Be sure with Vidyalankar

Since 1950

T.E. Sem. V [CMPN]

Advance Algorithm

Notes

Set - I

CONTENTS

Module No.	Topic	Page No.
•	Syllabus	(i)
1	Fundamental of Algorithms	1
2	Probabilistic Analysis and Randomized Algorithm	33
3	Advanced Data Structure	43
4	Maximum Flow	*
5	Computational Geometry	*
6	NP-Completeness and Approximation Algorithms	*

* will be distributed separately.

Scanned by CamScanner

Advance Algorithm

T.E. Sem. V [CMPN]

EVALUATION SYSTEM

	Time	Marks
Theory Exam	3 Hrs.	80
Internal Assessment (Two Test of 20 Marks each)	–	20 (Avg. of Two Test)

SYLLABUS

Module 1 Fundamental of Algorithms

Introduction

Complexity - complexity of recursive algorithms, finding complexity by tree method, master method, proving technique (contradiction, mathematical induction).

Amortized analysis - aggregate analysis, accounting analysis, potential analysis dynamic tables.

Module 2 Probabilistic Analysis and Randomized Algorithm

The hiring problem Indicator random variables Randomized algorithms Probabilistic analysis and further uses of indicator random variable.

Module 3 Advanced Data Structure

Introduction to trees and heap

Red-Black Trees : properties of red-black trees, Operations on Red-black trees.

Binomial Heaps : Binomial trees and binomial heaps, Operation on Binomial heaps.

Analysis of all above operations

Module 4 Maximum Flow

Flow networks, the ford Fulkerson method, max bipartite matching, push Relabel Algorithm, The relabel to front algorithm.

Module 5 Computational Geometry

Line Segment properties, Determining whether any pair of segment intersects, finding the convex hull, Finding the closest pair of points.

Module 6 NP-Completeness and Approximation Algorithms

NP-Completeness: NP-Completeness and reducibility, NP-Completeness proofs, NP-Complete problems-The vertex cover problem, The travelling salesman problem.

Mumbai University Question Paper Format :

1. Question paper will comprise of 6 questions, each carrying 20 marks.
2. Total four questions need to be solved.
3. Q.1 will be compulsory, based on entire syllabus wherein sub questions of 2 to 5 marks will be asked.
4. Remaining question will be randomly selected from all the modules.

❑ ❑ ❑ ❑ ❑

Scanned by CamScanner

Module

1

Fundamental of Algorithms

SYLLABUS Weightage : 19 Marks

Fundamental of Algorithms
Introduction

Complexity - complexity of recursive algorithms, finding complexity by tree method, master method, proving technique (contradiction, mathematical induction).

Amortized analysis - aggregate analysis, accounting analysis, potential analysis dynamic tables.

Help Line : For any query WhatsApp to 704 501 85 39 to get it Solved.

Scanned by CamScanner

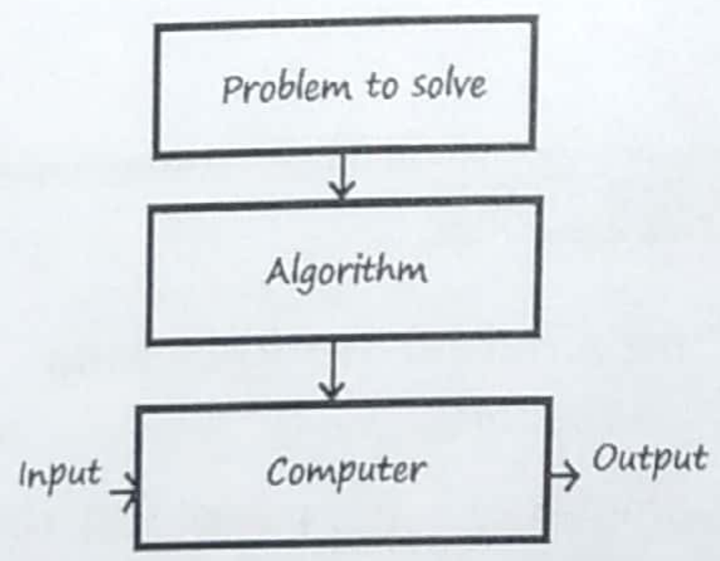
INTRODUCTION TO ANALYSIS OF ALGORITHM

Q.1 What is an algorithm?

Ans.: **Definition of algorithm:** An algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence that produce a definite output for a given set of input in a finite amount of time.

The term 'algorithm' is named after the ninth century Persian mathematician 'Al-Khowarizmi'. An algorithm is simply a set of rules used to perform some calculations, either manually or using a machine.

After understanding a problem statement, we must carefully create an algorithm to solve the problem. The algorithm is then converted to a programming language and the resulting program is fed into a computer. The computer then executes this program and gives us the desired output based on the inputs provided. This is shown in the diagram :



Q.2 Explain the various properties of Algorithms.

Ans.: Simply writing the sequence of instructions as an algorithm is not sufficient to accomplish certain task. It is necessary to have following properties associated with an algorithm:

- **Non-ambiguity:** Each step in an algorithm should be non-ambiguous. That means each instruction should be clear and precise. The instruction in an algorithm should not denote any conflicting meaning. This property also indicate the effectiveness of algorithm.
- **Range of input:** The range of input should be specified. This is because normally the algorithm is input driven and if the range of the input is not been specified then algorithm can go in an infinite state.
- **Multiplicity:** The same algorithm can be represented in several different ways. That means we can write in simple English the sequence of instructions or we can write it in the form of pseudo code. Similarly for solving the same problem we can write several different algorithms. For instance: for searching a number from the given list we can use sequential search or a binary search method. Here "searching" is a task and use of either a "sequential search method " or "binary search method" is an algorithm.
- **Speed:** The algorithms are written using some specific ideas (which is popularly known as logic of algorithm). But such algorithms should be efficient and should produce the output with fast speed.
- **Finiteness:** The algorithm should be finite. That means after performing required operations it should terminate.

Q.3 Explain the term 'Performance Analysis of Algorithms'.

Ans.: Efficiency of an algorithm can be in terms of time or space. Thus checking whether the algorithm is efficient or not means analysing the algorithm.

Now with these fundamental aspects of algorithm we will analyse the given algorithm. There is a systematic approach that has to be applied for analysing

Scanned by CamScanner

and given algorithm. This systematic approach is modelled by a framework called as Analysis framework.

Analysis Framework :

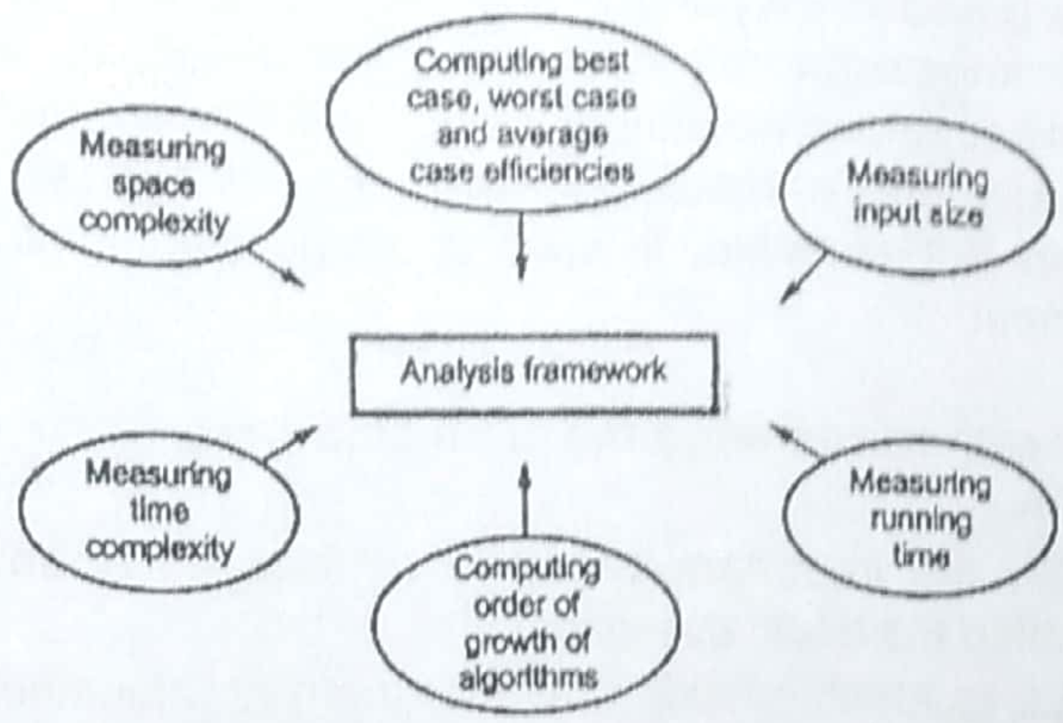


Fig. : Analysis framework.

The efficiency of an algorithm can be decided by measuring the performance of an algorithm. We can measure the performance of an algorithm by computing two factors.

1. Amount of time required by an algorithm to execute.
2. Amount of storage required by an algorithm.

This is popularly known as time complexity and space complexity of an algorithm.

Q.4 Explain the terms 'Space Complexity' and 'Time Complexity'.

Ans.: Space Complexity :

The space complexity can be defined as amount of memory required by an algorithm to run.

To compute the space complexity we use two factors: constant and instance characteristics. The space requirement S(p) can be given as

$$S(p) = C + Sp$$

When C is a constant i.e. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers. And Sp is a space dependent upon instance characteristics. This is a variable part whose space requirement depends on particular problem instance.

Time Complexity :

The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.

It is difficult to compute the time complexity in terms of physically clocked time. For instance in multiuser system, executing time depends on many factors such as:

- System load
- Number of other programs running
- Instruction set used
- Speed of underlying hardware

The time complexity is therefore given in terms of frequency count.

Frequency count is a count denoting number of times of execution of statement.

Scanned by CamScanner

Q.5 Explain how to measure input size.

Ans.: Measuring an Input Size :

It is observed that if the input size is longer, then usually algorithm runs for a longer time. Hence we can compute the efficiency of an algorithm as a function to which input size is passed as a parameter. Sometimes to implement an algorithm we require prior knowledge of input size. For example, while performing multiplication of two matrices we should know order of these matrices. Then only we can enter the elements of matrices. Sometimes the input size is taken as an approximate value. For example, in spell checking algorithms we can predict exact size of the input.

Q.6 Explain how to measure running time of an algorithm.

Ans.: From an algorithm :

- We first identify the important operation (core logic) of an algorithm. This operation is called the basic operation.
- It is not difficult to identify basic operation from an algorithm. Generally the operation which is more time consuming is a basic operation in the algorithm. Normally such basic operation is located in inner loop. For example in sorting algorithm the operation which is comparing the elements and then placing them at appropriate locations is a basic operation. The concept of basic operations can be well understood with the help of following example.

Table : Operations from input

Problem statement	Input size	Basic operation
Searching a key element from the list of n elements.	List of n elements.	Comparison of key with every element of list.
Performing matrix multiplication.	The two matrices with order n × n.	Actual multiplication of the elements in the matrices.
Computing GCD of two numbers.	Two numbers.	Division

- Then we compute total number of time taken by this basic operation. We can compute the running time of basic operation by following formula.
- Using this formula the computing time can be obtained approximately.

$$T(n) = C_{op}\, C(n)$$

Running tome of basic operation

Time taken by the basic operation to execute

Number of times the operation needs to be executed

Q.7 Explain the term "Order of Growth".

Ans.: Order of Growth: Measuring the performance of an algorithm in relation with the input size n is called order of growth. For example, the order of growth for varying input size of n is as given below.

Scanned by CamScanner

Table : Order of growth

n	log n	n log n	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

We will plot the graph for these values.

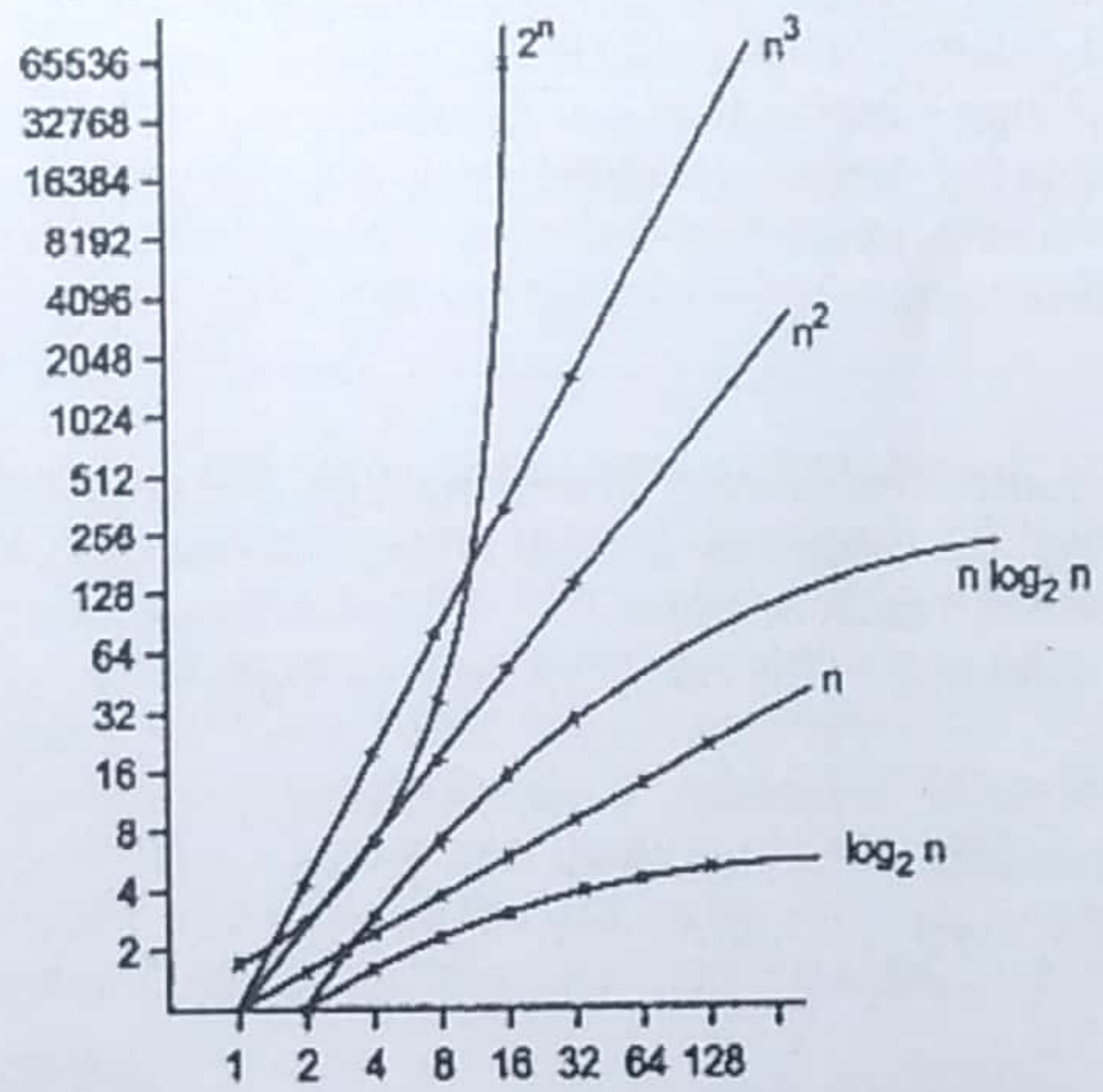


Fig.: Rate of growth of common computing time function

Q.8 Write a note on asymptotic notations. Explain the procedure to analyse and measure time complexity.

Ans.: Asymptotic Notations :

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.

Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time".

Various notations such as Ω, Θ and O used are called asymptotic notions. The most common way of calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to N approaches infinity. In general we can think of it like this:

```
statement;
```

Is constant. The running time of the statement will not change in relation to N.

```
for ( i = 0; i < n; i++ )
    statement;
```

Scanned by CamScanner

Is linear. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for ( i = 0; i < N; i++ ) {
    for ( j = 0; j < N; j ++ )
        statement;
)
```

Is quadratic. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while ( low <= high ) {
    mid = ( low + high ) / 2 ;
    if ( target < list [mid] )
        high = mid – 1;
    else if ( target > list [mid] )
        low = mid + 1;
    else break;
}
```

Is logarithmic. The running time of the algorithm is proportional to the number of times N can be divided by 2. This is because the algorithm divides the working area in half with each iteration.

```
void quicksort ( int list [ ], int left, int right )
{
    int pivot = partition ( list, left, right ) ;
    quicksort ( list, left, pivot – 1 ) ;
    quicksort ( list, pivot + 1, right ) ;
}
```

Is N * log (N). The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

In general, doing something with every element in one dimension is linear, doing something with every element in two dimensions is quadratic, and dividing the work area in half is logarithmic. There are other Big O measures such as cubic, exponential, and square root, but they're not nearly as common. Big O notation is described as O ( ) where is the measure. The quicksort algorithm would be described as O (Nlog(N)).

Q.9 Write the algorithm and derived the complexity of Binary Search algorithm.

Ans.: *Refer Answer of Q.8*

[M-17]

Q.10 Explain basic efficiency classes.

Ans.:

Table : Basic asymptotic efficiency classes.

Name of efficiency class	Order of growth	Description	Example
Constant	1	As input size grows the we get larger running time.	Scanning array elements.

Scanned by CamScanner

Logarithmic	logn	When we get logarithmic running time then it is sure that the algorithm does not consider all its input rather the problem is divided into smaller parts on each iteration.	Performing binary search operation.
Linear	n	The running time of algorithm depends on the input size n.	Performing sequential search operation.
nlogn	nlogn	Some instance of input is considered for the list of size n.	Sorting the elements using merge sort or quick sort.
Quadratic	n^2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cubic	n^3	When the algorithm has three nested loops then this type of efficiency occurs.	Performing matrix multiplication.
Exponential	2^n	When the algorithm has very faster rate of growth then this type of efficiency occurs.	Generating all subsets of n elements.
Factorial	n!	When an algorithm is computing all the permutations then this type of efficiency occurs.	Generating all permutations.

Q.11 An algorithm takes 0.5 ms for an input size of 100. How long will it take for an input size 500 if the running time is – (a) Quadratic (b) N log N.

Ans.: (a) When the function is quadratic (i.e. N^2), an input of N = 100 will generate

$N^2 = 100 \times 100$ lines of code = 10000 lines of code.

It has been given that these 10000 lines of code get executed in 0.5 ms.

Therefore, to execute one line of code, it takes 0.5/10000 = 0.00005 ms.

When the value of N = 500, the total number of lines of code will be:

$N^2 = 500 \times 500 = 25000$ lines of code.

To execute them, it will take a time $= 25000 \times 0.00005$ ms. = 1.25 ms.

Answer = 1.25 ms.

(b) When the function is N log N, an input of N = 100 will generate

N log N = 100 × log100 = 100 × 2 = 200 lines of code.

It has been given that these 200 lines of code get executed in 0.5 ms.

Therefore, to execute one line of code, it takes 0.5/200 = 0.0025 ms.

When the value of N = 500, the total number of lines of code will be:

N log N = 500 x log500 = 500 x 2.7 = 1350 lines of code.

To execute them, it will take a time $= 1350 \times 0.0025$ ms. = 3.375 ms.

Answer = 3.375 ms.

Q.12 Write an algorithm to find the sum of series and also find its time complexity: S = (Summation from i = 1 to i = N)i^2

Ans.: The algorithm is as follows,

1. START
2. dim i = 1
3. dim sum = 0
4. loop steps 5 and 6 till (i ! = N)
5. sum = sum + (i*i)
6. i = i + 1
7. STOP

Since the algorithm runs for N number of times, its complexity is O(N).

Scanned by CamScanner

Q.13 State the advantages and disadvantages of Recursion.

Ans.: Advantages :

(a) Although at most of the times a problem can be solved without recursion, but in some situations in programming, it is a must to use recursion. For example, a program to display a list of all files of the system cannot be solved without recursion.

(b) The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.

(c) Using recursion, the length of the program can be reduced.

Disadvantages :

(a) It requires extra storage space. The recursive calls and automatic variables are stored on the stack. For every recursive calls separate memory is allocated to automatic variables with the same name.

(b) If the programmer forgets to specify the exit condition in the recursive function, the program will execute out of memory. In such a situation user has to press Ctrl+ break to pause and stop the function.

(c) The recursion function is not efficient in execution speed and time.

If possible, we should try to solve a problem with iteration instead of recursion.

Q.14 Compare the time complexities of the following algorithms giving their complexities in terms of Big – O, Ω, Θ Quick Sort, Heap Sort, Shell Sort, Insertion Sort.

Ans.:

Algorithm/ Complexity	Quick Sort	Heap Sort	Shell Sort	Insertion Sort
Big – O	O (N log N)	O (N log N)	$O(N^2)$	$O(N^2)$
θ	θ (N log N)	θ (N log N)	$\theta(N^2)$	$\theta(N^2)$
Ω	Ω(N log N)	Ω (N log N)	$\Omega(N^2)$	$\Omega(N^2)$

Q.15 What is recursion? Write a routine to calculate Fibonacci series using it.

Ans.: Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.

The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

A common computer programming tactic is to divide a problem into sub-problems of the same type as the original, solve those sub-problems, and combine the results. This is often referred to as the divide-and-conquer method; when combined with a lookup table that stores the results of solving sub-problems (to avoid solving them repeatedly and incurring extra computation time), it can be referred to as dynamic programming or memorization.

Scanned by CamScanner

A recursive function definition has one or more base cases, meaning input(s) for which the function produces a result trivially (without recurring), and one or more recursive cases, meaning input(s) for which the program recurs (calls itself). For example, the factorial function can be defined recursively by the equations 0! = 1 and, for all n > 0, n! = n(n – 1)!. Neither equation by itself constitutes a complete definition; the first is the base case, and the second is the recursive case. Because the base case breaks the chain of recursion, it is sometimes also called the "terminating case".

The job of the recursive cases can be seen as breaking down complex inputs into simpler ones. In a properly designed recursive function, with each recursive call, the input problem must be simplified in such a way that eventually the base case must be reached.

Recursive algorithm for Fibonacci Series:

Algorithm: Fib (n)
Input: The number 'n'.
Output: n^{th} Fibonacci term is returned.

1) Start
2) if (n<=1) then
   return (n)
   x ← Fib (n–1)
   y ← Fib (n–2)
   return (x + y)
3) Stop

Q.16 Explain Big-O, Omega and Theta Notations with the help of diagrams. How do we analyse and measure time complexity of an algorithm?

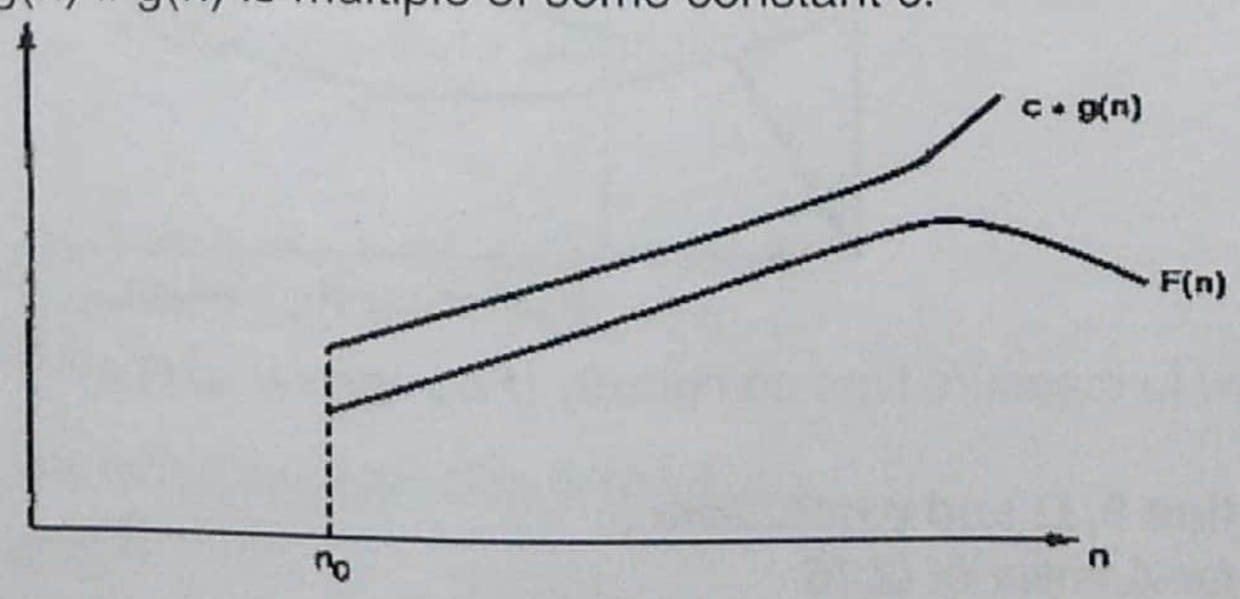
Ans.: Big oh Notation :

The Big oh notation is denoted by 'O'. It is a method of representing the upper bound of algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

Definition : Let F(n) and g(n) be two non-negative functions.
Let no and constant c are two integers such that n_o denotes some value of input and $n > n_0$. Similarly c is some constant such that c > 0. We can write

$$F(n) \leq c^*g(n)$$

then F(n) is big oh of g(n). It is also denoted as $F(n) \in O(g(n))$. In other words F(n) is less than g(n) if g(n) is multiple of some constant c.



F(n) ∈ O(g(n))

Scanned by CamScanner

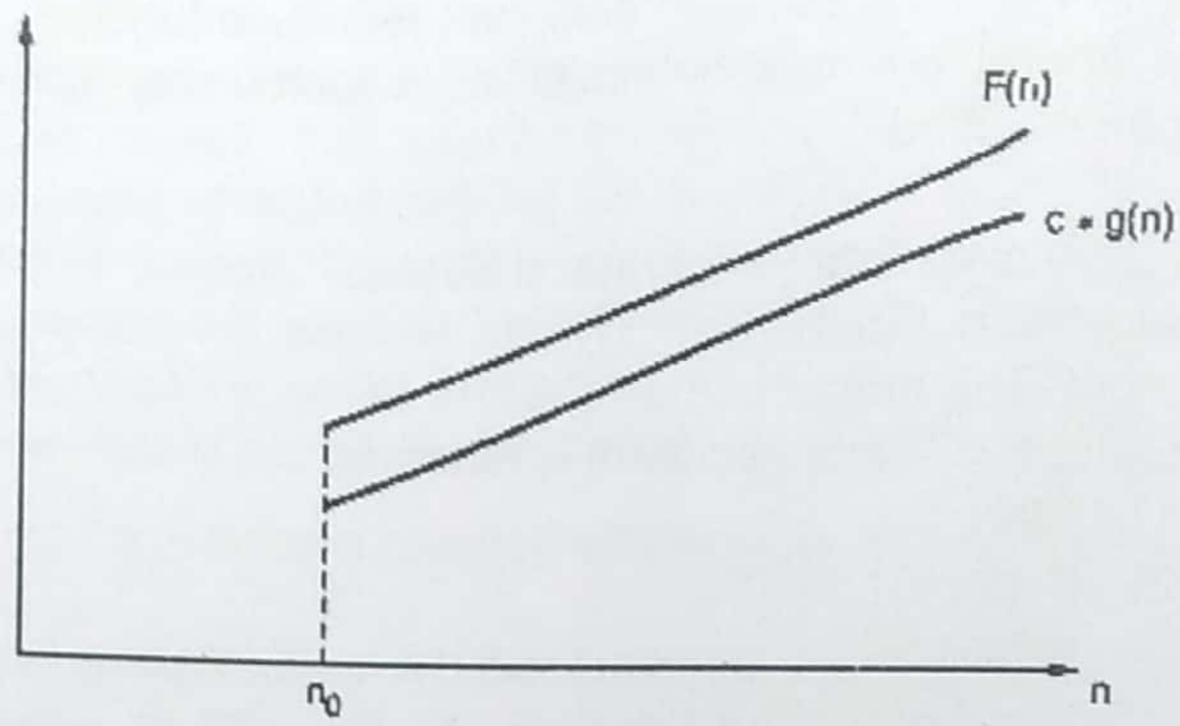
Omega Notation :

Omega notation is denoted by 'Ω'. This notation is used to represent the lower bound of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition : A function F(n) is said to be in Ω (g(n)) if F(n) is bounded below by some positive constant multiple of g(n) such that

$F(n) \geq c * g(n)$ For all $n \geq n_0$

It is denoted as $F(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.



F(n) ∈ Ω(g(n))

θ Notation :

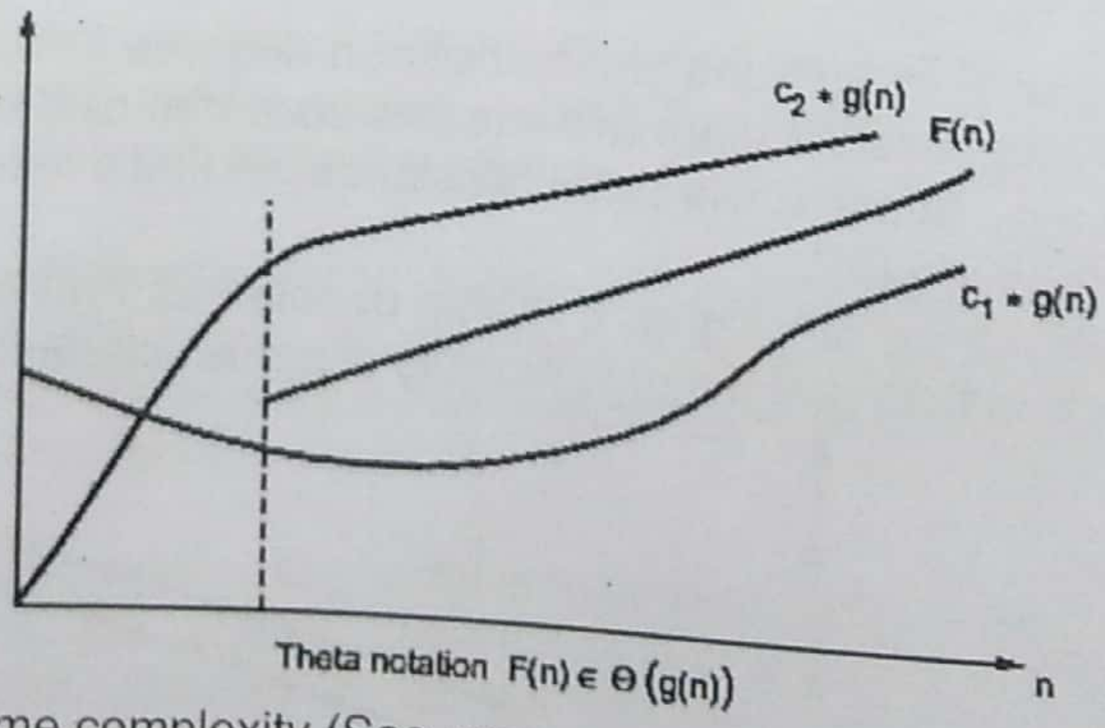
The theta notation is denoted by Θ. By this method the running time is between upper bounded and lower bound.

Definition : Let F(n) and g(n) be two non negative functions. There are two positive constants namely c_1 and c_2 such that

$c_1 \leq g(n) \leq c_2\, g(n)$

Then we can say that

$F(n) \in \Theta(g(n))$



Theta notation F(n) ∈ Θ(g(n))

How to measure time complexity (See answer to Q.4)

Q.17 Define 0, Ω and θ notations.

Ans.: *Refer Answer of Q.16*

Scanned by CamScanner

Q.18 Write short note on binary search and its complexity.

Ans.: *Refer Answer of Q.16*

Q.19 Arrange the following functions in increasing order :

$n, \log n, n^3, n^2 \log n, 2^n, n!$

Ans.: See the graph and table given in the answer to Question 07. Arrange these functions as given in that table. If this question is for four marks, you only need to write the correct order. If it is for greater number of marks, you need to draw the table and graph too.

Q.20 Explain forward and backward substitution using Factorial of a Number.

Ans.: The factorial of some number can be obtained by performing repeated multiplication.

For instance : If n = 5 then

Step 1 : n! = 5!

Step 2 : 4! * 5

Step 3 : 3! * 4 * 5

Step 4 : 2! * 3 * 4 * 5

Step 5 : 1! * 2 * 3 * 4 * 5

Step 6 : 0! * 1 * 2 * 3 * 4 * 5

Step 7 : 1 * 1 * 2 * 3 * 4 * 5 as 0! = 1

The above mentioned steps can be written in pseudocode form as-

```
Algorithm factorial (n)
/ /problem Description : This algorithm computes n! using
/ /recursive function
/ /Input : A non negative integer n
/ /Output :returns the factorial value.
    if (n = 0)
       return    1
    else
        return Factorial (n – 1)*n
```

Mathematical Analysis

Step 1 : The factorial algorithm works for input size n.

Step 2 : The basic operation in computing factorial is multiplication.

Step 3 : The recursive function call can be formulated as

$$F(n) = F(n-1) * n \qquad \text{where } n > 0$$

Then the basic operation multiplications is given as M(n). And M(n) is multiplication count to compute factorial(n).

$$M(n) = \underbrace{M(n-1)}_{\text{These multiplications are required to compute factorial } (n-1)} + \underbrace{1}_{\text{To multiply factorial } (n-1) \text{ by } n.}$$

Step 4 : In step 3 the recurrence relation is obtained.

$$M(n) = M(n-1) + 1$$

Scanned by CamScanner

Now we will solve recurrence using :

- Forward substitution

$$M(1) = M(0) + 1$$
$$M(2) = M(1) + 1 = 1 + 1 = 2$$
$$M(3) = M(2) + 1 = 2 + 1 = 3$$

- Backward substitution

$$M(n) = \underbrace{M(n-1)}_{} + 1 \qquad = \left[\underbrace{M(n-2)}_{} + 1\right] + 1 = M(n-2) + 2$$
$$= [M(n-3) + 1] + 1 + 1 \qquad = M(n-3) + 3$$

From the substitution methods we can establish a general formula as :

$$M(n) = M(n-i) + i$$

Now let us prove correctness of this formula using mathematical induction as follows.

Prove $M(n) = n$ by using mathematical induction.

Basis : Let $n = 0$ then

$M(n) = 0$

i.e. $M(0) = 0 = n$

Induction : If we assume $M(n-1) = n-1$ then

$$M(n) = M(n-1) + 1$$
$$= n - 1 + 1$$
$$= n$$

i.e. $M(n) = n$

Thus the time complexity of factorial function is $\Theta(n)$.

Solved Examples

Substitution Method :

Ex.1 $T(n) = T(n-2) + 1$

Ans.: Given: $T(1) = 1$

$\Rightarrow$ Guess $O(n)$,

$\therefore$ $T(n) \le c \cdot n \qquad \forall\, n \ge n_0$

I.H. $T(k-2) \le c \cdot (k-2)$

We have to show that,

$$T(k) \le c\,k$$
$$T(k) = T(k-2) + 1$$
$$\le c \cdot (k-2) + 1$$
$$\le c \cdot k - 2c + 1$$
$$\le c\,k$$

Finding constants,

$$c\,k - 2c + 1 \le c\,k$$
$$-2c + 1 \le 0$$
$$2c \ge 1$$
$$c \ge \frac{1}{2}$$

For $n = 3$,

$$T(n) \le c\,n$$
$$T(n-2) + 1 \le c\,n$$
$$T(1) + 1 \le 3c$$
$$2 \le 3c$$
$$c \ge \frac{2}{3}$$

$\therefore$ $T(n) = O(n) \qquad \forall\, n \ge 3,\ c = \frac{2}{3}$

Scanned by CamScanner

Ex.2 $T(n) = 2\,T(n/2) + n$

Ans.: Given $T(1) = 1$

$\Rightarrow$ Guess : $T(n) = 0\,(n \log n)$

By definition $T(n) \le c.\, n \log n$

I.H. $T\left(\frac{k}{2}\right) \le c.\ \frac{k}{2} \log\left(\frac{k}{2}\right)$

We have to show that, $T(k) \le ck.\ \log k$

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

$$\le 2.\left[c.\frac{k}{2}.\log\left(\frac{k}{2}\right)\right] + k$$

$$\le c.\ k \log\left(\frac{k}{2}\right) + k$$

$$\le c.k\,[\log k - \log 2] + k$$

$$\le ck \log k - ck + k \qquad \because \log 2 = 1$$

$$\le ck \log k$$

Finding constants,

$$ck \log k - ck + k \le ck \log k$$

$$-ck + k \le 0$$

$$ck \ge k$$

$$c \ge 1$$

At $n = 2$,

$$T(n) \le c\,.\,n \log n$$

$$2T\left(\frac{n}{2}\right) + n \le c\,.\,n \log n$$

$$2T(1) + c\,.\,2 \log 2$$

$$4 \le 2c$$

$$c \ge 2$$

$\therefore\ T(n) = 0\,(n \log n) \qquad \forall n \ge 2,\ c = 2$

Ex.3 $T(n) = T\,(n/2) + n$

Ans.: Given $T(1) = 1$

$\Rightarrow$ Guess $T(n) = O(n)$

By definition $T(n) \le c\,.\,n$

I.H. $T\left(\frac{k}{2}\right) \le c\,.\left(\frac{k}{2}\right)$

We have to show that $T(k) \le c\,.\,k$

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

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

$$\le ck$$

Finding constants,

$$\frac{ck}{2} + k \le ck$$

$$k \le \frac{ck}{2}$$

$$c \ge 2$$

At $n = 2$,

$$T(n) \le c\,.\,n$$

$$T\left(\frac{n}{2}\right) + n \le c\,.\,n$$

$$T(1) + 2 \le 2.\,c$$

$$3 \le 2c$$

Scanned by CamScanner

$c \geq \frac{3}{2}$

$\therefore \quad T(n) = O(n) \qquad \forall\, n \geq 2, c = 2$

Ex.4 **$T(n) = T(n/2) + 1$**

Ans.: Given $T(1) = 2$

$\Rightarrow$ Guess $T(n) = 0(\log n)$

By definition, $T(n) \leq c \log n$

I.H. $T\left(\frac{k}{2}\right) \leq c \,.\, \log\left(\frac{k}{2}\right)$

We have to show that, $T(k) \leq c \,.\, \log k$

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

$$\leq c \,.\, \log\left(\frac{k}{2}\right) + 1$$

$$\leq c\,[\log k - \log 2] + 1$$

$$\leq c \log k - c \log 2 + 1$$

$$\leq c \log k - c + 1$$

$$\leq c \log k$$

Finding constants,

$$c \log k - c + 1 \leq c \log k$$

$$-c + 1 \leq 0$$

$$c \geq 1$$

At $n = 2$,

$$T(n) \leq c.\log n$$

$$T\left(\frac{n}{2}\right) + 1 \leq c \,.\, \log n$$

$$T(1) + 1 \leq c \,.\, \log 2$$

$$2 \leq c$$

$$c \geq 2$$

$\therefore \quad T(n) = 0(\log n) \qquad \forall\, n \geq 2, c = 2$

Ex.5 **$T(n) = 2T(n/2) + 1$**

Ans.: Given : $T(1) = 1$

$\Rightarrow$ Guess $T(n) = O(n)$

By definition, $T(n) \leq c \,.\, n$

I.H. $T\left(\frac{k}{2}\right) \leq c\left(\frac{k}{2}\right)$

We have to show that $T(k) \leq c \,.\, k$

$$T(k) = 2\,T\left(\frac{k}{2}\right) + 1$$

$$\leq 2\,c\left(\frac{k}{2}\right) + 1$$

$$\leq c\,k + 1$$

It can't be shown as $\leq c\,k$

Scanned by CamScanner

Changing Guess : New Guess $T(n) \le c\,n - b$

I.H. $T\left(\frac{k}{2}\right) \le c\left(\frac{k}{2}\right) - b$

We have to show that $T(k) \le c\,k - b$

$$T(k) = 2T\left(\frac{k}{2}\right) + 1$$
$$\le 2\left[c\left(\frac{k}{2}\right) - b\right] + 1$$
$$\le c\,k - 2b + 1$$
$$\le c\,k - b$$

Finding constants,

$$ck - 2b + 1 \le ck - b$$
$$-2b + 1 \le -b$$
$$\therefore \quad b \ge 1$$

At $n = 2$,

$$T(n) \le c\,.\,n - b$$
$$2T\left(\frac{n}{2}\right) + 1 \le cn - 1 \quad \text{(Taking } b = 1\text{)}$$
$$2T(1) + 1 \le 2c - 1$$
$$3 \le 2c - 1$$
$$\therefore \quad 2c \ge 4$$
$$c \ge 2$$

$\therefore \quad T(n) = O(n) \qquad \forall\, n \ge 2, c = 2$

Ex.6 $\mathbf{T(n) = 8\,T(n/2) + n^2}$

Ans.: $T(1) = 1$

$\Rightarrow$ Guess $T(n) = O(n^3)$

By definition, $T(n) \le cn^3$

I.H. $T\left(\frac{k}{2}\right) = c\left(\frac{k}{2}\right)^3$

We have to show that, $T(k) \le ck^3$

$$T(k) = 8T\left(\frac{k}{2}\right) + k^2$$
$$\le 8c\left(\frac{k}{2}\right)^3 + k^2$$
$$\le 8c\,\frac{k^3}{8} + k^2$$
$$\le ck^3 + k^2$$

$ck^3 + k^2 \ne ck^3$ ... Mathematically

$\therefore$ Changing guess : $T(n) \le cn^3 - b$

I.H. : $T\left(\frac{k}{2}\right) \le c\left(\frac{k}{2}\right)^3 - b$

We have to show that $T(k) = ck^3 - b$

$$T(k) = 8T\left(\frac{k}{2}\right) + k^2$$

Scanned by CamScanner

$$\le 8\left[c\left(\frac{k}{2}\right)^3 - b\right] + k^2$$
$$\le ck^3 - 8b + k^2$$
$$\le ck^3 - b$$

Finding constants,

$$ck^3 - 8b + k^2 \le ck^3 - b$$
$$-8b + k^2 \le -b$$
$$7b \ge k^2$$
$$b \ge \frac{k^2}{7}$$

At n = 2,

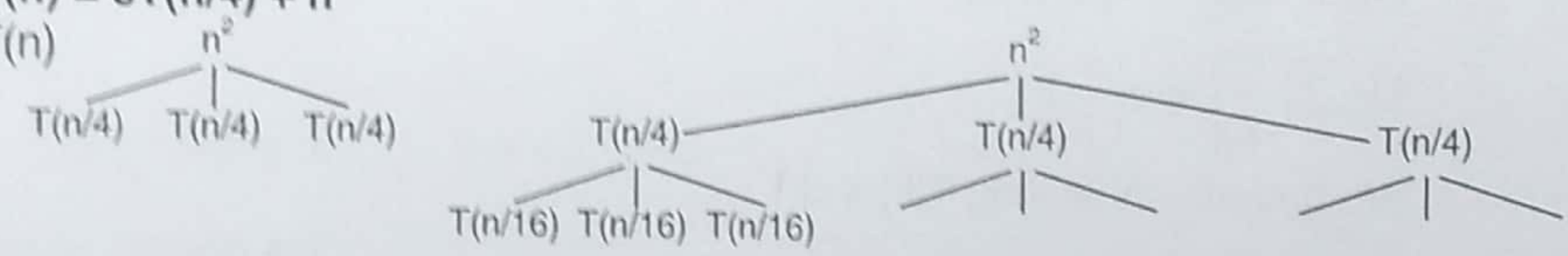
$$T(n) \le cn^3 - b$$
$$8T\left(\frac{n}{2}\right) + n^2 \le cn^3 - \frac{n^2}{7} \quad \text{... taking } b = \frac{n^2}{7}$$
$$8T(1) + 4 \le 8c - \frac{4}{7}$$
$$12 \le 8c - \frac{4}{7}$$
$$8c \ge 12 + \frac{4}{7}$$
$$8c \ge \frac{88}{7}$$
$$c \ge \frac{11}{7}$$

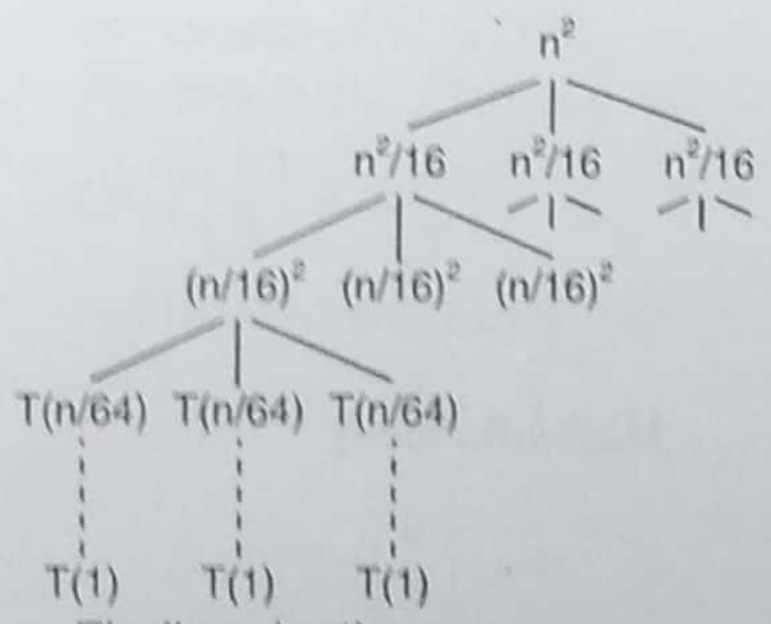
$\therefore \quad T(n) = O(n^3) \qquad \forall\, n \ge 2, c = 2$

Recursion Tree Method :

Ex.7 $T(n) = 3T(n/4) + n^2$

Ans.: T(n)





Level Cost	Problem size
n^2	n
$(3/16)n^2$	n/4
$(3/16)^2 n^2$	n/16
$(3/16)^3 n^2$	n/64

Finding depth,

Let depth = i

$\therefore$ Problem size at depth $= \dfrac{n}{4^i} = 1$

$\therefore \quad 4^i = n$

$i = \log_4 n$

Scanned by CamScanner

∴ No. of leaf nodes (node at level i) = $3^{\log_4 n} = 3^{\log_4 3}$

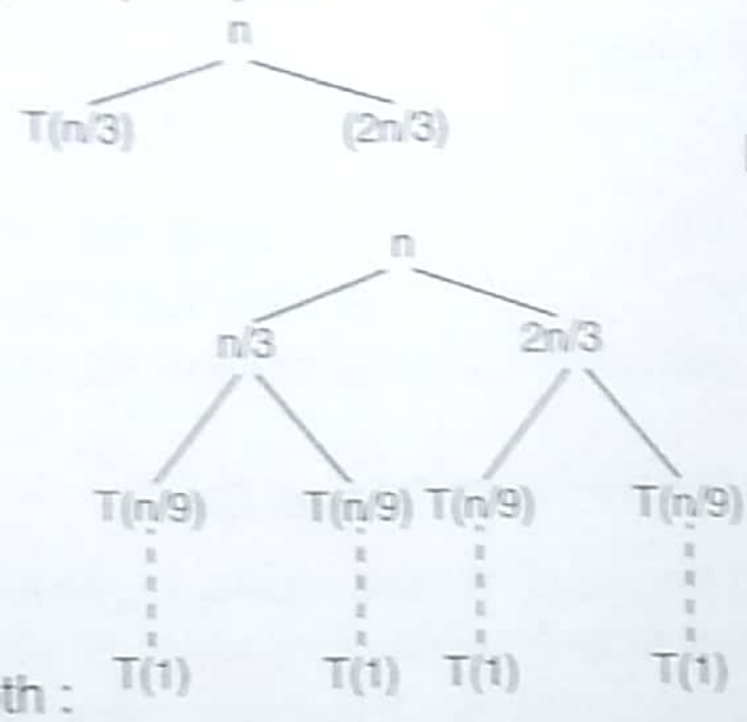
∴ Cost contribution from depth level $= \theta(n^{\log_4 3})$

∴ Total Cost = $\sum_{i=0}^{\log_{4-1} n} \left(\frac{3}{16}\right)^i n^2 + \theta(n^{\log_4 3})$

∴ Good guess = $O(n^2)$

Ex.8 $T(n) = T(n/3) + T(2n/3) + n$

Ans.: T(n)



Level Cost	Problem size
n	n
n	$\frac{n}{(3/2)}$
n	$\frac{n}{(3/2)^2}$

Finding depth :
Let depth = i

Size of problem at level i = $\frac{n}{(3/2)^i} = 1$

∴ $\left(\frac{3}{2}\right)^i = n$

∴ depth = i = $\log_{3/2} n$

∴ No. of leaf nodes (nodes at level i) $= 2^{\log_{3/2} n} = n^{\log_{3/2} 2}$

∴ Cost contribution from depth level $= \theta(n^{\log_{3/2} 2})$

Total cost = $n \times \log n + \theta(n^{\log_{3/2} 2})$

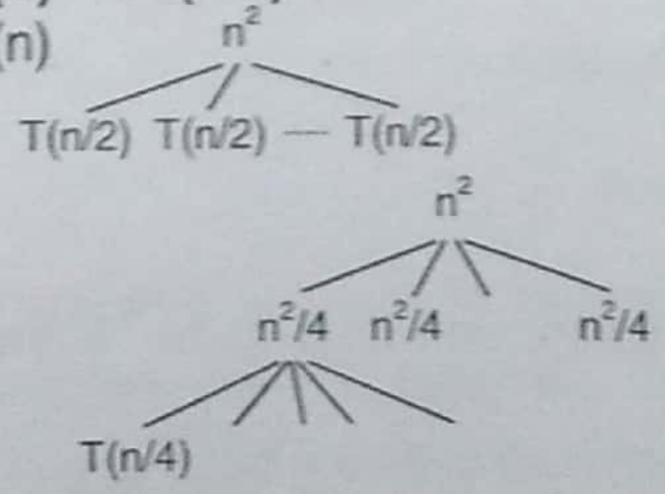
∴ Guess = $O(n \log n)$

Ex.9 $T(n) = 2T(n/2) + n$

Ex.10 $T(n) = 4T(n/2) + n^2$

Ex.11 $T(n) = 8T(n/2) + n^2$

Ans.: T(n)



Level Cost	Problem size
n^2	n
$2n^2$	n/2
$4n^2$	$n/2^2$

Finding depth :
Let i be depth

Scanned by CamScanner

Problem size at i $= \frac{n}{2^i} = 1$

$\therefore \quad 2^i = n$

$\therefore \quad i = \log_2 n$

No. of nodes at level, i $= 8^i = 8^{\log_2 n} = n^{\log_2 8}$

$= n^3$

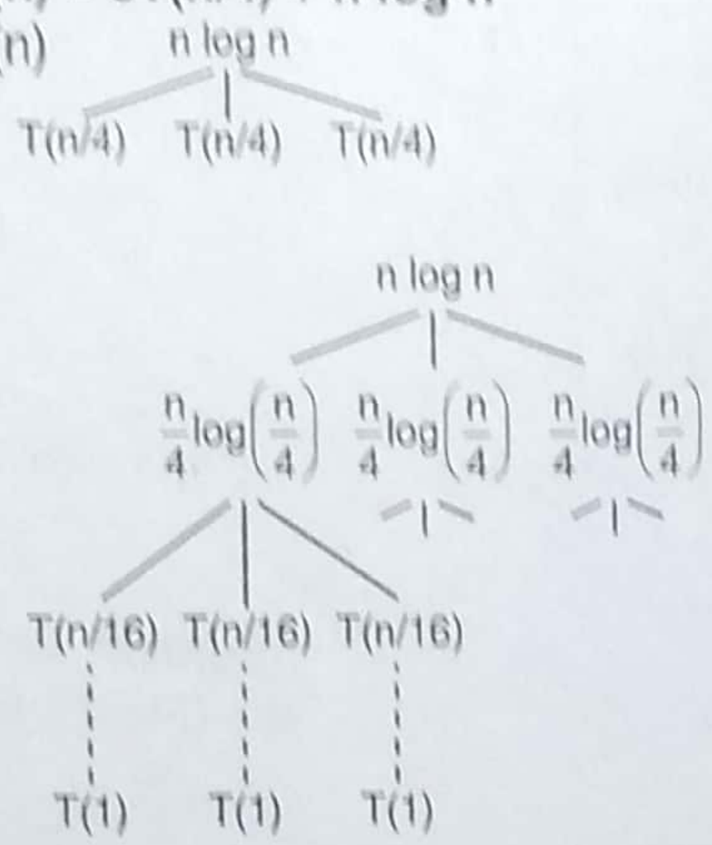
$\therefore$ Cost contribution from level, i $= \theta(n^3)$

$\therefore$ Total cost $= \sum_{i=0}^{\log_2 (n-1)} 2^i n^2 + \theta(n^3)$

$\therefore$ Guess $= \theta(n^3)$

Ex.12 $T(n) = 3T(n/4) + n \log n$

Ans.: T(n)



Level Cost	Problem size
n^2	n
$\frac{3}{4} n\log\left(\frac{n}{4}\right)$	$\frac{n}{4}$
$\left(\frac{3}{4}\right)^2 n\log\left(\frac{n}{4^2}\right)$	$\frac{n}{4^2}$

Finding depth :

Let depth be i

$\therefore$ Problem size at Level i, $\frac{n}{4^i} = 1$

$\therefore$ No. of nodes at Level i $= 3^i = 3^{\log_4 n} = 3^{\log_4 3}$

$\therefore$ Cost contribution from depth Level $= \theta(n^{\log_4 3})$

Total Cost $= \sum_{i=0}^{\log_4 (n-1)} \left(\frac{3}{4}\right)^i n \log\left(\frac{n}{4^i}\right) + \theta(n^{\log_4 3})$

$\therefore$ Good guess $= O(n \log n)$

Master Method :

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

$T(n) = a\,T(n/b) + f(n)$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then T(n) can be bounded asymptotically as follows.

Scanned by CamScanner

1. If $f(n) = O(n \log_b a - \varepsilon)$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n \log_b a)$.
2. If $f(n) = \Theta(n \log_b a)$, then $T(n) = \Theta(n \log_b a \,.\, \log n)$.
3. If $f(n) = \Omega(n \log_b a + \varepsilon)$ for some constant $\varepsilon > 0$, and if $a\, f(n/b) \leq c\, f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

PROVING TECHNIQUES

Different Types of Proving Techniques :

1) Proof by Contra Positive
2) Proof by contradiction
3) Proof by Mathematical Induction

1) Proof by Contrapositive :

- If A then B $\Rightarrow$ If not B then not A
- So if we want to prove first statement. It can also be done by proving second.

Solved Example

Ex.1 Let x be an integer prove that x^2 is an odd number iff x is an add number.

Ans.: 'iff' requires us to prove both directions of the implications.

$\therefore$ We must prove first that if x is an odd number then x^2 is an odd.

Then we should prove that if x^2 is odd then x is odd.

If x is odd then by definition, x can be written as 2k + 1 for some integer k.

$\therefore \quad x^2 = (2k+1)^2 \quad = 4k^2 + 4k + 1 \quad = 2(2k^2 + 2k) + 1$

$\therefore$ k is an integer, $2k^2 + 2k$ is also integer

$\therefore \quad x^2 = 2\ell + 1 \qquad$ where $\ell = 2k^2 + 2k$

$\therefore$ x^2 is odd integer.

Now we need to prove the second statement.

Second statement is logically equivalent to it's contrapositive.

So its sufficient to prove that if x is an even number then x^2 is even.

Suppose x is even, then x = 2k for some integer k,

$\therefore \quad x^2 = 4k^2 \quad = 2(2k^2)$

$\because$ k is integer, $2k^2$ is also integer.

$\because \quad x^2 = 2\ell \qquad$ where $\ell = 2k^2$

$\therefore$ x^2 is an even number.

2) Proof by Contradiction :

- In this, we assume the statement is not true and then derive a contradiction.

Solved Examples

Ex.1 Prove that $\sqrt{2}$ was irrational.

Ans.: Suppose $\sqrt{2}$ was rational.

$\therefore$ $\sqrt{2}$ can be written as m/n for some integer m and n.

$\therefore \quad \sqrt{2} = \dfrac{m}{n}$

Scanned by CamScanner

$\therefore \quad 2 = \dfrac{m^2}{n^2}$

$\therefore \quad 2 = 2n^2$

Now any square like x^2 must have even number of prime factors.

$\therefore$ m^2 must have even number of prime factors.

$\therefore$ Now n^2 would also have been even number of prime factors and 2 is also prime.

$\therefore$ $2n^2$ has odd number of prime factors which is contradiction as $m^2 = 2n^2$.

$\therefore$ Our initial assumption is wrong $\sqrt{2}$ must be irrational number.

Ex.2 Prove for all integer n, if $n^3 + 5$ is odd then n is even.

Ans.: Let n be any integer.

For contradiction, suppose that both $n^3 + 5$ and n are odd.

Let there exists integers j & k such that $n^3 + 5 = 2k + 1$, $n = 2j + 1$

$$\begin{aligned} \therefore \quad 2k + 1 &= n^3 + 5 \\ &= (2j + 1)^3 + 5 \\ &= 8j^3 + 3(2j)^2 .1 + 3(2j)\, 1^2 + 1^3 + 5 \\ 2k + 1 &= 8j^3 + 12j^2 + 6j + 5 \end{aligned}$$

$$\therefore \quad k - 4j^3 - 6j^2 - 3j = \frac{5}{2}$$

This is contradiction as $\dfrac{5}{2}$ is non-integer while $k - 4j^3 - 6j^2 - 3j$ is an integer by closure properties for integers.

$\therefore$ Our assumption $n^3 + 5$ is add then n is odd is false, so n must be even.

Ex.3 There exist no integers a and b for which 18a + 6b = 1.

Ans.: Assuming that integer a and b can be found for which 18a + 6b = 1.

Dividing by 6, we obtain,

$$3a + b = \frac{1}{6}$$

This is contradiction, since if 3a + b is an integer and can't be equal to $\dfrac{1}{6}$ by closure properties.

$\therefore$ There must be no integers a and b exist for which 18a + 6b = 1.

Ex.4 The sum of rational and irrational number is irrational.

Ans.: Rational number can be represented in $\dfrac{p}{q}$ form.

Let's assume that sum of rational & irrational number is rational.

$\therefore \quad \dfrac{p}{q} + x = \dfrac{m}{n}$ where x is irrational.

$\therefore \quad x = \dfrac{m}{n} - \dfrac{p}{q}$

$\therefore$ but that shows x is rational, by closure property.

$\therefore$ It's a contradiction.

$\therefore$ Sum of rational and irrational number is irrational.

Scanned by CamScanner

Ex.5 The negative of any irrational number is irrational.

Ans.: Suppose negative of any irrational number is rational.

$\therefore$ If x is irrational then $-x$ is rational, then $-x$ can be represented as $\frac{a}{b}$.

$\therefore \quad -x = \frac{a}{b}$

$\therefore \quad x = \frac{-a}{b}$

$\because$ $-a$ and b are integers and $b \neq 0$.

$\therefore$ x is ratio of two integers and hence x is rational.

But it contradicts our assumption.

Hence, negative of irrational is irrational.

Ex.6 There are infinite natural numbers.

Ans.: Suppose there are finite natural numbers.

$\therefore$ Let x be the last natural numbers.

$x \in N$

$x + 1 \in N$ and $x + 1 > x$

So it's a contradiction.

$\therefore$ There are infinite natural numbers.

Ex.7 Product of two even number is also even.

Ans.: Assume product of two even numbers is odd.

Let x and y be even numbers.

$\therefore \quad x = 2c \qquad c \in N$

$y = 2p \qquad p \in N$

$x \cdot y = (2c)(2p) = 4\,cp$

$= 2\,(2cp)$

It's a contradiction

Hence product of two even numbers is even.

Ex.8 There are infinitely many prime numbers.

Ans.: Suppose, there are finite prime numbers.

Let's assume that biggest prime number is N.

Let there be integer $K = 1 \times 2 \times 3 \times \ldots\ldots \times N$

$\therefore$ There would be integer $M = k + 1$

Divide M by 2 ..... remainder 1

Divide M by 3 ..... remainder 1

⋮ ⋮

Divide M by N ..... remainder 1

$\therefore$ Lowest factor $k + 1$ is prime number $> N$

It's a contradiction, hence there are infinitely many prime numbers.

3) Proof by Mathematical Induction :

- It's a mathematical proof technique which is used to prove that a given property holds for every natural number.

Scanned by CamScanner

- The method of induction requires two cases to be proved.
- The first case is called 'base case' which is used to prove that property holds for number 0 (say).
- The second case is called induction step. Which proves that if property holds for one natural number n then it holds for next natural number n + 1.

Solved Example

Ex.1 **Prove that $p(n) = 0 + 1 + 2 + \ldots n = \frac{n(n+1)}{2}$ for all natural numbers.**

Ans.: **Base Case:** Show that the statement holds for n = 0.
P(0) is easily seen to be time.

$$0 = \frac{0.(0+1)}{2}$$

Inductive step : Show that P(k) holds then also P(k + 1) holds.
Assume that P(k) holds,

$$\therefore \quad 0 + 1 + 2 + \ldots k = \frac{k(k+1)}{2}$$

We have to show that P(K+1) holds

i.e. $\quad 0 + 1 + 2 + \ldots k + (k + 1) = \frac{(k+1)(k+2)}{2}$

LHS $= 0 + 1 + 2 + \ldots k + (k + 1)$

$$= \frac{k \cdot (k+1)}{2} + (k+1) \qquad = \frac{k(k+1)+2(k+1)}{2} \qquad = \frac{(k+1)(k+2)}{2}$$

$= \text{RHS}$

$\therefore$ It holds true for P(k + 1).

Ex.2 **Prove that $1 + 3 + 5 + \ldots. + (2n - 1) = n^2$**

Ans.: Let's show for n = 1
LHS = 1
RHS = $(1)^2 =$ $\quad \therefore$ LHS = RHS

Assuming for n = k,
$1 + 3 + 5 + \ldots + (2k - 1) = k^2$
We have to show that for n = k + 1
$\therefore \quad 1 + 3 + 5 + \ldots + (2k - 1) + [2(k + 1)] = (k + 1)^2$

$$\text{LHS} = \frac{1+3+5+\ldots.+(2k-1)+(2k+1)}{k^2+2k+1}$$

$= (k + 1)^2$
$= \text{RHS}$
Hence proved.

Ex.3 **Prove that $3 + 7 + 11 + \ldots. + (4n - 1) = n(2n + 1)$**

Ans.: Base Case : n = 1
LHS = 3, RHS = (1) (2.1 + 1) = 3
$\therefore$ LHS = RHS

Scanned by CamScanner

Induction case ∴ Assume for $n = k$,

$$3 + 7 + 11 + \ldots + (4k - 1) = k(2k + 1)$$

We have to show that, $n = k + 1$

$$3 + 7 + 11 + \ldots + (4k - 1) + (4k + 3) = (k + 1)(2k + 3)$$

$$\text{LHS} = \frac{3 + 7 + 11 + \ldots + (4k - 1) + (4k + 3)}{k(2k + 1)(4k + 3)}$$

$= k(2k + 1)(4k + 3) \qquad = 2k^2 + k + 4k + 5$

$= 2k^2 + 5k + 3$

$\text{RHS} = (k + 1)(2k + 3) \qquad = 2k^2 + 3k + 2k + 3$

$= 2k^2 + 5k + 3$

∵ LHS = RHS

Hence proved.

Ex.4 Prove that $1^3 + 2^3 + \ldots + n^3 = \frac{n^2(n+1)^2}{4}$

Ans.: Base case

For $n = 1$, $1^3 = \frac{1^2(1+1)^2}{4} \frac{4}{4} = 1$

Induction case : assuming true for $n = k$

$$1^3 + 2^3 + \ldots k^3 = \frac{k^2(k+1)^2}{4}$$

We have to show that for $n = k + 1$

$$1^3 + 2^3 + \ldots k^3 + (k + 1)^3 = \frac{(k+1)^2[(k+1)+1]^2}{4}$$

$\text{LHS} = 1^3 + 2^3 + \ldots k^3 + (k + 1)^3$

$= \frac{k^2(k+1)^2}{4} + (k+1)^3 \qquad = \frac{k^2[k^2 + 2k + 1] + 4(k+1)^3}{4}$

$= k^4 + 2k^3 + k^2 + 4[k^3 + 3k^2 + 3k + 1]$

$= k^4 + 2k^3 + k^2 + 4k^3 + 12k^2 + 12k + 4$

$= k^4 + 6k^3 + 13k^2 + 12k + 4$

$\text{RHS} = \frac{(k+1)^2[(k+1)+1]^2}{4} \qquad = \frac{(k+1)^2(k+2)^2}{4} \qquad = \frac{(k+1)^2(k+2)^2}{4}$

$= [k^2 + 2k + 1][2k^2 + 4k + 4]$

$= k^4 + 4k^3 + 4k^2 + 2k^3 + 8k^2 + 8k + k^2 + 4k + 4$

$= k^4 + 6k^3 + 13k^2 + 12k + 4$

$= \text{LHS}$

Hence proved.

Ex.5 Prove that $1 + 2 + 2^2 + 2^3 \ldots + 2^{n-1} = 2^2 - 1$

Ans.: Base case : for $n = 1$

$$1 = 2^1 - 1 = 1$$

Inductive case : for $n = k$

$$1 + 2 + 2^2 + 2^5 + \ldots + 2^{k-1} = 2^k - 1$$

Scanned by CamScanner

We have to show for $n = k + 1$

$$1 + 2 + 2^2 + \ldots + 2^{k-1} + 2^k = 2^{k+1} - 1$$

LHS $= 1 + 2 + 2^2 + 2^5 \ldots + 2^{k-1} + 2^k$

$= 2^{k-1} + 2^k$

$= 2^{k+1} - 1$

$=$ RHS

Hence proved.

Ex.6 Prove that $9^2 - 2^n$ = is divisible by 7.

Ans.: Base case for $n = 1$

$9^1 - 2^1 = 7$ is divisible by 7

Inductive case for $n = k$

$9^k - 2^k = 7m$ for some $m \in N$

We have to show that, for $n = k + 1$

$9^{k+1} - 2^{k+1} = 7n$

We have, $9^k - 2^k = 7m$

$9^k = 7m + 2^k$

LHS $= 9^{k+1} - 2^{k+1}$

$= 9^k \cdot 9 - 2^k \cdot 2$

$= [7m + 2^k] \cdot 9 - 2^k \cdot 2$

$= 7m \cdot 9 + 2^k \cdot 9 - 2^k \cdot 2$

$= 7(9m) + 2^k (9 - 2)$

$= 7 \cdot (9m) + 7 \cdot 2^k$

$= 7[9m + 2^k]$

$= 7n$ where $n = 9m + 2^k$

$=$ RHS

Hence proved.

Ex.7 Prove that $8^n - 1$ is divisible by 7.

Ans.: Base case for $n = 1$, $8^1 - 1 = 7$

Inductive case $8^k - 1 = 7m$

We have to prove for $n = k + 1$, $8^{k+1} - 1 = 7n$

$8^k - 1 = 7m$

$\therefore \; 8^k = 7m + 1 \quad \ldots (1)$

LHS $= 8^{k+1} - 1 \quad = 8^k \cdot 8 - 1 \quad = [7m + 1] \cdot 8 - 1$

$= 7(8m) + 8 - 1 \quad = 7(8m) + 7 \quad = 7(8m + 1)$

$= 7n$

$=$ RHS

Hence proved.

Ex.8 Prove that $3^n < (n + 1)!$ for $n \geq 4$.

Ans.: Base case : $n = 4$, $3^4 = 81 \qquad (4 + 1)! = 100$

Inductive case : $n = k$, $3^k < (k + 1)!$

We have to show that for $n = k + 1$, $3^{k+1} < (k + 2)!$

We have, $3^{k+1} = 3^k \cdot 3$

$\leq (k + 1)!\, 3$

$\leq (k + 2)!$

$k \geq 4$

$4 \leq k$

$3 \leq k$

$3 \leq k + 2$

Scanned by CamScanner

$$3(k+1)! \leq (k+2)(k+1)!$$
$$3(k+1)! \leq (k+2)!$$

Hence proved.

Ex.9 Prove that $2^n \geq 1 + n$ for $n \geq 1$.

Ans.: Base case : $n = 1,\ 2^1 \geq 1 + 1$

Inductive case : for $n = k,\ 2^k \geq 1 + k$

We have to show that for $n = k + 1,\ 2^{k+1} \geq 1 + (k + 1)$

i.e. $2^{k+1} \geq 2 + k$

$$\begin{aligned} 2^{k+1} &= 2^k . 2 \\ &\geq (1 + k) . 2 \\ &\geq 2 + 2k \\ &\geq 2 + k \qquad (\because\ 2k \geq k \text{ for } k \geq 1) \end{aligned}$$

Hence proved.

Ex.10 Prove that $2^n \geq 2n$ for $n \geq 1$.

Ans.: Base case : $n = 1,\ 2^1 \geq 2 . 1$

Inductive case : for $n = k,\ 2^k \geq 2 . k$

We have to show that for $n = n + 1,\ 2^{k+1} \geq 2 . (k + 1)$

$$\begin{aligned} 2^{k+1} &= 2^k . 2 \\ &\geq (2 . k) . 2 \\ &\geq 4k \\ &\geq 2k + 2k \\ &\geq 2k + 2 \qquad (\because\ 2k \geq 2) \\ &\geq 2(k + 1) \end{aligned}$$

Hence proved.

AMORTIZED ANALYSIS

In this, the time required to perform sequence of operations is averaged over all operations performed.

Amortized analysis can be used to show that the average cost of an operation is small, if one averages over sequences of operations, even though a single operation within the sequence might be expensive.

Techniques used in Amortized Analysis :

1) Aggregate Analysis 2) Accounting Method 3) Potential Method

1) Aggregate Analysis :

- In this, we determine an upper bound T(n) on total cost of sequence of n operations.
- Average cost per operation is then, T(n)/n.
- The average cost would be have same amortized cost, so that all operations would have same amortized cost. Example, Stack operations.
- Fundamental stack operations are push (s, x) and Pop(s), each of these takes O(1).
- Now we add multiPop (s, k) operation which removes k top objects of stack S or Pops entire stack if contains fewer than k objects.

Scanned by CamScanner

```
Function Multipop (s, k)
1.  Start
2.  While is empty (s) ≠ true and k ≠ 0 do
    {
3.  Pop (s)
4.  k = k–1
    }
5.  Return
```

Analysis :

No. of iterations of while = min(n, k)

i.e. no. of objects popped off the stack

For each iteration of loop, one call is made to Pop function which is O(1)

∴ Total cost of Multipop = min(n, k)

Actual running time is linear function of this cost.

- Let us analyze sequence of n Push, Pop and Multipop operations on initially empty stack.
- Worst case cost of MultiPop operation in sequence is O(n), as stock size is atmost n.
- Using aggregate analysis, we can obtain a better upper bound that considers entire sequence of n operations.
- Although, a single MultiPop operation can be expensive, any sequence of n Push. Pop and Multipop operations, on an initially empty stack can cost at most O(n).
- Each object can be Popped at most once for each time it is pushed.
- Therefore, number of times Pop can be called on a non-empty stack, including calls within MultiPop is at most number of push operations which is at most n.

∴ Average cost of an operation is $\frac{O(n)}{n} = O(1)$

∴ Amortized cost of all three stack operations = O(1)

Incrementing a Binary Counter :

- Consider problem of implementing a k–bit binary counter that counts upward from 0.
- We use array A[0 ....k–1], storing x with LSB in A[0] and MSB in A[k–1]

∴ $x = \sum_{i=0}^{k-1} A[i].2^i$

```
Function increment (A, k)
1.  start
2.  i = 0
3.  While i < length [A] and A[i] = 1 do
    {
4.      A[i] = 0
5.      i = i + 1
    }
6.  if i < length [A] then
7.      A[i] = 1
8.  return
```

Scanned by CamScanner

- ◊ Single execution of increment ( ) takes $\theta(k)$ in worst case where A contains all 1's.
- ◊ Thus sequence of n 'increment' operations takes $O(nk)$ in worst case.
- ◊ We can tighten our analysis to yield a worst–case cost of $0(n)$ for sequences of n increments by observing that not all bits flips for each increment.
- ◊ A[D] → flips each times – n times

$A[1] \rightarrow \frac{n}{2}$ times

$A[2] \rightarrow \frac{n}{4}$ times

In general, $A[i] \rightarrow \frac{n}{2^i}$ times for i = 0 to log n

$$\therefore \text{ total number of flips} = \sum_{i=0}^{\log n}\left[\frac{n}{2^i}\right]$$

$$< n \sum_{i=0}^{\infty}\left[\frac{1}{2^i}\right]$$

$$= 2n$$

∴ Worst case for n increment operations = O(n)

∴ Average cost or Amortized cost per operation = $\frac{O(n)}{n} = O(1)$

2) Accounting Method :

- In this, we assign different charges to different operations, with some operations charged more or less than their actual cost.
- The amount charged for operation is called amortized cost.
- When amortized cost of operation exceeds actual cost then that difference is assigned to specific object sin data structure as credit.

  ∴ credit = amortized cost – actual cost
- Credit can help pay for later operations having amortized cost less than actual cost.
- Every operation can have different amortized cost.
- This method differs from aggregate analysis in which all operations have same amortized cost.
- For showing that worst case average cost per operation is smaller by analyzing with amortized cost then we must ensure that the total amortized cost of sequence of operations provides an upper bound on total actual cost of the sequence.
- C_i = actual cost, $\hat{C}_i$ = amortized cost

  then $\sum_{i=1}^{n}\hat{C}_i \geq \sum_{i=1}^{n} C_i$ for all sequences of n operations

  $$\therefore \text{ total credit} = \sum_{i=1}^{n}\hat{C}_i - \sum_{i=1}^{n} C_i$$
- Total credit associated with data structure must be non-negative at all times.

Scanned by CamScanner

Example :

Stack operations :

Push → 1

Pop → 1

MultiPop (k, s) → min(k, s)

- Let us assign following amortized cost
  Push → 2
  Pop → 0
  MultiPop → 0
- We have to show that, we can pay for any sequence of stack operations by charging the amortized cost.
- Let us use dollar bill to represent each unit of cost.
- Starting with empty stack, we can push a plate on stack by using 1 dollar to pay actual cost.
- We left with credit of 1 dollar which can be kept on top of the plate.
- At any point of time, every late on stack has a dollar of credit on it which serves as a prepayment for the cost of popping from stack.
- So during Pop operation don't charge anything and pay its actual cost using credit stored on plate.
- Even MultiPop operation need not be charged anything as it can be considered as series of pop operations only.
- Hence, each plate on stack has credit of 1 dollar and stack always has non-negative number of plates which ensures that credit is always non-negative.
- So, that amortized cost would be upper bound for total actual cost.
- For sequence of n Push, Pop and MultiPop operation,
  total amortized cost would be O(n) which is same as total actual cost.

Incrementing a Binary Counter

- In this, we analyze increment operation on a binary counter that starts at zero.
- Running time of increment operation is proportional to no. of bits flipped which we use as cost.

- For amortized analysis, let us change an amortized cost of 2 dollars to set a bit to 1.
- So for setting a bit, 1 dollar can be used to pay actual cost and place other dollar as credit which can be used to reset the bit.
- Increment procedure sets at most one bit & hence amortized cost of increment operation is at most 2 dollars.
- Number of 1's in counter never becomes negative and hence total credit always stays non–negative.
- Therefore, for n increment operations, the amortized cost is O(n) which bounds the total actual cost.

3) Potential Method

- It represents the prepaid work as "potential energy" or "potential" which can be released to pay for future operations.

Scanned by CamScanner

- It associates potential with perform n operations with an initial data structure as a whole rather than with specific objects within data structure.
- The potential method will perform n operations with an initial data structure D_0.
- C_i = actual cost of ith operation
  D_i = data structure that results after applying i^{th} operation to data structure D_{i-1}
- A potential function ϕ maps each data structure D_i to a real number $\phi(D_i)$ which is potential associated with D_i.
- Then, amortized cost $\hat{e}_i$ of ith operation with respect to potential function ϕ is defined as follows:

  $$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$
- Total amortized cost of n operations is given below :

  $$\sum_{i=1}^{n} \hat{C}_i = \sum_{i=1}^{n} C_i + \phi(D_i) - \phi(D_{i-1}) \quad = \sum_{i=1}^{n} C_i + \phi(D_n) - \phi(D_0)$$
- If $\phi(D_n) = \phi(D_0)$ then total amortized cost given an upper bound on total actual cost.
- If $\phi(Di) > \phi(D_0)$ then we guarantee that we pay in advance.
- Usually we define $\phi(D_0) = 0$ and then show that $\phi(D_i) \geq 0$ for all i.
- If $\phi(D_i) - \phi(D_{i-1})$ of ith operation is positive then amortized cost $\hat{C}_i$ represents an overcharge to i^{th} operation and potential of data structure increases.
- If this potential difference is negative then amortized cost represents an undercharge to i^{th} operation and decrease in potential pays for actual cost of the operation.
- Amortized cost depends upon the selection of potential function.

Example : Stack Operations

◊ Potential function ϕ is defined as no. of objects in the stack.

∴ For an empty stack,

$$\phi(D_0) = 0$$

◊ Since no. of objects in stack is never negative, hence stack D_i that results after i^{th} operation has non-negative potential.

Thus, $\phi(D_i) \geq 0$ (i.e. $\phi(D_0)$)

◊ Therefore, total amortized cost of n operations with respect to ϕ represents an upper bound on actual cost.

◊ Computing amortized cost of stack operations :

1. **Push :**

– if i^{th} operation on stack containing s objectives is Push operations, then the potential difference is as follows:

$$\phi(D_i) - \phi(D_{i-1}) = (s + 1) - s = 1$$

∴ Amortized cost of Push :

$$\hat{C}_i = C_i + [\phi(D_i) - \phi(D_{i-1})] = 1 + 1 = 2$$

2. **MultiPop :**

– Suppose i^{th} operation on stack is MultiPop(s, k) which causes k' = min(s, k) objects to be popped off the stack.

– Actual cost of operation is k' and potential difference is as follows :

$$\phi(D_i) - \phi(D_{i-1}) = -k'$$

Scanned by CamScanner

- ∴ Amortized cost of MultiPop operation

  $\hat{C}_i = C_i + [\phi(D_i) - \phi(D_{i-1})] \quad = k' - k' \quad = 0$
- Similarly, amortized cost of Pop is also zero.

  ∴ Amortized cost of each of three operations is 0(1) and total amortized cost of sequence of n operations is 0(n)
- Total amortized cost is upper bound on total actual cost.
- The worst case cost of n operations is 0(n)

3. **Incrementing Binary Counter :**
- Define potential of counter after i^{th} increment operation to be b_i i.e. no. of 1's in the counter after i^{th} operation.
- Let us compute amortized cost of increment operation.
- Suppose i^{th} increment operation resets t_i bits.
- The actual cost of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1.
- If $b_i = 0$, then i^{th} operation resets all k-bits so $b_{i-1} = t_i + 1$

  ∴ The potential difference is,

  $\phi(D_i) - \phi(D_{i-1}) \le (b_{i-1} - t_i + 1) - b_{i-1}$

  ∴ $\phi(D_i) - \phi(D_{i-1}) = 1 - t_i$

  ∴ Amortized cost of increment,

  $\hat{C}_i = C_i + [\phi(D_i) - \phi(D_{i-1})] \quad = (t_i + 1) + (1 - t_i) \quad = 2$
- If counter starts at zero, then $\phi(D_0) = 0$
- Since $\phi(D_i) \ge 0$ for all i, total amortized cost of sequence of n increment operations is an upper bound on total actual cost.
- So, the worst case cost of n increment operations is 0(n).

DYNAMIC TABLES

- When we don't know, how much space would be needed in advance then we allocate some space which gets filled with some operations.
- When table must be reallocated with larger size by copying all items from original table into large new table.
- Similarly, if many items are deleted then, it would be preferable to reallocate table with smaller size.
- This problem can be defined as expanding and contracting the table.
- Using amortized analysis, we can show that amortized cost of insertion and deletion is only 0(1), even though actual cost of an operation is large when it triggers expansion or contraction.
- We introduce a concept of load factor as follows:

$$\text{Load factor of non-empty table} = \frac{\text{No. of items stored}}{\text{Size of table}}$$

- For an empty table, load factor is taken as 1.
- Currently, we would restrict our discussion to expansion operation only.
- When an item is inserted into a full table, a table can be expanded by allocating a new table having double the original size.
- As the table needs to be contiguous, we need to allocate new table and copy the elements.
- The following algorithm represents insertion operation :

Scanned by CamScanner

```
Function table-insert (t, x)
1.  start
2.  if size (t) = 0 then
    {
3.      allocate table (t) with 1 slot
4.      size (t) = 1
    }
5.  if num (t) = size (t) then // table is full
    {
6.      allocate new table with 2 * size (t) slots
7.      copy all items from table (t) to new table
8.      free table (t)
9.      table (t) = new table
10.     size (t) = 2 * size (t)
    }
11. insert x into table (t)
12. num (t) = num (t) + 1
13. return
```

- Let us analyze sequence of n table insert operations on an initially empty table.
- If C_i is cost of ith operation then
  $C_i = 1$ if elementary insertion is done
  $C_i = i$ if expansion is done
- Worst case of an operation is 0(n) and hence for n–operations it would be $0(n)^2$.
- This bound is not tight as cost of expansion is not required for every operation.
- Specifically, i^{th} operation needs an expansion only when i – 1 is an exact power of 2.
- Using aggregate analysis, we can show that the cost of each operation is 0(1).
- Cost of ith operation is given as follows :

$$C_i \begin{cases} = i & \text{if } i-1 \text{ is power of 2} \\ = 1 & \text{otherwise} \end{cases}$$

∴ total cost of n table insert operations

$$\sum_{i=1}^{n} Ci \le n + \sum_{j=0}^{\log n} 2^j$$
$$\le n + 2n$$
$$= 3n$$

Therefore, total cost of n operations is 0(n).

Scanned by CamScanner

1	2	3	4	5	6	7	8	9							

Insert	Operations
1	1
2	2 + 1 + 1
3	4 + 2 + 1
4	1
5	8 + 4 + 1
6	1
7	1
8	1
9	16 + 8 + 1

insert → 1 * size + 1
insert → 3

$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow \ldots\ldots \rightarrow 2N$

Expansion = 3 * (1+2+…+2N) + log 2N
= 3 * [4N–1] + log 2N
≈ 12 N

Elementary Insert = N
∴ total = 13N
∴ Complexity = 0(N)

❑❑❑❑❑

Module

2

Probabilistic Analysis and Randomized Algorithm

SYLLABUS

Weightage : 18 Marks

Probabilistic Analysis and Randomized Algorithm
The hiring problem Indicator random variables Randomized algorithms Probabilistic analysis and further uses of indicator random variable.

Help Line : For any query WhatsApp to 704 501 85 39 to get it Solved.

Scanned by CamScanner

HIRING PROBLEM

- Suppose that we need to hire a new office assistant and we decide to use an employment agency.
- The employment agency will send one candidate each day. We will interview that person and then decide to either hire that person or not.
- We must pay the employment agency a small fee to interview an applicant.
- To actually hire an applicant is more costly, since we must fire our current office assistant and pay a large hiring fee to the employment agency.
- We decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, we will fire the current office assistant and hire the new applicant.
- We are willing to pay the resulting price of this strategy, but wish to estimate what that price will be.
- The procedure HIRE-ASSISTANT, given below, expresses this strategy for hiring in pseudocode.
- It assumes that the candidates for the office assistant job are numbered 1 through n.
- The procedure assumes that we are able to, after interviewing candidate i, determine if candidate i is the best candidate we have seen so far.
- To initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

```
HIRE-ASSISTANT(n)
best = 0   // candidate 0 is a least-qualified dummy candidate
for i = 1 to n do
    {
        interview candidate i
        if candidate i is better than candidate best then
            best =i
        hire candidate i
    }
```

- We are not concerned with the running time of HIRE-ASSISTANT, but instead with the cost incurred by interviewing and hiring.
- In either case, we are counting the number of times certain basic operations are executed. Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h.
- Let m be the number of people hired.
- Then the total cost associated with this algorithm is $O(nc_i + mc_h)$.
- No matter how many people we hire, we always interview n candidates and thus always incur the cost nc_i associated with interviewing.
- We therefore concentrate on analyzing mch, the hiring cost. This quantity varies with each run of the algorithm.
- The hiring problem models how often we update our notion of which element is currently winning.

Worst-case analysis :

- In the worst case, we actually hire every candidate that we interview.
- This situation occurs if the candidates come in increasing order of quality, in which case we hire n times, for a total hiring cost of $O(nc_h)$.

Scanned by CamScanner

- It might be reasonable to expect, however, that the candidates do not always come in increasing order of quality.
- In fact, we have no idea about the order in which they arrive, nor do we have any control over this order.
- Therefore, it is natural to ask what we expect to happen in a typical or average case.

Probabilistic analysis :

- Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm.
- Sometimes, we use it to analyze other quantities, such as the hiring cost in procedure HIRE-ASSISTANT.
- In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs.
- Then we analyze our algorithm, computing an expected running time.
- The expectation is taken over the distribution of the possible inputs. Thus we are averaging the running time over all possible inputs.
- We must be very careful in deciding on the distribution of inputs. For some problems, it is reasonable to assume something about the set of all possible inputs, and we can use probabilistic analysis as a technique for designing an efficient algorithm and as a means for gaining insight into a problem. For other problems, we cannot describe a reasonable input distribution, and in these cases we cannot use probabilistic analysis.
- For the hiring problem, we can assume that the applicants come in a random order.
- We assume that we can compare any two candidates and decide which one is better qualified.
- We can therefore rank each candidate with a unique number from 1 through n, using rank(i) to denote the rank of applicant i , and adopt the convention that a higher rank corresponds to a better qualified applicant.
- The ordered list <rank(1), rank(2), . . . , rank(n)> is a permutation of the list <1, 2, . . . , n>. Saying that the applicants come in a random order is equivalent to saying that this list of ranks is equally likely to be any one of the n! permutations of the numbers 1 through n.
- Alternatively, we say that the ranks form a uniform random permutation; that is, each of the possible n! permutations appears with equal probability.

Indicator Random Variable

In order to analyze many algorithms, including the hiring problem, we will use indicator random variables. Indicator random variables provide a convenient method for converting between probabilities and expectations. Suppose we are given a sample space S and an event A. Then the indicator random variable I {A} associated with event A is defined as

$$I\{A\} = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$$

As a simple example, let us determine the expected number of heads that we obtain when flipping a fair coin. Our sample space is S = {H, T}, and we define a random variable Y which takes on the values H and T, each with probability 1/2.

We can then define an indicator random variable X_H, associated with the coin coming up heads, which we can express as the event Y = H. This variable counts the number of

Scanned by CamScanner

heads obtained in this flip, and it is 1 if the coin comes up heads and 0 otherwise. We write

$$X_H = I\{Y = H\} = \begin{matrix} 1 & \text{if } Y = H \\ 0 & \text{if } Y = T \end{matrix}$$

The expected number of heads obtained in one flip of the coin is simply the expected value of our indicator variable X_H:

$$\begin{aligned} E[X_H] &= E[I\{Y = H\}] \\ &= 1 \cdot \Pr\{Y = H\} + 0 \cdot \Pr\{Y = T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 \end{aligned}$$

Thus the expected number of heads obtained by one flip of a fair coin is 1/2.
The expected value of an indicator random variable associated with an event A is equal to the probability that A occurs.

Indicator random variables give us a simple way to arrive at the result of when we compute the number of heads in n coin flips.
We can let Xi be the indicator random variable associated with the event in which the ith flip comes up heads.
Letting Yi be the random variable denoting the outcome of the ith flip, we have that

$$X_i = I\{Y_i = H\}$$

Let X be the random variable denoting the total number of heads in the n coin flips, so that

$$X = \sum_{i=1}^{n} X_i$$

We wish to compute the expected number of heads, so we take the expectation of both sides of the above equation to obtain

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right]$$

The left side of the above equation is the expectation of the sum of n random variables.
The expectation of the sum is equal to the sum of the expectations of the n random variables.

We now can easily compute the expected number of heads:

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n}\left(\frac{1}{2}\right) = \frac{n}{2}$$

Analysis of the hiring problem using indicator random variables :

Returning to the hiring problem, we now wish to compute the expected number of times that we hire a new office assistant.
In order to use a probabilistic analysis, we assume that the candidates arrive in a random order.
Let X be the random variable whose value equals the number of times we hire a new office assistant.
We could then apply the definition of expected value from equation to obtain

$$E[X] = \sum_{x=1}^{n} x \Pr\{X = x\}$$

Scanned by CamScanner

But this calculation would be cumbersome. We shall instead use indicator random variables to greatly simplify the calculation.

To use indicator random variables, instead of computing E [X] by defining one variable associated with the number of times we hire a new office assistant, we define n variables related to whether or not each particular candidate is hired.

In particular, we let X_i be the indicator random variable associated with the event in which the ith candidate is hired.

Thus, $X_i = I$ {candidate i is hired} = 1 if candidate i is hired
0 if candidate i is not hired

and $X = X_1 + X_2 + \cdots + X_n$

$E[X_i] = Pr$ {candidate i is hired}

and we must therefore compute the probability that lines 5–6 of HIRE-ASSISTANT are executed.

Candidate i is hired, in line 5, exactly when candidate i is better than each of candidates 1 through i – 1. Because we have assumed that the candidates arrive in a random order, the first i candidates have appeared in a random order.

Any one of these first i candidates is equally likely to be the best-qualified so far. Candidate i has a probability of 1/i of being better qualified than candidates 1 through i – 1 and thus a probability of 1/i of being hired.

We conclude that,

$$E[X_i] = \frac{1}{i}$$

Now we can compute E[X] :

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n}\left(\frac{1}{i}\right) = \ln n + O(1)$$

Even though we interview n people, we actually hire approximately log n of them, on average.

Therefore, the total hiring cost of HIRE-ASSISTANT has a total running cost of $O(c_h.\log n)$ which is a significant improvement over the worst case hiring cost of $O(n.c_h)$.

THE BIRTHDAY PARADOX

- How many people must there be in a room before there is a 50% chance that two of them were born on the same day of the year?
- The answer is surprisingly few.
- The paradox is that it is in fact far fewer :
  - than the number of days in a year, or
  - even half the number of days in a year

An analysis using indicator random variables :

- We use indicator random variables to provide a simple but approximate analysis of the birthday paradox.

Scanned by CamScanner

- For each pair (i, j) of the k people in the room, de ne the indicator random variable X_{ij}, for $1 \le i < j \le k$, by

  $X_i = l$ {i and j have the same birthday}
  $= 1$ {i and j have the same birthday}
  $= 0$ otherwise
- Once birthday b_i for is chosen, the probability that b_j is chosen to be the same day is 1/n, where n = 365

  $E[X_{ij}]$ = Pr {i and j have the same birthday} = 1/n
- Let X be a random variable counting the number of pairs of individuals having the same birthday

  $$X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}$$
- Taking expectations of both sides and applying linearity of expectation, we obtain

  $$E[X] = E\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} E[X_{ij}] = \frac{k}{2}\frac{1}{n} = \frac{k(k-1)}{2n}$$
- When $k(k-1) \ge 2n$, the expected number of pairs of people with the same birthday is at least 1.
- Thus, if we have at least $\sqrt{2n} + 1$ individuals in a room, we can expect at least two to have the same birthday.
- For n = 365, if = 28, the expected number of pairs with the same birthday is

  $$\frac{(28 \,.\, 27)}{(2 \,.\, 365)} \approx 1.0356$$
- With at least 28 people, we expect to find at least one matching pair of birthdays.
- Mars has n = 687 days, need k = 38 aliens
- Analysis using only probabilities gives a different exact number of people, but same asymptotically: $\Theta(\sqrt{n})$

BALLS AND BINS

- Consider tossing identical balls randomly into b bins, numbered 1,2, … , b
- Tosses are independent, and on each toss the ball is equally likely to end up in any bin.
- The probability that a tossed ball lands in any given bin is 1/b.
- The ball-tossing process is a sequence of Bernoulli trials with a probability 1/b of success ≡ the ball falls in the given bin.
- How many balls fall in a given bin?
  - The number of balls that fall in a given bin follows the binomial distribution b(k; n, 1/b)
  - If we toss n balls, the expected number of balls that fall in the given bin is n/b.
- How many balls must we toss, on the average, until a given bin contains a ball?
  - The number of tosses until the given bin receives a ball follows the geometric distribution with probability 1/b and
  - the expected number of tosses until success is 1/(1/b) = b
- How many balls must we toss until every bin contains at least one ball?
  - Call a toss in which a ball falls into an empty bin a "hit"
  - We want to know the expected number n of tosses required to get b hits.

Scanned by CamScanner

- We can partition the n tosses into stages.
- The ith stage consists of the tosses after the $(i-1)^{st}$ hit until the ith hit.
- The first stage consists of the first toss, since we are guaranteed to have a hit when all bins are empty.
- During the ith stage, $i-1$ bins contain balls and $b-i+1$ bins are empty.
- For each toss in the ith stage, the probability of obtaining a hit is $(b-i+1)/b$.
- n_i is the number of tosses in the ith stage.
- The number of tosses required to get b hits is

$$n = \sum_{i=1}^{b} n_i$$

- Each n_i has a geometric distribution with probability of success $(b-i+1)/b$.

$$E[n_i] = \frac{b}{b-i+1}$$

$$E[n_i] = E\left[\sum_{i=1}^{b} n_i\right] = \sum_{i=1}^{b} E[n_i] = \sum_{i=1}^{b} \frac{b}{b-i+1} = b\sum_{i=1}^{b} \frac{1}{i}$$

$$= b\,(\ln b + 0(1))$$

- By harmonic series :
- It therefore takes approximately b $\ln$ b tosses before we can expect that every bin has a ball.
- This problem is also known as the coupon collector's problem, which says that a person trying to collect each of b different coupons expects to acquire approximately b $\ln$ b randomly obtained coupons in order to succeed.

RANDOMIZED ALGORITHMS

Q.1 Write a short note on Randomized Algorithms.

Ans.: A randomized algorithm is an algorithm which employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables.

One has to distinguish between algorithms that use the random input to reduce the expected running time or memory usage, but always terminate with a correct result (Las Vegas algorithms) in a bounded amount of time, and probabilistic algorithms, which, depending on the random input, have a chance of producing an incorrect result (Monte Carlo algorithms) or fail to produce a result either by signaling a failure or failing to terminate.

In the second case, random performance and random output, the term "algorithm" for a procedure is somewhat questionable. In the case of random output, it is no longer formally effective. However, in some cases, probabilistic algorithms are the only practical means of solving a problem.

Consider the problem of finding an 'a' in an array of n elements.

Input : An array of $n \geq 2$ elements, in which half are 'a's and the other half are 'b's.

Scanned by CamScanner

Output: Find an 'a' in the array.

We give two versions of the algorithm, one Las Vegas algorithm and one Monte Carlo algorithm.

Las Vegas algorithm :

```
findiangA_LV (array A, n)
begin
    repeat
        Randomly select one element out of n elements.
    until 'a' is found
end
```

This algorithm succeeds with probability 1. The run time of a single call varies and can be arbitrarily large, but the expected run time over many calls is (n).

Monte Carlo algorithm:

```
findingA_MC (array A, n, k)
begin
i = 0
repeat
    Randomly select one element out of n elements.
    i = i + 1
until i = k or 'a' is found
end
```

If an 'a' is found, the algorithm succeeds, else the algorithm fails.

Q.2 Explain Randomized algorithm with example.

Ans.: *Refer Answer of Q.1*

In this, instead of assuming a distribution of inputs, we impose a distribution.
For hiring problem in particular, before running the algorithm, we randomly permute the candidates in order to enforce the property that every permutation is equally likely.

This modification does not change our expectation of hiring a new office assistant roughly log n times.

The algorithm here is deterministic; for any particular input, the number of times a new office assistant is hired will always be the same.

Furthermore, the number of times we hire a new office assistant differs for different inputs, and it depends on the ranks of the various candidates.

Since this number depends only on the ranks of the candidates, we can represent a particular input by listing, in order, the ranks of the candidates, i.e., <rank(1), rank(2), . . . , rank(n)>.

Given the rank list A1 = <1, 2, 3, 4, 5, 6, 7, 8, 9, 10>, a new office assistant will always be hired 10 times, since each successive candidate is better than the previous one.

Scanned by CamScanner

Given the list of ranks A2 = <10, 9, 8, 7, 6, 5, 4, 3, 2, 1>, a new office assistant will be hired only once, in the first iteration.

Given a list of ranks A3 = <5, 2, 1, 8, 4, 7, 10, 9, 3, 6>, a new office assistant will be hired three times, upon interviewing the candidates with ranks 5, 8, and 10.

Consider, on the other hand, the randomized algorithm that first permutes the candidates and then determines the best candidate. In this case, the randomization is in the algorithm, not in the input distribution.

Each time we run the algorithm, the execution depends on the random choices made and is likely to differ from the previous execution of the algorithm.

For this algorithm and many other randomized algorithms, no particular input elicits its worst-case behavior.
The random permutation makes the input order irrelevant.
The randomized algorithm performs badly only if the random-number generator produces an "unlucky" permutation.
For the hiring problem, the only change needed in the code is to randomly permute the array.

```
RANDOMIZED-HIRE-ASSISTANT(n)
    randomly permute the list of candidates
    best ← 0                    // candidate 0 is a least-qualified dummy candidate
    for i ← 1 to n do
    {
    interview candidate i
    if candidate i is better than candidate best then
        {
            best ← i
            hire candidate i
        }
    }
```

With this simple change, we have created a randomized algorithm whose performance matches that obtained by assuming that the candidates were presented in a random order.

Randomly Permuting Arrays :
Many randomized algorithms randomize the input by permuting the given input array. Here, we shall discuss two methods for doing so. We assume that we are given an array A which, without loss of generality, contains the elements 1 through n. Our goal is to produce a random permutation of the array.

Method 1: Randomize by Sorting
One common method is to assign each element A[i] of the array a random priority P[i], and then sort the elements of A according to these priorities.

Scanned by CamScanner

For example if our initial array is A = <1, 2, 3, 4> and we choose random priorities P = <36, 3, 97, 19>, we would produce an array B = <2, 4, 1, 3>, since the second priority is the smallest, followed by the fourth, then the first, and finally the third.

We call this procedure PERMUTE-BY-SORTING:

```
PERMUTE-BY-SORTING(A)
1   n ← length[A]
2   for i ← 1 to n do
3       P[i] = RANDOM(1, n^3)
4   sort A, using P as sort keys
5   return A
```

Line 3 chooses a random number between 1 and n^3. We use a range of 1 to n^3 to make it likely that all the priorities in P are unique.

Let us assume that all the priorities are unique.
After sorting, if P[i] is the jth smallest priority, then A[i] will be in position j of the output. In this manner we obtain a permutation.

Method 2: Randomize by Place
A better method for generating a random permutation is to permute the given array in place. The procedure RANDOMIZE-IN-PLACE does so in O(n) time.

In iteration i, the element to be swapped with A[i] is chosen randomly from among elements A[i] through A[n]. Subsequent to iteration i, A[i] is never altered.

```
RANDOMIZE-IN-PLACE(A)
1   n ← length[A]
2   for i ← 1 to n
3   do swap A[i] ↔ A[RANDOM(i, n)]
```

❑❑❑❑❑

Scanned by CamScanner

Module 3 Advanced Database Structure

SYLLABUS Weightage : 28 Marks

Advanced Data Structure
Introduction to trees and heap
Red-Black Trees : properties of red-black trees, Operations on Red-black trees.
Binomial Heaps : Binomial trees and binomial heaps, Operation on Binomial heaps.
Analysis of all above operations

Help Line : For any query WhatsApp to 704 501 85 39 to get it Solved.

Scanned by CamScanner

RED AND BLACK TREE

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK.

By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Each node of the tree now contains the fields color, key, left, right, and p. If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL.

We shall regard these NIL's as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

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

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

We call the number of black nodes on any path from, but not including, a node x down to a leaf the black-height of the node, denoted bh(x). We define the black-height of a red-black tree to be the black-height of its root. A red-black tree with n internal nodes has height at most 2 log (n + 1).

- **Rotations :**

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take O(lg n) time. Because they modify the tree, the result may violate the red-black properties. To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

We change the pointer structure through rotation, which is a local operation in a search tree that preserves the binary-search-tree property.
There are two kinds of rotations: left rotations and right rotations.

Algorithm for Left Rotation :

```
LEFT-ROTATE(T, x)
y ← right[x]                    // Set y
right[x] ← left[y]              // Turn y's left subtree into x's right subtree.
p[left[y]] ← x
p[y] ← p[x]                     // Link x's parent to y.
if p[x] = nil[T] then
      root[T] ← y
else if x = left[p[x]] then
            left[p[x]] ← y
      else right[p[x]] ← y
```

Scanned by CamScanner

```
10) left[y] ← x              // Put x on y's left.
11) p[x] ← y
```

Algorithm for Right Rotation :

```
RIGHT-ROTATE(T, x)
1)  y ← left[x]              // Set y
2)  left[x] ← right[y]       // Turn y's right subtree into x's left subtree.
3)  p[right[y]] ← x
4)  p[y] ← p[x]              // Link x's parent to y.
5)  if p[x] = nil[T] then
6)      root[T] ← y
7)  else if x = left[p[x]] then
8)          left[p[x]] ← y
9)      else right[p[x]] ← y
10) right[y] ← x             // Put x on y's right
11) p[x] ← y
```

- **Insertion in RB Tree :**

Insertion of a node into an n-node red-black tree can be accomplished in O(lg n) time. We use a slightly modified version of the TREE-INSERT procedure to insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red. To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT(T, z) inserts node z, whose key field is assumed to have already been filled in, into the red-black tree T .

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

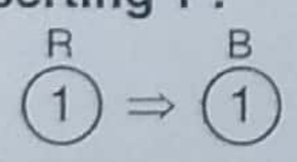
Coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in RB-INSERT to restore the red-black properties.

Scanned by CamScanner

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED do
{
    if p[z] = left[p[p[z]]] then
    {
        y ← right[p[p[z]]]
        if color[y] = RED then
        {
            color[p[z]] ← BLACK                 // Case 1
            color[y] ← BLACK                    // Case 1
            color[p[p[z]]] ← RED                // Case 1
            z ← p[p[z]]                         // Case 1
        }
        else if z = right[p[z]] then
        {
            z ← p[z]                            // Case 2
            LEFT-ROTATE(T, z)                   // Case 2
        }
        color[p[z]] ← BLACK                     // Case 3
        color[p[p[z]]] ← RED                    // Case 3
        RIGHT-ROTATE(T, p[p[z]])                // Case 3
    }
    else (same as then clause with "right" and "left" exchanged)
    color[root[T]] ← BLACK
}
```

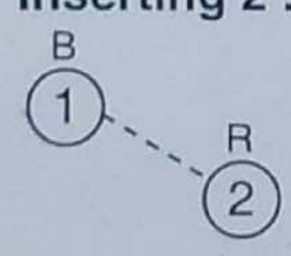
Red and Black Tree Insertion :

Inserting 1 :

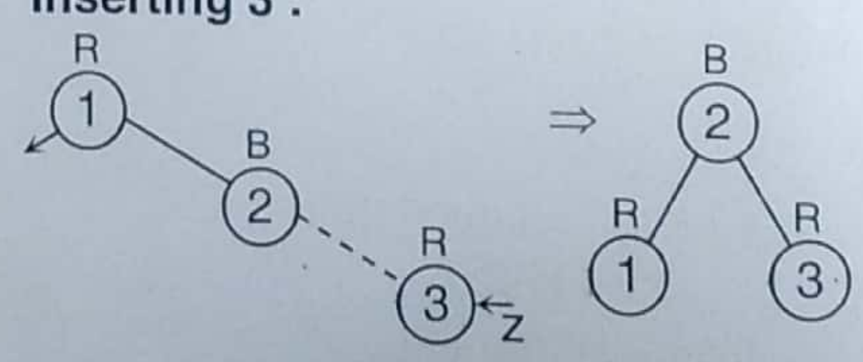


Converting root as black

Inserting 2 :

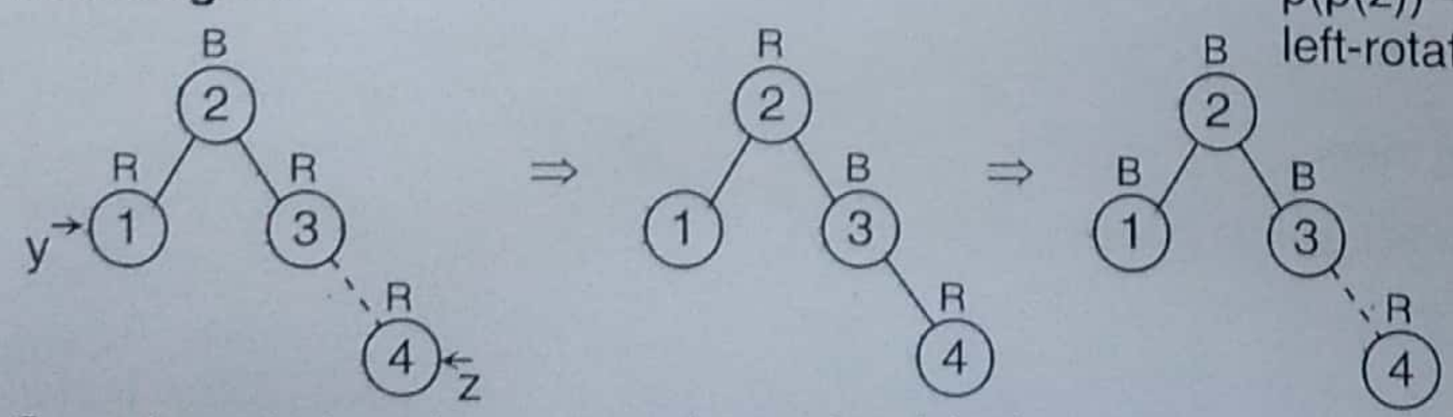


Inserting 3 :



Case 3 : p(z) → black
p(p(z)) → red
left-rotate at p(p(z))

Inserting 4 :

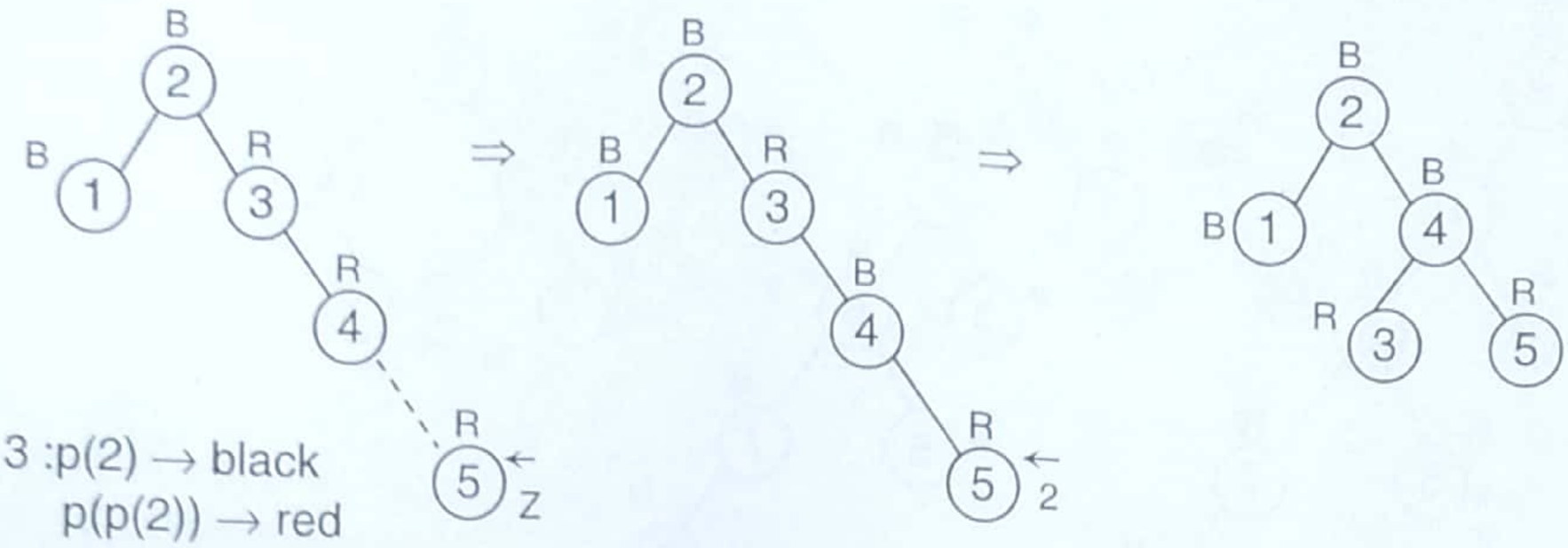


Case 1: uncle is red
p(z) → black
y → black
p(p(z)) → red
z → p(p(z))

color root as black

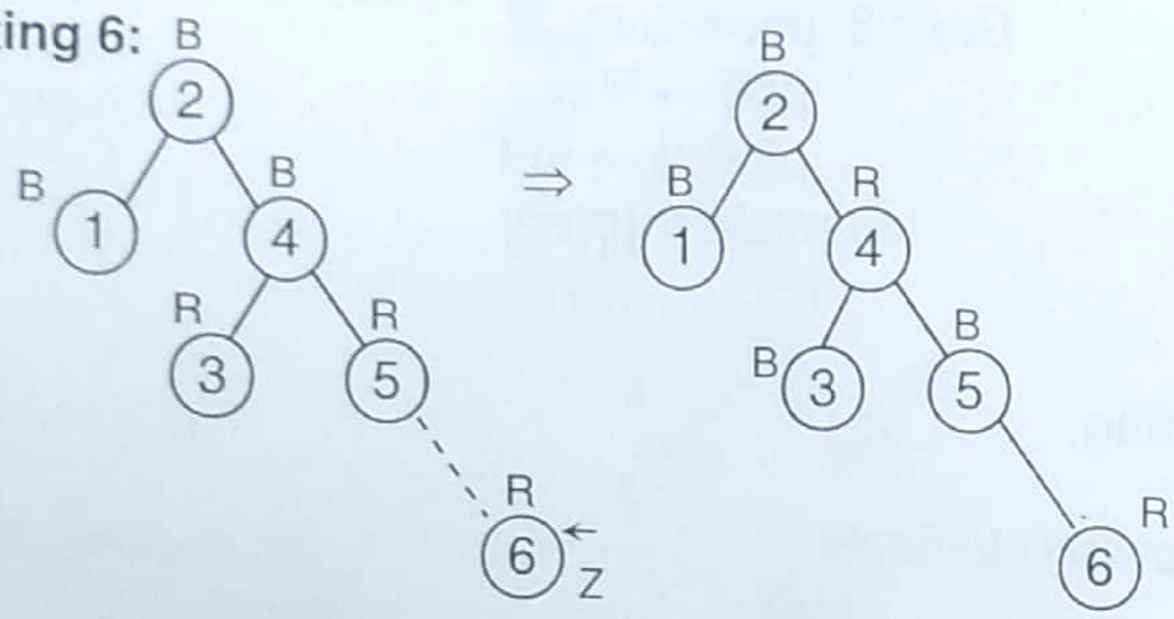
Scanned by CamScanner

Inserting 5:



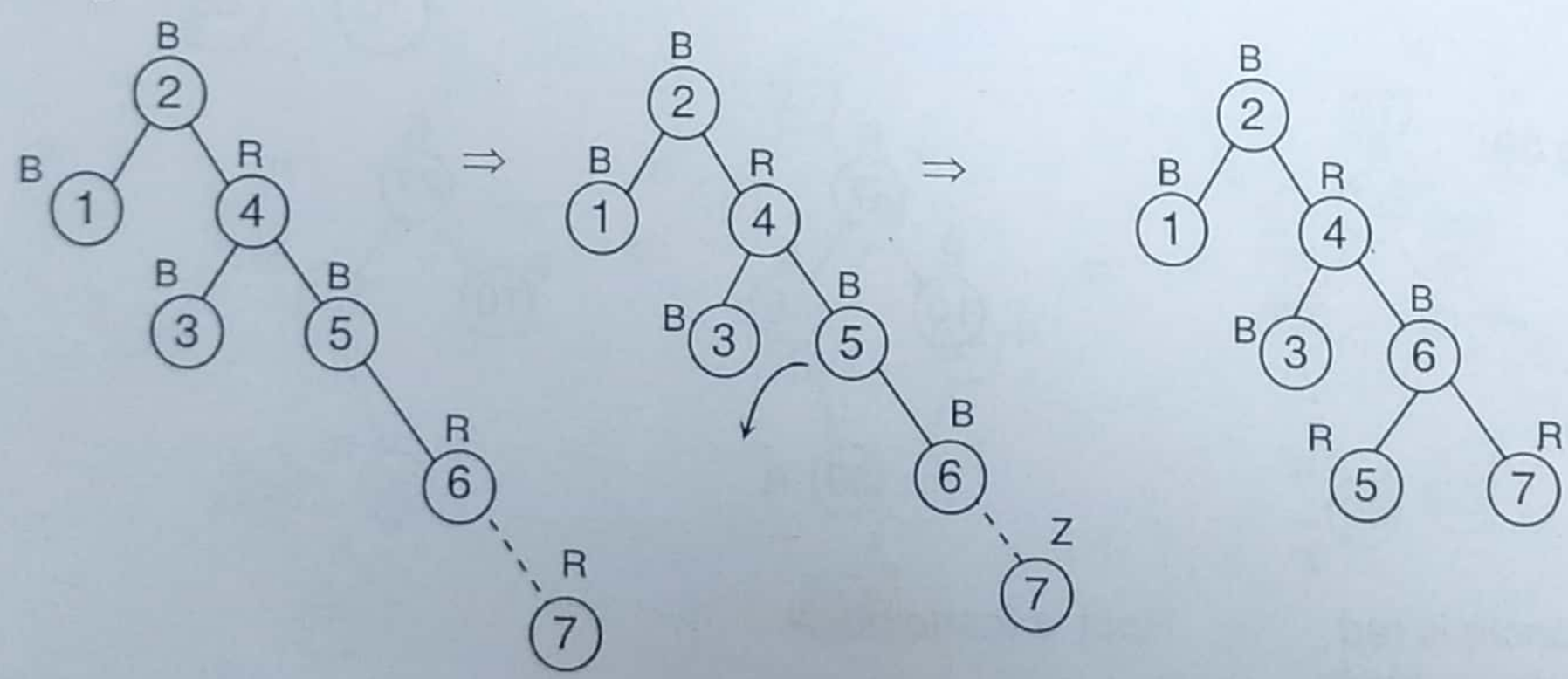
Case 3 :p(2) → black
p(p(2)) → red
rotate-left (p(p(z))

Inserting 6:



Case 1 = p(z) → black
y → black
p(p(z)) → red
z → p(p(z))

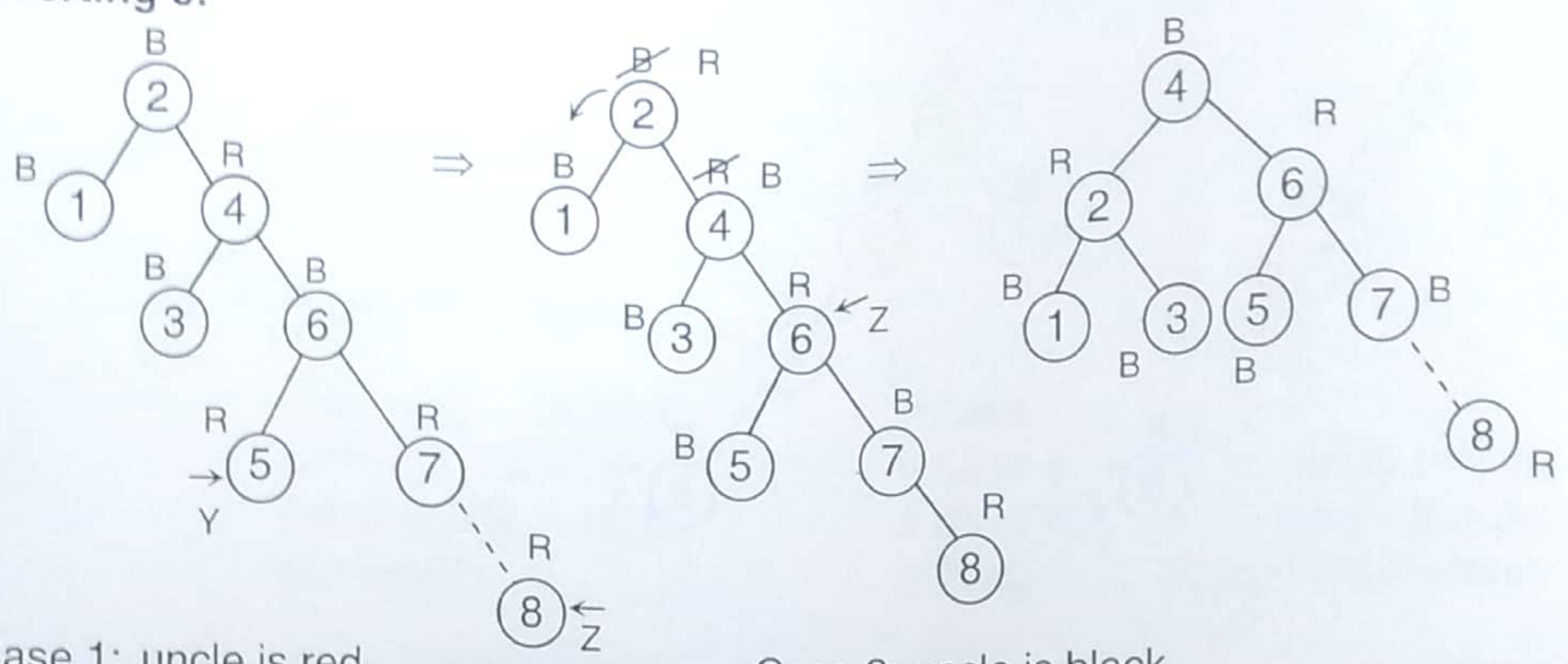
Inserting 7:



Case 3 = p(z) → black
p(p(z)) → red
left-rotate (p(p(z)))

Scanned by CamScanner

Inserting 8:



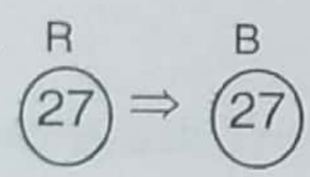
Case 1: uncle is red
$p(z) \rightarrow$ black
$y \rightarrow$ black
$p(p(z)) \rightarrow$ red
$z \rightarrow p(p(z))$

Case 3: uncle is black
$p(z) \rightarrow$ black
$p(p(z)) \rightarrow$ red
left rotate $(p(p(z)))$

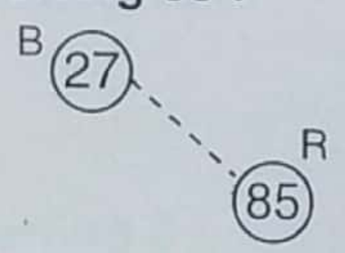
Analysis $\Rightarrow$ Insert takes $O(\log_2 n)$ time

* Construct RB Tree for the following elements:
27, 85, 10, 39, 30, 65, 70

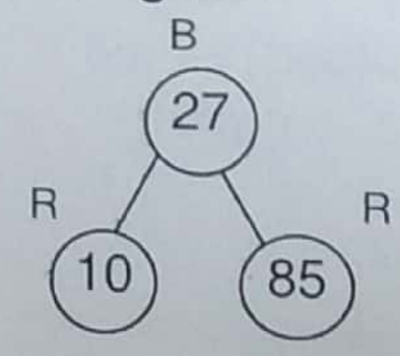
Inserting 27 :



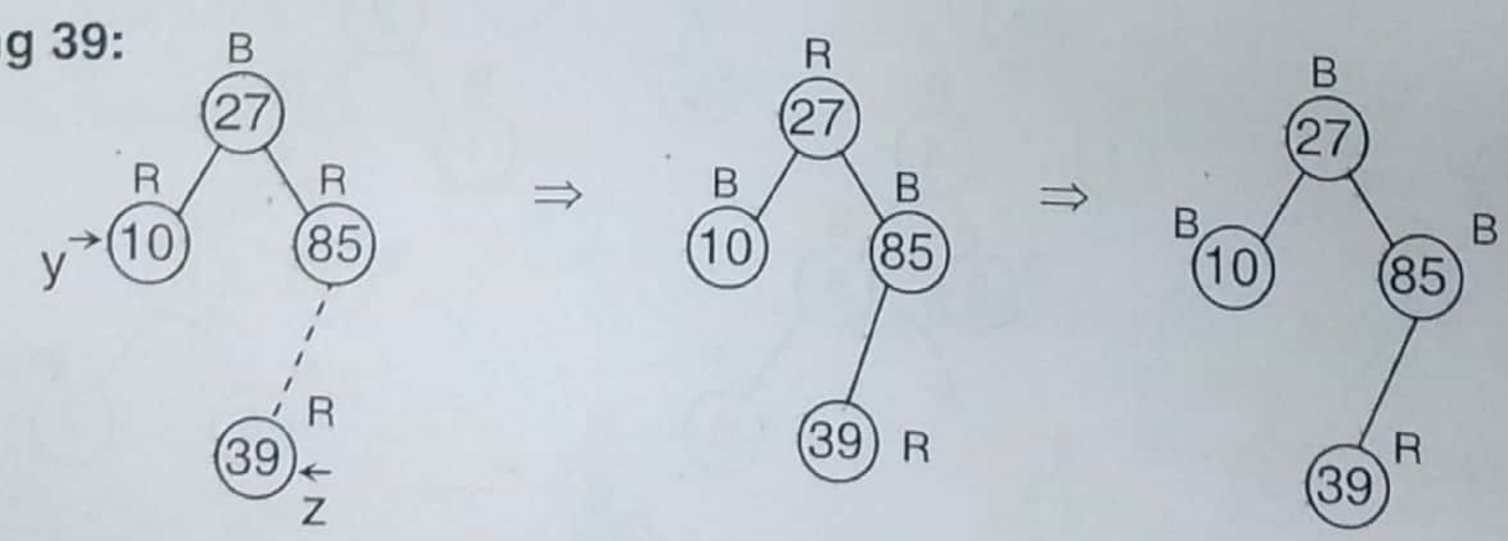
Inserting 85 :



Inserting 10 :



Inserting 39:

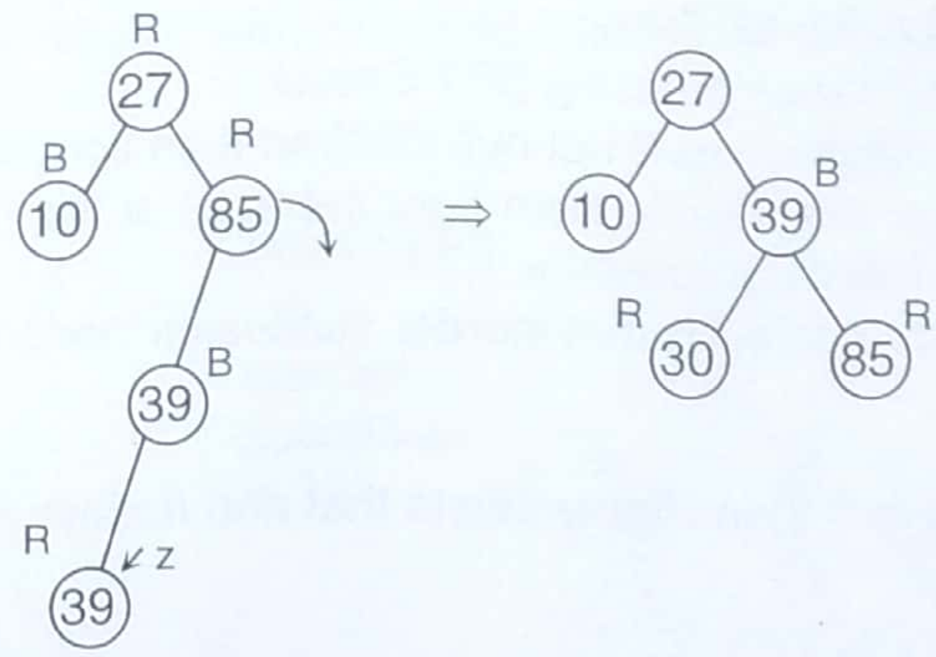


Case 1: uncle is red
$p(z) \rightarrow$ black
$y \rightarrow$ black
$p(p(z)) \rightarrow$ red
$z \rightarrow p(p(z))$

Root is made black

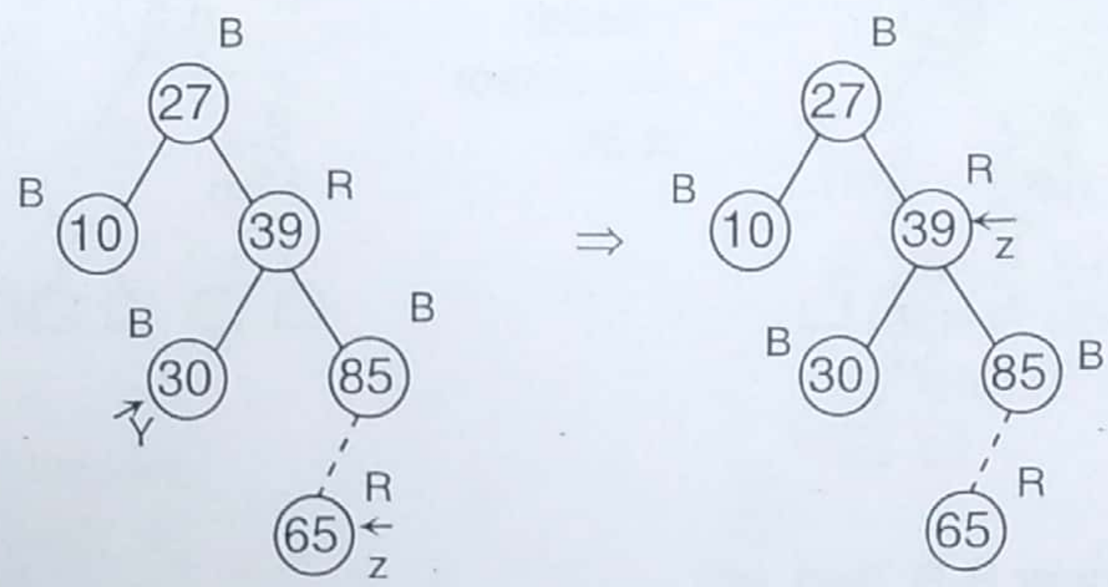
Scanned by CamScanner

Inserting 30:



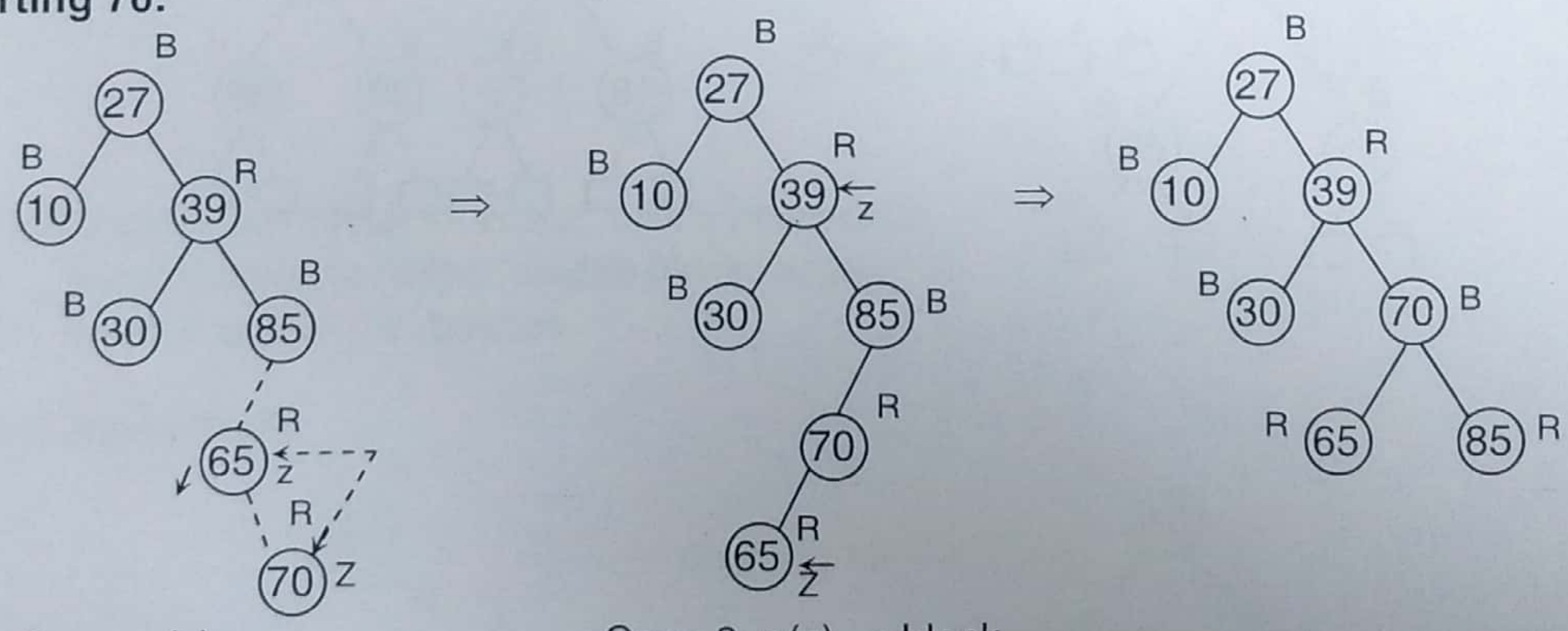
Case 3: p(z) → black
p(p(z)) → red
right = rotate (p(p(z)))

Inserting 65:



Case 1: uncle is red
p(z) → black
y → black
p(p(z)) → red
z → p(p(z)

Inserting 70:



Case 2: z = p(z)
left-rotate(z)

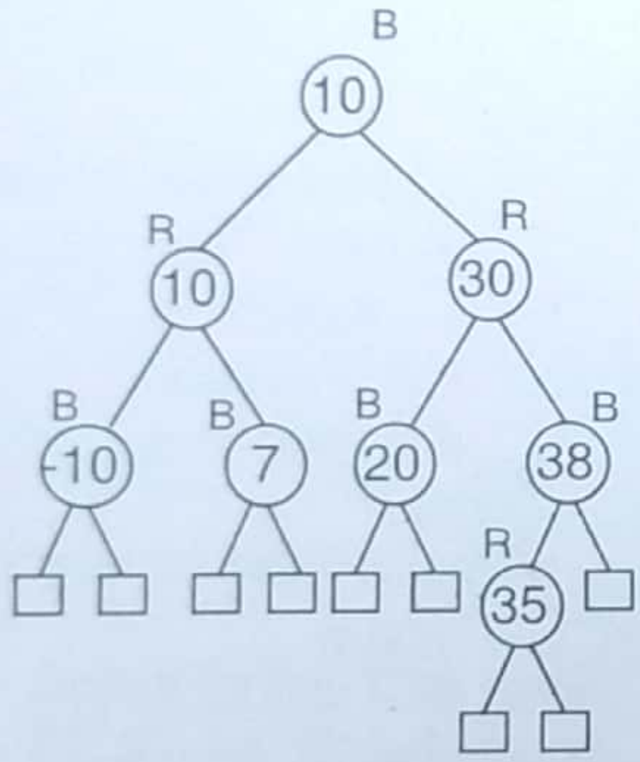
Case 3: p(z) → black
p(p(z) → red
right-rotate (p(p(z)))

Scanned by CamScanner

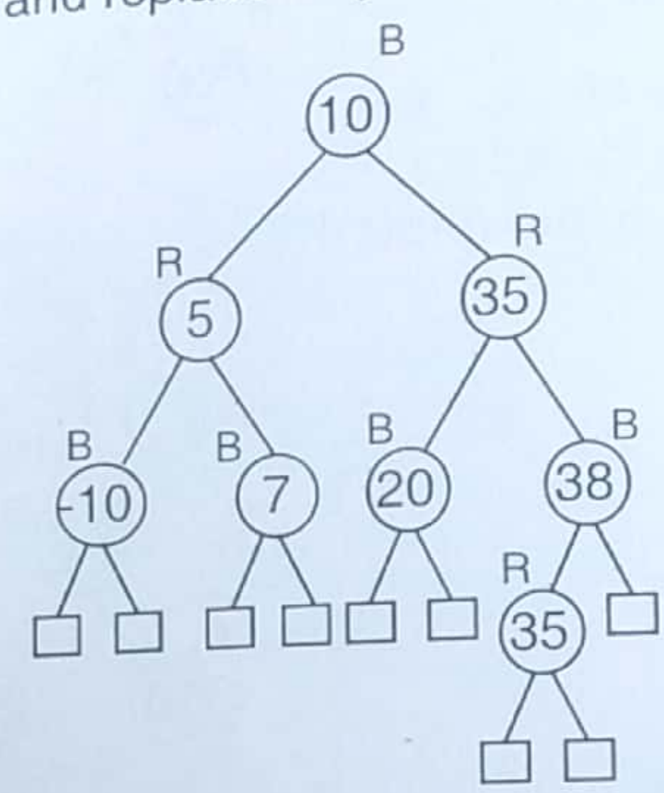
Deletion from Red & Black Tree :

- Search node to be deleted using BST Search
- If a node to be deleted has 2 not null children then convert it to 0 or 1 not null child.
- Find inorder successor (min. from right subtree) of the node and replace node to be deleted with its inorder successor.
- Now, it reduced to deleting that inorder successor node which would either have 0 or 1 not null child.

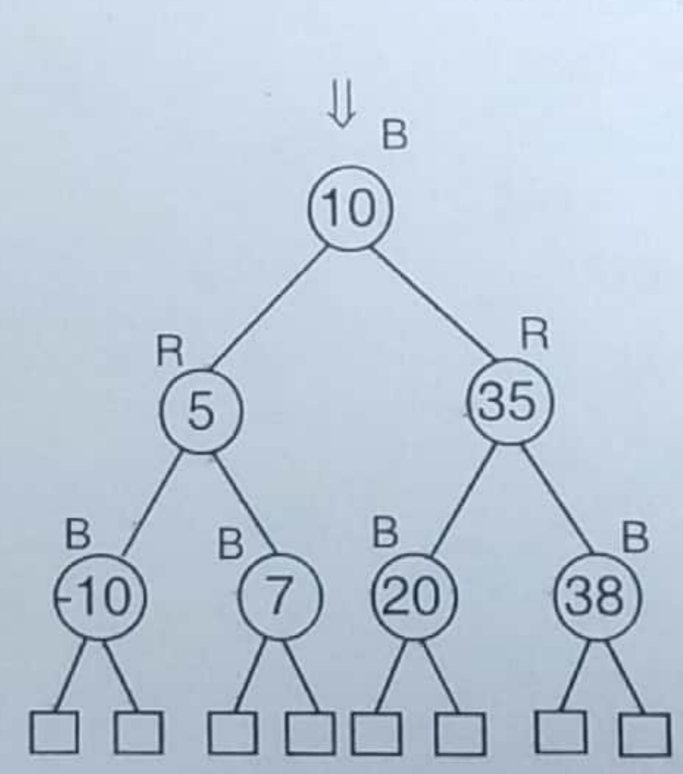
1. If that node is "Red" then simply delete that and replace it by null child.



deleting 30
⇒
inorder
successor
is 35

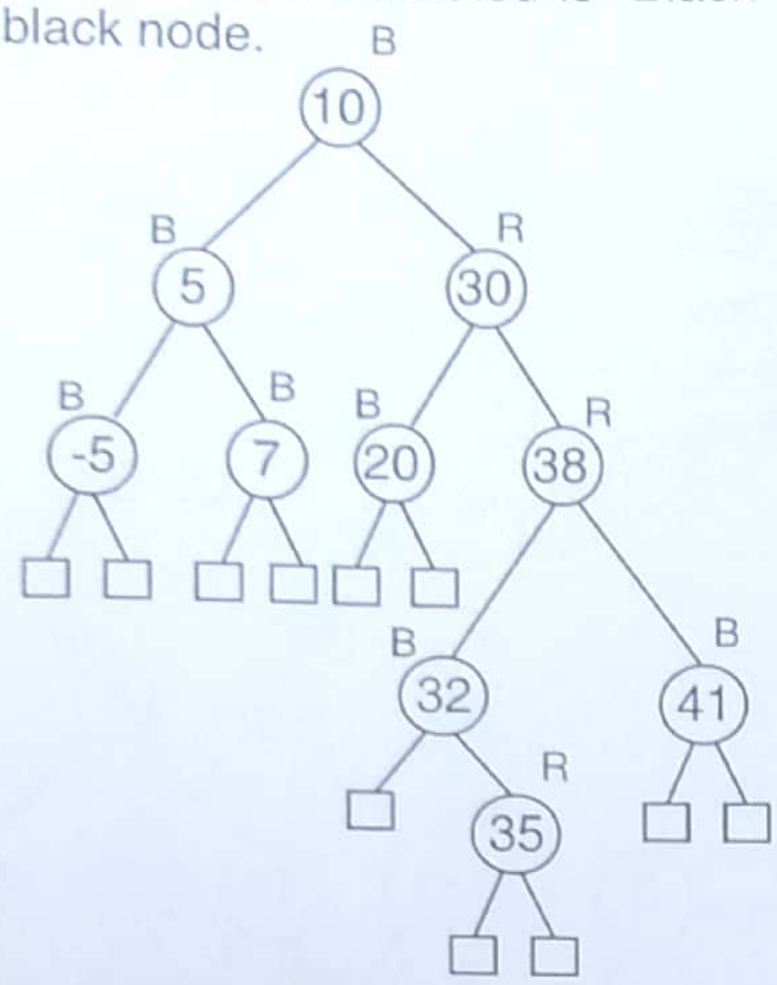


- It's proper R-B Tree, as deleting red node does not affect the black height of the tree.

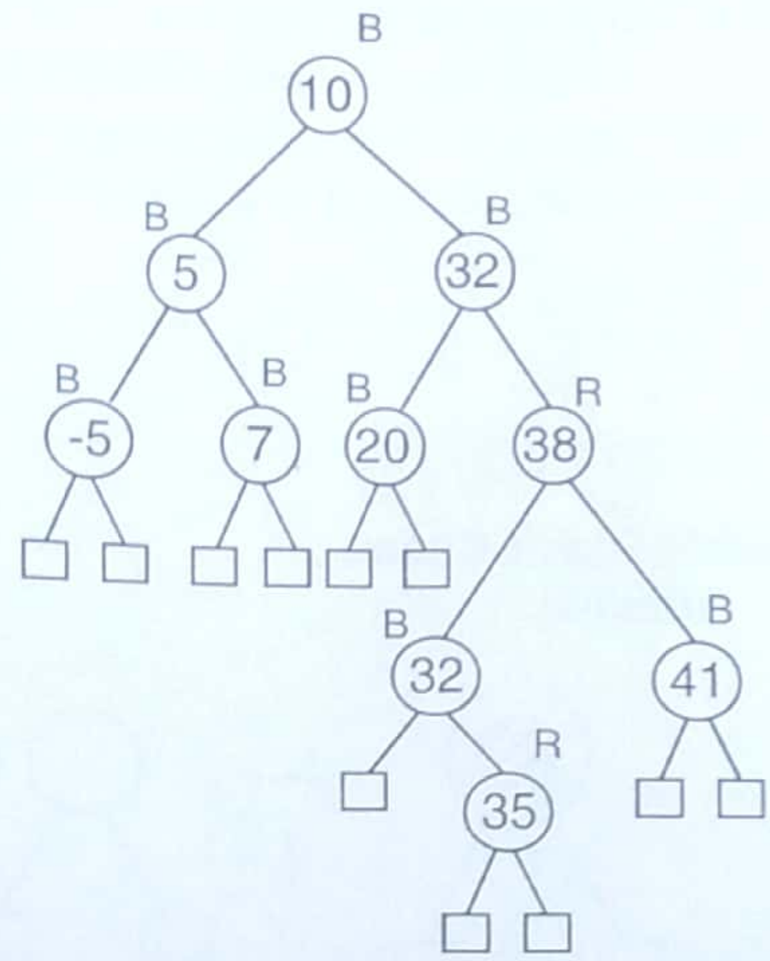


Scanned by CamScanner

2. If a node to be deleted is "Black" with red child then, delete black and turn red into black node.

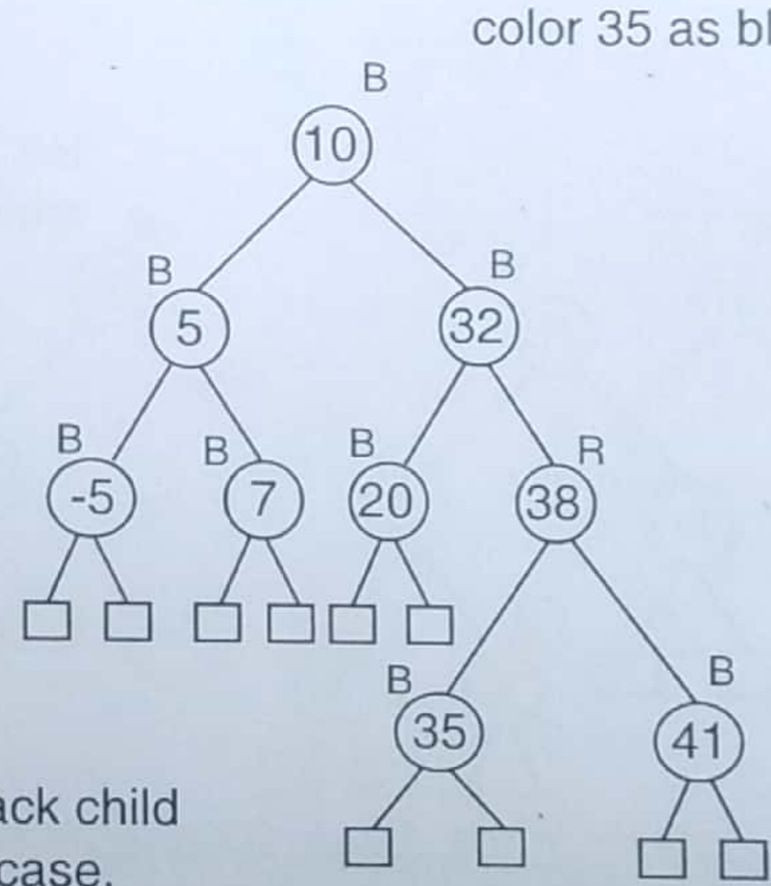


deleting 30
⇒
inorder
successor
is 32



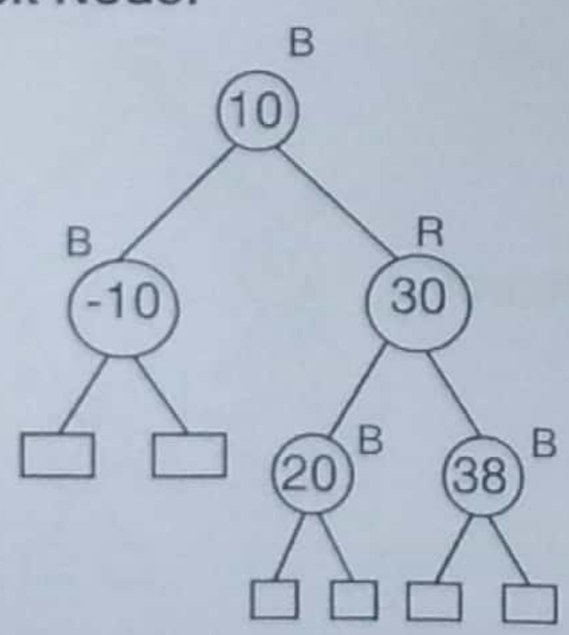
⇓ delete 32, replace
it with 35
color 35 as black

- This is valid R – B Tree, as black height of the tree is maintained.

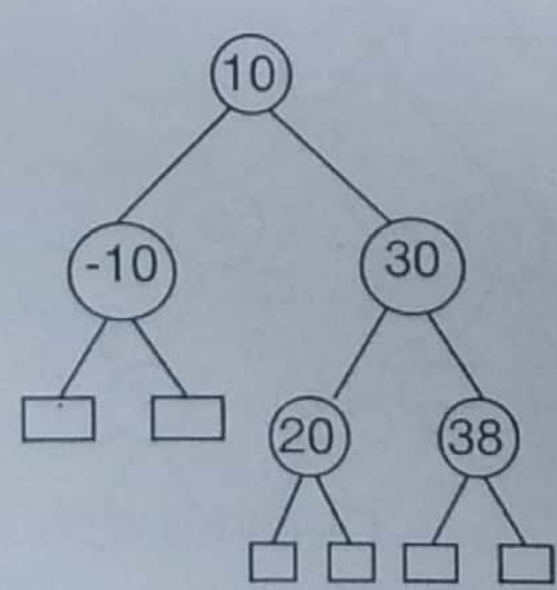


3. Deleting a node which is "Black" with a black child
   - This situation is called "double black" case.
   - It has 6 different subcases.

Double Black Node:



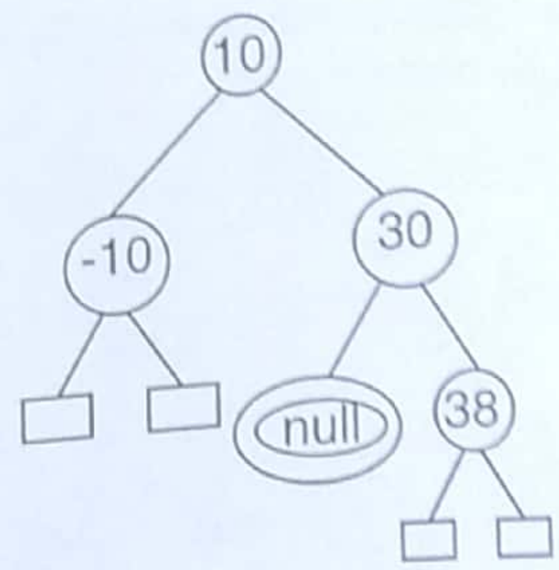
deleting 10
⇒
inorder
successor
is 20



Scanned by CamScanner

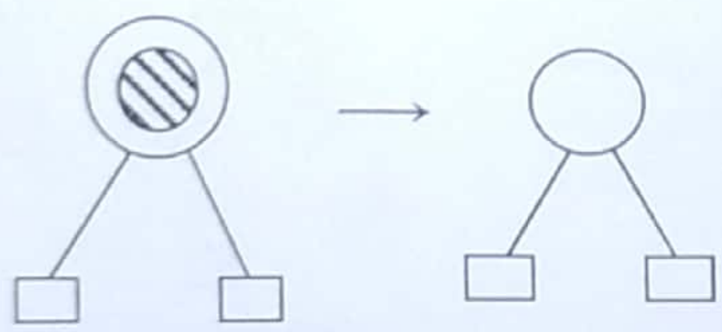
Deleting 20

- Delete 20, replace it with null node
- it hampers the black height in that branch
- that null node carries weight of extra black node, hence it is called as "double black" node.

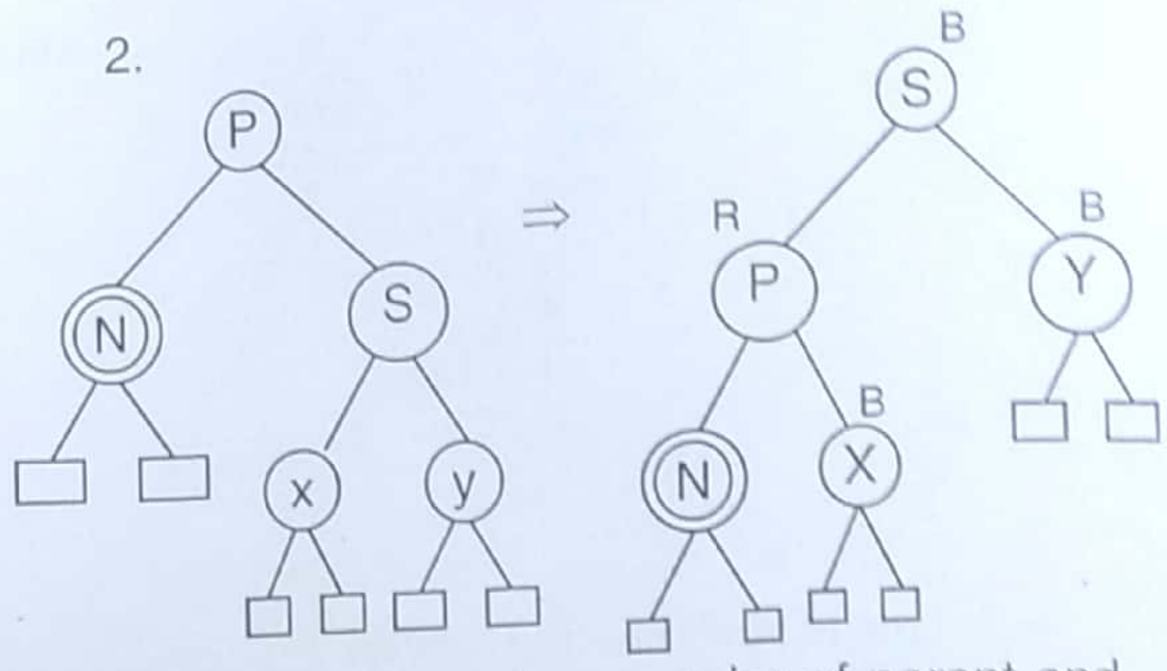


Double Black Cases :

1. Terminal

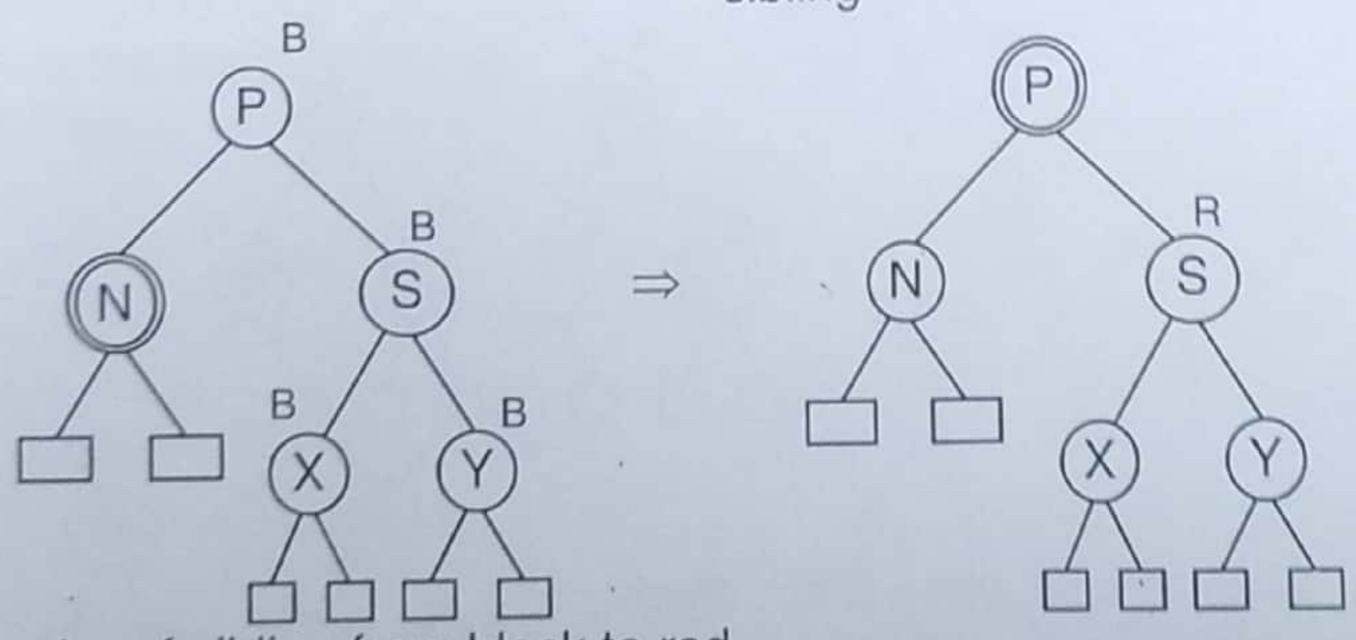


2.



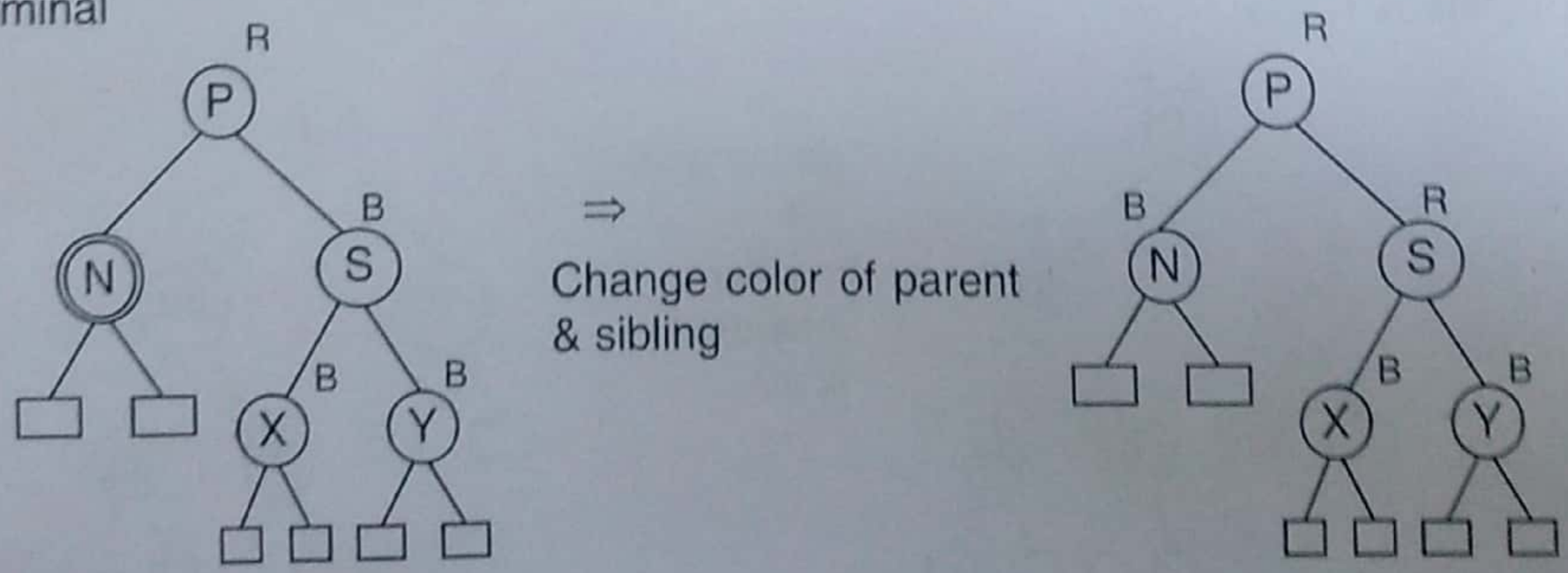
left rotate at parent, change color of parent and sibling

3.



Change color of sibling from black to red that makes parent as double black

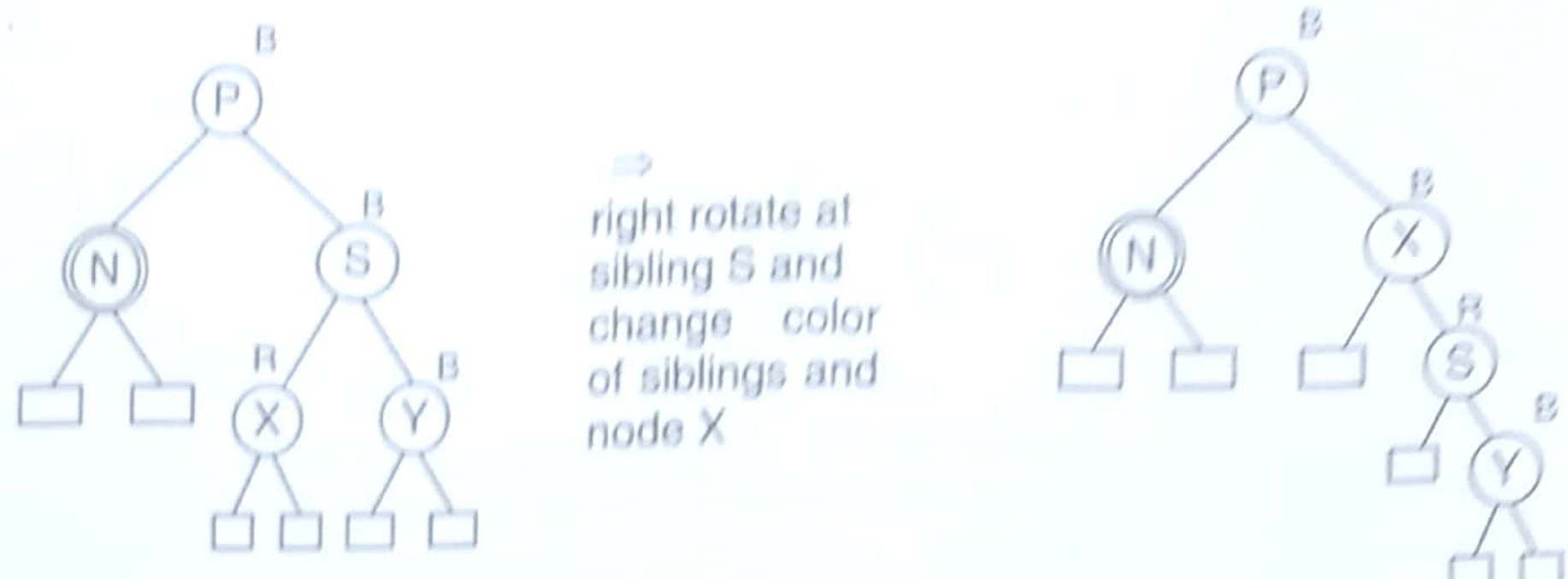
4. Terminal



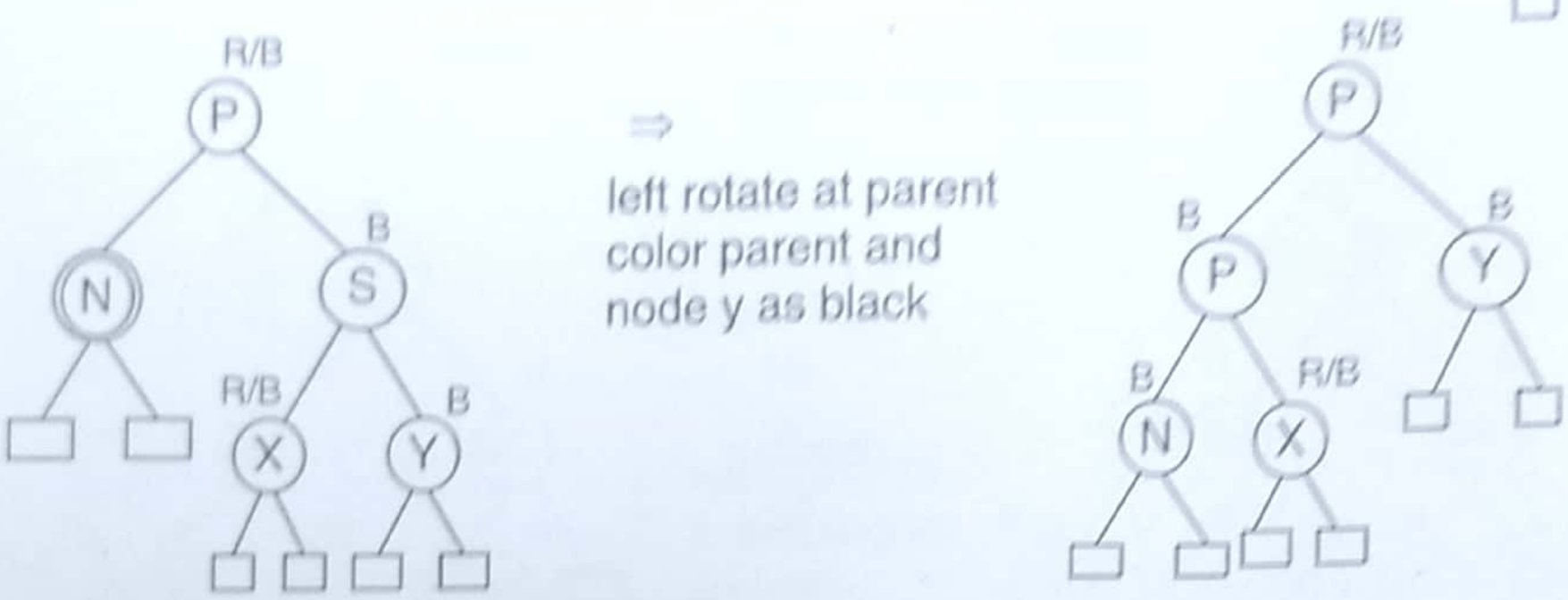
Change color of parent & sibling

Scanned by CamScanner

5.

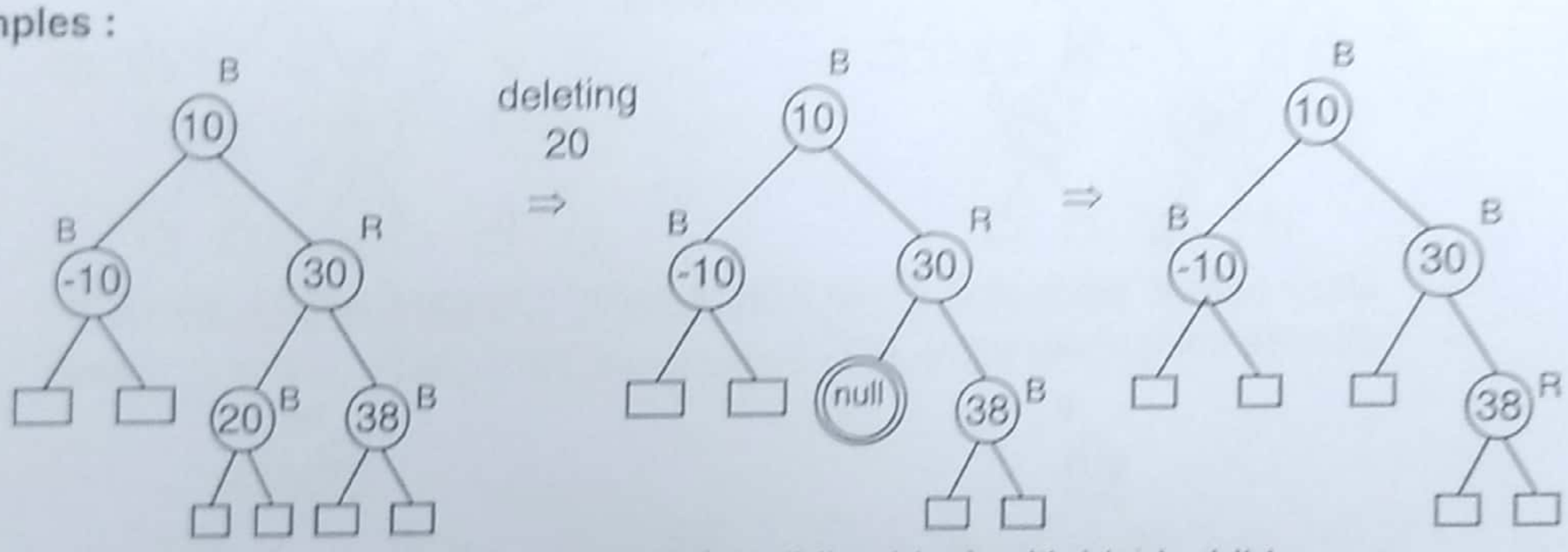


6.



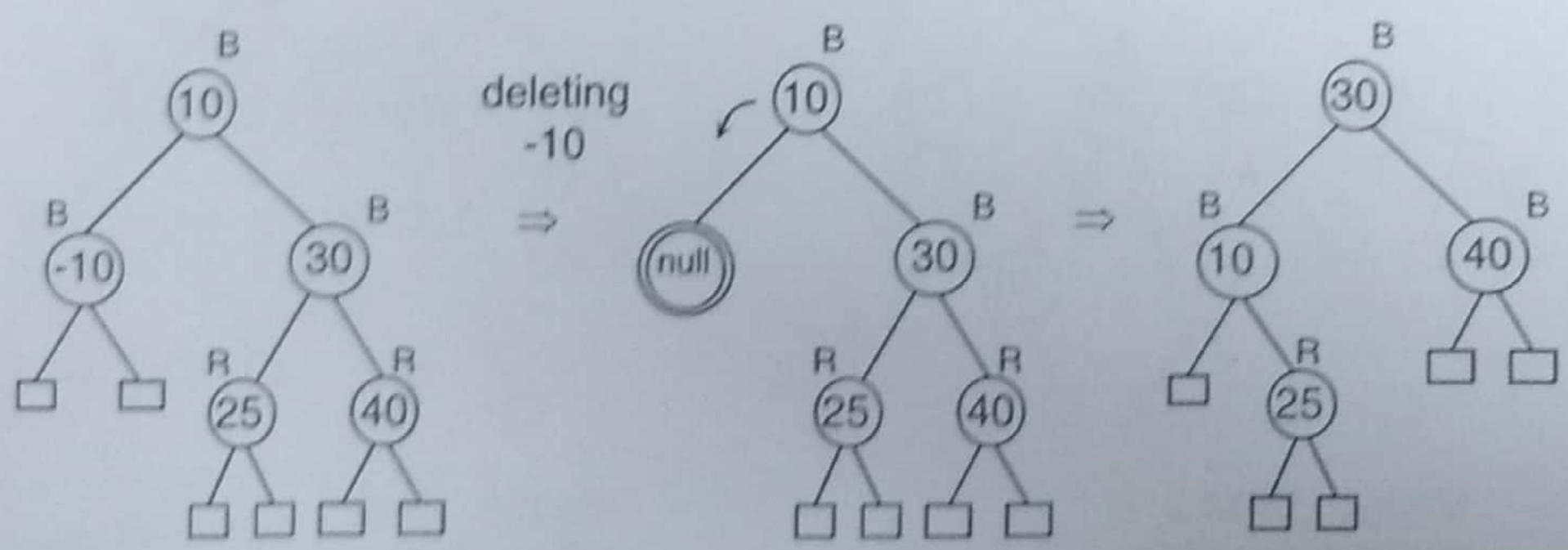
Examples :

1.



- Parent red, sibling black with black children
- Case 4 ⇒ swap colors of parent and sibling

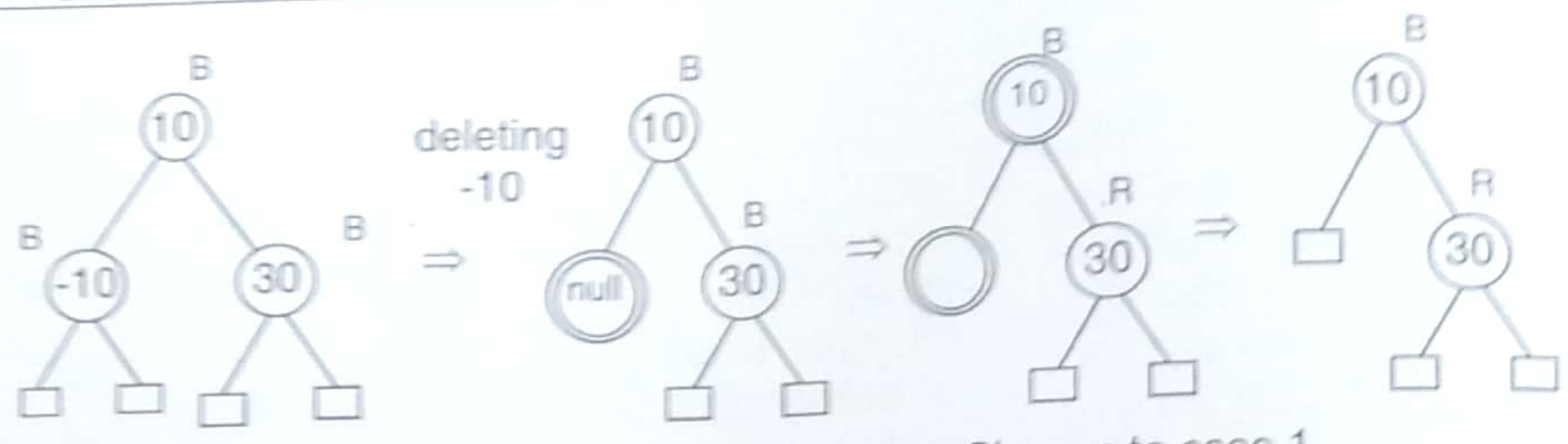
2.



- right child of sibling is red & sibling is black
- Case 6 ⇒ left rotate at parent
- color parent & right child of sibling as black

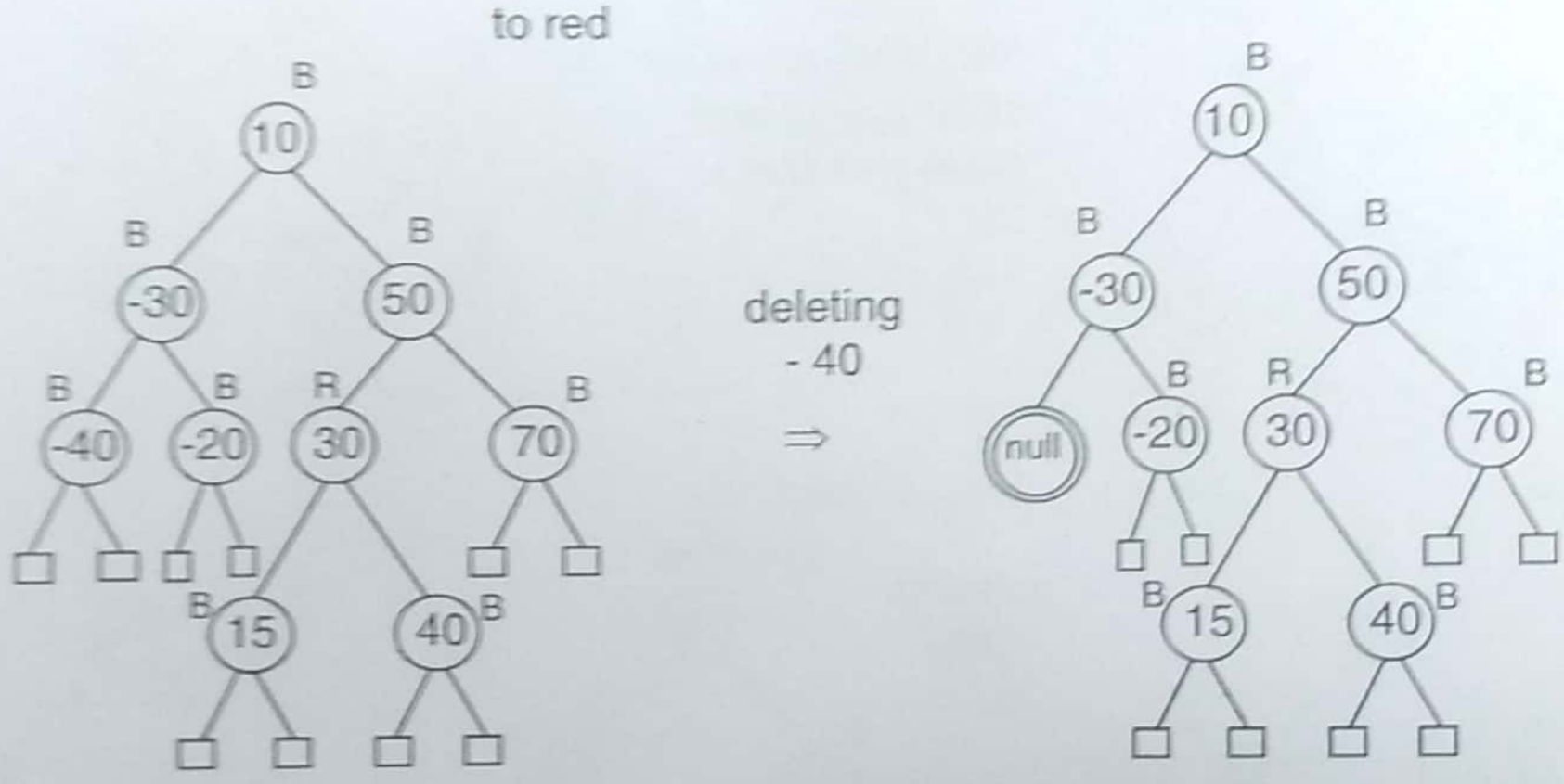
Scanned by CamScanner

3.

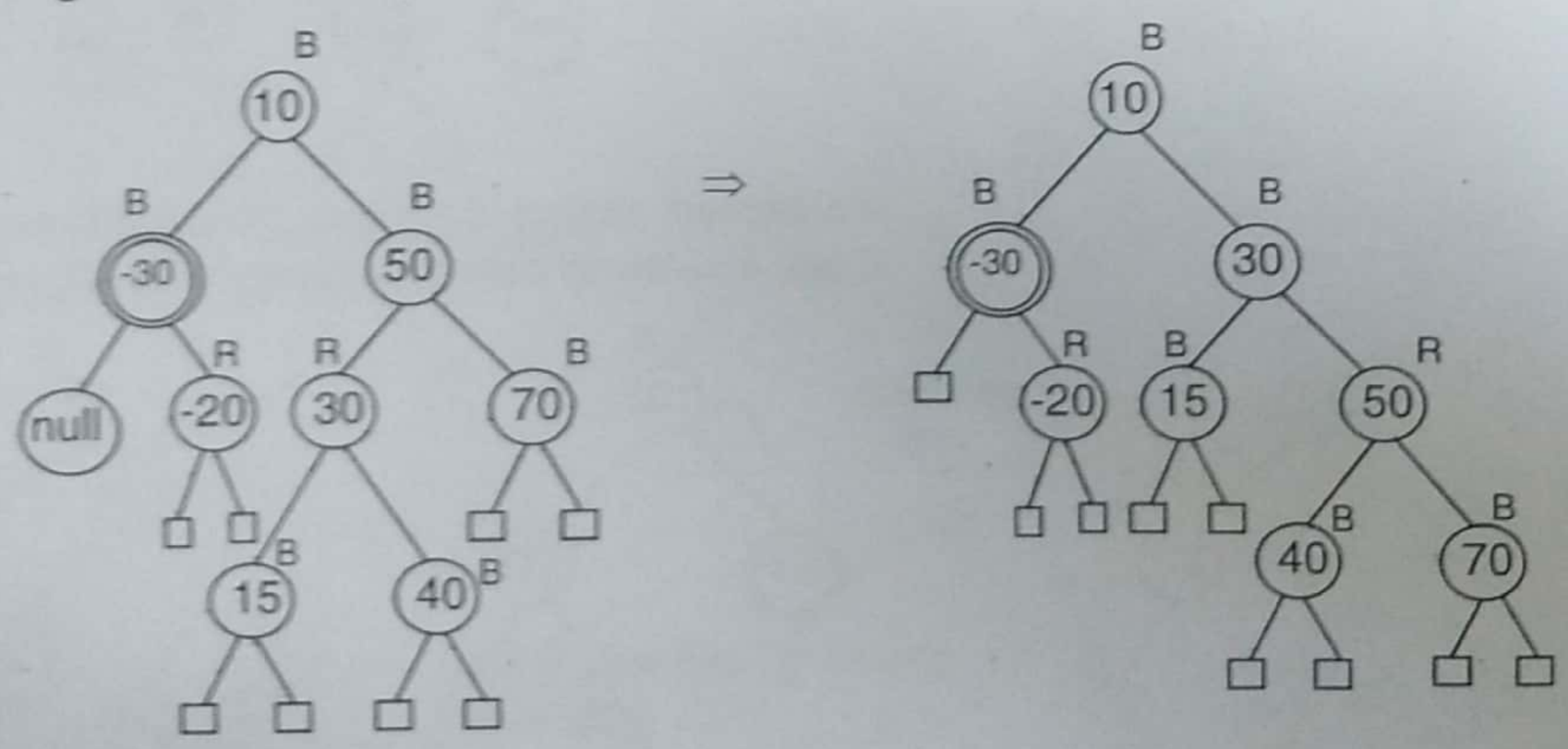


- black parent & sibling with 2 black children
- Case 3 ⇒ Change color of right sibling to red
- Change to case 1
- as 10 is root, it simply can be avoided

4.

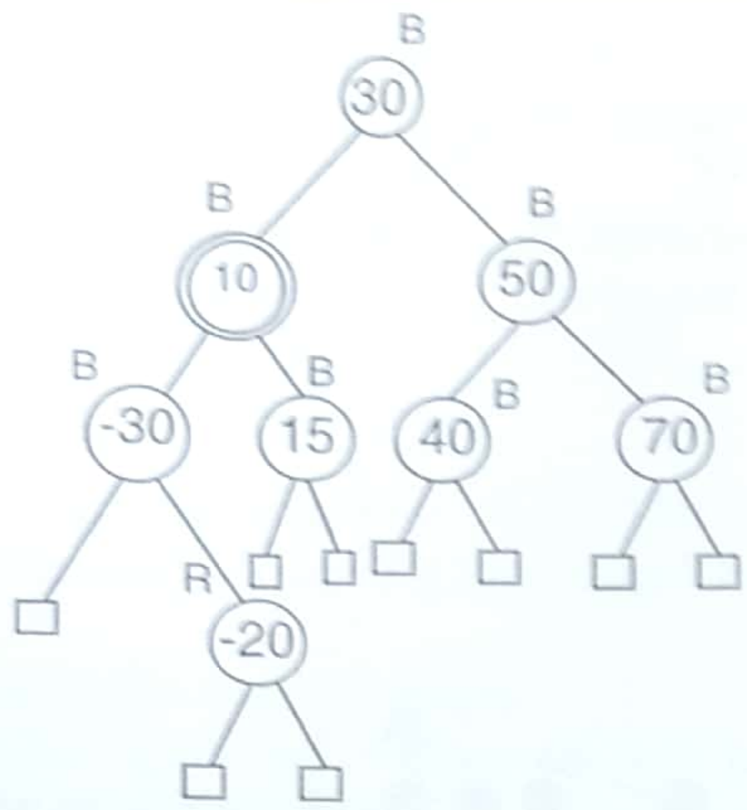


- black parent, black sibling with 2 black children hence Case 3 is satisfied
- change color of sibling to "red"

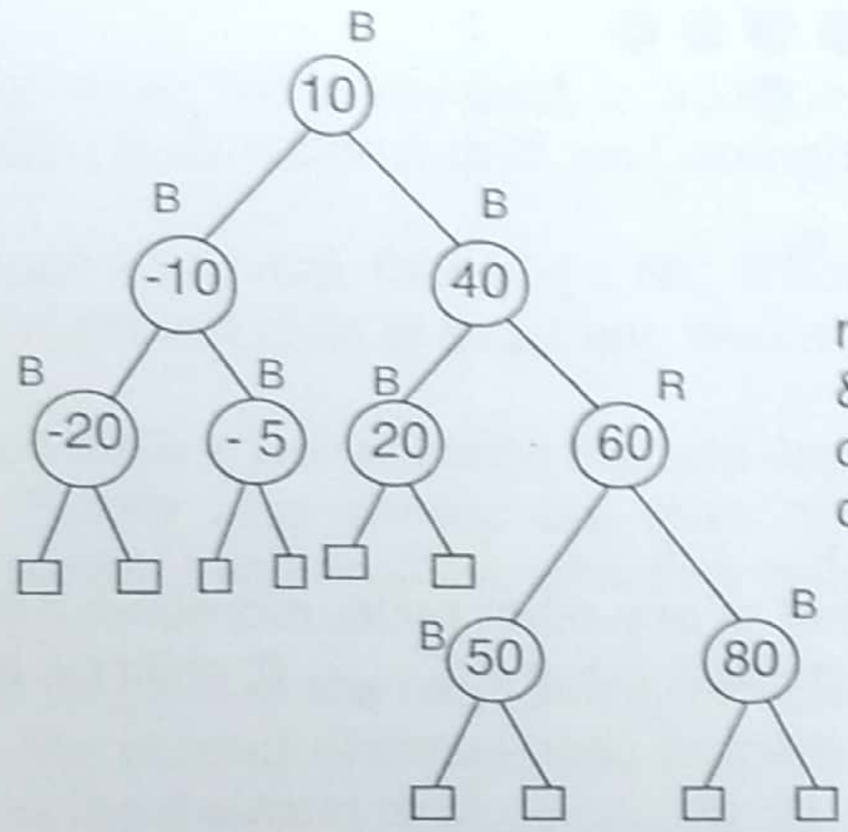


- Maps to Case 5
- right rotate at sibling (50)
- change color and it's left child sibling
- Maps to Case 6
- left rotate at parent
- color parent and right child of sibling as black

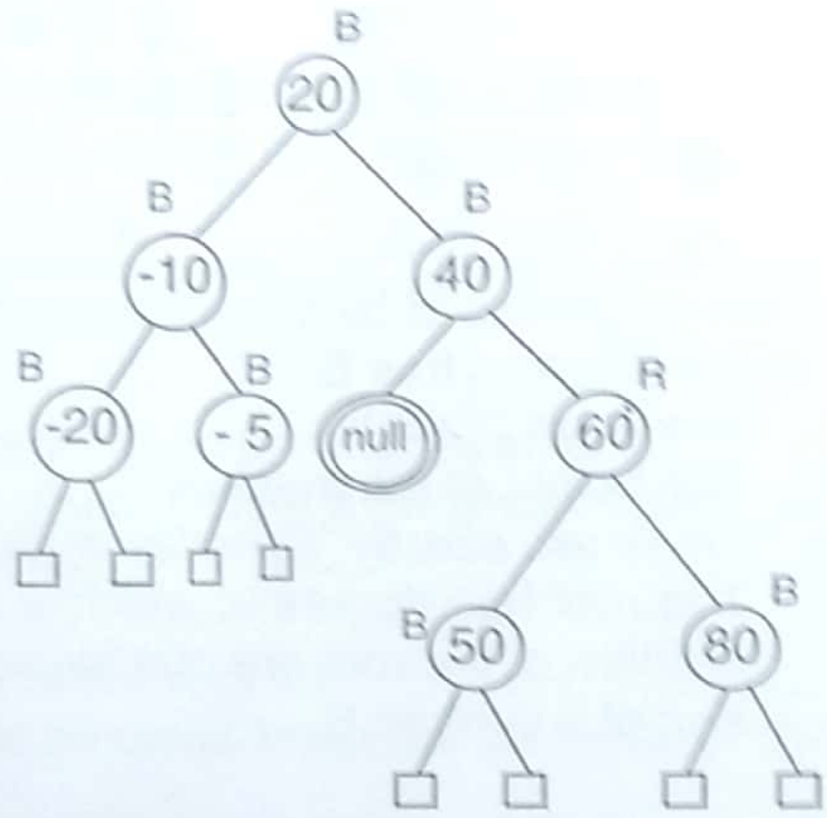
Scanned by CamScanner



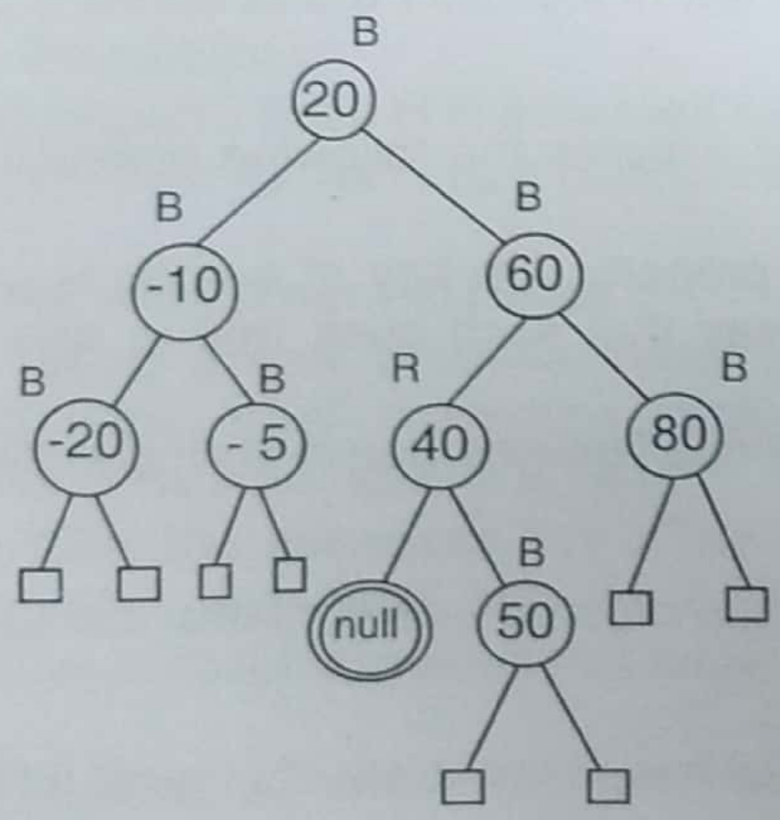
5.



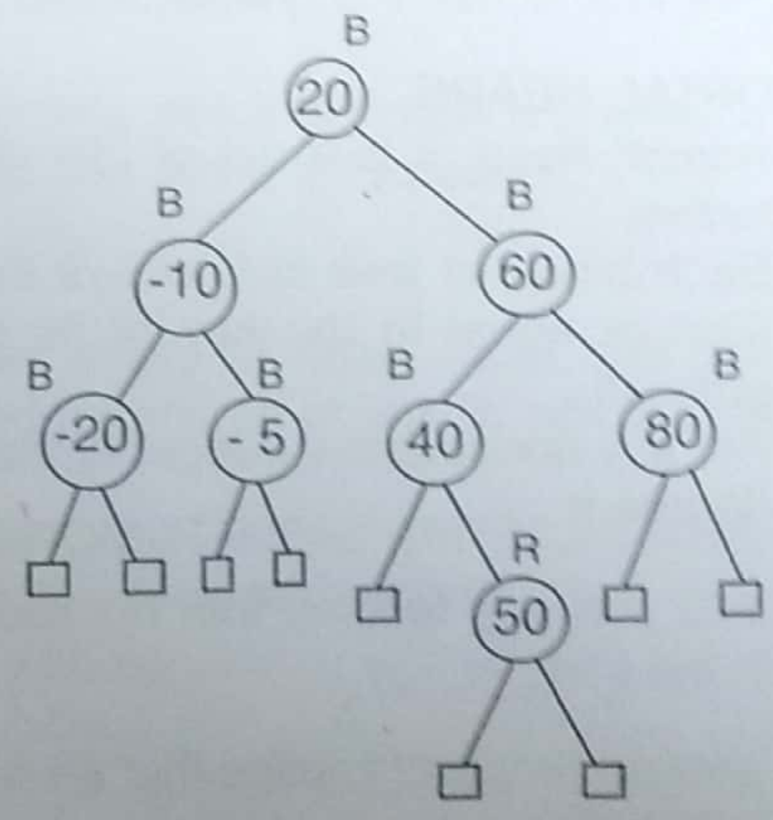
deleting
10
⇒
replacing 20
& deleting 20
creates
double black



- Parent black, sibling red & both children of sibling are black
- Case 2 ⇒ rotate left at parent, swap colors of parent & sibling



⇒

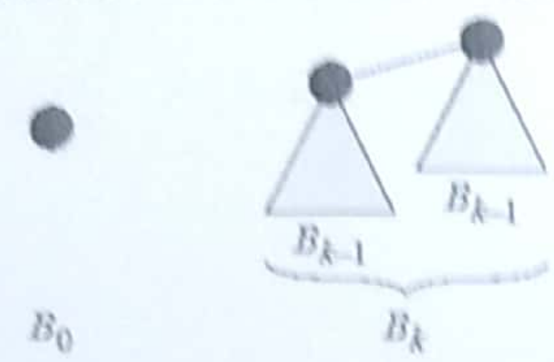


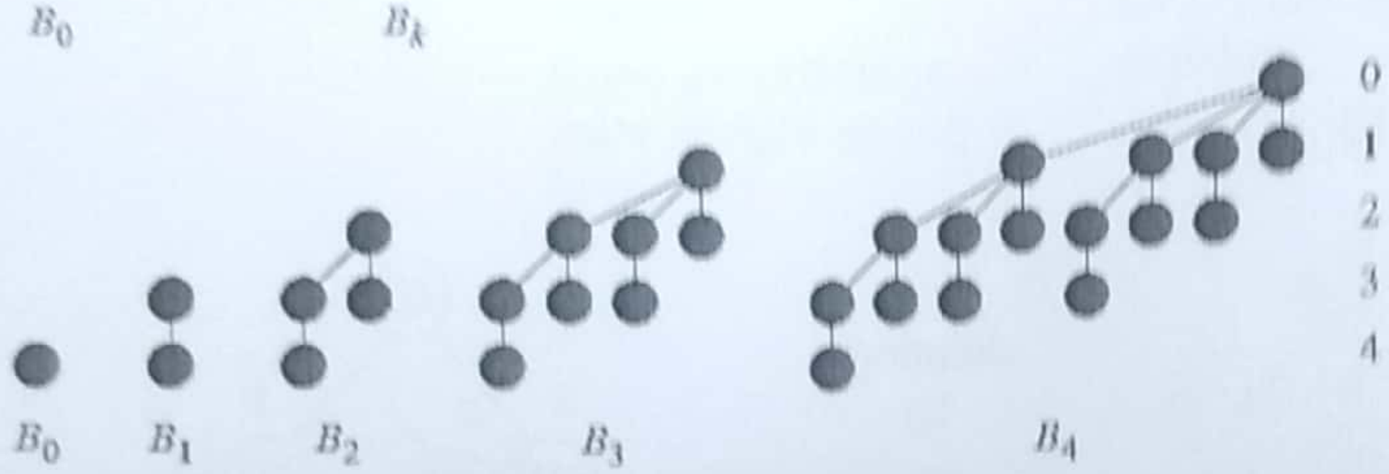
- Now convert to Case 4
- Swap parent & sibling color

Scanned by CamScanner

BINOMIAL TREES

- The binomial tree B_k is an ordered tree defined recursively.
- The binomial tree B_0 consists of a single node.
- The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together the root of one is the leftmost child of the root of the other.





Some properties of binomial trees are as follows :

For the binomial tree B_k,

1. There are 2^k nodes
2. The height of the tree is k
3. There are exactly ${}^kC_i$ nodes at depth i for i = 0, 1, . . . , k
4. The root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by k – 1, k – 2, . . . , 0, child i is the root of a subtree B_i.

The maximum degree of any node in an n-node binomial tree is log n.

The tree is called 'binomial tree' since the terms ${}^kC_i$ are binomial coefficients.

BINOMIAL HEAPS

A binomial heap H is a set of binomial trees that satisfies the following binomial-heap properties.

1. Each binomial tree in H obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. We say that each such tree is min-heap-ordered.
2. For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

The first property tells us that the root of a min-heap-ordered tree contains the smallest key in the tree.

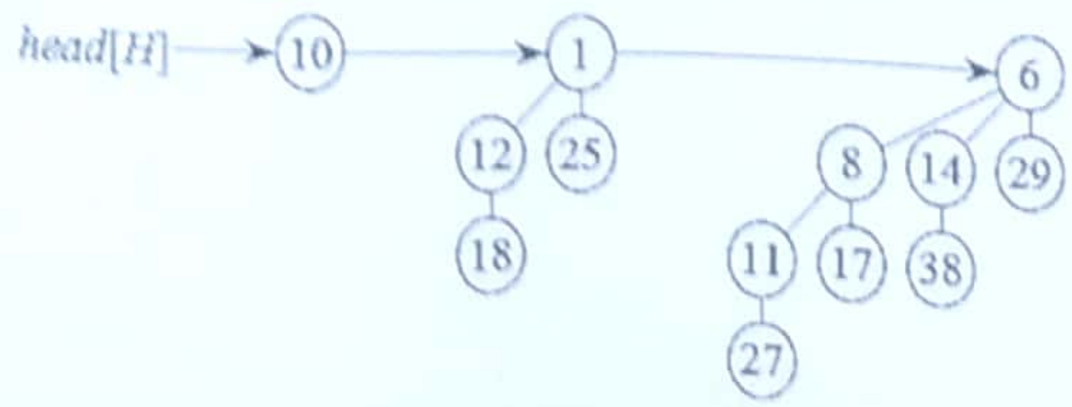
The second property implies that an n-node binomial heap H consists of at most ⌊lg n⌋ + 1 binomial trees.

Observe that the binary representation of n has ⌊lg n⌋ + 1 bits, say <b⌊lg n⌋, b⌊lg n⌋ – 1, …, b0>

Therefore, binomial tree Bi appears in H if and only if bit bi = 1. Thus, binomial heap H contains at most ⌊lg n⌋ + 1 binomial trees.

Scanned by CamScanner

Figure shows a binomial heap H with 13 nodes.



The binary representation of 13 is <1101>, and H consists of min-heap-ordered binomial trees B3, B2, and B0, having 8, 4, and 1 nodes respectively, for a total of 13 nodes.

Representing Binomial Heaps :

As shown in Figure, each binomial tree within a binomial heap is stored in the left-child, right-sibling representation.

Each node has a key field. In addition, each node x contains pointers p[x] to its parent, child[x] to its leftmost child, and sibling[x] to the sibling of x immediately to its right.

If node x is a root, then p[x] = NIL. If node x has no children, then child[x] = NIL, and if x is the rightmost child of its parent, then sibling[x] = NIL.

Each node x also contains the field degree[x], which is the number of children of x.
As Figure also shows, the roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the root list.

The degrees of the roots strictly increase as we traverse the root list.
By the second binomial-heap property, in an n-node binomial heap the degrees of the roots are a subset of {0, 1, . . . , ⌊lg n⌋}.

The sibling field has a different meaning for roots than for non-roots. If x is a root, then sibling[x] points to the next root in the root list. (As usual, sibling[x] = NIL if x is the last root in the root list.)
A given binomial heap H is accessed by the field head[H], which is simply a pointer to the first root in the root list of H. If binomial heap H has no elements, then head[H] = NIL.

Operations on Binomial Heaps :

- **Creating a new binomial heap :** To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object H, where head[H] = NIL. The running time is Θ(1).
- **Finding the minimum key :** The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. This implementation assumes that there are no keys with value ∞.

Binomial-Heap-Minimum(H)

```
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL do
{
```

```
5         if key[x] < min then
          {
6             min ← key[x]
7             y ← x
          }
8         x ← sibling[x]
    }
9   return y
```

Since a binomial heap is min-heap-ordered, the minimum key must reside in a root node. The BINOMIAL-HEAP-MINIMUM procedure checks all roots, which number at most $\lfloor \lg n \rfloor + 1$, saving the current minimum in min and a pointer to the current minimum in y. When called on the binomial heap of previous figure (13 nodes), BINOMIAL-HEAP-MINIMUM returns a pointer to the node with key 1.

Because there are at most $\lfloor \lg n \rfloor + 1$ roots to check, the running time of BINOMIAL-HEAP-MINIMUM is O(log n).

- **Uniting two binomial heaps :**

The operation of uniting two binomial heaps is used as a subroutine by most of the remaining operations.

The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the B_{k-1} tree rooted at node y to the B_{k-1} tree rooted at node z; that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

Binomial-Link(y, z)

```
1   p[y] ← z
2   sibling[y] ← child[z]
3   child[z] ← y
4   degree[z] ← degree[z] + 1
```

The BINOMIAL-LINK procedure makes node y the new head of the linked list of node z's children in O(1) time.

It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the tree: in a B_k tree, the leftmost child of the root is the root of a B_{k-1} tree.

The following procedure unites binomial heaps H1 and H2, returning the resulting heap. It destroys the representations of H1 and H2 in the process. Besides BINOMIAL-LINK, the procedure uses an auxiliary procedure BINOMIALHEAP-MERGE that merges the root lists of H1 and H2 into a single linked list that is sorted by degree into monotonically increasing order.

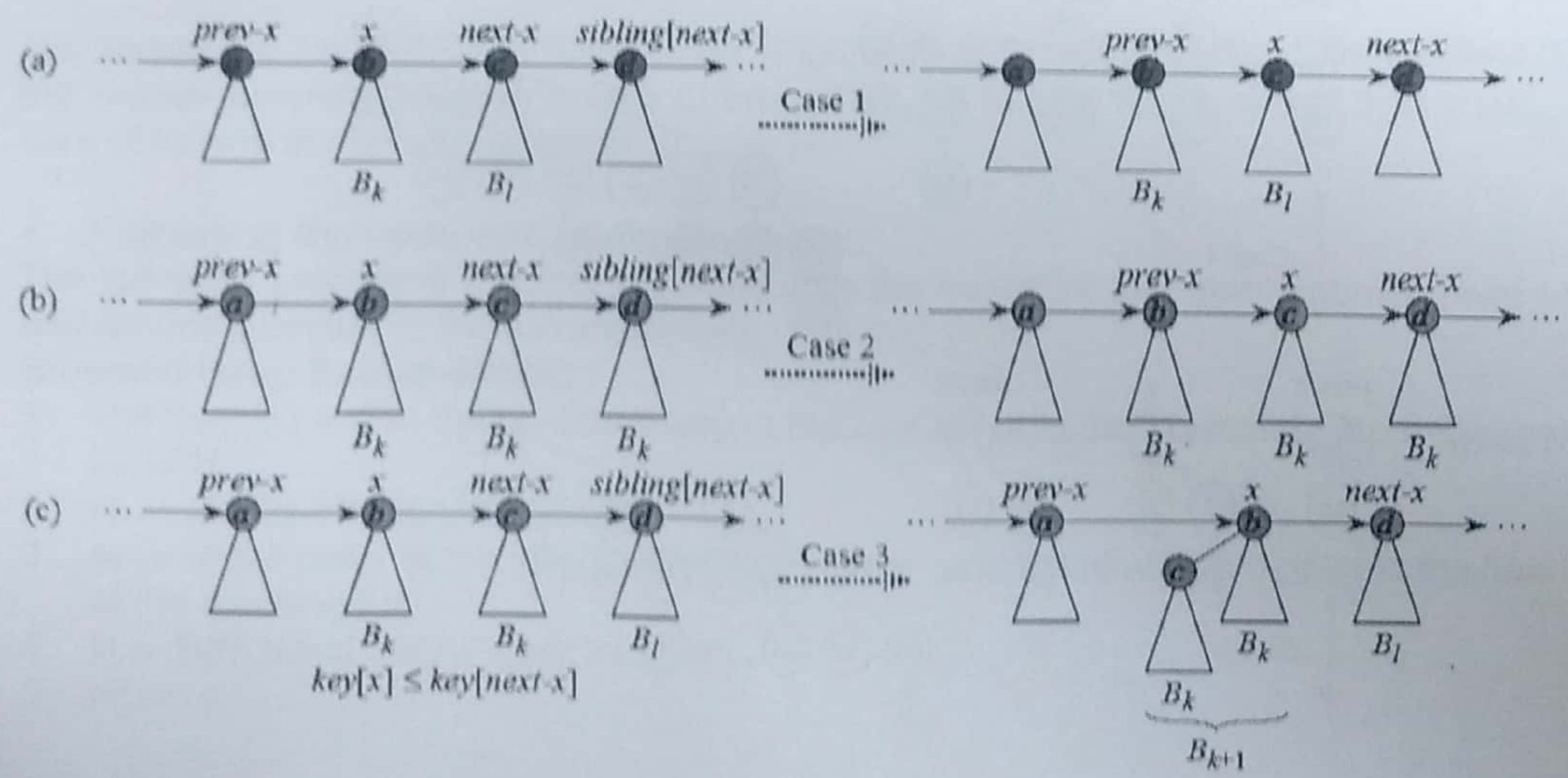
Binomial-Heap-Union(H1, H2)

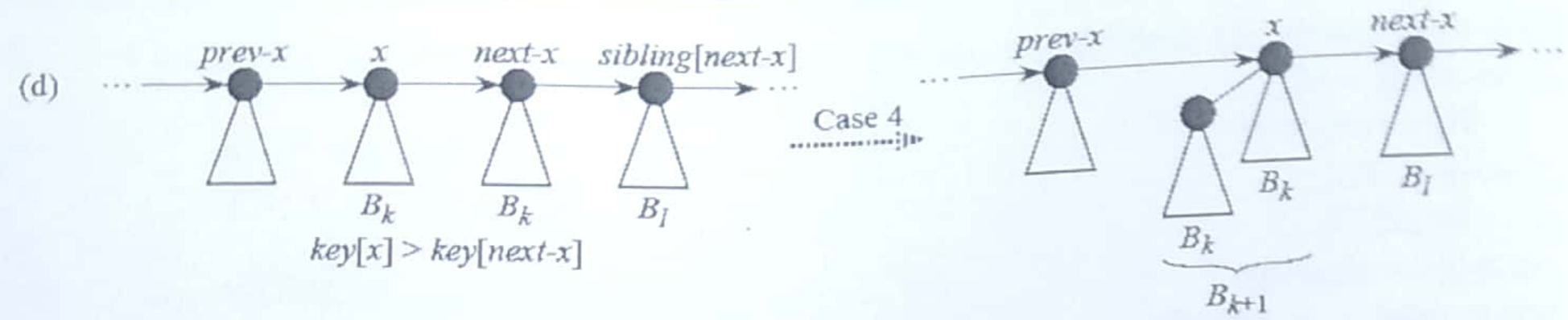
```
1   H ← MAKE-BINOMIAL-HEAP( )
2   head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
```

Scanned by CamScanner

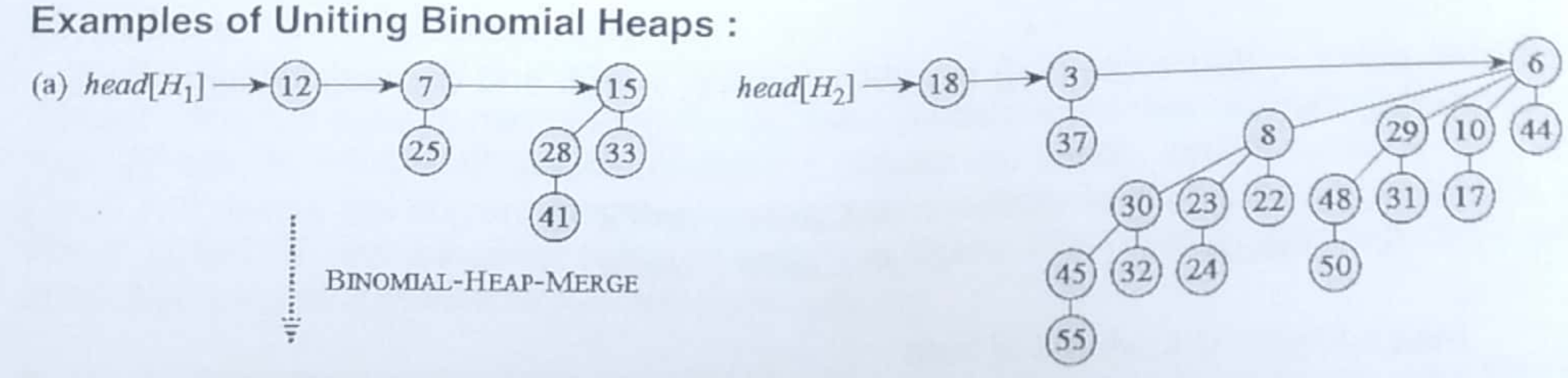
```
free the objects H1 and H2 but not the lists they point to
if head[H] = NIL
    then return H
prev-x ← NIL
x ← head[H]
next-x ← sibling[x]
while next-x ≠ NIL do
{
if (degree[x] ≠ degree[next-x]) or (sibling[next-x] ≠ NIL and degree[sibling[next-x]] =
degree[x]) then
    {
        prev-x ← x                      // Cases 1 and 2
        x ← next-x                      // Cases 1 and 2
    }
    else if key[x] ≤ key[next-x] then
    {
    sibling[x] ← sibling[next-x]        // Case 3
    BINOMIAL-LINK(next-x, x)            // Case 3
    }
    else if prev-x = NIL then           // Case 4
            head[H] ← next-x            // Case 4
    else
    {
        sibling[prev-x] ← next-x        // Case 4
    }
    BINOMIAL-LINK(x, next-x)            // Case 4
    x ← next-x                          // Case 4
    next-x ← sibling[x]
}                                       // end of while
return H
```

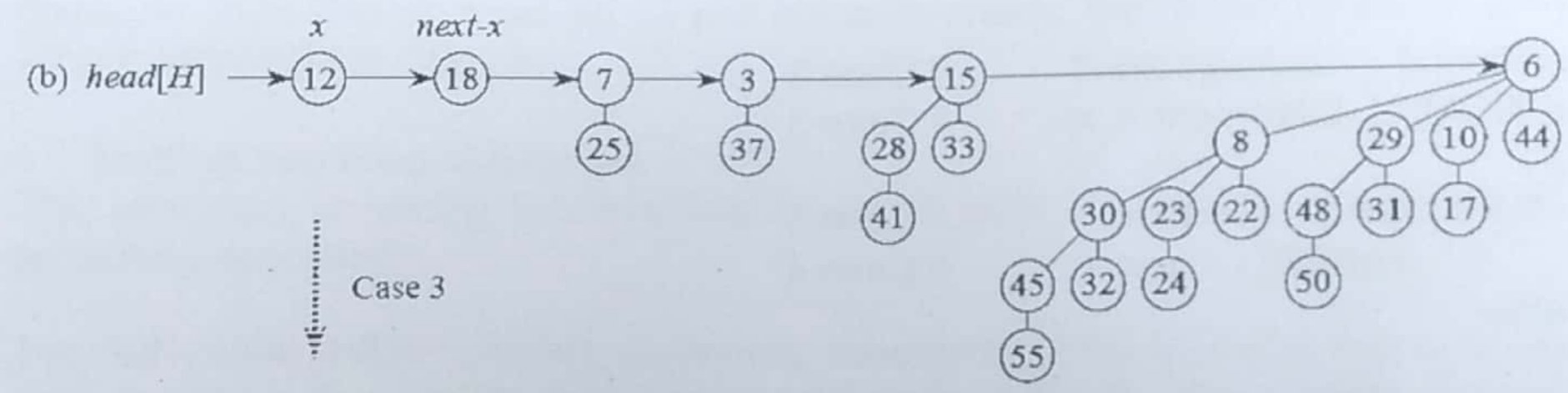
- **Cases of Uniting Binomial Heaps :**

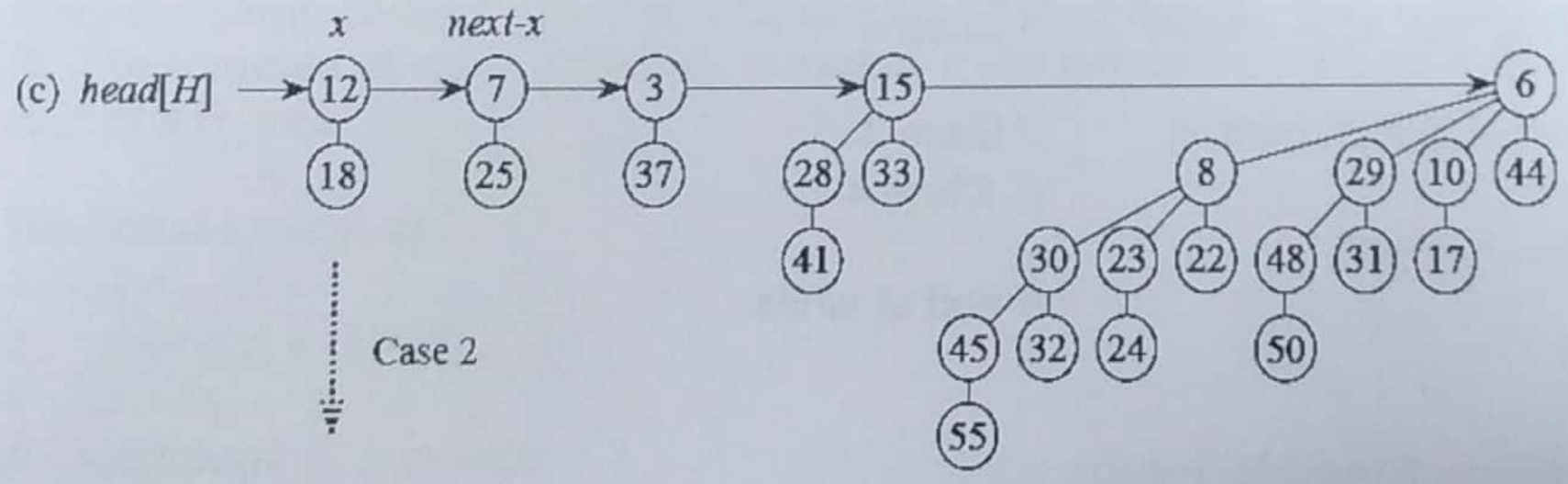


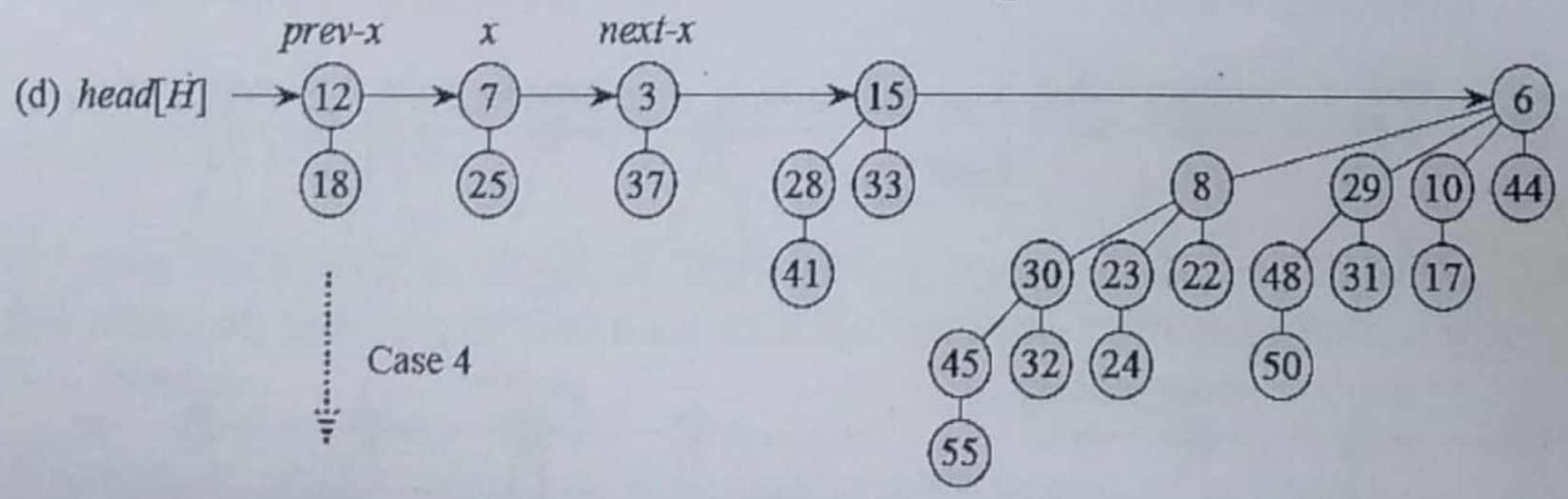


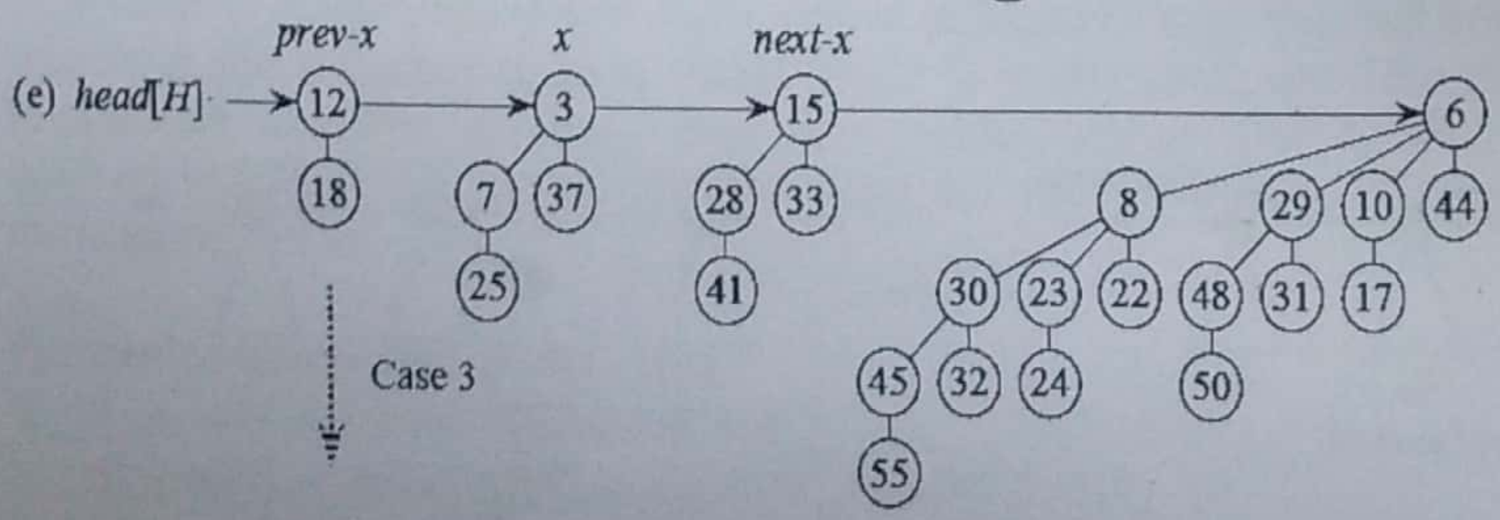
Examples of Uniting Binomial Heaps :



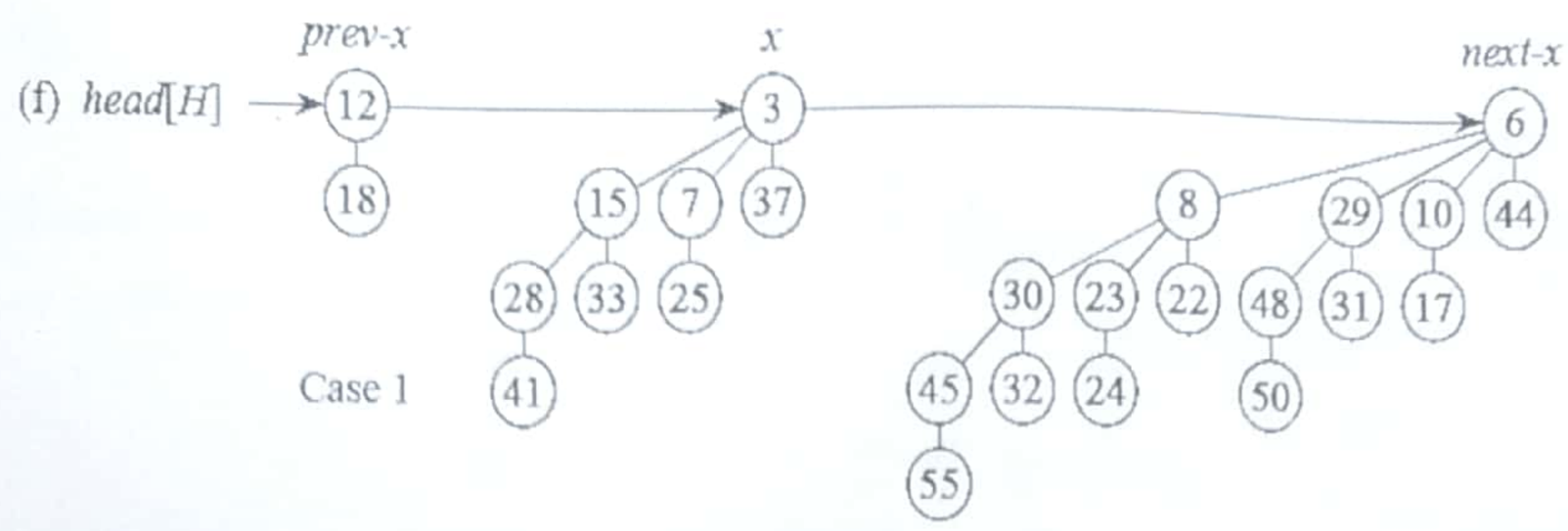








Scanned by CamScanner



Analysis :

The running time of BINOMIAL-HEAP-UNION is O(lg n), where n is the total number of nodes in binomial heaps H1 and H2. We can see this as follows. Let H1 contain n1 nodes and H2 contain n2 nodes, so that n = n1+n2. Then H1 contains at most ⌊lg n1⌋+1 roots and H2 contains at most ⌊lg n2⌋+1 roots, and so H contains at most ⌊lg n1⌋+⌊lg n2⌋+2 ≤ 2 ⌊lg n⌋+2 = O(lg n) roots immediately after the call of BINOMIAL-HEAP-MERGE. The time to perform BINOMIAL-HEAP-MERGE is thus O(lg n). Each iteration of the while loop takes O(1) time, and there are at most ⌊lg n1⌋ + ⌊lg n2⌋ + 2 iterations because each iteration either advances the pointers one position down the root list of H or removes a root from the root list. The total time is thus O(lg n).

- **Inserting a node :**

The following procedure inserts node x into binomial heap H, assuming that x has already been allocated and key[x] has already been filled in.

Binomial-Heap-Insert(H, x) :

```
H′ ← MAKE-BINOMIAL-HEAP()
p[x] ← NIL
child[x] ← NIL
sibling[x] ← NIL
degree[x] ← 0
head[H′] ← x
H ← BINOMIAL-HEAP-UNION(H, H′)
```

The procedure simply makes a one-node binomial heap H′ in O(1) time and unites it with the n-node binomial heap H in O(lg n) time. The call to BINOMIAL-HEAPUNION takes care of freeing the temporary binomial heap H′.

- **Extracting the node with minimum key :**

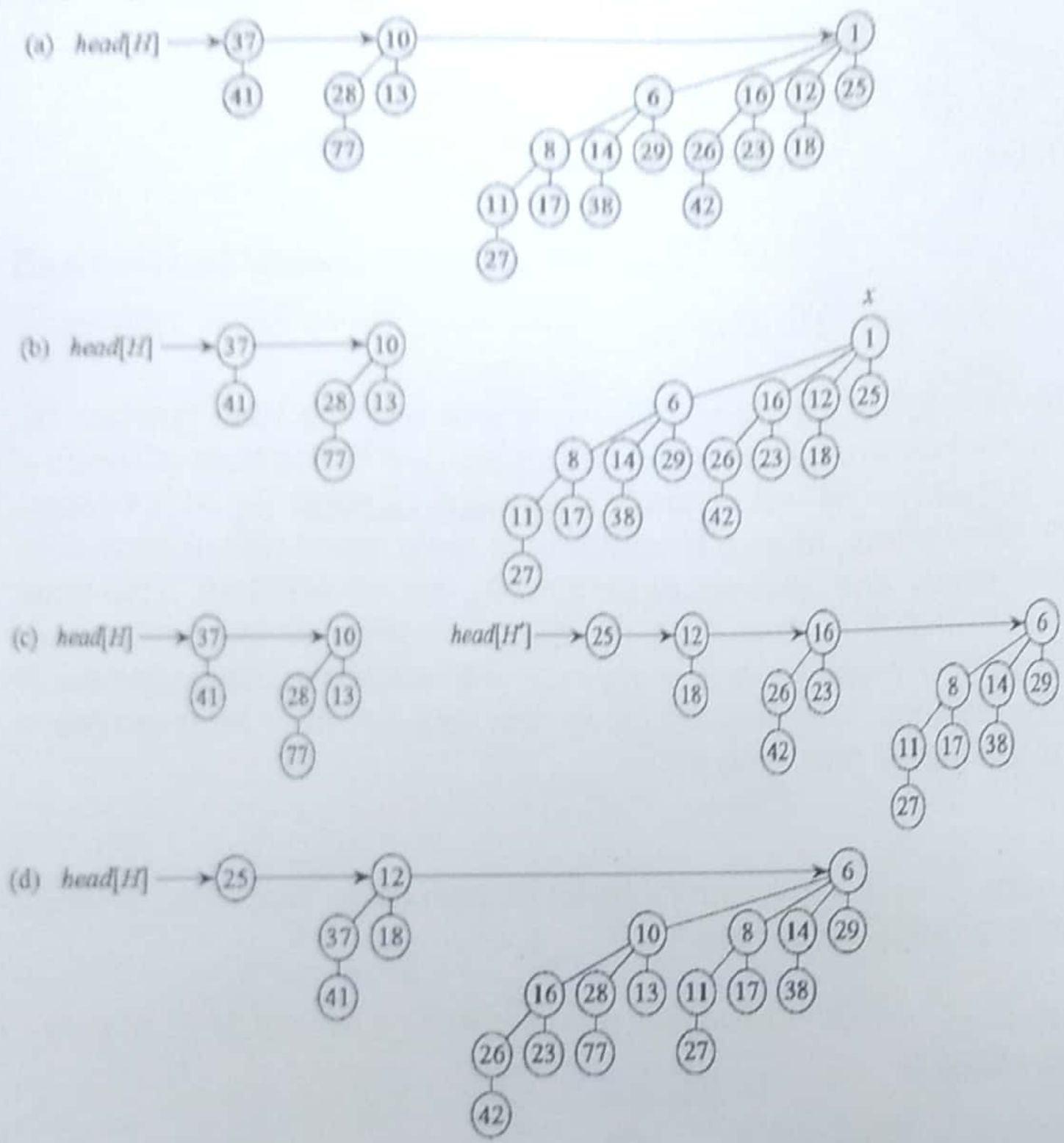
The following procedure extracts the node with the minimum key from binomial heap H and returns a pointer to the extracted node.

Binomial-Heap-Extract-Min(H) :

```
find the root x with the minimum key in the root list of H, and remove x from the root
list of H
H′ ← MAKE-BINOMIAL-HEAP()
reverse the order of the linked list of x's children, and set head[H′] to point to the head
of the resulting list
H ← BINOMIAL-HEAP-UNION(H, H′)
return x
```

Scanned by CamScanner

Example of Extracting Minimum Key :



Analysis :
Since each of lines 1–4 takes O(lg n) time if H has n nodes, BINOMIAL-HEAPEXTRACT-MIN runs in O(lg n) time.

- **Decreasing a key :**

The following procedure decreases the key of a node x in a binomial heap H to a new value k. It signals an error if k is greater than x's current key.

Binomial-Heap-Decrease-Key(H, x, k) :

```
1   if k > key[x] then
    {
2       error "new key is greater than current key"
3       return
    }
4   key[x] ← k
5   y ← x
6   z ← p[y]
7   while z ≠ NIL and key[y] < key[z] do
    {
8       exchange key[y] ↔ key[z]
9       y ← z
```

Scanned by CamScanner

```
10      z ← p[z]
    }
```

Example of Decrease Key : (node y has its key decreased to 7)

Analysis :

The BINOMIAL-HEAP-DECREASE-KEY procedure takes O(lg n) time. The maximum depth of x is ⌊lg n⌋, so the while loop of lines 6–10 iterates at most ⌊lg n⌋ times.

- **Deleting a key :**

It is easy to delete a node x's key and satellite information from binomial heap H in O(lg n) time. The following implementation assumes that no node currently in the binomial heap has a key of $-\infty$.

Binomial-Heap-Delete(H, x) :

```
1   BINOMIAL-HEAP-DECREASE-KEY(H, x,−∞)
2   BINOMIAL-HEAP-EXTRACT-MIN(H)
```

The BINOMIAL-HEAP-DELETE procedure makes node x have the unique minimum key in the entire binomial heap by giving it a key of $-\infty$. It then bubbles this key and the associated satellite information up to a root by calling BINOMIAL-HEAP-DECREASE-KEY. This root is then removed from H by a call of BINOMIAL-HEAP-EXTRACT-MIN. The BINOMIAL-HEAP-DELETE procedure takes O(lg n) time.

❑ ❑ ❑ ❑ ❑

Scanned by CamScanner

Module

4

Maximum Flow

SYLLABUS Weightage : 19 Marks

Maximum Flow

Flow networks, the ford Fulkerson method, max bipartite matching, push Relabel Algorithm, The relabel to front algorithm.

Help Line : For any query WhatsApp to 704 501 85 39 to get it Solved.

Scanned by CamScanner

FLOW NETWORKS

We can interpret a directed graph as a "flow network" and use it to answer questions about material flows.

Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed. The source produces the material at some steady rate, and the sink consumes the material at the same rate.

The "flow" of the material at any point in the system is the rate at which the material moves.

Flow networks can be used to model liquids flowing through pipes, parts through assembly lines, current through electrical networks, information through communication networks, and so forth.

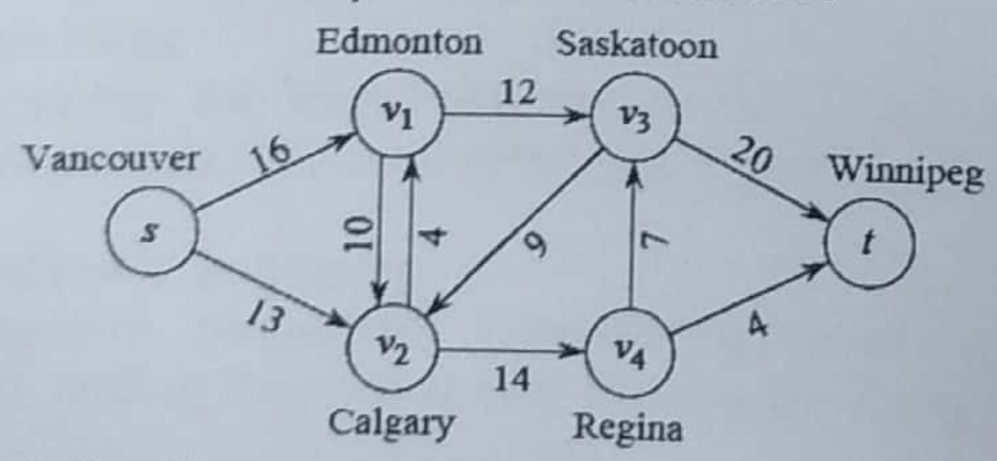
The rate at which material enters a vertex must equal the rate at which it leaves the vertex.

We call this property "flow conservation," and it is equivalent to Kirchhoff's Current Law when the material is electrical current.

In the maximum-flow problem, we wish to compute the greatest rate at which material can be shipped from the source to the sink without violating any capacity constraints.

Flow Networks and Flows :

A flow network G = (V, E) is a directed graph in which each edge (u, v) ∈ E has a nonnegative capacity c(u, v) ≥ 0.
If (u, v) ∉ E, we assume that c(u, v) = 0.
We distinguish two vertices in a flow network: a source s and a sink t.
The graph is connected, and |E| ≥ |V| – 1.
The following figure shows an example of a flow network :



We are now ready to define flows more formally.
Let G = (V, E) be a flow network with a capacity function c.
Let s be the source of the network, and let t be the sink.
A flow in G is a real-valued function f : V × V → R that satisfies the following three properties:
Capacity constraint : For all u, v ∈ V, we require f (u, v) ≤ c(u, v).
Skew symmetry: For all u, v ∈ V, we require f (u, v) = – f (v, u).
Flow conservation: For all u ∈ V – {s, t}, we require

$$\sum_{v \in V} f(u,v) = 0$$

Scanned by CamScanner

The quantity f(u, v), which can be positive, zero, or negative, is called the flow from vertex u to vertex v.

The value of a flow f is defined as :

$$|f| = \sum_{v \in V} f(s, v)$$

that is, the total flow out of the source.

In the maximum-flow problem, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value.

Let us briefly explore the three flow properties.
The capacity constraint simply says that the flow from one vertex to another must not exceed the given capacity.

Skew symmetry is a notational convenience that says that the flow from a vertex u to a vertex v is the negative of the flow in the reverse direction.

The flow-conservation property says that the total flow out of a vertex other than the source or sink is 0. By skew symmetry, we can rewrite the flow-conservation property as

$$\sum_{v \in V} f(u, v) = 0$$

for all $v \in V - \{s, t\}$. That is, the total flow into a vertex is 0.

When neither (u, v) nor (v, u) is in E, there can be no flow between u and v, and f (u, v) = f (v, u) = 0.

Our last observation concerning the flow properties deals with flows that are positive. The total positive flow entering a vertex v is defined by

$$\sum_{\substack{v \in V \\ f(u, v) > 0}} f(u, v)$$

The total positive flow leaving a vertex is defined symmetrically. We define the total net flow at a vertex to be the total positive flow leaving a vertex minus the total positive flow entering a vertex.

One interpretation of the flow-conservation property is that the total positive flow entering a vertex other than the source or sink must equal the total positive flow leaving that vertex. This property, that the total net flow at a vertex must equal 0, is often informally referred to as "flow in equals flow out."

THE FORD-FULKERSON METHOD

The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems: residual networks, augmenting paths, and cuts.

These ideas are essential to the important max-flow min-cut theorem which characterizes the value of a maximum flow in terms of cuts of the flow network.

Scanned by CamScanner

The Ford-Fulkerson method is iterative.
We start with f (u, v) = 0 for all u, v ∈ V, giving an initial flow of value 0.
At each iteration, we increase the flow value by finding an "augmenting path," which we can think of simply as a path from the source s to the sink t along which we can send more flow, and then augmenting the flow along this path.

We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

Ford-Fulkerson-Method(G, s, t)

```
initialize flow f to 0
while there exists an augmenting path p
do augment flow f along p
return f
```

Residual networks :
Given a flow network and a flow, the residual network consists of edges that can admit more flow.

More formally, suppose that we have a flow network G = (V, E) with source s and sink t.
Let f be a flow in G, and consider a pair of vertices u, v ∈ V.

The amount of additional flow we can push from u to v before exceeding the capacity c(u, v) is the residual capacity of (u, v), given by

$$c_f(u, v) = c(u, v) - f(u, v)$$

For example, if c(u, v) = 16 and f (u, v) = 11, then we can increase f(u, v) by $c_f(u, v) = 5$ units before we exceed the capacity constraint on edge (u, v).

When the flow f(u, v) is negative, the residual capacity $c_f(u, v)$ is greater than the capacity c(u, v).
For example, if c(u, v) = 16 and f(u, v) = –4, then the residual capacity $c_f(u, v)$ is 20.

We can interpret this situation as follows :
There is a flow of 4 units from v to u, which we can cancel there by pushing a flow of 4 units from u to v.
We can then push another 16 units from u to v before violating the capacity constraint on edge (u, v).

We have thus pushed an additional 20 units of flow, starting with a flow f (u, v) = –4, before reaching the capacity constraint.

Given a flow network G = (V, E) and a flow f, the residual network of G induced by f is

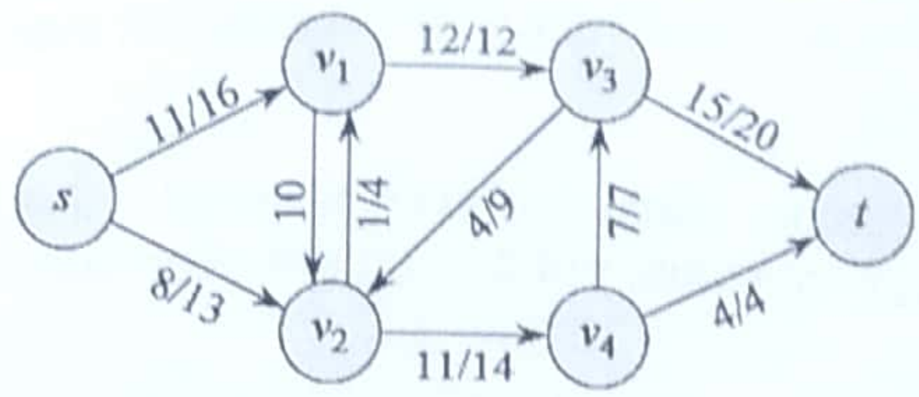
$$G_f = (V, E_f)$$

where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$

That is, each edge of the residual network, or residual edge, can admit a flow that is greater than 0.

Scanned by CamScanner

Figure shows the corresponding residual network G_f of the flow network shown earlier.

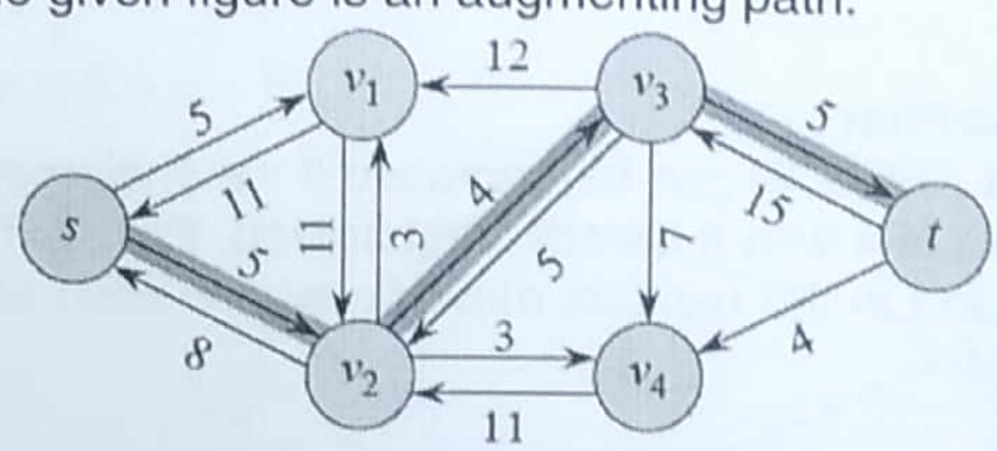


Augmenting Paths :

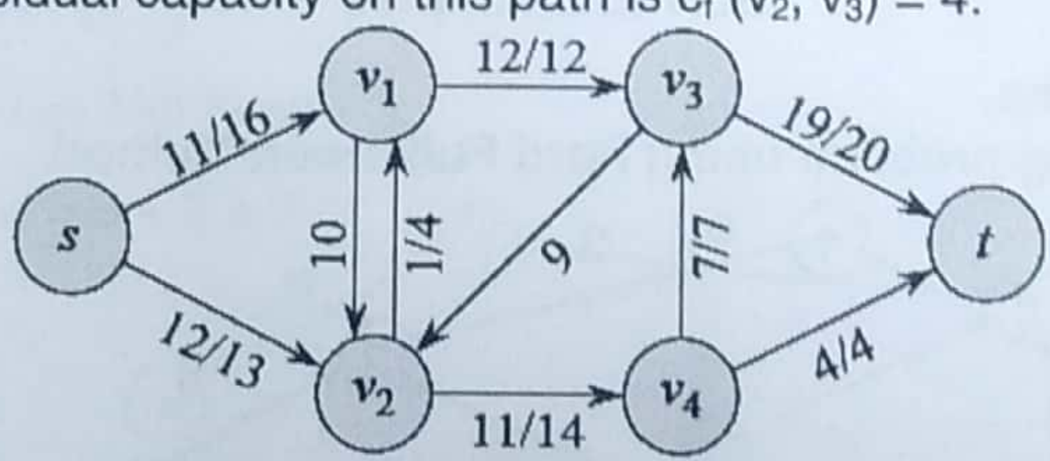
Given a flow network G = (V, E) and a flow f, an augmenting path p is a simple path from s to t in the residual network G_f.

By the definition of the residual network, each edge (u, v) on an augmenting path admits some additional positive flow from u to v without violating the capacity constraint on the edge.

The shaded path in the given figure is an augmenting path.

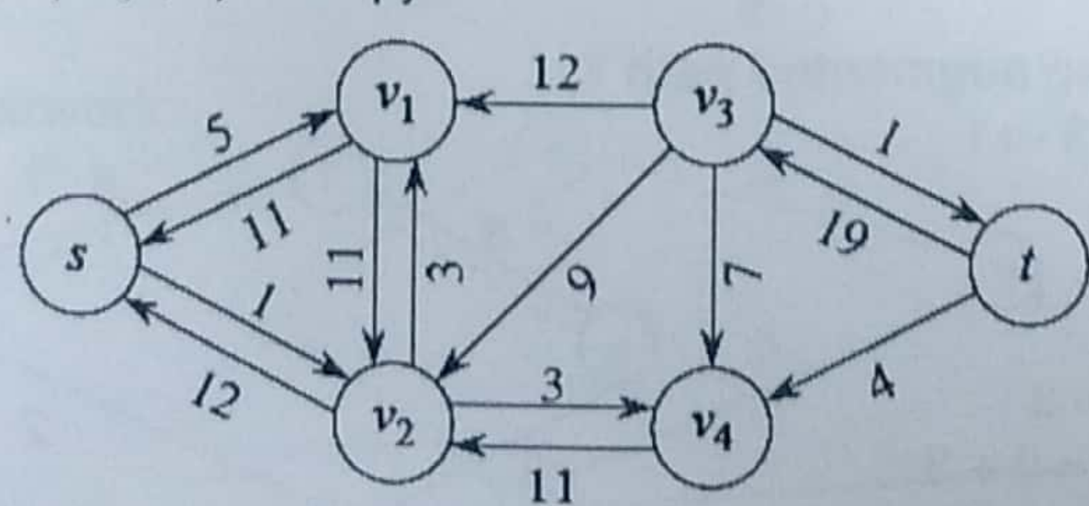


Treating the residual network G_f in the figure as a flow network, we can increase the flow through each edge of this path by up to 4 units without violating a capacity constraint, since the smallest residual capacity on this path is $c_f(v_2, v_3) = 4$.



We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the residual capacity of p, given by

$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is on } p\}$



Scanned by CamScanner

Analysis of Ford-Fulkerson :
The running time of FORD-FULKERSON depends on how the augmenting path p in line 4 is determined.

If it is chosen poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value.

If the augmenting path is chosen by using a breadth-first search, however, the algorithm runs in polynomial time.

A straightforward implementation of FORD-FULKERSON runs in time $O(E|f^*|)$, where f^* is the maximum flow found by the algorithm.

The analysis is as follows. Lines 1-3 take time $\Theta(E)$. The while loop of lines 4-8 is executed at most $|f^*|$ times, since the flow value increases by at least one unit in each iteration.

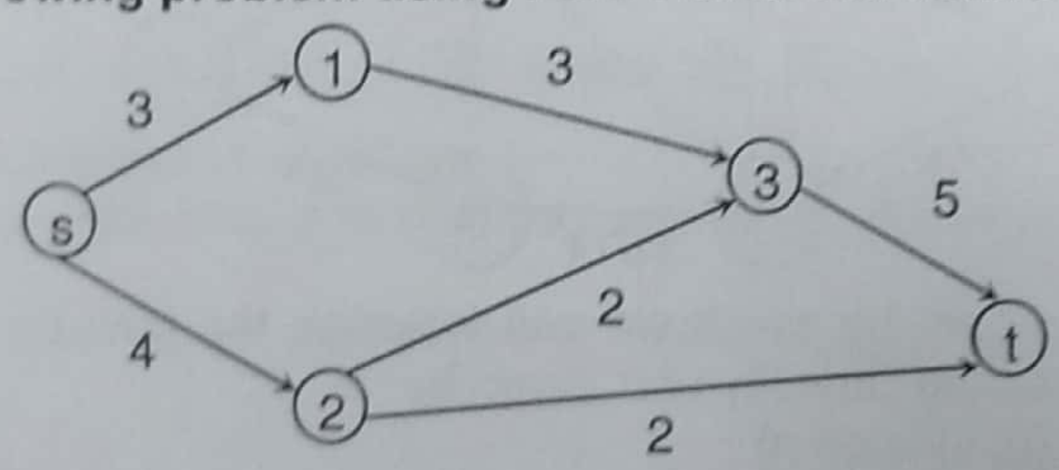
The Edmonds-Karp algorithm :
The bound on FORD-FULKERSON can be improved if we implement the computation of the augmenting path p in line 4 with a breadth-first search, that is, if the augmenting path is a shortest path from s to t in the residual network, where each edge has unit distance (weight).

We call the Ford-Fulkerson method so implemented the Edmonds-Karp algorithm. The Edmonds-Karp algorithm runs in $O(V E^2)$ time.

Solved Examples

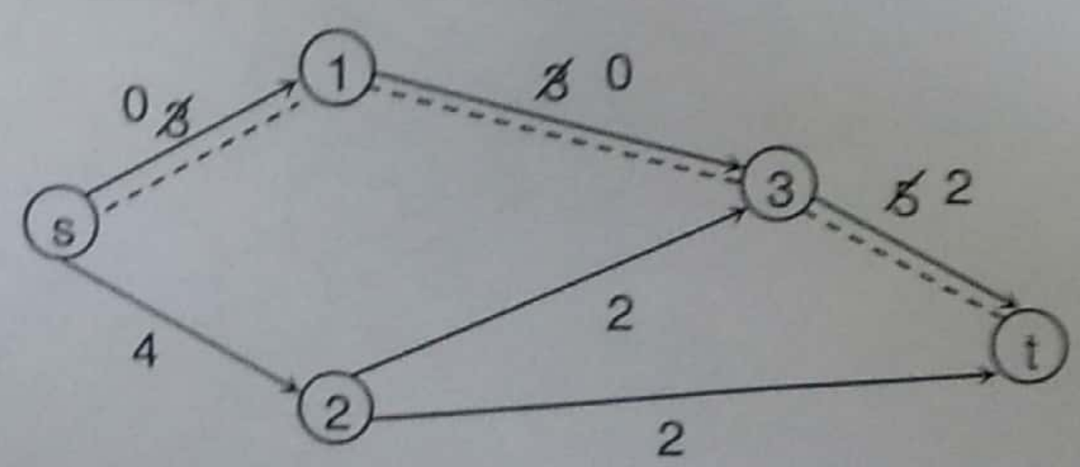
Network Flow Problems

Ex.1 Solve following problem using Ford Fulkerson Method.



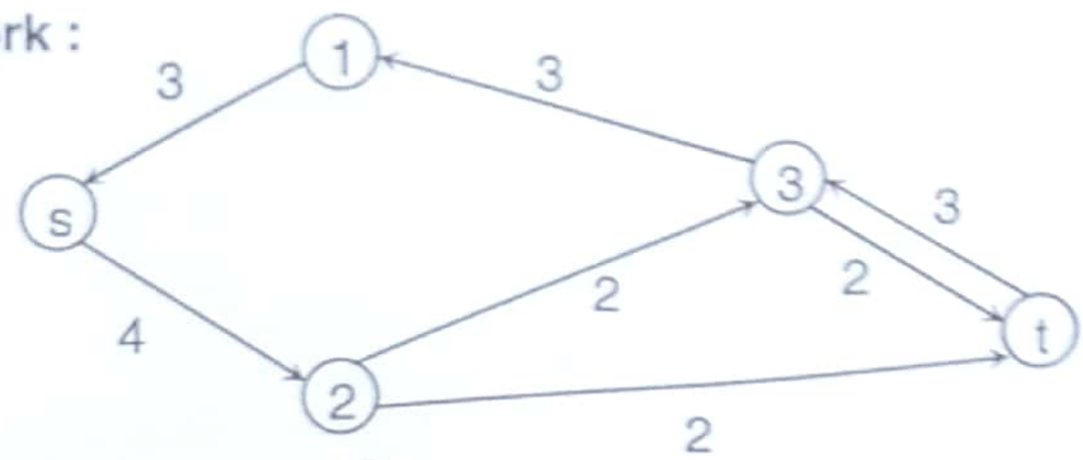
Ans.: Step 1 : Finding augmented path 1

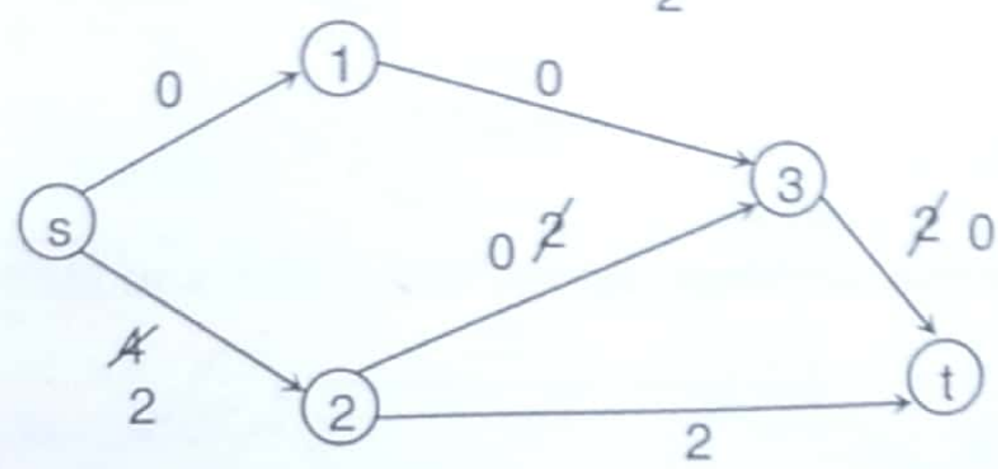
* $s \to 1 \to 3 \to t$
$s \to 1 = 3$
$1 \to 3 = 3$
$3 \to t = 5$
$\therefore$ Min. Flow = 3
$\therefore$ Total Flow = 0 + 3



Scanned by CamScanner

Residual Network :



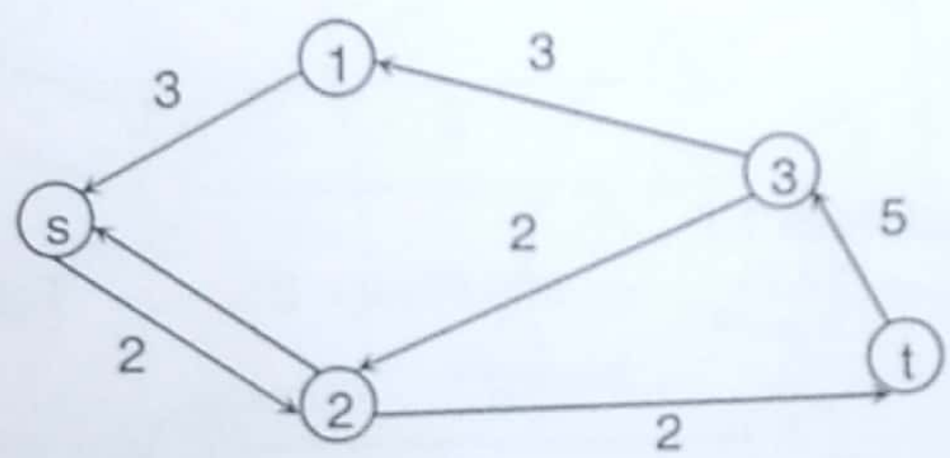


Step 2 :

$s \xrightarrow[4]{} 2 \xrightarrow[2]{} 3 \xrightarrow[2]{} t \Rightarrow$ Min Flow = 2

$\therefore$ Total Flow = 3 + 2 = 5

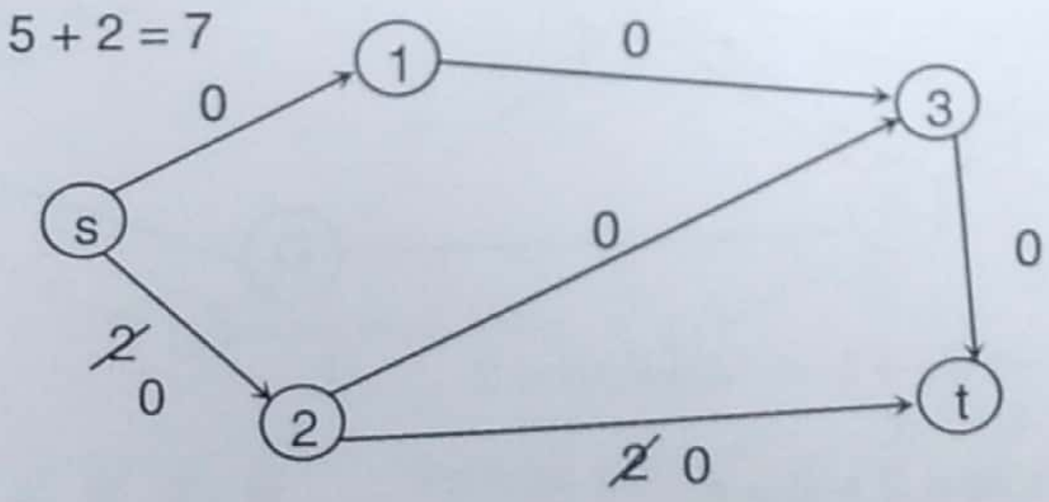
Residual Network :



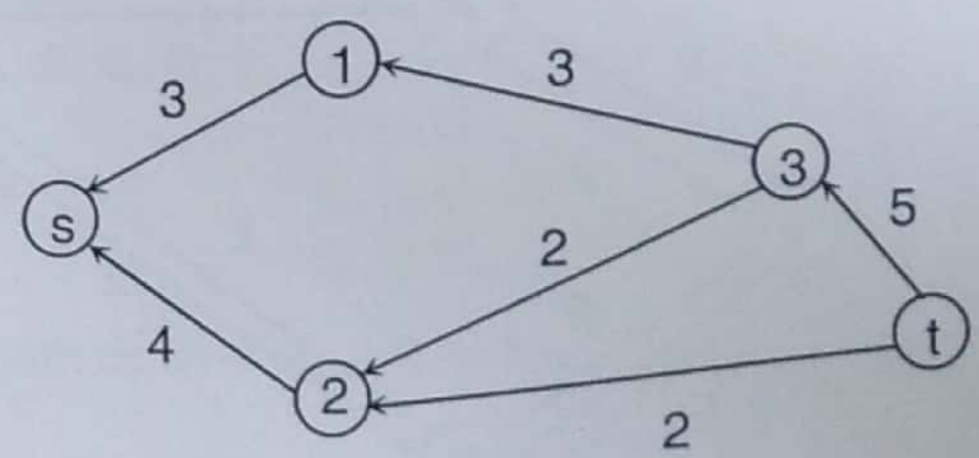
Step 3 :

$s \xrightarrow[2]{} 2 \xrightarrow[2]{} t \Rightarrow$ Min Flow = 2

$\therefore$ Total Flow = 5 + 2 = 7



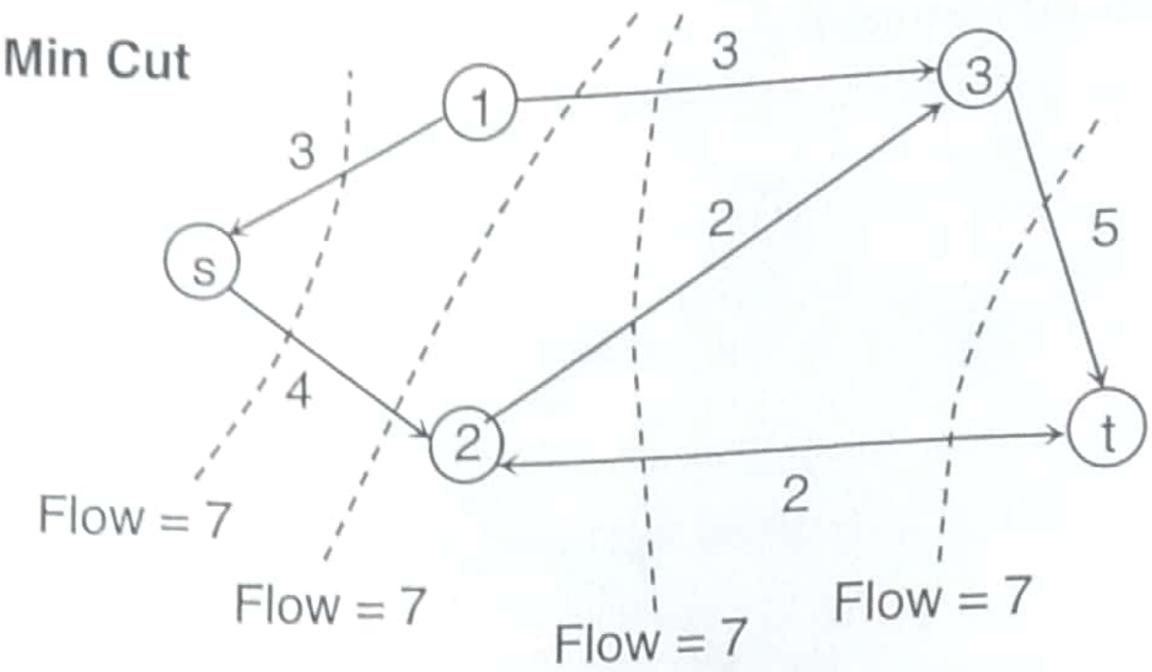
Residual Network :



$\therefore$ Total Flow = 7

Scanned by CamScanner

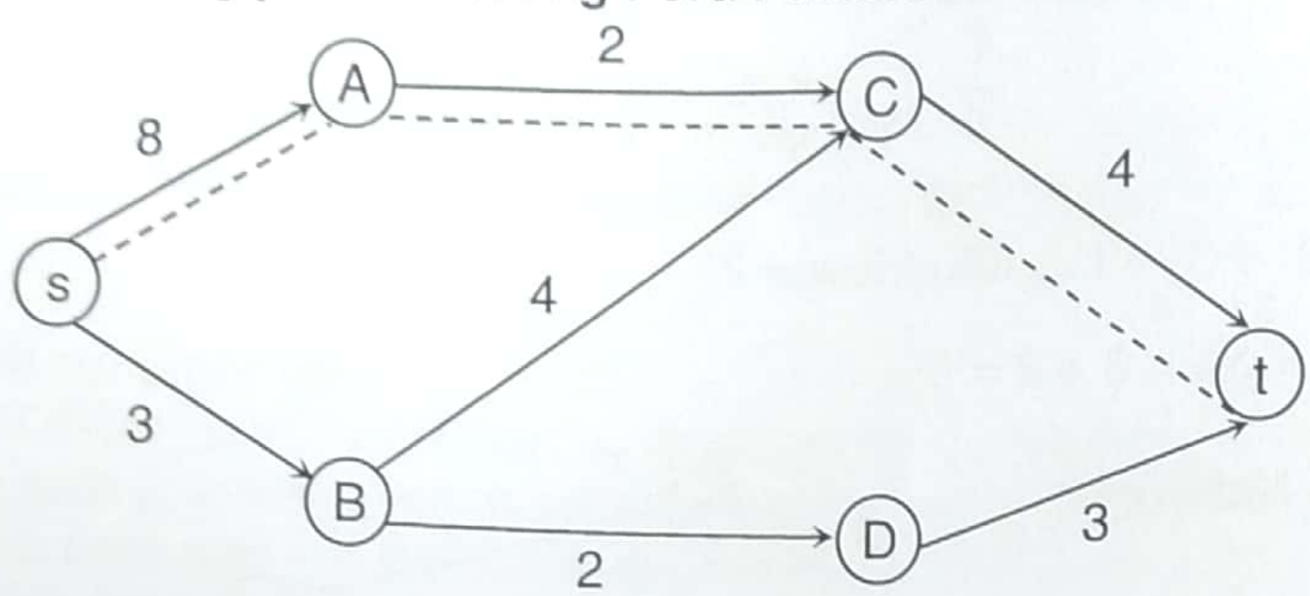
Method 2 : Using Min Cut



∴ Min Cut = 7

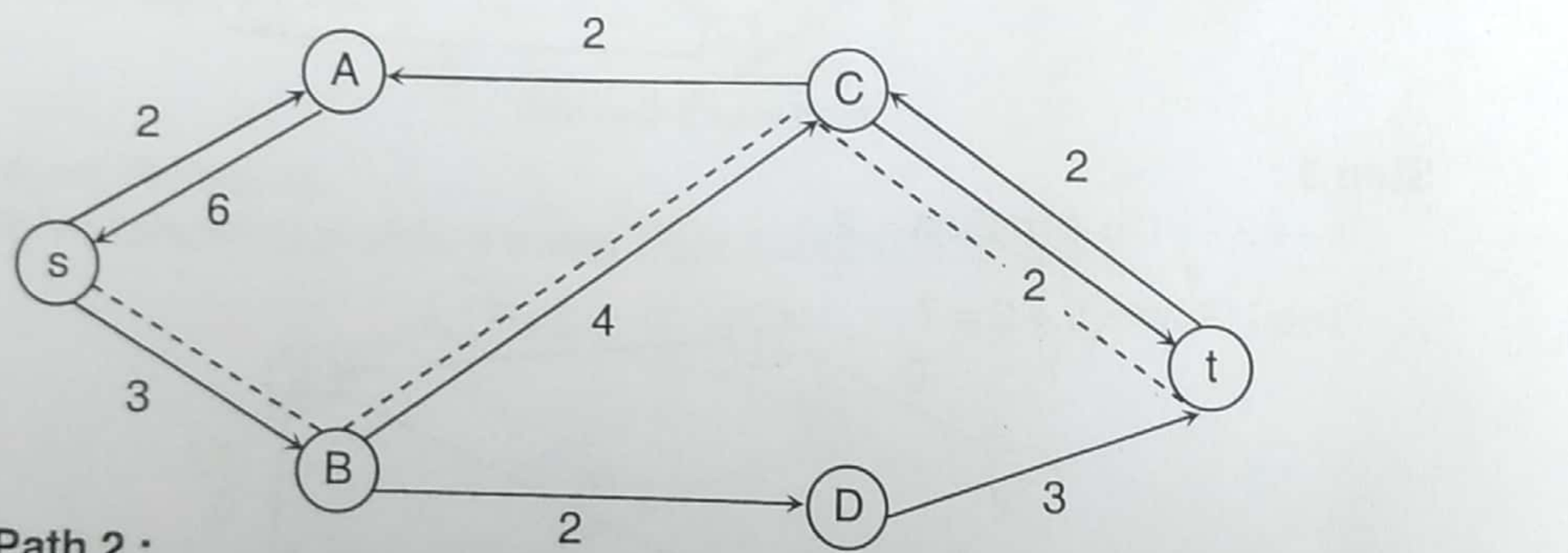
∴ Max Flow = 7

Ex.2 **Solve following problem using Ford Fulkerson Method.**



Ans.: **Augmented Path 1 :**

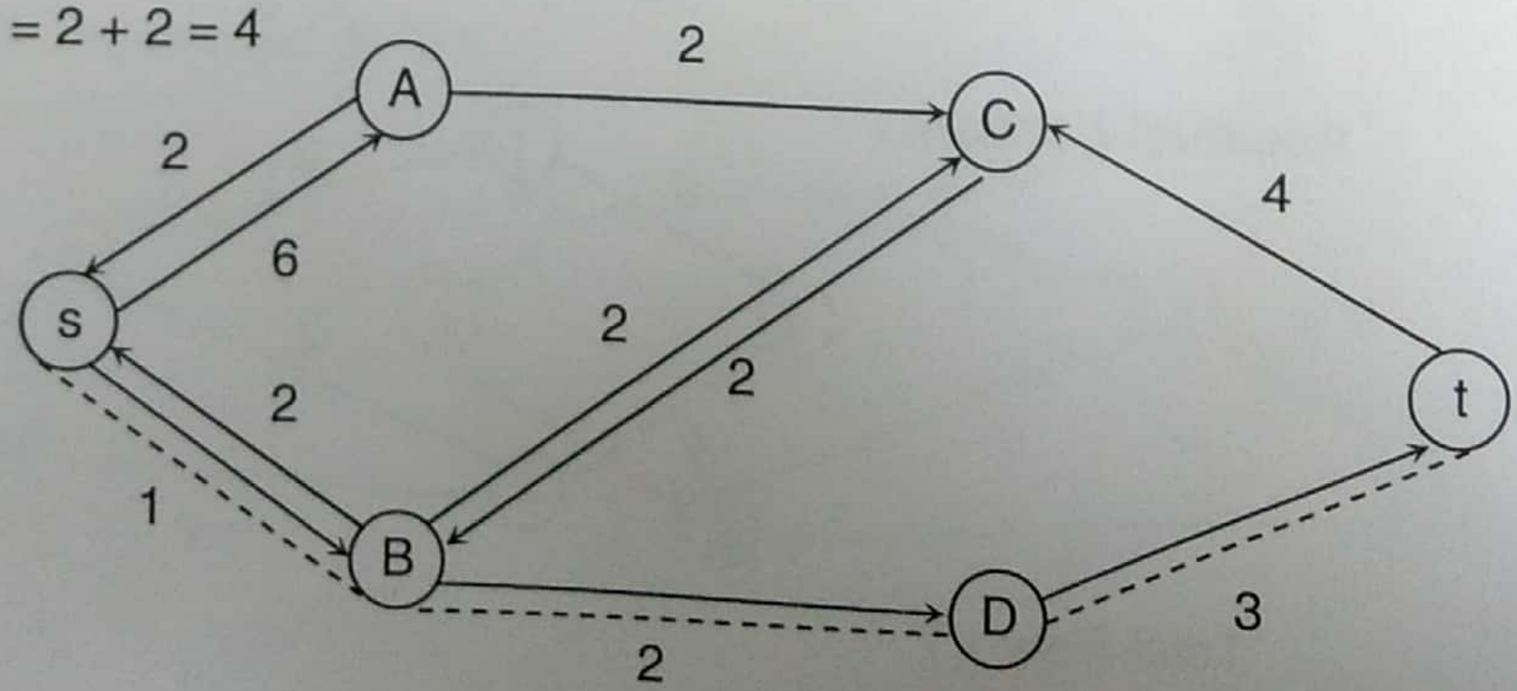
$S \xrightarrow[8]{} A \xrightarrow[2]{} C \xrightarrow[4]{} t$ ⇒ Min Flow = 2 ∴ Total flow = 0 + 2 = 2



Path 2 :

$S \xrightarrow[3]{} B \xrightarrow[4]{} C \xrightarrow[2]{} t$ ⇒ Min flow = 2

Total Flow = 2 + 2 = 4

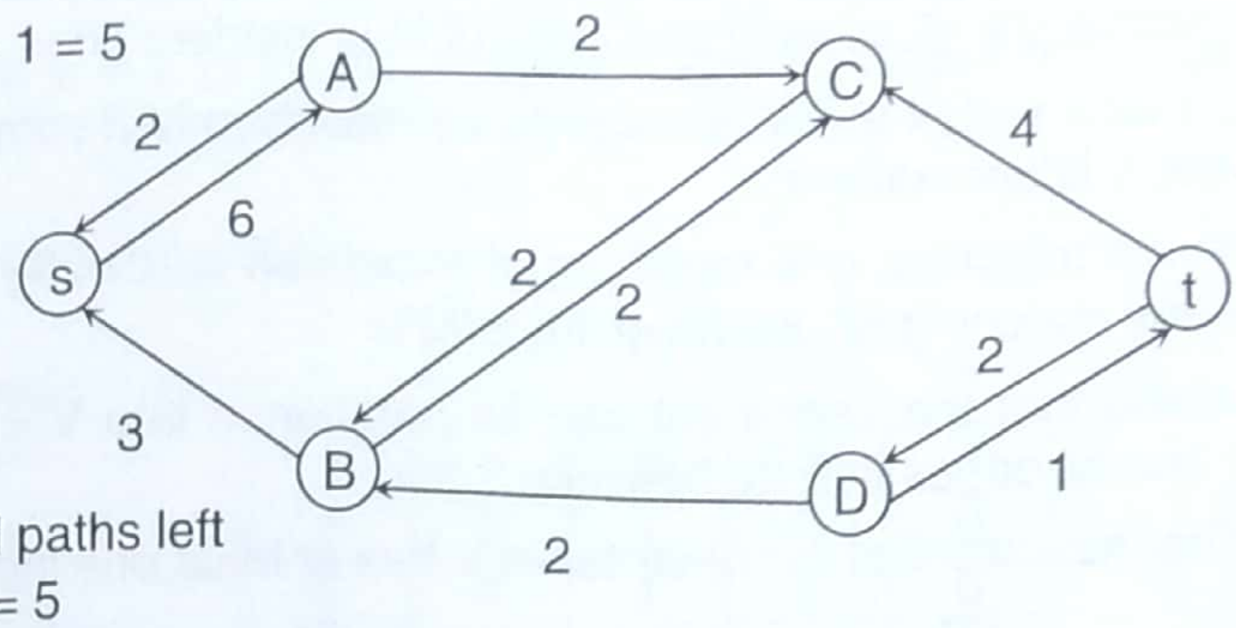


Scanned by CamScanner

Path 3 :

$S \underset{1}{\rightarrow} B \underset{2}{\rightarrow} D \underset{3}{\rightarrow} t \Rightarrow$ Min Flow = 1

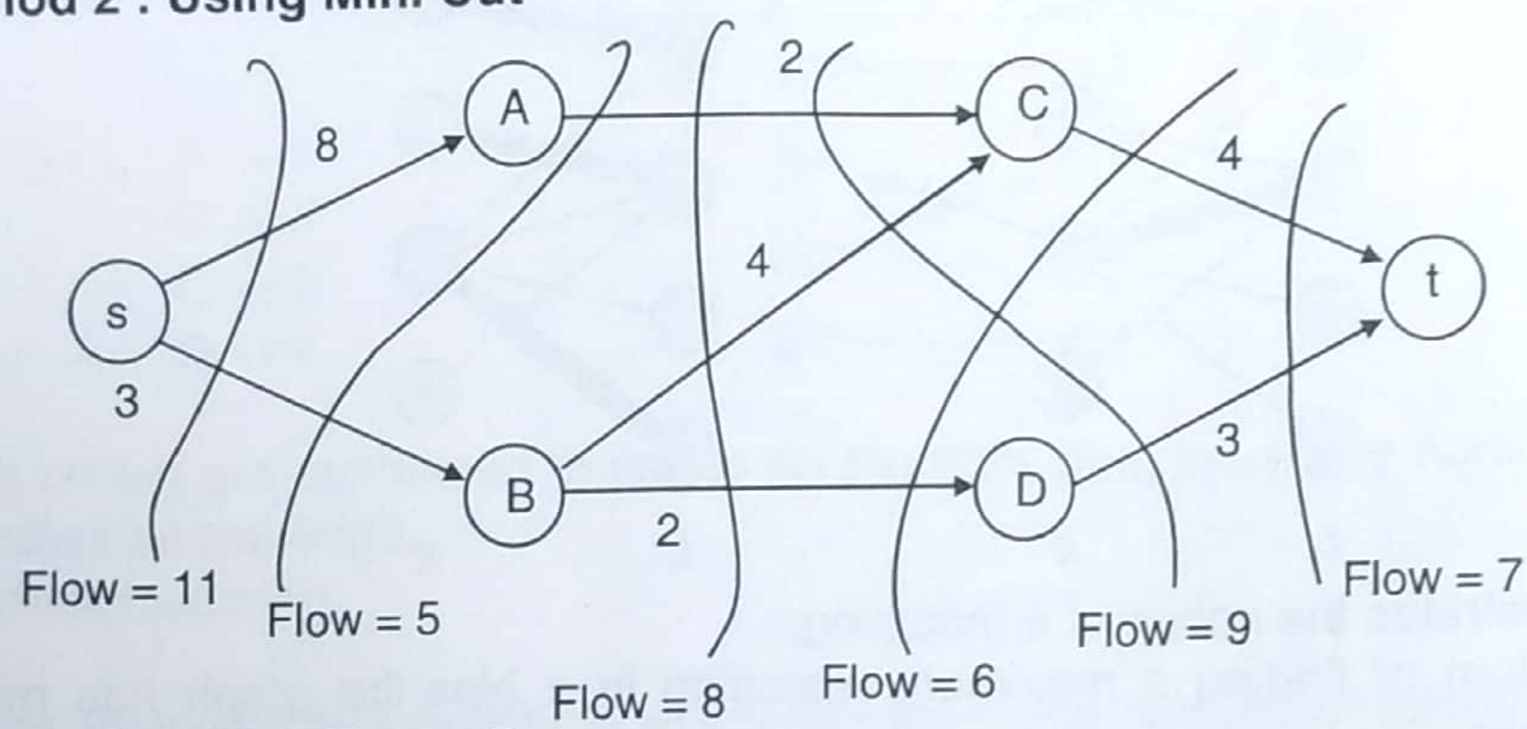
Total Flow = 4 + 1 = 5



No more augmented paths left

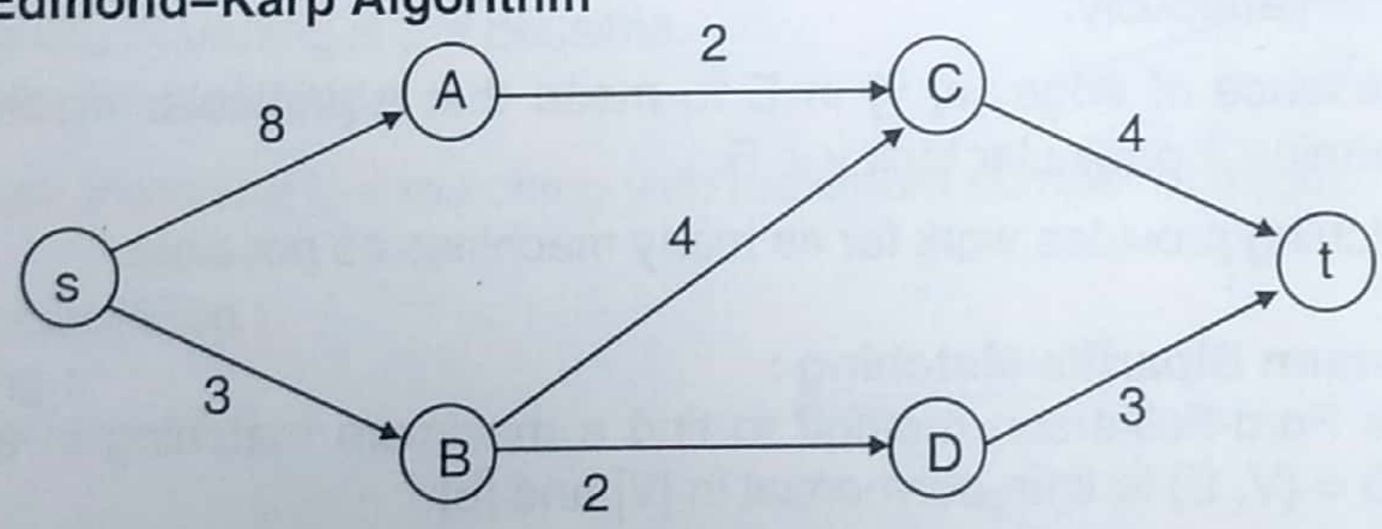
$\therefore$ Total Max Flow = 5

Method 2 : Using Min. Cut



$\therefore$ Min cut = Max Flow = 5

- **Edmond–Karp Algorithm**



Queue $\Rightarrow$ ~~A~~, ~~B~~, ~~C~~, D

Visited $\Rightarrow$ s, A, B, C, D, t

Current $\Rightarrow$ ~~s~~, A, B, C

Parent List

$s \rightarrow A$

$s \rightarrow B$

$A \rightarrow C$

$B \rightarrow D$

$C \rightarrow t$

$\therefore$ Augmented path $\Rightarrow s \rightarrow A \rightarrow C \rightarrow t$

Scanned by CamScanner

MAXIMUM BIPARTITE MATCHING

Given an undirected graph G = (V, E), a matching is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.

We say that a vertex $v \in V$ is matched by matching M if some edge in M is incident on v; otherwise, v is unmatched.

A maximum matching is a matching of maximum cardinality, that is, a matching M such that for any matching M′, we have $|M| \geq |M'|$.

We assume that the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R.

We further assume that every vertex in V has at least one incident edge.

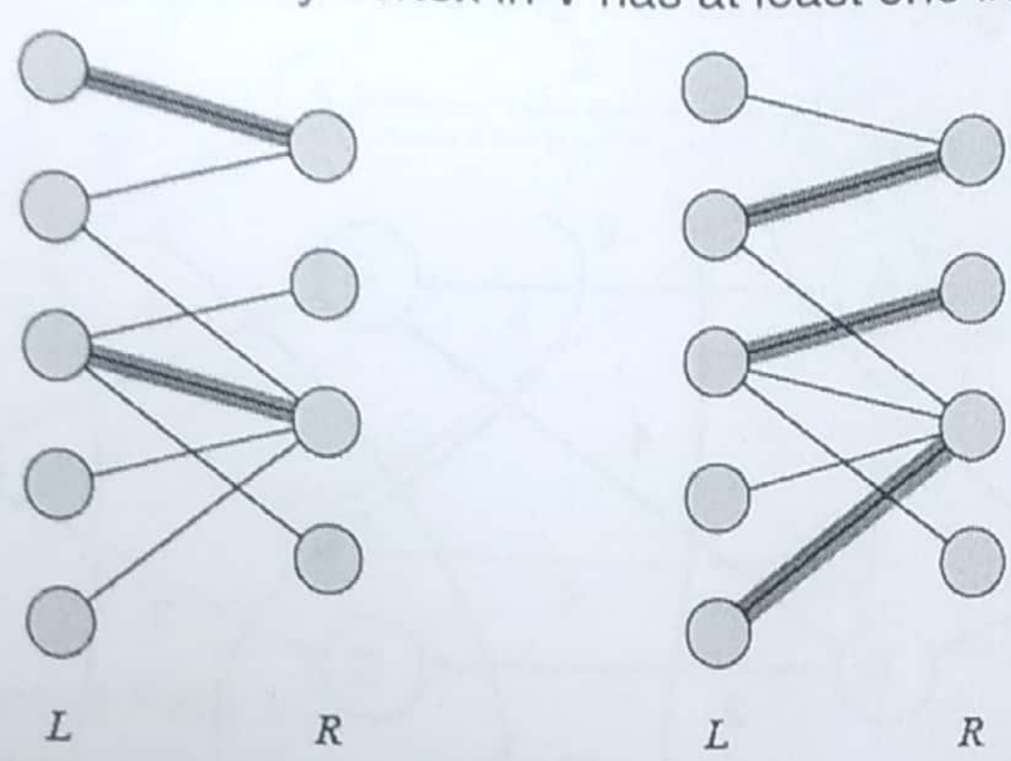


Figure illustrates the notion of a matching.

The problem of finding a maximum matching in a bipartite graph has many practical applications.

As an example, we might consider matching a set L of machines with a set R of tasks to be performed simultaneously.

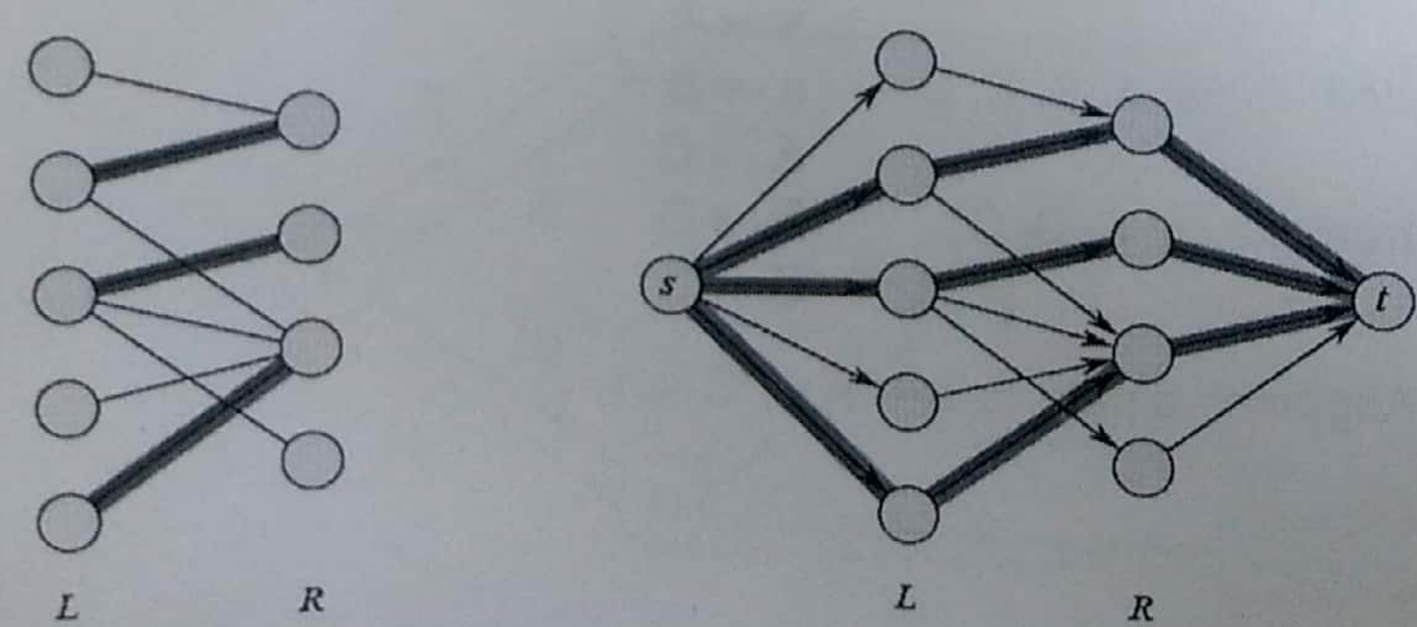
We take the presence of edge (u, v) in E to mean that a particular machine $u \in L$ is capable of performing a particular task $v \in R$.

A maximum matching provides work for as many machines as possible.

Finding a Maximum Bipartite Matching :

We can use the Ford-Fulkerson method to find a maximum matching in an undirected bipartite graph G = (V, E) in time polynomial in |V| and |E|.

The trick is to construct a flow network in which flows correspond to matchings, as shown in Figure.



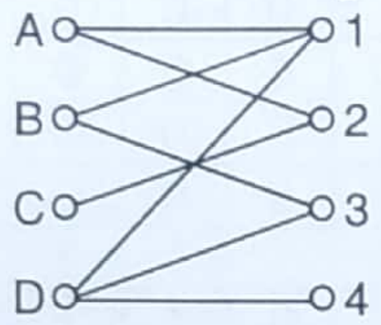
Scanned by CamScanner

We define the corresponding flow network $G' = (V', E')$ for the bipartite graph G as follows. We let the source s and sink t be new vertices not in V, and we let $V' = V \cup \{s, t\}$.

If the vertex partition of G is $V = L \cup R$, the directed edges of G' are the edges of E, directed from L to R, along with V new edges:

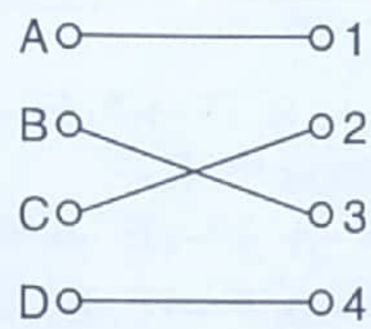
$E' = \{(s, u) : u \in L\}$
$\cup\{(u, v) : u \in L, v \in R, \text{ and } (u, v) \in E\}$
$\cup\{(v, t) : v \in R\}$

Maximum Bipartite Matching :



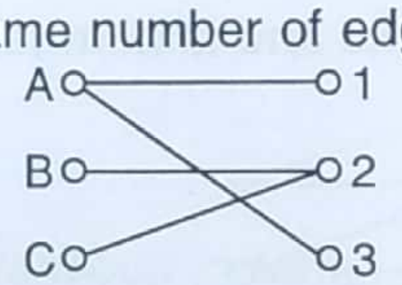
	1	2	3	4
A	1	1	0	0
B	1	1	1	0
C	0	1	0	0
D	1	0	1	1

Initial matching : $A \to 1$
$B \to 3$
$C \to 2$
$D \to 4$



$\because$ All nodes on left are connected to nodes on the right side, i.e. every node on either side is connected by one edge.
$\therefore$ It is complete matching.

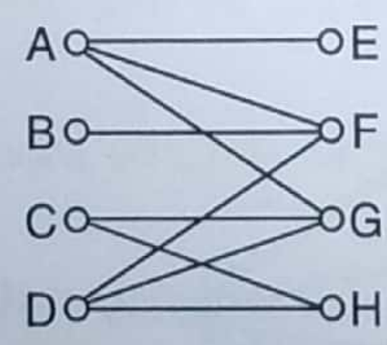
- A complete matching is a matching which has the same number of edges as nodes on one side.
  Initial matching = $1 \to A$, $2 \to B$
  $\therefore$ Complete matching is not possible.
  It's a maximum matching.



- A maximum matching is a matching with maximum number of edges.

Improving a matching :

Initial matching :
$A \to E$
$C \to H$
$D \to F$
It's not complete matching B on left and G on right are not connected.



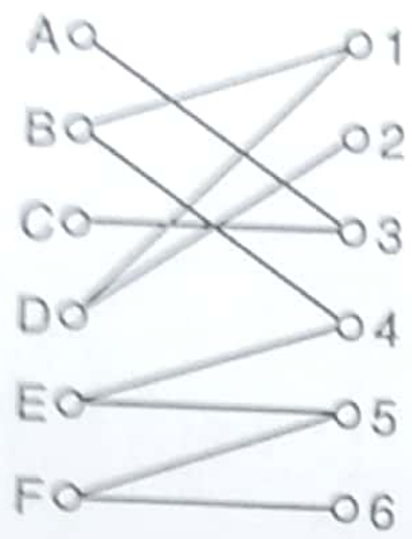
- Matching can be improved by finding an alternating path starting from non-connected node from left to non-connected node on right.
- start at B and look for G.
- From left to right $\to$ non-connected edges
  From right to left $\to$ connected edge
  $\therefore$ $B \not\to F \to D \not\to G$ ... Alternating path
- For getting better matching, use alternating path and switch connections.

Scanned by CamScanner

Connect : $B \to F$
Disconnect : $F \to D$
Connect : $D \to G$
$\therefore$ maximum matching : $A \to E$ $B \to F$
$C \to H$ $D \to G$

It's a complete matching.

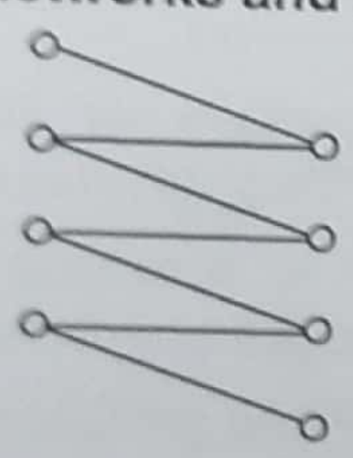
Example :



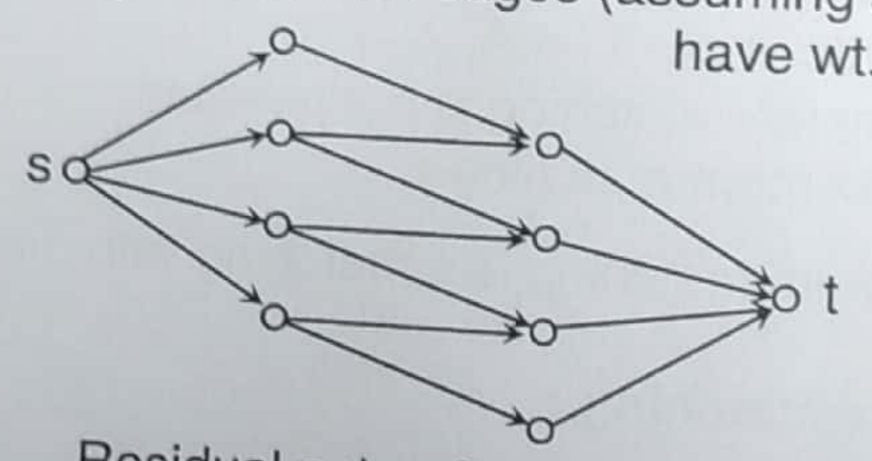
	1	2	3	4	5	6
A	0	0	1	0	0	0
B	1	0	0	1	0	0
C	0	0	1	0	0	1
D	1	1	0	0	0	0
E	0	0	0	1	1	0
F	0	0	0	0	1	1

Initial matching : $B \to 4, C \to 3, D \to 1, E \to 5, F \to 6$
Improving : Start at A and look for 2.
$A \not\to 3 \to C \not\to 6 \to F \not\to 5 \to E \not\to 4 \ldots \to B \not\to 1 \to D \not\to 2$
Connect : $A \to 3, C \to 6, F \to 5, E \to 4, B \to 1, D \to 2$
Disconnect : $3 \to C, 6 \to F, 5 \to E, 4 \to B, 1 \to D$
$\therefore$ Maximum Matching : $A \to 3, C \to 6, F \to 5, E \to 4, B \to 1, D \to 2$
This is complete matching.

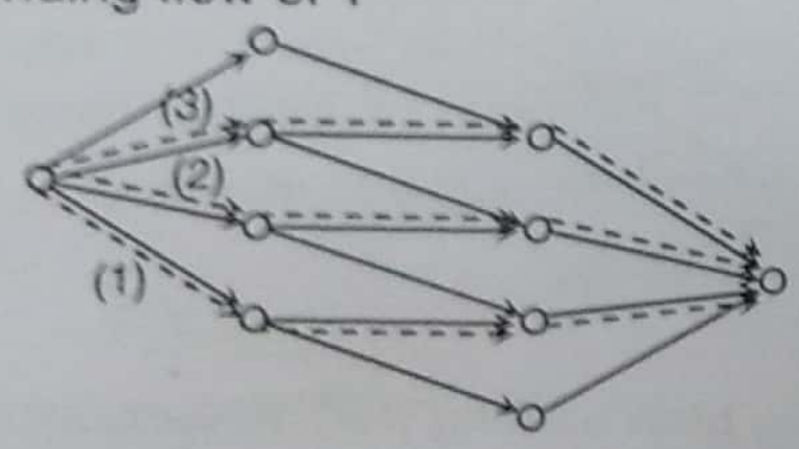
Flow Networks and Maximum Bipartite Matching :



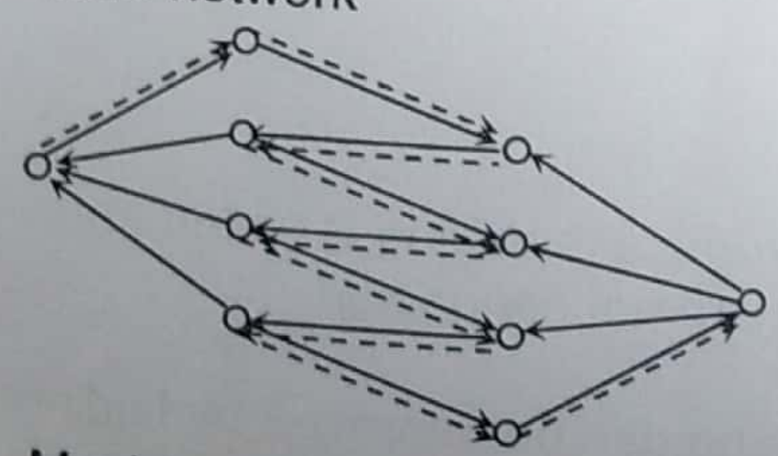
Assuming s and t with edges (assuming all edges have wt. 1)

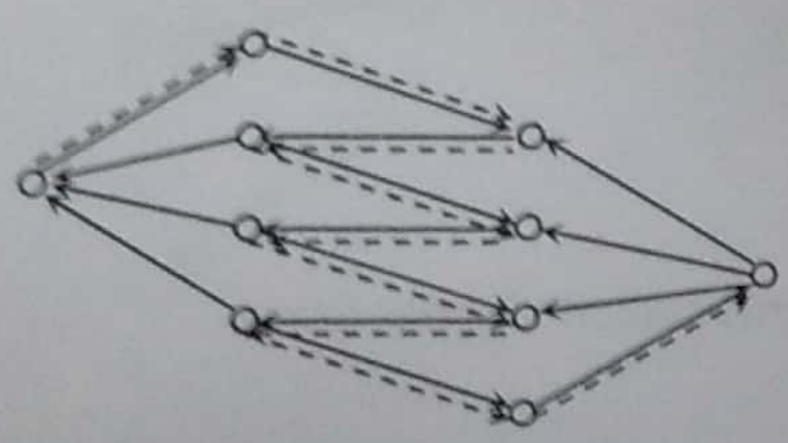


Sending flow of 1

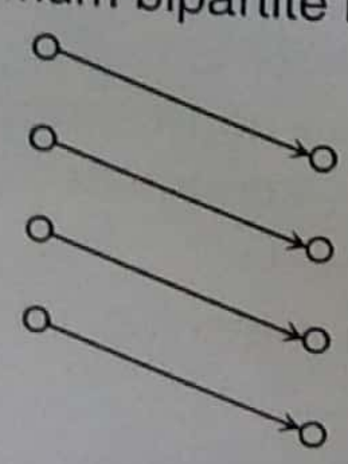


Residual network



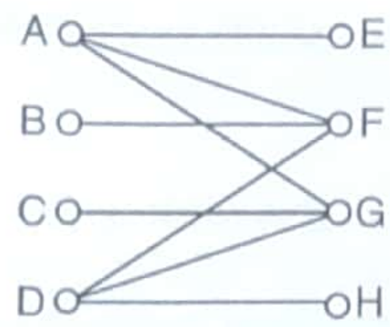


Maximum bipartite matching

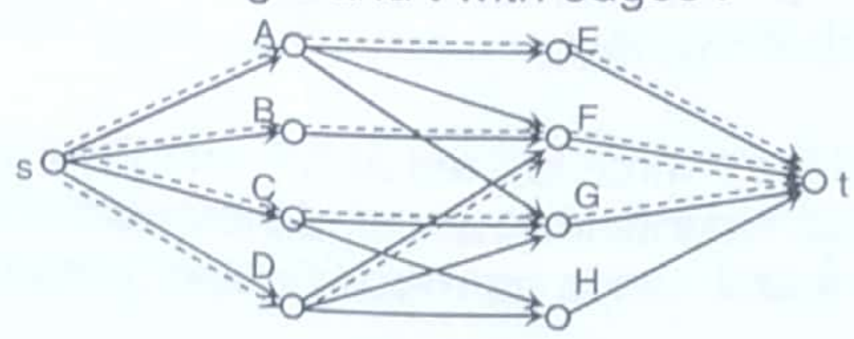


Scanned by CamScanner

Example 1 :

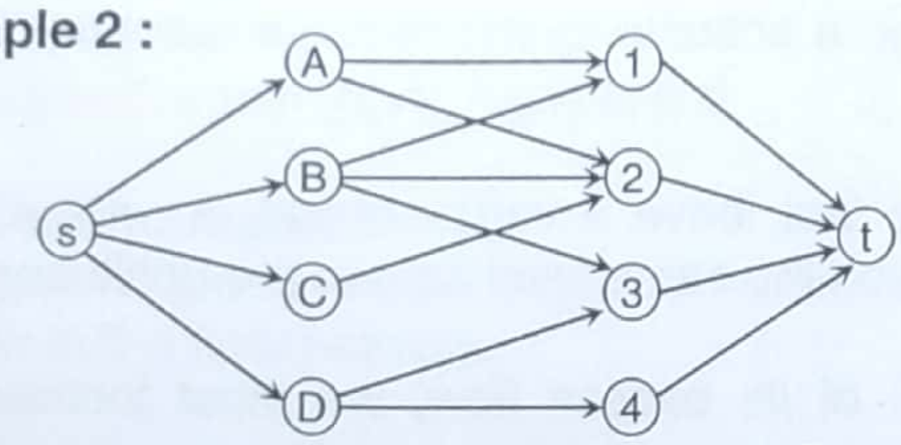


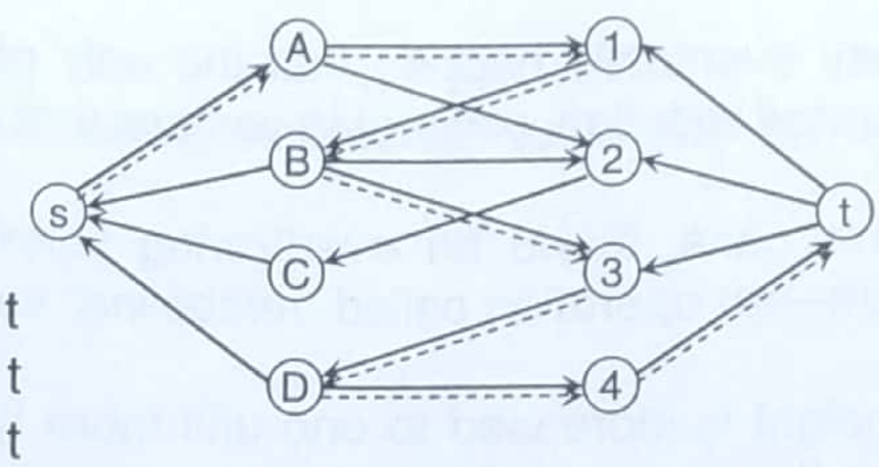
Adding s and t with edges :



1. s – A – E – t
2. s – B – F – t
3. s – C – G – t
4. s – D – H – t

Example 2 :





s – B – 1 – t
s – C – 2 – t
s – D – 3 – t

Left over :

s → A → 1 → B → 3 → D → 4 → t

Matching : A → 1
B → 3 add C → 2
D → 4

PUSH RELABEL ALGORITHM

The intuition behind the push-relabel method is best understood in terms of fluid flows: we consider a flow network G = (V, E) to be a system of interconnected pipes of given capacities.

Applying this analogy to the Ford-Fulkerson method, each augmenting path in the network gives rise to an additional stream of fluid, with no branch points, flowing from the source to the sink.

The Ford-Fulkerson method iteratively adds more streams of flow until no more can be added.

The generic push-relabel algorithm has a rather different intuition. As before, directed edges correspond to pipes. Vertices, which are pipe junctions, have two additional properties.

First, to accommodate excess flow, each vertex has an outflow pipe leading to an arbitrarily large reservoir that can accumulate fluid.

Second, each vertex, its reservoir, and all its pipe connections are on a platform whose height increases as the algorithm progresses.

Vertex heights determine how flow is pushed: we only push flow downhill, that is, from a higher vertex to a lower vertex.

Scanned by CamScanner

The flow from a lower vertex to a higher vertex may be positive, but operations that push flow only push it downhill.

The height of the source is fixed at |V|, and the height of the sink is fixed at 0.
All other vertex heights start at 0 and increase with time.
The algorithm first sends as much flow as possible downhill from the source toward the sink.

The amount it sends is exactly enough to fill each outgoing pipe from the source to capacity; that is, it sends the capacity of the cut (s, V – s).

When flow first enters an intermediate vertex, it collects in the vertex's reservoir. From there, it is eventually pushed downhill.

It may eventually happen that the only pipes that leave a vertex u and are not already saturated with flow connect to vertices that are on the same level as u or are uphill from u.

In this case, to rid an overflowing vertex u of its excess flow, we must increase its height—an operation called "relabeling" vertex u.

Its height is increased to one unit more than the height of the lowest of its neighbors to which it has an unsaturated pipe.

After a vertex is relabeled, therefore, there is at least one outgoing pipe through which more flow can be pushed.

Eventually, all the flow that can possibly get through to the sink has arrived there.
No more can arrive, because the pipes obey the capacity constraints; the amount of flow across any cut is still limited by the capacity of the cut.

The Push Operation :
The basic operation PUSH(u, v) can be applied if u is an overflowing vertex, $c_f(u, v) > 0$, and $h(u) = h(v) + 1$.

The pseudocode below updates the preflow f in an implied network G = (V, E).
The excess flow stored at a vertex u is maintained as the attribute e[u], and the height of u is maintained as the attribute h[u].

The expression $d_f(u, v)$ is a temporary variable that stores the amount of flow that can be pushed from u to v.

function PUSH(u, v)
1) //Applies when: u is overflowing, $c_f(u, v) > 0$, and $h[u] = h[v] + 1$.
2) //Action: Push $d_f(u, v) = \min(e[u], c_f(u, v))$ units of flow from u to v.
3) $d_f(u, v) \leftarrow \min(e[u], c_f(u, v))$
4) $f[u, v] \leftarrow f[u, v] + d_f(u, v)$
5) $f[v, u] \leftarrow -f[u, v]$
6) $e[u] \leftarrow e[u] - d_f(u, v)$
7) $e[v] \leftarrow e[v] + d_f(u, v)$

Scanned by CamScanner

The Relabel Operation :
The basic operation RELABEL(u) applies if u is overflowing and if $h[u] \leq h[v]$ for all edges $(u, v) \in E_f$.

In other words, we can relabel an overflowing vertex u if for every vertex v for which there is residual capacity from u to v; flow cannot be pushed from u to v because v is not downhill from u.

RELABEL(u)
1) // Applies when: u is overflowing and for all $v \in V$ such that $(u, v) \in E f$, we have $h[u] \leq h[v]$.
2) // Action: Increase the height of u.
3) $h[u] \leftarrow 1 + \min \{h[v] : (u, v) \in E f\}$

The Generic Algorithm :
The generic push-relabel algorithm uses the following subroutine to create an initial preflow in the flow network.

Initialize-Preflow(G, s)
1) for each vertex $u \in V[G]$ do
   {
2) $h[u] \leftarrow 0$
3) $e[u] \leftarrow 0$
   }
4) for each edge $(u, v) \in E[G]$ do
   {
5) $f[u, v] \leftarrow 0$
6) $f[v, u] \leftarrow 0$
   }
7) $h[s] \leftarrow |V[G]|$
8) for each vertex $u \in Adj[s]$ do
   {
9) $f[s, u] \leftarrow c(s, u)$
10) $f[u, s] \leftarrow -c(s, u)$
11) $e[u] \leftarrow c(s, u)$
12) $e[s] \leftarrow e[s] - c(s, u)$
   }

Initialization followed by a sequence of push and relabel operations, executed in no particular order, yields the GENERIC-PUSH-RELABEL algorithm:

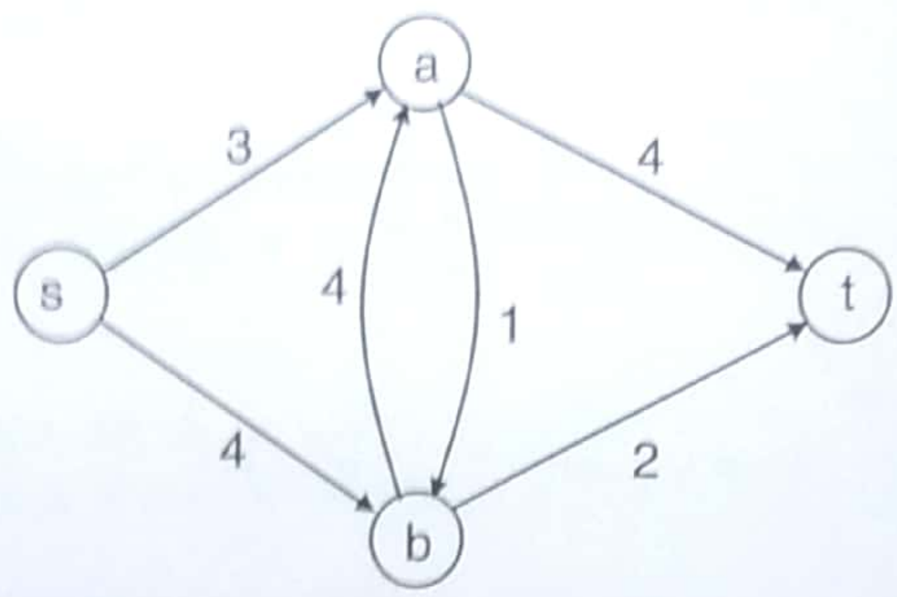
Generic-Push-Relabel(G)
1) INITIALIZE-PREFLOW(G, s)
2) while there exists an applicable push or relabel operation do
3) select an applicable push or relabel operation and perform it

Analysis of the Push-Relabel Method :
Generic Push-Relabel Algorithms runs in $O(V^2E)$ time.

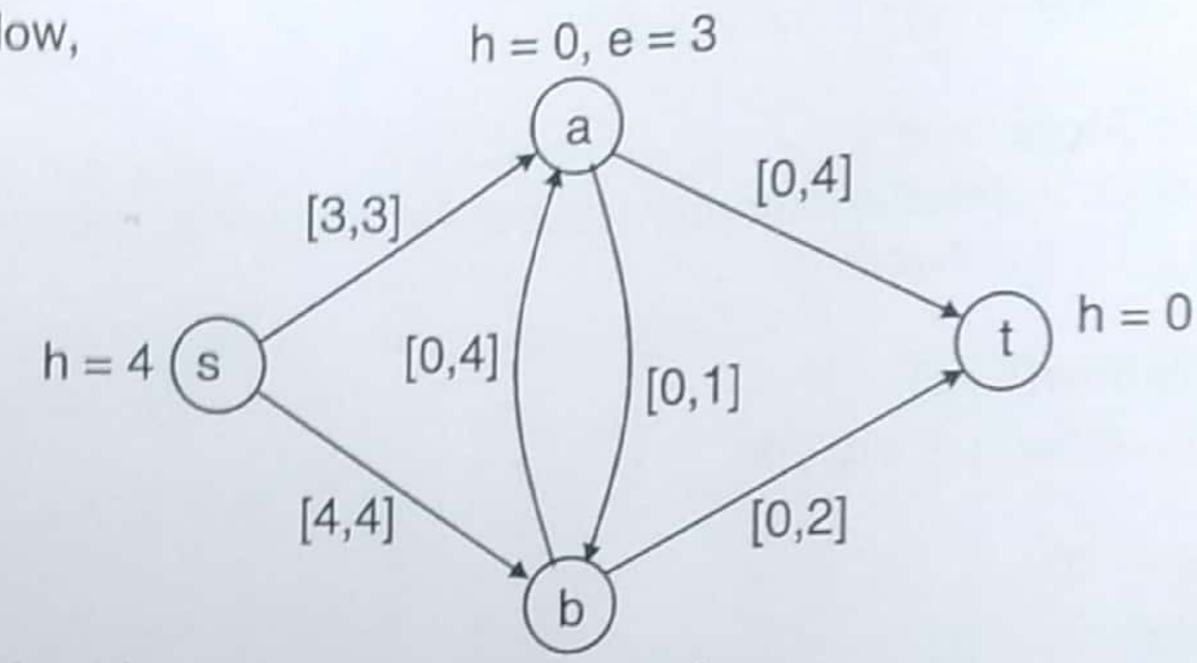
Scanned by CamScanner

Solved Examples

Q.1



Ans.: Initializing Pre-flow,



Edges available from b, h = 0, e = 4

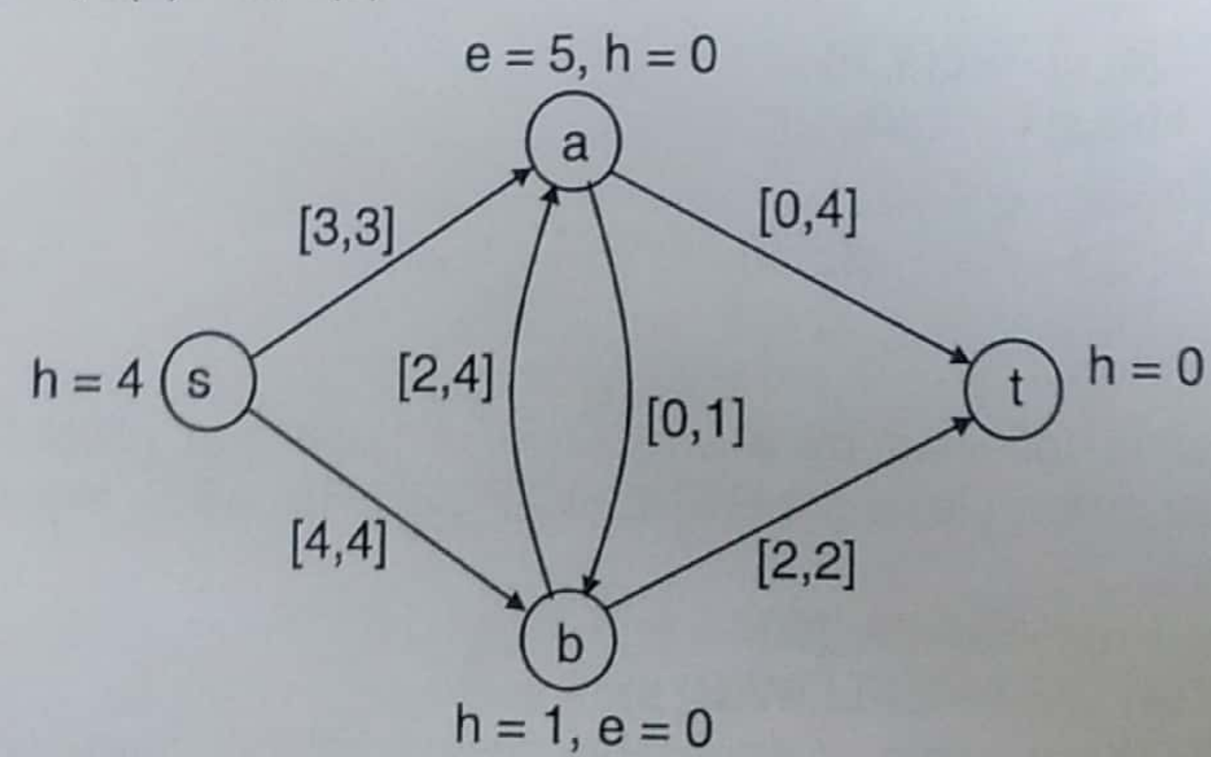
$s \to b$ (reverse flow)

$b \to a$ $\because$ c(b,a) = 4

$b \to t$ $\because$ c(b,t) = 2

Relabel b, h(b) = min{h(s)+1, h(a)+1, h(t)+1}
= min {5,1,1}
= 1

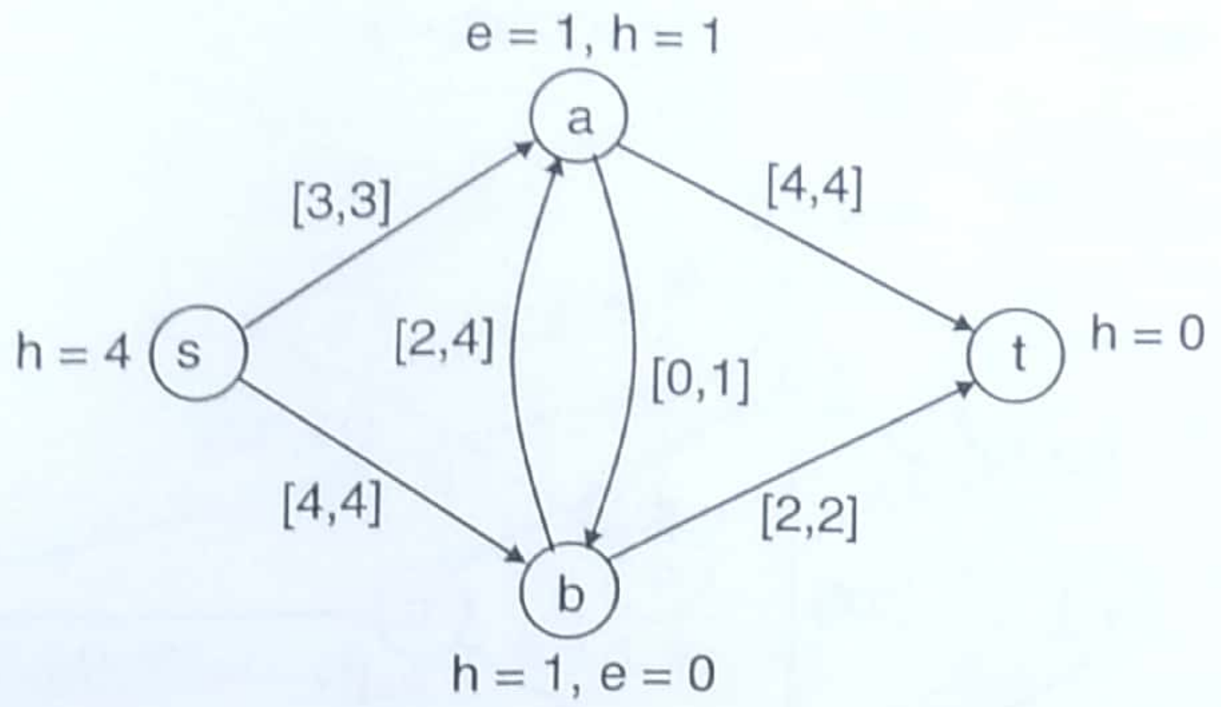
Push on bt : $P_{bt} = 2$, e(b) = 2
$P_{ba} = 2$, e(a) = 5, e(b) = 0



Relabel a,
h(a) = min{h(s) + 1, h(b)+1, h(t)+1}
= min{5, 2, 1}
= 1

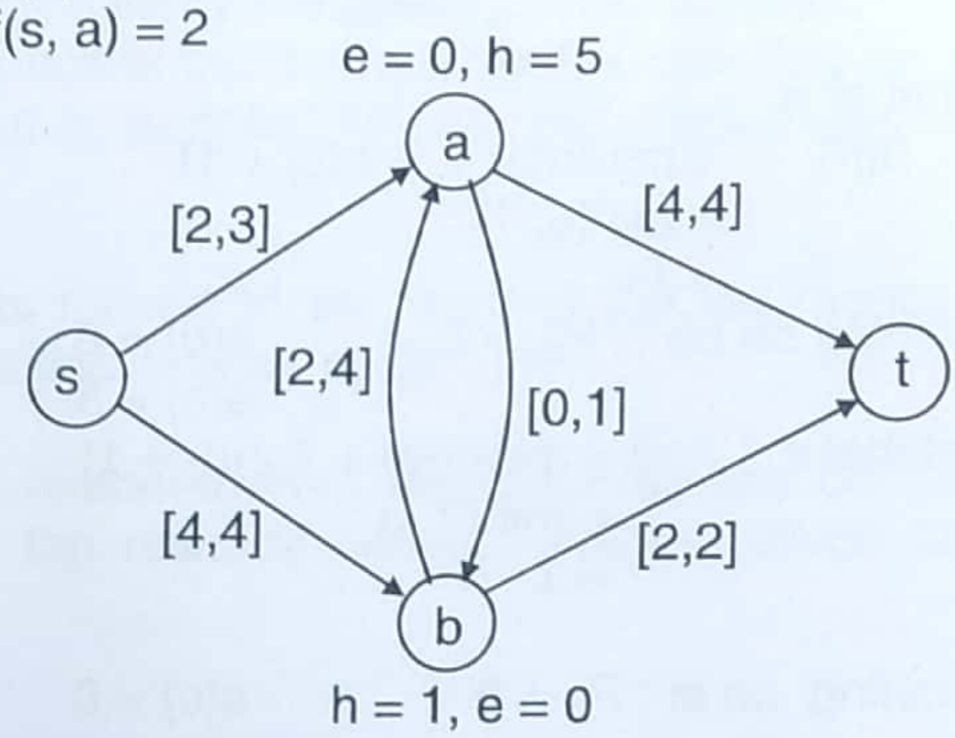
Push on at, $P_{at} = 4$ e(a) = 1

Scanned by CamScanner



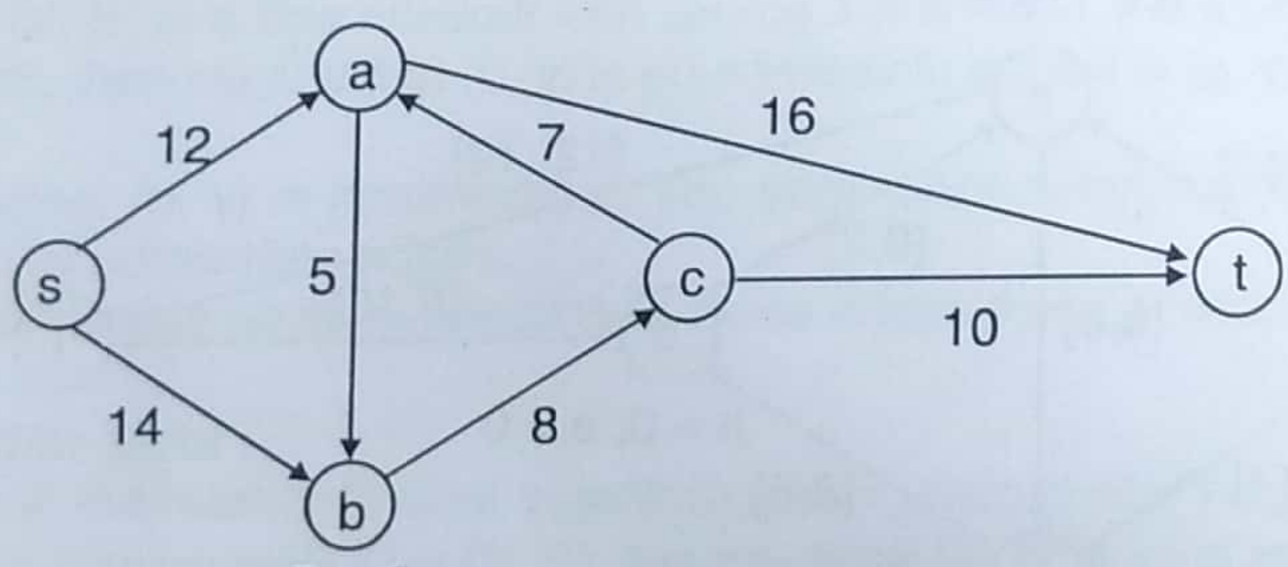
Relabel a, h(a) = min{h(s) + 1} pushing back to source
= 5

Push on as : $P_{as} = 1$ e(a) = 0
f(s, a) = 2

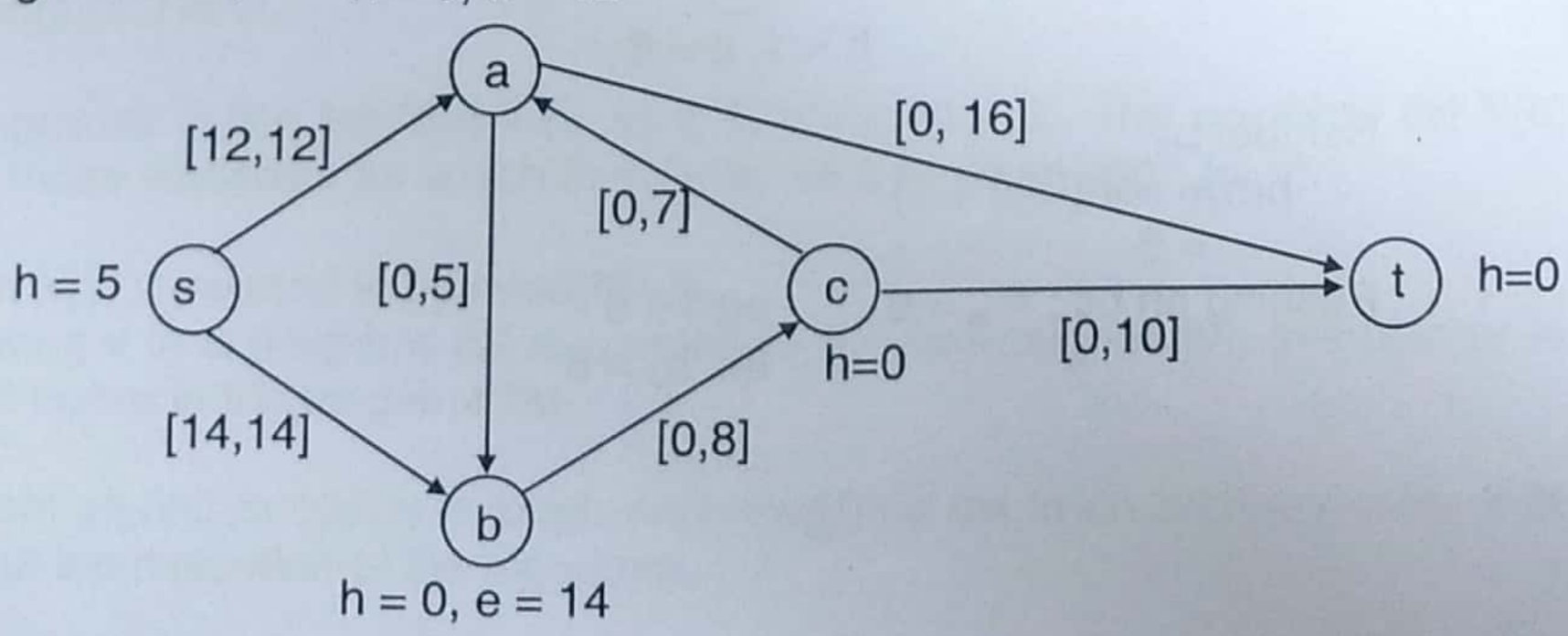


∴ Maximum Flow = 6.

Q.2



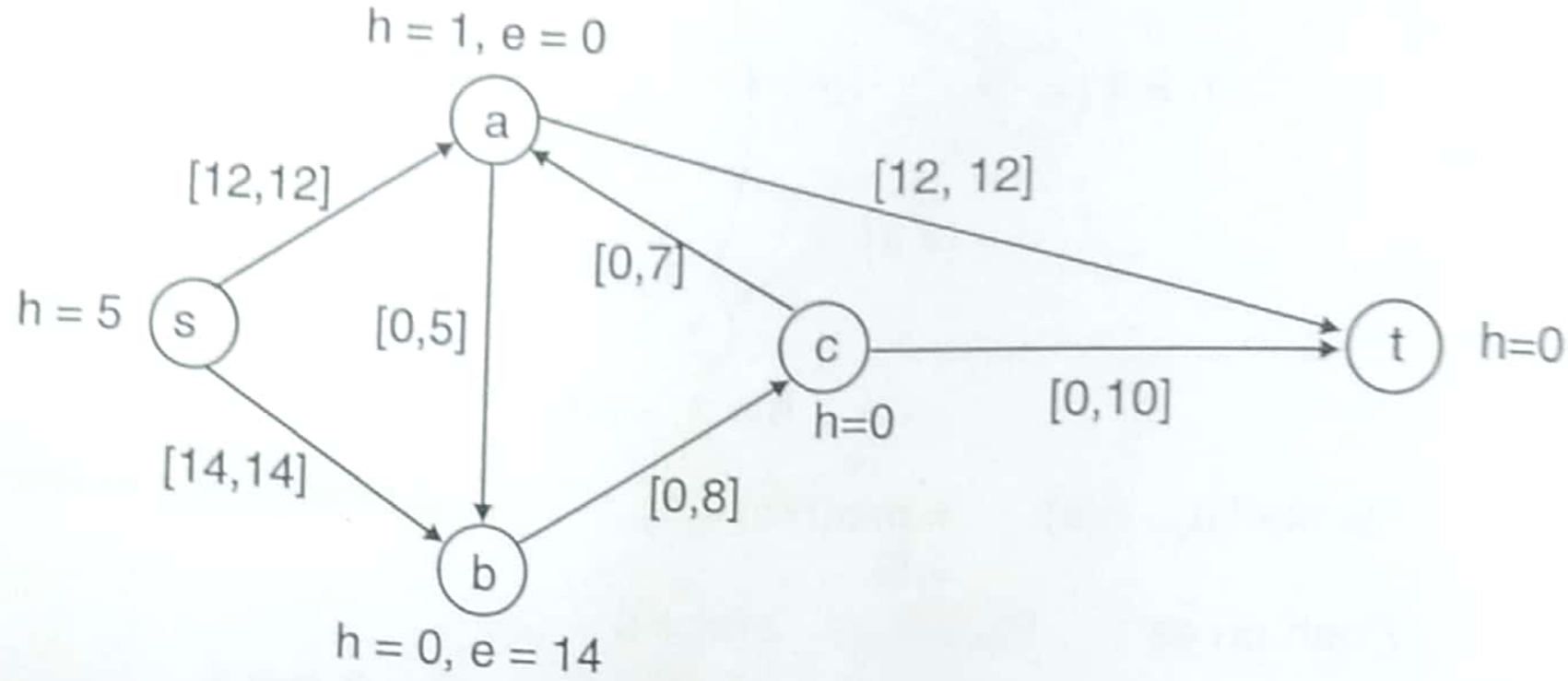
Ans.: Initializing Pre-flow, h = 0, e = 12



Scanned by CamScanner

Relabel a, $h(a) = \min\{h(s)+1, h(b)+1, h(t)+1\}$
$= \min\{6, 1, 1\}$
$= 1$

Push on at : $P_{at} = 12$ $e(a) = 0$

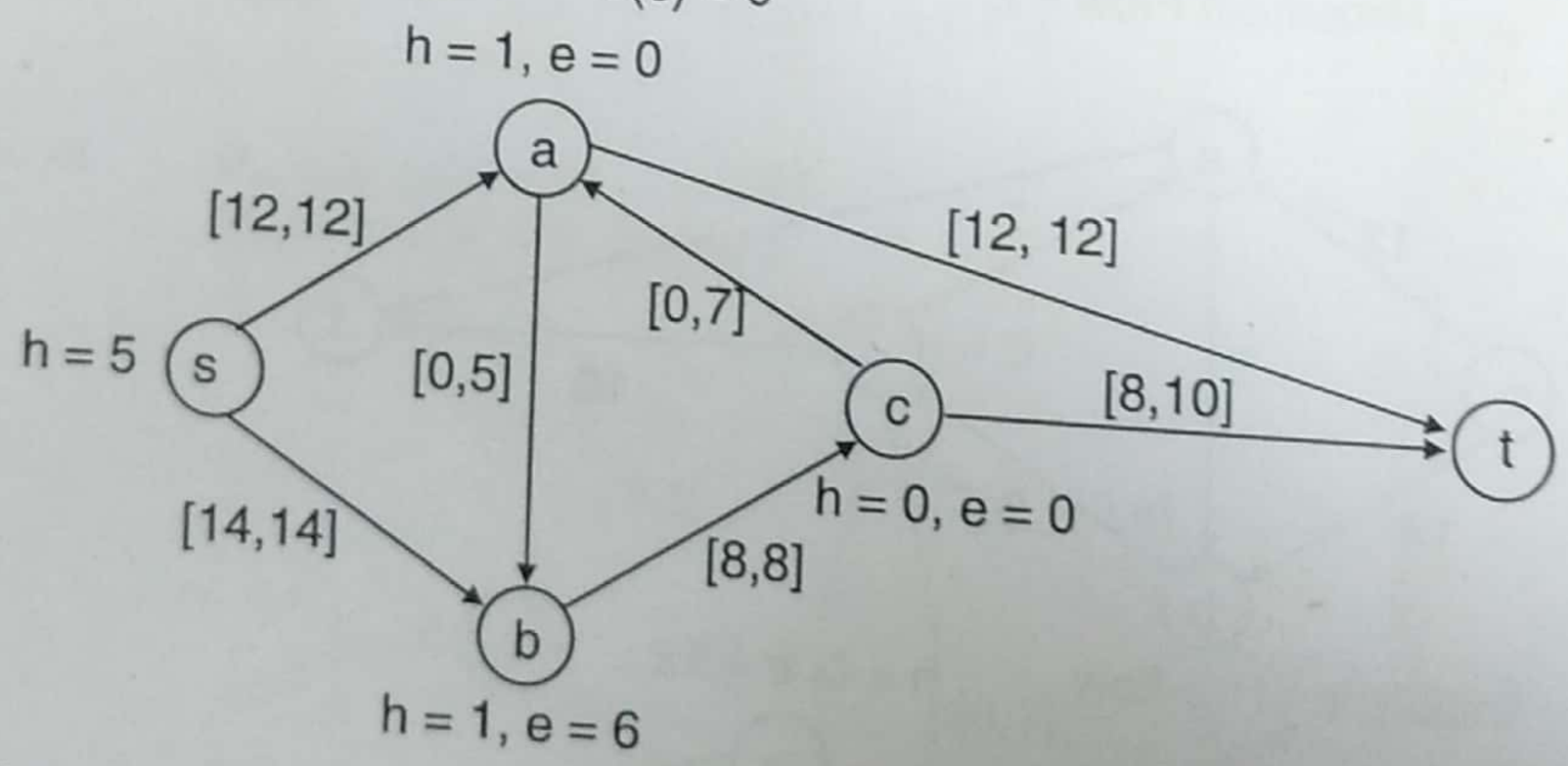


Relabel b,
$h(b) = \min\{h(s) + 1, h(c) + 1\}$
$= \min\{6, 1\}$
$= 1$

Pushing on bc : $P_{bc} = 8$ $e(b) = 6$
$e(c) = 8$

Relabel c, $h(c) = \min\{h(a) + 1, h(t) + 1\}$
$= \min\{2,1\}$
$= 1$

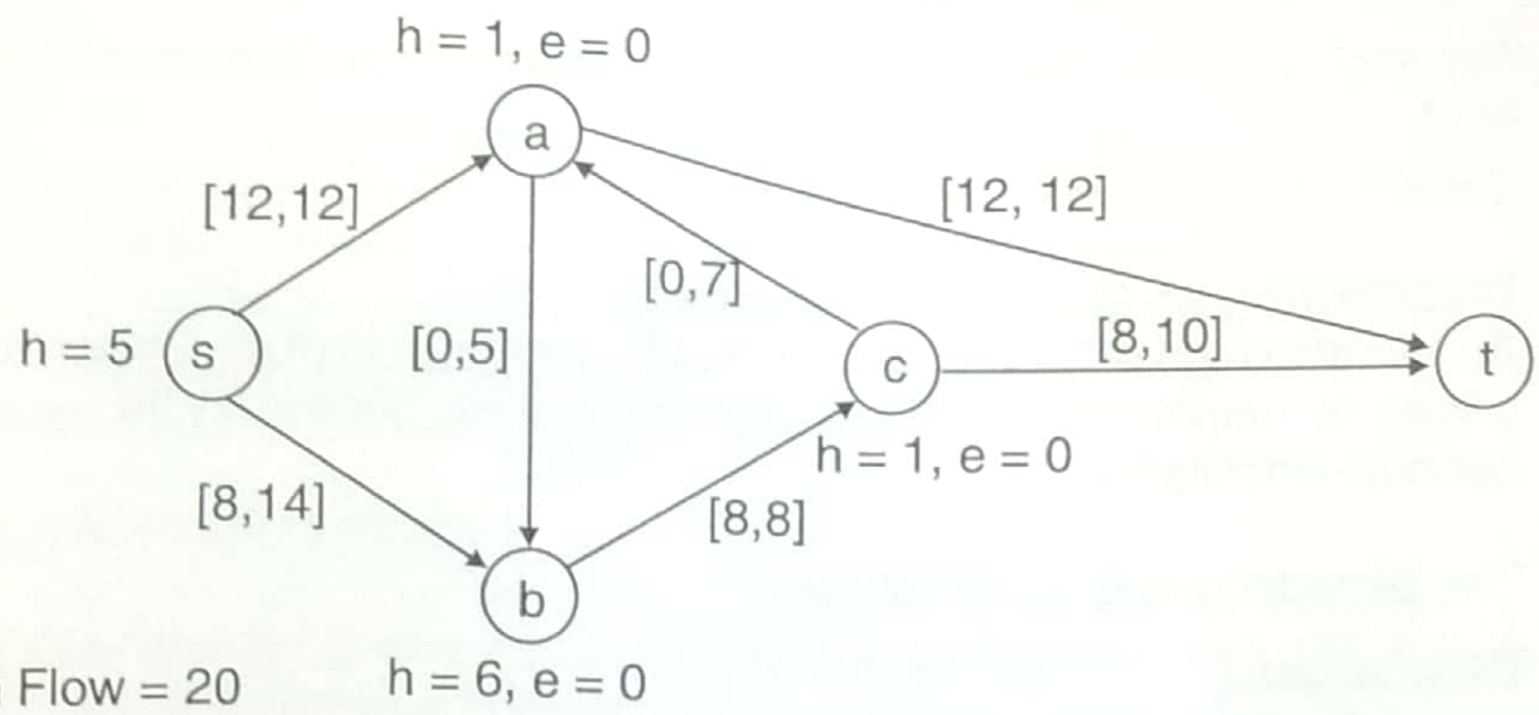
Pushing on ct : $P_{ct} = 8$ $e(c) = 0$



Relabel b_1
$h(b) = \min\{h(s) + 1\}$
$= 6$

Pushing on bs : $P_{bs} = 6$ $e(b) = 0$
$f(s, b) = 8$

Scanned by CamScanner



∴ Maximum Flow = 20

RELABEL TO FRONT ALGORITHM

The relabel-to-front algorithm maintains a list of the vertices in the network.
Beginning at the front, the algorithm scans the list, repeatedly selecting an overflowing vertex u and then "discharging" it, that is, performing push and relabel operations until u no longer has a positive excess.

Whenever a vertex is relabeled, it is moved to the front of the list (hence the name "relabel-to-front") and the algorithm begins its scan again.

The correctness and analysis of the relabel-to-front algorithm depend on the notion of "admissible" edges: those edges in the residual network through which flow can be pushed.

Admissible Edges and Networks :
If G = (V, E) is a flow network with source s and sink t, f is a pre-flow in G, and h is a height function, then we say that (u, v) is an admissible edge if c_f (u, v) > 0 and h(u) = h(v) + 1.

Otherwise, (u, v) is inadmissible. The admissible network is $G_{f,h}$ = (V, $E_{f,h}$), where $E_{f,h}$ is the set of admissible edges.
The admissible network consists of those edges through which flow can be pushed.

Neighbor Lists :
Edges in the relabel-to-front algorithm are organized into "neighbor lists."
Given a flow network G = (V, E), the neighbor list N[u] for a vertex u ∈ V is a singly linked list of the neighbors of u in G.

Thus, vertex v appears in the list N[u] if (u, v) ∈ E or (v, u) ∈ E. The neighbor list N[u] contains exactly those vertices v for which there may be a residual edge (u, v).

The first vertex in N[u] is pointed to by head[N[u]].
The vertex following v in a neighbor list is pointed to by next-neighbor[v]; this pointer is NIL if v is the last vertex in the neighbor list.

The relabel-to-front algorithm cycles through each neighbor list in an arbitrary order that is fixed throughout the execution of the algorithm.

Scanned by CamScanner

For each vertex u, the field current[u] points to the vertex currently under consideration in N[u].
Initially, current[u] is set to head[N[u]].

Discharging an Overflowing Vertex :
An overflowing vertex u is discharged by pushing all of its excess flow through admissible edges to neighboring vertices, relabeling u as necessary to cause edges leaving u to become admissible.

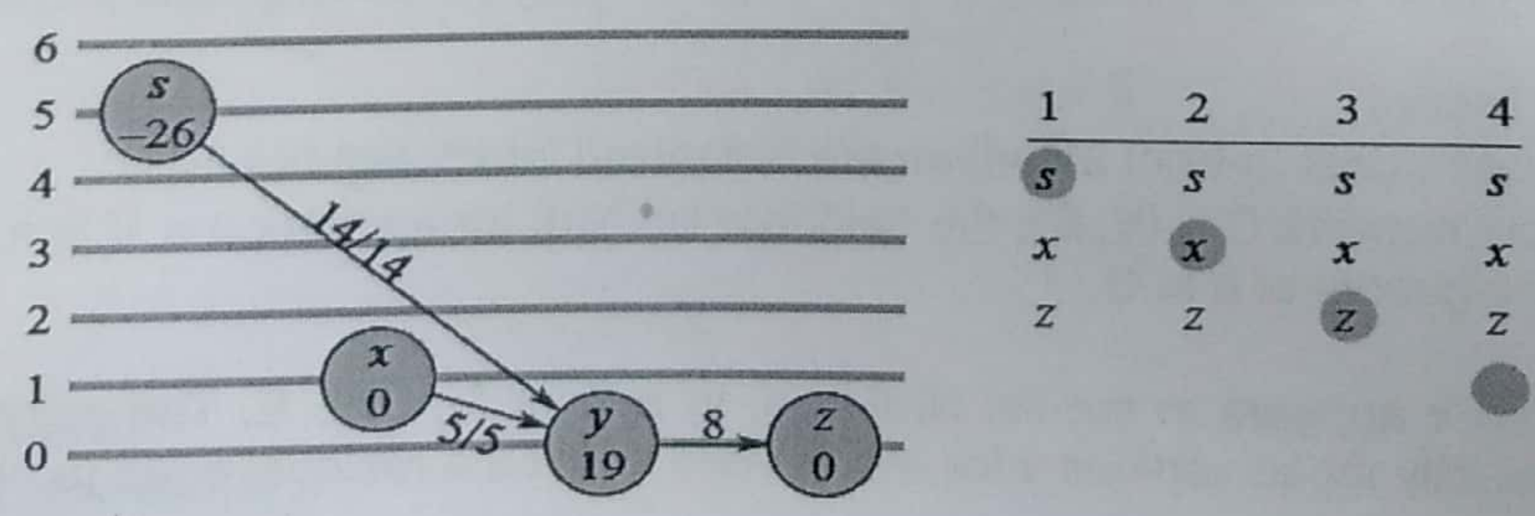
The pseudocode goes as follows :

```
Discharge(u)
1)  while e[u] > 0 do
    {
2)      v ← current[u]
3)      if v = NIL then
        {
4)          RELABEL(u)
5)          current[u] ← head[N[u]]
        }
6)      else if c_f (u, v) > 0 and h[u] = h[v] + 1 then
7)          PUSH(u, v)
8)      else current[u] ← next-neighbor[v]
    }
```

Algorithm steps through several iterations of the while loop of lines 1–8, which executes as long as vertex u has positive excess. Each iteration performs exactly one of three actions, depending on the current vertex v in the neighbor list N[u].

1. If v is NIL, then we have run off the end of N[u]. Line 4 relabels vertex u, and then line 5 resets the current neighbor of u to be the first one in N[u].
2. If v is non-NIL and (u, v) is an admissible edge (determined by the test in line 6), then line 7 pushes some (or possibly all) of u's excess to vertex v.
3. If v is non-NIL but (u, v) is inadmissible, then line 8 advances current[u] one position further in the neighbor list N[u].

(a)



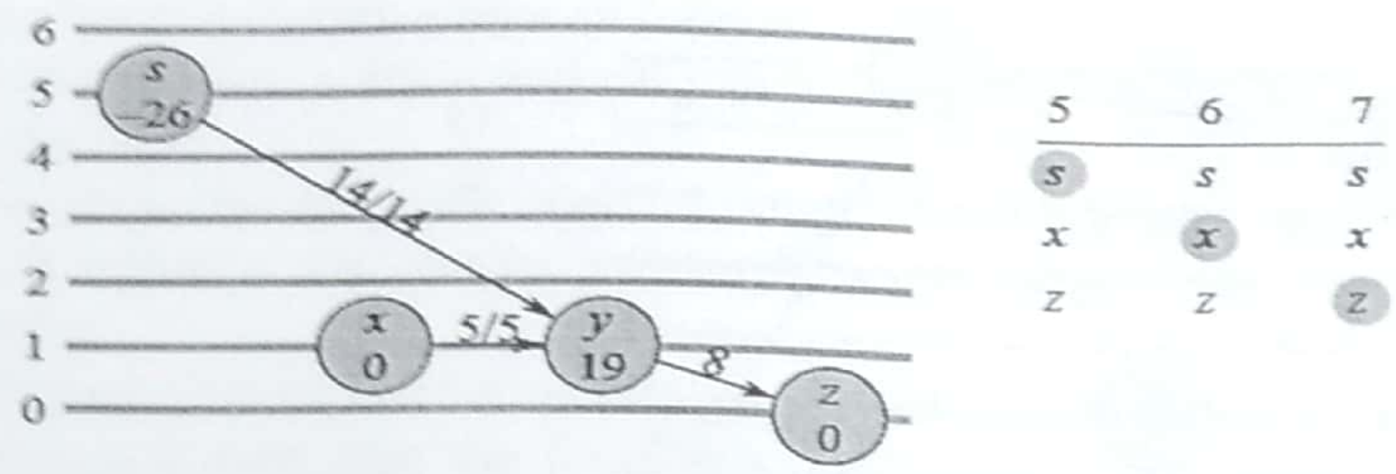
(a) Initially, there are 19 units of excess to push from y, and current[y] = s.

Iterations 1, 2, and 3 just advance current[y], since there are no admissible edges leaving y.

In iteration 4, current[y] = NIL (shown by the shading being below the neighbor list), and so y is relabeled and current[y] is reset to the head of the neighbor list.

Scanned by CamScanner

(b)

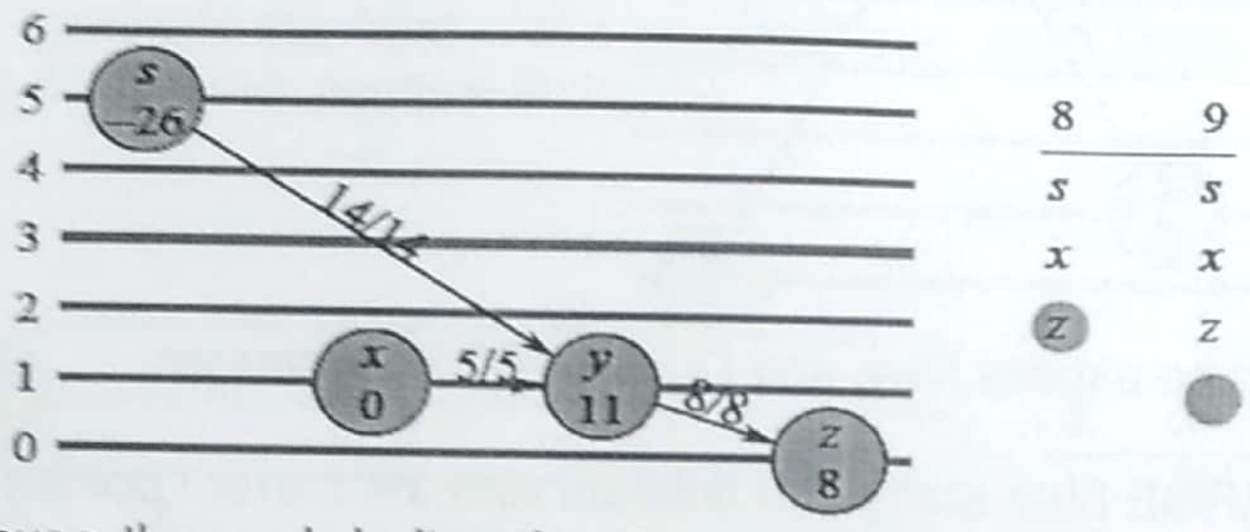


(b) After relabeling, vertex y has height 1.

In iterations 5 and 6, edges (y, s) and (y, x) are found to be inadmissible, but 8 units of excess flow are pushed from y to z in iteration 7.

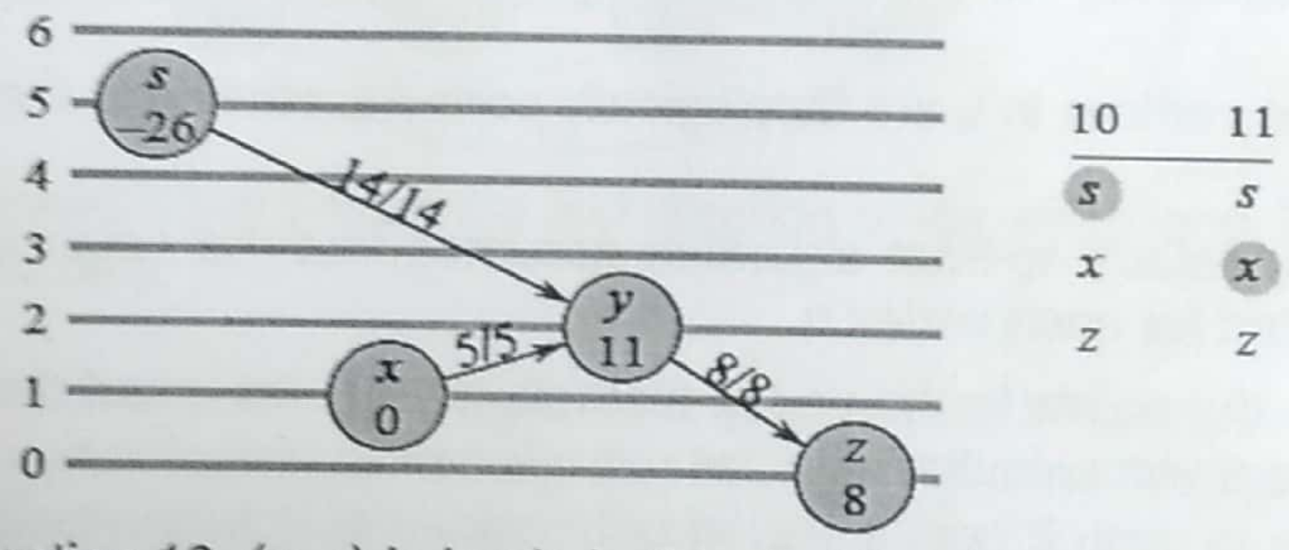
Because of the push, current[y] is not advanced in this iteration.

(c)



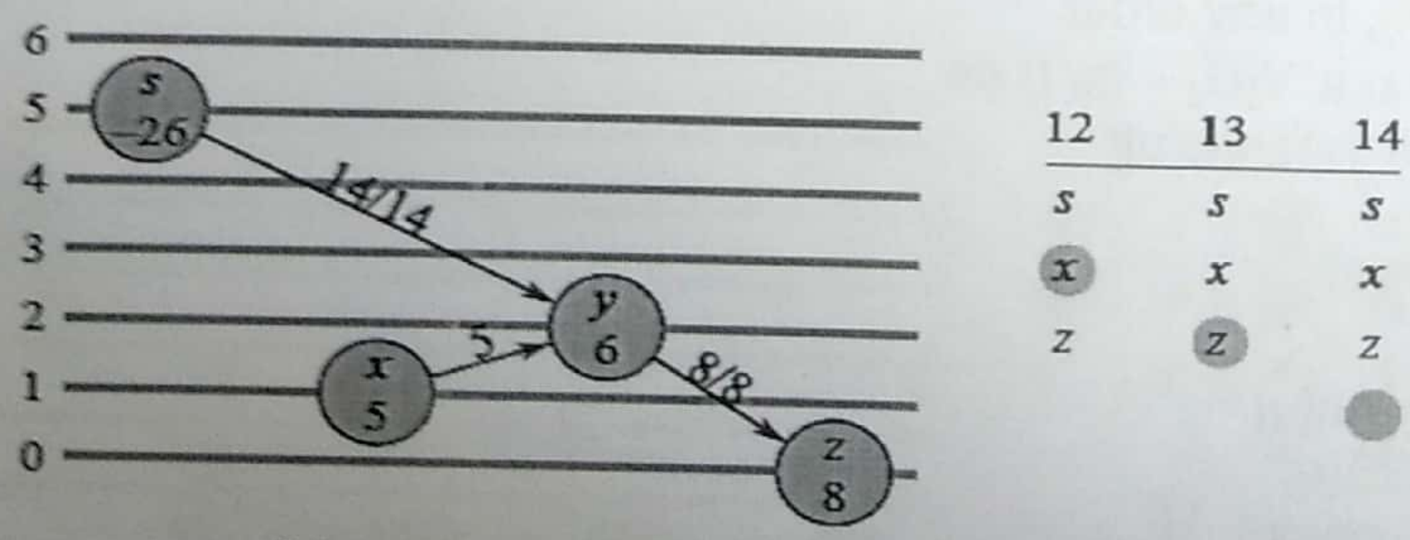
(c) Because the push in iteration 7 saturated edge (y, z), it is found inadmissible in iteration 8. In iteration 9, current[y] = NIL, and so vertex y is again relabeled and current[y] is reset.

(d)



(d) In iteration 10, (y, s) is inadmissible, but 5 units of excess flow are pushed from y to x in iteration 11.

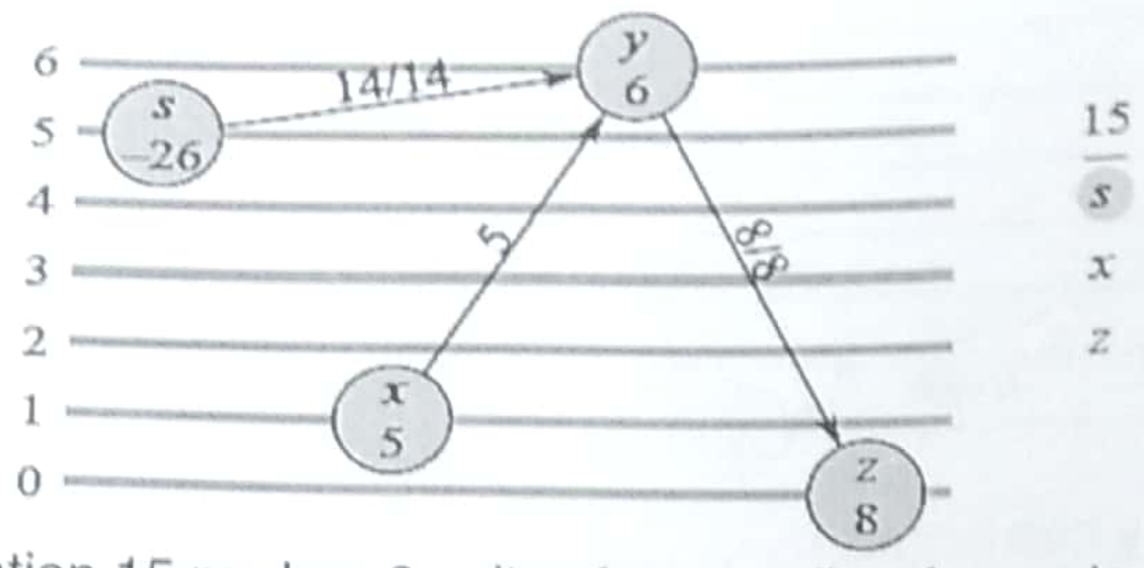
(e)



(e) Because current[y] was not advanced in iteration 11, iteration 12 finds (y, x) to be inadmissible. Iteration 13 finds (y, z) inadmissible and iteration 14 relabels vertex y and resets current[y].

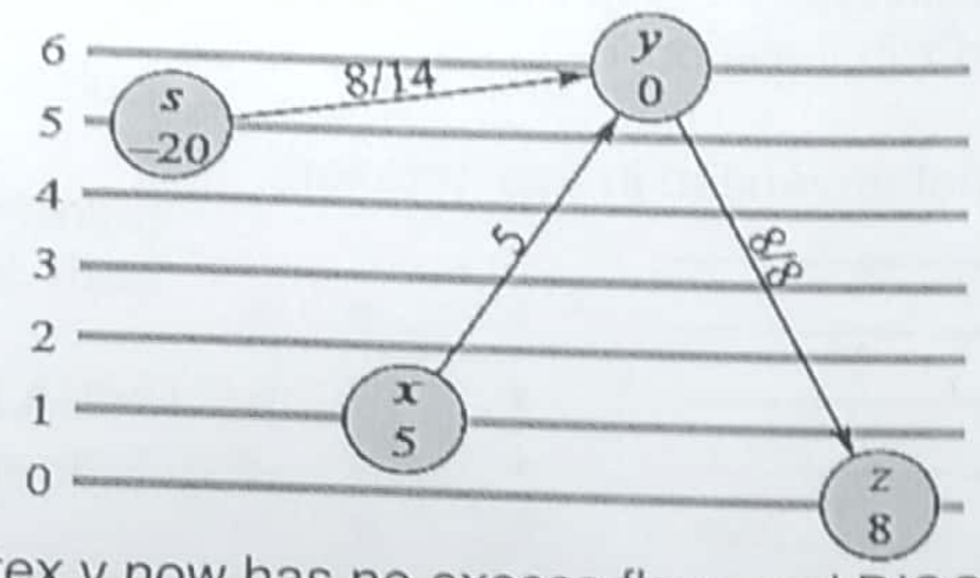
Scanned by CamScanner

(f)



(f) Iteration 15 pushes 6 units of excess flow from y to s.

(g)



(g) Vertex y now has no excess flow, and DISCHARGE terminates.

In this example, DISCHARGE both starts and finishes with the current pointer at the head of the neighbor list, but in general this need not be the case.

The relabel-to-front algorithm :

In the relabel-to-front algorithm, we maintain a linked list L consisting of all vertices in V – {s, t}.

A key property is that the vertices in L are topologically sorted according to the admissible network.

The pseudocode for the relabel-to-front algorithm assumes that the neighbor lists N[u] have already been created for each vertex u.

It also assumes that next[u] points to the vertex that follows u in list L and that, as usual, next[u] = NIL if u is the last vertex in the list.

Relabel-To-Front(G, s, t)

```
INITIALIZE-PREFLOW(G, s)
L ← V[G] – {s, t}, in any order
for each vertex u ∈ V[G] – {s, t} do
        current[u] ← head[N[u]]
u ← head[L]
while u ≠ NIL do
    {
        old-height ← h[u]
        DISCHARGE(u)
        if h[u] > old-height
            then move u to the front of list L
        u ← next[u]
    }
```

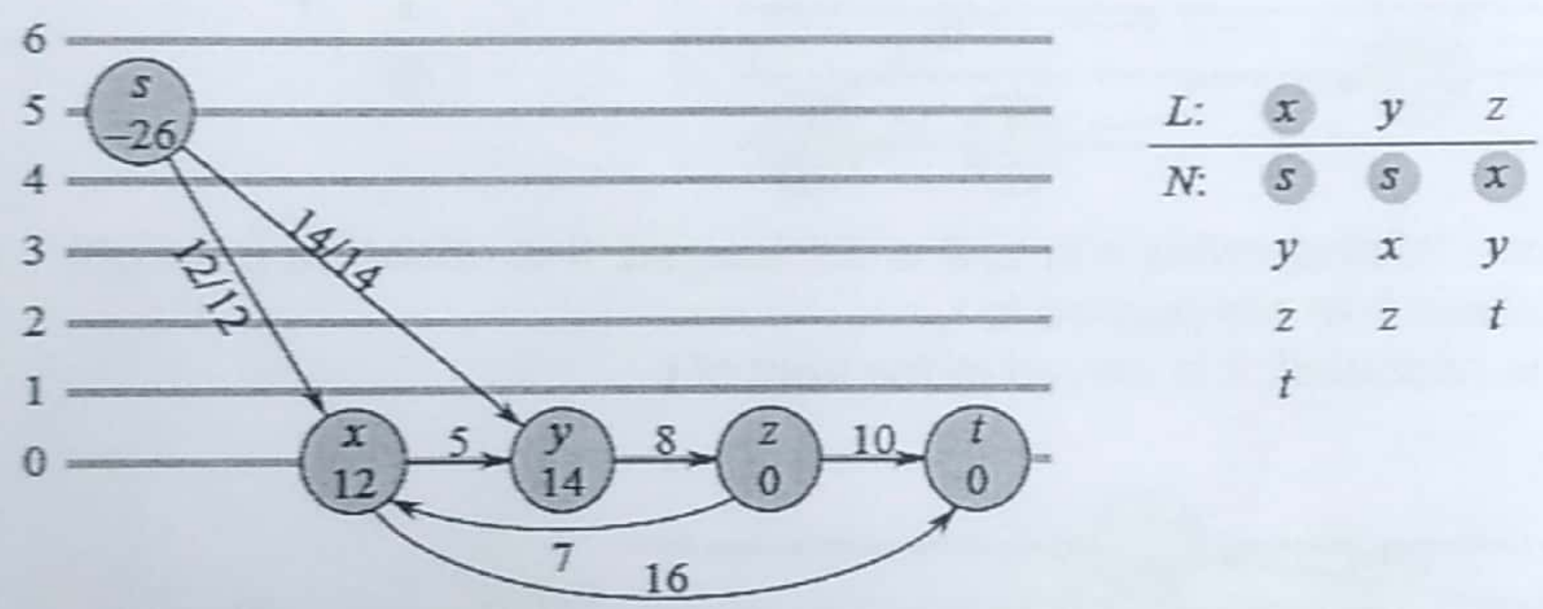
Scanned by CamScanner

The relabel-to-front algorithm works as follows :

- Line 1 initializes the preflow and heights to the same values as in the generic push-relabel algorithm.
- Line 2 initializes the list L to contain all potentially overflowing vertices, in any order.
- Lines 3–4 initialize the current pointer of each vertex u to the first vertex in u's neighbor list.
  The while loop of lines 6–11 runs through the list L, discharging vertices.
- Line 5 makes it start with the first vertex in the list. Each time through the loop, a vertex u is discharged in line 8.
  If u was relabeled by the DISCHARGE procedure, line 10 moves it to the front of list L. This determination is made by saving u's height in the variable old-height before the discharge operation (line 7) and comparing this saved height to u's height afterward (line 9).
- Line 11 makes the next iteration of the while loop use the vertex following u in list L. If u was moved to the front of the list, the vertex used in the next iteration is the one following u in its new position in the list.

Problem :

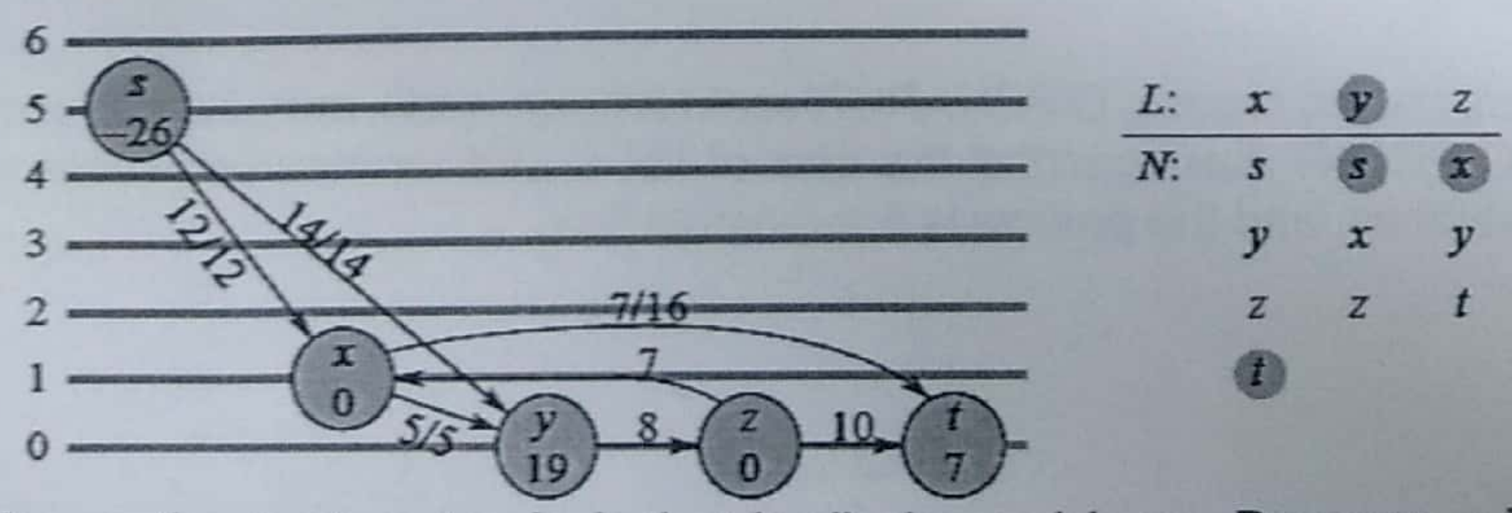
(a)



(a) A flow network just before the first iteration of the while loop. Initially, 26 units of flow leave source s.

On the right is shown the initial list L = <x, y, z>, where initially u = x.
Under each vertex in list L is its neighbor list, with the current neighbor shaded.
Vertex x is discharged. It is relabeled to height 1 then 5 units of excess flow are pushed to y, and the 7 remaining units of excess are pushed to the sink t.
Because x is relabeled, it is moved to the head of L, which in this case does not change the structure of L.

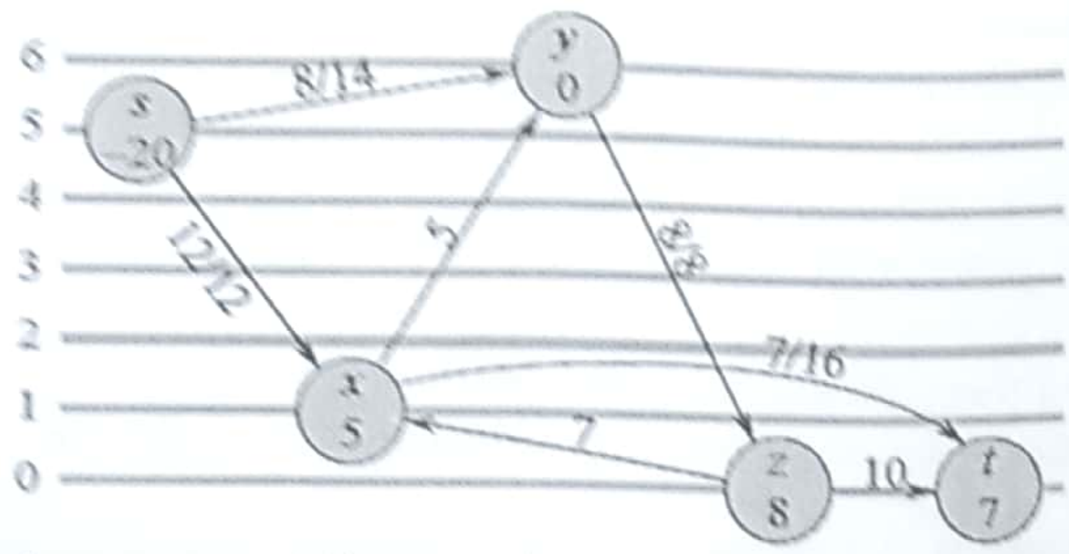
(b)



(b) After x, the next vertex in L that is discharged is y. Because y is relabeled, it is moved to the head of L.

Scanned by CamScanner

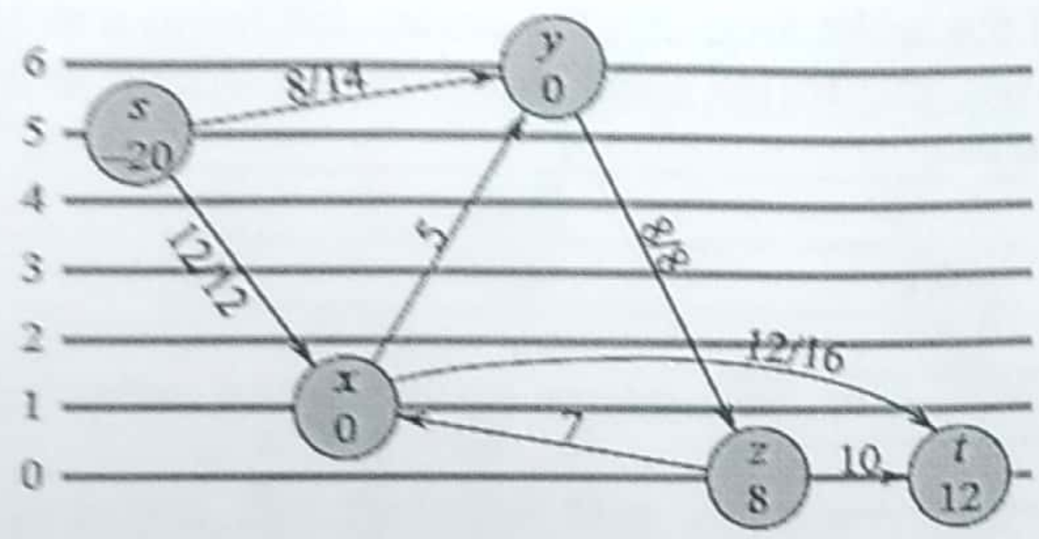
(c)



L:	y	x	z
N:	s	s	x
	x	y	y
	z	z	t
		t	

(c) Vertex x now follows y in L, and so it is again discharged, pushing all 5 units of excess flow to t.
Because vertex x is not relabeled in this discharge operation, it remains in place in list L.

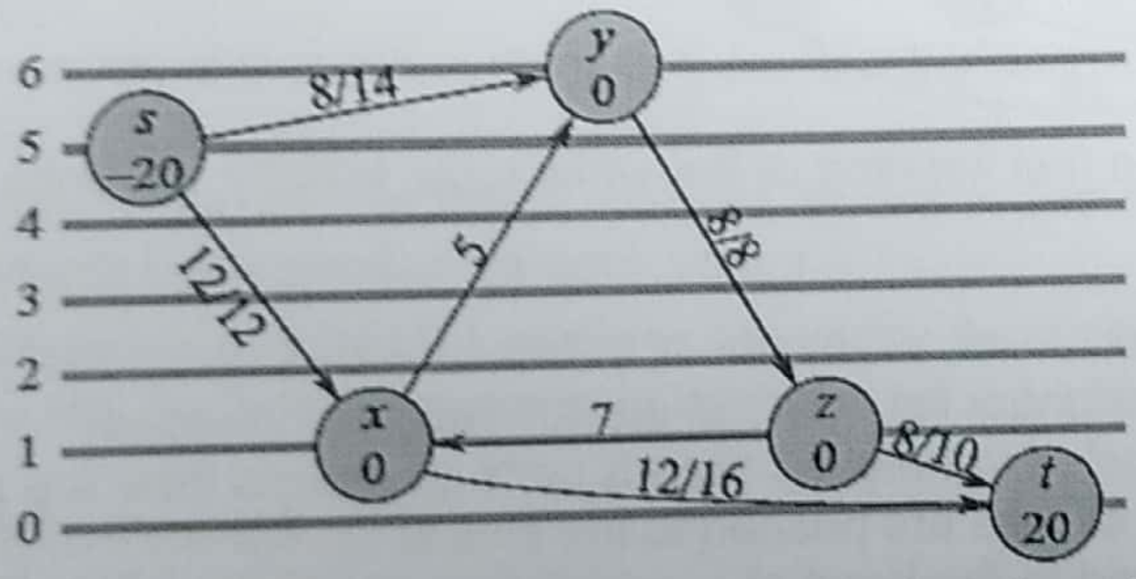
(d)



L:	y	x	z
N:	s	s	x
	x	y	y
	z	z	t
		t	

(d) Since vertex z follows vertex x in L, it is discharged. It is relabeled to height 1 and all 8 units of excess flow are pushed to t.
Because z is relabeled, it is moved to the front of L.

(e)



L:	z	y	x
N:	x	s	s
	y	x	y
	t	z	z
			t

(e) Vertex y now follows vertex z in L and is therefore discharged. But because y has no excess, DISCHARGE immediately returns, and y remains in place in L. Vertex x is then discharged.

Because it, too, has no excess, DISCHARGE again returns, and x remains in place in L. RELABEL-TO-FRONT has reached the end of list L and terminates. There are no overflowing vertices, and the preflow is a maximum flow.

❑❑❑❑❑

Scanned by CamScanner

Module

5

Computational Geometry

SYLLABUS Weightage : 18 Marks

Computational Geometry

Line Segment properties, Determining whether any pair of segment intersects, finding the convex hull, Finding the closest pair of points.

Help Line : For any query WhatsApp to 704 501 85 39 to get it Solved.

Scanned by CamScanner

LINE SEGMENT PROPERTIES

Convex Combinations :

A convex combination of 2 distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some α in the range of $0 \le \alpha \le 1$.
We have $x_3 = \alpha x_1 + (1 - \alpha) x_2$ and $y_3 = \alpha x_2 + (1 - \alpha) y_2$
We can also write this as $p_3 = \alpha p_1 + (1 - \alpha) p_2$

Thus p_3 is any point that is on the line passing through p_1, p_2 and is on or between p_1 and p_2 on the line.

Line Segment :

Given two distinct points p_1 and p_2, the line segment p_1p_2 (over line) is the set of convex combinations of p_1 and p_2.
We call p_1 as left endpoint and p_2 as right endpoint of segment P_1P_2.

Directed Segment :

Directed segment p_1p_2 (over right arrows) is referred as the vector p_2 with p_1 as the origin (0,0).

Convex Combination :

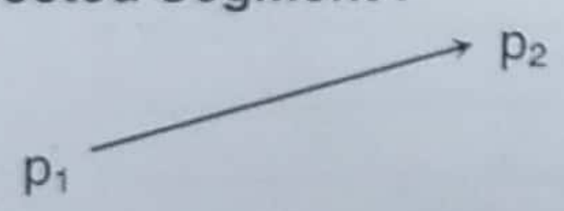
p_1 (2, 2) ———————————— p_2(5, 2)

$p_3 = \alpha p_1 + (1 - \alpha) p_2$

Line Segment :

p_1p_2 (Over Line)

Directed Segment :

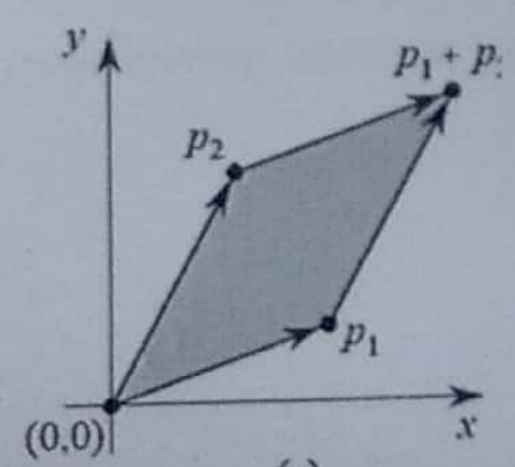


p_1p_2 (Over Right Arrow)

Computational geometry requires answers to the questions about the properties of line segments :

1. Given 2 directed segments p_0p_1 and p_0p_2, is p_0p_1 is clockwise from p_0p_2, with respect to their common end point p_0?
2. Given 2 line segments p_0p_1 and p_1p_2, if we traverse p_0p_1 and then p_1p_2, do we make a left turn at point p_1?
3. Do line segments p_1p_2 and p_3p_4 intersect?

1.

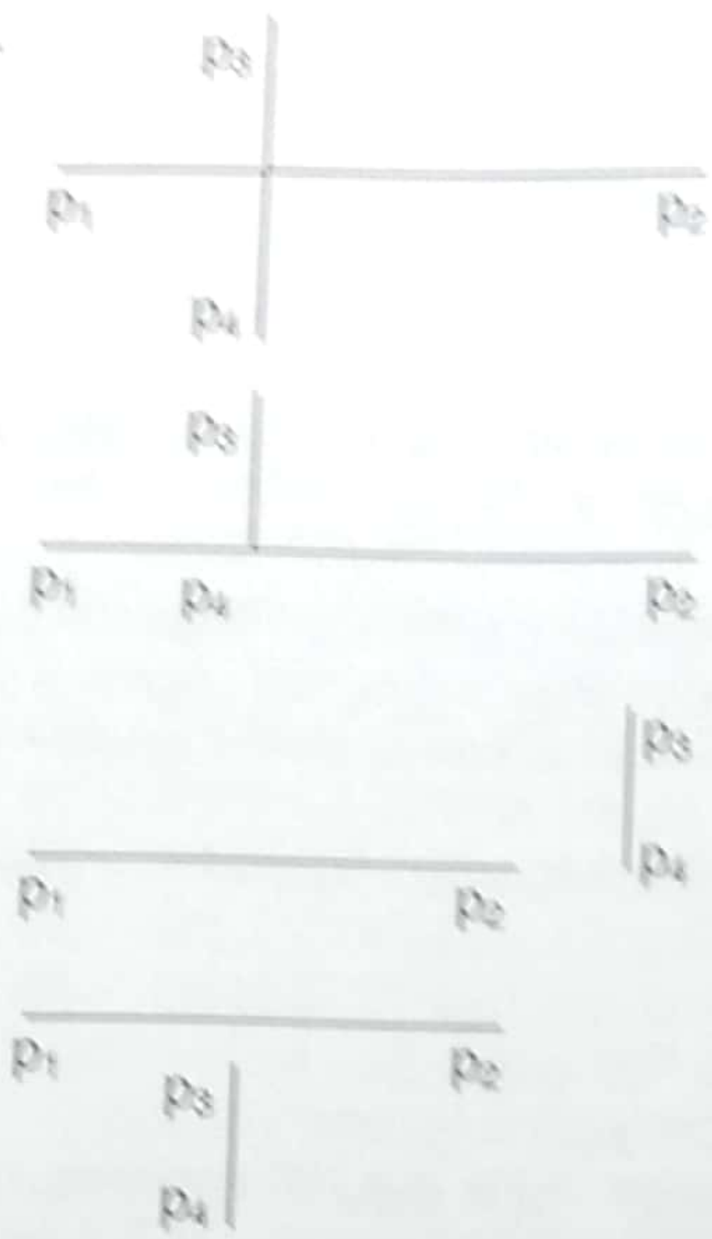


(a)

Scanned by CamScanner

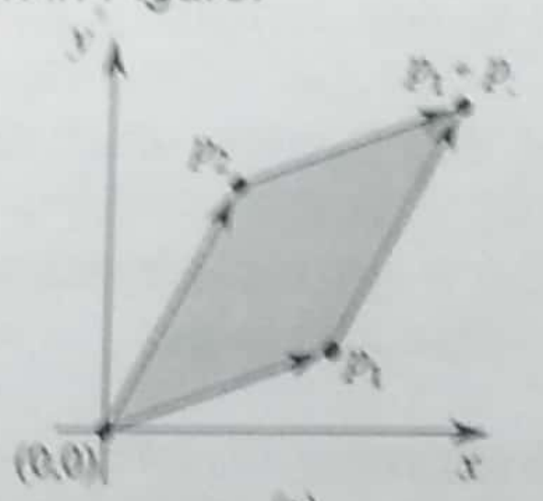
2.

3.



Cross products :

Consider vectors p_1 and p_2, shown in Figure.



(a)

The cross product $p_1 \times p_2$ can be interpreted as the signed area of the parallelogram formed by the points (0, 0), p_1, p_2, and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$.

An equivalent, but more useful, definition gives the cross product as the determinant of a matrix:

$$p_1 \times p_2 = \det\begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix}$$

$$= x_1 y_2 - x_2 y_1$$

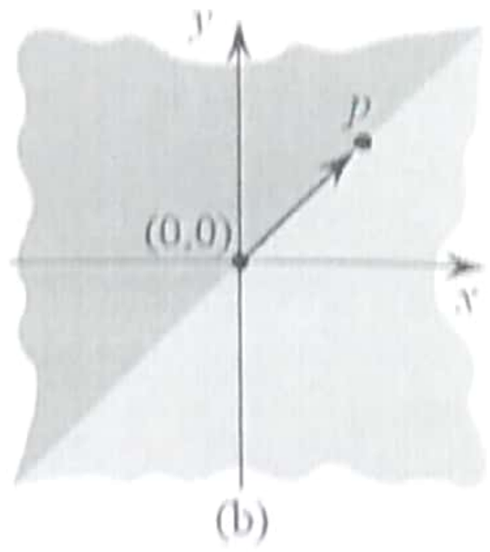
$$= - p_2 \times p_1$$

If $p_1 \times p_2 > 0$, then p_1 is clockwise from p_2

If $p_1 \times p_2 < 0$, then p_1 is anti-clockwise from p_2

Scanned by CamScanner

Figure shows the clockwise and counterclockwise regions relative to a vector p.



(b)

Light shaded → the vectors clock-wise from p
Dark shaded → the vectors anti-clockwise from p

A boundary condition arises if the cross product is 0; in this case, the vectors are collinear, pointing in either the same or opposite directions.

To determine whether a directed segment p_0p_1 is clockwise from a directed segment p_0p_2. To determine whether a directed segment p_0p_1 is clockwise from a directed segment p_0p_2 with respect to their common endpoint p_0, we simply translate to use p_0 as the origin.

That is, we let $p_1 - p_0$ denote the vector $p_1' = (x_1', y_1')$, where $x_1' = x_1 - x_0$ and $y_1' = y_1 - y_0$, and we define $p_2 - p_0$ similarly.

We then compute the cross product

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

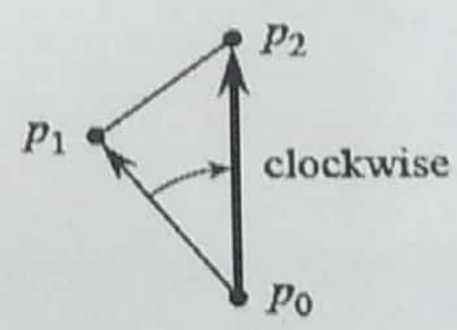
If this cross product is positive, then p_0p_1 is clockwise from p_0p_2 if negative, it is counterclockwise.

To determine whether consecutive segment turn left or right.

To determine whether consecutive line segment p_0p_1 and p_1p_2 turn left or right at point p_1.

In other words we want a method to determine which way $p_0p_1p_2$ turns.

A positive cross product $(p_2 - p_0) \times (p_1 - p_0)$ indicates p_0p_2 is clockwise w.r.t. p_0p_1 and it makes right turn at p_1.



Determining Whether 2 Lines Intersect :

To detect whether 2 lines intersect, we check whether each segment straddles the line containing the other.

A line segment p_1p_2 straddles a line if point p_1 lies on one side of the line and point p_2 lies on the other side.

A boundary case arises if p_1 and p_2 lie directly on the line.

Thus, 2 line segments intersect iff either or both of the following conditions hold true.

1) Each segment straddles the line containing other.
2) An endpoint of one lies on the other segment.

Scanned by CamScanner

This can be implemented using the following procedures :

1. DIRECTION ( ) – computes relative orientation
2. ON-SEGMENT ( ) – determines whether a point to be collinear with a segment lies on that segment.
3. SEGMENT_INTERSECT( ) – returns true if the segments intersect otherwise returns false.

```
Algorithm DIRECTION (pi, pj, pk)
// pi and pj are endpoints of one line segment and pk is one point (endpoint) of the line
segment
1)  return (pk-pi) x (pj-pi)            //cross product  O(1)
```

```
Algorithm ON_SEGMENT(pi, pj, pk)
// pi and pj are endpoints of one line segment and pk is one point (endpoint) of the line
segment
1)  if min(xi,xj) ≤ xk ≤ max(xi,xj) and min(yi,yj) ≤ yk ≤ max(yi,yj) then
2)          return TRUE
3)  else return FALSE
```

```
Algorithm SEGMENTS-INTERSECT(p1, p2, p3, p4)
1)  d1 ← DIRECTION(p3, p4, p1)
2)  d2 ← DIRECTION(p3, p4, p2)
3)  d3 ← DIRECTION(p1, p2, p3)
4)  d4 ← DIRECTION(p1, p2, p4)
5)  if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
          ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0)) then
6)          return TRUE
7)  else if d1 = 0 and ON-SEGMENT(p3, p4, p1) then
8)          return TRUE
9)          else if d2 = 0 and ON-SEGMENT(p3, p4, p2) then
10)             return TRUE
11)             else if d3 = 0 and ON-SEGMENT(p1, p2, p3) then
12)                 return TRUE
13)                 else if d4 = 0 and ON-SEGMENT(p1, p2, p4) then
14)                     return TRUE
15)                 else return FALSE
```

Determining whether any pair of segments intersect :

This section presents an algorithm for determining whether any two line segments in a set of segments intersect.

The algorithm uses a technique known as "sweeping," which is common to many computational-geometry algorithms.

The algorithm runs in O(n log n) time, where n is the number of segments we are given. It determines only whether or not any intersection exists; it does not print all the intersections.

In sweeping, an imaginary vertical sweep line passes through the given set of geometric objects, usually from left to right.

Scanned by CamScanner

The spatial dimension that the sweep line moves across, in this case the x-dimension, is treated as a dimension of time.

Sweeping provides a method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them.

The line-segment-intersection algorithm in this section considers all the line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

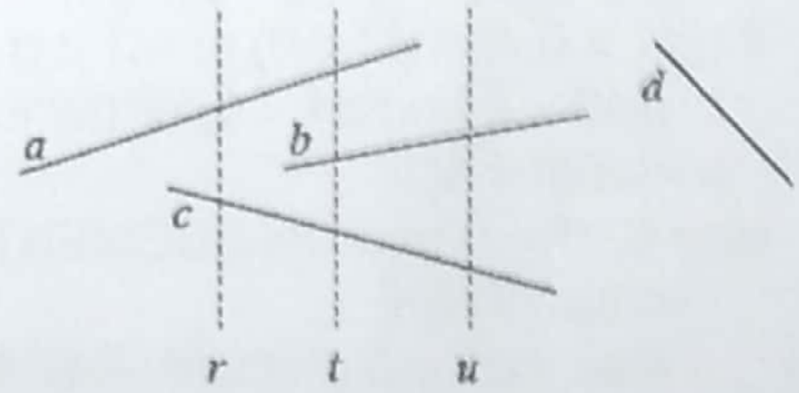
To describe and prove correct our algorithm for determining whether any two of n line segments intersect, we shall make two simplifying assumptions.

First, we assume that no input segment is vertical. Second, we assume that no three input segments intersect at a single point.

Ordering segments :
Since we assume that there are no vertical segments, any input segment that intersects a given vertical sweep line intersects it at a single point.
Thus, we can order the segments that intersect a vertical sweep line according to the y-coordinates of the points of intersection.

To be more precise, consider two segments s_1 and s_2. We say that these segments are comparable at x if the vertical sweep line with x-coordinate x intersects both of them.

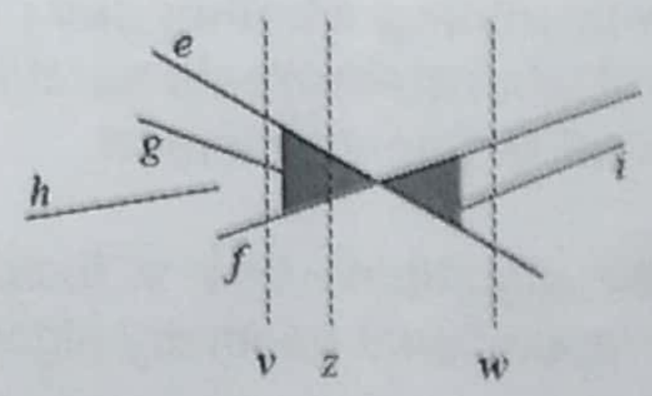
We say that s_1 is above s_2 at x, written $s_1 >_x s_2$, if s_1 and s_2 are comparable at x and the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line.



In Figure, for example, we have the relationships $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$, and $b >_u c$.
Segment d is not comparable with any other segment.
A segment enters the ordering when its left endpoint is encountered by the sweep, and it leaves the ordering when its right endpoint is encountered.

What happens when the sweep line passes through the intersection of two segments?

As Figure shows, their positions in the total order are reversed.
Sweep lines v and w are to the left and right, respectively, of the point of intersection of segments e and f, and we have $e >_v f$ and $f >_w e$.



Note that because we assume that no three segments intersect at the same point, there must be some vertical sweep line x for which intersecting segments e and f are consecutive in the total order $>_x$.

Any sweep line that passes through the shaded region of Figure, such as z, has e and f consecutive in its total order.

Scanned by CamScanner

Moving the Sweep Line

Sweeping algorithms typically manage two sets of data:

1. The sweep-line status gives the relationships among the objects intersected by the sweep line.
2. The event-point schedule is a sequence of x-coordinates, ordered from left to right, that defines the halting positions of the sweep line.

We call each such halting position an event point. Changes to the sweep-line status occur only at event points.

The algorithm determines the event points statically, based solely on simple properties of the input data.

In particular, each segment endpoint is an event point. We sort the segment endpoints by increasing x-coordinate and proceed from left to right. (If two or more endpoints are covertical, i.e., they have the same x-coordinate, we break the tie by putting all the covertical left endpoints before the covertical right endpoints. Within a set of covertical left endpoints, we put those with lower y-coordinates first, and do the same within a set of covertical right endpoints.)

We insert a segment into the sweep-line status when its left endpoint is encountered, and we delete it from the sweep-line status when its right endpoint is encountered. Whenever two segments first become consecutive in the total order, we check whether they intersect.

The sweep-line status is a total order T , for which we require the following operations:

- INSERT(T, s): insert segment s into T
- DELETE(T, s): delete segment s from T
- ABOVE(T, s): return the segment immediately above segment s in T
- BELOW(T, s): return the segment immediately below segment s in T

If there are n segments in the input, we can perform each of the above operations in O(log n) time using red-black trees.

Recall that the red-black-tree operations involve comparing keys. We can replace the key comparisons by comparisons that use cross products to determine the relative ordering of two segments.

Segment-intersection pseudocode

The following algorithm takes as input a set S of n line segments, returning the boolean value TRUE if any pair of segments in S intersects, and FALSE otherwise.

The total order T is implemented by a red-black tree.

ANY-SEGMENTS-INTERSECT(S)

1. $T \leftarrow \emptyset$
2. sort the endpoints of the segments in S from left to right, breaking ties by putting left endpoints before right endpoints and breaking further ties by putting points with lower y-coordinates first
3. for each point p in the sorted list of endpoints do
   {

Scanned by CamScanner

```
4.  if p is the left endpoint of a segment s then
    {
5.      INSERT(T, s)
6.      if (ABOVE(T, s) exists and intersects s)
            or (BELOW(T, s) exists and intersects s) then
7.      return TRUE
    }
8.  if p is the right endpoint of a segment s then
    {
9.      if both ABOVE(T, s) and BELOW(T, s) exist
            and ABOVE(T, s) intersects BELOW(T, s) then
10.     return TRUE
11.     DELETE(T, s)
    }
    }
12. return FALSE
```

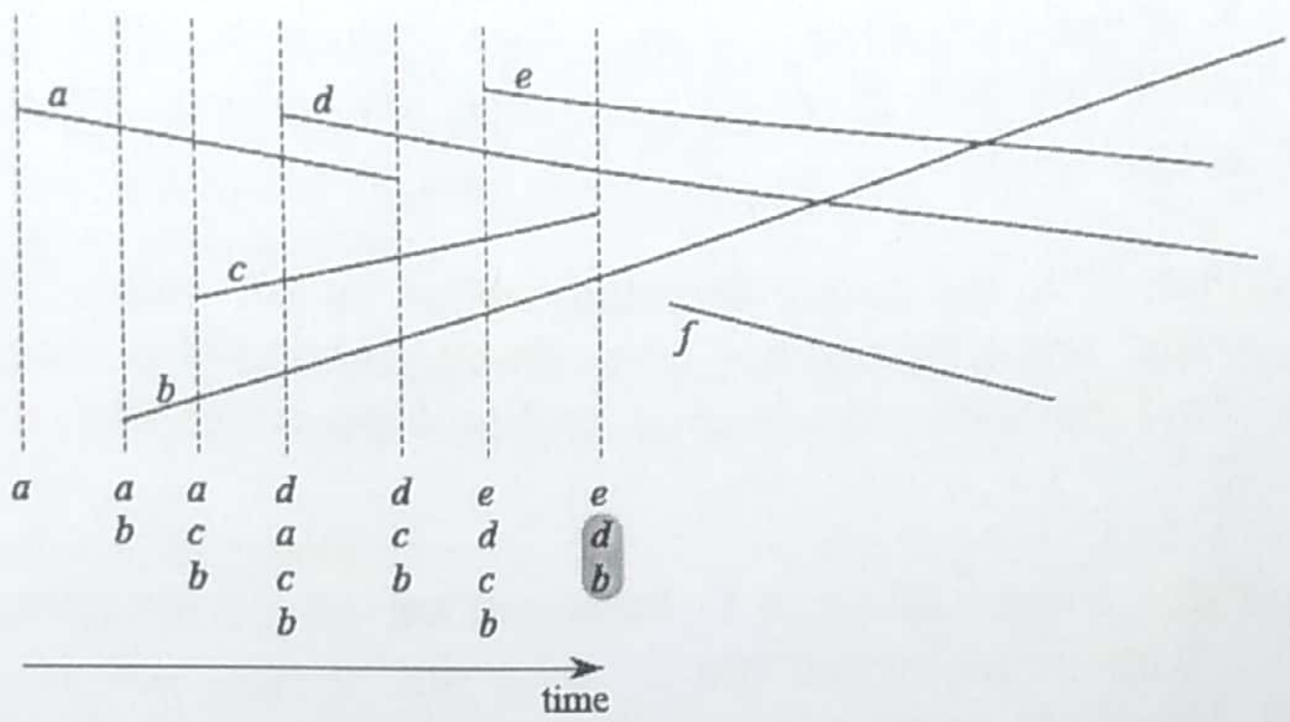


Figure illustrates the execution of the algorithm.

Line 1 initializes the total order to be empty.
Line 2 determines the event-point schedule by sorting the 2n segment endpoints from left to right, breaking ties as described above.

Note that line 2 can be performed by lexicographically sorting the endpoints on (x, e, y), where x and y are the usual coordinates and e = 0 for a left endpoint and e = 1 for a right endpoint.

Each iteration of for loop of lines 3–11 processes one event point p.
If p is the left endpoint of a segment s, line 5 adds s to the total order, and lines 6–7 return TRUE if s intersects either of the segments it is consecutive with in the total order defined by the sweep line passing through p.
(A boundary condition occurs if p lies on another segment s′. In this case, we require only that s and s′ be placed consecutively into T.)

If p is the right endpoint of a segment s, then s is to be deleted from the total order. Lines 9–10 return TRUE if there is an intersection between the segments surrounding s in the total order defined by the sweep line passing through p; these segments will become consecutive in the total order when s is deleted.

Scanned by CamScanner

If these segments do not intersect, line 11 deletes segment s from the total order.
Finally, if no intersections are found in processing all the 2n event points, line 12 returns FALSE.

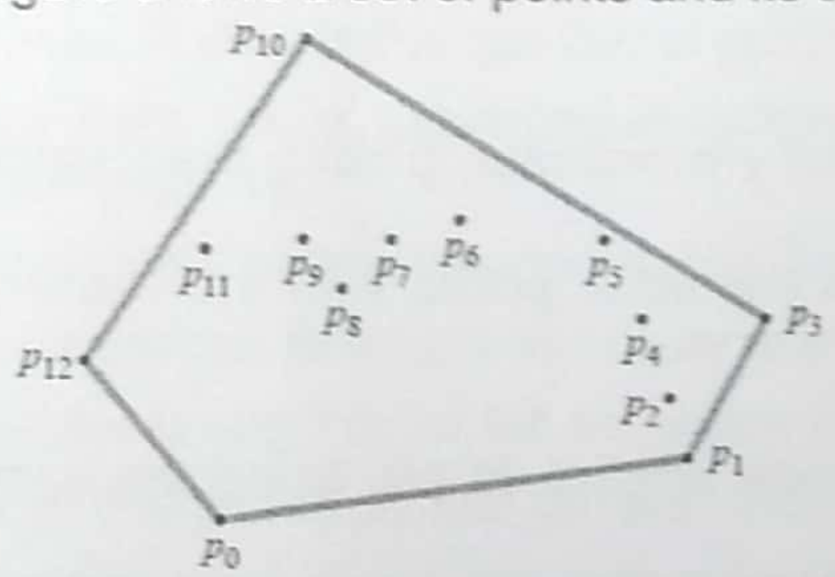
Running time :
If there are n segments in set S, then ANY-SEGMENTS-INTERSECT runs in time O(n log n).
Line 1 takes O(1) time.
Line 2 takes O(n log n) time, using merge sort or heapsort.

Since there are 2n event points, the for loop of lines 3–11 iterates at most 2n times.
Each iteration takes O(log n) time, since each red-black-tree operation takes O(log n) time and, each intersection test takes O(1) time.
The total time is thus O(n log n).

Finding the convex hull :
The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

We denote the convex hull of Q by CH(Q).
Intuitively, we can think of each point in Q as being a nail sticking out from a board. The convex hull is then the shape formed by a tight rubber band that surrounds all the nails.

Figure shows a set of points and its convex hull.



In this section, we shall present algorithm that computes the convex hull of a set of n points known as 'Graham's Scan'.
It outputs the vertices of the convex hull in counterclockwise order.
Graham's scan algorithm, runs in O(n log n) time.

Graham's scan :
Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points.

Each point of the input set Q is pushed once onto the stack, and the points that are not vertices of CH(Q) are eventually popped from the stack.

When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary.

The procedure GRAHAM-SCAN takes as input a set Q of points, where $|Q| \geq 3$.

It calls the functions TOP(S), which returns the point on top of stack S without changing S, and NEXT-TO-TOP(S), which returns the point one entry below the top of stack S without changing S.

Scanned by CamScanner

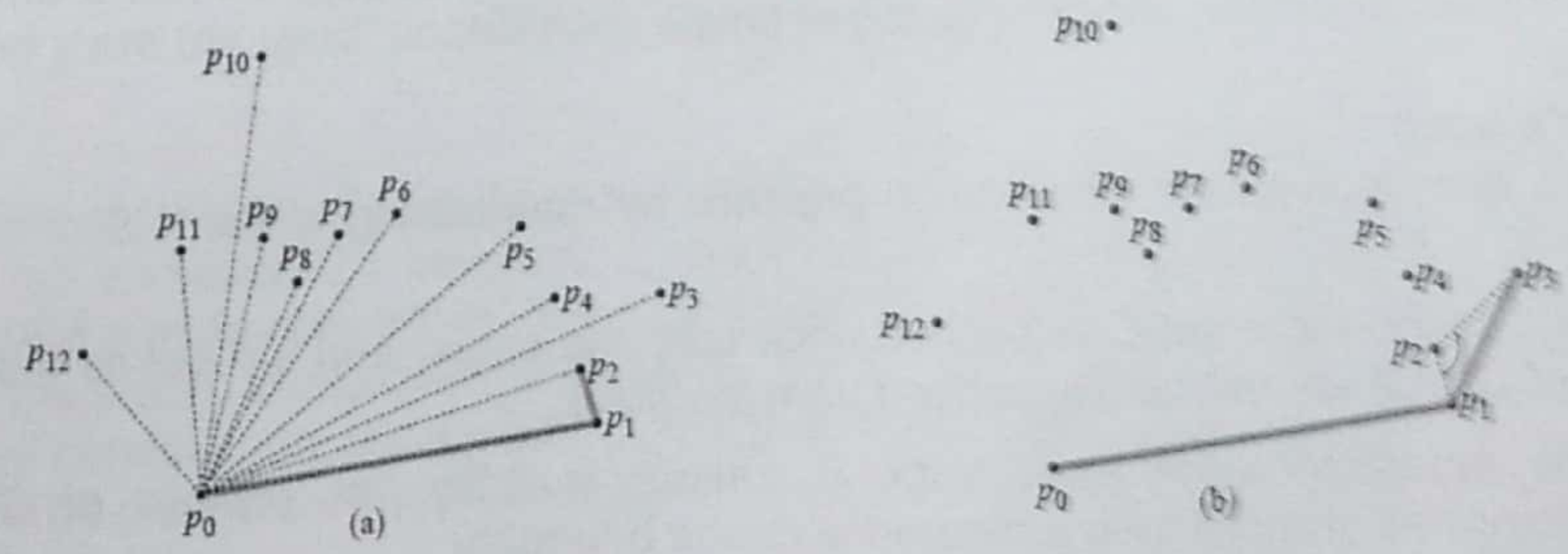
The stack S returned by GRAHAM-SCAN contains, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

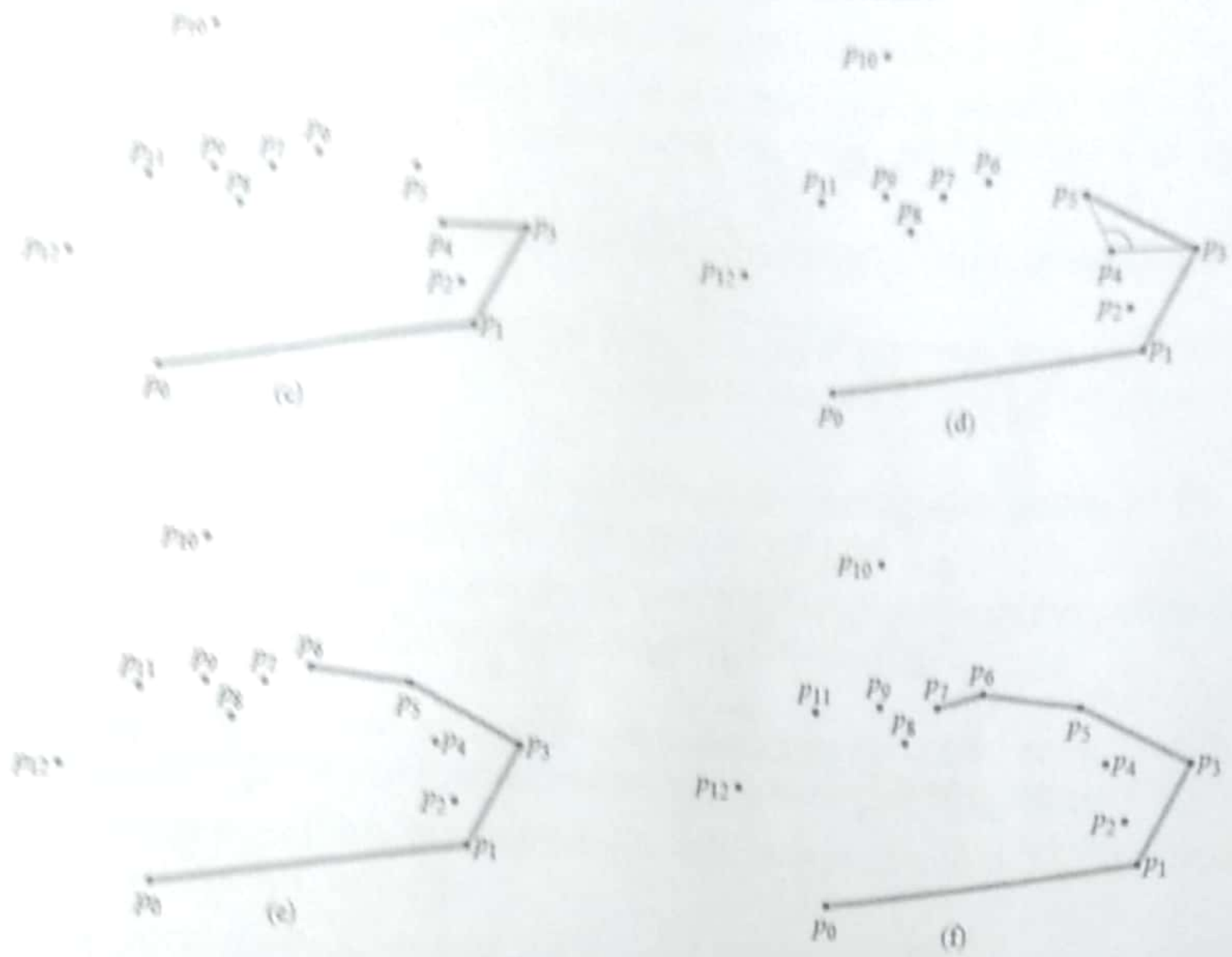
GRAHAM-SCAN(Q)

1) Let p0 be the point in Q with the minimum y-coordinate, or the leftmost such point in case of a tie
2) Let <p1, p2, . . . , pm> be the remaining points in Q, sorted by polar angle in counterclockwise order around p0 (if more than one point has the same angle, remove all but the one that is farthest from p0)
3) PUSH(p0, S)
4) PUSH(p1, S)
5) PUSH(p2, S)
6) for i = 3 to m do
   {
7) while the angle formed by points NEXT-TO-TOP(S), TOP(S), and pi makes a nonleft turn do
   {
8) POP(S)
   } // end of while
9) PUSH(pi, S)
   } // end of for
10) return S

Figure illustrates the progress of GRAHAM-SCAN.
Line 1 chooses point p0 as the point with the lowest y-coordinate, picking the leftmost such point in case of a tie.
Since there is no point in Q that is below p0 and any other points with the same y-coordinate are to its right, p0 is a vertex of CH(Q).

Line 2 sorts the remaining points of Q by polar angle relative to p0, using the same method—comparing cross products.
If two or more points have the same polar angle relative to p0, all but the farthest such point are convex combinations of p0 and the farthest point, and so we remove them entirely from consideration.





We let m denote the number of points other than p_0 that remain.

The polar angle, measured in radians, of each point in Q relative to p_0 is in the half-open interval $[0, \pi)$.

Since the points are sorted according to polar angles, they are sorted in counterclockwise order relative to p_0.

We designate this sorted sequence of points by $<p_1, p_2, \ldots, p_m>$.

Note that points p1 and pm are vertices of CH(Q).

The figure (a) shows the ealrier points sequentially numbered in order of increasing polar angle relative to p_0.

The remainder of the procedure uses the stack S.

Lines 3–5 initialize the stack to contain, from bottom to top, the first three points p_0, p_1, and p_2. Figure (a) shows the initial stack S.

The for loop of lines 6–9 iterates once for each point in the subsequence $<p_3, p_4, \ldots, p_m>$.

The intent is that after processing point pi, stack S contains, from bottom to top, the vertices of $CH(\{p_0, p_1, \ldots, p_i\})$ in counterclockwise order.

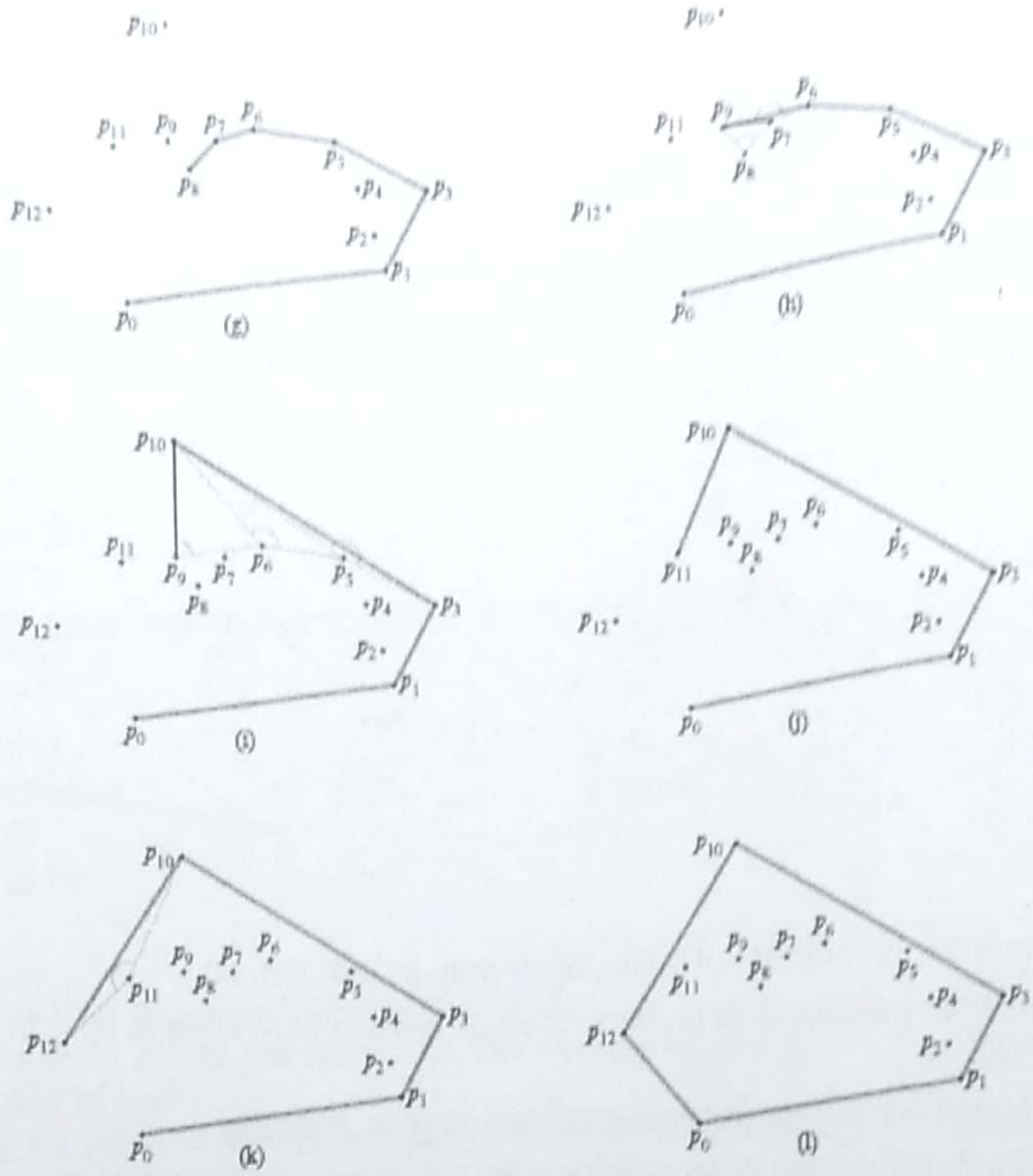
The while loop of lines 7–8 removes points from the stack if they are found not to be vertices of the convex hull. When we traverse the convex hull counterclockwise, we should make a left turn at each vertex.

Thus, each time the while loop finds a vertex at which we make a nonleft turn, the vertex is popped from the stack.

After we pop all vertices that have nonleft turns when heading toward point p_i, we push p_i onto the stack.

Figures (b)–(k) show the state of the stack S after each iteration of the for loop.

Scanned by CamScanner



Finally, GRAHAM-SCAN returns the stack S in line 10.
Figure (l) shows the corresponding convex hull.

Finding the closest pair of points :

We now consider the problem of finding the closest pair of points in a set Q of $n \geq 2$ points.

"Closest" refers to the usual euclidean distance: the distance between points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is square root of $((x_1 - x_2)^2 + (y_1 - y_2)^2)$.

This problem has applications in, for example, traffic-control systems. A system for controlling air or sea traffic might need to know which are the two closest vehicles in order to detect potential collisions.

A brute-force closest-pair algorithm simply looks at all the ${}^nC_2 = 2(n^2)$ pairs of points.

In this, we shall describe a divide-and-conquer algorithm for this problem whose running time is described by the recurrence $T(n) = 2T(n/2) + O(n)$. Thus, this algorithm uses only $O(n \log n)$ time.

The divide-and-conquer algorithm :

Each recursive invocation of the algorithm takes as input a subset $P \subseteq Q$ and arrays X and Y , each of which contains all the points of the input subset P.

The points in array X are sorted so that their x-coordinates are monotonically increasing.

Scanned by CamScanner

Similarly, array Y is sorted by monotonically increasing y-coordinate.

A given recursive invocation with inputs P, X, and Y first checks whether $|P| \leq 3$. If so, the invocation simply performs the brute-force method of trying all ${}^{|P|}C_2$ pairs of points and return the closest pair.

If $|P| > 3$, the recursive invocation carries out the divide-and-conquer paradigm as follows.

Divide: It finds a vertical line l that bisects the point set P into two sets PL and PR such that $|PL| = |PR| = |P|/2$, all points in PL are on or to the left of line l, and all points in PR are on or to the right of l.

The array X is divided into arrays XL and XR, which contain the points of PL and PR respectively, sorted by monotonically increasing x-coordinate.

Similarly, the array Y is divided into arrays YL and YR, which contain the points of PL and PR respectively, sorted by monotonically increasing y-coordinate.

Conquer: Having divided P into PL and PR, it makes two recursive calls, one to find the closest pair of points in PL and the other to find the closest pair of points in PR.

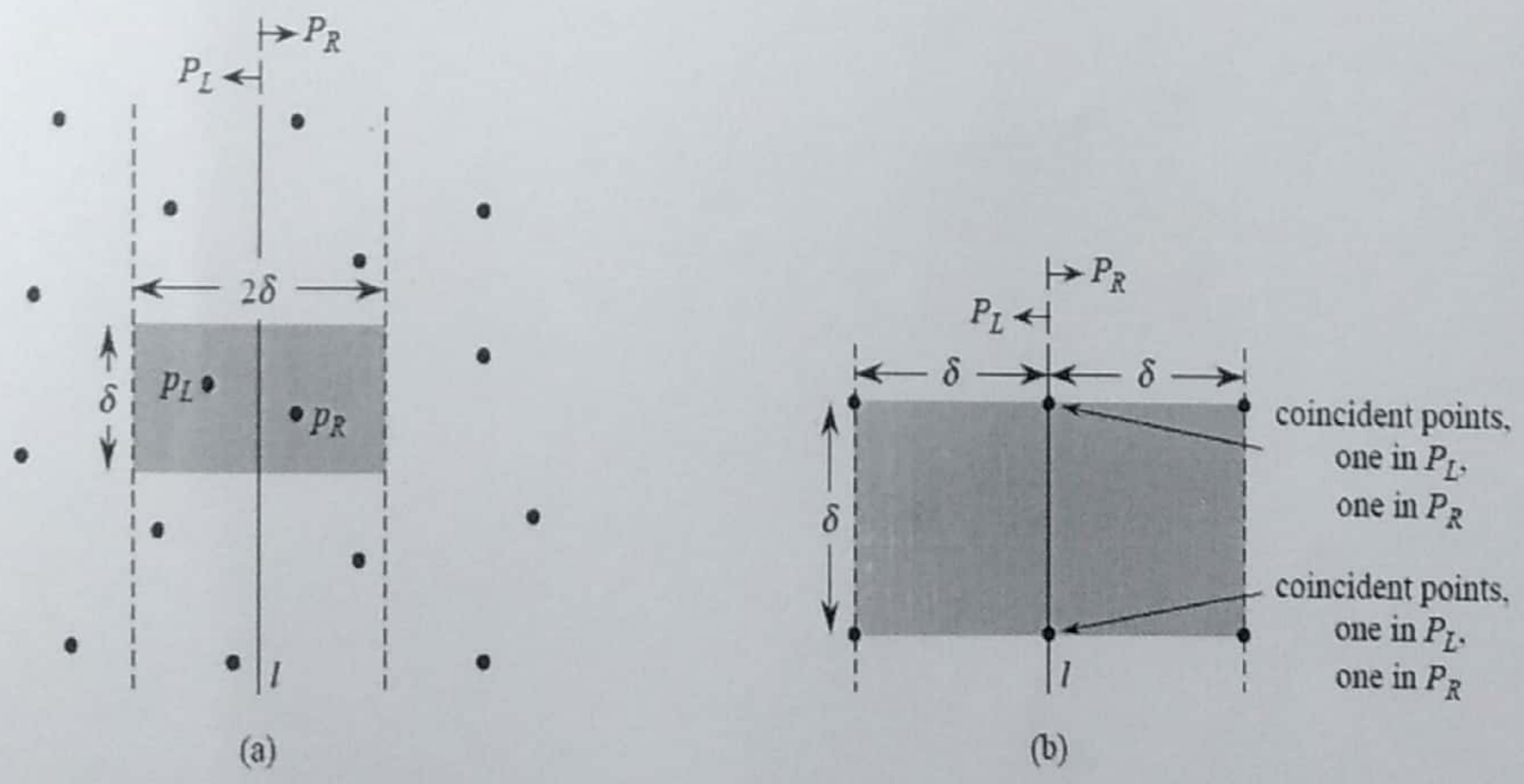
The inputs to the first call are the subset PL and arrays XL and YL; the second call receives the inputs PR, XR, and YR.

Let the closest-pair distances returned for PL and PR be δL and δR, respectively, and let $\delta = \min(\delta L, \delta R)$.

Combine: The closest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair of points with one point in PL and the other in PR.

The algorithm determines if there is such a pair whose distance is less than δ.

Observe that if there is a pair of points with distance less than δ, both points of the pair must be within δ units of line l. Thus, as Figure shows, they both must reside in the 2δ-wide vertical strip centered at line l.



To find such a pair, if one exists, the algorithm does the following :

1. It creates an array Y ', which is the array Y with all points not in the 2δ-wide vertical strip removed. The array Y ' is sorted by y-coordinate, just as Y is.
2. For each point p in the array Y ', the algorithm tries to find points in Y ' that are within δ units of p. As we shall see shortly, only the 7 points in Y ' that follow p need be considered. The algorithm computes the distance from p to each of these 7 points and keeps track of the closest-pair distance δ' found over all pairs of points in Y'.
3. If δ' < δ, then the vertical strip does indeed contain a closer pair than was found by the recursive calls. This pair and its distance δ' are returned. Otherwise, the closest pair and its distance δ found by the recursive calls are returned.

❑❑❑❑❑

Scanned by CamScanner

Module

6

NP-Completeness and Approximation Algorithms

SYLLABUS

Weightage : 18 Marks

NP-Completeness and Approximation Algorithms

NP-Completeness: NP-Completeness and reducibility, NP-Completeness proofs, NP-Complete problems-The vertex cover problem, The travelling salesman problem.

Help Line : For any query WhatsApp to 704 501 85 39 to get it Solved.

Scanned by CamScanner

NON-DETERMINISTIC POLYNOMIAL ALGORITHMS

Polynomial Time

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is O(n^k) for some nonnegative integer k, where n is the complexity of the input.

Polynomial-time algorithms are said to be "fast."
Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time.

Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

Polynomial Time Verification

Given a problem instance and solution (certificate), the process of verifying in polynomial time whether the given solution solves the problem or not is called polynomial time verification.

P, NP and NPC Classes :

- **P Class :** The class P consists of those problems that are solvable in polynomial time. Specifically, they are the problems that can be solved in time $O(n^k)$ for some constant k, where n is the size to the input problem.

- **NP Class :** The NP class consists of those problems that are verifiable in polynomial time. It means that if we were given a certificate of a solution, then we would verify that the certificate is correct in polynomial time of the size of the input.
  Any problem in P is also in NP, since if the problem is in P then we can solve it in polynomial time without using certificate.

- **NPC Class** : The problem is in NP Complete (NPC) class, if it is in NP and is as hard as any problem in NP.
  If any NP Complete problem has polynomial time solution then all NP Complete problem has polynomial time solution.
  Most of the theoretical computer scientists consider NP Complete problems to be intractable, since given the wide range of NP Complete problems studied till date without anyone having discovered a polynomial time solution to any of them.

Reducibility

Let us consider a decision problem A which we would like to solve in polynomial time.
Now suppose there is a different decision problem B, that we already know how to solve in polynomial time.

Suppose that we have a procedure that transforms any instance (input) x of A into some instance y of B with the following characteristics.

1. The transformation takes polynomial time.
2. The answers are the same. That is, the answer of x is 'yes' if and only if answer of y is also 'yes'.

Scanned by CamScanner

Such a procedure is called 'polynomial time reduction algorithm' and it provides a way to solve problem A in polynomial time:

1. Given instance x of problem A, use the polynomial time reduction algorithm to transform it to instance y of problem B.
2. Run the polynomial time decision algorithm for B on the instance y.
3. Use the answer for y as answer for x.

As long as each of these steps takes polynomial time, all these together would also take polynomial time.

So by reducing solving problem A to solving problem B, we have used the easiness of B to prove the easiness of A.

Taking the idea further, we could use polynomial time reductions to show that no polynomial time algorithm can exist for particular problem B.

Suppose we have a decision problem A for which we already know that no polynomial time algorithm can exist.

Further we have a polynomial time reduction algorithm transforming instances of A to instances of B.

Now a simple proof by contradiction can be used to show that no polynomial time algorithm can exist for problem B.

Suppose B has a polynomial time algorithm, then using the above method we can show that problem A also has polynomial time algorithm.

But then it contradicts with our basic assumption that there is no polynomial time algorithm for problem A.

NP Completeness :

Polynomial time reductions provide formal means of showing that one problem is at least as hard as another, to within a polynomial time factor.

Now we can define a set of NP Complete problems.

A problem X is said to be NP complete if

1. X belongs to NP
2. Every other problem Y of NP can be reduced to X in polynomial time.

If the problem satisfied the property 2 but not necessarily property 1 then the problem is said to be NP-hard.

NP Completeness Proofs :

The method for proving that the given problem is NP complete is as follows:

1. Prove the given problem X belongs to NP.
2. Select a known NP complete problem Y.
3. Describe an algorithm that computes function f mapping every instance $x \in \{0,1\}^*$ of problem Y to an instance f(x) of problem X.
4. Prove that the function f satisfies $x \in Y$ if and only if $f(x) \in X$ for all $x \in \{0,1\}^*$.
5. Prove that algorithm computing f runs in polynomial time.

Vertex Cover :

Vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set.

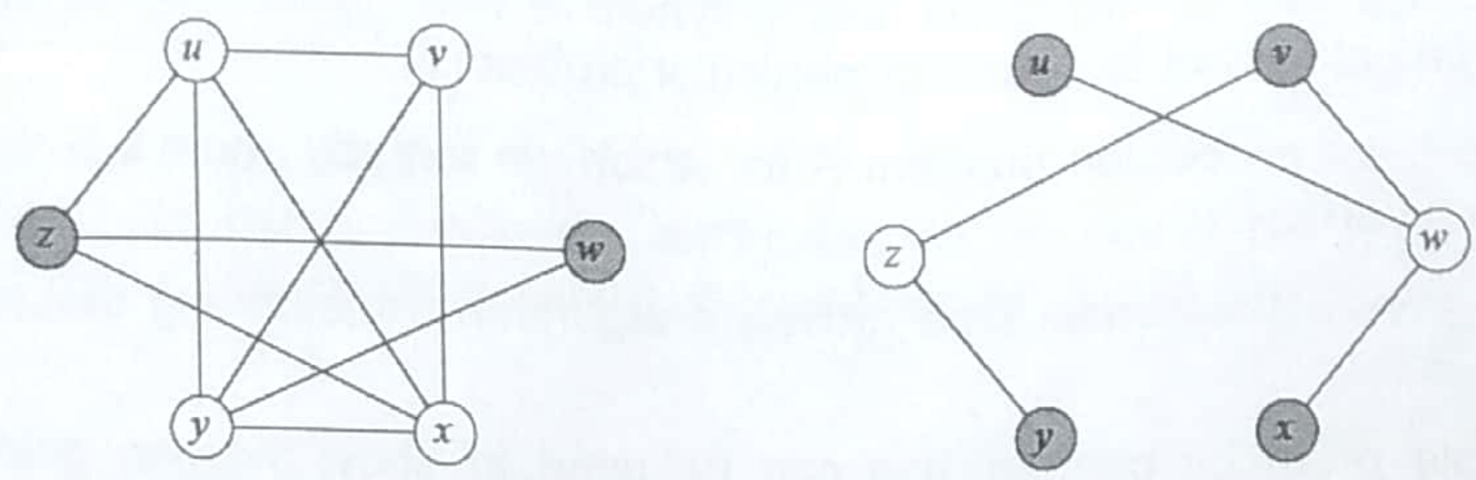
Scanned by CamScanner

Clique :
A clique is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent.

The Vertex-Cover problem :
A vertex cover of an undirected graph G = (V, E) is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both).

That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E.

The size of a vertex cover is the number of vertices in it. For example, the graph in Figure (b) has a vertex cover {w, z} of size 2.



(a) (b)

The vertex-cover problem is to find a vertex cover of minimum size in a given graph. Restating this optimization problem as a decision problem, we wish to determine whether a graph has a vertex cover of a given size k.
As a language, we define

VERTEX-COVER = {<G, k>: graph G has a vertex cover of size k}

Prove that the vertex-cover problem is NP-complete.
We first show that VERTEX-COVER $\in$ NP. Suppose we are given a graph G = (V, E) and an integer k.
The certificate we choose is the vertex cover $V' \subseteq V$ itself. The verification algorithm affirms that $|V'| = k$, and then it checks, for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$. This verification can be performed straightforwardly in polynomial time.

We prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_P$ VERTEX-COVER. This reduction is based on the notion of the "complement" of graph as G (Bar).

Reduction algorithm takes instance (G,k) of Clique problem and calculate G(Bar) in polynomial time.

Output is instance (G-Bar, |V|-k) of vertex cover problem.

To complete a proof we have to show that a graph G has a clique of size k iff G-Bar has a vertex cover of size |V|-k.

Suppose G has a clique V' such that |V'| = k then we claim that V-V' is vertex cover in G-Bar.

Let (u,v) be any edge in E-Bar then $(u,v) \notin E$ which implies at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge in E.

Scanned by CamScanner

Equivalently, at least one of u or v is in V-V' which means edge (u,v) is covered by V-V'.

Since (u,v) was chosen arbitrarily from E-Bar hence every edge in E-Bar is covered by vertex in V-V'.

Hence, set V-V' which has size |V| - k forms a vertex cover for G-Bar.

Conversely, suppose G-Bar has a vertex cover V' where |V'|= |V|-k.

Then for all u, v $\in$ V, if (u, v) $\in$ E-Bar then u $\in$ V' or v $\in$ V' or both.

The contrapositive of this implication is that for all u, v $\in$ V if u $\notin$ V' and v $\notin$ V' then (u, v) $\in$ E.

In other words, V – V' is a clique of size |V| - |V'| = k.

So Clique is polynomially reducible to Vertex cover.

Hence vertex cover is also NP Complete.

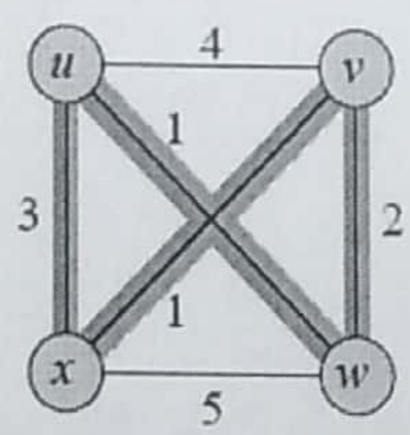
The Traveling-Salesman Problem :

In the traveling-salesman problem, which is closely related to the Hamiltonian cycle problem, a salesman must visit n cities.

Modelling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a tour, or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from.

There is an integer cost c(i, j) to travel from city i to city j, and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.

For example, in Figure, a minimum-cost tour is <u, w, v, x, u>, with cost 7.



The formal language for the corresponding decision problem is

TSP = {<G, c, k> : G = (V, E) is a complete graph,

c is a function from $V \times V \rightarrow Z$,

k $\in$ Z, and

G has a traveling-salesman tour with cost at most k}

Prove that the traveling-salesman problem is NP-complete.

We first show that TSP belongs to NP. Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour.

The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k.

This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE $\leq$ P TSP.

Let G = (V, E) be an instance of HAM-CYCLE. We construct an instance of TSP as follows.

Scanned by CamScanner

We form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, and we define the cost function c by

$$c(i,j) = 0 \quad \text{if } (i, j) \in E$$
$$= 1 \quad \text{if } (i, j) \notin E$$

(Note that because G is undirected, it has no self-loops, and so $c(v, v) = 1$ for all vertices $v \in V$.) The instance of TSP is then $(G', c, 0)$, which is easily formed in polynomial time.

We now show that graph G has a Hamiltonian cycle if and only if graph G′ has a tour of cost at most 0.

Suppose that graph G has a Hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G′. Thus, h is a tour in G′ with cost 0.

Conversely, suppose that graph G′ has a tour h′ of cost at most 0.

Since the costs of the edges in E′ are 0 and 1, the cost of tour h′ is exactly 0 and each edge on the tour must have cost 0.

Therefore, h′ contains only edges in E. We conclude that h′ is a Hamiltonian cycle in graph G.

❑❑❑❑❑

Scanned by CamScanner