29/08/23 ADSA

Definition of B-Tree, Hight of B-Tree

B-Tree T is a rooted tree having properties like

1) Every node x has the following attributes
   - → x.n, the no. of keys currently stored in node x,
   - → the x.n keys themselves, $x.key_1, x.key_2, \ldots x.key_n$ stored in non decreasing order so that $x.key_1 \le x.key_2 \le \ldots \le x.key_{x.n}$
   - → x.leaf 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, x.n+1 pointers, $x.c_1, x.c_2, \ldots x.c_{x.n+1}$ to its children. Leaf node have no children.
3) The $x.key_i$ seperates the ranges of keys stored in each subtree, if k_i is any key stored in the subtree with root $x.c_i$ then
   $$k_1 \le x.key_1 \le k_2 \le x.key_2 \ldots$$
4) All leaves have the same ~~height~~ depth which is the height of tree.
5) Node have lower & upper bounds on the number of keys they can contains. We express these bounds in terms of fixed integer $t \ge 2$ called the minimum degree of B-tree

Height of B-Tree

If $n \ge 1$, then for any n-key B-Tree T of height h & minimum degree $t \ge 2$ then

$$h \le \log_t \frac{n+1}{2}$$

Proof:

$n \le$

Create a B-Tree for the list

15, 31, 27, 19, 22, 16, 11, 32, 17, 9, 10, 6

with minimum degree $t = 2$ by successive insertion method.

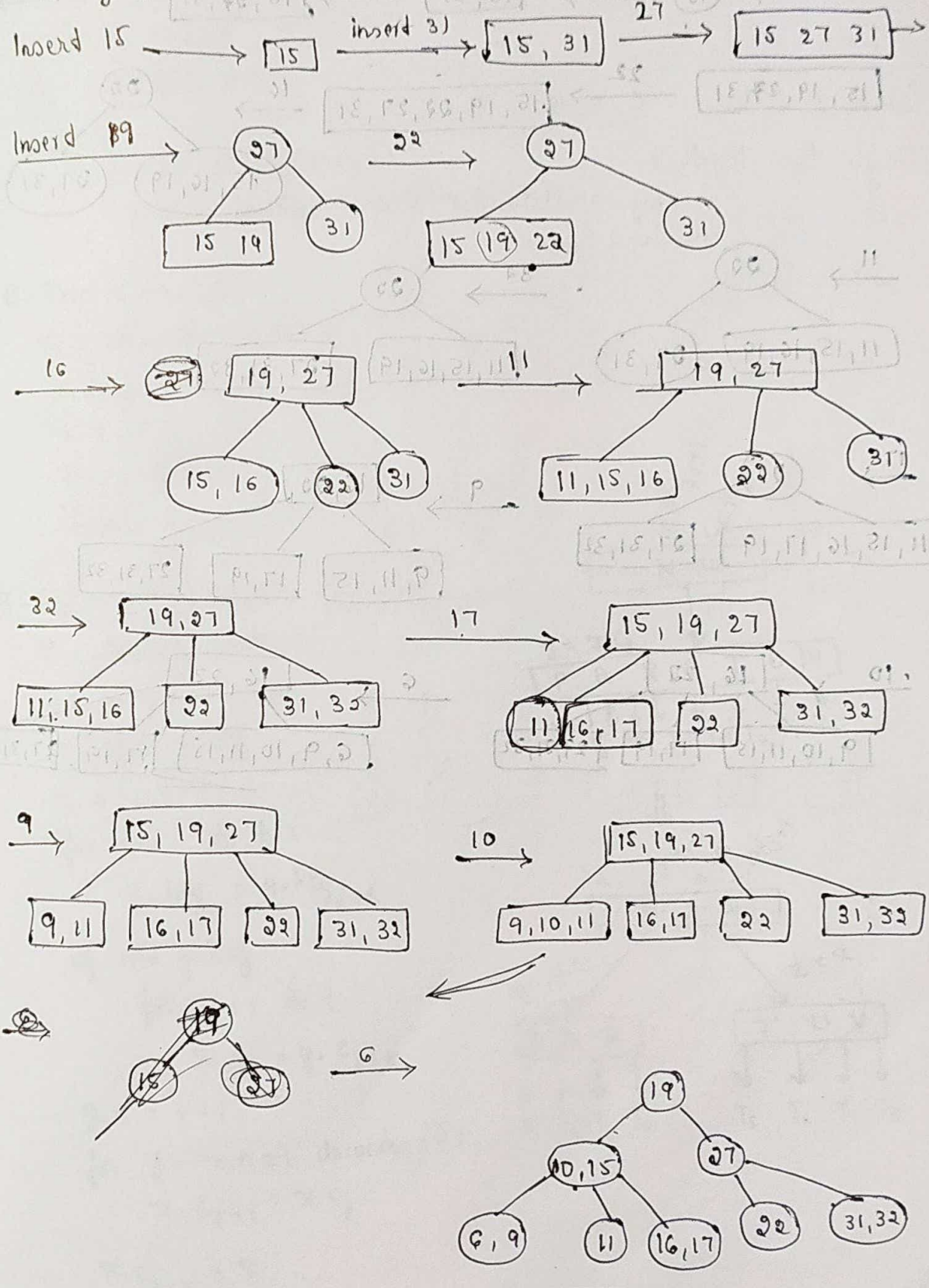
Soln:

minkey = $t - 1 = 1$

maxkey = $2t - 1 = 3$.

minlink = $t = 2$

maxlink = $2t = 4$

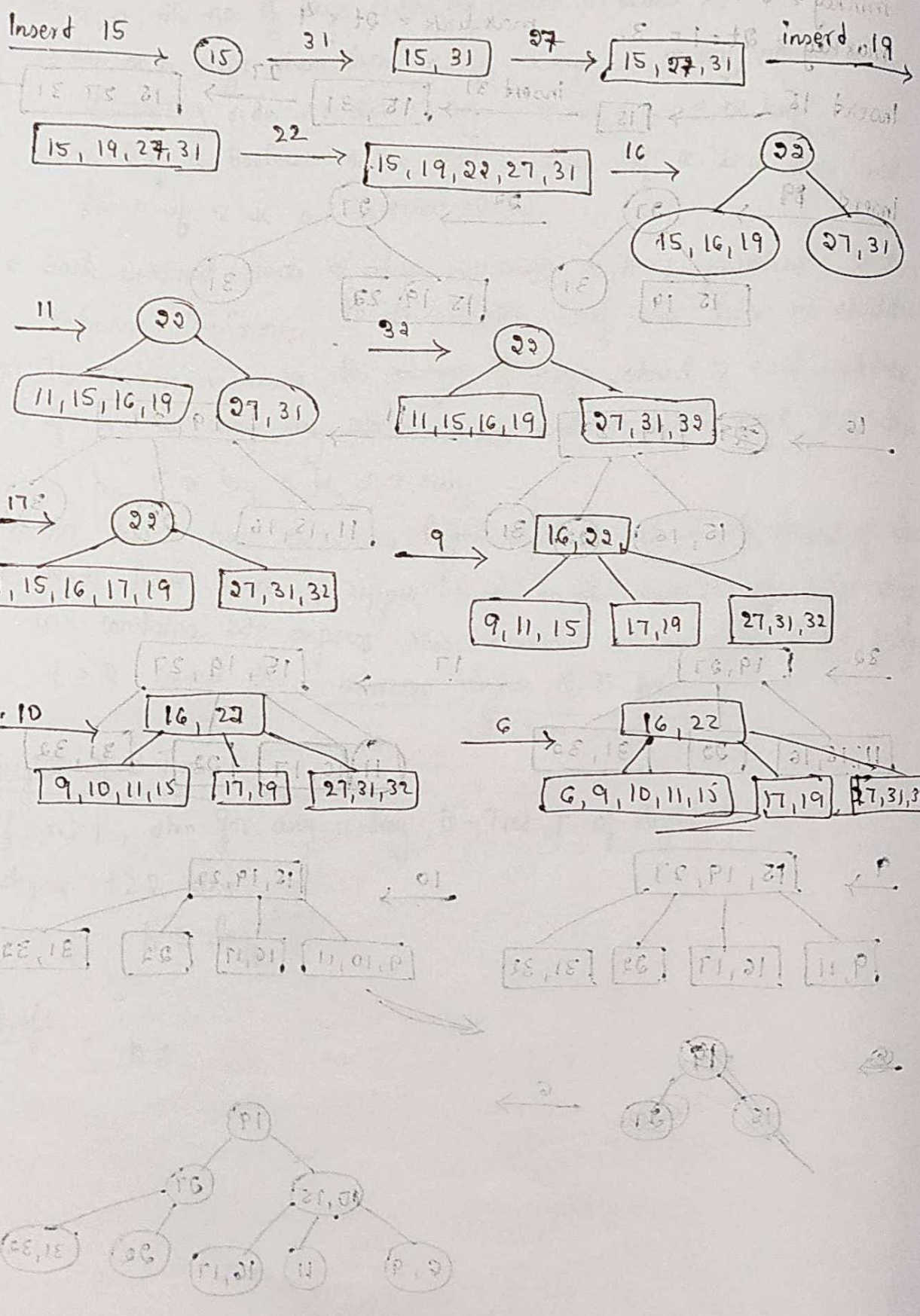


For same list $t = 3$,

Solut

minkey = $t - 1 = 2$ — minlink = $t = 3$

maxkey = $2t - 1 = 5$ — maxlink = $2t = 6$

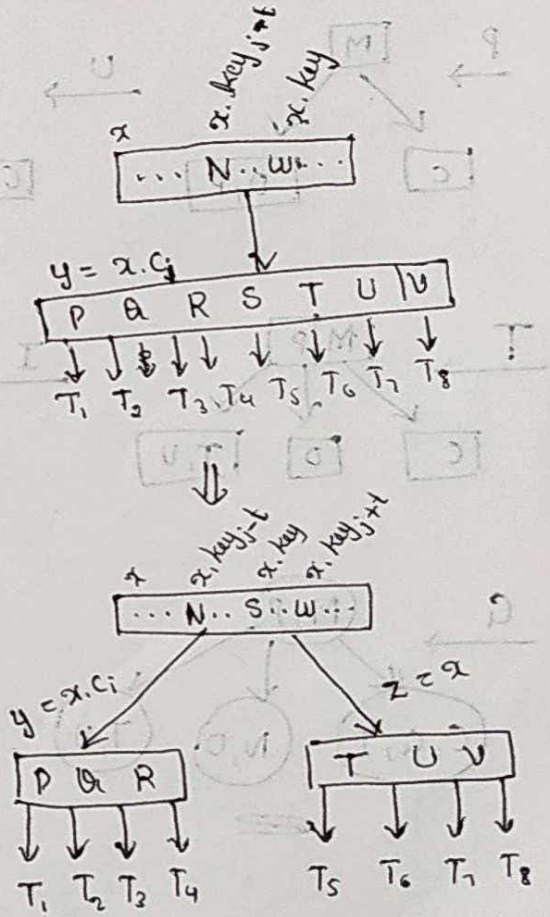


Searching & Creating B-Tree :-

```
B-Tree-Search (x, k)
    i = 1
    while i ≤ x.n and k > x.key_i
        i = i + 1
    if i ≤ x.n and k == x.key_i
        return (x, i)
    else if x.leaf
        return NIL
    else
        DISK-READ(x.c_i)
        return B-Tree-Search(x.c_i, k)
```

```
B-Tree-Create(T)
    x = Allocate-Node()
    x.leaf = True
    x.n = 0
    Disk-Write(x)
    T.root = x
```

```
B-Tree-Split-Child(x, i)
    z = Allocate-Node()
    y = x.c_i
    z.leaf = y.leaf
    z.n = t - 1
    for j = 1 do t-1
        z.key_j = y.key_{j+t}
    if not y.leaf
        for j = 1 do t
            z.c_j = y.c_{j+t}
    y.n = t - 1
    for j = x.n + 1 downto i + 1
        x.c_{j+1} = x.c_j
    x.c_{i+1} = z
```



for $j = x.n$ downto i

$x.key_{j+1} = x.key_j$

$x.key_i = y.key_i$

$x.n = x.n + 1$

Disk-Write(y)

Disk-Write(z)

Disk-Write(x)

Construct B-Tree, C, O, M, P, U, T, I, N, G and $t = 2$ using Single Pass Method.

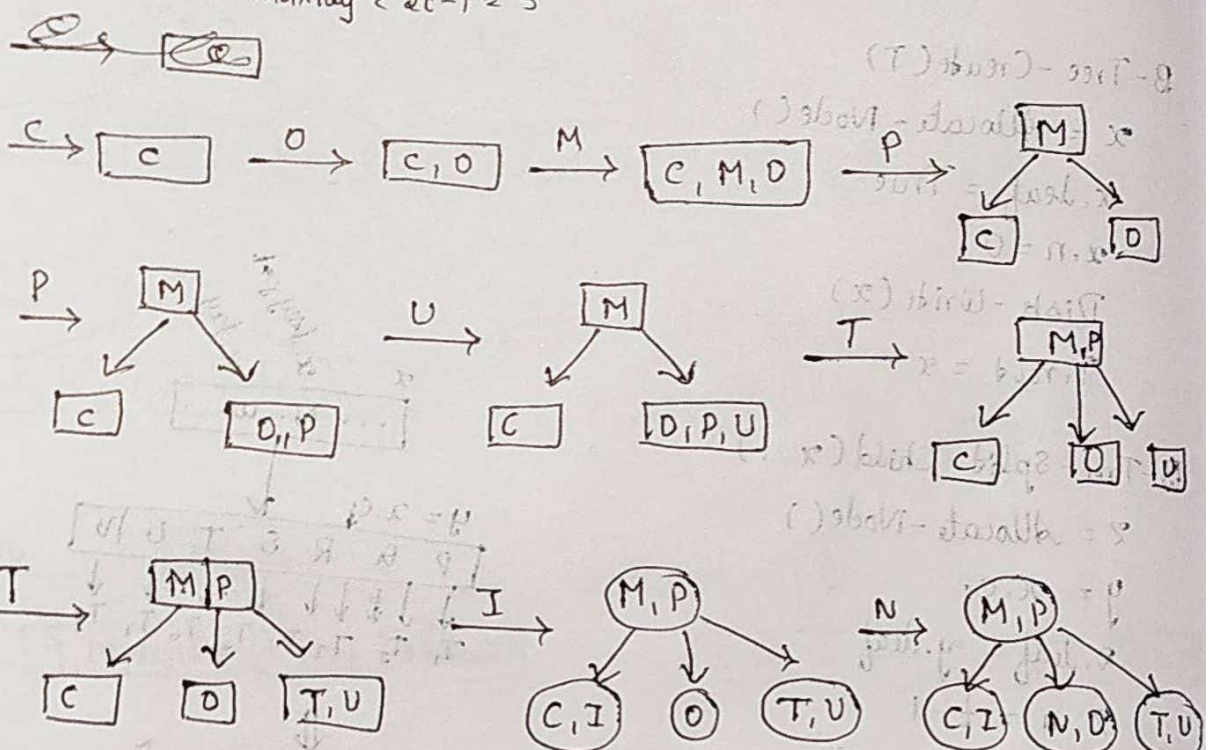
Soln:

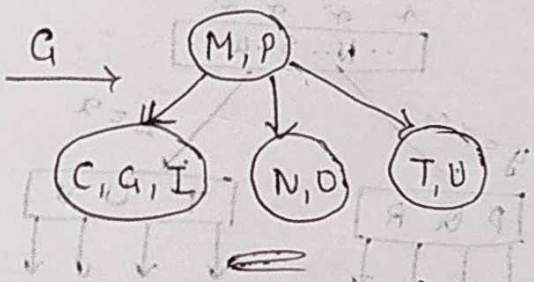
minkey = $t - 1 = 1$

maxkey = $2t - 1 = 3$

minlink = $t = 2$

maxlink = $2t = 4$





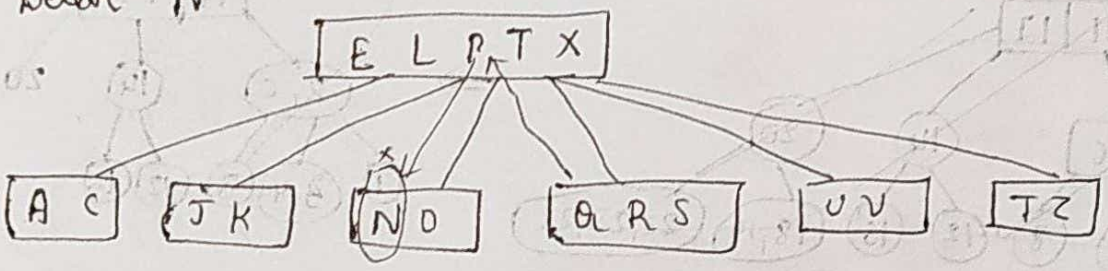
```
B-Tree-Insert (T, k)
    r = T.root
    if r.n == 2t-1
        s = Allocate-Node()
        T.root = s
        s.leaf = False
        s.n = 0
        s.c1 = r
        B-Tree-Split-Child (s, 1)
        B-Tree-Insert-Nonfull (s, k)
    else
        B-Tree-Insert-Nonfull (r, k)
```

```
B-Tree-Insert-Nonfull (x, k)
    i = x.n
    if x.leaf
        while i ≥ 1 and k < x.key_i
            x.key_{i+1} = x.key_i
            i = i-1
        x.key_{i+1} = k
        x.n = x.n + 1
        Disk-Write (x)
    else
        while i ≥ 1 and k < x.key_i
            i = i-1
        i = i+1
        Disk-Read (x.c_i)
        if x.c_i.n == 2t-1
            B-Tree-Split-Child (x, i)
            if k > x.key_i
                i = i+1
        B-Tree-Insert-Nonfull (x.c_i, k)
```

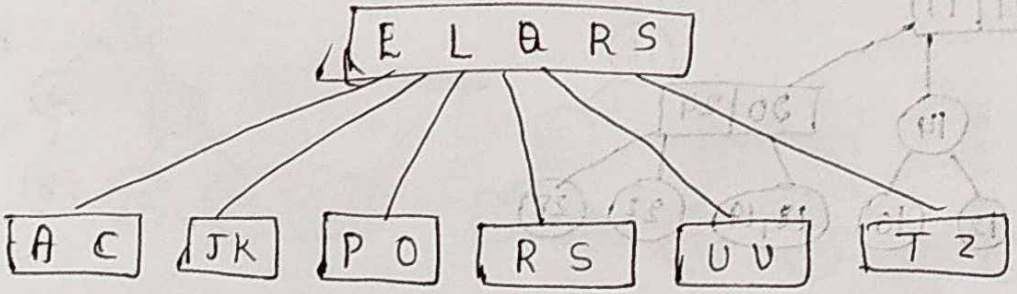
Deleting a key from B-Tree:

1. If the key k is in node x and x is leaf delete the key from x.
2. If the key k is in the node,

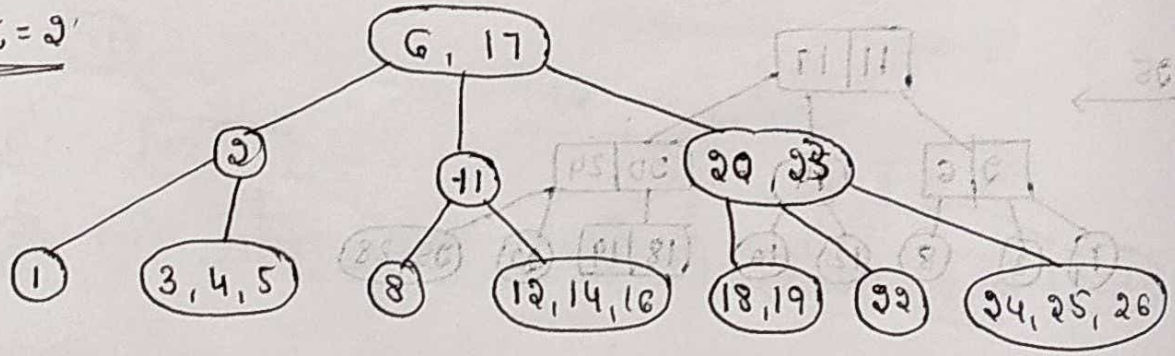
Eg: Delete 'N'



⇒



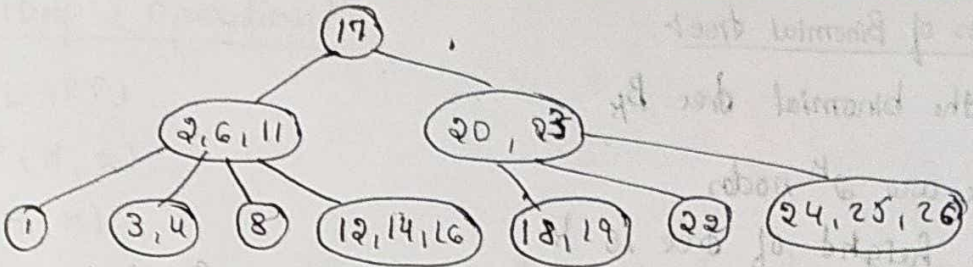
Eg: $t = 2$



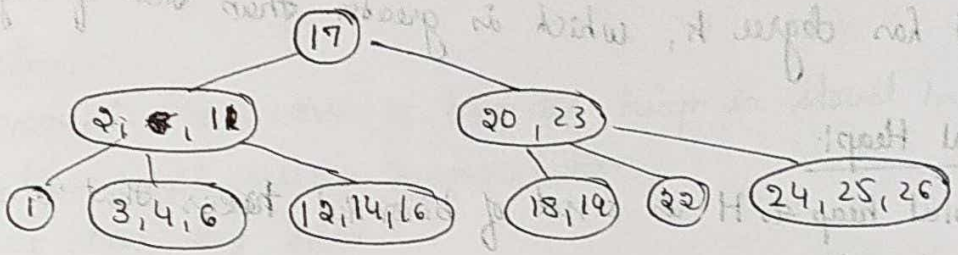
Delete -5

minkey = $t-1 = 1$

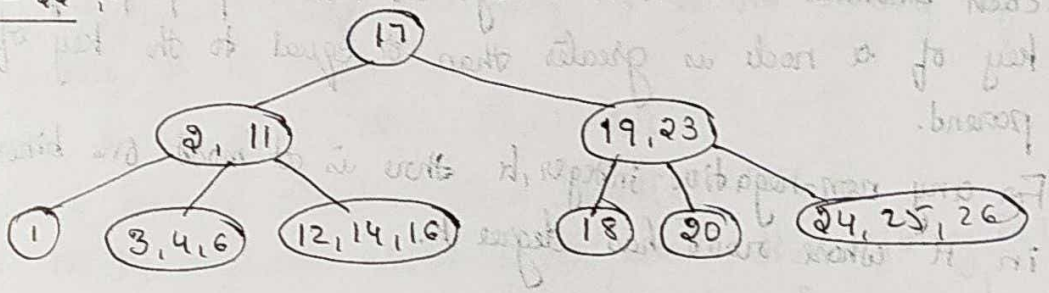
max key = $2t-1 = 3$



Delete - 8 :-



Delete - 22 :-



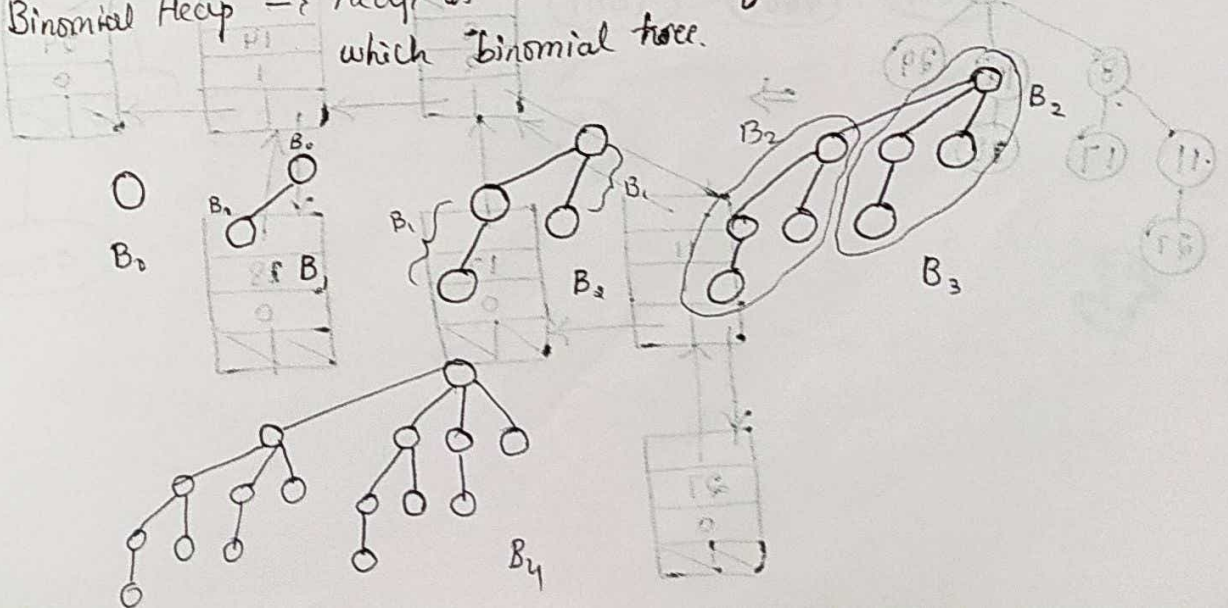
Binomial Heap :-

2 property

1) Tree Shaped → Almost complete binary tree. or complete BT
2) Parental Dominance → Min Heap & Max Heap
   P < C , P > C
   P → Parent
   C → Child

Binary Heap → Heap is a single tree, which is complete Binary tree.

Binomial Heap → Heap is a collection of smaller tree, each of which binomial tree.



Properties of Binomial tree:

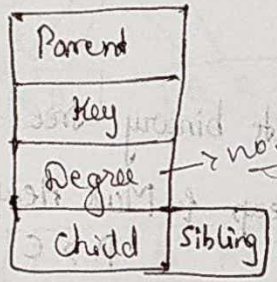
for the binomial tree B_k

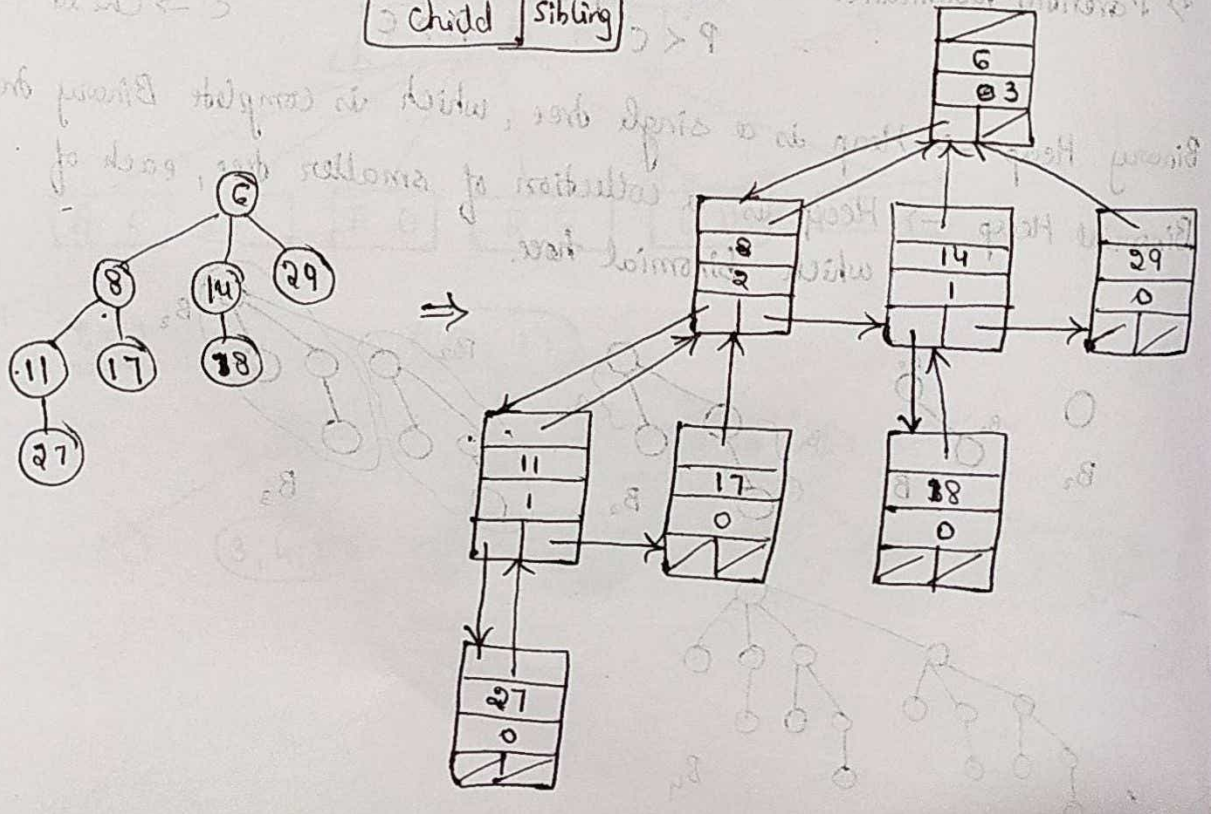
1) there are 2^k nodes
2) the height of tree is k
3) there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0 \ldots k$
4) Root has degree k, which is greater than that of any other node

Binomial Heap:

* Binomial heap H is a set of binomial trees that satisfies the properties
  - → Each binomial tree in H obeys the min-heap property: the key of a node is greater than or equal to the key of its parent.
  - → For any non-negative integer k, there is at most one binomial tree in H whose root has degree k.

Structure:





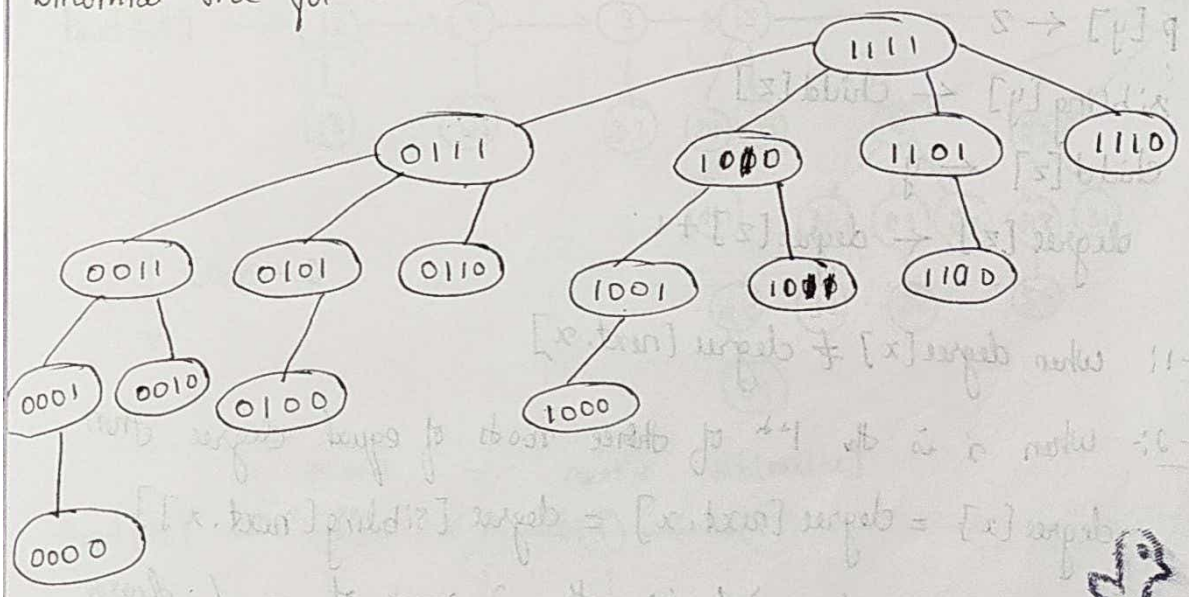
Mergable Heap (Operations):-

1) MAKE_HEAP()
2) INSERT (H, x)
3) MINIMUM (H)
4) EXTRACT_MIN(H) → delete node from H whose key is minimum. & return pointer
5) UNION (H_1, H_2)

Representation:-

* Each binomial tree within a binomial heap is stored in the left-child, right-sibling representation
* Each node has key field
* Each node 'x' contains pointer p[x] to its parent, child[x] to its leftmost child & sibling[x] to the sibling of 'x' immediately to its right.
* If node 'x' is root, then P[x] = NIL
* If node 'x' has no children then child[x] = NIL
* If 'x' is the rightmost child of its parent, then sibling [x] = NIL
* Each node 'x' also contains the field degree[x], which is the no. of children of 'x'.

Binomial tree for $2^4 = 16$



Creating a Binomial Heap:-

To create a empty BH the MAKE-BINOMIAL-HEAP procedure simply allocate & return an object H, where head[H] = NIL it is $\Theta(1)$.

Finding min key:-

```
Binomial-Heap-Minimum (H)
    y ← NIL
    x ← head[H]
    min ← ∞
    while x ≠ NIL
        do if key[x] < min
            then min ← key[x]
                 y ← x
            x ← sibling[x]
    return y
```

Uniting two BH:-

```
Binomial-Link (y, z)
    p[y] ← z
    sibling[y] ← child[z]
    child[z] ← y
    degree[z] ← degree[z] + 1
```

Case-1: when degree[x] ≠ degree[next.x]

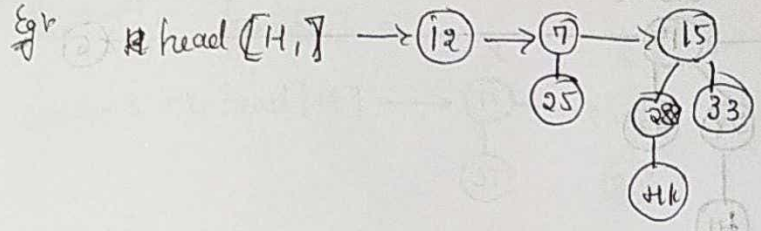
Case-2:- When x is the 1st of three roots of equal degree then

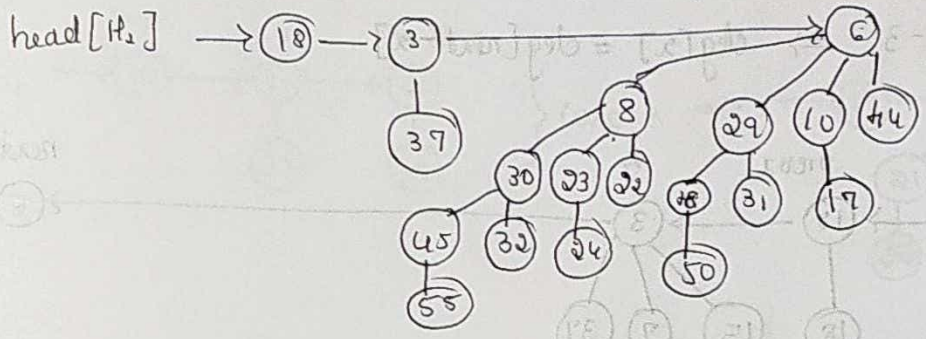
degree[x] = degree[next.x] = degree[sibling[next.x]]

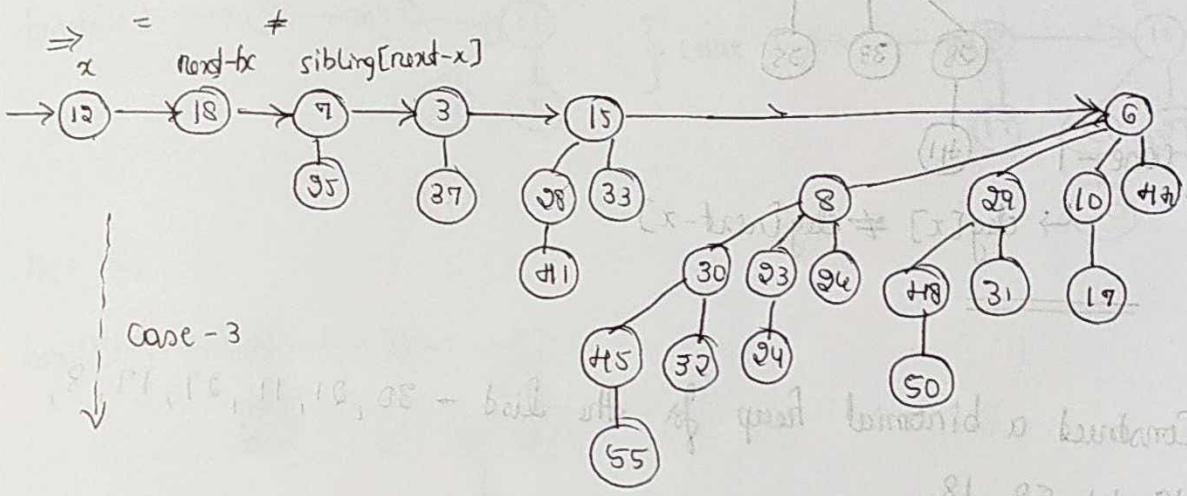
Case-3 & 4:- occures when x is the 2 roots of equal degree

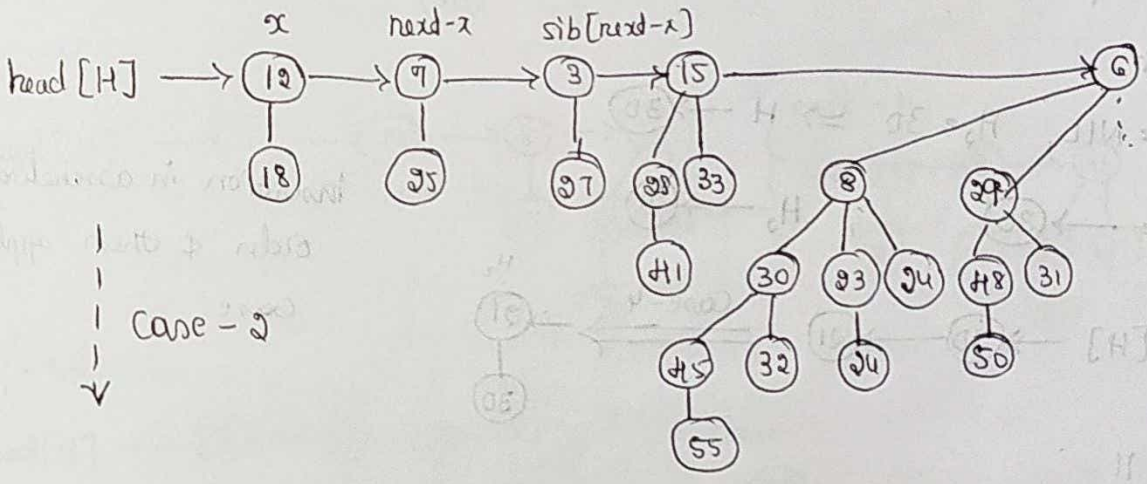
degree[x] = degree[next.x] ≠ degree[sibling[next.x]]

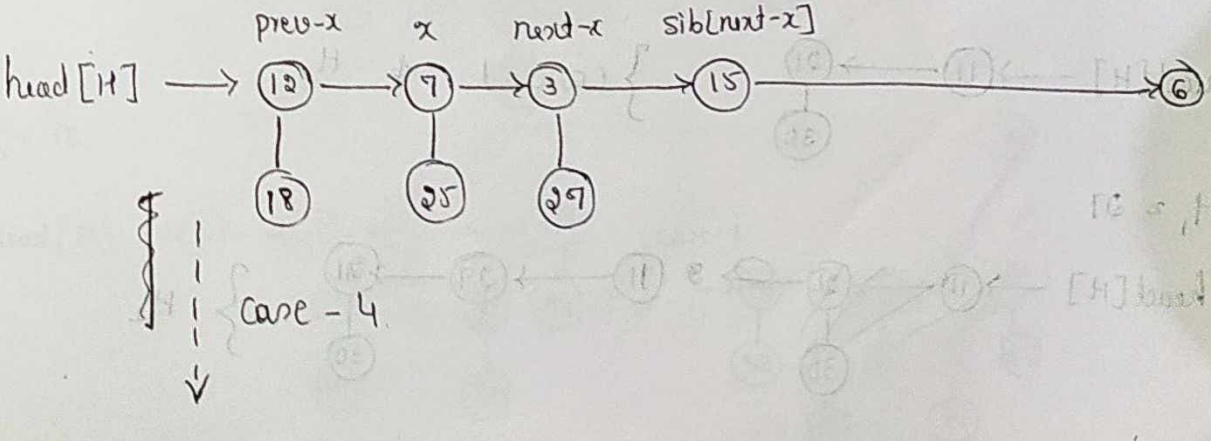
Eg:

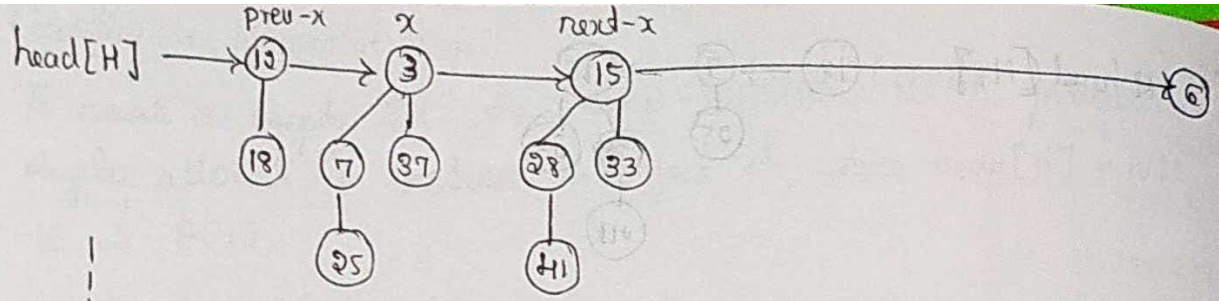

head [H1]
12
7
15
25
28
33
Hk


head [H2]
18
3
6
37
8
29
10
44
30
23
22
45
32
24
50
55

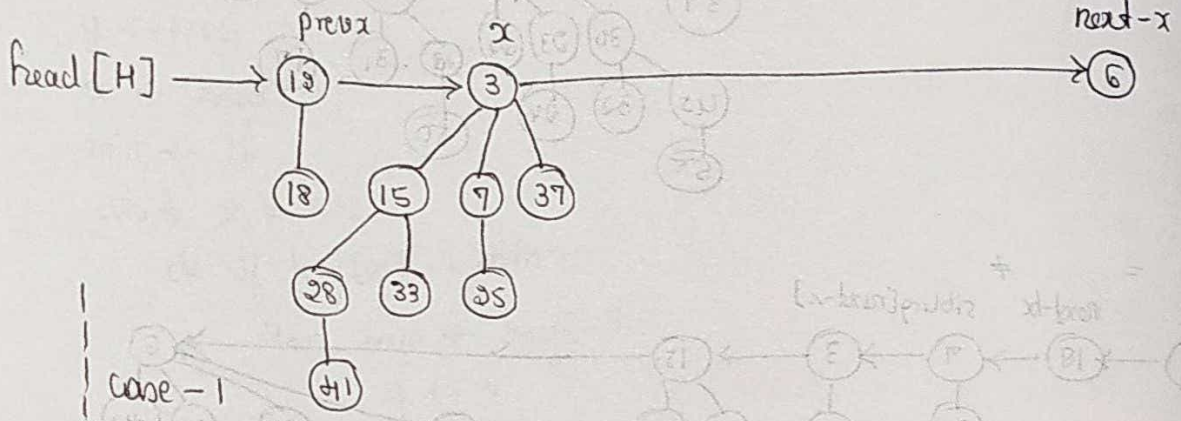

x
next-x
sibling[next-x]
12
18
7
3
15
6
25
37
28
33
8
29
10
44
41
30
23
24
45
32
24
48
31
17
50
55
Case - 3


head [H]
x
next-x
sib[next-x]
12
7
3
15
6
18
25
27
28
33
8
29
41
30
23
24
48
31
45
32
24
50
55
Case - 2


head [H]
prev-x
x
next-x
sib[next-x]
12
7
3
15
6
18
25
27
Case - 4



Case - 3 $\Rightarrow$ deg[x] = deg[next-x]



Case - 1

↳ deg[x] $\neq$ deg[next-x]

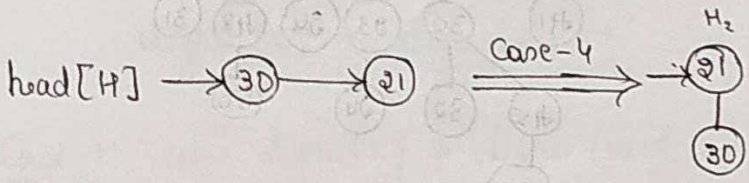
Eg: Construct a binomial heap for the list - 30, 21, 11, 27, 17, 8, 19, 41, 58, 18.

Soln:-

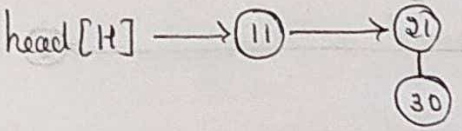
H_1 = NIL, H_2 = 30 $\Rightarrow$ H → (30)

H_1 → (30) , H_2 → (21)

Insertion in assending order & then apply case.

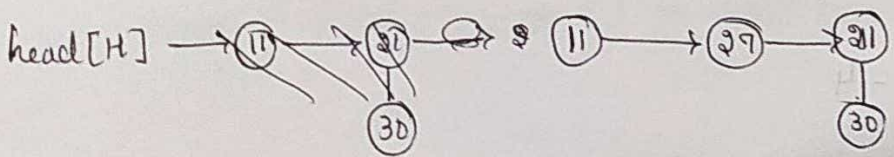


H_1 = 11



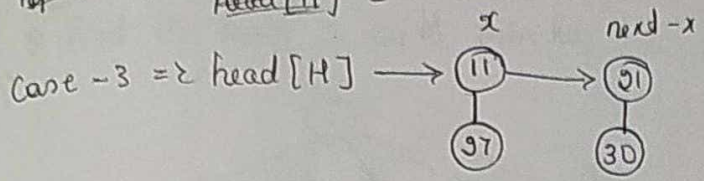
} Case - 1 & H_2

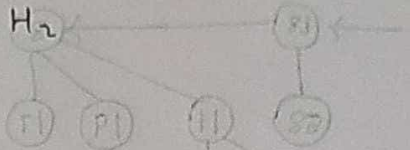
H_1 = 27



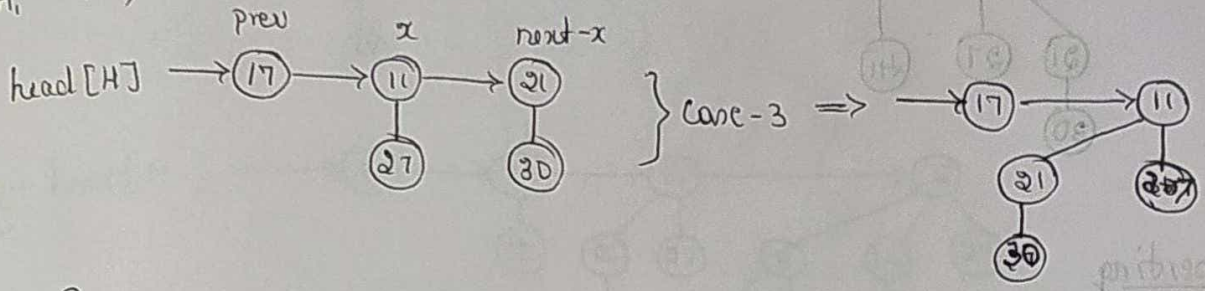
} H_2

$H_1 = 17$, head[H] =

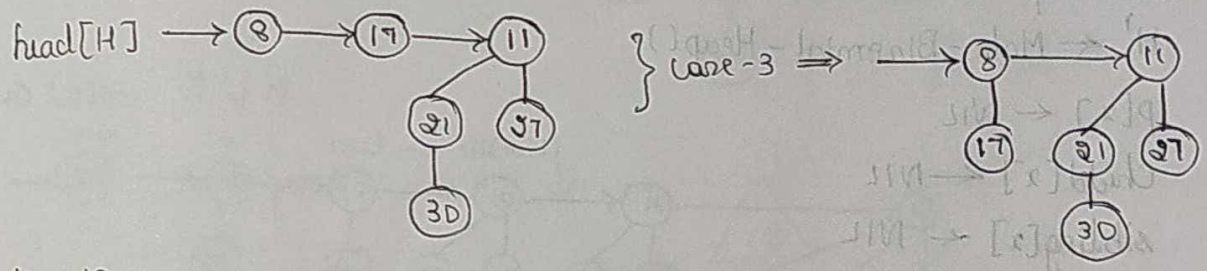




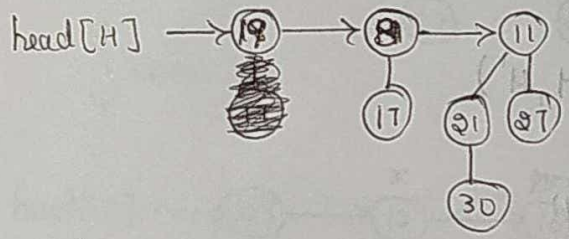
$H_1 = 17$,



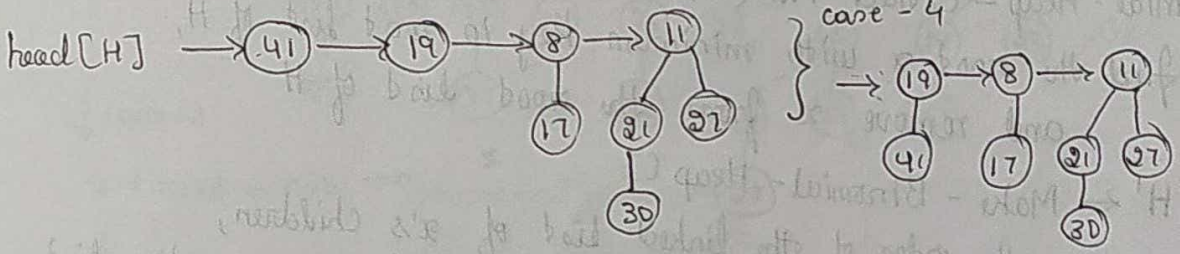
$H_1 = 8$,



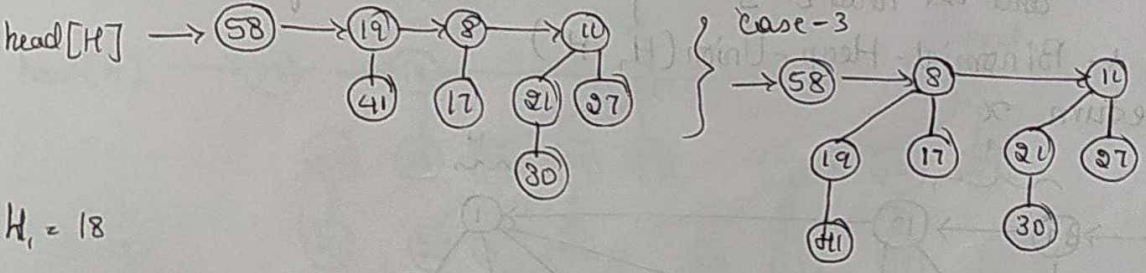
$H_1 = 19$,



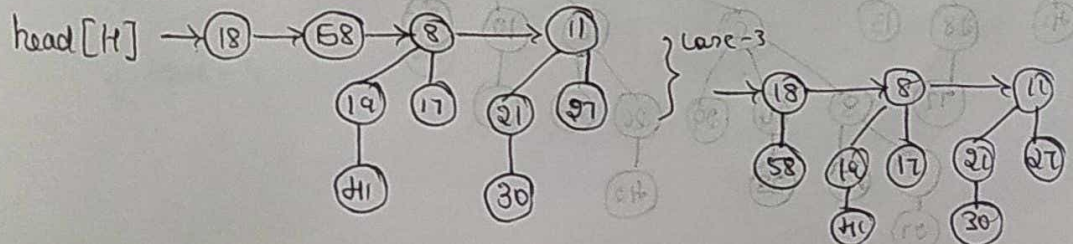
$H_1 = 41$



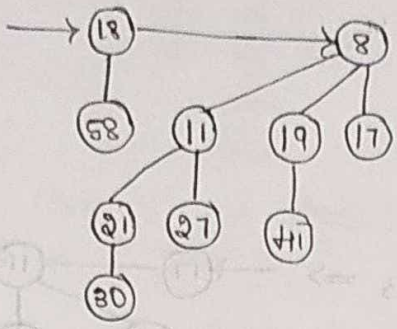
$H_1 = 58$



$H_1 = 18$



Case - 3



Inserting

Binomial-Heap-Insert (H, x)

H' ← Make-Binomial-Heap()

P[x] ← NIL

child[x] ← NIL

sibling[x] ← NIL

degree[x] ← 0

head[H'] ← x

H ← Binomial-Heap-Union(H, H')

Extracting the node with minimum key:-

Binomial-Heap-Extract-Min(H)

Find the root x with minimum key in root list of H,
and remove x from the root list of H

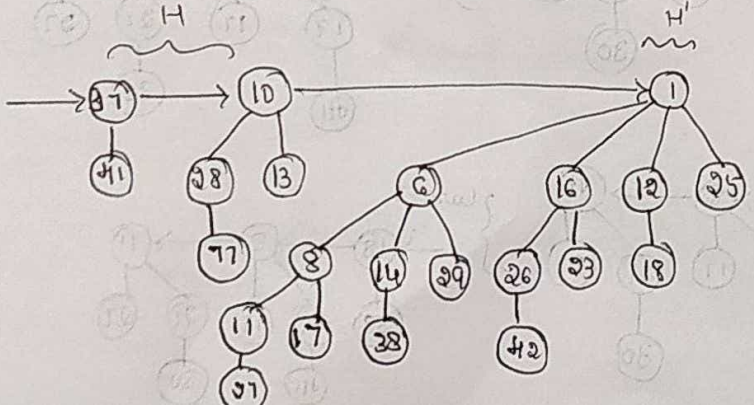
H' ← Make-Binomial-Heap()

reverse the order of the linked list of x's children,
and set head[H'] to point to the head of resulting list

H ← Binomial-Heap-Union(H, H')

return x

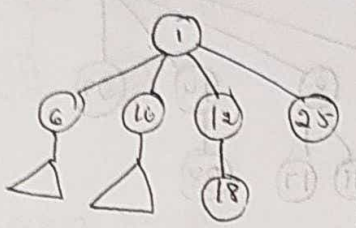
Eg:-



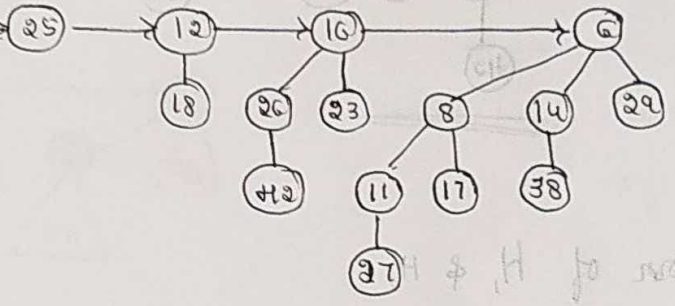
Soln:-

Q. Find the node x with min key = 1 & remove x

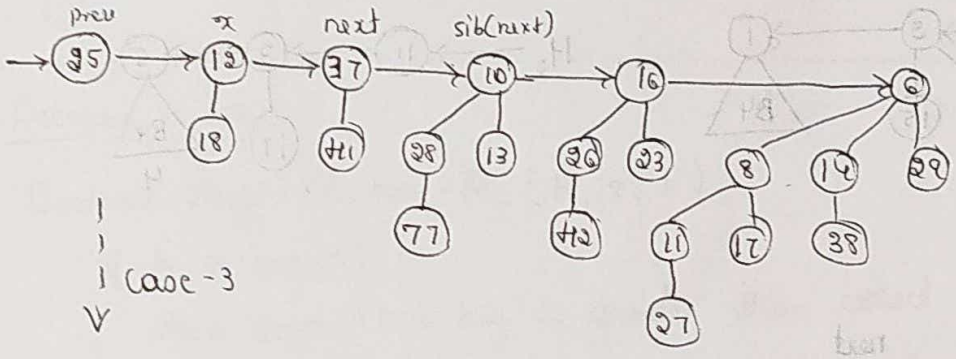
⇒ :.



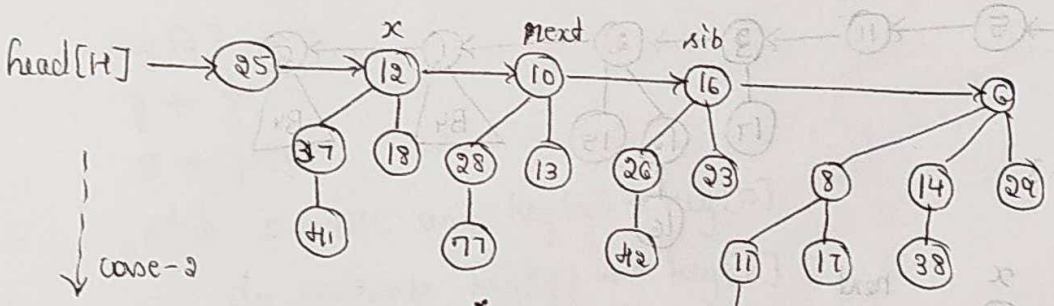
⇒ :. head[H'] →



⇒ Union H U H'



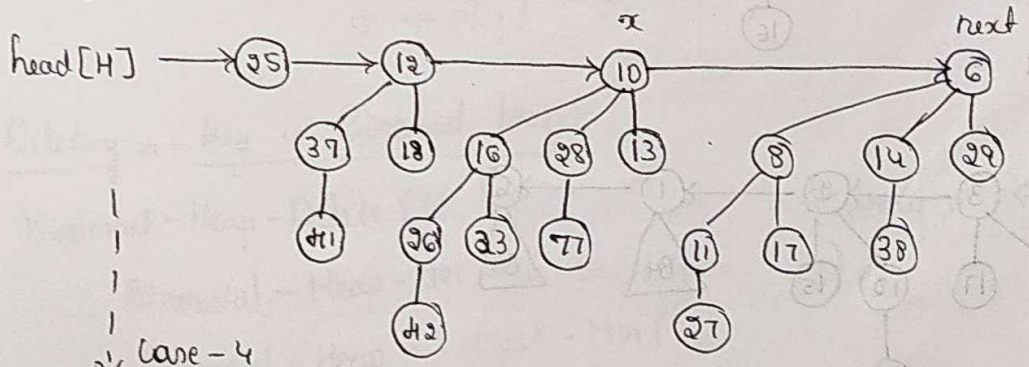
Case-3



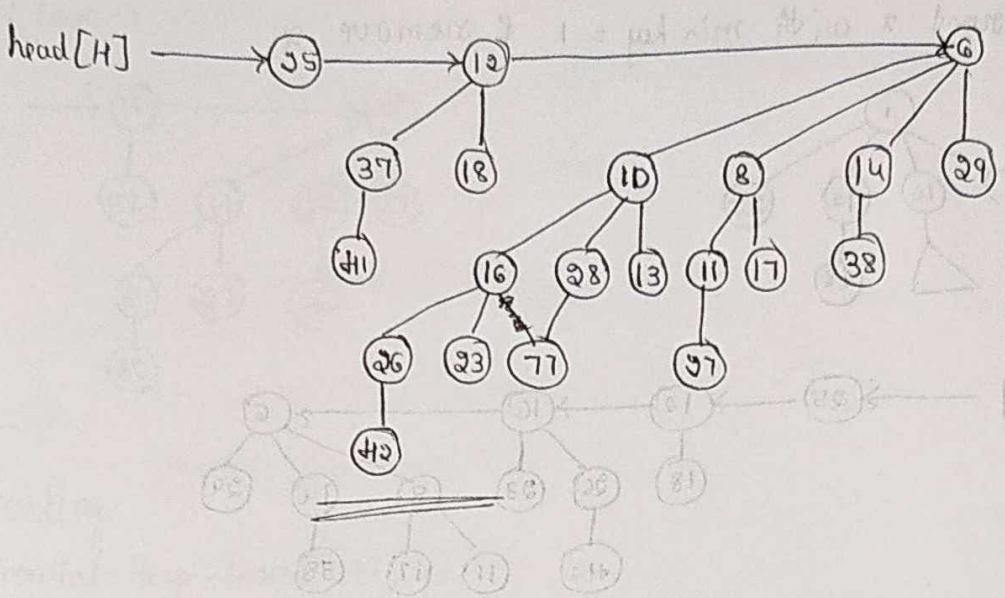
Case-2

x = next = sib ⇒

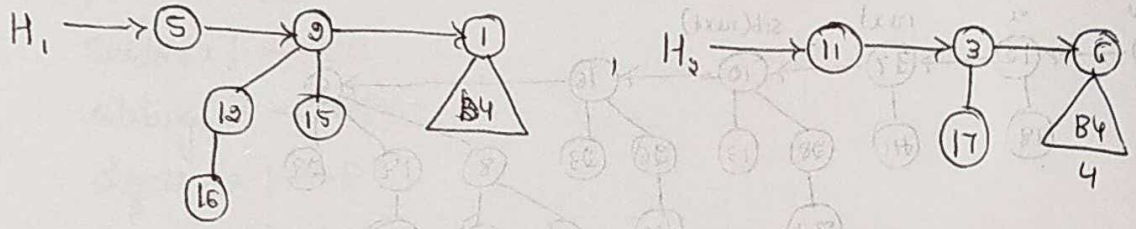
Case-3



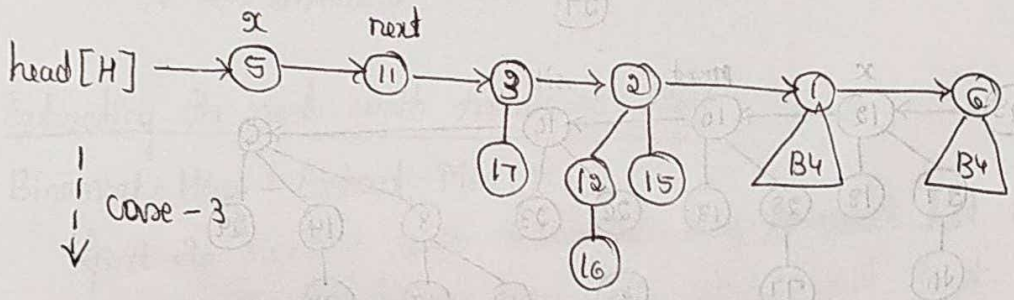
Case-4



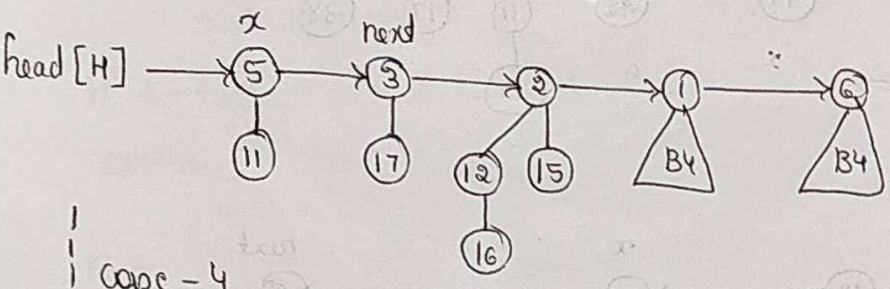
Find the union of H_1 & H_2



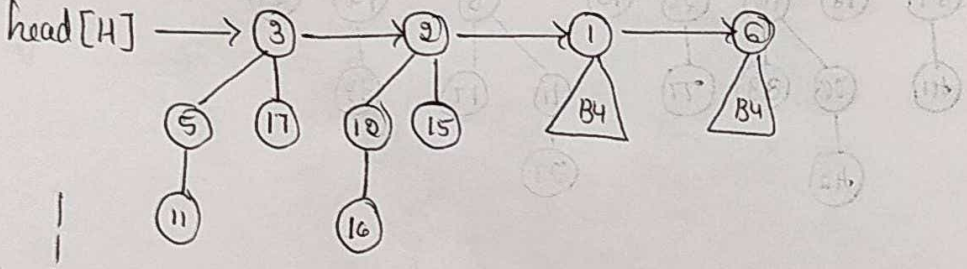
Soln:-



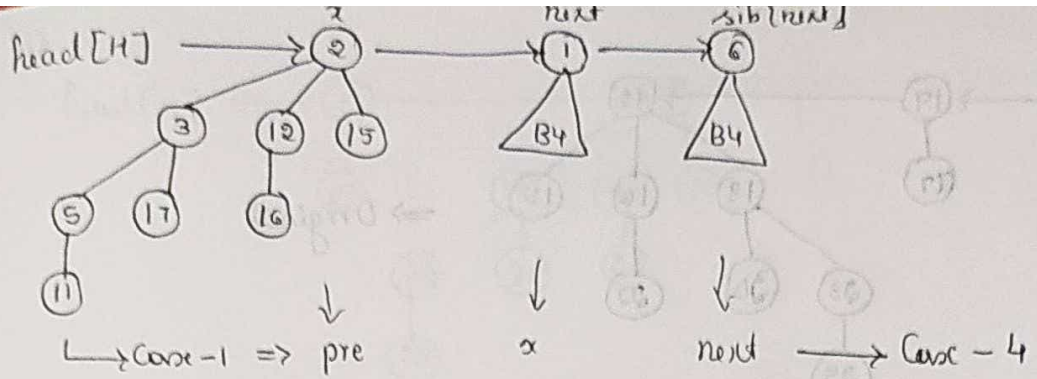
case - 3

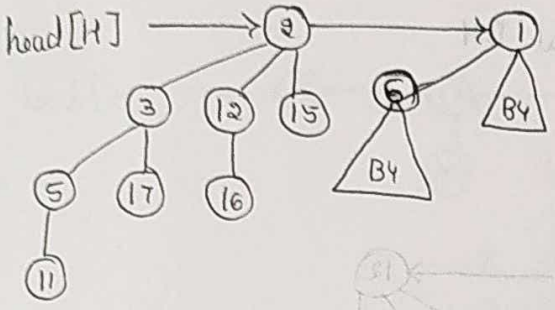


case - 4



case - 4





Decrease a key:-

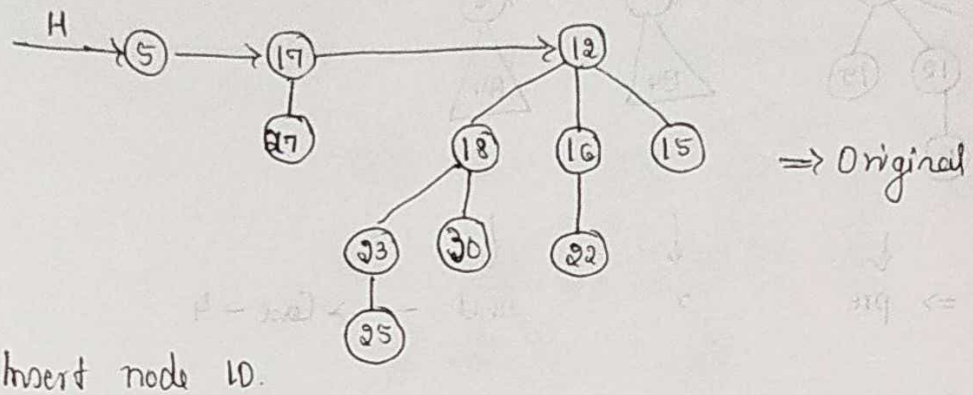
```
Binomial-Heap-Decrease-Key (H, x, k)
    if k > key[x]
       then error "new key is greater than correct key"
    key[x] ← k
    y ← x
    z ← p[y]
    while z ≠ NIL and key[y] < key[z]
          do exchange key[y] ↔ key[z]
             ▷ if y & z have satellite fields, exchange them too.
             y ← z
             z ← p[y]
```

Deleting a Key in Binomial Heaps:-

```
Binomial-Heap-Delete (H, x)
    Binomial-Heap-Decrease-Key (H, x, -∞)
    Binomial-Heap-Extract-Min (H)
```

→ very minimal value.

Given,



⇒ Original

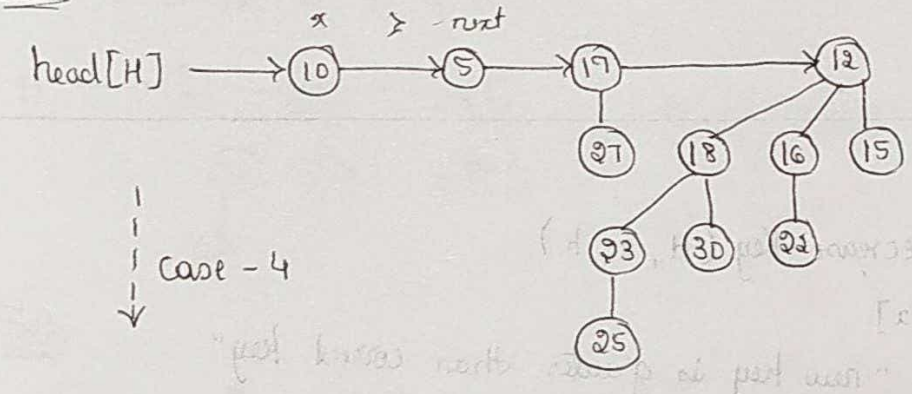
i) Insert node 10.

ii) Extract min from original BH

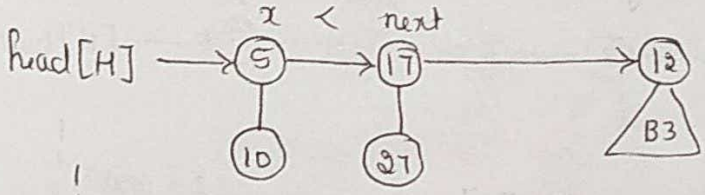
iii) Decrease node 23 → 9 from original BH

iv) Delete 16 from original BH

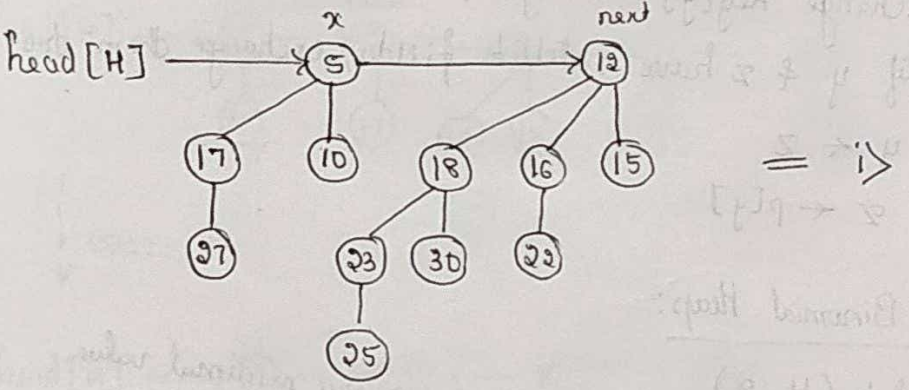
Soln:-



Case - 4

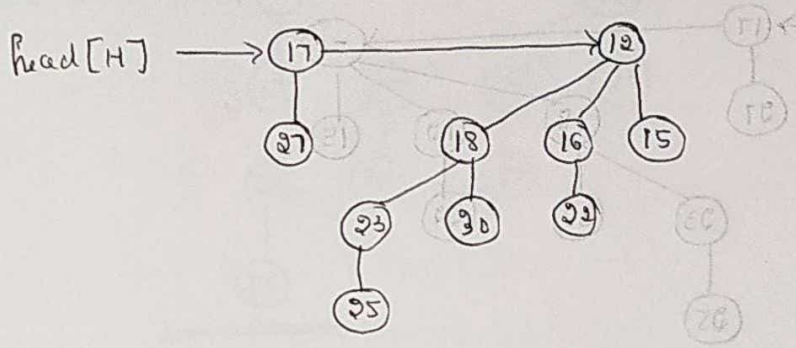


Case - 3

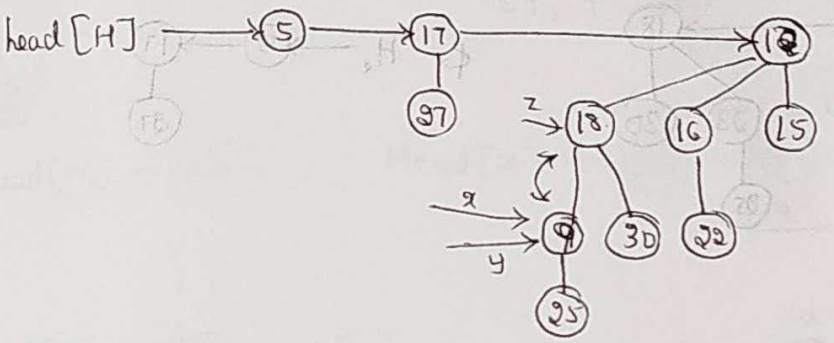


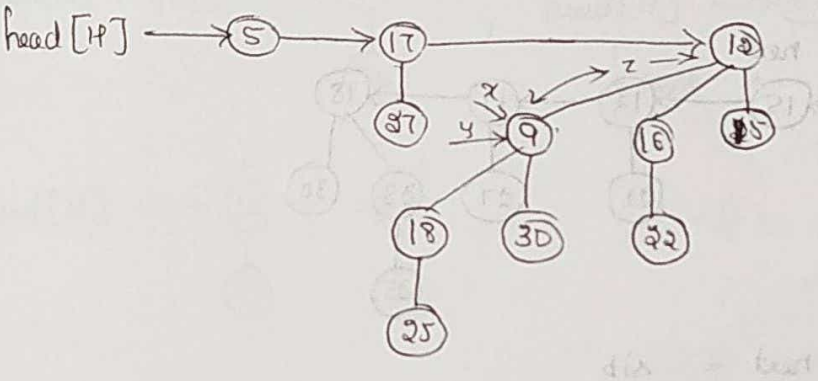
= i)

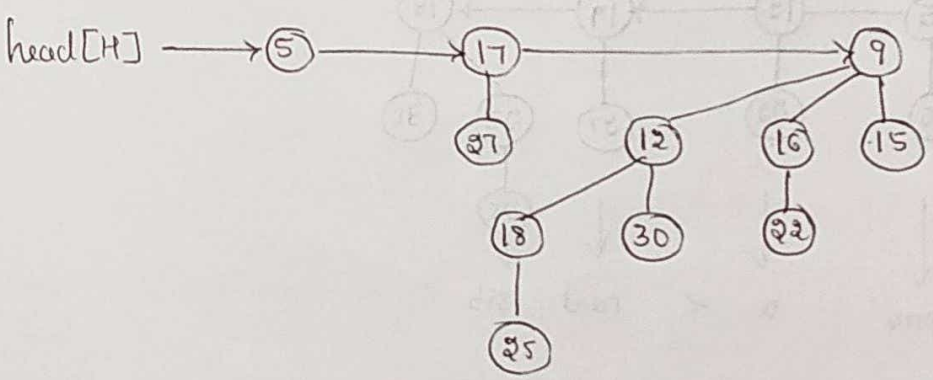
ii)



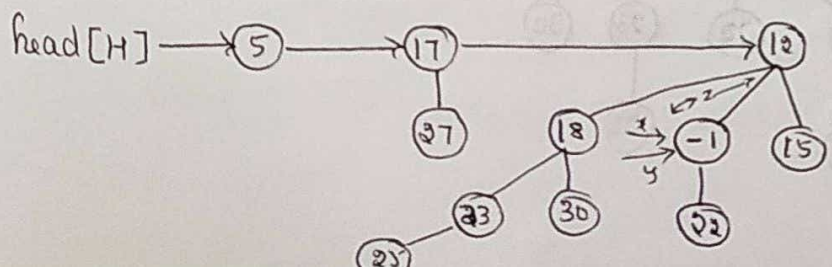
iii)

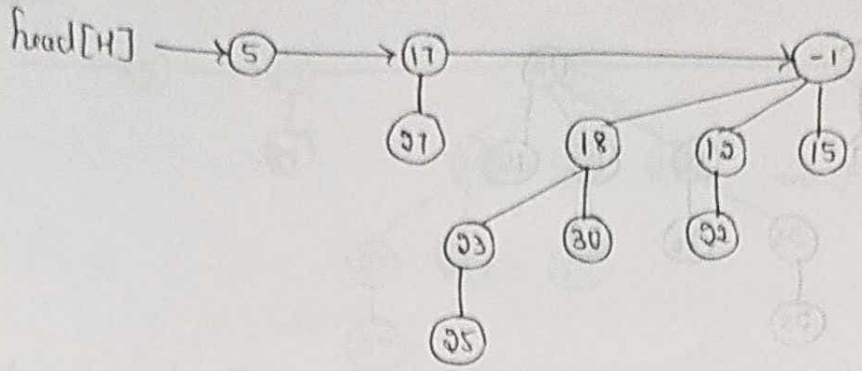




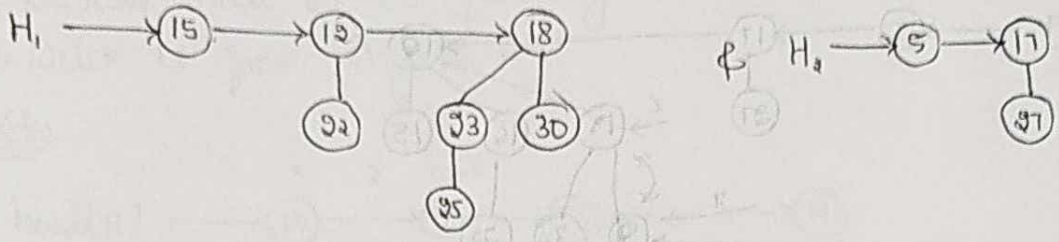


iv) Put K = -1 & replace 16 with K ⟶ Decrease-Key — (1)

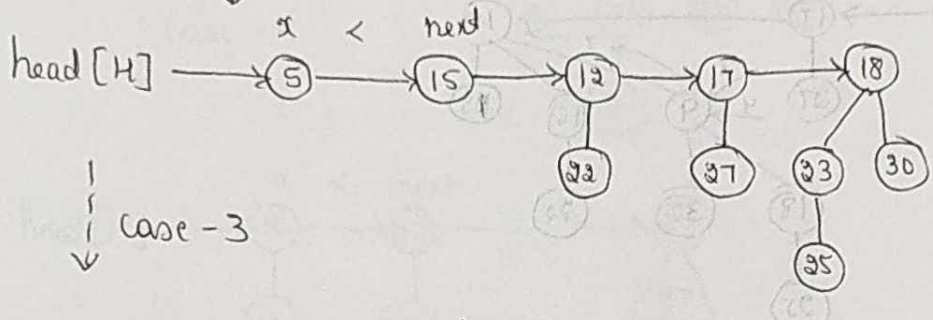




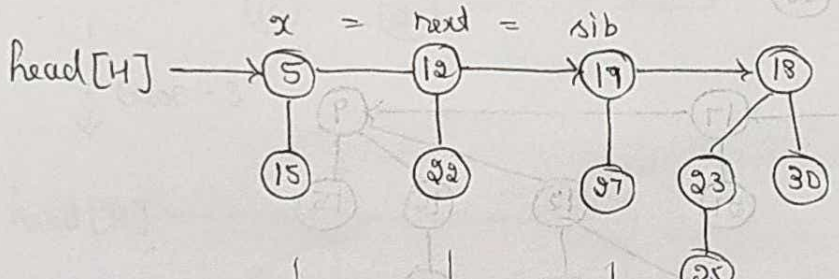
② Extract min ( )



$H_1 \cup H_2 \Rightarrow$



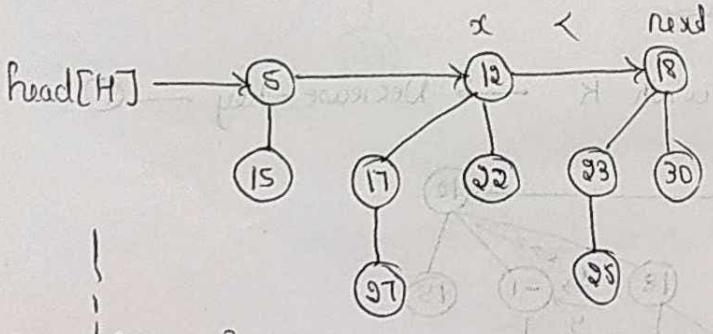
case - 3



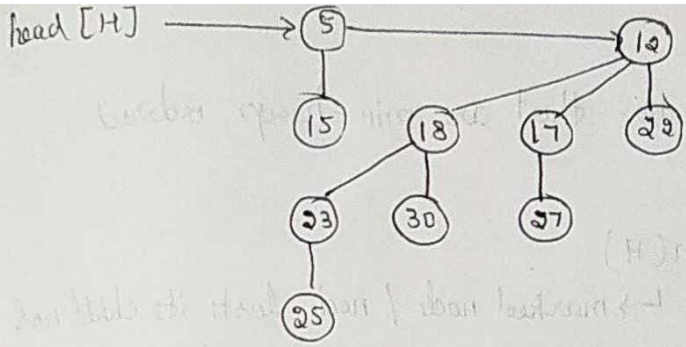
move 1 step

prev x < next sib

case - 3.

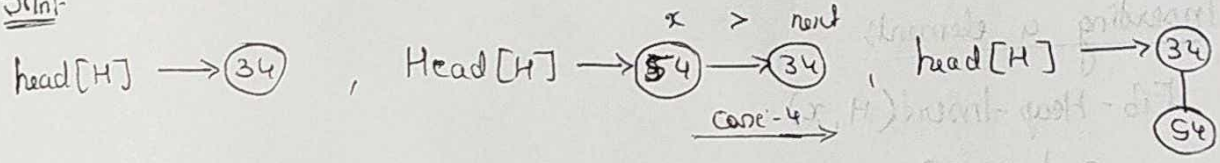


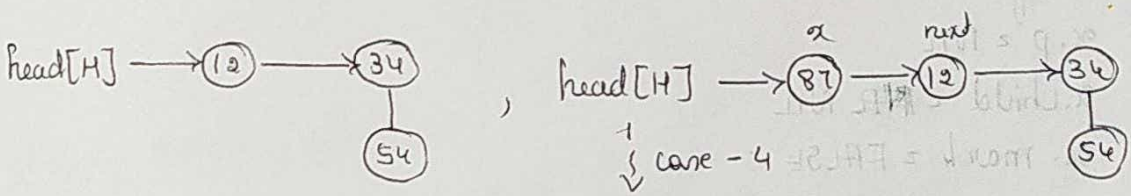
case - 3

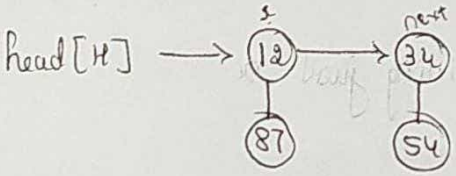


Constructed binomial Heap → 34, 54, 12, 87, 93, 19, 73, 64, 51, 59, 9, and N=11

Soln:-







Fibonacci Heap:-

* It is a collection of rooted tree that are min-heap ordered.
* Potential function,

$$\phi(H) = t(H) + 2m(H)$$

$t(H)$ → Total no. of ordered nodes.
$m(H)$ → marked node / node lost its child node

Creating new Fibonacci Heap:-

MAKE-FIB-HEAP.

Inserting a element:

```
Fib-Heap-Insert(H, x)
    x.degree = 0
    x.p = NIL
    x.child = NIL
    x.mark = FALSE
    if H.min == NIL
        create a root list for H combining just x
        H.min = x
    else insert x into H's root list
        if x.key < H.min.key
            H.min = x
    H.n = H.n + 1
```

* Amortized cost H is existing Fib-Heap, H' is after inserting then

$$t(H') = t(H) + 1 \quad \& \quad m(H') = m(H)$$

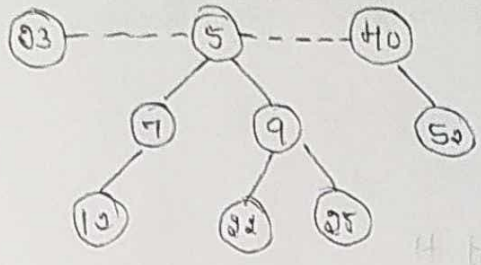
$$\phi(H) = t(H) + 2m(H)$$

$$\therefore \phi(H') - \phi(H) = [t(H') + 2m(H')] - [t(H) + 2m(H)]$$

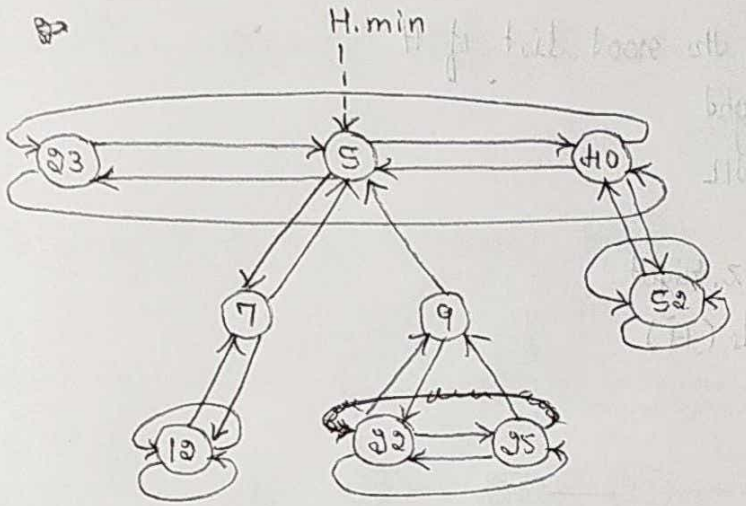
$$= t(H') + 1 + 2m(H) - t(H) - 2m(H)$$

$$= 1$$

Q:



⇒



Finding the minimum node:-

* The minimum node is pointer pointing to by H.min.
* So finding minimum node is O(1)

Uniting 2 Fibonacci Heap:-

Fib-Heap-Union(H_1, H_2)

H = Make-Fib-Heap()

H.min = H_1.min

Concatenate the root list of H_2 with the root list of H

if (H_1.min == NIL) or (H_2.min ≠ NIL and H_2.min.key < H_1.min.key)

H_1.min = H_2.min

H.n = H_1.n + H_2.n

return H.

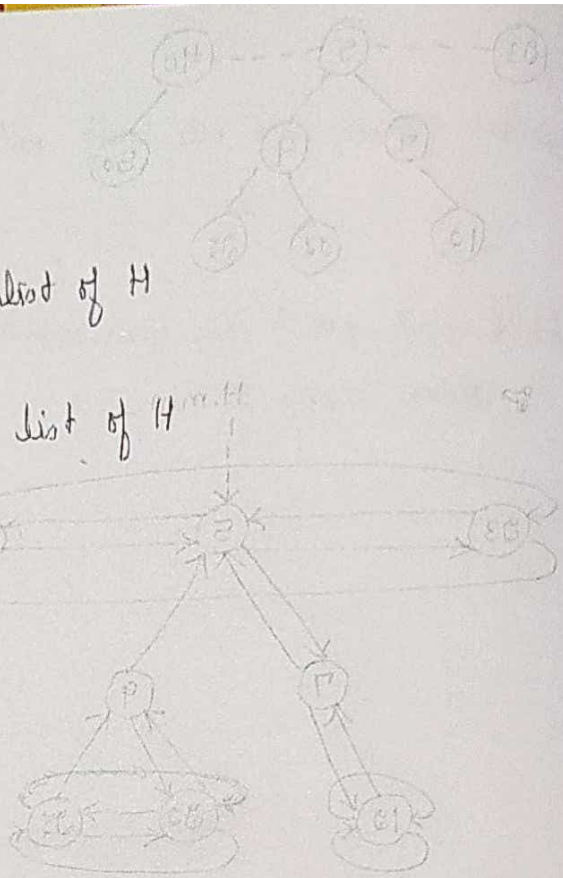
* The change in potential is,

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

$= [t(H) + 2m(H)] - [(t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))]$

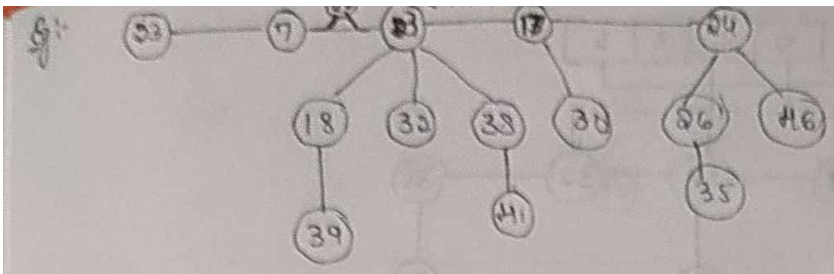
$= 0$ // $\because t(H) = t(H_1) + t(H_2)$ & $m(H) = m(H_1) + m(H_2)$

```
Fib-Heap-Extract-Min (H)
    z = H.min
    if z ≠ NIL
        for each child x of z
            add x to the root list of H
            x.p = NIL
        remove z from the root list of H
        if z == z.right
            H.min = NIL
        else
            H.min = z.right
            Consolidate (H)
        H.n = H.n - 1
    return z
```

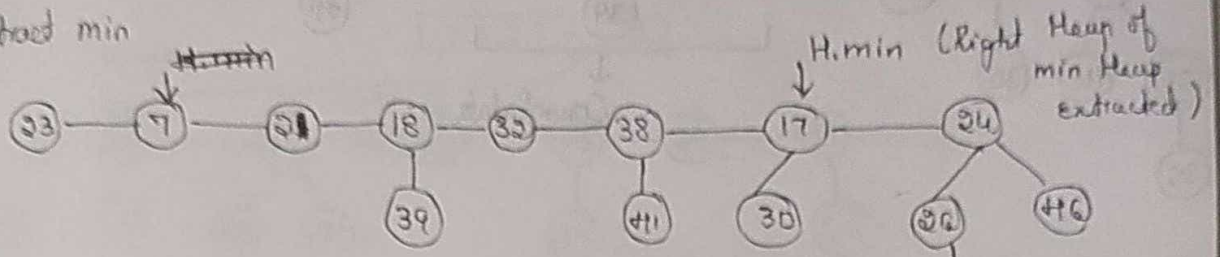


```
Consolidate(H)
    let A[0,... D(H.n)] be a new array
    for i=0 to D(H.n)
        A[i] = NIL
    for each node w in the root list of H
        x = w
        d = x.degree
        while A[d] ≠ NIL
            y = A[d]
            if x.key > y.key
                exchange x and y
            Fib-Heap-Link (H, y, x)
            A[d] = NIL
            d = d + 1
        A[d] = x
    H.min = NIL
    for i=0 to D(H.n)
        if A[i] ≠ NIL
            if H.min == NIL
```

Sol:

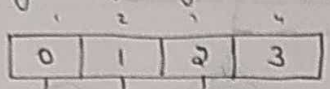


Extract min

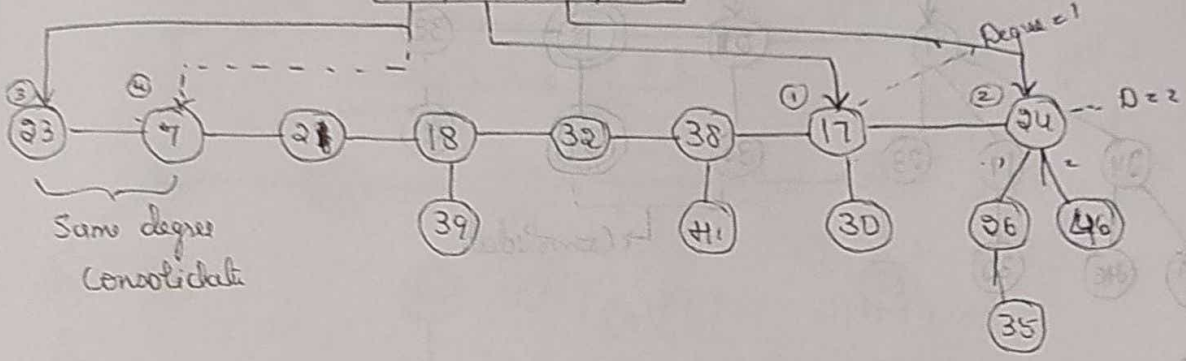


1)

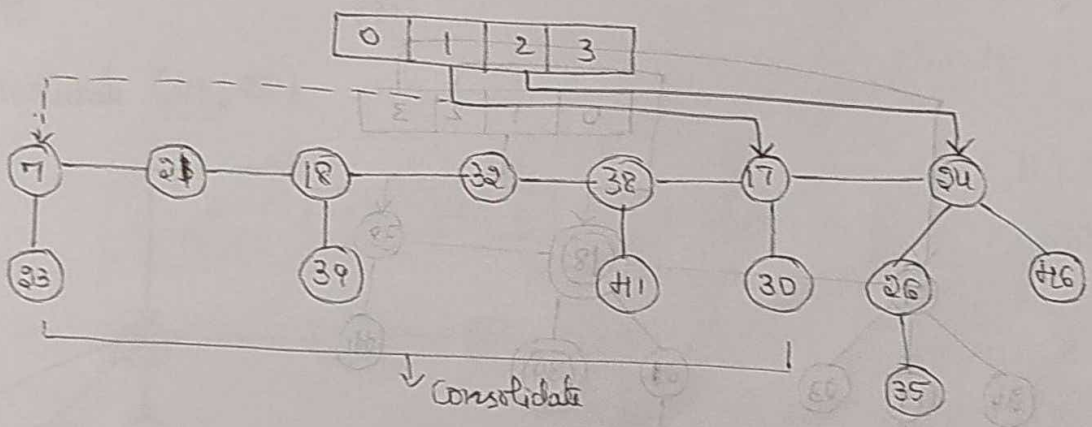
Create a array A of degree of original Heap + 1
3 + 1 = 4



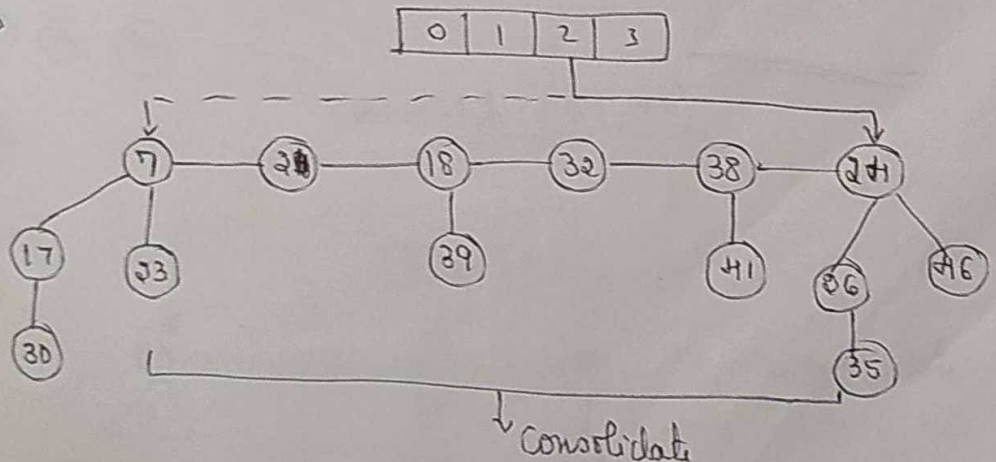
2)



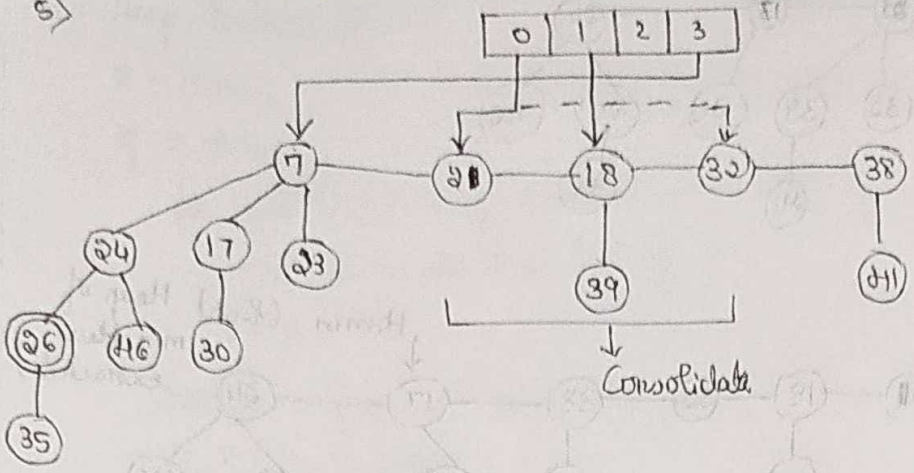
3)



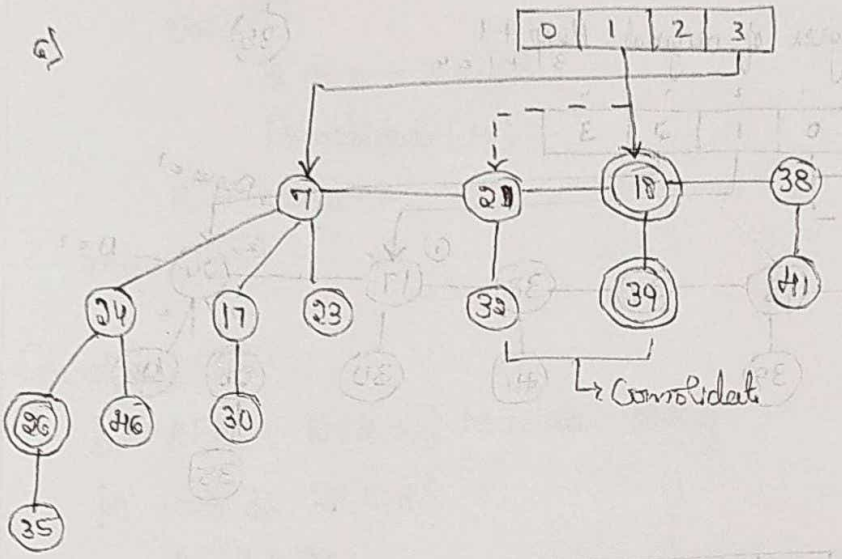
4)



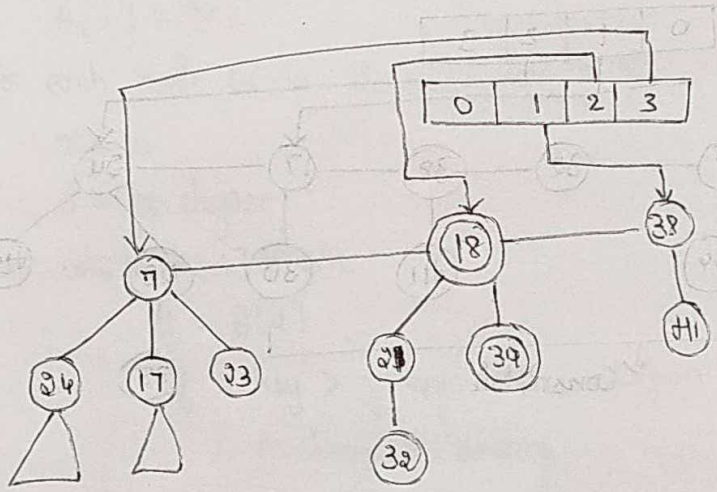
5)


0
1
2
3
7
21
18
32
38
24
17
23
39
41
26
46
30
35
Consolidate

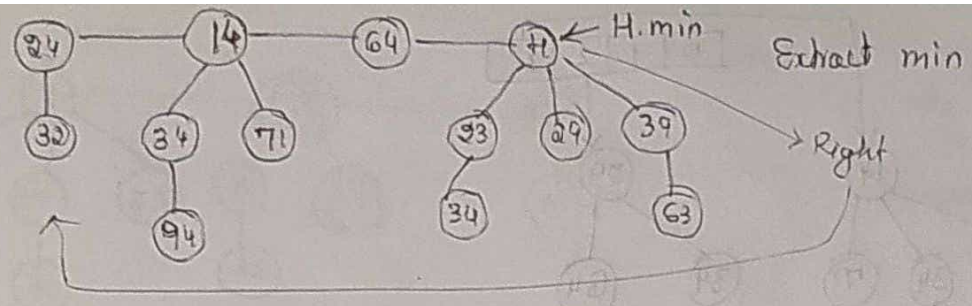
6)


0
1
2
3
7
21
18
38
24
17
23
32
39
41
26
46
30
35
Consolidate

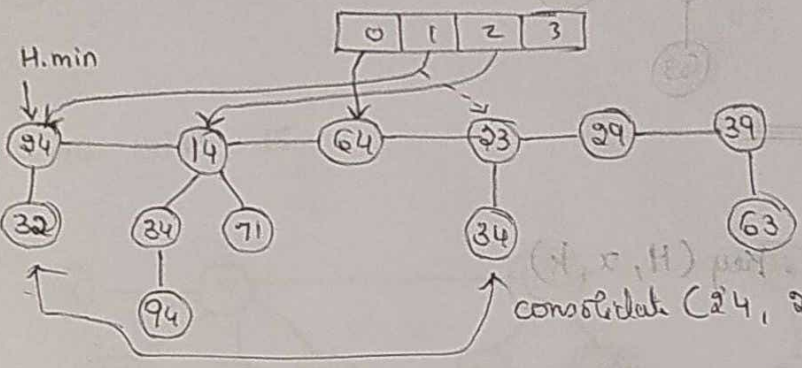
7)


0
1
2
3
18
38
7
41
24
17
23
21
39
32

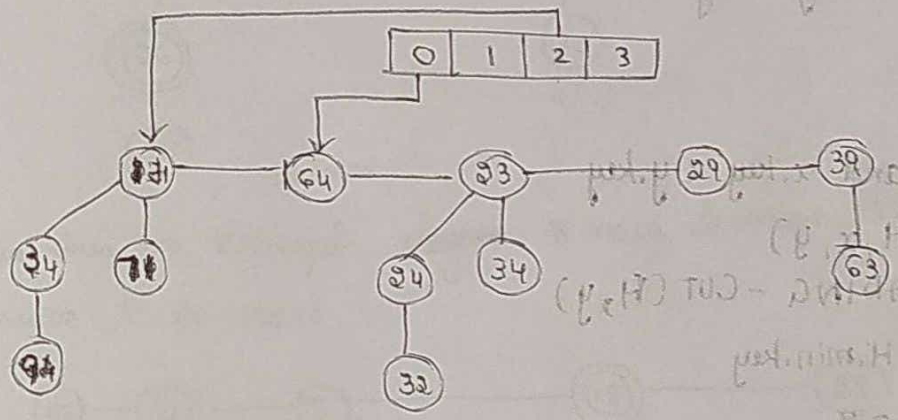
Eg:



Soln:-

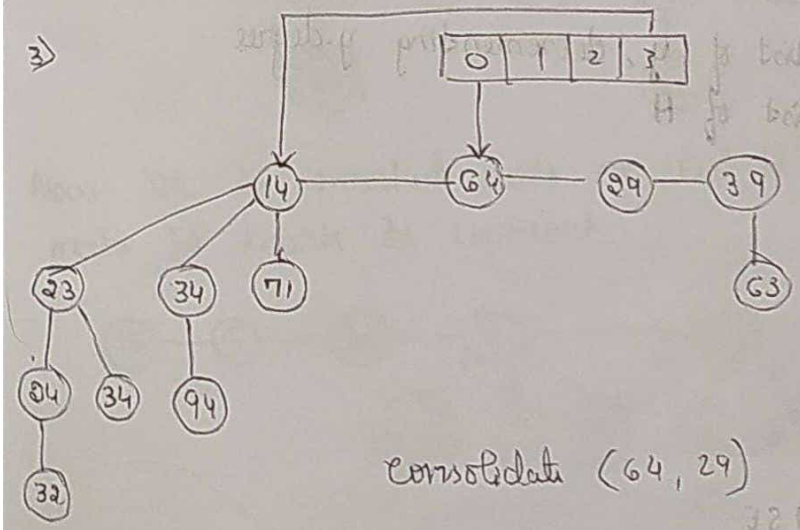


2)



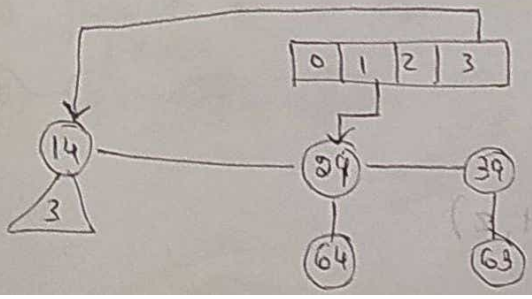
Consolidate (14, 23)

3)



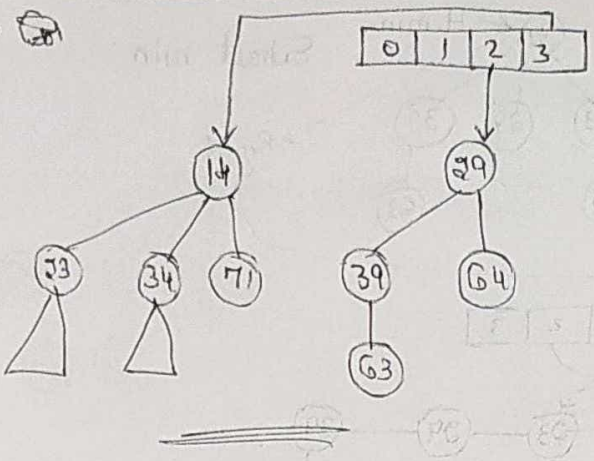
Consolidate (64, 29)

4)



Consolidate (29, 39)

5)



Fib-Heap-Decrease-Key (H, x, k)

```
if k > x.key
    error "new key is greater than current key"
x.key = k
y = x.p
if y ≠ NIL and x.key < y.key
    CUT (H, x, y)
    CASCADING-CUT (H, y)
if x.key < H.min.key
    H.min = x
```

CUT (H, x, y)

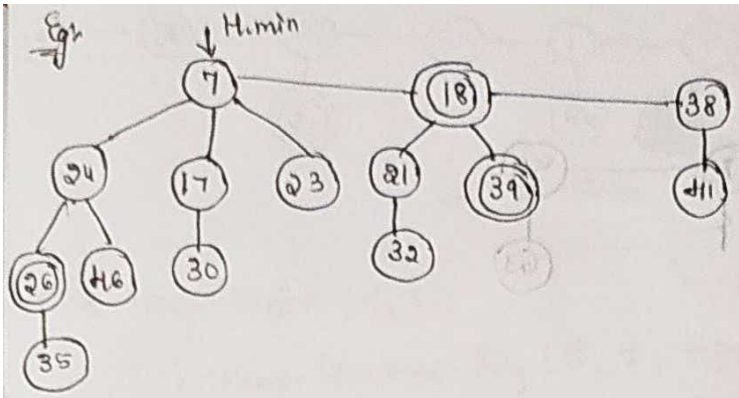
remove x from child list of y, decrementing y.degree
add x to the root list of H

```
x.p = NIL
x.mark = FALSE
```

CASCADING-CUT (H, y)

```
z = y.p
if z ≠ NIL
    if y.mark == FALSE
        y.mark = TRUE
    else
        CUT (H, y, z)
        CASCADING-CUT (H, z)
```

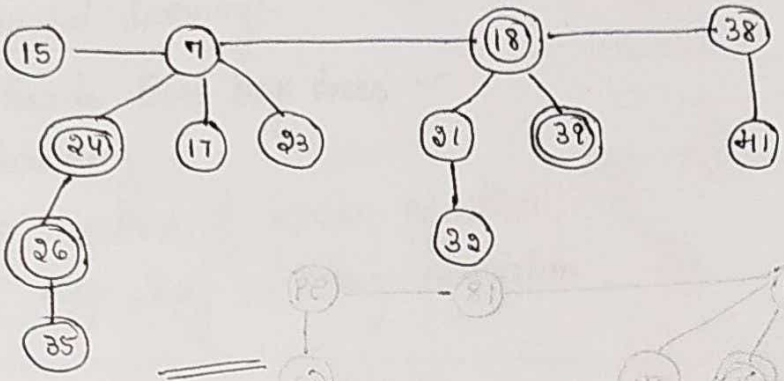
Eg:



Here (0) → represents the node loosts its child.
→ marked node

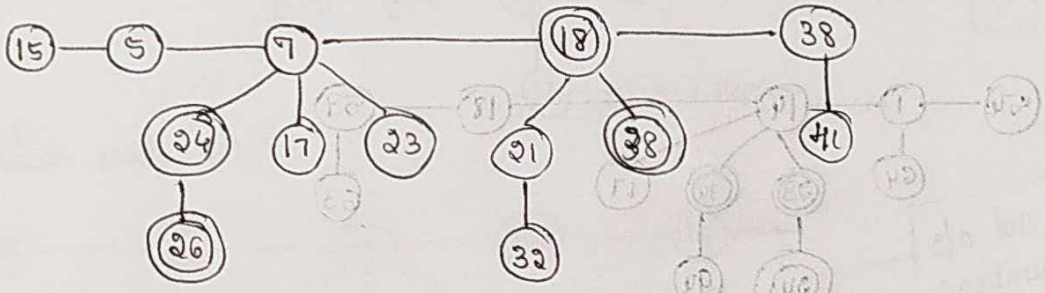
Decrease 46 with 15

Sol:

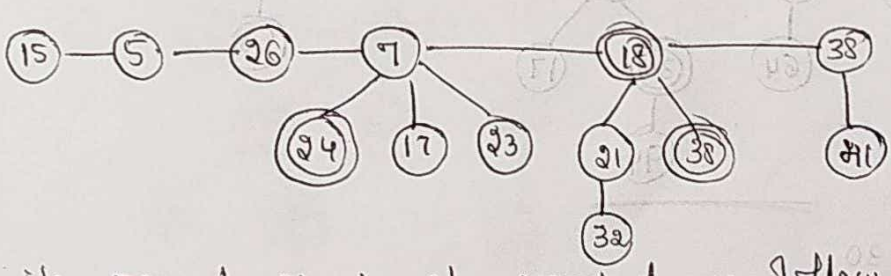


Take this as example ~~degree~~ & now decrease 35 → 5

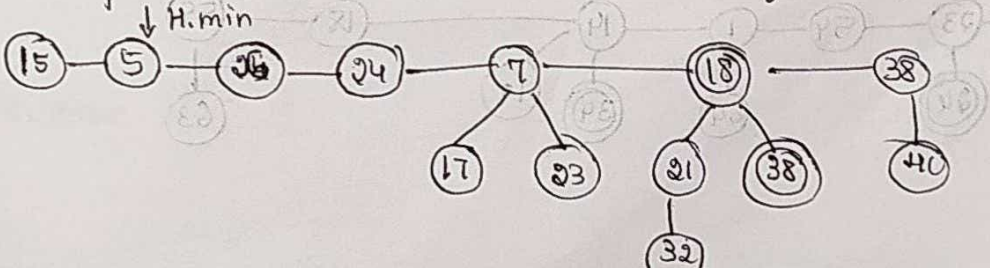
So move 5 to root



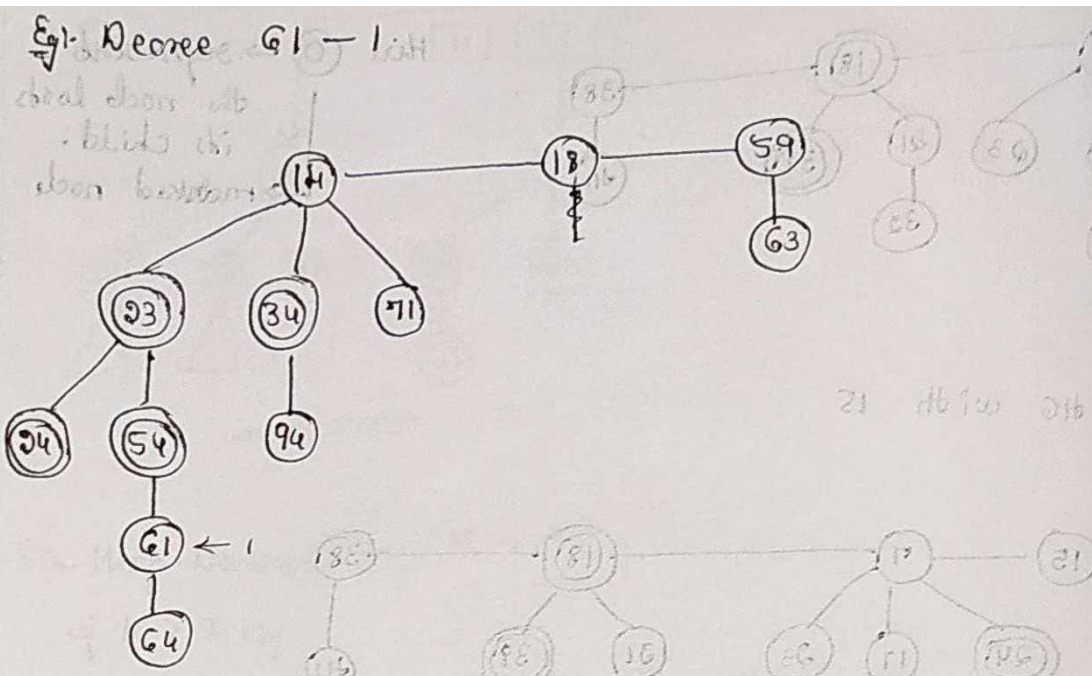
Now 26 is marked node so cut it & add to root. & make it mark to unmark



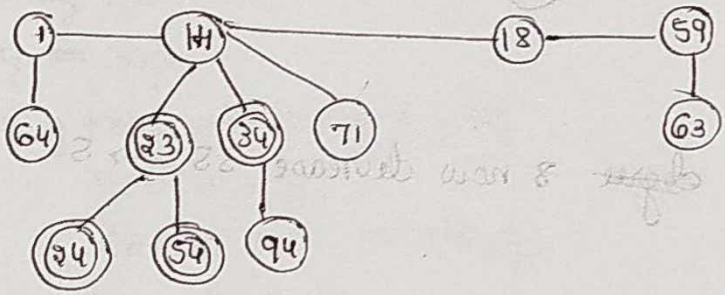
Now its parent 24 is also marked so follow previous procedure

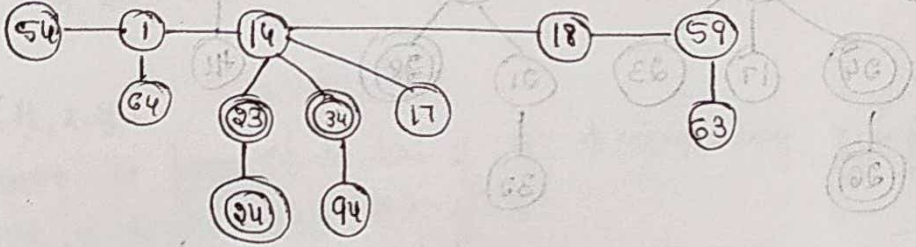


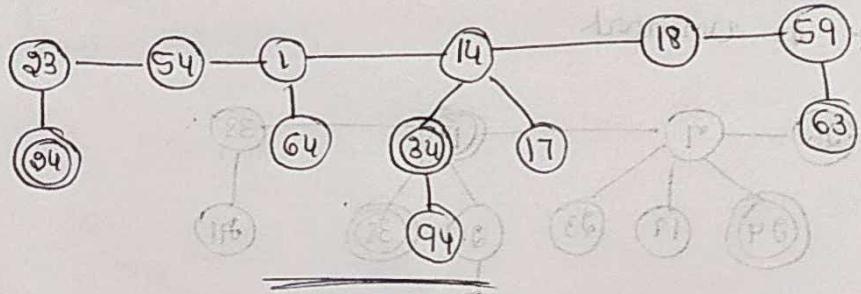
Eg:- Decree 61 → 1



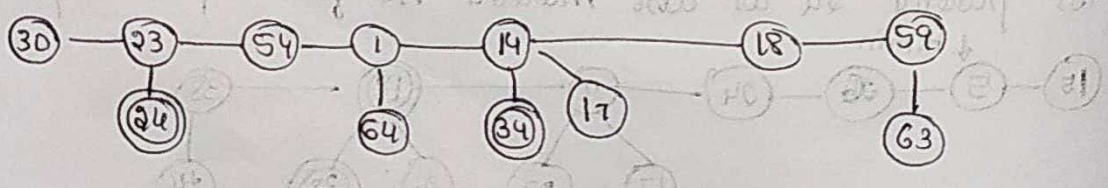
Sol:-

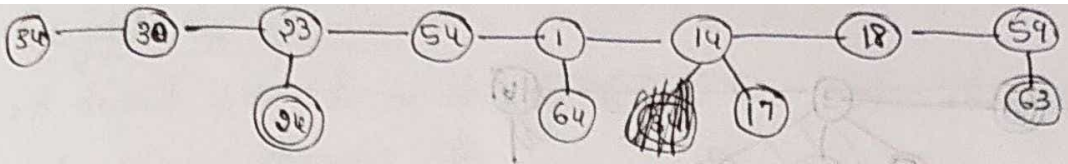






Decrease 94 → 30





Fib-Heap-Delete (H, x)

Fib-Heap-Decrease-Key (H, x, $-\infty$)

Fib-Heap-Extract-Min (H)

Skill directed learning:-

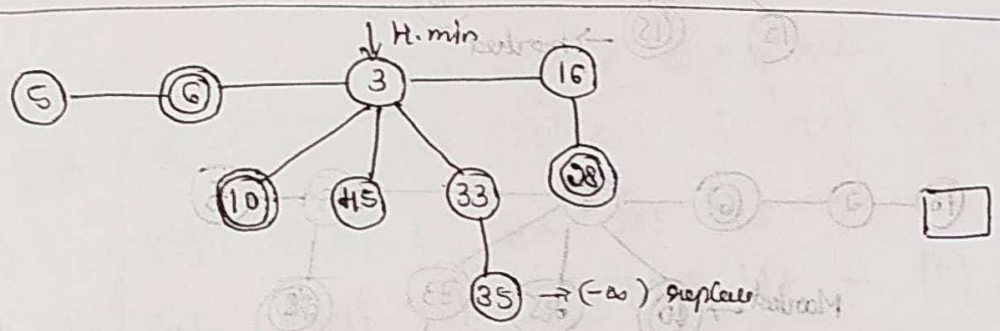
1) Van Aamde Boss ~~Kary~~ trees ✓

2) Multithreading ✓

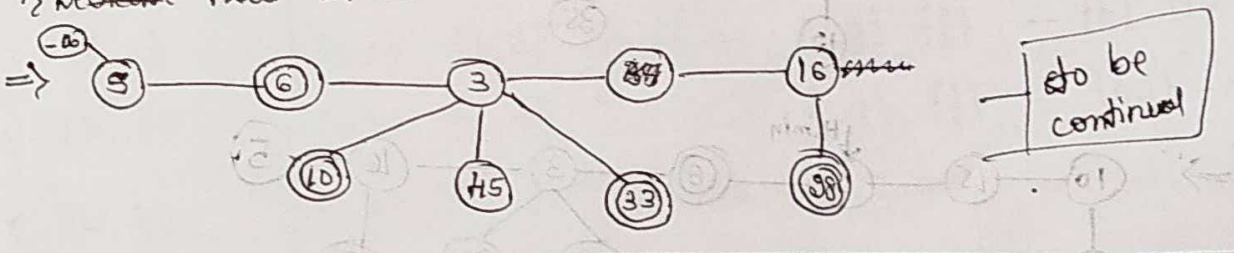
3) Solving Systems of linear equation

Robin Karp string matching algorithm

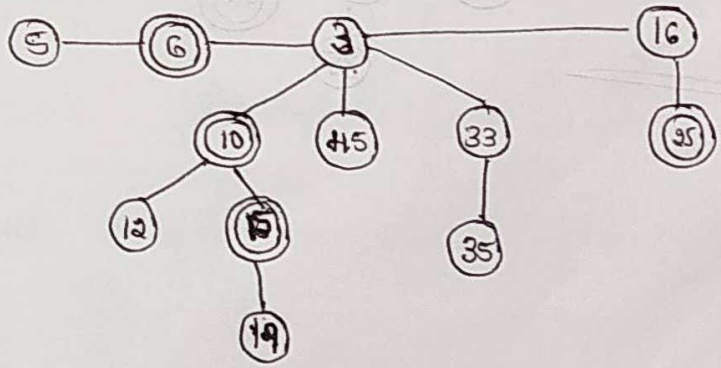
Eg:-



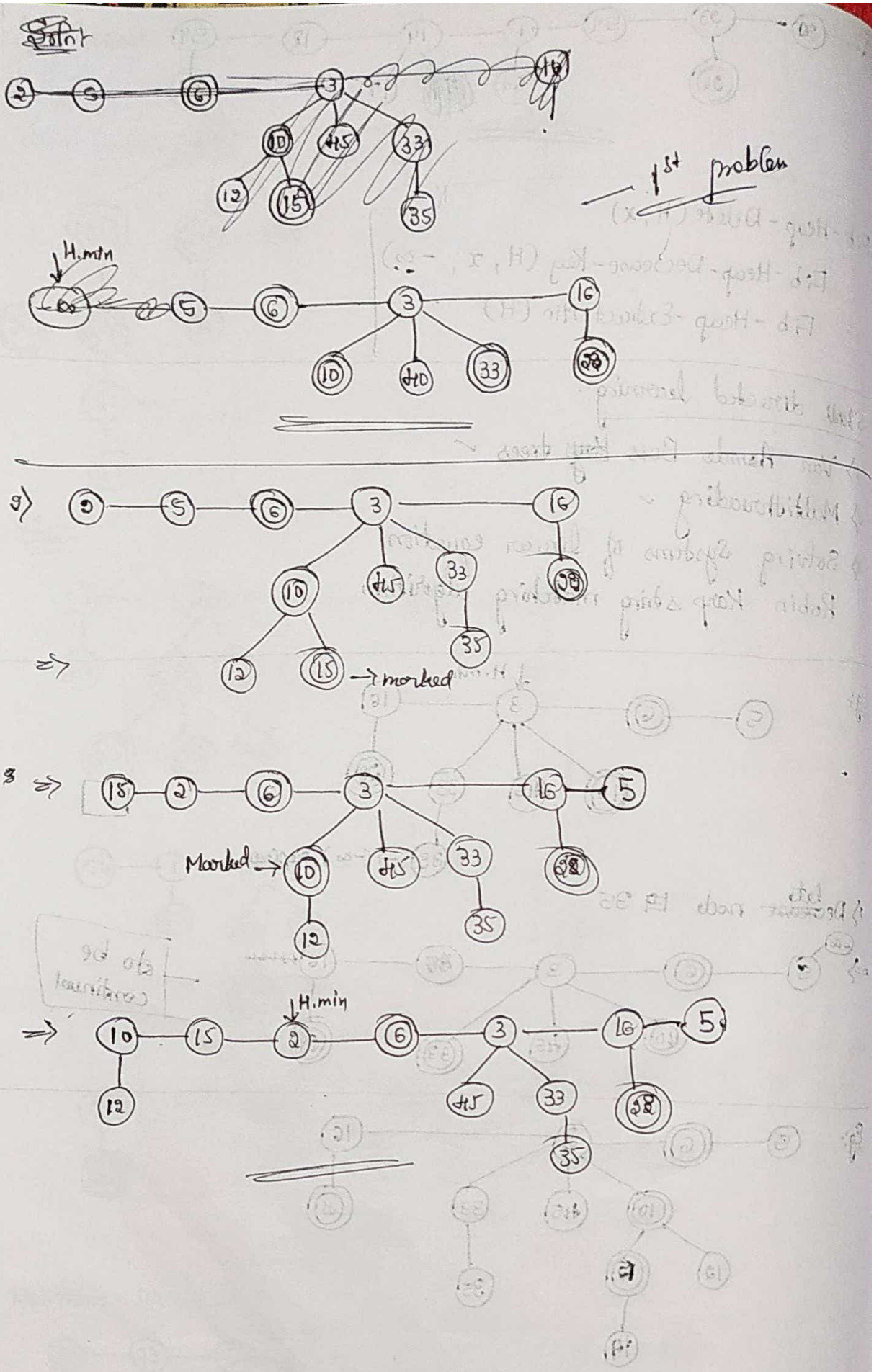
1) Decrease ~~Decrease~~ let node 35



Eg:-



1) Decrease 19 − 2

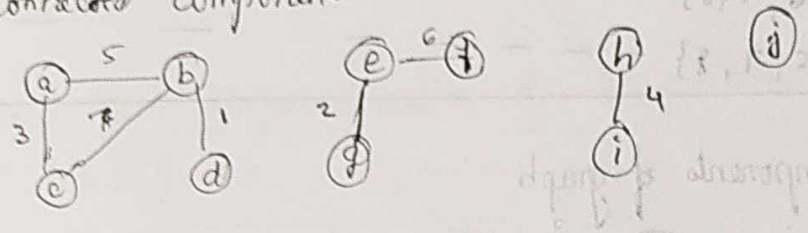

1st problem
H.min
H.min
marked
Marked
H.min

Disjoint Set Operations:

- A disjoint set data structure maintains a collection $S = \{S_1, S_2, \ldots S_k\}$ of disjoint dynamic sets.
- Find-Set(x) → returns a pointer to the representative of the (unique) set containing 'x'.
- Make-Set
- Union

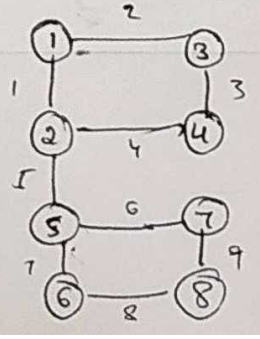
Application of disjoint set DS:

- Connected component



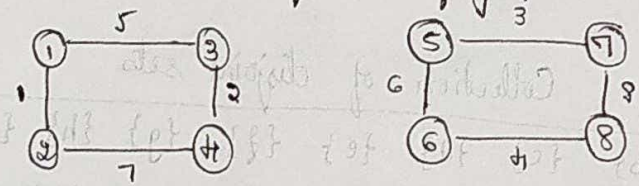
Edge Processed	Collection of disjoint sets									
initial ~~data~~ sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
1 (b, d)	{a}	{b,d}	{c}	~~X~~	{e}	{f}	{g}	{h}	{i}	{j}
2 (e, g)	{a}	{b,d}	{c}	—	{e,g}	{f}	—	{h}	{i}	{j}
3 (a, c)	{a,c}	{b,d}	—	—	{e,g}	{f}	—	{h}	{i}	{j}
4 (h, i)	{a,c}	{b,d}	—	—	{e,g}	{f}	—	{h,i}	—	{j}
5 (a, b)	{a,b,c,d}	—	—	—	{e,g}	{f}	—	{h,i}	—	{j}
6 (e, f)	{a,b,c,d}	—	—	—	{e,f,g}	—	—	{h,i}	—	{j}
7 (b, c)	{a,b,c,d}	—	—	—	{e,f,g}	—	—	{h,i}	—	{j}

Find the connected component of given graph.



Edge Processed	Collection of disjoint Set							
Initial Set	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}
(1,2)	{1,2}	—	{3}	{4}	{5}	{6}	{7}	{8}
(1,3)	{1,2,3}	—	—	{4}	{5}	{6}	{7}	{8}
(3,4)	{1,2,3,4}	—	—	—	{5}	{6}	{7}	{8}
(2,4)	{1,2,3,4}	—	—	—	{5}	{6}	{7}	{8}
(2,5)	{1,2,3,4,5}	—	—	—	—	{6}	{7}	{8}
(5,7)	{1,2,3,4,5,7}	—	—	—	—	{6}	—	{8}
(5,6)	{1,2,3,4,5,6,7}	—	—	—	—	—	—	{8}
(6,8)	{1,2,3,4,5,6,7,8}	—	—	—	—	—	—	—
(7,8)	{1,2,3,4,5,6,7,8}	—	—	—	—	—	—	—

Find the connected components of graph



Solⁿ:-

Edge Processed	Collection of disjoint Set							
Initial Set	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}
(1,2)	{1,2}	—	{3}	{4}	{5}	{6}	{7}	{8}
(3,4)	{1,2}	—	{3,4}	—	{5}	{6}	{7}	{8}
(5,7)	{1,2}	—	{3,4}	—	{5,7}	{6}	—	{8}
(6,8)	{1,2}	—	{3,4}	—	{5,7}	{6,8}	—	—
(1,3)	{1,2,3,4}	—	—	—	{5,7}	{6,8}	—	—
(5,6)	{1,2,3,4}	—	—	—	{5,6,7,8}	—	—	—
(2,4)	{1,2,3,4}	—	—	—	{5,6,7,8}	—	—	—
(7,8)	{1,2,3,4}	—	—	—	{5,6,7,8}	—	—	—

```
Connected-Components(G)
    for each vertex v ∈ G, V
        Make-Set(v)
    for each edge (u,v) ∈ G.E
        if Find-Set(u) ## ≠ Find-Set(v)
            Union(u,v)

Same-Component(u,v)
    if Find-Set(u) == Find-Set(v)
        return True
    return False
```

Linked list representation of Disjoint-Set

$S_1 = \{f, g, d\}$, $S_2 = \{c, h, e, b\}$