$= n\left[1 - \frac{1}{7/8}\right] + n^{\log_3(7/8)}$

$= n$

UNIT-2

Binomial Heap:- It is a collection of binomial tree, each of which satisfies regular heap property.

- In this case, we are considering min. heap property.

Binomial Tree:- It is a building block for binomial heap.

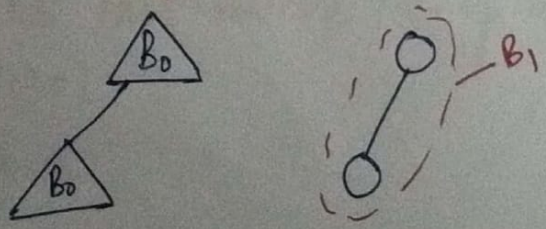
- Binomial tree B_K is an ordered tree defined recursively.

1) B_0 is a single node.

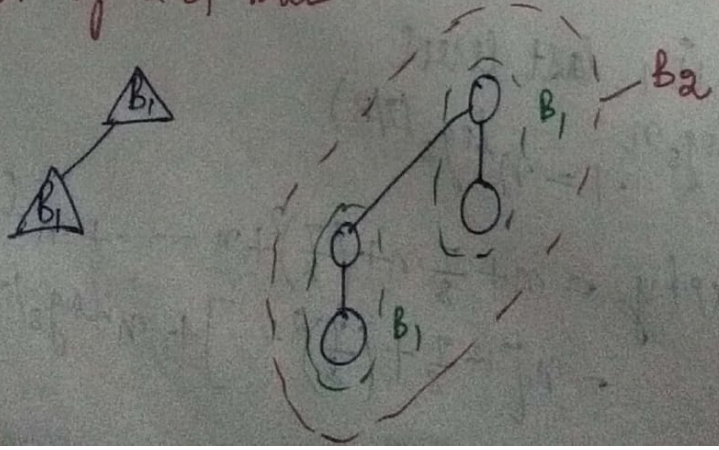
2) Binomial tree B_K consists of 2 binomial trees (B_{K-1}) that are linked together.
Root of one is leftmost child of root of another.

$B_0 = 0$

B_1: collection of 2 B_0 trees



B_2: collection of 2 B_1 trees



Properties of Binomial Tree :-

1) There are 2^k nodes.

2) Height of binomial tree is exactly equal to $\log_2 n$.

3) No. of nodes at depth $i = 0$ to k

$$\begin{bmatrix} k \\ i \end{bmatrix} = \frac{k!}{i!\,(k-i)!}$$

$$\begin{bmatrix} 3 \\ 0 \end{bmatrix} = \frac{\lfloor 3}{\lfloor 0 \,\lfloor 3} = 1 \qquad \begin{bmatrix} 3 \\ 1 \end{bmatrix} = \frac{\lfloor 3}{\lfloor 1 \,\lfloor 2} = 3$$

$$\begin{bmatrix} 3 \\ 2 \end{bmatrix} = \frac{\lfloor 3}{\lfloor 2 \,\lfloor 1} = 3 \qquad \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \frac{\lfloor 3}{\lfloor 3 \,\lfloor 0} = 1$$

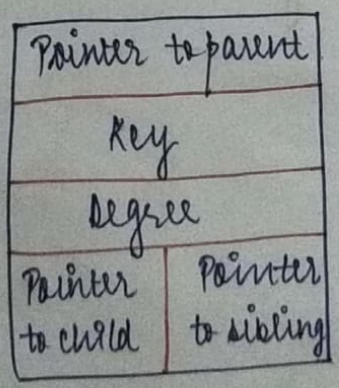
BINOMIAL HEAP :-

Binomial heap (H) is a set of binomial tree that satisfies following properties:-

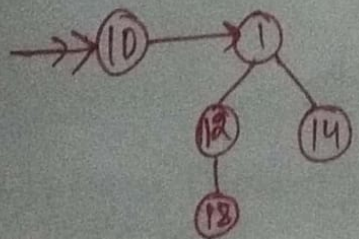
1) Each binomial tree obey min. heap property.

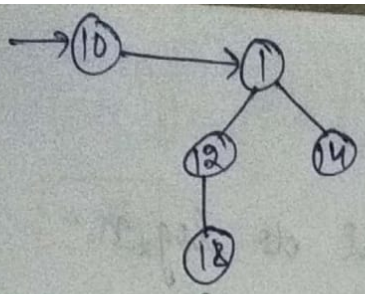
2) There at atmost $\lfloor \log n \rfloor + 1$ binomial trees in binomial heap with n nodes.

REPRESENTATION OF BINOMIAL HEAP :-

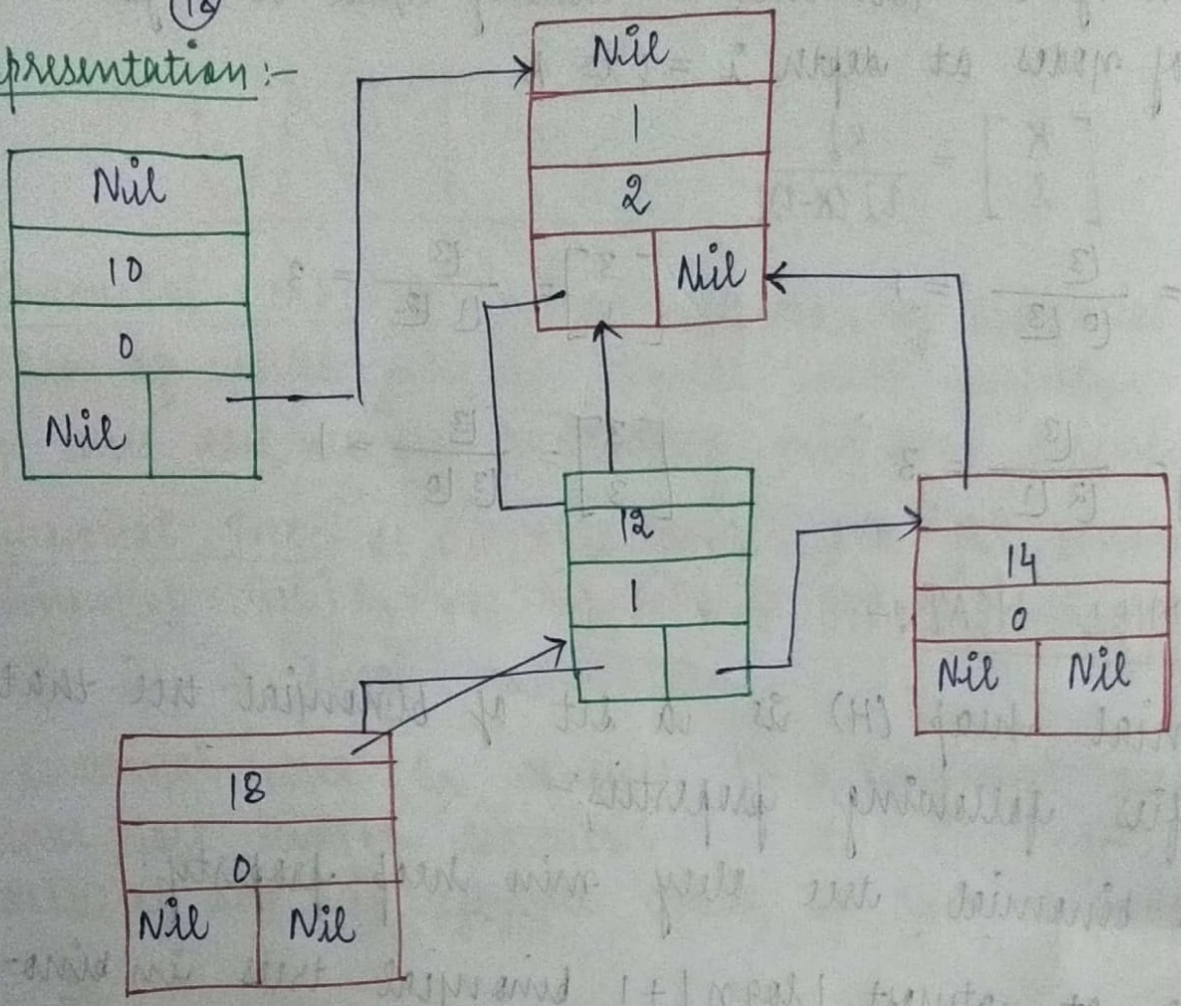


Ex→





Representation :-



OPERATIONS ON BINOMIAL HEAP :-

1) Finding minimum key

Binomial Heap min (H)

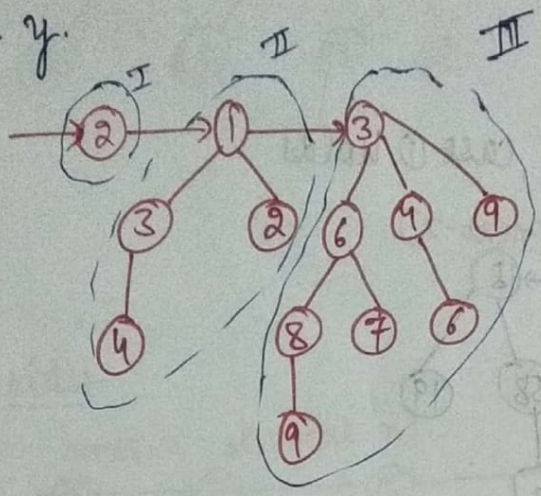
1) y ← nil
2) x ← Head [H]
3) min ← ∞
4) while x ≠ Nil
5) do if key [x] < min
6) then min ← key [x]

7) $y \leftarrow x$

8) $x \leftarrow sibling[x]$

9) return y.

Ex→



2) Union of binomial heap

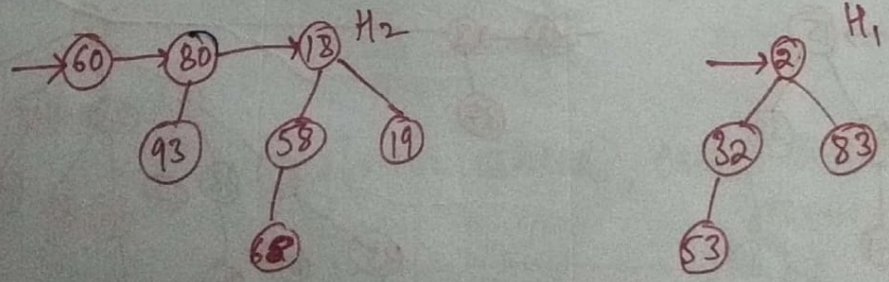
Case 1: If degree [x] ≠ degree [next-x], then move the pointers ahead.

Case 2: If degree of [x] = degree [next-x] = degree [sibling [next-x]] then move the pointers ahead.

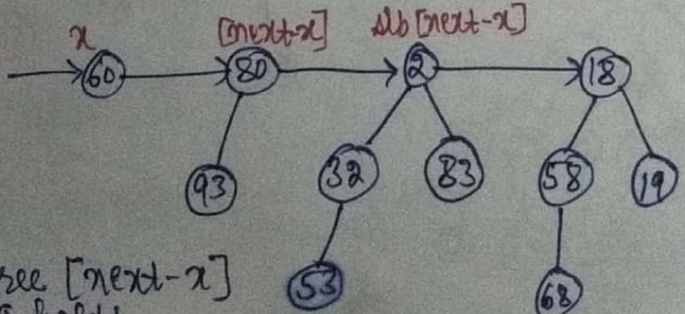
Case 3: If degree [x] = degree [next-x] ≠ degree [sibling [next-x] and key[x] ≤ key [next-x], then remove [next-x] from root list & attach it to x.

Case 4: If degree [x] = degree [next-x] ≠ degree [sibling [next-x] & key[x] > key [next-x], then remove x from root list & attach it to [next-x].

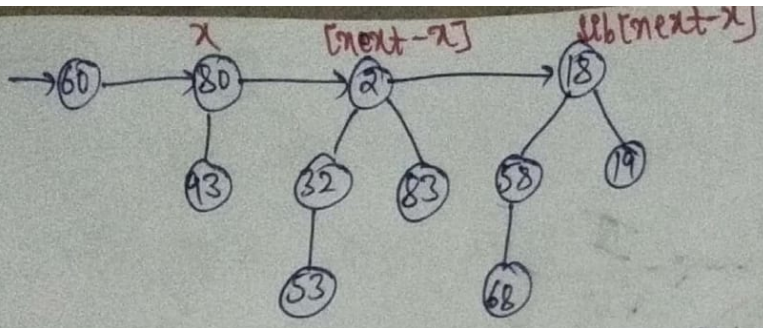
Ex→



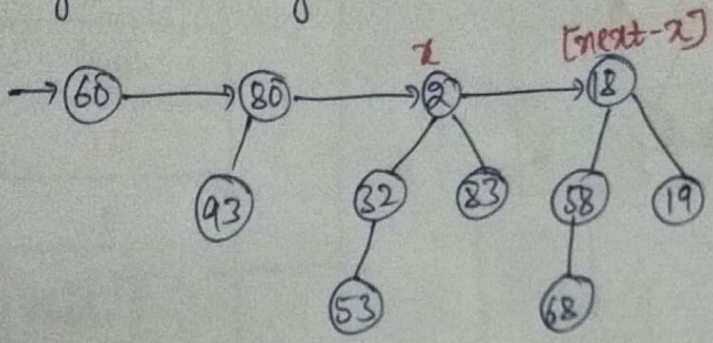
Merging:-



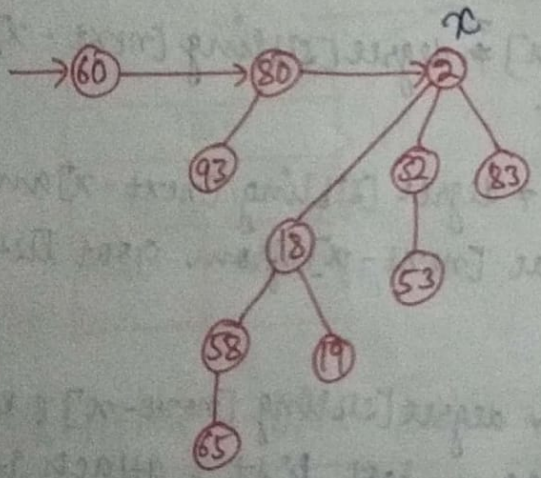
degree[x] ≠ degree [next-x]
Case 1 holds



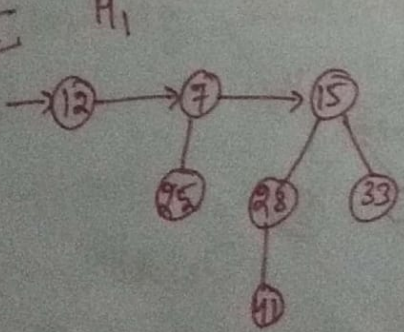
degree[x] ≠ degree[next-x], case ① holds

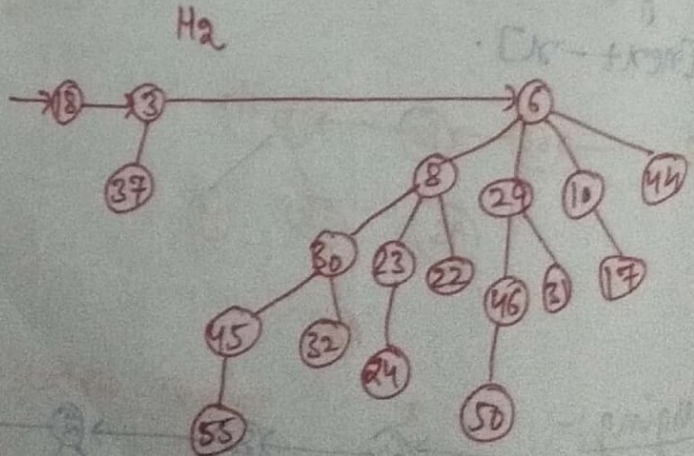


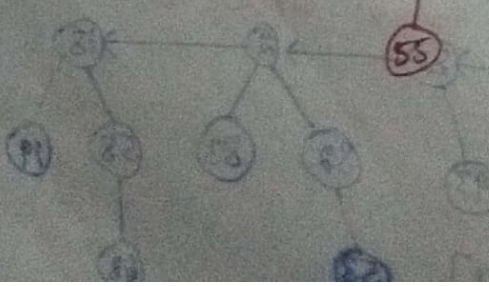
degree[x] = degree[next-x] ≠ degree[sib next-x]
& key[x] ≤ key[next-x], case ③ holds:-



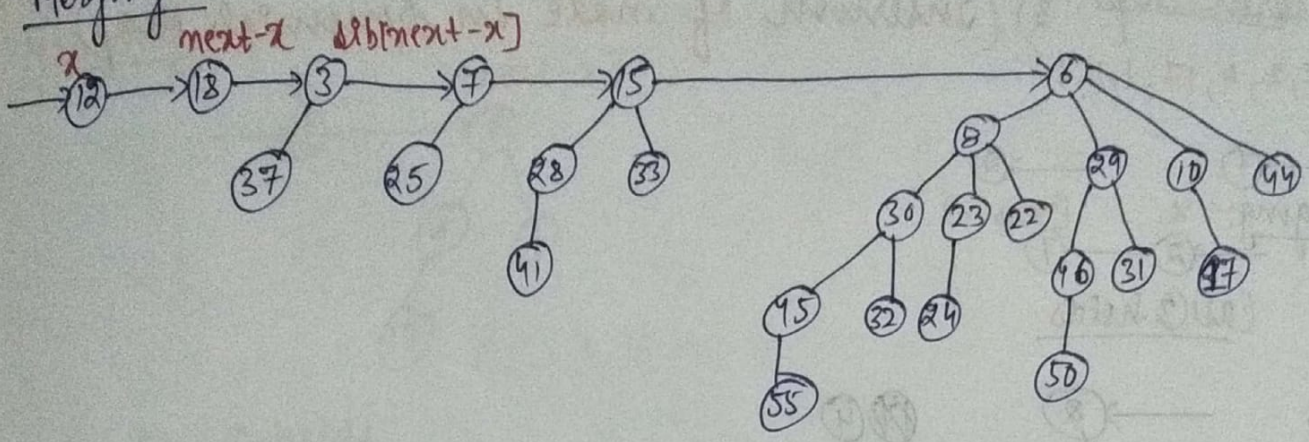
Q:



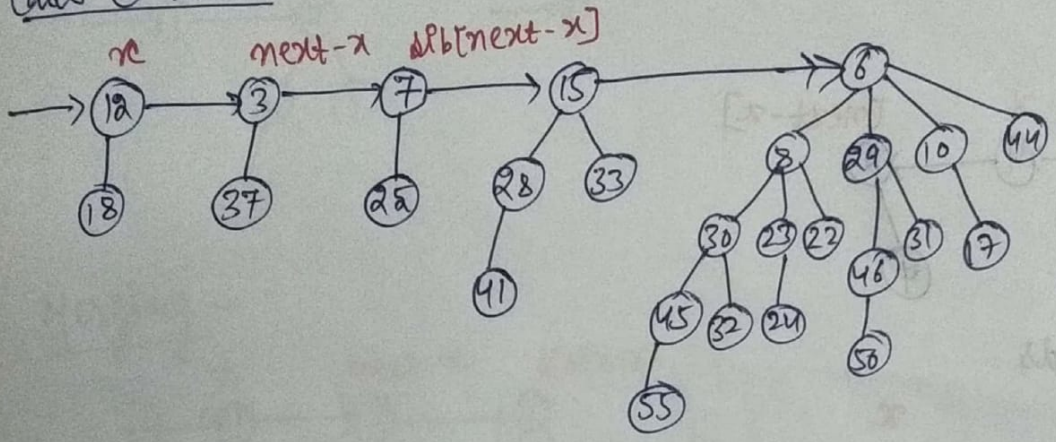




Merging :-

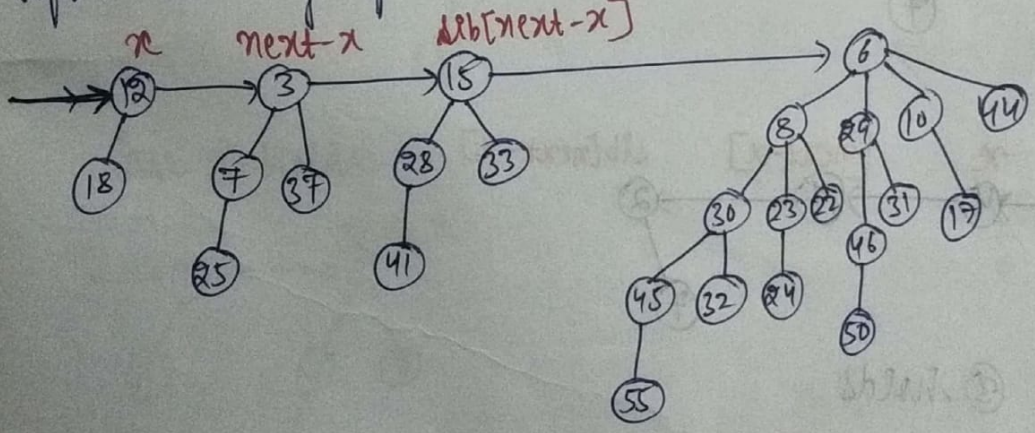


Case ③ holds:-



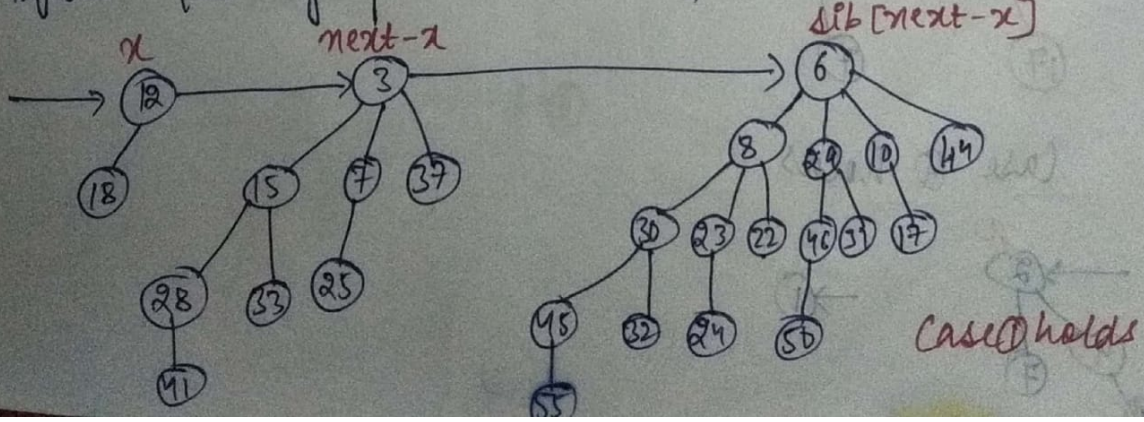
Case 2 holds:-

After moving pointers ahead, case ③ holds



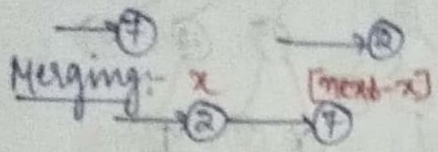
Case ① holds:-

After moving pointer ahead, case 3 holds

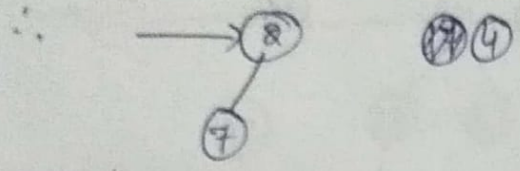


Form Heap 3) (Insertion of node in Binomial Heap)

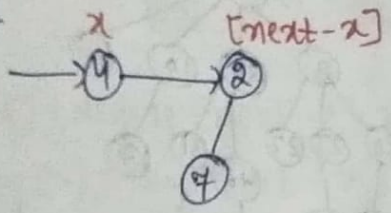
7, 2, 4, 17, 1, 11, 6



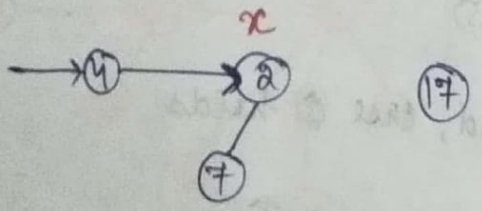
Case (3) holds



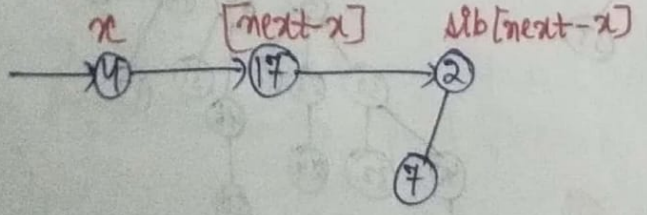
Merging:-



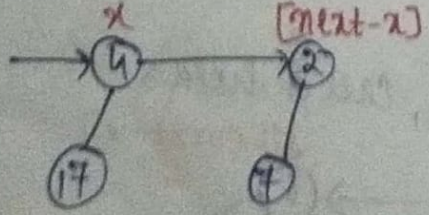
Case (1) holds



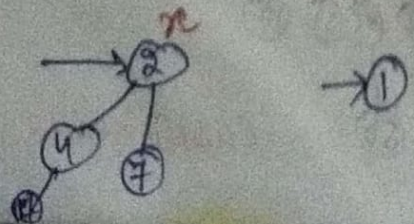
Merging:-



Case (3) holds

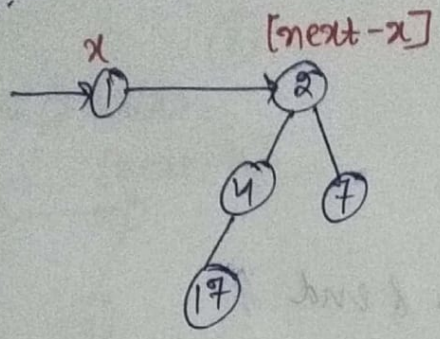


Case (4) holds

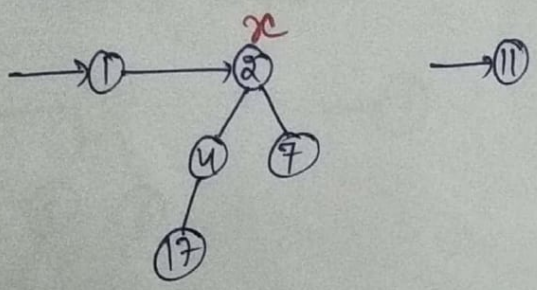


Case ① holds

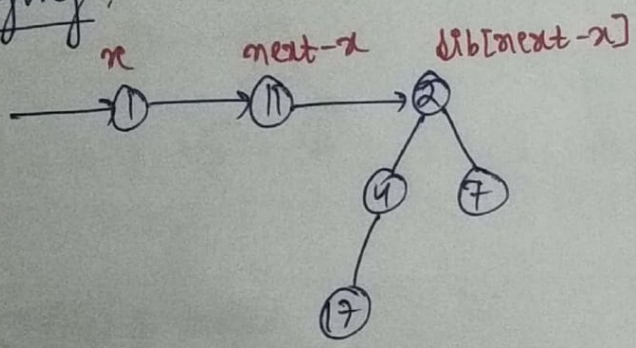
Merging :-



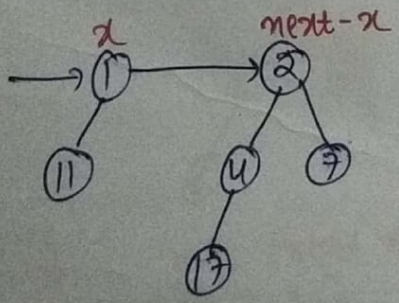
Case ① holds



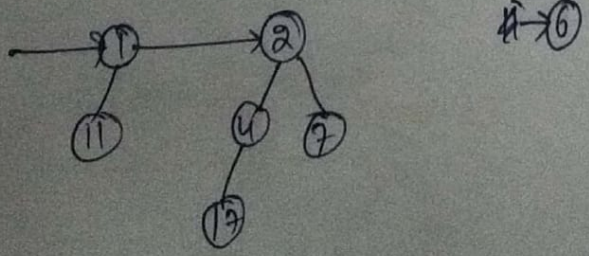
Merging :-



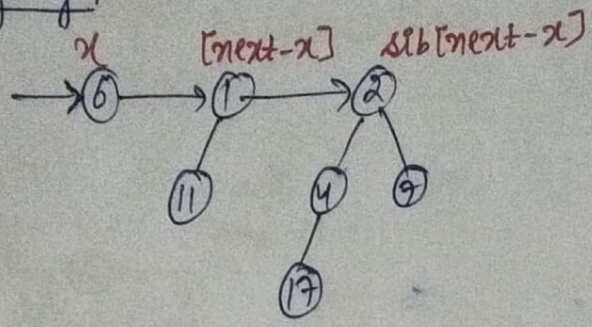
Case ③ holds :-



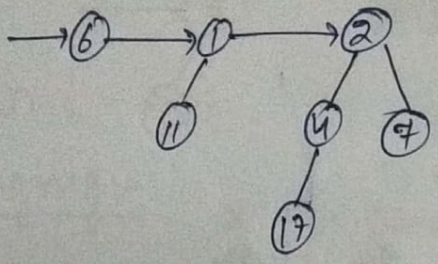
Case ① holds



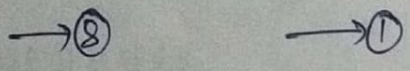
Merging :-



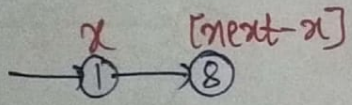
Case ① holds, then case ① holds & end



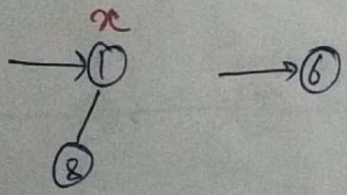
Q:- 8, 1, 6, 5, 7, 2



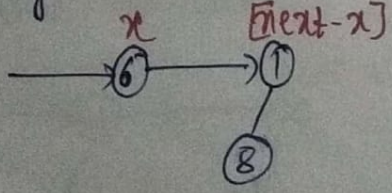
Merging :-



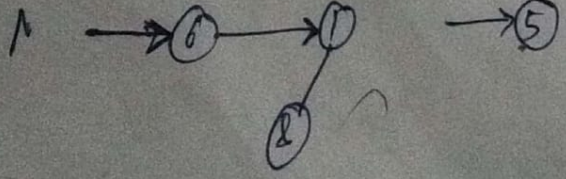
Case ③ holds :-

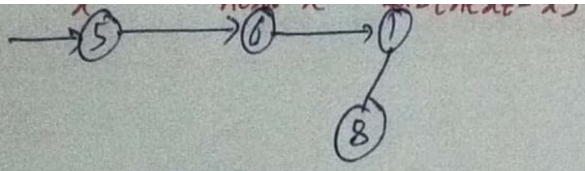


Merging :-

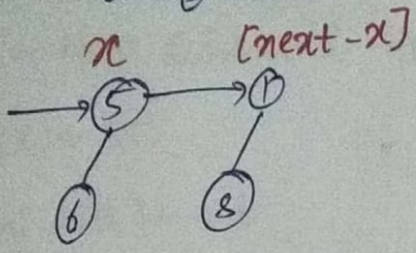


Case ① holds

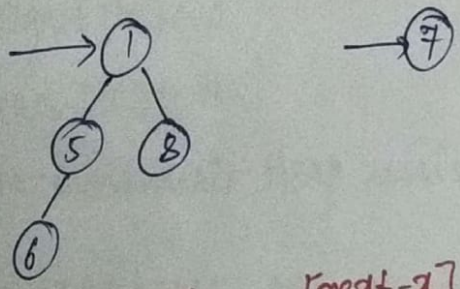




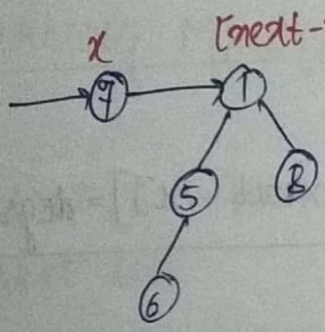
Case ② holds



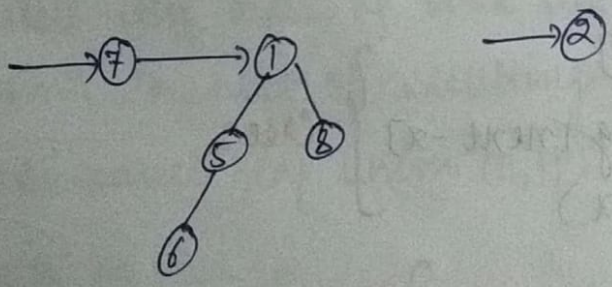
Case ④ holds



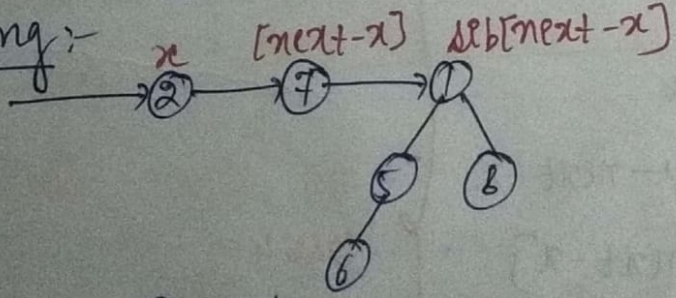
Merging:



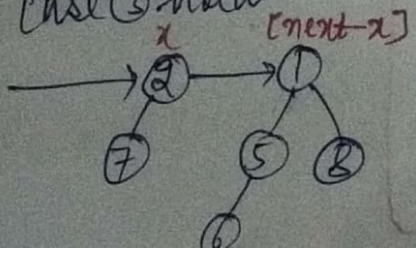
Case ① holds

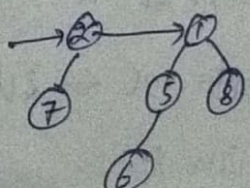


Merging:-



Case ③ holds





Case ① holds (end

UNION OF TWO BINOMIAL HEAP

Algo:-

Binomial Heap Union (H, H')

1) H ← Make-Binomial Heap()
2) Head[H] ← Binomial-Heap Merge (H_1, H_2)
3) if Head[H] = NIL
4) return H;
5) Prev_x ← NIL
6) x ← HEAD[H]
7) next-x ← sibling [x]
8) while next-x ≠ Nil
9) do if (degree [x] ≠ degree [next-x] or

case 1

sibling [next-x] ≠ Nil & degree [sibling [next-x]] = degree [x])

case 2

10) Prev-x ← x
11) x ← next x
12) else if Key[x] ≤ Key [next-x]
13) then sibling [x] ← sibling [next-x]
14) Binomial link (next x, x)

Case 3

15) else if Prev-x ← NIL
16) then Head [x] ← next-x
17) else sibling [Prev-x] ← next-x
18) Binomial link [x, next-x]
19) x ← next-x
20) next-x ← sibling [x]
21) return H.

Case 4

Insertion of node in Binomial Heap

Algo:-

Binomial Heap insert (H, x)

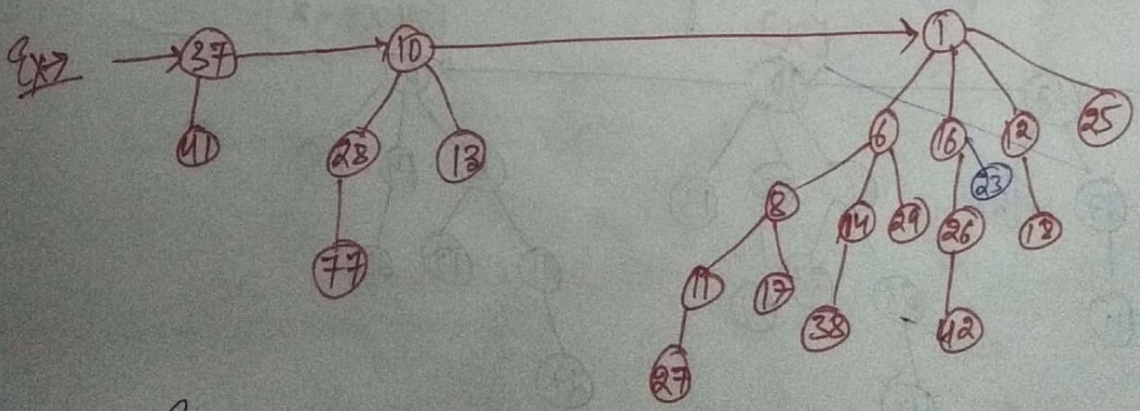
1) H' ← Make Binomial Heap
2) P[x] ← Nil
3) child[x] ← Nil
4) sibling[x] ← Nil
5) Degree[x] ← 0
6) Head[H'] ← x
7) H ← Binomial Heap union (H, H')

4) EXTRACTING MINIMUM KEY NODE :-

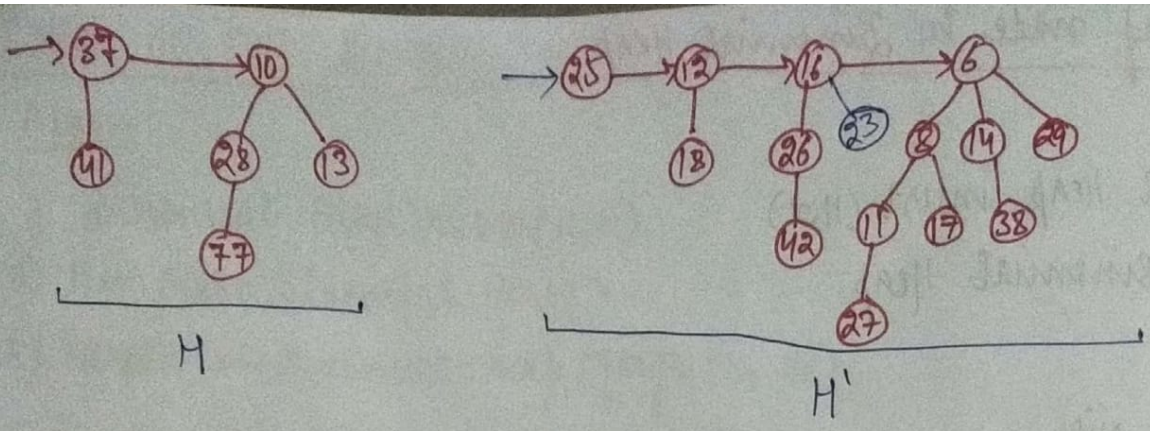
Algo:-

1) Find root x with min. key in root list of H & remove x from root list.
2) H' ← Make Binomial Heap()
3) Reverse the order of link list of x children & set Head(H') to point to head of resulting list.
4) H ← Binomial Heap union (H, H')
5) Return x.

Ex→

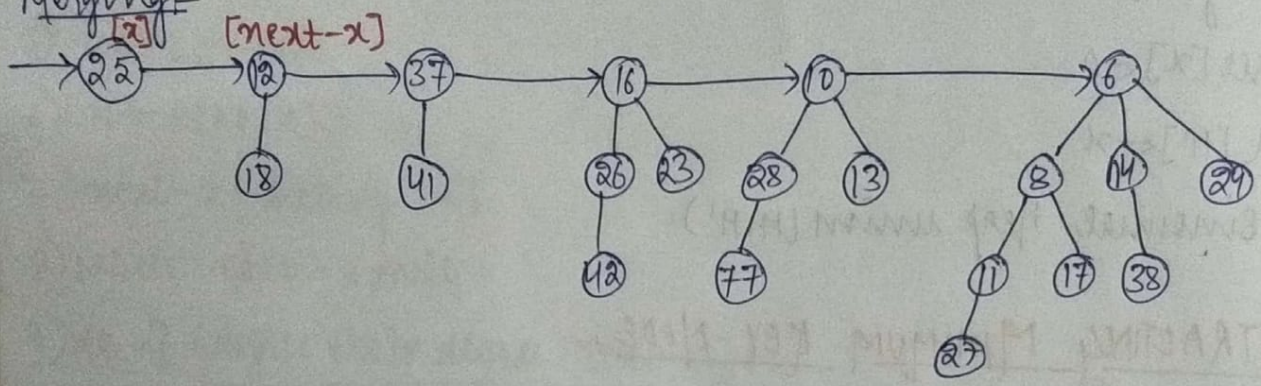


Remove : [Min. root value]



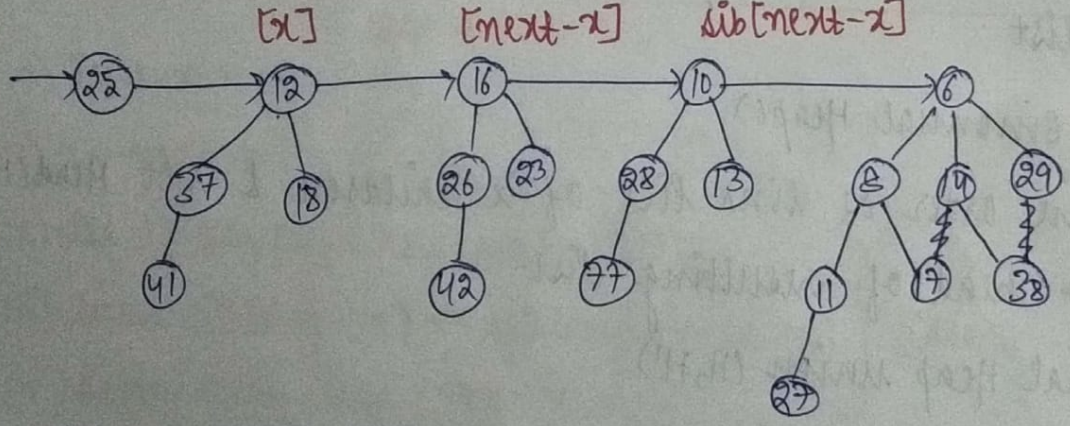
Perform Union:-

Merging:-

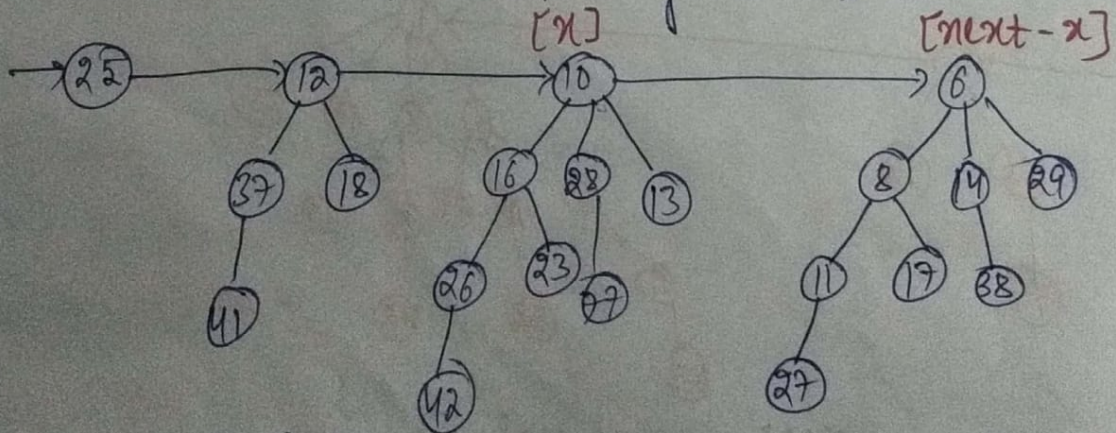


Case ① holds, move header

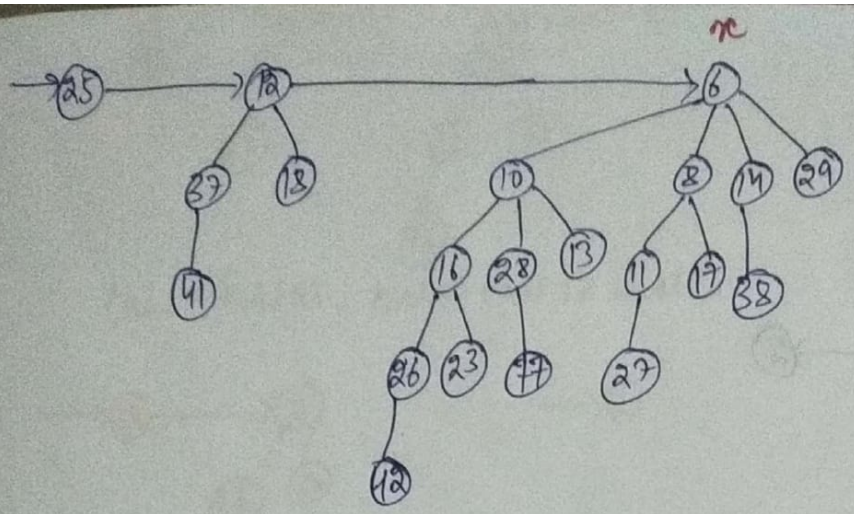
Then case ③ holds



Case 2 holds, after moving ahead, case 3 & 4 holds

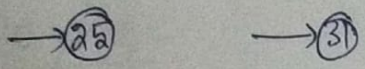


Case ④ holds:-

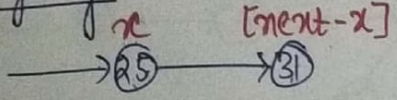


Q:- Construct Binomial Heap

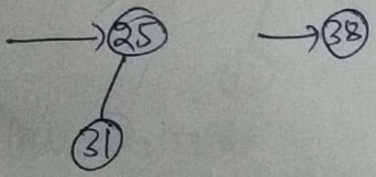
25, 31, 38, 76, 5, 60, 38, 8, 30, 15, 35, 17, 23, 53, 27, 43, 65, 48



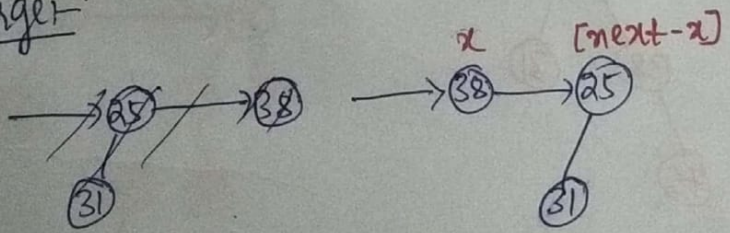
Merging:-



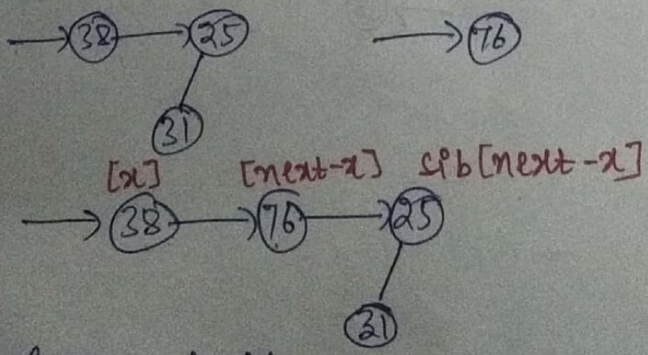
Case 3



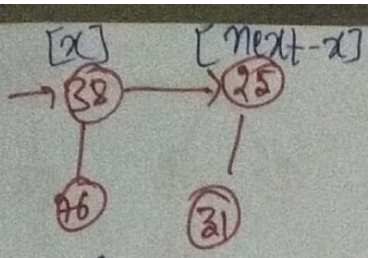
Merge:-



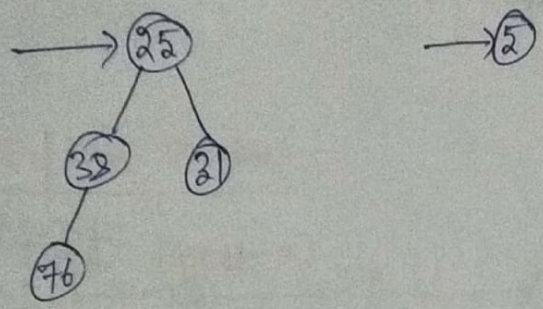
Case 1:-



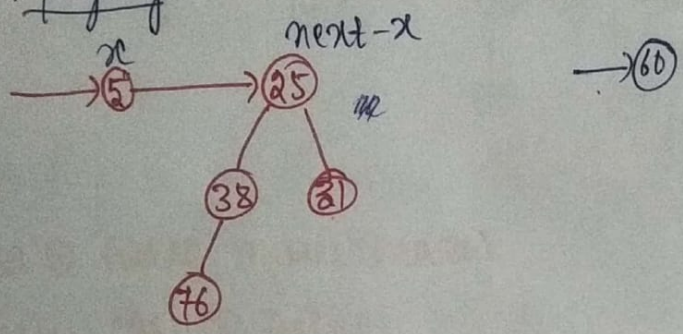
Case 3 holds:-



Case 4:-

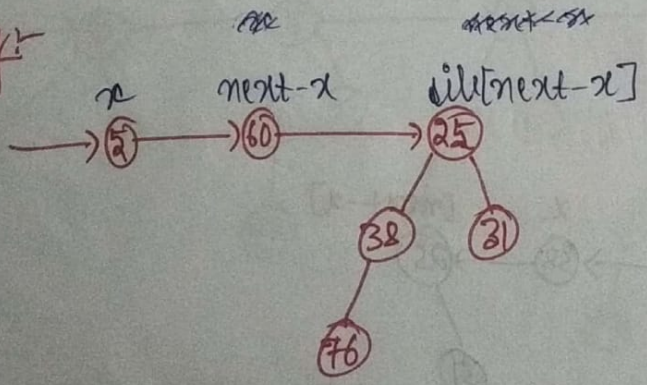


Merging:-

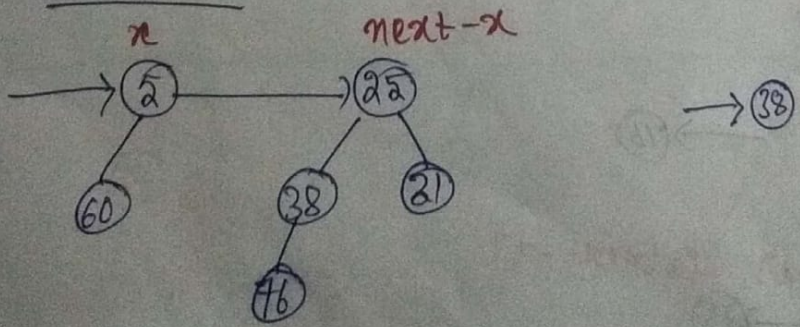


Case 1 holds

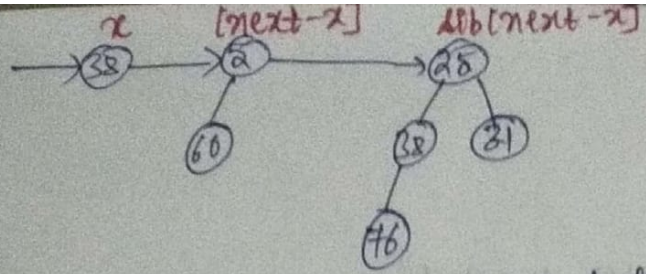
Merging:-



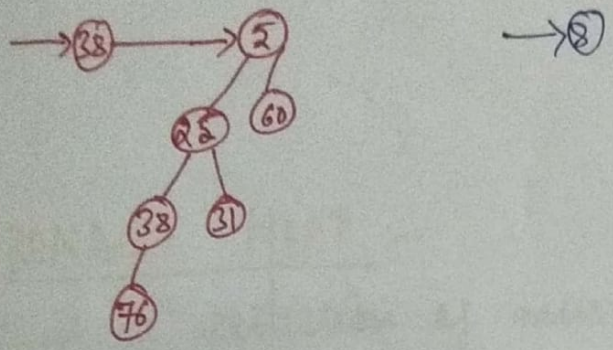
Case 3 holds:-



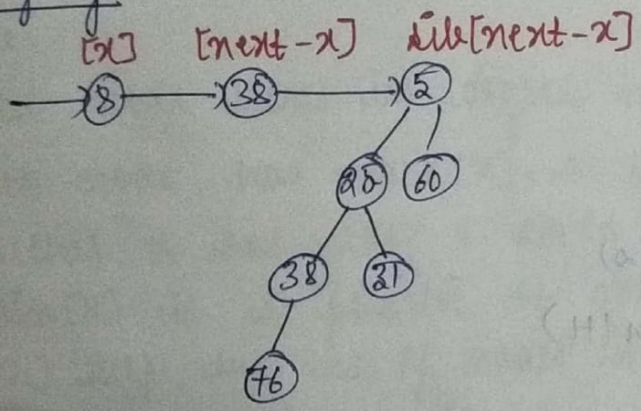
Case 1 holds



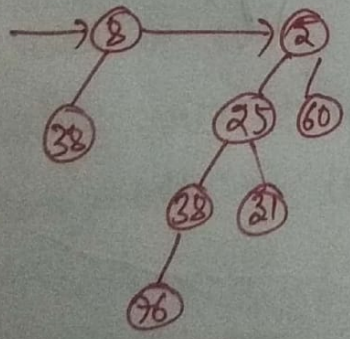
Case 1 holds, then case 2 holds



Merging:-



Case ③ holds



Decreasing a key

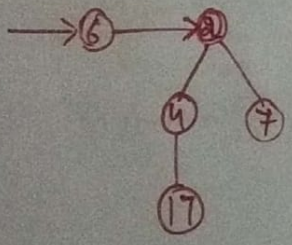
Binomial Heap Decrease (H, x, k)

1) if $k > key[x]$
2) then error "new key is greater than x".
3) $key[x] \leftarrow k$
4) $y \leftarrow x$
5) $z \leftarrow P[y]$
6) while $z \neq nil$ & $key[y] < key[z]$
7) do exchange $key[y] \leftrightarrow key[z]$
8) $y \leftarrow z$
9) $z \leftarrow P[y]$

Deleting a node

Binomial Heap Delete (H, x)

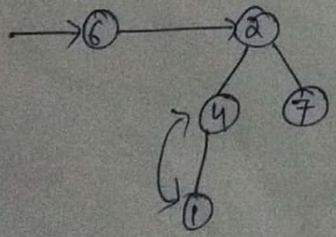
1) Binomial Heap Decrease (H, x, $-\infty$)
2) Binomial extract minimum (H)



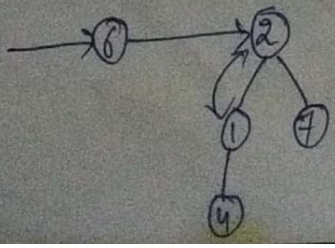
$\Leftarrow$ Ex → (Decreasing a key)

$1 < 17$

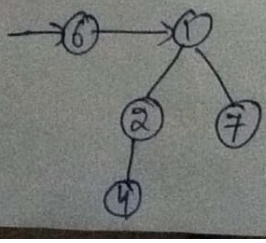
therefore, $1 \leftrightarrow 17$



(Not a min. heap)



$\Rightarrow$



FIBONACCI HEAP :-

→ It is a collection of min. heap ordered tree but tree in fibonacci heap is not constrained to be a binomial tree.

→ Each node stores its degree in degree[x].

→ Each node has mark[x], a boolean field indicating whether x has lost a child.

→ min[H] is a pointer to minimum root in root list

→ n[H] keep the no. of nodes in H.

Mark Nodes :-

1) A node x will be marked if x has lost a child.

2) Newly created nodes are unmarked.

3) When node x becomes the child of another node, then it will be unmarked.

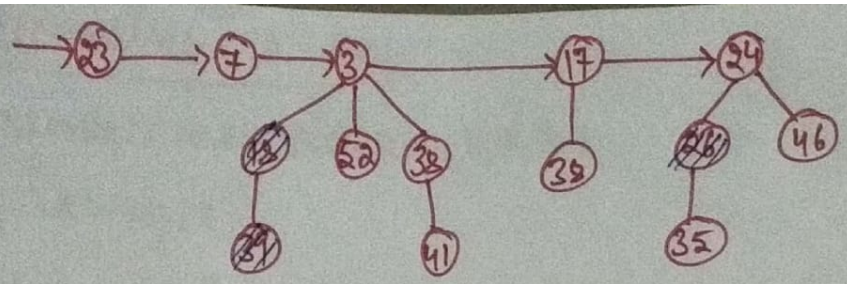
POTENTIAL FUNCTION :-

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

where $m(H)$ = no. of marked nodes

$t(H)$ = no. of nodes in root list.

Ex→



$t(H) = 5$

$m(H) = 3$

$\phi(H) = 5 + 2*3$

$= 5 + 6$

$\boxed{\phi(H) = 11}$

Operations on Fibonacci Heap

1) Creating a new fibonacci heap
2) Inserting a node
3) Finding the minimum node.
4) Union of 2 fibonacci heaps
5) Extracting a min. node → Imp.
6) Decreasing the key.
7) Deleting a node

1) Creating a new fibonacci heap

- Make-fib Heap() procedure allocates m/s & returns fib. heap object.

no. of nodes, $n(H) = 0$

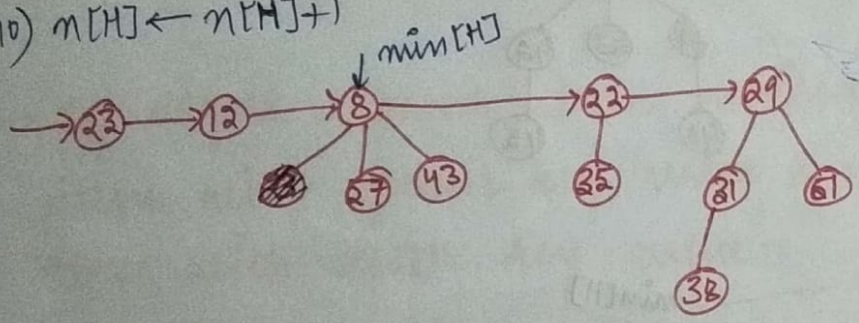
$min[H] = NIL$

$\therefore$ $\boxed{\text{Complexity} = O(1)}$

2) Inserting a node

FibHeap insert(x)

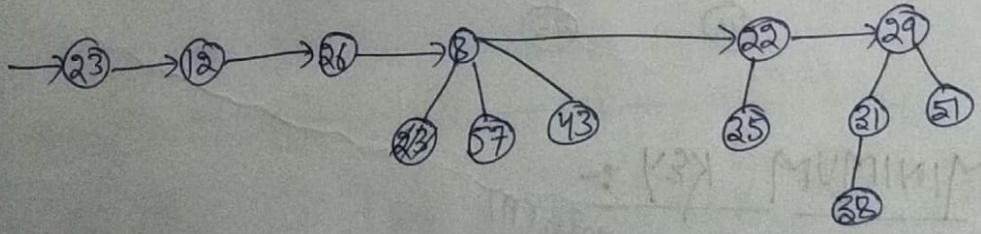
1) degree[x] ← 0
2) P[x] ← nil
3) child[x] ← nil
4) left[x] ← x
5) right[x] ← x
6) mark[x] ← false
7) Concatenate the root list containing x with root list H.
8) if min[H] = Nil or Key[x] < Key[min[H]]
9) then min[H] ← x
10) n[H] ← n[H] + 1



Insert 26:-

(26)

Concatenate it with root list & insert before min[H].



Check inserted element < min[H], if yes mark inserted element as min[H].

3) Finding min. node

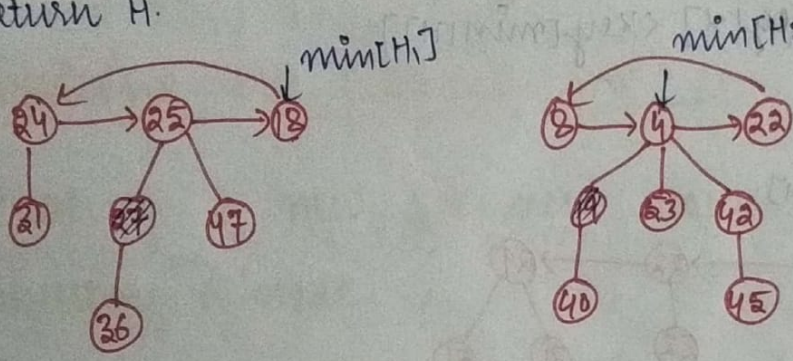
Min. node of the heap is given by pointer min[H].

Complexity = O(1)

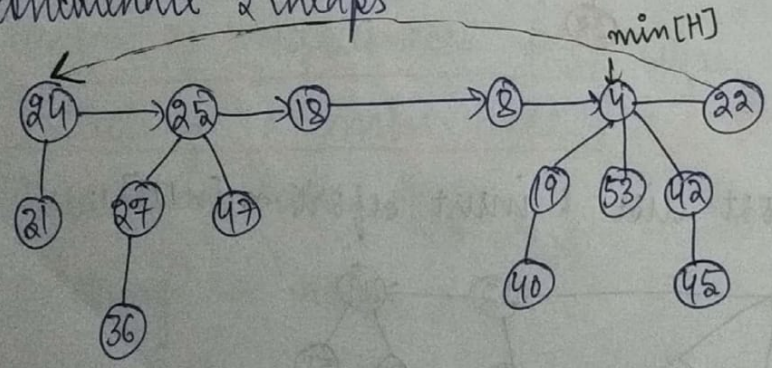
1) Union of 2 fibonacci heaps:

File_Heap_Union (H_1, H_2)

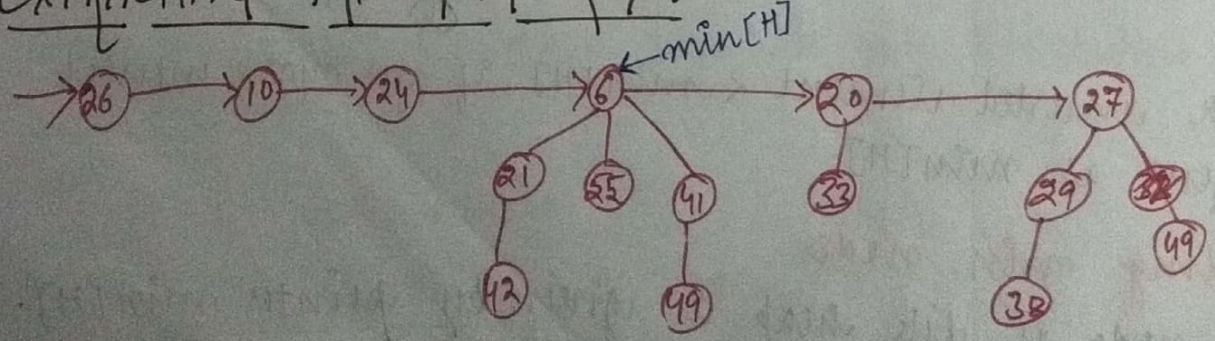
1) H ← Make_file_heap()
2) min (H) ← min [H_1]
3) Concatenate the root list of H_2 with root list H.
4) if (min [H_1] = Nil) or (Min (H_2 ≠ nil & min [H_2] < min [H_1])
5) then min[H] ← min [H_2]
6) n[H] ← n [H_1] + n[H_2]
7) return H.



Concatenate 2 heaps

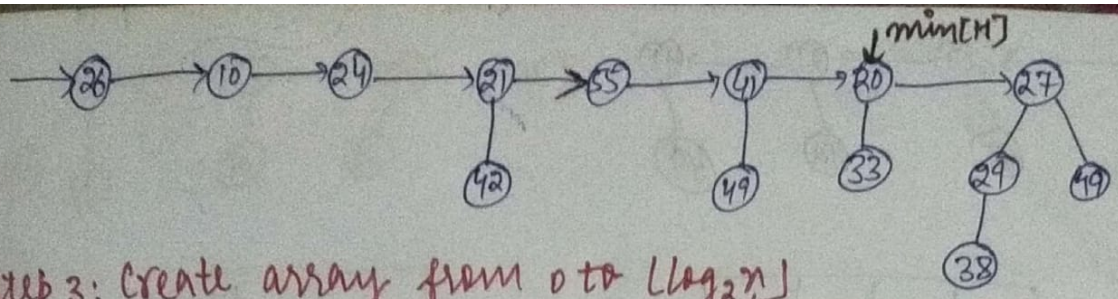


EXTRACTING MINIMUM KEY :-



1) Remove min. node i.e. 6

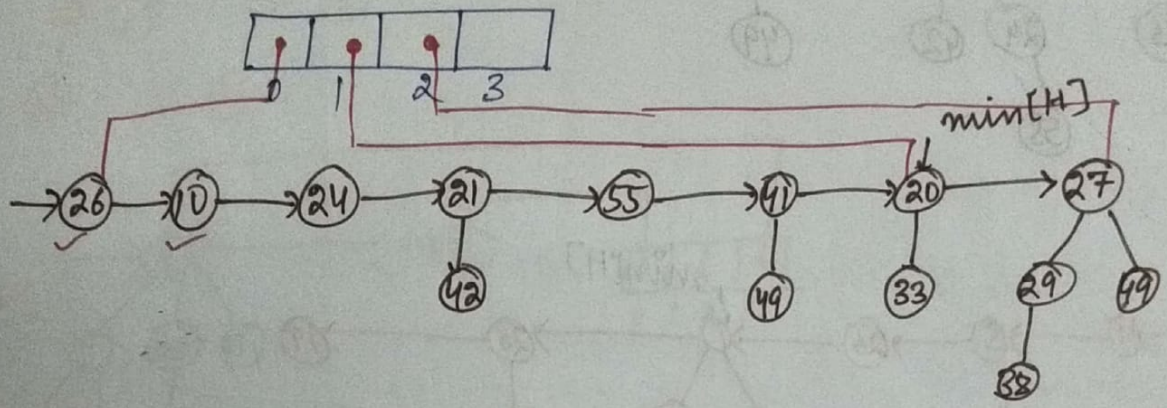
Step 2: (min[H] shifts to right of 6)



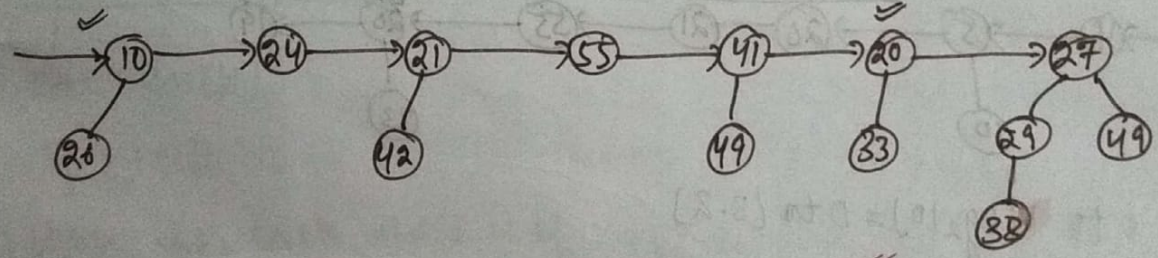
Step 3: Create array from 0 to $\lfloor \log_2 n \rfloor$ → no. of nodes

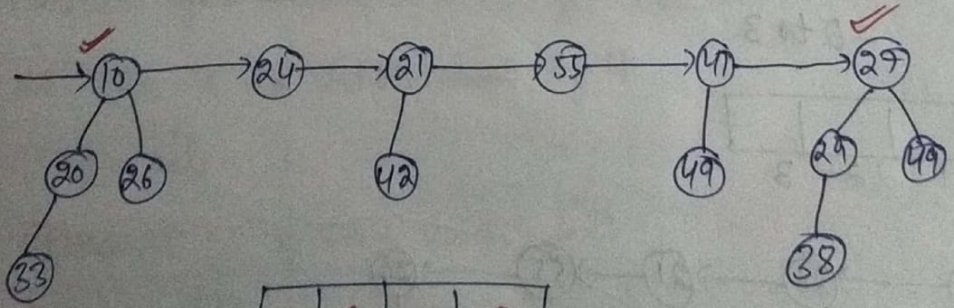
$n = 15$

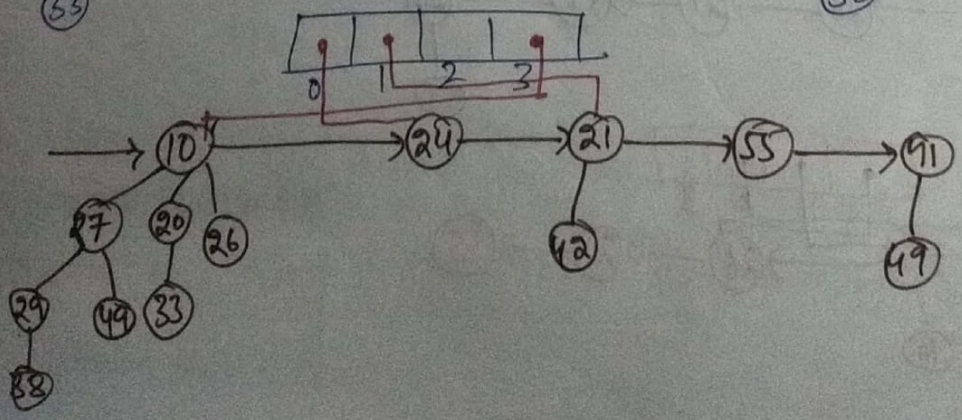
$\lfloor \log_2 n \rfloor = \lfloor \log_2 15 \rfloor = \lfloor 3.8 \rfloor = 3$

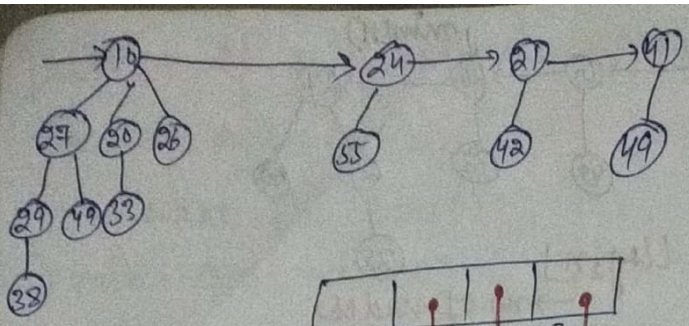


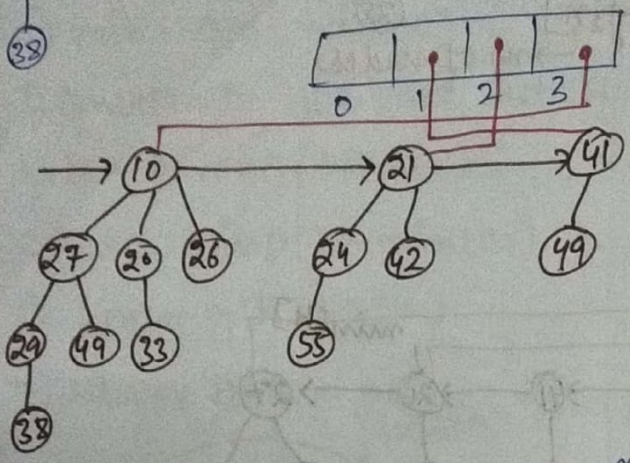
→ If degree is same, then remove from root list the item with greater key value & make it left child of item with lesser key value.

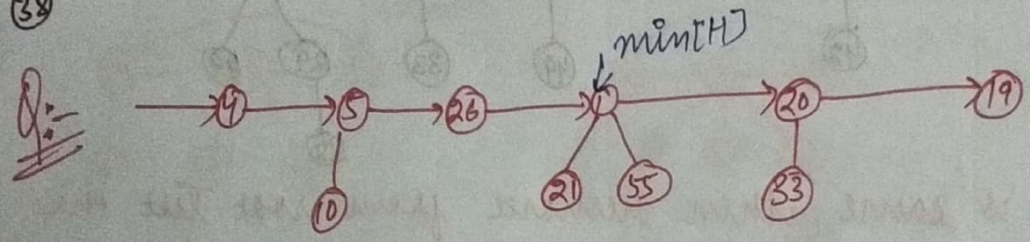




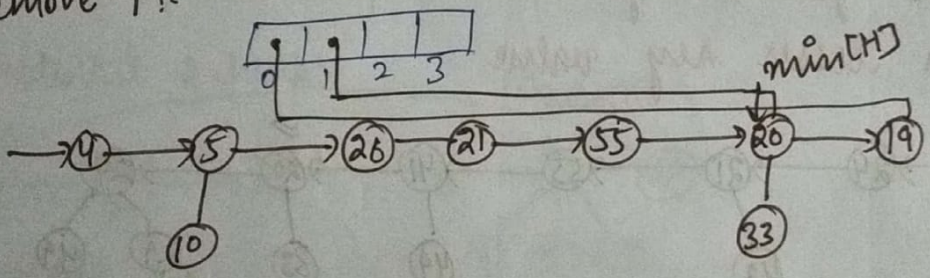






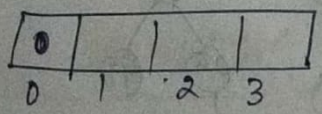


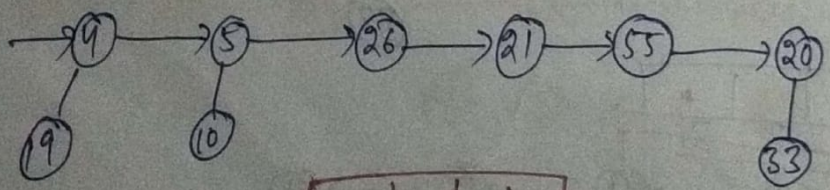
Remove 1 :-

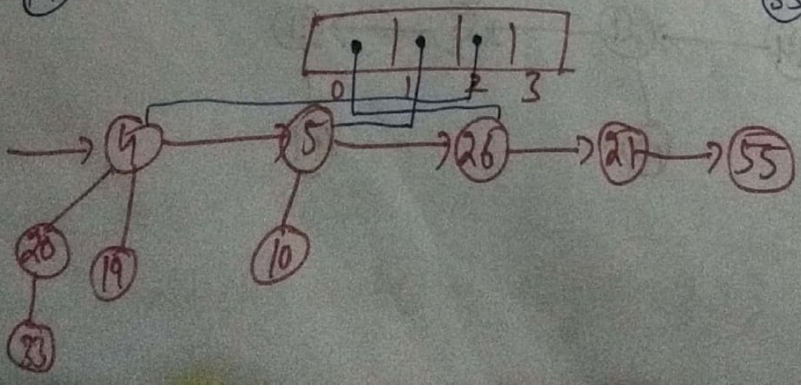


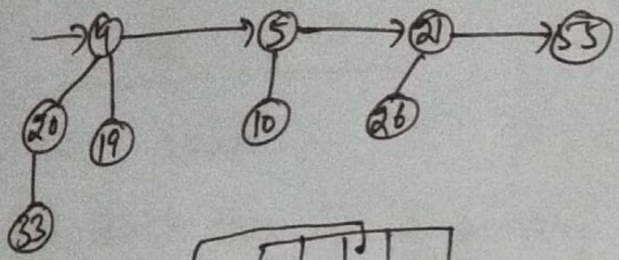
$0 \text{ to } \lfloor (\log_2 10) \rfloor = 0 \text{ to } \lfloor 3.2 \rfloor$

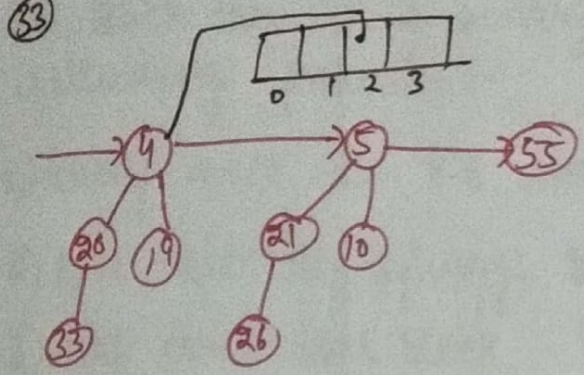
$= 0 \text{ to } 3$

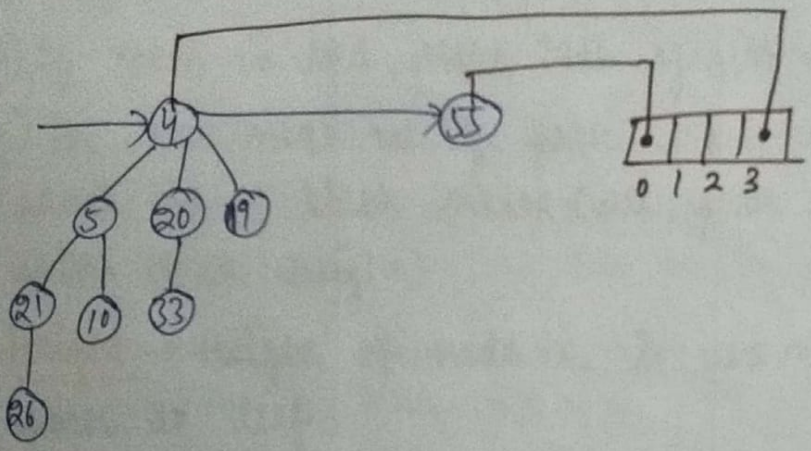












ALGORITHM:-

1) $z \leftarrow min[H]$
2) If $z \neq nil$
3) then for each child x of z
4) do add x to root list of H
5) $P(x) \leftarrow nil$
6) Remove z from root list
7) if $z = right[z]$
8) then $min[H] \leftarrow right[z]$
9) CONSOLIDATE [H]
10) $n[H] \leftarrow n[H] - 1$
11) return x

CONSOLIDATE(H)

1) for i ← 0 to ⌊log₂ n⌋
2) do A[i] ← nil
3) for each node w in root list
4) do x ← w
5) d ← degree[x]
6) while A[d] ≠ nil
7) do y ← A[d]
8) Key[x] ↔ Key[y]
9) then exchange x ↔ y
10) fib-Heap-Link(H, y, x)
11) A[d] ← nil
12) d ← d+1
13) A[d] ← x
14) min[H] ← nil
15) for i ← 0 to ⌊log₂ n⌋
16) do if A[i] ≠ nil
17) then add A[i] to root list H
18) if min[H] = Nil or Key(A[i]) < Key(min[H])
19) then min[H] ← A[i]

fib-Heap-Link(H, y, x)

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

R-B Tree

Red Black

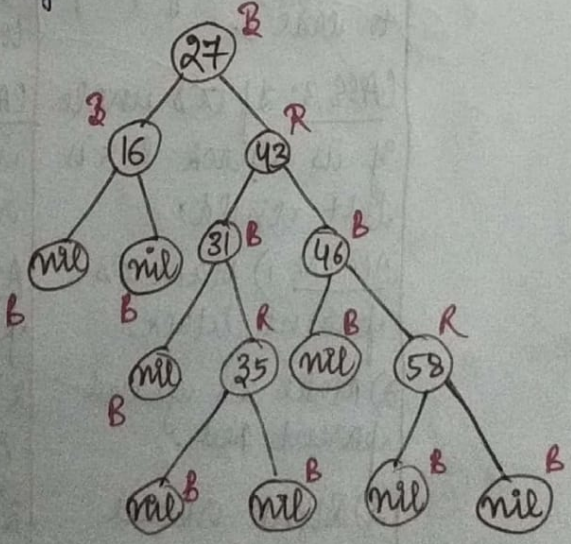
Red-Black tree is a binary search tree which satisfies the following properties :-

1) Every node in R-B tree is either red or black.
2) Root node is always black.
3) Leaf nodes are black.
4) If node is red, then both of its children must be black.
5) For each node, every path from node to leaf contains same no. of black nodes (all path from node to leaf have same black height).

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

Black height of a node → Black height of a node x is no. of black nodes including leaf nodes on the path x to leaf but not counting x.

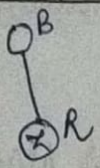
- Black height of a tree is black height of its root.



Black height (BH) bh(x) = 2

INSERTION IN RED-BLACK TREE

New node to be inserted is always red.

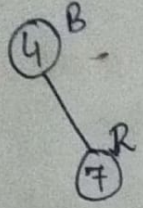
z is root node	z is a root node		
	z's parent is black	z's parent is red	
1) (R) → (B) simply color it black.	 There is nothing to do by adding red to parent black	z's parent is left child of grandparent of z	z's parent is right child of grandparent of z
		Case 1: If z's uncle y is red & z is left or right child, Action: 1) Make z's parent & uncle black. 2) Make z's grandparent red 3) Set z's grandparent as new z.	Case 1: Same
		CASE 2: If z's uncle y is black and z is right child, Action: 1) Set z's parent as new z 2) Left rotate around z. 3) Immediately jump to case 3.	CASE 2: If z's uncle y is black & z is left child, Action: 1) Set z's parent as new z. 2) Right rotate around z. 3) Immediately jump to case 3.
		CASE 3: If z's uncle y is black & z is left child, ACTION: 1) Make z's parent black. 2) Make z's grand-parent red. 3) Right rotate around z's grand-parent.	CASE 3: If z's uncle y is black & z is right child, ACTION: 1) Make z's parent black. 2) Make z's grandparent red. 3) Left rotate around z's grandparent.

Q: Insert 4, 7, 12, 15, 3, 5, 14, 18, 16 into empty R-B tree.

Insert 4

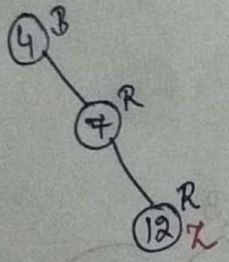
① (4)R → simply color it black as root node cannot be red (4)B

② Insert 7

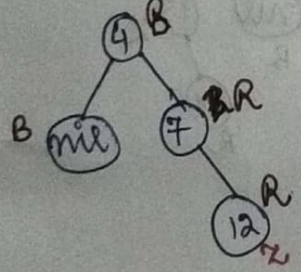


→ As red can be child of black node.

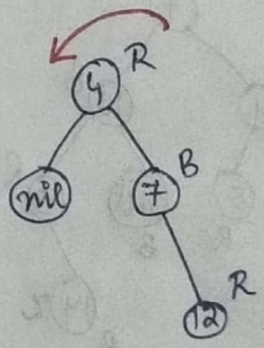
③ Insert 12

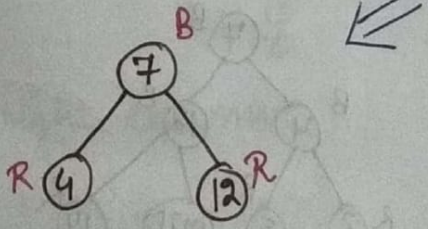


Red cannot be child of red

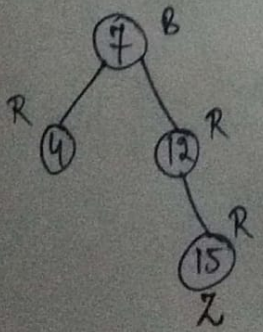


Case ③ of right column holds. ⇒

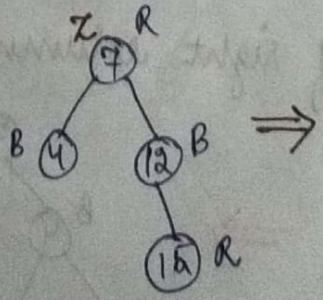




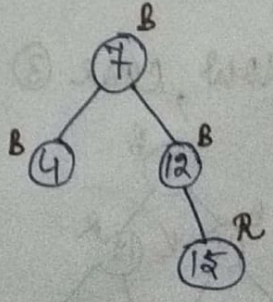
④ Insert 15



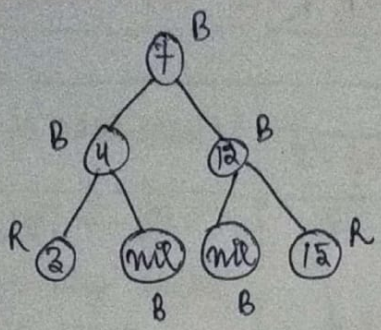
Case ① holds



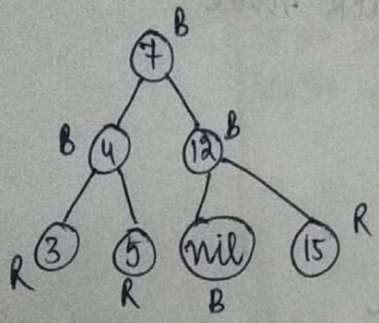
⇒



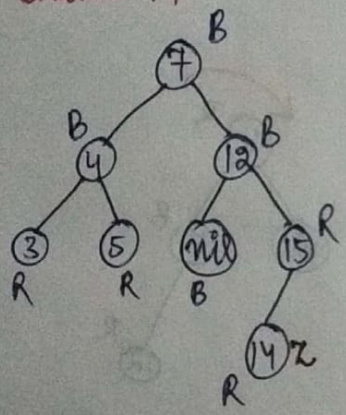
⑤ Insert 3



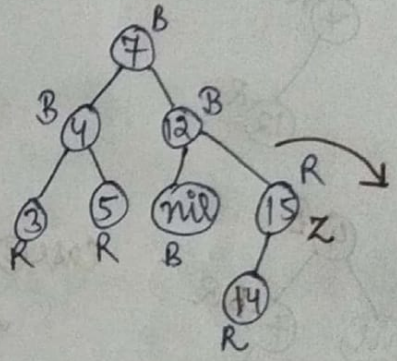
⑥ Insert 5

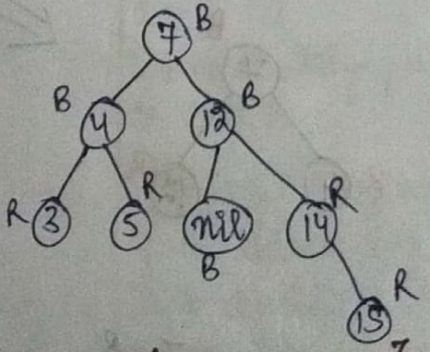


⑦ Insert 14

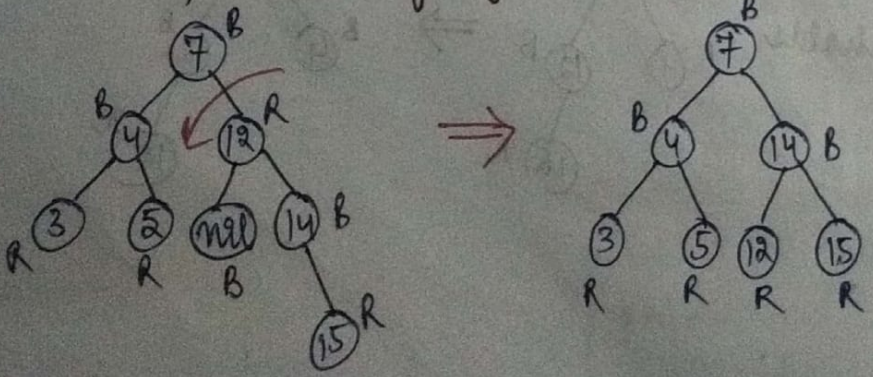


case ② of right column holds :

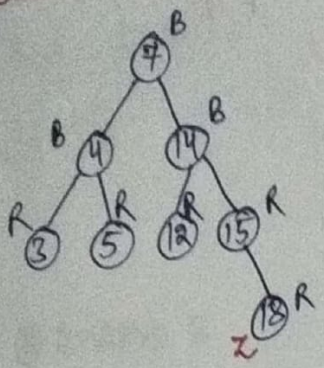




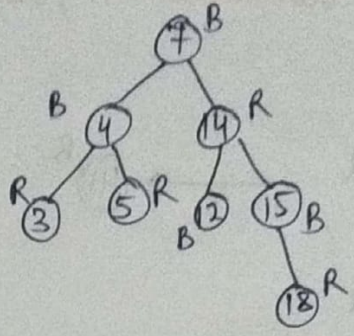
Now, case ③ of right column.



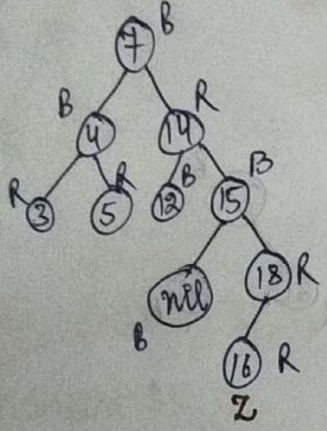
⑧ Insert 18



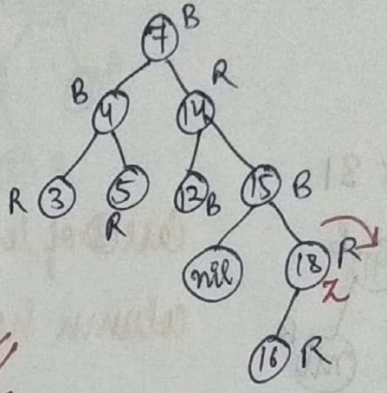
case ① holds

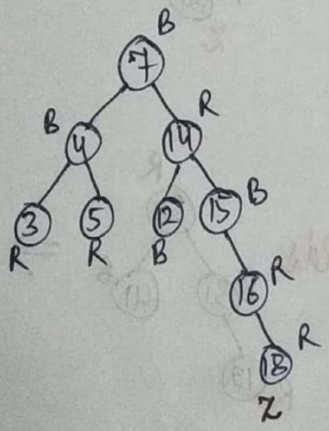


⑨ Insert 16

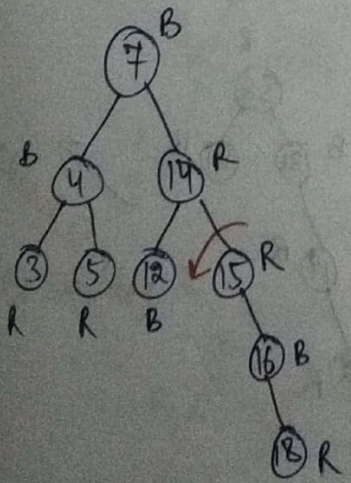


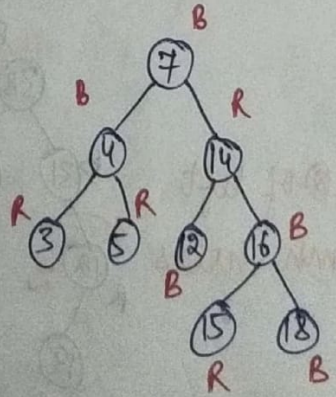
case ② of right column holds





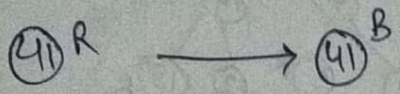
Now, case ③ of right column



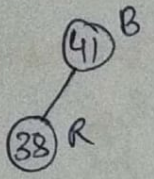


Q: Insert elements 41, 38, 31, 12, 19, 8 into empty R-B tree.

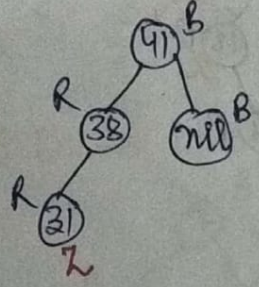
① Insert 41



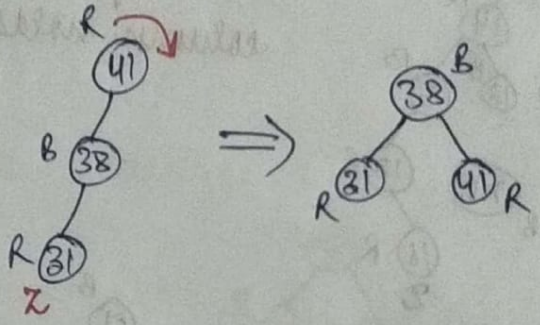
② Insert 38



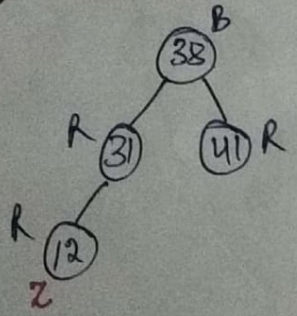
③ Insert 31



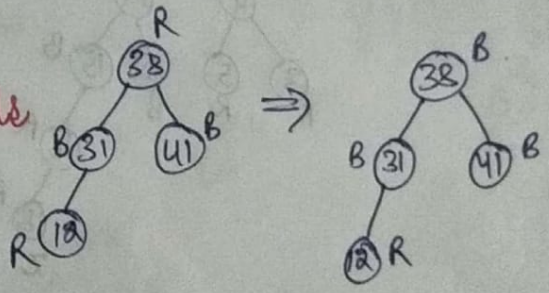
Case ③ of left column holds



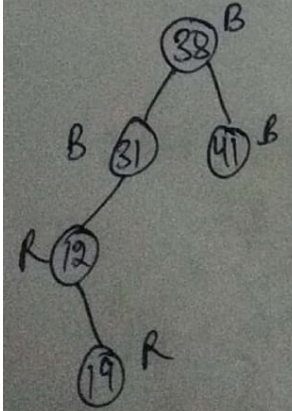
④ Insert 12



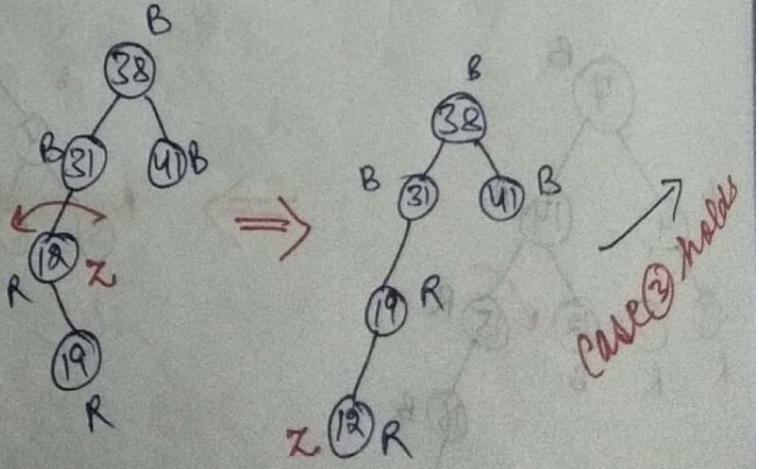
Case ① holds

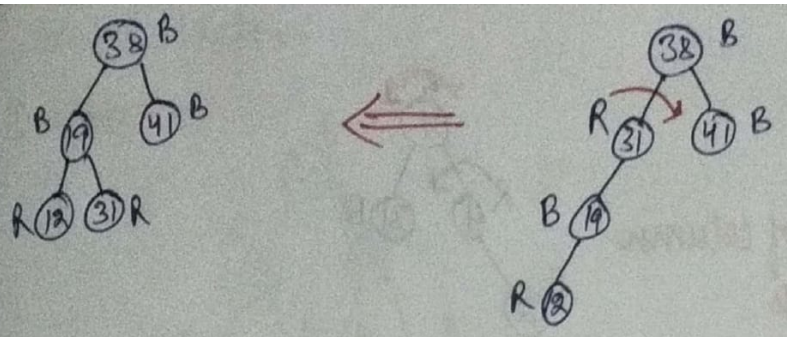


⑤ Insert 19

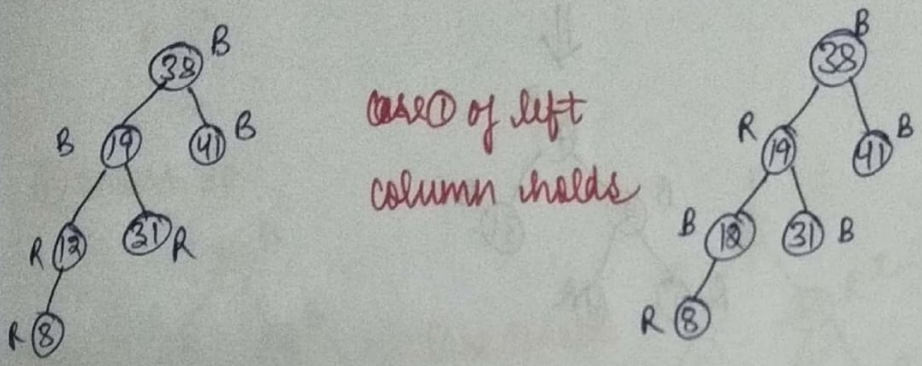


Case ② of left column holds



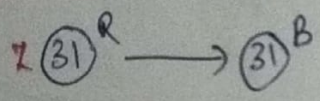


⑥ Insert 8

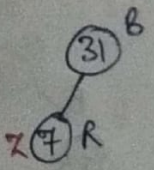


Q:- Insert 31, 7, 4, 2, 1, 19, 8

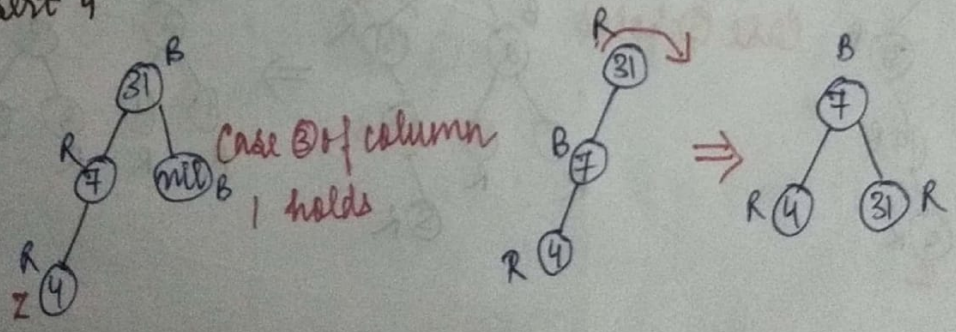
① Insert 31



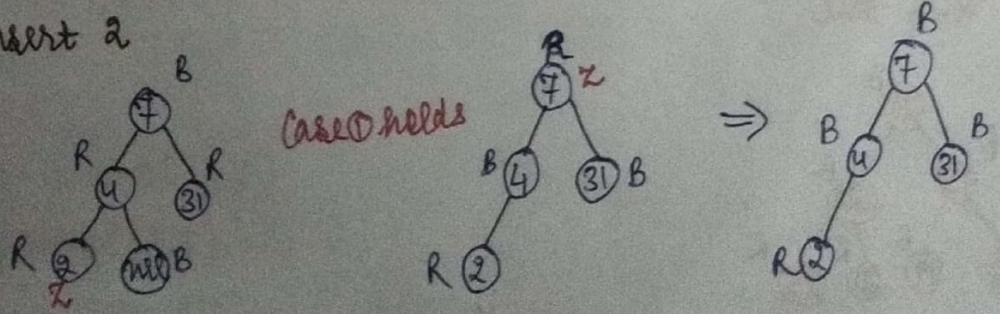
② Insert 7



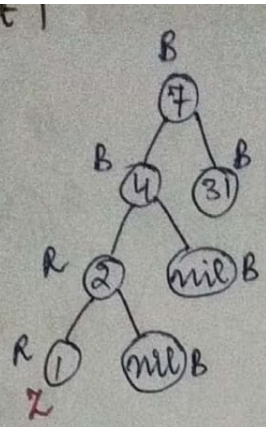
③ Insert 4



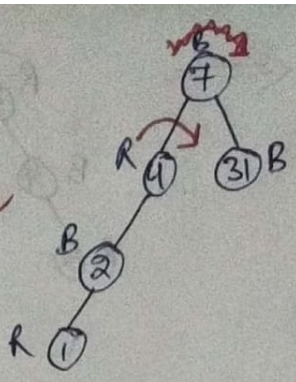
④ Insert 2

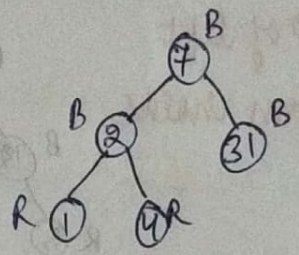


⑤ Insert 1

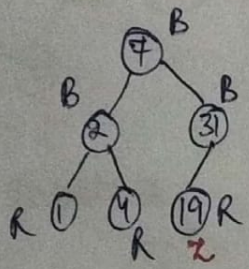


Case ② of column 1 holds

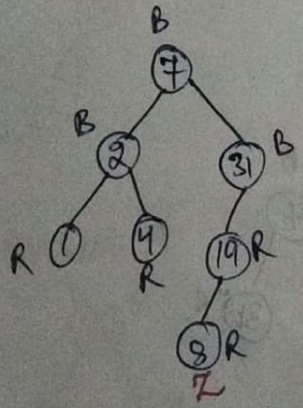




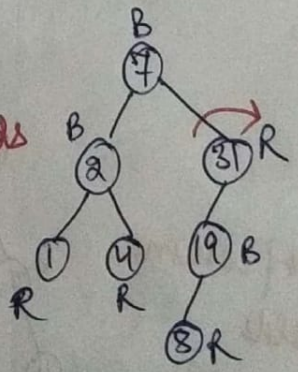
⑥ Insert 19

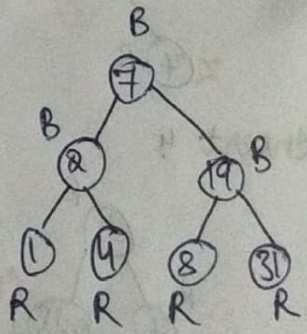


⑦ Insert 8

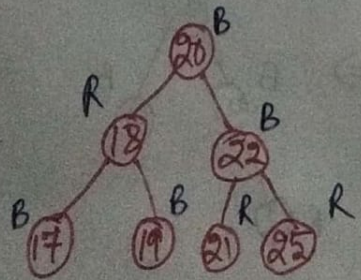


Case ③ holds



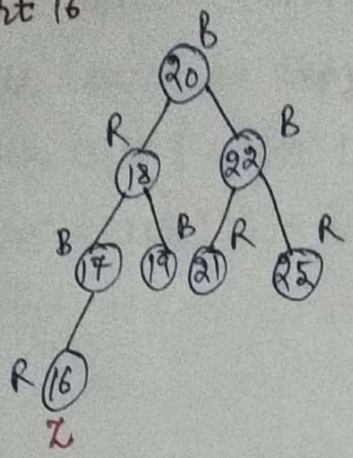


Q:- Given R-B tree

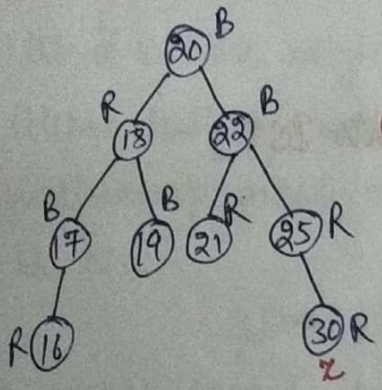


Insert 16, 30, 6

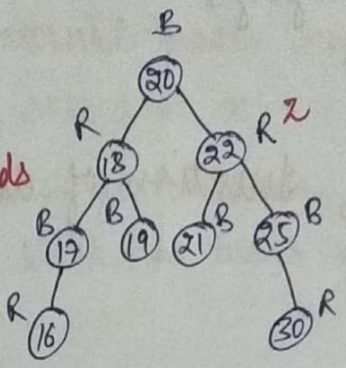
① Insert 16



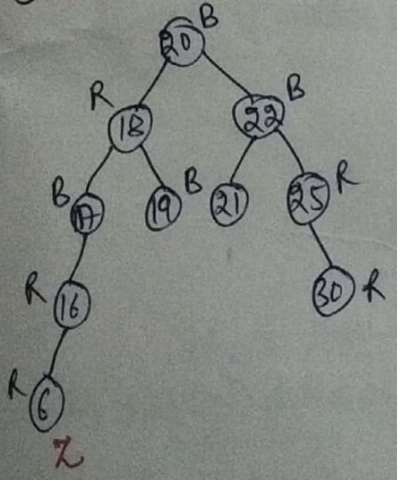
② Insert 30



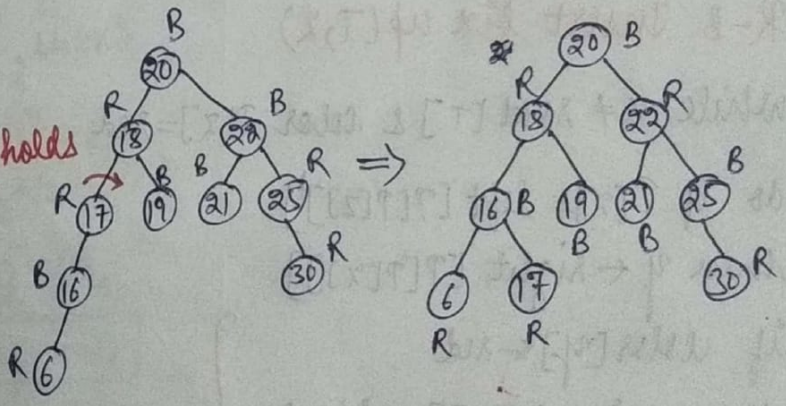
Case ① holds



③ Insert 6



Case ③ holds



INSERTION INTO R-B TREE

Algo :-

RB-Insert(T, z)

① y ← nil[T]

② x ← root[T]

③ while x ≠ nil[T]

④ do $y \leftarrow x$
⑤ if key[z] < key[x]
⑥ then $x \leftarrow$ left[x]
⑦ else $x \leftarrow$ right[z]

} Traversal in BST

⑧ $P[z] \leftarrow y$
⑨ if $y =$ nil[T]
⑩ then root[T] $\leftarrow z$
⑪ else if key[z] < key[y]
⑫ then left[y] $\leftarrow z$
⑬ else right[y] $\leftarrow z$
⑭ left[z] $\leftarrow$ nil
⑮ right[z] $\leftarrow$ nil

} Insertion of element into BST

⑯ color[z] $\leftarrow$ red
⑰ RB Insert fix up (T, z)

R-B Insert fix up (T, z)

① while $z \neq$ root[T] & color P[z] = red
② do if P[z] = left[P[P[z]]]
③ then $y \leftarrow$ right[P[P[z]]]
④ if color[y] $\leftarrow$ red
⑤ then color[P[z]] $\leftarrow$ black
⑥ color[y] $\leftarrow$ red
⑦ color[P[P[z]]] $\leftarrow$ red

} case ①

⑧ $z \leftarrow P[P[z]]$
⑨ else if z = right[z]
⑩ then $z \leftarrow P[z]$
⑪ left rotate (T, z)

} case ②

⑫ color[z] $\leftarrow$ black

(13) color P[P[x]] ← red
(14) right rotate (T, P[P[x]])
(15) else (same as then clause)
(16) with left & right exchange
(17) colour (root [T]) ← black

} Case (3)

Q. Prove that R-B tree with n internal nodes has height $h \geq 2\log(n+1)$

- let h_x be the height of tree rooted at x.
- Atleast half of nodes on any path from root to leaf not including root must be black, so black height of root must be $h/2$.

$$bh(x) \geq h/2$$

- A subtree rooted at any node x contains atleast $2^{bh(x)} - 1$ internal nodes.

So, no. of nodes $(n) > 2^{bh(x)} - 1$

$$n > 2^{h/2} - 1$$

$$n+1 > 2^{h/2}$$

$$\log(n+1) > h/2$$

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

As, $bh(x) \leq h \leq 2bh(x)$

DELETION OF A NODE FROM R-B tree :-

- let x be a node which takes the position of deleted node.

x is root node	x is not root node, x' color is red	x' color is black	
		x is left child of its parent	x is right child of its parent.

simply color it black.

simply color it black.

Case1: x sibling w is red.

Action: 1) Make w black
2) Make x parent red
3) left rotate around x parent.
4) Set right child of x parent as new w.

Case2: x sibling w is black & both of w's children are black.

Action: Make w red & set x parent as new x.

Case3: x sibling w is black & w's left child is red & right child is black.

Action: 1) Make w's left child black.
2) Make w red
3) Right rotate around w.
4) Set right side of x parent as new w.

Case4: x sibling w is black & w's right child is red.

Action: 1) Set w's color as x parent color.
2) Make x parent black.
3) Make w's right child black.
4) left rotate around x parent

Case1: x sibling w is red

Action: 1) Make w black.
2) Make x parent red.
3) Right rotate around x parent.
4) Set left child of x parent as w.

Same

Case3: x sibling w is black & w's left child is black & right child is red.

Action: 1) Make w's right child black.
2) Make w red
3) left rotate around w
4) Set left child of x parent as new w.

Case4: x sibling w is black & w's left child is red.

Action: 1) Set w's color as x's parent color.
2) Make x parent black.
3) Make w's left child black.
4) Right rotate around x parent.

ALGORITHM :-

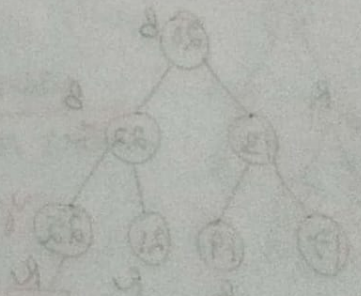
RB-Delete (T, z)

1) if left [z] = nil[T] or right [z] = nil[T]
2) then y ← z
3) else y ← Tree-Successor (z) • inorder
4) if left[y] ≠ nil[T]
5) then x ← left [y]
6) else x ← right [y]
7) P[x] ← P[y]
8) if P[y] = nil[T]
9) then root[T] = x
10) else if y = left [P[y]]
11) then left [P[y]] ← x
12) else right [P[y]] ← x
13) if y ≠ z
14) then Key [z] ← Key[y]
15) if color.[y] = black
16) then RB-Delete - fix- up (T, z)
17) return y.

RB Delete fixup (T, x)

1) while x ≠ root [T] & color [x] = black
2) do if x = left [P[x]]
3) then w ← right [P[x]]
4) if color.[w] = red
5) then color [w] = black
6) color [P[x]] ← red
7) left-rotate (T, P[x])
8) w ← right [P[x]]

Case-1



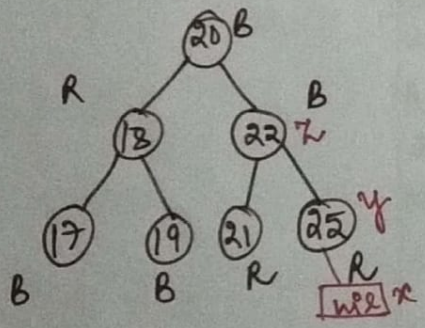
9) if color [left [w] = black & color [right [w] = black
10) then color [w] ← red
11) x ← P[x]
12) else if color [right [w]] = black
13) then color [left [w]] ← black
14) color [w] ← red
15) right-rotate (T, w)
16) w ← right [P[x]]
17) color [w] ← color [P[x]]
18) color [P[x]] ← black
19) color [right [w]] ← black
20) left-rotate [T, P[x]]
21) x ← root [T]
22) else as then clause with left & right exchanged.

case-2

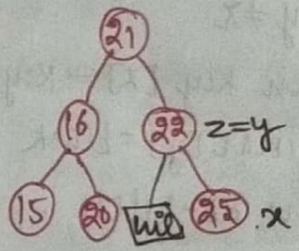
case-3

case-4

Q:-

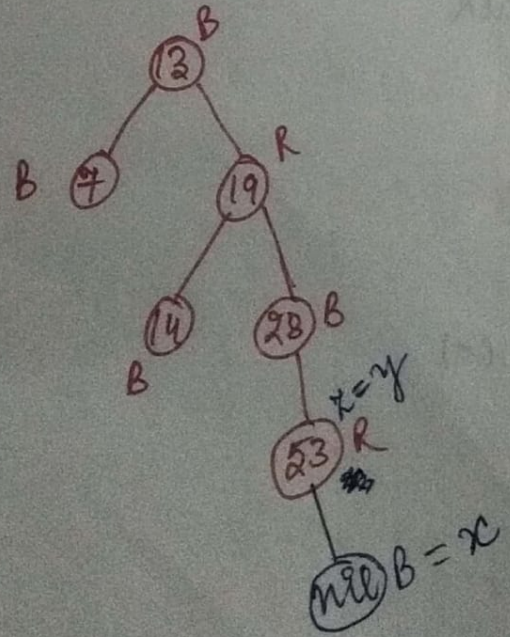


Delete 22



z → node to be deleted

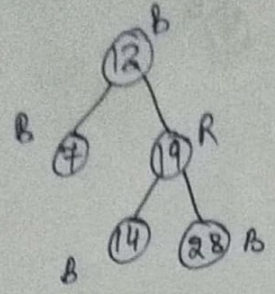
Q:-



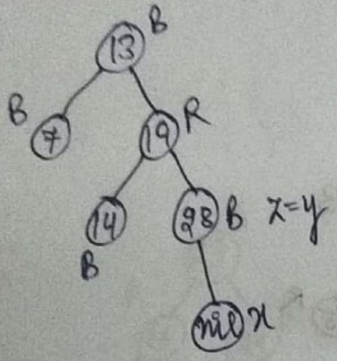
x = left child of y otherwise right child

Delete 53, 28, 19, 14, 13, 7

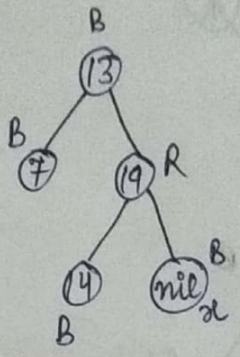
If color of y is red, then no action is performed.
∵ 53 is leaf node, so it is deleted simply



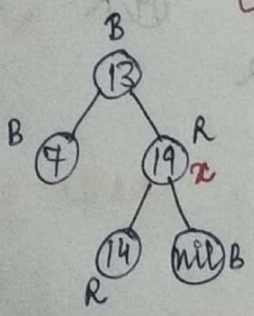
Delete 28:-



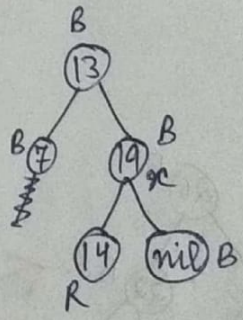
delete fixup called



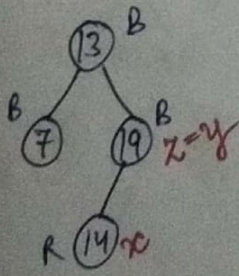
case ② holds



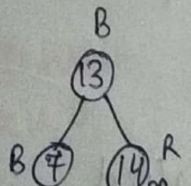
simply color x black



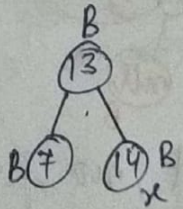
Delete 19:-



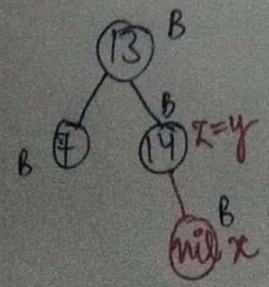
delete fixup called



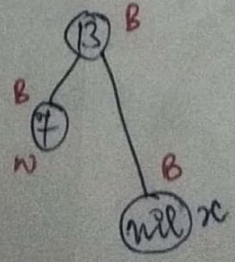
⇒



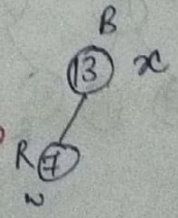
Delete 14:-



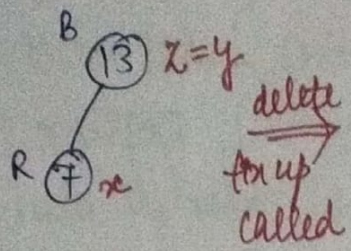
⇒

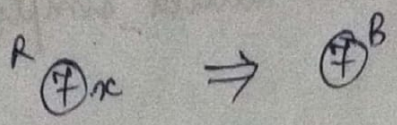


Case ② holds ⇒

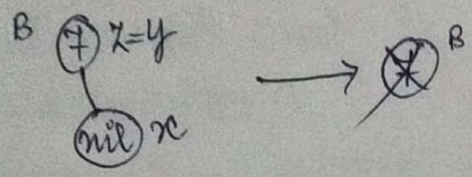


Delete 13:-

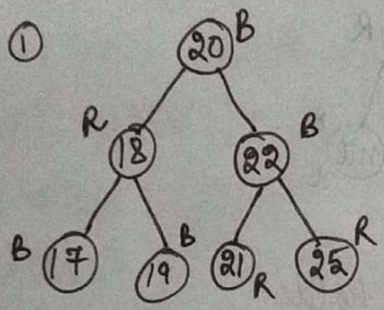




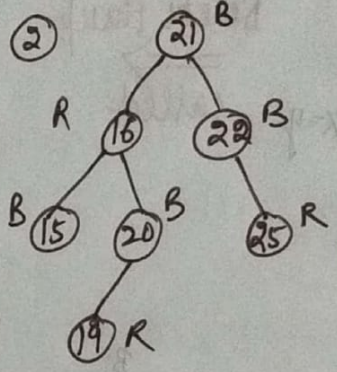
Delete 7:-



Ans: ①

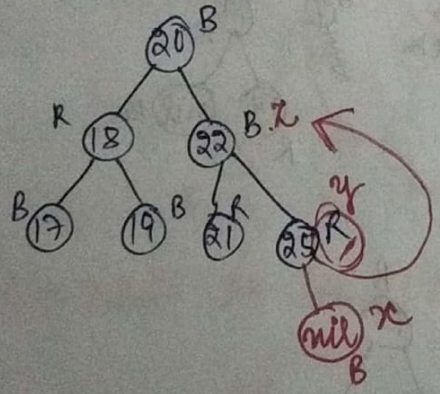


②



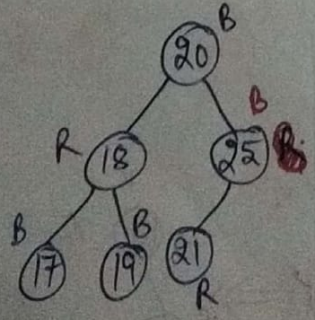
Delete 22

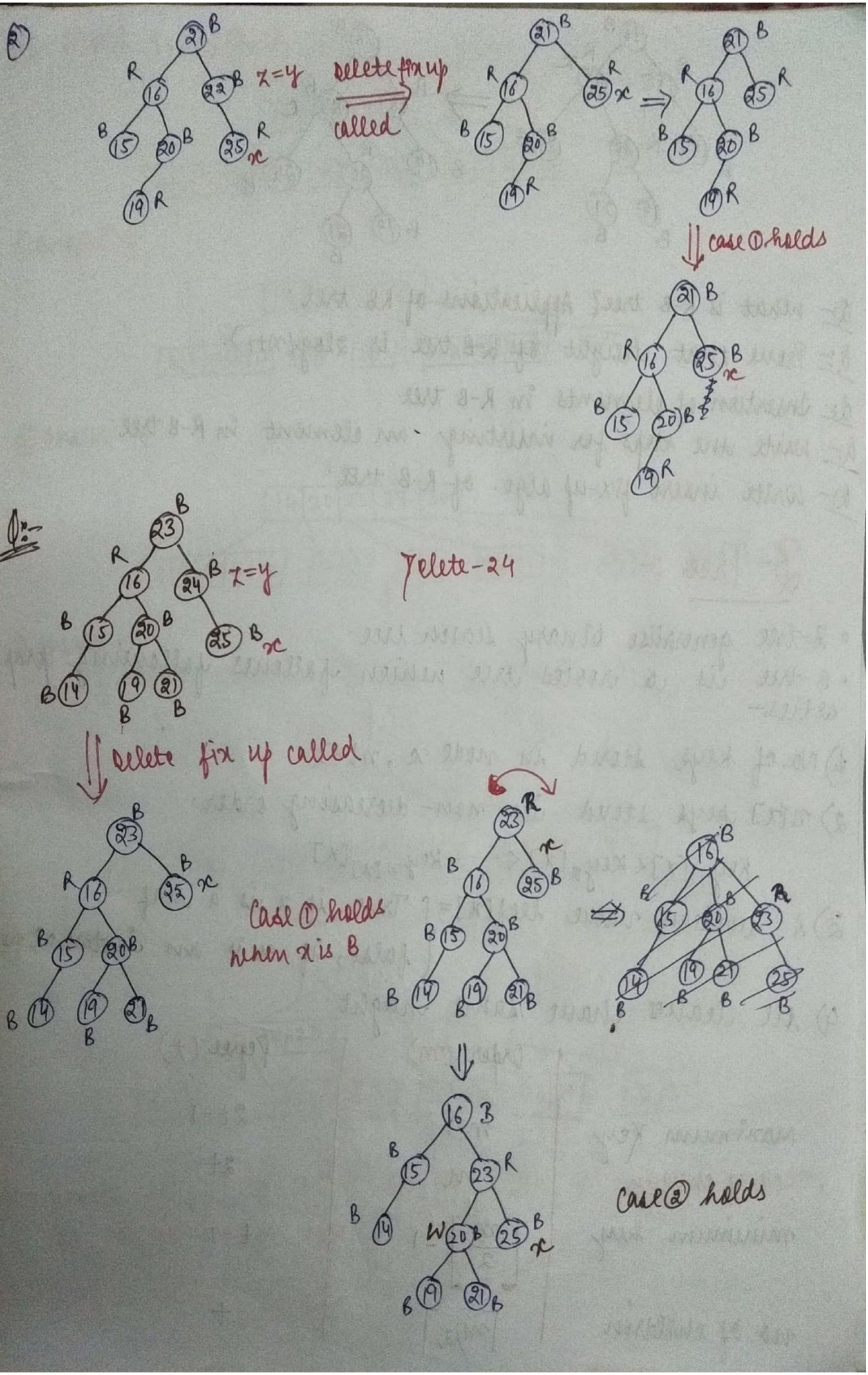
①

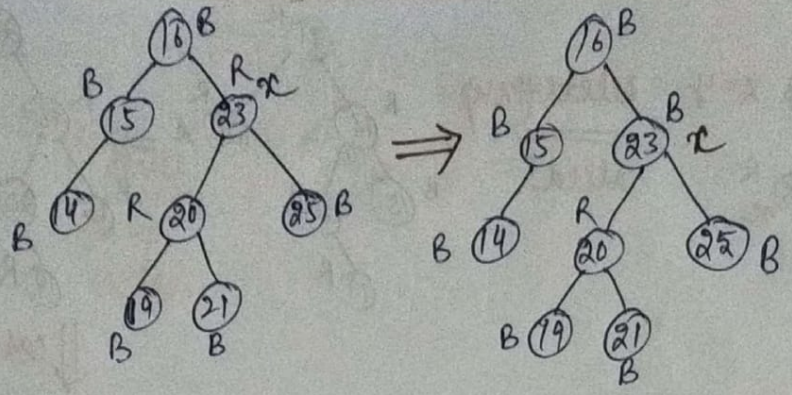


value copied

∵ y is red, no fix up called







Q:- what is R-B tree? Applications of RB tree.

Q:- Prove that height of R-B tree is $2\log(n+1)$.

Q:- Insertion of elements in R-B tree.

Q:- write the steps for inserting an element in R-B tree.

Q:- write insert fix-up algo. of R-B tree.

B-Tree :-

- B-tree generalise binary search tree.
- B-tree is a rooted tree which follows following properties:-

1) No. of keys stored in node x, $n[x]$.

2) $n[x]$ keys stored in non-decreasing order.

$$key_1[x] < key_2[x] < \longrightarrow key_{n[x]}[x]$$

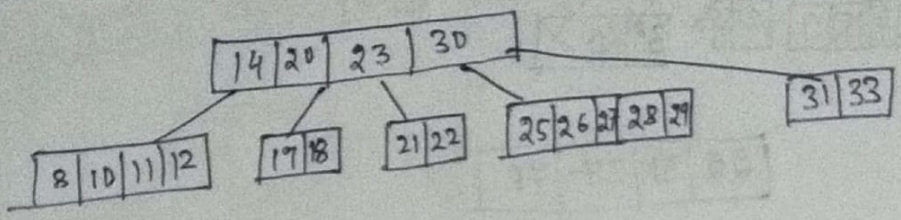
3) A boolean value $leaf[x] = \begin{cases} \text{True, if } x \text{ is a leaf} \\ \text{false, if } x \text{ is an internal node} \end{cases}$

4) All leaves have same height.

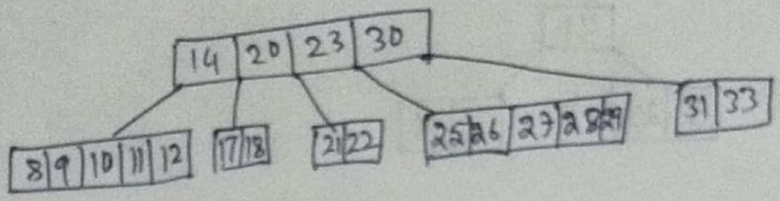
	Order (m)	Degree (t)
Maximum Key	$m-1$	$2t-1$
No. of children	m	$2t$
minimum key	$\left\lceil \frac{m}{2} \right\rceil - 1$	$t-1$
no. of children	$\lceil m/2 \rceil$	t

Q:- Insert 9, 24, 19, 13

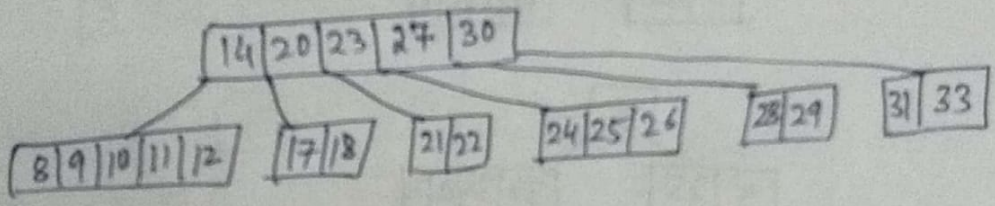
$t = 3$



① Insert 9

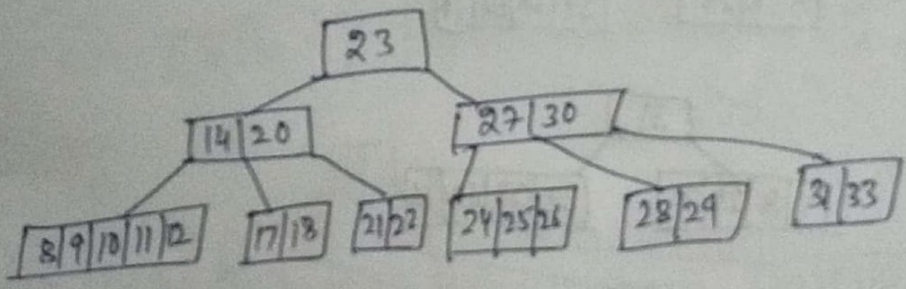


② Insert 24

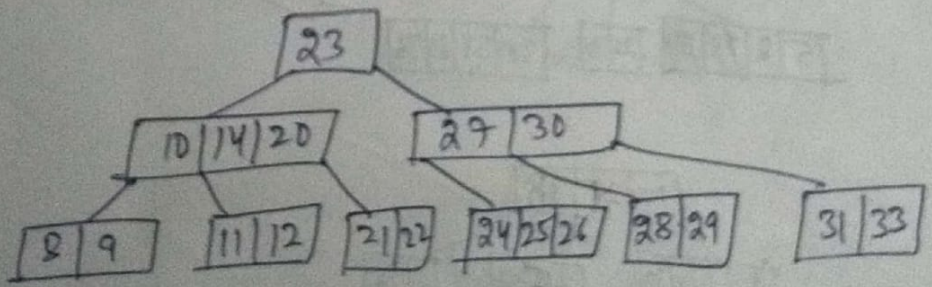


③ Insert 19

If root node is full, then first split it

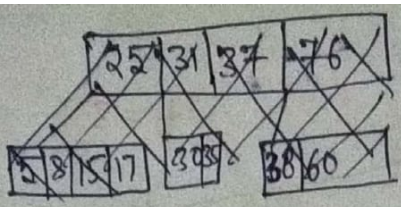


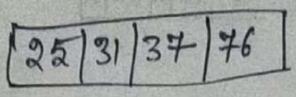
④ Insert 13

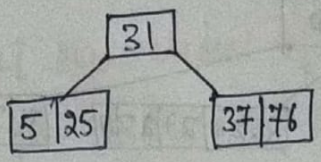


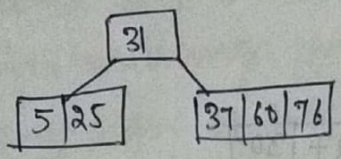
Q:- Insert 25, 31, 37, 76, 5, 60, 38, 8, 30, 15, 35, 17, 23, 53, 27, 43, 55, 48 in an empty B-tree.

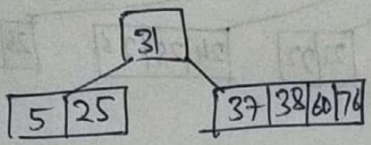
order = 5

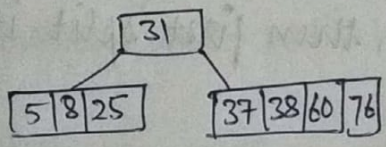


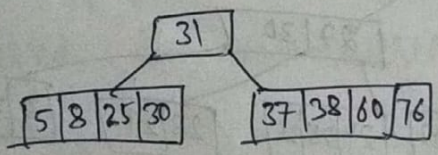
① 

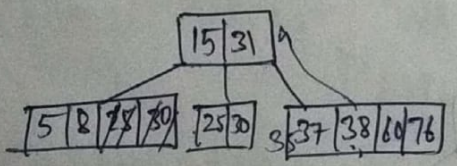
② 

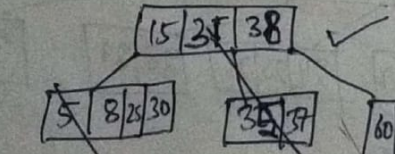
③ 

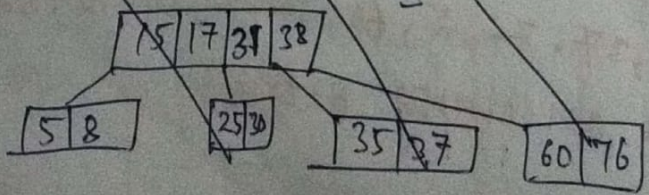
④ 

⑤ 

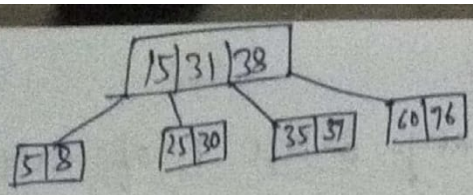
⑥ 

⑦ 

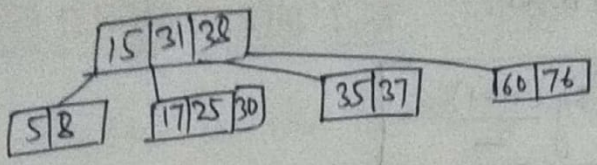
⑧ 

⑨ 

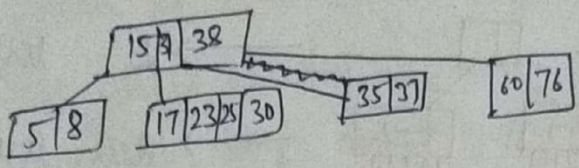
(10)



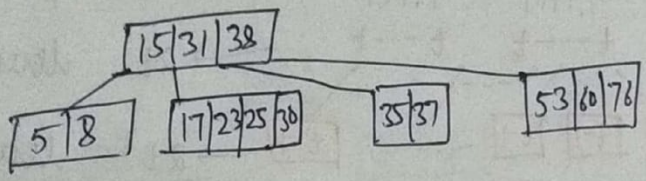
(11)



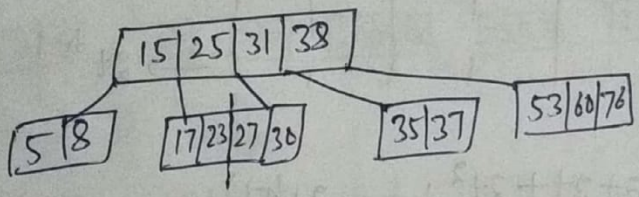
(12)



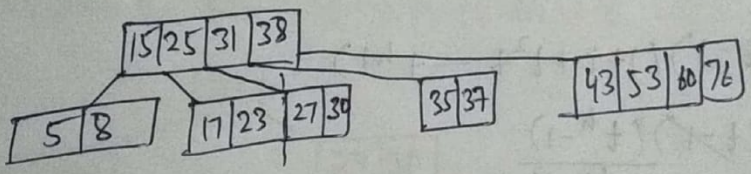
(13)



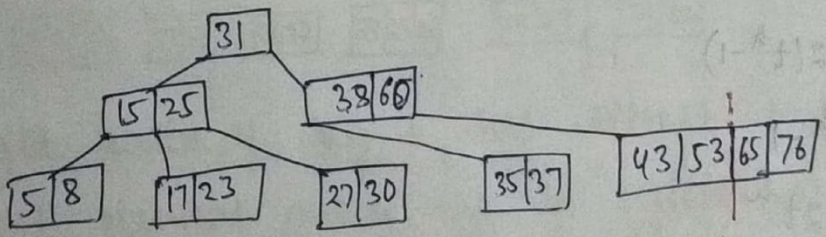
(14)



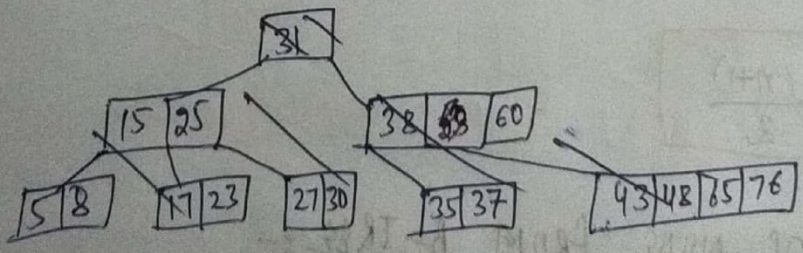
(15)



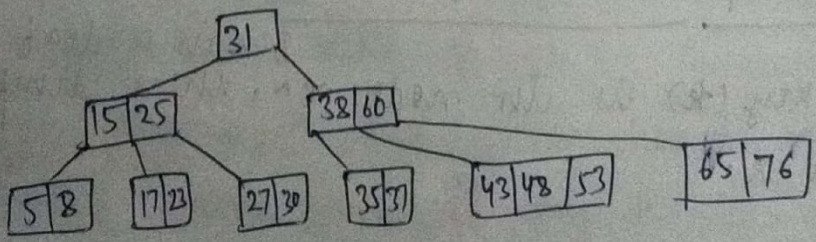
(16)



(17)



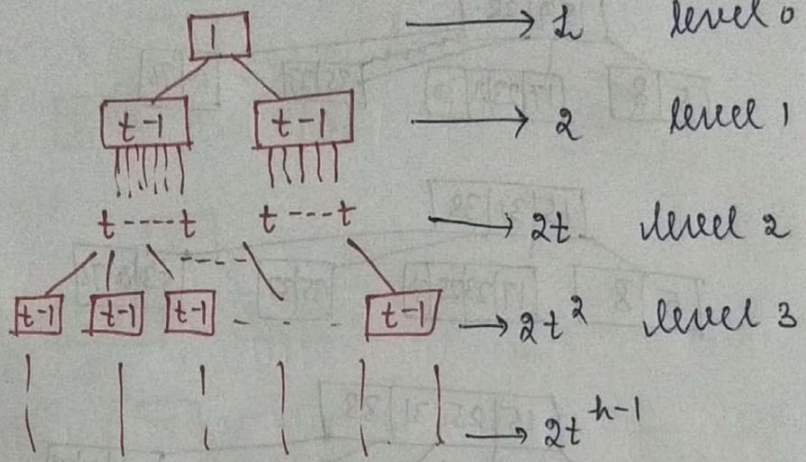
(17)



Theorem: Height of B-tree

If $n \geq 1$, then for any n-key B-tree T of height 'h' and minimum degree $t \geq 2$ is:-

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



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

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

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

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

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

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

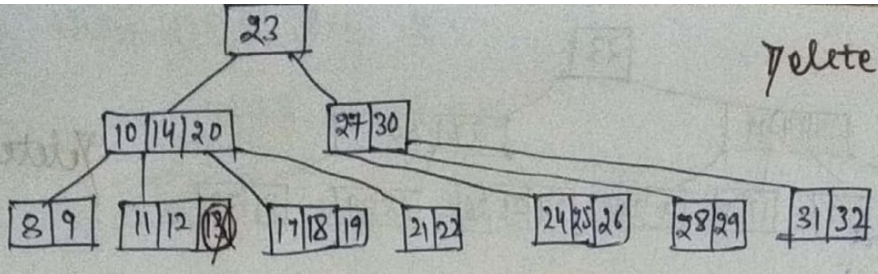
$$\frac{n+1}{2} \geq t^h$$

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

DELETION OF NODE FROM B-TREE :-

Case 1: If Key (k) is in node x & x is a leaf, then simply delete k from x.

Ex:

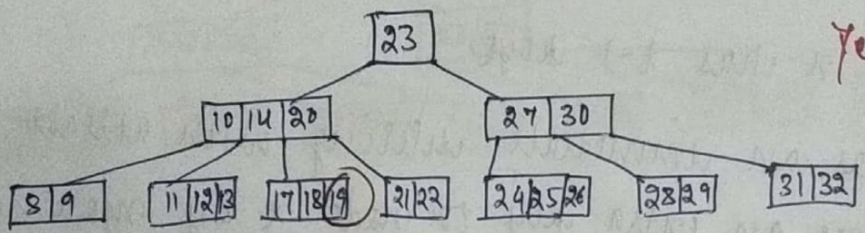


Delete 13

Case 2: a) If key(k) is in node x and x is an internal node, then do the following:-

i) If left child of key k has atleast t keys, then its greatest key moved upto replaced key k.

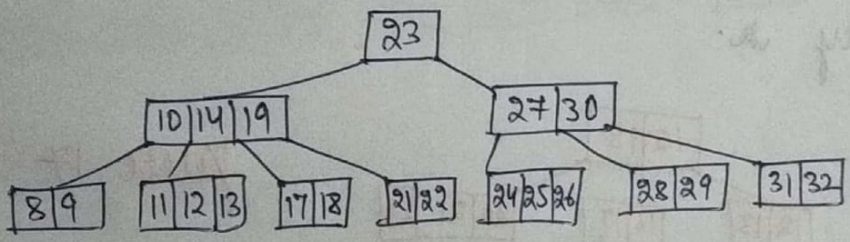
Ex:



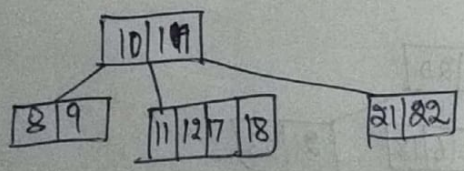
Delete 20

⇓

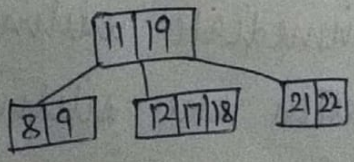
Condition true, replace 19 & delete 20



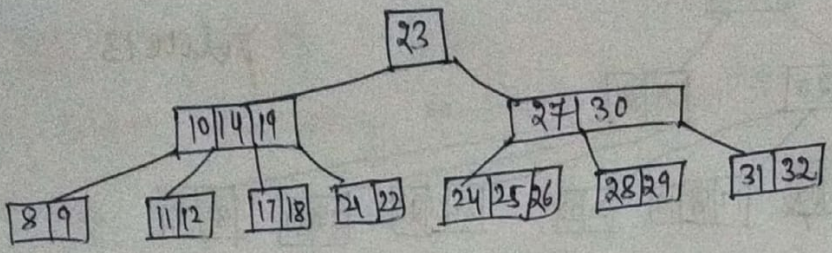
b) If right child of key k has atleast t keys, then smallest element move up to replace key k.



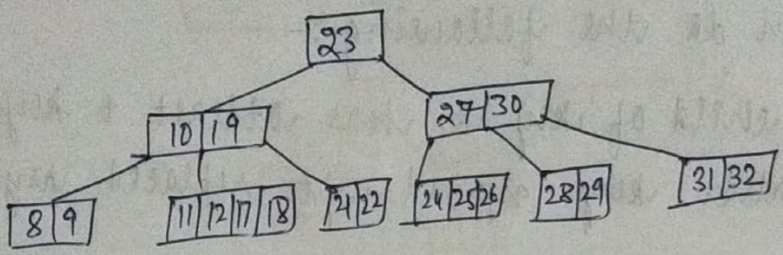
Delete 10



c) If neither left nor right child has t keys, then merge both childs to delete key k.

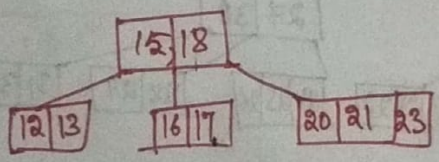


Delete 14



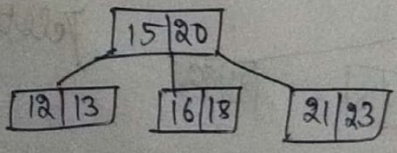
Case 3: If node x has t-1 keys.

i) If x has an immediate sibling, with atleast t keys, then give an extra key to node x by moving a key from x parent & a key move an appropriate from x immediate left. a right sibling up into x parent, then delete key k.



Delete 17

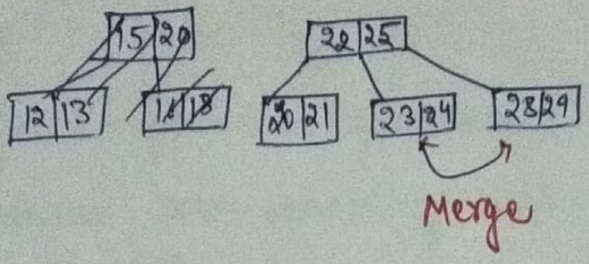
Move 18 from parent to child & smallest key from nearest sibling to parent



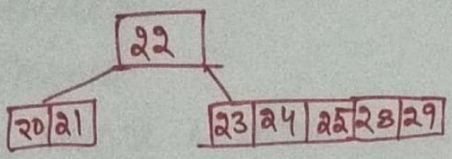
2) If x & both of x immediate siblings have t-1 keys, then merge x with one sibling which involves moving a key from x parent down into new merged node to become the median key for

that node, then delete key k.

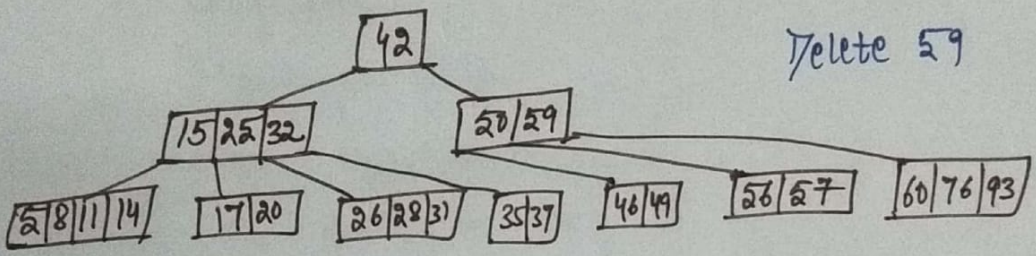
Ex→



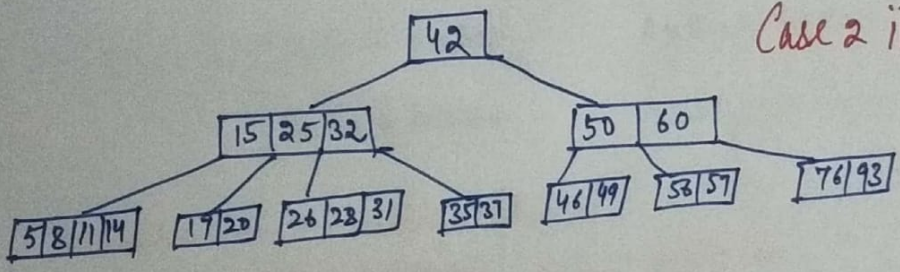
Delete 24



Q:-



Delete 59



Case 2 ii) holds.