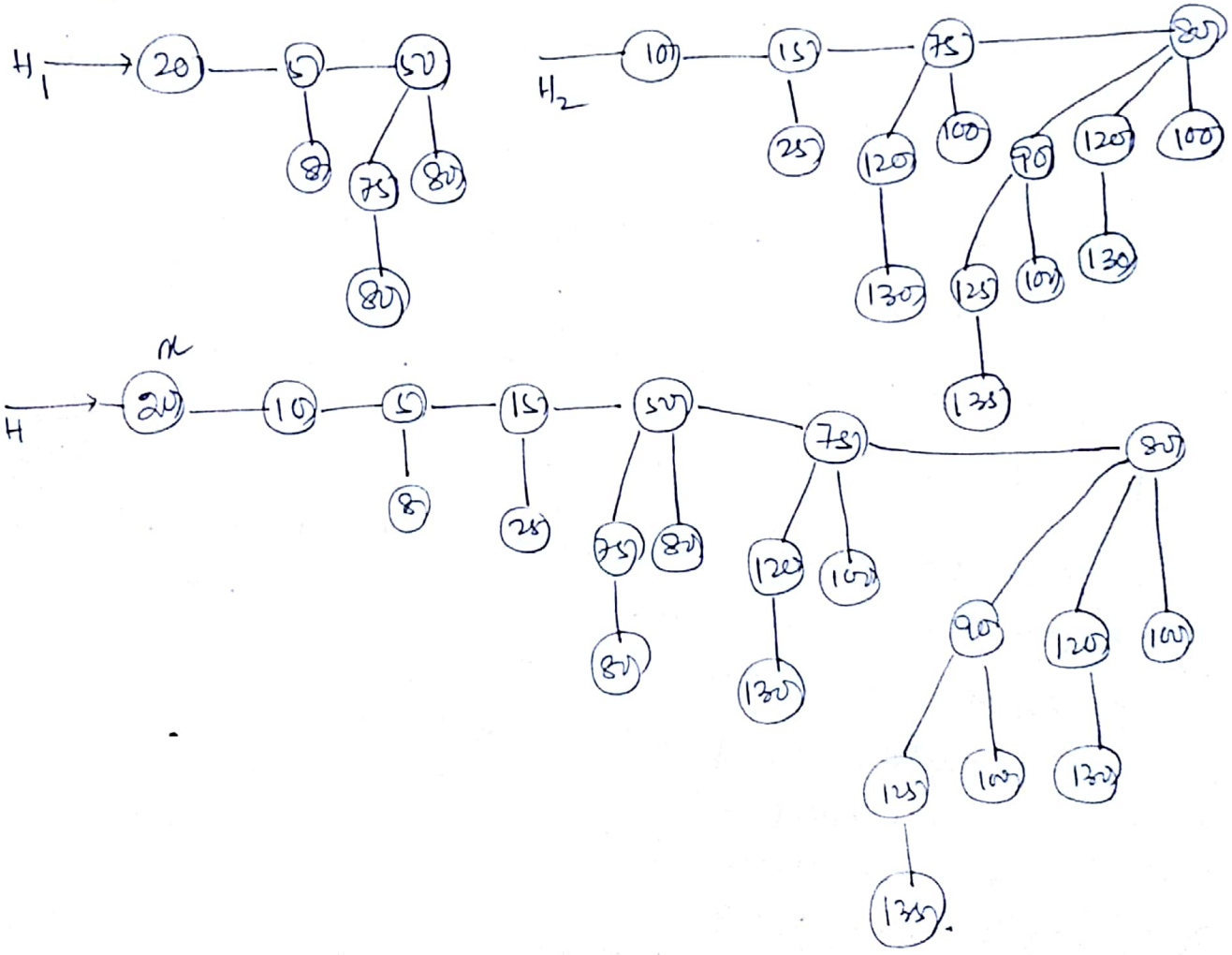
The procedure Binomial-heap-minimum returns a pointer to the node with minimum key in an n-node binomial heap H.

```
Binomial-heap-minimum(H)
1. y ← NULL
2. x ← Head[H]
3. min ← ∞
4. while x ≠ NULL
5.       do if key[x] < min
6.             then min ← key[x]
7.                  y ← x
8.       x ← sibling[x]
9. return y.
```

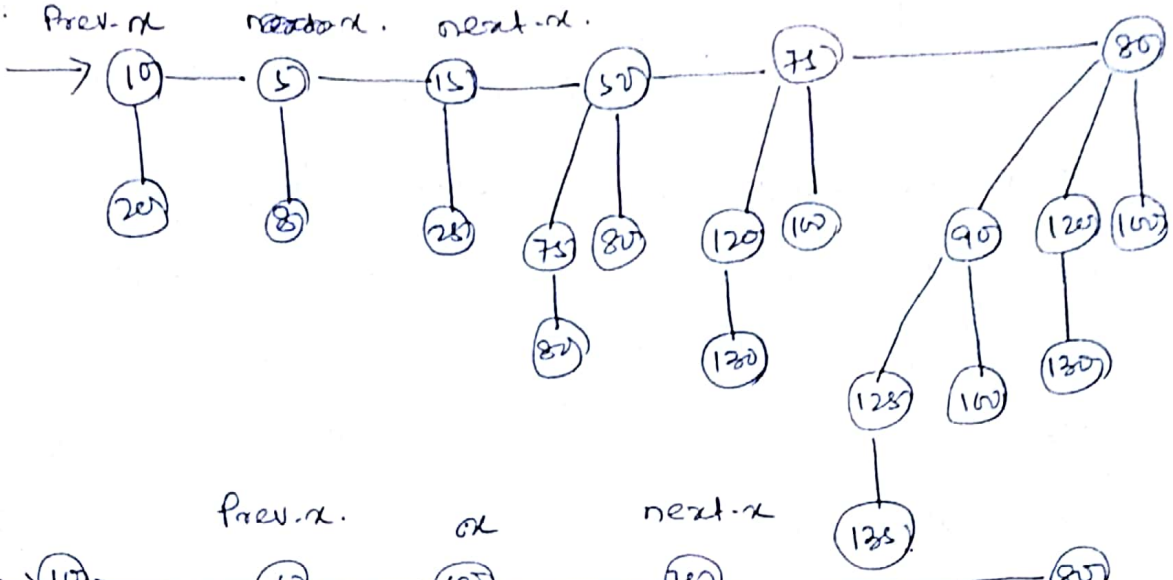
Since Binomial heap is minimum-heap ordered, the minimum key must reside in the root node. The Binomial-heap-minimum procedure checks all roots, which number at most $\lfloor \lg n \rfloor + 1$. When called on the binomial heap, Binomial-heap-minimum returns a pointer to the node with key 1.

Because there are at most $\lfloor \lg n \rfloor + 1$ roots to check the running time of Binomial-heap-minimum is $O(\lg n)$.

Q: Merge two Binomial heap by using Union algorithm

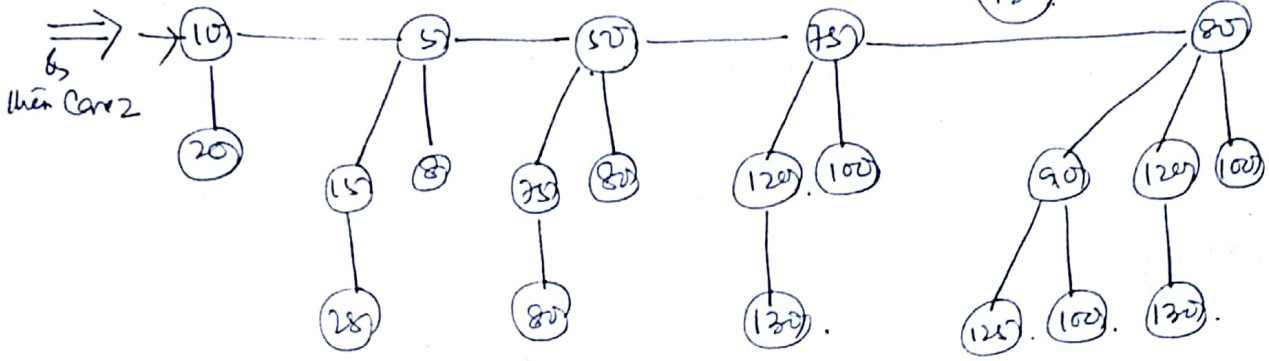


Case-4 (or Case-2 & then Case-2)

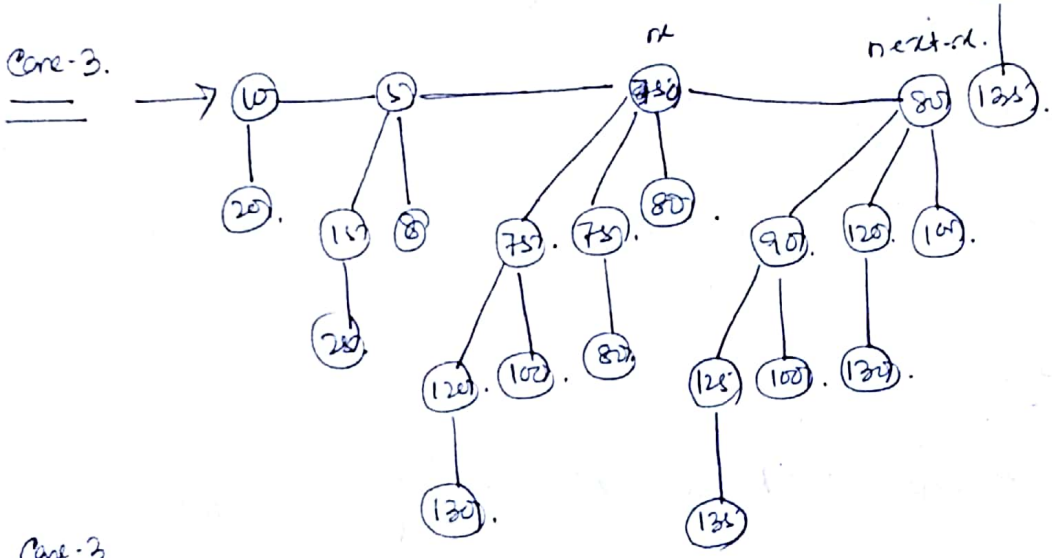


Case-3

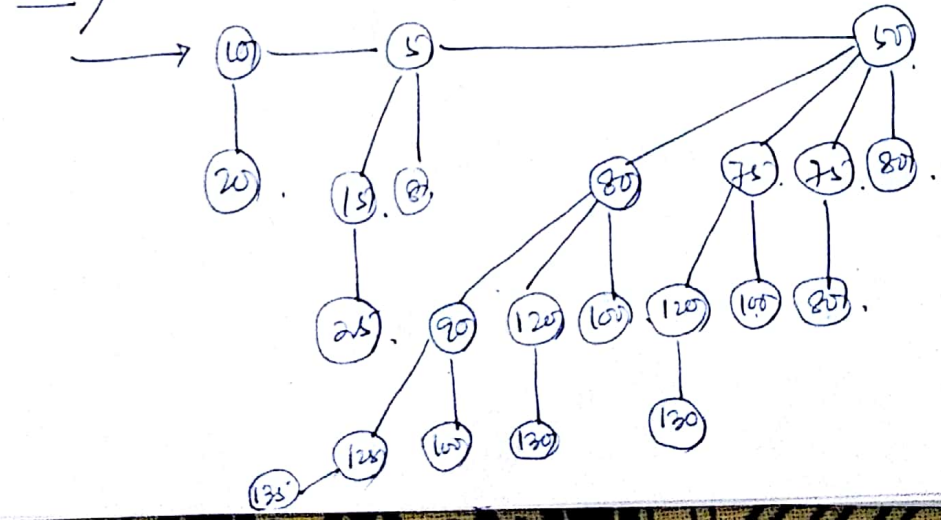
then Case-2



Case-3.



Case-3

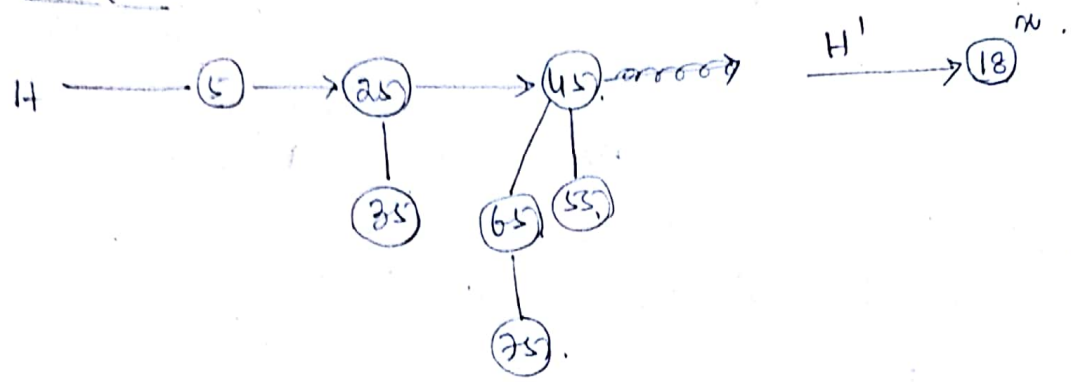


Inserting a node into Binomial heap.

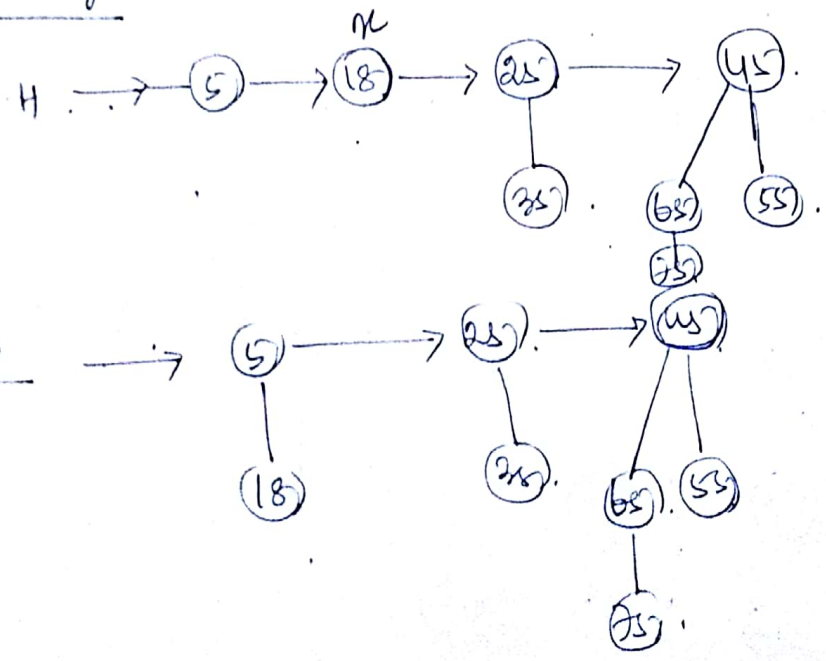
Binomial_heap_Insert (H, x).

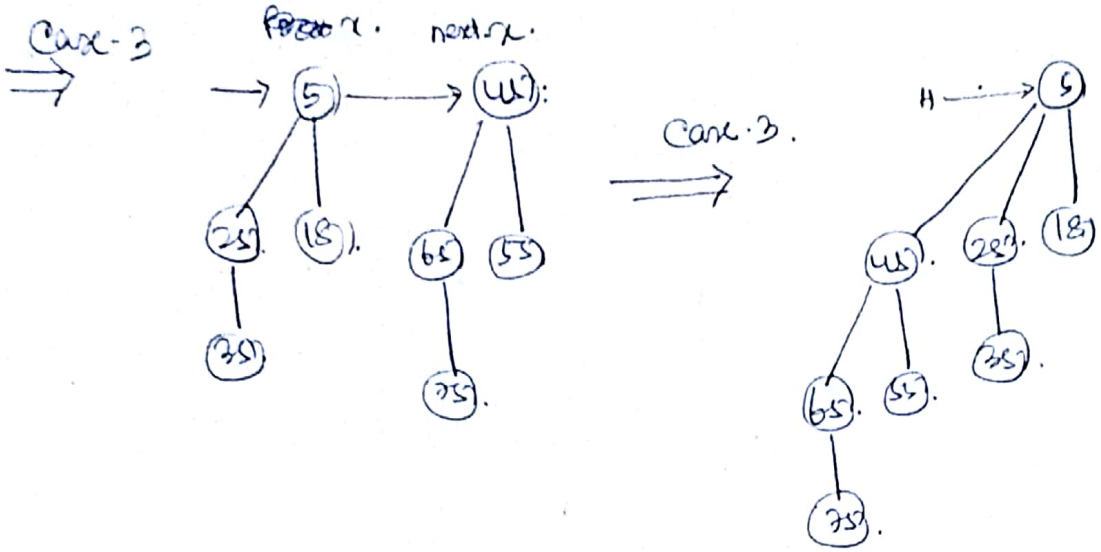
1. H' ← Make_Binomial_heap().
2. P[x] ← NULL
3. child [x] ← NULL
4. Sibling [x] ← NULL
5. degree [x] ← 0.
6. Head [H'] ← x.
7. H ← Binomial_Heap_Union (H', H)

Example:



After Merge



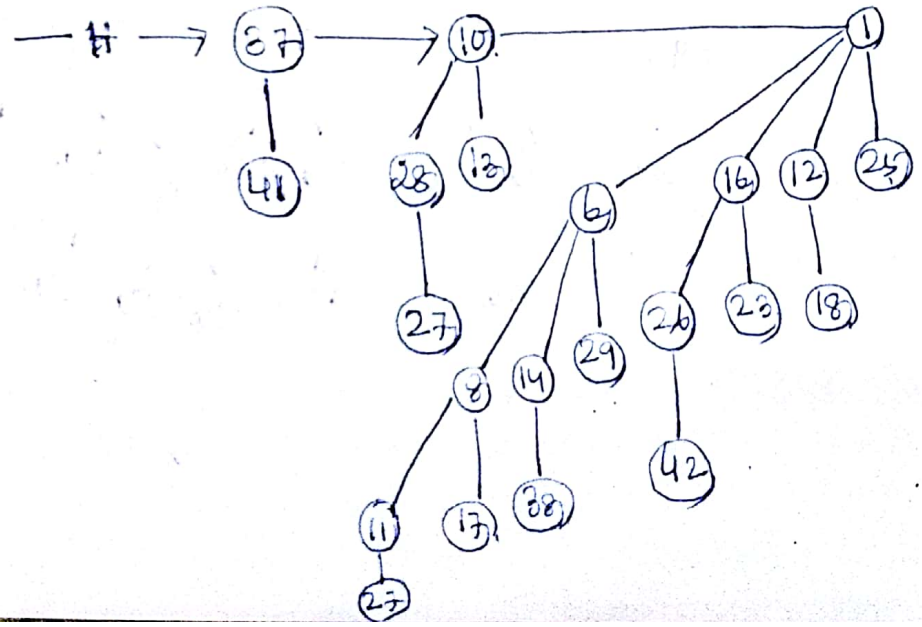


Extracting the node with minimum key.

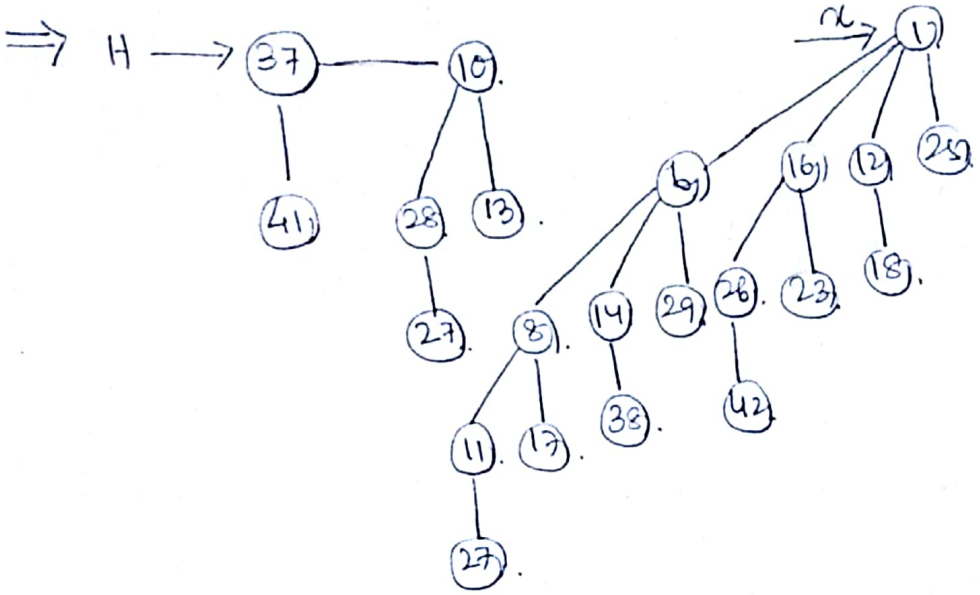
Binomial_Heap-Extract_Min(H).

1. Find the root x with minimum key for the root list H & return x from the root list H.
2. H' ← Make_Binomial_heap()
3. reverse the order of linked list x's children and set the head of resulting list.
4. H ← Binomial_heap_Union(H, H');
5. Return H.

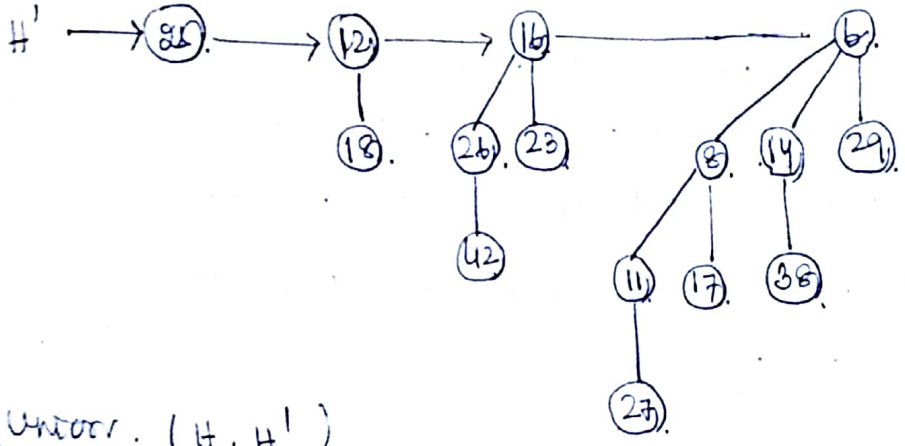
For Example:



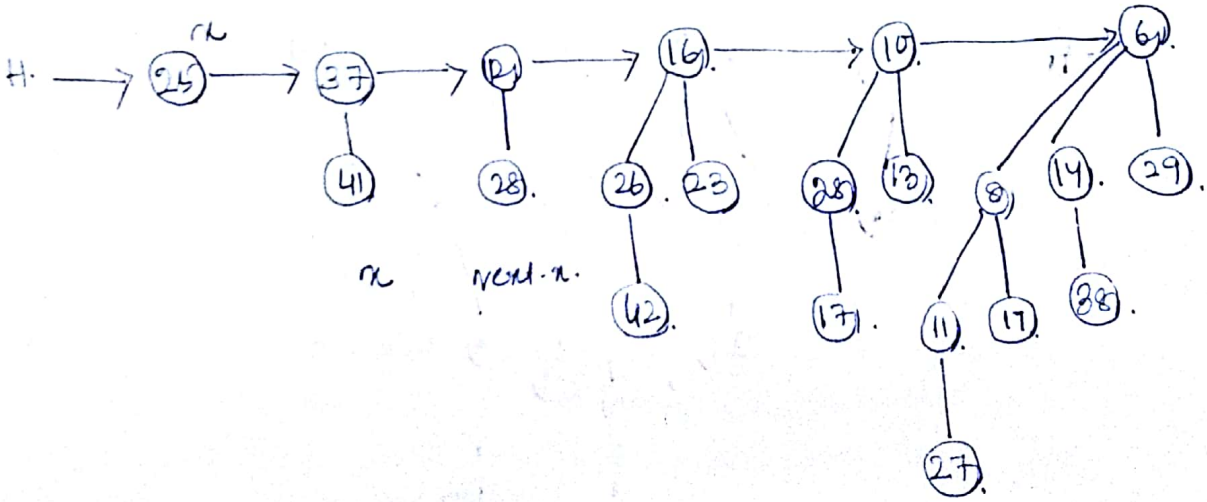
Minimum = 1 (i.e x) and remove it.



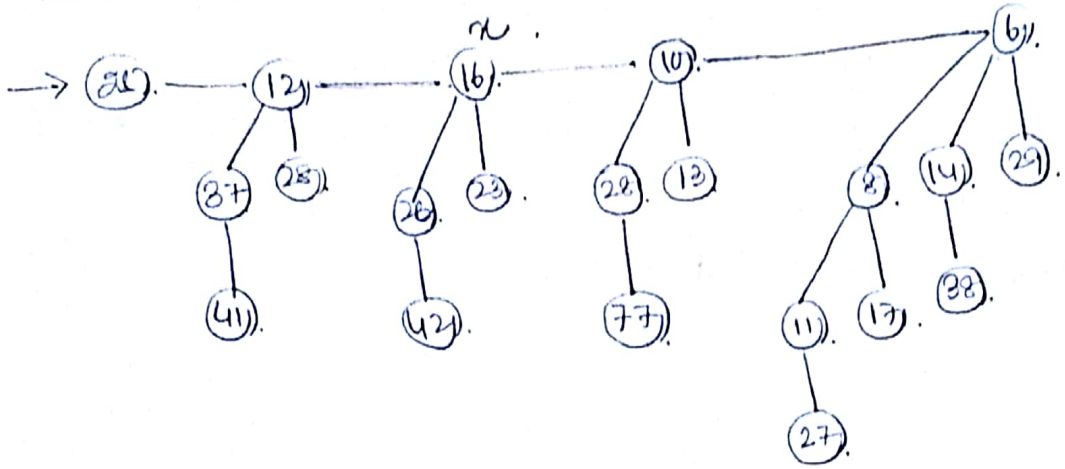
Reverse the order of



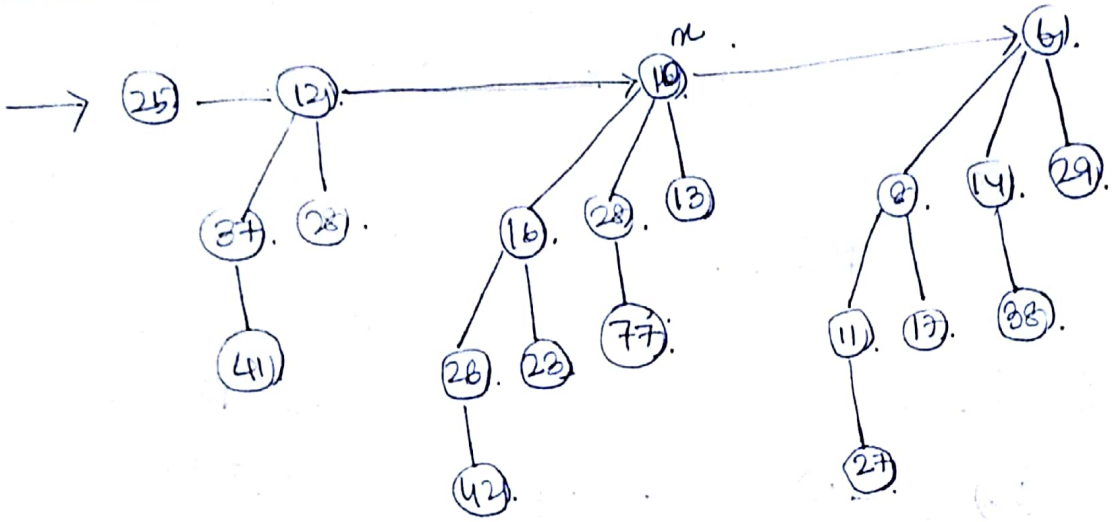
$\Rightarrow$ Union. (H, H')



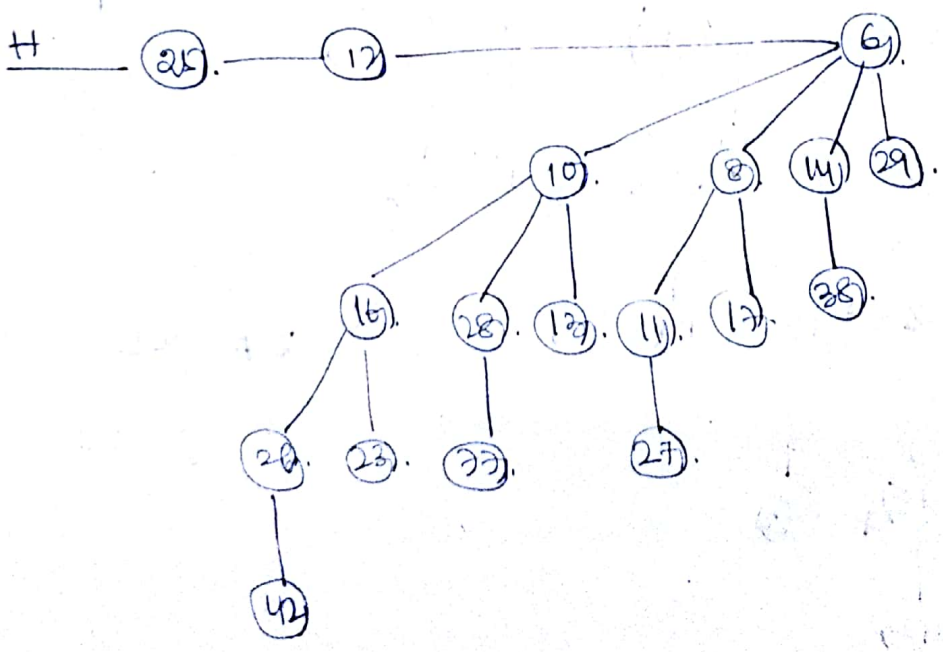
Apply case 1 and then case-4



Apply case-2 & then case-4



Apply case 4 →



Binomial heap decrease key:

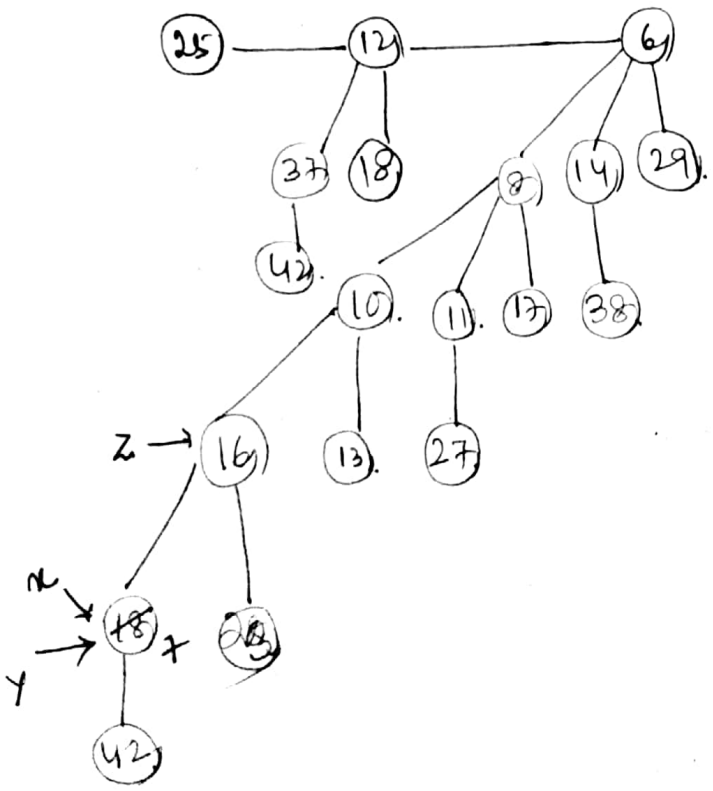
Key to be decreased

Binomial-heap-decreasekey (H, x, K)

↳ value (New)

1. If K > key[x]
2. then error and return.
3. key[x] ← K.
4. Y ← x.
5. z ← p[Y]
6. while (z ≠ NULL & key[y] < key[z])
7. do exchange key[y] ⟷ key[z].
8. Y ← z
9. z ← p[y].

for Example:



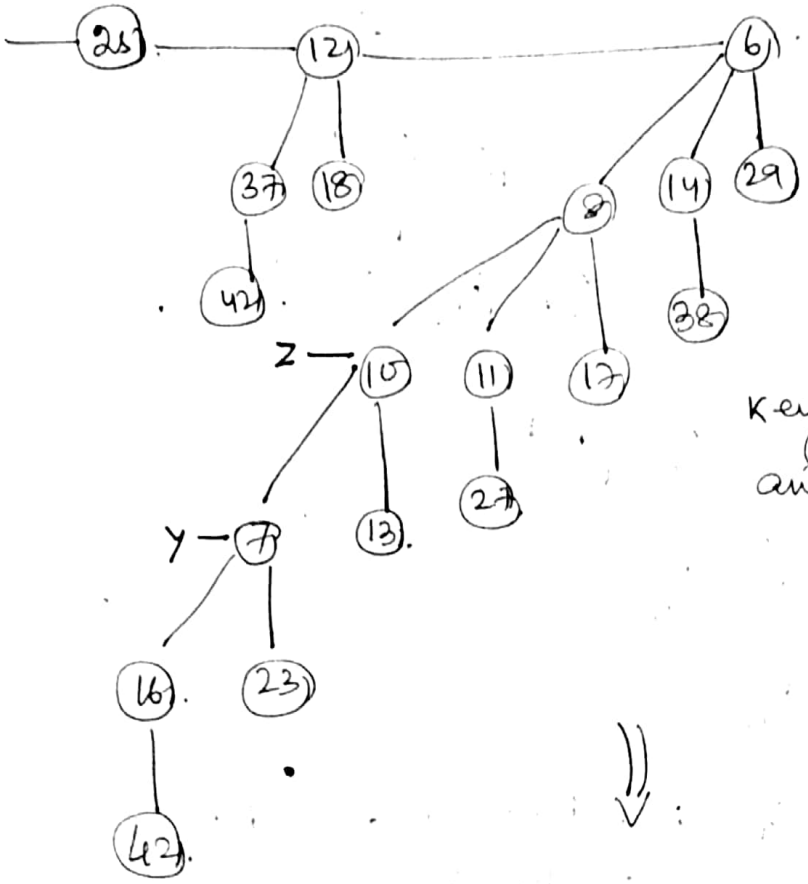
x = 18

K = 7.

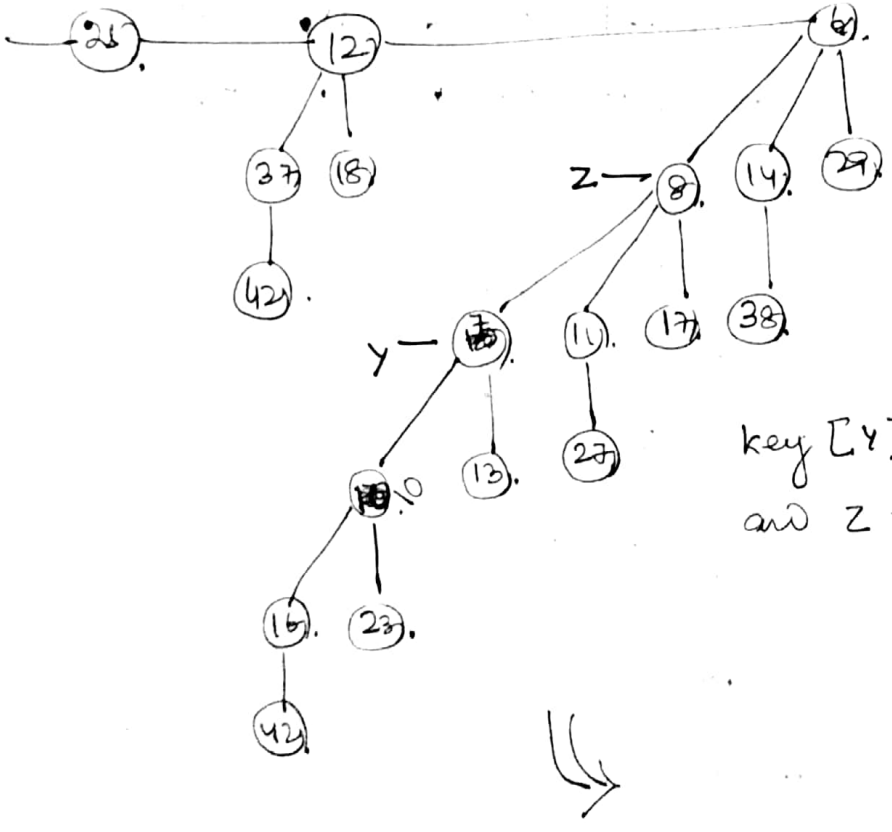
replace key[x] by K

z ≠ NULL and.

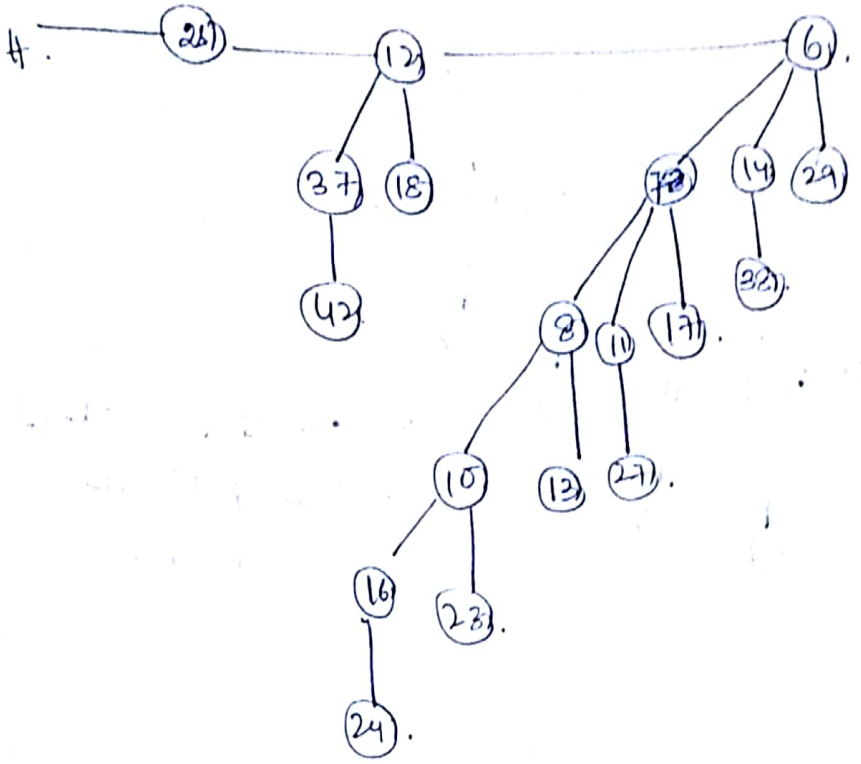
key[y] < key[z] is True.



key[y] < key[z]
and z ≠ NULL
is True.



key[y] < key[z]
and z ≠ NULL
is True.

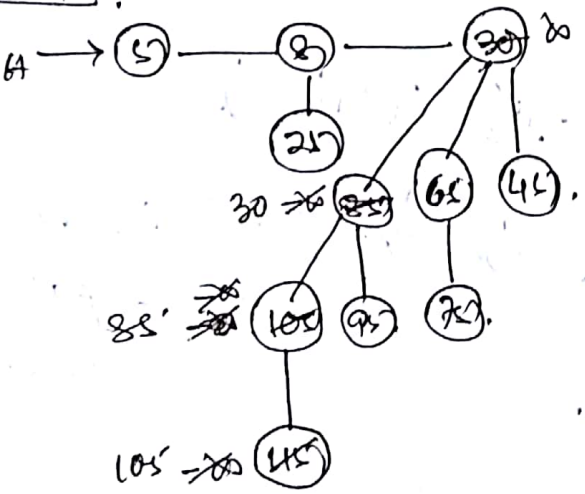


Delete a key from a Binomial heap.

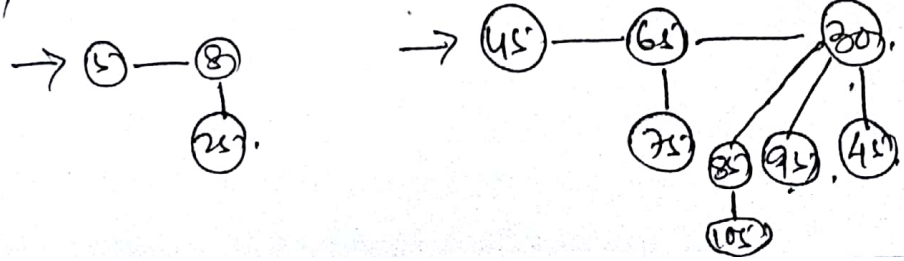
Binomial-heap-delete (H, x).

1. Binomial-heap-Decrease (H, x, $-\infty$)
2. Binomial-heap-Extract-Min (H)

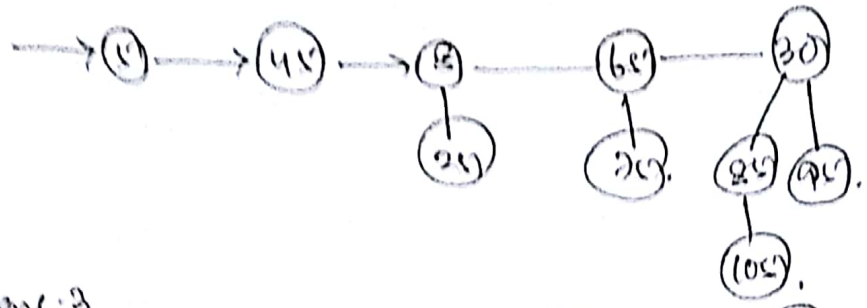
For Example.



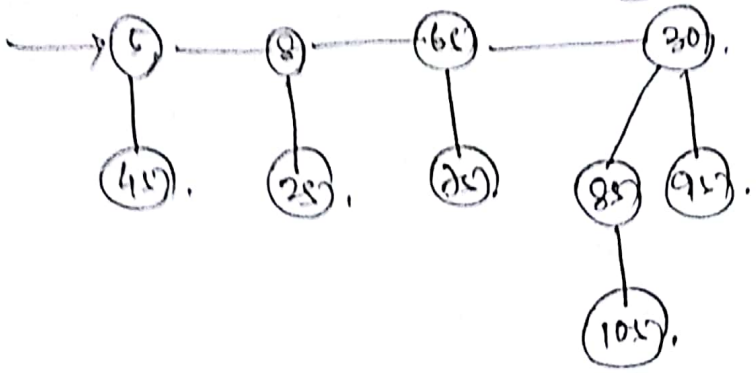
Apply Extract Min.



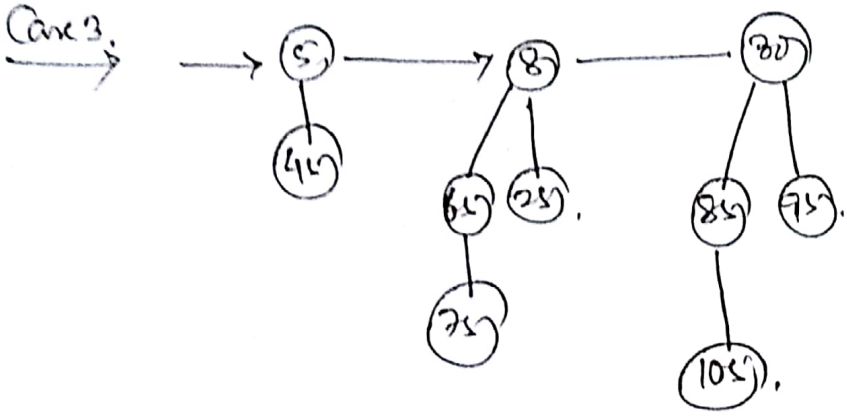
Then Merge two list.



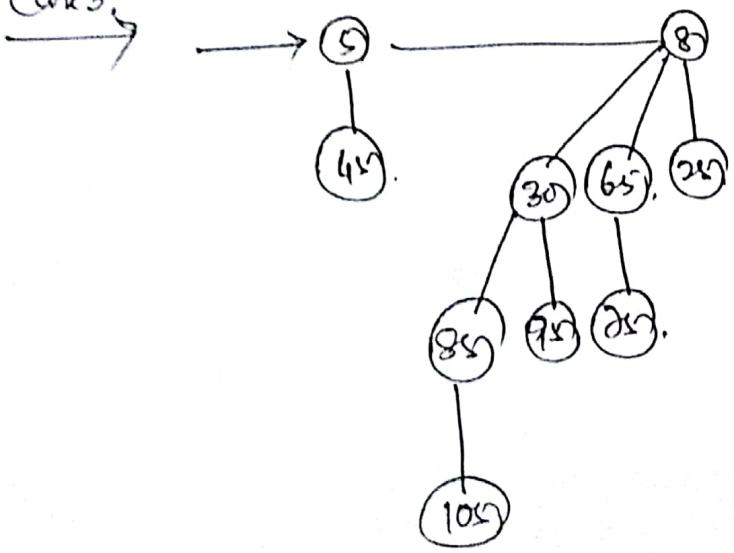
Case 3
→
Case 2



Case 3:



Case 3:



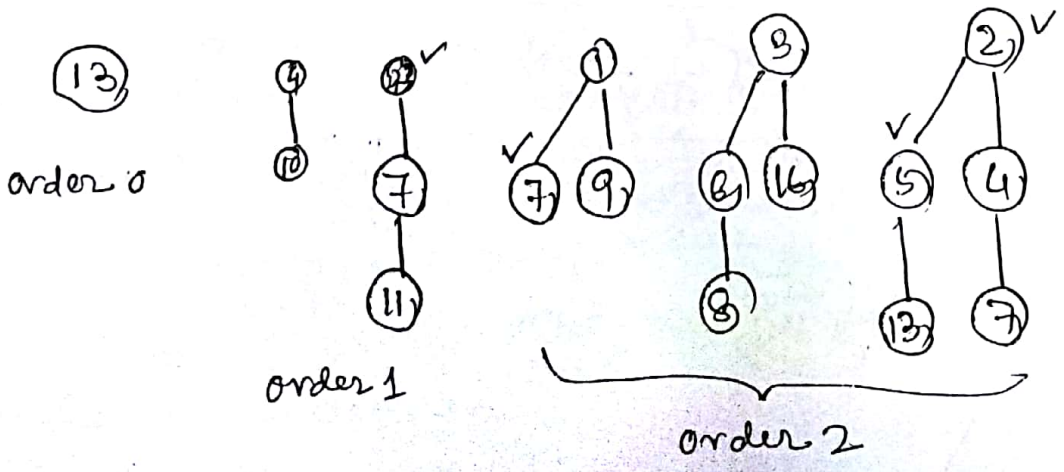
A fibonacci is a heap data structure similar to the binomial heap, only with a few modification and a looser structure.

Structure

Like the binomial heap, a fibonacci heap is a collection of heap-ordered trees. They do not need to be binomial trees however, this is where the relaxation of some of the binomial heap's property comes in.

Each tree has an has an order just like the binomial heap that is based on the number of children. Nodes within a fibonacci heap can be removed from their tree without restructering them, so the order does not necessarily indicate the maximum height of the tree or number of nodes it contain.

Some examples of trees of order 0, 1, and 2 ~~are~~ are given below. (the black nodes are i.e ✓ marked known as marked.



→ It is a collection of minimum heap ordered tree. Tree within fibonacci heap are rooted but not ordered in degree. (un-ordered),

→ Each node x contain following field.

p(x) ← pointer to its parent.

child(x) ← pointer to its child.

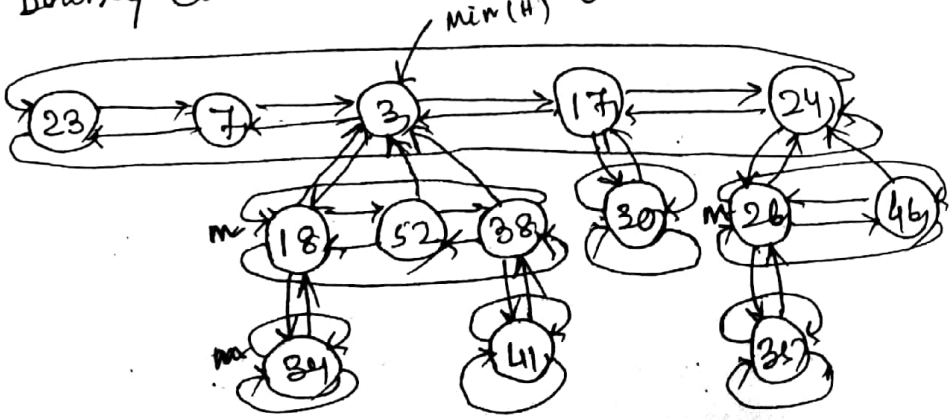
degree(x) ← no. of children in the child list of H.

mark(x) ← This is the boolean value. It indicates whether the node x has lost a child since the last time x was made the child of another node.

⇒ Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node.

Children of x are link together in a circular doubly link list. Each child y in the child list has two pointers left(y) & right(y).

An example of fibonacci heap with the help of Doubly circular list is given below,



The Fibonacci heap data structure serves a dual purpose.

→ First it supports a set of operations that constitutes "Mergeable heap",

→ Second, several fibonacci-heap operations run in constant amortized time, as shown in below table:

Procedure	Binary heap (worst-case)	Fibonacci-heap (Amortized)
Make heap	$O(1)$	$O(1)$
Insert	$O(\lg n)$	$O(1)$
Minimum	$O(1)$	$O(1)$
Extract-Min	$O(\lg n)$	$O(\lg n)$
Union	$O(n)$	$O(1)$
Decrease-key	$O(\lg n)$	$O(1)$
Delete	$O(\lg n)$	$O(\lg n)$.

Amortized Analyses.

In a Amortized analysis, we average a time required to perform a sequence of data structure operations overall all operations performed.

With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.

The three most common technique used for Amortized analysis are:

1. Aggregate Analysis.
2. Accounting Method Analysis.
3. Potential Method Analysis.

fibonacci heap support by potential method. This method is used to analyse the performance of fibonacci heap operation for a given fibonacci heap (H) indicated by

$t(H)$ ← no. of trees in root list of H.

$m(H)$ ← no. of mark nodes in heap H.

The potential function of fibonacci heap is defined as

$$\Phi(H) = t(H) + 2 \cdot m(H).$$

for the above eg.

$$\Phi(H) = 5 + 2*3 = 5 + 6 = 11$$

Operations of fibonacci Heap:-

1) Creating a new fibonacci Heap:

To make an empty fibonacci heap, we use Make_fib_Heap(), allocate and returns the fibonacci heap object H. where no. of nodes in H. is zero and min(H) = NULL and $\Phi(H) = 0$. The amortized cost of Make_fib_heap is thus equal to its $O(1)$ actual cost.

(ii) Fibonacci Heap Insertion :-

To insert a node in fibonacci heap we use the following procedure.

Fib_heap_Insert (H, x)

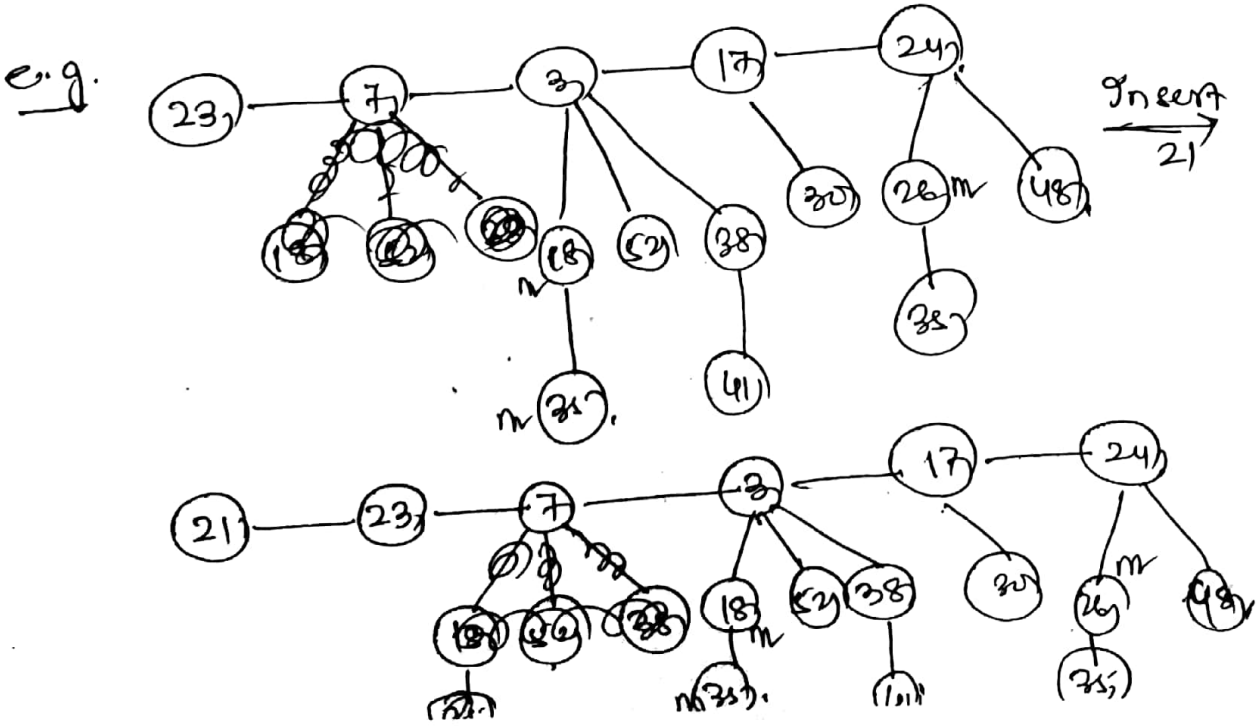
1. degree [x] = 0.
2. P [x] = NULL
3. child [x] = NULL
4. left [x] = NULL.
5. right [x] = NULL.
6. mark [x] = False.
7. Concatenate the root list containing x with rootlist of H.
8. If (Min (H) = NULL or key [x] < Min [H])
9. min (H) ← x.
10. n [H] = n [H] + 1

Amortized cost of Fib. Heap

Amortized cost = Actual cost + $\Delta(\Phi)$ (difference in potential cost)

Let
H ← I/P fibonacci Heap.
H' ← O/P fibonacci Heap.
then as per formula.
$[t(H) + 1 - 2\,m(H')] - [t(H) - 2\,m(H)]$
$= 1$;
So Amortized cost = $O(1) + 1 = O(1)$;

e.g.

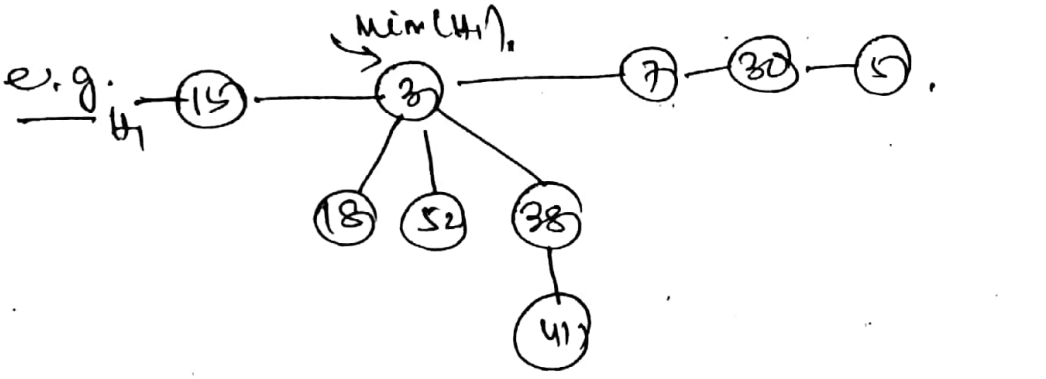


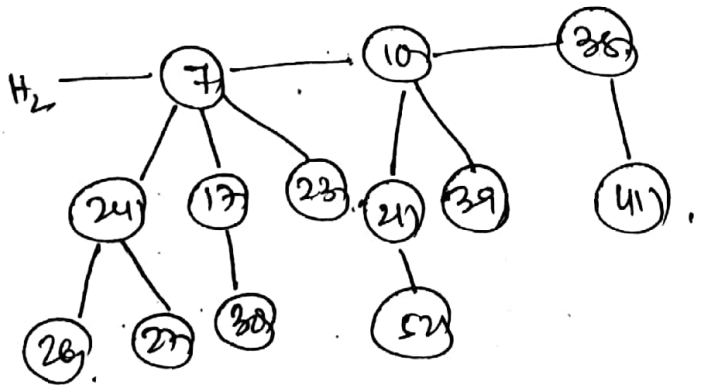
(iii) Uncor of fibonnaci Heap:

Fib-Heap-Uncor (H_1, H_2).

1. H ← Make_fib_heap()
2. min(H) = min(H_1)
3. Concatinate the root list of H_2 with root list H.
4. If (min(H_1) = NULL) or (min(H_2) ≠ NULL & min(H_2) < min(H_1))
5. then min(H) = min(H_2)
6. n[H] = n[H_1] + n[H_2]
7. free the object of H_1 & H_2
8. return H.

e.g.





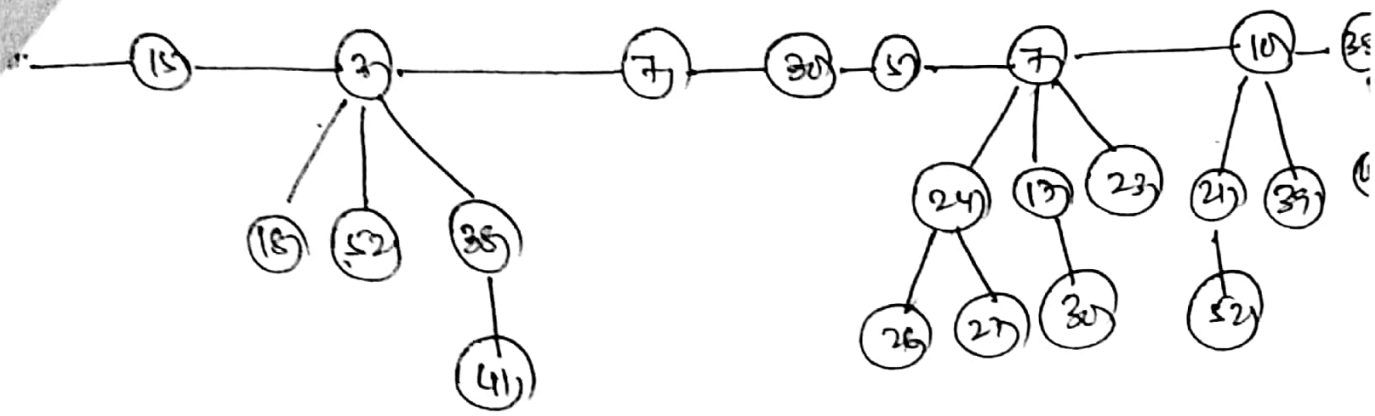
The change in potential cost.

$\phi(H) - (\phi(H_1) + \phi(H_2))$

$= 0$

[because $t(H) = t(H_1) + t(H_2)$

$m(H) = m(H_1) + m(H_2)$]

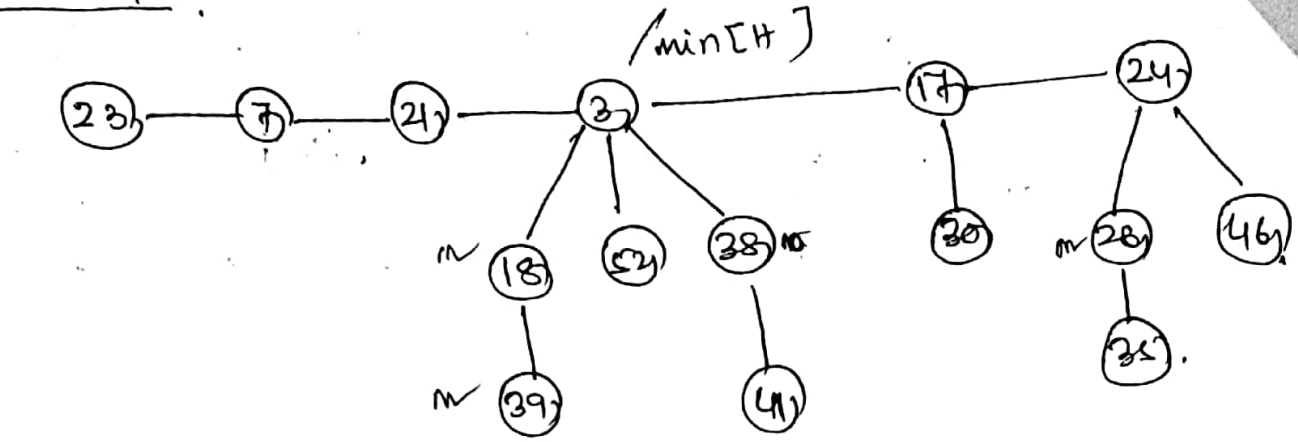


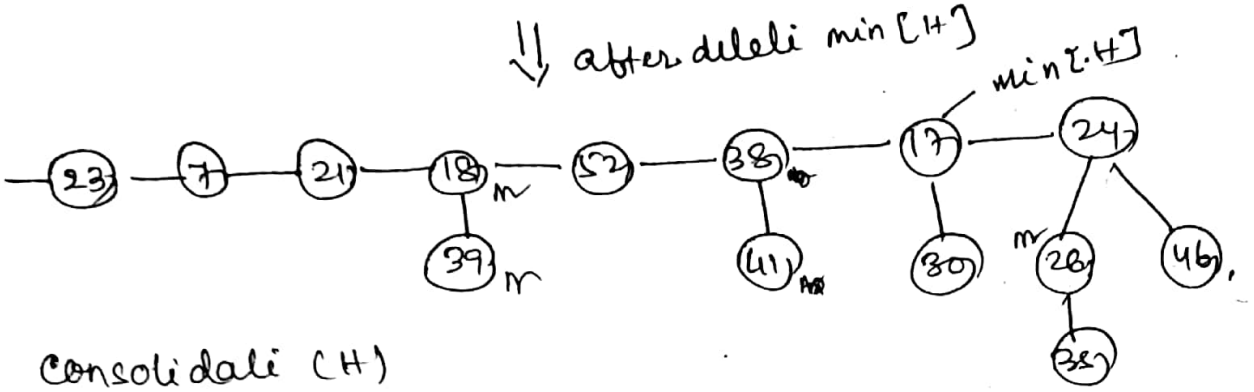
(iv) Extract min. element in a fibonacci Heap.

feb_heap - Extract Min (H).

1. $x \leftarrow$ Min(H)
2. If $x \neq$ NULL
3. for each child x of z
4. add x to the root list of H.
5. $p[x] =$ NULL
6. remove z from the root list of H.
7. If $z =$ right [z]
8. min [H] = NULL
9. else min [H] = right [z],
10. consolidate (H).
11. n[H] = n[H] - 1
12. return z.

for example:





Consolidate (H)

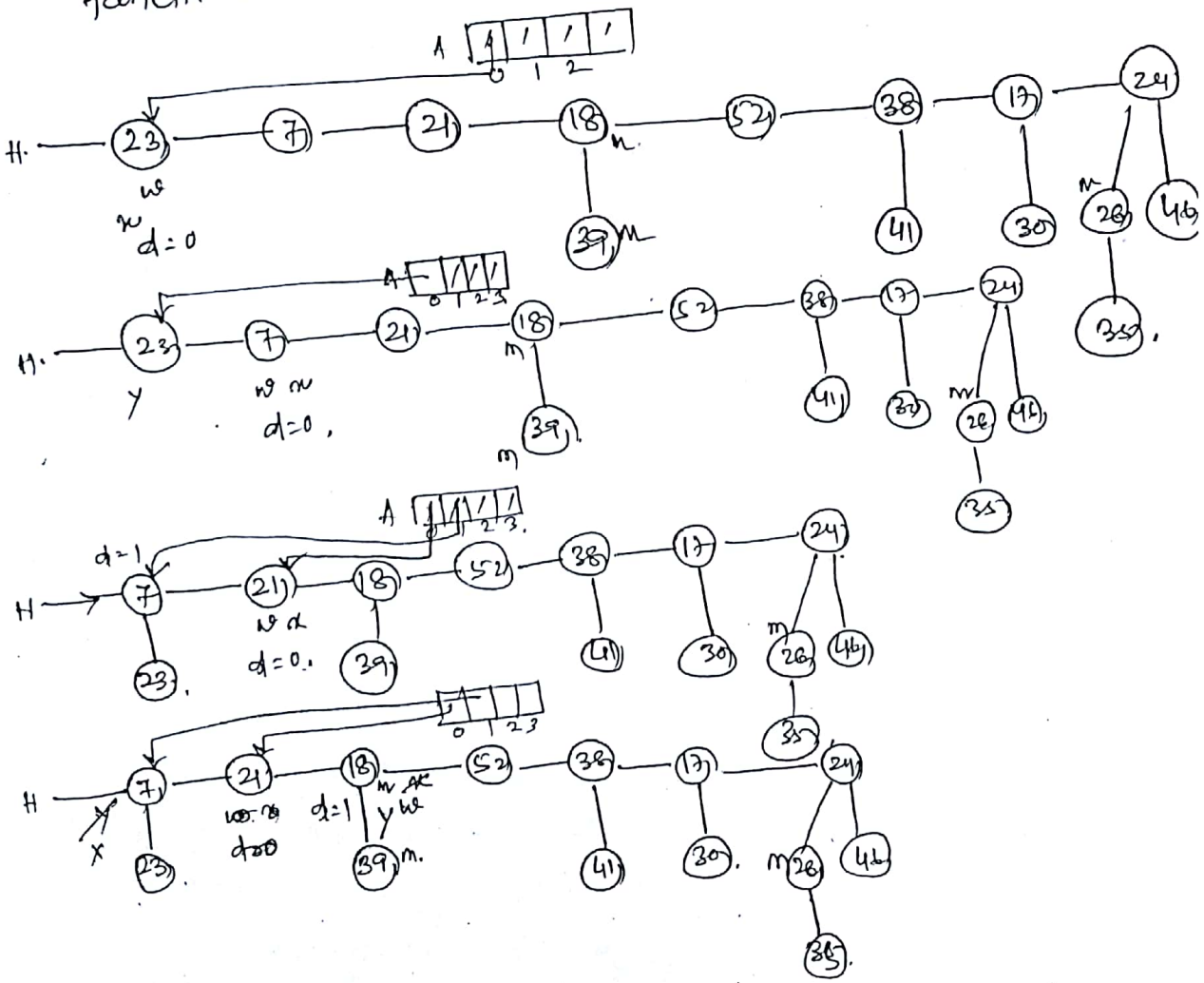
```
1. for i ← 0 to D[N[H]]
2.     do A[i] = NULL
3. for each node w in the root list of H.
4.     do x ← w
5.        d ← degree[x]
6.        while A[d] ≠ NULL
7.           do y ← A[d]
8.              if (key[x] > key[y])
9.                 then exchange(x ↔ y)
10.             fib-heap-link(H, y, x).
11.             A[d] ← NULL
12.             d ← d+1
13.          A[d] ← x.
14. min[H] ← NULL
```

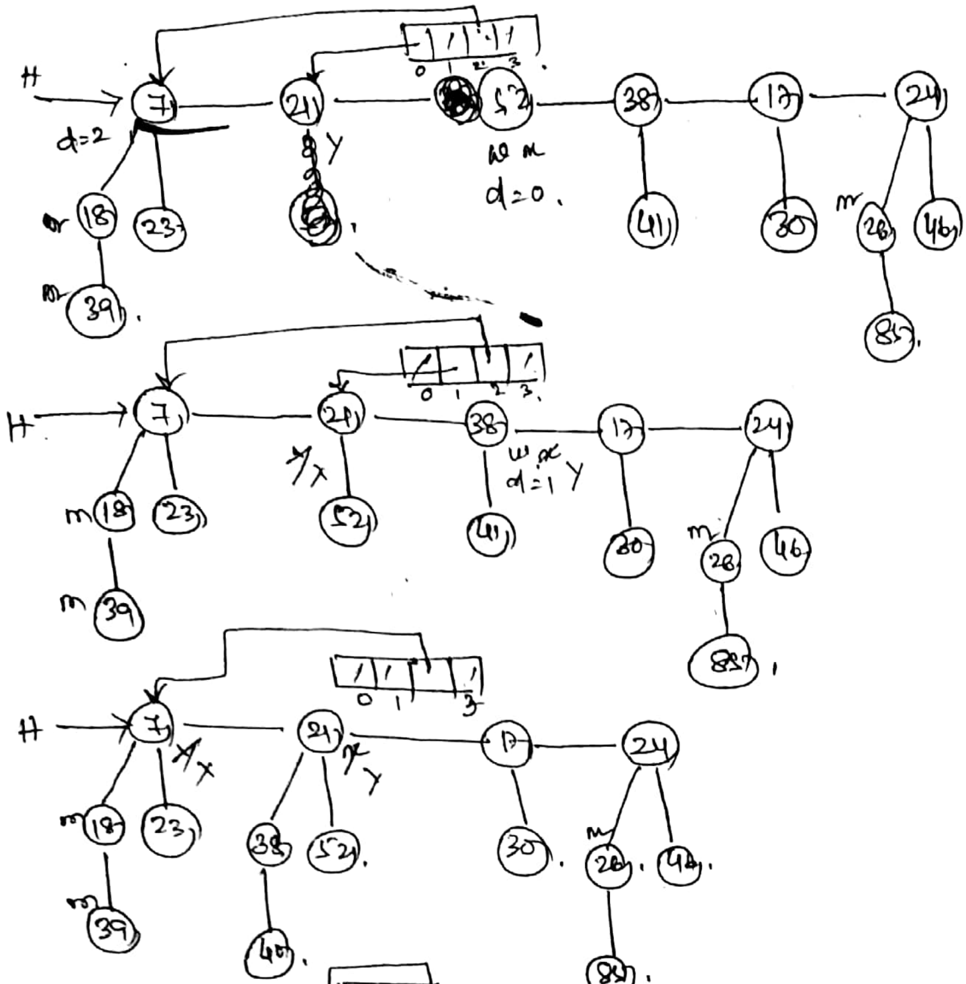
15. for i ← 0 to D(n[H])
16. do if A[i] ≠ NULL
17. then add A[i] to the root list of H.
18. if (min[H] = NULL) or (key[A[i]] < key[min[H]])
19. then min[H] ← A[i]

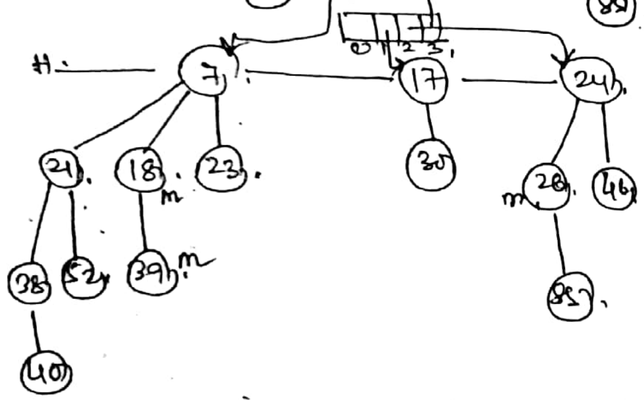
Fib-Heap-link (H, y, x)

1. Remove y from the root list of H.
2. make y a child of x, incrementing degree[x]
3. mark[y] ← false.

Now consolidate (H) function:







At the beginning min[H] = 17.

$i = 0$ A[0] ≠ NULL ~~True~~ False.

~~add A[0] in the root list of H.~~

~~if (min[H] == NULL) or (key[A[0]] < key[Min[H]])~~
~~False~~ True.

~~then min[H] = A[0] = 7.~~

$i = 1$ A[1] ≠ NULL True.

add A[1] in the root list of H.

if (min[H] == NULL) or (key[A[0]] < key[min[H]])
False False

so min[H] = 17.

i = 2 A[2] ≠ NULL True.

add A[2] in the root list of H.

if (min[H] == NULL) or (key[A[2]] < key[min[H]]),
False. False.

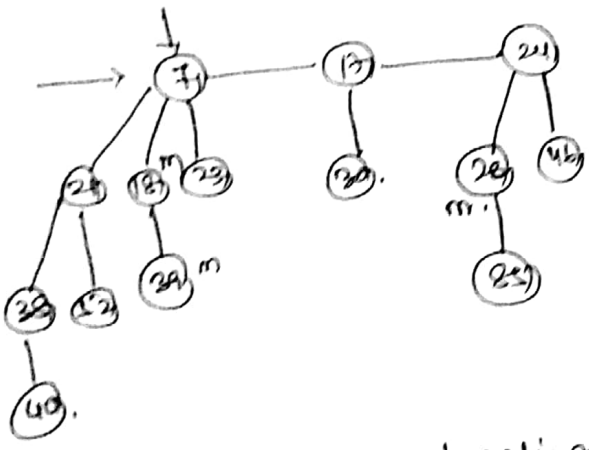
So min[H] = 17.

i = 3. (A[3] ≠ NULL) True

add A[3] in the root list of H.

if (min[H] == NULL) or (key[A[3]] < key[min[H]]).
False. True.

min[H] then min[H] = A[3] = 7.



The actual cost of extracting the minimum node can be accounted for as follows.

→ An O(D(n)) contribution comes from there being at most D(n) children of the minimum node that are processed in fib-heap-Extract-Min and from the work in line 1-2 and 14-19 of consolidate().

→ It remains to analyze the contribution from the for loop of line 3-13.

→ The size of the root list upon calling consolidate() is at most D(n) + t(H) - 1

- D(n): children of the extracted node.
- t(H): original root list node.
- -1: extracted root node.

→ Every time through the while loop of line 6-12, one of the roots is linked to another, and thus the total number of work performed in the for loop is at most proportional to $D(n) + t(H)$.

⇒ Hence the total actual work in extracting the minimum node is $O(D(n) + t(H))$

→ The potential before extracting the minimum node is $t(H) + 2m(H)$

→ The potential after extracting the minimum node is at most $[(D(n)+1) + 2m(H)]$. Since at most $D(n)+1$ roots remain and no nodes become marked during the operation.

→ The amortized cost is thus at most.

$$\text{Actual cost} + \Delta\Phi(H)$$

$$= O(D(n) + t(H)) + ((D(n)+1) + 2m(H)) - (t(H) + 2m(H))$$

$$= O(D(n)) + O(t(H)) + (D(n)+1) + 2m(H) - t(H) - 2m(H)$$

$$= O(D(n)) + O(t(H)) - t(H)$$

$$= O(D(n)) \qquad [\because D(n) = \log n]$$

$$= O(\lg n).$$

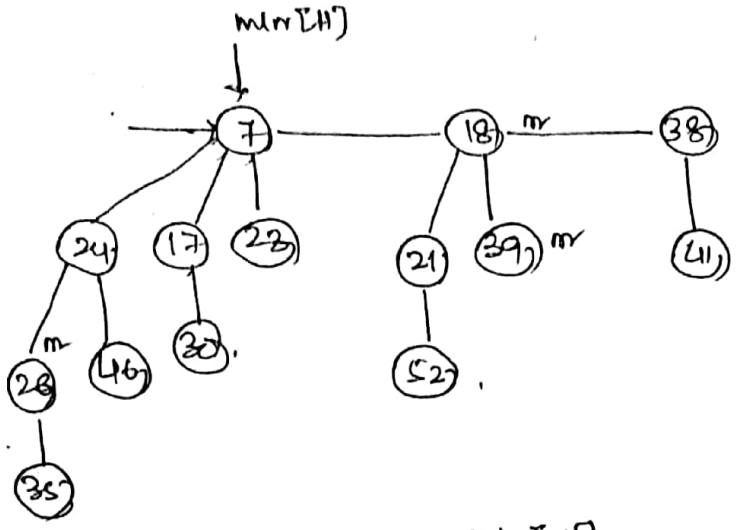
Decreasing key and deleting node

Decreasing a key:

```
fib-heap-decrease-key (H, x, k)
1. If k > key[x]
2.    then error "the new key is greater than current key"
3. key[x] ← k
4. y ← p[x]
5. If y ≠ NULL and key[x] < key[y]
6.       then cut (H, x, y)
7.            cascading-cut (H, y)
8. If key[x] < key [min[H]]
9.       then min[H] ← x
```

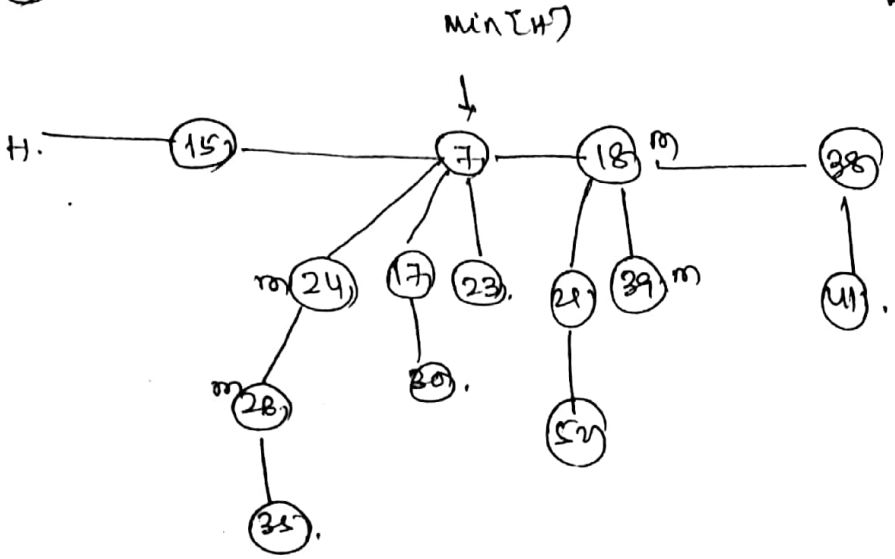
```
cut (H, x, y)
1. Remove x from the child list of y, and
   decrementing degree[y]
2. add x to the root list of H.
3. p[x] = NULL
4. mark[x] = false.
```

```
Cascading-cut (H, y).
1. z ← p[y]
2. If z ≠ NULL
3.    then If (mark[y] == false)
4.              then mark[y] = true.
5.              else cut (H, y, z)
6.                   cascading-cut (H, z)
```

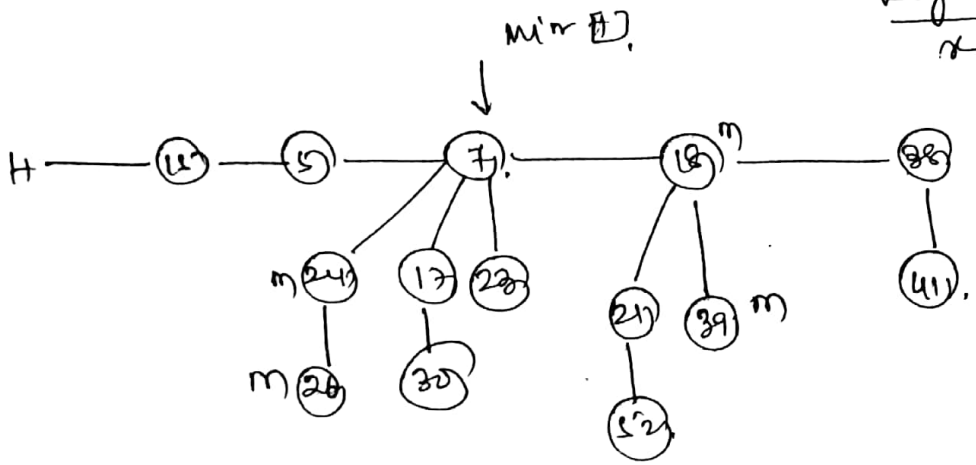


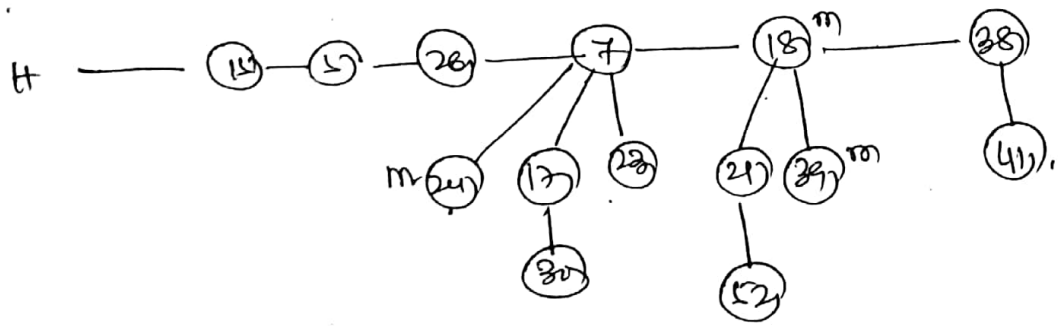
← the initial fibonacci heap.

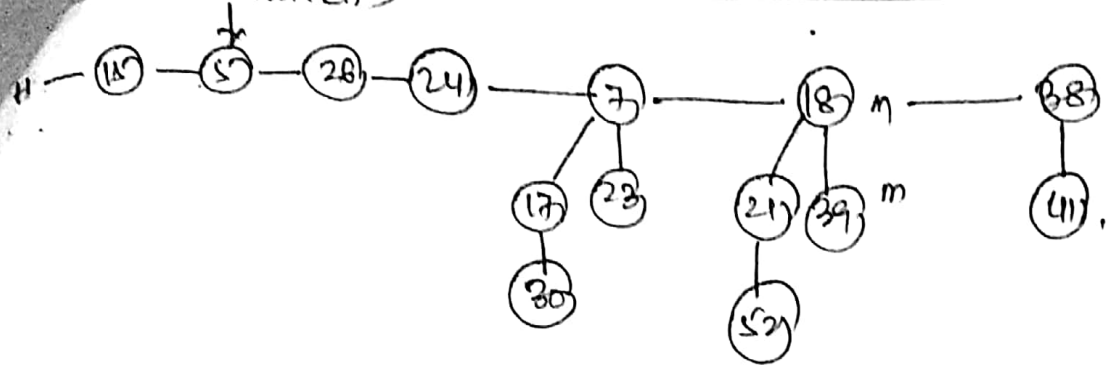
now decrease the key 46 with new key 15
x k.



now decrease the key -35 with new key 5
x k







The amortized cost of fib-heap-decrease-key is only O(1).

How to determining the actual cost?

The actual cost of fib-heap-decrease-key is

the · fib-heap-decrease-key procedure takes O(1)

+

time to perform the cascading cuts.

Suppose the cascading cut take c times recursively

then the actual cost of fib-heap-decrease-key include

all recursive call. i.e O(c).

now compute the potential cost.

→ Let 'H' denote the Fibonacci Heap just prior to the fib-heap-decrease-key operation.

→ Each recursive call of cascading-cut (except for last one), cut a marked node and clear the mark bit.

→ Then there are t(H)+c trees [the original tree(H), c-1 trees produced by cascading cuts, and the tree rooted at x] and at most m(H)−c+2 marked nodes (c-1 were unmarked by cascading cuts and the last call of cascading-cut may have marked a node.)

The change in potential cost. is there for.

$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))$

$= 4-c.$

Thus the amortized cost of fib-Heap-decrease-key is at most.

$O(c)+4-c = O(1).$

Deleting a node.

Fib-Heap-delete(H, x)

1. fib-heap-decrease-key(H, x, $-\infty$)
2. fib-heap-Extract-Min(H).

Since fib-heap-Extract-Min(H) take $\lg n$ time, the amortized time required for Fib-Heap-Delete is $O(\lg n)$.

Trie

A trie (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching. The main application for tries is in information retrieval. The name "trie" comes from the word "retrieval".

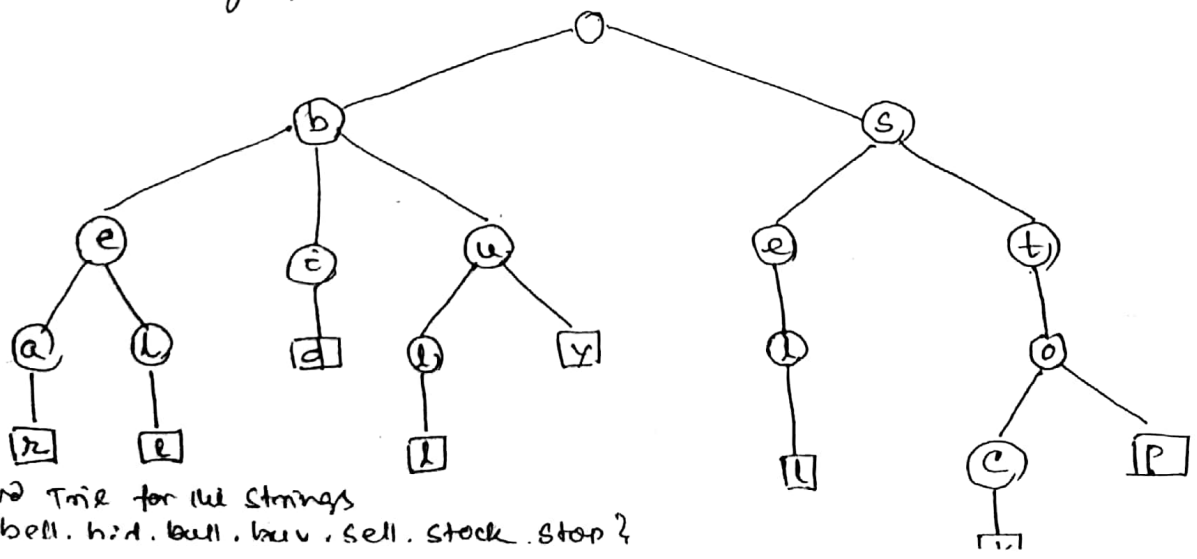
Defn. (Standard Tries)

Let S be a set of s strings from alphabet Σ, such that no string in S is a prefix of another string. A standard trie for S is an ordered tree T with the following properties.

→ Each node of T, except the root is labelled with a character of Σ.

→ The ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet Σ.

→ T has s external nodes, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to an external node v of T yields the string of S associated with v.



Standard Trie for the strings
{bear, bell, bid, bull, buy, sell, stock, stop}

Skip list

→ A skip list S for dictionary D contains a series of lists $\{S_0, S_1, \ldots, S_h\}$.

→ Each list S_i stores a subset of the items of D sorted by a nondecreasing key plus items with two special keys, denoted $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in D and $+\infty$ is larger than every possible key that can be inserted in D. In addition, the lists in S satisfy the following:

⇒ List S_0 contains every item of dictionary D (plus the special items with keys $-\infty$ to $+\infty$).

⇒ For $i = 1, \ldots, h-1$, list S_i contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the items in list S_{i-1}

⇒ list S_h contains only $-\infty$ and $+\infty$

An example of a skip list is shown in figure given below. It is customary to visualize a skip list S with list S_0 at the bottom and lists $S_1 \cdots S_h$ above it. Also we refer to h as the height of skip list S.