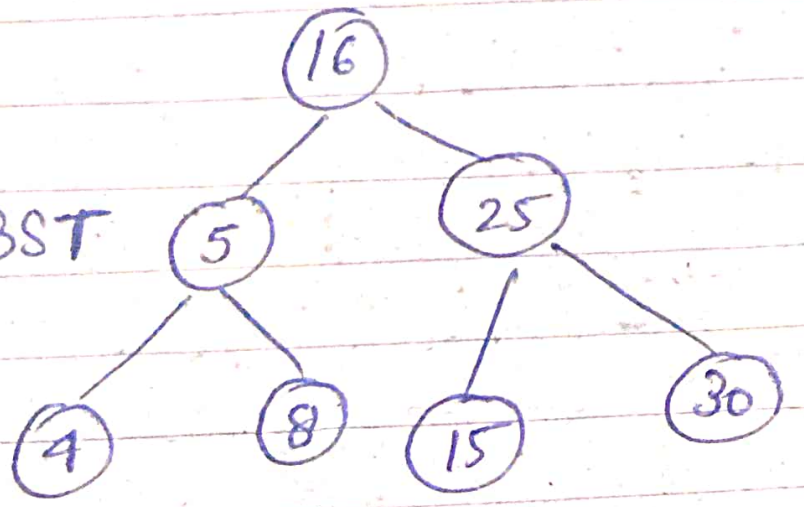
Red-Black Tree

$h = \log_2 n$

BST :-

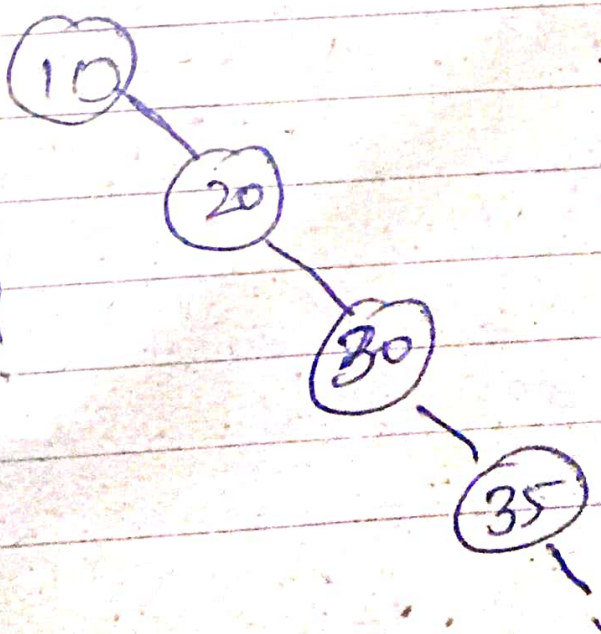


balanced BST

T.C = O(h)

T.C. = O($\log_2 n$)

T.C. = O(lg n) Searching Insertion deletion



skewed bst

T.C. = O(h)

T.C. = O(n)

Algo + Example + theory

AVL :- T.C = O(log n) (BST)

Searching, insertion, deletion → best, average & worst all cases

AVL is also self balancing (height) BST
⇒ but if need more no. of rotation if it want to perform insertion & deletion operation in AVL tree (10-20 operation)

→ AVL tree is more balanced than red-black tree
⇒ for some thing red black tree require atmost 2 rotation. & Sometimes recoloring is only required for th

If AVL have less rotation then it is pereffere over red-black tree as have less T.C.

⊙ Red Black tree is roughly height balanced
⊙ AVL trees is strictly hey balanced tree

Red Black Tree

A self-balancing BST with an additional attri-bute for its nodes: color which can be red or black. red-black tree guartees us to give the time complexity. order of log n for all the searching, insertion & deletion etc.

→ Searching is faster in AVL tree as it is strictly balanced tree
Insertion & deletion is faster in red-black tree as it requires very few rotation

Intro to Red - Black tree :-

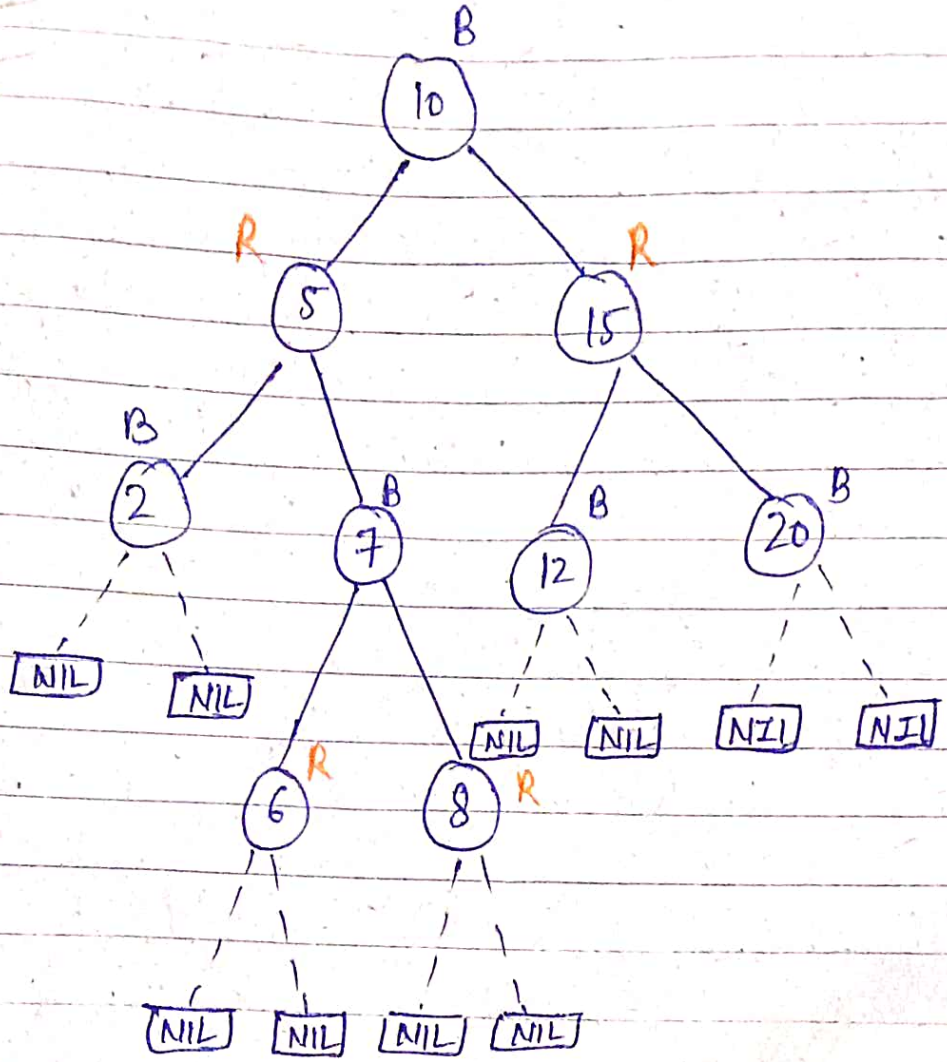
- → It is a self-balancing root.
- → Every node is either Black or Red.
- → root is always ~~root~~ Black.
- → Every leaf which is Nil is Black.
- → If node is Red then its children are Black.
- → Every path from a node to any of its descendes descendent NIL node has same no. of Black nodes

If node is Black then its children can be red & black both

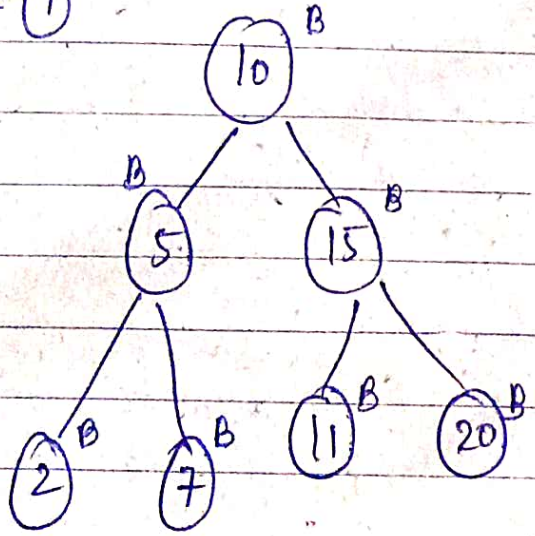
If a tree is a AVL tree & we color it red & black according to rules then it would be a red black tree. AVL trees are subset of red black trees. But if a tree is a red black tree then it is not true to be AVL tree (by removing coloring) because red black trees are roughly height balanced & AVL is strictly height balanced.

** - the longest path from root to its leaf node can't be more than twice the length of the shortest path.

Ex:-

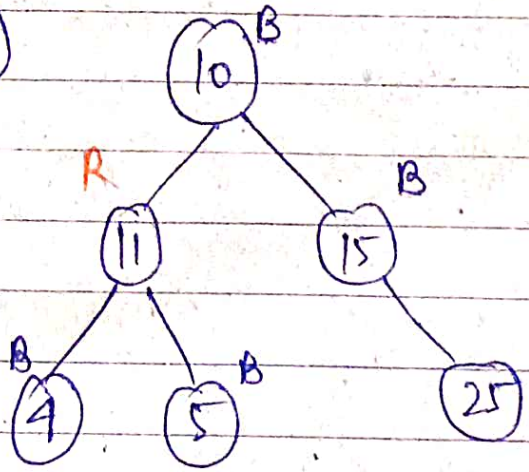


Ex:- (1)



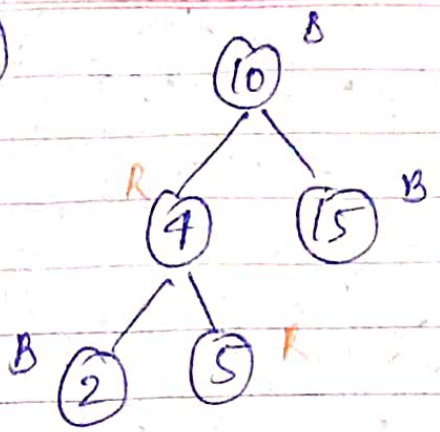
Red-black tree
(perfect B.T. having only black coloured node)

(2)



(Not B.ST so not Red-Black tree)

(3)



(Not Red-black tree as red parent have red children)

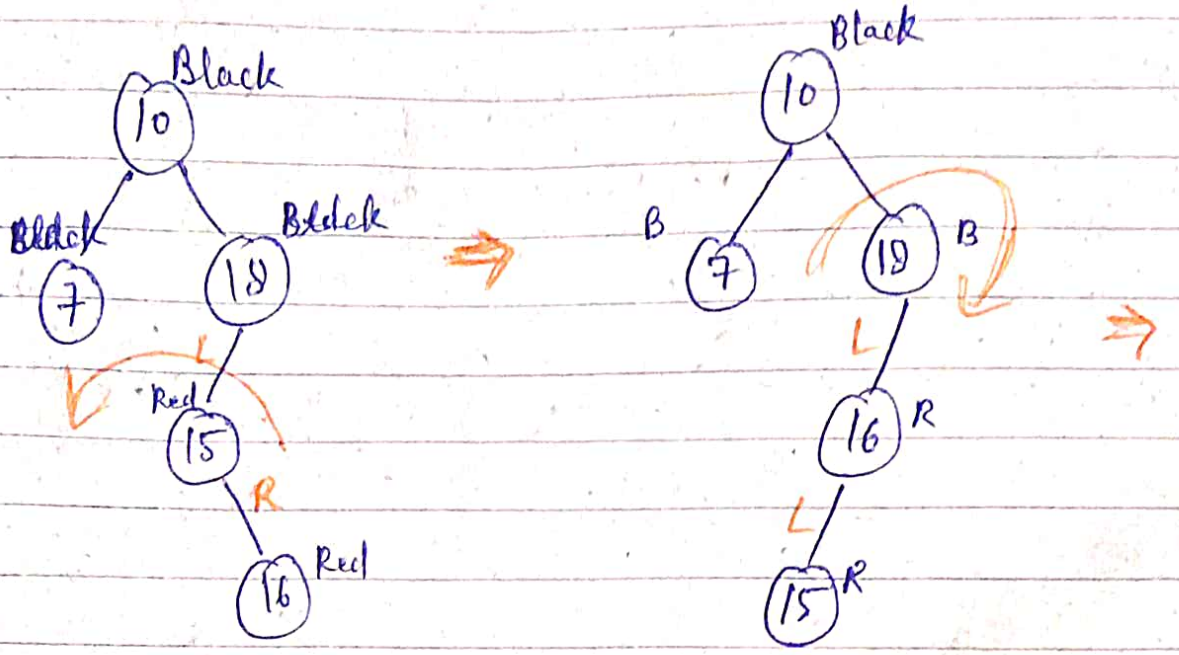
Insertion in Red-Black Tree :-

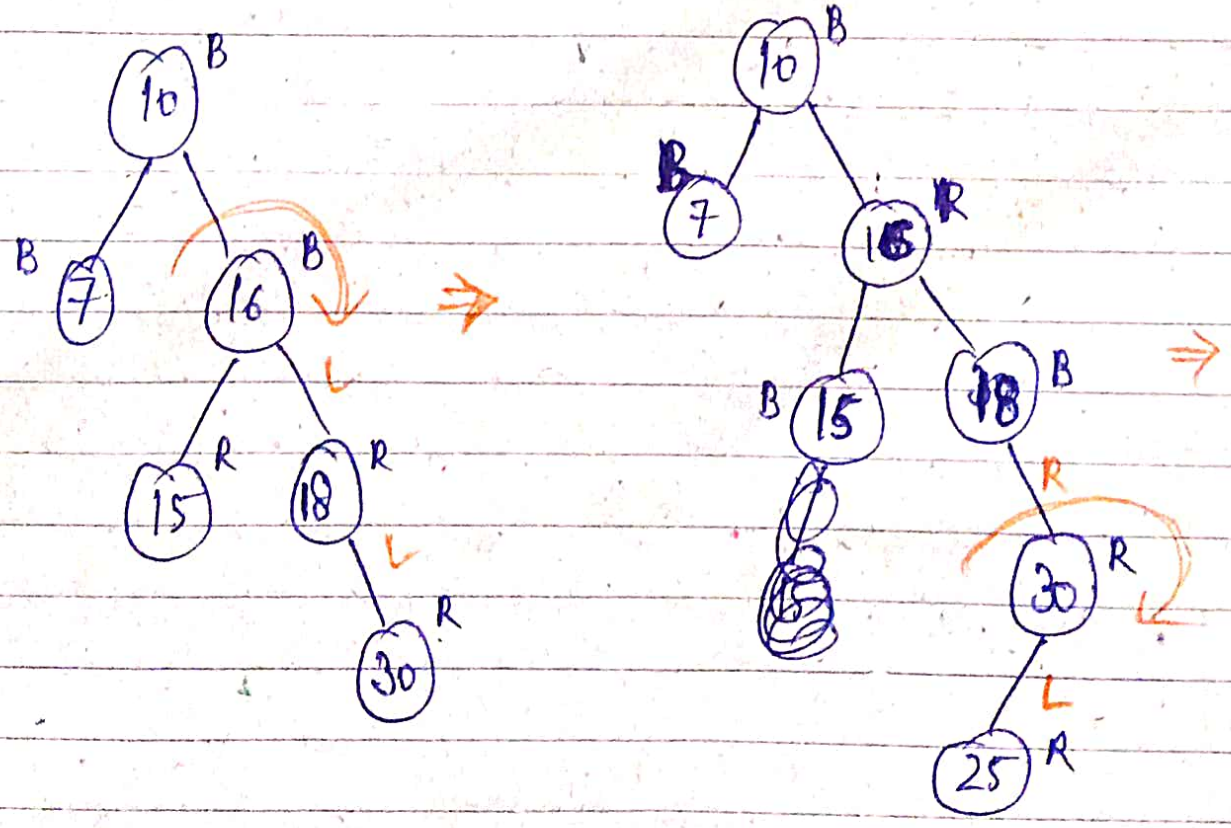
10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70

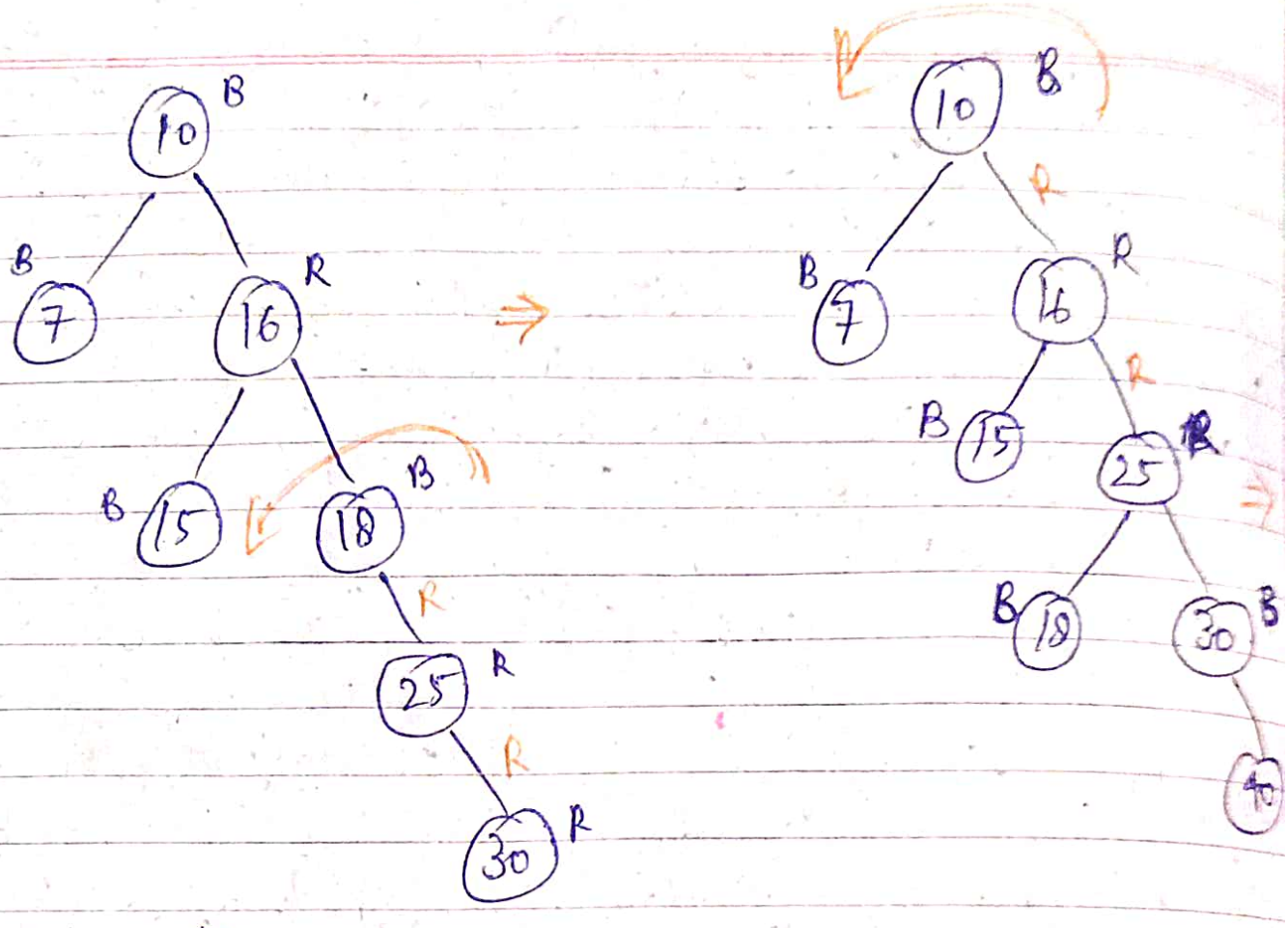
(1) If tree is empty, create newnode as root node with color black.
(2) If tree is not empty, create newnode as leaf node with color Red.
(3) If parent of newnode is black then exit.
(4) If parent of newnode is Red, then check the color of parent's sibling (uncle) of newnode

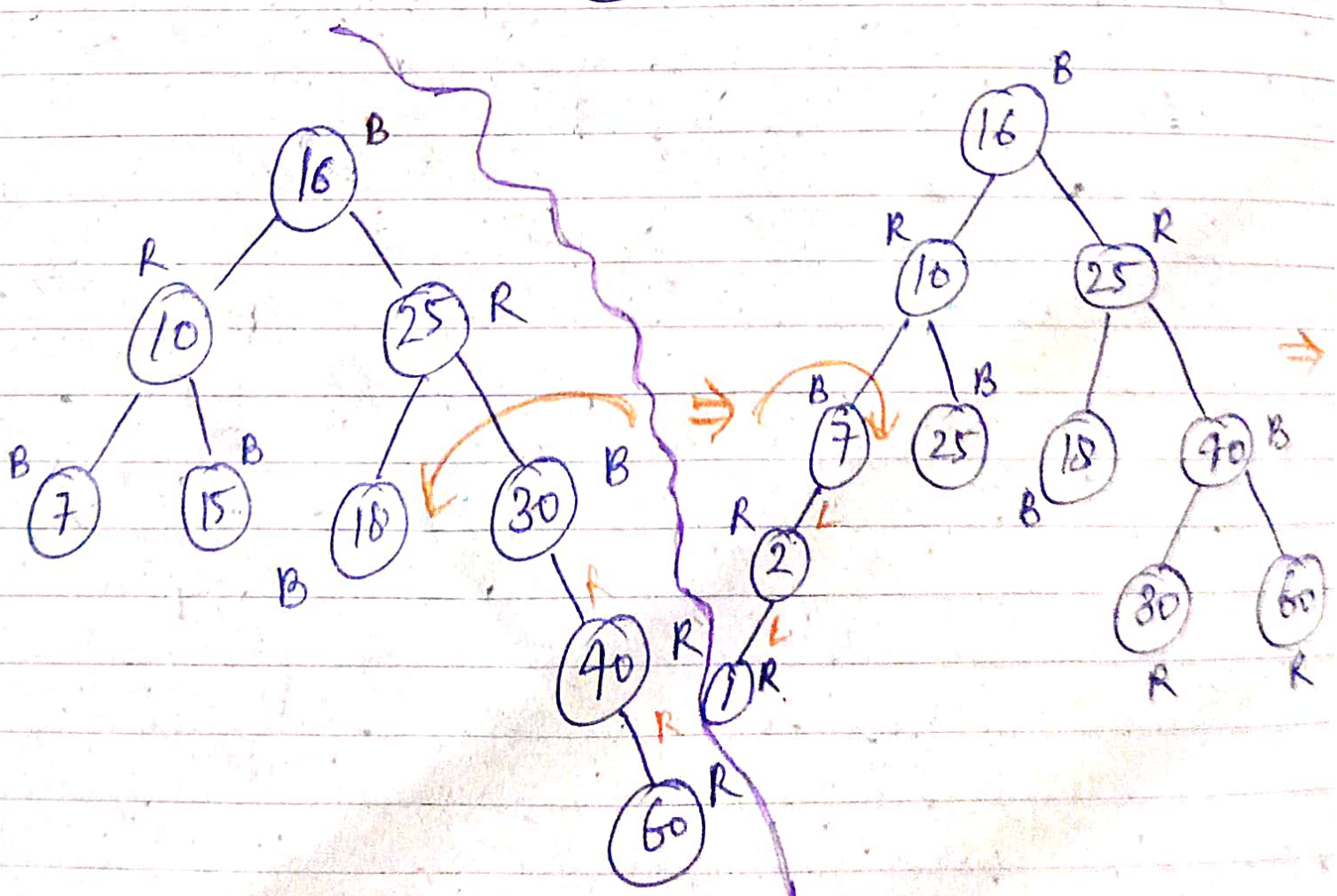
(a) If color is black or null then do suitable rotation & recolor
(b) If color is Red then recolor parents & its sibling & also check if parent's parent (grandparent) of newnode is not root node then recolor it & recheck for red-red conflict

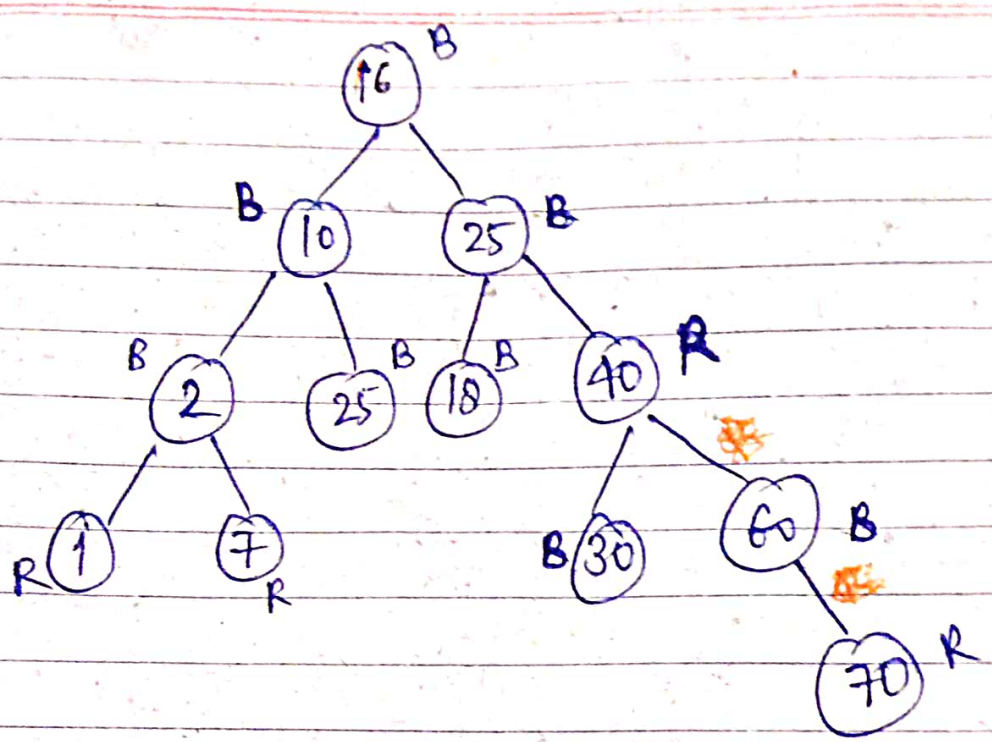
→ Root = Black
→ no two adjacent Red nodes
→ Count no. of black nodes in each path






10 B
7 B
16 R
15 B
18 B
25 R
30 R
10 B
7 B
16 R
15 B
25 R
18 B
30 B
40

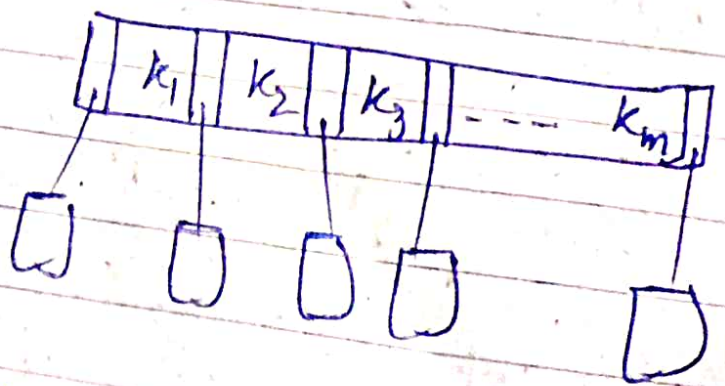

16 B
10 R
25 R
7 B
15 B
18 B
30 B
40 R
60 R
16 B
10 R
25 R
7 B
25 B
18 B
40 B
2 R
30 R
60 R



final Red-black tree

B-tree

- balanced (m)-way tree (m → order)
- Generalization of BST in which a node can have more than one key & more than 2 children
- maintains sorted data
- all leaf node must be at same level.
- B-tree of order m has following properties:
  - → Every node has max m children
  - → Min. children :- leaf → 0
    root → 2
    internal nodes → $\lceil \frac{m}{2} \rceil$
  - → Every node has max (m-1) keys
  - → Min keys :- root node → 1
    all other node → $\lceil \frac{m}{2} \rceil - 1$



⊙ $k_1, k_2, k_3 \dots k_n$ are keys

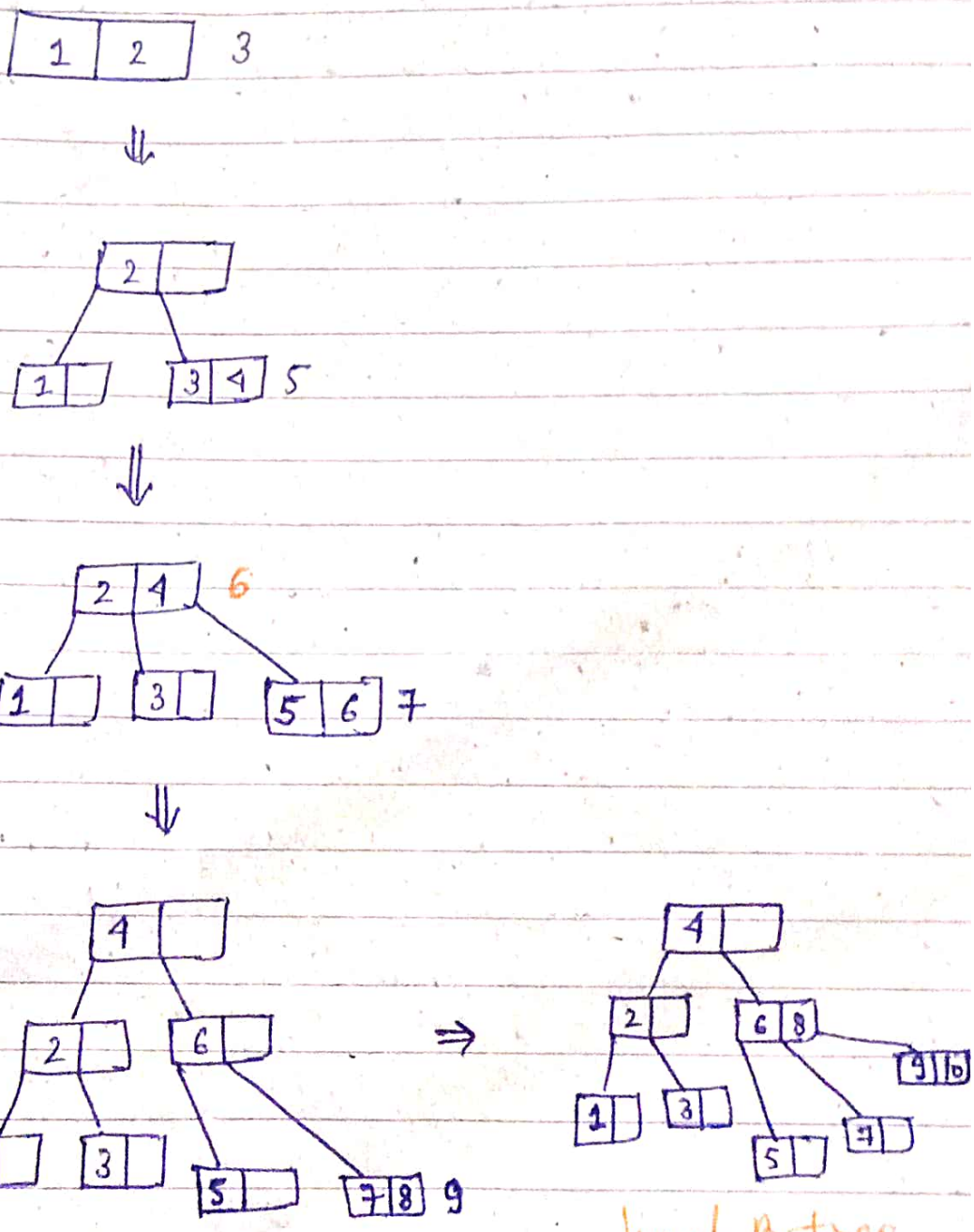
Ex:- If m = 5

$\lceil \frac{m}{2} \rceil = \lceil \frac{5}{2} \rceil = \lceil 2.5 \rceil$

Siting of $\frac{m}{2}$ = 3

Qm Create a B-tree of order 3 by inserting values from 1 to 10

$m = 3$, max. key $= (m-1) = 3-1 = 2$

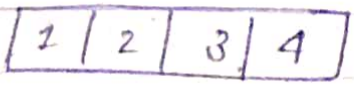


final B-tree

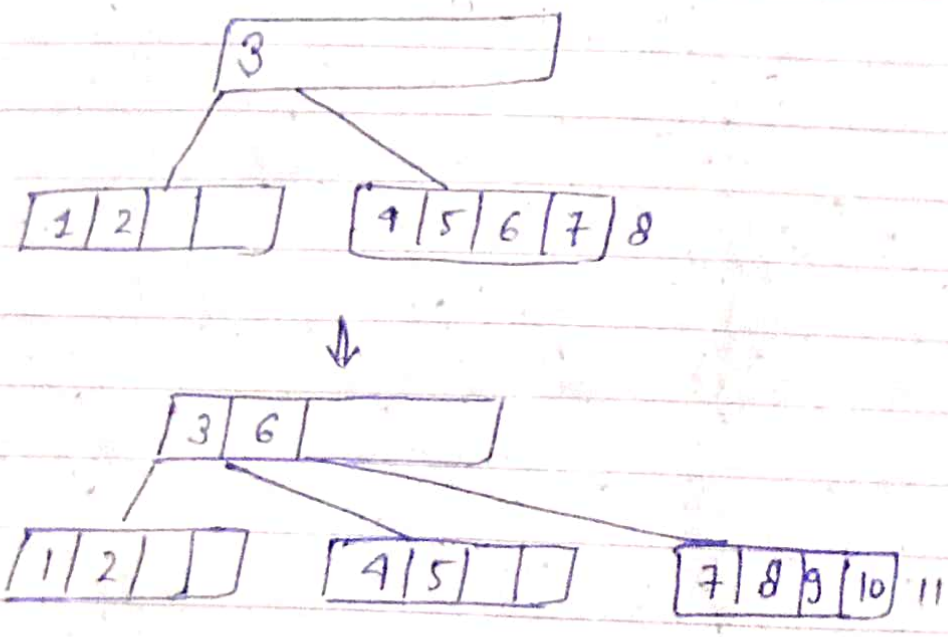
Ques: Create a B-tree of order 5 by inserting values from 1 to 20.

$m = 5$, max. Key $= (m-1) = 5-1 = 4$

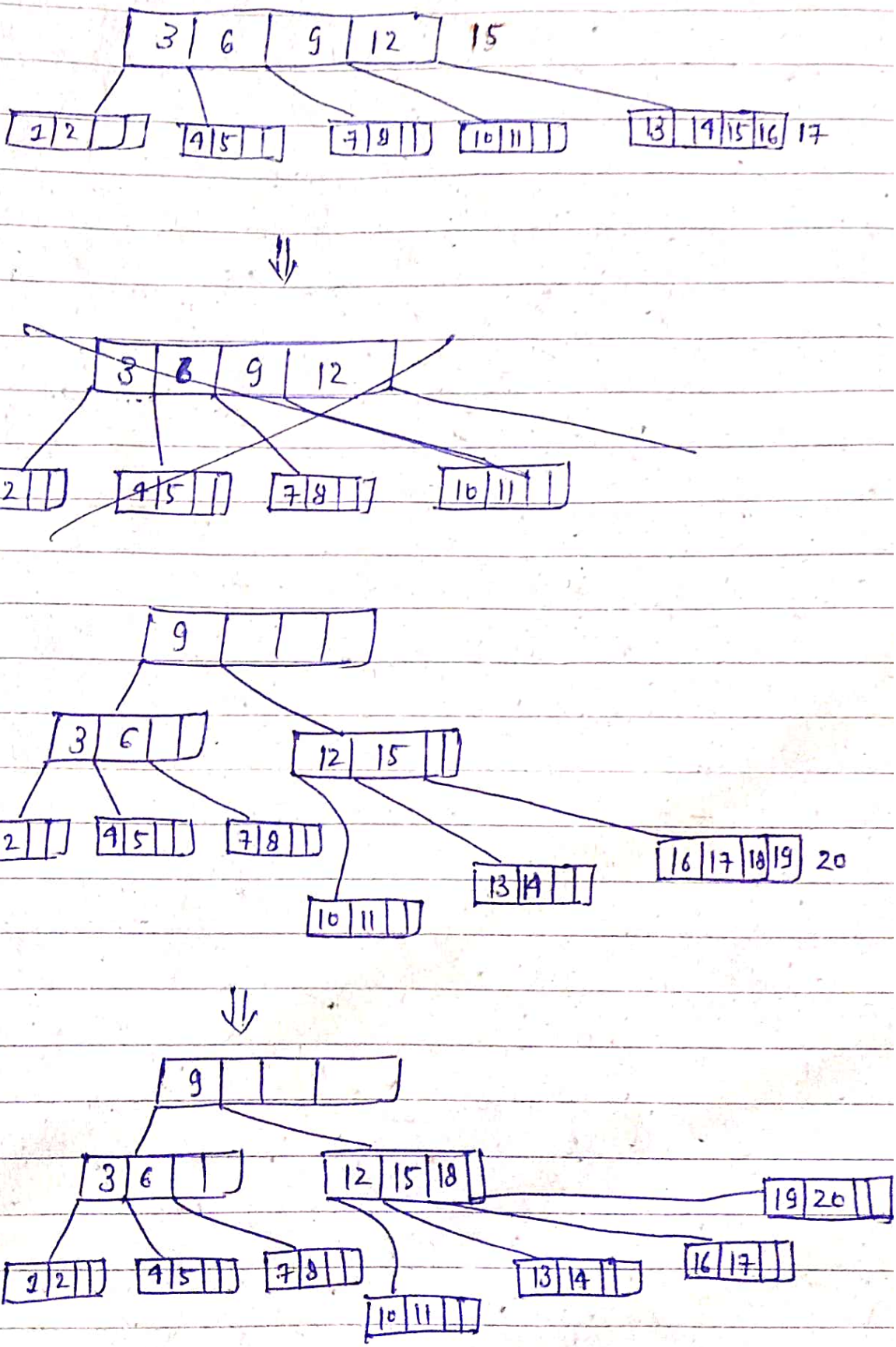
max. children $= 5$



⇓ median will go up & split below elements.



⇓


3 6 9 12 15
1 2
4 5
7 8
10 11
13 14 15 16 17
3 6 9 12
1 2
4 5
7 8
10 11
9
3 6
12 15
1 2
4 5
7 8
10 11
13 14
16 17 18 19 20
9
3 6
12 15 18
1 2
4 5
7 8
10 11
13 14
16 17
19 20

Jnal B-tree

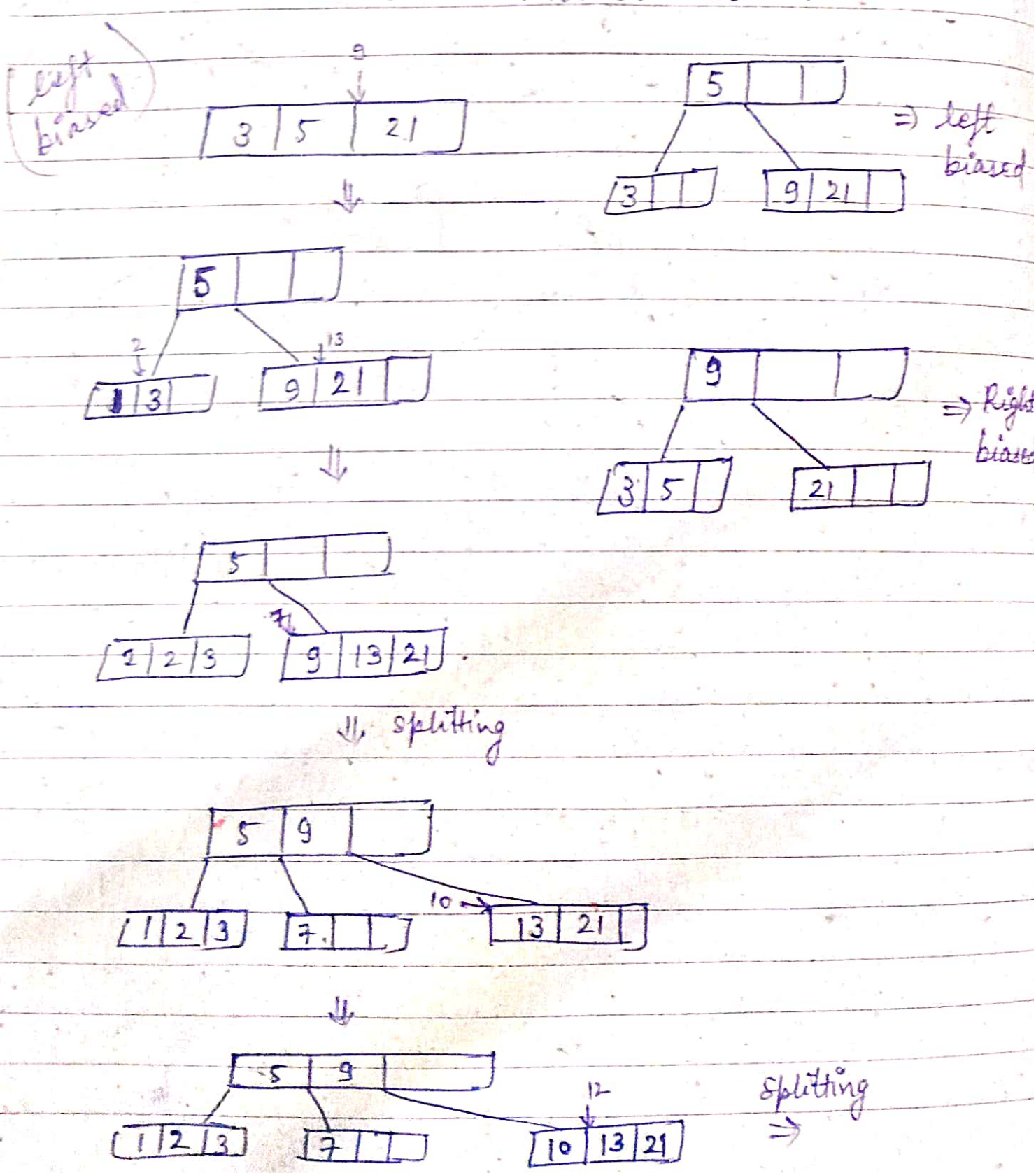
Ques Create a B-tree of order 4 with following set of data 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

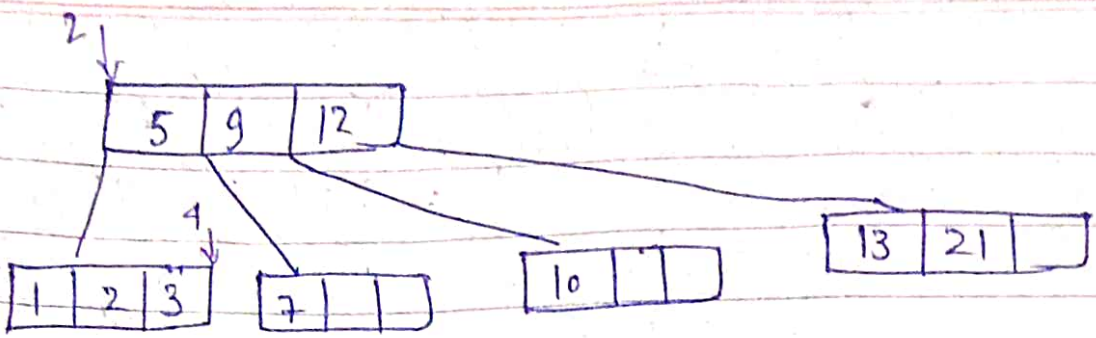
(key number is always be inserted in leaf node not in root node or internal node)

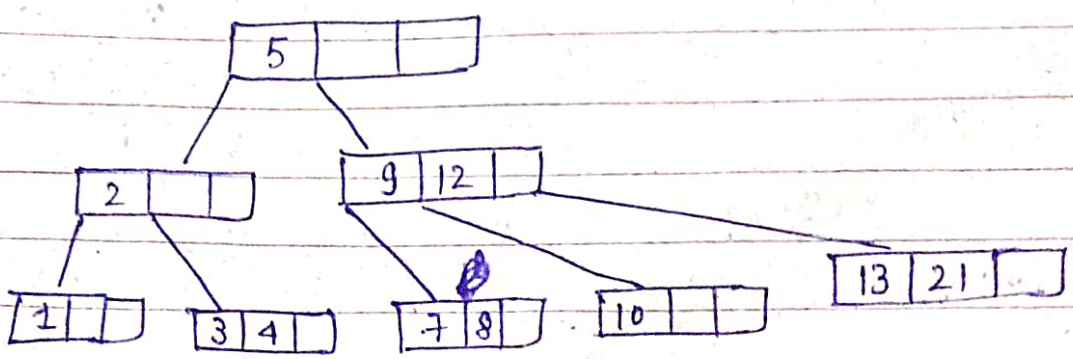
$m = 4$

max. key = $m - 1 = 4 - 1 = 3$

max. children = $m = 4$

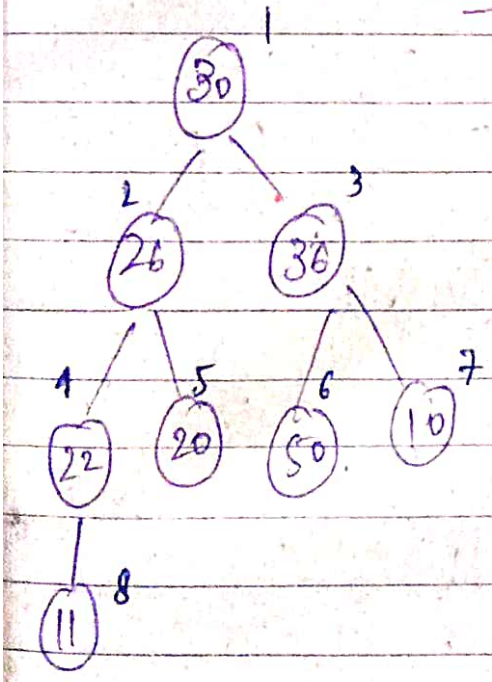






final b-tree of order-4

Heap



30	26	36	22	20	50	10	11
1	2	3	4	5	6	7	8

a at i

left child = $2i$

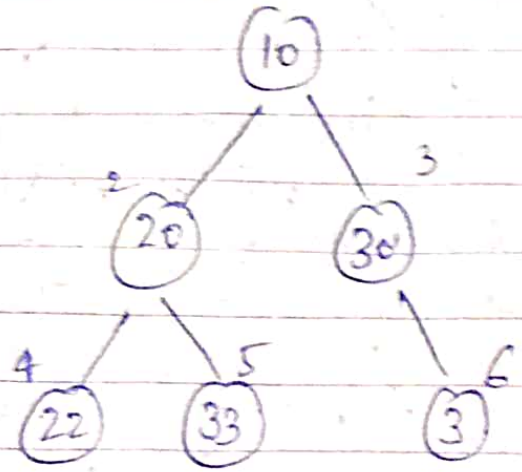
Right Child = $2i+1$

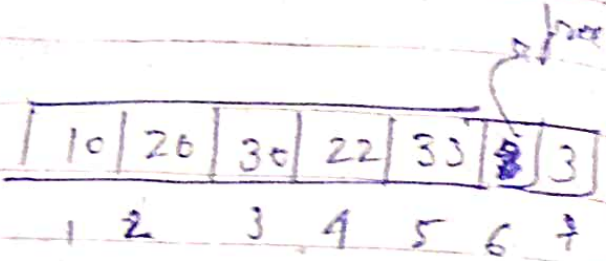
Parent of i = $\lfloor \frac{i}{2} \rfloor$

Complete binary tree

Array's space is free + excluding last level ~~+ Array's space~~

(as have 2^k element (where k = level of tree) in each level).





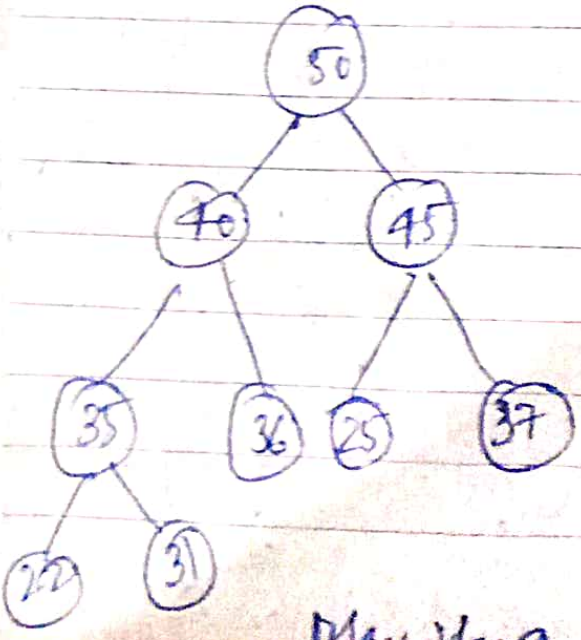
So no complete binary tree

Heap:- It is always a complete B.T. key value of each node is either greater than or less than their children

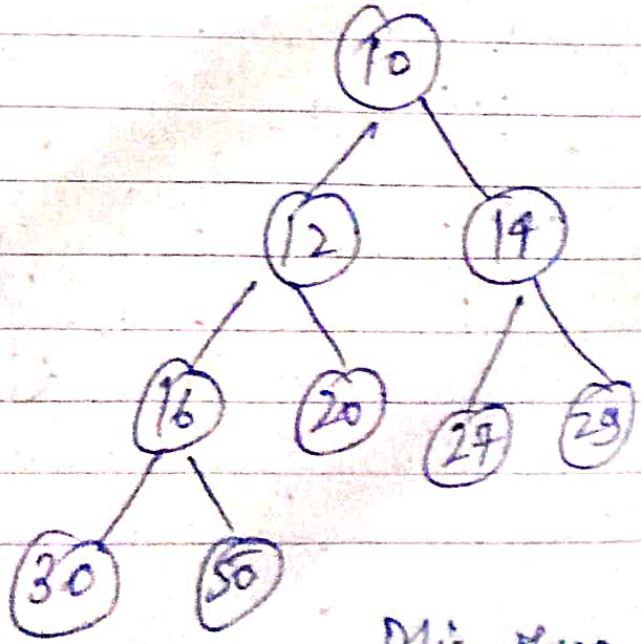
Two type of Heap:-

(1) Max Heap : Each node have greater key value than their children

(2) Min Heap : Each node have lesser key value than their children



Max Heap

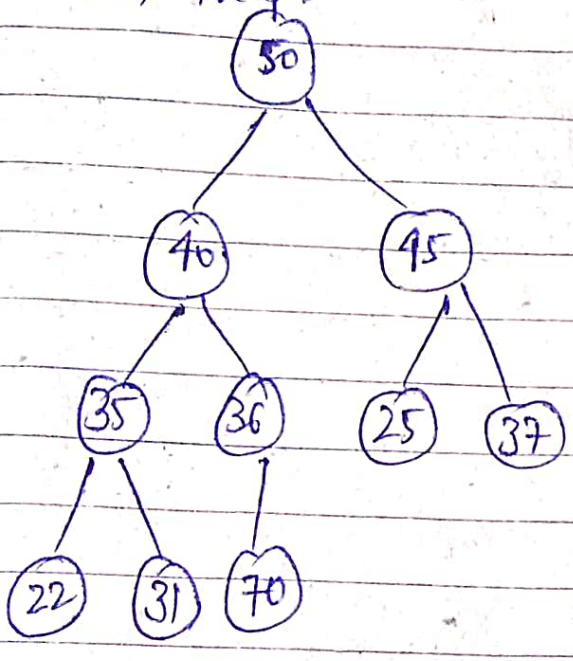


Min Heap

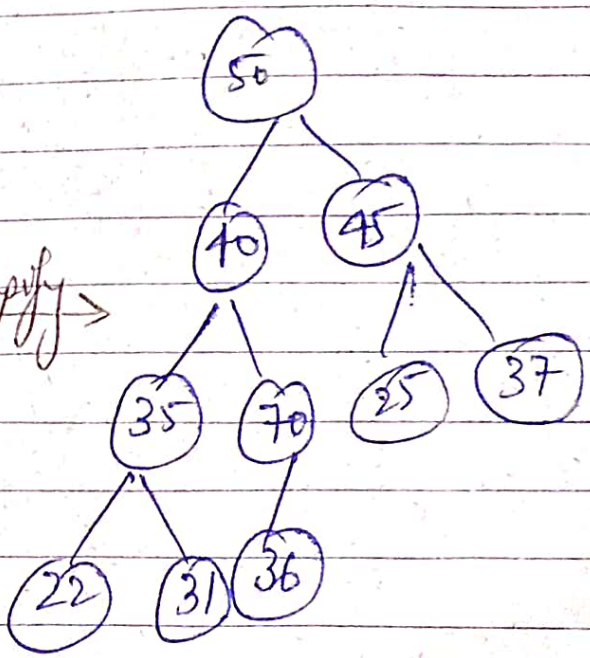
Insertion :-

Insertion will always take place at leaf

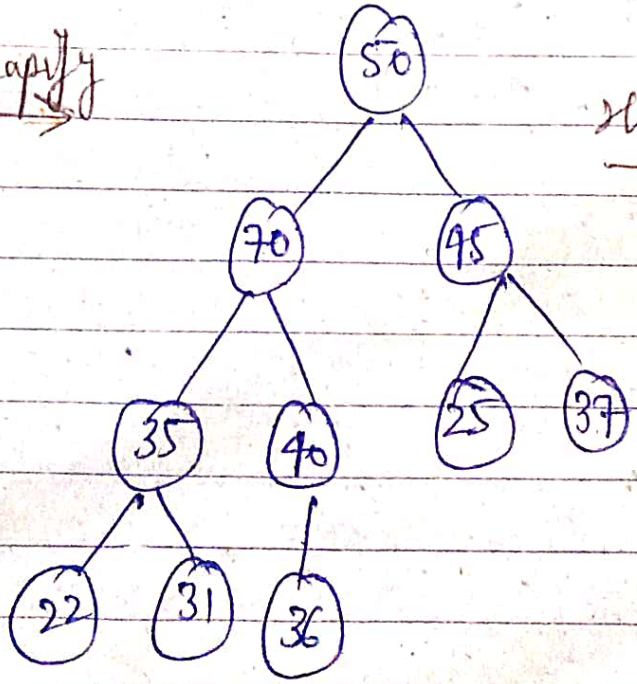
Max-Heap :-



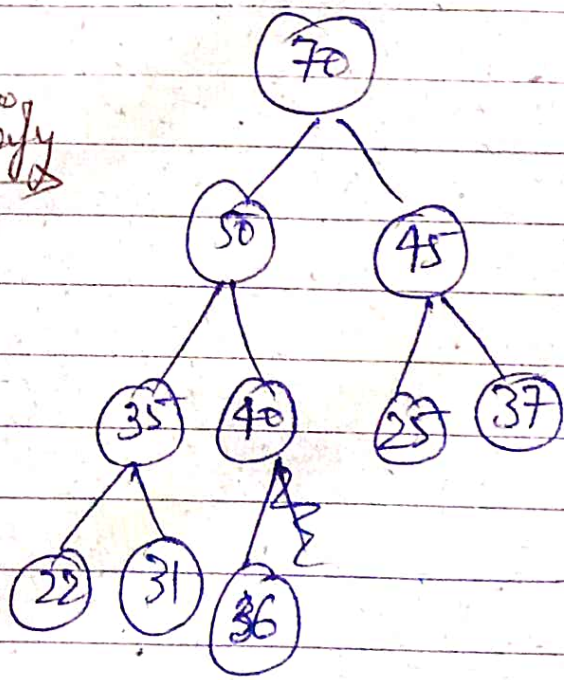
Heapify →



Heapify →

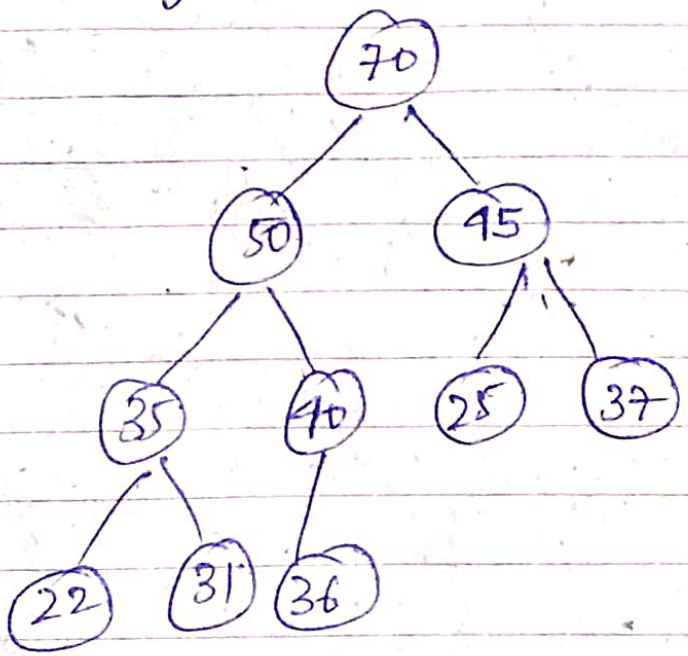


Heapify →

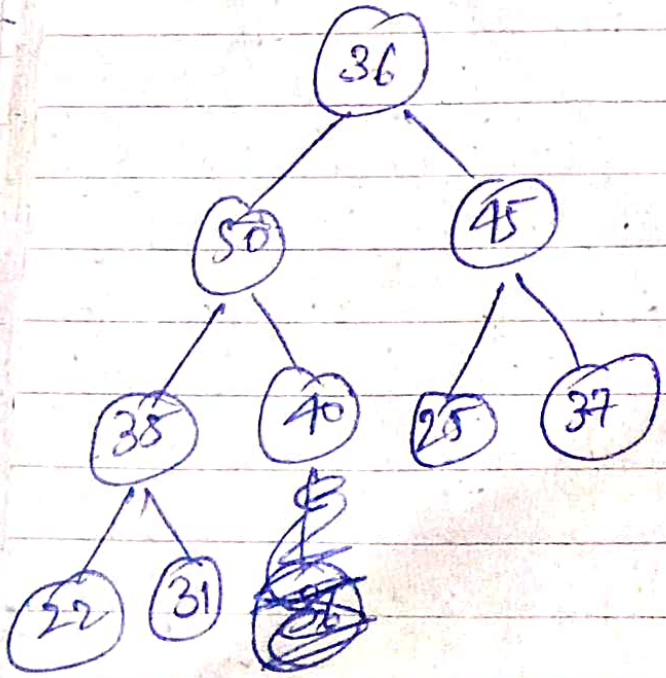


Deletion:-

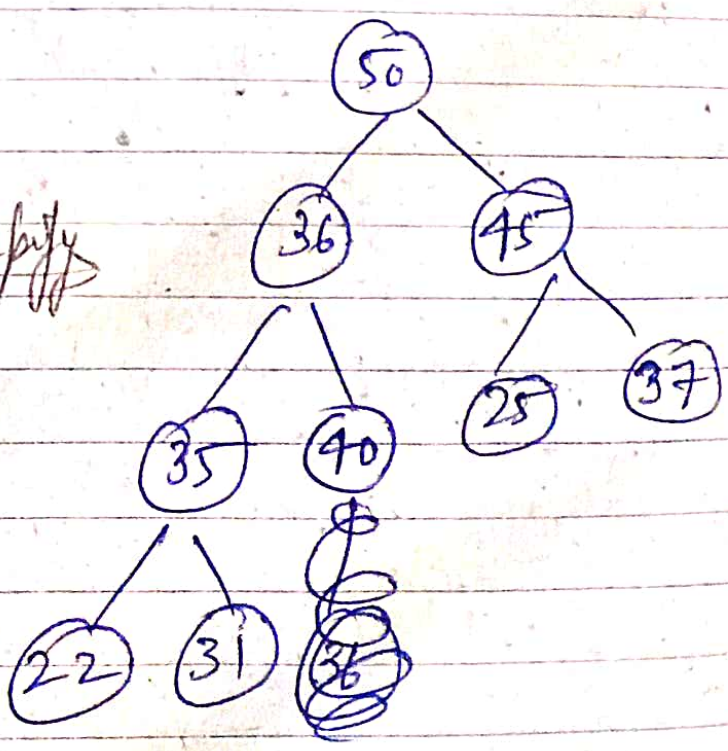
Deletion will be done from root node & after that last node will become root node & we do Heapify to make it according to rule of heap.



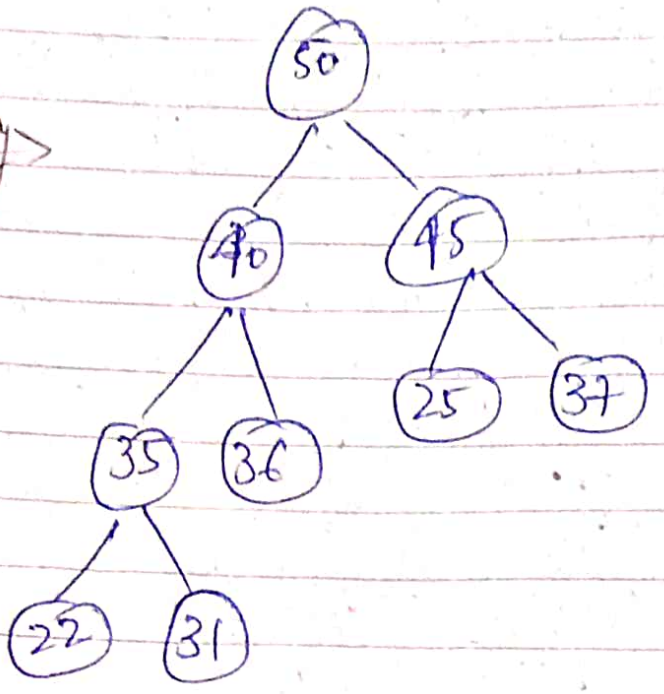
deletion of 70 & make 36 root



heapify



heapify>



Heap Sort :-

→ Heap Sort (A)

1. Build_Max-heap (A)
2. for i = A.length down to 2
   exchange A[1] with A[i]
   A.heapsize = A.heapsize - 1
3. max-heapify (A, 1)

→ Build_Max-heap (A, i)

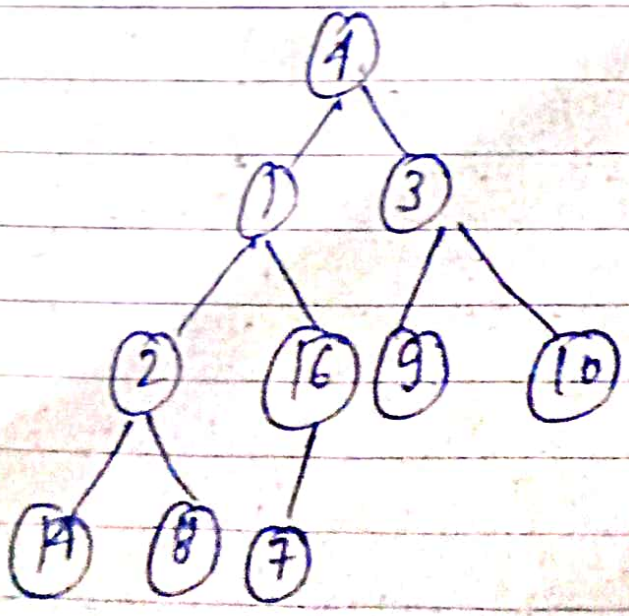
4.
1. A.heapsize = A.length
2. for $i = \left\lfloor \frac{A.length}{2} \right\rfloor$ down to 1

   Max-heapify (A, i)

→ Max-heapify (A, i)

1. $l = left(i)$
2. $r = right(i)$
3. if $l \leq$ A.length and $A[l] > A[i]$
   largest = l
4. else
   largest = i
5. if $r \leq$ A.length and $A[r] > A[largest]$
   largest = r
6. if largest $\neq i$
   exchange $A[i]$ with $A[largest]$
7. Max-heapify (A, largest)

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



do dry run of Heap sort by yourself.

Binomial Heap

A Binomial Heap is a collection of Binomial tree each of which satisfies the properties (min-heap).
To understand Binomial heap, we must understand Binomial tree.

Binomial tree :-

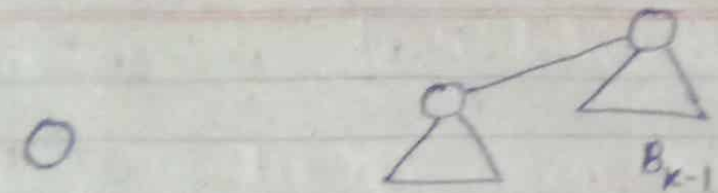
Binomial tree B_k is an ordered tree defined Recursively

* The Binomial tree B_0 consist single node.
* The Binomial Tree B_k consist ~~two Binary~~ Binomial tree B_{k-1} and B_{k-1} are linked together

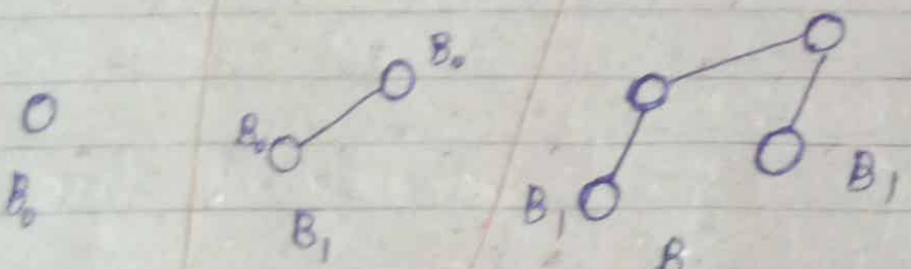
Properties of Binomial tree (B_k) :-

1. There are 2^k nodes.
2. The height of tree is k.
3. there are exactly ^kC_i or $\binom{k}{i}$ node at depth $i = 0, 1, \dots k$
4. the root has degree k which is greater than another node. the children of the root are numbered from left to Right $[k-1, k-2, \dots, 1, 0]$

$B_{k-1}, B_{k-2}, \dots B_1, B_0$

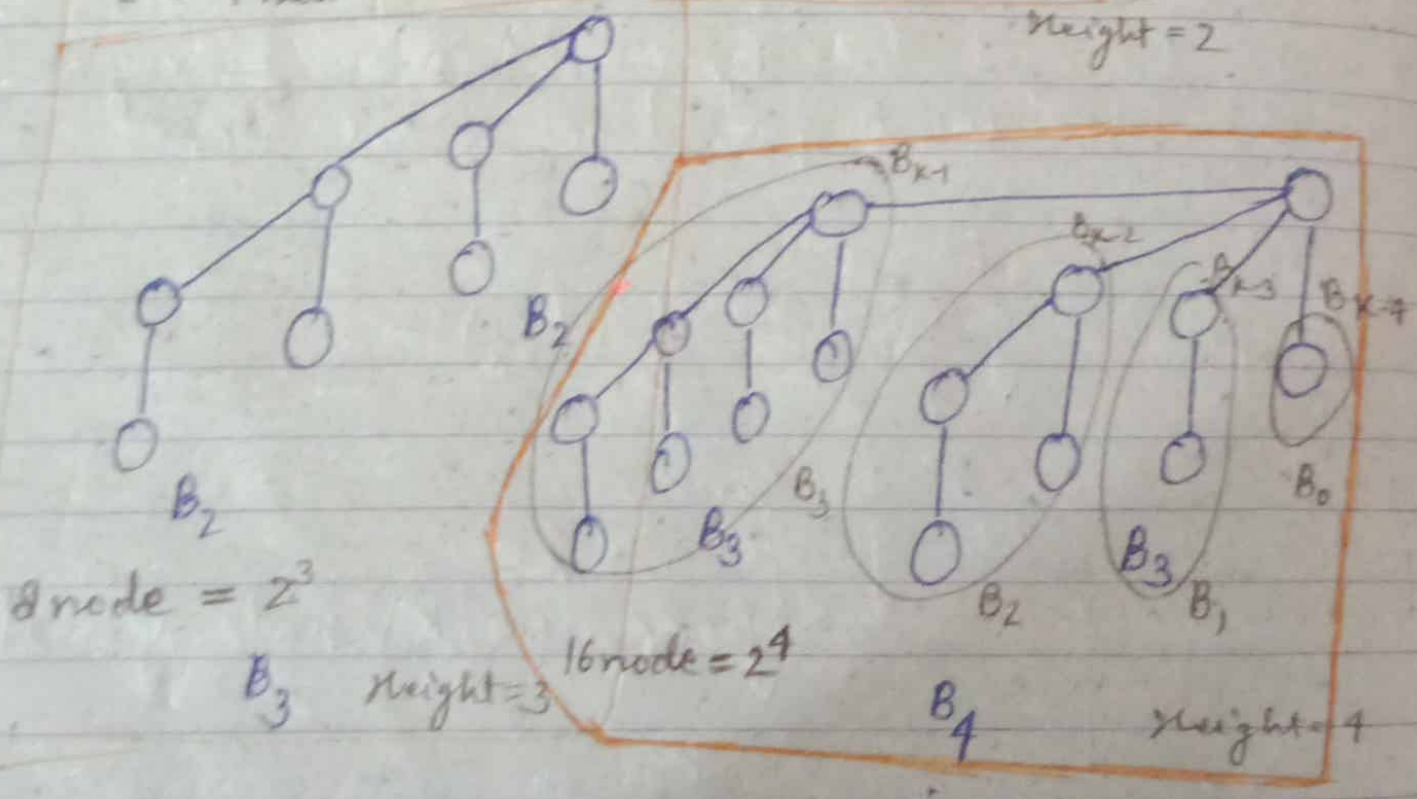


B_0 B_k



$2^0 = 1$ node 2 node $= 2^1$ 4 node $= 2^2$

Height $= 2$



8 node $= 2^3$ 16 node $= 2^4$

B_3 Height $= 3$ B_4 Height $= 4$

$k = 1$ $i = 2$

${}^4C_2 = 6$ nodes

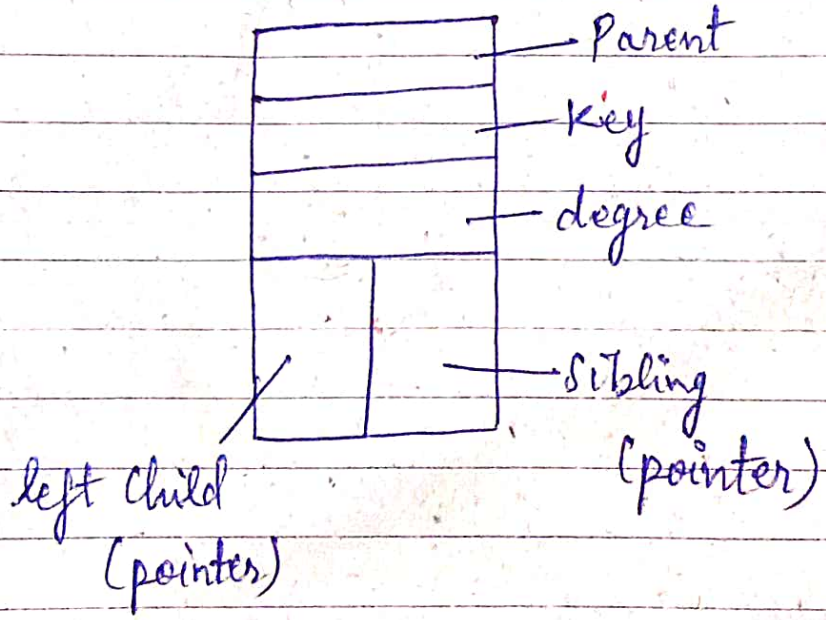
✓

Binomial Heap properties

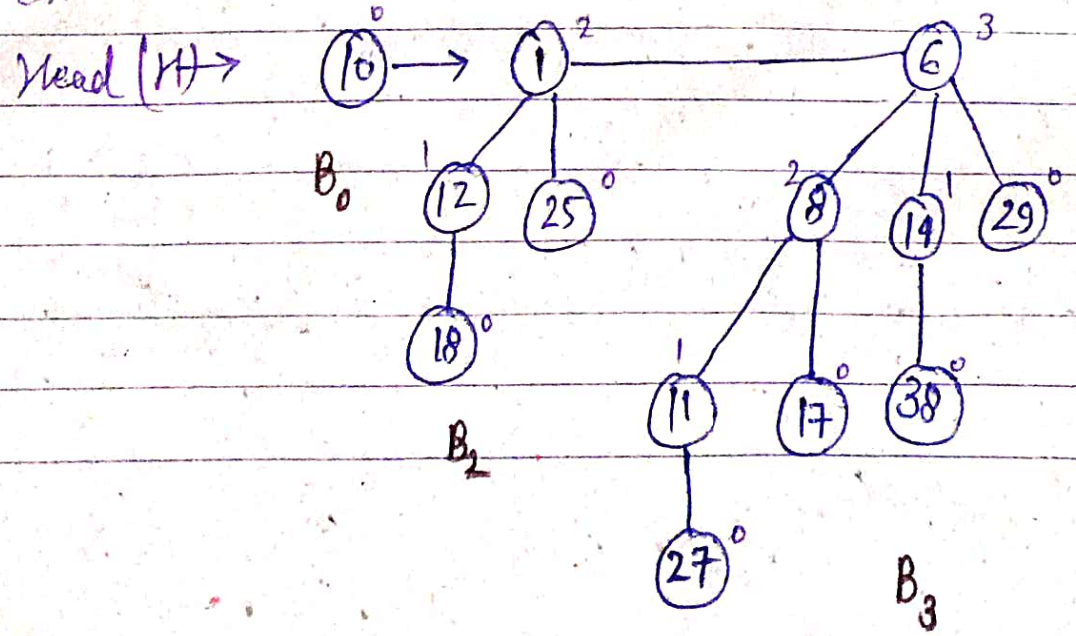
(1) Each Binomial heap (H) obeys the min-heap properties

(2) for any non-Negative integer k there is at most one Binomial tree in H whose root has degree k.

Representation of Binomial-heap Node :-



Ex :-



Node = 13

Binary Number (13) = $\overset{3\ 2\ 1\ 0}{1\ 1\ 0\ 1}$

$B_3\ B_2\ B_0$

(Binomial Heap having 13 nodes consist of B_3, B_2 & B_0)

Node = 16

Binary Number (16) = $\overset{4\ 3\ 2\ 1\ 0}{1\ 0\ 0\ 0\ 0}$

B_4

(Binomial heap having 16 nodes consist of B_4 only)

Node = 9

Binary Number (9) = $\overset{3\ 2\ 1\ 0}{1\ 0\ 0\ 1}$

$B_3\ B_0$

(Binomial heap having 9 nodes consist of B_3 & B_0).

Operations:-

1. Creating a new Binomial Heap → O(1)
2. Finding the minimum key → O(log n)
3. Union of two Binomial heap → O(log n)
4. Inserting a node → O(log n)

5. Extracting minimum key → O(log n)

6. Decreasing a key → O(log n)

7. Deleting a node → O(log n)

5. Extracting minimum key → O(log n)

6. Decreasing a key → O(log n)

7. Deleting a node → O(log n)

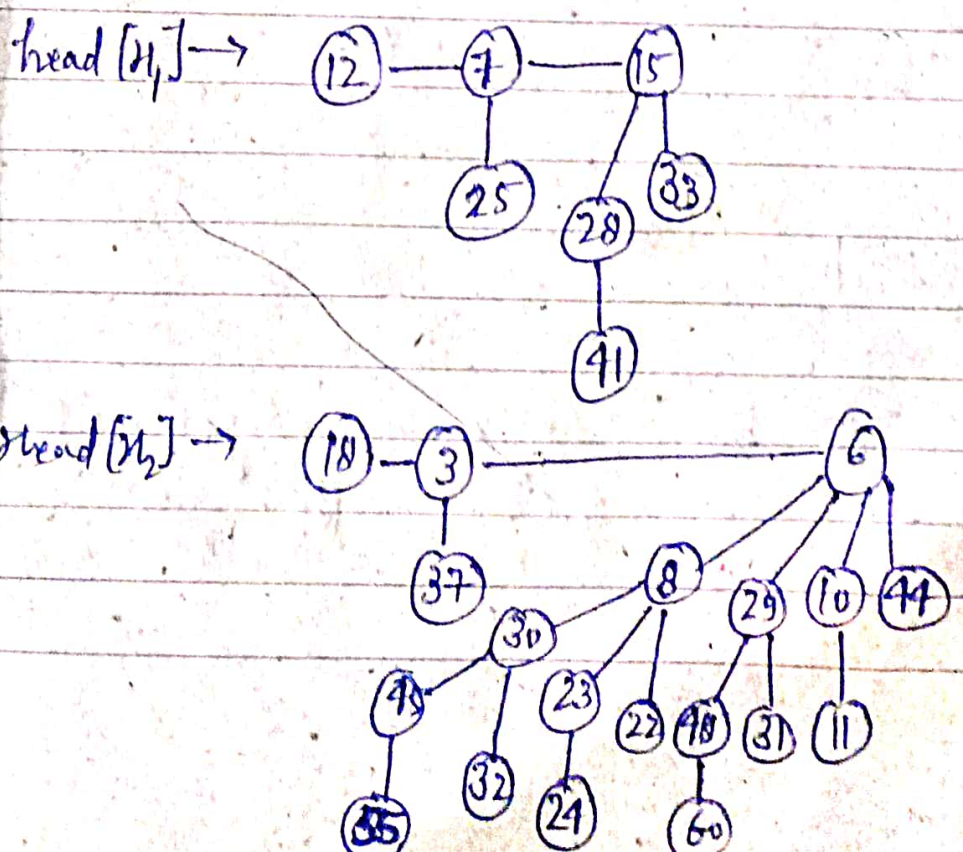
Union :-

Case (1) if degree [x] ≠ degree [next x] then move pointer ahead.

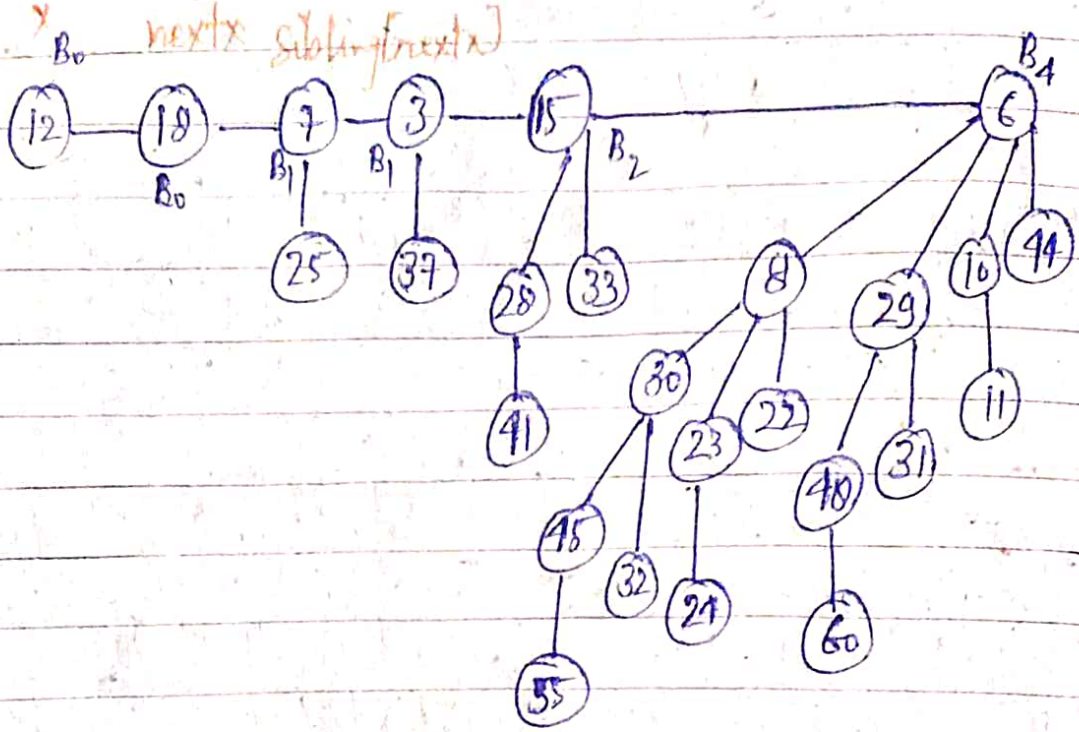
Case (2) if degree [x] = degree [next x] = degree [sibling [next x]] move pointer ahead.

Case (3) if degree [x] = degree [next x] ≠ degree [sibling [next x]] and key [x] ≤ key [Next x] then remove [next x] from root & attached to x

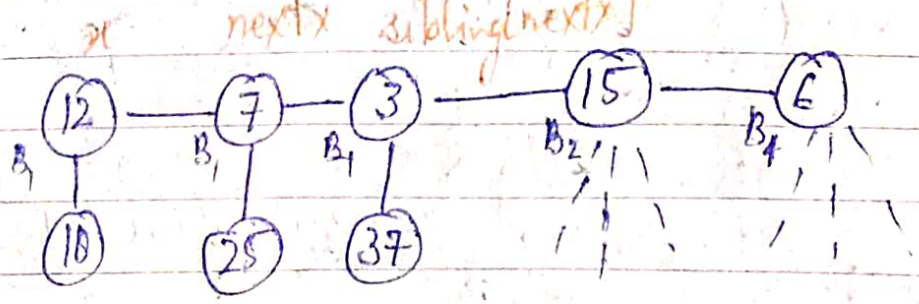
Case (4) if degree [x] = degree [next x] ≠ degree [sibling [Next x]] and key [x] > key [Next x] then remove x from root and attached to [Next x]

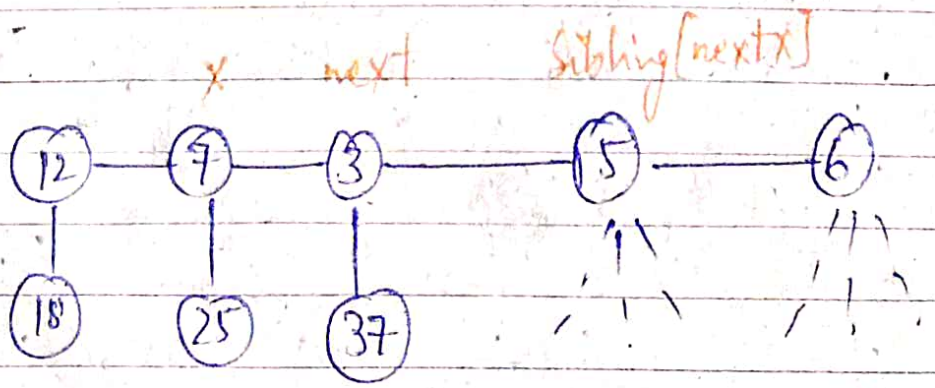


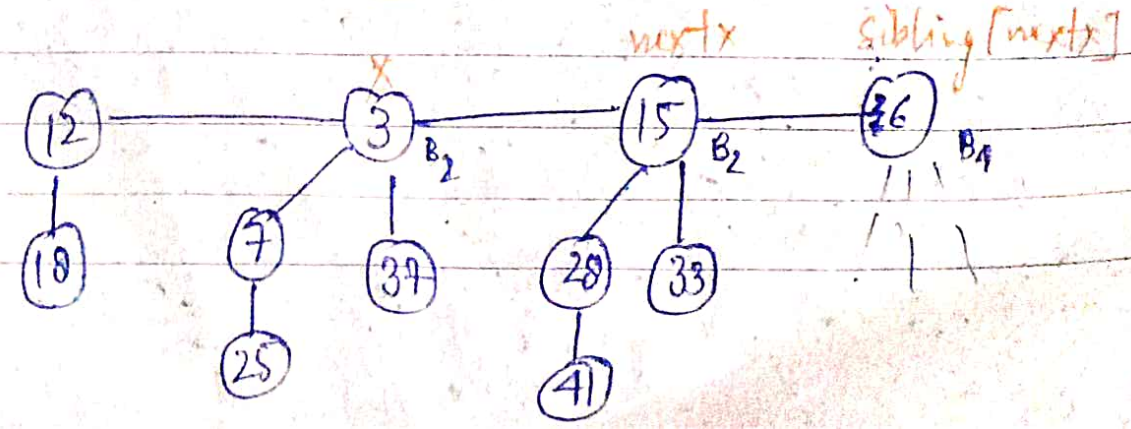
head[H] →

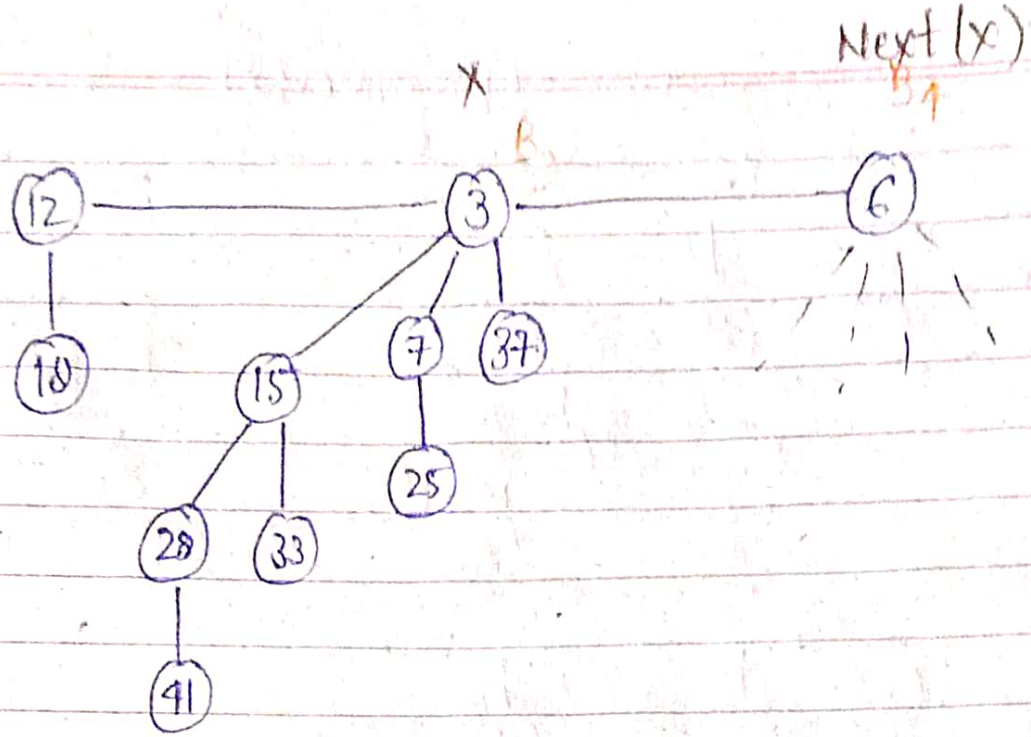


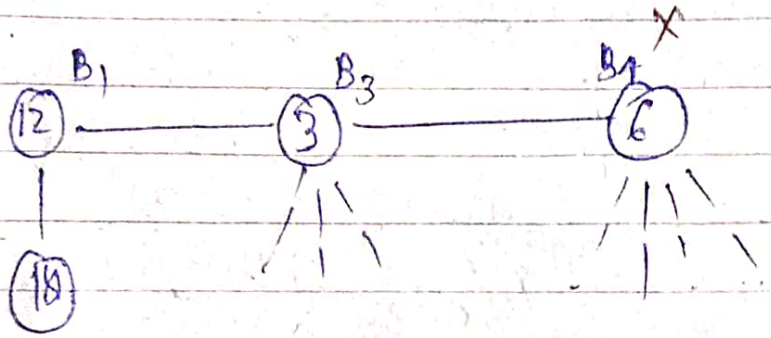
operations :-





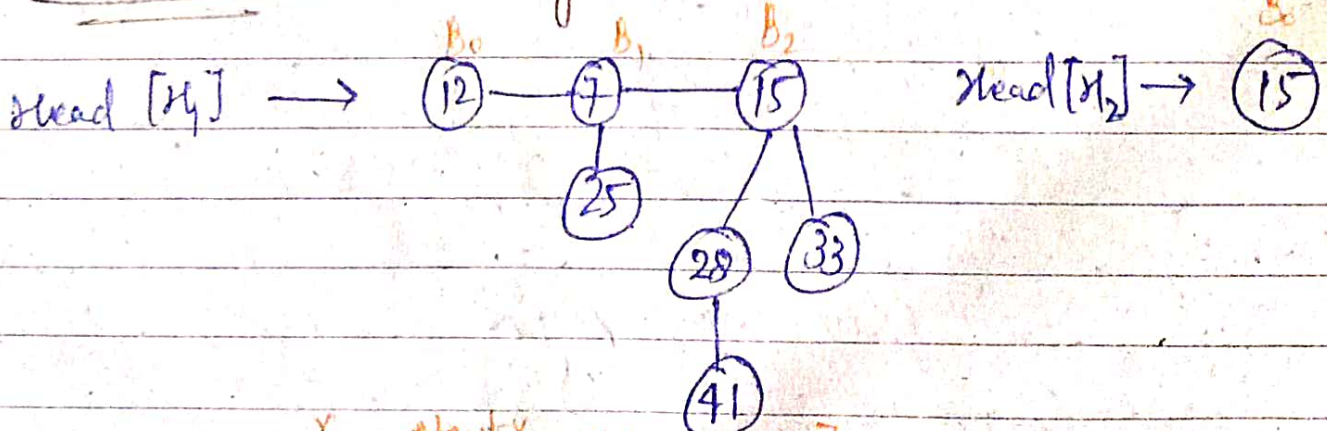


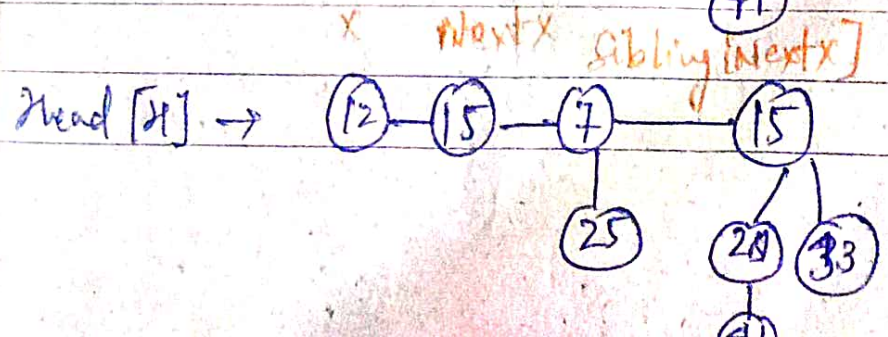


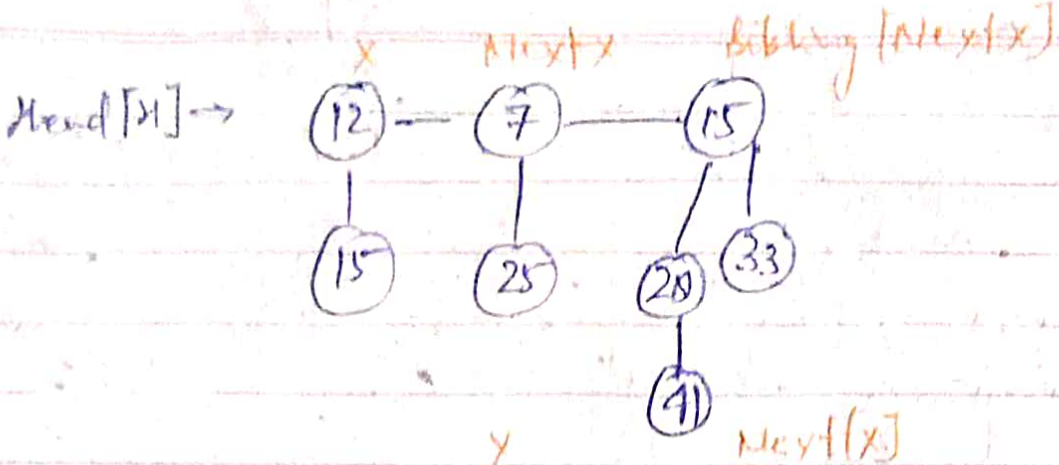


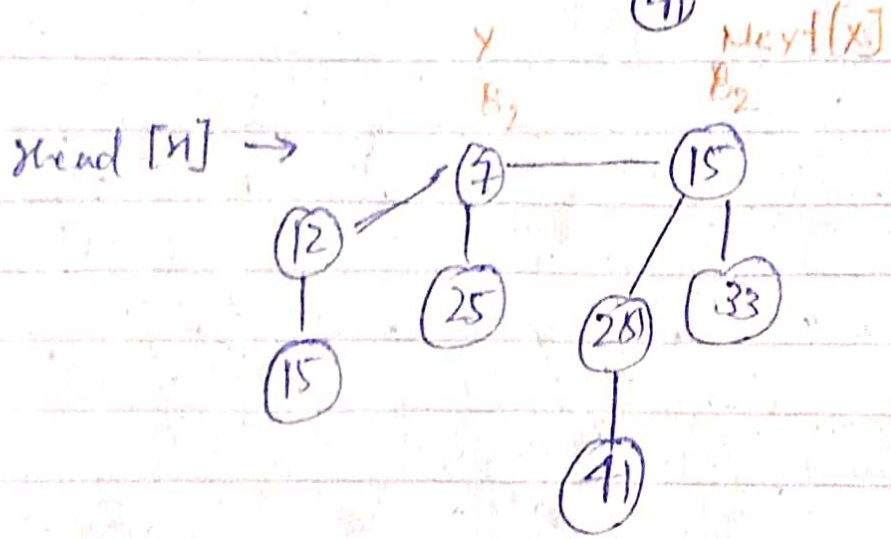
final Binomial Heap

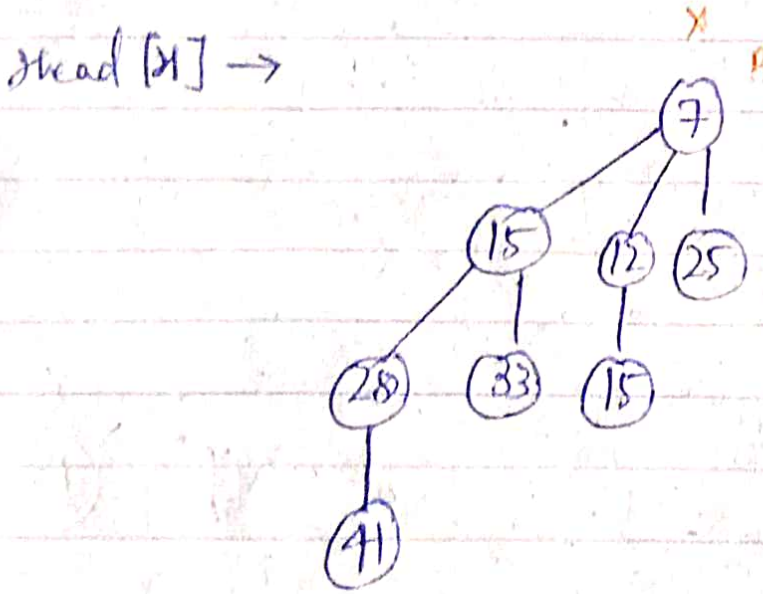
Insertion :- O(log n)



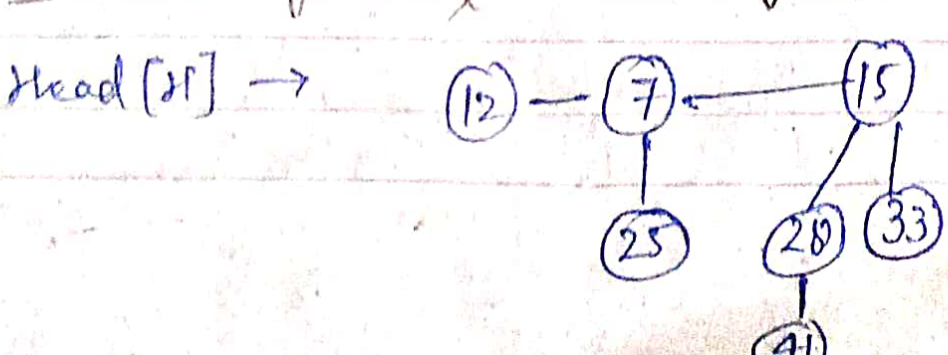




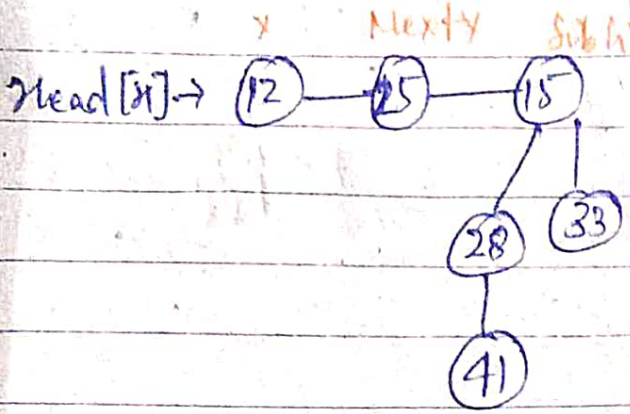


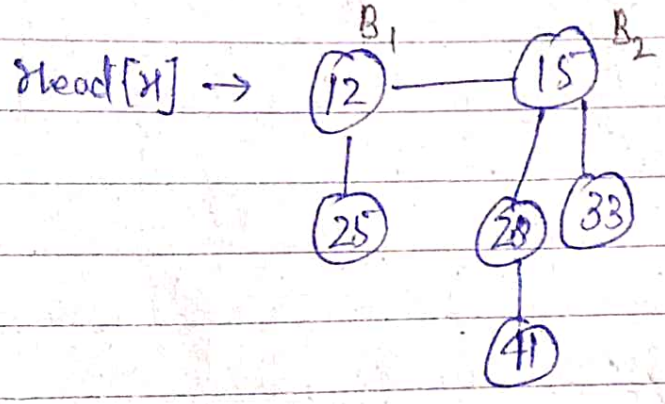


Extracting Minimum key:-

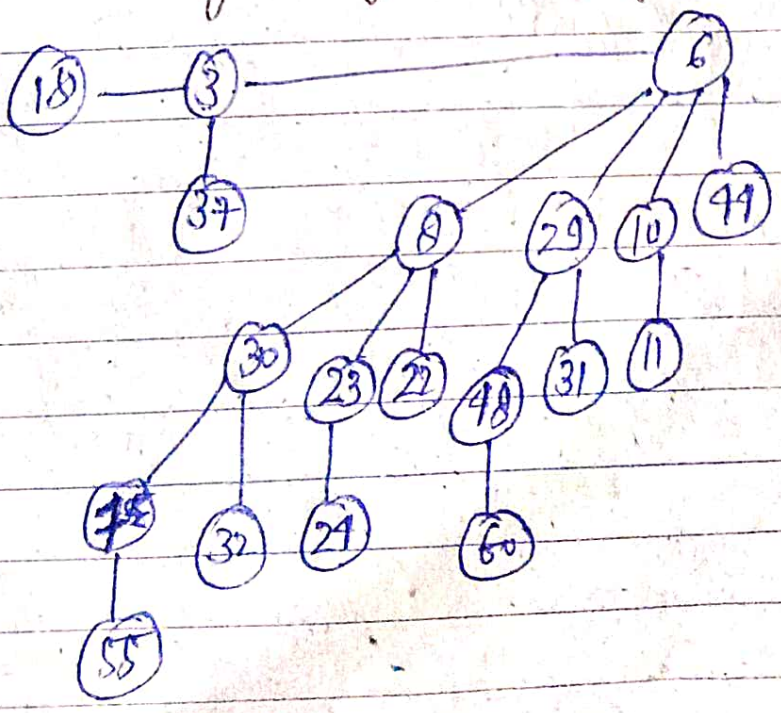


Remove minimum key 7, we get :-

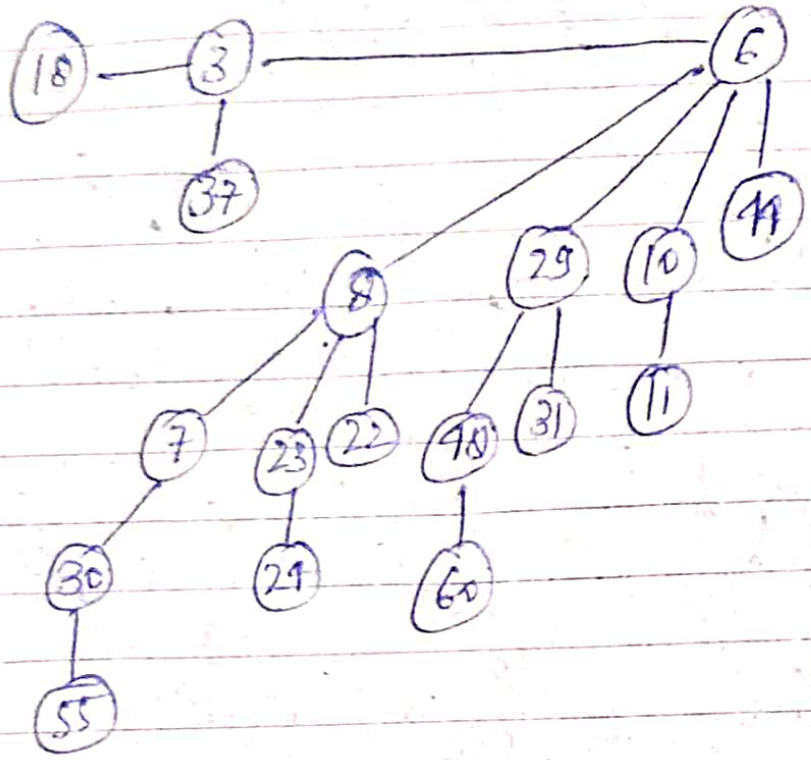




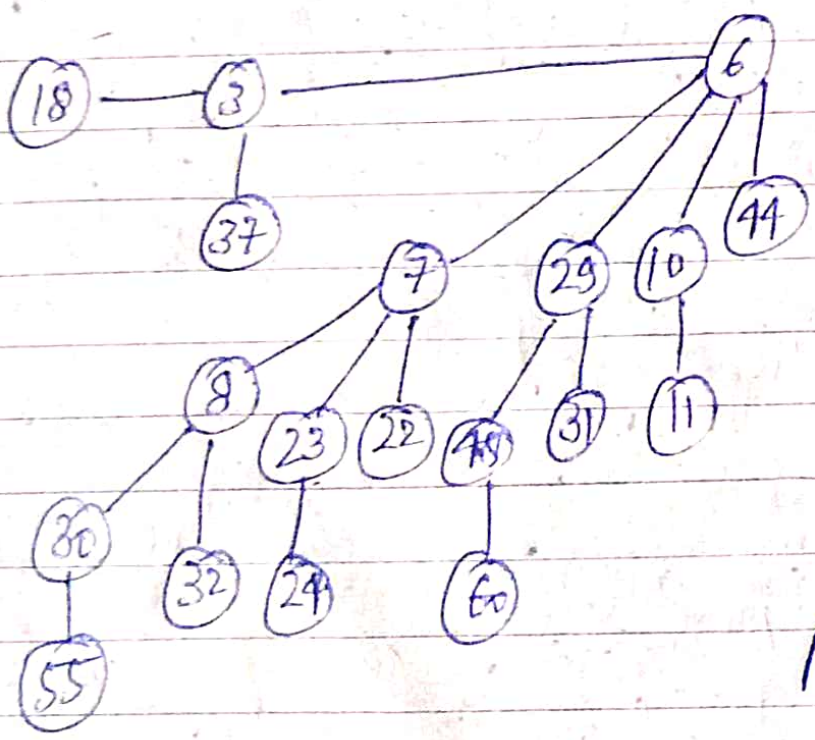
Decreasing key → O(log n)



heapify →



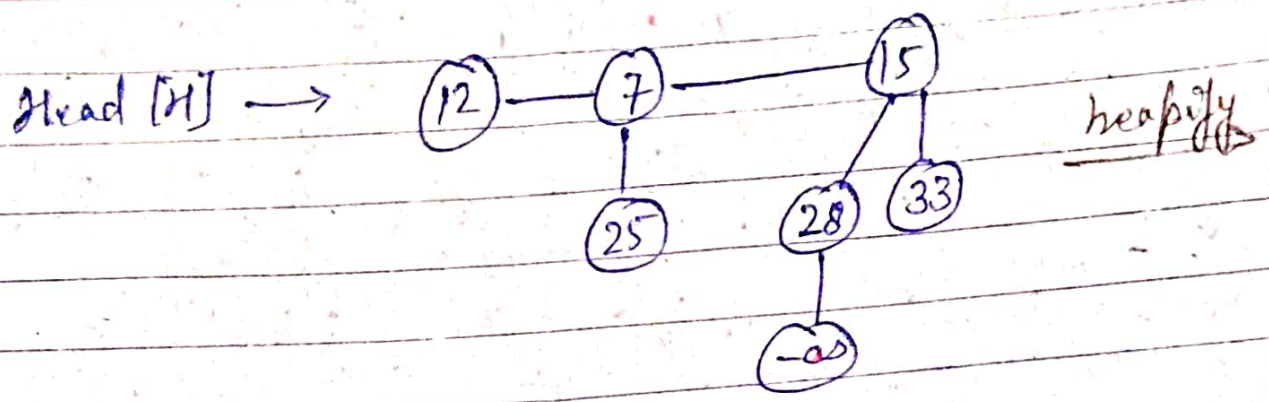
heapify

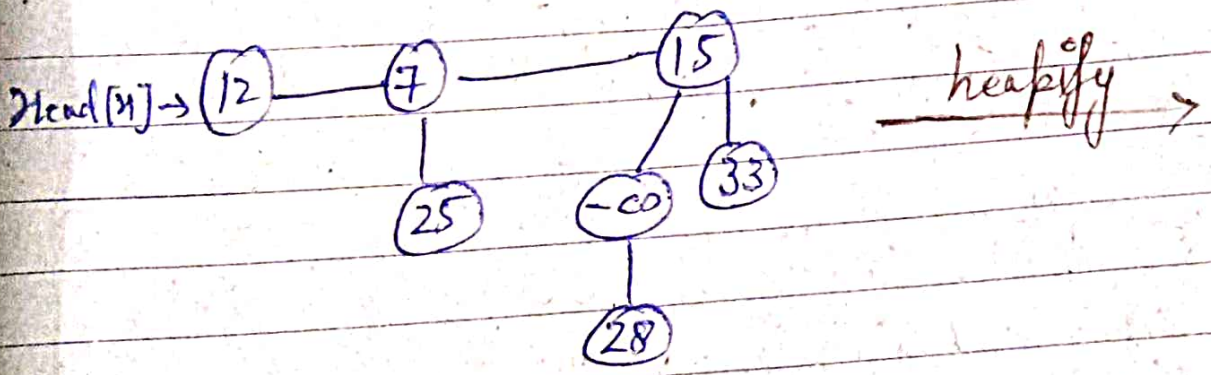


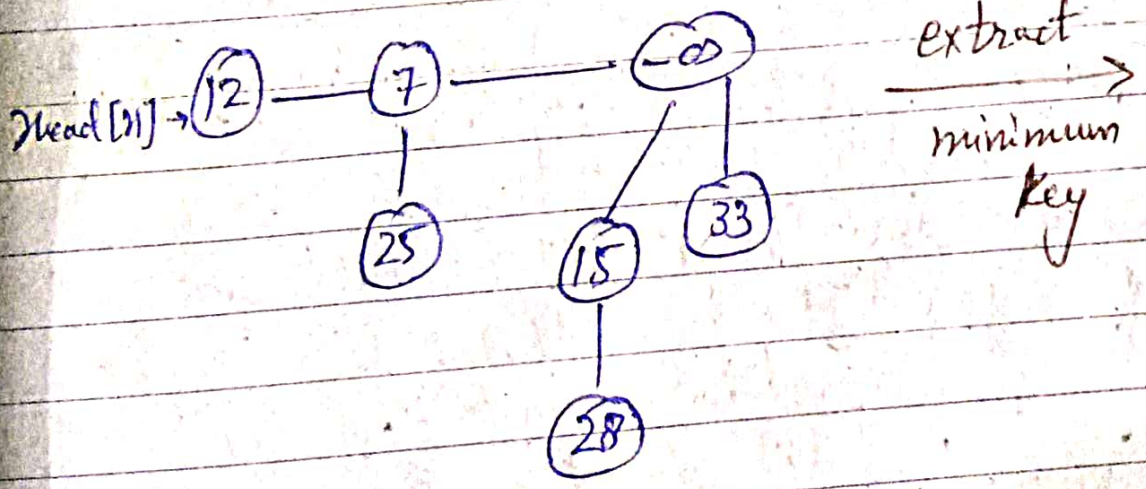
Ans

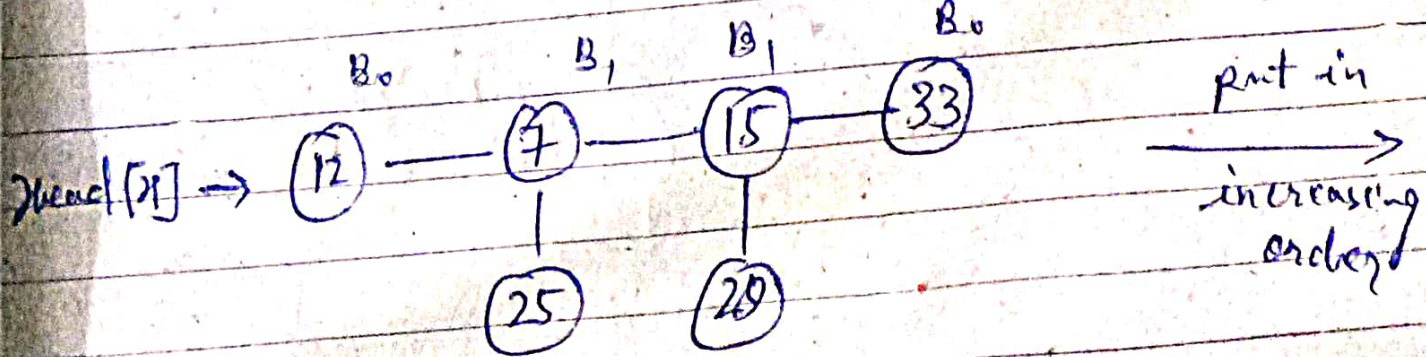
Deleting a Key :-

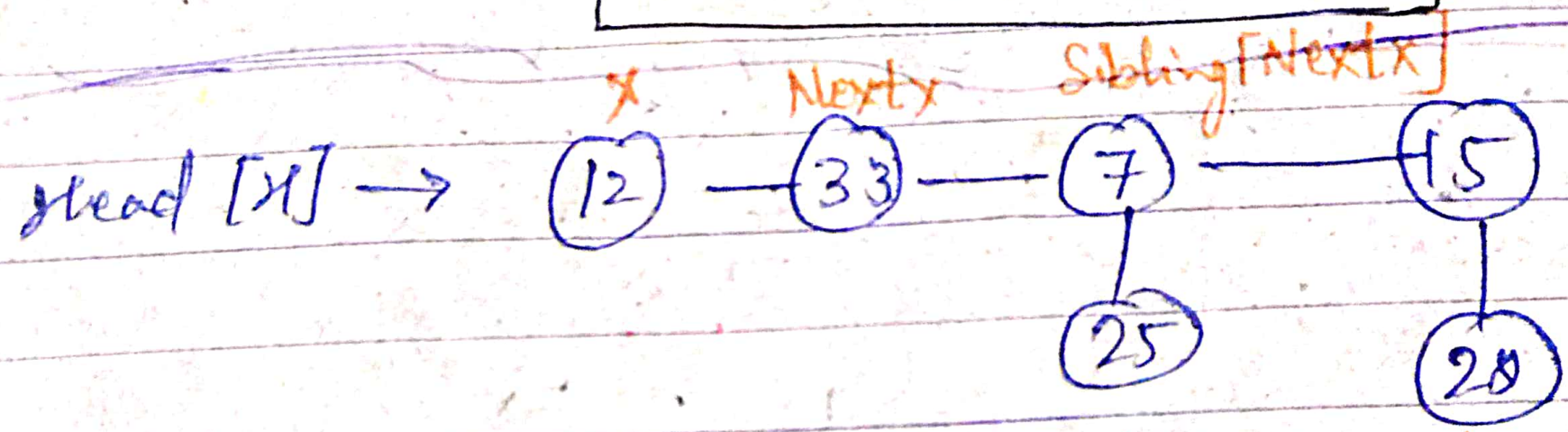
Replace the element which we have to delete with $-\infty$

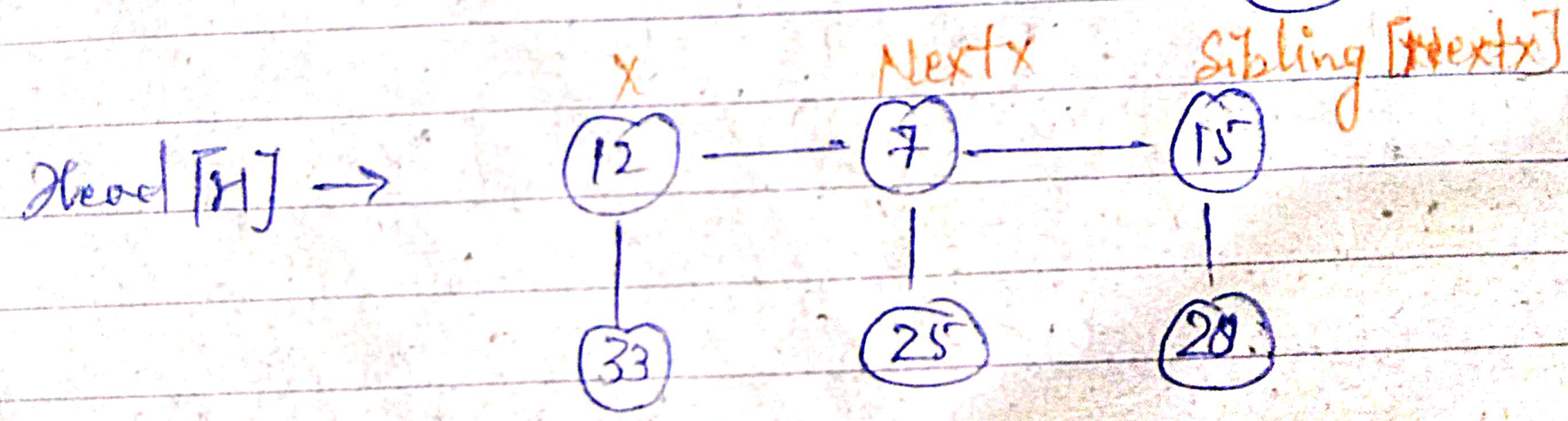

Head [H] →
12
7
15
25
28
33
-∞
heapify

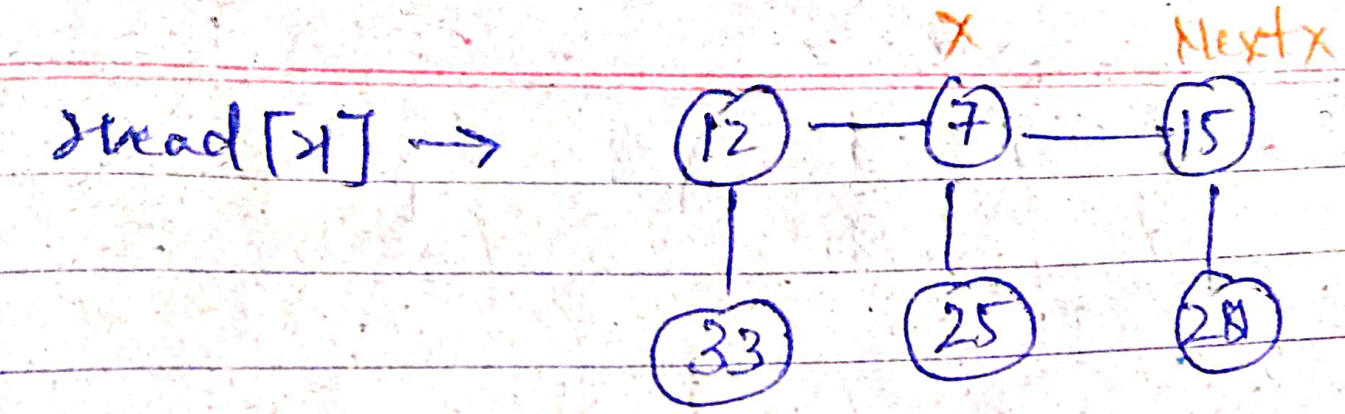

Head[H] →
12
7
15
25
-∞
33
28
heapify

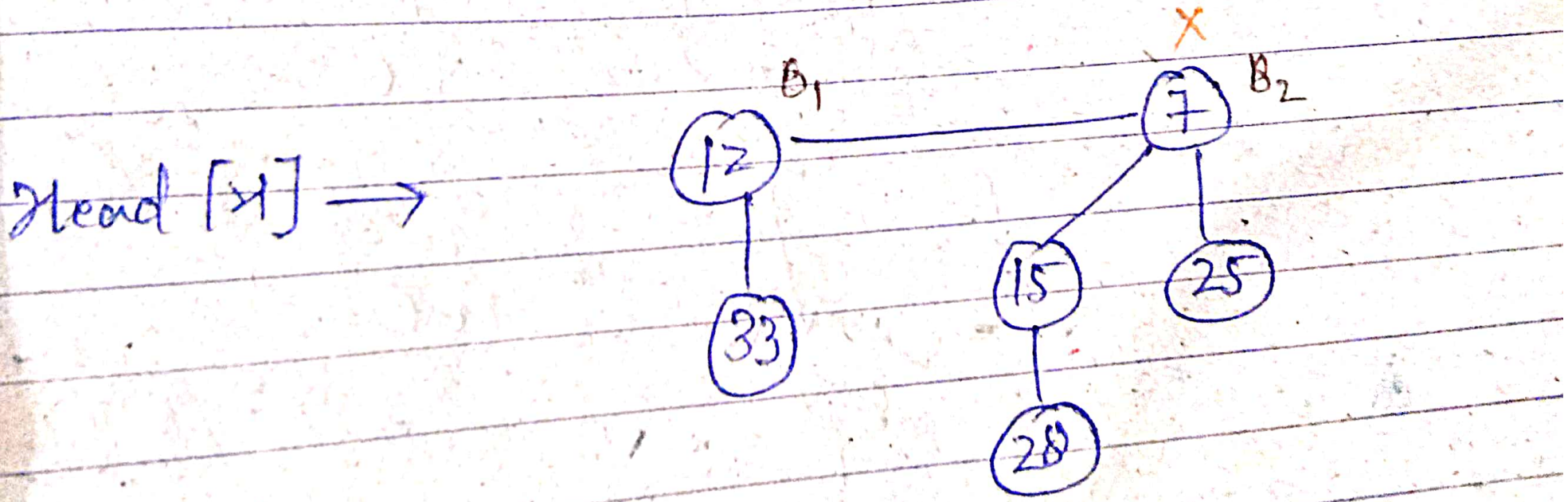

Head[H] →
12
7
-∞
25
15
33
28
extract
minimum
key


B0
B1
B2
B0
Head[H] →
12
7
15
33
25
20
put in
increasing
order


X
Nextx
Sibling[Nextx]
Head[H] →
12
33
7
15
25
20


X
Nextx
Sibling[Nextx]
Head[H] →
12
7
15
33
25
20


x
Next x
Head [x] →
12
7
15
33
25
20


x
B1
B2
Head [x] →
12
7
33
15
25
20