Date
Page

Unit - 1

Introduction to Data Structure

→ Generally, computer programs are written to solve the real world problems by the use of computing device.

A computer program consists of

Program = Algorithm + Data Structure

→ Algorithm is a step by step procedure to solve any problems.

Algorithm must have the following properties

(i) Algorithm must have the finite number of steps.

(ii) Algorithm must have some input and have to generate some outputs.

(iii) Algorithm must terminate after certain amount of time.

→ While writing an efficient program we need to select the best algorithm so that we can perform the computation in efficient time and space (computer memory).

→ There are different factors that affects the efficiency of an algorithm. Among those factors data structure is the most important factor.

→ Data structure is the way how we declare, store and arrange the variables, constants and other values during writing a program.

→ The efficient use of data structure leads to enhance the performance of an algorithm.

For example, while writing a program to find the largest number among 50 numbers, we can declare variables individually as well

CamScanner

as we can declare variable using array.

The use of array will be efficient in this case, it is because we can access the elements in an array faster compared to accessing the individuals. Thus the array representation helps to accessing the individuals. Thus the array will be used. reduce the acess time of an element.

→ Similarly, consider the problem to calculate the fibonacci series to any number, we can use the different approach to obtain series like iteration and recursion.

→ The selection of the algorithm implementation also affects the efficiency of the program. Sometimes iteration will be better and sometimes recursion will be better.

Classification of data structure

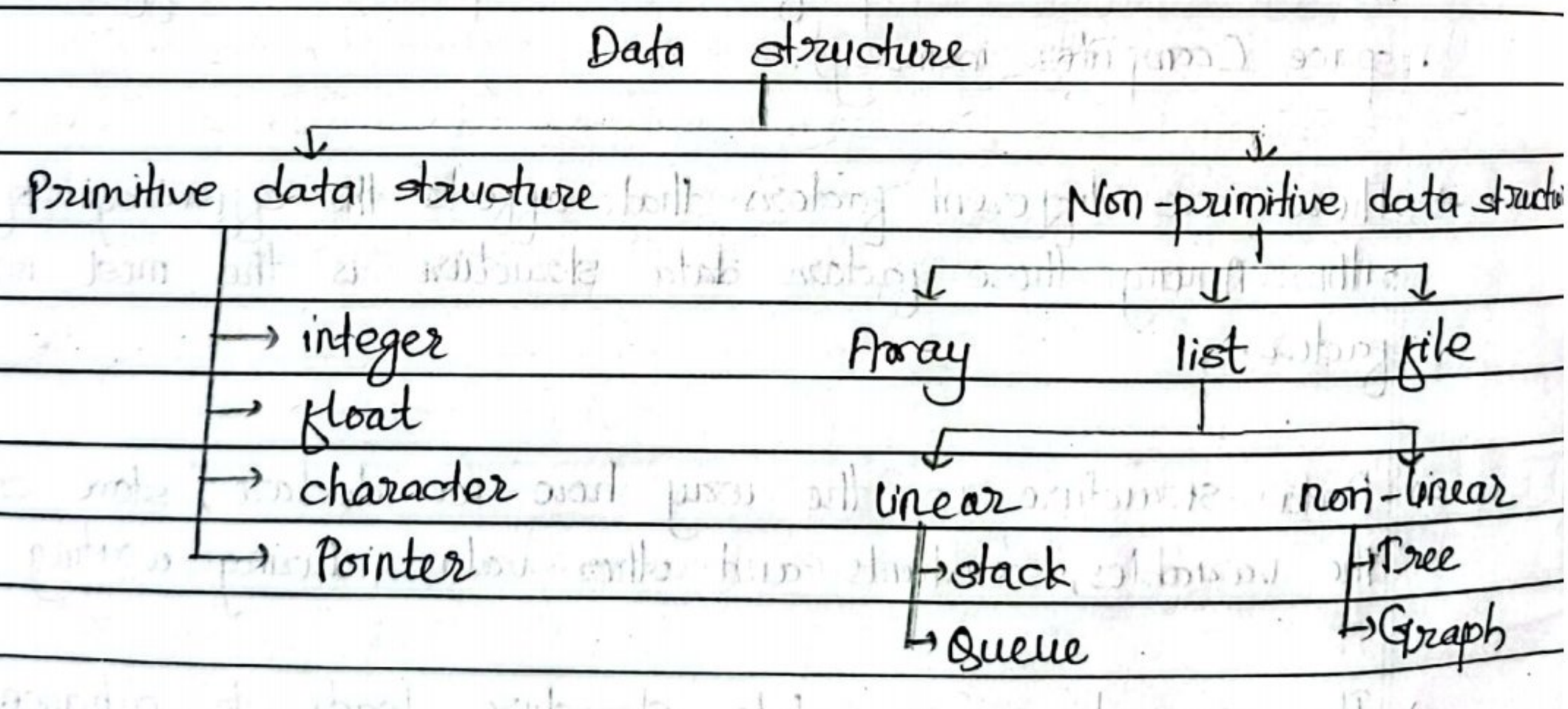


fig:- Classification of data structure

Basic Operations in data structures:

→ Operation in data structure means different action that can be applied upon it.

→ Some of the operations are as:

(a) Insert: Process of adding element in a data structure.

(b) Delete: Process of deleting/removing elements from the data structure.

(c) Traversing: Accessing each and every element of a data structure

(d) Update: Process of substituting one value by the other value in the data structure.

(e) Sorting: Arranging the data into some logical order (Alphabetical Ascending, Descending, etc).

(f) Searching: Process of finding the address allocated for some given value (key).

CamScanner

Types of Algorithms

(i) Divide and conque algorithms
(ii) Deterministic and non-deterministic algorithms.
(iii) Series and Parallel algorithms.
(iv) Greedy algorithms

* Divide and conquer algorithm

→ The divide and conquer algorithm is designed to breakdown the problem recursively into sub-problems until subproblems becomes small so that they can be directly solved.

→ Divide and conquer algorithms are best for solving the complex problems.

→ Divide and conquer algorithms are best for solving the complex problems. Major phase of this algorithm are as follows:-

* Breaking problems into sub-problems.
* Solving the trivial cases.
* Combining the solution of the subproblems to solve the original problem.

→ A simple varient of divide and conquer algorithm is called reduce and conquer approach that solves bigger problems by decreasing their sizes.

→ An example of this algorithm is Binary search Algorithm.

CamScanner

Date
Page

* Deterministic and non-deterministic algorithm

→ Deterministic algorithm solve the problem with exact decision at every step of the algorithm.

→ Given any input the algorithm will always produce same output and will always pass through same sequence of steps. in case of deterministic algorithm.

→ In case of non-deterministic algorithm the output of an algorithm can be different (cannot be determined explicitly).

→ Simple example of non-deterministic algorithm is the generation of random numbers.

* Series and Parallel algorithm

→ A sequential algorithm or serial algorithm is that which is executed serially i.e the instructions of an algorithm execute one at a time.

→ Serial algorithm are used to solve the serial problem i.e those problem which cannot be divided into pieces because they need the result of previous step to operate on next step.

→ Example of series algorithm can be calculation of factorial finding roots of quadratic equation. etc.

→ In case of parallel algorithm the problem can be divided into

CS CamScanner
CS CamScanner

pieces and each pieces can be executed at same time by many processing devices.

→ Example can be finding prime number ranging from 1 to 500 where we can assign a parallel algorithm by making subsets of numbers to the available processor and printing the results back together.

(iv) Greedy algorithm

→ The basic principle of greedy algorithm is to solve the problem more quickly.

→ The solution may be the best of all the actual solutions or it may simply approximate the exact solution.

→ Greedy algorithms are basically used in decision making problems whose solution are based on approximation.

→ Shortest path algorithm, travelling salesman problem, spanning tree generating algorithm are some of the greedy algorithms

Analysis of an algorithm

→ The major concern using algorithm to solve any problem is how efficiently and effectively a problem is solved.

→ There may be number of algorithm can be computed by determining the amount of resource consumed by the algorithm

This resource includes:-

(a) Time: the CPU time required to execute the algorithm

(b) Space: the amount of memory used by algorithm for execution.

→ Since memory is extensible but not time so the efficiency of an algorithm is determined on how efficiently it uses time.

→ The running time of a program is a function of 'n' where 'n' is the size of input data.

→ The rate at which the running time of an algorithm increases as a result of an increase in the volume of input data is called the order of growth of an algorithm.

Asymptotic Analysis,

→ It is defined as an idea of analysing the efficiency of an algorithm.

→ In asymptotic analysis we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time).

→ we calculate how the time or space is utilized by an algorithm with the increase in the input size of data.

→ There are three most commonly used notations for analysis of algorithm.

CamScanner

Date
Page

(a) Big O notation (O)
(b) Omega notation (Ω)
(c) Theta notation (Θ)

(a) Big O notation,

→ This notation provides an upper bound on the growth rate of an algorithm's running time or space usage

→ It represents the worst case scenario i.e the maximum amount of time or space an algorithm may use to solve the problem.

→ let's say if the algorithms running time is $O(n)$, then it means the running time of algorithm increases linearly with the input size 'n'.

→ Following table lists some possible order of growth and their corresponding big O notation.

Order of growth	Big O notation
constant	$O(1)$
linear	$O(n)$
logarithmic	$O(\log n)$
Quadradic	$O(n^2)$
Cubic	$O(n^3)$

CamScanner

```
function 1 ( )
{
    for (i=0; i<n; i++)
    {
        - - -
    }
}
```

running time : n

```
function 2 ( )
{
    for (i = 0; i<n; i++)
    {
        for (j=0; j<n; j++)
        {
            - - -
        }
    }
}
```

running time : n^2 $(n*n)$

→ According to their order of growth, the big O notation can be arranged in an increasing order as:

$$O(1) < O(\log n) < O(n) < O(n^2) < O(n^3)$$

CamScanner

Date
Page

(b) Omega notation,

→ This notation provides the lower bound on the growth rate of an algorithm's running time or space usage.

→ It represents the best case scenario i.e minimum amount of time and space an algorithm may need to solve a problem.

→ It is denoted as $\Omega(n)$, which means the running time of algorithm grows linearly with input size 'n' or more.

→ Furthermore this notation explains whether the set of functions will grow faster or at the same rate while executing.

(c) Theta notation,

→ This notation provides both upper and lower bound on the growth rate of an algorithm's running time or space usage.

→ It represents the average case-scenario.

→ It defines the average case of an algorithm's time complexity.

→ Theta notation defines when the set of function lies in both O (expression) and Ω (expression). In this case theta notation is used.

CamScanner

Measurement of complexity of algorithms

→ Based on the asymptotic notation, there are major three case to measure the complexity of algorithm.

(a) Best case
(b) Worst case
(c) Average case

(a) Best case :-

→ In the best case analysis, we calculate the lower bound on the running time of an algorithm.

→ In best case, we must know the case that causes a minimum numbers of operations to be executed.

→ For example; In the linear search problem, the best case occur if the element to be searched is present in the very first position in the array.

→ In the worst case, the number of operations to be performed is constant (don't depends on the size of array)

→ So the time complexity in the best case is $\Omega(1)$.

(b) Worst case :-

→ The upper bound running time of algorithm is calculated in worst case analysis.

CamScanner

→ For linear search problem, the worst case occurs when the element to be serched is located at the last index or the element is not present in the array.

→ In this case, the number of comparision equals to the maximum size of an array leading the increase in time complexity.

→ So, the worst case time complexity of linear search is $O(n)$.

(c) Average case :→

→ In average case, analysis, we take all possible inputs and calculate the ~~input~~ computing time for all of the inputs.

→ Sum all the calculated values and divide by the sum of total no of inputs.

→ In linear search the average case is given by $\left(\frac{n+1}{2}\right)$ i.e average of best and ~~worst~~ worst case.

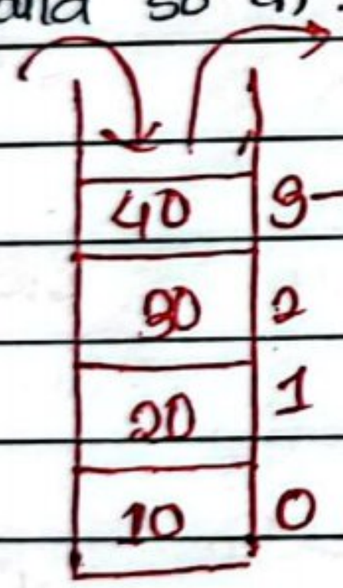
a+b infix
+ab prefix
ab+ postfix

UNIT- 2
STACK AND RECURSION

→ Stack is the linear, data structure, where the insertion & deletion of an element can be done from only one end called top of stacks (tos).

→ It is also called LIFO (Last in first Out) data structure as the element inserted last will be the first to be deleted.

→ Stack are used in various applications like undo sequence in text editor, maintaining history by web browsers, converting of infix to postfix expression, evaluation of postfix expression and so on.



if tos - 1, then empty (stack)
tos → maximum if tos = maximum range of array.

fig: Stack

→ There are two basic operations that can be performed on stack

(a) Push operation
(b) Pop operation

→ Inserting an element in a stack is called push operation and deleting an element from stack is called pop operation.

Stack overflow : when data is inserted when tos is max.
Stack underflow : when data is inserted at tos = -1.

→ Initially, the value of tos is set to -1, and whenever the data is inserted, the value of tos is incremented by 1 and whenever the data is deleted, the value of tos is decremented by 1.

→ Stack is said to be empty if tos = -1 and stack will be full if tos = Max size of array

→ When we try to perform the push operation in the full stack we encounter the stack overflow state and if we want to perform the pop operation on the empty stack we encounter the stack underflow state.

→ The graphical representation of push and pop operation can be shown as :-

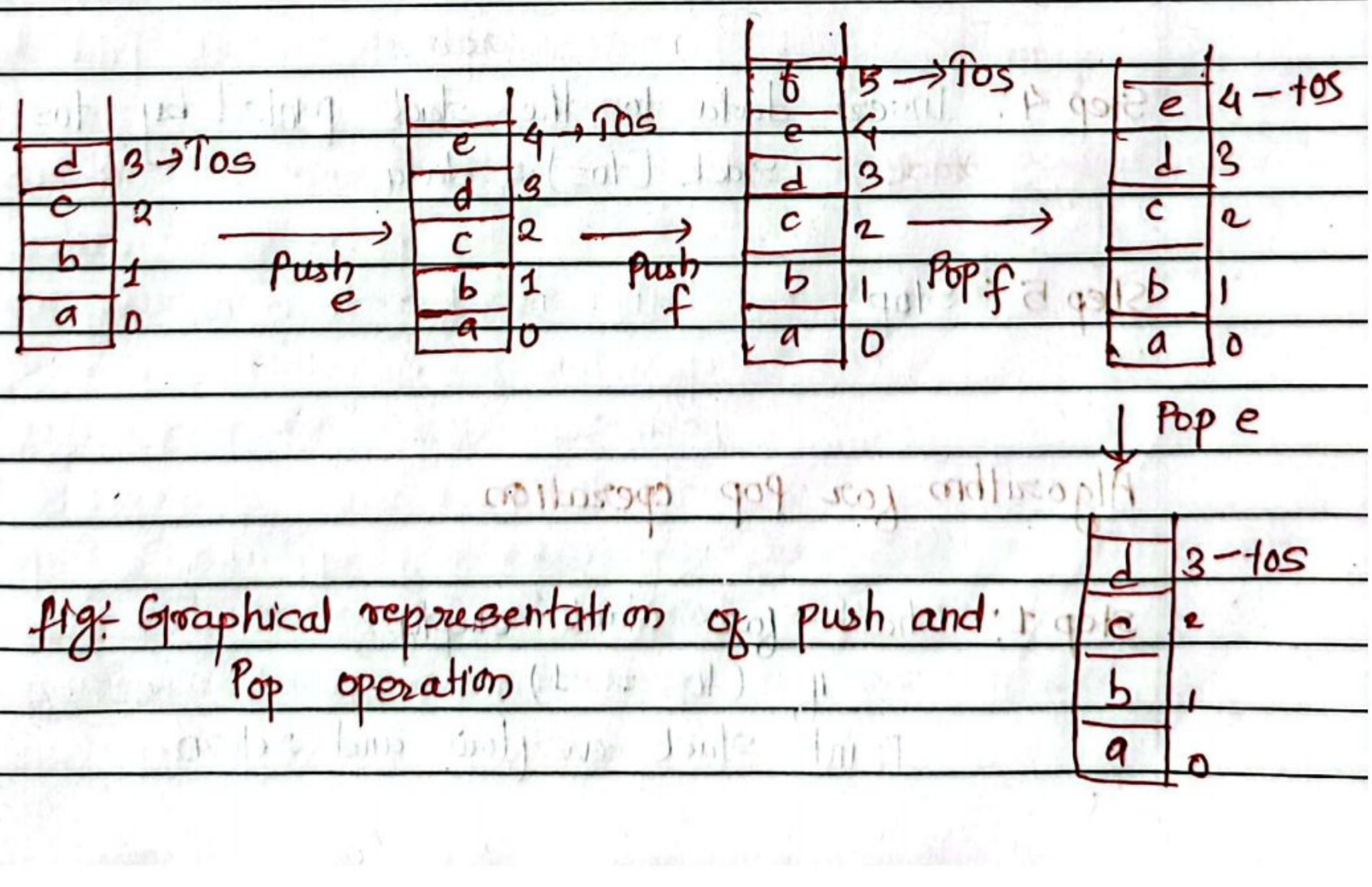


fig:- Graphical representation of push and Pop operation

4. Algorithm for push and pop operation:

Assumption

let stack(n) be the stack of size 'n'. let tos be the index variable which keeps track of the stack content. Stack is said to be full if tos = n-1 and stack is empty if tos = -1.

Algorithm for push operation,

Step 1: Check for stack full
if (tos = n-1)
Print stack overflow and return

Step 2: Read the data in the stack as "data".

Step 3: Increment the value of tos by 1.
i.e tos = tos +1 or tos ++

Step 4: Insert data to the stack pointed by tos index.
stack [tos] = data

Step 5: Stop

Algorithm for pop operation

Step 1: Check for stack empty
if (tos = -1)
Print stack overflow and return.

CamScanner

Step 2: Display the data pointed by tos as deleted
data = stack [tos]

Step 3: Decrement tos by 1.
i.e tos = tos - 1 or tos--

Step 4: Stop

Imp #

Conversion of infix expression to postfix expression

→ If operator is present in between the operand, then it is called infix expression.

→ If operation is present before the operand then it is prefix expression.

→ If operator is present after the operand then it is called postfix expression.

Algorithm to convert infix to postfix operation expression:

Step 1: Define operator-stack and postfix expression

Step 2: Read the infix expression.

Step 3: Scan the infix expression from left to right until end one symbol at a time and repeat.

3.1. If the scanned symbol is opening paranthesis, push the symbol to operator stack.

CamScanner

3.2. If the scanned symbol is closing parenthesis, then pop all the operator from operator stack to postfix expression until opening parenthesis is encounted then stop pop operation and discard both parenthesis.

3.3. If the scanned symbol is an operand, push it to postfix expression.

3.4 If the scanned symbol is an operator:

3.4.1: If the operator stack contains the opening parenthesis or empty, push the operator into operator stack.

3.4.2: If the operator stack contains the operator then compare

3.4.2.1: If the precedence order of scanned symbol is greater than the operator present in operator stack, push the scanned symbol into operator stack.

3.4.2.2: If the precedence order of scanned symbol is less than or equals to the operator present stack, pop the operator from operator stack to postfix expression until the precedence of scanned operator is greater than operator in operator stack

If during pop operation opening parenthesis is found stop. Pop operation and push

decreasing order of precedence ↑ { ^ ④ ; * or / ③ ; + or - ② }

the operator to operator stack.

Step 4: If the scanned symbol is empty, pop all the operator from operator stack to postfix expression.

Step 5: Display postfix expression

Step 6: Stop

Convert the following infix expression to postfix expression.

ⓐ a+b * (c-d)/e

S.N	Scanned Symbol	operator stack	Postfix expression
1.	a		a
2.	+	+	a
3.	b	+	ab
4.	*	+*	ab
5.	(	+*(	ab
6.	c	+*(	abc
7.	-	+*(-	abc
8.	d	+*(-	abcd
9.	)	+*	abcd-
10.	/	+/	abcd-*
11.	e	+/	abcd-*e
12			abcd-*e/+

Date
Page

Q. Convert the following infix expression to postfix expression.

(A + B * C / D) + E * F - (G * H + I - J)

SN	Scanned symbols	Operator stack	Postfix expression
1.	(	(	
2.	A	(	A
3.	+	(+	A
4.	B	(+	AB
5.	*	(+*	AB
6.	C	(+*	ABC
7.	/	(+/	ABC*
8.	D	(+/	ABC*D
9.	)		ABC*D/+
10.	+	+	ABC*D/+
11.	E	+	ABC*D/+E
12.	*	+*	ABC*D/+E
13.	F	+*	ABC*D/+EF
14.	-	-	ABC*D/+EF*+
15.	(	-(	ABC*D/+EF*+
16.	G	-(	ABC*D/+EF*+G
17.	*	-(*	ABC*D/+EF*+G
18.	H	-(*	ABC*D/+EF*+GH
19.	+	-(+	ABC*D/+EF*+GH*
20.	I	-(+	ABC*D/+EF*+GH*I
21.	-	-(-	ABC*D/+EF*+GH*I+
22.	J	-(-	ABC*D/+EF*+GH*I+J
23.	)	-	ABC*D/+EF*+GH*I+J-
			ABC*D/+EF*+GH*I+J--

The corresponding postfix expression is ABC*D/+EF*+GH*I+J--

CamScanner
CamScanner

Conversion of infix to prefix expression.

Algorithm:

Step 1 : Read the given infix expression

Step 2: Reverse the given infix expression.

Step 3: Convert the reversed infix expression to postfix expression.

Step 4 : Again reverse the obtained postfix expression.
(The result will be the required prefix expression)

Infix → A+B
Reverse of infix → B+A
Postfix of reverse → BA+
Reverse of postfix → +AB

Evaluation of Postfix expression

Assumption

let us consider we scan one symbol at a time from left to right during ~~education~~ evaluation of postfix expression.

Algorithm

Date
Page

Step 1:- Define operand-1, operand-2 and operator stack.

Step 2:- Read the postfix expression.

Step 3:- Scan the postfix expression from left to right one symbol at a time and repeat:

3.1:- If the scanned ~~signal~~ symbol is operand push it to operator stack.

3.2:- If the scanned symbol is operator, pop two operand from operator stack to operand-2 and operand 1 respectively. Perform the operation specified by operator and push the result back to operator stack.

Step 4:- when all the symbol is scanned then display the value of operator stack as the result of evaluate postfix expression.

Step 5:- Stop.

Evaluate the given postfix expression
6523 + 8* + 3 + *

SN	Scanned signal	operand 1	Operand 2	operator stack
1	6			6
2.	5			6,5
3.	2			6, 5,2
4.	3			6,5,2,3

CamScanner
CamScanner

5.	+	2	3	6, 5, 5
6.	8			6, 5, 5, 8
7.	*	5	8	6, 5, 40
8.	+	5	40	6, 45
9.	3			6, 45, 3
10.	+	45	3	6, 48
11.	*	6	48	288

The result of given postfix expression is 288.

Abstract Data Type (ADT),

→ Abstraction is the mechanism of hiding the implementation detail i.e hiding the complexity of the user.

→ Abstract datatype (ADT) is the process of hiding the information details from the programmers so that the complexity of the data type need not to be considered while working on the data type.

→ For example, while using the string data type, programmers may not be interested on how the memory is allocated for string, where the memory is allocated or other details, they are just concern with what operation can be done with that string.

→ Programmer just define the string variable & perform the string operation like strcpy, strcmp and so on

→ Therefore ADT is a mathematical model that is concerned with the types of data we are using and the basic operations that can be applied on those data types.

Implementation of ADT

→ Implementation of ADT can be done in two parts
- (a) Value Definition
- (b) Operation Definition
  - ↳ Header
  - ↳ Pre condition
  - ↳ Post condition.

{Stack as an ADT}

Stack as an ADT

→ The stack 'S' with the data item of type 'T' can be implemented as ADT with its value definition (say int) and different operators like;

(a) Empty(S): Returns true if stack is empty, otherwise returns false.

(b) Push (S, x): If stack is not full inserts the value 'x' at the top of stack.

(c) Pop (S): If the stack is not empty deletes the element from top of stack.

(d) Display (S): If the stack is not empty, displays the element of stack.

CamScanner

ADT of stack

Push

<ADT definition>
int data
int tos
} value definition

<process>
Process name : Push
Precondition: Stack must not be full
Method:- Increment the value of tos and insert element in stack pointed by tos index.
Post condition: stack = stack + data
tos = tos+1
<end process>

operation definition

Pop

<ADT definition>
int data
int tos
<Process>
Process name : Pop
Precondition :- Stack must not be empty.
Method :- {Decrement the value of tos} and {delete the respective data}
Post condition :- stack = stack - data
tos = tos - 1
<end process>

CamScanner

Date
Page

stops recursion — force to call recursive function

:# Recursion → (Base condition & inductive condition)

Eg.

(i) for fibonacci series

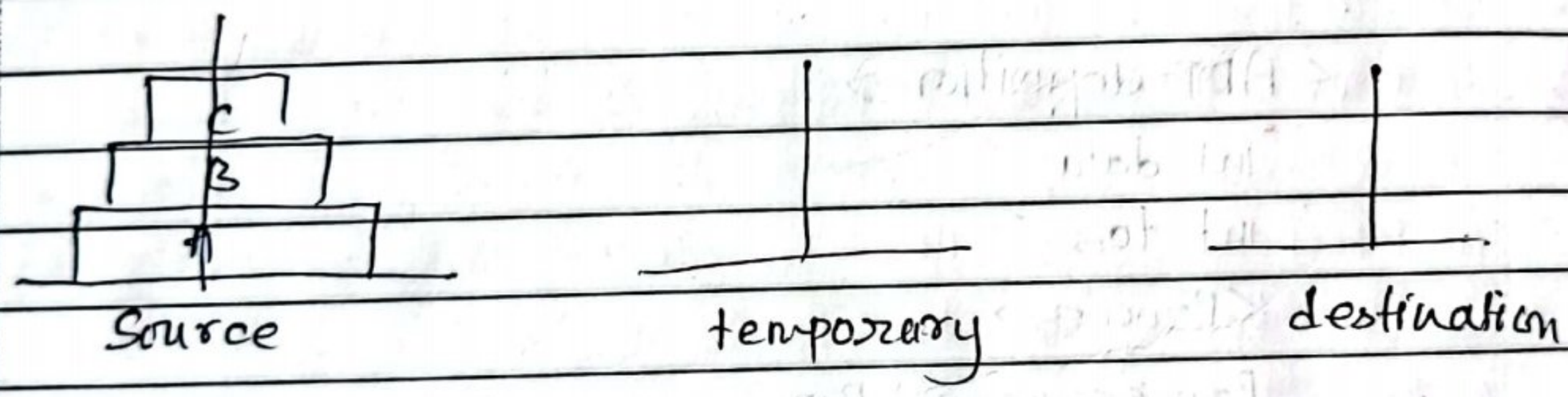
mathematically

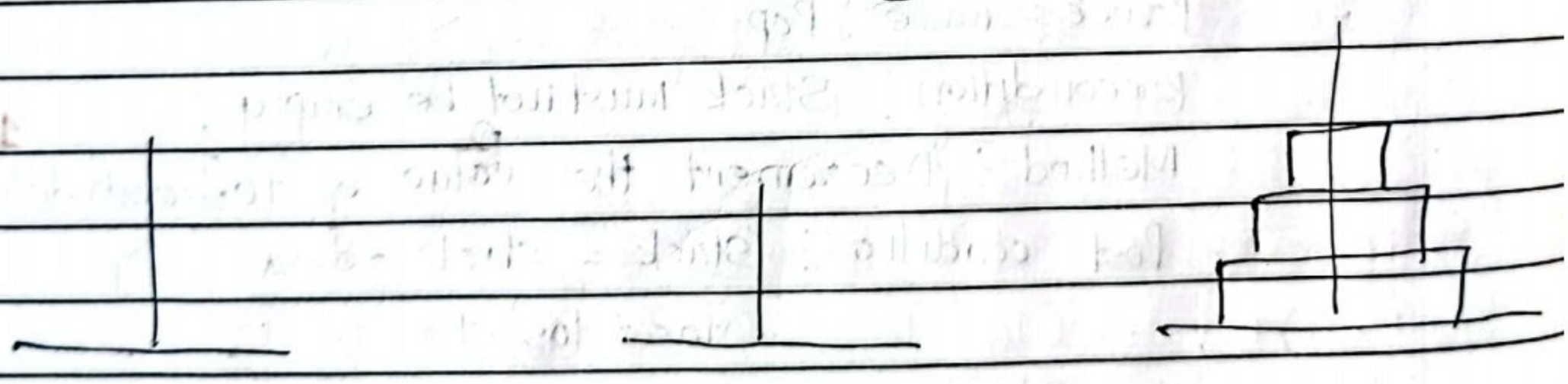
$$f(n) = \begin{cases} n & \text{if } n < 0 \rightarrow \text{base cond}^n \\ f(n-1) + f(n-2) & \text{otherwise} \rightarrow \text{inductive function.} \end{cases}$$

(ii) for factorial

$$f(n) = \begin{cases} 1 & \text{if } n == 0 \rightarrow \text{base cond}^n \\ n \times (n-1)! & \text{otherwise} \rightarrow \text{inductive cond}^n \end{cases}$$

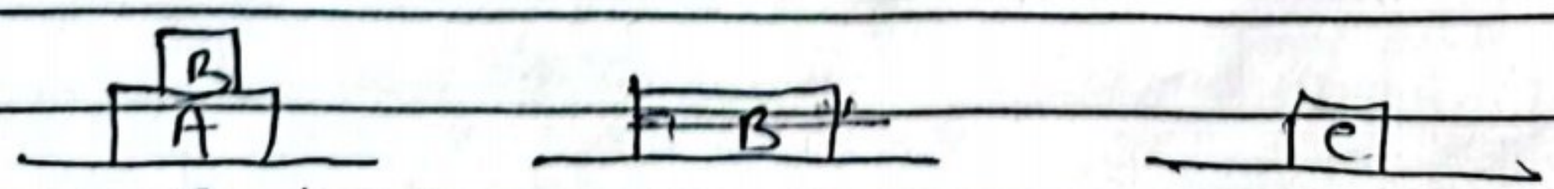
source to dest



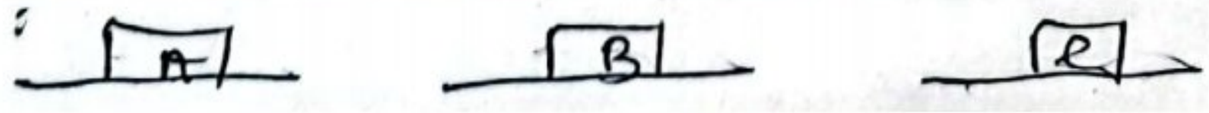


tower of Hanoi

step 1:

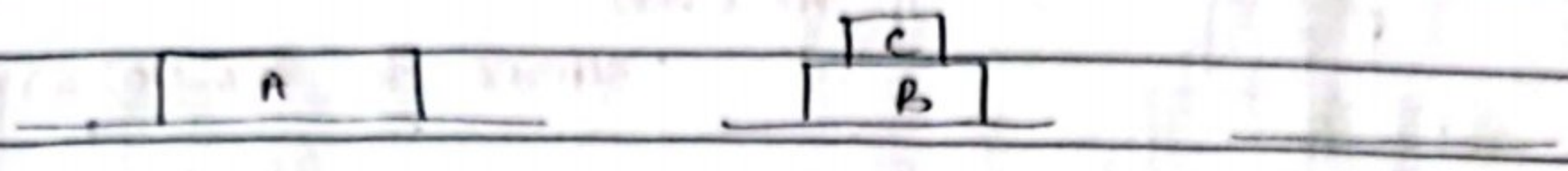


step 2:

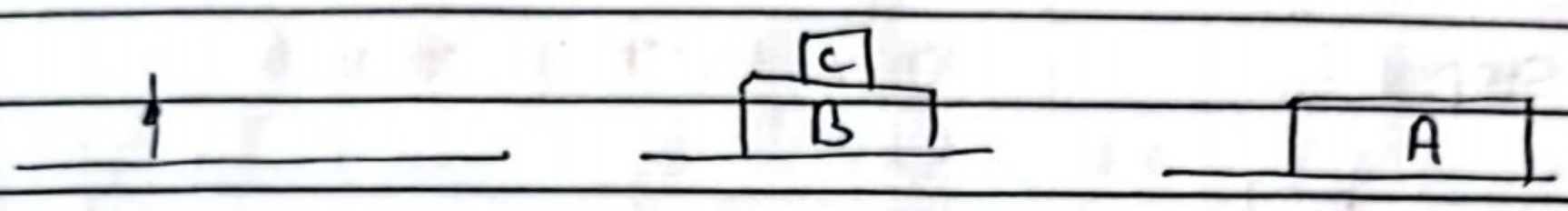


CamScanner

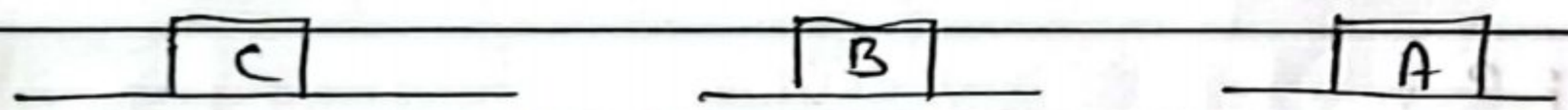
Step 2 :-



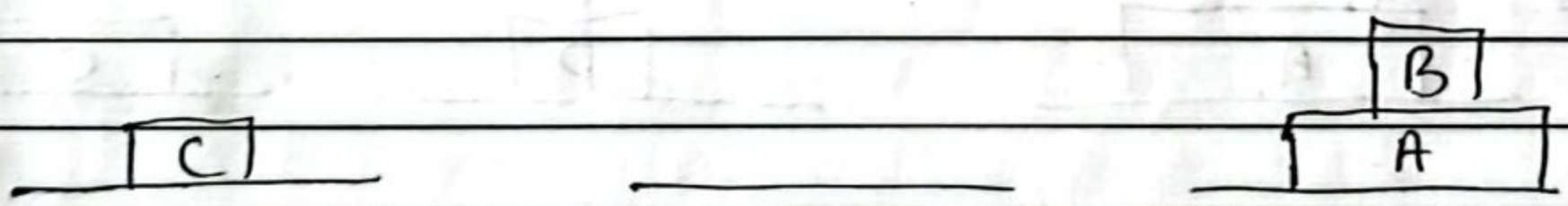
Step 3 :-



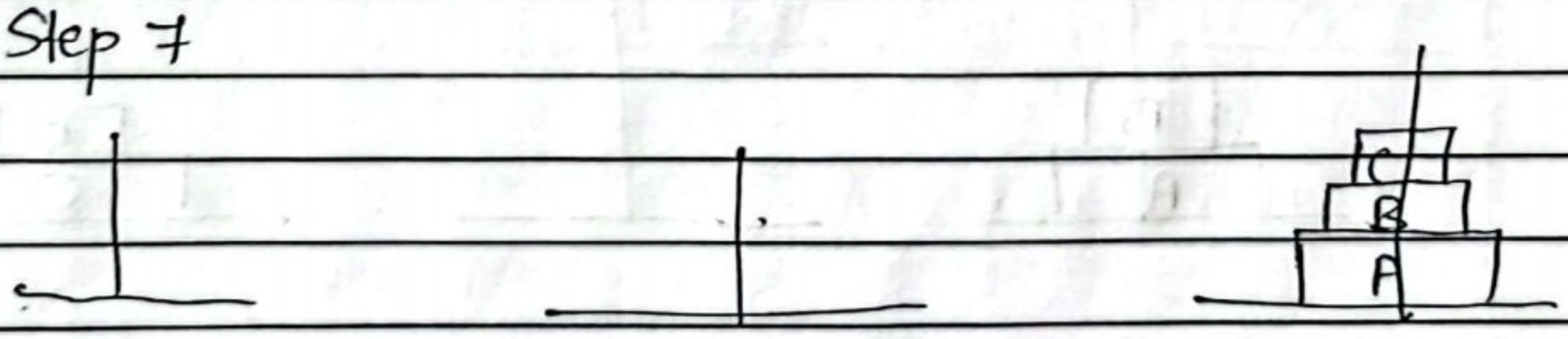
Step 4 :-



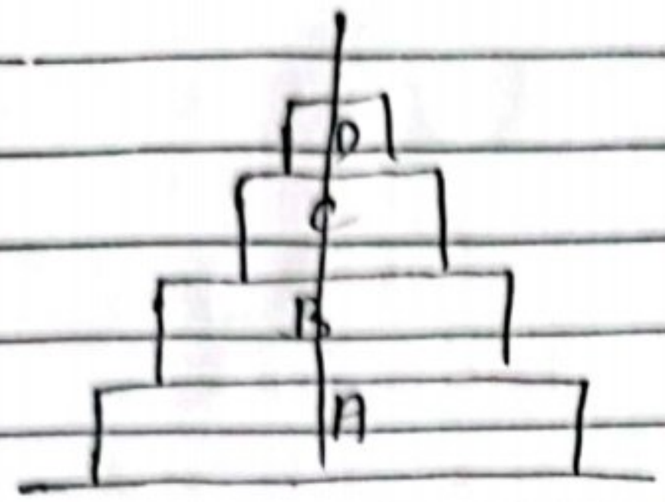
Step 5 :-



Step 7

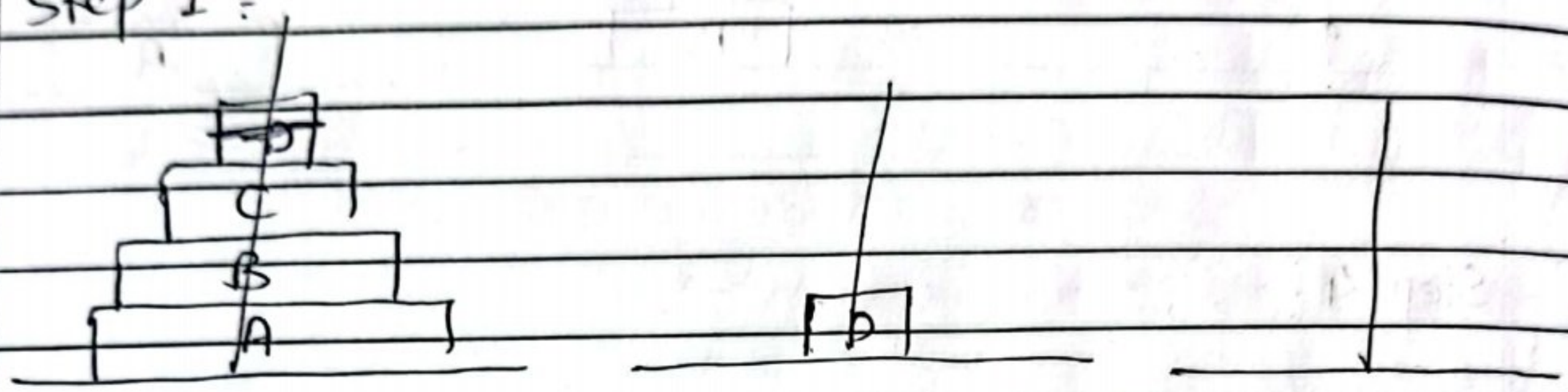


no of steps = $2^n - 1$
n = no of disk.

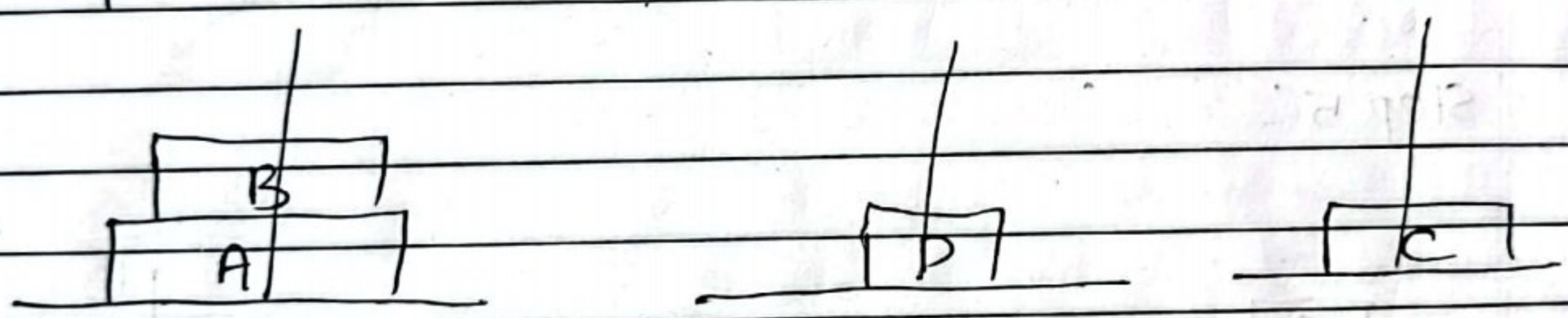


if n = odd.
step 1 : source to destination
if n = even
source to temporary

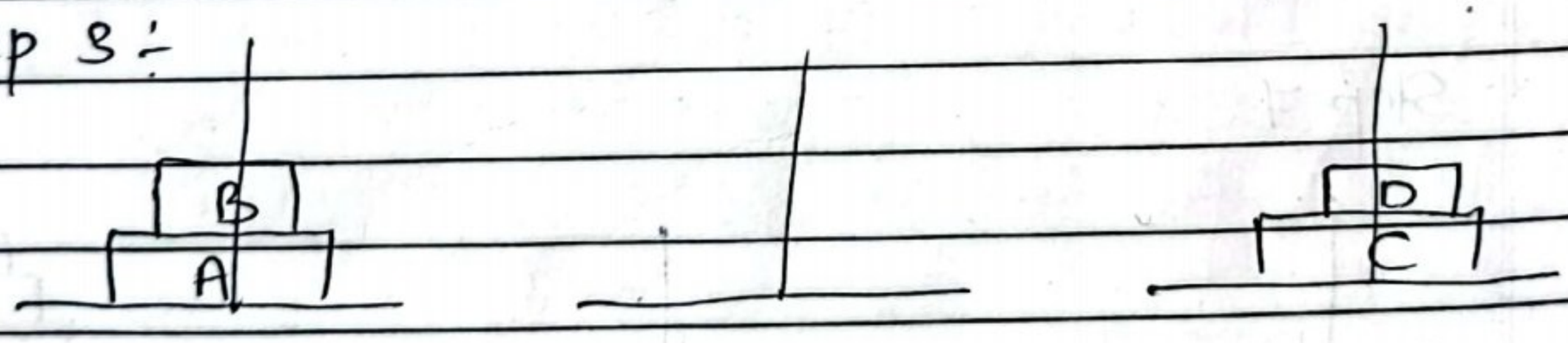
Step 1 :



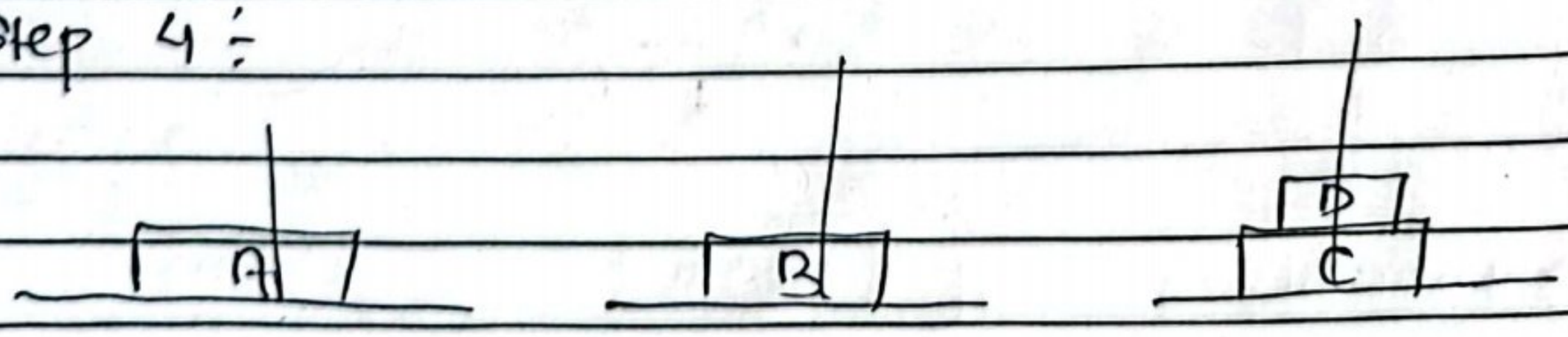
Step 2 :



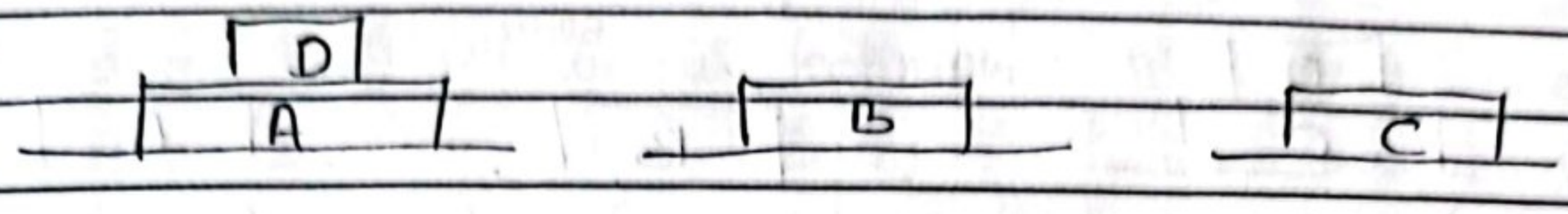
Step 3 :



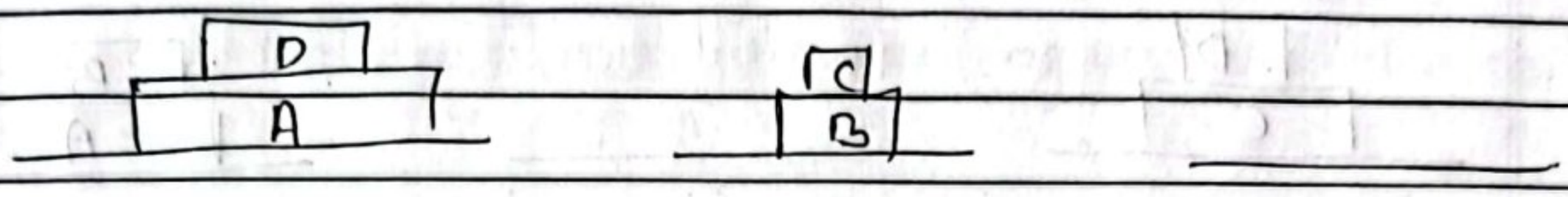
Step 4 :



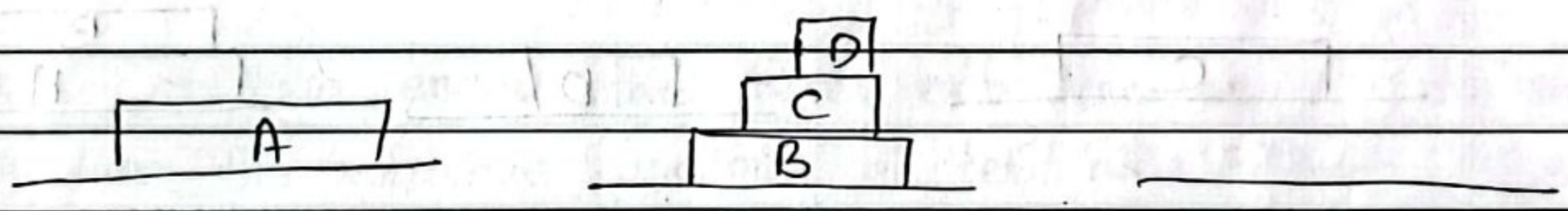
Step 5 :



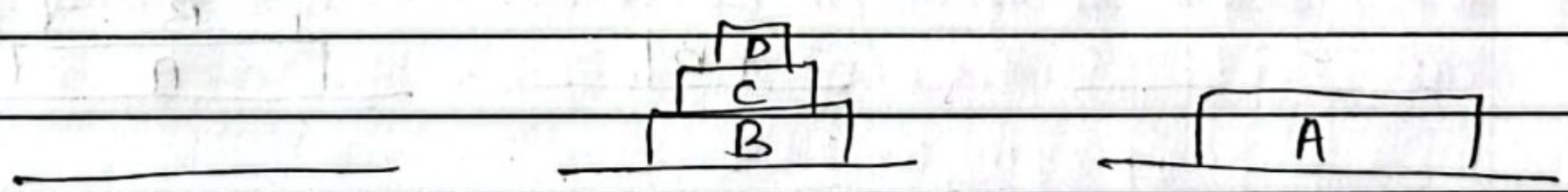
Step 6 :



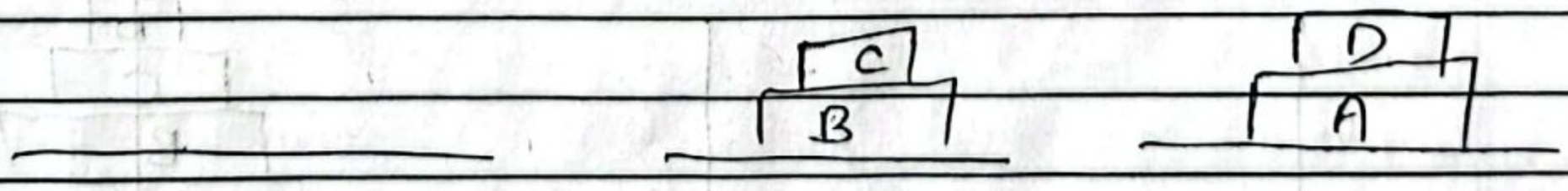
Step 7 :



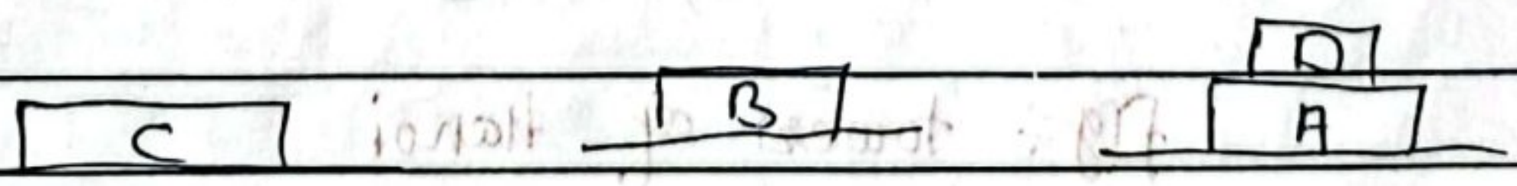
step 8 :



Step 9 :

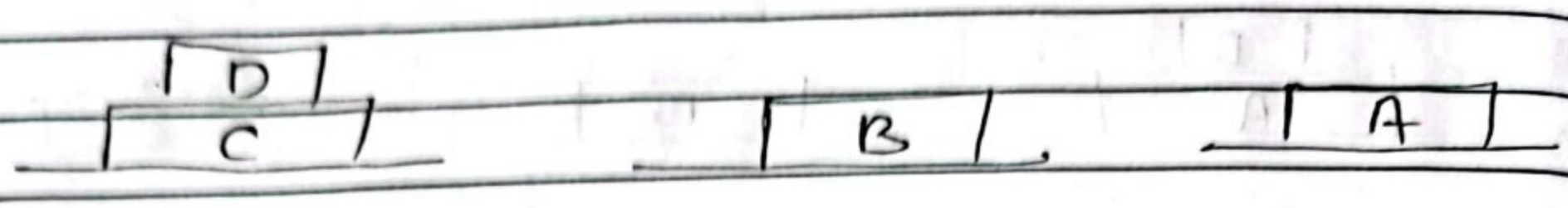


Step 10 :

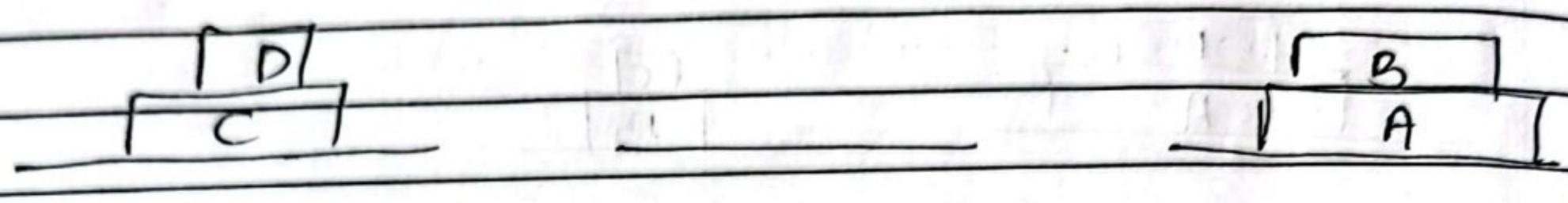


Date
Page

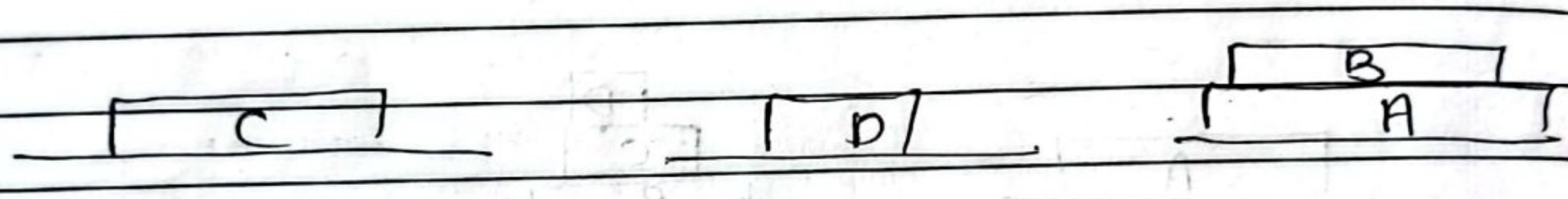
Step 11 :



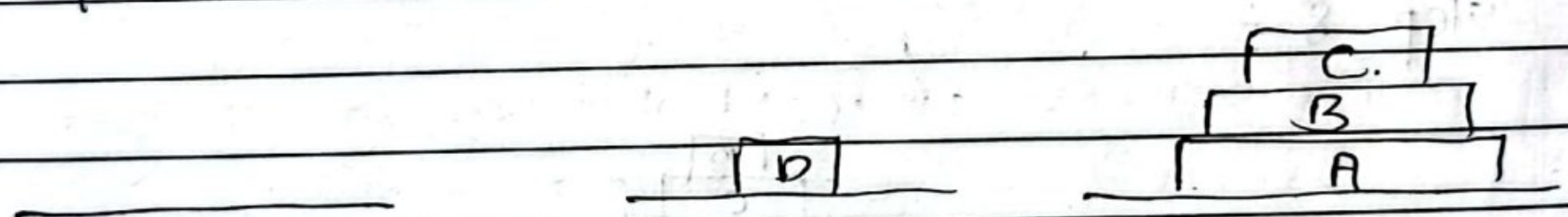
Step 12 :



Step 13



Step 14 :



Step 15 :-

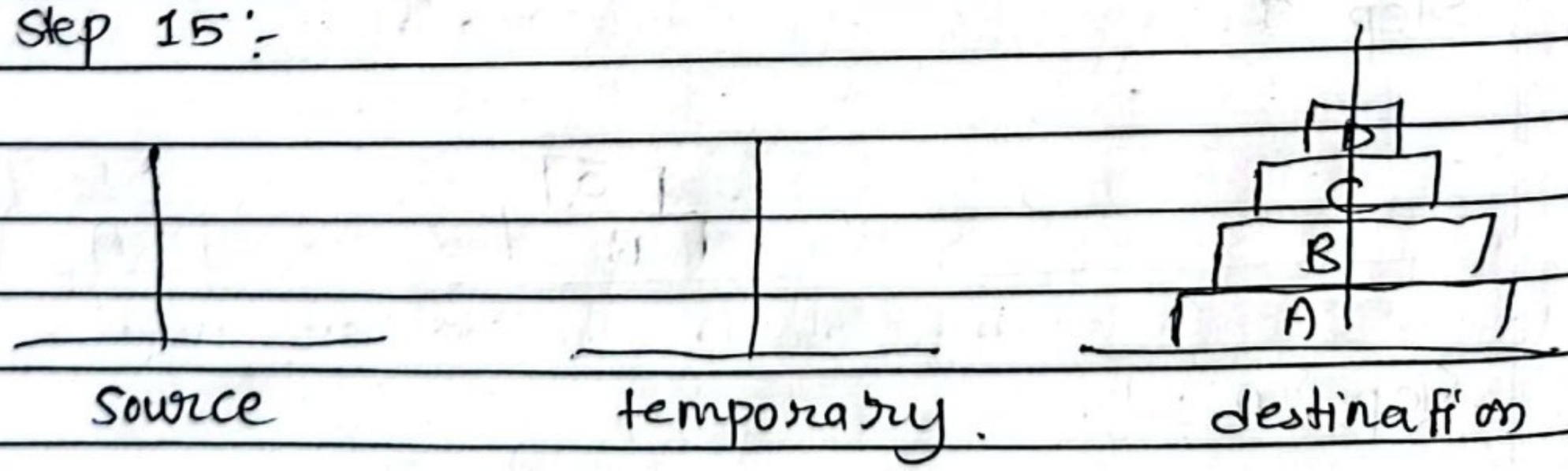


fig : tower of Hanoi

CamScanner

Date
Page

Recursion

→ Recursion is a powerful technique to implement the respective logic.

→ Instead of writing the same instruction or logic again and again, we can define a function with the repeated instruction and call that function again and again until the criteria is meet.

→ The recursive logic must have two parts :-

ⓐ Base criteria

ⓑ Inductive part

→ Base criteria ensures that after executing any finite number of time the recursive function will terminate or stops. Also called stopping or terminating criteria.

→ Inductive part is concerned with calling the function again and again until the base criteria is meet.

→ After every function call, the inductive step must gurantee that the function is converging towards the base criteria so that the recursive call will terminate after finite number of call.

→ Eg. Sum of natural number can be recursively calculated as.

$F(1) = 1$

$F(2) = F(1) + 2$

CS CamScanner

$F(3) = F(2) + 3$
$F(4) = F(3) + 4.$
$\vdots$
$F(n) = F(n-1) + n$

$$\therefore f(n) = \begin{cases} 1 & \text{if } n == 1 \quad \text{[base criteria]} \\ f(n-1) + n & \text{otherwise} \quad \text{[inductive step]} \end{cases}$$

Types of Recursion,

(i) Direct and in-direct Recursion
(ii) Tailed and non-tailed Recursion
(iii) Nested Recurssion
(iv) ~~Executive~~ Recurssion
Excessive

Direct and in-direct recursion,

→ A recursion is said to be direct recursion if a function calls itself.

e.g.
```
int function1 ()
{
    function 1 ()
}
```

→ A function is said to be indirect recussion if it contains a function call to another function which ultimately calls itself.

e.g.
```
int function1 ()
{
```

```
function 2( );
}{
int function 2 ()
{
function 1();
}
```

Tailed and non-tailed recursion

→ A recursion is said to be tailed recursion if there are no pending operation to be performed on return from the function call.

→ In case of non-tailed recursion, there will be some pending operations until the final result after recursive function call.

E.g.

non-tailed

```
int factorial (int n)
{
    if (n = = 0)
    return 1;
else return (n* factorial (n-1));
}
```

Algorithm - 1

tailed

```
int factorial (int n, result f)
{
    if (n = = 0)
    return f;
else return factorial (n-1, n*f);
}
```

Algorithm - 2

→ Algorithm-1, is non tailed recursion as there is a pending operation i.e multiplication that will be executed after the function call.

Algorithm-2 is a tailed recursion as there are no pending operations after the function call.

(iii) Nested recursion

→ In case of nested recursion, the function not only calls itself but also the parameter of function is also a function

Eg

$$H(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ H(2 + H(2n)) & \text{if } n \leq 4 \end{cases}$$

→ In above function H(n), there is a function that not only ~~contains~~ calls itself but has a parameter which is a function itself so it is a nested recursion.

Calculate : H(2), H(1) & H(3).

$$\begin{aligned} H(2) &= H(2 + H(2 \times 2)) \\ &= H(2 + H(4)) \\ &= H(2 + H(2 + H(2 \times 4))) \\ &= H(2 + H(2 + H(8))) \\ &= H(2 + H(2 + 8)) \\ &= H(2 + H(10)) \\ &= H(2 + 10) \\ &= H(12) \\ &= 12 \end{aligned}$$

$$\begin{aligned} H(1) &= H(2 + H(2 \times 1)) \\ &= H(2 + H(2)) \\ &= H(2 + H(2 + H(4))) \\ &= H(2 + H(2 + H(2 + H(8)))) \\ &= H(2 + H(2 + H(2 + 8))) \end{aligned}$$

CS CamScanner
CS CamScanner

$= H(2 + H(2 + H(10)))$

$= H(2 + H(2+10))$

$= H(2 + H(12))$

$= H(2 + 12)$

$= 14$

$H(3) = H(2 + H(2 \times 3))$

$= H(2 + H(6))$

$= H(2+6)$

$= H(8)$

$= 8$

(iv) Excessive recursion :

→ A recursion is said to be excessive recursion if the price of using recursion is slowing down the execution time and storing the value in the memory which may not require in the non-recursive approach.

→ Sometimes the excessive recursion may run out of stack (i.e no space for storing the values from each function call in the memory).

→ Eg. Generation of the fibonacci series

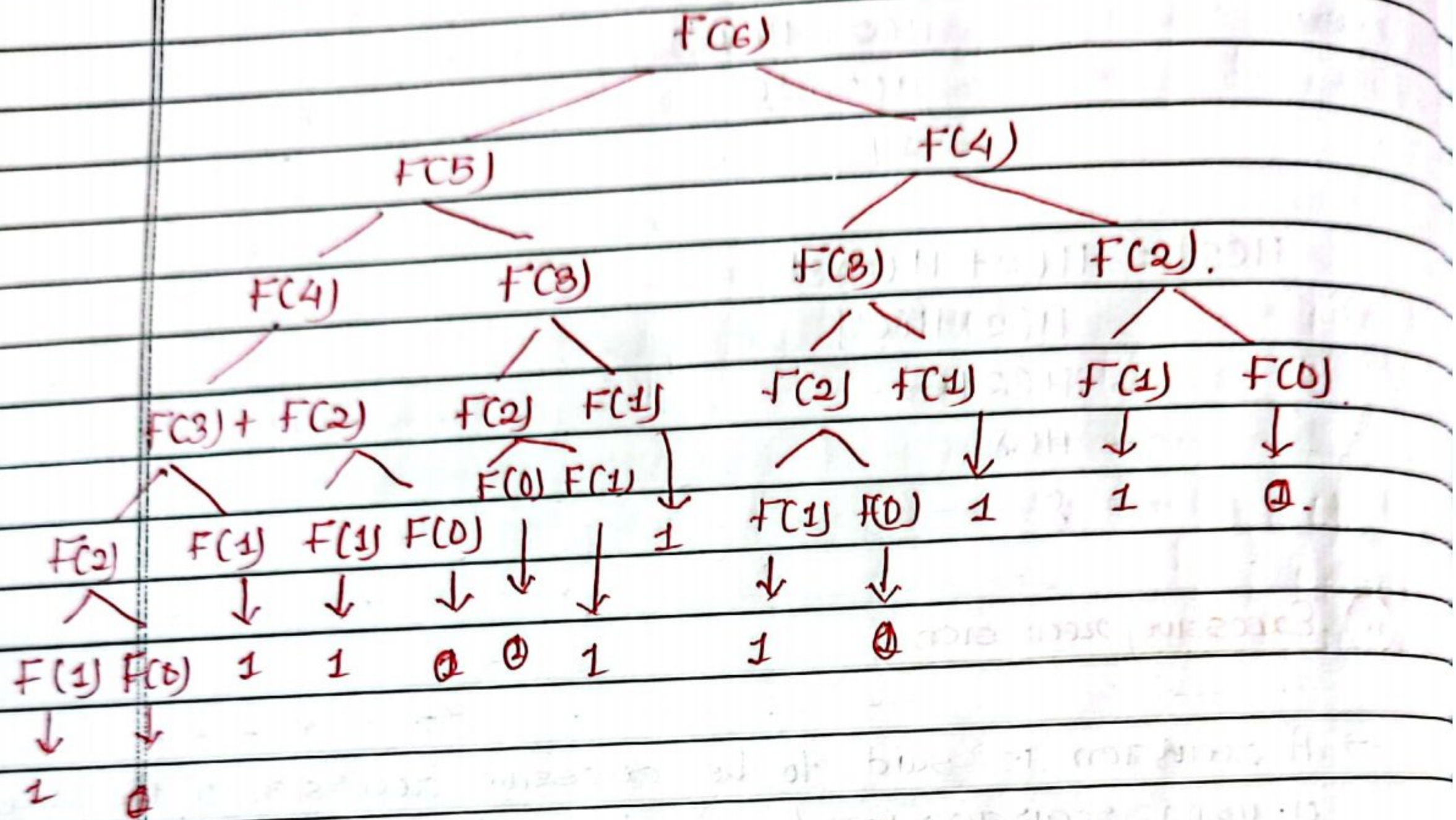
The fibonacci series is given by

0, 1, 1, 2, 8, 5, 8, 13, 21 ---

$$f(n) = \begin{cases} n & \text{if } n < 2 \\ f(n-1) + f(n-2) & \text{otherwise.} \end{cases}$$

CamScanner

for f(6)


F(6)
F(5)
F(4)
F(4)
F(3)
F(3)
F(2)
F(2)
F(1)
F(2)
F(1)
F(1)
F(0)
F(0)
F(1)
F(1)
F(0)
F(1)
F(0)
F(1)
F(0)
1
1
1

→ It is a classical game problem where there are 'n' number of disk maintained on peg (a) (source) and there are other two pegs, peg (b) (temporary) and Peg(c) (Destination).

→ In peg a disk are arranged so that the disk having the bigger diameter is at the bottom and above that disk, the disk having less diameter is placed.

→ Our task is to arrange the disk as shown in ~~fig~~ intial state to the final states.

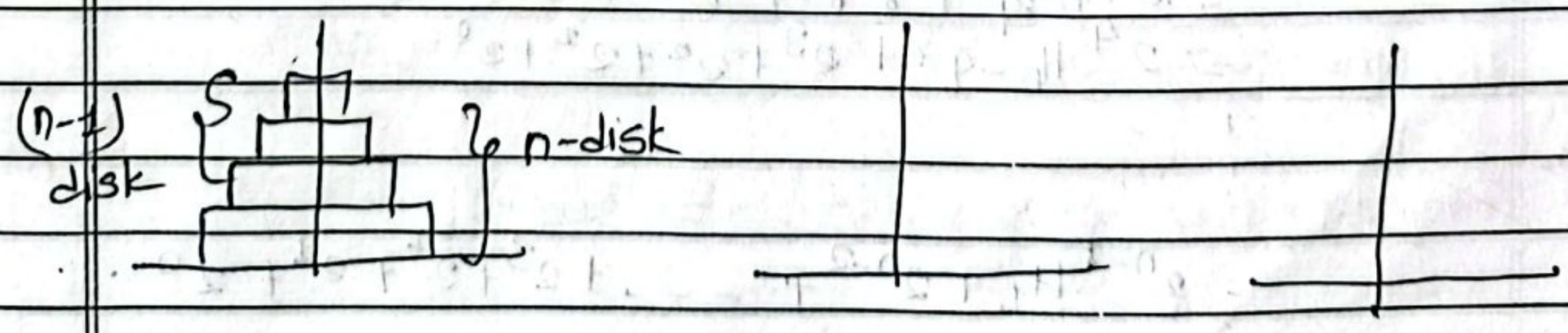
→ Some of the constraints (rules) to perform the task are:

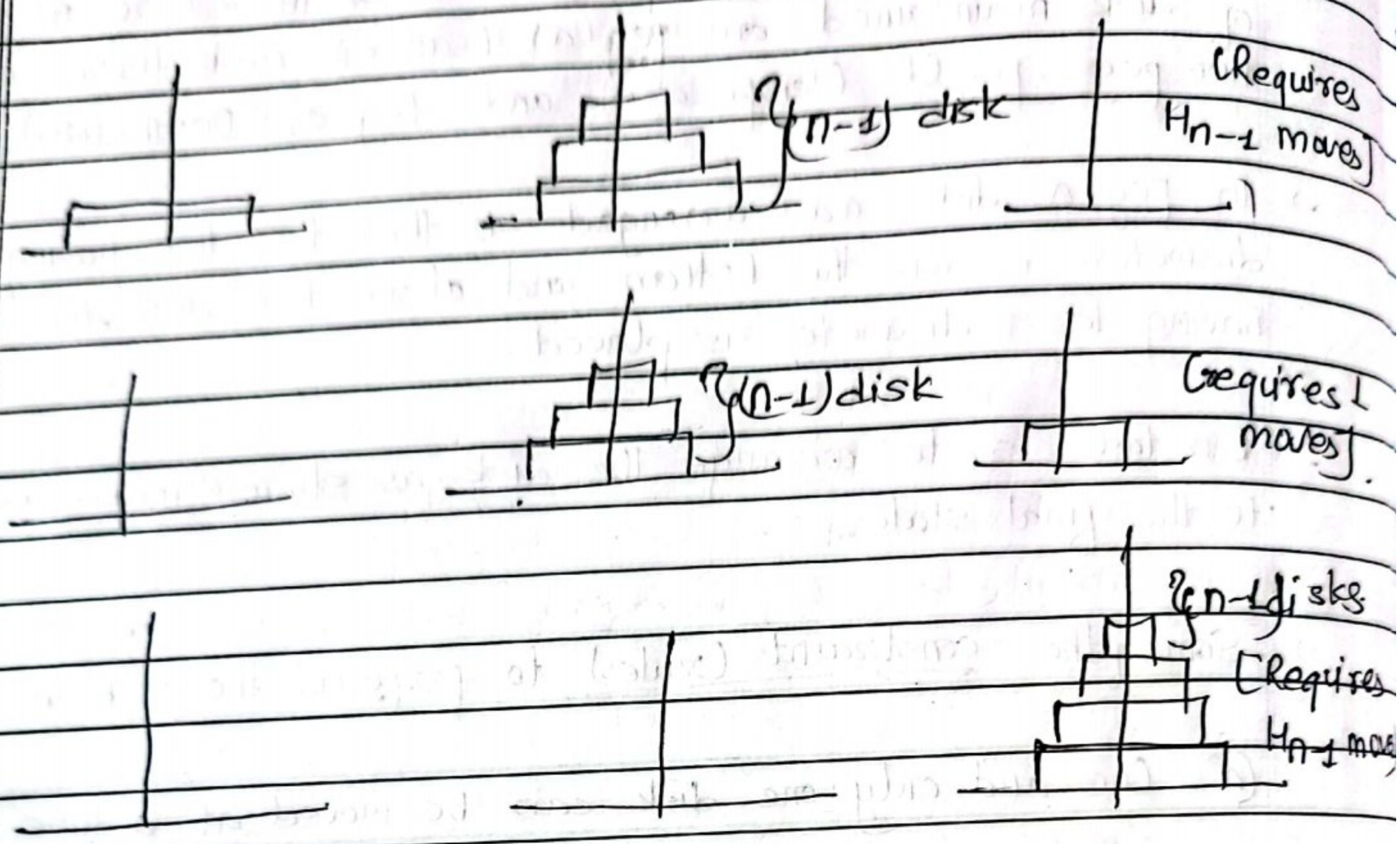
(a) One and only one disk can be moved at a time.

(b) we can only arrange the disk with lesser diameter above the disk with greater diameter.

(c) we can have only one extra peg (temporary) other than source and destination.

→ let H_n be the total number of moves required to move 'n' disk and H_{n-1} be the number of move required to move $(n-1)$ disks.





→ Total number of move required can be written as

$$
\begin{aligned}
H_n &= H_{n-1} + 1 + H_{n-1} \\
&= 2H_{n-1} + 1 \\
&= 2[2H_{n-2} + 1] + 1 \\
&= 4H_{n-2} + 2 + 1 \\
&= 4[2H_{n-3} + 1] + 2 + 1 \\
&= 8H_{n-3} + 4 + 2 + 1 \\
&= 8[2H_{n-4} + 1] + 4 + 2 + 1 \\
&= 16H_{n-4} + 8 + 4 + 2 + 1 \\
&= 2^4 H_{n-4} + 2^3 + 2^2 + 2^1 + 2^0 \\
&\vdots \\
&= 2^{n-1} H_1 + 2^{n-2} + \cdots + 2^3 + 2^2 + 2^1 + 2^0
\end{aligned}
$$

CamScanner

$$H_n = 2^{n-1} + 2^{n-2} + \dots + 2^3 + 2^2 + 2^1 + 2^0$$

$[\because H_1 = 1]$.

which is the geometric series with common ratio $(r) = 2$ and first term $(a) = 1$.

The sum of geometric series is given by:

$$S_n = \frac{a[r^n - 1]}{r-1}$$

$$\text{or, } H_n = \frac{1[2^n - 1]}{2-1}$$

$$\therefore \boxed{H_n = 2^n - 1}$$

which is the required moves.

Algorithm for TOH problem

Assumption

let us consider we have 'n' disk and three tower named source, temporary and destination.

Algorithm

Step 1: Check number of disk

if $(n = 1)$

move the disk from source to destination and exit.

Step 2: Move upper $(n-1)$ disk from source to temporary using destination as temporary.

Move $(n=1$, source, destination, temporary$)$.

Step 3: Move largest disk from source to destination

Step 4: Move upper $(n-1)$ disk from temporary to destina-

tion using source as temporary.
Move (n-1, temporary, source, destination)

Step 5: Stop

Simpler Iteration Solution

(a) For even number of disk
* Make legal move from one source to temporary tower.
* Make legal move from source to destination tower.
* Make legal move from temporation. to destination.
* Repeat until complete.

(b) For odd number of disk
* Make legal move from source to destination tower.
* Make legal move from source to temporary tower.
* Make legal move from destination to temporary tower.
* Repeat until complete.

Greatest common Divisor (GCD) using recursion

Assumption
let a and b are two numbers whose Greatest common Divisor (GCD) is to be computed

Algorithm

Step 1: Check if a and b are equal.
if (a == b)
return 'a' as GCD and exit.

Date
Page

Step 2 : Compute the modulus among a and b.

Step 3 : Check if (a>b)
return gcd (a-b,b)
else,
return gcd (a, b-a)

Step 4 : Stop

Recursive Algorithm for factorial calculation.

Step 1 : Check if (n==0)
return 1
else return (n* fact (n-1)

CamScanner
CamScanner

Date
Page

Unit 8:
Queue and linked list

Queue:

→ like stack, Queue is also a linear data structure but the data are inserted and deleted from two different end points.

→ The end from where data is inserted is called "rear" end and the end from where data are deleted is called "frontend.

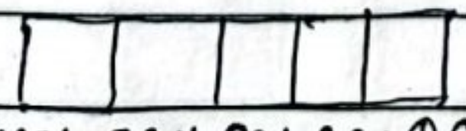
→ It is also called FIFO (First in First out) data structure as the first element inserted will be the first element to be deleted.

→ To insert the data we increment the "rear" index and for the deletion we increment the "front" index.

→ Basic operations performed in queue are
(i) Enqueue
(ii) Dequeue

→ Insertion of element from "rear" end is called enqueue operation and deletion of ~~op~~ element from queue is dequeue operation.

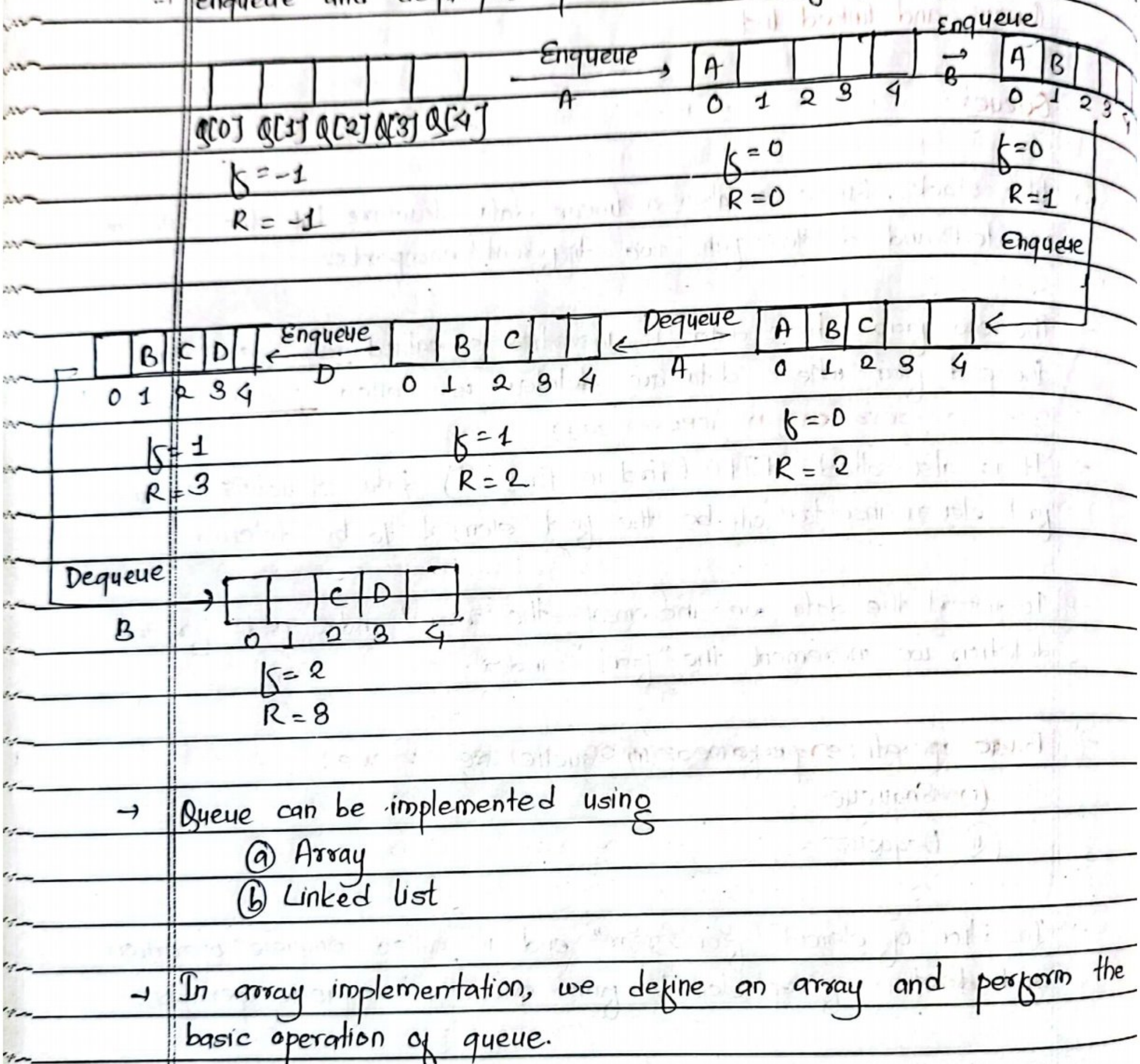
→ Initially the "front" and "rear" values are set to -1 and are incremented while dequeue and enqueue operation respectively.

→ 

front = -1, rear = -1.

CamScanner

→ Enqueue and dequeque operation can be graphically shown as



→ Queue can be implemented using
- (a) Array
- (b) Linked list

→ In array implementation, we define an array and perform the basic operation of queue.

→ Types of queue
- (a) Linear queue
- (b) Circular queue
- (c) Priority queue
- (d) Double ended queue.

CamScanner

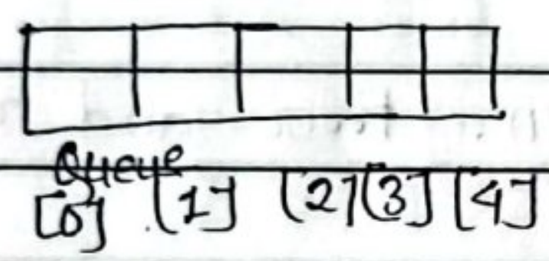
(a) Linear Queue

→ let Queue [n] be the array of size 'n'. Also, let two index front and rear for tracking, deletion and insertion process respectively in the queue.

Initially front and rear are both said to be -1.

Queue is said to be full if "rear" point the maximum size of array and is said to be empty if rear and front equals to -1.

Enqueue operation in linear queue



front = -1

rear = -1

Algorithm for enqueue operation in linear queue.

Assumption

let Queue[n] be an array of size 'n'. Front and rear are two index set to -1 initially.

Algorithm

Step 1: Check for queue full

if (rear = n-1)

Print queue full and return

Step 2: If (rear = front = -1).

set front = 0

Date
Page

rear = 0
else increment rear by 1.
rear = rear + 1

Step 3 :- Read element from user to insert in queue as "data".

Step 4 : Insert element in queue pointed by rear index.
Queue [rear] = data

Step 5 : Stop

Algorithm for Dequeue operation in linear queue

Assumption

let Queue [n] be array of size 'n'. Front and rear are two index set to -1 initially

Algorithm

Step 1 : Check for queue empty
if (front = rear = -1)
Print queue empty and return

Step 2 : ~~else~~ Delete element from queue pointed by front index as data.
data = ~~da~~Queue [front].

Step 3 : Increment front by 1.
front = front + 1

Step 4 : Stop

CamScanner
CamScanner

Date
Page

Algorithm for display operation in linear Queue.

Assumption

let Queue [n] be array of size 'n'. front and rear are two index set to -1 initially.

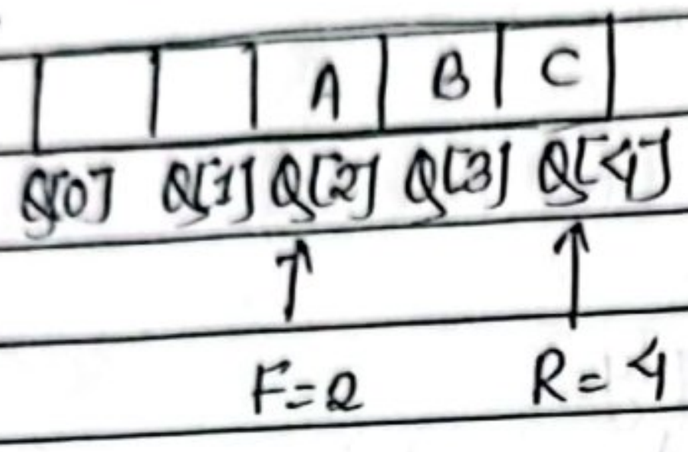
Algorithm

Step 1: Check for queue empty
if (front = rear = -1)
Print queue empty and return.

CamScanner

Circular Queue

→ The main problem with the linear queue is that the queue becomes full even though there are empty space in the queue.



→ In the above queue, it is said to be full as rear points to the maximum size of array, but there are empty space in the queue so there is the wastage of memory.

→ It is because the value of front and rear are always increamented but never decremented.

→ To avoid the problem associated with linear queue modification is made that introduces circular queue.

→ In circular queue elements are arranged in circular fashion i.e if the front and rear are at the maximum size of array then it is arranged to point the very first location of the array.

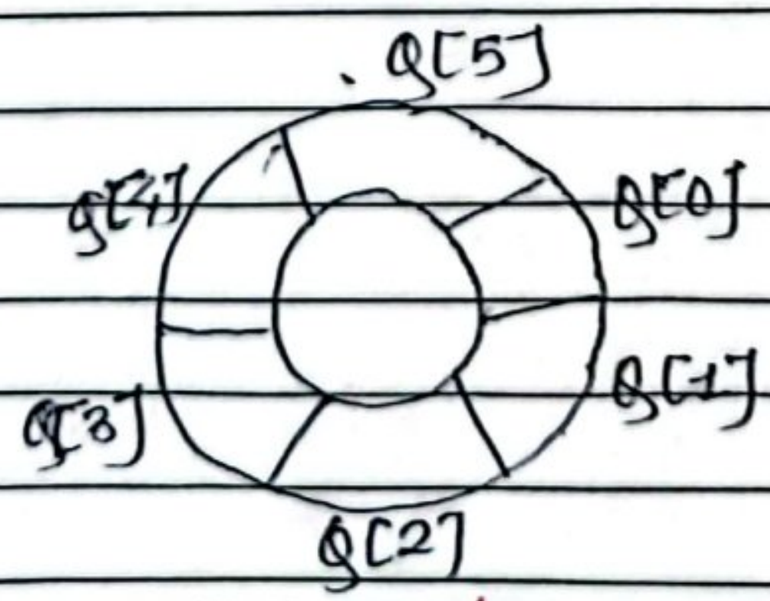


fig: circular queue

→ In the above queue, the element are arranged in circular manner as a[n-1] follows Q[0].

→ Basic operation in circular queue

ⓐ Enqueue

ⓑ Dequeue

→ In the enqueue operation the value of rear is updated by an expression.

rear = (rear +1) % max size of array

→ In dequeue operation the value of front is updated by :

front = (front +1) % max size of array.

Algorithm for enqueue operation in circular queue.

Assumption

let Queue [n] be the circular queue of size 'n'. Also let front and rear are two index set to -1 initially.

Algorithm →

Step 1: Check for queue full.

if(front = 0 and rear = n-1) or front = rear + 1))

Print queue full and return.

Step 2: Check if (front = rear - 1)

set front = 0

rear = 0

else,

update rear using expression
rear = (rear + 1) % n

Step 8: Read element from user as 'data'.

Step 9: Insert the element in the queue pointed by rear index
Queue [rear] = data

Step 5: Stop.

Algorithm for dequeue operation in circular queue.

Assumption

let Queue[n] be the circular queue of size 'n'. Also let front and rear are two index set to -1 initially.

Algorithm

Step 1 : Check for queue empty
if (front = rear = -1)
Print queue empty and return

Step 2: Delete element from queue pointed by front index as 'data'.
data = Queue [front]

Step 3:- Check if (front = rear) // Queue has only one element
set front = -1
rear = -1
else update front using expression
front = (front + 1) % n

Algorithm to display

Step 4: Stop

Double Ended Queue (Deque)

→ In normal queue the insertion and deletion is done from two different ends i.e rear and front respectively.

→ In case of double ended queue (deque) insertion and deletion is done from the both ends.

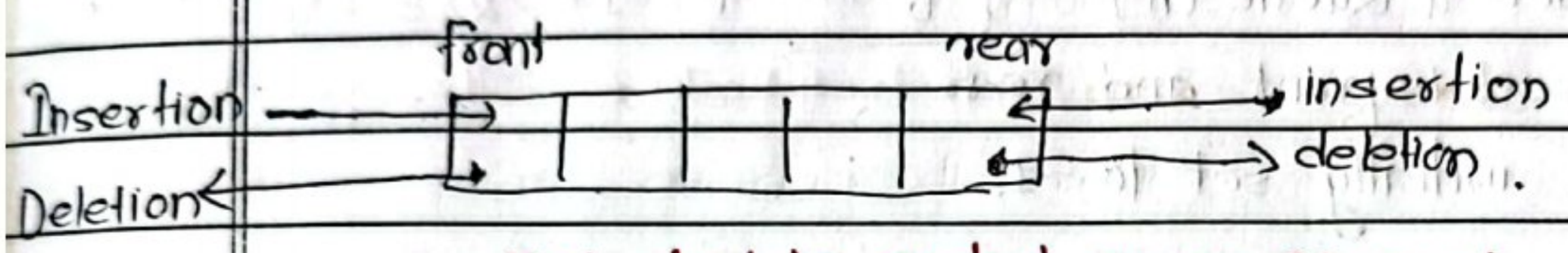


fig: double ended queue (Deque)

→ Basic operation that can be performed on double ended queue are:

(a) Insertion from front end
(b) Insertion from rear end
(c) Deletion from front end
(d) Deletion from rear end

→ Types of double ended queue

(a) Input / Insertion restricted dequeue
(b) Output / Deletion restricted dequeue.

→ In input restricted deque deletion can be done from both end but insertion can be done from only one end.

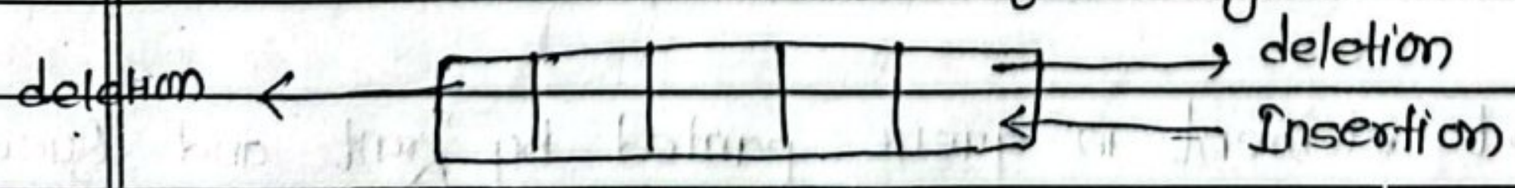
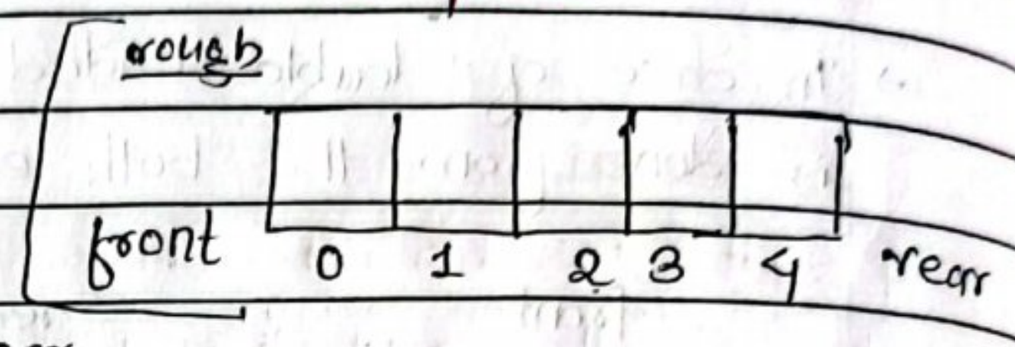


fig: Input restricted dequeue

→ In output restricted dequeue deletion can be done from only one end but insertion can be done from both ends.

Fig: output restricted dequeue

Algorithm to insert from front end in Dequeue.



Assumption

let us consider a circular queue Queue [n] of size n. Also let front and rear are two index initially set to -1.

Algorithm

Step 1: Check for queue full
if ((front = 0) and rear = n-1) or front = rear + 1)
Print queue full and return

Step 2: Check if (front = rear = -1)
set front = 0
rear = 0
else if (front = 0), update front as
front = N-1
else update front as
front = front - 1

Step 3: Read element to insert in queue as "data".

Step 4: Insert element in queue painted by front and Queue[front] = data

Date
Page

Step 5: Stop.

Algorithm to delete from rear end.

Assumption

let us consider a circular queue, Queue[n] of size 'n'. Also let front and rear are two index initially set to -1.

Algorithm

Step 1: Check for queue empty

if (front = rear = -1)

Print queue empty and return.

Step 2: Delete element from queue pointed by rear end as "data".

data = Queue[rear]

Step 3: Check if (front = rear) // only one element in queue

set front = -1

rear = -1

else if (rear = 0)

set rear = n-1

else update rear as

rear = rear - 1

Step 4: Stop

CamScanner

Date
Page

Priority queue :

→ In normal queue the elements are processed on the basic of first come first serve.

→ But in the real world problem we may require to alter the normal principle of queue while processing element managed in queue.

→ In such case we access the element based on their priority rather than their arrival time, which introduces the priority queue.

→ In priority queue, element are inserted in the queue along with their priority value and are deleted according to their priority value

→ If the element has arrived late but has the greatest priority then we first process that element.

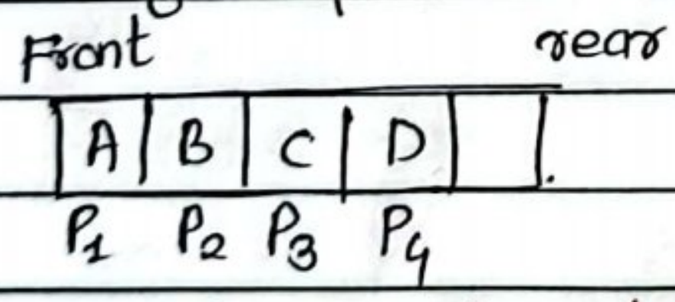


fig:- Queue elements along their priorities

→ Element are inserted from rear and but deletion is made according their priority value.

→ The rule for deleting or processing according to priority value is

ⓐ If two elements have the same priority, then processing

is done on FCFS basis.

(b) If the two element has priority P_1 and P_2 and $P_2 > P_1$ then the element with priority P_1 will be processed first even though that element entered late in the queue.

→ Types of priority queue

(a) Ascending priority queue → a/c to increasing order of priority

(b) Descending priority queue. " " decreasing " " "

CamScanner

Date
Page

Static and dynamic list

→ In static memory allocation we define the memory prior to execution which leads in the wastage of memory.

→ when memory is allocated at the compile time, we cannot grow or shrink the size of memory i.e there will be fixed memory.

→ If the memory allocated is large enough and ~~alloca~~ the value to the stored are less and if the memory allocated is small but we need to store large data in both of the case. There is no optimal utilization of memory.

→ To overcome such problem associated with memory utilization dynamic memory allocation is used

→ In dynamic memory allocation we can grow the memory size when needed and can free memory when not in use which results in optimal utilization of memory.

→ In dynamic memory allocation the memory is allocated and de-allocated during run time.

→ List are the linear data structure where element are stored in continuous manner and element can be inserted and deleted from any valid positions

→ List can be implemented
ⓐ Statically ⓑ Dynamically

CamScanner

Dynamic implementation of list

→ In the dynamic implementation of list, the element are represented using a special structure called node.

→ Basically, node is divided into two parts:

a) info
b) next.

→ The info part holds the exact value to be stored by a node. It is also called value or data part.

→ The next part of node will hold the adress of the next node to which it is linked with. It is also called pointer, link, etc.

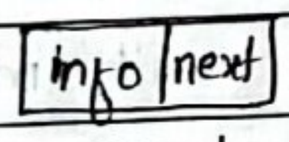


fig: Simple node

→ The interconnection of such node leads to the data structure called linked list.

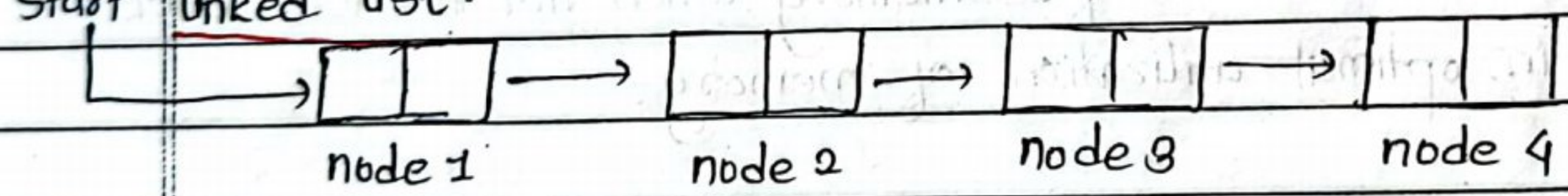


fig: linked list

→ The linked list is the linear collection of nodes where nodes are connected with each other through links i.e the each nodes holds the address of next node which it point to.

→ To access the element of the linked list we use the external pointer variable called START which holds the address of the

very first node in the list.

→ Linkedlist is said to be empty if START = NULL

→ Also the next part of the last node holds NULL i.e doesn't point to any other node in the linked list.

→ Types of linked list

ⓐ Singly linked list
ⓑ Doubly linked list
ⓒ Circularly singly linked list.
ⓓ Circularly doubly linked list.

ⓐ Singly linked list

→ In singly linked list, the each node has single links, that points to next node in the list.

→ Basic operation in singly linked list are:

ⓐ Insertion of a node in linked list
ⓑ Deletion of node from linked list
ⓒ Traverse (Travel) in the linked list.

Insertion of a node in linked list

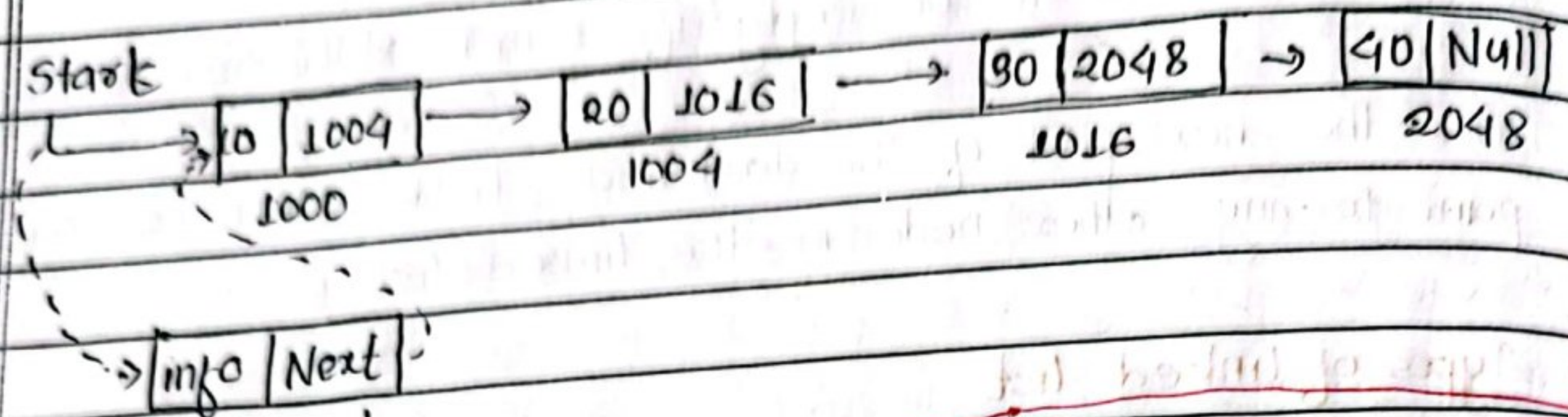
→ Allocating a memory for a new node and linking that node to the list is called insertion operation.

→ Nodes can be inserted in the linked list at

ⓐ Beginning of singly linked list.
ⓑ End of singly linked list.
ⓒ Specific position of singly linked list.

CamScanner

Algorithm to insert a node at the begining of singly linked li




```
Struct node
{
  int info
  node * next
};
struct node *newnode,*Temp,*Tem
```

self referential structure for no and pointer variable that hold address of node

Assumption

let START be the external pointer variable that holds the adress of the first node of the linked list. linked list is said to be empty if START = NULL. Also newnode, Temp, Temp 1 are the pointer variable that can hold the address of a node.

Algorithm

Step 1: Create a new node

newnode = (struct node*) malloc (size of (struct node));

Step 2: Read value from the user as "data" and put in info field of new node.

newnode → info = data

Step 3: set the next part of new node to START

newnode → next = START

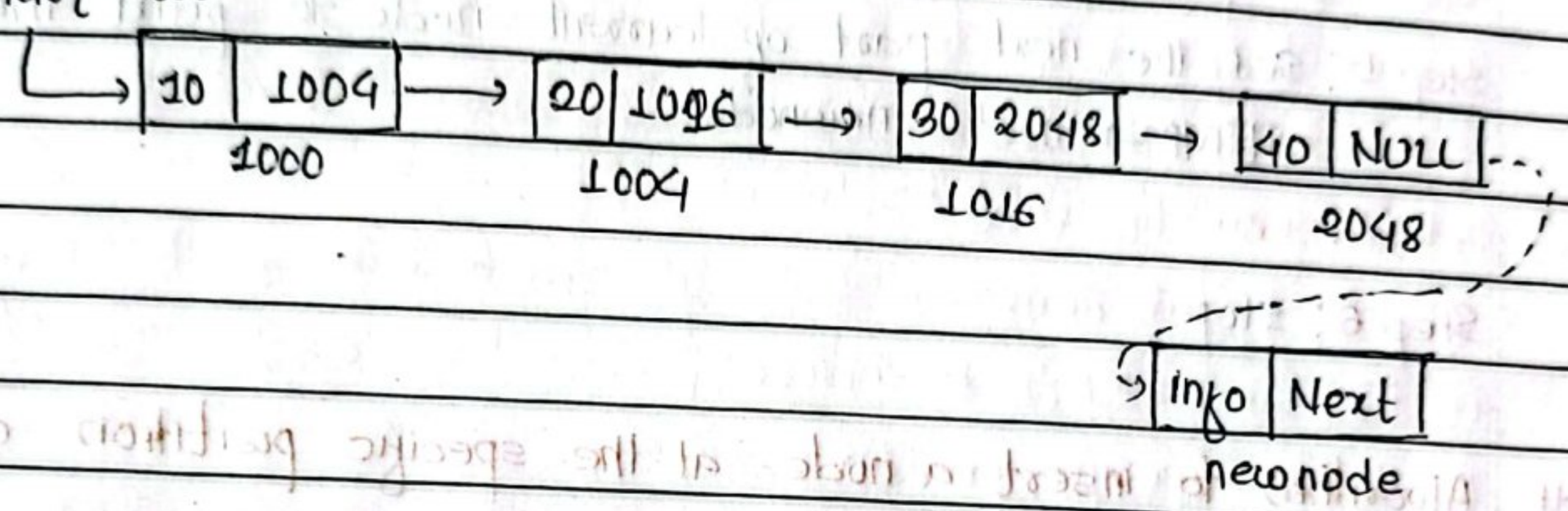
Date
Page

Step 4: Update START to point newnode.

START = newnode

Step 5: Stop.

Algorithm to insert a node at the end of singly linked list.



Algorithm

Step 1: Create a newnode

newnode = (struct node*) malloc (size of (struct node)),

Step 2: Read value from user as "data" and put in info field of new node.

newnode → info = data.

Step 3: Set the next part of newnode to NULL.

newnode → next = NULL

Step 4: Check if linked list is empty.

if (START = NULL)

set START = newnode and exit

else,

CamScanner

Travel in the linked list until end is reached

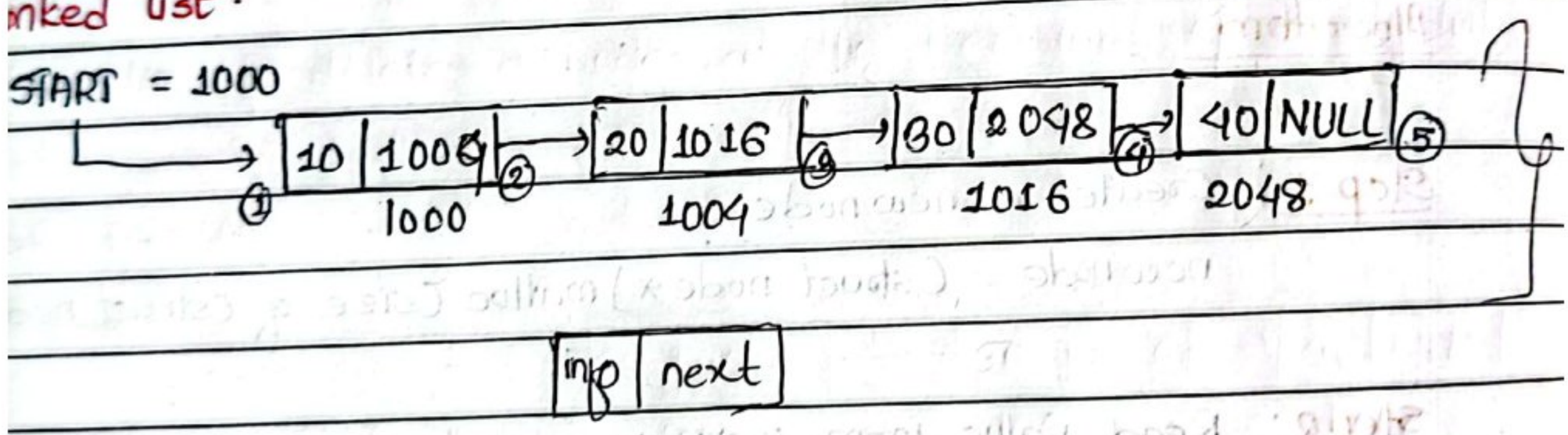
```
TEMP = START
while( TEMP -> next != NULL)
{
    TEMP = TEMP -> next
}
```

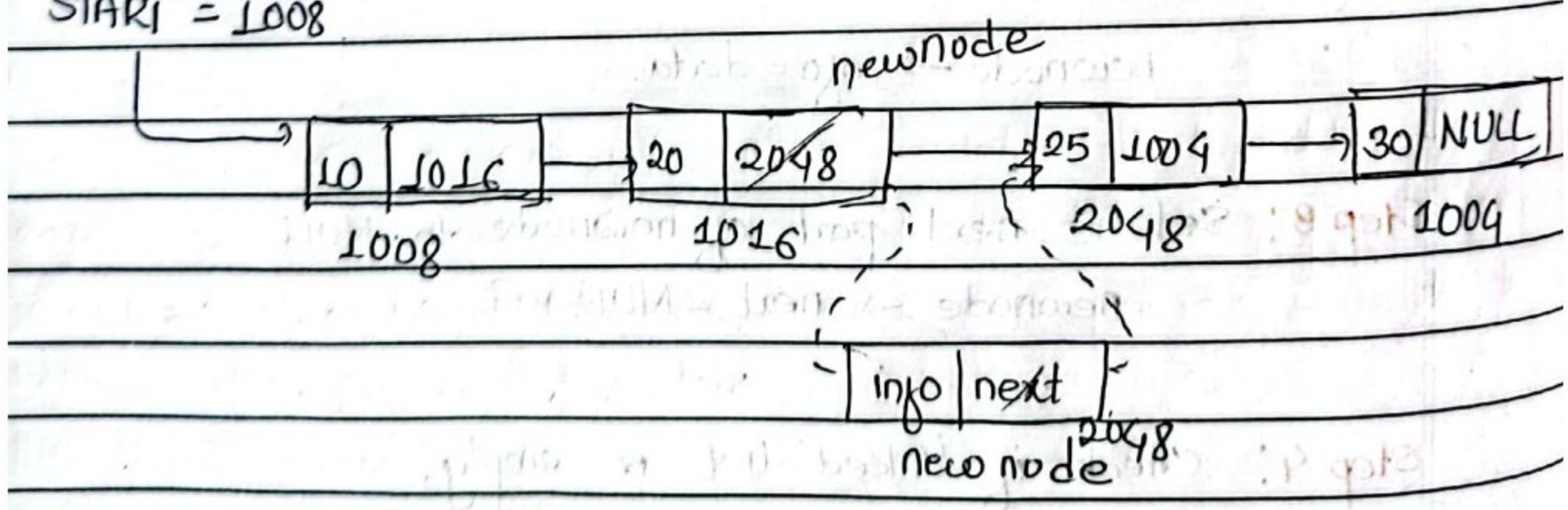
Step 5: Set the next part of current node to point the newnode

```
TEMP -> next = newnode
```

Step 6: Stop

Algorithm to insert a node at the specific position of singly linked list.





Algorithm :-

Date
Page

Step 1: Read position from user to insert a node as "pos".

Step 2: Check if the position is valid or not.

```
count = 0, TEMP = START
while (count < pos - 1)
{
temp TEMP = TEMP → next
if (TEMP = NULL)
Print invalid position and exit.
else,
   count ++
}
```

Step 3: Create a newnode

```
newnode = (struct node *) malloc (sizeof (struct node)).
```

Step 4: Read data from user and put it in the info field of newnode.

```
newnode → info = data
```

Step 5: set the next part of newnode to point the next part of current node.

```
newnode → next = TEMP → next
```

Step 6: Set the next part of current node to point newnode

```
TEMP → next = newnode
```

Step 7: Stop.

CamScanner
CamScanner

Delete operation in singly linked list

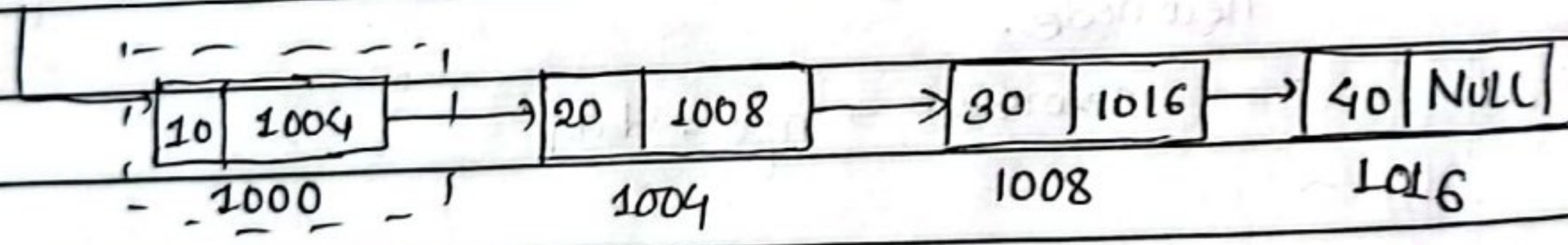
→ The process of removing node from the linked list as well as de-allocating memory occupied by that node is called deletion of node.

→ we use "free" function to release the memory occupied by deleted node in the linked list.

→ Deletion can be done:

(a) At the beginning
(b) At the end.
(c) At the specific position.

Algorithm to delete a node at the beginning of singly linked list.



Assumption

let START be the external pointer variable that holds the adress of first node. List is said to be empty whenever START = NULL. Also TEMP and TEMP1 are pointer variable to hold address of a node. "Free" function is used to release the memory occupied by any node in linked list.

Algorithm:

Step 1: check for the list empty.

if (START = NULL)

Print list empty and exit.

Step 2: Store the address of first node in temporary variable.

TEMP = START

Step 3: Update start to print the second node in linked list.

~~fre~~ START = START → next

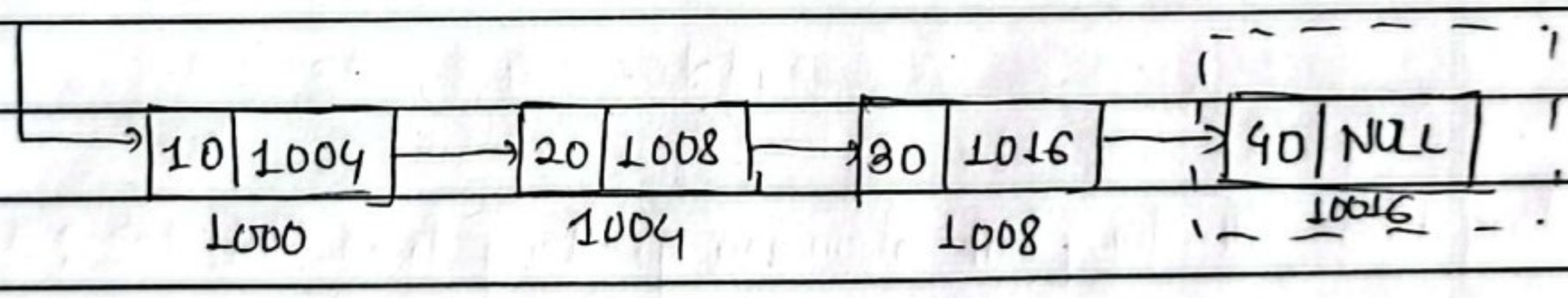
Step 4: Release the memory accepted by deleted node.

free(TEMP)

Step 5: Stop.

Algorithm to delete a node at the end of singly linked list.

START = 1000



Algorithm

Step 1: Check for list empty.

if (START = NULL)

print list empty and exit.

Step 2: Check if the list has only one node.

if (START → next = NULL)

Release the memory occupied by deleted node and exit
free(TEMP)

else,
Travel in the list until end is reached.
TEMP = START
TEMP1 = START → next
while (TEMP1 → next != NULL)
TEMP = TEMP1
TEMP1 = TEMP1 → next

Step 3: Update the next part of current node to NULL.
TEMP → next = NULL

Step 4: Release the memory occupied by deleted node.
free(TEMP1)

Step 5: Stop.

CamScanner

Date
Page

Basic operations in Doubly linked list

(a) Insertion of node
(b) Deletion of node

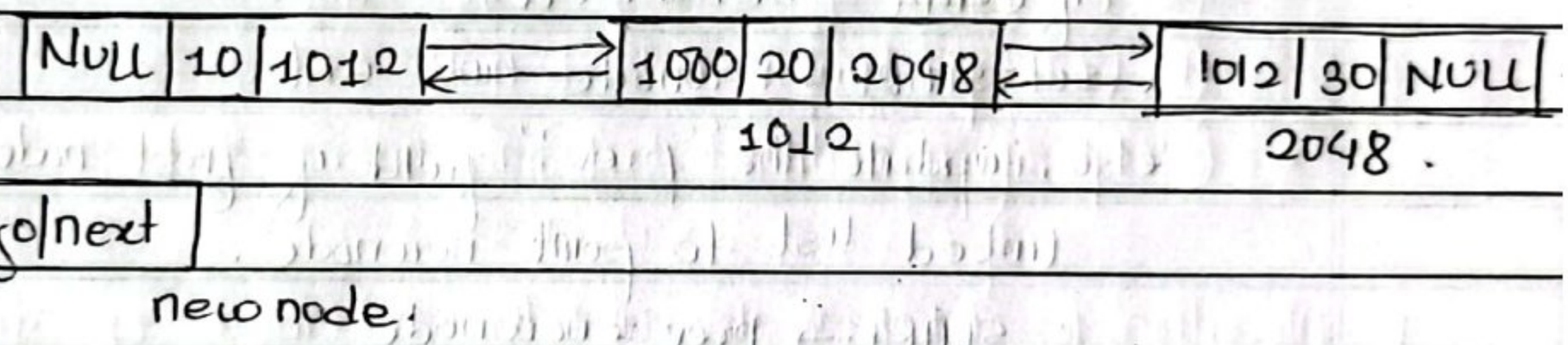
(a) Insertion of node in doubly linked list

→ Insertion can be done at

(a) Begining of doubly linked list
(b) End of doubly linked list
✗ (c) Specific position of doubly linked list.

Algorithm to insert node at the begining of doubly linked list.

START = 1000



Assumption

let START be the external pointer variable that holds the address of first node in doubly linked list. The 'prev' field of first node is NULL and 'next' field at last node is NULL. TEMP, TEMP1. are pointer variable that can hold the address of node. List is said to be empty when START = NULL.

```
struct node
{
int info
node *prev
node *next
};
struct node* newnode,
TEMP,TEMP1
```

self referential structure of node in doubly linked list.

Algorithm

Step 1: Create a newnode

newnode = (struct node *) malloc (size of (struct node))

Step 2: Read the value of from user as "data" and put in the info field of newnode.

newnode → info = data.

Step 3: Set the previous part of newnode to NULL.

newnode → prev = NULL.

Step 4: Set the next part of newnode to point the address of first node of existing linked list.

newnode → ~~nod~~next = START.

Step 5: Check ~~th~~ for list empty.

if (~~start~~ START = NULL)

set START = newnode and exit.

else, update the "prev" field of first node of existing linked list to point newnode.

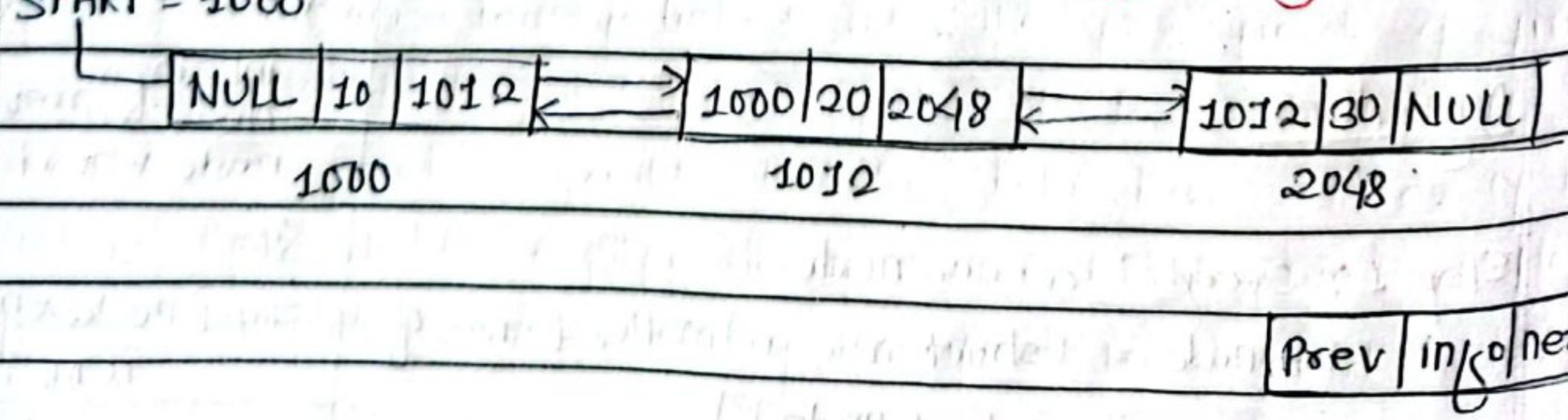
START → prev = newnode

Step 6: Update ~~start~~ START to point the new node.

START = newnode

Step 7: Stop.

Algorithm to insert node at the end of doubly linked list.



Date
Page

Assumption

let START be the external pointer variable that holds the address of first node in doubly linked list. The "prev" field of first node is NULL and "next" field of last node is NULL. TEMP, TEMP1 are pointer variable that can hold the address of node. list is said to be empty when START = NULL.

Algorithm

Step 1: Create a newnode

```
newnode = (struct node *)malloc(size of (struct node))
```

Step 2: Read the value of from user as "data" and put in the info field of newnode.

```
newnode → info = data
```

Step 3: Set the next part of newnode to NULL.

```
newnode → next = NULL
```

Step 4: Check for list empty.

```
if (START = NULL)
    set newnode → prev = NULL
        START = newnode and exit.
```

Step 5: Travel in the list until end is reached.

```
TEMP = START
while (TEMP → next != NULL)
    TEMP = TEMP → next.
```

Step 6: Update the 'next' part of last node of existing linked

CamScanner

list to point newnode.

TEMP → next = new node

Step 7: Update the "prev" part of new node to point the last node of existing linked list.

newnode → prev = TEMP

Step 8: Stop.

Deletion of node from double linked list

→ Deletion can be made from

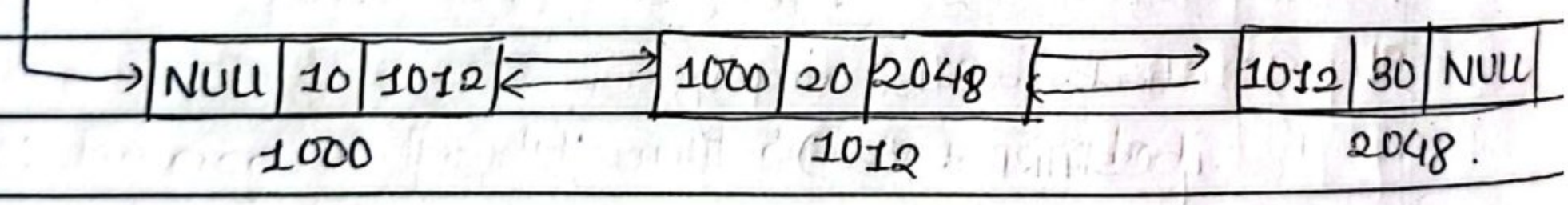
(a) Begining of doubly linked list

(b) End of doubly linked list

x (c) Specific position of doubly linked list.

Algorithm to delete node at the begining of doubly linked list

START = 1000



Assumption

let START be the external pointer variable that holds the address of first node in doubly linked list. The "prev" field of first node is called NULL and "next" field at last node is NULL. TEMP, TEMP1 are pointer variable that can hold the address of node. list is said to be empty when START = NULL. We use free function to release the memory occupied by deleted node.

CamScanner

Date
Page

Algorithm

Step 1: Check for list empty
if (START = NULL)
Print list empty and exit.

Step 2: Hold the address of first node in TEMP.
TEMP = START

Step 3: Check if the list has only one node.
if (START → next = NULL)
set START = NULL
free (TEMP) & exit.

Step 4: Hold the address of second node in TEMP1.
TEMP1 = START → next

Step 5: Update the "prev" part of second node of existing ~~ao~~ list to NULL.
TEMP1 → prev = NULL

Step 6: Update START to point the second node of existing list.
START = TEMP1
or
START = START → next

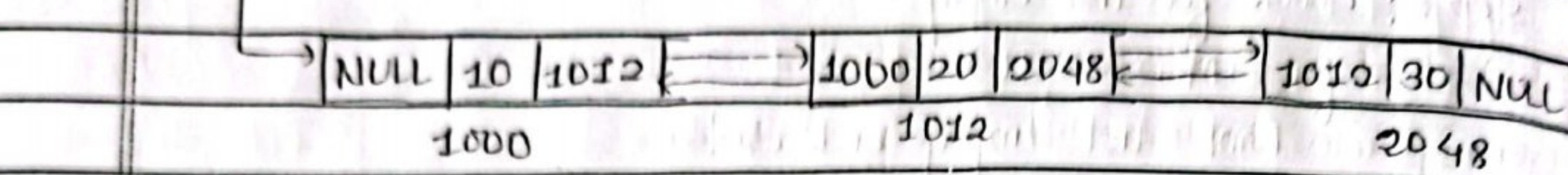
Step 8: Stop.

Step 7: free memory allocated for deleted node.
free (TEMP).

CamScanner

Algorithm to delete node at the end of doubly linked list.

START = 1000



Assumption (same).

Algorithm

Step 1: Check for list empty

if (START = NULL)

print list empty and exit.

Step 2: Check if the list has only one node.

if (START → next = NULL)

TEMP = START

START = NULL

free (TEMP) and exit.

Step 3: Traverse in the list until end is reached.

TEMP = START

TEMP 1 = START → next

while (TEMP1 → next != NULL)

TEMP = TEMP 1

TEMP1 = TEMP1 → next

Step 4: Update the next part of second last node to NULL.

TEMP → next = NULL

CamScanner

Date
Page

Step 5: Free the memory allocated for last node.
Free (TEMP 1)

Step 6: Stop.

Polynomial Representation of linked list.

→ The polynomial equation of form
$Ax^n + Bx^{n-1} + Cx^{n-2} + --- Dx + E$ can be represented with the ~~list~~ linked list having the node as follows.

coefficient	Power of x	next

→ If the polynomial contains the multiple variable (like x, y, z) and the structure of node will be

Power of x	power of y	Power of z	coefficient	next

Qn) Represent the following polynomial with linked list representation.

(a) $P(x) = 3x^4 - 2x^2 + 7x - 8$

(b) $P(x,y,z) = 7x^4y^3z^2 + 4x^3yz + 3x^2 + 8$

for (a) START

3	4	→	-2	2	→	7	1	→	-8	0	NULL

for (b)

START

4	3	2	7	→	3	1	1	4	→	2	0	0	-3	→	0	0	0	8	NULL

CamScanner

Linked list implementation of stack

→ Stack can also be implemented using linked list.

→ To implement stack with linked list we create a linked list and perform the basic operation of stack (i.e Push and pop) on that linked list.

→ In stack, the basic operation is performed on the reference of 'tos' (top of stack).

→ So we define the external pointer variable named 'tos' tos on linked list and perform those basic operation of stack.

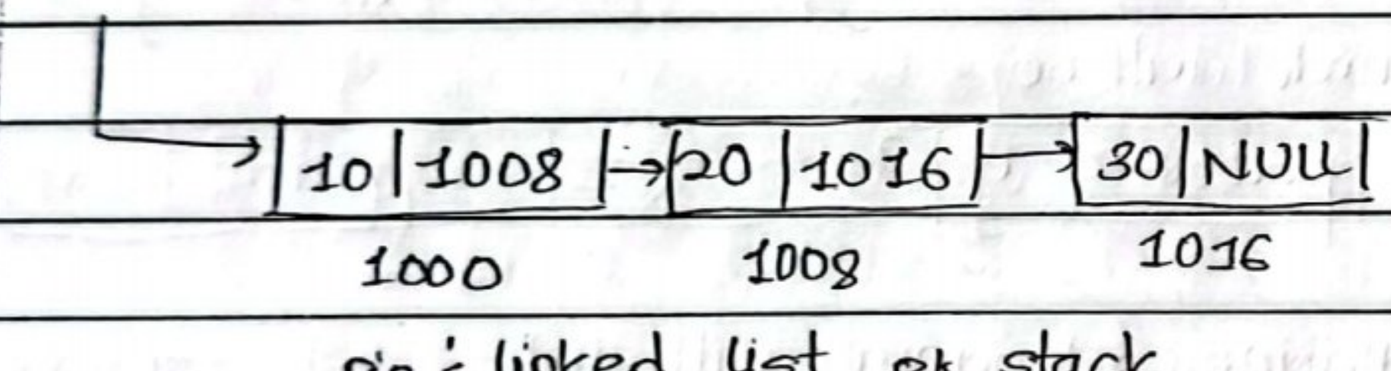
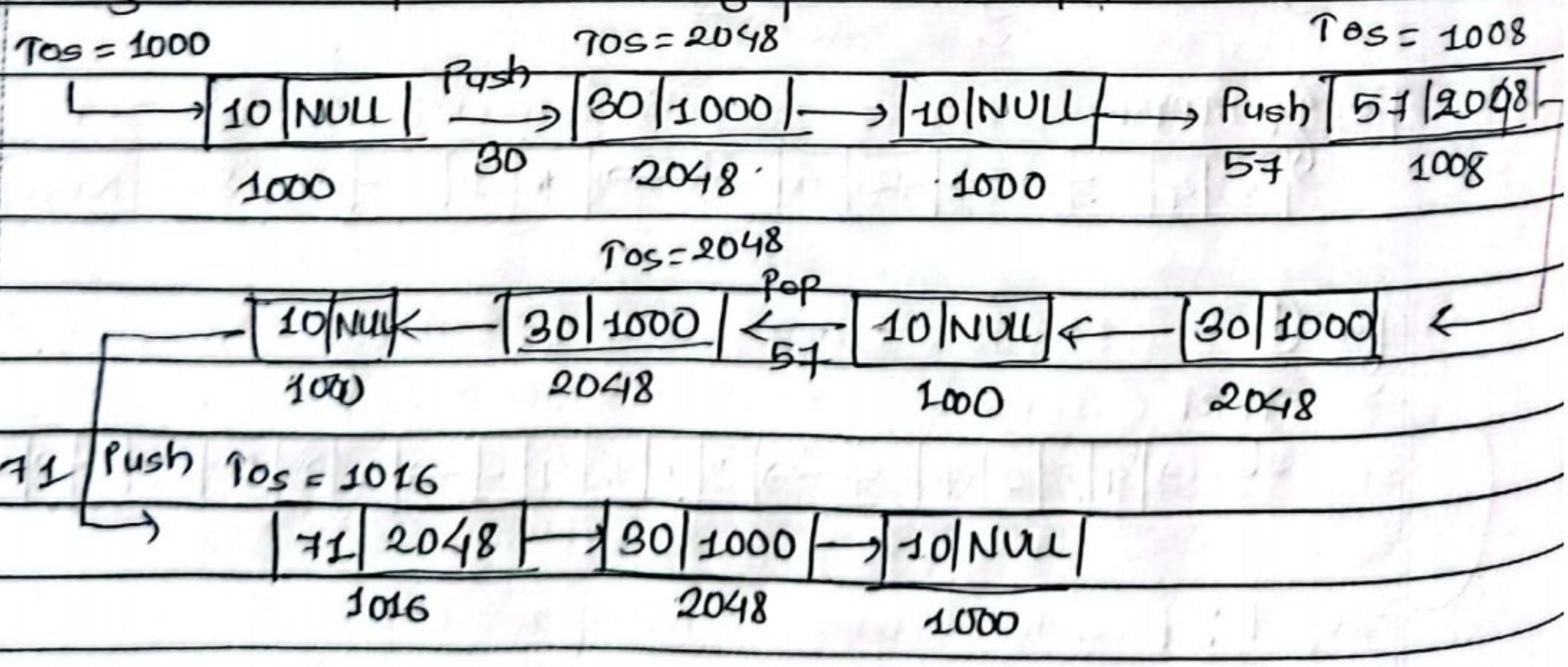


fig: linked list of stack.

→ The graphical representation of push and pop can be shown as



Date
Page

→ Stack is said to be empty if tos = NULL

Algorithm for push operation in stack with linked list implementation

Assumption

→ let TOS be the external pointer variable or reference variable from which push and pop operation is performed. Stack is said to be empty if TOS = NULL. Also let TEMP, $TEMP_1$ are pointer variable that can hold the address of Node.

Algorithm

CamScanner

Linked list implementation of Queue

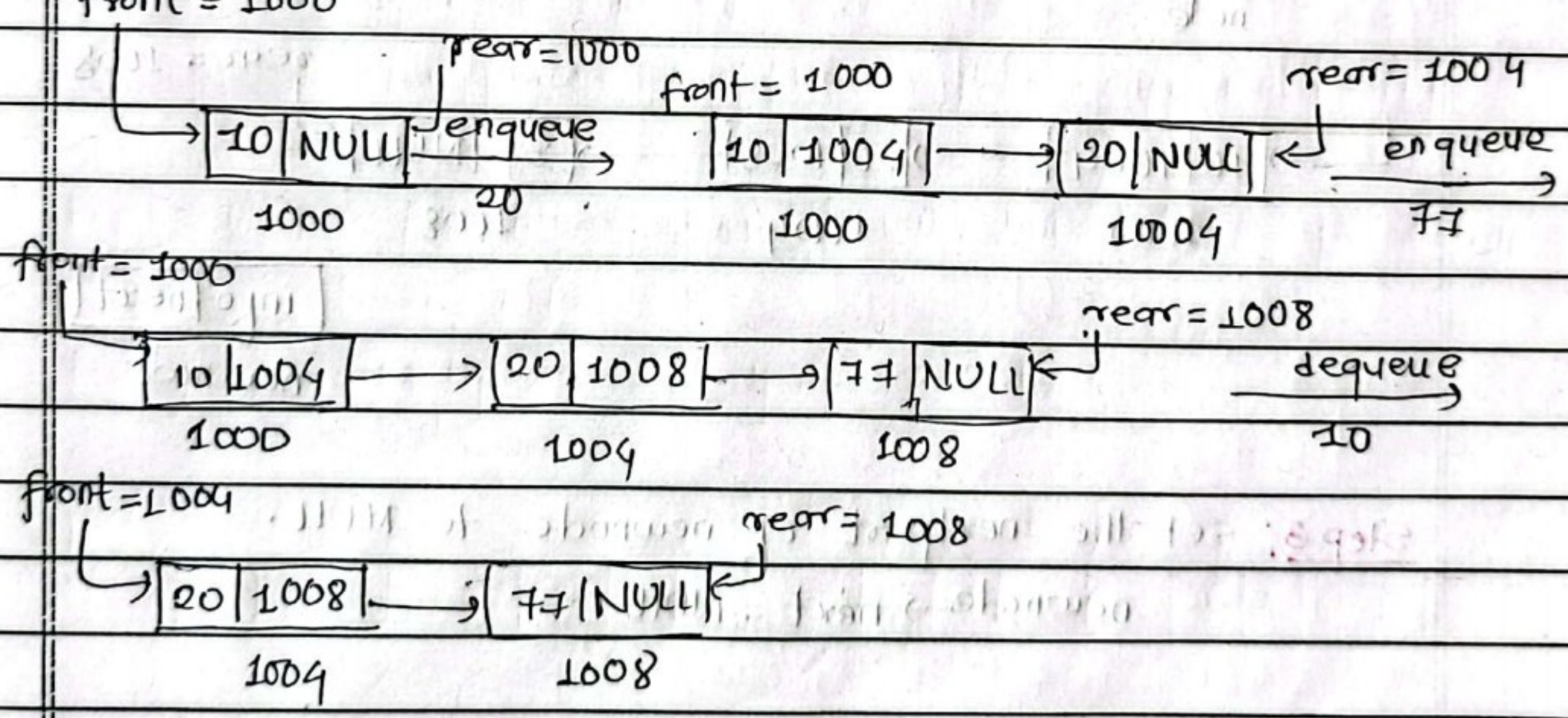
→ Like stack, queue can also be be implemented using linked list.

→ For this, we create a linked list and perform the basic operation of queue (en-queue and de-queue) on it)

→ we define two external pointer variable named as "front" and "rear" as a reference of de-queue and enqueue node respectively in the queue.

→ Queue is said to be empty if front = NULL and rear = NULL.

→ Enqueue and dequeue can be shown graphically as:



→ Front holds the address of first node in linked list and rear holds the address of last node in the linked list implementation of queue.

Algorithm for enqueue operation in queue with linked list

Linked list implementation of stack implementation.

Assumption

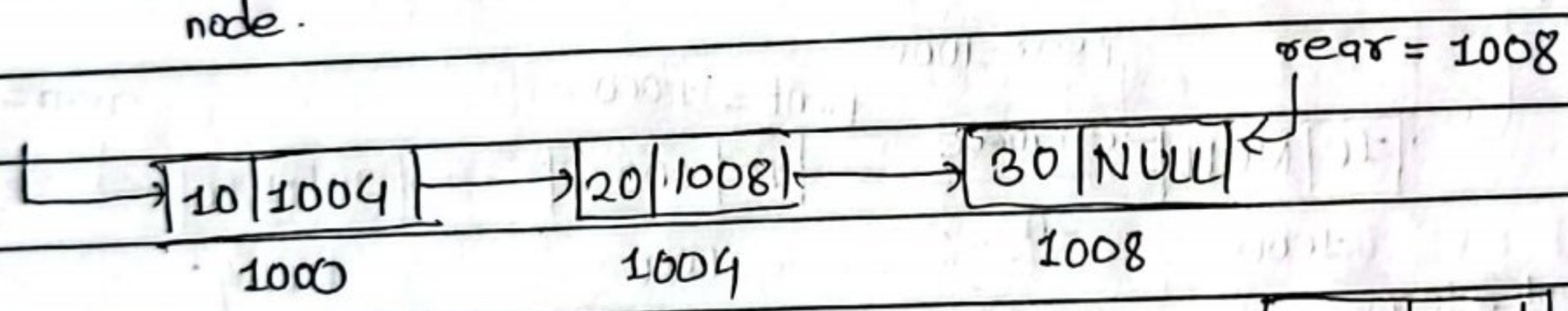
→ let "front" and "rear" are two external pointer variable that hold of first and last node of queue respectively. Queue is said to be empty if front = NULL and rear = NULL. Also TEMP, TEMP1 are pointer variable that can hold the address of node.

Algorithm

Step 1: Create a newnode.

newnode = (struct node*) malloc(size of (struct node))

Step 2: Read data from user and put in the info field of new node.



Step 3: set the nextpart of newnode to NULL.

newnode→ next = NULL

Step 4: Check for queue empty

if (front = NULL and rear = NULL)
set front = newnode
rear = newnode and exit

Step 5: Update the next part of rear to point newnode

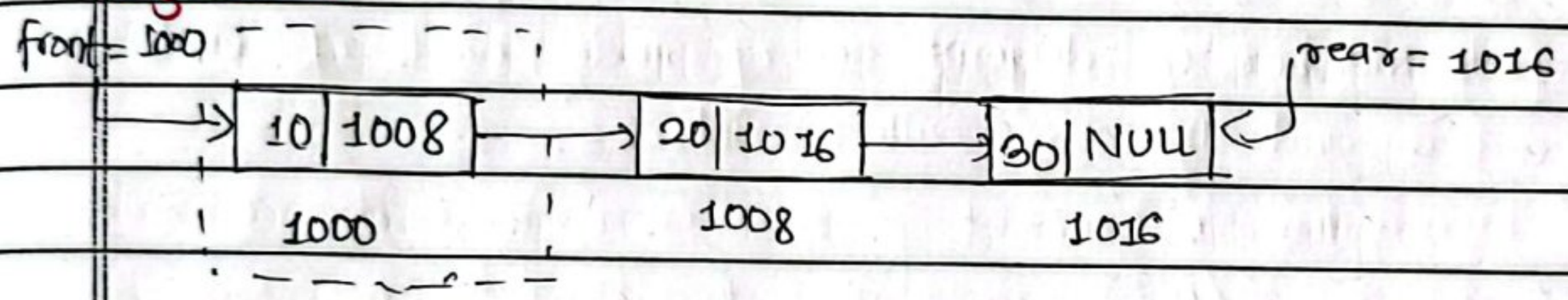
rear->next = newnode.

Step 6: Update rear to point newnode
rear = newnode

Step 6: Stop

Algorithm for dequeue operation in queue with linked list implementation

Algorithm



Step 1: Check for queue empty
if (front = NULL and rear = NULL)
Print queue empty and exit.

Step 2: Hold the address of first node in TEMP variable
TEMP = front

Step 3: Check for only one element in queue
if (front → next = NULL)
set front = NULL
rear = NULL
free (TEMP) and exit.

Step 4: Update front to point the second node in queue.
front = front → next

Date
Page

Step 5 : Release the memory occupied by deleted node.
free (TEMP)

Step 6 : Stop.

CamScanner
CamScanner

Date
Page

UNIT - 4

TREE

→ A tree is a non-linear data structure.

→ Formally a tree is defined as a finite set of one or more node such that :

(a) There is a special data item in the hierarchial arrangement called as root of the tree.

(b) It's remaining nodes (data item) are partitioned into a finite number of disjoint set of node, each of which is a tree and the are called subtrees.

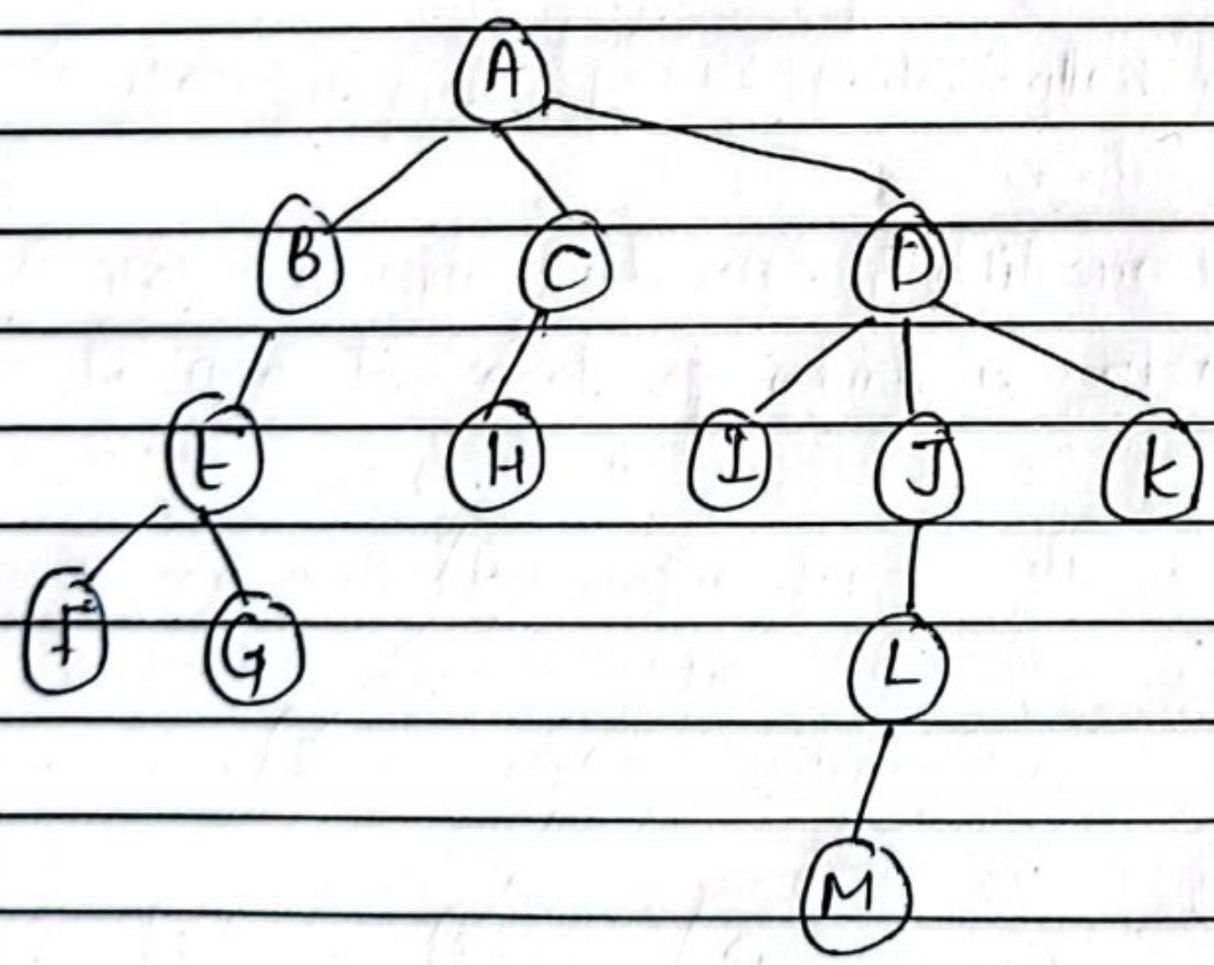


fig 1 : Simple structure of tree

→ In fig 1, A is the root of the tree and remaining nodes are partitioned into 3-disjoint sets.

{B, E, F, G}, {C, H}, {D, I, J, K, L, M}

CamScanner

Terminologies on tree

ⓐ **Root:** It is the first node in the hierarchial arrangement. In fig 1, 'A' is root node.

ⓑ **Parent:** A node that points to another node is called the parent node to those nodes while the nodes pointed are called nodes.

Every node has exactly one parent.

ⓒ **Degree of node:** It is the number of subtree of a node. Simply the count of children of any node gives it degree. In fig 1, the degree of A, D, C, M are 3, 3, 1 and 0 respectively.

ⓓ **level of subtree:** The entire tree is levelled in such a way that the root is always at level '0' and as we move down the tree, the level of node increases in such a way that if a parent is at level 'n' then its children will be at level (n+1)

→ In fig 1, A is at level 0, B, C and D are at level 1, E, H, I, J, K are at level 2 and so on.

ⓔ **leaf node:** Node with degree '0' are called leaf node or node with no-children are called leaf node. In fig, F, G, H, I, M and K are leaf node. They are also called terminal node.

ⓕ **Non terminal node/ non-leaf node:**

→ Any node except root node whose degree is

non-zero is called non-leaf node. Here B, C, D, E, I are non-terminal node.

(g) Siblings: Two or more nodes are siblings to each other if they the same parent node. Here node {B, C, D} and {F, G} are siblings to each other.

(h) Depth of tree: The depth of a tree is the maximum level plus one of any tree. Depth of tree is also called height of tree.

Binary tree

→ A binary tree is a set of zero or more nodes T such that

(a) There is a special node called root of the tree.

(b) The remaining node are partitioned into two disjoint set T_1 and T_2 each of which is also a binary tree.

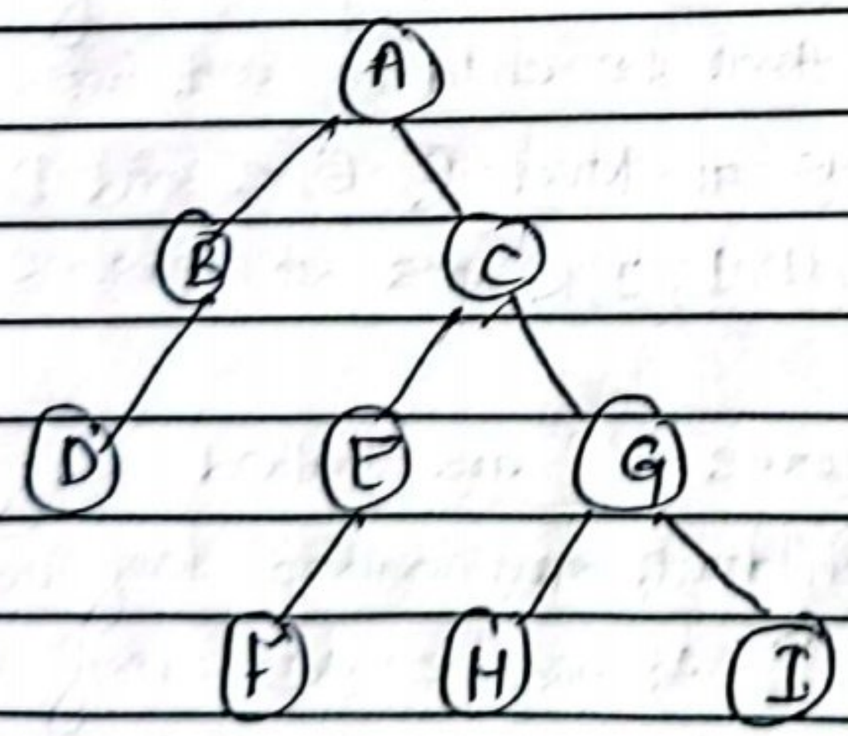


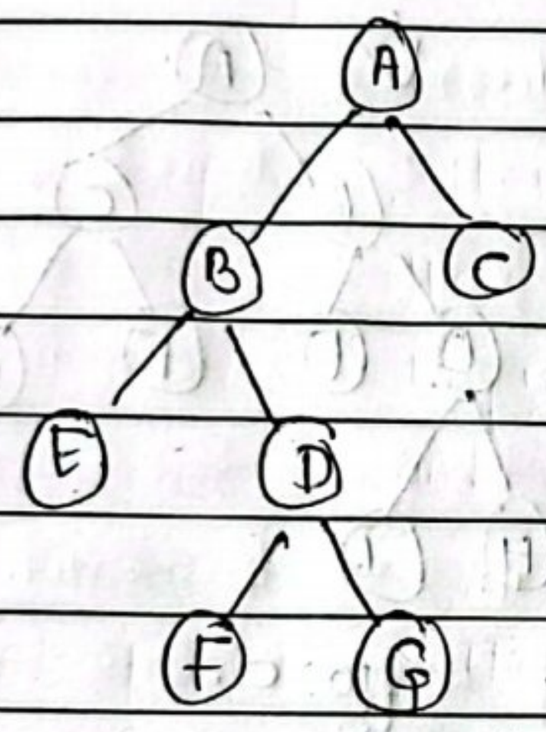
Fig: Binary tree

→ In a binary tree, the max^m degree of any node is at most 2 i.e there may be a node with 0, 1 and 2 degree.

Types of binary tree

(a) Strictly binary tree

If every non-leaf node in a binary tree has non-empty left and right sub tree then such tree is called strictly binary tree



degree: either 0 or 2.

fig: Strictly binary tree.

(b) Complete binary tree

→ A binary tree is a complete binary tree if and only if
(a) A tree is strictly binary tree and
(b) All the leaf node are at same level.

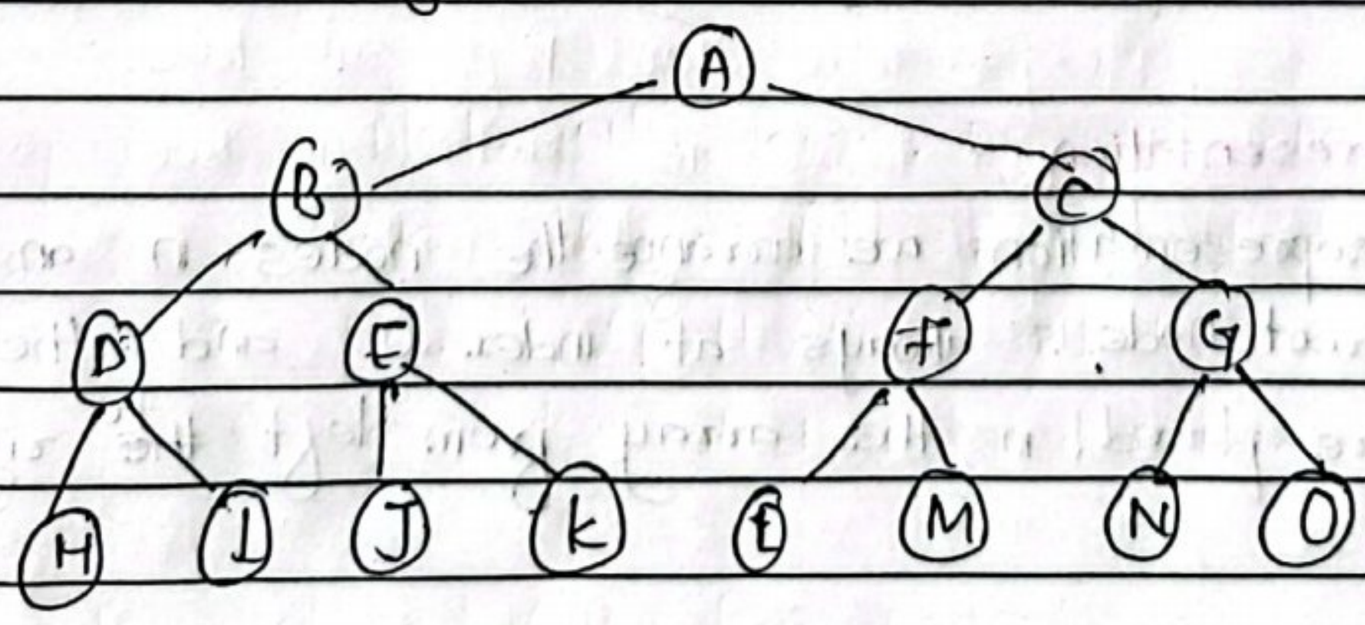


fig: Complete binary tree.

Date
Page

(c) Almost complete binary tree:

→ A binary tree of depth 'd' is an almost comple binary tree if.

(a) All node at level (d-1) has two children.

(b) For any node 'x' in the tree with right descendent at level d, x must have left Child and every left descendent of x is either a leaf of node at level 'd' or has two children

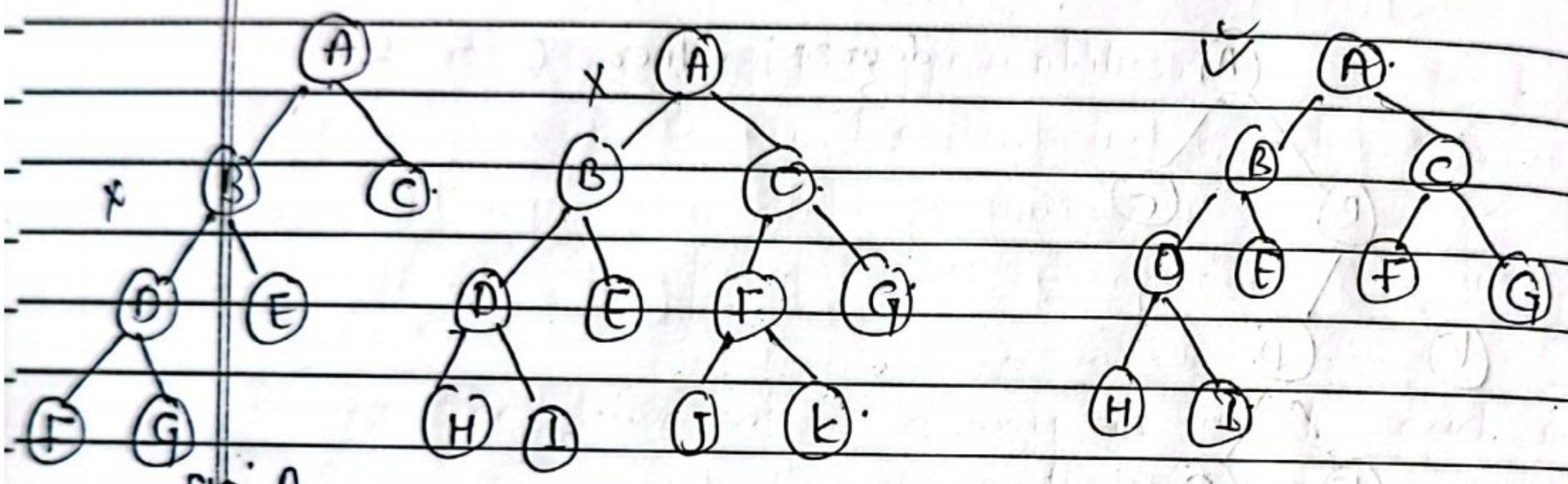


fig: A

fig: b

fig: c.

Representation of binary tree

→ we can represent the tree with

(a) Array representation

(b) Linked list representation

(a) Array representation

→ In array representation we arrange the nodes in an array such that root node is always at index '0' and the remaining nodes are placed in the array from left ~~the~~ to right otherwise.

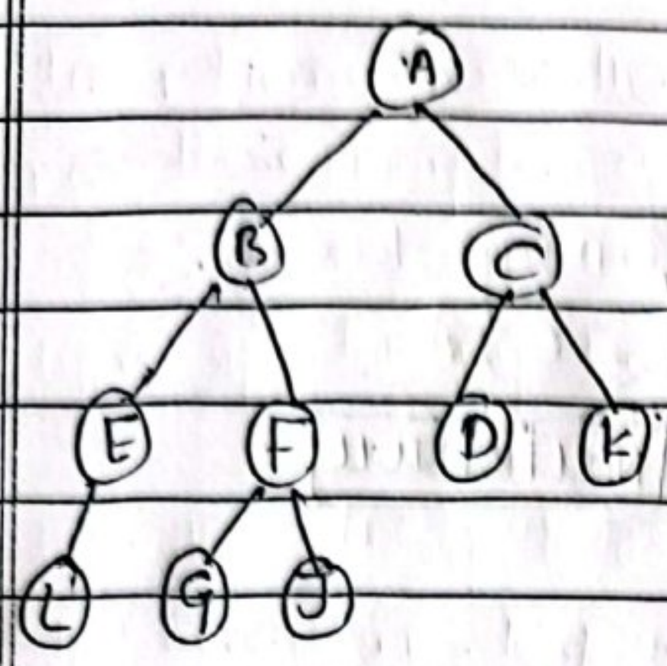


fig :-

Array representation of above tree is.

A	B	C	E	F	D	K	L	G	J
0	1	2	3	4	5	6	7	8	9

(b) Linked list representation

→ In linked list representation for any node we have three parts Lchild, info, Rchild.

Lchild	info	Rchild

→ Lchild holds the address of left child of a node, info holds the exact data the node stores and Rchild holds the address of right child.

→ The Lchild part will be NULL if node donot have left child and Rchild part will be NULL if node don't have right child.

→ The Lchild & Rchild of leaf node are always NULL.

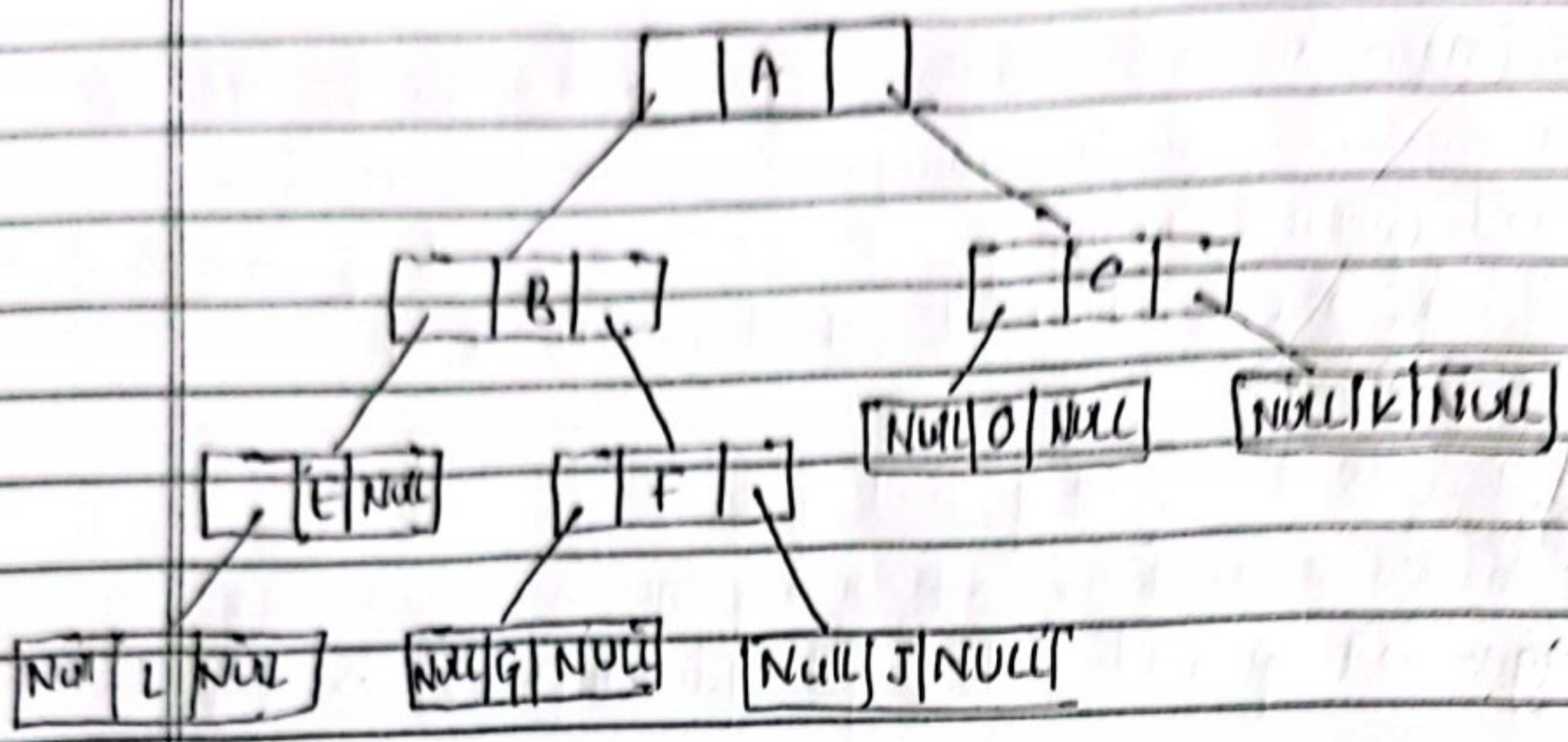


fig: Linked list representation of binary tree.

Traversal in binary tree

→ Binary tree traversal is a way in which each node in the tree is visited ~~exp~~ exactly only.

→ Following are the different possible order in which a binary tree can be traversed.

(a) VLR (Pre-order traversal)
(b) LVR (In-order traversal)
(c) LRV (Post-order traversal)

where L stands for traversing left sub-tree.
'R' stands for traversing right sub-tree.
'N' stands for visiting the node.

(a) Pre-order Traversal (VLR)

Algorithm

→ visit the node

→ traverse in the left sub-tree.
→ traverse in the right sub-tree.

E.g

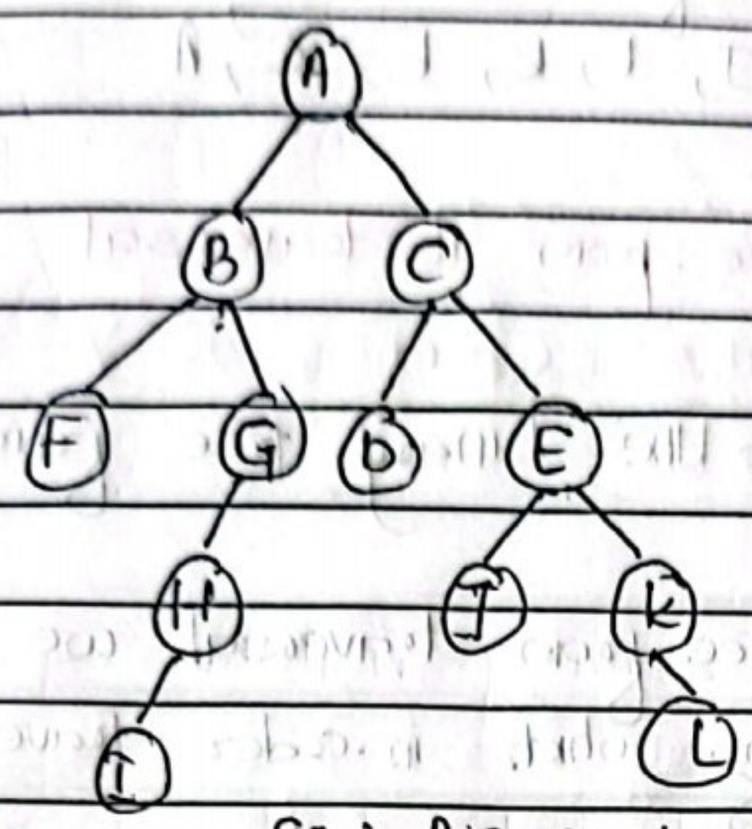


fig (a)

fig: Binary tree

→ The pre-order traversal of above binary tree is:

A, B, F, G, H, I, C, D, E, J, K, L

(b) The Inorder traversal (LVR)

Algorithm

→ ~~Trav~~ Traverse in the left sub-tree.
→ Visit the node
→ Traverse in the right sub-tree.

The in-order traversal of above binary tree is,

F, B, I, H, G, A, D, C, J, E, K, L

(c) Post-order Traversal (LRV)

Algorithm

→ Traverse in the left sub-tree
→ Traverse in the right sub-tree.

→ Visit the node.

The post-order traversal of above binary tree is:
F, I, H, G, B, D, J, L, K, E, C, A

Construction of binary tree from its traversal

→ we can reversely draw the binary tree from their traversal.

→ To construct a binary tree from traversal we need two traversal sequence, among which in-order traversal is compulsory. i.e
we need in-order and pre-order or in-order and post-order traversal.

Algorithm to construct the binary tree from pre-order and in-order traversal.

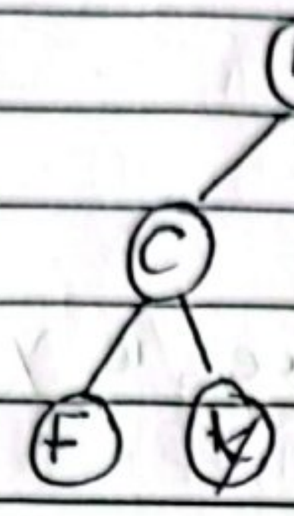
Step 1: Scan the pre-order traversal from left to right until only one node in the traversal sequence.

Step 2: Obtain the left sub-tree and right sub-tree of scanned node from the in-order traversal.

Step 3: Insert ~~one~~ one node at a time in the tree while constructing until in-order traversal becomes empty.

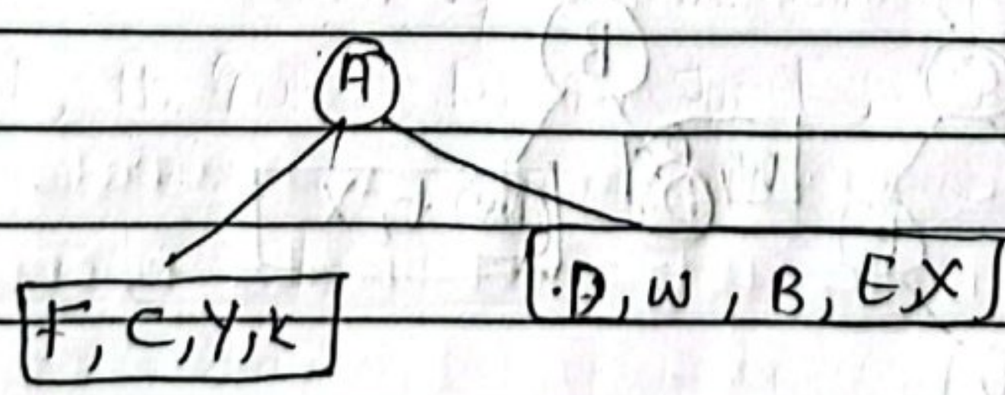
Q: Consider the following traversal & construct the binary tree:

Pre-order: A, C, F, K, Y, B, D, W, E, X.
In-order: F, C, Y, K, A, D, W, B, E, X

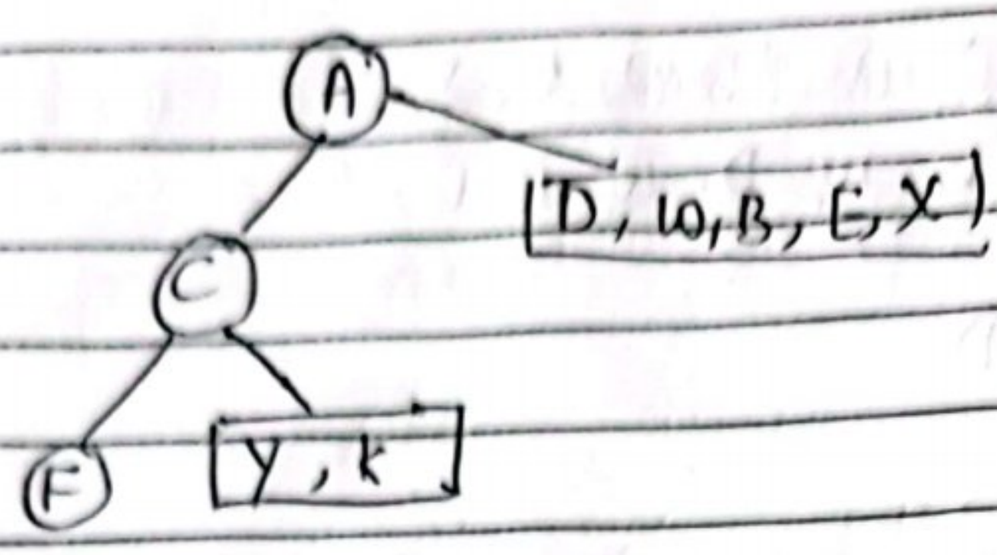


Solution

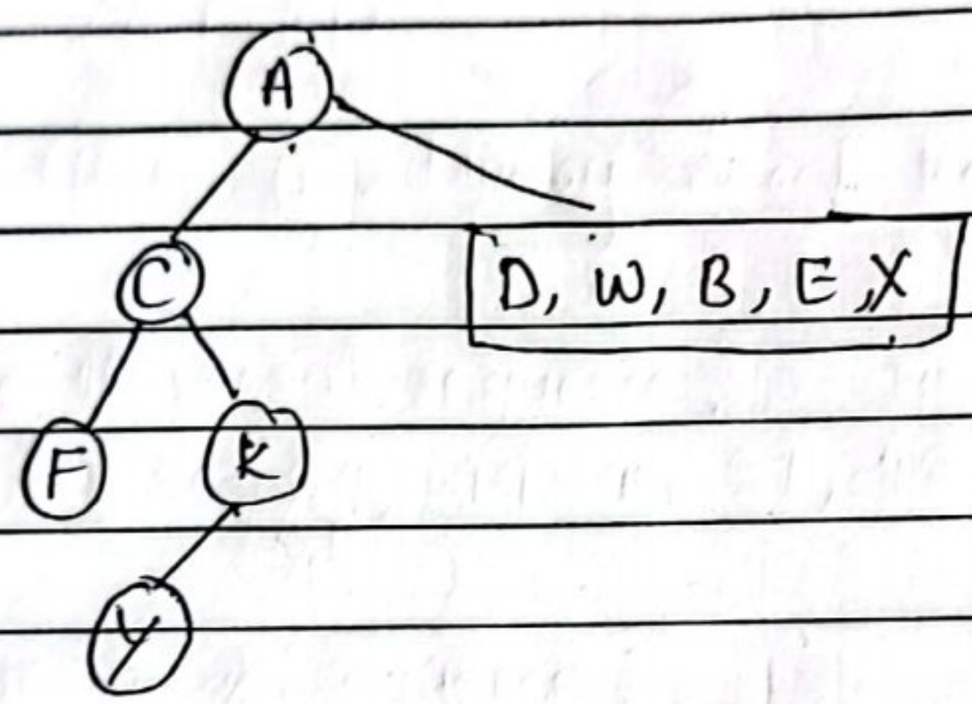
→ Scan the pre-order traversal from left to right, here scanned node is 'A' so the root node of tree to be constructed By obtaining the left subtree & right sub-tree of 'A' from In-order traversal i.e F, Y, K are left node & D, W, B, E, X are nodes.



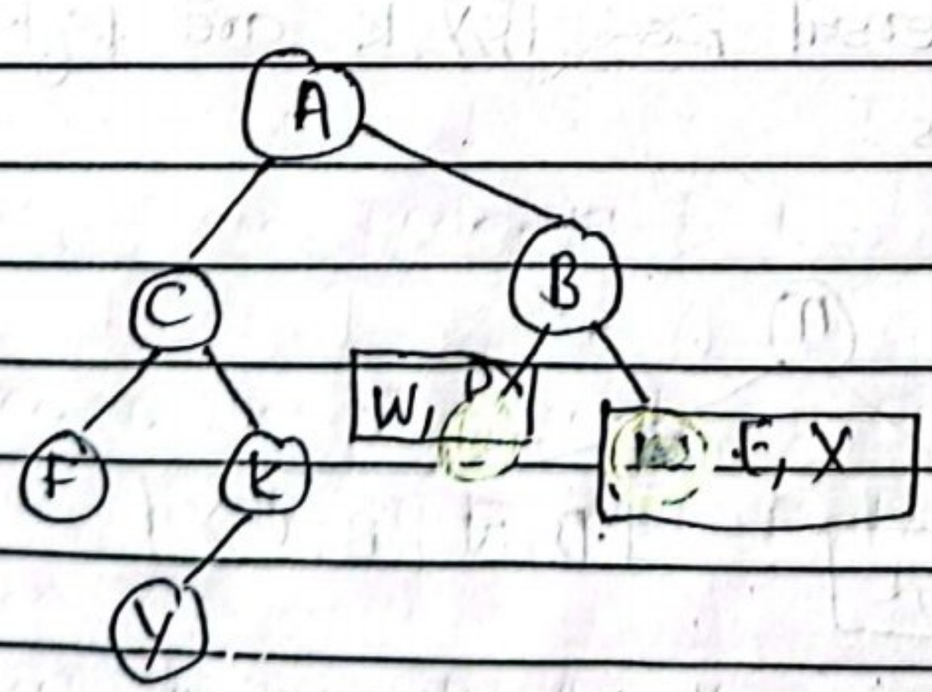
→ Next symbol in pre-order sequence is 'C' so it is the immediate left child of A & F and F and Y, K are its left and right sub-tree respectively.



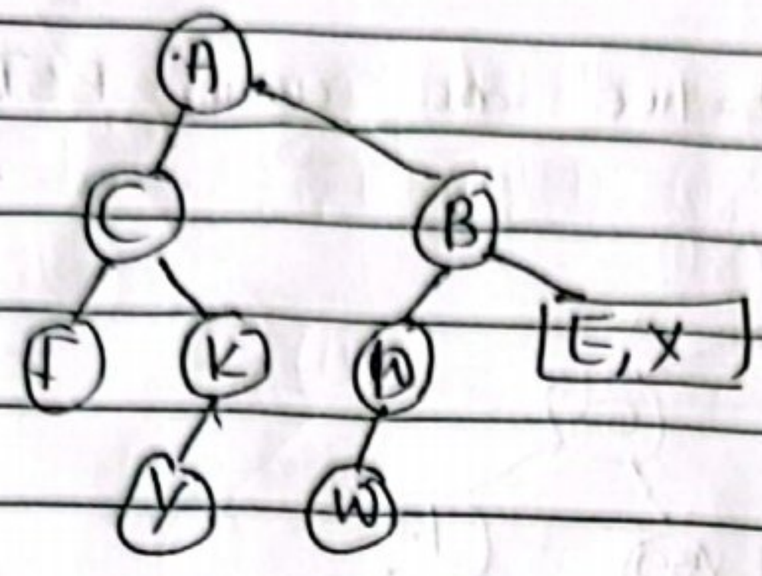
Next symbol in pre-order sequence is Y so it is immediate left child of K.



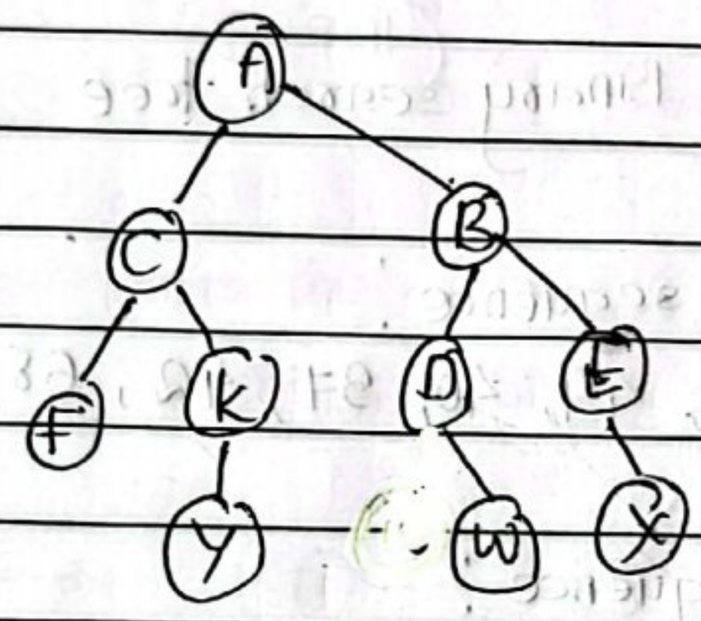
Next symbol in preorder sequence is 'B' so it is the right sub-child of A & W, ~~D~~ & ~~D~~ E, X are its ~~right~~ left & right subtree respectively.



Next symbol in pre-order sequence is D so it is its immediate left sub-child & ~~DV~~ is its right sub-child.



Next symbol in pre-order sequence is E which is immediate right sub-child of B & X is its sub-child.



Binary search-tree (BST)

→ A Binary search tree (BST) is a binary tree that is either empty or each node contains a value (key) and satisfy the following condition.

(i) All node in the left sub-tree of root are smaller than the key in the root.

(ii) All key in the right sub-tree of root are greater than the key in the root.

(iii) The left and right sub-tree are again BST.

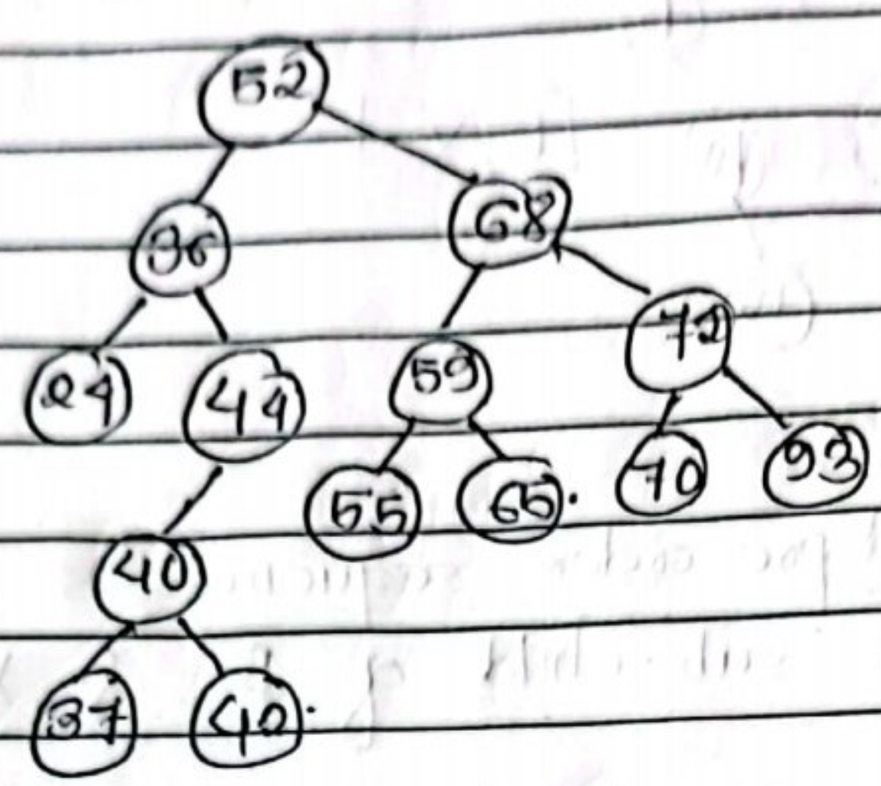


fig: Binary search tree

Pre-order traversal sequence:

52, 36, 24, 44, 40, 37, 42, 68, 59, 55, 65, 72, 70, 93

In-order traversal sequence:

24, 36, 37, 40, 42, 44, 52, 55, 59, 65, 68, 70, 72, 93

Post-order traversal sequence:

24, 37, 42, 40, 44, 36, 55, 65, 59, 70, 93, 72, 68, 52

Algorithm to insert node in BST

(1) If the tree is empty, create a new node 'n' storing the key 'x' in the node and make new node as root of the tree and exit.

else, compare 'x' to the key stored in the root 'R'

Date
Page

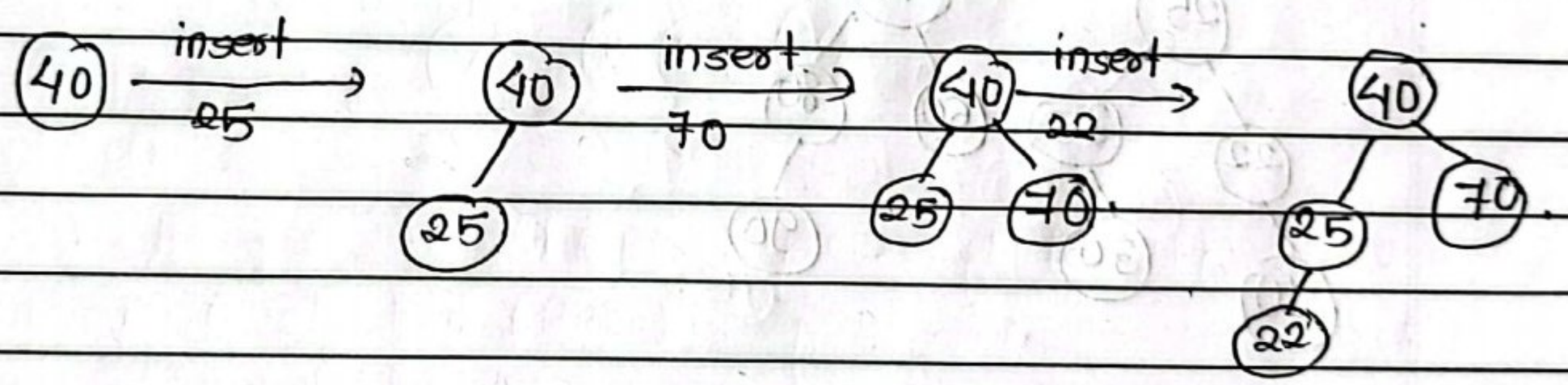
(2) If 'x' is identical to key in the root 'R' then stop i.e don't change the tree.

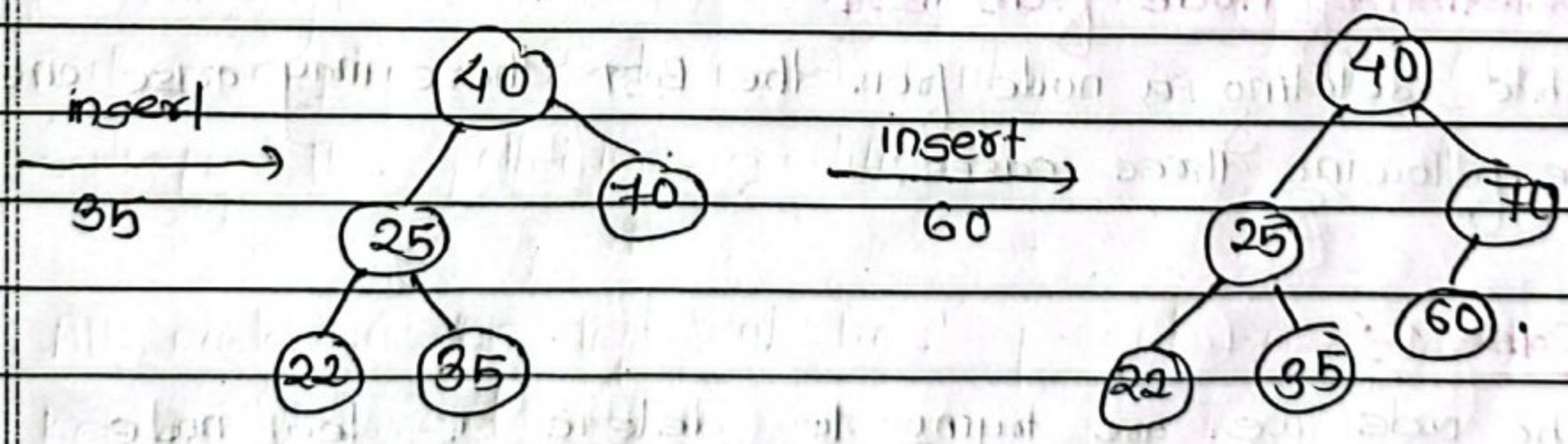
(3) If 'x' is smaller than the key in the root 'R' insert new item into left sub-tree of root.

(4) If 'x' is greater than the key in the root 'R' then insert new item in the right sub-tree of root.

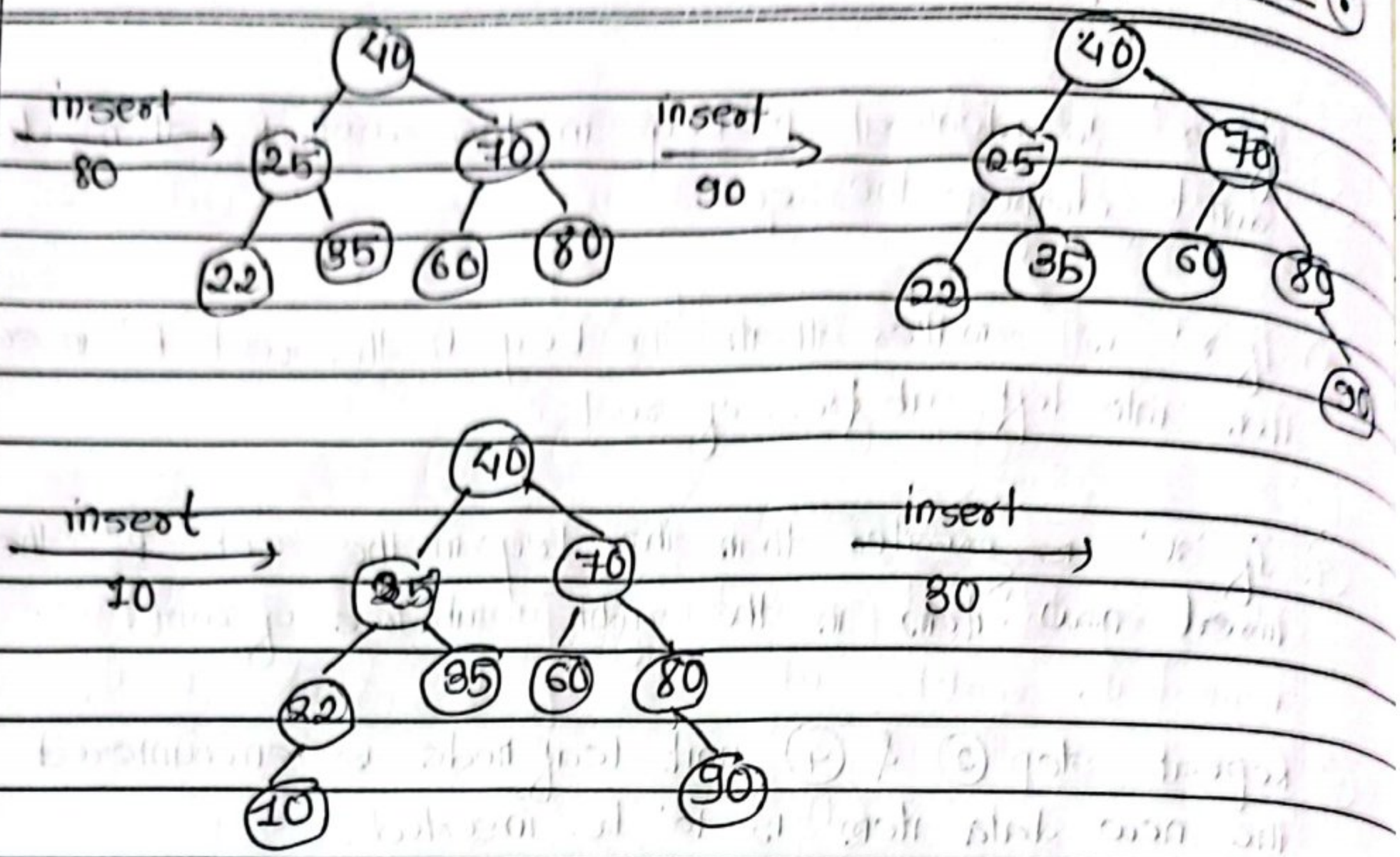
(5) Repeat step (2) & (4) unit leaf node is encountered where the new data item is to be inserted.

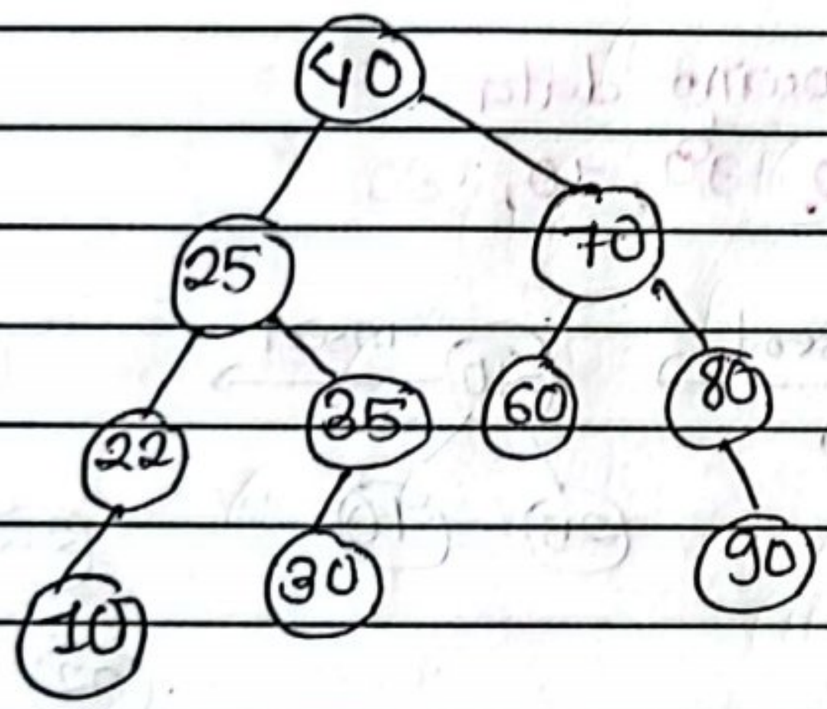
Construct BST from the following data
40, 25, 70, 22, 35, 60, 80, 90, 10, 30





CamScanner





Deletion of node from BST

While deleting a node from the BST there may arise one of the following three cases:-

Case 1:-

→ The node we are trying to delete is leaf node

→ In this case we immediately delete the node and update the left or right pointer of its parent node to null.

Date
Page

Case-2 :

→ The node we are deleting is the node with one child.
→ In this case we direct the child of deleted node to point the parent of deleted node.

Case-3 :

→ The node deleted is the node with two children.
→ The pointer of child node of deleted node is r
→ In this case we re-placed the deleted node in two ways:

(a) The deleted node can be replaced with the right most node of left sub-tree.
(b) The deleted node can be replaced with the left-most node of right sub-tree.

E.g

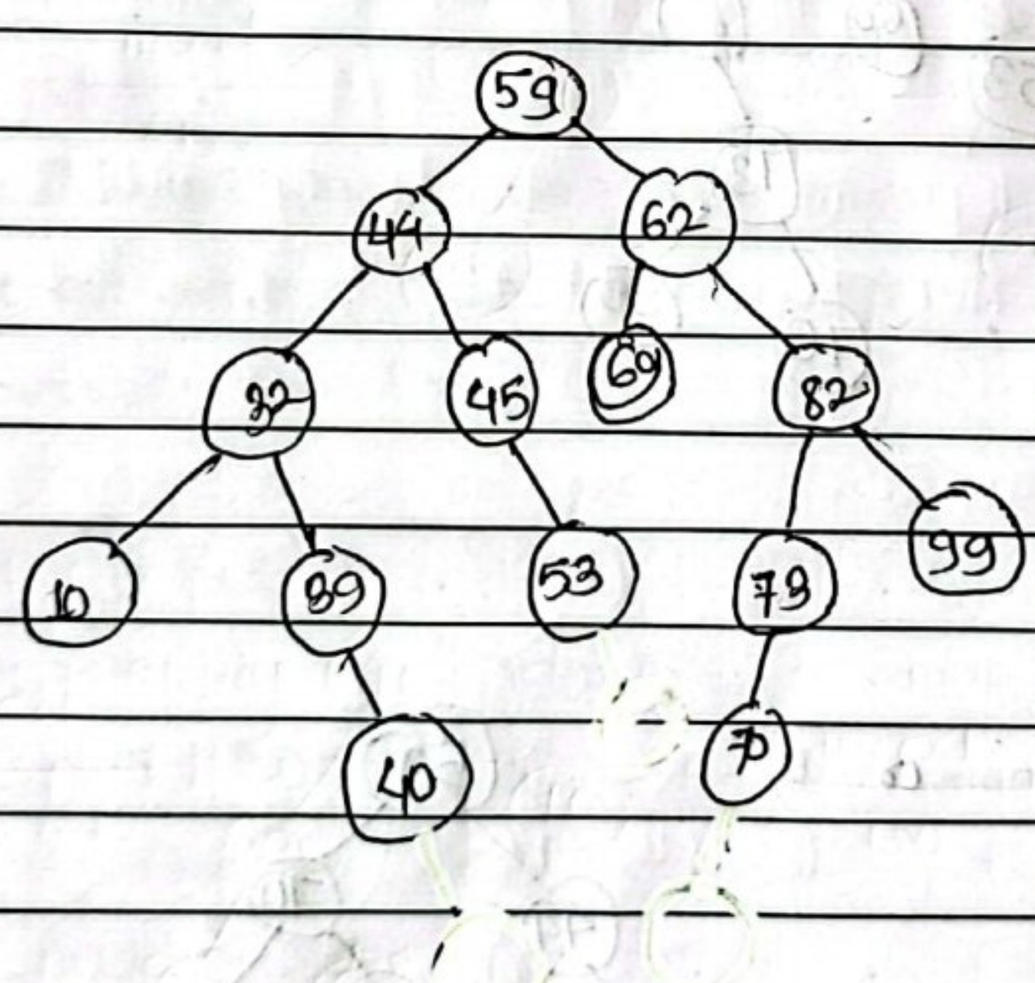
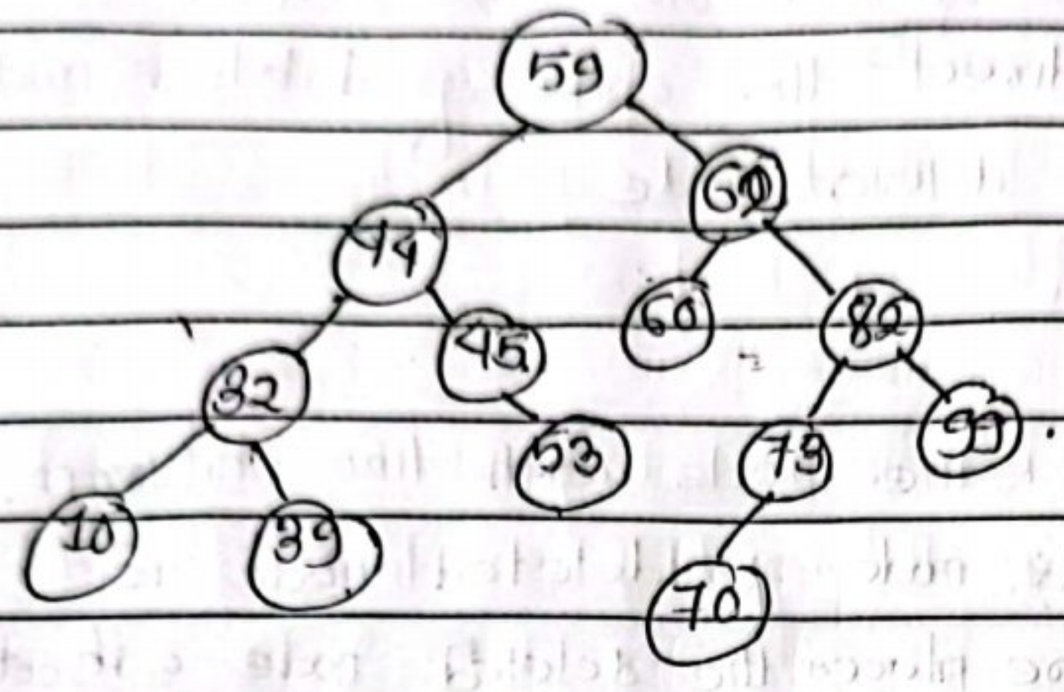


fig: BST

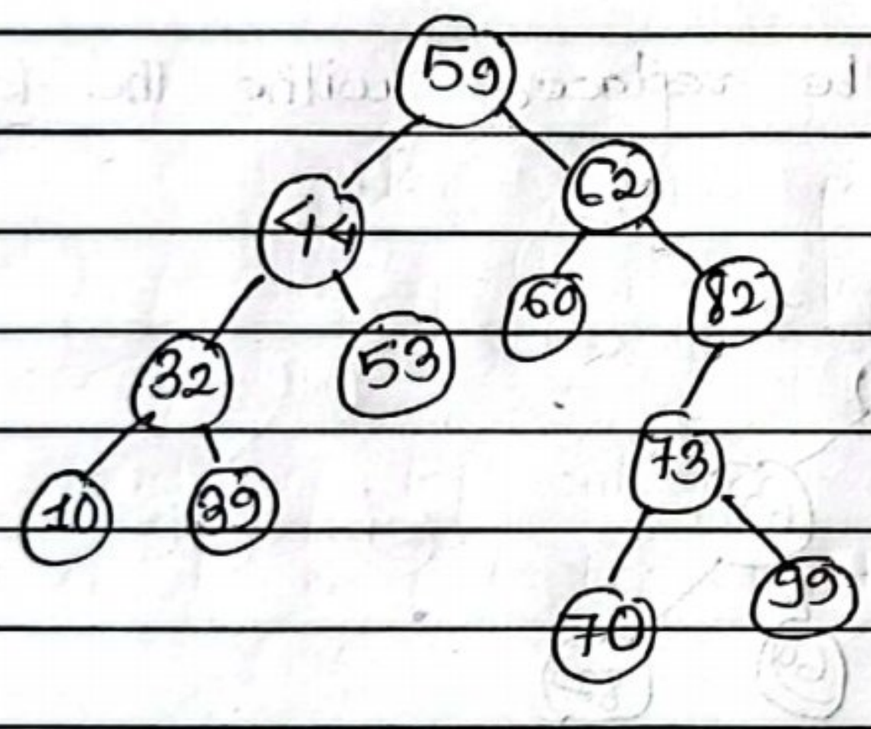
Update the above tree after deletion of node (40), (45) & (62).

Date
Page

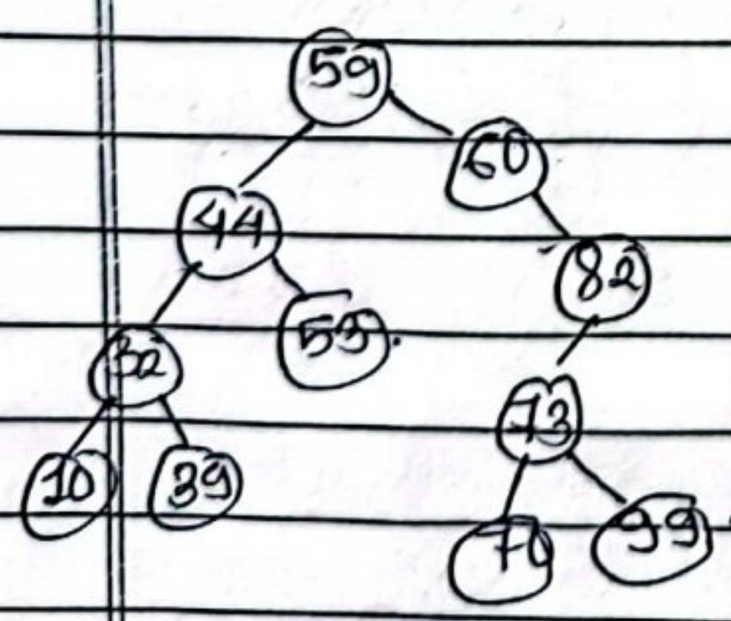
Deletion of (90).



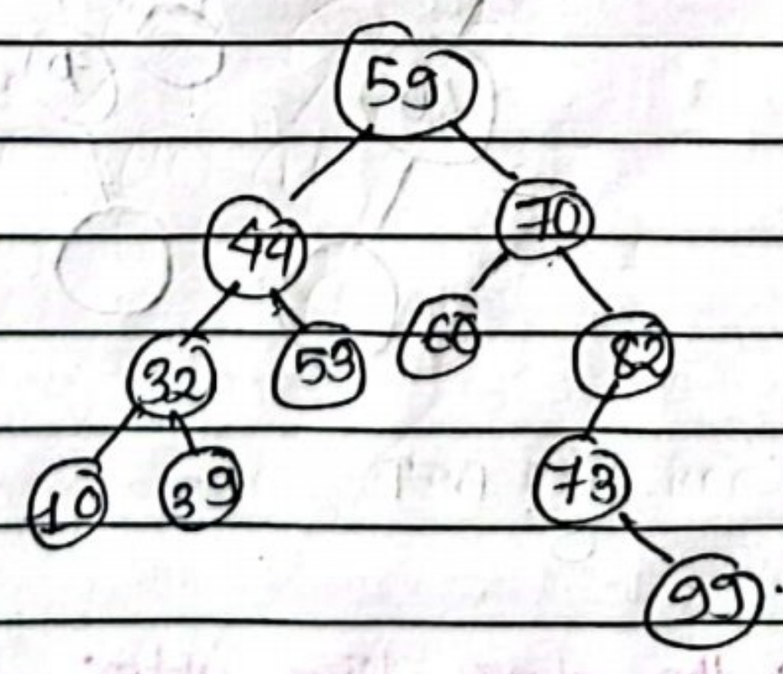
Deletion of (48)



Deletion of (62).

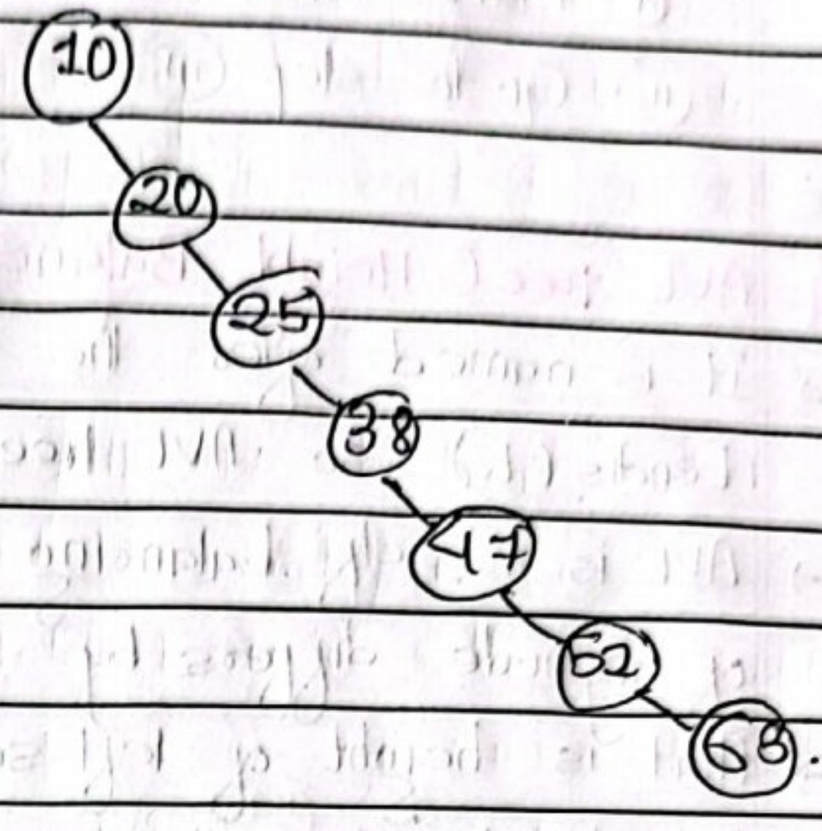


or



CamScanner

10, 20, 25, 38, 47, 52, 68



Searching a node in BST

→ The search operation in BST refers to the process of searching for a specific value (key) in the tree.

Algorithm

(i) Input "~~Data~~ DATA" to be searched.

(ii) Make a current node to point root node of tree.

(iii) If current node is NULL

  (a) Display not found

  (b) Exit

(iv) Else, compare the "DATA" with the value of current node.

  (a) If the "DATA" equals to the value of current node.

    (i) Display "DATA" found.

    (ii) Exit.

  (b) If the "DATA" is less than the value of current node

    (i) Make current node to point the left sub-tree.

    (ii) Go to step (iii)

CamScanner

(c) If "DATA" is greater than the value of current node
(i) Make current node to point its right sub-tree.
(ii) Go to step (iii).

AVL tree (Height Balanced Tree)

→ It is named after the mathematician Adelson (A), Velskii (V) & Lendis (L) so AVL tree.

→ AVL is a self balancing binary search tree where the height of any node differs by almost "one".

→ That is height of left subtree and right sub-tree of node differs by almost 1.

→ It is also called height balanced tree.

→ we can define a term balance factor (bf) for any node which is just a difference of height of left sub-tree and height of right sub-tree.

bf = height of left sub-tree − height of right sub-tree

→ The value of "bf" in AVL tree must be 0, 1 or −1 for any node.

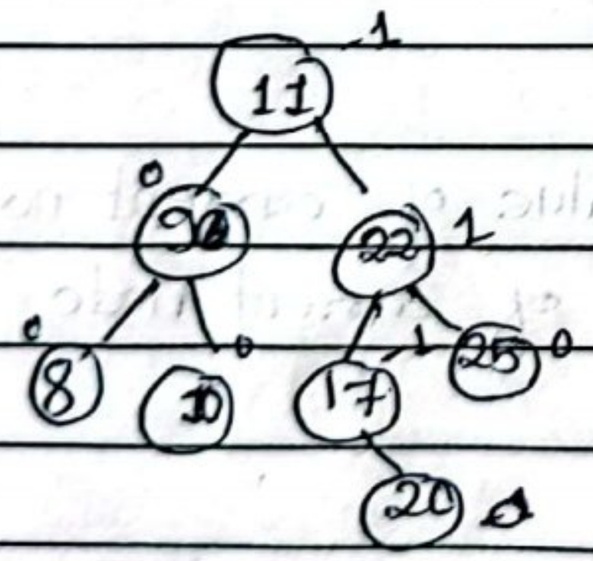


fig: AVL tree

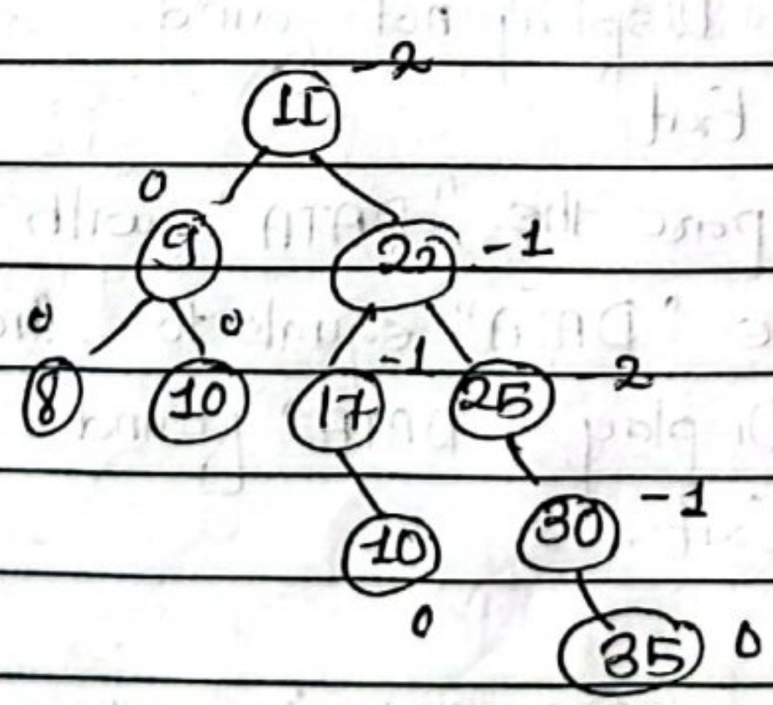


fig: Non-AVL tree.

Insertion of node in AVL tree

(i) Insert the node in the same way as BST.

(ii) After every insertion check the balancing factor of each node. If it is -1, 0 and 1 continue insertion of node otherwise stop insertion.

(iii) If any node are im-balanced and three nodes are in a straight line perform single rotation.

(iv) If three nodes are in dog leg pattern then perform double rotations.

Rotations in AVL tree

→ when an insertion of a node in AVL tree Create imbalance, then in order to balance the tree, we have to perform rotations.

→ There are three types of rotations:

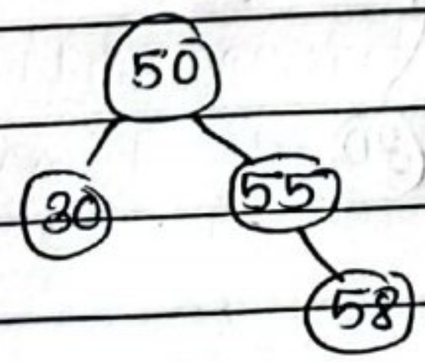
(a) Single left rotation (SLR)
(b) Single Right Rotation (SRR)
(c) Double rotations
  (i) Double left rotation (DLR)
    SLR + SRR
  (d) (ii) Double right rotation (DRR)
    SRR + SLR

(a) Single left Rotation (SLR)

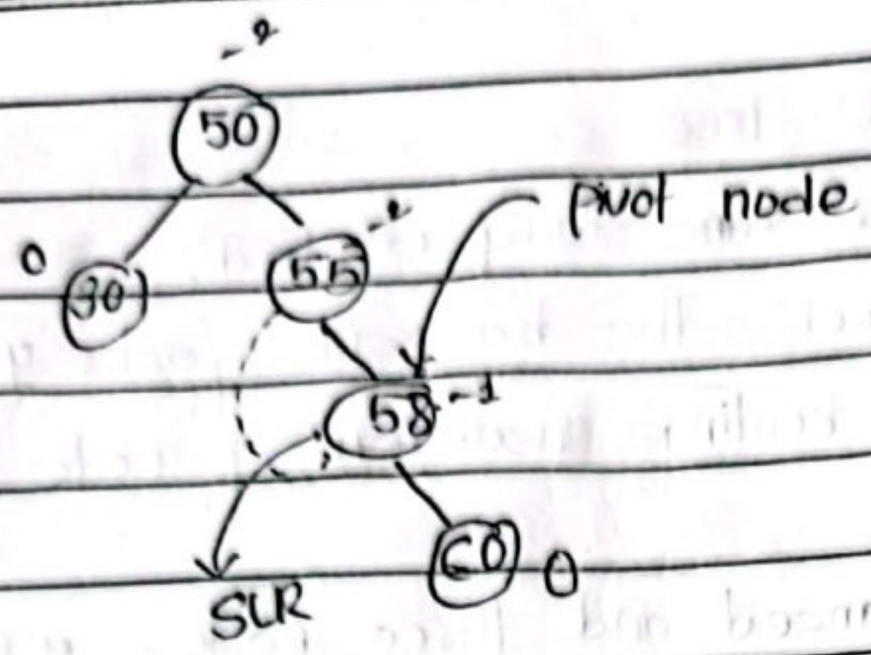
→ Imbalanced is occured due to insertion in the right sub-tree of right ~~subchild~~ of the pivot node.

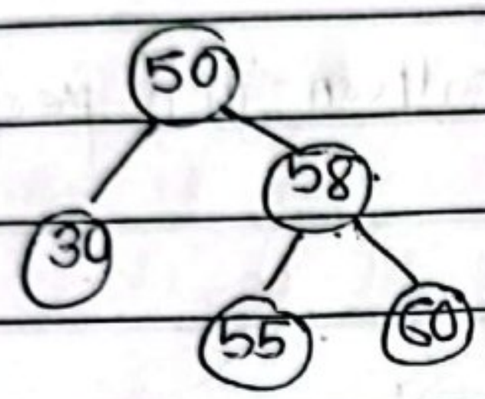
→ Balancing can be made by applying SLR on three nodes.

E.g.



Insert 60

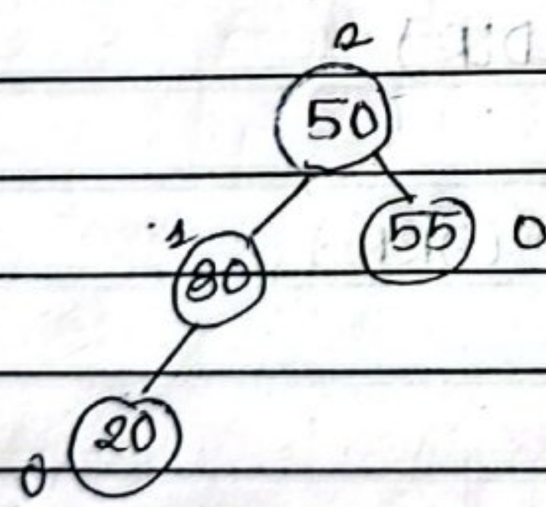




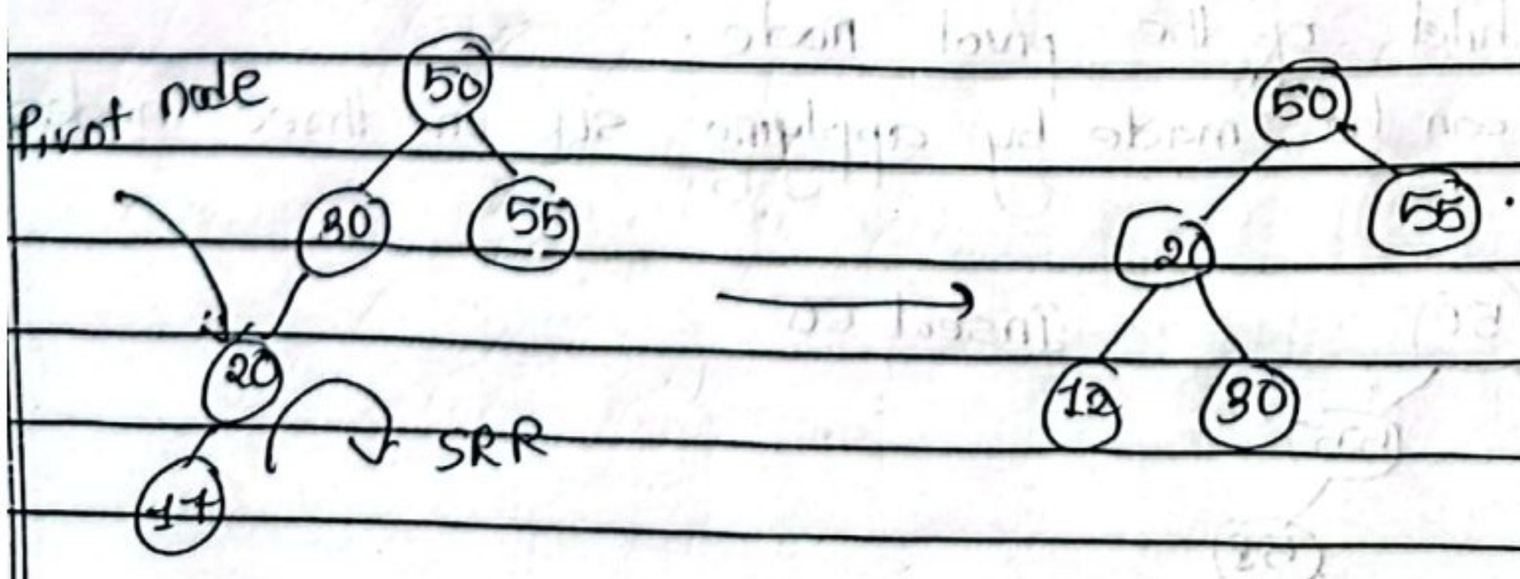
Single Right Rotation (SRR)

In imbalanced is occured due to insertion in the left sub-tree of left child of the pivot node.
Balancing can be made by applying SRR on three nodes.

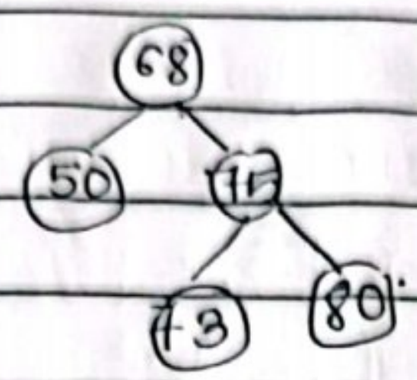
E.g.



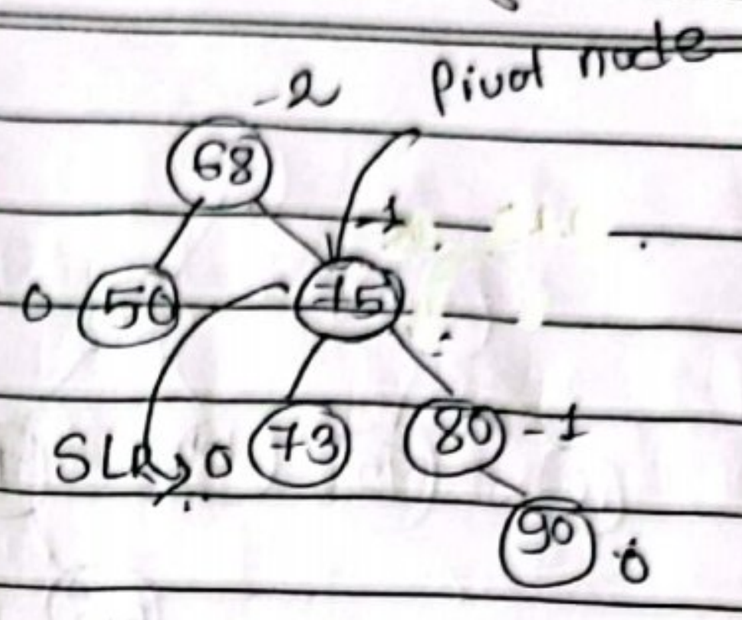
Insert (17).

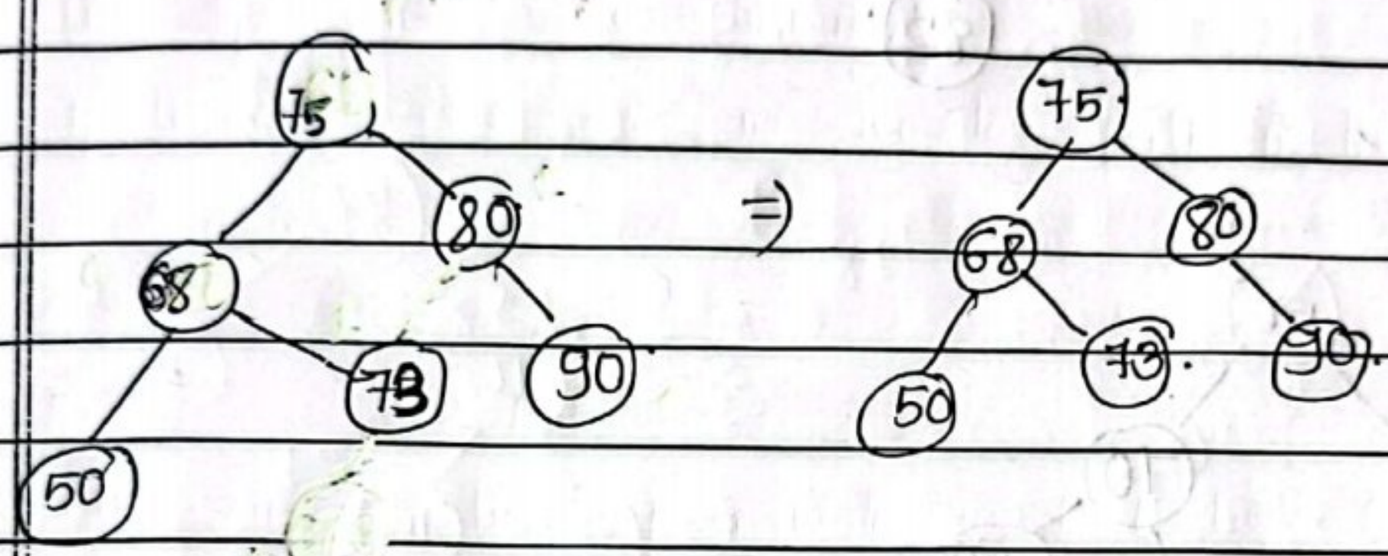


eg



insert 90.





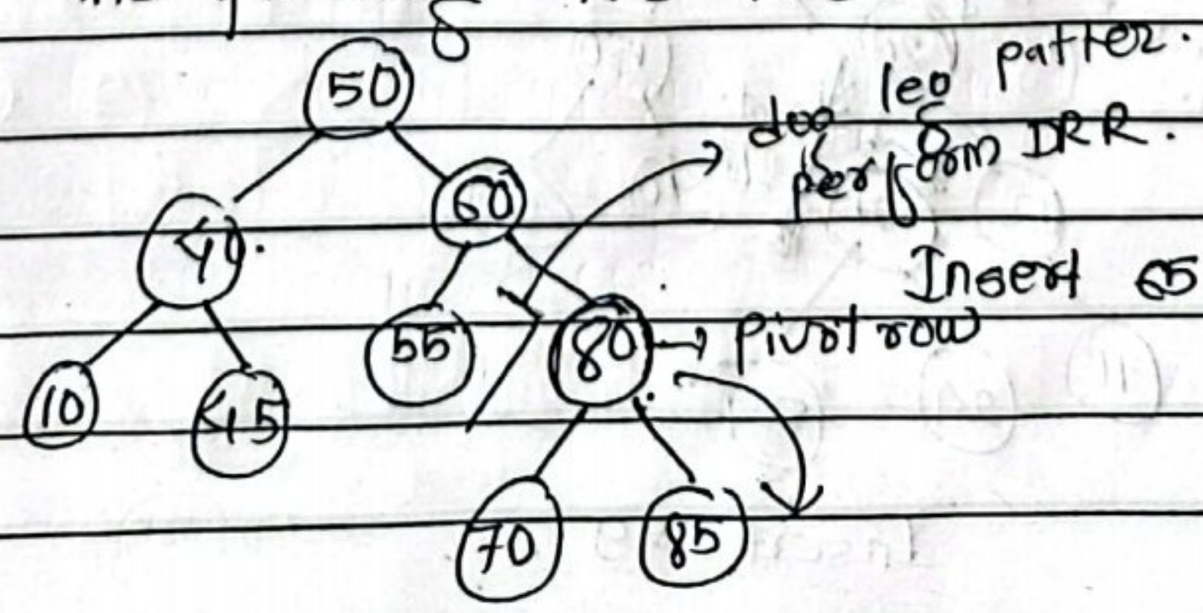
Double Rotations

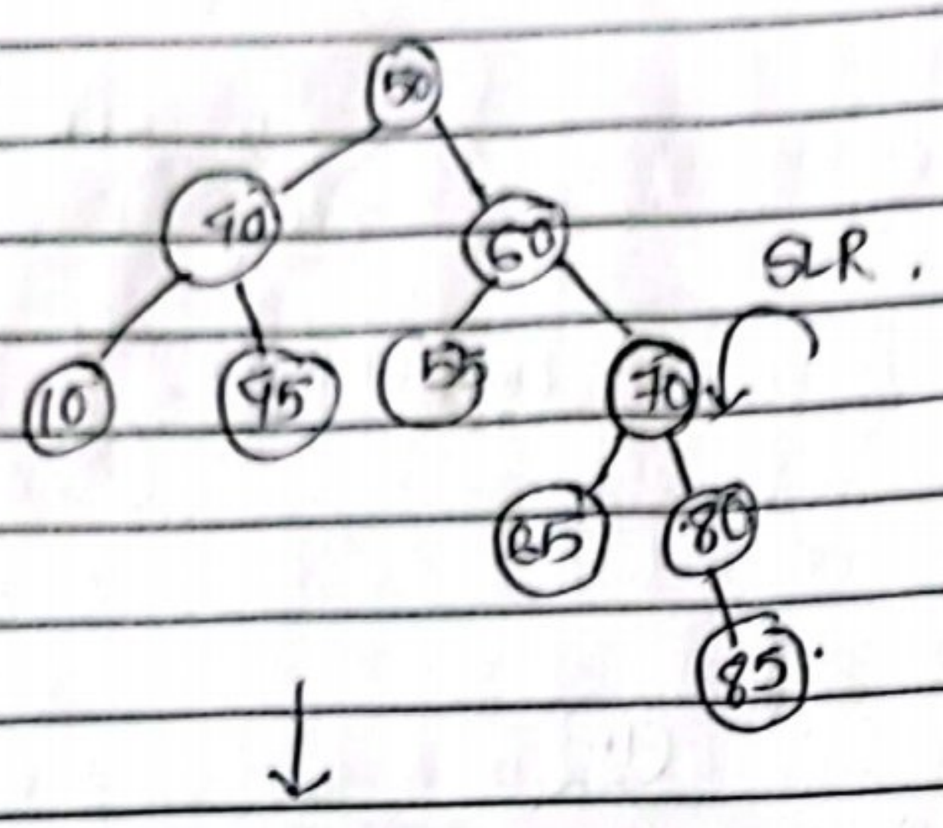
→ If imbalance is occured and if three nodes are in dog leg pattern, then we perform double rotations.

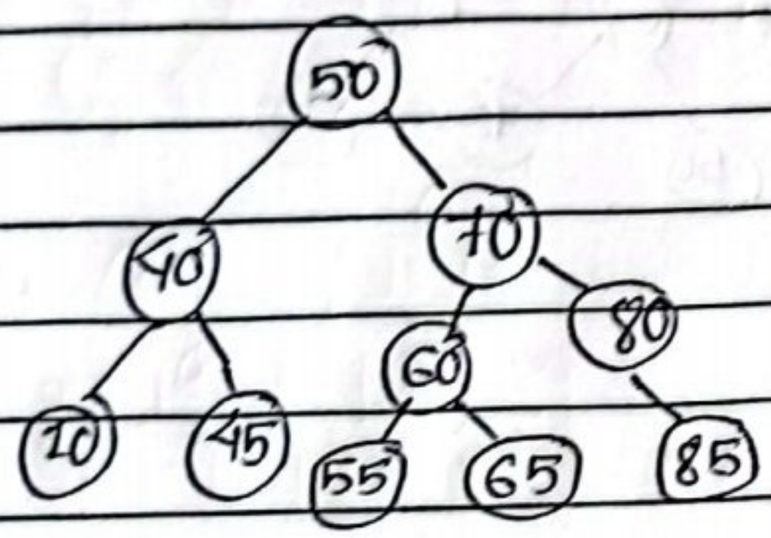
(a) Double Right Rotation (DRR)

→ Imbalance is occured due to insertion of a node in the left sub-tree of a right child of pivot node, then balancing is done by performing DRR i.e two single rotation in opposite directions.
SRR + SLR.

Consider the following AVL tree.



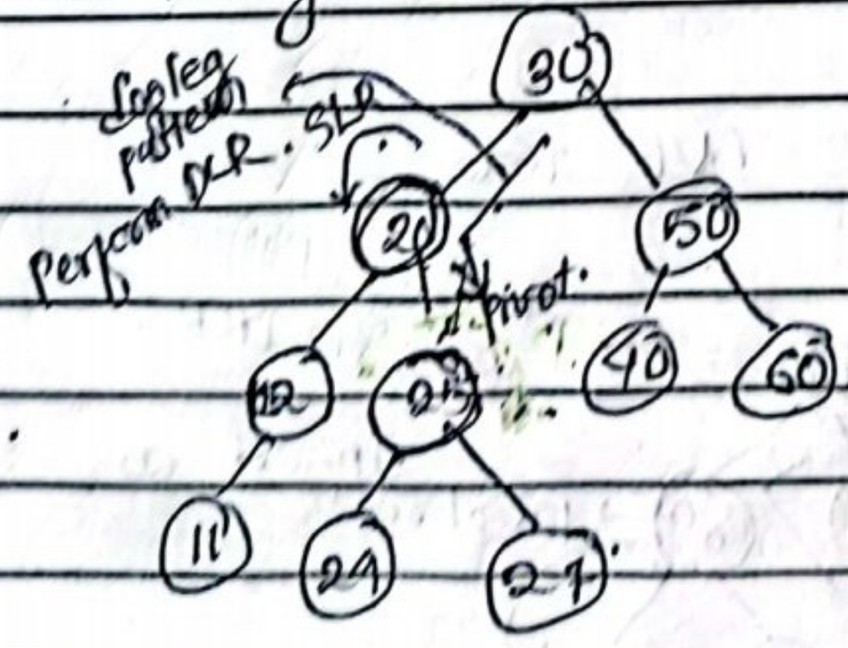




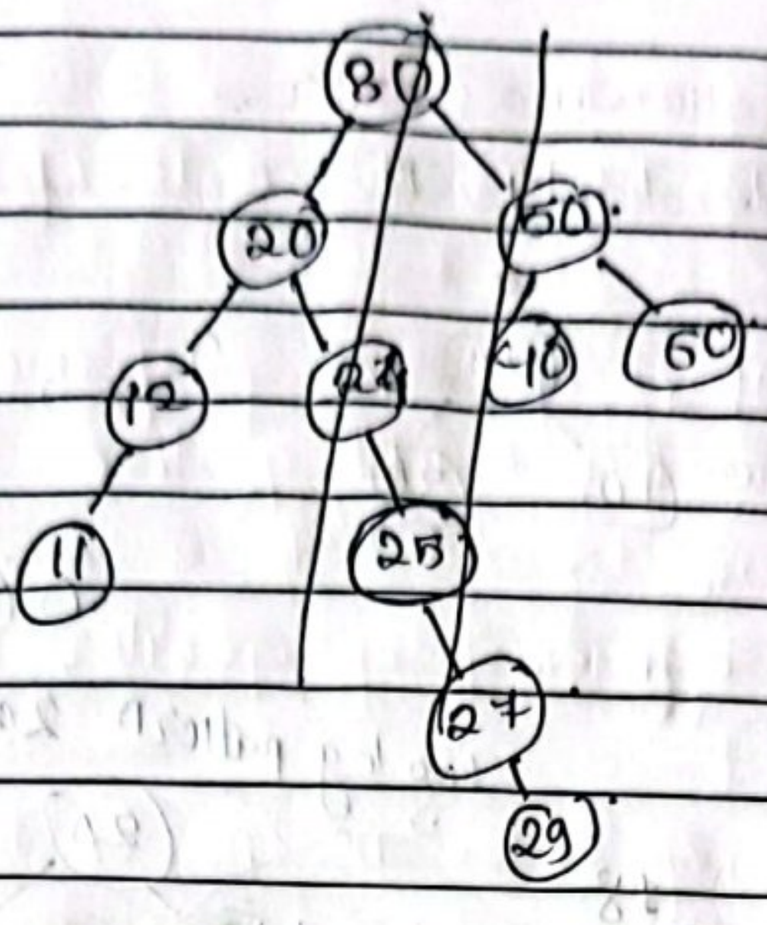
(b) Double left Rotation (DLR)

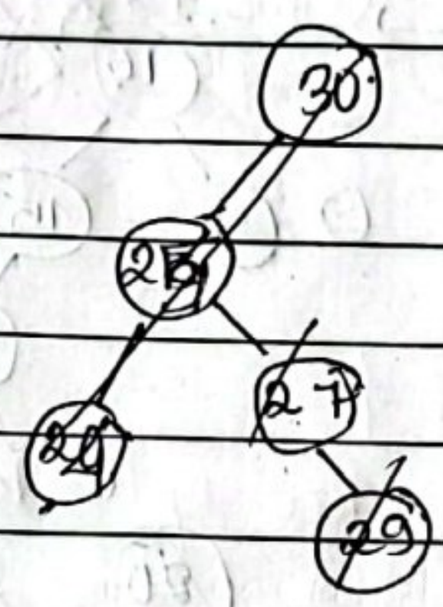
→ Imbalance is occured due to insertion of a node in the right sub-tree of a left child of pivot node, then balancing is done by performing DLR i.e two single rotation in opposite dir^ns.
SLR + SRR.

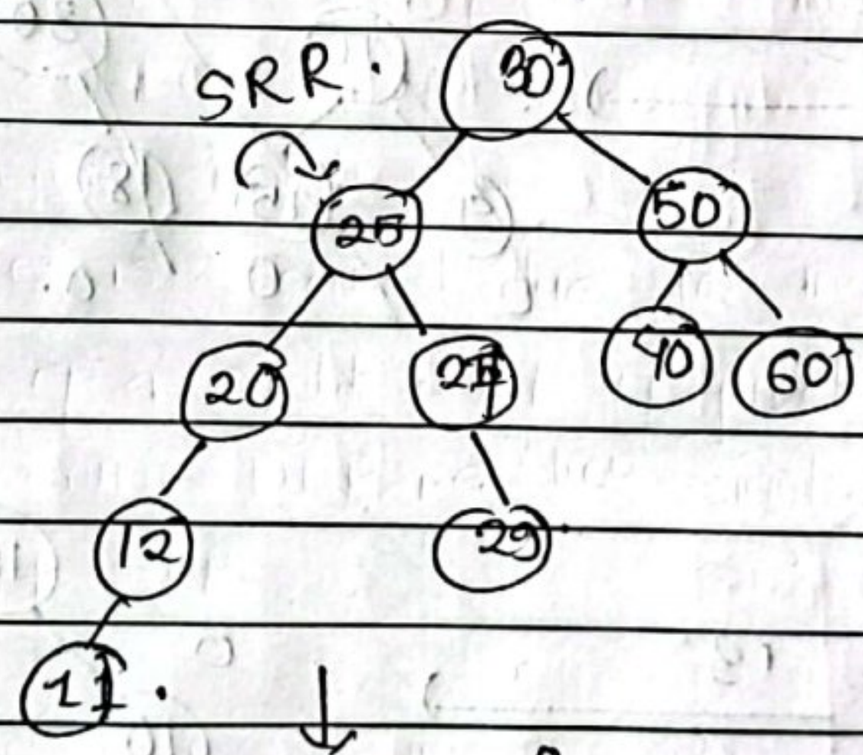
consider the given AVL tree.

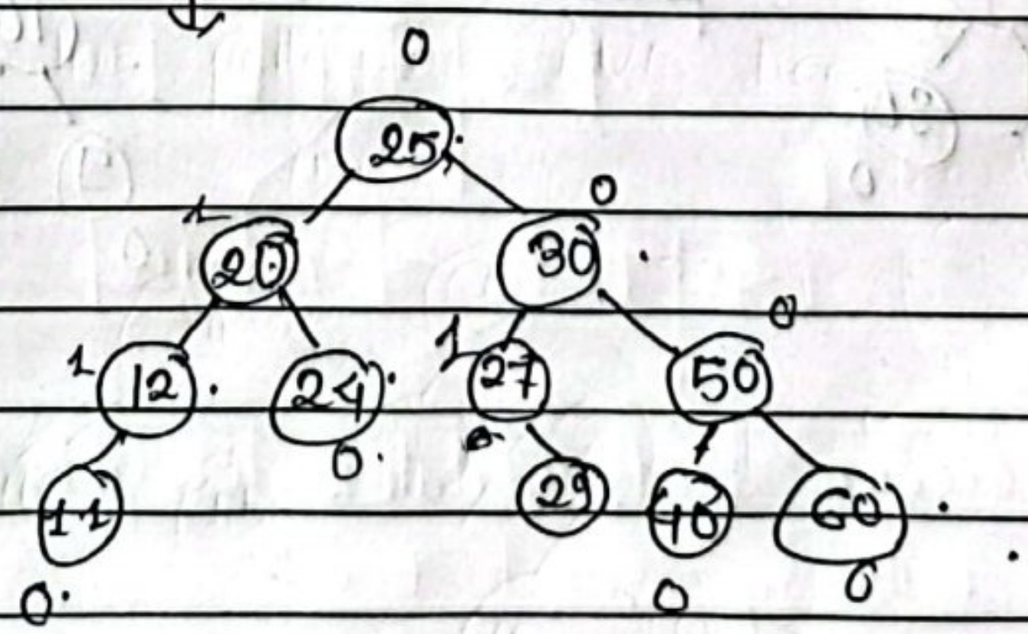


Insert 29.


30
20
50
12
24
40
60
11
25
27
29

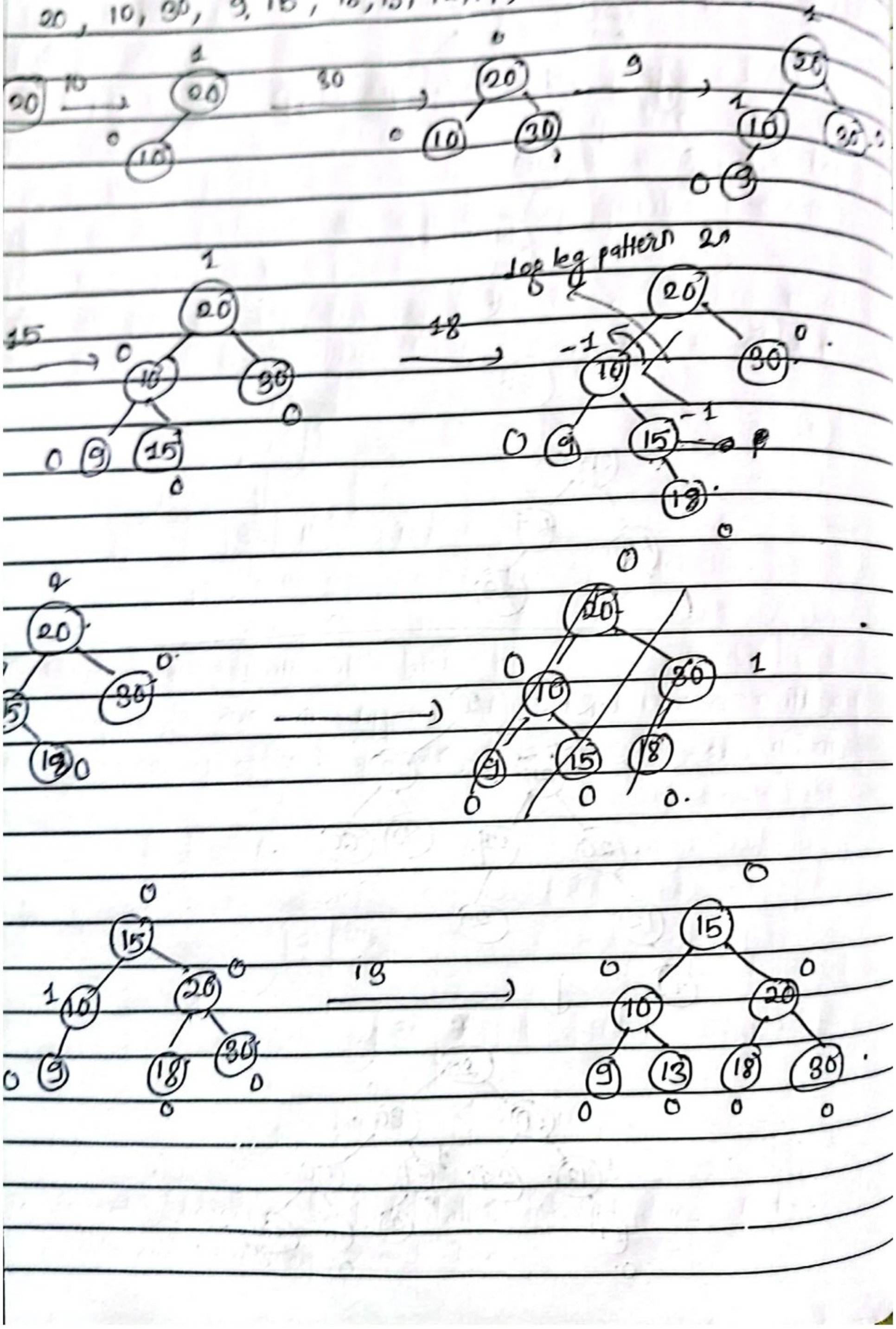

30
25
27
24
29


SRR
30
25
50
20
24
40
60
12
29
11

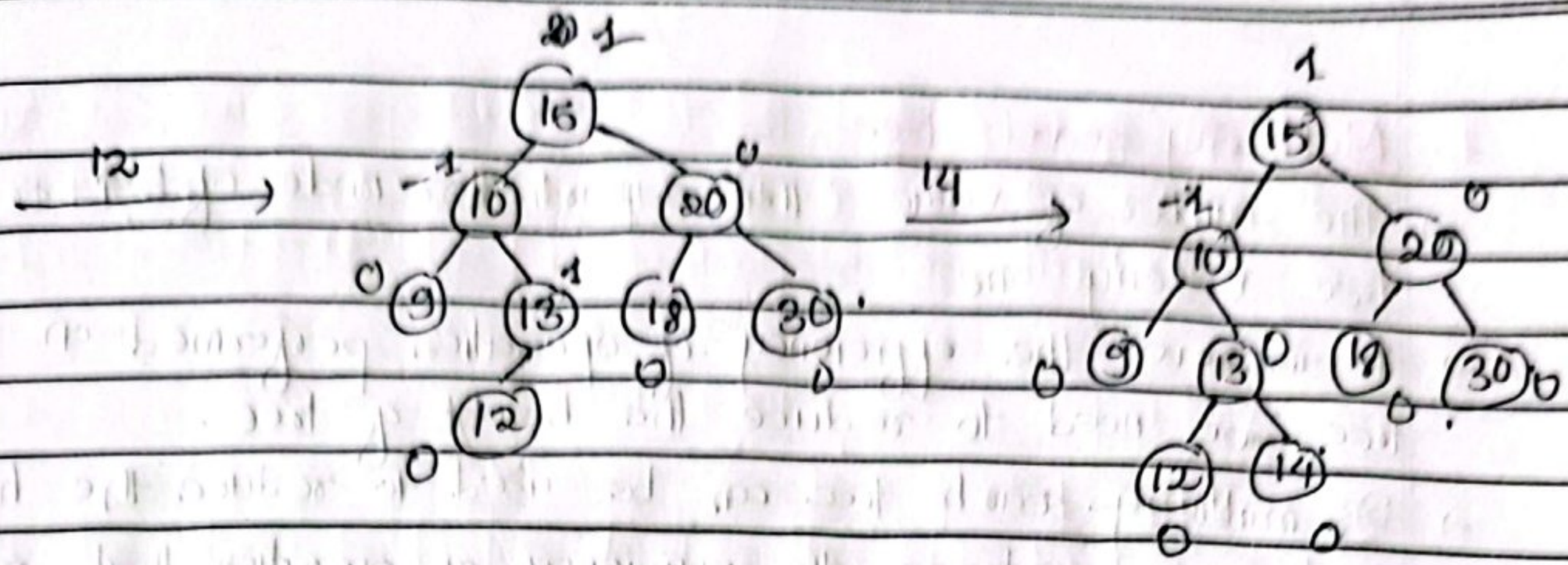

0
25
1
0
20
30
1
0
1
12
24
27
50
0
0
11
29
40
60
0
0
0

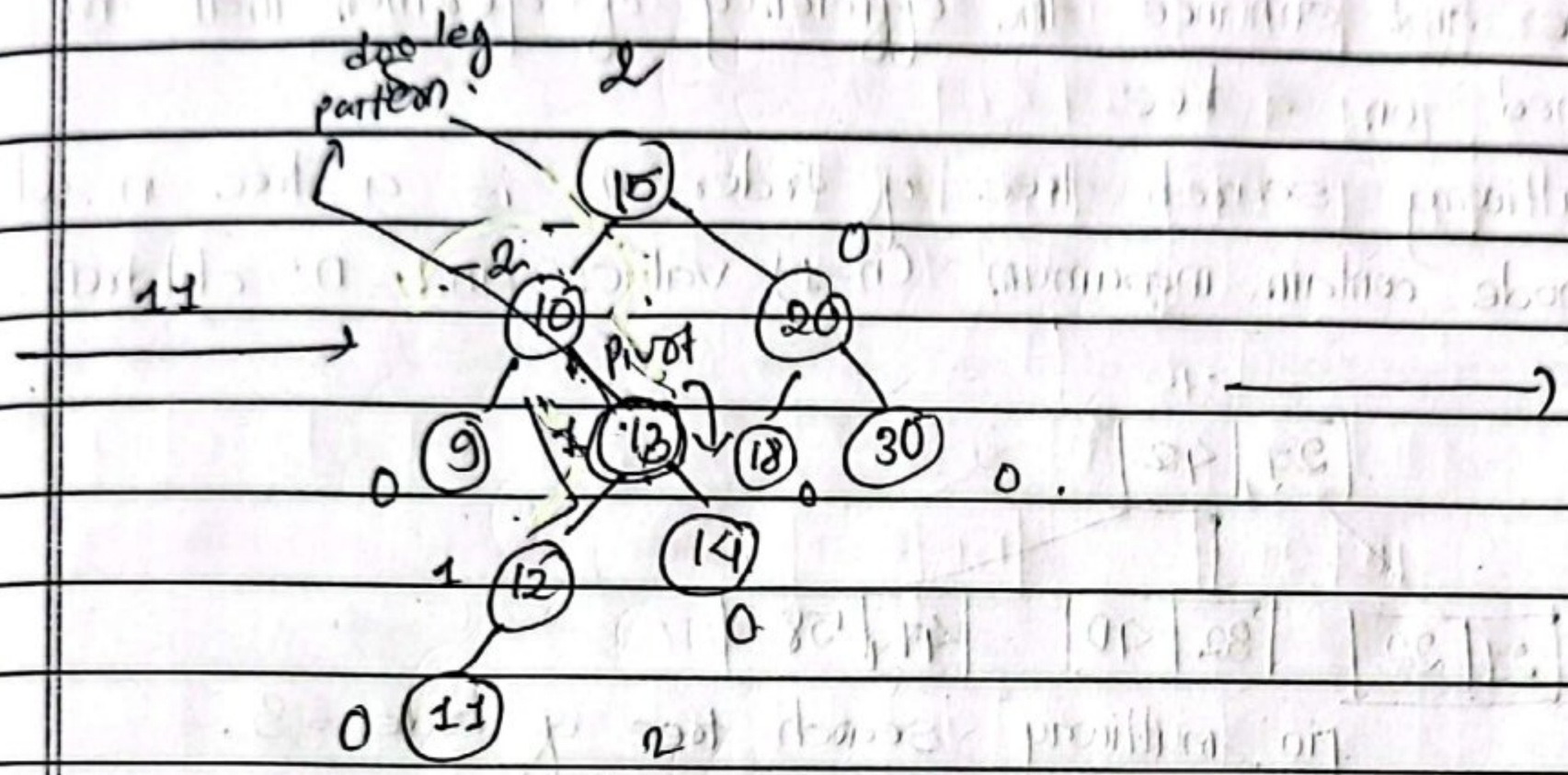
Construct AVL tree for following data.

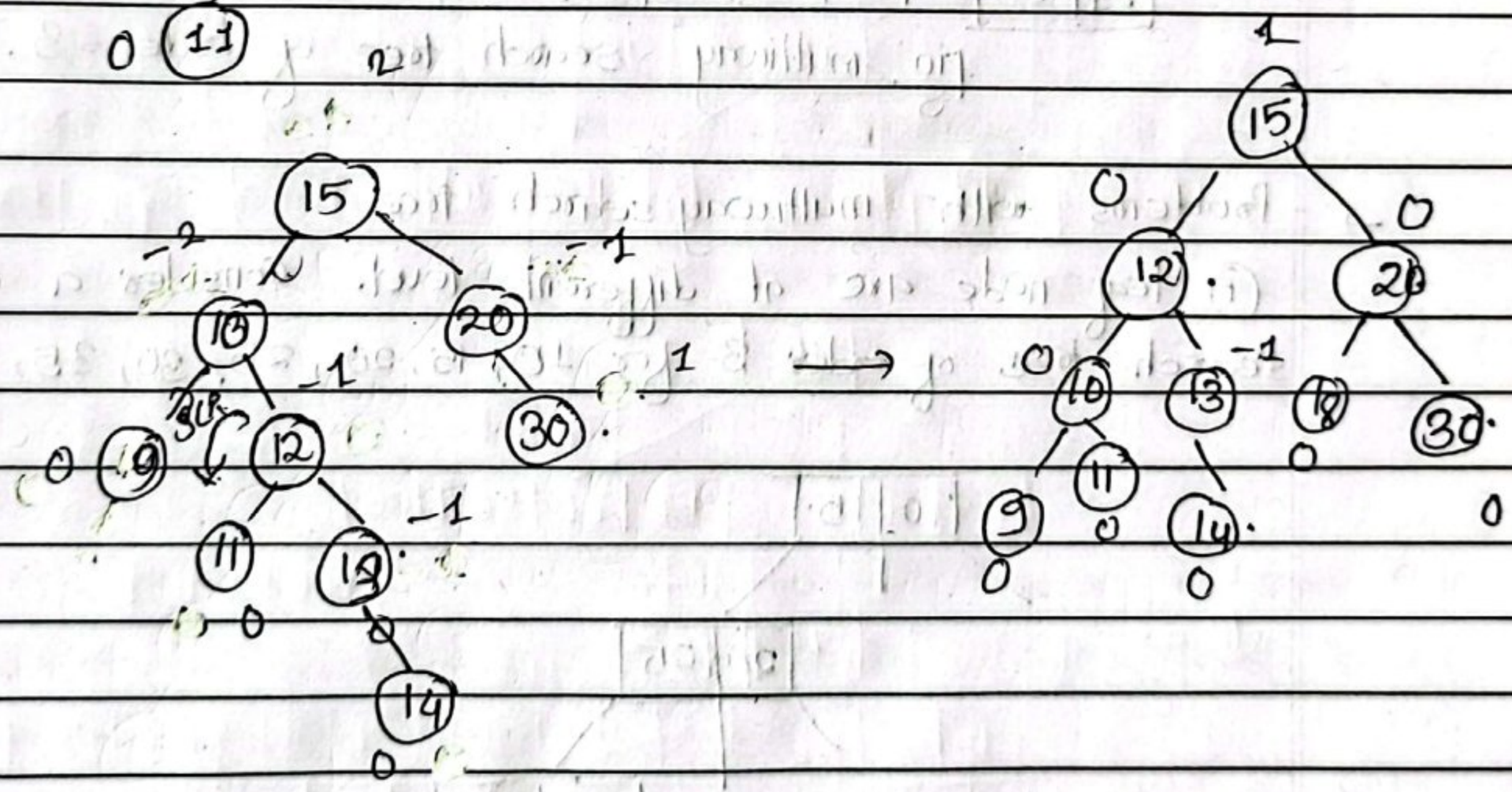
20, 10, 30, 9, 15, 18, 13, 12, 14, 11



Right to left RDLR
left to right DLR







Multiway search tree

→ The number of values that a particular node of BST or AVL tree is only one

→ To improve the efficiency of operation performed on a tree, we need to reduce the height of tree.

→ So, multiway-search tree can be used to reduce the height of tree and enhance the efficiency of operation that are performed on a tree.

→ A multiway search tree of order 'n' is a tree in which any node contain maximum (n-1) values and 'n' children.

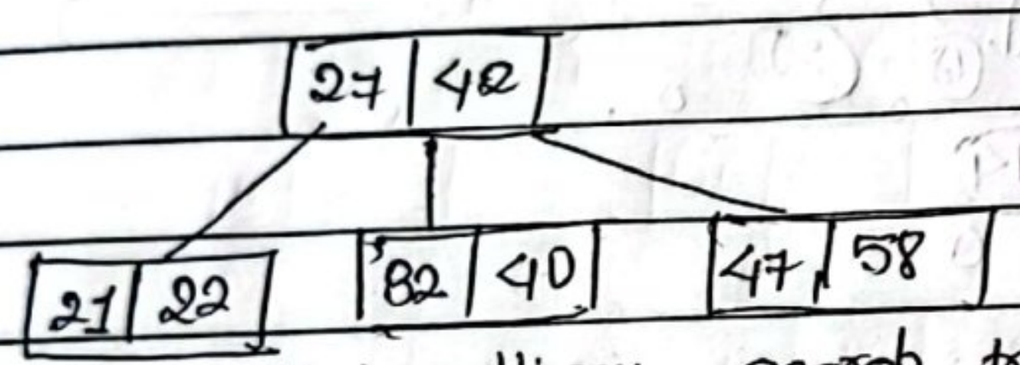
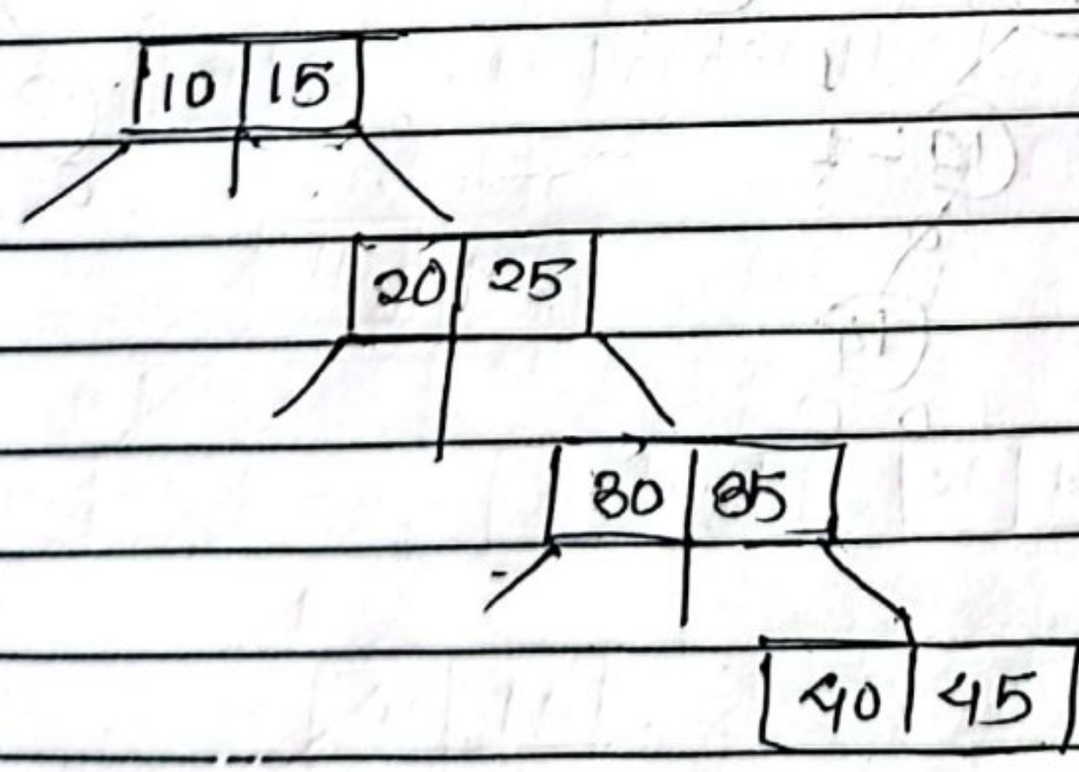


fig: multiway search tree of order - 3.

→ Problems with multiway search tree

(i) leaf node are at different level. Consider a multiway search tree of order 3 for 10, 15, 20, 25, 30, 35, 40, 45.



→ To ~~search~~ overcome such limitation we introduce B-tree which is a balanced multiway search tree.

CamScanner

Date
Page

B - Tree

→ B-tree of order 'm' is a multiway search tree in which.

* All leaf node are at same level.
* All the node except root have at least m/2 children and at most 'm' - children.
* All the node except root have at least (m-1)/2 keys & at most (m-1) keys.
* when a key (new data) is to be inserted into a full node it is split into two nodes and the key with median value is inserted in parent node.
* All the values (key) in a particular node are in inserting order.

Algorithm to insert node in B-Tree

→ Traverse the B-tree until a leaf node suitable for insertion is reached.

→ we have two cases for inserting the key.

(i) Node is not full

→ Insertion is done in the key node in increasing order of keys.

(ii) Node is full

→ Split the node into two node at its medium at same level and push the median element up by one level in the parent node.

→ If the parent node is not full, accomodate the element otherwise repeat the same procedure in case (ii).

CamScanner
CamScanner

Construct B-tree of order-5 with keys given below:

10, 20, 50, 60, 40, 80, 100, 70, 130, 90, 30, 120, 140, 25, 35, 160, 180

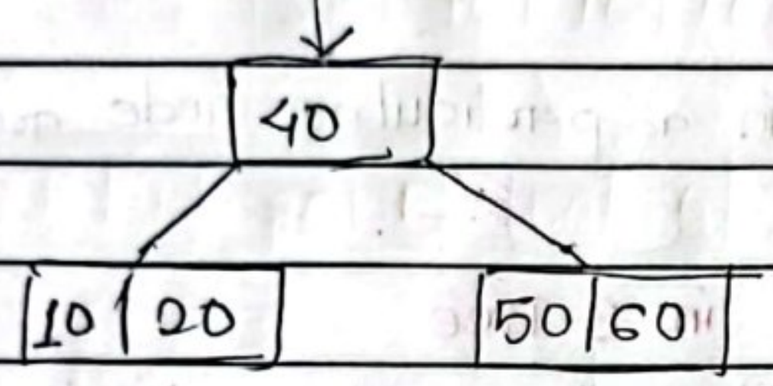
10 → [10]

20 → [10 | 20]

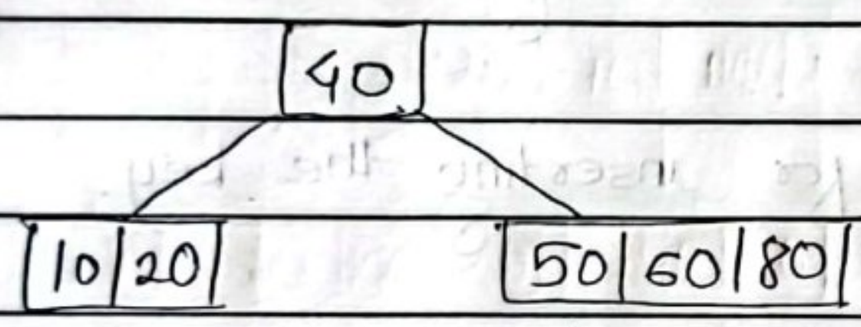
50 → [10 | 20 | 50]

60 → [10 | 20 | 50 | 60]

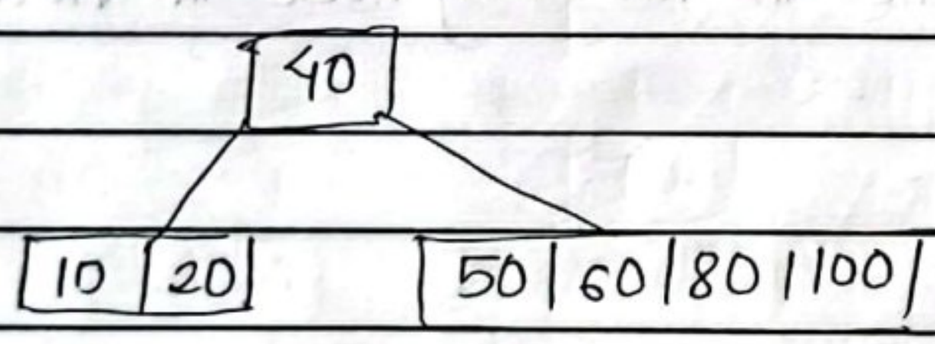
40 → [10 | 20 | 50 | 50 | 60]



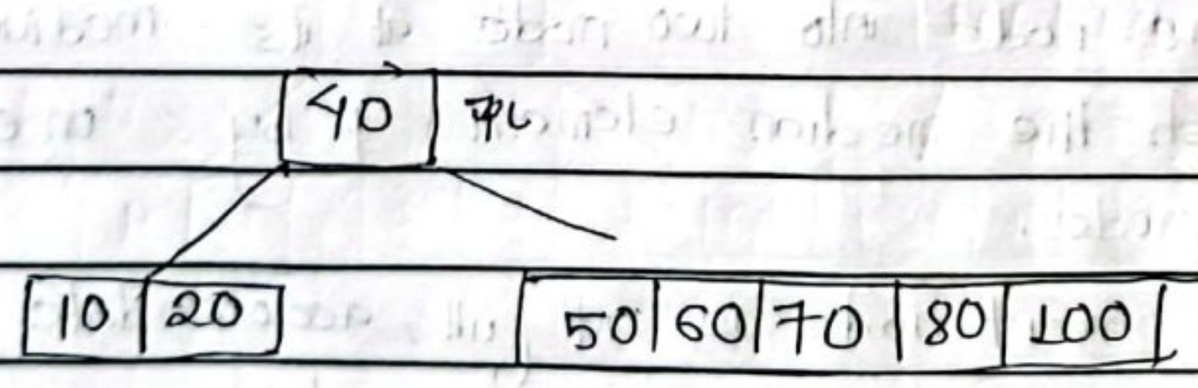
80 →

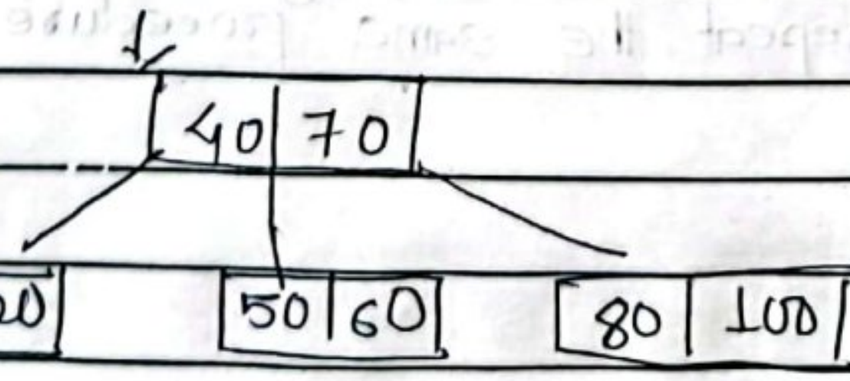


100 →

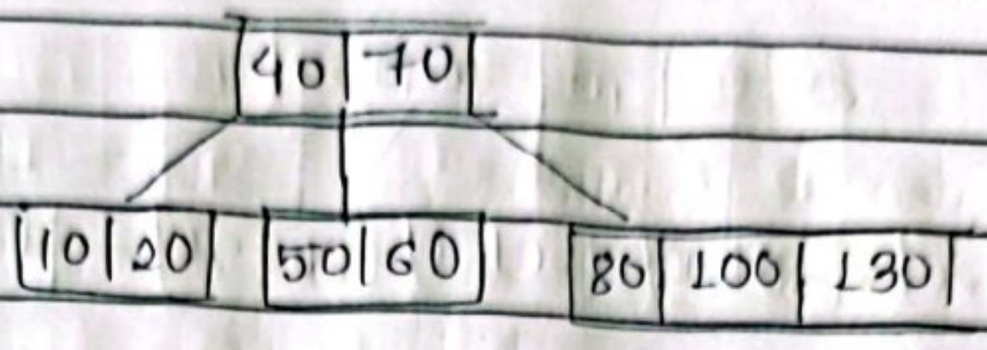


70 →

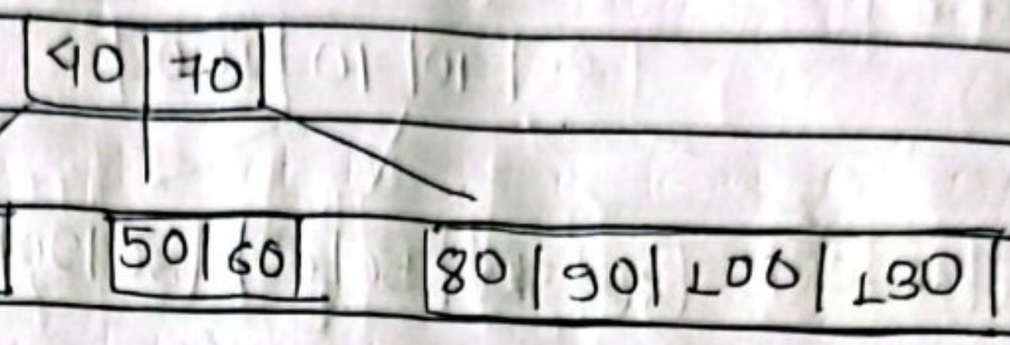




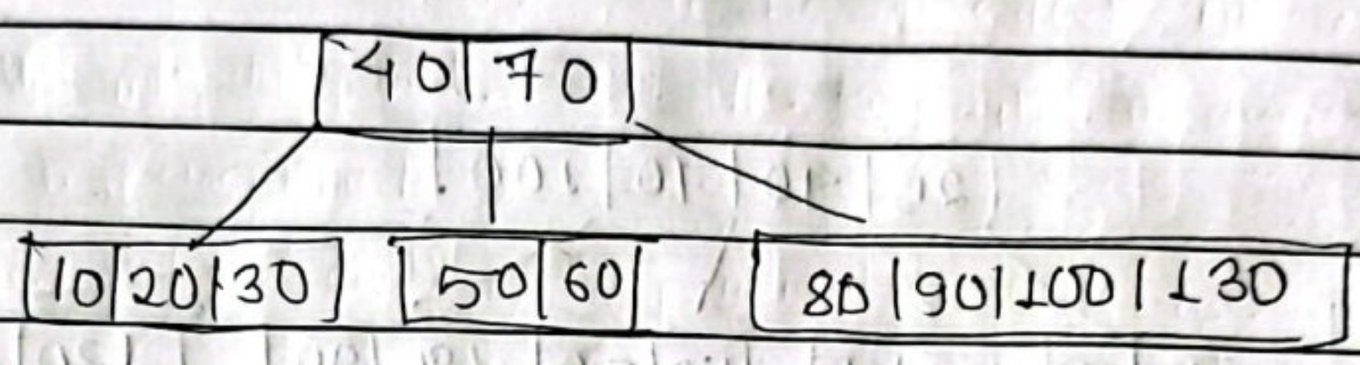
130 →



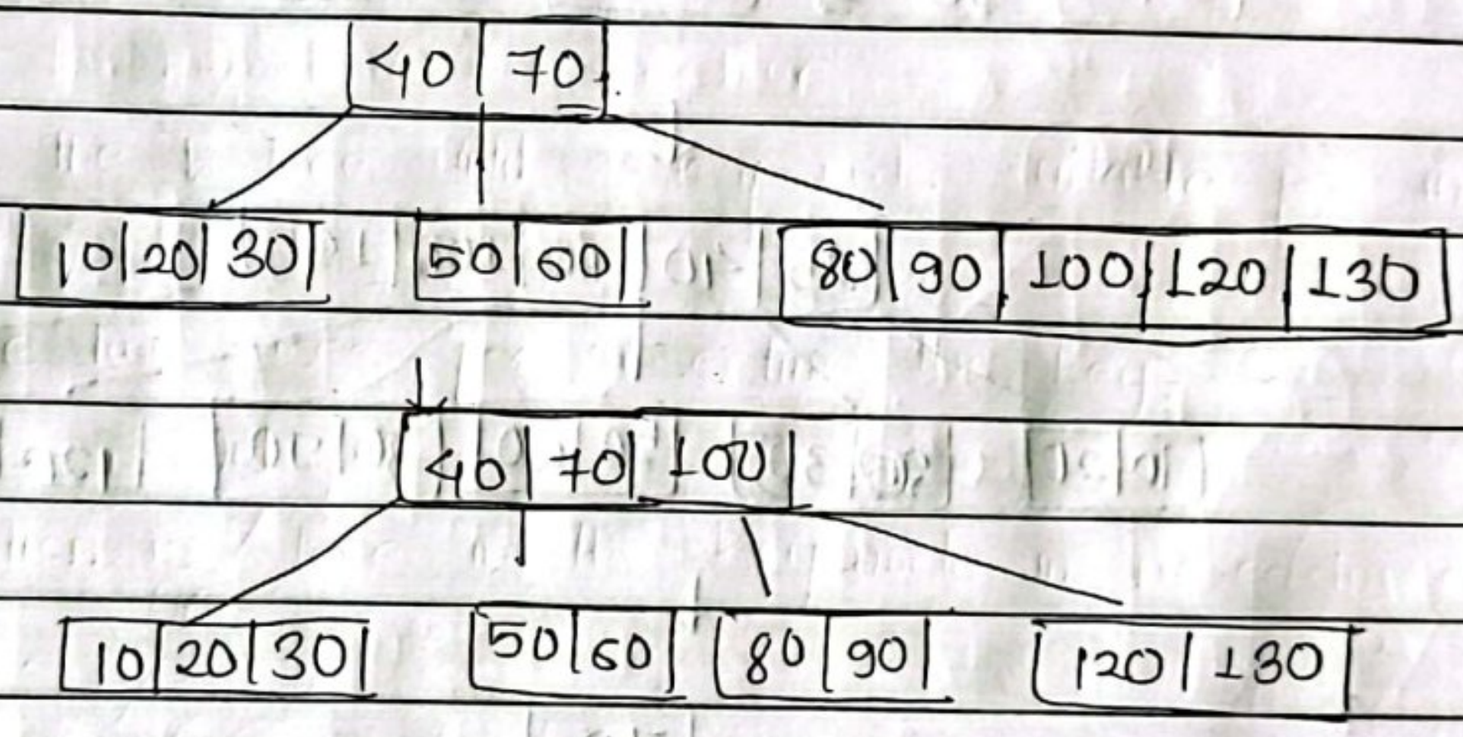
90 →



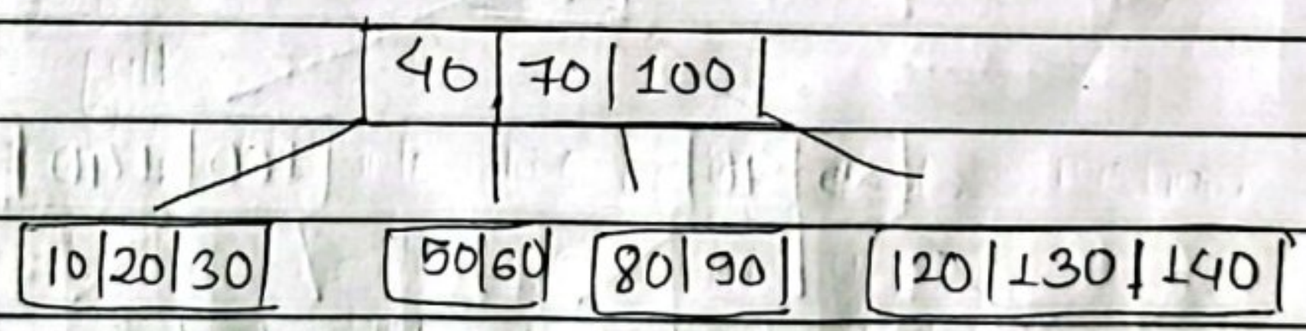
30 →



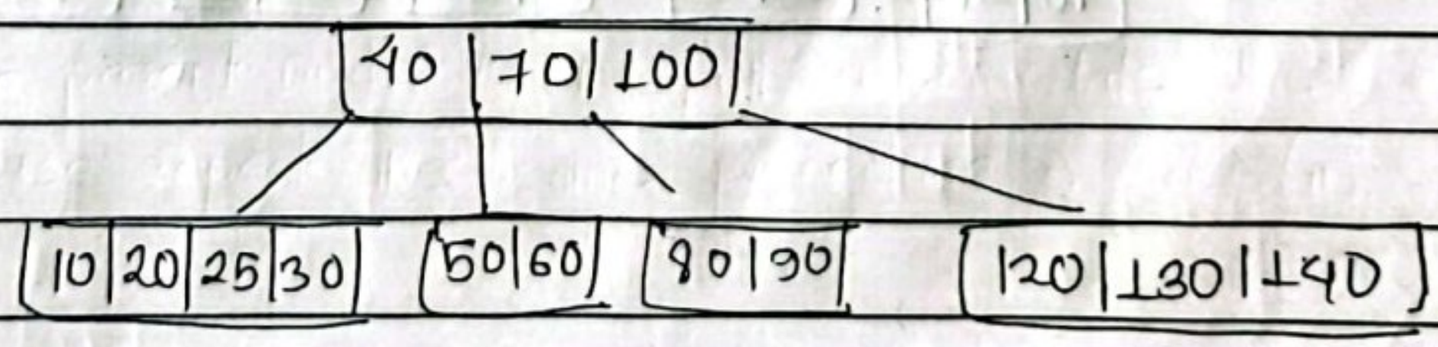
120 →



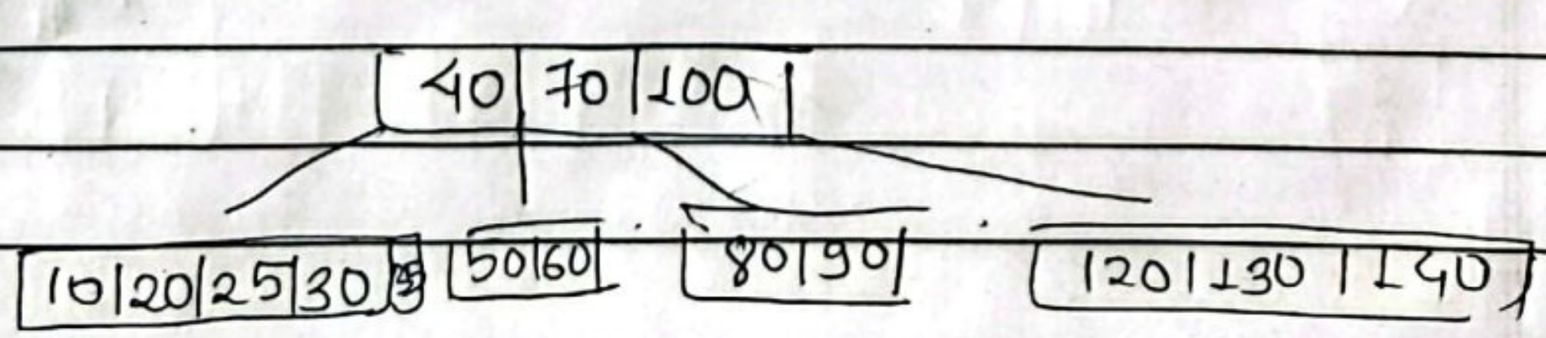
140 →

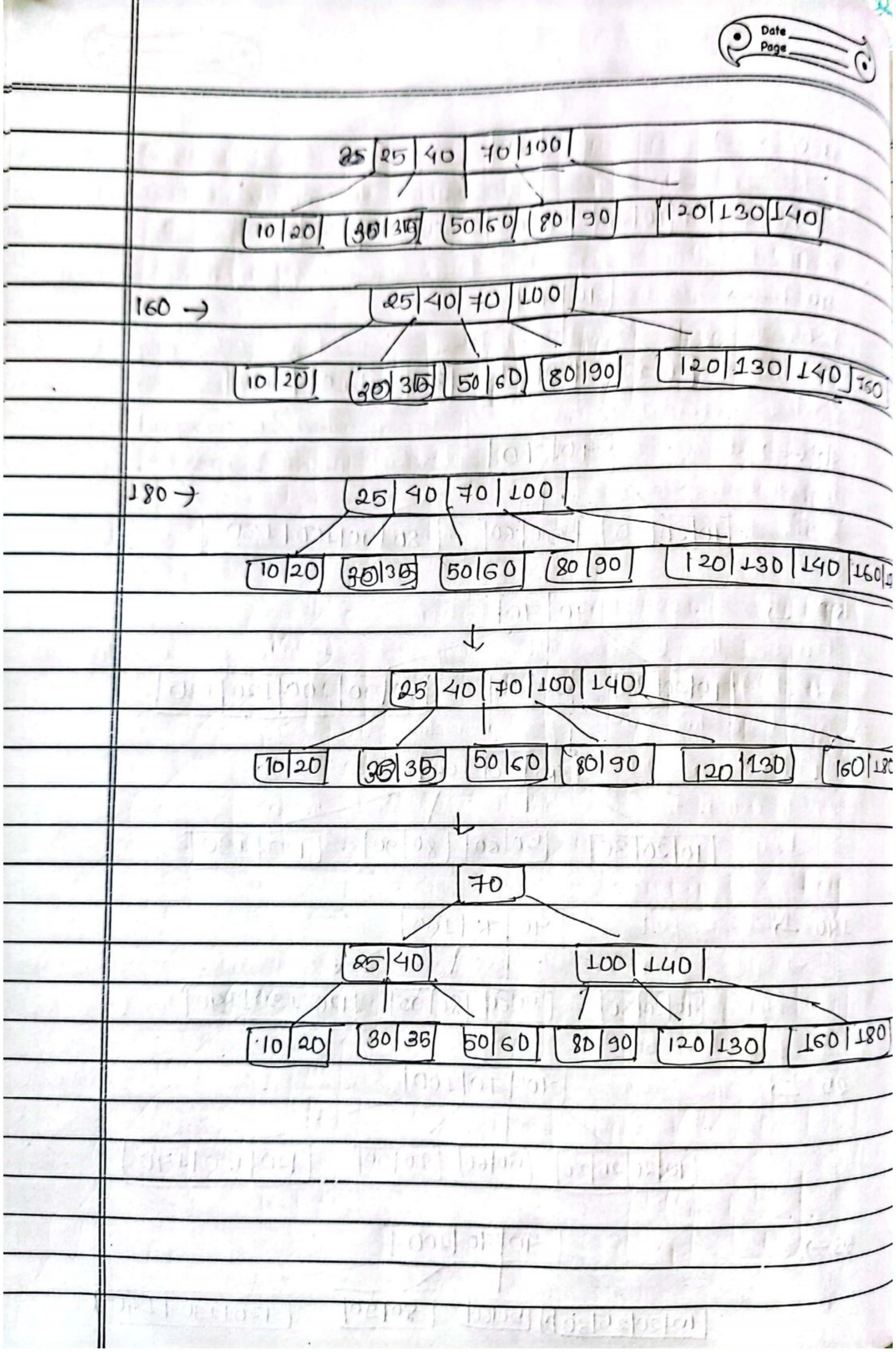


25 →



65 →




25 | 40 | 70 | 100
10 | 20
30 | 35
50 | 60
80 | 90
120 | 130 | 140
160 →
25 | 40 | 70 | 100
10 | 20
30 | 35
50 | 60
80 | 90
120 | 130 | 140 | 160
180 →
25 | 40 | 70 | 100
10 | 20
30 | 35
50 | 60
80 | 90
120 | 130 | 140 | 160
25 | 40 | 70 | 100 | 140
10 | 20
30 | 35
50 | 60
80 | 90
120 | 130
160
70
25 | 40
100 | 140
10 | 20
30 | 35
50 | 60
80 | 90
120 | 130
160 | 180

Building Huffman tree using Huffman Algorithm

→ A special type of tree called Huffman tree can be used to generate the code for the variable length coding sentence.

→ In Huffman tree, each leaf node is labelled with symbol.

→ A binary code can be assigned to each symbol as follows:-

* Associate a 0 to path from a node to its left sub-tree.
* Associate 1 to path from a node to its right sub-tree.

→ Now code for each symbol is obtained by following the path from root to the leaf node containing the symbol.

Algorithm

(a) Get the string

(b) Get the frequency of occurance of each symbol and arrange in ascending order.

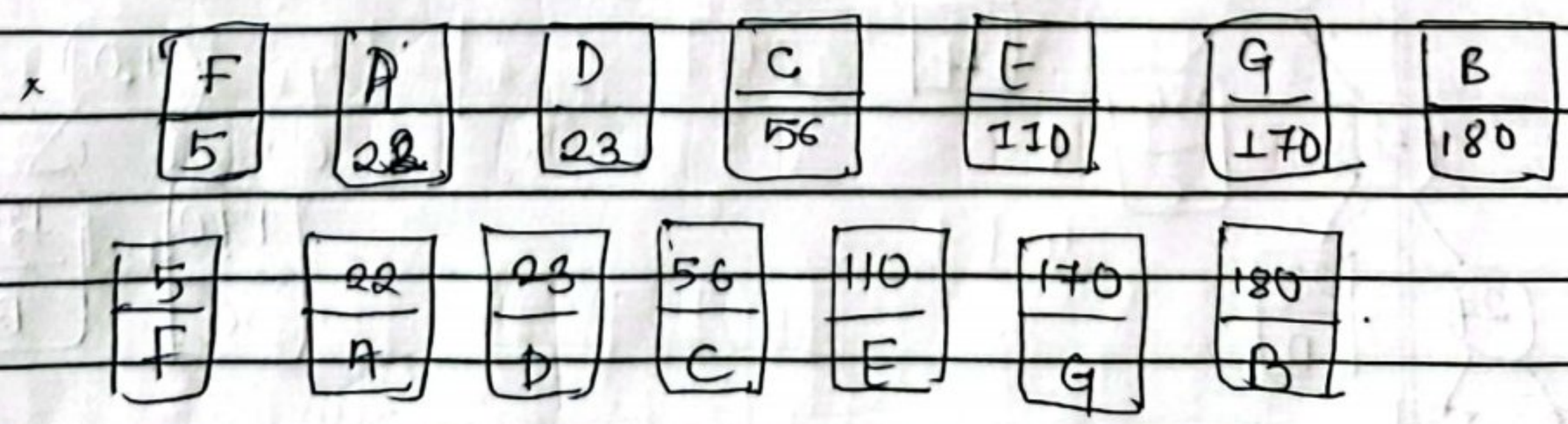
(c) Select two symbol having least probability & combine into single.

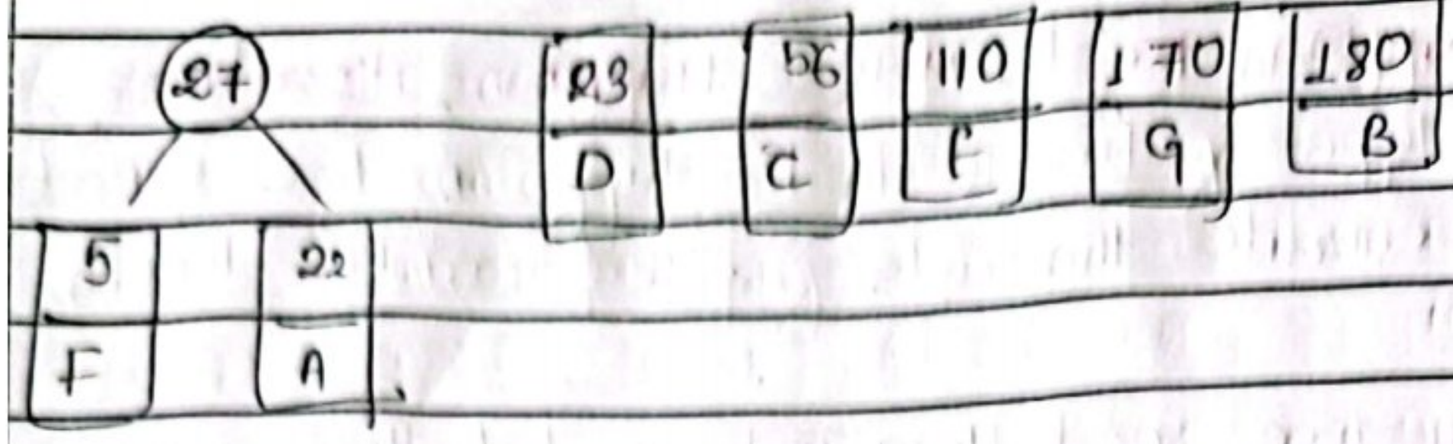
(d) Repeat (c) until first symbol is reached.

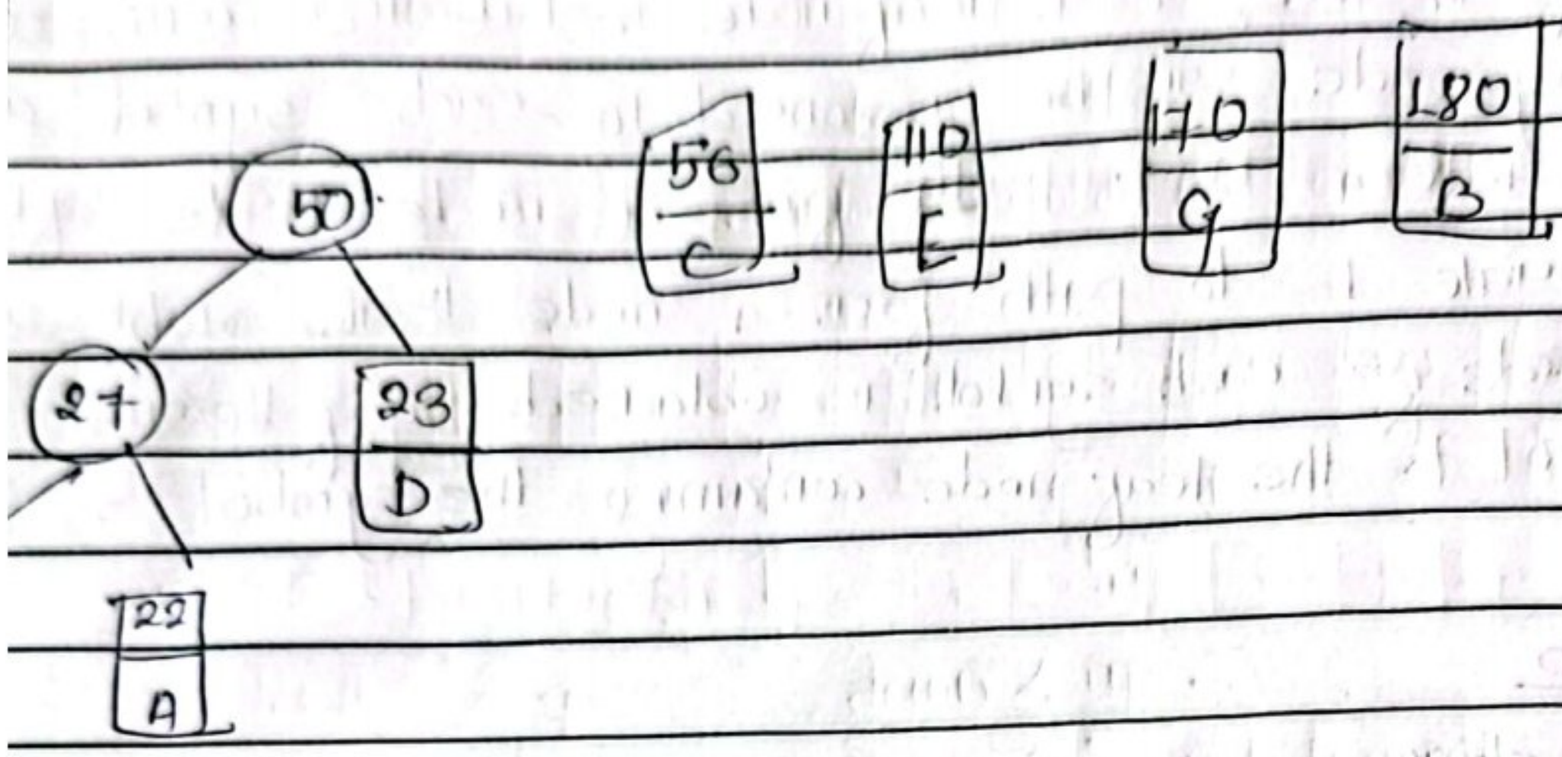
Using the following table generate Huffman tree and Huffman code for each symbol.

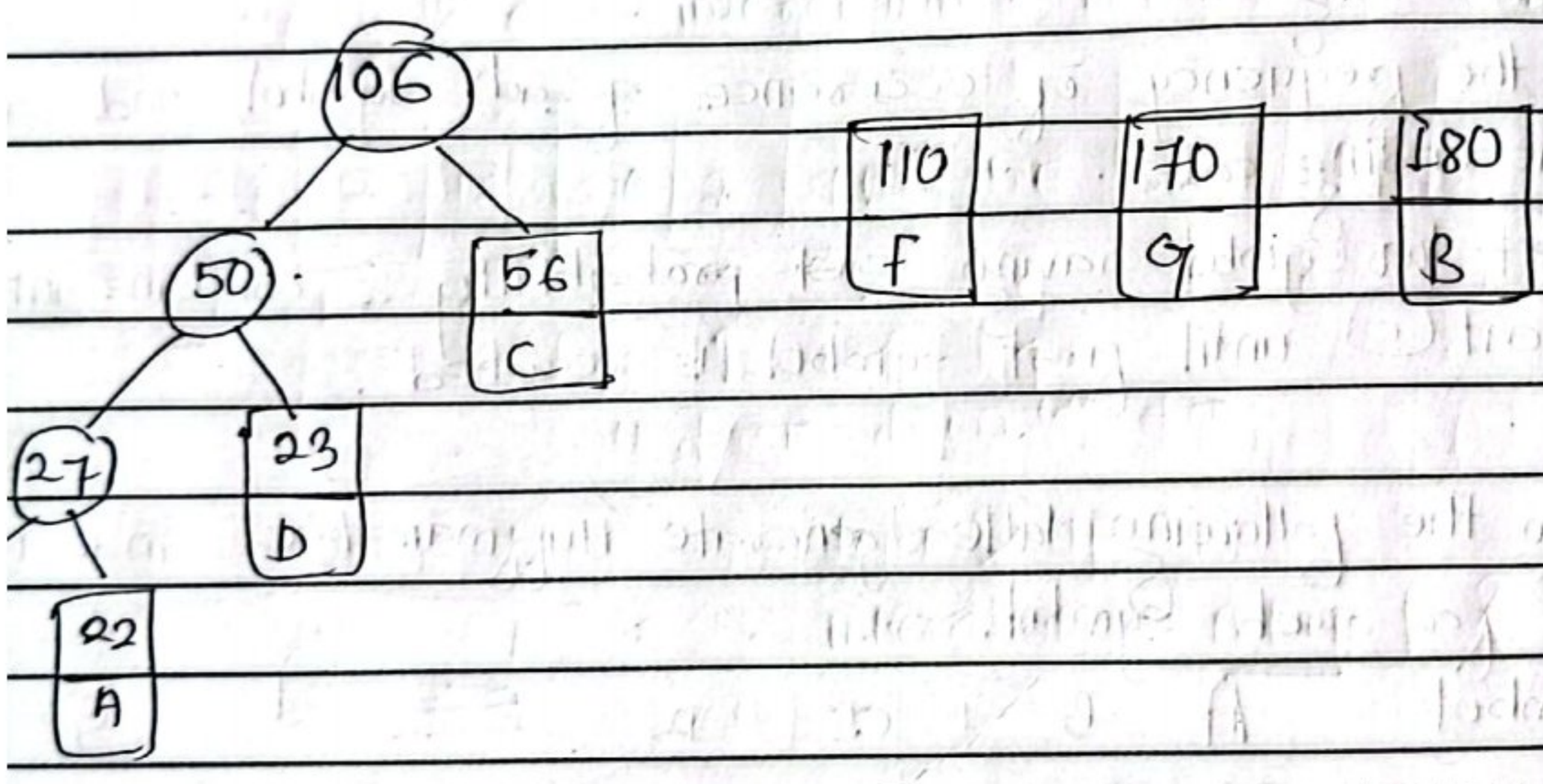
symbol	A	B	C	D	E	F	G
frequency	22	180	56	23	110	5	170

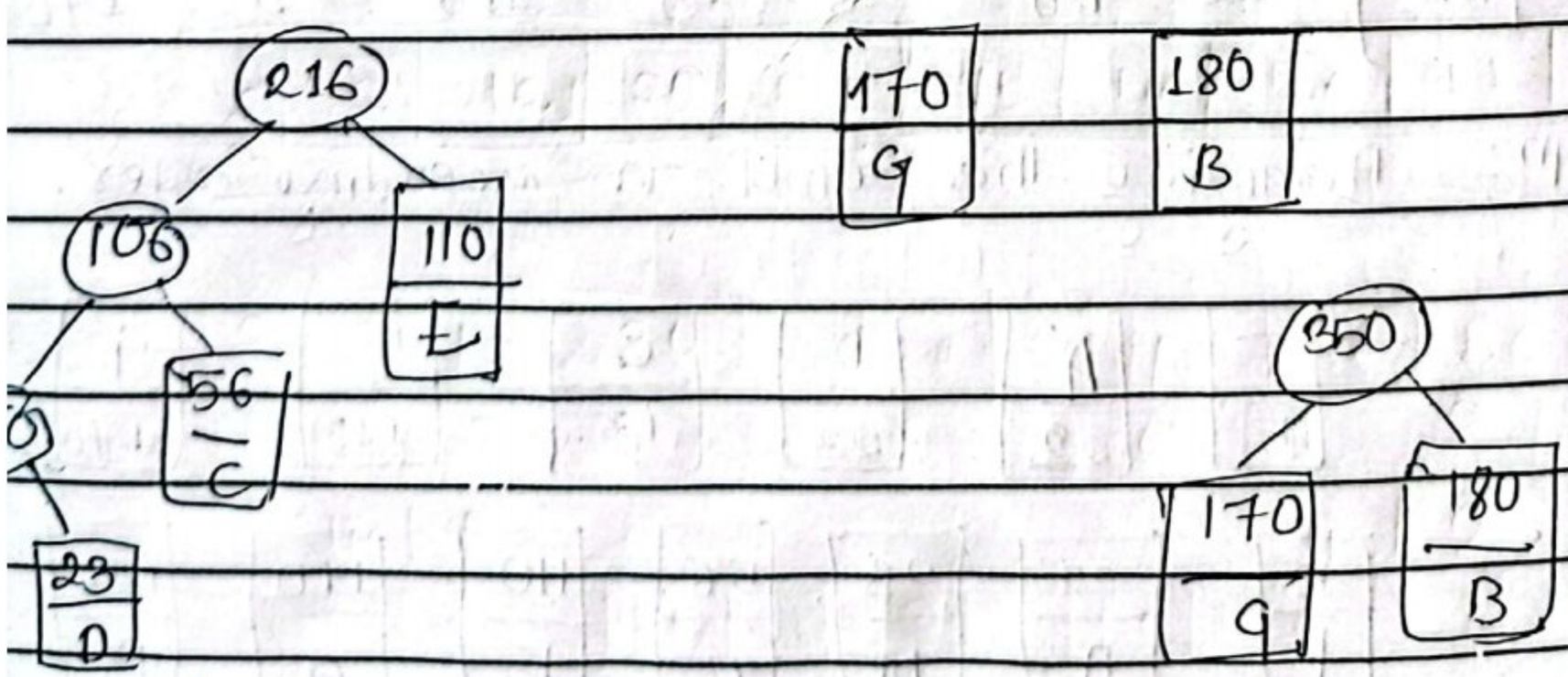
Soln:- Arranging the symbol in ascending order.




27
23
D
56
C
110
F
170
G
180
B
5
F
22
A


50
56
C
110
E
170
G
180
B
27
23
D
22
A


106
110
F
170
G
180
B
50
56
C
27
23
D
22
A


216
170
G
180
B
106
110
E
56
C
23
D
350
170
G
180
B

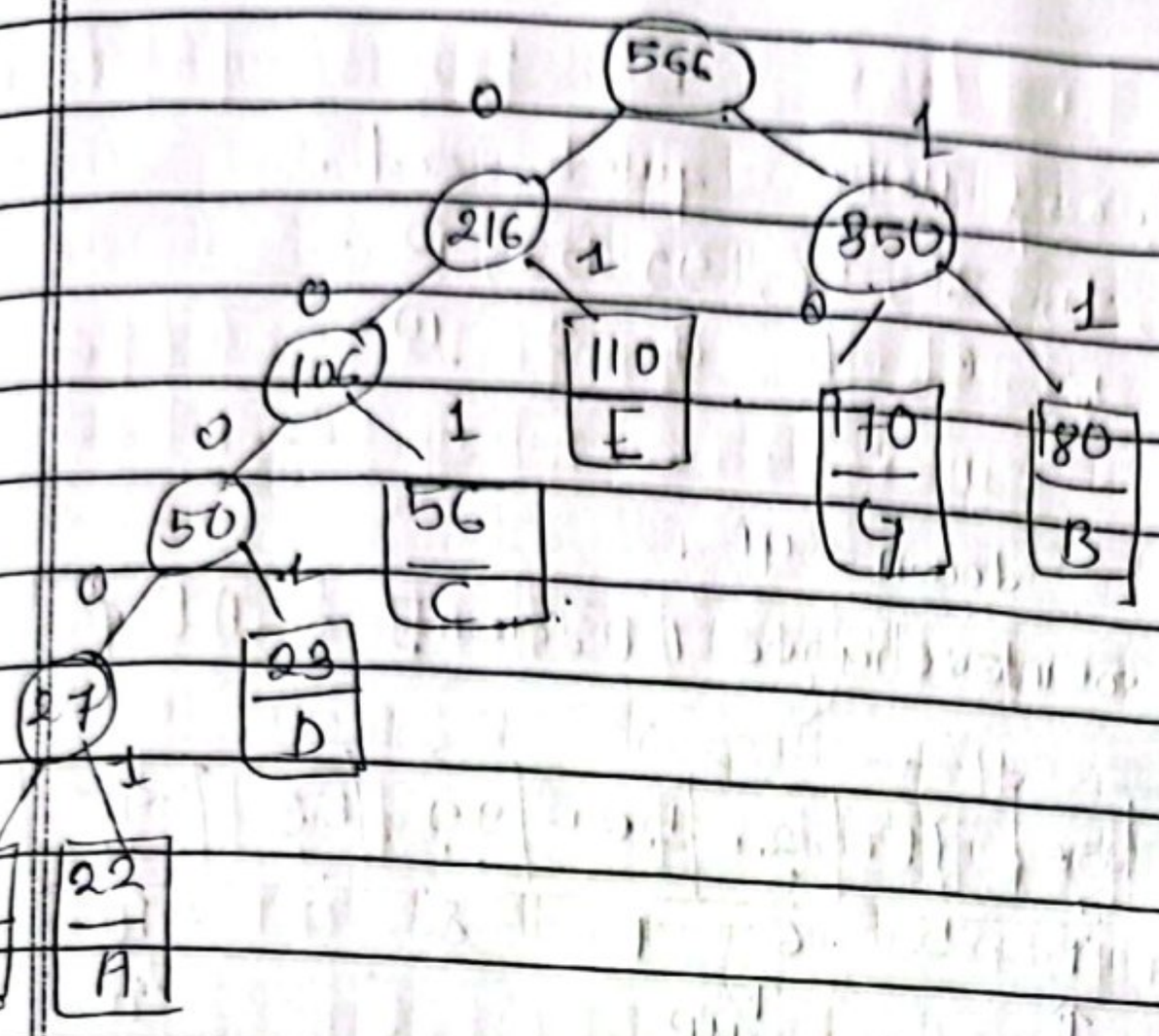


fig: Huffman Tree

Huffman code:

A : 00001
B : 11
C : 001
D : 0001
E : 01
F : 00000
G : 10.

up = N

Date
Page

Quick sort

→ Sort the following numbers using quick sort.

24, 2, 45, 20, 56, 75, 2, 56, 99, 53, 12.

0	1	2	3	4	5	6	7	8	9	10

key = 24

down = ~~0~~ 2 down < up.

up = 10 so interchange A[down] & A[up]

24	2	12	20	56	75	2	56	99	53	45
0	1	2	3	4	5	6	7	8	9	10

up

down = ~~2~~ 4 down < up

up = ~~10~~ 6 interchange A[down] & A[up]

24	2	12	20	2	75	56	56	99	53	45
0	1	2	3	4	5	6	7	8	9	10

up down down up

down = 5 down > up

up = 4 interchange A[up] & key

2	2	12	20	24	75	56	56	99	53	45

2	2	12	20
0	1	2	3

list 1

24

75	56	56	99	53/45
0	1	2	3	4 5

list 2.

CamScanner

list 1.

2	2	12	20
0	1	2	3
down			up

key = 2.

down → 2. down > up.

up → 1

interchange A[up] & key.

2	2	12	20	⇒	2	2	12	20
					list 11		list 12.	

Take list 12.

12	20
0	1
down	up

key 12

down = 1

up = 0.

int.

12	20
list 121	list 122..

Taking list 2.

75	56	56	99	53	45
0	1	2	3	4	5.

key = 75

down = ~~0~~ 3

up = 5

down < up.

interchange A[down] & A[up].

75	58	56	45	53	99
0	1	2	3	4	5

down = ~~3~~ 5 down > up

up = ~~5~~ 4 interchange A[up] & A[down] key

53	56	56	45	75	99

⇓

53	56	56	45	75	99

list 21 list 22.

Taking list 21

key = 53

53	56	56	45
0	1	2	3
down			up

down = 1 up < down

up = 0 interchanging A[up] & A[down].

53	45	56	56
0	1	2	3

down = 2 down > up

up = 1 interchanging A[up] & key

45	53	56	56
0	1	2	3

⇓

45	53	56	56

list 211 list 212

Date
Page

Taking list 210

56	56
0	1

up > down

interchange[up] & A[down]

down > up.

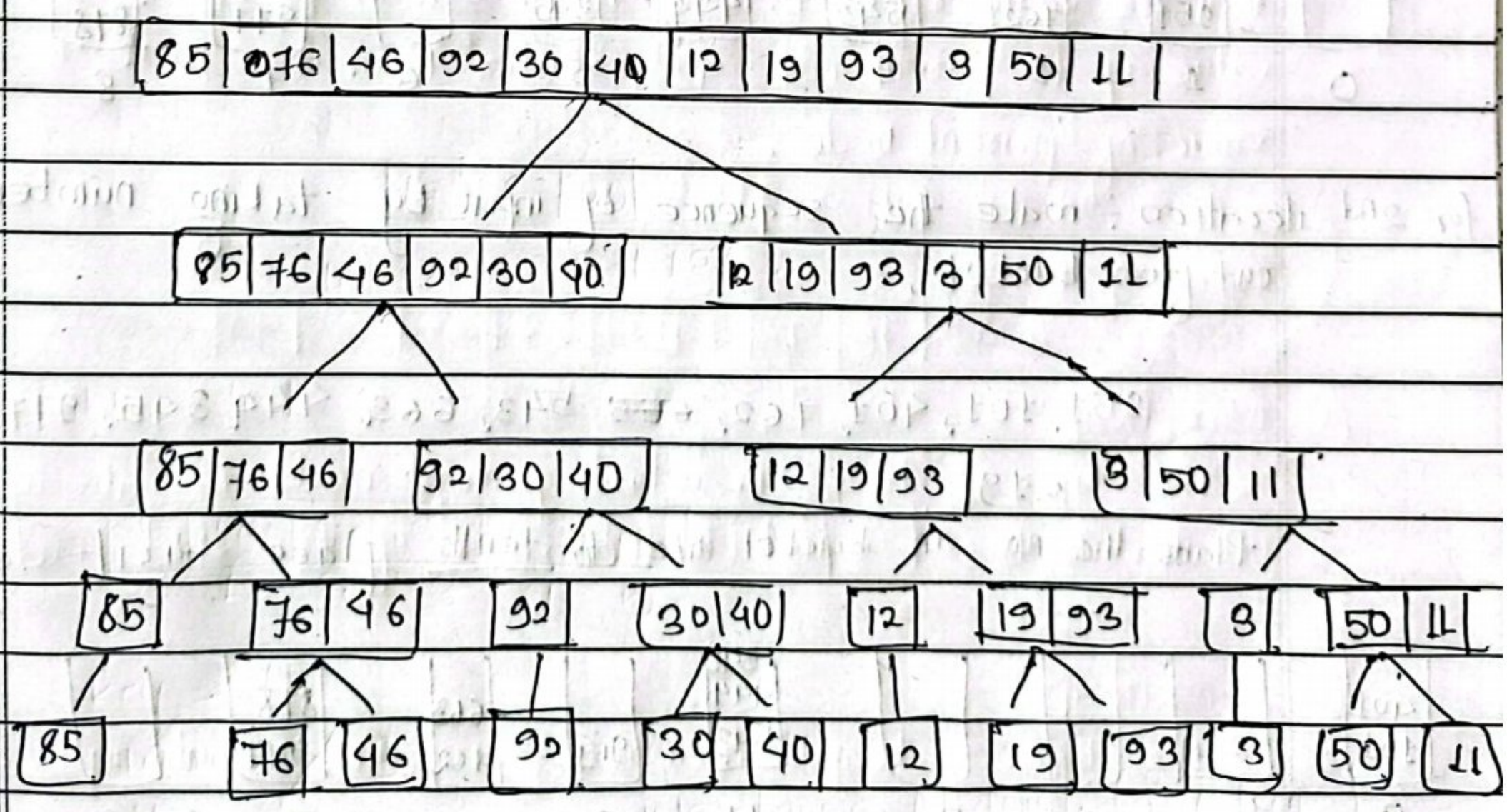
interchange key & A[up].

56	56

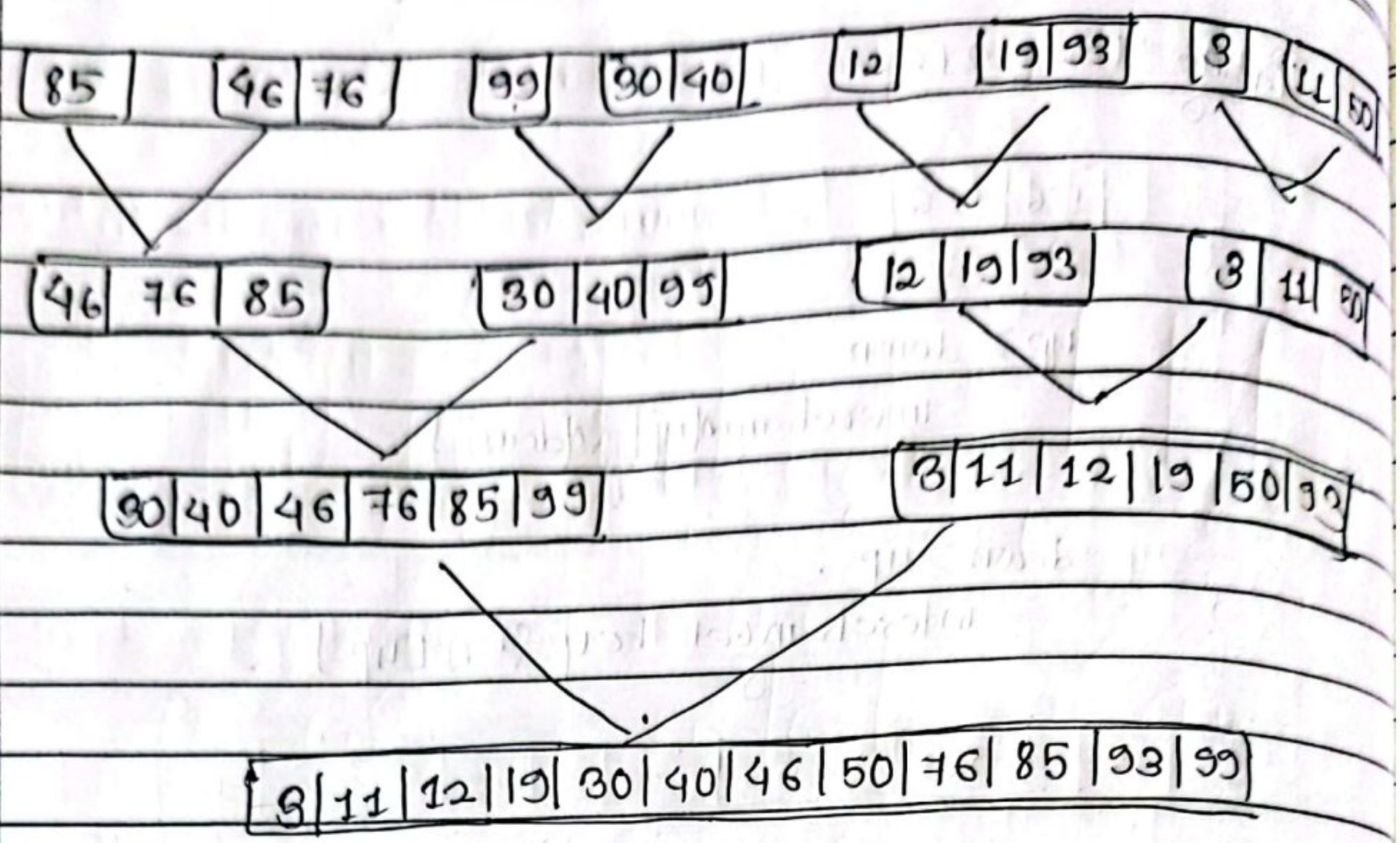
2	2	12	20	24	45	53	56	56	75	99

Merge sort.

85, 76, 46, 92, 30, 40, 12, 19, 93, 3, 50, 11.



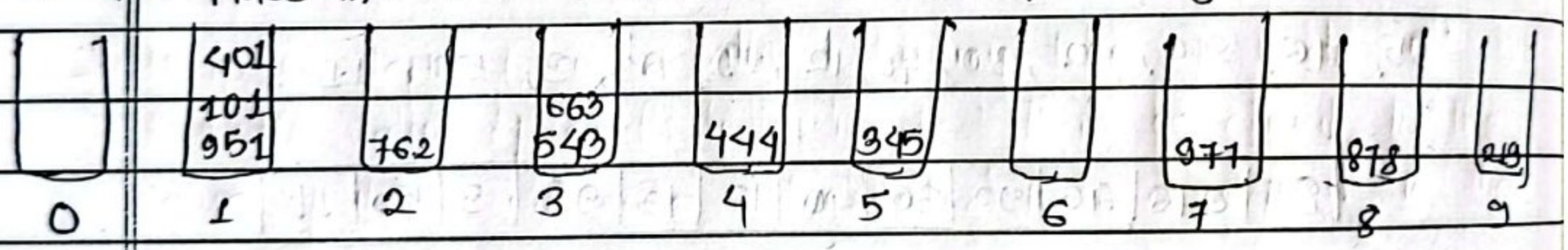
CamScanner



Radix sort

Sort: 762, 951, 101, 543, 345, 977, 219, 663, 878, 444, 401.

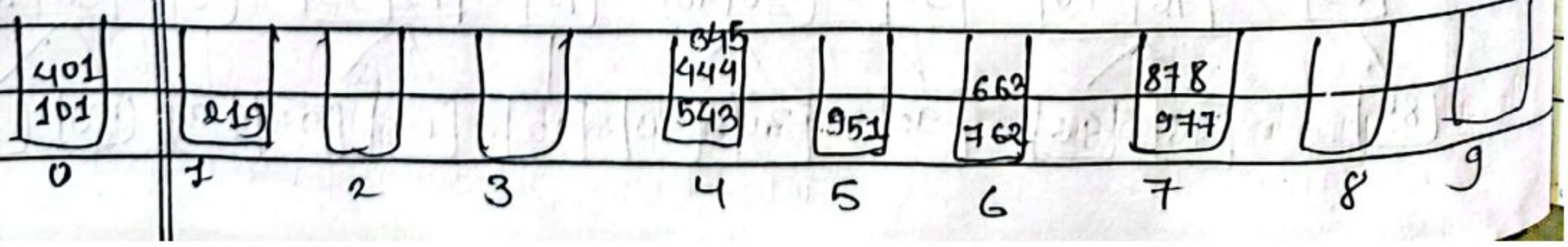
1st iteration: Place the no in bucket a/c to unit place digit.



for 2nd iteration: make the sequence of input by taking number out from bucket,

951, 101, 401, 762, ~~663~~ 543, 663, 444, 345, 977, 878, 219

Place the no in bucket a/c to tenth place digit.

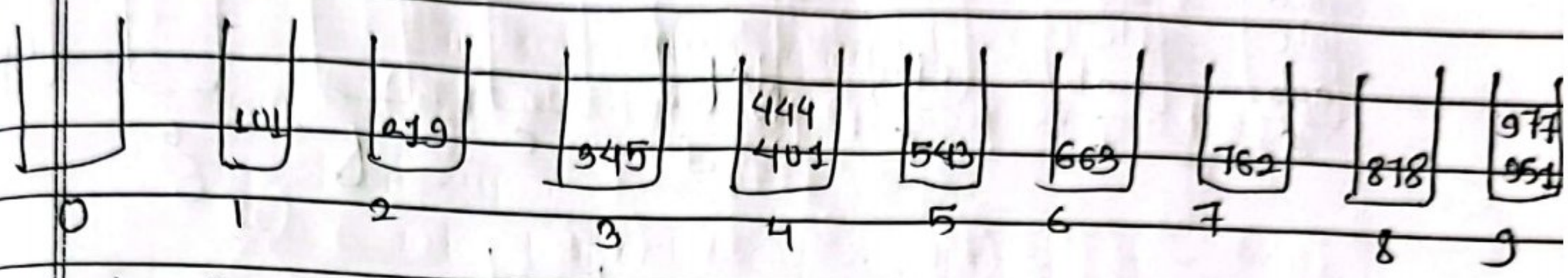


Date
Page

for 3rd iteration: make the sequence of input by taking no out from bucket.

101, 401, 219, 543, 444, 345, 951, 762, 663, 977, 878

place the no in bucket a/c to hundradth place digit.



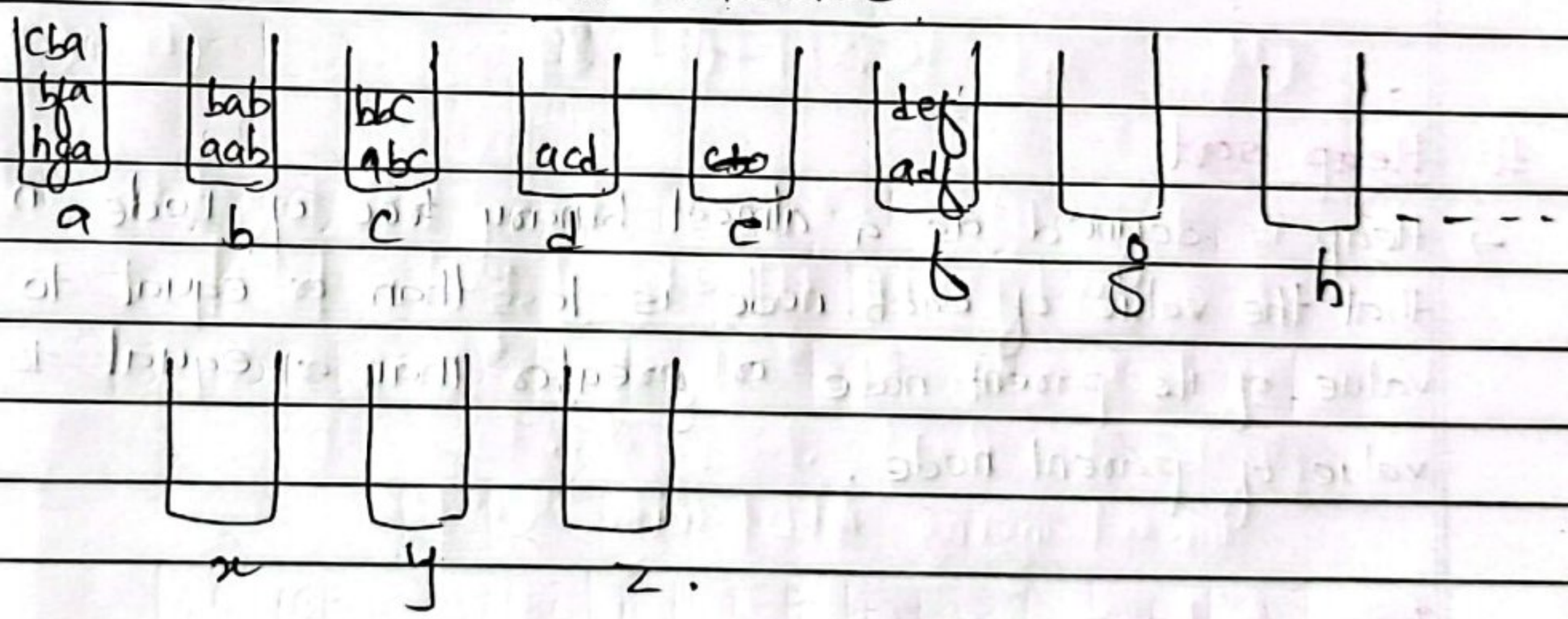
make the sequence by taking no out from bucket.

101, 219, 345, 401, 444, 543, 663, 762, 878, 951, 977

* Sort the following:

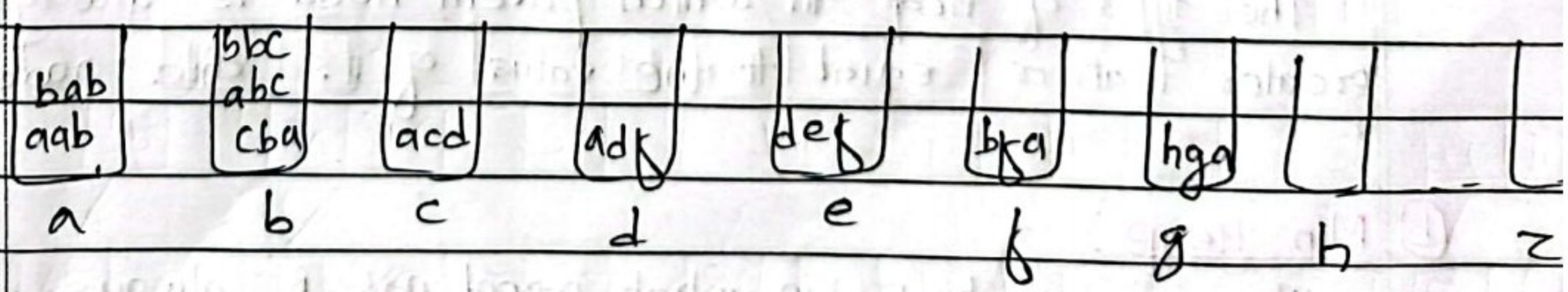
abc, aab, aac, bab, adf, acd, hga, bfa, bbc, cba, def.

last character.



hga, bfa, cba, aab, bab, abc, bbc, acd, adf, def.

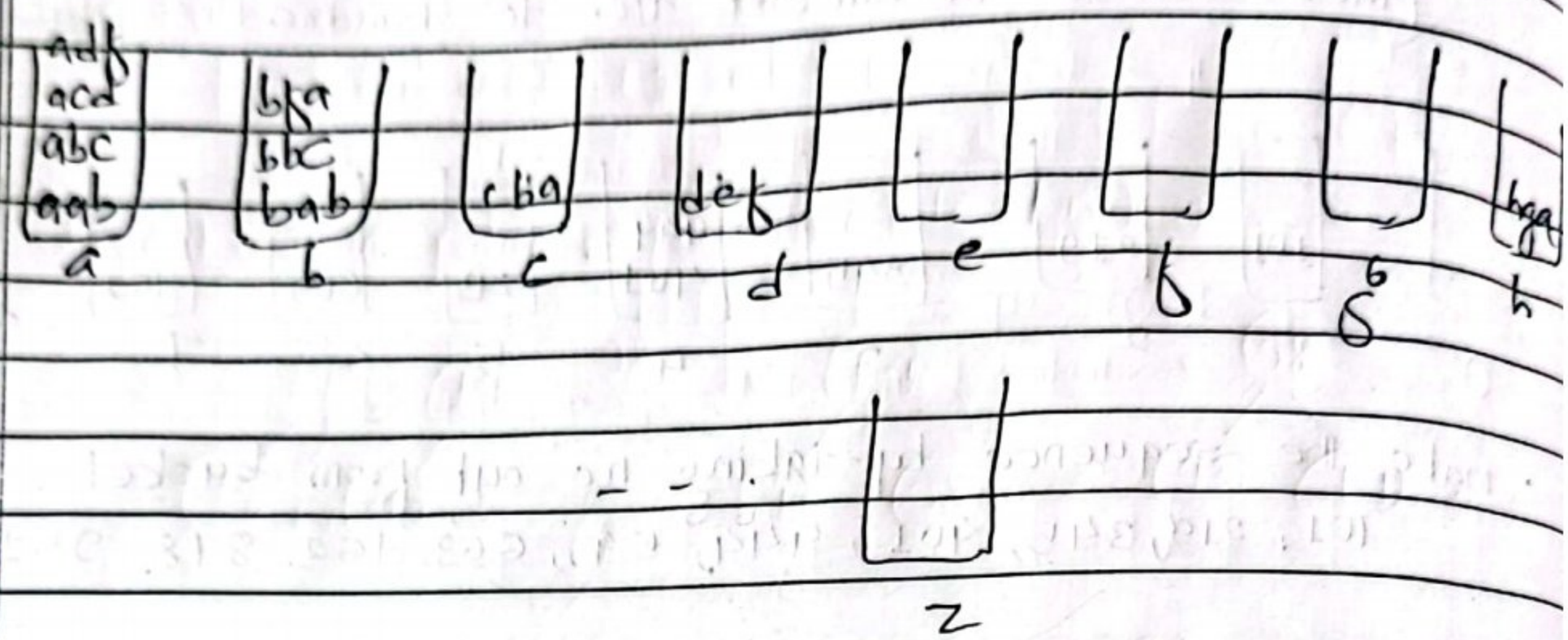
2nd iteration second last



CamScanner

aab, bab, cba, abc, bbc, acd, adf, def, bfa, hog.

3rd iteration: 3rd last string.



~~adf~~ aab, abc, acd, adf, bab, bbc, bfa, cba, def, hog.

Heap sort

→ Heap is defined as a almost binary tree of node 'n' such that the value of each node is less than or equal to the value of its parent node or greater than or equal to the value of parent node.

Type of heap

(a) **Max Heap:**

The type of heap in which parent node is always greater than or equal to the value of its child node.

(b) **Min Heap:**

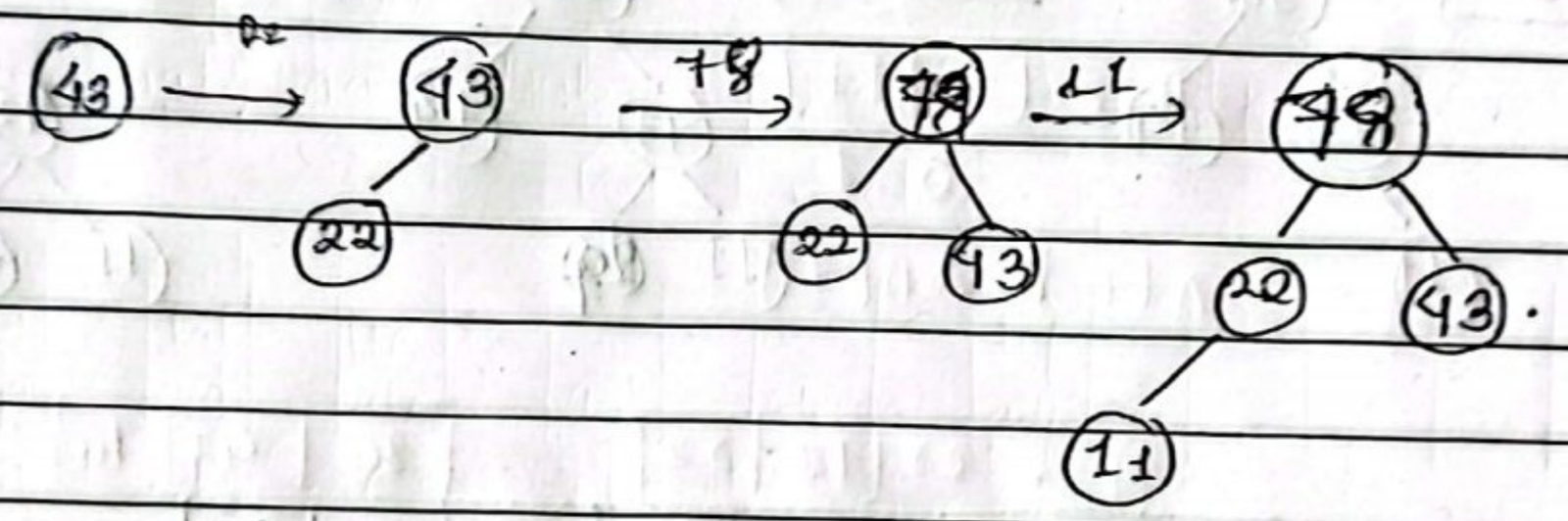
The type of heap in which parent node is always smaller

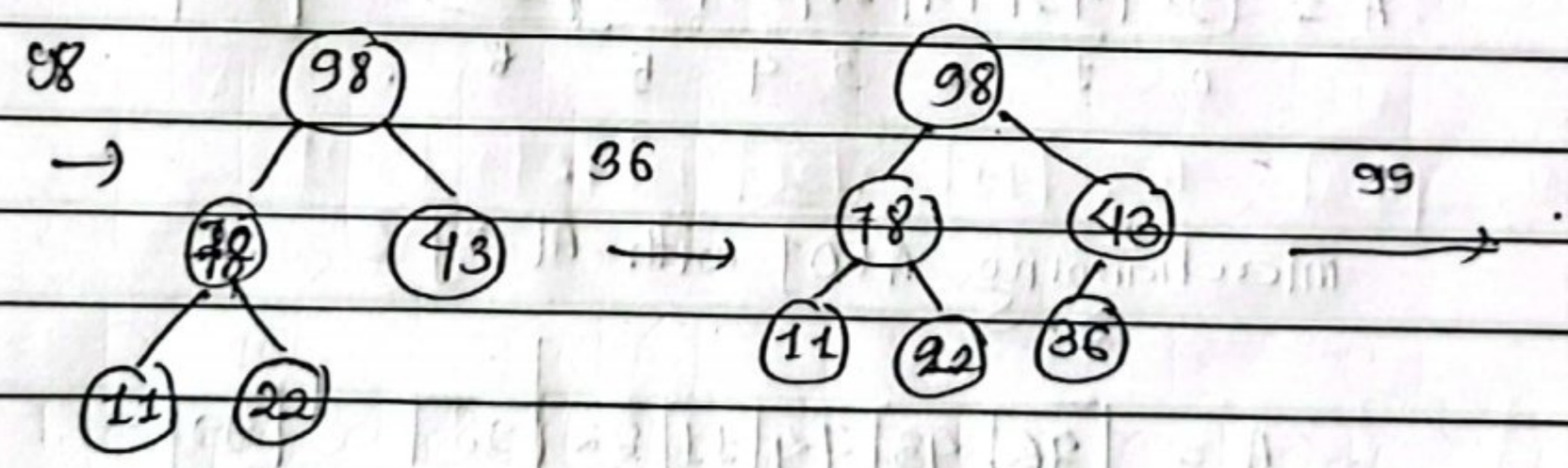
than or equal to the value of its child node.

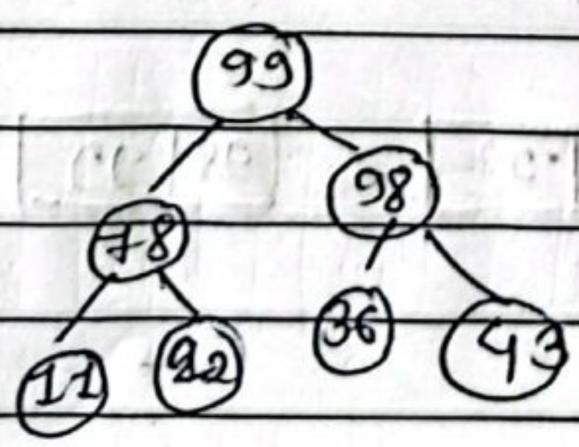
Algorithm of Heap sort (using Max heap)

(i) Put the value at missing leaf (first left and then right).
(ii) Switch with its parent if parent is smaller.
(iii) Repeat (i) & (ii) until all elements are inserted.
(iv) Switch the A[0] with A[N-1].
(v) Discard A[N-1] and repeat (i) to (iv) unit all elements are sorted.

Qn. Sort: 43, 22, 78, 11, 98, 36, 99 using heap sort (use max heap).







tree with array

initially

43	22	78	11	98	36	99

finally

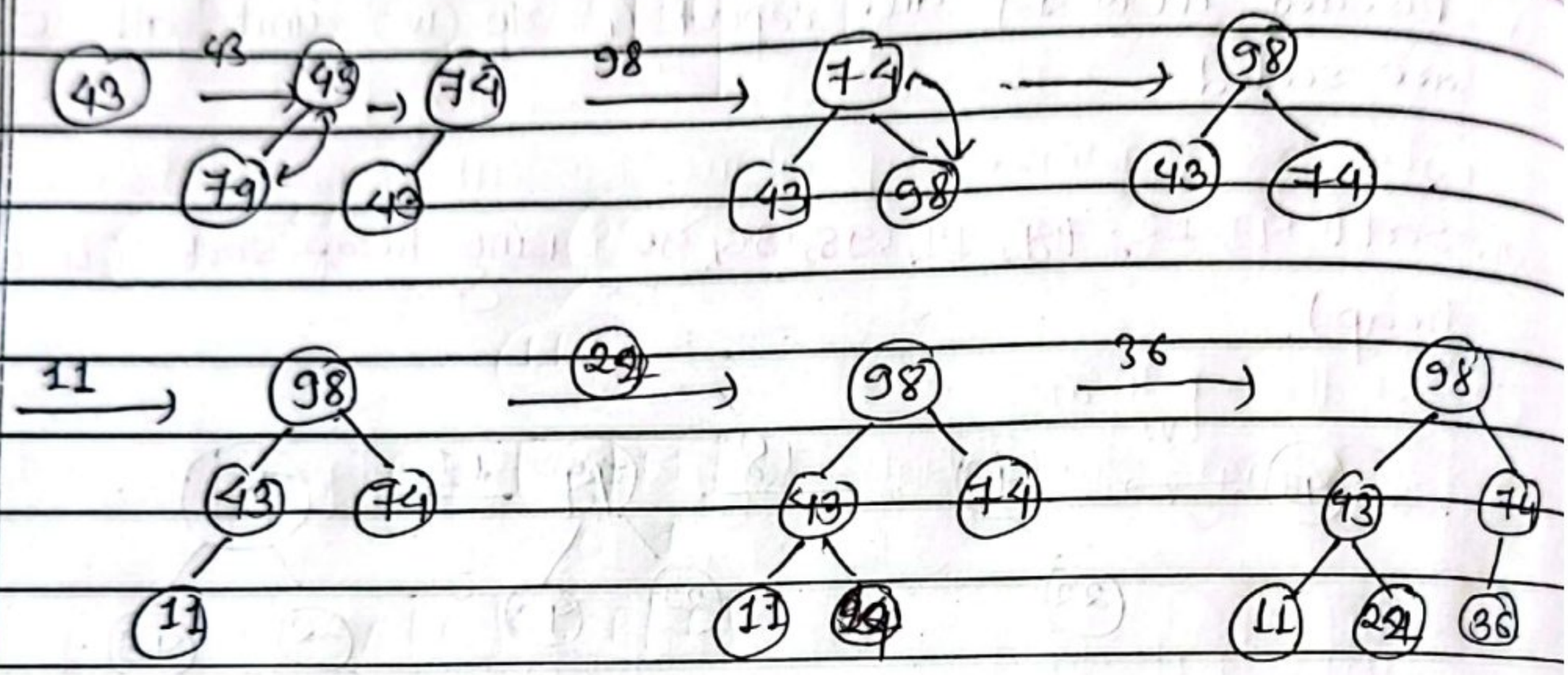
99	98	78	43	36	22	11

A =	99	79	98	11	29	36	48
	0	1	2	3	4	5	6

interchanging A[0] and A[6].

	43	79	98	11	29	36	99
	0	1	2	3	4	5	6

Repeat same process for A[0] to A[6]



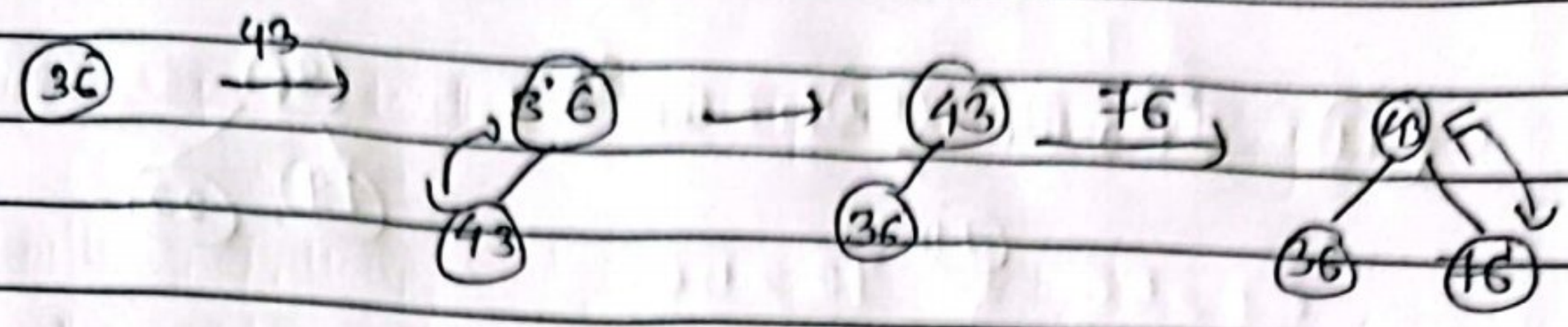
A =	98	43	74	11	22	36	~~99~~		99
	0	1	2	3	4	5	6		

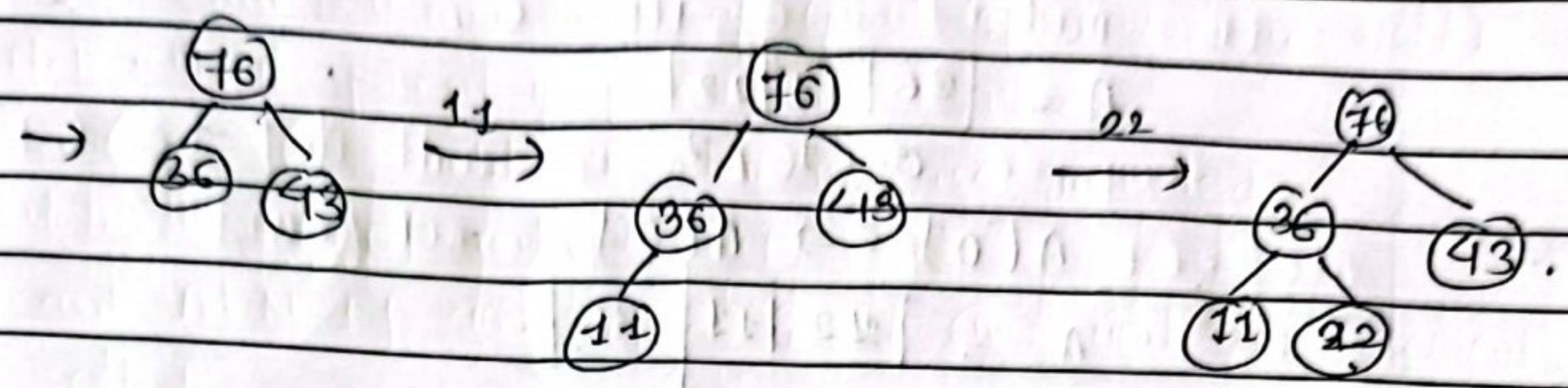
interchanging A[0] with A[5].

A =	36	43	74	11	22	98		99
	0	1	2	3	4	5		

⇓

	36	43	74	11	22		98	99





A = | 76 | 36 | 43 | 11 | 22 |

0 1 2 3 4

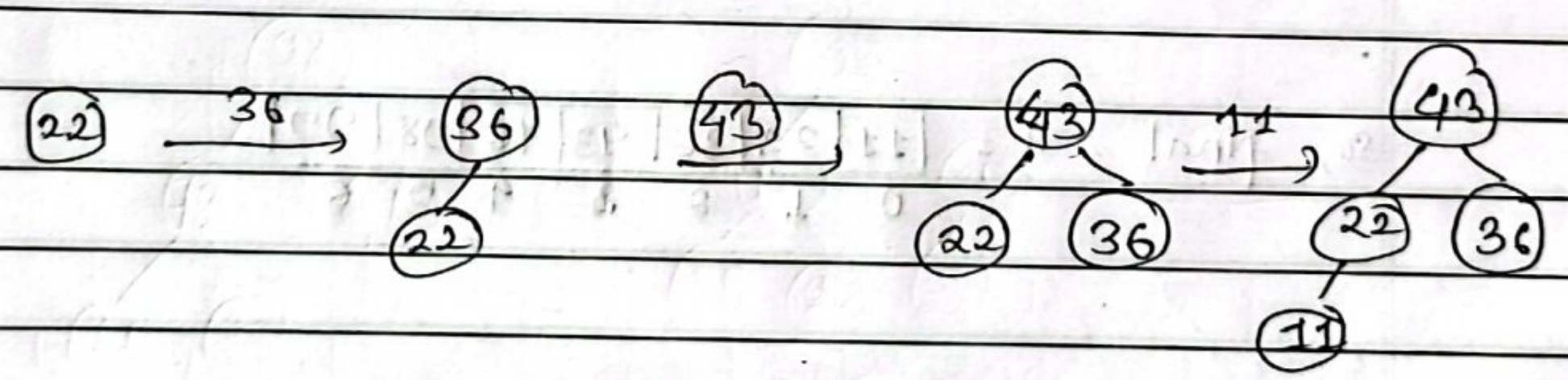
interchange A[0] & A[4].

A = | 22 | 36 | 43 | 11 | 76 |

⇓

| 22 | 36 | 43 | 11 | | 76 | 98 | 99 |

0 1 2 3



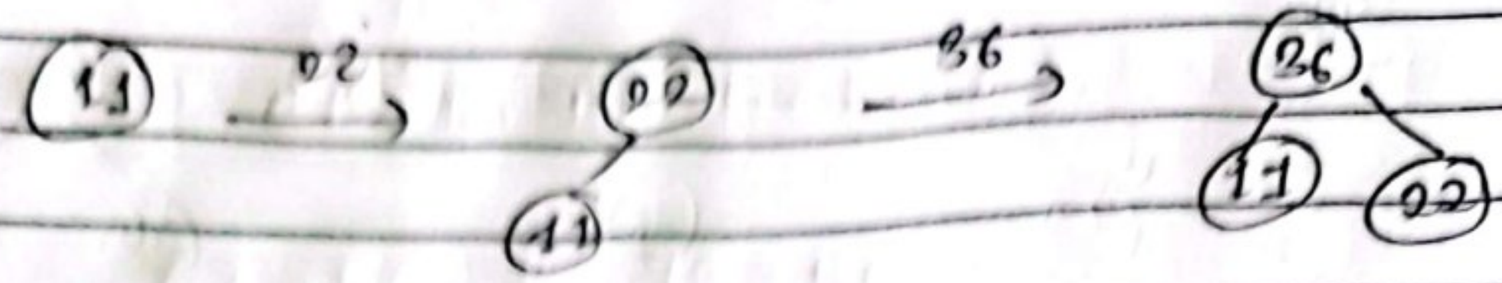
A = | 43 | 22 | 36 | 11 | | 76 | 98 | 99 |

0 1 2 3

A[0] ⇔ A[3]

A = ~~| 11 | 36 |~~

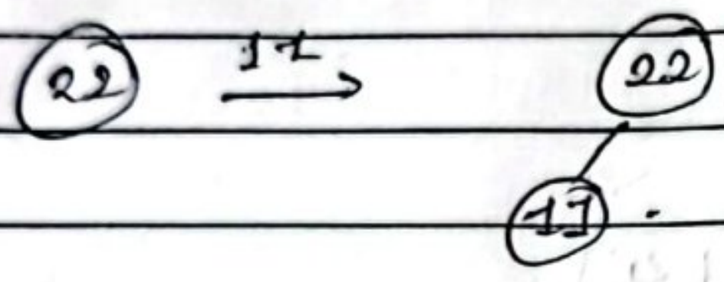
~~22 | 3~~ | 11 | 22 | 36 | 43 |



36	11	22
0	1	2

A[0] ⟷ A[2]

22	11	36



22	11
0	1

A[0] ⟷ A[1]

A = | 11 | 22 | | 36 | 43 | 76 | 98 | 99 |

So, final A =

11	22	36	43	76	98	99
0	1	2	3	4	5	6

Date
Page

Binary Search

→ only in sorted type.

→ compare with mid. $mid = \frac{UB + UB}{2}$, $LB = 0$, $UB = Max - 1$

3 conditions

(i) $Key = A[mid]$

(ii) $Key < A[mid]$.

$UB = Mid - 1$, $LB = LB$

(iii) $key > A[mid]$. again compare

$UB = UB$, $LB = mid + 1$ & key $A[mid]$.

→ till $LB \leq UB$.

Ex

10	17	28	34	45	56	67	82	91	99
0	1	2	3	4	5	6	7	8	9

search 67 & 72

$LB = 0$, $UB = 9$ $\quad Mid = \frac{LB+UB}{2} = 4.5 = 4$

compare 67 & A[4] ie 67 & 46.

$67 > 46$.	$72 > 46$.
so, $UB = 9$, $LB = mid + 1 = 5$	$LB = 5$
$mid = \frac{9+5}{2} = 7$	$mid = 7$.
compare 67 & A[7] ie 67 & 82.	compare with A[7].
$67 < 82$	$72 < 82$
$LB = 5$, $UB = 6$	$LB = 5$, $UB = 6$.
$mid = \frac{6+5}{2} = 5$	$mid = 5$

CS CamScanner
CS CamScanner

& 72.

compare 67 with A[6] i.e 56

67 > 56 & 72 > 56

LB = mid + 1 = 6, UB = 6.

compare 67 & 72 with A[6] i.e 67.

A[6] = 67 72 → not found.

Linear Probing method (Example)

→ For the given data 1, 16, 49, 36, 25, 64, 0, 81, 4 and 9. Show the contents of the hash table having the size 10 using linear probing. Assume hash function be: H(K) = k % table size.

⇒ Solution:-

(key+1) % table size H(1) = 1 % 10 = 1

(key+2) H(16) = 16 % 10 = 6

H(49) = 49 % 10 = 9

(key+n) H(36) = 36 % 10 = 6

H(25) = 25 % 10 = 5

H(64) = 64 % 10 = 4.

H(0) = 0 % 10 = 0.

H(81) = 81 % 10 = 1

H(4) = 4 % 10 = 4

H(9) = 9 % 10 = 9

9	49
8	4
7	36
6	16
5	25
4	64
3	9
2	81
1	1
0	0

CamScanner

Date
Page

Quadratic. $(h+i^2)\%$

$(h-i^2)\%$

~~H(k)~~

if. $(h-i^2) = -ve$

hash value = tablesize - address

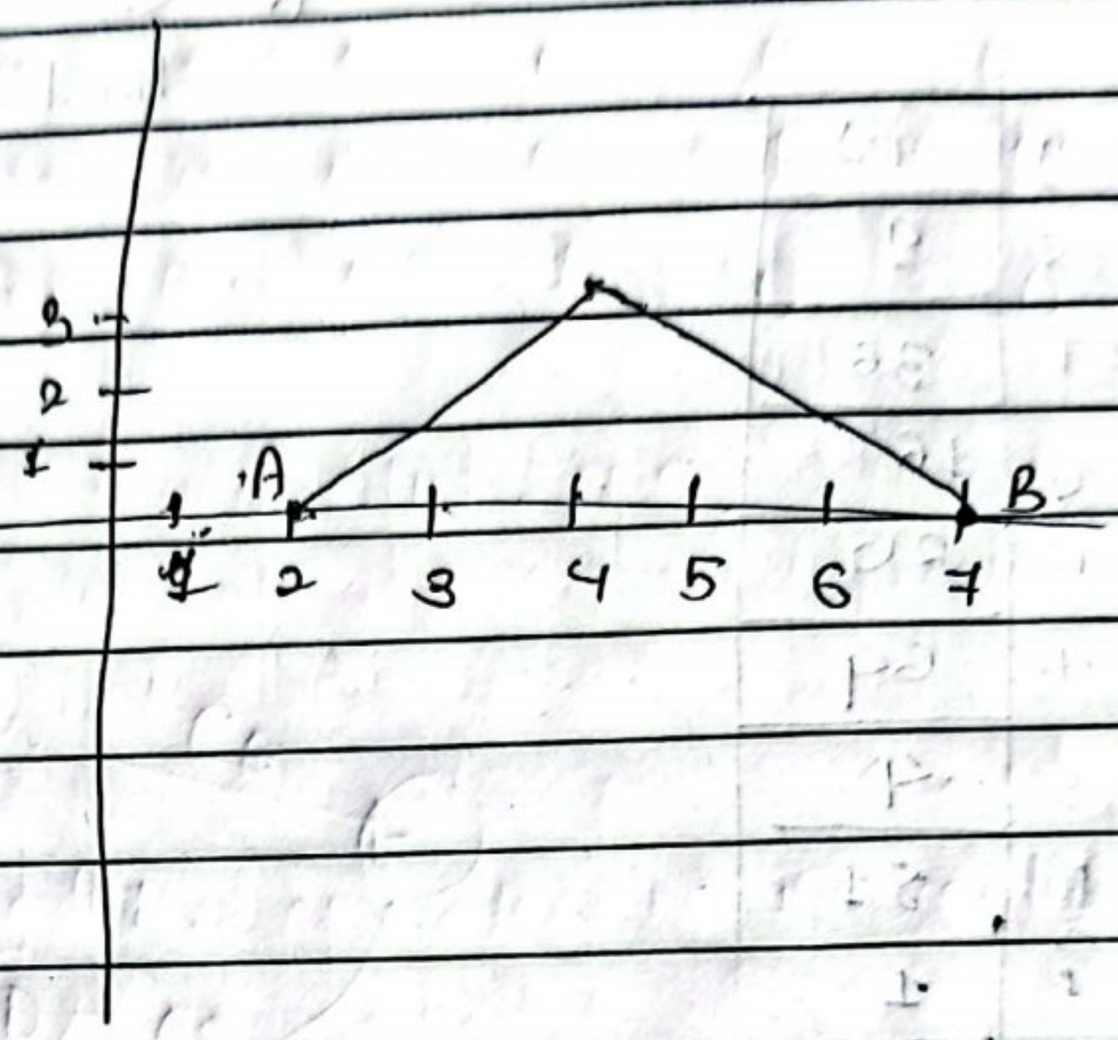
9	49
8	9
7	36
6	16
5	25
4	64
3	4
2	81
1	1
0	0

→ For chaining →

Qn For the following keys 80, 92, 25, 76, 48, 150, 215, 216 and 38. Show the content of the hash table having table size 10 using chaining method for collision resolution. Given hash function as H(k) = k Mod 10.

Graph Theory

Plot a triangle A(2, 0), B(7, 0) & C(4, 3).

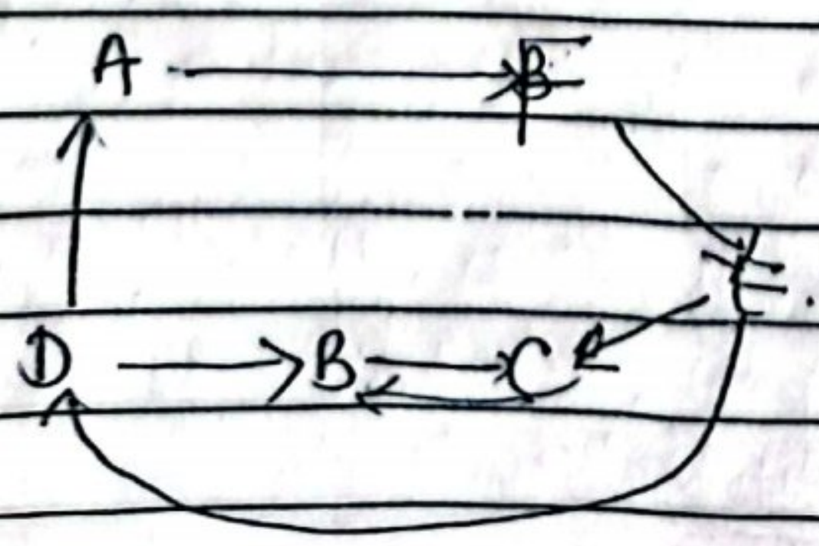


Qn.

Vertex	Adjacency list
A	F
B	C
C	B
D	A, B
E	C, D
F	E

let G be graph represented by this adjacency list.

1. Draw graph.
adjacency matrix
Is G directed?
Is G cyclic.



cyclic, directed.

↓

AFEDA

Date
Page

adjacency matrix

$$A = \begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & 0 & 0 & 0 & 0 & 1 \\ B & 0 & 0 & 1 & 0 & 0 & 0 \\ C & 0 & 1 & 0 & 0 & 0 & 0 \\ D & 1 & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 1 & 1 & 0 & 0 \\ F & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

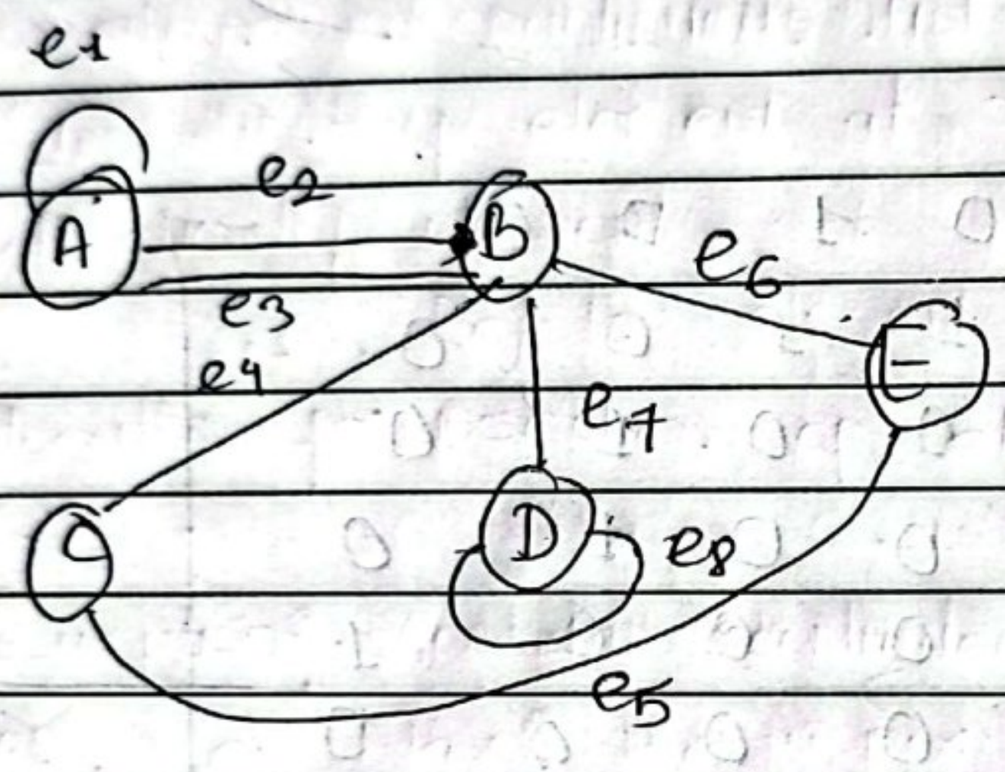
$$A^T = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

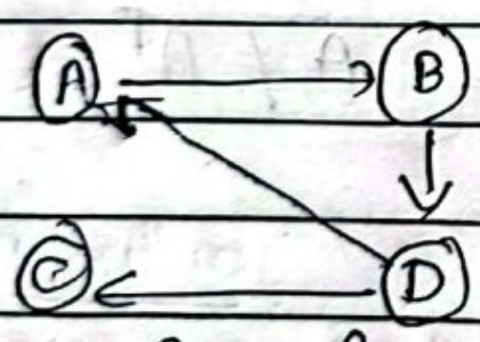
So, Graph is directed since $A \neq A^T$

CamScanner

Draw graph from given incidence matrix.

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
A	1	1	1	0	0	0	0	0
B	0	1	1	1	0	1	1	0
C	0	0	0	1	1	0	0	0
D	0	0	0	0	0	0	1	1
E	0	0	0	0	1	1	0	0





Graph G

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} = W^{[0]}.$$

$W^{[1]}$ making {A} as intermediate

$$W^{[1]} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Prim's algorithm } → Greedy algorithm
Dijikstra algorithm }

$W^{[2]} = \{A, B\}$.

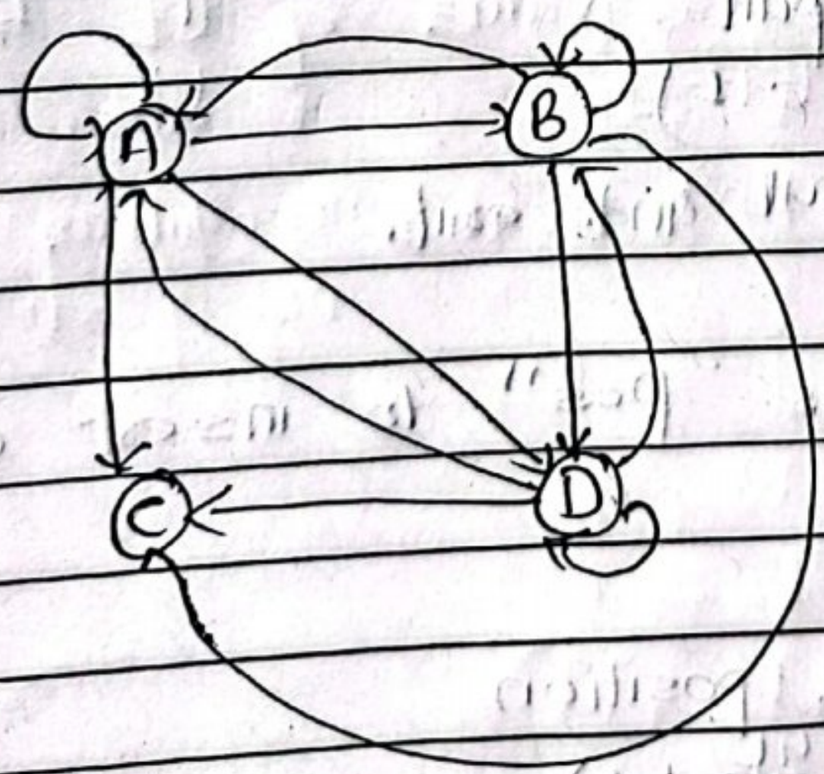
$$W^{[2]} = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 0 & 0 & 1 \\ C & 0 & 0 & 0 & 0 \\ D & 1 & 1 & 1 & 1 \end{array}$$

$W^{[3]} = \{A, B, C\}$

$$W^{[3]} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$W^{[4]} = \{A, B, C, D\}$.

$$W^{[4]} = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 1 & 1 & 1 & 1 \\ B & 1 & 1 & 1 & 1 \\ C & 0 & 0 & 0 & 0 \\ D & 1 & 1 & 1 & 1 \end{array}$$



Array implementation of list

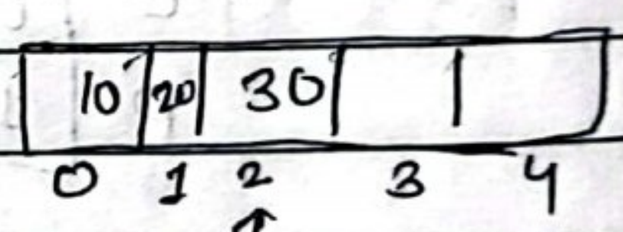
→ Static implementation is used as a collection of same type of element placed in contiguous memory.

→ Element of contagious memory can be accessed individually using index.

→ operations in contiguous list are;
ⓐ Insertion of element at required position.
ⓑ Deletion of element.

ⓐ Algorithm for insertion of element in contiguous list:

Assumption: Consider contiguous list as C_List [Max] of size "Max". The upper value of list is represented by "UL". Initially "UL" is set to -1 i.e empty list.



Algorithm

Step 1: check for list full
if (UL = Max - 1)
Print list full and exit.

Step 2: Read position as "Pos" to insert an element.

Step 3: Check for valid position
if (Pos > ~~pos~~ UL+1)
Print invalid position & exit.

Date
Page

Step 4: Read "data" to insert in list.

Step 5: Travel in list until position to insert element is reached.

```
Set, TEMP = UL
while (Temp >= pos)
{
    c_list [Temp+1] = c_list [Temp]
    Temp = Temp - 1
}
```

Step 6: Insert element in list specified by "pos" value.

```
c_list[POS] = data
```

Step 7: Update UL by 1

i.e UL = UL + 1

Step 8: Stop

CamScanner

Algorithm to delete element from continuous list

Algorithm

10	15	7	98	
0	1	2	3	4

UL = 3

Step 1: Check for list empty

if (UL = -1)

Print list empty & exit

Step 2: ~~Print~~ Read position as "pos" to delete the element.

if (pos > = 0 and pos < = UL)

3.1: Delete element from C_list specified by "pos"

data = C_list[pos]

3.2. Shift all element left until "pos" is reache

Temp = pos

while (Temp < UL)

C_list [Temp] = C_list[Temp + 1]

Temp = Temp + 1

}

3.3. Decrement the value of UL by 1.

UL = UL - 1

else : print invalid position & exit.

Step 4: Stop

CamScanner

Validity of expression

→ An expression generally consists of following delimeters
ⓐ Parenthesis "(" ")" "{" "}" and "[" "]".
ⓑ comments "/*" "*/"

→ An expression is said to be valid if
ⓐ Number of left delimeter equals to no of right delimeter.
ⓑ Each right delimeter is preceed by matching left delimeter.

e.g {(A+B) - (C* D)}] is invalid.
[(A+B) - (C*D)] is valid.

Algorithm

1: Scan the element from left to right until end of expression is reached.

2: If the scanned element is:
2.1 (or { or [push it to stack.

2.2 "/" scan the next character, if the next character is "*". Skip all the character until "*/" is found and print invalid if end of expression is reached before "*/" is ~~found~~ found.

2.3.1: Check if the popped delimeter matches ~~with~~ with each other or not if it doesnot match then display invalid and exit.

Date
Page

Validity of expression

→ An expression generally consists of following delimeters:
ⓐ Parenthesis "(" ")" "{" "}" and "[" "]".
ⓑ comments "/*" "*/".

→ An expression is said to be valid if
ⓐ Number of left delimeter equals to no of right delimeter.
ⓑ Each right delimeter is preceed by matching left delimeter.

e.g. {(A+B) - (C* D)}] is invalid.
[(A+B) - (C*D)] is valid.

Algorithm

1: Scan the element from left to right until end of expression is reached.

2: If the scanned element is:
2.1 (or { or [push it to stack.

2.2 "/" scan the next character, if the next character is "*". Skip all the character until "*/" is found and print invalid if end of expression is reached before "*/" is found.

2.3.1: Check if the popped delimeter matches with each other or not if it doesnot match then display invalid and exit.

CamScanner

2.3. 2. If during pop operation stack is found to be empty display invalid i.e no of right delimeter > No of left delimeter and exit.

2.4: other symbol, ignore them.

3: If the scanned symbol is end of expression, and doesn't get empty print invalid i.e No of left delimeter > no of right delimeter and exit, otherwise print valid expression.

4: Stop.

Qn. [{(A+B) - (C*D)}.

SN	Scanned symbol	Stack	
	[	[	
	{	[, {	
	(	[, {, (	
	A	[, {, (	
	+	[, {, (	
	B	[, {, (	invalid, because the
	)	[, {,	no of left delimeter
	-	[, {	no of rt delimeter
	(	[, {, (	
	C	[, {, (	
	*	[, {, (	
	D	[, {, (	
	)	[, {	
	}	[	