- **Algorithm:** It is a combination of sequence of finite steps to solve a particular problem.

Properties of Algorithm:
- output should be generated after finite time.
- There should be at least one input.
- It's independent from programming language.

Difference between Algorithm and Program:

Algorithm	Program
• Written at Design stage	• written at implementation stage
• Need domain expert	• Need programer
• Written in any language	• written in programming language
• H/W or OS independent	• H/W or OS dependent
• Can be Analyze	• can be tested

Pseduo code: • It's a description of an algorithm that is more structured than usual prose but less formal than a programming language.

- Pseduo code is our preferred notation for describing algorithms.

Measuring the running time of an algorithm:

App 1: Experimental study: write a program that implements the algorithm.

App 2: Frequency count Method:

Prob 1:

```
main()
{
  x = y + z;  → 1
}
```

$O(1)$

Prob 2:

```
main()
{
  x = y + z;  → 1
  for (i = 1; i <= n; i++)
  {  x = y + z;  → n
  }
}
```

$\overline{n+1}$

$O(n)$

Prob 3:

```
main()
{  x = y + z;  → 1
   for (i = 1; i <= n; i++)
   {   x = y + z;  → n
   }
   for (i = 1; i <= n; i++)
   {  for (j = 1; j <= n; j++)
      {  x = y + z;  → n^2
      }
   }
}
```

$\overline{n^2+n+1}$

$O(n^2)$

Prob 4:

```
main()
{
  while (n >= 1)
  {
    n = n - 10  → n/10
  }
}
```

$\Rightarrow O(n)$

Prob 5:

```
main()
{ i = 0
  while (i <= n)
  { i = i + 5  → n/5
  }
}
```

$\Rightarrow O(n)$

Prob 6:

```
main()
{
  while (n >= 1)
  {  n = n/2;
  }
}
```

$$\boxed{\begin{array}{ll} n & K = \log_2 n \\ | & \\ \frac{n}{2} & O(\log_2 n) \\ | & \\ \frac{n}{2^2} & \frac{n}{2^K} = 1 \end{array}}$$

Prob7:

```
main ()
{ i=1
while (i<=n)
{ i = 2*i;
}
}
```

$i=1$

$2*1=2$

2^2

$\vdots$

$2^k=n$

$k=\log_2 n$

$$\boxed{O(\log_2 n)}$$

Prob8:

```
main ( )
{
while (n>2)
{
    n = n^(1/2)
}
}
```

n

$n^{1/2}$

$n^{1/2^2}$

$\vdots$

$n^{1/2^k}$

$n^{\frac{1}{2^k}}=2$

$\frac{1}{2^k}\log_2 n=\log_2 2$

$\log_2 n=2^k$

$k=\log_2\log_2 n$

$$\boxed{O(\log_2\log_2 n)}$$

Prob9:

```
main()
{
while(n>23)
{
    n = n^(1/255)
}
}
```

$n^{\frac{1}{255^k}}=23$

n

$n^{\frac{1}{255}}$

$n^{\frac{1}{255^2}}$

$\vdots$

$n^{\frac{1}{255^k}}$

$\frac{1}{255^k}\log_{23}n=\log_{23}23$

$k=\log_{255}\log_{23}n$

$$\boxed{O(\log_{255}\log_{23}n)}$$

Prob 10:

```
main()
{
  while(n>=15)
  {
     n = n^{1/5}
  }
}
```

n

$n^{1/5}$

$n^{1/5^2}$

$\vdots$

$n^{\frac{1}{5^k}}$

$n^{\frac{1}{5^k}} = 15$

$\frac{1}{5^k} \log_{15} n = \log_{15} 15$

$\log_{15} n = 5^k$

$k = \log_5 \log_{15} n$

$\boxed{O(\log_5 \log_{15} n)}$

Prob 11:

```
main()
{  i=2
   while (i<n)
   { i = i^2
   }
}
```

2

2^2

$(2^2)^2$

$\vdots$

$(2^2)^k = n$

$2^k = \log_2 n$

$k = \log_2 \log_2 n$

$O(\log_2 \log_2 n)$

Prob12:

```
main()
{ i = 3
  while(i<n)
  { i = i^2
  }
}
```

3

3^2

$(3^2)^2$

$\vdots$

$(3^2)^k = n$

$2^k = \log_3 n$

$k = \log_2 \log_3 n$

$\boxed{O(\log_2 \log_3 n)}$

→ **Asymptotic Notations:** The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants.

- Asymptotic notations are the mathematical notations used to describe the running time (Time complexity) of an algorithm.

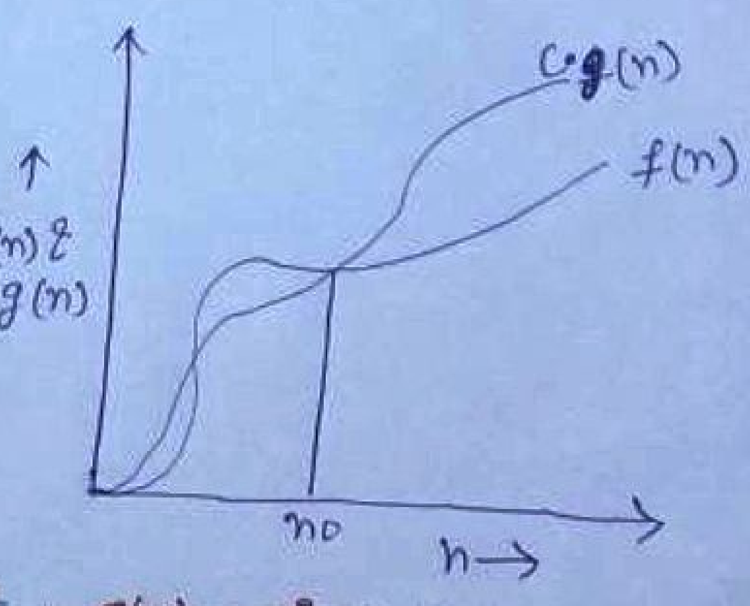
1. **Big O Notation:** • Worst case
   • Upper bound

Let $f(n)$ & $g(n)$ are two +ve functions.

$$f(n) = o(g(n))$$

iff

$f(n) \le c \cdot g(n)$, $\forall n$, $n \geq n_0$ where c is a constant and value of $c > 0$, n_0 is constant and value of $n_0 \geq 1$.



P-1 $g(n) = n^2$, $f(n) = n^2+n+10$

P-2 $f(n) = n+10$, $g(n) = n-10$

P-3 $f(n) = n^2$, $g(n) = n$

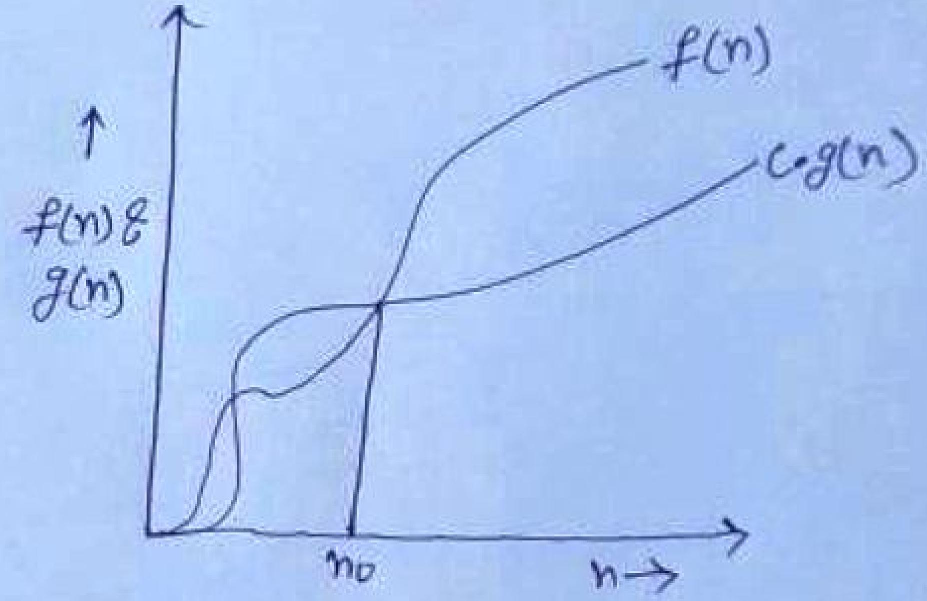
2. Big omega Notation (Ω)

Let f(n) & g(n) are two +ve functions.

$$f(n) = \Omega(g(n))$$

iff

$f(n) \geq c \cdot g(n), \forall n, n \geq n_0$, where c is a constant and value of $c > 0$, n_0 is constant and value of $n_0 \geq 1$.



P-1 $f(n) = n, \ g(n) = n+10$

P-2 $f(n) = n^2+n+10, \ g(n) = n^2$

P-3 $f(n) = n, \ g(n) = n^2$

3. Theta Notation (θ)

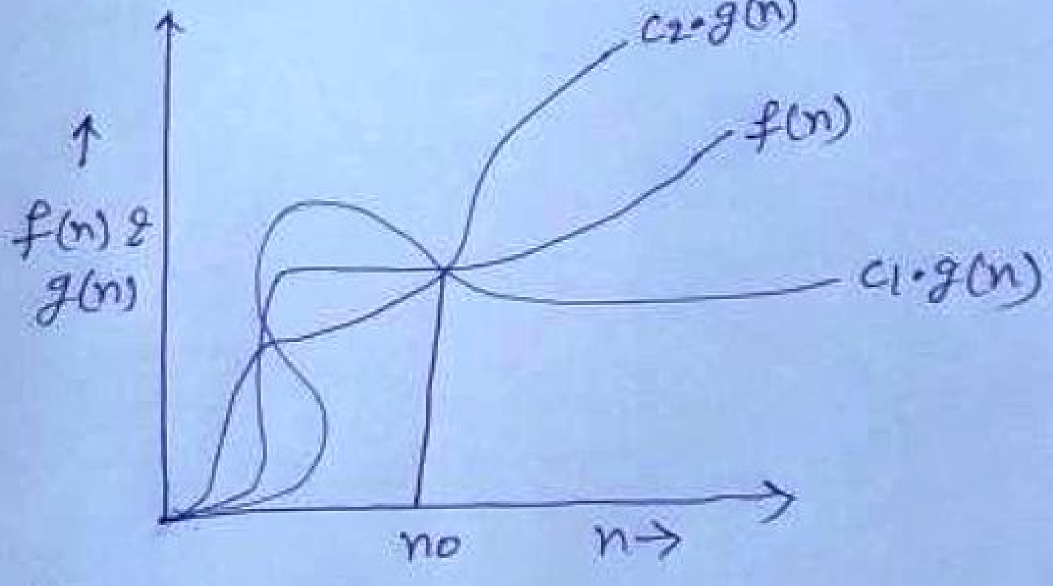
Let $f(n)$ & $g(n)$ are two +ve functions.

$$f(n) = \theta(g(n))$$

iff

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall\ n, n \geq n_0,$$

where c_1, c_2 are constants and value of $c_1, c_2 > 0$, n_0 is a constant and value of $n_0 \geq 1$.



P-1 $f(n) = n$, $g(n) = n+10$

P-2 $f(n) = n$, $g(n) = n$

P-3 $f(n) = n^2$, $g(n) = n^2+n+10$

P-4 $f(n) = n^2$, $g(n) = n$

→ **Recursion:-** A function is calling that itself to solve a particular problem is called a recursion.

- Recursion is nothing but solving bigger problem in terms of smaller problem.
- To Execute the recursive program we used stack data structure.
- Every recursion program should have termination condition.

→ Recurrence relation of factorial:

$$fact(n) = \begin{cases} 1, \text{ if } n \leq 1 \\ n * fact(n-1), \text{ otherwise} \end{cases}$$

→ Recurrence relation of fibonacci series:

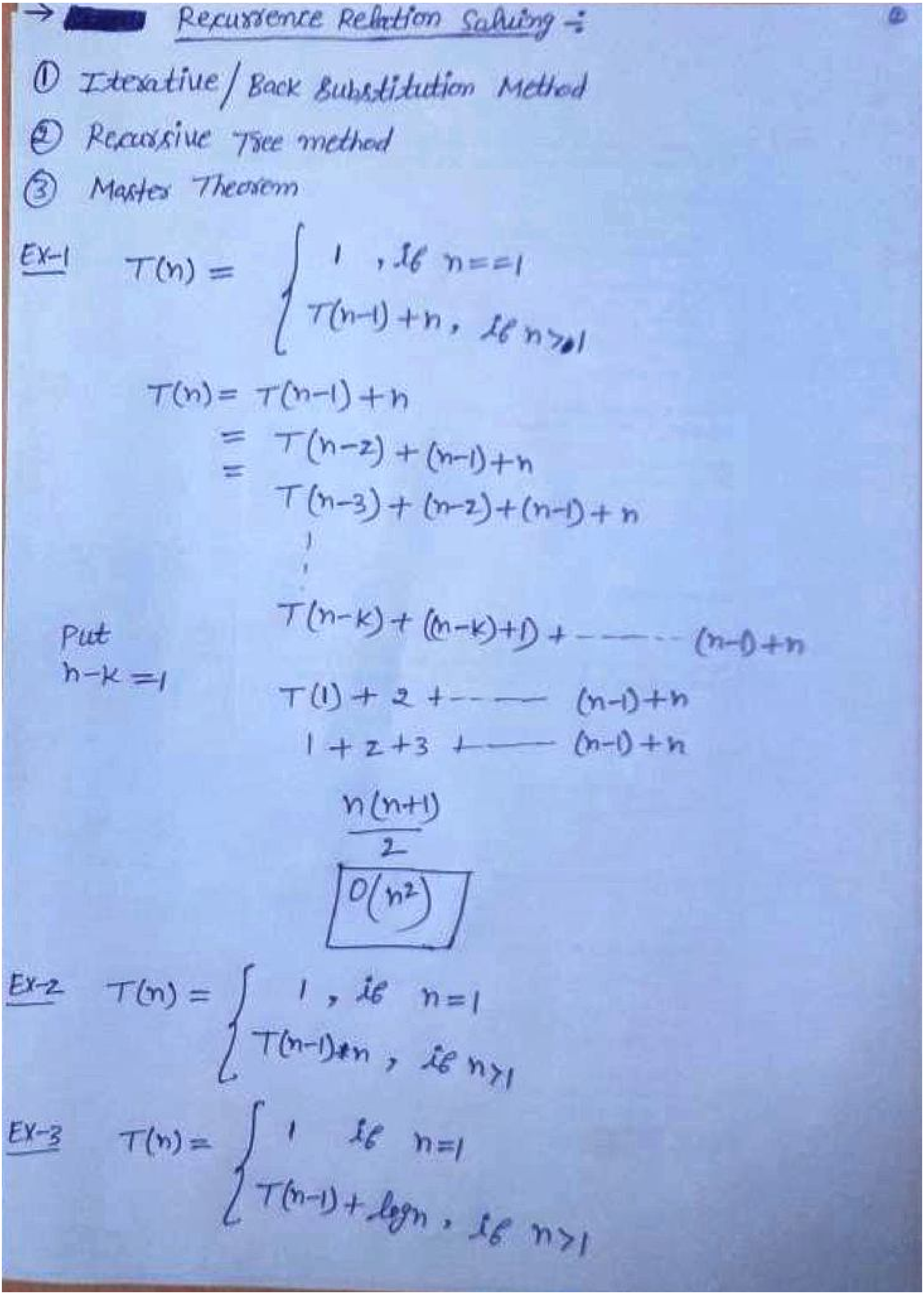
$$fib(n) = \begin{cases} n, \text{ if } n==0 \,||\, n==1 \\ fib(n-1) + fib(n-2), \text{ otherwise} \end{cases}$$

→ Recurrence relation of GCD:

$$GCD(m,n) = \begin{cases} \infty, & \text{if } m==0 \ \& \ n==0 \\ m, & n==0 \\ n, & m==0 \\ GCD(n\%m, m), & \text{otherwise} \end{cases}$$

→ Recurrence relation of multiplication of two number:

$$mul(m,n) = \begin{cases} 0, & m==0 \,||\, n==0 \\ m + mul(m, n-1), & \text{otherwise} \end{cases}$$



→ [redacted] Recurrence Relation Solving :- ②

① Iterative / Back Substitution Method

② Recursive Tree method

③ Master Theorem

Ex-1 $$T(n) = \begin{cases} 1, & \text{if } n==1 \\ T(n-1)+n, & \text{if } n>1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1)+n \\ &= T(n-2)+(n-1)+n \\ &= T(n-3)+(n-2)+(n-1)+n \\ &\ \vdots \\ &\quad T(n-k)+((n-k)+1)+\text{------}(n-1)+n \end{aligned}$$

Put $n-k=1$

$$T(1)+2+\text{------}(n-1)+n$$

$$1+2+3+\text{------}(n-1)+n$$

$$\frac{n(n+1)}{2}$$

$$\boxed{O(n^2)}$$

Ex-2 $$T(n) = \begin{cases} 1, & \text{if } n=1 \\ T(n-1)*n, & \text{if } n>1 \end{cases}$$

Ex-3 $$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)+\log n, & \text{if } n>1 \end{cases}$$

Ex-4

$$T(n)= \begin{cases} 0, & \text{if } n=0 \\ T(n-2)+n^2, & \text{if } n>0 \end{cases}$$

Ex-5

$$T(n) = \begin{cases} 0, & \text{if } n=0 \\ T(n-2)+\log n, & \text{if } n>0 \end{cases}$$

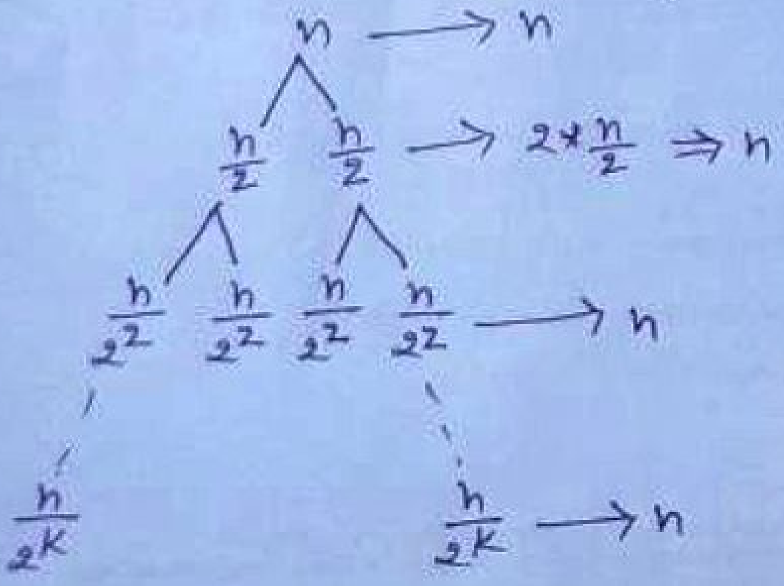
Ex-6

$$T(n) = \begin{cases} 1, & \text{if } n=1 \\ T\left(\frac{n}{2}\right)+n, & \text{if } n>1 \end{cases}$$

→ Recursive Tree Method:

Ex-1

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

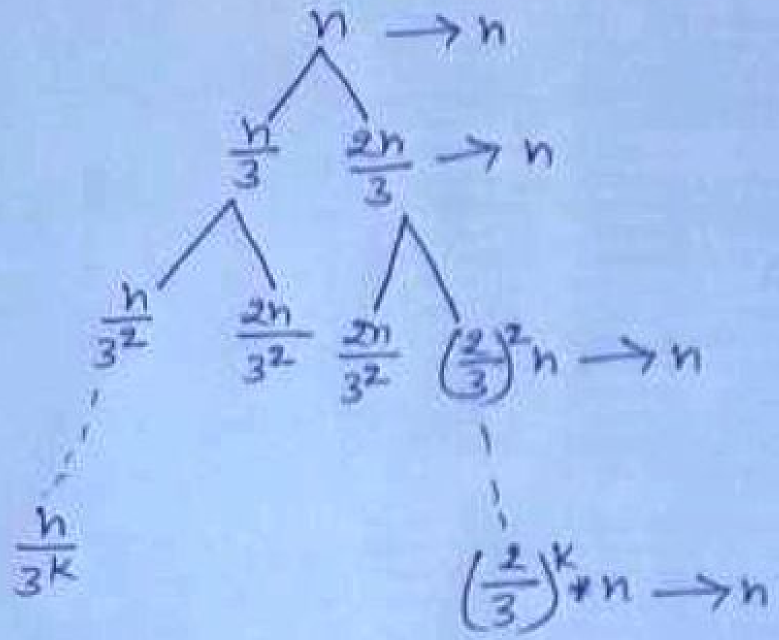


$n \longrightarrow n$
$\frac{n}{2}$ $\frac{n}{2}$ $\longrightarrow 2*\frac{n}{2} \Rightarrow n$
$\frac{n}{2^2}$ $\frac{n}{2^2}$ $\frac{n}{2^2}$ $\frac{n}{2^2}$ $\longrightarrow n$
$\frac{n}{2^k}$
$\frac{n}{2^k} \longrightarrow n$

$$\frac{n}{2^k}=1, \quad k=\log_2 n, \quad T(n) = O(n\log n)$$

$$T(n) = \Omega(n \log n)$$

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


$n \rightarrow n$
$\frac{n}{3}$
$\frac{2n}{3} \rightarrow n$
$\frac{n}{3^2}$
$\frac{2n}{3^2}$
$\frac{2n}{3^2}$
$\left(\frac{2}{3}\right)^2 n \rightarrow n$
$\frac{n}{3^k}$
$\left(\frac{2}{3}\right)^k * n \rightarrow n$

$\frac{n}{3^k} = 1$, $k = \log_3 n$ $\left(\frac{2}{3}\right)^k * n = 1$, $k = \log_{3/2} n$

$T(n) = \Omega(n \log_3 n)$, $T(n) = O(n \log_{3/2} n)$

EX-3 $T(n) = n + T\left(\frac{n}{100}\right) + T\left(\frac{99n}{100}\right)$

EX-4 $T(n) = n + T\left(\frac{n}{5}\right) + T\left(\frac{4 3n}{5}\right)$

→ <u>Master Theorem:</u>

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$a \geq 1$, $b > 1$, a & b are constants, $f(n) \rightarrow +ve$.

Find $n^{\log_b a}$

<u>Case-I</u> If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $\boxed{T(n) = \Theta\left(n^{\log_b a}\right)}$

<u>Case-II</u> If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $\boxed{T(n) = \Theta\left(n^{\log_b a} * \log n\right)}$

<u>Case-III</u> If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and if $a f\left(\frac{n}{b}\right) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n, then

$$\boxed{T(n) = \Theta(f(n))}$$

<u>P-1</u> $T(n) = 8T\left(\frac{n}{2}\right) + n^2$

<u>P-2</u> $T(n) = 8T\left(\frac{n}{2}\right) + n^4$

<u>P-3</u> $T(n) = 2T\left(\frac{n}{2}\right) + n$

<u>Note:</u> If $n^{\log_b a}$ is logarithimic time smaller then $f(n)$ then if $f(n) = \Theta(n^{\log_b a} . (\log n)^k$, where k is constant, $k \geq 0$

$$T(n) = \Theta\left(n^{\log_b a} . (\log n)^{k+1}\right)$$

<u>P-1</u> $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$

Note: If recurrence relation contain root operator:

Ex-1 $T(n) = T(\sqrt{n}) + c$

Assume $n = 2^k$, $k = \log_2 n$

$$T(2^k) = T(2^{k/2}) + c$$

$$T(2^k) = S(k)$$

$$S(k) = S\left(\frac{k}{2}\right) + c$$

$a = 1$, $b = 2$, $f(n) = c$

case III hold

$$S(k) = \theta(\log k)$$

$$T(2^k) = \theta(\log k)$$

$$\boxed{T(n) = \theta(\log \log n)}$$

Ex-2 $T(n) = 2T(\sqrt{n}) + \log n$

Assume $n = 2^k$

$$T(2^k) = 2T(2^{k/2}) + \log 2^k$$

$$T(2^k) = S(k)$$

$$S(k) = 2S\left(\frac{k}{2}\right) + \log 2^k$$

case III hold

$$S(k) = \theta(k \log k)$$

$$T(2^k) = \theta(k \log k)$$

$$\boxed{T(n) = \theta(\log n \log \log n)}$$

Heap Sort

```
HeapSort (A)
Build-Max-Heap (A)
for i = length(A) to 2
    Swap (A[1] with A[i])
    heap-size [A] = heap-size[A] - 1
    Max-Heapify (A,1)
```

- Comparision based Sorting
- Unstable Sorting
- In place Sorting

```
Build-Max-Heap (A)
  heap-size [A] = length [A]
  for i = length [A]/2 to 1
    Max-Heapify (A,i)
```

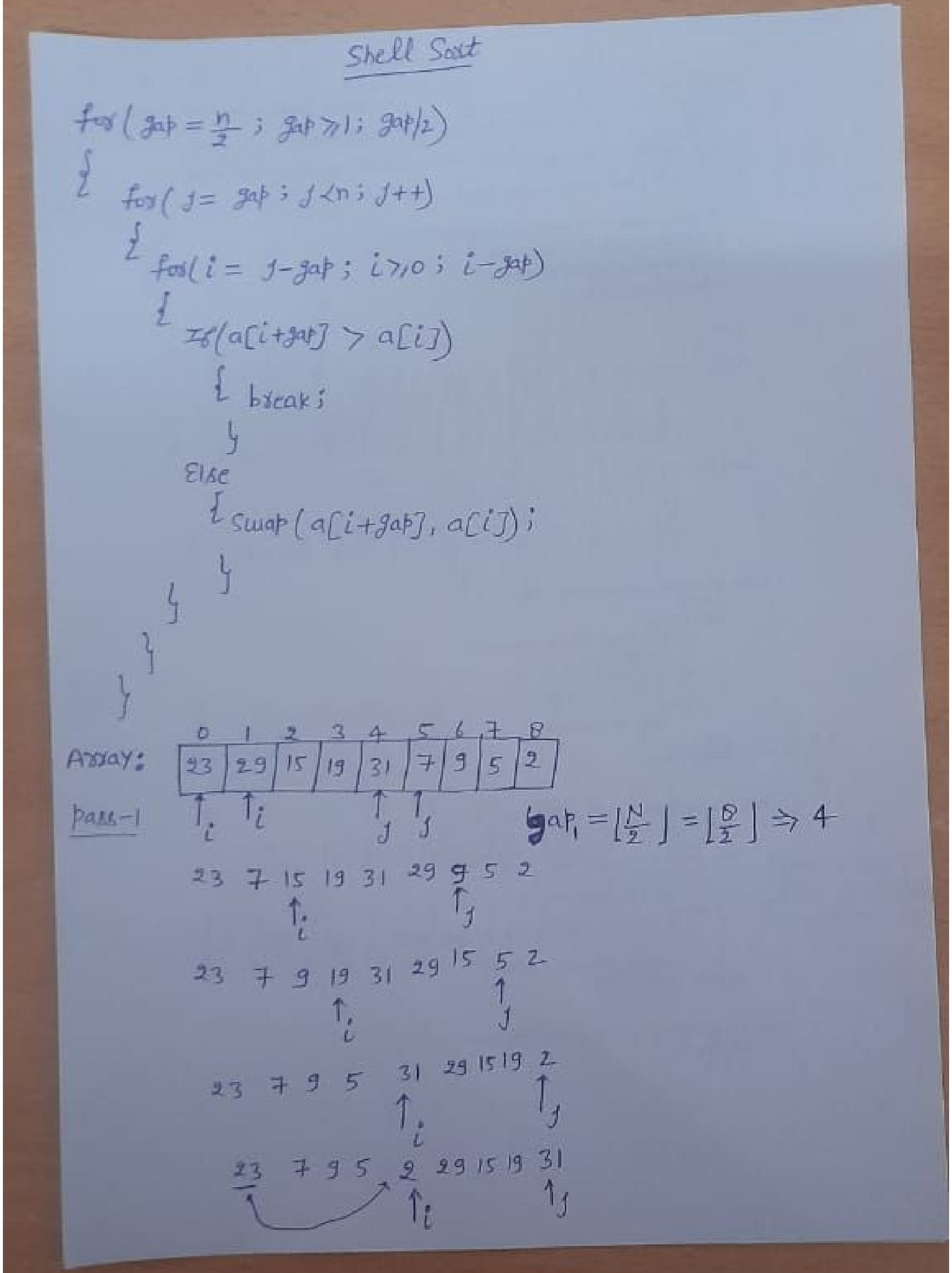
Time Complexity

BC = O(n)

AC = O(n logn)

WC = O(n logn)

```
Max-Heapify (A, i)
  L = left(i)
  R = Right (i)
  if L ≤ A.heap-size and A[L] > A[i]
     Largest = L
  Else
     Largest = i
  if R ≤ A.heap-size and A[R] > A[Largest]
      Largest = R
  if Largest ≠ i
  Swap A[i] with A[Largest]
  Max-Heapify (A, Largest)
  End
```



Shell Sort

```
for( gap = n/2 ; gap >= 1; gap/2)
{
   for( j= gap ; j<n; j++)
   {
     for( i = j-gap ; i>=0 ; i-gap)
     {
       If(a[i+gap] > a[i])
         { break;
         }
       Else
         { swap ( a[i+gap], a[i]);
         }
     }
   }
}
```

Array:

0	1	2	3	4	5	6	7	8
23	29	15	19	31	7	9	5	2

pass-1

```
↑i  ↑i        ↑j  ↑j
```

$gap_1 = \lfloor \frac{N}{2} \rfloor = \lfloor \frac{8}{2} \rfloor \Rightarrow 4$

```
23  7  15  19  31  29  9  5  2
       ↑i              ↑j

23  7  9  19  31  29  15  5  2
          ↑i              ↑j

23  7  9  5  31  29  15  19  2
             ↑i              ↑j

23  7  9  5  2  29  15  19  31
             ↑i             ↑j
```

```
2  7  9  5  23  29  15  19  31
            i               ↑j
```

$gap_2 = \frac{gap_1}{2} = \frac{4}{2} = 2$

pass-2

```
2   7   9   5   23   29   15   19   31
↑i  ↑i  ↑j  ↑j

2   5   9   7   23    29   15   19   31
        ↑i  ↑i  ↑j↑i  ↑j   ↑j

2   5   9   7   15   29   23   19   31
                ↑i   ↑i   ↑j   ↑j

2   5   9   7   15   19   23   29   31
                     ↑i   ↑i   ↑j   ↑j
```

pass-3

```
2   5     9     7   15   19   23   29   31
↑i  ↑j↑i  ↑j↑i  ↑j

2   5   7   9     15    19    23    29    31
        ↑i  ↑j↑i  ↑j↑i  ↑j↑i  ↑j↑i  ↑j↑i  ↑j
```

$gap_3 = \frac{gap_2}{2}$

$\Rightarrow \frac{2}{2} = 1$

works as like Insertion sort

2	5	7	9	15	19	23	29	31

B-Trees

- B-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices.
- B-Trees are better at minimizing disks I/o operations.
- Database systems use B-Trees and its variants for indexing.

→ A B-tree T is a rooted tree(whose root is T.root) having following properties:

1. Every node x has the following attributes:

   a. x.n, the number of keys currently stored in node x.

   b. The x.n keys themselves, x.key1, x.key2 —— x.keyn, stored in non-decreasing, so that x.key1 <= x.key2 --- x.keyn

   c. x.leaf, a Boolean value that is true if x is a leaf and false if x is an internal node.

2. Each internal node x also contain x.n+1 pointers x.c1, x.c2 --- x.n+1 to its children. Leaf nodes have no children, and so their ci attributes are undefined.

3. The key x.keyi seperate the ranges of keys stored in each subtree, if ki is any key stored in the subtree with root x.ci then,

$$k_1 \leq x.key_1 \leq x.key_2 \text{ ---- } x.key_n \leq k_{n+1}$$

4. All leaves have the same deapth, which is the tree's height h.

5. Nodes have lower and upper bounds, on the number of keys, they can contain.

These bounds are fixed integer $t \geq 2$ is called minimum degree of B-tree:

a. Every node other than the root, must have at least $(t-1)$ keys.

Every internal node other than the root thus has at least 't' children. If the tree is non-empty, the root must have at least one key.

b. Every node must contain at most $(2*t-1)$ keys.

Therefore an internal node may have atmost $2*t$ children.

A node is full if it has $(2*t-1)$ keys.

B-Trees - Creation

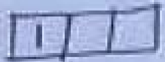
Array = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Degree $t = 2$

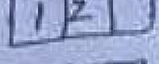
Soln: Min keys $= (t-1) = 1$

Max keys $= (2t-1) = 3$

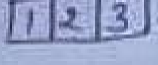
Step-1 Insert-1



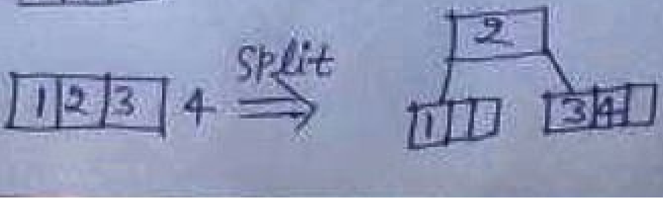
Step-2 Insert-2



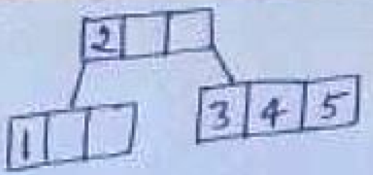
Step-3 Insert-3



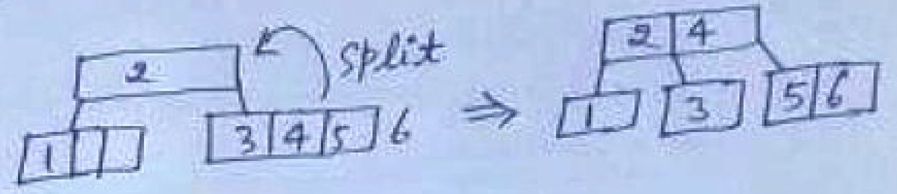
Step-4 Insert-4



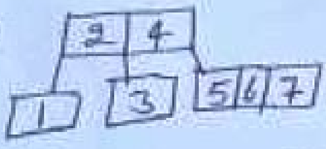
Step-5 Insert-5



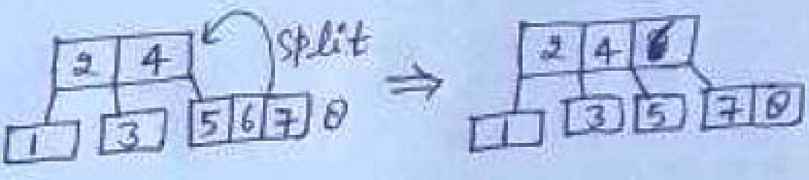
Step-6 Insert-6



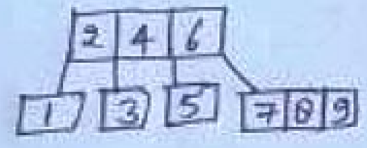
Step-7 Insert-7



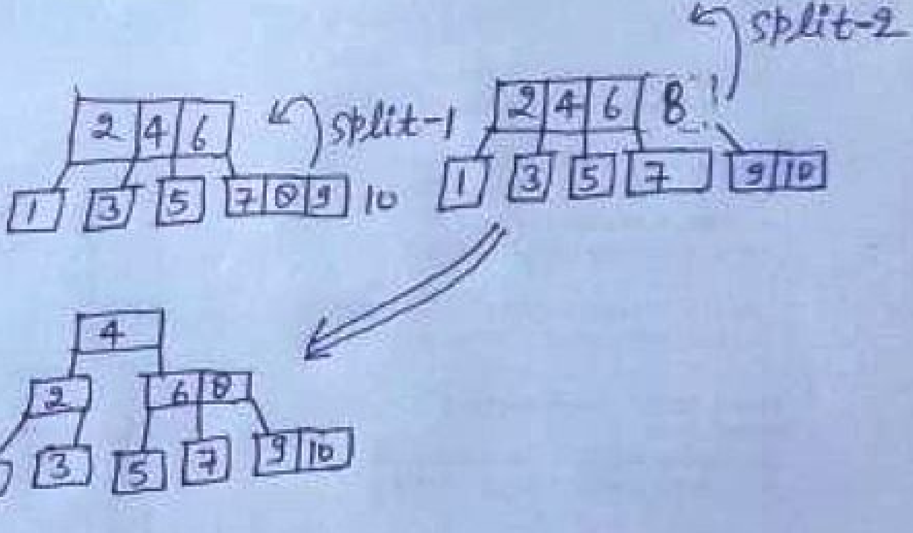
Step-8 Insert-8



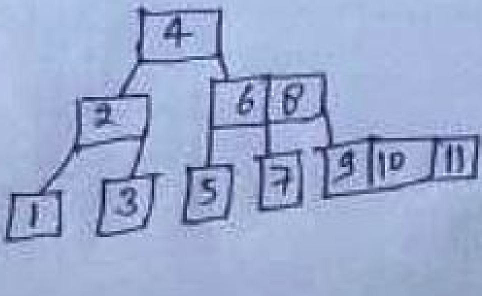
Step-9 Insert-9



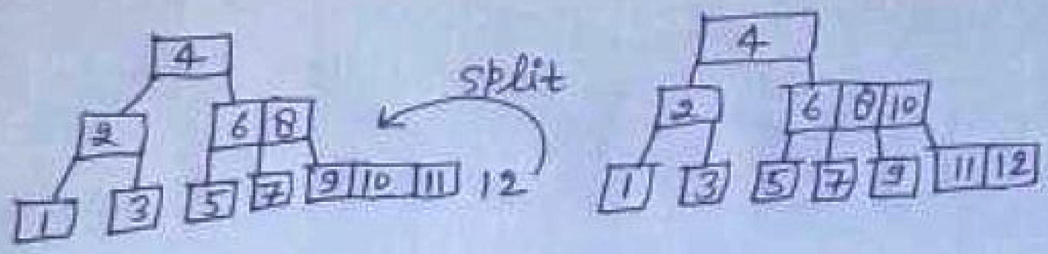
Step-10 Insert-10



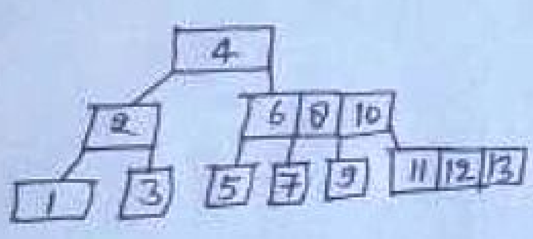
Step-11 Insert-11



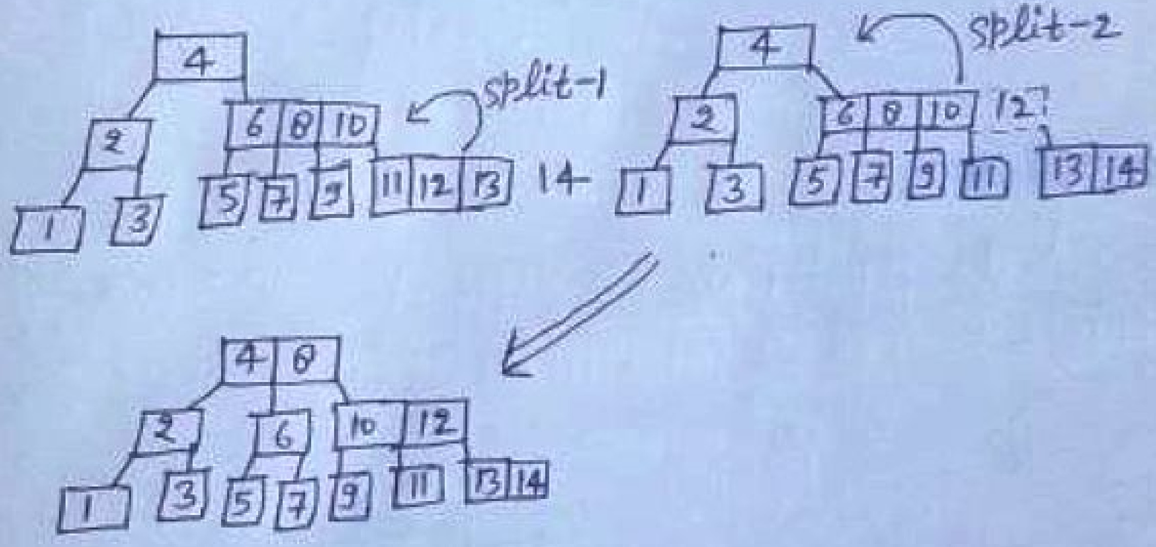
Step-12 Insert-12



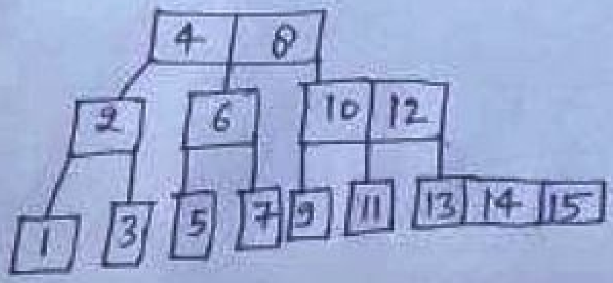
Step-13 Insert-13



Step-14 Insert-14



Step-15 Insert-15



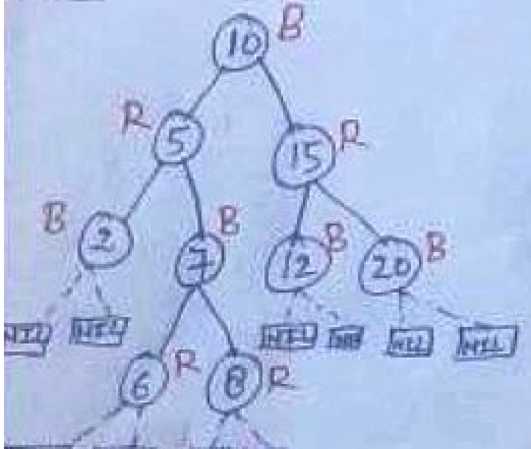
Red-Black Trees

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensures that no such path is more than twice as long as any other, so that the tree is approximately balanced.

A red-black tree is a binary tree that satisfies the following red-black properties: → Red-Black tree is a self-balancing BST.

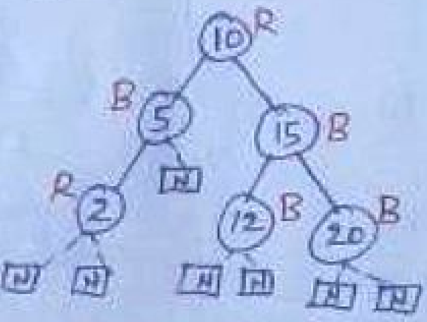
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

EX-1



R-B Tree (✓)

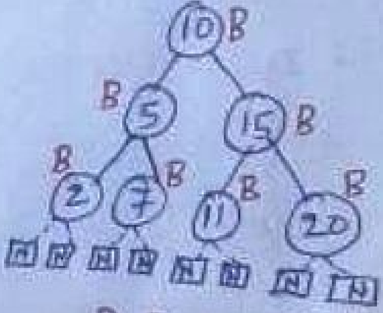
EX-2



R-B Tree (X)

(Violate Property 2)

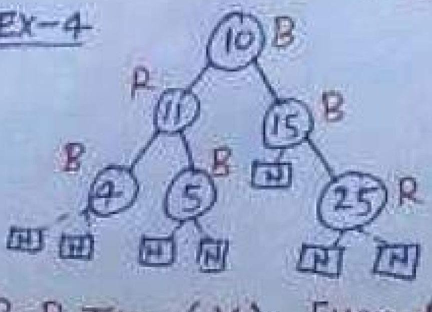
EX-3



R-B Tree (✓)

Even not a single red color node in a tree.

EX-4



R-B Tree (X) Even though it follows all the properties of R-B tree but it's not a BST.

Insertion in Red-Black Tree

Algorithm:

① If tree is empty, [redacted] create new node as root node with color Black.

② If tree is not empty, create newnode as leaf node with color Red.

③ If Parent of newnode is 'black' then exit.

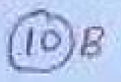
④ If Parent of newnode is 'red' then check the color of parents sibling of new node:
(or (uncle))

ⓐ If color is black or null then do suitable rotation & recolor.

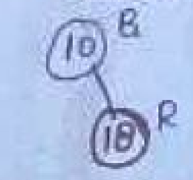
ⓑ If color is red then recolor both parent and sibling & also check if parent's parent of new node is not root node then recolor it & recheck.

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

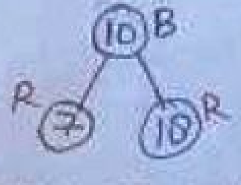
Step 1: Insert element 10



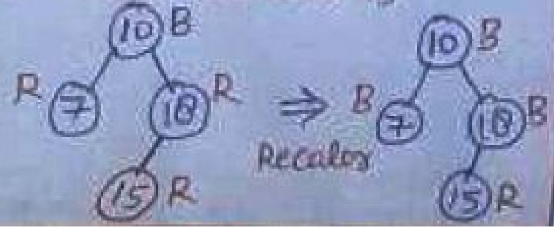
Step 2: Insert element 18



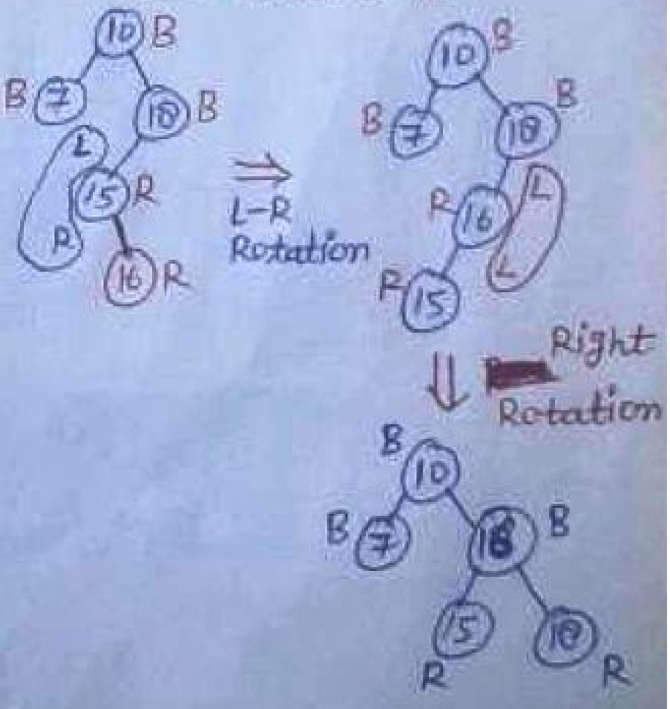
Step 3: Insert element 7



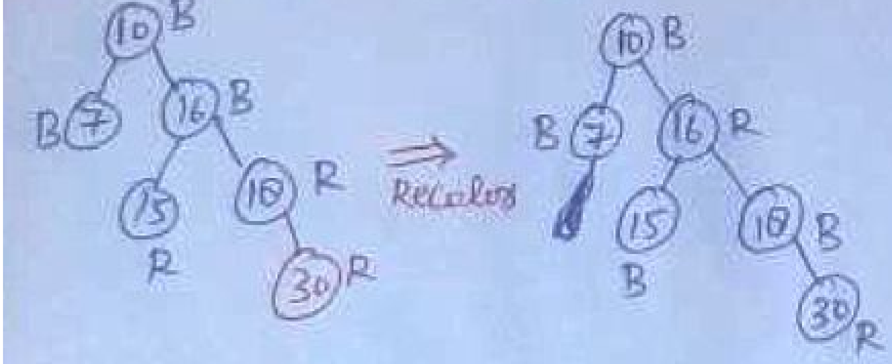
Step 4: Insert element 15



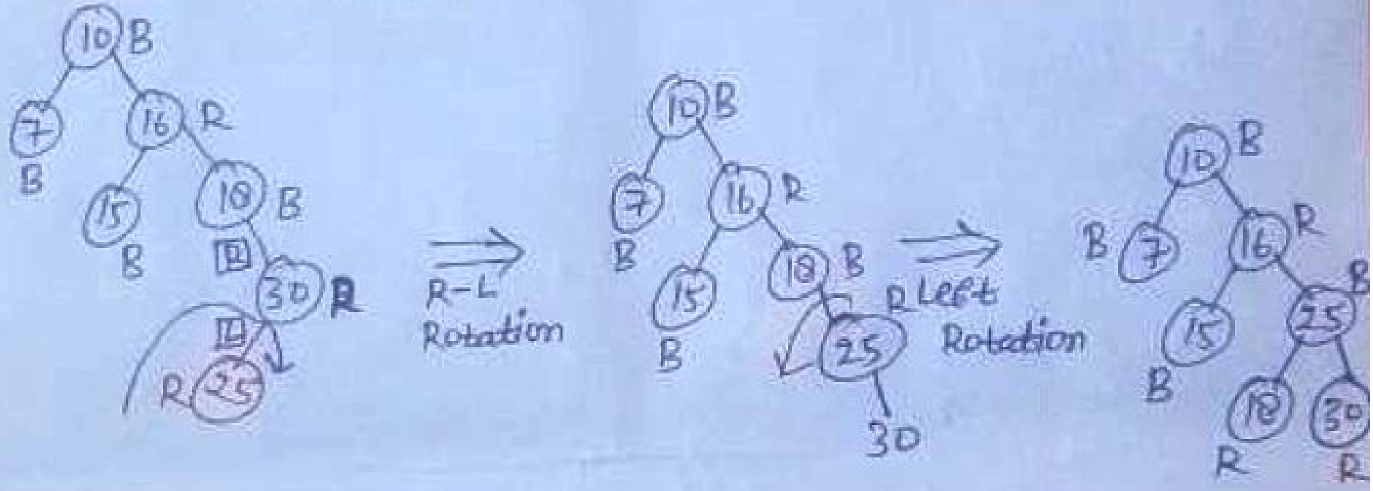
Step 5: Insert element 16



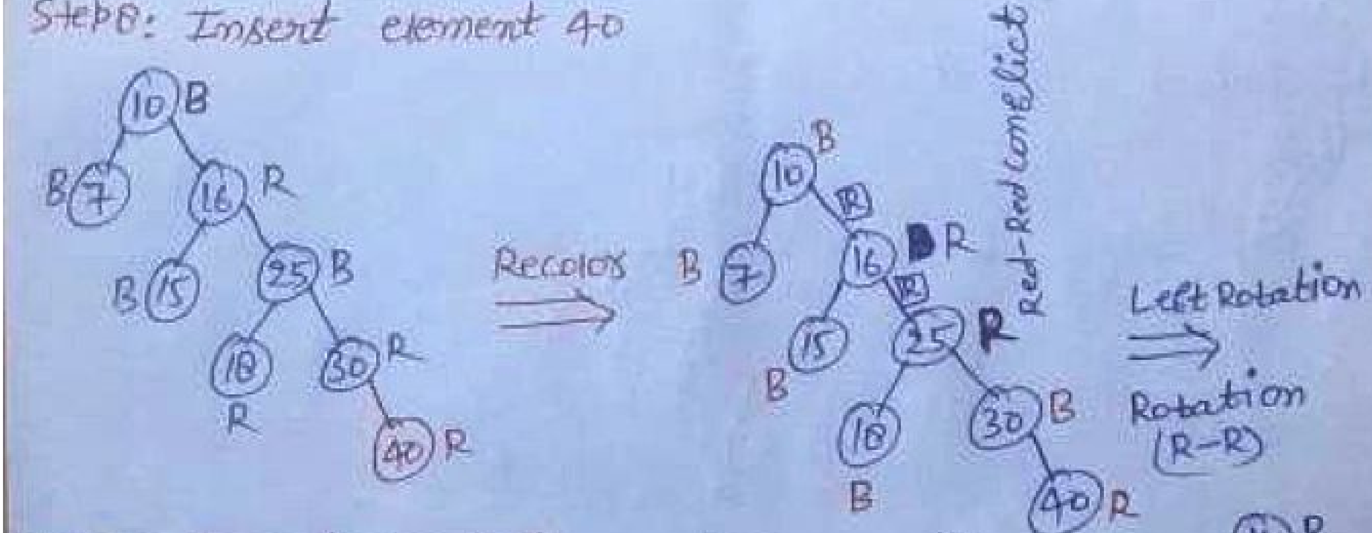
Step 6: Insert element 30



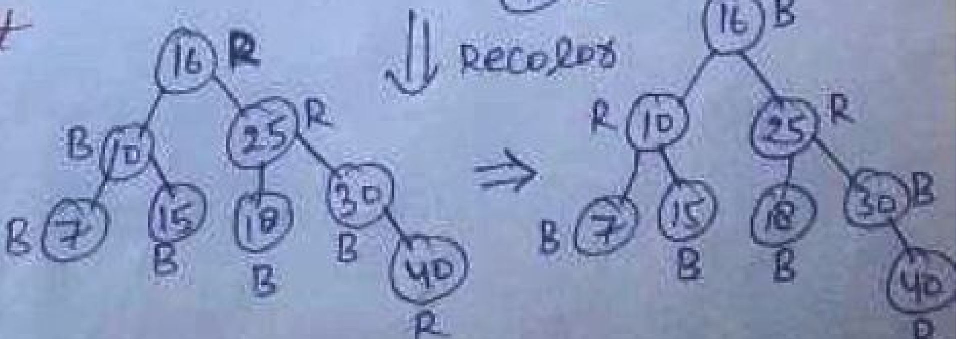
Step 7: Insert element 25



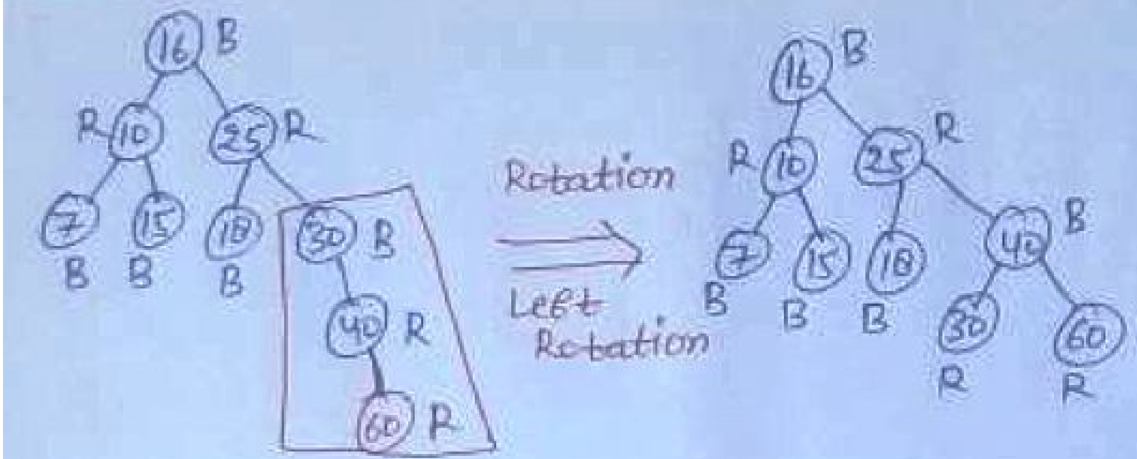
Step 8: Insert element 40



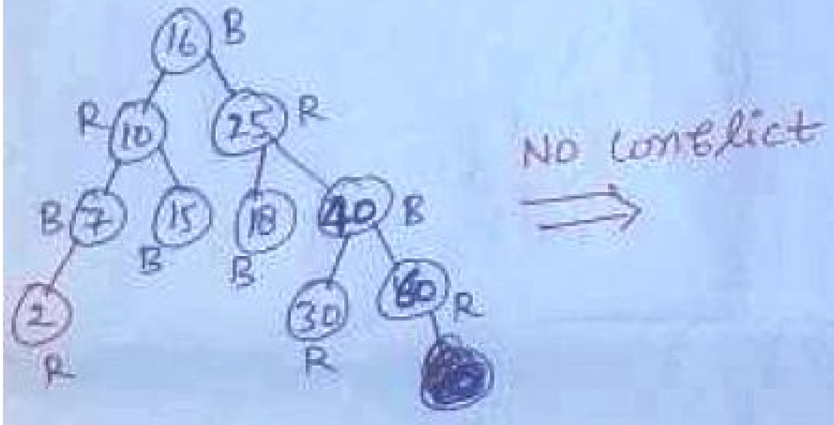
~~Step 9: Insert element~~



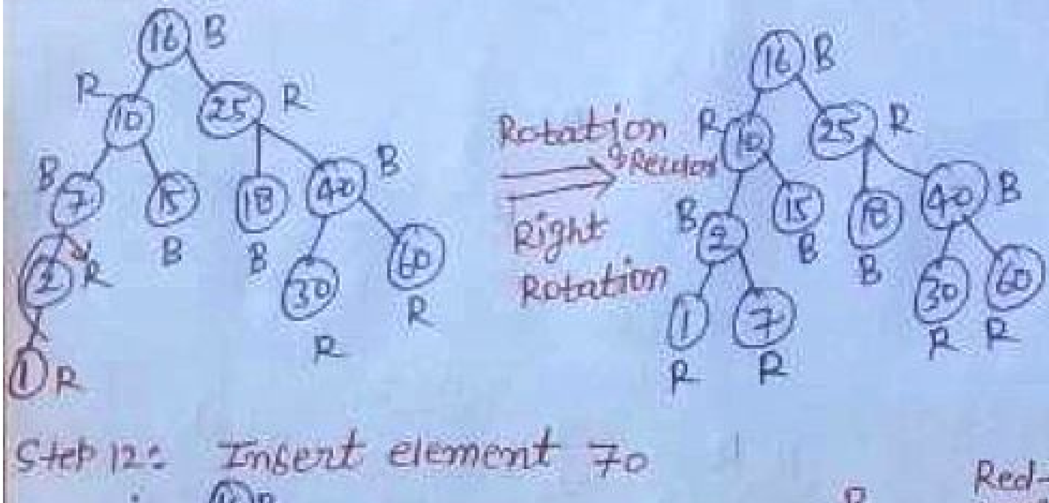
Step 9: Insert element 60



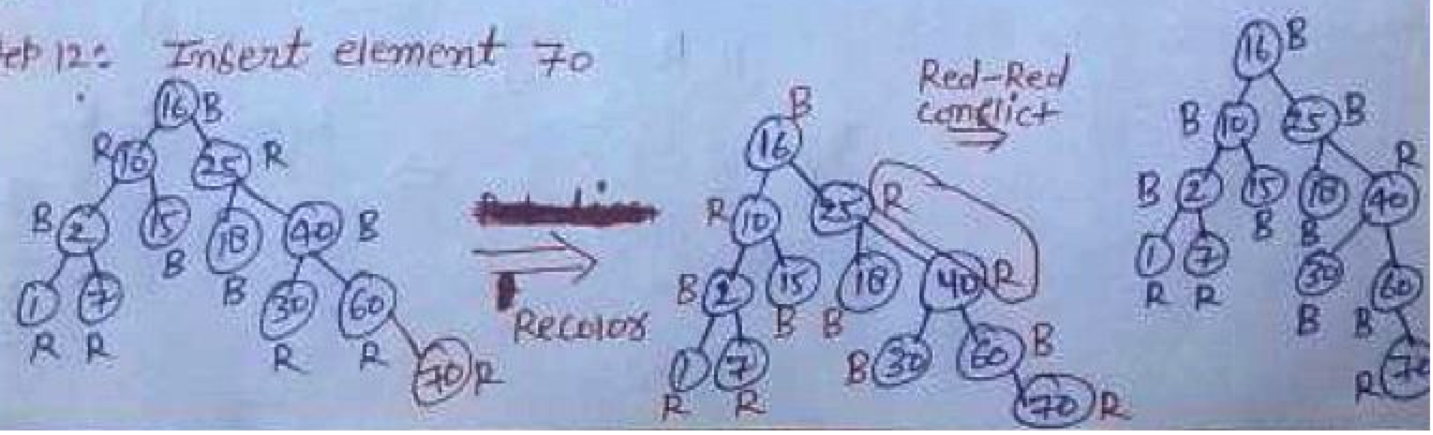
Step 10: Insert element 2



Step 11: Insert element 1



Step 12: Insert element 70



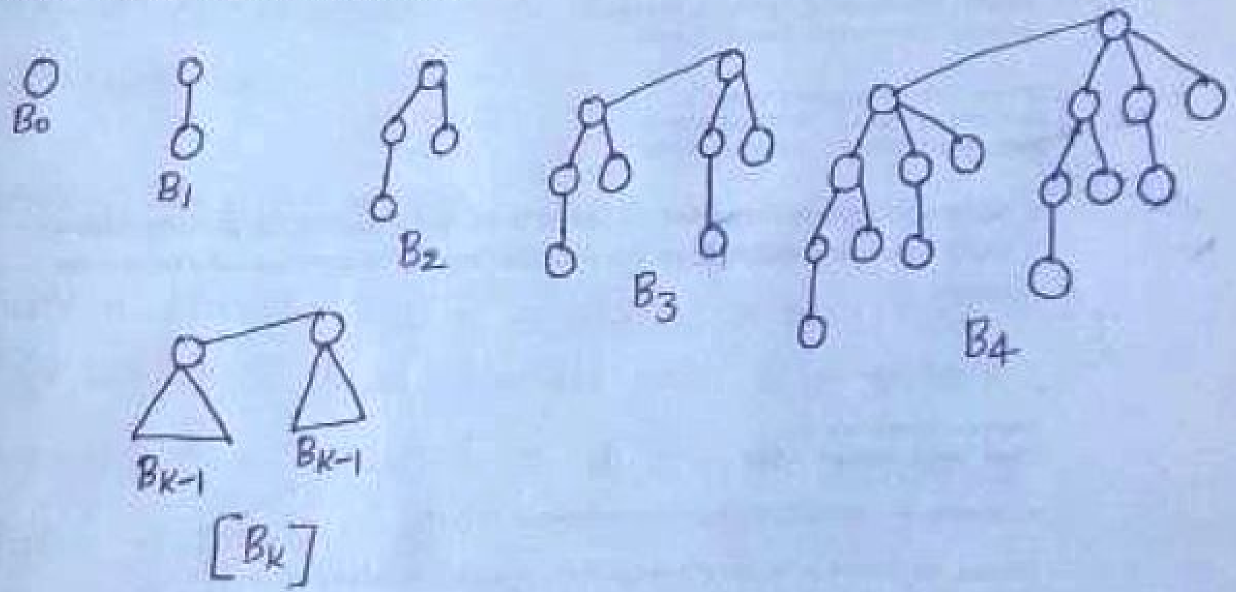
Binomial Heap

→ Binomial Heap is a collection of Binomial Tree.

→ The Binomial Tree B_K is an ordered tree defined recursively.

→ The Binomial Tree B_0 consist of a single node.

→ The Binomial Tree B_K consist of two Binomial Tree B_{K-1} that are linked together.



Properties of Binomial Tree (B_K)

1. There are 2^K nodes.
2. The height of the tree is K.
3. There are exactly $_KC_i$ nodes at depth i for $i = 0, 1, 2 --- K$.
4. The root has degree K, which is greater than that of any other node.
5. If i the children of the root are numbered from left to right by K−1, K−2, ---- 0 child i is the root of a subtree B_i.

Binomial Tree :- Binomial Tree B_K is an ordered tree defined recursively.

- The Binomial tree B_0 consist node.
- The Binomial Tree B_K consist two Binomial Tree B_{K-1} and B_{K-1} are linked together.

Properties of Binomial Heap

→ No two binomial trees in the collection have the same size.

→ Each node in the collection has a key.

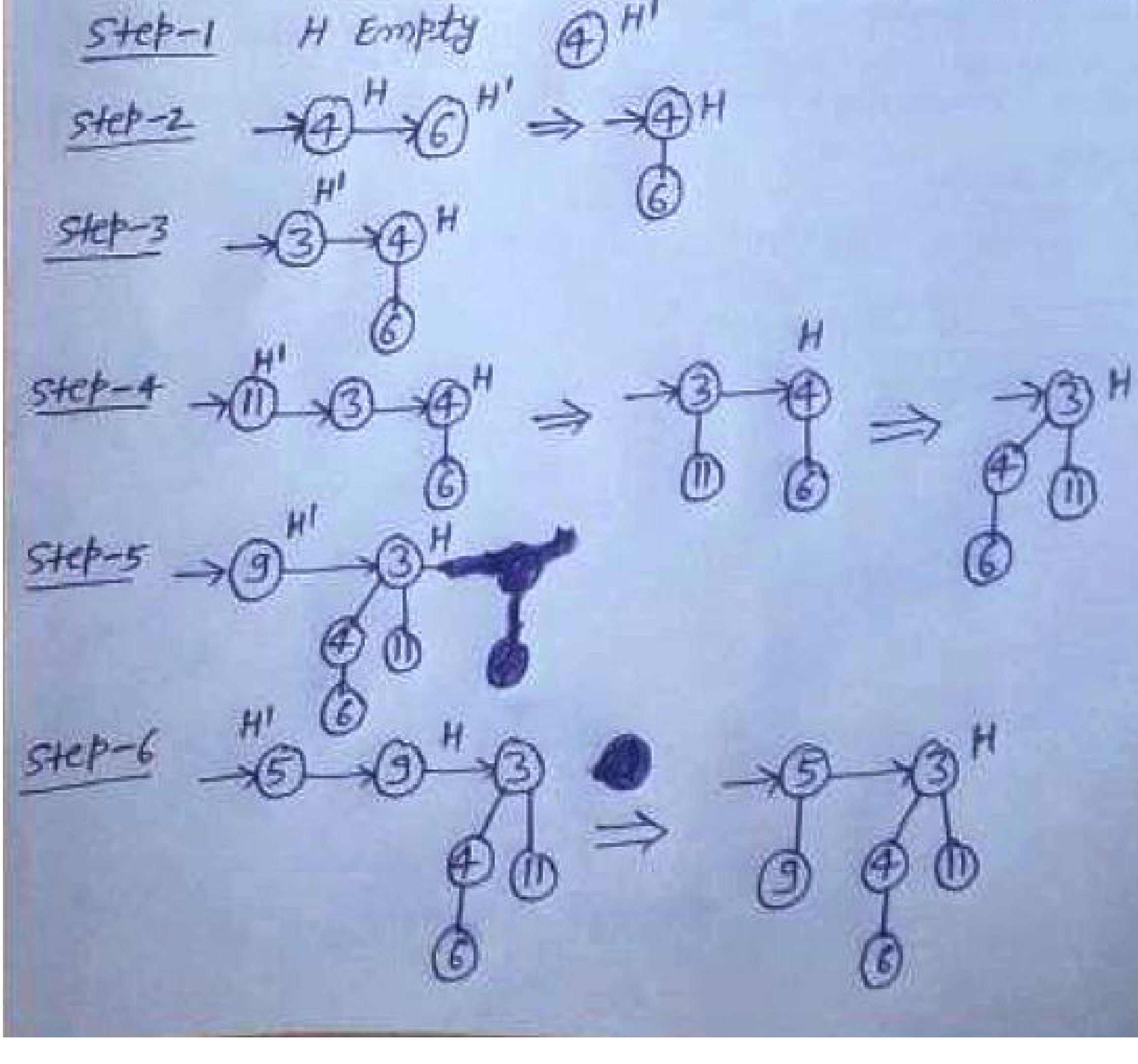
→ Each binomial tree in the collection satisfies the heap order property.

→ Roots of the binomial trees are connected and are in increasing order.

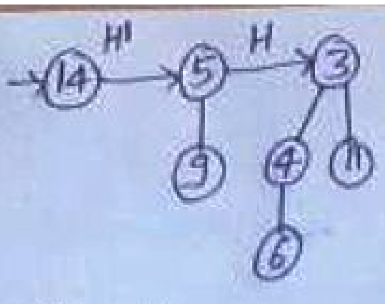
→ **Binomial Heap creation:**

① Create a binomial heap H' containing new element.
② Apply union of two binomial min heap H and H'.

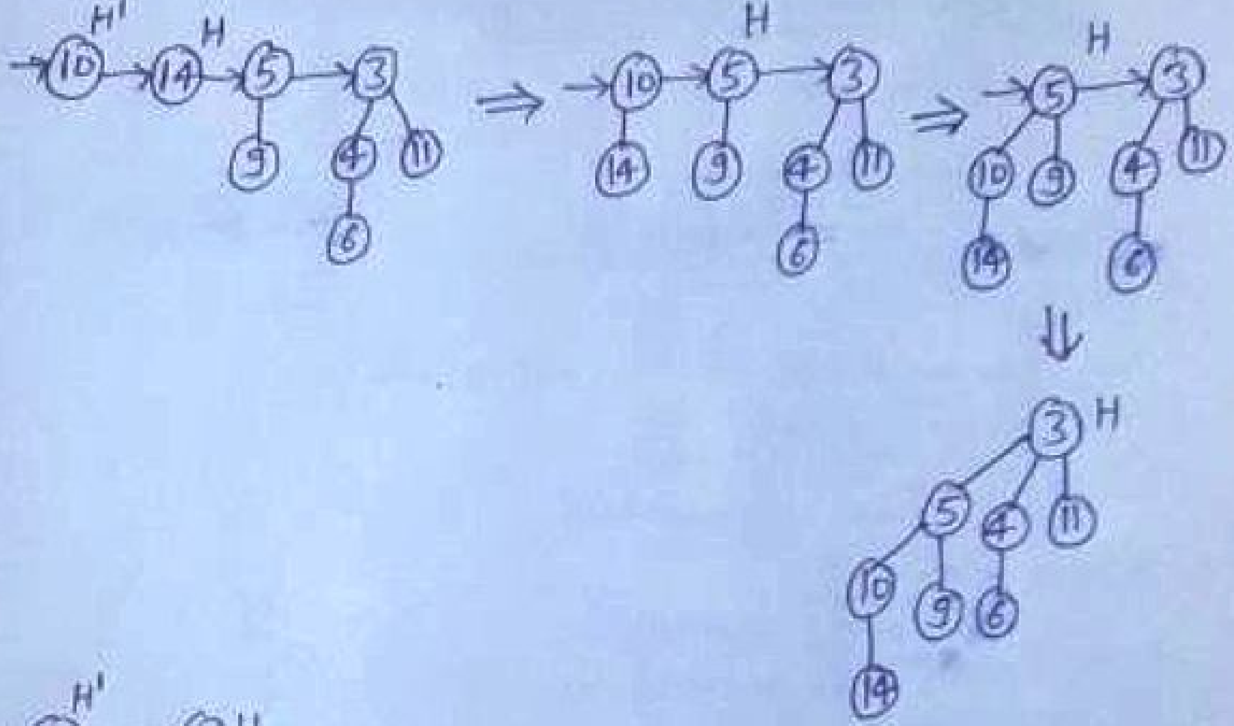
Given array: 4, 6, 3, 11, 9, 5, 14, 10, 21, 7, 13, 20, 2



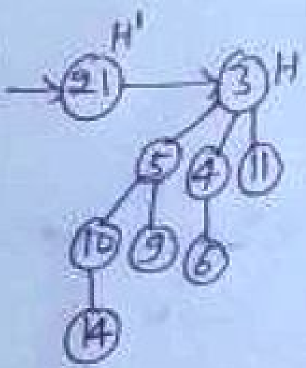
Step-7



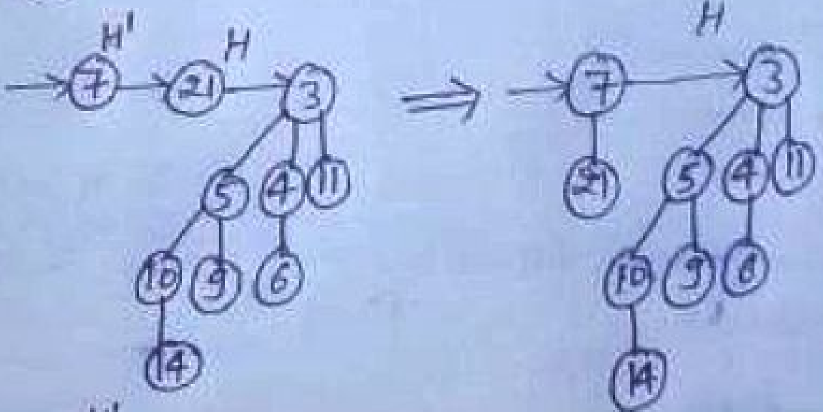
Step-8



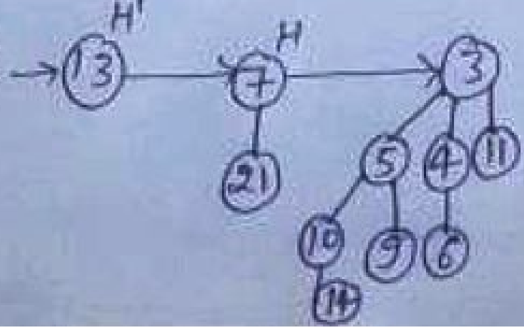
Step-9



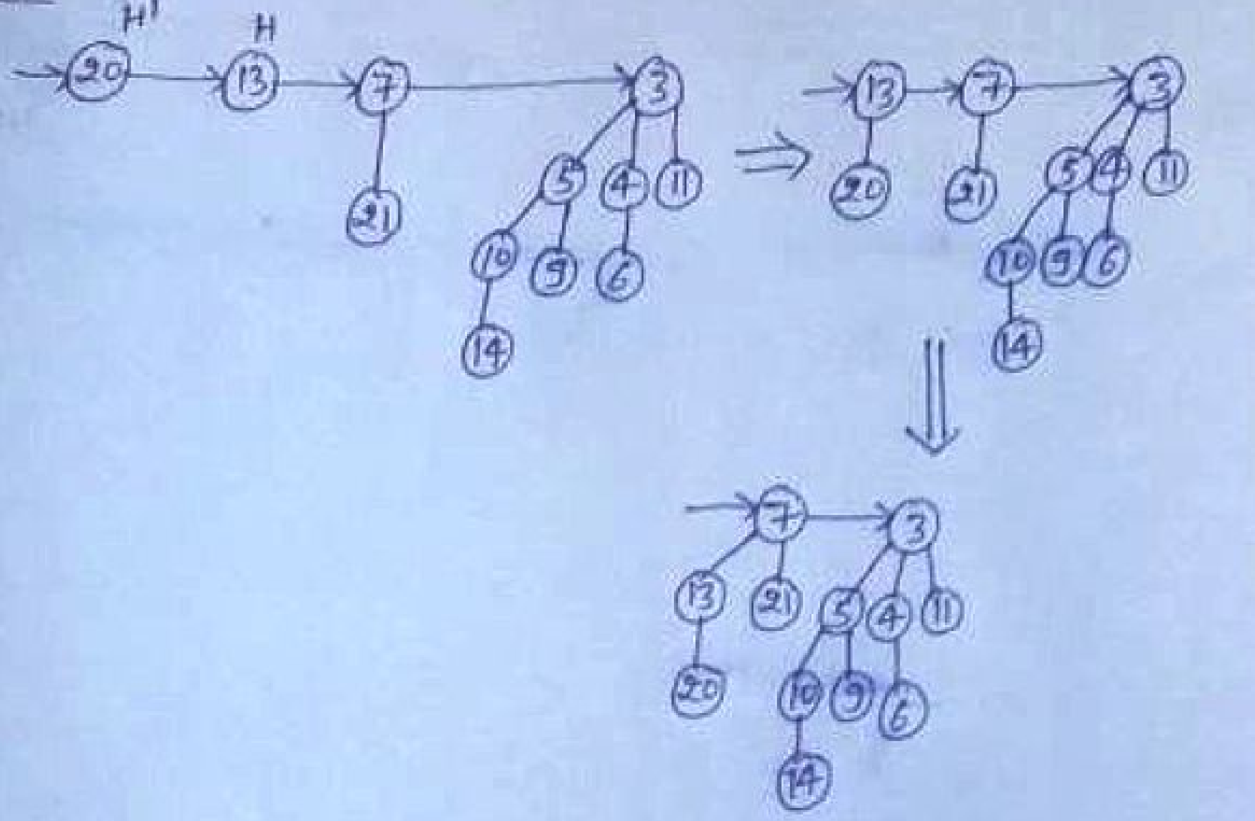
Step-10



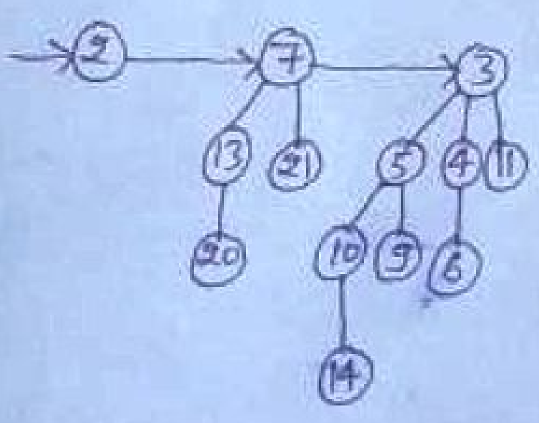
Step-11



Step-12



Step-13

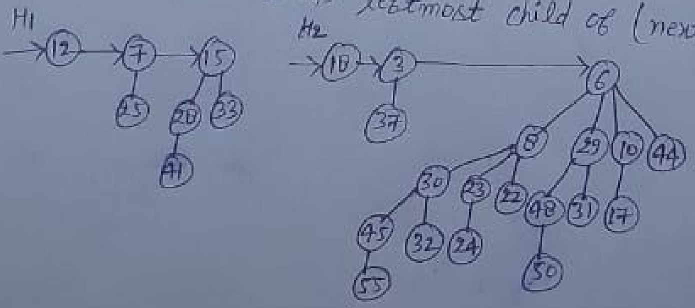


Binomial min Heap UNION

① Merge the root lists of binomial min heaps H_1 and H_2 into a single linked list that is stored in non decreasing order of their degree.

② Link the roots of equal degree until atmost one root remains of each degree.

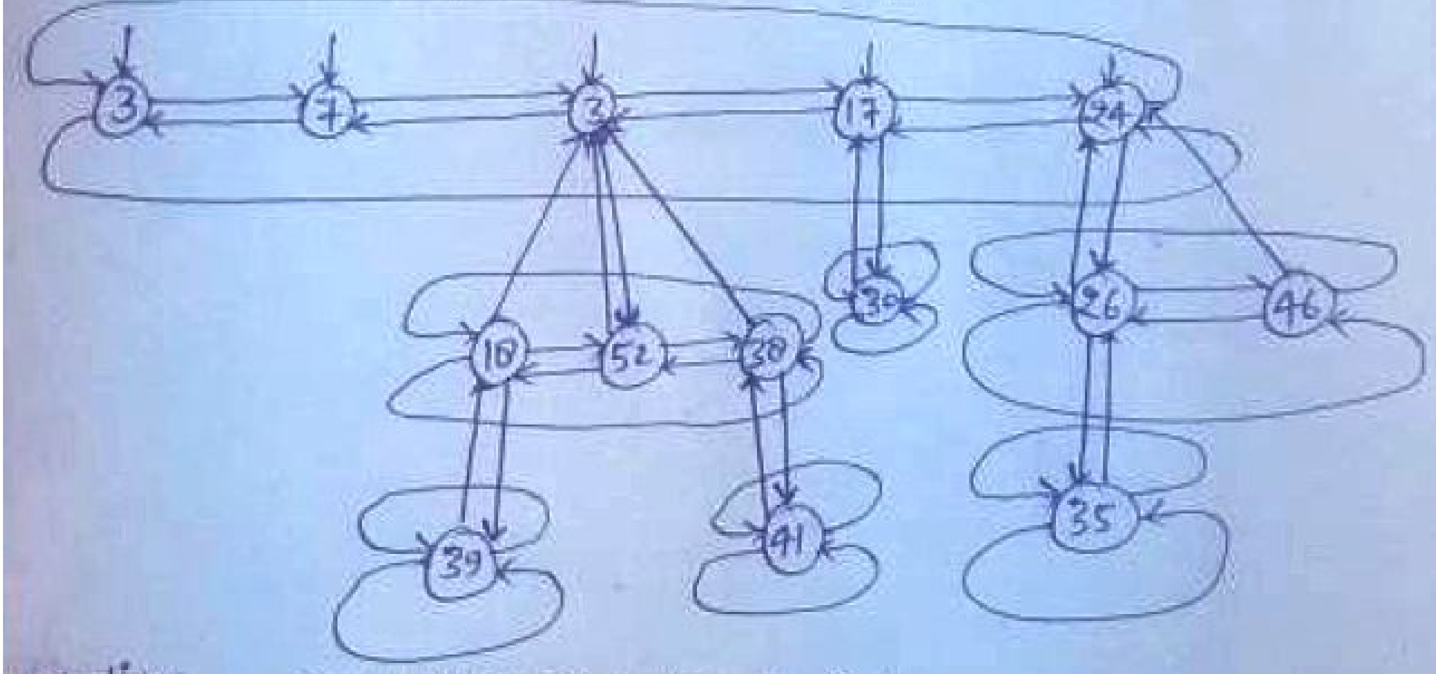
```
ⓐ if ( degree[x] ≠ degree[next_x] or
       degree[x] = degree[next_x] = degree[sibling[next_x]])
       then move the pointers one position right.

ⓑ if ( degree[x] = degree[next_x] ≠ degree[sibling[next_x]])
   (i) if ( key(x) < key(next_x))
        then make (next_x) as leftmost child of x.
   (ii) if( key(x) > key(next_x))
        then make x as leftmost child of (next_x).
```



Fibonacci Heaps

- → Collection of trees with each tree following the heap ordering (min heap) property.
- → Trees may be in any order in the root list. (Unlike Binomial Heaps)
- → A pointer to the min element of the heap always maintained.
- → Siblings are connected through a circular doubly linked list.
- → Each child points to its parent.
- → Each parent points to any one child.
- → degree(x): No. of children of root of a tree.
- → Mark(x): 1 – Lost one of its child
  0 – Lost no child

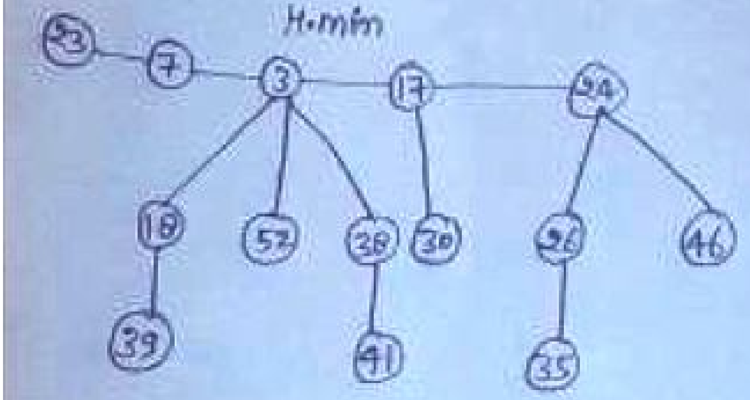
Structure of Fibonacci Heap



Notation:

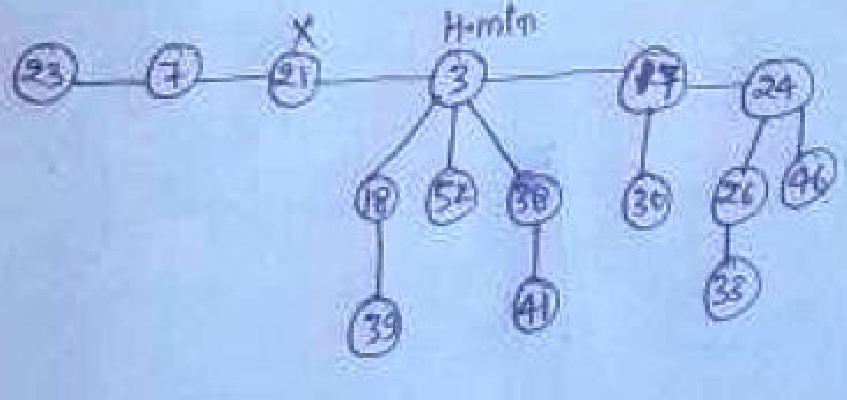
n: Number of nodes in heap.
rank(x): Number of children of node x.
rank(H): Max rank of any node in heap H.
trees(H): Number of trees in heap H.
marks(H): Number of marked nodes in heap H.

Insertion of key



```
Fib-Heap-Insert (H,x)
1. x.degree = 0
2. x.P = NIL
3. x.child = NIL
4. x.mark = FALSE
5. If H.min == NIL
6. create a root list for H
   containing just x.
7. H.min = x
8. else insert x into H's root list
9. If x.key < H.min.key
10.  H.min = x.
11.  H.n = H.n + 1
```

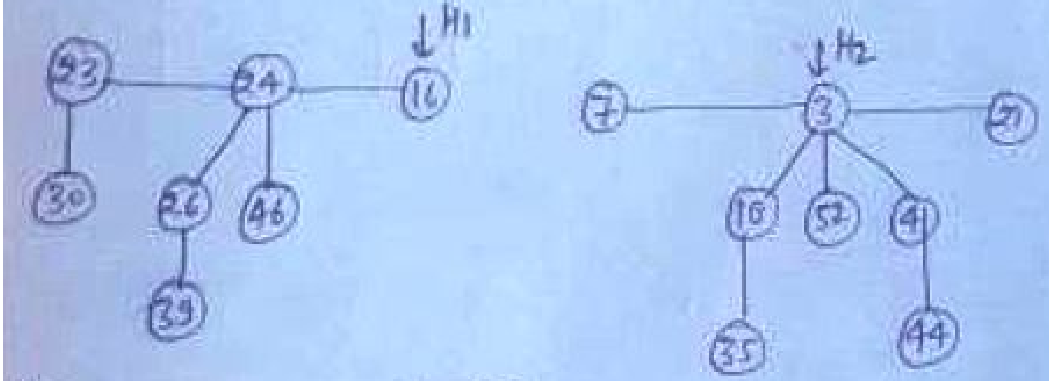
X.degree	0
X.P	NULL
X	X
X.mark	FALSE
X.child	NULL



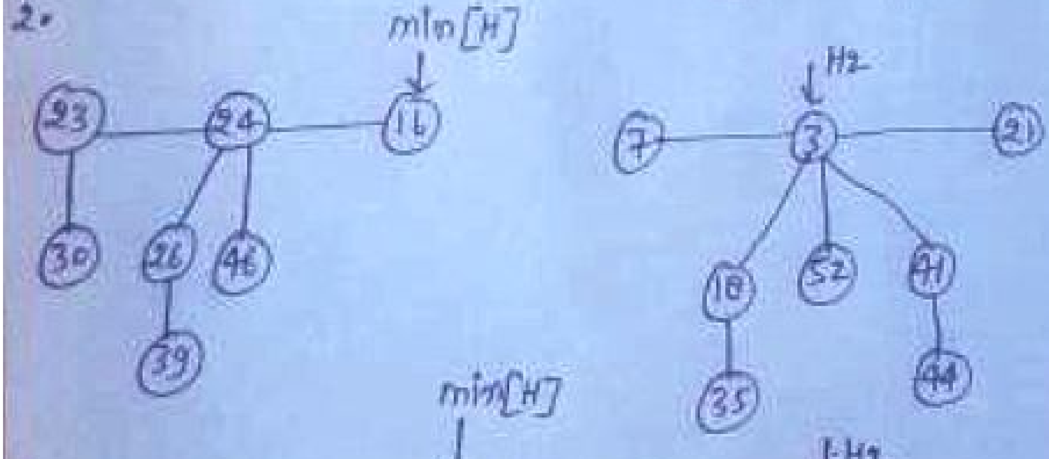
$\boxed{TC = O(1)}$

UNION of Fibonacci Heaps

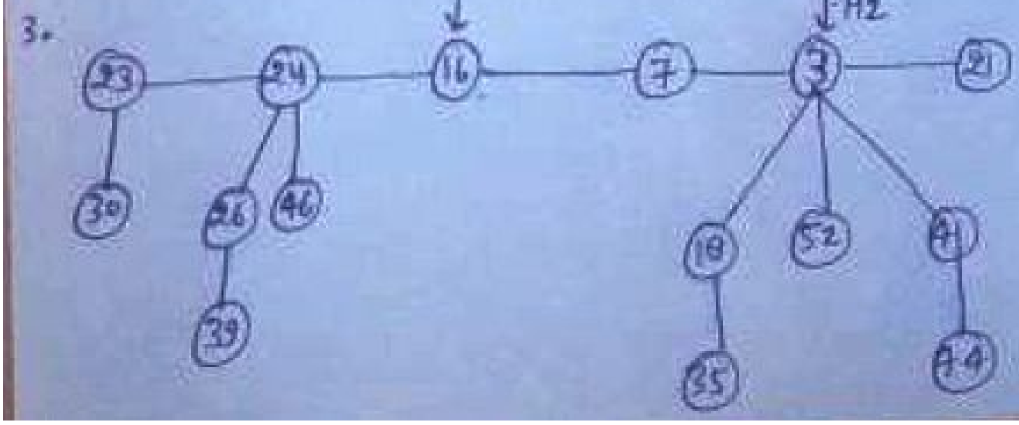
```
Fib_Heap_UNION(H1, H2)
1. H ← Make_Fib_Heap()
2. min[H] ← min[H1]
3. concatenate the root list of H2 with
   the root list of H
4. If (min[H1] = NIL) or
      (min[H2] ≠ NIL and min[H2] < min[H1])
5.   then min[H] ← min[H2]
6. n[H] ← n[H1] + n[H2]
7. free the objects H1 and H2
8. return H
```



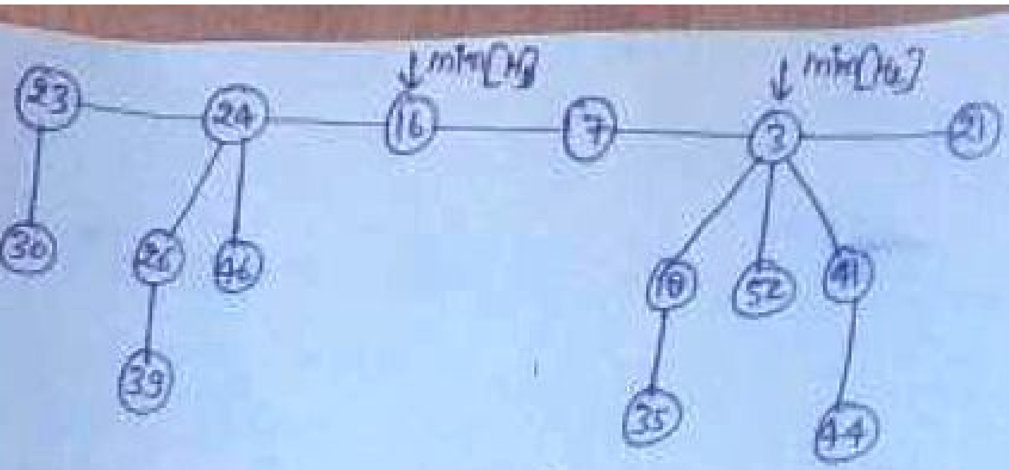
2.



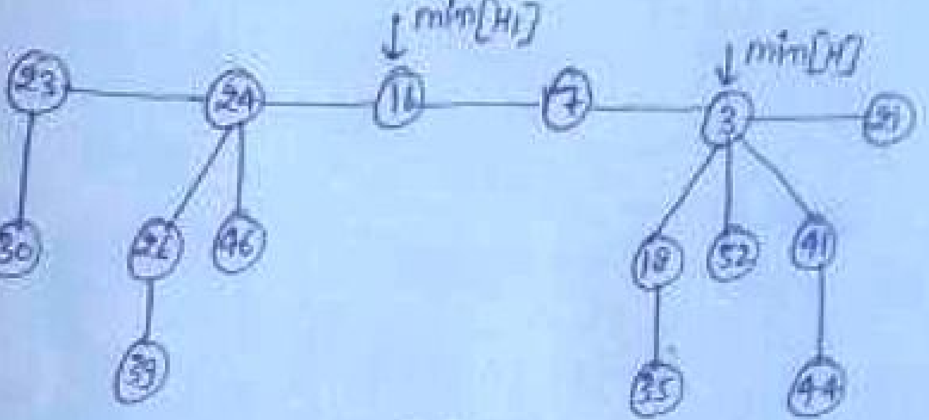
3.



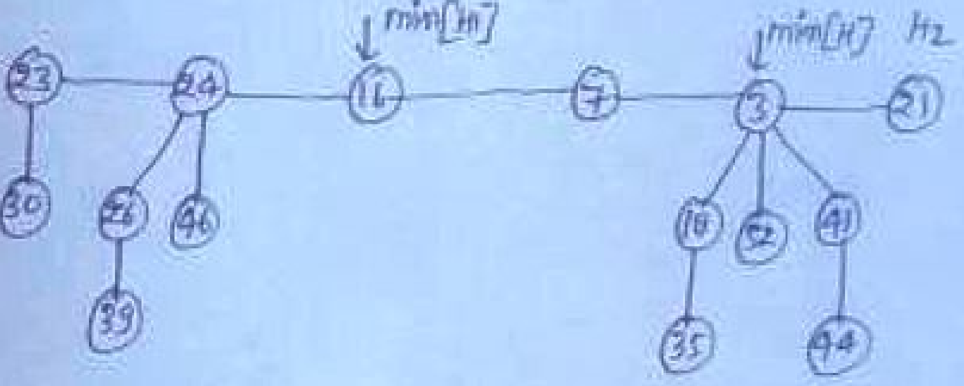
.



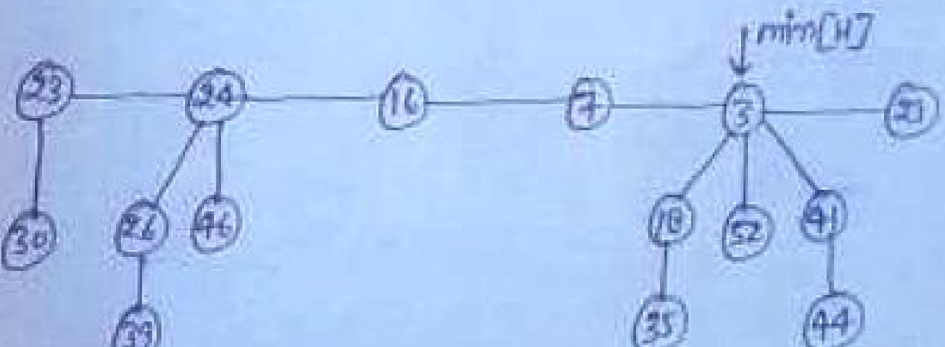
5.



6.



7.



$\boxed{TC = O(1)}$

Applications

1. priority Queue Implementation.
2. Large amount of data reperesentation.
3. Decrease key operation is used in minimum spanning tree algorithm.
4. single source shortest path.