Unit-②

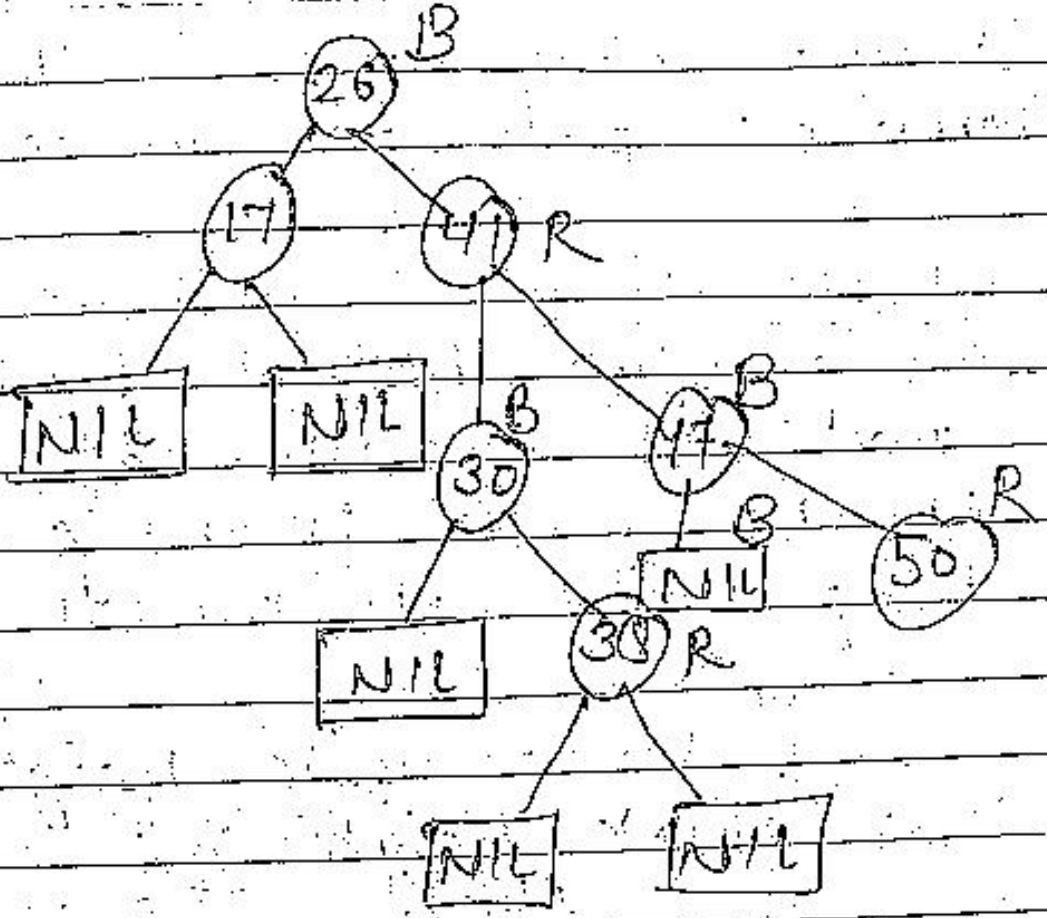
Advanced data structure

Red-black tree: ⇒ Binary tree with extra attributes, its node color is red & black is as Red-black tree. R-B tree inherit all the attribute from binary search tree.

attribute: key, left, right, parent.

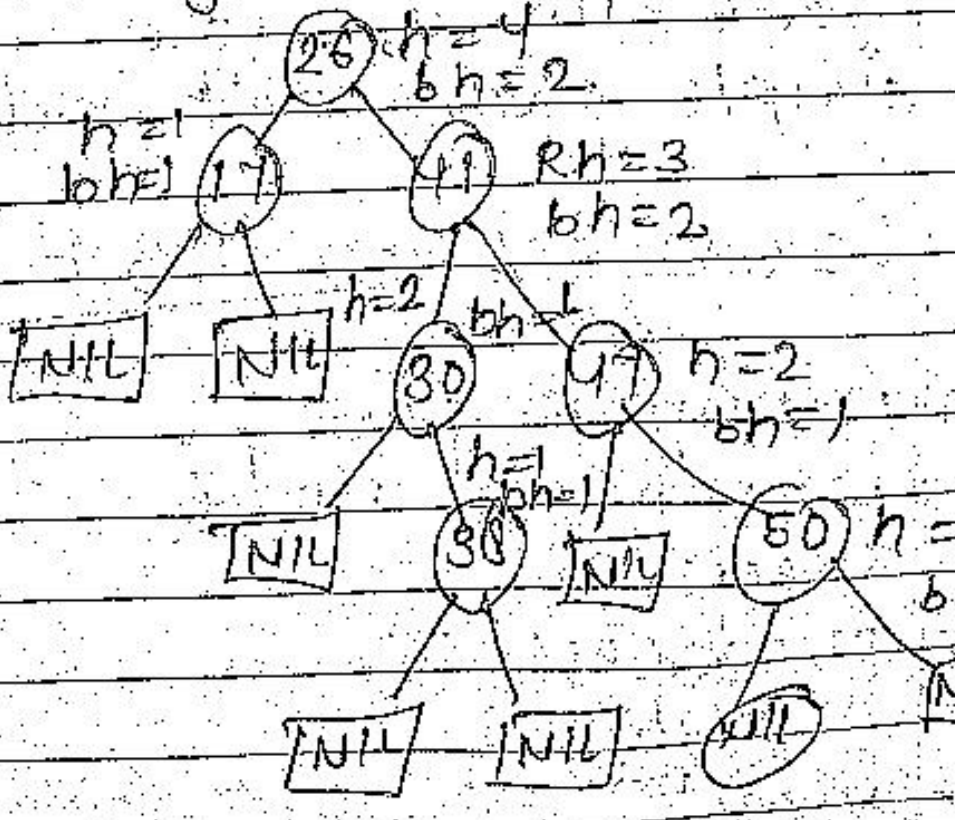
Properties of R-B tree: ⇒

1. Every node is either black or red.
2. root is always black
3. Every leaf node is black
4. If a node is red then both of its children is black.
5. No two red node in a path (leaf can exist).
6. each node from the every path, root to leaf node will contain same no. of black node



color[NIL[T]] = BLACK.

★ Black height of a node :⇒



no. of edges

Height of a node is the no. of edges in the longest path to the leaf

black height :- is the no. of black height of bh(x) node is the no. of leaf on the path from x to leaf node. No counting at x, any node with height has black height (bh) $\geq \frac{h}{2}$

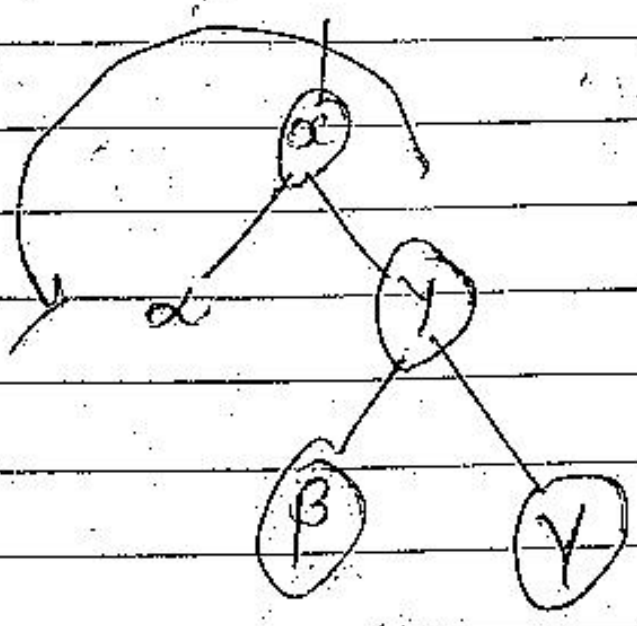
because the property of RB tree (if the node is red both of children is black & hence too no red node in a row on a simple path can't exist).

Operations on RB tree :⇒

→ minimum find
→ maximum node
→ successor
→ predecessor
→ search
→ insert
→ delete

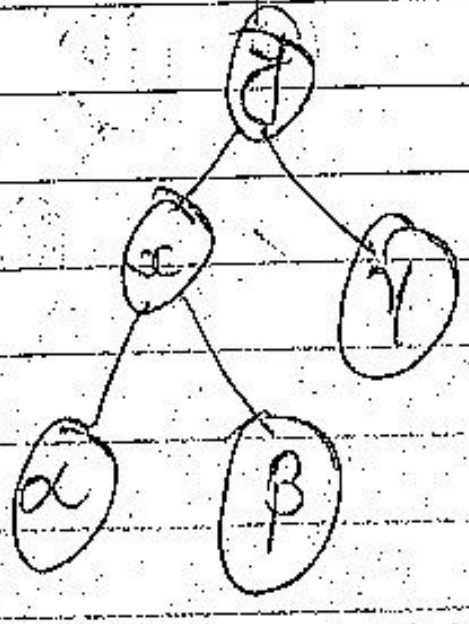
Rotation :- ⇒ To maintain the property of R-B tree & B-S tree.

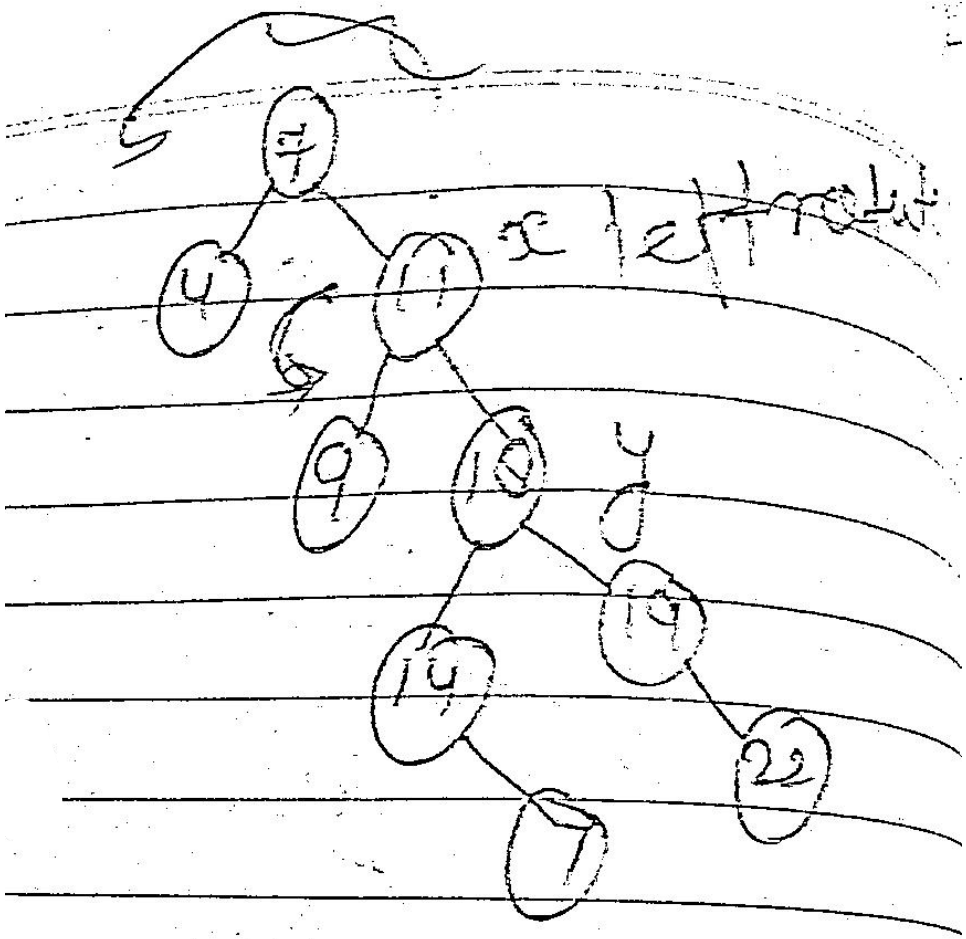
left rotation :- ⇒

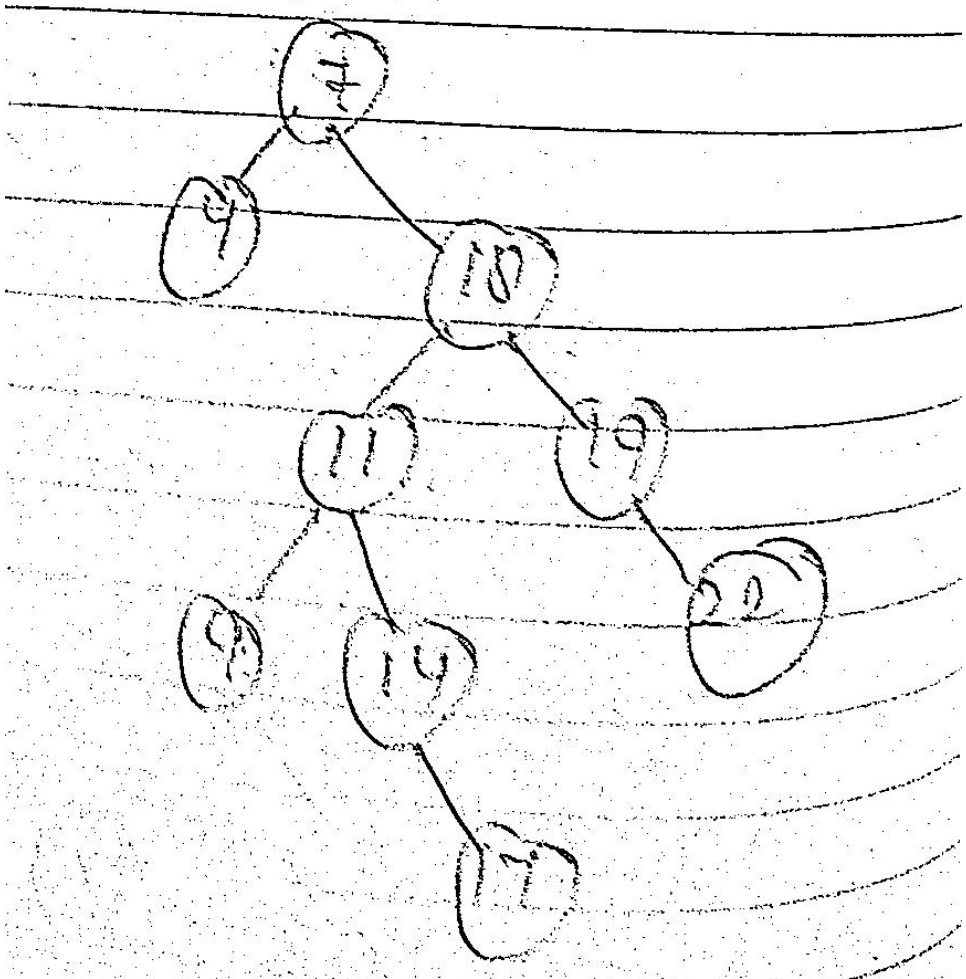


(T, x)

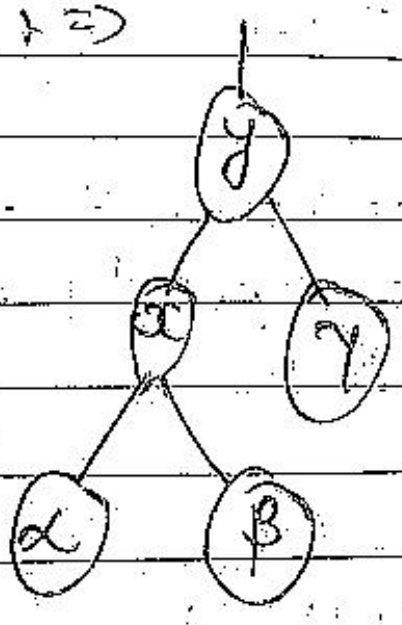
(left rotate)



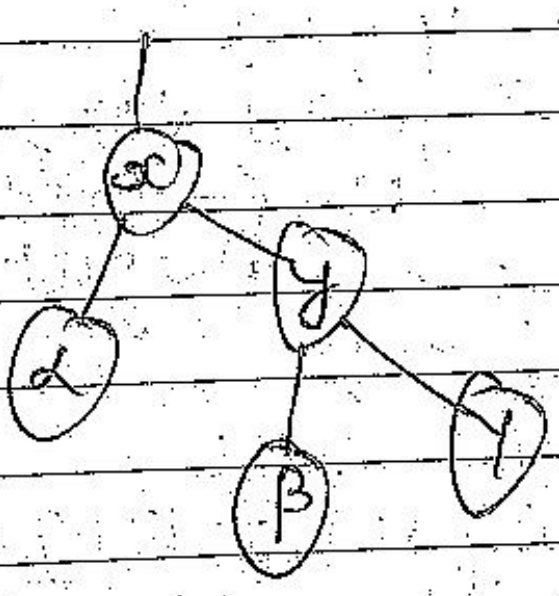

7
4
11
9
10
19
14
22
7


7
4
9
10
11
19
22
9
14
7

→ Right :=>



(T, y)

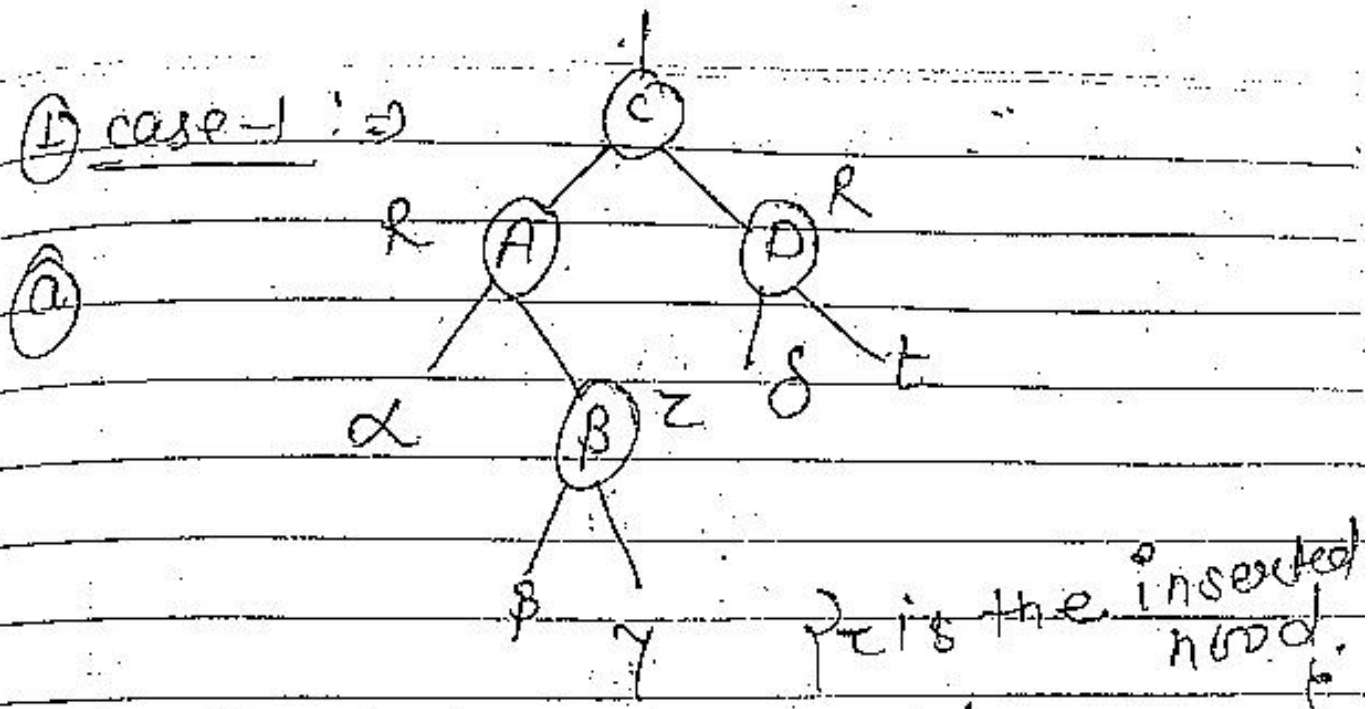


★ RB-INSERT :=

(1) color of new node = RED
(2) Maintain property of
(3) Restore the R-B

```
(1)  y ← NIL
(2)  x ← root[T]
(3)  while x ≠ NIL
(4)  do y ← x
(5)  if key[z] < key[x]
(6)  then x ← left[x]
(7)  else x ← right[x]
(8)  p[z] ← y
(9)  if y = NIL
(10) then root[T] ← z
(11) else if key[z] < key[y]
(12)      then left[y] ← z
(13)      else right[y] ← z
(14) left[z] ← NIL
(15) right[z] ← NIL
(16) color[z] ← RED
(17) RB-INSERT-FIXUP(T,z)
```

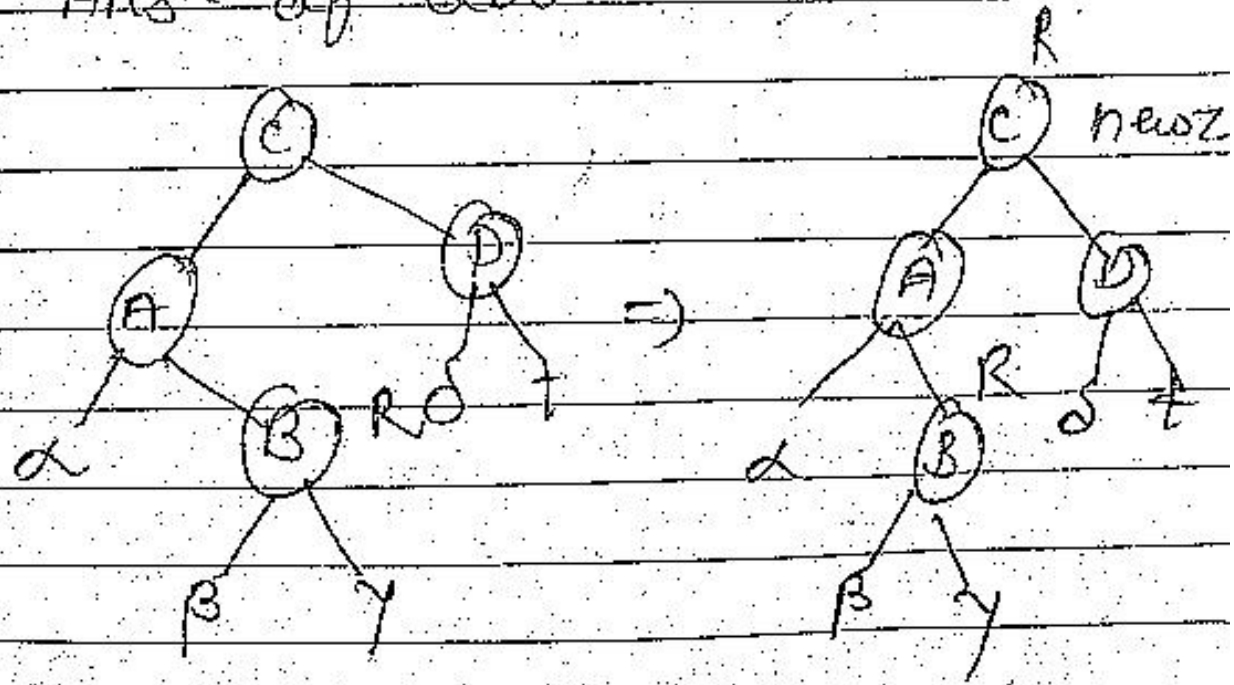
(1) case-1 :⇒



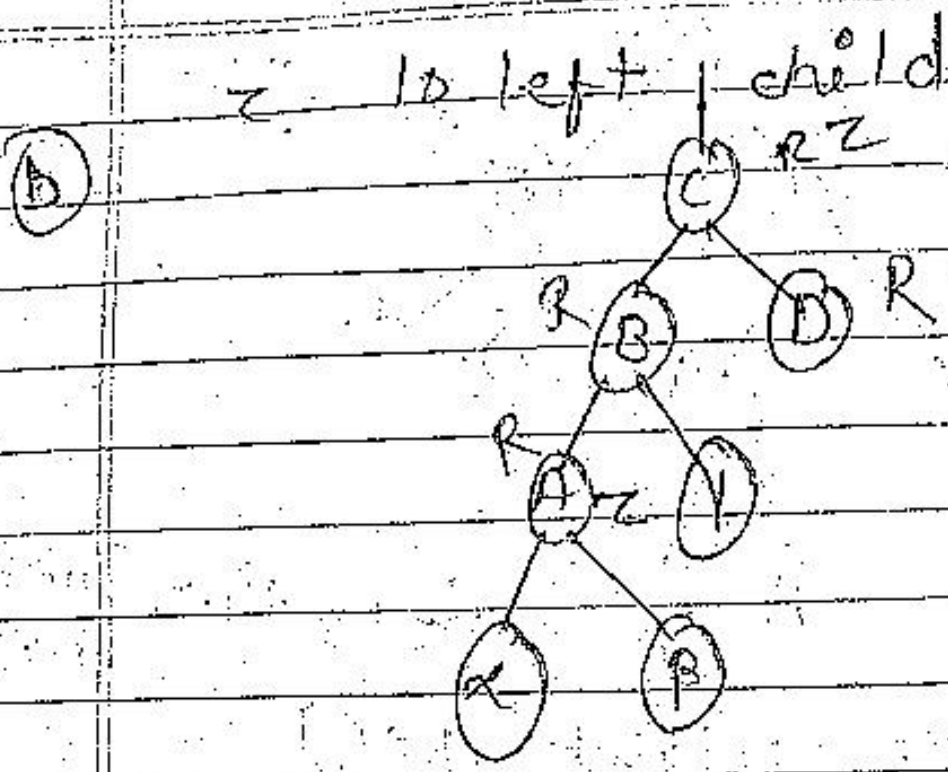
z is the inserted node.

z is the right child
p[p[z]] must be
Black & z & P[z] are red
color P[z] Black
color y black
color p[p[z]] = Red
make z = P[P[z]]

Ans - of above two



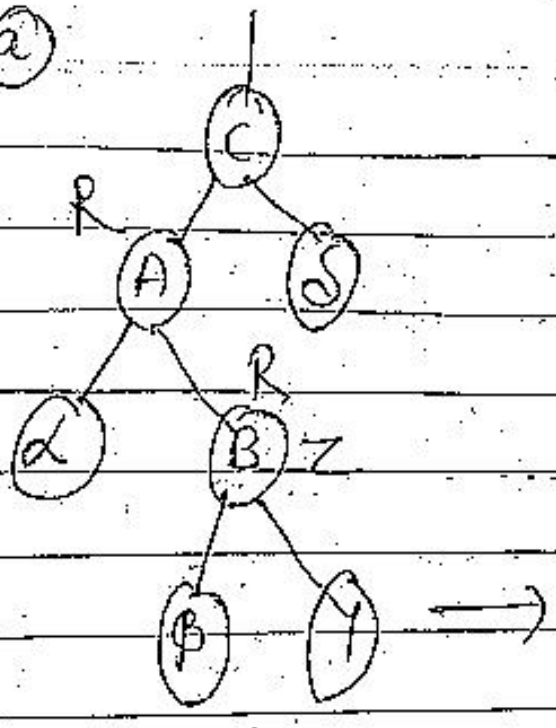
z is left child

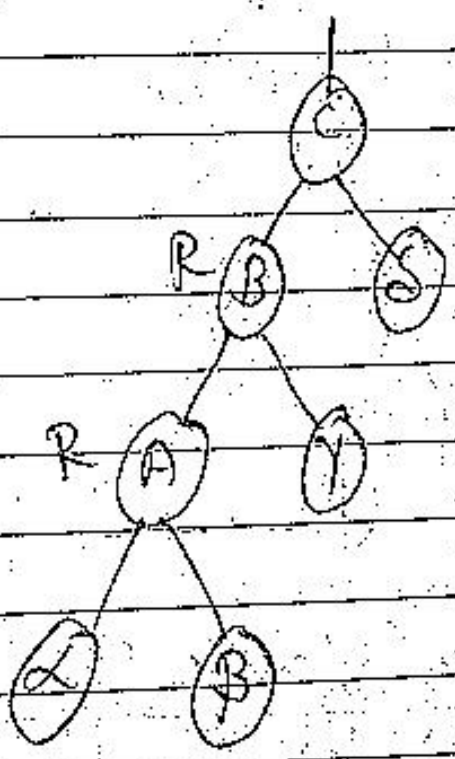


z is left child
p[p[z]] must be black
& z & p[z]
are red
color p[z] = Black
color of y = black
color p[p[z]] = red
make z = p[p[z]]

(a)

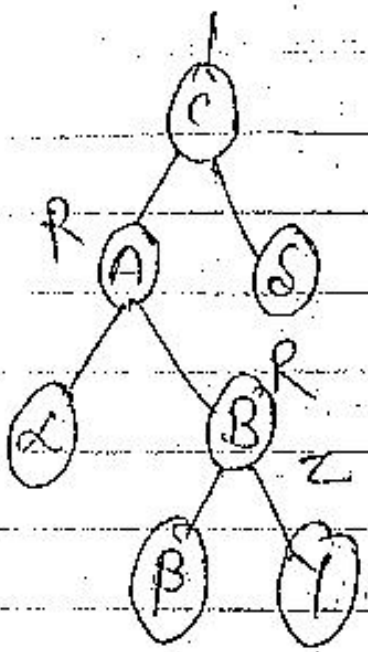
case-2 :-)

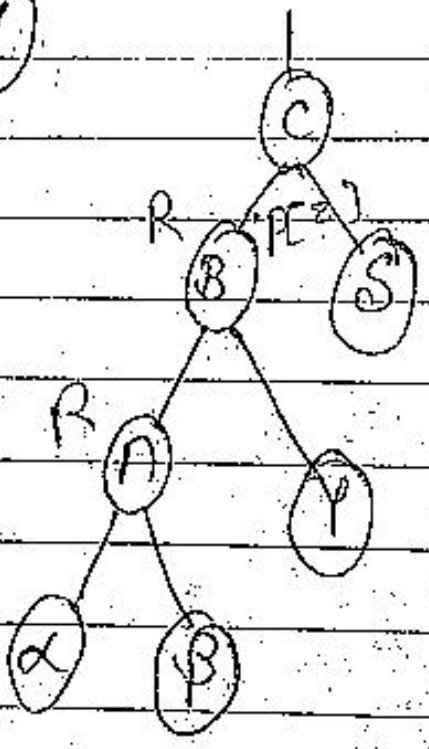




z uncle is black] →
z right child
z ← p[z]
left ROTATE (T, z)
case 3 called

(b)

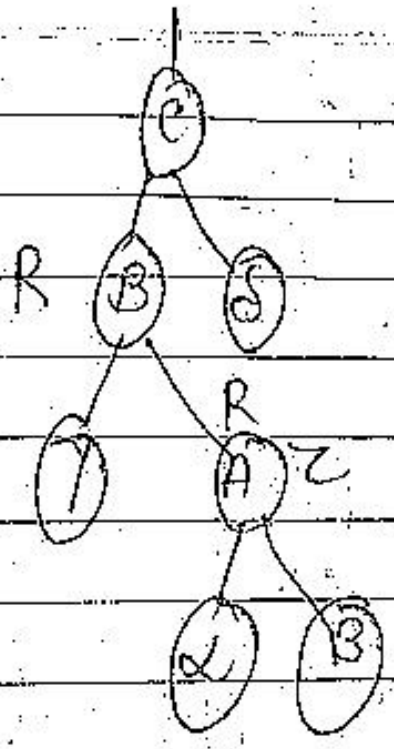


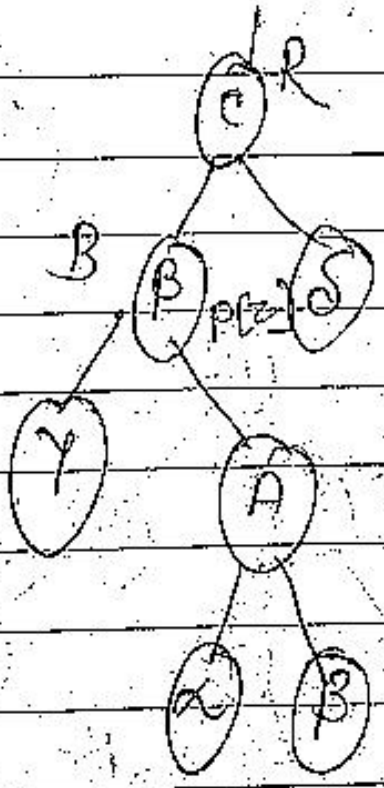


z uncle is blacks
& z left child →

[z ← p[z]
Right ROTATE (T, z)
case 3 called]

case-3 :-



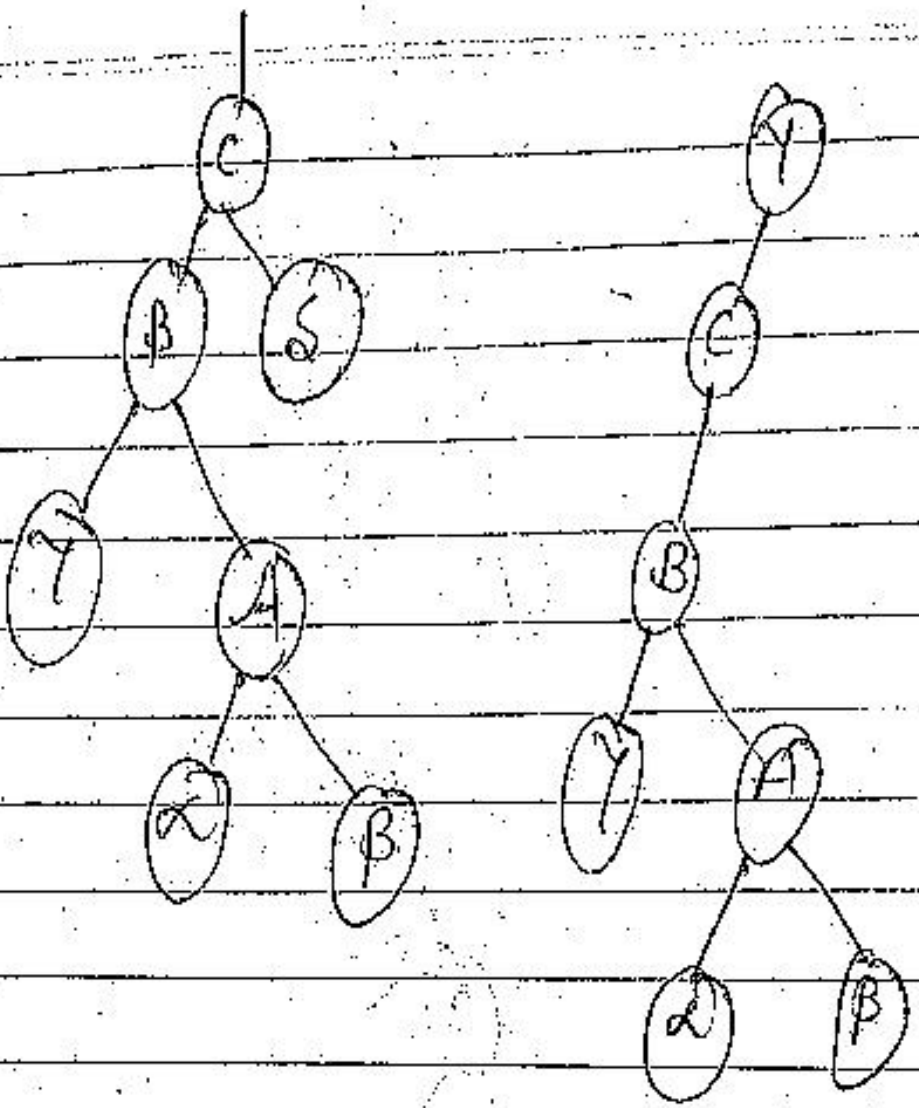


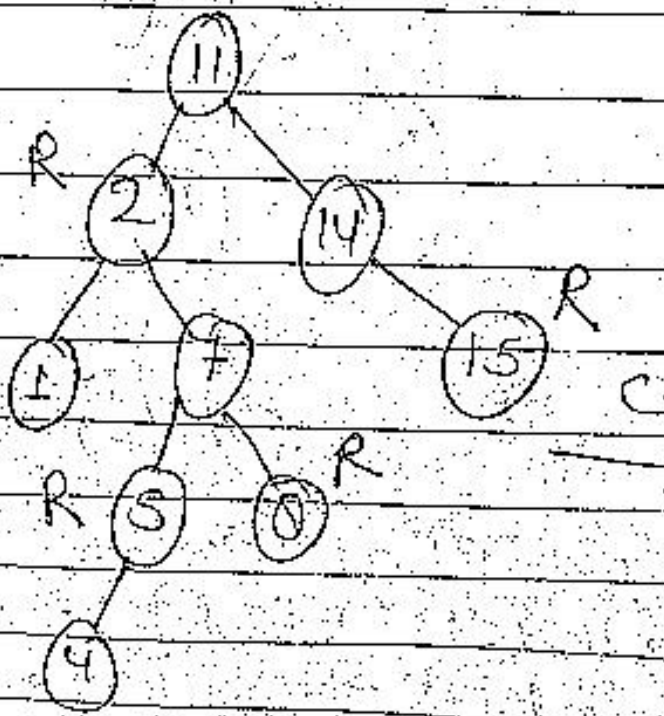
color p[z] ← Black

color p[p[z]] ← red

left Rotate [T, p[p[z]]]

p[z] is now black





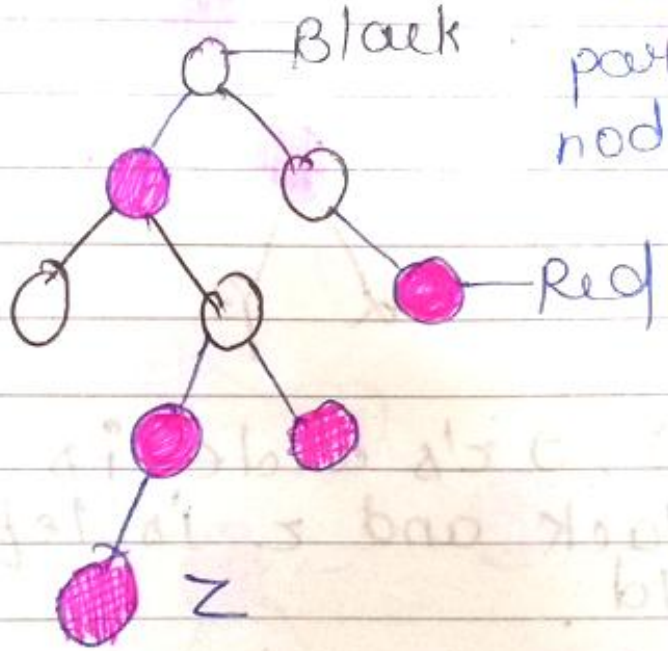
case 1

R-B tree Insertion :→

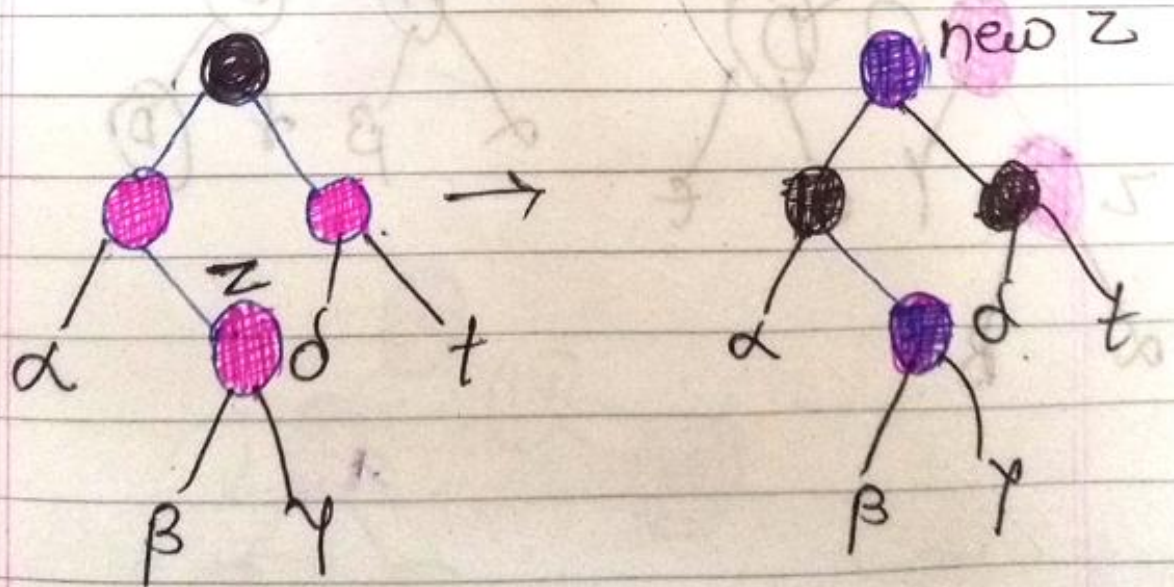
1. z is root } when ever we insert a new node then it will be color of red }

Oz

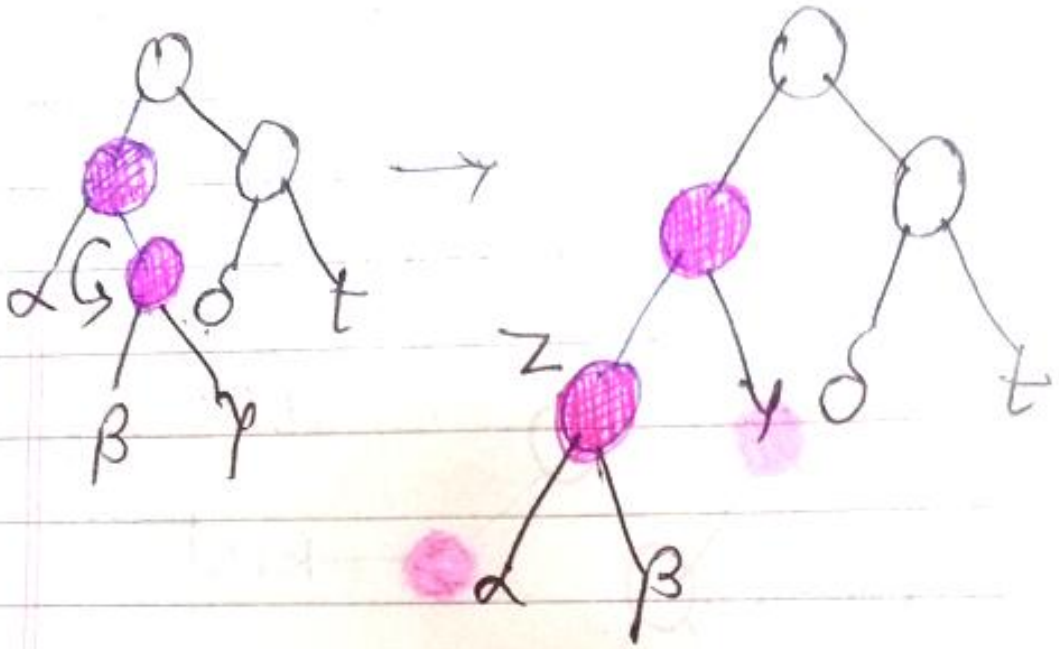
2. parent of node z is red



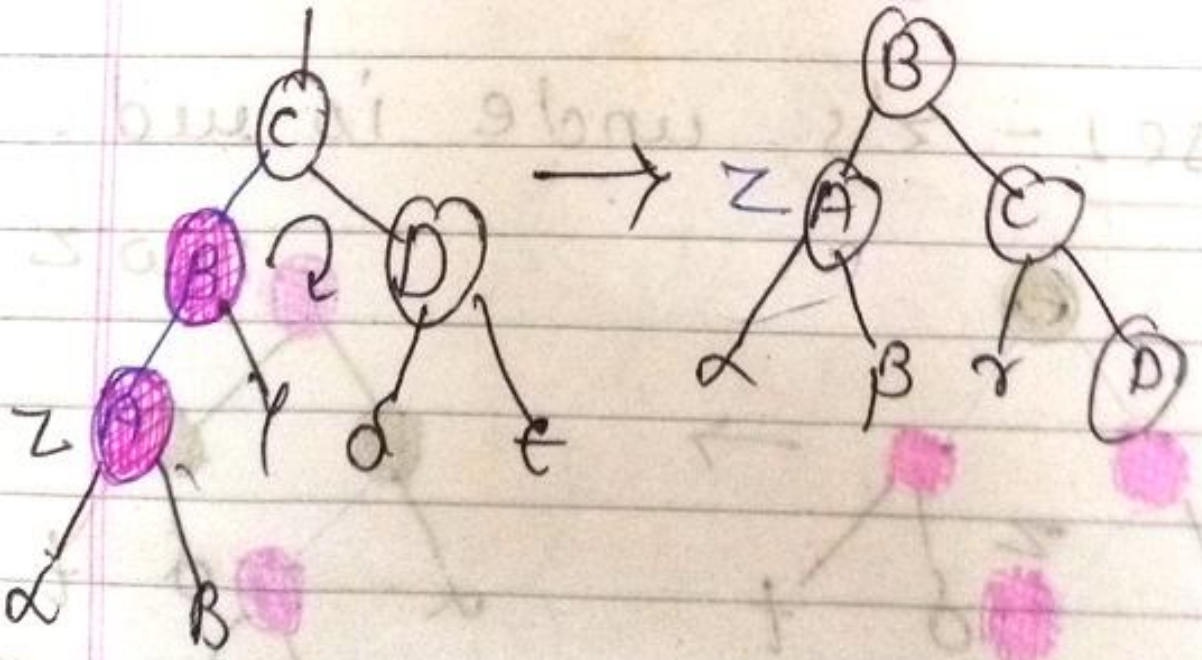
case 1 - z's unde is red.



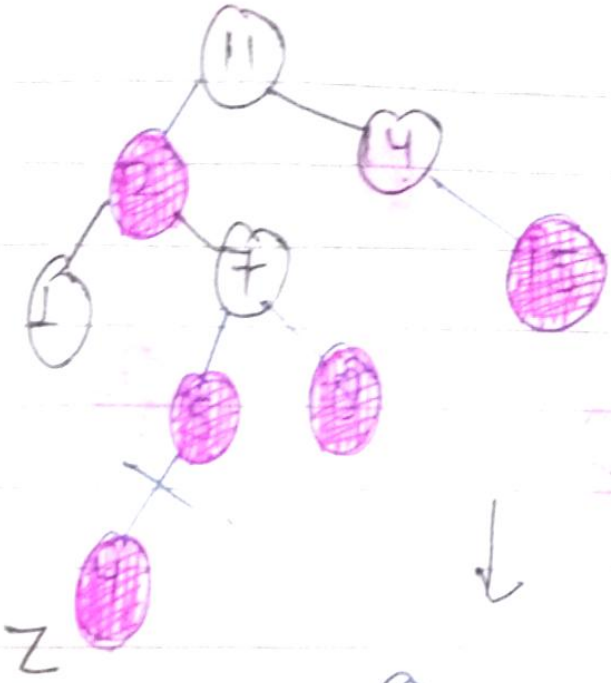
case-2:- z's uncle is black and z is a right child.



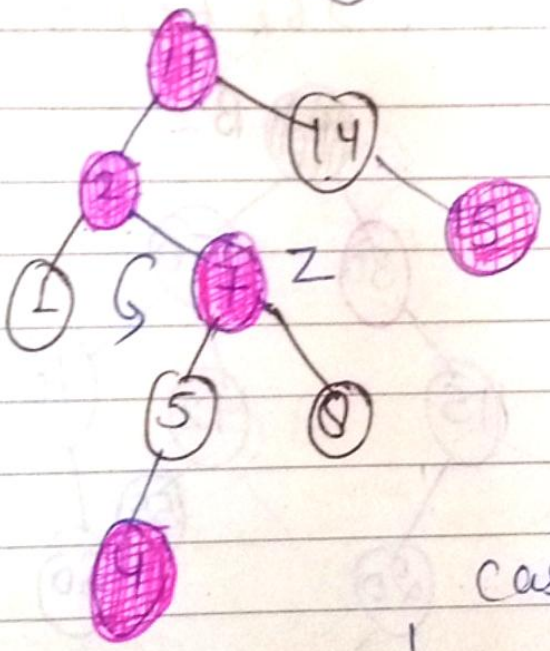
case-(3):⇒ z's uncle is black and z is left child.



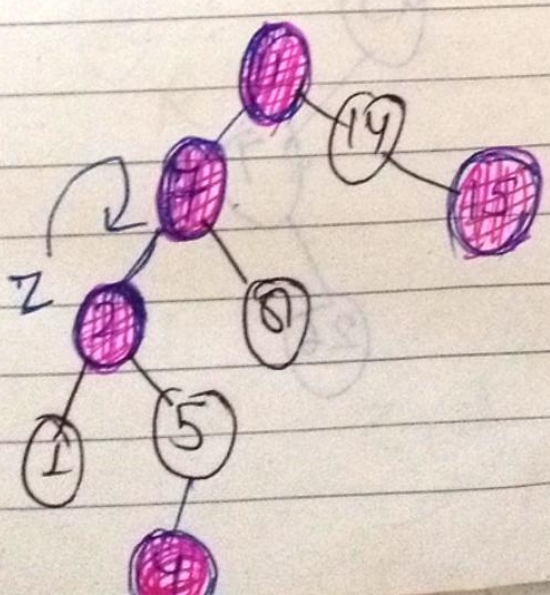
eg :-)



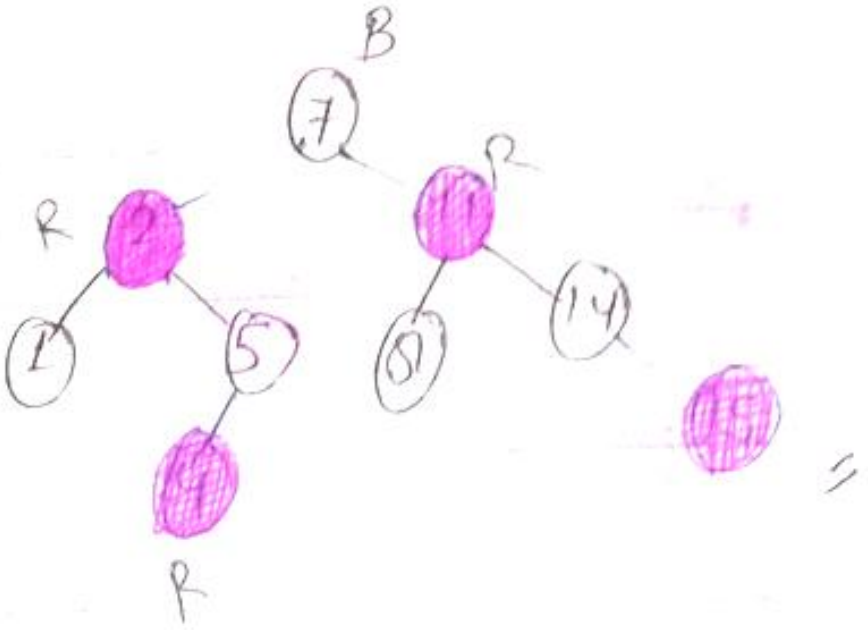
case 1



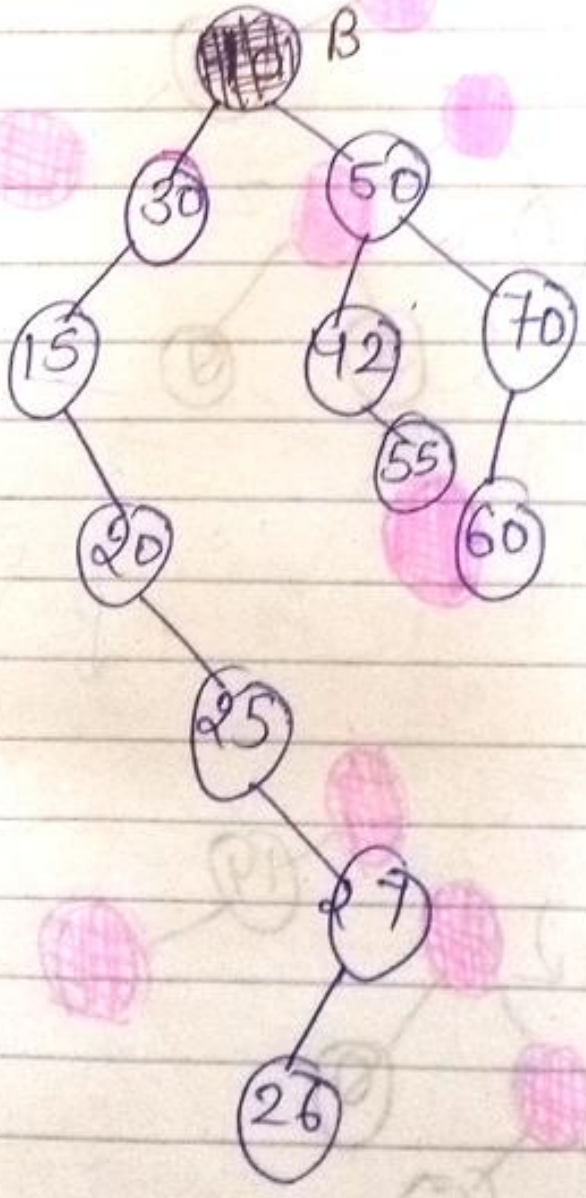
case 2



case 3



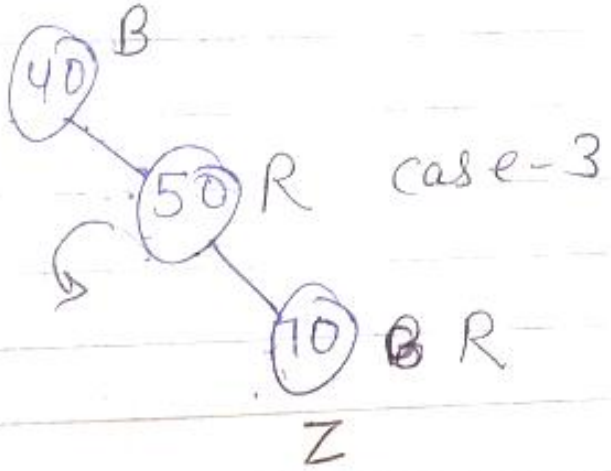
(2) 40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 60, 55

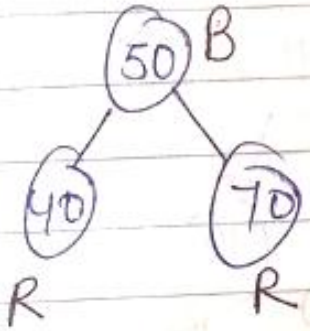


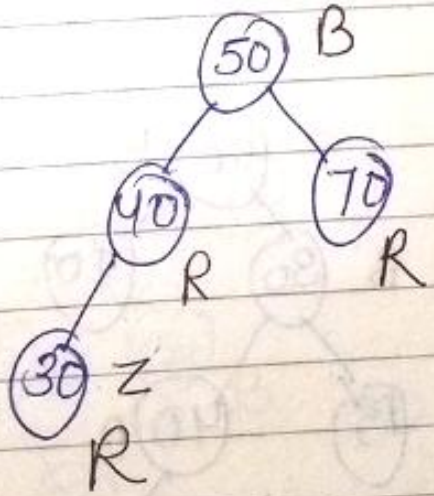
RB-INSERTION

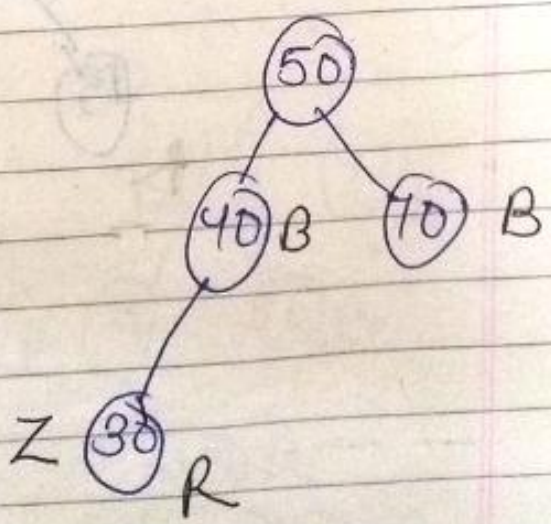
40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 60, 55

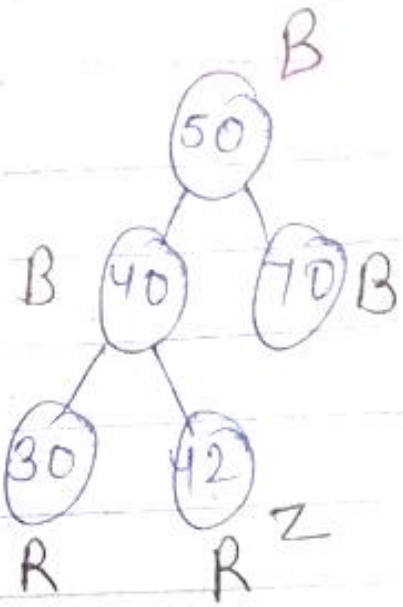
(1) insert 40 → 

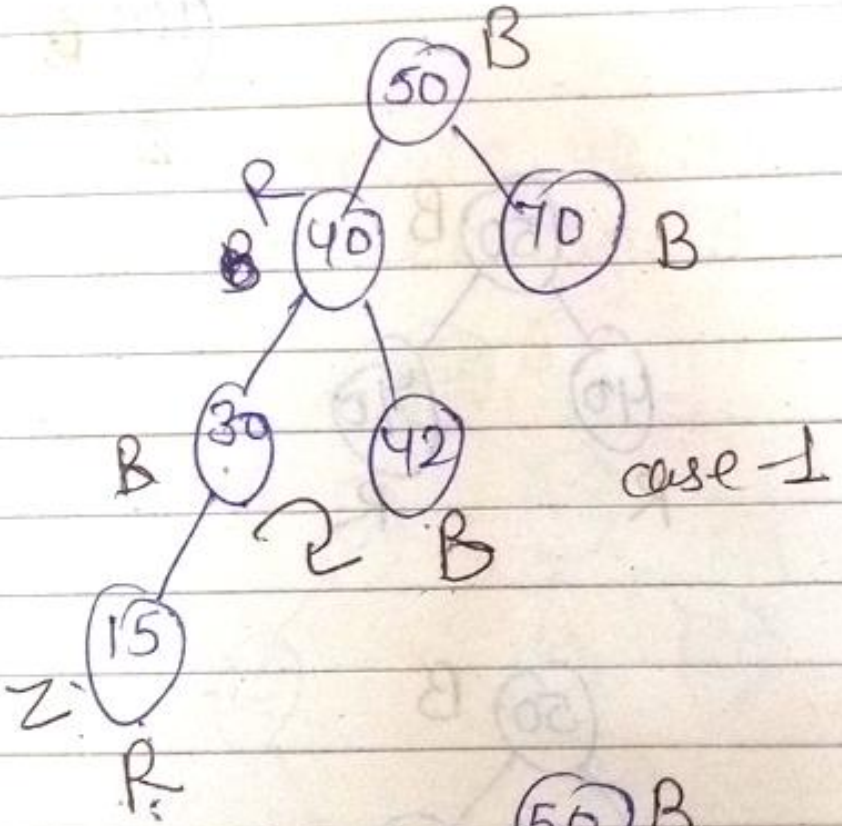


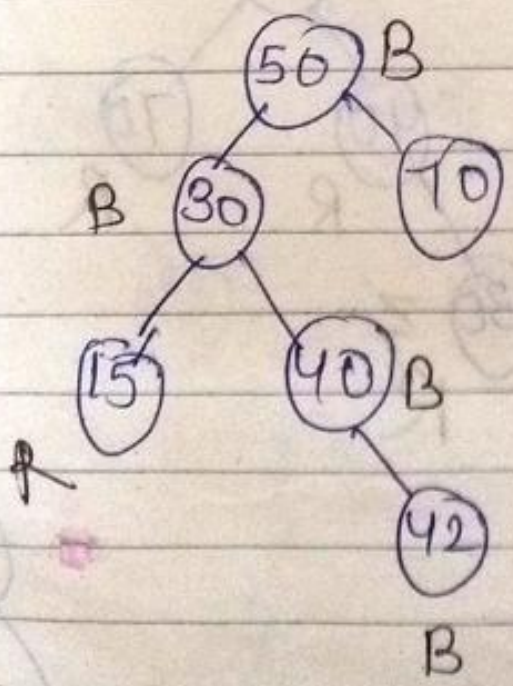







B
50
B
40
70
B
30
42
R
R


B
50
R
B
40
70
B
B
30
42
case-1
B
15
R


50
B
B
30
70
15
40
B
R
42
B

Deletion R-B tree

↳ Top down deletion

(1) Examine the root node if both of children are black then
(a) make the root node red
(b) move x to the appropriate child of the ~~node~~ root
(c) procced the step-(2)
(d) otherwise designate the root as an x & procced to step 2(b).

(2) traverse down the tree, we continuously until we reach the node to be deleted.

Now x is black, P is red, T is black

color x = RED

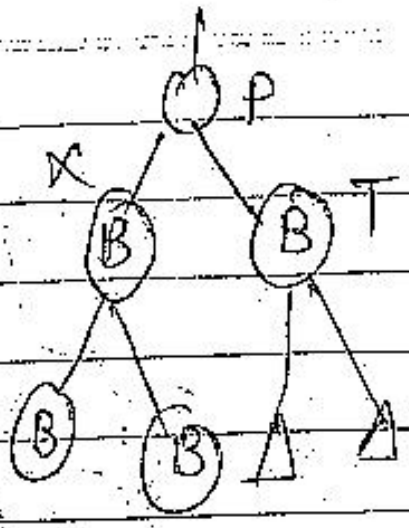
the recolor other node &

do rotation

2A

2B

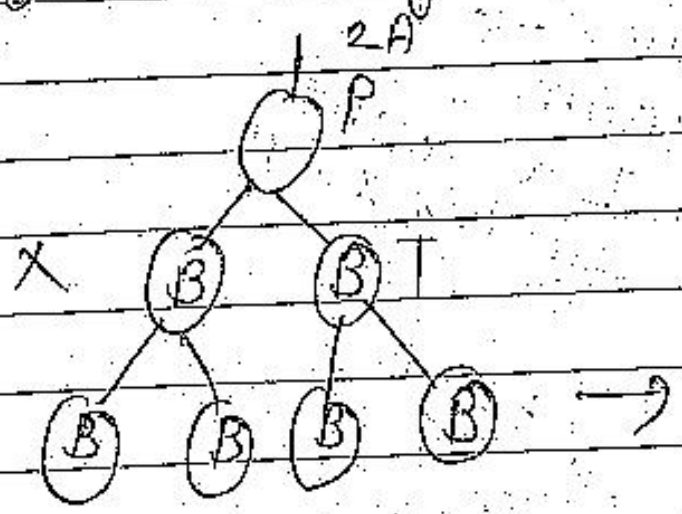
(a) x has two black child

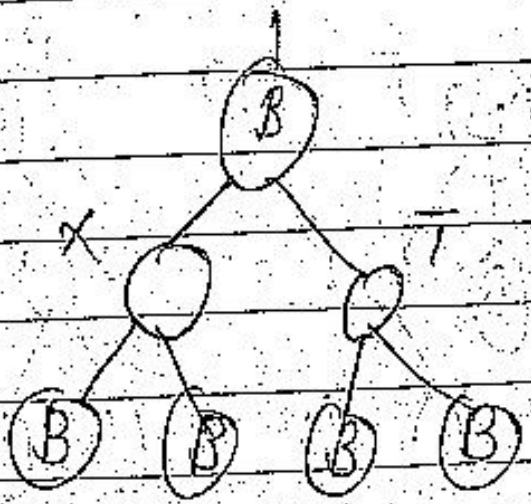


A_1 : Has a Black child

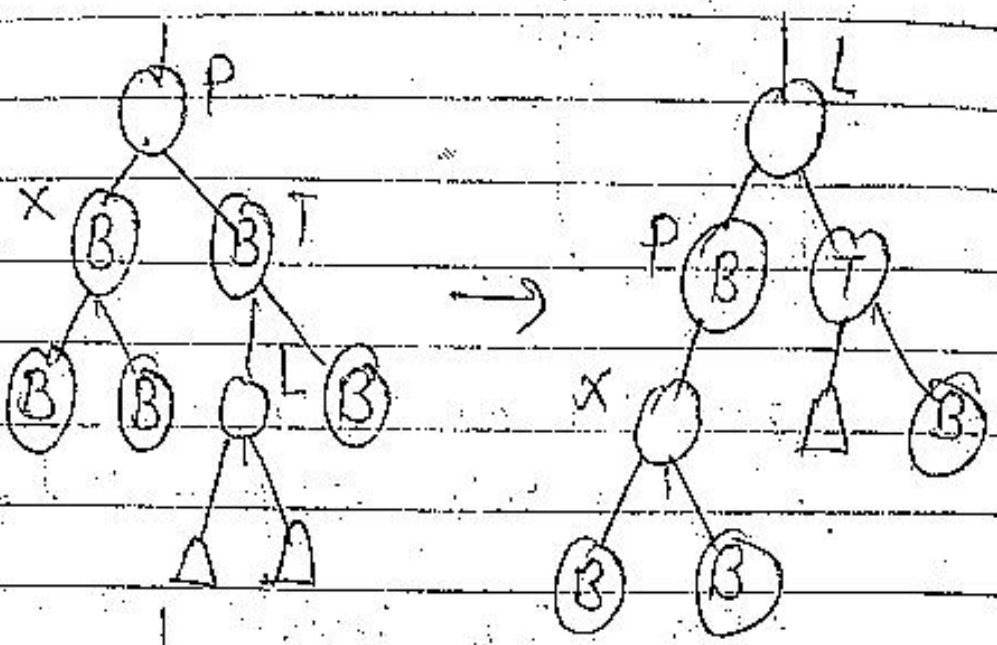
A_2 : T's left child red

A_3 : T's right child red





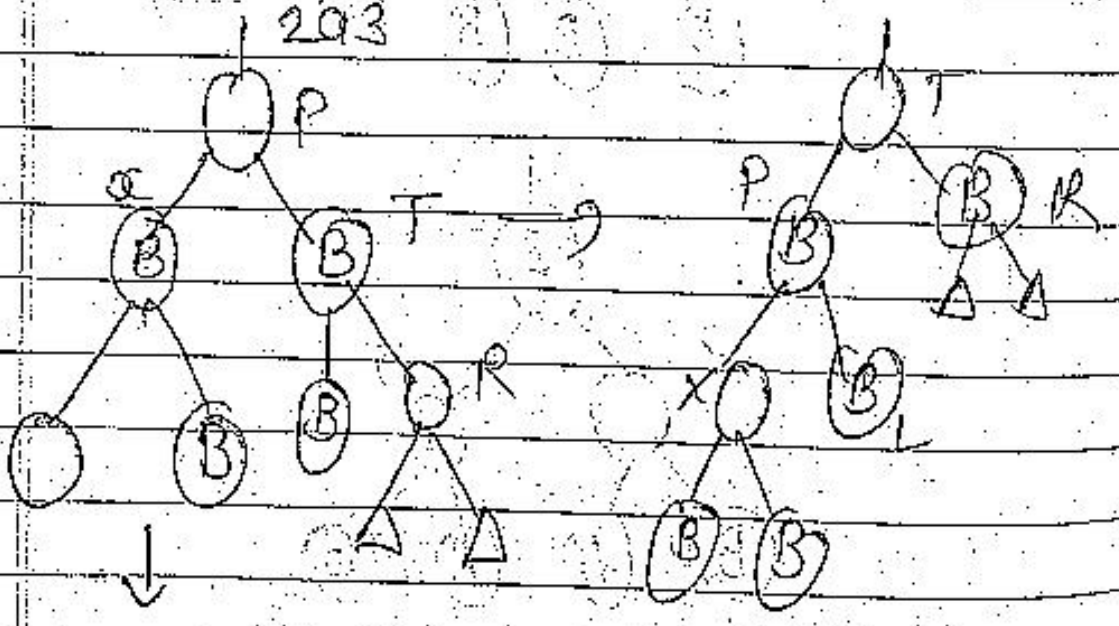
case 2



↓

rotate L around T then L around P swap x & P.

case 3



↓

rotate T around P swap P & T & continue down

case-2 B ⇒

x has atleast one Red child

if the new x is Red, continue down again

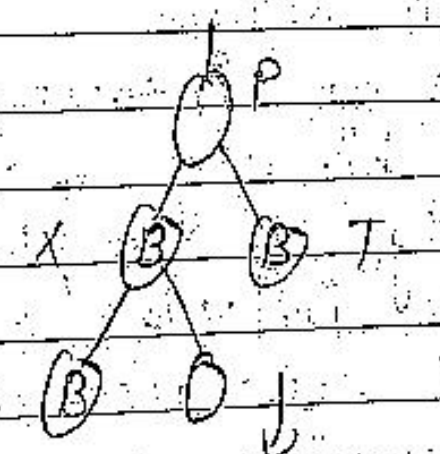
→ if the new x is red, continue down again.

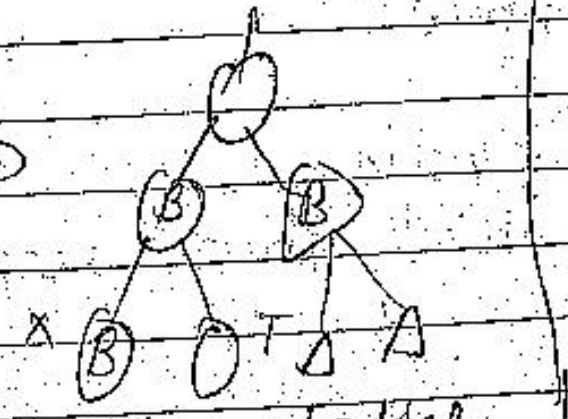
→ if the new x is black (T is Red, P is black)

rotate T around P

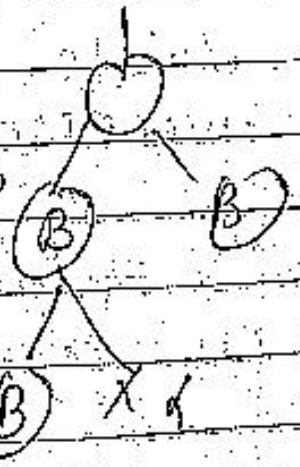
recolor P & T

→ Back to main case-2





if we have to the black child rotate T around P, recolor P & T Back to step 2



if move to red child move down again

case-(3) - Now you will find the node as a leaf node.

case-(4) :- Color the root as black.

(1) Deletion :-

Delete as we delete from binary search tree.

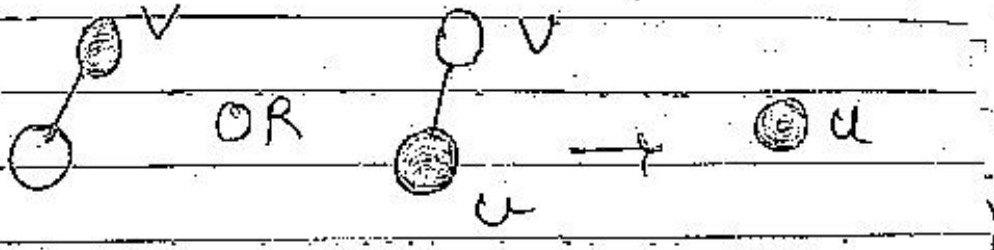
→ End up deleting the node which is either a leaf or has one child.

→ We delete an internal node from a binary search tree simply by replacing it by its inorder successor & then we recursively call delete operation on inorder successor node.

→ v → deleted node

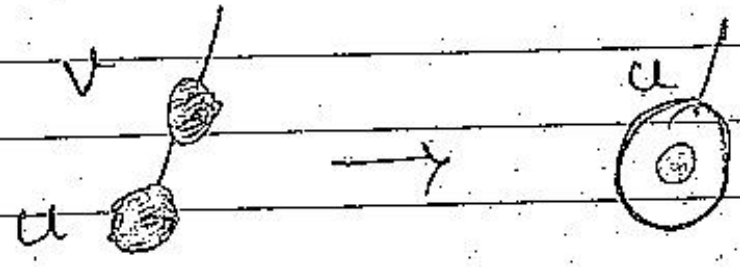
u → the child that replaces v

either 'u' or 'v' is red



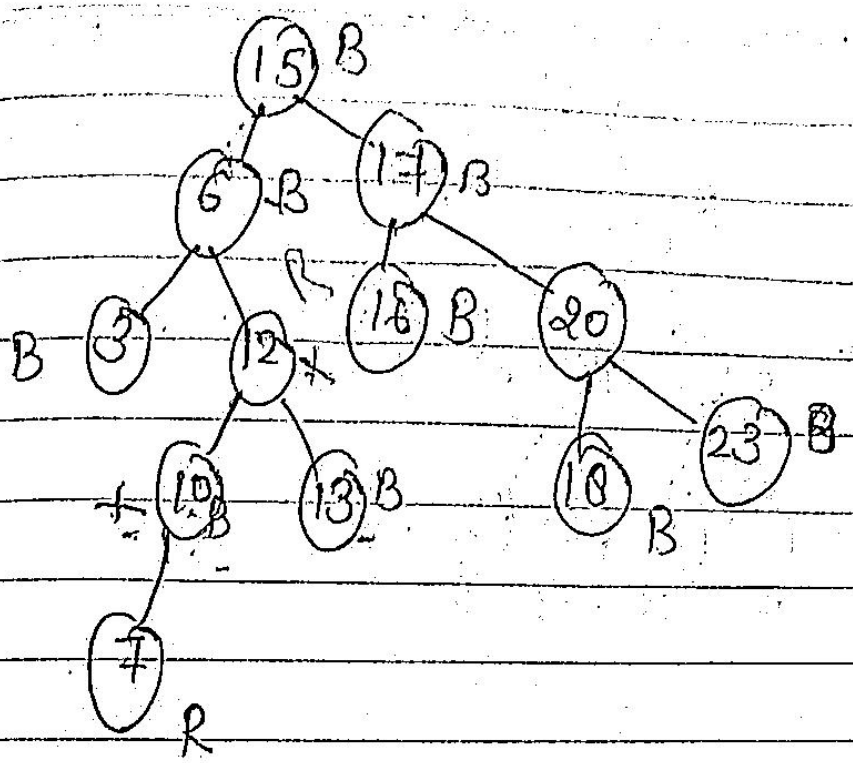
→ replace v by u, u will be a black node

→ Both u & v are black

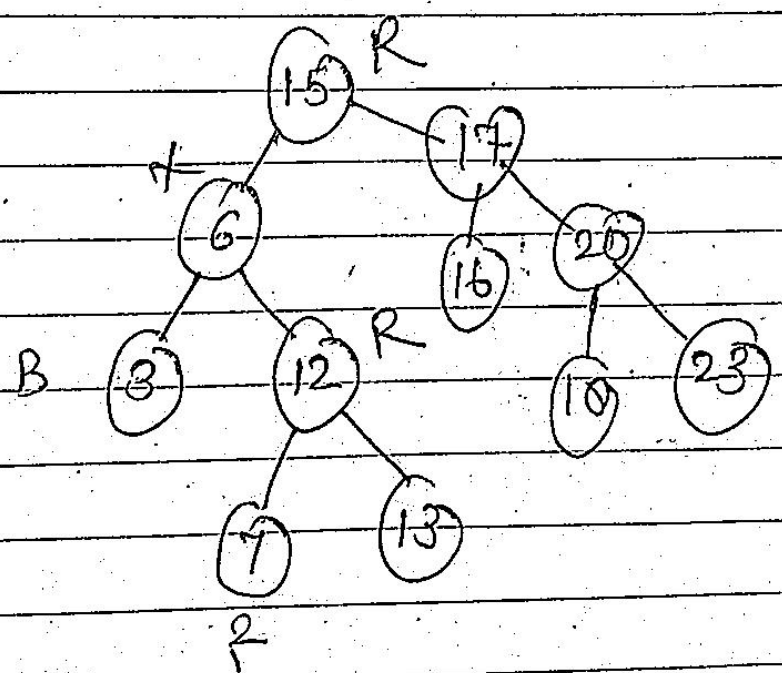


→ if node u is root, make it single black

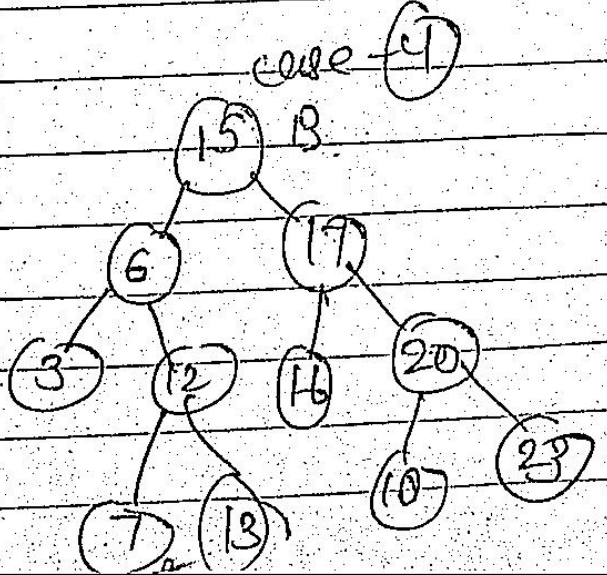
Que:

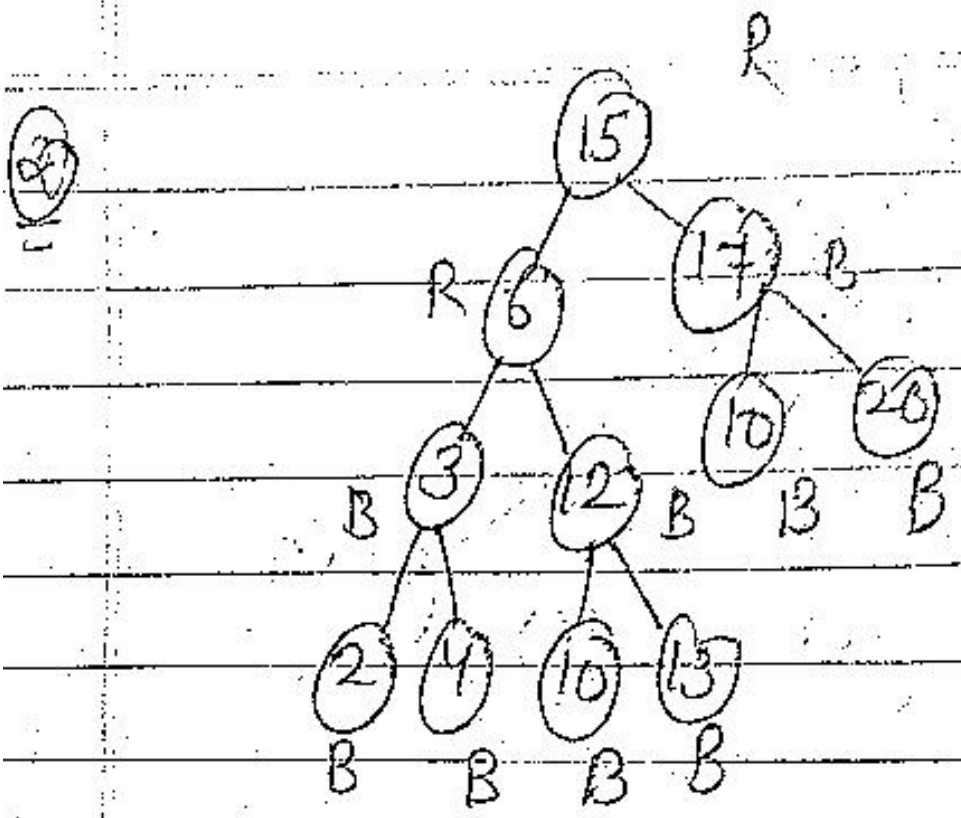


Delete 10



Case 4





(0,1) (1,2)

d = 2

0 0 X 1 ✓ 2

B Tree

→ B tree is the m way tree (a tree where each node may have upto m children max - 5 children)

→ Height of tree must be minimum

(1) tree subtree shouldn't empty

(2) ~~leaved~~ leaf nodes of the tree should be at same level.

(3) All nodes except leaf should have atleast few children.

(4) If the B-tree of order m then root has two child and atmost m child.

(5) no of keys in each internal node is one less then the no of child node.

every node may contain max. $q-1$ keys. & every node other then root atleast $(t-1)$ key.

B tree :) B - Tree is special case of m - way search tree

→ B - Tree of order m satisfies the following properties

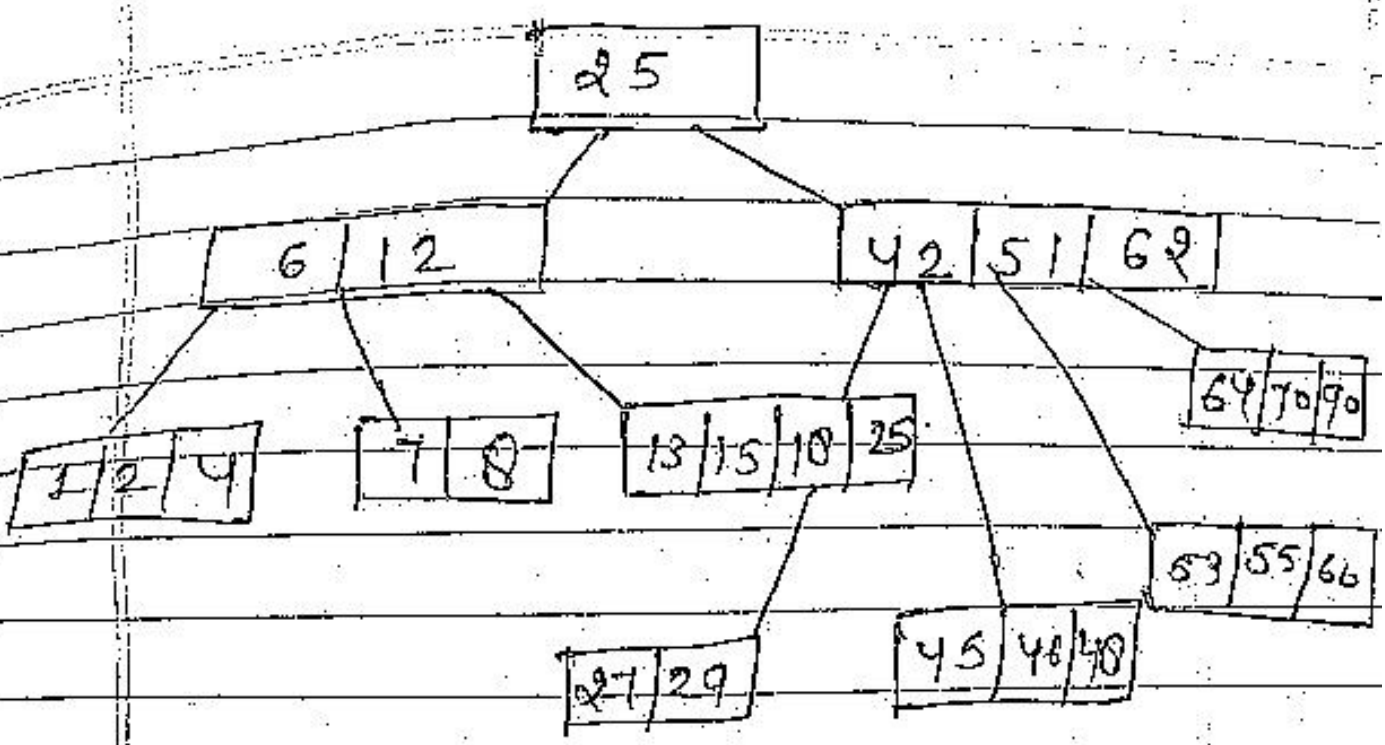
(1) Each node has at most m children.

(2) Each internal nodes has at least ceiling of m/2 children

(3) root node has at least 2 children if it not leaf

(4) A non leaf node with K children has K-1 keys

(5) All leaf nodes appear in the same level

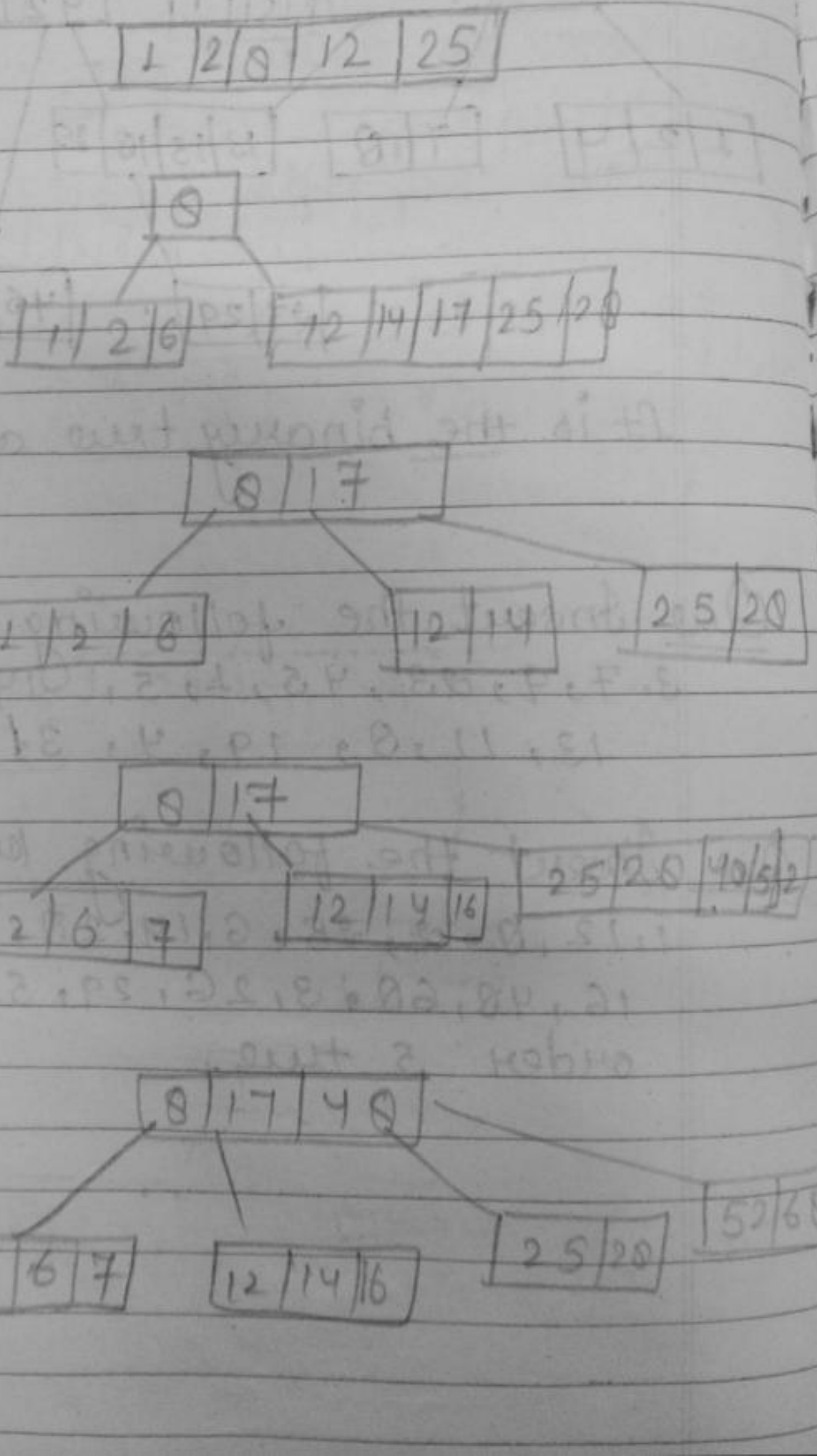


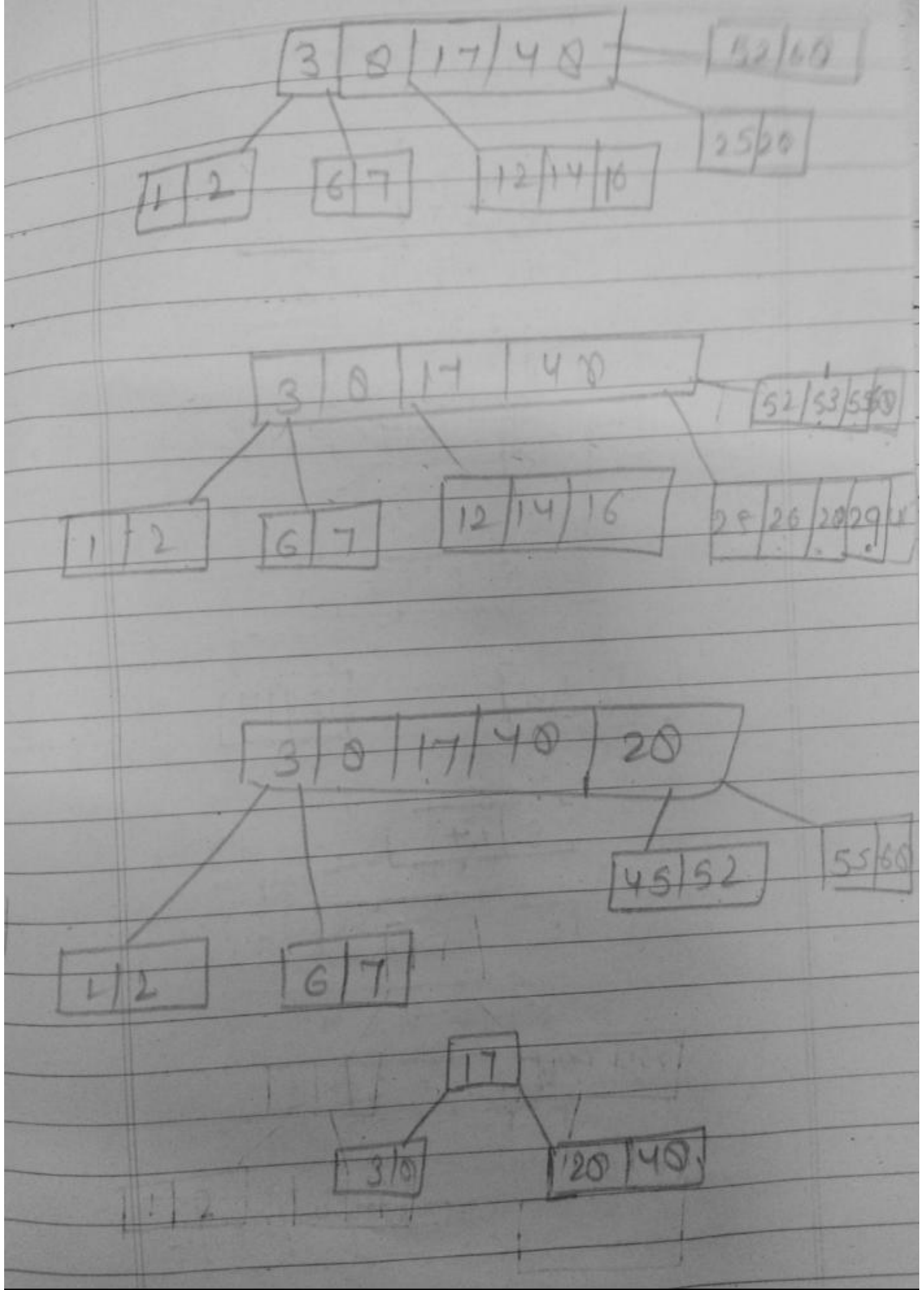
It is the binary tree of $m = 5$

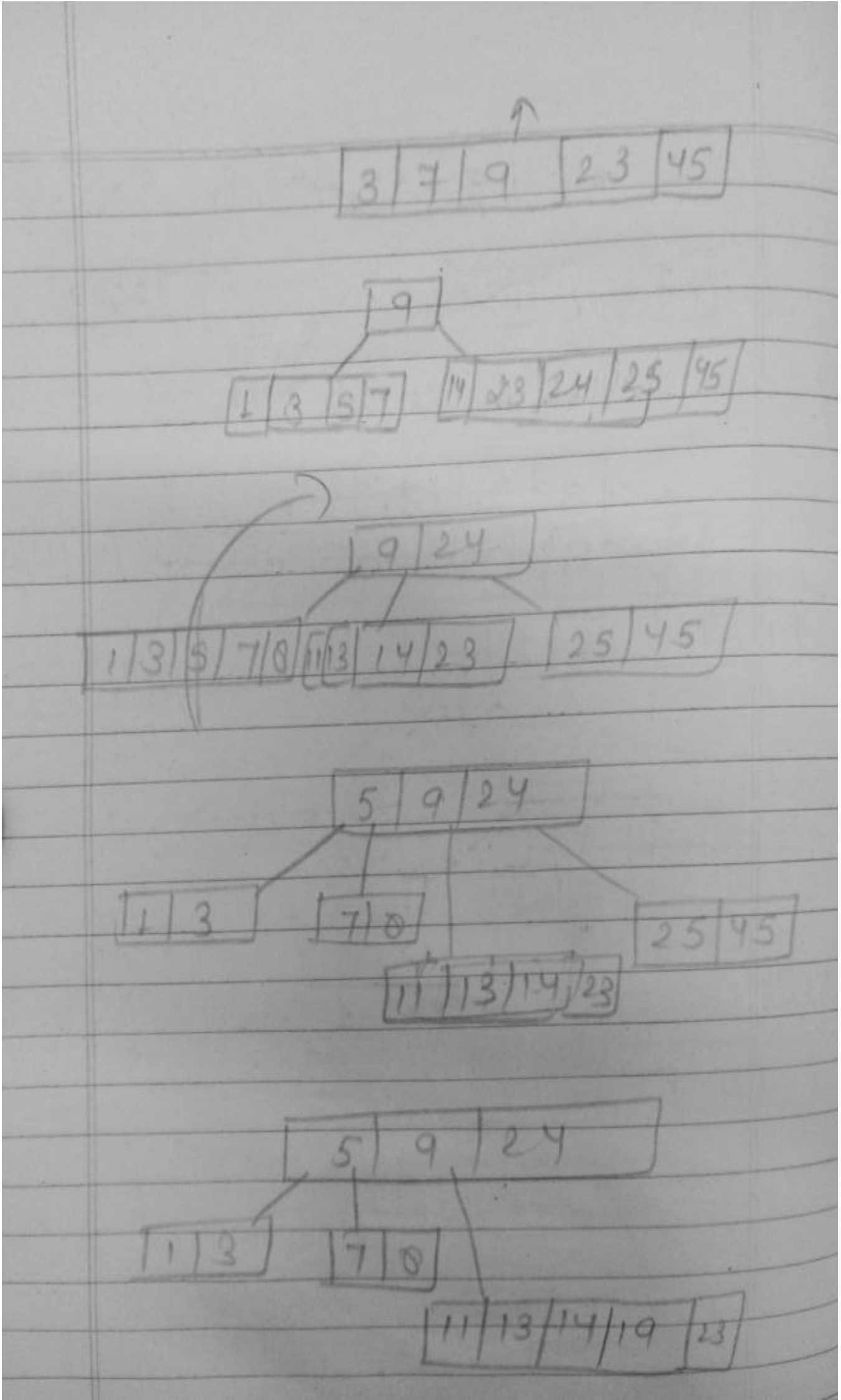
Qu Insert the following keys 5-way
3, 7, 9, 23, 45, 1, 5, 14, 25, 24
13, 11, 8, 19, 4, 31, 35, 56

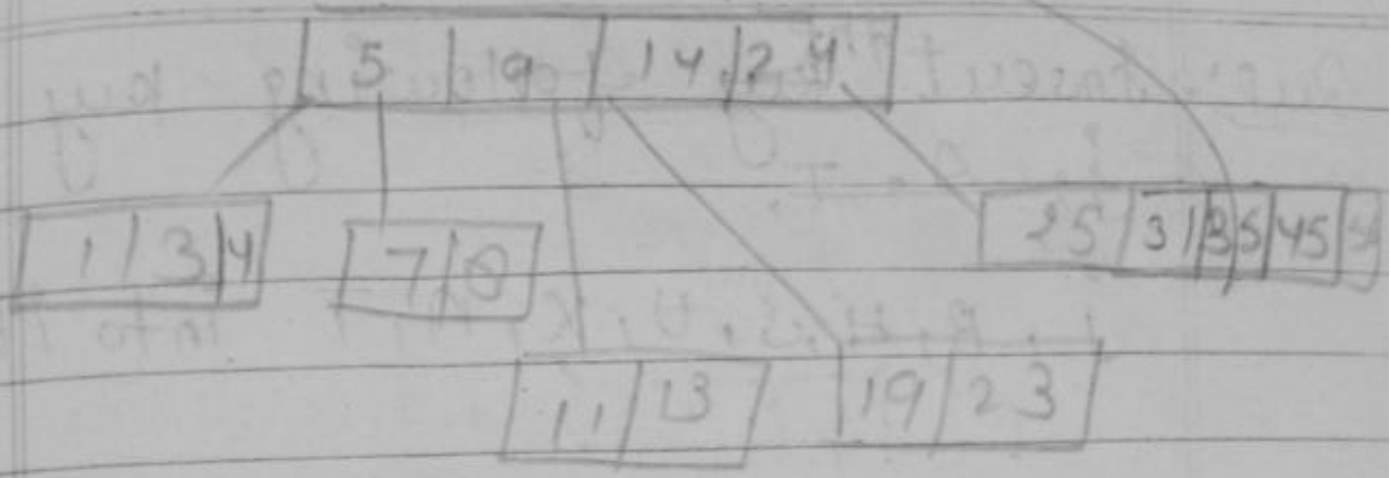
Qu Insert the following keys
1, 12, 8, 2, 25, 6, 14, 28, 17, 7, 52,
16, 48, 68, 3, 26, 29, 53, 55, 45
order 5 tree.

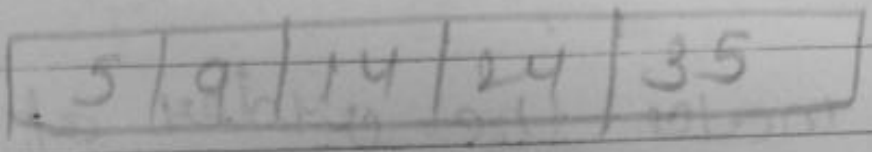
1

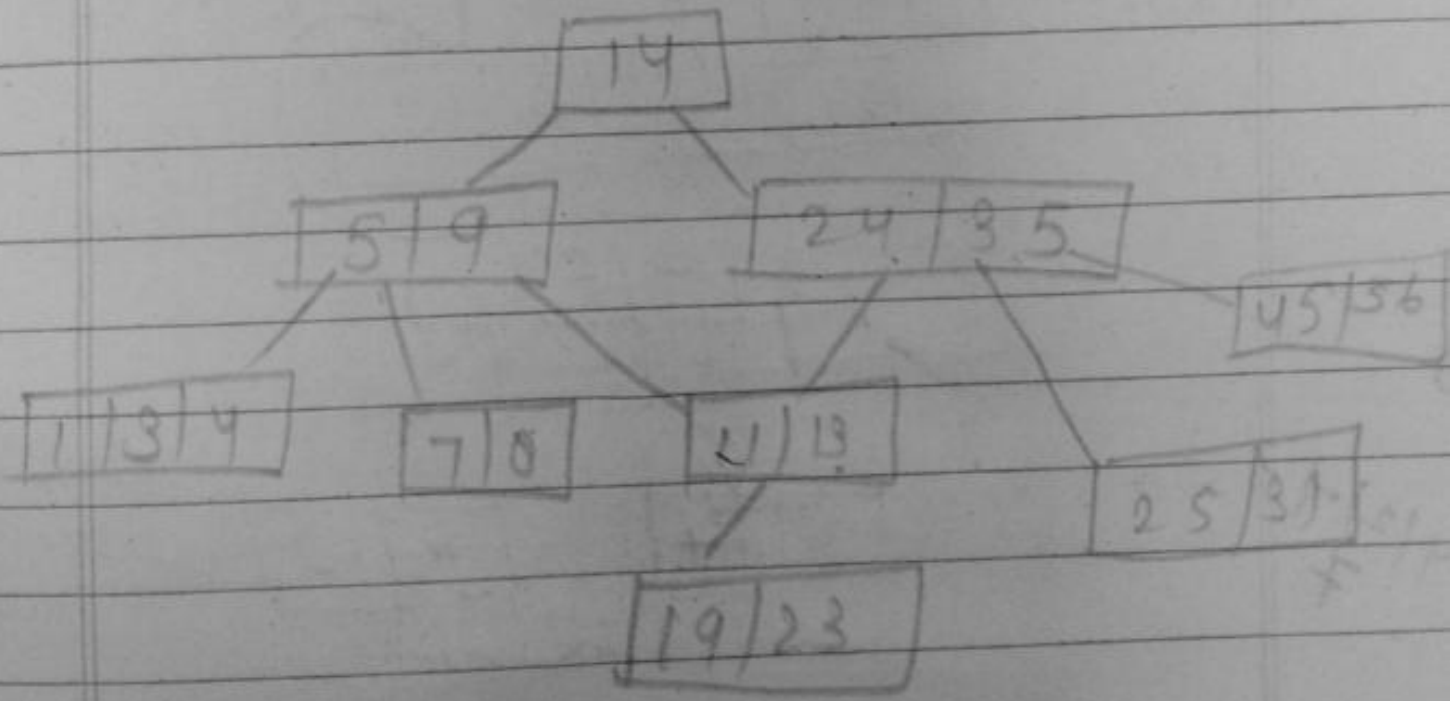

1 2 8 12 25
8
1 2 6
12 14 17 25 20
8 17
1 2 6
12 14
25 20
8 17
1 2 6 7
12 14 16
25 20 40 52
8 17 40
1 2 3 6 7
12 14 16
25 20
52 60


3
8
17
40
1
2
6
7
12
14
16
25
50
52
60
3
8
17
40
1
2
6
7
12
14
16
52
53
20
45
52
55
60
17
30
20
40


3 7 9 23 45
9
1 3 5 7
11 23 24 25 45
9 24
1 3 5 7 9
11 13 14 23
25 45
5 9 24
1 3
7 9
11 13 14 23
25 45
5 9 24
1 3
7 9
11 13 14 19 23

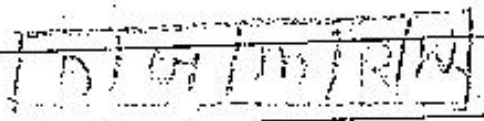

5 | 9 | 14 | 24
1 | 3 | 4
7 | 8
11 | 13
19 | 23
25 | 31 | 35 | 45


5 | 9 | 14 | 24 | 35

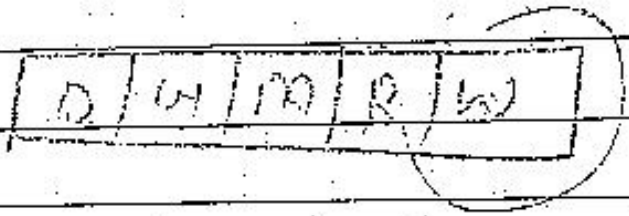

14
5 | 9
24 | 35
1 | 3 | 4
7 | 8
11 | 13
45 | 56
25 | 31
19 | 23

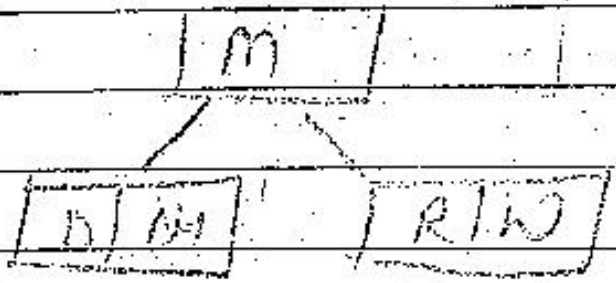
Ques:- Insert the ~~key~~ following key ~~j, A, T,~~

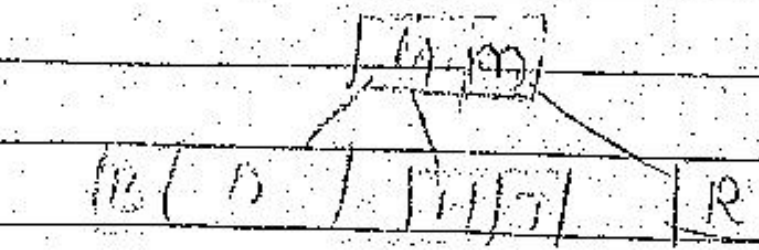
J, B, H, S, U, X, A, T into it

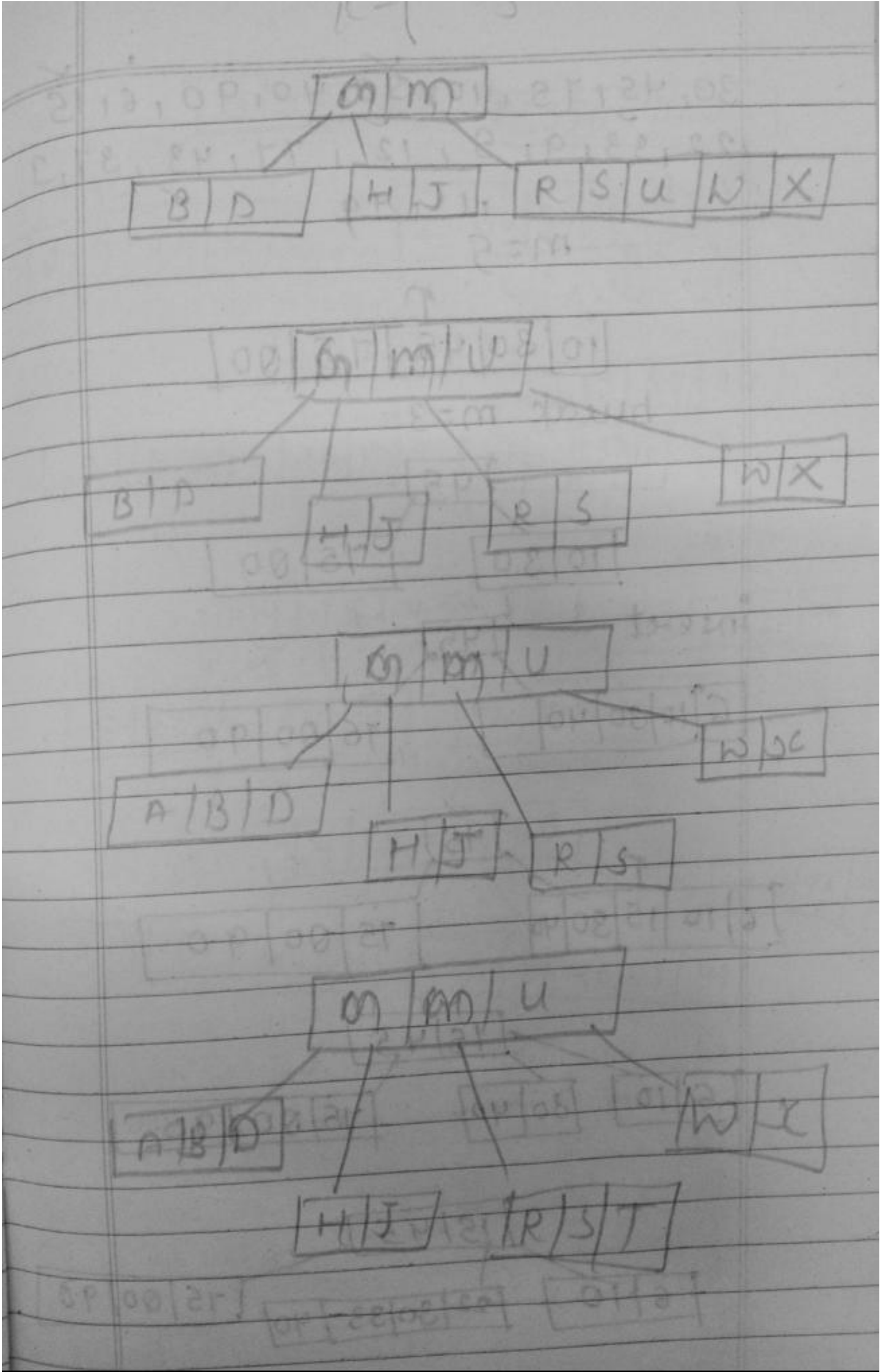


& make the order of degree.



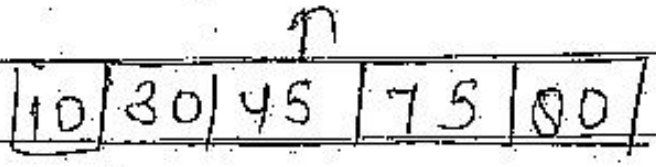




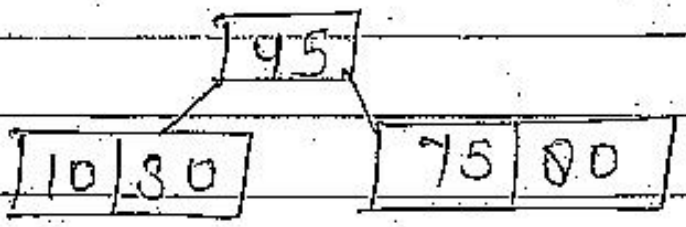

m
B D
H J
R S U W X
m U
B D
H J
R S
W X
m U
A B D
H J
R S
W X
m U
A B D
H J
R S T
W X

30, 45, 75, 10, 80, 40, 90, 6, 15
22, 33, 91, 5, 12, 77, 43, 37, 3
, 1, 49

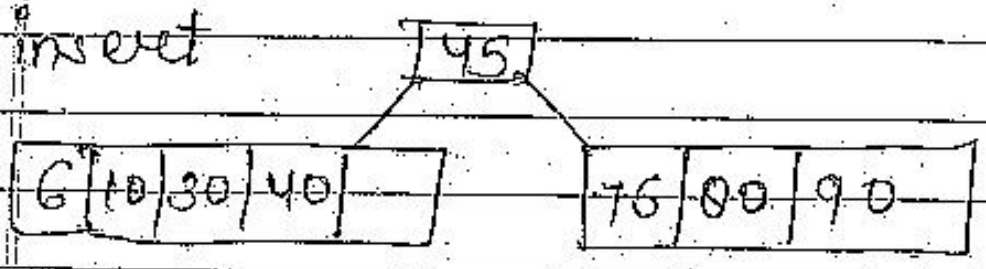
m = 5

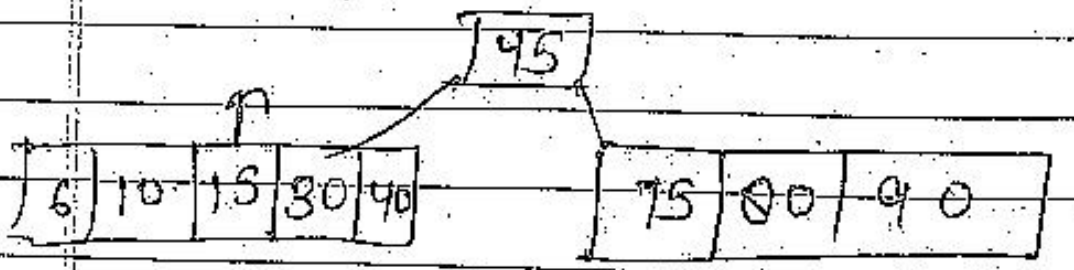


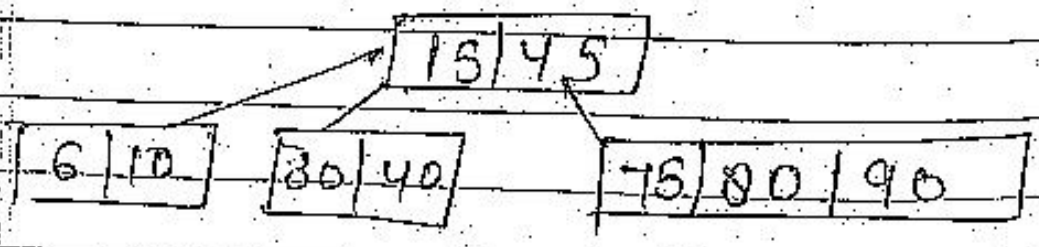
break m = 3

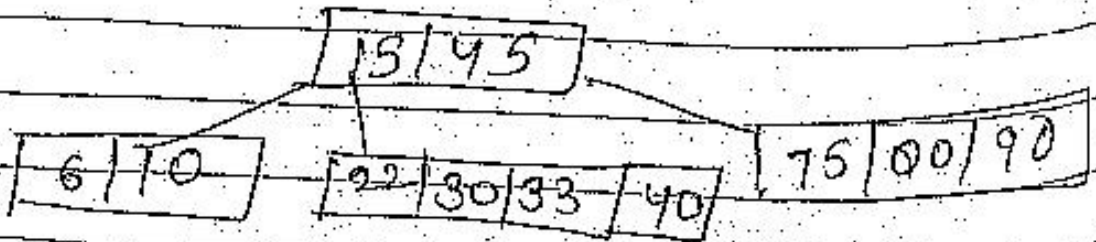


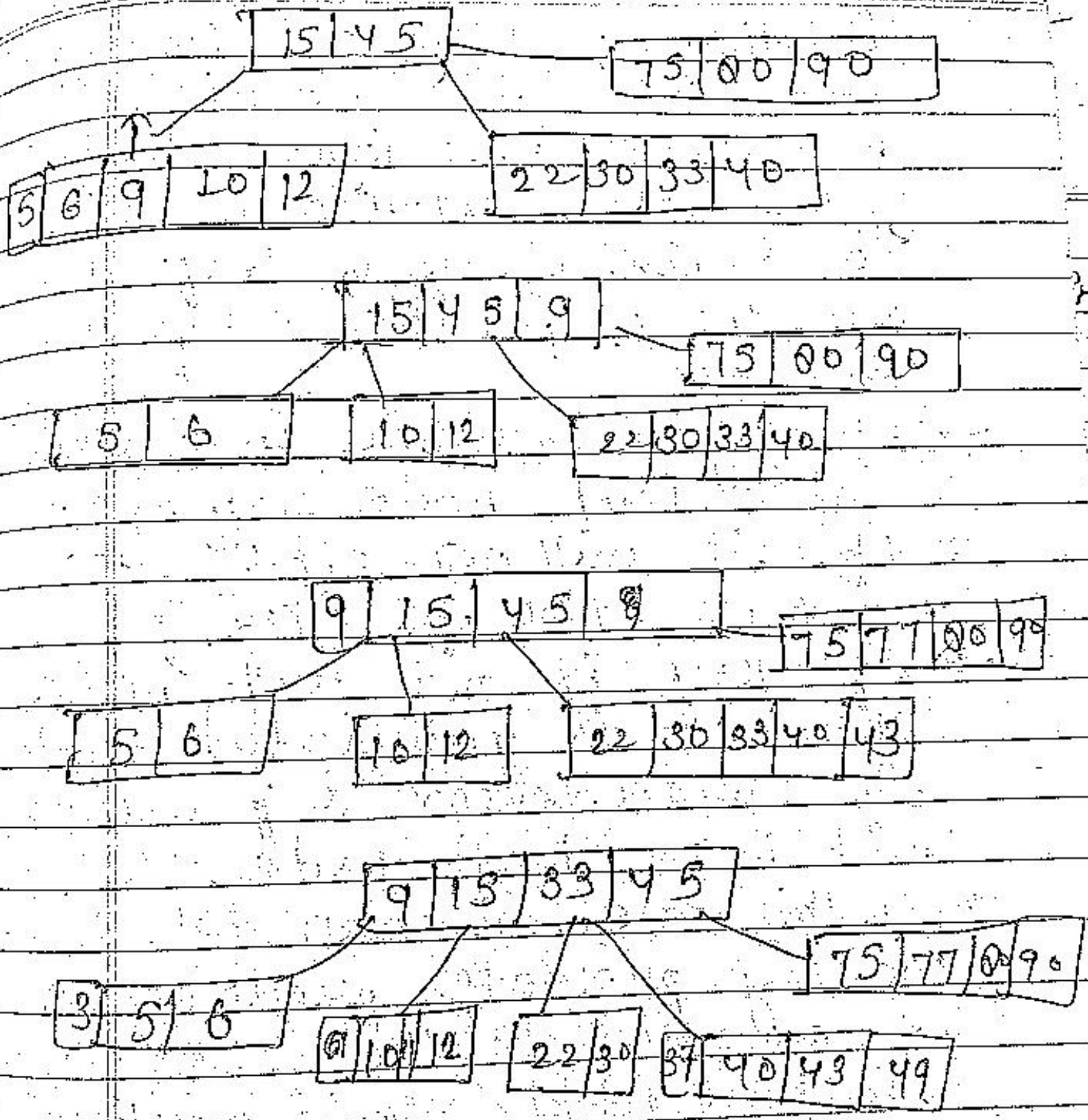
insert










15 45
75 00 90
5 6 9 10 12
22 30 33 40
15 45 9
75 00 90
5 6
10 12
22 30 33 40
9 15 45 88
75 77 00 99
5 6
10 12
22 30 33 40 43
9 15 33 45
75 77 00 90
3 5 6
10 12
22 30
37 40 43 49

Deletion of B Tree :⇒ During insertion of key always goes into a leaf node for deletion we wish to remove from a leaf there are three possible ways we can do

→ If key is already in a ~~node~~ leaf node remove

→ If it will not affect b-tree (too few keys)

→ If key is not leaf node then it is gauranted that its predecessor & successor will be a leaf node.

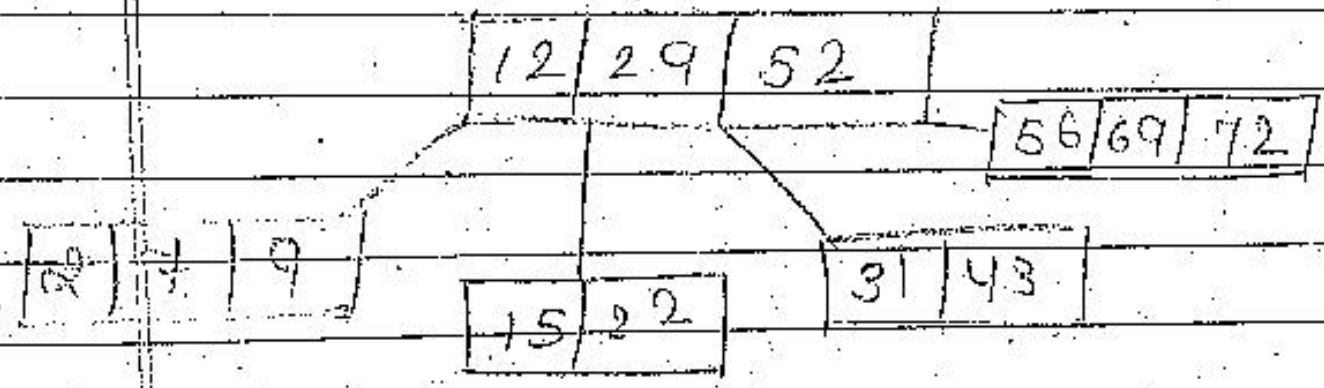
→ In this case we can delete the key promote predicessor or successor to the non-leaf deleted key position

→ If one & two case lead to a leaf node containing less than minimum no. of keys than we have to look at the sibling immediatly adjacent to the leaf.

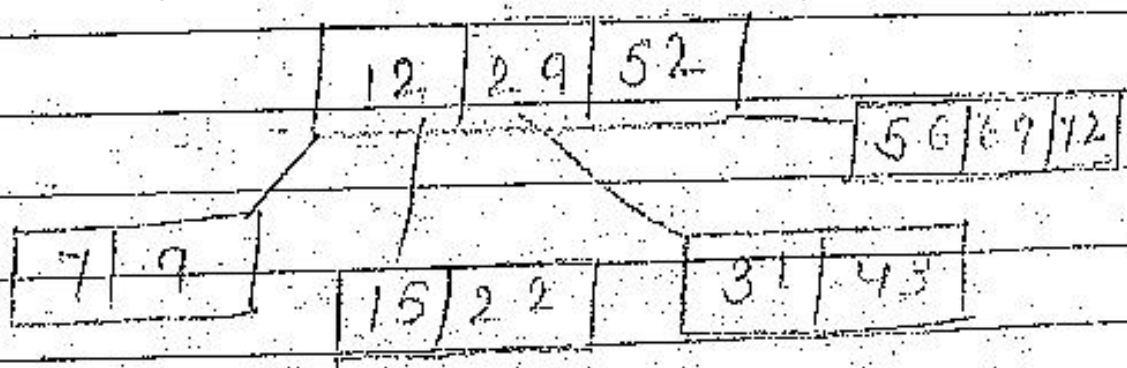
(1) if one of them has more than minimum no

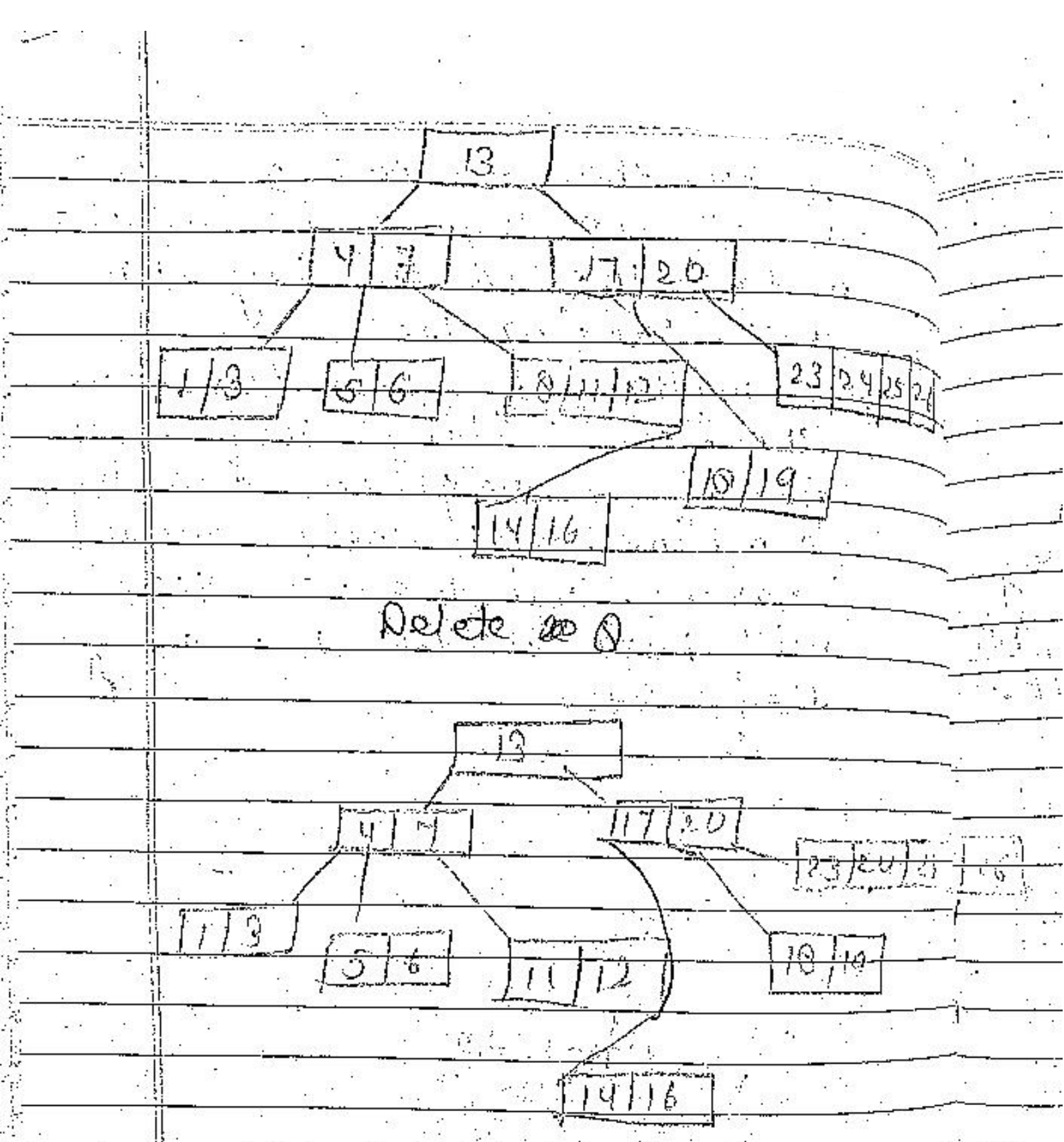
of keys than we can promote
1 of the key to be parent
& take the parent key into
our lacking leaf.

(2) if neither of them has more than min. no. of keys the lacking leaf and one of its neighbour can be combine with their shared parent and the new leaf will have correct no. of keys.



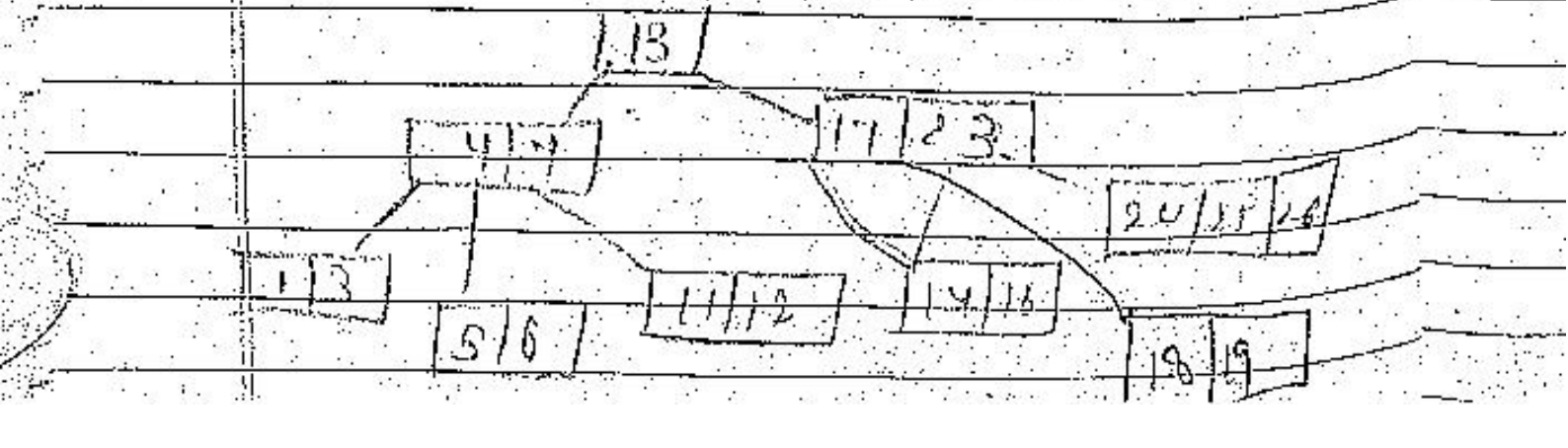
Delete - 9 (k-1) m=3



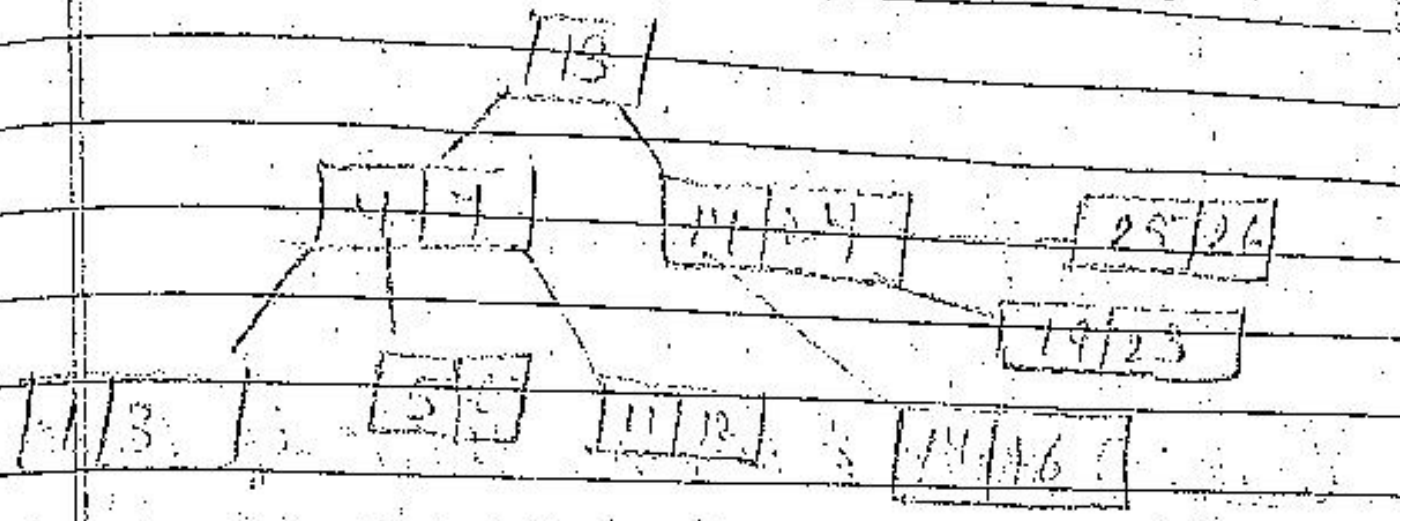


Delete 20

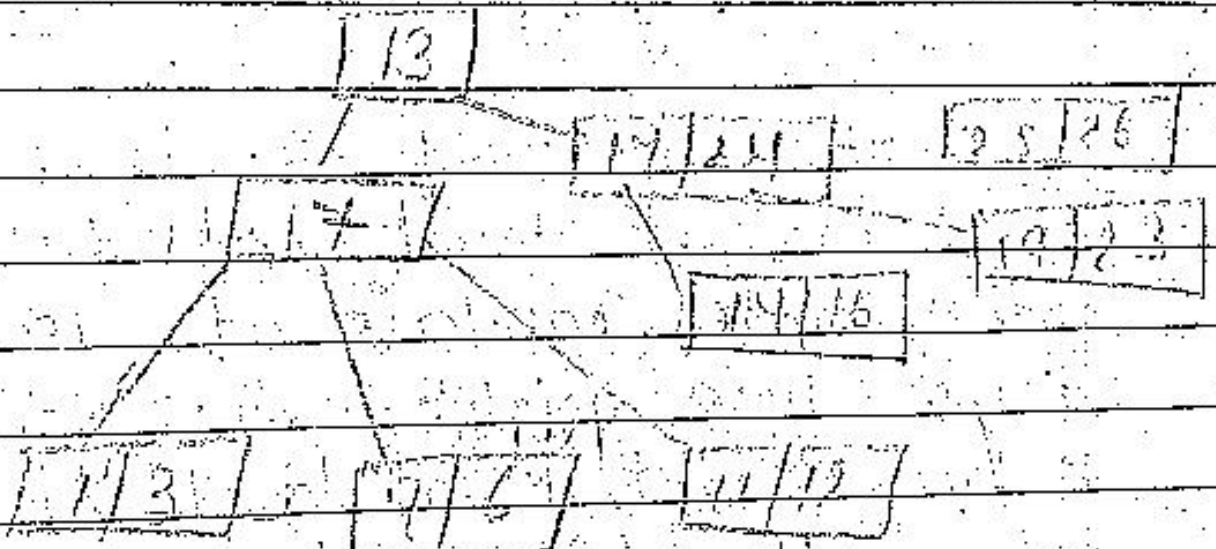
delete - 20

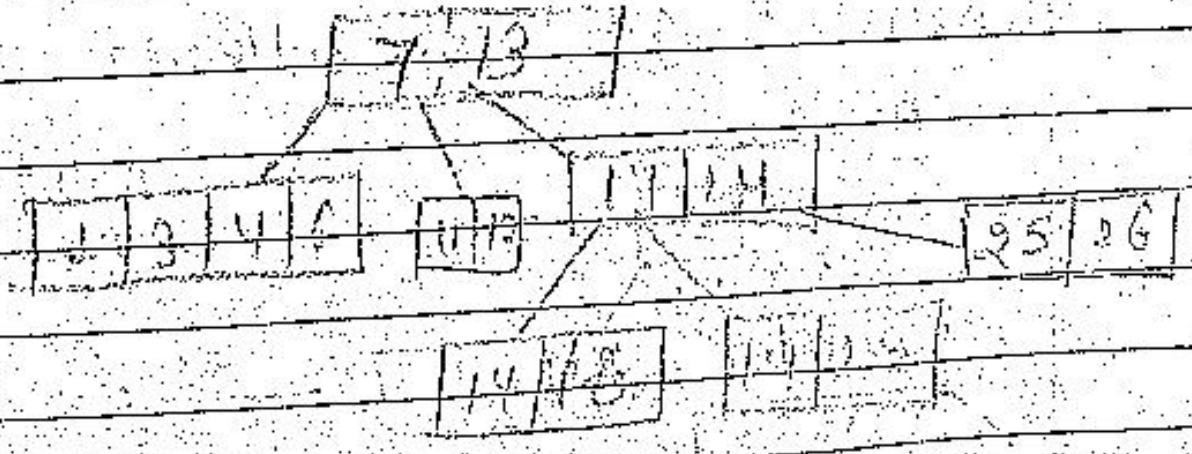


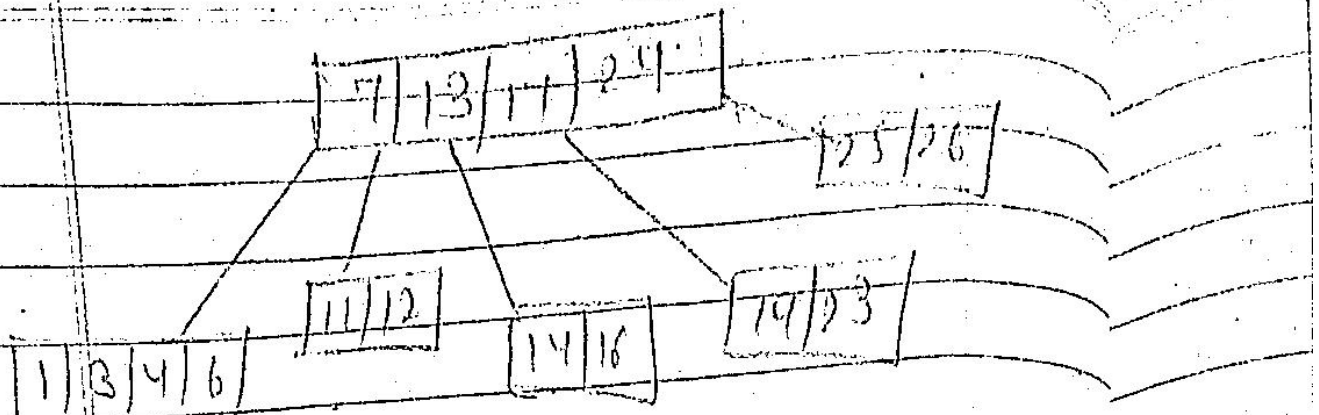
delete - 10



delete - 5

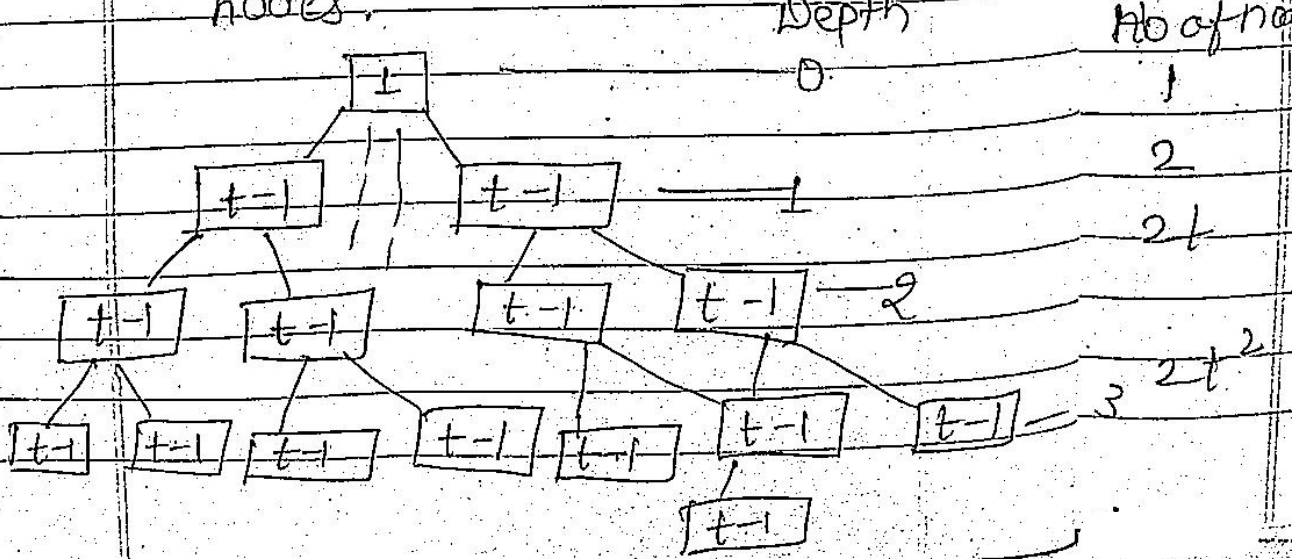






Qu:- state & prove that if $n \geq 1$, then for any n keys binary tree of height 'h' and of minimum degree $t \geq 2$ is $h \leq \log_{t} \frac{n+1}{2}$

Proof:- The root of B-Tree contains at least one key & all other nodes contain at least $t-1$ keys whose height h has at least 2 nodes at dept 1, at least $2t$ nodes at depth 2 & atleast $2t^2$ nodes at dept 3 & soon until at depth h of t has atleast $2t^{h-1}$ nodes.



$b^n = a \quad \log_b a = n \quad \log_t\left(\frac{n+1}{2}\right) = h$

no. of keys satisfies the inequality

$$n \geq 1 + (t-1)\sum_{i=1}^{h} t^{i-1}$$

$$n \geq 1 + (t-1)\left[\frac{t^h - 1}{(t-1)}\right] \quad (\text{G.P})$$

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

$$t^h \leq \left(\frac{n+1}{2}\right)$$

taking log of base t

$$\boxed{h \leq \log_t\left(\frac{n+1}{2}\right)}$$

RB

total no of internal node $= \frac{h}{2} - 1$

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

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

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

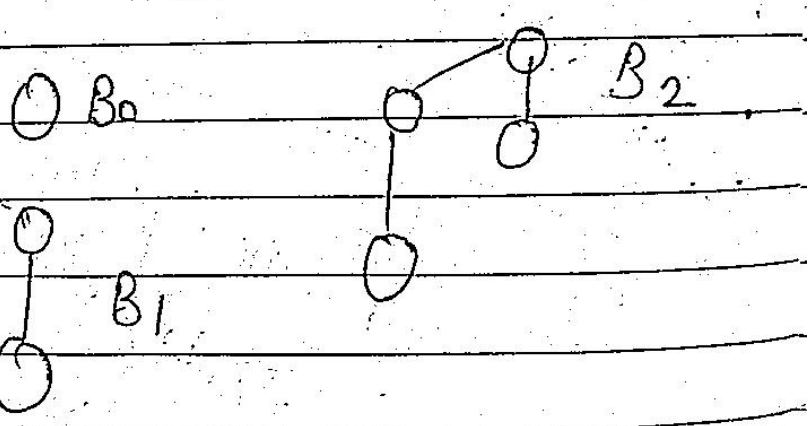
$$\log_2(n+1) \geq \frac{h}{2}$$

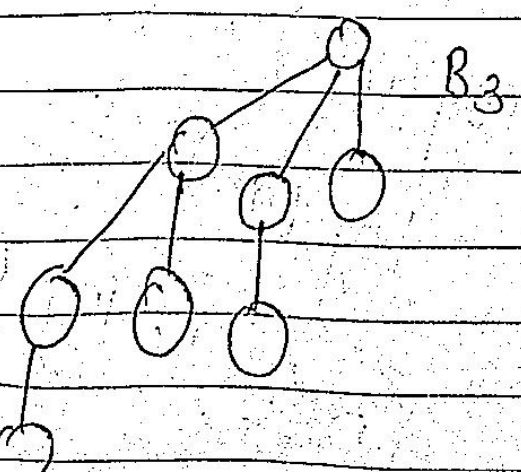
$$\boxed{h \leq 2 \cdot \log_2(n+1)}$$

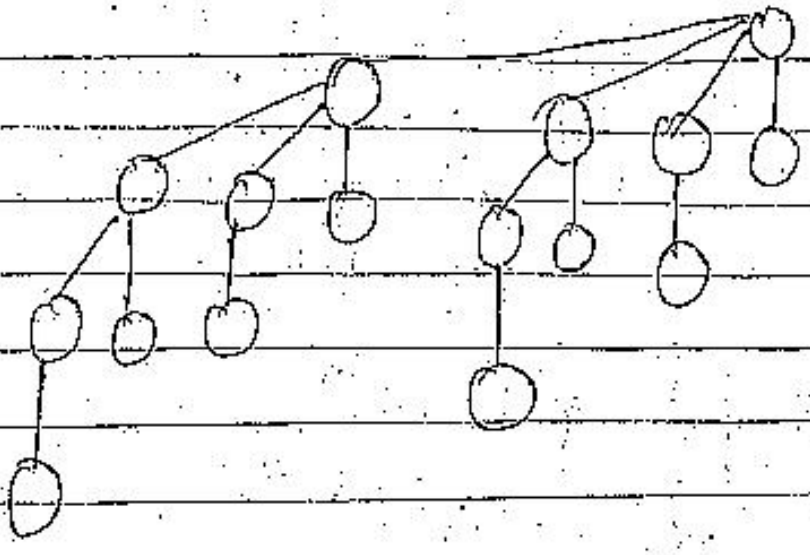
Binomial Heap

Binomial heap is the way of implementing mergable heaps there are following operation that are performed on by binomial heap.

1. Make heap
2. Insert a node
3. minimum of h
4. Union of two heap or merge.
5. Decrease a key.
6.



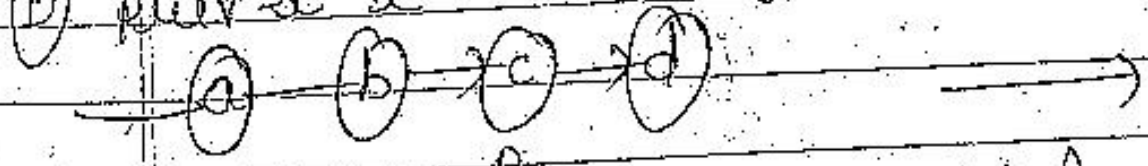




Depth	no. of nodes
0	1
1	4
2	6
3	4
4	1

Union :-

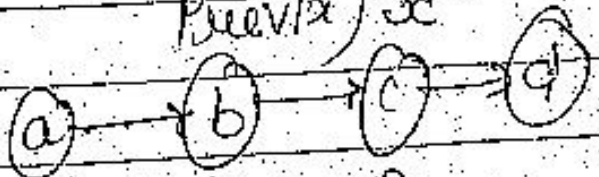
(1) prev x x next[x] sibling (next[x])



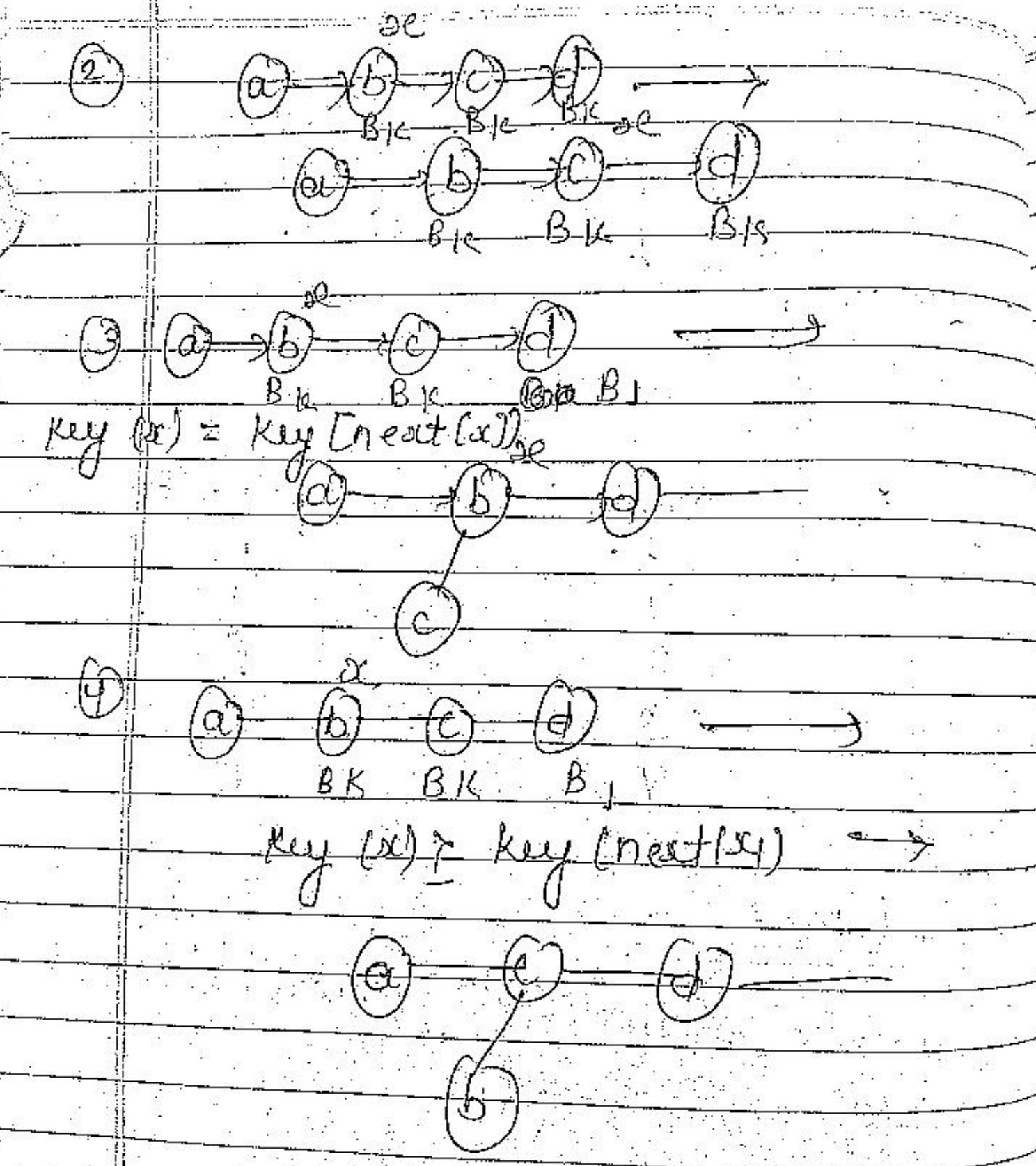
B_K B_1

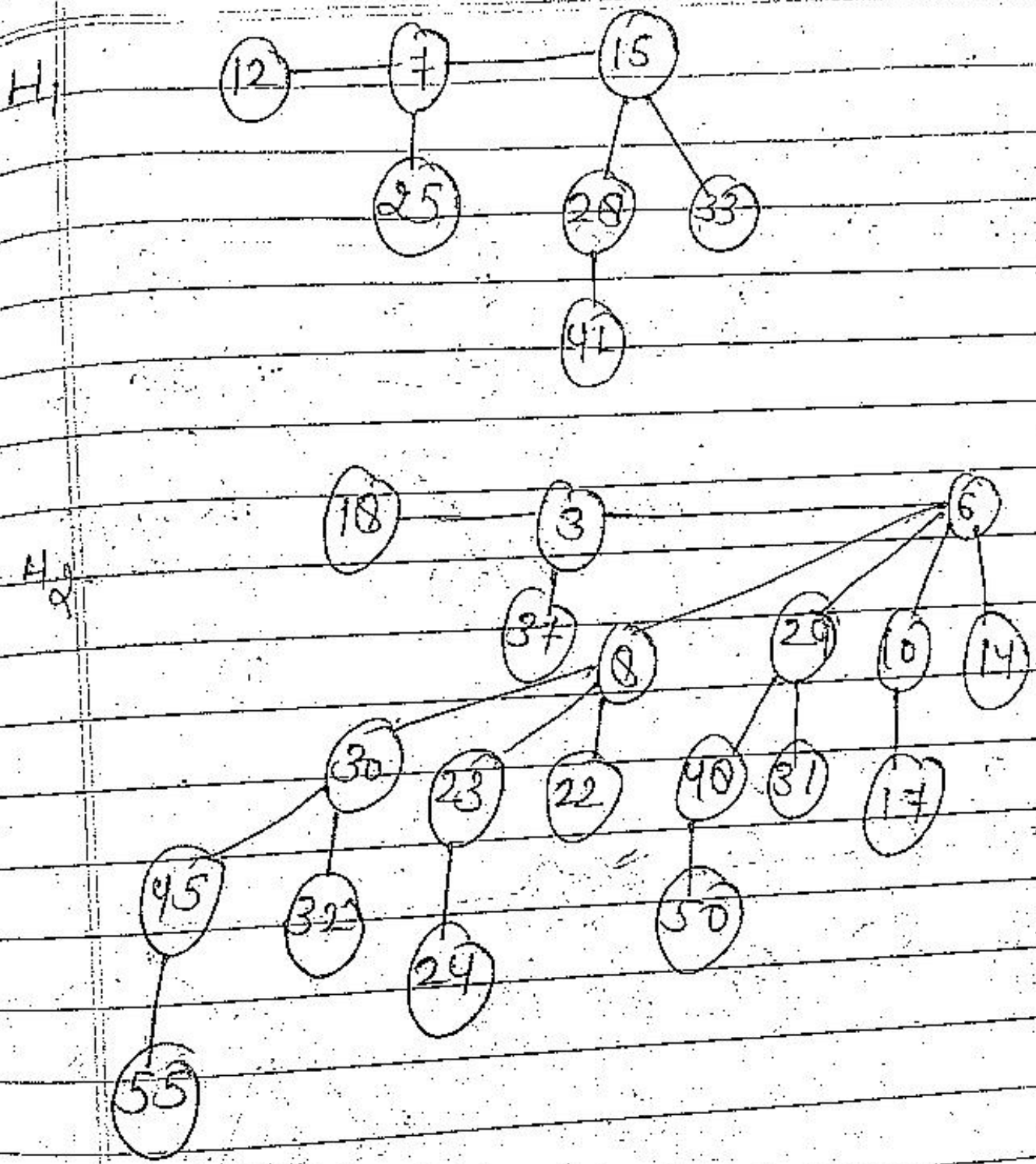
next(x)

prev(x) x

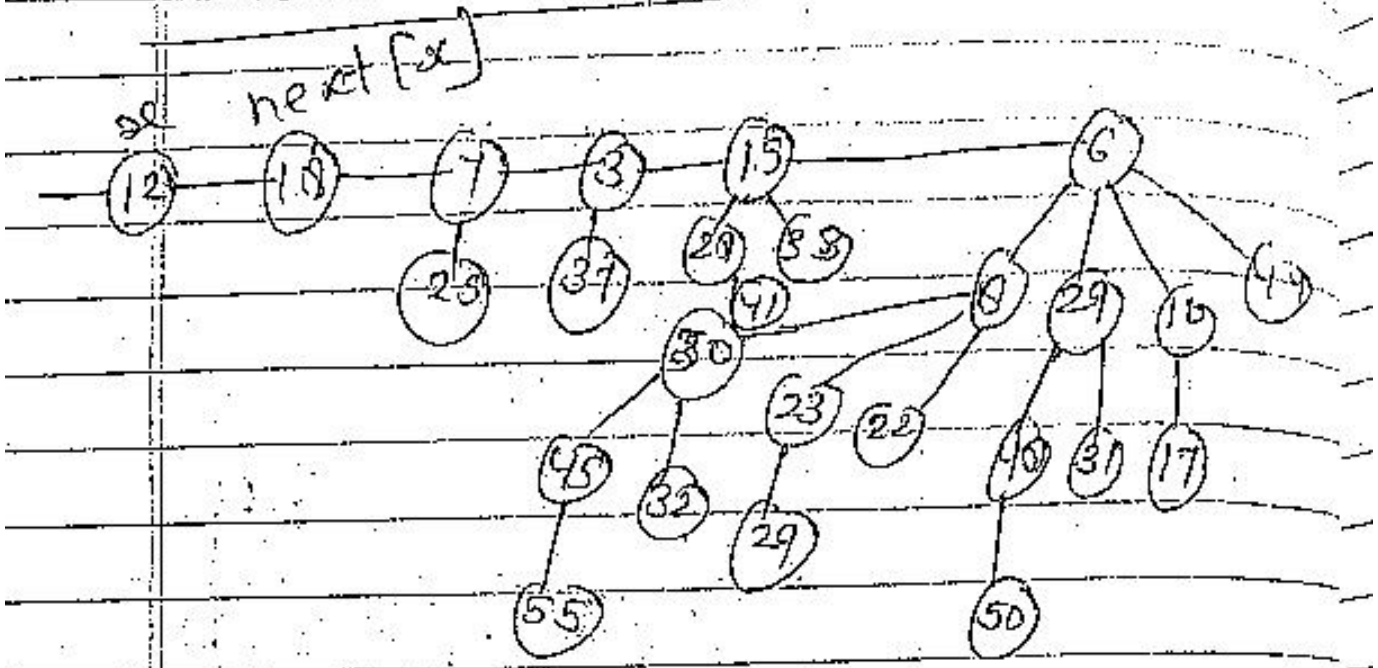


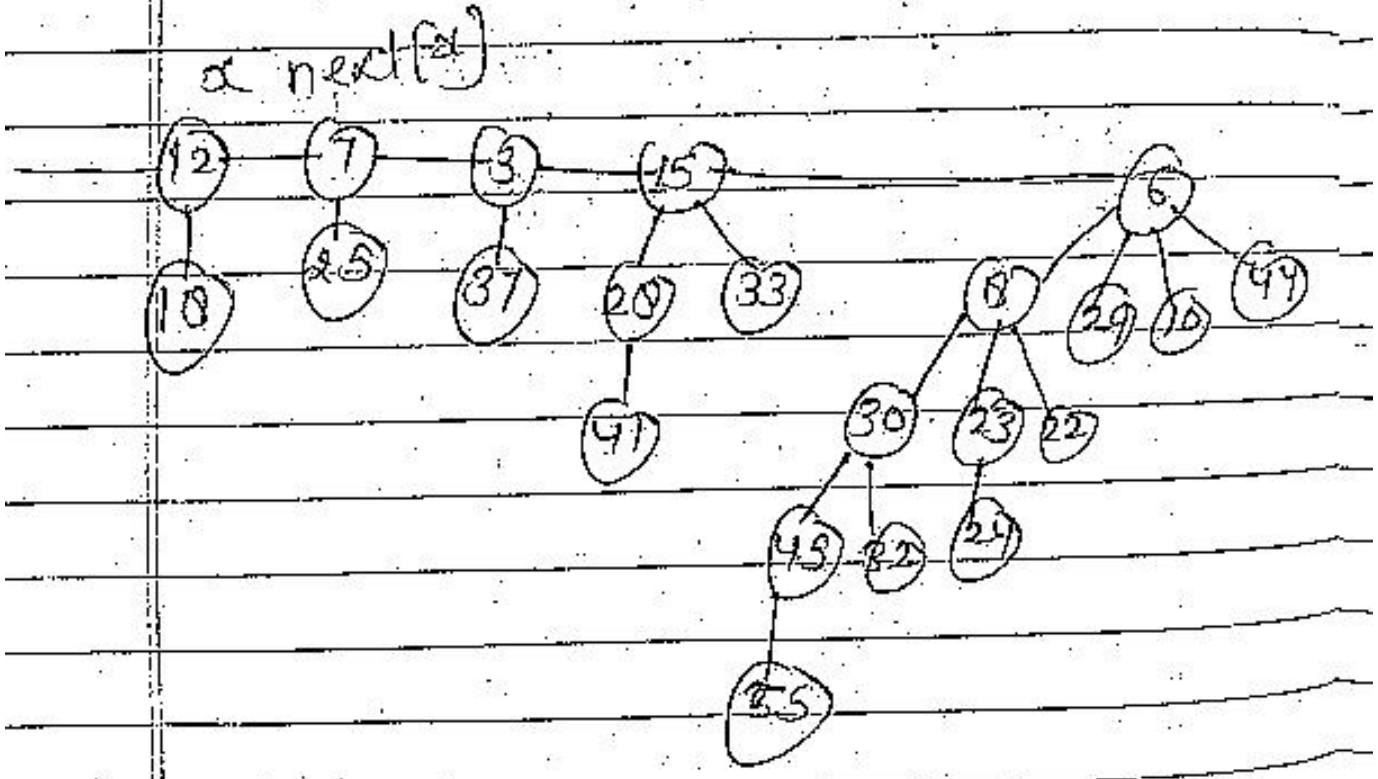
B_K B_1

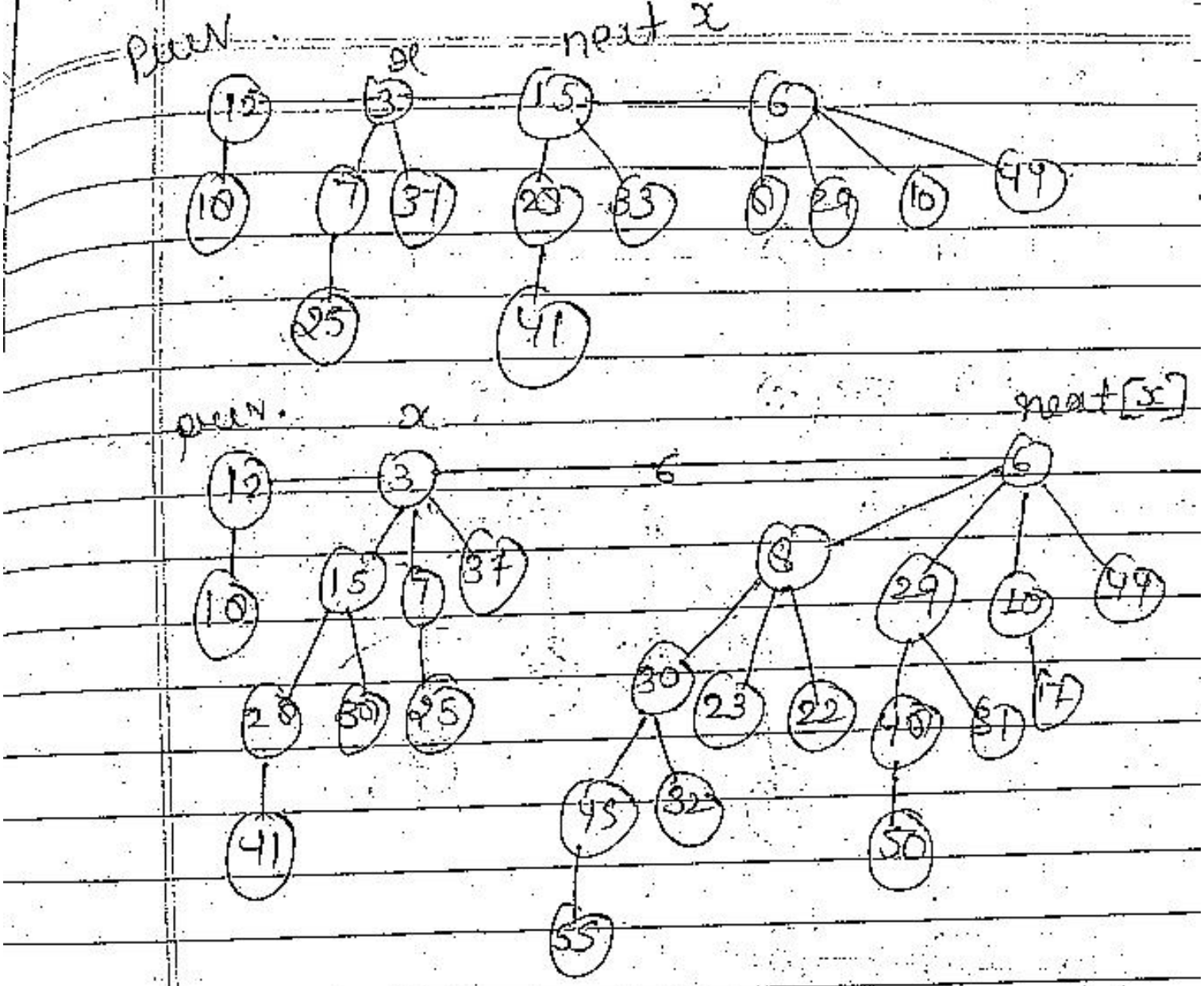

2
a
b
c
d
BIC
BIC
BIC
BIC
a
b
c
d
BIC
BIC
BIS
3
a
b
c
d
BIC
BIC
BI
Key (x) ≥ Key [next (x)]
a
b
d
c
4
a
b
c
d
BK
BK
B
Key (x) ≥ key (next (x))
a
c
d
b


H1
12
7
15
25
28
33
41
H2
18
3
6
37
8
29
10
14
30
23
22
40
31
17
45
32
24
50
55

Binomial Heap :-

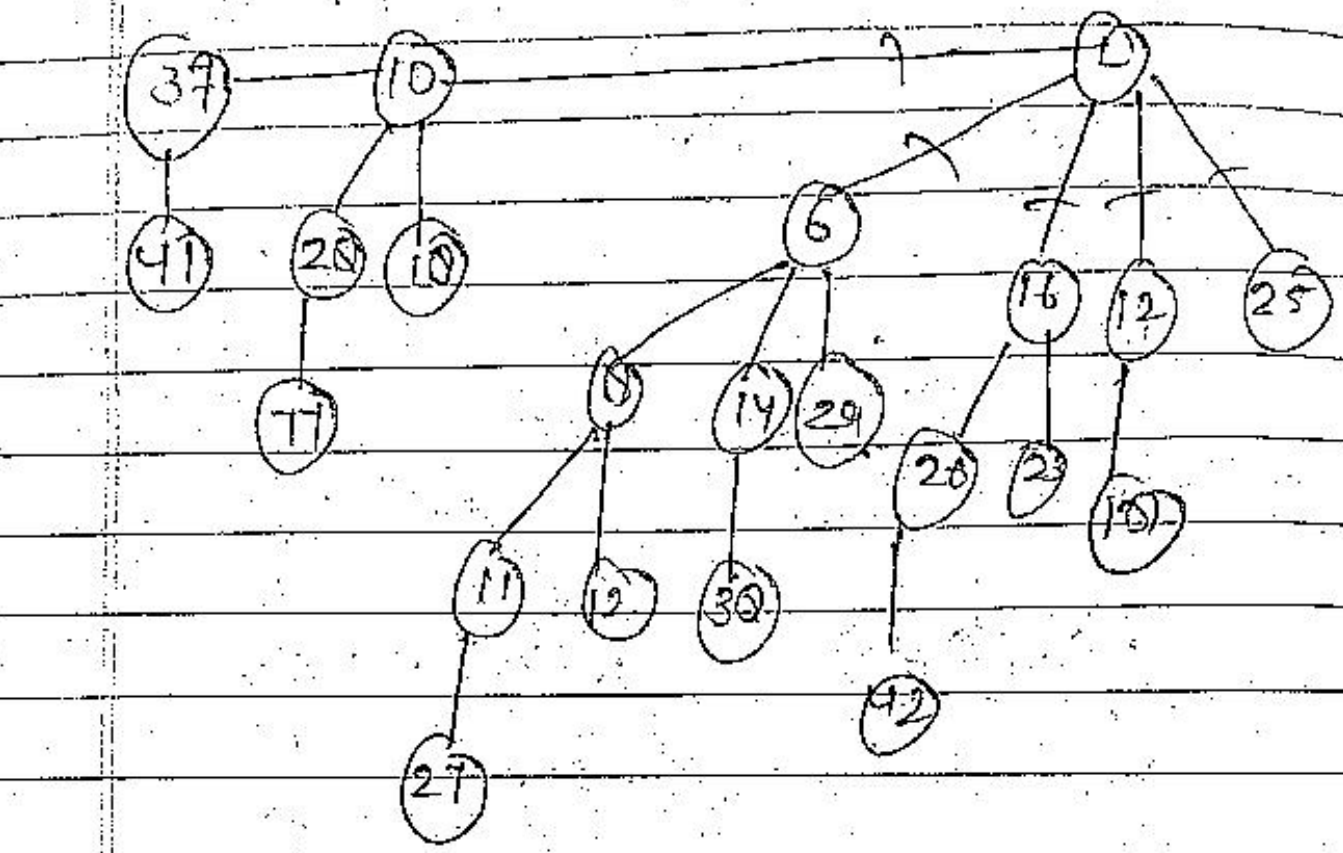


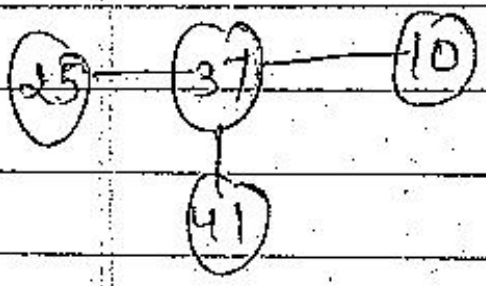



p[x]
x
next x
12
3
15
6
10
7
37
20
33
6
29
10
49
25
41

p[x]
x
next[x]
12
3
6
10
15
7
37
8
29
10
49
20
30
23
22
40
31
17
45
32
50
41
55

(2) Extract min :-





UNITING TWO BINOMIAL HEAPS

Unite two binomial heaps H_1 & H_2.

→ merge the root lists of H_1 & H_2 into a single linked list H.

→ roots of binomial trees in H must be in order of increasing degree.

1. x
2. previous - x
3. next - x

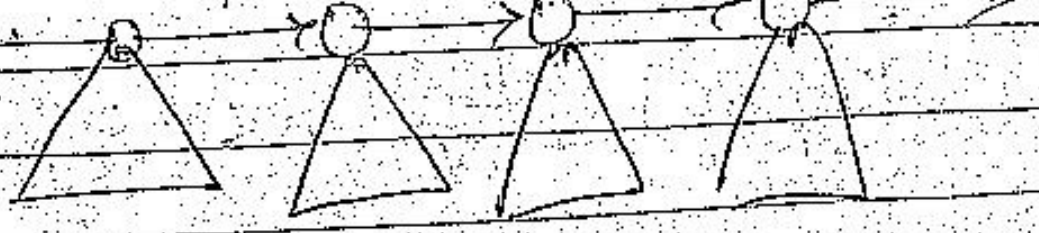
Rules :- (1)

previous → x x next - x

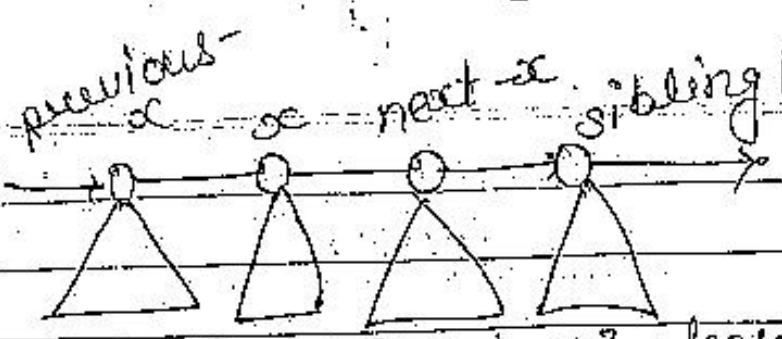
0 → 0 → 0 → 0 →

degree[x] ≠ degree[next-x]

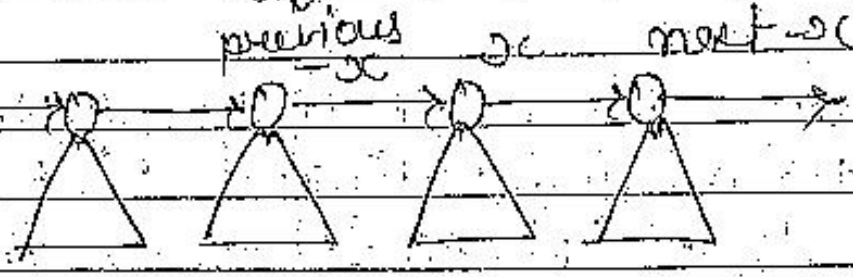
previous → x x next - x



(2)

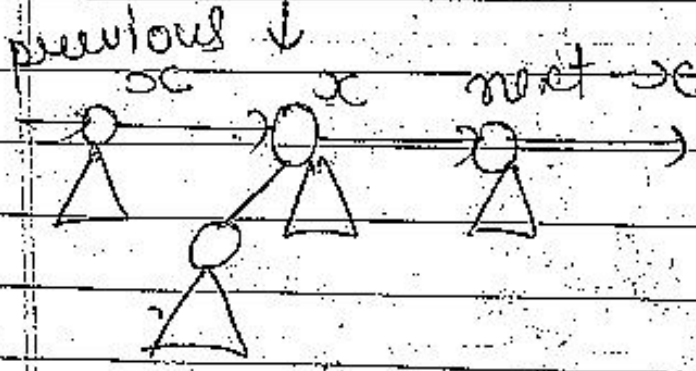


degree [x] = degree [next-x] = degree [sibling [next-x]]

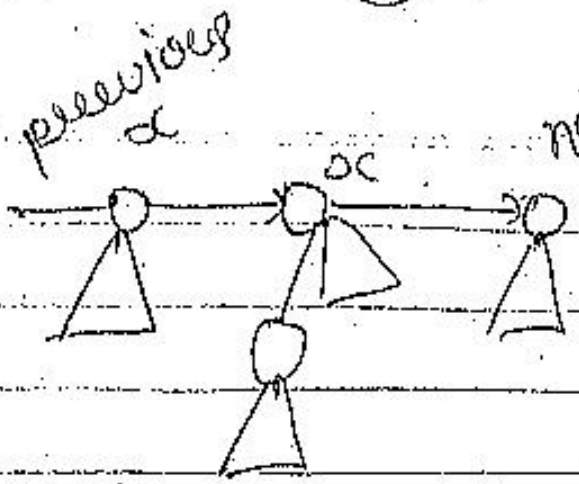


(3)

degree [x] = degree [next-x] ≠ degree [sibling [next-x]]



key [x] ≤ key [next-x]



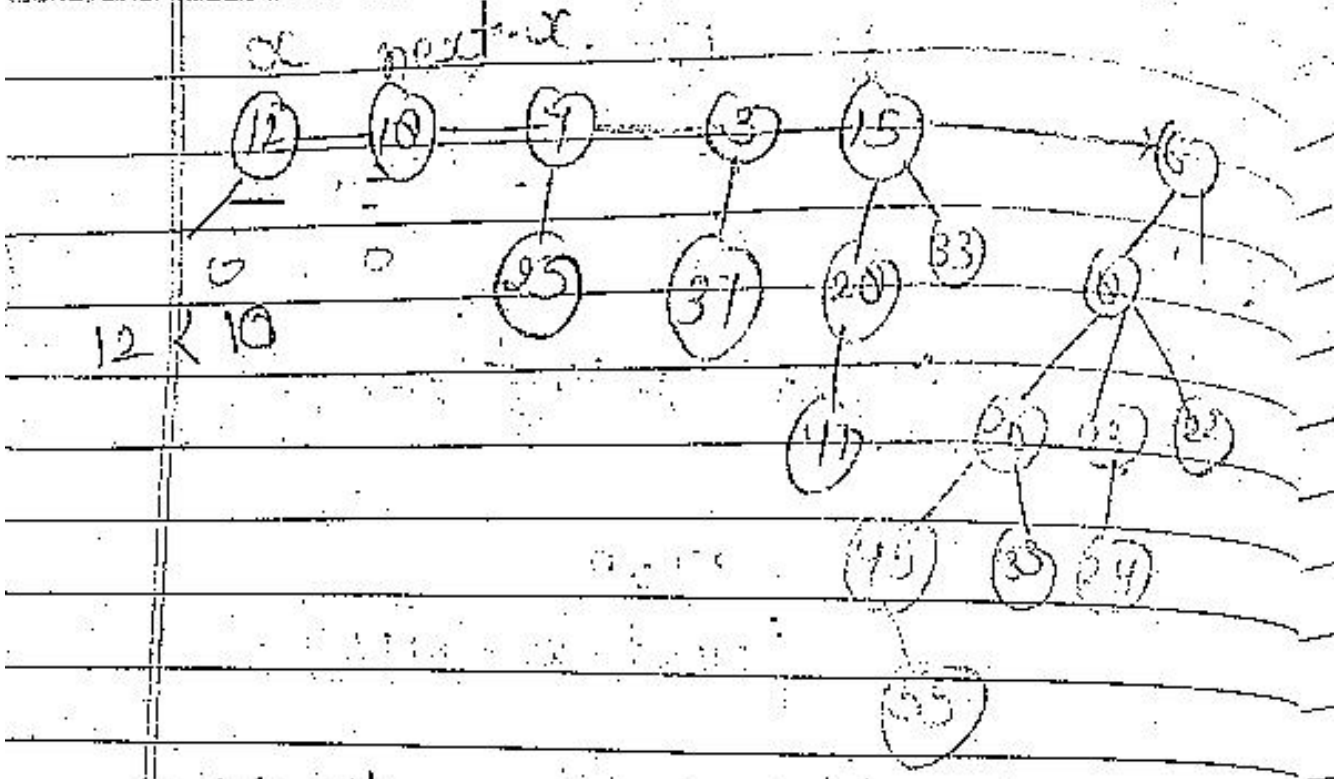
key [x] > key [next-x]

STOP

next-x = NIL

★ properties of binomial tree

(1) B_K tree has 2^K nodes
(2) Degree of root is K
(3) Height = K
(4) there are exactly ^kC_i nodes at depth i
where $^kC_i = \frac{k!}{(k-i)!\,i!}$

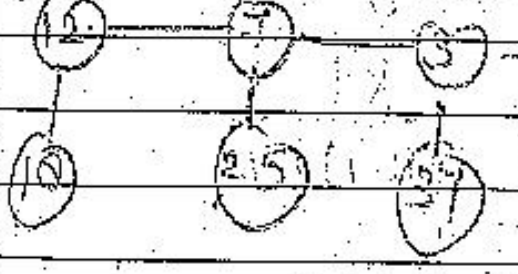


Heap:

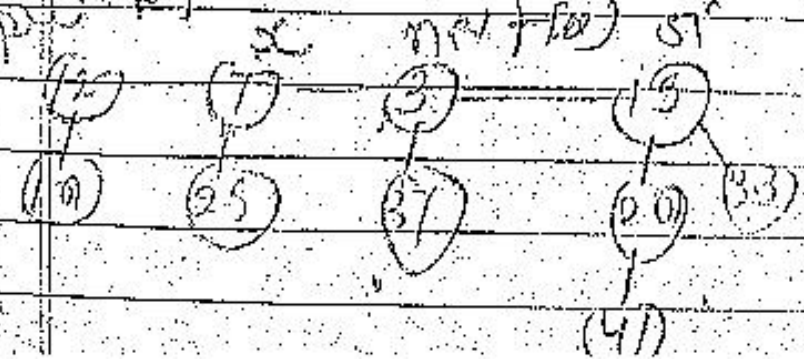
degree[x] = degree[next-x]
≠ degree of sibling

12 < 10.

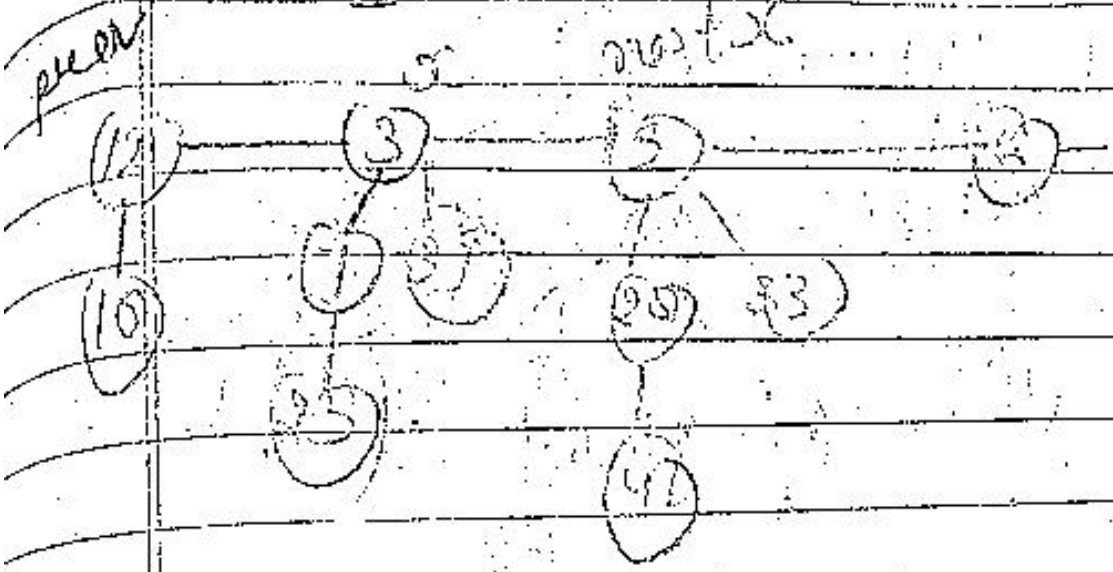
next-x sibling of x



degree[x] = degree[next-x]
= degree of sibling

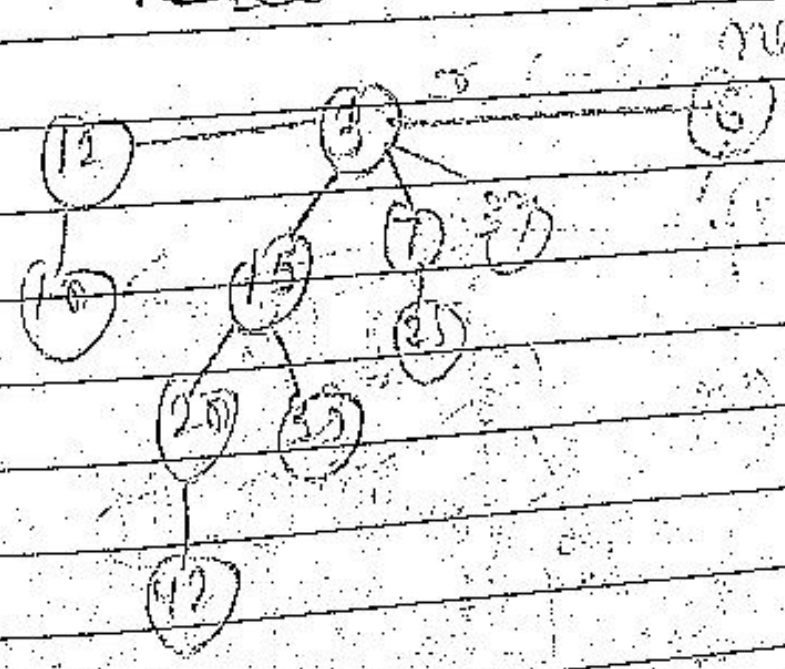


7 2 3



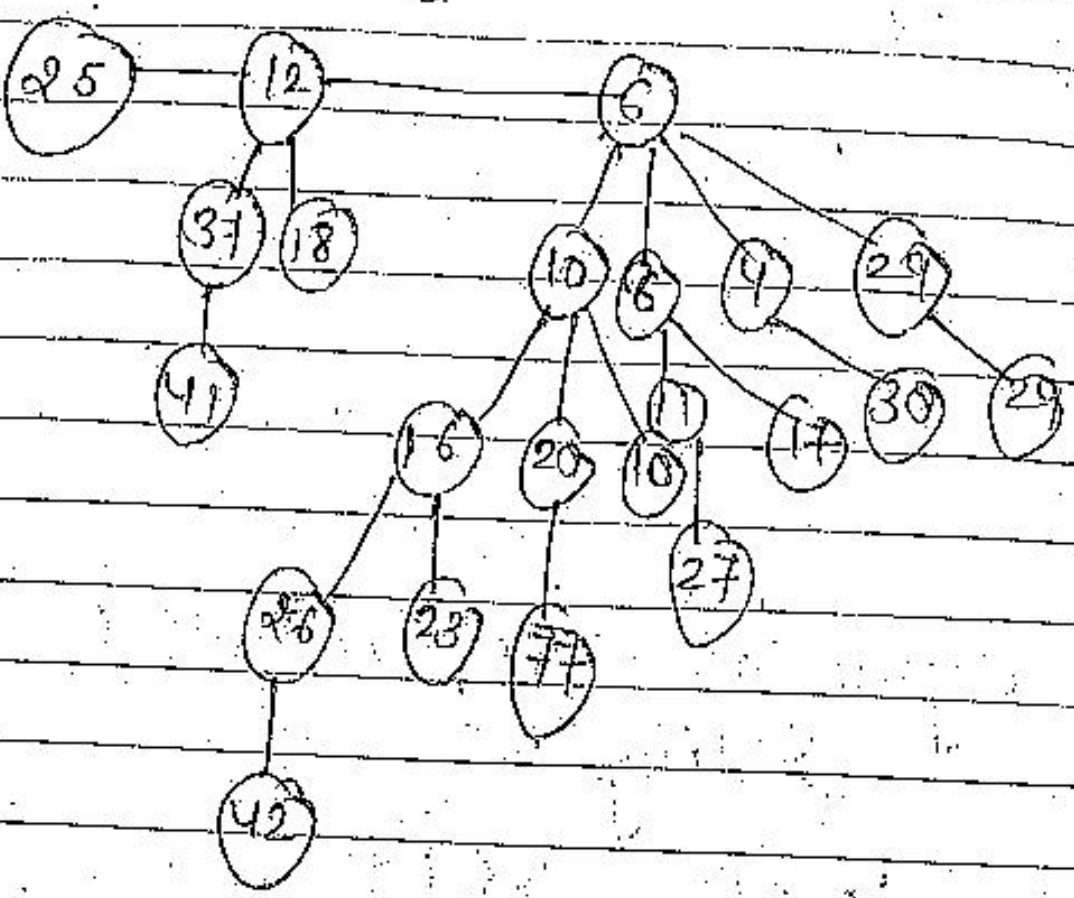
degree [x] = degree [next-x] ≠ sibling

5 2 6 3 2 15

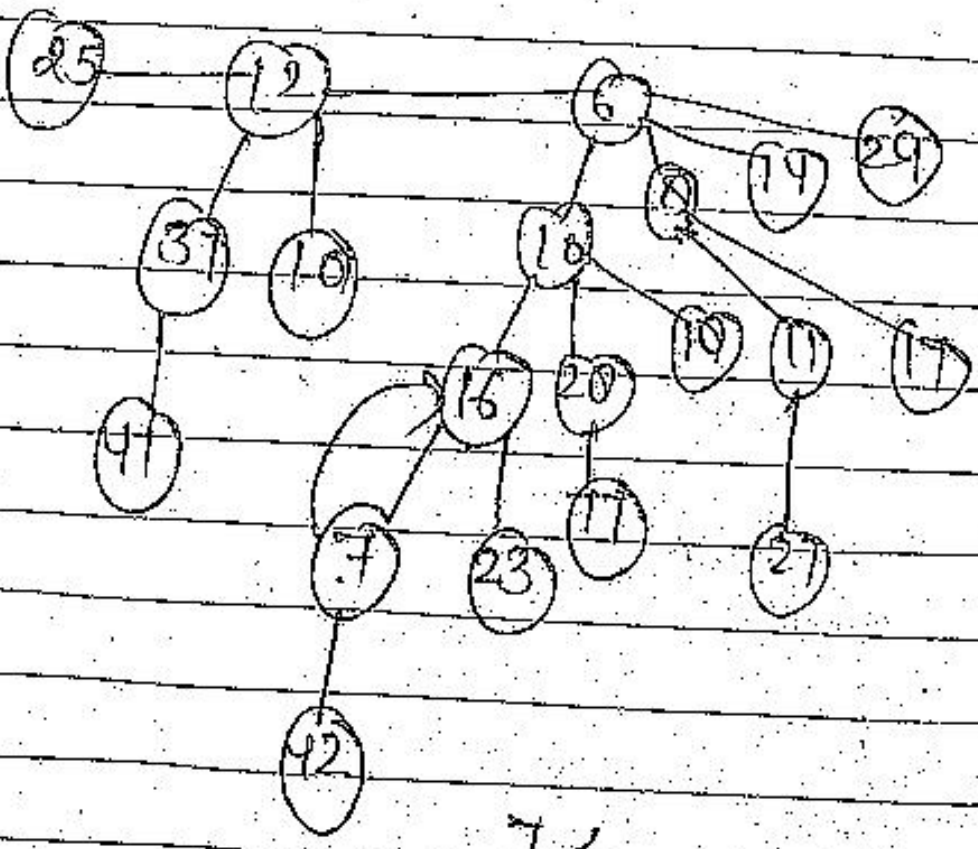


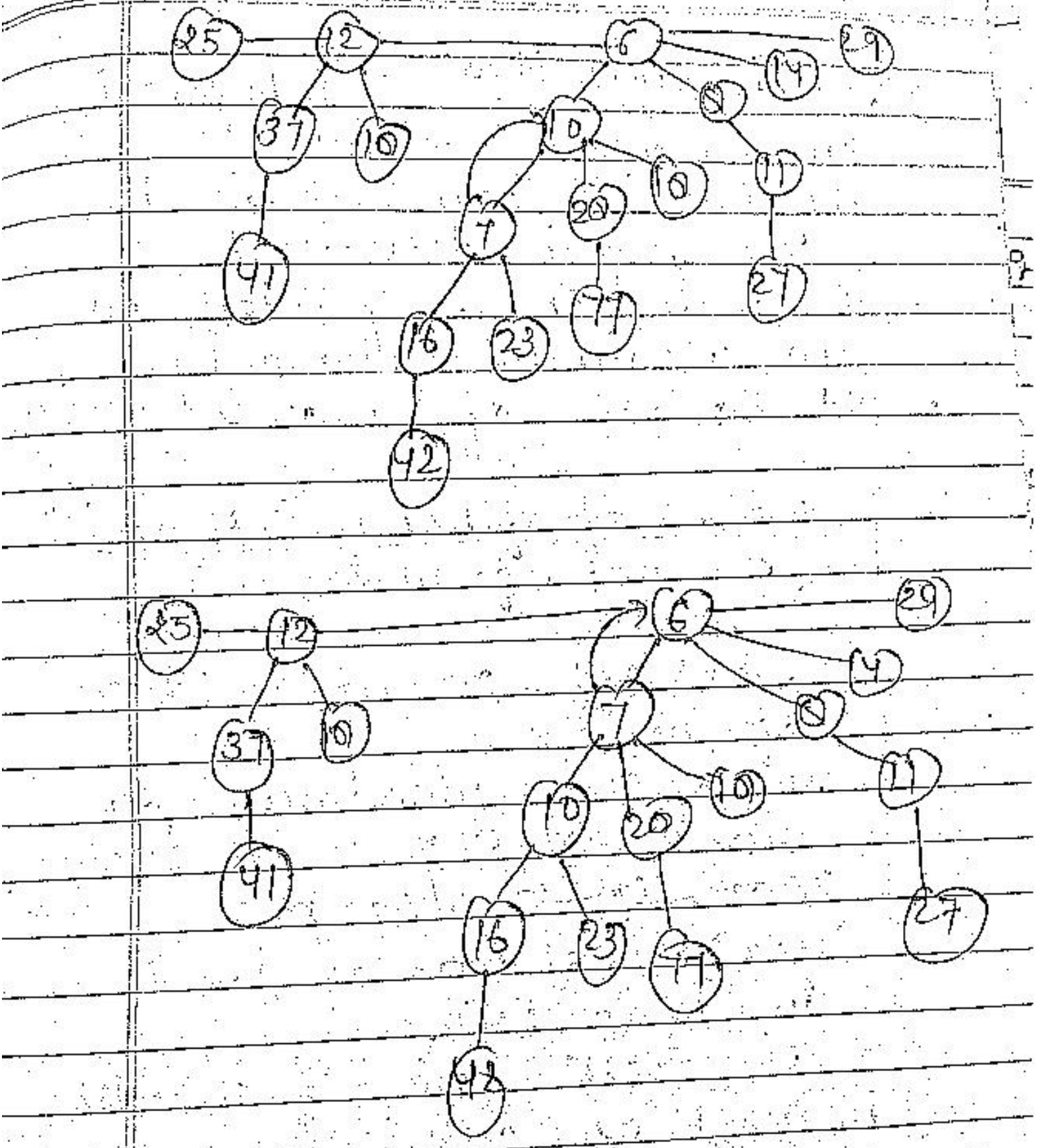
degree

Decrease key.



26 → 7




25
12
6
29
14
37
10
8
10
20
10
11
7
41
27
16
23
11
42

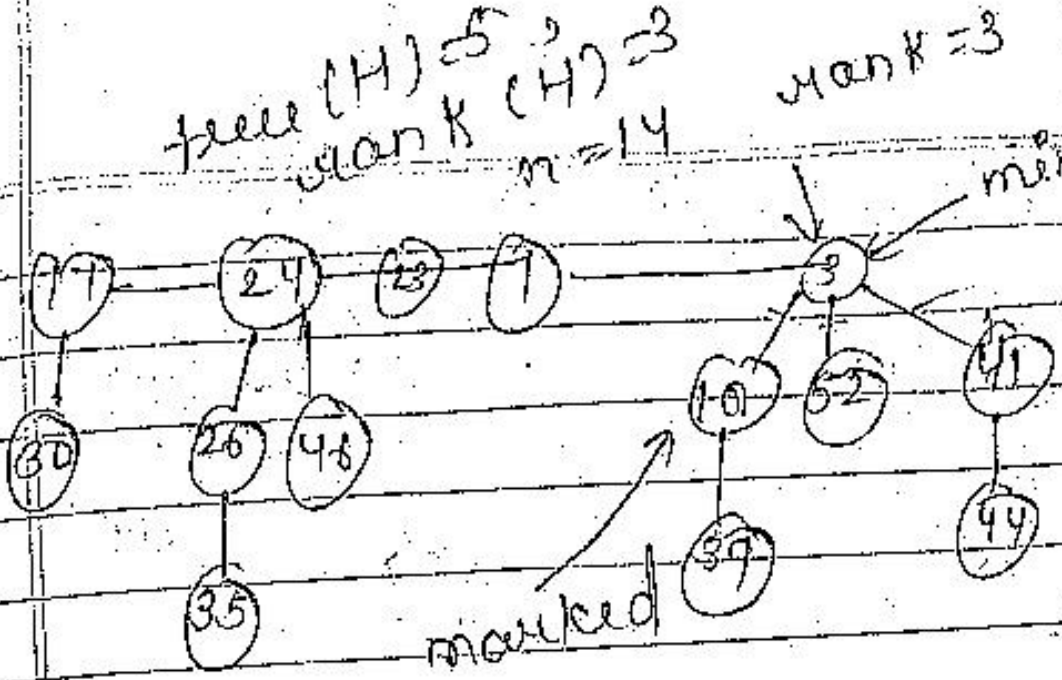
25
12
6
29
4
37
10
7
8
10
20
10
11
41
16
23
11
27
42

Delete binomial heap ⇒ do ourself

Fibonacci Heap ⇒ fibonacci Heap is very similar to binomial heap but it has less rigid structure.

→ It maintain a pointer to the minimum element.
→ It has a set of marked node.

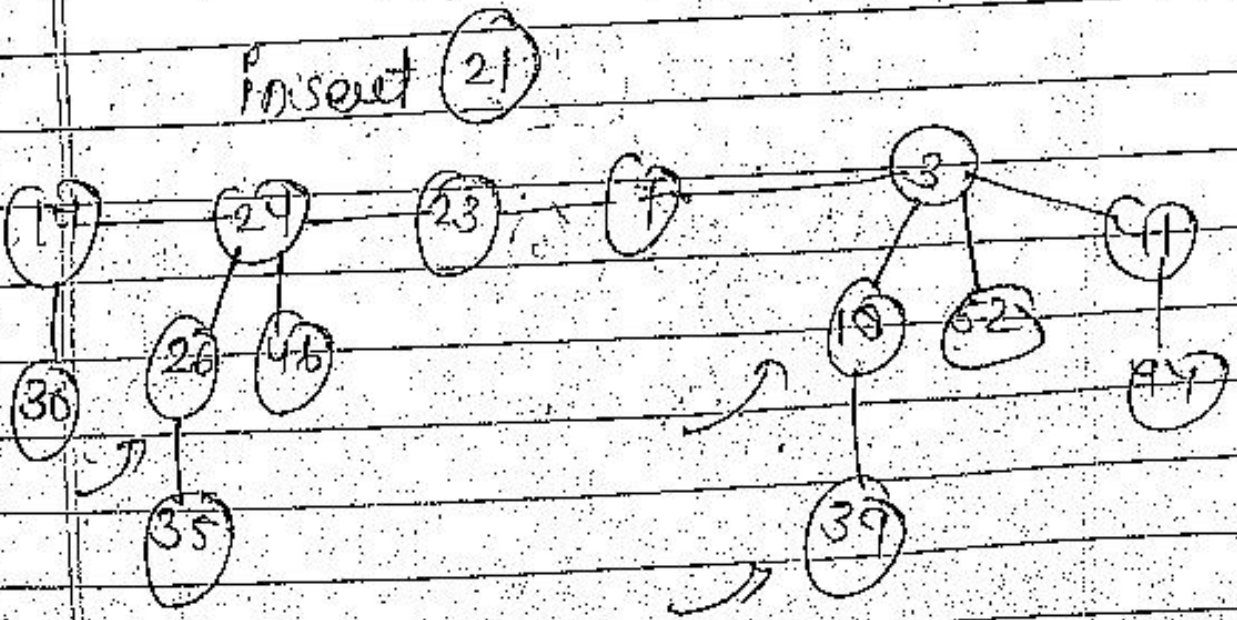
notations of fibonacci heap ⇒
n = no of nodes in a heap.
rank(x) = no. of children of node x.
rank(H) = maximum of rank of any node in a heap H
tree(H) = no. of tree in heap (H).
marks(H) = no. of marks in a Heap (H).
potential function
$\phi(n) =$ ~~Heap~~ tree(H) + 2 × mark(H)
$= 5 + 2 \times 2$
$\phi(n) = 9$

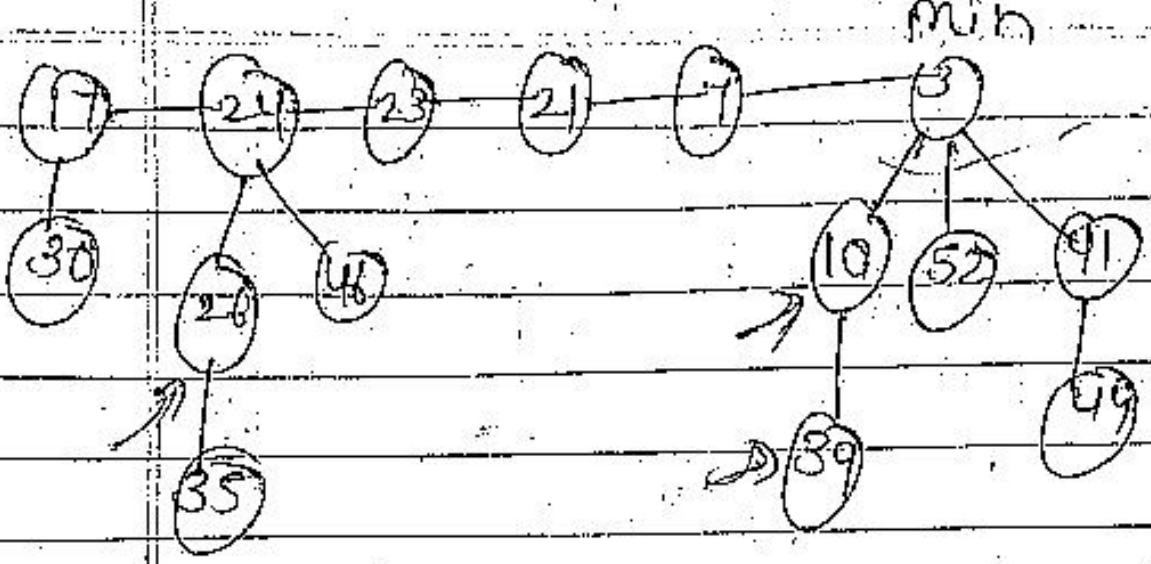


Operations on Fibonacci :-

→ Insertion :- Insertion in a fibonacci heap can be done in two way.

1. Create a new singleton tree.
2. Add to the root list and update minimum pointer & name it.

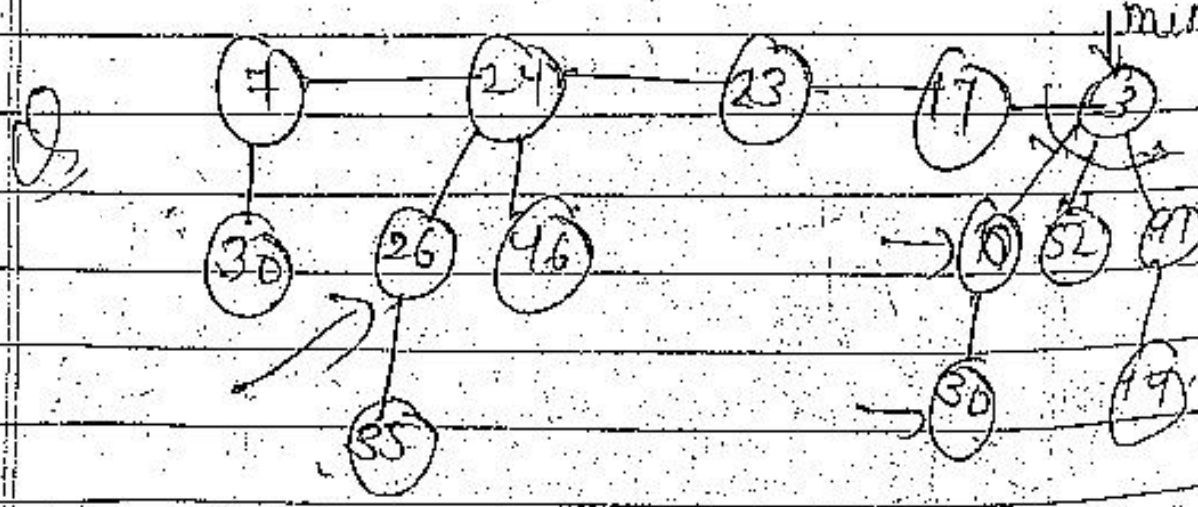


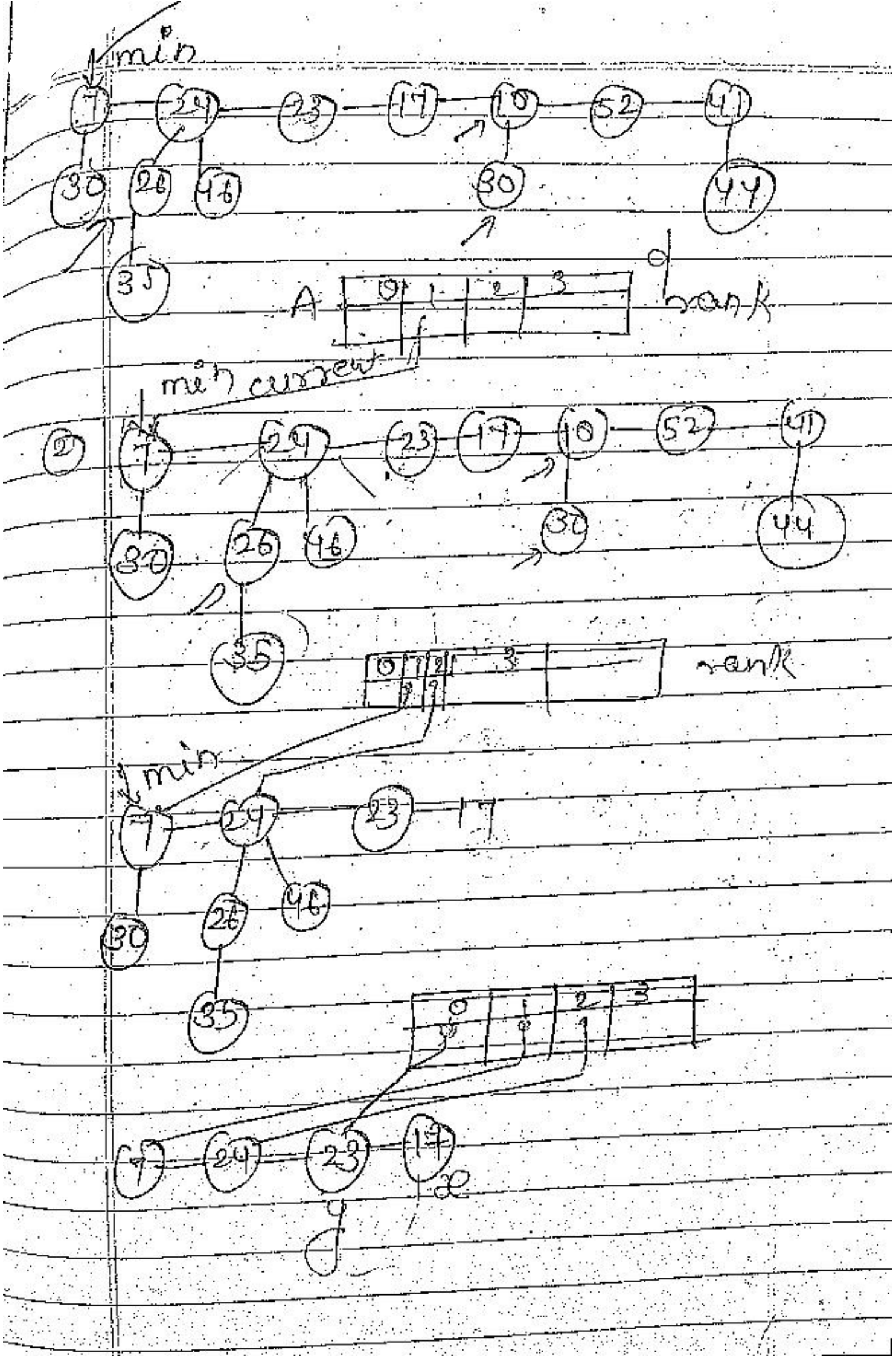


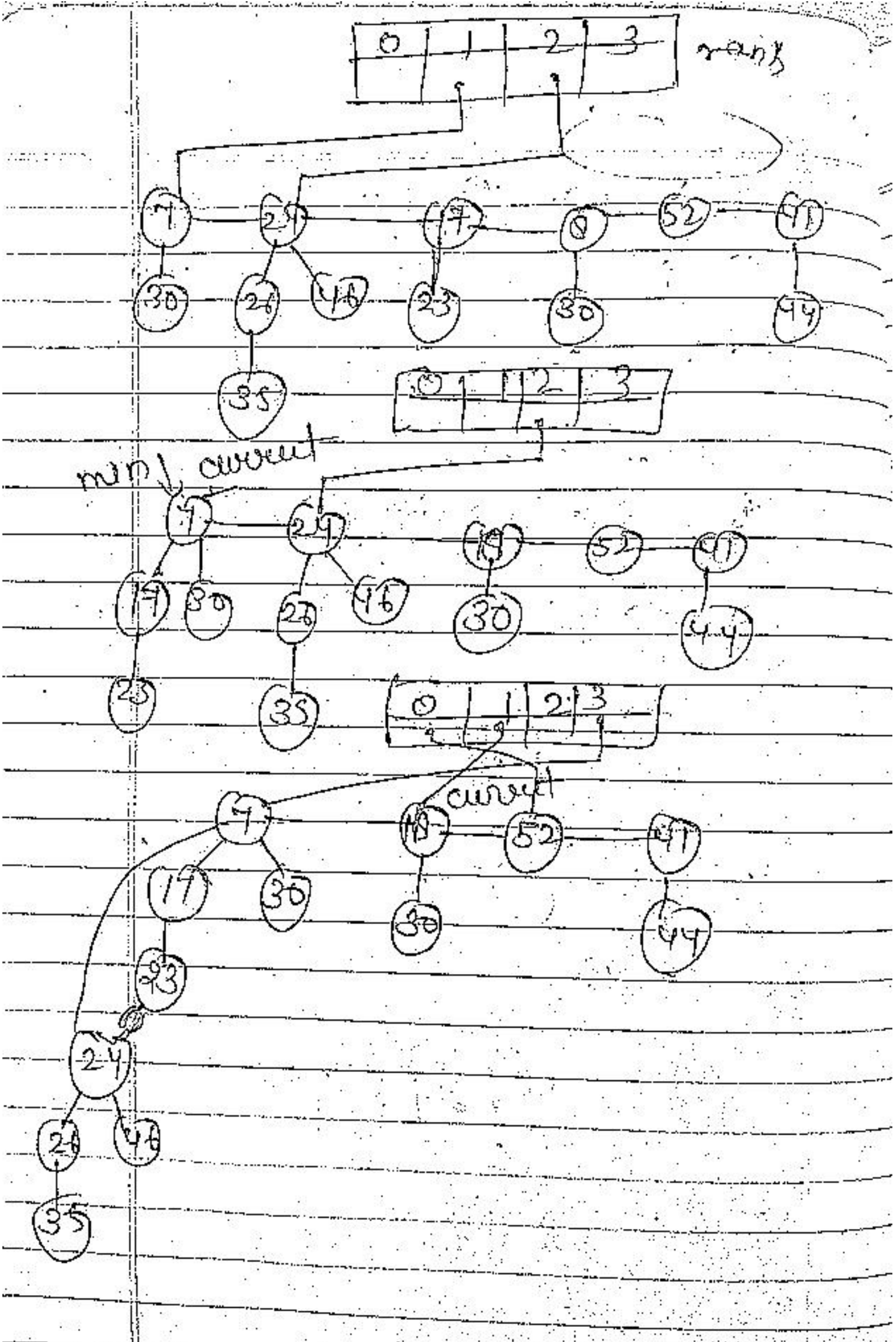
→ Delete minimum :⇒ Delete minimum can be done into two -

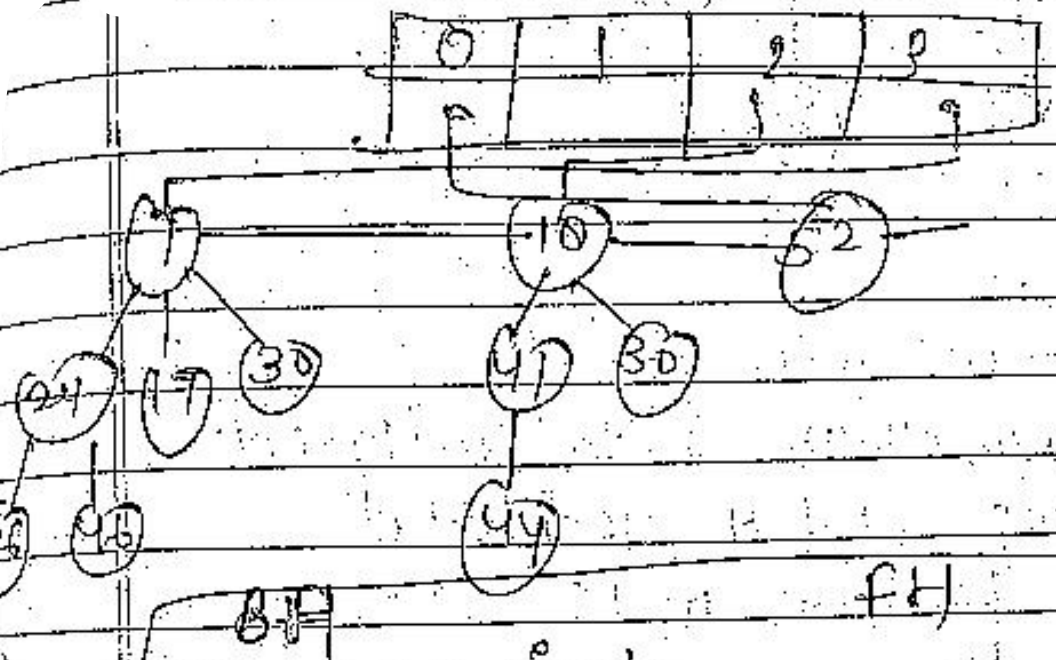
(1) delete minimum, & make its children into root list, and update minimum.

(2) consolidate tree so that no two roots have same rank.




min
7
24
23
17
10
52
41
30
26
46
30
44
35
A
0
1
2
3
bank
min current
7
24
23
17
10
52
41
30
26
46
30
44
35
rank
min
7
24
23
17
30
26
46
35
7
24
23
17


0 1 2 3
rank
7 24 7 8 52 41
30 26 46 23 30 44
35
0 1 2 3
min
current
7 24 18 52 41
17 30 26 46 30 44
23 35
0 1 2 3
current
7 18 52 41
17 30 30 44
23
24
26 46
35



BH FH

(1) BHs use a singly linked circular link list & fibonacci heaps use a doubly linked circular linked list

(2) every BH is a fibonacci heap

(3) delete-min in BHs involves the combining of trees.

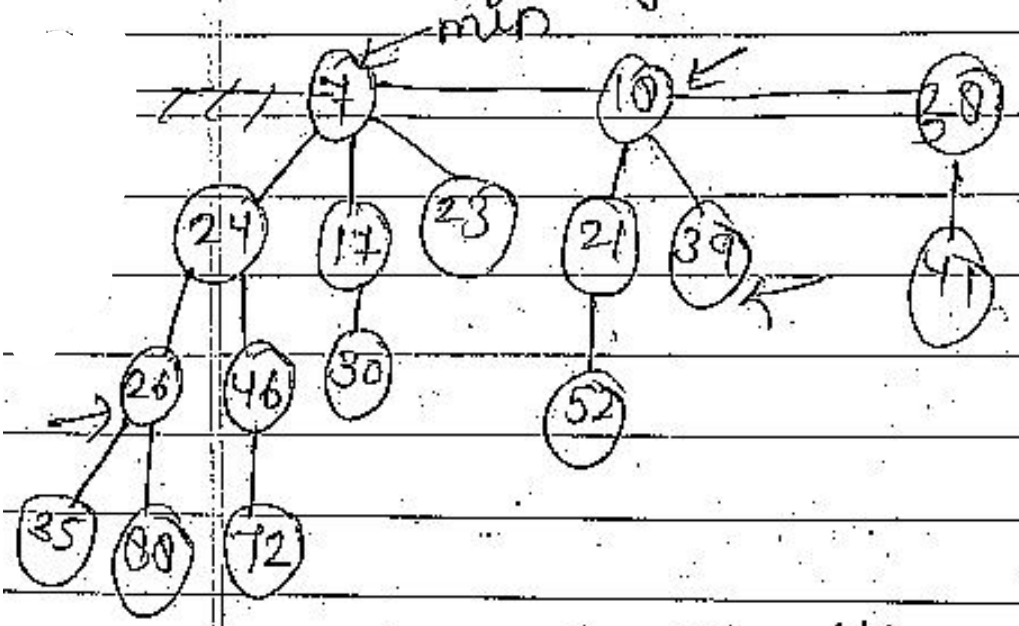
(4) Data members in a node for Binomial heaps are data, link, degree and child,

(2) but every fibonacci heap isn't BH.

(3) whereas delete min or delete in FHs is performed without joining the trees obtained after deletion

(4) whereas in fibonacci heaps we have data, parent, child, child link, degree.

→ Decrease key :-



two main condⁿ

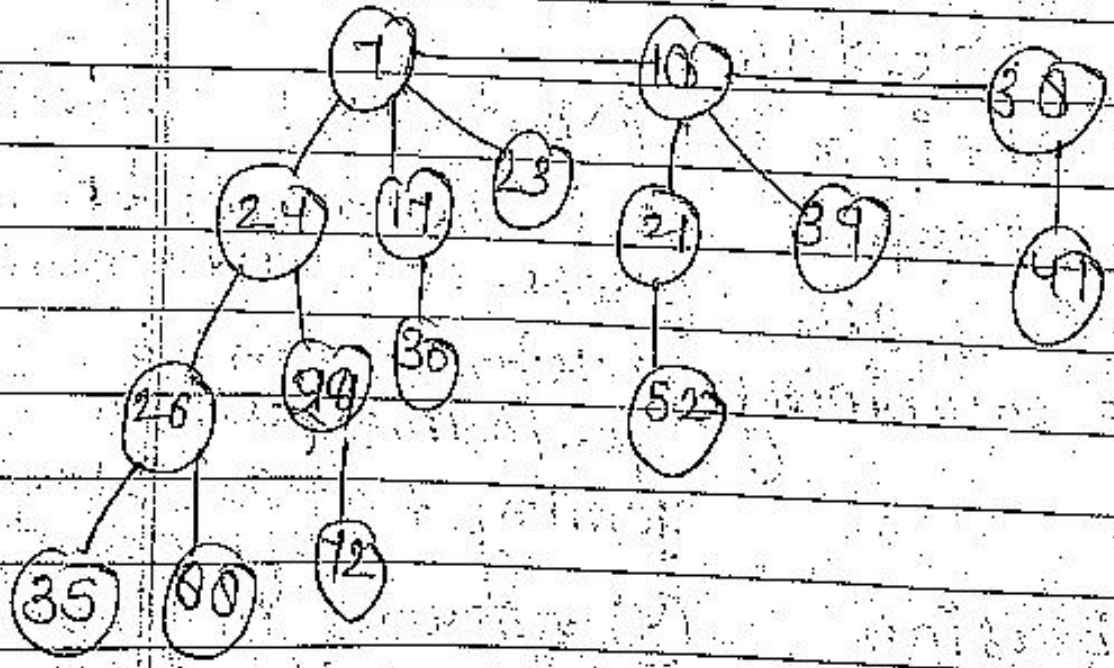
(1) Heap order not violated

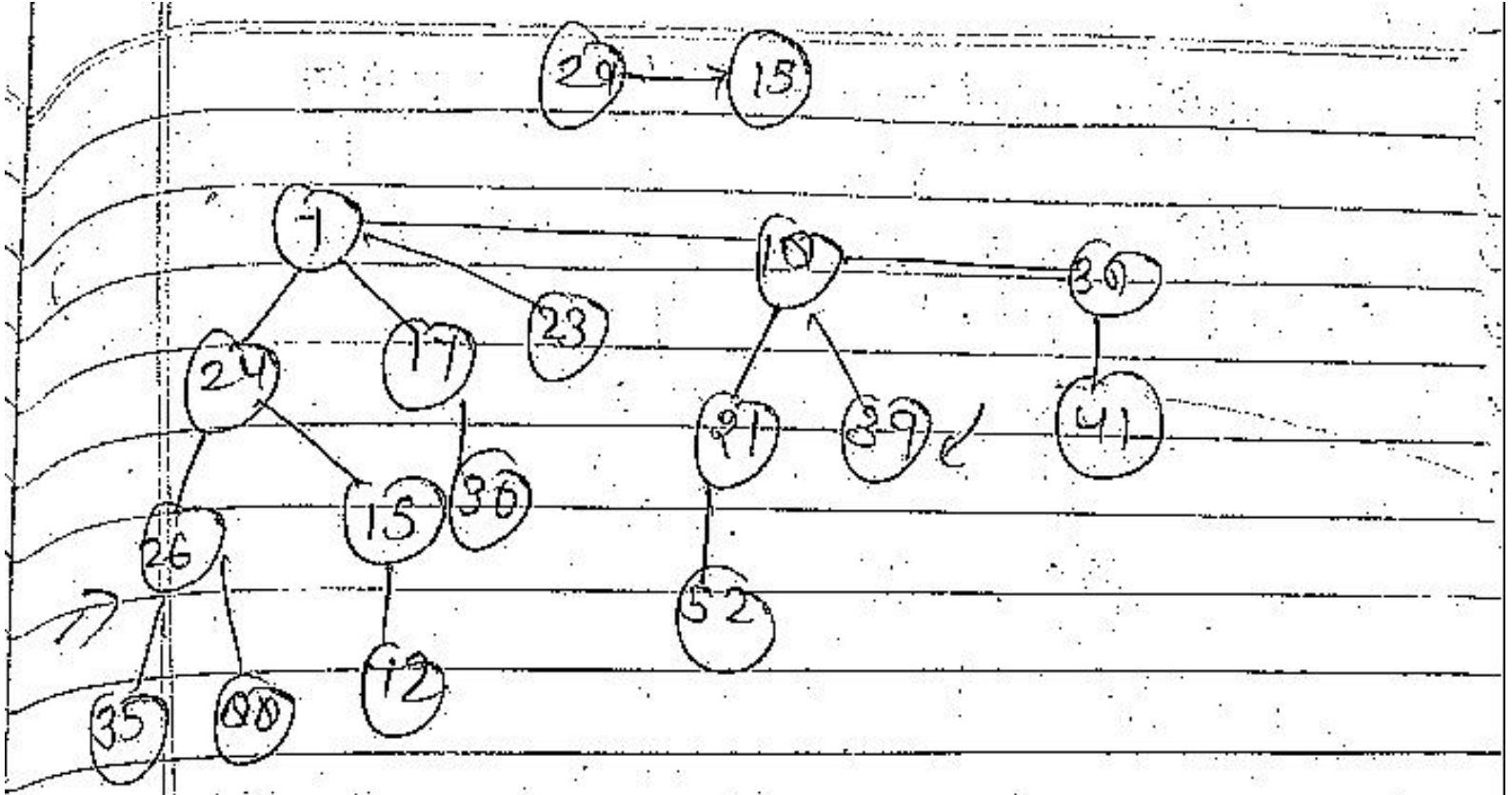
(2) Heap order violated

— mark node

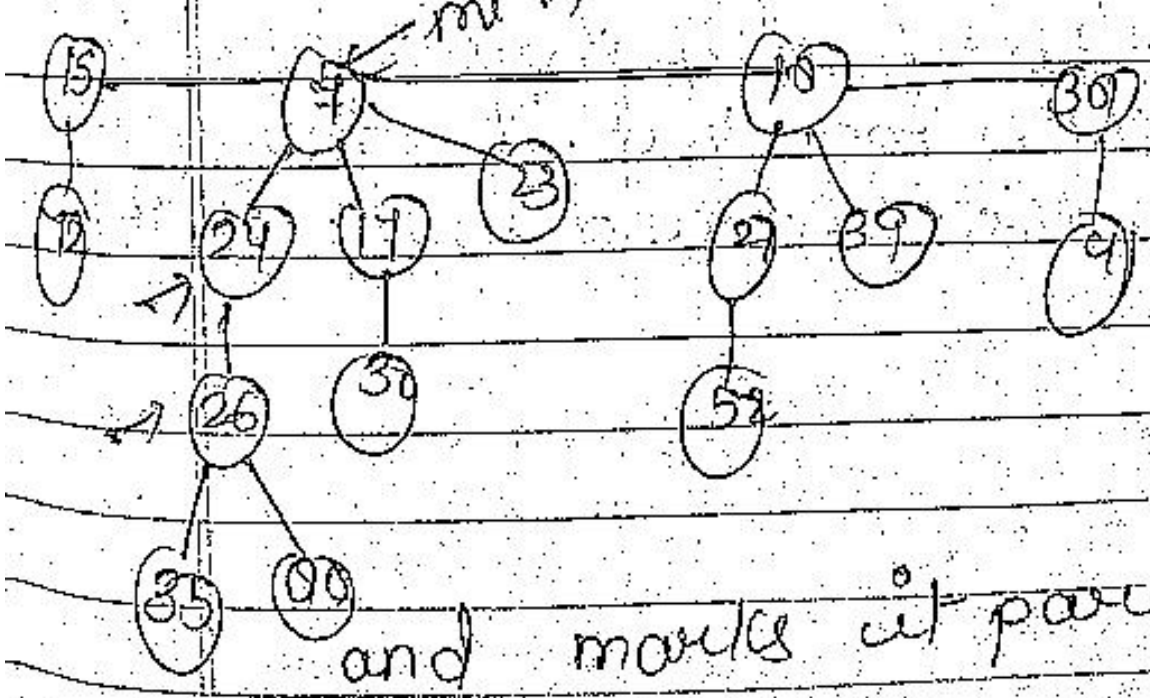
— Not a mark node

decr 46 — 24

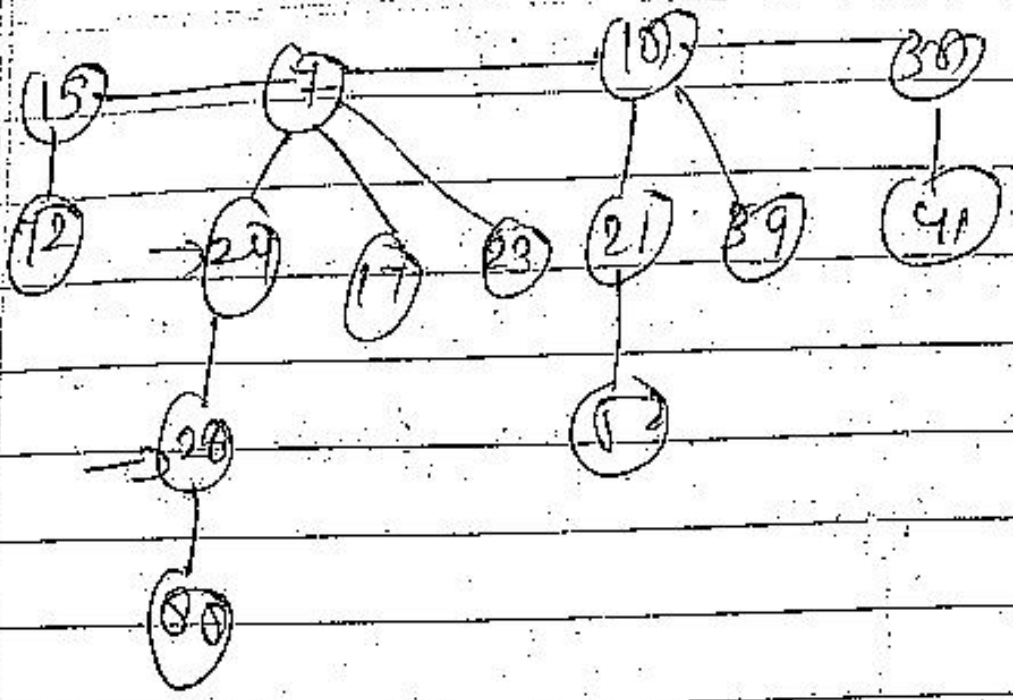




if decrease node is node a max key & its parent also not a mark node then cut it & add to the root list.



and marks it parent.



If such a case exist if the parent is mark node then cut the child & add it to the root list
make parent as unmarked

(1) if violated the property then cut and make it root list and if parent of x is unmarked than marked it. otherwise if marked then cut and make it into root list.