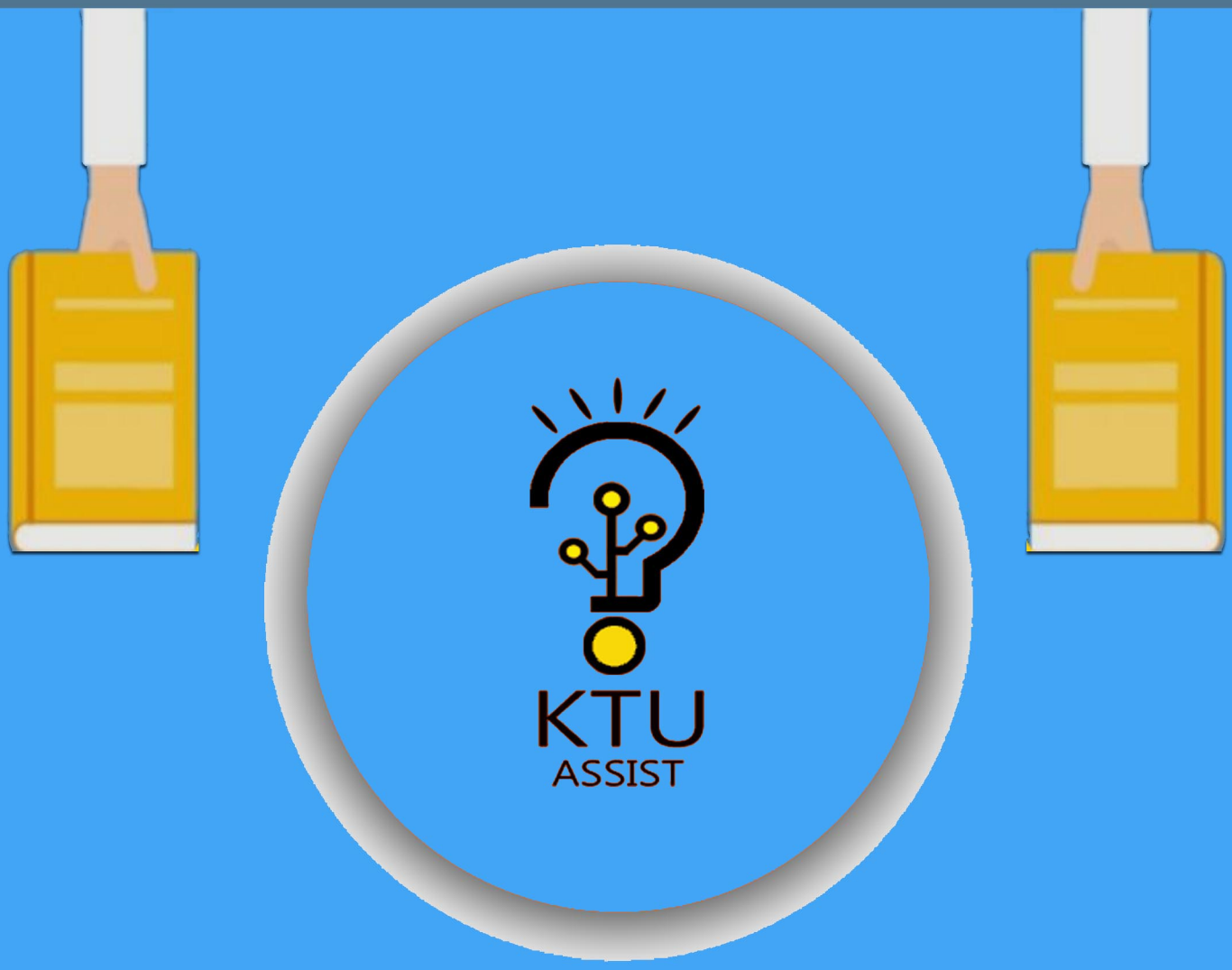
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



# Module 2

# Asymptotic Notations & AVL Trees

## ASYMPTOTIC NOTATIONS

**Ques 1) Explain master theorem with example.**

**Ans: Master's Theorem**

Let $a \geq 1$ and $b > 1$ be constants, let f(n) be a function, and let T(n) be de nonnegative on the nonnegative integers by the recurrence

$T(n) = aT(n/b) + f(n)$

where one interpret n/b as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then T(n) can be bounded asymptotically as follows:

**Case1**: If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

**Case2:** If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

**Case3:** If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = \Theta(f(n))$.

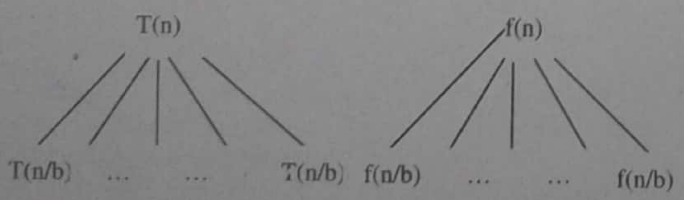
Consider $T(n) = aT(n/b) + f(n)$.
Suppose f(n) is small enough
Say $f(n) = 0$, ie. $T(n) = aT(n/b)$.

Then we have a recursion tree where the only contribution is at the leaves. There will be $\log_b n$ levels, with $a^l$ leaves at level l.

$$T(n) = a^{\log_b n} = n^{\log_b a}$$

so long as f(n) is small enough that, we have case 1 of the Master Theorem.

Suppose f(n) is large enough. If we draw the recursion tree for $T(n) = aT(n/b) + f(n)$.




If f(n) is a big enough function, the one top call can be bigger than the sum of all the little calls.

**For example, let consider the following** recurrence:

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

Here $a = 9, b = 3, f(n) = n$

$$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$$

writing f(n) in appropriate form.

$$f(n) = O(n^{\log_3 9 - 1}) = O(n)$$

so the solution is $\Theta(n^{\log_b a})$, i.e.,

$$\Theta(n^{\log_3 9}) = \Theta(n^2)$$

**Ques 2) Solve the recurrence by Master method.**

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

**Ans:** Here $a = 1, b = 3/2, f(n) = 1$

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

which is equal to f(n). Thus solution is,

$$T(n) = \Theta(n^{\log_b a} . \lg n) = \Theta(1 \lg n) = \Theta(\lg n)$$

**Ques 3) Solve the recurrence by Master Method.**

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

**Ans:** Here, $a = 3, b = 4, f(n) = n \lg n$

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

$$f(n) = \Omega(n^{(\log_4 3 + \varepsilon)}),$$

where $\varepsilon \approx 0.2$,

Here we can see that regularity condition

i.e., $af\left(\frac{n}{b}\right) \leq cf(n)$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq \left(\frac{3}{4}\right) n \lg n = cf(n) \text{ for } c = \frac{3}{4}$$

Therefore, solution to the recurrence is

$$T(n) = \Theta(n \lg n) = \Theta(f(n)).$$

**Ques 4) Solve the following recurrence:**

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

**Ans:** By Master method,

$a = 4, b = 2, f(n) = n^3$

$$n^{\log_b a} = n^2 = O(n^2)$$

By case 3, $f(n)=\Omega(n^{\log_b a+\varepsilon})=\Omega(n^{2+1}), \varepsilon = 1$

thus case 3 can be applied, now check for the regularity condition $4f\left(\frac{n^3}{2^3}\right) \le cf(n^3)$, this is true for $c=\frac{1}{2}$

hence solution to this recurrence is $\Theta(f(n))$,

i.e., $T(n)=\Theta(n^3)$

**Ques 5) What are the various asymptotic notations?**
**Or**
**What is difference between O-Notation (Big-oh) and o-Notation (Little-oh)?**

**Ans: Asymptotic Notations and its Properties/ Notations of Space and Time Complexity**

Asymptotic notation is a shorthand way to write down. Using asymptotic notations we can give time complexity as 'fastest possible' and 'slowest possible' running times for the algorithm, using high and low bounds on speed.

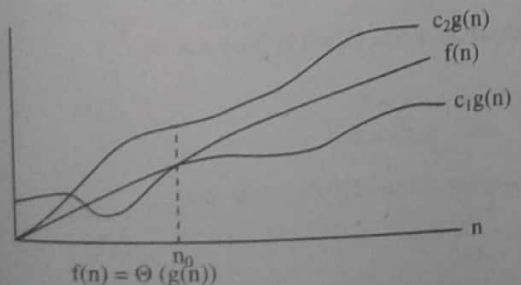
Asymptotic notation deals with the behavior of functions in the limit that is for sufficiently large values of its parameter. Asymptotic means a line that tends to converge to a curve, which may or may not eventually touch the curve. It is a line that stays within bounds.

Asymptotic notation is used to describe the running time of an algorithm. This shows the order of growth of function. Asymptotic notation describes the algorithm efficiency and performance in a meaningful way.

Asymptotic notation also describes the behavior of time or space complexity for large instance characteristics. The asymptotic running time of an algorithm is defined in terms of functions. The domains of these functions are set of natural numbers and real numbers.

The most prominent of notations are given below:

1) **Theta Θ-Notation (Average Bound):** This notation bounds a function to within constant factors. We say $f(n) = \Theta(g(n))$ if there exist positive constants $n_0$, $c_1$ and $c_2$ such that to the right of $n_0$ the value of f(n) always lies between $c_1$ g(n) and $c_2$ g(n) inclusive.



f(n) = Θ (g(n))

$\Theta(g(n)) = \{f(n)$ : there exist positive constants $c_1$, $c_2$ and $N_o$ such that $0 <= c_1, g(n) < = f(n) <= c_2 g(n)$ for all $n >= N_0\}$.

We say that g(n) is an asymptotically tight bound of f(n) and write $f(n) = \Theta(g(n))$.

2) **Big-oh (O)-Notation (Upper Bound):** This notation gives an upper bound for a function to within a constant factor. O(g(n)) is pronounced "big-oh of g of n" or sometimes just "big of g of n"

It is considered as the longest amount of time taken by algorithm.

3) **Little – oh (o) Notation:** Little oh can be defined as follows:
$o(g(n)) = \{ f(n)$ : for any positive constant $c > 0$, there exist a constant $N_0 > 0$ such that $0 <= f(n) < cg(n)$ for all $n >= N_0\}$.

One say that g(n) is an upper bound of f(n) but not asymptotically tight, and write $f(n) = o(g(n))$

**Difference between O-Notation (Big-oh) and o-Notation (Little-oh)**

The definitions of O-notation and o-notation are similar.

The main difference is that in $f(n)= O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for "some constant $c > 0$, but in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ hold for "all constants $c > 0$".

In o-notation, the function f(n) becomes insignificant relative to g(n) as $n \to \infty$

i.e., $\lim_{n\to\infty} \frac{f(n)}{g(n)}=0$

4) **Big-Omega (Ω) Notation (Lower Bound):** Ω-notation provides an asymptotic lower bound. For a given function g(n), we denoted by Ω (g(n)) (pronounced "big-omega of g of n" or sometimes just "omega of g of n")

$\Omega(g(n)) = \{f(n)$ : there exist positive constants c and $n_0$ such that $0 \le c(g(n)) \le f(n)$ for all $n \ge n_0\}$

5) **Little-Omega (ω) Notation:** Little-Omega for any positive constant $c > 0$, there exist a constant $N_0 > 0$ such that $0 <= cg(n) < f(n)$ for all $n >= N_0\}$.

It is considered as the shortest amount of time taken by algorithm. One say that g(n) is a lower bound of f(n) but not asymptotically tight and write:

$f(n) = \text{omega}(g(n) = \omega(g(n))$

**Ques 6) What are the main properties of asymptotic notations?**

**Ans:** The main properties of asymptotic notations are as follows:

1) **Transitivity**

$f(n)= \Theta(g(n))$ and $g(n)=\Theta(h(n))$ imply $f(n)= \Theta(h(n))$,
$f(n)= O(g(n))$ and $g(n)= O(h(n))$ imply $f(n)= O(h(n))$,
$f(n)=\Omega(g(n))$ and $g(n)= \Omega(h(n))$ imply $f(n)=\Omega(h(n))$,
$f(n)= o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$,
$f(n)= \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n)= \omega(h(n))$,

2) **Reflexivity**
$f(n) = \Theta(f(n))$,
$f(n) = O(f(n))$,
$f(n) = \Omega(f(n))$,

3) **Symmetry**
$f(n) = (g(n))$ if and only if $g(n) = \Theta(f(n))$.

4) **Transpose Symmetry**
$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$,
$f(n) = O(g(n))$ if and only if $g(n) = \omega(f(n))$.

Because these properties hold for asymptotic notations, one can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

| | | |
|---|---|---|
| $f(n) = O(g(n))$ | $\approx$ | $a \le b$, |
| $f(n) = \Omega(g(n))$ | $\approx$ | $a \ge b$, |
| $f(n) = \Theta(g(n))$ | $\approx$ | $a = b$, |
| $f(n) = o(g(n))$ | $\approx$ | $a < b$, |
| $f(n) = \omega(g(n))$ | $\approx$ | $a > b$. |

One say that f(n) is asymptotically smaller than g(n) if $f(n) = o(g(n))$, and f(n) is asymptotically larger than g(n) if $f(n) = \omega(g(n))$.

One property of real numbers, however, does not carry over to asymptotic notation:

5) **Trichotomy:** For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

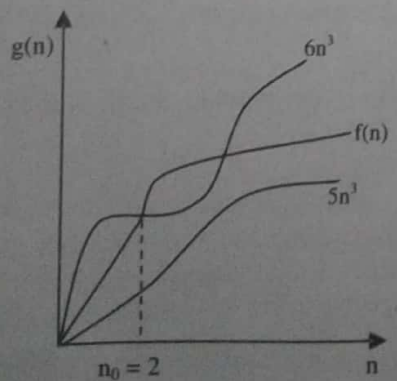
**Ques 7) Find Θ-notation for $5n^3 + n^2 + 3n + 2$.**

**Ans:** Let $f(n) = 5n^3 + n^2 + 3n + 2$
for finding Θ-notation we show the given function bounded by upper bound and lower bound by a function g(n). Therefore, our goal is to find g(n).

$$5n^3 < 5n^3 + n^2 + 3n + 2 \forall n \ge n_0$$

Compare this with $c_1 g(n) \le f(n)$
$\Rightarrow c_1 = 5, g(n) = n^3$
Again, $5n^3 + n^2 + 3n + 2 < 6n^3 \forall n \ge n_0 = 2$

Again compare it with $f(n) \le c_2 g(n)$
$\Rightarrow c_2 = 6$
$g(n) = n^3$
since, $f(n) = \Theta(g(n))$
$\Rightarrow f(n) = \Theta(n^3)$




**Ques 8) If $f(n) = 27n^2 + 16n + 25$ then find the Θ-notation for the function.**

**Ans:** Given that:
$$f(n) = 27n^2 + 16n + 25 \quad \ldots\ldots (1)$$

From the given function, we can say that the given polynomial is of order 2, i.e., m = 2.

Then by applying theorem which is:
$$f(n) = \Theta(n^m)$$
$$\Rightarrow \quad f(n) = \Theta(n^2)$$

**Theorem 2:** If f(n) and g(n) be two functions such that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then $f \in \Theta g(n)$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

**Ques 9) By applying theorem 2 for Θ-notation for the given function $f(n) = 10n^2 + 7$.**

**Ans:** Given that
$$f(n) = 10n^2 + 7 \quad \ldots\ldots (1)$$

Assume that $g(n) = n^2$

then by applying theorem 2 as $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$

$$\Rightarrow \lim_{n\to\infty} \frac{10n^2+7}{n^2} \quad \Rightarrow \lim_{n\to\infty} \left(\frac{10n^2}{n^2} + \frac{7}{n^2}\right)$$

$$\lim_{n\to\infty} \left(10 + \frac{7}{n^2}\right) \quad \Rightarrow \left(10 + \frac{7}{\infty}\right)$$

$= 10$ (constant).

i.e., $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{10n^2+7}{n^2} = 10 = \text{constant}$
$\Rightarrow f(n) = \Theta(g(n))$
$\Rightarrow f(n) = \Theta(n^2)$

**Ques 10) If $3n + 5 = O(n^2)$ is $3n + 5 = o(n^2)$?**

**Ans:** Let $f(n) = 3n + 5$ .....(1)
and $g(n) = n$ .....(2)

For o-notation (little-oh) we show that:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad \ldots\ldots (3)$$

i.e., $\lim_{n\to\infty} \frac{3n+5}{n} = \lim_{n\to\infty} \left(\frac{3n}{n} + \frac{5}{n}\right) = \lim_{n\to\infty} \left(3 + \frac{5}{n}\right) = 3 \ne 0$

Hence, the condition given in equation (3) for o-notation (little-oh) is not satisfied.

This shows that:
$$3n + 5 \ne o(n^2)$$

**Ques 11) If $4n^3 + 2n + 3 = O(n^3)$ is $4n^3 + 2n + 3 = o(n^3)$?**

**Ans:** Let
$f(n) = 4n^3 + 2n + 3$
And $g(n) = n^3$ .....(1)
For o-notation (little-oh) we show that: .....(2)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad .....(3)$$

$$\Rightarrow \lim_{n\to\infty}\left(\frac{4n^3+2n+3}{n^3}\right) = \lim_{n\to\infty}\left(\frac{4n^3}{n^3}+\frac{2n}{n^3}+\frac{3}{n^3}\right)$$

$$= \lim_{n\to\infty}\left(4+\frac{2}{n^2}+\frac{3}{n^3}\right) = 4 \neq 0$$

Thus, condition for o-notation (little-oh) is not satisfied, therefore,
$4n^3 + 2n + 3 \neq o(n^3)$

**Ques 12)Find out Ω-notation for the function $4n^3 + 2n + 3$**

**Ans:** Let $f(n) = 4n^3 + 2n + 3$ .....(1)
For showing a function which is asymptotically bounded by Ω-notation, we have to show the lower bound of f(n) as
$cg(n) \leq f(n)$ .....(2)
$\Rightarrow 4n^3 \leq 4n^3 + 2n + 3 \ \forall n$ .....(3)

Comparing equation (3) with equation (2) we get,
$c = 4$
And $g(n) = n^3$
Since $f(n) = \Omega(g(n))$
$\Rightarrow f(n) = \Omega(n^3)$

**Ques 13) Find Ω-notation for $f(n) = 4 * 2^n + 3n$.**

**Ans:** Given that
$f(n) = 4 * 2^n + 3n$ ..... (1)

For Ω-notation, we show that function f(n) is asymptotically bounded by its lower bound as:
$cg(n) \leq f(n)$ ..... (2)
$\Rightarrow 4 * 2^n \leq 4 * 2^n + 3n \ \forall n$ ..... (3)

Compare equation (2) and equation (3) we get
$c = 4$
And $g(n) = 2^n$
$\Rightarrow f(n) = \Omega(cg(n))$
$\Rightarrow f(n) = \Omega(2^n)$

**Ques 14) Discuss about the Big-O notation with example.**

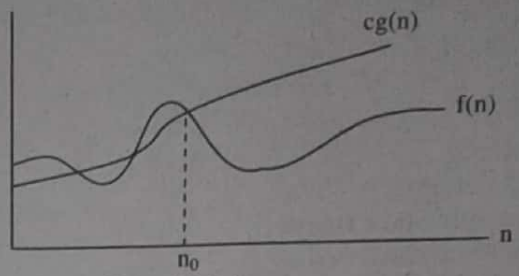
**Or**

**What are the limitations of Big-O notation? Show the big-Oh notation for sequential searching?**

**Ans: Big-oh (O)-Notation**
Big-Oh notation also known as 'upper bound' calculates an upper bound for a function within a constant factor. O(g(n)) is pronounced as "big-oh of g of n" or just "big of g of n".

**It represents the longest amount of execution time that an algorithm takes.**

We write
1) **$f(n) = O(g(n))$ :** If there are positive constants $n_0$ and c such that the value of f(n) always lies on or below cg(n) to the right of $n_0$.
2) **$O(g(n)) = \{f(n)$ :** There exist positive constants c and $n_0$ such that $0 <= f(n) <= cg(n))$ for all $n >= n_0\}$.



f(n) = O(g(n))

**Figure 2.1**

3) **f(n) = O(g(n)):** This shows that g(n) is an asymptotically upper bound of f(n).

Big Oh is a characteristic scheme that measures properties of algorithm complexity performance and/or memory requirements. The algorithm complexity can be determined by eliminating constant factors in the analysis of the algorithm. Clearly, the complexity function f(n) of an algorithm increases as 'n' increases.

**For example,** let us find out the algorithm complexity by analysing the **sequential searching algorithm**. In the sequential search algorithm we simply try to match the target value against each value in the memory. This process will continue until we find a match or finish scanning the whole elements in the array. If the array contains 'n' elements, the maximum possible number of comparisons with the target value will be 'n' i.e., the worst case. That is the target value will be found at the $n^{th}$ position of the array.
$f(n) = n$

i.e., the worst case is when an algorithm requires a maximum number of iterations or steps to search and find out the target value in the array.

The best case is when the number of steps is less as possible. If the target value is found in a sequential search array of the first position (i.e., we need to compare the target value with only one element from the array)—we have found the element by executing only one iteration (or by least possible statements):
$f(n) = 1$

Average case falls between these two extremes (i.e., best and worst). If the target value is found at the n/2nd position, on an average we need to compare the target value with only half of the elements in the array, so:
$f(n) = n/2$

The complexity function f(n) of an algorithm increases as 'n' increases. The function f (n) = O(n) can be read as "f of n is big Oh of n" or as "f (n) is of the order of n". The total running time (or time complexity) includes the initializations and several other iterative statements through the loop.

The generalized form of the theorem is
$f(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + ..... + c_2 n^2 + c_1 n^1 + c_0 n^0$

Where the constant $c_k > 0$

Then, $f(n) = O(n_k)$

Based on the time complexity representation of the big Oh notation, the algorithm can be categorized as:
1) Constant time O(1)
2) Logarithmic time Olog(n)
3) Linear time O(n)
4) Polynomial time $O(n^c)$ Where $c > 1$
5) Exponential time $O(c^n)$

**Limitations of Big-O Notation**
Big Oh Notation has following two basic limitations:
1) It contains no effort to improve the programming methodology. Big-Oh Notation does not discuss the way and means to improve the efficiency of the program, but it helps to analyse and calculate the efficiency (by finding time complexity) of the program.
2) It does not exhibit the potential of the constants. **For example,** one algorithm is taking $1000n^2$ time to execute and the other $n^3$ time. The first algorithm is $O(n^2)$, which implies that it will take less time than the other algorithm which is $O(n^3)$. However in actual execution the second algorithm will be faster for $n < 1000$.

**Ques 15) Find O-notation for the following function given as:**

$$f(n) = 5n^3 + n^2 + 3n + 2$$

**Ans:** We have,

$f(n) = 5n^3 + n^2 + 3n + 2$

For $n \geq 2$

$5n^3 + n^2 + 3n + 2 \leq 5n^3 + n^2 + 3n + n \leq 5n^3 + n^2 + 4n$

For $n^2 \geq 4n$

$5n^3 + n^2 + 4n \leq 5n^3 + n^2 + n^2 \leq 5n^3 + 2n^2$

For $n^3 \geq 2n^2$

$5n^3 + 2n^2 \leq 5n^3 + n^3 \leq 6n^3 \forall\ n \geq n_0 = 2$

Thus, we find that $c = 6$
and $g(n) = n^3$
Since, $f(n) = O(g(n))$
$\Rightarrow f(n) = O(n^3)$

**Ques 16) Find O-notation for the function $f(n) = 2^n + 6n^2 + 3n$.**

**Ans:** Given function $f(n) = 2^n + 6n^2 + 3n$ ....(1)

for $n^2 \geq 3n$

$2^n + 6n^2 + 3n \leq 2^n + 6n^2 + n^2 \leq 2^n + 7n^2$

i.e., we are showing the function is asymptotically tight by upper bound

for $2^n \geq n^2$ [$n \geq 4$].

$2^n + 7n^2 \leq 2^n + 7n^2 * 2^n \leq 8 * 2^n \quad \forall\ n \geq n_0 = 4$

Hence, we find that $c = 8$
and $g(n) = 2^n$

Since, we know from the definition of O-notation
$f(n) = O(g(n))$
$f(n) = O(2^n)$

**Ques 17) Find out O-notation for the function $f(n) = 3n^3 + 4n$.**

**Ans:** Given that
$f(n) = 3n^3 + 4n$
Here, $m = 3$ (order of polynomial)
$\Rightarrow f(n) = O(n^m)$
$\Rightarrow f(n) = O(n^3)$

**Ques 18) What is the application of asymptotic Notations in algorithm analysis?**

**Ans: Application of Asymptotic Notations in Algorithm Analysis**
In some cases one can draw conclusions about the relative behaviour of algorithms, for sufficiently large inputs, based on asymptotic relationships between the algorithms' growth functions.

**For example,** if two algorithms $A_1$ and $A_2$ solve the same problem, the worst case running time for $A_1$ is in $O(f_1)$, and the worst case running time for $A_2$ is in $\Omega(f_2)$, for asymptotically positive functions $f_1$ and $f_2$, and if $f_1 \in o(f_2)$, then algorithm $A_1$ will be faster than $A_2$, when run on at least some large inputs.

If $A_1$'s worst case running time is in $O(f_1)$, $A_2$'s worst case running time is in $\Omega(f_2)$, and $f_1 \in O(f_2)$, then $A_1$ might not ever be faster than $A_2$ – but there will exist some constant c such that the time used by $A_1$ is at most c times that used by $A_2$, on infinitely many large inputs. One cannot conclude either of these things, if he/she has only established that $A_2$'s running time is in "$O(f_2)$" instead of in "$\Omega(f_2)$" (why?).

Obviously, this is not all one want to be able to prove when comparing the performance of two algorithms – but it's a start. If one has established the first of the above two results, then he/she could perform a more careful analysis in order to discover how large the inputs must be in order for $A_1$ to be faster than $A_2$. One might also try to discover whether the "worst case" is a typical, or whether it's close to the "average case" as well.

In the second case (one knows that $f_1 \in O(f_2)$ and not that $f_1 \in o(f_2)$), he/she could perform a more careful analysis, in order to determine what the above constant c really is. The result is not very interesting if c is large, and one cannot really consider $A_1$ to be superior based on the above results unless $c < 1$.

**Ques 19) What are the common functions?**

**Ans: Common Complexity Functions**

1) **Monotone Function:** A function f(n) is monotonically increasing if $m \le n$ implies $f(m) \le f(n)$. Similarly, it is monotonically decreasing if $m \le n$ implies $f(m) \ge f(n)$. A function f(n) is strictly increasing if $m < n$ implies $f(m) < f(n)$ and strictly decreasing if $m < n$ implies $f(m) > f(n)$.

2) **Rounding Function:** For any real number x, denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x"). For all real x,

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1.$$

For any integer n,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n,$$

And for any real number $n \ge 0$ and integers $a, b > 0$,

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil,$$

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor,$$

$$\lceil a/b \rceil \le (a + (b-1))/b,$$

$$\lfloor a/b \rfloor \ge ((a - (b-1))/b.$$

The floor function $f(x) = \lceil x \rceil$ is monotonically increasing, as is the ceiling function $f(x) = \lfloor x \rfloor$.

3) **Polynomial Functions:** Given a non-negative integer d, a polynomial in n of degree d is a function p(n) of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i,$$

where the constants $a_0, a_1, \ldots, a_d$ are the coefficients of the polynomial and $a_d \ne 0$. A polynomial is asymptotically positive if and only if $a_d > 0$. We say that a function f(n) is polynomially bounded if $f(n) = O(n^k)$ for some constant k.

4) **Exponential Functions:** For all real $a > 0$, m, and n, we have the following identities:

$$a^0 = 1,$$
$$a^1 = a,$$
$$a^{-1} = 1/a,$$
$$(a^m)^n = a^{mn},$$
$$(a^m)^n = (a^n)^m,$$
$$a^m a^n = a^{m+n}.$$

For all n and $a \ge 1$, the function $a^n$ is monotonically increasing in n. When convenient, one shall assume $0^0 = 1$.

The rates of growth of polynomials and exponentials can be related by the following fact. For all real constants a and b such that $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0,$$

from which we can conclude that

$$n^b = o(a^n).$$

Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

5) **Logarithm Function:** One shall use the following notations:

$$\lg n = \log_2 n \text{ (binary logarithm)},$$
$$\ln n = \log_e n \text{ (natural logarithm)},$$
$$\lg^k n = (\lg n)^k \text{ (exponentiation)},$$
$$\lg \lg n = \lg(\lg n) \text{ (composition)}.$$

An important notational convention adopted is that logarithm functions will apply only to the next term in the formula, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg(n+k)$. If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

For all real $a > 0$, $b > 0$, $c > 0$, and n

$$a = b^{\log_b a},$$

$$\log_c (ab) = \log_c a + \log_c b,$$

$$\log_b a^n = n \log_b a,$$

$$\log_b a = \frac{\log_c a}{\log_c b},$$

$$\log_b (1/a) = -\log_b a,$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a},$$

where, in each, logarithm bases are not 1.

A function f(n) is poly algorithmically bounded if $f(n) = O(\lg^k n)$ for some constant k. One can relate the growth of polynomials and polylogarithms by substituting lg n for n and $2^a$ for a in equation below yielding

$$\lim_{n \to \infty} \frac{\lg^b n}{\left(2^a\right)^{\lg n}} = \lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0.$$

From this limit, we can conclude that

$$\lg^b n = o(n^a)$$

for any constant $a > 0$.

**Ques 20) What are the common complexity functions?**

**Ans: Table 2.1 below** shows the common complexity functions:

**Table 2.1**

| Function | Name |
|---|---|
| c | Constant |
| log N | Logarithmic |
| $\log^2 N$ | Log-squared |
| N | Linear |
| N log N | N log N |
| $N^2$ | Quadratic |
| $N^3$ | Cubic |
| $2^N$ | Exponential |

# AVL, RED-BLACK AND B-TREES

**Ques 21) Define AVL trees. Explain its rotation operations with example.**

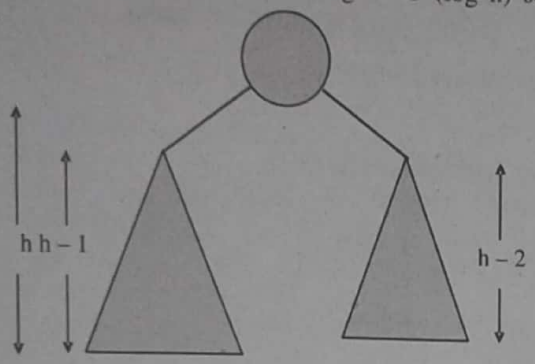
**Or**

**Describe all rotations in AVL tree.**

**Ans: AVL Tree**

An AVL tree is another balanced binary search tree. Named after their inventors, **Adelson-Velskii** and **Landis**, they were the first dynamically balanced trees to be proposed.

In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced tree maintaining an O (log n) search time.




Addition and deletion operations also take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation.

Insertions and deletions may require the tree to be re-balanced by one or more tree rotations.

An AVL Tree the worst case scenario for a search is O(log n).

An AVL tree is a binary search tree which has the following properties:

1) The subtrees of every node differ in height by at most one.
2) Every subtree is an AVL tree.

**BF (balance factor) = (Height of Right Subtree) – (Height of Left Subtree)**

**For example,**

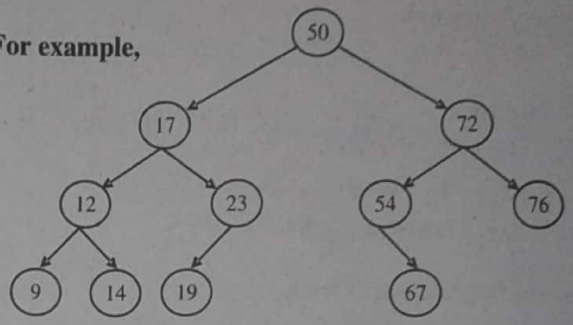
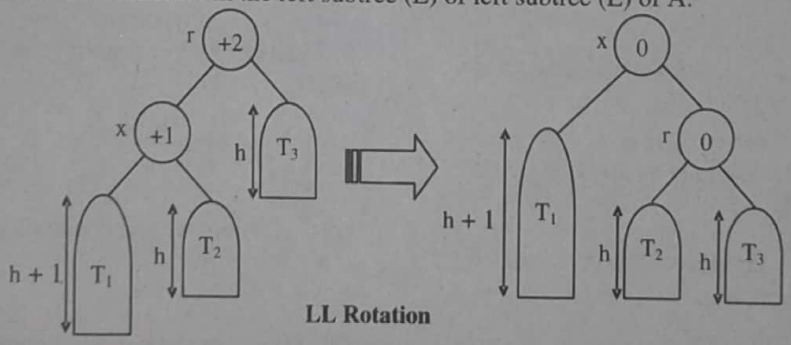



**Figure 2.2: Height-Balanced Tree**

**AVL Tree Rotations**

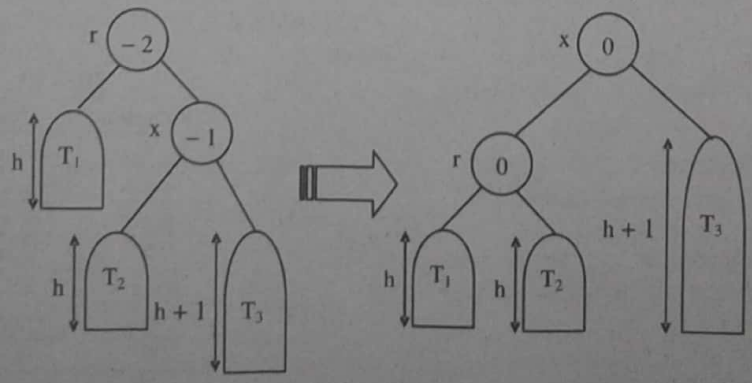
An AVL Tree and the nodes it contains must meet strict balance requirements to maintain is O(log n) search capabilities. These balance restrictions are maintained using various rotation functions. Below is a diagrammatic overview of the four possible rotations that can be performed on an unbalanced AVL Tree, illustrating the before and after states of an AVL Tree requiring the rotation:

1) **LL Rotation:** New node is inserted in the left subtree (L) of left subtree (L) of A.



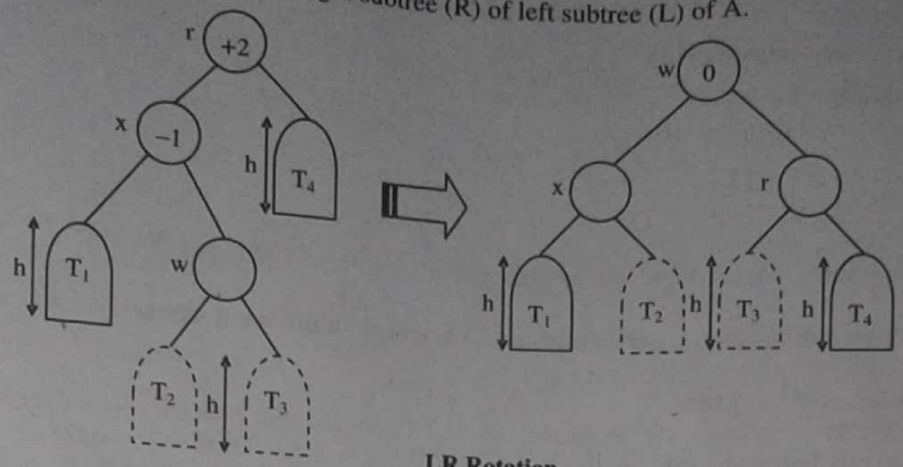

**LL Rotation**

2) **RR Rotation:** New node is inserted in the right subtree (R) of right subtree (R) of A.



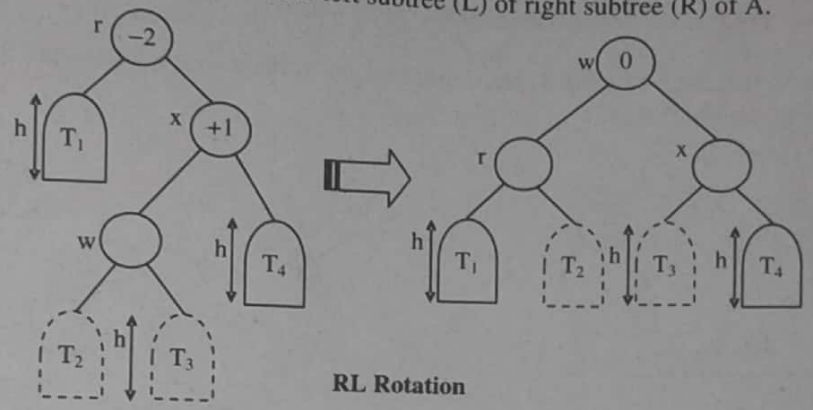

**RR Rotation**

3) **LR Rotation:** New node is inserted in the right subtree (R) of left subtree (L) of A.




**LR Rotation**

4) **RL Rotation:** New node is inserted in the left subtree (L) of right subtree (R) of A.




**RL Rotation**

**Ques 22) What is the maximum height of any AVL tree with 7 nodes?**

**Ans:** Maximum Nodes in an AVL tree with height n is $H_n = H_{n-1} + H_{n-2} + 1$.

$H_0 = 1$.

$H_1 = 2$

$H_2 = H_1 + H_0 + 1 = 2+1+1 = 4$

$H_3 = H_2 + H_1 + 1 = 4+2+1 = 7$.

So the max height with 7 nodes is 3.

**Ques 23) What is the minimum number of nodes in an AVL tree of height 5?**

**Ans:** If h is the height of an AVL tree, then minimum number of nodes can be obtained for such AVL tree using the following formula:

$N(h) = 1 + N(h-1) + N(h-2)$ .....(1)

If there is only 1 node then height h is 0. Similarly for 2 nodes the maximum height can be 1. We can formulate and say;

$N(0) = 1, N(1) = 2$

Hence we can obtain minimum number of nodes for height 5 using equation (1) as;

$N(5) = 1 + N(4) + N(3)$ .....(2)

Let us compute, N (4)

$N(4) = 1 + N(3) + N(2)$ .....(3)

Let us compute, N (3)

$N(3) = 1 + N(2) + N(1)$ .....(4)

$N(2) = 1 + N(1) + N(0) = 1 + 2 + 1$

$N(2) = 4 \rightarrow$ put this value in equation (4)

$N(3) = 1 + N(2) + N(1)$
$N(3) = 7 \rightarrow$ put this value in equation (3)

$N(4) = 1 + N(3) + N(2) = 1 + 7 + 4$
$N(4) = 12 \quad \rightarrow$ put this value in equation (2)

$N(5) = 1 + N(4) + N(3) = 1 + 12 + 7$
$N(5) = 20$

Thus, there could be minimum 20 nodes those can be arranged in an AVL tree of height 5.

**Ques 24) Explain the insertion operation performed on AVL tree with different cases. Give an example.**

**Ans: Insertion Operation on AVL Tree**
Implementations of AVL tree insertion rely on adding an extra attribute, the balance factor to each node. This factor indicates whether the tree is left-heavy (the height of the left subtree is 1 greater than the right subtree), balanced (both subtrees are the same height) or right-heavy (the height of the right subtree is 1 greater than the left subtree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.

A new item has been added to the left subtree of node 1, causing its height to become 2 greater than 2's right subtree. A right-rotation is performed to correct the imbalance.

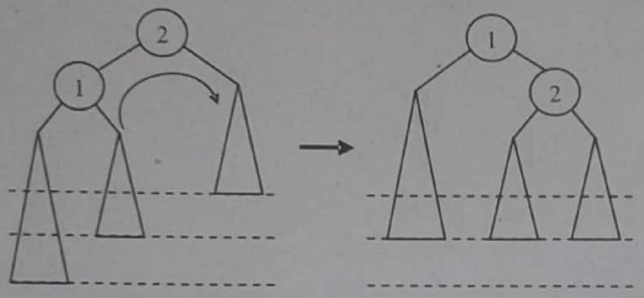
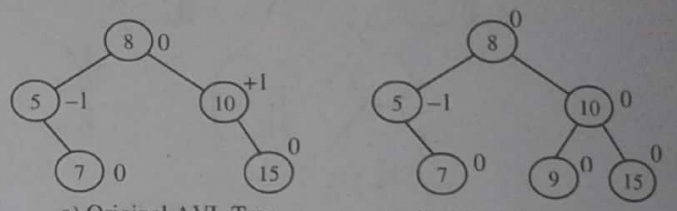



Figure 2.3

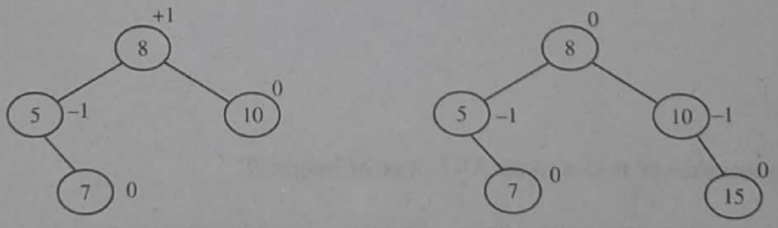



a) Original AVL Tree b) After Inserting Value 9

Figure 2.4

In insertion following cases are aroused owing to the need of rotation or not and the various situations of rotations:
**Case 1:** Neither the balance factor of the root nor the height of the AVL tree is affected

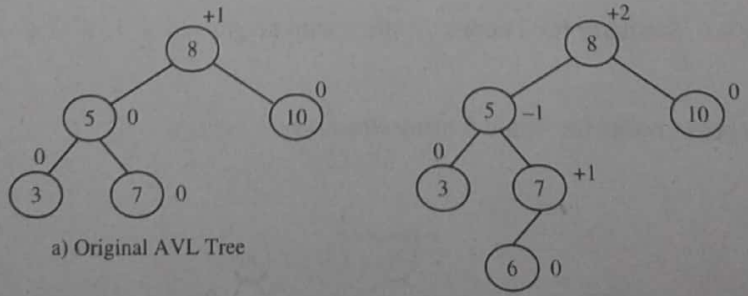
**Case 2:** Height remains unchanged but the balance factor of the root gets changed




a) Original AVL Tree b) After Inserting Value 15

Figure 2.5

**Case 3:** Height as well as balance factor gets changed. It needs re-balancing about root node




a) Original AVL Tree b) After Inserting Value 6

Figure 2.6

**For example,** Cases for insertion in AVL Tree are demonstrated below through examples:
**Case I:** Inserted node is in the left subtree of left subtree of node A.

Here the balance property is restored by single right rotation.

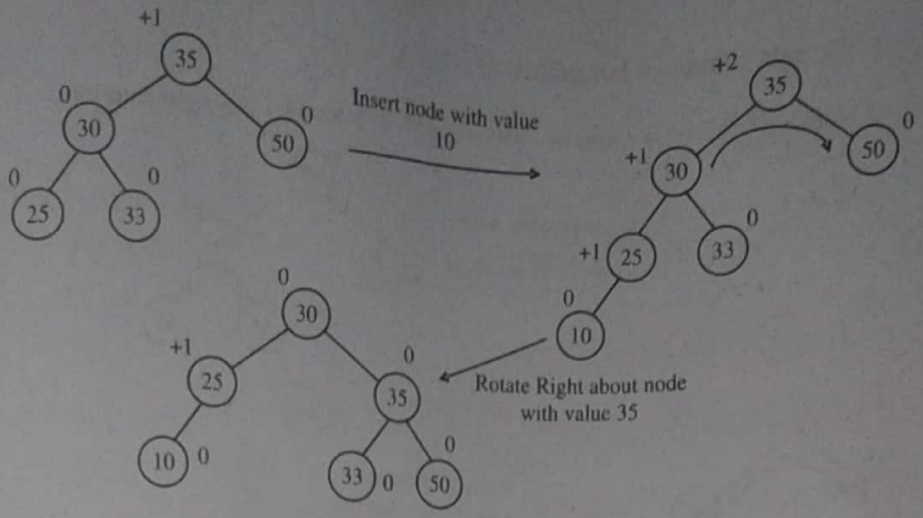



**Figure 2.7: Restoring Balance by Right Rotation**

**Case II:** Inserted node is in the right subtree of right subtree of node A.

Here the balance property is restored by single left rotation.

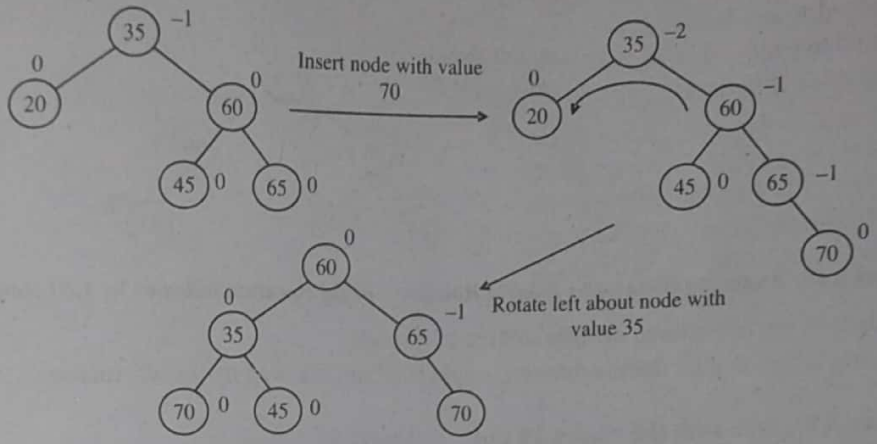



**Figure 2.8: Restoring Balance by Left Rotation**

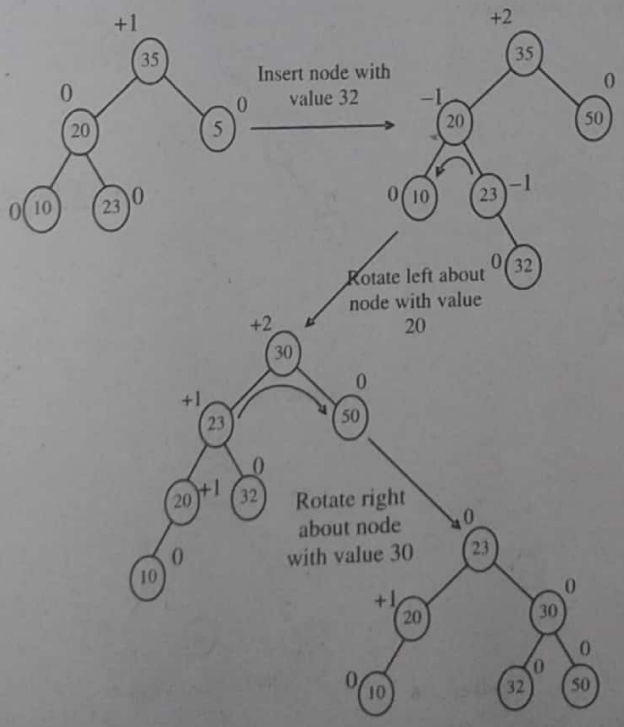



**Figure 2.9: Restoring Balance by Double Rotation (Left Rotation followed by Right Rotation)**

**Case III:** Inserted node is in the right subtree of left subtree of node A.

Here the balance property is restored by double rotation – left rotation followed by the right rotation.

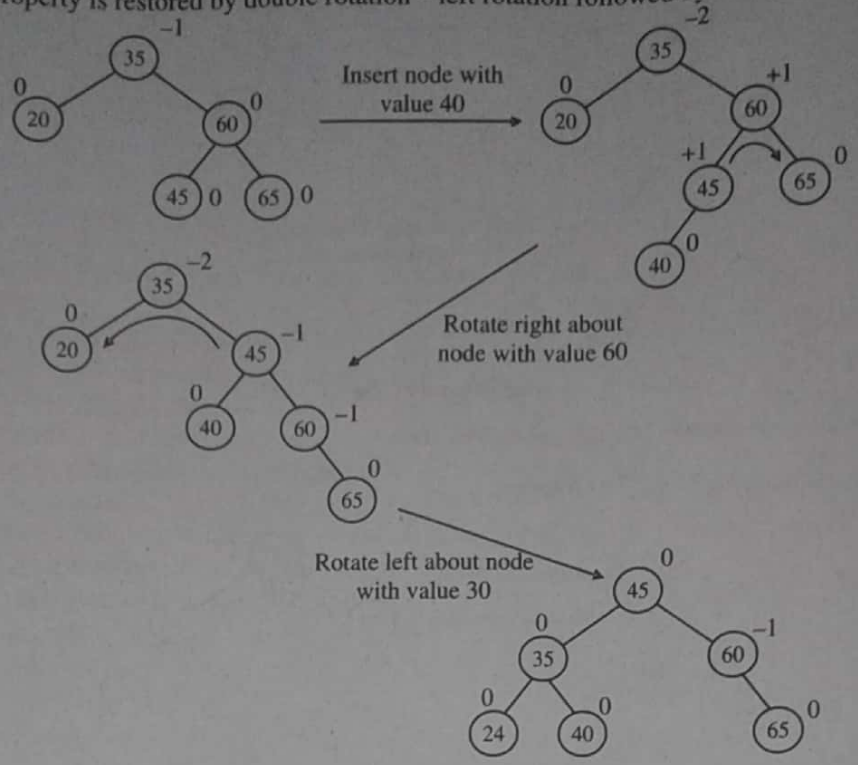


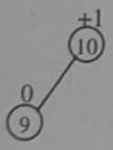

**Figure 2.10: Restoring Balance by Double Rotation (Right Rotation followed by Left Rotation)**

**Case IV:** Inserted node is in the left subtree of right subtree of node A.
Here the balance property is restored by double rotation – right rotation followed by the left rotation.

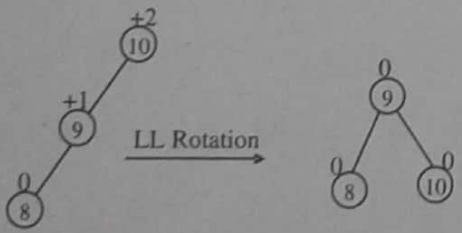
**Ques 25) Construct an AVL tree with the values 10 to 1 numbers into an initially empty tree.**

**Ans: AVL Tree with the Values 10 to 1 Numbers**

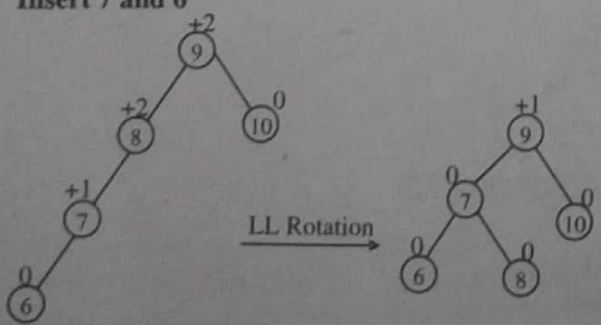
**Insert 10 and 9**



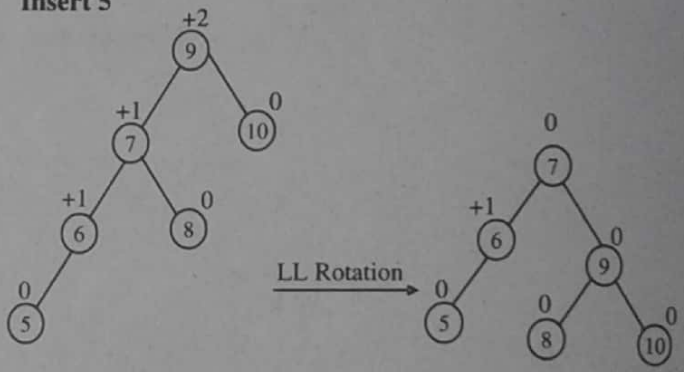
**Insert 8**



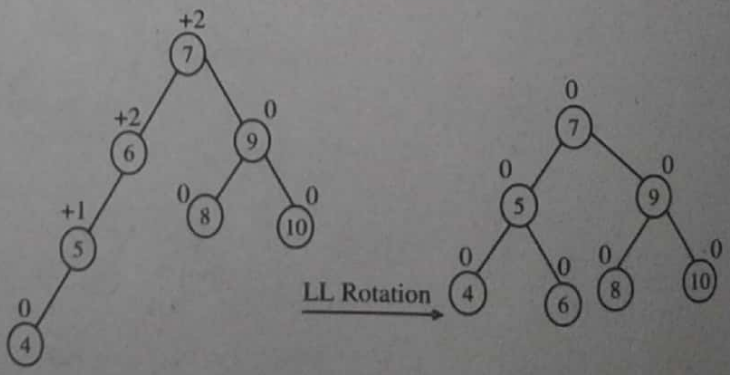

**Insert 7 and 6**



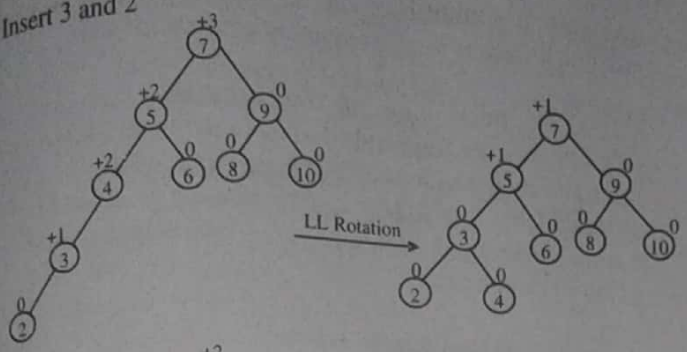

**Insert 5**



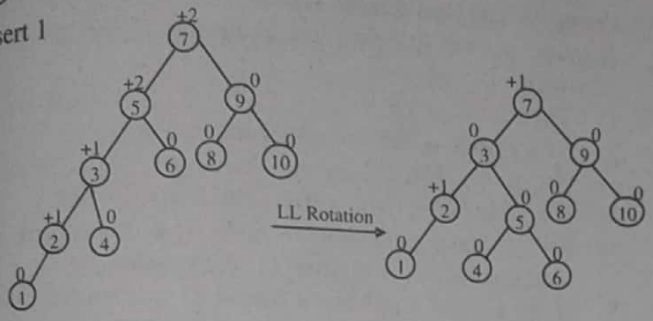

**Insert 4**



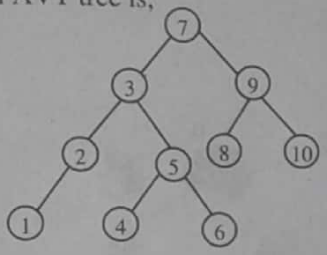
Insert 3 and 2




Insert 1

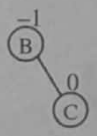



Hence, the final AVT tree is,

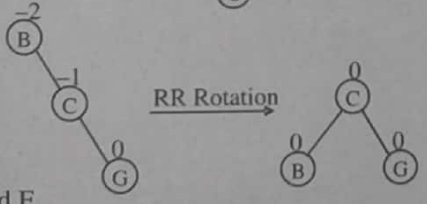


**Ques 26) Construct AVL tree from the following nodes: B, C, G, E, F, D, A.**

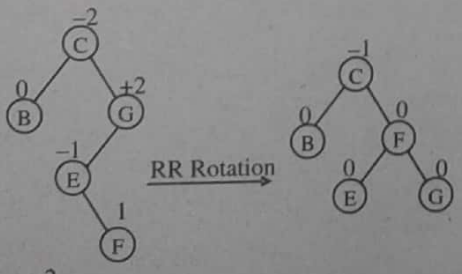
**Ans:** Insert B and C



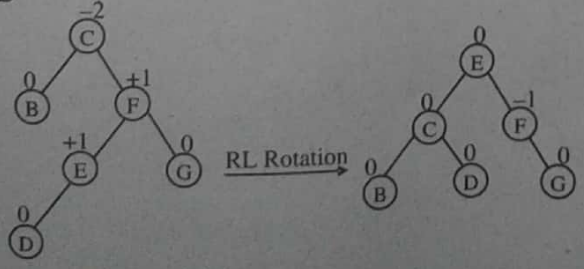
Insert G



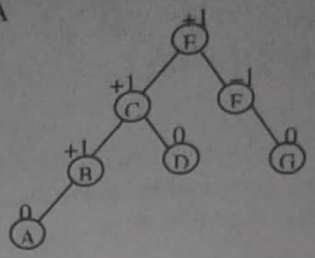

Insert E and F




Insert D




Insert A



This is the required AVL tree.

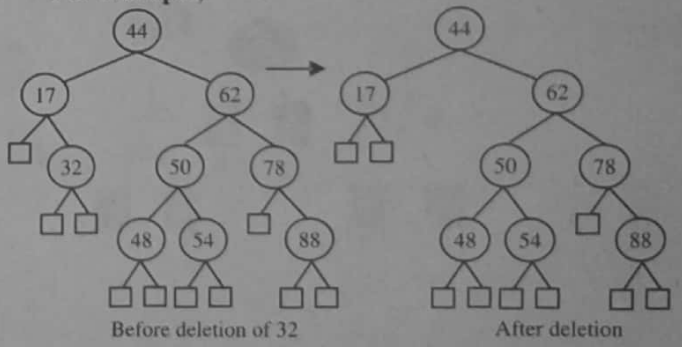
**Ques 27) Define deletion operation performed on AVL tree.**

**Ans: Deletion from AVL Tree**

The deletion algorithm for AVL Trees is a little more complex, as there are several extra steps involved in the deletion of a node. If the node is not a leaf node (i.e., is has atleast one child), then the node must be swapped with either its in-order successor or predecessor (based on availability). Once the node has been swapped it can be deleted (and have its parent pick up any children it may have provided that it will only ever have at most one child).

If a deletion node was originally a leaf node, then it can simply be removed. Now, as with the insertion algorithm, traverse back up the path to the root node, checking the balance of all nodes along the path. If an unbalanced node is encountered an appropriate rotation performed to balance the node.

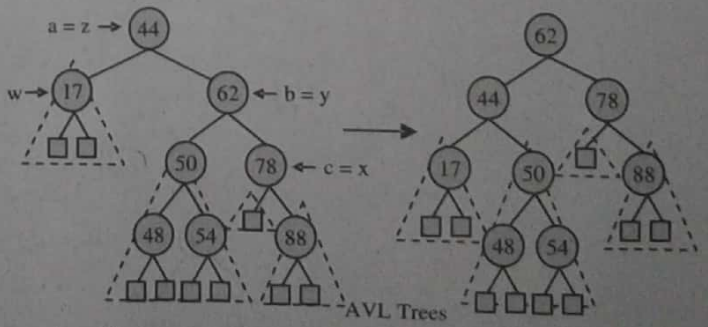
**For example,**




**Re-Balancing after Deletion**

Let z be the first unbalanced node encountered while traveling-up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.

Performing re-structure (x) to restore balance at z.




As this re-structuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached.

**Ques 28) What is Red-Black Tree? What are the main properties of Red-Black Tree? List the different operations performed on Red-Black Tree.**

**Ans: Red-Black Trees**

A red-black tree is a type of self-balancing binary search tree, a data structure used in computer science to organise pieces of comparable data, such as numbers.

The self-balancing is provided by painting each node with one of two colors (these are typically called 'red' and 'black', hence the name of the trees) in such a way that the resulting painted tree satisfies certain properties that don't allow it to become significantly unbalanced. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

The leaf nodes of red–black trees do not contain data. These leaves need not be explicit in computer memory—a null child pointer can encode the fact that this child is a leaf—but it simplifies some algorithms for operating on red–black trees if the leaves really are explicit nodes.

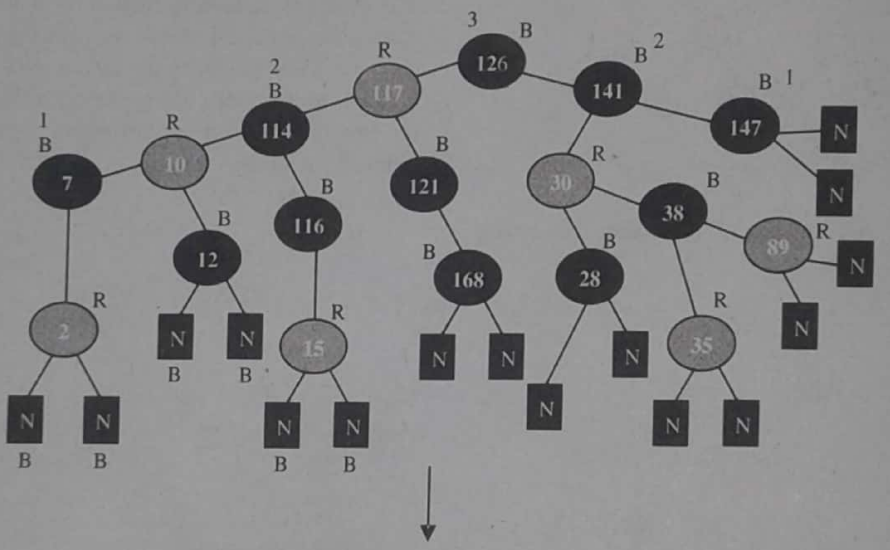
Red–black trees, like all binary search trees, allow efficient in-order traversal (that is: in the order Left–Root–Right) of their elements. The search-time results from the traversal from root to leaf, and therefore a balanced tree of n nodes, having the least possible tree height, results in O(log n) search time.

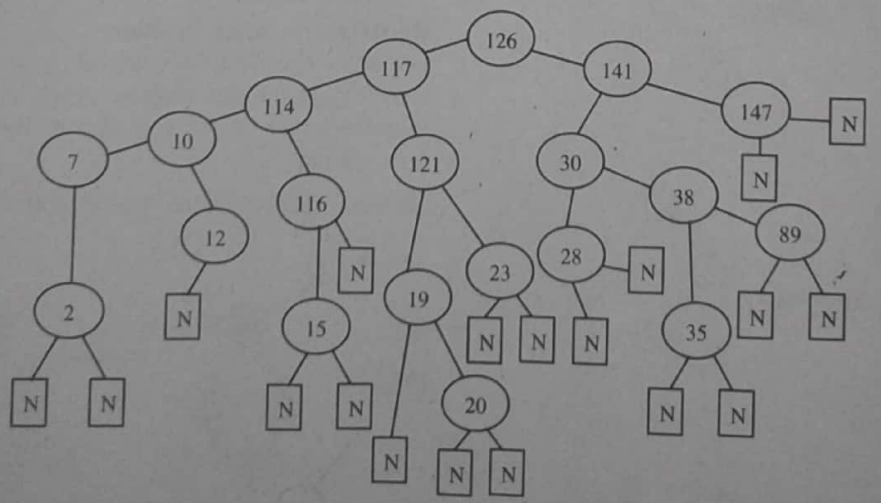
**Properties of Red-Black Trees**

A binary search tree is a red black tree if it satisfies the following properties:

1) The root is black.
2) Every node is colored either red or black.
3) Every leaf (NIL pointers) is black.
4) If a node is red then both its children are black.
5) Every single path from a node to a descendant leaf contains the same number of black nodes.
6) If the root of a red-black tree is black then we cannot color it red because for one of its children might be red.

**For example,**




R – Red Node
B – Black Node
N – Nil

**Note:** The black leaf nodes are the Nil nodes.

For a red-black tree T, the sentinel nil [T] is an object with the same fields as an ordinary node in the tree. Its color field is BLACK, and its other fields- p, left, right and key can be set to arbitrary values. The figure above shows all pointers to NIL are replaced by pointers to the sentinel nil [T]. The sentinel can be used to treat a NIL child of a node x as an ordinary node whose parent is x. The internal nodes of red black tree hold the key values.

**Operations in Red-Black Tree**

The operations in Red-Black tree are:

1) Rotation
2) Insertion
3) Deletion

**Ques 29) Proof the theorem that a red-black tree with n internal nodes has height at most 2 lg (n + 1).**

**Ans: Proof**

The strategy followed is that first bound the number of nodes in any subtree, then bound the height of any subtree. Any subtree rooted at x has at least $2bh(x) - 1$ internal nodes, where bh(x) is the black height of node x.

**Proof by Induction**

$bh(x) = 0 \rightarrow x$ is a leaf, $\rightarrow 2^0 - 1 = 0$

Now assume it is true for all tree with black height < bh(x).If x is black, both subtrees have black height bh(x) – 1. If x is red, the subtrees have black height bh(x).Therefore, the number of internal nodes in any subtree is

$n \geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 \geq 2^{bh(x)} - 1$

Now, let h be the height of red-black tree. At least half the nodes on any single path from root to leaf must be black if the root is ignored. Thus $bh(x) \geq h/2$ and $n \geq 2^{h/2} - 1$, so $n + 1 \geq 2^{h/2}$

This implies that $\lg(n + 1) \geq h/2$,so $h \leq 2 \lg(n + 1)$.

Therefore red-black trees have height at most twice optimal. A balanced search tree satisfies the red-black tree structure under insertion and deletion.

**Ques 30) Discuss the rotation operation of Red-Black tree.**

**Or**

**What are the different types of rotations?**

**Ans: Rotation Operation of Red-Black Tree**

The basic restructuring step for binary search trees is left and right rotation. Rotation is a local operation in a search tree which preserves the binary search tree property.

An in-order search tree before a rotation stays an in-order search tree. In a rotation, one subtree gets one level closer to the root and one subtree one level further from the root. The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take O (lg n) time. To restore these properties, one must change the colors of some of the nodes in the tree and also change the pointer structure.

The pointer structure is changed through **rotation**, which is a local operation in a search tree that preserves the binary-search-tree property. When a left rotation on a node x is done as shown in **figure below**, it is right child Y is not nil [T]; x may be any node in the tree whose right child is not nil [T]. The left rotation "pivots" around the link from X to Y. It makes Y the new root of the subtree, with X as Y's left child and Y's left child X's right child.

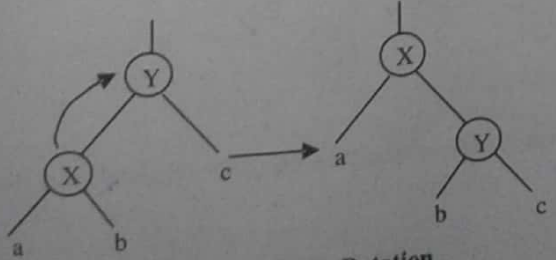



**Figure 2.11: Right Rotation**

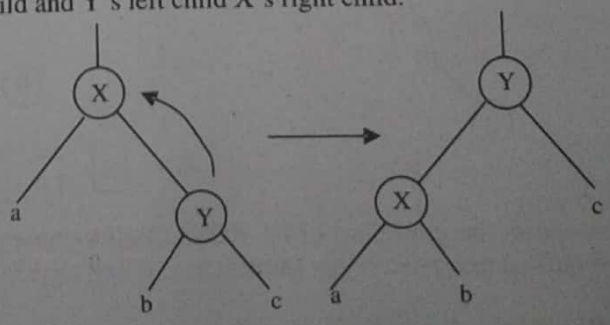
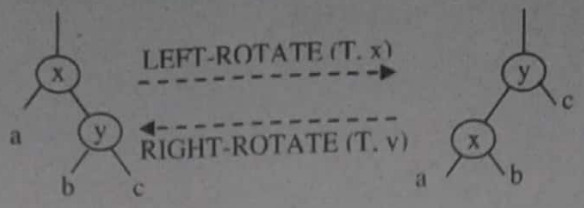



**Figure 2.12: Left Rotation**

**Types of Rotation**

**Figure below** shows the two kinds of rotations

1) Left rotations
2) Right rotations




The letters a, b, and c represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: The keys in a precede key [x], which precedes the keys in b, which precede key [y], which precedes the keys in c. The in-order property is preserved.

The pseudocode for **LEFT-ROTATE** assumes that right [x] ≠ nil [T] and that the root's parent is nil [T].

**Figure above shows** the rotation operations on a binary search tree. The operation **LEFT-ROTATE (T, x)** transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation **RIGHT ROTATE (T, y).**

**Algorithm: LEFT-ROTATE (T, x)**

**Step 1:** y ← right [x] //Set y.
**Step 2:** right [x] ← left [y] //Turn y's left subtree into x's right subtree.
**Step 3:** p[left[y]] ← x
**Step 4:** p[y] ← p [x] // Link x's parent to y.
**Step 5:** if p[x] = nil [T]
**Step 6:** then root [T] ← y
**Step 7:** else if x = left [p[x]]
**Step 8:** then left [p[x]] ← y
**Step 9:** else right [p[x]] ← y.
**Step 10:** left [y] ← x //Put x on y's left.
**Step 11:** p [x] ← y

**For example,**

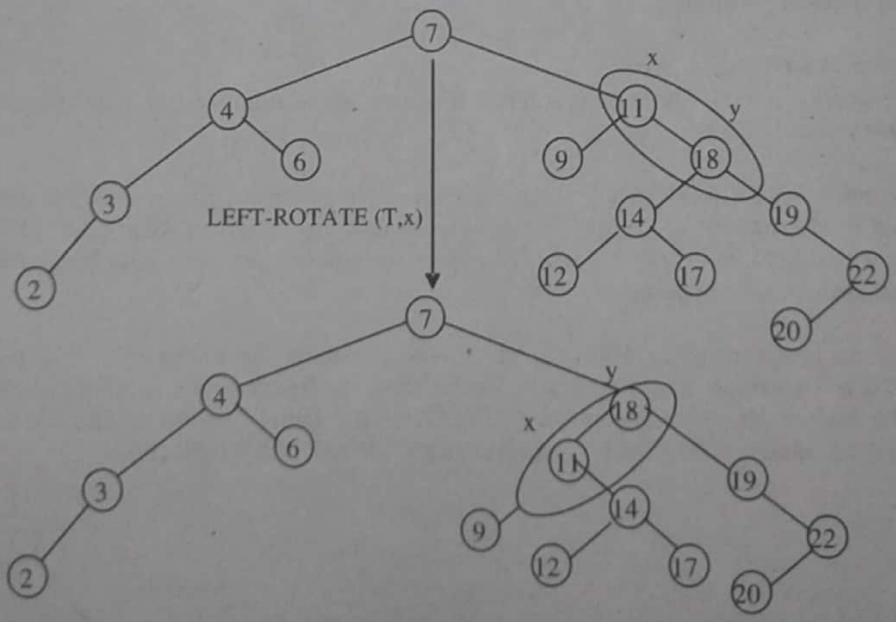



**Figure above** shows the procedure LEFT-ROTATE (T, x) modifies a binary search tree. In order tree walks of the input tree and the modified tree produce the same listing of key values.

**Analysis of Rotation**

LEFT-ROTATE run in O (1) time. Only pointers are changed by a rotation; all other fields in a node remain the same.

**Ques 31) Discuss how to insert an element in Red-Black Tree?**

**Ans: Insertion**

Consider the insertion of an element x with key into a red-black tree T. The insertion algorithm initially proceeds as in a binary search tree. Namely, one search for k in T until an external node of T is reached and replace this node with an internal node z, storing (key, x) and having two external-node children. If z is the root of T, color z black, else color z red.

**Algorithm : RB-INSERT (T, z)**

**Step 1:** y ← nil[T]
**Step 2:** x ← root[T]
**Step 3:** while x ≠ nil[T]
**Step 4:** do y ← x
**Step 5:** if key[z] < key[x]
**Step 6:** then x ← left[x]
**Step 7:** else x ← right[x]
**Step 8:** p[z] ← y
**Step 9:** if y = nil[T]
**Step 10:** then root[T] ← z
**Step 11:** else if key[z] < key[y]
**Step 12:** then left[y] ← z
**Step 13:** else right[y] ← z
**Step 14:** left[z] ← nil[T]
**Step 15:** right[z] ← nil[T]
**Step 16:** color[z] ← RED
**Step 17:** RB-INSERT-FIXUP (T, z)

After insertion the red black property is disrupted so in order to regain RB property RB-INSERT-FIXUP algorithm is employed.

**Algorithm : RB-INSERT-FIXUP (T, z)**

**Step 1:** while color [p[z]] = RED
**Step 2:** do if p[z] = left[p[p[z]]]
**Step 3:** then y ← right[p[p[z]]]
**Step 4:** if color[y] = RED
**Step 5:** then color[p[z]] ← BLACK //CASE 1
**Step 6:** color[y] ← BLACK //CASE 1
**Step 7:** color[p[p[z]]] ← RED//CASE 1
**Step 8:** z ← p[p[z]] //CASE 1
**Step 9:** else if z = right[p[z]]
**Step 10:** then z ← p[z] //CASE 2
**Step 11:** LEFT-ROTATE (T, z) //CASE 2
**Step 12:** color [p[z]] ← BLACK
**Step 13:** color[p[p[z]]] ← RED
**Step 14:** RIGHT-ROTATE (T, p[p[z]])
**Step 15:** else (same as then clause with "right" and "left" exchanged)
**Step 16:** color[root[T]] ← BLACK

**Figure 2.13 and Figure 2.14** shows how RB-INSERT-FIXUP (T, z) operates. The children of z are colored black .In addition, it preserves the root, external and depth properties of T, but it may violate the internal property. Indeed, if z is not the root of T and the parent v of z is red, then one have a parent and child (namely, v and z) that are both red. The root property states that v cannot be the root of T and by the internal property the parent u of v must be black. Since z and its parent are red, but z's grandparent u is black, hence this is the violation of the internal property, a double red at node z.

**Cases in Red Black Tree Insertion**

To remedy a double red, consider two cases:

**Case 1: Sibling w of v is Black (Figure 2.13):** In this case, the double red denotes the fact that one has created a deformed replacement in red-black tree, which has as its children the four black children of u, v and z. The deformed replacement has one red node (v) that is the parent of another red node (z), while want it to have the two red nodes as sibling instead. To fix this problem, we perform a trinode restructuring of T. The trinode restructuring is done by the operation restructure (z), which consists of the following steps:

1) Take node z, its parent v and grandparent u and temporarily re-label them as a, b and c, in left-to-right order, so that a, b and c, will be visited in this order by an in-order tree traversal.

2) Replace the grandparent u with the node labeled b and make nodes a and c the children of b, keeping in-order relationships unchanged.

After performing the restructure (z) operation, color b black and color a and c red. Thus, the restructuring eliminates the double-red problem.

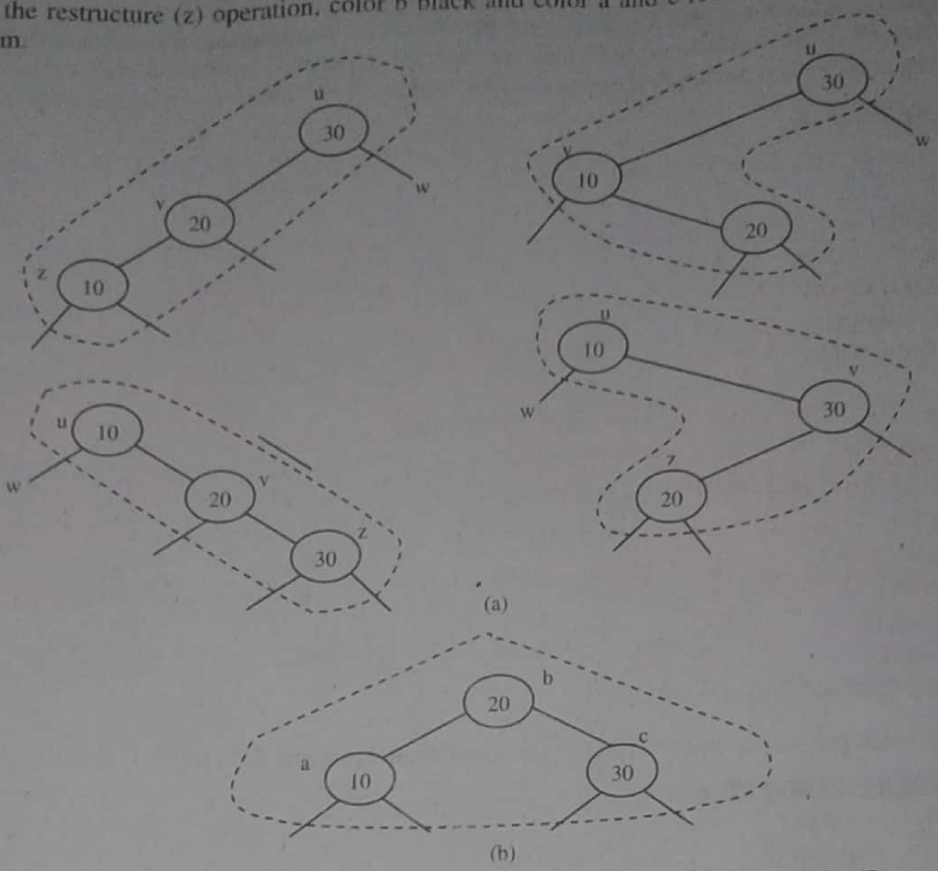



**Figure 2.13: Restructuring a Red-Black Tree to Remedy a Double Red: a) The Four Configurations for u, v, z before Restructuring; b) After Restructuring**

**Case 2: Sibling w of v is Red (Figure 2.14):** In this case, the double red denotes an overflow in the tree T. To fix the problem, perform the equivalent of a split operation. Namely, a re-coloring is done: color v and w black and their parent u red (unless u is the root, in which case, it is colored black). It is possible that, after such a re-coloring, the double-red problem reappears, albeit higher up in the tree T, since u may have a red parent. If the double-red problem reappears at u, then repeat the consideration of the two cases at u. Thus, a re-coloring either eliminates the double-red problem at node z or propagates it to the grandparent u of z. Continue going up T performing re-colorings until one finally resolve the double-red problem (with either a final re-coloring or a trinode restructuring). Thus, the number of re-colorings caused by an insertion is no more than half the height of tree T, i.e., no more than log (n + 1).

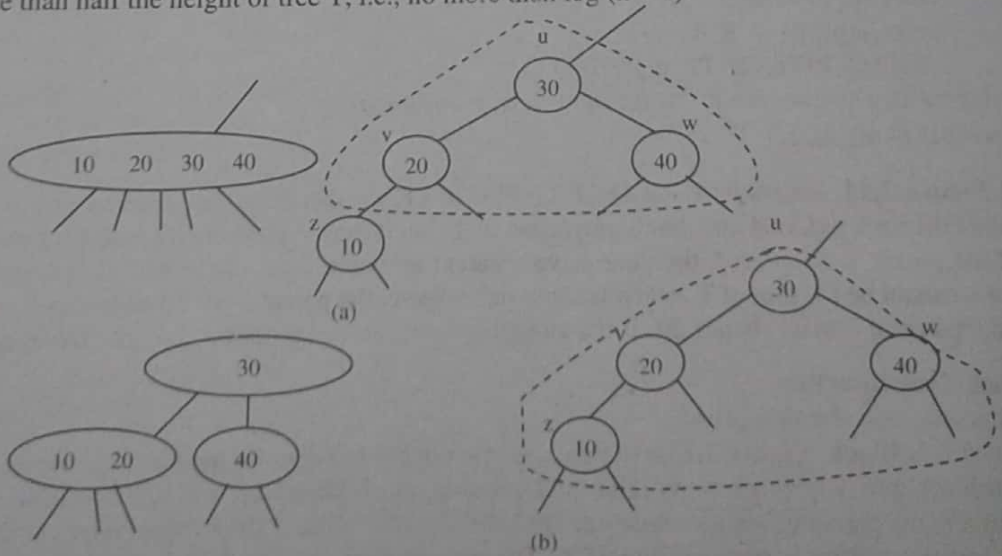



**Figure 2.14: Re-coloring to Remedy the Double-Red Problem: (a) Before re-coloring and the corresponding 5-node in the associated tree before the split; (b) After the re-coloring (and corresponding nodes in the associated tree after the split)**

**Analysis of Insertion**

The height of a red black tree on n nodes is O(lg n), lines 1-16 of RB-INSERT (T, z) takes O(lg n) time. In RB-INSERT-FIXUP (T, z) the while loop repeats only if Case 1 is executed. The total number of times the while loop can be executed is O(lg n) time. Thus RB-INSERT takes a total of O(lg n) time.

The cases for insertion imply an interesting property for red-black trees. Namely, since the Case 1 action eliminates the double-red problem with a single trinode restructuring and the Case 2 action performs no restructuring operations, at most one restructuring is needed in a red-black tree insertion. By the above analysis and the fact that a restructuring or re-coloring takes O(1) time. The insertion of a key-element item in a red-black tree storing n items can be done in O(log n) time and requires at most O(log n) re-colorings and one trinode restructuring (a restructure operations).

**Ques 32) Make a red black tree by inserting the elements 4, 7, 12, 15, 3, 5, 14, 18, 16, 17 in sequential manner.**

**Ans: Figure 2.15** and **2.16** show a sequence of insertions in a red-black tree.

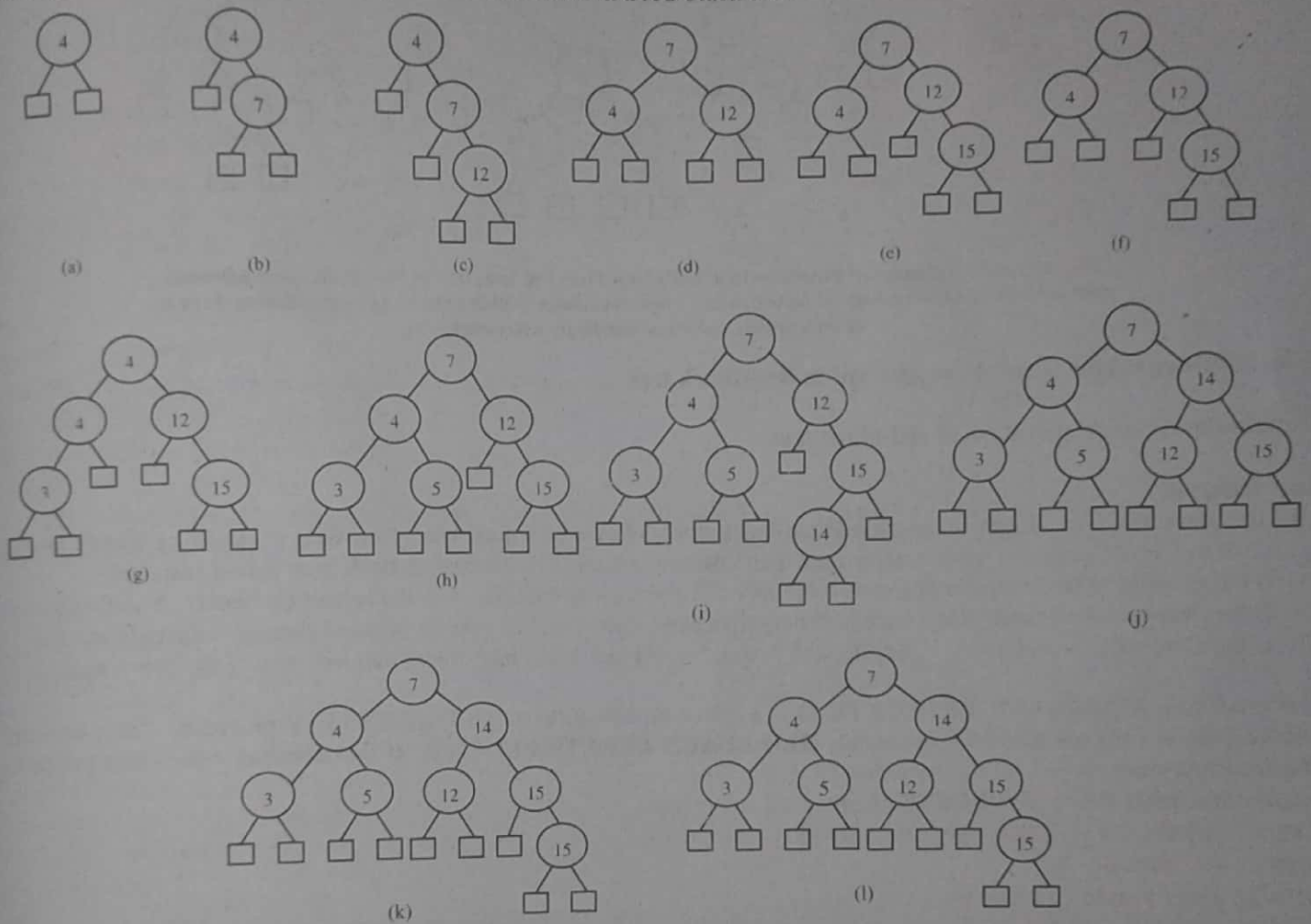
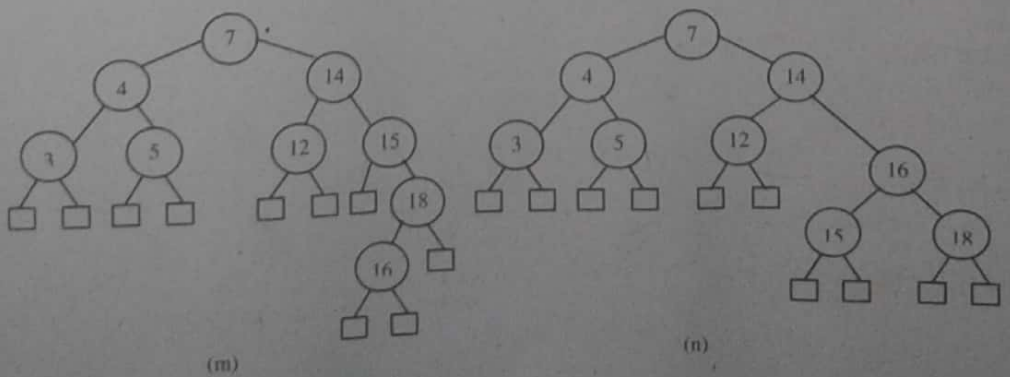



**Figure 2.15: Sequence of Insertions in a Red-Black Tree: (a) Initial tree; (b) Insertion of 7; (c) Insertion of 12, which causes a double red; (d) After restructuring; (e) Insertion of 15, which causes a double red; (f) After re-coloring (the root remains black); (g) Insertion of 3; (h) Insertion of 5; (i) Insertion of 14, which causes a double red; (j) After restructuring; (k) Insertion of 18, which causes a double red; (l) After re-coloring.**



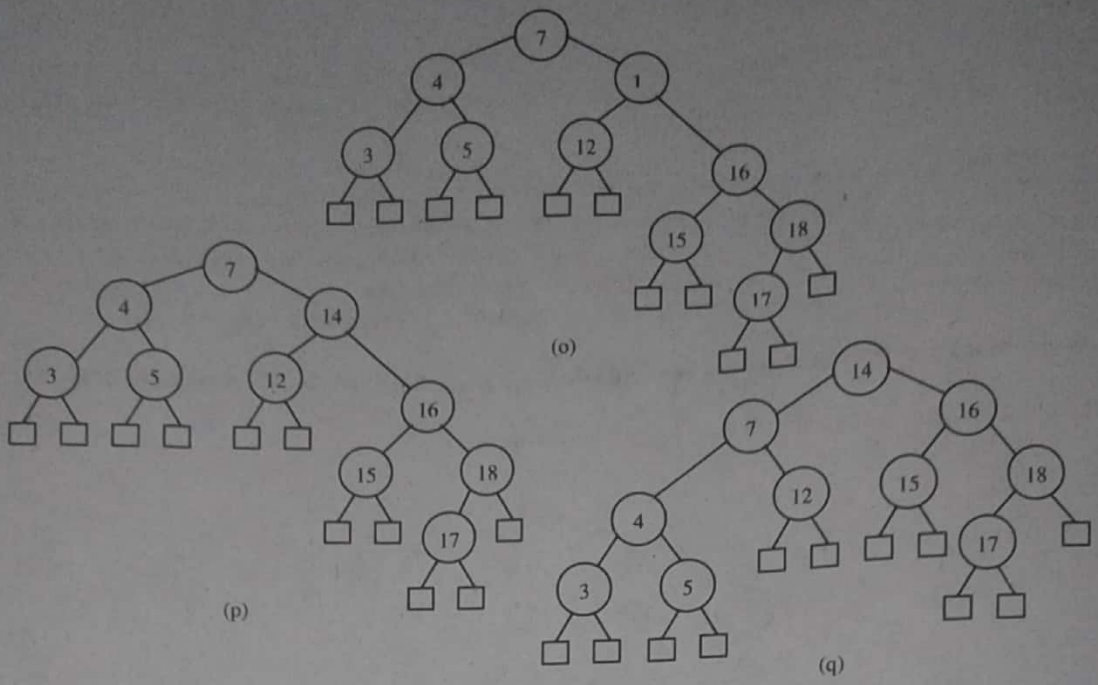



**Figure 2.16: A Sequence of Insertions in a Red-Black Tree: (m) Insertion of 16, which causes a double red: (n) After restructuring; (o) Insertion of 17, which causes a double red; (p) After re-coloring there is again a double red, to be handle by a Restructuring**

**Ques 33) Discuss how to delete an element in Red-Black tree.**

**Or**

**Explain the deletion operation of red-black tree.**

**Ans: Deletion**

Like the other basic operations on an n-node red-black tree, deletion of a node takes time O(lg n). Deleting a node from a red-black tree is only slightly more complicated then inserting a node. The element to be deleted is first searched:

1) If the element to be deleted is in a node with only left child, swap the node with the largest element in the left sub tree.
2) If the element to be deleted is in a node with only right child, swap the node with the smallest element in the right sub tree.
3) If the element to be deleted is in a node with both left child and right child, then swap with any of the above ways.

The procedure **RED-BLACK-DELETE (T, z)** is a minor modification of the TREE-DELETE procedure. After merging out a node, it calls an auxiliary procedure **RED-BLACK-DELETE-FUNC (T, x)** that changes colors and performs rotations to restore the red-black properties.

**Algorithm: RED-BLACK-DELETE (T, z)**

**Step 1:** if left (z) = nil [T] or right [z] = nil [T]
**Step 2:** then i ← z
**Step 3:** else i ← tree-continue (z)
**Step 4:** if left [i] ≠ nil [T]
**Step 5:** then j ← left [i]
**Step 6:** else j ← right [i]
**Step 7:** K[j] ← K[i] //Assigning value of K[i] into k[j]
**Step 8:** If K[i] = nil [T]
**Step 9:** then root [T] ← i
**Step 10:** else if j = left [K[i]]
**Step 11:** then left (K[i]) ← j
**Step 12:** else right (K[i]) ← j
**Step 13:** if i ≠ z
**Step 14:** then temp [z] ← temp [i]
**Step 15:** copy successor data into z
**Step 16:** if color [i] = Black
**Step 17:** then Red-Black-Delete-Funct (T,z)
**Step 18:** Return i

The procedures **RED-BLACK-DELETE-FUNC(T, x)** restore properties 1,2 and 4. The goal of the while loop in step 1-22 is to move the extra black up the tree until:

1) x points to a red black node, in which color x black in step23.
2) x points to the root , in which case the extra black can be simply removed.
3) Suitable rotations and re-colorings can be performed.

**Algorithm: RED-BLACK-DELETE-FUNC (T,x)**

**Step 1:** while (x ≠ root (T) && color [x] = Black
**Step 2:** do if (x = left (K[x]))
**Step 3:** then m ← right ([K[x]])
**Step 4:** if color (m) = Red
**Step 5:** then color [m] ← Black
**Step 6:** color K[x] ← Red
**Step 7:** LEFT-ROTATE (T,K[i])
**Step 8:** m ← right (T,K[x])
**Step 9:** if (color [left [m]]) = Black && right[m] = Black)
**Step 10:** then color [m] ← Red
**Step 11:** x ← K[x]
**Step 12:** else if color [right [m]] ← Black
**Step 13:** then color [left [m]] ← Black
**Step 14:** color [m] ← Red
**Step 15:** RIGHT-ROTATE (T, m)
**Step 16:** m ← right [K[x]]
**Step 17:** color[m] ← color [K[x] ]
**Step 18:** color[K[x]] ← Black
**Step 19:** color[right[m] ← Black
**Step 20:** color [K[x]] ← Black
**Step 21:** LEFT-ROTATE (T, K[x])
**Step 22:** j ← root [T]
**Step 23:** else (otherwise remain same with left & right interchange)
**Step 24:** color [x] ← Black

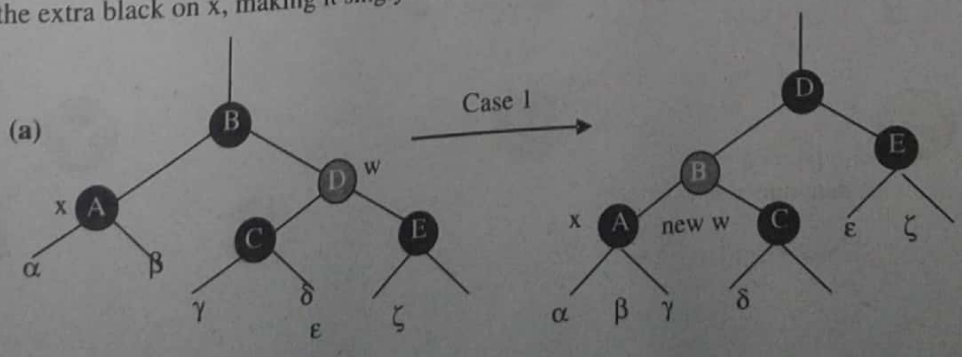
**Cases in Deletion**

**Case 1: x's Sibling w is Red:** Case 1 (lines 5-8 of **RED-BLACK-DELETE-FUNC (T, x)** and **figure 2.17(a)**) occurs when node w, the sibling of node x, is red. Since w must have black children, one can switch the colors of w and p[x] and then perform a left-rotation on p[x] without violating any of the red-black properties.

**Case 2: x's Sibling w is Black and both of w's Children are Black:** In case 2 (lines 10-11 of **RED-BLACK-DELETE-FUNC (T, x)**and **figure 2.17(b)**), both of w's children are black. Since w is also black, one take one black off both x and w, leaving x with only one black and leaving w red. To compensate for removing one black from x and w, one would like to add an extra black to p[x], which was originally either red or black. This is done so by repeating the while loop with p[x] as the new node x.

**Case 3: x's Sibling w is Black, w's Left Child is Red and w's Right Child is Black:** Case 3 (lines 13-16) and **figure 2.17(c)**) occurs when w is black, its left child is red and its right child is black. One can switch the colors of w and its left child left [w] and then perform a right rotation on w without violating any of the red-black properties.

**Case 4: x's Sibling w is Black and w's Right Child is Red:** Case 4 (lines 17-21 and **figure 2.17(d)**) occurs when node x's sibling w is black and w's right child is red. By making some color changes and performing a left rotation on p[x], one can remove the extra black on x, making it singly black, without violating any of the red-black properties.



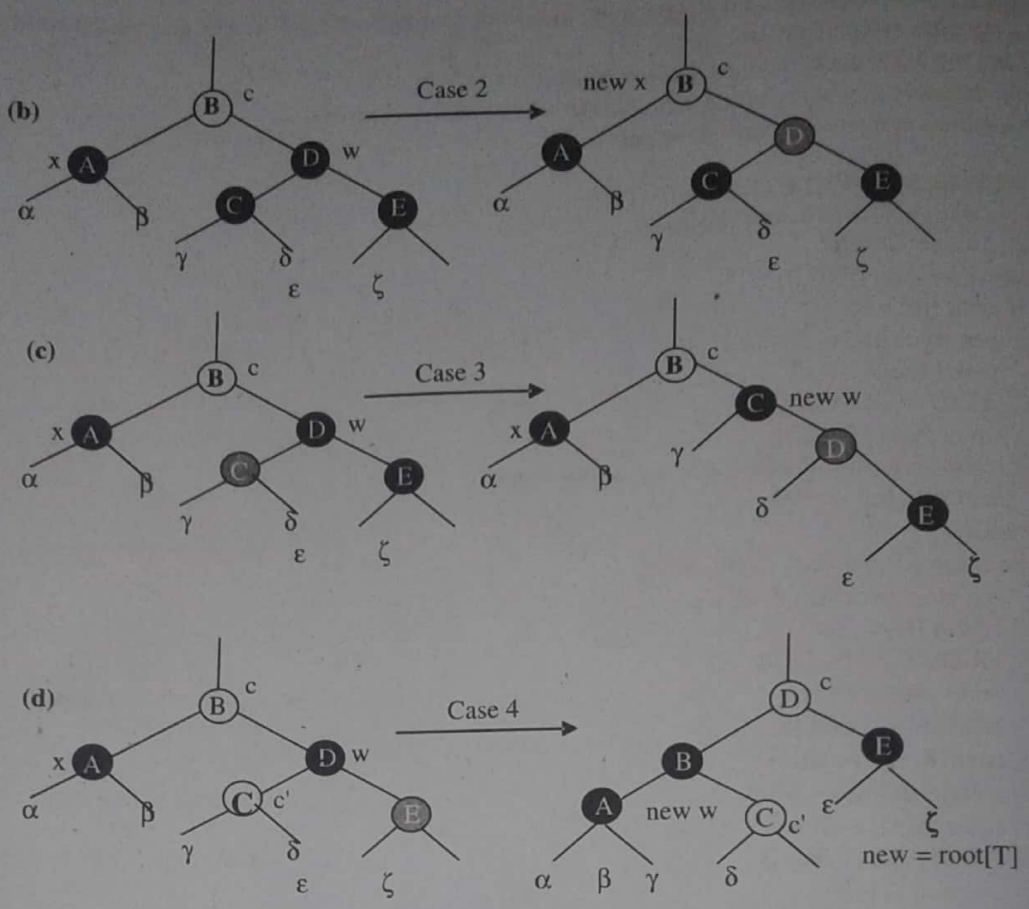



**Figure 2.17**

**Figure 2.17** shows the cases in the **while** loop of the procedure **R B-DELETE-FIXUP**. Darkened nodes have color attributes black, heavily shaded nodes have color attributes RED, and lightly shaded nodes have color attributes represented by c and c', which may be either RED or BLACK. The letters α, β ..... ζ represent arbitrary subtrees. In each case, the configuration on the left is transformed into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black.

The only case that causes the loop to repeat is case 2.

**Figure 2.17(a)** depicts Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation.

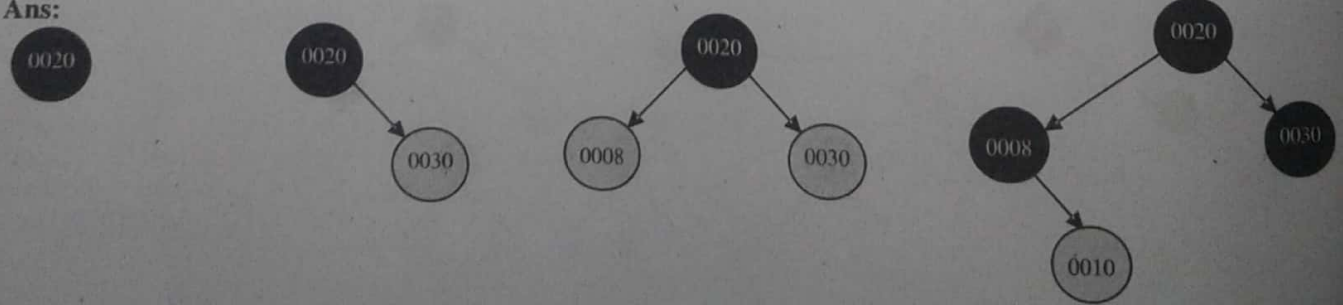
**Figure 2.17(b)** depicts In case 2, the extra black represented by the pointer x is moved up the tree by coloring node D red and setting x to point to node B. If we enter case 2 through case 1, the **while** loop terminates because the new node x is red-and-black, and therefore the value c of its color attribute is RED.

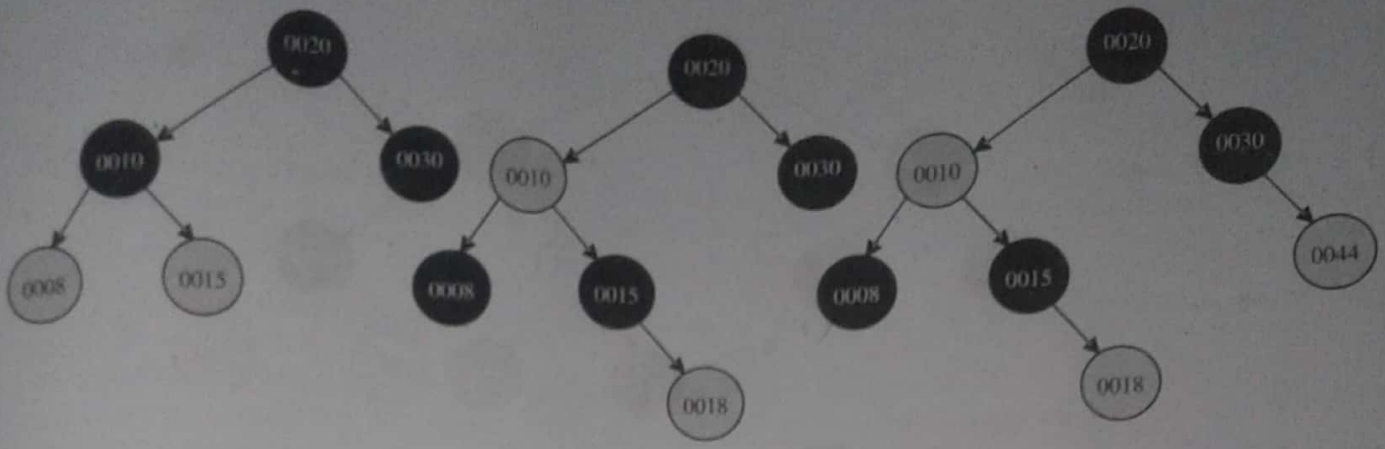
**Figure 2.17(c)** depicts Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation.

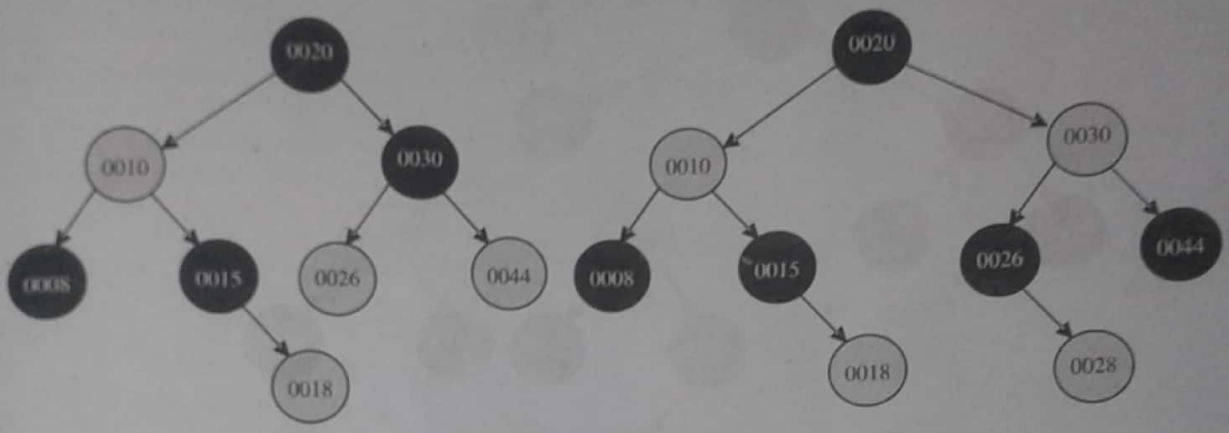
**Figure 2.17(d)** depicts In case 4, the extra black represented by x can be removed by changing some colors and performing a left rotation (without violating the red-black properties), and the loop terminates

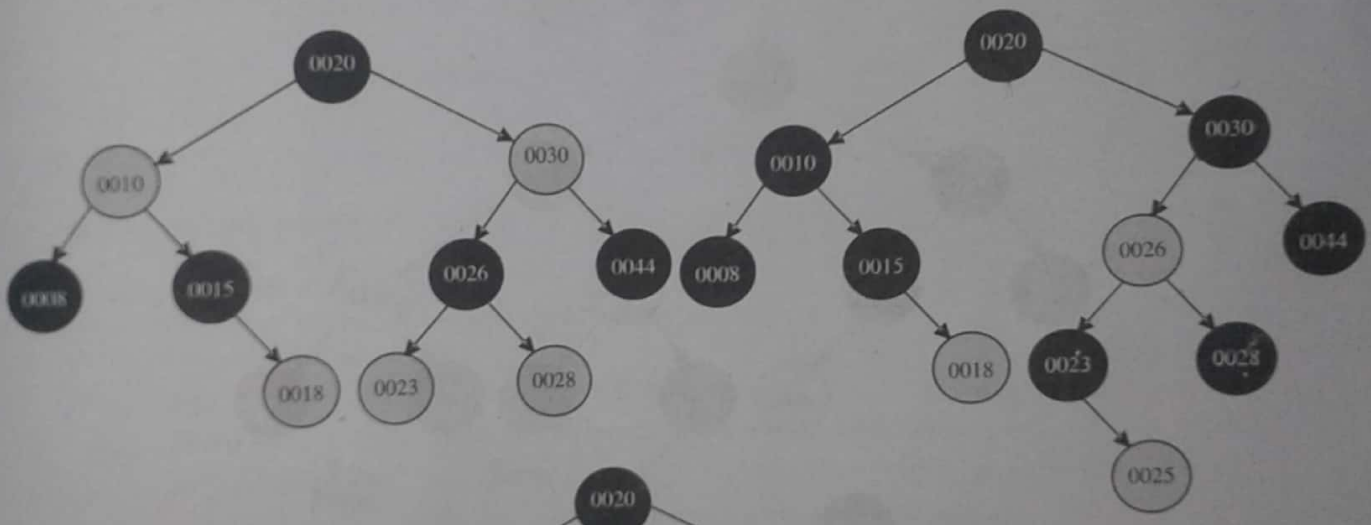
**Ques 34) Draw a red-black tree that stores the following sequence of data:**
**20, 30, 8, 10, 15, 18, 44, 26, 28, 23, 25,43, 55, 36, 49, 39**

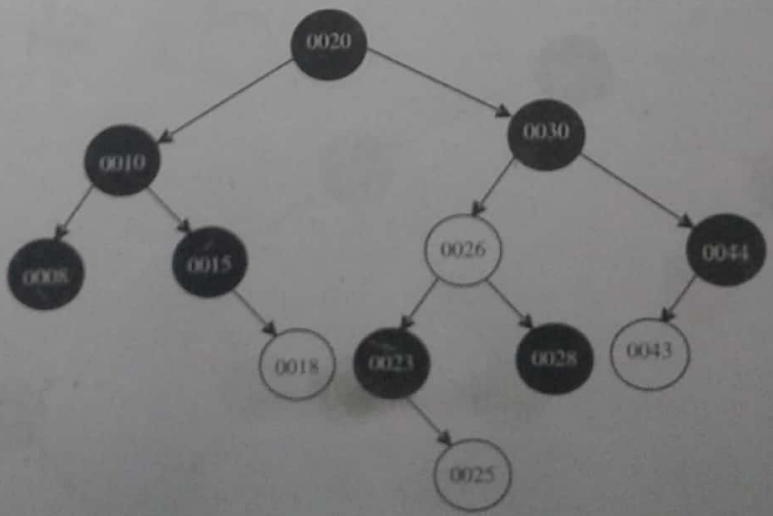
**Ans:**

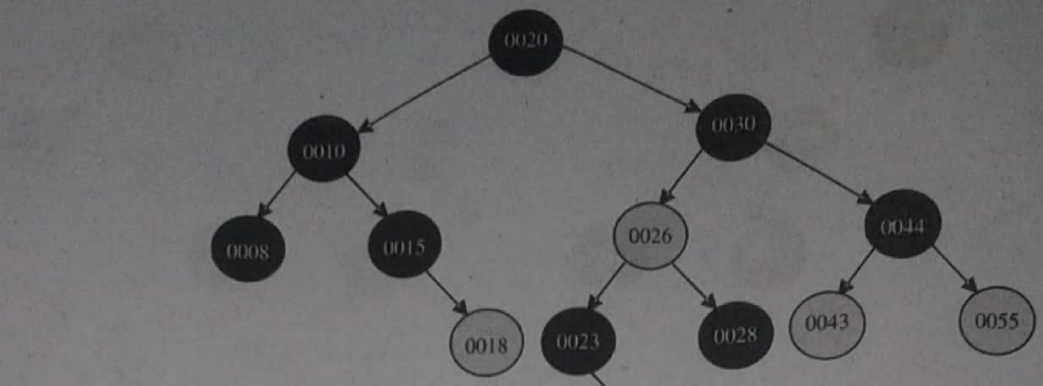


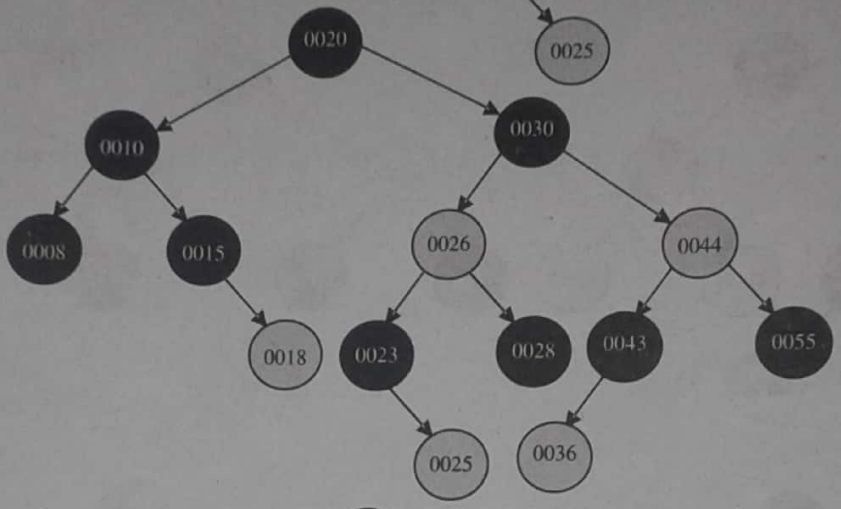


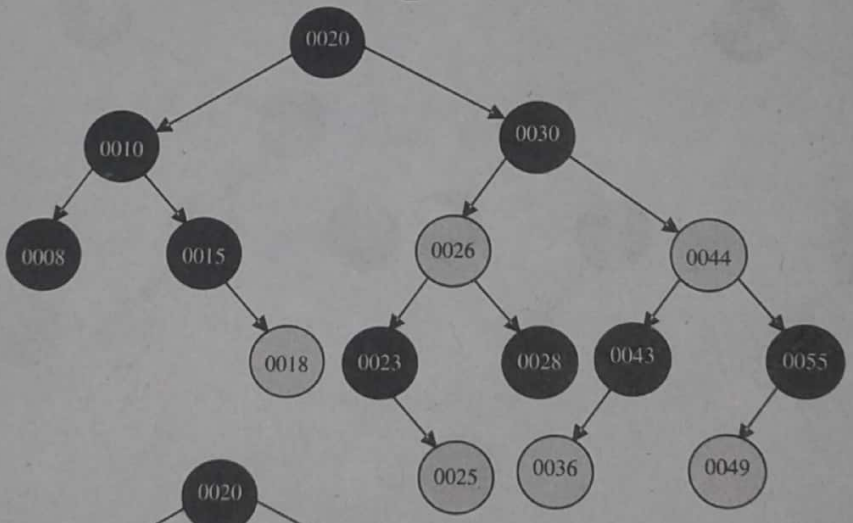



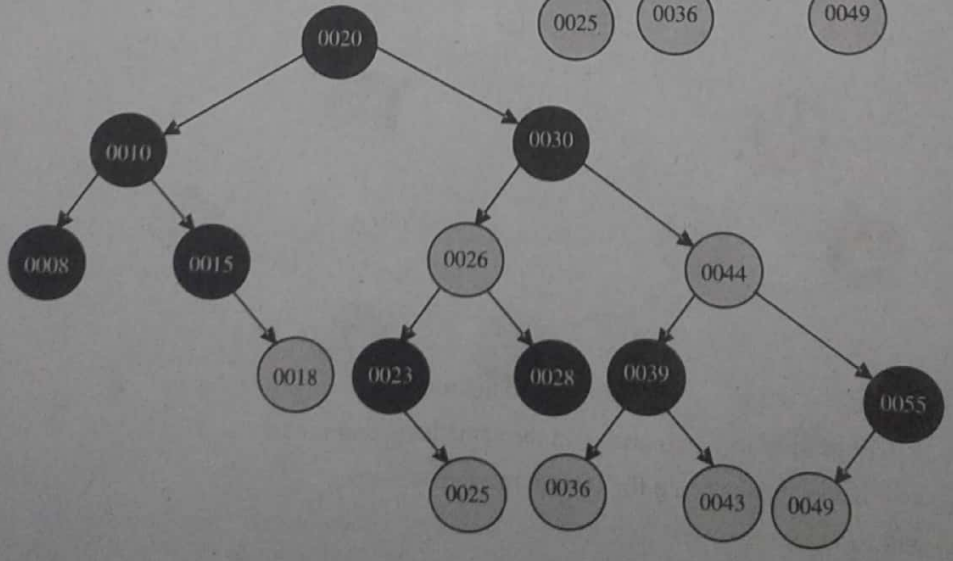





**Ques 35) What is B tree?**
**Or**
**Write a note on B-tree.**

**Ans: B-Trees**
A data structure which is a height balanced version of m-way search tree is known as a B-tree of order m. When the growth of an m-way search tree is left uncontrolled then in the worst case it yield a complexity of O(n). This shows deterioration in performance.

A B-tree of order m is an m-way search tree and it may be empty. If not empty then the following properties are to be satisfied by the extended trees:

1) The root node should have a minimum of two children and a maximum of m children.
2) All the internal nodes except the root node should have a minimum of [m/2] non-empty children and a maximum of m non-empty children.
3) All the external nodes are at the same level.
4) A leaf node must have minimum [m/2 – 1] and maximum m – 1 element.

B-tree with order 3 are also referred to as 2-3 trees as their internal nodes can have only two or three children. **Figure 2.18** shows a B-tree of order 3. B-trees of order 4 are also referred to as 2-3-4 or 2-4 trees:

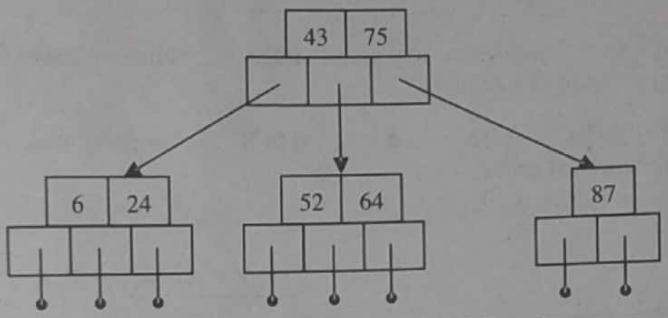



**Figure 2.18: B-Tree of Order 3(2-3 Trees)**

**Ques 36) What are the main properties of B-Tree? Also define its height and order.**
**Or**
**What are the applications of B tree?**

**Ans: Properties of B-Tree of Order 'm'**
A B-tree of order m is a m-way search tree with the following properties:

1) All the leaves are on the same level.
2) All nodes except the root and the leaves have m/2 and m children.
3) The root is either a leaf or has at most m children but may have atleast two children.
4) The number of keys in each node is one less than the number of its children with a maximum of m–l keys.
5) The keys at a level appears in an increasing order from left to right like binary search tree.
6) In case there is no place to insert a key in the node, the node is split into two and median key moves upto make place for a new node and median key becomes the new root.
7) The order of the B-tree refers to the maximum of children, m.
8) A B-tree of order m is called m-(hyphen)(m–1) tree. If the m is 5, B-tree will be called 5-4 tree. The 5 is the order of the B-tree and 4 is the number of keys as each node in B-tree has m children and m-1 keys.
9) A B-tree is also called a balanced sort key as the height of the tree is kept minimum; no empty subtrees above the leaves are kept; all the leaves are kept at the same level and lastly all nodes except the leaves has at least some minimum number of children.
10) Unlike binary tree, B-tree grows from its leaves upward to the root whereas the binary trees grow at their leaves.
11) If node is full, the node will be split into two nodes on the same level and median key is inserted into the parent node.
12) Splitting of node into two nodes makes place for several more insertions.
13) Splitting is not confined to children nodes. Splitting can take place even at main root level if it is full and a node is sent to it.

**Height of B-Trees**

For n greater than or equal to one, the height of an n-key B-tree T of height h with a minimum degree t greater than or equal to 2.

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

The worst case height is O(log n). Since the "branchi-ness" of a B-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures.

Although this does not affect the asymptotic worst case height, B-trees tend to have smaller heights than other trees with the same asymptotic height.

**Order of B-Tree**

The word order and degree are used in B-tree in different sense by different authors. The order refers to the maximum number of children.

In a non-root node contains atleast (m–l)/2 keys and it is called **B-tree of order m**. Each node in a tree has a maximum of m–1 keys and m sons.

**Applications of B-Tree**

B-tree is used in databases and file systems. Databases are generally huge and cannot be maintained entirely in memory; b-trees are used to index the data and to provide fast access.

B-tree structures are also used in file systems. Directories are indexed in NTFS. Indexing makes searching of specific entry fast. They are stored in a B-tree in alphabetical order.

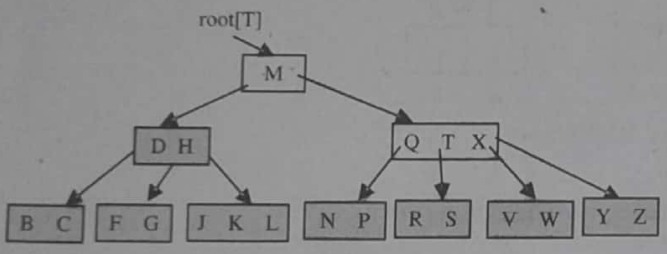



**Figure 2.19: B-Tree Whose Keys are the Consonants of English. An internal node x containing n[x] keys has n[x] + 1 children. All leaves are at the same depth in the tree. The highly shaded nodes are examined in a search for the letter R**

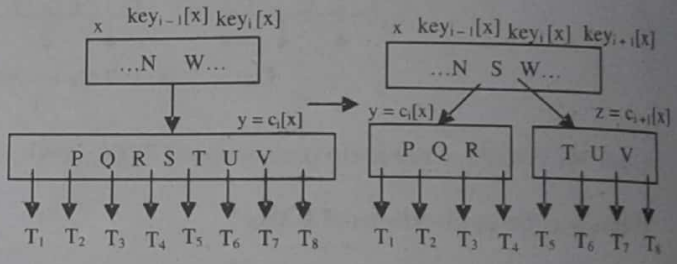



**Figure 2.20: Splitting a Node in B-Tree**

**Ques 37) Discuss the various operations performed on B tree.**

**Or**

**Describe the insertion and deletion operations of B-Tree.**

**Ans: Operations on B-Tree**

The following operations are applied on the B-Tree:

1) **Creation of B-Tree:** The B-TREE-CREATE operation creates an empty B-tree by allocating a new root node that has no keys and is a leaf node. Only the root node is permitted to have these properties; all other nodes must meet the criteria of B tree. The B-TREE-CREATE operation runs in time O(1).

**Algorithm: B-TREE-CREATE (T)**

**Step 1:** x ← ALLOCATE-NODE ()

**Step 2:** leaf[x] ← TRUE

**Step 3:** n[x] ← 0

**Step 4:** DISK-WRITE(X)

**Step 5:** root[T] ← x

2) **Insertion in B-Tree:** The process of insertion of a key in a B-tree is as follows:
Search is made to see that the key is not already in any node. If the key is not there, new key will be added to the node in an increasing order.

**Algorithm: B-TREE-INSERT (T, k)**
**Step 1:** r ← root[T]

**Step 2:** if n[r] = 2t – 1
then s ← ALLOCATE-NODE()
root[T] ← s
leaf[s] ← FALSE
n[s] ← 0
$c_1$ ← r
B-TREE-SPLIT-CHILD(s, 1, r)
B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)

3) **Splitting a Node in a B-Tree:** Splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way upto the root and may require splitting the root node. This approach requires two passes. The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes

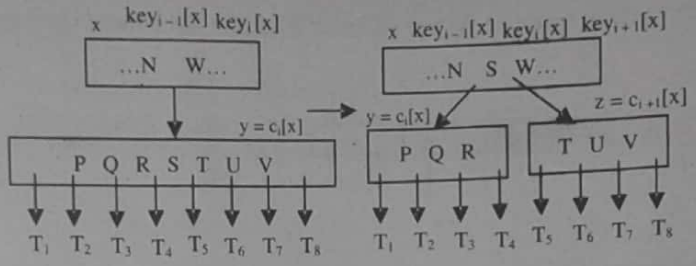



**Figure 2.21: Splitting a Node in B-Tree**

The following algorithm splits the node whenever B-Tree property is hindered.

**Algorithm: B-TREE-SPLIT-CHILD(x, i, y)**
**Step 1:** z ← ALLOCATE-NODE()

**Step 2:** leaf [z] ← leaf[y]

**Step 3:** n[z] ← t – 1
**Step 4:** for j ← 1 to t – 1
do key$_j$ [z] ← key$_{j+t}$ [y]

**Step 5:** if not leaf[y]
then for j ← 1 to t
do $c_j$ [z] ← $c_{j+t}$ [y]
n[y] ← t – 1

**Step 6:** for j ← n[x] + 1 downto i + 1
do $c_{j+1}$[x] ← $c_j$ [x]
$c_{i+1}$ ← z

**Step 7:** for j ← n[x] downto i
do key$_{j+1}$[x] ← key$_j$[x]
key$_i$ [x] ← key$_t$ [y]
n[x] ← n[x] + 1

**Step 8:** DISK-WRITE(Y)

**Step 9:** DISK-WRITE(Z)

**Step 10:** DISK-WRITE(X)

4) **Deletion of a Key from B-Tree:** Deletion of a key depends on the position of the key to be deleted from the B-tree.

The key to be deleted may be found in the following places:

i) **Key is a Leaf with More Than Minimum Keys in the Node:** Key is a leaf and the number of keys in the node from where key is to be deleted is more than the minimum number required. The minimum number is given by n(n-1)/2. Key is deleted and rest of the keys in the node is compacted.

ii) **Key is not a Leaf with More than Minimum Keys in the Node:** Key is not a leaf key and its successor has more than minimum keys in the node. The immediate successor key of key to be deleted is promoted into the position of the key to be deleted and key is deleted.

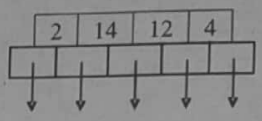
iii) **Key is a Leaf with Less than Minimum Keys in the Node:** Key is a leaf with less than minimum number of keys in the node. The status is underflow. When an underflow occurs, the deletion process may take one of the following two paths.

a) If the sibling/brother of the key to be deleted has more than n(n-l)/2 keys, spare key in the sibling node is pushed in the parent node and a key from parent node is pulled down to take the place of deleted node.

b) There is no spare key with the sibling/brother. Any key cannot be pulled from the parent node as it would leave the parent node with less than minimum keys. The remedial action will be re-distribution of keys. The node is deleted and its left-over key is combined with one of siblings and with the median key from the parent node. Since parent node is left less than minimum keys, all the top nodes are combined into a single node.

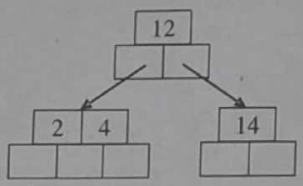
**Ques 38) Create a B-tree of order 5 for the following data:**
**2, 14, 12, 4, 22, 8, 16, 26, 20, 10, 38, 18, 24, 6, 24, 28, 40, 42, 32**

**Ans:** The steps to create a B-Tree of order 5 are as follows:
**Step 1:** Pick up first 4 nodes and sort them in order. A B-tree of order 5 cannot have more than four keys.
Nodes 2 14 12 4

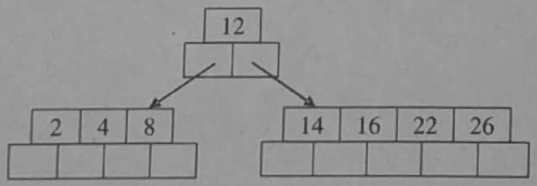



Four nodes will have five pointers.

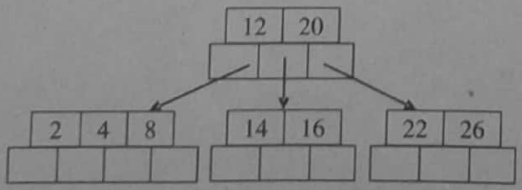
**Step 2:** Pick up the next node 22 and insert it in the proper place. 24 12 14 22. The node cannot accommodate more than 4 keys. The middle key 12 is kept as the root and keys 2 4 and 14 22 as the left and right subtrees of middle node 12 as shown in the figure.



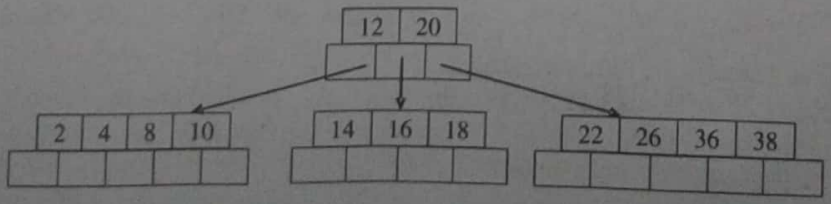

**Step 3:** Keys 8 16 26 are inserted without any changes.



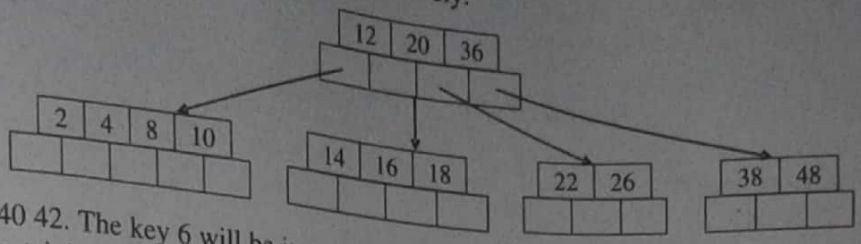

**Step 4:** Key 20 is inserted at 14 16 20 22 26. Since node cannot accommodate more than 4 keys, the middle key 20 is moved up and node is spilt-up into two nodes of 14 16 and 22 and 26 keys.



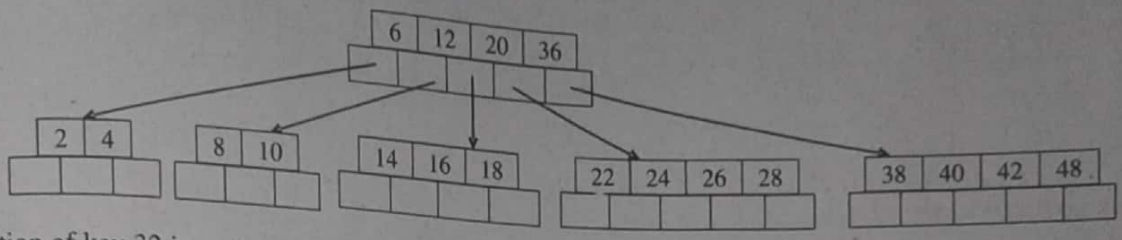

**Step 5:** Keys 1038 18 36 are inserted in ordered places in the nodes.



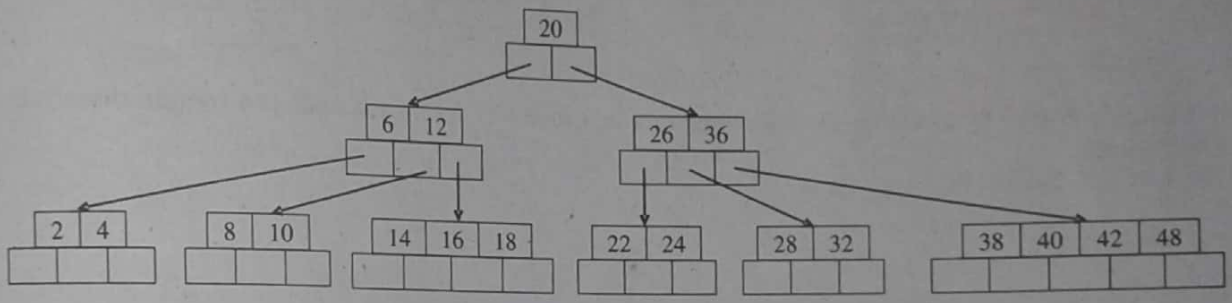
**Step 6:** Insert 48 in 22 24 36 38 48. As in the previous cases, the middle value 36 is moved up and node is spilt into two nodes of consisting of 22 24 keys and 36 and 38 keys respectively.




**Step 7:** Insert 6 24 28 40 42. The key 6 will be inserted in node 2468 10. The middle value will move up and node will be spilt into two nodes containing keys 2 4 and 8 10 respectively. The rest keys are inserted in proper places without any changes.



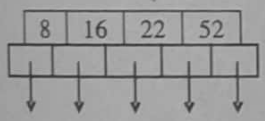

**Step 8:** Insertion of key 32 in node 22 24 26 28 32 upsets the B-tree structure completely. The middle key 26 is moved up in root node 6 12 20 26 36. The root node is unable to accommodate the fifth node, so the middle node 20 is moved up and keys 6 12 and 26 36 are inserted in new nodes below the new root node 20. The node 22 24 26 28 is spilt-up into two nodes. These changes are shown the following **figure** below:



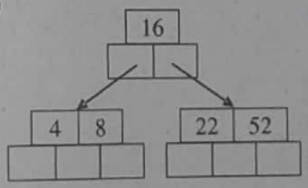

**Ques 39) Draw a B-tree of order 5, each node holding four values by inserting the following data 8, 16, 22, 52, 4, 32, 34, 30, 2, 38, 46, 40, 6, 24, 28, 50, 26**

**Ans:**

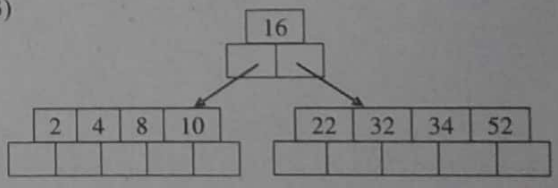
1)



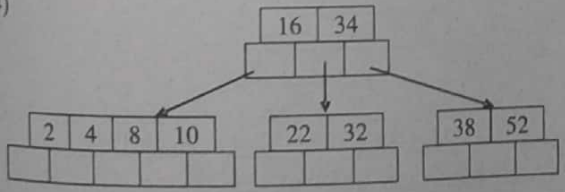

2)



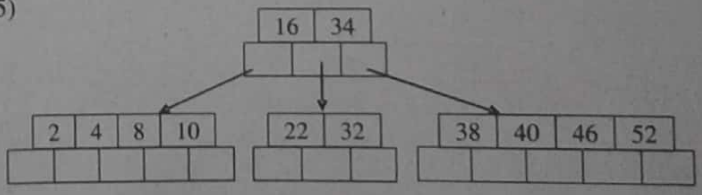

3)



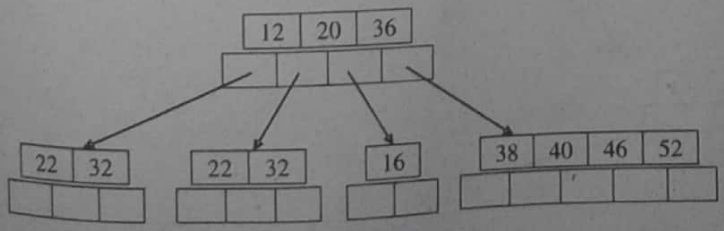

4)



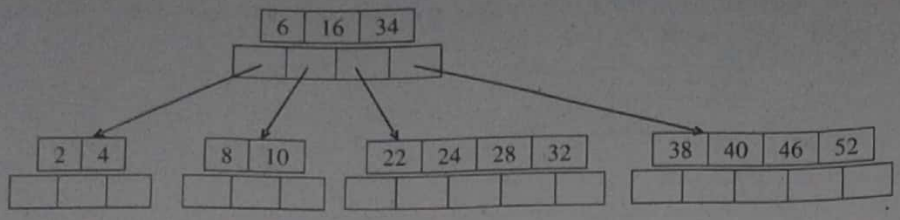

5)



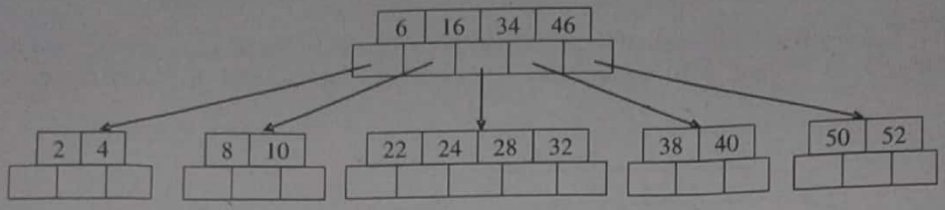

6)



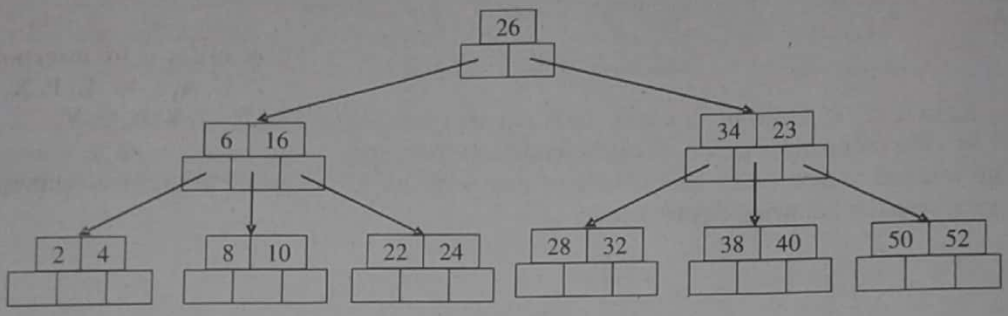
7)




8)



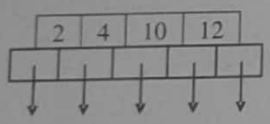

9)



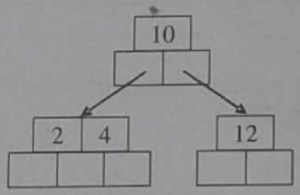

**Ques 40) Draw a B-tree by inserting the following data in a four-way tree; each node can contain three values and have 4 branches:**
**2, 10, 12, 4, 16, 22, 26, 36, 40, 14, 18**

**Ans:**

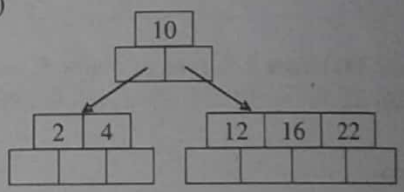
1)



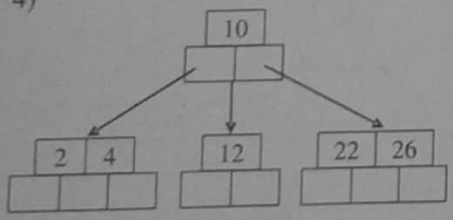

2)



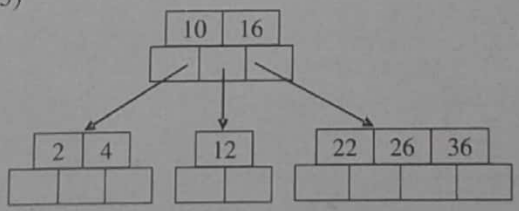

3)



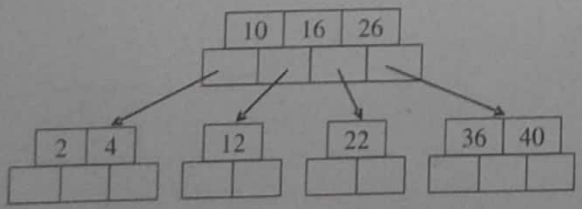

4)



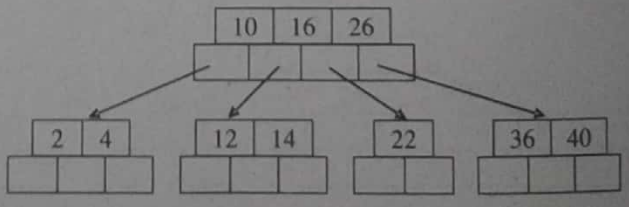

5)



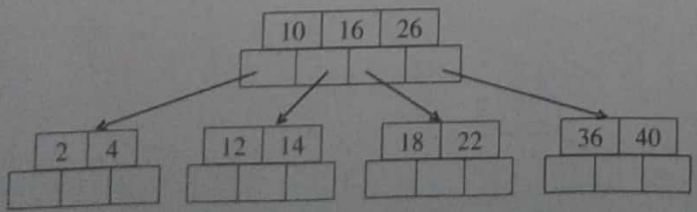

6)




7)



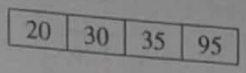

8)



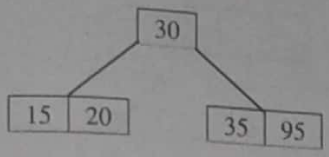
**Ques 41) Create B-Tree of order 5 from the following list of elements:**
**30, 20, 35, 95, 15, 60, 55, 25, 5, 65, 70, 10, 40, 50, 80, 45**

**Ans:** List of elements:
30, 20, 35, 95, 15, 60, 55, 25, 5, 65, 70, 10, 40, 50, 80, 45

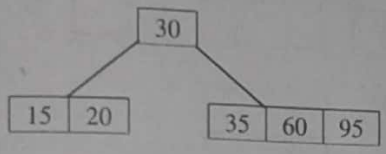
B-Tree of order 5
Insert 30, 20, 35, 95



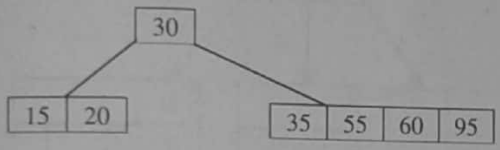

Insert 15:



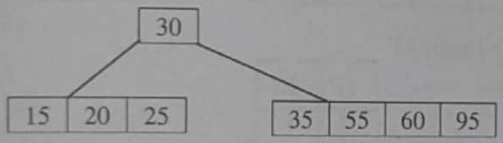

Insert 60:



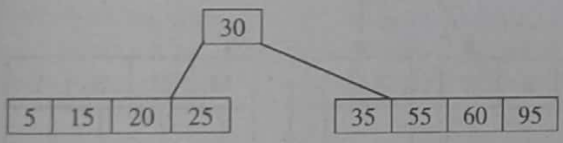

Insert 55:



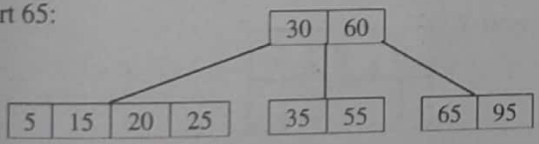

Insert 25:



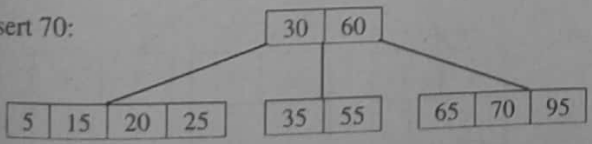

Insert 5:



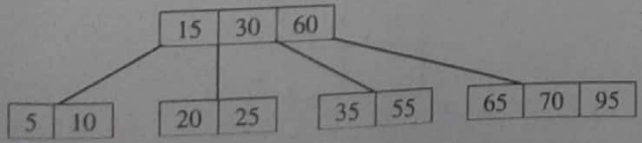

Insert 65:



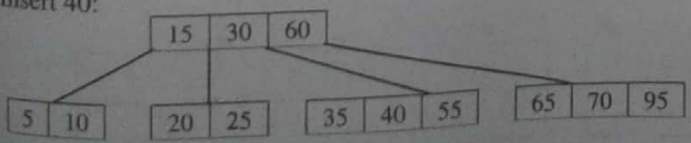

Insert 70:



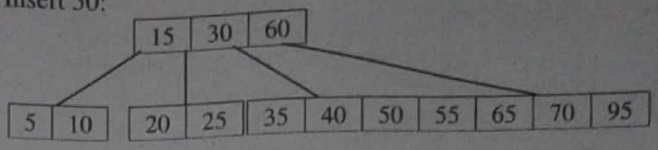

Insert 10:



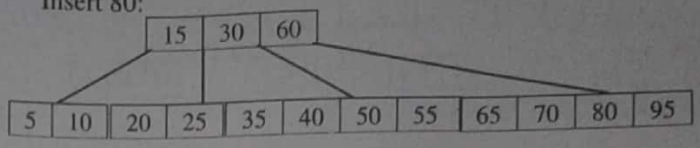

Insert 40:



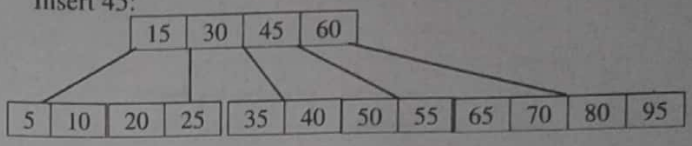

Insert 50:




Insert 80:

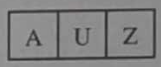



Insert 45:

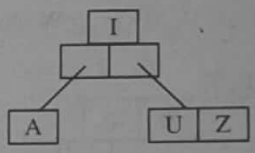



**Ques 42) Draw a B-tree of order 4 by insertion of the following keys in order: Z, U, A, I, W, L, P, X, C, J, D, M, T, B, Q, E, H, S, K, N, R, G, Y, F, O, V.**

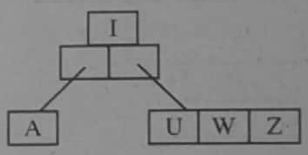
**Ans:** The order 4 means at most 3 keys are allowed.
Insert Z, U, A



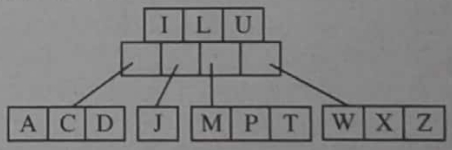

Insert I



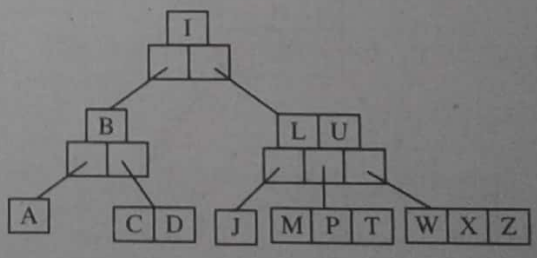

Insert W



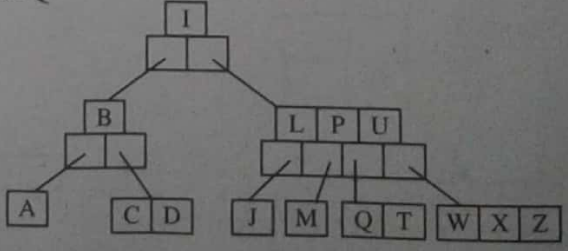

Insert L, P, X, C, J, D, M and T



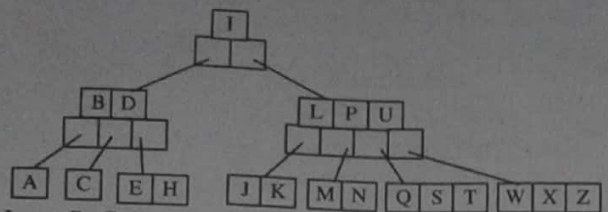

Insert B



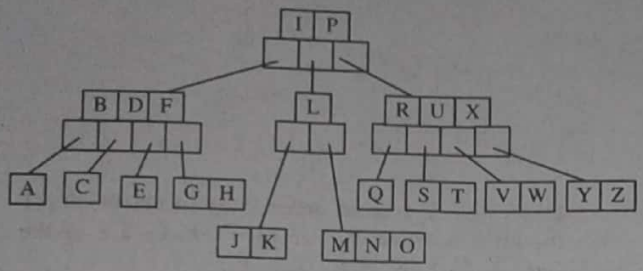

Insert Q



Insert E, H, S, K and N




Insert R, G, Y, F, O and V

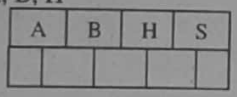



**Ques 43) Show the result of inserting keys – S, A, B, H, I, E, K, J, N, V, M, Y, Z, W, in order into an empty B-tree of order 5.**

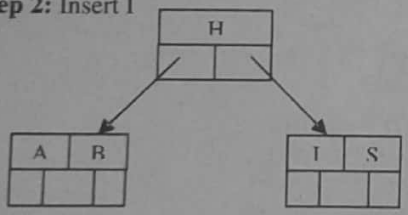
**Ans: B-Tree Insertion**

S, A, B, H, I, E, K, J, N, V, M, Y, Z, W Order 5

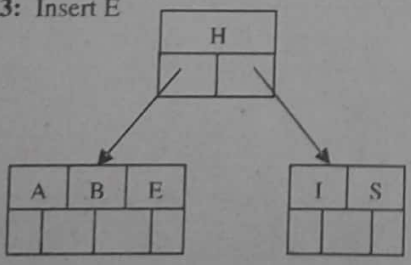
**Step 1:** Insert S, A, B, H



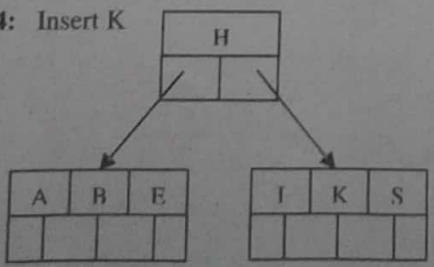

**Step 2:** Insert I



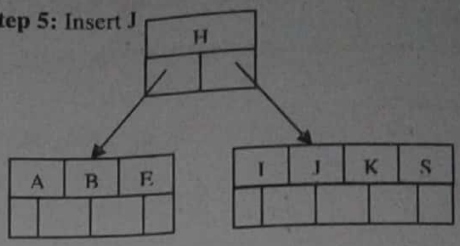

**Step 3:** Insert E



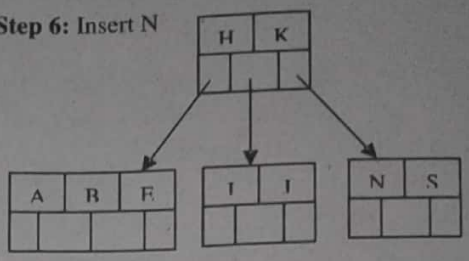

**Step 4:** Insert K



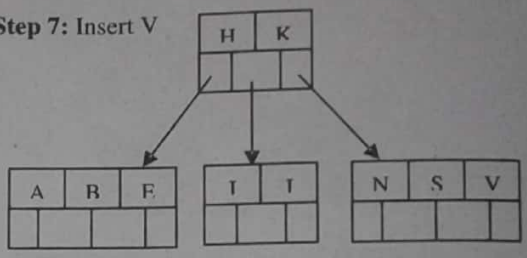

**Step 5:** Insert J



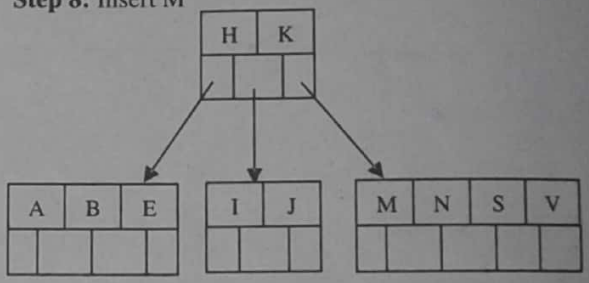

**Step 6:** Insert N



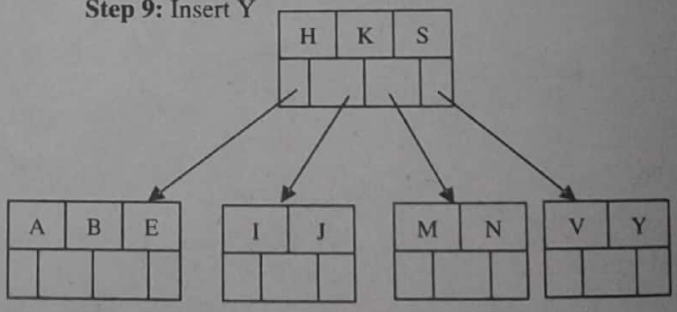

**Step 7:** Insert V



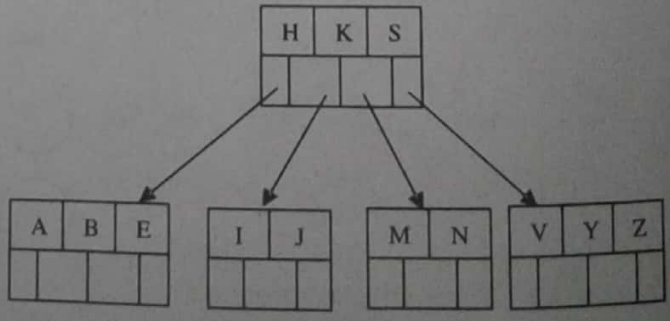

**Step 8:** Insert M



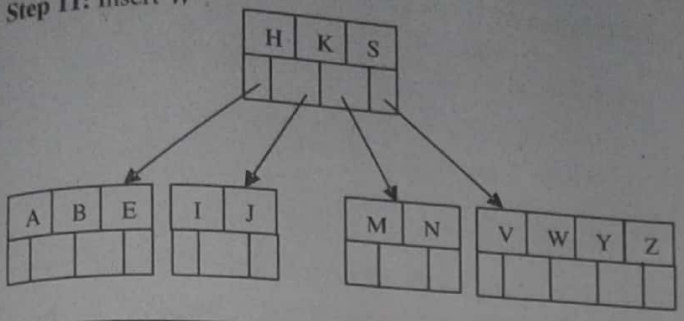

**Step 9:** Insert Y




**Step 10:** Insert Z



**Step 11:** Insert W




# SETS AND DISJOINTS SETS

**Ques 44) What is Sets and Disjoint sets?**

**Ans: Sets**

A set is a collection of distinct, unordered objects. Sets are typically collections of numbers, though a set may contain any type of data (including other sets). **For example,** the set of finite natural numbers (positive integers or counting numbers) A:

$$A = \{22, 26, 24, 25\}$$

Where A is set whose members are 22, 26, 24, 25.

The set of infinite natural numbers (positive integers, or counting numbers) N:

$$N = \{1, 2, 3, 4.....\}$$

A set and an element of a set concern with category of primary notions, for which it's impossible to formulate the strict definitions. So, we imply as sets usually collections of objects (elements of a set), having certain common properties. **For example,** a set of books in a library, a set of cars on a parking lot, a set of stars in the sky, a world of plants, a world of animals – these are examples of sets.

**Disjoint Sets**

A disjoint-set is a collection $S = \{S_1, S_2 \ldots S_k\}$ of distinct dynamic sets. Each set is identified by a member of the set, called **representative**. Given a set of elements, it is often useful to break them up or partition them into a number of separate, non-overlapping sets. A disjoint-set data structure is a data structure that keeps track of such a partitioning.

If we have two sets $S_x$ and $S_y$, $x \neq y$, such that $S_x = \langle 3,4,5,6,7 \rangle$ and $S_y = \langle 1,2 \rangle$ then these sets are called disjoint sets as there is no element which is common in both sets.

**Ques 45) What are the different disjoint set operations?**

**Ans: Disjoint Set Operations**

Let x denoting an object, then the disjoint operations supported are:

1) **MAKE-SET(x):** Create a new set with only x. Assume x is not already in some other set.
2) **UNION(x, y):** Combine the two sets containing x and y into one new set. A new representative is selected. If $x \in S_x$, $y \in S_y$, and then $\{S_x \cup S_y\}$. Unites the dynamic sets that contain x and y into a new set that is the union of these two sets.
   i) Representative of new set is any member of $S_x \cup S_y$, often the representative of one of $S_x$ and $S_y$.
   ii) Destroys $S_x$ and $S_y$ (since sets must be disjoint).
3) **FIND-SET(x):** Return the representative of the set containing x.

**Ques 46) What are the applications of disjoint set?**

**Ans: Applications of Disjoint-Set**

One of the many applications of disjoint-set data structure arises in determining the connected-components of an undirected graph. The procedure CONNECTED-COMPONENTS that follows the disjoint-set operations to compute the connected components of a graph. Once CONNECTED-COMPONENTS has been run as a pre-processing step, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component.

The set of vertices of a graph G is denoted by V[G], and the set of edges is denoted by E[G].

**Algorithm: CONNECTED-COMPONENTS (G)**

```
1) for each vertex v ∈ V[G]
2)     do MAKE-SET(v)
3) for each edge (u,v) ∈ E[G]
4)     do if FIND-SET(u) ≠ FIND-SET(v)
5)     then UNION(u,v)
```

**Algorithm: SAME-COMPONENT (u, v)**

```
1) if FIND-SET(u) = FIND-SET(v),
2)     then return TRUE
3) else return FALSE.
```

The procedure Connected-Components initially places each vertex v in its own set. Then for each edge (u, v) it unites the sets containing u and v. Connected-Components computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component.

If actually implementing **connected components** following points are noted:

i) Each vertex needs a handle to its object in the disjoint-set data structure,
ii) Each object in the disjoint-set data structure needs a handle to its vertex.

**For example,** consider the connected graph below:

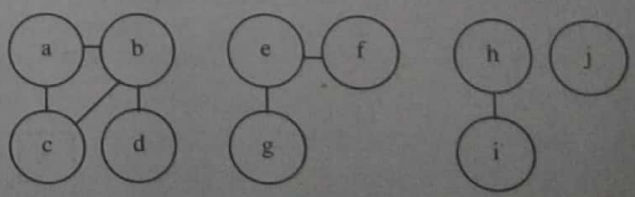



**Figure 2.22: Connected Graphs**

Collection of Disjoints Sets after each Edge of connected graph **shown above** is processed (**table 2.2**).

**Table 2.2**

| Edge Processed | Collection of Disjoint Sets | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Initial sets | {a} | {b} | {c} | {d} | {e} | {f} | {g} | {h} | {i} | {j} |
| (b, d) | {a} | {b, d} | {c} | | {e} | {f} | {g} | {h} | {i} | {j} |
| (e, g) | {a} | {b, d} | {c} | | {e, g} | {f} | | {h} | {i} | {j} |
| (a,c) | {a, c} | {b, d} | | | {e, g} | {f} | | {h} | {i} | {j} |
| (h,i) | {a, c} | {b, d} | | | {e, g} | {f} | | {h, i} | | {j} |
| (a,b) | {a, b, c, d} | | | | {e, f, g} | {f} | | {h | | {j} |
| (e,f) | {a, b, c, d} | | | | {e, f, g} | | | {h | | {j} |
| (b,c) | {a, b, c, d} | | | | {e, f, g} | | | {h | | {j} |

**Other Applications**

Disjoint set data structures have lots of applications. **For example, Kruskal's minimum spanning** tree algorithm relies on such a data structure to maintain the components of the intermediate spanning forest. Another application might be maintaining the connected components of a graph as new vertices and edges are added. In both these applications, one can use a disjoint-set data structure, where one keep a set for each connected component, containing that component's vertices.

**Ques 47) Proof the theorem that "A sequence of m MAKEWEIGHTEDSET operations and n WEIGHTEDUNION operations takes O(m + n log n) time in the worst case."**

**Ans: Proof**

Whenever the leader of an object x is changed by a WEIGHTEDUNION, the size of the set containing x increases by atleast a factor of two. By induction, if the leader of x has changed k times, the set containing x has atleast $2^k$ members. After the sequence ends, the largest set contains at most n members. (Why?) Thus, the leader of any object x has changed at most [lg n] times.

Since each **WEIGHTEDUNION** reduces the number of sets by one, there are m – n sets at the end of the sequence and at most n objects are not in singleton sets.

Since each of the non-singleton objects had O(log n) leader changes, the total amount of work done in updating the leader pointers is O(n log n).

The aggregate method now implies that each WEIGHTEDUNION has amortized cost O(log n).

**Ques 48) Discuss about the union-find operations on disjoint sets.**

**Ans: Union-Find Operations**

A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

1) **Union**: Combine or merge two sets into a single set.
2) **Find**: Determine which set a particular element is in. Also useful for determining if two elements are in the same set.

**Algorithm: Union**

```
union (i, j)
{
root1 = find (i);
root2 = find (j);
if (root1 != root2)
parent [root2] = root1; // attaching parent node
}
```

**Algorithm: Find**

```
find (i)
{   if (parent [i] < 0)
return i;
else
return parent [i] = find (parent [i]);}
```

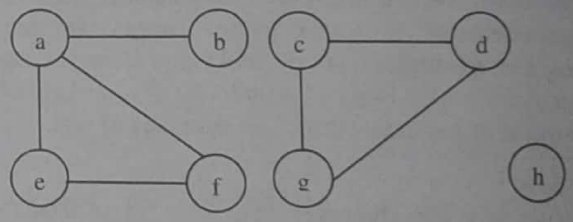
Usually the implementation of disjoint set is done using tree data structure.

if (find(i) = find(j)) then that means both the elements i and j belong to same set. Hence only one set name is returned.

**Analysis of Union-Find Algorithms**

The time complexity of union is O(n) as n time is required to scan the complete array of n elements. And the time complexity of find depends upon the height of the tree. If the height of tree is kept minimum then find algorithm will work more efficiently.

**For example,** one application of disjoint-set data structure is to determine the connected components of an undirected graph. **Figure 2.23** shows a graph with three connected components. The edges are added dynamically and we need to maintain the connected components as each edge is added. Let V denote the set of vertices and E the set of edges of the graph.



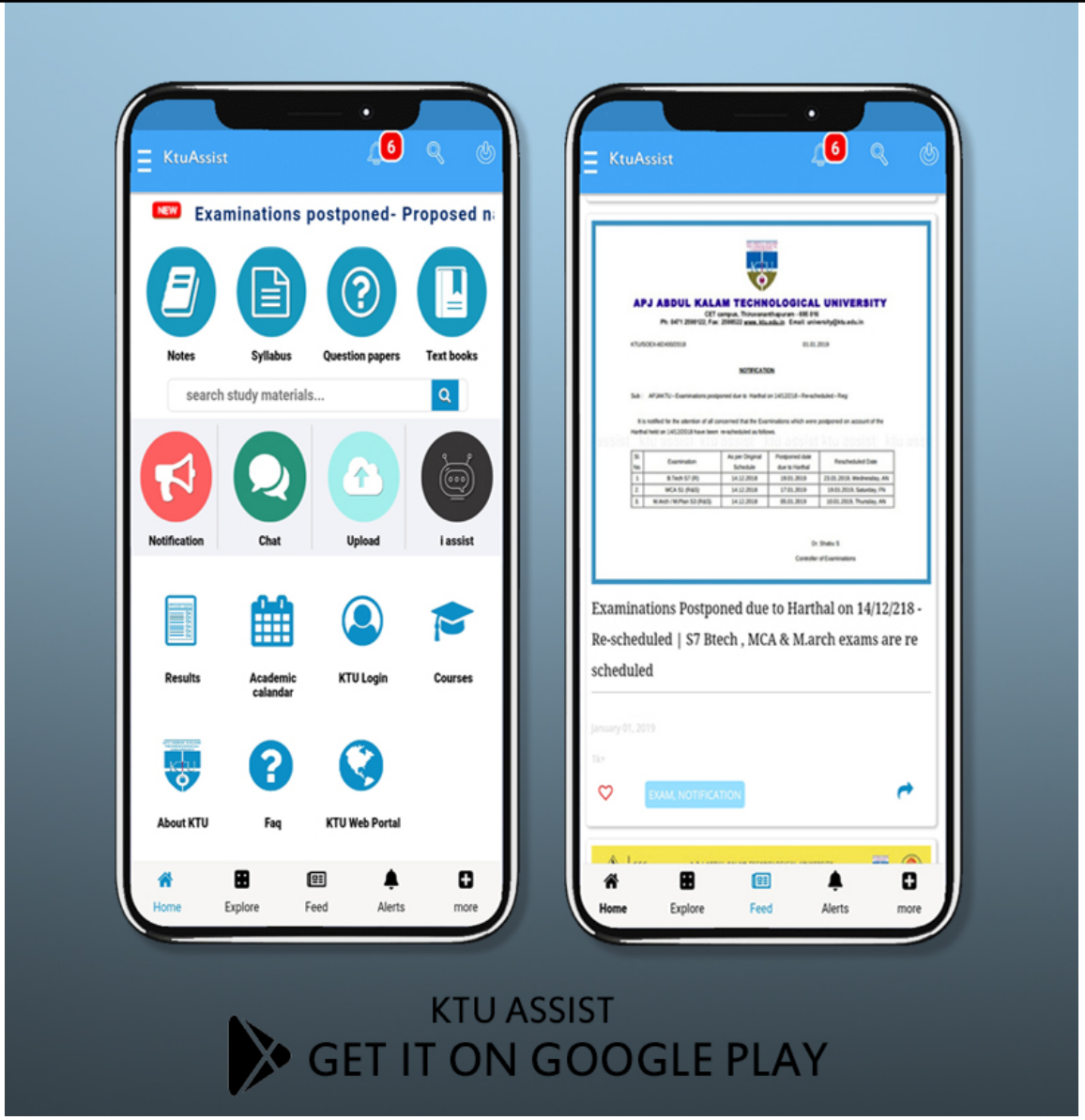

V = {a, b, c, d, e, f, g, h}
E = {(a, b), (a, e), (a, f), (e, f), (c, d), (c, g), (g, d)}

**Figure 2.23: Undirected Graph with Three Connected Components**

**The following steps illustrate the collection of disjoint sets after each edge is processed:**

Initial sets: {a} {b} {c} {d} {e} {f} {g} {h}
Process (a, b): {a, b} {c} {d} {e} {f} {g} {h}
Process (a, e): {a, b, e} {c} {d} {f} {g} {h}
Process (a, f): {a, b, e, f} {c} {d} {g} {h}
Process (e, f): {a, b, e, f} {c} {d} {g}{h}
Process (c, d): {a, b, e, f} {c, d} {g} {h}
Process (c, g): {a, b, e, f} {c, d, g} {h}
Process (g, d): {a, b, e, f} {c, d, g} {h}

Initially, each vertex is placed in its own set. Then, the edge (a, b) unites the sets containing a and b, edge (a, e) unites the sets containing a and e, and so on. After all the edges are processed, two vertices are in the same connected component if and only if the corresponding objects are in the same set.




# END

facebook.com/ktuassist

instagram.com/ktu_assist