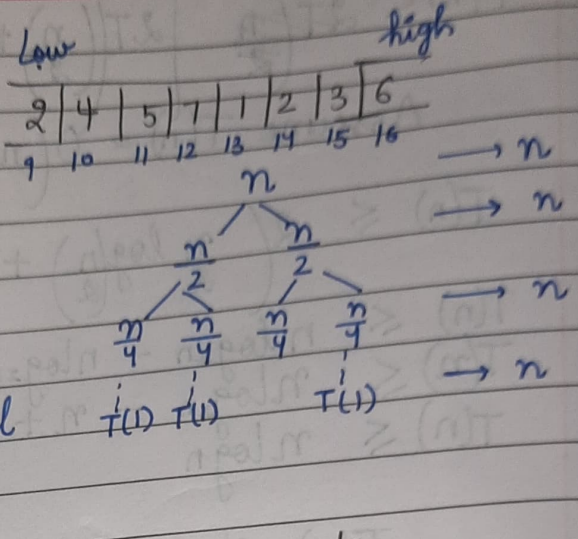
Merge Sort

↳ Divide & Conquer

No cost in dividing

$T(n) = 2T\left(\frac{n}{2}\right) + n$

$Ht = \lfloor \log_2 n \rfloor + 1$



k × Cost at each level

$O(n \log_2 n)$

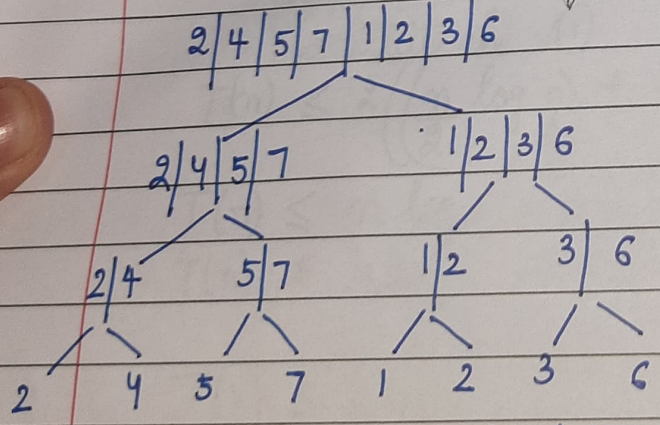
How to break the stability of mergesort

Since Merge Sort is outplace. we require extra storage

Space Complexity $O(n)$

Low = p High = r

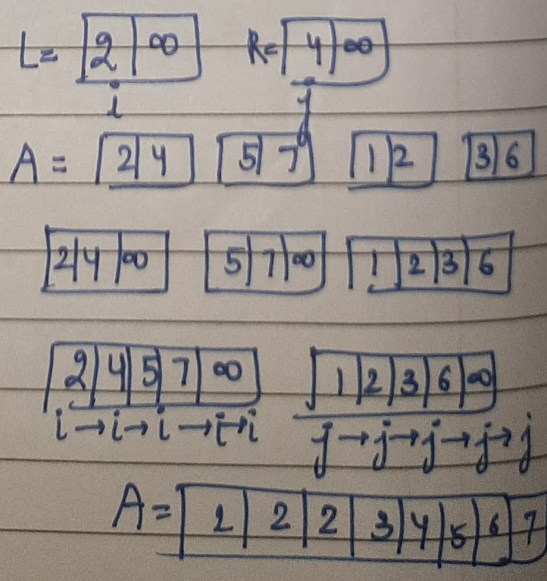
$q = \lfloor (p+r)/2 \rfloor = \lfloor 25/2 \rfloor = \lfloor 12.5 \rfloor = 12$




```
MergeSort (A, p, r)
{ if (p<r)
  { q = ⌊(p+r)/2⌋
  MergeSort (A, p, q)
  MergeSort (A, q+1, r)
  Merge (A, p, q, r)
  }
}
```

```
k ← p to r
if (L[i] <= R[j])
  A[k] ← L[i]
  i = i+1
else
  A[k] ← R[j]
  j ← j+1
```

L = [2 | ∞] R = [4 | ∞]



Merge(A, p, q, r)

① $n_1 \leftarrow q-p+1$ → O(1)
② $n_2 \leftarrow r-q$ → O(1)
③ // Create array from [1 to n_1+1] to [1 to n_2+1] → O(1)
④ for $i \leftarrow 1$ to n_1 → $O(n_1)$
⑤ $L[i] \leftarrow A[p+i-1]$ → O(1)
⑥ for $j \leftarrow 1$ to n_2 → $O(n_2)$
⑦ $R[j] \leftarrow A[q+j]$ → O(1)
⑧ $L[n_1+1] \leftarrow \infty$ → O(1)
⑨ $R[n_2+1] \leftarrow \infty$ → O(1)
⑩ $i \leftarrow 1$ → O(1)
⑪ $j \leftarrow 1$ → O(1)
⑫ for $k \leftarrow p$ to r → O(n)
⑬ if ($L[i] <= R[j]$) → O(1)
⑭ then $A[k] \leftarrow L[i]$ → O(1)
⑮ $i \leftarrow i+1$ → O(1)
⑯ else →
⑰ $A[k] \leftarrow R[j]$ → O(1)
⑱ $j \leftarrow j+1$ → O(1)

Total = $O(n_1) + O(n_2) + O(n)$
= $O(2n)$
= $\underline{\underline{n}}$

A sorting algorithm is called stable if it does not exchange relative pos of items with equal keys

Inplace Sorting → If sorting is occuring within an array We don't require external storage.

Space Complexity $\underline{\underline{O(n)}}$.

Date 23/7/22

Space Complexity O(1) Not Stable
Inplace sorting

Quick Sort → Divide & conquer

r → Last index

Initial index	0	1	2	3	4	5	6	7
	2	8	7	1	3	5	6	4

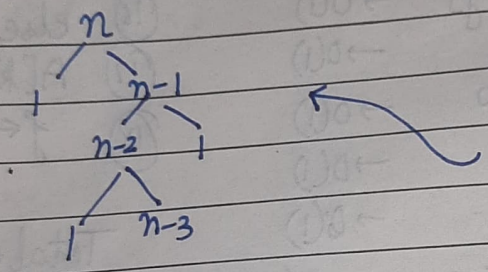
← Pivot

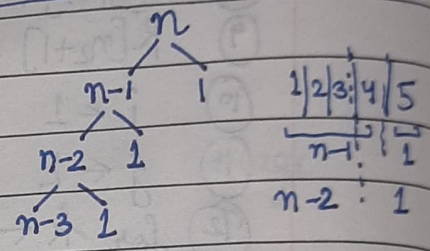
Pivot $x = A[r]$

After partitioning pivot in mid

Best $O(n \log n)$
Avg
Worst $O(n^2)$
↳ When list is already sorted

If list is reverse sorted





If array contains same no of elements 4/4/4...
Worst case occurs
Complexity $O(n^2)$

Worst Case

$T(n) = T(n-1) + n$ → dividing

Best

0	1	2	3	4	5	6	7
2	8	7	1	4	5	6	4

4 → q

$T(n) = 2T\left(\frac{n}{2}\right) + n$

$\Rightarrow O(n \log n)$

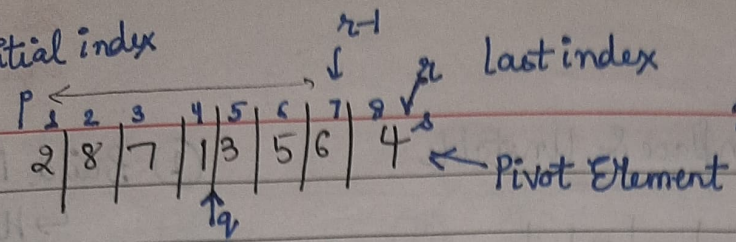
Avg $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + n.$

$T(n) = T\left(\frac{n}{8}\right) + T\left(\frac{7n}{8}\right) + n$

$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$

Initial index



$x = A[r]$

q → pos of pivot element

```
Quick Sort(A,p,r)
{ if(p<r)
  { q ← Partitioning(A,p,q,r)
    Quicksort(A,p,q-1)
    Quicksort(A,q+1,r)
  }
```

~~i ← p-1~~
~~for(j ← p to r-1)~~

j ← p to r-1
i ← p-1
j

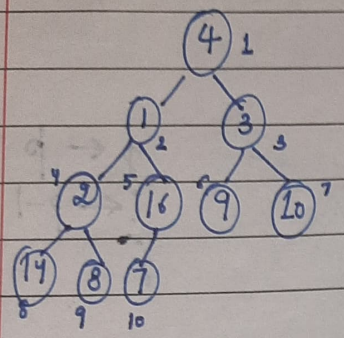
- i=0 j=1
  2<4 True
  i → i+1 2|8|7|1|3|5|6|4
  1 j=2

```
Partitioning(A,p,q,r)
{
  x ← A[r]
  i ← p-1
  for j ← p to r-1
  { if (A[j] ≤ x)
    { i ← i+1
    Exchange A[i] ↔ A[j]
    }
}
  Exchange A[i+1] ↔ A[r]
  return i+1
```

Heap Sort Inplace & Unstable

4	1	3	2	16	9	10	14	8	7

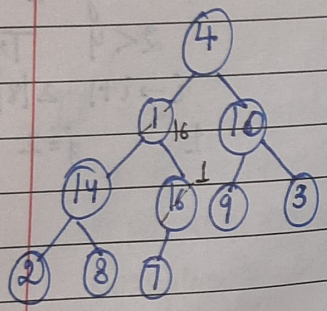
① Build Max Heap
↳ Heapify algo
② Max Heap generated.
Heap Sort applied.



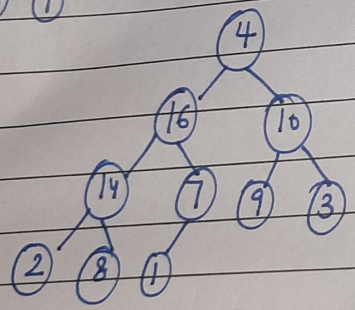
$\lfloor \frac{10}{2} \rfloor$ to 1
5 to 1

Root ko left child se compare

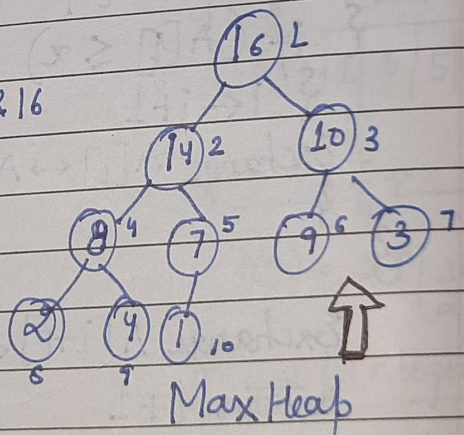
pos 3
9 ko 3 se compare
then 9 ko 10 se



→ To satisfy property 1 compared with 7

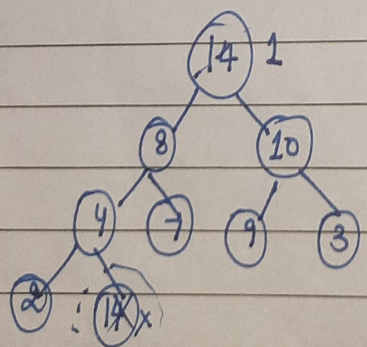


→ 4 & 16



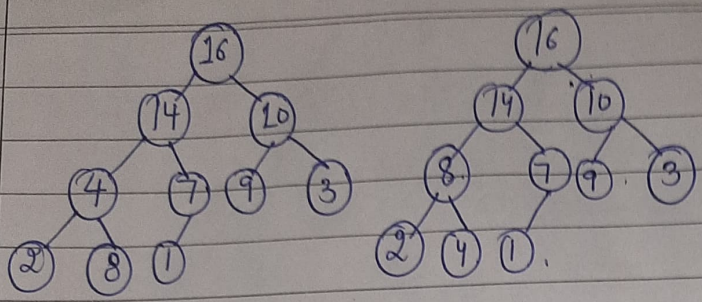
Max Heap

1 & 16 swap. & then decr. heap size

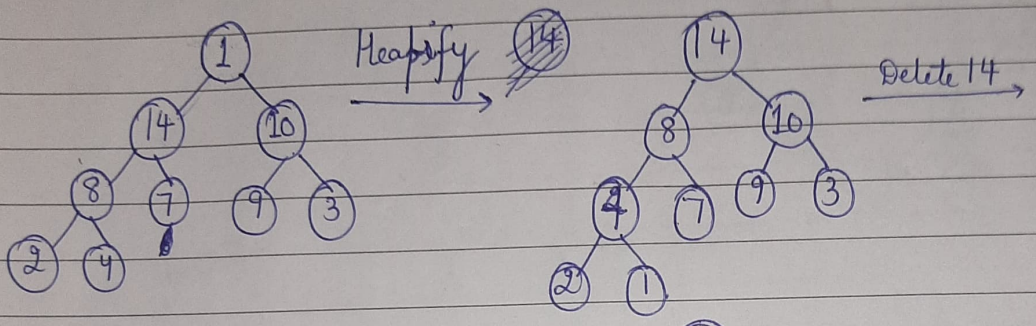


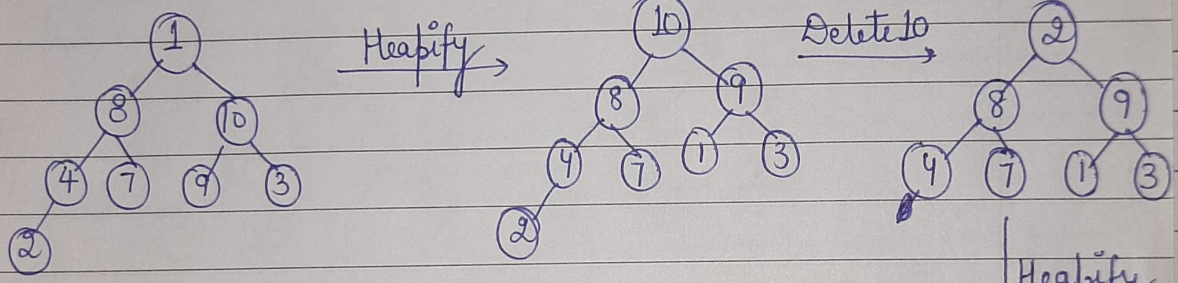
Date...../...../.....

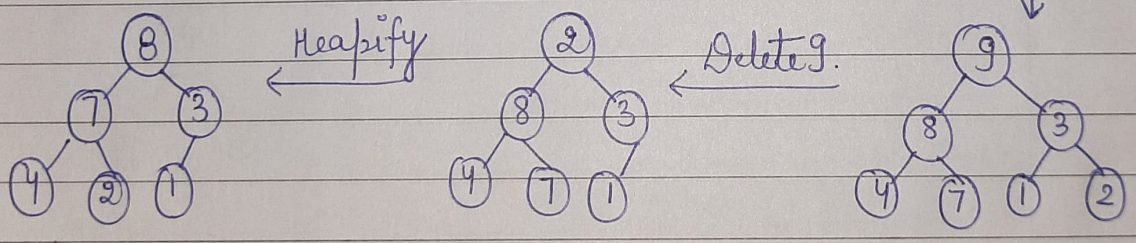
Page.....

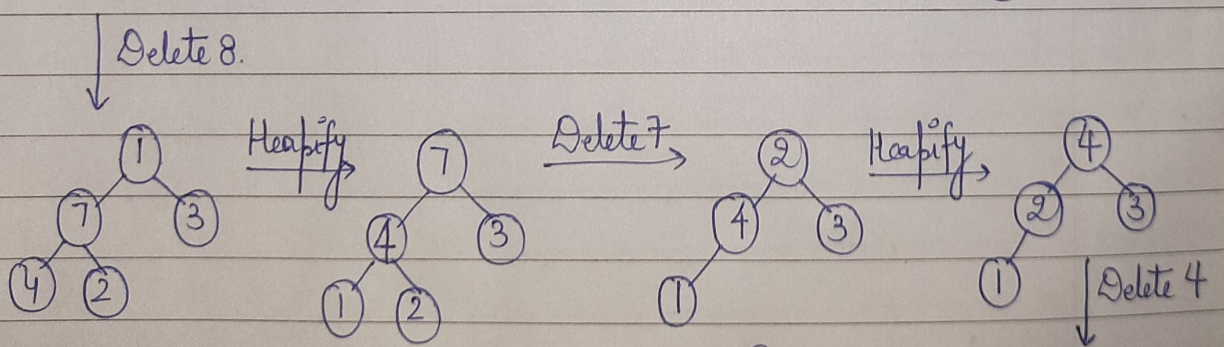


16	14	10	9	8	7	4	3	2	1





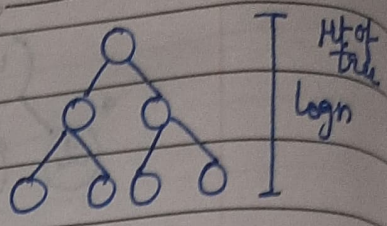




Build Max_heap(A)

1. Heap_size [A] ← length[A]
2. for i ← ⌊length(A/2)⌋ down to 2
3. do Max_heapify(A, i)

O(1)



Max_heapify(A, i)

O(logn)

1. l ← left(i)
2. r ← right(i)
3. if l ≤ heapsize [A] and A[l] > A[i]
4. then largest ← l
5. else largest ← i
6. if r ≤ heapSize[A] and A[r] > A[largest]
7. then largest ← r
8. if largest ≠ i
9. then exchange A[largest] ↔ A[i]
10. Max_heapify(A, largest)

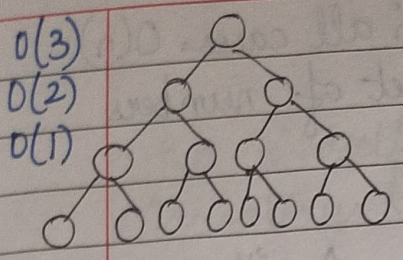
HeapSort(A)

1. Build Max_Heap(A) — O(n)
2. for i ← length(A) down to 1 — n times
3. exchange A[i] ↔ A[1] — O(1)
4. Heapsize [A] ← Heapsize[A] − 1 — O(1)
5. do Max_heapify(A, 1) — O(logn)

$n + n \log n$

$O(n \log n)$

→ O(h) - single node considered.



$\lfloor \log_2 n \rfloor + 1$

Complete BT or Almost Complete BT $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

no of nodes at height 0 $= \left\lceil \frac{15}{2} \right\rceil = \lceil 7.5 \rceil = 8$

" " " " 1 $= \left\lceil \frac{15}{4} \right\rceil = 4$

$2 = \left\lceil \frac{15}{8} \right\rceil = 2$

Running time at one node = O(h)

Total amt of Work done in a heap:

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} * O(h)$$

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} * ch$$

$$\sum_{h=0}^{\log n} \frac{n}{2^h \cdot 2} * ch = \frac{n}{2} * c \sum_{h=0}^{\log n} \frac{h}{2^h}.$$

$$\leq \frac{nc}{2} \left(\sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$ → Harmonic Series

$$\frac{nc}{2} * 2$$

$$\leq cn$$

$$= O(n)$$

Counting Sort → Complexity linear in all cases = O(n)

↳ only work for finite set of numbers

Stable Sort

A = | 3 | 6 | 4 | 1 | 3 | 4 | 1 | 4 |
(1) (2) (3) (4) (5) (6) (7) (8)

Auxillary array vary from 0 to max

Stable Auxilliary array → O(n)

C = | 0 | 2 | 0 | 2 | 3 | 0 | 1 |
(0) (1) (2) (3) (4) (5) (6)

Commulative array C = | 0 | 2 | 2 | 4 | 7 | 7 | 8 |
(0) (1) (2) (3) (4) (5) (6)

↳ 1 index1

Last index represent of each element

A = | 3 | 6 | 4 | 1 | 3 | 4 | 1 | 4 |
(1) (2) (3) (4) (5) (6) (7) (8)

B = | 1 | 1 | 3 | 3 | 4 | 4 | 4 | 6 |
(1) (2) (3) (4) (5) (6) (7) (8)

C = | 0 | 2 | 2 | 4 | 7 | 6 | 8 | 7 |
(0) (1) (2) (3) (4) (5) (6)

for j ← length[A] down to 1
j ← 8 to 1

index

| 0 | 0 | 2 | 2 | 4 | 7 | 7 |
(0) (1) (2) (3) (4) (5) (6)

Counting Sort(A)

1. for i ← 0 to k] Initialisation O(k)
2. C[i] ← 0
3. for j ← 1 to n] frequency O(n) Count
4. do C[A[j]] ← C[A[j]] + 1
5. for i ← 0 to k] Commulative array O(k) for indexing
6. do C[i] ← C[i] + C[i-1]
7. for j ← length[A] down to 1 → Stable] Sorted array O(n)
8. do B[C[A[j]]] ← A[j]
9. C[j] ← C[j] - 1

Unit-2

Red Black Tree

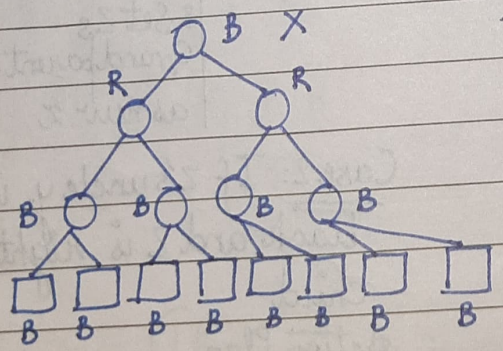
Red parent ka red child ✗

RB is itself self balancing tree
Mandatory Requirement → It is a Binary Search Tree

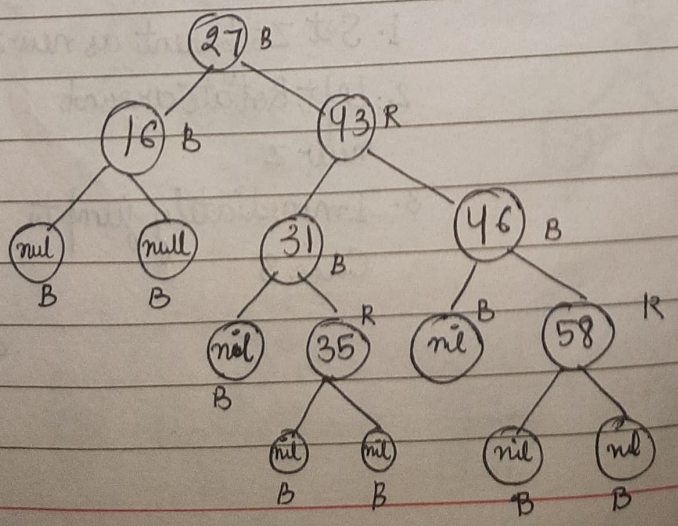
Red Black is a BST which satisfy

1. Every node in RB Tree should be red or black
2. Root node is always black
3. Leaf nodes are always 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 number of black nodes (all path from node to leaf have same black height)

Black Height → A Black Height of a node x is number of black nodes including leaf nodes on the path x to leaf but not counting x.



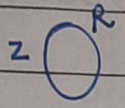
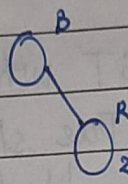
Black Height = 2



bh(46) = 1
bh(58) = 1
bh(43) = 2
bh(27) = 2

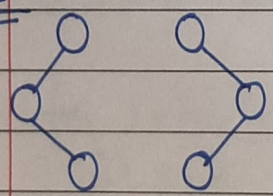
1

Insertion in Re-B Tree

Z is a root node	Z is not root node		
	Zs Parent is Black	Z's Parent is Red	
		Zs Parent is left child of grandparent of Z	Zs Parent is right child of grandparent of Z
			
Action: Simply Color it Black	Action: Nothing to do Because Red can be child of 1 Black	Case 1: If Zs uncle y, is red & Z is left or right child.	Same.
		Action Plan 1. Make Zs Parent & Uncle y Black 2. Make Z.s. Grandparent Red 3. Set Zs Grandparent as new z	

Zig Zag Tree

Case 2:



Case 2: If z's uncle y, is Black and Z is right child

Action Plan

1. Set Zs parent as new z
2. Left Rotate around new z
3. Immediately jump to Case 3

Case 2: If Zs uncle y, is black & Z is left child.

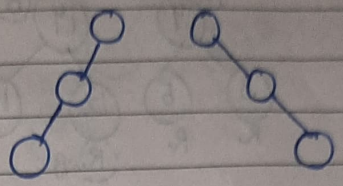
Action Plan

1. Same
2. Right Rotation around new z
3. Immediately jump to cases.

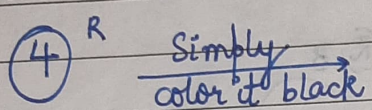
Case3: If z's uncle y is black and z is left child

Action Plan

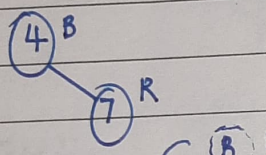
1. Make z's Parent black
2. Make z's grandparent Red
3. Right Rotate around z's grandparent



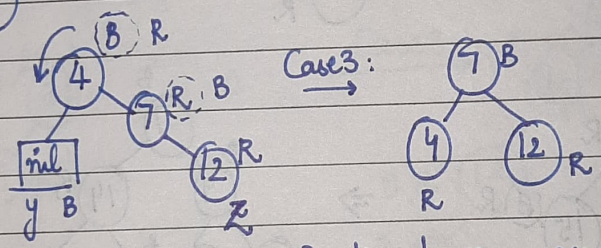
Insertion in RB Tree 4, 7, 12, 15, 3, 5, 14, 18, 16.



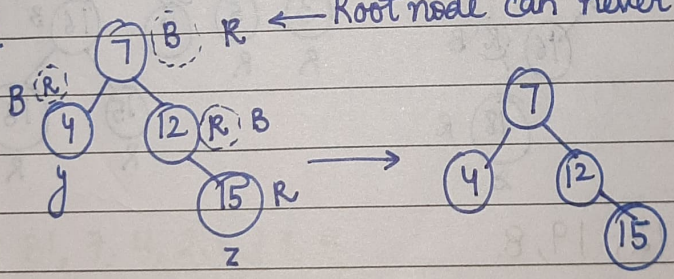
Insert 7



Insert 12 (Case2 or Case3)

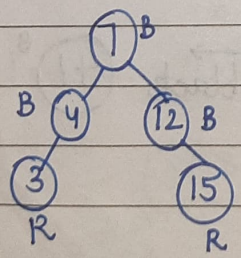


Insert 15

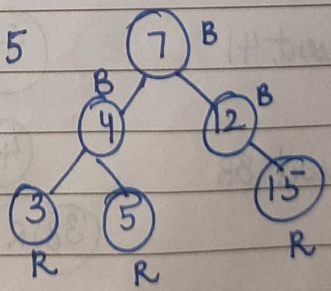


Case 1:

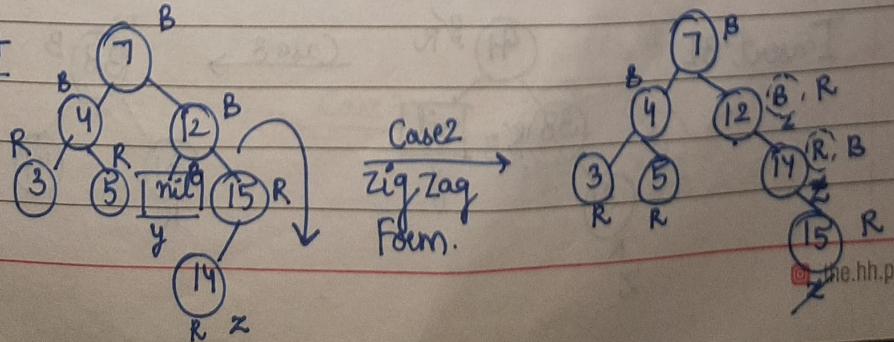
Insert 3



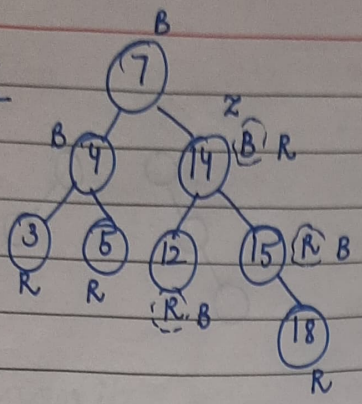
Insert 5



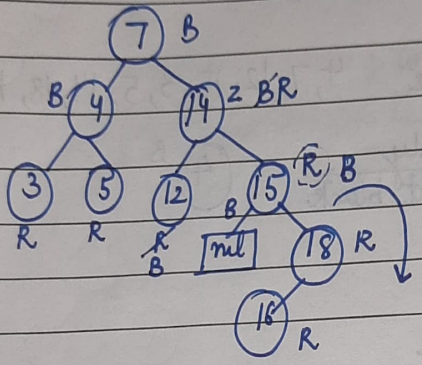
Insert 14

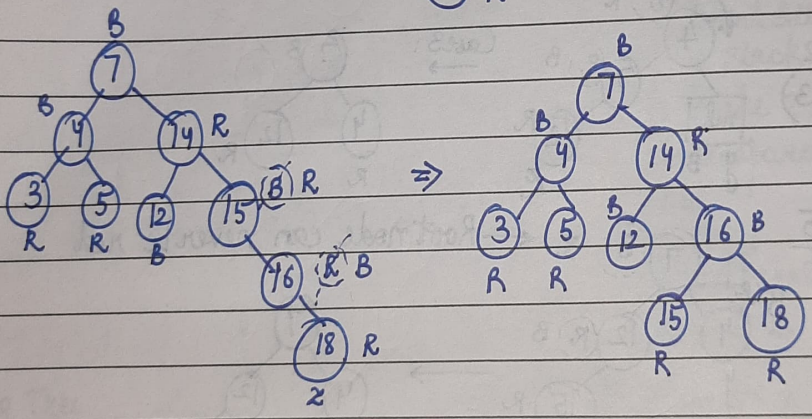


Case3:



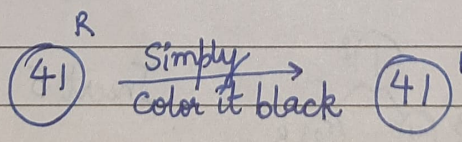
Insert 16



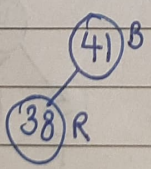


2. 41, 38, 31, 12, 19, 8.

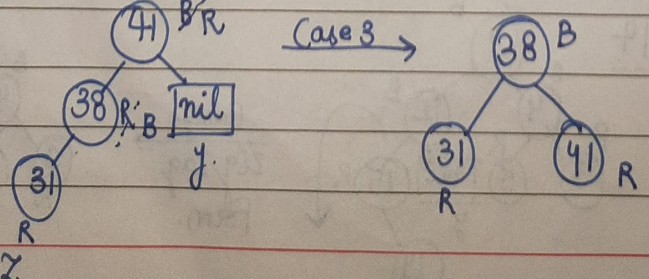
Insert 41



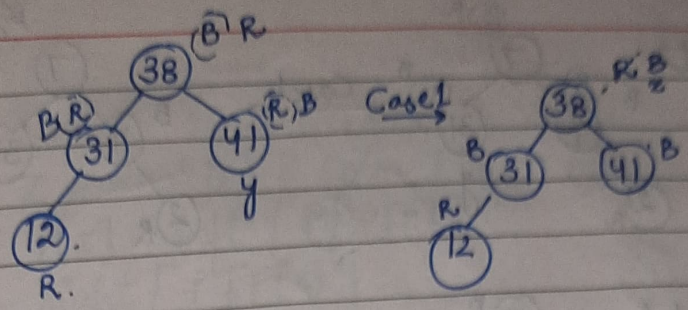
Insert 38



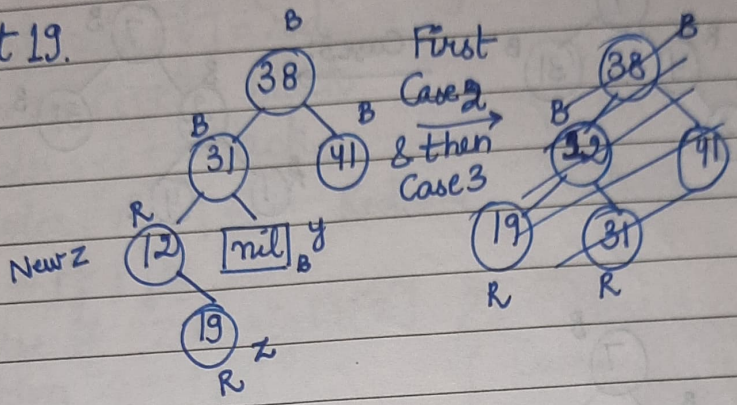
Insert 31



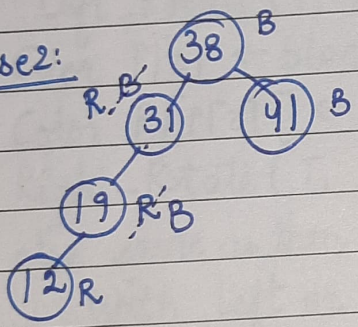
Insert 12



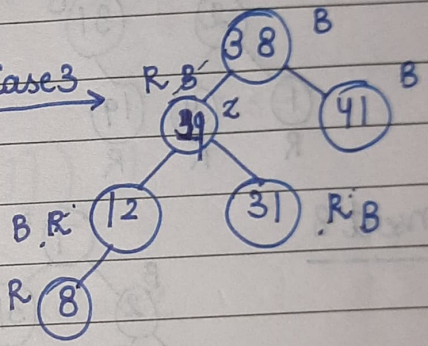
Insert 19.



Case2:

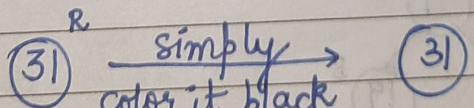


Case3

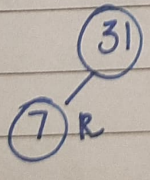


D. Insert 31, 7, 4, 2, 1, 19, 8.

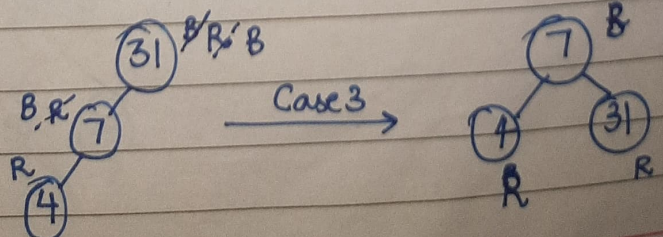
Insert31



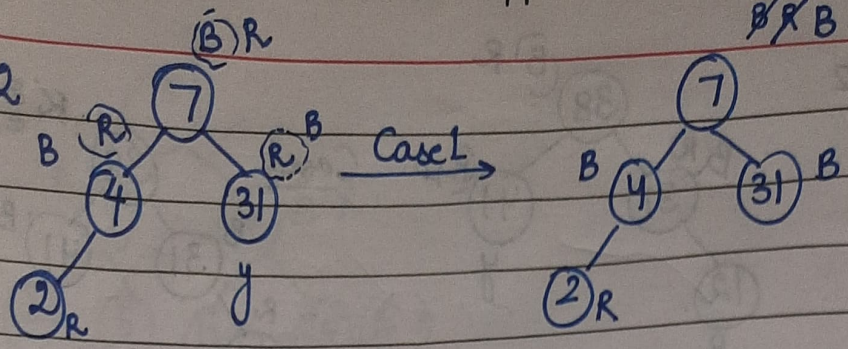
Insert7



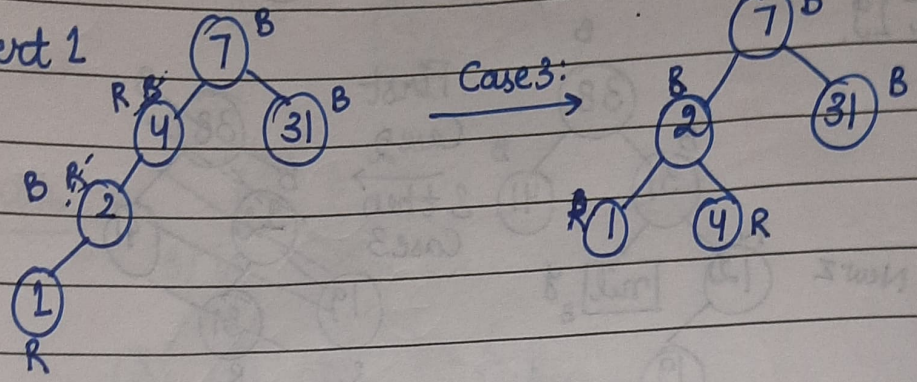
Insert4



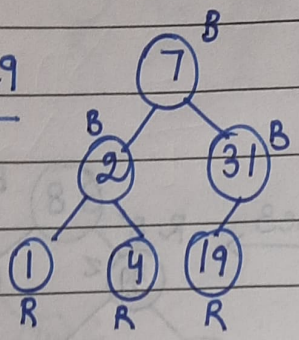
Insert 2



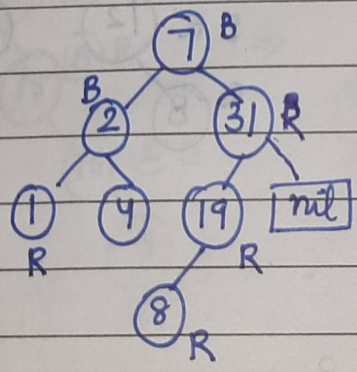
Insert 1

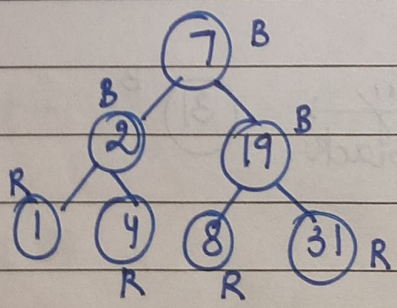


Insert 19



Insert 8





Algorithm

R-B Tree Insert-fixup (T, z)

1. While z ≠ root[T] & Color P[z] = red
2. do if P[z] = left [P[P[z]]
3. y ← right [P[P[z]]
4. if color[y] = red

Case 1:
5. Color P[z] ← black
6. Color[y] ← black
7. Color[P[P[z]] ← Red
8. z ← P[P[z]]

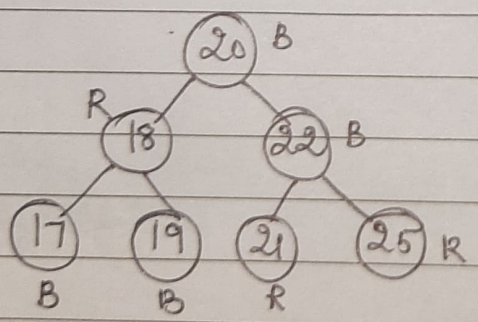
9. else if z ← right[z]
10. then z ← P[z]
11. Left_Rotate (T, z).

Case 2

12. Color P[z] ← Black
13. Color P[P[z]] ← Red
14. Right_Rotate (T, P[P[z]])

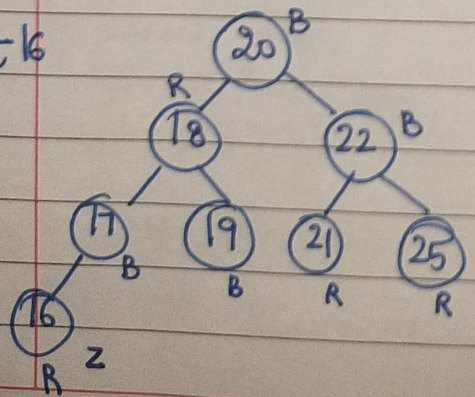
Case 3

15. else same as then clause
With Left or right exchange
16. Color root[T] ← Black

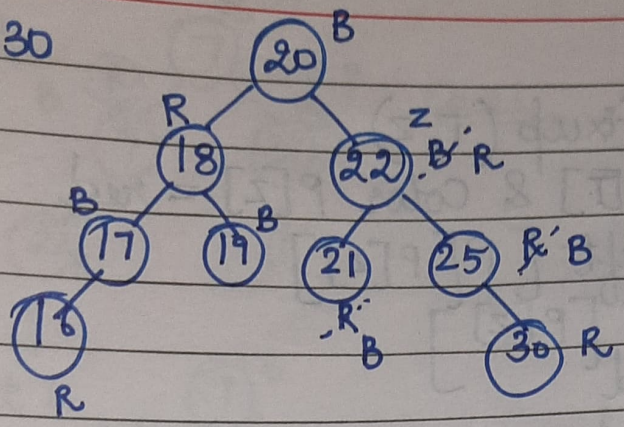


Insert 16, 30, 6

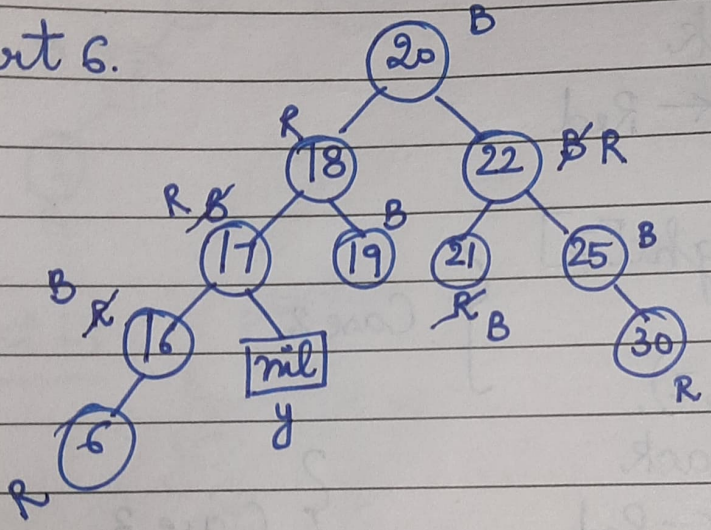
Insert 16



Insert 30



Insert 6.



Date
Page

* Deletion from BST tree

Let x be a node which takes the position of deleted node

x is the root node	x is not root node		
	x's Color is Red	x's color is Black	
		x is left child of Parent	x is right child of Parent.
Simply color it black	Simply color it black	Case 1: 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	3. Right Rotate around x Parent 4. Set left child of x Parent as new w
		Case 2: x Sibling w is black & w's both children are black Action Make w red and set x Parent as new x	Case 2: Same
Case 4: 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		Case 3: x Sibling w is black and w's left child is red and right child is black Action Make w's left child black. 2. Make w red 3. Right rotate around w 4. Set right child of x Parent as new w	Case 3: x sibling w is black & w's right child is red & left child is black Make w's right child black Make w red Left rotate around w 4. Set left child of x Parent as new w